Iterated Local Search with Trellis-Neighborhood for the Partial Latin Square Extension Problem*

Kazuya Haraguchi

Faculty of Commerce, Otaru University of Commerce, Japan
haraguchi@res.otaru-uc.ac.jp

Abstract

A *partial Latin square* (*PLS*) is a partial assignment of n symbols to an $n \times n$ grid such that, in each row and in each column, each symbol appears at most once. The *partial Latin square extension* problem is an NP-hard problem that asks for a largest extension of a given PLS. We consider the local search such that the neighborhood is defined by (p, q)-*swap*, i.e., the operation of dropping exactly p symbols and then assigning symbols to at most q empty cells. As a fundamental result, we provide an efficient (p, ∞)-neighborhood search algorithm that finds an improved solution or concludes that no such solution exists for $p \in \{1, 2, 3\}$. The running time of the algorithm is $O(n^{p+1})$. We then propose a novel swap operation, Trellis-swap, which is a generalization of (p, q)-swap with $p \leq 2$. The proposed Trellis-neighborhood search algorithm runs in $O(n^{3.5})$ time. The iterated local search (ILS) algorithm with Trellis-neighborhood is more likely to deliver a high-quality solution than not only ILSs with (p, ∞)-neighborhood but also state-of-the-art optimization solvers such as IBM ILOG CPLEX and LocalSolver.

1 Introduction

We address the *partial Latin square extension* (*PLSE*) problem. Let $n \geq 2$ be a natural number. Suppose that we are given an $n \times n$ grid of *cells*. A *partial Latin square* (*PLS*) is a partial assignment of n *symbols* to the grid so that the *Latin square condition* is satisfied. The Latin square condition requires that, in each row and in each column, every symbol should appear at most once. Given a PLS, the PLSE problem asks to fill as many empty cells with symbols as possible so that the Latin square condition is not violated. The problem is NP-hard (Colbourn, 1984) and has various applications such as combinatorial design, scheduling, optical routers, and combinatorial puzzles (Barry and Humblet, 1993; Colbourn and Dinitz, 2006; Gomes and Shmoys, 2002).

In this paper, we propose an effective *iterated local search* (ILS) algorithm for the PLSE problem. Being a well-known algorithmic framework, local search starts with an appropriate initial solution and then repeats moving to an improved solution as long as the *neighborhood* of the current solution contains one. The neighborhood in general is a set of solutions that are obtained by making "slight" modification on the current solution. Then local search is realized by repetition of a *neighborhood search algorithm*, which finds an improved solution in the neighborhood or concludes that no such solution exists.

For the modification on the current solution, we focus on *swap* operations throughout the paper. Given a solution PLS and non-negative integers p, q $(p < q)$, (p, q)-*swap* is an operation of dropping exactly p symbols from the solution and then inserting at most q symbols into empty cells. The (p, q)-*neighborhood* is the set of all possible solutions that are obtained by performing a (p, q)-swap on the current solution.

*The preliminary version of this paper appears in the proceedings of CPAIOR 2015 (Haraguchi, 2015). This work is partially supported by JSPS KAKENHI Grant Number 25870661.

First, as a fundamental result, we provide efficient (p, ∞)-neighborhood search algorithms for $p \in \{1, 2, 3\}$. By $q = \infty$, we mean that we insert as many symbols into the solution as possible after p symbols are dropped. The running time is $O(n^{p+1})$ time and we regard this time bound as efficient; when $p = 1$, the time bound is linear with respect to the solution size.

We then invent a novel type of swap operation, *Trellis-swap*, which is a generalization of (p, q)-swap with $p \leq 2$ and contains certain cases of $3 \leq p \leq n$. The proposed Trellis-neighborhood search algorithm runs in $O(n^{3.5})$ time.

For randomly generated instances, the ILS algorithm with Trellis-neighborhood is much more likely to deliver a better solution than not only ILS variants with (p, ∞)-neighborhoods ($p \in \{1, 2, 3\}$) but also such state-of-the-art optimization softwares as IP and CP solvers from IBM ILOG CPLEX and a general heuristic solver from LocalSolver.

To achieve these results, we reduce the PLSE problem to the *maximum independent set* (*MIS*) problem, a well-known NP-hard problem (Garey and Johnson, 1979). We then utilize Andrade et al.'s local search methodology (Andrade et al., 2012). Our approach is not merely a simple application of their strategy; we improve the efficiency by utilizing the problem structure peculiar to the PLSE problem.

The PLSE problem was first studied by Kumar et al. (1999). It has been studied especially in the context of constant-ratio approximation algorithms (Gomes et al., 2004b; Hajirasouliha et al., 2007; Haraguchi and Ono, 2014; Kumar et al., 1999). Currently the best approximation factor is achieved by a local search algorithm based on the (p, q)-neighborhood (Cygan, 2013; Fürer and Yu, 2014; Hajirasouliha et al., 2007). To the best of the author's knowledge, there is no literature that investigates efficient implementations of local search.

The decision problem version of the PLSE problem is known as the *quasigroup completion* (*QC*) problem in AI, CP and SAT communities (Ansótegui et al., 2004; Gomes and Selman, 1997; Gomes and Shmoys, 2002). The QC problem has been one of the most frequently used benchmark problems in these areas and variant problems are studied intensively, e.g., Sudoku (Crawford et al., 2008, 2009; Lambert et al., 2006; Lewis, 2007; Simonis; Soto et al., 2013), mutually orthogonal Latin squares (Appa et al., 2006a; Ma and Zhang, 2013; Vieira Jr. et al., 2011), and spatially balanced Latin squares (Gomes et al., 2004a; Le Bras et al., 2012; Smith et al., 2005). Our local search may be helpful for those who develop exact solvers for the QC problem since the local search itself or metaheuristic algorithms employing it would deliver a good initial solution or a tight lower estimate of the optimal solution size quickly.

The paper is organized as follows. Preparing terminologies and notations in Sect. 2, we explain efficient implementations of (p, ∞)-neighborhood search algorithms for $p \in \{1, 2, 3\}$ in Sect. 3. We introduce the definition of Trellis-swap and present a Trellis-neighborhood search algorithm in Sect. 4. Then we show the ILS algorithm in Sect. 5 and present experimental results in Sect. 6. Finally, we conclude the paper in Sect. 7.

2 Preliminaries

Let us begin with formulating the PLSE problem. Suppose an $n \times n$ grid of cells. We denote $[n] = \{1, 2, \ldots, n\}$. For any $i, j \in [n]$, we denote the cell in the row i and in the column j by (i, j). We consider a partial assignment of n symbols to the grid. The n symbols to be assigned are n integers in $[n]$.

We represent a PLS by means of an *orthogonal array* (Colbourn and Dinitz, 2006). Let us represent a partial assignment by a set of triples, say $T \subseteq [n]^3$, such that the membership $(v_1, v_2, v_3) \in T$ indicates that the symbol v_3 is assigned to (v_1, v_2). To avoid a duplicate assignment, we assume that, for any two triples $v = (v_1, v_2, v_3)$ and $w = (w_1, w_2, w_3)$ in T $(v \neq w)$, $(v_1, v_2) \neq (w_1, w_2)$ holds. Thus $|T| \leq n^2$ holds. For any two triples $v, w \in [n]^3$, we denote the Hamming distance between v and w by $\delta(v, w)$, i.e., $\delta(v, w) = |\{k \in [3] \mid v_k \neq w_k\}|$. We call a partial assignment $T \subseteq [n]^3$ a *PLS set* if, for any two triples $v, w \in T$ $(v \neq w)$, $\delta(v, w)$ is at least two. One easily sees that T is a PLS set iff it satisfies the Latin square condition. We say that two disjoint PLS sets S and S' are *compatible* if, for any $v \in S$ and $v' \in S'$, the distance $\delta(v, v')$ is at least two. Obviously, the union of such S and S' is a PLS set. The PLSE problem is then formulated as follows; given a

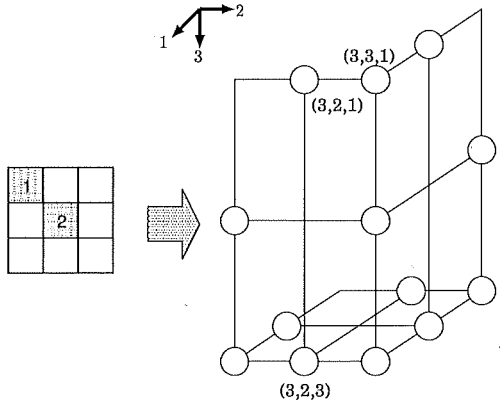

Figure 1: A PLS set $L = \{(1,1,1),(2,2,2)\}$ and the graph G constructed for it: a line segment between vertices does not indicate an edge but so does a grid line. We do not draw all grid lines in order to prevent the figure from being mess.

PLS set $L \subseteq [n]^3$, we are asked to construct a PLS set S of maximum cardinality such that S and L are compatible.

We review the MIS problem. An *undirected graph* (or simply a *graph*) $G = (V, E)$ consists of a set V of *vertices* and a set E of unordered pairs of vertices, where each element in E is called an *edge*. When two vertices are joined by an edge, we say that they are *adjacent*, or equivalently, that one of them is a *neighbor* of the other. For any vertex, the number of its neighbors is called the *degree*. An *independent set* is a subset $V' \subseteq V$ of vertices such that no two vertices in V' are adjacent. Given a graph, the MIS problem asks for a largest independent set.

Suppose a graph $G^* = (V^*, E^*)$ with vertex set $V^* = [n]^3$ and edge set $E^* = \{(v,w) \in V^* \times V^* \mid \delta(v,w) = 1\}$. The G^* is called the *intersection graph* in (Appa et al., 2006b). The following propositions are obvious.

Proposition 1 *A set $S \subseteq [n]^3$ of triples is a PLS set iff S, as a vertex set, is an independent set in G^*.*

Proposition 2 *Two PLS sets L and S are compatible with each other iff $L \cup S$ is an independent set in G^*.*

For a vertex $v \in V^*$, we denote by $N^*(v)$ the set of vertices adjacent to v, i.e., $N^*(v) = \{w \in V^* \mid \delta(v,w) = 1\}$. Clearly, we have $|N^*(v)| = 3(n-1)$. For a triple set $L \subseteq [n]^3$, we denote by $N^*(L)$ the union $\bigcup_{v \in L} N^*(v)$ over L. Then we see that the PLSE problem on a PLS set $L \subseteq [n]^3$ is equivalent to the MIS problem on a subgraph $G = (V, E)$ of G^* induced by $V = V^* \setminus (L \cup N^*(L))$. We hereafter consider solving the PLSE problem by means of solving the MIS problem.

For $v \in V$, we denote by $N(v) \subseteq N^*(v)$ the set of its neighbors in G. Since $|N(v)| \leq |N^*(v)| = 3(n-1)$ and $|V| = O(n^3)$, we have $|E| = O(n^4)$. A vertex v corresponds to a triple (v_1, v_2, v_3) and is regarded as a grid point in the 3D integral space, which is the intersection of three grid lines that are orthogonal to each other. Two vertices are adjacent iff there is a grid line that passes both of them. Hence, any independent set should contain at most one vertex among those on a grid line. A grid line is *in the direction* $d \in [3]$ if it is parallel to the d-th axis and perpendicular to the 2D plane that is generated by the other two axes. We denote by $\ell_{v,d}$ the grid line in the direction d that passes v.

In Fig. 1, we give an example of the graph G that is constructed for $L = \{(1,1,1),(2,2,2)\}$. In this figure, a line segment between vertices does not indicate an edge but so does a grid line. For example, the vertex $v = (3,2,1)$ has two neighbors: $(3,3,1)$ and $(3,2,3)$.

We call any independent set simply a *solution*. Given a solution $S \subseteq V$, we call any vertex $x \in S$ a *solution vertex* and any vertex $v \notin S$ a *non-solution vertex*. For a non-solution vertex v, we call any solution vertex in $N(v)$ a *solution neighbor of* v. We denote the set of solution

neighbors by $N_S(v)$, i.e., $N_S(v) = N(v) \cap S$. We have $|N_S(v)| \leq 3$ since v has at most one solution neighbor on one grid line and three grid lines pass v. We call the number $|N_S(v)|$ the *tightness of* v. We call v *t-tight* if $|N_S(v)| = t$. In particular, a 0-tight vertex is called *free*. We say that v is a *t-tight neighbor of* x if it is t-tight and a neighbor of x.

For each $t \in \{0, 1, 2, 3\}$, let $V_t(S)$ denote the set of t-tight vertices with respect to S. The vertex set V is partitioned into $V = S \dot{\cup} V_0(S) \dot{\cup} \ldots \dot{\cup} V_3(S)$. We expect by the following proposition that, the better the solution S is, the smaller $V_1(S)$ should be. We will utilize this expectation to improve the empirical efficiency of the local search.

Proposition 3 *For a given PLS set L, let S denote an independent set in the graph $G = (V, E)$. Then we have;*

$$|V_1(S)| \leq \frac{3n}{2}\big(n^2 - |L \cup S|\big).$$

PROOF: We defined the tightness on G, but extend the notion to G^* here. The union $L \cup S$ is an independent set in G^*. The sum of tightness with respect to $L \cup S$ over all vertices in $V^* \setminus (L \cup S)$ is exactly $3(n-1)|L \cup S|$ since every $x \in L \cup S$ has exactly $3(n-1)$ neighbors and contributes one to each neighbor's tightness. Let $V_1^*(L \cup S) \subseteq V^*$ denote the set of 1-tight vertices in G^*. The tightness sum should be bounded above by $|V_1^*(L \cup S)| + 3\big(n^3 - |L \cup S| - |V_1^*(L \cup S)|\big)$ since the tightness of each $v \notin L \cup S$ is at most three;

$$\begin{aligned} 3(n-1)|L \cup S| &\leq |V_1^*(L \cup S)| + 3\big(n^3 - |L \cup S| - |V_1^*(L \cup S)|\big), \\ |V_1^*(L \cup S)| &\leq \frac{3n}{2}\big(n^2 - |L \cup S|\big). \end{aligned}$$

Any vertex v in $V_1(S)$ belongs to $V_1^*(L \cup S)$ since, in G^*, v is not adjacent to any vertex in L due to $v \in V = V^* \setminus (L \cup N^*(L))$ but is adjacent to exactly one vertex in S due to $v \in V_1(S)$. Then we have $|V_1(S)| \leq |V_1^*(L \cup S)|$. □ □

Given a solution S, a *swap* operation in general drops a subset $D \subseteq S$ from S and then inserts a subset I into S so that $(S \setminus D) \cup I$ continues to be a solution. Dropping D from S makes certain vertices free: all vertices in D and non-solution vertices whose solution neighbors are completely contained in D. The inserted I should be an independent set among these free vertices. If there are D and I with $|D| < |I|$, then $(S \setminus D) \cup I$ is an improved solution.

For two integers p, q with $0 \leq p < q$, *(p, q)-swap* refers to a swap operation with $|D| = p$ and $|I| \leq q$. The *(p, q)-neighborhood of a solution S* is defined as the set of all possible solutions that are obtained by performing a (p, q)-swap on S. A solution is *(p, q)-maximal* if its (p, q)-neighborhood does not contain an improved solution. When $p \leq p'$ and $q \leq q'$, the (p, q)-neighborhood is a subset of the (p', q')-neighborhood. Hence, the (p, ∞)-neighborhood is the largest neighborhood for a fixed p. We call a solution *p-maximal* if it is (p, ∞)-maximal. In particular, we call a 0-maximal solution simply a *maximal* solution. Being p-maximal implies that S is also p'-maximal for any $p' < p$. Thus, p'-maximality is necessary for p-maximality.

A *(p, q)-neighborhood search algorithm* is one that finds an improved solution in the (p, q)-neighborhood of the input solution or decides that no such solution exists. Once a (p, q)-neighborhood search algorithm is established, it is immediate to design a local search algorithm that computes a (p, q)-maximal solution; starting with an initial solution, we repeat moving to an improved solution as long as the neighborhood search algorithm delivers one.

3 Efficient (p, ∞)-Neighborhood Search Algorithms

In this section, we present efficient (p, ∞)-neighborhood search algorithms for $p \in \{1, 2, 3\}$. We borrow the data structure from Andrade et al.'s local search on the MIS problem (Andrade et al., 2012), and improve the efficiency by introducing additional ideas. We explain the data structure in Sect. 3.1 and the algorithms in Sect. 3.2.

The time complexity of the algorithms is $O(n^{p+1})$. We claim that it should be far from trivial to achieve this time bound. Concerning previous local search algorithms for the MIS problem,

their direct usage would require more computation time. Andrade et al.'s $(1,2)$-neighborhood search algorithm (resp., $(2,3)$-neighborhood search algorithm) requires $O(|E|) = O(n^4)$ time (resp., $O(\Delta|E|) = O(n^5)$ time, where Δ denotes the maximum degree in the graph). Itoyanagi et al. (2011) extended Andrade et al.'s work to the maximum weighted independent set problem. Their $(3,4)$-neighborhood search algorithm runs in $O(\Delta^2|E|) = O(n^6)$ time.

3.1 Data Structure

The data structure mainly consists of an ordering of vertices and a 3D array of vertices. The ordering is motivated by Andrade et al. (2012) and used to store the solution structure. We can scan vertices of a particular type (e.g., solution vertices, free vertices) in linear time with respect to the number. We introduce the 3D array to store the graph $G = (V, E)$. We can access the vertex in a specified coordinate if it exists or decide that no such vertex exists in $O(1)$ time.

We denote an ordering of vertices by a bijection $\pi : V \to [|V|]$. In π, every solution vertex is ordered ahead of all the non-solution vertices. Among the non-solution vertices, every free vertex is ordered ahead of all the non-free vertices, and among the non-free vertices, every 1-tight vertex is ordered ahead of all 2-tight and 3-tight vertices. That is, $\pi(x) \leq \pi(v)$ holds whenever $x \in S$ and $v \notin S$, $\pi(v) \leq \pi(v')$ holds whenever $v \in V_0(S)$ and $v' \in V_1(S) \cup V_2(S) \cup V_3(S)$, and $\pi(v') \leq \pi(v'')$ holds whenever $v' \in V_1(S)$ and $v'' \in V_2(S) \cup V_3(S)$. In each of the four sections (i.e., solution vertices, free vertices, 1-tight vertices and other non-free vertices), 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 denote a 3D $n \times n \times n$ array by C. For each triple $(v_1, v_2, v_3) \in [n]^3$, 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, we let it have a null pointer. The 3D array stores the edge 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 for every $v_1', v_2', v_3' \in [n]$ such that $v_1' \neq v_1$, $v_2' \neq v_2$ and $v_3' \neq v_3$.

We list the maintained parameters as follows.

$\#_{\mathrm{sol}}$, $\#_0$ and $\#_1$: These are counters of the solution size $|S|$, the number $|V_0(S)|$ of free vertices and the number $|V_1(S)|$ of 1-tight vertices, respectively. Using them, we can access the head of each section of the vertex ordering in $O(1)$ time.

$\lambda_d(x)$ for $(x, d) \in S \times$ [3]: This is a counter of the number of 1-tight neighbors of x that are in the direction d. Thus $\lambda_d(x)$ is defined as;

$$\lambda_d(x) = \left|\left\{v \in N(x) \mid N_S(v) = \{x\},\ v_d \neq x_d\right\}\right|.$$

$\tau(v)$ for $v \in V \setminus S$: This is a counter of the tightness $|N_S(v)|$ of a non-solution vertex v.

$\rho_d(v)$ for $(v, d) \in (V \setminus S) \times$ [3]: This is a pointer to the solution neighbor of v in the direction d; when v has no such solution neighbor, we let $\rho_d(v)$ have a null pointer.

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. In Fig. 2, we illustrate how solution vertices and non-solution vertices are distributed in the 3D space. In the vertex ordering π, the solution vertices (e.g., x, y, z) are ordered ahead of non-solution vertices. There is no free vertex. Among the non-solution vertices, the 1-tight vertices (e.g., u, v, w, b) are ordered ahead of 2-tight and 3-tight vertices (e.g., a). Concerning the parameters above, we have $\#_{\mathrm{sol}} = 5$, $\#_0 = 0$ and $\#_1 = 5$. We have $(\lambda_1(x), \lambda_2(x), \lambda_3(x)) = (1, 1, 1)$ and $(\lambda_1(y), \lambda_2(y), \lambda_3(y)) = (0, 1, 0)$. The tightness τ of a non-solution vertex is indicated within the corresponding circle $\circ$. For the pointer ρ, we have $(\rho_1(u), \rho_2(u), \rho_3(u)) = (\textsc{Null}, x, \textsc{Null})$, for example.

We show time complexities of some elementary operations.

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

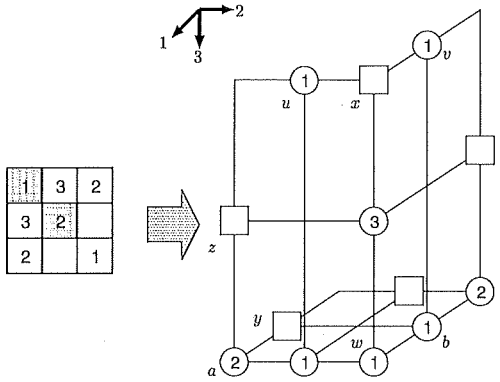

Figure 2: A solution $S = \{(1,2,3),(1,3,2),(2,1,3),(3,1,2),(3,3,1)\}$ and how vertices are distributed in the 3D space: A square □ indicates a solution vertex and a circle ○ indicates a non-solution vertex, where the number in ○ indicates the tightness.

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

Drop: We can drop a solution vertex x from S in $O(n)$ time as follows. We set its tightness $\tau(x)$ to zero and update π so that x falls into the free vertex section. For each neighbor $v \in N(x)$, we decrease the tightness $\tau(v)$ by one. If $\tau(v)$ becomes zero (resp., one), then we update π so that v falls into the free vertex (resp., 1-tight vertex) section. We also update the other parameters accordingly.

The procedure is summarized as DROP in Algorithm 1. First, in line 2, we set the tightness $\tau(x)$ to zero and the pointers $\rho_1(x), \rho_2(x), \rho_3(x)$ to solution neighbors to a null value as it has no solution neighbors. By lines 3 and 4, x falls into the free vertex section. The procedure EXCHANGE(p, q) updates π so that the p-th and q-th vertices are exchanged. For each neighbor $v \in N(x)$, we decrease the tightness $\tau(v)$ by one (line 6). Let d denote the direction of the grid line that passes both x and v. We release its pointer $\rho_d(v)$ to x since x is no longer a solution vertex (line 8).

- If $\tau(v)$ is decreased to zero, then v falls into the free vertex section (lines 9 to 11).
- If $\tau(v)$ is decreased to one, then v falls into the 1-tight vertex section. Note that v has a unique solution neighbor in a certain direction $d' \neq d$, which is stored as $\rho_{d'}(v)$. Let $x' = \rho_{d'}(v)$. Now that x' has a new 1-tight neighbor, we increase the number $\lambda_{d'}(x')$ by one (lines 12 to 16).

The total time complexity is $O(n)$. Let us introduce a brief example. In Fig. 2, if y is dropped, then y becomes free, and $\tau(a)$ and $\tau(b)$ are decreased to one and zero, respectively. Then a and b fall into the 1-tight vertex section and the free vertex section, respectively. The 1-tight a has the unique solution neighbor in the direction 3, that is z. We increase $\lambda_3(z)$ by one.

Insertion: We can insert a free vertex into S in $O(n)$ time in a manner analogous to the drop operation.

3.2 Algorithms

The (p, ∞)-neighborhood search algorithms have the similar structure among $p = 1$, 2 and 3. In principle, we need to search all subsets D of S of size p. We do this by means of scanning "trigger" vertex sets. Based on what we call a trigger, we generate a subset D to be dropped from S. For example, when $p = 1$, a trigger is a solution vertex x, and D is $\{x\}$ itself. When $p = 2$, it is a

Algorithm 1 A procedure for dropping a solution vertex x from a solution

global variables: parameters $\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.

procedure DROP(x)
  $\tau(x) \leftarrow 0$, $\rho_1(x), \rho_2(x), \rho_3(x) \leftarrow$ NULL
  EXCHANGE($\pi(x), \#_{\mathrm{sol}}$)
  $\#_{\mathrm{sol}} \leftarrow \#_{\mathrm{sol}} - 1$, $\#_0 \leftarrow \#_0 + 1$
  for all $v \in N(x)$ **do**
    $\tau(v) \leftarrow \tau(v) - 1$
    $d \leftarrow$ the direction of the grid line that passes both x and v
    $\rho_d(v) \leftarrow$ NULL
    if $\tau(v) = 0$ **then**
      EXCHANGE($\#_{\mathrm{sol}} + \#_0 + 1, \pi(v)$)
      $\#_0 \leftarrow \#_0 + 1$, $\#_1 \leftarrow \#_1 - 1$
    else if $\tau(v) = 1$ **then**
      EXCHANGE($\#_{\mathrm{sol}} + \#_0 + \#_1 + 1, \pi(v)$)
      $\#_1 \leftarrow \#_1 + 1$
      $d' \leftarrow$ the direction index such that $\rho_{d'}(v) \neq$ NULL
      $x' \leftarrow \rho_{d'}(v)$, $\lambda_{d'}(x') \leftarrow \lambda_{d'}(x') + 1$
    end if
  end for
end procedure

procedure EXCHANGE(p, q) ▷ exchange the p-th and q-th vertices in π
  $u \leftarrow \pi^{-1}(p)$, $v \leftarrow \pi^{-1}(q)$
  $\pi(u) \leftarrow q$, $\pi(v) \leftarrow p$
end procedure

2-tight vertex u, and D is the set of two solution neighbors of u. The point is that, for each trigger, we generate D in $O(1)$ time somehow.

Let $F = V_0(S \setminus D)$ denote the set of vertices free from $S \setminus D$, and G_F denote the subgraph induced by F. The inserted vertices should form an independent set within G_F. The point is that we count the MIS size of G_F, denoted by $\#_{\mathrm{MIS}}$, without finding an MIS itself. Surprisingly, we can count $\#_{\mathrm{MIS}}$ in $O(1)$ time. Only when $\#_{\mathrm{MIS}} > p$, we search for an MIS to be inserted, denoted by I, and thereby obtain an improved solution $(S \setminus D) \cup I$. The search for I requires $O(n)$ time. Since the $O(n)$-time task is done at most once during the search of triggers, the overall time complexity is linear with respect to the number of searched vertex sets.

Based on this top-level strategy, we realize the (p, ∞)-neighborhood search algorithms below.

Case of $p = 1$. Let S be a maximal solution. Figure 2 shows a situation in which the $(1, \infty)$-neighborhood contains an improved solution. If x is dropped from the solution, then the tightness of every neighbor is decreased by one. In particular, all 1-tight neighbors (i.e., u, v and w) become free. All of the three vertices can be inserted into the solution since any two of them are not adjacent to each other. We have an improved solution $(S \setminus \{x\}) \cup \{u, v, w\}$.

We generalize this example. The $(1, \infty)$-neighborhood contains an improved solution iff there are $x \in S$ and $u, v \notin S$ such that $(S \setminus \{x\}) \cup \{u, v\}$ is a solution. It is clear that u and v should be neighbors of x. They are 1-tight, and their unique solution neighbor is x. The u and v should not be adjacent, which implies that u and v are not on the same grid line. We define $\Lambda(x)$ as a subset of the direction indices in which x has a 1-tight neighbor, i.e., $\Lambda(x) = \{d \in [3] \mid \lambda_d(x) > 0\}$. Then $\#_{\mathrm{MIS}}$, the largest number of vertices that can be inserted into $S \setminus \{x\}$ simultaneously, is given by the cardinality $|\Lambda(x)|$.

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

Algorithm 2 A $(1, \infty)$-neighborhood search algorithm for a maximal solution

global variables: parameters $\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.
require: the parameters as a whole must represent a maximal solution.

```
function 1-NS
    for i = 1, 2, ..., #sol do
        x ← π^{-1}(i)
        if |Λ(x)| ≥ 2 then
            I ← PickUpOneTight(x)
            drop x from the solution, and insert each vertex in I into the solution
            return "an improved solution is found"
        end if
    end for
    return "the solution is 1-maximal"
end function

function PickUpOneTight(x)
    I ← ∅
    for all d ∈ Λ(x) do
        v ← PickUpOneTightOnGrid(x, d)
        I ← I ∪ {v}
    end for
    return I
end function

function PickUpOneTightOnGrid(x, d)
    v ← an arbitrary 1-tight vertex in N(x) in direction d
    return v
end function
```

PROOF: We assume the given solution S to be maximal; we can check in $O(1)$ time whether S is maximal or not. If it is not maximal, we have an improved solution by inserting any free vertex into S in $O(n)$ time.

An improved solution exists iff there is $x \in S$ with $|\Lambda(x)| \geq 2$. Then we use each solution vertex as what we call a trigger. All solution vertices can be searched by sweeping the first section of the vertex ordering π. There are at most n^2 solution vertices. For each solution vertex x, the number $|\Lambda(x)|$ can be computed in $O(1)$ time. If x with $|\Lambda(x)| \geq 2$ is found, we can determine an MIS I to be inserted in $O(n)$ time, searching the grid line $\ell_{x,d}$ for an arbitrary 1-tight vertex for every direction $d \in \Lambda(x)$. Then we have an improved solution in $O(n)$ time by dropping x from S and by inserting all vertices in I into $S \setminus \{x\}$. □ □

In Algorithm 2, we present a function 1-NS that outputs whether the solution maintained by the parameters is 1-maximal or not. The solution is assumed to be maximal. If it is not 1-maximal, then the function searches for an improved solution and updates the parameters so that the improved solution is maintained. It then outputs "an improved solution is found." Two functions PICKUPONETIGHT(x) and PICKUPONETIGHTONGRID(x, d) are used as subroutines to search for an improved solution. The function PICKUPONETIGHT(x) returns a maximal subset of 1-tight neighbors of x that are on different grid lines from each other. Any subset returned by PICKUPONETIGHT(x) is an independent set and can be inserted into $S \setminus \{x\}$. Note that the size of the independent set is at most three. To collect vertices in the independent set, the function PICKUPONETIGHTONGRID(x, d) is called for each $d \in \Lambda(x)$; it returns an arbitrary 1-tight neighbor of x on the grid line $\ell_{x,d}$. Given x, the two subroutines run in $O(n)$ time.

We may improve the empirical efficiency by using 1-tight vertices as triggers; we search 1-tight vertices, instead of searching solution vertices themselves. For each 1-tight vertex, we test its unique solution neighbor x as the vertex to be dropped. We never miss an improved solution

Algorithm 3 An alternative $(1, \infty)$-neighborhood search algorithm for a maximal solution

global variables: parameters $\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.
require: the parameters as a whole must represent a maximal solution.

```
function 1-NS*
    for i = 1, 2, ..., #_1 do
        u ← π^{-1}(#_sol + #_0 + i)                    ▷ u is a 1-tight vertex
        x ← the unique solution neighbor of u
                                  ▷ x can be decided by tracing the pointers ρ_d(u)
        if |Λ(x)| ≥ 2 then
            I ← PickUpOneTight(x)
            drop x from the solution, and insert each vertex in I into the solution
            return "an improved solution is found"
        end if
    end for
    return "the solution is 1-maximal"
end function
```

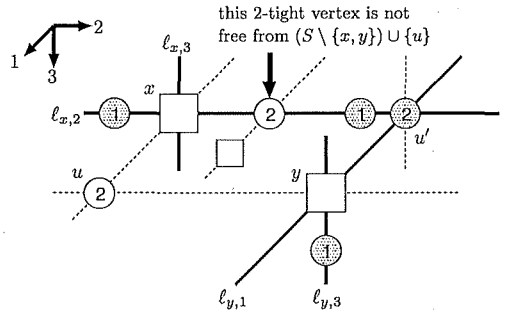

Figure 3: An illustration of the case in which a $(2, \infty)$-swap can be made: Shaded vertices are free from $(S \setminus \{x, y\}) \cup \{u\}$.

because all x'-s with $|\Lambda(x)| \geq 1$ are covered by this strategy. The number of searched vertices is $\#_1 = |V_1(S)|$, which is expected to be smaller than $\#_{\mathrm{sol}} = |S|$ when $|S|$ is large to some degree; this expectation comes from Proposition 3. We show this version of a $(1, \infty)$-neighborhood search algorithm in Algorithm 3, as the function named 1-NS*. In fact, to the extent of our preliminary experiments, 1-NS* runs faster than 1-NS most of the time.

Case of $p = 2$. Let S be a 1-maximal solution. The $(2, \infty)$-neighborhood contains an improved solution iff there exist $x, y \in S$ and $u, v, w \notin S$ such that $(S \setminus \{x, y\}) \cup \{u, v, w\}$ is a solution. These vertices should satisfy Lemmas 1 to 4 in (Andrade et al., 2012), which are conditions established for the general MIS problem. According to the conditions, each vertex in $\{u, v, w\}$ is either 1-tight or 2-tight, and at least one of them is 2-tight. The unique solution neighbor of a 1-tight vertex is either x or y, and the two solution neighbors of a 2-tight vertex are x and y.

Let u be a 2-tight vertex and let x and y be its solution neighbors. We denote by F the set of vertices free from $(S \setminus \{x, y\}) \cup \{u\}$. We would like to know the MIS size of G_F. To observe how vertices in F are distributed, see Fig. 3. In the figure, we take the coordinates so that x (resp., y) is the u'-s solution neighbor in the direction 1 (resp., 2). Indicated by shade, vertices in F are among the four solid grid lines, that is, $\ell_{x,2}$, $\ell_{x,3}$, $\ell_{y,1}$ and $\ell_{y,3}$. Let us denote the intersection point of $\ell_{x,2}$ and $\ell_{y,1}$ by u'. Formally, it is defined as $u' = (u'_1, u'_2, u'_3) = (x_1, y_2, u_3)$, where u_3 equals to x_3 and y_3. The u' is the only vertex in F that can be 2-tight. Note that it does not necessarily exist and that it is not necessarily 2-tight; it may have another solution neighbor in the direction 3. Then F consists of all 1-tight vertices on the four grid lines and u', where $u' \in F$ only when it exists and is 2-tight.

Lemma 1 *The MIS size of G_F is either $|\Lambda(x)\setminus\{1\}|+|\Lambda(y)\setminus\{2\}|$ or $|\Lambda(x)\setminus\{1\}|+|\Lambda(y)\setminus\{2\}|+1$.*

PROOF: The former is the number of solid grid lines that have a 1-tight vertex. We can construct an independent set of that size by taking a 1-tight vertex from each grid line because any two of such vertices are not adjacent. We cannot insert any other 1-tight vertex into this independent set, but if u' is in F, we may be able to insert it into the solution. □ □

Lemma 2 *The MIS size of G_F is $|\Lambda(x)\setminus\{1\}|+|\Lambda(y)\setminus\{2\}|+1$ iff the vertex u' exists, it is 2-tight, and $\lambda_2(x)=\lambda_1(y)=0$.*

PROOF: If the MIS size is $|\Lambda(x)\setminus\{1\}|+|\Lambda(y)\setminus\{2\}|+1$, every MIS should contain the only 2-tight vertex in F, that is u'. Then we have $\lambda_2(x)=\lambda_1(y)=0$ since, if not so, we could construct an MIS that does not contain u' by exchanging u' and any 1-tight vertex on $\ell_{x,2}$ or $\ell_{y,1}$. The sufficiency is obvious. □ □ By Lemmas 1 and 2, we have the following lemma immediately.

Lemma 3 *Given a 2-tight vertex u, the MIS size of G_F can be computed in $O(1)$ time.*

PROOF: We can recognize the solution neighbors x and y in $O(1)$ time by tracing the pointers $\rho_d(u)$. Then the MIS size of G_F can be computed in $O(1)$ time from the last two lemmas. □ □

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

PROOF: Similarly to the proof of Theorem 1, we assume that S is 1-maximal.

We use each 2-tight vertex u as a trigger. All 2-tight vertices can be searched by sweeping the last section of the vertex ordering, and their number is at most $|V\setminus S|\le|V|\le n^3$. For u, we can compute the MIS size of G_F in $O(1)$ time from Lemma 3, where F is the set of vertices free from $S\setminus\{x,y\}\cup\{u\}$. If the MIS size is no less than two, there exists an improved solution. An MIS of G_F is decided in $O(n)$ time by searching the four grid lines. □ □

In Algorithm 4, we present a function 2-NS that outputs whether the solution is 2-maximal or not. The solution is assumed to be 1-maximal. If it is not 2-maximal, then the function searches for an improved solution and updates the parameters so that the improved solution is maintained. It then outputs "an improved solution is found." The subroutine ISIMPROVABLE(u) is called for each 2-tight vertex u, in order to decide whether G_F constructed from u has an MIS of size no less than two. In this subroutine, the coordinates are permuted by a bijection $\phi:[3]\to[3]$ so that x (resp., y) is the u'-s neighbor in the direction $\phi(1)$ (resp., $\phi(2)$); see line 14. We have $|\Lambda(x)|\le 1$ for any solution vertex $x\in S$ since S is 1-maximal. Then $\#_{\mathrm{MIS}}\le 3$ holds and thus the total number of inserted vertices is at most four.

Case of $p=3$. We introduce the following result in order to reduce the search space.

Theorem 3 (Itoyanagi et al. (2011)) *Let S be a 2-maximal solution. Suppose that there exist $x,y,z\in S$ and $u,v,w,s\notin S$ such that $(S\setminus\{x,y,z\})\cup\{u,v,w,s\}$ is a solution. Without loss of generality, we assume $\tau(u)\ge\tau(v)\ge\tau(w)\ge\tau(s)$. Then we are in either of the following two cases.*

(I) *u is 3-tight such that $N_S(u)=\{x,y,z\}$.*

(II) *u and v are 2-tight such that $N_S(u)=\{x,y\}$ and $N_S(v)=\{x,z\}$.*

To cover (I), we use a 3-tight vertex as a trigger. To cover (II), we use a pair of non-adjacent 2-tight vertices that have exactly one solution neighbor in common as a trigger. The following Lemma 4 (resp., Lemma 5) states that, when we are given a trigger in the case (I) (resp., (II)), the largest number of vertices to be inserted can be decided in $O(1)$ time.

Algorithm 4 A $(2, \infty)$-neighborhood search algorithm for a 1-maximal solution

global variables: parameters $\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.
require: the parameters as a whole must represent a 1-maximal solution.

function 2-NS
  for $i = \#_{\mathrm{sol}} + \#_0 + \#_1 + 1, \ldots, |V|$ **do**
    $u \leftarrow \pi^{-1}(i)$ ▷ u is either 2-tight or 3-tight
    if $\tau(u) = 2$ **then**
      if IsImprovable(u) is True **then**
        return "an improved solution is found"
      end if
    end if
  end for
  return "the solution is 2-maximal"
end function

function IsImprovable(u)
  $x, y \leftarrow$ solution neighbors of u
  $\phi \leftarrow$ the permutation $\phi : [3] \to [3]$ of coordinates such that x (resp., y) is the neighbor of u in the direction $\phi(1)$ (resp., $\phi(2)$).
  $\#_{\mathrm{MIS}} \leftarrow |\Lambda(x) \setminus \{\phi(1)\}| + |\Lambda(y) \setminus \{\phi(2)\}|$
  if $\phi(2) \notin \Lambda(x)$, $\phi(1) \notin \Lambda(y)$, $N(x) \cap N(y)$ contains $u' \neq u$, and $\tau(u') = 2$ **then**
    $\#_{\mathrm{MIS}} \leftarrow \#_{\mathrm{MIS}} + 1$
  end if
  if $\#_{\mathrm{MIS}} \leq 1$ **then**
    return False
  end if
  $I \leftarrow \{u\}$
  for all $d \in \Lambda(x) \setminus \{\phi(1)\}$ **do**
    $I \leftarrow I \cup \{\mathrm{PickUpOneTightOnGrid}(x, d)\}$ ▷ see Algorithm 2
  end for
  for all $d \in \Lambda(y) \setminus \{\phi(2)\}$ **do**
    $I \leftarrow I \cup \{\mathrm{PickUpOneTightOnGrid}(y, d)\}$ ▷ see Algorithm 2
  end for
  if $\phi(2) \notin \Lambda(x)$, $\phi(1) \notin \Lambda(y)$, $N(x) \cap N(y)$ contains $u' \neq u$, and $\tau(u') = 2$ **then**
    $I \leftarrow I \cup \{u'\}$
  end if
  drop x and y from the solution, and insert each vertex in I into the solution
  return True
end function

Lemma 4 *Let u denote a 3-tight vertex. Let x, y and z be solution neighbors of u and F be the set of vertices free from $(S \setminus \{x, y, z\}) \cup \{u\}$. Once u is given, the MIS size of G_F can be decided in $O(1)$ time.*

Proof: The solution neighbors x, y and z can be decided in $O(1)$ time by tracing the pointers $\rho_d(u)$. We illustrate a typical situation in Fig. 4. The vertices in F are among solid grid lines. All of them are either 1-tight or 2-tight. Analogously to the case of $p = 2$, at most three vertices are 2-tight, that is, u', u'' and u'''. Note that they are not necessarily 2-tight; e.g., in Fig. 4, u'' is 3-tight and u''' does not exist. We see that the MIS size is between α and $\alpha + 3$, where $\alpha = |\Lambda(x) \setminus \{1\}| + |\Lambda(y) \setminus \{2\}| + |\Lambda(z) \setminus \{3\}|$. We can decide the exact value $\alpha + \beta$ in $O(1)$ time; the number β represents how many vertices in $\{u', u'', u'''\}$ belong to every MIS. We can compute it in $O(1)$ time in an analogous way to the proof of Lemma 2. □ □

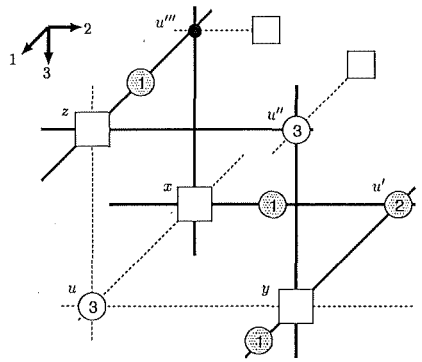

Figure 4: A typical situation discussed in the proof of Lemma 4: Shaded vertices are free from $(S \setminus \{x, y, z\}) \cup \{u\}$.

Lemma 5 *Let u and v denote 2-tight vertices that are not adjacent to each other and that have exactly one solution neighbor in common. Let $N_S(u) = \{x, y\}$, $N_S(v) = \{x, z\}$ and F be the set of vertices free from $(S \setminus \{x, y, z\}) \cup \{u, v\}$. Once u and v are given, the MIS size of G_F can be decided in $O(1)$ time.*

PROOF: We illustrate a typical situation in Fig. 5. The vertices in F are among solid grid lines. All of them are either 1-tight or 2-tight.

We construct another graph $G'_F = (I_1 \cup I_2, J)$ so that $|I_1 \cup I_2|$ and $|J|$ are constants and that G_F and G'_F have the same MIS size. By this, we can compute the MIS size of G_F in $O(2^{O(1)})$ time. We introduce the definition of G'_F; The I_1 is the set of "solid" grid lines that contain a 1-tight vertex; by a solid grid line, like one in the figure, we mean a grid line that passes one of $\{x, y, z\}$ but does not pass u or v. The I_2 is the set of 2-tight vertices that are at intersecting points of some two solid grid lines in I_1. Thus, a vertex in I_1 corresponds to a grid line, and a vertex in I_2 corresponds to a 2-tight vertex. Two vertices a and b are joined by an edge if they satisfy the appropriate condition among the following:

(i) $a, b \in I_1 \Rightarrow$ The grid lines a and b have only one 1-tight vertex, respectively, and these 1-tight vertices are adjacent to each other.

(ii) $a \in I_1$ **and** $b \in I_2 \Rightarrow$ The 2-tight vertex b is on the grid line a.

(iii) $a, b \in I_2 \Rightarrow$ The 2-tight vertices a and b are adjacent to each other.

In Fig. 6, we show G'_F that is constructed from F in the situation of Fig. 5.

We claim that G_F and G'_F have the same MIS size. A member of any independent set in G_F is either a 1-tight vertex on the grid line in I_1 or a 2-tight vertex in I_2. All of them appear as vertices in G'_F. Also, two vertices in G'_F are joined by an edge iff they should not belong to the same independent set in G_F.

Clearly, we have $|I_1| \leq 5$, $|I_2| \leq \binom{5}{2} = 10$, and $|J| \leq \binom{15}{2} = 105$. The remaining task is to show that, once u and v are given, we can construct G'_F in $O(1)$ time. The x, y and z can be decided in $O(1)$ time. We can decide I_1 and I_2 in $O(1)$ time. For each $(a, b) \in (I_1 \cup I_2) \times (I_1 \cup I_2)$, whether $(a, b) \in J$ or not can be identified in $O(1)$ time. We should give a remark on the case of (i), i.e., $a, b \in I_1$. We claim that no two grid lines in I_1 are parallel. Then a and b are either intersecting or skew, and only when they are skew, a 1-tight vertex on a and a 1-tight vertex on b can be adjacent. Furthermore, from elementary geometry, such a pair of adjacent 1-tight vertices is unique if it exists; see w and w' in Fig. 5. Whether both a and b have exactly one 1-tight vertex, respectively, can be identified in $O(1)$ time by using the parameter λ, and whether these 1-tight vertices are adjacent or not can be identified in $O(1)$ time by using the 3D array C. □ □

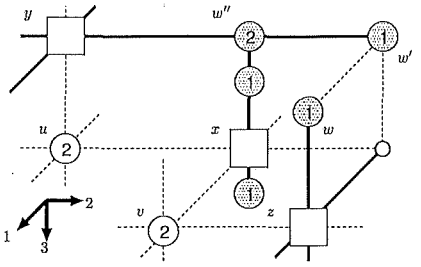

Figure 5: A typical situation discussed in the proof of Lemma 5: Shaded vertices are free from $(S \setminus \{x, y, z\}) \cup \{u, v\}$.

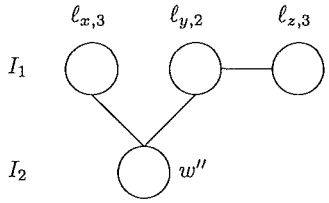

Figure 6: The graph $G'_F = (I_1 \cup I_2, J)$ constructed from F of Fig. 5

Theorem 4 *Given a solution S, we can find an improved solution in its $(3, \infty)$-neighborhood or conclude that it is 3-maximal in $O(n^4)$ time.*

PROOF: We assume S to be 2-maximal, similarly to the previous theorems.

Recall (I) and (II) in Theorem 3. We claim that all triggers are searched in $O(n^4)$ time; For (I), all 3-tight vertices are searched in $O(n^3)$ time by sweeping the last section of the vertex ordering π. For (II), all pairs of non-adjacent 2-tight vertices that have exactly one solution neighbor in common are searched in $O(n^4)$ time; we search every solution vertex $x \in S$ and every pair of its 2-tight neighbors that are not on the same grid line.

For each trigger, the largest number of vertices to be inserted is computed in $O(1)$ time from Lemmas 4 and 5. If the number exceeds three, then an improved solution exists; an MIS to be inserted can be decided in $O(n)$ time by searching the grid lines passing the three solution vertices to be dropped. □ □

The number of vertices to be inserted is at most six. Unfortunately, as we will observe in Sect. 6, the ILS with $(3, \infty)$-neighborhood is inferior to ILSs with other neighborhoods due to its inefficiency. We skip the description of the neighborhood search algorithm since it would be too lengthy.

4 Trellis-Neighborhood Search Algorithm

In this section, we propose a novel type of neighborhood, Trellis-neighborhood, and a neighborhood search algorithm that runs in $O(n^{3.5})$ time.

Trellis-neighborhood is a generalization of (p, ∞)-neighborhood with $p \leq 2$. Let $d \in [3]$ and $k \in [n]$ be any integers. We regard a vertex v as a 3D integral point $v = (v_1, v_2, v_3)$. Cutting the $n \times n \times n$ 3D integral cube by a 2D plane $v_d = k$, we have a 2D face. We call this face the (d, k)-*face*. There are possibly some vertices on the (d, k)-face. In Trellis-swap, the set D of solution vertices to be dropped is any subset of solution vertices on the face. We define the *Trellis-neighborhood of a solution S* as the set of all solutions that can be obtained by dropping such D from S and then inserting an independent set I into $S \setminus D$. Observe that (p, ∞)-neighborhood with $p \leq 2$ is a special case of Trellis-neighborhood in the sense that the cardinality $|D|$ is restricted to p.

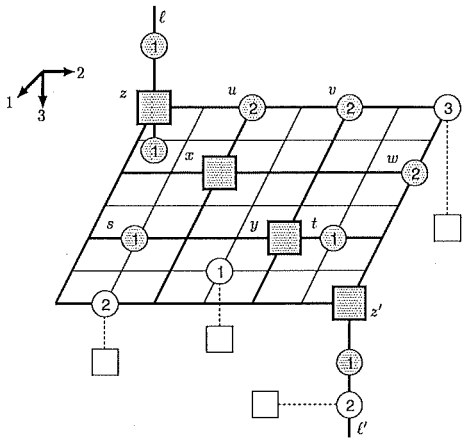

Figure 7: An example of trellis: Four bold squares on the 2D face are the solution vertices to be dropped. Vertices in the trellis are indicated by shade.

We claim that, even when D is maximal (i.e., D is the set of all solution vertices on a (d,k)-face), a largest I can be computed efficiently in $O(n^{2.5})$ time since the problem of finding a largest I is reduced to the maximum bipartite matching problem. In what follows, we restrict ourselves to such D. Then $|D|$ is at most n and there are at most $3n$ different D'-s. Let us define the (d,k)-*Trellis*, a certain subgraph of G.

Definition 1 *Suppose that we are given a solution S. For any $d \in [3]$ and $k \in [n]$, let $D \subseteq S$ be a subset $D = \{(v_1,v_2,v_3) \in S \mid v_d = k\}$. We denote by F_1 (resp., F_2) the set of all 1-tight (resp., 2-tight) vertices such that the unique solution neighbor is contained in D (resp., both of the solution neighbors are contained in D). We define the* (d,k)-trellis *as the subgraph of G induced by $D \cup F_1 \cup F_2$.*

The point is that any independent set in the trellis can be inserted into $S \setminus D$ since dropping D from S makes all vertices in $D \cup F_1 \cup F_2$ free. Furthermore, no other vertex can be free. This motivates us to find an MIS in the trellis. Below we call the (d,k)-face and the (d,k)-trellis simply the face and the trellis, respectively, when d and k are clear from the context.

We show an example of trellis in Fig. 7. The D consists of four solution vertices on the 2D face, i.e., $D = \{x,y,z,z'\}$. All vertices in the trellis are indicated by shade; F_1 is the set of five 1-tight vertices, and F_2 is the set of three 2-tight vertices. We see that all vertices in $D \cup F_2$ are on the 2D face. Two vertices in F_1 are also on the same face, while the three other vertices in F_1 are out of the face like a hanging vine.

Lemma 6 *Given $d \in [3]$ and $k \in [n]$, we can compute an MIS of the (d,k)-trellis in $O(n^{2.5})$ time.*

PROOF: We explain how to compute an MIS of the trellis. Let us partition F_1 into $F_1 = F_1' \cup F_1''$ so that F_1' (resp., F_1'') is the subset of vertices on the face (resp., out of the face). In the subgraph $G_{F_1''}$, each connected component is a clique, and every clique consists of 1-tight vertices on a grid line perpendicular to the face; e.g., in Fig. 7, ℓ and ℓ' are such grid lines. Let $H \subseteq F_1''$ be an independent set such that exactly one vertex is picked up from every clique of $G_{F_1''}$. It is easy to see that, among MISs of the trellis, there is one that contains H as a subset. Intending such an MIS, we ignore the vertices in F_1'' and their unique solution neighbors. Let $D'' \subseteq D$ be a subset such that $D'' = \bigcup_{u \in H} N_S(u)$; e.g., in Fig. 7, $D'' = \{z,z'\}$. We can no longer choose the vertices in D''. Let $D' = D \setminus D''$.

The remaining task is to compute an MIS of the remaining part, say $G_{D' \cup F_1' \cup F_2}$. All the vertices in $D' \cup F_1' \cup F_2$ are on the face. At most one vertex is chosen from a grid line on the face, and there are $2n$ grid lines in all. It is the problem of finding a maximum matching in a bipartite graph like Fig. 8; a vertex is associated with a grid line on the face, and the bipartition of vertices

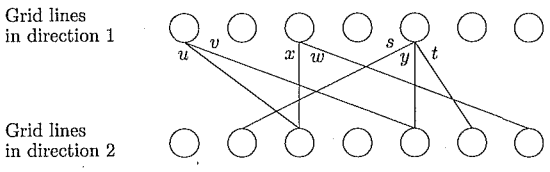

Figure 8: The bipartite graph in the proof of Lemma 6: Any matching corresponds to an independent set of the trellis. $\{x, y\}$ is a matching that consists of vertices in the current solution, and we see a larger matching such as $\{s, v, w\}$.

Algorithm 5 A Trellis-neighborhood search algorithm for a maximal solution

global variables: parameters $\pi, \#_{\text{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.
require: the parameters as a whole must represent a maximal solution.

```
1: function TR-NS
2:     for all d ∈ [3] and k ∈ [n] do
3:         D ←the set of solution vertices on the (d, k)-face
4:         I ←an MIS of the (d, k)-trellis
5:         if |I| > |D| then
6:             drop each vertex in D from the solution
7:             insert each vertex in I into the solution
8:             return "an improved solution is found"
9:         end if
10:    end for
11:    return "the solution is Trellis-maximal"
12: end function
```

is determined by the directions of grid lines. An edge joins two vertices whenever the trellis has a vertex on the intersecting point of the corresponding two grid lines. Let M be a maximum matching of this bipartite graph.

An MIS of the trellis is given by the union of H and M. It takes $O(n^2)$ time to recognize the vertex sets D, F_1 and F_2 and partitions $D = D' \cup D''$ and $F_1 = F_1' \cup F_1''$, to decide a subset $H \subseteq F_1''$, and then to construct the bipartite graph since the bipartite graph has $2n$ vertices and $O(n^2)$ edges. We need $O(n^{2.5})$ time to compute M (Hopcroft and Karp, 1973). □ □

Theorem 5 *Given a solution S, we can find an improved solution in the Trellis-neighborhood or conclude that it is Trellis-maximal in $O(n^{3.5})$ time.*

PROOF: There are $3n$ faces and thus $3n$ trellises. For a (d, k)-trellis, let D be the maximal subset of solution vertices on the trellis. We can identify whether there is an independent set I with $|D| < |I|$ or not in $O(n^{2.5})$ time from Lemma 6. When such D and I are found, we have an improved solution $(S \setminus D) \cup I$ by dropping D from S and then by inserting I into $S \setminus D$. This requires $O(n^2)$ time since $|D| \leq n$ and $|I| \leq 2n$. □ □

In Algorithm 5, we present a Trellis-neighborhood search algorithm. The function TR-NS outputs whether the current maximal solution is Trellis-maximal or not; if not, like the (p, ∞)-neighborhood search algorithms, the function searches for an improved solution and updates the parameters so that the improved solution is maintained. It then outputs "an improved solution is found."

As we did for $(1, \infty)$-neighborhood search algorithms in Sect. 3.2, we may improve the empirical efficiency by using 1-tight vertices as triggers. We claim that, if a (d, k)-trellis does not have a 1-tight vertex (i.e., $F_1 = \emptyset$), then any independent set I in the trellis satisfies $|I| \leq |D|$, that is, no improved solution is possible from the trellis. This is because, if $F_1 = \emptyset$, any vertex in the bipartite graph representation (corresponding to a grid line on the (d, k)-face) is either isolated

Algorithm 6 An alternative Trellis-neighborhood search algorithm

global variables: parameters $\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3$.
require: the parameters as a whole must represent a maximal solution.

```
1: function TR-NS*
2:     for i = 1, 2, ..., #_1 do
3:         u = (u_1, u_2, u_3) ← π^{-1}(#_sol + #_0 + i)        ▷ u is a 1-tight vertex
4:         for all d ∈ [3] do
5:             D ←the set of solution vertices on the (d, u_d)-face
6:             I ←an MIS of the (d, u_d)-trellis
7:             if |I| > |D| then
8:                 drop each vertex in D from the solution
9:                 insert each vertex in I into the solution
10:                return "an improved solution is found"
11:            end if
12:        end for
13:    end for
14:    return "the solution is Trellis-maximal"
15: end function
```

or matched with respect to the matching that corresponds to the current solution, and thus no augmenting path exists. Hence, we have only to search the trellises with $F_1 \neq \emptyset$. We summarize this version of Trellis-neighborhood search algorithm in Algorithm 6. In Sect. 6, we will see that the ILS algorithm with TR-NS* is more efficient than one with TR-NS.

5 Iterated Local Search (ILS) Algorithm

We present our ILS algorithm for the PLSE problem. The ILS algorithm iterates local searches until the computation time exceeds a given time limit, and then outputs the incumbent solution S^*, i.e., the best solution among those searched so far. Each local search begins with an initial solution and repeats a (p, ∞)-neighborhood search algorithm until it finds a p-maximal solution, say S. If S is not worse than the current S^* (i.e., $|S| \geq |S^*|$), then S^* is updated to S. The initial solution S_0 of the next local search is generated by "kicking" S^*.

This section is devoted to explaining how we generate S_0 from S^*. In Algorithm 7, we summarize the function GENERATEINITSOL that returns the parameter set for S_0 for the input parameter set for S^*. First we copy S^* to S_0, by making a duplicate of the parameter set for S^* (line 2). Then we forcibly insert k non-solution vertices into S_0. We choose the number k of inserted vertices with probability $1/2^k$ (line 3). Specifically, we repeat the following steps k times; we pick up a non-solution vertex u (lines 5 to 15), drop its solution neighbors from the solution (line 16), and insert u into the solution (line 17). If there appear free vertices, then one is chosen at random and inserted into the solution repeatedly until it becomes maximal (lines 18 to 21).

We choose k vertices from all the non-solution vertices, except the first one (see lines 5 to 12). We choose the first one by a special mechanism so that **(a)** trivial cycling is avoided and **(b)** the diversity of search is attained.

To realize (a), we choose the first vertex u from a special subset of $V \setminus S_0$. Let $S_0' \subseteq S_0$ be a subset of solution vertices that have a 1-tight neighbor;

$$S_0' = \{x \in S_0 \mid \exists d \in [3],\ \lambda_d(x) > 0\}.$$

We choose u not from $V \setminus S_0$ but from $N(S_0')$. (If S_0' is empty, then $N(S_0')$ is also empty. In this case we choose u from $V \setminus S_0$ instead.) Let us describe the motivation of using $N(S_0')$. See Fig. 9. In the upper and lower figures, u and u' are the non-solution vertices to be inserted, respectively, and $\{x, y\}$ and $\{x', y'\}$ are solution vertices to be dropped, respectively. The only difference of the two situations is the tightness of v and v'; the tightness of v in the upper figure is 1, whereas that of v' in the lower figure is 2. Observe that x is among S_0' since it has a 1-tight neighbor v, while x'

Algorithm 7 A function for generating the parameter set for the initial solution of the next local search

function GENERATEINITSOL($\pi^*, \#^*_{\mathrm{sol}}, \#^*_0, \#^*_1, \tau^*, \lambda^*_1, \lambda^*_2, \lambda^*_3, \rho^*_1, \rho^*_2, \rho^*_3$)
  $(\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3) \leftarrow (\pi^*, \#^*_{\mathrm{sol}}, \#^*_0, \#^*_1, \tau^*, \lambda^*_1, \lambda^*_2, \lambda^*_3, \rho^*_1, \rho^*_2, \rho^*_3)$ ▷ in what follows, we manipulate the solution represented by $(\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3)$
  $k \leftarrow$ a natural number chosen with probability $1/2^k$
  for $\kappa = 1, \ldots, k$ **do**
    if $\kappa = 1$ **then**
      $S'_0 \leftarrow \{\pi^{-1}(i) \mid i \in [\#_{\mathrm{sol}}],\ \exists d \in [3],\ \lambda_d(\pi^{-1}(i)) > 0\}$
      if $S'_0 \neq \emptyset$ **then**
        $P \leftarrow N(S'_0)$
      else
        $P \leftarrow \{\pi^{-1}(\#_{\mathrm{sol}} + 1), \ldots, \pi^{-1}(|V|)\}$
      end if
      $u \leftarrow$ the non-solution vertex in P that has been outside the solution for the longest time
    else
      $u \leftarrow$ an arbitrary non-solution vertex
    end if
    drop all solution neighbors of u from the solution
    insert u into the solution
    while $\#_0 > 0$ **do**
      $v \leftarrow$ an arbitrary free vertex
      insert v into the solution
    end while
  end for
  return $(\pi, \#_{\mathrm{sol}}, \#_0, \#_1, \tau, \lambda_1, \lambda_2, \lambda_3, \rho_1, \rho_2, \rho_3)$
end function

is not so. Then we have $u \in N(S'_0)$ and $u' \notin N(S'_0)$. In the upper figure, dropping u from S_0 and inserting x and y into $S_0 \setminus \{u\}$, we obtain a solution in the right side that is different from S_0; v is in the new solution since it became free when x was dropped. On the other hand, in the lower figure, we may encounter the cycling phenomenon in the subsequent local search; the right solution is not 1-maximal since we obtain an improved solution by a $(1, 2)$-swap that drops u' and inserts x' and y' into the solution. However, this improved solution is equivalent to the left solution.

We call this way of deciding the first inserted vertex the *Population Restricting Strategy* (*PRS*). In the next section, we will observe how this strategy is effective in comparison with the case in which the strategy is not activated.

To realize (b), we employ the *soft-tabu* approach that is utilized in Andrade et al.'s ILS for the MIS problem (Andrade et al., 2012). Specifically, among the non-solution vertices in $N(S'_0)$, we choose the one that has been outside the solution for the longest time.

6 Computational Study

In this section we demonstrate how the ILS with Trellis-neighborhood computes high-quality solutions efficiently through a computational study.

We conduct two experiments. In Sect. 6.1, we compare two variants of the ILS algorithms, TR-ILS and TR-ILS*. TR-ILS (resp., TR-ILS*) is the ILS algorithm such that, in its local search, the Trellis-neighborhood search algorithm in Algorithm 5 (resp., in Algorithm 6) is used to find a Trellis-maximal solution. We show that TR-ILS* is more likely to deliver a high-quality solution than TR-ILS. We also observe the effect of the PRS. Then in Sect. 6.2, we show that TR-ILS* outperforms other ILS variants (i.e., 1-ILS*, 2-ILS, 3-ILS) and two optimization softwares, IBM ILOG CPLEX and LOCALSOLVER.

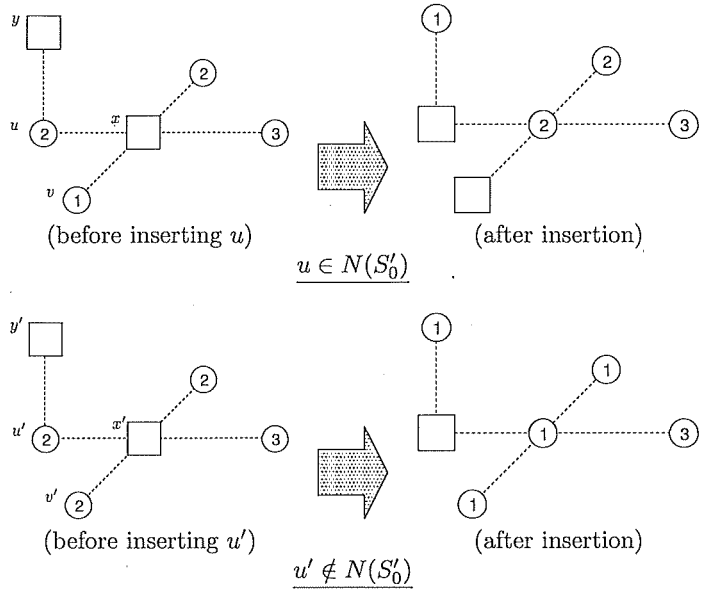

Figure 9: Situations in which a non-solution vertex is forcibly inserted: (upper) $u \in N(S'_0)$, (lower) $u' \notin N(S'_0)$

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 algorithms are implemented in C.

Benchmark instances are random PLSs. A PLS is parametrized by the order n and the ratio $r \in [0, 1]$ of pre-assigned symbols over the $n \times n$ grid. We take n from $\{50, 60, 70\}$ and r from $\{0.3, 0.4, \ldots, 0.8\}$. Given (n, r), we generate an instance by two well-known schemes in the literature, *quasigroup with holes* (*QWH*) and *quasigroup completion* (*QC*) (Barták, 2006; Gomes and Shmoys, 2002). Starting with an arbitrary Latin square, QWH generates a PLS by dropping $n^2 - \lfloor n^2 r \rfloor$ symbols from the grid so that $\lfloor n^2 r \rfloor$ symbols remain. On the other hand, starting from an empty assignment, QC repeats assigning a symbol to an empty cell randomly so that the resulting assignment is a PLS, until $\lfloor n^2 r \rfloor$ cells are assigned symbols. Note that a QWH instance always admits a complete Latin square as an optimal solution, whereas a QC instance does not. We use QWH instances to observe how often the algorithms can find an optimal solution and QC instances to observe how large the solutions found are.

We solve the PLSE problem by reducing it to the MIS problem. We show the averaged size of graph $G = (V, E)$ in Table 1. The graph sizes are not different by a significant margin between QWH and QC, and the average is taken over 200 QWH instances and 200 QC instances. A large graph is not necessarily hard in our case. It is known that the PLSE problem has easy-hard-easy phase transition (Gomes and Shmoys, 2002); an instance with an intermediate r (e.g., $0.5 \leq r \leq 0.7$) is harder in general.

For each instance, we generate an initial solution S_0 and feed it to the ILS algorithms or to the competitors. The S_0 is generated by a constructive algorithm named G5 in (Alidaee et al., 2008), which is a "look-ahead" minimum-degree greedy algorithm for the MIS problem. We confirmed in (Haraguchi, 2013) that G5 is the best among several simple constructive algorithms.

6.1 Comparison between Tr-ILS* and Tr-ILS.

For $n = 70$ and each $r \in \{0.3, 0.4, \ldots, 0.8\}$, we run TR-ILS* and TR-ILS on 200 QWH instances and 200 QC instances. We run one ILS algorithm 5 times for each instance, changing the seed of a pseudo random number. The time limit in each run is set to 10 seconds.

We show the result on QWH instances in Fig. 10. The horizontal axis indicates the pre-assigned

Table 1: Averaged size of the graph $G = (V, E)$: the column "density" indicates the ratio of $|E|$ to $|V|(|V|-1)/2$

n	r	$\lvert V\rvert$ ($\times 10^3$)	$\lvert E\rvert$ ($\times 10^3$)	density (%)
50	0.3	43.7	1600.7	0.17
	0.4	27.8	757.5	0.20
	0.5	16.4	315.2	0.23
	0.6	8.7	109.6	0.29
	0.7	3.9	29.1	0.38
	0.8	1.3	4.8	0.54
60	0.3	75.3	3312.4	0.12
	0.4	47.9	1561.9	0.14
	0.5	28.2	646.4	0.16
	0.6	14.8	222.8	0.20
	0.7	6.6	58.2	0.27
	0.8	2.2	9.5	0.39
70	0.3	119.2	6122.7	0.09
	0.4	75.8	2881.1	0.10
	0.5	44.5	1188.4	0.12
	0.6	23.3	406.9	0.15
	0.7	10.3	105.0	0.20
	0.8	3.4	16.8	0.29

ratio r. For each r, one ILS algorithm is run $200 \times 5 = 1000$ times. We show in the vertical axis how often the algorithm finds an optimal solution in the 1000 runs. To observe the effect of PRS, we run both ILS algorithms with PRS activated and with PRS inactivated. We also show the result on QC instances in Fig. 11. In this figure, the vertical axis indicates $|L| + |S|$, where L denotes the PLS given as an instance and S denotes the solution found. Note that $|L| = \lfloor n^2 r \rfloor$ is a constant for a given r. A bar represents the average of the medians over 200 instances, where the median is taken over 5 runs on one instance; an error bar represents the averages of maxima and minima over the 5 runs.

Generally, TR-ILS* with PRS ranks first, followed by TR-ILS with PRS, TR-ILS* with no PRS, and TR-ILS with no PRS.

QWH (Fig. 10): Clearly, we see this tendency for $r = 0.3$ to 0.6. When $r = 0.7$, the algorithms hardly find an optimal solution probably due to the problem hardness. When $r = 0.8$, TR-ILS* with PRS is slightly worse than TR-ILS with PRS, but the difference appears to be within a small error.

QC (Fig. 11): The tendency is shown in general not only for medians but also for maxima and minima. Furthermore, all medians of TR-ILS* with PRS are optimal from $r = 0.3$ to 0.5; the bar reaches $n^2 = 4900$.

Whether PRS is activated or not, TR-ILS* is better than TR-ILS. We introduced TR-NS* in Algorithm 6, aiming at avoiding vain neighborhood searches. It seems that TR-NS* achieves this goal to a large extent. As a numerical evidence, we show the averaged computation time of a single run of local search in Table 2. A local search in TR-ILS* is 2 to 4 times faster than a local search in TR-ILS.

We see that PRS contributes to the improvement of the performance in both ILS algorithms. In the next experiment, we activate PRS for all ILS algorithms.

6.2 Comparison between Tr-ILS* and Other Solvers

We compare TR-ILS* with other ILS variants and the competitors. For ILS variants, we take up 1-ILS*, 2-ILS and 3-ILS that use 1-NS* in Algorithm 3, 2-NS in Algorithm 4 and a $(3, \infty)$-neighborhood search algorithm (whose description is omitted in the paper), respectively. We

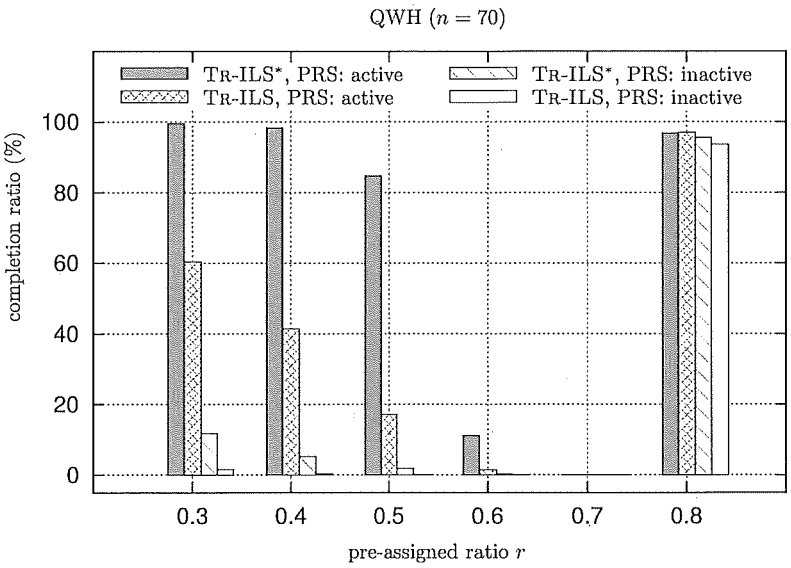

Figure 10: Comparison of completion ratios between TR-ILS* and TR-ILS (QWH, $n = 70$)

Table 2: Computation time (ms) of a single run of local search (QC, $n = 70$, PRS is active)

ILS algorithm	$r = 0.3$	0.4	0.5	0.6	0.7	0.8
TR-ILS*	8.29	4.50	2.03	0.77	0.45	0.29
TR-ILS	31.95	18.95	8.53	2.68	1.12	0.46
1-ILS*	5.00	2.85	1.38	0.58	0.29	0.13
2-ILS	8.14	4.52	2.04	0.98	0.51	0.23
3-ILS	150.73	85.63	39.41	12.24	3.89	1.22

employ 1-NS* instead of 1-NS in Algorithm 2 since we observe that the former yields better results in our preliminary experiments. We set the time limit of the ILS algorithms to 10 seconds.

For competitors, we employ two exact solvers and one heuristic solver. For the former, we employ the optimization solver for integer programming (CIP) and the one for constraint optimization (CCP) from IBM ILOG CPLEX ver. 12.6.1. It is easy to formulate the PLSE problem by these models (Gomes and Shmoys, 2002). For the latter, we employ LOCALSOLVER ver. 4.5 (LSOL), which is a general heuristic solver based on local search. Hopefully our ILS algorithms will outperform LSOL since ours is specialized to the PLSE problem, whereas LSOL is developed for general discrete optimization problems. All the parameters are set to default values except that, in CCP, `DefaultInferenceLevel` and `AllDiffInferenceLevel` are set to `extended`. The time limit of the competitors is set to 30 seconds, which is 3 times the time limit given to the ILS algorithms.

For each $(n, r) \in \{50, 60, 70\} \times \{0.3, \ldots, 0.8\}$, we generate 100 QWH instances and 100 QC instances. The same initial solution being given, each solver is run once on an instance. Table 3 shows how often the solvers find optimal solutions for QWH instances (in which the initial solutions are not optimal), and Table 4 shows the average of the improved size for QC instances. In both tables, a row corresponds to a pair (n, r), and a column corresponds to a solver. In particular, the 3rd column corresponds to G5, the initial solution generator. The column indicates how often the initial solution itself is an optimal solution in Table 3, while it indicates the initial solution size $|S_0|$ (plus $|L| = \lfloor n^2 r \rfloor$) in Table 4. A bold number indicates the largest value in a row.

The ILS algorithms outperform the competitors in most of the (n, r)'-s in both tables. We claim that TR-ILS* should be the best among the four ILS algorithms; 3-ILS is clearly inferior to others. The remaining three algorithms seem to be competitive, but TR-ILS* ranks first or second most frequently.

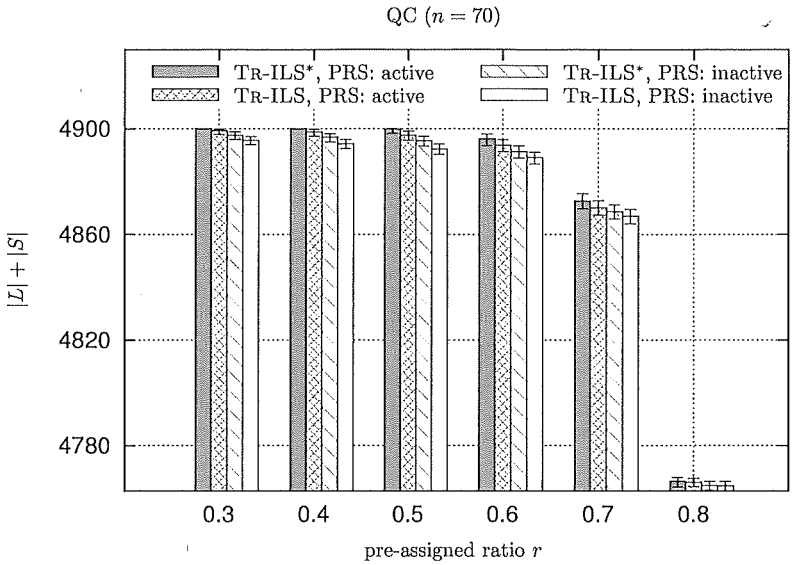

Figure 11: Comparison of the solution qualities between TR-ILS* and TR-ILS (QC, $n = 70$)

Table 3: Completion ratios (%) of the ILS algorithms and the competitors (QWH)

n	r	G5	TR-ILS*	1-ILS*	2-ILS	3-ILS	CIP	CCP	LSOL
50	0.3	9	**100**	**100**	**100**	95	0	93	1
	0.4	3	**100**	99	99	92	0	70	5
	0.5	0	**100**	96	96	83	0	12	6
	0.6	0	**36**	30	23	5	0	0	0
	0.7	0	0	0	0	0	0	0	0
	0.8	60	**100**	**100**	**100**	**100**	**100**	**100**	**100**
60	0.3	0	**100**	**100**	**100**	51	0	71	1
	0.4	0	**100**	96	99	52	0	22	0
	0.5	0	**95**	89	**95**	17	0	1	0
	0.6	0	**23**	16	12	0	0	0	0
	0.7	0	0	0	0	0	0	0	0
	0.8	0	99	98	**100**	99	**100**	**100**	99
70	0.3	0	99	**100**	**100**	19	0	34	0
	0.4	0	**98**	95	97	8	0	8	0
	0.5	0	84	82	**87**	0	0	0	0
	0.6	0	**10**	5	2	0	0	0	0
	0.7	0	0	0	0	0	0	0	0
	0.8	0	98	93	97	95	**100**	**100**	46

QWH (Table 3): TR-ILS* ranks first or second in all the 18 (n, r)'-s, and is among the first rank in 11 (n, r)'-s (except $r = 0.7$). We see that "under-constrained" instances with $r \leq 0.4$ and "over-constrained" instances with $r = 0.8$ are relatively easy since even G5 often finds an optimal solution ($n = 50$) and the competitors perform well on them. On the other hand, an instance with $r = 0.7$ is the hardest; no solver finds an optimal solution even when $n = 50$.

QC (Table 4): TR-ILS* ranks first or second in 17 (n, r)'-s except $(60, 0.8)$, and is among the first rank in 15 (n, r)'-s. It performs well especially on instances with $r \leq 0.7$. Concerning the competitors, CIP is good for "over-constrained" instances with $r = 0.8$ while CCP is suitable for "under-constrained" instances. In this maximization context, the heuristic solver LSOL performs relatively well, especially on hard instances with $r = 0.5$ to 0.7.

Table 4: Improved sizes brought by the ILS algorithms and the competitors (QC)

n	r	$\lvert L\rvert + \lvert S_0\rvert$	TR-ILS*	1-ILS*	2-ILS	3-ILS	CIP	CCP	LSOL
50	0.3	2496.03	**3.97**	**3.97**	3.95	3.93	0.00	3.84	0.32
	0.4	2493.78	**6.22**	6.20	**6.22**	6.08	0.00	4.24	0.87
	0.5	2488.52	**11.48**	11.37	11.43	10.73	0.00	1.40	4.44
	0.6	2476.21	**20.97**	20.02	20.09	18.46	0.00	2.66	13.00
	0.7	2442.21	**27.86**	27.26	27.57	25.56	4.19	8.83	21.24
	0.8	2382.07	12.07	12.07	12.04	12.02	**12.51**	6.03	11.60
60	0.3	3593.07	**6.93**	6.91	**6.93**	6.21	0.00	5.22	0.13
	0.4	3590.68	**9.32**	9.29	9.28	7.90	0.00	1.87	0.49
	0.5	3585.29	**14.65**	14.36	14.29	12.24	0.00	0.54	2.21
	0.6	3572.61	**24.06**	23.21	23.24	20.16	0.00	1.09	12.91
	0.7	3534.62	**37.50**	36.85	35.96	31.89	0.09	5.83	26.43
	0.8	3456.59	21.90	**22.00**	21.78	21.46	21.99	7.55	19.85
70	0.3	4890.20	**9.80**	9.78	9.78	7.12	0.00	3.55	0.05
	0.4	4887.73	**12.25**	12.23	**12.25**	8.67	0.00	0.63	0.25
	0.5	4881.09	**18.48**	18.32	18.35	12.88	0.00	0.08	1.81
	0.6	4868.21	**27.98**	27.09	26.72	20.31	0.00	0.53	9.56
	0.7	4829.65	**43.30**	42.76	41.32	34.73	0.00	2.29	30.06
	0.8	4731.35	34.56	**35.32**	34.46	32.58	30.09	6.38	29.82

Although 3-ILS deals with the largest neighborhood, the performance is the worst among the ILS algorithms. This must be due to its inefficiency. See Table 2 again. A local search in 3-ILS is about 10 to 40 times slower than those in the other three ILSs. Then 3-ILS may not iterate an enough number of local searches. The diversity of search must not be attained to a sufficient level.

The high performance of TR-ILS* is mainly due to the efficient implementation of its neighborhood search algorithm, TR-NS* in Algorithm 6. In Table 2 we see that a local search in TR-ILS* is at most only 2 times slower than one in 1-ILS*, and is competitive with one in 2-ILS although a Trellis-neighborhood is a superset of a (p, ∞)-neighborhood ($p \leq 2$) and, in the worst case analysis, the running times of TR-NS*, 1-NS* and 2-NS are bounded by $O(n^{3.5})$, $O(n^2)$ and $O(n^3)$, respectively.

7 Concluding Remarks

We have considered efficient local search algorithms for the PLSE problem. First, we presented (p, ∞)-neighborhood search algorithms for $p \in \{1, 2, 3\}$ running in $O(n^{p+1})$ time. We then introduced a new type of neighborhood, Trellis-neighborhood, and presented its neighborhood search algorithm that runs in $O(n^{3.5})$ time. We observed that TR-ILS*, an ILS algorithm with Trellis-neighborhood, outperforms not only other ILS variants but also IP and CP solvers from IBM ILOG CPLEX and LOCALSOLVER.

Our starting point is a reduction of PLSE to MIS. This enables us to apply the local search methodology of Andrade et al. (2012) to the PLSE problem. However, our study is not just a simple application. We have achieved the above result by making use of the graph structure peculiar to the problem. Specifically, the following are key properties of our graph:

- A vertex is regarded as a 3D integral point.
- The neighbors of a vertex are partitioned into $O(1)$ cliques and no two neighbors in different cliques are adjacent.

The idea in the paper may be extended to various problems. Here we describe two examples:

Colored Partial Latin Square Extension (C-PLSE): Suppose that each cell is colored by one of given n colors, say $c_1, \ldots, c_n$. Given such a colored grid, we say that a PLS *satisfies*

the color condition if, for any color $c \in \{c_1, \ldots, c_n\}$, each symbol in $\{1, 2, \ldots, n\}$ appears at most once in the cells having c. In this problem, given a PLS that satisfies the color condition with respect to a given colored grid, we are asked to find its largest extension.

Symmetric Partial Latin Square Extension (S-PLSE): A PLS L is *symmetric* if $(i, j, k) \in L$ implies $(j, i, k) \in L$. In this problem, we are asked to find a largest extension of a given symmetric PLS.

The two problems generalize some known problems; e.g., the C-PLSE problem contains the maximization version of completing Sudoku or Euler squares (Lewis, 2007; Rosenhouse and Taalman, 2012), and the S-PLSE problem contains the maximization version of completing single round-robin scheduling (Rasmussen and Trick, 2008). To extend our local search for these problems, we need to reconstruct the graph $G = (V, E)$ that is determined by the given PLS in the manner of Sect. 2. For the C-PLSE problem, we should connect any two vertices (i, j, k) and (i', j', k) by an edge if the cells (i, j) and (i', j') have the same color. For the S-PLSE problem, we should merge any vertex (i, j, k) $(i < j)$ with the vertex (j, i, k). Note that in the resulting graph the second property above does not hold any more. Recently, for the S-PLSE problem, we developed a (p, ∞)-neighborhood search algorithm $(p \leq 2)$ whose running time is $O(n^{p+1})$, extending the algorithms in this paper. The extension requires trickier arguments, and will be addressed in our future paper (Haraguchi).

Concerning experiments, we used random PLSs as benchmark instances and observed that ones with $r \in [0.6, 0.7]$ are harder than others. It is known in the literature that "balanced" instances are especially hard, that is, the number of empty cells is approximately the same over rows and columns (Gomes and Shmoys, 2002). Balanced instances are available from some websites; e.g., `http://www.cs.hbg.psu.edu/txn131/graphcoloring.html`. Unfortunately, our ILS hardly finds an optimal solution of a QWH balanced instance with $r \in [0.6, 0.7]$ even when $n = 30$.

Further comparison is left for future work. IBM ILOG CPLEX and LocalSolver are just two of existing optimization solvers. Also, they have many adjustable parameters for adapting themselves to instances. We should mention the approach by SAT. As mentioned in (Gomes and Shmoys, 2002), we do not consider that it is an effective strategy to solve PLSE by means of SAT. Let us describe our preliminary results; we solve the satisfiability problem on QWH instances by a CSP solver Sugar (ver. 2.2.1) (Tamura, 2014), where MiniSat (ver. 2.2.0) (Eén and Sörensson, 2010) is employed as the core SAT solver. Note that any QWH instance is satisfiable. This scheme decides the satisfiability (and thus finds an optimal solution) for about 50% of the instances within 30 seconds ($n = 40$ and $r \in \{0.3, 0.4, ..., 0.8\}$), while Tr-ILS finds an optimal solution for 78% of the instances within the same time limit. However, SAT technology is advancing year by year, and there are many sophisticated SAT/MaxSAT/MinSAT solvers (e.g., MaxSatz (Li et al., 2007), MinSatz (Li et al., 2012), and various participant solvers in The International SAT Competitions (`http://www.satcompetition.org/`)) and encoding techniques (e.g., MIS to MinSAT (Ignatiev et al., 2014)). It would be interesting to explore the best combination for solving PLSE.

Finally, from the viewpoint of approximation algorithms, we evaluate approximation factors of p-maximal and Trellis-maximal solutions and analyze the time complexities that are needed to compute them. In a maximization problem instance, a *ρ-approximate solution* ($\rho \in [0, 1]$) is a solution whose size is at least the factor ρ of the optimal size. Hajirasouliha et al. (2007) analyzed approximation factors of $(p, p+1)$-maximal solutions, using Hurkens and Schrijver's classical result on the general set packing problem (Hurkens and Schrijver, 1989). From this and Theorems 1, 2, 4 and 5, and since the solution size is at most n^2, we have the following theorem.

Theorem 6 *Any 1-maximal, 2-maximal, 3-maximal and Trellis-maximal solutions are* 1/2-, 5/9-, 3/5- *and* 5/9*-approximate solutions, respectively. Furthermore, these can be obtained in $O(n^4)$, $O(n^5)$, $O(n^6)$ and $O(n^{5.5})$ time by extending an arbitrary solution by means of the local search.*

Although the local search achieves the best approximation factor for the PLSE problem currently, no one has explored its efficient implementation in the literature. This paper resolves this issue to some degree.

Acknowledgements

We gratefully acknowledge very careful and detailed comments given by anonymous reviewers.

References

B. Alidaee, G. Kochenberger, and H Wang. Simple and fast surrogate constraint heuristics for the maximum independent set problem. *J. Heuristics*, 14:571–585, 2008.

D.V. Andrade, M.G.C. Resende, and R.F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18:525–547, 2012. The preliminary version appeared in Proc. 7th WEA (LNCS vol. 5038), pp. 220–234 (2008).

C. Ansótegui, A. Val, I. Dotú, C. Fernández, and F. Manyà. Modeling choices in quasigroup completion: SAT vs. CSP. In *Proc. National Conference on Artificial Intelligence*, pages 137–142, 2004.

G. Appa, D. Magos, and I. Mourtos. Searching for mutually orthogonal latin squares via integer and constraint programming. *European J. Operational Research*, 173(2):519–530, 2006a.

G. Appa, D. Magos, and I. Mourtos. A new class of facets for the latin square polytope. *Discrete Applied Mathematics*, 154(6):900–911, 2006b.

R.A. Barry and P.A. Humblet. Latin routers, design and implementation. *IEEE/OSA J. Lightwave Technology*, 11(5):891–899, 1993.

R. Barták. On generators of random quasigroup problems. In *Proc. CSCLP 2005*, pages 164–178, 2006.

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

C.J. Colbourn and J.H. Dinitz. *Handbook of Combinatorial Designs*. Chapman & Hall/CRC, 2nd edition, 2006.

B. Crawford, M. Aranda, C. Castro, and E. Monfroy. Using constraint programming to solve sudoku puzzles. In *Proc. ICCIT '08*, volume 2, pages 926–931, 2008.

B. Crawford, C. Castro, and E. Monfroy. Solving sudoku with constraint programming. In *Cutting-Edge Research Topics on Multiple Criteria Decision Making*, volume 35 of *Communications in Computer and Information Science*, pages 345–348, 2009.

M. Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *Proc. FOCS 2013*, pages 509–518, 2013.

N. Eén and N. Sörensson. The MiniSat Page (ver. 2.2.0). `http://minisat.se/Main.html`, 2010. Accessed 10 July 2015.

M. Fürer and H. Yu. Approximating the k-set packing problem by local improvements. In *Proc. ISCO 2014*, volume 8596 of *LNCS*, pages 408–420, 2014.

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Company, 1979.

C. Gomes, M. Sellmann, C. van Es, and H. van Es. The challenge of generating spatially balanced scientific experiment designs. In *Proc. CPAIOR 2004*, volume 3011 of *LNCS*, pages 387–394, 2004a.

C.P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. AAAI-97*, pages 221–227, 1997.

C.P. Gomes and D.B. Shmoys. Completing quasigroups or latin squares: a structured graph coloring problem. In *Proc. Computational Symposium on Graph Coloring and Generalizations*, 2002.

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):479–484, 2004b.

I. Hajirasouliha, H. Jowhari, R. Kumar, and R. Sundaram. On completing latin squares. In *Proc. STACS 2007*, volume 4393 of *LNCS*, pages 524–535, 2007.

K. Haraguchi. An efficient local search for the constrained symmetric latin square construction problem. in preparation.

K. Haraguchi. A constructive algorithm for partial latin square extension problem that solves hardest instances effectively. In *Recent Advances in Computational Optimization - Results of the Workshop on Computational Optimization WCO 2013 at FedCSIS 2013*, pages 67–84, 2013.

K. Haraguchi. An efficient local search for partial latin square extension problem. In *Proc. CPAIOR 2015*, volume 9075 of *LNCS*, pages 182–198, 2015.

K. Haraguchi and H. Ono. Approximability of latin square completion-type puzzles. In *Proc. FUN 2014*, volume 8496 of *LNCS*, pages 218–229, 2014.

J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2(4):225–231, 1973.

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 J. Discrete Mathematics*, 2(1):68–72, 1989.

A. Ignatiev, A. Morgado, and J. Marques-Silva. On reducing maximum independent set to minimum satisfiability. In *Proc.=SAT 2014*, volume 8561 of *LNCS*, pages 103–120. 2014.

J. Itoyanagi, H. Hashimoto, and M. Yagiura. A local search algorithm with large neighborhoods for the maximum weighted independent set problem. In *Proc. MIC 2011*, pages 191–200, 2011. The full paper is written in Japanese as a master thesis of the 1st author in Graduate School of Information Science, Nagoya University (2011).

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

T. Lambert, E. Monfroy, and F. Saubion. A generic framework for local search: Application to the sudoku problem. In *Proc. ICCS 2006*, volume 3991 of *LNCS*, pages 641–648, 2006.

R. Le Bras, A. Perrault, and C.P. Gomes. Polynomial time construction for spatially balanced latin squares. Technical report, Computing and Information Science Technical Reports, Cornell University, 2012. `http://hdl.handle.net/1813/28697`.

R. Lewis. Metaheuristics can solve sudoku puzzles. *J. Heuristics*, 13(4):387–401, 2007.

C.M. Li, F. Manyà, and J. Planes. New inference rules for max-sat. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

C.M. Li, Z. Zhu, F. Manyà, and L. Simon. Optimizing with minimum satisfiability. *Artificial Intelligence*, 190:32–44, 2012.

F. Ma and J. Zhang. Finding orthogonal latin squares using finite model searching tools. *Science China Information Sciences*, 56(3):1–9, 2013.

R. V. Rasmussen and M. A. Trick. Round robin scheduling - a survey. *European Journal of Operations Research*, 188(3):617–636, 2008.

J. Rosenhouse and L. Taalman. *Taking Sudoku Seriously.* Oxford University Press, 2012.

H. Simonis. Sudoku as a constraint problem. `http://4c.ucc.ie/~hsimonis/sudoku.pdf`, 2005. Accessed 10 July 2015.

C. Smith, C. Gomes, and C. Fernandez. Streamlining local search for spatially balanced latin squares. In *Proc. IJCAI'05*, pages 1539–1541, 2005.

R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes. A hybrid AC3-tabu search algorithm for solving sudoku puzzles. *Expert Systems with Applications*, 40(15):5817–5821, 2013.

N. Tamura. Sugar: a SAT-based Constraint Solver (ver. 2.2.1). `http://bach.istc.kobe-u.ac.jp/sugar/`, 2014. Accessed 10 July 2015.

H. Vieira Jr., S. Sanchez, K.H. Kienitz, and M.C.N. Belderrain. Generating and improving orthogonal designs by using mixed integer programming. *European J. Operational Research*, 215(3): 629–638, 2011.