A New Simplified Line Solver for Nonogram Puzzle Games

Lung-Pin Chen and Ching-Yi Hung

Abstract—Nonogram is a typical two-dimensional logical puzzle game of which the player paints each pixel by considering the constraints on the row and column intersecting on that pixel. Solving Nonogram puzzle efficiently has been considered challenging. The recently proposed approaches can efficiently solve many puzzles via the logical deduction based on 2-SAT formulas. In this paper, we first propose a simplified form of the recurrence function in the previous Nonogram algorithm. Our work simplifies the recurrence function from 7 formulas down to 3 formulas.

1 Introduction

A puzzle grid is a $h \times w$ 2D rectangular grid in which the individual entries, called *pixels*, are denoted by a_{yx}, $1 \leq y \leq h$, $1 \leq x \leq w$. In the grid, row y (or column x) refers to the collection of the pixels a_{yx} for all x from 1 to w (or for all y from 1 to h). A row or a column is referred to as a *line*. With loss of generality, this paper assumes that $h = w$ in order to simplify the algorithm representation. The proposed algorithm can be extended to arbitrary height and width straightforwardly.

A Nonogram puzzle with a $w \times w$ puzzle grid contains $2w$ lines, each is associated with a *line descriptor*. Hereinafter, line i refers to a row if $i \leq w$ and refers to a column otherwise (*i.e.* $i > w$). For each line i, the line descriptor $D_i = \langle d_{i,1}, d_{i,2}, \cdots, d_{i,k_i} \rangle$ is an ordered tuple of *clues* $d_{i,j}$. The goal of solving the puzzle is to paint all the pixels based on the constraints

on rows and columns given in the line descriptors as defined as follows.

The pixel color of a line is represented as a string (a sequence of charactors) $L = s_1 s_2 \cdots s_w$, where w denotes the number of pixels, and each charactor s_i represents the color of the i-th pixel in the line. The value of s_i, $1 \leq i \leq w$, must be 0, 1, or u, respectively representing *white*, *black* or *unknown* colors. For mathematical convinienece, for those invalid position indices, we define that

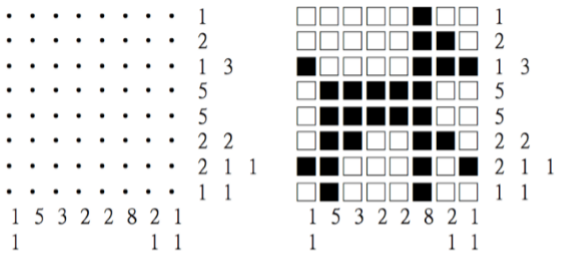

Figure 1. (a) A nonogram puzzle. (b) A solution painting.

$$s_i = \phi \text{ if } i < 1 \text{ or } i > w \tag{1}$$

Note that the null value ϕ occurs only when the index goes out of the range from 1 to w.

For a string L, a substring $L' = s_b s_{b+1} \cdots s_{b+l-1}$ starting from index b with length l forms a *segment* in terms of string L and color $\mathcal{C}$ if and only if

- $s_i = \mathcal{C}$, for all i, $b \leq i \leq b + l - 1$, and
- $s_{b-1} \neq \mathcal{C}$ and $s_{b+l} \neq \mathcal{C}$

In this definition, $s_{b-1} \neq \mathcal{C}$ occurs when s_{b-1} has different color than s_b's, or the index of s_{b-1} goes out of the bound, as mentioned in Equation 1. A segment with all black (or white) characters is called a *black segment* (or a *white segment*).

A Nonogram puzzle game requires to paint all the pixels of the grid to either black or white such that the segments of each line match the line descriptors. The formal constraints are described as follows: For each line i and its clue set D_i,

- Line i has exactly k_i black segments, and
- The length of j-th black segments is equal to j-th clue in line descriptor D_i, $1 \leq j \leq k_i$.

For example, in the puzzle shown in Figure 1 (b), line 3 has two black segments with length 1 and 3. The painting matches the clues of the line descriptor $D_3 = \langle 1, 3\rangle$ and $k_3 = 2$. Since the painting of all the lines satisfy the constraints, the painting shown in Figure 1 (b) is a valid solution.

Following [12], the patterns of strings representing the pixels of the lines are expressed in regular expression notation. Thus, given the description $D = \langle d_1, d_2, \cdots, d_k\rangle$, the pixels of the line L is consistent with D if it matches the following pattern [12]:

$$0^*1^{d_1}0^+1^{d_2}0^+\cdots 0^+1^{d_k}0^*$$

Let $\Gamma = \{0, 1, u\}$ be the finite alphabet. Γ^w denotes the set of all possible strings of the color characters with length w over Γ. When solving a puzzle grid, a line can be partially painted since the colors of some pixels remain unknown. Given a string S, a *fix* operation paints some unknown-color pixel (*i.e.* character) in S to white or black color. Formally, let $S = s_1s_2\cdots s_k \in \Gamma^w$ and $S' = s'_1s'_2\cdots s'_k \in \Gamma^w$ be two strings. String S' is a *fix* of S if and only if $S = S'$ except that $s_i = u$ and $s'_i = 0 \mid 1$ for some i (recall that $\mid$ represents 'or' in regular expression notation). The relation that S' is a fix of S is denoted by $S \to S'$. Furthermore, if there exists a sequence of $S_1, S_2, \cdots, S_m$ satisfying $S = S_1 \to S_2 \to \cdots \to S_m = S'$, we said that S' is fixable from S and also denote it by $S \to S'$. For example, $u1u0 \to u110 \to 0110$, thus, 0110 is fixable from $u1u0$.

2 New Simplified Maximal Painting

2.1 Previous Maximal Painting Algorithm

This section briefly describes the max-paint functions proposed in [12]. The function Fix tests fixability of line $L[i, i + l]$ in terms of its descriptor $D[j, j + m]$, defined as follows:

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

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

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

```
P(4,1)=P0(3,1)|P1(0,0)
u11u.........

P(5,1)=P0(4,1)|P1(1,0)    P(5,2)=P0(4,2)|P1(1)
uu1uu.........            ..............

                          P(6,2)=P0(5,2)|P1(2)
                          ..............

                          P(7,2)=P0(6,2)|P1(3)
                          1110111.......

                          P(8,2)=P0(7,2)|P1(4)    P(8,3)=P0(7,3)|P1(3,2)
                          u11uu11u......          ..............

                          P(9,2)=P0(8,2)|P1(5)    P(9,3)=P0(8,3)|P1(4,2)
                          uu1uuu1uu.....          ..............

                                                  P(10,3)=P0(9,3)|P1(5,2)
                                                  ..............

                                                  P(11,3)=P0(10,3)|P1(6,2)
                                                  ..............

                                                  P(12,3)=P0(11,3)|P1(7,2)
                                                  111011101111..

                                                  P(13,3)=P0(12,3)|P1(8,2)
                                                  u11uu11uu111u.

                                                  P(14,3)=P0(13,3)|P1(9,2)
                                                  uu1uuu1uuu11uu
```

Figure 2. Dynamic programming table of aligning clue $(3, 3, 4)$ on a 14-width line.

The function Paint paint all the fixiable pixels, defined as follows:

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

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

$$\mathrm{Paint1}(i,j) = \mathrm{Paint}(i-d_j-1, j-1)\cdot 0\cdot\sigma(L_j)$$

The details of the proof are ignored in this paper.

2.2 Simplified Maximal Painting

We define that a painting string s^w, $s \in \{\epsilon, 0, 1, \mathrm{u}\}$ with the following meanings:

- ϵ: null or conflicted
- 0: white color
- 1: black color
- u: unkown color

Hereinafter, we use $s[i]$ to refer to the i-th alphabet in string s. Also, we use $s[i, i']$ to refer to the substring of s,

starting from position i to position i'. Also, if a string s matches a regular expression reg, we denote it by $s \perp reg$. The following $P(i,j)$ function is called a *line solver* since it only paints pixels for a single line.

$\mathrm{P}(i,j)$ is called a *maximal painting* (*abbr.* max-painting) in terms of substring $s[1..i]$ in terms of clues $clue[1..j]$ as defined as follows:

$$\mathrm{P}(i,j) = \begin{cases} \mathrm{merge}(\mathrm{P0}(i,j), \mathrm{P1}(i,j)), \\ \quad \text{if } i > 0 \text{ and } j \geq 0 \\ \\ \epsilon^i, \text{ otherwise} \end{cases}$$

where

$$\mathrm{P0}(i,j) = \begin{cases} \mathrm{P}(i-1,j) \cdot 0, \\ \quad \text{if } s[i] \perp (0 \mid \mathrm{u}) \\ \\ \epsilon^i, \text{ otherwise} \end{cases}$$

and

$$\mathrm{P1}(i,j) = \begin{cases} \mathrm{P}(i-d_j-1,\, j-1) \cdot 0 \cdot 1^{d_j}, \\ \quad \text{if } s[i-dj, i] \perp ((0 \mid \mathrm{u}) \cdot (1 \mid \mathrm{u})^{dj}) \\ \\ \epsilon^i, \text{ otherwise} \end{cases}$$

The above merge function is employed to combine two strings as described as follows. Let $s_3 = \mathrm{merge}(s1, s2)$, $s3[i] = s1[i] \oplus s2[i]$ for each i, where $\oplus$ is an XOR-like operation that treats u as 0 or 1 simultaneously. Let $a, b \in \{\epsilon, 0, 1, u\}$, if both a and b are the same, $a \oplus b$ results in a (or b), otherwise, results in u. The formal definition is as follows:

- $\epsilon \oplus b = b$
- $\mathrm{u} \oplus b = \mathrm{u}$
- $0 \oplus b = 0$ if $b \in \{0, \epsilon\}$
- $0 \oplus b = \mathrm{u}$ if $b \in \{1, \mathrm{u}\}$
- $1 \oplus b = 1$ if $b \in \{1, \epsilon\}$
- $1 \oplus b = \mathrm{u}$ if $b \in \{0, \mathrm{u}\}$

Note that $\oplus$ is symmetric since $a \oplus b = b \oplus a$.

Figure 2 illustrates the dynamic programming table for aligning clue $(3,3,4)$ on a 14-width line. When implement, we represent the color code $\epsilon, 0, 1, \mathrm{u}$ to binary code $00, 01, 10, 11$, respectively. Thus, the $\oplus$ operation can be implemented by using the bitwise OR operator, as shown in Figure 2.

2.3 Ordered Propagation

The line solver finds maximal painting for a single line. After performing a max-panting, a pixel is said to be *triggered* if it is chaged from unsolved state to solved (*i.e.* is changed from color u to color 0 or 1). Moreover, if maximal-painting

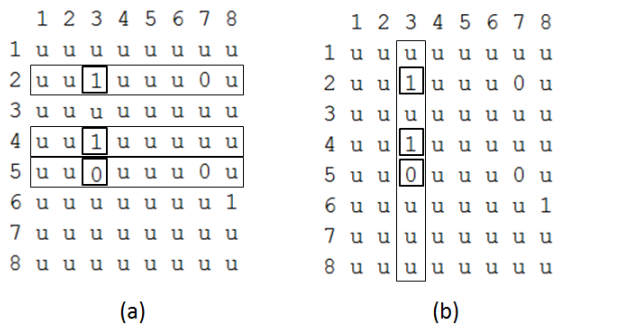

Figure 3. Ordered propagation: (a) performing max-paint on row $2, 4, 5$, and triggering column 3. (b) performing max-paint on column 3.

a row i, the triggered pixel $a(i,j)$ and triggered column j. if maximal-painting a column j, the triggered pixel $a(i,j)$ and triggered row i.

To solve the whole puzzle grid, the Nonogram algorithm [12] repeatedly invokes the PROPAGATION operation to obtain the maximal painting for all lines, as described as follows:

Procedure PROPAGATION(Puzzle G, Queue Q):

- While Q is not empty
  1) Let line L = delete a line from Q
  2) Do MAXPAINT(L)
  3) Add all the lines triggered in MAXPAINT to Q

In the above algorithm, during propagating, changing a pixel's color will trigger a sequence of max-painting on rows and columns alternatively. The max-paint operations incrementally paint more unsolved pixels, and never turn a solved pixel back to unsolved.

Observing that in the above propagation procedure, a same column (or a same row) can be triggered more than once if multiple rows have their painting updating the same position. For example, in Figure 3 (a), the propagation procedure performs max-paint on rows $2, 4, 5$, which update pixels $(2,3), (4,3), (6,3)$ respectively. In this case, all three rows trigger column 3. In worst case, the Nonogram puzzle algorithm may perform one propagation for each trigger.

In this paper, we propose the *ordered propagation*, which tries to enhance the performance by aggregating multiple max-paints, on a same line, into one. In the new approach, we separate the procedure into two stages. The first stage only paints on rows. These operations painting on rows shall trigger columns. All the columns are batch in do max-painting are painted in the second stage. The ordered propagation procedure is described as follows:

Procedure ORDERED_PROPAGATION(Puzzle G, Queue Q):

- While Q is not empty

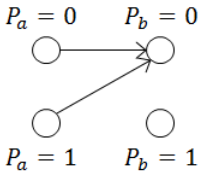

Figure 4. FP1 relation.

1) If Q contains a row,
   - L=delete a row from Q
   - else
   - L=delete a column from Q
2) Do maxPaint(L)
3) For all triggered lines L' in the above max-Paint operation
   - If L' does not exist in Q then add L' to Q

Figure 3 illustrates the benefits of ordering rows and columns when propagating. Consider that in Figure 3(a), we perform max-paint on rows $2, 4, 6$ and update pixels $(2, 3), (4, 3), (6, 3)$ in the grid. In the case of using RC-ordered as shown in Figure 3(b), we can perform one max-paint on column 3 to update this column. If no RC-ordered, the worst case, is as follows: steps than Figure approach using RC-ordering strategy.

3 Fully-Probing Deductions

For a puzzle gird G, let $G' = G \mid (P_a, \mathcal{C})$ denote a new grid $G' = G$ the same as G but additionally painting pixel P_a to color $\mathcal{C} \in \{\epsilon, 0, 1, \mathrm{u}\}$. A puzzle grid with some unsolved pixels is called an *incomplete grid*. An incomplete grid G is said *feasible*, in terms of the given clues, if we can solve the puzzle by painting all the unsolved pixels in G while leaving those solved pixels unchanged.

In [12], the authors demonstrates some Nonogram puzzle grids to which the propagation operation ends with partially-painted unsolved pixels. In this case, they suggested to make progress via a guess-and-check approach, called *probing*. For an incomplete feasible grid G and a pair of unsolved pixels P_a and P_b. If we perform the propagation by painting P_a to color $\mathcal{C} \in \{0, 1\}$, a sequence of max-painting on row/columns will be triggered. Suppose that after the propagation stops, the pixel b is changed from color u to color $\mathcal{D} \in \{0, 1\}$, we said that there is a *propagation relation* between a and b in terms of puzzle grid G, and denote it by

$$G \mid (P_a, \mathcal{C}) \longrightarrow G \mid (P_b, \mathcal{D}) \tag{2}$$

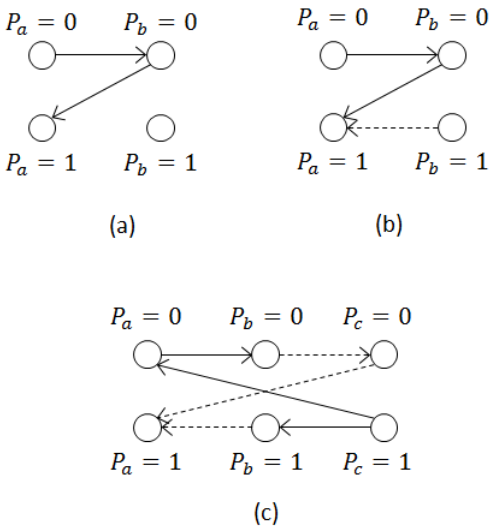

Figure 5. (a) A case that FP1 can not solve any pixel. (b) Use FP1 and FP2 to solve pixel P_a. (c) Use FP3 to solve pixel P_a.

For a pair of unsolved pixels P_a and P_b in the incomplete feasible grid G, the following property holds [12],

- FP1:

$$\begin{aligned} &(G \mid (P_a, 0) \longrightarrow G \mid (P_b, \mathcal{C})) \wedge \\ &(G \mid (P_a, 1) \longrightarrow G \mid (P_b, \mathcal{C})) \\ \Longrightarrow\ &G \mid (P_b, \mathcal{C}) \text{ is a feasible solution} \end{aligned}$$

For example, in Figure 4, let $\mathcal{C} = 0$, Property FP1 holds in the sense that, P_a's color must be either 0 or 1, both leading to P_b's color equals to 0.

Based on contrapositive logic $p \implies q \iff \neg q \implies \neg p$, we can write the *contrapositive propagation relation* as follows:

- FP2: Let $\mathcal{C}, \mathcal{D} \in \{0, 1\}$,

$$\begin{aligned} &(G \mid (P_a, \mathcal{C}) \longrightarrow G \mid (P_b, \mathcal{D})) \\ \iff\ &(G \mid (P_b, \neg\mathcal{D}) \longrightarrow G \mid (P_a, \neg\mathcal{C})) \end{aligned}$$

FP2 can help solve more pixels as demonstrated in Figure 4. In Figure 4 (a), we can not determine the color of pixel P_a or P_b since FP1 does not hold. However, as shown as in Figure 4 (b), if we incorporate the FP2 relation (the dashed edge), we can obtain that $P_a = 1$, since both $P_a = 0$ and $P_a = 1$ lead to $P_b = 1$.

Combining the FP1 and FP2 relations, we can obtain the *transitive propagation relation* that is denoted as follows:

$$G \mid (P_a, \mathcal{C}) \rightsquigarrow G \mid (P_b, \mathcal{D}) \tag{3}$$

FP3 extends FP1 based on the transitive propagation relations:

- FP3:

$$\begin{aligned}&(G \mid (P_a, 0) \rightsquigarrow G \mid (P_b, \mathcal{C})) \wedge \\ &(G \mid (P_a, 1) \rightsquigarrow G \mid (P_b, \mathcal{C})) \\ \Longrightarrow\ &G \mid (P_b, \mathcal{C}) \text{ is a feasible solution}\end{aligned}$$

4 Conclusion

The recently proposed approaches can efficiently solve many puzzles via the logical deduction based on 2-SAT formulas. In this paper, we propose a simplified form of the recurrence relation in the previous Nonogram algorithm. Our work simplifies the recurrence relation from 7 formulas down to 3 formulas.

References

[1] D Cohen P Jeavons M Gyssens “A unifie theory of structural tractability for constraint satisfaction problems” Journal of Computer System Sciences, vol. 74 no.5, 721–743, 2008.

[2] H.-H. Lin, D.-J. Sun, I.-C. Wu, and S.-J Yen “The 2011 TAAI Computer-Game Tournaments ” ICGA Journal vol. 34, no.1, pp. 51–54, 2011.

[3] H.-H. Lin, I.-C. Wu "An Efficient Approach to Solving the Minimum Su oku Problem ” ICGA Journal vol 34 no. 4, pp. 191–208, December 2011.

[4] T.-Y. Liu, I.-C. Wu, and D.-J. Sun, “Solving the Slitherlink Problem,” submitted to International Workshop for Computer Games (IWCG 2012) in November 2012.

[5] S Simpson “Nonogram Solver ” available at www.comp.lancs.ac.uk /~ss/nonogram/.

[6] D.-J. Sun, K.-C. Wu, I.-C. Wu, S.-J. Yen and K.-Y. Kao, “Nonogram Tournaments in TAAI 2011,” to appear in ICGA Journal, 2012.

[7] Wikipedia, “Nonogram”, at en.wikipedia.org/wiki/Nonogram.

[8] J Wolter “Effect of Line Solution Caching on Pbnsolve Run-times ” available at http://webpbn.com/survey/caching.html.

[9] J Wolter “The ‘pbnsolve’ Paint-by-Number Puzzle Solver ” available at http://webpbn.com/pbnsolve.html.

[10] I.-C. Wu, D.-J. Sun and S.-J. Yen, “HAPPYNURI Wins Nurikabe Tournament,” ICGA Journal, vol. 33 no. 4, pp. 236, December 2010.

[11] I.-C. Wu, D.-J Sun, L.-P Chen, K.-Y Chen, C.-H Kuo, H.-H Kang, and H.-H Lin, "An Efficient Approach to Solving Nonograms", IEEE Transactions on Computational Intelligence and AI in Games, VOL. 5, NO. 3, SEPTEMBER 2013

[12] K.-C. Wu ”TAAI2011 Nonogram Tournament Result,” available at http://kcwu.csie.org/ kcwu/nonogram/taai11/, 2012.