university of groningen
faculty of science and engineering
mathematics and applied mathematics

Complexity and solvability of Nonogram puzzles

Betawetenschappelijk onderzoek EC master

April 2017

Student: R.A. Oosterman

First supervisor: Prof.dr. J. Top

Second assessor: Prof. dr. W.H. Hesselink

Contents

Chapter 1

Introduction

Nonograms (Japanese puzzles) were invented in 1987 and are popular in many countries including Japan, England, and the Netherlands. The purpose is to colour multiple pixels on a grid to obtain a picture. Based on numbers corresponding to each row and column of the grid, we can derive which sequences of black pixels need to be coloured, in order to obtain the eventual picture. Solving such a puzzle can be seen as a special case of discrete tomography [1].

The computational complexity of a problem can sometimes be determined or estimated by either giving an algorithm to solve the problem (for so-called P-problems) or verify that a candidate solution satisfies the problem (for NP-problems). Another way of showing the complexity is by giving a reduction from a problem of which the complexity is known, to the given problem. That is, we show that if we could solve instances of our new problem, we could also solve instances of the problem of which the complexity is known. This will turn out to be useful for NP-complete and NP-hard problems.

In this text we intend to discuss the computational complexity of determining the existence of a solution for Nonograms. Before we can grasp the complexity of problems involving Nonogram puzzles, we need to give a few preliminary definitions from graph theory (section 2.1) and in particular constraint graphs (section 2.2), string theory (section 2.3), and complexity theory (section 2.4) in combination with satisfiability problems (section 2.5) of which the computational complexity is known.
In chapter 4, we prove the complexity of the existence problem for Nonograms. We introduce the bounded NCL-problem, and reduce 3-SAT to it. Furthermore, we reduce the bounded NCL-problem to the existence problem for Nonograms, in order to show to which complexity class it belongs.

In chapter 5, we use the concept Another Solution Problem (ASP) to determine the complexity of the uniqueness problem for Nonograms. We introduce the three dimensional matching problem (3DM) and show its complexity similarly to bounded NCL. We use the fact that when a problem has another solution and is reduced to a new problem, that new problem also has another solution. With this notion, we prove that the uniqueness problem for Nonograms is also NP-complete.

After having treated the theoretical complexity of Nonograms, we take a look at how these puzzles can be solved in chapter 6. Since the existence of a solution is NP-complete, it is not immediately obvious how to obtain a solution. We propose several heuristic solving steps to see whether we can determine several pixel values in a row or column, based on specific properties of its description. Furthermore, we compare the functionality of some already existing solvers by applying several examples of Nonogram puzzles. These are applied to cases where a unique solution is known to exist, and also to cases where the solution is not unique.

Chapter 2

Preliminaries

2.1 Graph theory

This section follows the lines of [3, Section 2.1 and 2.2].

A *simple, undirected graph* G is given by $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is the *vertex set*, and $E \subseteq \mathcal{P}_2(V)$ is the *edge set*. That is, the edge set consists of subsets e of V such that $\#e = 2$. The number of vertices n is called the *cardinality* of the graph G, and is denoted by $|G|$. If $\{v, v'\} \in E$ for two vertices v and v', we say that v and v' are *adjacent* vertices, and $\{v, v'\}$ is an adjacent edge of both vertices. A vertex v is said to be *isolated* if it is not adjacent to any other vertex, that is $\{v, v'\} \notin E$ for all vertices $v \neq v'$. We say G is *connected* if for any pair of distinct vertices v, v' there exists a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$ such that $v_0 = v$ and $v_k = v'$.

A *subgraph* $G_S = (S, E_S)$ consists of a subset S of V and a subset $E_S \subseteq \mathcal{P}_2(S) \cap E$, that is, for any edge $\{s, s'\} \in E_S$ it holds that $\{s, s'\} \in E$ and both s and s' are in S. A *path* between two vertices v and w is a sequence of vertices, say $v_0, v_1, \ldots, v_k$, with $v_0 = v$, $v_k = w$ and $\{v_i, v_{i+1}\} \in E$ for each i such that $0 \leq i \leq k - 1$. A *cycle* is a path from a vertex v to itself.

We speak of a *directed graph* if the edges are given a direction, thus $E \subseteq V \times V$ consists of ordered pairs of vertices. When $\{v, w\} \in E$ replaced by $(v, w) \in E$, definitions for undirected graphs are maintained for directed graphs. The edges are given a certain direction, and for an edge (v, v') we say that the edge points *outward* with respect to v, and inward with respect to v'.

In a directed graph, we say that a vertex v is adjacent to v' precisely when $(v', v) \in E$. Note that $v' = v$ is in principle possible here, in this case $(v, v) \in E$ is called a *loop* or *self-loop*. We say that a directed graph is *connected* if for each pair of vertices v, v' there is a sequence of vertices $v_0, v_1, \ldots v_k$ such that $v_0 = v$, $v_k = v'$ and either $(v_{i-1}, v_i) \in E$ or $(v_i, v_{i-1}) \in E$ for each $1 \leq i \leq k$.

A graph is said to be *weighted* if it is given with a function $w : E \to \mathbb{R}$. We say that an edge e has *weight* k when $w(e) = k$. In this thesis, we focus on

graphs that are both directed and weighted. The *(total) inflow* $\tau(v)$ of a vertex v is the sum of the weights of all edges that point inward with respect to v, that is:

$$\tau(v) = \sum_{v'} w((v', v)) \tag{2.1}$$

the sum taken over the vertices v' such that $(v', v) \in E$.

Example 2.1.1. We take the graph $G = (V, E)$ with

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

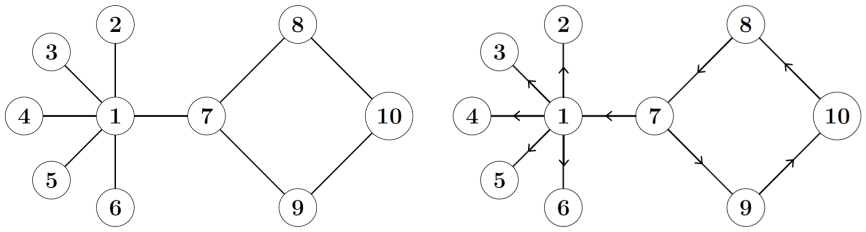
and

$$E = \{\{1,2\}, \{1,3\}, \{1,4\}, \{1,5\}, \{1,6\}, \{1,7\}, \{7,8\}, \{7,9\}, \{8,10\}, \{9,10\}\}$$

as seen in figure 2.1a. This is an undirected, simple graph. The graph does not have any isolated points, and is in fact even connected. The vertices 7, 8, 9 and 10 form a cycle. We can create a directed graph with the same vertex set V, however, we take the edge set

$$E = \{(1,2), (1,3), (1,4), (1,5), (1,6), (7,1), (7,9), (8,7), (9,10), (10,8)\}$$

instead, as we can see in figure 2.1b. This graph is still simple, although it is directed. By definition, 7, 8, 9 and 10 still form a cycle.



(a) Undirected (b) Directed

Figure 2.1: The graph $G = (V, E)$ in both its versions.

2.2 Constraint graphs

A *constraint graph* is a pair (Γ, m) in which Γ is a simple, directed, weighted graph, and $m : V \to \mathbb{R}$. The number $m(v)$ is called the *minimal inflow* of v, and is a lower bound for the inflow $\tau(v)$ of v, that is, $\tau(v) \geq m(v)$.

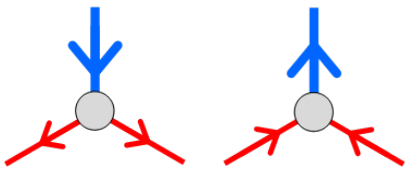
To render the concept of constraint graphs useful for application in games, these

graphs must have a few additional properties: we restrict ourselves to cases with weight function $w : E \to \{1, 2\}$ and do not allow self-loops. In order to make useful drawings of such graphs, edges with weight 1 are coloured red and drawn relatively thin, and edges with weight 2 are coloured blue and drawn relatively thick. From now on, we will speak of *red* and *blue* edges, when considering edge weights.

In contrast to ordinary directed graphs, the edge directions are not fixed, but are allowed to be reversed. A *legal move* is the reversal of a single edge e of a graph G_0, such that the resulting graph G_1 is still a constraint graph. That is, the condition $\tau(v) \geq m(v)$ is still satisfied for each vertex v. From now on we will only reverse an edge when it is legal, in order to satisfy the problems described in chapter 3.

Vertices of constraint graphs can represent logical operators. To begin with, we take a vertex with three adjacent edges and set its minimal inflow $m(v)$ equal to 2. This gives several possibilities. A vertex with an inward blue edge and two outward red edges represents an AND operator. It will therefore be referred to as an *AND vertex*, and it is given in figure 2.2a.

The functionality of this operator becomes clear when we try to reverse the inward blue edge. Since the minimal inflow equals 2, this can only be achieved when both the red edges have already been reversed as well (otherwise, the move would not be legal). This is drawn in figure 2.2b. As both of the edges have to be reversed, we can think of a logical AND.

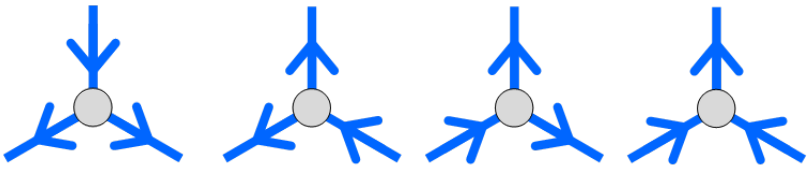


(a) Initial (b) Reversed

Figure 2.2: The AND vertex in both configurations.

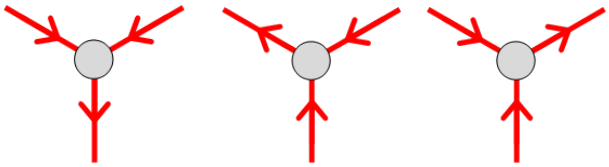
One could also think of a vertex as an OR-operator when it is adjacent to three blue edges, with one edge pointed inward and two edges outward, as drawn in figure 2.3a. When reversing the single inward edge, it suffices to have reversed either one of the output edges (or both of them, as we do not speak of an exclusive OR). This corresponds to a logical OR, as we can see in figure 2.3b.

The CHOICE vertex has three red edges, one pointing inward and two pointing outward. When one edge points outward, the others must be directed inward, to satisfy the minimal inflow of 2. This results in the fact that either one of the inward edges can be reversed, but not both of them, and this is what "CHOICE" relates to. Note that the single outward edge already has to be reversed in each case, although this does not change anything to the functionality.



(a) Initial (b) All possible reversals

Figure 2.3: The OR vertex in all configurations.

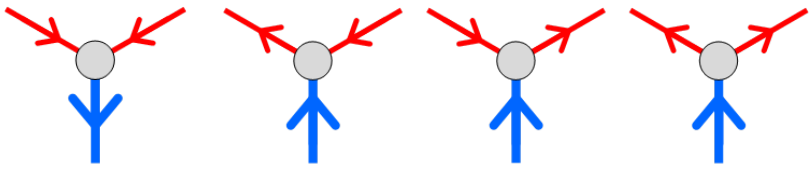


(a) Initial (b) Both possible reversals

Figure 2.4: The AND vertex in both configurations.

This vertex represents the functionality of a logical NOT in the following way: assign a logical variable to one inward edge, say x, and its negation to the other one, say $\bar{x}$. This means that if the x edge points outward, the $\bar{x}$ has to point inward and vice versa. When we imagine pointing outward as a variable being true, it immediately follows that its negated form has to be false, and the other way round. This will become useful in section 4.1, where we will construct graphs that apply to logical formulas.

The FANOUT vertex has two inward red edges and one single outward blue edge, see figure 2.5a. In principle, this vertex is the same as the AND vertex, although the roles of the edges have changed. The vertex has more or less the same functionality as the CHOICE vertex. However, with this vertex we can reverse either one of the inward edges, as well as both of them. Note that all reversals require the blue outward edge to be reversed on forehand, see figure 2.5b.



(a) Initial (b) All possible reversals

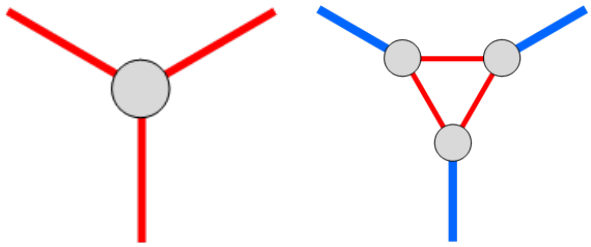
Figure 2.5: The FANOUT vertex in all configurations.

Note that is it also possible to have vertices with one red and two blue edges. These vertices turn out to be not very useful, and therefore we will omit them.

The vertices described above can be connected to each other to create a constraint graph. For each of the possible vertices then, we have to satisfy the minimal inflow of 2 in any case. However, in some cases it is necessary to connect two vertices by a single edge, although the edge needs to have different weights in order to reach the functionality of its adjacent vertices. Of course, this is impossible, and therefore we introduce vertices with two adjacent edges, a red one and a blue one. We name such a vertex a *red-blue vertex*, and it has a minimal inflow of 1. This means that one edge is pointed inward if and only if the other one is pointed outward. Since the minimal inflow is 1, it does not matter which edge is pointing inward, and therefore these can be reversed. These vertices are useful when constructing graphs in which the orientation of a certain edge has to be preserved, and meanwhile the weights have to be exchanged.

We may wonder whether there are other useful vertices with degree 2 and minimal inflow 1. One could think of a vertex as a NOT operator, that is, interchanging the orientation of an edge by a vertex. However, this is not very convenient, since it would mean that both edges are either pointing inward or outward. The latter case contradicts the principle of minimal inflow, as this would equal 0, and results in an illegal configuration. It would therefore not be possible to invert one of the edges, so this concept of a logical NOT is useless in the sense of constraint graphs. Note that we have already implemented a logical NOT in some sense by introducing the CHOICE vertex, which removes the necessity of a NOT vertex concerning strictly logical input (one vertex) and output (one vertex).

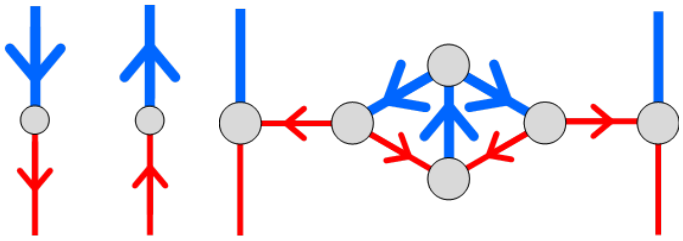
Graphs that contain AND combined with OR vertices, together with red-blue vertices, are said to be *AND/OR constraint graphs*. Note that the CHOICE and red-blue vertices are not contained in the definition of AND/OR constraint graphs. However, we can replace them with equivalent *AND/OR subgraphs*, that is, a structure of connected AND and OR vertices that will maintain the behaviour of these vertices. The CHOICE vertex is equivalent to the AND/OR subgraph drawn in figure 2.6b. Note that each of the red edges has been replaced by a blue one. With a red-blue vertex placed between each of the edges, all problems should be omitted.



(a) Original vertex (b) Equivalent subgraph

Figure 2.6: The replacement of a CHOICE vertex by an AND/OR equivalent subgraph.

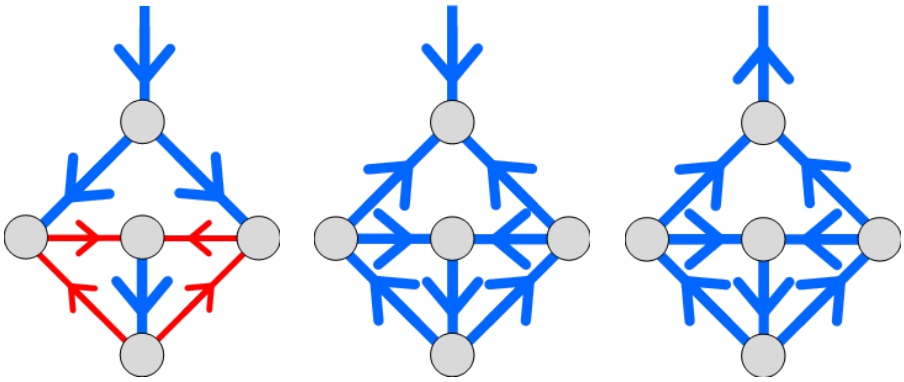
We have to notice that red-blue vertices always come in pairs, as they will be connected to the ends of either an AND or OR vertex. The equivalent subgraph of two red-blue vertices is shown in figure 2.7. Note that in figure 2.7b the orientation of the original edges is absent, as this orientation does not depend on the further structure of the subgraph. This is to be verified by the reader.



(a) Vertices (b) Equivalent subgraph

Figure 2.7: The replacement of two red-blue vertices by an AND/OR equivalent subgraph.

Sometimes we have to create an edge, without further connection to another vertex. Such edges are named *loose edges*, and they can be replaced by an equivalent subgraph as well. One has to note that loose edges have several possible purposes: either the edge has the freedom to be reversed, or its direction should be determined on forehand. In figure 2.8a, the equivalent AND/OR subgraph of a locked edge is shown, whereas figures 2.8b and 2.8c show the possible directions of a free loose edge, replaced by an equivalent subgraph.

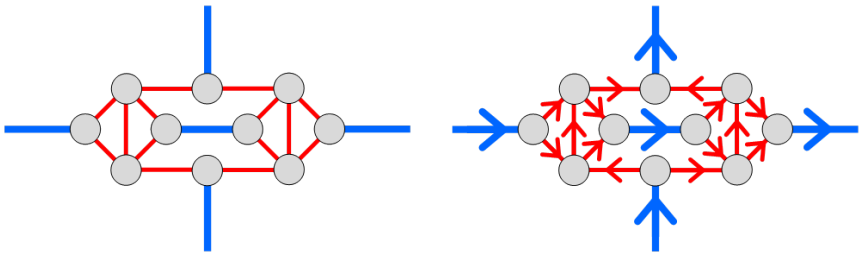


(a) Locked edge (b) Free edge inward (c) Free edge outward

Figure 2.8: Determination and orientation of loose edges.

For several problems, we specifically have to look at *planar graphs*. That is, graphs that do not have any self-intersections concerning the edges. We assume that edges are always drawn in straight lines. In order to avoid self-intersections, we can replace such an intersection by a subgraph that preserves the orientation

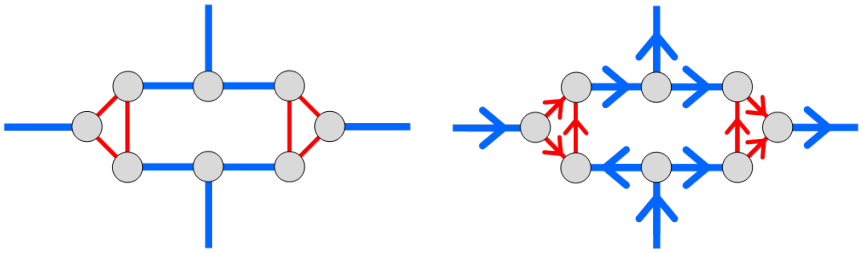
of opposite edges: the *crossover gadget*. In figure 2.9, the crossover gadget is shown in its directed and undirected form. Note that there are vertices with four red edges are contained in this subgraph, which means that at least two of its edges have to point inward. It is left to the reader to verify that, when one of the inward loose edges is reversed, the opposite edge must be reversed as well, and therefore, the orientation of these edges will always be preserved.



(a) Undetermined directions (b) Determined directions

Figure 2.9: The crossover gadget.

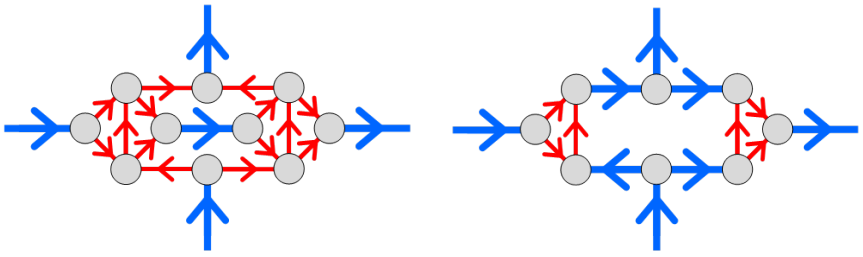
Note that this gadget is in no sense an AND/OR constrained (sub)graph, as there are vertices with four red edges involved. For this reason, we need to construct an equivalent subgraph, or at least a subgraph with the same functionality. The *half-crossover gadget* is a gadget for which at least two of the outer loose edges must face inward. In figure 2.10, a certain configuration of the half-crossover gadget is shown, as well as its undirected version.



(a) Undetermined directions (b) Determined directions

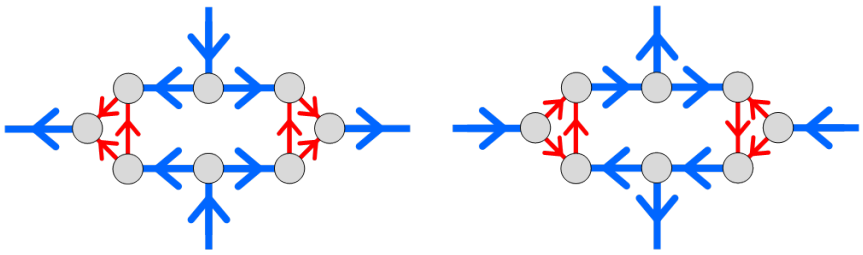
Figure 2.10: The half-crossover gadget.

We immediately have to show that this gadget indeed may preserve the orientation of the opposite loose edges, as shown in figure 2.11.
However, there are a few other configurations possible, namely either one pair of opposite loose edges turn inward and the other one pointing outward, see figure 2.12. There are several other configurations possible, as there must at least two of the loose edges direct inward. This is to be verified by the reader. The functionality of this gadget at least suffices for our purpose, namely maintaining the orientation of intersecting edges. When two orthogonal (that is, not opposite, as we cannot speak of adjacency) loose edges both point outward, it means that



(a) The crossover gadget (b) The half-crossover gadget

Figure 2.11: Equivalence of the two gadgets



(a) First configuration (b) Second configuration

Figure 2.12: The other two configurations of the half-crossover gadget.

their opposite edges must point inward, and therefore, their orientation will be passed on through the gadget.

With all the possible vertices, edges, subgraphs and gadgets, we can construct several constraint graphs. These graphs will be used to determine the complexity and solvability of several games and puzzles, such as the Nonogram puzzle, which we will describe in chapter 3.

2.3 String notation

These definitions are contained in [12, Section 2.1].

An *alphabet* Σ is a finite set of *symbols*. A *string* σ over Σ is a finite sequence of symbols. We denote the *set of all strings over* Σ by Σ^*. A *language* L over Σ is a subset of Σ^*, and is often given by a certain description. A *description* for a language L is a general formula, given by the different symbols, and the following descriptors:

- a^k with $k \in \mathbb{N}$ means the symbol a is written k times after each other.
- a^* means that a may appear any number of times after another. That is, a^* is a string contained by the infinite set $\{\emptyset, a, aa, aaa, \ldots\}$.

- a^+ means that a may appear any positive number of times after another. That is, a^+ is a string of the infinite set $\{a, aa, aaa, \ldots\}$.

2.4 Computational complexity

When looking at algorithms for solving problems, we want so say something useful about the possible efficiency of such algorithms. The *time* used by an algorithm can either be represented by calculation time or the number of calculation steps.

Definition 2.4.1. An algorithm is said to take *polynomial time* if a polynomial $f(n)$ exists which is an upper bound for the time used by this algorithm, where n is the input size of an instance to which the algorithm is applied.

With this notion, we can already classify several problems in two complexity classes.

Definition 2.4.2. A problem is said to be in the *polynomial class* or shortly P if there exists an algorithm that solves the problem in polynomial time.

Definition 2.4.3. A problem is said to be in the *nondeterministic polynomial class* or shortly *NP* if there exists an algorithm that verifies for a candidate solution in polynomial time of input size n whether it is a solution to the problem.

In short, we say that a problem *is* P respectively *NP*.When a similar statement for the amount of computational space holds, in addition to the amount of used time, we say the problem belong to the class *P-SPACE*, or simply is P-SPACE. Note that any problem in P is also P-SPACE, since any process using finite time will always use a finite amount of space. The converse is not always true: a process using only a finite amount of space can take infinite time. A well-known example is a while-loop. Although it only uses a finite amount of space (the programming code and thus far created variables), it can last forever as long as its conditions are not (yet) satisfied.

A problem A is said to be *reducible* to a problem B if there exists a polynomial time algorithm that links the outcomes of A surjectively to equivalent outcomes of B. We say that B is in the class *NP-hard* if this problem is reducible to any problem in NP in polynomial time. We say shortly that a problem is NP-hard in this case. Furthermore, a problem is in the class *NP-complete* if it is both NP and NP-hard. With this notion, we take certain problems of which they are known to be NP-complete, and reduce them to new problems in polynomial time. In this way, we can show for these problems to be NP-complete as well.

2.5 Satisfability problems

A *Boolean formula* is a formula containing variables that can be assigned either *true* or *false*, known as *Boolean variables*. These variables are combined with logical operators to create expressions, which can be true or false as well. These definitions are related to [9, Chapter 3]:

- The *conjunction* $x \wedge y$ of two variables x and y is true when both x and y are true. This operator is called a *logical AND*.

- The *disjunction* $x \vee y$ of x and y is true when either one of the variables is true, or both. This operator is called a *logical OR*.
- The *negation* $\bar{x}$ of one variable x is true if and only if x is false. The operator is said to be a *logical NOT*, and the symbols x and $\bar{x}$ are called respectively *positive literals* and *negative literals*.

Boolean variables can be denoted by x, y, z, etcetera, as well as x_1, x_2, x_3, etcetera. We will use the latter notation when we need to specify Boolean formulas by their numbers.

The *satisfiability problems* , shortly *SAT problems*, are based on the following question: given a Boolean formula, is there an assignment for the variables for which the formula is true? In other words, can this formula be satisfied?
A brute-force approach is verifying each possible combination of assignments for each variable. As each variable has two possible values, the number of combinations is 2^n, where n is the number of variables. Therefore, it is not immediately obvious to which complexity class this problem belongs. In order to classify the complexity of SAT problems, we look at formulas that are in *conjunctive normal form* (shortly CNF). That is, the formulas is a conjuction of *clauses*, where each clause is a disjunction of literals. Each Boolean formula can be converted to CNF, an algorithm for doing this is given in [9, Section 4.6].
The simplest satisfiability problem is the *2-SAT problem*. In this case, each of the clauses contains no more than two literals. It turns out that there are algorithms which solve this problem in polynomial time [8, Section 2.2]. We will not treat this result any further in this thesis, as we are interested in the *3-SAT problem*. That is, each of the clauses may contain at most three literals.

Example 2.5.1. A short example is $(\bar{x} \vee y \vee z) \wedge (y \vee \bar{z} \vee w)$. It consists of the two clauses $\bar{x} \vee y \vee z$ and $y \vee \bar{z} \vee w$. It is true when, for instance, x is true, y is false, z is true and w is true. Therefore, this formula is satisfiable.

In [8, Section 2], the following result is given.

Theorem 2.5.1. The 3-SAT problem is NP-complete

This means that we can show NP-completeness for other problems, if we can reduce these problems to the 3-SAT problem in polynomial time. This will be the further goal in this chapter.

Chapter 3

Nonograms

3.1 Definition

This section is analogous to [6, Section 2]. In this thesis, we will focus on the alphabet $\Sigma = \{0, 1\}$. Consider the symbols 0 and 1 as values that represent different colours of pixels. We will refer to 1 as a *black pixel* and 0 as a *white pixel*. Additionally, the value of a pixel can be undetermined, especially at the beginning of the solving process (i.e. all pixels are undetermined). In that case, we take ? as the value of the pixel, and we take $\Gamma = \{0, 1, ?\}$ as the set of all possible pixel values. As we will have to refer to pixel values as either known or unknown, we call the values 0 and 1 (respectively white and black) *known values* and the value ? the *unknown value*.

A *description* d is an ordered sequence of positive integers $d_1, d_2, \ldots, d_k$ of length k, which could be empty as well. We take a finite string s over Σ, that is, $s = 0^{a_1}1^{a_2}0^{a_3}1^{a_4}\ldots 0^{a_{2n-1}}1^{a_{2n}}$ with integers $a_1, a_2, \ldots a_{2n}$ that are possibly equal to zero. We say that s *adheres* to a description d if s satisfies an expression of the form $s = 0^*1^{d_1}0^+1^{d_2}0^+\ldots 1^{d_k}0^*$. A description is said to be *consistent* with respect to a string $s \in \Gamma^\ell$ if

$$\left(\sum_{i=0}^{k} d_i\right) + k - 1 \leq \ell, \tag{3.1}$$

that is, the length of the string s cannot exceed ℓ.

An *image* I represents a grid of pixel values, i.e. $(I_{ij}) \in \Gamma^{m\times n}$. Therefore, this image may be (partially) filled (contains ones and zeros) or empty (only contains "?"-symbols). A *Nonogram description* D is contained of m row descriptions $r_1, r_2, \ldots, r_m$ and n column descriptions $c_1, c_2, \ldots, c_n$. The row and column descriptions r_i and c_j correspond to respectively the i-th and j-th row of the image I.

Definition 3.1.1. A Nonogram $N = \{I, D\}$ is an image I combined with a Nonogram description D.

We say that the image I *adheres* to a description D when each row and column of I adheres to its corresponding description. In order to reach this, I must at least contain solely values of Σ, and in such a way that each description is satisfied. This is the main objective of solving a Nonogram puzzle, and we will discuss this further in section 3.2.

A Nonogram N can be parted into *subnonograms* $N^{sub,1}$, $N^{sub,2}$ etcetera. The subnonogram $N^{sub} = (I^{sub}, D^{sub})$ consists of a *subimage* I^{sub} and natural numbers n_{start}, n_{end}, m_{start}, m_{end} which are at most equal to respectively n and m, for which any pixel $I_{i,j}$ with $n_{start} \leq i \leq n_{end}$ and $m_{start} \leq j \leq m_{end}$ is also contained in I^{sub}. and not in I^{sub} if this demand is not fulfilled. The (i, j)-th pixel of I becomes the $n_{start} + i - 1, m_{start} + j - 1$-th pixel of I^{sub}, such that the order of pixels is maintained.

The *subdescription* D^{sub} is constructed such that it adheres to the subimage I^{sub}. Of course we need to know which description integers do exactly match with the subimage I^{sub} In this thesis, we will only construct the entire Nonogram N from these subnonograms, instead of taking part of the image I and the description D, as we would not know which parts of the description are meant to fit in the subnonogram N^{sub}. To avoid confusion, we use the abbreviations $I^{sub,1}$, $I^{sub,2}$ etc. and $D^{sub,1}$, $D^{sub,2}$ to distinguish between subnonograms, although in this thesis we are mostly concerned with one subnonogram at once.

3.2 Problems

Looking at Nonograms, we can set up the following problems:

1. Existence: given a Nonogram N, does there exist a solution. In other words, is this Nonogram solvable?

2. Solving: given a solvable Nonogram N, how can we obtain a solution?

3. Uniqueness: given a solution for a Nonogram N, does there exist another solution?

Note that the solvability of a Nonogram infers merely the same problem as the existence of solutions, since we can decide whether the Nonogram is solvable when we have found a certain solution. Although, the nature of these problems is quite different. The existence problem is a decision problem, that is, the problem is answered by either a "yes" or a "no". The solvability problem however, is answered with a complete solution. On the other hand, determining that there exists a solution for a certain Nonogram, does not immediately mean that it is also solvable.

Imagine we adapt the Solving problem a little bit, such that it can also apply to Nonograms that are not solvable. If the Nonogram is not solvable, we will not obtain a solution. Of course we can only say the converse if there exists an optimal algorithm that solves each solvable Nonogram. If this is the case, and the algorithm does not return a complete solution, we can conclude that the Nonogram is not solvable. Therefore, we will combine the first two problems to

one. The latter problem is often referred to as the *Another Solution Problem*, and from now on we will denote this problem with ASP.

One can imagine the brute-force approach of each possible image I, and then verify whether this image adheres to the Nonogram description D. To begin with, each image contains $n \cdot m$ values, corresponding to the values of the pixels. Since each pixel can have two different values, we need to construct $2^{n \cdot m}$ candidate images. Without even mentioning the decision process, we can say that it cannot in any way be executed in polynomial time. This algorithm is only an example, and it does not imply that there does not exist an algorithm that solves the solvability problem in polynomial time.

Chapter 4

The Existence Problem

4.1 Non-deterministic constrained logic

The problem on which we have to focus is given in [4, Section 5.1], and is the following. Given a certain AND/OR constraint graph G with the properties and restrictions as given in section 2.2, and an edge e. Is there a sequence of moves on G that will eventually reverse e, such that the final configuration is legal and each edge is reversed at most once? This problem is called the *Bounded Nondeterministic Constraint Logic problem*, or shortly the *Bounded NCL problem*.

A Boolean formula in 3-CNF can represent an AND/OR constraint graph by the following constructive process:

1. Create CHOICE vertices for each variable, where the two initially inward vertices correspond to its positive and negative literals. The outward edge remains as a free loose edge. By doing this, it is not possible to reverse both of the inward edges. Reversing an edge corresponding to a literal will mean that a certain literal is given the value "true", which is impossible for both of the literals.

2. For each clause of three literals, create two OR vertices, and connect them with one edge. In this way, an outward edge of one OR is the inward edge of the other one. Together, the vertices have three outward edges, which have to correspond to each of the literals of one clause.

3. When a literal has been used more than once in the formula, connect it with FANOUT vertices in the following way.

    - Create a FANOUT when a literal appears at least twice in the formula. Its outward blue edge should be connected to the literal with a red-blue vertex.
    - For each additional occurrence, create another FANOUT. Connect its outward blue edge to one of the inward red edges of a FANOUT created before, with a red-blue vertex in between. The result will not depend on the choice of red edges or FANOUTs.
    - The number of free inward red edges should equal the number of occurrences of the chosen literal.

4. Connect the outward edges of each connected pair of OR vertices to their corresponding literals, using red-blue vertices. If the literal appears more than once, connect the outward edge to one inward red edge of its FANOUT structure, again using red-blue vertices.

5. When there are at least two clauses, create AND vertices in the following way:

   - Create an AND vertex, where the two outward red edges should correspond to a certain clause.
   - For each additional clause, create another AND-vertex, and connect its inward blue edge to one of the outward red edges of one of the previously created AND vertices.
   - The number of free outward red edges should equal the number of clauses.

6. For each clause, connect the inward blue edge of its corresponding OR to one of the outward red edges of an AND.

Note that the constraint graph G created in this way is not immediately AND/OR constraint. This problem can be omitted by replacing each CHOICE vertex and pair of red-blue vertices by an AND/OR equivalent subgraph, as given in section 2.2. In this way, we obtain an AND/OR constraint graph G', equivalent to G.

We will show this process with an example. Our objective is to create a certain AND/OR constraint graph G corresponding to the Boolean formula $(\bar{x} \vee y \vee z) \wedge (y \vee \bar{z} \vee w)$.

1. For instance, for the variables x, y, z and w, create three CHOICE vertices of which the inward edges correspond to respectively x and $\bar{x}$, y and $\bar{y}$, z and $\bar{z}$ and finally w and $\bar{w}$, see figure 4.1.

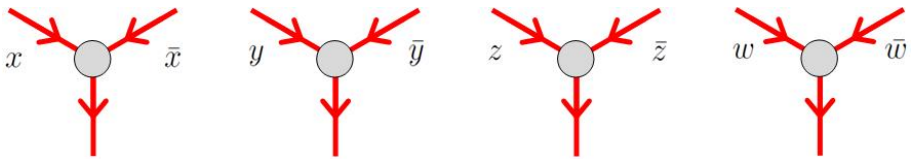


Figure 4.1: The CHOICE vertices corresponding to the variables

2. As there are two clauses, we create two pairs of OR vertices, and connect them with one edge in between, see figure 4.2.

3. Since the literal y is used more than once, we create a FANOUT for this literal, see figure 4.3.

4. We connect the input edges of the CHOICE and FANOUT vertices, which correspond to, to the correct OR vertices, see figure 4.4.

5. We only have two create a single AND vertex, as there are two clauses.

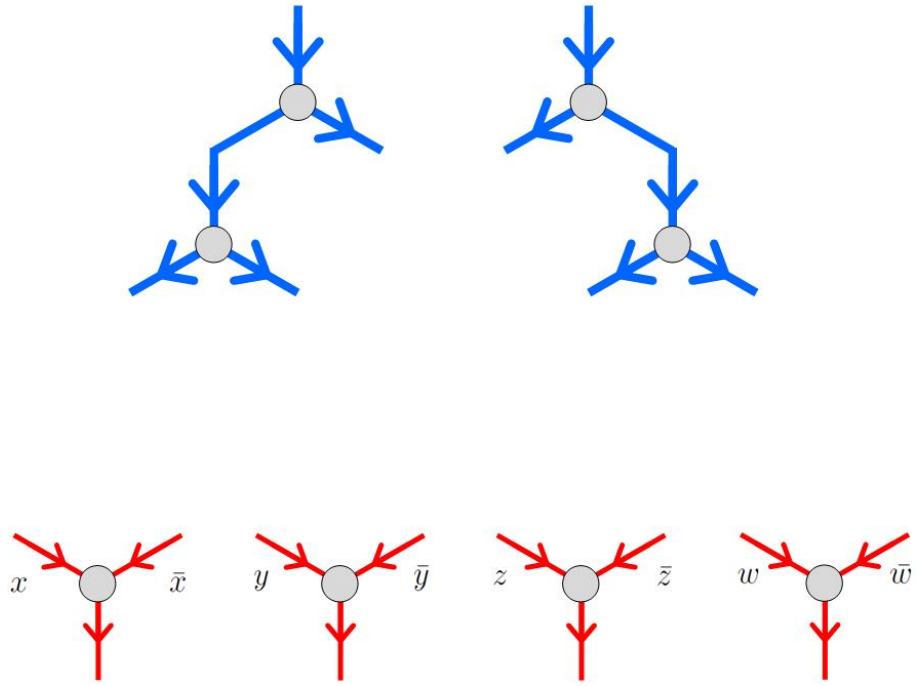


Figure 4.2: The OR vertices corresponding to the clauses

6. We connect it to the corresponding OR vertices. The final result has been drawn in figure 4.5.

One can think of a similar reduction from 2-SAT to the bounded NCL problem in polynomial time. As the 2-SAT problem is polynomial, it is completely pointless to reduce this problem to bounded NCL. As a P-problem does not tell us anything about the problem is it reduced to, we still do not know to which class the given problem belongs. As the 3-SAT problem is known to be NP-complete, we can conclude the following.

Theorem 4.1.1. The Bounded NCL problem is NP-complete.

Proof. Recall that the Bounded NCL problem contained of the question whether a certain set of legal moves could reverse a certain edge e, such that each edge is reversed at most once. If we construct an AND/OR constraint graph G corresponding to a certain Boolean formula, as described above, we can apply this problem to G. The objective is to reverse the output edge, that is, the edge that correspond to the entire Boolean formula. As we saw in section 2.2, the sequence of edges that is reversed must satisfy the following rules:

- The inward edge of all AND vertices will have to be reversed. Therefore, all their outward vertices must be reversed as well (in order to satisfy the constraints).
- The inward edge of an OR vertex may need to be reversed. In such a case, at least one of the outward edges connected to the OR vertices must be reversed.

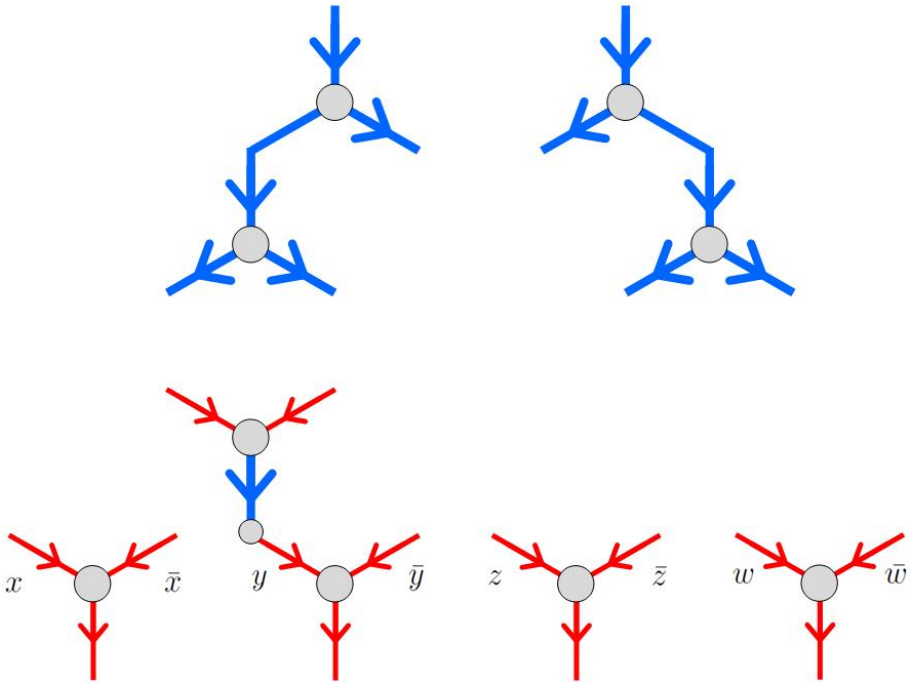


Figure 4.3: The added FANOUT vertex

- The inward edges of the CHOICE vertices cannot both be reversed, so at most one of the edges corresponding to the literals can be reversed.

If these rules cannot be satisfied, it means that the Boolean formula cannot be satisfied. With this method, we reduced the 3-SAT problem to the Bounded NCL problem. As 3-SAT is NP-complete, and therefore NP-hard, we can say that Bounded NCL is NP-hard as well.

Furthermore, we know that Bounded NCL is in NP. Given a certain sequence $e_1, e_2, \ldots, e_p$ of edge reversals, we can verify in polynomial time whether it reverses a certain edge e legally. To do that, we must verify for each reversal whether it is legal. That is, for each vertex we have to verify that its constraint is satisfied. We can do this by adding the edge weights of all connected inward edges of a certain vertex v, which can be done in polynomial time. For n vertices, this will take $n \cdot p$ calculations, since we have p reversal steps. In the end we have to verify whether the target edge has been reversed (which is either "true" or "false"). All together, the process takes polynomial time, and is therefore in NP.

The Bounded NCL problem is both NP and NP-hard. Therefore we can conclude that the Bounded NCL problem is NP-complete. □

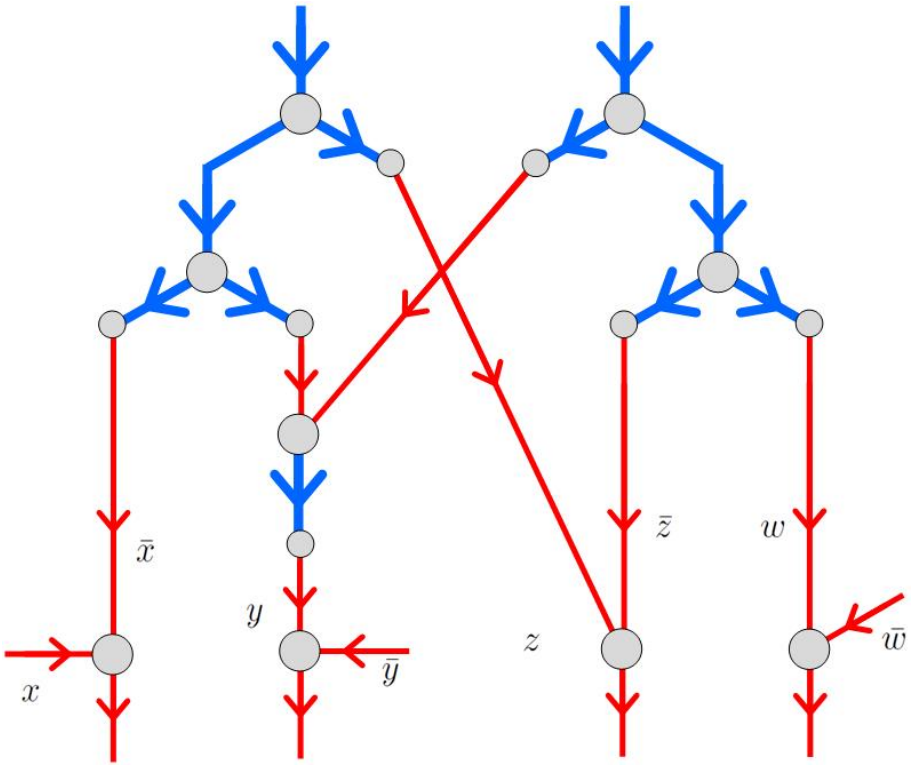


Figure 4.4: The connection of all the corresponding edges

4.2 Complexity of the solution problem

Now that we have shown in section 4.1 that the Planar Bounded NCL problem is NP-complete, we will focus on determining the complexity of the problems concerning Nonograms, given in section 3.2. In this section, we will show that the Planar Bounded NCL problem can be reduced to the Nonogram solution problem, and therefore, it is NP-complete as well.

We will construct a reduction by creating a framework for possible subnonograms, and linking several situations in the Nonogram solving process to the constaint logic gadgets, described in 2.2. In this way, we show that the Nonogram Solution problem is as least as hard as the Bounded NCL problem, and is therefore NP-hard. In addition, we can show that the Solution problem is NP, and therefore it is NP-complete.

We reduce the Nonogram solution problem to the Bounded NCL problem in the following way: we create a Nonogram in which we can perform pixel colourings that are equivalent to reversing edges in a planar constraint graph situations, and show that they correspond to eventually reversing an edge connected to an AND or an OR vertex. Note that we only have to represent a constraint graph that corresponds to a 3-SAT reduction, shown in section 4.1, as this would mean

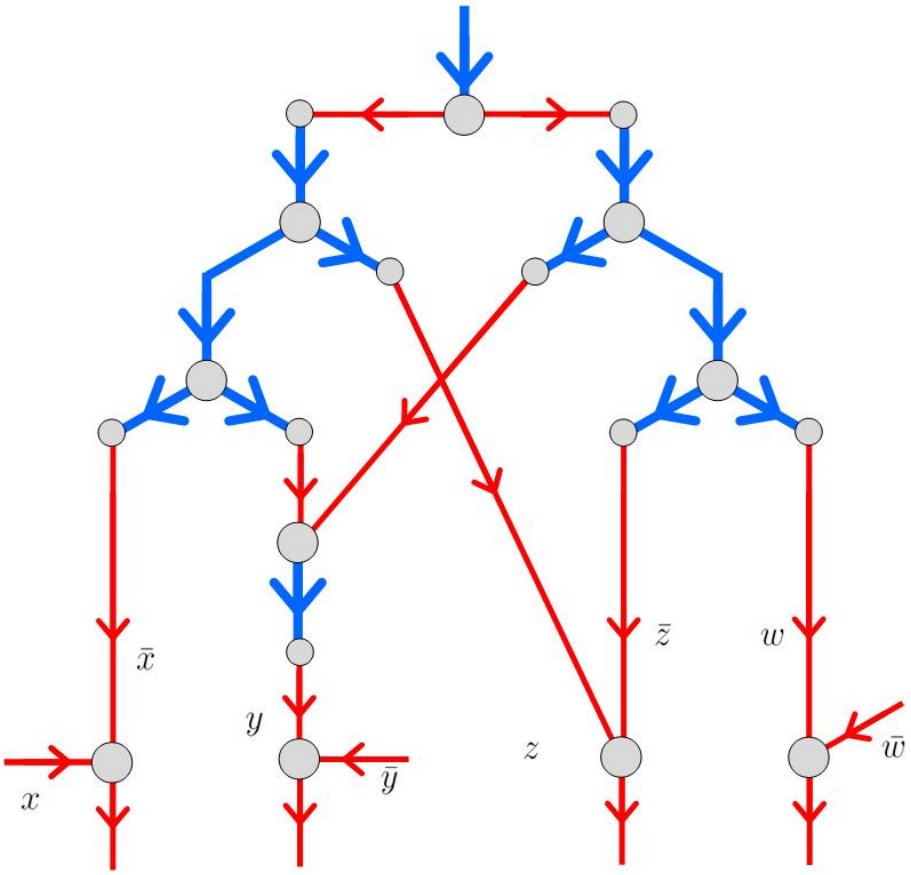


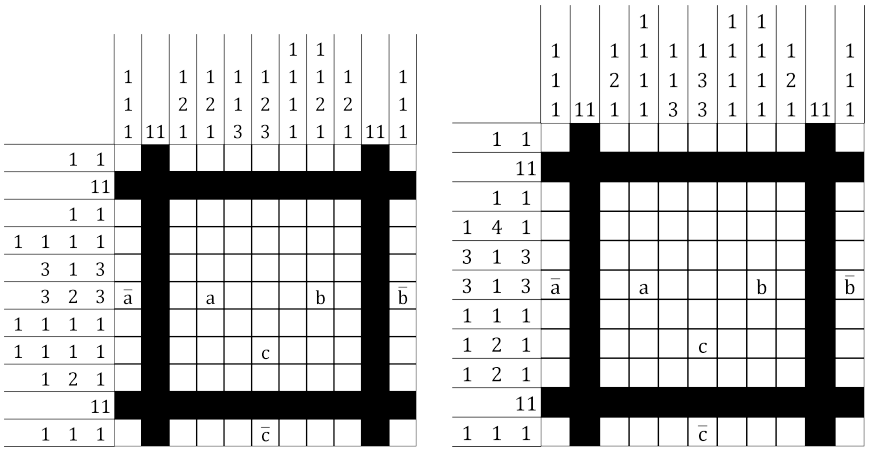
Figure 4.5: The final result of the graph G.

that the Nonogram solving problem is NP-hard, which is what we need to show.

This proving method is based on [15, Section 7.2]. First, we construct a framework of horizontal and vertical separation lines of black pixels, and empty square grids in between. These separation lines fill an entire row or column with black pixels, thus their description contains only of m or n for respectively horizontal and vertical lines. This means that these lines can always be filled completely based on solely their description.

For the empty squares, we design subnonograms that represent either a vertex of a constraint graph, or a gadget to connect vertices to each other. The AND gadget can be represented as in figure 4.6a. The variables a and c correspond to the red edges of an AND vertex, and b corresponds to the blue one. For each edge, one has the objective to either colour pixel a or pixel $\bar{a}$, but not both of them (similar for the edges b and c). The barred variables correspond to edges pointing outward with respect to the AND vertex. Its possible solutions are given in 4.7. Out of these we can conclude that the pixel b can be filled if and only if the pixels a and c are filled, which means that solving this gadget

corresponds to reversing and edge of an AND vertex. Note that the solutions sketched in figures 4.7b and 4.7c are not mentioned as possible configurations of the AND vertex in section 2.2, although these are allowed both in this structure and the vertex.



(a) The AND gadget (b) The OR gadget

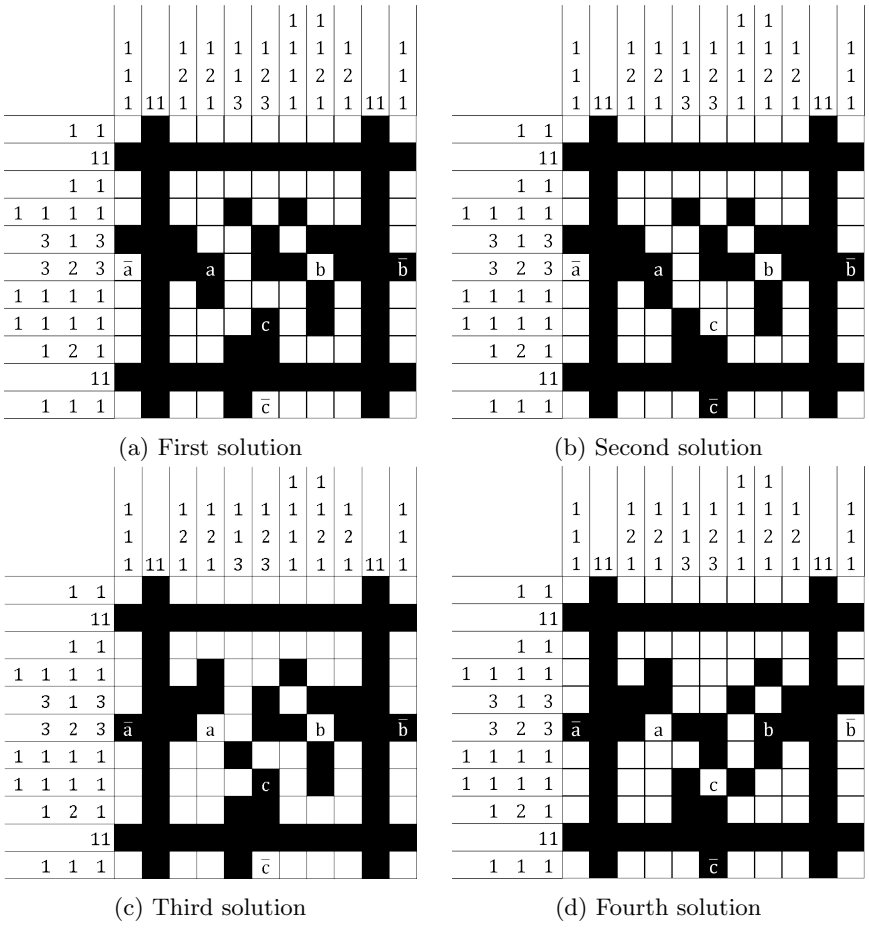
Figure 4.6: Constraint logic gadgets applied to Nonograms.

The OR gadget can be represented as in figure 4.6b. All the variables a, b and c, correspond to a blue edge of the OR vertex. Again, one has to fill either one of a and $\bar{a}$, but not both of them, and similar for b and c. Its possible solutions are given in 4.8. Therefore, solving this gadget corresponds to reversing an edge of an OR vertex.

Note that edge colours do not matter, as their properties are already contained in the Nonogram gadgets. For this reason, we do not have to worry about red-blue vertices. The CHOICE and FANOUT gadgets are not explicitly contained in the definition of an AND/OR constraint graph. However, it is convenient to replace these vertices by equivalent subnonograms as well, since these vertices occur specifically in the reduction from 3-SAT to Bounded NCL. The initial subnonograms corresponding to the CHOICE and FANOUT vertices and their solutions are shown in respectively 4.9 and 4.10.

However, this process does not guarantee that any type of constraint graph can be represented by such a Nonogram structure, since:

- There could be a shortage of space to fill with gadgets
- It is not sure whether there are cycles
- In the construction of a constraint graph according to section 4.1, we are allowed to let vertices cross each other. We have shown in section 2.2 that it is possible to interchange a crossover with a subgraph containing



(a) First solution

(b) Second solution

(c) Third solution

(d) Fourth solution

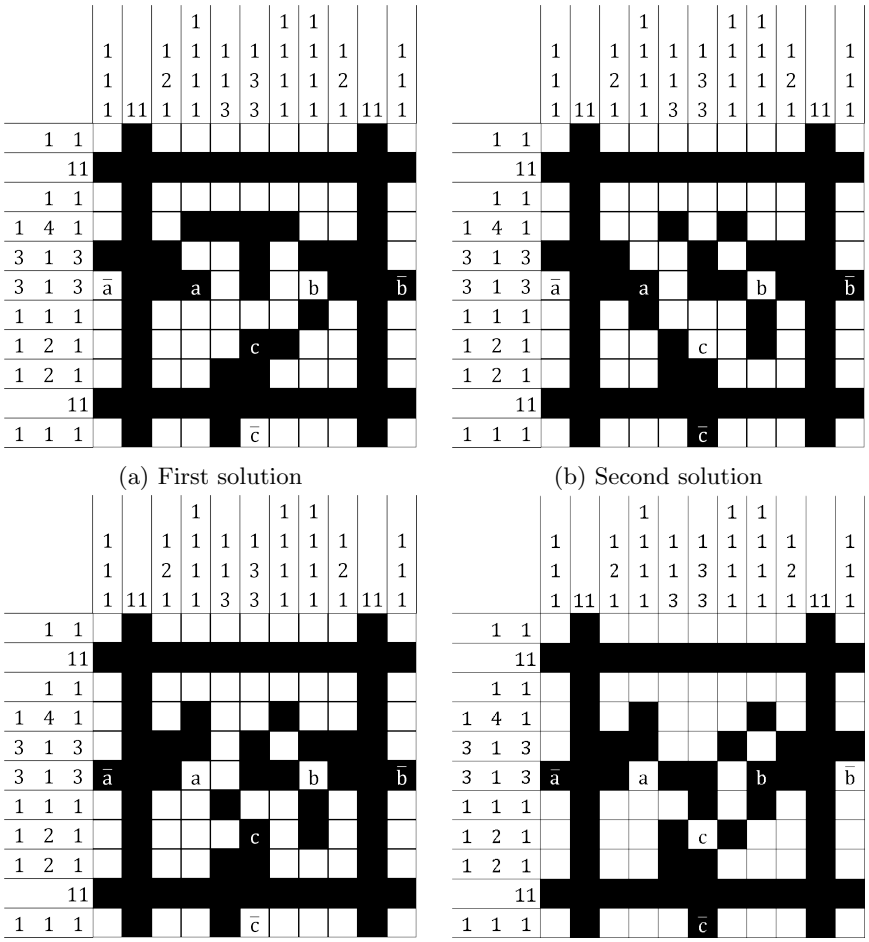
Figure 4.7: The four possible solutions of the AND gadget.

AND/OR vertices, although it is not to say that such a subgraph can be contained in this Nonogram framework.

- The loose edges are not contained.

To omit these problems, we create a few additional gadgets. First, we create a gadget that transfers a signal from one side to another. In this structure, the pixel d has to be coloured black if and only if d' is coloured black, and the same holds for $\bar{d}$ and $\bar{d}'$. This structure and its solutions are shown in figure 4.11. By mirroring and flipping, one can obtain the structure that is necessary to fit in the framework.

Constraint graphs representing 3-CNF formulas are *acyclic*. This means that there cannot be a cycle of edges, following the definition of section 2.1. To allow edges to intersect each other (which is not possible in a planar situation such as Nonogram solving, we can create a crossover gadget. Recall from section 2.2



(a) First solution

(b) Second solution

(c) Third solution

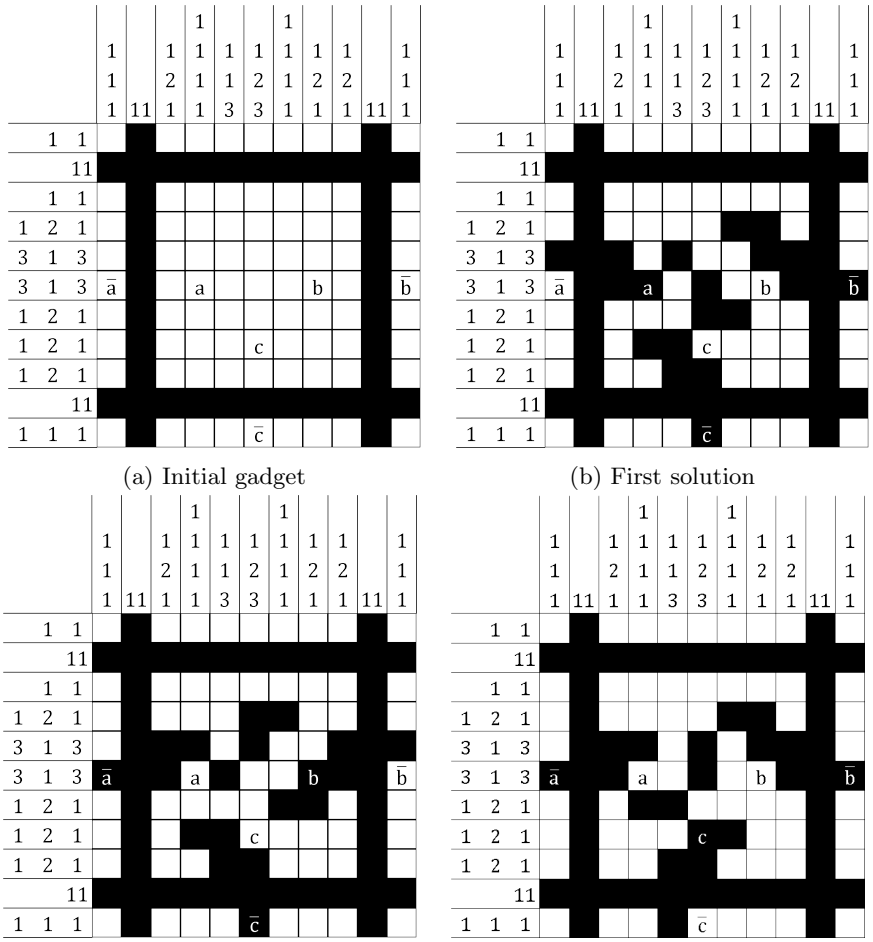
(d) Fourth solution

Figure 4.8: The four possible solutions of the OR gadget.

that it is possible to replace an intersection of edges by an equivalent AND/OR constraint graph, although we can create a gadget that is more convenient. This gadget is shown in figure 4.12, and its solutions in figure 4.13. Note that this gadget has more or less the same properties as the Distance gadget. Both the information from d and e is transferred equally to both d' and e'. Therefore, this gadget is a useful substitute for the crossover gadgets for constraint graphs in section 2.2.

The gadgets in figure 4.14 represent loose edges, either free or constraint.

Having defined all the subnonograms equivalent to NCL situations, we can fill the main framework in the following way:

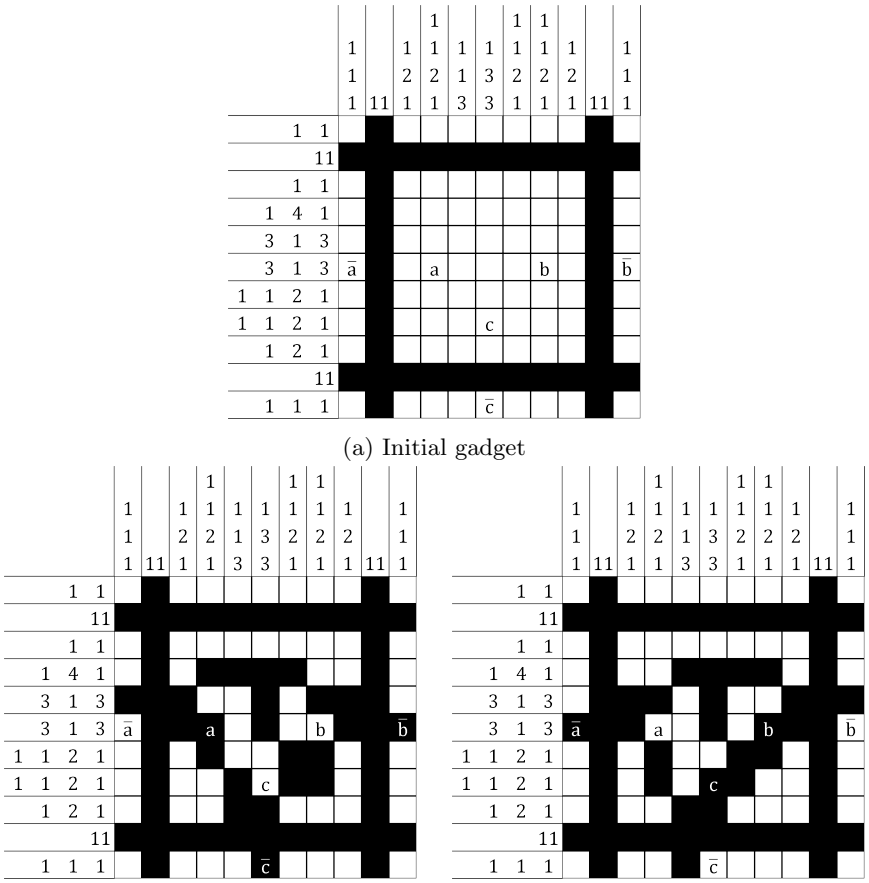


(a) Initial gadget (b) First solution

(c) Second solution (d) Third solution

Figure 4.9: The initial CHOICE gadget and its solutions.

1. Fill the uppermost empty square with a loose edge gadget. This represents the first red edge
2. Fill adjacent empty squares with distance gadgets.
3. Create AND, OR, CHOICE and FANOUT gadgets according to the graph created in section 4.1, each with a distance gadget in between. This is not necessarily required, although we now have the possibility to connect further
4. If we need more empty squares in between, create an additional row or column of empty squares (not to be confused with pixels) and a separation line. Then, connect each pair of ends with another distance gadget
5. If we need to intersect two distance gadget, replace this by a crossover gadget.



(a) Initial gadget

(b) First solution

(c) Second solution

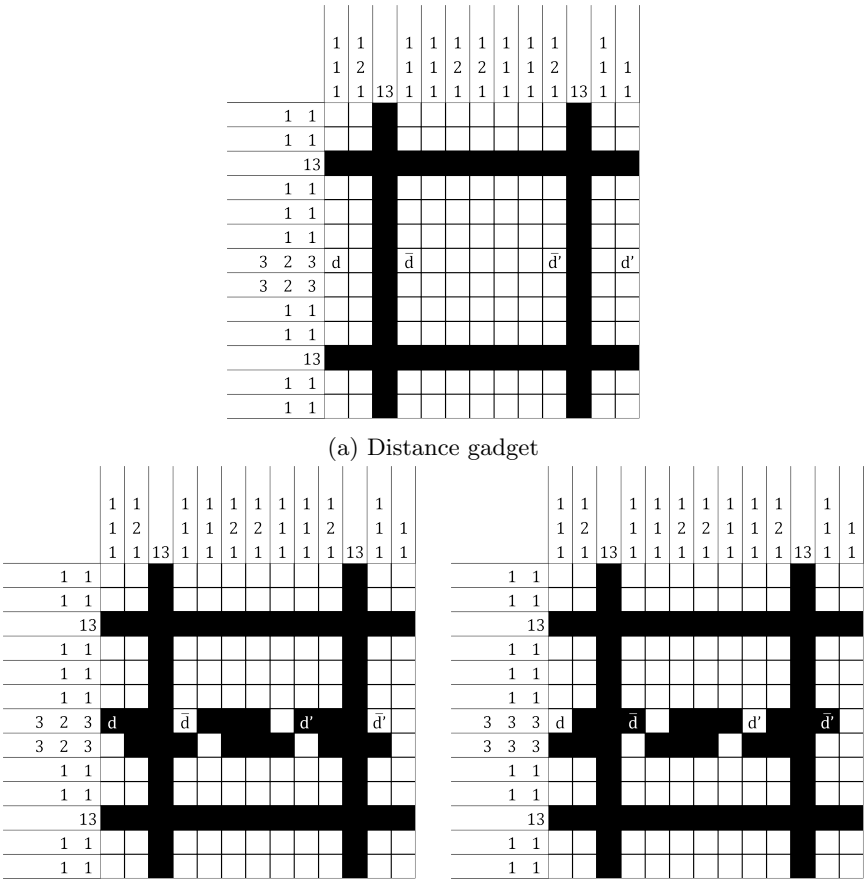
Figure 4.10: The initial FANOUT gadget and its solutions.

6. Finish with terminate loose edges, which correspond to the literals of the 3-CNF formula.

It is not immediately clear whether this process can be done in polynomial time, especially when stretching the distance between vertex gadgets to create space. When we have a vertex set V with cardinality $|V|$, we have to fill $|V|$ distance gadgets between any vertex in order to be able to connect each edge to another. Note that a crossover gadget is similar to a distance gadget. Therefore, the number of distance steps equals at most $|V|^2$. We can now conclude the following.

Theorem 4.2.1. The Nonogram solving problem is NP-complete

Proof. We showed above that the Planar Bounded NCL problem can be reduced in polynomial time to the Nonogram solution problem. The created Nonogram has no solution if and only if the initial edge cannot be reversed. Planar Bounded NCL is NP-complete, and therefore both NP and NP-hard. This means that the Nonogram solution problem is NP-hard as well.



(a) Distance gadget

(b) First solution

(c) Second solution

Figure 4.11: The Distance gadget, with initial state and two solutions.

It suffices to show furthermore that the Nonogram Solving problem is NP. To achieve this, we have to create an algorithm that verifies a solution image I, and determines whether it adheres to a given Nonogram description D,given a certain Nonogram $N = (D, I)$. The description D consists of n row and m column descriptions, respectively $r_1, r_2, \ldots, r_n$ and $c_1, c_2, \ldots, c_m$. The solution image I can be represented by an $n \times m$-matrix. Therefore, a certain row and column vector of I can be converted into a string s. This has to be of the form $0^*1^+0^+1^+0^+ \ldots 1^+0^*$, according to section 3.1.

Recall from section 3.1 that I is a solution of N if all rows and columns of I adhere to their corresponding descriptions. To decide whether I is a solution, we have to verify that for a certain description $d = \{d_1, d_2, \ldots, d_k\}$, the corresponding row or column string is of the form $s = 0^*1^{d_1}0^+1^{d_2}0^+ \ldots 1^{d_k}0^*$. This can be done in polynomial time by the algorithm `nonogramverify`, and can be found in appendix A.

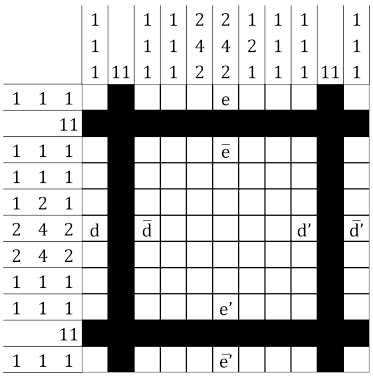
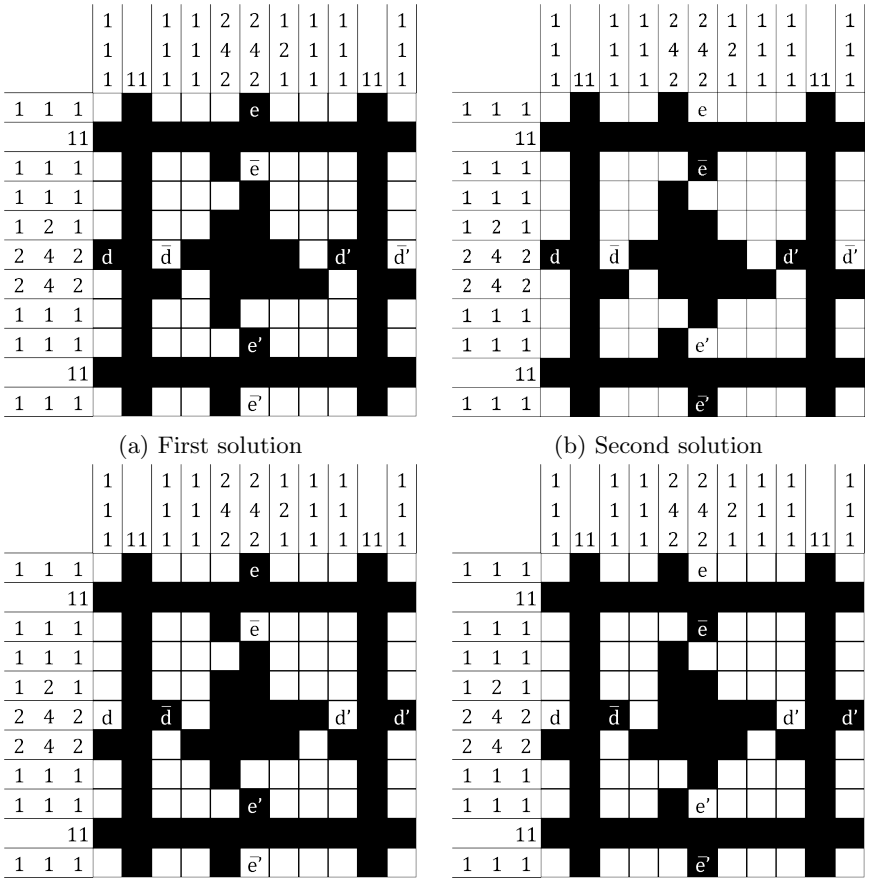


Figure 4.12: The crossover gadget in a Nonogram situation.



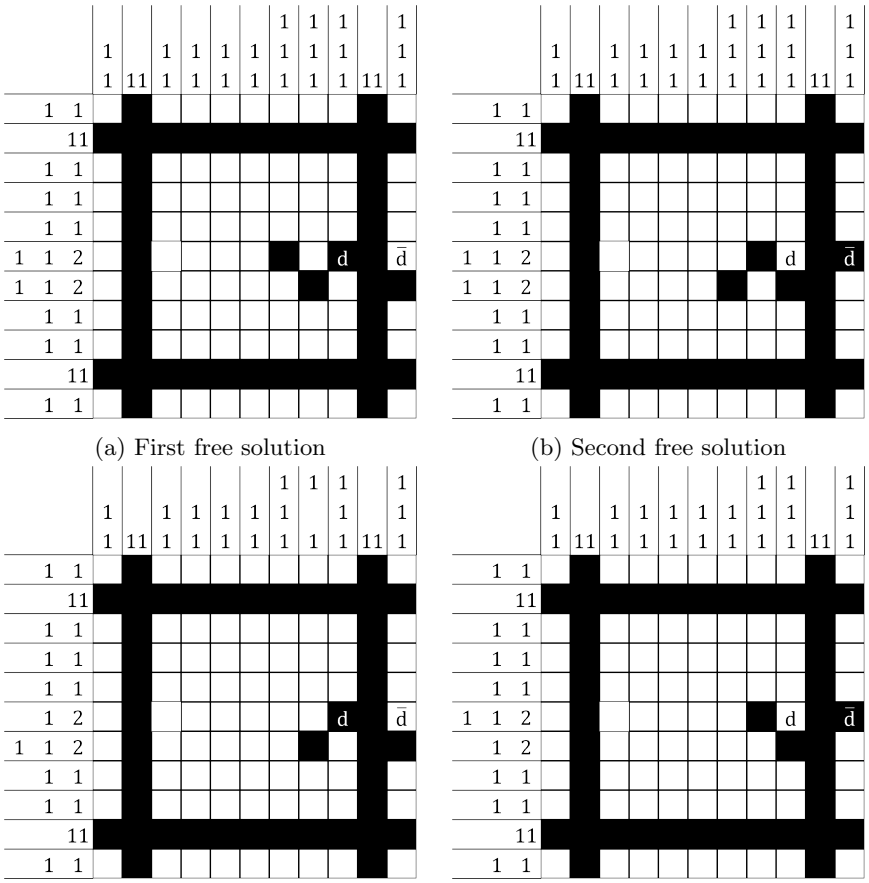
(a) First solution

(b) Second solution

(c) Third solution

(d) Fourth solution

Figure 4.13: Solutions for the crossover gadget.



(a) First free solution

(b) Second free solution

(c) First fixed solution

(d) Second fixed solution

Figure 4.14: The Loose edge gadgets, either free or fixed.

Therefore, the Nonogram solving problem is NP-complete. □

Chapter 5

The Uniqueness Problem

5.1 The three dimensional matching problem

In order to determine the complexity of the Another Solution Problem for Nonograms (which we will hereafter refer to as the Nonogram ASP), we introduce another problem for which we can show it is NP-complete.

Given three disjoint sets X, Y and Z with the same number of elements q, and a set $T \subseteq X \times Y \times Z$ of ordered triples. Does there exist a subset $T' \subseteq T$ that consists of exactly q elements, such that each element of respectively X, Y and Z appears in only one of its triples? Such a subset T' is called a *matching* of T.

Again, it is not immediately obvious to which complexity class this problem belongs. We could create each possible subset T' of T that has q triples, and verify whether it is a matching for T. When the sets X, Y and Z consist of q elements, the set T will consist of at most q^3 triples. The number of possible subsets T' with q triples will be at most

$$\binom{q^3}{q} = \frac{q^3!}{q! \cdot (q^3 - q)!}.$$

Set the input size of the instance equal to $n = q^3 + 3q$, the number of possible subsets T' cannot be bounded above by a finite polynomial $f(n)$.

To reduce the 3-SAT problem to the 3DM problem by a polynomial time reduction, We construct three sets X, Y and Z and a set of ordered triples $T \subseteq X \times Y \times Z$, such that the 3-SAT requirement is satisfied if and only if 3DM is satisfied.

Given a Boolean formula in 3-CNF with clauses $\{C_1, \ldots, C_m\}$ and variables $\{x_1, \ldots, x_n\}$. For each variable x_i we create a set of *core elements*

$$A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,2m-1}, a_{i,2m}\} \tag{5.1}$$

and *tip elements* or simply *tips*

$$B_i = \{b_{i,1}, b_{i,2}, \ldots, b_{i,2m-1}, b_{i,2m}\}.$$

From this, we create triples $t_{i,j} = \{a_{i,j}, a_{i,j+1 \mod 2m}, b_{i,j}\}$. The index $j+1$ mod $2m$ has the purpose not to exceed the number of elements of A, which means the last triple is of the form $t_{i,2k} = \{a_{i,2m}, a_{i,1}, b_{i,2m}\}$. Therefore, each element of A is contained in exactly two triples. We will hereafter refer to triples $t_{i,j}$ and tips $b_{i,j}$ with even j as *even triples* (and *tips*), and with odd j as *odd triples* (and *tips*).
For instance, we create a matching situation for the formula $(x_1 \vee \bar{x_2} \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee x_4)$. This means we have four variables x_1, x_2, x_3, x_4 and two clauses $C_1 = (x_1 \vee \bar{x_2} \vee x_3)$ and $C_2 = (x_2 \vee \bar{x_3} \vee x^4)$. Therefore, we have our sets $A = \{a_{i,j}, 1 \leq i \leq 4, 1 \leq j \leq 4\}$ and $B = \{b_{i,j}, 1 \leq i \leq 4, 1 \leq j \leq 4\}$. The corresponding triples are $t_{i,j} = \{a_{i,j}, a_{i,j+1 \mod 4}, b_{i,j}\}$ with $1 \leq i \leq 4$ and $1 \leq j \leq 4$. This is shown in figure 5.1

The idea is to either adopt odd triples or even triples in T'. This will correspond to assigning $x_i = 0$ respectively $x_i = 1$ to a certain variable. For x_1, this is shown in figure 5.2.

For each clause C_k we create a couple c_k, c'_k. Then we combine these couples with tips $b_{i,j}$ in the following way:

- A triple $t_{c_k,i} = \{c_k, c'_k, b_{i,j}\}$ with odd j is made when x_i appears in clause C_k.
- A triple $t_{c_k,i} = \{c_k, c'_k, b_{i,j}\}$ with even j is made when $\bar{x_i}$ appears in clause C_k.

This is always possible, since we have exactly m clauses, and for each variable m odd tips and m even tips. We will hereafter refer to these triples as *clause triples*. Adopting such a triple in T' will correspond to satisfying a literal. If we are not able to adopt at least one of these triples, it means we cannot assign a truth value to one the literals (and therefore the clause) to render the Boolean formula satisfied. Even though more than one literal is allowed to be satisfied in one clause in the 3-SAT problem, we will take up at most one clause triple for each clause, as it is the purpose of 3DM to have all elements occurred only once.

Since we have two clauses, we create the couples c_1, c'_1 and c_2, c'_2. For the first clause, we need triples $t_{c_1,1}, t_{c_1,2}, t_{c_1,3}$ with p_1, p'_1 and respectively $b_{1,1}$, $b_{2,2}$ and $b_{3,1}$. For the second clause, we need $c_{2,1}, c_{2,2}, c_{2,3}$ with p_2, p'_2 and respectively $b_{2,1}$, $b_{3,2}$ and $b_{4,1}$. Note that for any $b_{i,j}$ the odd numbers j can be interchanged with any other odd number, and the same holds for the even numbers j. The result is shown in figure 5.3.

In the 3DM problem, all elements have to occur exactly once throughout all triples. When adding triples to T', the tips $b_{i,j}$ that are not contained in one of the literals cannot be contained in a triple. To resolve this, we create additional couples d_l, d'_l, with $1 \leq l \leq 2n - k$ and infer each tip $b_{i,j}$. We have

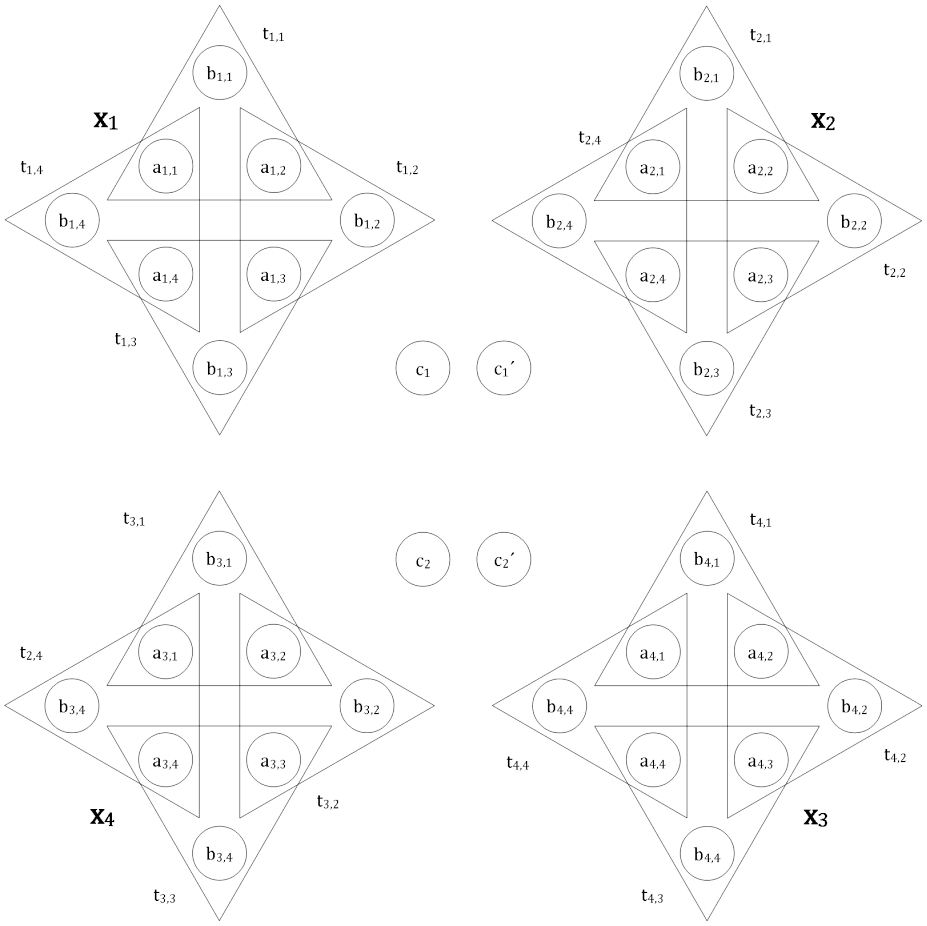
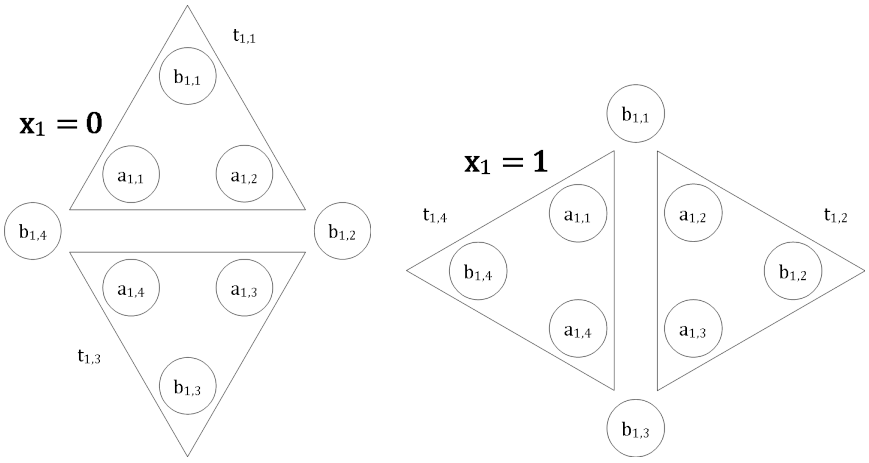


Figure 5.1: The core and tip elements for $(x_1 \vee \bar{x_2} \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee x^4)$.

to use such an extensive method, as we do not know which elements of B are going to be contained in a clause triple, and which ones are not. In this way, we are always ensured of a possible matching, as long as the clauses are satisfied.

In our example, we create couples p_m, p'_m with $3 \leq m \leq 8$ for the remaining elements of B, and create triples $\{p_m, p'_m, b_{i,j}\}$ for every b_i, j. The set of triples T consists of all triples described until now. For the clearness of the images, this is not sketched.

We say that a variable x_i is assigned TRUE if we insert the even triples in our matching, and FALSE if we add the odd ones. This means we can add a clause triple containing $b_{i,j}$, with j odd, only if for the corresponding variable $x_i = 1$, and similarly $\bar{x}_i = 0$. If we cannot add one of the clause triples, then we cannot safisfy the clause, and therefore the entire formula. This corresponds to the situation for the 3DM problem where not all elements of the sets can be contained in exactly one triple.



(a) Assigning FALSE to x_1 (b) Assigning TRUE to x_1

Figure 5.2: The two possible choices of core triples.

Now we try to determine our matching T'. There are several options, as long as the clauses C_1 and C_2 are satisfied. Two possible matchings, without the cleanup elements, as shown in figures 5.4 and 5.5. We can see that in each case, at least one clause triple is contained in the matching T' for each clause, which means that we can assign the variables x_1, x_2, x_3, x_4 in such a way that the formula is satisfied.
We now identify our sets X, Y and Z in the 3DM problem as follows:

- $X = \{a_{i,j}, j \texttt{ odd }\} \cup \{c_k\} \cup \{d_l\}$,
- $Y = \{a_{i,j}, j \texttt{ even }\} \cup \{c'_k\} \cup \{d_l\}$,
- $Z = \{b_{i,j}\}$.

Theorem 5.1.1. The 3DM problem is NP-complete

Proof. This proof is analogous to the proof in [2, Section 8.6].

We continue the steps sketched above. The last step is to show that 3-SAT is indeed reduced to 3DM in this way. If it is possible to satisfy each clause in a Boolean formula, it means each variable can be given a value such that the Boolean formula is satisfied, and therefore we can adopt a clause triple in T' for each clause. For a variable x_i, either all the even tips or odd tips are contained in a triple together with elements of A, and the other ones are contained in a triple with either c_k, c'_k or d_l, d'_l pairs. Therefore, each element of X, Y and Z is contained in exactly one triple, which satisfies the 3DM problem.

If it is however not possible satisfy the formula, it means that there is at least variable x_i for which we cannot find a suitable value. For the 3DM situation, it means that we are not allowed to contain both an odd and and even tip in a

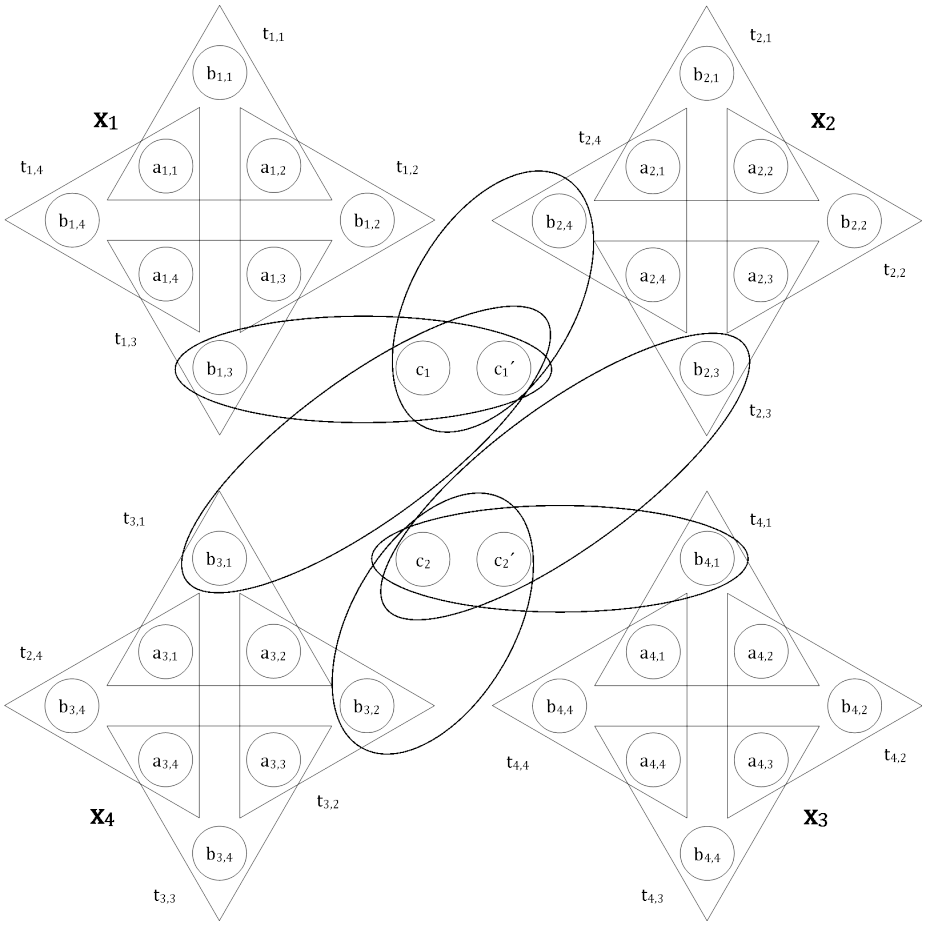


Figure 5.3: The additional clause elements and triples for $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee x_4)$.

clause triple. If this is the case for all three literals in a clause, it means that this clause cannot be satisfied, and not any triple with c_k and c'_k can be added to T'. Therefore, the 3DM problem cannot be satisfied.

It is to be verified by the reader that all reduction steps can be executed within polynomial time, which means the 3DM problem is NP-hard. The remainder is to show that the 3DM problem is NP. For the verification of a single candidate matching T', we only have to verify whether each element of X, Y and Z occurs only once throughout T'. This means that the 3DM problem is NP, and consequently, NP-complete. □

5.2 Complexity of the uniqueness problem

According to [11, Section 1], a *parsimonious reduction* is a reduction from problem A to B, with an algorithm that links the outcomes of A bijectively to

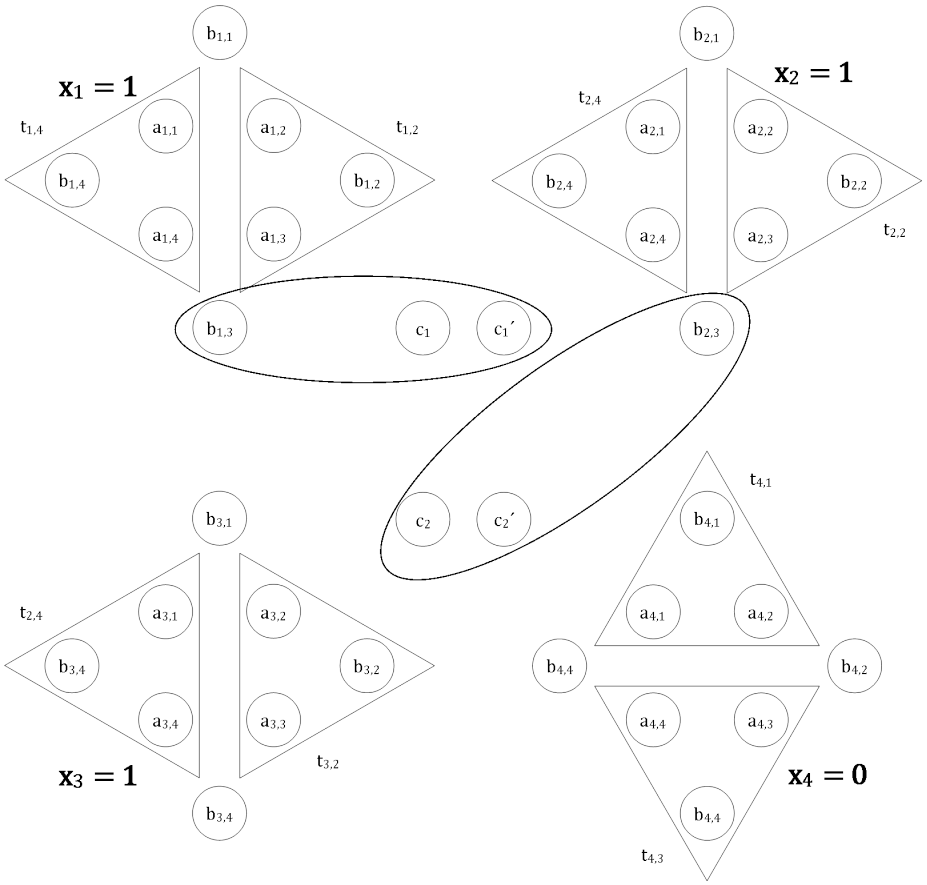


Figure 5.4: A possible matching T' for T

equivalent outcomes of B. Therefore, it preserves the number of solutions between the problems A and B. In other words, problem B has more than one solution whenever problem A has. Recall that the ASP for a problem means we have to verify whether there exists another solution for the given description. Therefore, we can conclude the following: If problem A can be reduced to problem B solution problem with a parsimonious reduction, and the ASP for problem A is NP-hard, then ASP for problem B is also NP-hard.

Theorem 5.2.1. The ASP for 3DM is NP-complete

Proof. One can imagine that the ASP for 3DM is similar to 3DM itself, and therefore can be reduced to 3DM. The details of the proof can be found in [11, Section 4] □

Theorem 5.2.2. The Nonogram uniqueness problem is NP-complete

The details of this proof can be found in [11, Section 3]. We will only sketch the outlines here.
We show NP-completeness by reducing the 3DM problem to the Nonogram existence problem. We create a Nonogram situation in which we obtain the same

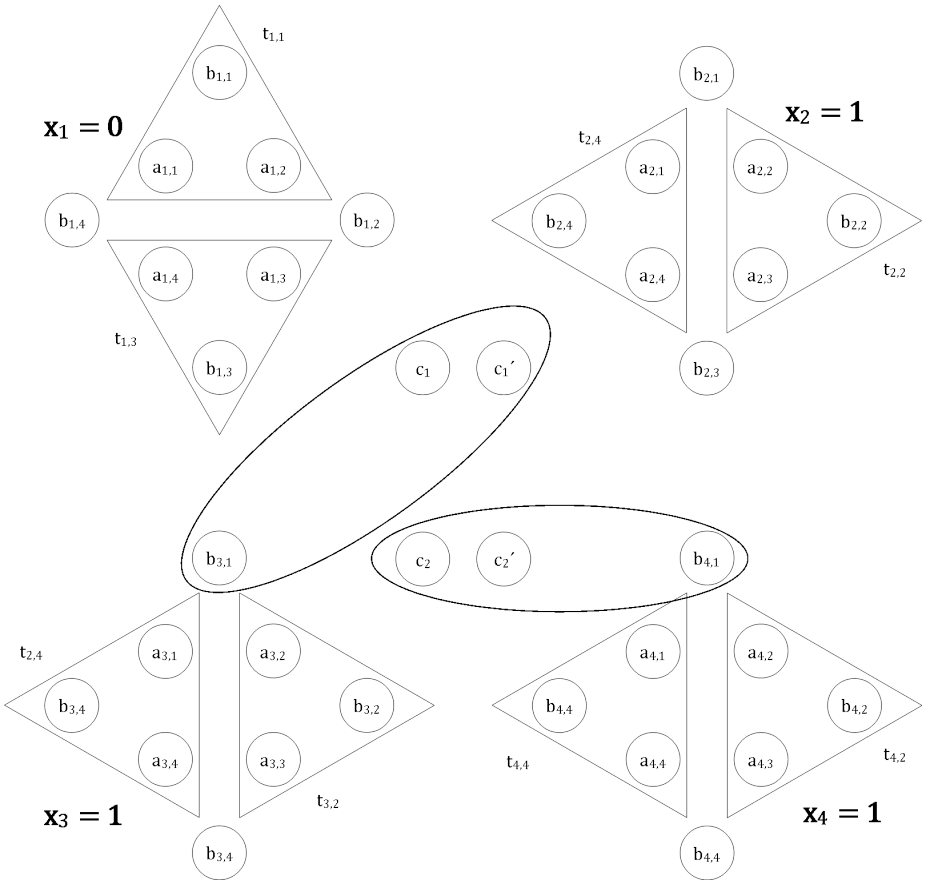


Figure 5.5: Another possible matching T' for T

number of results as in 3DM, such that we have a parsimonious reduction. Given three disjoint sets $X = \{x_1, \ldots, x_q\}$, $Y = \{y_1, \ldots, y_q\}$ and $Z = \{z_1, \ldots, z_q\}$, which all have q elements, and a set $T \subseteq X \times Y \times Z$ that consists of n triples, the following Nonogram $N = (D, I)$ is constructed:

1. The image I has a size of $2n$ rows and $6q + 2$ columns.
2. Except for the first and last column, each adjacent two columns correspond to a certain element of either X, Y or Z.
3. Each odd row corresponds to a certain triple of M, in the following way: the first and last pixels are coloured black, and there have to be description integers, such that there is a continuous row of black pixels between each pair of columns corresponding to elements inside the mentioned triple.
4. Each even row consists of white pixels
5. The description of each column consists of a number of ones, such that the image is satisfied

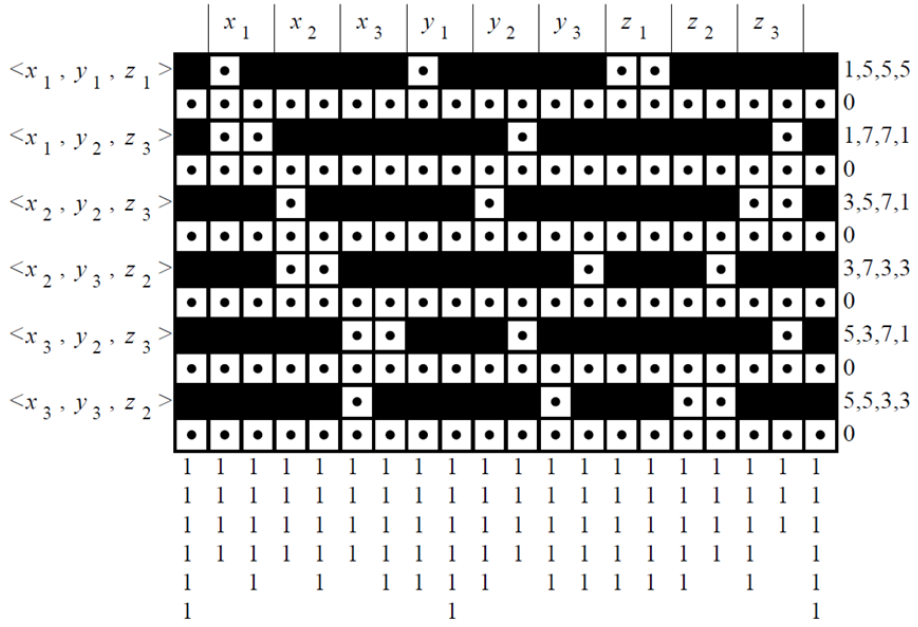


Figure 5.6: The example Nonogram for reduction from 3DM, taken from [11, Section 3]

An example with $q = 3$ is shown in figure 5.6
When it is possible to find another matching T', we will have an additional solution for the Nonogram N. This means that the number of possible matchings for X, Y and Z equals the number of solutions for our Nonogram. Therefore, we have a parsimonious reduction from 3DM to Nonogram.

Chapter 6

Solving Nonograms

Now that we have determined the complexity of both the solution and uniqueness problems for Nonograms in general, we can focus on the solving process itself. This solving process is NP-complete, as we saw in section 4.2. In this chapter, we will show several possible solving steps. Finally, we will apply several known examples of Nonograms to different solvers, and compare the outcome.
Apart from Nonograms that either have a unique solution or not, we distinguish two kinds of Nonograms here. *Simple Nonograms* can be solved by analysing each row or column separately. If this is not possible, we either have to look at the information from other rows or columns, or start probing for a solution. In this section, we look at solving steps applying to one row or column at once, so they mostly apply for simple Nonograms. Other Nonograms may have a few pixels assigned, although the solving process will terminate after a while.

6.1 Depth first search (DFS)

The key for depth first search, is analysing all the different possibilities in a certain row or column, or possibly the entire image, and accordingly determining which pixels have to or cannot be coloured black.
As far as possible, we will try to determine as many pixel values as possible by logical determination. If we get stuck, we can guess certain pixel values and abandon our decision when this leads to a contradiction.

6.1.1 Overlap

The first step in solving Nonograms is determining which pixels in a row or column are forced to be colored black, resulting from its description. Recall from section 3.1 that a row or column description $d = d_1, d_2, \ldots, d_k$ is consistent when

$$k - 1 + \sum_{i=0}^{k} d_i \leq \ell. \tag{6.1}$$

When there is equality, it means that the sequences of black pixels fit exactly in a row or column, and in that case we immediately have the solution for the

entire row or column. However, when there is no equality, it is still possible to colour several pixels black.

A Matlab script for colouring pixels black in that way, is named `overlap` and can be found in appendix A.

6.1.2 Determination by combinatorics

After having determined the first black (and possibly white) pixels, we have to focus on other solving processes, in order to determine additional black and white pixels. In [13, Section III.1], the following idea is sketched for determining additional black pixels using combinatorics. Find all possible combinations for a row or column. Then filter out all of the candidate solutions that do not adhere to the description, based on the pixels that already have been given a known value. If there are any pixels that always have to be coloured either black or white throughout all possibilities, assign these values to these pixels. We will extend this idea in this section.
First we have to find all candidate solutions in the following way:

1. Given a row or column description $d = d_1, d_2, \ldots, d_k$, we can compute the minimal space S_0 that has to be filled. This is equal to $\sum_{i=0}^{k} d_i + k - 1$, and corresponds to separating each connected sequence of black pixels 1^{d_i} by only one white pixel. Then we find the *free space* $F_0 = \ell - S_0$. For the string $s = 0^*1^{d_1}01^{d_2}0 \ldots 1^{d_k}0^*$ that takes the least amount of space, we have $F_0 + 1$ possibilities.

2. Then, we start dividing the free space F_0 along the separations. This gives

This process is not very convenient, as going through all possible combinations is not automatically a polynomial time process. Furthermore, it is not said that each pixel can be given a black or white value. Therefore, we will look at a few other, heuristic processes in which black pixels can be found.
This process is called *probing*.

6.2 Heuristic solving steps

In [13, Section III.2], several heuristic solving steps are given in order to determine the value of some pixels in a single row or column. To determine which pixels can be assigned a certain value, we make use of *run ranges*. In a row or column description, we assign each description integer d_i a *range start* $R_{s,i}$ and *range end* $R_{e,i}$. That is, the the lowest and highest pixel number that can contain black pixels of the run corresponding to the description integer d_k. The idea for colouring pixels by run ranges is given in [7, Section 2].

6.2.1 Setting run ranges

Given a certain row or column description $d_1, d_2, \ldots, d_k$ for a row or column with pixels $1, 2, \ldots, n$, we set the initial range starts as follows:

- The initial range start $R_{s,1}$ of d_1 is 1.

- For description integers d_i with $i > 1$, the initial range starts $R_{s,i}$ are given recursively by:

$$R_{s,i} = R_{s,i-1} + d_{i-1} + 1 \tag{6.2}$$

The idea behind this assignment is that the possibility for the run d_i to start which is the nearest from the beginning, is when all previous runs are put after another as closely as possible. That is, when each subsequent two runs are separated by only one white pixel. We set the initial range end in a similar way. This time, we start counting the possible run ends that are the furthest from the beginning:

- The initial range end $R_{e,k}$ of d_k is n.
- For description integers d_i with $i < k$, the initial range ends $R_{e,i}$ are given recursively by:

$$R_{e,i-1} = R_{e,i} - d_i - 1 \tag{6.3}$$

6.2.2 Filling pixels by run ranges

After having set the initial run ranges, we can start assigning pixels values by heuristics. In [7, Section 2], several steps for filling pixels are given.

If a certain run has overlap within its own range, we can use this to colour the overlapping pixels black. Given the range start $R_{s,i}$ and range end $R_{e,i}$ for a description integer d_i, the overlapping pixels can be coloured as a run of black pixels. This run starts at that pixel that lies exactly $d_i - 1$ pixels before the range end, and ends at the pixels that lies $d_i - 1$ pixels after the range start (one pixel is subtracted because the range starts and range ends themselves can also be part of the final black run):

- First black pixel: $R_{e,i} - d_i + 1$
- Last black pixel: $R_{s,i} + d_i - 1$

This corresponds to the overlap of runs we have in the DFS process. A Matlab script for colouring pixels black in this way, is named `overlapwithrange` and can be found in appendix A.

Some pixels may not be contained in any range at all. That is, for a pixel number p there is no description integer d_i for which $R_{s,i} \leq p \leq R_{e,i}$. This is either the case when:

- $1 \leq p < R_{s,1}$ (the pixel lies before the first range),
- $R_{e,i} < p < R_{s,i+1}$ (the pixel lies between two ranges),
- $R_{e,k} < p \leq n$ (the pixel lies after the last range).

We are sure that such pixels can never be coloured black and therefore we can colour these pixels white. A script for colouring these pixels is named `betweenranges` and can be found in appendix A.

When for a description integer d_i we have $R_{s,i} + d_i - 1 = R_{e,i}$, the run is complete. That is, a subsequent black run consists exactly of the number of black pixels it should contain (and by the previous two steps, the black pixels have already been filled). The single pixels before and after the run cannot be black, since this would result in a larger black run. Therefore, these pixels can be coloured white.

6.2.3 Updating run ranges

Only making use of the initial run ranges is not very clever, as this would only use the information that is given by the row or column description itself. Therefore, we have to update the run ranges according to the new information we have obtained in a certain row or column.
We can make use of the following known aspects of the image I:

1. Known white spaces can ensure that less space is need to fill black spaces, and therefore we can determine pixels to be black
2. Known black spaces can ensure that a row has to contain empty space, which determines pixels to be white.
3. Look for consecutive white and black pixels (not necessarily in that order), and determine which pixels have to be filled.
4. Verifying whether connected black pixels can fit within known white space.
5. Fixing pixels to be either black or white, by calculating the leftmost and rightmost possible solutions.

6.3 Comparison of solvers

It is not said that we can solve a certain Nonogram by only applying heuristic steps. After all, we have shown in section 4.2 that the solution problem for Nonograms is NP-complete. By combining heuristic solving steps and probing, that is, guessing a pixel value and change it when this guess leads to a contradiction, one could be able to solve Nonograms.

To do so, we use existing Nonogram solvers, and put in given examples of Nonograms of which it is known that they have a (unique or not) solution. We compare these solvers by analysing the (heuristic) solving steps are being applied and in which order, and determining when a solver starts to probe when no more heuristic steps are possible. Furthermore, we try several Nonogram examples of which it is known that they have a solution (though possibly not a unique one), and compare the performance of the solvers concerning these examples.

6.3.1 Teal's Nonogram Solver

This solver has been built as an online applet, and can be found on `http://a.teall.info/nonogram`. The input for this solver is given by a single string,

which contains the row and column descriptions one by one.

It turns out that this solver runs through all rows and columns one by one, and applies heuristic steps in order to fill pixels.

Remarkably, the solver leaves certain pixels undetermined that can be filled obviously by logical determination. In particular, these are pixels that need information from the last pixels in a run, mostly at the end of a row or column. This means that useful, straightforward information is missed, and even relatively small and simple Nonograms cannot be solved entirely.

For instance, filling in a 15 by 15 Nonogram description leaves a few (mostly not more that 10) pixels undetermined. This means that this solver cannot handle large puzzles. When trying several 35 by 70 puzzles, the solver in some cases terminates when the image is not even filled half!

6.3.2 Simpsons' Nonogram Solver

On the site `http://www.lancaster.ac.uk/~simpsons/nonogram/` one can find Simpsons' Nonogram Solver. Similar to the Teal solver in section 6.3.1, this is a web-based solver. It can either give the solution of the entire Nonogram at once, or start an applet that shows the solving steps in between. The input for this solver can be given by a text file, containing the row and column descriptions, along with certain additional parameters.

As well as Teal's solver, this solver always runs through the rows and columns one by one. It turns out that this program is not able to solve Nonograms that cannot be solved heuristically at all. The solver also clearly verifies the solution at the end.

6.3.3 Juraj Simlovic's Nonogram Solver

In contrast to the solvers above, this solver is not an online applet, but a standalone program. A link to download this solver for Windows can be found on `http://jsimlo.sk/griddlers/`.

The input for this solver is given by a plain text file with the .griddler extension. Such a file does not only contain the row and column descriptions, but also a pre-specified grid of the Nonogram solution consisting of undetermined and possibly determined pixel values on which the solver can already rely.

Another difference compared to the previous two solvers, is that this solver does not run through all rows and columns one by one and verify whether certain pixels can be filled. Instead, it analyses the row and column descriptions first, and determines which rows and columns are most likely to be filled as much as possible.

When a certain new pixel is being filled, the solver does not switch to the next row or column. Instead, it switches to the corresponding column of the pixel when a row description was used to fill the pixel and vice versa. After all,

Name:	Size:	Teal's:	Simpsons':	Simlovic's:
Paars...	15 × 15			
... gewijs	15 × 15			
Hoogstaand	15 × 15			
Dooie pier	15 × 15			
Nogal beladen	15 × 15			
Het dak gaat eraf	15 × 15			
Topfokker	15 × 15			
Vlammenwerper	15 × 15			
Stemverheffing	35 × 70			
Alle goede dingen	35 × 70			

Table 6.1: The predescribed Nonogram examples

that row or column contains new, possibly useful information that can be used to determine new pixel values.

It turns out that this solver gives the solution rather quickly (in terms of performance steps, not the time in which the Nonogram is solved) in comparison to the other two solvers.

6.3.4 Results for given examples

To compare these solvers more specifically, we apply several given Nonograms to these solvers and look at both the solving process and the outcome. We use examples that have turned up in puzzle booklets and of which it is said that they have a unique solution and are simple (they can be solved by using information of one row or column at a time). We plug in the following puzzles, varying from relatively small to enormous:
Furthermore, we look at a very simplistic, though useful kind of Nonograms, namely the ones that only consist of black and white squares of a certain size (this can be seen of some kind of checkerboard with squares of different sizes). This is useful since we can compare the results of the solvers in terms of the solving process efficiency of using information and handling puzzles without a unique solution very precisely.
We can roughly divide these Nonograms into three groups:

- Nonograms that are both simple and have a unique solution.
- Nonograms that have a unique solution, but are not simple.
- Nonograms that do not have a unique solution.

Obviously, a Nonogram cannot be simple if it does not have a unique solution

Name:	**Size**:	**Unique solution:**	**Simple nonogram:**
1×1 squares 1	5 × 5	Yes	Yes
1×1 squares 2	5 × 5	Yes	Yes
1×1 squares	10 × 10	No	No
2×2 squares	10 × 10	Yes	No
2×2 squares	10 × 10	Yes	No
5×5 squares	10 × 10	No	No
1×1 squares 1	15 × 15	Yes	Yes
1×1 squares 2	15 × 15	Yes	Yes
3×3 squares 1	15 × 15	Yes	No
3×3 squares 2	15 × 15	Yes	No
5×5 squares 1	15 × 15	Yes	Yes
5×5 squares 2	15 × 15	Yes	Yes
1×1 squares	20 × 20	No	No
2×2 squares	20 × 20	No	No
4×4 squares 1	20 × 20	Yes	No
4×4 squares 2	20 × 20	Yes	No
5×5 squares	20 × 20	No	No

Table 6.2: The Nonograms with black and white squares

Chapter 7

Conclusion

After introducing Nonograms in Chapter 3 and some basic notions from graph theory and complexity theory in Chapter 2, we present the main theorem on the complexity of solvability of Nonograms as Theorem 4.2.1. The proof, which consists of reducing the so-called Planar Bounded NCL Problem to this solvability, is discussed in Chapter 4.
Chapter 5 discusses the complexity of the uniqueness problem of solutions, with as main result Theorem 5.2.2. Both problems (solvability and uniqueness) turn out to be NP-complete.
Chapter 6 deals with a basic strategy for solving Nonograms. We implemented some of this in Matlab, and we compared the performance of a few online solvers. We found that none of these was able to completely solve all test problems we used to compare the solvers.

Bibliography

[1] K.J. Batenburg and W.A. Kosters, *A Discrete Tomography Approach to Japanese Puzzles*, preprint, 2005.

[2] J. Kleinberg, E. Tardos, *Algorithm Design*, Pearson Education Inc., 2005.

[3] M. Mesbahi, M. Egerstedt, *Graph Theoretic Methods in multiagent Networks*, Princeton University Press, 2010.

[4] R.A. Hearn, *Games, Puzzles and Computation*, Massachusetts Institute of Technology, 2006

[5] R.A. Hearn, E.D. Demaine, *The Nondeterministic Constraint Logic Model of Computation: Reductions and Applications*, ICALP, *Lecture Notes in Computer Science* volume 2380, 401-413, Springer, 2002.

[6] K.J. Batenburg, S. Henstra, W.A. Kosters, W.J. Palenstijn, *Constructing Simple Nonograms of Varying Difficulty*, Corvinus University of Budapest, Department of Mathematics, *Pure Mathematics and Applications*, 20(1), 1-15, 2009.

[7] C.H. Yu, H.L. Lee, L.H. Chen, *An efficient algorithm for solving nonograms*, Springer Science+Business Media, Springer Verlag, *Applied Intelligence* 35(1): 18-31, 2011

[8] D.S. Johnson, "A Catalog of Complexity Classes", *Algorithms and Complexity*. Ed. J. Van Leeuwen, Amsterdam: Elsevier Science Publishers, 1992. 67-162

[9] J. Barwise, J. Etchemendy, *Language, Proof and Logic*, CSLI Publications, 2003.

[10] S.R. Dunbarr, *Topics in Probability Theory and Stochastic Processes*,

[11] N. Ueda, T. Nagao, *NP-completeness Results for NONOGRAM via Parsimonious Reductions*

[12] W.H. Hesselink, *Dictaat Talen en Automaten*, Rijksuniversiteit Groningen, 2012.

[13] S. Salcedo-Sanz, E.G. Ort'iz-Garc'ia et al., *Solving Japanese Puzzles with Heuristics*, IEEE Symposium on Computational Intelligence and Games, 2007, 224-231, CIG, 2007.

[14] K.J. Batenburg, W.A. Kosters, *Solving Nonograms by combining relaxations*, Pattern Recognition, Volume 42, Issue 8, 1672?1683, Elsevier, 2009.

[15] J. N. van Rijn, *Playing Games: The complexity of Klondike, Mahjong, Nonograms and Animal Chess*, Master's thesis, Universiteit Leiden, 2012.

Appendix A

MATLAB codes for algorithms

betweenranges

```
function [newimage] = betweenranges(image,rowstarts,rowends,colstarts,colends)

% Betweenranges
%
% Input variables:
%
%   rowstarts:  matrix with range starts for each row description
%
%   rowends:    matrix with range ends for each row description
%
%   colstarts:  matrix with range starts for each column description
%
%   colstarts:  matrix with range ends for each column description
%
% Output variables:
%
%   image:  (partially) filled image of the nonogram in the form of a
%           matrix
%
% This function is supposed to fill the nonogram image as much as possible,
% due to the properties of underfull ranges.

% More convenient names for variables:

rowrangestarts      = rowstarts;
rowrangeends        = rowends;
columnrangestarts   = colstarts;
columnrangeends     = colends;

% Preallocation of variables:
```

```
newimage = image

numberofrows    = size(rowrangestarts,1)
numberofcolumns = size(columnrangestarts,2)

height = numberofrows
width = numberofcolumns

% For each row and column description entry, we have a range. The space in
% between these ranges will never be coloured black, so we can colour them
% white. The run for which this holds, is:
%
% "End of range" + 1
%
% till
%
% "Start of next range" - 1
%
% If the end of the white coloured run is before the start of the run, the
% program will just do nothing.
%

for rownumber = 1 : numberofrows

currentrow      = rowrangestarts(rownumber,:)
rangeentries    = nonzeros(currentrow)
numberofentries = nnz(rangeentries)

for entrynumber = 1 : (numberofentries-1)

currentrangeend = rowrangeends(rownumber,entrynumber)
nextrangestart = rowrangestarts(rownumber,entrynumber+1)

runstart = currentrangeend + 1
runend = nextrangestart - 1

% Now we can colour several pixels white

image(rownumber,runstart:runend) = -1

end

% In the end, we must colour the pixels white that lie before the first
% range and after the last range:

firstrangestart = rowrangestarts(rownumber,1)
lastrangeend = rowrangeends(rownumber,numberofentries)

image(rownumber,1:firstrangestart-1) = -1
```

```
image(rownumber,lastrangeend+1:width) = -1
end

% Now the same process for each column

for columnnumber = 1 : numberofcolumns

currentcolumn   = columnrangestarts(:,columnnumber)
rangeentries    = nonzeros(currentcolumn)
numberofentries = nnz(rangeentries)

for entrynumber = 1 : (numberofentries-1)

currentrangeend = columnrangeends(entrynumber,columnnumber)
nextrangestart = columnrangestarts(entrynumber+1,columnnumber)

runstart = currentrangeend + 1
runend = nextrangestart - 1

% Now we can colour several pixels white

image(runstart:runend,columnnumber) = -1

end

% In the end, we must colour the pixels white that lie before the first
% range and after the last range:

firstrangestart = columnrangestarts(1,columnnumber)
lastrangeend = columnrangeends(numberofentries,columnnumber)

image(1:firstrangestart-1,columnnumber) = -1
image(lastrangeend+1:width,columnnumber) = -1
end

% By now, one can produce a HeatMap of the image. Black stands for filled
% pixels, white stands for empty or undetermined pixels

% colormap = [1 1 1;1 1 1;0 0 0]
% HeatMap(flipud(image),'Colormap',colormap)

end
```

nonogramverify

```
function [adhere] = NonogramVerify(rowdescriptions,columndescriptions,image)

[height,width] = size(image);
```

```
if size(rowdescriptions,1) == height && size(columndescriptions,2) == width
else
error('The input variables are not of appropriate size!')
end

if round(rowdescriptions) ~= rowdescriptions
error('The row description is not contained of integers')
else
if round(columndescriptions) ~= columndescriptions
error('The column description is not contained of integers')
end
end

if image == logical(image)
else
error('The image is not contained of zeros and ones exclusively')
end

% Then, we can start verifying for each row column whether it adheres to
% its corresponding description.

adhere = 0;

% In case when the process is aborted early (by the 'return' command), it
% mostly means that the image is not correct. If it isn't the output value
% has already been set. If the image is correct, the value of 'adhere' will
% be changed.

numberofrows        = height;
numberofcolumns     = width;
rowverifications    = zeros(height,1);
columnverifications = zeros(1,width);


for rownumber = 1 : numberofrows
rowpixels   = image(rownumber,:);
descrow     = rowdescriptions(rownumber,:);

if verifywithcomments(rowpixels,descrow) == true
rowverifications(rownumber) = 1;
end

end % For-loop end

for columnnumber = 1 : numberofcolumns
columnpixels    = image(:,columnnumber);
desccolumn      = columndescriptions(:,columnnumber);

if verifywithcomments(columnpixels,desccolumn) == true
```

```
columnverifications(columnnumber) = 1;
end

end % For-loop end

rowverifications

columnverifications

if sum(rowverifications) == numel(rowverifications)
if sum(columnverifications) == numel(columnverifications)
disp('The image adheres to the description')
else
disp('The image does not adhere to the description')
end
end % If end

end % Function end
```

overlap

```
function [image] = overlap(rowdescription,columndescription)

% Overlap
%
% Input variables:
%
%   rowdescription:     matrix with description for each row of the
%                       nonogram
%
%   columndescription:  matrix with description for each row of the
%                       nonogram
%
% Output variables:
%
%   image:  (partially) filled image of the nonogram in the form of a
%           matrix
%
% This function is supposed to fill the nonogram image as much as possible,
% due to the properties of overlapping possible black pixels.

% Preallocation of variables:

height = size(rowdescription,1);
width =  size(columndescription,2);

image = zeros(height,width);
```

```
% For each row, it has to be determined whether several pixels in this row
% can be filled, due to overlap. Therefore, we need the minimal space
% needed to fill the row, which equals:
%
%       minimal space = "sum of entries" + "number of nonzero entries" - 1
%
% The solution for one particular row can be given entirely when:
%
%       minimal space = width
%
% At least a few pixels can be coloured when
%
%       width > minimal space > width - "greatest entry".
%
% And no black pixels at all can be returned when
%
%       minimal space <= width - "greatest entry".
%

for rownumber = 1:height

currentrow      = rowdescription(rownumber,:)
rowentries      = nonzeros(currentrow)
sumofentries    = sum(rowentries)
numberofentries = nnz(rowentries)

minimalspace = sumofentries + numberofentries - 1
maxentry = max(rowentries)

if minimalspace == width
startpixel = 1
for entrynumber = 1:numberofentries

numberofpixels = rowentries(entrynumber)
nextpixel = startpixel + numberofpixels
endpixel = nextpixel - 1

image(rownumber,startpixel:endpixel)= 1
if entrynumber ~= numberofentries
image(rownumber,nextpixel)      = -1
end

startpixel = nextpixel + 1
end
end

if minimalspace < width
if minimalspace > (width - maxentry)
freespace = width - minimalspace
startpixel = 1
```

```
for entrynumber = 1:numberofentries

numberofpixels = rowentries(entrynumber) - freespace
startpixel = startpixel + freespace
nextpixel = startpixel + numberofpixels
endpixel = nextpixel - 1

image(rownumber,startpixel:endpixel)= 1

startpixel = nextpixel + 1
end

end
end


end

% Now the same process for each column

for columnnumber = 1:width

currentcolumn   = columndescription(:,columnnumber)
columnentries   = nonzeros(currentcolumn)
sumofentries    = sum(columnentries)
numberofentries = nnz(columnentries)

minimalspace = sumofentries + numberofentries - 1
maxentry = max(columnentries)

if minimalspace == height
startpixel = 1
for entrynumber = 1:numberofentries

numberofpixels = columnentries(entrynumber)
nextpixel = startpixel + numberofpixels
endpixel = nextpixel - 1

image(startpixel:endpixel,columnnumber)= 1
if entrynumber ~= numberofentries
image(nextpixel,columnnumber)      = -1
end

startpixel = nextpixel + 1
end
end

if minimalspace < height
if minimalspace > (height - maxentry)
```

```
freespace = height - minimalspace
startpixel = 1

for entrynumber = 1:numberofentries

numberofpixels = columnentries(entrynumber) - freespace
startpixel = startpixel + freespace
nextpixel = startpixel + numberofpixels
endpixel = nextpixel - 1

image(startpixel:endpixel,columnnumber)= 1

startpixel = nextpixel + 1
end

end
end


end

% By now, one can produce a HeatMap of the image. Black stands for filled
% pixels, white stands for empty or undetermined pixels

colormap = [1 1 1;1 1 1;0 0 0]
HeatMap(flipud(image),'Colormap',colormap)

end
```

overlapwithrange

```
function [image] = overlapwithrange(rowdesc, coldesc, rowstarts,rowends,colstarts,colends)

% Overlapwithranges
%
% Input variables:
%
%   rowdesc:    matrix with description for each row of the nonogram
%
%   columndesc: matrix with description for each row of the nonogram
%
%   rowstarts:  matrix with range starts for each row description
%
%   rowends:    matrix with range ends for each row description
%
%   colstarts:  matrix with range starts for each column description
%
%   colstarts:  matrix with range ends for each column description
%
```

```
% Output variables:
%
%   image:  (partially) filled image of the nonogram in the form of a
%           matrix
%
% This function is supposed to fill the nonogram image as much as possible,
% due to the properties of overlapping possible black pixels according to
% ranges.

% More convenient names for variables:

rowdescription      = rowdesc;
columndescription   = coldesc;

rowrangestarts      = rowstarts;
rowrangeends        = rowends;
columnrangestarts   = colstarts;
columnrangeends     = colends;

% Preallocation of variables:

height = size(rowdescription,1);
width =  size(columndescription,2);

image = zeros(height,width);

numberofrows    = height
numberofcolumns = width

% For each row and column description entry, we have a range. According to
% these ranges, we have to determine which pixels we can colour black.
%
% "End of range" - "Size of entry" + 1
%
% till
%
% "Start of range" + "Size of entry" - 1
%
% If the end of the black coloured run is before the start of the run, the
% program will just do nothing.
%

for rownumber = 1 : numberofrows

currentrow      = rowdescription(rownumber,:)
rowentries      = nonzeros(currentrow)
numberofentries = nnz(rowentries)

for entrynumber = 1 : numberofentries
```

```
runsize = rowentries(entrynumber)

currentrangestart = rowrangestarts(rownumber,entrynumber)
currentrangeend = rowrangeends(rownumber,entrynumber)

runstart = currentrangeend - runsize + 1
runend = currentrangestart + runsize - 1

% Now we can colour several pixels black

image(rownumber,runstart:runend) = 1

end

end

% Now the same process for each column

for columnnumber = 1 : numberofcolumns

currentcolumn       = columndescription(:,columnnumber)
columnentries       = nonzeros(currentcolumn)
numberofentries     = nnz(columnentries)

for entrynumber = 1 : numberofentries

runsize = columnentries(entrynumber)

currentrangestart = columnrangestarts(entrynumber,columnnumber)
currentrangeend = columnrangeends(entrynumber,columnnumber)

runstart = currentrangeend - runsize + 1
runend = currentrangestart + runsize - 1

% Now we can colour several pixels black

image(runstart:runend,columnnumber) = 1

end

end

% By now, one can produce a HeatMap of the image. Black stands for filled
% pixels, white stands for empty or undetermined pixels

% colormap = [1 1 1;1 1 1;0 0 0]
% HeatMap(flipud(image),'Colormap',colormap)

end
```