Generation of sloped nonograms

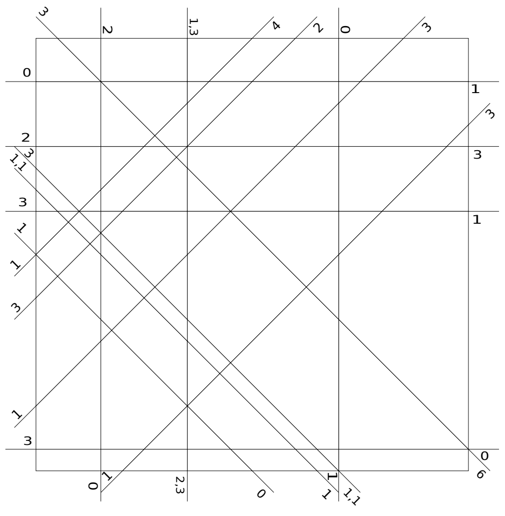
Raphaël Parment

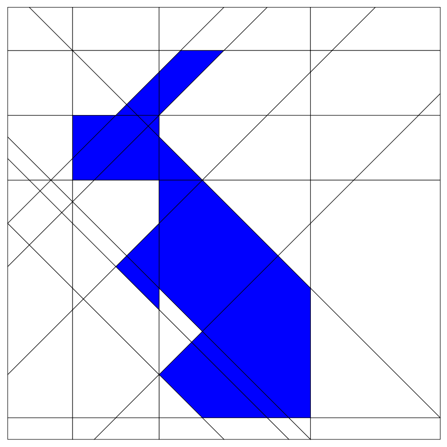
3493865

Master's Thesis

Department of Information and Computing Sciences
Universiteit Utrecht

Abstract





This thesis presents the automated generation of *sloped nonograms* based on input drawings. *Sloped nonograms* are similar to regular nonograms except for the fact that they are constructed using edges and faces as opposed to rows, columns and cells. The generated sloped nonograms satisfy three main constraints: they do not contain small faces, their topology remains identical to that of the input and they belong to the *simple* class of nonograms. *Simple* nonograms are those which can be solved by incrementally assigning a colour to faces without the need of backtracking or assumptions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nonograms, also known as Japanese puzzles, are logic puzzles based on a rectangular grid of arbitrary size. The goal is to determine the colour of each cell, either black or white, using the number sequence, called *description*, given for each row and column. Figure 1.1 shows a nonogram with its solution. Each number s_i part of a description $(s_1, ..., s_k)$ informs the user that a black run of s_i consecutive black cells is expected in the current row or column. There should be at least one white cell between consecutive black runs. This thesis introduces a new variation of nonograms called *sloped nonograms.* Those puzzles have edges and faces instead of rows, columns and cells. Figure 1.2 shows a generated sloped nonogram. We want to be able to generate such sloped nonograms from input drawings. Moreover, we do not want sloped nonograms to have small faces, since those are not pleasant to colour. In addition, we want the output sloped nonograms to be topologically identical to their input. We also want to make sure that the generated sloped nonograms are uniquely solvable and *simple*, meaning they can be solved iteratively and never require multiple pieces of information simultaneously.

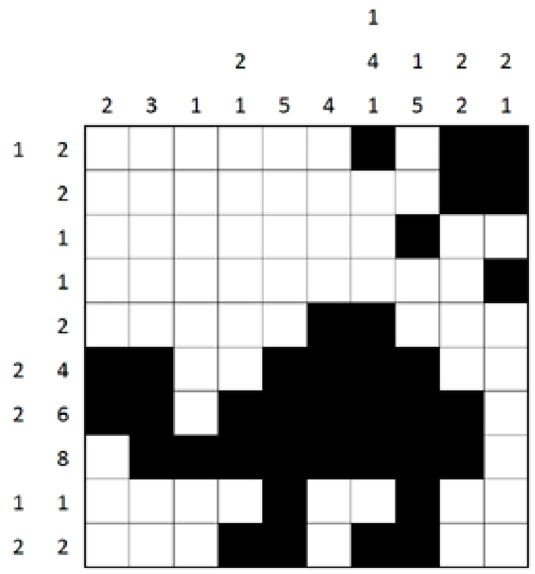


Figure 1.1: Regular nonogram with its solution [15].

A review on some of the key information regarding regular nonograms, as well as some of the geometric and algorithmic tools required for this thesis will be given in

Section 2. Section 3 will formally define the problem, describing and explaining any rule or assumption used throughout the thesis. Section 4 will describe the method used to generate sloped nonograms satisfying the aforementioned constraints. Section 5 will analyse the quality of output puzzles, the speed of the generation process, and more results. Section 6 will conclude the findings. Finally, section 7 will look at future work in the field.

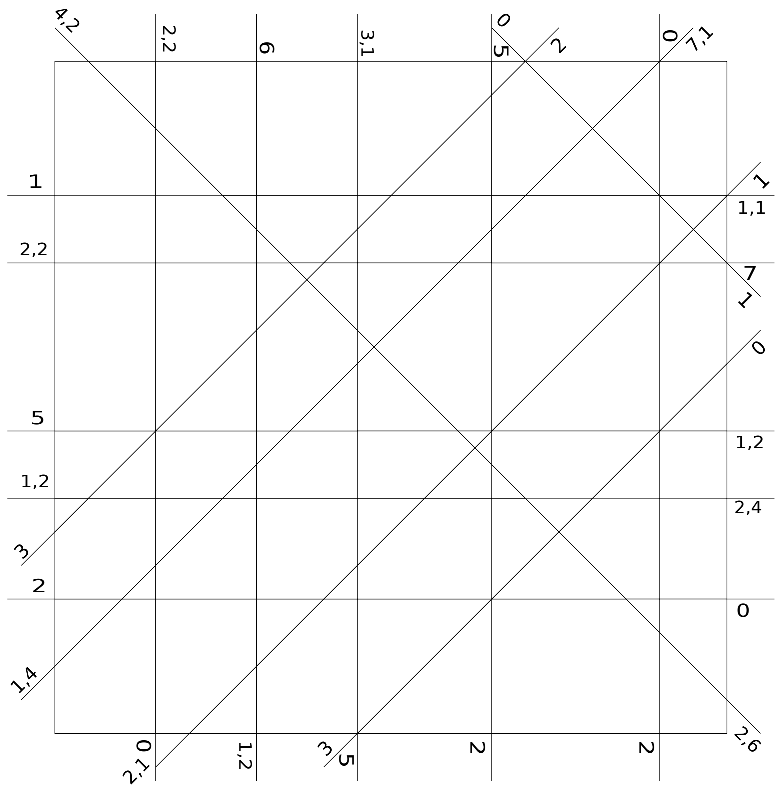


Figure 1.2: Generated sloped nonogram.

Chapter 2

Related work

2.1 Logic puzzles in computer science

Logic puzzles are popular in computer science research because they usually are straightforward to model while still allowing for vast solving performance investigations. Several logic puzzles such as: 'Pic-a-Pix', 'Cross-a-Pix', 'Fill-a-Pix', 'Link-a-Pix', 'Sym-a-Pix', 'Maze-a-Pix' and 'Dot-a-Pix' [7] are generated using input drawings. Multiple interesting extensions of the 'Dot-a-Pix' puzzle have been investigated. The classical version amounts to generating a set of dots, each associated with a number. The puzzle is solved by starting at the dot with associated number 0 and connecting it with the dot with associated number 1, and so on. The output usually is a drawing. Extensions investigated in [12] and [11] use direction indicators, and distance and colour respectively as opposed to numbers to indicate the order in which points must be connected.

2.2 Literature review

Solvable nonograms are split into two categories: *simple* and *non-simple* nonograms. *Simple* nonograms can be solved iteratively by finding the colour of as many cells as possible in single rows or columns. *Non-simple* nonograms however, require information on multiple rows and columns to determine the colour of single cells [1]. They sometimes also require assumption making. Nonograms may also be either unique and have a single solution, or non-unique and have multiple solutions. We can therefore classify four types of nonograms:

- *Simple* nonograms; they have one unique solution and are solvable iteratively row by row and column by column, see Figure 2.1.
- *Non-simple* unique nonograms; they have one unique solution but require information about multiple rows and columns simultaneously and sometimes assumptions for solving, see Figure 2.2.
- *Non-simple* non-unique; they have multiple solutions and need information about multiple rows and columns simultaneously and sometimes assumptions for solving, see Figure 2.3.
- Non-solvable; they have no solution, see Figure 2.4.

In nonograms shown in Figures 2.1, 2.2, 2.3 and 2.4 a grey cell represents unknown colour.

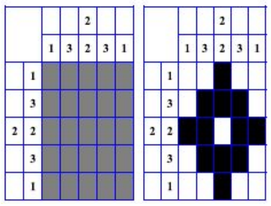


Figure 2.1: Left: empty *simple* nonogram. Right: the solution [17].

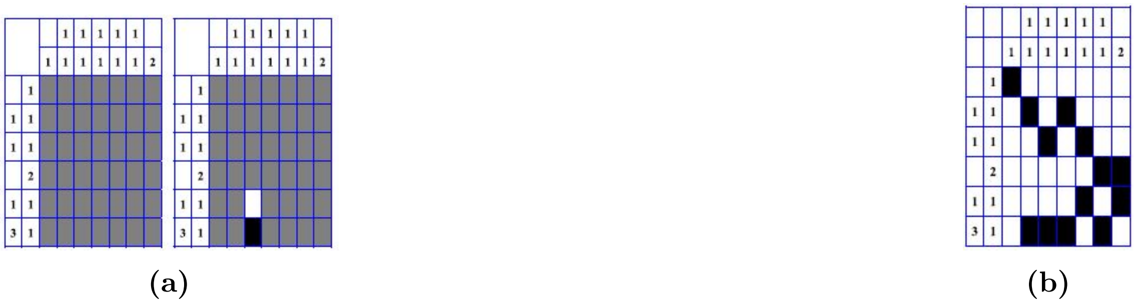


Figure 2.2: *Non-simple* unique nonogram. Figure 2.2a shows where iterative solving gets stuck. Figure 2.2b shows the final solution [17].

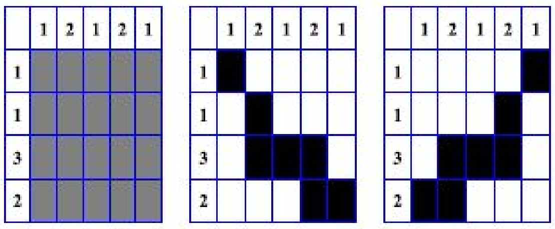


Figure 2.3: *Non-simple* non-unique nonogram. Last two puzzles show the possible solutions [17].

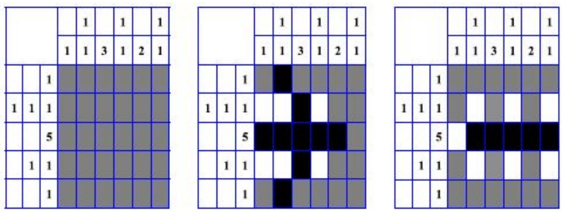


Figure 2.4: Non-solvable nonogram. In the middle puzzle, if the first five cells are coloured in row 3, there is no solution in rows 2 and 4 [17]. If the last five cells are coloured in row 3, there is no solution in row 4 [17].

Most puzzles handed out to the public are *simple*, so most solvers tackle *simple* nonograms. We shall look at some key ideas put forward in [4] for solving *simple* nonograms, as well as more complex ideas presented in [17] to solve any nonogram.

For *simple* nonograms, solvers operate row by row and column by column. Showing that solving a single row or column can be done in polynomial time is a sufficient condition to develop a fast algorithm. For a row or column description $d = (d_1, ..., d_k)$, where k is the number of values in the description, we have $d_j = \sigma_j\{a_j, b_j\}$, where $\sigma_j \in \Sigma$ (an alphabet $\{0, 1\}$ in which 0 is white and 1 is black), and $a_j, b_j \in \{0, 1, 2...\}$ with $a_j \leq b_j$ $(j = 1, 2, ..., k)$ [4]. The a_j and b_j can be considered as the leftmost starting index and rightmost ending index, in the row or column, respectively for a black run j. For instance, for a row of length 5 with description $d = d_1 = (3)$, the leftmost starting index and rightmost ending index are 0 and 4 respectively. In the two extreme configurations we have $a_1 = 0$ and $b_1 = 2$, and $a_1 = 2$ and $b_1 = 4$, thus we get $\{a_1, b_1\} = \{0, 4\}$. A description $(0\{0, \infty\}$, $1\{a_1, a_1\}$, $0\{1, \infty\}$, $1\{a_2, a_2\}$, ..., $1\{a_k, a_k\}$, $0\{0, \infty\})$, where ∞ is a suitably large number, is equivalent to the nonogram-type description $d = (a_1, a_2, ..., a_k)$ [2]. This description can be written as $0^*1^{a_1}0^+1^{a_2}0^+...1^{a_k}0^*$. A finite string s over Σ *adheres* to such a description $d = (a_1, a_2, ..., a_k)$ if s satisfies the regular expression $0^*1^{a_1}0^+1^{a_2}0^+...1^{a_k}0^*$ [1]. The following recursion can then be defined:

$$Fix(i, j) = \bigvee_{p=\max(i-b_j, A_{j-1}, L_i^{\sigma_j}(s))}^{\min(i-a_j), B_{j-1}} Fix(p, j-1), \tag{2.1}$$

where $A_j = \sum_{p=1}^{j} a_p$, $B_j = \sum_{p=1}^{j} b_p$, and $A_0 = B_0 = 0$. For a string $s = s_1, ..., s_\ell$ of length ℓ and description $d = (d_1, ..., d_k)$, the value of $Fix(\ell, k)$ determines whether s is fixable with respect to d. Given a string s over an alphabet Γ ($\Sigma \cup \{x\}$, where x represents the unknown cell colour) which is fixable to a description d, if in all fixes, a certain cell has the same value from Γ, then it must have that value in the solved puzzle. The process can be repeated until the colour of every cell is known. Refer to [4] for more information and for proof of the recursion.

A procedure capable of solving *simple* as well as *non-simple* nonograms is developed in [17]. The procedure described to solve *simple* nonograms closely resembles the one presented in [4], except that it is shaped as a set of logical rules instead of a recursion. The authors also describe a depth-first search procedure which, when combined with the aforementioned logical rules, can prune the search space and yield all possible solutions for any nonogram. The authors explain that depth-first search is a valid brute force approach for solving nonograms with small dimensions but it becomes infeasible time-wise as the dimensions increase. Thus they develop logical rules to determine the colour of as many cells as possible in between depth-first search runs, in order to extensively prune the search space. The logical rules apply to single rows and columns and can be encompassed into the recursion shown in equation 2.1. We shall describe the first logical rule, but refer to [17] for the full set. The authors use the *range* notation (r_{js}, r_{je}) for a black run j. The range of a black run j is defined as the leftmost starting index r_{js}, and the rightmost ending index r_{je}. The first logical rule is: For a black run j of length LB_j, cell c_i will be coloured if $r_{js} + u \leq i \leq r_{je} - u$, where $u = (r_{je} - r_{js} + 1) - LB_j$ [17]. It basically states that if in all possible configurations of black run j, a cell is coloured, then it must be coloured in the solved nonogram.

Rules are divided into three categories: determining the colour of as many cells as possible, refining ranges of description pieces, and combining both aforementioned goals. The mechanism for the overall nonogram solving procedure is as follows: first, use the logical rules on each row and column to find the colour of as many cells as possible and refine as many ranges as possible. We are either done or stuck once this step terminates. If stuck, generate all possible solutions for the first unsolved row. Verify whether the solutions generated are in accordance with column descriptions. If so, visit the first solution and apply the logical rules again until no more information can be discovered. Repeat this procedure until the colour of all cells is determined. Whenever a generated solution does not satisfy the column descriptions, it is discarded. This routine quickly discovers all solutions for a nonogram, thanks to the pruning obtained by using the logical rules. Refer to [17] for benchmarking of the algorithm.

2.3 Doubly connected edge list

Our solution for generating sloped nonograms deals with holding information about a geometric subdivision. The optimal data structure for such a task is the doubly connected edge list (DCEL). In this section we shall discuss the advantages and strengths of this structure. The DCEL consists of three main entities: the 'Vertex', the 'Half-Edge' and the 'Face'. Each of those entities has several useful attributes used to explore a subdivision [6].

A vertex v stores its coordinates, as well as a pointer to each of its incident half-edge.

A half-edge e stores a pointer to its origin, a vertex instance. It also stores a pointer to its twin half-edge. For a half-edge e, with origin P_0, the twin of e, $twin(e)$, has origin P_1, where P_1 is the destination of e and P_0 the destination of $twin(e)$. A half-edge also stores a pointer to its incident face, the adjacent face which has e as part of its boundary. The boundary is always defined counter-clockwise. Finally a half-edge stores a pointer to its next and previous half-edge, both on the boundary of the incident face of e. From a half-edge, one can easily travel along the boundary of the incident face by recursively using the next or previous pointers.

A face f stores a pointer to its outer component, a half-edge on the boundary of the f. It also stores a list of inner components, which amounts to a pointer to one half-edge on the boundary of each hole present in the face [6].

2.4 Linear programming

The next necessary ingredient to be able to generate sloped nonograms from drawings involves optimising the size of the faces in the subdivision. To optimise these, we chose to use linear and quadratic programming. In this section we will review the basic layout of linear programmes. We will also briefly discuss the simplex algorithm for solving such problems.

A basic linear programme consists of two main items: an objective function, to maximise or minimise, and a set of constraints to satisfy. Consider the following linear programme defined as:

$$\begin{aligned}
\text{Minimise } & c_1x_1 + c_2x_2 + ... + c_nx_n \\
\text{Subject to: } & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \geq b_1 \\
& a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \geq b_2 \\
& \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \geq b_m \\
& x_1, x_2, ..., x_n \geq 0
\end{aligned} \tag{2.2}$$

or equivalently:

$$\begin{aligned}
\text{Minimise } & \mathbf{cx} \\
\text{subject to } & \mathbf{Ax} \geq \mathbf{b} \\
& \mathbf{x} \geq 0
\end{aligned} \tag{2.3}$$

where:

$$\mathbf{c} = \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \tag{2.4}$$

The set $x_1, ..., x_n$ represents the *decision variables.* The first line in equation 2.2 is the objective function, which depends on the decision variables and on the *cost coefficients* $c_1, ..., c_n$. The following three lines represent the constraints, with a_{ij} and b_i given for $i = 1, ..., m$ and $j = 1, ..., n$. Finally the last line represents the nonnegativity constraints. To solve the problem, one needs to find the optimal value for each x_i, $i = 1, ..., n$ such that the objective function is minimised, while all constraints are still satisfied [5]. To solve such problems, we use the simplex algorithm. The next section will briefly explain how this procedure works, refer to [5] for more details.

Given a linear programme:

$$\begin{aligned}
\text{Min } & \mathbf{cx} \\
\text{subject to } & \mathbf{Ax} \geq \mathbf{b} \\
& \mathbf{x} \geq 0,
\end{aligned} \tag{2.5}$$

where bold symbols represent vectors or matrices.

- First, add one *slack variable* x_{s_i} per constraint to transform it into an equality. The matrix $\mathbf{A} = [\mathbf{a}_1, ..., \mathbf{a}_n]$, becomes $[\mathbf{a}_1, ..., \mathbf{a}_n, \mathbf{a}_{s_1}, ..., \mathbf{a}_{s_m}]$, where $\mathbf{a}_{s_i}$ represents the coefficient vector of slack variable x_{s_i} in $\mathbf{A}$.

- Find a basic feasible solution. It can be extracted from $\mathbf{A}$. The basic feasible solution basis $\mathbf{B}$ must be of correct dimensions, namely performing $\mathbf{B}^{-1}\mathbf{b}$ must be possible. Call $x_i, ..., x_k$ basic variables $\mathbf{x}_B$, if they define the basis of $\mathbf{B}$. Call all other variables non-basic $\mathbf{x}_N$.

- Once $\mathbf{B}$ is found, solve $\mathbf{B}\mathbf{x}_B = \mathbf{b}$ for $\mathbf{x}_B$, obtaining $\mathbf{x}_B = \bar{b}$. Let $\mathbf{x}_B = \bar{b}$, $\mathbf{x}_N = \mathbf{0}$ and z = $\mathbf{c}_B\mathbf{x}_B$, where $\mathbf{c}_B$ represents the coefficients of the basic variables.

- Solve the system $\mathbf{wB} = \mathbf{c}_B$ for $\mathbf{w}$, where $\mathbf{w} = (w_1, ..., w_{dim(\mathbf{c_B})})$. Calculate $z_j - c_j = \mathbf{w}\mathbf{a}_j$ - c_j for all non-basic variables. Let $z_k - c_k = \max_{j \in R} z_j - c_j$, where R represents the indices associated with the non-basic variables. If $z_k - c_k \leq 0$, stop with the current solution as optimal solution. Else, solve $\mathbf{B}\mathbf{y}_k = \mathbf{a}_k$. If $\mathbf{y}_k \leq \mathbf{0}$ then stop with the conclusion that the optimal solution is unbounded. Else, x_k enters the basis and x_{B_r} leaves the basis, where r is determined by the following minimum ration test:

$$\frac{\bar{b_r}}{y_{rk}} = \min_{1 \leq i \leq m} \left\{ \frac{\bar{b_i}}{y_{ik}} : y_{ik} > 0 \right\} \tag{2.6}$$

  Update the basis $\mathbf{B}$, where $\mathbf{a}_k$ replaces $\mathbf{a}_{B_r}$, the index set R and repeat the procedure.

Chapter 3

Formal definition

This section formally introduces the problem, going into great details regarding choices and rules further applied in the thesis. As previously mentioned, the aim of the thesis is to build a method for generating good sloped nonograms from input drawings. In other words, output solved sloped nonograms must resemble their input.

Regular nonograms are played on a rectangular grid, although extensions may use non rectangular grids, with a description for each row and column. The description is a set of k numbers $s_1, ..., s_k$, representing how the cells in the row or column must be coloured. For a row with description $d = (s_1, ..., s_k)$, the row must contain k black runs [17] , with length $s_1, ..., s_k$ respectively, with at least one white cell in between consecutive blocks. In other words, the colouring must satisfy the following regular expression: $0^*1^{s_1}0^+...1^{s_k}0^*$ [1], where 0 represents a white cell and 1 a black cell.

The difference between regular nonograms and sloped nonograms is that instead of having rows, columns and cells, we will use edges and faces inside a square bounding box. From a visual point of view, every regular unsolved nonogram looks similar, simply an empty grid. In addition, all cells have the same size which brings a sense of balance to the puzzle. For sloped nonograms however, that is not the case. The overall subdivision is less balanced because faces may have arbitrarily many sides. Another visual disadvantage of sloped nonograms is that the size of faces is not fixed. Depending on the geometry of the input, one may obtain a wide variety of face sizes. This aspect may become problematic for users when very small faces are present in the puzzle. We realise that colouring small faces can be cumbersome and may make solving puzzles less enjoyable. Moreover, sets of adjacent small faces cramp the subdivision making it less visually appealing. It may not always be the case that small faces can be dealt with. Large faces are not an issue from a visual point of view and they are, to some extent, advantageous for solving purposes, because they tell more about the final drawing than smaller faces. Overall, we wish to create sloped nonograms which do not contain any small faces, yet, when solved, they must resemble their input.

We can partially solve both aforementioned issues by using well defined constraints on edges in the subdivision. To tackle the number of sides in faces, we can constrain the slope of each edge in the subdivision. This is derived from the tangram idea. Tangram drawings only encompass raw contours while not taking any details into account. Different classes of tangram exist, however most are constructed using lines constrained to four slopes: horizontal lines (slope 0°), vertical lines (slope 90°), positively diagonal lines (slope 45°) and negatively diagonal lines (slope -45°). We will use these slope constraints to decrease the maximum number of sides for faces in our subdivision to 8.

For the problem regarding the size of faces, we will use linear and quadratic programming to shift entire lines in the subdivision, such that no more small faces are present in the puzzle. This method may not work for all inputs. However, since we want output solved sloped nonograms to still resemble their input, we must minimise total deformation. To obtain this result, we need to define a distance threshold w_t, such that if a face is smaller than w_t, it is considered small. We also need a proper distance measure to keep track of the size of a face.

The distance measure used to compute the size of a face is the width. The width is the smallest distance out of all distances defined by opposite sides of a face. Using the width as distance measure, resulting puzzles will probably not be as deformed as with other measures such as area, since the width takes the geometry of faces into account.

Let us assert that a face is small if its width is less than 2 millimetres ($w_t = 2$). We chose such a w_t because faces with that width are large enough to colour. It is not certain yet whether 2 millimetres is the best value for w_t. Therefore, we shall perform optimisation three times using $w_t = 2$, $w_t = 3$ and $w_t = 4$.

Another constraint on output sloped nonograms is that of topology. Output sloped nonograms must be topologically identical to their input. This constraint is defined in order to keep the optimised puzzle similar to the input, and simplify the modelling process. This topology constraint amounts to certifying that no face is created or removed, from the initial subdivision, during optimisation. To implement this constraint, we define the following rules to be satisfied during optimisation: triple and quadruple intersections must remain, edges must keep the same neighbours (defined by the edges left and right of the intersection point of the current edge with the bounding box), and edges may not change side of the bounding box with which they originally intersected. In addition we force the bounding box to stay fixed, such that the optimised output sloped nonogram lies inside the same bounding box as its input drawing. The reason is that we do not want to find scaled-up optimised sloped nonograms which satisfy the face size and topology constraints, because that would cause problems for rendering those sloped nonograms.

Lastly we must make sure that the sloped nonograms generated are *simple*, namely that they are uniquely solvable in an iterative way. We chose to apply this rule because from a user point of view, the solution must be unique in order to make sure that the original drawing re-appears when the sloped nonogram is solved. In addition, documentation on solving *simple* nonograms is widely available as opposed to the case of *non-simple* nonograms. Lastly, the main goal is to generate sloped nonograms, making them very difficult is not a priority yet.

Now that we have put forward the primary goals, let us discuss secondary interesting investigations undertaken in this thesis. As previously stated, we are using linear and quadratic programming to optimise subdivisions such that each face has a width larger or equal to w_t (2, 3, and 4 millimetres). In principle, all lines can move, however we want them to shift as little as possible, while still satisfying the constraints on face size and topology, such that the puzzle, when solved, still resembles the input. For minimising line movement from original position, defined by the input, we will use three different objective functions. We will use: min sum of squares ($\min \sum^2$), min sum ($\min \sum$) and min max ($\min \max$). We chose these three functions because they are often used in the field and also because they often exhibit different results.

When using $\min \sum^2$, the optimisation direction, namely positive or negative, does not matter; however this function is highly sensitive to outliers [8]. We believe that an optimised sloped nonogram will not show very large edge shifts as these greatly penalise the objective score.

Min$\sum$ derives from minimising the sum of the absolute values of the decision variables. This function is used to avoid getting the large values obtained when using $\min \sum^2$ [8]. However a programme using $\min \sum$ is not linear because of the absolute value involved. Nevertheless, it may be transformed into a set of linear programmes.

Min max amounts to minimising the maximum *displacement* such that an optimal solution is found in which each decision variable can take a value of magnitude at most *displacement*. This objective may involve many small shifts, with magnitude less or equal than *displacement*, as those do not penalise the overall score. This objective function is often used in problems involving preferences. Using other objective functions such as $\min \sum$, the overall sum of preferences is minimised without looking at personal preferences [8].

An interesting investigation is to compare all three objective functions according to both visual results and speed. Visual results are important to the users as they do not want the final product to be completely deformed from their initial drawing. The speed criterion is crucial since generating a sloped nonogram should be fast.

We also want to compute the maximum possible minimal width (*max minwidth*) for each sloped nonogram such that all faces have a width greater or equal to *max minwidth*. From those results, a conclusion may be drawn between *max minwidth* and the number of lines in the sloped nonogram. This can allow us to understand the features needed for a sloped nonogram to have large faces. This can be used by users to design inputs which will lead to sloped nonograms with large faces.

We also want to find the difficulty of each sloped nonogram. This is useful to users who do not want to start with a complicated sloped nonogram, or conversely want to be challenged. In addition, investigating how the difficulty increases based on the number of lines is also interesting. Users can be informed of these results, and use them to design inputs which will lead to complex sloped nonograms.

Chapter 4

Method

4.1 Brief recipe

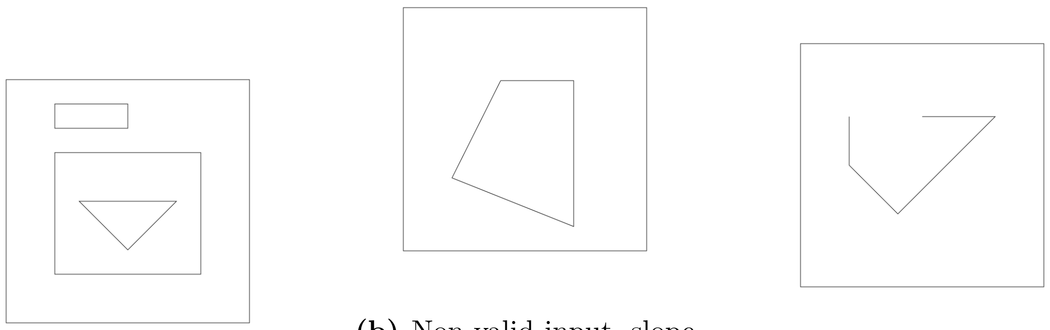
In this section, an overall recipe for generating sloped nonograms from input drawings is given, allowing for further in-depth explanations. The nonograms are generated from specific inputs drawn inside a square bounding box of fixed size. An input is transformed into a set of coordinates representing the vertices of the drawing. Edges are created connecting pairs of vertices, then extended to the bounding box. From the set of extended edges and the bounding box edges, we create a DCEL to hold relevant information regarding faces, edges and vertices of the subdivision.

We use linear and quadratic programming to try to increase the width of all small faces up to w_t while other faces still remain large enough. A decision variable is associated with each extended edge, monitoring the shift of the extended edge. For each face in the subdivision, a set of constraints is generated, forcing the extended edges to be far enough from each other so that the width of the face is larger or equal to w_t. To keep the subdivision topologically identical to the input, we generate constraints forcing triple and quadruple intersections to stay intact, and set specific values to both upper and lower bounds on the magnitude of each extended edge shift.

If optimisation was successful, we update the subdivision. As the topology of the optimised subdivision is the same as that of the input, the same faces must be coloured in both versions. A depth-first search procedure on the original subdivision is done to assign a state, either *coloured* or *white*, to each face in the subdivision. Descriptions are now easily computable by just walking along extended edges and checking whether incident faces are coloured or not. The sloped nonogram is now essentially constructed, remains only to solve it to make sure it is of the *simple* type. If that last test is successful, we render the puzzle.

4.2 Allowed inputs

The sloped nonograms built using the aforementioned recipe require fairly strict inputs. Our sloped nonograms only allow four slopes: horizontal ($0°$), vertical ($90°$), positively diagonal ($45°$) and negatively diagonal ($-45°$), therefore inputs should only allow these slopes. Any input where two consecutive vertices cannot be joined by an edge of valid slope will be considered non-valid (see Figure 4.1b). Moreover, an input must be a single or multiple closed polygons. A set of edges creating an open polygon will be rejected even if, when extended, a closed polygon is obtained (see Figure 4.1c). The model will still run if polygons intersect but it then becomes ambiguous as to which face should be coloured and which should be a hole, so it is better to avoid these intersections. Figure 4.1 shows a example of a valid and two non-valid inputs.



(a) Allowed input.

(b) Non-valid input, slope constraints are not satisfied.

(c) Non-valid input, polygon is not closed.

Figure 4.1: Valid vs non-valid inputs. Outer square represents the bounding box.

Inputs are originally stored as 'scalable vector graphics' (.svg) files and need to be converted into sets of Cartesian coordinates representing vertices. A polygon is represented as a sequence of points: $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, which can be converted into a set of n line segments: $((x_0, y_0), (x_1, y_1))$, ...,$((x_n, y_n), (x_0, y_0))$, where for a general line segment: $((x_k, y_k), (x_{k+1}, y_{k+1}))$, (x_k, y_k) represents its starting point and (x_{k+1}, y_{k+1}) its ending point. For each polygon, all of its line segments are extended to the bounding box B using a line to line intersection algorithm [16]. For two line segments L_1 and L_2 represented as $((x_1, y_1), (x_2, y_2))$ and $((x_3, y_3), (x_4, y_4))$, their intersection point (P_x, P_y) is:

$$(P_x, P_y) = \left(\frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \right) \tag{4.1}$$

If L_1 and L_2 are parallel, one obtains $(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) = 0$, which reflects that the gradients of the two line segments are equal, which results in undefined x and y coordinates for the intersection point. Using equation 4.1, the intersection point of each line segment with all four line segments of the bounding box is computed. They are then checked to verify which, out of the four, lie within the range of the bounding box, as shown in Figure 4.2. From the extended edges previously computed and the bounding box, a DCEL is built.

4.3 Optimisation

In this section, the linear programming formulation is given and explained. In addition, the formulation of the constraints and the objective functions are detailed.

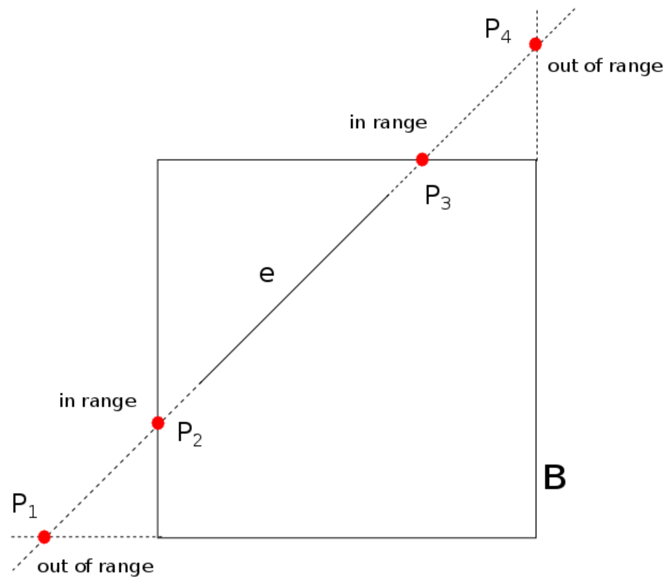


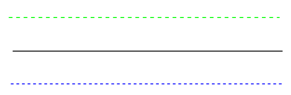
Figure 4.2: Four intersection points between extended edge e and B, only P_2 and P_3 are in range.

The criterion we are dealing with is the size of each face. Since we extended the edges to the bounding box, all faces in the subdivision are convex: "A subset S of the plane is called *convex* if and only if for any pair of points $p, q \in S$ the line segment $\overline{pq}$ is completely contained in S" [6].

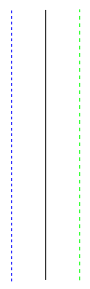
We want to use a distance measure which takes into account the geometry of the face, therefore disallowing overly stretched faces. The area, as a measure, does not satisfy the aforementioned condition. For instance, suppose we have a unit such that 100 of it is considered large. Then a rectangular face of 1 by 100, with $\mathcal{A} = 100$, is large. However its geometry is overly stretched, and according to another measure, which takes the geometry into account, it may be considered small, since two of its sides are of length 1. Using the area as distance measure would enable the linear programming solver to abuse these characteristics and deliver an optimised subdivision containing overly stretched faces.

The width of a face is a better distance measure. The width of a polygon is the smallest distance out of all distances from one side of the polygon to its opposite. In the previous example, the width of our rectangular polygon is $\min(100, 1) = 1$, so this rectangular face would probably require optimisation to increase its width. This distance measure is well-suited to the problem as it takes the geometry of the face into account.

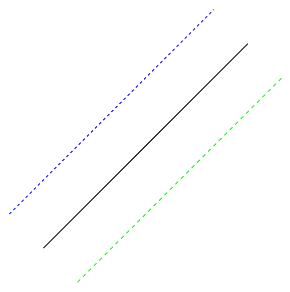
For each extended edge e_i we create a decision variable δ_i with initial value 0, which monitors the amount by which e_i shifts. Edges can shift positively or negatively. For a vertical, positively diagonal and negatively diagonal edge, a positive shift relates to a shift rightwards, and a negative shift relates to a shift leftwards. For a horizontal edge, a positive shift relates to a shift upwards and a negative shift relates to a shift downwards, refer to Figure 4.3.



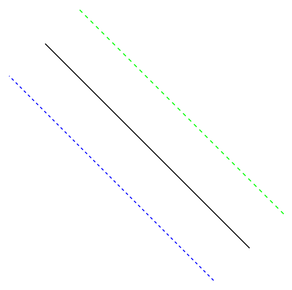
(a) Horizontal edge.



(b) Vertical edge.



(c) Positively diagonal edge.



(d) Negatively diagonal edge.

Figure 4.3: The four different types of edges. The blue dotted line represents a negative shift and the green dotted line represents a positive shift.

4.3.1 Face width constraints

Generating constraints which force the width of each face to be larger than w_t is greatly simplified by the slope constraints.

Lemma 1. *With only four allowed slopes, each face has four different distances between its opposite sides, of which one or more is the width:*

- *a horizontal distance: from the rightmost point or edge to the leftmost point or edge,*
- *a vertical distance: from the highest point or edge to the lowest point or edge,*
- *a positively diagonal distance: from the highest leftmost point or edge to the lowest right point or edge,*
- *a negatively diagonal distance: from the lowest leftmost point or edge to the highest right point or edge.*

Proof. Let us define an axis-aligned square $S = \{(x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1)\}$, with (x_0, y_0) being its bottom left vertex, and the sequence of vertices moving counter-clockwise. The first maximum distance from one side to another in S is

$$dist(((x_0, y_0), (x_1, y_0)), ((x_1, y_1), (x_0, y_1))),$$

which is the vertical distance, i.e. the length of the vertical sides. The second maximum distance in S is

$$dist(((x_1, y_0), (x_1, y_1)), ((x_0, y_1), (x_0, y_0))),$$

which is the horizontal distance, i.e. the length of the horizontal sides. The last two maximum distances in S are the diagonal distances,

$$dist((x_0, y_0), (x_1, y_1))$$

and

$$dist((x_1, y_0), (x_0, y_1)).$$

The same characteristics can be found for convex polygons up to degree 8 satisfying the slope constraints. Therefore there are never more than four constraints per face to check if the width is large enough, since one or several of these distances must be the smallest, hence the width. Refer to Figure 4.4 for the four distances in a triangular face. □

The analysis can be taken further. As a matter of fact, in some cases, some of the aforementioned distances can be discarded as they will never define the width of the face. For the square S, we can identify that the vertical and horizontal distances are always shorter than the diagonal distances.

Lemma 2. *Due to the restrictions on slopes, any point-to-point maximum distance will never define the width of the given polygon.*

Proof. Assume without loss of generality that there is a polygon P with extreme left and right points P_1 and P_2 respectively. Assume that the width of P is defined by the distance between l_1 and l_2, two vertical lines through P_1 and P_2 respectively. Any clockwise or counter-clockwise simultaneous rotation of l_1 and l_2 will decrease the distance from l_1 to l_2 while still enclosing the polygon, decreasing the width. The width can always be decreased up until either l_1 or l_2 is collinear with one of the edge of P. Therefore two extreme points cannot define the width. □

In addition, the geometry of the faces can be further exploited in the case of triangular faces.

Lemma 3. *A triangle satisfying the slope restrictions must be right-angled isosceles.*

Lemma 4. *The width of a right-angled isosceles triangle is always its height.*

So for any triangle, the only constraint is that its height must be larger or equal to w_t.

For an arbitrary face, if all maximum distances are larger than w_t, the width is also larger than w_t.

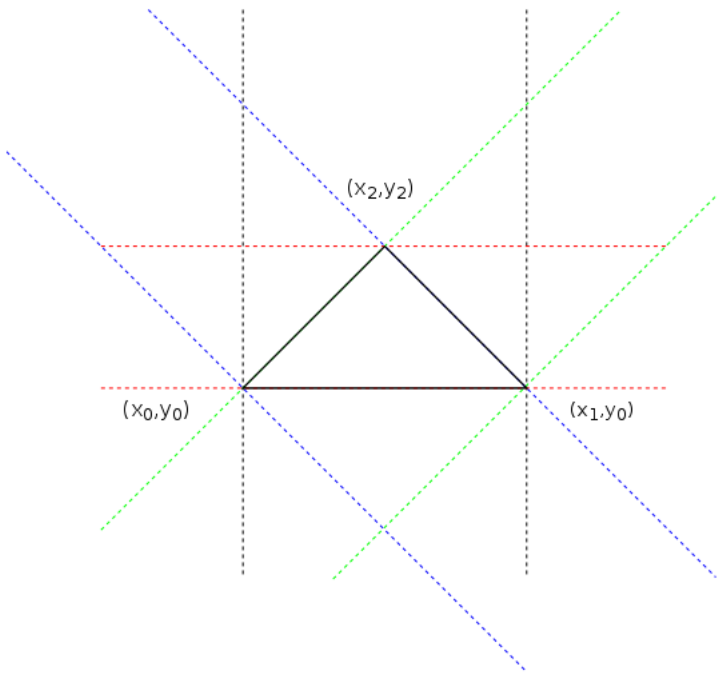


Figure 4.4: Right-angled isosceles triangle, distances are the length between dotted lines of the same colour. Horizontal distance (black), vertical distance (red), positively diagonal distance (green) and negatively diagonal distance (blue).

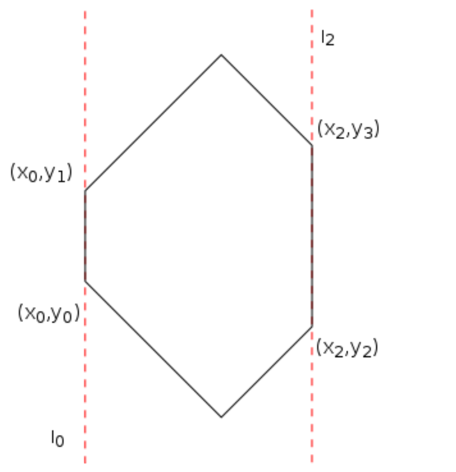
Let us now demonstrate how to compute a horizontal constraint. For the horizontal constraint, we first compute the number of vertical edges in the face, it may be zero, one or two. Convex properties are such that if the face has two vertical edges, the horizontal distance is defined by the distance between those two edges. To compute the distance, we use the x-coordinate of the vertical lines.

Let there be a face with two vertical lines segments $l_0 = ((x_0, y_0), (x_0, y_1))$ and $l_2 = ((x_2, y_2), (x_2, y_3))$ with $x_2 > x_0$ (so l_0 is the leftmost vertical line segment and l_2 is the rightmost vertical line segment in the face), refer to Figure 4.5a. Each line segment l_i has an associated variable δ_i which monitors how much the line segment l_i shifts, leftwards or rightwards. The constraint is formulated as:

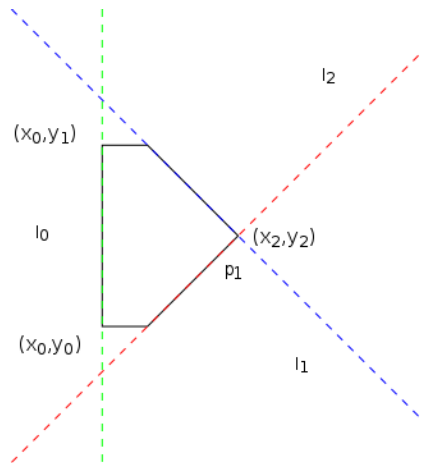
$$\begin{aligned}(x_2 + \delta_2) - (x_0 + \delta_0) &\geq w_t \\ \delta_2 - \delta_0 &\geq w_t - x_2 + x_0\end{aligned} \tag{4.2}$$

Therefore, on the one hand, if the distance $|x_2 - x_0|$ is already equal to or larger than w_t, we may set δ_0 and δ_2 to 0 or negative to decrease the distance between l_0 and l_2. On the other hand, if the distance is not large enough, the values of δ_0 and δ_2 must be set such that the constraint is satisfied.

Let there be a new face in which the horizontal distance is defined by one line segment $l_0 = \{(x_0, y_0)(x_0, y_1)\}$ and one point $p_1 = (x_2, y_2)$, which is the intersection point of line segments l_1 and l_2, refer to Figure 4.5b. The horizontal constraint will be generated using the distance from l_0 to p_1. It can be formulated as:



(a) Horizontal constraint defined by two line segments.



(b) Horizontal constraint defined by one line segment and one point.

Figure 4.5: Two cases for computing the horizontal constraint.

$$\begin{aligned} (x_2 + \frac{\delta_2}{\sqrt{2}} + \frac{\delta_1}{\sqrt{2}}) - (x_0 + \delta_0) \geq w_t \\ \frac{\delta_2}{\sqrt{2}} + \frac{\delta_1}{\sqrt{2}} - \delta_0 \geq w_t - x_2 + x_0 \end{aligned} \tag{4.3}$$

If there is no vertical line segment in the face, then the horizontal distance is defined by two points, and therefore may never define the actual width of the face; so this case is not explored.

The vertical constraint is computed similarly, except that instead of using the x-coordinates of the vertical line segments or points to measure the distance, we use the y-coordinates of horizontal line segments or points.

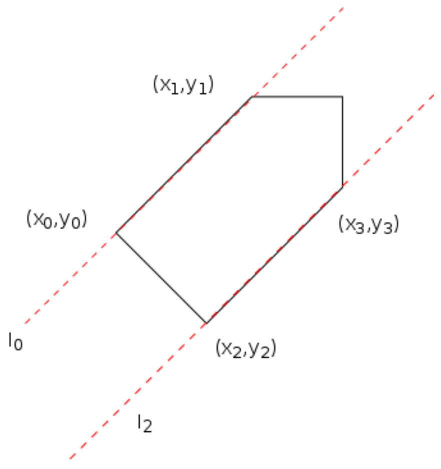
For the positively diagonal constraint, the number of positively diagonal line segments in the face is computed, either zero, one or two. If two positively diagonal line segments are found, then the positively diagonal distance is defined by the distance between those two line segments.

To get the distance between two diagonal line segments, we need to find a reference point for each line segment. The y-intersect is used as reference point for each line segment mainly because it is easy to compute.

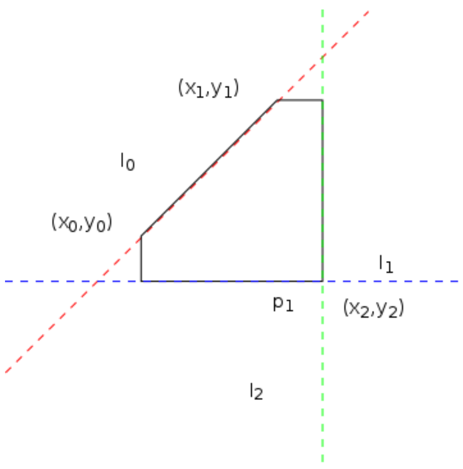
Lemma 5. *A positively diagonal line segment containing a point $P = (x, y)$ intersects the y-axis at point $P_{intersection} = (0, x - y)$. A negatively diagonal line segment containing a point $P = (x, y)$ intersects the y-axis at point $P_{intersection} = (0, x + y)$.*

For a face with two positively diagonal line segments $l_0 = ((x_0, y_0), (x_1, y_1))$ and $l_2 = ((x_2, y_2), (x_3, y_3))$, where l_0 is above l_2 (refer to Figure 4.6a), the positively diagonal constraint can be formulated as:

$$\begin{aligned} \frac{1}{\sqrt{2}}[(y_0 - x_0 - \sqrt{2}\delta_0) - (y_2 - x_2 - \sqrt{2}\delta_2)] \geq w_t \\ -\sqrt{2}\delta_0 + \sqrt{2}\delta_2 \geq \sqrt{2}w_t - y_0 + x_0 + y_2 - x_2 \end{aligned} \tag{4.4}$$



(a) Positively diagonal constraint defined by two line segments.



(b) Positively diagonal constraint defined by one line segment and one point.

Figure 4.6: Two cases for computing the positively diagonal constraint.

For the positively diagonal constraint when the positively diagonal distance is defined by one line segment $l_0 = ((x_0, y_0), (x_1, y_1))$ and one point $p_1 = (x_2, y_2)$, as shown in Figure 4.6b, the constraint can be formulated as:

$$\begin{gathered}\frac{1}{\sqrt{2}}[(y_0 - x_0 - \sqrt{2}\delta_0) - (y_2 - x_2 - \delta_2 + \delta_1)] \geq w_t \\ -\sqrt{2}\delta_0 + \delta_2 - \delta_1 \geq \sqrt{2}w_t - y_0 + x_0 + y_2 - x_2\end{gathered} \tag{4.5}$$

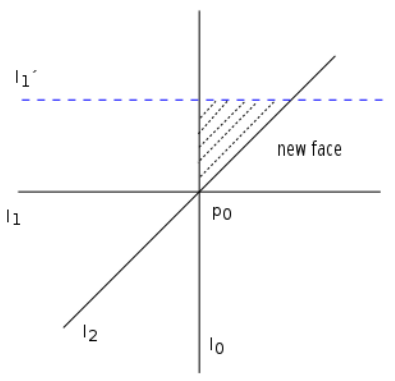
If the face does not have any positively diagonal line segments, then the positively diagonal distance is defined by two points and therefore will not define the width of the face, so it will be disregarded. The negatively diagonal constraint is computed analogously.

For each face in the subdivision, we generate up to four constraints, forcing each maximum distance (horizontal, vertical, positively diagonal and negatively diagonal) to be larger or equal than w_t. As at least one of those four distances must define the width, having them all large enough ensures that the width of the face is large enough.

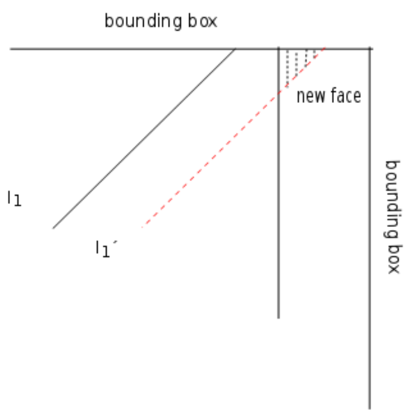
4.3.2 Intersection constraints and bounds

The constraints ensuring that faces are large enough do not prevent creation or deletion of faces. While optimising, new small faces may be created, which would be very counter-productive. Not handling triple and quadruple intersections during optimisation results in the creation of new faces, see Figure 4.7a. In addition, if edges in the subdivision are not bounded on the amount they can shift, faces bordering the bounding box may be created or removed, see Figure 4.7b.

In order to not create or remove faces, we must keep triple and quadruple intersections intact and bound each δ_i. Let there be three line segments l_0 with slope 90°, l_1 with slope 0° and l_2 with slope 45°, intersecting at point p_0, see Figure 4.7a.



(a) Face creation at triple intersection after optimisation, l'_1 is the optimised l_1.



(b) Face creation near bounding box, l'_1 is the optimised l_1.

Figure 4.7: Possible face creation when optimising only with constraints on the size of faces.

In order to keep the triple intersection in the optimised subdivision, the following constraint must be satisfied:

$$\delta_1 - \delta_0 + \sqrt{2}\delta_2 = 0 \tag{4.6}$$

If one or more of the line segments l_0, l_1, and l_2 are shifted during optimisation, the others will be forced to shift accordingly, to keep the triple intersection intact. Let l_1 shift 1 unit up ($\delta_1 = 1$), see Figure 4.7a. Due to this shift, a new face is created, therefore either l_0 or l_2 or both must be shifted to keep the triple intersection and thus removing that new face. Assume that δ_0 is set to 0 by some other constraint. Only l_2 can be shifted to keep the triple intersection. We have $\delta_1 = 1$ and $\delta_0 = 0$:

$$\begin{aligned}(1) - (0) + \sqrt{2}\delta_2 &= 0\\ \sqrt{2}\delta_2 &= -1\\ \delta_2 &= \frac{-1}{\sqrt{2}}\end{aligned} \tag{4.7}$$

The triple intersection constraint forces l_2 to shift $\delta_2 = \frac{-1}{\sqrt{2}}$ units. Table 4.1 shows the possible configurations for triple intersections and their respective constraint. To generate a quadruple intersection constraint, just model it as two triple intersection constraints.

Let us introduce the concept of neighbour for extended edges, crucial to bounding δ_i for all extended edges e_i in the subdivision. Each extended edge intersects the bounding box twice. Extended edges may have up to two neighbours on sides of the bounding box with which they intersect. They have no neighbour on the other sides of the bounding box. We call extended edge e_j the neighbour of extended edge e_i on a particular side of the bounding box, $side_k$, if they both intersect the bounding box

slope l_0	slope l_1	slope l_2	Constraint
0	45	90	$\delta_0 - \delta_1 + \sqrt{2}\delta_2 = 0$
0	-45	90	$\delta_0 + \delta_1 - \sqrt{2}\delta_2 = 0$
0	45	-45	$\delta_0 + \frac{\sqrt{2}}{2}\delta_1 - \frac{\sqrt{2}}{2}\delta_2 = 0$
90	45	-45	$\delta_0 - \frac{\sqrt{2}}{2}\delta_1 - \frac{\sqrt{2}}{2}\delta_2 = 0$

Table 4.1: Triple intersection constraints.

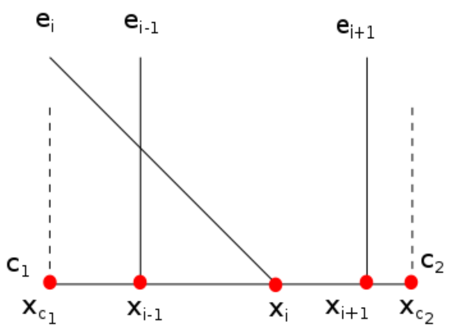
on $side_k$ at (x_j, y_j) and (x_i, y_i) respectively, and if (x_j, y_j) is one of the potentially two closest point to (x_i, y_i) on $side_k$.

For each extended edge e_i, we must bound the amount by which it can shift. We define the lower and upper bound of δ_i as lb_i and ub_i respectively. We must enforce $lb_i < \delta_i < ub_i$, where lb_i is the magnitude of the maximal possible negative shift ($lb_i \leq 0$), such that e_i keeps the same previous neighbour, and ub_i is the magnitude of the maximal positive shift ($ub_i \geq 0$), such that e_i keeps the same next neighbour. Three cases can be distinguished:

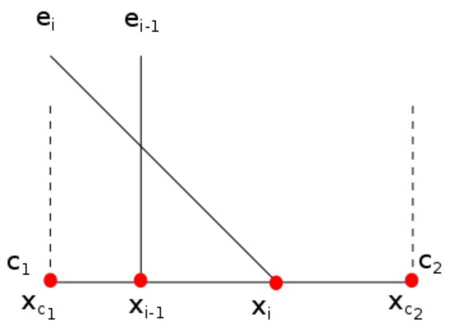
- an extended edge e_i intersecting the bounding box has two neighbouring extended edges, a previous neighbour e_{i-1} and a next neighbour e_{i+1}, see Figure 4.8a,
- an extended edge e_i intersecting the bounding box has only one neighbouring extended edge, either a previous neighbour e_{i-1} or a next neighbour e_{i+1}. On the other side is the corner of the bounding box, either c_2 or c_1, see Figure 4.8b,
- an extended edge e_i intersecting the bounding box has no neighbouring extended edge, only corners of the bounding box c_1 and c_2, see Figure 4.8c.

For the first case, see Figure 4.8a, extended edge e_i intersects the bottom side of the bounding box at (x_i, y_i), its previous neighbour is extended edge e_{i-1}, intersecting the bottom side of the bounding box at (x_{i-1}, y_i) and its next neighbour is extended edge e_{i+1}, intersecting the bottom side of the bounding box at (x_{i+1}, y_i), with $x_{i-1} < x_i < x_{i+1}$. The lower bound on δ_i is $lb_i = \frac{-1}{\sqrt{2}}|(x_i - x_{i-1})|$, and the upper bound on δ_i is $ub_i = \frac{1}{\sqrt{2}}|(x_{i+1} - x_i)|$. Depending on the slope of e_i, the factor of $\frac{1}{\sqrt{2}}$ may need to be replaced by 1.

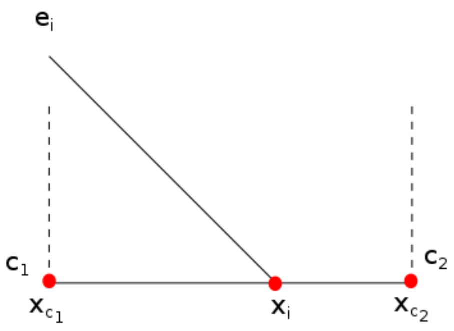
In the case in which the two intersection points of e_i with the bounding box have the corners of the bounding box as neighbours, the two sets of bounds on δ_i found are pairwise the same. All other cases yield two sets of lower and upper bounds on δ_i. The lowest absolute value for the lower and upper bounds are kept as the actual lower and upper bounds for δ_i. Remember that in order to simplify the model, we do not allow extended edges to change the side of the bounding box with which they intersect. Therefore, for the example shown in Figure 4.8b, the lower bound on δ_i is $lb_i = \frac{-1}{\sqrt{2}}|(x_i - x_{i-1})|$ and the upper bound on δ_i is $ub_i = \frac{1}{\sqrt{2}}|(x_{c_2}) - x_i|$. If there is no neighbour, we use the corner of the bounding box on both sides to define the bounds of δ_i. For the example shown in Figure 4.8c, we get $lb_i = \frac{-1}{\sqrt{2}}|(x_i - x_{c_1})|$ and $ub_i = \frac{1}{\sqrt{2}}|(x_{c_2} - x_i)|$. Since the bounding box must remain the same, we force the δ values for its edges to be 0 by fixing both their lower and upper bound to 0.



(a) Extended edge e_i with its previous neighbour e_{i-1} and its next neighbour e_{i+1}.



(b) Extended edge e_i with only one neighbouring extended edge e_{i-1} and the corner of the bounding box c_2 on the other side.



(c) Edge e_i with no neighbouring edge, only the two corners of the bounding box, c_1 and c_2.

Figure 4.8: Three possible arrangements for a single extended edge e_i intersecting the bottom side of the bounding box. Points c_1 and c_2 represent the corners of the bounding box, with x-coordinates x_{c_1} and x_{c_2} respectively.

4.3.3 Objective functions

For our application, it is interesting to use multiple objective functions in order to compare visual outputs originating from optimisation. We will be using three functions: min sum of squares ($\min \sum^2$), min sum ($\min \sum$) and min max ($\min \max$). The function $\min \sum^2$ minimises the sum of the squared values of the δs, see equation 4.8. This function does not take into account the direction in which extended edges are shifted. However it greatly penalises large extended edge shifts because of the squaring function. When optimising with $\min \sum^2$, the most natural output seems to be small shifts over many extended edges.

$$\min \sum_{i=0}^{n'} \delta_i^2, \tag{4.8}$$

where n' is the total number extended edge (n) plus the four bounding box edges.

The second objective function is $\min \sum$. It amounts to minimising the sum of all δs. We previously saw that this problem is not linear because of the absolute value involved. Yet we can transform it into a set of linear programmes. To do so, we create 2^n different linear programmes, where n is the number of extended edges excluding the bounding box edges. Each different linear programme has a different configuration of the bounds on the δs, they are either forced to be positive ($lb = 0$) or negative ($ub = 0$). If an extended edge e_i is forced to shift negatively by setting $ub_i = 0$, its value in the objective is changed to $-1 \times \delta_i$, since minimising a negative value does not make sense.
For instance, if we have a subdivision with two extended edges e_0 and e_1, the two optimisation variables are δ_0 and δ_1. We construct 2^n different linear programmes, see Table 4.2. We can then solve all 2^n linear programmes.

δ_0	δ_1	Objective
≤ 0	≤ 0	$\min -\delta_0 - \delta_1$
≤ 0	≥ 0	$\min -\delta_0 + \delta_1$
≥ 0	≤ 0	$\min \delta_0 - \delta_1$
≥ 0	≥ 0	$\min \delta_0 + \delta_1$

Table 4.2: All 2^n different linear programmes for a two extended edges subdivision.

The final objective function is $\min \max$. It minimises the maximum shift on all edges such that all constraints are satisfied. This function heavily penalises large shifts. A new optimisation variable *displacement* is defined, as well as several constraints involving *displacement*. The linear programme becomes:

$$\min displacement$$

such that face size, topology constraints, triple and quadruple constraints and new constraints:

$$\delta_i \leq displacement, -\delta_i \leq displacement,$$

$\forall \delta_i, i = 0, ..., n'$ are satisfied.

4.4 Sloped nonogram construction

Once the optimisation of the subdivision is completed, we shift each extended edge e_i by its optimisation value δ_i. For a vertical extended edge e_i originally from $p_a = (x_a, y_a)$ to $p_b = (x_a, y_b)$ being optimised by δ_i, the new endpoints become $p'_a = (x_a + \delta_i, y_a)$ and $p'_b = (x_a + \delta_i, y_b)$. Similarly, for a horizontal extended edge e_j originally from $p_a = (x_a, y_a)$ to $p_b = (x_b, y_a)$ optimised by δ_j, the new endpoints become $p'_a = (x_a, y_a + \delta_j)$ and $p'_b = (x_b, y_a + \delta_j)$. For a positively diagonal extended edge e_k from $p_a = (x_a, y_a)$ to $p_b = (x_b, y_b)$ optimised by δ_k, if its starting point p_a intersects the left side of the bounding box and p_b the top side of the bounding box, the new endpoints become $p'_a = (x_a, y_a - \sqrt{2}\delta_k)$ and $p'_b = (x_b + \sqrt{2}\delta_k, y_b)$. If e_k's starting point p_a intersects with the bottom side of the bounding box and p_b with the right side of the bounding box, then the new endpoints become $p'_a = (x_a + \sqrt{2}\delta_k, y_a)$ and $p'_b = (x_b, y_b - \sqrt{2}\delta_k)$. For a negatively diagonal extended edge e_l optimised by e_l, always shift the appropriate endpoint by $+\sqrt{2}\delta_l$. This new subdivision represents the optimised puzzle satisfying the face size and topology constraints.

4.4.1 Depth-first search

This section describes how faces in the original subdivision are visited and assigned the status of *coloured* or that of *white*. We use the original subdivision since our optimisation is topologically identical to the original, therefore the same faces are *coloured* or *white* in both subdivisions. We can travel through the faces of the subdivision using a depth-first search (DFS) method and check whether input edges are crossed. Input edges are edges which belong to the input only, thus not the extensions to the bounding box. When crossing such an edge, we are either entering or exiting the drawing, so if the previous face was *white*, the new one must be *coloured* and vice versa. We need a way of knowing whether the first face in which we start our search is *white* or *coloured*. Outside the subdivision, faces are always *white*, so if the boundary of the first face is part of the input, then the first face is *coloured* else it is *white*. The search ends when all faces have been explored and assigned either *coloured* or *white*.

4.4.2 Collinearity and distance

Note that in the DFS procedure a DCEL is used to store the faces, edges and vertices of the subdivision. Some of those DCEL-edges might only be small parts of input edges. We can easily compute whether a DCEL-edge is collinear to an input edge, then using a distance property, we can compute whether the endpoints of the DCEL-edge are on or within the endpoints of the input edge. If those two conditions are met, then moving across the DCEL-edge amounts to moving across an input edge. Collinearity between three points a, b, c is true if the following expression is true:

$$\begin{aligned}(b_x - a_x)(c_y - a_y) &= (c_x - a_x)(c_y - a_y)\\ \frac{c_y - a_y}{c_x - a_x} &= \frac{b_y - a_y}{b_x - a_x}\\ m_{ac} &= m_{ab},\end{aligned} \tag{4.9}$$

where m_{ab} represents the gradient of the line segment from point a to point b.

If both gradients are equal and both line segments have point a as an endpoint, then the three points are collinear. Checking whether the endpoints of the DCEL-edge are on or in between the endpoints of the input edge is done by comparing the x-coordinates or the y-coordinates of the endpoints. If three points a, b, c are collinear on a non-vertical line segment, we can check if c is between a and b by comparing the x-coordinates: $a_x < c_x < b_x$. If the line is vertical, use the y-coordinates: $a_y < c_y < b_y$.

4.4.3 Total edges

Total edges are custom data structures used to gather information about extended edges, such as description (sloped nonogram related) and the number of adjacent faces. For any extended edge e_i, two total edges are needed, see Figure 4.9. The first one, T_1 monitors information about the above adjacent faces to the extended edge e_i and T_2 monitors information about below adjacent faces to the extended edge e_i.

Using the DCEL-edges from the subdivision, we can easily step into the adjacent DCEL-faces and check whether they are *coloured* or *white*. Moving along an extended edge e_i amounts to moving along a set of DCEL-edges. DCEL-edges are separated by DCEL-vertices, from which we can travel to any incident DCEL-edge. To travel along an extended edge e_i, we start at one of the DCEL-vertex of e_i on the bounding box. We walk along the first DCEL-edge of e_i. We remember the slope of that DCEL-edge. Once arrived at a DCEL-vertex, we check whether that vertex has an incident DCEL-edge with the same slope as the DCEL-edge we came from, while not being the twin of the DCEL-edge we came from. If it does, walk along that new DCEL-edge and repeat the procedure, else we must be at the other side of the bounding box and done with our walk.

Each time we walk along a new DCEL-edge, we check if its incident DCEL-face is *coloured* or *white*. It is important to realise that we only use DCEL-faces which are *edge adjacent* to DCEL-edges, as opposed to *vertex adjacent*. This means that for a face to be adjacent to an edge, they must share an edge, and not a vertex only. This detail becomes very important when solving sloped nonograms. We can construct the description: $s_1, ..., s_k$, for k black runs along the total edge, since we know which adjacent DCEL-faces are *coloured* and which are *white*. We repeat this process for every total edge.

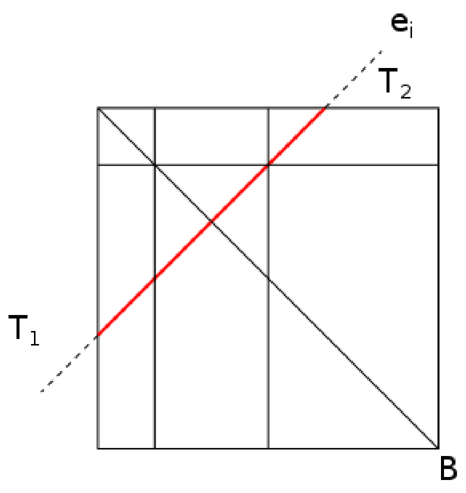


Figure 4.9: An extended edge e_i and its two respective total edges T_1 and T_2.

4.4.4 Solvers

This section explores the two solvers used throughout the project, the linear and quadratic programming solver and the *simple* regular nonogram solver. Firstly let us discuss the Gurobi LP solver used for the optimisation of the size of the faces in the subdivision. Secondly we will discuss the *simple* nonogram solver.

Gurobi solver

To optimise the subdivision, the Gurobi LP solver was used. Its basic functionality closely resembles the theoretical recipe, with definition of decision variables, generation of an objective function, to be either minimised or maximised, creation of constraints and setting of lower and upper bounds for each decision variable.

Nonogram solver and difficulty measure

Given a nonogram, we must be able to say whether it is uniquely solvable and how difficult it is. We used a third party software which can solve all *simple* regular nonograms [14]. Although our generated sloped nonograms are not regular, since they use faces instead of cells, and edges instead of rows and columns, they can be solved analogously to regular nonograms. For a regular nonogram, most solvers operate row by row and column by column, so no information regarding multiple rows and columns is needed simultaneously. The solver starts in a row or column and generates all possible solutions for that row or column. In a solution, each cell is either *coloured*, *white* or *unknown*. If some cells are already *coloured* or *white*, the solver takes that into consideration and returns fewer possible solutions. From all the different solutions, if one cell has the same state (*coloured* or *white*) in all of them, then that cell must have that particular state in the solved version. The solver determines the final state of as many cells as possible in the current row and updates the entire structure with those new states. For instance in Figure 4.10, we see a row with 4 cells and a description of d = (*1,2*). The only possible solution for the solver is: *coloured*, *white*, *coloured*, *coloured*. In this case there is only one solution which completely solves the row. The puzzle would then be updated and the solver would move onto another column. When no more information can be determined about a certain row or column, the solver moves onto the next column or row. The solver stops when the state of all cells is either *coloured* or *white*, or when no more progress can be done in any row or column, in other words, the puzzle is not *simple*.

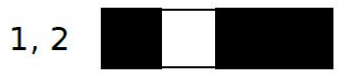


Figure 4.10: A solved row with four cells and description d = (*1,2*).

The main strength of the solver is that it only changes the state of a cell if and only if the new state is certain. As mentioned, the solver cannot solve all nonograms, because more complex puzzles require assumptions or even information about multiple rows and columns to determine the state of a single cell. Those features are not implemented in the algorithm.

As aforementioned, the solving routine for sloped nonograms is analogous to that of regular nonograms. For our puzzles, the solver works as follows: the state of all faces is initialised to *unknown*. The solver starts at a total edge. From the description of that total edge and the configuration of faces adjacent to that total edge, the solver generates all possible solutions and compares them to see if one or more faces have the same state in all solutions. If so, the state of those faces are set and the whole subdivision is updated. While the state of all faces is not known, the procedure is repeated using the next total edge in either clockwise or counter-clockwise order. Eventually, the state of all faces will be found and the puzzle solved. It can happen that for a total edge, the state of none of its adjacent faces can be determined. If so, then the solver moves onto the next total edge. If for all total edges, the solver cannot find the state of any face, then it gets stuck and the puzzle cannot be considered of *simple* type. The difficulty measure of a solvable *simple* sloped nonograms is the number of total edges, where progress is made (the state of at least one face is determined), required to know the state of all faces in the subdivision. It may be more than the amount of total edges, if after going through all total edges, the state of some faces is still *unknown*. We defined the difficulty as such because it is simple and straightforward to compute, and we believe it will scale well with the complexity of the sloped nonograms. The main shortcoming of this measure is that it truly computes the number of steps required to solve the puzzle, as opposed to the actual difficulty. As a matter of fact, a puzzle solved through many straightforward steps will be considered more difficult than a puzzle consisting of few complex steps according to our difficulty measure. However, since we are only investigating iteratively solvable sloped nonograms, our difficulty measure is quite acceptable. Appendix 8.2 shows several of our generated sloped nonograms with their solution.

Chapter 5

Experiments

5.1 Experimental set-up

The sloped nonogram generation process must be a fast process, as we want interactive use with users, so the waiting time for generating a sloped nonogram from an input drawing should be short. Moreover, the process should generate quality output sloped nonograms. The criterion for the quality of an output sloped nonogram is the resemblance, once solved, with its input drawing. As the puzzle generation process is based on optimisation, we want to compare how well each objective function: $\min \sum^2$, $\min \sum$ and $\min \max$ performs from both a speed and quality output point of view. In addition, we want to investigate the relationship between the maximum minimal width obtainable, and difficulty, both as functions of the number of lines n (excluding bounding box lines). To test the generation process speed for each objective function, we will benchmark the speed[1] of the optimisation process, because other parts of the generation process such as formatting data and rendering the puzzle are the same in all three generation processes (using $\min \sum^2$, $\min \sum$ and $\min \max$). With regards to quality of output sloped nonograms, we will visually determine whether an output sloped nonogram, when solved, resembles its input. The maximum minimal width will be calculated using a binary-search technique. Finally, the difficulty of each puzzle will be monitored using the nonogram solver previously described. For experimenting we will use the inputs shown in Figure 5.1. An explanation of why such inputs were chosen is given in Appendix 8.1.

It seems clear that solving all 2^n linear programmes for $\min \sum$ becomes infeasible as n becomes too large. Using 7 inputs with n = 3, 6, 9, 12, 13, 14 and 15 respectively (Figures: 5.2a, 5.2b, 5.2c, 5.2d, 5.2e, 5.2f and 5.2g), we timed how fast $\min \sum$ performed for solving all 2^n linear programmes for each input, see Table 5.1. Using these results we determined the value of n, n_{max}, until which we allow $\min \sum$ to solve all 2^n linear programmes such that the time taken by $\min \sum$ is feasible. For inputs with $n > n_{max}$ for $\min \sum$, we shall only create m, with $m < 2^n$, different linear programmes and solve these. Those linear programmes will be constructed through the use of random numbers. For all m linear programmes, a different random number k in the range $0 \leq k \leq 2^n - 1$ is drawn, converted to binary of length n. Each of the $bin_n(k)_i$, $(bin_n(k)_0, bin_n(k)_1, ..., bin_n(k)_n)$ where $bin_n(k)$ is the binary

[1]Throughout the thesis, we use elapsed time, however all computations do not require I/O or user input, thus we believe elapsed time to be comparable to CPU time.

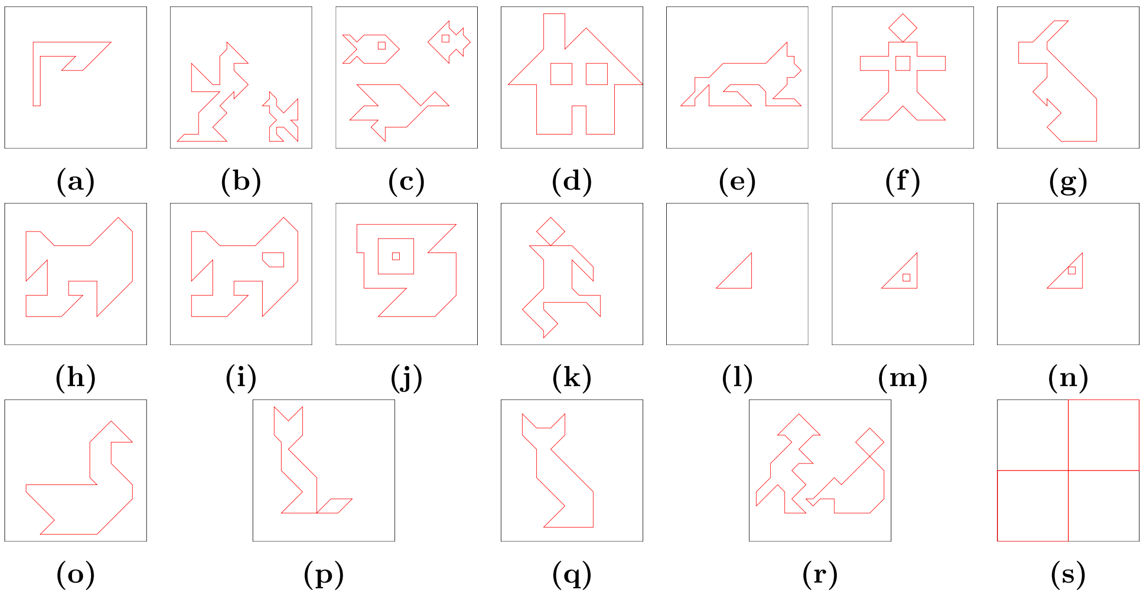


Figure 5.1: The 19 inputs used for testing.

equivalent of k with length n, and i is used for indexing, is either 0 or 1. If it is 0 then δ_i is forced to be negative, else it is forced to be positive. By choosing different values of k, we always create different linear programmes.

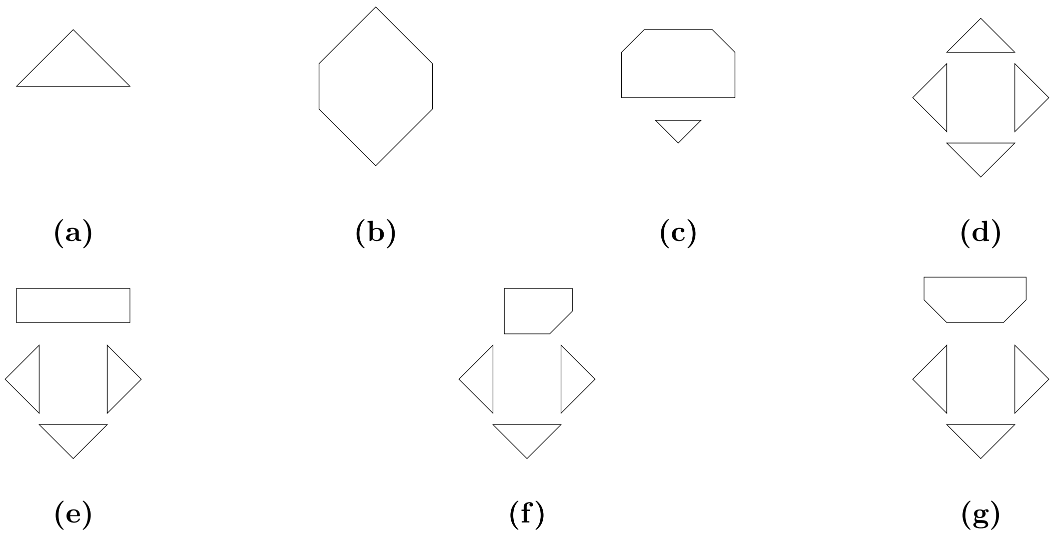


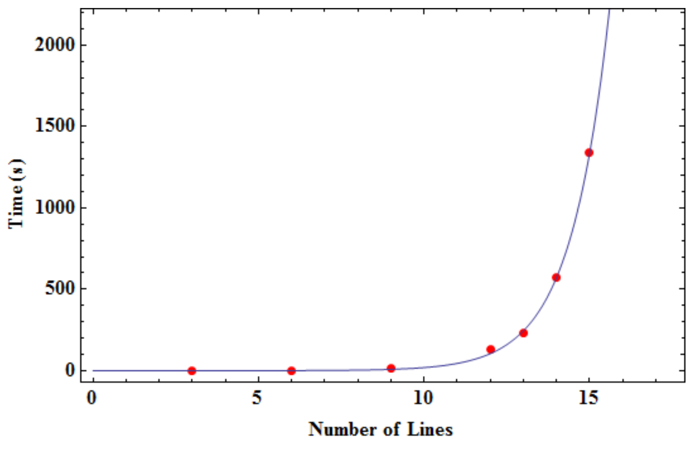
Figure 5.2: Inputs with $n = 3, 6, 9, 12, 13, 14, 15$, Figures 5.2a, 5.2b, 5.2c, 5.2d, 5.2e, 5.2f and 5.2g respectively, used to time $\min \sum$.

Since $\min \sum$ requires solving 2^n linear programmes, we expect a relationship of the form $O(2^n)$ between optimisation solving time and n. From the data shown in Table 5.1, we notice an overall relationship slightly faster than doubling. We believe that this is due to the setting up of the linear programmes before solving. As a matter of fact, the larger n, the more δ variables need to be assigned bounds (either positive or negative). This set-up increases the total time required for $\min \sum$, so we get an increase slightly faster than doubling. We see that for small values of n the

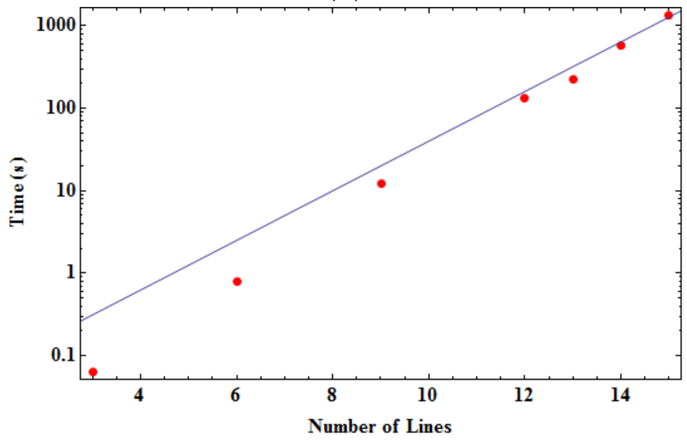
n	time (s)
3	0.063
6	0.797
9	12.105
12	134.143
13	227.647
14	572.110
15	1341.302

Table 5.1: Optimisation time for min $\sum$ for inputs with n = 3, 6, 9, 12, 13, 14 and 15.

relationship grows faster than for larger values of n. It is explained by the fact that the time required for solving the linear programmes, once they have been set-up, for small n, is very short, so the setting up time heavily affects the overall time. For large values of n, the solving time, once the linear programmes have been set-up, is rather large compared to the setting up time, so the influence is not as large. Figure 5.3 shows a superposition of the data with a best fit function of the form $f(n) = a2^n$ plotted on both a regular and a logarithmic scale.



(a)



(b)

Figure 5.3: Figure 5.3a shows min $\sum$ data for $n = 3, 6, 9, 12, 13, 14, 15$ and the best fit function. Figure 5.3b uses logarithmic scale on the y-axis.

As n increases, the time-feasibility of solving all 2^n linear programmes in $\min \sum$ shrinks. To solve this issue, we chose $n_{max} = 12$. In the extreme case $n = 12$, $2^{12} = 4096$ linear programmes require solving, which is feasible from a time point of view (see Table 5.1). Therefore, for inputs with 12 lines or fewer, $\min \sum$ will solve all 2^n linear programmes. For inputs with more than 12 lines, $\min \sum$ will use the aforementioned sampling process, using several values for m, $m = 10, 50, 100, 500, 1000, 5000$. We used values only up to 5000 because it was a close value to $2^{12} = 4096$, and is still large enough to probably generate linear programmes with configurations of δs which allow for feasible optimisation with $\min \sum$.

We want to visually compare the solutions found using each objective function for $w_t = 2$, $w_t = 3$ and $w_t = 4$, to check which objective function yields the best visual results, where we judge how similar the output solved sloped nonograms are to their input. The more deformed an output is from its input, the worse it visually appears.

For each input, the final value of the objective for $\min \sum^2$, $\min \sum$ and $\min \max$ will be monitored for comparison. When optimising the subdivision according to a method A, we shall measure the value of the objective for the other two methods B and C, according to decision variable values found through optimising according to A. The objective value obtained when optimising according to method A should be smaller or equal to that of the objective value for method A measured from the optimised variables found through optimising according to B or C.

The maximum minimal possible width threshold of each puzzle will be calculated, to gain insight on how the number of lines n affects the final maximum minimal width of all faces in the subdivision. To compute this maximum minimal possible width, we use a binary-search procedure. We know that optimisation is always possible for a target width of length 0. Therefore, we label $passedthreshold = 0$. If optimisation is possible for a target width of length 1, then $passedthreshold = 1$ and target threshold becomes $2 \times passedthreshold$, else $failedthreshold = 1$ and target threshold becomes the middle point between the last $passedthreshold$ and the $failedthreshold$. While no $failedthreshold$ is found, target threshold is obtained by doubling the last $passedthreshold$. This search is continued until the width result starts to converge towards a value.

Lastly we want to investigate the difficulty of a sloped nonogram, according to the difficulty measure described in Section 4.4.4, as a function of the number of lines n. The difficulty measure keeps track of the number of total edge passes, where progress is made, the solver requires to completely solve the sloped nonogram. This measure is not as representative of difficulty as of the number of steps required to solve the sloped nonogram; however in our case that is acceptable.

5.2 Results

All tests were performed using Python 2.7.9 on a machine with an Intel(R) Core (TM) i7-4720HQ @ 2.60GHz with 8 GB of RAM. Table 5.2 shows the abbreviation used to reference input files (input drawings) in plots, as well as information regarding the inputs.

file / input drawing	figure reference	abbreviation	n
back_seven	5.1a	b	8
dragon_bird	5.1b	db	33
fish	5.1c	f	35
house	5.1d	h	16
lion	5.1e	l	21
man_holes	5.1f	mh	18
rabbit	5.1g	r	15
random	5.1h	ra	16
random_hole	5.1i	rah	20
recurs_holes	5.1j	re	17
running	5.1k	ru	18
simple_triangle	5.1l	s	3
simple_triangle_hole	5.1m	sh	7
simple_triangle_hole_tangent	5.1n	sht	7
swan	5.1o	sw	14
tan_cat	5.1p	t	14
tan_cat_new	5.1q	tn	13
two_lads	5.1r	tl	27
two_squares	5.1s	ts	2

Table 5.2: Input drawing names, references, abbreviations and n.

5.2.1 Visual comparison

Given an input drawing, we wish to generate a sloped nonogram which satisfies the face size and topology constraints while still looking similar to its input once solved. For this investigation, we shall look at three different values for w_t: $w_t = 2$, $w_t = 3$ and $w_t = 4$. Inputs used for investigating $w_t = 2$ and $w_t = 3$ are: 'house', 'swan' and 'tan_cat'. We chose those inputs because optimisation results were more noticeable than for other inputs. For $w_t = 4$, in both 'house' and 'tan_cat' optimisation was infeasible, we therefore used 'back_seven', 'swan' and 'simple_triangle_hole_tangent'.

Optimisation for $w_t = 2$ yielded uninteresting results, since none of the inputs contain faces with a width smaller than 2. Therefore, face size constraints were satisfied without use of optimisation. So all our inputs can be converted into sloped nonograms in which each face has a width larger or equal to $w_t = 2$. We thus decided to investigate output sloped nonogram using $w_t = 2.6$ instead. We chose this value since it is fairly larger than 2, so optimisation may be more fruitful, while still being far enough away from the next step, namely $w_t = 3$.

Let us now compare the objective values obtained using $\min \sum^2$, $\min \sum$ and $\min \max$ for $w_t = 2.6$ (Table 5.3), $w_t = 3$ (Table 5.4) and $w_t = 4$ (Table 5.5). We measured the objective values for each objective function using the decision variable values obtained using each objective function sequentially. Tables 5.4, 5.3 and 5.5

show the score of each objective function, for inputs where optimisation occurred, thus excluding infeasible models and models in which no optimisation was required. The first main column shows values for $\min \sum^2$, $\min \sum$ and $\min \max$ when optimised according to $\min \sum^2$. The second main column, when optimising using $\min \sum$ ($m = 5000$ if $n > 12$), and the third main column when optimising using $\min \max$. We use $m = 5000$ in $\min \sum$, because the probability of obtaining a better solution is higher than when using smaller values for m. The sub-columns show the score of the three objective functions. The coloured values show the best optimal values according to different objective functions, red is for $\min \sum^2$, blue is for $\min \sum$ and green is for $\min \max$.

input	min $\sum^2$			min $\sum$			min max		
	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max
h	0.02	0.2	0.1	0.02	0.2	0.1	0.09	0.9	0.1
mh	0.28	1.56	0.2	0.28	1.56	0.2	0.28	1.56	0.2
r	0.31	1.46	0.35	0.31	1.46	0.35	0.54	2.29	0.29
ra	0.06	0.34	0.2	0.06	0.34	0.2	0.31	1.87	0.2
ru	1.98	4.45	0.68	2.06	4.2	0.71	2.77	5.55	0.64
sw	0.018	0.22	0.13	0.020	0.14	0.14	0.06	0.73	0.094
t	0.06	0.75	0.14	0.09	0.72	0.14	0.12	1.13	0.12
tn	0.104	0.66	0.28	0.107	0.60	0.28	0.75	2.72	0.28

Table 5.3: Objective values for each input in which optimisation occurred, for $w_t = 2.6$.

input	min $\sum^2$			min $\sum$			min max		
	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max
h	0.5	1	0.5	0.5	1	0.5	2.1325	4.5147	0.5
sw	0.4565	1.1090	0.6456	0.5	0.7071	0.7071	1.4722	3.6618	0.4714
t	16.6338	12.6188	1.9411	16.6338	12.6188	1.9411	17.7810	13.6899	1.9411
tn	2.6068	3.3086	1.4142	2.6875	3.0178	1.4142	7.8583	7.7279	1.4142

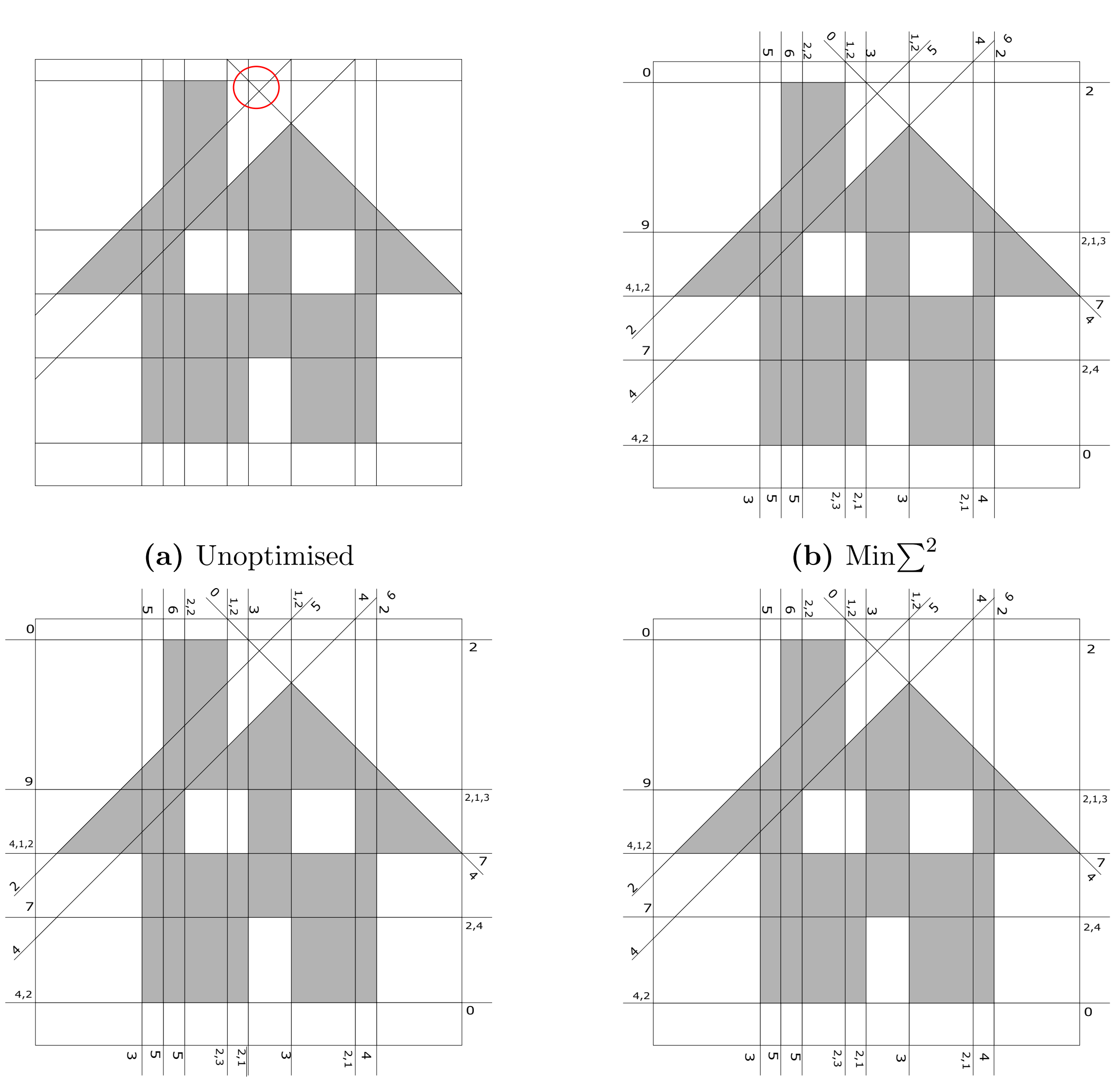
Table 5.4: Objective values for each input in which optimisation occurred, for $w_t = 3$.

input	min $\sum^2$			min $\sum$			min max		
	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max	min $\sum^2$	min $\sum$	min max
b	0.49	1.34	0.47	0.65	1.12	0.66	0.96	2.58	0.38
sh	0.11	0.56	0.23	0.22	0.46	0.46	0.26	1.35	0.19
sht	0.57	1.62	0.44	0.86	1.31	0.66	0.89	2.47	0.38
sw	4.32	4.26	1.91	4.93	3.05	2.12	12.3	11.0	1.41

Table 5.5: Objective values for each input in which optimisation occurred, for $w_t = 4$.

For each objective function the best score is obtained when optimising according to the same function; in other words, sub-column $\min \sum^2$ shows its best results in column $\min \sum^2$ and so on. It may be the case that the same score is also obtained in another sub-column, nevertheless, we never encounter the case in which a score, when optimised using method A, is worse than the score of method A computed using the results found when optimising with method B or C. Whenever $\min \sum$ found multiple solutions, only the one with best $\min \sum$ score is shown.

This section explores the output solved sloped nonograms generated for 'house', 'swan' and 'tan_cat' when using $w_t = 2.6$. For the three inputs chosen: 'house', 'swan' and 'tan_cat', we display four output solved puzzles. The first is a solved unoptimised puzzle, equivalent to the user's input. The small faces are circled in red. The second is a solved puzzle optimised using $\min \sum^2$. The third is the best, out of the solutions found, solved puzzle optimised using $\min \sum$. The last is a solved puzzle optimised using $\min \max$.



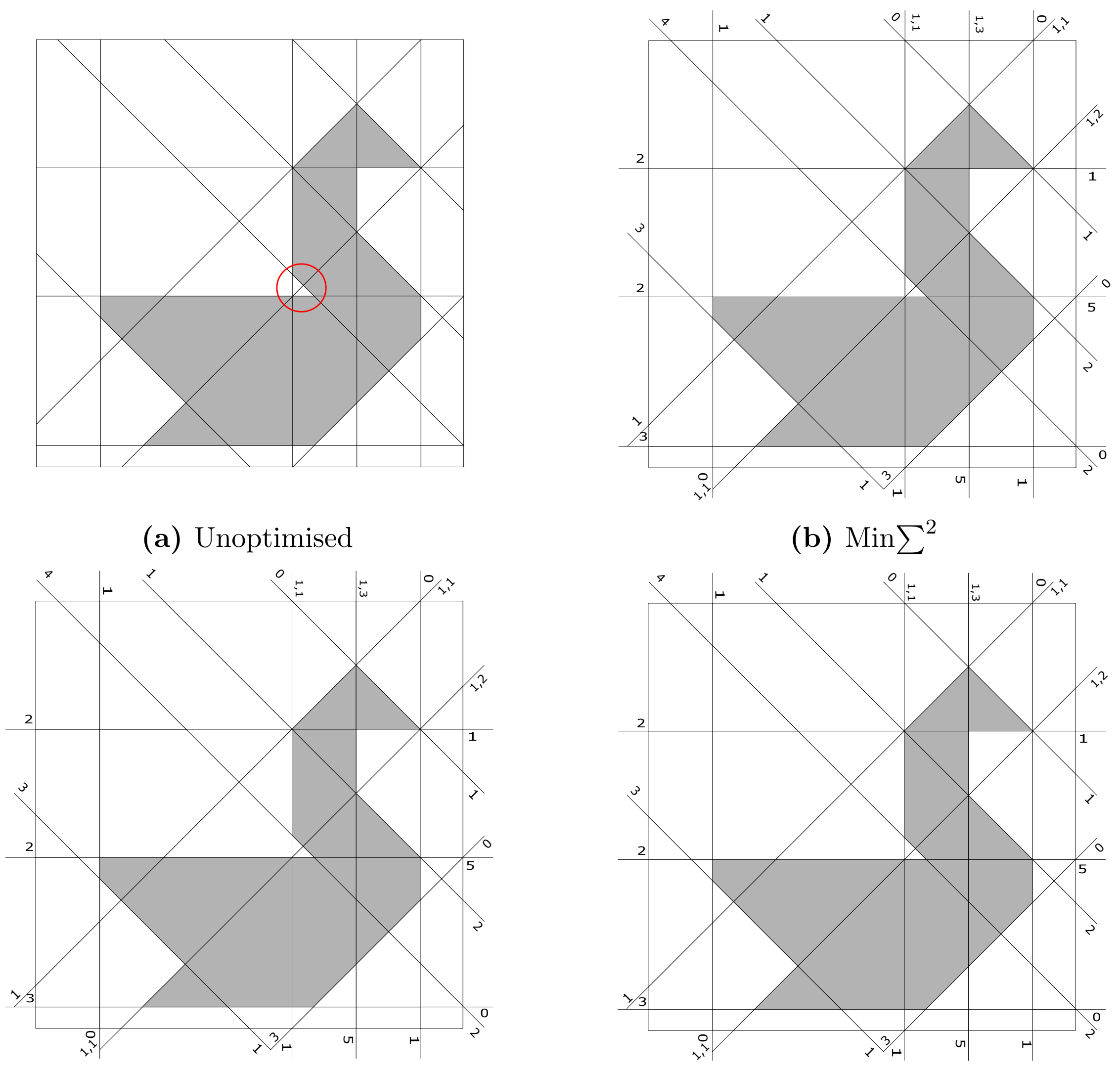
(a) Unoptimised

(b) $\text{Min}\sum^2$

(c) $\text{Min}\sum$

(d) Min max

Figure 5.4: Input 'house', unoptimised vs $\min \sum^2$ vs $\min \sum$ vs $\min \max$, $w_t = 2.6$.

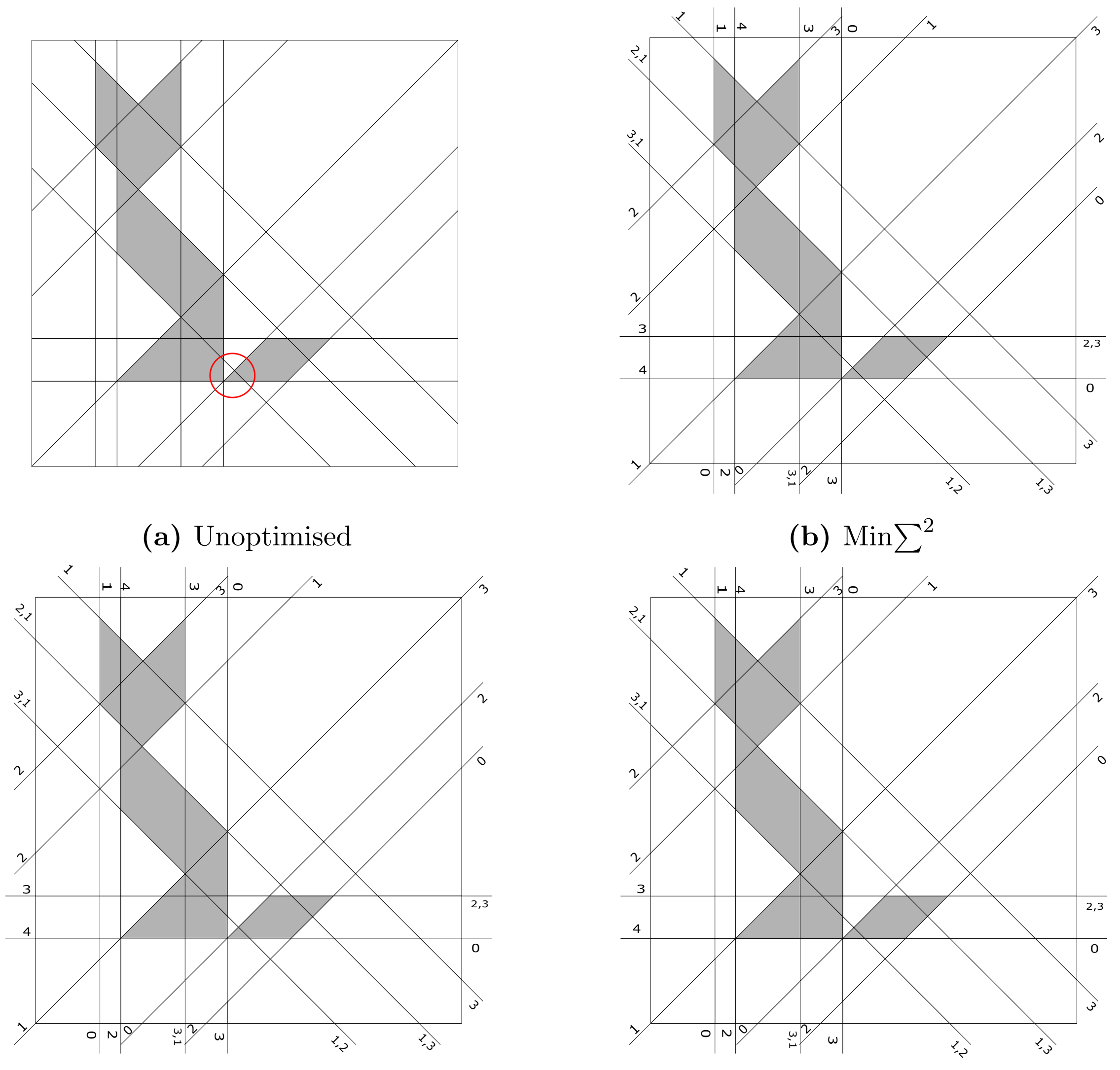


(a) Unoptimised **(b)** Min$\sum^2$

(c) Min$\sum$ **(d)** Min max

Figure 5.5: Input ‘swan’, unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 2.6$.

For all three inputs, using $w_t = 2.6$ yielded some results. Nevertheless, looking at Figures 5.4, 5.5 and 5.6 we can hardly notice a difference between the unoptimised and optimised versions. It can be noticed that in the three optimised outputs, the width of the small faces have been increased. From Table 5.3 we can see that for ‘house’, min $\sum^2$ and min $\sum$ both obtained the same scores, namely (0.02, 0.2, 0.1) whereas min max scored (0.09, 0.9, 0.1). The solution of min $\sum$ may not be the best possible since we only sample 5000 different linear programmes as opposed to the $2^{16} = 65536$ possible. Looking closely at Figures 5.4b and 5.4c we can see that the solutions are identical. Figure 5.4d however shows a different solution. We see that when using min max, multiple extended edges shifted to satisfy the face size and topology constraints. This can be explained by the nature of the min max objective function. As a matter of fact, shifting arbitrarily many lines by amounts smaller or equal to 0.1, in the case of ‘house’, does not worsen the overall score of min max, which is what occurred in Figure 5.4d. Evaluating which of the three objective functions yields the best visual output is difficult since all three outputs (two different solutions), closely resemble the unoptimised version (Figure 5.4a), and



(a) Unoptimised

(b) Min$\sum^2$

(c) Min$\sum$

(d) Min max

Figure 5.6: Input 'tan_cat', unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 2.6$.

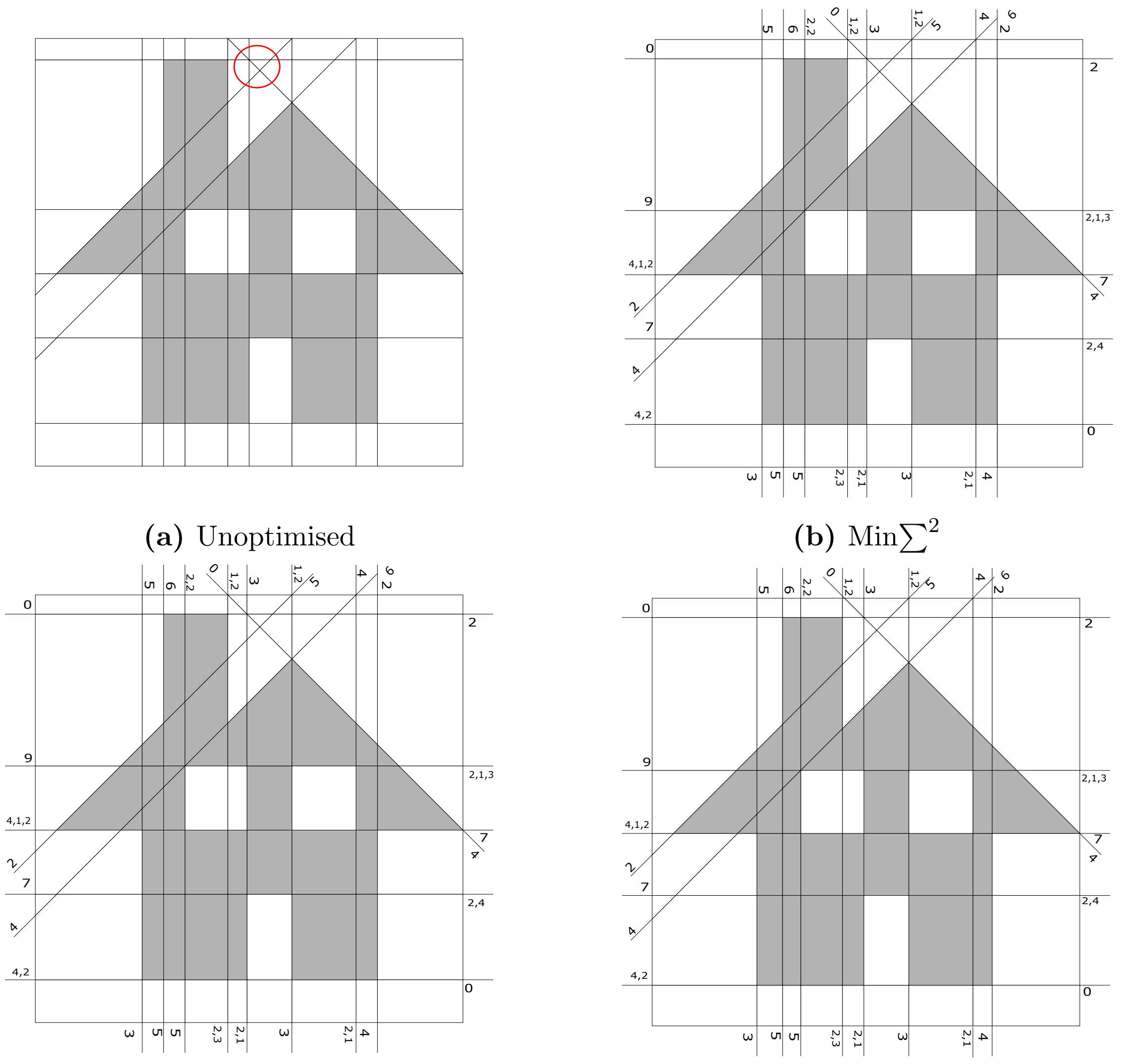
each other. Therefore, visually speaking, it seems that all three objective functions perform equally well for input 'house' and $w_t = 2.6$.

For input 'swan', a different solution was found by each objective function. From Table 5.3, we notice that the solution found using min $\sum$ only shifts a single extended edge by 0.14. We can infer this since the minimum maximal shift has magnitude 0.14 and the sum of all shifts is also 0.14. As $n = 14$ for 'swan', we used 5000 different configurations of the problem. Looking at the unoptimised solved puzzle of 'swan', we notice that the width of the two small faces can be increased by positively shifting the negatively diagonal edge bounding them. This extended edge only bounds fairly larger faces on its right side, hence a small positive shift will not create new small faces. Out of the 5000 random configurations, one was created in which the bounds on the δs allowed for such a shift, thus solving the problem. Had we used a smaller value for m, this configuration may not have been created and a worse solution may have been found.

The solution found using $\min\max$ shifted many extended edges by small amounts. As a matter of fact, the minimum maximal shift obtained using $\min\max$ is 0.094, but the score for sub-columns $\min\sum$ is 0.73. This means that the sum of all shifts, where shifts have a magnitude of 0.094 or less, equals 0.73, so many extended edges were shifted. From a visual point of view, it is once again hard to notice a difference between the three solutions as they all closely resemble the unoptimised solved puzzle. Therefore, it seems that all three objective functions visually performed equally well for input 'swan' and $w_t = 2.6$.

For input 'tan_cat', three different solutions were found. Similarly to the other two inputs, the solution found by $\min\max$ shifted many extended edges, that is inferred by the fact that the measurement of $\min\sum$ in the $\min\max$ column of Table 5.3 shows a score of 2.73, as opposed to the much lower score obtained for the $\min\sum$ scores in the $\min\sum^2$ and $\min\sum$ columns. However, it is difficult to visually notice a great difference between the three solutions. Once again, since there is no clear visible deformation in each solution, it seems that the three objective functions visually performed equally well for input 'tan_cat' and $w_t = 2.6$.

Let us now investigate output puzzles when using $w_t = 3$. Once again, each Figure shows a solved unoptimised puzzle, equivalent to the user's input, where the small faces are circled in red, a solved puzzle optimised using $\min \sum^2$, the best solved puzzle, out of the solutions found, optimised using $\min \sum$ and a solved puzzle optimised using $\min \max$.

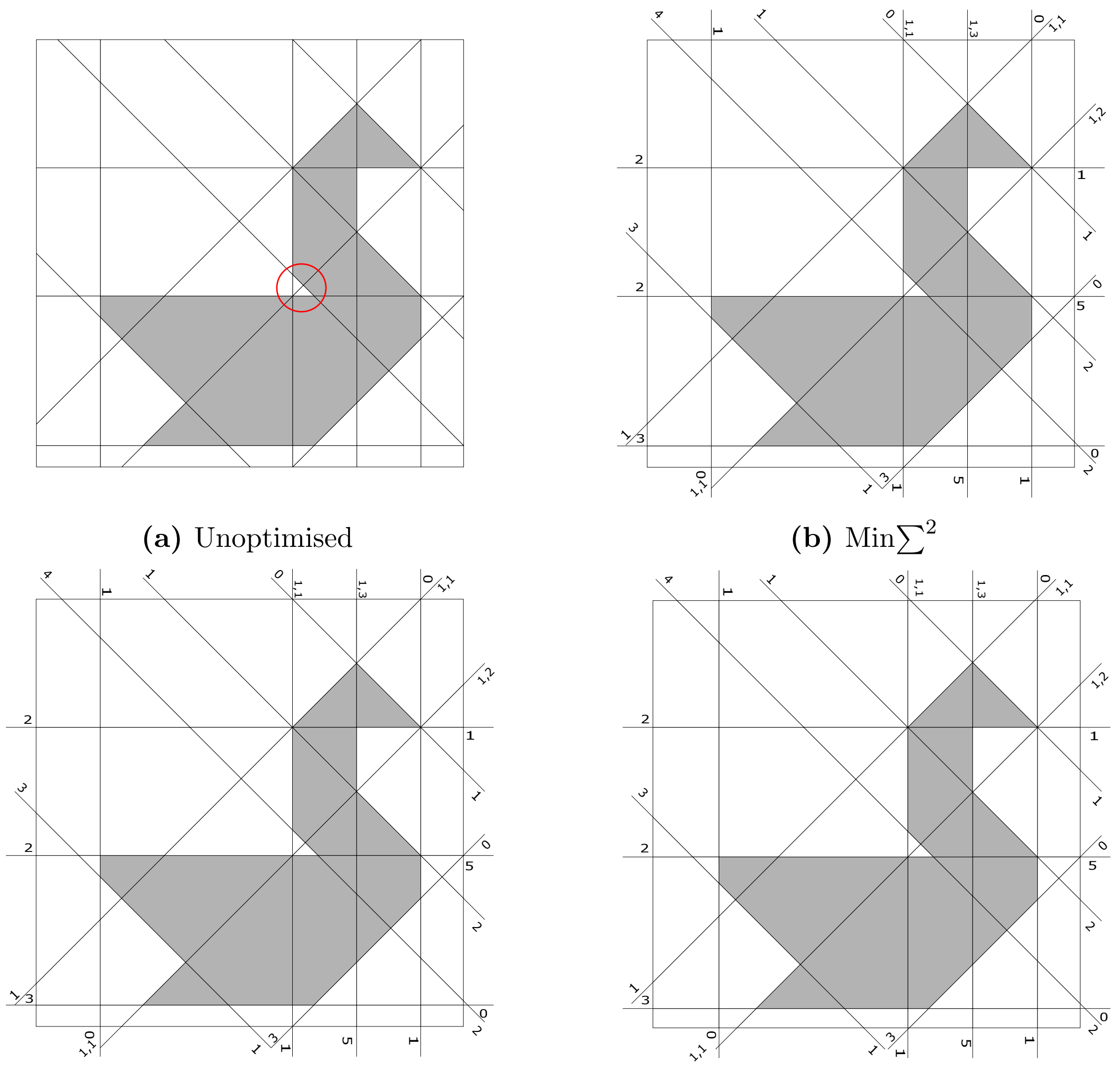


(a) Unoptimised **(b)** $\text{Min}\sum^2$

(c) $\text{Min}\sum$ **(d)** Min max

Figure 5.7: Input 'house', unoptimised vs $\min \sum^2$ vs $\min \sum$ vs $\min \max$, $w_t = 3$.

Regarding the results for 'house', looking closely at Figures 5.7b and 5.7c, we see that they are identical. As a matter of fact the solution found using $\min \sum^2$ and $\min \sum$ involves shifting the same extended edges as for $w_t = 2.6$ except by a greater magnitude. Min max, however, found a different solution, in which many extended edges were shifted by small magnitudes. Overall, it is difficult to say which of the two different solutions displayed looks better, since they look so similar. It is clear however that the two solutions very closely resemble the unoptimised. From our solutions for 'house' with $w_t = 3$, the three objective functions seem to visually perform equally well.



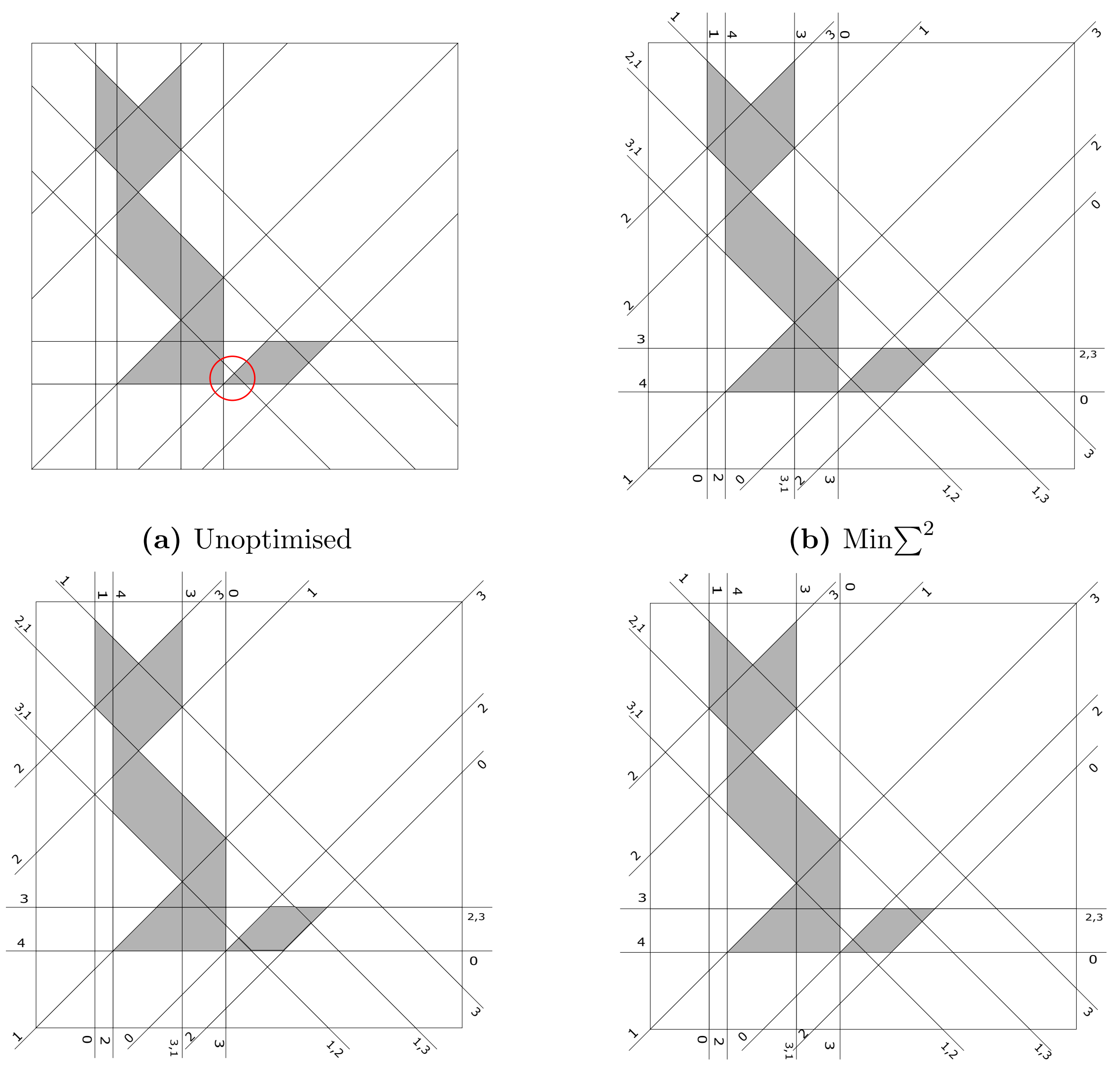
(a) Unoptimised **(b)** Min$\sum^2$

(c) Min$\sum$ **(d)** Min max

Figure 5.8: Input 'swan', unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 3$.

These results for 'swan' show that when using min $\sum^2$ and min max, several extended edges were shifted, whereas when using min $\sum$ only a single extended edge was shifted. The solution found using min $\sum$ is the same as for $w_t = 2.6$, except that the extended edge which moved was shifted by a larger amount, which make sense as the width threshold increased from $w_t = 2.6$ to $w_t = 3$. Even though different extended edges shifted in all three output puzzles, they all look very similar to the unoptimised. From those outputs puzzles, there is no clear better objective function for 'swan' with $w_t = 3$.

The output puzzles for 'tan_cat', resulting from optimisation using min $\sum^2$ and min $\sum$ are identical, but differ from the one resulting from min max. Once again, the difference between the two solutions and the unoptimised puzzle is hardly noticeable, so we cannot state that one objective function yields better visual results than the others for 'tan_cat' and $w_t = 3$.



(a) Unoptimised

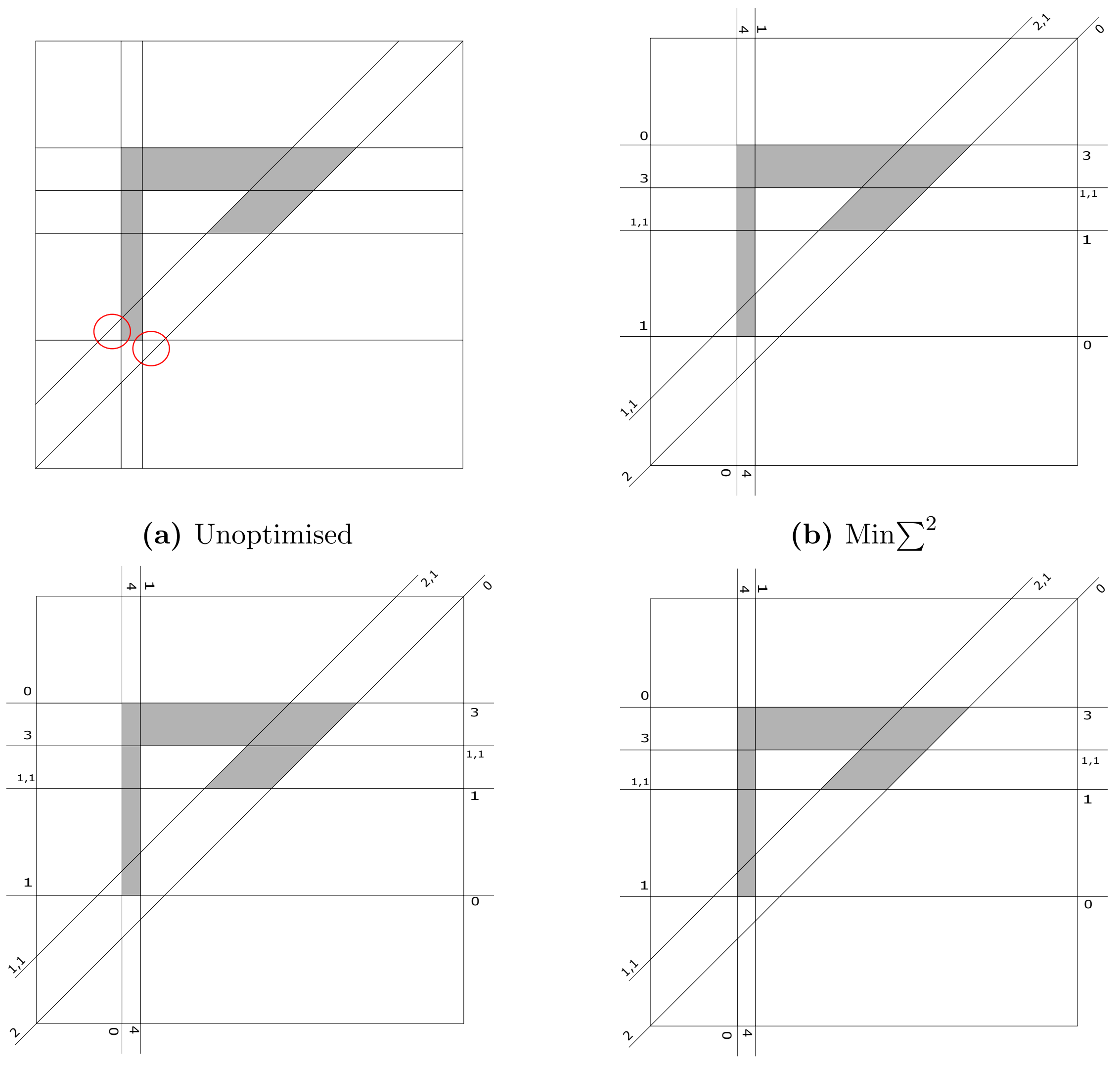
(b) Min$\sum^2$

(c) Min$\sum$

(d) Min max

Figure 5.9: Input 'tan_cat', unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 3$.

Let us now investigate output puzzles when using $w_t = 4$. We are using two different inputs: 'back_seven' and 'simple_triangle_hole_tangent', since optimisation with $w_t = 4$ in both 'house' and 'tan_cat' is infeasible. Once again, each Figure shows a solved unoptimised puzzle, equivalent to the user's input, where the small faces are circled in red, a solved puzzle optimised using $\min \sum^2$, the best solved puzzle, out of the solutions found, optimised using $\min \sum$ and a solved puzzle optimised using $\min \max$.



(a) Unoptimised

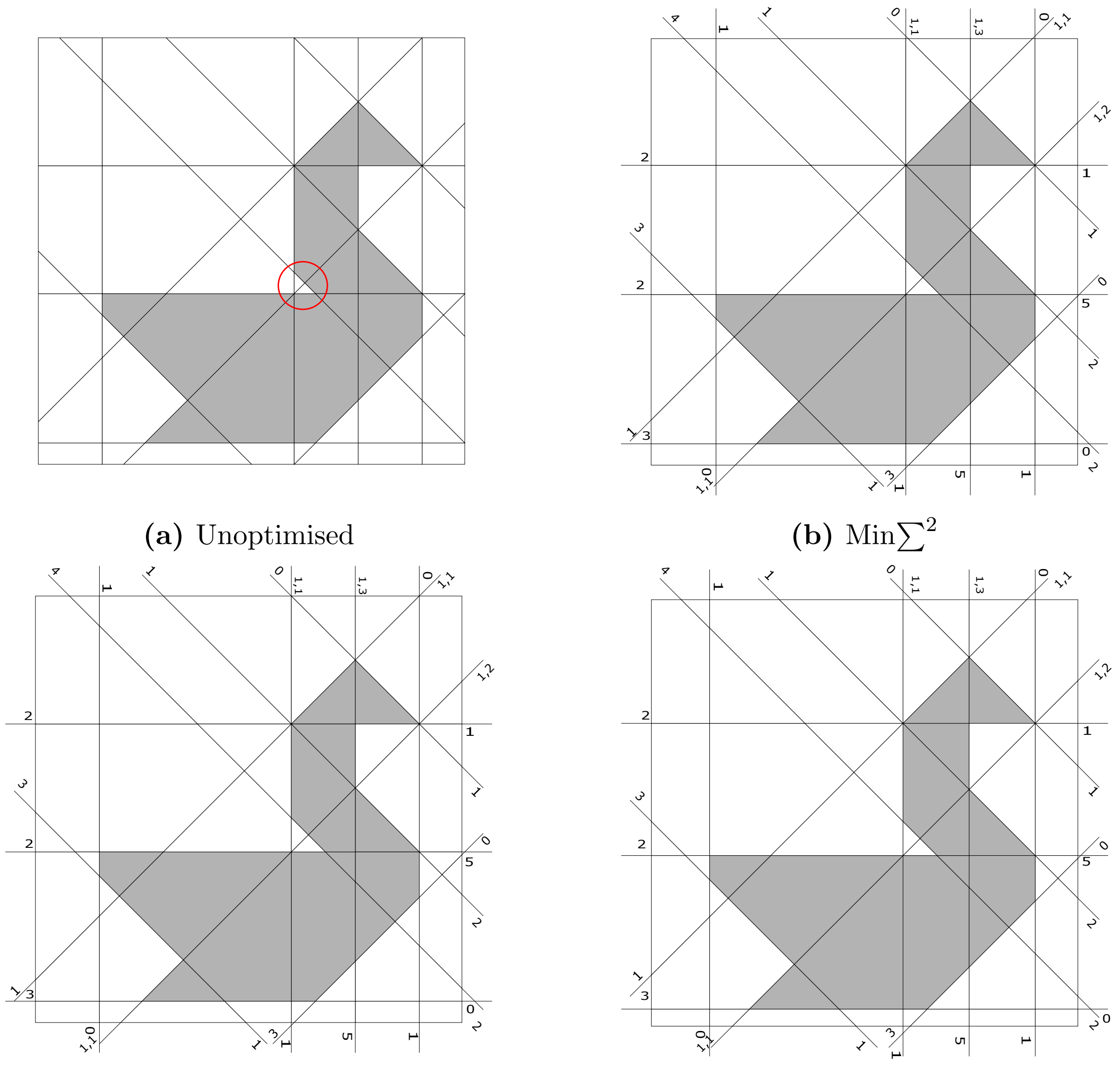
(b) Min$\sum^2$

(c) Min$\sum$

(d) Min max

Figure 5.10: Input 'back_seven', unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 4$.

With $w_t = 4$, we start to notice a visual difference between the unoptimised version and the solutions found using $\min \sum^2$, $\min \sum$ and $\min \max$. Regarding 'back_seven', we clearly see that in the optimised versions, the vertical part of the inverted seven is thinner. From the input, Figure 5.10a, we see that the two small faces are located on both sides of the vertical part of the inverted seven. Therefore, one way to increase the width of those faces is to thin-out the vertical part of the inverted seven, which is what occurs in all three solutions. The solution found using min max once again shifts many more extended edges than the other two solutions. Although a difference is noticeable between the input and the solutions found, the solutions look very similar to each other, so we cannot state that one objective function performs better than the others for input 'back_seven' and $w_t = 4$.



(a) Unoptimised

(b) Min$\sum^2$

(c) Min$\sum$

(d) Min max

Figure 5.11: Input 'swan', unoptimised vs min $\sum^2$ vs min $\sum$ vs min max, $w_t = 4$.

For input 'swan', we see a clear difference between the unoptimised and optimised puzzles. The small faces, at the bottom of the neck are much larger in the optimised versions. Although the three solution all look different from the input, we cannot say which objective function produces the best output puzzle for input 'swan' and $w_t = 4$, as all solutions still look like swans and do not heavily differ from each other.

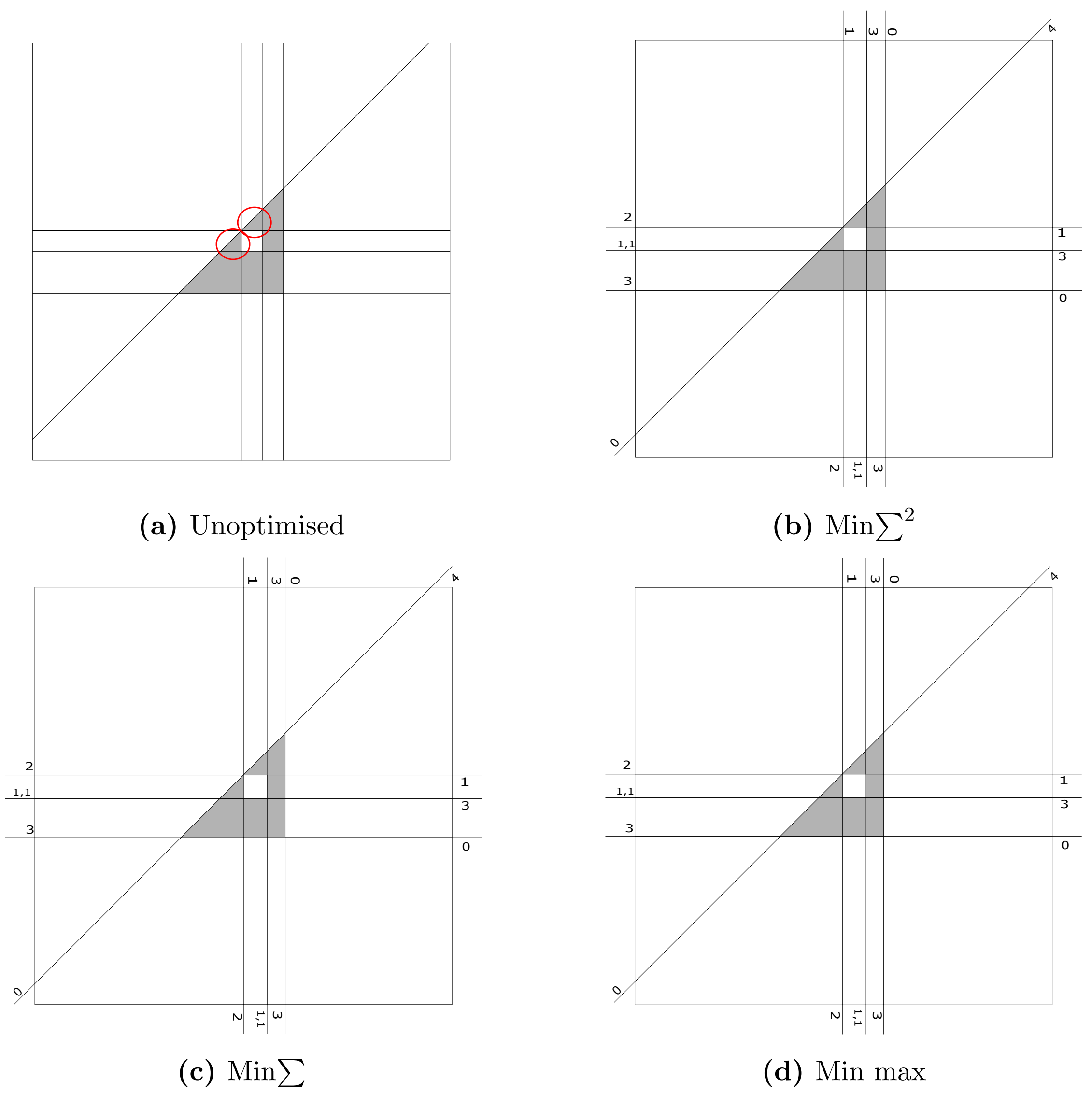


Figure 5.12: Input 'simple_triangle_hole_tangent', $w_t = 4$.

For input 'simple_triangle_hole_tangent', we once again notice a visual difference between the unoptimised and the optimised puzzles. All three solutions look similar to each other, so we cannot state that one objective function is better than the others for 'simple_triangle_hole_tangent' with $w_t = 4$.

Overall, for width threshold $w_t = 2.6$ and $w_t = 3$, differences between unoptimised puzzles and their optimised solved output puzzles, using either objective function, were hardly noticeable. For $w_t = 4$, we can see a difference between unoptimised puzzle and the optimised output solved puzzles, however, there is not clear difference between the solutions themselves. It is therefore not possible, given the current data, to state that one objective function yields better output sloped nonograms than the others. From our results, we never experienced a great deformation in which the input is not recognisable anymore. It seems, given the current data, that the optimised puzzles still look very similar to their input, no matter which objective function is used. More work should be done, using more inputs, to clearly identify whether the use of one particular objective function yields better sloped nonograms.

5.2.2 Speed

We compared the speed of all three optimisation techniques: $\min \sum^2$, $\min \sum$ and $\min\max$. The prediction is that $\min \sum^2$ and $\min\max$ will be approximately equally fast, since they both only require solving a single quadratic / linear programme. We predict that $\min \sum$ will perform 2^n times slower than $\min \sum^2$ and $\min\max$ for inputs with $n \leq 12$, and m times slower for inputs with $n > 12$. The prediction originates from the number of linear programmes $\min \sum$ is required to solve.

The data is gathered from running the optimisation on all inputs for $w_t = 3$. Optimisation was performed six times, each time varying m, ($m = 10$, 50, 100, 500, 1000, 5000). For all inputs we calculated the average time required to optimise a single quadratic / linear programme for each optimisation routine. For $\min \sum^2$ and $\min\max$, that was just the time taken to solve the whole problem, whereas for $\min \sum$, that was $\frac{t_{\min \sum}}{2^n}$ if $n \leq 12$, where $t_{\min \sum}$ is the time taken by $\min \sum$ to solve 2^n linear programmes, and $\frac{t_{\min \sum m}}{m}$ if $n > 12$, where $t_{\min \sum m}$ is the overall time taken by $\min \sum$ for solving m linear programmes. The results are shown in Table 5.6, as well as in Figure 5.13. In Figure 5.13 red dots represent individual times for $\min \sum^2$, blue dots for $\min \sum$ and green dots for $\min\max$.

input	n	$t_{average}$ $\min \sum^2$	$t_{average}$ $\min \sum$	$t_{average}$ $\min\max$
b	8	0.0158	0.0127	0.0158
db	33	0.2782	0.2793	0.3017
f	35	0.2863	0.2850	0.3123
h	16	0.0312	0.0349	0.0367
l	21	0.0957	0.0953	0.1030
mh	18	0.0678	0.0713	0.0810
r	15	0.0415	0.0422	0.0467
ra	16	0.0547	0.0486	0.0547
rah	20	0.0782	0.0801	0.0856
re	17	0.0470	0.0413	0.0442
ru	18	0.0833	0.0713	0.0780
s	3	0.0155	0.0033	0.0160
sh	7	0.0157	0.0094	0.0155
sht	7	0.0150	0.0086	0.0155
sw	14	0.0317	0.0332	0.0440
t	14	0.0337	0.0393	0.0392
tn	13	0.0363	0.0370	0.0390
tl	27	0.1403	0.1434	0.1563
ts	2	0.0155	0.0040	0.0160

Table 5.6: Average solving time for a single programme using $\min \sum^2$, $\min \sum$ and $\min\max$ for $w_t = 3$.

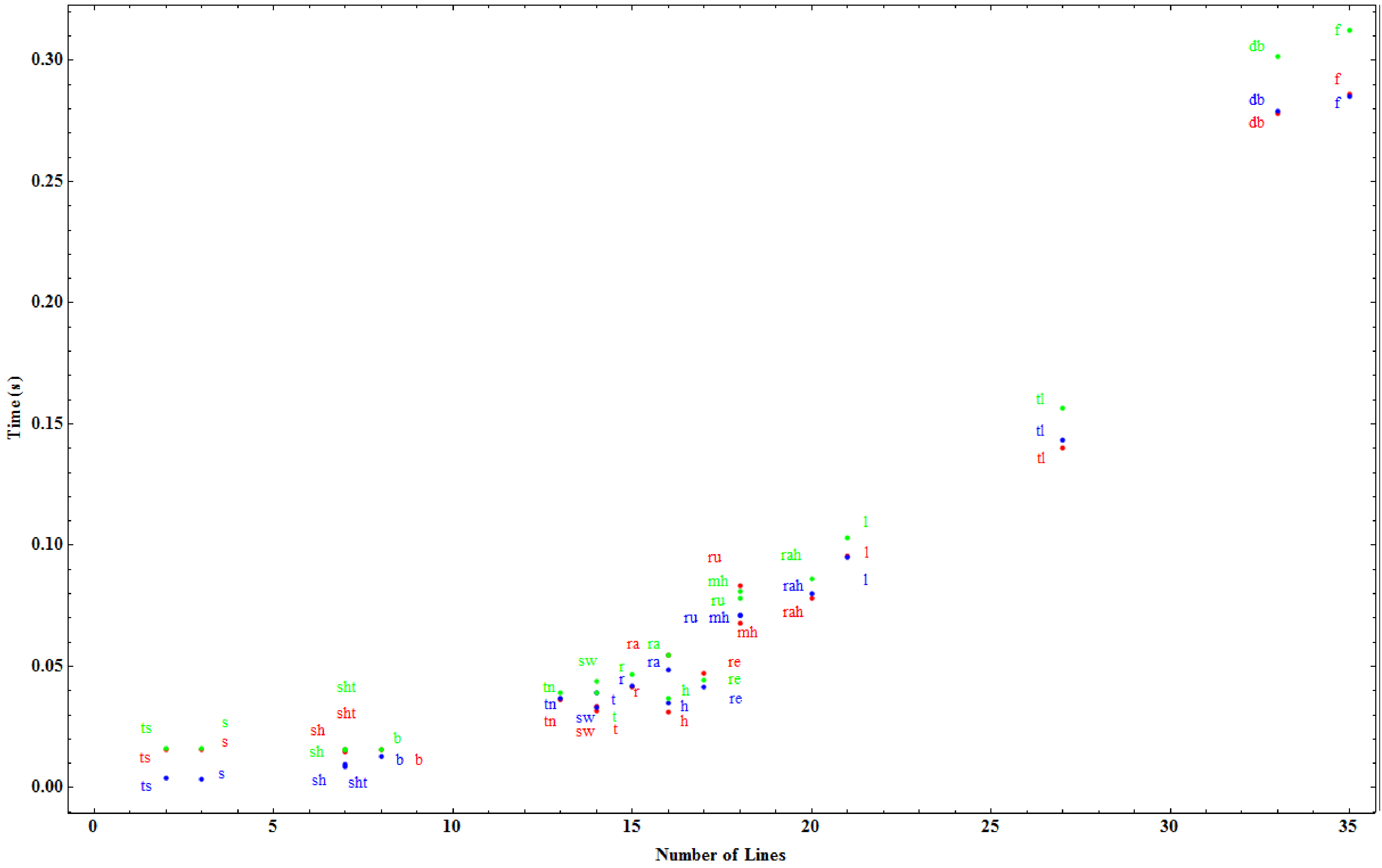


Figure 5.13: Average optimisation speed for solving a single programme for each input for $w_t = 3$. Red is min $\sum^2$, blue is min $\sum$ and green is min max.

Figure 5.13 shows that as n increases, the time required for optimising a single programme, no matter the objective function, also increases. This is what was expected, as with larger n, more constraints are generated, increasing the time needed to find a solution. The actual relationship between optimisation time of a single programme and n could be determined because the inputs used do not spread the n range evenly enough. As a matter of fact, many inputs contained similar number of lines. Fitting a function to the data would have been heavily affected by the few inputs with much smaller or larger n, such as 'two_squares' and 'fish' respectively. Moreover, we did not have much information regarding the theoretical running time of the linear programming solver as a function of the number of constraints.

The data from Table 5.6 shows that for most inputs, the time taken to solve a single programme using $\min \sum^2$, $\min \sum$ and $\min \max$ is approximately the same. A large difference, between $\min \sum$ and the rest, is noticed for 'two_squares' and 'simple_triangle', which are the inputs with fewest lines, $n = 2$ and $n = 3$ respectively. This is also noticed, to a lesser extent, for inputs with $n < 10$. Overall, we believe that a single programme optimised by $\min \sum$ should be solved faster than when using $\min \sum^2$ or $\min \max$, because the search space for values of δ is greatly reduced by forcing each δ_i to be either positive or negative. As n increases, setting up the $\min \sum$ problem becomes longer, since we must assign bounds to each δ, so the pruning of the search space may not make as much of a difference in terms of optimisation speed compared to $\min \sum^2$ and $\min \max$.

It also seems that as n grows, $\min \max$ becomes slower than the other two techniques, for solving a single programme. This is observed in 'lion', 'two_lads', 'dragon_bird' and 'fish'. A reason may be that by nature, $\min \max$ requires two extra constraints per decision variable compared to $\min \sum^2$ and one extra constraint compared to a single linear programme from $\min \sum$. The time spent satisfying those extra constraints may only become apparent when n grows.

For the rest of our inputs, $t_{average} \min \sum^2 \approx t_{average} \min \sum \approx t_{average} \min \max$, so we can conclude that $\min \sum^2$ is approximately as fast as $\min \max$, and that $\min \sum$ is slower than the other two by a factor of 2^n when $n \leq 12$ or by a factor of m when $n > 12$, which is what was predicted. More work on larger inputs should be done to verify whether $\min \max$ does slow down compared to $\min \sum^2$, and if so, how fast. If that were the case, $\min \sum^2$ would be considered the fastest technique.

5.2.3 Maximum minimal width

Let us now look at another experiment, aimed at finding the maximum minimal possible width *max minwidth* such that all faces in the subdivision after optimisation have a width greater or equal to *max minwidth*. This measurement is interesting because it gives us insight on how much the width of each face can be optimised as a function of n. The prediction for this measurement is that as n increases, the maximum minimal possible width for each face decreases. The reason for such a prediction is fairly straightforward: when more lines are added to the subdivision, more constraints are generated, making optimisation for a large width more difficult and sometimes impossible. Results on the maximum minimal width for each input are shown in Table 5.7.

input	n	*max minwidth*
b	8	7.125
db	33	2.5
f	35	2.5
h	16	3.0625
l	21	2.5625
mh	18	2.75
r	15	2.75
ra	16	2.75
rah	20	2.5
re	17	3.75
ru	18	2.625
s	3	23.5625
sh	7	10.25
sht	7	11.25
sw	14	4.25
t	14	3.015625
tn	13	3.125
tl	27	2.5
ts	2	50

Table 5.7: Maximum minimal width *max minwidth* for each input, such that all faces have width larger or equal to *max minwidth*.

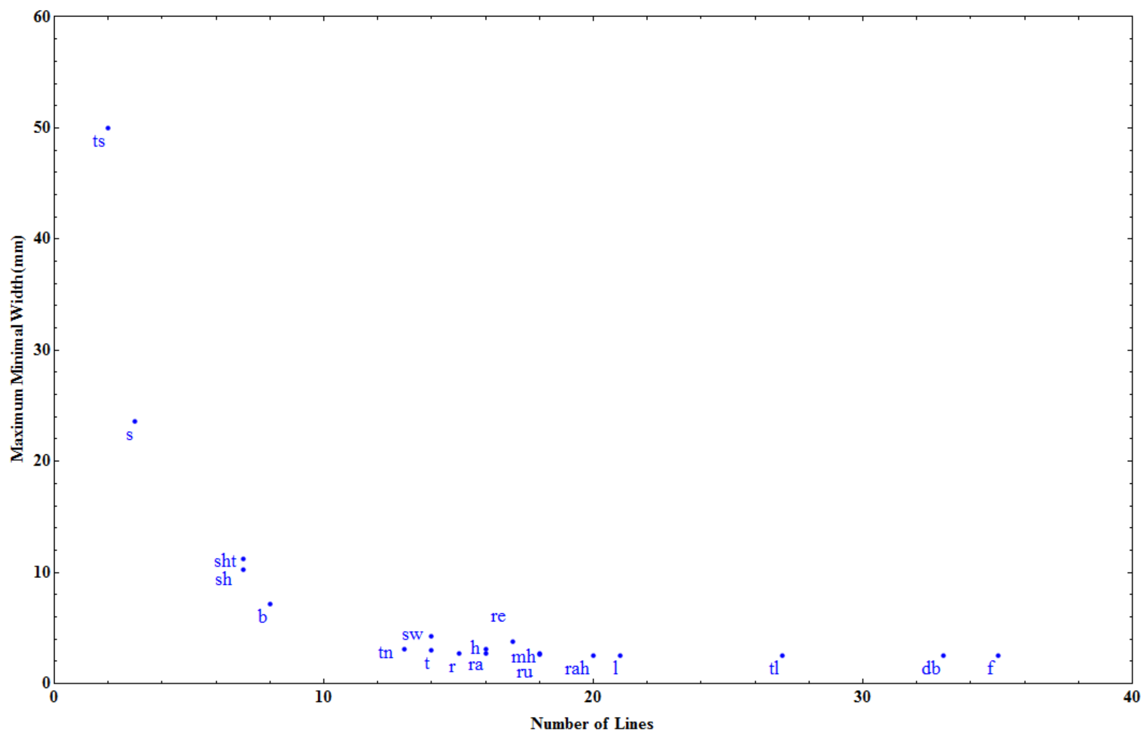


Figure 5.14: Relationship between maximum minimal width *max minwidth* and number of lines n.

Figure 5.14 shows a rapidly decreasing relationship between the maximum minimal width and n for small n and a slowly decreasing relationship for large n. Using the pigeon-hole principle, we can explain that the relationship should be an inverse function of order $O(\frac{1}{n})$. Let us have a subdivision with n lines arranged as $\frac{n}{2}$ horizontal and $\frac{n}{2}$ vertical lines equally spaced creating a grid with $\frac{n}{2}+1$ square cells per row and column. The entire subdivision therefore has $(\frac{n}{2}+1)^2$ square cells, which is $O(n^2)$. We use that arrangement such that the maximum size of the smallest cell can be identified. Label the size of a cell in the grid arrangement s_{cell}. Using any other arrangement would create larger and smaller faces than the cells, therefore the smaller face would have size $s_{face} < s_{cell}$, so no upper bound on the size of the smallest face could be identified. However, we can derive an upper bound on the size of cells in the grid arrangement. If the entire subdivision has area 1, then the upper bound on the area of cells is $\frac{1}{(\frac{n}{2}+1)^2}$ which is $O(\frac{1}{n^2})$. Since all cells are squares and their areas are $O(\frac{1}{n^2})$, then their width must be $O(\frac{1}{n})$. Mathematically speaking, as $n \to \infty$, then *max minwidth* $\to 0$. The can be explained by the fact that with more lines, faces become smaller, so optimisation for large width is no longer possible. Therefore as the number of lines increases in the subdivision, the maximum minimal possible width *max minwidth* for all faces decreases.

Figure 5.15 shows a best fit inverse function of the form $f(x) = \frac{b}{x}$, namely:

$$f(x) = \frac{84.4303}{x} \tag{5.1}$$

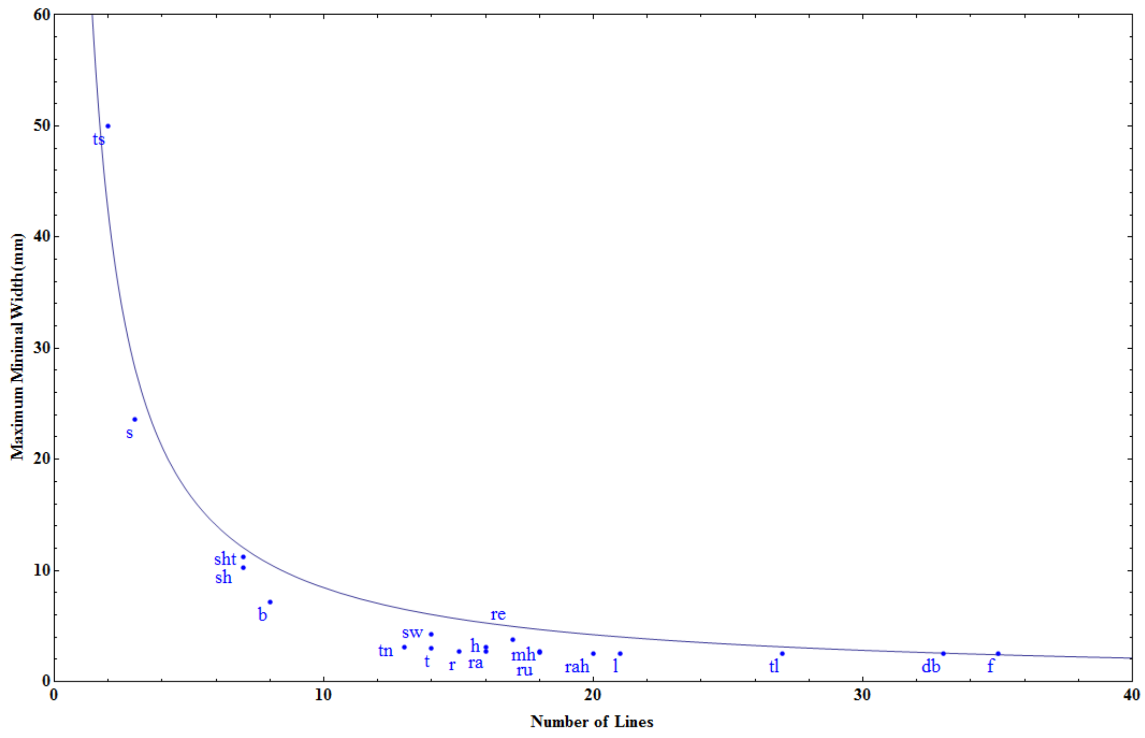


Figure 5.15: Relationship between maximum minimal width *max minwidth* and number of lines n, with fitting inverse function.

5.2.4 Difficulty

Table 5.8 shows the difficulty as computed by the solver for each input. Notice that for 'two_squares', the solver got stuck and returned -1. That is what was expected as 'two_squares' does not belong to the class of *simple* nonograms.

It is obvious that as n increases the difficulty will increase too, but predicting by how much is complicated. With more lines, more faces are created, thus creating more possibilities for colouring the faces. Figure 5.16 shows the difficulty of puzzles as a function of n. As expected, from the data, we see that as n increases, so does

input	n	*difficulty*
b	8	11
db	33	79
f	35	89
h	16	32
l	21	42
mh	18	34
r	15	23
ra	16	25
rah	20	28
re	17	26
ru	18	32
s	3	4
sh	7	12
sht	7	11
sw	14	22
t	14	23
tn	13	22
tl	27	56
ts	2	-1

Table 5.8: Difficulty of each input.

the difficulty. Fitting a function to the data cannot be done properly since we have neither enough data, nor a theory from which the basic shape of the fitting function could be determined. It seems that n plays a large role in determining the difficulty, however as much variation in difficulty, for puzzles with $13 \leq n \leq 20$, occurs, it seems to indicate that n is not the only factor which affects the difficulty. We believe that the geometry of the input also plays a large part in determining the difficulty of a puzzle. More work should be done to identify all parameters which affect the difficulty and determine how they affect difficulty.

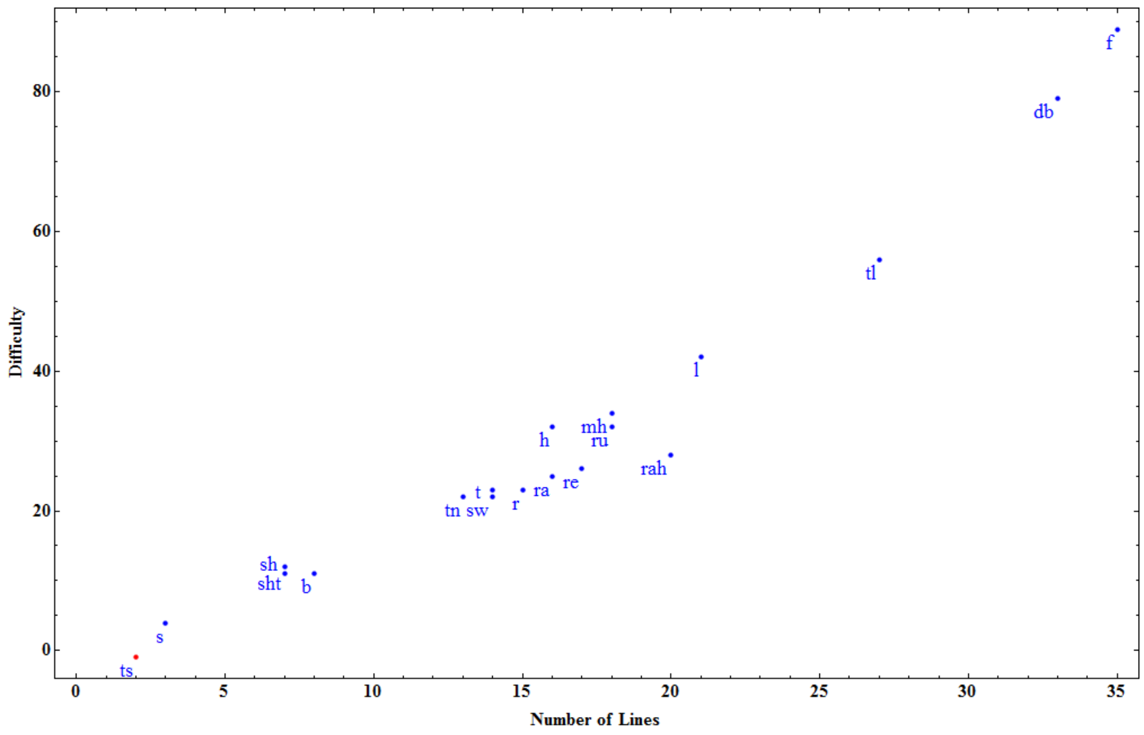


Figure 5.16: Relationship between difficulty and the number of lines n.

Chapter 6

Conclusion

The main goal of this project was to create a method for generating good sloped nonograms from input drawings. Those nonogram were forced to satisfy multiple constraints: the width of their faces had to be larger or equal than some threshold w_t, they had to be topologically the same as their input, and they had to belong to the *simple* nonogram class. Moreover, the solved sloped nonograms had to resemble their input. Some of those constraints were not always satisfied, due to the nature of the inputs. All inputs could be converted into sloped nonograms in which each face has a width larger or equal to $w_t = 2$. That was the original size we considered small. Regarding larger values of w_t, for many inputs, optimisation was infeasible. In addition, all inputs but 'two_squares' belong to the *simple* type of nonogram, so they are uniquely solvable and can be solved iteratively. 'Two_squares' has two solutions, so it is not *simple*.

Looking back at the project, we can generate sloped nonograms from input drawings if the drawings are valid. The face size constraint is taken care of through the optimisation procedure. For some inputs, optimisation is infeasible. Optimisation enforces minimal deformation of the inputs, so the solved output nonograms look very similar to their input. Optimisation also enforces the topology of the puzzle to stay intact. The simpleness of the output nonograms is checked by the nonogram solver. Therefore the main goal was achieved. Regarding our other goals: investigating which objective function performs best, investigating the relationship between the maximum minimal width and the number of lines n, and investigating the relationship between the difficulty and the number of lines n, the results were not as clear as we wanted.

Finding which objective function performed best was based on speed and visual aspects of the output solved sloped nonograms. Both $\min \sum^2$ and $\min\max$ seem to perform at similar speeds for inputs with $n < 30$. We noticed that for larger inputs, $\min\max$ seems to be slower, however, more investigating needs to be done to confirm this observation. Regarding $\min \sum$, it performs slower by a factor of 2^n for inputs with $n \leq 12$ and by a factor of m for inputs with $n > 12$ since we sampled m random linear programmes. Therefore from the speed point of view $\min \sum$ definitely is the worse objective function. We cannot yet say which of $\min \sum^2$ and $\min\max$ is fastest at the moment.

With regard to visual quality of output solved sloped nonograms, we cannot state which objective function performs better, as with all inputs tested, solutions found by each objective function very closely resembled the input. We never experienced a great deformation in which the input was no longer recognisable. It seems that, with our tested inputs, the choice of objective function does not heavily influence the quality of output puzzles.

Investigating the relationship between the maximum minimal width obtained through optimisation against the number of lines n yielded very interesting results. The inverse relationship found of order $O(\frac{1}{n})$ tells us roughly how fast the maximum minimal width obtainable through optimisation shrinks as n increases.

Computing the difficulty measure was done using a nonogram solver. The main issue with the measure used is that it does not really take into account difficulty, but rather the number of steps required for solving. However, because all our puzzles were of the simple type, the main degree of difficulty is the number of steps required for solving. Investigating the relationship between the difficulty and n did not yield great results. There were variations in the data, which probably means that difficulty also depends on other factors than n. We believe that the geometry of the input influences the difficulty greatly, however we did not investigate this further.

Chapter 7

Future work

Regarding the work done in this thesis, much still remains to be done to either strengthen our findings, or actually further investigate questions we did not manage to answer. Determining which of $\min \sum^2$ and $\min \max$ is fastest needs to be looked at for large inputs. We noticed a time increase in solving time for $\min \max$ for large n, but we only have few inputs with such characteristics, which is not enough to reach any conclusions. Regarding visual output quality, more investigating should be done to determine whether one of the objective function yields better visual results than the others.

Exploring in greater details the parameters which affect the difficulty, as defined in the thesis, is needed. We saw that n does heavily influence the difficulty but the data seems to show that other parameters also play a role. Moreover, we realise that using the number lines n for determining relationships with optimisation speed, maximum minimal width and difficulty was not the best solution. Using the number of faces would have yielded more interesting results, because those are clearly connected to the number of constraints in the problem.

Regarding the field, a great amount of research is still possible. First of all, defining a proper difficulty measure is required, as the one we used monitors the number of steps required for solving the puzzle. For a proper difficulty measure to be defined, there first needs to be work done on creating difficult sloped nonograms. Research is being done on creating difficult regular nonograms, which could then be converted into sloped nonograms. Moreover, extensions of sloped nonograms such as coloured sloped nonogram could be investigated.

The initial idea was to create sloped nonograms with arbitrary slopes. Trying to implement arbitrary sloped nonograms should be interesting and quite challenging, since the geometry of the faces is not as simple as when only using 4 slopes. Along the same lines is the generation of nonograms based on curves as opposed to lines. The overall framework is fairly similar to arbitrary sloped nonograms, but the implementation differs.

Bibliography

[1] K. J. Batenburg, S. Henstra, W. A. Kosters, and W. J. Palenstijn. "Constructing Simple Nonograms of Varying Difficulty". In: *Pure Mathematics and Applications* (2009).

[2] K. J. Batenburg and W. A. Kosters. "A Reasoning Framework for Solving Nonograms". In: *Combinatorial Image Analysis*. 2008.

[3] K. J. Batenburg and W. A. Kosters. "On the difficulty of Nonograms". In: *ICGA Journal* (2012).

[4] K. J. Batenburg and W. A. Kosters. "Solving Nonograms by combining relaxations". In: *Pattern Recognition* (2009).

[5] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. Wiley-Interscience, 2004.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 2008.

[7] *Conceptis Puzzles*. [accessed: 2015]. URL: `http://www.conceptispuzzles.com/`.

[8] R. Fourer. *Optimization Models, III Beyond Linear Optimization*. 2005.

[9] H. Freeman and R. Shapira. "Determining the Minimum-area Encasing Rectangle for an Arbitrary Closed Curve". In: *Commun. ACM* (1975).

[10] M. Goldwasser. *Computer Science 493, Special Topics: Computational Geometry*. 2007. URL: `http://mathcs.slu.edu/~goldwasser/courses/slu/csci493/2007_Fall/assignments/hw02/`.

[11] T. van Kapel. "Connect The Closest Dot Puzzles". MA thesis. Universiteit Utrecht, Department of Information and Computing Sciences, 2014.

[12] G. Klappe. "Connect the dots puzzles with direction indicators". MA thesis. Universiteit Utrecht, Department of Information and Computing Sciences, 2014.

[13] B. A. McCarl and T. H. Spreen. *Applied Mathematical Programming Using Algebraic Systems*. Texas A&M University 2011, 1997.

[14] K. Mulka. *nonogram-solver*. 2012. URL: `https://github.com/mulka/nonogram-solver`.

[15] M. Schervish and A. Katipally. *Nonogram Solver*. 2013. URL: `http://www.andrew.cmu.edu/user/akashr/project_template/index.html`.

[16] E. W. Weisstein. *Line-Line Intersection. From MathWorld–A Wolfram Web Resource.* [accessed: 2014]. URL: `http://mathworld.wolfram.com/Line-LineIntersection.html`.

[17] C.-H. Yu, H.-L. Lee, and L.-H. Chen. “An efficient algorithm for solving nonograms”. In: *Applied Intelligence* (2011).

Chapter 8

Appendix

8.1 Inputs

- Back_seven (Figure 8.1a) is a fairly simple input, consisting of 8 lines. From its geometry, it is quite clear that the width of its faces can be increased by a considerable amount, making it an interesting example from the optimisation point of view more than from the sloped nonogram point of view.

- Dragon_bird (Figure 8.1b) is a very complex input. It contains 33 lines and two entities, which allows for more complicated total edge descriptions.

- Fish (Figure 8.1c) is the most complicated input, as far as number of lines goes. It contains 35 lines and three entities, one on the left, one in the middle and one on the right, each at somewhat different heights. A very complicated puzzle should be generated for this input.

- House (Figure 8.2a) is somewhat complex, consisting of 16 lines. Moreover it contains holes to depict the windows. However it seems like its optimisation possibilities are quite limited, as the geometry of the figure will not allow for face width optimisation.

- Lion (Figure 8.2b) is a somewhat complex input with 21 lines.

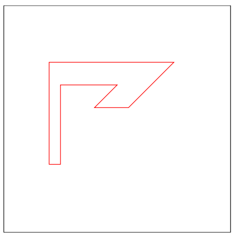
- Man_hole (Figure 8.2c) is a more interesting input as it is complex and represents an actual entity. It consists of 18 lines forming three main polygons: the body and the head, and the hole in the chest, whose sole purpose is to complicate the input.

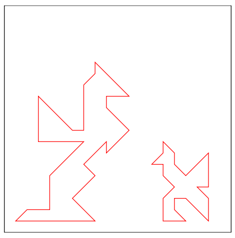
- Rabbit (Figure 8.3a) is interesting as its left bottom side, where the paws and legs are located is a fairly compact and dense region. The results of the optimisation for that particular area should to be interesting. Moreover, it is quite different from the previous inputs as most of its lines are diagonals.

- Random (Figure 8.3b) does not represent anything, nevertheless it was constructed under specific guidelines. As a matter of fact, we can see that in two parts of the input, the lines go into the centre of the drawing while not creating an entire hole. It should be interesting to see how such characteristics will behave under face width optimisation.

- Random_hole (Figure 8.3c) is the same as the previous input with the addition of a non-rectangular hole. It should be interesting to see how this input differs from the previous one, and the conclusions which can be drawn on the effects of holes on face width optimisation. Moreover, non-rectangular holes may behave differently from rectangular ones.
- Recurs_holes (Figure 8.4a) also does not represent anything, but its main characteristic is that it consists of a polygon within a hole. From a nonogram point of view, it is fairly interesting as it will undoubtedly generate more complex total edge descriptions, which can then be further exploited by the user for solving purposes.
- Running (Figure 8.4b) is quite a complex input. Face optimisation will probably not result in great changes as it seems that some faces will be constrained to remain small. The nonogram on its own should to be interesting and fairly complex to solve.
- Simple_triangle (Figure 8.4c) is very simple. The reason for such an input is to see how far face width optimisation can go. It seems like the triangle can grow until the faces bounded by the bounding box decrease in width up until they are just large enough to satisfy the face size constraint. Therefore this input should be a good test for face width optimisation, yet from a puzzle point of view, it is totally superfluous.
- Simple_triangle_hole (Figure 8.5a) is the same as the previous input except for the hole it contains. Once again, the input will be used to compare optimisation results of a subdivision, with and without holes.
- Simple_triangle_hole_tangent (Figure 8.5b) is also the same as 5.1l except that the hole is now tangent to the hypotenuse of the triangle. The difference is that now we have a quadruple intersection which must stay intact in the final subdivision. Comparing the face optimisation results of this input with that of simple_triangle_hole will give some insight on how the placement of a hole affects face optimisation results.
- Swan (Figure 8.5c) is interesting because its contour is fairly round. However, it is not as complicated as other inputs, so it should yield good optimisation and puzzle results.
- Tan_cat (Figure 8.6a) is an input with some interesting characteristics such as the narrow part of the tail. The tail will create two small triangular faces, which will need to be optimised.
- Tan_cat_new (Figure 8.6b) is slightly less complex than the previous tangram cat. Comparisons with the other tangram cat could be interesting.
- Two_lads (Figure 8.6c) is one of the most complex input. It consists of several distinct polygons. Optimisation-wise it probably will not perform well because the subdivision will be very cramped and will not allow small faces to increase in width. From a puzzle point of view, it is very interesting as it should be quite complex to solve. Its difficulty should probably be the highest of all inputs.

- Two_squares (Figure 8.7a) is interesting from the solver point of view because it is not uniquely solvable. Therefore it is a good test to check whether the solver will get stuck or not.


(a) Back_seven


(b) Dragon_bird

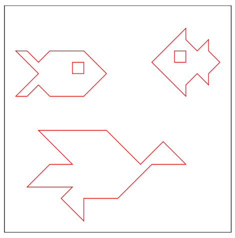
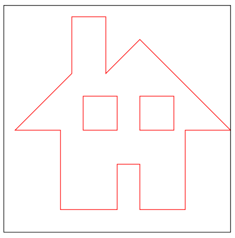
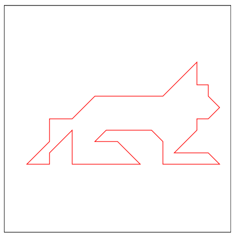

(c) Fish

Figure 8.1: Back_seven, dragon_bird, and fish.


(a) House


(b) Lion

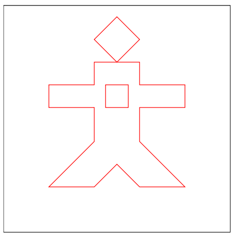
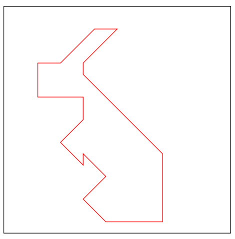
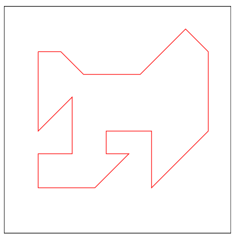

(c) Man_hole

Figure 8.2: House, lion and man_hole.


(a) Rabbit


(b) Random

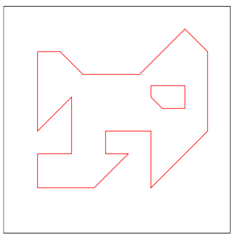
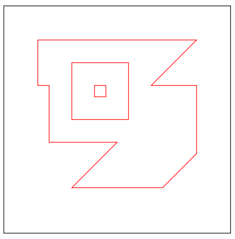
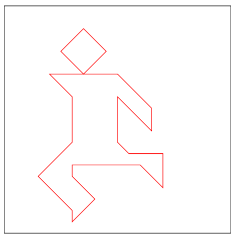

(c) Random_hole

Figure 8.3: Rabbit, random and random_hole.


(a) Recurs_holes


(b) Running

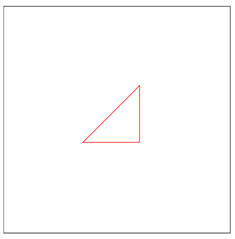
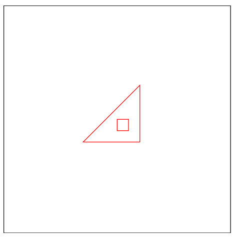
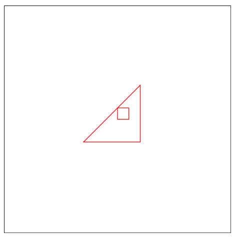

(c) Simple_triangle

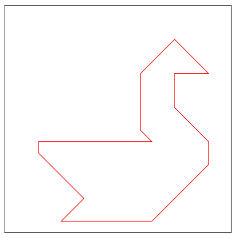
Figure 8.4: Recurs_holes, running and simple_triangle.



(a) Simple_triangle_hole

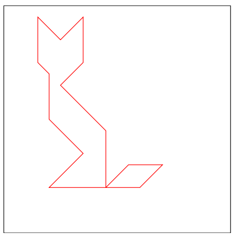


(b) Simple_triangle_hole_tangent

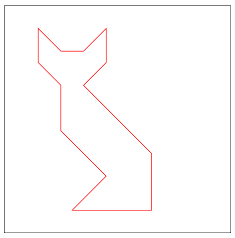


(c) Swan

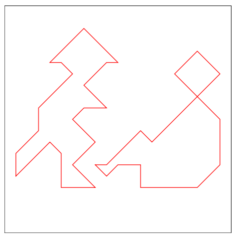
Figure 8.5: Simple_triangle_hole, simple_triangle_hole_tangent and swan.



(a) Tan_cat

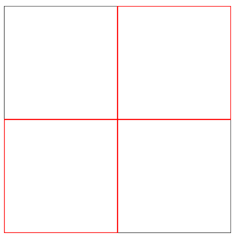


(b) Tan_cat_new



(c) Two_lads

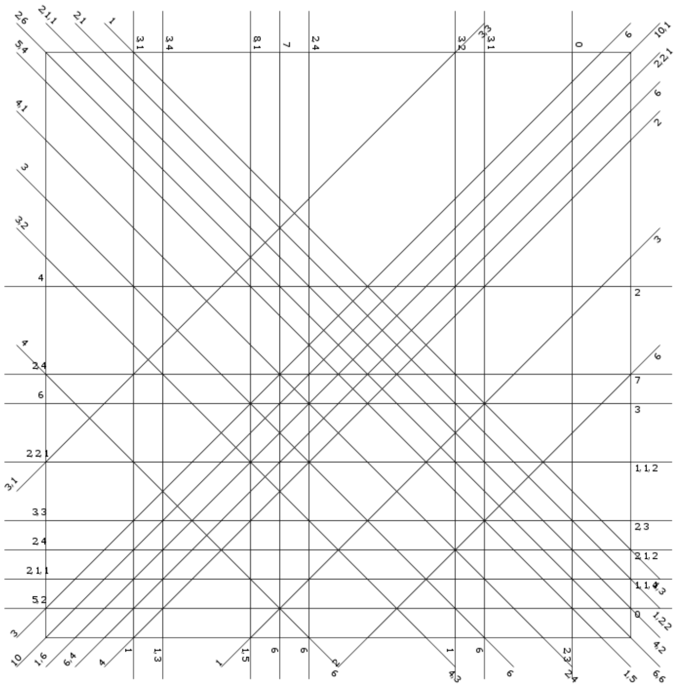
Figure 8.6: Tan_cat, tan_cat_new and two_lads.



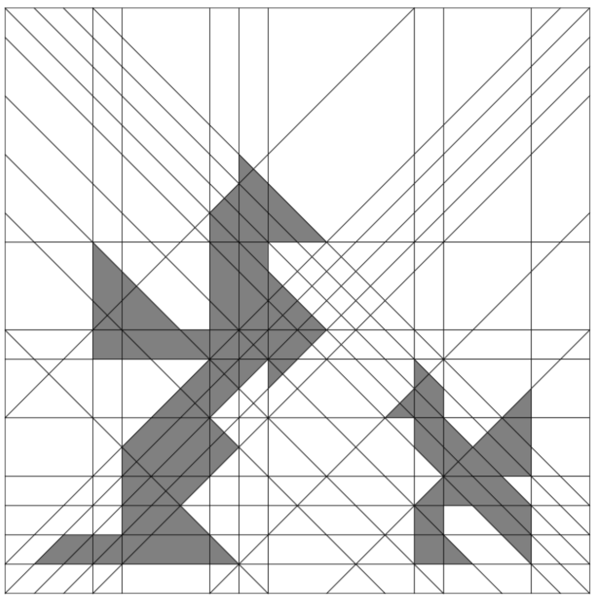
(a) Two_squares

Figure 8.7: Two_squares.

8.2 Sample sloped nonograms

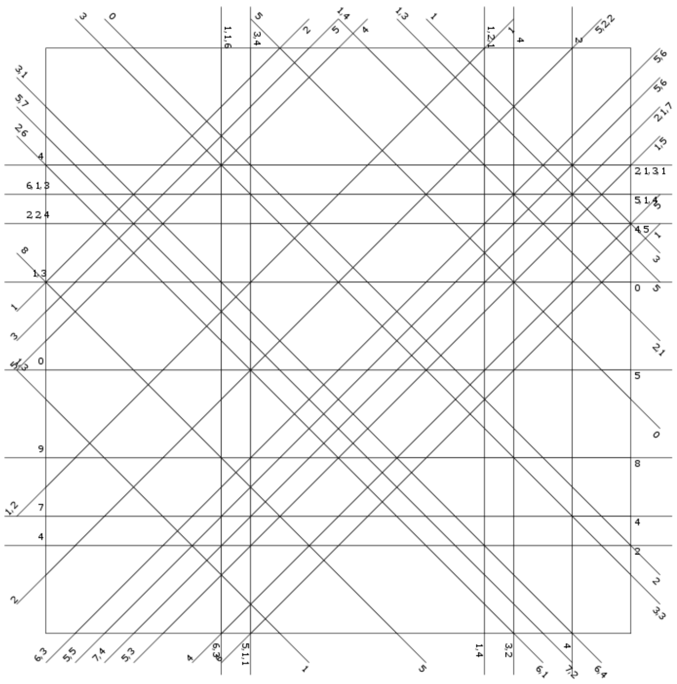


(a) Dragon_bird

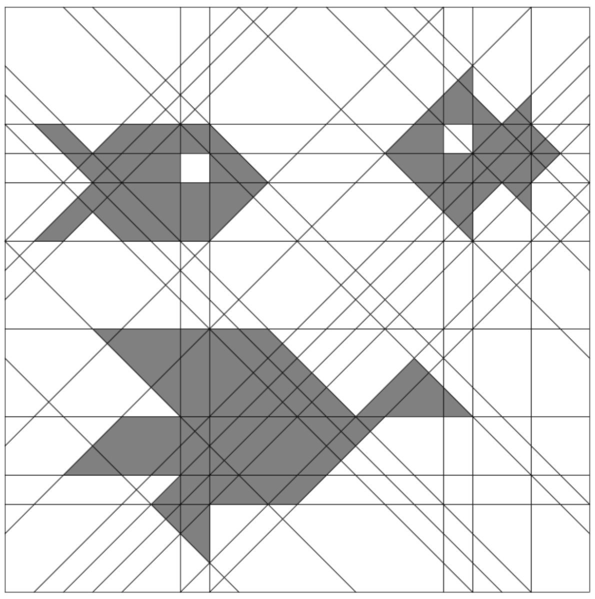


(b) Dragon_bird solved

Figure 8.8: Dragon_bird nonogram.

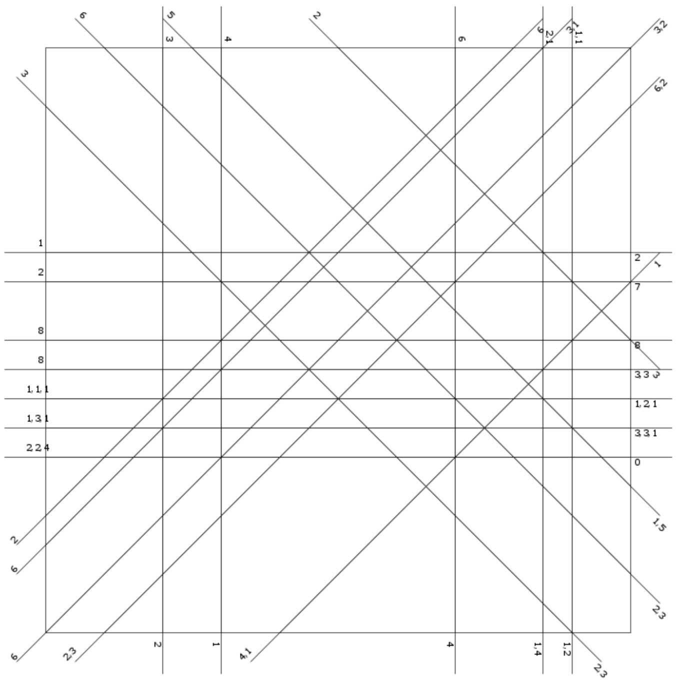


(a) Fish

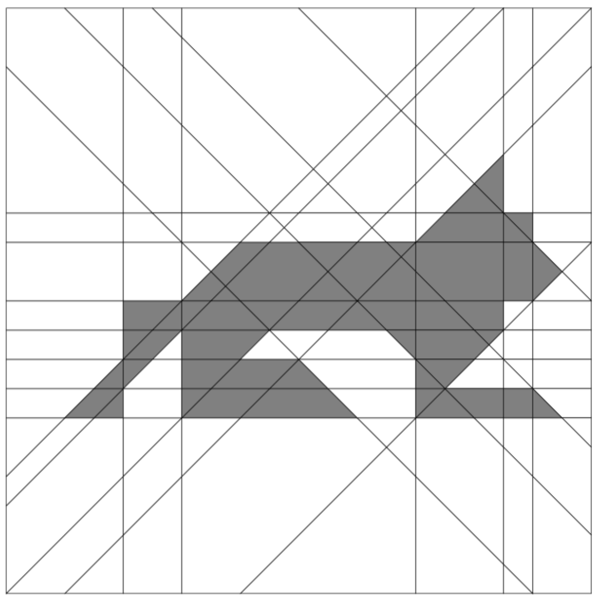


(b) Fish solved

Figure 8.9: Fish nonogram.



(a) Lion



(b) Lion solved

Figure 8.10: Lion nonogram.