Utrecht University

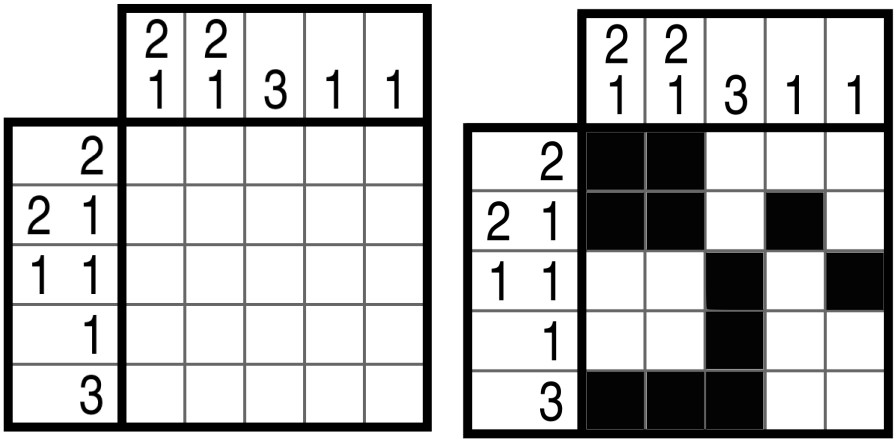
Improved Automatic Generation of Curved Nonograms

Mees van de Kerkhof

Master Thesis ICA-3822443
Computer Science
Utrecht University

supervised by
Marc van Kreveld
Maarten Löffler

November 21, 2017



(a) unsolved (b) solved

Figure 1: A regular nonogram

Abstract

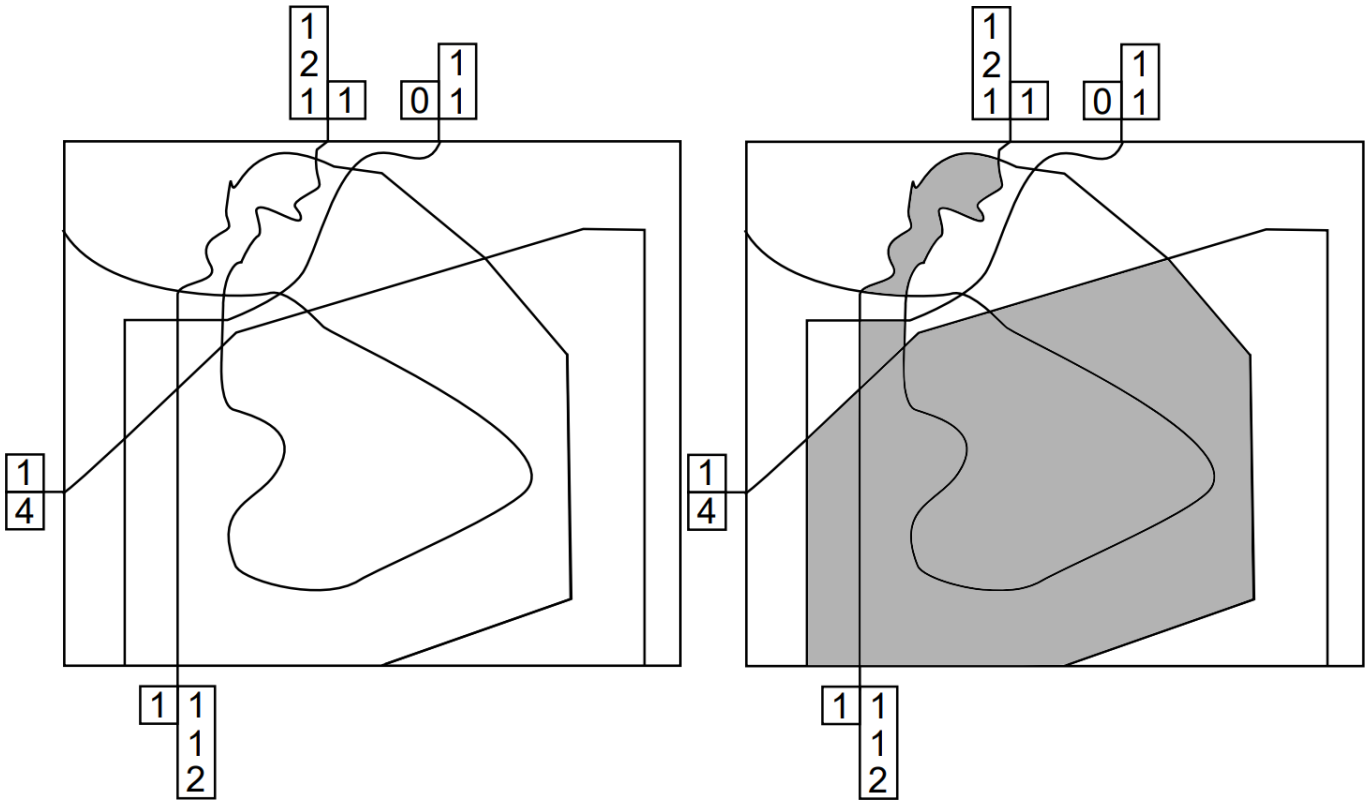
In this thesis we propose a new algorithm for automatically generating curved nonograms, which are a variation on the popular nonogram puzzle. Aside from a core algorithm, several optional features are introduced which can modify the algorithm to try and generate curved nonograms that best fulfill the aesthetic requirements for publishing.
The core algorithm and additional features are tested on several different input images. We find that the algorithm is able to generate usable puzzles in many cases. Some of the optional features prove to be promising, but require further study and adjustment to get their desired effect.

1 Introduction

Nonograms (also known as Picross, Paint-by-numbers puzzles, Griddlers, Japanese Puzzles and several other names) are a type of pen-and-paper puzzle where the puzzler draws an image by coloring in cells of a square grid based on information given about the rows and columns of the grid. An example of a nonogram is given in Figure 1. The numbers associated with each row and column (called "descriptions") reveal the lengths of the sequences of cells that should be colored in that column. For example, the number sequence "2,1" in the second row of the nonogram in Figure 1 reveals that the row should have a sequence of two colored cells, followed by at least one uncolored cell, followed by another colored cell with any number of white cells before and after the colored cells. By combining the descriptions associated with different rows and columns the puzzler can figure out the solution.

De Jong [7] introduced the *curved nonogram*, a variation on nonograms that does not use a grid. Instead, the puzzle cells are given by a set of arbitrary curves enclosed by a border. The curves are completely enclosed by the border and their endpoints are on the border, so all of the area inside the border is divided by the curves into cells of heterogeneous shape. An example of a curved nonogram is given in Figure 2. The row and column descriptions of a normal nonogram are replaced by descriptions for each side of each curve. The descriptions reveal the length of sequences of colored cells that are incident to the curve side being described in the order that they are encountered when tracing the curve from one endpoint to the other.

The abolishing of the grid makes that a curved nonogram can have any possible input image as a solution, unlike regular nonograms. The possibility of self-intersections or faces that are incident to the same curve side more than once introduces new types of information that a



(a) unsolved (b) solved

Figure 2: A curved nonogram (image taken from [7])

puzzler can use to solve a curved nonogram, making solving them a new experience even for people who are familiar with normal nonograms.

When creating a new nonogram puzzle a puzzle creator typically works backwards. The puzzle creator starts with a solution image and then breaks this into cells. The descriptions then follow automatically by how the image fits into the cells. Then the image needs to be slightly adjusted so that the nonogram has a unique solution, by changing the color of a few cells while maintaining a strong visual resemblance between the original image and the solution image. When making a regular nonogram usually the only way to change how the image is broken into cells is by changing the resolution of the grid, as the cells will always be homogeneous in size and shape. The total amount of freedom when creating a nonogram based on an input image is thus quite low. When making a curved nonogram, however, there is an infinite number of possible ways to break the image into cells, given that the cells are made by intersecting curves without a lot of restrictions. However, a lot of these combinations are unusable for creating a puzzle that needs to be solvable by humans. This is because a lot of combinations are visually ambiguous, meaning that a human puzzler cannot accurately see which cells are incident to which curve, cannot accurately trace curves or is otherwise prevented from understanding how the descriptions match the cells. It is also possible to create puzzles that have multiple solutions, which makes them unwanted. (This is also a problem for regular nonograms.) Aside from straight up unusable puzzles, among the usable puzzles there are typically still puzzles that are better than other puzzles by virtue of being even less visually ambiguous than other puzzles, having a different puzzle difficulty, being more asthetically pleasing, or other factors.

Creating a curved nonogram by hand is a time- and skill-intensive task considering the different quality factors that have to be considered. Not everyone who wants to create a curved nonogram will want to make this investment. The way puzzles are distributed is also changing. Aside from only being published on paper, nonograms and other puzzles are increasingly being published in digital apps or games. As the number of such apps increases, the demand for new puzzles increases as well. Some games offer near infinite puzzles by generating them procedurally. Other games allow users to easily create their own puzzles using computer assistance and share them with their friends. These developements have caused there to be more study into automated puzzle generation. Likewise the automated generation of curved nonograms is an interesting topic of study. By developing algorithms that can either provide computer assistance to a manual designer or generate new curved nonograms outright we can enable people to easily make their own curved nonograms without having to worry about their skill level . We can also make it possible to create a large number of curved nonograms in a short amount of time to be used in a commercial application if curved nonograms would reach the mainstream puzzle market.

De Jong introduced the first algorithm for automatically generating curved nonograms. This thesis builds upon his work to try and find a better algorithm. The focus of this new algorithm is on finding curved nonograms of minimal visual ambiguity, although there is also some effort in trying to obfuscate the solution image. We ideally want a puzzler to only realize what the solution image to a curved nonogram is going to be when he has almost solved it, instead of being able to guess the solution image immediately upon first looking at the puzzle.

2 Related Work

2.1 Nonograms

Nonograms have been studied in a variety of contexts. The difficulty of solving nonograms has been proven to be NP-hard [20][17]. However, not all individual nonograms actually are NP-hard. A polynomial time algorithm will be able to fully solve most published nonograms correctly. To differentiate between different kinds of nonograms, a classification system has been proposed [2][1]. In this system, nonograms are classified according to solvability. They can be unsolvable, solvable by multiple solutions, or uniquely solvable. Uniquely solvable nonograms that are solvable in polynomial time are called *simple* nonograms. Several algorithms for solving nonograms efficiently

have been proposed based on a variety of techniques [22][2][18]. Aside from the curved nonograms proposed by De Jong [7] another variant on nonograms called *sloped nonograms* has also been proposed [14].

2.2 Puzzle Generation

Several algorithms for automatic generation of (regular) nonograms have been proposed [1][13]. Aside from nonograms, there have also been studies done on automating the design of other types of pen-and-paper puzzles such as connect-the-dots puzzles [12] and sudoku [6], as well as more general automated puzzle design that is bound by constraints [19].

2.3 Automated Drawing

Aside from puzzles, there are other fields of study that focus on drawing in a way that optimizes given aesthetic qualities of the subject matter. A lot of study has been done into the aesthetic qualities of graphs [16][15][21] as well as into algorithms to automatically draw graphs that have these qualities [5][8]. Research has also been done into quantifiable metrics concerning the aesthetics of objects other than graphs, such as the degree of symmetry of images [23] or the similarity of polygons [11]. These metrics can be used in optimization techniques based on Local Search such as the one used in this thesis.

3 Definitions

Nonogram A *nonogram* (see Figure 1a for an example) is a pairing of a grid of uniformely sized square cells (also called *faces*) and a set of *descriptions* associated with the rows and cells of the grid. Each description is a sequence of numbers that restrict the allowed *color strings* associated with the cells in the associated row or column. A color string of a row or column is a string with characters from the color alphabet $\Sigma = \{b, w\}$ (black and white) where the sequence of the colors corresponds to the sequence of the cells of the row or column. Together the color strings make a *coloring* C of the nonogram. Let s be a color string. It is then of the form

$$s \in \Sigma^l \tag{1}$$

where l is the length of the row or column. Let d be the description of that row or column. It is then of the form

$$d = c_1 c_2 \dots c_k \tag{2}$$

where c_i is a positive integer.
s is said to *adhere* to d if it matches the following regular expression:

$$s \in w^* b^{c_1} w^+ b^{c_2} w^+ \dots b^{c_k} w^* \tag{3}$$

So the color string must be a set of k segments that are colored black that have lengths equal to the associated integer in the description. The segments are separated by at least one cell that is colored white, and the set of segments can be trailed on either side with any number of white cells. There is also an exceptional case possible where the description is a single 0. In that case the entire color string must be only white. If each color string in a coloring C adheres to their associated description, the coloring is a valid *solution* to the nonogram. A nonogram can have any number of solutions, or no solution at all. However, only nonograms that have exactly one solution are suitable for publishing.

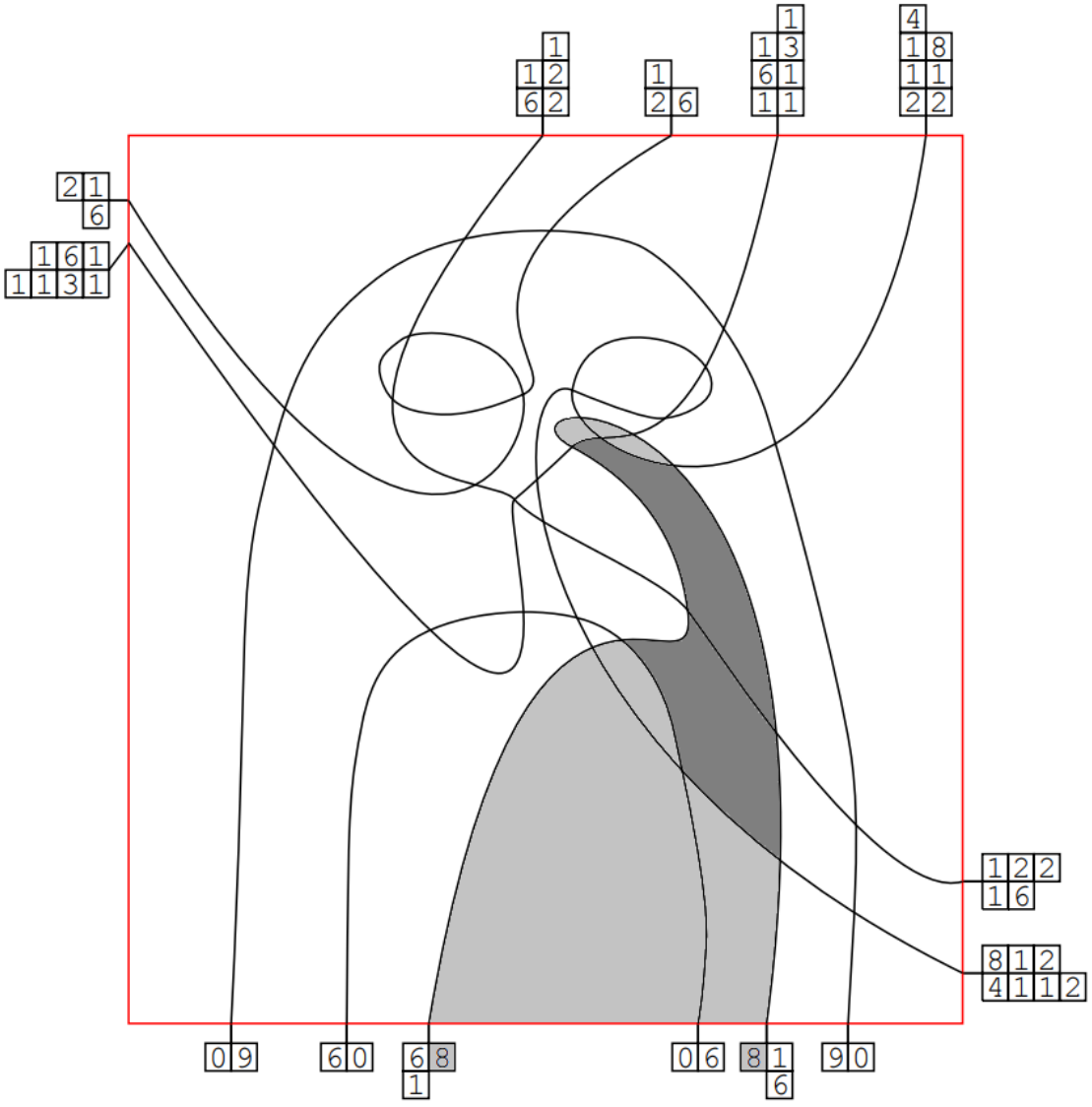


Figure 3: A curved nonogram. The faces highlighted in gray are incident to the curve side with the highlighted description. The faces highlighted in dark gray are incident multiple times in the sequence. Image adapted from [7]

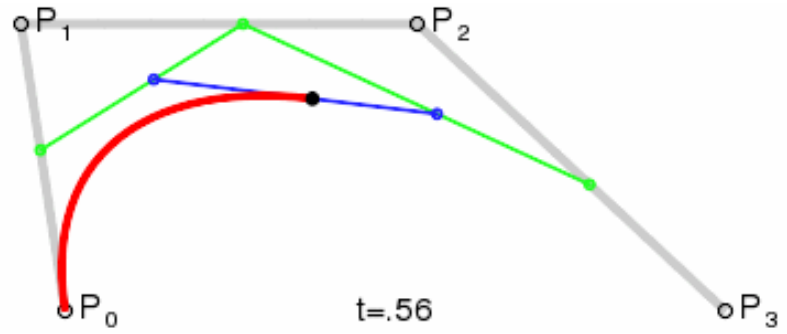


Figure 4: A cubic Bézier curve with points drawn up until $t = 0.56$. Image taken from Wikipedia [24].

Curved Nonogram Instead of having a grid and descriptions associated with the rows and columns of that grid, a *curved nonogram* (see Figure 2a for an example) consists of a set of curves that induce a set of faces that can take any shape and size. The curves that make up the curved nonogram each fall into one of three categories: There is exactly one *boundary curve*, which is a closed curve that encloses all other curves. Most curves are usually *puzzle curves*, which are open curves with both their endpoints lying on the boundary curve. Each puzzle curve also has a description associated with it. Lastly, there are *background curves* [7], which are open or closed curves that completely fall within the boundary curve and have no further information associated with them. The faces of the arrangement that is induced by the curves are the cells of the curved nonogram.

Each of the curves has two sequences of cells associated with it; one for each curve side. These sequences are sequences of incident cells that are encountered when tracing the associated curve side from one endpoint to the other. It is also possible for the same cell to be a part of the same sequence more than once as seen in Figure 3. Each of the sequences has a description associated with it that limits the accepted colorings of the sequence, making the sequences analogous to the rows and columns of a regular nonogram. Color strings for the sequence adhere to the description if it has segments of cells that are colored black with the same lengths as the numbers in the description and in the same order, separated by at least one white cell each. If a coloring adheres to all descriptions at once, it is a valid solution to the curved nonogram.

Bézier Curves A *Bézier Curve* [9] is a parametric curve that is frequently used in computer graphics and other fields. It is defined by a number of *control points*. The Bézier Curve always lies within the convex hull of its control points. Bézier curves are useful because they can be easily manipulated by performing transformations on the control points. Let $B_{P_0,P_1,\ldots,P_n}$ be a bezier curve defined by $n+1$ control points. The points on the Bézier curve are then given by the following recursive formula:

$$B_{P_0}(t) = P_0 \tag{4}$$

$$B(t) = B_{P_0,P_1,\ldots,P_n}(t) = (1-t)B_{P_0,P_1,\ldots,P_{n-1}}(t) + tB_{P_1,\ldots,P_n} \tag{5}$$

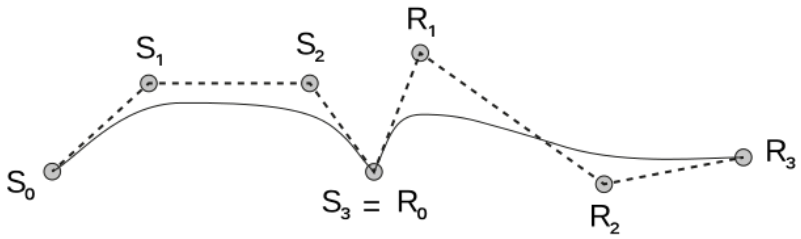
for $0 \leq t \leq 1$
This means that each point is the result of linearly interpolating between the control points and recursively linearly interpolating between those results until we get a single point. In this thesis we only use Bézier curves with exactly 4 control points, called *cubic Bézier curves*. Unrolling the

recursion in the formula for the points on such a curve we get:

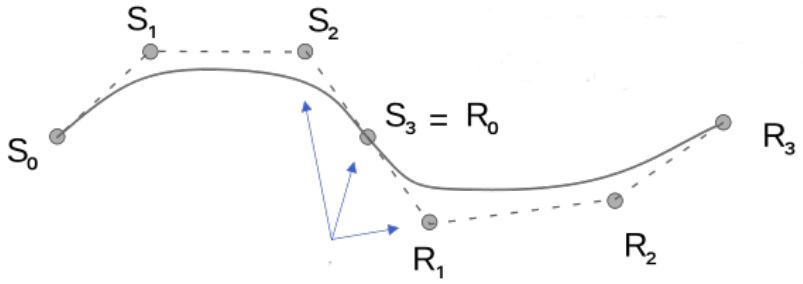
$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \tag{6}$$

for $0 \leq t \leq 1$

An example of a cubic Bézier is given in Figure 4. In this figure the points have been generated up until $t = 0.56$. Each point has been generated by linearly interpolating with the given t between each pair of consecutive control points to get the control points of a new Bézier curve. Then the proces is repeated until a single point remains.



(a) A C^0-continuous curve connection. The connection is not smooth and the individual curves can be clearly distinguished.



(b) A C^1-continuous curve connection. Note that S_2, S_3 and R_1 are colinear.

Figure 5: Comparison between a C^0- and C^1-continuous curve connections. Images taken from Wikipedia [25].

Parametric Continuity Bézier curves can be connected to each other to make *composite Bézier curves*. Such a connection can be made by having the endpoint of one Bézier curve be equal to the endpoint of another Bézier curve. To describe the smoothness of a composite Bézier curve, we use the notion of *parametric continuity*. The order to which two curves are continuous is written as C^n, where n is the degree of continuity. Two curves that are C^{-1} continuous are not continuous at all and do not share endpoints. If the curves are C^0-continuous, they share an endpoint, but can have radically different directions as seen in Figure 5a. It can be clearly seen where one curve ends and the other begins. If a curve is C^1-continuous, not only are the two curves continuous, their derivatives are as well. There are infinite orders of parametric continuity. Two curves can be said to be C^n-continuous if their first n derivatives are continuous with each other. If two curves are C^1-continuous their transition looks smooth and it can no longer be seen where one curves goes into the next. Let S be a Bézier curve with control points $\{S_0 \dots S_n\}$ and R be a Bézier curve with control points $\{R_0 \dots R_n\}$. For S and R to be connected in a way that is C^1-continuous, their control points must satisfy the following equations [7]:

$$R_0 = S_n \tag{7}$$

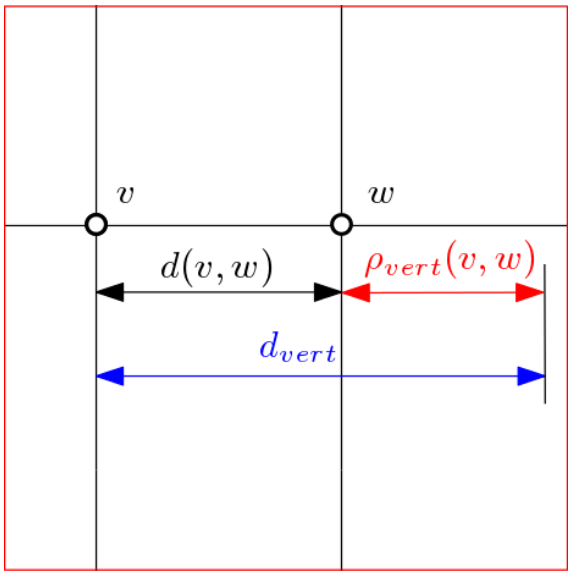


Figure 6: The distance between vertices v and w, $d(v, w)$ is lower than the threshold d_{vert}, so a penalty $\rho_{vert}(v, w)$ is incurred equal to the difference

$$R_1 = S_n + (S_n - S_{n-1}) \tag{8}$$

See Figure 5b for an example. It is of course also possible to connect R_0 to S_0 or R_n to S_n. The rest of the control points then need to be flipped accordingly in the equation.

Some curves do not have parametric continuity while still having *geometric continuity*. The orders of geometric continuity are written as G^n. Two curves are G^n continuous if they can be parameterized to have C^n continuity without necessarily being C^n-continuous already. For B and Q to be continuous R_1 and S_{n-1} would have to have opposite directions from the join point, whereas for C^1-continuity they must also have equal distance to the join point.

4 Measuring curved nonogram quality

To be able to try to find an optimal curved nonogram, we must define a measure of puzzle quality. The main objective of this study is to generate curved nonograms that are visually unambiguous. So we score our puzzle on the presence of a number of factors that negatively affect the visual unambiguity. The lower the score, the better the puzzle. The factors that we consider are listed below.

Vertex Distance If two of the vertices of the arrangement created by curve intersections are close together this indicates that the puzzle is ambiguous. So we use the distance between every pair of vertices as a measure of ambiguity. Once the distance between a pair of vertices is large enough, however, further distance between the two does not decrease ambiguity. So we only measure the distance between vertices if that distance is below a given threshold d_{vert}. If two are below a closeness threshold, they contribute a *Vertex Penalty* to the score of our arrangement equal to the distance threshold minus their distance, see Figure 6 for an example. By structuring it like this penalties can be calculated quickly and are bound between 0 and d_{vert}, which means the score can be normalized.

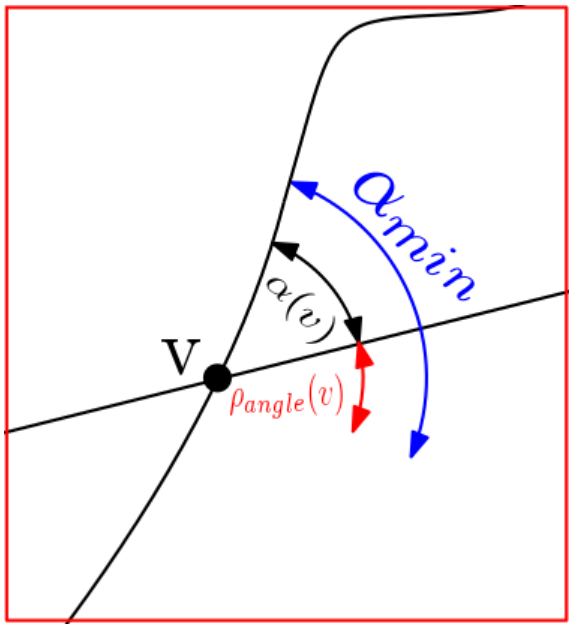


Figure 7: The intersection v has a smaller angle $\alpha(v)$ than the threshold α_{min}, so a penalty $\rho_{angle}(v)$ is incurred equal to the difference

Because we want to be able to compare the scores of puzzles that were created using a different input file, we want to normalize the penalty. The theoretical maximum vertex penalty is the number of possible vertex pairs multiplied by d_{vert}. By dividing the final penalty by this number we get a score between 0 and 1. (Because a normal solution will never come close to being as tightly clustered as this theoretical maximum, the penalty is always very close to 0.) By doing this normalization we can compare our penalty to that of a solution that has a different number of vertices. The total vertex penalty ρ_{vert} is thus given by

$$\rho_{vert} = \frac{\sum_{v,w \in V | v \neq w \wedge d(v,w) < d_{vert}} d_{vert} - d(v,w)}{\sum_{v,w \in V | v \neq w} d_{vert}} \tag{9}$$

where $d(v, w)$ gives the (Euclidean) distance between two vertices v and w.

Vertex Angle Curve intersections that have a very small intersection angle make it hard to tell if curves are intersecting or merely "touching", increasing ambiguity. So we use intersection angle as a measure for ambiguity. Like with vertex distance, once the angle is over a certain threshold a further increase does not lower the visual ambiguity. So we only measure the angles that are smaller than a given threshold α_{min}. If an intersection angle is smaller than this threshold, it adds an *Angle Penalty* to the score equal to the angle threshold minus the intersection angle. For each intersection, we only consider the smallest angle between curves included in the intersection.

To normalize the penalty as we did with the vertex distance, we divide the penalty by the theoretical maximum penalty that would be given in the situation where each intersection is 0 degrees. The total angle penalty ρ_{angle} is then given by

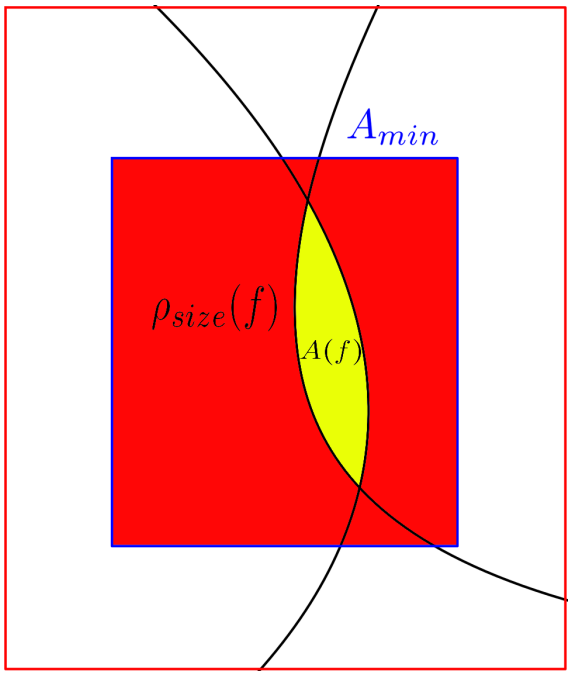


Figure 8: The area of face f (shown in yellow) $A(f)$ is smaller than the minimum area A_{min} (shown in blue), so it incurs a penalty $\rho_{size}(f)$ equal to the difference (shown in red)

$$\rho_{angle} = \frac{\sum_{v \in V | \alpha(v) < \alpha_{min}} \alpha_{min} - \alpha(v)}{\sum_{v \in V} \alpha_{min}} \tag{10}$$

where $\alpha(v)$ gives the smallest angle between the curves incident to vertex v.

Face Size Small faces are easily missed when tracing a curve and feature many curve intersections near each other which makes it hard to keep track of the curves, so they can greatly increase the ambiguity of a puzzle. To use the face size as a measure of visual ambiguity, we include the area of each face in our score metric. Again, once a face is big enough, making the face bigger does not significantly decrease visual ambiguity. (In fact, faces that are too big are undesirable for aesthetic reasons. We do not optimize for this though, as our focus is on making the puzzle visually unambiguous.) To reflect this in our puzzle score, we look at the arrangement faces that have an area smaller than a preset area threshold A_{min}, and have them contribute a *Size Penalty* to the score equal to A_{min} minus the face area. We again normalize the penalty. The total size penalty ρ_{size} is then given by

$$\rho_{size} = \frac{\sum_{f \in F | A(f) < A_{min}} A_{min} - A(f)}{\sum_{f \in F} A_{min}} \tag{11}$$

where F is the collection of faces in our arrangement and $A(f)$ is a function that gives the area of a face f.

Face Shape Faces that are highly concave or elongated are more ambiguous than convex, compact faces. They are also more annoying to color when trying to solve the puzzle using pen and paper. As a measure of the quality of the shape of a face we use the *Compactness* [4] of each face. This is the area of the face over its perimeter squared. As with the other components of our scoring function, we do not care about the compactness once it is over a certain threshold, as compactness is maximised if a face is circular, but we do not want only circular faces. So we only look at faces with a compactness lower than a preset compactness threshold C_{min}. Faces with a small compactness contribute a *Shape Penalty* to the score equal to C_{min} minus the compactness of the face. We again normalize the penalty. The total shape penalty ρ_{shape} is then given by

$$\rho_{shape} = \frac{\sum_{f \in F | C(f) < C_{min}} C_{min} - C(f)}{\sum_{f \in F} C_{min}} \tag{12}$$

where $C(f) = \frac{A(f)}{perimeter(f)^2}$.

Obfuscation Although the optimization is mainly focused on minimizing visual ambiguity, this is not the only consideration. Another concern is obfuscating the solution image. If the puzzler correctly solves the puzzle he is rewarded with an image. However, because the boundaries of this image are also curves in the puzzle, it is possible to see what the image is going to be before solving the puzzle. We do not want this because it ruins the surprise, and may tempt puzzlers into cheating by using knowledge of what the image is likely to look like to determine which cells to color, rather than the curve descriptions.

To obfuscate the solution image, we want the curves we add during our algorithm to look the same as the input curves, that way a puzzler will not be able to tell which is which as easily. To determine if two curves look the same, we consider their curvature. Curves that are straight lines look alike, and look different from curves with a high curvature. Of course, we cannot compare curves on a pairwise basis. The total set of added curves needs to resemble the total set of input curves, but a single curve does not (and usually cannot) have to look like every single other curve. To compare the set of input curves to the set of added curves, we convert the sets to histograms, using Algorithm 1. We get a number of equidistant points from each curve in the set and compute the curvature at each point by calculating the reciprocal of the radius of the osculating circle at that point. The curvatures are then divided into a preset number of bins of equal width. Instead of calculating k_{min} and k_{step} for both histograms, they are only calculated for the Histogram of input curve curvatures and then also used for filling the added curve Histogram.

After both histograms are made, they are normalized. We then calculate the vector distance between the histograms. Our *Obfuscation penalty* ρ_o is thus given by

$$\rho_o = \sqrt{(H_i[0] - H_a[0])^2 + (H_i[1] - H_a[1])^2 + \ldots + (H_i[n-1] - H_a[n-1])^2} \tag{13}$$

where H_i and H_a are the (normalized) histograms of the curvatures of the input curves and added curves respectively.

Combining penalties For the evaluation function that we use during our algorithm we combine the different penalties described above by multiplying them with a weight and adding the weighted scores to get our final score. So our score is given by:

$$score = w_{vert}\rho_{vert} + w_{angle}\rho_{angle} + w_{size}\rho_{size} + w_{shape}\rho_{shape} + w_o\rho_o \tag{14}$$

Algorithm 1: Constructing a curvature Histogram from a set of cubic Bézier curves

Data: Set of cubic Béziers C, point distance d_{point}, step size ϵ, curvature function $\kappa : (Bézier, Point) \to \mathbb{R}$, number of bins n

Result: Histogram H with n bins

$K \leftarrow \emptyset$
for $c \in C$ **do**
  $t \leftarrow \epsilon$
  $p_{prev} \leftarrow c[t]$
  $K \leftarrow K \cup \kappa(c, p_{prev})$
  while $t < 1$ **do**
    $t \leftarrow t + \epsilon$
    if $d(c[t], p_{prev}) > d_{point}$ **then**
      $p_{prev} \leftarrow c[t]$
      $K \leftarrow K \cup \kappa(c, p_{prev})$
    end
  end
end
$k_{min} \leftarrow \min_{k \in K} k$
$k_{max} \leftarrow \max_{k \in K} k$
$k_{step} \leftarrow \frac{k_{max} - k_{min}}{n}$
for $i \leftarrow 0$ *to* $n - 1$ **do**
  $H[i] \leftarrow 0$
end
for $k \in K$ **do**
  $k_\delta \leftarrow k - k_{min}$
  $j \leftarrow \lfloor \frac{k_\delta}{k_{step}} \rfloor$
  if j is smaller than 0 or larger than $n - 1$, set it to 0 or $n - 1$ respectively
  $H[j] \leftarrow H[j] + 1$
end
return H, k_{step}, k_{min}

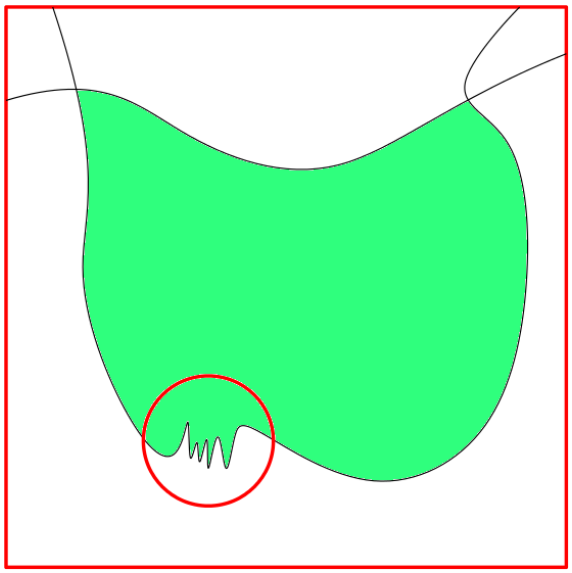


Figure 9: The green face has an ambiguous section (marked by the red circle.) Due to the large area of the face it might still pass the compactness threshold in spite of this ambiguity. The dilation penalty would detect this ambiguity.

with $w_{vert}, w_{angle}, w_{size}, w_{shape}, w_o \geq 0$

where $w_{vert}, w_{angle}, w_{size}, w_{shape}$ and w_o are the weights for our penalties. The lower the score an arrangement has the better it is. The best possible score achievable is 0.

Unused measures Many of the penalties included in our score function were also used by de Jong [7]. Of the penalties used by their algorithm we did not include the *dilation penalty*, which calculates the ratio between the Euclidean distance between points and their graph distance when interpreting the puzzle as a plane graph. The reason for not including it is how long it takes to compute dilation, not that it would not add anything to the quality of our score function.

There are situations where clear examples of visual ambuguity would not be caught by our current score function. For example we could have an extremely large face that has a section that is visually ambiguous, see Figure 9. Because of the large area of the face, the shape penalty might not catch this ambiguity, while the dilation penalty would catch it. So there is a trade-off between computation speed and effectivity of the score function.

The reason we can get away with not including dilation is that sections such as those in Figure 9 are unlikely to arise as a result of our algorithm. As we only add cubic Béziers the amount of complexity created by single curve chains is limited unless the ambiguous areas are already present in the input image. In this latter case we would never be able to remedy the ambiguity, even if we did manage to detect it. So the benefits of including dilation do not weigh up against the significant speed decrease.

Aside from the dilation penalty, other measures could also be thought up to include in the score function. For example the total length or curvature of the added curves. For each of these measures, we need to consider the trade-off between the increased accuracy and the increased computation time. Another factor we have to take into account when thinking of including more measures into our score function is that each additional measure reduces the total impact on the score of all of the other measures. If we would add measures that do not provide additional capabilities for detecting visual ambiguity, the score function worsens as the more accurate measures

get drowned out. Our score function as given in the above paragraph is therefore only one possible solution to this great trade-off, and further research might discover other score functions with different measures that work just as well or possibly even better.

5 Algorithm

In this section we describe the algorithm that we use to generate curved nonograms. Our goal is to create curved nonograms of minimum visual ambiguity, of which the solution image cannot be immediately spotted. Our algorithm takes in a solution image and produces a set of curves that can be used to create a curved nonogram of high quality. The final step of writing a description to associate with each curve is considered out of scope for this thesis as the placement and look of the descriptions does not influence the visual ambiguity or obfuscation of the puzzle. So if someone would want to publish a puzzle generated by the algorithm the descriptions would have to be added by hand. The algorithm also does not consider puzzle solvability or difficulty as they are outside of the scope of this thesis. We do know that each curved nonogram we create has at least one solution, namely the input image.

5.1 Overview

Our algorithm attempts to arrive at a good solution by generating an initial (often bad) solution and iteratively adjusting it to improve it to an acceptable level. An overview of the algoritm can be seen in Algorithm 2. (In the pseudocode, variables with names written in cursive (e.g. $cooling\_rate$) are constants that can be set beforehand.) We start by preprocessing the input by cutting the curves up into pieces. We interconnect some of the pieces, and extend the rest of the pieces to the boundary to get an initial arrangement. We optimize the extensions using Simulated Annealing. We then split all of our curve extensions at each intersection and optimize the pieces using a pairwise iterative local search. The steps of the algorithm are explained in more detail below.

Algorithm 2: Generate Curved Nonogram

Data: Set of input curves C_{in}, Boundary rectangle R, Score function f_{score}
Result: A curved nonogram
$C_{puzzle} \leftarrow$ `CutCurves(`C_{in}`)`;
$C_{puzzle} \leftarrow$ `InterConnect(`$C_{puzzle}, connect\_rate$`)`;
$C_{ext} \leftarrow$ `Extend(`C_{puzzle}`)`;
$C_{ext} \leftarrow$ `OptimizeBySA(`$C_{ext}, f_{score}, starting\_temperature, cooling\_rate$`)`;
$C_{ext} \leftarrow$ `OptimizePairwise(`$C_{ext}, pair\_iterations$`)`;
$puzzle \leftarrow$ `BuildPuzzle(`C_{puzzle}, C_{ext}, R`)`;
return $puzzle$

5.2 Input

Our input consists of a set of chains of cubic Bézier curves (such a chain is called an *input curve*, contained in a bounding rectangle (instead of a rectangle other shapes that form a closed boundary could also be used for curved nonograms, but our focus is on bounding rectangle as that is all our test inputs use.) The curves induce a planar arangement. The faces of this arrangement each have a color associated with them. Because we ignore solvability in our algorithm, the color information is ignored as well.

The reason for using cubic Bézier curves is their compatibility with the SVG file format and that they can be easily adjusted by shifting their control points. The input images we use for our experiments are the ones that were created by De Jong so we can compare the results of the different algorithms. The images can be seen in Appendix B

Our core algorithm also uses the same solution space as De Jong, which is expanded by some optional algorithm adjustments. In this solution space, instead of looking at all possible variations of curves, we break up the input curves into C^1-continuous pieces and extend them with cubic Bézier curves to the solution boundary. This solution space works well for creating curved nonograms because it limits the total number of curves to be optimized, and limits the amount of data that needs to be stored for each curve. This allows for a relatively fast optimization process. This solution space also has the advantage that every generated puzzle has at least one valid solution that is exactly equal to the input image. Our larger solution space that is created when interconnecting ECEs (see Section 5.5) or applying pairwise optimization (see Section 5.7) maintains these advantages. The number of curves we optimize is limited and for each curve we only need to store four control points.

5.3 Finding ECEs

The first part of our algorithm is breaking up the input curves into C^1-continuous pieces so that they can be extended. The endpoints of these pieces are called *Extendable Curve Ends (ECEs)* because we will extend these points with new Bézier curves. To find the ECEs, we examine each curve chain of the input. We check each pair of consecutive curves to see if they are C^1-continuous. If they are not, we break the chain into two new chains, and add the endpoints to the list of ECEs.

5.4 Extending ECEs

To get an intial solution that we can later optimize, we want to extend each ECE with a cubic Bézier curve that connects the rest of the curve chain to the boundary.

Let b_0 be the curve we are extending that has control points q_0, q_1, q_2, q_3. (We assume b_0 to be directed so that q_3 is the ECE) We then add a new curve b_1 with control points p_0, p_1, p_2, p_3. De Jong demonstrates [7] that our new curve b_1 has three degrees of freedom: We can choose the x- and y-coordinates of p_2 as long as they are within the boundary, and we can place p_3 anywhere on the boundary. p_0 and p_1 each have only one valid location, because placing them anywhere else will either break the curve chain or break C^1-continuity, which would make the puzzle ugly. b_1 needs to extend b_0's curve chain, so p_0 must be equal to q_3. To preserve C^1-continuity, p_1 must be equal to $q_3 + (q_3 - q_2)$. Like in de Jong's algorithm, if the second control point would be placed out of bounds, we instead place it on the boundary so that it is colinear with q_2 and q_3. We also make sure that p_1 has a minimum distance d_{min} from p_0 to prevent extremely sharp turns right after connecting. So the actual equation for p_1 is equal to

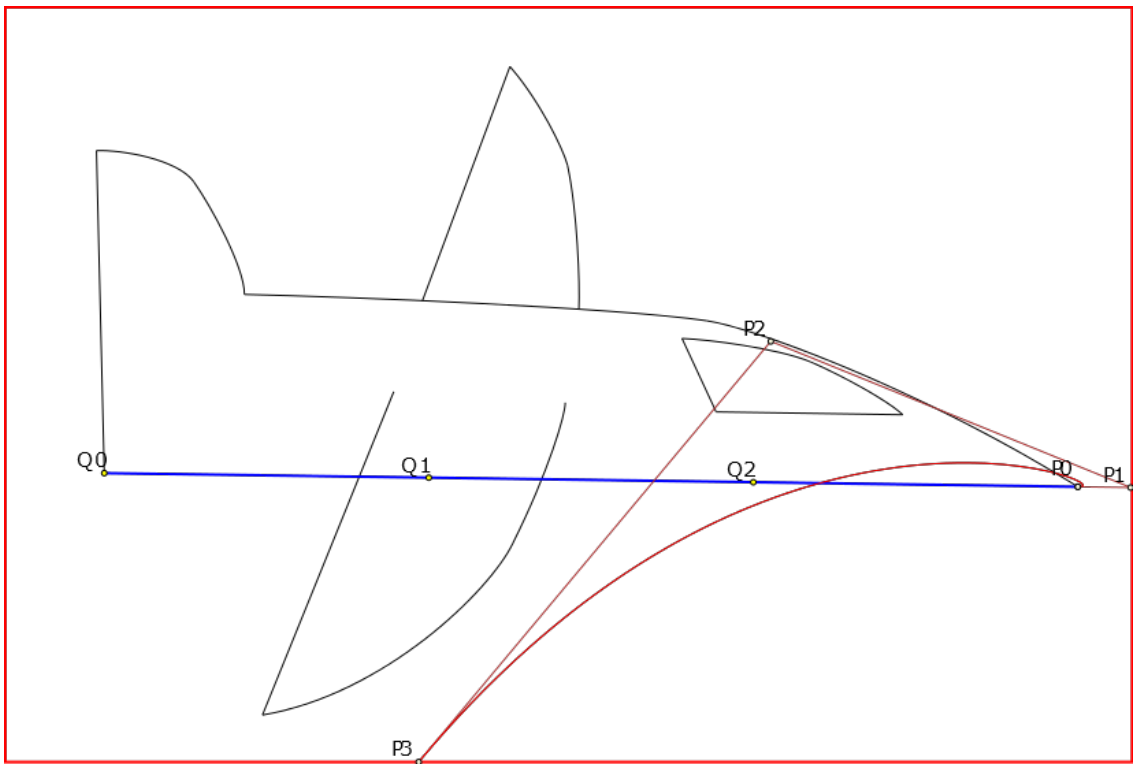
$$p_1 = q_3 + s(q_3 - q_2) \tag{15}$$

where s is a scale factor determined by the following piecewise function:

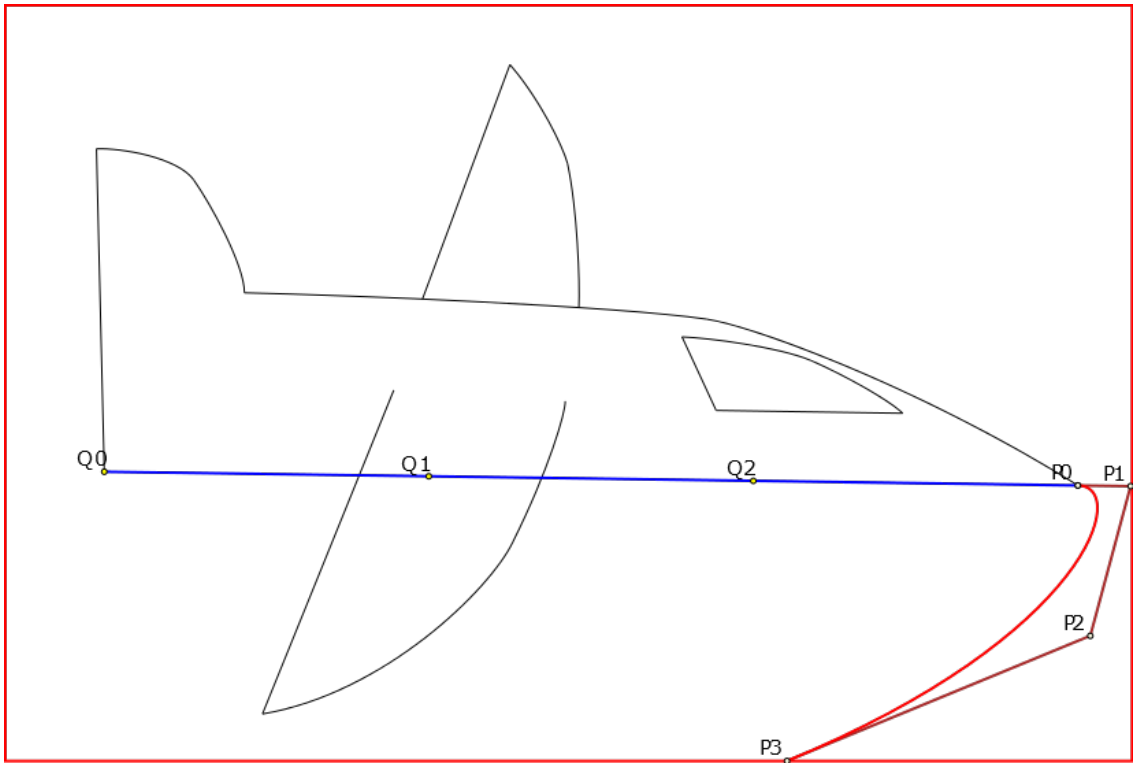
$$s = \begin{cases} \frac{d(p_0, p_b)}{q_3 - q_2} & p_1^* \notin R \\ \frac{d_{min}}{q_3 - q_2} & d(p_1^*, p_0) \leq d_{min}, p_1^* \in R \\ 1 & otherwise \end{cases} \tag{16}$$

where $p_1^* = q_3 + (q_3 - q_2)$, R is the bounding rectangle and p_b is the intersection point of the line through q_2 and q_3 and R that is closest to p_1^*. Our added curves are thus C^1-continuous unless prevented by our constraints, and G^1-continuous otherwise.

In De Jong's algorithm, each of the puzzle curves is fully random (within these 3 degrees of freedom) to create an initial solution which is improved by an optimization algorithm. A random solution created like this has a high probability of having puzzle curves that make extremely sharp hairpin turns very close to p_0 if p_0 is near the boundary. These turns are undesirable. Optimization wil of course smooth them out somewhat, but it is quite easy to limit their appearance by limiting where p_2 is placed as well as ensuring a minimum distance between p_0 and p_1 as mentioned above. The sharp turns are prevalent near the boundary because p_1 is always an extension of the previous



(a) The placement of $P2$ causes an extremely sharp turn right after connecting



(b) By ensuring $P2$ is placed to the right of $P0$, a sharp turn is prevented

Figure 10: During curve extension, we can improve the initial solution by restricting curve placement

curve and thus usually lies very close to the border. See Figure 10 for an example. There is then a very large chance when placing p_2 randomly that it will be far away on the opposite side of p_0 compared to p_1, making a sharp turn. To combat this, for the initial solution, we do not place p_2 randomly. Instead we divide our area into four quarters with the dividing axis aligned with the bounding rectangle and centering on p_0. We then put p_2 into the same quarter as p_1, so the distance between them is limited. This is an easy way to improve the quality of our initial solution that is easy to implement and quick to compute. During optimization, we allow p_2 to be placed anywhere, to ensure we are not trapping ourselves in a local minimum. It would also be possible to divide the space into two halves with the line that is perpendicular to the vector from p_0 to p_1 that passes through p_0 and making sure p_2 is on the same side of the line as p_1, but this version has not been tested in this thesis.

If we would further want to improve our initial solution, we could consider biased placement of p_3 as well where it has a higher chance of being placed near p_0 if it is close to the boundary, but this has also not been tested in this thesis due to time constraints.

5.5 Connecting ECEs

An optional step to do before extending our ECEs to the bounding box we examine is to connect the ECEs to each other, rather than connecting each ECE to the boundary. This reduces the number of free ECEs in the puzzle, making it more simple to optimize. It also makes for a more interesting puzzle with a more diverse structure.

To connect two ECEs we add a new cubic Bézier that connects the two curves. Because we want to ensure C^1-continuity, any two ECEs can be connected by only one specific cubic Bézier. Let b_0 and b_1 be the two curves we want to connect with new cubic Bézier curve b_2. Let b_0's control points be $q_0 \dots q_3$, b_1's control points be $r_0 \dots r_3$ and b_2's be $p_0 \dots p_3$. We assume b_0 and b_1 to be directed so that q_3 and r_0 are the ECEs. To preserve connectedness and C^1-continuity with both b_0 and b_1 p_0 must be equal to q_3, p_1 must be equal to $q_3 + s_1(q_3 - q_2)$, p_2 must be equal to $r_0 + s_2(r_0 - r_1)$ and p_3 must be equal to r_0, where s_1, s_2 are scale factors so the points lie within the boundary like in Section 5.4. So b_2 has zero degrees of freedom.

In our algorithm, we choose the ECEs to connect at random with the condition that we do not create a closed loop of curves. If connecting two curves would create a loop, we choose new ECEs to connect instead. Other conditions can also be imposed (e.g. choosing new ECEs to connect if the connecting would create a self-intersection, see Section 5.8.) The percentage of ECEs to connect is set by a parameter *connect_rate* beforehand. See Figure 14 in Appendix B for an example of a curved nonogram that was created while interconnecting the ECEs.

5.6 Optimization by Simulated Annealing

Our optimization is based on Simulated Annealing [10], which is a well known optimization technique that reduces the risk of getting trapped in a local minimum. The optimization algorithm is shown in Algorithm 3. We continuously generate a neighbor solution C_n by making a small change to our current solution C_{curr}. If this new solution is better than our old one we accept it as the new C_{curr}. If the new solution is worse it still has a chance of being accepted to prevent getting stuck in a local minimum. The probability of acceptance P_{accept} is based on the difference in quality as well as the temperature T, which is a parameter that decreases as the algorithm goes on. As T decreases, the probablility of accepting inferior solutions decreases also. Once T is small enough we consider the optimization finished and return the best solution found overall. The individual steps of the optimization are further explained below. The score function we use during Simulated Annealing is given in Section 4.

5.6.1 Generating neighbors

We generate a neighbor by taking the current solution and changing one of the curves that have been added during the algorithm initialization. To do this, we first select an appropriate curve.

Algorithm 3: Optimization by Simulated Annealing

Data: Set of curves C_{in}, Score function f_{score}, Starting temperature T_0, Cooling rate k
Result: Optimized set of curves
$C_{best} \leftarrow C_{curr} \leftarrow C_{in}$;
$T \leftarrow T_0$;
while $T > 1$ **do**
  $C_n \leftarrow$ `GetNeighbor`(C_{curr}, f_{score});
  $\Delta_{score} \leftarrow \frac{f_{score}(C_n)}{0.01 * f_{score}(C_{curr})} - 100$;
  $P_{accept} \leftarrow e^{\frac{-\Delta_{score}}{T}}$;
  $r \leftarrow$`GetRandomNumber`$(0, 1)$;
  if $r < P_{accept}$ **then**
    $C_{curr} \leftarrow C_n$;
    if $f_{score}(C_{curr}) < f_{score}(C_{best})$ **then**
      $C_{best} \leftarrow C_{curr}$;
    end
  end
  $T \leftarrow T * k$;
end
return C_{best}

The usability of our final puzzle result determined by its "weakest link". If most of the puzzle is good but there is one section that is too ambiguous to be able to finish the puzzle, the entire puzzle cannot be published. So, in our optimizations we want to focus on fixing these weakest links. We take this into account when selecting which curve we want to change.

When deciding on a curve, for each optimizable curve we compute how much it contributes to the score of the puzzle. To do this, we add the weighted and normalized partial penalties that are incurred only by those parts of the puzzle that are incident to the curve. Because the obfuscation is calculated globally rather than locally, it is omitted in this computation. So the partial score for curve b, $score_b$ is equal to the following equation:

$$score_b = w_{vert}\rho_{vert}(b) + w_{angle}\rho_{angle}(b) + w_{size}\rho_{size}(b) + w_{shape}\rho_{shape}(b) \tag{17}$$

where

$$\rho_{vert}(b) = \frac{\sum_{v,w \in V | v \neq w \wedge d(v,w) < d_{vert} \wedge v \mathbf{I} b} d_{vert} - d(v,w)}{\sum_{v,w \in V | v \neq w} d_{vert}} \tag{18}$$

$$\rho_{angle}(b) = \frac{\sum_{v \in V | \alpha(v) < \alpha_{min} \wedge v \mathbf{I} b} \alpha_{min} - \alpha(v)}{\sum_{v \in V} \alpha_{min}} \tag{19}$$

$$\rho_{size}(b) = \frac{\sum_{f \in F | A(f) < A_{min} \wedge f \mathbf{I} b} A_{min} - A(f)}{\sum_{f \in F} A_{min}} \tag{20}$$

$$\rho_{shape}(b) = \frac{\sum_{f \in F | C(f) < C_{min} \wedge f \mathbf{I} b} C_{min} - C(f)}{\sum_{f \in F} C_{min}} \tag{21}$$

where $a\mathbf{I}b$ means that a and b are incident. (See Section 4 for explanations of the other terms.)

We add the partial penalties for each curve together to get our *sum of partial penalties* (this is a larger number than the score of the current solution, as the scored features are incident

to multiple curves.) To select a curve to optimize, we then assign each curve a probability of being selected that is equal to the ratio of the partial penalty for that curve and the sum of partial penalties. We then select a curve at random using these probabilities. In this way, curves that cause a lot of ambiguity are more likely to be changed, without us completely ignoring curves that have a smaller impact on the total ambiguity.

When we have selected the curve we want to optimize, we change it. As mentioned in Section 5.4, the curve has three degrees of freedom. We choose the new values uniformly at random within the relevant bounds. After applying the changes, we check if the updated curve is not responsible for degenerate arrangements. As the number of curves in the input rises, the chance of creating an arrangement with a triple intersection or overlapping curves increases quickly. We also need to include a margin for error when trying to detect these degeneracies, as our implementation in C# does not handle the precision of numbers with high precision optimally. Because of this, a lot of curve changes result in an invalid arrangement. When this happens, the new values for the curve are randomized again. If we cannot find a suitable curve in a preset number tries $r_{attempts}$ we restore the original curve and try optimizing a different curve. Although this is not ideal, the impact of this margin of error on the final result is limited since the degeneracies it prevents give a bad fitness score, so it is unlikely that such arrangements would get accepted as the new current solution in either case.

5.6.2 Accepting or Rejecting the Neighbor

Once we have generated a neighbor, we score it using the evaluation function described above. We then compare this score with the score of the solution that was used to generate the neighbor. The choice to accept or reject is based on the difference between the fitness score of the neighbor solution and the previous solution. We choose by generating a random number between 0 and 1 and see if it is lower than the result of the equation

$$P_{accept} = e^{\frac{-\Delta_{score}}{T}} \tag{22}$$

where P_{accept} is the probability of accepting the solution, T is the Temperature and Δ_{score} is the percentage point increase of the new score compared to the old.

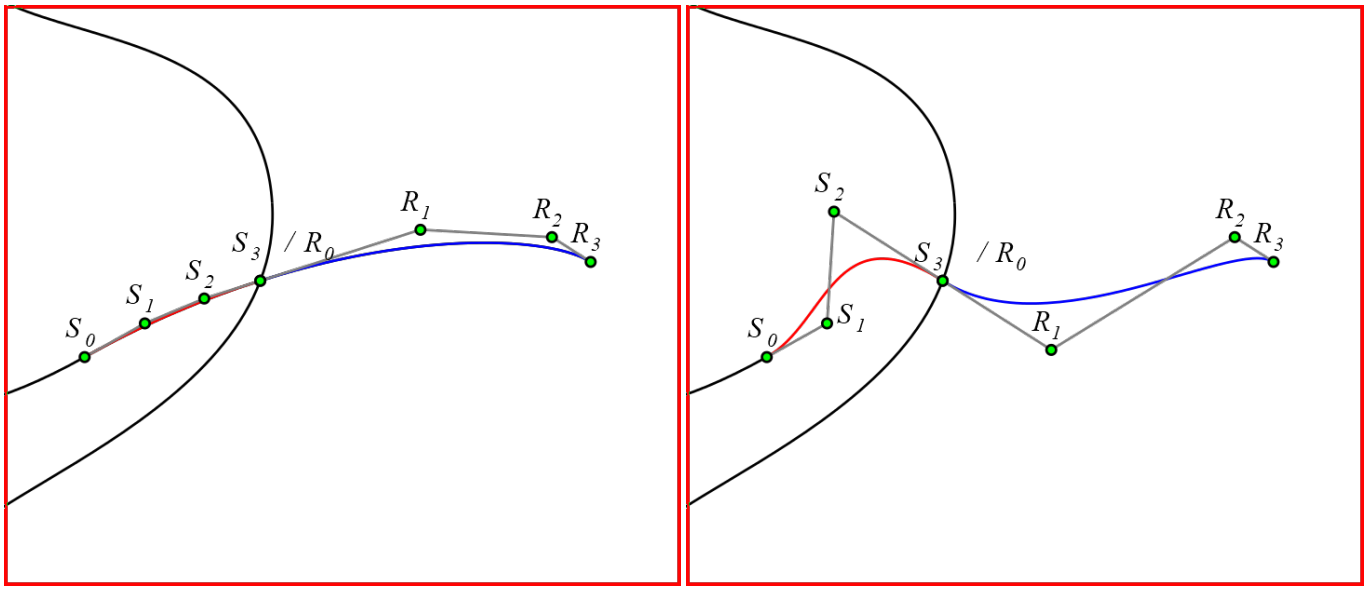
The temperature is continually decreasing as the algorithm runs. Every time a solution is tested, the temperature is multiplied with a the cooling rate k with $0 \leq k \leq 1$ afterwards. The decreasing temperature means that the probability of accepting a worse solution decreases as the algorithm nears its end when we should be closer to a globally optimal solution. The reason that Δ_{score} is on a percentage basis rather than comparing actual scores is that the scores nearing the end of optimization can be several orders lower than the scores at the start of optimization. By expressing the score as a percentage of the score of the current solution, the scores more acurately reflect the impact of the changes on the ambiguity of the solution.

Our algorithm stops when T becomes smaller than 1. It then returns the overall best solution that was found.

5.7 Secondary Optimization Round

After we have done our primary optimization, we consider the topological aspects of our solution to be settled. We have determined which of our curves intersect and where, and which faces and vertices are created by those intersections.

To try and improve our solution further, we want to see if minor changes to some curve trajectories can improve our score by slightly increasing the area of some faces or changing the shape of some faces. To do this without changing the location of the curve intersections, we first break each extension curve that has a nonzero partial penalty associated with it into pieces. Each curve is broken up so that each piece (which is also a cubic Bézier curve) is as large as possible



(a) Before a pairwise change. The red and blue curve pieces are G^1-continuous.

(b) After a pairwise change to S_2 and R_1, the curve pieces are C^1-continuous again.

Figure 11: Pairwise change to a pair of curve pieces. If the new arrangement scores better, it becomes our new solution.

without having intersections in its interior. Then, for a set number of iterations, we select two curve pieces that are C^1-continuous curve pieces. We then further optimize on those pieces. The algorithm for this second optimization round is given in Algorithm 4. The individual steps are explained in more detail below.

5.7.1 Breaking the curves

Because we have already finished our optimization using Simulated Annealing, parts of the puzzle may be visually unambiguous and add nothing to the score. We do not need to further optimize these parts. Other curves may still influence the score, e.g. by bounding a face that has a very small area. We want to improve these curves without causing major changes to the rest of the puzzle. So as optimizable curves for our second optimization round, we start by only considering the curves that have a nonzero partial penalty (see Section 5.6.1 on how partial penalties are computed.) For each such curve, we then look at each curve intersection it is a part of. At each such intersection point, we split the curve into two new cubic Bézier curves that together follow the exact trajectory of the original curve. The intersection point is the point where the two curves overlap, and is a control point to both of the curves.

We maintain a list of each pair of curve pieces we create in this way. We continue splitting the curve like this until each of its pieces has no more intersections in its interior. As a consequence of how curve splitting is implemented the curve pieces are parameterized so that they are G^1-continuous with each other rather than C^1-continuous. They can be reparameterized to be C^1-continuous but we don't need to do it in this step as our optimization makes the pieces C^1-continuous with each other as well. During our optimization, we continually choose one of these pairs at random and try changing it to improve the score of the puzzle.

5.7.2 Changing a Pair

We cannot change a curve piece by itself without affecting the smoothness of the connections between the chain of curve pieces. By themselves they have zero degrees of freedom just like

a Bézier we would use to connect two ECEs as explained in Section 5.5. However, if we move two pieces at once, we can change them while reestablishing C^1-continuity. Let S and R be the curve pieces we want to optimize with controlpoints $S_0 \dots S_3$ and $R_0 \dots R_3$ respectively. See Figure 11. S_0, S_3, R_0, and R_3 cannot be moved because the curves are connected to the rest of their curve chains there. S_1 and R_2 cannot be moved because it would break the C^1-continuity with the rest of the curve chain. However, the constraints on S_2 and R_1 can be formulated like this

$$S_2 = R_0 + (R_0 - R_1) \tag{23}$$

$$R_1 = S_3 + (S_3 - S_2) \tag{24}$$

But considering S_3 and R_0 are the same point the relationship between S_2 and R_1 can be written as

$$S_2 = 2S_3 - R_1 \tag{25}$$

If we keep S_3 constant, we can change S_2 without breaking C^1-continuity as long as we change R_1 in an opposite fashion. To do this in our algorithm, in each iteration we choose a new value for S_2 at random with the constraint that it lies within a set maximum distance d_{max} of its previous value, so as to limit the degree to which the rest of the puzzle is affected. R_1's new value is then computed so that the C^1-continuity constraint is satisfied. If the puzzle has a lower score using these new curve pieces, it becomes our new working solution.

5.8 Disallowing self-intersecting puzzle curves

The puzzle curves we end up creating in our algorithm run from one point on the boundary to another, twisting and turning along the way and intersecting other curves. It is also possible for a puzzle curve to intersect itself. These intersections are interesting because they add a new type of information that a puzzler can use when trying to solve the puzzle. They do make the curve significantly harder to trace, however, and the new type of information can be tricky to use for less experienced puzzlers. Because of this, it can be useful to be able to prevent the occurence of these self-intersections.

This is implemented by letting the algoritm check every time new random values are chosen for a curve that there is no intersection between two cubic Béziers that are part of the same puzzle curve. It also checks that there are no cubic Béziers that have an internal self-intersections. If it finds a self intersection, the change is rejected and new values have to be chosen for the curve, just like how curves that induce degeneracies are limited in Section 5.6.1. It starts checking for self-intersections during the initial extension of the ECEs.

Due to the iterative nature of extending the ECEs where ECEs are extended one at a time until they have all been extended, it is possible that one ECE cannot possibly be extended to the boundary by a cubic Bézier without intersecting the rest of the puzzle curve someplace. This case is hard to detect however. To prevent the algorithm from getting stuck in a loop where it keeps trying new random values for an extension curve that get rejected due to self-intersections, we include a safety switch where after a preset number of attempts have failed, the extension curve is marked as an exception that is allowed to have a self-intersection. This means that there is no guarantee that the puzzle is completely free of self-intersections, but they are limited in the frequency with which they appear. If someone would want an absolute guarantee that there are no self-intersections, it is possible to disable the possibility of making exceptions, but then the algorithm will have to be restarted occasionaly if it is unable to find an acceptable curve.

Algorithm 4: Pairwise optimization of curve pieces

Data: Set of curves C_{in}, Score function f_{score}, number of iterations n
Result: Optimized set of curve pieces
$C_{pairs} \leftarrow$`BreakCurves(`C_{in}`)`;
for 1 *to* n **do**
  Get a random pair P from C_{pairs};
  $P_{changed} \leftarrow$ `ChangePair(`P`)`;
  $C_{changed} \leftarrow (C_{pairs} \setminus P) \cup P_{changed}$;
  if $f_{score}(C_{changed} < C_{pairs}$ **then**
    $C_{pairs} \leftarrow C_{changed}$;
  end
end
return C_{pairs}

Image	ECEs
penguin	10
butterfly	12
coffee_cup	12
lamb	16
monitor	16
rocket	16
mask	18
airplane	22
hut	24
catface	26
church	32
space	36
pumpkin	38

Table 1: Our input images and their number of ECEs

6 Experiments

This section describes several experiments that have been performed to study the influence of several different features of the algorithm. The main goal is to find the ways in which these features could be useful when making an automated system for creating curved nonograms.

6.1 Experimental Settings

6.1.1 Test Set

As inputs we use the same input files that were used by de Jong. This is done so that the final puzzles generated by the two algorithms can be compared. This comparison has not been done in this thesis however, as a meaningful comparison of the results on visual ambiguity requires a user study. Not the full set of images is used. “car”, “fish” and “flower” are dropped. The three images use arcs in their input rather than just cubic Béziers which have not been implemented in the code used to run the experiments. The full test set is given in Table 1. The images can be viewed in Appendix B.

Name	Description	Value
w_{vert}	Vertex Penalty Weight	1000
w_{angle}	Angle Penalty Weight	3
w_{size}	Size Penalty Weight	3
w_{shape}	Shape Penalty Weight	7
d_{vert}	Vertex Distance Threshold	11
α_{min}	Vertex Angle Threshold	20
A_{min}	Face Area Threshold	55
C_{min}	Face Compactness Threshold	0.025
k	Cooling rate of T during SA	0.97
$r_{attempts}$	Maximum Number of Attempts for Randomizing Curve	10
d_{point}	Histogram Sample Point Distance	5
ϵ	Histogram Sample Step Size	0.01
n	Number of Histogram Bins	5
d_{max}	Pairwise Optimization Maximum Move Distance	100
d_{min}	Minimum distance first extension curve control points (See Section 5.4)	60

Table 2: The static settings of our algorithm

6.1.2 Test Settings

The algorithm has a lot of different parameters and settings. The settings that do not change between different experiments are given in Table 2. The settings for d_{vert}, α_{min} and A_{min} have been copied from De Jong's Algorithm setting c) for comparability of the results. The Compactness Penalty replaces the Dilation Penalty of De Jong, so a good value for C_{min} had to be determined experimentally. The current value gave lower scores for generated curved nonograms than other tested values.

The values for the weights are chosen so that the different parts of our score function have about equal impact on the score. Before weighting the values for ρ_{vert} are a lot smaller than the other penalties so it gets a large weight. After weighting the penalties lie closer together. As the complexity of the input increases the relative difference between ρ_{vert} and the other unweighted penalties will increase because ρ_{vert} is normalized by dividing by a factor that grows quadratically with input size, where the other dividing factors grow linearly. For the current inputs keeping w_{vert} constant seems to work fine, but as inputs with a larger range of complexity get tested it may be worthwile to switch to a dynamic weight that depends on input size or a different normalization method for ρ_{vert} so that the weighted penalties are more in line again.

The other static parameters have values that have been chosen experimentally, but they have not been tested as much, so better values likely exist.

Due to time constraints, most settings have been run only once for each input. Simulated Annealing has a significant random component, so ideally each setting would be tested a large amount of times to account for variance, but that has not been done for this thesis. Some of the variance is accounted for due to each setting being run for each image in the test set. Any conclusions are based on the average over all of these input images so there is at least some repetition of each setting, but this may not be enough.

6.2 Verifying the Algorithm using visual inspection

Throughout all of the experiments described below and others, 120 curved nonograms have been created and inspected visually. A number of these can be seen in Appendix B. The average score of the 88 curved nonograms that passed visual inspection (meaning that they contain no sections that are so visually ambiguous that it would prevent the puzzle from being published) is 0,158057. The average score for the 32 curved nonograms that failed this inspection is 0,401166. Furthermore, there are 29 curved nonograms that achieved the best possible score of 0. All of these curved

nonograms passed visual inspection.

This seems to indicate that our score function gives a somewhat accurate indication of visual ambiguity, although the separation between puzzles that passed and failed visual inspection is not entirely clear-cut. The puzzle with the highest score that passed has a score of 0,693041, with 28 of the puzzels that fail inspection having a lower score. The puzzle with the lowest score that failed inspection has a score of 0,000546, with 57 puzzles passing inspection while having a higher score. So while using our score function during our optimization does help move our puzzle in a less ambiguous direction, the score function is not so accurate that it could be used as a classifier for testing if an individual puzzle can be called visually ambiguous.

Important to note is also that visual inspection like this is subjective by necessity, and at the moment it is especially subjective given that all of the visual inspection was done by just one person. To reduce this amount of subjectivity the visual inspection should be repeated by as large a group of people as possible in a user study, see Section 8.

Aside from looking at the scores, the average runtime of the algorithm is around an hour. So we can also conclude that the algorithm has a good chance to create an acceptable curved nonogram in an acceptable timeframe for such an application. Increasing the number of iterations the algorithm runs could be used to further improve the quality while still taking an acceptable amount of time.

In the generated curved nonograms we can see that a lot of ambiguity is still created by sharp turns near the image edge. Despite efforts to limit the occurence of these sharp turns in our initial solution (see Section 5.4) and the visual ambiguity that sharp turns bring that increases the score, the turns still return during optimization. Although this indicates that the situation with these sharp turns would be less visually ambiguous than whatever visual ambiguities were removed during optimization, they are still unwanted.

6.3 Testing the effect of disallowing self-intersections in curve chains

As described in Section 5.8, it can be beneficial to disallow puzzle curves in our solution to self-intersect. To test the influence of disallowing these self-intersections, two sets of curved nonograms were generated. One with self-intersections allowed, and one with them disallowed. The experiment settings of the two sets are given in Table 3. The obfuscation penalty, interconnecting ECEs and pairwise optimization were disabled to more directly focus the experiment on the effects of having self-intersections enabled.

The results of the experiment are given in Table 4. It gives a comparison between the final scores, the time taken to run the algorithm and the number of self-intersections in the final image. The table shows that disallowing self-intersections generally improves the score, while also taking slightly longer. The improvement in score is relatively greater than the time increase: The curved nonograms with disallowed self-intersections have a score average that is 14.5% lower than that of curved nonograms with self-intersections, while the average time is only 3.5% higher. This suggests that even if the puzzle designer does not care if there are self-intersections in the puzzle, it can still be beneficial to turn them off, as it improves the score/time ratio.

The table also shows that the algorithm can easily prevent self-intersections if the input image is simple. For images with a lot of ECEs, however, it could not prevent all self-intersections. It did reduce their numbers. This result implies that self-intersections make a puzzle more visually ambiguous. The reason that the time increase when disallowing self-intersections is not that big even though a lot more curve randomizations are rejected is that the randomization "fails faster" if it is rejected due to a self-intersection. As soon as a self-intersection is found we can discard the curve without having to check the rest of the puzzle for degeneracies or having to update the arrangement.

6.4 Testing the effect of connecting ECEs

To test the effects of connecting some of the ECEs to each other instead of the boundary, we generated four sets of curved nonograms with different levels of connectivity. The experiment

Parameter	Description	Set 1	Set 2
w_o	Obfuscation Penalty Weight	0	0
$connect\_rate$	Percentage of ECEs to Connect	0%	0%
$pair\_iterations$	# Iterations to run Pairwise Optimization	0	0
T_0	Starting Temperature for SA	72	72
$self\_intersections$	Acceptance of Self-Intersections	Allowed	Disallowed
$exception\_attempts$	# Attempts until allowing Self-Intersections	N/A	10

Table 3: The settings of our Experiment involving disallowing self-intersections. Other values are given in Table 2

	Score		Time		# Self-Intersections	
InputName	Allowed	Disallowed	Allowed	Disallowed	Allowed	Disallowed
penguin	0,014273	0	1839	2321	1	0
butterfly	0	0	2844	2826	3	0
coffee_cup	0,016777	0	1772	2276	6	0
lamb	0,031632	0,006218	4033	3356	4	0
monitor	0	0,009432	1666	3541	2	0
rocket	0	0	2947	2364	0	0
mask	0,037112	0,076584	4847	3148	4	0
airplane	0,088418	0,042659	3170	3303	1	0
hut	0,138924	0,142949	2771	2473	4	0
catface	0,188053	0,158126	3779	4070	6	3
church	0,481681	0,408648	3916	4723	6	1
space	0,541120	0,474626	7029	7358	6	1
pumpkin	0,479699	0,425446	6367	6913	6	0
Average	0,155207	0,134207	3614	3744	3,769231	0,384615

Table 4: Comparison between puzzles generated with Self-Intersections either Allowed or Disallowed. (Corresponding to Set 1 and Set 2 in Table 3 respectively.) Times are given in seconds.

Parameter	Set 1	Set 2	Set 3	Set 4
w_o	0	0	0	0
$connect\_rate$	0%	25%	50%	75%
$pair\_iterations$	0	0	0	0
T_0	72	72	72	72
$self\_intersections$	Allowed	Allowed	Allowed	Allowed
$exception\_attempts$	N/A	N/A	N/A	N/A

Table 5: The settings of our experiment varying the connection rate. Other values are given in Table 2, parameter explanations in Table 3

settings are given in Table 5. The results of the experiment are in Table 6.

The results show that the best scores are achieved without connecting curves. The time needed does decrease as connectivity increases. High connectivity directly correlates with a lower runtime.

The score increase caused by connecting ECEs can be explained by the fact that each curve we add to connect two ECEs has zero degrees of freedom concerning its placement. (See Section 5.5 .) If this placement causes ambiguous sections in our puzzle they can never be fixed, because the curve never gets moved. See Figure 12 for an example: only the curves marked red in Figure 12a can be moved during our optimization algorithm. All of the visual ambiguity that is already present in Figure 12b cannot possibly be removed.

The time decrease that occurs when connecting curves is nice, but it is important to note that the time needed to process the puzzle per optimizable curve in it increases as we connect more curves. In other words: inputs that have fewer curves can be processed faster than inputs that have more curves which we then connect to get the same number of curves.

A possible way to prevent the score from increasing when connecting ECEs may be to use a more complex curve to connect them. If we connect them with a Bézier curve of higher order (or some other complex curve), it gets degrees of freedom and can be moved around if it creates ambiguity in our puzzle. This should be studied further, see Section 8. It should also be studied if scores can be improved by putting more effort into determining which curves to connect, rather than picking the ECEs at random. If we iteratively connect the curves that have the lowest impact on visual ambiguity, we may limit the occurence of areas with a lot of immovable visual ambiguity.

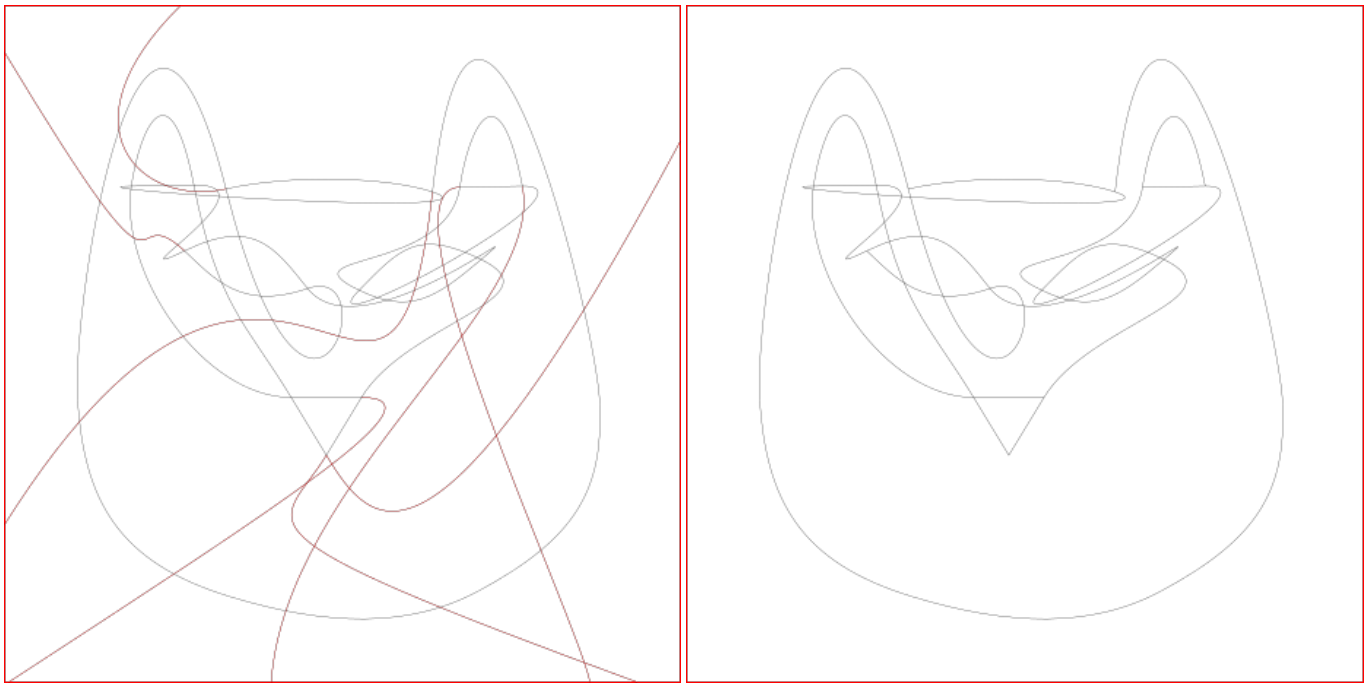
6.5 Testing the effect of including a secondary optimization round

To test the effects of including the pairwise optimization step in our algorithm as shown in Section 5.7, we generated two sets of curved nonograms using the settings in Table 7. A starting temperature of 72 lets our SA run for 140 iterations. A starting temperature of 21 lets it run for 100. So both settings make it so our algorithm does 140 optimization steps total. For a number of input images, there was already a solution found of score 0 during the first 100 iterations of SA. Because this does not make for a useful comparison, we have excluded these inputs from the test set for this experiment.

The results of the experiment are in Table 8. In this table, the data on the curved nonograms that were generated with pairwise optimization are split. “SA-Score” shows the score after 100 iterations of Simulated Annealing but before Pairwise Optimization has taken place, “Final Score” shows the score after Pairwise Optimization. The time needed to generate the curved nonogram is split likewise. The progression of the scores of the puzzles as the algorithm iterates can be seen in Appendix A. The red lines show the effects of 140 iterations of Simulated Annealing.The algorithm starts with $T = 72$ at the first iteration and ends when $T < 1$, with T being multiplied by 0.97 each iteration. The blue lines show our combined algorithm: the first 100 iterations are done by Simulated Annealing starting at $T = 21$ at the first iteration with $T < 1$ at

	Score				Time			
InputName	0	0.25	0.5	0.75	0	0.25	0.5	0.75
penguin	0,014273	0	1,014617	0,318093	1842	1382	1978	1565
butterfly	0	0	0	0,000026	2846	1557	1380	1505
coffee_cup	0,016778	0,053078	0,070708	0,242361	1775	1577	1286	1177
lamb	0,031632	0,027717	0,007402	0,0976	4039	2134	2357	1340
monitor	0	0	0,060398	0,121641	1671	1583	1456	907
rocket	0	0	0,199395	0,063284	2961	2118	1831	1574
mask	0,037112	0,038631	0,299532	0,505217	4871	2394	3121	3072
airplane	0,088418	0,201988	0,140205	0,316368	3199	2372	2459	3234
hut	0,138924	0,124257	0,151273	0,322466	2792	2150	2425	2312
catface	0,188053	0,5631	0,691809	1,467871	3818	3579	3150	3054
church	0,481681	0,991716	0,230514	0,244792	3956	3876	2682	1724
space	0,54112	0,634889	0,52567	0,693041	7195	6620	4460	4823
pumpkin	0,479699	0,439557	0,433597	0,673097	6566	5132	5287	3092
Average	0,155207	0,236533	0,29424	0,389681	3656	2806	2606	2260

Table 6: Comparing four different levels of curve connectivity. Times are in seconds



(a) Optimizable curves marked in red

(b) Without optimizable curves

Figure 12: Curves interconnecting ECEs can create visual ambiguity that cannot be removed

Parameter	No Second Optimization Round	Second Optimization Round
w_o	0	0
$connect\_rate$	0%	0%
$pair\_iterations$	0	40
T_0	72	21
$self\_intersections$	Allowed	Allowed
$exception\_attempts$	N/A	N/A

Table 7: The settings of our Experiment testing pairwise optimization. Other values are given in Table 2

	SA-Only	Pairwise		SA-Only	Pairwise	
InputName	Score	SA-Score	Final Score	Time	SA-time	Final Time
airplane	0,031796	0,059697	0,040538	1133	829	1048
catface	0,507891	0,470861	0,350880	1630	1250	1691
church	0,362922	0,726080	0,682776	1590	1154	1738
hut	0,000546	0,009733	0,015108	874	564	724
mask	0,070977	0,165301	0,240669	1324	834	1108
pumpkin	0,677896	0,748872	0,794321	2435	1783	2455
space	0,602420	0,364679	0,313025	2818	1465	1933
Average	0,322064	0,363603	0,348188	1686	1125	1528

Table 8: Comparing the results of Pairwise Optimization and pure Simulated Annealing. Times are in seconds

iteration 100. Then there are 40 iterations of Pairwise Optimization.

The results show that replacing 40 iterations of Simulated Annealing with 40 iterations of Pairwise Optimization increases the average score by about 8%. However, the time needed to generate the curved nonogram is decreased by 10%. Looking at the score progressions, we can see that the SA-iterations have a far greater effect on the score than the pairwise iterations. This was to be expected as SA makes greater changes and the Pairwise optimization is more focused on refining the puzzle after Simulated Annealing has already removed most of the visual ambiguities. However, when generating curved nonograms using only Simulated Annealing, the average score at the end of 140 iterations is 42% lower than after 100 iterations. The 40 iterations of pairwise optimization only manage to reduce the average score by 4.5%. So it seems to follow that until our Simulated Annealing has fully or almost fully reached its optimum, it is better to spend more iterations on Simulated Annealing than on Pairwise Optimization. If time is not an issue then including some iterations of Pairwise Optimization after Simulated Annealing could help reduce the score by some amount to truly get the best puzzle possible with this algorithm.

Also included in Appendix A are graphs showing the score progression during another experiment done with pairwise optimization. In this second experiment, instead of having a set maximum distance d_{max} we let the maximum distance a point can be moved during a pairwise iteration be equal to its distance to the intersection point. This way, it becomes a lot less likely that the changed curve pieces to intersect another curve making changes more likely to improve the score. From the score progressions we can see that the effect of SA is still much greater on decreasing the score, with almost no pairwise iterations making an improvement, so the conclusion on pairwise optimization remains the same.

Important to note on these experiments is that because the test set is smaller, the test results for these experiments are less reliable than those of other experiments. The variance in the random components of our algorithm has a larger influence on the average result compared to the rest of this thesis. See Section 6.1.2.

Parameter	Without Obfuscation	With Obfuscation
w_o	0	1
$connect\_rate$	0%	0%
$pair\_iterations$	0	0
T_0	72	72
$self\_intersections$	Allowed	Allowed
$exception\_attempts$	N/A	N/A

Table 9: The settings of our Experiment testing the Obfuscation Penalty. Other values are given in Table 2

	No Obfuscation Penalty				With Obfuscation Penalty			
Input	Score$-\rho_o$	ρ_o	Score	Time	Score$-\rho_o$	ρ_o	Score	Time
penguin	0	0,0448	0,0448	1815	0,0386	0,0110	0,0496	2542
butterfly	0	0,0457	0,0457	1609	0	0,0411	0,0411	2070
coffee_cup	0	0,0201	0,0201	3703	0	0,0210	0,0210	1960
lamb	0	0,0129	0,0129	2330	0,0078	0,0151	0,0229	2230
monitor	0	0,4448	0,4448	1675	0,0303	0,4433	0,4737	1509
rocket	0	0,0032	0,0032	2405	0	0,0023	0,0023	2090
mask	0,0549	0,0694	0,1243	3019	0,0019	0,0665	0,0684	3362
airplane	0,0378	0,1683	0,2061	3664	0,1009	0,1467	0,2476	2838
hut	0	0,3000	0,3000	1852	0,0732	0,3381	0,4113	1753
catface	0,4723	0,0669	0,5392	3914	0,4409	0,0566	0,4976	3646
church	0,2653	0,3353	0,6006	7708	0,6235	0,2332	0,8568	5206
space	0,5462	0,0159	0,5621	6464	0,4531	0,0130	0,4661	6489
pumpkin	0,4965	0,1533	0,6498	5700	0,5585	0,1509	0,7094	5983
Average	0,1441	0,1293	0,2734	3528	0,1791	0,1184	0,2975	3206

Table 10: Comparing the results of either including or excluding an Obfuscation penalty. Times are in seconds

6.6 Trying to obfuscate the solution image

To test the effects of including an obfuscation penalty, we generated two sets of curved nonograms using the settings in Table 9. The results are given in Table 10. One set was scored without an obfuscation penalty, where the ρ_o-column shows what the obfuscation penalty of the final puzzle is if we would calculate it, without the obfuscation penalty being used do determine scores during optimization.

The results show that including the obfuscation penalty during optimization increases the other penalties that are applied to the puzzle. Including it does help to lower the obfuscation penalty of the final puzzle, although this reduction is smaller than the increase in the other penalties.

Upon visual inspection, the inclusion of the obfuscation penalty does not seem to help that much for actually hiding the image. The solution images can still easily be seen. Just as with the visual inspection in Section 6.2, this is highly subjective. Another complicating factor is that the person doing the visual inspection already knows what the solution image is going to look like, so finding it is easier. To accurately decide if the obfuscation penalty helps to hide the solution image a user study will have to be performed with a large number of participants who have not seen the solution images before. Conducting this user study is not part of this thesis, but it is important future work. See Section 8.

7 Conclusions

In this thesis, we introduced a new algorithm for automatically generating curved nonograms that builds on the work done by De Jong. Aside from a core algorithm based on Simulated Annealing, we also introduced several optional features that can be used to adjust the algorithm: Self-intersections can be prevented, a percentage of input curves to be connected to each other instead of to the boundary can be set, a secondary optimization can be done to change the trajectories of the extension curves after the main topologial aspects of the puzzle have been settled, and the increasing degree to which the input image is hidden in the puzzle can be included as an optimization goal.

Because visual ambiguity is a nebulous concept based on human perception that computations can only approach, a user study would need to be done before strong conclusions can be drawn about how the algorithm's output compares to that of De Jong. The new algorithm does seem to be faster (1 hour versus 13 hours), although this all depends on the number of iterations that are chosen for both algorithms as well as the implementation. Our new algorithm replaces the dilation penalty, which was a bottleneck in De Jong's algorithm, with a penalty based on compactness that can be calculated very efficiently, but the two algorithms might require a vastly different number of iterations to reach their optimum. Extensively researching the time and quality difference is outside of the scope of this thesis however.

We can conclude our core algorithm to be successful in being able to generate curved nonograms that pass a simple visual inspection in an acceptable timeframe. Of the optional features we introduced, being able to prevent self-intersections seems to be very useful. Aside from making the puzzles more easily understood for beginners, it has on average a positive effect on the score.

We were not able to improve the average score of our puzzles by interconnecting curves, although this seems to be the fault of our restrictions that each curve must be a cubic Bézier curve and that all connected curves must be C^1-continuous. Either by connecting curves using higher order Bézier curves with more degrees of freedom or by allowing the curves to break C^1-continuity we might be able to lessen the effect of unmovable curves being stuck in unfavorable positions. If we can prevent this then connecting curves seems to be a good idea for decreasing our score, as the total number of ECEs will decrease.

Our secondary optimization round did improve the scores of our puzzle, but was outclassed by just running our core algorithm for more iterations. If time is less of a factor and the best quality puzzle is desired, it does seem to be a valid tool for further improving the score after our core optimization has sufficiently approached its optimum.

We were able to create and implement a scoring component for taking the obfuscation of the solution image into account, but we cannot as of yet conclude on its effectivity.

8 Future Work

Further study on the algorithm A number of aspects of the algorithm proposed in this thesis can be studied in more depth. A more in-depth comparison can be made between our new algorithm and that of De Jong. For the best comparison, both algorithms would need to run for the same amount of time. The different outputs would then have to be evaluated with a user study, to see which outputs are deemed by people to be less ambiguous.
Aside from comparisons with De Jong, a user study can also be done to test the effectiveness of including the obfuscation penalty, where test participants indicate if the solution image can be less easily seen in curved nonograms that used it. Such a user study could also be used to test the other, more intangible aspects of our generated curved nonograms such as how well the evaluation function corresponds with actual perceived visual ambiguity, how fun people think solving curved nonograms is, how difficult it is to solved curved nonograms compared to regular ones, how much

self-intersections contribute to perceived difficulty, et cetera.

Another important way the algorithm can be studied further is simply by running all of the experiments several more times. Due to time constraints most combinations of settings and inputs have been run only once so far. Ideally, each experiment setting is run several times more so that the variance caused by the random elements of our algorithm can be better accounted for.

Because our proposed algorithm ignores information on solvability, it still needs to be studied into what solvability category the generated curved nonograms fall although De Jong's result of 80/80 generated puzzles falling in the simple category indicates that this is likely the case for most of our nonograms as well.

It could also be studied how the different optional features of our algorithm interact when they are combined. Our implementation of connecting curves suffers from a lack of freedom of placement. This problem could possibly be mitigated during pairwise optimization. It is also unclear how connecting curves impacts the obfuscation of the solution image.

Furthermore, while some of the parameters for our algorithm were experimentally determined, some values were chosen for other reasons, such as corresponding to De Jong's settings, or just chosen by intuition. Our algorithm could probably be improved by experimentally setting the value for each parameter.

Study into other algorithms for generating curved nonograms Aside from further studying the proposed algorithm, other algorithms or improvements to the algorithm can also be studied. A restriction that causes some problems is the demand that all connected Béziers must be C^1-continuous with each other. It can be studied how much deviation there can be from this standard without it becoming detrimental to the aesthetic qualities of the generated puzzles. By removing the restriction, we increase the degrees of freedom of each Bézier by letting them move their second control point more freely. Alternatively, a switch can be made to Bézier curves of a higher order so they can have more complex trajectories. An even more radically different algorithm could be developed by stopping with the focus on Bézier curves altogether, although the current algorithms are doing well enough that such a radical shift may not be necessary to achieve a puzzle quality that is suitable for commercial applications.

Alternate versions of the optional features of our algorithm could also be studied. Our current implementation chooses ECEs at random when interconnecting curves, but it could be studied if iteratively connecting the ECEs whose connection has the least impact on the score would yield better results. A version of pairwise optimization could also be developed where the intersection point can be moved along one of the two curves.

Other types of optimization could also be used for algorithms, such as a force-directed model comparable to those used in graph visualization [5], where visually ambiguous elements exert a force on the control points of our Bézier curves to move them to a more advantageous position.

Our optimization can also be tweaked by looking at different penalties to be included in our score function. Aside from Compactness, we can also determine the regularity of puzzle faces by looking at their *fatness* [3], for example.

The current algorithm also does not allow any difference between the input image and the solution image of the puzzle. By allowing minor changes between the images the freedom of curve placements can be further increased or visual ambiguities inherent to the input curves removed, although this needs to be balanced against the decreased aesthetic value of the solution image.

A lot more research can also be done into good metrics for the obfuscation of the solution image. A metric could for example be developed based on the difference in perceived symmetry between the input curves and the curves that are added later. Because hiding an image requires tricking the human eye, ideally some multi-disciplinary research can be done into good ways of hiding the image, where knowledge of geometric algorithms is combined with knowledge on human image processing.

References

[1] K.J. Batenburg, S. Henstra, W.A. Kosters, W.J. Palenstijn, Constructing simple nonograms of varying difficulty, *Pure Mathematics and Applications (Pu. MA)*, 20: 115, 2009.

[2] K.J. Batenburg, W.A. Kosters, Solving nonograms by combining relaxations., *Pattern Recognition*, 42.8: 1672-1683, 2009.

[3] M. de Berg, A.F. van der Stappen, J. Vleugels, M.J. Katz, Realistic input models for geometric algorithms, *Algorithmica*, 34.1: 81-97, 2002.

[4] E. Bribiesca, Measuring 2-D shape compactness using the contact perimeter, *Computers & Mathematics with Applications* 33.11: 1-9, 1997.

[5] T. M. Fruchterman, E.M. Reingold, Graph drawing by force-directed placement, *Software: Practice and Experience*, 21.11: 1129-1164, 1991.

[6] M. Hunt, C. Pong, G. Tucker, Difficulty-driven sudoku puzzle generation, *Journal of Undergraduate Mathematics and Its Applications (UMAPJournal)*, p. 343, 2007.

[7] T. de Jong, *The concept and automatic generation of the curved nonogram puzzle*, Master Thesis ICA-4123689, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 2016.

[8] M. Jünger, P. Mutzel, Maximum planar subgraphs and nice embeddings: Practical layout tools, *Algorithmica* 16.1: 33-59, 1996.

[9] M. Kamermans, *A Primer on Bézier Curves*, `https://pomax.github.io/bezierinfo/`, 2017.

[10] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi., Optimization by simulated annealing, *Science*, 220.4598: 671-680, 1983.

[11] L.J. Latecki, R. Lakämper, Shape similarity measure based on correspondence of visual parts, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.10: 1185-1190, 2000.

[12] M. Löffler, M. Kaiser, T. van Kapel, G. Klappe, M. van Kreveld, F. Staals, The connect-the-dots family of puzzles: design and automatic generation, *ACM Transactions on Graphics (TOG)*, 33.4: 72, 2014.

[13] E.G. Ortiz-García, S. Salcedo-Sanz, J.M. Leiva-Murillo, A.M. Pérez-Bellido, J.A. Portilla-Figueras, Automated generation and visualization of picture-logic puzzles, *Computers & Graphics*, 31.5: 750-760, 2007.

[14] R. Parment, *Generation of sloped nonograms*, Master Thesis ICA-3493865, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 2015.

[15] H.C. Purchase, Metrics for graph drawing aesthetics, *Journal of Visual Languages & Computing*, 13.5: 501-516, 2002.

[16] H.C. Purchase, R.F. Cohen, M. James, Validating graph drawing aesthetics, *Graph Drawing* pp. 435-446, Springer, 1996.

[17] J.N. van Rijn, *Playing Games: The complexity of Klondike, Mahjong, Nonograms and Animal Chess*, Master Thesis Internal Report 2012-01, LIACS, Leiden University, Leiden, The Netherlands, 2012.

[18] S. Salcedo-Sanz, E.G. Ortiz-García, A.M. Pérez-Bellido, J.A. Portilla-Figueras, X. Yao, Solving Japanese puzzles with heuristics., *IEEE Symposium on Computational Intelligence and Games.*, 224-231, 2007.

[19] A. M. Smith, E. Butler, Z. Popovic, Quantifying over play: Constraining undesirable solutions in puzzle design, *Foundations of Digital Games conference 2013*, pp. 221-228, 2013.

[20] N. Ueda, T. Nagao, *NP-completeness results for NONOGRAM via parsimonious reductions*, Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan, 1996.

[21] K. Xu, C. Rooney, P. Passmore, D.H. Ham, P.H. Nguyen, A user study on curved edges in graph visualization, *IEEE Transactions on Visualization and Computer Graphics*, 18.12: 2449-2456, 2012.

[22] C.-H. Yu, H.-L. Lee, L.-H. Chen, An efficient algorithm for solving nonograms., *Applied Intelligence*, 35.1: 18-31, 2011.

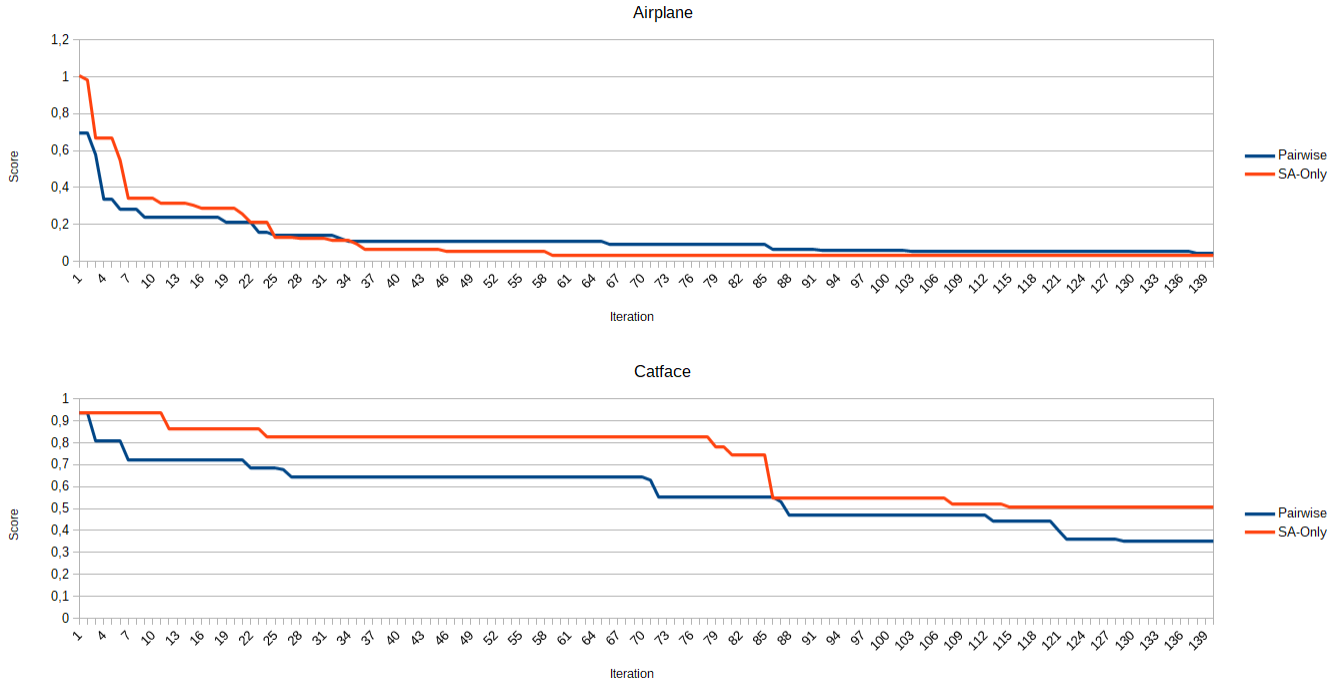
[23] H. Zabrodsky, S. Peleg, D. Avnir, Symmetry as a continuous feature, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.12: 1154-1166, 1995.

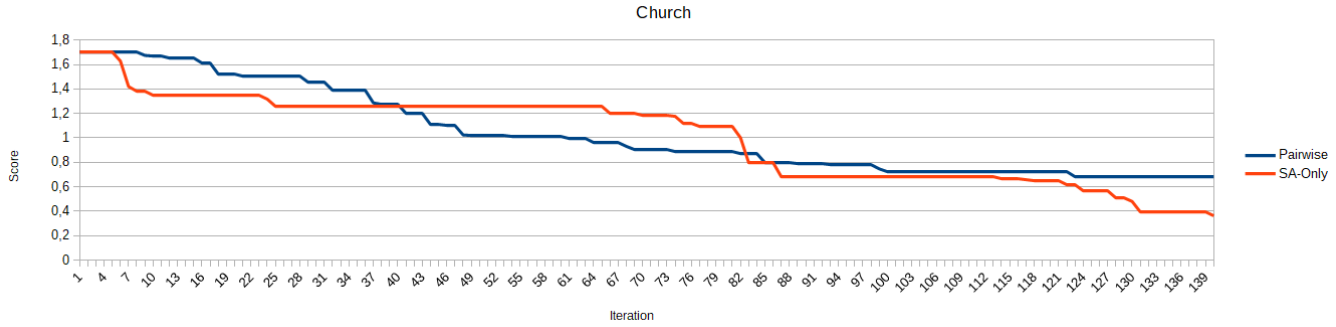
[24] `https://en.wikipedia.org/wiki/B%C3%A9zier_curve`

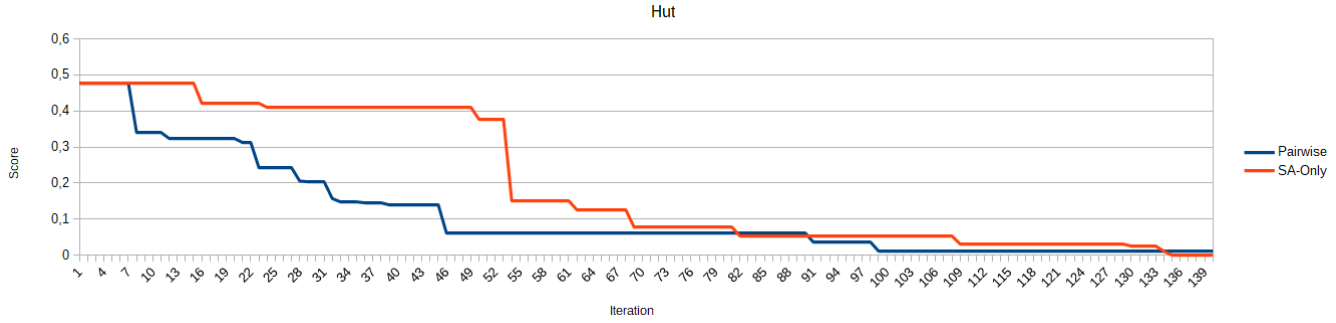
[25] `https://en.wikipedia.org/wiki/Smoothness`

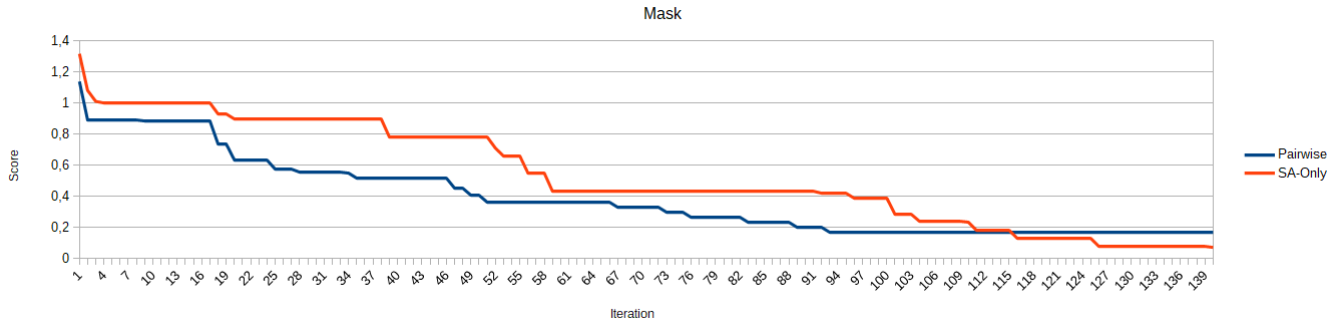
A Score Progression of (Pairwise) Optimization

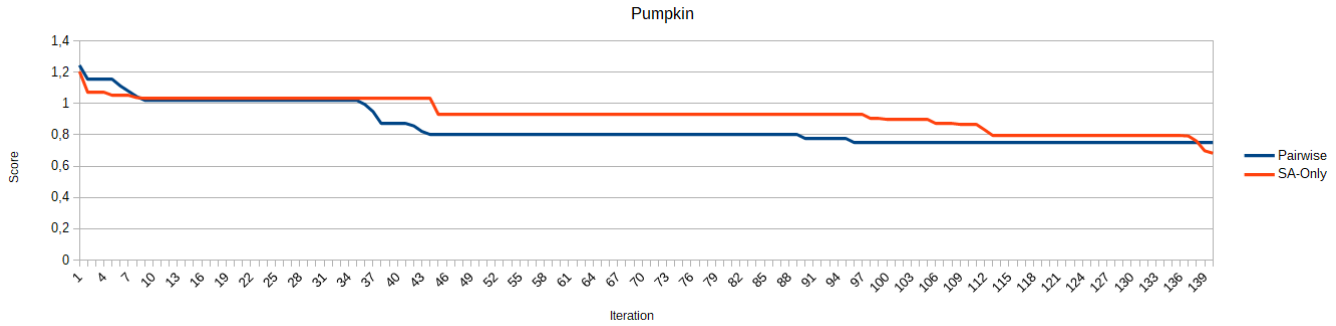
Progression of scores with a fixed d_{max} of 100.

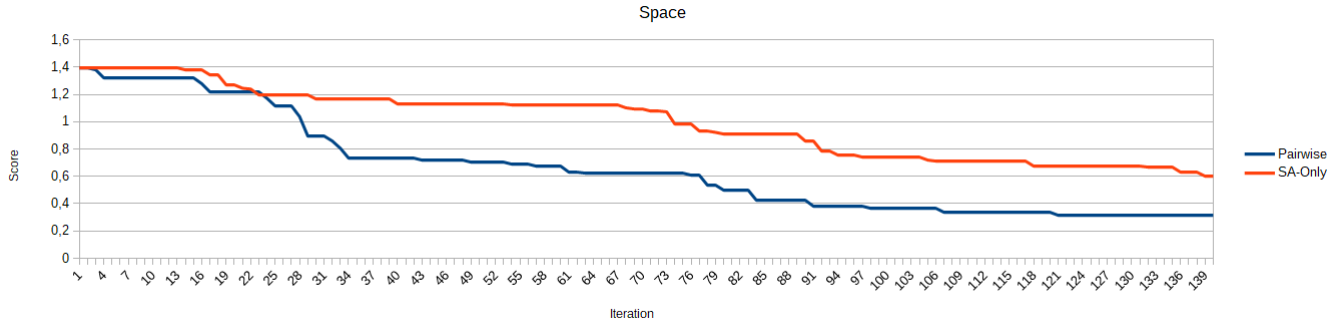



Church
Score
Iteration
Pairwise
SA-Only

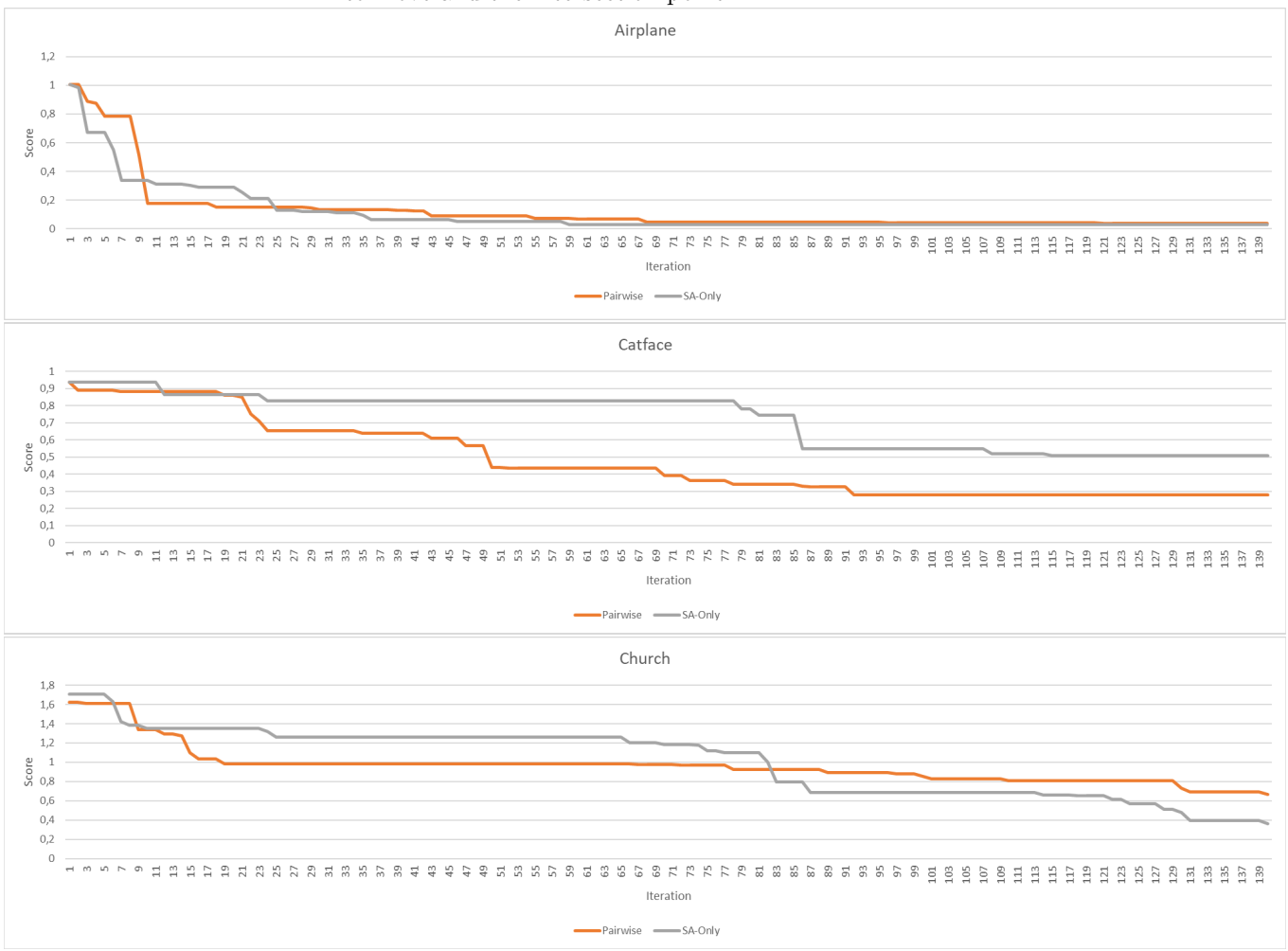

Hut
Score
Iteration
Pairwise
SA-Only

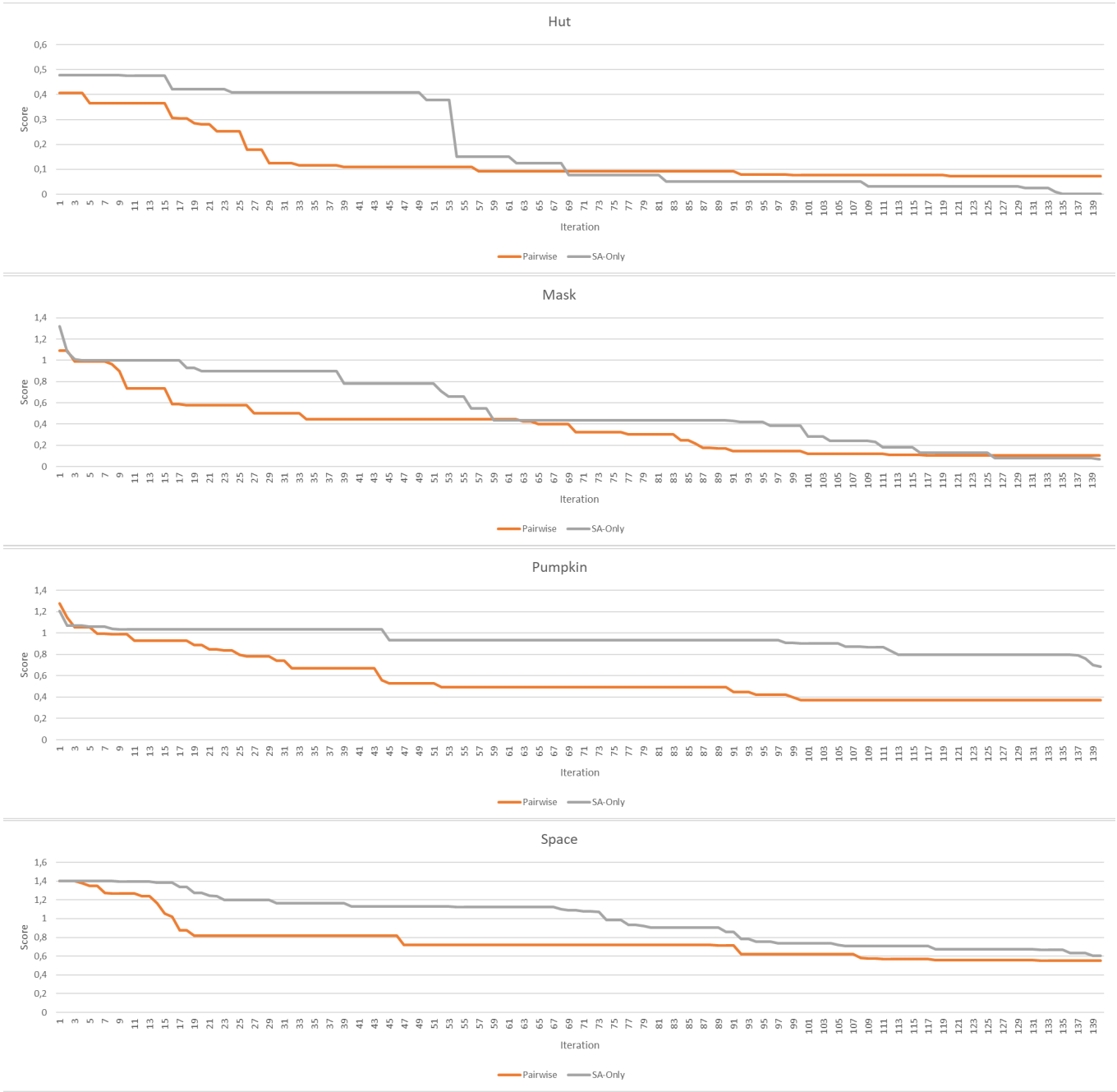

Mask
Score
Iteration
Pairwise
SA-Only


Pumpkin
Score
Iteration
Pairwise
SA-Only



Progression of scores with d_{max} changing to be equal to the distance between the control point to move and the intersection point.





B Inputs and Outputs

This section contains the input images that were tested, as well as some of the outputs that were generated by the algorithm working with these images.

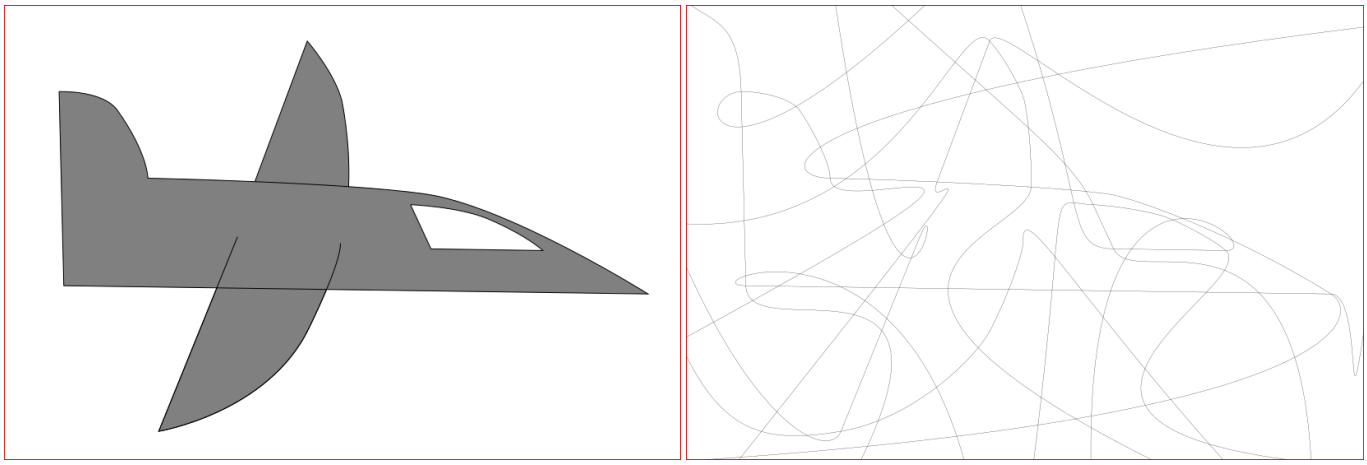


Figure 13: The “airplane” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only

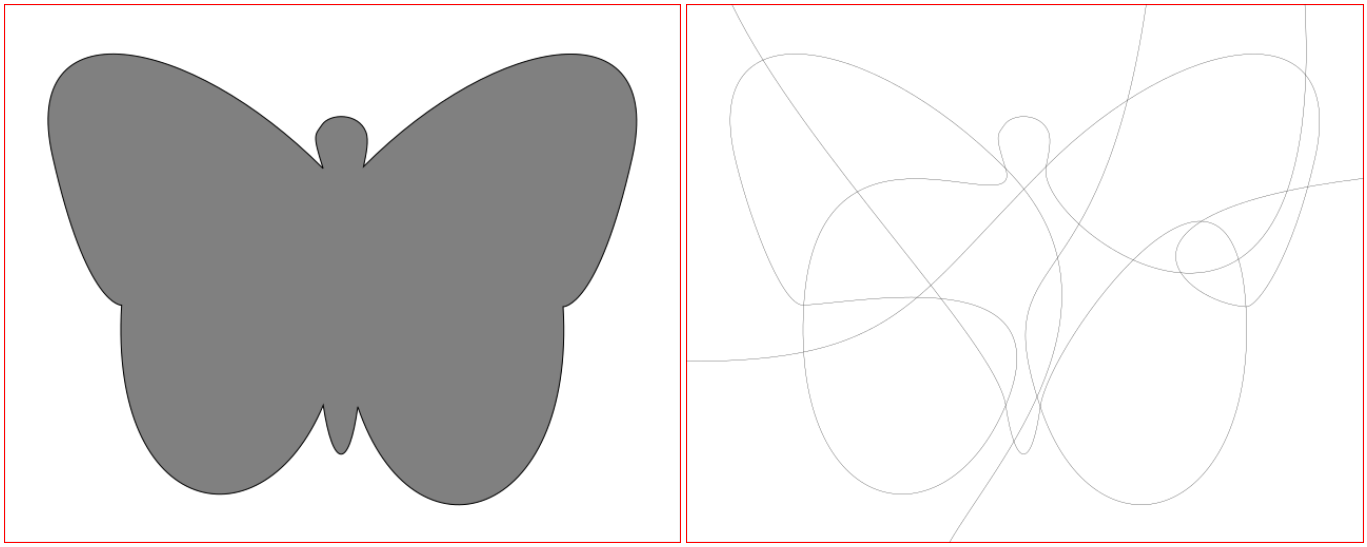


Figure 14: The “butterfly” input file. Puzzle generated during the experiment described in Section 6.4 with $connect\_rate = 50\%$

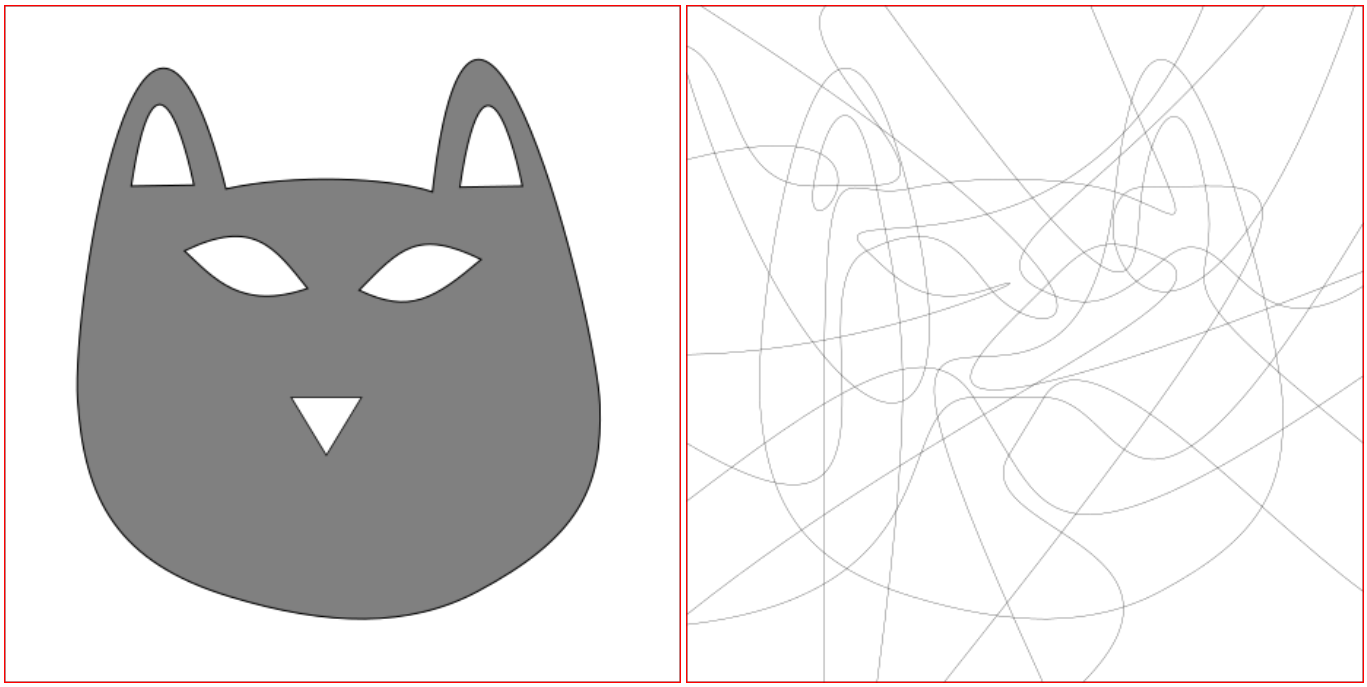


Figure 15: The “catface” input file. Puzzle generated during the experiment described in Section 6.6 with $w_o = 1$

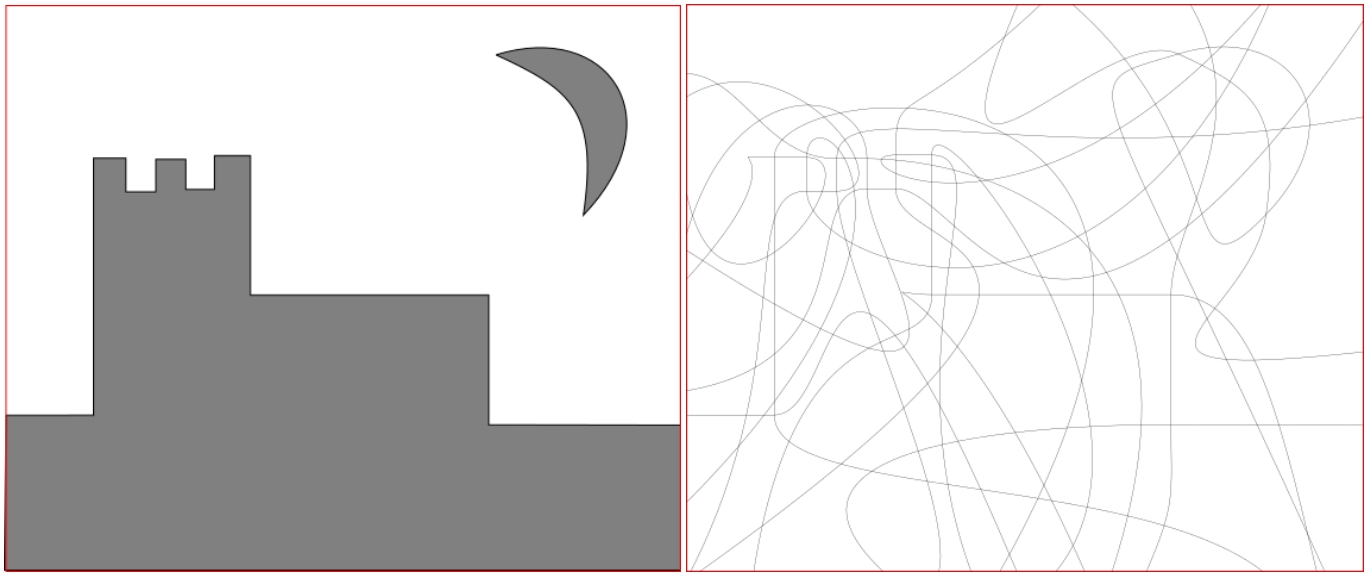


Figure 16: The “church” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only

Figure 17: The "coffee_cup" input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only

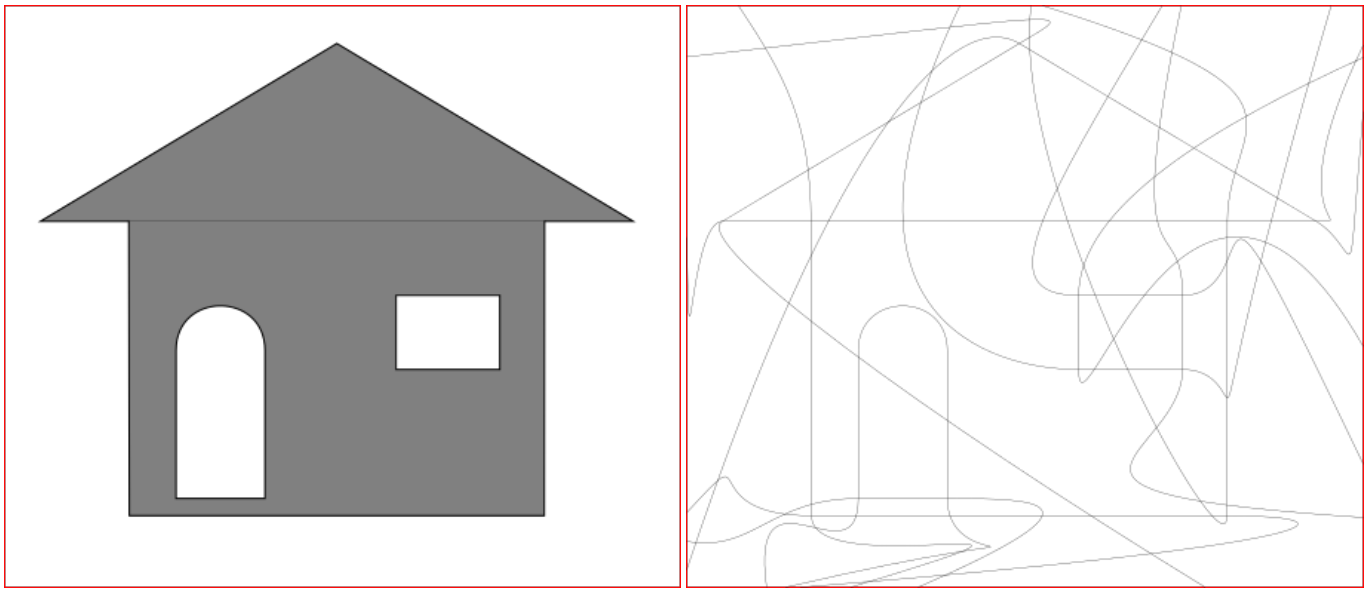


Figure 18: The "hut" input file. Puzzle generated during the experiment described in Section 6.3 with self-intersections disallowed

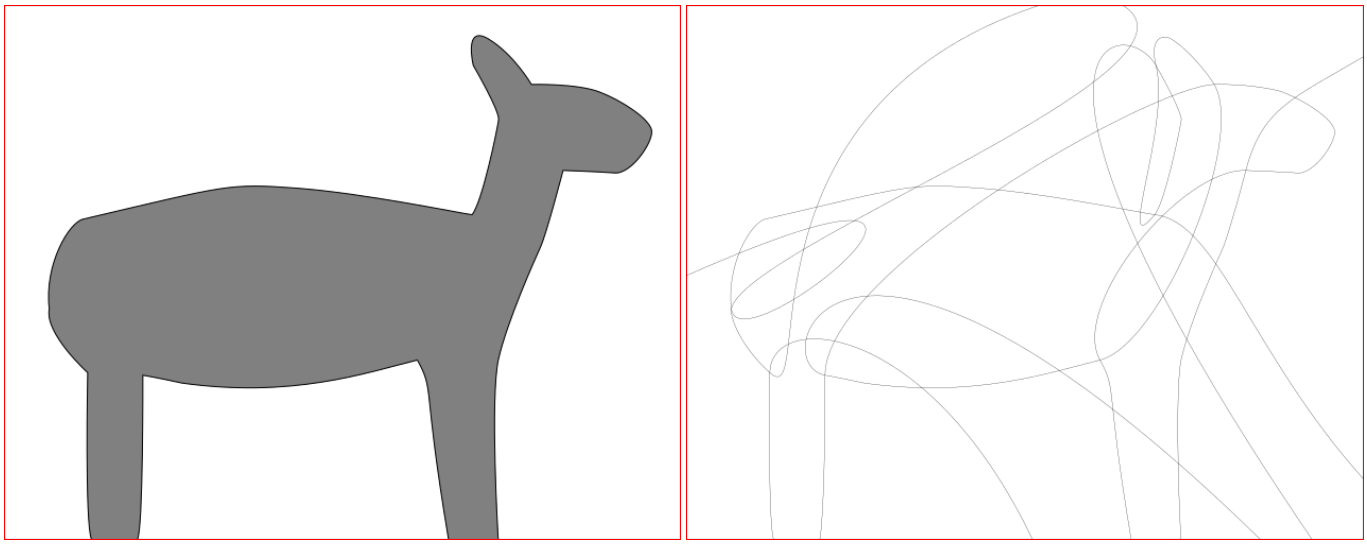


Figure 19: The "lamb" input file. Puzzle generated during the experiment described in Section 6.4 with $connect\_rate = 50\%$

Figure 20: The "mask" input file. Puzzle generated during the experiment described in Section 6.6, with $w_o = 1$.

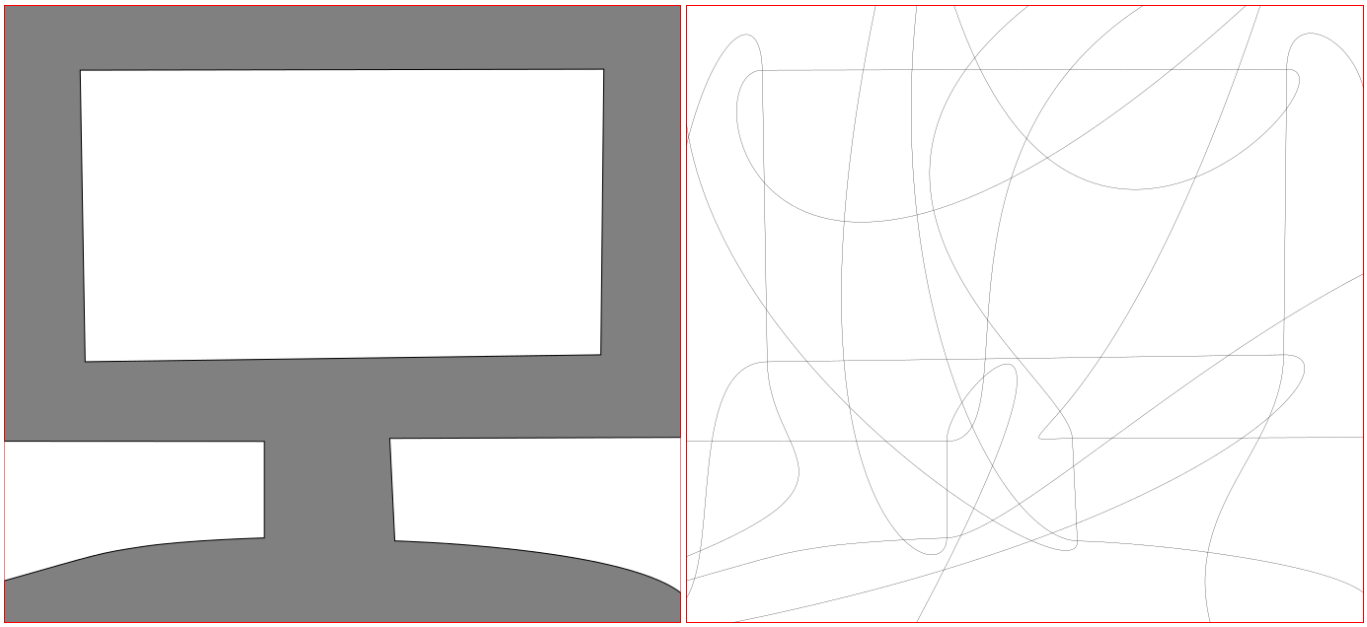


Figure 21: The “monitor” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only (this result has not been included in Table 8 because the score reached 0 before 100 iterations had passed)

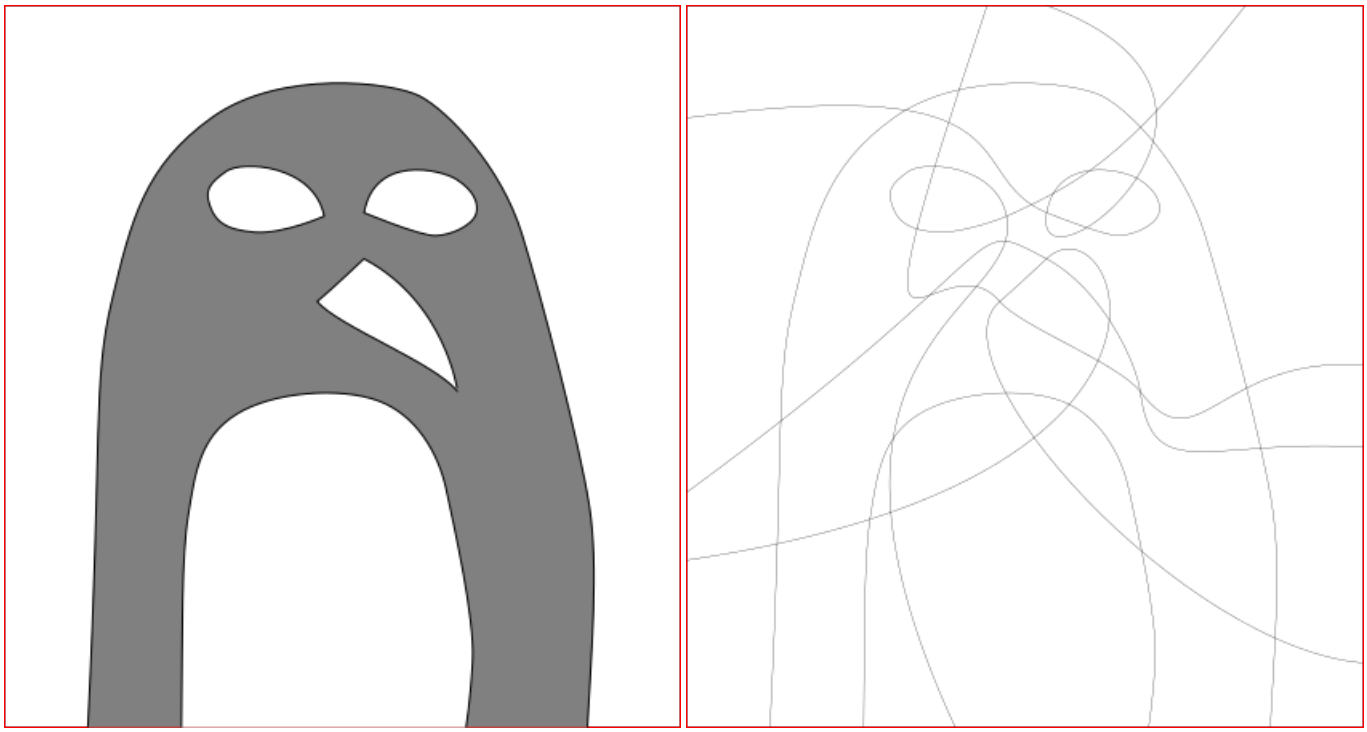


Figure 22: The “penguin” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only (this result has not been included in Table 8 because the score reached 0 before 100 iterations had passed)

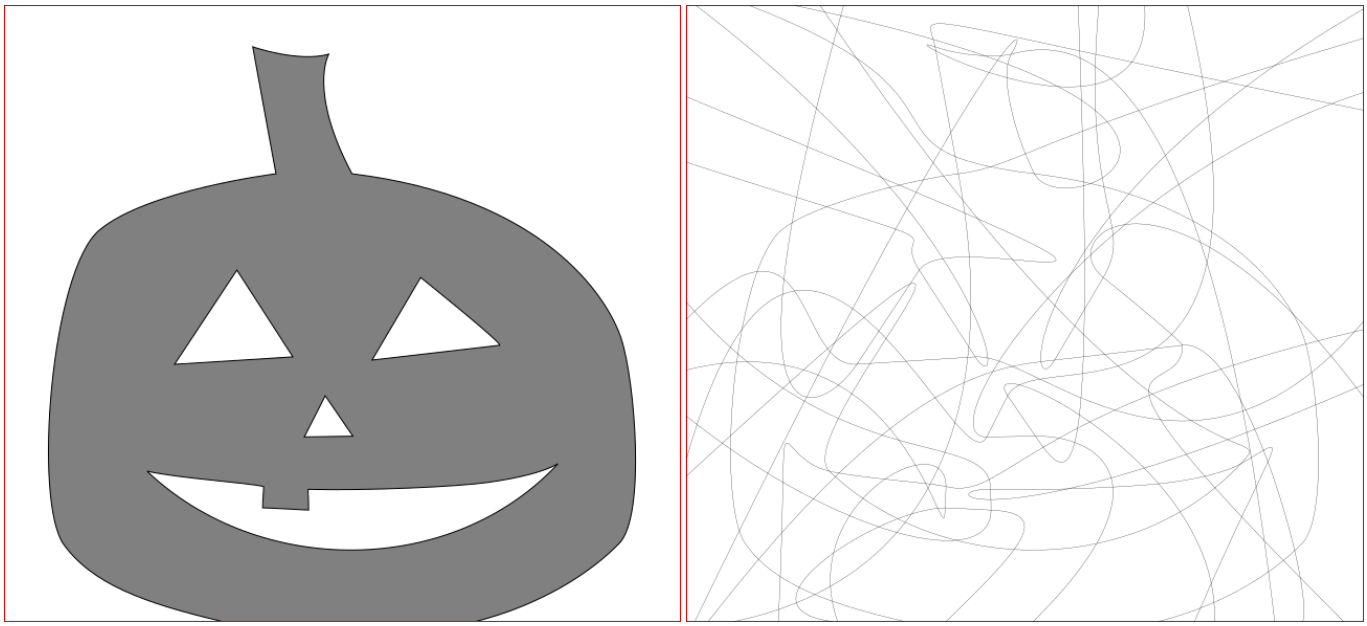


Figure 23: The “pumpkin” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only

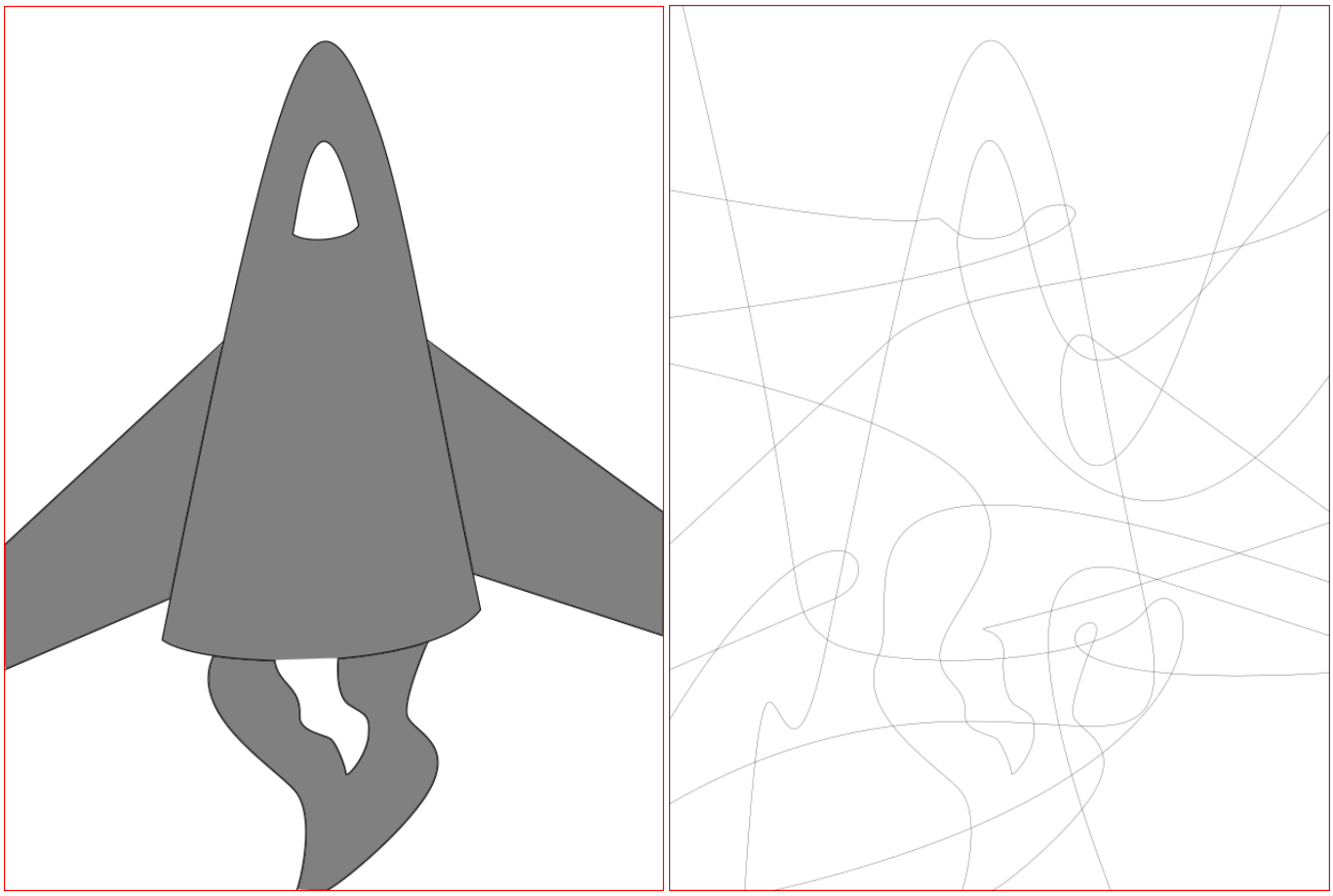


Figure 24: The “rocket” input file. Puzzle generated during the experiment described in Section 6.5 optimized with SA only

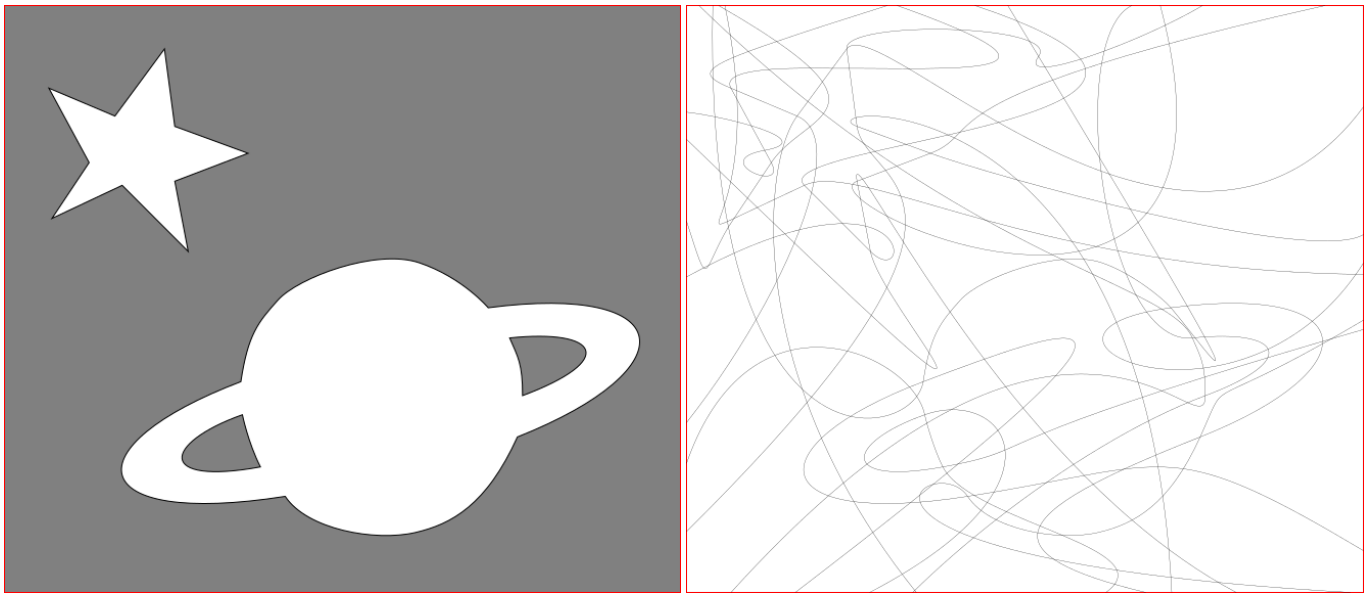


Figure 25: The "space" input file. Puzzle generated during the experiment described in Section 6.5 optimized with Pairwise Optimization

C Notes on CurvednonogramsGenerator.exe

The program used by this thesis to conduct experiments can be used by calling CurvednonogramsGenerator.exe with the appropriate command line arguments. This transforms one input file into one curved nonogram. If multiple nonograms are wanted a batch file must be written.

C.1 Input

Input must initially consist of a .svg file. In this file needs to be at least one group (< g> tag) with at least one rectangle (< rect> tag.) The bounding rectangle must be the first rectangle in the first group. All other rectangles are ignored.

The rest of the file can contain any number of SVG paths (< path> tag) and any number of groups. But each path must be in a group to be processed.
The paths are converted into cubic bezier curves. The following path commands are supported: "c,C,h,H,l,L,m,M,v,V,z,Z". The "translate" and "matrix" transforms are supported, but others are not (They can be emulated by choosing the right matrix parameters.)

The program generates .xml files with arrangements that can also be loaded back in. However, the only .xml files that should be used are the ones generated by the program itself. .svg files generated by the problem can be loaded back in, but this is not recommended. To load an old arrangement back in the .xml should be loaded.

C.2 Command Line Arguments

CurvednonogramsGenerator.exe can be called with the command line arguments listed in Table 11. In this table, arguments of the form -arg can be included without further parameters. arg=[s] means a string must also be supplied. arg=[d,d] means two doubles separated by commas must be supplied etc. A typical call to CurvednonogramsGerator looks like this:

```
CurvednonogramsGenerator.exe file="inputs/input1.svg" outdir="outputs/" outfile="1.svg"
savefile="1.xml" -af -am -me weights="1000,3,3,7,0"
```

With this call we process the file "input1.svg". By using -af we can put the entire file path in the file argument instead of having to split it between file and indir arguments. We use -am to not have to think about what filemode to use. Because we are creating a new arrangement from an .svg we want to break the curves and create new ECEs so the -me argument is included. If we would want to load in an existing .xml file, we would need to leave out -me. It is important to specify the weights because otherwise all parts of the score function would be turned off by default. This call generates a curved nonogram of the file using only simulated annealing with default parameters.

Argument	Description	Default value
required arguments		
indir=[s]	sets the directory to look in for the input file	-
file=[s]	sets the input file	-
Scoring arguments		
thresholds=[d,d,d,d]	sets $d_{vert}, \alpha_{min}, A_{min}, C_{min}$ (See Section 4)	11, 20, 55, 0.025
weights=[d,d,d,d,d]	sets $w_{vert}, w_{angle}, w_{size}, w_{shape}, w_o$ (See Section 4)	0,0,0,0,0
-wone	include to set all score weights to 1	-
Optional arguments		
-af	include to auto-split input file and dir. names, replaces indir	-
outdir=[s]	sets the directory to place the output file	indir
outfile=[s]	sets the name of the output file	"output.svg"
savefile=[s]	sets the .xml file to save the arrangement to	"saved.xml"
mode={0,1}	sets filetype to load. 0 for .svg, 1 for .xml	0
-am	include to choose load mode based on file extension	-
seed=[i]	sets the seed for the RNG.	random seed
-me	include to generate new ECEs in the arrangement	-
-npsi	include to disallow self-intersections in puzzle curves	-
cratio=[d(0.0-1.0)]	sets percentage of ECEs to interconnect. Requires -me	0
temp=[d]	sets T_0 for SA. Set to less than 1 to disable SA	72
coolrate=[d(0.0-1.0)]	sets the cooling rate of the temperature during SA	0.97
pairits=[i]	sets number of iterations for pairwise optimization	0
rattempts=[i]	sets max. #attempts for randomizing a curve. 0 = no limit	10
sirattempts=[i]	sets the max.#attempts to randomize a curve before-allowing self-intersections if -npsi is included. 0 = no limit	10
ras=[i]	sets the max. # attempts to randomize a curve before-restarting the entire program. 0 = no limit	0
-dre	include to prohibit extending ECEs	-

Table 11: Available command line arguments for CurvednonogramsGenerator.exe

C.3 Output

The program generates output in the folder specified with the outdir argument (or the indir by default.) In the following descriptions, OUTFILE and SAVEFILE should be read as the whatever values are specified in the command line arguments.

The program generates OUTFILE, which is a .svg file showing the generated curved nonogram. It also generates coloredOUTFILE, which is the curved nonogram but with each face given a random color. markedOUTFILE shows the nonogram while marking the vertices and faces that add a penalty to the score. Vertex penalties are marked purple, angle penalties orange, area penalties red, shape penalties yellow. SAVEFILE is a file containing the entire arrangement including a full description of the full DCEL. It can be used to load the arrangement back in if we want the program to operate on it further. afterConnects.xml is a saved copy of the arrangement before any optimization is done or ECEs are extended, but after ECEs are connected to each other if applicable. If ECEs are extended, the program saves the arrangement as duringECEsolve.xml

and updates it after each ECE that is extended, in case it is needed for debugging. Lastly timesSAVEFILE is generated, which is a log of the amount of time the algorithm has taken. In this file, all randomization steps are timed, and the different phases of the algorithm as well, which also include all of the overhead that is not taken into account when timing individual randomization steps.