An Efficient Approach to Solving Nonograms

I.-Chen Wu, *Member, IEEE*, Der-Johng Sun, Lung-Ping Chen, Kan-Yueh Chen, Ching-Hua Kuo, Hao-Hua Kang, and Hung-Hsuan Lin

***Abstract*—A nonogram puzzle is played on a rectangular grid of pixels with clues given in the form of row and column constraints. The aim of solving a nonogram puzzle, an NP-complete problem, is to paint all the pixels of the grid in black and white while satisfying these constraints. This paper proposes an efficient approach to solving nonogram puzzles. We propose a fast dynamic programming (DP) method for line solving, whose time complexity in the worst case is $O(kl)$ only, where the grid size is $l \times l$ and k is the average number of integers in one constraint, always smaller than l. In contrast, the time complexity for the best line-solving method in the past is $O(kl^2)$. We also propose some fully probing (FP) methods to solve more pixels before running backtracking. Our FP methods can solve more pixels than the method proposed by Batenburg and Kosters (before backtracking), while having a time complexity that is smaller than theirs by a factor of $O(l)$. Most importantly, these FP methods provide useful guidance in choosing the next promising pixel to guess during backtracking. The proposed methods are incorporated into a fast nonogram solver, named LalaFrogKK. The program outperformed all the programs collected in webpbn.com, and also won both nonogram tournaments that were held at the 2011 Conference on Technologies and Applications of Artificial Intelligence (TAAI 2011, Taiwan). We expect that the proposed FP methods can also be applied to solving other puzzles efficiently.**

***Index Terms*—Backtracking, fully probing (FP), nonogram, NP-completeness, painted by number, puzzles.**

I. Introduction

NONOGRAM [19], also known as *Hanjie*, *Paint by Numbers*, or *Griddlers*, is one of the popular logic puzzles invented by a Japanese graphics editor named Non Ishida in 1988. In 1990, the U.K. newspaper *The Sunday Telegraph* started publishing nonogram puzzles on a weekly basis [14].

A nonogram puzzle is played on a given rectangular grid of cells, alternatively referred to as pixels, with clues given in the form of row and column constraints. The aim of solving a nonogram puzzle is to paint all the pixels of the grid in black and white while satisfying these given constraints. A nonogram puzzle is illustrated in Fig. 1(a), where in an 8×8 grid row constraints are given in the rightmost column and column constraints are given in the bottom row. In this paper, symbol "•" stands for a pixel that is to be painted, and "■" and "□" stand for a pixel painted in black and white, respectively. The integers in these constraints are used to indicate the lengths of black segments in the solution shown in Fig. 1(b). A black segment is a set of maximally consecutive black pixels in a row (or a column). For example, the constraints of the third row, 1 3, indicate that the row in the solution has two, and only two, black segments where the length of the first segment (leftmost) is 1 and that of the second segment is 3. It is not required to have unique solutions for nonogram puzzles, but most puzzles published for human players normally have unique solutions.

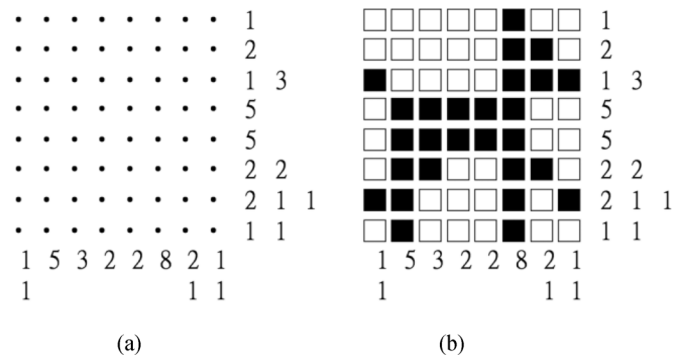


Fig. 1. (a) A nonogram puzzle given in [21] and (b) its solution.

Solving a nonogram puzzle efficiently is challenging, especially those that are large. Ueda and Nagao showed [17] that the general problem of determining whether a nonogram puzzle has a solution is NP-complete.

In the past, several researchers studied how to solve nonogram puzzles by translating them into other problems. Bosch [5] translated a nonogram problem into an integer linear programming (IPL) problem and solved it accordingly. Faase [8] translated it into an exact cover problem and we can then use Knuth's dancing-links method [9] to solve it. Unfortunately, the sizes of the translated problems are usually too large to solve efficiently. Batenburg [1] modified an evolutionary algorithm for discrete tomography (DT) to solve nonogram puzzles, while Wiggers [18] used a genetic algorithm. However, both methods do not guarantee exact solutions.

Many researchers have been engaged in the study of solving nonogram puzzles efficiently. In [24], Yu *et al.* proposed an algorithm to solve nonogram puzzles based on specific logical rules, using backtracking to solve undetermined cells and logical rules to improve the search efficiently. These logical rules are mainly used to help line painting (either for rows or columns), with the goal of painting as many pixels in each line as possible. Some simpler line painting methods were also mentioned in [13] and [15]. However, these methods are not guar-

Manuscript received September 28, 2012; revised January 09, 2013 and February 19, 2013; accepted February 27, 2013. Date of publication March 08, 2013; date of current version September 11, 2013. This work was supported by the National Science Council of the Republic of China (Taiwan) under Contracts NSC 99-2221-E-009-102-MY3 and NSC 99-2221-E-009-104-MY3.

I.-C. Wu, D.-J. Sun, K.-Y. Chen, C.-H. Kuo, H.-H. Kang, and H.-H. Lin are with the Department of Computer Science, National Chiao-Tung University, Hsinchu 30050, Taiwan (e-mail icwu@csie.nctu.edu.tw).

L.-P. Chen is with the Department of Computer Science and Information Engineering, Tung-Hai University, Taichung 40704, Taiwan.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2013.2251884

anteed to solve lines. In this paper, line solving means painting the maximum number of pixels for given lines.

For line solving, Batenburg and Kosters [2] proposed a dynamic programming (DP) method instead of specific logical rules. In general, this method paints more pixels than those in [13], [15], and [24], and runs much faster. Assume that the grid size of a given puzzle is $l \times l$ and the average number of integers in one constraint is k, which is smaller than l. In the worst case, the time complexity for solving each line using the DP method is estimated to be $O(kl^2)$. Furthermore, Batenburg and Kosters [2] also used a 2-Satisfiability (2-SAT) method to help paint more pixels in whole grids (before backtracking), and the time complexity for the method was $O(l^7)$, estimated in Section III-D. In addition to the above research, many nonogram solvers have been designed, collected, and published in [21], and nonogram-generating algorithms have been discussed in [2] and [3].

In this paper, based on the research in [2], we propose a faster DP method for line solving, whose time complexity in the worst case is $O(kl)$ only, which improves that in [2] by $O(l)$. In addition, we propose three fully probing (FP) methods, named FP1, FP2, and FP3, to paint more pixels before backtracking. The time complexities for both FP1 and FP2 (before backtracking) are $O(kl^5)$, while that for FP3 is $O(l^6)$. In the case that the FP methods alone cannot paint the entire grid, backtracking will be used to finish the process. Most importantly, the FP methods also provide backtracking with useful information as guidance in choosing the next pixel to guess.

The proposed methods were also incorporated into a nonogram solver, named LALAFROGKK, which won both nonogram tournaments that were held at the 2011 Conference on Technologies and Applications of Artificial Intelligence (TAAI 2011, Taiwan) [10]. The program also outperformed all the programs collected in [21].

In the rest of this paper, Section II describes our methods for line solving and propagation. Section III proposes three FP methods to solve more pixels before backtracking. Section IV describes backtracking. Experiments are presented in Section V, and concluding remarks are given in Section VI.

II. LINE SOLVING AND PROPAGATION

In this section, first, we specify the definitions and notations in Section II-A. Then, we describe our methods for line solving in Section II-B. Propagation of line solving in the whole grid is given in Section II-C.

A. Definitions and Notation

As described in Section I, a nonogram puzzle is given initially with a $w \times h$ rectangular grid, a sequence of row descriptions $\langle D_1^R, D_2^R, \ldots, D_h^R \rangle$ for row constraints, and a sequence of column descriptions $\langle D_1^C, D_2^C, \ldots, D_w^C \rangle$ for column constraints. Each D_i^R is again a sequence of integers $\langle d_{i1}^R, d_{i2}^R, \ldots, d_{ir_i}^R \rangle$, where r_i is the number of integers in the sequence. Similarly, each D_i^C is a sequence of integers $\langle d_{i1}^C, d_{i2}^C, \ldots, d_{ic_i}^C \rangle$, where c_i is the number of integers in the sequence. The goal of solving the puzzle is to paint all the pixels of the grid in black or white such that the following conditions hold.

1) Each row i (column i) in the grid includes r_i (c_i) black segments (defined in Section I).
2) Each integer d_{ij}^R (d_{ij}^C) indicates the length of the jth black segment from the left in row i (from the top in column i).

For simplicity of discussion, consider a line, instead of a row or a column. Let a line be represented by a string $S = s_1 s_2 \ldots s_l$, where l denotes the length of the line (or the number of pixels in the line), and each s_i indicates the color of the ith pixel in the line. The substring $s_i s_{i+1} \ldots s_j$ is a segment of the line, where $1 \le i \le j \le l$. Let s_i be a symbol 0 (or 1), if the ith pixel in the line is painted in white (or black). Segment $s_i s_{i+1} \ldots s_j$, where $1 \le i \le j \le l$, is a black segment, if all $s_i, \ldots, s_j$ are 1's, and both s_{i-1} and s_{j+1}, if they exist, are 0's. Following the notation in [2] for the string representation, let Σ be the finite alphabet $\{0, 1\}$, and Σ^l be the set of all the strings over Σ of length l. For simplicity, let Σ^* be the set of all the strings over Σ of all lengths. For simplicity of discussion, a pattern for strings is expressed in a regular expression notation. For example, both 0100111 and 001100 match pattern $0^+1^+0^21^*$.

As defined above, each line (either row or column) is given by a description, say $D = \langle d_1, d_2, \ldots, d_r \rangle$. Line $S = s_1 s_2 \ldots s_l \in \Sigma^l$ is said to be consistent with D, if r is the number of black segments in the line, and each d_i is the length of the ith black segment (starting from s_1). More specifically, given description D, line S is consistent with D if S matches the following pattern:

$$0^*1^{d_1}0^+1^{d_2}0^+ \ldots 0^+1^{d_r}0^*.$$

Let $\sigma(d)$ denote segment $0^1 1^d$ of length $d+1$. Let $0S$ denote the concatenation of 0 and S. The above can be rewritten as follows for line consistency. Given D, S is consistent with D if $0S$ matches the following pattern:

$$0^*\sigma(d_1)0^*\sigma(d_2)0^* \ldots 0^*\sigma(d_r)0^*.$$

Among all the strings that match the pattern, the minimal string length is $\sum_{p=1}^{r}(d_p + 1) - 1$.

When a line is partially painted, some pixels in the line may not be painted yet, since the colors of these pixels are still unknown. In line $S = s_1 s_2 \ldots s_l$, let s_i be the symbol u if the ith pixel has not yet been painted. Following the notation in [2] for the string representation, let $\Gamma = \Sigma \cup \{u\}$ be the finite alphabet, and Γ^l be the set of all the strings over Γ of length l. For line $S \in \Gamma^*$, if some $s_i = u$, the ith pixel s_i will need to be changed to either 0 or 1, but not *vice versa*. For line $S \in \Sigma^*$, no further changes are required.

String $S' = s'_1 s'_2 \ldots s'_l \in \Gamma^l$ is said to be an *assignment* of another string $S = s_1 s_2 \ldots s_l \in \Gamma^l$, if the following three conditions hold: 1) if s_i is 0, then s'_i is 0; 2) if s_i is 1, then s'_i is 1; and 3) if s_i is u, then s'_i is one of 0, 1, or u. Let $S \longrightarrow S'$ denote the assignment relation between S and S'. For example, 0100, 0110, and $11u0$ are assignments of $u1u0$, that is, $u1u0 \longrightarrow 0100$, $u1u0 \longrightarrow 0110$, and $u1u0 \longrightarrow 11u0$. Following [2], an assignment S' of S is also called a fix of S, if $S' \in \Sigma^*$. For simplicity, let $S \longrightarrow^{\Sigma} S'$ denote $S \longrightarrow S'$ and $S' \in \Sigma^*$.

String $S \in \Gamma^*$ is said to be fixable with respect to description D, if there exists some fix S' with $S \longrightarrow^{\Sigma} S'$ such that S' is

consistent with D. For example, $u1u0$ is fixable with respect to $D = \langle 3 \rangle$ since fix 1110 (for $u1u0 \longrightarrow 1110$) is consistent with D. In addition, string $S \in \Gamma^*$ is said to match pattern p, if there exists fix S' of S such that S' matches p. For example, $u1u0$ matches $1^3 0^*$.

String $S \in \Gamma^*$ is said to be partially fixable to string $S' \in \Gamma^*$ with respect to description D, if S' is an assignment of S and S' is fixable with respect to D. For simplicity, the relation is denoted by $S \xrightarrow{D} S'$. For example, we have $u1uu \xrightarrow{\langle 3 \rangle} 1110$, $u1uu \xrightarrow{\langle 3 \rangle} u110$, and $u1uu \xrightarrow{\langle 3 \rangle} u11u$.

B. Line Solving

This section studies methods to paint a partially painted line with respect to a description. We investigate two problems, fixability and line solving, as described in Sections II-B1 and II-B2, and then give a discussion in Section II-B3.

1) Fixability: The fixability problem is as follows: Given string $S = s_1 s_2 \dots s_l \in \Gamma^l$ and description $D = \langle d_1, d_2, \dots, d_k \rangle$, verify whether S is fixable with respect to D. As mentioned in Section II-A, S is consistent with D if $0S$ matches the following pattern: $0^*\sigma(d_1)0^*\sigma(d_2)0^* \dots 0^*\sigma(d_r)0^*$. For simplicity, let us assume that s_1 is already 0 in the rest of this section. If s_1 is not 0, we simply add one 0 at the beginning of S.

For this fixability problem, we propose a DP method, which is more efficient than the one in [2]. Our method is to build a recurrence $\mathrm{Fix}(i, j)$ for all $i \geq 0$ and $j \geq 0$ as follows. Let $S^{(i)}$ denote string $s_1 s_2 \dots s_i$, and $D^{(j)}$ denote description $\langle d_1, d_2, \dots, d_j \rangle$. Given $S^{(i)}$ and $D^{(j)}$, $\mathrm{Fix}(i, j)$ verifies whether $S^{(i)}$ is fixable with respect to $D^{(j)}$, as shown in Proposition 1. In addition, let $\mathrm{Fix0}(i, j)$ indicate whether string $S^{(i)}$ is fixable with respect to $D^{(j)}$ in the case that s_i matches 0, and, similarly, let $\mathrm{Fix1}(i, j)$ indicate fixability for the case where s_i matches 1. An example is illustrated in Fig. 2. The recurrences for $\mathrm{Fix}(i, j)$ are formulated in detail as follows:

$$\mathrm{Fix}(i,j) = \begin{cases} \text{true}, & \text{if } i = 0 \text{ and } j = 0 \\ \text{false}, & \text{if } i = 0 \text{ and } j \geq 1 \\ \mathrm{Fix0}(i,j) \bigvee \mathrm{Fix1}(i,j), & \text{otherwise} \end{cases}$$

$$\mathrm{Fix0}(i,j) = \begin{cases} \mathrm{Fix}(i-1,j), & \text{if } s_i \in \{0, u\} \\ \text{false}, & \text{otherwise} \end{cases}$$

$$\mathrm{Fix1}(i,j) = \begin{cases} \mathrm{Fix}(i-d_j-1, j-1), & \text{if } j \geq 1, i \geq d_j + 1, \text{ and} \\ & s_{i-d_j} \dots s_i \text{ matches } \sigma(d_j) \\ \text{false}, & \text{otherwise.} \end{cases}$$

Proposition 1: For the above recurrences, $\mathrm{Fix}(i, j)$ is true if $S^{(i)}$ is fixable with respect to $D^{(j)}$, and false, otherwise. □

The solution to the fixability problem, therefore, is the value of $\mathrm{Fix}(l, k)$. To derive this value, we use the typical DP method [7] by calculating a table with size $l \times k$, where the table is a 2-D array of data returned from $\mathrm{Fix}(i, j)$ for all i, j. Since it takes only constant time to calculate each $\mathrm{Fix}(i, j)$, the time complexity for evaluating $\mathrm{Fix}(l, k)$ is clearly $O(kl)$. In contrast, the time complexity given in [2] is $O(kl^2)$. In practice, the above method can be further improved by checking whether

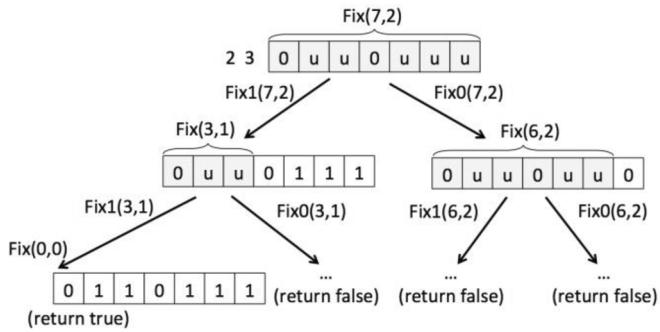


Fig. 2. Fix0(7, 2) for a string $S = 0uu0uuu$ with respect to $D = \{2, 3\}$.

$i \geq \sum_{p=1}^{j}(d_p + 1)$ in all $\mathrm{Fix}(i, j)$. If not, return false immediately. This is because the minimal length of strings fixable with respect to $D^{(j)}$ is $\sum_{p=1}^{j}(d_p + 1)$, as mentioned in Section II-A.

2) Line Solving: This section describes our line-solving method. Before discussing it, we define the equivalence of two strings with respect to a description in Definition 1.

Definition 1: Let string $S \in \Gamma^l$. Let $\Psi(S, D)$ be the set of all fixes $S'' \in \Sigma^l$, where $S \xrightarrow{D} S''$. For another string $S' \in \Gamma^l$, S and S' are called to be equivalent with respect to D, denoted by $S \overset{\mathrm{D}}{\equiv} S'$, if $\Psi(S, D) = \Psi(S', D)$. □

Furthermore, a painting and the maximal painting of a string are defined in Definition 2.

Definition 2: String S' is called a painting of S with respect to D, denoted by $S \overset{D}{\Longrightarrow} S'$, if $S \overset{\mathrm{D}}{\equiv} S'$ and $S \longrightarrow S'$. Painting S' is called the maximal painting of S with respect to D, denoted by $S \overset{D,\max}{\Longrightarrow} S'$, if the following condition holds: for all S'', where $S'' \overset{\mathrm{D}}{\equiv} S$, we have $S'' \overset{D}{\Longrightarrow} S'$. □

The above definitions are illustrated by an example with string $S = u0u1uu$ and description $D = \langle 3 \rangle$. Clearly, the fix set $\Psi(S, D)$ is $\{001110, 000111\}$. All the strings, $u0u11u, 00u1uu$, and $00u11u$, are paintings from S with respect to D, since their fix sets with respect to D are the same as $\Psi(S, D)$. Among these paintings, $00u11u$ is the maximal painting of S, that is, $S \overset{D,\max}{\Longrightarrow} 00u11u$.

The line-solving problem is as follows: Given string S and description D, find the maximal painting S' of S, that is, $S \overset{D,\max}{\Longrightarrow} S'$. The maximal painting can be derived from $\mathcal{S}(\Psi(S, D))$, the intersection of all possible fixes, as follows. Given a set of fixes $\Psi \subseteq \Sigma^l$, $\mathcal{S}(\Psi)$ is defined to be string $S = s_1 s_2 \dots s_l \in \Gamma^l$ satisfying the following properties for each s_i:

1) $s_i = 0$, if the ith pixel is 0 for all strings $S' \in \Psi$;
2) $s_i = 1$, if the ith pixel is 1 for all strings $S' \in \Psi$;
3) $s_i = u$, if the ith pixel is 0 for some string $S' \in \Psi$ and 1 for some other string $S'' \in \Psi$.

Proposition 2: From the above definitions, given string $S \in \Gamma^l$ and D, the following properties are satisfied:

1) $\mathcal{S}(\Psi(S, D))$ is the maximal painting of S with respect to D;
2) if $S \longrightarrow S'$, then $\Psi(S, D) \supseteq \Psi(S', D)$;
3) if $\Psi \supseteq \Psi'$, then $\mathcal{S}(\Psi) \longrightarrow \mathcal{S}(\Psi')$;
4) if $S \longrightarrow S'$, then $\mathcal{S}(\Psi(S, D)) \longrightarrow \mathcal{S}(\Psi(S', D))$ (from the previous two). □

Proposition 2 shows some properties. The first property is that $\mathcal{S}(\Psi(S,D))$ is the maximal painting of S with respect to D. From the previous example, $S = u0u1uu$ and $D = \langle 3\rangle$. We have $\Psi(S,D) = \{001110, 000111\}$, and $\mathcal{S}(\Psi(S,D)) = 00u11u$, which is the maximal painting of S. The next three properties are illustrated by an example with an additional $S' = u011uu$, where $S \longrightarrow S'$. The fix sets $\Psi(S,D)$ and $\Psi(S',D)$ are $\{001110, 000111\}$ and $\{001110\}$, respectively, satisfying $\Psi(S,D) \supseteq \Psi(S',D)$. Both maximal paintings are, respectively, $00u11u$ and 001110, satisfying $\mathcal{S}(\Psi(S,D)) \longrightarrow \mathcal{S}(\Psi(S',D))$.

For the line-solving problem, we also use a DP method, modified from the one used for the computation of $\mathrm{Fix}(i,j)$ to find the maximal painting of a given $S = s_1 s_2 \ldots s_l \in \Gamma^*$ with respect to a given $D = \langle d_1, d_2, \ldots, d_k\rangle$. Similarly, let $S^{(i)}$ denote string $s_1 s_2 \ldots s_i$, and $D^{(j)}$ denote description $\langle d_1, d_2, \ldots, d_j\rangle$. Given $S^{(i)}$ and $D^{(j)}$, where $\mathrm{Fix}(i,j)$ is true (that is, $S^{(i)}$ is fixable with respect to $D^{(j)}$), the recurrence $\mathrm{Paint}(i,j)$ for all $i \geq 0$ and $j \geq 0$ finds the maximal painting of $S^{(i)}$ with respect to $D^{(j)}$, as shown in Lemma 1. The recurrences are described in the equation at the bottom of the page.

Lemma 1: For the above recurrences, if $\mathrm{Fix}(i,j)$ is true, $\mathrm{Paint}(i,j)$ returns $\mathcal{S}(\Psi(S^{(i)}, D^{(j)}))$.

Proof: In the case that $i = 0$, it is trivial that $\mathrm{Paint}(i,j)$ simply returns null string ϕ. Note that if $i = 0$ and $j \geq 1$, then $\mathrm{Fix}(i,j)$ is false, and, thus, $\mathrm{Paint}(i,j)$ is not called.

In the case that $i \geq 1, \mathrm{Paint}(i,j)$ returns $\mathrm{Paint}'(i,j)$, which then derives $\mathcal{S}(\Psi(S^{(i)}D^{(j)}))$ from $\mathrm{Paint0}(i,j)$ and $\mathrm{Paint1}(i,j)$, respectively, for cases $s_i = 0$ and $s_i = 1$. Recurrence $\mathrm{Paint0}(i,j)$ returns $\mathcal{S}(\Psi(S^{(i-1)}D^{(j)}))$ if $\mathrm{Fix0}(i,j)$ is true, and $\mathrm{Paint1}(i,j)$ returns $\mathcal{S}(\Psi(S^{(i-1)}1, D^{(j)})) = \mathcal{S}(\Psi(S^{(i-d_j-1)}\sigma(d_j), D^{(j)}))$, if $\mathrm{Fix1}(i,j)$ is true.

Since $\mathrm{Fix}(i,j)$ is true from the assumption of this lemma and $\mathrm{Fix}(i,j) = \mathrm{Fix0}(i,j) \bigvee \mathrm{Fix1}(i,j)$ from Section II-A, one of the three conditions holds: 1) $\mathrm{Fix0}(i,j)$ is true and $\mathrm{Fix1}(i,j)$ is false; 2) $\mathrm{Fix0}(i,j)$ is false and $\mathrm{Fix1}(i,j)$ is true; and 3) both $\mathrm{Fix0}(i,j)$ and $\mathrm{Fix1}(i,j)$ are true.

Assume that the first condition holds. Then, all fixes in $\Psi(S^{(i)}, D^{(j)})$ are also in $\Psi(S^{(i-1)}0, D^{(j)})$, that is, $\Psi(S^{(i)}, D^{(j)}) = \Psi(S^{(i-1)}0, D^{(j)})$. Hence, $\mathrm{Paint}'(i,j)$ simply returns $\mathrm{Paint0}(i,j)$, which returns $\mathcal{S}(\Psi(S^{(i-1)}0, D^{(j)}))$. Similarly, assuming that the second condition holds, $\mathrm{Paint}'(i,j)$ simply returns $\mathrm{Paint1}(i,j)$.

Assume that the third condition holds. Let Ψ_0 and Ψ_1 denote $\Psi(S^{(i-1)}0, D^{(j)})$ and $\Psi(S^{(i-1)}1, D^{(j)})$, respectively. Obviously, $\Psi(S^{(i)}, D^{(j)}) = \Psi_0 \cup \Psi_1$. Hence, $\mathrm{Paint}'(i,j)$ derives $\mathcal{S}(\Psi(S^{(i)}, D^{(j)}))$ by merging $\mathcal{S}(\Psi_0)$ and $\mathcal{S}(\Psi_1)$, which are returned by $\mathrm{Paint0}(i,j)$ and $\mathrm{Paint1}(i,j)$, respectively. Let $\mathcal{S}_0$ and $\mathcal{S}_1$ denote $\mathcal{S}(\Psi_0)$ and $\mathcal{S}(\Psi_1)$, respectively. $\mathcal{S}_0$ and $\mathcal{S}_1$ are merged using the above function Merge. Let $\mathcal{S}_0$ and $\mathcal{S}_1$ be $s_1 s_2 \ldots s_i$ and $t_1 t_2 \ldots t_i$, respectively. Function Merge merges each pair of pixels pairwise by $\mathrm{Merge}C$. First, the kth pixel of the returning string is 0 if both s_k and t_k are 0, for the following reason. Since $s_k = 0$, the kth pixels for all strings in Ψ_0 must be 0. Similarly, since $t_k = 0$, the kth pixels for all strings in Ψ_1 must be 0. Thus, kth pixels in all strings $\Psi_0 \cup \Psi_1$ must be 0 as well. Therefore, the kth pixel of the returning string is 0. For the same reason, in function Merge, the kth pixel of the returning string is 1 if both s_k and t_k are 1.

For all the other cases for both s_k and t_k, function $\mathrm{Merge}C$ returns u for the kth pixel for the following reason. The rest of the cases can be classified into the following two subcases: 1) one of s_k and t_k is 0 and the other is 1; and 2) at least one of s_k and t_k is u. In the first subcase, since one is for 0 and the other for 1, the pixel must be u for property 3 of maximal painting. In the second subcase, since one is already u, indicating that some is for 0 and some other for 1, the pixel must be u as well. □

Let us illustrate this by using the above example: Given $S = u0u1uu$ and $D = \langle 3\rangle$, use $\mathrm{Paint}(6,1)$ to derive $\mathcal{S}(\Psi(S^{(6)}, D^{(1)}))$, that is, $\mathcal{S}(\Psi(S,D))$. In this example, both $\mathrm{Fix0}(6,1)$ and $\mathrm{Fix1}(6,1)$ are true, and both $\mathrm{Paint0}(i,j)$ and $\mathrm{Paint1}(i,j)$ return $\mathcal{S}(\Psi(S^{(5)}0, D^{(1)})) = \{001110\}$ and $\mathcal{S}(\Psi(S^{(5)}1, D^{(1)})) = \{000111\}$, respectively. Thus, $\mathrm{Paint}'(6,1)$ merges both, and returns $\mathcal{S}(\Psi(S,D)) = 00u11u$. Thus, $\mathrm{Paint}(6,1)$ returns $00u11u$.

For deriving the maximal painting $\mathrm{Paint}(l,k)$, we also use DP to calculate a table with size $l \times k$, where the table is a 2-D array of data returned from $\mathrm{Paint}(i,j)$ for all i,j. Since it takes $O(i)$ or $O(l)$ time to merge two strings, the time complexity for evaluating $\mathrm{Paint}(l,k)$ is $O(kl^2)$. In fact, the time complexity can be further reduced to $O(kl)$ theoretically, as explained in the Appendix. For this reason, the time complexity $O(kl)$ will be used in the analysis in the rest of this paper.

3) Discussion: In this section, two issues are discussed. The first issue to consider is the case of general multicolor nono-

$$\mathrm{Paint}(i,j) = \begin{cases} \phi, & \text{if } i = 0 \\ \mathrm{Paint}'(i,j), & \text{otherwise} \end{cases}$$

$$\mathrm{Paint}'(i,j) = \begin{cases} \mathrm{Paint0}(i,j), & \text{if } \mathrm{Fix0}(i,j) \text{ and not } \mathrm{Fix1}(i,j) \\ \mathrm{Paint1}(i,j), & \text{if not } \mathrm{Fix0}(i,j) \text{ and } \mathrm{Fix1}(i,j) \\ \mathrm{Merge}(\mathrm{Paint0}(i,j), \mathrm{Paint1}(i,j)), & \text{otherwise} \end{cases}$$

$$\mathrm{Paint0}(i,j) = \mathrm{Paint}(i-1,j)0$$

$$\mathrm{Paint1}(i,j) = \mathrm{Paint}(i-d_j-1, j-1)\sigma(d_j)$$

$$\mathrm{Merge}(s_1 s_2 \ldots s_i, t_1 t_2 \ldots t_i) = m_1 m_2 \ldots m_i, \qquad \text{where } m_k = \mathrm{Merge}C(s_k, t_k)$$

$$\mathrm{Merge}C(s_k, t_k) = \begin{cases} 0, & \text{if } s_k = t_k = 0 \\ 1, & \text{if } s_k = t_k = 1 \\ u, & \text{otherwise.} \end{cases}$$

grams, where Σ has more than two elements. In these kinds of puzzles, the descriptions should also include the colors of each segment. A general method for $\mathrm{Fix}(i,j)$ is to use $\mathrm{Fix}c(c,i,j)$, instead of $\mathrm{Fix1}(i,j)$, where c is the color of the last segment. $\mathrm{Fix}c(c,i,j)$ indicates whether string $S^{(i)}$ is fixable with respect to $D^{(j)}$ in the case where the last character s_i matches c in the description. In this way, the time complexity is still the same.

Another issue is of practical design. Since the line lengths (widths or heights) of most nonogram puzzles are normally below or around 30, we can use bit vectors to implement lines. For example, we use two bits to indicate a pixel, bits 01 for 0, 10 for 1, and 11 for u, and use a 64-bit word to represent a line with no more than 32 pixels. Thus, the merging operation simply performs one bitwise-or operation on two 64-bit words. In this sense, the time complexity can also be viewed as $O(kl)$.

C. Propagation

This section applies the algorithm of line solving, described in Section II-B, to solving nonogram puzzle G. Let the given nonogram puzzle have a $w \times h$ rectangular grid G of pixels, and $p_{ij} \in \Gamma$ denote the pixel at row i and column j. Let the puzzle have h rows, denoted by $S_1^R, S_2^R, \ldots, S_h^R$, where each S_i^R represents $p_{i1}p_{i2}\ldots p_{iw}$, and w columns, denoted by $S_1^C, S_2^C, \ldots, S_w^C$, where each S_j^C represents $p_{1j}p_{2j}\ldots p_{hj}$. The propagation problem involves repeatedly painting pixels with values of u into 0 or 1 by applying line solving to all rows and all columns, until none of the pixels in the grid can be painted. In this section, we design an efficient routine, named PROPAGATE, to solve the propagation problem as follows.

---

procedure PROPAGATE(G)
1. $\Pi(G) \leftarrow \emptyset$; // $\Pi(G)$ is a set of updated pixels
2. *Put all rows and columns of G into $\mathcal{L}(G)$*
3. **while** ($\mathcal{L}(G) \neq \emptyset$) **do**
4. *Retrieve one line L from $\mathcal{L}(G)$*
5. *Let the line be S_i^R, that is, row i*; // for simplicity of discussion
6. **if** ($\mathrm{notFix}(S_i^R, D_i^R)$) **then** $\mathrm{status}(G) \leftarrow$ CONFLICT; **return**
7. $S_i^R \leftarrow \mathrm{Paint}(S_i^R, D_i^R)$
8. *Let Π be the newly painted pixels of S_i^R*
9. *For each pixel in Π, put its column into the list $\mathcal{L}(G)$, if not in $\mathcal{L}(G)$ yet*
10. $\Pi(G) \leftarrow \Pi(G) \cup \Pi$; // collect all painted cells in this propagate
11. **end while**
12. **if** (*all colors of pixels in G are in Σ*) **then** $\mathrm{status}(G) \leftarrow$ SOLVED
13. **else** $\mathrm{status}(G) \leftarrow$ INCOMPLETE

end procedure

In PROPAGATE, $\mathcal{L}(G)$ denotes a list of lines (columns or rows) in G that are to be solved. For simplicity of analysis, let $w = h = l$. Initially, the queue contains all $2l$ lines (l rows and l columns). During each iteration (between lines 4 and 10 in the pseudocode above), retrieve from $\mathcal{L}(G)$ only one line, say row i, without loss of generality. Then, S_i^R and D_i^R are the string and the description of the row, respectively. Then, use Fix to check (in line 6) whether a conflict is detected in the row. Assume that $\mathrm{Fix}(S_i^R, D_i^R)$ is false. Then, the row is not fixable for the puzzle, which implies a conflict with the hints of the puzzle in the current G. In this case, set the status of G to CONFLICT, then return. Alternatively, assume that $\mathrm{Fix}(S_i^R, D_i^R)$ is true. We continue to derive the maximal painting of S_i^R by $\mathrm{Paint}(S_i^R, D_i^R)$ (in line 7), and collect all the newly painted pixels of S_i^R into the set Π, that is, the pixels which were u but became 0 or 1. For each $p \in \Pi$, if p is in column j, column S_j^C needs to be further updated by line solving. Therefore, the column is put into the list $\mathcal{L}(G)$ (in line 9), unless it is already in the list. Set Π is merged into $\Pi(G)$, a set of newly updated pixels associated with G (in line 10). The propagation method is complete when no more lines can be retrieved from the list. Then, the status of G is set to SOLVED (in line 12) if all the pixels are painted, or INCOMPLETE (in line 13) otherwise.

The time complexity for PROPAGATE is $O(kl^3)$ for the following reason. From our method above, a line is put into the list $\mathcal{L}(G)$ in the following two cases: when list $\mathcal{L}(G)$ is initialized and when some pixels in the line are painted. For the former, at most $2l$ lines are put in the list (in line 2), while for the latter at most l^2 lines are put in the list. Thus, at most $2l + l^2$ lines are put into the list $\mathcal{L}(G)$. Since it takes $O(kl)$ time to solve each line, the total time complexity is $O(kl^3)$.

The routine PROPAGATE can be modified to support incremental propagation efficiently with the same time complexity $O(kl^3)$. By incremental propagation, we mean that the routine is performed iteratively whenever some of the pixels, denoted by Π, are newly painted between two consecutive iterations.

III. FULLY PROBING

In Section II, a propagation method is used to paint as many pixels as possible by using the line-solving algorithm. When no more pixels are to be painted after PROPAGATE, a common method is to employ backtracking to search all possible solutions. Yet another problem is which pixel to guess at in order to solve the puzzle more efficiently. This paper proposes an important technique, called fully probing (FP), to be used before running backtracking. In FP, we attempt to guess a color on each pixel to see whether more pixels can be painted. The FP method is different from backtracking in that the FP method is not recursive. That is, each pixel is only tested once, just enough to provide backtracking with more accurate guidance when choosing pixels to guess. The tradeoff is an increase in time complexity from $O(kl^3)$ to $O(kl^5)$ or $O(l^6)$. This paper proposes three FP methods, as described in Sections III-A–III-C, respectively. A discussion is given in Section III-D. The backtracking method is described in Section IV.

A. The First FP Method

The idea of FP is to make guesses for all unpainted pixels p_{ij} (with a value of u) in advance, and then perform propagation for each guess. The first proposed FP method, named FP1, maintains pairs of grids $(G_{p,0}, G_{p,1})$ for all pixels p, where both $G_{p,0}$ and $G_{p,1}$ represent the painted grids after guessing p to be 0 and 1, respectively. The routine for FP1 is described as follows.

procedure FP1(G)
1. *Initialize $G_{p,0}, G_{p,1}$ for all unpainted pixel p*
2. **repeat**
3. PROPAGATE(G)
4. **if** ($\mathrm{status}(G)$ *is* CONFLICT *or* SOLVED) **then return**
5. UPDATEONALLG(G)
6. **for** (*each unpainted pixel p in G*) **do**
7. PROBE(p)
8. **if** ($\mathrm{status}(G)$ *is* CONFLICT *or* SOLVED) **then return**
9. **if** ($\mathrm{status}(G)$ *is* PAINTED) **then break**
10. **end for**
11. **until** $\Pi(G) = \emptyset$

end procedure

Procedure PROBE(p)
1. PROBEG$(p, 0)$; // guess $p = 0$ and probe $G_{p,0}$
2. PROBEG$(p, 1)$; // guess $p = 1$ and probe $G_{p,1}$
3. **if** (*both* $\mathrm{status}(G_{p,0})$ *and* $\mathrm{status}(G_{p,1})$ *are* CONFLICT) **then** $\mathrm{status}(G) \leftarrow$ CONFLICT; **return**
4. **if** ($\mathrm{status}(G_{p,0})$ *is* CONFLICT) **then**
5. *Let Π be the set of newly painted pixels in $G_{p,1}$ with respect to G*
6. **else if** ($\mathrm{status}(G_{p,1})$ *is* CONFLICT) **then**
7. *Let Π be the set of newly painted pixels in $G_{p,0}$ with respect to G*
8. **else**
9. *Let Π be the set of pixels with the same value 0 or 1 in both $G_{p,0}$ and $G_{p,1}$ with respect to G*
10. **end if**
11. **if** ($\Pi \neq \emptyset$) **then** UPDATEONALLG(Π); $\mathrm{status}(G) \leftarrow$ PAINTED
12. **else** $\mathrm{status}(G) \leftarrow$ INCOMPLETE

end procedure

In the method, FP1 initializes $G_{p,0}$ and $G_{p,1}$ by letting them be G, then painting the pixel p with 0 and 1, respectively. Then, we repeatedly perform the instructions between lines 3 and 10, until no more pixels can be painted in G. Between lines 2 and 11, perform PROPAGATE on G initially, then return when G is done, where it will either have a status of CONFLICT or SOLVED. In the case where G is not yet complete, update all $G_{p,0}$ and $G_{p,1}$ from $\Pi(G)$ using the procedure UPDATEONALLG in line 5, where $\Pi(G)$ is the set of newly painted pixels in G by PROPAGATE. UPDATEONALLG updates the pixels in $\Pi(G)$ from G to all $G_{p,0}$ and $G_{p,1}$. Next, for all pixels p not painted in G yet, the procedure PROBE is performed.

The procedure PROBE probes, respectively, $G_{p,0}$ and $G_{p,1}$ initially by the procedure PROBEG. In the FP1 method, PROBEG(p, c) simply calls PROPAGATE$(G_{p,c})$. That is, both PROPAGATE$(G_{p,0})$ and PROPAGATE$(G_{p,1})$ are invoked initially. Then, we have the following four cases.

1) Both statuses of $G_{p,0}$ and $G_{p,1}$ are CONFLICT. In this case, G must be CONFLICT as well.
2) Only $G_{p,0}$ is CONFLICT. In this case, p must not be 0, that is, p is 1. Thus, G can be updated to $G_{p,1}$. In the routine, simply let Π be the set of newly painted pixels in $G_{p,1}$ with respect to G in line 5. The pixels in Π will be painted to G as described below.
3) Only $G_{p,1}$ is CONFLICT. In this case, G can be updated to $G_{p,0}$ in a similar way to the previous case.
4) Neither $G_{p,0}$ nor $G_{p,1}$ is CONFLICT. In this case, we try to find the pixels with the same painted colors in both $G_{p,0}$ and $G_{p,1}$, and put them into Π.

For the last three cases, Π indicates the set of pixels to be painted in G. If Π is not empty, the status of G is set to PAINTED, so that G will be updated from set Π. If Π is empty, G cannot be updated yet and the status of G is set to INCOMPLETE, which normally happens in the fourth case when no pixels with the same colors are found.

The time complexity of this method is only $O(kl^5)$, as explained in the following four aspects.

1) The routine PROBE is invoked $O(l^4)$ times. The loop between lines 2 and 11 in FP1 will be performed at most l^2 times, since at least one pixel is painted each time when G is PAINTED in line 9. The loop between lines 6 and 10 is performed clearly at most l^2 times.
2) The total time spent in PROPAGATE is $O(kl^5)$. For all the $2l^2 + 1$ grids, including G and all $G_{p,0}$ and $G_{p,1}$, we use incremental propagation to perform PROPAGATE more efficiently, in line 3 of FP1, and lines 1 and 2 of PROBE. Since the time for incremental propagation on each grid is $O(kl^3)$ from Section II-C, the total time for PROPAGATE is $O(kl^5)$.
3) It takes $O(l^4)$ to derive the newly painted pixels of all $G_{p,0}$ and $G_{p,1}$ in lines 5, 7, and 9 of PROBE, since the total number of these pixels is at most l^2 for each $G_{p,0}$ or $G_{p,1}$.
4) It takes $O(l^4)$ to perform UPDATEONALLG entirely. Since the set Π contains the newly painted pixels of G and the number of pixels that UPDATEONALLG uses to update all $G_{p,0}$ and $G_{p,1}$ is at most l^2, the number of pixel updates via UPDATEONALLG is $O(l^4)$.

B. The Second FP Method

In this section, first, we investigate some cases where the FP1 method cannot paint, then propose a second method FP2 to help paint more. The major problem of FP1 is that it does not make use of the contrapositive, that is, an implication $p_a \rightarrow p_b$ implies the contrapositive $\neg p_b \rightarrow \neg p_a$. In FP1, if we obtain $p_b = 1$ by performing PROBE on $G_{p_a,1}$ (assuming $p_a = 1$), then this implies $p_a \rightarrow p_b$, which implies its contrapositive $\neg p_b \rightarrow \neg p_a$ Then, the contrapositive implies $p_a = 0$ on $G_{p_b,0}$ (assuming $p_b = 0$). Unfortunately, the FP1 method does not request to set (or paint) $p_a = 0$ on $G_{p_b,0}$ in this case. These implications and contrapositives, such as $p_a \rightarrow p_b$ and $\neg p_b \rightarrow \neg p_a$, are also called pixel relations in this paper.

Consider the example shown in Fig. 3. In this example, we obtain $p_b = 1, p_a = 0$, and $p_b = 0$, by performing PROPAGATE (in PROBE) on the grids $G_{p_a,1}$, $G_{p_c,1}$, and $G_{p_c,0}$, respectively. In terms of logic, we have $p_a \rightarrow p_b, p_c \rightarrow \neg p_a$, and $\neg p_c \rightarrow \neg p_b$. From these, we can derive their corresponding contrapositives $\neg p_b \rightarrow \neg p_a, p_a \rightarrow \neg p_c$, and $p_b \rightarrow p_c$, respectively. These

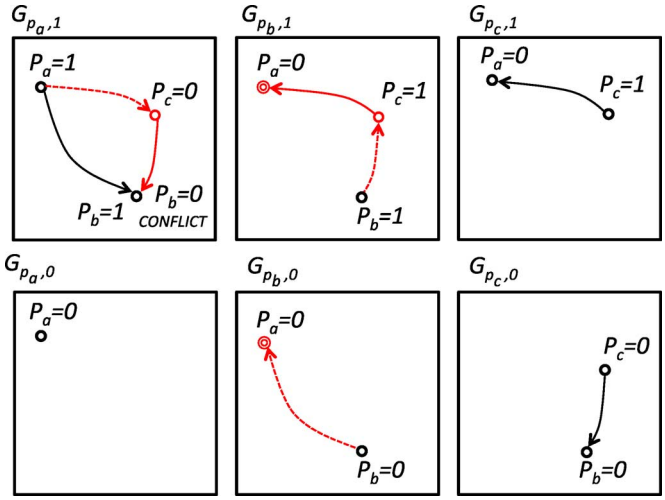


Fig. 3. Case where FP1 cannot find $p_a = 0$.

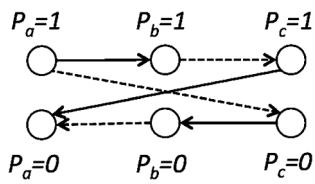


Fig. 4. Simplified presentation of Fig. 3.

pixel relations are depicted in Fig. 4. Note that the solid lines indicate the derivations via PROPAGATE, also called PROPAGATE implications, and the dashed lines indicate the contrapositives in the rest of this paper.

From Fig. 3 (or Fig. 4), it is easy to derive $p_a = 0$, since it is true in both cases p_b (due to $p_b \rightarrow p_c \rightarrow \neg p_a$ from above) and $\neg p_b$ (due to $\neg p_b \rightarrow \neg p_a$). The result can also be derived in another way by showing a conflict with the assumption $p_a = 1$, which leads to a contradiction $p_a \rightarrow p_b \rightarrow p_c \rightarrow \neg p_a$. Unfortunately, the FP1 method cannot obtain this result $p_a = 1$. The FP1 method can only derive those solid lines, and, therefore, is unable to derive $p_a = 1$.

In order to solve this problem, we propose the second FP method, named FP2, to add the contrapositives into grids as follows. In the new method, all grids, say $G_{p,0}$, are associated with a list of pairs of pixels and their values, denoted by $\mathrm{pclist}(G)$, used to record the derived values for given pixels, mainly for the contrapositive. For example, in Fig. 3, for $p_a \rightarrow p_b$, its contrapositive $\neg p_b \rightarrow \neg p_a$ is stored by putting the pair $(p_a, 0)$ into $\mathrm{pclist}(G_{p_b,0})$. When probing a grid, update the cells of the grid from the list before using PROPAGATE. For example, if $\mathrm{pclist}(G_{p_b,0})$ contains $(p_a, 0)$, the routine PROBE probing $G_{p_b,0}$ first sets the pixel p_a to 0 from $\mathrm{pclist}(G_{p_b,0})$, and then performs PROPAGATE.

The next issue is when to probe a grid. For this issue, we maintain a list of pixels p, denoted by $\mathcal{P}$, to indicate both grids $(G_{p,0}, G_{p,1})$ to be probed. Whenever a new contrapositive is added into $\mathrm{pclist}(G_{p,0})$ or $\mathrm{pclist}(G_{p,1})$, pixel p is put into $\mathcal{P}$, unless p is already in $\mathcal{P}$. The routine for FP2 is described as follows.

---

procedure FP2(G)
1. *Initialize* $G_{p,0}, G_{p,1}$ *for all unpainted pixel* p
2. *Initialize the list* $\mathcal{P}$ *to contain all unpainted pixels* p *in* G
3. PROPAGATE(G)
4. **if** (status(G) *is* CONFLICT *or* SOLVED) **then return**
5. **while** ($\mathcal{P} \neq \emptyset$) **do**
6. *Retrieve one pixel* p *from* $\mathcal{P}$
7. PROBE(p)
8. **if** (status(G) *is* CONFLICT *or* SOLVED) **then return**
9. **end while**

end procedure

---

procedure PROBEG(p, c)
1. *Update pixels of* $G_{p,c}$ *from the pixel list* $\mathrm{pclist}(G_{p,c})$
2. PROPAGATE($G_{p,c}$)
3. **if** $\mathrm{status}(G_{p,c})$ *is* CONFLICT **then return**
4. $\Pi \leftarrow \Pi(G_{p,c})$
5. **for** *each pixel* p' *with value* c' *in* Π **do**
6. *Put the contrapositive,* $(p, \bar{c})$, *into* $\mathrm{pclist}(G_{p',c'})$, *where* $\bar{c}$ *is complement of* c
7. *Add* p' *into* $\mathcal{P}$, *if* p' *is not in* P *yet*
8. **end for**

end procedure

---

procedure UPDATEONALLG(Π)
1. **for** *each* $p \in \Pi$ *with value* c *in* G **do**
2. *Put the pair* (p, c) *into all the lists of* G, *all* $G_{p,0}$ *and,* $G_{p,1}$
3. *Add* p' *into* $\mathcal{P}$, *if* p' *is not in* $\mathcal{P}$ *yet*
4. **end for**

end procedure

The FP2 method is changed as shown above. The routine PROBE is the same as the one in the FP1 method, except that the two routines PROBEG and UPDATEONALLG are modified as shown above. Now, examine how FP2 can obtain $p_a = 0$ in the example in Fig. 3. Here, we consider two situations. The first situation is to retrieve p_c from $\mathcal{P}$ earlier than p_a. In this situation, probe p_c is retrieved earlier than p_a. When probing p_c, store the contrapositive $p_a \rightarrow \neg p_c$ into $G_{p_a,1}$. Next, when probing p_a, set p_c initially to be 0 in $G_{p_a,1}$ from the contrapositive $p_a \rightarrow \neg p_c$, then perform PROPAGATE, which results in a conflict due to both $\neg p_c \rightarrow \neg p_b$ and $p_a \rightarrow p_b$. Thus, we establish that $p_a = 0$ in line 7 of PROBE, and UPDATEONALLG is subsequently called in line 11 to set it to all grids including G. The second situation is to retrieve p_a first from $\mathcal{P}$, then p_c and p_b. After probing p_a and p_c, two contrapositives $p_b \rightarrow p_c$ and $\neg p_b \rightarrow \neg p_a$ are stored in $G_{p_b,1}$ and $G_{p_b,0}$, respectively. Next, when probing p_b, we obtain $p_a = 0$ from $p_c = 1$ in $G_{p_b,0}$. Since $p_a = 0$ in both $G_{p_b,1}$ and $G_{p_b,0}$, we establish that $p_a = 0$ in line 9 of PROBE.

Consider another puzzle given in [2], as shown in Fig. 5(a), which actually has more than one solution. Batenburg and Kosters [2] mentioned that their method was unable to paint the

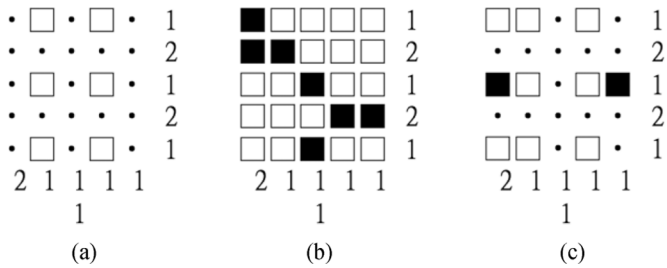


Fig. 5. (a) Puzzle given in [2]. (b) Initial $G_{p_{11},1}$, where p_{11} is in the upper left corner. (c) $G_{p_{53},0}$, where p_{53} is in the middle of the rightmost column.

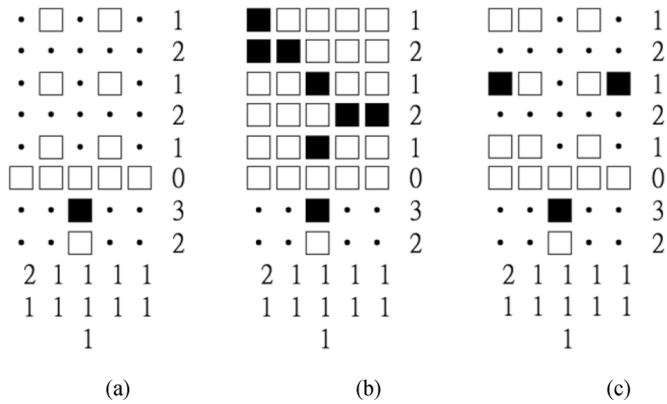


Fig. 6. An example modified from Fig. 5.

pixel p_{53} in white in the middle row of the rightmost column, though six white pixels can be painted as shown in Fig. 5(a). For this puzzle, FP1 cannot paint it either. However, FP2 can paint p_{53} in white in the following way. First, when probing pixel p_{11} in the upper left corner, we can obtain $G_{p_{11},1}$, as shown in Fig. 5(b). This implies that $p_{11} \rightarrow \neg p_{53}$. In FP2, we can obtain its contrapositive $p_{53} \rightarrow \neg p_{11}$, and, similarly obtain $p_{53} \rightarrow \neg p_{15}$. Then, when probing p_{53} for the grid $G_{p_{53},1}$, as shown in Fig. 5(c), performing PROPAGATE in the leftmost column will paint p_{13} (the middle of the leftmost column) in black, and then detect a conflict in the third row. Hence, we establish that p_{53} should be painted in white.

For this example, one might argue that the FP method would be sufficient if it simply returned a solution immediately, when probing on $G_{p_{11},1}$ yields a valid solution, then continuing from $G = G_{p_{11},0}$. Assume that the aforementioned method is used instead. We can demonstrate that a solution is not guaranteed simply by probing. The example illustrated in Fig. 6 is similar to Fig. 5, except there are now extra rows in the bottom of the puzzle that cause no solutions to be found like that in Fig. 5. Again, in this example, FP2 can paint pixel p_{53}, while FP1 and the method in [2] cannot.

The time complexity of FP2 remains $O(kl^5)$ only, as explained in the following two aspects.

1) For each of the $2l^2 + 1$ grids G and all $G_{p,0}$ and $G_{p,1}$, we still use incremental propagation to perform PROPAGATE, so the total time complexity for these is $O(kl^5)$, as described in Section III-A.
2) The routine PROBE is also invoked $O(l^4)$ times, since there are at most l^2 pairs of $(G_{p,0}, G_{p,1})$ in total, and each pair of $(G_{p,0}, G_{p,1})$ is put into the list $\mathcal{P}$ at most $2l^2 + 1$ times for the following reason. Each p for the pair of grids $(G_{p,0}, G_{p,1})$ is put into the list P only when it is being initialized, or when at least one pixel is newly painted (in PROBEG or UPDATEONALLG) in both $G_{p,0}$ and $G_{p,1}$, where there are at most $2l^2$ pixels. In addition, at most l^2 contrapositives for each of the $2l^2$ grids can be stored into *pclist* in line 6 in PROBEG, so the time for the loop between lines 5 and 8 is $O(l^4)$. Moreover, the time for UPDATEONALLG is $O(l^4)$, as described in Section III-A. Hence, the time for updating pixels in line 1 of PROBEG is $O(l^4)$ as well.

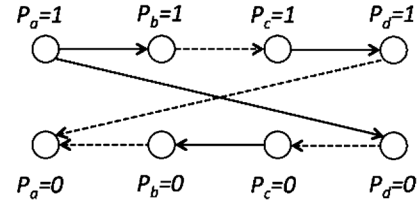


Fig. 7. Case where FP2 cannot find $p_a = 0$.

From the above, FP2 can paint more pixels over FP1 without a higher tradeoff in terms of time complexity, while FP1 has the advantage of being easier to implement.

C. The Third FP Method

In this section, we further discuss some cases where the FP2 method cannot paint, and then propose a third method, named FP3. However, the tradeoff is an increase in the time complexity to $O(l^6)$, slightly higher than $O(kl^5)$, where the value k is normally smaller than l.

The FP2 method is able to deduce PROPAGATE implications from contrapositives. For example, the contrapositive $p_b \rightarrow p_c$ in Fig. 7 helps deduce $p_b \rightarrow p_d$ as follows. After the contrapositive $p_b \rightarrow p_c$ is set from $\neg p_c \rightarrow \neg p_b$, the probing on $G_{p_b,1}$ results in $p_d = 1$ in $G_{p_b,1}$ by deducing from $p_c = 1$ to $p_d = 1$ in PROPAGATE. Unfortunately, FP2 cannot perform deductions that are the other way around. That is, the contrapositive $p_b \rightarrow p_c$ does not help deduce $p_a \rightarrow p_c$, since it involves going from a PROPAGATE implication to a contrapositive. Similarly, for the same reason, we cannot derive $\neg p_c \rightarrow \neg p_a$.

In fact, from PROPAGATE implications and their corresponding contrapositives in Fig. 7, condition $p_a = 0$ obviously holds from the following two deductions: $p_a \rightarrow p_b \rightarrow p_c \rightarrow p_d \rightarrow \neg p_a$ or $p_a \rightarrow \neg p_d \rightarrow \neg p_c \rightarrow \neg p_b \rightarrow \neg p_a$. Unfortunately, using FP2, we can only derive two more, $p_b \rightarrow p_d$ and $\neg p_d \rightarrow \neg p_b$, which do not contribute to the result of $p_a = 0$.

In order to solve this problem, we use a technique such as graph traversal for solving 2-SAT [6] in the FP3 method as follows. For a newly painted pixel in $G_{p,1}$, say $p' = 1$ (implying $p \rightarrow p'$), all newly painted pixels in $G_{p',1}$ (with respect to G) are also updated and painted in $G_{p,1}$. The update operation is done recursively for any newly painted pixels. For example, in Fig. 7, when updating $p_b = 1$ in $G_{p_a,1}$, also update $p_c = 1$ and $p_d = 1$ if both are already set in $G_{p_b,1}$. The routine PROBEG in the method is modified as follows.

1) In line 1, when painting pixel p' with the value c' for each pair (p', c') in list $\mathrm{pclist}(G_{p,c})$, update all the newly

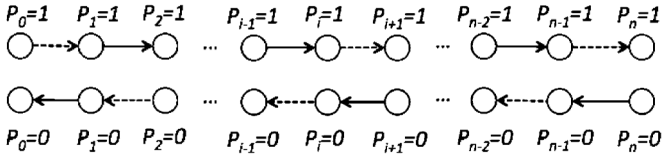


Fig. 8. General deduction.

painted pixels p'' from $G_{p',c'}$ to $G_{p,c}$, as described above, where p'' are the painted pixels in $G_{p',c'}$, including those in list $\mathrm{pclist}(G_{p',c'})$, but not in $G_{p,c}$. Note that a conflict occurs in $G_{p,c}$ when a pixel is set with different values.

2) In line 2, for each pixel p' that is newly painted with c' in PROPAGATE, update all the newly painted pixels from $G_{p',c'}$ to $G_{p,c}$ as above.
3) In line 4, add all the newly painted pixels in lines 1 and 2 into Π.

The above modification ensures that all deductions via PROPAGATE implications and contrapositives can be found. Namely, in the case shown in Fig. 8, we can deduce the following: 1) $p_i \rightarrow p_j$; and 2) $\neg p_j \rightarrow \neg p_i$, for all $i < j$.

Now, let us compare our method with the method proposed in [2] by Batenberg and Kosters without considering tomographic constraints.[1] Using the above modifications for FP3, all the pixels that are painted by their method as above can also be painted by FP3 for the following reason. Their method checks all relations between all pairs of pixels in the same lines (rows or columns) and then uses a 2-SAT method to derive more pixel relations, if any. In FP3, we probe a pixel with a color to derive all PROPAGATE implications to other pixels, not limited to the same line, and then use the above update operation (a kind of 2-SAT operation) to derive more pixel relations, if any. Since probing a pixel with a color derives more pixel relations (not limited to the same lines), all the pixel relations obtained in their method [2] are also obtained in FP3.

The time complexity of this method becomes $O(l^6)$ as explained in the following. For the first two modification items in the routine PROBEG, as specified above, the update operation for each newly painted pixel p takes $O(l^2)$ time to check whether grids $G_{p,0}$ and $G_{p,1}$ have more painted pixels to paint. Since the total number of newly painted pixels is $O(l^4)$, the total time for checking grids is $O(l^6)$.

D. Discussion

This section compares our methods (before backtracking) with those in [2]. From our analysis, the time complexity for each line solving, such as Fix in [2] (using DP) is estimated to be $O(kl^2)$, as mentioned above. In their 2-SAT method, they first calculated all the 2-SAT relations between any two pixels in the same line (rows or columns), and then used a 2-SAT algorithm to paint more pixels.

For the former, the method from [2] initially collects all the 2-SAT relations between any two pixels in the same line. It is easy to derive the number of pairs to be $O(l^3)$. Then, whenever a pixel is painted, it is required to recalculate all 2-SAT relations on any pair of pixels in the column or row containing it. Thus, the number of recalculated 2-SAT relations due to this painted pixel is $O(l^2)$. Since at most l^2 pixels are painted, the total number of recalculated relations is $O(l^4)$. Thus, the total time for calculating 2-SAT relations is $O(kl^6)$, which is higher than ours. Even if they use our line-solving method, the total time will still be $O(kl^5)$, the same as those of FP1's and FP2's.

For the latter, they used a 2-SAT algorithm to detect conflicts and, therefore, paint more pixels. Starting from each p among all l^2 pixels, the method is to explore a path to $\neg p$ by traversing the graph with $O(l^3)$ relations, as described above. Thus, it takes $O(l^5)$ time to detect conflicts for one round. Whenever one pixel is updated, the whole process needs to be performed for a new round. Thus, since there are at most l^2 pixels to be updated, the total time required is $O(l^7)$.

In regards to the number of pixels that are painted, our methods such as FP2 and FP3 are able to paint more pixels than the 2-SAT method. Examples are shown in Figs. 5 and 6. On the other hand, Section III-C also shows that all the pixels that the 2-SAT method can paint can also be painted by FP3. Our experiments in Section V also confirm this.

IV. BACKTRACKING

In many puzzles, all pixels in the grids can be completely painted following an FP method: FP1, FP2, or FP3. However, if the grids are not completely painted, we still need to use the backtracking method to paint the whole grid. In our backtracking method, first, we use an FP method to paint as many pixels as possible. If a conflict is not detected or a solution is not yet found in the grid, we use choose-pixel heuristics[2] in a routine named CHOOSEPIXEL to choose the next pixel p to extend. After choosing p, we paint the value 0 to p, then recursively call the backtracking method. The process is repeated similarly for painting 1 on p. The routine BACKTRACKING is written as follows.

---

procedure BACKTRACKING(G)
1. *INITIALIZE all $G_{p,0}$ and $G_{p,1}$ for all p*
2. FP3(G); // or FP1(G), FP2(G)
3. **if** ($\mathrm{status}(G)$ *is* CONFLICT) **then return**
4. **if** ($\mathrm{status}(G)$ *is* SOLVED) **then**
5. *Continue to solve more or stop depending on the requirement*.
6. **endif**
7. $p =$ CHOOSEPIXEL()
8. $G = G_{p,0}$
9. BACKTRACKING($G_{p,0}$)
10. $G = G_{p,1}$
11. BACKTRACKING($G_{p,1}$)

end procedure

The time complexity for backtracking is inherently exponential in the worst case. Thus, careful consideration must be given when choosing the next pixel p to extend in CHOOSEPIXEL, in order to achieve good performance. Fortunately, our FP

[1]Although the method may possibly paint more pixels, our experiments show little improvement, as discussed in Section V.

[2]The heuristics are similar to dynamic variable ordering heuristics in constraint satisfactory problem (CSP) [4].

methods have an important feature: all the grids $G_{p,0}$ and $G_{p,1}$ are probed in advance; that is, all p's are extended tentatively in advance. This feature offers more accurate information for choose-pixel heuristics. With this advantage, it is much easier to design choose-pixel heuristics when deciding which pixel to extend to. For example, we can choose pixel p that maximizes $m_{p,0} + m_{p,1}$, where value $m_{p,c}$ denotes the number of extra pixels painted in $G_{p,c}$, not in G. In general, the larger $m_{p,c}$ is, the better it is to extend on p. This is because extending on p with c generally results in more pixels painted. We tried many choose-pixel heuristics in our experiments in Section V.

V. Experiments

We performed experiments to analyze our methods for nonogram solvers, the results of which are described in this section. Most of our experiments were run using an Intel® Core™ i5-2400 CPU 3.10 GHz. In our experiments, two main sets of test puzzles were used for performance analysis. The first set contained 1000 25×25 puzzles generated at the TAAI 2011 Nonogram Solver Tournament [10], [16]. In this tournament, a nonogram random generator [23] was used to produce these puzzles at random, where the density of black cells in these 1000 puzzles ranged from 50% to 35%, in linearly decreasing order. It is worth noting that these puzzles do not guarantee unique solutions. According to [2], the most difficult puzzles contain about 20%–35% black cells; the higher the density of black cells (more than 35%), the easier the puzzles are. This implies that the generated puzzles (from 50% to 35%) were ordered roughly from easy to hard.

The second set contained 100 puzzles produced by Wu [23] in another nonogram tournament at TAAI 2011, where each participant generated 100 puzzles for opponents to solve. These puzzles were required to have unique solutions and were usually much harder. For simplicity of analysis, nonogram programs were required to find second solutions for all the puzzles. Since these puzzles all had unique solutions, the programs were forced to search completely.

In our experiments, the various choose-pixel heuristics that were tried are listed as follows.

1) Sum: Choose
$$p^* = \text{argmax}_p(m_{p,0} + m_{p,1}).$$
2) Min: Choose
$$p^* = \text{argmax}_p \, \mathrm{V}_{\min}(p)$$
where $\mathrm{V}_{\min}(p) = \min(m_{p,0}, m_{p,1})$.
3) Max: Choose
$$p^* = \text{argmax}_p \, \mathrm{V}_{\max}(p)$$
where $\mathrm{V}_{\max}(p) = \max(m_{p,0}, m_{p,1})$.
4) Mul: Choose
$$p^* = \text{argmax}_p \, ((m_{p,0} + 1) \times (m_{p,1} + 1)).$$
5) Sqrt: Choose
$$p^* = \text{argmax}_p \left(\mathrm{V}_{\min}(p) + \sqrt{\frac{\mathrm{V}_{\max}(p)}{\mathrm{V}_{\min}(p)+1}} \right).$$
6) Min-logm: Choose
$$p^* = \text{argmax}_p(\mathrm{V}_{\min}(p) + \mathrm{V}_{\log}(p, 0) \times \mathrm{V}_{\log}(p, 1))$$
where $\mathrm{V}_{\log}(p, c) = \log(1 + m_{p,c})+1$.
7) Min-logd: Choose
$$p^* = \text{argmax}_p(\mathrm{V}_{\min}(p) + |\mathrm{V}_{\log}(p, 0) - \mathrm{V}_{\log}(p, 1)|).$$

In our experiments, we also supported caching for all lines, as suggested in [20], since the solving of identical lines is highly likely. For example, a row to be solved may be the same as one already solved earlier. Since caching is not the core of this paper, we simply use a sufficiently large cache.

The experimental results are shown in Section V-A. Section V-B compares our nonogram solver with other solvers.

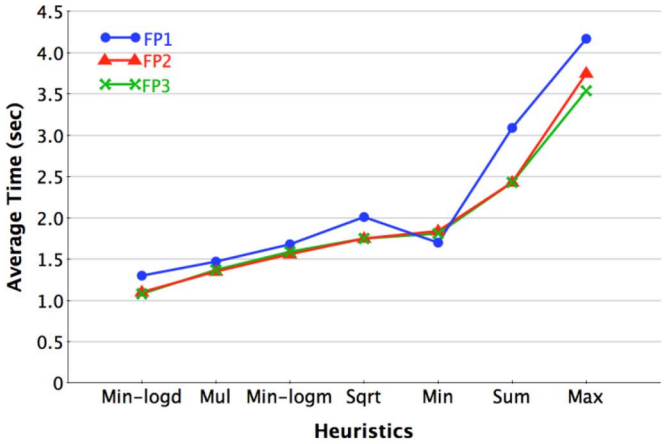
A. Performances of Test Cases

In this section, the two sets of puzzles were run with the seven choose-pixel heuristics and with the three FP methods, FP1, FP2 and FP3, respectively. The results are shown in Tables I and II, and the average times in Tables I and II are depicted in Fig. 9. The columns for "#Calls" indicate the average number of backtracking calls (calls to BACKTRACKING), and those for "Time" indicate the average time in seconds taken for each puzzle. In Tables I and II, the heuristics are sorted according to the corresponding times of FP3.

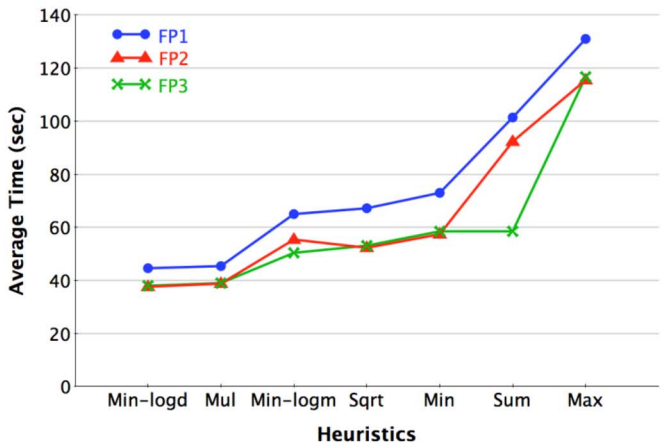
From Tables I and II, we discuss the results in the following three aspects. First, the number of calls for FP1 is generally higher than that for FP2 by a factor of 30%, while that for FP2 is very close to, specifically only 1% higher than, that for FP3. The results show that the improvement from FP1 to both FP2 and FP3 is significant and that from FP2 to FP3 is less significant. This implies that a significant improvement is achieved by adding contrapositives in FP2, as described in Section III-B. In contrast, propagating all the PROPAGATE implications and contrapositives in FP3 does not yield much improvement.

Second, in terms of computation time, both FP2 and FP3 run, in general, faster than FP1 by a factor of 19%. In fact, the FP2 method incurs some overhead over FP1 in maintaining a list of pixel values. The average time taken for each call in FP1 is about 0.55 ms, while those in both FP2 and FP3 are about 0.60 ms. The overhead for FP2 and FP3 is incurred by the maintenance of the data structure for pclist. FP3 runs only 1% faster than FP2. Although the overhead incurred by FP3 (over FP2) is high in theory, as shown in Section III-C, in practice the overhead is actually low if bitwise operations are used. The results show that the additional overhead for FP3 is less significant.

Third, the results in Tables I and II indicate that the Min-logd heuristic clearly performs the fastest in all cases. For this reason, we use Min-logd for the rest of the experiments.



(a)



(b)

Fig. 9. The average times for all heuristics using FP1, FP2, or FP3 for (a) the first set and (b) the second set.

TABLE I
PERFORMANCE RESULTS FOR THE FIRST SET OF TEST CASES

Choose-Pixel heuristics	FP1		FP2		FP3	
	#Calls	*Time*	*#Calls*	*Time*	*#Calls*	*Time*
Min-logd	2,821.1	1.30	1,199.0	1.10	1,195.2	1.08
Mul	3261.4	1.47	2488.6	1.35	2486.3	1.37
Min-logm	2,545.7	1.68	2,008.5	1.56	2,001.3	1.59
Sqrt	2,644.9	2.01	1,888.1	1.75	1,875.6	1.75
Min	2,685.1	1.70	2,064.1	1.84	2,048.2	1.81
Sum	6,175.5	3.09	4,177.3	2.43	4,139.0	2.43
Max	9,646.6	4.17	6,758.4	3.75	6,687.4	3.54

Next, we wanted to investigate the performance of FP methods from the perspective of the average numbers of painted pixels before backtracking. Table III shows the average numbers of painted pixels, after the initial PROPAGATE on the grid G and the first call to FP1, FP2, and FP3, respectively. The experimental results for the first set show that the initial PROPAGATE can solve only 32 pixels on average, but the three

TABLE II
PERFORMANCE RESULTS FOR THE SECOND SET OF TEST CASES

Choose-Pixel heuristics	FP1		FP2		FP3	
	#Calls	*Time*	*#Calls*	*Time*	*#Calls*	*Time*
Min-logd	60,618.3	44.6	42,814.3	37.6	42,646.6	38.0
Mul	68,678.3	45.4	50,407.0	38.8	50,263.8	39.0
Min-logm	85,184.3	65.0	57,975.3	55.4	57691.6	50.4
Sqrt	89,538.6	67.2	63,333.6	52.3	63085.8	53.1
Min	89,807.7	73.0	60,853.1	57.4	60,634.8	58.5
Sum	190,544.5	101.4	155,508.6	92.3	155,168.1	93.6
Max	247,925.2	131.0	197,864.7	115.5	197,112.5	116.7

TABLE III
THE AVERAGE NUMBERS OF PAINTED PIXELS AFTER INITIAL PROPAGATE, THE 2-SAT METHOD IN [2], FP1, FP2, AND FP3

Average number of painted pixels using Min-logd	PROPAGATE	2-SAT	FP1	FP2	FP3
The first set	32.3	94.9	201.2	210.6	211.3
The second set	8.9	19.5	39.2	43.2	43.2

methods FP1, FP2, and FP3 improve by painting 201–211 pixels, about 170 more pixels. The results also show that both FP2 and FP3 are clearly better than FP1 by solving 9.4 and 10.1 more pixels on average, respectively. However, the improvement from FP2 to FP3 is very small, with only 0.7 more pixels on average. For the second set, the experimental results show a similar relation. However, since the set of puzzles is more difficult, the numbers of painted pixels are generally smaller than those in the first set.

Next, we wanted to compare our method with the method proposed in [2] without DT (as mentioned in Section III-C). In the column "2-SAT" of Table III, the values indicate the average numbers of painted pixels after finishing the 2-SAT method. From the table, FP3 can solve 116.4 more pixels on average for the first set and 23.7 more pixels for the second set. This shows that our FP methods can paint many more pixels than theirs, while our methods also perform more efficiently (as described in Section III-D). In addition to 2-SAT, Batenburg [1] and Batenburg and Kosters [2] also proposed a DT method to help paint more pixels. However, our experiments also show that a 2-SAT method together with DT solves almost the same number of pixels as that for the 2-SAT method alone, but it takes much more time. More specifically, for all the puzzles in the above two sets (1100 puzzles), the 2-SAT method with DT can paint in total only five more pixels than the 2-SAT method. This shows that DT is not critical when attempting to paint more pixels.

B. Comparison to Other Solvers

In this section, we compare our nonogram solver with other solvers in other sets of puzzles. First, consider the first set of

TABLE IV
5000 PUZZLES GIVEN IN [21]

Solver	*0.00s to 0.10s*	*0.10s to 1.00s*	*1.00s to 10.0s*	*10.0s to 60.0s*	*1.0m to 2.0m*	*2.0m or more*	*Average Time (sec)*
LalaFrogKK	4,910	63	14	3	1	0	0.07
NAUGHTY (naugty, v88)	4,286	517	183	9	5	0	0.28
PBNSOLVE (pbn, 1.09)	4,530	333	97	19	7	14	8.65

1000 puzzles from the TAAI 2011 Nonogram Solver Tournament, as mentioned in Section V-A. In this tournament, our program named LALAFROGKK solved all 1000 puzzles in 645 s [10] on a machine equipped with Intel® Core™ i7-970 CPU 3.60 GHz, and outperformed the other two teams, YAYAGRAM and NAUGHTY. YAYAGRAM solved 766 puzzles in 120 min (the time limitation of this tournament), while NAUGHTY solved 509 puzzles.

Second, consider the set of 5000 puzzles generated by a nonogram random generator in [21] where the density of black cells is about 50%. The site also collected many public nonogram solvers, including NAUGHTY, but not yet LALAFROGKK. NAUGHTY was ranked the fastest among all the programs on this site, and PBNSOLVE was ranked the second fastest. Table IV shows the computation times of the two programs as well as ours, after we reran all 5000 puzzles for all three solvers on an Intel® Core™ i5-2400 CPU 3.10 GHz. The result shows that the average computation time for LALAFROGKK is the fastest one, and that the number of puzzles that required over 1 min to solve is only one for LALAFROGKK, but five for NAUGHTY and 21 for PBNSOLVE.

VI. CONCLUSION

This paper proposes some methods to design an efficient nonogram solver. The contributions are listed as follows.

1) We propose a fast DP method for line solving, whose time complexity in the worst case is $O(kl)$ only, where the grid size is $l \times l$, and k is the average number of integers in one constraint, always smaller than l. In contrast, the DP method in [2] is $O(kl^2)$.
2) We propose three FP methods, named FP1, FP2, and FP3, respectively, to solve more pixels before backtracking. This contribution has significant value in the following two senses. First, both FP2 and FP3 can solve more pixels than the method in [2] (before backtracking), while the time complexities for FP2 and FP3 are faster than theirs by a factor of $O(l)$.
   Second, these FP methods can provide backtracking with useful guidance when choosing the next pixel to guess. Our experiments also showed that FP3 performed the fastest, improving performance over FP1 significantly, but improving performance over FP2 only slightly.
3) We incorporate the proposed methods into a fast nonogram solver, named LALAFROGKK, which won a nonogram tournament at TAAI 2011. In addition, in the 5000 test cases given in [21], our program clearly outperformed other programs collected in [21].

The FP method proposed in this paper is actually a generic method for many puzzle problems. In practice, we are applying this method to some other puzzle problems such as *Nurikabe* [22], *Slitherlink* [12], and *Sudoku* [11]. We believe this method is an important generic method for solving puzzles like dancing links [9].

APPENDIX

In this Appendix, we design a line-solving algorithm whose time complexity is $O(kl)$. We follow the definitions and notation in Section II-B2. The algorithm maintains three arrays: $\mathrm{fix}_0[l]$, $\mathrm{fix}_1[l]$, and $\mathrm{low}_1[l]$. Each $\mathrm{fix}_0[i]$, initialized to be false, indicates whether there exists some fix $S \in \Psi(S, D)$ with the ith pixel 0. Similarly, each $\mathrm{fix}_1[i]$ indicates for the ith pixel 1. Each $\mathrm{low}_1[i]$ is the lowest index j such that $s_j, s_{j+1}, \ldots, s_i$ are all 1's.

The algorithm starts calculating from $\mathrm{Paint}(l, k)$, and uses memorization to prevent repeatedly calculating recurrences for both $\mathrm{Fix}(i, j)$ and $\mathrm{Paint}(i, j)$, as described in [7]. $\mathrm{Paint}(i, j)$ is calculated only when $\mathrm{Fix}(i, j)$ is true. Similarly, $\mathrm{Paint0}(i, j)$ is calculated only when $\mathrm{Fix0}(i, j)$ is true, and $\mathrm{Paint1}(i, j)$ is calculated only when $\mathrm{Fix1}(i, j)$ is true.

Assume that $\mathrm{Fix0}(i, j)$ is true. Then, set Ψ_0 must be nonempty. This implies that there exists some $S \in \Psi_0$ with the ith pixel 0, and this also implies that there exists some $S \in \Psi(S, D)$ with the ith pixel 0. Thus, in $\mathrm{Paint0}(i, j)$, also set $\mathrm{fix}_0[i]$ to true.

Similarly, assume that $\mathrm{Fix1}(i, j)$ is true. Then, set Ψ_1 must be nonempty. This implies that there exists some $S \in \Psi_1$ such that $s_{i-d_j} s_{i-d_j+1} \ldots s_i = \sigma(d_j)$. Since $\sigma(d_j) = 01^{d_j}$, we have $s_{i-d_j} = 0$ and $s_{i-d_j+1} \ldots s_i = 1^{d_j}$. A straightforward method for $\mathrm{Paint1}(i, j)$ is to set $\mathrm{fix}_0[i - d_j]$ to true, and set all $\mathrm{fix}_1[i - d_j + 1], \ldots, \mathrm{fix}_1[i]$ to true. However, the computation time increases the order, using this method. So, we set $\mathrm{fix}_1[i]$ to true, and $\mathrm{low}_1[i]$ to $\min(\mathrm{low}_1[i], i - d_j + 1)$. If $i - d_j + 1$ is lower than the original $\mathrm{low}_1[i]$, $\mathrm{low}_1[i]$ becomes $i - d_j + 1$ to indicate implicitly that all $\mathrm{fix}_1[i - d_j + 1], \ldots, \mathrm{fix}_1[i]$ are set to true. On the other hand, if the original $\mathrm{low}_1[i]$ is lower than $i - d_j + 1$, all of these fix_1 are implicitly set to true already, so no more operations are needed.

When $\mathrm{Paint}(l, k)$ is completed, first, we scan all $\mathrm{low}_1[i]$ from s_l back to s_1 and set the corresponding fix_1 according to $\mathrm{low}_1[i]$. It is not hard to perform the above operation in linear time. Then, determine the final $\mathcal{S}(\Psi(S, D))$ by merging all $\mathrm{fix}_0[i]$ and $\mathrm{fix}_1[i]$, in the following way. For all i, if $\mathrm{fix}_0[i]$ is true and $\mathrm{fix}_1[i]$ is false, set $s_i = 0$; if $\mathrm{fix}_0[i]$ is false and $\mathrm{fix}_1[i]$ is true, set $s_i = 1$; and if both $\mathrm{fix}_0[i]$ and $\mathrm{fix}_1[i]$ are true, set $s_i = u$. Note that it is impossible to have both $\mathrm{fix}_0[i]$ and $\mathrm{fix}_1[i]$ to be false, since set $\Psi(S, D)$ is nonempty if $\mathrm{Fix}(l, k)$ is true.

The time complexity of the above algorithm is only $O(kl)$ for the following reason. The time for merging all $\mathrm{fix}_0[i]$ and $\mathrm{fix}_1[i]$ is $O(l)$. The time for calculation on each $\mathrm{Fix}(i, j)$ and

$\mathrm{Paint}(i, j)$ is $O(1)$. Thus, the total time for calculating all $\mathrm{Fix}(i, j)$ and $\mathrm{Paint}(i, j)$ is $O(kl)$.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees and T.-Han Wei for their valuable comments.

REFERENCES

[1] K. J. Batenburg, "A network flow algorithm for reconstructing binary images from discrete X-rays," *J. Math. Imag. Vis.*, vol. 27, no. 2, pp. 175–191, 2007.
[2] K. J. Batenburg and W. A. Kosters, "Solving nonograms by combining relaxations," *Pattern Recognit.*, vol. 42, no. 8, pp. 1672–1683, 2009.
[3] K. J. Batenburg, S. Henstra, W. A. Kosters, and W. J. Palenstijn, "Constructing simple nonograms of varying difficulty," *Pure Math. Appl.*, vol. 20, pp. 1–15, 2009.
[4] F. Bacchus and P. van Run, "Dynamic variable ordering in CSPs," in *Principles and Practice of Constraints Programming (CP-95)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 1995, vol. 976, pp. 258–277.
[5] R. A. Bosch, "Painting by numbers," *Optima*, vol. 65, pp. 16–17, 2001.
[6] D. Cohen, P. Jeavons, and M. Gyssens, "A unified theory of structural tractability for constraint satisfaction problems," *J. Comput. Syst. Sci.*, vol. 74, no. 5, pp. 721–743, 2008.
[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[8] F. Faase, "Nonogram to exact cover," [Online]. Available: http://www.iwriteiam.nl/D0906.html#28
[9] D. E. Knuth, "Dancing links," in *Millenial Perspectives in Computer Science*. Basingstoke, U.K.: Palgrave, 2000, pp. 187–214.
[10] H.-H. Lin, D.-J. Sun, I.-C. Wu, and S.-J. Yen, "The 2011 TAAI computer-game tournaments," *Int. Comput. Games Assoc. J.*, vol. 34, no. 1, pp. 51–54, 2011.
[11] H.-H. Lin and I.-C. Wu, "An efficient approach to solving the minimum Sudoku problem," *Int. Comput. Games Assoc. J.*, vol. 34, no. 4, pp. 191–208, Dec. 2011.
[12] T.-Y. Liu, I.-C. Wu, and D.-J. Sun, "Solving the slitherlink problem," in *Proc. Conf. Technol. Appl. Artif. Intell.*, 2012, pp. 284–289.
[13] M. Olšák and P. Olšák, "Griddlers solver," [Online]. Available: http://www.olsak.net/grid.html#English
[14] The Puzzle Museum, "Origins of cross reference grid & picture grid puzzles," 2012 [Online]. Available: puzzlemuseum.com/griddler/gridhist.htm
[15] S. Simpson, "Nonogram solver," [Online]. Available: www.comp.lancs.ac.uk/~ss/nonogram/
[16] D.-J. Sun, K.-C. Wu, I.-C. Wu, S.-J. Yen, and K.-Y. Kao, "Nonogram tournaments in TAAI 2011," *Int. Comput. Games Assoc. J.*, vol. 35, no. 2, Jun. 2012.
[17] N. Ueda and T. Nagao, "NP-completeness results for nonogram via parsimonious reductions," Dept. Comput. Sci., Tokyo Inst. Technol., Tokyo, Japan, Tech. Rep. TR96-0008, 1996.
[18] W. A. Wiggers, "A comparison of a genetic algorithm and a depth first search algorithm applied to Japanese nonograms," in *Proc. Twente Student Conf. IT*, Jun. 2004, pp. 1–6.
[19] Wikipedia, "Nonogram," [Online]. Available: en.wikipedia.org/wiki/Nonogram
[20] J. Wolter, "Effect of line solution caching on Pbnsolve run-times," [Online]. Available: http://webpbn.com/survey/caching.html
[21] J. Wolter, "The 'Pbnsolve' paint-by-number puzzle solver," [Online]. Available: http://webpbn.com/pbnsolve.html
[22] I.-C. Wu, D.-J. Sun, and S.-J. Yen, "HAPPYNURI wins Nurikabe tournament," *Int. Comput. Games Assoc. J.*, vol. 33, no. 4, p. 236, Dec. 2010.
[23] K.-C. Wu, "TAAI2011 Nonogram Tournament result," 2012 [Online]. Available: http://kcwu.csie.org/~kcwu/nonogram/taai11/
[24] C.-H. Yu, H.-L. Lee, and L.-H. Chen, "An efficient algorithm for solving nonograms," *Appl. Intell.*, vol. 35, no. 1, pp. 18–31, 2009.

I.-Chen Wu (M'10) received the B.S. degree in electronic engineering and the M.S. degree in computer science from the National Taiwan University (NTU), Taipei, Taiwan, in 1982 and 1984, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1993.

Currently, he is with the Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan, and also serves as the Director of the Institute of Multimedia Engineering. His research interests include artificial intelligence, Internet gaming, volunteer computing, and cloud computing.

Dr. Wu introduced the new game, *Connect6*, a kind of six-in-a-row game, in 2005. Since then, *Connect6* has become a tournament item in the Computer Olympiad. He led a team developing various game-playing programs, winning over 20 gold medals in international tournaments, including the Computer Olympiad. He wrote over 80 papers, and served as a chair and a committee member in over 30 academic conferences and organizations, including the Games Technical Committee of the IEEE Computational Intelligence Society.

Der-Johng Sun received the M.S. degree in botany from the National Taiwan University (NTU), Taipei, Taiwan, in 1996. He is currently working toward the Ph.D. degree in computer science at the National Chiao-Tung University, Hsinchu, Taiwan.

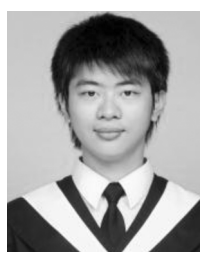
His research interests include information extraction, artificial intelligence, and cloud computing.

Lung-Pin Chen received the B.S. degree from Soochow University, Suzhou, Jiangsu, China, in 1991, the M.S. degree from the National Chung-Cheng University, Minxueng, Chiayi, Taiwan, in 1993, and the Ph.D. degree from the National Chiao-Tung University, Hsinchu, Taiwan, in 1999, all in computer science.

He is an Associate Professor at the Department of Computer Science and Information Engineering, Tung-Hai University, Taichung, Taiwan. His research interests include distributed algorithm, service computing, and pervasive computing.

Kan-Yueh Chen received the M.S. degree in computer science from the National Chiao-Tung University, Hsinchu, Taiwan, in 2012.

His research interests include artificial intelligence and cloud computing.



Ching-Hua Kuo received the M.S. degree in computer science from the National Chiao-Tung University, Hsinchu, Taiwan, in 2012.

His research interests include artificial intelligence and cloud computing.

Hao-Hua Kang received the M.S. degree in computer science from the National Chiao-Tung University, Hsinchu, Taiwan, in 2012.

His research interests include artificial intelligence and cloud computing.

Hung-Hsuan Lin received the B.S. degree in computer science from the Department of Computer Science, National Chiao-Tung University, Hsinchu, Taiwan, in 2007, where he is currently working toward the Ph.D. degree in computer science.

His research interests include artificial intelligence, computer game, volunteer computing, and cloud computing.