Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 347–354

A Maximum Matching Based Heuristic Algorithm for Partial Latin Square Extension Problem

Kazuya Haraguchi*, Masaki Ishigaki and Akira Maruoka†
Department of Information Technology and Electronics
Faculty of Science and Engineering
Ishinomaki Senshu University
Ishinomaki, Miyagi 986-8580, Japan
Email: kzyhgc@gmail.com*, amaruoka@isenshu-u.ac.jp†

***Abstract*—A partial Latin square (PLS) is an 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 (PLSE) problem asks to find such a PLS that is a maximum extension of a given PLS. The PLSE problem is NP-hard, and in this paper, we propose a heuristic algorithm for this problem. To design a heuristic, we extend the previous $\frac{1}{2}$-approximation algorithm that utilizes the notion of maximum matching. We show the empirical effectiveness of the proposed algorithm through computational experiments. Specifically, the proposed algorithm delivers a better solution than the original one and local search. Besides, when computation time is limited due to an application reason, it delivers a better solution than IBM ILOG CPLEX, a state-of-the-art optimization solver, especially for large scale "hard" instances.**

I. Introduction

THROUGHOUT the paper, we consider the *partial Latin square extension* (PLSE) problem. Let n denote a natural number. Suppose that we are given an $n \times n$ grid. 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, each symbol should appear at most once. Given a PLS, the PLSE problem asks to find such a PLS that is a maximum extension of the given one in terms of the number of the filled cells. Specifically, we are asked to assign symbols additionally to as many empty cells as possible so that the Latin square condition is satisfied. The PLSE problem is NP-hard since its decision problem version is NP-complete [1]. The problem has various applications (e.g., scheduling, optical routers [2]) and was first introduced by Kumar, Russel and Sundaram [3], where they proposed several constant-factor approximation algorithms.

In this paper, we propose a heuristic algorithm for the PLSE problem. Assuming practical use, we aim to develop such a heuristic algorithm that delivers better solutions for more instances. We are interested just in empirical performance. We do not make theoretical analysis on approximation ratio, as previous studies for the PLSE problem do.

Let us explain the reason why we dare to propose such a heuristic algorithm. Researchers from discrete optimization or operations research may see that the problem can be formulated as a 0-1 integer programming (IP) problem. The global optimal solution can be found within practical time by the state-of-the-art optimization solvers (e.g., Gurobi optimizer [6], IBM ILOG CPLEX [7]) only when n is moderately small (e.g., no more than 30 in our experience). When n is larger, the solvers may not find even a feasible solution within practical time. For such large scale instances, a heuristic algorithm that delivers a good solution quickly is an alternative. For another application example, a heuristic algorithm may assist the work of the solvers. Many modern solvers admit us to input an initial solution. The solvers utilize the input solution as the first incumbent solution in the branch-and-bound tree. During the solution search, they may prune the solution subspaces where there is no hope of finding a better solution than the incumbent one, which may improve the efficiency of the search. Then the solver that is given a good initial solution is expected to find a better solution than the solver that is given no initial solution (or a poor initial solution) within the same time limit. An effective heuristic algorithm may help us generate a good initial solution.

Our heuristic algorithm is based on the $\frac{1}{2}$-approximation algorithm proposed in [3] that determines the assignment of symbols by solving the n relevant maximum matching problem instances iteratively. We try to improve its empirical performance by extending the model and by giving a reasonable scheme to decide the order in which the maximum matching problem instances are solved.

There are two approximation algorithms that have better approximation ratios than $\frac{1}{2}$. The best bound is $\frac{2}{3}-\varepsilon$, achieved by Hajirasouliha, Jowhari, Kumar and Sundaram [4], where ε is any positive constant. Their algorithm is based on local search, and the constant ε is automatically determined by the parameter on the neighborhood scale. The smaller ε we wish, the longer the running time becomes since we need to set the neighborhood scale large. For example, to beat the second best bound $1-\frac{1}{e}$, achieved by Gomes, Regis and Shmoys [5], the running time becomes $O(n^{26})$, which is surely polynomial but not practical. The second best algorithm [5] employs the linear programming (LP) relaxation approach. The idea is sophisticated but it is not easy to realize the mechanism to improve the empirical performance. On the other hand, the empirical performance of the maximum matching based

This work is supported by JSPS KAKENHI Grant Number 25870661.

978-1-4673-4471-5/$25.00

$\frac{1}{2}$-approximation algorithm can be improved by a smaller ingenuity. This is the highlight of the paper.

We show the empirical effectiveness of the proposed algorithm through computational experiments. Specifically, the proposed algorithm tends to deliver a better solution than the original one and the local search based approximation algorithm. Besides, when computation time is limited, as is often the case in practice, the proposed algorithm tends to deliver a better solution than IBM ILOG CPLEX for large scale "hard" instances.

The paper is organized as follows. In Section II, we prepare terminologies and notations. Then we present the proposed heuristic algorithm in Section III. We report the results of the computational experiments in Section IV, and then give concluding remarks in Section V.

II. Preliminaries

A. The PLSE Problem

First we introduce notations on the $n \times n$ grid. The grid consists of n^2 cells. Let us 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 denote the set of the n cells in the row i by R_i, and the set of the n cells in the column j by C_j, i.e.,

$$R_i = \{(i, 1), (i, 2), \ldots, (i, n)\} \text{ and}$$
$$C_j = \{(1, j), (2, j), \ldots, (n, j)\}.$$

We also denote the family of R_i'-s by $\mathcal{R}$, and the family of C_j'-s by $\mathcal{C}$, i.e.,

$$\mathcal{R} = \{R_1, R_2, \ldots, R_n\} \text{ and } \mathcal{C} = \{C_1, C_2, \ldots, C_n\}.$$

Clearly we have $\bigcup_{R_i \in \mathcal{R}} R_i = \bigcup_{C_j \in \mathcal{C}} C_j = [n]^2$.

Next we introduce notations on assignment of symbols to the grid. For simplicity, we represent the n symbols by the integers $1, 2, \ldots, n$ in the set $[n]$. We represent an assignment of symbols by an $n \times n$ array, denoted by L. For each cell (i, j), we denote the assigned symbol by $L_{i,j} \in [n] \cup \{0\}$, where $L_{i,j} = 0$ indicates that (i, j) is empty. We define the *domain of* L as $\mathrm{Dom}(L) = \{(i, j) \in [n]^2 \mid L_{i,j} \neq 0\}$. The cardinality $|\mathrm{Dom}(L)|$ equals to the number of the cells that are assigned symbols by L. For simplicity, we may write $|\mathrm{Dom}(L)|$ as $|L|$. An assignment L is an *extension* of L' if $\mathrm{Dom}(L) \supseteq \mathrm{Dom}(L')$ and $L_{i,j} = L'_{i,j}$ holds for any $(i, j) \in \mathrm{Dom}(L')$. When L is an extension of L', we write $L \geqslant L'$. We call L a *partial Latin square* (*PLS*) if, in each row and in each column, every symbol appears at most once. In particular, if all the cells are assigned symbols (i.e., $\mathrm{Dom}(L) = [n]^2$), then we simply call L a *Latin square* (*LS*). We call a PLS L *extensible* (resp., *blocked*) if there exists (resp., does not exist) a PLS $L' \geqslant L$ $(L' \neq L)$. Two PLSs L and L' are *compatible* if the following two conditions hold:

(i) For each (i, j), at least one of $L_{i,j} = 0$ and $L'_{i,j} = 0$ holds.

(ii) The assignment $L \oplus L'$ defined as follows is a PLS;

$$(L \oplus L')_{i,j} = \begin{cases} L_{i,j} & \text{if } L_{i,j} \neq 0 \text{ and } L'_{i,j} = 0, \\ L'_{i,j} & \text{if } L_{i,j} = 0 \text{ and } L'_{i,j} \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

An assignment of symbols can be also represented by a set of triples. Let T be a subset of $[n]^3$. The membership $(i, j, k) \in T$ represents that the symbol k is assigned to (i, j). In order to avoid duplicate assignments on the same cell, we assume that, for any different triples (i, j, k) and (i', j', k') in T, at least one of $i \neq i'$ and $j \neq j'$ holds. To convert the triple set representation into the array representation, we define the array $A(T)$ as follows;

$$A(T)_{i,j} = \begin{cases} k & \text{if there is } k \in [n] \text{ such that } (i, j, k) \in T, \\ 0 & \text{otherwise.} \end{cases}$$

For $(i, j, k), (i', j', k') \in T$, when at least two of $i \neq i'$, $j \neq j'$ and $k \neq k'$ hold, we say that they are compatible. Then T is a PLS if any two triples in T are compatible. Let us simplify some notations. When L and $A(T)$ are compatible, we may say that L and T are compatible, and use the expression $L \oplus T$ instead of $L \oplus A(T)$. When a singleton $T = \{(i, j, k)\}$ is compatible with L, we say that (i, j, k) (instead of $\{(i, j, k)\}$) is compatible with L.

We summarize the PLSE problem as follows.

The Partial Latin Square Extension (PLSE) Problem	
Input:	A PLS L_0.
Output:	A PLS $L \geqslant L_0$ that attains the maximum $\lvert L \rvert$.

A polynomial time algorithm for the PLSE problem is called a *ρ-approximation algorithm* if, for any input PLS L_0, it finds $L \geqslant L_0$ such that;

$$\frac{|L| - |L_0|}{|L^*| - |L_0|} \geq \rho,$$

where L^* denotes a global optimal solution. The bound ρ is called the *approximation ratio*.

B. Graph, Maximum Matching, Independent Set

An *undirected graph* (or simply a *graph*) $G = (V, E)$ consists of a set V of *nodes* and a set E of unordered pairs of nodes, where each element in E is called an *edge*. The *degree* of a node $v \in V$ is the number of the edges incident to v. A graph is *bipartite* when V can be partitioned into two disjoint nonempty sets, say V_1 and V_2, so that every edge joins a node in V_1 and a node in V_2. When we emphasize that G should be bipartite, we may write $G = (V_1 \cup V_2, E)$.

A *matching* E' is a subset of E such that no two edges in E' have a node in common. A *maximum matching* is such a matching that attains the largest cardinality. In particular, a maximum matching is called a *perfect matching* if it covers all the nodes in the graph. The size of maximum matching is called a *matching number*, and is denoted by $\nu(G)$. Suppose that the graph $G = (V, E)$ is bipartite. We can find a maximum matching in $O(\sqrt{|V|}|E|)$ time [8]. Note that maximum matching is not necessarily unique. Any edge e in E is classified into one of the following three classes with respect to how it appears in the possible maximum matchings:

Mandatory edge: e appears in every maximum matching.
Admissible edge: e appears in at least one (but not every) maximum matching.
Forbidden edge: e appears in no maximum matching.

The sets of mandatory, admissible and forbidden edges in G are denoted by $ME(G)$, $AE(G)$ and $FE(G)$, respectively. The edge set E of a bipartite graph G can be decomposed into three disjoint sets $ME(G)$, $AE(G)$ and $FE(G)$ by the Dulmage-Mendelsohn decomposition technique [9]. The computation time is dominated by finding a maximum matching, and thus the decomposition can be made in $O(\sqrt{|V|}|E|)$ time. The proposed algorithm repeatedly solves maximum matching problems on bipartite graphs such that $|V_1| = |V_2| = n$. We denote the upper bound on the computation time for one bipartite graph by τ_n, i.e., $\tau_n = O(n^{5/2})$.

Let $G = (V, E)$ be a general graph (not necessarily bipartite). An *independent set* is a subset of V such that any two nodes in the subset are not adjacent. A *maximum independent set* is such an independent set that attains the largest cardinality. The problem of finding a maximum independent set is known as NP-hard [10]. The proposed algorithm solves this problem for a certain purpose. To construct a nearly maximal independent set, we employ the $(\Delta + 2)/3$-approximation algorithm [11], where Δ denotes the maximum degree in the graph. The algorithm is a greedy method such that, starting from $S = \emptyset$, it inserts a node of the smallest degree into S and removes the inserted node and all the adjacent nodes (along with all the incident edges) from the graph. The process is repeated until all nodes are removed from the graph, and finally S is output. The computation time is linear in the numbers of nodes and edges [11].

III. A Heuristic Algorithm for the PLSE Problem

In this section, we propose a heuristic algorithm for the PLSE problem. The algorithm runs like a typical heuristic algorithm for a packing problem; we are given several containers (what we call *compatibility graphs*), each of which is given its capacity (matching number). Then we are asked to pack as many objects (matching edges) in the containers as possible, where packing an object in a container may decrease the capacities of other containers. To construct an approximate solution, we repeat choosing a certain container and packing the objects up to the capacity. The proposed algorithm is inspired by the $\frac{1}{2}$-approximation algorithm that was given in [3]. First we describe the $\frac{1}{2}$-approximation algorithm in Section III-A. Then in Section III-B, we present the proposed algorithm.

A. The Maximum Matching Based $\frac{1}{2}$-Approximation Algorithm

The $\frac{1}{2}$-approximation algorithm works as follows; starting from $L = L_0$, we repeat choosing a PLS L' that is compatible with L and updating $L \leftarrow L \oplus L'$ iteratively. For convenience, when L is updated to $L \oplus L'$, we say that L' is *added* to the PLS L. To decide L', the algorithm utilizes a maximum matching of what we call a compatibility graph. Given a PLS L, the compatibility graph is constructed for any symbol $k \in [n]$. The *compatibility graph for the symbol* k is denoted by $G_L^{\mathrm{symb},k} = (\mathcal{R} \cup \mathcal{C}, E_L^{\mathrm{symb},k})$, where $\mathcal{R}$ and $\mathcal{C}$ are the node sets and $E_L^{\mathrm{symb},k}$ is the edge set such that;

$$E_L^{\mathrm{symb},k} = \big\{\{R_i, C_j\} \subseteq \mathcal{R} \cup \mathcal{C} \mid (i,j,k) \text{ is compatible with } L\big\}.$$

Let $M \subseteq E_L^{\mathrm{symb},k}$ denote any matching of $G_L^{\mathrm{symb},k}$. We define the triple set $T(M)$ as follows;

$$T(M) = \big\{(i,j,k) \mid \{R_i, C_j\} \in M\big\}.$$

Any two triples (i,j,k) and (i',j',k') in $T(M)$ satisfy $i \neq i'$ and $j \neq j'$ since M is a matching of $G_L^{\mathrm{symb},k}$. Thus $T(M)$ is a PLS. Each $(i,j,k) \in T(M)$ is compatible with L from the definition of $E_L^{\mathrm{symb},k}$. Then $T(M)$ and L are compatible.

In the order $k = 1, 2, \ldots, n$, the algorithm finds a maximum matching M^* of $G_L^{\mathrm{symb},k}$ and adds $T(M^*)$ to L iteratively. We show the algorithm in Algorithm 1. Naïvely implemented, the algorithm runs in $O(n\tau_n) = O(n^{7/2})$ time.

Algorithm 1 The $\frac{1}{2}$-approximation algorithm given in [3]

```
L ← L_0
for k = 1, 2, ..., n do
    M* ← a maximum matching of G_L^{symb,k}
    L ← L ⊕ T(M*)
end for
output L
```

B. The Proposed Algorithm

Let us describe our idea for how we improve the empirical performance of the above approximation algorithm. We have denoted the available symbols by integers $1, 2, \ldots, n$ only for convenience. Their numerical order does not have any significant meaning, whereas the above algorithm chooses k in that order. Thus there is room for developing a smart criterion to choose a compatibility graph. (The approximation ratio is still guaranteed even if we choose k arbitrarily; see the proof in Section 5.2 of [3] for detail.)

Besides, the compatibility graphs for the symbols are not the only "containers" in which we can "pack" the maximum matching to increase the objective value. Motivated by the fact that the dimensions of PLS in the triple set representation are symmetric, we introduce the compatibility graph also for each row and for each column. The *compatibility graph for the row* i is denoted by $G_L^{\mathrm{row},i} = (R_i \cup [n], E_L^{\mathrm{row},i})$, where R_i and $[n]$ denote the node sets and $E_L^{\mathrm{row},i}$ is the edge set such that;

$$E_L^{\mathrm{row},i} = \big\{\{(i,j),k\} \subseteq R_i \cup [n] \mid (i,j,k) \text{ is compatible with } L\big\}.$$

Similarly, the *compatibility graph for the column* j is denoted by $G_L^{\mathrm{col},j} = (C_j \cup [n], E_L^{\mathrm{col},j})$, where C_j and $[n]$ denote the

node sets and $E_L^{\mathrm{col},j}$ is the edge set such that;

$$E_L^{\mathrm{col},j} = \big\{\{(i,j),k\} \subseteq C_j \cup [n] \mid (i,j,k) \text{ is compatible with } L\big\}.$$

We show an example of the compatibility graphs in Fig. 1. Let us denote any matching of $G_L^{\mathrm{row},i}$ or $G_L^{\mathrm{col},j}$ by M. We define the triple set $T(M)$ as follows;

$$T(M) = \big\{(i,j,k) \mid \{(i,j),k\} \in M\big\}.$$

Analogously with the case of $G_L^{\mathrm{symb},k}$, the triple set $T(M)$ is a PLS and compatible with L. Thus $T(M)$ can be added to L.

Finally, given a PLS L, we have $3n$ compatibility graphs. Using all of them as containers, we expect to obtain a better solution than the case when we use only the n compatibility graphs for symbols. We denote the set of the n compatibility graphs for symbols by $\mathcal{G}_L^{\mathrm{symb}} = \{G_L^{\mathrm{symb},1}, \ldots, G_L^{\mathrm{symb},n}\}$, the set of the n compatibility graphs for rows by $\mathcal{G}_L^{\mathrm{row}} = \{G_L^{\mathrm{row},1}, \ldots, G_L^{\mathrm{row},n}\}$ and the set of the n compatibility graphs for columns by $\mathcal{G}_L^{\mathrm{col}} = \{G_L^{\mathrm{col},1}, \ldots, G_L^{\mathrm{col},n}\}$. We also denote the union of these graph sets by $\mathcal{G}_L = \mathcal{G}_L^{\mathrm{symb}} \cup \mathcal{G}_L^{\mathrm{row}} \cup \mathcal{G}_L^{\mathrm{col}}$.

Let us introduce the criterion by which we choose one of the $3n$ compatibility graphs. Recall that any edge corresponds to a triple $(i,j,k) \in [n]^3$. When $L_{i,j} = k$, no edge corresponding to (i,j,k) appears in any compatibility graph. For any compatibility graph $G_L \in \mathcal{G}_L$, we denote by $\rho(G_L)$ the number of node pairs that do not appear as edges due to $L_{i,j} = k$. For example, when G_L is the compatibility graph $G_L^{\mathrm{symb},k}$ for the symbol k, $\rho(G_L^{\mathrm{symb},k})$ is defined as follows;

$$\rho(G_L^{\mathrm{symb},k}) = \big|\big\{\{R_i, C_j\} \subseteq \mathcal{R} \cup \mathcal{C} \mid L_{i,j} = k\big\}\big|.$$

In other words, $\rho(G_L^{\mathrm{symb},k})$ indicates the number of cells to which L already assigns the symbol k. Analogously, $\rho(G_L^{\mathrm{row},i})$ and $\rho(G_L^{\mathrm{col},j})$ indicate the numbers of cells in the row i and in the column j to which L assigns certain symbols, respectively. Now we define the criterion function $f(G_L)$ as follows.

$$f(G_L) = \begin{cases} \nu(G_L) + \rho(G_L) & \text{if } \nu(G_L) > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Basically, our heuristic algorithm runs as follows; starting from $L = L_0$, the algorithm chooses the compatibility graph $G_L \in \mathcal{G}_L$ such that $f(G_L)$ is the largest. Finding a maximum matching M^* of the chosen graph, the algorithm updates the solution by $L \leftarrow L \oplus T(M^*)$. The above operation is repeated while L is extensible. The repetition is at most $3n$ times; suppose that a compatibility graph G_L^s is chosen at a certain step of the algorithm, where s denotes the superscript of the compatibility graph. Let us denote the updated PLS by $L' = L \oplus T(M^*)$. For any extension $L'' \geqslant L'$, there is no edge in the compatibility graph $G_{L''}^s$. Then we have $\nu(G_{L''}^s) = 0$ and $f(G_{L''}^s) = 0$. Thus $G_{L''}^s$ is never chosen in the rest execution of the algorithm. Finally, when $f(G_L) = 0$ holds for all the $3n$ G_L'-s, L is blocked and the algorithm halts.

The algorithm utilizes the function f in (1) as the criterion to choose a compatibility graph. It may prefer a compatibility graph G_L such that more triples can be added (i.e., larger $\nu(G_L)$) and/or more cells are already assigned symbols (i.e., larger $\rho(G_L)$). The only matching number $\nu(G_L)$ appears to be a more natural criterion. However, it did not yield good results in our preliminary experiments.

We summarize the heuristic algorithm in Algorithm 2. The structure is as above stated, but in Line 3, we introduce the subroutine named COLLECTME for further improvement. The subroutine constructs a PLS that is compatible with L, from the triples of mandatory edges over the $3n$ compatibility graphs. Then it adds the PLS to L, before adding the triple set of a maximum matching of a certain compatibility graph to L.

Let us describe the motivation. Recall that a maximum matching is not necessarily unique in a graph. For example, given the PLS L in Fig. 1, there are 4 maximum matchings in the compatibility graph $G_L^{\mathrm{symb},1}$. We denote these 4 maximum matchings by $M_1^*, \ldots, M_4^*$, where we define;

$$M_1^* = \big\{\{R_1, C_4\}, \{R_2, C_1\}, \{R_3, C_2\}, \{R_4, C_3\}\big\},$$
$$M_2^* = \big\{\{R_1, C_2\}, \{R_2, C_1\}, \{R_3, C_4\}, \{R_4, C_3\}\big\},$$
$$M_3^* = \big\{\{R_1, C_2\}, \{R_2, C_3\}, \{R_3, C_4\}, \{R_4, C_1\}\big\},$$
$$M_4^* = \big\{\{R_1, C_4\}, \{R_2, C_3\}, \{R_3, C_2\}, \{R_4, C_1\}\big\}.$$

The greedy method would choose $G_L^{\mathrm{symb},1}$ for the container since $f(G_L^{\mathrm{symb},1})$ is the largest (which is 4) among all the compatibility graphs. Then the maximum matching algorithm finds arbitrary one M_x^* $(x = 1, \ldots, 4)$ and $T(M_x^*)$ is added to L. No matter which $T(M_x^*)$ is added, the symbol 1 is assigned to exactly 4 cells. Then the symbols 2 and 3 are assigned to the remaining empty cells, and the final solution is obtained. How many cells are filled in the final solution depends on which $T(M_x^*)$ is added. More precisely, when either $T(M_1^*)$ or $T(M_2^*)$ is added, the final solution becomes worse than the case when either $T(M_3^*)$ or $T(M_4^*)$ is added. To analyze the reason, see Fig. 2 for the final solutions that are obtained by extending $L \oplus T(M_1^*)$, $\ldots$, $L \oplus T(M_4^*)$. For $L \oplus T(M_3^*)$, there is another final solution such that the symbol 2 is assigned not to $(4,3)$ but to $(4,4)$, but it does not matter since we discuss how many cells are filled. We claim that the problem should come from the number of the bold cells occupied by the symbol 1. The bold cells in each grid correspond to the mandatory edges in $G_L^{\mathrm{symb},2}$ and $G_L^{\mathrm{symb},3}$, that is, $\{R_3, C_4\}, \{R_4, C_3\} \in E_L^{\mathrm{symb},2}$ and $\{R_1, C_4\}, \{R_2, C_1\} \in E_L^{\mathrm{symb},3}$; see also these graphs in Fig. 1. The matching number $\nu(G_L^{\mathrm{symb},k})$ gives an upper bound on the number of cells that the symbol k is assignable. Occupying bold cells by the symbol 1 may diminish the matching numbers of $G_L^{\mathrm{symb},2}$ and $G_L^{\mathrm{symb},3}$. When $T(M_1^*)$ or $T(M_2^*)$ is added, 3 out of the 4 bold cells are occupied by the symbol 1, while only 1 bold cell is occupied when $T(M_3^*)$ or $T(M_4^*)$ is added.

Based on the above, we consider that mandatory edges should be treated with a greater care in order to approach a

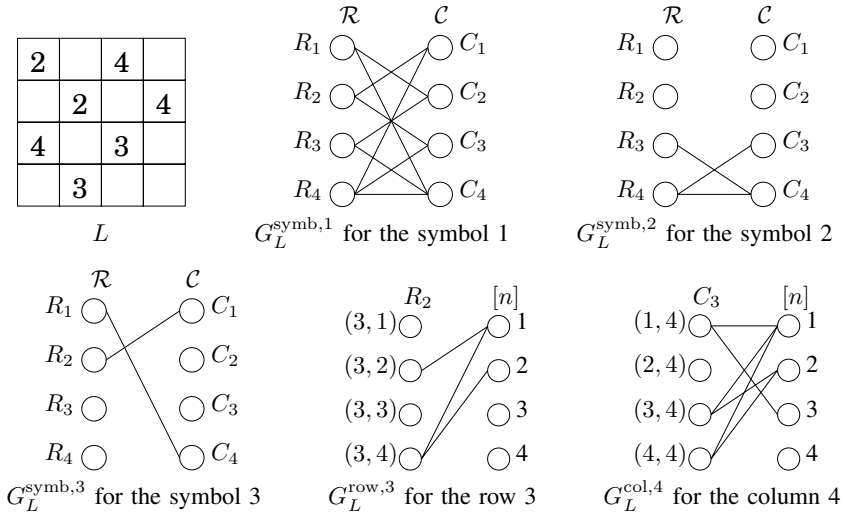

Fig. 1. An example of compatibility graphs for a PLS L

Algorithm 2 The proposed heuristic algorithm for the PLSE problem

```
1: L ← L_0
2: while L is extensible do
3:     L ← CollectME(L)
4:     construct the set G_L of the compatibility graphs for the PLS L
5:     G_L ← a compatibility graph in G_L such that f(G_L) in (1) is the largest
6:     M* ← a maximum matching of G_L
7:     L ← L ⊕ T(M*)
8: end while
9: output L
```

better solution. Our idea is to add as many triples coming from mandatory edges to L as possible. Let ME_L denote the set of all the mandatory edges over the $3n$ compatibility graphs in $\mathcal{G}_L$. We would like to add all the triples in $T(ME_L)$ if possible, but we cannot always do so since there are incompatible triples in $T(ME_L)$ in general. Let us define the graph $H_L = (T(ME_L), F_L)$ such that the triple set $T(ME_L)$ is the node set and the edge set F_L is defined as follows;

$$F_L = \big\{\{(i,j,k),(i',j',k')\} \subseteq T(ME_L) \mid (i,j,k) \text{ and } (i',j',k') \text{ are incompatible}\big\}.$$

Clearly, any independent set of H_L is a PLS and compatible with L. This motivates us to solve the maximum independent set problem on H_L. Since the problem is NP-hard, we find a nearly maximum solution by the greedy algorithm [11] that we described in Section II-B.

We summarize the subroutine CollectME in Algorithm 3. The subroutine repeats enumerating all mandatory edges over the $3n$ compatibility graphs, computing a nearly maximum independent set, and adding the independent set to L until there is no mandatory edge in any compatibility graph.

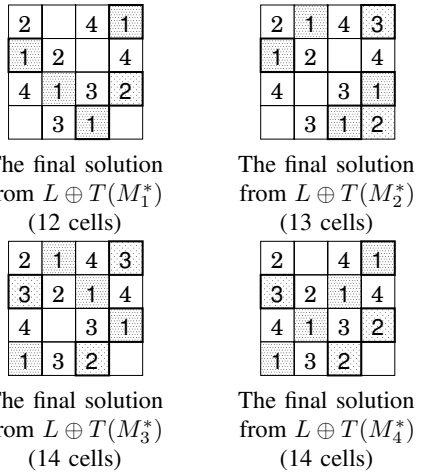

Fig. 2. The final solutions obtained by extending $L \oplus M_1^*, \ldots, L \oplus M_4^*$: the number of the filled cells is indicated in the parentheses

Algorithm 3 The subroutine CollectME

```
1: loop
2:     construct the set G_L of compatibility graphs
3:     ME_L ← the set of the mandatory edges over the 3n compatibility graphs in G_L
4:     if ME_L = ∅ then
5:         break the loop
6:     end if
7:     construct the graph H_L = (T(ME_L), F_L)
8:     I ← an independent set in the graph H_L
9:     L ← L ⊕ I
10: end loop
11: return L
```

C. Computational Complexity

We claim that the proposed algorithm should run in $O(n^{11/2})$ time. The algorithm consists of the main routine in Algorithm 2 and the subroutine COLLECTME in Algorithm 3. In the main routine, the most time-consuming task is to find maximum matchings of all the compatibility graphs. Since the iteration times of the main routine is at most $3n$, the running time is $3n \times 3n \times \tau_n = O(n^{9/2})$.

The iteration times of the subroutine amounts to $O(n^2)$ over the whole execution of the algorithm since there are at most $3n^2$ mandatory edges; there are $3n$ compatibility graphs, and in each graph, there are at most n mandatory edges. The most time-consuming task is not the greedy algorithm for the maximum independent set problem but to find maximum matchings of all the compatibility graphs; one can see that the former takes $O(n^3)$ time, whereas the latter takes $O(n^{7/2})$ time. Then the running time is $O(n^2) \times O(n^{7/2}) = O(n^{11/2})$.

IV. COMPUTATIONAL EXPERIMENTS

In this section, we present some experimental results to show how the proposed algorithm is effective in practice. Specifically, we show that the proposed algorithm tends to deliver a good solution quickly in comparison with other approximation algorithms. We also show that, when computation time is limited, the proposed algorithm tends to deliver a better solution than IBM ILOG CPLEX [7], a state-of-the-art optimization solver.

A. Experimental Settings

We describe how to generate problem instances. This issue has been studied in the field of *constraint programming*. In our experiments, we generate PLSE instances based on the QCP (quasigroup completion problem) framework that was discussed in [12]. An instance is characterized by two parameters. One is a natural number n, the side length of the grid, and the other is p $(0 \leq p \leq 1)$, which is the ratio of the number of pre-assigned cells over n^2. The QCP framework starts with the empty $n \times n$ grid and assigns a random symbol to a random empty cell so that the resulting assignment is PLS. The assignment is repeated until $\lfloor pn^2 \rfloor$ cells are assigned symbols. In the context of the decision problem, when the pre-assigned ratio p is small (resp., large), an instance is more (resp., less) likely to be satisfiable since there are less (resp., more) constraints. Many papers have reported the easy-hard-easy phase transition with respect to p (e.g., [13]); the instances are less computationally tractable when p is intermediate, e.g., $0.4 \leq p \leq 0.7$.

We refer to the proposed algorithm as OURS. For comparison, we will consider the algorithm that does not call the subroutine COLLECTME. We refer to this version of the algorithm to OURS-NOSUB. We also use 4 methods: MATCHING, LS, CPMATH and CPCP. The point is that MATCHING and LS are approximation algorithms that do not necessarily provide a global optimal solution, whereas CPMATH and CPCP are such softwares that retain exact algorithms.

a) MATCHING: The $\frac{1}{2}$-approximation algorithm [3] that forms the basis of the proposed algorithm and was explained in Section III-A.

b) LS: The $\frac{2}{3} - \varepsilon$ approximation algorithm [4] based on local search that achieves the best approximation ratio among those studied so far. Let T_0 denote the triple set representation of the given PLS L_0. Suppose that a non-negative number r is given. For any set $T \subseteq [n]^3$ of triples such that $T \geqslant T_0$, we denote the *neighborhood of* T by $\mathcal{N}_r(T)$ that is defined as follows;

$$\mathcal{N}_r(T) = \{T' = (T \setminus S) \cup S' \mid S \subseteq T \setminus T_0,\ |S| \leq r,\ S' \subseteq [n]^3,\ |S'| = |S| + 1,\ T' \text{ is a PLS}\}.$$

We call r the *radius* of neighborhood. LS starts with $T = T_0$ and iterates choosing a solution $T' \in \mathcal{N}_r(T)$ and updating $T \leftarrow T'$ until no solution exists in the neighborhood, i.e., $\mathcal{N}_r(T) = \emptyset$. The larger r is, the better the approximation ratio is, but at the same time, the more the computation time may become. Although $r \geq 7$ defeats the second best bound $1 - \frac{1}{e}$ given in [5], the time bound becomes $O(n^{26})$ in naïve implementation $(r = 7)$, which is not practical. We use $r \leq 4$ since any larger r was too time consuming in our preliminary experiments.

c) CPMATH: IBM ILOG CPLEX Optimizer (version 12.4) that solves the PLSE problem by means of mathematical programming. The PLSE problem can be formulated as a 0-1 integer programming problem. See [3] for the formulation. We set the time limit parameter to 6.0×10^2 seconds, i.e., if the computation time exceeds 6.0×10^2 seconds, the computation is terminated and the best solution found so far is output. We set all the other parameters to default values.

d) CPCP: IBM ILOG CPLEX CP Optimizer (version 12.4) that solves the PLSE problem by means of constraint programing. We formulate the PLSE problem as a constraint optimization problem as follows.

$$\begin{array}{ll} \text{maximize} & |L| \\ \text{subject to} & \forall i \in [n],\ \text{all-different\_except\_0}(L_{i,1}, \ldots, L_{i,n}), \\ & \forall j \in [n],\ \text{all-different\_except\_0}(L_{1,j}, \ldots, L_{n,j}), \\ & \forall (i,j) \in [n]^2,\ (L_0)_{i,j} \neq 0 \Rightarrow L_{i,j} = (L_0)_{i,j}, \\ & \forall (i,j) \in [n]^2,\ L_{i,j} \in [n] \cup \{0\}. \end{array}$$

In the above formulation, the all-different_except_0 constraint [14] is a special case of the typical all-different constraint, requiring that the variables should take all-different values except the variables assigned 0. We set the level of default inference (`DefaultInferenceLevel`) and the level of all-different inference (`AllDiffInferenceLevel`) to `extended`, i.e., the most sophisticated constraint propagation technique is used. We set the time limit parameter to 6.0×10^2 seconds. We set all the remaining parameters to default values.

All the experiments are conducted by our PC that carries 2.80 GHz CPU and 4GB main memory.

TABLE I
THE NUMBER OF WINS, TIES AND LOSTS OF OURS AGAINST OURS-NOSUB AND MATCHING OVER THE 1000 INSTANCES

			the pre-assigned ratio p			
			0.2	0.4	0.6	0.8
$n = 10$	vs. OURS-NOSUB	(win)	34	273	136	17
		(tie)	957	577	376	976
		(lost)	9	150	488	7
	vs. MATCHING	(win)	744	966	633	190
		(tie)	247	24	217	804
		(lost)	9	10	150	6
$n = 80$	vs. OURS-NOSUB	(win)	809	915	913	809
		(tie)	73	23	20	31
		(lost)	118	62	67	160
	vs. MATCHING	(win)	998	1000	1000	999
		(tie)	1	0	0	1
		(lost)	1	0	0	0

TABLE II
THE NUMBER OF WINS, TIES AND LOSTS OF OURS AGAINST 10 LS'-S OVER THE 100 INSTANCES

vs. LS		the pre-assigned ratio p						
		0.2	0.3	0.4	0.5	0.6	0.7	0.8
$n = 10$	(win)	570	815	839	412	145	32	0
$r = 4$	(tie)	413	176	116	218	281	689	962
	(lost)	17	9	45	370	574	279	38
$n = 20$	(win)	920	975	994	999	796	157	88
$r = 3$	(tie)	61	16	4	1	76	143	508
	(lost)	19	9	2	0	128	700	404
$n = 30$	(win)	1000	1000	1000	1000	1000	774	283
$r = 2$	(tie)	0	0	0	0	0	69	173
	(lost)	0	0	0	0	0	157	544

B. Results

First, we compare OURS with OURS-NOSUB and MATCHING. We generate 1000 instances for each pair (n, p) such that $n \in \{10, 20, \ldots, 80\}$ and $p \in \{0.1, 0.2, \ldots, 0.9\}$. We solve all the generated instances by each algorithm and compare the algorithms in terms of $|L|$. We show the typical results in Table I. The table shows the numbers of wins, ties and losts of OURS against the other algorithms over the 1000 instances. When n is small (i.e., $n = 10$), OURS is competitive with OURS-NOSUB in general, except that it is rather worse for $p = 0.6$. OURS outperforms MATCHING for smaller p, and for larger p, they are rather competitive. When n is larger, OURS is more likely to outperform the rest two algorithms. In particular, when $n = 80$, OURS wins over OURS-NOSUB in more than 80% of the instances and wins over MATCHING in almost all the instances. Based on these, we claim that the subroutine COLLECTME should play a significant role in improving the solution especially when n is large. We also claim that OURS should be superior to MATCHING, the original approximation algorithm. Let us discuss the computation time. MATCHING takes less than 1 seconds in all the instances from $n = 10$ to 80. Given the side length n, OURS takes more computation time for the instances that are generated by smaller pre-assigned ratio p. For every $n = 10, 20, \ldots, 80$, the average computation time for $p = 0.1$ is twice to three times of that for $p = 0.9$ approximately. For example, when $n = 10$ (resp., 80), the average computation time for $p = 0.1$ is 7.7×10^{-3} (resp., 2.0×10^{1}) seconds, whereas that for $p = 0.9$ is 4.1×10^{-3} (resp., 7.5×10^{0}) seconds. We consider the reason as follows; when p is smaller, there are more edges in the compatibility graphs in the earlier steps of the algorithm. Then it should take more time to compute a maximum matching since the algorithm runs in $O(\sqrt{|V|}|E|)$ time, which is proportional to the number of edges, while $|V| = 2n$ holds for any compatibility graph. The current implementation of OURS is rather naïve, and we believe that it is possible to improve the computation time to some extent by devising the data structure. This is left for future work.

Next, we compare OURS with LS. This time we generate 100 instances for given n and p. We solve each instance by OURS once, whereas we solve it by LS 10 times; in our implementation, LS proceeds to a solution randomly chosen from the neighborhood. Changing the seed of pseudo random numbers, we solve the instance by executing LS 10 times. We examine the $100 \times 10 = 1000$ cases to count the number of wins, ties and losts of OURS. We show the results in Table II, along with the values of the radius r. We describe the reason why we set r to the indicated values. The computation time of LS is affected by r significantly. LS is just a heuristic, and its computation time should be shorter than the time needed for the optimization solvers (i.e., CPMATH or CPCP) to find global optimum solutions. We set the radius r based on this philosophy. For example, when $n = 30$ and $p = 0.2$, the solvers find global optimum solutions in only 1.0×10^{1} seconds for any instance. However, when $r = 3$, LS takes at least 3.0×10^{2} seconds until it outputs a local optimal solution, which we infer from our preliminary experiments. Thus we set $r = 2$ even though the average computation time is still 2.4×10^{1} seconds. Note that OURS is much faster than LS; it takes less than 1 second to solve any instance. As expected from Table II, OURS should outperform LS in this perspective for $n \geq 30$ and $p \leq 0.7$. Note that this range includes "hard" instances in the context of the phase-transition.

Finally, we compare OURS with CPMATH and CPCP. When n gets larger, the solvers are less likely to find global optimum solutions in practical time. To illustrate this, in Table III, we show the number of instances such that the solvers can find global optimum solutions in 6.0×10^{2} seconds. In this experiment, we generate 100 instances for each (n, p). It is shown that, when p is smaller (resp., larger), CPCP finds global optimal solutions in more (resp., less) instances than CPMATH. When p is intermediate ($p = 0.6$ and 0.7 in particular), the solvers hardly find global optimum solutions. When the solvers cannot find a global optimum solution within the time limit, they output the best solution among those searched. In such cases, OURS yields better solutions than the solvers although the computation time is much shorter. We illustrate this for $n = 60$ in Fig. 3. The figure shows the average of $|L|$ (vertical axis) with respect to the change of p (horizontal axis). OURS+CPCP represents CPCP that is given an incumbent solution generated by OURS. In the

TABLE III
THE NUMBER OF INSTANCES THAT THE OPTIMIZATION SOLVERS CAN FIND A GLOBAL OPTIMAL SOLUTION WITHIN 6.0×10^2 SECONDS

		the pre-assigned ratio p						
		0.2	0.3	0.4	0.5	0.6	0.7	0.8
$n = 40$	CPMATH	0	1	3	71	7	0	100
	CPCP	100	100	99	89	3	0	0
$n = 50$	CPMATH	0	0	0	0	0	0	100
	CPCP	99	94	72	18	0	0	0
$n = 60$	CPMATH	0	0	0	0	0	0	11
	CPCP	92	66	25	1	0	0	0

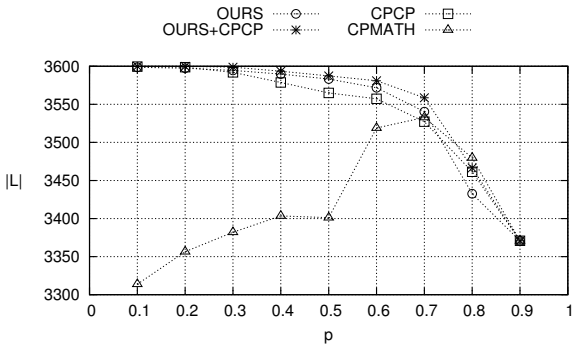

Fig. 3. The averaged objective value $|L|$ with respect to the change of p ($n = 60$)

solution search, CPCP retains the incumbent solution to prune solution subspaces such that there is no hope of finding a better solution. Giving a good incumbent solution may improve the efficiency of the search, or equivalently, it may result a better solution than the default CPCP within the same time limit. When $p \leq 0.2$, OURS and OURS+CPCP are competitive with CPCP and are much better than CPMATH. In this case, CPCP finds a global optimal solution within the time limit in almost all instances (see Table III), and thus OURS is also expected to do so. Surprisingly, when $0.3 \leq p \leq 0.7$ (i.e., "hard" instances), OURS is better than the solvers, and OURS+CPCP is even better. Note that OURS solves an instance with $n = 60$ within only 6.0×10^0 seconds on average, while the solvers take at most 6.0×10^2 seconds. We claim that these results should show the empirical effectiveness of the proposed algorithm.

V. CONCLUDING REMARKS

In this paper, we proposed a heuristic algorithm for the PLSE problem by extending the $\frac{1}{2}$-approximation algorithm [3] that utilizes the notion of maximum matching. We showed how the proposed algorithm is effective in practice through computational experiments.

This paper just shows the possibilities of our approach for the PLSE problem. We believe that the algorithm can be improved further by analyzing its behavior for various instances and the structures of compatibility graphs carefully. It is an alternative to apply metaheuristic techniques such as *genetic algorithm* and *simulated annealing* (*SA*). In fact, SA was applied to *sudoku* problem [15] that asks to find a PLS that is a maximum extension of a given PLS and that satisfies additional constraints. Different from SA, there is no adjustable parameter in the proposed algorithm whose tuning is often exhaustive. Being rather simple, the proposed algorithm delivers a good solution quickly. We can say that the proposed algorithm is a $\frac{1}{3}$-approximation algorithm since it delivers a blocked PLS and any blocked PLS is a $\frac{1}{3}$-factor solution [3]. It is interesting and challenging future work to analyze a nontrivial approximation ratio of the proposed algorithm.

The decision problem version of the PLSE problem has been studied in the field of constraint programming intensively. The problem is whether there is a Latin square that is an extension of a given PLS L_0. The answer is yes only when there is a perfect matching for every compatibility graph G_{L_0}. Any forbidden edge can be ignored in the solution search and the typical constraint programming technique eliminates all of the forbidden edges [16]. Our algorithm is different from this in that ours does not utilize forbidden edges but utilizes mandatory edges.

We have considered a heuristic algorithm for the PLSE problem, assuming practical use. We hope that other researchers take interest in this problem and work on it in the nearest future.

REFERENCES

[1] C. J. Colbourn, "The complexity of completing partial Latin squares," *Discrete Applied Mathematics*, vol. 8, 1984, pp. 25–30.
[2] R. A. Barry and P. A. Humblet, "Latin routers, design and implementation," *IEEE/OSA Journal of Lightwave Technology*, vol. 11-5, 1993, pp. 891–899.
[3] R. Kumar, A. Russel, and R. Sundaram, "Approximating Latin square extensions," *Algorithmica*, vol. 24-2, 1999, pp. 128–138.
[4] I. Hajirasouliha, H. Jowhari, R.Kumar, and R. Sundaram, "On completing Latin squares," In *Proceedings of STACS 2007*, Lecture Notes in Computer Science vol. 4393, 2007, pp. 524–535.
[5] C. P. Gomes, R. G. Regis, and D. B. Shmoys, "An improved approximation algorithm for the partial Latin square extension problem," *Operations Research Letters*, vol. 32-5, 2004, pp. 479–484.
[6] Gurobi Optimizer, http://www.gurobi.com/
[7] IBM ILOG CPLEX, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[8] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2-4, 1973, pp. 225–231.
[9] R. Cymer, "Dulmage-Mendelsohn canonical decomposition as a generic pruning technique," *Constraints*, vol. 17, 2012, pp. 234–272.
[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York and Oxford; 1979.
[11] M. M. Halldórsson and J. Radhakrishnan, "Greed is good: approximating independent sets in sparse and bounded-degree graphs," *Algorithmica*, vol. 18, 1997, pp. 145–163.
[12] R. Barták, "On generators of random quasigroup problems," In *Proceedings of CSCLP 2005*, 2006, pp. 164–178.
[13] C. P. Gomes and D. Shmoys, "Completing quasigroups or Latin squares: a structured graph coloring problem," In *Proceedings of Computational Symposium on Graph Coloring and Generalizations*, 2002.
[14] N. Beldiceanu, M. Carlsson and, J. X. Rampson, "Global constraint catalog," *Technical Report Swedish Institute of Computer Science*, T2005-08, 2006.
[15] R. Lewis, "Metaheuristics can solve sudoku puzzles," *Journal of Heuristics*, vol. 13-4, 2007, pp. 387–401.
[16] J. C. Régin, "A filtering algorithm for constraints of difference in CSPs," In *Proceedings of the twelfth national conference on artificial intelligence*, 1994, pp. 362–367.