


# Solving and Generating Nagareru Puzzles

Masakazu Ishihata  

NTT Communication Science Laboratories, Kyoto, Japan

Fumiya Tokumasu

National Institute of Technology, Nara College, Nara, Japan

---

## Abstract

Solving paper-and-pencil puzzles is fun for people, and their analysis is also an essential issue in computational complexity theory. There are some practically efficient solvers for some NP-complete puzzles; however, the automatic generation of interesting puzzle instances still stands out as a complex problem because it requires checking whether the generated instance has a unique solution. In this paper, we focus on a puzzle called Nagareru and propose two methods: one is for implicitly enumerating all the solutions of its instance, and the other is for efficiently generating an instance with a unique solution. The former constructs a ZDD that implicitly represents all the solutions. The latter employs the ZDD-based solver as a building block to check the uniqueness of the solution of generated instances. We experimentally showed that the ZDD-based solver was drastically faster than a CSP-based solver, and our generation method created an interesting instance in a reasonable time.

**2012 ACM Subject Classification** Computing methodologies → Combinatorial algorithms; Theory of computation → Generating random combinatorial structures; Mathematics of computing → Graph algorithms

**Keywords and phrases** Paper-and-pencil puzzle, SAT, CSP, ZDD

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2022.2

**Supplementary Material** *Software (Source Code and Data):*

<https://github.com/masakazu-ishihata/Nagareru>

archived at `swh:1:dir:6c6604df4d55d8e2f019dfde5e19225060e8ce83`

## 1 Introduction

Paper-and-pencil puzzles are a type of logic puzzle; a player gradually fills in parts of a solution on the puzzle board without violating any rules and eventually constructs a single consistent solution. Solving a puzzle is much fun for puzzle fans, but it has also attracted the extensive attention of theoretical computer scientists because of an interest in computational complexity [13]. They have been competing to prove the computational complexity of various puzzles, and the following is just a small selection of the list of puzzles that have so far proved to be NP-complete to solve: Bag (Corral) [3], Cross Sum [23], Country Road [5], Dosun-Fuwari [8], Herugolf [7], Hiroimono [2], Makaro [7], Moon-or-Sun [9], Nagareru [9], Nurikabe [4], Nurimeizu [9], Nurimisaki [10], Sashigane [10], Slitherlink [23], Sudoku (Number Place) [23], Tatamibari [1], Yajilin [5], Yosenabe [6], and more. The above series of studies is essential from the point of view of computational complexity theory; however, not so crucial for puzzle fans because it does not directly help them enjoy puzzles more. In contrast, the automatic generation of puzzle instances is one of the most promising computer science techniques for puzzle fans. They believe that one of the necessary conditions for an interesting puzzle instance is that the instance admits precisely one solution. However, for some puzzles, checking the uniqueness of solutions of an instance is an equally or more difficult task than solving the instance. Given an instance and its solution, finding another solution is called



© Masakazu Ishihata and Fumiya Tokumasu;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Experimental Algorithms (SEA 2022).

Editors: Christian Schulz and Bora Uçar; Article No. 2; pp. 2:1–2:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

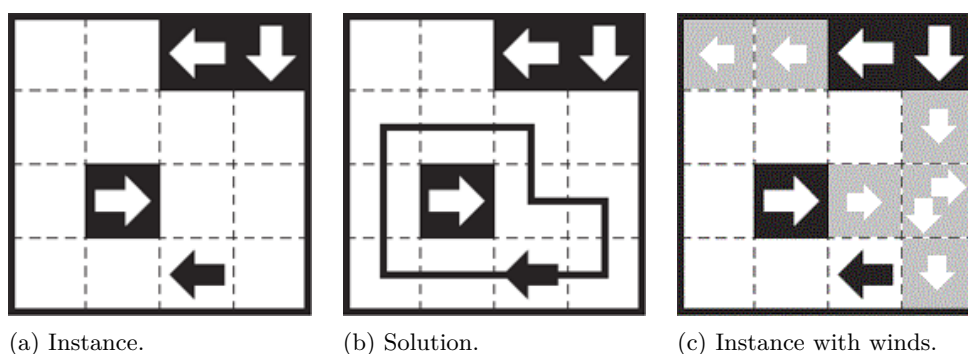
another solution problem (ASP) [21]. The ASP of an NP-complete problem is not necessary NP-complete; however, it has been shown that many puzzles are NP-complete not only in finding one solution but also in finding another solution [7, 10, 9].

In contrast to the theoretical difficulties of solving puzzles, practically efficient puzzle solvers have been proposed. One of the most popular approaches for solving puzzles is formulating a puzzle instance as a SAT problem, or its variants, including a constraint satisfaction problem (CSP) and satisfiability modulo theories (SMT), and solving it by a general constraint solver. For instance, Sugar [20], one of the latest CSP solvers, can solve a wide range of real-world instances of various puzzles, including a part of the above list [18]. Furthermore, some methods for generating puzzle instances have been proposed [22] that employ a SAT/CSP/SMT-based puzzle solvers as their building block to check the uniqueness of the generated instance; however, it has been reported that such a generator is too slow to generate a realistic instance (e.g., a  $10 \times 10$  grid) because it calls the solver so many times.

Another promising approach to solving puzzles is using zero-suppressed decision diagrams (ZDDs) [14]. A ZDD is a compact graph representation of a set family and provides a variety of queries, including counting, sampling, and set operations, in linear time for its size. The frontier-based search (FBS) [11] is a meta-algorithm for constructing a ZDD representing constrained subgraphs of a target graph. Many FBS examples for various constraints have been proposed, e.g., trees, cycles, simple paths, and more complex constraints [12, 15]. Once a puzzle is formulated as a constrained subgraph finding problem, one can construct a ZDD-based puzzle solver by designing the FBS for the problem. For instance, Slitherlink is a puzzle played on a graph  $G = (V, E)$  to find a single cycle  $C \subseteq E$  consistent with given all *hints*, where a hint is a pair of an edge set  $H \subseteq E$  and a positive number  $n$  and restricts  $C$  to  $|C \cap H| = n$ . Hence, Slitherlink can be formulated as a cycle finding problem with some cardinality constraints of some edge sets. It has been reported that a ZDD-based Slitherlink solver [24] performs faster than a CSP-based one [19], even though the former implicitly enumerates all the solutions, whereas the latter finds only one solution. In addition, the ZDD-based solver is helpful to generate puzzle instances because it can compute the number of solutions by the counting query of ZDDs. ZDD-based puzzle instance generators have been proposed for Slitherlink [24], Numberlink [24], and Minesweeper [17].

We focus on a puzzle called Nagareru [16], which has recently been proven to be NP-complete to solve and find another solution [9], and propose practically efficient methods for solving and generating its instance. Nagareru is a puzzle to draw a cycle that satisfies certain constraints like Slitherlink, but the cycle must have a global orientation consistent with some local orientation constraints (detailed rules are explained later), unlike Slitherlink. The FBS for Nagareru cannot be realized by combining existing FBS examples; namely, a new FBS is desired to solve Nagareru puzzles. The main contributions of this paper are threefold. First, we propose a ZDD-based Nagareru solver; we formulate Nagareru as a constraint cycle finding problem and propose the FBS for the constraints. Second, we propose an efficient Nagareru instance generator that employs the ZDD-based solver as its building block. Third, we empirically show that our solver outperforms a CSP-based solver and also that our generator creates an interesting instance in a reasonable time. Note that our generator is very different from those for other puzzles because the definition of “interesting” depends strongly on the target puzzle.

The rest of this paper is organized as follows: In Section 2, we review the rules of Nagareru and formulate it as a constrained cycle finding problem. We formulate a problem to find a constrained cycle as CSP in Section 3. In Section 4, we propose a new FBS for constructing a ZDD that implicitly enumerates all the constrained cycles; namely, it represents all the



■ **Figure 1** The grid board (a) is an instance of Nagareru, and the grid board (b) indicates its solution. The grid board (c) is the same instance as (a) with gray cells representing winds.

solutions of a Nagareru instance. In addition, we propose a new efficient generator of an “interesting” Nagareru instance that employs our ZDD-based solver to check the uniqueness of the solution in Section 5. We show the experimental results of the above methods in Section 6 and then conclude this paper in Section 7.

## 2 Problem Definition

### 2.1 Nagareru Puzzles

Nagareru is a paper-and-pencil puzzle played according to the following rules on a grid board [16]:

1. Draw a line to make a single continuous loop.
2. The line passes through the centers of cells, horizontally, vertically, or turning. It cannot cross itself, branch off, or go through the same cell twice.
3. The line must go through the white cells with a black arrow, and when you go along the arrows of the loop, that becomes the direction of all of the loop.
4. The white arrows in the black cells show that wind is blowing in the direction of the arrow till it reaches another black cell or the border. In cells where the wind blows, the line cannot advance against the wind.
5. When the line enters a cell where the wind blows, it must move at least one cell in that direction. When the line is blown like this (bent by a side wind), it cannot progress to or enter cells to hit the borders or enter black cells.

For example, the left grid board of Figure 1 is an instance of Nagareru, and the middle grid board indicates its solution.

We begin by formulating an instance of Nagareru. For any positive integer  $n \in \mathbb{Z}_+$ , let  $[n] \equiv \{1, \dots, n\}$ . Given  $w, h \in \mathbb{Z}_+$ , a  $w \times h$  grid board consists of  $w$  columns and  $h$  rows; namely, it has  $w \times h$  cells. For any  $w' \in [w]$  and  $h' \in [h]$ , let  $i = w' + w(h' - 1)$  refer to the cell in the  $w'$ th column from the left and the  $h'$ th row from the top. For any  $i \in [wh]$ , let  $\text{adj}(i) \subset [wh]$  be a set of adjacent cells of  $i$ . Let  $D \equiv \{\text{Up}, \text{Down}, \text{Left}, \text{Right}, \text{No}\}$  denote a set of directions, where **No** indicates non-directional. For any  $d \in D \setminus \{\text{No}\}$ , let  $d^{-1} \in D$  denote the opposite direction of  $d$ . For any  $i, j \in [wh]$ , let  $\text{rel}(i, j) \in D$  denote the relative direction from  $i$  to  $j$  if  $i$  and  $j$  are adjacent each other, and  $\text{rel}(i, j) = \text{No}$  if otherwise. For any  $i \in [wh]$  and  $d \in D \setminus \{\text{No}\}$ , there exists at most one adjacent cell  $j \in \text{adj}(i)$  satisfying  $\text{rel}(i, j) = d$ , denoted by  $i_d$ , where  $i_d = \text{Null}$  denotes  $\text{rel}(i, j) \neq d$  for any  $j \in \text{adj}(i)$ . Then, an instance of Nagareru is defined as follows:

► **Definition 1** (An instance of Nagareru). Let  $W \subseteq [wh] \times (D \setminus \{\text{No}\})$  be white cells with (black) arrows and  $B \subseteq [wh] \times D$  be black cells with (white) arrows.  $P \equiv (w, h, W, B)$  is an instance of Nagareru on a  $w \times h$  grid board if  $P$  satisfies  $\forall \{(i, d), (i', d')\} \subseteq W \cup B, i \neq i'$ .

► **Theorem 2** (Hardness of finding a solution of Nagareru [9]). For input Nagareru instance  $P$ , checking whether  $P$  admits a solution or not is NP-complete.

Let  $\omega \equiv (\omega_1, \dots, \omega_L) \in [wh]^L$  be a cell sequence of  $L$ -length. For any black cell  $(i, d) \in B$  and  $j \in [wh]$ ,  $\omega$  is a *wind path* from  $i$  to  $j$  if  $\omega$  satisfies the following conditions:

- $\omega_1 = i$  and  $\omega_L = j$ ,
- $\forall l \in [L - 1], \forall d' \in D, (\omega_{l+1}, d') \notin B$ :  $\omega$  has no other black cell than  $(i, d)$ ,
- $\forall l \in [L - 1], \text{rel}(\omega_l, \omega_{l+1}) = d$ :  $\omega$  is a straight path of direction  $d$ .

Namely, the wind path  $\omega$  from  $i$  to  $j$  indicates that a wind of direction  $d$  goes from  $i$  to  $j$ . For any  $i \in [wh]$ , we use  $D_i^{\text{wind}} \subseteq D$  to denote the directions of winds blowing on  $i$ ; namely,  $d \in D_i^{\text{wind}}$  indicates that there exist a black cell  $(j, d) \in B$  and a wind path  $\omega$  of direction  $d$  from  $j$  to  $i$ . We here introduce new colors, *gray* and *colorless* to make the explanation easier: for any  $i \in [wh]$  such that  $\forall d \in D, (i, d) \notin W \cup B$ ,  $i$  is *gray* if  $D_i^{\text{wind}} \neq \emptyset$  and *colorless* if  $D_i^{\text{wind}} = \emptyset$ ; namely, a cell with no arrow is gray if it is blown, and colorless if otherwise. For example, the grid board (c) of Figure 1, obtained by adding gray and colorless to the grid board (a), consists of three black cells, one white cell, six gray cells, and six colorless cells.

## 2.2 Formulating Nagareru as a constrained cycle finding problem

We formulate the problem of finding a solution of a Nagareru instance as a constrained cycle finding problem on a graph representing the instance. Let  $C \equiv \{\text{White}, \text{Black}, \text{Gray}, \text{No}\}$  denote a set of colors, where **No** indicates colorless. For any set  $V$  and  $n \in \mathbb{Z}_+$ , let  $\binom{V}{n} \equiv \{S \subseteq V \mid |S| = n\}$ . For any  $i \in [wh]$  and  $d \in D$ , let  $E_{i,d}^\perp \equiv \{\{i, j\} \mid j \in \text{adj}(i), \text{rel}(i, j) \notin \{d, d^{-1}\}\}$  be edges of  $i$  that are orthogonal to direction  $d$ .

► **Definition 3** (A graph representation of a Nagareru instance). Let  $G \equiv \langle V, E, \text{col}, \text{dir} \rangle$  where  $V \subseteq [wh]$  is a vertex set,  $E \subseteq \binom{V}{2}$  is an edge set,  $\text{col} : V \rightarrow C$  defines the color of each vertex, and  $\text{dir} : V \rightarrow 2^D$  defines the direction set of each vertex.  $G$  represents a Nagareru instance  $P$  if  $G$  satisfies following conditions:

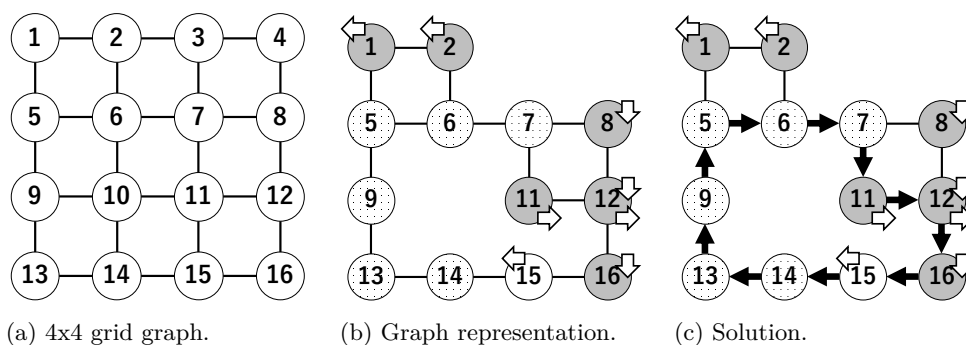
$$V \equiv [wh] \setminus \bigcup_{(i,d) \in B} \{i\}, \quad (1)$$

$$E \equiv \left\{ \{i, j\} \in \binom{V}{2} \mid j \in \text{adj}(i) \right\} \setminus \bigcup_{(i,d) \in W} E_{i,d}^\perp, \quad (2)$$

$$\text{col}(i) \equiv \begin{cases} \text{White} & \exists d \in D, (i, d) \in W \\ \text{Gray} & \forall d \in D, (i, d) \notin W, D_i^{\text{wind}} \neq \emptyset, \\ \text{No} & \forall d \in D, (i, d) \notin W, D_i^{\text{wind}} = \emptyset \end{cases}, \quad (3)$$

$$\text{dir}(i) \equiv \begin{cases} \{d\} & (i, d) \in W \\ D_i^{\text{wind}} & \text{otherwise} \end{cases}. \quad (4)$$

For any color  $c \in C$ , let  $V_c \equiv \{i \in V \mid \text{col}(i) = c\}$ . Equation (1) leads to  $V_{\text{Black}} = \emptyset$ . Equation (2) indicates that  $E$  has no edge inconsistent with (orthogonal to) an arrow of a white vertex. Figure 2(a) indicates a 4x4 grid graph, and Figure 2(b) is the graph representation of the instance shown in Figure 1.



■ **Figure 2** The graph (a) indicates a 4x4 grid graph with 16 vertices and 24 edges. The graph (b) is the graph representation of the instance shown in Figure 1, where white, gray, and dotted circles represent vertices colored by white, gray, and colorless, respectively, and white arrows attached to each vertex  $i$  indicate  $\text{dir}(i)$ . A directed cycle of bold black directed edges in the graph (c) forms a solution of Nagareru.

For any  $F \subseteq E$ , let  $V[F] \equiv \bigcup_{e \in F} e$  denote a set of all endpoints of  $F$ ,  $G[F] \equiv \langle V[F], F, \text{col}, \text{dir} \rangle$  denote an edge-induced subgraph of  $G$ ,  $\text{nei}_F(i) \equiv \{j \in V[F] \mid \{i, j\} \in F\}$  denote the neighbors of  $i$  on  $G[F]$ , and  $\text{deg}_F(i) \equiv |\text{nei}_F(i)|$  denote the degree of  $i$  on  $G[F]$ . Let  $n, m, n_F$ , and  $m_F$  denote  $|V|, |E|, |V[F]|$ , and  $|F|$ , respectively.

► **Definition 4** (A solution of a Nagareru instance). *An edge set  $F \subseteq E$  is a solution of a Nagareru instance  $P$  if there exists a permutation of  $V[F]$ , denoted by  $p \equiv (p_1, \dots, p_{n_F})$ , such that the following conditions are satisfied, where let  $p_0 \equiv p_{n_F}, p_{n_F+1} \equiv p_1, e_l \equiv \{p_l, p_{l+1}\}$ , and  $r_l \equiv \text{rel}(p_l, p_{l+1})$  for any  $l \in [n_F]$ .*

$$\forall i \in V_{\text{White}}, \exists l \in [n_F], p_l = i, \quad (5)$$

$$\forall l \in [n_F], \text{deg}_F(p_l) = 2 \wedge e_l \in F, \quad (6)$$

$$\forall l \in [n_F], \text{col}(p_l) = \text{White} \implies \forall d \in \text{dir}(p_l), r_{l-1} \neq d^{-1}, r_l \neq d^{-1}, \quad (7)$$

$$\forall l \in [n_F], \text{col}(p_l) = \text{Gray} \implies \forall d \in \text{dir}(p_l), r_{l-1} \neq d^{-1}, r_l \neq d^{-1}, \quad (8)$$

$$\forall l \in [n_F], \text{col}(p_l) = \text{Gray} \implies \forall d \in \text{dir}(p_l), \{e_{l-1}, e_l\} \neq E_{p_l, d}^\perp. \quad (9)$$

Equation (5) guarantees that every white vertex is contained in  $G[F]$ . Equation (6) restricts  $G[F]$  to be a cycle. Equation (7) (resp. (8)) prohibits  $G[F]$  from advancing against any wind of a white (resp. gray) vertex. Equation (9) prohibits  $G[F]$  from orthogonal to (crossing) any wind of a gray vertex. Consequently,  $G[F]$  forms a solution of  $P$ . Let  $\mathcal{F}_P \subseteq 2^E$  be the set of all the solutions of  $P$ .  $P$  is said to be *invalid*, *valid*, and *good* if it has no solution ( $|\mathcal{F}_P| = 0$ ), at least one solution ( $|\mathcal{F}_P| \geq 1$ ), exactly one solution ( $|\mathcal{F}_P| = 1$ ), respectively. For example, let  $P$  be a Nagareru instance shown in Figure 1,  $G$  be its graph representation shown in Figure 2(b), and  $F$  be the set of bold black arrows (edges) in Figure 2(c). Then,  $G[F]$  forms a solution of  $P$ , and  $P$  admits no other solution; namely,  $P$  is a good instance.

### 3 A CSP-based Nagareru solver

We propose a baseline method for finding a solution of a Nagareru instance. The method consists of three steps: (1) translating a constrained subgraph finding problem on a Nagareru instance as a CSP instance, (2) solving the CSP instance by a CSP solver, and (3) converting the obtained CSP solution to the solution of Nagareru.

A CSP instance is denoted by a triplet of variables, variable domains, and constraints. We first introduce variables and their domains. Because  $G[F]$  for any  $F \in \mathcal{F}_P$  forms a directed cycle, we introduce the same idea of a CSP formulation of Slitherlink [19] to represent a cycle constraint; introducing auxiliary variables to represent a visiting order of variables that forms a cycle. Let  $\mathbf{D}_E \equiv \{(i, j) \mid i < j, \{i, j\} \in F\}$ . Then, our CSP formulation contains following four types of variables and domains:  $u_{i,j} \in \{-1, 0, +1\}$ ,  $d_i \in \{0, 2\}$ ,  $q_i \in \{0, 1, \dots, m\}$ , and  $s_i \in \{0, 1\}$  for any  $(i, j) \in \mathbf{D}_E$ .  $u_{i,j}$  denotes the use of edge  $\{i, j\}$  ( $i < j$ ) with direction:  $u_{i,j} = 0$  indicates  $\{i, j\} \notin F$  and  $u_{i,j} = +1$  (resp.  $-1$ ) indicates  $\{i, j\} \in F$  with the forward direction  $\text{rel}(i, j)$  (resp. backward direction  $\text{rel}(j, i)$ ).  $d_i$  denotes  $\deg_F(i)$ . A set of  $q_i$  denotes a permutation of  $V[F]$ :  $q_i = 0$  indicates  $i \notin V[F]$  and the rest  $q_i$  define a permutation  $p = (p_1, \dots, p_{n_F})$  such as  $p_{q_i} = i$ .  $s_i$  denotes the starting vertex of the permutation  $p$ :  $q_i = 1$  is represented by  $s_i = 1$  and  $s_{i'} = 0$  for  $i' \in [wh] \setminus \{i\}$ .

We then introduce constraints such that an assignment of the above variables satisfying the conditions if-and-only-if  $P$  is valid; in other words, we describe Equation (5), (6), (7), (8), and (9) of Definition 4 as formulas of the above variables. We first introduce the following sets of doublets and triplets of indices:

$$\begin{aligned} \mathbf{D}_i &\equiv \{(j, k) \in \mathbf{D}_E \mid i \in \{j, k\}\}, \\ \mathbf{F}_c &\equiv \{(i, j) \in \mathbf{D}_E \mid \exists k \in \{i, j\}, \text{col}(k) = c, \text{rel}(i, j) \in \text{dir}(k)\}, \\ \mathbf{B}_c &\equiv \{(i, j) \in \mathbf{D}_E \mid \exists k \in \{i, j\}, \text{col}(k) = c, \text{rel}(j, i) \in \text{dir}(k)\}, \\ \mathbf{T}_\perp &\equiv \{(i, j, k) \mid (i, j), (j, k) \in \mathbf{D}_E, \text{col}(j) = \text{Gray}, \exists d \in \text{dir}(j), \{\{i, j\}, \{j, k\}\} = E_{j,d}^\perp\}. \end{aligned}$$

Then, the constraints of our CSP formulation are follows:

$$\bigwedge_{i \in V_{\text{White}}} (q_i > 0), \quad (10)$$

$$\left( \bigwedge_{i \in V} \left( d_i = \sum_{(j,k) \in \mathbf{D}_i} |u_{j,k}| \right) \right) \wedge \left( \bigwedge_{i \in V} \left( \sum_{j: (j,i) \in \mathbf{D}_i} u_{j,i} = \sum_{j: (i,j) \in \mathbf{D}_i} u_{i,j} \right) \right), \quad (11)$$

$$\bigwedge_{i \in V} (q_i > 0 \implies d_i > 0) \wedge (q_i = 1 \iff s_i = 1), \quad (12)$$

$$\bigwedge_{i \in V} (u_{i,j} = +1 \implies (q_i + 1 = q_j \vee q_j = 1)), \quad (13)$$

$$\bigwedge_{i \in V} (u_{i,j} = -1 \implies (q_i = q_j + 1 \vee q_i = 1)), \quad (14)$$

$$\sum_{i \in V} s_i = 1, \quad (15)$$

$$\left( \bigwedge_{(i,j) \in \mathbf{F}_{\text{White}}} (u_{i,j} \neq -1) \right) \wedge \left( \bigwedge_{(i,j) \in \mathbf{B}_{\text{White}}} (u_{i,j} \neq +1) \right), \quad (16)$$

$$\left( \bigwedge_{(i,j) \in \mathbf{F}_{\text{Gray}}} (u_{i,j} \neq -1) \right) \wedge \left( \bigwedge_{(i,j) \in \mathbf{B}_{\text{Gray}}} (u_{i,j} \neq +1) \right), \quad (17)$$

$$\bigwedge_{(i,j,k) \in \mathbf{T}_\perp} (u_{i,j} = 0 \vee u_{j,k} = 0), \quad (18)$$

Equation (10) corresponds to Equation (5). Equation (11) defines  $d_i = \deg_F(i)$ . Equation (12), (13), (14), and (15) jointly represent Equation (6). Equation (16), (17), and (18) correspond to Equation (7), (8), and (9), respectively.

► **Proposition 5** (A CSP formulation of Nagareru). *The triplet of the above variables, domains, and constraints is a CSP formulation for finding a solution  $F \in \mathcal{F}_P$  of a Nagareru instance  $P$ , and  $F$  is constructed from a CPS solution as  $F = \{\{i, j\} \mid (i, j) \in \mathbf{D}_E, u_{i,j} \neq 0\}$ .*

## 4 A ZDD-based Nagareru Solver

We here propose a ZDD-based Nagareru solver that constructs a ZDD representing  $\mathcal{F}_P$ . We first review a ZDD, a compact graph expression of a set family, and FBS, a meta-algorithm

to construct a ZDD for constrained subgraphs. Then, we propose a new FBS for a ZDD of  $\mathcal{F}_P$ .

#### 4.1 ZDDs for subgraphs

A ZDD  $Z$  is a compact graph representation of a set family over a universe set  $E$  and let  $\mathcal{F}_Z \subseteq 2^E$  be the set family represented by  $Z$ . When the universe set  $E$  is an edge set of a graph  $G \equiv \langle V, E \rangle$ ,  $Z$  can be regarded as a set of edge-induced subgraphs  $G[F]$  for each  $F \in \mathcal{F}_Z$ . To avoid confusing two graphs  $Z$  and  $G$ , we use terms *nodes* and *arcs* for describing  $Z$ . A ZDD requires a total order on  $E$  denoted by  $\succ$  and let  $e_l$  denote the  $l$ th smallest element of  $E$ . Then, a ZDD  $Z$  and its set family  $\mathcal{F}_Z$  are defined as follows.

► **Definition 6** (A ZDD). *Let  $Z \equiv \langle N, A_0, A_1, \ell \rangle$  where  $N$  is a set of nodes,  $A_b \subseteq N \times N$  is a set of  $b$ -arcs for any  $b \in \{0, 1\}$ , and  $\ell : N \rightarrow E \cup \{\text{Null}\}$  defines the label of each node.  $Z$  is a ZDD if it satisfies the following conditions:*

- $N$  has exactly one root node denoted by  $\rho$  and exactly two terminal nodes denoted by  $\tau_0$  and  $\tau_1$  such as  $\ell(\tau_0) = \ell(\tau_1) = \text{Null}$ .
- Each non-terminal node  $\nu \in N \setminus \{\tau_0, \tau_1\}$  has exactly one outgoing  $b$ -arc for each  $b \in \{0, 1\}$ , and is labeled by some element of  $E$ :  $\ell(\nu) \in E$ .
- For any arc  $(\nu, \nu') \in A_0 \cup A_1$ ,  $\ell(\nu) \succ \ell(\nu')$  holds where let  $e \succ \text{Null}$  for any  $e \in E$ .

► **Definition 7** (A set family represented by a ZDD). *For any non-terminal node  $\nu \in N \setminus \{\tau_0, \tau_1\}$  and  $b \in \{0, 1\}$ , let  $\nu_b$  be the node pointed by the  $b$ -arc of  $\nu$ , referred to as the  $b$ -child of  $\nu$ . For any  $\nu \in N$ , let  $\mathcal{F}_\nu \subseteq 2^E$  be a set family recursively-defined as*

$$\mathcal{F}_{\tau_0} \equiv \emptyset, \quad \mathcal{F}_{\tau_1} \equiv \{\emptyset\}, \quad \mathcal{F}_\nu \equiv \mathcal{F}_{\nu_0} \cup \{F \cup \{\ell(\nu)\} \mid F \in \mathcal{F}_{\nu_1}\}.$$

Then,  $Z$  is said to represent  $\mathcal{F}_\rho$  denoted by  $\mathcal{F}_Z$ .

Definition 6 restricts  $Z$  to a rooted directed cyclic graph (DAG). Definition 7 defines  $\mathcal{F}_\nu$  in a bottom-up manner; however, it has another intuitive definition as follows. Let  $\pi \equiv (\pi_1, \dots, \pi_L) \in N^L$  be a directed path from  $\pi_1$  to  $\pi_L$  on  $Z$  of  $L$ -length, and also let  $F_\pi \equiv \{\ell(\pi_l) \mid l \in [L-1], (\pi_l, \pi_{l+1}) \in A_1\}$ ; namely,  $F_\pi$  contains  $\ell(\nu)$  if  $\pi$  contains the 1-arc of  $\nu$ . Let  $\Pi_{\nu \rightarrow \nu'}$  be a set of all directed path from  $\nu$  to  $\nu'$  on  $Z$ , and also let  $\mathcal{F}_{\nu \rightarrow \nu'} \equiv \{F_\pi \mid \pi \in \Pi_{\nu \rightarrow \nu'}\}$ . Then, for any  $\nu \in N$ ,  $\mathcal{F}_\nu = \mathcal{F}_{\nu \rightarrow \tau_1}$  holds [14]. Consequently, checking  $|\mathcal{F}_Z| > 0$  corresponds to finding a directed path from  $\rho$  to  $\tau_1$  on  $Z$ , and counting  $|\mathcal{F}_Z|$  is equivalent to counting such paths. Since  $Z$  is a rooted DAG, dynamic programming solves both tasks in  $O(|N|)$  time.

#### 4.2 FBS for constrained subgraphs

A ZDD  $Z$  is said to represent constraint subgraphs of  $G$  if  $G[F]$  for all  $F \in \mathcal{F}_Z$  satisfies the target constraint but not for any  $F \notin \mathcal{F}_Z$ . FBS is a meta-algorithm to construct such  $Z$ . The basic idea of FBS is layer-wise top-down construction. For any  $l \in [m]$ , let  $N_l$  be non-terminal nodes labeled by  $e_l$ , and also let  $l$  be referred to as *layer*. FBS initializes  $N_m \equiv \{\rho\}$  and repeats the generation of  $N_{l-1}$  from  $N_l$  in order from the top layer  $m$  to the bottom layer 1, and the resulting  $N = (\cup_{l \in [m]} N_l) \cup \{\tau_0, \tau_1\}$  forms a ZDD. The essential idea of FBS is introducing a *state* to each node  $\nu$ , denoted by  $S_\nu$ . Each  $S_\nu$  is a set of some variables, and its specific definition depends on the constraint of interest. We use  $S_\nu.x$  to denote the value of the variable  $x$  in  $S_\nu$ . Given  $\nu \in N_l$  and its state  $S_\nu$ , its  $b$ -child  $\nu_b$  and its state  $S_{\nu_b}$  is generated by *only* using the information of  $S_\nu$  and  $G$ . In other words, all the necessary information to create the children of  $\nu$  should be concentrated in  $S_\nu$ . In addition,

---

**Algorithm 1** ConstructZDD.

---

```

1: Let  $\rho$  be a new node and  $\ell(\rho) \leftarrow m$ .  $\triangleright$  Initialize the root
2:  $N_m \leftarrow \{\rho\}$ ,  $N_l \leftarrow \emptyset$  for  $l \in [m-1]$   $\triangleright$  Initialize the node set  $N$ 
3:  $A_b \leftarrow \emptyset$  for  $b \in \{0, 1\}$   $\triangleright$  Initialize the arc sets  $A_0$  and  $A_1$ 
4: for  $l = m, \dots, 1$  do  $\triangleright$  The layer-wise top-down construction
5:   for  $\nu \in N_l$  do
6:     for  $b \in \{0, 1\}$  do
7:        $\nu_b \leftarrow \text{getChild}(S_\nu, l, b)$   $\triangleright$  Create  $\nu_b$ , the  $b$ -child of  $\nu$ 
8:       if  $\nu_b \in \{\tau_0, \tau_1\}$  then  $\triangleright$  Branching
9:          $\triangleright$  do nothing  $\triangleleft$ 
10:      else if  $\exists \nu' \in N_{l-1} : S_{\nu_b} = S_{\nu'}$  then  $\triangleright \nu_b$  has an identical existing node  $\nu'$ 
11:         $\nu_b \leftarrow \nu'$   $\triangleright$  Merge  $\nu'$  and  $\nu_b$ 
12:      else  $\triangleright \nu_b$  has no identical existing node
13:         $\ell(\nu_b) \leftarrow l - 1$ 
14:         $N_{l-1} \leftarrow N_{l-1} \cup \{\nu_b\}$   $\triangleright$  Add  $\nu_b$  to  $N_{l-1}$  as a new node
15:         $A_b \leftarrow A_b \cup \{(\nu, \nu_b)\}$   $\triangleright$  Add  $(\nu, \nu_b)$ , the  $b$ -arc of  $\nu$ , to  $A_b$  as a new arc
16:  $N \leftarrow \left( \bigcup_{l \in [m]} N_l \right) \cup \{\tau_0, \tau_1\}$   $\triangleright$  Unite all layers
17: return  $\langle N, A_0, A_1, \ell \rangle$ 

```

---

nodes with the same state must have the same children. Hence, such identical nodes in the same layer can be merged into a single node. Algorithm 1 is the pseudo-code of FBS, where a subroutine  $\text{getChild}(S_\nu, l, b)$ , which returns the  $b$ -child of  $\nu$  with its state, is defined depend on the target constraint.

► **Theorem 8** (The complexity of FBS [11]). *For any  $l \in [m]$ , let  $\kappa_l$  be the number of different realizations of the state at the  $l$ th layer; namely,  $|N_l| = \kappa_l$  holds. Let  $\kappa \equiv \max_{l \in [m]} \kappa_l$ . Then, the space and time complexity of FBS is  $O(m\kappa)$  under the assumption that the identical state can be found in  $O(1)$  time.*

#### 4.2.1 Example: the size constraint

Let us consider the FBS for the size constraint  $|F| \leq K$ . For any non-terminal node  $\nu$ , let  $S_\nu$  consist only of an integer variable  $x_{\text{used}}$  that indicates the number of passed 1-arcs from the root  $\rho$  to  $\nu$ ; namely,  $|F_\pi| = S_\nu.x_{\text{used}}$  holds for any  $\pi \in \Pi_{\rho \rightarrow \nu}$  of  $Z$  under construction.  $S_\nu.x_{\text{used}} = K$  indicates that  $F_\pi$  cannot adopt any more edges. Two nodes  $\nu$  and  $\nu'$  are equivalent if  $S_\nu = S_{\nu'}$  because the paths from  $\rho$  to them passed the same number of 1-arcs.

In summary, Algorithm 2 describes  $\text{getChild}(S, l, b)$  of the size constraint. Since  $\kappa_l \leq K$  holds for any  $l \in [m]$ , the resulting ZDD size is  $O(mK)$  by Theorem 8.

---

**Algorithm 2**  $\text{getChild}(S, l, b)$  for a size constraint  $|F| \leq K$ .

---

```

1: Let  $S'$  be a new state.
2:  $S'.x_{\text{used}} \leftarrow (l = m) ? 0 : S.x_{\text{used}}$   $\triangleright$  Initialize & Copy
3: if  $b = 1$  then  $\triangleright$  Adopt  $e_l$ 
4:    $S'.x_{\text{used}} \leftarrow S'.x_{\text{size}} + 1$   $\triangleright$  Update  $S'$ 
5:   return  $\tau_0$  if  $S'.x_{\text{size}} > K$   $\triangleright$  Pruning: detect  $|F| > K$ 
6: return  $\tau_1$  if  $l = 1$   $\triangleright$  Termination: reach the end without the violation
7: return a new node  $\nu$  with  $S_\nu = S'$ 

```

---



■ **Algorithm 3** getChild( $S, l, b$ ) for a cycle constraint.

---

```

1: Let  $e_l = \{i, j\}$ ,  $S'$  be a new state.
2:  $S'.m_k \leftarrow S.m_k$  for each  $k \in V_l \cap V_{l+1}$  ▷ Copy
3:  $S'.m_k \leftarrow k$  for each  $k \in V_l \setminus V_{l+1}$  ▷ Initialize
4:  $m_k \leftarrow S'.m_k$  for each  $k \in V_l$  ▷ Abbreviate
5: if  $b = 1$  then ▷ Adopte  $e_l$ 
6:   return  $\tau_0$  if  $m_i = \text{Null} \vee m_j = \text{Null}$  ▷ Pruning: detect a branching
7:   if  $m_i = j \wedge m_j = i$  then ▷ Detect a cycle
8:     return  $\tau_0$  if  $\exists k \in V_l \setminus e_l, m_k \in V_k \setminus \{k\}$  ▷ Pruning: detect a redundant endpoint
9:     return  $\tau_1$  ▷ Termination: complete a single cycle
10:  ▷ Update  $S'$  ◁
11:   $S'.m_{m_i} \leftarrow m_j$ 
12:   $S'.m_{m_j} \leftarrow m_i$ 
13:   $S'.m_i \leftarrow \text{Null}$  if  $S'.m_j \neq i$ 
14:   $S'.m_j \leftarrow \text{Null}$  if  $S'.m_i \neq j$ 
15: return  $\tau_0$  if  $\exists k \in V_l \setminus V_{l-1}, S'.m_k \in V_{l-1}$  ▷ Pruning: detect a leaving endpoint
16: return  $\tau_0$  if  $l = 1$  ▷ Pruning: reach the end without completing a cycle
17: return a new node  $\nu$  with  $S_\nu = S'$ 

```

---

## 4.2.2 Example: the cycle constraint

Let us consider the FBS for the cycle constraint. For each layer  $l \in [m]$ , let  $E_{\leq l} \equiv \{e_{l'} \in E \mid l' \leq l\}$ ,  $E_{\geq l} \equiv \{e_{l'} \in E \mid l' \geq l\}$ ,  $V_l \equiv \{i \in V \mid \exists e \in E_{\leq l}, \exists e' \in E_{\geq l}, i \in e \cap e'\}$ , and  $\lambda \equiv \max_{l \in [m]} |V_l|$ , where  $V_l$  and  $\lambda$  are referred to as the *frontier* of the  $l$ th layer and the maximum frontier size, respectively.

For each non-terminal node  $\nu \in N_l$ , let  $S_\nu \equiv \{m_k \mid k \in V_l\}$  where  $m_k \in V_l \cup \{\text{Null}\}$  is referred to as the *mate* of  $k$ . The mate  $m_k$  indicates the connectivity of  $k$  in  $G[F_\pi]$  for any  $\pi \in \Pi_{\rho \rightarrow \nu}$ : (1)  $m_k = k$  indicates that  $k$  is an isolated fragment (vertex), (2)  $m_k \in V_l \setminus \{k\}$  indicates that  $k$  and  $m_k$  are the endpoints of a path fragment, and (3)  $m_k = \text{Null}$  indicates that  $k$  is an intermediate vertex of a path fragment.

We next consider what will happen if an edge  $e_l = \{i, j\}$  is adopted to  $F_\pi$ . If  $m_i = \text{Null} \vee m_j = \text{Null}$  (i.e.,  $i$  and/or  $j$  is an intermediate vertex of a path fragment), adopting  $e_l$  violates the cycle condition because it causes a branching. Otherwise, it connects two fragments to which  $i$  belongs and  $j$  belongs. More specifically, if  $m_i \neq j \wedge m_j \neq i$  (i.e.,  $i$  and  $j$  belong to different fragments), it constructs a new path fragment whose endpoints are  $m_i$  and  $m_j$ . If  $m_i = j \wedge m_j = i$  (i.e.,  $i$  and  $j$  are the endpoints of the same path fragment), it completes a cycle; in addition, if  $G[F_\pi]$  has no redundant path fragment (i.e.,  $\forall k \in V_l \setminus e_l, m_k \notin V_l \setminus \{k\}$ ),  $G[F_\pi]$  forms a single cycle. Similarly, leaving an endpoint  $k$  (i.e.,  $m_k \in V_l \setminus \{k\}$ ) from the frontier violates the cycle constraint because  $k$  has no chance to join a cycle anymore; namely,  $k$  is fixed as an endpoint of a redundant path fragment.

In summary, Algorithm 3 describes getChild( $S, l, b$ ) of the cycle constraint. Since  $S_\nu$  corresponds to a matching in the complete graph  $(V_l, \binom{V_l}{2})$ ,  $\kappa_l \leq 2^{|V_l|}$  holds and the resulting ZDD size is  $O(m2^{\lambda^2})$  by Theorem 8. In practice, however,  $S_\nu$  does not take as many realizations as  $2^{\lambda^2}$ , the actual ZDD size is empirically much smaller.

### 4.3 The FBS for the Nagareru constraints

We here propose the FBS for the Nagareru constraints shown in Definition 4. For any non-terminal node  $\nu$ , let  $S_\nu$  consist of three types of variables:  $m_k \in V_l \cup \{\text{Null}\}$ ,  $u_k \in \{0, 1\}$ , and  $d_k \in D \cup \{\text{Null}\}$  for any  $k \in V_l$ .  $m_k$  is the exactly same as the mate of Example 2 and indicates the connectivity of  $k$  on  $G[F_\pi]$ .  $u_k$  indicates the upper stream of path fragments on  $G[F_\pi]$ .  $d_k$  indicates the relative direction from the neighbor of  $k$  to  $k$  on  $G[F_\pi]$ . More specifically,  $u_k$  and  $d_k$  are defined as follows: If  $m_k \notin V_l \setminus \{k\}$  (i.e.,  $k$  is an isolated vertex or an intermediate vertex of a path fragment), let  $u_k = 0$  and  $d_k = \text{Null}$ . If  $m_k \in V_l \setminus \{k\}$  (i.e.,  $k$  is an endpoint of a path fragment on  $G[F_\pi]$ ), let  $u_k = 1$  indicate that  $k$  must be upper stream of a path fragment whose endpoints are  $k$  and  $m_k$ , and also let  $d_k \equiv \text{rel}(k', k)$  where  $\{k', k\} \in F_\pi$ . By regarding  $u_k$  as a Boolean variable,  $u_k \wedge u_{m_k}$  must always be false and  $\neg u_k \wedge \neg u_{m_k}$  indicates that the upper stream of the path fragment is not yet decided.

Let us consider what will happen if  $e_l = \{i, j\}$  is adapted to  $F_\pi$ . As Example 2, if  $m_i \neq j \wedge m_j \neq i$ , adapting  $e_l$  connects two different path fragments on  $G[F_\pi]$  and constructs a new path fragment whose endpoints are  $m_i$  and  $m_j$ . If  $u_i \wedge u_j$  (resp.  $u_{m_i} \wedge u_{m_j}$ ), it corresponds to connecting up streams (resp. down streams) of the two path fragments on  $G[F_\pi]$ ; namely,  $G[F_\pi]$  has no consistent direction. If  $u_i \vee u_{m_j}$  (resp.  $u_j \vee u_{m_i}$ ), the up stream of the resulting path fragment should be  $m_j$  (resp.  $m_i$ ). In addition, the direction of  $e_l$  decided by its colored endpoints must be consistent with this direction. If  $\text{col}(i) = \text{White} \vee \text{col}(j) = \text{White}$ ,  $e_l$  must be adapted to satisfy Equation (5) of Definition 4. Let  $l_{\text{White}} \equiv \min\{l \in [m] \mid \exists k \in e_l, \text{col}(k) = \text{White}\}$ . Then, completing a cycle before reaching  $l_{\text{White}}$ th layer deduces that at least one edge with a white endpoint is unused; namely, Equation (5) is violated. In summary, Algorithm 4 describes  $\text{getChild}(S, l, b)$  of the Nagareru constraints shown in Definition 4.

The complexity of the proposed FBS is following: Because  $\kappa_l$  is less than  $2^{|V_l|^2} \times 2^{|V_l|} \times |D|^{|V_l|}$ , the product of the domain size of each variable in the state,  $\kappa_l = O(2^{|V_l|^2})$  holds. Hence, its complexity is  $O(m2^{\lambda^2})$  that is the same as Example 2, where  $\lambda$  depends on  $G$  and the total order  $\succ$  on  $E$ . For instance, when  $\succ$  is defined as  $e \succ e' \Leftrightarrow (\min e < \min e') \vee (\min e = \min e' \wedge \max e < \max e')$ ,  $\lambda$  of an  $n \times n$  grid graph is  $n$ .

► **Proposition 9** (The FBS for the Nagareru constraints). *Given a Nagareru instance  $P$ , the FBS shown in Algorithm 1 with  $\text{getChild}(S, l, b)$  shown in Algorithm 4 constructs a ZDD representing  $\mathcal{F}_P$ , a set of all the solutions of  $P$  defined by Definition 4. The complexity of the proposed FBS and the resulting ZDD size is  $O(m2^{\lambda^2})$ .*

## 5 An efficient Nagareru instance generator

In this section, we first define the “interesting” instance of Nagareru and propose an efficient Nagareru instance generator that generates interesting instances using our ZDD-based Nagareru solver as its building blocks.

### 5.1 An interesting instance of Nagareru

Let us begin by defining an *interesting* instance. Given an instance  $P$  and its graph  $G$  of Definition 3, we introduce *infeasible*, *ineffective*, and *redundant* cells as follows: A white cell  $(i, d) \in W$  is infeasible if  $D_i^{\text{wind}} \not\subseteq \text{dir}(i) \vee \deg_E(i) < 2$ ; namely, there exists a wind inconsistent to its arrow or there is not enough number of neighbors to follow its arrow. Consequently,  $P$  with an infeasible cell has no solution. A black cell  $(i, d) \in B$  is ineffective if

■ **Algorithm 4** getChild( $S, l, b$ ) for the Nagareru constraints.

---

```

1: ▷ Initialize  $\mathcal{E}$  Copy                                <
2: Let  $S'$  be a new state.
3:  $S'.m_k \leftarrow k, S'.u_k \leftarrow 0, S'.d_k \leftarrow \text{Null}$  for each  $k \in V_l \setminus V_{l+1}$ 
4:  $S'.m_k \leftarrow S.m_k, S'.u_k \leftarrow S.u_k, S'.d_k \leftarrow S.d_k$  for each  $k \in V_l \cap V_{l+1}$ ,
5: ▷ Abbreviate                                        <
6: Let  $e_l = \{i, j\}$ .
7:  $m_k \leftarrow S'.m_k, u_k \leftarrow S'.u_k, d_k \leftarrow S'.d_k$  for each  $k \in V_l$ 
8: ▷ Direction that  $e_l$  must follow                    <
9:  $d_l \leftarrow \text{Null}$ 
10:  $d_l \leftarrow \text{rel}(i, j)$  if  $u_{m_i} \vee u_j$ 
11:  $d_l \leftarrow \text{rel}(j, i)$  if  $u_{m_j} \vee u_i$ 
12: if  $b = 1$  then                                     ▷ Adopt  $e_l$ 
13:   ▷ Pruning: check Equation (6) of Definition 4     <
14:   return  $\tau_0$  if  $m_i = \text{Null} \vee m_j = \text{Null}$ 
15:   return  $\tau_0$  if  $(u_i \wedge u_j) \vee (u_{m_i} \wedge u_{m_j})$ 
16:   ▷ Pruning: check Equation (7) of Definition 4     <
17:   return  $\tau_0$  if  $\text{col}(i) = \text{White} \wedge d_l^{-1} \in \text{dir}(i)$ 
18:   return  $\tau_0$  if  $\text{col}(j) = \text{White} \wedge d_l^{-1} \in \text{dir}(j)$ 
19:   ▷ Pruning: check Equation (8) of Definition 4     <
20:   return  $\tau_0$  if  $\text{col}(i) = \text{Gray} \wedge d_l^{-1} \in \text{dir}(i)$ 
21:   return  $\tau_0$  if  $\text{col}(j) = \text{Gray} \wedge d_l^{-1} \in \text{dir}(j)$ 
22:   ▷ Pruning: check Equation (9) of Definition 4     <
23:   return  $\tau_0$  if  $\text{col}(i) = \text{Gray} \wedge d_i = \text{rel}(i, j) \wedge (\exists d \in \text{dir}(i), d_i \notin \{d, d^{-1}\})$ 
24:   return  $\tau_0$  if  $\text{col}(j) = \text{Gray} \wedge d_j = \text{rel}(j, i) \wedge (\exists d \in \text{dir}(j), d_j \notin \{d, d^{-1}\})$ 
25:   if  $m_i = j \wedge m_j = i$  then                       ▷ Detect a cycle
26:     return  $\tau_0$  if  $l > l_{\text{White}}$                    ▷ Pruning: detect a unused white vertex
27:     return  $\tau_0$  if  $\exists k \in V_l \setminus e_l, m_k \in V_l \setminus \{k\}$  ▷ Pruning: detect a redundant endpoint
28:     return  $\tau_1$                                      ▷ Termination: complete a solution
29:   ▷ Update  $S'$                                        <
30:    $S'.m_{m_i} \leftarrow m_j, S'.m_{m_j} \leftarrow m_i,$ 
31:    $S'.m_i \leftarrow \text{Null}$  if  $m_i \neq i$ 
32:    $S'.m_j \leftarrow \text{Null}$  if  $m_j \neq j$ 
33:    $S'.d_i \leftarrow (m_i = i) ? j : \text{Null}$ 
34:    $S'.d_j \leftarrow (m_j = j) ? i : \text{Null}$ 
35:    $S'.u_{m_i} \leftarrow u_j, S'.u_{m_j} = u_i$ 
36:    $S'.u_{m_i} = 1$  if  $\exists k \in e_l, (\text{col}(k) \in \{\text{White}, \text{Gray}\}) \wedge (\text{rel}(i, j) \in \text{dir}(k))$ 
37:    $S'.u_{m_j} = 1$  if  $\exists k \in e_l, (\text{col}(k) \in \{\text{White}, \text{Gray}\}) \wedge (\text{rel}(j, i) \in \text{dir}(k))$ 
38: else                                               ▷ Does not adopt  $e_l$ 
39:   ▷ Pruning: check Equation (5) of Definition 4     <
40:   return  $\tau_0$  if  $\text{col}(i) = \text{White} \vee \text{col}(j) = \text{White}$ 
41: return  $\tau_0$  if  $\exists k \in V_l \setminus V_{l-1}, m_k \in V_{l-1}$  ▷ Pruning: detect a leaving endpoint
42: return  $\tau_0$  if  $l = 1$                              ▷ Pruning: reach the end without completing a cycle
43: return a new node  $\nu$  with  $S_\nu = S'$ 

```

---

$i_d = \text{Null}$ ; namely, there is no cell affected by its wind. A white or black cell  $(i, d) \in W \cup D$  is redundant if  $|\mathcal{F}_P| = |\mathcal{F}_{P'}|$  where  $P'$  is obtained by removing  $(i, d)$  from  $P$ ; namely, removing it does not change the number of solutions. We define that  $P$  is *interesting* if  $P$  is *good* (i.e.,  $|\mathcal{F}_P| = 1$ ) and contains neither infeasible, ineffective, nor redundant cell.

## 5.2 The proposed Nagareru instance generator

We propose a new efficient method to generate an interesting Nagareru instance  $P$  as follows:

1. Let  $W = B = \emptyset$  and  $P \equiv (w, h, W, B)$
2. Enumerate  $A_W \subseteq [wh] \times D$  that is a set of non-infeasible white cells, and  $A_B \subseteq [wh] \times D$  that is a set of non-ineffective black cells.
3. If  $A_W = A_B = \emptyset$ , restart this algorithm. Otherwise uniformly sample  $(i, d)$  from  $A_W$  (or  $A_B$ ) without replacement, and add  $(i, d)$  to  $W$  (or  $B$ .)
4. If  $P$  is good, go to 5. If  $P$  is not good but valid, repeat 2-3. If  $P$  is invalid, delete  $(i, d)$  from  $P$  and go to 3.
5. Delete all redundant cells on  $P$  and output  $P$ .

The Algorithm employs our ZDD-based solver as its building block. In Step 4, the Algorithm constructs the ZDD  $Z$  of  $\mathcal{F}_P$  and checks whether  $P$  is good, valid, or not by computing  $|\mathcal{F}_P|$  on  $Z$ . In Step 5, for each white and black cell on  $P$ , the Algorithm constructs the ZDD of  $\mathcal{F}_{P'}$  where  $P'$  is obtained by removing the cell from  $P$  and checks  $|\mathcal{F}_P| = |\mathcal{F}_{P'}|$  or not. If every cell on  $P$  is non-redundant, the Algorithm outputs  $P$  as an interesting instance.

## 6 Experiments

We conducted several experiments and confirmed the following two facts:

1. The ZDD-based Nagareru solver works more efficiently than the CSP-based solver,
2. The ZDD-based Nagareru generator creates interesting instances with realistic board sizes in a reasonable time.

We have uploaded our code and datasets used in the experiments to the following GitHub repository: <https://github.com/masakazu-ishihata/Nagareru>.

### 6.1 Experimental Setting

Our CSP-based Nagareru solver was implemented using Sugar [20], a state-of-the-art CSP solver. Our ZDD-based Nagareru solver was implemented in C++ using TdZdd<sup>1</sup> that is a C++ library for FBS. Our Nagareru instance generator was also implemented in C++ using the ZDD-based solver as a building block. The ZDD-based solver and generator were compiled by g++ 11.0.3 with the -O3 option. All experiments were conducted on a 64-bit mac OS Big Sur 11.2.3 with six Intel Core i7 3.2 GHz CPU and 64 GB RAM; however, we ran all programs on a single core. The timeout for solving each instance is 100 seconds throughout the experiment.

The whole dataset for evaluation consisted of 10 synthetic datasets and one handcrafted dataset. Each synthetic dataset consisted of 100 interesting instances generated by our generator with the different grid size  $(w, h) = (5, 5), (6, 6), \dots, (14, 14)$ . The handcrafted dataset consisted of 97 interesting instances on  $(10, 10)$  grid board obtained by crawling some puzzle creators' blogs and collecting instances in PUZ-PRE format, where PUZ-PRE<sup>2</sup> is a web application for editing and playing paper-and-pencil puzzles.

<sup>1</sup> <https://github.com/kunisura/TdZdd>

<sup>2</sup> <http://pzv.jp/>

■ **Table 1** The averages (Ave.), variance (Var.), and median (Med.) of the computation time (sec) and the numbers of solved instances (Sol.) of the CSP-based and the ZDD-based solver, respectively. Timeout instances were excluded when calculating the averages, variances, and medians.

Datataset	$(w, h)$	The CSP-based solver				The ZDD-based solver			
		Ave.	Var.	Med.	Sol.	Ave.	Var.	Med.	Sol.
Synthetic	(5, 5)	0.521	0.001	0.523	100	0.007	0.000	0.007	100
	(6, 6)	0.571	0.001	0.556	100	0.007	0.000	0.007	100
	(7, 7)	0.673	0.060	0.611	100	0.007	0.000	0.007	100
	(8, 8)	2.672	73.400	0.909	100	0.008	0.000	0.008	100
	(9, 9)	1.402	11.518	0.948	99	0.009	0.000	0.008	100
	(10, 10)	1.088	1.344	0.855	100	0.009	0.000	0.008	100
	(11, 11)	1.306	0.682	1.186	100	0.010	0.000	0.010	100
	(12, 12)	2.185	24.127	1.309	100	0.015	0.000	0.011	100
	(13, 13)	4.169	105.999	1.565	100	0.022	0.001	0.013	100
	(14, 14)	2.791	18.174	1.851	98	0.022	0.001	0.016	100
Handcrafted	(10, 10)	1.089	0.008	1.084	97	0.008	0.000	0.008	97

## 6.2 Experimental Results

Table 1 shows the computation times and numbers of solved instances of the CSP- and ZDD-based solver, respectively. It indicates that the ZDD-based solver is drastically faster than the CSP-based solver for each dataset; even though the former implicitly enumerates all the solutions, the latter finds only one solution. It also shows that the variance of the ZDD-based solver is significantly smaller than that of the CSP-based solver. Similar results have been reported for solving Slitherlink [24].

Table 2 indicates the statistics of each dataset; it shows that the average generation time increases exponentially with the grid size, whereas the average number of calls of the ZDD-based solver rises almost linearly. This result implies that the computation time of the ZDD-based solver increases exponentially with the grid size, which is consistent with its computational complexity shown in Proposition 9. It also shows that synthetic instances slightly tend to have more white cells, fewer black cells, and smaller solutions than handcrafted instances; however, it is unknown whether the proportion of white and black cells directly contributes to the fun of instances that humans feel. The interesting instances of (14, 14) grid with the smallest/largest  $|W|$ ,  $|B|$ , and  $|F|$  are shown in Appendix; the one with the smallest  $|F|$  seems too easy for humans to solve; however, the others seem complicated enough to enjoy solving. Note that our generator allows adjusting the ratio of white and black cells by changing the sampling distribution of its Step 3 and adjusting the size of the solution by adding a size constraint to Nagareru constraints.

## 7 Conclusion

We proposed an efficient solver and generator for Nagareru. Our solver constructs a ZDD representing all the solutions of a Nagareru instance by the FBS designed for this problem. Our generator employs our solver to guarantee that a generated instance is interesting; namely, it admits precisely one solution and has no redundant cell. We conducted some experiments and confirmed that our ZDD-based solver was drastically faster than a CSP-based one and our generator created interesting instances in a reasonable time.

■ **Table 2** The first and second columns indicate the database type and grid size, respectively. The third and fourth ones indicate the averages of the construction time (sec) and the number of calls of the ZDD-based solver of each instance generation, respectively. The fifth and sixth ones indicate the average number of white and black cells, respectively. The seventh one indicates the average of the solution size  $|F|$ .

Dataset	$(w, h)$	Ave. Time	Ave. # calls	Ave. $ W $	Ave. $ B $	Ave. $ F $
Synthetic	(5, 5)	0.020	26.280	2.050	2.850	11.500
	(6, 6)	0.040	39.930	2.920	3.430	16.360
	(7, 7)	0.085	59.880	3.900	4.810	22.680
	(8, 8)	0.212	84.460	5.200	6.470	30.580
	(9, 9)	0.677	110.640	6.800	8.530	42.940
	(10, 10)	2.638	148.380	8.800	10.720	53.980
	(11, 11)	11.660	197.880	10.780	12.710	65.620
	(12, 12)	58.152	240.010	12.590	15.780	76.680
	(13, 13)	317.790	281.710	14.630	18.510	88.840
Handcrafted	(10, 10)	-	-	6.814	16.278	66.020

---

## References

- 1 Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, and Jayson Lynch. Tatamibari is np-complete. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms, FUN 2021, May 30 to June 1, 2021, Favignana Island, Sicily, Italy*, volume 157 of *LIPICs*, pages 1:1–1:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.FUN.2021.1.
- 2 Daniel Andersson. HIROIMONO is np-complete. In Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci, editors, *Fun with Algorithms, 4th International Conference, FUN 2007, Castiglioncello, Italy, June 3-5, 2007, Proceedings*, volume 4475 of *Lecture Notes in Computer Science*, pages 30–39. Springer, 2007. doi:10.1007/978-3-540-72914-3\_5.
- 3 Erich Friedman. Corral puzzles are np-complete. *Technical Report*, 2002. URL: <https://erich-friedman.github.io/papers/corral.pdf>.
- 4 Markus Holzer, Andreas Klein, Martin Kutrib, and Oliver Ruepp. Computational complexity of NURIKABE. *Fundam. Informaticae*, 110(1-4):159–174, 2011. doi:10.3233/FI-2011-534.
- 5 Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. Np-completeness of two pencil puzzles: Yajilin and country road. *UTILITAS MATHEMATICA*, 88:237–246, July 2012.
- 6 Chuzo Iwamoto. Yosenabe is np-complete. *J. Inf. Process.*, 22(1):40–43, 2014. doi:10.2197/ipsjip.22.40.
- 7 Chuzo Iwamoto, Masato Haruishi, and Tatsuaki Ibusuki. Herugolf and makaro are np-complete. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, June 13-15, 2018, La Maddalena, Italy*, volume 100 of *LIPICs*, pages 24:1–24:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.FUN.2018.24.
- 8 Chuzo Iwamoto and Tatsuaki Ibusuki. Dosun-fuwari is np-complete. *J. Inf. Process.*, 26:358–361, 2018. doi:10.2197/ipsjip.26.358.
- 9 Chuzo Iwamoto and Tatsuya Ide. Moon-or-sun, nagareru, and nurimeizu are np-complete (in japanese). In *Winter LA Symposium 2019*, 2019.
- 10 Chuzo Iwamoto and Tatsuya Ide. Nurimisaki and sashigane are np-complete. In Zachary Friggstad and Jean-Lou De Carufel, editors, *Proceedings of the 31st Canadian Conference on Computational Geometry, CCCG 2019, August 8-10, 2019, University of Alberta, Edmonton, Alberta, Canada*, pages 184–194, 2019.

- 11 Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 100-A(9):1773–1784, 2017. doi:10.1587/transfun.E100.A.1773.
- 12 Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Colorful frontier-based search: Implicit enumeration of chordal and interval subgraphs. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis of Experimental Algorithms - Special Event, SEA<sup>2</sup> 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2019. doi:10.1007/978-3-030-34029-2\_9.
- 13 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of np-complete puzzles. *J. Int. Comput. Games Assoc.*, 31(1):13–34, 2008.
- 14 Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, pages 272–277. ACM Press, 1993. doi:10.1145/157485.164890.
- 15 Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shin-ichi Minato. Implicit enumeration of topological-minor-embeddings and its application to planar subgraph enumeration. In M. Sohel Rahman, Kunihiko Sadakane, and Wing-Kin Sung, editors, *WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings*, volume 12049 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2020. doi:10.1007/978-3-030-39881-1\_18.
- 16 Nikoli Co., Ltd. Puzzles: Nagareru [Nikoli]. Available online, 2021. <https://www.nikoli.co.jp/en/puzzles/nagareru.html>, (accessed on 20th April 2021).
- 17 Hirofumi Suzuki, Sun Hao, and Shin-ichi Minato. Generating all solutions of minesweeper problem using degree constrained subgraph model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 356. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2016.
- 18 Naoyuki Tamura. Solving Puzzles with Sugar Constraint Solver (in Japanese). Available online, 2013. <https://cspSAT.gitlab.io/sugar-puzzles/>, (accessed on 20th April 2021).
- 19 Naoyuki Tamura. Solving Slither Link Puzzles with Sugar Constraint Solver (in Japanese). Available online, 2013. <https://cspSAT.gitlab.io/sugar-puzzles/slitherlink.html>, (accessed on 20th April 2021).
- 20 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. doi:10.1007/s10601-008-9061-0.
- 21 Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. Technical report, Technical Report, TR96-0008, 1996.
- 22 Gerhard van der Knijff, H Zantema, and JH Geuvers. Solving and generating puzzles with a connectivity constraint. *Bachelor thesis of Radboud University*, 2021.
- 23 Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 86-A(5):1052–1060, 2003. URL: [http://search.ieice.org/bin/summary.php?id=e86-a\\_5\\_1052](http://search.ieice.org/bin/summary.php?id=e86-a_5_1052).
- 24 Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5(2):176–213, 2012. doi:10.3390/a5020176.

## **A** Solving blank instances

A Nagareru instance with a small number of white and black cells has a small number of constraints; that is, it has many solutions. For instance, the number of solutions of the  $n \times n$  blank instance, which is a grid with no colored cell, is the same as the number of cycles on

the grid that is exponential in  $n$ . Whereas it is trivial for humans to find a cycle in the blank instance, it is not trivial for the CSP-based and ZDD-based solvers because, in most cases of CSP, constraints help to reduce the search space. The CSP-based solver has to find a cycle with no additional directional constraint, and the ZDD-based solver has to enumerate all cycles in the grid. Table 3 indicates the computation time of solving the  $n \times n$  blank instance ( $n = 5, 6, \dots, 14$ ) of the CSP-based and the ZDD-based solvers. Compared to Table 1 in our main manuscript, the computation time of the blank instance is relatively more significant than that of an interesting instance. Our generator constructs the ZDD of the blank instance in the first step, which accounts for a large part of the total generation time. Thus, we can quickly scale up our generator by initializing the grid with many colored cells.

■ **Table 3** The computation time of solving the  $n \times n$  blank instance of the CSP-based and the ZDD-based solver. T.O. indicates timeout, meaning that it takes more than 100 seconds.

$n$	CSP	ZDD
5	0.586906	0.005276
6	0.722846	0.006490
7	0.867803	0.009694
8	1.106015	0.020341
9	1.102662	0.056783
10	1.430713	0.172517
11	4.939170	0.544814
12	4.022487	1.838747
13	T.O.	6.246969
14	T.O.	22.105007

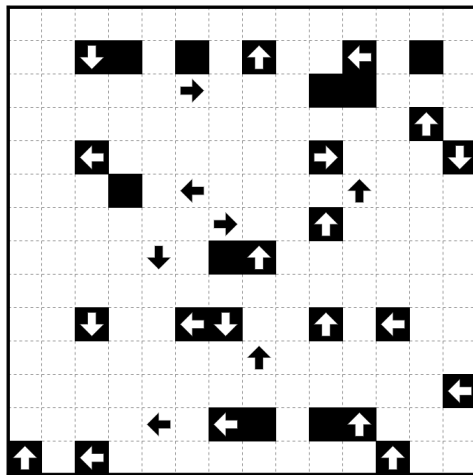
## B Various interesting instances generated by our Nagareru generator

Figure 3 shows a part of interesting instances generated by our generator. Figure 3(a) and (b) have the smallest and the largest number of  $|W|$ , respectively. Figure 3(c) and (d) have the smallest and the largest number of  $|B|$ , respectively. Figure 3(e) and (f) have the largest number of  $|F|$  and  $|B \cup W|$ , respectively. Figure 3(c) has also the smallest  $|F|$  and  $|B \cup W|$ .

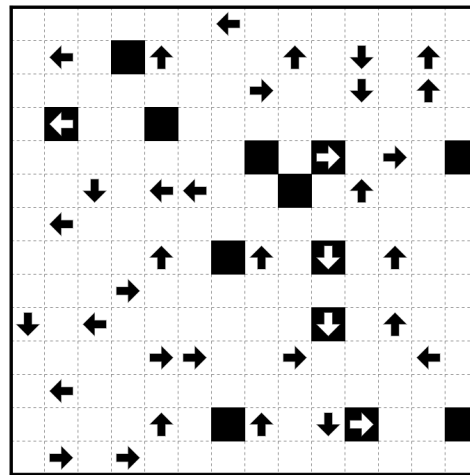
Figure 3(c) looks easy to solve for humans; however, the CSP-based solver could not solve it in 100 seconds. In this instance, the variables corresponding to the bottom four blank rows are not constrained. We consider that the CSP-based solver wasted much time determining the values of such non-constraint variables.

In contrast to Figure 3(c), the other instances seem complex enough for humans to enjoy solving. The first step of solving a Nagareru instance is extending each arrow in each direction of its head and tail by one cell length and creating some line fragments. For example, Figure 3(b) has the largest number of white cells; therefore, the first step can create many long line fragments. On the other hand, Figure 3(a) and (d) have a few white cells, and the first step creates a few short line fragments. However, an instance with many line fragments is not always easy to solve because it is not obvious how to connect them to construct a single consistent cycle. In fact, in Figure 3(b), one has to connect those line fragments carefully to form a single consistent cycle.

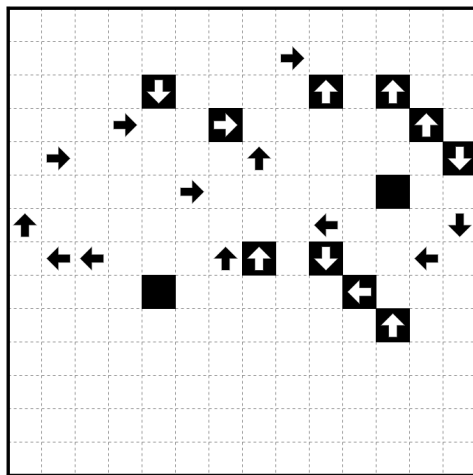




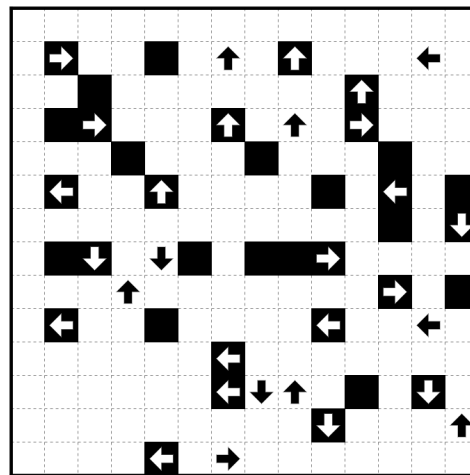
(a) Smallest  $|W|$ .



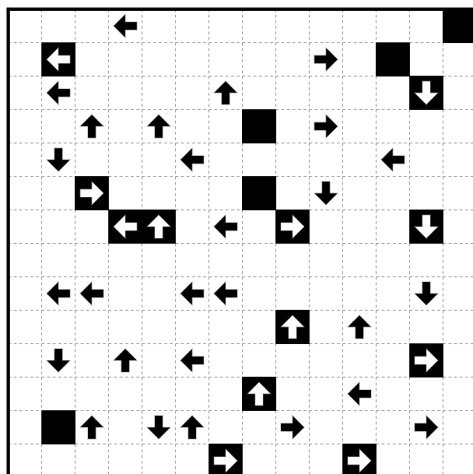
(b) Largest  $|W|$ .



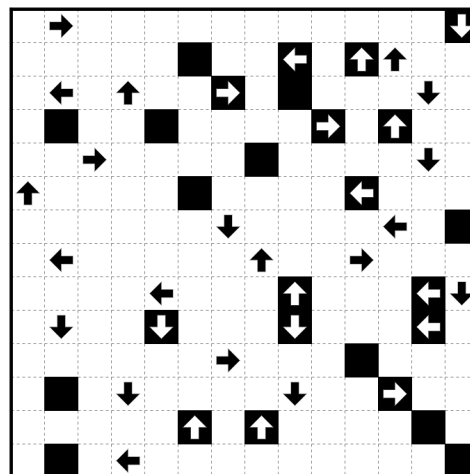
(c) Smallest  $|B|$ .



(d) Largest  $|B|$ .



(e) Largest  $|F|$ .



(f) Largest  $|B \cup W|$ .

■ **Figure 3** Some examples of 14x14 Nagareru instances generated by our method.