Journal of the Operations Research Society of Japan
Vol. 60, No. 4, October 2017, pp. 439–460

AN EFFICIENT LOCAL SEARCH FOR THE CONSTRAINED SYMMETRIC LATIN SQUARE CONSTRUCTION PROBLEM

Kazuya Haraguchi
Otaru University of Commerce

(Received April 11, 2016; Revised March 29, 2017)

Abstract A *Latin square* is a complete assignment of $[n] = \{1, \dots, n\}$ to an $n \times n$ grid such that, in each row and in each column, each value in $[n]$ appears exactly once. A *symmetric Latin square* (*SLS*) is a Latin square that is symmetric in the matrix sense. In what we call the *constrained SLS construction* (*CSLSC*) problem, we are given a subset F of $[n]^3$ and are asked to construct an SLS so that, whenever $(i,j,k) \in F$, the symbol k is not assigned to the cell (i,j). This paper has two contributions for this problem. One is proposal of an efficient local search algorithm for the maximization version of the problem. The maximization problem asks to fill as many cells with symbols as possible under the constraint on F. In our local search, the neighborhood is defined by *p-swap*, i.e., dropping exactly p symbols and then assigning any number of symbols to empty cells. For $p \in \{1,2\}$, our neighborhood search algorithm finds an improved solution or concludes that no such solution exists in $O(n^{p+1})$ time. The other contribution is to show its practical value for the CSLSC problem. For randomly generated instances, our iterated local search algorithm frequently constructs a larger partial SLS than state-of-the-art solvers such as IBM ILOG CPLEX, LocalSolver and WCSP.

Keywords: Algorithm, combinatorial optimization, Latin square, local search, scheduling

1. Introduction

We address the *constrained symmetric Latin square construction* (*CSLSC*) problem. Let $n \geq 2$ be an integer. We denote $[n] = \{1, \dots, n\}$. Consider assigning given n symbols to an $n \times n$ grid of *cells*. We represent the symbols by n integers of $[n]$. For any $i, j \in [n]$, we denote the cell in the row i and in the column j by (i,j). We represent a partial assignment of $[n]$ to the grid by a subset S of $[n]^3$, where the membership $(i,j,k) \in S$ indicates that the symbol k is assigned to the cell (i,j). We assume $(i,j) \neq (i',j')$ for any $(i,j,k), (i',j',k') \in S$ in order to prevent a duplicate assignment. Accordingly $|S| \leq n^2$ holds. We call S a *partial Latin square* (*LS*) if what we call *the Latin square condition* is satisfied; the condition requires that, in each row and in each column, every symbol should appear at most once. In other words, for any $(i,j,k), (i',j',k') \in S$, at least two of $i \neq i'$, $j \neq j'$ and $k \neq k'$ hold. The S is a *partial symmetric Latin square* (*SLS*) if it is symmetric in the matrix sense, i.e., $(i,j,k) \in S$ implies $(j,i,k) \in S$. In particular, partial LS and SLS are called *complete LS and SLS* respectively if they are complete assignments.

Now we are ready to formulate the CSLSC problem.

Problem CSLSC
Input: An integer $n \geq 2$ and a subset F of $[n]^3$.
Output: Decide whether there exists a complete SLS S such that $S \cap F = \emptyset$. If yes, provide such S.

We call F a *forbidden set* and each entry (i, j, k) in F a *forbidden entry*. It prohibits us from assigning the symbol k to the cell (i, j). The CSLSC problem is NP-hard in general [6] and has strong applications in sports scheduling, as we mention in Section 2. In this paper we mainly consider the maximization version of the problem. We call it the *MaxCSLSC* problem, which is summarized as follows.

Problem MaxCSLSC
Input: An integer $n \geq 2$ and a subset F of $[n]^3$
Output: A partial SLS S of the maximum cardinality such that $S \cap F = \emptyset$.

We have two contributions in this paper. One is an algorithmic result, that is proposal of an efficient local search for the MaxCSLSC problem. The other contribution is a computational result that shows the practical value of the local search for the CSLSC problem.

Local search is a well-known framework of approximation algorithms for hard combinatorial problems [1, 13, 30]. It is regarded as repetition of a *neighborhood search algorithm*; given a solution S, the algorithm outputs an improved solution in the neighborhood if one exists or concludes that no improved solution exists. The neighborhood of S in general is defined as the set of solutions that are obtained by performing "slight" modification on S. For this slight modification, we take up *p-swap*, which is an operation of dropping exactly p elements from S and then adding any number of elements to S. We call the neighborhood defined by p-swap the *p-neighborhood*. For $p \in \{1, 2\}$, we propose a p-neighborhood search algorithm that runs in $O(n^{p+1})$ time. The data structure is based on Andrade et al.'s local search for the maximum independent set problem [3]. We emphasize that, however, our work should not be its simple application.

We develop a metaheuristic algorithm based on the *iterated local search* that exploits the proposed efficient local search. The iterated local search is a metaheuristic framework that attempts to find good solutions by repeating a local search (possibly) many times. For randomly generated instances, our metaheuristic algorithm frequently constructs a larger partial SLS than exact IP and CP solvers from IBM ILOG CPLEX [21] and heuristic solvers LocalSolver [29] and WCSP [32], within the same time limit.

This paper is organized as follows. We describe the background of the research in Section 2. After preparing terminologies and notations in Section 3, we explain the local search algorithm in Section 4, mainly on the p-neighborhood search algorithms. Then we present experimental results in Section 5 and conclude the paper in Section 6.

2. Background

We start the research on local search for LS completion-type problems, motivated by constant-factor approximation algorithms for the *partial LS extension (PLSE)* problem that were studied in [11, 15, 27]. This problem asks for a largest extension of a given partial LS and is summarized as follows.

Problem PLSE
Input: An $n \times n$ partial LS L, represented by a subset of $[n]^3$.
Output: A partial LS S of the maximum cardinality such that $S \supseteq L$.

Hajirasouliha et al. [15] showed that the local search based on p-swap is a $(2/3 - \varepsilon)$-approximation algorithm, using a classical result on local search for the k-set packing problem [20]. In [17], for $p \in \{1, 2, 3\}$, the author proposed a neighborhood search algorithm that runs in $O(n^{p+1})$ time. He also invented a generalization of 2-swap operation, *Trellis-swap*, and proposed a neighborhood search algorithm that runs in $O(n^{3.5})$ time.

The MaxCSLSC problem is different from the PLSE problem in two points; the solution should be symmetric, and the constraint is given by means of the forbidden set F, which is an arbitrary subset of $[n]^3$. As was done for the PLSE problem in [17], we reduce the MaxCSLSC problem to the maximum independent set problem and utilize efficient local search that is developed by Andrade et al. [3].

The problem of scheduling a *single round robin tournament* (*SRRT*), which is a fundamental but indispensable issue in sports scheduling, is among the most significant applications of the CSLSC problem. Suppose that n is even. In SRRT, there are n teams, and each team meets every other team exactly once; so it plays $n - 1$ games in all. The games should be held in $n - 1$ *time slots* one by one. A typical SRRT scheduling problem asks for an SRRT under such constraints as game constraints and home-away table [33], many of which prohibit us from making the match between teams i and j in time slot k for various collections of (i, j, k).

An SRRT is represented by an $n \times n$ complete SLS. For each non-diagonal cell (i, j) (i.e., $i \neq j$), assign the symbol k to (i, j) if teams i and j meet in time slot k, and for each diagonal cell (i, i), assign the symbol n to (i, i). In other words, we fill up the $n \times n$ grid with symbols so that the symbol assigned to (i, j) indicates the time slot when teams i and j play, while the symbol n in diagonal cells represents a dummy slot. Clearly it is a complete SLS. Hence the SRRT scheduling problem is regarded as the CSLSC problem for a certain forbidden set F.

We dare to say that de Werra is among ones who started mathematical studies of sports scheduling problems, back to the 1980s (e.g., [7]). Concerning the constrained SRRT scheduling problem, constraint programming (CP) approach [18] and integer programming (IP) approach [36] were examined in the first half of the 2000s. The problem itself is rather fundamental, and many variants and extensions have been studied so far; e.g., carry-over effect (COE) minimization [14, 35], home-away table (HAT) feasibility [5, 19, 31], double round robin tournament problems such as traveling tournament problem (TTP) [8–10, 22]. There are some nice reviews for this research field [24, 33, 34].

In this paper we consider the MaxCSLSC problem, the maximization version of the CSLSC problem. Since many applications require a complete SLS, it appears vain efforts to find an approximate solution (i.e., a partial SLS) by local search. In fact, in the literature of SRRT scheduling, constructive algorithms have been avoided since they may fall into a locally maximal solution (called a "mature set" in [33]).

Even so, we claim that effective and efficient heuristic algorithms should be practically meaningful due to the following two reasons. First, such an algorithm can serve as an initial solution generator for exact solvers based on IP/CP. When we encounter a large and/or hard instance of combinatorial optimization problems, it must be reasonable to try exact solvers at first as their performance has become so high these days. The computation time could be too long, but may be reduced if a better initial solution is input to the solvers [26]. Second, even though there is no guarantee that a heuristic algorithm delivers a complete SLS, it may do so for some hard instances much faster than modern exact solvers. Therefore, given a hard instance, there is some merit in trying a heuristic algorithm before running an exact solver for a longer time. With these in mind, we develop efficient local search for the MaxCSLSC problem.

3. Preliminaries

Since we consider a partial symmetric assignment, we restrict our attention to the lower triangular part of the $n \times n$ grid. Let $U = \{(i,j,k) \in [n]^3 : i \geq j\}$. We assume a forbidden set F to be a subset of U and consider constructing a solution among $U \setminus F$.

We reduce the MaxCSLSC problem to the *maximum strong independent set* (*MaxSIS*) problem and utilize the local search in [3]. A *hypergraph*, denoted by $H = (V, E)$, consists of a set V of vertices and a collection E of hyperedges, where each hyperedge in E is a subset of V. For two vertices v and w ($v \neq w$), v is a *neighbor* of w, or equivalently, v and w are *adjacent* (to each other) if there exists a hyperedge $e \in E$ such that $v, w \in e$. We denote by $N(v)$ the set of all neighbors of v. A *strong independent set* (*SIS*) in a hypergraph is a subset of vertices that intersects any hyperedge in at most one element [4]; in other words, no two vertices are adjacent to each other. The MaxSIS problem asks for a largest SIS in a given hypergraph.

Given a forbidden set F, we construct a hypergraph $H = (V, E)$ as follows. We set the vertex set to $V = U \setminus F$. Note that each vertex (v_1, v_2, v_3) in V is regarded as an integral point in the 3D space. We define a hyperedge as a maximal subset of vertices such that any two of them cannot belong to an SLS simultaneously. Specifically, the collection E of hyperedges consists of what we call *vertical edges* (*v-edges* for short) and what we call *horizontal edges* (*h-edges* for short). The term "vertical" or "horizontal" indicates the direction in which the hyperedge stretches in the 3D space. Denoting by E^{ver} and E^{hor} the collections of v-edges and h-edges respectively, we define $E = E^{\text{ver}} \cup E^{\text{hor}}$. For any $i, j \in [n]$ with $i \geq j$, we denote by $e^{\text{ver}}(i,j)$ the v-edge that contains all vertices (v_1, v_2, v_3) with $v_1 = i$ and $v_2 = j$, that is,

$$e^{\text{ver}}(i,j) = \{(v_1, v_2, v_3) \in V : (v_1 = i) \wedge (v_2 = j)\}.$$

On the other hand, for any $i, k \in [n]$, we denote by $e^{\text{hor}}(i|k)$ the h-edge that contains all vertices (v_1, v_2, v_3) such that at least one of $v_1 = i$ and $v_2 = i$ holds and $v_3 = k$, that is,

$$e^{\text{hor}}(i|k) = \{(v_1, v_2, v_3) \in V : ((v_1 = i) \vee (v_2 = i)) \wedge (v_3 = k)\}.$$

We illustrate how vertices and hyperedges are distributed in the 3D space in Figure 1. We see that there are 11 vertices, which are indicated by circles. H-edges are indicated by polylines (e.g., a bold real line indicates an h-edge $e^{\text{hor}}(1|v_3)$), while v-edges are not depicted to prevent the figure from being mess. The v-edge $e^{\text{ver}}(2,1)$ contains black vertices (i.e., $(2,1,1)$, $(2,1,3)$ and $(2,1,4)$), while the h-edge $e^{\text{hor}}(3|2)$ contains shadowed vertices (i.e., $(3,1,2)$, $(3,3,2)$ and $(4,3,2)$).

Each vertex (v_1, v_2, v_3) is included in at most three hyperedges, that is, $e^{\text{ver}}(v_1, v_2)$, $e^{\text{hor}}(v_1|v_3)$ and $e^{\text{hor}}(v_2|v_3)$; if $v_1 = v_2$, then the latter two are identical. The sets E^{ver} and E^{hor} are defined as $E^{\text{ver}} = \{e^{\text{ver}}(i,j) : i, j \in [n],\ i \geq j\}$ and $E^{\text{hor}} = \{e^{\text{hor}}(i|k) : i, k \in [n]\}$. Since $|E^{\text{ver}}| \leq n(n+1)/2$ and $|E^{\text{hor}}| \leq n^2$, we have an upper bound on the number of hyperedges; $|E| \leq n(3n+1)/2$. Hence $|V| = O(n^3)$ and $|E| = O(n^2)$.

Obviously a subset S of U represents an SLS that does not intersect with F iff it is an SIS in the hypergraph $H = (V, E)$. Hereafter we concentrate on solving the MaxSIS problem on $H = (V, E)$. Here we show some elementary properties of H which are almost clear from Figure 2.

Proposition 1 *Two h-edges $e^{hor}(i|k)$ and $e^{hor}(i'|k)$ $(i \neq i')$ intersect with at most one vertex.*

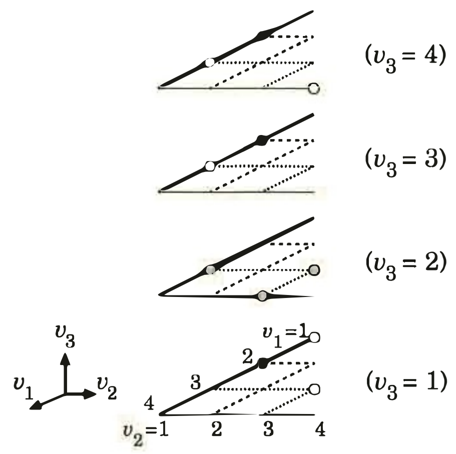
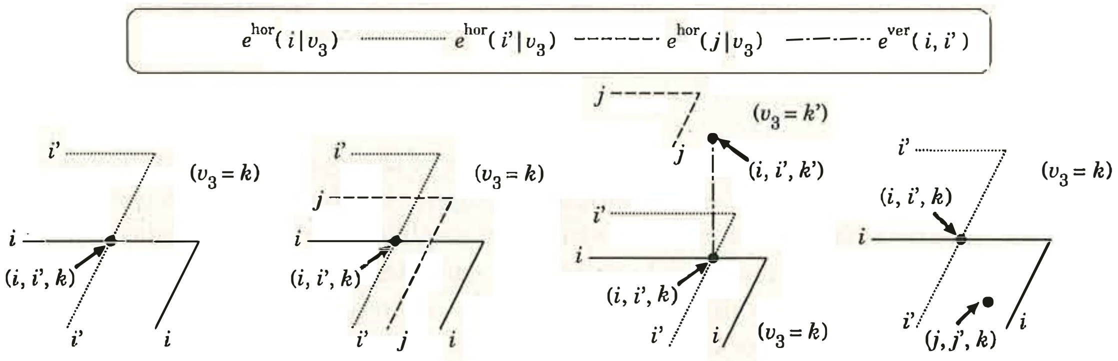


Figure 1: How vertices and hyperedges are distributed in the 3D space ($n = 4$)



(Proposition 1) (Proposition 2) (Proposition 3) (Proposition 4)

Figure 2: Illustration of Propositions 1 to 4

Proof. Assuming $i > i'$, we have $e^{\text{hor}}(i|k) \cap e^{\text{hor}}(i'|k) \subseteq \{(i,i',k)\}$. □

Proposition 2 *A vertex (i,i',k) has at most two neighbors among any h-edge $e^{hor}(j|k)$ such that $j \notin \{i,i'\}$.*

Proof. The vertex (i,i',k) is included in at most two h-edges, $e^{\text{hor}}(i|k)$ and $e^{\text{hor}}(i'|k)$. Each intersects $e^{\text{hor}}(j|k)$ with at most one vertex from Proposition 1. The intersecting vertices are the only possible vertices in $e^{\text{hor}}(j|k)$ that are adjacent to (i,i',k). □

Proposition 3 *A vertex (i,i',k) has at most one neighbor among any h-edge $e^{hor}(j|k')$ such that $k' \neq k$.*

Proof. Among vertices in the 2D plane of $v_3 = k'$, the only possible neighbor of (i,i',k) is (i,i',k'). □

Proposition 4 *A vertex (i,i',k) has at most one neighbor among any v-edge $e^{ver}(j,j')$ such that $(i,i') \neq (j,j')$.*

Proof. The only possible neighbor is (j,j',k). □

We introduce notations and terminologies on local search for the MaxSIS problem. We call an SIS in H simply a *solution*. Given a solution S, we call a vertex in S a *solution vertex* and a vertex out of S a *non-solution vertex*. Let v denote a non-solution vertex. A solution vertex in $N(v)$ is called a *solution neighbor of* v. The number of solution neighbors of v (i.e., $|N(v) \cap S|$) is called the *tightness of* v. Since a hyperedge includes at most one solution vertex and v is included in at most three hyperedges, the tightness is at most three. When the tightness is t, we call v *t-tight*. In particular, a 0-tight vertex is called *free*. The set of t-tight vertices is denoted by $V_t(S)$. The vertex set V is partitioned into $V = S \cup V_0(S) \cup \cdot \cup V_3(S)$. A *$t$-tight neighbor of a solution vertex* x is a t-tight vertex in $N(x)$.

For an integer $p \geq 0$, *p-swap* refers to an operation of dropping a subset D of S ($|D| = p$) from S and adding any number of free vertices (with respect to $S \setminus D$) to $S \setminus D$ so that the resulting set continues to be a solution. The *p-neighborhood of* S is a set of all solutions that are obtained by performing a p-swap on S. A solution S is called *p-maximal* if the p-neighborhood contains no improved solution S' such that $|S'| > |S|$. In particular, we call a 0-maximal solution simply a *maximal* solution.

4. Local Search

In this section, we present the main component algorithm of the local search, a *p-neighborhood search algorithm*. Given a solution S, it computes an improved solution in the p-neighborhood or concludes that no such solution exists. Once a p-neighborhood search algorithm is established, it is immediate to design a local search algorithm that computes a p-maximal solution; starting with an appropriate initial solution, we repeat moving to an improved solution as long as the p-neighborhood search algorithm delivers one.

We present p-neighborhood search algorithms for $p \in \{1, 2\}$ and show that the running times are $O(|S|) = O(n^2)$ and $O(|V|) = O(n^3)$, respectively. The basic idea is borrowed from Andrade et al.'s local search for the maximum independent set problem for an ordinary graph [3]. However, our work is not merely a simple application of their work. They deal with local search based on just $(p, p+1)$-*swap*, that is, the operation of dropping p vertices from the solution and adding $p + 1$ vertices to the solution, whereas the number of added vertices is unbounded in our p-swap. Furthermore, if we applied their methodology as is, then the time bound of the $(p, p+1)$-neighborhood search algorithm would be up to $O(n^{p+3})$.

The strategy is more or less similar to [17] in which the author developed efficient p-neighborhood search algorithms for the PLSE problem. For the MaxCSLSC problem, we need trickier arguments to achieve the time bounds above.

We introduce the data structure in Section 4.1 and present p-neighborhood search algorithms in Section 4.2.

4.1. Data structure

The data structure mainly consists of a 3D array of vertices and an ordering of vertices. The notion of the latter is first introduced by Andrade et al. [3].

The 3D array is used to store the hypergraph $H = (V, E)$. We denote a 3D array by C. For each triple $(v_1, v_2, v_3) \in U$, if $(v_1, v_2, v_3) \in V$, then we let $C[v_1][v_2][v_3]$ have a pointer to the vertex object, and otherwise (i.e., $(v_1, v_2, v_3) \notin V$), we let it have a null value. We can access the vertex in specified coordinates if it exists or decide that no such vertex exists in $O(1)$ time. The 3D array stores the hyperedge set E implicitly; the neighbors of (v_1, v_2, v_3) are among $C[v_1'][v_2][v_3]$'-s, $C[v_1][v_2'][v_3]$'-s and $C[v_1][v_2][v_3']$'-s.

The ordering of vertices enables us to scan vertices of a particular type (e.g., solution vertices, free vertices) in linear time with respect to the number. We denote the ordering by a bijection $\pi : V \to [|V|]$. In π, every solution vertex is ordered ahead of all the non-solution vertices, and the non-solution vertices are ordered in the non-decreasing order of tightness. In each of the five sections (i.e., solution vertices and t-tight vertices for each $t \in \{0,1,2,3\}$), the vertices are ordered arbitrarily. We maintain not only π but also the inverse function π^{-1} so that the i-th vertex, i.e., $\pi^{-1}(i)$, can be accessed in $O(1)$ time.

We maintain the following parameters during the execution of the local search.

Counters of numbers:

- $\#_{\text{sol}}$: the size of the current solution S.
- $\#_t$: the number of t-tight vertices with respect to S ($t \in \{0,1,2,3\}$).
- $\#_1(e)$: the number of 1-tight vertices in a hyperedge $e \in E$.
- $\tau(v)$: the tightness of a non-solution vertex $v \notin S$.

Pointers to vertices: If the corresponding vertex does not exist, then we let the pointer have a null value.

- $\rho_{\text{sol}}(e)$: the pointer to the solution vertex in a hyperedge $e \in E$.
- $\rho_1(e)$: the pointer to the "smallest" 1-tight vertex among a hyperedge $e \in E$, with respect to a fixed total order $\leqslant$ on V. The total order $\leqslant$ can be an arbitrary one if we can search vertices in e in the ascending order in linear time. For example, we can take the lexicographic order on coordinates. Note that $\leqslant$ should be fixed during the local search and is independent of the total order induced by π.

Clearly the size of the data structure is $O(n^3)$. We can construct it in $O(n^3)$ time, as preprocessing of local search. We show time complexities of some elementary operations.

(Maximality check) We can check whether the current solution is maximal or not in $O(1)$ time since it suffices to see whether $\#_0 = 0$ or $\#_0 > 0$.

(Scan of neighbors) We can scan all neighbors of a vertex in $O(n)$ time by using the 3D array C.

(Scan of solution vertices or t-tight vertices) We can scan all solution vertices or all t-tight vertices in linear time with respect to their number by using the vertex ordering and the parameters $\#_{\text{sol}}$ and $\#_t$.

(Addition) We can add a free vertex $v = (v_1, v_2, v_3)$ to the solution in $O(n)$ time. Roughly, we update π so that v falls into the solution vertex section. For each neighbor w, we increase its tightness $\tau(w)$ by one and update π so that w falls into the appropriate section. We also update the other parameters accordingly. The detail is described as follows;

- Let u denote the first vertex in the section of free vertices in the current ordering, that is, $u = \pi^{-1}(\#_{\text{sol}} + 1)$. We update π by exchanging v and u, and let $\#_{\text{sol}} \leftarrow \#_{\text{sol}} + 1$ and $\#_0 \leftarrow \#_0 - 1$. Now v falls into the section of solution vertices.
- For each hyperedge $e \in \{e^{\text{ver}}(v_1, v_2), e^{\text{hor}}(v_1|v_3), e^{\text{hor}}(v_2|v_3)\}$ that includes v, let $\rho_{\text{sol}}(e) \leftarrow v$ since v has become the only solution vertex among e. We search all neighbors $w \in e$ ($w \neq v$) in the ascending order of $\leqslant$. Let w be t-tight. We increase its tightness by one, that is, $\tau(w) \leftarrow \tau(w) + 1$. We update the ordering so that w falls into the section of $(t+1)$-tight vertices, and let $\#_t \leftarrow \#_t - 1$ and $\#_{t+1} \leftarrow \#_{t+1} + 1$. Furthermore, if $\tau(w)$ has become either one or two, then we do the followings;

  Case of $\tau(w) = 1$: We let $\#_1(e) \leftarrow \#_1(e) + 1$. If $\rho_1(e)$ is null or if $w \leqslant \rho_1(e)$, then we let $\rho_1(e) \leftarrow w$.

Case of $\tau(w) = 2$: We let $\#_1(e) \leftarrow \#_1(e) - 1$. If $\rho_1(e) = w$, then we let $\rho_1(e)$ have a null value.

(Drop) We can drop a solution vertex from the solution in $O(n)$ time in the similar way to the addition above.

4.2. p-Neighborhood search algorithms

Let us describe the key idea on how to achieve efficiency in p-neighborhood search. Assume that a $(p-1)$-maximal solution S is given. Hereafter we do not consider a p-swap such that a dropped vertex is immediately returned to the solution since such a swap is degenerated into a $(p-1)$-swap, which is in vain; S is already $(p-1)$-maximal.

Let D denote an arbitrary subset of S. Dropping D from S makes some vertices free. Using our notation, they are vertices of $V_0(S \setminus D)$. This set consists of vertices of D and non-solution vertices (with respect to S) whose solution neighbors are completely incl ed in D. We denote the set of vertices in the latter type by $F(D)$. Hence $V_0(S \setminus D)$ is partitioned into $V_0(S \setminus D) = D \cup F(D)$. We denote by $H(D)$ the subgraph of H induced by $F(D)$. A set of vertices that are added to $S \setminus D$ should be an SIS in $H(D)$. This motivates us to consider the MaxSIS problem on the subgraph $H(D)$.

Let $\alpha(H(D))$ denote the (strong) *independence number* of $H(D)$, that is, the cardinality of a MaxSIS in $H(D)$. The point is that, for every searched subset D, we can determine the independence number $\alpha(H(D))$ in $O(1)$ time and construct a MaxSIS in $H(D)$ in $O(n)$ time.

The p-neighborhood search algorithm runs in the following way; it searches subsets D of the current solution S such that $|D| = p$. If D with $\alpha(H(D)) > p$ is found, then the algorithm constructs a MaxSIS I and halts. By this, we have an improved solution $(S \setminus D) \cup I$. If no such D is found, then the algorithm concludes that S is p-maximal. The running time is linear with respect to the number of searched subsets D since a MaxSIS I is constructed for at most one subset and we can add I to $S \setminus D$ in $O(n)$ time as $|I|$ is bounded by a constant, as we will observe later.

In the remainder of this section, we explain how to compute $\alpha(H(D))$ in $O(1)$ time and to construct a MaxSIS in $H(D)$ in $O(n)$ time. We also mention how to search D efficiently.

4.2.1. Case of $p = 1$

Let S be a maximal solution. When $p = 1$, the dropped subset D is a singleton $D = \{x\}$ of a solution vertex x in S. Let us abbreviate $F(\{x\})$ into $F(x)$ for simplicity. The subset $F(x)$ consists of 1-tight neighbors of x. Let us denote $x = (x_1, x_2, x_3)$. Since x is included in at most three hyperedges, $F(x)$ is partitioned into at most three subsets according to which hyperedge each $u \in F(x)$ belongs to;

$$F(x) = T_1(e^{\text{hor}}(x_1|x_3)) \cup T_1(e^{\text{hor}}(x_2|x_3)) \cup T_1(e^{\text{ver}}(x_1, x_2)),$$

where $T_t(e)$ denotes the set of t-tight vertices in a hyperedge e (i.e., $T_t(e) = V_t(S) \cap e$). For convenience, while we consider the case of $p = 1$, we represent the hyperedges by e_1, e_2 and e_3 as follows;

$$(e_1, e_2, e_3) = \begin{cases} (e^{\text{hor}}(x_1|x_3), e^{\text{hor}}(x_2|x_3), e^{\text{ver}}(x_1, x_2)) & \text{if } x_1 \neq x_2, \\ (e^{\text{hor}}(x_1|x_3), \emptyset, e^{\text{ver}}(x_1, x_2)) & \text{if } x_1 = x_2. \end{cases} \tag{1}$$

When $e = \emptyset$, we assume $T_1(e) = \emptyset$, $\#_1(e) = 0$ and $\rho_1(e) = \text{NULL}$ for convenience.

Consider the subgraph $H(x)$ of H that is induced by $F(x)$. The independence number $\alpha(H(x))$ is at most three since at most one vertex can be picked up from each of the three subsets, $T_1(e_1)$, $T_1(e_2)$ and $T_1(e_3)$. A MaxSIS should include a vertex from $T_1(e_3)$ regardless of which vertices in $T_1(e_1) \cup T_1(e_2)$ are included, due to the following lemma.

Lemma 1 *A MaxSIS in $H(x)$ includes exactly one vertex from $T_1(e_3)$ iff $T_1(e_3)$ is non-empty.*

Proof. As mentioned above, an SIS I includes at most one vertex from a hyperedge, so $|I \cap T_1(e_3)| \leq 1$. The necessity is obvious. For the sufficiency, each vertex in $T_1(e_1) \cup T_1(e_2)$ has at most one neighbor in e_3, and the only possible neighbor is x (Proposition 4). Since $x \notin T_1(e_3)$, no vertex in $T_1(e_1) \cup T_1(e_2)$ is adjacent to one in $T_1(e_3)$. If an SIS I is a subset of $T_1(e_1) \cup T_1(e_2)$, then I is not maximal since we can add any vertex in $T_1(e_3)$ to I. $\square$

Next, we consider how many vertices in $T_1(e_1) \cup T_1(e_2)$ are included in a MaxSIS. The number is at most two. Clearly it is zero iff both $T_1(e_1)$ and $T_1(e_2)$ are empty. Whether it is one or two is completely characterized by the following Lemma 2.

Lemma 2 *Let $m_1 = |T_1(e_1)|$ and $m_2 = |T_1(e_2)|$. A MaxSIS in $H(x)$ includes exactly one vertex from $T_1(e_1) \cup T_1(e_2)$ iff one of the following conditions holds:*
- $\max\{m_1, m_2\} \geq 1$ *and* $\min\{m_1, m_2\} = 0$.
- $m_1 = m_2 = 1$ *and the only vertices in $T_1(e_1)$ and $T_1(e_2)$ are adjacent to each other.*

Proof. The sufficiency is obvious. For the necessity, at least one of $T_1(e_1)$ and $T_1(e_2)$ should be non-empty, that is, m_1 and/or m_2 should be larger than zero. If one is non-zero and the other is zero, then we are done. Suppose that both are larger than zero and that $m_1 \geq m_2 \geq 1$ without loss of generality.

We show $m_1 = m_2 = 1$ by contradiction. Suppose $m_1 > 1$. Let v denote any vertex in $T_1(e_2)$. We claim that, in $T_1(e_1)$, there should be a vertex that is not adjacent to v; From Proposition 2, v has at most two neighbors in e_1. The x is one of them, and thus v has at most one neighbor in $T_1(e_1)$. Since $m_1 - 1 \geq 1$, there is a vertex in $T_1(e_1)$ that is not adjacent to v, say w. Then there exists a MaxSIS I in $H(x)$ such that $\{v, w\} \subseteq I$, which is contradiction.

In this way, we have $m_1 = m_2 = 1$. The only vertices should be adjacent to each other, since otherwise there would be an SIS of size two. $\square$

Corollary 1 *A MaxSIS in $H(x)$ includes exactly two vertices from $T_1(e_1) \cup T_1(e_2)$ iff one of the following conditions holds:*
- $\max\{m_1, m_2\} \geq 2$ *and* $\min\{m_1, m_2\} \geq 1$.
- $m_1 = m_2 = 1$ *and the only vertices in $T_1(e_1)$ and $T_1(e_2)$ are not adjacent to each other.*

Lemma 3 *For $x \in S$, we can determine the independence number $\alpha(H(x))$ in $O(1)$ time and construct a MaxSIS in $H(x)$ in $O(n)$ time.*

Proof. We present a constant-time algorithm to determine $\alpha(H(x))$ in Algorithm 1. Recall that the cardinality $|T_1(e)|$ is maintained by the counter $\#_1(e)$. The $\rho_1(e)$ is the pointer to the smallest 1-tight vertex in e in the sense of $\leqslant$. In particular, if $\#_1(e) = 1$, then $\rho_1(e)$ should point to the only 1-tight vertex in e.

We can construct a MaxSIS in $H(x)$ in $O(n)$ time; Starting with $I = \emptyset$, we add vertices to I according to how Algorithm 1 flows. Specifically, when we pass Lines 3, 7, 9 and 11, we decide the vertex to be added in the following way.

Line 3: We add *any* 1-tight vertex in $T_1(e_3)$ to I.

Algorithm 1 A constant-time algorithm to determine $\alpha(H(x))$

```
1: $\alpha \leftarrow 0$
2: **if** $\#_1(e_3) > 0$ **then**
3:     $\alpha \leftarrow \alpha + 1$                                        ▷ Lemma 1
4: **end if**
5: **if** $\max\{\#_1(e_1), \#_1(e_2)\} \geq 1$ **then**
6:     **if** $\min\{\#_1(e_1), \#_1(e_2)\} = 0$ **then**
7:         $\alpha \leftarrow \alpha + 1$                      ▷ 1st condition in Lemma 2
8:     **else if** $\#_1(e_1) = \#_1(e_2) = 1$ and $\rho_1(e_1)$ and $\rho_1(e_2)$ are adjacent **then**
9:         $\alpha \leftarrow \alpha + 1$                      ▷ 2nd condition in Lemma 2
10:    **else**
11:        $\alpha \leftarrow \alpha + 2$
12:    **end if**
13: **end if**
14: **return** $\alpha$
```

Line 7: We add *any* 1-tight vertex in $e \in \{e_1, e_2\}$ to I such that $T_1(e)$ is non-empty (i.e., $\#_1(e) > 0$).

Line 9: We add either $\rho_1(e_1)$ or $\rho_1(e_2)$ to I.

Line 11: The construction is analogous to the proof of Lemma 2. Let us assume $\#_1(e_1) \geq \#_1(e_2) \geq 1$ without loss of generality. First, we pick up *any* 1-tight vertex v in $T_1(e_2)$. In $T_1(e_1)$, there is at least one 1-tight vertex, say w, that is not adjacent to v. We add $\{v, w\}$ to I.

The constructed I is a MaxSIS. The construction can be done in $O(n)$ time since a hyperedge e contains at most n vertices and thus it takes $O(n)$ time to choose a vertex to be added from e. □

Theorem 1 *Given a maximal solution S, we can find an improved solution in the 1-neighborhood or conclude that S is 1-maximal in $O(n^2)$ time.*

Proof. We see $|S| \leq n(n+1)/2$. We can scan all solution vertices in $O(n^2)$ time. For each $x \in S$, the independence number $\alpha(H(x))$ of the subgraph $H(x)$ is decided in $O(1)$ time, and if $\alpha(H(x)) \geq 2$, then we construct a MaxSIS in $O(n)$ time from Lemma 3, by which we have an improved solution. To update the solution S, it takes $O(n)$ time to drop x from S, and $O(n)$ time to add the vertices in the MaxSIS to $S \setminus \{x\}$ since the size is at most three. The overall time complexity is $O(n^2)$. □

Before going to the case of $p = 2$, we give a necessary condition of 1-maximality.

Theorem 2 *Let S be a 1 maximal solution. For any solution vertex (x_1, x_2, x_3) in S, let e_1, e_2 and e_3 be the hyperedges defined by Equation* (1), $m_1 = |T_1(e_1)|$, $m_2 = |T_1(e_2)|$ and $m_3 = |T_1(e_3)|$. Then we have the following:

- Either $m_1 + m_2 = 0$ or $m_3 = 0$ holds.
- If $m_1 + m_2 > 0$, then either $\max\{m_1, m_2\} \leq 1$ or $\min\{m_1, m_2\} = 0$ holds. Furthermore, if $m_1 = m_2 = 1$, then the only vertices in $T_1(e_1)$ and $T_1(e_2)$ are adjacent to each other.

Proof. If they do not hold, then we would have an improved solution by 1-swap; the first condition is due to Lemma 1 and the second condition is from Corollary 1. □

4.2.2. Case of $p = 2$

Let S be a 1-maximal solution. There are $\binom{|S|}{2} = O(n^4)$ pairs of solution vertices that are candidates to be dropped. Nevertheless, we have only to search certain $O(n^3)$ pairs among them due to the following lemma, which provides a necessary condition on the existence of an improved solution.

Lemma 4 (Implications of Lemmas 1 to 4 in [3]) *Let x, y, u, v, w be vertices such that $x, y \in S$ and $u, v, w \notin S$. If $(S \setminus \{x, y\}) \cup \{u, v, w\}$ is a solution, then we have the followings:*

(1) *u, v and w are not adjacent to one another;*

(2) *one in $\{u, v, w\}$ is 2-tight and adjacent to both x and y,*

(3) *another in $\{u, v, w\}$ is adjacent to x, maybe to y, and to no other vertex in S;*

(4) *the other in $\{u, v, w\}$ is adjacent to y, maybe to x, and to no other vertex in S.*

By (2), it suffices to search such pairs of solution vertices that have the same 2-tight neighbor in common. The number of the pairs is $O(n^3)$ since the number of 2-tight vertices is $O(n^3)$. Furthermore, we can scan all the pairs in linear time with respect to the number of 2-tight vertices, using the vertex ordering of the data structure.

As in Lemma 4, we denote solution vertices by x and y, and non-solution vertices by u, v and w. We assume that u is 2-tight and is adjacent to both x and y. The 2-tight vertex u should be included in an improved solution. The tightness of v and w is either one or two, and neither of them should belong to $\{u\} \cup N(u)$; they should be among the vertex set F', which is defined as $F' = F(x, y) \setminus (\{u\} \cup N(u))$. We illustrate in Figure 3 where the vertices in F' are distributed in the 3D space. (The definitions of the two vertices u' and v' will be given later.) The vertices in F' are among hyperedges that are represented by bold real polylines. We denote by e_x and e'_x (resp., e_y and e'_y) the two hyperedges that include x (resp., y) and that are represented by bold real polylines. We give their formal defin ns as follows.

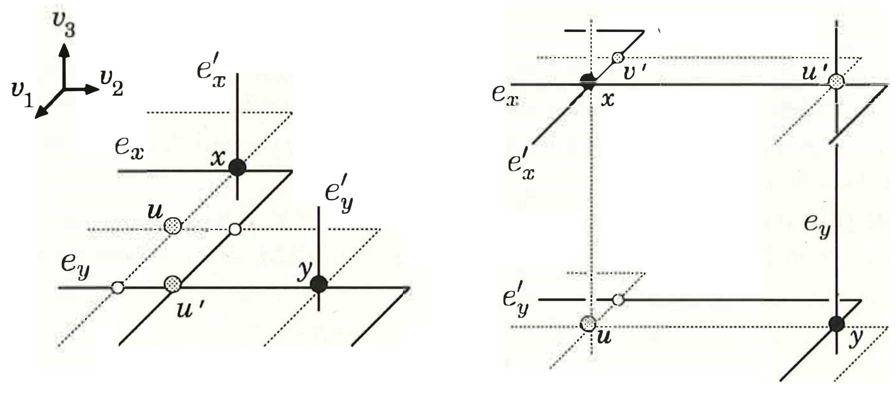
(Case of $x_3 = y_3$)

$$e_x = \begin{cases} \text{the h-edge } e \text{ such that } x \in e \text{ and } u \notin e & \text{if } x_1 \neq x_2, \\ \emptyset & \text{if } x_1 = x_2, \end{cases}$$
$$e'_x = e^{\text{ver}}(x_1, x_2),$$
$$e_y = \begin{cases} \text{the h-edge } e \text{ such that } y \in e \text{ and } u \notin e & \text{if } y_1 \neq y_2, \\ \emptyset & \text{if } y_1 = y_2, \end{cases}$$
$$e'_y = e^{\text{ver}}(y_1, y_2).$$

(Case of $x_3 \neq y_3$) Without loss of generality, we assume that x and u are connected by a v-edge and that y and u are connected by an h-edge. We denote the latter h-edge by e'. The e_x is defined as the h-edge that includes x and that is parallel (not skew) to e'.

$$e'_x = \begin{cases} \text{the h-edge } e \text{ such that } x \in e \text{ and } e \neq e_x & \text{if } x_1 \neq x_2, \\ \emptyset & \text{if } x_1 = x_2, \end{cases}$$
$$e_y = e^{\text{ver}}(y_1, y_2),$$
$$e'_y = \begin{cases} \text{the h-edge } e \text{ such that } y \in e \text{ and } u \notin e & \text{if } y_1 \neq y_2, \\ \emptyset & \text{if } y_1 = y_2. \end{cases}$$

One can easily verify that u is included in none of $\{e_x, e'_x, e_y, e'_y\}$. It is possible that one hyperedge in $\{e_x, e'_x\}$ and one in $\{e_y, e'_y\}$ intersect. When in this case, it is e_x and e_y that intersect.



$(x_3 = y_3)$ $(x_3 \neq y_3)$

Figure 3: How the 2-tight vertices u and u' and two solution vertices x and y are distributed in the 3D space

We denote by H' the subgraph induced by F'. The problem of finding an improved solution is reduced to the MaxSIS problem on H'. If there is an SIS I with $|I| \geq 2$, then we have an improved solution $(S \setminus \{x, y\}) \cup (I \cup \{u\})$.

First, we observe that F' includes at most one 2-tight vertex; hence, F' consists of 1-tight vertices and at most one 2-tight vertex.

Lemma 5 *There is at most one 2-tight vertex among F'.*

Proof. Any 2-tight vertex in F' should be at the intersection point of a hyperedge from $\{e_x, e'_x\}$ and a hyperedge from $\{e_y, e'_y\}$. The only pair of hyperedges that can intersect is $\{e_x, e_y\}$. □

Lemma 6 *Whether F' includes a 2-tight vertex or not can be decided in $O(1)$ time.*

Proof. The coordinates of the unique candidate u' are decided in $O(1)$ time. We can conclude $u' \in F'$ if $u' \in V$ and $\tau(u') = 2$. □

Suppose that F' includes a 2-tight vertex u'. The vertex set F' is partitioned into subsets as follows; $F' = T_1(e_x) \cup T_1(e'_x) \cup T_1(e_y) \cup T_1(e'_y) \cup \{u'\}$. When we count the independence number $\alpha(H')$, we take u' into account only when it is included in all MaxSISs. Whether we are in the case or not is characterized by the following Lemmas 7 and 8.

Lemma 7 *Suppose $x_3 = y_3$ and that there is a 2-tight vertex u' in the subgraph H'. The u' is included in all MaxSISs in H' iff $T_1(e_x) = T_1(e_y) = \emptyset$.*

Proof. For the necessity, if one of $T_1(e_x)$ and $T_1(e_y)$ is non-empty and includes a 1-tight vertex v, then we would have a MaxSIS that does not include u'; Let I denote a MaxSIS that includes u'. We see that $(I \setminus \{u'\}) \cup \{v\}$ is also a MaxSIS. The sufficiency is obvious. □

Lemma 8 *Suppose $x_3 \neq y_3$ and that there is a 2-tight vertex u' in the subgraph H'. The u' is included in all MaxSISs in H' iff $T_1(e_y) = \emptyset$ and either (i) or (ii) holds:*

(i) *$T_1(e_x) = \emptyset$, $T_1(e'_x) \neq \emptyset$, and $T_1(e'_x)$ includes a vertex that is not adjacent to u'.*

(ii) *$|T_1(e_x)| = |T_1(e'_x)| = 0$ or 1.*

Proof. Let us denote $m_1 = |T_1(e_x)|$ and $m_2 = |T_1(e'_x)|$. From the necessary condition of 1-maximal solution in Theorem 2, we have $\max\{m_1, m_2\} \leq 1$ or $\min\{m_1, m_2\} = 0$. We are in one of the four cases with respect to m_1 and m_2: **(a)** $m_1 \geq 1$ and $m_2 = 0$, **(b)** $m_1 = 0$ and $m_2 \geq 1$, **(c)** $m_1 = m_2 = 1$, and **(d)** $m_1 = m_2 = 0$.

For the necessity, $T_1(e_y)$ should be empty since, if not so, we could construct a MaxSIS that does not include u' by exchanging u' and any vertex in $T_1(e_y)$. Among (a) to (d) above, the case (a) is not possible due to the same reason. Thus we are in (b), (c) or (d). Suppose that we are in (b). If $m_2 = 1$, then the only vertex in $T_1(e'_x)$ should not be adjacent to u' (as v' in the right of Figure 3) since otherwise we could construct a MaxSIS $(I \setminus \{u'\}) \cup \{v'\}$. If $m_2 > 1$, then there is a vertex in $T_1(e'_x)$ that is not adjacent to u' since at most two vertices in e'_x are adjacent to u' and one of them is x from Proposition 2. This observation is summarized as (i). The cases (c) and (d) are summarized as (ii).

For the sufficiency, suppose $T_1(e_y) = \emptyset$. If we are in (i), or if we are in (ii) and $m_1 = m_2 = 0$, then it is obvious that u' is included in all MaxSISs. Suppose that we are in (ii) and $m_1 = m_2 = 1$. Let v (resp., w) be the only vertex in $T_1(e_x)$ (resp., $T_1(e'_x)$). From Proposition 2, the vertex w has at most two neighbors among e_x. One of them is x, and from Theorem 2, the other should be v. The vertices w and u' are not adjacent to each other. We see that all MaxSISs include exactly two vertices from $T_1(e_x) \cup T_1(e'_x) \cup T_1(e_y) \cup \{u'\}$ and that they should be u' and w. □

Lemma 9 *For a 2-tight vertex u, let x and y be its solution neighbors. We can determine the independence number $\alpha(H')$ in $O(1)$ time and construct a MaxSIS in H' in $O(n)$ time.*

Proof. We summarize a constant-time algorithm to determine $\alpha(H')$ in Algorithm 2. In Lines 3 and 21, it checks whether the conditions of Lemmas 7 and 8 are satisfied respectively. The check be done in $O(1)$ time. When the conditions are satisfied, we do not take the hyperedges e_x and e_y into account any longer since no vertex there belongs to a MaxSIS.

Let I denote a MaxSIS in H' The set $I \cup \{u\}$ is a MaxSIS in $H(x, y)$. We have $|I \cup \{u\}| = \alpha(H') + 1 \leq 4$. As we did for the case of $p = 1$, we can construct I in $O(n)$ time according to how Algorithm 2 flows. We omit the detail as it would be too lengthy and is not so difficult. □

Theorem 3 *Given a 1-maximal solution S, we can find an improved solution in the 2-neighborhood or conclude that S is 2-maximal in $O(n^3)$ time.*

Proof. Since there are at most n^3 2-tight vertices, we can scan all 2-tight vertices in $O(n^3)$ time by using the vertex ordering. For each 2-tight vertex u, its solution neighbors x and y can be decided in $O(1)$ time by using pointers $\rho_{\text{sol}}(e^{\text{hor}}(u_1|u_3))$, $\rho_{\text{sol}}(e^{\text{hor}}(u_2|u_3))$ and $\rho_{\text{sol}}(e^{\text{ver}}(u_1, u_2))$. The independence number $\alpha(H')$ is decided in $O(1)$ time by Lemma 9. We have $\alpha(H(x, y)) = \alpha(H') + 1$. If $\alpha(H(x, y)) \geq 3$, then we construct a MaxSIS. This can be done in $O(n)$ time, also due to Lemma 9. It takes $O(n)$ time to drop x and y from S, and it takes $O(n)$ time to add the vertices in the MaxSIS to $S \setminus \{x, y\}$ since the size of the MaxSIS is at most four. The overall time complexity is $O(n^3)$. □

Before closing this section, we describe how to find a 2-maximal solution. Let S be an arbitrary initial solution. If S is not maximal, then we construct a maximal solution by adding free vertices to S repeatedly until no free vertex is left. If S is not 1-maximal, then we construct a 1-maximal solution by conducting the 1-neighborhood search until S becomes 1-maximal. We then construct a 2-maximal solution from S by performing the 2-neighborhood search. During the search, if an improved solution is found, then we go back

Algorithm 2 A constant-time algorithm to determine $\alpha(H')$

```
1: $\alpha \leftarrow 0$
2: if $x_3 = y_3$ then
3:     if the condition of Lemma 7 is satisfied then
4:         $\alpha \leftarrow \alpha + 1$
5:     else if $\max\{\#_1(e_x), \#_1(e_y)\} \geq 1$ then
6:         if $\min\{\#_1(e_x), \#_1(e_y)\} = 0$ then
7:             $\alpha \leftarrow \alpha + 1$
8:         else if $\#_1(e_x) = \#_1(e_y) = 1$ and $\rho_1(e_x)$ and $\rho_1(e_y)$ are adjacent then
9:             $\alpha \leftarrow \alpha + 1$
10:        else
11:            $\alpha \leftarrow \alpha + 2$
12:        end if
13:    end if
14:    if $\#_1(e'_x) \geq 1$ then
15:        $\alpha \leftarrow \alpha + 1$
16:    end if
17:    if $\#_1(e'_y) \geq 1$ then
18:        $\alpha \leftarrow \alpha + 1$
19:    end if
20: else
21:    if the condition of Lemma 8 is satisfied then
22:        $\alpha \leftarrow \alpha + 1$
23:        if $\#_1(e'_x) \geq 1$ then
24:            $\alpha \leftarrow \alpha + 1$
25:        end if
26:    else
27:        if $\max\{\#_1(e_x), \#_1(e'_x)\} \geq 1$ then
28:            $\alpha \leftarrow \alpha + 1$    ▷ At most one vertex in $T_1(e_x) \cup T_1(e'_x)$ belongs to a MaxSIS
   from Theorem 2
29:        end if
30:        if $\#_1(e_y) \geq 1$ then
31:            $\alpha \leftarrow \alpha + 1$
32:        end if
33:    end if
34:    if $\rho_1(e'_y)$ is not null and ($\#_1(e'_y) \geq 2$ or $u$ and $\rho_1(e'_y)$ are not adjacent) then
35:        $\alpha \leftarrow \alpha + 1$    ▷ In the right of Figure 3, any 1-tight vertex except the white one
   belongs to $H'$
36:    end if
37: end if
38: return $\alpha$
```

to the 1-neighborhood search since the improved solution may not be 1-maximal. Otherwise, the current solution is 2-maximal.

5. Computational Studies

In this section, we demonstrate how practically meaningful the proposed local search is. For this, we develop a metaheuristic algorithm based on *iterated local search* (*ILS*) [13]. This is just a heuristic algorithm for the MaxCSLSC problem, but constructs a larger partial SLS or even a complete SLS (i.e., an optimal solution) for random instances faster and more frequently than exact IP and CP solvers from IBM ILOG CPLEX [21] and two general heuristic solvers, LocalSolver [29] and WCSP [32].

We describe the ILS algorithm in Section 5.1 and experimental settings in Section 5.2. We then show computational results in Section 5.3.

5.1. Iterated local search

The ILS algorithm iterates our local search until a certain termination condition is satisfied. It is sketched as follows.

1. Generate a maximal solution S_0. Let $S^* \leftarrow S_0$.
2. Compute a 2-maximal solution S by local search, using S_0 as the initial solution.
3. If $|S| \geq |S^*|$, then let $S^* \leftarrow S$.
4. If the termination condition is satisfied, then output S^* and halt.
5. Generate a maximal solution S_0 by "kicking" S^*. Go to **2**.

We give some remark to the algorithm. In **1**, we generate S_0 by a constructive algorithm named G5 in [2], which is a "look-ahead" minimum-degree greedy algorithm for the maximum independent set problem for an ordinary graph. Regarding each hyperedge as a clique, one can easily convert the hypergraph into an ordinary graph. We confirmed in [16] that G5 is among the best constructive algorithms for the PLSE problem, which is a version of the MaxCSLSC problem that does not require symmetry of Latin square. The S^* denotes the incumbent solution. In **4**, we terminate the algorithm if the computation time exceeds 10 seconds; we observe that, in our preliminary experiments, most of the improvement is achieved in 10 seconds.

For "kicking" in **5**, we employ the mechanism that the author used for the PLSE problem in [17]. Let us review it briefly. Copying S^* to S_0 at first, we forcibly add k non-solution vertices into S_0, where the natural number k is chosen with probability $1/2^k$. Specifically, we repeat the following steps k times; we pick up a non-solution vertex u, drop its solution neighbors from S_0, and add u into the solution. After the addition, if there are free vertices, then one is chosen at random and added into S_0 repeatedly until S_0 becomes maximal.

For $i \in \{1, .\ \ , k\}$, the i-th vertex to be added is chosen randomly from all the non-solution vertices, except the case of $i = 1$. The first vertex is chosen from P, which is defined as the set of non-solution vertices such that there is a solution neighbor that has at least one 1-tight neighbor;

$$P = \{u \in V \setminus S_0 : \exists x \in N(u) \cap S_0,\ \exists e \in E,\ x \in e,\ T_1(e) \neq \emptyset\}.$$

Note that P can be empty.

When $P \neq \emptyset$, we pick up the first vertex from P as we would like to avoid trivial cycling. Suppose adding a non-solution vertex u that is not in P. Before the addition, all solution neighbors of u should be dropped, that is, the nodes in $N(u) \cap S_0$. As they do not have 1-tight neighbors, it is possible that u is the only vertex that becomes free. In such a case, only u is added into the solution. A solution generated in this way faces higher risk of cycling; we may have the solution S_0 again by a subsequent 1-swap such that u is dropped and the nodes in $N(u) \cap S_0$ are added.

Based on this observation, when $P \neq \emptyset$, we pick up the first vertex to be inserted from P. Furthermore, aiming at diversifying the search, we choose the one that has been outside the solution for the longest time. This strategy is called *soft-tabu* and employed in the ILS algorithm for the maximum independent set problem [3]. In the case of $P = \emptyset$, we use the set of all non-solution vertices instead of P. We confirm that this strategy works effectively in our preliminary experiments.

5.2. Experimental settings

We introduce two types of benchmark problems, which we call RAND and SLSWH. Recall that a MaxCSLSC instance is given by a forbidden set F. Since $V = U \setminus F$, deciding F is equivalent to deciding the vertex set V.

RAND: An instance is given by a random vertex set V. We generate an instance by choosing V randomly. Specifically, we choose $\lfloor r|U| \rfloor$ vertices randomly from U, where r is a parameter between 0 and 1.

SLSWH: It is an abbreviation of "symmetric Latin square with holes". An SLSWH instance is given by a partial SLS L that is generated by removing symbols from a complete SLS L^* that is arbitrarily taken. The problem asks to construct a complete SLS by filling all "holes" (i.e., empty cells) with symbols.

To generate L, we construct L^* by the *polygon method* [25] and then shuffle rows (along with columns) and symbols. We then remove symbols from randomly chosen cells so that there remain $\lfloor (1-\gamma)n^2 \rfloor$ symbols, where γ is a parameter between 0 and 1. Once L is given, the forbidden set F is automatically determined by:

$$F = \{(i,j,k) \in U : \exists (i',j',k') \in L,\ |\{i,j\} \cap \{i',j'\}| + |\{k\} \cap \{k'\}| \geq 2\}.$$

Since $V = U \setminus F$, it is expected that, the larger γ is, the larger the vertex set V is.

An optimal solution of a RAND instance is not necessarily a complete SLS, whereas that of an SLSWH instance is always a complete SLS. For each grid length $n \in \{30, 40, 50\}$, we generate instances by changing the parameters r (for RAND) and γ (for SLSWH). For a fixed n, intuitively, an instance that has a large (resp., a small) portion of U as the vertex set V should be *under-constrained* (resp., *over-constrained*) in the sense that the forbidden set F is small (resp., large). To grab this intuition, let us observe extreme two cases: when V is the largest (i.e., $V = U$ and $F = \emptyset$), any complete SLS is an optimal solution, and when V is the smallest (i.e., $V = \emptyset$ and $F = U$), no feasible solution exists. For RAND, an instance generated by a large r must have a complete SLS as its optimal solution, whereas one generated by a small r may not do so. As mentioned above, an optimal solution of an SLSWH instance is a complete SLS, but hardness for finding it may change along with the parameter γ. In other words, we may observe phase transition of hardness, as is observed in other LS completion-type problems; e.g., the PLSE problem [12], the Sudoku completion problem [28]. An instance generated by an intermediate γ is expected to be harder than one generated by a large γ (i.e., under-constrained) or one generated by a small γ (i.e., over-constrained)

We compare the performance of the ILS algorithm with two exact solvers and two heuristic solvers. For the exact solvers, we employ IP and CP solvers from IBM ILOG CPLEX (ver. 12.6) [21]. We denote the IP and CP solvers by CPX-IP and CPX-CP respectively. We employ straightforward IP and CP formulations used in [12], except that, in CP, we minimize the number of empty cells. In order to admit the CP solver to assign no symbol to a cell (i,j), we set its domain to $[n] \cup \{\phi_{ij}\}$, where ϕ_{ij} is a peculiar value to (i,j) and represents that no symbol is assigned. For the heuristic solvers, we employ LocalSolver

Table 1: Results on RAND instances: how the solvers improve the initial solution S_0

n	$r = \|V\|/\|U\|$	$\|S_0\|$	ILS	LSSOL	WCSP	CPX-IP	CPX-CP
30	0.8	450.5	**∗14.5**	13.3	14.4	0.7	**∗14.5**
	0.7	448.9	**∗16.1**	14.5	14.1	1.4	16.1
	0.6	446.0	**∗19.0**	16.5	14.0	2.3	15.9
	0.5	443.4	**∗21.6**	17.9	13.2	2.1	13.0
	0.4	438.7	**26.0**	20.5	12.5	3.0	11.0
	0.3	429.6	**31.7**	25.5	11.9	8.7	13.1
	0.2	410.4	**38.6**	32.6	13.2	21.9	19.3
40	0.8	797.4	**∗22.6**	17.7	21.2	0.0	17.3
	0.7	795.9	**∗24.1**	18.3	19.8	0.0	12.8
	0.6	792.7	**∗27.3**	20.3	19.6	0.0	8.1
	0.5	787.7	**32.3**	23.5	18.8	0.0	5.4
	0.4	780.8	**38.5**	28.1	15.0	0.0	4.3
	0.3	770.5	**45.1**	33.4	16.3	0.0	5.9
	0.2	746.1	**58.1**	45.5	17.2	0.0	10.9
50	0.8	1244.4	**∗30.6**	18.1	27.9	0.0	12.0
	0.7	1241.7	**∗33.3**	19.0	27.5	0.0	4.4
	0.6	1237.0	**∗38.0**	22.5	27.2	0.0	1.6
	0.5	1231.5	**43.4**	25.7	18.3	0.0	0.6
	0.4	1221.9	**51.8**	31.6	18.5	0.0	0.3
	0.3	1207.8	**61.5**	40.4	19.4	0.0	0.4
	0.2	1179.3	**77.9**	55.2	22.9	0.0	0.9

(ver. 6.0) [29] and WCSP (ver. 0.49) [32]. We denote LocalSolver by LSSOL. LSSOL is a solver for general discrete optimization problems and is based on local search. WCSP is a solver for the weighted constrained satisfaction problem and is based on tabu search. For LSSOL and WCSP, we use the same formulations as the ones used for CPX-IP and CPX-CP, respectively.

All the experiments are conducted on a workstation that carries an Intel® Core™ i7-4770 Processor (up to 3.90GHz by means of Turbo Boost Technology) and 8GB main memory. The installed OS is Ubuntu 14.04.1. The ILS solver is implemented in C. Similarly to ILS, the competitors start from an initial solution that is generated by G5 [2], and the time limit of computation time is set to 10 seconds. All the parameters of the competitors are set to default values except that, in CPX-CP, `DefaultInferenceLevel` and `AllDiffInference Level` are set to `extended`.

5.3. Results

We generate 100 RAND instances for each $(n, r) \in \{30, 40, 50\} \times \{0.2, \ldots, 0.8\}$. We run the ILS solver 10 times for an instance, changing the random seed, while we run each competitor once. We summarize the result in Table 1. The table shows how the solvers improve an initial solution S_0. The averaged size of S_0 is shown in the column "$|S_0|$." The averaged improved size is then shown in the rightmost columns for each solver. Boldface indicates the largest improvement in each row. The asterisk $*$ represents a case such that a complete SLS is found for all trials.

Clearly, ILS outperforms the competitors in all tested (n, r). In particular, it founds a complete SLS for all under-constrained instances with $r \geq 0.6$. Concerning the competitors,

Table 2: Results on SLSWH instances: how often the solvers find optimal solutions

n	γ	$\|V\|/\|U\|$	ILS	LSSOL	WCSP	CPX-IP	CPX-CP
30	0.80	0.53	**100.0**	66	**100**	4	**100**
	0.70	0.37	**100.0**	53	97	25	**100**
	0.60	0.25	**100.0**	32	40	53	**100**
	0.50	0.15	**100.0**	9	6	96	**100**
	0.40	0.08	87.7	1	0	**99**	**99**
	0.33	0.05	44.3	13	1	**100**	**100**
	0.30	0.04	95.7	88	44	**100**	**100**
	0.20	0.02	**100.0**	**100**	**100**	**100**	**100**
40	0.80	0.52	**100.0**	9	92	0	**100**
	0.70	0.36	**100.0**	8	32	0	99
	0.60	0.24	**100.0**	11	2	0	92
	0.50	0.14	**100.0**	1	0	0	46
	0.40	0.08	**81.2**	0	0	1	4
	0.33	0.05	8.2	0	0	**19**	0
	0.30	0.04	6.8	0	0	**75**	25
	0.20	0.01	**100.0**	**100**	99	**100**	**100**
50	0.80	0.52	**100.0**	0	60	0	61
	0.70	0.36	**100.0**	0	7	0	38
	0.60	0.23	**100.0**	0	0	0	10
	0.50	0.14	**100.0**	0	0	0	0
	0.40	0.07	**80.9**	0	0	0	0
	0.33	0.05	**5.4**	0	0	0	0
	0.30	0.03	0.0	0	0	0	0
	0.20	0.01	**100.0**	97	73	**100**	**100**

exact solvers perform worse when n is larger, while heuristic solvers work relatively well for all n. WCSP and CPX-CP perform well especially for under-constrained instances, whereas LSSOL and CPX-IP are good for over-constrained instances. This phenomenon can be explained by the nature of the solver. For example, CPX-CP is good at under-constrained instances since such instances must have many optimal solutions; the backtracking technique of CPX-CP may be able to find one of them quickly.

Next, we show the result on SLSWH instances in Table 2. This table shows how many times the solver finds an optimal solution (i.e., a complete SLS) among 100 instances. The value for ILS is fractional since it is the average over 10 random seeds. We see that each solver performs relatively worse for a certain range of γ. Roughly speaking, all the solvers except CPX-IP are not good at instances generated by $0.3 \leq \gamma \leq 0.4$; we see the phase transition around this range. Exceptionally, CPX-IP performs well for these instances, but it is by no means effective for under-constrained instances (e.g., $\gamma \geq 0.6$).

We claim that ILS should have a higher scalability than the competitors since its performance does not deteriorate comparatively along with the increase of n. It is true that CPLEX solvers are more effective for "hard" instances with $0.3 \leq \gamma \leq 0.4$ when $n = 30$ or 40. When $n = 50$, however, ILS still finds optimal solutions for some instances while CPLEX solvers do not. Furthermore, ILS finds optimal solutions for all under-constrained instances with $\gamma \geq 0.5$ and all over-constrained instances with $\gamma = 0.2$, regardless of n, while the competitors do not perform like this. The scalability must be supported by the efficient p-neighborhood search algorithms.

6. Concluding Remarks

In this paper, we have considered fast local search technique for the MaxCSLSC problem. Specifically, we proposed p-neighborhood search algorithms for $p \in \{1, 2\}$ that run in $O(n^{p+1})$ time. For randomly generated instances, the ILS algorithm finds better solutions more frequently than the exact solvers (i.e., IP and CP solvers from IBM ILOG CPLEX [21]) and the heuristic solvers (i.e., LocalSolver [29] and WCSP [32]). We also observed that the ILS algorithm has a higher scalability than the competitors.

We could develop a 3-neighborhood search algorithm that runs in $O(n^4)$ time, by extending the 3-neighborhood search algorithm for the PLSE problem [17]. The algorithm takes into account the observation on Itoyanagi et al.'s 3-neighborhood search in the maximum independent set problem for an ordinary graph [23]. However, we do not go to this direction as we do not expect its practical value. In the computational studies for the PLSE problem in [17], the ILS algorithm with 3-neighborhood performs worst among the ILSs with various types of neighborhoods, mainly due to its inefficiency. Furthermore, the proofs would become too complicated.

Unfortunately, as the No Free Lunch Theorems go [37], the ILS algorithm does not necessarily perform well on all possible instances. In our preliminary experiments, it performs ill on the problem of constructing a complete SLS under a home-away table. This problem arises in sports scheduling [5, 19, 31], and let us review it briefly. There are n teams, where n is even, and each team has its own venue. We are given an $n \times (n-1)$ table T, where each (p, q) element, denoted by $T_{p,q}$, is either H (home) or A (away). If $T_{p,q} = \mathrm{H}$ (resp., A), then a team p should play at their own venue (resp., at the competitor's venue) in a time slot q. In other words, teams p and p' should not play in a time slot q whenever $T_{p,q} = T_{p',q}$. The problem asks for an SRRT (and thus a complete SLS) that satisfies this constraint.

For $n \in \{20, 22, \ldots, 28\}$, we generate all home-away tables that satisfy certain conditions mentioned in [31] (i.e., those having minimum "breaks" and satisfying a necessary condition for admitting a complete SLS). The ILS succeeds in constructing complete SLSs for 100%, 97%, 80%, 49% and 24% of the home-away tables for each n, respectively, while CPX-IP and CPX-CP construct complete SLSs for all home-away tables.*

We claim that the result should not diminish the value of our achievement in this paper. It just tells that the CPLEX solvers perform better than the ILS algorithm for the CSLSC problem under a home-away table. On the other hand, as we observed in Section 5, the ILS algorithm is better than the CPLEX solvers for RAND and SLSWH instances.

Based on these, we claim that the proposed efficient local search should be among the valuable technique for the CSLSC problem.

Acknowledgments

We gratefully acknowledge very careful and detailed comments given by anonymous reviewers. This work is partially supported by JSPS KAKENHI Grant Number 25870661.

References

[1] E. Aarts and J.K. Lenstra (eds.): *Local Search in Combinatorial Optimization* (John Wiley & Sons, Inc., New York, 1997).

*For larger n, CPX-CP finds complete SLSs much faster than CPX-IP. We recommend those who are interested in this issue to try a CP solver rather than an IP solver.

[2] B. Alidaee, G. Kochenberger, and H. Wang: Simple and fast surrogate constraint heuristics for the maximum independent set problem. *Journal of Heuristics*, **14** (2008), 571–585.

[3] D.V. Andrade, M.G.C. Resende, and R.F. Werneck: Fast local search for the maximum independent set problem. *Journal of Heuristics*, **18** (2012), 525–547.

[4] C. Berge: *Hypergraphs: Combinatorics of Finite Sets.* volume 45 of *North-Holland Mathematical Library* (Elsevier, Amsterdam, 1989).

[5] D. Briskorn: Feasibility of home-away-pattern sets for round robin tournaments. *Operations Research Letters*, **36** (2008), 283–284.

[6] C.J. Colbourn: Embedding partial steiner triple systems is NP-complete. *Journal of Combinatorial Theory, Series A*, **35** (1983), 100–105.

[7] D. de Werra: Scheduling in sports. *Annals of Discrete Mathematics*, **11** (1981), 381–395.

[8] K. Easton, G. Nemhauser, and M.A. Trick: The traveling tournament problem: description and benchmarks. In T. Walsh (ed.): *Proceedings of CP'01*, volume 2239 of *Lecture Notes in Computer Science* (2001), 580–584.

[9] M. Goerigk, R. Hoshino, K. Kawarabayashi, and S. Westphal: Solving the traveling tournament problem by packing three-vertex paths. In C.E. Brodley and P. Stone (eds.): *Proceedings of 28th AAAI Conference on Artificial Intelligence* (2014).

[10] M. Goerigk and S. Westphal: A combined local search and integer programming approach to the traveling tournament problem. In D. Kjenstad, A. Riise, T.E. Nordlander, B. McCollum, and E. Burke (eds.): *Proceedings of PATAT 2012* (2012), 45–56.

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

[12] C.P. Gomes and D.B. Shmoys: Completing quasigroups or latin squares: a structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations* (2002).

[13] T.F. Gonzalez (ed.): *Handbook of Approximation Algorithms and Metaheuristics* (Chapman & Hall/CRC, 2007).

[14] A.C.B. Guedes and C.C. Ribeiro: A heuristic for minimizing weighted carry-over effects in round robin tournaments. *Journal of Scheduling*, **14** (2011), 655–667.

[15] I. Hajirasouliha, H. Jowhari, R. Kumar, and R. Sundaram: On completing latin squares. In W. Thomas and P. Weil (eds.): *Proceedings of STACS 2007*, volume 4393 of *Lecture Notes in Computer Science* (2007), 524–535.

[16] K. Haraguchi: A constructive algorithm for partial latin square extension problem that solves hardest instances effectively. In S. Fidanova (ed.): *Recent Advances in Computational Optimization: Results of WCO 2013* (2015), 67–84.

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

[18] M. Henz, T. Müller, and S. Thiel: Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, **185** (2004), 92–101.

[19] A. Horbach: A combinatorial property of the maximum round robin tournament problem. *Operations Research Letters*, **38** (2010), 121–122.

[20] C.A.J. Hurkens and A. Schrijver: On the size of systems of sets every t of which have

an SDR, with an application to the worst-case ratio of heuristics for packing problems. *SIAM Journal on Discrete Mathematics*, **2** (1989), 68–72.

[21] IBM ILOG CPLEX: `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`. accessed on April 2, 2016.

[22] S. Imahori, T. Matsui, and R. Miyashiro: A 2.75-approximation algorithm for the unconstrained traveling tournament problem. *Annals of Operations Research*, **218** (2014), 237–247.

[23] J. Itoyanagi, H. Hashimoto, and M. Yagiura: A local search algorithm with large neighborhoods for the maximum weighted independent set problem. In *Proceedings of MIC 2011* (2011), 191–200.

[24] G. Kendall, S. Knust, C.C. Ribeiro, and S. Urrutia: Scheduling in sports: an annotated bibliography. *Computers and Operations Research*, **37** (2010), 1–19.

[25] T.P. Kirkman: On a problem in combinations. *The Cambridge and Dublin Mathematical Journal*, **2** (1847), 191–204.

[26] E. Klotz and A.M. Newman: Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, **18** (2013), 18–32.

[27] R. Kumar, A. Russel, and R. Sundaram: Approximating latin square extensions. *Algorithmica*, **24** (1999), 128–138.

[28] R. Lewis: Metaheuristics can solve sudoku puzzles. *Journal of Heuristics*, **13-4** (2007), 387–401.

[29] LocalSolver: `http://www.localsolver.com/`. accessed on April 2, 2016.

[30] W. Michiels, E. Aarts, and J. Korst: *Theoretical Aspects of Local Search*. Monographs in Theoretical Computer Science, an EATCS Series (Springer-Verlag New York, Inc., Secaucus, 2007).

[31] R. Miyashiro, H. Iwasaki, and T. Matsui: Characterizing feasible pattern sets with a minimum number of breaks. In E. Burke and P. de Causmaecker (eds.): *Proceedings of PATAT 2002*, volume 2740 of *Lecture Notes in Computer Science* (2003), 78–99.

[32] K. Nonobe and T. Ibaraki: An improved tabu search method for the weighted constraint satisfaction problem. *INFOR*, **39** (2001), 131–151.

[33] R.V. Rasmussen and M.A. Trick: Round robin scheduling - a survey. *European Journal of Operational Research*, **188** (2008), 617–636.

[34] C.C. Riberio: Sports scheduling: problems and applications. *International Transactions in Operations Research*, **19** (2012), 201–226.

[35] K.G. Russel: Balancing carry-over effects in round robin tournaments. *Biometrika*, **67** (1980), 127–131.

[36] M.A. Trick: Integer and constraint programming approaches for round-robin tournament scheduling. In E. Burke and P. de Causmaecker (eds.): *Proceedings of PATAT 2002*, volume 2740 of *Lecture Notes in Computer Science* (2003), 63–77.

[37] D.H. Wolpert and W.G. Macready: No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, **1** (1997), 67–82.

Kazuya Haraguchi
Faculty of Commerce
Otaru University of Commerce
Midori 3-5-21, Otaru, Hokkaido,
047-8501, JAPAN
E-mail: haraguchi@res.otaru-uc.ac.jp