On Solving Simple Curved Nonograms

Maarten Löffler[1][0009−0001−9403−8856], Günter Rote[2][0000−0002−0351−5945], Soeren Terziadis[3][0000−0001−5161−3841], and Alexandra Weinberger[4][0000−0001−8553−6661]

[1] Utrecht University, Utrecht, The Netherlands, m.loffler@uu.nl
[2] Freie Universität Berlin, Germany, rote@inf.fu-berlin.de
[3] TU Eindhoven, Eindhoven, The Netherlands, s.d.terziadis@tue.nl
[4] FernUniversität in Hagen, Germany, alexandra.weinberger@fernuni-hagen.de

Abstract. Nonograms are a popular type of puzzle, where an arrangement of curves in the plane (in the classic version, a rectangular grid) is given together with a series of hints, indicating which cells of the subdivision are to be colored. The colored cells yield an image. Curved nonograms use a curve arrangement rather than a grid, leading to a closer approximation of an arbitrary solution image. While there is a considerable amount of previous work on the natural question of the hardness of solving a classic nonogram, research on curved nonograms has so far focused on their creation, which is already highly non-trivial. We address this gap by providing algorithmic and hardness results for curved nonograms of varying complexity.

Keywords: Nonogram · Arrangement · Puzzle · Dynamic Programming · Complexity

1 Introduction

Nonograms, also known as *Japanese puzzles*, *paint-by-numbers*, or *griddlers*, are a popular puzzle type where one is given an empty grid in which some grid cells are to be colored (filled); the remaining cells remain empty (unfilled). For every row and column, there is a *clue sequence* (sometimes called *description*) that constrains the set of colored grid cells in this row or column. The clue sequence specifies how many consecutive blocks of cells should be filled and how large these blocks are. Two filled blocks need to be separated by one or more unfilled cells. A solved nonogram typically results in a picture (see Figure 1).

Nonograms provide an accessible and contained environment for logical deduction. They have been used successfully to teach logical thinking [10,32], and have been shown to stimulate brain activity to prevent dementia [16].

Batenburg et al. [2] introduce the notion of a *simple* nonogram, which can be solved efficiently. A nonogram is *simple* when it can be solved by only looking at a single row or column at a time. More precisely, they consider a nonogram simple if it can be solved by repeatedly focusing a row or column, considering all possible solutions for it that are consistent with the fixed cells determined so far, and fixing all cells which have the same value in every possible solution.

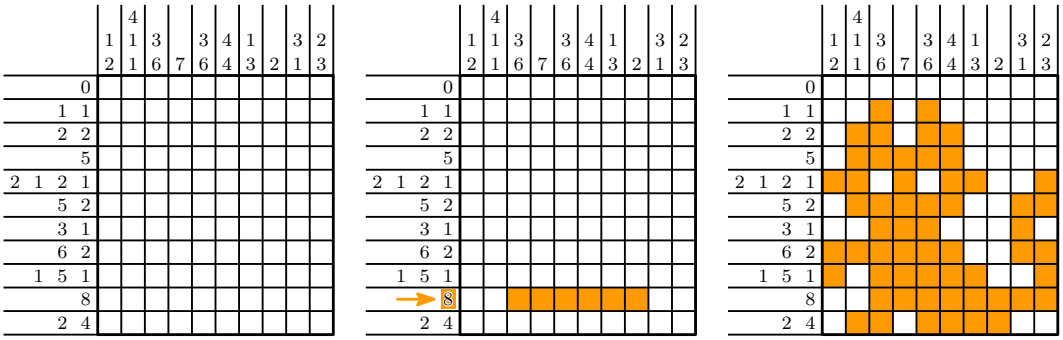


Fig. 1: (a) A classic nonogram puzzle. (b) An inference based on the highlighted clue. (c) The solved nonogram.

This procedure is called *settling a row/column* (or simply SETTLE) and will be considered in more detail in the preliminaries (Section 2). Note that since repeated application of settling a row or column is a deterministic process, the existence of multiple solutions for a nonogram immediately implies that it cannot be simple; however, the converse is not true, i.e., there are uniquely solvable nonograms that are not simple. In fact, Batenburg and Kosters introduce a whole hierarchy of complexity for nonograms, depending on the number of rows and columns which have to be considered simultaneously (by a specific solver) in order to definitively identify a cell whose status can be settled; puzzles with unique solutions can be found at all levels of this hierarchy.

Nonogram puzzles that appear in newspapers or similar platforms tend to be of this simple type [3], which contradicts to some extent the popular opinion that all interesting games and puzzles are NP-hard [11,29].

1.1 Solving Nonograms

A large amount of research on solving nonograms appears in software repositories, discussion forums, or on personal web pages, as collected in an online survey [31] of Jan Wolter. Besides, there has also been substantial academic interest in nonograms. The natural question is to come up with an algorithm to decide whether a given nonogram can be solved. A number of solvers using various strategies have been presented in the literature. These include heuristic approaches [21], DFS-based solving methods [15,33,23], genetic algorithms [24,6,22,1,9], line-by-line solving combined with probing (using low probability guesses to quickly achieve contradictions) [5], SAT solvers [20], integer linear programming [17], and a combination of heuristics and neural networks [8]. The performance of two general solving strategies (DFS and so called soft computing) has been experimentally compared on a small set of four nonogram instances [30].

The computational problem of deciding if a nonogram has a solution is NP-complete, as was first shown by Uada and Nagao [25]; see also [14,28]. This

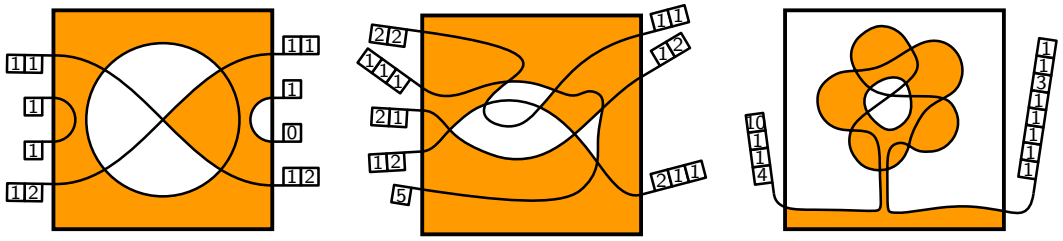


Fig. 2: Three types of curved nonograms of increasing complexity [26], shown with solutions. (a) *Basic* puzzles have no popular faces. (b) *Advanced* puzzles may have popular faces, but no self-intersections. (c) *Expert* puzzles have self-intersecting curves.

of course implies that computing this solution is also at least NP-hard. Uada and Nagao additionally prove that, given a nonogram and a solution, deciding if this solution is unique is also NP-hard, via a parsimonious reduction from three-dimensional matching.

In contrast, Batenburg and Kosters [3] gave a polynomial-time algorithm that, given a sequence of partially settled cells and a corresponding clue sequence, finds a cell that is either filled or unfilled in every possible solution, if such a cell exists. Their procedure can be used to either solve a given nonogram in polynomial time or decide that it is not simple.

1.2 Curved Nonograms

Van de Kerkhof et al. [26] introduced *curved* nonograms, a variant in which the puzzle is no longer played on a grid but on any arrangement of curves (an example is shown in Figure 2); see also [27]. For distinction, we will refer to the nonograms played on a grid as *classic nonograms*. In curved nonograms, the numbers of filled faces of the arrangement in the sequence of faces that appear along a side of a curve, are specified by a clue sequence (one on each side). Curved nonograms allow cells with more organic shapes than classic nonograms, and thus lead to clearer or more specific pictures. Van de Kerkhof et al. focus on heuristics to automatically generate such puzzles from a desired solution picture by extending curve segments to a complete curve arrangement.

Additionally, they define three different levels of complexity of curved nonograms — not in terms of how hard it is to *solve* a puzzle, but how hard it is to understand the rules (see Figure 2). It turns out that these difficulty levels nicely correspond with properties of the underlying curve arrangement as observed by De Nooijer et al. [13] (see [12] for the conference version). Specifically, *basic* curved nonograms are exactly the puzzles in which each clue sequence corresponds to a sequence of *distinct* faces. The analogy with clue sequences in classic nonograms is straightforward. In an *advanced* curved nonogram, a face may be incident to the same curve multiple times, but only on the same side,

and therefore a face can appear more than once in a sequence. If such a face is filled it is also counted multiple times when checking consistency with a clue sequence; in particular, it is no longer true that the sum of the numbers in a clue sequence is equal to the total number of filled faces incident to the curve. *Expert* curved nonograms may have clue sequences in which a single face is incident to the same curve on *both* sides (which corresponds to the presence of a self-intersecting curve in the arrangement).

Research on curved nonograms has so far focused on their production. Klute, Löffler and Nöllenburg [18] investigate the geometric problem of adding clue sequences to the ends of curves and provide polynomial-time algorithms for restricted cases and hardness results for the general problem. De Noojier et al. [13] aim to eliminate all faces with multiple incidences to the same curve (so-called *popular faces*) from a nonogram by adding one additional curve to the arrangement. The same goal was recently pursued by reconfiguring the curve arrangement through local crossing resolution [7].

1.3 Contribution

In this paper, we investigate for the first time the computational problem of *solving* curved nonograms. In particular, we investigate how the concept of *simple* nonograms translates to curved nonograms. After some preliminaries in Section 2, we present in Section 3 a dynamic program which leverages the nested structure of popular faces in advanced nonograms to check for a given sequence of faces along a curve, some of which are already filled or unfilled, if it can still be extended to a solution that is consistent with a given clue sequence. This implies a procedure solving simple advanced curved nonograms in $O(l^7)$ time, where l is the length of the longest clue sequence. This runtime can be improved to $O(l^6)$ by using an additional top-down phase of the dynamic program. In the case the nonogram is basic, the dynamic program coincides with a special case of the one presented by Batenburg and Kosters [3], showing that simple basic curved nonograms can be solved in the same way as simple classic nonograms. Then Section 4 shows that self-intersecting curves likely make curved nonograms significantly harder to solve, since even simple curved expert nonograms are at least as hard to solve as classic nonograms with a guaranteed unique solution. We close with some further research questions in Section 5.

This work will be presented at the 36th International Workshop on Combinatorial Algorithms (IWOCA 2025), in Bozeman, Montana, in July 2025. The version in the proceedings of this workshop [19] uses a different terminology.

2 Preliminaries

In this section we introduce the basic concepts and notation as well as the basic problems, which naturally arise in the context of solving nonograms.

2.1 Nonograms

Let $\mathcal{A}$ be a curve arrangement consisting of h curves $A_1, \ldots, A_h$ all contained in and starting and ending at a rectangle called the *frame*. Every piece of a curve A between two consecutive intersections (or the start or end of A) is a curve segment of A. A face of $\mathcal{A}$ (also called cell) is *popular* if two or more curve segments incident to the face belong to the same curve. Every cell initially has the value *unsettled* which we denote with `?`. If a value is assigned one of the two values *empty* (`0`) or *filled* (`1`) we say that the cell is *settled*. Here we follow the notation of Batenburg and Kosters [3].

We choose an arbitrarily orientation for each curve; accordingly, a face f incident to a curve segment s is said to be *on the left* or *on the right* side of s. Let $s_1, s_2, \ldots, s_k$ be the curve segments of a curve ℓ in $\mathcal{A}$. We call the list of faces $f_1, \ldots, f_k$, s.t. f_i is on the right (left) of s_i the *right (left) sequence* S^r_ℓ (S^l_ℓ) of ℓ. Popular faces can appear multiple times in the same sequence, and if ℓ is a self-intersecting curve, faces can appear in both sequences.

A progress descriptor for a sequence S is a string $\Psi^S = \psi_1\psi_2\ldots\psi_k \in \{\texttt{0}, \texttt{1}, \texttt{?}\}^k$, and it encodes the current state of knowledge about the faces in the sequence. If Ψ^S contains no `?`, then it is a *fix*. If the sequence in question is clear from context, we may omit the superscript. If for two progress descriptors Ψ and Ψ' of the same sequence S it holds that either $\psi_i = \texttt{?}$ or $\psi_i = \psi'_i$ for all i, we say that Ψ' *refines* Ψ.

A clue sequence $D = d_1, \ldots, d_t$ is a list of t numbers. One such number d_i will be called a *clue* of D. A fix Ψ of S is *consistent with* D if and only if Ψ contains exactly t maximal blocks of consecutive `1`s and the i-th block consists of exactly d_i `1`s. Since these blocks are maximal, consecutive blocks are separated by one or more `0`s. A progress descriptor Ψ of S is *consistent with* D if there exists a fix that is consistent with D and refines Ψ.

In a curved nonogram, a face can appear more than once along a curve. This leads to additional constraints in the form of equations $\psi_i = \psi_j$. We encode this by a sequence of letters $f_1 \ldots f_l$, like `abcdefdbgbh`, where repeated letters indicate positions that belong to the same face. For example, the 2nd, 8th, and 10th edges lie on a common face, marked `b`. We call this the *face pattern* of the sequence.

We will only consider progress descriptors Ψ that fulfill all equality constraints.

A *curved nonogram* C consists of a curve arrangement together with a set of clue sequences and progress descriptors (one for each sequence in C respectively). If all progress descriptors are fixes, we say the nonogram is solved and conversely *solving* a given nonogram means obtaining a fix for every progress descriptor that is consistent with its clue sequence. For any $i \leq j$ we write $i \,..\, j$ for the list of numbers between i and j (including both).

2.2 Settling and Nonogram Complexity

Given a progress descriptor Ψ, which is consistent with a clue sequence D, obtaining a progress descriptor Ψ' that refines Ψ and is still consistent with D

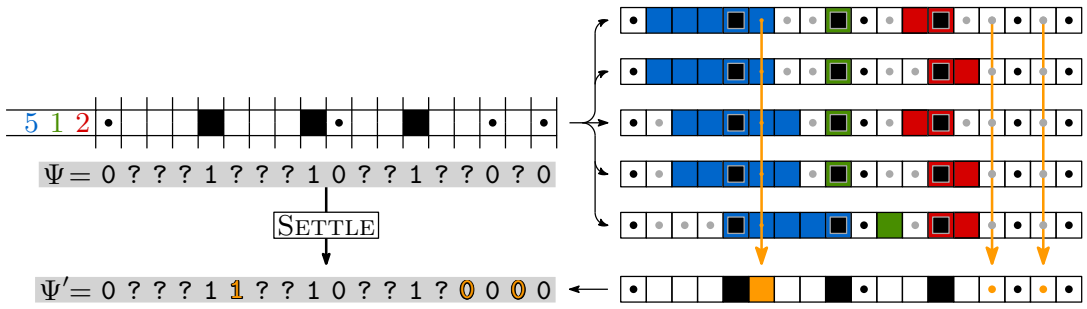


Fig. 3: An example of the SETTLE function. Filled cells are shown with a filled square, empty cells with a dot; all other cells are unsettled. For example, in a row with progress descriptor Ψ and clue sequence D = 5-1-2 (shown in the top left), there are five possible fixes of Ψ consistent with D, as shown in the top right. Cells already settled in Ψ are colored black, others gray. The blocks corresponding to the three parts of D are indicated in corresponding colors. One cell is filled and two are empty for all five fixes. This is indicated with yellow entries.

is called *making progress on* Ψ. The procedure SETTLE(Ψ, D) takes a progress descriptor and a clue sequence and (if possible) returns a progress descriptor Ψ', which refines Ψ and is consistent with D. It does so by settling any unsettled cells to be filled (or empty) if they have the same value in all possible fixes which refine Ψ and are consistent with D. This procedure is illustrated in Figure 3.

Note that there can be exponentially many such fixes. However, while SETTLE is defined via equality of the value of a cell over all possible fixes, an implementation of SETTLE does not necessarily need to enumerate all possible fixes to find such a cell. For example in a classic nonogram, the dynamic program of Batenburg and Kosters [3] finds such a cell in polynomial time or decides that no such cell exists. Applying SETTLE to all rows and columns of a nonogram until no progress can be made is called a FULLSETTLE.

If every progress descriptor of a nonogram has a fix consistent with its clue sequence it is *solvable* and correspondingly we will call a nonogram in which every progress descriptor is a fix the *solution* of the nonogram. If a nonogram is solvable via a FULLSETTLE it is called *simple*. We remark that this definition is in line with [3], whose algorithm can solve simple classic nonograms in polynomial time.

3 Solving Simple Advanced Curved Nonograms

In this section we present a dynamic program which, given a sequence S together with a progress descriptor Ψ and a clue sequence D decides in polynomial time if there exists a fix Ψ' consistent with D that refines Ψ. This is analogous to the existing dynamic program by Batenburg and Kosters for classic nonogram [3]. Readers familiar with their work will easily spot the parallels; however, the presence of popular faces requires the maintenance of an additional data structure.

The application of our algorithm to simple basic curved nonograms (i.e., those without popular faces) is discussed at the end of the section.

The property of advanced nonograms that is crucial for us is that the equality constraints are properly nested:

Observation 1 *Let i, j, k and l be four indices of letters in the face pattern of a sequence S belonging to a curve A in an advanced curved nonogram, such that $f_i = f_j \neq f_k = f_l$. W.l.o.g. assume $\min(i, j, k, l) = i$ and $k < l$. Then either (i) $j < k \wedge j < l$ or (ii) $j > k \wedge j > l$.*

Proof. We only have to exclude the order $i < k < j < l$. Assume w.l.o.g. that we are considering the left side of A. Let a and b be points on the i-th and j-th segment of A, respectively. Since these segments lie on a common face, we can connect a and b by a curve B in that face, on the left side of A. Let $A[a, b]$ be the subcurve of A between a and b. Then $C = A[a, b] \cup B$ is a Jordan curve (simple and closed). If $i < k < j < l$, C encloses the face on the left side of the k-th segment, but it does not enclose the face on the left side of the l-th segment. Hence, these faces cannot be identical, contrary to our assumption $f_k = f_l$, and therefore, the order $i < k < j < l$ is impossible. □

We mention that the nesting property of Observation 1 is the only property on which our algorithm relies. If a curve A has self-intersections but there are no faces that lead to a violation of the nesting property, our algorithm can be applied.

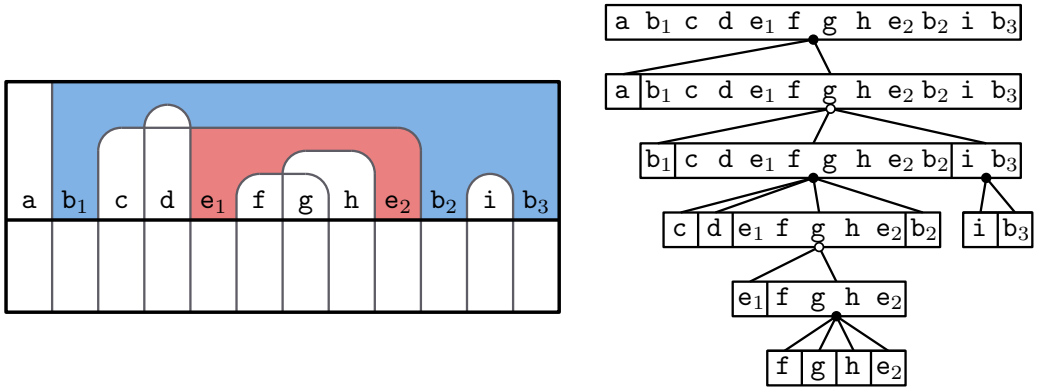
Theorem 1. *Consistency of a sequence S of length l with a clue sequence D with $\sum_{d \in D} d = k$ can be decided in time $O(k^3 l) = O(l^4)$.*

Proof. In a bottom-up phase of the dynamic program we try to match larger and larger intervals of the progress descriptor with larger and larger parts of the clue sequence. In a subsequent top-down phase we will discover which assignments are consistent with an overall solution.

We translate $D = d_1 d_2 \ldots d_t$ to a clue bitstring $= a_1 a_2 \ldots a_k$ of `0`s and `1`, by creating blocks of d_i `1`s for every $1 \leq i \leq t$ and concatenating them with one `0` between consecutive blocks. Additionally we artificially pad the list by an extra `0` at the beginning and at the end. This assumption implies that every row and column starts and ends with an empty cell. Every nonogram can obviously be padded with empty rows and columns to achieve this. For example, $D =$ `5-1-2` is translated to $D^{01} =$ `011111010110`, with the understanding that a `0` has the potential to stretch to an arbitrary larger number of unfilled cells. A progress descriptor is consistent with this clue bitstring if one can create two equal strings by replacing every `?` in the progress descriptor with either `0` or `1` and replacing any `0` in the clue sequence with one or more `0`. The following example shows this for $D =$ 5-1-2.

clue bitstring $D^{01} = \quad a_1 a_2 \ldots a_i \ldots a_k \quad = \texttt{011111010110}$

progress descriptor $\Psi = \psi_1 \psi_2 \psi_3 \ldots \psi_j \ldots \psi_{l-1} \psi_l = \texttt{0???1???10??1??0?0}$



(a) (b)

Fig. 4: (a) A schematic representation of a curve arrangement indicating the face incidences for top side of the horizontal line and (b) its hierarchical decomposition into subintervals. For clarity, multiple occurrences of the same face (such as $\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3$) are distinguished by indices. A white node denotes a decomposition of a complete group into brackets; a black node denotes decomposition of a bracket into complete groups. Black and white nodes occur in alternate levels of the tree.

In the finished nonogram, all `?`'s should be turned into `0`'s or `1`'s, subject to the requirement that the resulting sequence fits the progress descriptor. We want to know whether a particular `?` can be turned only into `0` or only into `1` in *all* possible solutions, because then this `?` can be fixed to this value. In other words we aim to implement the SETTLE procedure.

Recall that the finished solution must satisfy certain equations $\psi_i = \psi_j$ when two edges are incident to a common face, which we have encoded by a face pattern $f_1 \dots f_l$, like `abcdefdbgbh`, where repeated letters indicate that edges belong to the same face. According to Observation 1, these repeated occurrences are *nested*: The pattern $\dots x \dots y \dots x \dots y \dots$ cannot occur in the sequence.

We solve the following subproblems $Match_{j\,..\,j'}^{i\,..\,i'}$, for every $1 \le i \le i' \le k$ and for a certain set J of $2l-1$ selected intervals $j \,..\, j'$ with $1 \le j \le j' \le l$:

> Can the `?`'s in $\psi_j \,..\, \psi_{j'}$ be turned into `0`'s or `1`'s such that the resulting string is consistent with the clue bitstring $a_i \,..\, a_{i'}$?

The subproblem $Match_{j\,..\,j'}^{i\,..\,i'}$ results in a Boolean value *true* or *false*. Accordingly, we will say that a subproblem is consistent or inconsistent. See Figure 6 for an example.

The set J of intervals $j \,..\, j'$ of the curve that we consider is defined as follows.

Suppose that the cells $j_1 < j_2 < \dots < j_m$ are the cells belonging to some common face: $f_{j_1} = f_{j_2} = \dots = f_{j_m}$. We call the interval $j_1 \,..\, j_m$ a *complete group*, and we call the intervals $j_1 \,..\, j_p$, for $p = 1, \dots, m$ the *progressive sleuths*. The first progressive sleuth is the singleton interval $j_1 \,..\, j_1$. If a face occurs only once along the curve, at position j, then the singleton interval $j \,..\, j$ forms a complete group.

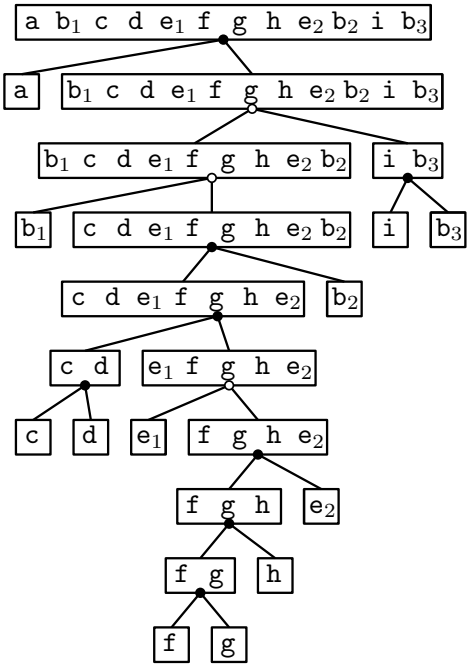


Fig. 5: The binary composition tree $\mathcal{T}$ corresponding to the tree of Figure 4b, in which nodes of higher degree have been replaced by sequences of binary nodes. This is a binary tree whose leaves are the singleton intervals.

An interval $j_p{+}1 \,..\, j_{p+1}$ between two successive occurrences of the same face, including the second occurrence but not the first, is called a *bracket*. A bracket consists of a nonempty sequence of complete groups, followed by an occurrence of the face of the enclosing group at position j_{p+1}. We consider also the whole interval $1 \,..\, l$ as a bracket although it lacks the final element of the enclosing group (see Figure 4 for an illustration).

We will build up the whole curve $1 \,..\, l$, starting from singleton intervals $j \,..\, j$. These can be seen as the leaves of a binary composition tree $\mathcal{T}$ (see Figure 5) This binary tree represents how we will combine certain pairs of consecutive intervals $j \,..\, j'$ and $j'+1 \,..\, j''$ into larger intervals $j \,..\, j''$. This is done as follows.

Every complete group is built up from left to right by successive addition of brackets:

$$[j_1 \,..\, j_p] \cup [j_p + 1 \,..\, j_{p+1}] = [j_1 \,..\, j_{p+1}]$$

Similarly, every bracket is built up from left to right by successive addition of the complete groups that make it up (plus the final cell of the enclosing group). In total, a set J of $2l - 1 = O(l)$ intervals $j \,..\, j''$ are considered. Each such interval with $j < j''$ – an internal node in $\mathcal{T}$ – is built in a unique way from two disjoint subintervals in J:

$$[j \,..\, j''] = [j \,..\, j'] \cup [j'+1 \,..\, j''] \tag{1}$$

See Figure 5 for an example.

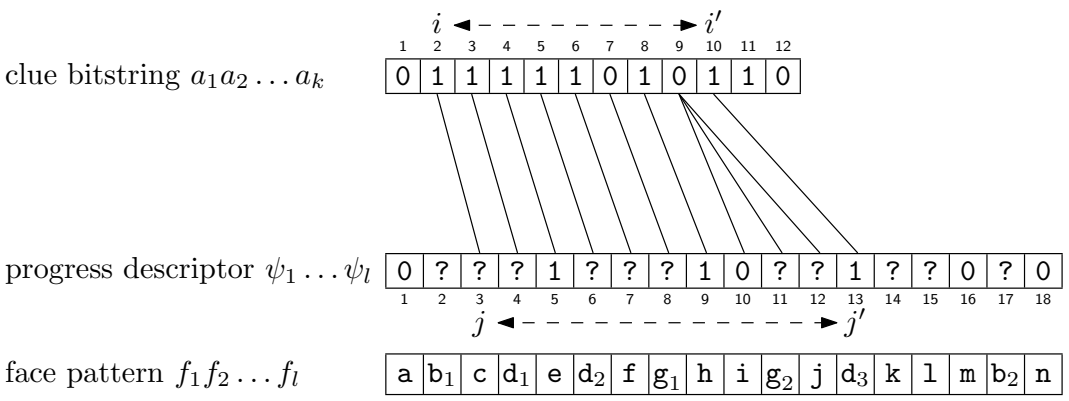


Fig. 6: Illustration of a subproblem $Match^{i..i'}_{j..j'} = Match^{2..10}_{3..13}$. A possible correspondence between $a_2 \dots a_{10}$ and $\psi_3 \dots \psi_{13}$ is indicated, showing that this subproblem is consistent.

The singleton subproblems of the form $Match_{j..j}$ are trivial to solve: For a clue bitstring of length 1, we have $Match^{i..i}_{j..j} \iff \psi_j = \texttt{?} \vee \psi_j = a_i$, while $Match^{i..i'}_{j..j}$ is trivially inconsistent for $i < i'$, since a single cell can never be consistent with a clue bitstring of length larger than 1.

Following the decomposition (1), each subproblem of the type $Match_{j..j''}$ with $j'' > j$ is associated to two families of smaller subproblems $Match_{j..j'}$ and $Match_{j'+1..j''}$, for some fixed j'. We solve these subproblems by the following recursion:

$$\begin{aligned} Match^{i..i''}_{j..j''} \iff & \bigvee_{i':i\le i'\le i''-1} \left(Match^{i..i'}_{j..j'} \wedge Match^{i'+1..i''}_{j'+1..j''} \wedge a_{i'} = a_{i''}\right) \\ & \vee \bigvee_{i':i\le i'\le i''} \left(a_{i'} = 0 \wedge Match^{i..i'}_{j..j'} \wedge Match^{i'..i''}_{j'+1..j''} \wedge a_{i'} = a_{i''}\right), \end{aligned} \tag{2}$$

where the final condition $a_{i'} = a_{i''}$ is present only in case of composing a progressive sleuth with a bracket. This extra condition ensures that occurrences of the same face have the same color `0` or `1`; when combining two complete groups, there are no shared faces that need to be considered, and the condition $a_{i'} = a_{i''}$ is omitted.

The first clause considers all possibilities of splitting the interval $i \,..\, i''$ into two disjoint parts $i \,..\, i'$ and $i'+1 \,..\, i''$. The second clause considers in addition the possibility that the two parts of the curve can use overlapping parts of the clue bitstring if the overlap is a `0`.

This completes the description of the bottom-up phase. The *target problem* $Match^{1..k}_{1..l}$ describes the original problem: consistency of the whole progress descriptor Ψ with the complete clue bitstring D^{01}.

In total, there are $O(k^2 l)$ subproblems, and each subproblem can be evaluated by trying $O(k)$ choices for i', for a total running time of $O(k^3 l)$. □

3.1 Making progress

Having solved the consistency problem, we immediately get a polynomial-time solution algorithm for making progress.

Proposition 1. *Given a single sequence S of length l and a clue sequence D with $\sum_{d\in D} d = k$ we can make progress or decide that no progress can be made in time $O(k^3l^2) = O(l^5)$.*

Proof. We tentatively set a `?` letter to `0` or `1` and check consistency again. If one of the options is inconsistent, then we know that `?` must be replaced by the other letter, thus making progress. This is repeated most l times, for each occurrence of `?`. □

However, we can solve this more efficiently and avoid the additional factor l by a top-down phase, in which we mark certain subproblems as *extensible*.

Theorem 2. *Given a single sequence S of length l and a clue sequence D that describes k ones we can make progress or decide that no progress can be made in time $O(k^3l) = O(l^4)$.*

Proof. We call a subproblem $Match_{j\,..\,j'}^{i\,..\,i'}$ *extensible* if it is consistent and in addition, some solution that fits the clue bitstring $a_i \,..\, a_{i'}$ can be extended to a complete solution by setting the remaining `?`'s outside the substring $b_j \,..\, b_{j'}$ appropriately.

We begin by marking the target problem $Match_{1\,..\,l}^{1\,..\,k}$, as extensible, assuming it is consistent. Then we use the recursion (2) in reverse. If $Match_{j\,..\,j''}^{i\,..\,i''}$ is extensible, then, if any of the parenthesized clauses on the right-hand side of (2) holds for some i', we mark the two corresponding subproblems $Match_{j\,..\,j'}^{i\,..\,i'}$ and $Match_{j'+1\,..\,j''}^{i'(+1)\,..\,i''}$ as extensible.

Finally we look at each unsettled position j with $b_j = \texttt{?}$, and we check for which clue bitstring positions i the problem $Match_{j\,..\,j}^{i\,..\,i}$ is extensible. If all extensible problems among these have $a_i = \texttt{0}$, we can conclude that b_j must be set to `0`, and settle an unsettled color in this way. Similarly, if all extensible problems have $a_i = \texttt{1}$, we can fix the unsettled value b_j to `1` at this position. □

Theorem 2 can be used to obtain the following corollary by the simple fact that there are only a linear number of rows and columns and cells in the nonogram. After applying the dynamic program once to every row and column, we must have made progress on at least one sequence, so there is at most an overhead of $O(l^2)$.

Corollary 1. *Simple advanced curved nonograms can be solved in time $O(l^6)$.*

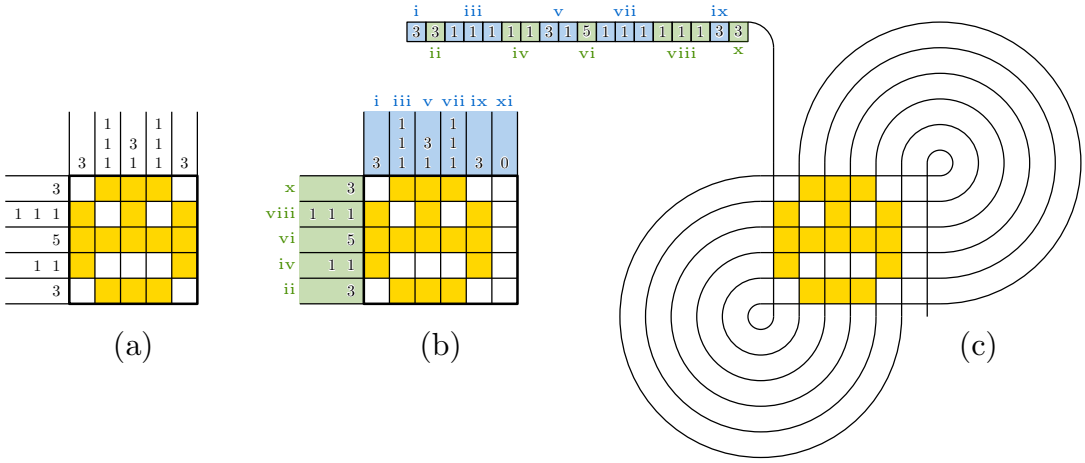


Fig. 7: (a) A classic nonogram with $w = 5$ columns and $h = 5$ rows. (b) In the reduction we pad a given nonogram to guarantee it has one more column than rows. The result here is a padded nonogram with $w = 6$ columns and $h = 5$ rows. The hints are annotated with the order in which they are collected into (c) a single self-intersecting *vital* curve that contains all grid lines of the classic nonogram and all clue sequences in one *vital* clue sequence.

3.2 Back to Basic Nonograms

When our algorithm is applied to a curve in a basic nonogram, there are no groups, and the whole sequence is just one bracket whose sequence of "complete groups" consists of singletons. The decomposition tree degenerates, and the algorithm simply grows the intervals $1 \,..\, i$ and $1 \,..\, j$ by adding one symbol at at time. Here our dynamic program reduces to the seminal algorithm of Batenburg and Kosters [3] (which is actually more general because it can deal with a specified *range* of lengths for each 1-block instead of a fixed length).

4 Solving Simple Expert Curved Nonograms

In this section we show that solving a simple curved expert nonogram is at least as hard as finding the solution to a not necessarily simple classic nonogram provided that the classic nonogram has a unique solution.

Unique Solution Nonogram (1-SN). *Given a classic nonogram N with the guarantee it has a unique solution, find the solution for the nonogram N.*

Note that testing if a classic nonogram has a solution is NP-hard in general, as shown by Ueda and Nagao [25]. This of course implies that finding such a solution is also NP-hard. Ueda and Nagao [25] also show that testing if a given nonogram has more than one solution, even if we are given a solution, is NP-hard.

However this does not directly imply that finding a solution for a classic nonogram is still NP-hard if we are guaranteed that it has a unique solution. By providing a reduction from 1-SN to Simple Curved Expert Nonogram

we still show that finding the solution to a simple curved expert nonogram is at least as hard as solving 1-SN.

4.1 High Level Overview

The high-level overview of our reduction is as follows. We first describe the construction of a curved expert nonogram C based on a given not-necessarily-simple classic nonogram N. If we are guaranteed that N has a unique solution, then C will equally have a unique solution, and additionally we will show that C is simple. Next we argue that the sequences along all but one curve can be trivially filled (using the SETTLE procedure) by simply filling all sequences whose clue sequence requires the entire sequence to be colored black. This will yield a partially filled curved expert nonogram, in which all cells that are not yet colored are part of the right sequence S^r_ℓ along a single curve ℓ. Moreover the progress descriptor Ψ of this sequence includes already filled chains of cells and there is a one-to-one correspondence between any chain of unsettled cells to a row or column of the input classic nonogram. Given any fix which refines Ψ and is consistent with the clue sequence of S^r_ℓ will fill in all remaining cells of C and immediately yields a solution for N.

Since C is simple, it can be solved with an application of FULLSETTLE. Therefore a polynomial time algorithm for FULLSETTLE on curved expert nonograms would imply that classic nonograms can be solved in polynomial time, if we are guaranteed that their solution is unique.

4.2 Constructing the curved expert nonogram

Consider a classic nonogram N with w columns and h rows, as in Figure 7(a). We will assume w.l.o.g. that we have $w = h + 1$; this can be achieved through appropriate padding, see Figure 7(b). Note that adding empty (or completely filled) rows or columns does not change the difficulty of the puzzle. All cells of N will also be contained in the curved expert nonogram C we created based on N. We will call these cells the *original cells*.

Now, conceptually, we will trace a single *vital* curve through all $w+1$ vertical and $h+1$ horizontal line segments that make up the grid of the puzzle (excluding the section that contains the clue sequences); refer to Figure 7(c). Doing this will concatenate the clue sequences from all rows and columns of the original nonogram into a single clue sequence; specifically, it will intersperse the clue sequences of the columns (from left to right) and the rows (from top to bottom). We will refer to the resulting clue sequence as the *vital* clue sequence.

However, this alters the difficulty of the puzzle, as the information which sections of the vital clue sequence belong to separate rows or columns is lost. To solve this, we again pad the original nonogram, but now with rows and columns, which we will force to be entirely colored in a solution of C as follows.

We let $k = 1 + \max\left(\lfloor \frac{w}{2} \rfloor, \lfloor \frac{h}{2} \rfloor\right)$; this value is chosen to ensure that $2k$ is more than either w or h. We first construct another padded $w + 2 + 2k$ by $h + 2 + 2k$ grid: the original grid with a single empty and k full rows added on all sides.

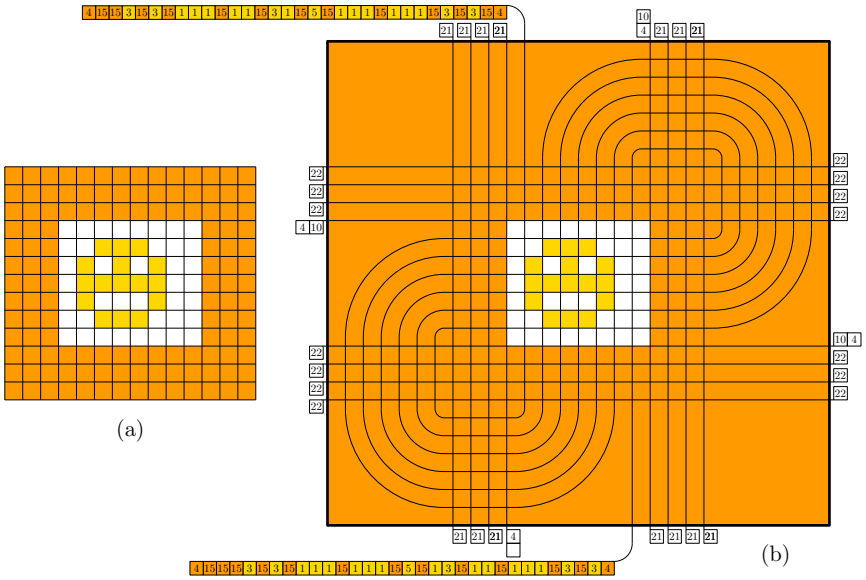


Fig. 8: (a) An even more padded version of the nonogram from Figure 7. (b) The final construction including $k = 3$ additional rows and columns of filled cells on all sides. Original filled cells are yellow; padding cells are orange. In the vital clue sequence, numbers are orange if they are at least $2k$ and yellow otherwise.

Refer to Figure 8(a). All filled/empty cells that are added by this procedure are called the filled/empty *padding cells*.

Then, we construct a curved nonogram C which consists of this grid surrounded by some additional potentially non-rectangular cells (which will be called *boundary cells*). In total, it consists of $4k + 5$ curves: $k + 1$ straight lines on each side of the input picture, plus 1 very long curve ℓ which contains all original grid lines. Refer to Figure 8(b). Note that by construction all clue sequences which consists of a single clue require their entire sequence to be filled. Settling all cells of these sequences to be filled also uniquely determines a fix for all sequences with clue sequences consisting of two clues and we state the following observation.

Observation 2 *Any sequence other than S^r_ℓ has a clue sequence of length one or two. Moreover all boundary and filled padding cells can trivially be settled to be filled and all empty padding cells can trivially be settled to be empty. Every unsettled cell is an original cell and contained in S^r_ℓ.*

Next we consider the vital clue sequence D. Note that we can partition D into $2(w+h+4)+1$ (possibly empty) parts, s.t., these parts alternatingly correspond to the clue sequence of a column or row of N and clues which require $4k - 1$ consecutive filled cells (with the exception of the first and last part, which require exactly $k + 1$ filled cells). We call the parts requiring $4k - 1$ cells *blockers*. Since the first and last cell in the vital sequence are filled, the first and last clue of the vital clue sequence are necessarily already fulfilled. Note two things. First there is a matching of already settled cells along the vital sequence and blockers, s.t.,

all blockers are fulfilled and second there are either $w + 2$ or $h + 2$ unsettled cells between two consecutive chains of $4k - 1$ already filled cells and therefore no blocker can be fulfilled in such a space. Therefore this matching is the only possible realization of the blockers and we know that every chain of unsettled original cells in C has to accommodate exactly the clues that the original row or column in N had to realize. With this we state the following observation.

Observation 3 *If we restrict the solution of C to its original cells, we obtain exactly the solution of N.*

Lemma 1. *If N has a unique solution the constructed curved expert nonogram C also has only a single solution. Moreover C is simple.*

Proof. The first part of the lemma follows as a direct consequence of Observations 2 and 3. Over all solutions of C, any padding and boundary cell in C can have exactly one value (filled or unfilled) and any original cell can have at most as many values as the corresponding cell in N over all valid solutions of N. If this solution of N is unique, every original cell can have only one such value.

The second part of the lemma statement is a consequence of Observation 2. Since the value of all cells, which are not part of the vital sequence can trivially be settled, if we would apply the Settle procedure to the vital sequence, we would settle all remaining cells, since they can have only one value in a solution (because this solution is unique). □

4.3 Correctness

We are now ready to prove the main theorem.

Theorem 3. *Solving Simple Curved Expert Nonogram is (a) at least as hard as 1-SN and (b) in NP.*

Proof. To prove statement (a) it suffices to show that we can construct C based on a given nonogram N in polynomial time and given a solution of Simple Curved Expert Nonogram, i.e., a filled version of C, we can construct a solution to 1-SN for the instance N in polynomial time. The first part is immediate as the construction as described in Section 4.2 which yields C based on a given N adds only a polynomially many cells to N and the curves can be obtained by connecting at most a polynomial number of grid lines. Since N has a unique solution by definition of 1-SN, it follows from Lemma 1.

The second part, i.e., constructing a solution for N based on a given solution for C follows from Observation 3. By simply settling all cells in N according to the value of the original cells in C, we obtain the solution.

To prove statement (b) it suffices to observe that, given a solution for Simple Curved Expert Nonogram, we can enumerate all polynomially many sequences, and check in polynomial time if their fix is consistent with their clue sequence. This concludes the proof. □

As previously mentioned we only show with Theorem 3 that Simple Curved Expert Nonogram is at least as hard as 1-SN. While it seems reasonable to expect 1-SN to be NP-hard (equivalent to the generalized problem, i.e., finding a solution to any classic nonogram), the exact complexity of 1-SN remains an open question.

We note that the construction of our hardness proof produces a simple curve arrangement, i.e., there are no three curves intersect in the same point, no two curves touch without crossing, and no curves locally overlap in more than a single point.

5 Conclusions

We have shown that the concept of *simple* nonograms extends to curved nonograms to some extent. In general, simple curved nonograms are not necessarily easy to solve: even the problem of testing for progress on a single clue sequence is already as hard as solving a classic nonogram under the assumption that it has a unique solution and while we suspect this problem to be NP-hard, the complexity of solving a simple curved expert nonogram remains an open question. However, for the restricted classes of *basic* and *advanced* curved nonograms, we show that simple puzzles can be solved in polynomial time. It would be of interest how other measures of difficulty like the ones proposed by Batenburg and Kosters [4] extend to curved nonograms.

References

1. H. Alkhraisat and H. Rashaideh. Dynamic inertia weight particle swarm optimization for solving nonogram puzzles. *Int. J. Comput. Sci. Appl.*, 7(10):277–280, 2016. doi:10.14569/IJACSA.2016.071037.
2. K. Batenburg, S. Henstra, W. Kosters, and W. Palenstijn. Constructing simple nonograms of varying difficulty. *Pure Mathematics and Applications (PU. M. A.)*, 20:1–15, 2009.
3. K. Batenburg and W. Kosters. Solving nonograms by combining relaxations. *Pattern Recognit.*, 42(8):1672–1683, 2009. doi:10.1016/j.patcog.2008.12.003.
4. K. J. Batenburg and W. A. Kosters. On the difficulty of nonograms. *J. Int. Comput. Games Assoc.*, 35(4):195–205, 2012. doi:10.3233/ICG-2012-35402.
5. D. Berend, D. Pomeranz, R. Rabani, and B. Raziel. Nonograms: Combinatorial questions and algorithms. *Discret. Appl. Math.*, 169:30–42, 2014. doi:10.1016/J.DAM.2014.01.004.
6. A. Bobko and T. Grzywacz. Solving nonograms using genetic algorithms. In *17th International Conference Computational Problems of Electrical Engineering (CPEE)*, pages 1–4, 2016. doi:10.1109/CPEE.2016.7738765.
7. F. Brunck, H.-C. Chang, M. Löffler, T. Ophelders, and L. Schlipf. Reconfiguring popular faces. In M. Buchin, A. Lubiw, A. de Mesmay, S. Schleimer, and F. Brunck, editors, *Computation and Reconfiguration in Low-Dimensional Topological Spaces (Dagstuhl Seminar 22062)*, volume 12, issue 2, pages 24–34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022. doi:10.4230/DagRep.12.2.17.

8. J. M. Buades Rubio, A. Jaume-i-Capó, D. López González, and G. Moyà Alcover. Solving nonograms using neural networks. *Entertain. Comput.*, 50:100652, 2024. doi:10.1016/j.entcom.2024.100652.
9. Y. Chen and S. Lin. A fast nonogram solver that won the TAAI 2017 and ICGA 2018 tournaments. *J. Int. Comput. Games Assoc.*, 41(1):2–14, 2019. doi:10.3233/ICG-190097.
10. J. Cigas and W.-J. Hsin. Teaching proofs and algorithms in discrete mathematics with online visual logic puzzles. *Journal on Educational Resources in Computing*, 5(2), 2005. doi:10.1145/1141904.1141906.
11. G. Cormode. The hardness of the Lemmings game, or "Oh no, more NP-completeness proofs". In *Proceedings of 3rd International Conference on Fun with Algorithms*, pages 65–76, 2004.
12. P. de Nooijer, S. Terziadis, A. Weinberger, Z. Masárová, T. Mchedlidze, M. Löffler, and G. Rote. Removing popular faces in curve arrangements. In *31st International Symposium on Graph Drawing and Network Visualization (GD)*, volume 14466 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2023. doi:10.1007/978-3-031-49275-4_2.
13. P. de Nooijer, S. Terziadis, A. Weinberger, Z. Masárová, T. Mchedlidze, M. Löffler, and G. Rote. Removing popular faces in curve arrangements. *J. Graph Algorithms Appl.*, 28(2):47–82, 2024. doi:10.7155/jgaa.v28i2.2988.
14. H. J. Hoogeboom, W. Kosters, J. van Rijn, and J. Vis. Acyclic constraint logic and games. *J. Int. Comput. Games Assoc.*, 37(1):3–16, 2014. doi:10.3233/ICG-2014-37102.
15. M.-Q. Jing, C.-H. Yu, H.-L. Lee, and L.-H. Chen. Solving japanese puzzles with logical rules and depth first search algorithm. In *International Conference on Machine Learning and Cybernetics*, pages 2962–2967, 2009. doi:10.1109/ICMLC.2009.5212614.
16. V. Kasinathan, A. Mandhana, A. Mustapha, and N. S. A. Adam. How logical and mathematical games treat patients and adults suffering from dementia. *Int. J. Curr. Res.*, 12(18):9–13, 2020. doi:10.31782/IJCRR.2020.121829.
17. K. A. Khan. Solving nonograms using integer programming without coloring. *IEEE Trans. Games*, 14(1):56–63, 2020. doi:10.1109/TG.2020.3036687.
18. F. Klute, M. Löffler, and M. Nöllenburg. Labeling nonograms: Boundary labeling for curve arrangements. *Comput. Geom.: Theory Appl.*, 98:101791, 2021. doi:10.1016/j.comgeo.2021.101791.
19. M. Löffler, G. Rote, S. Terziadis, and A. Weinberger. On solving simple curved nonograms. In H. Fernau and B. Zhu, editors, *36th International Workshop on Combinatorial Algorithms (IWOCA 2025), Bozeman, Montana, July 2025*, Lecture Notes in Computer Science. Springer-Verlag, 2025. to appear. arXiv:2505.01554.
20. A. Metodi, M. Codish, and P. J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res.*, 46:303–341, 2013. doi:10.1613/jair.3809.
21. S. Salcedo-Sanz, E. G. Ortiz-Garcia, A. M. Pérez-Bellido, A. Portilla-Figueras, and X. Yao. Solving japanese puzzles with heuristics. In *IEEE Symposium on Computational Intelligence and Games*, pages 224–231. IEEE, 2007. doi:10.1109/CIG.2007.368102.
22. R. Soto, B. Crawford, C. Galleguillos, and E. Olguín. Solving nonogram using genetic algorithms. In *11th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–4, 2016. doi:10.1109/CISTI.2016.7521597.

23. D. Stefani, A. Aribowo, K. V. I. Saputra, and S. Lukas. Solving pixel puzzle using rule-based techniques and best first search. In *International Conference on Engineering and Technology Development (ICETD)*, 2012.
24. J. Tsai. Solving Japanese nonograms by Taguchi-based genetic algorithm. *Appl. Intell.*, 37(3):405–419, 2012. doi:10.1007/S10489-011-0335-7.
25. N. Ueda and T. Nagao. NP-completeness results for NONOGRAM via parsimonious reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, 1996. CiteSeerX 10.1.1.57.5277, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5277.
26. M. van de Kerkhof, T. de Jong, R. Parment, M. Löffler, A. Vaxman, and M. van Kreveld. Design and automated generation of Japanese picture puzzles. In *40th Annual Conference of the European Association for Computer Graphics*, 2019. doi:10.1111/cgf.13642.
27. M. J. van Kreveld. On nonogram and graph planarity puzzle generation. In *30th Canadian Conference on Computational Geometry (CCCG)*, pages 326–327, 2018. URL: http://www.cs.umanitoba.ca/~cccg2018/papers/invited_paper_3.pdf.
28. J. van Rijn. Playing games: The complexity of Klondike, Mahjong, nonograms and animal chess. Master's thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2012.
29. G. Viglietta. Gaming is a hard job, but someone has to do it! *Theory Comput. Syst.*, 54:595–621, 2014. doi:10.1007/s00224-013-9497-5.
30. J. Więckowski and A. Shekhovtsov. Algorithms effectiveness comparison in solving nonogram boards. *Procedia Comput. Sci.*, 192:1885–1893, 2021. doi:10.1016/j.procs.2021.08.194.
31. J. Wolter. Survey of paint-by-number puzzle solvers. https://webpbn.com/survey/. [Accessed 2025-02-22].
32. Y. Yang, D. Zhang, T. Ji, L. Li, and Y. He. Designing educational games based on intangible cultural heritage for rural children: A case study on "Logic Huayao". In *Advances in Human Factors in Wearable Technologies and Game Design (AHFE)*, pages 378–389, 2019. doi:10.1007/978-3-319-94619-1_38.
33. C. Yu, H. Lee, and L. Chen. An efficient algorithm for solving nonograms. *Appl. Intell.*, 35(1):18–31, 2011. doi:10.1007/S10489-009-0200-0.

A Python problem for the settle algorithm

```python
# indices start at 0 (following Python conventions) and not at 1 (as in the paper)

from collections import defaultdict

def decomposition_tree(f, show=False):
    """construct the binary decomposition tree.
    f is a the "face pattern", a string (or list of numbers or other objects)
    with one entry for each segment of the curve.
    Equal letters indicate segments belonging to the same face.

    Tree has three types of nodes, see Figure 5:
    * "Singleton", a leaf.
    * "Separate", with two children (white node).
    * "Progressive", with two children (black node): progressive sleuth + bracket
    "Separate" and "Progressive" store a triplet (i,j,k) of indices:
    f[i..k] (f[i:k+1] in Python notation) is the current range,
    and it is split after f[j].  """
    assert all(c!=d for c,d in zip(f,f[1:])) # Consecutive letters must be distinct.
    where = defaultdict(list)
    for i,c in enumerate(f):
        where[c].append(i) # store in which positions each letter appears

    # The tree is built by the recursive procedure "decompose". This is
    # not described in the paper.
    def decompose(i,j):
        """Either f[i]=f[j]=c and i is the first occurrence of c, or i=0,
        or the symbol f[i-1] occurs again after f[i..j] but not in f[i..j]."""
        c = f[j] # pick pieces from the end
        first = where[c][0]
        if first<i: # illegal pattern
            d = f[i-1]
            raise ValueError(c,first,j, d,i-1,f.index(d,i))
        if i==j:
            return ("Singleton",i)
        if first>i:
            return("Separate", (i,first-1,j),
                   decompose(i,first-1),
                   decompose(first,j)) # complete group
        # else: # first==i
        where[c].pop()
        prev=where[c][-1]
        return("Progressive", (i,prev,j),
               # progressive sleuth
                   decompose(i,prev),
               # plus bracket
                   ("Separate", (prev+1,j-1,j),
                      decompose(prev+1,j-1),
                      ("Singleton",j)))
```

```
    try:
        Tree =  decompose(0,len(f)-1)
    except ValueError as err:
        c,i1,i2, d,j1,j2 = err.args
        print(f"Error: Sequence {f} has interleaving faces.")
        print(f"f[{i1}]=f[{i2}]={c}, f[{j1}]=f[{j2}]={d}")
        assert f[i1]==f[i2]==c and f[j1]==f[j2]==d
        assert c!=d
        assert i1<j1<i2<j2
        return
    if show:
        print("decomposition tree for",repr(f)+":")
        print_tree(Tree,f)
    return Tree

def print_tree(t,f,level=0):
    typ = t[0]
    if typ == "Singleton":
        print("  "*level,*t, f[t[1]])
    else:
        ranges = t[1]
        print("  "*level,typ,ranges,f[ranges[0]:ranges[-1]+1])
        for sub in t[2:]:
            print_tree(sub,f,level+1)

def settle(progress_descriptor, clue_sequence, face_pattern, show_tree=False):
    """progress_descriptor is a string of '0', '1', and '?'.
    It is assumed that it contains the added artificial '0' at the
    beginning and at the end.

    clue_sequence is a sequence of positive integers (clues)

    face_pattern gives the sequence of incident faces along the curve segments.
    face_pattern has the same length as progress_descriptor.
    """
    f = face_pattern
    clue_bitstring = [0]
    for n in clue_sequence:
        clue_bitstring += [1]*n+[0]
    a = ''.join(str(c) for c in clue_bitstring) # convert to 0-1 string
    k = len(a)
    l = len(face_pattern)
    assert l==len(progress_descriptor)
    psi = progress_descriptor
    print(f"{a=} {k=}\n{f=} {l=}\nprogress_descriptor {psi=}")
    T = decomposition_tree(face_pattern, show=show_tree)

    Match = defaultdict(bool) # defaults to False
    # For convenience, we use dictionaries instead of arrays.

```

```
    def solve_bottom_up(t):
        if t[0]=="Singleton":
            j = t[1]
            for i in range(k):
                Match[i,i, j,j] = psi[j]=="?" or psi[j]==a[i]
        else:
            typ,split, sub1,sub2 = t
            solve_bottom_up(sub1) # visit subproblems first
            solve_bottom_up(sub2)

            lenient = typ=="Separate" # (strict for typ=="Progressive")
            j1,j2,j3 = split
            for i1 in range(k):
                for i3 in range(i1,k): # recursion (2) of the paper
                    Match[i1,i3, j1,j3] = (
                        any(
                            Match[i1,i2,   j1,j2] and
                            Match[i2+1,i3, j2+1,j3] and
                            (lenient or a[i2]==a[i3])
                               for i2 in range(i1,i3))
                        or
                        any(a[i2]=="0" and
                            Match[i1,i2, j1,j2] and
                            Match[i2,i3, j2+1,j3] and
                            (lenient or a[i2]==a[i3])
                               for i2 in range(i1,i3+1))
                        )
    def solve_top_down(t):
        if t[0]=="Singleton":
            j = t[1]
            if psi[j]=="?":
                for i in range(k):
                    if Extensible[i,i,j,j]:
                        Possible_letters[j,a[i]] = True
        else:
            typ,split, sub1,sub2 = t
            lenient = typ=="Separate" # need not check extra equality
            j1,j2,j3 = split
            for i1 in range(k):
                for i3 in range(i1,k):
                    if Extensible[i1,i3, j1,j3]:
                        for i2 in range(i1,i3):
                            if (Match[i1,i2, j1,j2] and
                              Match[i2+1,i3, j2+1,j3] and
                              (lenient or a[i2]==a[i3])):
                                Extensible[i1,i2,   j1,j2] = True
                                Extensible[i2+1,i3, j2+1,j3] = True
                        for i2 in range(i1,i3+1):
                            if (a[i2]=="0" and
                              Match[i1,i2, j1,j2] and
```

```
                                Match[i2,i3, j2+1,j3] and
                                (lenient or a[i2]==a[i3])):
                                  Extensible[i1,i2, j1,j2] = True
                                  Extensible[i2,i3, j2+1,j3] = True
            solve_top_down(sub1)
            solve_top_down(sub2)

    solve_bottom_up(T)
    result = Match[0,k-1, 0,l-1]
    assert result

    Extensible = defaultdict(bool)
    Extensible[0,k-1, 0,l-1] = True
    Possible_letters = defaultdict(bool)

    solve_top_down(T)

    Settlestring = list(psi)
    for i,ps in enumerate(psi):
        if ps=="?":
            if not Possible_letters[i,"0"]:
                Settlestring[i]="1"
            if not Possible_letters[i,"1"]:
                Settlestring[i]="0"
    return "".join(Settlestring)

####### A FEW TEST CASES #######

print("Example from the paper, Fig.4/5")
decomposition_tree("abcdefghebib", show=True)

print("\nBad example: not nested, should raise an error.")
decomposition_tree("ababcdefghebib", show=True)

print("\nExample from the paper, Fig.3 (simple nonogram)")
Psi = "0???1???10??1??0?0"
se = settle( Psi, (5,1,2), list(range(len(Psi)))) # f is trivial (basic nonogram)
print(f"settled:                '{se}'\n")
assert se=="0???11??10??1?0000" # as claimed in Fig.3

print("Example from the paper, Fig.6 (advanced nonogram)")
print("settled:", settle( Psi, (5,1,2), "abcdedfghigjdklmbn", show_tree = True))
```