Labeling Curved Nonograms

Eva Timmer

June 26, 2023

Abstract

A curved nonogram is a variation of a classic nonogram, which is a type of logic pen-and-paper puzzle. While a classic nonogram takes the shape of a uniform grid, a curved nonogram is an arrangement of curves. The puzzler colors in certain cells or faces based on provided clues. A correctly solved nonogram usually reveals a recognizable image. In a curved nonogram, clues are placed in labels around the boundary of the puzzle, in this thesis we implement three algorithms that automatically place these labels. Each algorithm tries to find a valid labeling, where each label is assigned a port and leader length, and none of the placed labels overlap with each other.

The first algorithm uses 2-SAT to find a labeling by assigning each label to one of two ports, but requires a given fixed leader length for each label. The second algorithm uses dynamic programming to assign a leader length to each label, but requires a given fixed port assignment. The final algorithm uses MaxSAT to assign both a leader length and port assignment.

Each algorithm is tested on several different input puzzles. The Dynamic Programming and MaxSAT algorithms are also tested on randomly generated sets of labels that need to be placed on a single edge.

1 Introduction

Nonograms, also known as Paint by Numbers or Japanese puzzles, are a popular type of logic pen-and-paper puzzle. The puzzler colors in cells of a grid, based on the provided clues. The clues are a sequence of numbers for each column and row, denoting the amount of consecutive squares that need to be colored. Figure 1 shows an example of a simple nonogram puzzle, where the solution shows an image of a smiley face.

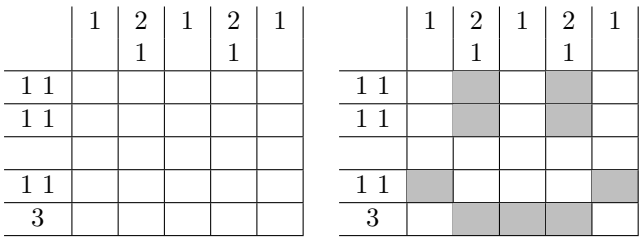

Figure 1: An example of a simple nonogram puzzle.

A variation of the nonogram is the curved nonogram, introduced by De Jong [8], where instead of a uniform grid, the puzzle consists of a set of nonogram lines which are arbitrary curves enclosed by a boundary, see Figure 2. Each nonogram line lies completely inside the boundary, and starts and ends on the boundary. In addition, no more than two lines may intersect in the same point, to avoid ambiguity.
Instead of a sequence of numbers for each row and column, there is a sequence of numbers for each side of the line. The numbers denote which of the faces that touch the line with an edge should be colored in on that side. These clues are shown as labels and can be placed on either end of each line.

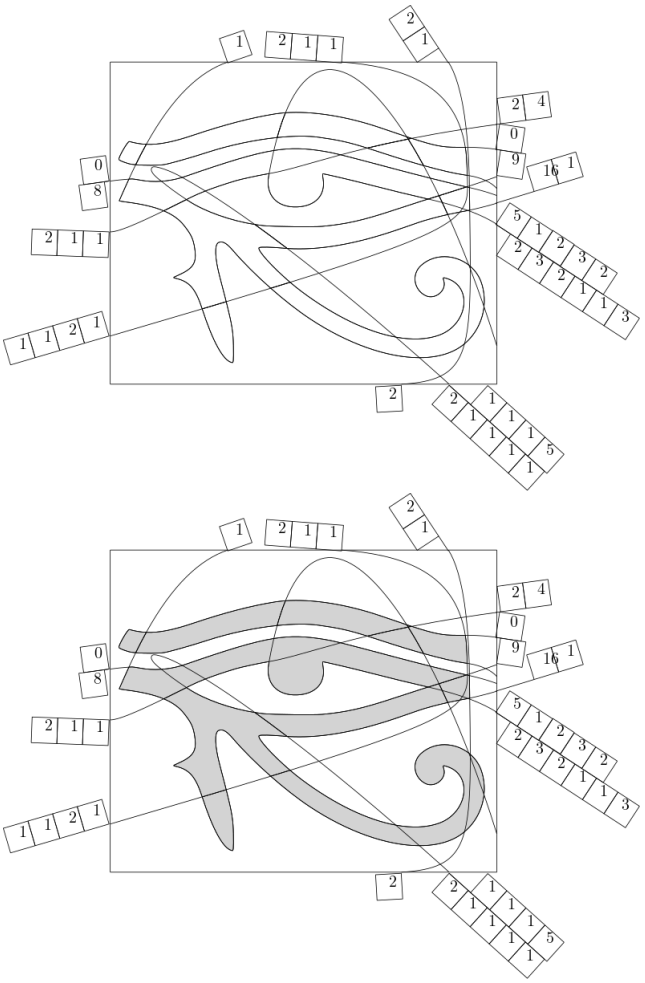

Figure 2: An example of a curved nonogram puzzle

Curved nonograms allow us to use more detailed images as a basis for creating a nonogram. However, doing this by hand takes a long time. Van de Kerkhof [14] introduced an algorithm to automatically generate a curved nonogram from a given image, that is aesthetically pleasing and unambiguous. However, this algorithm only focuses on the puzzle itself, not the placement of the clue labels. Placing the labels in a naive manner could result in labels overlapping with each other, making them unreadable. This may be avoided by placing the labels on the other endpoint of the nonogram line, or by extending the nonogram line past the boundary.

Placing labels by hand is very tedious, so in this thesis we will look at several algorithms outlined by Klute et al. [9] that provide a solution to the problem of automatic label placement, in a way that is aesthetically pleasing and unambiguous. To this end we define some criteria for the placement of labels:

1. Labels should not overlap other labels.

2. The label leaders should extend in the same direction as the curve.

3. The labels should be placed as close to the boundary as possible.

2 Related Work

2.1 Nonograms

There have been several studies on generating and solving nonograms. Simple nonograms are nonograms that have a unique solution and can be solved by looking at only one row or column at a time. There is a classification of difficulty of simple nonograms [2].

Different methods have been proposed for solving nonograms using heuristics [11], depth-first search [16] and combining relaxations [3]. Solving nonograms has been proven to be NP-hard [13].

There have been studies on the generation of classic nonograms as well as on newer variations of nonograms, like colored nonograms, curved nonograms [14] and sloped nonograms. With the latter two types, the problem of labeling becomes considerably more complicated.

2.2 Labeling

The problem of labeling nonograms is related to the problem of boundary labeling. Algorithms have been proposed to solve the problem of boundary labeling. There are algorithms that optimize different aspects, like minimizing bends or minimizing length [5].

There have been defined several leader types, like straight (s), orthogonal (o), perpendicular (p), diagonal (d) and variations thereof (op, opo) [4]. Nonogram labeling is like boundary labeling with straight leaders that are restricted in their direction, but can have variable length and are restricted to two possible ports on either side of the line. With nonogram labeling whatever is going on inside the boundary cannot be changed to suit the labeling.

2.3 Puzzle/label aesthetics

Vollick et al. [15] have defined several properties of good label layouts and state that preventing overlap between labels is incredibly important for a readable layout. Another property that could be applied to nonogram labeling is that they should be as close to the boundary as possible. They also provide an energy function that can be used to measure the quality of a label configuration.

3 Definitions

3.1 Nonograms

Classic nonograms are grid puzzles with a uniform grid, see Figure 1. Each row and column has an associated clue, which is a sequence of numbers. Each number represents an uninterrupted sequence of cells in that row or column that should be colored in. Together these clues describe how the whole grid is to be colored in. A simple nonogram has exactly one unique solution.

The clues for a classic nonogram are generally placed above or to the left of the corresponding column or row.

Curved nonograms do not not adhere to a uniform grid, but consist of a set of arbitrary curves, see Figure 2. In this case each nonogram line has two descriptions, one for the adjacent cells above the line, and one for the adjacent cells below the line. A curved nonogram may also include straight nonogram lines.

3.2 Nonogram labeling

A set of clues for one line/row/column we call a *label*. Each label can be placed on the boundary in one of two ways. For a classic nonogram, the label can be placed on either side of each row or column. For curved nonograms, each label can be placed on either intersection of the line with the boundary. These points on the boundary are called *ports*. In the case of classic nonograms, the labels can be placed at the start or end of a row/column and will never overlap, because of the uniform grid. However, for the other two types, this is not usually the case.

Unless the line is perpendicular to the boundary, it needs to be extended in order to avoid overlapping the label with the boundary. This part of the line that is extended past the boundary we call the *leader*. The leader can be further extended to avoid overlapping between labels. A label that is assigned to one of two ports of a nonogram line, and is assigned a leader extension length is called a *placed label*. A *labeling* is a set of placed labels, a labeling that includes one placed label for each nonogram line, and where no two labels overlap we call a *valid labeling*. A curved nonogram may have several valid labelings. The valid labeling for which the total extension length is minimized, we call a *minimum-length labeling*.

3.3 Problem description

We describe a nonogram labeling problem as:

1. a simple polygon B which represents the nonogram frame;
2. a set of *unplaced labels* L, each $\ell \in L$ defining a pair (p_ℓ, q_ℓ) of *ports* and a list of integer *clues*.

The output of an algorithm that solves this problem should be a valid *minimum-length labeling* of L, such that:

1. each $\ell \in L$ is assigned to either p_ℓ or q_ℓ;
2. each $\ell \in L$ is assigned an extension length e;
3. no two labels $\ell \in L$ and $k \in L$ intersect;
4. there exists no other labeling for L with a smaller total extension length.

4 Algorithms

4.1 Overview

There are three main parts of the nonogram labeling problem. First, each label needs to be assigned to a port. Second, the extension length for each leader in a minimum-length labeling needs to be determined. Finally, post processing takes care of edge-cases like labels around boundary corners.

In this thesis we will implement three algorithms and compare the results. The first algorithm uses a 2-SAT formula to find a port assignment for labels that have a fixed leader length. The second uses dynamic programming to determine the length of the leaders, given a fixed port assignment. We have implemented two versions of this algorithm, one that can be used for inputs where all leaders have a slope of ± 1 and is optimized for such inputs, and one that is not optimized but can be used for inputs with leaders of any slope. Finally, we introduce a third algorithm that uses MaxSAT to both assign each label to a port, and determine the length of each leader. See Table 1 for an overview of these algorithms. In the next sections, we will explain the theory behind these algorithms in more detail.

Algorithm	Section	Restrictions
2-SAT	4.2	No solution for puzzles that require extendable leaders
Dynamic Programming (optimized for slopes of ± 1)	4.3	Restricted to inputs with only leaders with a slope of ± 1
Dynamic Programming	4.3	Can try only a limited number of port assignments due to time restrictions
MaxSAT	4.4	Requires the MaxHs program

Table 1: Algorithms

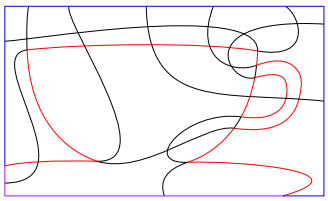

Figure 3: Example input

4.2 Fixed leader length (2-SAT)

The first algorithm uses 2-SAT to find a valid labeling, without extensible leaders. That is, each label ℓ assigned to port p has exactly one possible extension length. This is the length of the leader when the label touches the boundary, but does not overlap with it.

Theorem 4.1. *Given a nonogram labeling instance, we can decide in polynomial time if a valid labeling with fixed leader length exists.*

Proof. We can create a 2-SAT formula φ to describe this problem. For each nonogram line $l \in L$ with labels a and b above and below the line respectively, we define two variables x_l^a and x_l^b where $x_l^a = 1 (x_l^a = 0)$ if the label above l is assigned to port $p_l(q_l)$ and $x_l^b = 1 (x_l^b = 0)$ if the label above l is assigned to port $p_l(q_l)$.
First we need to ensure that the resulting label assignment does not include overlapping labels. For each pair of labels that overlap when assigned to their respective ports we ad a clause to exclude this combination from the solution. For instance if the label above line l overlaps with the label below line l' when l is assigned to port p_l and l' is assigned to port $q_{l'}$, we add a clause $\neg x_l^a \vee x_{l'}^b$.
In order to obtain a labeling that is also balanced, we can add two extra clauses $x_l^a \vee x_l^b$ and $\neg x_l^a \vee \neg x_l^b$ for each $l \in L$.
Solving the 2-SAT instance takes linear time in the size of φ [1], and the number of clauses is linear in the number of nonogram lines and the number of overlapping label pairs.
Assuming that a satisfying assignment of φ is found, we can construct a valid labeling as follows. For each nonogram line $l \in L$ with labels a and b we place label a at port p if $x_l^a = 1$ and at port p if $x_l^a = 0$, and place label b at port p if $x_l^b = 1$ or at port q if $x_l^b = 0$. □

4.3 Extensible Leaders, Fixed port assignment

The second algorithm uses dynamic programming to find a valid labeling with extensible leaders. The algorithm consists of two parts, in the first part, one or more label assignments are chosen. In the second part, the algorithm finds a minimum length labeling (if such a labeling exists). We assume the boundary B is a rectangle. We also assume all leaders have slopes of ± 1 and that the labels have a width of 1.

Choosing assignments As is the case with the algorithm in Section 4.2, in a valid labeling each pair of labels belonging to each nonogram line $l \in L$ needs to be assigned to one of two ports, p_l and q_l. Again we can use a 2-SAT formula φ to find such a labeling. We create two variables x_l^a and x_l^b where $x_l^a = 1 (x_l^a = 0)$ if the label above l is assigned to port $p_l(q_l)$ and $x_l^b = 1 (x_l^b = 0)$ if the label above l is assigned to port $p_l(q_l)$.
Using only these two clauses, given n labels, there are 2^n such assignments. However, we can limit the number of assignments by adding additional clauses. We can add clauses that exclude certain assignments that do not have a valid labeling, irrespective of the extension length. There are situations where a pair of labels belonging to two nonogram lines will always overlap, regardless of the assigned extension lengths. For instance in Figure 4 the label ℓ above nonogram line l and ℓ' above nonogram line l' will always overlap, regardless of the assigned extension length. In such cases we add a clause $x_l^a \vee x_{l'}^a$ to φ.

To determine additional possible assignments, we add a clause that excludes previously found assignments from the solution. So if the solution for the previous iteration is $x_1 = \{0, 1\}, \ldots, x_n = \{0, 1\}$ then the clause $\neg x_1 \vee l \ldots, \vee \neg x_n$

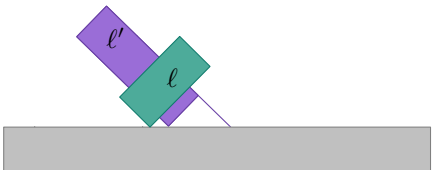

Figure 4: Labels ℓ and ℓ' will always overlap, irrespective of the assigned extension length.

is added to φ. This is repeated until no new solution can be found, or the maximum amount of assignments is reached.

Observation 4.2. *In a minimum length labeling with extensible leaders, where all leaders have slopes of ± 1, each label ℓ is blocked from being closer to the boundary either by the boundary itself, in which case the extension length is either 0 or 1 (depending on which side is labeled), or it is blocked by another label ℓ'.*

Lemma 4.3. *Given a nonogram labeling instance I with n nonogram lines with slopes of ± 1 and a fixed port assignment, we can reduce the set of possible extension lengths for each label to $O(n^2)$ relevant extension lengths. If a solution for I exists, we can find a minimum length labeling by assigning each label one of these extension lengths. The first and last label on an edge of the boundary have $O(n)$ relevant extension lengths.*

Proof. Without loss of generality we assume that the label ℓ on line l with slope $+1$ is assigned to a port on the (horizontal) top edge of the boundary B. Label ℓ is blocked by a label ℓ' on line l' which is assigned to a port on the same edge. We can define two distinct cases in this situation, see Figure 6:

1. In the first case ℓ' belongs to a line l' of a different slope (-1), this can only occur if the label is on the right side of l. The extension length of ℓ is then equal to the distance between the port p and the intersection point of the lines l and l', $+1$ if ℓ' is flipped to the right side of l'. There are $O(n)$ such extension points for ℓ.

2. In the second case ℓ' belongs to a line l' of the same slope. This can only occur if ℓ is facing towards line l'. Here ℓ' can itself be blocked by another label ℓ'', forming a chain of labels.

   (a) If all labels have the same slope and are labeled on the same side, then there is one possible extension length of ℓ that is equal to the label lengths of all labels in the chain, $+1 + x$ (See Figure 5 if the labels are on the right side, or $-x$ if the labels are on the left side. Where x has one possible value that depends on the distance between ℓ and the label in the chain that is closest to the boundary.

   (b) If all labels have the same slope but some of them are labeled on the other side, then the extension length of ℓ is at least equal to the sum of the lengths of the labels that are labeled on the same side (possibly $+1 \pm x$). However, any prefix of the labels that are labeled on the other side can add their label length to the total extension length of ℓ. There are $O(n)$ such possible extension lengths.

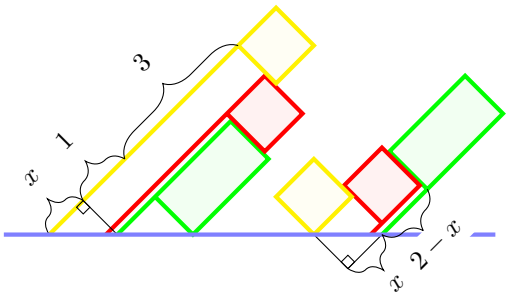

Figure 5: Extension length for a chain of labels.

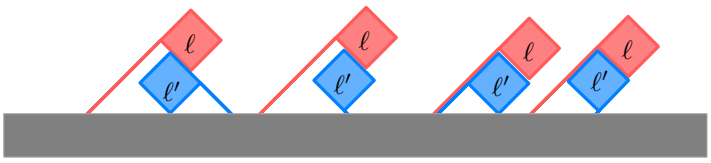

Figure 6: Cases where label ℓ is blocked by another label ℓ'.

(c) Thirdly, a chain can be interrupted by one or more orthogonal labels. Such chains can be prefixed by a (b) chain and there are $O(n)$ possible interrupting labels, therefore there are $O(n^2)$ possible extension lengths.

Corners Finally, we consider the case where the port of a label ℓ is on the far right side of the top edge of B and blocked by a label ℓ' on the right edge of B.

1. If the slope is $+1$ we have again three subcases:
   - ℓ is blocked by a parallel (a) chain. There is one possible extension length for ℓ.
   - ℓ is blocked by a parallel (b) chain. There are $O(n)$ possible extension lengths for ℓ
   - If ℓ' is labeled on the left side of a line l' that has slope of -1, there is one possible extension length.
2. If the slope of l is -1 and ℓ is on the right side it can only be blocked by a label ℓ' on the left side of a line l' with slope -1. There is one such extension length.

This implies that the amount of extension lengths that need to be considered for a label ℓ is at most $O(n^2)$. On this basis we can create a dynamic programming algorithm that solves a sub-instance of the minimum label length problem defined by one of the edges of B. □

4.3.1 Algorithm

Lemma 4.4. *Given a nonogram labeling instance I with a rectangular boundary B, an edge e of the boundary that is crossed by n nonogram lines with slopes of ± 1, a fixed side assignment for each label, and a fixed extension length for the left- and rightmost label on e. We can decide in $O(n^9)$ time if a minimum-length labeling exists.*

Proof. We define a sub-instance of the minimum length labeling problem $\mathcal{I}$ as the labeling of a single edge, with a fixed length for the two outer labels. Without loss of generality we assume e is the top edge of B and the nonogram lines that cross this edge are indexed $l_1, \ldots, l_n$ in order from left to right, and that each nonogram line has a slope of ± 1. The corresponding ports on edge e are $p_1, \ldots, p_n$. For line l_1 and l_n the extension lengths δ_1 and δ_n are defined in the input. Any solution where the extension lengths of l_1 and l_n are not δ_1 and δ_n are considered invalid.
We also add two dummy lines l_0 with slope -1 and port p_0 one unit to the left of the left endpoint of e and l_{n+1} with slope $+1$ and port p_{n+1} one unit to the right of the right endpoint of e. We assign them a length δ_{max} that is greater than the maximum possible extension length of all lines on e.
For a line l_i let Δ_i be the set of relevant extension lengths. For l_1 and l_n we set $\Delta_1 = \{\delta_1\}$ and $\Delta_n = \{\delta_n\}$, respectively.

We define further sub-instances of $\mathcal{I}$ by two boundary lines l_i and l_j $(i < j)$ with assigned extension lengths a and b. Given n lines and $O(n^2)$ relevant extension lengths for each line, there are $O(n^6)$ such sub-instances.
In each sub-instance, a line l_k with a port p_k $(i < k < j)$ is restricted by the region bounded by l_i and l_j and the horizontal line through the top-most point of the shortest of the two labels ℓ_i and ℓ_j.

Let $T[i, j, a, b]$ be the minimum total extension length of the instance defined by line l_i with extension length a and line l_j with extension length b, if a valid labeling exists, otherwise it is ∞.

To solve a sub-instance recursively we use dynamic programming and optimize over all lines l_k $(i < k < j)$:
$T[i, j, a, b] = \min\{T[i, k, a, c] + T[k, j, c, b] - c \mid i < k < j, c \in \Delta_k, c \leq \min\{a, b\}, l_k \text{valid}\}$

The recursion ends when $i = j - 1$, where $T[i, j, a, b] = a + b$. Each entry considers the $O(n^3)$ combinations of a line k and an extension length $c \in \Delta_k$ and verifies that the label ℓ_k with assigned extension length c is valid; it remains within the bounded region as shown in Figure 7.
The minimum length of $\mathcal{I}$ is obtained from cell $T[0, n + 1, \delta_{max}, \delta_{max}]$, which takes $O(n^9)$ time to compute. □

Finally, we compose a minimum-length solution for the full puzzle by combining the solutions of the sub-instances for each of the four sides of the frame B.

Theorem 4.5. *Given a nonogram labeling instance I with n lines with slopes of ± 1 and a fixed side assignment for each label, we can decide in $O(n^{11})$ time, if a valid minimum-length labeling exists. We can also find a minimum-length labeling in $O(n^{11})$ time.*

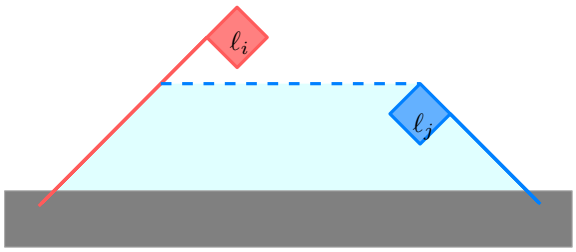

Figure 7: The bounded region defined by l_i, l_j and the horizontal line through the topmost point of the shortest of the two labels ℓ_i and ℓ_j

Proof. Given that computing a single instance $\mathcal{I}$ with a fixed length for the outer labels takes $O(n^9)$, and there are $O(n^2)$ possible combinations of lengths for the outer labels, all instances on all four edges take $O(n^{11})$ time to solve. This results in $O(n^2)$ sub-solutions for each edge, therefore there are $O(n^8)$ combinations of sub-solutions in total.
For each of these combinations it remains to validate if this results in a valid labeling for the complete puzzle, by checking if the outermost labels for each are free of overlaps. This can be done in constant time. From all valid solutions, we choose the one with the minimum total extension length. □

4.4 No fixed assignment, extensible leaders

Now we have two algorithms, one for assigning labels to ports without using extensible leaders (Section 4.2), and one for assigning an extension length to labels with a fixed port assignment (Section 4.3). However, in many cases, it is impossible to find a valid minimum-length labeling without extending leaders. On the flip side, the chance of the dynamic programming algorithm finding a solution depends heavily on the used port assignment.

In this section we will present an algorithm that both determines the port assignment and the leader length for a minimum-length labeling by formulating it as a MaxSAT problem and using MaxHs to solve it. MaxSAT is a generalization of the SAT problem that uses weighted clauses. There are two types of clauses: soft clauses, which have a finite weight, and hard clauses which have infinite weight. We use hard clauses for making sure each label is assigned to one port, and for avoiding pairs of labels that overlap. We use soft clauses to optimize the extension length.

For each label ℓ with possible port assignments p and q we define variables x_e^p and x_e^q for $e = 1, 2, \ldots, e_{\max}$ (if step size is 1) where e is the extension length and $e_{\max}$ is the maximum extension length to try. We use similar clauses to the 2-SAT algorithm in Section 4.2. For each pair of variables (x, y) for the same label we add a hard clause $\neg x \vee \neg y$ to ensure each label has only one assigned port and extension length as well as a clause $x_1 \vee x_2 \ldots x_{2e-1} \vee x_{2e}$ for each label. For each pair of overlapping labels (p, p') with extension lengths (e, e') another hard clause $\neg x_e^p \vee \neg x_{e'}^{p'}$ is added to exclude this combination from the solution.

For each extension length variable we add a soft clause with weight equal to $e_{\max} - e$.

It is possible to reduce the amount of variables by using a step size larger than 1.

5 Experiments

In this section the setup of the experiments is explained. We will discuss the practical implementation of the algorithms in Section 5.1, the test set in Section 5.2 and parameters used when running the different algorithms in Section 5.3.

5.1 Practical Implementation

5.1.1 Impop pipeline

The algorithms are written in Haskell, making use of the Labeling Pipeline from the Impop interface [10]. The pipeline provides the clues and port locations. We also made extensive use of the HGeometry libraries [12].

Input format The pre-existing pipeline allows us to use ipe-files as input files. These files consist of a set of curves enclosed by a frame. Different elements of the puzzle are marked by colors. The frame is marked blue, the edges of the solution picture are red, or if a curve is both part of the solution and the frame, it is marked purple. All other curves should be black, see Figure 3.

5.1.2 Fixed leader length

For the implementation of the 2-SAT solution, the Haskell library mios is used, which is a minisat based implementation.

5.1.3 Dynamic programming algorithm (optimized for ± 1 slope)

We make use of Haskell's lazy evaluation for implementing the Dynamic Programming (± 1 slope) algorithm described in section 4.3.1. We create an array R, which stores the solution for each sub-instance $T[i, j, a, b]$. Then the solution for an edge with dummy labels ℓ_0 and $\ell_{(n+1)}$ and maximum extension length of M, will be found at $R[\ell_0, \ell_{(n+1)}, M, M]$. We did not implement the handling of overlapping labels from different edges, but instead focused on implementing and comparing the algorithms for one edge at a time.

5.1.4 Dynamic programming algorithm (not optimized)

We have implemented a second dynamic programming algorithm that can be used for puzzles that have leaders with slopes other than ± 1. In this algorithm all extension lengths $0..e_{\max}$ are tried for each label.

5.1.5 MaxHS

We use the data we get from the pipeline to create an input file that follows the wcnf format [7]. This file is then fed into the MaxHS solver created by Davies [6], which will return a solution (if found). The MaxHS solver is a hybrid solver that uses both SAT and Integer Programming to find a MaxSAT solution.

5.2 Test set

We use curved nonograms that are automatically generated from input vector images as input files, as well as handcrafted puzzles. To facilitate testing and comparing the Dynamic Programming and MaxSAT algorithms we also built a generator that generates a set of labels to be placed on a single edge.

5.2.1 Generated Nonograms

We use the generator built by Van de Kerkhof [14] to generate (unlabeled) curved nonogram puzzles from an input image. This program takes a set of curves in the form of an svg-file as input. The program attempts to create a curved nonogram that satisfies certain aesthetic requirements for publishing. It does this by attempting to decrease visual ambiguity caused by, among other things, small faces and intersections that are very close together, and extending the curves onto the boundary.

We use 9 such generated curved nonograms in our test set, see Table 2. The puzzles have between 8 and 34 labels and are of varying sizes. The test set also includes several puzzles generated with the same input image "eye of horus". As the generated curves are random, running the generator on the same input twice results in two different puzzles ("eye of horus 1" and "eye of horus 2"). We also include two smaller scaled versions of "eye of horus 1", the idea being that a smaller puzzle presents a more complex problem as there is a bigger chance that labels will overlap.

Puzzle	Size	Labels	Requires extended labels	Total possible port assignments
fox	623 × 692	20	No	1048576 (1 million)
flowers	320 × 320	34	Yes	17179869184 (17 billion)
hearts	1000 × 787	8	No	256
eye of horus 1	1000 × 832	16	No	65536
eye of horus 1 small	240 × 200	16	Yes	65536
eye of horus 1 smallest	112 × 93	16	Yes	65536
eye of horus 2	1000 × 832	16	No	65536
owl	1000 × 833	34	No	68719476736 (69 billion)
tree	528 × 480	44	Yes	17592186044416 (18 trillion)

Table 2: The characteristics of the different puzzle inputs, including whether or not extended labels are required to find a valid labeling, and the amount of different ways the labels can be assigned to ports.

5.2.2 Handcrafted Nonograms

We also made some simple puzzles by hand, in order to have more control over the amount of nonolines and possible intersections in the puzzle.

5.2.3 Single edge

The nonogram generator may take several hours per puzzle to generate valid input puzzles, and doing this by hand is also very tedious. However, the algorithms only need a boundary B and a set of unplaced labels L, the nonogram lines inside the boundary are not relevant to the problem.

Therefore, in addition to the generated and handcrafted nonogram puzzles, we also use inputs where only a single edge needs to be labeled. This allows us to compare the performance of the Dynamic Programming (Section 4.3) and MaxSAT (Section 4.4) fairly. It is impossible to have a fair comparison on full puzzles, as the Dynamic Programming algorithm requires ports to be assigned first. To obtain this dataset we have created a program that generates a series of unplaced labels based on several parameters. Each port has a direction with an angle of between 1 and 179 degrees with the boundary. The number of different directions ports are able to have and how they are distributed is decided by a_{dir} and d_{dir}, these directions can be either random, or uniformly distributed. For example if $a_{\text{dir}} = 3$ and $d_{\text{dir}} = \text{Uniform}$, then all ports will have a direction that has an angle of either 45, 90 or 135 degrees with the boundary.
The parameters a_{ports} and d_{ports} decide the number of ports and how they are distributed along the edge. We don't want ports on the endpoints of the edge, so for an edge of length 256 the ports can be between 1 and 255 distance from the start of the edge. For example, if $a_{\text{ports}} = 3$ and $d_{\text{ports}} = \text{Uniform}$, the set of possible port positions is $1, 128, 255$.

Finally, r_{clues} defines a range of number of clues for each label, for instance $r_{\text{clues}} = (1, 3)$ means that all labels have 1,2 or 3 clues.

Name	Description	Values
a_{dir}	The amount of different directions ports are able to have	Integer
d_{dir}	The distribution of the chosen directions	Uniform or Random
a_{ports}	The amount of ports that need to be labeled	Integer
d_{ports}	The distribution of the ports along the edge	Uniform or Random
r_{clues}	The range of number of clues for each unplaced label	Tuple of integers

Table 3: Parameters

5.3 Test settings

In this section we detail the settings used for running the experiments. All sizes are in Postscript points, which is the measurement unit used in ipe-files.

Clue size All clue boxes are of size 16x16, due to time constraints, we did not include clue boxes of different sizes in our experiments. Instead, we have included in the test set the same puzzle multiple times, each a different scale.

Maximum extension length We can also limit the amount of extension lengths tried. We use maximum extension lengths $e_{\max}$ of 64 and 100.
This does not apply to the dynamic programming algorithm that is optimized for slopes of ± 1, as $e_{\max}$ is set by the algorithm itself.

Extension length step size We also limit the extension lengths by a step size for some inputs. I.e., if the step size is set to 2 then the set tried extension lengths is $[0, 2, 4 \dots e_{\max}]$. However, since not all relevant extension lengths are tried, this does mean the solution may not actually be the minimum length labeling. It is also possible that a solution is not found at all, even if a solution does exist.
The step size is not used for the dynamic programming algorithm that is optimized for slopes of ± 1, as the algorithm already reduces the possible extension lengths.

Assignments tried This only applies to the Dynamic Programming algorithm in Section 4.3. Each puzzle has 2^n possible assignments. It is not always reasonable to run the Dynamic Programming algorithm for each of these possible assignments. We already limit the number of assignments tried by excluding pairs of label assignments that will always overlap, regardless of the extension length, and including label assignments that will never overlap with another label. We also try assignments in order of how balanced they are. However, it is still necessary to set a cap to the amount of assignments tried. We try at most 500 assignments.
Unfortunately, if we limit the amount of assignments, it is possible that there is a better solution possible with an assignment that hasn't been tried. For

instance, the tree puzzle has 44 labels, this means there are over 17 trillion possible port assignments. Even after reducing the number of assignments as explained previously, the possibility that a valid labeling is found by trying only 500 assignments is incredibly small. And even if a valid labeling was found, it is unlikely that this solution is a minimum length labeling.

6 Results

In this section we will discuss the results of the performed experiments. We first discuss the results of the experiments with each of the puzzle inputs. Secondly we go into detail about the results of the experiments using the generated single edge inputs.

6.1 Puzzle inputs

The time it takes each algorithm to find (or not find) a solution for each of the puzzle inputs can be seen in Table 6. The 2-SAT-algorithm in Section 4.2 is fastest in each case, with a runtime between 102ms (for the hearts puzzle) and 526ms (for the "tree small" puzzle). However, only the triangle, hearts and fox puzzles can be solved using the 2-SAT algorithm.

When extensible leaders are required for a valid labeling, the MaxSAT algorithm in Section 4.4 is always fastest in finding a minimum length labeling. The runtimes of the MaxSAT algorithm for the "eye of horus" puzzles are all around 40 seconds.

The MaxSAT algorithm is also the only algorithm to find a minimum length labeling for almost all inputs. The only input where no complete solution is found with the MaxSAT algorithm is the "eye of horus 1 smallest" puzzle, however, a valid labeling where all labels are placed is impossible in this case. The unoptimized Dynamic Programming algorithm finds a complete solution for 5 of the 11 puzzles, and the 2-SAT algorithm finds a solution for 3 of the 11 puzzles (these are the 3 puzzles that do not require extensible leaders).

6.2 Edge inputs

As explained in Section 5.2.3 it is beneficial to not only experiment with full puzzles, but also try the algorithms on inputs where all ports lie on the same edge of the frame, and the nonogram lines are all straight. As each nonogram label can only be placed on one port, there is no need to assign the labels to a port. This makes the problem much less complex. In addition, as the inputs don't have to be based on real puzzles, we can easily generate hundreds of inputs. This allows us to run experiments on a much larger test set.

6.2.1 Dynamic Programming (optimized for ± 1 slope)

In order to compare the performance of the Dynamic Programming algorithm described in Section 4.3 to the MaxSAT algorithm, we have generated a set of 796 edge inputs with 3 to 10 ports with random positions and a fixed direction

of either $(1, 1)$ or $(-1, 1)$. Of these inputs, 143 have a valid minimum length labeling. For this experiment we used a step size of 1 and a maximum extension length of 100 for the MaxSAT algorithm, to ensure that a minimum-length labeling is always found, if one exists.

The Dynamic Programming algorithm was faster in finding a solution, or determining that no solution is possible for all 796 inputs. The average runtime per number of labels can be seen in Table 4. The table shows that the Dynamic Programming algorithm was significantly faster in all cases, being between 5 and 6 times as fast for all inputs.

Ports	Solved			Unsolved			All	
	MaxSAT	DP	Cases	MaxSAT	DP	Cases	MaxSAT	DP
3	3767	702	68	3331	719	31	3631	708
4	6089	922	39	5313	966	60	5619	949
5	8816	1603	17	8214	1513	81	8318	1529
6	12628	2161	9	11694	2013	91	11778	2026
7	17231	2602	7	15767	2792	93	15870	2778
8	22729	3362	1	20695	3836	99	20715	3831
9	28378	1821	1	26401	4639	99	26421	4611
10	35471	2316	1	32500	6479	99	32529	6437
Total			143			653		

Table 4: Comparison of run time in milliseconds for the MaxSAT and Dynamic Programming (± 1 slope) algorithms on generated single-edge inputs with a maximum extension length of 100 and step size 1. The generated inputs have a clue-box size of 16, random positions and directions of either $(1, 1)$ or $(-1, 1)$.

6.2.2 Dynamic Programming (not optimized)

We used a total of 3900 generated single edge of length 256 inputs with random port positions and directions and ran both the MaxSAT and the Dynamic Programming algorithms on all of these inputs. The average run time it takes the MaxSAT and Dynamic Programming algorithm to find (or not find) a solution for the single edge inputs can be seen in Table 7. The table shows that the Dynamic Programming algorithm is generally faster on inputs with a small amount of ports (less than 6), and the MaxSAT algorithm is generally faster on inputs with a larger amount of ports (more than 6), with MaxSAT being over 1.5 times as fast for the inputs with 10 ports that were solved. However, for the inputs with 10 ports that weren't solved, MaxSAT was only 1.2 times as fast as the Dynamic Programming algorithm. There are some cases where the MaxSAT algorithm was able to find a solution, where the Dynamic Programming algorithm was not.

As is illustrated in Figure 8, the runtime for unsolved inputs is generally less than for solved inputs. Cases with 9 or more ports are not included in this comparison, as these inputs have over 500 possible port assignments, so the Dynamic Programming algorithm wouldn't be able to consider all of them. Therefore it wouldn't be a fair comparison between the runtime of the Dynamic Programming algorithm and MaxSAT algorithm, as the latter does consider all possible port assignments.

There are also a few cases where the Dynamic Programming algorithm is faster than the MaxSAT algorithm, even with 8 ports. There are 14 such cases out of 45 inputs, see Table 5. The table also shows that the amount of possible intersections between labels where the Dynamic Programming algorithm is faster, is higher than the amount of possible intersections where the MaxSAT algorithm is faster.

Ports	Cases DP is faster	Avg. intersections	Cases MS is faster	Avg. intersections
2	250	125.65	0	-
3	201	430.94	0	-
4	121	823.09	0	-
5	121	1305.09	7	710.86
6	41	2198.34	36	1317.17
7	14	3546.50	31	2384.94
8	1	4675.00	18	2484.11
9	0	-	12	3217.50

Table 5: The amount of cases where each algorithm was faster and the average amount of possible intersections in those cases.

7 Conclusions

In this thesis we implemented 4 algorithms to solve the problem of finding a minimum length labeling for a curved nonogram. The algorithm in Section 4.2, which uses 2-SAT, is by far the fastest algorithm, never taking more than a second to find a solution, or to determine that a solution can not be found without using extensible leaders. Unfortunately, due to this algorithm being restricted to using leaders of a fixed length, it is not always possible to find a valid labeling.

The other two algorithms, the Dynamic Programming algorithm in Section 4.3 and the MaxSAT algorithm in Section 4.4, do use extensible leaders. As evidenced by the results in Section 6 the MaxSAT has the best performance for inputs where the labels can have any slope, while the (optimized) Dynamic Programming algorithm is faster for inputs with only leaders of ± 1 slope. For the puzzle inputs the MaxSAT algorithm takes about 5 minutes to find a solution for the most complicated inputs with leaders of any slope, while the Dynamic Programming algorithm was still not able to find a solution after 6 hours. Besides the fact that the Dynamic Programming algorithm is not optimized for inputs with leaders of any slope, this is also due to the fact that the Dynamic Programming algorithm requires a fixed port assignment, and we have not yet found an efficient way to predetermine which assignments will result in a valid labeling. There is some evidence, as shown in Section 6.2.2 which suggests that the Dynamic Programming algorithm performs better on more complicated puzzle inputs, if a fixed port assignment is already given. Finally, the Dynamic Programming algorithm is not able to avoid overlapping labels near corners (for example in Figure 26), as this was not implemented due to time restrictions.

We conclude that the 2-SAT Algorithm is the most useful in applications

where a puzzle needs to be generated in real-time, due to its short running time and the fact that it is able to find a solution for a significant amount of inputs. In any case, the 2-SAT Algorithm could be used to determine if a solution is possible using fixed leader lengths. For applications where it is acceptable to have a longer running time, the MaxSAT algorithm is the most useful as it is able to find a solution within reasonable time for puzzle inputs, with no restrictions on the slopes the labels may have, and it is able to find a minimum length labeling (if one exists) with or without using extensible leaders.

8 Future work

8.1 Improvements to Dynamic Programming algorithm

From Section 6.1 it is clear that the Dynamic Programming algorithm (Section 4.3) does not perform well on the puzzle inputs, which is mostly due to the fact that it requires a fixed port assignment, and therefore each port assignment has to be run through the algorithm separately.
There are several ways the Dynamic Programming algorithm can be improved, but this did not fit in the scope of this thesis. Firstly, each assignment tried is solved as a separate problem, even if this means that the same sub-instances are solved many times over. The algorithm could be significantly sped up for puzzles that have a large number of possible assignments if these solved sub-instances are carried over to subsequently tried assignments.
Secondly, we could try to further reduce the amount of assignments that need to be tried. For instance we could split up the set of labels into several subsets of labels, such that all labels in a subset can only be blocked from being placed closer to the boundary by other labels in the same subset. Then we can use the Dynamic Programming algorithm on each subset separately, trying all the assignments. Then we'd need to combine the separate solutions into all possible complete labelings, for instance using a 2-SAT formula. Finally we would choose the minimum length labeling from these labelings.
Thirdly, the Dynamic Programming algorithm is optimized for inputs with slopes of ± 1 by limiting the amount of possible extension lengths. It might be possible to reduce the number of possible extension lengths for other inputs as well.

8.2 Simplifying the problem

There are several ways the problem of finding a valid labeling could be simplified. While it is essential that labels do not overlap in order to be legible, it may not be necessary to always have a minimum length labeling. In the experiments we used a step size of 2 to reduce the amount of extension lengths that are tried. We could further reduce the amount of extension lengths by using a large step size, and repeating the algorithm with a smaller step size until a valid labeling is found. Then we could still reduce the total extension length by shortening the length of any label that is not blocked by the boundary or another label, until it is blocked by the boundary or another label.

Another way of simplifying the problem would be to allow leaders to cross

up to a certain number of times, or to allow labels to overlap a little bit in a way that the clues are still readable. As long as the clue boxes don't overlap with other leaders, the clues themselves will still be visible. It would be interesting to see how much easier the problem becomes when allowing 1,2,3 or more crossing leaders, and how this affects the legibility of the layout.

Thirdly, it is not always necessary that all nonogram lines are labeled in order for the puzzle to be solved. In fact, removing redundant labels could make the puzzle easier because there are less clues to focus on, or harder because there is less information. Depending on the preference of the puzzler, this might even make the puzzles more entertaining. It would be interesting to see how much easier the problem becomes when labels are removed, and how the difficulty changes.

Finally, in the case of nonograms with curved nonogram lines, instead of having a fixed direction for each port, we could allow a range of directions for the leader, or even a curved leader. Of course, this also increases the amount of ways a label can be placed, which should be taken into account. This approach could also be used in combination with allowing some clue box overlap, by changing the leader so the labels no longer overlap

8.3 Other boundary shapes

One advantage of sloped and curved nonograms is that the boundary does not have to be a rectangle. However, in this thesis we only considered puzzles with rectangular boundaries. It would be interesting to see how the algorithms need to be adapted in order to work with other boundary shapes.

References

[1] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan, *A linear-time algorithm for testing the truth of certain quantified boolean formulas*, Information processing letters **8** (1979), no. 3, 121–123.

[2] K Joost Batenburg, Sjoerd Henstra, Walter A Kosters, and Willem Jan Palenstijn, *Constructing Simple Nonograms of Varying Difficulty*, 15.

[3] K.J. Batenburg and W.A. Kosters, *Solving Nonograms by combining relaxations*, Pattern Recognition **42** (2009), no. 8, 1672–1683.

[4] Michael A. Bekos, Michael Kaufmann, Martin Nöllenburg, and Antonios Symvonis, *Boundary Labeling with Octilinear Leaders*, Algorithmica **57** (2010), no. 3, 436–461.

[5] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff, *Boundary labeling: Models and efficient algorithms for rectangular maps*, Computational Geometry **36** (2007), no. 3, 215–236.

[6] Jessica Davies, *Solving MAXSAT by Decoupling Optimization and Satisfaction*.

[7] Jessica Davies and Fahiem Bacchus, *WDIMACS input format*.

[8] Tim de Jong, *The concept and automatic generation of the Curved Nonogram puzzle*, Master's thesis, Utrecht University, 2016.

[9] Fabian Klute, Maarten Löffler, and Martin Nöllenburg, *Labeling Nonograms*, 2020.

[10] Maarten Löffler, *Impop*, https://github.com/mloffler/impop.

[11] Sancho Salcedo-Sanz, Emilio G. Ortiz-Garcia, Angel M. Perez-Bellido, Antonio Portilla-Figueras, and Xin Yao, *Solving Japanese Puzzles with Heuristics*, 2007 IEEE Symposium on Computational Intelligence and Games (Honolulu, HI, USA), IEEE, April 2007, pp. 224–231.

[12] Frank Staals, *HGeometry*, https://hgeometry.org/.

[13] Nobuhisa Ueda and Tadaaki Nagao, *NP-completeness Results for NONOGRAM via Parsimonious Reductions*, 9.

[14] Mees van de Kerkhof, *Improved Automatic Generation of Curved Nonograms*, Master's thesis, Utrecht University, 2017.

[15] Ian Vollick, Daniel Vogel, Maneesh Agrawala, and Aaron Hertzmann, *Specifying Label Layout Styles by Example*, 2007, p. 10.

[16] Chiung-Hsueh Yu, Hui-Lung Lee, and Ling-Hwei Chen, *An efficient algorithm for solving nonograms*, Applied Intelligence **35** (2011), no. 1, 18–31.

A Results

Puzzle	Assignments tried	Time(ms)			Length			Labeled		
Puzzle		MaxSAT	DP	2-SAT	MaxSAT	DP	2-SAT	MaxSAT	DP	2-SAT
triangle	50	9389	22014	128	15	15	29	6/6	6/6	6/6
hearts	50	12750	70550	121	15	52	67	8/8	8/8	8/8
eye of horus 1 smallest	50	39508	322109	164	138	154	-	15/16	8/16	0/16
eye of horus 1 small	50	40217	395232	198	74	234	100	16/16	8/16	16/16
eye of horus 1	50	40651	607160	179	20	79	420	16/16	16/16	16/16
eye of horus 2	50	40559	635092	294	37	8	177	16/16	6/16	16/16
fox	50	63091	439518	221	34	48	96	20/20	15/20	20/20
fruit	50	119842	929521	282	115	74	248	28/28	18/28	28/28
flowers	50	175545	1451075	321	99	300	-	34/34	16/34	0/34
owl	50	195871	1956627	363	223	117	286	36/36	9/36	36/36
tree	500	302835	21403296	671	162	61	-	44/44	8/44	0/44
tree small	50	313944	-	622	291	-	-	44/44	-	0/44

Table 6: Test results using puzzle inputs, time in ms, for all inputs a clue box size of 16x16, step size 2 and maximum extension length of 64 was used

Ports	MaxSAT				DP			
	Solved		Unsolved		Solved		Unsolved	
	Time	Runs	Time	Runs	Time	Runs	Time	Runs
2	1293	250	1227	50	724	250	675	50
3	1488	201	1437	99	926	201	849	99
4	1695	121	1674	179	1220	120	1120	180
5	2394	128	2329	372	2108	127	1860	373
6	2820	77	2651	423	2719	77	2363	423
7	3321	45	3079	455	3501	45	2977	455
8	3620	19	3564	481	4491	18	3772	482
9	4643	12	4104	488	5979	12	4712	488
10	4636	2	4729	498	7505	2	5743	498

Table 7: Comparison of the run time in milliseconds for the MaxSAT and Dynamic Programming algorithms for the generated single-edge inputs, using labels of size 16, step size 2, and random positions and directions for the ports

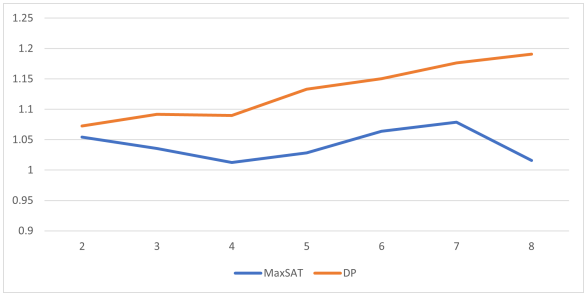

Figure 8: Average run time for solved inputs divided by average run time for unsolved inputs, per algorithm.

B Outputs

B.1 Triangle

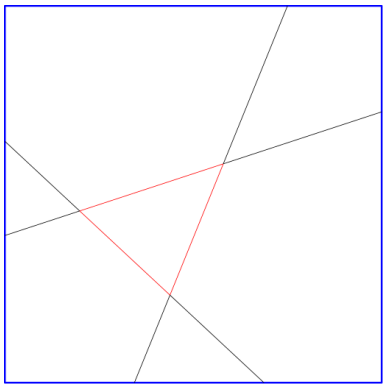

Figure 9: The "triangle" puzzle input

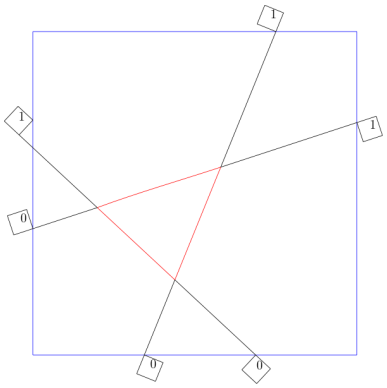

Figure 10: The output of the Dynamic Programming algorithm with the "triangle" input

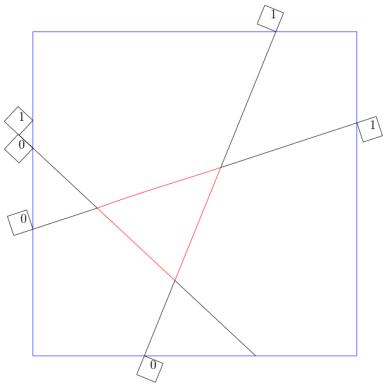

Figure 11: The output of the MaxSAT algorithm with the “triangle” input

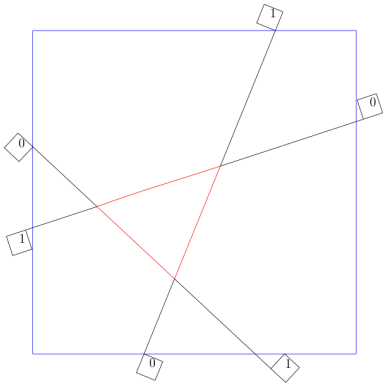

Figure 12: The output of the 2-SAT algorithm with the “triangle” input

B.2 Hearts

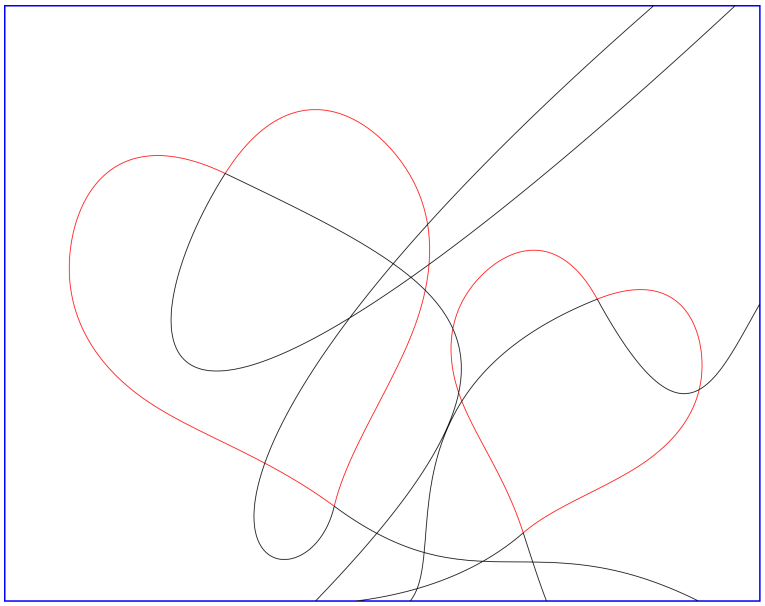

Figure 13: The “hearts” puzzle input

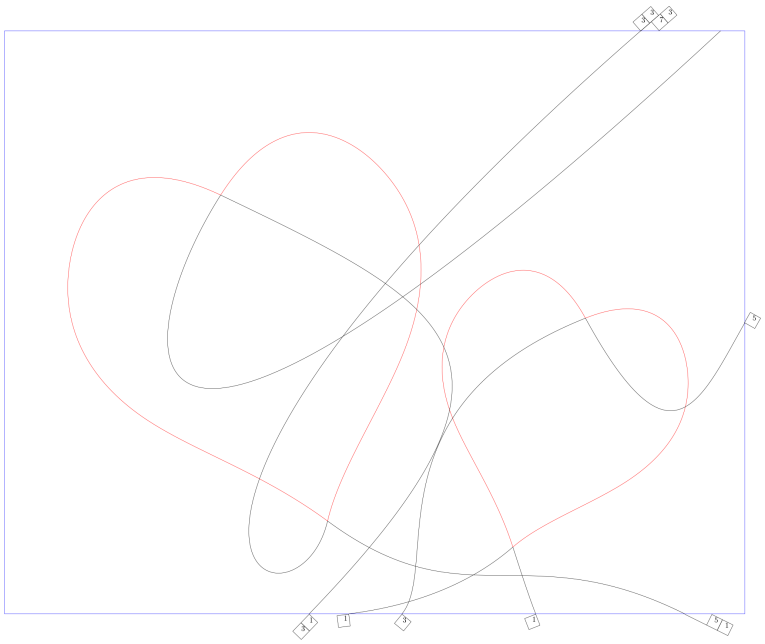

Figure 14: The output of the Dynamic Programming algorithm with the “hearts” input

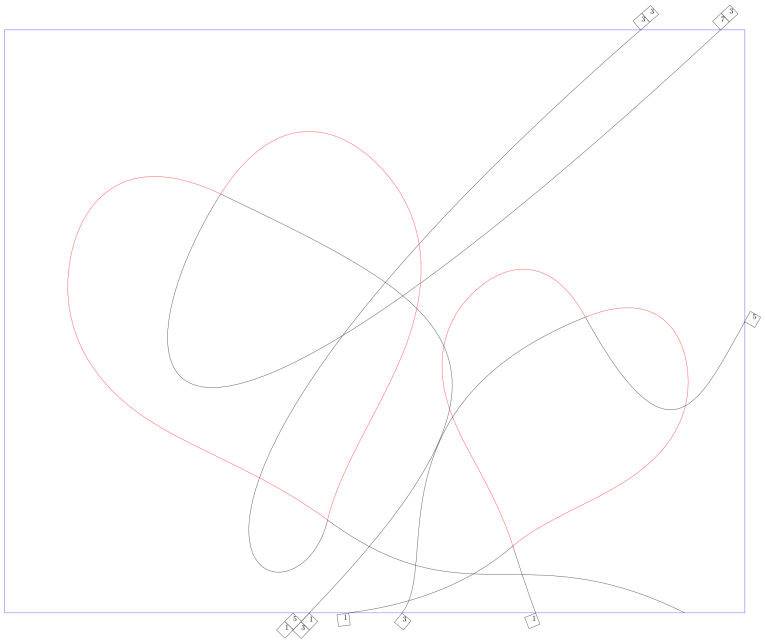

Figure 15: The output of the MaxSAT algorithm with the “hearts” input

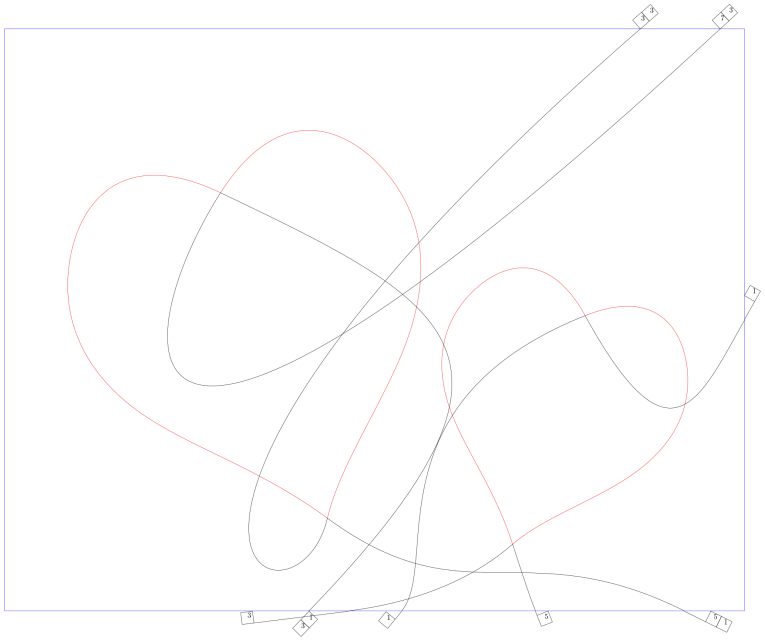

Figure 16: The output of the 2-SAT algorithm with the “hearts” input

B.3 Eye of Horus

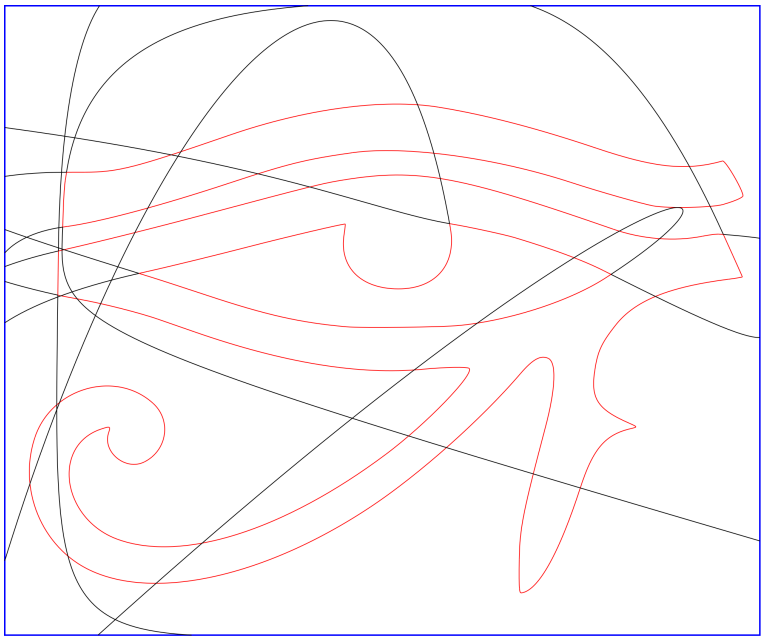

Figure 17: The “eye of horus” puzzle input

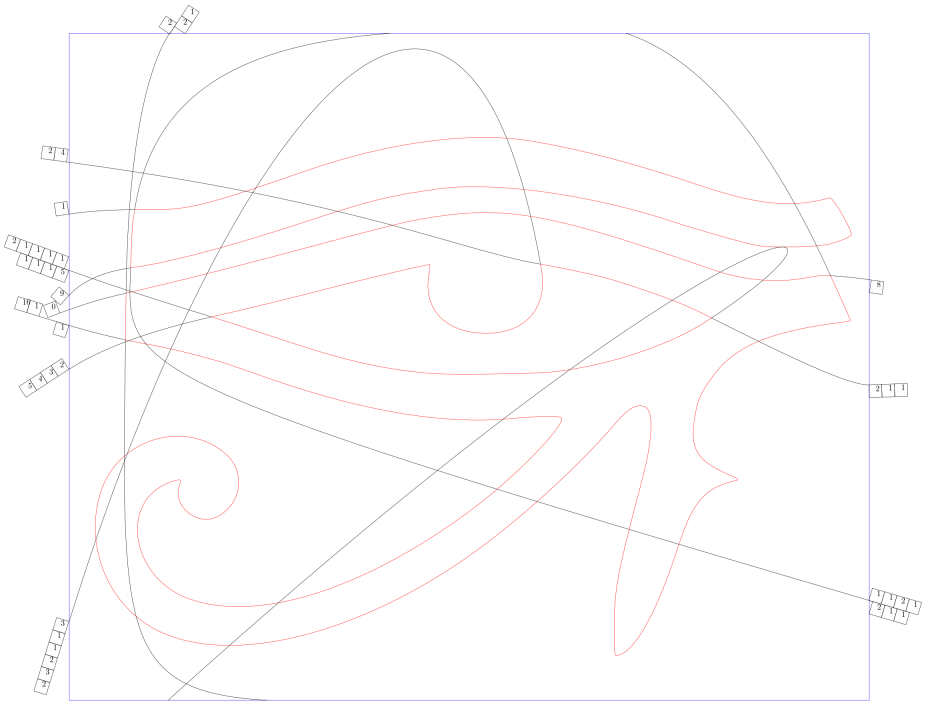

Figure 18: The output of the Dynamic Programming algorithm with the “eye of horus” input

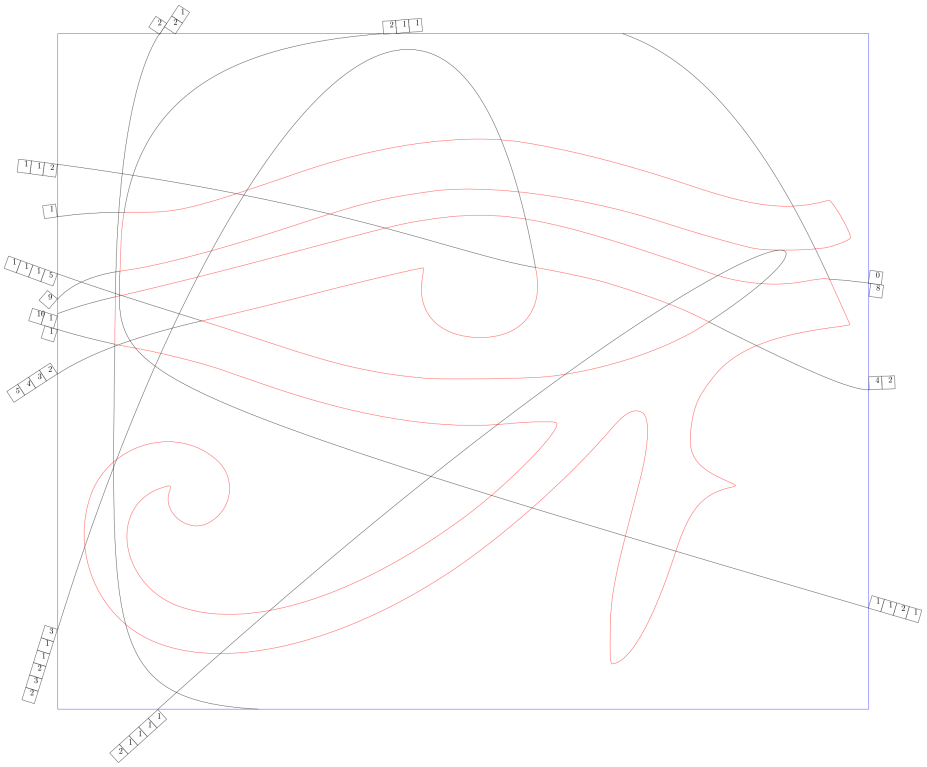

Figure 19: The output of the MaxSAT algorithm with the “eye of horus” input

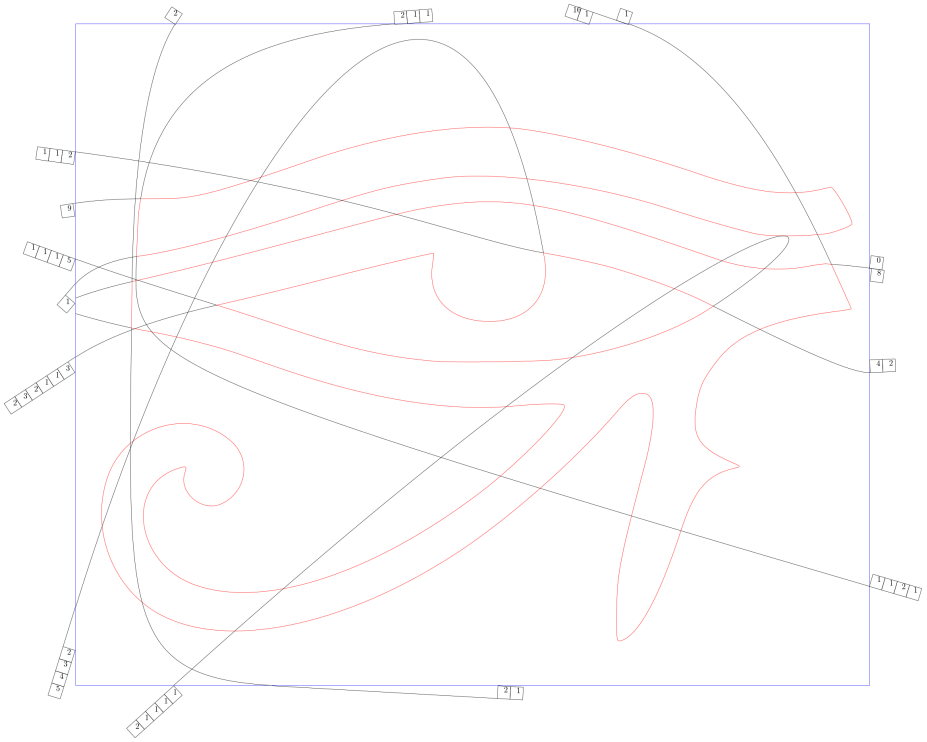

Figure 20: The output of the 2-SAT algorithm with the “eye of horus” input

B.4 Eye of Horus small

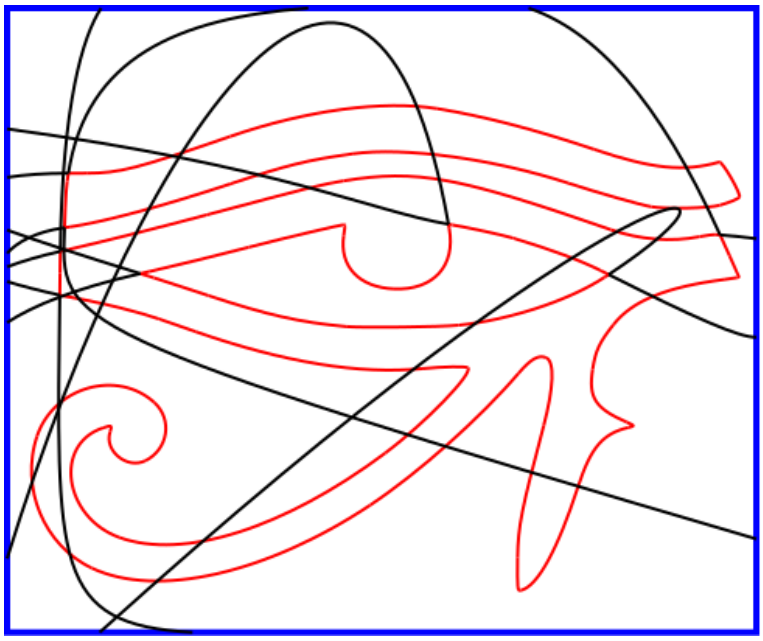

Figure 21: The "eye of horus small" puzzle input

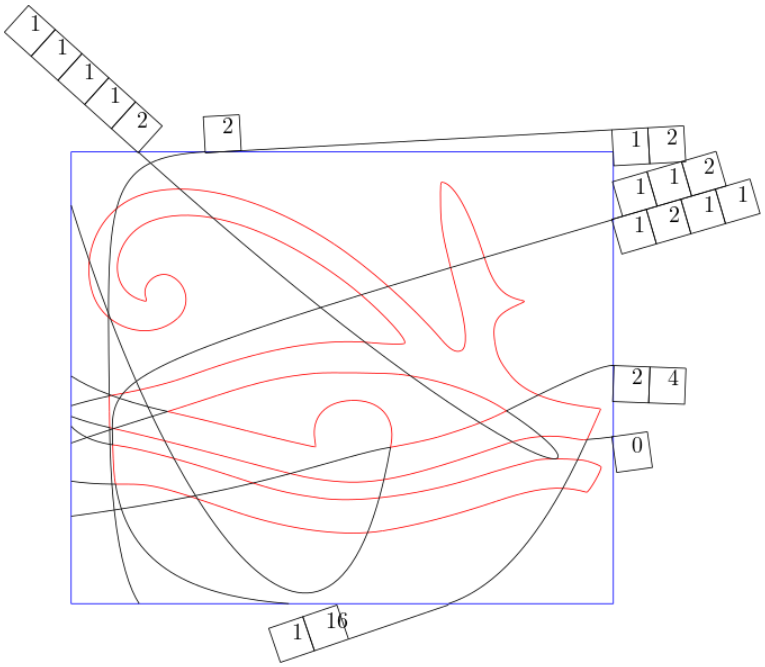

Figure 22: The output of the Dynamic Programming algorithm with the "eye of horus small" input

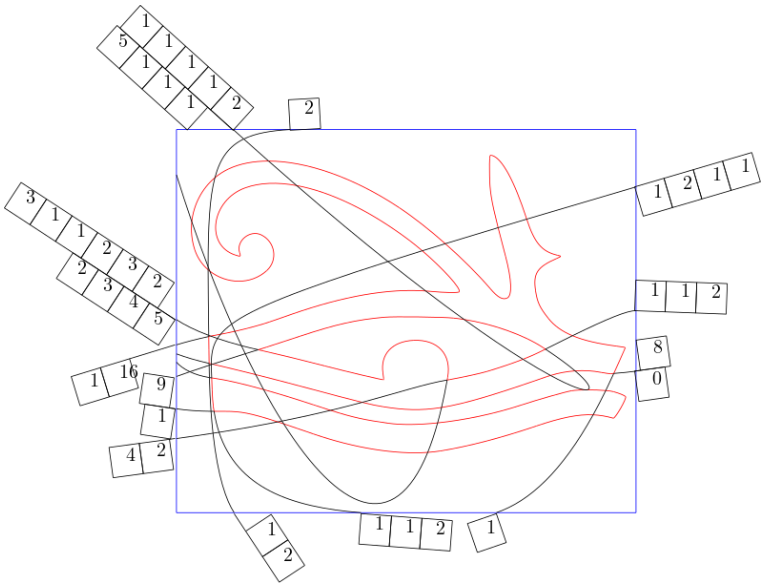

Figure 23: The output of the MaxSAT algorithm with the “eye of horus small” input

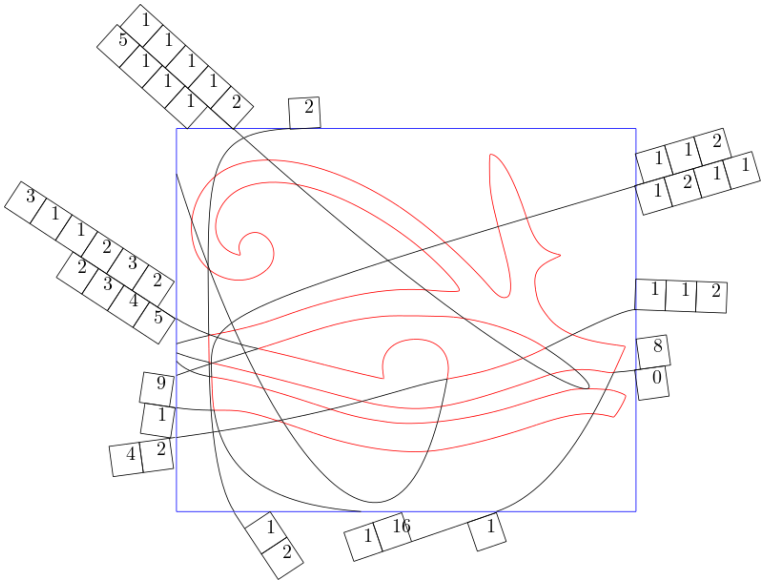

Figure 24: The output of the 2-SAT algorithm with the “eye of horus small” input

B.5 Eye of Horus smallest

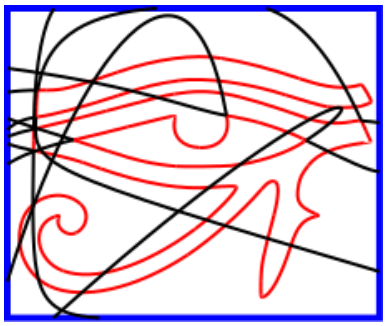

Figure 25: The “eye of horus smallest” puzzle input

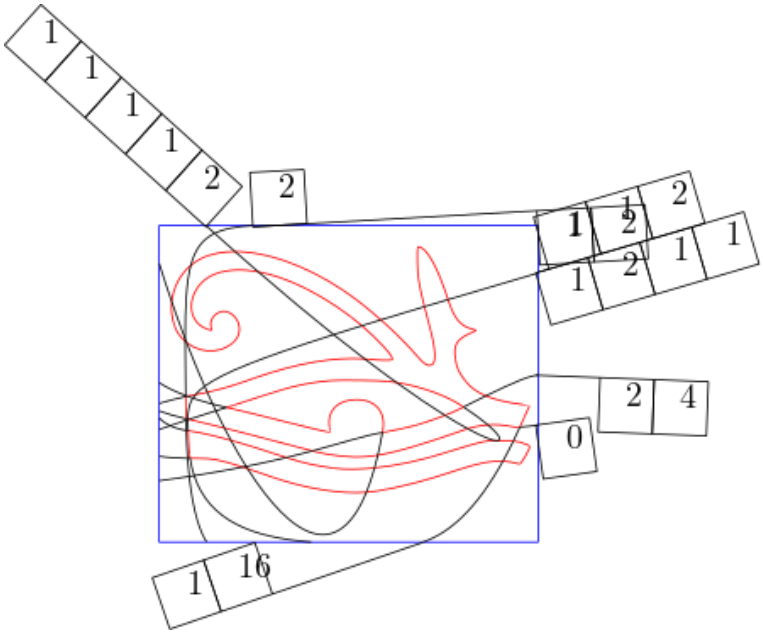

Figure 26: The output of the Dynamic Programming algorithm with the “eye of horus smallest” input

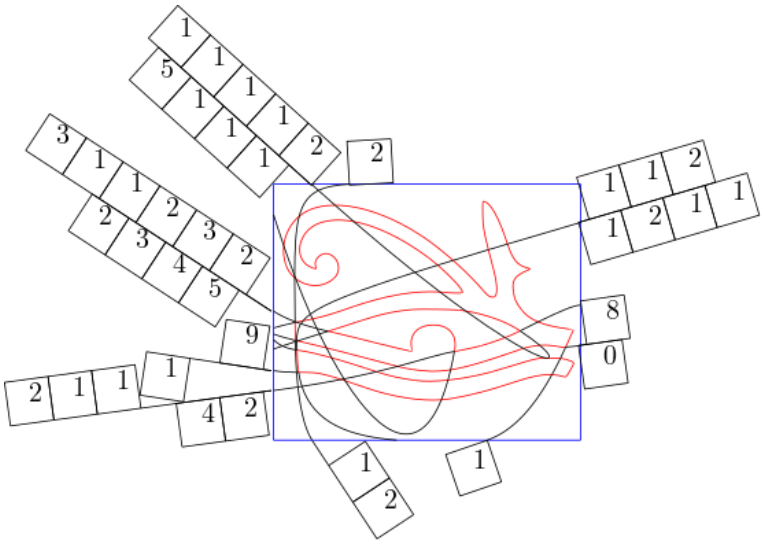

Figure 27: The output of the MaxSAT algorithm with the "eye of horus smallest" input

B.6 Eye of Horus 2

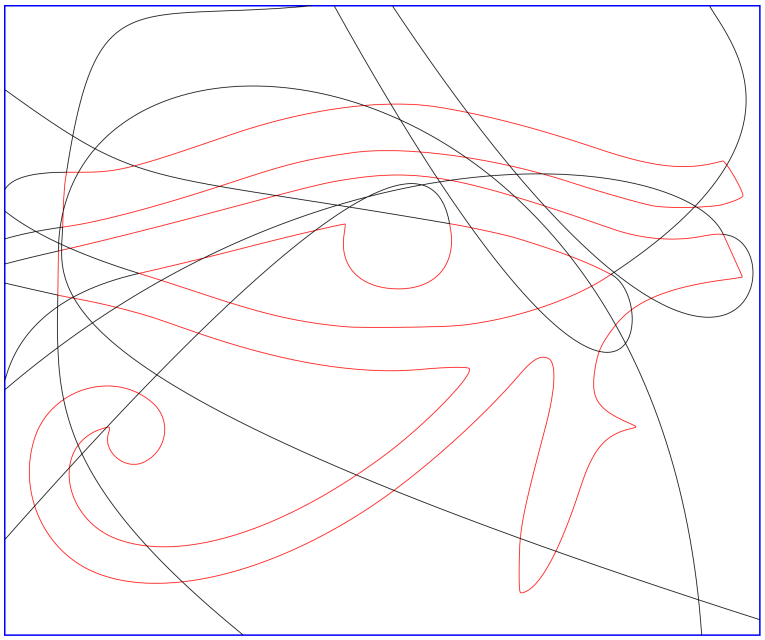

Figure 28: The "eye of horus 2" puzzle input

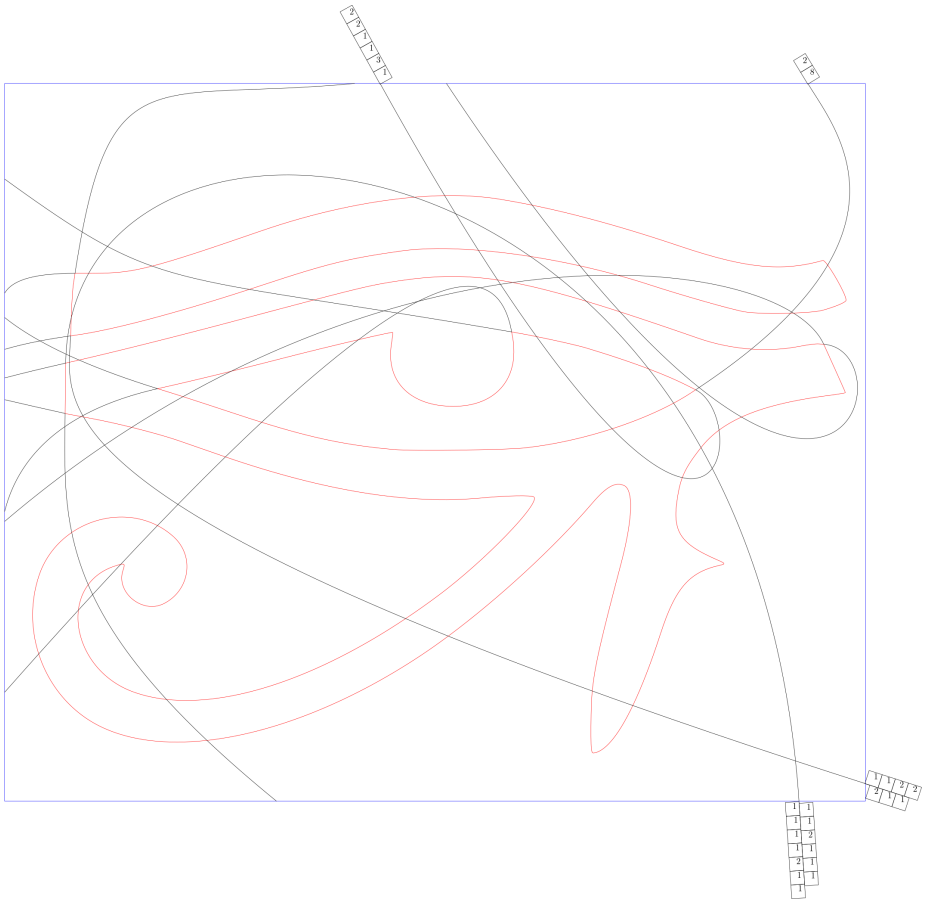

Figure 29: The output of the Dynamic Programming algorithm with the “eye of horus 2” input

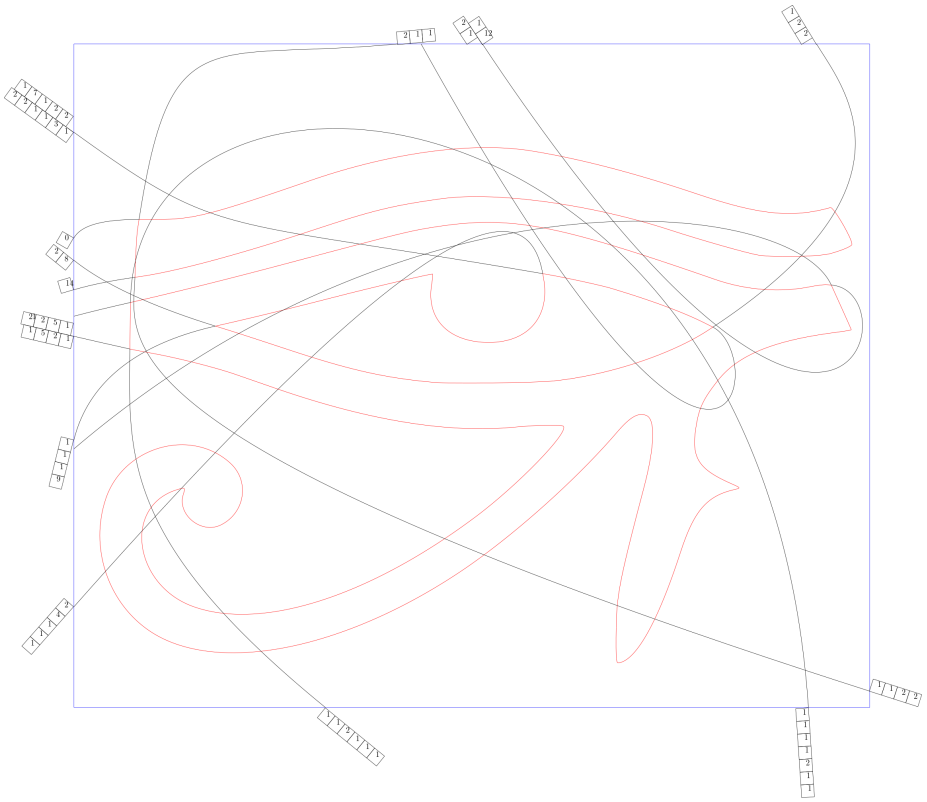

Figure 30: The output of the MaxSAT algorithm with the “eye of horus 2” input

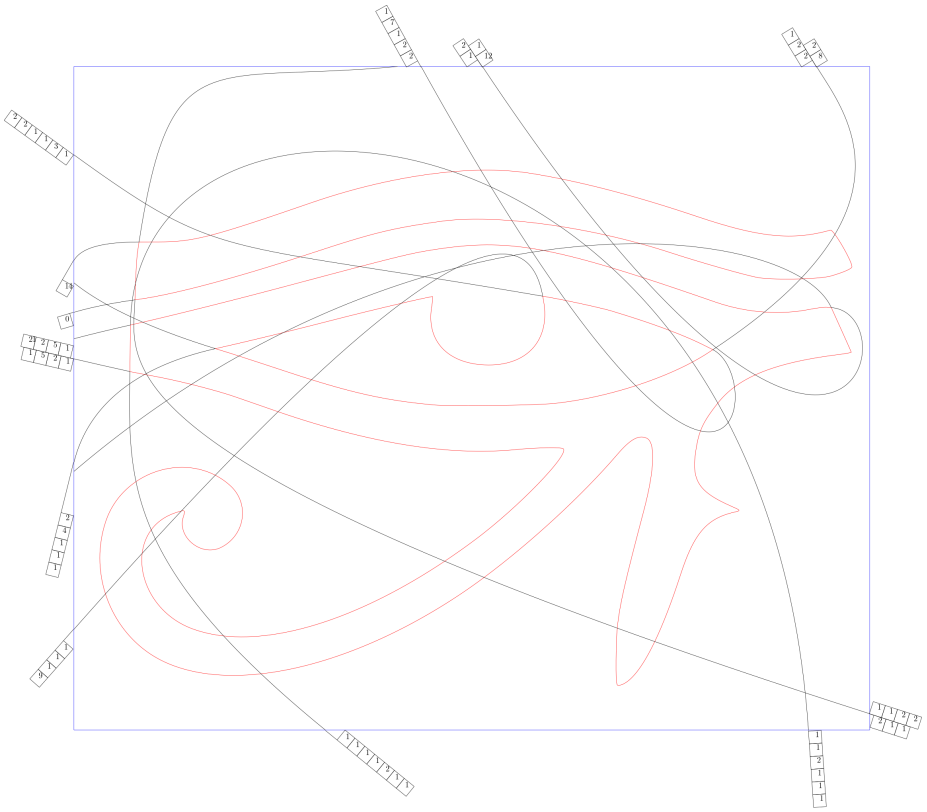

Figure 31: The output of the 2-SAT algorithm with the “eye of horus 2” input

B.7 Fox

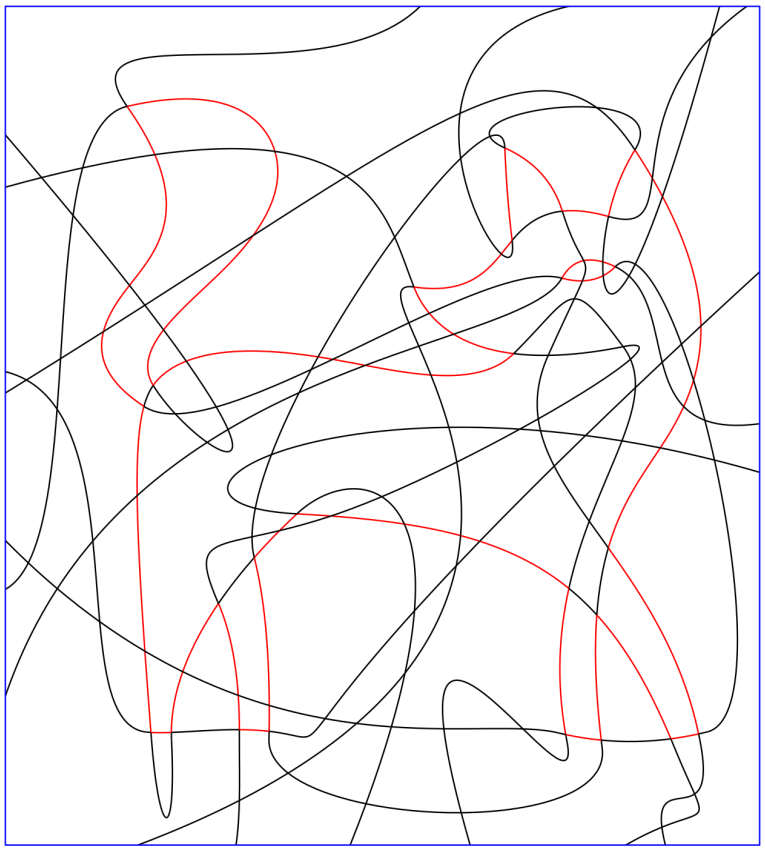

Figure 32: The “fox” puzzle input

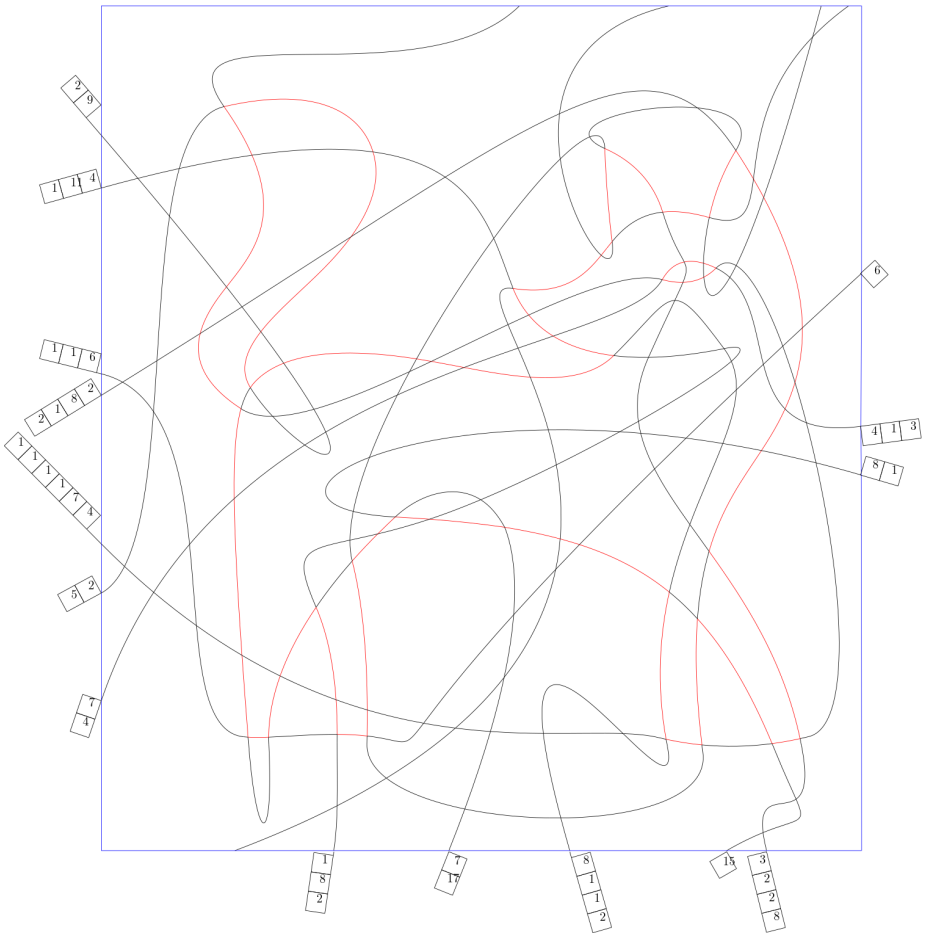

Figure 33: The output of the Dynamic Programming algorithm with the “fox” input

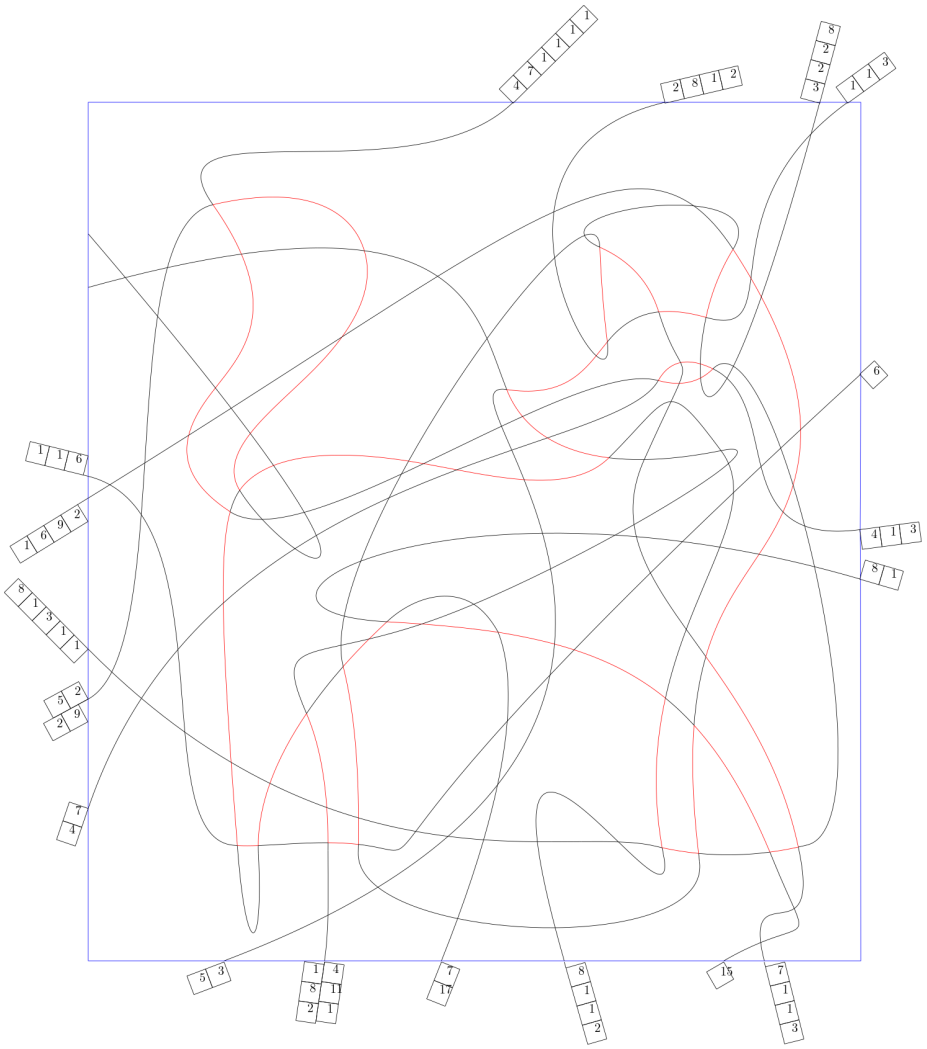

Figure 34: The output of the MaxSAT algorithm with the “fox” input

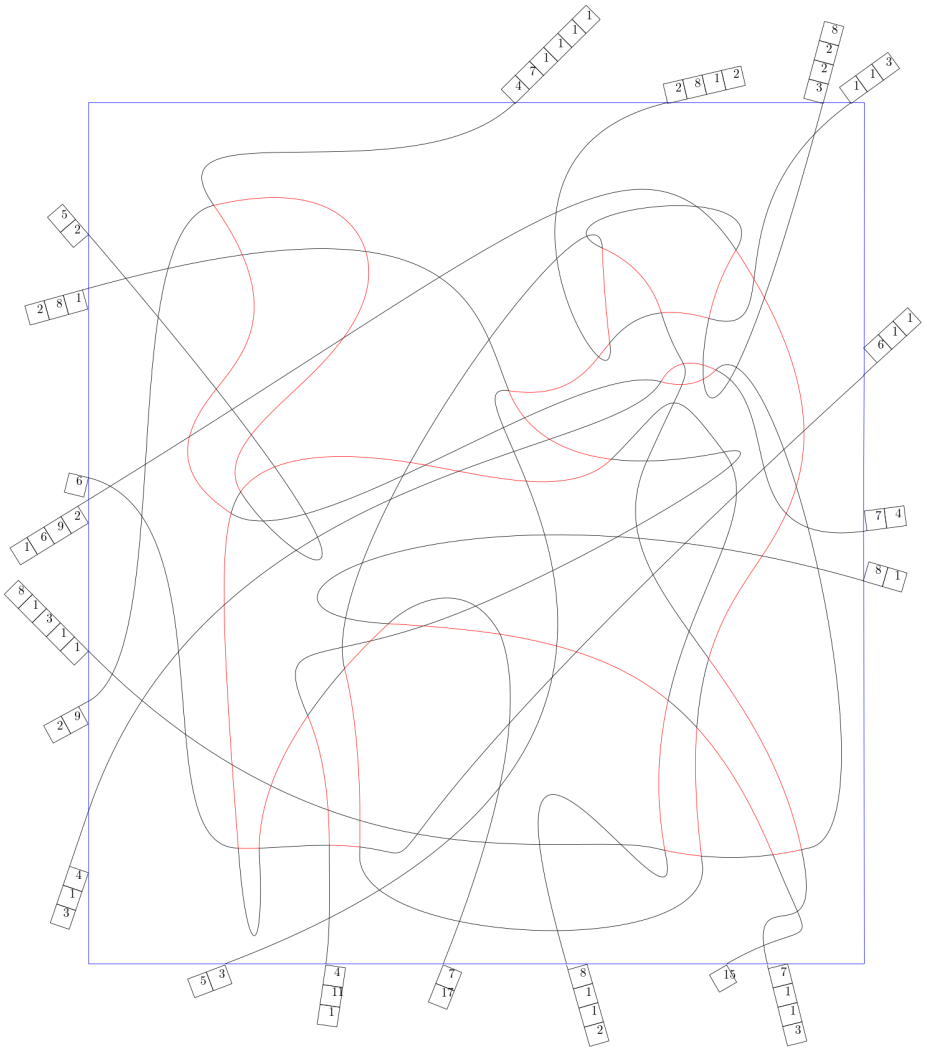

Figure 35: The output of the 2-SAT algorithm with the “fox” input

B.8 Fruit

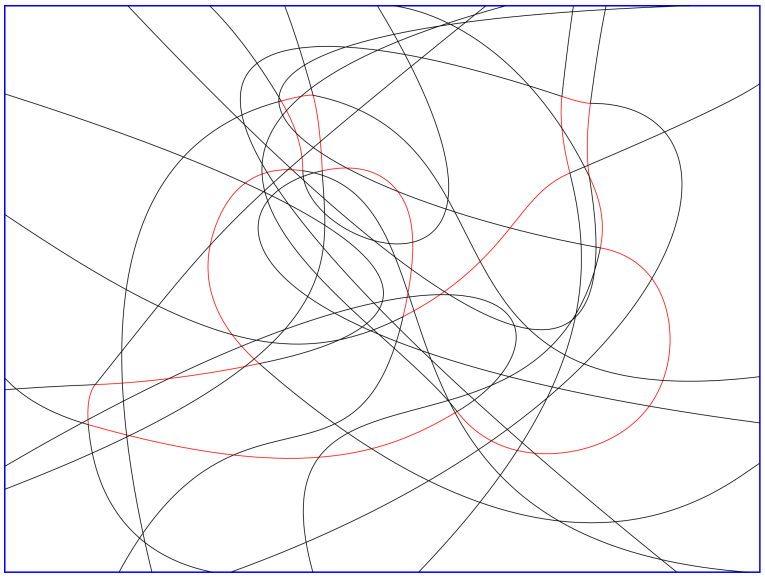

Figure 36: The “fruit” puzzle input

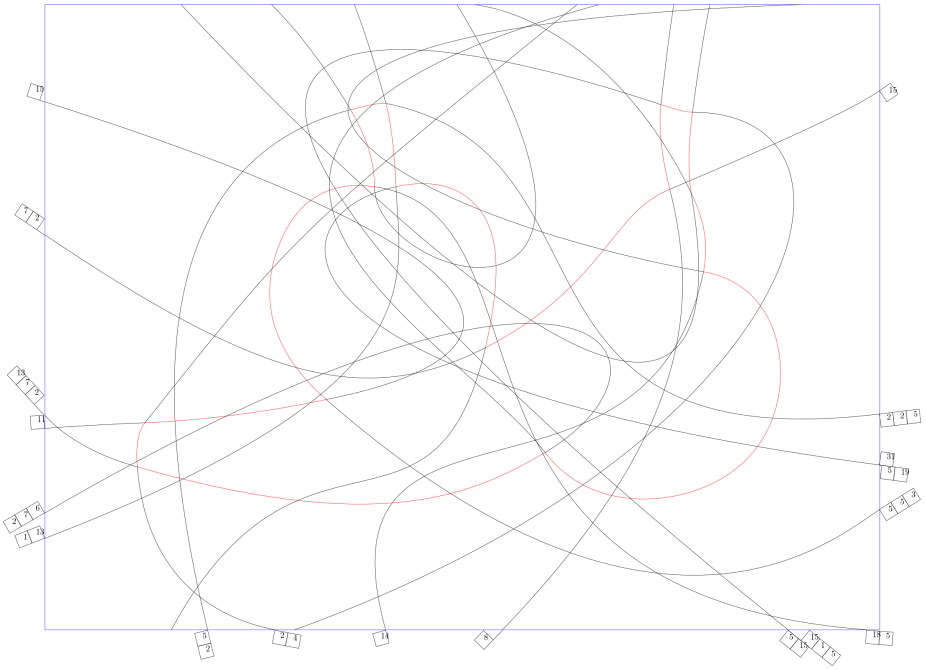

Figure 37: The output of the Dynamic Programming algorithm with the “fruit” input

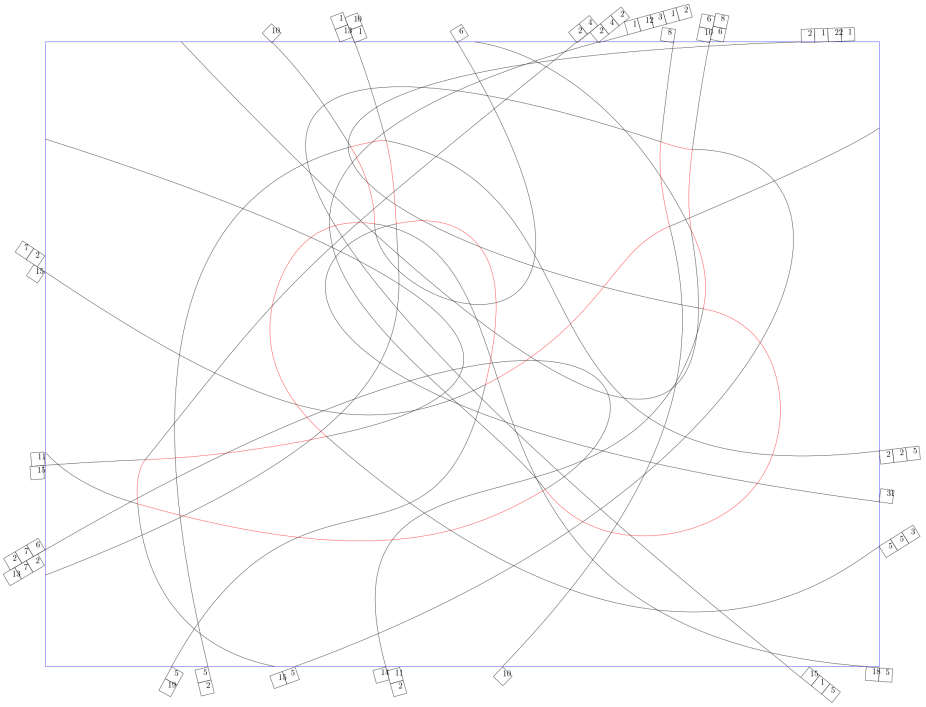

Figure 38: The output of the MaxSAT algorithm with the “fruit” input

B.9 Flowers

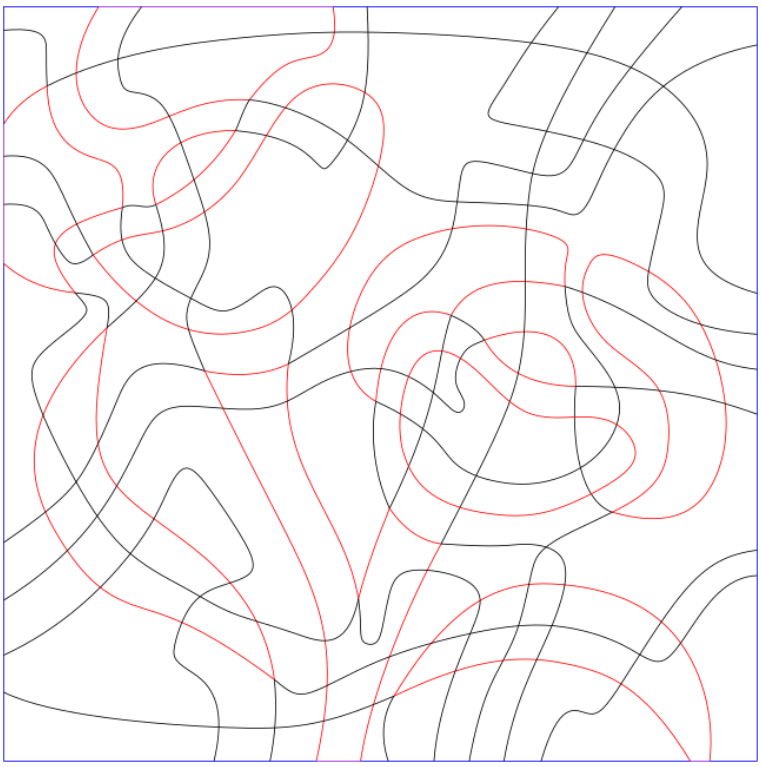

Figure 39: The “flowers” puzzle input

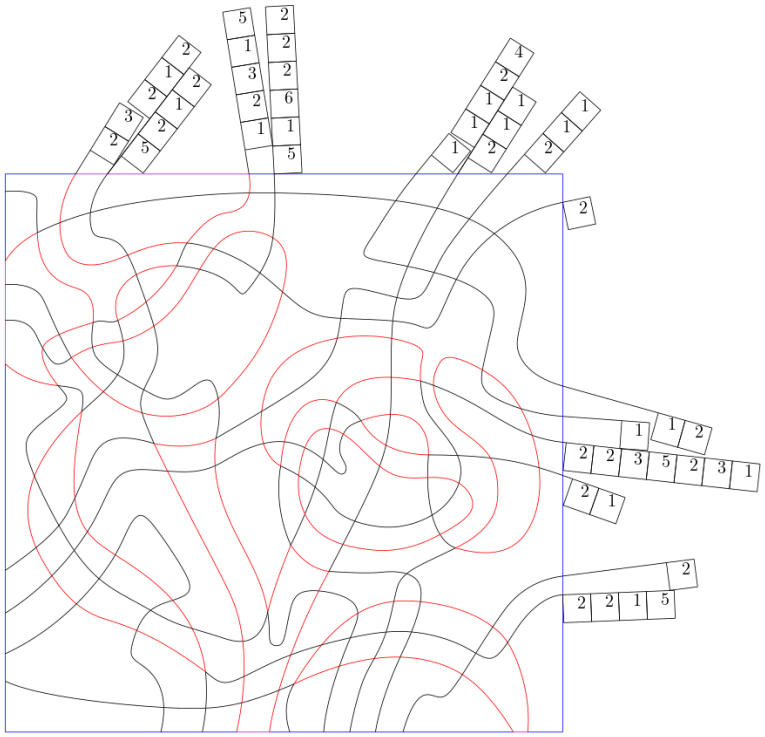

Figure 40: The output of the Dynamic Programming algorithm with the “flowers” input

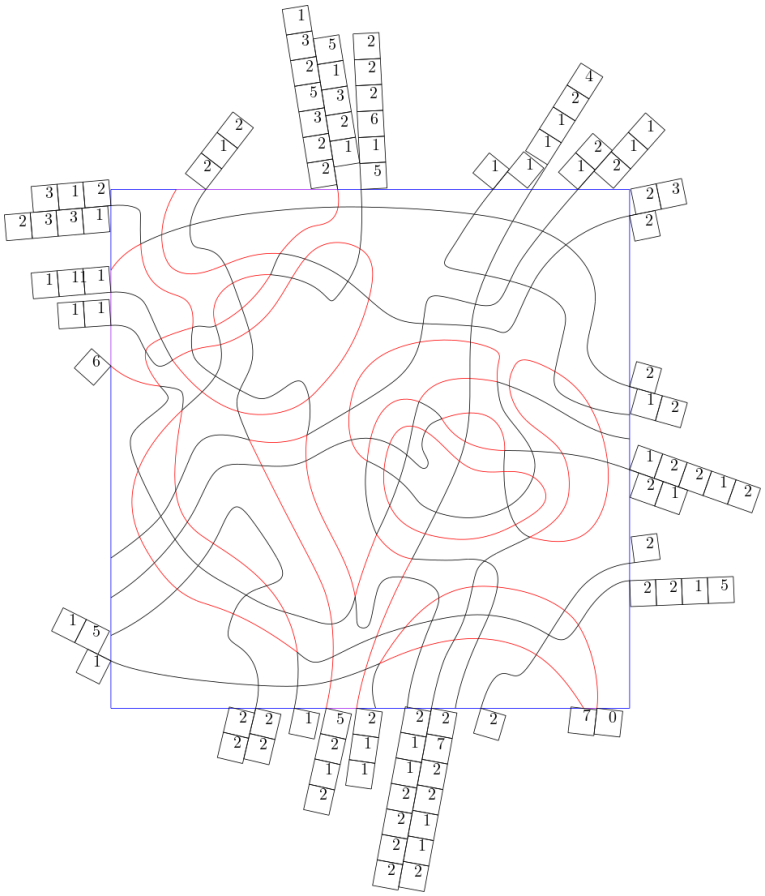

Figure 41: The output of the MaxSAT algorithm with the “flowers” input

B.10 Owl

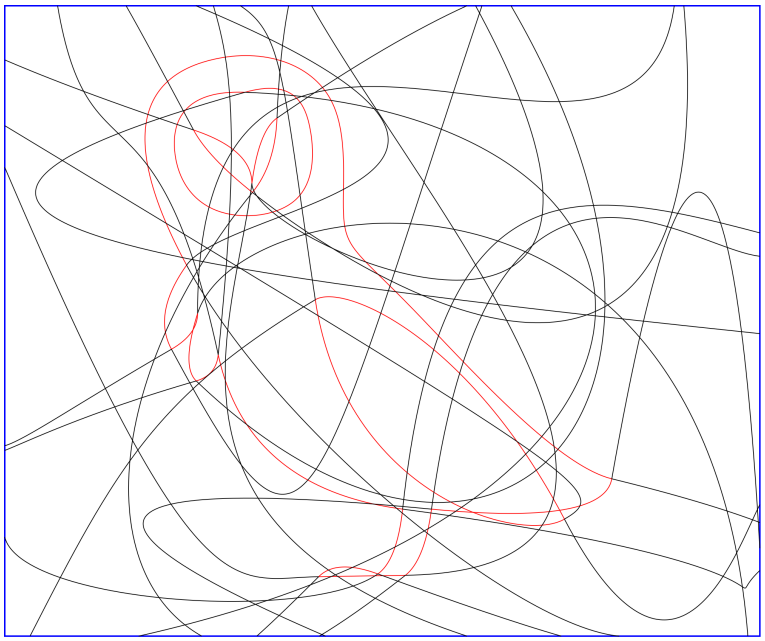

Figure 42: The “owl” puzzle input

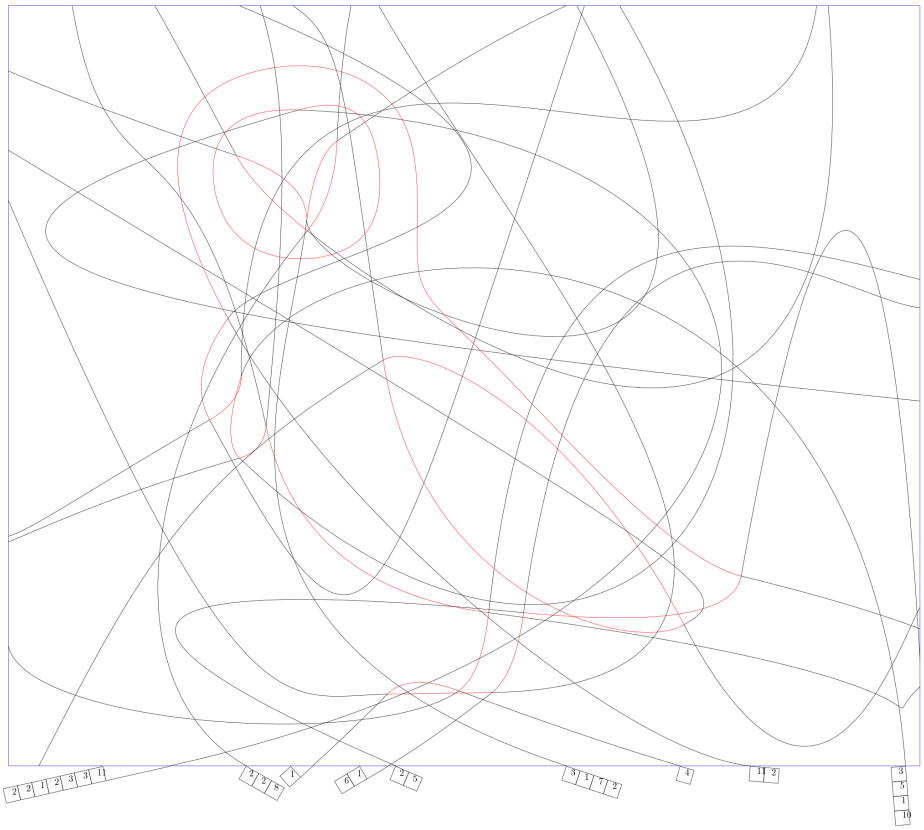

Figure 43: The output of the Dynamic Programming algorithm with the “owl” input

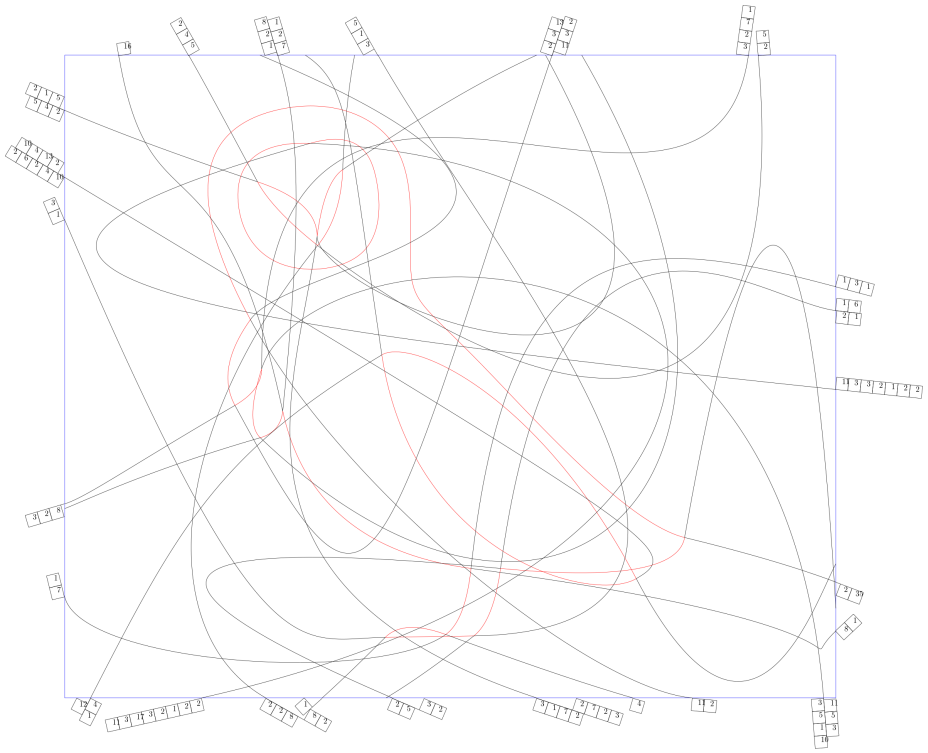

Figure 44: The output of the MaxSAT algorithm with the “owl” input

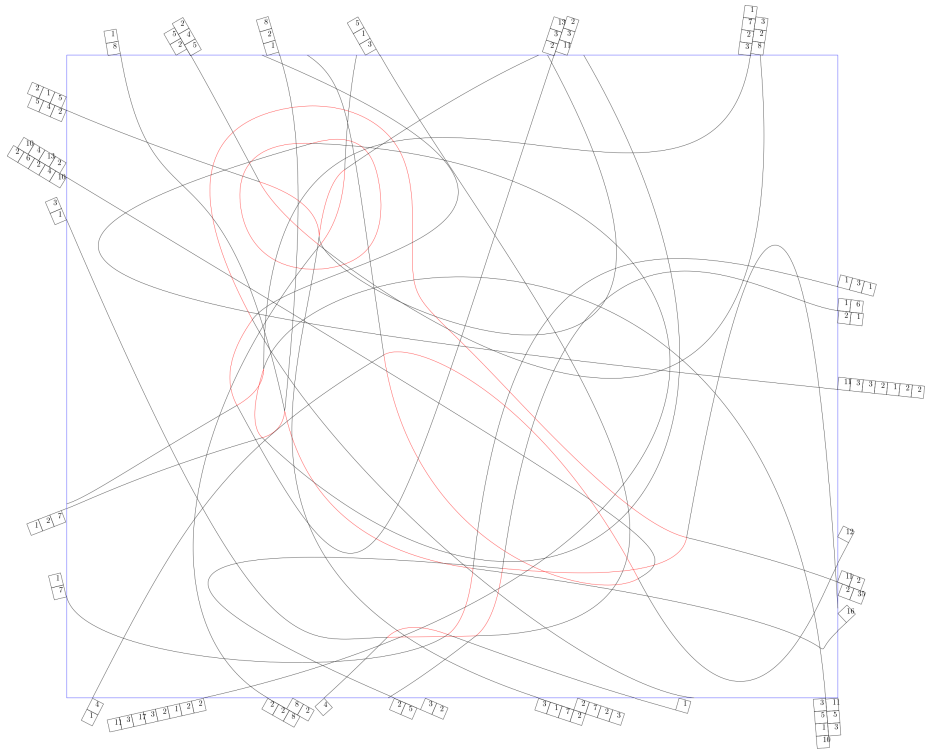

Figure 45: The output of the 2-SAT algorithm with the “owl” input

B.11 Tree

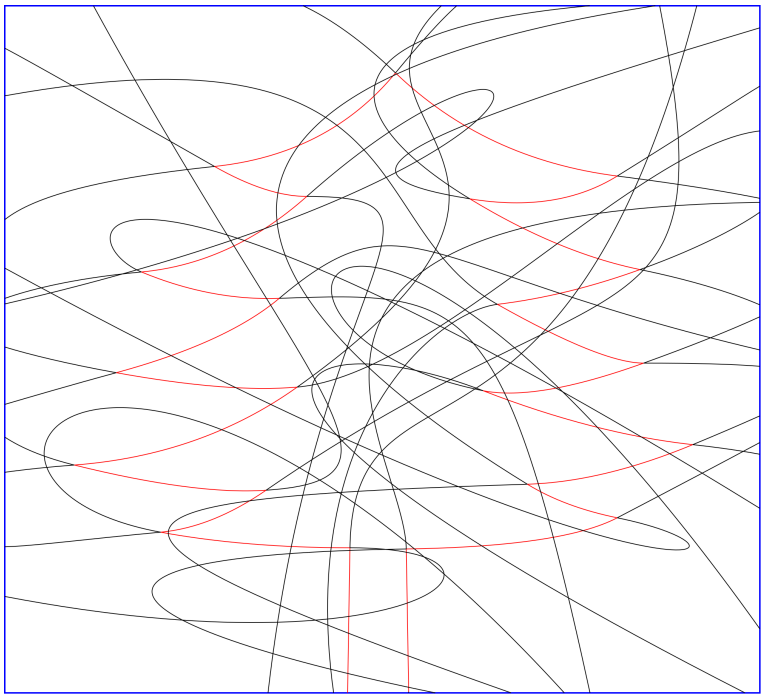

Figure 46: The “tree” puzzle input

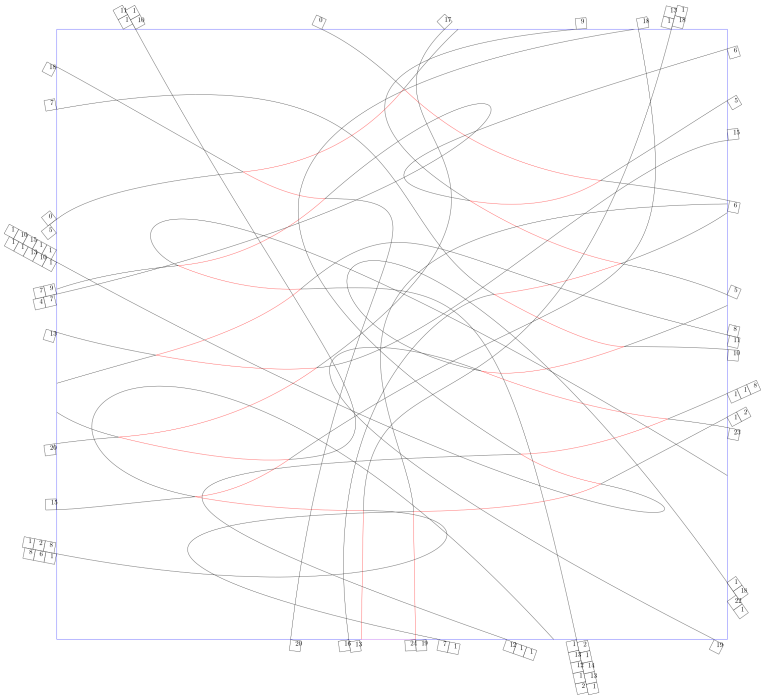

Figure 47: The output of the MaxSAT algorithm with the “tree” input

B.12 Tree small

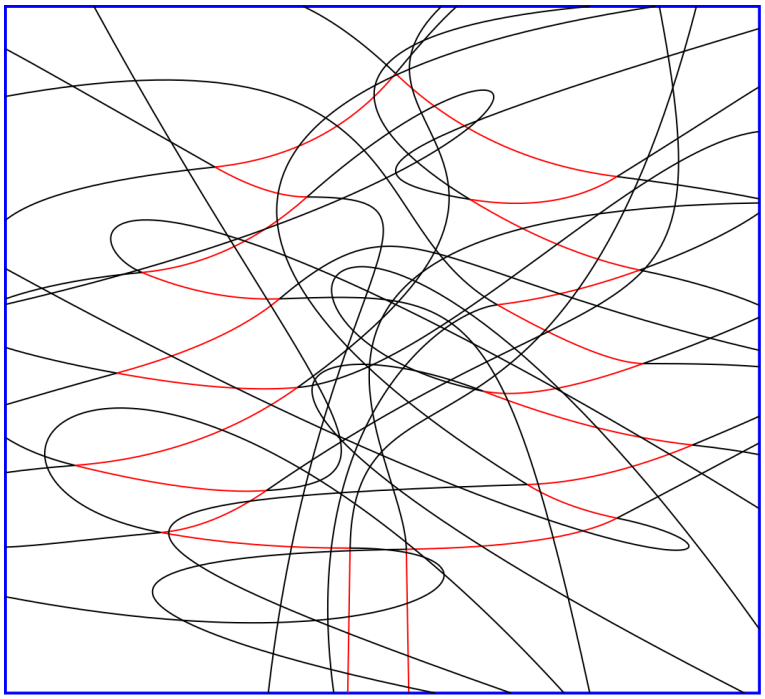

Figure 48: The “tree small” puzzle input

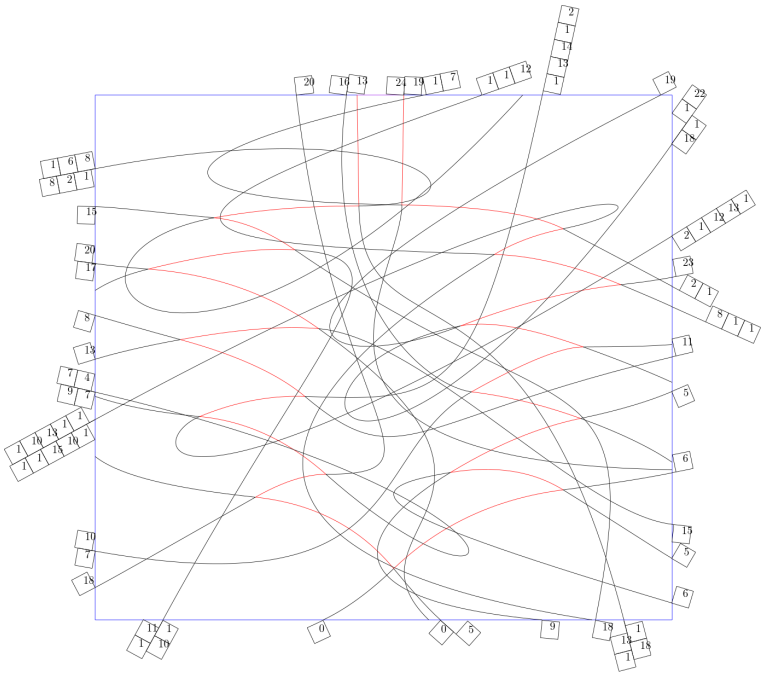

Figure 49: The output of the MaxSAT algorithm with the “tree small” input