Journal of Information Processing Vol.20 No.3 727–737 (July 2012)

[DOI: 10.2197/ipsjjip.20.727]

Regular Paper

How to Produce BlockSum Instances with Various Levels of Difficulty

Kazuya Haraguchi[1,a)] Yasutaka Abe[1] Akira Maruoka[1]

Received: August 31, 2011, Accepted: March 2, 2012

Abstract: We propose a framework that yields instances of certain combinatorial puzzles. To explore such a framework, we focus on certain types of puzzles that ask an assignment of numbers to the cells of an $n \times n$ grid so that it satisfies certain constraints as well as the Latin square condition, that is, each row and column contains all of the numbers in $\{1, 2, \ldots, n\}$. Our algorithm based on the framework automatically yields puzzle instances whose difficulties to solve can be adjusted by means of puzzle inference rules built into the algorithm. Taking up BlockSum puzzle for example, we performed experiments to demonstrate that, as is expected, human solvers tend to solve puzzle instances correctly that are produced with easy inference rules, whereas they tend to fail to solve those produced with sophisticated rules.

Keywords: automatic puzzle generation, BlockSum puzzle, difficulty adjustment

1. Introduction

It has been often reported that practice to solve certain combinatorial puzzles develops human skill to perform intelligent tasks [7], [8]. Although it would be an interesting theme to explore what type of puzzles are effective to leverage such skill and how such skill can be obtained in our brain, it might be quite difficult to answer these questions. If we would try to investigate these themes by an experimental approach, we need numerous puzzle instances with various levels of difficulty. With that in mind, we propose a framework that yields instances of certain combinatorial puzzles such that solving these instances would hopefully leverage human potential to perform certain types of computational tasks. The point of our framework is that it yields instances of certain combinatorial puzzles that have an intended difficulty level.

To explore such a framework, we focus on certain types of puzzles that ask an assignment of numbers to the cells of an $n \times n$ grid that satisfies certain constraints as well as the Latin square condition, that is, each row and column contains integers in $[n]$ exactly once, where $[n]$ denotes $\{1, 2, \ldots, n\}$. It is shown that an algorithm based on our framework automatically yields puzzle instances with such difficulty that can be adjusted by means of inference rules plugged into the algorithm. We performed experiments that demonstrate that, as is expected, human solvers tend to solve puzzle instances correctly that are produced with simple inference rules, whereas they tend to fail to solve those that are produced with somewhat sophisticated rules.

The framework that we propose could be applied to many puzzles given in terms of the Latin square condition such as Sudoku and its variants [5]. In this paper, we focus our arguments on BlockSum [*1]. An instance of BlockSum is given as a partition of an $n \times n$ grid into blocks, each block being associated with a natural number. We restrict ourselves to partitions that consist of connected blocks, that is, blocks made out of side-adjacent cells. BlockSum is a puzzle to ask for an assignment of numbers to the cells of an $n \times n$ grid such that the sum of numbers assigned to cells in each block coincides with the number associated with the block and at the same time the assignment satisfies the Latin square condition. **Figure 1** gives an instance of BlockSum puzzle together with its solution, where n is equal to 4.

Before proceeding to explore how to specify the difficulty of BlockSum puzzle, we explain how to generate instances of the puzzle. The fundamental structure of the algorithm is as follows.

1. Generate a Latin square randomly and associate each number of the Latin square with the corresponding cell.
2. Specify somehow a partition of an $n \times n$ grid into blocks.
3. To each block, assign the sum of the numbers of cells in the block.
4. Delete the numbers allocated to all the cells based on the Latin square.

The idea of the fundamental structure is that first we somehow generate a Latin square together with a partition of an $n \times n$ grid into blocks. Then we compute the sum of each block's numbers, given in terms of the Latin square, and finally hide the numbers of all the cells in the Latin square. To figure out how the structure of the algorithm works, we present **Fig. 2** which gives what remains after each stage of the algorithm.

[1] Department of Information Technology and Electronics, Faculty of Science and Engineering, Ishinomaki Senshu University, Ishinomaki, Miyagi 986–8580, Japan

[a)] kzyhgc@gmail.com

[*1] Although there is a different puzzle that has the same name (e.g., http://infotech.rim.zenno.info/products/blocksum/en/), we use BlockSum for the name of our puzzle as we consider this name the most suitable.

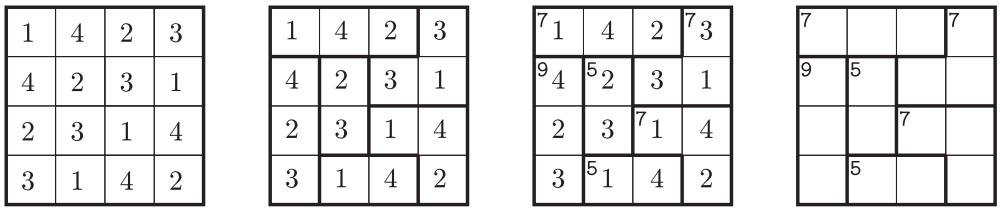

Fig. 2 The four 4×4 grids that are produced just after the four stages of the structure of the algorithm.

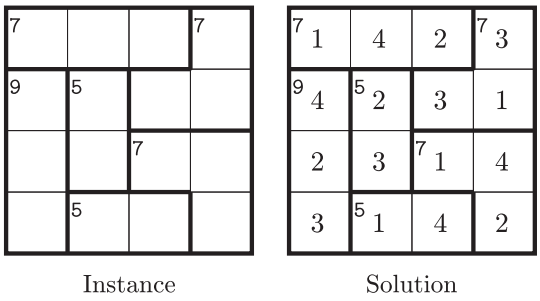

Fig. 1 Examples of an instance of BlockSum puzzle and its solution, where $n = 4$. Small digits indicate the sums of blocks and large digits give integers allocated to cells.

The main theme of this paper is how to make the partition in **2** so that the resultant puzzle instance has a given difficulty level. Obviously, the Latin square generated in **1** becomes a solution of the instance produced. We may call it a certificate since it guarantees that the instance has a solution. In addition to a given difficulty level, we also require that the instance generated has the certificate as the unique solution. Throughout this paper, we assume a certificate that is given in terms of a random Latin square.

We proceed to roughly describe how the algorithm to produce an instance adjusts its difficulty level. Once an algorithm to yield an instance is designed, it just behaves mechanically. On the other hand, the difficulty of solving puzzles is inherently a subjective matter. Then the point is how the algorithm adjusts the difficulty level of a puzzle instance while behaving in the way described above. Our approach to tackle this problem is to assume a collection of inference rules of a solver and to somehow incorporate the collection into the algorithm. An inference rule states that, if an instance and a partially filled solution satisfy a certain condition, we can assign an integer to a certain empty cell. For example, in the instance produced in **4** of Fig. 2, we can assign 1 to the upper left cell since the sum of all 4 integers in one column (or one row) is $1 + 2 + 3 + 4 = 10$ and the sum of all 3 integers in the 1st column except the upper left cell is 9, which is given as the sum of a block. The inference rules may be sophisticated or simple, depending on whether a puzzle solver we assume is an expert or a novice. We plug into the algorithm a collection of certain inference rules. The algorithm built in such a collection behaves to yield as output the "hardest" puzzle instance among those that can be solved by using only the inference rules in the collection. Consequently, the algorithm produces a difficult instance or an easy instance, depending on the inference rules in the collection and whether they are sophisticated or simple. In this way, we can adjust the difficulty level of an instance by choosing appropriate inference rules in the collection that are plugged into the algorithm.

The paper is organized as follows. We first present notations and terminologies in Section 2. In Section 3, we describe two algorithms to specify such a partition that induces a BlockSum instance with a given level of difficulty. In Section 4, we introduce various inference rules for BlockSum puzzle. Then in Section 5, we validate the effectiveness of our approach by experimental studies. Finally, we present concluding remarks in Section 6.

2. Preliminaries

Let a cell in the i th row and in the j th column be denoted by $(i, j) \in [n]^2$. We say that two cells (i, j) and (i', j') are *adjacent* if $|i-i'|+|j-j'| = 1$. Let $B \subseteq [n]^2$. In particular, (i, j) and (i', j') are *adjacent on* B if they are adjacent and belong to B. The relation of adjacency on B is denoted by $\sim_B$, i.e., when (i, j) and (i', j') are adjacent on B we write $(i, j) \sim_B (i', j')$. The transitive closure of $\sim_B$ is denoted by $\hat{\sim}_B$. We say that two cells (i, j) and (i', j') are *connected on* B if $(i, j)\hat{\sim}_B(i', j')$. A subset $B \subseteq [n]^2$ is *connected* if any two cells in B are connected on B. We call a connected subset B a *block*. Throughout this paper, we restrict ourselves to a partition of $[n]^2$ into connected blocks. In particular, we call a block consisting of a single cell a *unit* block.

An *assignment* of integers in $[n]$ to the cells in an $n \times n$ grid is denoted by a function $\varphi : [n]^2 \to [n]$. The function is partial in general. Alternatively, such a function φ is denoted by the set of triples given by $\{(i, j, \varphi(i, j)) \mid \varphi(i, j) \text{ is defined}\}$. Such a set associated with φ is denoted by T_φ. By using this notation, the condition that φ is an extension of φ' is written $T_\varphi \supseteq T_{\varphi'}$. If φ satisfies the following *Latin square condition*, then φ is called a Latin square.

Latin square condition: For any i and j in $[n]$,

$$\{\varphi(i, 1), \ldots, \varphi(i, n)\} = [n]$$

and

$$\{\varphi(1, j), \ldots, \varphi(n, j)\} = [n].$$

A function φ is called a *partial Latin square* if there exists an extension of φ that is a Latin square.

Let P denote a partition of n^2 cells into blocks. Recall that, throughout this paper, any block of the partition is assumed to be connected. Assuming that P consists of m blocks, let us denote P by $\{B_1, \ldots, B_m\}$. Given a partition P, σ denotes a function from P to $[n^2(n + 1)/2]$. The function σ will be used as a condition that a solution of BlockSum puzzle has to satisfy. The condition called the *BlockSum condition* is given in terms of a function σ:

BlockSum condition: For any B in P,

$$\sum_{(i,j)\in B} \varphi(i,j) = \sigma(B).$$

We are ready to define an instance of BlockSum puzzle. An instance of BlockSum puzzle is defined to be a pair of partition P of n^2 cells and function $\sigma : P \to [n^2(n+1)/2]$. Let us denote an instance (P, σ) by I. We call σ a *BlockSum function*. A solution of an instance (P, σ) is defined to be a total function $\varphi : [n]^2 \to [n]$ that satisfies both the BlockSum condition and the Latin square condition. A partial solution of instance I is defined to be a partial function φ such that there exists a solution $\varphi_{\text{total}} : [n]^2 \to [n]$ that is an extension of φ. It should be noticed that an instance (P, σ) arbitrarily given does not necessarily have a solution. As we explained in Section 1, in order to generate an instance that has a solution, we first generate a total function φ randomly and a partition P, which together specify an instance (P, σ) as indicated in Fig. 2. If we specify an instance (P, σ) in the way that Fig. 2 shows, then obviously the instance has at least one solution, that is φ we first generate randomly. Such a random function, denoted by $\varphi_{\text{r}\ell\text{s}}$, will be called a *certificate* because it guarantees that the instance generated this way has at least one solution given by the certificate.

Let $\mathcal{P}$ denote the set of all partitions of n^2 cells into blocks. We introduce a partial order on $\mathcal{P}$ in terms of refinement. That is, a coarser partition is "larger than" a finer partition. More precisely, P *refines* P' if for any B in P there exists B' in P' such that $B \subseteq B'$. When P refines P', we write $P \leqslant P'$. Clearly $\leqslant$ is a partial order on $\mathcal{P}$.

The partially ordered set $(\mathcal{P}, \leqslant)$ has the maximum element, denoted by $P_\top$, and the minimum element, denoted by $P_\bot$, which are given as follows:

$$P_\top = \{[n]^2\},$$
$$P_\bot = \{\{(i,j)\} \mid (i,j) \in [n]^2\}.$$

We call a partition P *non-trivial* if $P \neq P_\top$ and $P \neq P_\bot$. Two blocks B and B' are called *adjacent* if there exist adjacent cells $(i,j) \in B$ and $(i',j') \in B'$. If partitions P_1 and P_2 satisfy $P_1 < P_2$ and there exists no P with $P_1 < P < P_2$, then we say that P_2 *covers* P_1. Clearly, if P_2 covers P_1, then there exist adjacent B and B' in P_1 such that

$$P_2 = (P_1 \setminus \{B, B'\}) \cup \{B \cup B'\}.$$

To put it another way, we can say that P_2 is obtained from P_1 by merging adjacent two blocks B and B' in P_1. Recall that any block in a partition we deal with is connected. We define $\mathcal{U}(P_1)$ to be the set of partitions that cover P_1. The partially ordered set $(\mathcal{P}, \leqslant)$ is drawn as a *Hasse diagram* that consists of nodes associated with partitions and edges from P_1 to P_2 when P_2 covers P_1.

3. Algorithm to Produce a BlockSum Instance

In this section, we shall explain how to specify the partition in **2** in the algorithm described in Section 1 so that we can obtain an instance with desired difficulty.

1. Generate a Latin square $\varphi_{\text{r}\ell\text{s}}$ randomly.
2. Compute somehow a partition P of an $n \times n$ grid.
3. For each block B of P, compute

$$\sigma(B) = \sum_{(i,j)\in B} \varphi_{\text{r}\ell\text{s}}(i,j).$$

4. Output instance (P, σ).

Fig. 3 Structure to produce a BlockSum instance.

3.1 Uniqueness Condition

Before proceeding to discuss how to specify the partition, we rewrite in **Fig. 3** the structure of the algorithm presented in Section 1 in terms of the notations introduced in Section 2.

Throughout this paper, we assume a random Latin square, denoted by $\varphi_{\text{r}\ell\text{s}}$. Since $\varphi_{\text{r}\ell\text{s}}$ is assumed as a certificate, we can take any partition as an instance as described so far. In fact, we impose a further condition on the output partition in **4**. That is, the partition P in **2** is chosen so that $\varphi_{\text{r}\ell\text{s}}$ becomes a unique solution for instance (P, σ). In what follows, we identify a partition with an instance when no confusion arises. Thus, although in general an instance does not necessarily have a solution, we do not need to care about an instance with no solution because we assume a certificate. After all, what we want to do is to derive a partition such that the instance associated with the partition has difficulty that we desire. Before going into details, we describe an idea on how to accomplish it.

The difficulty of an instance is an inherently subjective matter, so we need, first of all, to somehow formalize the difficulty. For that purpose, we introduce a notion of inference rules that are supposed to reflect the intellectual ability of solvers who try to solve instances. More specifically, we introduce various collections of inference rules that reflect the ability of solvers, depending on whether they are novice solvers or expert solvers, and build one of the collections of rules into the algorithm. The idea is that the algorithm shall yield the hardest instance among the ones that can be solved by using only the inference rules that belong to the collection plugged into the algorithm. Thus, we can say that the more sophisticated the inference rules in the postulated collection, the more difficult the instance that the algorithm yields as output tends to be. In order to present what is explained above more clearly, we need to describe it more precisely.

Let us consider how instances with various difficulties represented by the partitions are laid out in the Hasse diagram. The extreme instances are the one denoted by $P_\bot$ at the bottom and the one denoted by $P_\top$ at the top. In case of $P_\bot$, σ gives the values of all the cells in the $n \times n$ grid, so we are done. On the other hand, in case of $P_\top$, any Latin square can be a solution of the instance determined by $P_\top$. After all, we exclude such extreme cases from our consideration.

In order to obtain the instances described above, we only deal with instances that satisfy the following condition.

Uniqueness condition: An instance has the unique solution that is given by the certificate.

Focusing on the hardest instances among the ones that have the certificate as the unique solution, you might expect that an in-

stance with appropriate difficulty appears somewhere between the top and the bottom in the Hasse diagram.

It should be noticed here that obviously the Uniqueness condition is monotone decreasing in the sense that, for any partitions P and P', $P \leqslant P'$ implies $C_{\mathrm{unq}}(P) \geq C_{\mathrm{unq}}(P')$, where $C_{\mathrm{unq}}(P)$ takes value 1 if the instance associated with P has $\varphi_{\mathrm{r\ell s}}$ as the unique solution, and takes value 0 otherwise. To prove the fact, let us assume in contradiction that $P \leqslant P'$ holds, but $C_{\mathrm{unq}}(P) \geq C_{\mathrm{unq}}(P')$ does not hold. The latter condition is equivalent to that both $C_{\mathrm{unq}}(P) = 0$ and $C_{\mathrm{unq}}(P') = 1$ hold. Let the BlockSum functions induced from P and P' be denoted by σ_P and $\sigma_{P'}$, respectively. Obviously, $C_{\mathrm{unq}}(P) = 0$ implies that there exists a solution φ, other than $\varphi_{\mathrm{r\ell s}}$, that satisfies σ_P. On the other hand, since P refines P', $\sigma_{P'}$ is satisfied by not only $\varphi_{\mathrm{r\ell s}}$ but also by φ, contradicting to $C_{\mathrm{unq}}(P') = 1$. Because of the monotonicity of the Uniqueness condition, it makes sense to consider the unique maximal partition, that is, the maximal partition among the ones that satisfy the Uniqueness condition.

We developed an algorithm that yields as output one of the unique maximal partitions [3]. Roughly, the algorithm finds the unique maximal partition by starting at the bottom $P_\perp$ and climbing upward along a path in the Hasse diagram until it reaches one of the maximal partitions such that any partition, immediately above the maximal partition, no longer satisfies the Uniqueness condition. However, unfortunately, the partitions obtained this way turn out to be too difficult for human solvers to solve. This observation motivates us to explore how to produce instances with various difficulties desired.

Basically, the algorithm that produces an instance behaves as described in Fig. 3. Since we can roughly figure out how the algorithm behaves in the three stages except Stage **2**, we shall describe how to specify a partition of $n \times n$ grid which, together with random Latin square $\varphi_{\mathrm{r\ell s}}$, gives an instance.

3.2 How to Specify a Partition with the Desired Difficulty Level

In order to specify a partition with desired difficulty, we introduce a collection of inference rules, denoted by R, which is supposed to be used when human solvers try to solve instances. We call a partition *R-completable* if the instance associated with the partition can be solved by using only the inference rules in R. In other words, a partition is R-completable if all the cells in an $n \times n$ grid of the corresponding instance can be filled out by "reasoning" based on inference rules in R, but without *trial-and-error*. We omit R in R-completable when it is clear from the context. We will present concrete examples of inference rules in Section 4.

We are now ready to explain how to specify a partition. First of all, we specify desired difficulty of an instance in terms of a collection R of inference rules. The algorithm tries to search for a partition that is completable and at the same time that induces a "hardest" instance among the completable partitions. We introduce two ideas on which partitions induce "hardest" instances among the completable partitions, and describe algorithms to search for such a partition.

The first idea is a maximal partition in terms of the refinement relation among the completable partitions. Roughly, the larger a partition becomes, the more difficult the corresponding instance becomes. This is because the larger partition has less constraints on the BlockSum conditions, and hence it becomes more difficult to find the unique solution, namely, the random Latin square, among a larger number of candidates for the unique solution. Therefore, we can say that the completable maximal partition is the "hardest" one among the completable partitions.

SearchMaximal
1. Let $P \leftarrow P_\perp$.
2. Repeat the following:
  2-1. Let $\mathcal{U}_R(P) \leftarrow \{P' \in \mathcal{U}(P) \mid P' \text{ is } R\text{-completable}\}$. The $\mathcal{U}_R(P)$ can be constructed by checking whether each $P' \in \mathcal{U}(P)$ is R-completable by using the procedure CheckCompletability.
  2-2. If $\mathcal{U}_R(P) = \emptyset$, then output P and halt. Otherwise, choose $P' \in \mathcal{U}_R(P)$ arbitrarily and $P \leftarrow P'$.

Procedure CheckCompletability
1. Let $T_\varphi \leftarrow \emptyset$.
2. In the instance associated with P', **while** there is an empty cell (i, j) that we can fill with an integer v by applying a certain inference rule in R, **do** $T_\varphi \leftarrow T_\varphi \cup \{(i, j, v)\}$.
3. If $|T_\varphi| = n^2$ (i.e., all the cells are filled), then P' is R-completable. Otherwise, P' is not R-completable.

Fig. 4 Algorithm SearchMaximal to find an R-completable maximal partition for given R that is used in Stage **2** of Fig. 3.

Now we present an algorithm to output a completable maximal partition. Starting at the bottom $P_\perp$, the algorithm goes upward step by step along a path in the Hasse diagram as long as there exists a completable partition. After all, the algorithm finds out and yields as output a partition P such that no partition above P is completable. In **Fig. 4**, we summarize the algorithm named SearchMaximal that works to output a completable maximal partition. For a partition P, let $\mathcal{U}_R(P)$ denote the subset of $\mathcal{U}(P)$ such that each $P' \in \mathcal{U}_R(P)$ is R-completable. Starting with $P = P_\perp$, the algorithm goes to an arbitrary partition in $\mathcal{U}_R(P)$ repeatedly unless $\mathcal{U}_R(P) = \emptyset$. The algorithm needs to construct $\mathcal{U}_R(P)$, and to do this, it checks whether each $P' \in \mathcal{U}(P)$ is completable or not by the procedure CheckCompletability in Fig. 4. The CheckCompletability tries to solve the instance associated with P' by applying the inference rules in R; it repeatedly applies an appropriate rule in R to fulfill an empty cell with an integer and decides whether or not all the cells in an $n \times n$ grid can be filled with integers. Starting with function φ with all the cells being empty, and with the partition P', CheckCompletability applies an inference rule to revise φ. More precisely, each time an inference rule is applied, φ is updated by filling out an empty cell with an integer based on φ so far obtained and the partition P'. In fact, instead of φ, the algorithm deals with set T_φ that corresponds to φ:

$$T_\varphi = \{(i, j, v) \in [n]^3 \mid \varphi(i, j) = v\}.$$

Figure 5 schematically illustrates the notion of unique maximal and that of R-completable maximal. Let us notice that an R-completable partition is "less than" a unique maximal partition. This is because an R-completable partition is automatically a unique partition. It should be noticed that changing the collection R of rules to a more sophisticated one leads to moving

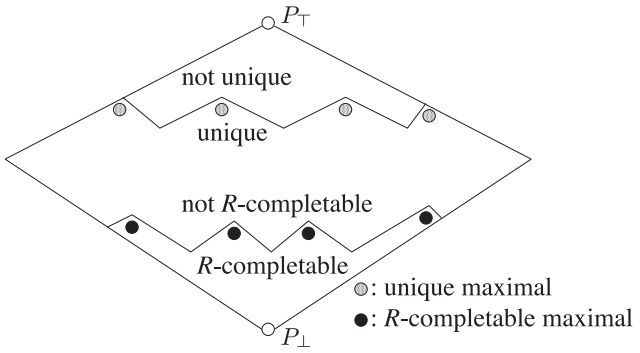

Fig. 5 Unique maximal partitions and R-completable partitions in the Hasse diagram.

upward the boundary between the R-completable region and the not R-completable region. In what follows, this boundary will be called *R-boundary*.

We proceed to the second idea to yield a "hardest" instance among the completable partitions. Recall the definition of completability. Completability guarantees that the instance is solvable by somehow applying inference rules in R, but how they are applied does not matter. It is natural to consider that an instance is more difficult if we need to apply sophisticated rules more often to solve it. Here we consider producing a "hardest" instance based on this observation. For this, to each inference rule r in R, we assign a real number $w(r)$ as the weight so that $w(r)$ reflects the sophistication level of the rule; we assign larger weights to sophisticated rules as compared to simple rules. We then evaluate a completable partition by the sum of the weights of the inference rules that we need to apply in order to solve the associated instance. More precisely, let P denote an R-completable partition. Assume that the completability of P is guaranteed by a sequence $s = ((i_1, j_1, r_1), (i_2, j_2, r_2), \ldots, (i_{n^2}, j_{n^2}, r_{n^2}))$ such that the instance associated with P can be solved by fulfilling from (i_1, j_1) to (i_{n^2}, j_{n^2}) by applying r_1 to r_{n^2}, respectively. We define the *weight of partition P with respect to s*, denoted by $W_s(P)$, as the sum of the weights of $r_1, r_2, \ldots, r_{n^2}$ in the sequence s;

$$W_s(P) = w(r_1) + w(r_2) + \cdots + w(r_{n^2}). \quad (1)$$

Observe that there may exist a couple of sequences that guarantee completability of P. Among these, we use such s that is obtained as follows; When we try to solve the instance, we need to decide (i_k, j_k, r_k) for $k = 1, 2, \ldots, n^2$. If there are a few candidates for (i_k, j_k, r_k), we choose the (i_k, j_k, r_k) that attains the minimum weight $w(r_k)$ since we would like to model the behavior of human solvers who may prefer simple rules that have small weights. In the sequel, we denote the weight of P with respect to such obtained s by $W(P) = W_s(P)$, and simply call $W(P)$ the *weight of P*.

We expect that, the larger the weight of a partition becomes, the harder the associated instance should be. We do not aim, however, at finding the completable partition of the maximum weight since there are a vast number of completable partitions in general, meaning that the search space can be too huge. Instead, we only search one path from $P_\bot$ to a completable maximal partition and employ the one that has the largest weight in the path.

In **Fig. 6**, we summarize the algorithm named SearchLargeWeight to output such a partition. This algorithm behaves in a similar way as SearchMaximal in Fig. 4. Starting with $P = P_\bot$, both algorithms go upward in the Hasse diagram by choosing a certain $P' \in \mathcal{U}_R(P)$ and updating $P \leftarrow P'$ until they reach a completable maximal partition. However, the algorithms are different in the following two points. The first point is the way the algorithm chooses $P' \in \mathcal{U}_R(P)$. The SearchLargeWeight chooses such P' that attains the largest weight among those in $\mathcal{U}_R(P)$, like a typical greedy method, while SearchMaximal chooses an arbitrary $P' \in \mathcal{U}_R(P)$. The second point is that SearchLargeWeight outputs the completable partition of the largest weight among those searched, whereas SearchMaximal outputs a completable maximal partition. For this, SearchLargeWeight maintains the partition P^* that attains the largest weight among those searched so far. After the search is over, SearchLargeWeight outputs P^*, which is not necessarily maximal in terms of the refinement relation. In Section 5, we will report experimental results on whether we can produce BlockSum instances with intended difficulty levels by these two algorithms.

SearchLargeWeight
1. Let $P \leftarrow P_\bot$ and $P^* \leftarrow P_\bot$.
2. Repeat the following:
2-1. Let $\mathcal{U}_R(P) \leftarrow \{P' \in \mathcal{U}(P) \mid P' \text{ is } R\text{-completable}\}$. The $\mathcal{U}_R(P)$ can be constructed by checking whether each $P' \in \mathcal{U}(P)$ is R-completable by using the procedure CheckCompletability in Fig. 4.
2-2. If $\mathcal{U}_R(P) = \emptyset$, then output P^* and halt.
Otherwise, find the partition P' that attains the largest $W(P')$
among the ones in $\mathcal{U}_R(P)$. Let $P \leftarrow P'$.
2-3. If $W(P) > W(P^*)$, then let $P^* \leftarrow P$.

Fig. 6 Algorithm SearchLargeWeight to find an R-completable partition that has a large weight for given R that is used in Stage **2** of Fig. 3.

3.3 Conditions Imposed on Partitions

In order to search for the desired instances, we introduced the Uniqueness condition and the condition of R-completable. We also introduced the notions of unique maximal and R-completable maximal to specify the instances the algorithms yield. These frameworks described so far can be easily generalized. By considering an appropriate condition C, we can introduce a notion of C-maximal in a similar way to these cases. For example, C is taken as a condition that an instance should be solvable by a certain solving technique. However, we are not interested in instances that we need to apply trial-and-error to solve, or more specifically *backtracking*, which is often performed when we deal with search problems. To avoid them, we use the condition of R-completable based on inference rules. We will introduce examples of inference rules in Section 4.

Before proceeding further to explain inference rules, we would like to mention a certain technique that is explored when puzzle-type problems are discussed. The technique that we take up is so called *constraint propagation* [11]. In our case, this technique could be applied when the algorithm checks whether or not a given partition is R-completable, which is done by CheckCompletability in Fig. 4. Assume that we have a partial solution φ to

the instance associated with the partition. The point of the technique is that, in the course of making the decision, the algorithm keeps a list of possible candidate integers for each cell (i, j) and iterates updating the lists by eliminating integers based on which integers each list contains. For example, we can eliminate such a candidate integer v that will violate the Latin square condition or the BlockSum condition if (i, j) were filled with v. The algorithm may identify whether v will violate at least one of the two conditions in the course of completing an assignment $T_\varphi \cup \{(i, j, v)\}$ by applying appropriately chosen inference rules repeatedly; if we encounter a violating assignment, then we can eliminate v from the list of (i, j). In principle, we can apply such a constraint propagation based technique to our framework. However, we have not yet applied such a technique, which will be left for future work.

4. Collection *R* of Inference Rules

The algorithm that computes an instance in the way described so far needs to check if the partition in question is R-completable repeatedly. Concretely, the check is done by the procedure CheckCompletability in Fig. 4. In this section, we introduce examples of inference rules for BlockSum puzzle that are to be included in R. An inference rule states that, if a given partition P and a partial solution φ to the instance associated with P satisfy a certain condition, then we can specify an integer v that should be assigned to a certain empty cell (i, j). In this case, we can update the partial solution $T_\varphi \leftarrow T_\varphi \cup \{(i, j, v)\}$.

We introduce 9 inference rules. Among these, 4 inference rules are derived from the Latin square condition, and the other 5 inference rules are derived from the BlockSum condition. In **Figs. 7** and **8**, we illustrate the situations in which the inference rules are applied. The inference rules have different levels of sophistication. Some inference rules are simple in the sense that they are directly derived from the Latin square condition or the BlockSum condition, whereas others are sophisticated in the sense that the derivation is not so simple. In what follows, the empty cell that an inference rule fills with an integer is denoted by (i, j) and the integer assigned to the (i, j) is denoted by v.

The 4 Inference Rules Derived from The Latin Square Condition

We denote the 4 inference rules derived from the Latin square condition by $r_1^{\ell\text{ts}}$, $r_2^{\ell\text{ts}}$, $r_3^{\ell\text{ts}}$ and $r_4^{\ell\text{ts}}$. The suffix represents the sophistication level of the inference rule that we decide based on our experience, e.g., we consider $r_2^{\ell\text{ts}}$ more sophisticated than $r_1^{\ell\text{ts}}$. The simplest rule $r_1^{\ell\text{ts}}$ states that, if all the $n-1$ cells in the i th row (or in the j th column) other than (i, j) are already filled with $n-1$ distinct integers, then fill (i, j) with the remaining integer v. The inference rule $r_2^{\ell\text{ts}}$ is a generalization of $r_1^{\ell\text{ts}}$. Let $X \subseteq [n]^2$ denote the set of all the cells in the i th row and in the j th column other than (i, j). Obviously, $|X| = 2(n-1)$. The inference rule $r_2^{\ell\text{ts}}$ states that, if $n-1$ distinct integers are already assigned to cells

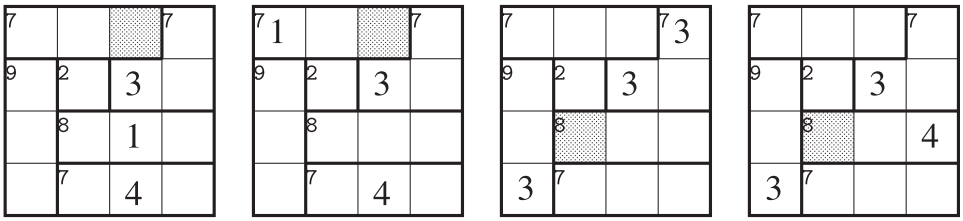

Fig. 7 Situations where the 4 inference rules derived from the Latin square condition are applied. A cell filled with an integer is denoted by (i, j), and the integer is denoted by v. The cell (i, j) is designated by a shaded square. These conventions are also used in Fig. 8.

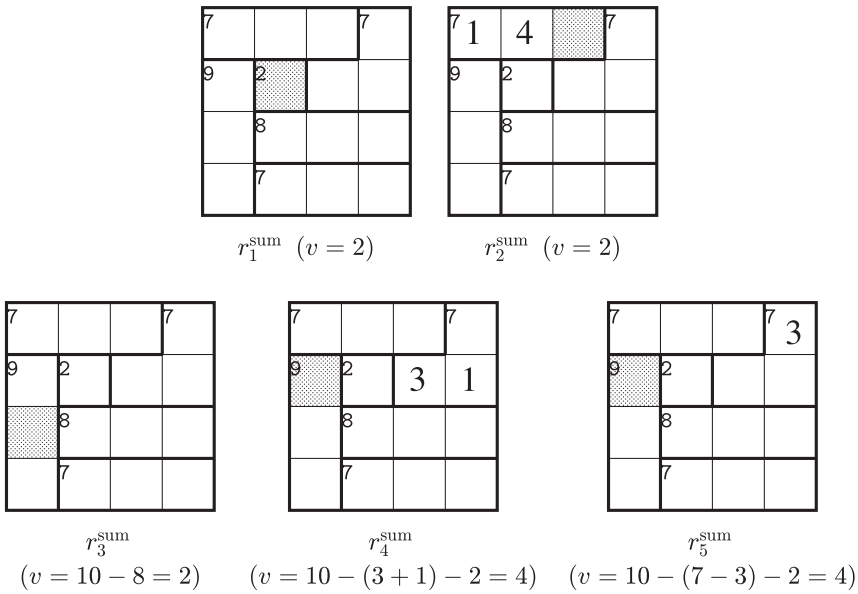

Fig. 8 Situations where the 5 inference rules derived from the BlockSum condition are applied.

in X, then fill (i, j) with the remaining integer v.

Let Y be the set of cells in the i th row (resp., the j th column) other than (i, j). Obviously, $|Y| = n - 1$. If there exists an integer v such that any cell in Y is judged not to be filled with v, then we can fill (i, j) with v. We adopt the following two conditions that v does not appear on any cell in Y, denoted by (i, ℓ) with $\ell \neq j$ (resp., (ℓ, j) with $\ell \neq i$).

(i) Integer v appears in a cell outside the i th row (resp., the j th column), that is, a cell (k, ℓ) with $k \neq i$ (resp., (ℓ, k) with $k \neq j$).

(ii) An integer other than v appears on cell (i, ℓ) (resp., (ℓ, j)).

The inference rule $r_3^{\ell\text{ts}}$ states that, if any cell in Y is in the case (i), then fill (i, j) with the integer v. The inference rule $r_4^{\ell\text{ts}}$ is a generalization of $r_3^{\ell\text{ts}}$. It states that, if any cell in Y is in either (i) or (ii), then fill (i, j) with the integer v. We do not take up the situation in which any cell in Y is in the case (ii) since we already considered this situation as the inference rule $r_1^{\ell\text{ts}}$.

The 5 Inference Rules Derived from The BlockSum Condition

We denote the 5 inference rules derived from the BlockSum condition by r_1^{sum}, r_2^{sum}, r_3^{sum}, r_4^{sum} and r_5^{sum}. Again, the suffix represents the sophistication level that we decide based on our experience. The simplest inference rule r_1^{sum} states that, if (i, j) is contained in a unit block B (i.e., B consists solely of a single cell (i, j)), then fill (i, j) with $v = \sigma(B)$ that is clearly given to the solver. The inference rule r_2^{sum} is a generalization of r_1^{sum}. It states that, if there exists a block B such that all the cells in B except (i, j) are filled with integers, then fill cell (i, j) in B with

$$v = \sigma(B) - \sum_{(k,\ell)\in B:\ (k,\ell)\neq(i,j)} \varphi(k, \ell).$$

If there exists a row (resp., a column) such that the sum of the integers of all the cells except (i, j) in the row (resp., the column) can be computed, then fill cell (i, j) with $n(n + 1)/2$ minus the sum. Let Y denote the set of cells except (i, j) in the i th row (resp., the j th column). In order to compute the sum, consider the following three cases where the integer of a cell in Y or the sum of the integers of a subset of Y can be specified.

(iii) A cell is already filled out with an integer by φ.

(iv) There exists a block B contained within Y so that the sum of the integers in B is given by $\sigma(B)$.

(v) There exists a block B such that all the cells in B outside the row (resp., the column) are filled with integers. Let the set of such outside cells filled with integers be denoted by A. The set $B \setminus A$ is assigned with $\sigma(B)$ minus the sum of the integers of cells in A.

The inference rule r_3^{sum} (resp., r_4^{sum} and r_5^{sum}) states that, if the entire Y consists of such cells and subsets that are in the case (iv) (resp., in the cases (iii) or (iv), and in the cases (iii), (iv) or (v)), then fill (i, j) with the integer v that is given by $n(n + 1)/2$ minus the sum of integers that are specified as above. Clearly, the inference rule r_4^{sum} is a generalization of r_3^{sum}, and r_5^{sum} is a generalization of r_4^{sum}.

In **Fig. 9**, we summarize the generalization relationship among the 9 inference rules and their weights for SearchLargeWeight. The weight values are used in the experiments in Section 5. We use powers of two as weights for all the inference rules except

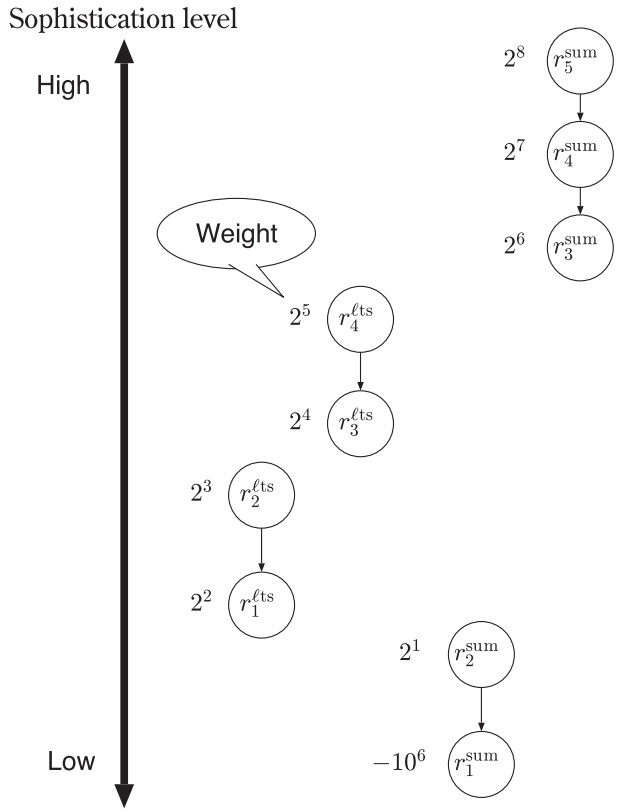

Fig. 9 The generalization relationship among the 9 inference rules and their weights. A directed edge indicates that the inference rule on the starting point is a generalization of the one on the end point.

r_1^{sum}. The inference rule r_1^{sum} specifies the integer that should be assigned to the cell of a unit block. As it must be obvious to human solvers, we would like to remove unit blocks by setting $w(r_1^{\text{sum}})$ to an extremely small value.

Before closing this section, let us give some remarks on R-completability and on the weight of an R-completable partition when R is a subset of the 9 inference rules. Recall that some inference rules are a generalization of others. For example, $r_2^{\ell\text{ts}}$ is a generalization of $r_1^{\ell\text{ts}}$. Suppose that R contains $r_2^{\ell\text{ts}}$ but does not contain $r_1^{\ell\text{ts}}$ (i.e., $r_2^{\ell\text{ts}} \in R$ and $r_1^{\ell\text{ts}} \notin R$). Then adding $r_1^{\ell\text{ts}}$ to R does not change the R-boundary. In other words, if we denote $R' = R \cup \{r_1^{\ell\text{ts}}\}$, any R-completable partition is R'-completable, and any R'-completable partition is R-completable. This is because the situation where $r_1^{\ell\text{ts}}$ can be applied is also the situation where $r_2^{\ell\text{ts}}$, its generalization, can be applied. Hence whether we choose R or R' does not matter in the context of completability, whereas it matters in the context of weight. Recall that we evaluate a completable partition P by the weight $W_s(P)$ with respect to the sequence $s = ((i_1, j_1, r_1), \ldots, (i_{n^2}, j_{n^2}, r_{n^2}))$ that was defined in (1). To compute s, we choose (i_k, j_k, r_k) of the smallest $w(r_k)$ among the candidates for $k = 1, 2, \ldots, n^2$. Therefore, since we take $w(r_1^{\ell\text{ts}}) < w(r_2^{\ell\text{ts}})$, then R and R' may give different weights to completable partitions, and SearchLargeWeight may output different partitions between R and R'.

5. Experimental Results

To adjust the difficulty level, the basic idea is to decide the collection R of inference rules that assumed human solvers may have, and to search for such an R-completable partition that in-

duces a "hardest" instance among all the R-completable ones. We introduced two ideas on what completable partition induces a "hardest" instance. One is a maximal completable partition in terms of the refinement relation. The other is a completable partition that requires us to apply sophisticated rules in R many times to solve, the degree of which is measured by the weight. We described two algorithms, SearchMaximal and SearchLargeWeight, to search for a completable partition of these two types.

In this section, we investigate whether BlockSum instances produced by the algorithm really have the intended difficulty levels. Although instances are basically produced by our mechanical algorithm, evaluating difficulty in solving those instances is inherently subjective in nature. One way to validate that the instances have actually intended difficulties is to conduct experiments in which we obtain empirical data by making human solvers solve BlockSum instances and then evaluate difficulties of the instances in terms of the number of cells filled with the correct integers.

We conduct two experiments. In Section 5.1, we compare difficulty levels of the instances generated with different R'-s, where we produce instances by generating R-completable maximal partitions by SearchMaximal. In Section 5.2, for a fixed R, we compare difficulty levels of the instances generated with two algorithms to specify a partition, SearchMaximal and SearchLargeWeight.

5.1 Comparison of Difficulty Levels of the Instances Produced with Different Rule Collections

In our expectation, the more sophisticated R we build into the algorithm, the more difficult instance the algorithm should produce. We validate this expectation by an experiment using BlockSum instances produced with different R'-s.

Let us describe the protocol of the experiment precisely. First we prepared 3 collections of inference rules, denoted by R_+, R_{++} and R_{+++}, where we intend a simple rule collection by R_+, an intermediate rule collection by R_{++}, and a sophisticated rule collection by R_{+++}. We take R_+ as a small collection of simple inference rules so that there exist non-trivial partitions (which are neither $P_\bot$ nor $P_\top$) that are R_+-completable. Here we use $R_+ = \{r_1^{\ell\mathrm{ts}}, r_2^{\mathrm{sum}}\}$. We take the intermediate rule collection R_{++} as a random subset of the 9 inference rules so that $R_+ \subseteq R_{++}$ holds. We take R_{+++} as the set of the 4 general inference rules, that is, $R_{+++} = \{r_2^{\ell\mathrm{ts}}, r_4^{\ell\mathrm{ts}}, r_2^{\mathrm{sum}}, r_5^{\mathrm{sum}}\}$. Note that adding any other inference rule to R_{+++} does not change the R_{+++}-boundary. Observe that the R_{++}-boundary is not below the R_+-boundary and that the R_{+++}-boundary is not below the R_{++}-boundary.

Next, we constructed a computer system on which human solvers can play BlockSum puzzle. For a rule collection R that we choose, the system produces a BlockSum instance by generating an R-completable maximal partition by SearchMaximal, based on a randomly chosen certificate. A human solver can play the produced BlockSum instances repeatedly on the system. The solver can skip an instance if he or she cannot solve it. While the solver is playing the instances, the system stores the data on which integer is assigned to which cell.

We collected 38 students in our institution as experimental solvers. We divided them into 3 groups in order to make them solve the instances generated with different rule collections. Each student is given the ID number by the institution. We denote by S_+ (resp., S_{++} and S_{+++}) the group of students such that the residue of the ID number divided by 3 is 0 (resp., 1 and 2). There are 14 (resp., 13 and 11) students in the group S_+ (resp., S_{++} and S_{+++}). We consider that the numbers are not different from each other to a large extent. The students in S_+ (resp., S_{++} and S_{+++}) will be assigned the instances generated with R_+ (resp., R_{++} and R_{+++}). We told the students neither which groups they belong to nor the purpose of the experiment. Since no student had great experience in playing BlockSum puzzle, we taught the students the rule of the puzzle and let them work on some practice. Seating the students in front of PCs, we asked them to solve as many BlockSum instances as possible on our system within 10 minutes. Each student was assigned instances that were generated with the rule collection of the group which the student belongs to. We restricted all the instances to $n = 4$ for which the system produces an instance in less than 0.1 seconds. Then we regard that almost all 10 minutes were devoted to solving the instances.

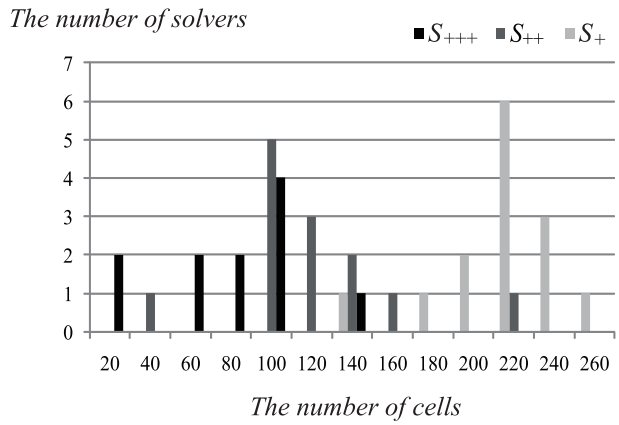

Fig. 10 The distributions of the total numbers of cells to which the solvers in S_{+++}, S_{++} and S_+ gave the correct integers. For a number x in the horizontal axis, the height of the bar represents the number of solvers who gave the correct integers in $x - 19$ to x cells.

The system stored the data on this 10-minute test successfully. From the data, we calculated how many cells each student filled with the correct integers. We show the distributions of the 3 groups in **Fig. 10**. The students in S_+ tend to give more correct integers than those in S_{++} and S_{+++}; 13 out of the 14 students in S_+ gave the correct integers to more than 160 cells, while 23 out of the 24 students in S_{++} and S_{+++} gave the correct integers to at most 160 cells. Although both S_{++} and S_{+++} have their peaks at 81 to 100 cells, we claim that the students in S_{++} should tend to give more correct integers than those in S_{+++}; 5 students in S_{++} and 4 students in S_{+++} gave the correct integers in 81 to 100 cells, but 7 out of the remaining 8 students in S_{++} gave the correct integers to more than 100 cells, while 6 out of the remaining 7 students in S_{+++} gave the correct integers to at most 80 cells. These results illustrate that difficulty levels are adjusted to some extent by means of the rule collection.

For the criterion for whether the instances have the intended difficulty, we used the number of cells that are filled with the correct integers, while there may be various alternative criteria. An

example of such criteria is the time needed to solve an instance. We expect that, the more difficult an instance becomes, the more time a human solver should need to solve it. Therefore, we also expect that, the more sophisticated rule collection we build into the algorithm, the more time a human solver needs to solve the generated instance. However, collecting the data on the time to solve is more time-consuming than collecting the data on the number of cells filled with the correct integers. The time to solve is not measured until the solver solves an instance completely, that is, fills all n^2 cells with the correct integers. On the other hand, we can measure the number of cells filled with the correct integers within an appropriate time-limit even if the solver does not solve an instance completely or skips it. This convenience is the reason why we employed the number of correct cells as the criterion in the experiment.

5.2 Comparison of Difficulty Levels of the Instances Produced with Different Algorithms to Specify a Partition

In the last experiment, we confirmed that we can adjust the difficulty level by choosing R appropriately, where we produced an instance by generating an R-completable maximal partition by SearchMaximal, regarding such a partition as the "hardest" among all the R-completable partitions. Recall that we introduced two ideas on which completable partition induces a "hardest" instance. Next we compare difficulty levels of the "hardest" instances of the two types. In other words, we observe which algorithm to specify a partition, SearchMaximal or SearchLargeWeight, results more difficult instances for a fixed R. Here we take R as the set of all the 9 inference rules introduced in Section 4. For SearchLargeWeight, we use the values indicated in Fig. 9 as the weights of the inference rules.

We performed the experiment in a similar way as Section 5.1. We collected 26 students in our institution as experimental solvers. The students do not overlap with those in the last experiment. We divided the students into two groups in order to assign them with the instances produced by different algorithms. The division was made at random so that the two groups have the same number of students. We denote the two groups by S_{weight} and S_{maximal}, which were assigned the instances produced by SearchLargeWeight and SearchMaximal respectively. We made the students solve so many BlockSum instances as possible within 10 minutes. To try various experimental environments, we did not utilize the system that was used in the last experiment, but utilized handouts; we made two types of handouts such that one type has 30 instances produced by SearchLargeWeight and the other type has 30 instances produced by SearchMaximal. Each student in S_{weight} (resp., S_{maximal}) was given a handout in the former (resp., latter) type. We found that 30 instances are sufficient for the time-limit of 10 minutes. Other major settings are as in Section 5.1.

After the 10-minute test finished, we collected the handouts and calculated how many cells each student filled with the correct integers. In **Fig. 11**, we show the distributions of S_{weight} and S_{maximal}. The two distributions have an overlap; 7 students in S_{weight} and 4 students in S_{maximal} gave the correct integers in 41 to 100 cells. However, all the remaining 6 students in S_{weight} gave the correct integers to at most 40 cells, whereas all the remaining 9 students in S_{maximal} gave the correct integers to at least 100 cells. Based on these, for a given R, we conclude that SearchLargeWeight tends to produce more difficult instances than SearchMaximal.

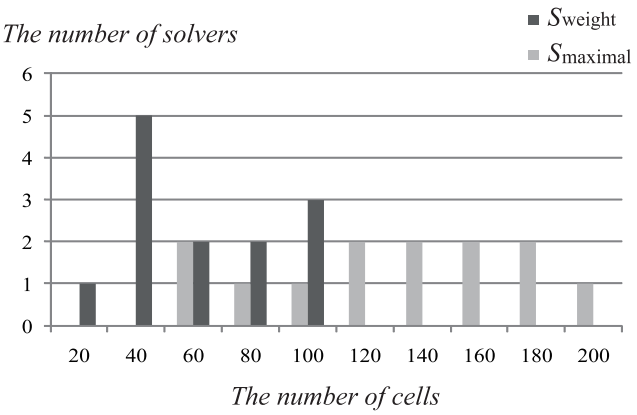

Fig. 11 The distributions of the total numbers of cells to which the solvers in S_{weight} and S_{maximal} gave the correct integers.

Table 1 The weights of produced partitions and the ratios of R-completable maximal partitions among them.

n	SearchMaximal (Weight)	(Ratio)	SearchLargeWeight (Weight)	(Ratio)
4	$3.0\times(-10^6)+152.0$	100%	$1.4\times(-10^6)+365.6$	100%
5	$5.7\times(-10^6)+397.4$	100%	$2.7\times(-10^6)+629.5$	87%
6	$8.3\times(-10^6)+577.4$	100%	$5.3\times(-10^6)+742.7$	95%
7	$14.5\times(-10^6)+468.3$	100%	$8.6\times(-10^6)+974.3$	92%
8	$19.7\times(-10^6)+703.7$	100%	$14.1\times(-10^6)+1146.5$	92%
9	$26.7\times(-10^6)+762.6$	100%	$19.5\times(-10^6)+1355.2$	89%

5.3 Comparison of the Two Algorithms on Other Aspects

We compare the two algorithms on other aspects. Throughout this subsection, we use the collection R of all the 9 inference rules in Section 4. First, we compare the algorithms in terms of the weights of produced partitions. For any $4 \le n \le 9$, we produce 1,000 partitions by each algorithm based on random certificates. We then compute the average of the weights of the produced partitions. We show the results in **Table 1**. In the table, the weights are represented as the sum of the averaged number of unit blocks multiplied by -10^6, which is the weight of r_1^{sum}, and the values derived from the other inference rules. Recall that r_1^{sum} has the smallest weight among the inference rules. When we compute the weight of a partition, r_1^{sum} is surely applied for each unit block. We can see that the partitions produced by SearchLargeWeight have larger weights and fewer unit blocks than SearchMaximal. In Table 1, we also show the ratio of R-completable maximal partitions among the 1,000 partitions. SearchMaximal achieves 100% by definition, whereas SearchLargeWeight does not do so for $n \ge 5$. This means that R-completable partitions of the largest weights are not necessarily maximal in terms of the partial order $\leqslant$.

Next, we compare the two algorithms in terms of computation time needed to specify a partition, which is the most time-consuming part in our algorithm to produce a BlockSum instance. For any $4 \le n \le 9$, we measured the average of computation time over 1,000 generations of partitions. We show the averages in

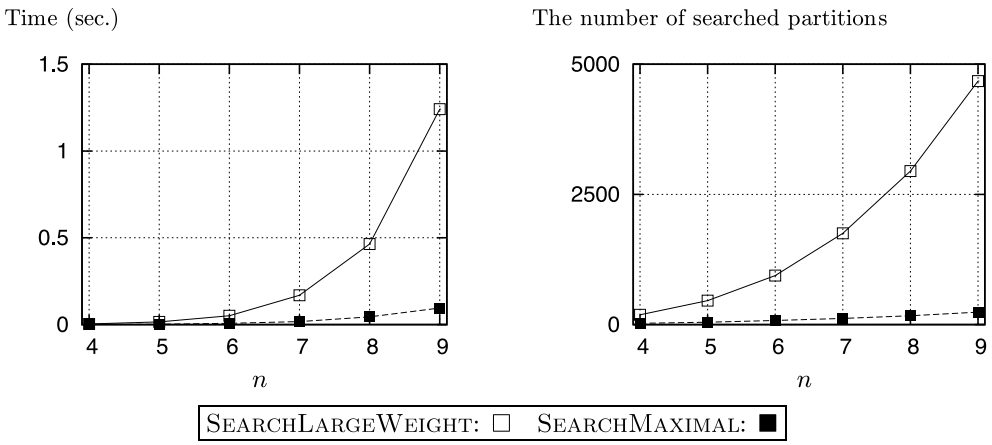

Fig. 12 (Left) Computation time taken to specify a partition. (Right) The number of searched partitions.

the left of **Fig. 12**. Our PC carries 2.83 GHz CPU. As shown, SearchMaximal specifies a partition in less than 0.1 seconds for any $n \leq 9$, whereas SearchLargeWeight takes about 0.5 seconds for $n = 8$ and more than 1 second for $n = 9$. The reason why both algorithms are different in computation time comes from the difference in the number of searched partitions, which is shown in the right of Fig. 12. Recall the two algorithms in Figs. 4 and 6. Suppose that we are now with a completable partition P. In principle, both algorithms search all the partitions in $\mathcal{U}(P)$ to construct the subset $\mathcal{U}_R(P)$ of completable partitions that cover P. Observe that, however, SearchMaximal can go to the first found completable partition without searching the remaining ones since an arbitrary partition in $\mathcal{U}_R(P)$ is admitted as the next partition. In fact, our current implementation searches $\mathcal{U}(P)$ in a random order and goes to the first found completable partition. On the other hand, SearchLargeWeight needs to search all the partitions in $\mathcal{U}(P)$ in order to find the completable partition of the largest weight in $\mathcal{U}(P)$.

6. Discussion and Concluding Remarks

In this paper, we presented a framework to produce BlockSum instances with various difficulty levels. In our framework, we produce a BlockSum instance in the way that we first determine a random Latin square $\varphi_{\mathrm{r}\ell\mathrm{s}}$ as the certificate and then specify a partition P of n^2 cells. To adjust the difficulty level, we build into the algorithm a collection R of inference rules that assumed players may have; the more sophisticated rules R contains, the more difficult instance we expect to be generated. The algorithm generates such an instance that has $\varphi_{\mathrm{r}\ell\mathrm{s}}$ as its unique solution and at the same time that is the "hardest" among the instances that can be solved by using only the inference rules in R. In Section 3, we presented two algorithms to specify a partition, SearchMaximal and SearchLargeWeight, based on two ideas on which partition induces the "hardest" instance. In Section 4, we introduced 9 inference rules of various sophistication levels. In Section 5, we validated whether the instances produced by the algorithm have intended difficulty levels. As is expected, the more sophisticated rules R contains, the less correct integers human solvers are likely to give. We also observed that, given appropriate weights to inference rules, SearchLargeWeight tends to produce more difficult instances than SearchMaximal.

The framework discussed so far can be applied to combinatorial puzzles that ask to assign integers in $\{1, 2, \ldots, n\}$ to all the cells of the $n \times n$ grid so that the assignment satisfies not only the Latin square condition but also certain other conditions. Sudoku is a good example of such combinatorial puzzles. We already studied how to apply the framework to generation of a Futoshiki [2], [5] instance, and will report the results in our future papers.

The highlight of this paper is that, to the best of our knowledge, it is the first paper that considers the instance generation task as the main theme. Most of the previous papers deal with one of the following two themes: **(i)** To solve a given instance by such mathematical methods as *constraint programming* [1], [11] and *metaheuristics* [6], where the methods are mainly evaluated by how fast they solve the instance. **(ii)** To rate the difficulty level of a given instance based on how it is solved by a mathematical method [1], [4], [9], [11], where the rating is evaluated by how it agrees with the difficulty level given by a human puzzle designer. The instance generation task includes repetition of solving candidate puzzle instances as sub-tasks. We need to define the instance space, to design the algorithm that searches the space by solving candidate instances, and to determine the mechanism to select the output among the instances searched so far. We formulate the instance space as a partially ordered set of partitions, each of which corresponds to one instance that has the certificate as its solution. Our algorithm starts from the bottom of the Hasse diagram and goes upward along a path. The algorithm checks the completability of any searched partition. After finding a completable maximal partition, it outputs the instance associated with the maximal partition (SearchMaximal) or the one associated with the partition of the largest weight among those searched (SearchLargeWeight).

Simonis [11] applied constraint programming techniques to solve Sudoku instances and also sketched a procedure to generate a Sudoku instance. His procedure first decides an $n \times n$ Latin square that serves as the certificate, fills all the cells in the $n \times n$ grid with the integers of the Latin square, and then repeats removing an integer in the grid unless a termination condition is satisfied. Hence, his procedure and our algorithm have something

in common. He gave neither the details of the generation procedure nor experiments on generated Sudoku instances. We can regard that this paper concretizes Simonis's idea and applies it to generation of a BlockSum instance.

We have two major future works. **(i)** We are interested in what kind of instances human players can solve or not. In our experience, an instance associated with a unique maximal partition is too difficult for human solvers in general, whereas an instance associated with an R-completable partition is not so difficult as long as we take R as a subset of the inference rules introduced in Section 4. In order to observe what instances lie between the uniqueness boundary and the R-boundary, we should perform experiments using more difficult instances. Thus we need to apply such a technique as constraint propagation to our framework, which was described in Section 3.3. **(ii)** We need to exploit better experimental schemes. We dealt with experimental human solvers in order to validate whether instances produced by the algorithm have the intended difficulty levels. We have not seen such experiments in the literature until the latest work of Palanek [10]. Palanek uses data on the time needed to solve Sudoku instances, where the data is obtained by making anonymous human solvers solve instances on the Internet. To design an experimental scheme, we need to define the data type to be collected (e.g., the number of cells filled with the collect integers within a time-limit, or the time needed to solve an instance) and to design the protocol to obtain the data. In order to obtain the data, we collected students in our institution and made them solve puzzle instances in front of us. In our experimental scheme, it is not easy to collect many experimental solvers, but it is easy to make them solve the instances intensively. Therefore, the data obtained are expected to be good for analysis. On the other hand, if we constructed a system based on the Internet, we could collect a lot of experimental solvers more easily, but it would be difficult to force them to work on puzzles intensively; the data obtained would be rather noisy. There are various alternatives for experimental schemes. We should clarify their characteristics and use the most suitable one for our purpose.

Our goal is to establish a framework to produce puzzle instances with various difficulty levels, not limited to development of a fast algorithm to solve a given puzzle instance. We believe that this is a challenging research theme in such fields as artificial intelligence and primary education.

Acknowledgments We gratefully acknowledge very careful and detailed comments given by anonymous reviewers. This work is partially supported by Grant-in-Aid for Scientific Research (C, 20500760) from the Japan Society for the Promotion of Science (JSPS), and by the Foundation for the Fusion of Science and Technology (FOST).

References

[1] Davis, T.: The Mathematics of Sudoku (online), available from ⟨http://www.geometer.org/mathcircles/sudoku.pdf⟩ (accessed 2011-06-01).
[2] Flueckiger, M.: Sudoku-Puzzles.net (online), available from ⟨http://www.sudoku-puzzles.net/⟩ (accessed 2011-06-01).
[3] Haraguchi, K., Hiraoka, Y. and Maruoka, A.: How to Construct Solvable Instances for BlockSum Puzzle, *Proc. 11th Japan-Korea Joint Workshop on Algorithms and Computation* (*WAAC08*), pp.85–92 (2008).
[4] Johnson, A.: Simple Sudoku (online), available from ⟨http://www.angusj.com/sudoku/⟩ (accessed 2011-06-01).
[5] Leeuwen, M.: Sudoku Variants, Passion for Puzzles (online), available from ⟨http://www.passionforpuzzles.com/sudoku-variants/⟩ (accessed 2011-06-01).
[6] Lewis, R.: Metaheuristics can solve sudoku puzzles, *Journal of Heuristics*, Vol.13, pp.387–401 (2007).
[7] Miyamoto, T.: *Kyouikuron* (*The Art of Teaching without Teaching*), Discover (2004). Japanese book.
[8] Miyamoto, T.: *Chou Kyouikuron* (*The Super Art of Teaching without Teaching*), Discover (2006). Japanese book.
[9] Ono, S., Miyamoto, R., Nakayama, S. and Mizuno, K.: Difficulty estimation of number place puzzle and its problem generation support, *Proc. ICCAS-SICE 2009*, pp.4542–4547 (2009).
[10] Pelánek, R.: Difficulty Rating of Sudoku Puzzles by a Computational Model, *Proc. Florida Artificial Intelligence Research Society Conference* (online), available from ⟨http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS11/paper/view/2517⟩ (2011).
[11] Simonis, H.: Sudoku as a Constraint Problem, *Proc. 4th International Workshop of Modelling and Reformulating Constraint Satisfaction Problems*, pp.13–27 (2005).

Kazuya Haraguchi received his B.E., Master of Informatics, and Doctor of Informatics from Kyoto University, in 2001, 2003 and 2007, respectively. He is currently with the Department of Information Technology and Electronics, Faculty of Science and Engineering, Ishinomaki Senshu University. His interest includes algorithms, optimization, and their application to artificial intelligence and operations research.

Yasutaka Abe received his B. Sci. Eng. from Ishinomaki Senshu University in 2010. He is currently in the master course in the Department of Material Engineering at the Graduate School of Science and Engineering, Ishinomaki Senshu University.

Akira Maruoka graduated in 1965 from the Faculty of Engineering, Tohoku University, and received Dr. of Eng. degree from Tohoku University in 1971. He joined the Faculty of Engineering, Tohoku University in 1971, and had been a professor in the Department of Information Engineering since 1985. In 2006 he moved to Ishinomaki Senshu University as a professor. His research interests include circuit complexity, computational complexity and computational learning theory.