DOI: 10.1111/cgf.13642
EUROGRAPHICS 2019 / P. Alliez and F. Pellacini
(Guest Editors)
Volume 38 (2019), Number 2

Design and Automated Generation of Japanese Picture Puzzles

Mees van de Kerkhof[1], Tim de Jong[2], Raphael Parment[3], Maarten Löffler[†1], Amir Vaxman[1], and Marc van Kreveld[‡1]

[1]Dept. of Information and Computing Sciences, Utrecht University, the Netherlands
[2]Milvum, the Netherlands
[3]Macq, Belgium

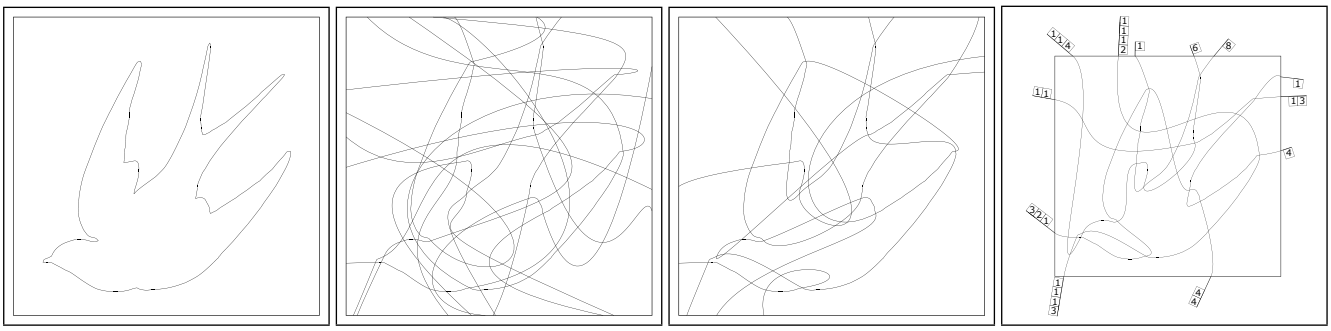


Figure 1: *Progression from an input image to a puzzle*

Abstract

We introduce the generalized nonogram, an extension of the well-known nonogram or Japanese picture puzzle. It is not based on a regular square grid but on a subdivision (arrangement) with differently shaped cells, bounded by straight lines or curves. To generate a good, clear puzzle from a filled line drawing, the arrangement that is formed for the puzzle must meet a number of criteria. Some of these relate to the puzzle and some to the geometry. We give an overview of these criteria and show that a puzzle can be generated by an optimization method like simulated annealing. Experimentally, we analyze the convergence of the method and the remaining penalty score on several input pictures along with various other design options.

CCS Concepts

• ***Applied computing*** → *Computer games;* • ***Human-centered computing*** → *Visualization systems and tools;*

1. Introduction

An application where design and computation come together is drawing puzzle generation. In a drawing puzzle, the puzzler may need to draw outlines, fill in areas, and use colors. Examples are connect-the-dots puzzles, picture mazes (where the solution path is a picture), and nonograms, also known as Japanese picture puzzles. Nonograms are grid-based puzzles where clues are given for each row and each column of the grid; *clues* tell which grid cells should be filled to retrieve the hidden picture. This picture is usually black-and-white pixel art, but color versions are also common. The puzzle aspect comes from the fact that we do not get information on any cell explicitly, only about the whole row and the whole column in which a cell lies. Algorithms to solve nonograms exist, but the problem is generally NP-hard. The puzzle instances given in puzzle booklets and on websites are usually much simpler and can be solved by applying simple reasoning.

In this paper we introduce new types of nonograms that generalize the basic type. We show that puzzles can be constructed based on any set of lines and even of curves. For these new types of generalized nonograms we review the design choices and let these inspire an automated method to generate a nonogram from a drawing. For nonograms based on curves, the curves replace the grid lines of basic nonograms and form an *arrangement* of cells with varying shape and size. Since rows and columns no longer exist, we need a new way to give clues indicating which cells should be filled by the

† Supported by the Netherlands Organisation for Scientific Research on grant no. 614.001.504.
‡ Supported by the Netherlands Organisation for Scientific Research on grant no. 612.001.651.

puzzler. Our idea to realize this is simple and effective. The new types of nonograms allow for new reasoning steps that do not exist in the grid-based nonogram.

We outline a global process that starts with a drawing of one or more filled shapes (called *picture*) and ends with a puzzle whose solution is again that drawing. The curves in our nonograms use the boundary curves of the input picture and extend them to the boundary of the puzzle (called *frame*), where the clues can be placed. How to place these curve extensions so that a clear curve arrangement is formed is the main algorithmic focus of this paper. Here we essentially sample the space of curves with certain restrictions so that problems like tiny cells and small intersection angles are avoided. The arrangement of curves becomes the puzzle once the clues are determined and placed.

1.1. Related work

The automated (procedural) generation of graphical content has a long history and many applications [Pru86, PL12, Sti80]. Early applications in graphics concern digital mountain landscapes, later extended with trees, rivers clouds, and leaves, for example. Procedural generation of buildings, road networks, and cities are the main examples for man-made structures. The main techniques used are L-systems and shape grammars, while agent-based systems have also been employed (see [HMVDVI13] and the references therein). Within AI (computational intelligence), the automated generation of levels of puzzles and games [Ash10, MK07, LH14, NMPL15, STN16, TP11, WKDAS12] and even of games themselves [Bro11] received considerable research attention. The most common methods are evolutionary or rule-based [STN16].

Puzzles are a type of one-player game where dexterity is less important and reasoning is more important. Drawing puzzles are puzzles where the correct solution reveals a drawing of a picture that is not visible at the start. Recently new types of connect-the-dots puzzles were introduced [LKvK*14]. Their design and automated generation from drawings uses concepts from polyline simplification. Our focus is on nonograms, also known as Picross, Pic-a-Pix, Hanjie, and Griddlers. The scientific study of nonograms usually concentrates on the algorithmic complexity of solving them [BK09, BPRR14, MA09, Tsa12, WSC*13]. Nonograms need a coarse pixel image which can be derived from smooth vector art by rasterization. Beyond the classic research by Bresenham, more recent research focuses on generating pixel art [GDA*13, IVK13] or representing polygonal shapes onto the grid [BKvK*16] with bounded error. An interesting link exists with computer tomography, where one also gets information on cross-sections and the objective is to reconstruct the model [BK04]. There is other research involving various types of picture puzzles scattered throughout the scientific literature [ATG12, JSC13, OU09, OSL*07, YLK08].

1.2. Outline

In Section 2 we describe the new types of nonograms and the new way to give clues. We describe the design aspects that influence the quality of a puzzle, in particular the ones that concern the geometry. These design aspects allow us to develop an automated nonogram puzzle generator that takes a picture and outputs a puzzle whose solution is that picture; this is described in Section 3. The focus is on producing suitable curve arrangements. In Section 4 we discuss experiments and their outcomes that test certain design choices. In Section 5 we conclude by giving directions for future research. The supplementary material is a puzzle booklet with various generalized nonogram puzzles whose curve arrangements were automatically generated from drawings.

2. Nonogram types and design choices

In this section we begin by describing the basic black-and-white and colored versions of nonograms. Then we generalize to puzzles based on lines with more than two orientations, and then we generalize further to curves. We discuss both the puzzle possibilities and design considerations for obtaining good generalized nonograms.

2.1. Nonogram types

In a basic nonogram, each row and each column has a clue that gives an abstract description of which cells are to be filled. Suppose a row is eight cells long. Then a clue 1 3 specifies that in the row, one cell should be filled, and later in that row another three cells in sequence. Between the one and the three filled cells, there is at least one non-filled cell. Also, before the one filled cell and after the three filled cells, there are zero or more non-filled cells. In binary, where 1 corresponds to a filled cell, 00101110 and 10011100 are both valid filling choices for the row of cells corresponding to the clue. A nonogram puzzle consists of a grid with no cells filled, see Fig. 2. A solution (or filling) is correct if for every row and column, the filled cells are an option for the given clue. There are clue sets that do not have any solution; there are also clue sets that have more than one solution. Obviously these do not give good puzzles. In the colored version, the numbers in the clues have a color, and this color should be used when filling cells. Two numbers of different colors give filled cell sequences that can be directly adjacent, unlike in the black-and-white type.

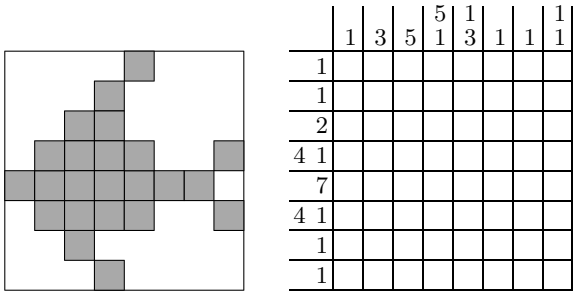


Figure 2: *Basic nonogram (right) of the pixel image shown left.*

The first new type of nonogram we present is the *tangram nonogram*, so named because many tangram puzzle assignments can become tangram nonogram puzzles. They contain lines that are horizontal, vertical, or have slope -1 or $+1$. The idea is to give a clue *for all cells that are incident to the line, for each side*. Each line has two sides, and also a sequence of cells incident to the line on that side (see Fig. 3 for the cells incident to the top side of the bold line). This sequence of cells is treated like a row or column in a

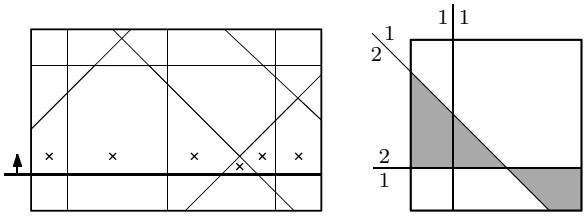


Figure 3: *Sequence of six cells incident to the indicated side of a line, marked with a cross. Solved very small tangram nonogram.*

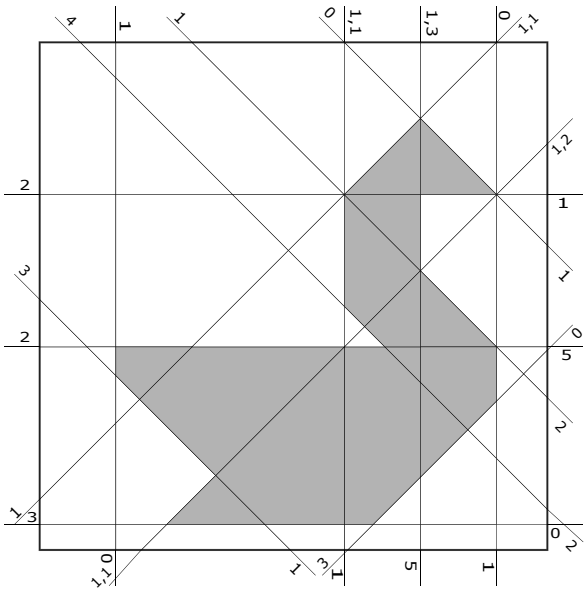


Figure 4: *Tangram nonogram with the solution shown.*

basic nonogram. Fig. 3 shows a simple puzzle to the right, with its solution. Fig. 4 shows a larger one.

In tangram nonograms we may decide to allow triple and quadruple intersections of lines (e.g., the tip of the bill in Fig. 4). The puzzle rules should specify what incidence of a line and cell means, for example, only edge-incident cells are considered incident to a line. A possible visual artifact in tangram nonograms are tiny triangular faces appearing between three lines. We can try to enlarge them by shifting lines slightly, but this may change the final picture and cause other cells to become tiny.

The second new type of nonogram is the *curved nonogram*. Like lines, curves also have two sides for which we can provide the clues. We can take splines as curves, like cubic Bézier splines. Triple-intersections are much easier to avoid by moving a control point. We may get self-intersections in a single curve. For the puzzle validity this is not a problem, because sides of curves do not change at intersections. However, for the puzzler it may be conceptually harder to solve a puzzle. Each curve starts and ends on the frame, and all curves together form an arrangement. If a curve bounds the same filled cell twice (or even more often), it will be represented in the clue both times.

2.2. Design choices for generalized nonograms

To solve a generalized nonogram puzzle as a puzzler, it is necessary to be able to trace a curve easily and see how it continues at every intersection. Furthermore, it must be easy to see which cells are incident to a side of a curve. This puts constraints on the arrangements and when they are considered suitable as a puzzle.

The following geometric criteria are used to evaluate nonogram puzzles on their geometry:

- Intersection angle: the angle of intersection of two curves should not be too small. If the angle is small, it is difficult to see whether the curves intersect or are tangent but do not intersect.
- Intersection distance: two intersections along a curve should be sufficiently separated. Otherwise, it is difficult to see whether the cell incident to the curve in between is really incident to the curve.
- Cell size: cells should not be too small. They would be hard to color, and in extreme cases it may be hard to see that there is a cell between the curves. Note that an arrangement of curves can have its vertices far apart but still have cells of arbitrarily small area. Conversely, we also do not want cells to be too big. Big cells are tedious to color, are less aesthetically pleasing and they make it easier to see the solution image of the puzzle before completing it.
- Cell shape: we wish to avoid oddly shaped cells because they are harder to color and less aesthetic in general.
- Curve length: puzzle curves that are long compared to the puzzle size are often incident to many cells, therefore more difficult to use the clues on, and also more tedious to trace by eye.

Besides these geometric criteria, we require that the puzzle is uniquely solvable and we wish to know its difficulty for a human solver. Furthermore, the clues need to be placed without overlap in a clear manner.

In our generation method we will use an optimization approach for the five geometric criteria listed above. We discuss the solvability and level of difficulty later; these are not taken into consideration during the main part of the generation. The placement of clues is a type of *boundary labeling*, a topic that recently received considerable attention in the graph drawing and computational geometry communities [BKPS10, BKSW07, BS05, BHKN09, KNR*16]. Each clue can be placed at either (or both) curve ends. We will not discuss clue placement any further; this is an independent task and our current implementation does it in a basic manner only.

We list further considerations and possibilities for generalized nonograms.

- We can use any outer shape of the puzzle, like a circle instead of a rectangle.
- We can base nonograms on triangular, pentagonal or other tilings. Hexagonal tilings do not work well because they are not composed of lines or curves that run from frame side to frame side.
- We may decide to allow or disallow self-intersections of curves.
- It is often not necessary to give clues to the sides of all curves. Many puzzles are still uniquely solvable if we leave out some clues.

- It is not necessary that each curve continues to the frame. A curve may stop in the middle at a T-junction. It is still clear what a clue means.
- A puzzle can have closed curves, like circles. A circle would not intersect the frame so we cannot place a clue outside the frame. Some puzzles with closed curves are uniquely solvable even without the clues of the closed curves.
- Open curves can end at one or two T-junctions in the middle and one or zero frame sides. If both curve ends are in the middle, a clue cannot be placed outside the frame.

As an example, the (hand-designed) puzzle of Fig. 5 is a valid, uniquely solvable generalized nonogram. Solving it will show the new ways of reasoning when solving generalized nonograms. For example, two clues can give information about more than one cell, which is not the case in basic nonograms. Furthermore, a curve that bounds the same cell more than once gives new reasoning steps as well.

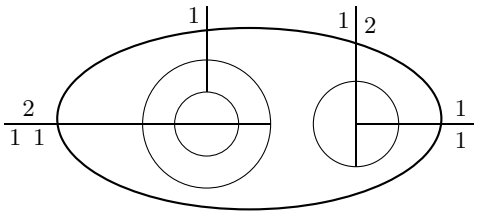


Figure 5: *Generalized nonogram that has closed curves, T-junctions, and an omitted clue (besides the absent clues of the closed curves).*

3. Automated generation

In this section we describe the computational techniques used to generate generalized nonograms from a picture. We briefly describe a simple method for tangram nonograms and then treat curved nonograms more extensively.

To generate a tangram nonogram from a picture bounded by segments in the four major orientations, we extend every segment to a line that reaches the frame in both directions. We get an arrangement with convex cells that have between 3 and 8 vertices. Note that intersection distance and cell size are the only two geometric criteria we need to consider. To avoid small distances and small cells we take the positions of the lines as variables, allowing them to shift but not rotate. We do not allow the combinatorial structure of the arrangement to change, because that may change the solution picture too much. The *width of a cell* in an arrangement is the smallest distance between any two parallel lines (from the space of all lines) such that the cell lies in between. A lower bound of δ on the width of all cells immediately implies a lower bound of δ on the distance between any two intersections and a lower bound of δ^2 on the area of any cell in tangram nonograms. We can express the width of a cell in the variables of the lines that bound the cell, and this expression turns out to be linear. Hence, we can maximize the width of the smallest cell in the arrangement by linear programming (after introducing an extra variable for the unknown, minimum required width of all cells, which we maximize). The other criteria like intersection angle, cell shape, and curve length do not play a role in tangram nonograms. Three examples of generated tangram nonograms are shown in Fig. 6. We can observe that the distance between parallel lines is often the same. This is caused by the objective of maximizing the minimum width of cells.

To generate a curved nonogram from a picture (globally; details follow soon), we extract the bounding curves and break them at G^1-discontinuities, yielding what we call *picture curves*. At every break we extend both ending curves in a G^1-continuous way to the frame; we call these curves *connection curves*. We may also connect two picture curve ends to each other with a so-called *bridge curve*. We ensure that no closed loops are made when creating a bridge curve. Fig. 7 illustrates the process. There is one bridge curve: between the tail ends. The challenge is how to choose the connection and bridge curves so that the desirable criteria are met. This can be done by designing penalty functions for violations and using an optimization approach like simulated annealing to decide how the connection and bridge curves are placed. We describe this next.

3.1. Penalties and fitness

To avoid intersection angles that are too small, we use a threshold angle α_t below which a penalty is given. When the intersection angle is smaller than α_t, the penalty increases linearly in how much smaller the angle is, from 0 up to α_t. The penalties are summed over all intersections. When two picture curves meet at an angle in $[0,\alpha_t)$ or in $(\pi-\alpha_t,\pi)$ then we cannot avoid incurring a penalty, because we can manipulate only the connection curves and bridge curves.

To avoid closeness of intersections in the curve arrangement, we use a threshold distance d_t below which a penalty is given. When two intersections are closer than d_t, they incur a penalty that increases linearly in how much closer the intersections are, from 0 up to d_t. The penalties are summed over all pairs of intersections that are adjacent on a curve. If any two ends of picture curves are closer than d_t, then we cannot avoid incurring a penalty.

For cell size we use the area of each cell, and again use a threshold s_t below which a penalty is given. We let this penalty scale quadratically since cell size is a very important factor in nonogram quality. Even a single tiny cell that can be missed will ruin the puzzle. So we want an extra large penalty on extremely small cells that dominates the penalties for cells that are only slightly too small. There is also a maximum size threshold b_t for cells, above which a penalty is given. This penalty scales linearly in the amount the cell is larger than b_t. Penalties are summed over the cells.

For cell shape we use a measure based on the detour factor of its boundary. The cost of a cell is the maximum, over any two points on its boundary, of the ratio of the (shorter of the two) boundary distance and the crow-flight Euclidean distance. A threshold h_t is subtracted, only positive remaining costs are taken and squared, and then summed over all cells to get the penalty for cell shape.

Finally, for curve length we use a maximum curve length ℓ_t that is still allowed without penalty; note that we use the full curve between its two ends on the frame. We let the penalty increase linearly

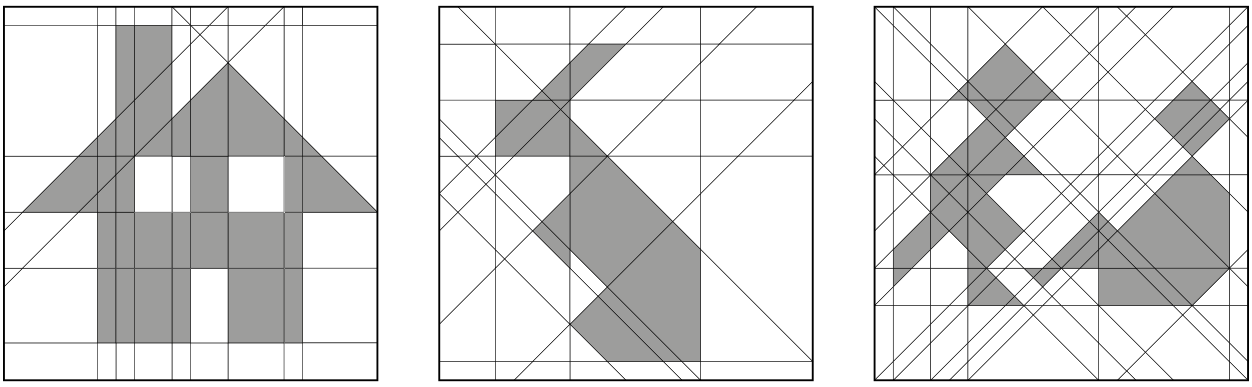


Figure 6: *Three tangram nonograms optimized by shifting lines, solved, without the clues shown.*

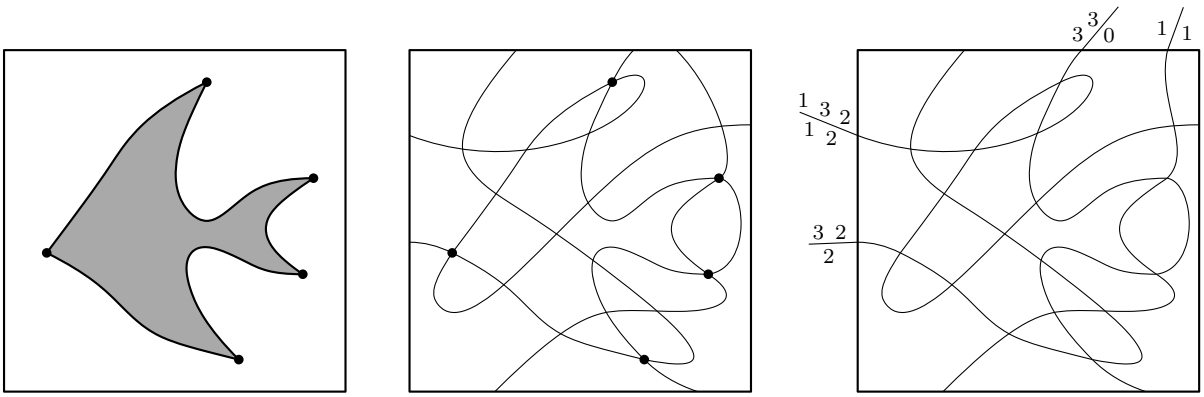


Figure 7: *Construction of a curved nonogram from a picture comparable to Fig. 2. Left, picture curves are made from the picture boundaries between G^1-discontinuities. Middle, picture curves are extended using connection curves and bridge curves. Right, clues are generated and placed at extensions outside the frame.*

for longer curves. A curve of length $\ell > \ell_t$ has penalty $(\ell - \ell_t)/\ell_t$. We add the penalties of the curves.

The penalties for the six criteria are combined using a weighted linear combination. Weights are set by trial and error. The result is a fitness function that is used in simulated annealing.

$$\text{fitness} = w_1 \cdot A + w_2 \cdot D + w_3 \cdot S + w_4 \cdot B + w_5 \cdot H + w_6 \cdot L$$

where A, D, S, B, L, and H are the total intersection angle penalty, intersection distance penalty, small cell size penalty, big cell size penalty, cell shape penalty, and curve length penalty, respectively. Values of the weights and thresholds are specified later; see Table 1.

In the implementation of the penalties, we approximate each Bézier curve by a polygonal line with constant length edges; the length is set to one percent of the longest side length of the frame. Intersection points are always added to these polygonal approximations. The penalties are computed exactly on these approximations; for the cell shape we determine the detour factor for every pair of vertices in the polygonal approximation of a cell. Therefore, this is a fairly expensive penalty to compute.

3.2. Simulated annealing

To set up a simulated annealing (SA) based algorithm we preprocess the input picture in a few steps, shown in Fig. 7. We assume the picture consists of one or more filled regions, each of which is bounded by one or more linked cubic Bézier curves. We break these linked Bézier curves at all G^1-discontinuities, creating a set of linked G^1-continuous cubic Bézier curves, the picture curves. Each such curve has two ends that must be extended to the frame which we do with a new, single cubic Bézier curve—the connection curve—in a G^1-continuous manner. The first control point of a connection curve must coincide with the picture curve end for continuity, the second control point must be aligned to achieve G^1-continuity, the third control point can be placed freely, and the fourth control point must be somewhere on the frame. Hence, a single connection curve has four degrees of freedom.

Besides connecting a picture curve end to the frame, we can also connect it to another picture curve end with a bridge curve. Bridge curves reduce the total number of curves in the puzzle by 1 and the total number of clues by 2. One can expect a less complex arrangement, and furthermore, clues will more often be a sequence of multiple values, yielding more interesting puzzles. On the other hand,

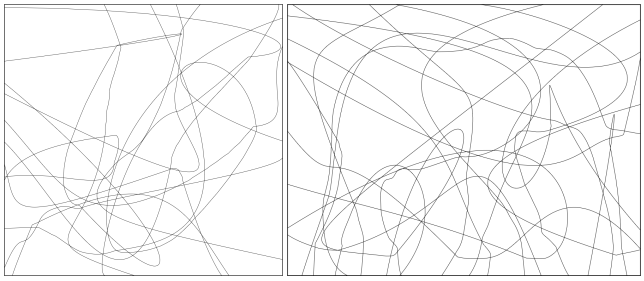


Figure 8: *Two examples of initial curve arrangements for the pictures Swallow and Elephant.*

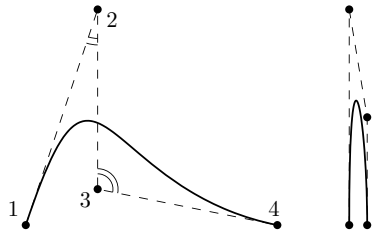


Figure 9: *Angles at the second and third vertices of the control polygon, and cusp-like feature when such an angle is small.*

connecting picture curves by bridge curves yields longer curves, and the penalty on curve length will ensure that curves do not get too long. If we connect two picture curves together with a cubic Bézier curve, we have very limited flexibility because the first and fourth control points are fixed and the second and third control points must be aligned to get G^1-continuity, leading to two degrees of freedom. We will ensure that no loops of curves are created since their clues would have to be placed inside the frame, which is not aesthetic and leaving out these clues may mean the puzzle is no longer uniquely solvable.

3.2.1. Initialization

We start with the picture curves inside the frame and represent them by a topological structure. We generate a G^1-continuous connection curve randomly for each picture curve end to connect it to the frame. Then we insert it into a doubly-connected edge list structure, forming an arrangement. The generated connection curve is not accepted when it passes extremely close to an existing intersection of the arrangement or violates any of our other criteria for curves listed in the next paragraph. The reason for aborting curves is two-fold. Most importantly, a curve which violates these criteria can cause numerical instabilities, challenging the robustness of our implementation. Secondly, it leads to features which a clear puzzle should avoid; the idea however is that simulated annealing will take care of such situations anyway so this reason is less important. When a curve is not accepted, we generate a new one and try again. After the initialization we have a nonogram without bridge curves and probably a bad score. Fig. 8 shows two examples.

3.2.2. Simulated annealing iterations and termination

Each iteration in the SA algorithm works as follows.

(1) We choose a random curve from the set of all connection and bridge curves, and then a second one. The probability that a curve is chosen is proportional to the penalty it incurs (specified in more detail later). If both chosen curves are connection curves, then we try to replace them by a single new bridge curve, unless a loop is formed. If not both are connection curves, we take only the first chosen curve. If it is a bridge curve, then we try to replace it by two connection curves. If it is a connection curve, then we try to replace it with a different connection curve.

(2) The replacement curve(s) is (are) generated G^1-continuous but otherwise random. If the replacement curve goes outside the frame, we restrict it to the part up to the first intersection with the frame. If the replacement curve is (i) extremely close to an existing intersection we abort, and also if (ii) the Bézier curve self-intersects, (iii) the Bézier curve produces an intersection very close to the frame, (iv) the Bézier curve produces an intersection with a very small angle, or (v) the Bézier curve makes a sharp turn close to one of its intersections. When aborting we generate a new random replacement curve by continuing at (2). If we abort too often we restore the situation and continue at (1).

(3) In all other cases we have a viable replacement. We determine the fitness before and after the operation and accept it or not in the usual, probabilistic SA manner depending on the temperature. In the implementation we remove one or two curves from the arrangement (its topological structure); then we insert one or two curves. Then we reduce the temperature by multiplying it with a cooling rate factor < 1 and continue at (1) if the temperature is still > 1. When a temperature of 1 is reached the algorithm keeps iterating with a temperature of 1 until it has converged. We say it has converged when it does 100 consecutive iterations without any improvement.

3.2.3. Further algorithmic details

The generation of a random connection or bridge curve works as follows. A connection curve from a curve end to the frame has its first control point set at the curve end. Its second control point is chosen on a line to ensure G^1 continuity with the curve it is extending. The distance to the first control point is at least *frame-min-side-length*$/12$ and at most 10 times this minimum (*frame-min-side-length* is the length of the smallest side of the frame); in this range it is chosen uniformly at random. The third control point is chosen randomly inside the whole frame. The fourth and last control point is chosen randomly on the frame. Furthermore, the angle between the edges of the control polygon of the cubic Bézier curve (at the second and third control point) must be at least 8 degrees, otherwise the Bézier curve has a point of high curvature which appears as a G^1-discontinuity, see Fig. 9. Bridge curves are generated in the same manner, but with only two random choices. Note that the second control point of a connection curve and the middle two control points of a bridge curve can be outside the frame.

The probability that a connection or bridge curve is chosen depends linearly on the penalty it incurs. This penalty is the total

Table 1: *Parameters of the algorithm for all inputs and all runs.*

Parameter	Value
Angle penalty weight w_1	1
Distance penalty weight w_2	2
Small cell penalty weight w_3	0.085
Big cell penalty weight w_4	0.00005
Cell shape penalty weight w_5	10
Curve length penalty weight w_6	75
Angle threshold α_t	20
Distance threshold d_t	$0.2 \cdot$ *frame-min-side-length*
Small cell threshold s_t	$0.001 \cdot$ *frame-area*
Big cell threshold b_t	$0.04 \cdot$ *frame-area*
Cell shape threshold h_t	3.5
Curve length threshold ℓ_t	$0.75 \cdot$ *frame-min-side-length*
Starting temperature SA	153
Cooling rate factor SA	0.99
Number of aborts until restart	50

penalty that relates to all intersection angles on it, all intersection distances in which it is involved, all cell sizes and shapes of adjacent cells, and the curve length. So the weight (penalty) is given by the fitness function, using only the vertices on it, the cells incident to it, and the curve length.

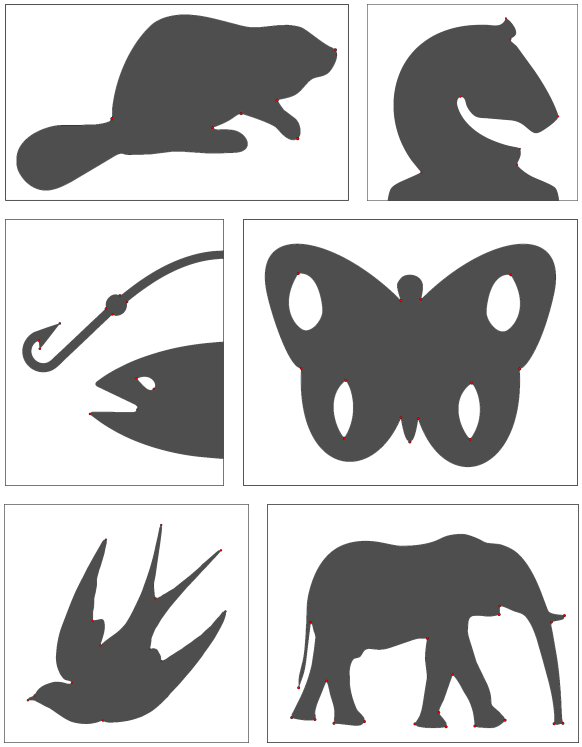
When we have a replacement, incurring its own penalty, we choose the replacement in one of two cases: (1) the penalty is reduced, or (2) if the penalty is not reduced, with a probability $P = e^{-\Delta/T}$, where Δ is the relative increase in penalty and T is the current temperature. For the relative increase, the penalty before the replacement is normalized to 100 and the penalty after the replacement is recomputed using this normalization. The difference of these values yields Δ.

4. Experiments and assessment

The main objective of our implementation is to determine if we can generate clear puzzles that satisfy the criteria given. In particular, we wish to know how stable SA is over different runs, how fast it converges, and how the different criteria are reduced. Furthermore, we wish to know how many bridge curves are used and whether the generated puzzles are uniquely solvable. We also want to examine the difference between allowing and disallowing our curves to self-intersect. Self-intersecting curves make the puzzle harder to solve for the puzzler, so we want to examine if it is possible to generate good curved nonograms where no curves self-intersect. For these experiments, whenever a randomly generated curve would cause a self-intersection, the curve is aborted. We are also interested in the efficiency of automatically generating puzzles, but since our code is not optimized these results are less meaningful.

We use six input pictures inside a frame and scale the input so that the largest dimension of the frame is 1000. We use one setting for all parameters and thresholds on which the algorithm depends, given in Table 1.

In the first experiment we examine convergence of SA. We plot the development of the score (penalty) function over the iterations



Name	No. picture curves	No. open curve ends
Beaver	6	12
Knight	8	14
Bait	13	22
Butterfly	13	26
Swallow	13	26
Elephant	20	40

Figure 10: *The six input pictures and their basic specifications.*

Table 2: *Score average, standard deviation of score, average number of bridge curves, and standard deviation of the number of bridge curves for the successful runs on the input, where self-intersections are allowed.*

Input	Score		Bridges		# runs
	Avg.	St.dev.	Avg.	St.dev.	
Beaver	580	332	2.18	0.75	28
Knight	276	159	2.33	0.98	30
Bait	30,006	25,126	5.10	0.76	29
Butterfly	4,398	5,524	6.00	1.06	30
Swallow	3,964	3,274	6.35	0.97	31
Elephant	21,771	17,213	7.96	1.00	25

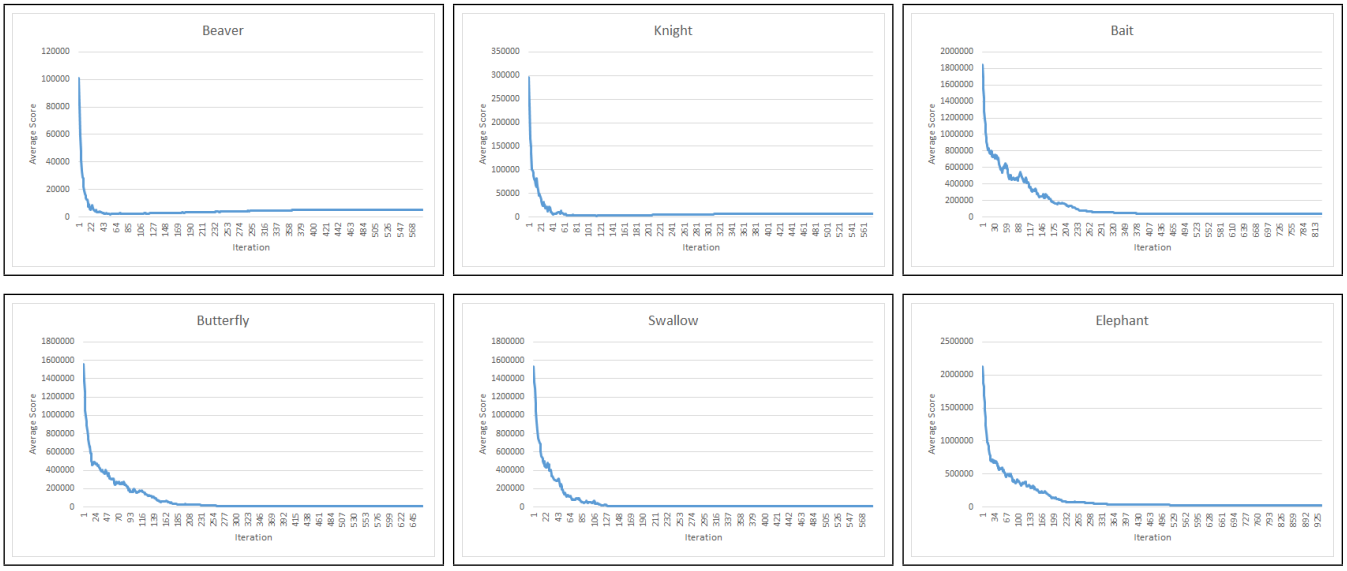


Figure 11: *Convergence plots of SA. Horizontal the iteration number, vertical the score averaged over all successful runs.*

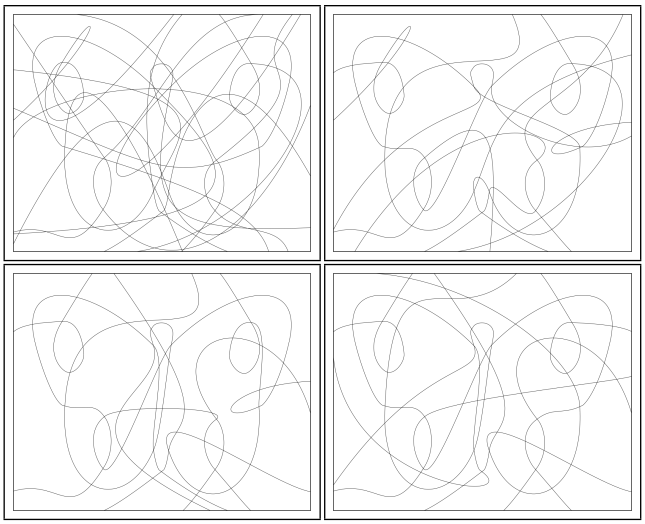


Figure 12: *Top-to-bottom and left-to-right: The arrangement of curves initially, after 100 iterations, after 200 iterations, and after convergence (after 371 iterations) of SA.*

in Fig. 11. Since the method is randomized, we make many different runs for input and plot the average score of the finished runs. We started 32 runs for each input image where self-intersections are allowed and 32 runs where they are not allowed, but since initialization can fail if we have to abort curves too many times not every attempt runs to completion. For the final arrangements we show the average score and standard deviation after SA in Table 2 and Table 3. We also show the average number of bridge curves and their standard deviation per input in this table. Visually, we can see the effects of SA from the initial situation, after 100 iterations, after 200 iterations, and at the end in Fig. 12.

Table 3: *Score average, standard deviation of score, average number of bridge curves, and standard deviation of the number of bridge curves for the successful runs on the input, where self-intersections are not allowed.*

Input	Score		Bridges		# runs
	Avg.	St.dev.	Avg.	St.dev.	
Beaver	473	186	1.4	0.49	15
Knight	245	122	1.7	0.64	27
Bait	34,804	27,153	4.27	0.86	11
Butterfly	N/A	N/A	N/A	N/A	0
Swallow	2,814	2,200	5.45	0.80	20
Elephant	19,956	19,115	7	1.05	18

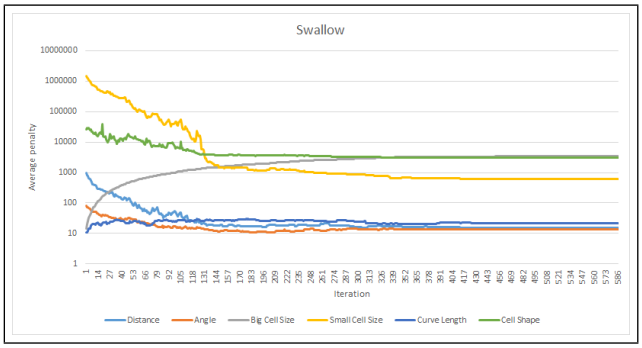


Figure 13: *The six penalties shown separately and their convergence averaged over all runs for Swallow.*

We also study the break-down of the score over the constituting criteria. The six penalties are shown together in Fig. 13 for the picture Swallow, taking the weights into account.

We observe that for all six inputs, the SA algorithm helps consid-

Table 4: *Efficiency in time and number of iterations; CPU time in seconds averaged over the number of successful runs; number of iterations averaged over the number of successful runs. Shown are the average and standard deviation. The upper half shows values with self-intersections allowed; in the bottom half they are not allowed.*

Input	Time		Iterations		#
	Avg.	St.dev.	Avg.	St.dev.	runs
Beaver	1,694	300	364.14	72.81	28
Knight	1,866	359	345.20	87.22	30
Bait	5,291	1,393	416.83	123.54	29
Butterfly	3,091	854	310.03	108.09	30
Swallow	4,900	1,218	290.23	104.84	31
Elephant	8,484	2,631	490.68	186.93	25
Beaver	2,497	265	449.00	50.94	15
Knight	2,409	458	408.89	71.65	27
Bait	7,505	2,066	568.18	174.43	11
Butterfly	N/A	N/A	N/A	N/A	0
Swallow	5,755	1,263	335.70	93.93	20
Elephant	10,487	2,285	577.61	133.50	18

erably to lower the (penalty) score. It appears that the algorithm has nearly converged after 100–300 iterations depending on the picture, but not fully yet. The resulting scores depend weakly on the number of input curves. Some pictures with not so many input curves have complex areas, which leads to higher scores, in particular for "bait". We also note that the standard deviation of the close to 30 runs is not low: different runs give different final scores, suggesting many local minima with different scores in the optimization problem. This is not a surprise, given the nature of the optimization.

Disallowing self-intersections reduces the number of runs that finish, and the successful runs have a lower score on average (except for Bait), but this is not significant. The resulting arrangements also have a lower number of bridge curves than the arrangements where self-intersections are allowed. We note that Butterfly will always have a curve with self-intersections because one wing hole has only one vertex on it, causing a self-intersecting curve immediately. This explains why there are no successful runs for this picture when no self-intersections are allowed. In general, excluding self-intersecting curves decreases the search space. Hence, not having a higher average score on the successful runs than when self-intersections are allowed is an interesting result. It may suggest that self-intersecting curves are associated with negative nonogram features, and that explicitly disallowing them prevents the algorithm from descending into certain local minima.

In Table 4 we report the average time needed for SA with and without self-intersections for the six inputs. Each nonogram is generated in about 30 minutes to 2.5 hours depending on the input image and other settings. We also list the standard deviations. Because of the abort possibilities, even a single iteration of SA can cost a significant amount of time. We notice that generation with self-intersections is on the average faster than without self-intersections. Possibly this is because more curve replacements have to be aborted in the latter case.

Finally, we wish to know whether the generated nonograms are uniquely solvable. We tested this by adapting the solving algorithm of Batenburg and Kosters [BK08] to work for curved nonograms. All of the puzzles we have generated during our experiments proved to be uniquely solvable. They all fall in the *simple* class of nonograms as defined by Batenburg and Kosters. We can often increase the difficulty of puzzles by leaving out some of the clues. If we leave out too many the puzzle will no longer be uniquely solvable, or it may become hard to solve the puzzle by hand. It is encouraging to notice that our puzzles are uniquely solvable when all clues are given, so it appears we do not have to take solvability into account during simulated annealing, where the arrangement is formed.

The six arrangements with the best scores are shown in Fig. 14.

4.1. Limitations

We have studied many of the output arrangements of the runs and visually examined the undesirable features.

Some output still has small or oddly shaped cells after SA has converged even though these features incur heavy penalties. This implies that it can be hard to remove these features using our current SA scheme, especially so for input images with many curves or local sections of the input where many picture curves lie close together. Simply increasing the weight of the small cell size and cell shape penalties will not be enough to let SA remove the features. For example, the little ball on the line of Bait has curve ends whose extensions cannot circumvent penalties.

For most input images it is also difficult to complete the initialization while excluding self-intersections. Depending on how curves are randomly connected it can become impossible to connect a given picture curve to the boundary without creating a self-intersection. Some inputs, like "butterfly" even make it impossible to generate a puzzle without self-intersections.

One of the most interesting aspects of curved nonograms is the possibility of creating a puzzle out of any possible picture rather than just pictures made up of squares, like a regular nonogram. However, in practice some input pictures are unsuitable. This is often caused by G^1-discontinuities where the angle is close to 0 or π, or a cluster of curve ends whose smooth extensions necessarily make small faces. Still, for many input pictures, geometrically good curved nonogram puzzles exist and can be generated automatically.

Our code is not optimized for efficiency; our first goal was to determine whether good nonograms can be generated automatically. Part of the inefficiency is caused by the abort cases, another part by the convergence rate of SA. Optimizations to compute the penalties more efficiently are also possible.

5. Conclusions

This paper introduced new types of nonograms that give rise to better pictures when solved on top of new puzzle possibilities: the reasoning used to solve generalized nonograms extends that of grid-based nonograms (for example in Fig. 5). Curved nonograms are essentially curve arrangements with clearly identifiable cells

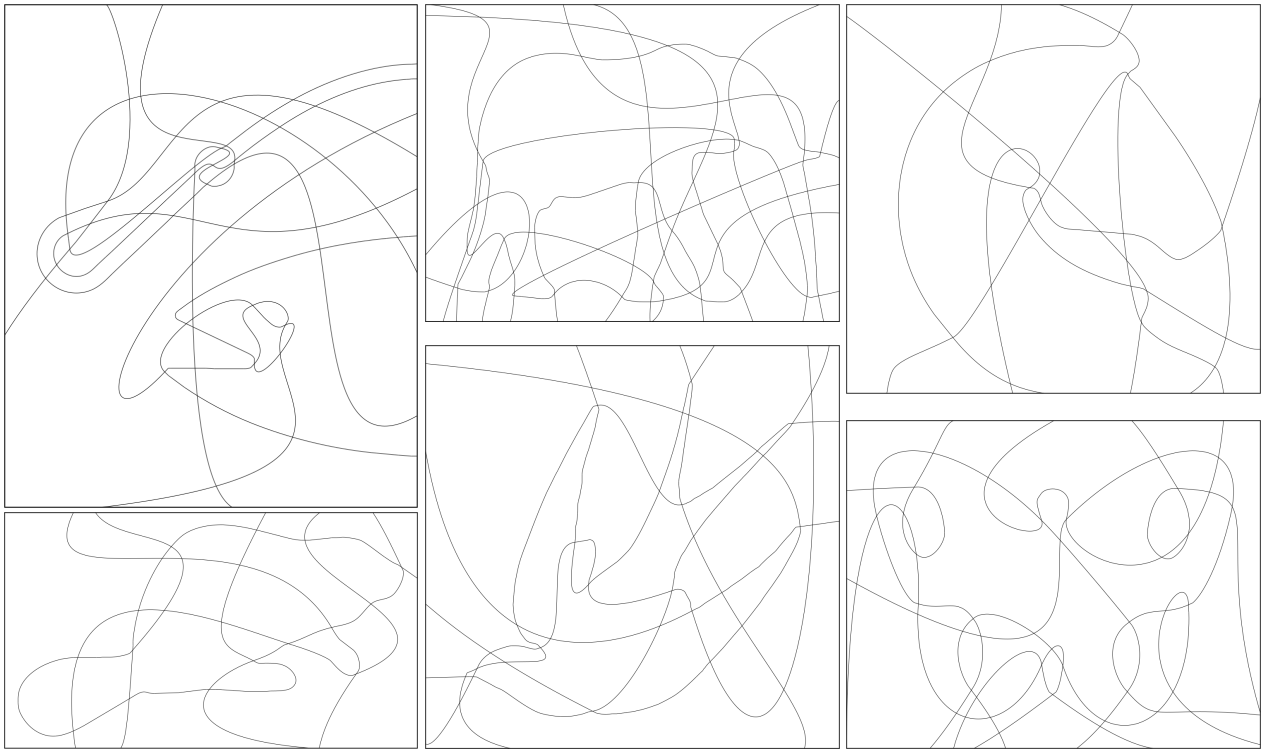


Figure 14: *Six arrangements with the best scores, over all 64 runs per picture (with and without self-intersections).*

and vertices, and with clues. We formalized geometric criteria that make such arrangements good and clear to be suitable as puzzles and showed that these arrangements can be computed in a few steps where one uses simulated annealing. We tested our implementation on several input pictures to analyze quality, convergence, efficiency, and solvability.

The results are promising but improvements are possible. On the one hand we generated good curve arrangements for the six test cases, obeying the set criteria, and simulated annealing converges. On the other hand, different runs on the same data can give quite different output and it is certain that a global optimum is often not reached. Furthermore, it appears that in several cases one can "see" the picture in the arrangement of curves, so more effort should be spent on generating curve arrangements where the picture curves are harder to distinguish from the added curves. One way to do this is to add a few curves that do not contribute to the picture boundary. Such curves always have the same clues on both sides. They can also make the curve arrangement have more small faces and other reasons for additional penalties.

An option to reduce the remaining penalty is to allow simulated annealing to move certain picture curves a little, not just the connection and bridge curves. This may spread curves a bit better, but the solution of the puzzle will no longer be exactly the same as the original picture.

To demonstrate the potential of both the puzzle idea and the implementation we generated a puzzle booklet as supplementary material. The booklet includes various other automatically generated puzzles than the test examples shown in this paper. An example is given in Fig. 15.

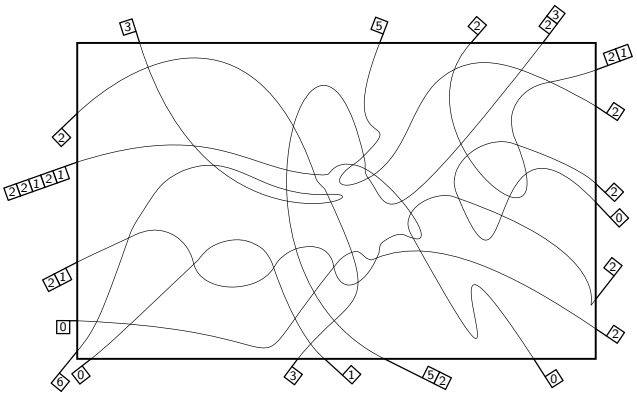


Figure 15: *Nonogram made with the help of our implementation.*

References

[Ash10] ASHLOCK D.: Automatic generation of game elements via evolution. In *Proc. 2010 Conference on Computational Intelligence and Games* (2010), pp. 289–296. 2

[ATG12] ANDALÓ F. A., TAUBIN G., GOLDENSTEIN S.: Solving image puzzles with a simple quadratic programming formulation. In *25th SIBGRAPI Conference on Graphics, Patterns and Images, SIBGRAPI* (2012), IEEE Computer Society, pp. 63–70. 2

[BHKN09] BENKERT M., HAVERKORT H. J., KROLL M., NÖLLENBURG M.: Algorithms for multi-criteria boundary labeling. *Journal of Graph Algorithms and Applications 13*, 3 (2009), 289–317. URL: http://jgaa.info/accepted/2009/BenkertHaverkortKrollNollenburg2009.13.3.pdf. 3

[BK04] BATENBURG K., KOSTERS W.: A discrete tomography approach to japanese puzzles. In *Proceedings of the 16th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC)* (2004), pp. 243–250. 2

[BK08] BATENBURG K. J., KOSTERS W. A.: A reasoning framework for solving nonograms. In *International Workshop on Combinatorial Image Analysis* (2008), Springer, pp. 372–383. 9

[BK09] BATENBURG K. J., KOSTERS W. A.: Solving nonograms by combining relaxations. *Pattern Recognition 42*, 8 (2009), 1672–1683. URL: https://doi.org/10.1016/j.patcog.2008.12.003, doi:10.1016/j.patcog.2008.12.003. 2

[BKPS10] BEKOS M. A., KAUFMANN M., POTIKA K., SYMVONIS A.: Area-feature boundary labeling. *Comput. J. 53*, 6 (2010), 827–841. URL: https://doi.org/10.1093/comjnl/bxp087, doi:10.1093/comjnl/bxp087. 3

[BKSW07] BEKOS M. A., KAUFMANN M., SYMVONIS A., WOLFF A.: Boundary labeling: Models and efficient algorithms for rectangular maps. *Computational Geometry 36*, 3 (2007), 215–236. URL: https://doi.org/10.1016/j.comgeo.2006.05.003. 3

[BKvK*16] BOUTS Q. W., KOSTITSYNA I., VAN KREVELD M., MEULEMANS W., SONKE W., VERBEEK K.: Mapping polygons to the grid with small Hausdorff and Fréchet distance. In *24th Annual European Symposium on Algorithms, ESA* (2016), LIPIcs, pp. 22:1–22:16. URL: https://doi.org/10.4230/LIPIcs.ESA.2016.22, doi:10.4230/LIPIcs.ESA.2016.22. 2

[BPRR14] BEREND D., POMERANZ D., RABANI R., RAZIEL B.: Nonograms: Combinatorial questions and algorithms. *Discrete Applied Mathematics 169* (2014), 30–42. URL: https://doi.org/10.1016/j.dam.2014.01.004, doi:10.1016/j.dam.2014.01.004. 2

[Bro11] BROWNE C.: *Evolutionary Game Design*. Springer, 2011. 2

[BS05] BEKOS M. A., SYMVONIS A.: Bler: A boundary labeller for technical drawings. In *Graph Drawing, 13th International Symposium, GD* (2005), vol. 3843 of *Lecture Notes in Computer Science*, Springer, pp. 503–504. URL: https://doi.org/10.1007/11618058_45, doi:10.1007/11618058_45. 3

[GDA*13] GERSTNER T., DECARLO D., ALEXA M., FINKELSTEIN A., GINGOLD Y., NEALEN A.: Pixelated image abstraction with integrated user constraints. *Computers & Graphics 37*, 5 (2013), 333–347. 2

[HMVDVI13] HENDRIKX M., MEIJER S., VAN DER VELDEN J., IOSUP A.: Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) 9*, 1 (2013), 1. 2

[IVK13] INGLIS T. C., VOGEL D., KAPLAN C. S.: Rasterizing and antialiasing vector line art in the pixel art style. In *Proc. of the Symposium on Non-photorealistic Animation and Rendering* (2013), ACM, pp. 25–32. 2

[JSC13] JIN J., SHIN H. J., CHOI J.: SPOID: a system to produce spot-the-difference puzzle images with difficulty. *The Visual Computer 29*, 6-8 (2013), 481–489. URL: https://doi.org/10.1007/s00371-013-0812-6, doi:10.1007/s00371-013-0812-6. 2

[KNR*16] KINDERMANN P., NIEDERMANN B., RUTTER I., SCHAEFER M., SCHULZ A., WOLFF A.: Multi-sided boundary labeling. *Algorithmica 76*, 1 (2016), 225–258. URL: https://doi.org/10.1007/s00453-015-0028-4, doi:10.1007/s00453-015-0028-4. 3

[LH14] LIM C.-U., HARRELL D. F.: An approach to general videogame evaluation and automatic generation using a description language. In *Proc. 2014 IEEE Conference on Computational Intelligence and Games* (2014), pp. 1–8. 2

[LKvK*14] LÖFFLER M., KAISER M., VAN KAPEL T., KLAPPE G., VAN KREVELD M., STAALS F.: The Connect-The-Dots family of puzzles: design and automatic generation. *ACM Trans. Graph. 33*, 4 (2014), 72:1–72:10. URL: http://doi.acm.org/10.1145/2601097.2601224, doi:10.1145/2601097.2601224. 2

[MA09] MINGOTE L., AZEVEDO F.: Colored nonograms: An integer linear programming approach. In *Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, EPIA* (2009), vol. 5816 of *Lecture Notes in Computer Science*, Springer, pp. 213–224. URL: https://doi.org/10.1007/978-3-642-04686-5_18, doi:10.1007/978-3-642-04686-5_18. 2

[MK07] MANTERE T., KOLJONEN J.: Solving, rating and generating Sudoku puzzles with GA. In *IEEE Congress on Evolutionary Computation (CEC)* (2007), pp. 1382–1389. doi:10.1109/CEC.2007.4424632. 2

[NMPL15] NEUFELD X., MOSTAGHIM S., PEREZ-LIEBANA D.: Procedural level generation with answer set programming for general video game playing. In *Proc. 7th Computer Science and Electronic Engineering Conference (CEEC)* (2015), pp. 207–212. 2

[OSL*07] ORTÍZ-GARCÍA E. G., SALCEDO-SANZ S., LEIVA-MURILLO J. M., PÉREZ-BELLIDO Á. M., PORTILLA-FIGUERAS J. A.: Automated generation and visualization of picture-logic puzzles. *Computers & Graphics 31*, 5 (2007), 750–760. URL: https://doi.org/10.1016/j.cag.2007.08.006, doi:10.1016/j.cag.2007.08.006. 2

[OU09] OKAMOTO Y., UEHARA R.: How to make a picturesque maze. In *Proceedings of the 21st Annual Canadian Conference on Computational Geometry* (2009), pp. 137–140. 2

[PL12] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer Science & Business Media, 2012. 2

[Pru86] PRUSINKIEWICZ P.: Graphical applications of L-systems. In *Proceedings of Graphics Interface* (1986), vol. 86, pp. 247–253. 2

[Sti80] STINY G.: Introduction to shape and shape grammars. *Environment and Planning B: planning and design 7*, 3 (1980), 343–351. 2

[STN16] SHAKER N., TOGELIUS J., NELSON M. J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016. 2

[TP11] TAYLOR J., PARBERRY I.: Procedural generation of sokoban levels. In *Proc. 6th Annual North American Conference on AI and Simulation in Games (GAMEON-NA)* (2011), pp. 5–12. 2

[Tsa12] TSAI J.: Solving japanese nonograms by taguchi-based genetic algorithm. *Appl. Intell. 37*, 3 (2012), 405–419. URL: https://doi.org/10.1007/s10489-011-0335-7, doi:10.1007/s10489-011-0335-7. 2

[WKDAS12] WILLIAMS-KING D., DENZINGER J., AYCOCK J., STEPHENSON B.: The gold standard: Automatically generating puzzle game levels. In *Proc. 8th Artificial Intelligence and Interactive Digital Entertainment Conference* (2012). 2

[WSC*13] WU I., SUN D., CHEN L., CHEN K., KUO C., KANG H., LIN H.: An efficient approach to solving nonograms. *IEEE Trans. Comput. Intellig. and AI in Games 5*, 3 (2013), 251–264. URL: https://doi.org/10.1109/TCIAIG.2013.2251884, doi:10.1109/TCIAIG.2013.2251884. 2

[YLK08] YOON J., LEE I., KANG H.: A hidden-picture puzzles generator. *Comput. Graph. Forum 27*, 7 (2008), 1869–1877. URL: https://doi.org/10.1111/j.1467-8659.2008.01334.x, doi:10.1111/j.1467-8659.2008.01334.x. 2