ON THE ESTIMATION OF THE NUMBER OF SOLUTIONS FOR NONOGRAMS

Lappeenranta-Lahti University of Technology LUT

Master's Program in Computational Engineering , Master's Thesis

2022

Henrik Valve

Examiner: Professor Tapio Helin
Professor Lassi Roininen

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
LUT School of Engineering Science
Computational Engineering

Henrik Valve

On the estimation of the number of solutions for nonograms

Master's Thesis
2022
87 pages, 47 figures, 0 table, 1 appendix
Examiners: Professor Tapio Helin, and Professor Lassi Roininen

Keywords: Nonograms, Japanese Puzzle, Number of Solution, Approximation, Estimation, NP-completeness

How possible is it to estimate the number of solutions of a nonogram problem? Although this thesis does not fully answer this question, it shows there is a category of nonograms where polynomial time can be achieved. The proposed algorithm for estimation partial solves a nonogram and then finds switching components from the partial solution. Switching component types are identified and the number of solutions for a type is estimated. The thesis concludes with a discussion of the feasibility of extending the algorithm by adding more types. Most likely, extending the algorithm will lead to a non-polynomial time algorithm. This could be because there is no finite amount of switching component types or detectors for a combination of types. Regardless, more types could be added to the algorithm.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
LUTin insinööritieteiden tiedekunta
Laskennallinen tekniikka

Henrik Valve

Nonogrammin ratkaisujen määrän arviointiin liittyen

Diplomityö
2022
87 pages, 47 figures, 0 table, 1 appendix
Tarkastajat: Professori Tapio Helin, ja Professori Lassi Roininen

Avainsanat: Nonogrammi, Japanilainen ristikko, Vastausten lukumäärä, Approksimaatio, Estimaatio, NP-täydellisyys

Kuinka mahdollista on NP-täydellinen nonogrammin vastausten määrän estimointi? Tähän ei tämä työ täysin vastaa, mutta työ osoittaa nonogrammien kategorian, jota voidaan polynomisessa ajassa estimoida. Ehdotettu algoritmi estimointiin osittaisratkaisee nonogrammin, ja sitten etsii osittaisratkaisusta vaihto komponentit. Vaihto komponentin tyyppi tunnistetaan ja vastausten määrän estimointi tehdään tyypille. Työn lopussa pohditaan algoritmin jatkoa tyyppejä lisäämällä. Todennäköisesti, tyyppien lisääminen päättyy epäpolynomiseen algoritmiin. On mahdollista, että vaihto komponentteja ei saada luokiteltua rajattomalla tavalla tai tyyppien yhdistelmiä ei pystytä havainnoida. Joka tapauksessa tyyppejä voidaan lisätä algoritmiin.

LIST OF SYMBOLS

Number sets

$\mathbb{N}$	Set of natural numbers $(0, 1, 2, 3, 4, ...)$.
$\mathbb{N}_{\leq k}$	Set of natural numbers less than or equal to k.

Graph Theory

V	Set of vertices.
E	Set of edges. Pairs (v_0, v_1) of vertices of V.
(V, E)	A graph.

Complexity Theory

$\mathbf{O}(f)$	Big-Oh notation.
PROBLEM	Set of instances of some problem or just the problem.
ASP-PROBLEM	ASP version of some problem.
nASP-PROBLEM	nth ASP version of some problem.
#PROBLEM	Counting version of decision problem.
CLASS	Complexity class (set of problems).
CLASS-complete	Complete subset of complexity class under some reduction.

Nonogram

c_x	xth pixel in the line. Possible values are black, white, and unknown.
$c_{x,y}$	Pixel at column $x$and row y.
b_j	jth block in the line's description.
$(b_{j,L}, b_{j,R})$	Block range of jth block in the line's description.
$b_{c,l,j}$	Block j in l line index of column descriptions.
$b_{r,l,j}$	Block j in l line index of row descriptions.
Q_i	jth pixel group in the line.
N	Nonogram.
T	Partial solution or proposed partial solution.
s	Switching component.
S	Set of switching components.

CONTENTS

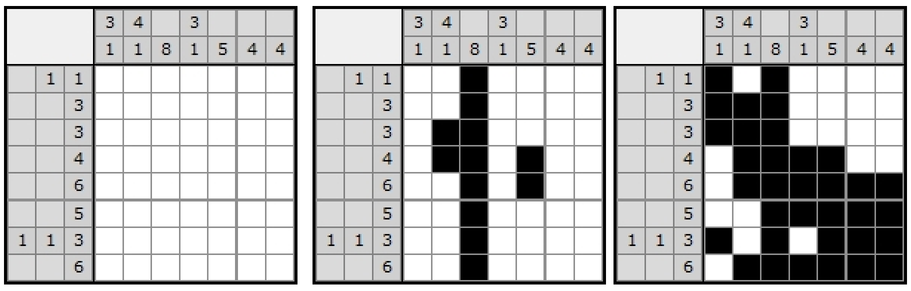
1 INTRODUCTION

1.1 Background

Aim of the Thesis it to investigate feasibility of Fully polynomial-time randomized approximation scheme (FPRAS) for counting number of solutions of a nonogram. This is done because:

- In the literature there are no attempts to make FPRAS to solve the number of nonogram solutions.
- If no FPRAS exists (which is the conjecture at the moment), then it is not obvious why. Is there a small subset of special cases which do not allow it? What difference is there between instances which allow FPRAS and which do not?

Nonogram is a puzzle where the player colours a grid of *pixels* to make a pixelated image based upon numbers at the sides called *descriptions*. Each row and column has one description. Numbers in descriptions are lengths of the black pixel *blocks*. Rows and columns of nonograms are generally called *lines* in this Thesis. Blocks have to be in the same order in the line as they appear in the description. Between each block there has to be at least one white pixel. Crossovers of row and column descriptions define where black pixels are. Figure 1 shows an empty (a), partially coloured (b), and coloured nonogram of a cat (c).



(a) (b) (c)

Figure 1. Nonogram of a cat. (a) Empty, (b) partially coloured, and (c) fully solved. [1]

Nonogram can have multiple solution. Figure 2 shows five by four (5×4) nonogram's six solutions. There are two sets of solutions. Solutions, on Subfigures (a) to (d), have a 2 by 2 black box in the upper left corner. Three black pixels down right are allowed in this configuration to "move" around. Last two solutions, on Subfigures(e) and (f), have jagged pattern in the upper left. This allows only two black pixels on left to "move" around.

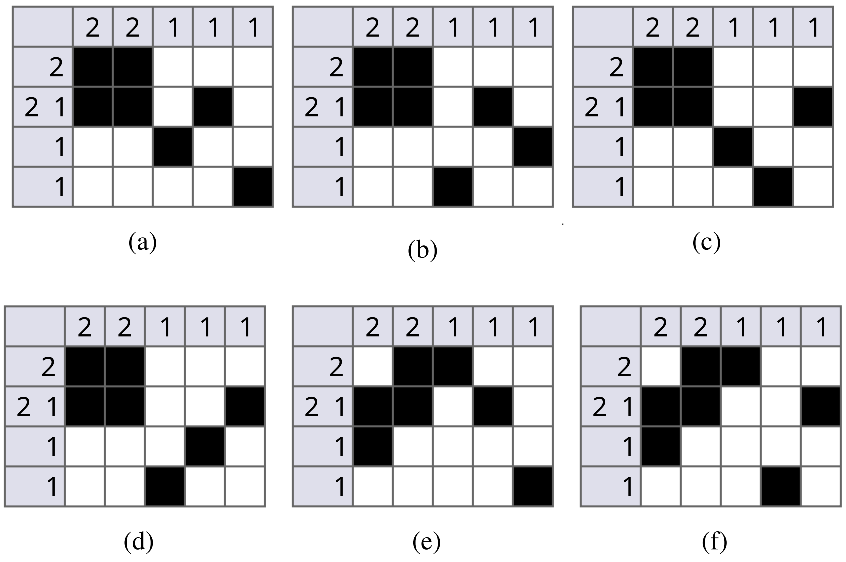


Figure 2. All solutions of a five times four nonogram.

Nonogram's solutions may not have simple structure. An example of very different solutions for a nonogram was presented in [2] shown at Figure 3.

Although nonograms are solved by people for fun, computationally solving Nonogram is known to be **NP**-complete problem [3]. Nonogram's solution counting is also known to be #**P**-complete. Problem #3SAT is #**P**-complete[4]. Problem 3SAT has parsimonious reduction to 3DM making #3DM #**P**-complete [5]. Since reduction from 3DM to NONOGRAM is parsimonious, #NONOGRAM is #**P**-complete [3].

Nonogram being **NP**-complete and #**P**-complete means that, if FPRAS exists for counting number of solution of a nonogram then **RP** is equal to **NP** which is not expected. However, that does not mean there is not subset of instance, which are easily solvable. For example, counting number of solutions for Boolean Constraint Satisfaction Problem has subset of instances which can be calculated **exactly** in polynomial time [6].

Nonogram's multiple solutions are caused by switching components. Switching components are sets of pixels in the solution which can move around to give another solution [2]. The simplest switching component is Elementary switching component (ESC). Example of ESC is shows as Figure 4. This is 2 by 2 nonogram where every description is one block length of one. In the Subfigure (a) nonogram is empty. Subfigures (b) and (c) are solutions to ESC.

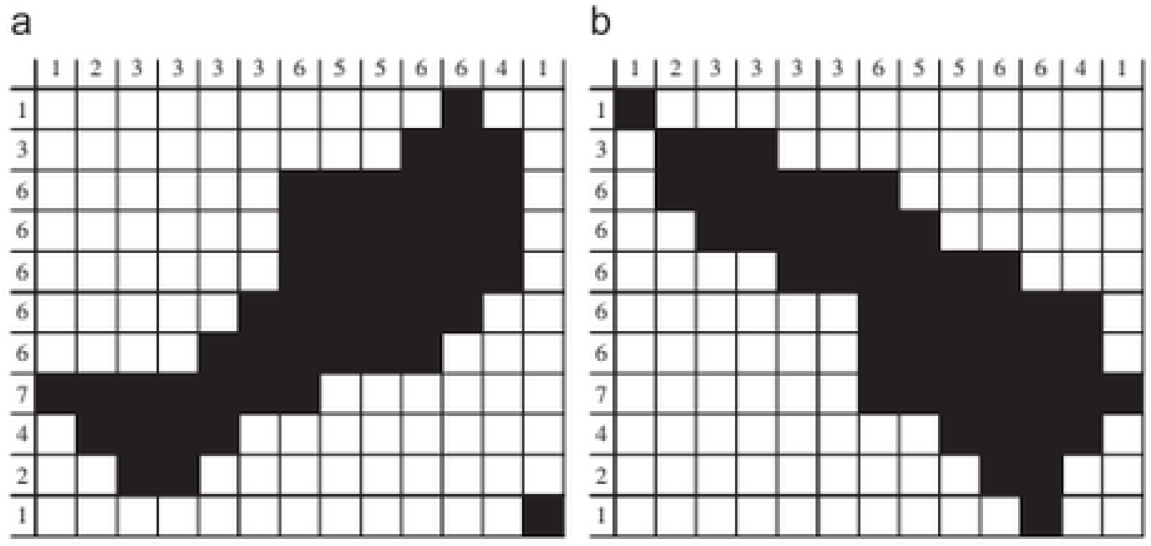


Figure 3. Two solutions for nonogram which are rather different [2].

Four pixels in a ESC do not have to be next to each other [2].

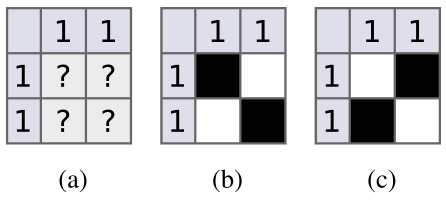


Figure 4. Most basic ESC. (a) Nonogram empty. (b) First solution. (c) Second solution.

One nonogram may have multiple switching components depend upon surrounding pixels. Pixels in the Figure 5 have values black, white, or unknown. Pixel in the Figure 5 value is shown at left corner of the pixel. Black pixel has black filled square. White pixel has white filled square. Unknown pixel has question mark at the corner. Figure 5 shows two isolated (as described in [7]) ESC as unknown pixel groups. Arrows drawn between pixels are block's ranges of uncoloured blocks. They show range in which uncoloured block has to be. More on them later. Isolation layer is the immediate surrounding pixels of the ESC. In isolation layer, pixels along the line next to unknown pixels must be white. Isolation layer does not need to have black pixels as corners.

For number of solutions approximation, literature does not have anything nonogram specific algorithm. Daniel Berend *et al.* in [7] gave exponential time algorithm for getting **exact** number of solutions. Their algorithm calculates number of solutions recursively by finding every possible colour configuration of a column while considering already coloured columns.

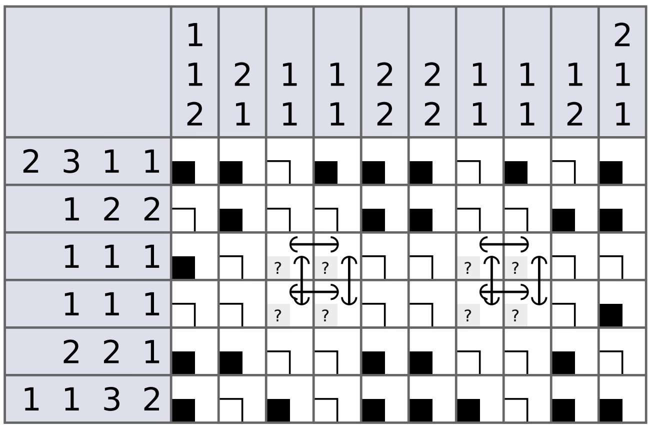


Figure 5. Two isolated ESC in a partial solution of nonogram.

If all the columns are coloured so that rows are valid, then count is increment by one. Otherwise, (or if column has no valid configuration) zero is added. It is a brute force method with some consideration. Problem of using brute force method is that potential solution count of nonogram is exponentially high. So visiting all solution of a nonogram can take too much time. There is potential in [7] algorithm in terms of modifying to sampling but that is taken as outside of scope.

Literature did not have FPRAS for counting number of solution for nonogram. However, general methods for counting do exists. Vazirani in his book about approximation [8] has an chapter on counting problems. It focuses on problems in which random variable X over solutions which expectation is number of solutions. Essentially, algorithm works by generating solution with such a probability that value for X will cause the expectation to be number of solution. If variance is only in "polynomial range", then algorithm is FPRAS.

Previous method requires that solution can be generated by sampling. For **NP**-complete problem this is not possible because solution generator can be a solution searcher, so it would imply **P** is equal to **NP**. Hence, some form of Monte Carlo is used to sample a space which solutions are a subset. Still, every sampling method for counting does come back to having random variable X in which expectation is number of solutions and variance is "polynomial range" from expectation. For example, in [9] Rubinstein experimented with more advanced Monte Carlo methods *cross-entropy* and *minimum entropy*. To thesis purposes sampling is avoided as it can be tacked on later when more information about nonogram itself does exists.

There are multiple attempts of using machine learning algorithms to solve NONOGRAM (for example see [10]). These studies usually experiences exponential behaviour, but given methods are modifiable for probabilistic approximation of a solution. Thesis does not jump to these machine learning methods. Rather, starting point is logical rules from [11].

1.2 Objectives and delimitations

Objective of this thesis is to investigate the feasibility of efficient probabilistic approximation algorithm for #NONOGRAM. This is done by partially solving nonogram and then investigating the sets of unknown pixels (switching components). Unfortunately, due to time limitations, fully working approximation algorithm was not possible to implement. In fact, the thesis does not get to probabilistic approximation at all.

To be more precise, the objectives of this Thesis are:

- Provide algorithm for 1 and most of its subalgorithms:
  - partial solver using line rules,
  - nonogram verifier with partial solution support,
  - algorithm to find switching components from the partial solution,
- Provide complexity analysis main algorithms both mathematically and empirically.

The biggest delimitations of the study are:

- function ***estimateSwitchComponent*** is restricted to **exactly** counting of one-black colourable one-pixel SSC,
- it is assumed that it is possible that #NONOGRAM has a FPRAS even though there are two reasons to doubt:
  - there is no proof that NONOGRAM is self-reducible which would indicate Nonogram does not have a FPRAS,
  - as nonogram is **NP**-complete and #**P**-complete there is conjecture against #NONOGRAM having a FPRAS which involves **P** vs **NP**,

Reason for assuming #NONOGRAM has a FPRAS because Thesis still generates information about boundary between Nonograms which solution count can be approximated.

Francisco Pesquita's *nonogram-solver* program (found at [12]) was used as inspiration for the Thesis software (found at [13]).

1.3 Structure of the thesis

Previous sections already used Complexity Theory terminology. If this field is alien to reader, **Appendix** 1 is provided those how want overview into the subject. Chapters of thesis focus on different parts of the Algorithm 1 and theoretics.

Chapter 2 explain notation used in the Thesis, main definition, and describes the overall idea of the algorithm.

Chapter 3 explains the partial solver. Partial solver uses logical rules per line deduce information. These rules are mostly directly from the [11] and explained in Section 3.1. In the Subsection 3.2 is more detailed discussion on the implementation of the partial solver.

Chapter 4 is about post-processing useful information out of a proposed partial solution. In Subsection 4.1 algorithm for verifying is proposed partial solution a partial solution, a solution, or invalid result is explained. Subsection 4.2 introduces unknown pixel graph and an algorithm to generated it from a partial solution.

Chapter 5 Explains detection algorithms for ESC and one-black colourable one-pixel SSC.

Chapter 6 provides complexity analysis of Algorithm 1 and it's direct sub-algorithms in Subsection 6.2. Both mathematical and empirical methods are used. In Subsection 6.1 is short explanations of generator used to drive empirical testing results.

Chapter 7 discusses FPRAS feasibility and **Chapter** 8 concludes the Thesis. **Chapter** 7 Subsection 7.1 discusses more future work regarding nonograms.

2 OVERVIEW

To formalize some definitions, *Nonogram* is a pair of finite sequences of descriptions. Description is finite sequence of blocks. They are marked $\{b_{c,l,j}\}$ for column description and $\{b_{r,l,j}\}$ for row description where l is index for the line and j is index for the block. If only blocks of one line is discussed a shorter version of b_j is used for the blocks. If b_j is used in arithmetic, it means length of the block. Otherwise, notation means the block "object".

Block range is a pair of pixel indexes $(b_{j,L}, b_{j,R})$ for block b_j where $b_{j,L}$ is the leftmost possible position of the block and $b_{j,R}$ is the rightmost. Refinement or updating the block range is to make ether $b_{j,L}$ or $b_{j,R}$ closer to other.

Partial solution to a nonogram (N) is table of pixels T which is logically deduced. For pixel $c_{x,y}$ in T subscript variables are column index x and the row index y. When discussing single line in nonogram, only column index is used. In Thesis purposes, pixel $c_{x,y}$ can have three values. Two colours black and white, and unknown for pixels the colour is undecided. What logically deduced means is that, if T is fully coloured to a solution, then number of such colourings would be same as #NONOGRAM(N). Before partial solver solution table for nonogram is full of unknown pixels.

A solution T to a nonogram N is a table of pixels with every $c_{x,y}$ having ether value black or white, and lines of T are coloured according to descriptions of N. If white and black are encoded as 0 and 1, the solution is valid if following equations apply for each row and column:

The sum over pixels on every column x_0 before zeroth block starts ($b_{c,x_0,0,L}$) is 0 (pixels are white)

$$\sum_{y=0}^{b_{c,x_0,0,L}-1} \left(c_{x_0,y}\right) = 0. \tag{1}$$

The sum over pixels on every row y_0 before zeroth block starts ($b_{r,y_0,0,L}$) is 0

$$\sum_{x=0}^{b_{r,y_0,0,L}-1} \left(c_{x,y_0}\right) = 0. \tag{2}$$

On every column x_0 and every block index j between block starting ($b_{r,x_0,j,L}$) and ending ($b_{r,x_0,j,R}$) there has to be block amount of black pixels, without any white pixels. Hence, since black pixel is encoded 1

$$\sum_{y=b_{c,x_0,j,L}}^{b_{c,x_0,j,R}+b_{c,x_0,j}-1} \left(c_{x_0,y}\right) = \sum_{y=b_{c,x_0,j,R}-b_{c,x_0,j}+1}^{b_{c,x_0,j,R}} \left(c_{x_0,y}\right) = b_{c,x_0,j}. \tag{3}$$

On every row y_0 and every block index j between block starting ($b_{r,y_0,j,L}$) and ending ($b_{r,y_0,j,R}$) there has to be block amount of black pixels, without any white pixels. Hence, since black pixel is encoded 1

$$\sum_{x=b_{r,y_0,j,L}}^{b_{r,y_0,j,R}+b_{r,y_0,j}-1} \left(c_{x,y_0}\right) = \sum_{x=b_{r,y_0,j,R}-b_{r,y_0,j}+1}^{b_{r,y_0,j,R}} \left(c_{x,y_0}\right) = b_{r,y_0,j}. \tag{4}$$

On every column x_0 block $b_{c,x_0,0}$ range must at pixel zero at least

$$0 \leq b_{c,x_0,0,L}. \tag{5}$$

On every row x_0 zeroth block must end at pixel zero at least

$$0 \leq b_{r,y_0,0,L}. \tag{6}$$

On every column y_0 between blocks (b_j and b_{j+1}) has to be space for at least one white pixel

$$b_{c,y_0,j,R} < b_{c,y_0,j+1,L} + 1. \tag{7}$$

On every row x_0 between blocks (j and $j+1$) has to be space for at least one white pixel

$$b_{r,x_0,j,R} < b_{r,x_0,j+1,L} + 1. \tag{8}$$

On every column y_0 last block $b_{c,y_0,k}$ must be less than length of the line (m)

$$b_{c,x_0,k,R} < m. \tag{9}$$

On every row x_0 last block $b_{r,x_0,k}$ must be less than length of the line (n)

$$b_{r,y_0,k,R} < n. \tag{10}$$

On every column x_0 between two blocks (j and $j+1$) is only white pixels

$$\sum_{y=b_{c,x_0,j,R}+1}^{b_{c,x_0,j+1,L}-1} \left(c_{x_0,y}\right) = 0. \tag{11}$$

On every row y_0 between two blocks (j and $j+1$) is only white pixels

$$\sum_{x=b_{r,y_0,j,R}+1}^{b_{r,y_0,j+1,l}-1} \left(c_{x,y_0}\right) = 0. \tag{12}$$

On every column x_0, with m pixels, after last block (k been last block in description) pixels has to be white

$$\sum_{y=b_{c,x_0,k-1,R}+1}^{m-1} \left(c_{x_0,y}\right) = 0. \tag{13}$$

On every row y_0, with n pixels, after last block (k been last block in description) pixels has to be white

$$\sum_{x=b_{r,y_0,k-1,R}+1}^{n-1} \left(c_{x,y_0}\right) = 0. \tag{14}$$

Rules here are written in the same vein as in [14] which describe an Integer Programming formulation for nonogram. Main difference between here and [14] is that [14] indicated starts and end of the blocks by "boolean" per pixel, block, and line rather than variables $b_{j,L}$ and $b_{j,R}$ indicating, which pixel in the line block starts and ends.

From equations two trivial types of nonogram with no solutions can be identified. One is total number of pixels in column descriptions vs. row descriptions, other one is regarding maximum number of blocks description can have for given line. Thesis assumes that nonogram inputted to algorithms are not these trivial cases as they can be detected in polynomial time before algorithm is run.

Nonogram does not have any solutions, if row and column descriptions do not colour the same amount of pixels. This is, because some pixel would have to be white according to one line and black according to an other. This would be failing of Equation 3, or 4.

Length of the description is minimum length of the line for description to fit into a line [15]. It is defined as

$$\boldsymbol{l}(\{b_j\}) = \sum_{j=0}^{k-1} (b_j) + k - 1, \tag{15}$$

where k is number blocks in the description. Sum is number of black pixels line blocks take and last term $k-1$ is the minimum number of white pixels the line must have. Minimum is number of white pixels occurs if zeroth block starts at pixel zero, kth block ends at last pixel of the line, and between blocks there is only one white pixel. If $\boldsymbol{l} > n$, where n is the length of the line, then description does not fit to the line, hence there is no solution. If nonogram has description with length longer than the line then description would end up violating one of the Equations 1 to 14 as there simple is not enough space.

In [15] the description is called *forceless* if

$$\boldsymbol{max}\left(\{b_j\}\right) + \boldsymbol{l}\left(\{b_j\}\right) \leq n$$

, where n is length of the line. From this Thesis call *forceful* description (or line) is as the logical opposite. Special case of forceful description where

$$\boldsymbol{l}\left(\{b_j\}\right) = n$$

is called *static*.

In [2] switching component and its relation to Discrete Tomography was discussed but no formal definition was given. To fix this, let's define *proposed switching component* as set of unknown pixels in which colouring of one of them affects colour or block range covering of other unknown pixel in the set. *Switching component* is then a proposed switching component with multiple colourings. By the definition proposed switching component does not necessarily have multiple solutions, or colouring one pixel in switching component does not imply colour of all the other unknown pixels in the set.

Since nonogram with multiple switching components do not affect each other total number of solutions switching components cause to a nonogram is simply product of number colourings switching components have (as seen used in the Algorithm 1).

Theorem 2.1. *If partial solution T for Nonogram N has $U+1$ switching components $A_0,...,A_U$ with $C_0,...,C_U$ colourings for each switching component then* #NONOGRAM $(N) = \prod_{i=0}^{U}(C_i)$.

By definition of switching component colouring pixels of A_0 will not affect any other switching components $A_1,...,A_U$ colouring. Hence, by simple multiplication principle #NONOGRAM $(N) = \prod_{i=0}^{U}(C_i)$.

As an analogue, situation is the same as having U number of different sized (C_i) decks of unique cards. If a person takes one card from each deck, then number of configurations of cards is also $\prod_{i=0}^{U}(C_i)$. ■

If it is possible to estimate the number of possible colourings for every type of switching component, then algorithm to estimating #NONOGRAM would reduce down to finding and identifying switching components in the nonogram. Algorithm 1 shows the overall idea. It takes in description of a nonogram and calculates a partial solution (here function ***PartialSolver*** does this). The role of partial solver is to get the switching components to be as small as possible. Algorithm uses modified nonogram's polynomial time verifier algorithm, (***NonoVerifier***) to check, if partial solver ends up to a solution, to state where nonogram does not have any solution, or to a partial solution T. If output is a partial solution T, then switching component finding algorithm (***FindSwitchesConponents***) is called. This creates set S of proposed switching components in the partial solution T.

Each member of set S contribute to the number of solutions of the nonogram with their own way. Function ***estimateSwitchComponent*** estimates the number of solutions a switching component causes. It returns *count* it estimates for the switching component and boolean *found* telling did it detect the type. The variable *count* stores the number of nonogram solutions approximation. The variable *undected* stores number of switching components which were not recognized by Function ***estimateSwitchComponent***. If type was not detected (*found* is `false`) then ***estimateSwitchComponent*** returned estimation is one. This prevents variable *count* be effect by detection failure.

If ***estimateSwitchComponent*** is a FPRAS or better for every type of switching component, then Algorithm 1 is also a FPRAS.

Algorithm 1 NonoEstimateSolutionCount

Input: Nonogram N

Output: Pair $(count, undetected)$where $count$ is estimation for number of solutions of N and $undetected$ is number of switching components not detected.

```
1.  $(T,e) := \textbf{NonoPartialSolver}(N)$        ; Run partial solver
2.  if $e$ is true return $(0,0)$    ; Error happened in partial solver which means no solutions
3.  $verification := \textbf{NonoVerifier}(N,T)$; Check is $T$ a solution, invalid, or partial solution
4.  if $verification$ = isSolution
5.      return $(1,0)$        ; $T$ is solution for $N$ so it is only solution
6.  if $verification$ = isInvalid
7.      return $(0,0)$        ; $T$ can not produce solution for $N$
8.  $count := 1$
9.  $undetected := 0$
10. $S := \textbf{FindSwitchesConponents}(N,T)$        ; Get switching components
11. for each $s$ in $S$
12.     $(temp, found) := \textbf{EstimateSwitchComponent}(N,T,s)$
13.     $count := temp \cdot count$
14. if $found$ is false then $undetected := undetected + 1$
15. return $(count, undetected)$
```

3 PARTIAL SOLVER

Partial solver's idea is to take in a nonogram and solve as much as possible. Figure 6 shows image of partial solver taking a nonogram and outputting proposed partial solution. Partial solver gives out **proposed** partial solution because algorithm can give out the unique solution of the nonogram or an end up in an error if invalid nonogram was given. **Note** that in the Figure 6 nonogram N is represented as empty nonogram but in the implementation nonogram stores only size of itself and descriptions without pixel table. In fact pixel table is allocated in the partial solver algorithm. Other hand proposed partial solution T stores the pixel table and block range information but no description or size information.

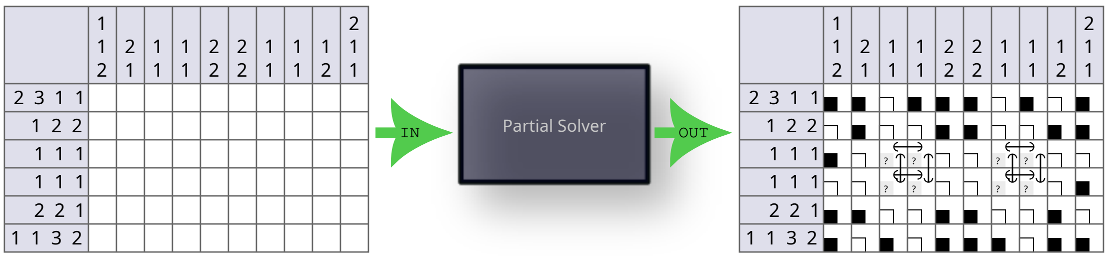


Figure 6. Partial solver takes in Nonogram N and outputs Proposed partial solution T.

There is plenty of solvers in the literature. Partial solver used in this Thesis is from [11] with some rules added and initialization made more clear. Solver in [11] uses line rules to ”reduce the [solution] search space” for a “chronological backtracking” (a form of depth-first search). Line rules are consecutively applied to a line in nonogram. After that algorithm moves to next line for the same treatment. After this is done for every line, and if any lines were updated, then new run of applying rules to lines is executed. If run does not have any updates, then chronological backtracking is used to get a solution. Thesis is not interested in chronological backtracking as it is to find a solution from partial solution.

Algorithm 2 is the algorithm presented in [11] without chronological backtracking. At beginning of Algorithm 2 is allocation of returning proposed partial solution (T) and initialization of it. Allocation here is for both pixel table and block ranges. It is also possible to initialization to detect in-valid nonogram so error check is performed in step 3. The main “do-while” loop of the Algorithm 2 (steps 4 to 9) loops until T is not updated. Inside main loop is “for each” loop that goes through every line (rows and columns) in nonogram N. Inside “for each” loop is second “for-each” loop which applies rules in set H to the line in partial solution T. Line rules are explained in detail in Section 3.1.

For the initialization function [11] gave an initialization rule. Reference did not give implementation on how to detect updates in T for the main “do-while” loop. In Section 3.2 more detailed version of Algorithm 2 is given. There initialization function is opened up, update detection explained, and error detection capability added. Section 3.2 also details hypothetical and real problems encountered with the implementation.

Algorithm 2 Partial Solver in [11]

```
Input: Nonogram $N$
Output: Pair of partial solution and a boolean indicating an error $(T,e)$.
 1. Allocate proposed partial solution $T$.
 2. $(T,e) :=$ Initialize$(T)$                 ; Initialize the partial solution.
 3. if $e$ is false                  ; check for an error during initialization.
 4. do
 5.     for each row and column $line$ in $N$
 6.         for each rule $\in H$
 7.             $(T,e) :=$ rule$(line,T)$
 8.             if $e$ is true break            ; error has happened end the loop.
 9. while there was an update in $T$
10. return $(T,e)$
```

Reason for choosing partial solver in [11] rather than some other one was its simple implementation. Algorithm in [11] does have cleaner mathematical properties for **partial** solving. For example, it is harder to adapt simulated annealing solver presented in [10] to partial solving, because it is not clear how visible switching components are during simulated annealing.

3.1 Rules to partially solve nonograms

Nonogram’s lines (rows or columns) are in some colour configuration (of unknown, black, and white pixels) during the solving process. Line rules update this state to be closer to fully coloured state (less unknown pixels). Line rules presented here are mostly from [11]. It divided line rules to 4 categories:

- initialization,
- rules which deduce pixel colour (1.1 to 1.5),
- rules which refine block ranges (2.1 to 2.3),
- rules which do both, pixel colour deduction and refining block ranges (3.1 to 3.3).

While testing individual rules in the implementation, they had problems to get block ranges to correct position. To solve these problems, rules 2.4 and 2.5 where added. Even these additions seem not to be enough when partial solver was tested. There were single solution nonograms which could not be solved (example in Section 3.2). Rule 0.0 was also added to have special handling for zero description lines in the initialization.

Common subalgorithm need in rules and in Chapter 4 is finding blocks which cover a pixel. Cover means pixel is in the block range. Algorithm 3 find the set of blocks which cover pixel c_x. Algorithm just goes through the blocks of given line and adds them to the set of pixel c_x is within block range.

Algorithm 3 FindBlocksCoveringPixel

Input: Nonogram N, Proposed solution T, Line *line*, pixel index x

Output: finite sequence of blocks $\{b_j\}_j$ covering pixel c_x of line l.

```
1. for each $b_j$ of $line$'s description in $N, T$.
2. if $b_{j,L} \leq x \leq b_{j,R}$
3.         add $b_j$ to $\{b_j\}_j$
4. return $\{b_j\}_j$
```

Rule 0.0 was added which marks every pixel in zero description line to white. Rule was added to initialization phase even though rule 1.2 does cover the same area, because it is faster to handle zero description lines on initialization phase.

In *initialization* starting block ranges are calculated by looking at the line's leftmost and rightmost arrangement for the blocks. For line size of n and with description $\{b_i\}_{i=0}^{k-1}$ then:

$$\begin{aligned}
&b_{0,L} := 0\\
&\forall j \in \{1,...,k-1\} \left(b_{j,L} := \textstyle\sum_{i=1}^{j-1} (b_i + 1)\right)\\
&\forall j \in \{0,...,k-2\} \left(b_{j,R} := n - 1 - \textstyle\sum_{i=j}^{k-1} (b_i + 1)\right)\\
&b_{k-1,R} := n - 1
\end{aligned} \tag{16}$$

Rule 1.1 checks every block (b_j) in the line. Pixel c_i is black if exists block b_j such that $b_{j,R} + 1 - b_j \leq i \leq b_{j,L} - 1 + b_j$. Figure 7 shows how initialization and rule 1.1 work. In Subfigure (a) line has unknown pixels before initialization. Initialization looks at leftmost and rightmost solution of the line. These are Subfigures (b) and (c). Resulting blocks ranges are in Subfigure (d). Applying rule 1.1 length of block 1 is 3 so that it has "middle" pixel (fourth in the line) which has to be black. This is shown in Subfigure (e). With Rule 1.1 you can fully solve *static lines.*

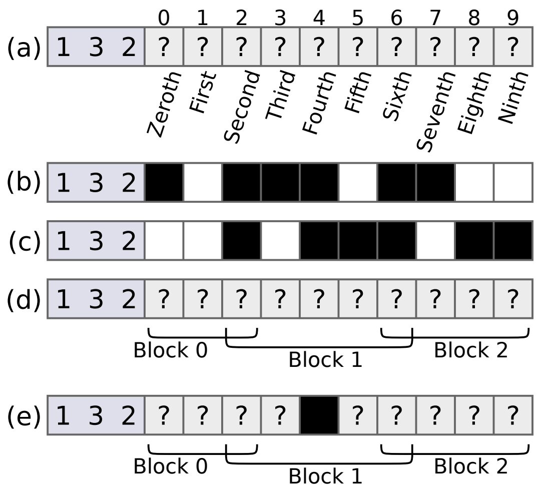


Figure 7. How initialization and rule 1.1 work. (a) The line before initialization, (b) leftmost solution of the line, (c) rightmost solution of the line, (d) the line after initialization, (e) line after rule 1.1 is applied to it.

Rule 1.2 handles situations where range update leaves pixels outside of any block range. Pixels in this state must be white, because Equations 1, 2, 11, 12, 13, and 14. Such pixels c_i satisfies following logical formula:

$$0 \leq i < b_{0,L} \vee b_{k-1,R} < i \leq n-1 \vee \exists j \in \{0,...,k-2\} \left(b_{j,R} < i < b_{j+1,L}\right) \tag{17}$$

How rule 1.2 works is shown in Figure 8. In Subfigure (a) pixels 0, 3, and 9 are outside of any block range because all black pixels of block 0 are known and block 1 range refined. Using rule 1.2 pixels 0, 3 and 9 can be deduced to be white. This is shown in Subfigure (b).

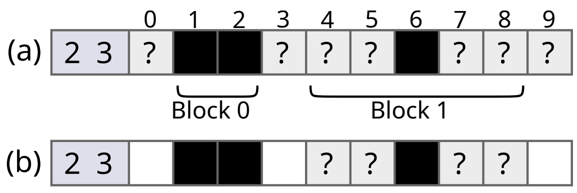


Figure 8. Shows how rule 1.2 works. (a) is the line before, (b) is the line after.

Rule 1.3 looks at each block b_j. If pixel $c_{b_{j,L}}$ or $c_{b_{j,R}}$ is black, and covered by other block ranges which are length of one then $c_{b_{j,L}-1}$ or $c_{b_{j,R}+1}$ is white. Figure 9 shows how rule is used. Subfigure (a) is the initial state of the line. Pixel 8 is black and $b_{3,L} = 8$. Pixel 8 ether belongs to second block or third block. In the Subfigures (b) pixel 8 belongs to second block, and in the Subfigure (c) to third block. Pixel 7 is white in both choices so pixel 7 should be white, as shown in (d). Do note that pixel 10 being black is not requirement of rule 1.3.

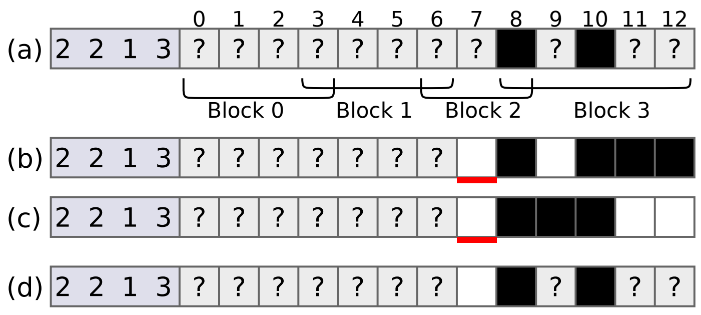


Figure 9. Visual of rule 1.3 in action. (a) is the line before, (b) has the pixel 8 is part of block 2 length of one, (c) has the pixel 8 be part of the block 3, (d) pixel 7 is set to white as it was white in both (b) and (c).

Rule 1.4 line has three consecutive pixels c_{i-1}, c_i, and c_{i+1} where c_{i-1}, and c_{i+1} are black. If biggest block covering c_i is less than block which be created, if c_i would be black, then c_i is white. Figure 10 shows an example case of rule 1.4. Subfigure (a) is the state of the line before rule 1.4. Block 1 has pixels 2, 4, and 5 as black in its range. In Subfigure (b) it is tested how long the block would be if the third pixel would be black. Block would be four pixels long which is bigger than block 1 which is a length of three. This means according to rule 1.4 that the third pixel is white, as pixel 2 has to be part of block 0, and pixels 4 and 5 has to be part of block 1.

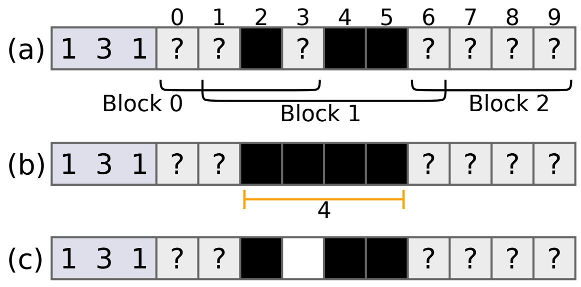


Figure 10. Rule 1.4 in action. (a) is line before applying the rule. (b) shows middle step in rule 1.4 computation if pixel 3 would be black line would have block length of 4. (c) shows that after rule 1.4 pixel 3 can be deduced to be white.

Rule 1.5 handles a scenario where pixel c_i is black and there is other pixel c_w which is closest white pixel left to c_i and the minimum block covering c_i is K. If $w \in \{i-K+1,...,i-1\}$ then $\forall p \in \{i+1,...,w+K\}\ (c_p := \text{black})$. This also works to right side of c_i so if exists $w \in \{i+1,...,i+K-1\}$ where c_w is the closest white pixel to c_i then $\forall p \in \{w-K,...,i-1\}\ (c_p := \text{black})$. Additionally, if all the blocks covering c_i are the same length and, if black pixels next to c_i create a block of length K starting at c_s ending to c_e, then $c_{s-1} :=$ white and $c_{e+1} :=$ white, if they exist.

Idea of the Rule 1.5 is that c_i been black means it must be part of some block. Necessarily, it is not known which block, but the nearby white pixel gives a boundary. Taking account this boundary, leftmost and rightmost placement of the block is narrowed down. If in every possible case block “goes over” the c_i in this narrowed down region, then pixels “going over” which every case agrees must be black. The smallest block is used because it’s possibilities cause the least amount of colouring. The additional colouring of pixels to white is made, because there is ready-made block in the line.

Figure 11 shows rule 1.5 in action. Subfigure (a) is the line before the rule is applied, where the third pixel is white and the fifth pixel is black. This pixel can be part of ether block 0 or block 1. Also, the fourth pixel could be either black or white. This causes four possible answers round the fifth pixel. These answers are shown in Subfigures (b) to (e). Note that the sixth pixel in every answer is black. These are the kinds of pixels rule 1.5 detects. The resulting line is shown in Subfigure (f).

Figure 12 shows an example where the additional last step is used to colour pixels 4 and 7 to white. Block 1 and 2 lengths are two. Pixels 5 and 6 are coloured black from rule 1.5 previous steps. This means that these pixels are ether block 1 or block 2 but either way fourth and seventh pixel can not be black or the block would be too long.

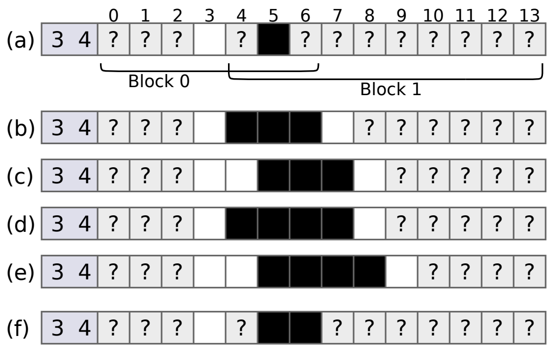


Figure 11. Idea behind rule 1.5 without last step of checking block lengths. (a) line before applying rule 1.5. (b)-(e) show every possible answer round fifth pixel. (f) is line after rule 1.5.

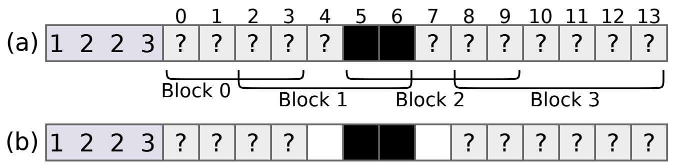


Figure 12. Example of rule 1.5 last step. (a) line has initial state before rule 1.5. (b) is line after rule 1.5 is applied.

Rule 2.1[*] checks that block b_j range satisfies block order requirement of Equations 7 and 8. If not block range can be refined to such state via logical formulas:

$$\begin{aligned} b_{j,L} \leq b_{j-1,L} + b_j &\Rightarrow b_{j,L} := b_{j-1,L} + b_{j-1} + 1 \\ b_{j,R} \geq b_{j+1,R} - b_j &\Rightarrow b_{j,R} := b_{j+1,R} - b_{j+1} - 1. \end{aligned} \tag{18}$$

Figure 13 shows an example of rule 2.1. In Subfigure (a) is the line before rule 2.1 is applied. In it, block 0 has been solved to be the third pixel, because pixels 2 and 4 are marked white. This means that block 1 can not be left of pixel 4, as it would break block order (Equation 7 and 8). Subfigure (b) is after rule 2.1 was applied. This means block 1 range is inside block 2 range. Applying rule 2.1 again moves block 2 leftmost further right as shown in Subfigure (c).

Rule 2.2 attempts to leave space between the blocks. If the pixel before the leftmost or after the rightmost pixel of the block range is black, then the block range can be tightened by one pixel, because blocks have to have at least one white pixel between (Equation 7 or 8). More

[*]Rule 2.1 trigger is more sensitive than in [11]

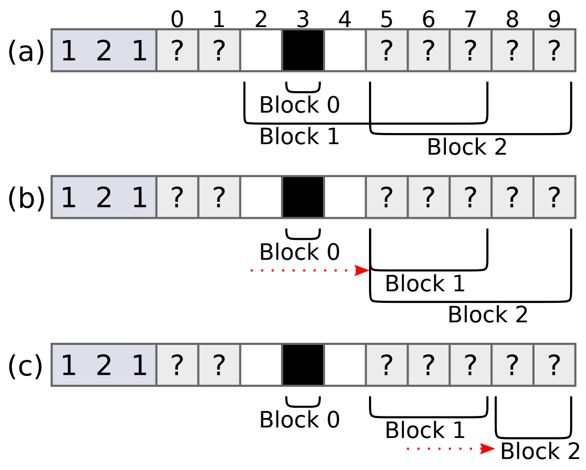


Figure 13. Example of rule 2.1. (a) before rule 2.1 is applied. (b) is after rule 2.1 is applied once. (c) is after rule 2.1 is applied second time.

formally rule is:

$$\begin{aligned} c_{b_{j,L}-1} &= \text{black} \Rightarrow b_{j,L} := b_{j,L} + 1 \\ c_{b_{j,R}+1} &= \text{black} \Rightarrow b_{j,R} := b_{j,R} - 1 \end{aligned} \tag{19}$$

In Figure 14 rule 2.2 is illustrated. Subfigure (a) shows the initial situation. Subfigure (b) shows line after the rule 2.2 was applied. Block 0 range end can be moved one to left because pixel 7 is black, and block 0 range rightmost is the sixth pixel. Block 1 range leftmost pixel can be moved one to the right because pixel 3 is black, and block 1 range starts at the fourth pixel.

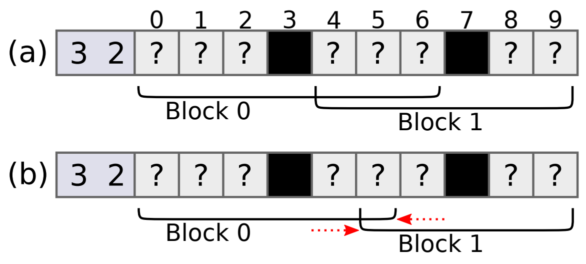


Figure 14. Example of rule 2.2. (a) the line before. (b) the line after. Note the red arrows.

Rule 2.3 checks that block b_j range does not cover a black pixel group which is bigger than b_j on it's left or right side. *Pixel Group* means pixels in the line which are the same "colour" (black, white, or unknown) or not that "colour" (non-black, non-white, or non-unknown) and are right next to each other (if c_{x-1}, c_x, and c_{x+1} are black and c_{x-1}, c_{x+1} are in black pixel group Q_o then c_i is also in Q_o). Let's define set of black pixels groups in b_j's range as $Q_j := \{Q_{i,j}\}_{i=0}^{|Q_j|-1}$ where $|Q_j|$ is the number of these groups. Let's also define $c_{s_{i,j}}$ and

$c_{e_{i,j}}$ pixels which start and end black pixel group $Q_{i,j}$ for b_j. With these rule 2.3 formally becomes:

$$\begin{aligned}
&0 < \mathbf{min}\left\{o|e_{o,j} - s_{o,j} \le b_j\right\} \\
&\quad \Rightarrow b_{j,L} := e_{\mathbf{min}\left\{o|e_{o,j}-s_{o,j}\le b_j\right\}-1,j} + 2 \\
&\left|Q_j\right| - 1 > \mathbf{max}\left\{o|e_{o,j} - s_{o,j} \le b_j\right\} \\
&\quad \Rightarrow b_{j,R} := s_{\mathbf{max}\left\{o|e_{o,j}-s_{o,j}\le b_j\right\}+1,j} - 2
\end{aligned} \tag{20}$$

Constant plus 2, and minus 2 refines block range out from group which is too large and leaves space for a white pixel.

Figure 15 shows an example of rule 2.3. In block 1 range there are two black pixel groups. The first one from the left is the length of three pixels. The second one has a length of one. According to the description, block 1 is the length of two, hence the group, whose length is three, can not be block 1. The block range is changed from Subfigure (a) to (b).

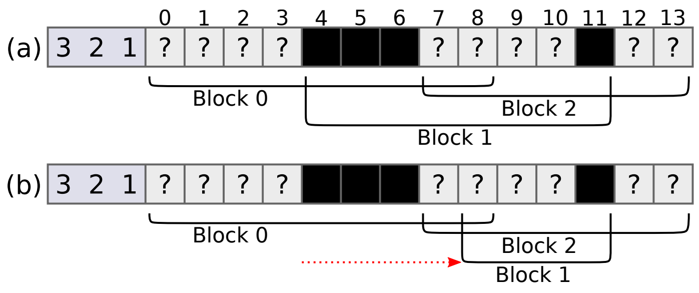


Figure 15. Example of rule 2.3. (a) the line before the rule is applied. (b) the line after the rule is applied.

Rule 2.4 refines block range to ready-made block. Ready-made means white pixel c_w followed by black pixel group Q_b which is followed by another white pixel c_v. Pixel group Q_b must be one of the blocks in the description, as no black pixel can be added to Q_b without writing over a white pixel. (Just to be extra clear $w < v \land \forall c_x \in Q_b\,(w < x < v)$.) Hence, if there is **one** block of length $|Q_b|$ covering pixels of Q_b, then that block range can be reduced to bounds of black pixel group. If multiple such blocks are found, then rule 2.4 tries to reduce block ranges of first such found block and last such found block. Since pixel pattern has to be explained by some block, deduction is made that leftmost block could explain the pattern. It cannot have any pixels more right than the pattern. For rightmost block, that could explain the pattern, opposite holds. It cannot have pixels more left than the pattern.

Figure 16 shows an example of Rule 2.4 in both cases. In Subfigure (a) is initial setup where there is two ready-made blocks. In Subfigures (b) and (c) these are marked by blue number on top of the line as zero and one pixel groups. In the initial setup block ranges are almost

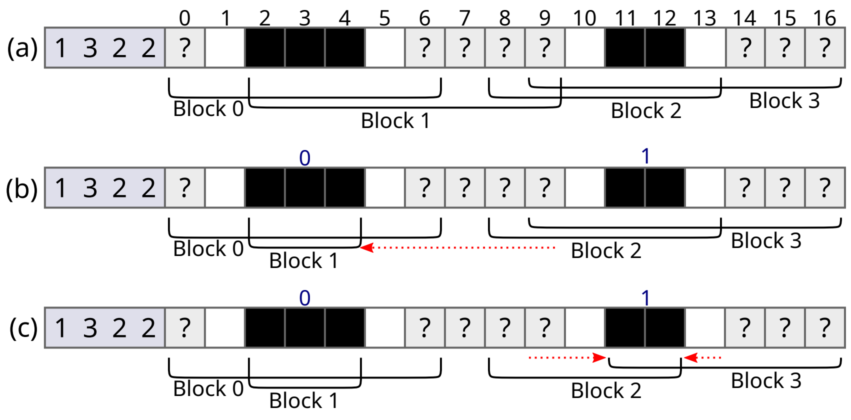


Figure 16. Example of rule 2.4. (a) the line before the rule is applied. (b) the line after rule 2.4 is applied to zeroth pixel group. (c) the line after rule 2.4 is applied to first pixel group.

in the position of which Initialization rule would give to them. In Subfigure (b) zeroth pixel group has a three black pixels. Since there is only block 1 covering pixel group zero, which length is 3, then block 1 is only block which could explain the pixel group 0. Hence, block 1 range is reduce to the pixel group. Rule 2.4 does not touch block 0 range since rule 2.1 would handle this situation. Subfigure (c) show rule 2.4 applied to pixel group 1. In this case, there are two blocks with length of two covering black pixel group of length two. Hence, block 2 right side has to be at least at pixel group 1 and block 3 left side has to be at least at pixel group 1.

Rule 2.5 was added in the same vein as rule 2.4. Rule 2.5 calculates most leftmost position and rightmost position of whole description of the line, taking into account white pixels. If calculated leftmost position for a block would be higher than block range left, then update would move border to leftmost position. If block b_j would fit between white pixels with space to spare, then algorithm would check does block b_{j+1} (if it exists) fit to that spare space accounting white pixel between the blocks. If block b_{j+1} does not fit, then algorithm would be moved to right of the white pixels. Same process is also done to right to left.

Figure 17 goes through step by step of rule 2.5 for example line. In Subfigure (a) is initial state of the line. Pixels 1, 7, and 11 are white and blocks 0, 1, and 2 are in the positions caused by initialization rule. There is four unknown pixel groups which block can be separated by white pixels. Processing of rule 2.5 takes two passes at the line. Left-right and then right-left. In Subfigure (b) left to right processing starts by trying to find location for block 0. Between starts of the line and leftmost pixel is unknown pixels group of length 1. (Pixel group and its length is shown as yellow bar bottom the Subfigure (b).) Length of block 0 is one hence block 0 can fit in that group. Since block 0 range left end starts at that pixel, no refined is made. Processing moves on from this to block 1 and pixel 2. In Subfigure

(c) same test as Subfigure (b) is made. Block 1 fits the unknown pixel (marked in yellow is the location and size). This time there is empty space after block 1 leftmost fit. Hence, in the Subfigure (d) block 2 is tested on that remaining space. After taking account the one white space needed to separate the blocks, Block 2 fits in the remaining space. This does not refine any block range as both block 2 and 3 leftmost location possible is the same as block ranges leftmost end. After this there is no more remaining space but also there is no more blocks to handle so left to right pass concludes. Right to left pass does refine block ranges. In Subfigure (e), on the right, there is a unknown pixel group of length one. However, block 3 in processing cannot fit in to that space. Hence, block 3 range is refined left to the white pixel (Subfigure (f)). Test is again made and, as shown in the Subfigure (g), this time block 3 does fit in the space. There is some remaining space left of the fit location but taking account for white pixel between the blocks this space is zero pixels. Zero pixel space is marked in Subfigure (h). Since block 2 cannot fit with block 3, its rightmost edge has to be on the next unknown pixel group. This means Block 1 range can be refined, as it is still in the righter pixel group (seen in the Subfigure (i)). In Subfigure (j) it is shown that block 0 and 1 can share the next unknown pixel group (pixels 2 to 6). However, the block 0 calculated rightmost position is more left than block 0 range rightmost. Hence, as shown in Subfigure (k), block 0 range is refined.

Rule 2.5 does also have nice feature of reducing block range if range's end is at white pixel because rule 2.5 calculates leftmost or rightmost position of the block. This is expressed as equations for later use:

$$\begin{aligned} &\left(c_{b_{j,L}} = \text{white}\right) \Rightarrow b_{j,L} := b_{j,L} + 1 \\ &\left(c_{b_{j,R}} = \text{white}\right) \Rightarrow b_{j,R} := b_{j,R} - 1 \end{aligned} \tag{21}$$

Rule 3.1 deals with black pixels which are covert only by one block range (b_j), and have unknown pixel between them. This causes unknown pixels to be black and to b_j range update when more of the block is known. First black pixel after $b_{j-1,R}$ is c_n and last black pixel before $b_{j+1,L}$ is c_m then:

$$\begin{aligned} &\forall p \in \{n+1, ..., m-1\} \left(c_p := \text{black}\right) \\ &b_{j,L} := m + 1 - b_j \\ &b_{j,R} := n - 1 + b_j \end{aligned} \tag{22}$$

Figure 18 shows rule 3.1 in action on a line where block 1 can be refined. Pixels 4 and 6 are black and are only inside block 1 range. Pixel 5 has to be black since otherwise block 1 would be violate Equation 3, or 4. Block 1 range can be refined as pixel 2 or 8 being black and belonging to block 1 would make block 1 too long.

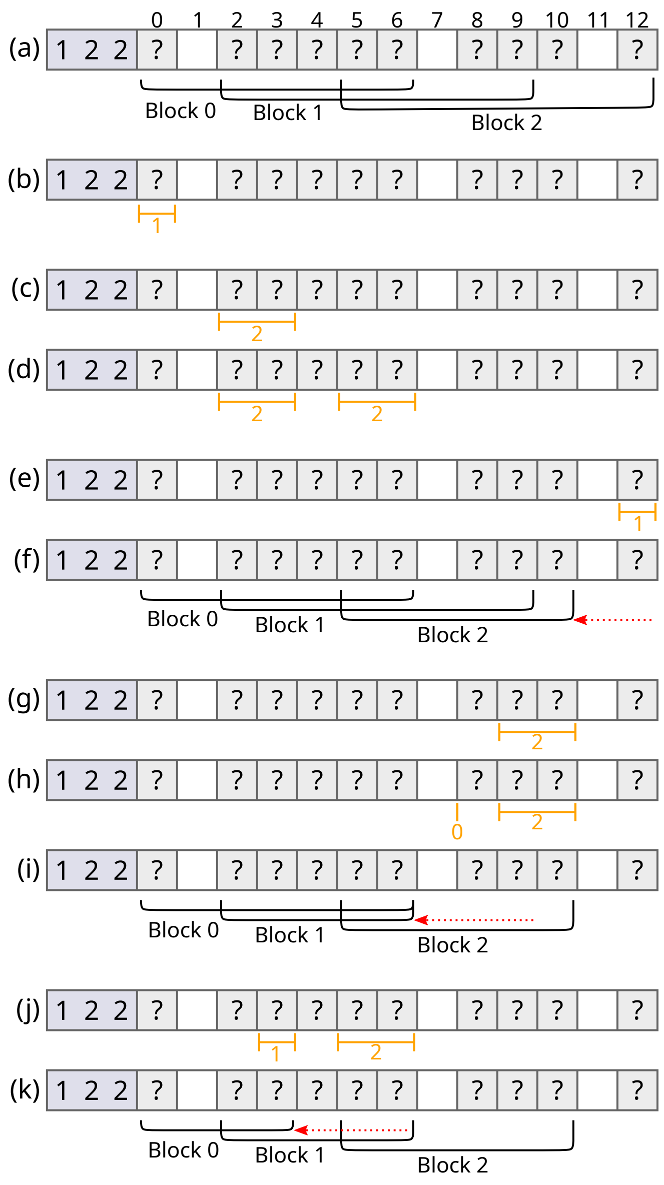


Figure 17. Example or rule 2.5 passing a line from left to right and then right to left. (a) In the initial state of the line. (b) is the test for placing block 0 to the left. (c) is test to place block 1 to the left most position. (d) is test to place block 2 to leftmost position. (e) is test to place block 2 to rightmost position. (f) block 2 range is refined to left as test failed. (g) is the test to place block 2 to new rightmost position which succeeds. (h) is test to place block 1 to rightmost position. (i) is block 1 range refined after test fails. (j) is the test to place block 1 and 0 to same unknown pixel group. (h) is refined of block 0 range which results.

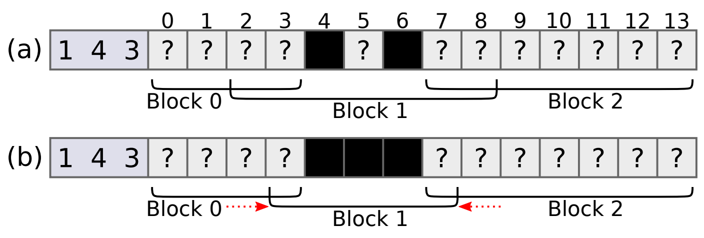


Figure 18. Example of rule 3.1. (a) the line before the rule is applied. (b) line after rule 3.1.

Rule 3.2 solves somewhat opposite of rule 3.1. If in the block b_j range is white pixels, then non-white pixels between white pixels can be grouped to finite sequence $Q := \{Q_i\}_{i=0}^{|Q|-1}$ where $|Q|$ is number of these groups in b_j range. If a group which is at the side of block b_j range is less than block length, it can be removed from the range. This can be done for the block's range left and right end separately. From the left or the right, range refining is stopped when a non-white pixel group, in which b_j fits, is found. If no group is found, then line is unsolvable. Block b_j range refining does not have to end from the left or the right to the same non-white pixel group. This can mean that there is non-white pixel groups in the middle of the two groups which are too small for block b_j. If these groups have pixels that are just in block b_j range, then pixels has to be white as no block can utilize them.

The algorithm for rule 3.2 is shown in Algorithm 4. Remainder, that in the input and output b_j means the whole block but in the arithmetic operation it means length of the block. The Algorithm starts by iterating the non-white pixel group Q from the left. After that, iteration is done from the right. Finally, (starting from step 15) groups in the middle are checked.

Figure 19 is an example of rule 3.2. In the Subfigure, a line is in the state where non-white pixel groups 1 and 5 are in block 1 range. Block 1 has a length of 3. Non-white pixel group 1 is one pixel and 5 two pixels. Block 1 can not fit into non-white pixel groups 1 and 5 so according to rule 3.2 block range can be reduced from both ends to the start of non-white group 2 and start of non-white group 4. Checking does block 2 fit into group 2 or group 4, the algorithm notices that block 1 fits into both, hence block 1 range reduction portion of rule 3.2 is done. Non-white pixel group 3 haves a length of 1 pixel. Since no other block range covers the group and block 1 would not fit into the group, it can be deduced to be white.

Rule 3.3 has three sub-cases for situation where block b_j range does not overlap with b_{j-1} or b_{j+1} range. Sub cases are explained for b_{j-1} range does not overlap with b_j range, but symmetry applies, and rules work for b_{j+1} range not overlapping long as directions are reversed.

Algorithm 4 Rule 3.2

Input: Line of pixels c_i, line length n, block b_j, finite sequence of non-white pixel groups Q.
Output: New Line of pixels c_i, and refined block b_j.

```
1.  Set $i_L := 0$                          ; Variable $i_L$ iterates $Q$ from the left to right
2.  If $|Q_{i_L}| \geq b_j$ go to 6          ; Try to find group which in $b_j$ fits
3.  Set $i_L := i_L + 1$                    ; Increment the iterator
4.  If $i_L = |Q|$ then stop                 ; Error as block $b_j$ can not be fitted anywhere
5.  Go to 2                                  ; Run other iteration of a loop
6.  Find pixel $c_s$ which is first pixel from left in $Q_{i_L}$
7.  Set $b_{j,L} := s$                       ; Removes left side groups too small for the block
8.  Set $i_R := |Q| - 1$                     ; Variable $i_R$ iterates $Q$ from right to left
9.  If $|Q_{i_R}| \geq b_j$ or $i_R = i_L$ go to 12.   ; Try to find group which in $b_j$ fits before $Q_{i_L}$
10. Set $i_r := i_r - 1$                     ; Decrement the iterator
11. Go to 9.                                 ; Run other iteration of the loop
12. Find pixel $c_e$ which is last pixel from left in $Q_{i_R}$
13. Set $b_{j,R} := e$                       ; Removes right side groups too small for the block
14. If $i_L = i_R$ stop                      ; No non-white groups middle of $Q_{i_L}$ and $Q_{i_R}$ to handle
15. Set $i_M := i_l + 1$                     ; Variable $i_M$ iterates groups middle of $Q_{i_L}$ and $Q_{i_R}$ left to right
16. If $|Q_{i_M}| \geq b_j$ go to 19         ; Jump over since group is big enough for $b_j$
17. For all pixels $c_p$ in $Q_{i_M}$        ; Colour were pixel $c_p$ is not in other block range
18.     if $b_{j-1,R} < p < b_{j+1,L}$ set $c_p :=$ white
19. Set $i_M := i_M + 1$                     ; Increment the iterator
20. If $i_M < i_R$ go to 16                  Loop until $i_M$ gets to most right group which $b_j$ fits
```

Rule 3.3-1 is for situation when leftmost pixel of the block b_j range is black and block left

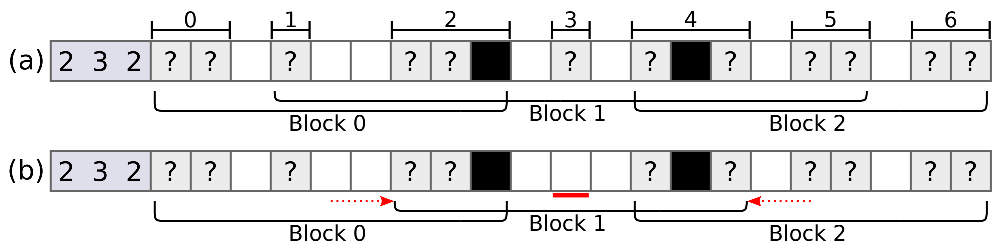


Figure 19. Example of rule 3.2. (a) the line before the rule. (b) the line after the rule. Note the red markers.

of b_j and b_j do not overlap. Following applies in this situation:

$$\begin{aligned}
&\forall p \in \{b_{j,L}+1,...,b_{j,L}+b_j-1\} \left(c_p := \text{black}\right)\\
&b_{j,R} := b_{j,L}+b_j-1\\
&b_{j,R}-1 \geq b_{j+1,L} \Rightarrow b_{j+1,L} := b_{j+1,R}+2\\
&b_{j-1,R} = b_{j,l}-1 \Rightarrow b_{j-1,R} := b_{j,L}-2\\
&c_{b_{j,L}-1} := \text{white}\\
&c_{b_{j,L}+b_j} := \text{white.}
\end{aligned} \tag{23}$$

Figure 20 shows how rule 3.3-1 works. Subfigure (a) is before rule 3.3-1 was used. Subfigure (b) is the line after Equation 23's black pixel colouring. Pixels 7 and 8 can be coloured black as the leftmost pixel in block 1 range is black meaning the rest of the pixels in the block must be right of it. Subfigure (c) has range updates of the blocks according to Equation 23. As block 1 pixels are known, block 1 range is updated to those pixels meaning that $b_{1,L}$ is reduced as $b_{1,R}$ is already in place. Since block 2 range overlaps with new block 2 range, $b_{2,R}$ can be moved right leaving space for a white pixel. Block 0 range is also updated to put space for a white pixel. The finally sentences in Equation 23 marks pixels 5 and 9 to be white as they are not in any block range.

Rule 3.3-2 is case where additionally there is black pixel c_k which is then followed by white pixel c_w ($k < w$) in b_j's range. The deduction here is that b_j should not need to reach the other side of the white pixel, since c_k is either part of it or b_{j+1}. As b_{j-1} does not overlap with b_j it can be ignored to have anything to do with c_k. Blocks have to be in order (Equations 7 and 8) so $b_{j,R} = w-1$.

Figure 21 shows practical example of the rule. Subfigure (a) is a line before rule 3.3-2 was applied. The line's zeroth block is solved making block 0 range and 1 range not overlap. Pixel 9 (black) and pixel 12 (white) are covered of block 1 range and block 2 range. No deduction can be made to tell, if pixel 9 is part of block 1 or 2. However, with rule 3.3-2

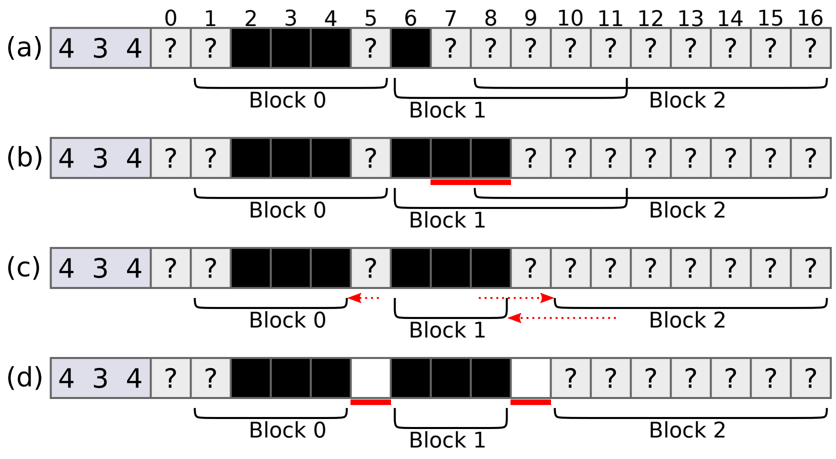


Figure 20. Example of rule 3.3-1. (a) the line before the rule. (b) first part colouring additional black pixels. (c) updating block ranges. (d) marking pixels outside of ranges to white.

deduction can be made because the order of blocks. If pixel 9 is part of block 2, then block 1 range right side ($b_{1,R}$) has to be less than 9. On the other hand, if pixel 9 is part of block 1 then pixel 11 is the rightmost pixel possible for block 1. Otherwise, block 1 would have white pixel in the middle of it, cutting it to two blocks. According to rule 3.3-2 block 1 range rightmost side is reduced to pixel 11.

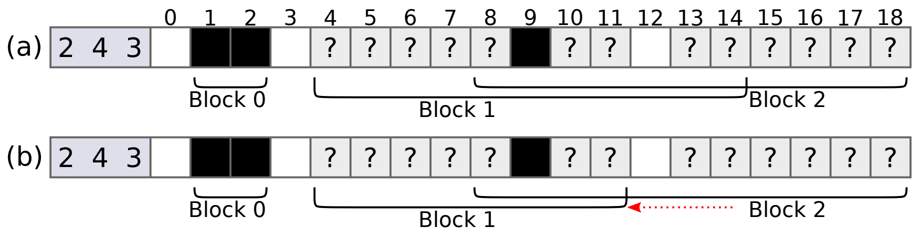


Figure 21. Example of rule 3.3-2. (a) is the line before. (b) is the line after rule 3.3-2 was applied.

Rule 3.3-3 checks that if there is black pixel groups ($Q := \{Q_i\}_{i=0}^{|Q|-1}$) in the b_j range so that they are not too far away from each other to make block b_j with correct length. Block b_j range can be reduced if groups are too far from each other.

Algorithm 5 shows how rule 3.3-3 is done. Figure 22 shows how rule 3.3-3 works. Rule 3.3-3 is applied to block 1 in the example. Subfigure (a) is a line before rule 3.3-3 is applied to the line. Block 1 does not overlap with block 0 and several pixels are known to be black. In Block 1 range there is three black pixel groups. Black pixel 4 is indexed as group 0. Pixel 6 is black indexed as group 1. Pixels 9 and 10 are black and are indexed as group 2. Pixel 1 (black) is **not** indexed as it is **not** in block 1 range. In block 1 range are black

Algorithm 5 Rule 3.3-3

Input: Line of pixels c_i, line length n, block b_j, finite sequence black pixel groups Q.

Output: Refined block b_j.

```
1. Find the first black pixel $c_f$ in $Q_0$          ; Variable $f$ tells where block starts
2. Set $i := 1$                                       ; Variable $i$ is iterator for $Q$
3. If $i = |Q|$ stop                                  ; Stopping here means that everything fits to $b_j$
4. Find first black pixel $c_o$ and last black pixel $c_u$ in $Q_i$
5. If $u - f + 1 > b_j$ go to 8                       ; Check that $b_j$ is too short to be between $Q_0$ and $Q_m$
6. Set $i := i + 1$                                   ; Increment the iterator
7. Go to 3                                            ; Make another iteration
8. Set $b_{j,R} := o - 2$                             ; update the $b_j$ range to remove groups which make $b_j$ too long
```

pixel groups which are visible in Subfigure (b) and (c) as blue numbers on top of the line. Algorithm 5 starts with finding the first pixel of black pixel group 0 in block 1 range. This is pixel 4, hence $f = 4$. Then the algorithm sets variable i to 1. The next steps find the first and last black pixel of Q_i (that is group 1). Both the first and last pixel is pixel 6 ($o = u = 6$). Subfigure (b) shows the next step. The step is to check that number of black pixels for block 1, if unknown pixels between the black pixel groups would be black, is lower than length of block 1 (b_2). With current values $u - f + 1 = 6 - 4 + 1 = 3$ which is less than blocks 1 length of 4. This means that the algorithm's "go to" does not happen. Rather, i is incremented to 2 and algorithm jumps back to stopping check 3. Stopping check makes sure that the algorithm stops if all black pixels groups in block 1 range are processed. If there are still unprocessed black pixel groups, new i value is used to search the first and last black pixel of the group i. First black pixel is 9 ($o = 9$) and last black pixel is 10 ($u = 10$). The length check is done again illustrated by Subfigure (c). This time $u - f + 1 = 10 - 4 + 1 = 7$ which is more than length of block 1 ($b_1 = 4$). Hence, go to of step 5 does happen. The algorithm jumps to block 1 range refining (step 8). As block 2 would be too large if pixels 9 and 10 are part of the block, right side $b_{1,R}$ can be reduced to omit pixels 9 and 10. To do this, and count for a white pixel to separate block 1 from block 2, first pixel c_o of the group i is used as a reference point and two is reduced to make space for a white pixel. The finished line is shown at Subfigure (d) with red arrows indicating range reduction.

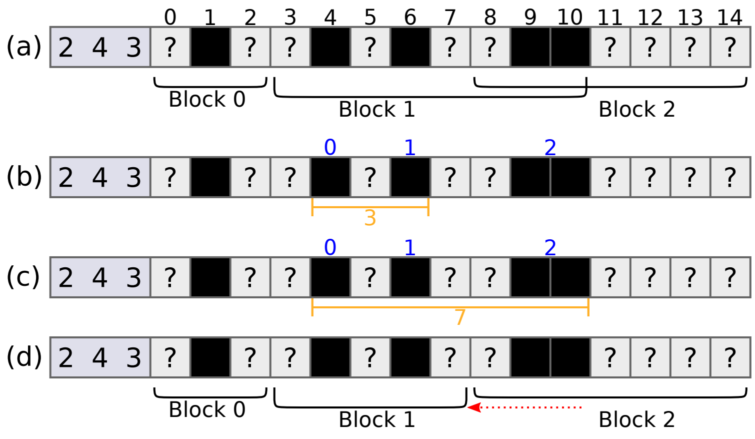


Figure 22. Example of rule 3.3-3. (a) the line before (b) first length check between groups (c) second length check between groups (d) line after the rule is applied.

3.2 Partial solver implementation detail

Implemented partial solver Algorithm 6 is similar to Algorithm 2 at start of Chapter 3. First loop initializes block ranges according to Equation 16 then applies rule 0.0 since it is only needed to be run once. Finally, first loop runs rule 1.1 to get some possible initial black pixel to the solution. Set H is the set of logical rules excluding initialization and rule 0.0

$$H = \{\boldsymbol{rule}1.1, \boldsymbol{rule}1.2, \boldsymbol{rule}1.3, \boldsymbol{rule}1.4, \boldsymbol{rule}1.5, \boldsymbol{rule}2.1 \\ , \boldsymbol{rule}2.2, \boldsymbol{rule}2.3, \boldsymbol{rule}2.4, \boldsymbol{rule}2.5, \boldsymbol{rule}3.1, \boldsymbol{rule}3.2, \boldsymbol{rule}3.3\}. \quad (24)$$

Rule implementations take in partial solution and the line, and return tuple$(T, e, u)$where T is the possible updated partial solution, e is a boolean to indicate error, and u for number of updates rule performed. Update counter adds up all updates done in one “do-while” loop. By resetting the update counter every “do-while” loop, “do-while” is stopped when no update is made by any rule in H.

There are two hypothetical problems which have to be addressed. First problem is that some rule is recolouring a pixel. This is addressed by error detection. Second problem is block ranges swapping. This is addressed with function ***NonoCheckBlockRangeSwap***.

There is a possibility, that at the crossings of two lines, the rule applied from the other directing causes recolouring. Forced example of this is show in Figure 23. Nonogram is first empty as shown in Subfigure (a). Subfigure (b) is after Algorithms 6 initialization (steps 2 to 8) for columns. Since zeroth and second column are static lines and first column is zero, the whole nonogram is coloured. Now in Subfigure (c) Rule 1.1 is applied to the lowest row.

Algorithm 6 NonoPartialSolver

```
Input: Nonogram $N$
Output: Pair proposed partial solution and a boolean indicating an error $(T,e)$
 1. Allocate proposed partial solution $T$ for $N$
 2. for each row and column $line$ in $N$
 3.     $(T,e,u) := \textbf{blocksRangeInit}(line,T)$
 4.     if $e$ is true return $(T,e)$                        ; Initialization rule at Equation 16
 5.     $(T,e,u) := \textbf{rule0.0}(line,T)$
 6.     if $e$ is true return $(T,e)$                        ; This only needs to be done once
 7.     $(T,e,u) := \textbf{rule1.1}(line,T)$                ; To get possible initial pixels
 8.     if $e$ is truereturn $(T,e)$
 9. do
10.     $update := 0$
11.     for each row and column $line$ in $N$
12.         for each $\textbf{rule} \in H$                  ; Rules returns amount updates it did
13.             $(T,e,u) := \textbf{rule}(line,T)$
14.             if $e$ is true then return T,e              ; Something was recoloured
15.             $update := u + update$
16. while $update > 0$                                      ; If updates were done go for another round
17. if $\textbf{NonoCheckBlockRangeSwap}(T)$ is true
18.             return $(T,\texttt{true})$                  ; Error check for block range swap
19. return $(T,e)$
```

This would recolour the middle pixel to black. This goes against first column to be all white. There is conflict on what value pixel marked red should be.

Since recolouring is an error, rules which colour pixel are given the ability to detect this. Hence, it is assumed that output from partial solver does not have this.

An other error that hypothetically can happen is one in which block range swaps. Block

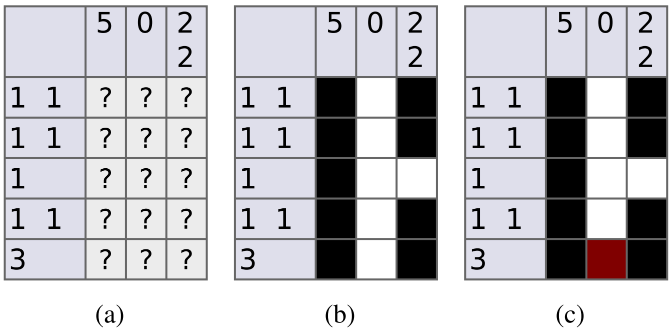


Figure 23. Example of recolouring. (a) Example nonogram empty. (b) example nonogram after columns where initialized. (c) lowest row has to repainted a pixel (marked red) white to black.

range swap means that $b_{j,R} < b_{j,L}$ for some block b_j. However, because rules 1.1 and 1.2, it is not clear, if this can ever happen without recolouring. On the other hand, hypothetically rule 2.1 can reduce block ranges a lot. To get rule 2.1 trigger condition is hard to imagine, and does not guarantee recolouring would not happen later. Regardless, it is a clear violation of Equations 3 or 4. It could be proven that recolouring does handle block range error but for time limitations/scope simple algorithm for checking partial solution was added instead. This is shown as Algorithm 7. Algorithm goes through every block checking that no swap has happened.

Algorithm 7 NonoCheckBlockRangeSwap

Input: Nonogram N, Proposed Partial solution T

Output: true if there is block's range swap, false if there is not.

```
1. for each row and column $line$ in $T$
2.     for each block $b_j$ in $line$
3.         if $b_{j,L} > b_{j,R}$                    ; check for swap
4.             return true                          ; swap was found
5. return false
```

While testing the partial solver (Algorithm 6) on known one solution nonograms, it was noticed that lines with the description and state (colour configuration and block ranges) shown in Figure 24 could not be deduced further, even thought deduction does exist. Subfigure (a) is state of the line before deduction. Block ranges overlap with each other rather much. From the pixels, which have known values, deduction can be made, that pixel 20 belongs ether block 2 or 3. Deduction that can be made here, is that leftmost pixel of block 3 can only be pixel 20 as the block 4, the size of which is 2, can not fit with pixel 20. Hence, block 3 cannot be further left, and if pixel 20 belongs to block 2, then pixel 22 or 23 belongs block 3. After block 3 range is reduced, block 4 range is also reduced by rule 2.1 to pixel 22. This state is shown as Subfigure (b).

Further deduction can be made by pixel 13 being black. If Block 2 range rightmost is less than 13, then no block explains pixel 13 being black. Hence, Block 2 leftmost has to be pixel 13. This is show as Subfigure (c).

No rule makes deduction for block 3 range reduction. Rule 2.4 can deduce block 3 and 4 order but reducing block 2 range to be right of pixel 9 does not happen. Block ranges overlap just fine to not trigger rule 2.1. Rule 2.2 does not trigger, because block 3 covers the only black pixels. Rule 2.3 triggers only if black pixel group is bigger than block itself, which is not case here for any black pixel group. Rule 2.5 does not trigger, as there is plenty of space for leftmost arrangement. Pixel 20 is covered by three block ranges, so rule 3.1 does not trigger. Rule 3.2 does not trigger as it only looks at one block at the time and does not understand block order, and rules 3.3 need non-overlapping block ranges, which do not exist in the line. Rule for this deduction is not programmed as it is too much out of scope of the Thesis.

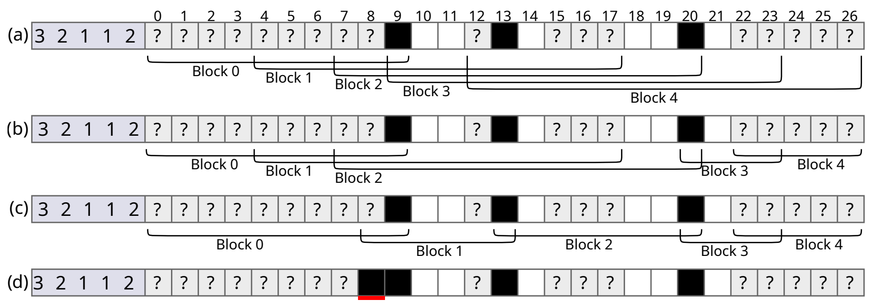


Figure 24. Deduction which rules did not make. (a) is state of the line before deduction. (b) is deduction of block ranges. (c) is final deduction of black pixel.

4 PARTIAL SOLUTION POST-PROCESSING

After partial solver has run, Algorithm 1 process more information out of the results. These steps are verification of the result, and finding possible switching components if result was partial solution. Verification Algorithm 9 in Section 4.1 is used. This algorithm is build on top of NONOGRAM's solution verifier.

Internals of Subalgorithm ***FindSwitchingComponent*** are shown in Algorithm 8. It is made of two parts. Algorithm which produces an unknown pixel graph (Algorithm 11) and Algorithm to "find the connected components of a graph" (***FindGraphComponents***). Idea is to make a graph where each vertex is an unknown pixel of the Partial solution T such that unknown pixels, which directly effect each others colours have an edge between them. Since switching components by definition do not affect each other, then switching components are components of the unknown pixel graph. Set of switching components S is then internally a set of components of an unknown pixel graph.

More on unknown pixel graph in Section 4.2. Function ***FindGraphComponents*** is not presented in this Thesis as it is well known algorithm from literature. In [16] is explanation of it and proof that complexity of algorithm is $\boldsymbol{O}(\boldsymbol{max}(|V|,|E|))$.

Algorithm 8 FindSwitchingComponent

```
Input: Nonogram $N$, Solution $T$
Output: Set of switching components $S$.
  1. $(V,E) := \textbf{NonoProducePixelGraph}(N,T)$                ; Algorithm 11
  2. $S := \textbf{FindGraphComponents}((V,E))$                    Algorithm from [16]
  3. return $S$
```

4.1 Partial solution verification

Since NONOGRAM is a **NP** problem, for it exists a polynomial time solution verifier. This algorithm can be determent from the Equations 1 to 14. However, solution verifier does not handle unknown pixels as it is designed for checking full solution of given nonogram. Algorithm 9 add navy handling of unknown pixels to verifier. Point is to identify if partial solver produces a partial solution, a solution, or that nonogram cannot have any solution. Algorithm 9 takes in a nonogram N and proposed partial solution T. Output is enumerator value on partial solution, a solution, or no solution possibilities.

Algorithm 9 starts with variable *value* set to `isSolution`. Value of the variable will be returned to the caller of the Algorithm at the end. It's value may change to indicate a partial solution if unknown pixels is found (steps 8 and 15). If line has unknown pixel, then processing of that line is stopped.

Since Equations 1 to 14 are per line Algorithm 9 also can just look at one line at the time. Hence, step 2 starts "for each" loop per line in N and T. Meaning of "in N and T" is just because information of the line is divided between N and T. So "for each" iteration variable line hold information from both N (description for that particular line) and T (pixels and block ranges for that particular line).

There is two inner loops in the "for each" loop for lines. First one of these checks per block line has first white pixels and then block amount of black pixels. Second inner loop checks that after blocks have been found there is only white pixels (as Equations 13 and 14 want). Pixels which are unknown, are not accepted in the second inner loop since there should not be a block for black pixels and something like rule 1.2 should colour these pixels white anyway.

First loop has two inner inner loops. First inner inner loop (steps between 5 to 10) moves the x over white pixels left of block b_j. This includes white pixels before first block (as Equations 1 and 2 want) and white pixels between the blocks (as Equations 11 and 12 want). When black is found at first inner inner loop processing is moved to second inner inner loop. Second inner inner loop checks that block amount of black pixels exist. This is requirement of Equations 3 and 4. Check is done by counting number of black pixels to variable *count* until white pixels is found. When white pixel is found processing is moved on a step 19 which checks that block amount of black pixels was counted.

Example how this rejects in valid proposed partial solutions. Algorithm 9 indexes pixels in the line with variable x which is used in both inner loops. In the first inner loop if it happens that descriptions first blocks take too much space (by leaving too much empty space between them) then variable x will hit length of the line (*line.length*). This causes inner inner loops not to run which means that when current block or next block is in processing counting for that block does not happen. This then causes step 19 return `isInvalid`.

Other interesting example is zero description. In this case, the first inner inner loop increments *x* to *line.length*. This causes second inner inner loop to be jumped over but because variable *count* was set to zero check at step 19 does not trigger processing continues to next line.

Algorithm 9 could be improved. Currently, block's ranges are not considered at all. If Algorithm 9 would consider block range fully coloured blocks which are after unknown pixels could be checked. For our purposes however, Algorithm 9 is enough.

Algorithm 9 NonoVerifier

Input: Nonogram N, proposed partial solution T

Output: answer is T is a solution, partial solution, or not valid.

```
1.  value := isSolution
2.  for each row and column line in N and T
3.      x := 0                                             ; line's pixel index
4.      for each block b in the line.description
5.          while x < line.length          ; go through white pixels between black pixels
6.              if c_x for line is black then break  ; start of a block found so go to next loop
7.              else if c_x for line is unknown
8.                  value := isPartialSolution                  ; mark solution as partial
9.                  continue at step 2        ; line is not fully coloured so go to next line
10.             x := x + 1
11.         count := 0
12.         while x < line.length
13.             if c_x of line is black then count := count + 1        ; counting black pixels
14.             else if c_x of line is unknown
15.                 value := isPartialSolution
16.                 continue at step 2                          ; mark solution as partial
17.             else break                                 ; white pixel so stop counting
18.             x := x + 1
19.         if count ≠ b then return isInvalid             ; count must equal block length
20.     while x < line.length                        ; check for end is only white pixels.
21.         if c_x of line is black or unknown
22.             return isInvalid         ; remaining pixels should be expected to be white
23.         x := x + 1
24. return value
```

4.2 Unknown pixel graph

To represent switching components in the algorithms graph is used. This is because switching component can be very few unknown pixels from the total amount of pixels. In the literature [2] presented dependency graph for solving nonograms. Dependency graph is build via constructing 2SAT clauses from partial solution. These clauses are constructed by noting that if certain pixel is coloured it may imply other unknown pixels are coloured. These implications can be represented by a graph. Figure 25 shows example of this type of graph made from nonogram in Figure 4. In side of the node $p_{i,j}$ means statement: “pixel at column i and row j is black”. Negation of it means: “pixel at column i and row j is white”. If in the graph exists path from $p_{i,j}$ to $\neg p_{i,j}$ then choosing a black colour for pixel would lead to contradiction and no valid solution for the instance. Implication other way works as well. In Figure 25 implications go both directions however this does not need to happen.

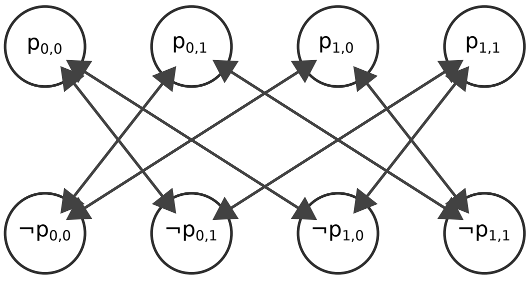


Figure 25. Dependency graph of ESC

Problem with dependency graph are:

- it does not handle block ranges at all,
- deriving 2SAT clauses in general is hard,
- having two vertexes per unknown pixel is excessive.

Better graph is to have unknown pixels connected to its nearest unknown pixels in the line. Additional information can put in to edges of the graph to speed up processing of getting the type information. Types of switching components are then dictated by how graph is structures and the additional information on the edges.

Reason edges are created along the lines of the nonogram is because nonograms validity equations (Equations 1 to 14) look at pixel along the lines. This means any consequence that comes from colouring a pixel are felt at the line as well. This is why it is possible to get separate switching components be top of each other. In figure 26 is and example of two switching component top of each other. In the “middle” is unknown pixels $c_{3,3}$, $c_{7,7}$, $c_{3,7}$, and $c_{7,3}$ form a switching component (an ESC to be exact). Outer edge and centre pixel is separate from the switching component of unknown pixels $c_{1,1}$, $c_{5,1}$, $c_{9,1}$, $c_{1,5}$, $c_{1,9}$, $c_{5,5}$, $c_{5,9}$, $c_{9,5}$, and $c_{9,9}$. This can be seen if one enumerated all 12 solutions for this nonogram. Switching component finding algorithm can separate these out when it creates edges along the lines.

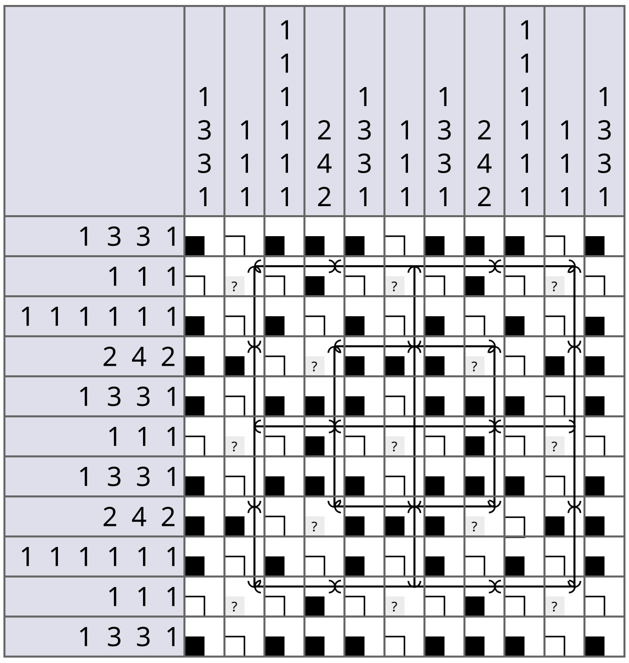


Figure 26. Example of switching component overlapping with other switching component.

There is a state where even when unknown pixels along the line should not have edge between them. Back at 1 Figure 4 showed such a case. If unknown pixels c_{x_0,y_0} and c_{x_1,y_0} are next in line, but they are covered by different blocks which do not overlap in block ranges, then colouring one of them does not cause colour of other (unless there is an other unknown pixel which through implication comes).

Figure 27 shows example case where edge should not be created between unknown pixels. Subfigure (a) gives initial state. There is two unknown pixels c_{x_0,y_0} and c_{x_0,y_1}. There is two blocks b_{j_0} covering c_{x_0,y_0} and b_{j_0+1} covering c_{x_0,y_1}. Blocks do not overlap. In Subfigure (b) is a case when unknown pixel c_{x_0,y_0} is coloured black. Immediate result of this is shown in Subfigure (c). Other unknown pixel left of the c_{x_0,y_0} will be coloured to black or white depend upon length of the block b_{j_0}. Pixel c_{x_0,y_1} will not be effected. In Subfigure (d) unknown pixel c_{x_0,y_0} is coloured to white. In Subfigure (e) result of this is shown. Block b_{j_0} range is refined to left and depend upon length of b_{j_0} other unknown pixel left of is coloured ether black or white. Nothing happens to colour of c_{x_0,y_1}. In Subfigure (f) unknown pixel c_{x_0,y_1} is coloured white. Result of this, shown in Subfigure (g), is that block b_{j_0+1} is refined out of the segment of the line we are looking at. Nothing happens to unknown pixel c_{x_0,y_0}. In Subfigure (h) unknown pixel c_{x_0,y_1} is coloured to black. In Subfigure (i) is the results. Since block b_{j_0+1} is only block which can explain black pixel c_{x_0,y_1}, block b_{j_0+1} is reduce to it. Nothing is done for unknown pixel c_{x_0,y_0}.

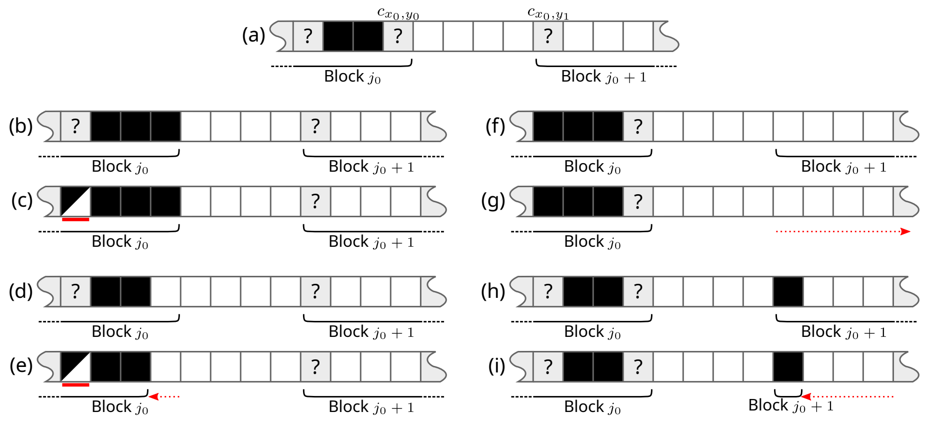


Figure 27. Example where two unknown pixels do not effect each other. (a) is initial state of the line segment. (b) is state after colouring c_{x_0,y_0} to black. Subfigure (c) is consequence of colouring c_{x_0,y_0} to black. (d) is state after colouring c_{x_0,y_0} to white. Subfigure (e) is consequence of colouring c_{x_0,y_0} to white. (f) is state after colouring c_{x_0,y_1} to white. (g) is consequence of colouring c_{x_0,y_1} to white. Subfigure (h) is state after colouring c_{x_0,y_1} to black. (i) is consequence of colouring c_{x_0,y_1} to black.

Additional information to edges should tell about pixels between the unknown pixels. These edges can be typed to four categories. Edge type is referred to as an edge state since there is simple state machine to deduce correct one. Figure 28 shows state diagram of those states. When leaving, unknown pixel (*lastunknown*) starting state is `nextto` meaning next unknown pixel is right next to last one. If next pixel read is white, then state is changed to `fullwhite`. This state correspondence to edge which all the in between pixels are white. If next pixel read from `nextto` is black, then state is changed to `fullblack` which correspondence to edge which in between pixels are all black. State in both `fullwhite` and `fullblack` keeps the state on itself if pixel continues to be same colour but if colour changes then state is changed to `mix`. State `mix` means there is mixture of black and white pixels in the between. If following pixels are black or white state `mix` will keep itself on that state. From every state (`nextto`, `fullwhite`, `fullblack`, and `mix`) if next pixel is an unknown pixel which means new vertex in the unknown pixel graph then type of the edge is the last state the machine was in.

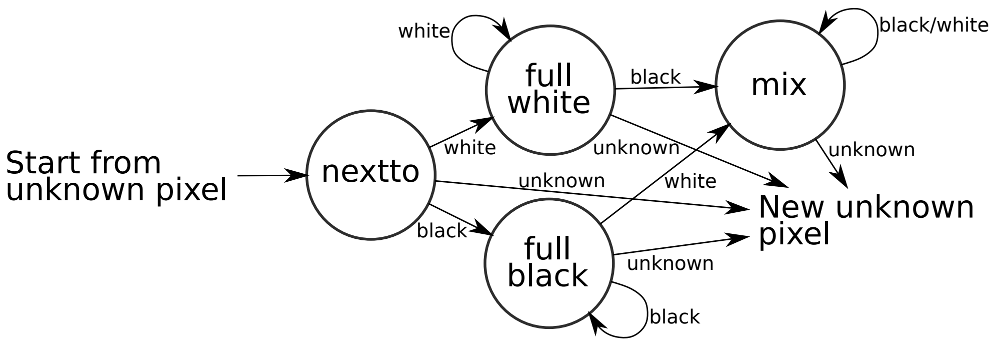


Figure 28. Edge state diagram.

These four where selected because each of them does have an implication cross the edge. Figures 29, 30, 31, and 32 show example of each edge in line segment and implication of colouring the unknown pixel groups of the edge.

Figure 29 shows how `nextto` edge works. Subfigure (a) is a segment of line relevant to implication. No other block covers unknown pixel groups Q_{u_L} or Q_{u_R}. Subfigures (b), (c), and (d) show how implication works if Q_{u_L} is coloured black. Subfigures (e), (f), and (g) show same implication steps if Q_{u_R} is coloured black. In Subfigure (b) and (e) respected groups are coloured. In Subfigures (c) and (f) block b_{j0} range is reduced to only cover the black pixel group. Reasoning for this is that avoid violations of 3 and 4 or application of Rules 3.3-1 and/or 2.4. Subfigures (d) and (g) shows colouring of the unknown pixel group left outside of block range to white as what is pixel are not covered by any block (Equations 11 and 12 or rule 1.2).

Figure 30 shows a `fullblack` edge with colouring example. In Subfigure (a) is the relevant segment's initialize state on the line. Block b_{j0} has length of 11 pixels and it covers whole segment of the line. In the top of each Subfigure is markings for each pixel group in yellow. No other block covers unknown pixel groups Q_{u_L} or Q_{u_R}. Subfigure (b) and (c) show implication where unknown pixel group is coloured black. In Subfigure (b) unknown pixel group Q_{u_L} is coloured black and block b_{j0} range is reduced to cover only 11 black pixels to avoid violations of 3 and 4 or by rule 3.3-1 and/or 2.4. In Subfigure (c) rule 1.2 is applied to the line colouring Q_{u_R} to white. Subfigures (d) and (e) show implication when unknown pixel group Q_{u_L} is coloured white. Subfigure (d) shows the colouring of Q_{u_L} and block b_{j0} range reduction done by rules 2.5. Subfigure (e) is colouring of Q_{u_L} to black by rule 3.3-1.

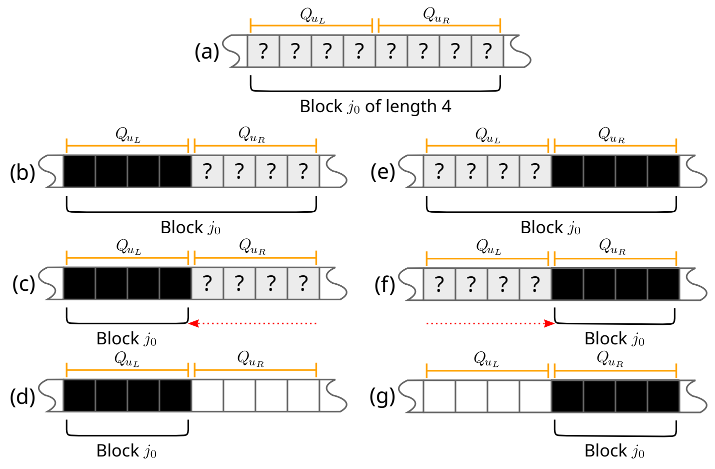


Figure 29. Shows an example of `nextto` where block is length of 4. (a) is initial state of segment of line. Subfigures (b), (c), and (d) shows implication step by step if Q_{u_L} is coloured black and Subfigures (e), (f), and (g) shows implication step by step if Q_{u_R} is coloured black.

Figure 31 show example of `fullwhite` edge and the deduction when colouring Q_{u_L} to black. Subfigure (a) is the initial state of the relevant segment of line. Subfigure (b) colouring of Q_{u_L} to black is shown. Subfigure (c) is the reduction of the block b_{j_0} range to black pixels by rule 3.3-1 (Equation 23) which form the block b_{j_0}. Subfigure (d) is the colouring of the pixel group Q_{u_R} to white as it is no longer covered by any block range (Rule 1.2).

Figure 32 show a `mix` edge of length 4 and a consequence when Q_{u_R} is coloured black. In Subfigure (a) is the initial state before colouring Q_{u_R} black. No other block covers unknown pixel groups Q_{u_L} or Q_{u_R}. In Subfigure (b) Q_{u_R} is coloured black. In Subfigure (c), to avoid violations of 3 and 4 or rule 3.3-1 and/or 2.4 is used to refine block b_{j_0+3} to Q_{u_R}. After that rule, 2.4 can be used again to refine blocks b_{j_0+2}, b_{j_0+1}, and b_{j_0}. This is shown in Subfigures (d) to (f). In Subfigure (g) unknown pixel group Q_{u_L} is without block range hence it is coloured to white by rule 1.2.

Additional information given to edge value depends upon what edge state is outputted. For `nextto` and `fullwhite` no additional information is needed. For state `nextto` this is because there are no pixels to make information out of. For state `fullwhite` this is because only interesting matter is to detect if block has changed, in which point no edge should be created between the two unknown pixels. For state `fullblack` additional information is number of pixels between the unknown pixels. In Figure 30 unknown pixel group is used with black pixel group to create the block. To make checking calculation for this easier, length of black pixel group is basically cached for every `fullblack` edge. For state `mix` additional information is triplet of: length of the biggest black group in the `mix` edge, length of the latest black group or zero (used to calculate the biggest black group), and number of

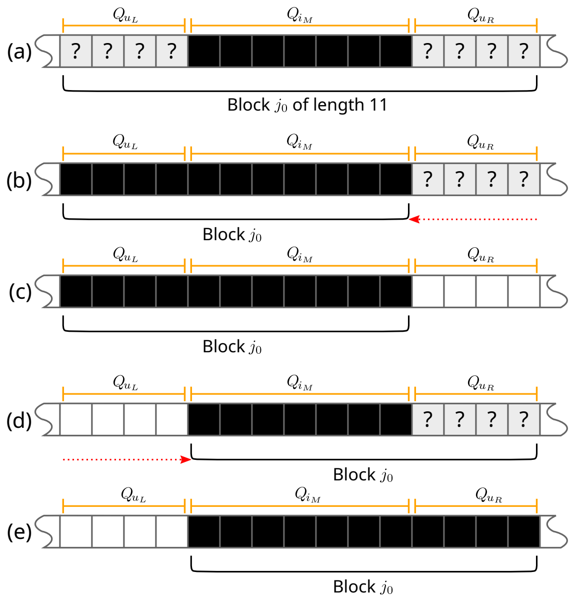


Figure 30. Shows example of `fullblack`edge with colouring example on a segment of the line. (a) is the initialize state, (b) and (c) are the deductions if Q_{u_L} is coloured black, and (d) and (e) if Q_{u_L} is coloured white.

black pixel groups in the `mix` edge.

Algorithm 10 updates the edge state according to Figure 28 and keeps track of the additional information. Algorithm 10 takes in previous edge state ($state$), previous additional information ($stateinfo$), and new non-unknown pixel colour c. The algorithm first asks: is c black or white, then it asks what edge state was give. Variable $stateinfo$ is an integer after state is moved to `fullblack` and triplet on `mix`. Before moving in these states $stateinfo$ is assumed to be `null`. When changing edge state from `nextto` to `fullblack` variable $stateinfo$ is set to 1 (step 4) because c is already first black pixel. Similarly, reason why `fullwhite` change to `mix` variable $stateinfo$ triplet are set to all one (step 8). When changing edge state from `fullblack` to `mix`, previously collected information about number of pixels in the $stateinfo$ is used to initialize the largest black pixel group (step 17). There needs to be a system which updates stateinfo in `mix` state. When in `mix` state and c is a black pixel, then size of the current block group's size is incremented and check is made if group is now the largest detected (steps 10 to 12). If c would be white then determination is made if last pixel was black (by asking: is black group length counter more than zero at step 19) and then group count is incremented if so. Variable inc is used to in step 21 to increment group count by one or zero. Step 21 also reset black pixel group length count to zero for next black pixel group.

Algorithm 11 produces the unknown pixel graph. In the algorithm, unknown pixel graph is expressed as standard pair of two sets. Set V for vertices of the graph and set E for the

Algorithm 10 UpdateEdgeState

Input: previous edge state *state* and additional information *stateinfo*, pixel colour *c*.

Output: updated *state* and *stateinfo*

```
1.  if $c =$ black
2.      if $state =$ nextto
3.          $state :=$ fullblack                                ; Change state.
4.          $stateinfo := 1$
5.      else if $state =$ fullblackthen $stateinfo := stateinfo + 1$
6.      else if $state =$ fullwhite
7.          $state :=$ mix                                      ; Change state.
8.          $stateinfo := (1,1,1)$                              ; Change to triplet.
9.      else if $state =$ mix
10.         $stateinfo := stateinfo + (0,1,0)$                  ; element-wise addition.
11.         if $stateinfo_0 < stateinfo_1$    ; is zeroth element smaller than first element.
12.             $stateinfo := (stateinfo_1, stateinfo_1, stateinfo_2)$
13. else if $c =$ white
14.     if $state =$ nexttothen $state :=$ fullwhite            ; Change state.
15.     else if $state =$ fullblack
16.         $state :=$ mix                                      ; Change state.
17.         $stateinfo = (stateinfo, 0, 1)$                     ; Change to tuple.
18.     else if $state =$ mix
19.         if $stateinfo_1 > 0$ then $inc := 1$    ; Should black pixel groups count be incremented.
20.         else $inc := 0$
21.         $stateinfo := (stateinfo_0, 0, stateinfo_2 + inc)$  ; Reset the count.
22. return $(state, stateinfo)$
```

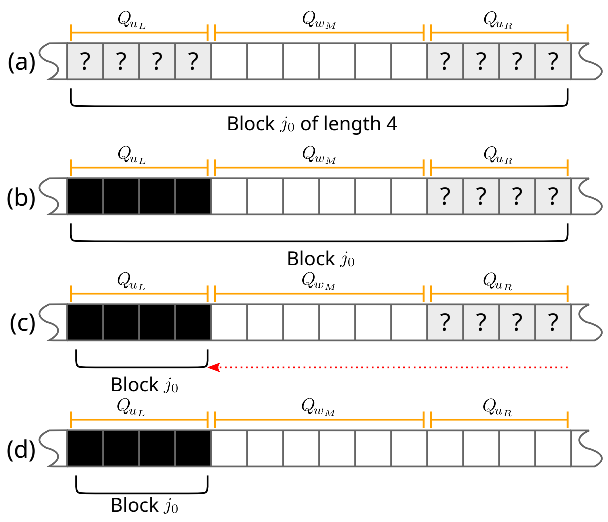


Figure 31. Shows an example of `fullwhite` edge where block is length of 4. Subfigure (a) is the segment of the line initially, (b) is colouring of pixel group Q_{u_L} to black, (c) is block range reduction to black pixels, and (c) is colouring of pixel group Q_{u_R} to white.

(undirectional) edges which are pairs of vertices of V. Algorithm 11 rows and columns are handled in separate loops. Row loop is steps 1 to 26 and column loop is steps 27 to 51. Both loops differ only that row loop creates a vertex (steps 7 and 8) and column loops finds a vertex (step 33).

Variable *lastunknown* is used to store previous loop's iterations vertex. Variable *lastunknown* is set to `null` for first iteration of the line because there is no previous unknown pixel to create edge to. Vertex creation and found is done regardless to store correct value to variable *lastunknown*.

Two variables are store edge value information in Algorithm 11. Variable *state* for edge state and variable *stateinfo* for the additional information. Variable *state* is initialized to `nextto` as it is starting point of the state machine and variable *stateinfo* is initialized to `null` for Algorithm 10. Inner loop goes through every pixel in the line (steps 5 to 24). In both loops if inner loop going through the pixels does not found an unknown pixels and variable *lastunknown* is not null Algorithm 10 (steps 25 and 50). Variable *lastunknown* check is done to avoid running it when it is not needed.

Algorithm 11 avoids creating an edge between unknown pixels where they are not covered by same block or do not have `amix` edge between them. Check is done in both inner loops steps 10 to 18 and steps 35 to 43. For the `mix` edge sanity check is done to make sure there is no overlapping chain of block ranges before skipping the edge creation. If there is chain of overlapping block ranges, then most likely then edge is created. For `nextto`, edge is always

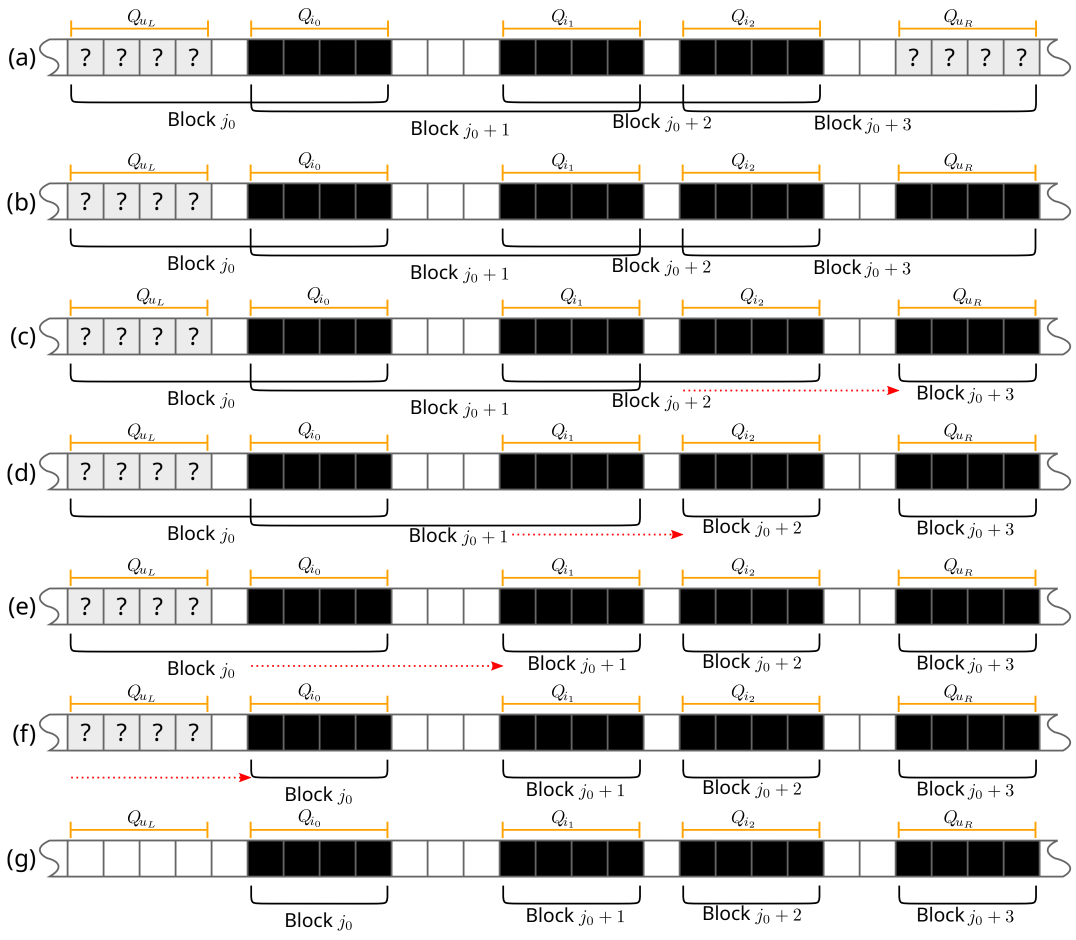


Figure 32. Shows an example of edge `mix`. (a) is state of relevant line segment initially, (b) is colouring of Q_{u_R} to black, (c) to (f) are chaining block's range reductions, and (g) .

created. For `fullblack` and `fullwhite` edge if only one block covers the found unknown pixel then check is performed to check that block range does include *lastunknown*. If it does not, then edge should not be created.

Algorithm 11 NonoProducePixelGraph

Input: Nonogram N, Partial solution T

Output: Unknown pixel graph (V,E)

```
 1. for each row $y$ in the $T$
 2.     $lastunknown$ := null
 3.     $state$ := nextto          ; edge type. Default state is next pixel being unknown
 4.     $stateinfo$ := null        ; information associated with some edge states
 5.     for each column $x$ in the $T$
 6.         if $c_{x,y}$ = unknown          ; Add a vertex for unknown pixel
 7.             create new (labelled) vertex $v_{x,y}$
 8.             add vertex $v_{x,y}$ to $V$
 9.             if $lastunknown \neq$ null
10.                 find blocks $\{b_{c,y,j}\}_j$ which covers $c_{x,y}$          ; Algorithm 3
11.                 set $j_0$ leftmost index of $\{b_{c,y,j}\}_j$
12.                 if $state =$ mix          ; different check for mix edge
13.                     set $b_{last}$ to right most block covering $lastunknown$     ; Algorithm 3
14.                     if between $b_{last}$ and $b_{c,y,j_0}$ is non-overlapping blocks
15.                         goto 24          ; do not create edge
16.                 else if $\left|\{b_{c,y,j}\}_j\right| = 1 \wedge state \neq$ nextto
17.                     if $x_{lastunknown} < b_{c,y,j_0,L}$
18.                         goto 24          ; do not create edge
19.                 create new (undirectional) edge $(v_{x,y}, lastunknown)$
20.                 add edge $(v_{x,y}, lastunknown)$ to $E$          ; column directional edge
21.                 set value of the edge to $(type, typeinfo)$
22.                 $state$ := nextto          ; reset edge state
23.                 $stateinfo$ := null
24.             $lastunknown$ := $v_{x,y}$
25.         else if $lastunknown \neq$ null          ; else means that $c_{x,y}$ was not unknown
26.             update $state$ and $stateinfo$          ; Algorithm 10
```

```
27. for each column $x$ in the $T$
28.     $lastunknown := \texttt{null}$
29.     $state := \texttt{nextto}$          ; edge type. Default state is next pixel being unknown
30.     $stateinfo := \texttt{null}$          ; edge type sometimes stores extra info
31.     for each row $y$ in the $T$
32.         if $c_{x,y}$ is unknown          ; find existing vertex for unknown
33.             find vertex $v_{x,y}$ from $V$
34.             if $lastunknown \neq \texttt{null}$
35.                 find blocks $\{b_{r,x,j}\}_j$ which covers $c_{x,y}$          ; Algorithm 3
36.                 set $j_0$ leftmost index of $\{b_{r,x,j}\}$
37.                 if $state = \texttt{mix}$          ; different check for mix
38.                     set $b_{last}$ to rightmost block covering $lastunknown$          ; Algorithm 3
39.                     if between $b_{last}$ and $b_{r,x,j_0}$ is non-overlapping blocks
40.                         goto 49          ; do not create edge
41.                 else if $\left|\{b_{r,x,j}\}_j\right| = 1 \wedge state \neq \texttt{nextto}$
42.                     if $y_{lastunknown} < b_{r,x,j_0,L}$
43.                         goto 49          ; do not create edge
44.                 create new (undirectional) edge $(v_{x,y}, lastunknown)$
45.                 add edge $(v_{x,y}, lastunknown)$ to $E$          ; row directional edge
46.                 set value of the edge to $(state, stateinfo)$
47.                 $state := \texttt{nextto}$          ; Reset edge state for new edge
48.                 $stateinfo := \texttt{null}$
49.             $lastunknown := v_{x,y}$
50.         else if $lastunknown \neq \texttt{null}$          ; else means that $c_{x,y}$ was not unknown
51.             update $state$ and $stateinfo$          ; Algorithm 10
52. return $(V, E)$
```

5 ESTIMATION ALGORITHM AND TYPE SUBCASE

Idea to probabilistically approximate the number of colourings of a switching component needs a detector for the type of switching component and function to get the approximation for that type. Algorithm 12 does this if build-in set of switching component types Z is finite in size and complete in covering of types. Randomization for FPRAS algorithm can happen in the approximation phase. Randomization in detector phase would need some guarantee other type detector to pick up the approximation.

Every type in Z detector is called until type is detected or there is no more types to test. If switching component type z is detected variable $istype$ equals `true`. If this happens Algorithm 12 allows types approximator function to be called. Return value out of Algorithm 12 is estimated number of colourings for s and indicator was type found or not (`found` or `notfound`). When s is not a type in Z, estimated number of colourings returned is given as one. This is done so that counting back at caller (Algorithm 1) would not be effected.

Algorithm 12 EstimateSwitchComponent

Input: Nonogram N, Partial solution T, Switching component s .

Output: Pair $(count,found)$ where $count$ is the estimate for the number colourings switching component has and $found$ is ether `true` or `false` to indicated was s type in Z.

```
1. for each switching component type $z$ in $Z$
2.     Set $istype$ to answer of the detector for type $z$ when $S$ in $T$ of $N$ is given
3.     if $istype = \texttt{true}$                          ; If $istype$ is false go to next type.
4.         Set $count$ to result to type $z$ approximator when $S$ in $T$ of $N$ is given.
5.         return $(count, \texttt{true})$
6. return $(1, \texttt{false})$                               ; $s$ was not type in $Z$
```

This Thesis does not provide complete set of types for Z. In Chapter 7 challenge to provide such a set is discussed. Rest of this Chapter focus on a subcase of switching components which can be **exactly** counted if detected.

5.1 One-black colourable one-pixel SSC detector

There is a simple instance of #NONOGRAM that do not need the randomizing to estimate the solution count. One such, set of instance are SSCs. SSC is any "square" arrangement unknown pixels from the set

$$\begin{aligned}
&\Big\{\Big\{v_{x_i,y_j}\Big|\forall i,j \in \mathbb{N}_{\leq u-1}\left(\forall p \in \mathbb{N}_{\leq u-1}\left(x_{i,p} = x'_i \wedge y_{p,j} = y'_j\right)\right)\\
&\quad \wedge \forall i \in \mathbb{N}_{\leq u-2}, j \in \mathbb{N}_{\leq u-1}\left(\left(v_{x_i,y_j}, v_{x_{i+1},y_j}\right) \in E \wedge \left(v_{x_j,y_i}, v_{x_j,y_{i+1}}\right) \in E\right)\Big\}\Big|\\
&\quad \exists \left\{x'_i\right\}_{i=0}^{u-1} \subseteq \mathbb{N}_{\leq n-1}, \left\{y'_j\right\}_{j=0}^{u-1} \subseteq \mathbb{N}_{\leq m-1}\Big\}
\end{aligned} \tag{25}$$

for some u. Value u is the side length of the "square" in number of unknown pixels. For example, the three by three Nonogram of Figure 33 is a SSC with side length of 3. Partial solution in Subfigure (a) is show as unknown pixel graph of in Subfigure (b).

Equation 25 states two conditions needed to be SSC. First, every vertex in a line relevant to SSC must have u vertices. Second vertexes in the line must have path keeping to the line. This is stated in Equation 25 as each vertex $v_{x_i,y}$ which i is less than or equal to $u-2$ must have an edge to $v_{x_{i+1},y}$ and each vertex v_{x,y_j} which $y_j$is less than or equal to $u-2$ must have edge to $v_{x,y_{j+1}}$. Less then equal is $u-2$ because first indexing starts at zero. Second vertexes with the highest index number are at edge of a SSC and hence cannot have edge to higher index vertex.

In Subfigure (b) unknown pixel graph satisfies these two conditions for side length of 3. Every line has three vertexes. For example, Row 0 (relevant to SSC) has vertexes $v_{0,0}$, $v_{1,0}$, and $v_{2,0}$. Vertexes $v_{0,0}$, $v_{1,0}$, $v_{2,0}$, $v_{0,1}$, $v_{1,1}$, and $v_{2,1}$ have an edge to higher index vertex on the row and vertexes $v_{0,0}$, $v_{1,0}$, $v_{0,1}$, $v_{1,1}$, $v_{0,2}$, and $v_{1,2}$ have edge to higher index vertex on the column.

It does not matter to the definition of SSC what state the edge value is. In Subfigure (b) every edge is a `nextto` edge but in Figure 34 this is not the case. Subfigure (a) is the partial solution of the nonogram. Unknown pixels form a SSC of size 3. Subfigure (b) is unknown pixel graph of partial solution of Subfigure (a). Unknown pixel graph is the same as before but coordinates of the vertexes are different and some edges are not `nextto`.

Most of the time, it is easy to count **exactly** number of solution a SSC causes for a nonogram.

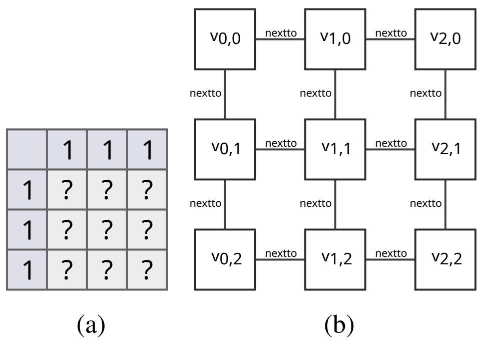


Figure 33. 3 by 3 Nonogram with SSC length of 3. (a) is the partial solution of the nonogram. (b)unknown pixel graph of (a).

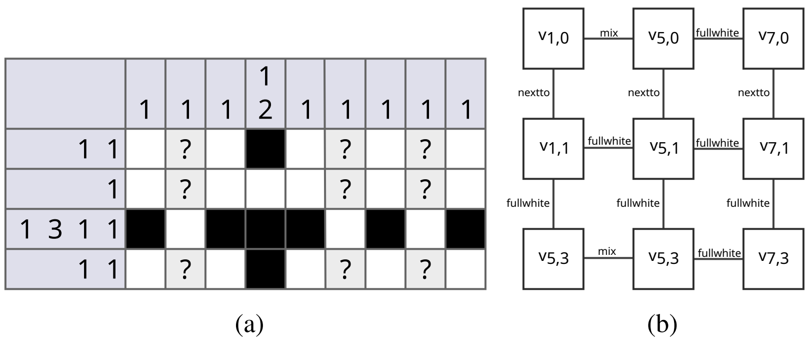


Figure 34. Partial solution 9 by 4 nonogram which has SSC and where unknown pixels are not next to each other. (a) is a partial solution of the Nonogram. where unknown pixels are not next to each other. (a) is the unknown pixel graph of the partial solution.

The exceptions from this in SSCs have an `fullblack` edge state and have size larger than 2. To focus on this easy **exactly** countable subset two additional properties are defined. First *p*-pixel, and then second one-black colourable.

Property *p*-pixel means that block to be coloured by colouring unknown pixels is size of *p* on every line relevant to switching component. Focus is on the one-pixel as scaling does not appear to effect number of solutions.

One-black colourable means that there is only one block not fully coloured in the line. When combined with one-pixel (one-black colourable one-pixel) property of colouring one unknown pixel on the line, then rest of the line's unknown pixels (relevant to switching component) will be coloured white. This property is in symbolic logic formulation for a line relevant to unknown pixels is:

$$\forall i \in \mathbb{N}_{\le u}\left(c_{x_i} = \text{black} \Rightarrow \forall j \in \mathbb{N}_{\le u}\left(i \neq j \Rightarrow c_{x_j} = \text{white}\right)\right). \tag{26}$$

This section focuses on making a detector for one-black colourable one-pixel SSC. For this type amount of solutions is $u!$ so Function $approximate$ is known. Algorithm 14 is the detector for one-black colourable one-pixel line SSC. This algorithm needs to check every edge is compliant to one-black colourable one-pixel line. For this compliant check Algorithm 13 is used.

Algorithm 13 input is the edge ($(v_{x_0,y_0}, v_{x_1,y_1})$) under inspection (this includes the unknown pixel locations), leftmost block covering pixel c_{x_0,y_0} (b_{j_0}), and boolean ($fullbackallowed$) to indicate if fullblack edge is allowed. Output from the Algorithm 13 is boolean which is true, if the edge is compliant with one-black colourable one pixel line. The input parameter b_{j_0} is assumed to be rightmost block covering the pixel c_{x_1,y_1} got from result of Algorithm 3. (Note that pixels are in same line $line$ but since it is not known is this column or row both index lower index are different.)

Algorithm 13's steps 1 and 3 are just setup. Value of the edge is extracted to variables $state$ and $stateinfo$. Variable b_{j_1} is set to rightmost block covering the pixel c_{x_1,y_1}.

Running of the Algorithm 13 divides into three parts based upon value in $state$. Steps from 5 to 6 handle case where $state$ is a `nextto` or `fullwhite`. Steps from 8 to 11 handle case where $state$ is a `fullblack`. Steps from 13 to 15 handle case where $state$ is a `mix`.

Two checks are preformed for `nextto` or `fullwhite`. First check makes sure block indexes are the same. This means pixels c_{x_0,y_0} and c_{x_1,y_1} are covered by the same block. This block has to be size of 1 which is the other check performed. Size check is needed for 1 pixel requirement of one-black colourable one-pixel line.

For `fullblack` edge same block index check as in `nextto` or `fullwhite` edges is performed. Since that block must be length of one plus black pixel group in the middle, check is performed that block size minus the length of edge (in between pixels) is 1. Algorithm 11 stores the block size to $stateinfo$. Other test performed is the allowance of fullblack edges in the line relevant to a switching component.

For `mix` edge block size has to be one for the one-black colourable one-pixel line. This test is done for both blocks, because for the `mix` edge they are not the same. Blocks have to overlap on black pixel groups and by definition of one-black colourable one-pixel every black pixel group must be size of one. This requirement reduces down to checking that biggest black pixel group is size of one (hence all of them are size of 1) and checking that between j_0 and j_1 is a block for every black pixel group and for one of the unknown pixels. Block range are not needed to be checked because of rule 2.2 makes sure that block range end is not next to black pixel, rule 2.5's Equation 21 makes sure no block range ends to white pixel, and fact that mix edge exists (hence blocks overlap).

Algorithm 13 input parameters b_{j_0}, and $fullbackallowed$ are used in these steps. Boolean $fullbackallowed$ gets its value from the question $J = 2$. For b_{j_0} support array $\{rowblks_j\}_j$ and variable $clmblk$ are used to store block index of a block covering the unknown pixel under investigation. Array $\{rowblks_j\}_j$ and variable $clmblk$ are needed since mixed edge change the block in between the two unknown pixels. Array $\{rowblks_j\}_j$ stores block index of the rows and $clmblk$ for each column. Their value is updated by the second element in the output pair of Algorithm 14.

Algorithm 13 CheckOneBlackColourableOnePixelCompliance

Input: edge $(v_{x_0,y_0}, v_{x_1,y_1})$, block previously on line $line$ is b_{j_0}, and boolean $fullbackallowed$.
Output: Pair (a,b), where a true or false on is edge compliant and b is leftmost block covering c_{x_1,y_1}

```
1.  set variable $state$ to $(v_{x_0,y_0}, v_{x_1,y_1})$'s value's zeroth member.
2.  set variable $stateinfo$ to $(v_{x_0,y_0}, v_{x_1,y_1})$'s value's first member.
3.  set $b_{j_1}$ to rightmost block that covers pixel $c_{x_1,y_1}$ on line $line$
4.  if $state = \text{nextto}$ or $state = \text{fullwhite}$
5.      if $j_0 = j_1 \wedge b_{j_0} = 1$ then return $(\texttt{true}, b_{j_0})$      ; only acceptable block size is one.
6.      else then return $(\texttt{false}, b_{j_0})$
7.  else if $state = \text{fullblack}$
8.      if $b_{j_0} - stateinfo = 1 \wedge fullbackallowed$      ; only one black pixel is to be coloured
9.          if $j_0 = j_1$ then return $(\texttt{false}, b_{j_0})$      ; check that blocks keeps itself to same
10.         else then return $(\texttt{true}, b_{j_0})$
11.     else then return $(\texttt{false}, b_{j_0})$
12. else if $state = \text{mix}$
13.     if $b_{j_0} > 1 \wedge b_{j_1} > 1$ then return $(\texttt{false}, b_{j_0})$      ; only acceptable block size is one.
14.     if $stateinfo_1 = 1 \wedge j_0 + stateinfo_2 = j_1$ then return $(\texttt{true}, b_{j_1})$
15.     else return $(\texttt{false}, b_{j_0})$
```

Actual detector (Algorithm 14) has three parts: setup (steps 1 to 14), loop for handling zeroth column (steps 15 to 25), and double loop (26 to 49) to handle rest of the columns. Main idea to the algorithm is to check that every vertex in a column has edges:

- to higher y index vertex without skipping a row,
- and that there is an edge to lower x index vertex that is same as the previously handled column.

The setup starts with finding of the boundary coordinates of the switching component (steps 1 to 4). Since one-black colourable one pixel SSC is a square then vertices $v_{x_{min},y_{min}}$, $v_{x_{min},y_{max}}$, $v_{x_{max},y_{min}}$, and $v_{x_{max},y_{max}}$ have to be the four corners of the that rectangle. Corners $v_{x_{max},y_{min}}$, and $v_{x_{max},y_{max}}$ are checked in the third part but for the zeroth column existence of vertexes $v_{x_{min},y_{min}}$, and $v_{x_{min},y_{max}}$ is checked in step 5. If nether exist then given unknown pixel graph (V,E) cannot be one-black colourable one pixel SSC.

Rest of the setup (steps 7 to 14) initializes variables for the second and third parts loops. Variable u is number of unknown pixels squares side has. It is used to memory allocation big enough for other variables in the setup. This **cannot** be calculated by $y_{max} - y_{min}$ since these are coordinates in partial solution T which is **not** number of unknown pixels. In the case where (V,E) is a one-black colourable one-pixel SSC, variable u can be got by counting number of vertices in the shortest path between $v_{x_{min},y_{min}}$ and $v_{x_{min},y_{max}}$. In the implementation the shortest path finder should **not** be used because there is cases of non-SSC which can give more than needed. Better implementation, for example, is to just pass through zeroth column to get the variable u.

As Algorithm 14 checks compliance with Algorithm 13 block which edge starts is needed. Variables $clmblk$ and $rowblks_j$ store current block covering unknown pixels for this check. These variables are updated by the return value of the Algorithm 13. Variable $clmblk$ is recalculated by start of every column unlike array $rowblks_j$.

Array Y_j is the previous columns row indexes. These values should not change column to column in a SSC hence there are initialized ones in the zeroth column loop.

Variables v_{p_x,p_y}, p_{next} and j are used to index the iteration. Vertex v_{p_x,p_y} is the vertex which edges are under checking. Hence, p_x is the column under process and p_y is the row under process. Variable p_{next} is the next row index in the column. Vertex v_{p_x,p_y} is initialized to $v_{x_{min},y_{min}}$ (which is top left corner) and is initialized to next higher row index. Variable j indexes the arrays $rowblks_j$ and Y_j.

Zeroth column loop (steps 15 to 25) runs as long as there is an edge to higher row index (step (15)). Right after this is usage of Algorithm 13 to check that edge $\left(v_{p_x,p_y}, v_{p_x,p_{next}}\right)$ is in compliance. If the compliance of the edge is fine, then loop updates the $clmblk$ add adds row index to Y_j. After that v_{p_x,p_y} , p_{next}, and j are incremented for the next iteration. This will go through all the rows of the zeroth column. When loop exits, there is final check row loop ended on y_{min}.

Next is double loop handling rest of the columns (steps 15 to 25). Outer loop goes through the columns along zeroth row index. If given (V,E) does not have zeroth row index with all of the columns, then (V,E) is not a SSC. This is checked by two ways. One is that last column iterated was x_{max} (step 49), other one is using variable g_x to check that current column p_x has an edge to previous column in step 33. Other check done in that step is that row matches with what is stored in Y.

Inner loop (steps 32 to 45) goes through the rows of the column p_x and check that edge to column lower and row higher is exists and is in compliance with one-black colourable one-pixel line. For cycle, there is no check that column has lower row index since if that is the case there would be y_{min} value which would have been rejected at step 5.

Algorithm 14 NonoDetectOneBlackColourableOnePixelSquareSwitchingComponent

Input: Nonogram N, Partial solution T, Unknown pixel graph (V,E).

Output: answer (true or false) to is a (V,E) a one-black colourable one-pixel square switching component.

```
1.  $x_{max} := \mathbf{max}\left(\left\{x \mid v_{x,y} \in V\right\}\right)$    ; Get the bounds of the unknown pixel graph.
2.  $y_{max} := \mathbf{max}\left(\left\{y \mid v_{x,y} \in V\right\}\right)$
3.  $x_{min} := \mathbf{min}\left(\left\{x \mid v_{x,y} \in V\right\}\right)$
4.  $y_{min} := \mathbf{min}\left(\left\{y \mid v_{x,y} \in V\right\}\right)$
5.  if vertexes $v_{x_{min},y_{min}}$ and $v_{x_{min},y_{max}}$ do not exist
6.      return false
7.  set $u$ to number of vertices in the shortest path between $v_{x_{min},y_{min}}$ and $v_{x_{min},y_{max}}$
8.  set $clmblk$ to block that covers pixel $c_{x_{min},y_{min}}$ on column $x_{min}$
9.  allocate $\left\{rowblks_j\right\}_{j=0}^{u-1}$    ; Array of blocks for each row in switching component
10. set $rowblks_j$ to block that covers pixel $c_{x_{min},y_{min}}$ on row $y_{min}$
11. allocate $\left\{Y_j\right\}_{j=0}^{u-1}$    ; Previous columns row indexes
12. $v_{p_x,p_y} := v_{x_{min},y_{min}}$    ; Also sets iterator variables $p_x$ and $p_y$
13. $j := 0$
14. $p_{next} = \mathbf{min}\left(\left\{y \mid v_{p_x,y} \in V \wedge p_y < y\right\}\right)$    ; Next row
15. while $\left(v_{p_x,p_y}, v_{p_x,p_{next}}\right) \in E$
16.     $result$ from Algorithm 13 for $\left(v_{p_x,p_y}, v_{p_x,p_{next}}\right)$, $clmblk$, and $u = 2$
17.     if $result_0 =$ false    ; Zeroth member of the variable
18.         return false
19.     $clmblk := result_1$
20.     $Y_j := p_y$
21.     $v_{p_x,p_y} := v_{p_x,p_{next}}$    ; Move to next edge
22.     $p_{next} = \mathbf{min}\left(\left\{y \mid v_{p_x,y} \in V \wedge p_y < y\right\}\right)$
23.     $j = j + 1$
24. if $p_y \neq y_{max}$
25.     return false
```

```
26. $p_x \coloneqq x_{min}$
27. while $\boldsymbol{min}\left(\left\{x \mid v_{x,y_{min}} \in V \wedge p_x < x\right\}\right) \leq x_{max}$
28.     $g_x \coloneqq p_x$
29.     $p_x \coloneqq \boldsymbol{min}\left(\left\{x \mid v_{x,y_{min} \in V \wedge p_x < x}\right\}\right)$
30.     $p_y \coloneqq y_{min}$
31.     $j \coloneqq 0$
32.     while $p_y \leq p_{max}$
33.         if $\neg\left(\left(v_{g_x,p_y}, v_{p_x,p_y}\right) \in E \wedge p_y = Y_j\right)$
34.             return false
35.         $result$ from Algorithm 13 for $\left(v_{g_x,p_y}, v_{p_x,p_y}\right)$, $rowblks_j$, and $u = 2$
36.         if $result_0 =$ false
37.             return false
38.         $rowblks_j \coloneqq result_1$
39.         $p_{next} \coloneqq \boldsymbol{min}\left(\left\{y \mid v_{p_x,y} \in V \wedge p_y < y\right\}\right)$
40.         $result$ from Algorithm 13 for $\left(v_{p_x,p_y}, v_{p_x,p_{next}}\right)$, $clmblk$, and $u = 2$
41.         if $result_0 =$ false
42.             return false
43.         $clmblk \coloneqq result_1$
44.         $j \coloneqq j + 1$
45.         $p_y \coloneqq p_{next}$
46.     if $p_y \neq y_{max}$
47.         return false
48. if $p_x \neq x_{max}$
49.     return false
50. return true
```

Algorithm 14 uses features of ***NonoPartialSolver***(Algorithm 6) to get guarantee that detected switching component is always one-black colourable one-pixel *SSC*. In partial solution of Algorithm 6:

- Chain of overlapping block ranges starts and ends to an unknown pixel.
- There is no unknown pixel group sandwiched between black pixel groups covered by one block.
- There is no pattern unknown pixel group, white pixel group, and unknown pixel group sandwiched between blackpixel groups covered by one block.

By rule 2.5 guarantee (Equation 21), white pixels can not be and start and end of chain of overlapping block ranges. For black pixel Rule 3.3-1 (Equation 23) would refine block ranges of the starting block or ending block so that no overlap would exist. Hence, only scenario left is one in which unknown pixels starts and ends a chain of overlapping block ranges. This is important since Algorithm 6 can assume when processing is at the border of the unknown pixel graph block range also starts at that border and does not go beyond it.

Figure 35 shows example of overlapping chain of blocks. In Subfigure (a) is the initial state of the line segment. There are four blocks overlapping with each other; block b_{j_0}, block b_{j_1}, block b_{j_2}, and block b_{j_3}. Block b_{j_0} range leftmost pixel is black and block b_{j_3} rightmost is white pixel. Block b_{j_0} is size of 2 and it does not have overlapping block left to it. Hence, block b_{j_0} can be refined by rule 3.3-1 to pixels c_{x_0} and c_{x_1} removing block b_{j_0} from overlap chain. Rule 2.5 other hand refines block b_{j_3} to rightmost from pixel c_{x_7} to pixel c_{x_6} These changes are reflected at Subfigure (b). Notice that now overlapping block chain leftmost and rightmost pixels are unknown. In the middle of the chain, however, block ranges can end on black pixel (c_{x_3} compared to c_{x_4} or c_{x_5}).

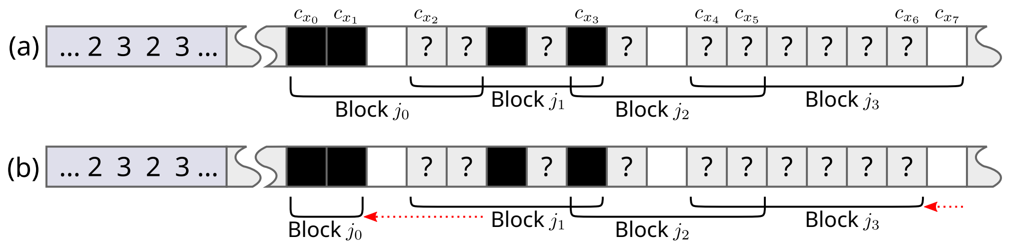


Figure 35. Example chain of overlapping block sequences. (a) is initial state of the line. (b) is refined state of the line.

For latter two, Rule 3.1 would be colour the middle of black pixel groups which middle is only covered by one block. For the case, where middle has white pixel group recolouring would happen which would be detected by Rule 3.1, whereas without unknown pixel would be coloured black. Rule 2.2 would also make sure that, if block range would start or end in middle of black pixel group then it would be refined out of the black pixel group. These two means there is no need to take account situations where **two** `fullblack` edges are covered by one block as they do not happen.

Figure 36 shows proof visually. Initially line segment is shown at Subfigure (a). Initially block b_{j_0} is covers unknown pixel group, black pixel group, unknown pixel group, black pixel group, unknown pixel group and overlaps with blocks b_{j_0-1} and b_{j_0+1}. In Subfigure (b) is resulting segment of the line. Rule 2.2 refines block b_{j_0-1} and b_{j_0+1} ranges. Block b_{j_0-1} rightmost is refined to c_{x_2} from c_{x_3} and block b_{j_0+1} leftmost is refined from pixel c_{x_6} to c_{x_7}. After this rule 3.1 would colour pixels c_{x_4} and c_{x_5} to black and refine block b_{j_0} leftmost to pixel c_{x_1} from c_{x_0} since including pixel c_{x_0} would make 14 pixels long block. Block b_{j_0} is 13 pixels.

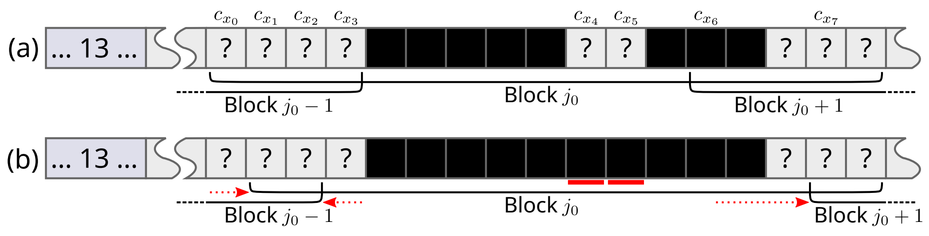


Figure 36. Shows proof visually. (a) is the line segment initially. (b) is line segment after rule 2.2 and then rule 3.1 are applied.

Figure 37 shows example of the situation. In Subfigure (a) is the initial state of the line segment. Segment has on block b_{j_0} which covers an black pixel group, unknown pixel group, white pixel group (Q_{w_M}), unknown pixel group, black pixel group. In Subfigure (b) is what result when Rule 3.1 is used. Red coloured pixels mark recolouring what would happen.

Algorithm 11 checks (mostly with Algorithm 13) that every line relevant to given unknown pixel graph (V,E) that is shaped in SSC

- If there is one block b_0 covering the line relevant to (V,E) then
  - If $b_0 = 1$.
  - If line has `fullblack` edge then line relevant to (V,E) has just two unknown pixels.

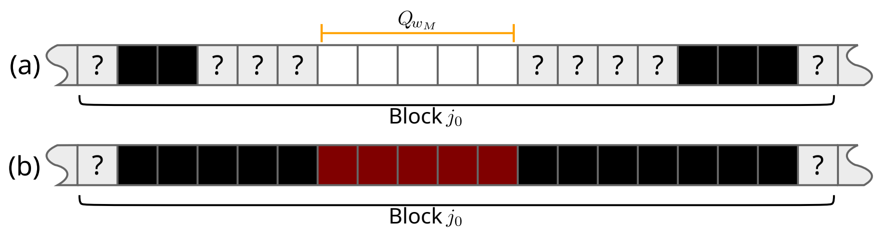


Figure 37. Example of pattern unknown ,white, and unknown covered by one block. (a) is the initial state of the line. (b) is the resulting recolouring from rule 2.3.

  - If line has no `fullblack` edge then edges can be a mixture of `fullwhite` pixel edges and next to edges.

- If there is multiple blocks $\{b_j\}_{j=0}^{k-1}$ covering whole line relevant to (V,E) then
  - If every block is length of one ($b_j = 1$).
  - If exists `mix` edges which total black group amount is $k-1$.
  - If every black pixel group in `mix` edge is length of one.
  - If no `fullblack` edge exists in line relevant to (V,E)

then (V,E) is a one-black colourable one-pixel SSC.

Edge state being `fullblack` and part of a one-black colourable one-pixel line can only happen in ESC. As mentioned, unknown pixels group between the `fullblack` edges is not possible. Having `fullblack` edge, `fullwhite` edge, `fullblack` edge would not be explained with one block. Any black edge created by Algorithm 6 could not be bigger than block as no rule used in Algorithm 6 has that ability. Hence, possibility for `fullblack` edge happening outside of ESC is that of unknown pixel groups larger than 1 sandwiches the `fullblack` edge. This does not produce one-black colourable one-pixel line since colouring edge unknown pixels black would cause two black pixel groups under one block. Having `mix` edge middle between `fullblack` edges has a problem with block size. Since `mix` edges have to change the block, there is not big enough unknown pixel group with one block covering it. Otherwise, more than one black pixel would be coloured hence it is not possible be part of one-black colourable one-pixel line anyway.

Other edges on their own do produce valid one-black colourable one-pixel line (see Figures 29, 30, 31, and 32) . Mixture of these edges is also a one-black colourable one-pixel line. Only property is needed to be checked is regard to `mix` edge has one less black pixel group middle of it then blocks cross it.

6 ANALYSIS OF ALGORITHMS

This chapter analysis the complexity of the Algorithm 1 and it's subalgorithms. Analysis was done mathematical without assuming much computation model and empirically using implemented software [13]. Test samples for empirical data were generated via support program. Both methods are used to demonstrate that algorithms run efficiently (meaning in polynomial time). They are **not** used to show best exact running time.

Software was run on AMD Ryzen 7 2700X (3.7 GHz) with 32 gigabytes of RAM. Software utilizes only one thread of execution. Operating system used was Linux 5.15.63. To gather the timing data, Linux's system call *clock_gettime* with *CLOCK_PROCESS_CPUTIME_ID* flag was used. With this flag, time given is the amount of CPU time process (every thread) has spent. It was chosen, because it does not include time spend on other processes or in the kernel [17].

It was observed that there is some natural variance on the timing information. This is why every nonogram generated was sampled 10 times. For some plots average over 10 samples was plotted.

Timing information was taken:

- On total time of the implemented (Algorithm 1).
- On partial solver (Algorithm 6).
- On verification and switching component finder (Algorithm 9).
- On switching component detection (steps 11 to 14 of algorithm 1) .
- On one switching component's estimation (Algorithm 14).

6.1 Empirical data generation

To create the nonograms to test the software with, generator program was created. This generator is able make nonograms with one-black colourable one-pixel SSC most of the time. There is a change created nonogram does not have one-black colourable one-pixel SSC or nonogram with solutions at all. This is a good thing as program's failure of detection is also tested. Nonogram with no solutions which are noticed by verifier are ignored from data gathering.

Generator works by first choosing between `nextto` edge and other edges. This is because next to `nextto` has to be other `nextto` edge. Probability for generating this `nextto` line is weight so that each edge has about the same probability being `nextto`, `fullwhite`, or `mix` when $u > 2$ or when $u = 2$ with `nextto`, `fullwhite`, `mix`, or `fullblack`. Probability for the line of `nextto` edges exactly would be $(2^{u-1}+1)^{-1}$ where u is the size of the switching component. To estimate this probability generator uses is $(2^{n+1})^{-1}$. If line of `nextto` edges is not generated, then length of line of edges in pixel is randomly chosen. After that each edge is uniformly assigned ether `mix`, or `fullwhite` edge type. If $u = 2$, then special case of having `fullblack` edge is allowed.

After generating edge type of each edge and their length, generator fills these segments of the with pixels according to the type. Rest of the nonogram is filled so that coloured pixels not part of the switching component edge next to unknown pixels are white and rest are black. This filling is done so that lines not part of switching component would be static. However, this is not perfect and occasion static line filling fails and switching component larger than indented is generated which can also lead to non-valid nonogram.

Generator can generate nonograms with one-black colourable one-pixel SSC or multiple one-black colourable one-pixel SSC as a column. Nonogram with column of one-black colourable one-pixel SSC places some padding between switching components. Padding grows the probability switching components are isolated from each other. Column nonograms are generated to have data on nonograms which include multiple switching components.

6.2 Results

Often used support Algorithm 3 complexity is $\mathcal{O}\left(a_{n,m}^2\right)$ where $a_{n,m}$ is ether n or m. This is proven in Theorem 6.1.

Theorem 6.1. *Algorithm 3 runs in* $\mathcal{O}\left(a_{n,m}^2\right)$.

Algorithm 3 runs an operation per block in the line. As discussed in the Chapter 2, description length longer than line would not be valid nonogram. These caps amount of block sane description can have to less than number of pixels in the line. Hence, complexity is $\mathcal{O}\left(a_{n,m} f\left(a_{n,m}\right)\right)$. Function f here is deferment by “if“ sentence and addition to set operations. At worst both of these operations are linear time in complexity. This mostly depends on does underline computational model need to go through the blocks for these operations. Hence, Algorithm 3 runs in $\mathcal{O}\left(a_{n,m}^2\right)$. ■

Complexity of partial solver is proven to be $\mathcal{O}\left(n^4 m^4\right)$ in Theorem 6.2.

Theorem 6.2. *Partial solver (Algorithm 6) runs in* $\boldsymbol{O}\left(n^4m^4\right)$.

Algorithm 6 starts with allocation of a partial solution. This step takes at max $\boldsymbol{O}(nm)$.

Next step is initialization loop (steps 2-8). This runs initialization rule, rule 0.0, and rule 1.1 $n+m$ times. Initialization rule goes through every block once hence it is $\boldsymbol{O}\left(a_{n,m}\right)$ complexity, where $a_{n,m}$ is ether n or m depend upon is line column or row. Rule 0.0 goes if triggered goes through every pixel in the line so it is also $\boldsymbol{O}\left(a_{n,m}\right)$. Rule 1.1 goes through every block and colours at max the block amount of pixels. Hence, assuming (unrealistically) that there is $a_{n,m}$ blocks of $a_{n,m}$ length complexity of rule 1.1 is $\boldsymbol{O}\left(a_{n,m}^2\right)$.

Next loop (9-16) applies rules until no updates are made to T. At max partial solver makes $nm+1$ updates if every update cycle makes one update. The biggest rule complexity wise are rules 1.5, 2.4. Rules 1.5, and 2.4 go through the pixels and calls find blocks covering pixels making them $\boldsymbol{O}\left(a_{n,m}^3\right)$.

Block swap guard goes through each block ones adding $\boldsymbol{O}(nm)$ to the complexity.

Some portion of pixels which are updated per cycles are updated by rule applied to a row others to a column. Marking portion when update is on a row by σ means that complexity of update loop is

$$\begin{aligned}
&\boldsymbol{O}(nm)+\boldsymbol{O}\left(\sigma(mn+1)n^3+(1-\sigma)(mn+1)m^3\right)+\boldsymbol{O}(nm)\\
&=2\boldsymbol{O}(nm)+\boldsymbol{O}\left(\sigma(mn^4+n^3)+(1-\sigma)(m^4n+m^3)\right)\\
&=\boldsymbol{O}(nm)+\boldsymbol{O}\left(\sigma(m^4n^4)+(1-\sigma)(m^4n^4)\right)\\
&=\boldsymbol{O}\left(m^4n^4\right).
\end{aligned}$$

■

Empirically complexity of partial solver seems to be somewhere between second degree polynomial and linear. In Figure 38 red curve is 3rd degree polynomial. Green curve is 2nd degree polynomial without any coefficients. Blue curve is a linear line. Yellow curve is a manually adjusted 2nd degree polynomial to find what coefficient blue dots follow. Blue dots are average over 10 samples. Figure 39 is same plot but 10-base logarithm has been taken in both dimensions. Complexity is better than analysed because complexity analysis was done without assuming any computational model and because in practise implementation has max sized nonogram it can handle. Latter, for example, means that addition arithmetic is in constant time until breaking point.

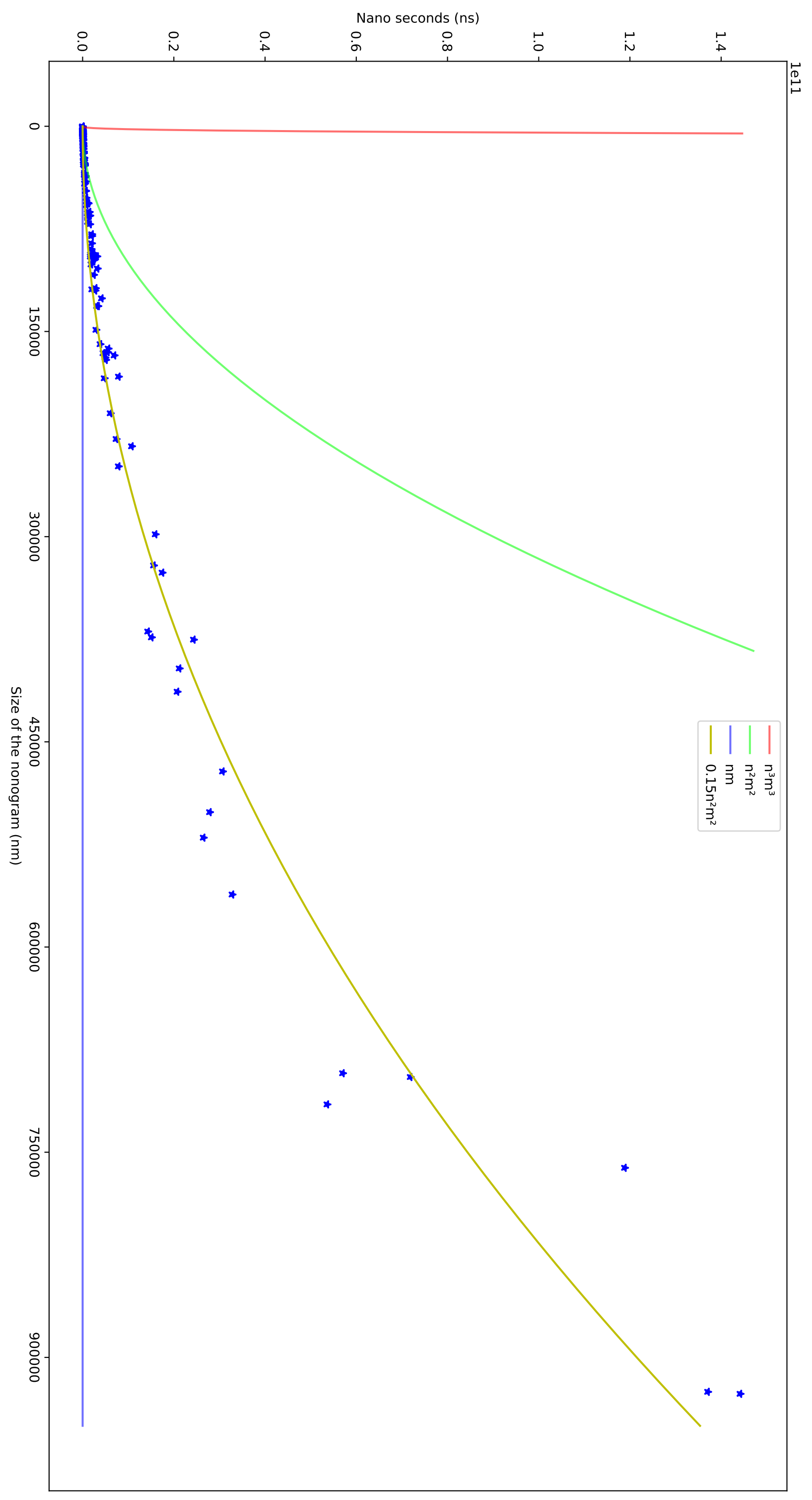


Figure 38. Plot of partials solvers complexity

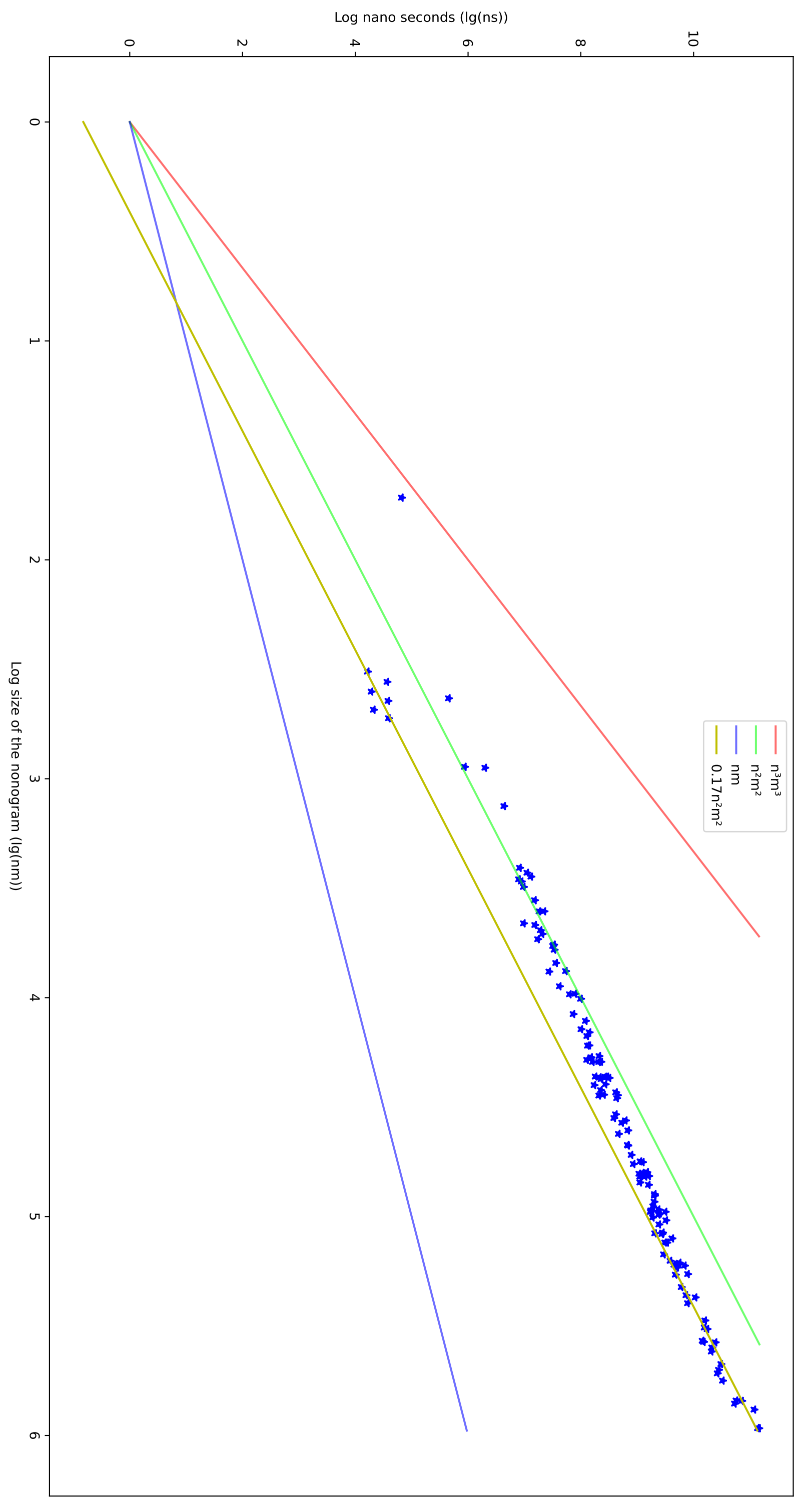


Figure 39. Log-log plot of partial solvers complexity

Complexity of verifier is $\boldsymbol{O}(nm)$ as it just goes through every pixel in partial solution T twice. Complexity of switching component finder does have more going on with it. Theorem 6.3 analyses complexity of switching component finder analytically. Figure 40 shows the empirical results of the implementation.

Theorem 6.3. *Algorithm 8 runs in $\boldsymbol{O}\left(n^4m^4\right)$ where proposed solution T is for Nonogram of n by m size.*

Algorithm 8 calls algorithm 11 and component separator algorithm.

Algorithm 11 goes through every line twice so $\boldsymbol{O}(nm)$ times. Inside of the loops is vertex/edge creation, set addition, finding vertex, finding blocks covering the vertex, counting members of the set, checking block overlap, and updating edge state information.

Vertex and edge creation runs in $\boldsymbol{O}\left(a_{n,m}\right)$ (just memory allocation).

Find from and adding to a set are depended upon computation model and data structure used. However, both at worst go through pixels again so $\boldsymbol{O}(mn)$. This is because at worst read-write head has to be moved over from other side. In reality, implementation has addition $\boldsymbol{O}(1)$ and find $\boldsymbol{O}\left(a_{n,m}\right)$ because set at max has every block of line which for sane description is less than $a_{n,m}$, since description the length of valid nonogram has to less than or equal to the length of the line.

Algorithm 3 is by Theorem 6.1 as $\boldsymbol{O}\left(a_{n,m}^2\right)$. Counting could take constant time if underline data structure already stores the information but at worst it is $\boldsymbol{O}\left(a_{n,m}\right)$.

Checking block overlap is just $\boldsymbol{O}\left(a_{n,m}\right)$ so checking multiple overlapping blocks is at worst $\boldsymbol{O}\left(a_{n,m}^2\right)$.

Function 10 is a constant time algorithm and so does not contribute to overall complexity among other processing. As such, assumption is made that every inner loop iteration ends up being unknown pixel.

For first loop's inner (steps 5 to 24) first run is $\boldsymbol{O}(1)+\boldsymbol{O}(nm)$ (because $lastunknown = \texttt{null}$ jumps over lot of processing). Succeeding iteration amount of processing depends upon value of *state*. The worst case scenario here is that every *state* is valid `mix` edge as non-overlap check is $\boldsymbol{O}\left(a_{n,m}^2\right)$ and block cover find is $\boldsymbol{O}\left(a_{n,m}^2\right)$. This does not really make sense with the assumption every pixel is unknown but it simplifies the analysis. Hence, at worst succeeding iterations are $\boldsymbol{O}\left(a_{n,m}\right)+\boldsymbol{O}(nm)+\boldsymbol{O}\left(a_{n,m}\right)+\boldsymbol{O}\left(a_{n,m}^2\right)+\boldsymbol{O}\left(a_{n,m}^2\right)+\boldsymbol{O}\left(a_{n,m}^2\right)+\boldsymbol{O}\left(a_{n,m}\right)+\boldsymbol{O}(nm)+$

$\boldsymbol{O}(a_{n,m}) + \boldsymbol{O}(a_{n,m}) + \boldsymbol{O}(a_{n,m}) + \boldsymbol{O}(a_{n,m})$. These reduces to in big Oh terms to $\boldsymbol{O}(nm)$ and $\boldsymbol{O}(nm + n^2)$ since first loop handles rows $a_{n,m} = n$ and rest of the terms are constants that do not matter.

Second outer loop's inner loop (steps 31 to 49) similarly has $\boldsymbol{O}(nm)$ for first iteration and $\boldsymbol{O}(nm + m^2)$ for succeeding iterations. Inner loop will goes every in the pixel therefore $\boldsymbol{O}(a_{m,n}\boldsymbol{f}(a_{n,m}, m, n))$ with $a_{n,m} = n$ for first inner loop and $a_{n,m} = m$ for second. However, because first iteration is different for both inner loops, first inner loop's complexity is $\boldsymbol{O}((n-1)\boldsymbol{O}(nm + n^2) + \boldsymbol{O}(nm))$ and second inner loop's complexity is $\boldsymbol{O}((m-1)\boldsymbol{O}(nm + m^2) + \boldsymbol{O}(nm))$.

To reduce these further $\boldsymbol{O}(nm)$ can be removed. Reduce even further $(n-1)\boldsymbol{O}(n)$. and $(m-1)\boldsymbol{O}(nm + m^2) = \boldsymbol{O}(nm^2 + m^3) = \boldsymbol{O}(n^3m^3)$.

Outer loops are $\boldsymbol{O}(a_{m,n}\boldsymbol{f}(a_{n,m}, m, n))$ except $a_{n,m}$ has different values. First outer loop has m and second outer loop has n. From previous paragraph $\boldsymbol{O}(m\boldsymbol{O}(n^3m^3))$ $= \boldsymbol{O}(n^4m^4)$ and $\boldsymbol{O}(n\boldsymbol{O}(n^3m^3)) = \boldsymbol{O}(n^4m^4)$.

Putting both loops together complexity comes to $\boldsymbol{O}(n^4m^4)$.

Complexity of Component finding algorithm for a graph according to [16] is $\boldsymbol{O}(\boldsymbol{max}(|V|, |E|))$ which for us means $\boldsymbol{O}(nm)$. This is morphed by complexity of Algorithm 11 hence complexity of Algorithm 8 is $\boldsymbol{O}(n^4m^4)$. ■

Empirical results of verifier and switching component finder combination is show in Figure 40. Their complexity in the implementation seems to be linear. Blue line is linear line without coefficient. Yellow line is manually adjusted linear line to see where about exact complexity is. Here exact complexity seem to be around $11nm$ plus some constant. Blue dots are average over 10 samples. In the implementation these three are combined to together so that only two passes are taken through partial solution T for verification and production of unknown pixel graph. This does not explain the order difference complete. The cause probably is the simplification in the analytical analyses of every pixel in partial solution T is a mix edge. This is probably over killing simplification.

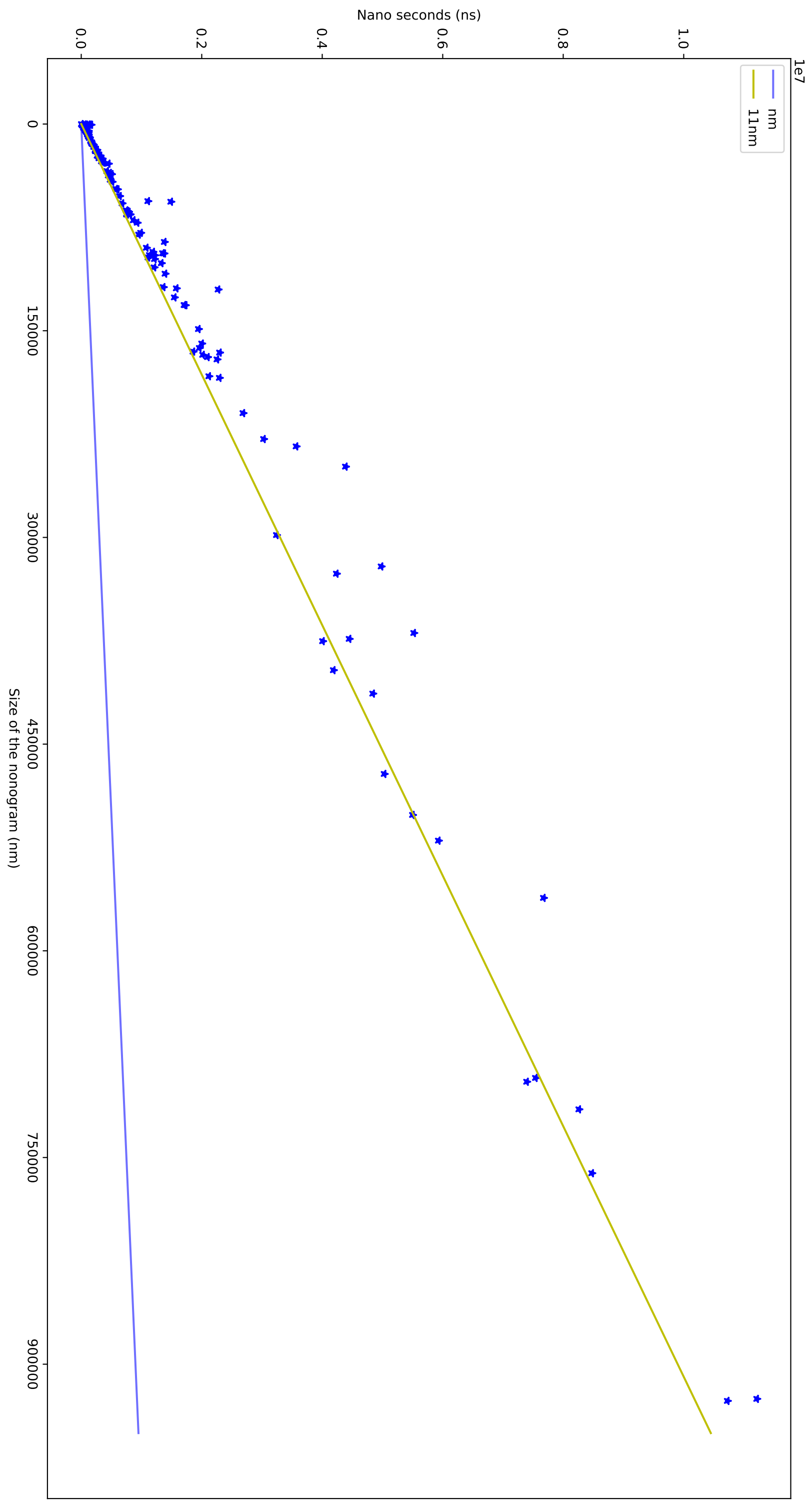


Figure 40. Plot of Algorithms 9 and 8 complexity

Complexity of the Algorithm 14 is proven in Theorem 6.4 to be $\boldsymbol{O}\left(n^3m^3\right)$.

Theorem 6.4. *Algorithm 14 is $\boldsymbol{O}\left(n^3m^3\right)$ algorithm.*

The boundary calculation in steps 1 to 4 can be group up to a single Algorithm which goes through every vertex. Since only operation done in functions like $\boldsymbol{max}$ and $\boldsymbol{min}$ are simple comparisons (so at max $\boldsymbol{O}\left(a_{n,m}\right)$) then going through every pixel these steps take $\boldsymbol{O}\left(m^2n^2\right)$. Number of switching components per Nonogram has to be less than mn since unrealistically every pixel could be a switching component.

Initializing arrays can take $\boldsymbol{O}\left(m\right)$ depend upon computational model. This is because J is at worst height of the nonogram.

Rest of the Algorithm 14 basically goes through every edge ones. Overall algorithm would be $\boldsymbol{O}\left(nm\boldsymbol{f}\left(n,m\right)\right)$.

Function $\boldsymbol{f}$ is a bit complicated in Algorithm 14. This is because Algorithm 13 has to find block cover (step 3) which means usage of Algorithm 3. Since Algorithm 13 runs Algorithm 3 for both dimensions so it safe to take $\boldsymbol{f} = \boldsymbol{O}\left(n^2m^2\right)$ because Theorem 3.

One could perhaps reduce this down to $\boldsymbol{O}\left(nm\right)$ by algorithm which takes a starting point and would not always run to the last block. This is not analysed in the Thesis because just knowing algorithm is polynomial in more important.

Hence, Algorithm 3 runs in $\boldsymbol{O}\left(n^3m^3\right)$. ■

Time spend for Algorithm 14 and calculating factorial for the detected one-black colourable one-pixel SSC is plotted on Figure 41 over size of the switching component. Figure 42 gives log-log plot on same plot as in Figure 41. It appears that green curve of 2nd degree polynomial is higher than running time of the Algorithm 11. Blue curve in the Figure 41 is linear line. Blue dots are made per sample. This is why there are columns of blue dots caused by natural error of time measurement. Lower dots forming a line are most likely switching component which are not one-black colourable one-pixel SSC. Two reason for this speculation are:

- Rejecting path is faster than accepting path for same sized switching components.
- Because generator most like makes larger switching components then intended one-black colourable one-pixel SSC when it fails.

Calculating factorial is just $\boldsymbol{O}\left(\boldsymbol{min}\,(n,m)\,(\boldsymbol{log}\,(\boldsymbol{min}\,(n,m))\,\boldsymbol{log}\left(\boldsymbol{log}\,(\boldsymbol{min}\,(n,m))\right)^2)\right)$ so it is smaller than complexity of the detector [18].

One degree lower complexity in the Figure 41 is mostly likely caused by faster block finding than what Theorem 6.4 would suggest. It is good to note that switching component size is always less than or equal to nonogram size. At worst, for the detector, there is one switching component same size as the whole nonogram. Most of the time though, detector gets far more smaller switching component to analyse.

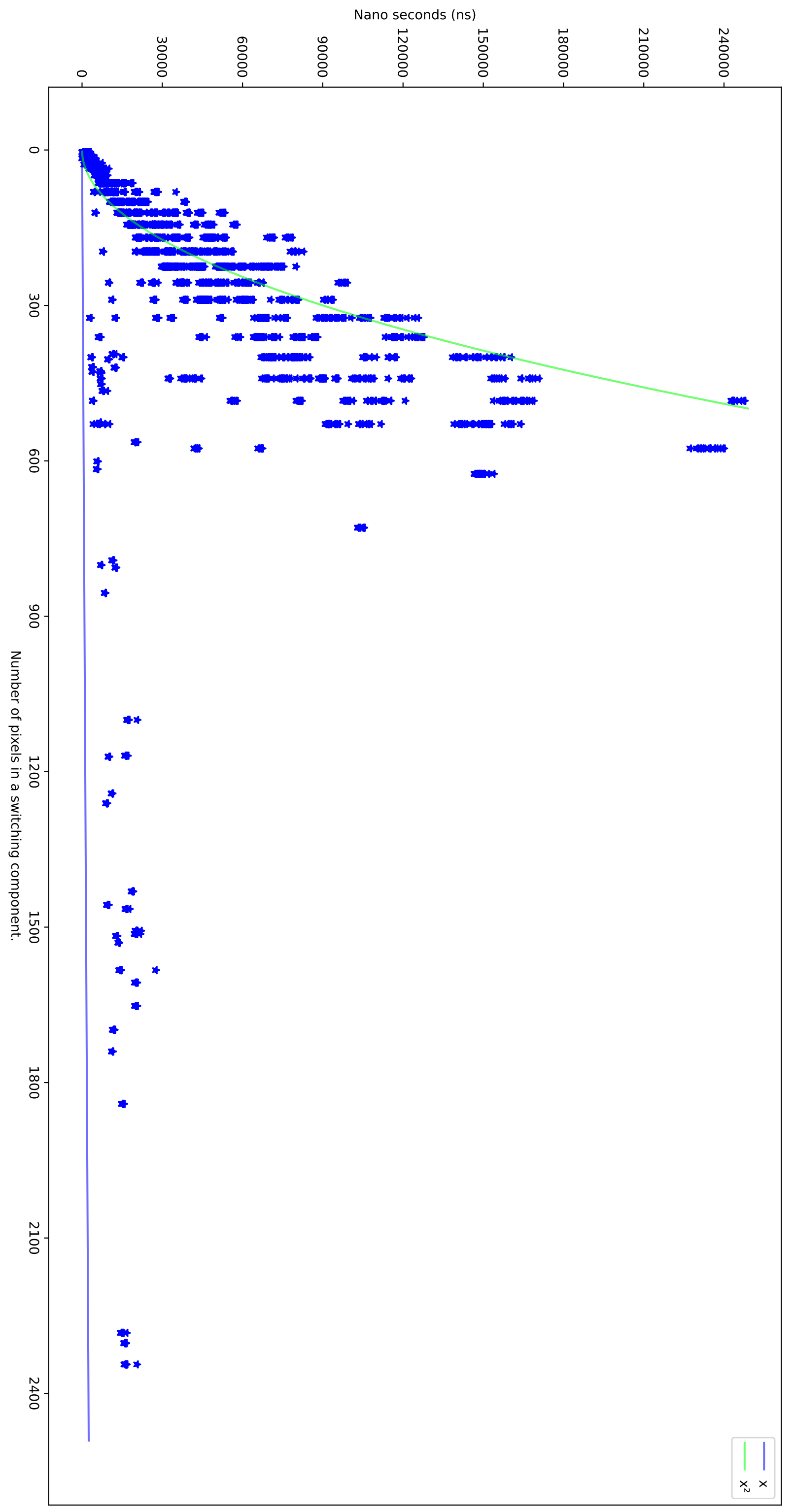


Figure 41. Plot of Algorithm 11 complexity

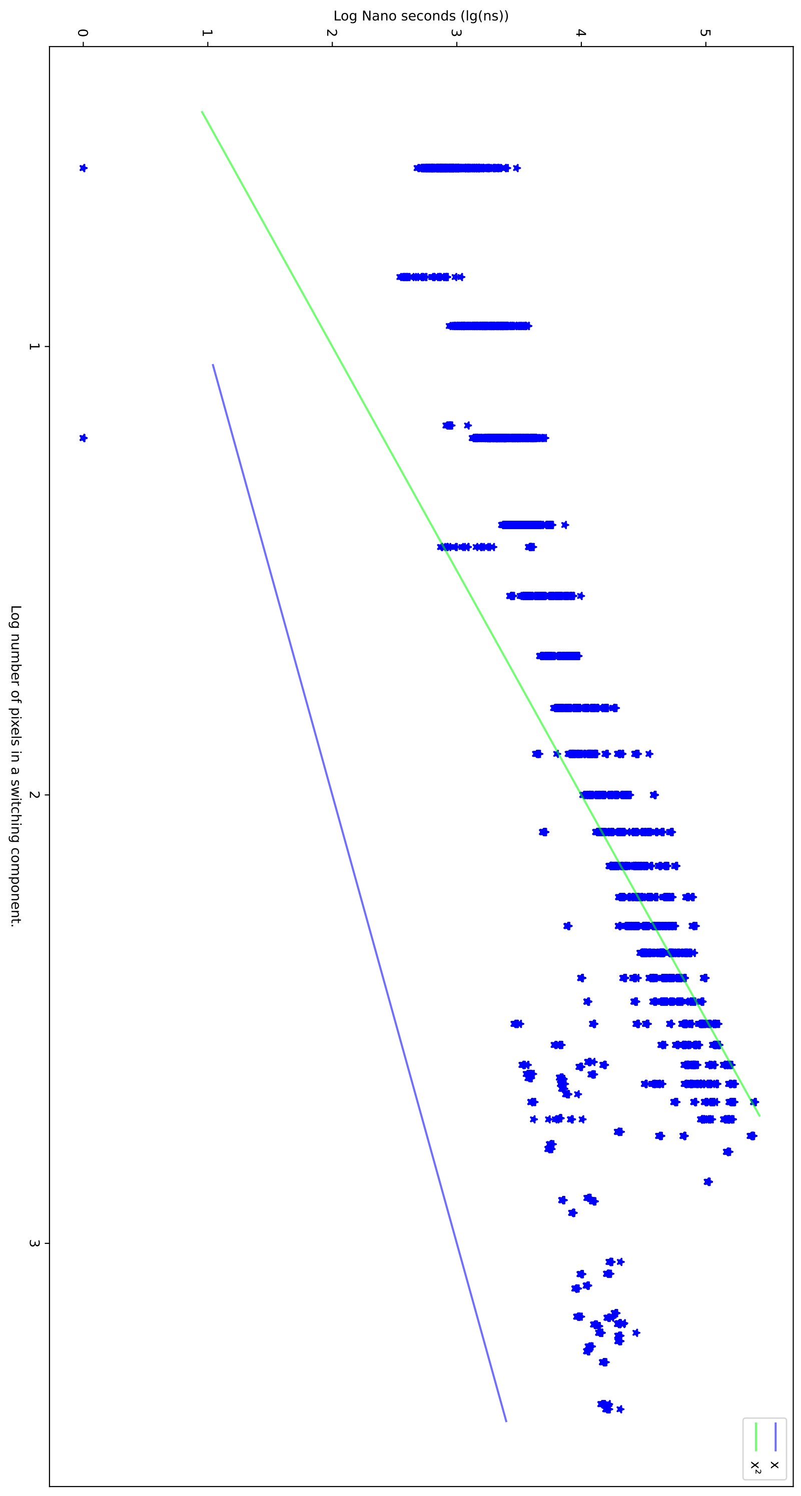


Figure 42. Log-log plot of 11 complexity

From Theorems 6.2, 6.3, and 6.4 overall complexity of the algorithm seem to be $\boldsymbol{O}\left(n^4m^4\right)$. Figure 43 show the empirical complexity of the whole algorithm (thus far). Figure 44 shows the log-log plot on same plot as in Figure 43. The overall estimators time complexity empirically similar as partial solvers. This is because around 95% of the computation time was spent on partial solver. Curves on the plot are same as last time but that blue dots should be different. Blue dots are average over 10 samples.

This result is not that surprising when reasons for reduced complexity in empirical result are considered. Algorithm 8 runs much faster when implemented and Algorithm 14 is faster than partial solver because more switching components nonogram has smaller switching components are. Partial solver also has high coefficient hidden behind big Oh notation.

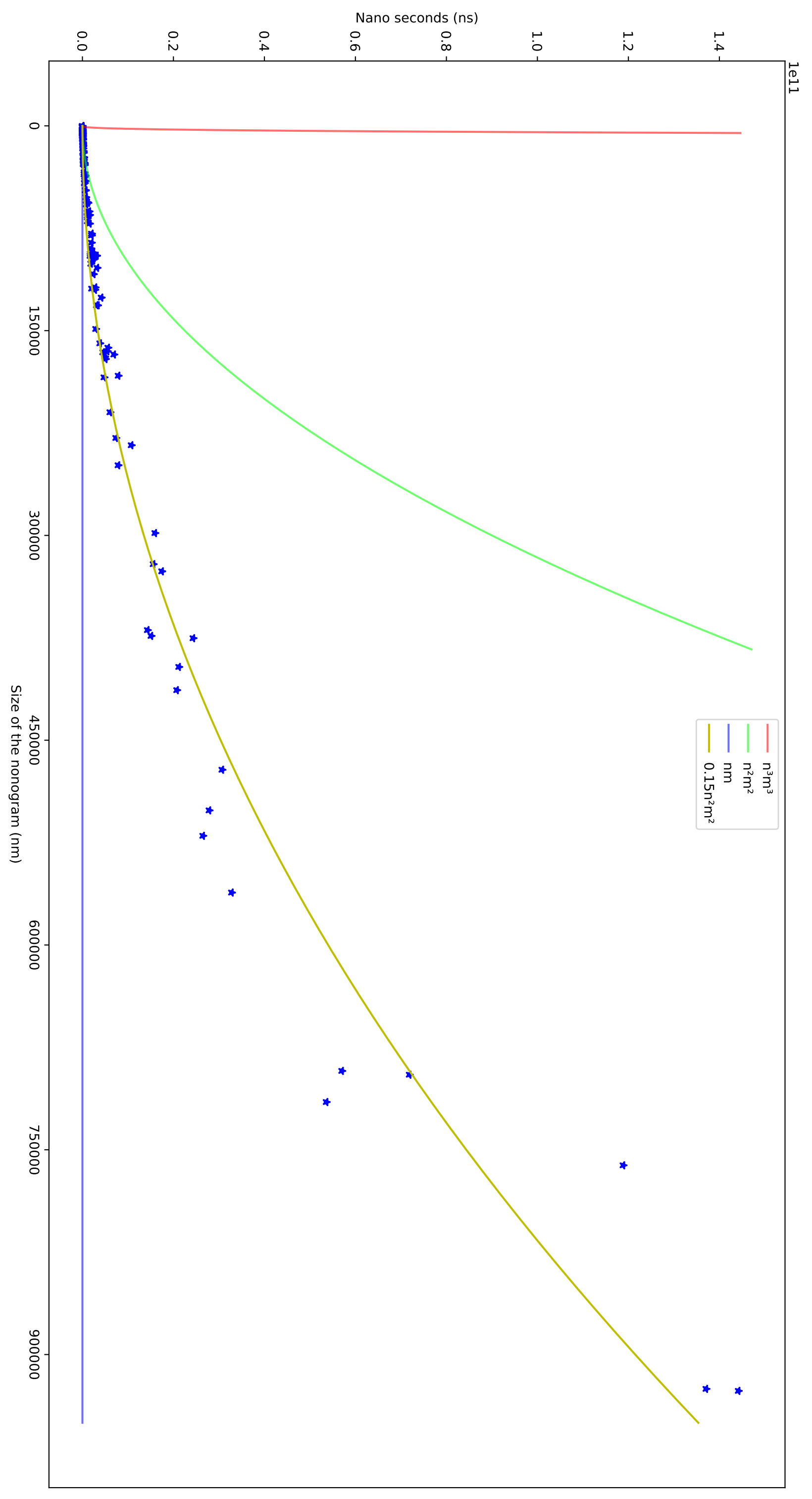


Figure 43. Plot of Algorithm 1

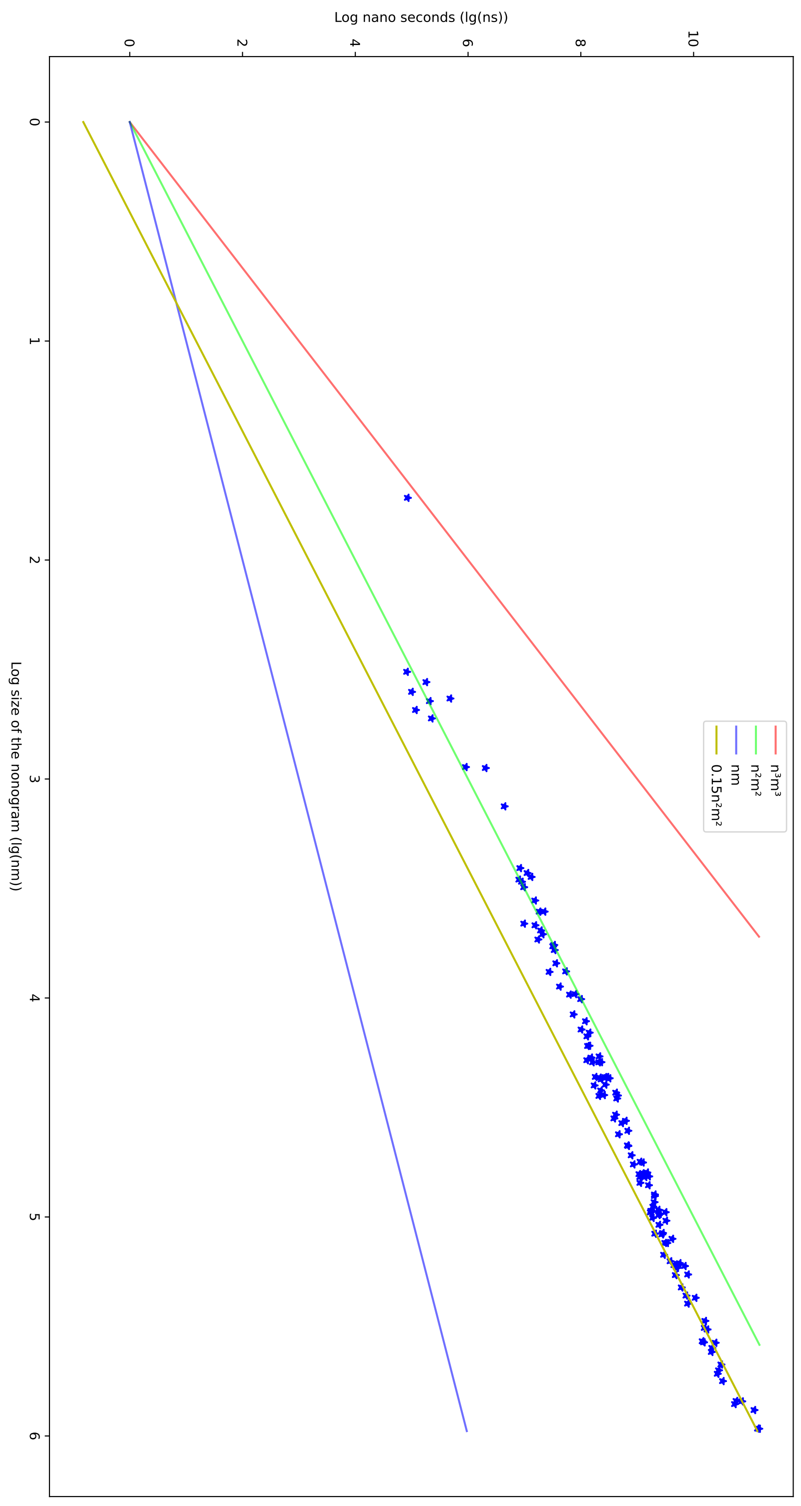


Figure 44. Log-log plot of Algorithm 1

7 DISCUSSION

Algorithm 1 thus far would be complexity of $O\left(n^4m^4\right)$. However, total algorithm does not have enough type detectors and type estimators to be for every possible nonogram. For undetected switching components sampling method could be used. This algorithm would count number of black pixels are missing per column of switching component. Daniel Berend *et al.* algorithm in [7] could also be used here to have some early rejection. However, right now adding sampling method for undetected switching components would not work. One of the missing types is large category of one-black colourable SSC. This is category still have the feature that if one black block is fully coloured in the line relevant to switching component other unknown pixels in the line relevant to switching component will be coloured white. Hence, this type can have switching components with factorially growing colourings. Figure 45 is an example of one-black colourable SSC. Figure 45 is essentially ESC scaled up width by 3 and height by 2.

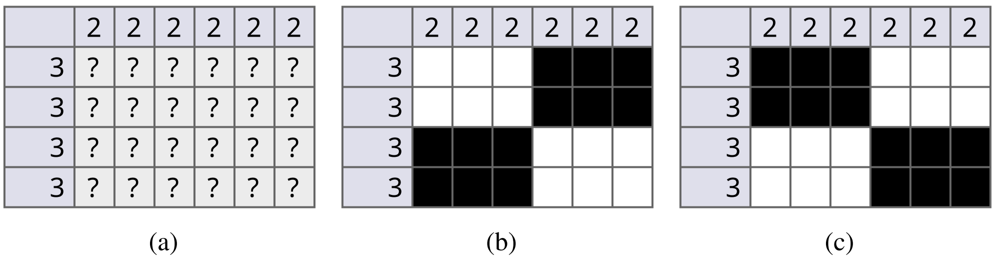


Figure 45. Example of one-black colourable SSC. (a) is the nonogram empty. (b) is a solution to the nonogram. (c) is an other solution to the nonogram.

Since, there can be other switching components which have same amount of unknown pixels then one-black colourable SSC, but only polynomial amount of solutions, general sampling method cannot distinguish between the two as it is restricted to have polynomial amount of attempts. For example, Figure 46 shows a nonogram with same amount unknown pixels than nonogram in Figure 45 and same amount solutions. Point is that, if patterns in theses nonogram is grown, then solution count of pattern in Figure 45 grows factorial whereas Figure 46 would have same 2 solutions regardless of size increase. Figure 46 type could be called a Rectangle switching component (RSC) since it is a rectangle rather than SSC.

But, for argument sake, assumed it is possible to detect and estimate any SSC and RSC ether by probabilistically or deterministically. Is this enough to create a FPRAS for nonogram? Simple, no. There are switching component types which are not SSC or RSC so it is not

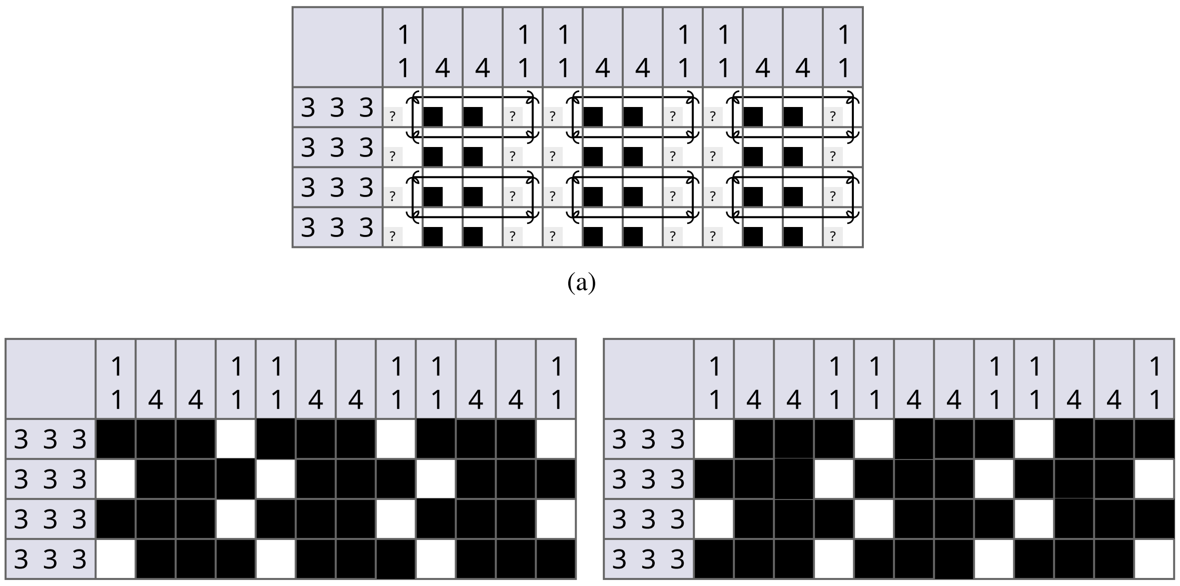
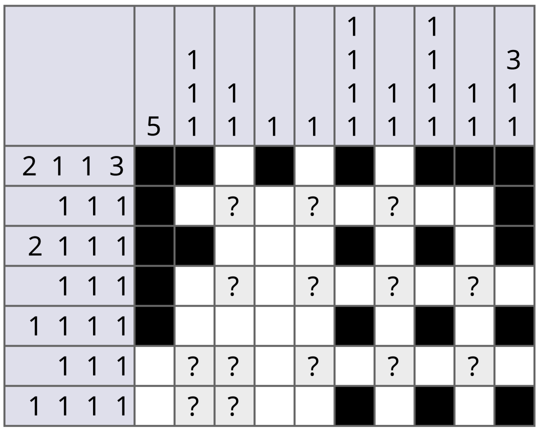


Figure 46. Example pattern which always will have two solution if nonogram is grown with the pattern. (a) is the nonogram after partial solver. (b) is a solution for the nonogram. (c) is another solution for the nonogram.

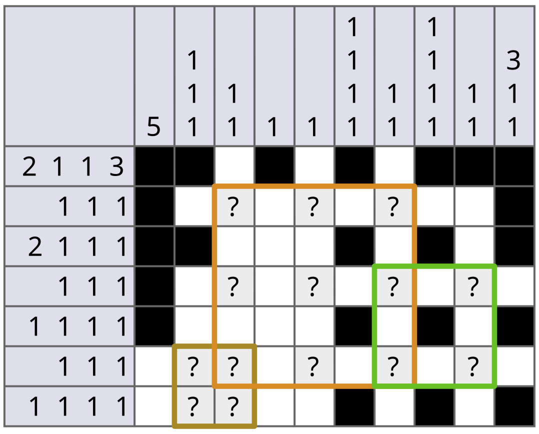
known if type categorizing or sampling would work on them.

Most important problem however, is that one can “overlap” switching components together to create bigger switching components. For example, Figure 47 shows how overlapping three SSCs creates a new switching component. In Subfigure (a) is the results of partial solver. In Subfigure (b) SSCs locations are coloured, and in Subfigure (c) is example that colouring SSC with size of 3 in the middle. This leaves two ESCs to be coloured. Green marked ESC in Subfigure (b) does not have same unknown pixels in Subfigure (c). This is because colouring SSC colours overlapping pixels but leaves other to possibilities.

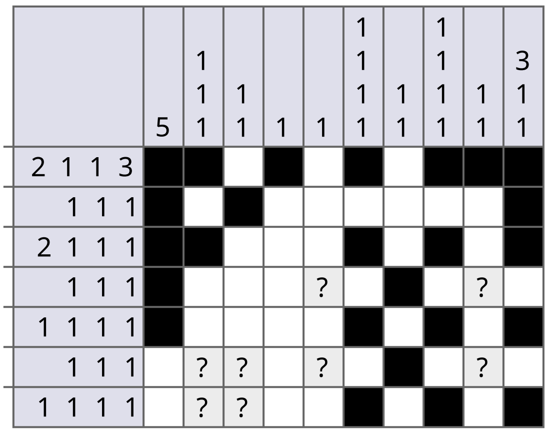
To make detector for this one maybe have to call another detector inside a detector to get what is overlapping and adjust calculations accordingly. However, as nonograms grow possibilities to create overlaps grows. This may at some point cause too deep of a recursion tree. Another way do to the detector for overlap is to do it per shape which will not work. As Nonogram grows number of shapes will grow hence one would have to make infinite detectors and estimators. As such, it could be very like that the multi-solution nonogram with SSC or RSC are small group that do have an FPRAS.



(a)



(b)



(c)

Figure 47. Example of nonogram with overlapping SSC. (a) is the partial solution of nonogram after partial solver. (b) shows were overlapping SSCs are in the partial solution. (c) is example of two ESCs remain if bigger SSC is coloured out.

7.1 Future work

First and form most, since Logical rules used in the thesis where not enough and overlapped with each other in terms of deductions:

- What is the smallest set of logical line rules which do not overlap?
- Can solve any line with only one possible configuration?
- Does such set even exist? To get this complete set of rules consequences of the nonograms validity constraint should be used.

Further, rather than using mathematical formalization used in this thesis it would be better to develop more standard formalization. Discrete tomography can easily be turned into algebra by column and row sums. Could similar technique be used to turn Nonogram simple algebra where rules are the operations of the algebra? This would help describe the more complex switching components. Specially proposed switching components internal colouring implications.

More general theoretical questions left for future work relate how much solving a nonogram is effected. Limiting multi solution nonogram to SSC or RSC, is it possible build FPRAS for these instances? Right now, algorithm does not need to use probability so there is lot of room here. For solving nonograms, how many more instance can said have a solution in polynomial time because of it?

For the software implementation, how to memory efficiently implement unknown pixel graph could use some work. Current implementation just creates memory pointers between data structures storing information about the edges and the unknown pixel. If switching component involves only few pixels of the whole partial solution, then current implementation is efficient. However, if most pixel of the nonogram are involved, meaning unknown pixels are right next to each other, implementation is rather wasteful and scales poorly as memory address cost lot of memory. For this method where some time efficiency is traded for grouping unknown pixel next to each other together could fix the scaling issue.

8 CONCLUSION

There was not any algorithm that would approximate number of solution to a nonogram. Exact number of solution algorithm does exist in the literature. This Algorithm is exponential and is essentially solves all the solutions of a nonogram. Current thesis lays looks into creation of FPRAS algorithm estimating number of solutions for #NONOGRAM. Proposed estimator for number of solutions a nonogram has comes down to processing of switching components from a partial solution. If there is finite amount of types which all have probabilistic polynomial time detector and estimator FPRAS is possible. This Thesis show that type one-black colourable one-pixel SSC can be detected and estimated in deterministic polynomial time. This type has $u!$ solutions where u is the side length of the switching component.

Full algorithm with all the parts presented in this Thesis was calculated to be $\boldsymbol{O}\left(n^4m^4\right)$. In empirical testing implementation behaved like $\boldsymbol{O}\left(n^2m^2\right)$ due to assumptions real implementation can have. Complexity of partial solver (Algorithm 6) is analytically calculated to be $\boldsymbol{O}\left(n^4m^4\right)$ but empirical testing shows $\boldsymbol{O}\left(n^2m^2\right)$. Complexity of switching component finder (Algorithm 8) is analytically $\boldsymbol{O}\left(n^4m^4\right)$. Empirically verifier (9) and switching component finder (Algorithm 8) combination seems linear in complexity. Switching component finder should be bigger of the two complexity vice. Most like analytical analyses of the Algorithm 8 is making a big worst case assumption. Detector for one-black colourable one-pixel SSC analytical complexity is $\boldsymbol{O}\left(n^3m^3\right)$. Empirical Complexity seems to be $\boldsymbol{O}\left(n^2m^2\right)$. Surprisingly, implementation spends around 95% of computation time in the partial solver (Algorithm 8).

At discussion, speculation is made that idea behind Algorithm 1 does not work as FPRAS because switching components types can be combined to make new switching components. Hence, there is problem of finding basic types which can explain combined types in polynomial time. There is also no guarantee that it is possible to have finite amount types or that these types would keep in polynomial time constraint.

At the background of the Thesis is a software develop for it. This software's source code is accessible at [13].

REFERENCES

[1] ”cat”. (), [Online]. Available: https://www.nonograms.org/nonograms/i/8563 (visited on 09/05/2020).

[2] Batenburg, K. Joost, and W. A. Kosters, ”Solving nonograms by combining relaxations”, *Pattern Recognition*, vol. 42, no. 8, pp. 1672–1683, 2008.

[3] N. Ueda and T. Nagao, ”Np-completeness results for nonogram via parsimonious reductions”, Tokoy Institute Of Technology, Tech. Rep., 1996.

[4] Miklós, István, *Computational complexity of counting and sampling*. CRC Press, 2019.

[5] ”Why is the reduction from 3-sat to 3-dimensional matching parsimonious?” (), [Online]. Available: https://cstheory.stackexchange.com/questions/47491/why-is-the-reduction-from-3-sat-to-3-dimensional-matching-parsimonious (visited on 06/07/2022).

[6] M. Dyer, L. A. Goldberg, and M. Jerrum, ”An approximation trichotomy for boolean# csp”, *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 267–277, 2010.

[7] D. Berend, D. Pomeranz, R. Rabani, and B. Raziel, ”Nonograms: Combinatorial questions and algorithms”, *Discrete Applied Mathematics*, vol. 169, pp. 20–42, 2014.

[8] V. V. Vazirani, *Approximation algorithms*. Springer, 2003, ISBN: 978-3-642-08469-0. [Online]. Available: https://www.springer.com/gp/book/9783540653677.

[9] R. Y. Rubinstein, ”How many needles are in a haystack, or how to solve# p-complete counting problems fast”, *Methodology and Computing in Applied Probability*, vol. 8, no. 1, pp. 5–51, 2006.

[10] W.-L. Wang and M.-H. Tang, ”Simulated annealing approach to solve nonogram puzzles with multiple solutions”, *Procedia Computer Science*, vol. 36, pp. 541–548, 2014.

[11] C.-H. Yu, H.-L. Lee, and L.-H. Chen, ”An efficient algorithm for solving nonograms”, *Applied Intelligent*, vol. 35, no. 1, pp. 18–31, 2011.

[12] F. " Pesquita. ”Nonogram solver”. (), [Online]. Available: https://github.com/Stabbath/nonogram-solver.

[13] H. A. Valve. ”Nonosolver application”. (), [Online]. Available: https://github.com/HenrikAkseliValve/Nonogram.

[14] R. A. Bosch, ”Painting by Numbers”, *Optima*, vol. 65, pp. 16–17, 2001.

[15] R. Mullen, ”On Determining Paint by Numbers Puzzles with Nonunique Solutions”, *Journal Of Integer Sequences*, vol. 12, Article 09.6.5, 2009.

[16] J. Hopcroft and R. Tarjan, ”Algorithm 447: Efficient algorithms for graph manipulation”, *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[17] ”Clock_getres(2) — linux manual page, Linux programmer’s manual”. (), [Online]. Available: `https://man7.org/linux/man-pages/man2/clock_gettime.2.html` (visited on 09/12/2022).

[18] P. B. Borwein, ”On the complexity of calculating factorials”, *Journal of Algorithms*, vol. 6, no. 3, pp. 376–380, 1985, ISSN: 0196-6774. DOI: `https://doi.org/10.1016/0196-6774(85)90006-9`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0196677485900069`.

[19] A. Sanjeev and B. Barak, *Computational complexity: a modern approach*, 1st. Cambridge University Press, 2009.

[20] D. E. Knuth and O. Patashnik, *Concrete mathematics: a foundation for computer science*. Addison-Wesley, 2003.

[21] A. Sinclair, *Algorithms for random generation and counting: a Markov chain approach*. Springer Science & Business Media, 2012.

Appendix 1. Complexity Theory for uninitiated

Computational complexity theory is study of *complexity classes* and what problems are in what complexity class. Complexity classes are defined to be set of *computation problems* computable in some resource limit. Computation problem is a function mapping $\{0,1\}^*$ to subset of $\{0,1\}^*$ (where $\{0,1\}^*$ is every possible pattern of 0 and 1 like 000 or 1011). *Decision problems* or *languages* are computation problem which maps to $\{0,1\}$ as no and yes answers. Decision problems are usually represented by subset of $\{0,1\}^*$ which maps to 1 [19]. For a bit pattern $x \in \{0,1\}^*$ and a decision problem $\mathsf{L} \subseteq \{0,1\}^*$ if $x \in \mathsf{L}$ then x is called a *(yes) instance* of L[8]. *Search problem* is a problem where computation finds a solution (some times called certificate) to given instance of the problem. Any search problem has associated decision problem which asks does any solution exist [19].

Every decision problem L computable by deterministic algorithm† in polynomial time is in P. Computable means that if $x \in \mathsf{L}$ then 1 or "yes" is outputted by the algorithm. Deterministic algorithm means that if algorithm's input and state at time step t are known then there is only one possible state for time step $t+1$. Polynomial time means that exists $a \in \mathbb{N}$ such that algorithm runs in $\boldsymbol{O}(|x|^a)$ time steps for all $x \in \{0,1\}^*$ where $|x|$ is length of the input (number of bits). Set P is referred as everything "efficiently" computable.[19]

Function $\boldsymbol{O}$ (pronounced as big O or big Oh) means that if $\boldsymbol{f} = \boldsymbol{O}(\boldsymbol{g})$, then there are natural numbers c and N such that $\forall n \in \mathbb{N}\,(n \geq N \Rightarrow \boldsymbol{f}(n) \leq c\boldsymbol{g}(n))$. Idea behind using notation is to get overall complexity of the algorithm somewhat independent of detail of computational model. [19] It is good to note that $\boldsymbol{f} = \boldsymbol{O}(\boldsymbol{g})$ equal sign here is one way.

Graham, Knuth, and Patashnik in [20] gave some rules on how to use big Oh in arithmetic.

$$\boldsymbol{f} = \boldsymbol{O}(\boldsymbol{f}) \tag{27}$$

$$m \leq m' \Rightarrow a^m = \boldsymbol{O}\left(a^{m'}\right) \tag{28}$$

$$\boldsymbol{O}(\boldsymbol{f}) + \boldsymbol{O}(\boldsymbol{g}) = \boldsymbol{O}(|\boldsymbol{f}| + |\boldsymbol{g}|) \tag{29}$$

$$c\boldsymbol{O}(\boldsymbol{f}(X)) = \boldsymbol{O}(\boldsymbol{f}(X)) \tag{30}$$

$$\boldsymbol{O}(\boldsymbol{O}(\boldsymbol{f})) = \boldsymbol{O}(\boldsymbol{f}) \tag{31}$$

$$\boldsymbol{f} \cdot \boldsymbol{O}(\boldsymbol{g}) = \boldsymbol{O}(\boldsymbol{f} \cdot \boldsymbol{g}) \tag{32}$$

†More formal definition would be using Turing machine. For this thesis, Turing machines are not relevant.

From Equations 28 and 30 polynomials can be reduced to the highest term in big Oh notation.

$$\sum_{n} (a_n x^n) = \boldsymbol{O}(x^n) \tag{33}$$

Time complexity of an algorithm is determent by knowing complexity of basic operation and subalgorithms, and counting how many times they are called. So for example, consecutive bit-flipping operations would take constant time ($\boldsymbol{O}(1)$) because steps are not rerun and operation itself is a constant time. If bit operations are in a non-constant loop, then analysis is figuring out how many cycles loop has. For example, loop that flips every bit on input x to it's opposite would be $\boldsymbol{O}(|x|)$.

Complexity class **NP** is every decision problem which can be calculated with deterministic algorithm (called *verifier*) in polynomial time providing certain extra bit pattern called a *certificate* which length is a polynomial to input length.[‡] Extra bits like names suggest is a solution for the problem (for example nonogram it would be fully coloured table). Essentially proposed solution has to be verifiable in polynomial time problem to be considered be in **NP**.

It is known that **P** is subset of **NP** but it is not known are they equal. This question is sometimes referred as **P** vs **NP** [19]. It is wildly held that they are not equal considering absurdity which would result [19].

Finding an algorithm for a new problem which matches definition of some complexity class is sometimes hard. This is where *reductions* help a lot. Idea is to take input x meant for problem L and pass x to a function which outputs x' which is input of problem L'. Essentially, if L' is in some complexity class, then reducing to it may show L is also in the class. There is a lot of different types of reductions with different properties. Important one is *polynomial-time Karp* reduction which takes polynomial time and $x \in \mathsf{L}$ if and only if $x' \in \mathsf{L}'$ [19].

There are decision problems in **NP**, which any other problem in **NP** reduce to by polynomial-time Karp reduction. Set of these problems is called **NP**-complete. Because of this **NP**-complete is regarded as hardest decision problems in **NP**. If one decision problem in **NP**-complete would be proven also to be in **P** then **P** = **NP** [19].

Counting problem version of a decision problem $\mathsf{L} \in$ **NP** is a computational problem counting number of certificates of $x \in \{0,1\}^*$ for algorithm computing L. This function is marked #L.

[‡]There is other equivalent definition which involves decision problem being computable for non-deterministic machine in polynomial time.

Every counting problem version of a decision problem $\mathsf{L} \in \mathbf{NP}$ form complexity class #**P**[§]

This class has complete subset #**P**-complete. If $\boldsymbol{f} \in$ #**P**-complete then every other $\boldsymbol{g} \in$ #**P** exists two polynomial time functions $\boldsymbol{R}$ and $\boldsymbol{S}$ which can compute $\boldsymbol{g}(x) = \boldsymbol{S}(x, \boldsymbol{f}(\boldsymbol{R}(x)))$ for all $x \in \{0,1\}^*$ [8].

There are multiple ways to show that counting problem is #**P**-complete. One is to show reduction from decision version of #**P**-complete to target problem is parsimonious. *Parsimonious* reduction keeps the solution count same after reduction [4].

The simplest form of approximation would be guarantee answer to be within some ratio. In literature, algorithm $\boldsymbol{A}$ is *ρ-approximation* of algorithm $\boldsymbol{f}$ if and only if for all inputs $x \in \{0,1\}^*$:

$$\rho(|x|)\boldsymbol{f}(x) \leq \boldsymbol{A}(x) \leq \frac{\boldsymbol{f}(x)}{\rho(|x|)}. \tag{34}$$

If for $L \in$ #**P** existed polynomial time ρ-approximation then **P** vs. **NP** would be solved. This is because zero would not be included in the lower-bound which would imply that L has a solution[21].

Previous definition's problem is general for approximating counting. This is fixed (somewhat) by using probability. Usually this is done by using FPRAS. Algorithm $\boldsymbol{A}$, which takes in x and additional error ratio ϵ, is FPRAS for $\boldsymbol{f}$ if and only if for all inputs $x \in \{0,1\}^*$:

$$\left(\boldsymbol{Pr}\left(\epsilon \boldsymbol{f}(x) \leq \boldsymbol{A}(x) \leq \frac{\boldsymbol{f}(x)}{\epsilon}\right) \geq \frac{3}{4}\right) \tag{35}$$

and $\boldsymbol{A}$ runs in polynomial time in $|x|$ and ϵ^{-1}.[21] Probability in the Equation 35 could be any value higher than $\frac{1}{2}$. This is because calling $\boldsymbol{A}$ multiple times will increase the probability to higher value [21]. Variable ϵ is a constant not a function because polynomial growing error ratio would be equivalent to FPRAS [21].

It is not guaranteed that #**P**-complete problem has an FPRAS [8]. Problem which are *self-reducible* have a FPRAS. Requirements for self-reducibility are that problem is roughly-speaking in #**P** and that it's set of solutions for given instance x can be partitions to smaller instance of the same problem.

[§]Definition does exist where problem being in **NP** is not needed [19].