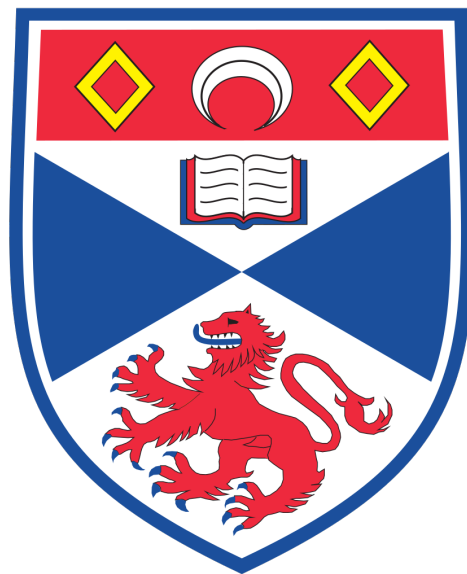

Automated creation of Nonogram puzzle game with Constraint Programming



University of St Andrews
SCHOOL OF COMPUTER SCIENCE

MSc Artificial Intelligence

Author:

Aigerim YESSENBAYEVA

Student ID: 090017799

Supervisor:

Dr. Christopher JEFFERSON

August 17, 2017

Abstract

People solve a lot of decision-making problems in everyday life using the main logic of Constraint Programming without awareness of it, such as timetabling, scheduling and planning. The most popular representation of the decision-making problem in academia is a puzzle game. This dissertation discusses in detail the design, implementation and analysis of models to solve and generate puzzles of Nonogram automatically using different approaches, such as reading white and black gridded images, random generation with any sizes and generation of instance with provided parameters from the user of the system. The system allows to check whether the instance of the Nonogram is solvable or not, as well as can iteratively change it to have only one valid solution. The system can generate different levels of difficulty of Nonogram by using conjectured algorithms that need to be assessed by computer and compared with the human players evaluations.

Declaration

I hereby certify that this dissertation, which is approximately <N> words in length, has been composed by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree. This project was conducted by me at The University of St Andrews from 09/2016 to 08/2017 towards fulfillment of the requirements of the University of St Andrews for the degree of MSc under supervision of Dr. Christopher Jefferson.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Aigerim Yessenbayeva

August 17, 2017

Table of Contents

	Page
List of Figures	v
1 Introduction	1
1.1 Constraint Programming	2
1.2 Nonogram	2
1.3 Project Objectives	3
1.3.1 Primary Objectives	3
1.3.2 Secondary Objectives	3
1.3.3 Tertiary Objectives	3
2 Context Survey	4
3 Ethics	6
4 Design & Implementation	7
4.1 Design of Nonogram	7
4.2 Logic of solving Nonograms	9
4.3 Constraint Modelling	9
4.3.1 Constraint Satisfaction Problem	10
4.3.2 Constraint Satisfaction Problem of Nonogram	11
4.4 Implementation of instances for the solver	15
4.4.1 Reading image files	15
4.4.2 Random generation	16
4.4.3 Generation of instance by prompting the user	16
4.5 Types of Nonogram instances	16
4.5.1 ways of the enhancement of the instance	17
5 User Study	18
6 Evaluation & Conclusion	28
6.1 Extent of completeness	28

TABLE OF CONTENTS

6.1.1	Primary Objectives	28
6.1.2	Secondary Objectives	30
6.1.3	Tertiary Objectives	31
6.1.4	Conclusion	32
A	User Manual	33
	Bibliography	35

List of Figures

Figure	Page
1.1 Graphic representation	2
5.1 Boxplot for entire data of all levels	24
6.1 Hard case instance: "Human and dog" it involves SACBounds tag of Minion	29
6.2 One solution instance that is generated by the system and displayed	31
A.1 Parameter file format example	34

Introduction

Constraint Programming (CP) is very powerful tool that is being increasingly used in academia and industry for solving combinatorial problems which are hard to express mathematically. It is widely researched in Computer Science, Artificial Intelligence and operational research [1]. Constraint modelling is every important and at the same is hard [2]. Puzzle games are very convenient in representation of **Constraint Satisfaction Problems**. By solving the problems using puzzle games the obtained knowledge could be applied to the other relevant to the problem areas such as traffic engineering [3, 4].

There are so many works and researches was dedicated to the puzzle games such as Sudoku, but not many related to the Nonogram [5]. The Nonogram was chosen as another example of the combinatorial problems, which need to be automatically generated by using constraint programming approaches. The initial step was to create a solver using constraints of the puzzle. For the implementation of the solver it was used Essence' constraint programming language. Then by using this constraint model (i.e. solver) create a system which will generate instances of the Nonogram. Solvable instances are the ones which has only one valid solution. Randomly generated instances could have more than one solution or could take too long to be solved. That happens because of several reasons which will be discovered and analysed. Based on these obtained knowledge it could become more clear to come up with some algorithms which will be used for generation of different levels of the puzzle.

1.1 Constraint Programming

Constraint programming is the way of programming where instead of doing a continuous sequence of task to be executed, it requires to write the properties of the problem, which is not need to be defined in particular order. The main idea of the Constraint Programming is to declare **variables**, define set of possible values for each variable (each set of values for the variable is called **domain**) and specify a relations between the variables, which are called **constraints**. **Figure** shows simple example of the Constraints[1, 6?].

Example 1:

Variables	Domains	Constraints
X_1	$X_1 \in \{1, 2, 3\}$	$X_1 > X_2$
X_2	$X_2 \in \{1, 2, 3\}$	$X_1 > X_3$
X_3	$X_3 \in \{2, 3\}$	$X_2 \neq X_3$

Table 1.1: Declaration of Variables, Domains and Constraints.

After applying constraints on the variables, it will remove the impossible values from their domains:

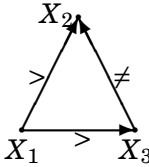


Figure 1.1: Graphic representation

Value elimination	Value assignment
$X_1 \in \{1, 2, 3\}$	$X_1 = 3$
$X_2 \in \{1, 2, 3\}$	$X_2 = 1$
$X_3 \in \{2, 3\}$	$X_3 = 2$

Table 1.2: Assignment of variables

1.2 Nonogram

As puzzle games can be used as good example of constraint satisfaction problems (CSP) there are many researches related to the puzzle games such as Sudoku, N queens problem and Nonogram [5].

Nonogram was invented in 1980s in Japan and was called as “Hanjie” [4]. Nonogram also known as Picross or Griddlers in different parts of the world [4, 6]. The classic version of the game is black and white version, where nonogram players must colour some cells of the $m \times n$ size grid to the black or leave them white accordingly a given numbers in the two sided matrices. The numbers and their orders in these matrices specifies the amount of continuously coloured cells in one block in the any given row or column. Each block should be separated by at least one

cell. The main goal of the game is to complete the colouring of the required cells of the grid by satisfying each number (called **clue**) on the sided matrices, and revealing the hidden image [6]. The Nonogram could be a colourful as well and then each clue specifies not just the amount of continuously coloured cells but also the colour of the block. For the this dissertation it was decided to use a classic one (i.e. black and white).

The detailed information about the rules and definitions of the game will be discussed in the design and implementation chapter.

1.3 Project Objectives

This project is the individual research work about the development of the puzzle game solver, called Nonogram, based on the concept of the Constraint Satisfaction Problem (CSP). The entire project was initilly outlined into the following three parts.

1.3.1 Primary Objectives

- (a) The main objective of the project is to automatically generate puzzles. The implemented software should be done based on the constraints of the game and solve any levels of the provided instances of the game. The instances should be provided by the text file.
- (b) Write a system which can generate random instances of the puzzle, which are valid but may not have an answer. Use the system from (a) to check if these instances are solvable.

1.3.2 Secondary Objectives

- (a) Write a guided system which generates instances of the puzzle, and tries to make solvable instances. This will recursively call the solver from part (a), and be guided towards correct and solvable instances.
- (b) Implement the game itself, which will use the system from 2.2(a) and 2.1(a) to produce problems and check they are solvable.

1.3.3 Tertiary Objectives

- (a) Investigate how to generate harder levels, and both create and solve levels of the puzzle faster.
- (b) Compare how the computer rates the difficulty of levels with how human players rate difficulty of levels.

Context Survey

There are some problems in the world which are difficult to be solved in mathematical representation. The Constraint Programming provides a very powerful tool to solve these combinatorial problems related to scheduling, distribution of traffic and logic puzzles in a very high level. The user provides the properties of the problem and the constraint solver find the solution by itself. Therefore, this approach of solving problems becomes more and more popular in the industry and academia. Due to this reason there are so many researches related to the creation of algorithms that can solve the constraint satisfaction types of problems more efficiently and there are so many different packages and libraries that can be used to solve these types of problems [2].

Due to the fact that most of the puzzle games not necessarily have only one way of solving it. There is always a possibility to develop a new approach to solve this game more efficiently. This will help to develop new algorithms that could be applied to any other real-life problems [7].

Such that the most popular example of combinatorial problem is Sudoku puzzle game. Which could be very easily solved by the constraint solver using `AllDiff()` function. Search algorithm will definitely take longer to implement and solve the game. Sudoku becomes very well researched puzzle game in constraint programming[5].

With the same reason this project is related to the writing of constraint solver of another puzzle NP-complete problem, called Nonogram. There are some previous researches related to the Nonogram solving algorithms such as reinforcement learning, constraint programming[4, 6] as well as search algorithms using different heuristics[8]. There is a detailed paper of comparing the

efficiency of different solvers based on depth-first search algorithm and the second is constraint programming. In that paper it was discovered that the solve of the Nonogram based on the depth-first search algorithm performs faster than constraint programming.

In this project there is no comparison of the several other algorithms except the ones which are based on the Constraint Programming. For instance in this project for the compilation of the Essence' constraint model we use Savile Row which runs the Minion solver. The more details of definition of Constraint Satisfaction Problem will be discussed in the Design and Implementation chapter.

Ethics

The project had a tertiary objective that was about about the comparison of how human players rate the levels of Nonogram against the rates made by computer. In order to conduct this study it was needed to involve some group of people, who would volunteer to participate in this study. The most of the participants were involved from the School of Computer Science in University of St Andrews. Essentially the idea of the experiment was to ask participants to solve the instances of the Nonogram. The players were provided with the instruction of the study as well as with the rules of the Nonogram. The users were provided with the same instances of the game in the same order with the same maximum limit of time to solve these instances. Each noted interval of time to solve one instance was the main purpose of the experiment. All volunteers were provided with the right to stop the experiment at any time when they wish. All users were kept anonymised. At the end of the experiment each participant was only asked about the rates of the puzzle levels and about the previous experience in this game. It was not that hard to involve the participants since the experiment was not involving any sensitive and private information that need to be analysed or unclosed. The only needed information from this study was the average time spent to solve particular type of instances by all participants.

To conduct this study it was necessary to sing an approval form from the Ethical committee of the School of Computer Science in the University of St Andrews.

Design & Implementation

This chapter will consider the Design and Implementation of the project in details. The entire project could be split to the several stages: (1) modelling of the constraint satisfaction problem for Nonogram; (2) implementation of the system which can produce instances for the model; (3) enhancement of the system to produce solvable instances; (4) development of algorithms that generate different levels of difficulty of the puzzle; (5) implementation of the visualization form of the puzzle to be printed on paper; (6) assessment of levels of difficulty by comparison of computer and human rates.

The system has been written mostly in Python language with the usage of the constraint model for Nonogram that was implemented in Essence'. As well as the analysis part was implemented in R (statistical programming language).

4.1 Design of Nonogram

The general description of the game was given in the introduction chapter. This part will talk in details about the black and white type Nonogram's properties and needed definitions for the puzzle, which will be used further in this chapter.

- Clues: the numbers which are provided in this board.
- Blocks: some amount of consequently coloured cells in any row or column.
- Solution Board: it is an essential gridded area, which need to be filled by the player when the puzzle is solved. Initially this grid is empty (i.e. all cells are white). The grid has m

rows and n columns (i.e. $m \times n$ size). The values of m and n could be equal or not, as well as they could be even or odd.

- **Top Matrix:** is the one of the given matrices, which is located at the top or at the bottom of the solution board. This matrix normally has clues, which restricts the number and length of the coloured blocks per column. The size of this matrix has the same number of columns as the solution board (i.e. n columns). For the number of rows it should not be the same as for the solution board. Since in the worst cast, when all clues on that columns are ones, the number of rows should be approximately twice less than m . As the number of rows of the solution board is m , then the number of rows for the top matrix should be $\frac{m}{2}$. However, if m is odd value then it should be $\frac{m}{2} + 1$. Thus the size is $(\frac{m}{2} + 1) \times n$ or $(\frac{m}{2}) \times n$ when it m is odd or even respectively. Example 4.1 demonstrates this case.

		1					1					1		
		1					1					1		
		1		1			1		1			1		1
1	1				1	1				1	1			
	1					1					1			
	1					1					1			

Table 4.1: Example of the Nonogram

In this example the size of solution board is 5×4 (rows = 5; columns = 4). Where the top matrix has the worst case when all the clues are ones. In 5 allowed cells we can place maximum only 3 blocks in one cell. This proves that the formula for the odd value is correct (i.e. $\frac{m}{2} + 1 = \frac{5}{2} + 1 = 3$).

- **Left Matrix:** is the second given matrix that is very similar to the top matrix. But it is normally located on the left or on the right side of the solution board and provides the restrictions to the rows of the solution board. The size of it has the same logic as top matrix but reverse (the number of rows equal to the number of solution board's rows and the number of columns of left matrix is almost twice less amount of columns): $size = m \times (\frac{n}{2} + 1)$ for odd n and $size = m \times (\frac{n}{2})$ for even n .

This design was used for the definition of the CSP. In the next section it will be considered in more details.

4.2 Logic of solving Nonograms

Nonogram is the logic puzzle game. In order to complete the Nonogram it is required to have some logical strategy to solve it faster:

- It is easier to start with any row or column, which either do not have any clues and allows to cross out all cells on that row or column; or to start with the biggest clues that are equal to the number of cells on that row or column, which allows to colour all of them.
- If the first case does not exist, then it is probably better to start with biggest provided clues first. The biggest clues are the most convenient in terms of finding out which cells need to be coloured for certain. If the value of the clues is bigger than 50% of the number of cell on the row or column, then it is possible to colour the intersection of possible blocks, the part of the row or column that will be coloured for certain. For instance (see Tables 4.2 - 4.4 below), we are given with one clue of value 7 on the 10 cells row/column. The seven size block can be located in four possible ways. Therefore, we can count 7 cells from the beginning and mark where it finishes and count 7 cells from the end and make a second mark, which will be a different cell. The cells between these two marks can be coloured to the black for sure. In this case it will be 4 cells in between (indexes 4-7).
- After finishing with the biggest clues it is better to move to the rows and columns, which already has some coloured cells. By counting the cells it becomes easier to solve even hard cases as well.

	1	2	3	4	5	6	7	8	9	10
7										

Table 4.2: Example 1: demonstration of "intersection"

	1	2	3	4	5	6	7	8	9	10
7										

Table 4.3: Example 2: demonstration of "intersection"

	1	2	3	4	5	6	7	8	9	10
7										

Table 4.4: Example 3: demonstration of "intersection"

4.3 Constraint Modelling

Constraint modelling is the process of converting the properties of the existing problem to the Constraint Satisfaction Problem(CSP)[1, 6]. This used as an input to the constraint solver to

search for solutions [9].

4.3.1 Constraint Satisfaction Problem

To implement a solver of the problem with Constraint Programming it is not needed to write the program in traditional way, which executes everything in the normal sequential order. In fact it is necessary to define and write the rules of the Nonogram [6] (i.e. CSP of Nonogram). Accordingly the definition of the CSP that was described in the introduction chapter, the CSP is formally defined as triple (variables, domains and constraints $\langle X, D, C \rangle$) [6, 10].

- Variables: $X = \{X_1, X_2, \dots, X_n\}$ in this case each variable of the Nonogram is the single cell on the solution board.
- Domains: $D = \{D_1, D_2, \dots, D_n\}$ every variable X has a finite set of values to which it could be assigned. This set is called domain. Each variable has its own domain. Variables may have the same domain or different. The values from the domain can be eliminated after applying constraint between the variables. When domain is left with one value it must be assigned to the variable.
- Constraints: $C = \{C_1, C_2, \dots, C_m\}$. One constraint is the relation between variables. It allows to eliminate some values from the domains of those variables that were involved in this constraint.

The CSP is considered to be solved when all the constraints are satisfied and every variables assigned to one value.

Constraint Satisfaction Problem can be solved in the following way:

- **Given:**
 1. A finite set of **decision variables**.
 2. For each decision variable, a finite **domain** of potential values.
 3. A finite set of **constraints** on the decision variables.
- **Find:**
 - An assignment of values to variables such that all constraints are satisfied.

?????ref

4.3.2 Constraint Satisfaction Problem of Nonogram

After getting the definition of the Constraint Satisfaction Problem, it is possible to define the CSP of Nonogram in detail.

Due to the fact that the goal of the game is the determination of which cells of the solution board can be coloured to the black and which need to be left white, the solution board will be considered as a set of variables that need to be assigned at the end of CSP. The total amount of the variables in the solution board is $m \times n$ (amount of rows and columns). The domain of each variable has maximum 2 values: white or black, in the modelling of the problem it could be replaced by binary values 0s and 1s (0 - white, 1 - black). Two matrices that provide the conditions of the game (i.e. top and left matrices) should be given to the solver as a set of parameters. Thus every single cell/clue of these matrices is a given variable. The maximum value of the clue in these matrices should not exceed the number of cells on any row or column. For instance if the solution board has size $m \times n$ then the domain of each cell of the top matrix varies between 0 and m , since the maximum number of rows is m . The same for the left matrix's cells that varies between 0 and n

$cell_{top(i,j)} \in \{0, \dots, m\}$, where i, j - are indices of the top matrix.

$cell_{left(i,j)} \in \{0, \dots, n\}$, where i, j - are indices of the left matrix.

As the Essence' prime is not a procedural languages and cannot remember the last updates to the 'object'. For instance it cannot save the current updates to the solution board and read them in the next iteration. For instance, is the clue is equal to 4 and the number of cells on that row or column is 5, then the block can start with index 1 or 2 (if the indexing starts with 1). In order to keep the index where the solver should start colouring cells in continuous form it was decided to use additional two matrices for top and left matrices' indices. These two matrices should have the same sizes as the top and left matrices. If the cell in the top and left matrices are empty then they should be empty on the additional matrices. Otherwise, it should have index where it starts on the solution board.

The general design discussion above will help to define the CSP using mathematical notations.

- **Given:**

\$parameter declaration:

`mainRow`: the number of rows of the solution board and left matrices
`mainCol`: the number of columns of the solution board and top matrices
`topRows`: the number of rows of the top matrix. This number is almost twice less than `mainRow`
`leftCols`: the number of columns of the left matrix
`topMatrix`[`topRows`][`mainCol`]: each cell of the matrix has domain: $\{0, \dots, \text{mainRow}\}$
`leftMatrix`[`mainRow`][`leftCols`]: each cell of the matrix has domain: $\{0, \dots, \text{mainCol}\}$
`topIndexMatrix`[`topRows`][`mainCol`]: each cell of the matrix has domain: $\{0, \dots, \text{mainRow}\}$
`leftIndexMatrix`[`mainRow`][`leftCols`]: each cell of the matrix has domain: $\{0, \dots, \text{mainCol}\}$

- **Find:**

\$variable declaration:

`solutionBoard`[`mainRow`][`mainCol`]: the board which need to be filled with only possible values 0s and 1s (domain = $\{0, 1\}$)

- **such that:**

\$constraints:

1. Since it was decided to define coloured cells as 1s and empty cells as 0s on the solution board. Consequently the sum of the clues in the column of the top matrix is equal to the sum of the sum of the ones in the same columns of the solution board.

$$\begin{aligned}
 &\forall col \in \{1, \dots, \text{mainCol}\}, \\
 &\sum_{j=1}^{\text{topRow}} \text{topMatrix}[j][col] = \sum_{i=1}^{\text{mainRow}} \text{solutionBoard}[i][col]
 \end{aligned}$$

2. The next constraint will help to define the additional matrices for keepint the indices of the beginning of the blocks on the solution board. Since these matrices have absolutely the same size and domain as top and left matrices it will help to define the empty cells and non-empty cells:

$$\begin{aligned}
 &\forall row \in \{1, \dots, \text{topRows}\}, \\
 &\forall col \in \{1, \dots, \text{mainCol}\}, \\
 &\text{topMatrix}[row][col] = 0 \rightarrow \text{topIndexMatrix}[row][col] = 0
 \end{aligned}$$

3. Since the additional matrices stores the indices of the beginning of each block, they must be in the ascending order. As the blocks of cells need to be coloured in the same order as the clues in any row/column.

Each block should have at least one cell in between in any given row or column. The last two constraints were combined in one expression. It happens quite often when the top matrix has needed number of clues and other cells are empty on this column. The possible mistake could be to compare next empty cell, which is zero with the previous non empty cell, which is greater than zero. So the constraint should specify first condition that both compared cells in the index matrix should not be zero elements, and only in this case it possible to make a proper comparison, which is constraint.

$$\begin{aligned}
 &\forall row \in \{1, \dots, topRows\}. \\
 &\forall col \in \{1, \dots, mainCol\}, \\
 &(topIndexMatrix[row][col] \neq 0 \wedge topIndexMatrix[row+1][col] \neq 0) \rightarrow \\
 &((topIndexMatrix[row][col] < topIndexMatrix[row+1][col]) \rightarrow \\
 &(topIndexMatrix[row+1][col] - topIndexMatrix[row][col] - topMatrix[row][col] \geq 1))
 \end{aligned}$$

4. The last constraint allows to colour cells in continuous blocks and in the right indecies. For instance if the clue is 3 and it starts at position 2 then it should colour only cells on positions: 2, 3, 4. But just by adding 3+2 would give index 5, which should not be coloured, since it is more than needed and could overflow the matrix.

$$\begin{aligned}
 &\forall row \in \{1, \dots, topRows\}. \\
 &\forall col \in \{1, \dots, mainCol\}, \\
 &(topMatrix[row][col] > 0) \rightarrow \\
 &\forall i \in \{1, \dots, topMatrix[row][col]\}, \\
 &(solutionBoard[topIndexMatrix[row][col] + i - 1][col] = 1)
 \end{aligned}$$

All these constraints need to be applied to the left matrix as well, but with symmetrical change of indecies and positions. These mathematical representation were translated to the Essence' constraint modelling language in the followig way:

```

language ESSENCE' 1.0
given mainRow: int    $num of rows
given mainCol: int    $num of cols
given topRows: int    $topMatrix[b][mainCol]

```

```
given leftCols: int    $topMatrix[b][mainCol]

given topMatrix: matrix indexed by [int(1..topRows), int(1..mainCol)]
of int(0..mainRow)
given leftMatrix: matrix indexed by [int(1..mainRow), int(1..leftCols)]
of int(0..mainCol)

find topIndexMatrix: matrix indexed by [int(1..topRows), int(1..mainCol)]
of int(0..mainRow)
find leftIndexMatrix: matrix indexed by [int(1..mainRow), int(1..leftCols)]
of int(0..mainCol)

find solutionBoard: matrix indexed by [int(1..mainRow), int(1..mainCol)]
of int(0..1)
such that

1. forAll col:int(1..mainCol).
   ((sum j: int(1..topRows).topMatrix[j,col])
    =(sum i: int(1..mainRow).solutionBoard[i,col])),

2. forAll row: int (1..topRows).
   forAll col:int(1..mainCol).
   topMatrix[row, col]=0 -> topIndexMatrix[row, col]=0,

3. forAll row: int (1..topRows).
   forAll col:int(1..mainCol).
   (topIndexMatrix[row, col]!=0 /\ topIndexMatrix[row+1,col]!=0) ->
   ((topIndexMatrix[row, col]<topIndexMatrix[row+1, col]) /\
    ((topIndexMatrix[row+1, col]-topIndexMatrix[row, col]-topMatrix[row, col])>=1)),

4. forAll row:int(1..topRows).
   forAll col:int(1..mainCol).
   (topMatrix[row, col]>0) -> forAll run:int(0..topMatrix[row,col]).
   (run > 0 -> (solutionBoard[topIndexMatrix[row, col]+run-1,col]=1)),
```

The Essence' constraint model was tested by passing different instances, which were provided in the .param files and writing them by hand. The obtained results which were shown in the binary representation was hard to determine whether the picture looks fine. For that it was

decided to use Ruby to visualise the output of the Essence model. Also at this stage it was possible to test the instances that could have more than one solution. The instances which are not solvable were producing zero solutions.

Even in thi stage of the implementation of the solver there were some interesting instances like "human and dog", which was running forever to solve *itwasspent2hourstowaitthesolvertosolveit*. However with the usage of the "SACBound" it takes seconds to solve te same instance.

4.4 Implementation of instances for the solver

The imaplementation of instances in the format of .param files was implemented in Python programming language. There are several approaches were used to make the instances of the solver:

1. Reading the black and white images
2. random generation of any size and different forms
3. to allow user of the system to specify the size of the solution board and the amount of the percentage which need to be coloured by the players later

4.4.1 Reading image files

The fact that Essence' was not allowing to see the images of the solved instance, it was decided to use the library of Python called "skimage.io" to read the image and assign the floating point value close to 1 of white colour and for the black it is close to 0. and for drawing the image it was decided to use the drawing application, in this case called "Paint2". Essentially drawing application would be suitable for this. At the beginning of creating the drawing space in the Paint it is needed to choose the size in pixels and start drawing on putting any squares. Once it was finished with drawing it is needed to save the image in the .jpg or .png format in the directory called "image_instances". The Python code will read it and return matrix of floating point numbers either close to 0 or 1, nothing in between, which need to be converted to the binary matrix. This matrix is filled solution board, based on which the system will count coloured blocks of cells and generate clues to the top and left matrices. Then all this matrices are written to the .param format file for the Essence' constraint and pass it to the Savile Row to solve it. It is more likely that the instance will have at least one solution or even more. The reason of that is that the top and left matrices are valid and generated based on the actual image, but it might have more than one solution. This is a different problem which will be considered late.

4.4.2 Random generation

In the random generation approach it randomly generates the size of the solution board. The range of the size varies between 1 and 25. The maximum size could be 25×25 and number 25 was chosen since it was noticed in the modelling stage that the instances with the size greater than 20 rows and 20 columns get very slow in solving them. The values for the cells of the solution board are also spread randomly. Therefore some instances which have the same size still might be different in solving time due to their intensity of the black cells and the location of the cells. Then creation of the instances should be created in the same way as it was describe earlier.

4.4.3 Generation of instance by prompting the user

This instances are generated by asking the user for the three parameters: number of rows of solution board, number of columns and the percentage of the coloured cells should be coloured out of the total amount of cells. The function for generation of the solution board counts the exact amount of the cells need to be assigned to ones. After that it randomly generates the coordinate of ones on the solution board and checked whether that cells is empty, if so it adds there one and increments the counter which counts how many ones were already placed. It iterates till every single one will be placed on the solution board. This function is needed in order to evaluate the level of difficulties of the Nonogram.

4.5 Types of Nonogram instances

There are several types of instances:

- instances which was inserted by hands to the .pram file and has some error inside. So that this conflict will not allow to solve it. Normally these instances just don't have solutions.
- instances which were generated by the system with respect to the solver of the Nonogram (Essence' model). All of these generated instances definitely has one or more solution for certain. However, among these valid instances it is still need to be mark to several categories:
 1. instances with one solution. They are 'ideal' instances
 2. instances which has more than one solution but less than 10 solutions. These instances can be enhanced iteratively by adding the pixel by pixel to the solution board. The algorithm which helps to determine the position of new cell should also be experimented and then if possible to evaluate it: (adding cells to the first solution after comparing this solution with another one and add the cells which are different from the first)

3. instances with extremely many solutions might take too much time to be enhanced. Therefore those, instance should be dropped and considered as unsolvable.
4. the same dropping approach should be applied to the instances which take too more than 20 seconds to be solved. Since if the computer takes more than 20 seconds to solve it, then it is more likely that the human player will not be able to solve it at all.

4.5.1 ways of the enhancement of the instance

All instances that has more than one solution but less than 10, can be tried to be improved. There are two approaches of improving it:

1. Even if the instance has more than two solutions, the system will pick just first two solutions and compare them. All the cells which are different from each other will be saved in the special matrix which has the same size, but different locations of some cells will be added to the first solution one by one. If after iterating through all the cells which were different the instance will still have more than one solution it should be considered as not solvable as well.
2. In this approach all possible solutions will be compared and the differences between all solutions should be also stored and be iterated by adding the new cells to the first solution. In this it showed that this approaches iterates so many times so it is more likely that the instance will be improved and return one solution.

User Study

To evaluate the difficulty of levels in the game it was decided to conduct the study with the users who should be able to solve the game of different levels. All levels of difficulty was assumed based on seen results of Savile Row Solver (implemented in Essence Prime).

The study involves multiple methods to make game harder in each level. In order to show that the conjecture is correct by using these methods it was needed to prove the following hypothesis:

H1: There are some parameters which can influence to the level of the hardness: size of the solution board, the amount of given information based on the percentage of the coloured cells on the board, and the last but not the least is the number of given clues on the top and left matrices.

you can find this in example of chapter 1: 1.1

Method*Participants:*

We recruited a sample of 12 participants from the School of Computer Science, University of St Andrews, where the number of males was 8 and females 4. Four of them had the experience on playing Nonograms before the study and the rest never played this game before.

Apparatus and Material

The generated instances of Nonogram using various algorithms for producing different levels of the difficulty was displayed and printed on the papers to conduct the user performance study.

Levels	Sizes (rxc)	Proportion coloured cells	Total number of clues
Level 1	7x7	70	24
Level 2	7x7	70	31
Level 3	7x7	50	20
Level 4	7x7	50	36
Level 5	7x7	70	48

Table 5.1: Shows the properties of each level of Nonogram

Overall there were 5 levels:

These 5 levels were generated to check the reliability of the assumed algorithms to generate different levels of difficulty of Nonogram measure by the time of solving them. In overall, we came out to three parameters which could be used to make the game more difficult:

1. Having the same size the same number of coloured cells on the solution board, but having different number of clues per rows and columns, might make the game harder. The program prompts the user to insert the size for the “solution board” and the percentage of the “coloured cells”, and can generate several instances. Some of them are with one solution and some of them with zero or many solutions. But we will consider only those which has only one solution. For the study with participants, in order to save the time for running through all possible combinations, it was decided to run through only 15 iterations and out of that range take two instances, which must have one solution, and one of them has the biggest amount of clues in top and left matrices and another is the smallest amount. The minimum amount of clues would be considered the easiest one and the maximum ones as the hardest from the list of generated instances with the same size and the same number of coloured cells. This idea came because it was thought that if the amount of clues are less than it might mean that the values in the top matrix and left matrix are very big, which probably makes easier to find the intersecting parts of the big block and colour them at first steps of the solving process. (human logic). Otherwise the game is assumed to be harder, when digits in top and left matrix get smaller and it makes harder to colour the first steps of the game.
2. Another assumed parameter which might make the game more difficult or easier is the percentage of coloured cells. If the amount of black cells on the solution board are small, then the generated instance will have less amount of information to solve the game, this makes the game harder to solve, and oppositely if the amount of the coloured cells are big, it makes the instance more full of information and easier to solve the game.
3. The last idea about the way to make game more difficult is the the size of the Nonogram. If the size of the game to increase and to keep the same proportion of the coloured cells

as for previous levels. It will demonstrate whether the instance takes more time to solve it.

In order to check whether these parameters are affecting to the level of difficulty and in which direction. Each of the five levels were thought in the way so that we could check only one parameter at a time.

- **Level 1** and **Level 2**, as it was mentioned before, they all has only one solution, which means they are solvable. The same size and the same proportion of the coloured cells (number of rows: 7, number of columns: 7, 70% of the solution board should be coloured), but they have different amount of clues. That will allow us to compare the influence of the amount of clues to the level of difficulty. Level 1 has the minimum amount of clues out of 15 instances (24 clues, including all clues from top and left matrices). Level 2 the largest amount of clues out of those 15 instances (31 clues)(see Table 5.1).
- **Level 3** and **Level 4** have one solutions, the same size (rows = 7, columns = 7) as Level 1 and Level 2, but they have less information given at the beginning. Since only 50% of the solution board will be coloured. Level 3 has the minimum amount of clues and Level 4 has the maximum Later for analysis we could compare Level 1 with Level 2 and Level 3 with Level 4 to compare the time spent to solve them with different number of clues. Since Level 1 with Level 3 and Level 2 with Level 4 have the same size and the minimum amount of clues, it is more likely to compare their times to check the influence of given percentage of coloured cells to the levels of difficulty as well.
- **Level 5** was generated to check how size of the board can change the level of difficulty. For this level it was taken the instance out of 15 with one solution and smallest number of clues, where the size is 10x10 and the proportion of coloured cells is 70%. This level can be compared with the level 1, which also has the smallest amount of clues out of its instances and 70% of coloured cells, but the size is smaller (i.e. 7x7).

Procedure:

At the beginning of the experiment the participant was explained with the rules of the game. All participants were given with identical instances and in the same order, assuming that they are allocated in the reasonable order with the respect to the levels of difficulty. In overall participant had to solve five instances and record the time spent to solve the puzzle. For each instance the participant was given with maximum 5 minutes to solve it. In total for the entire experiment each person should spend 25-30 minutes (including the explanation of the rules). To prevent the cases of spending too much time on solving one instance it was decided that the user will spend at maximum 5 minutes per instance, and in total should not spend more

than 30 minutes for the experiment including the explanation of rules and fast survey. Five minutes per instance was chosen, since in general an experienced user who knows the rules and can solve the instance completely can spend 2-4 minutes. In general the participant could have three possible outcomes of solving the instance:

- (i) successfully solved instance in less than 5 minutes;
- (ii) not completed instance due to time limitation (spent 5 minutes exactly);
- (iii) not completed due to found mistake, which is normally hard to fix (generally was happening in less than 5 minutes.)

In the cases when the participant solves the game in less than 5 minutes, he can record the obtained time and with restarting the timer starts the next instance. In the case of the second outcome the user just must stop the game and move to the new instance. In the third case the user just marks that he/she couldn't solve it and moves to the next level. Even if users will not complete the puzzle this will still allow us to know how difficult the game was since we still can see how correctly and how much of the board was solved in this maximum 5 minutes time range.

At the end of the game the participants were asked several questions how they would rate the instances in terms of their levels of hardness and whether they played this game before.

Results In total we collected around of **4 hours 13 minutes** data (25 minutes maximum per participant, the number of which was 12). Just some of the users completed the experiment faster.

After the experiment was conducted all the data was gathered to the Excel spreadsheet, where we written the index of the participant and the amount of time spent per each level. Since some of the levels were not completed and some of the levels failed for some users and the maximum limit per instance was 5 minutes. Thus the user could not exceed the limit 5 minutes, it was decided to denote those who just didn't complete it fully with 6 minutes and those who failed were denoted with 7 minutes. Since it shows that the level became so difficult enough so that the participant would need more time to complete it or even probably more time to fix the mistake. Therefore the table does not have any "Not Available" data. This approach was decided just for the sake of easiness. Thus all "Not Completed" and "Failed" states would be denoted also with longer time, 6 and 7 minutes respectively.

Therefore, in the filtered process we are still able to check when the time is equal either to 6 or 7 put them on different list from the other time less than 6 minutes, i.e. successfully completed lists.

Since we have got the information about previous experience in this game, we can separate people who played this puzzle game before from those who saw this game first time during the experiment. Despite the experience the user still should spend longer time on the more difficult levels. However, since we asked participants to stop the game despite they did not complete the level once the 5 minutes reached, we cannot rely on it since for some participants all levels were taking longer than 5 minutes, due to their lack of experience in this game. Moreover the people who had experience in this game has more consistent results, while the people with no experience might be improving in each level, which might skew the assumption of making game harder.

Therefore it is probably better approach to split the participants to two groups and analyse the people with more experience separately from all. And then for general analysis, it is still possible to consider all data together.

Initially it was split to two groups and checks how well the participants solves the game.

Accordingly all these pie charts in each level the proportion of successfully completed people in the list of experienced people is more than in the list of not experienced people.

Level 1 and Level 2 are the most easiest, since they have the smallest given size and the biggest percentage of coloured cells, which assumed to make game easier. Accordingly the list of experienced people, in both levels all users completed the game successfully. However, in the list of the non experienced people , it can be seen that the number of successfully completed the game in the Level 1 is less than in Level 2, which could be explained due to their learning skills.

Level 3 and 4 definitely gets harder in comparison with levels 1 and 2, since even in the list of the experienced people the amount of successfully completed is reduced. In the Level 3 it is 75% completed and 25% just did not complete it due to time limitation, while in the Level 4 the percentage of successfully solved game people reduced sharply to 25% only, 25% did not complete and 50% failed. This demonstrates our belief that the incrementation of amount clues makes the game a bit more challenging. However, for some reason the amount of failed in level 3 is 50% and in Level 4 is 38%, which probably tells that Level 3 is not easier than Level 4. Therefore, these pie charts does not clearly demonstrate that the theory of more clues makes the instance more difficult to solve.

In case if it possible to compare Level 1 with Level 3 or Level 2 with Level 4, then it clearly could be stated that the number the reduction of the coloured cells proportion makes game harder. Since the number of successfully completed people decreases in both columns, experienced and not experienced participants.

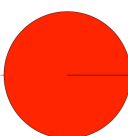
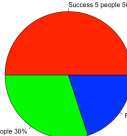
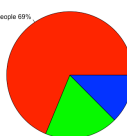
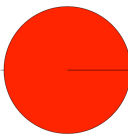
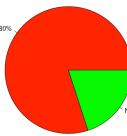
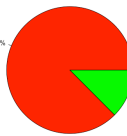

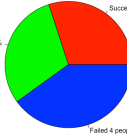

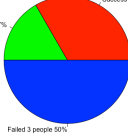

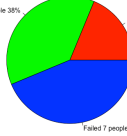
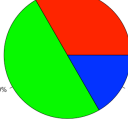
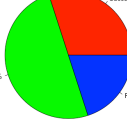
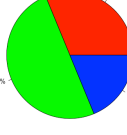
Level	Experienced	Not Experienced	Total
1	<p>Pie chart of Experienced users: Level 1</p>  <p>Success 6 people 100%</p>	<p>Pie chart of Not Experienced users: Level 1</p>  <p>Success 5 people 50% Failed 2 people 20% Not completed 3 people 30%</p>	<p>Pie chart of all users: Level 1</p>  <p>Success 11 people 69% Failed 2 people 12% Not completed 3 people 19%</p>
	<p>Pie chart of Experienced users: Level 2</p>  <p>Success 6 people 100%</p>	<p>Pie chart of Not Experienced users: Level 2</p>  <p>Success 8 people 80% Not completed 2 people 20%</p>	<p>Pie chart of all users: Level 2</p>  <p>Success 14 people 88% Not completed 2 people 12%</p>
	<p>Pie chart of Experienced users: Level 3</p>  <p>Success 4 people 67% Failed 2 people 33%</p>	<p>Pie chart of Not Experienced users: Level 3</p>  <p>Success 3 people 30% Failed 4 people 40% Not completed 3 people 30%</p>	<p>Pie chart of all users: Level 3</p>  <p>Success 7 people 44% Failed 6 people 38% Not completed 3 people 19%</p>
	<p>Pie chart of Experienced users: Level 4</p>  <p>Success 2 people 33% Failed 3 people 50% Not completed 1 people 17%</p>	<p>Pie chart of Not Experienced users: Level 4</p>  <p>Success 1 people 10% Failed 4 people 40% Not completed 5 people 50%</p>	<p>Pie chart of all users: Level 4</p>  <p>Success 3 people 19% Failed 7 people 44% Not completed 6 people 38%</p>
	<p>Pie chart of Experienced users: Level 5</p>  <p>Success 2 people 33% Not completed 3 people 50% Failed 1 people 17%</p>	<p>Pie chart of Not Experienced users: Level 5</p>  <p>Success 3 people 30% Failed 2 people 20% Not completed 5 people 50%</p>	<p>Pie chart of all users: Level 5</p>  <p>Success 5 people 31% Failed 3 people 19% Not completed 8 people 50%</p>
2			
3			
4			
5			

Table 5.2: The performance of each level for experienced, not experienced and all together

Level 5 for experienced people was the same, it was one person who still solved it in less than 5 minutes, but it shows that level 5 is easier than Level 3 and Level 4 since there is no people who could fail it. The rest just did not complete it. While in the Level 4, 50% of the experienced people just failed it. For non experienced people the percentage of successfully completed level 5 is more than in level 4 and the same as in level 3. However the number of failed people is much less than in level 3 and 4.

Also better to check the correlation between the levels of difficulty and three suggested parameters. For that it was decided to use simple T-test and visualise them on the boxplots, which will make them easier to analyse.

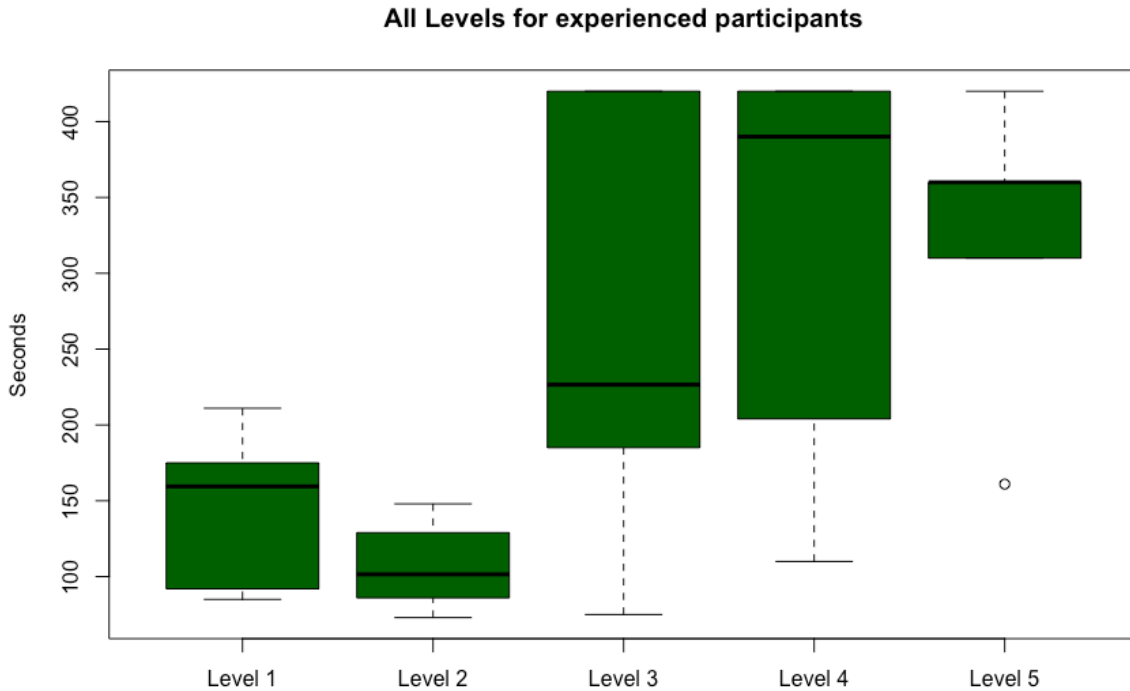


Figure 5.1: Boxplot for entire data of all levels

The boxplot demonstrates the time spent for each level of experienced participants. People spent more time in Level 1 rather than in Level 2, which probably means that the idea of adding clues is not really demonstrates the ability of making the game more difficult. Definitely the means of Level 3 and Level 4 are higher and there is more possibilities that some people even failed it or did not complete. Thus the idea of giving less information by setting less percentage of cells on the instances is more likely will make game more difficult. The last Level 5 has less mean value than for Level 4, which also makes sense since the Level 5 just should

take more time so solve than Levels 1 and 2, but still be easy solvable.

The graph 1 is more reliable, since it matches the opinions of the participants who say that Level 2 was easier than Level 1, and Level 3 is easier than Level 4. Thus it is quite confusing to proof that the more number of clues will make the game harder. Level 5 is easy in general as Levels 1 and 2 but just will take more time because of bigger size. Therefore the most of the people would change the order of the game to placing in this order (see Table 3). This order seems to be reasonable, since in Level 5 there is less people who failed the game. The most of them were failing in levels 3 and 4 since they are much harder and accordingly the survey people feel in level 3 and 4 that they most of the time guess randomly at some steps and traverse back to get right answer, and because of that they are failing.

#	Levels
1	Level 2
2	Level 1
3	Level 5
4	Level 3
5	Level 4

Table 5.3: The rate of difficulty of each level

Levels	Means <i>seconds</i>	Time <i>MM : SS</i>
<i>Level1</i>	237.9	3 : 57.9
<i>Level2</i>	169.8	2 : 49.8
<i>Level3</i>	324.6	5 : 24.6
<i>Level4</i>	353.9	5 : 53.9
<i>Level5</i>	337.8	5 : 37.8

Table 5.4: Total means per each level

Levels	Means <i>seconds</i>	Time <i>MM : SS</i>
<i>Level1</i>	237.9	3 : 57.9
<i>Level2</i>	169.8	2 : 49.8
<i>Level3</i>	324.6	5 : 24.6
<i>Level4</i>	353.9	5 : 53.9
<i>Level5</i>	337.8	5 : 37.8

Table 5.5: Means of experienced participants

Levels	Means <i>seconds</i>	Time <i>MM : SS</i>
<i>Level1</i>	171	3 : 57.9
<i>Level2</i>	169.8	2 : 49.8
<i>Level3</i>	324.6	5 : 24.6
<i>Level4</i>	353.9	5 : 53.9
<i>Level5</i>	337.8	5 : 37.8

Table 5.6: Mean of completed levels

T-test:

- Step1: $H1_0$ Level 1 and Level 2 are the same
 Level 3 and Level 4 are the same
 $H1_1$ Level 1 and Level 2 are not the same because different number of clues
 Level 3 and Level 4 are not the same because different number of clues
- Step2: $H2_0$ Level 1 and Level 3 are the same
 Level 2 and Level 4 are the same
 $H2_1$ Level 1 and Level 3 are not the same because different number of clues
 Level 2 and Level 4 are not the same because different number of clues
- Step3: $H3_0$ Level 1 and Level 5 are the same
 $H3_1$ Level 1 and Level 2 are not the same because different number of clues

T-test will be applied to the columns which need to be checked and if the p-value will be less than 0.05 it will allow us reject the null hypothesis. Otherwise, we will fail in rejection of Null Hypothesis.

T-test	p-value	Reject $H1_0$
Level 1 with Level 2	0.1195	Failed to reject $H1_0$
Level 3 with Level 4	0.4346	Failed to reject $H1_0$

Table 5.7: The rate of difficulty of each level

T-test	p-value	Reject $H2_0$
Level 1 with Level 3	0.1062	Failed to reject $H2_0$
Level 2 with Level 4	0.009892	Reject $H2_0$

Table 5.8: The rate of difficulty of each level

T-test	p-value	Reject $H3_0$
Level 1 with Level 5	0.002554	Reject $H3_0$

Table 5.9: The rate of difficulty of each level

To sum up it seems that for the Hypothesis # 1 we cannot reject the Null Hypothesis: that the levels are the same. In other words, we cannot reject the idea that they are the same (Level 1 with Level 2 and Level 3 with Level 4). Since the previous result of the boxplots does and the survey does not prove that bigger number of clues makes the game more difficult.

Table 6 allows us to reject the Null Hypothesis, which were about the percentage of the given cells. Therefore the proportion of the given cells obviously can influence to the level of difficulty of the game.

In the last t-test it was checked whether the size of the Nonogram influences to the level of difficulty. Accordingly the obtained results it seems that it is more likely to be true. Since the p-value is less than 0.05.

Evaluation & Conclusion

This chapter will draw a parallel line between the initial objectives of the project and the accomplished work. As well as it will cover essential observations during this research. It will assess the accomplished work and do some testing.

6.1 Extent of completeness

As it was mentioned in the introduction chapter the entire project was initially outlined to the 3 main objectives, where each of them has two subsections.

6.1.1 Primary Objectives

- (a) The implementation for the generic solver of any level of Nonogram was modelled in **Essence'** constraint programming language, which is using the tools of Constraint Satisfaction Problems called **Savile Row** and **Minion**. The instances of the model is provided by text files, and in the **Essence'** language all instance files will be having a `.param` extension. The solver can solve all valid instances, which has at least one valid solution. The instances with one solution will be solved and those instances which has many solutions will also be solved by the **Essence'** model. The invalid instances will not have a solution. However, it is not necessarily that it is needed to deal with the instances which has too many solutions. During the implementation it was noticed that some of the instances were having tremendous amount of solutions and could crash the entire computer. Another instances that have big sizes and very complicated distribution of coloured cells, might take too long to be solved. In those cases the problem is still solvable and has one solution

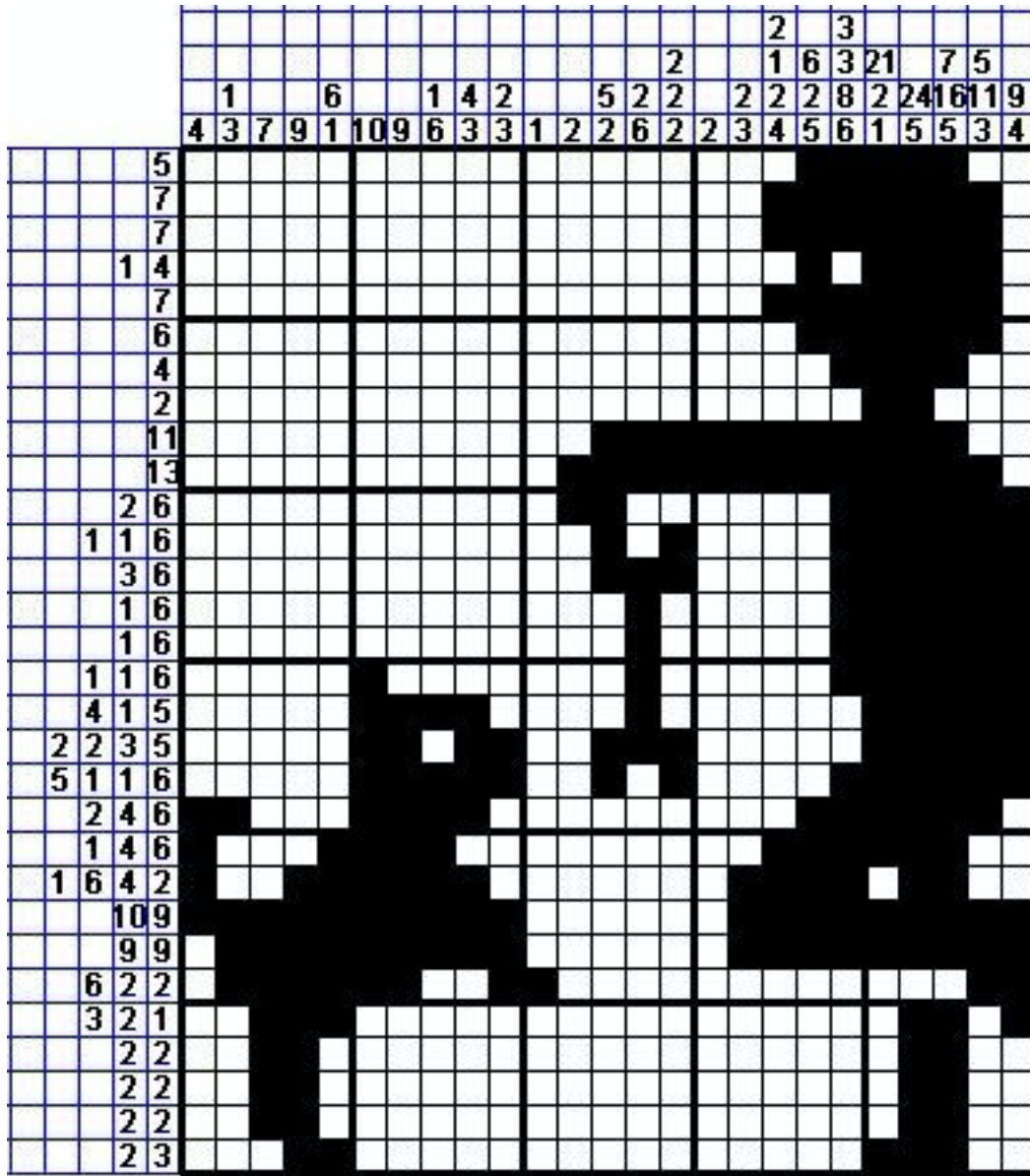


Figure 6.1: Hard case instance: "Human and dog" it involves **SACBounds** tag of Minion

but just probably taking too long to be solved. That cases were solved with the help of Minion's preprocessing, which was called **SACBounds** (shown in Figure 6.1).

- (b) Based on this solver it was possible to implement a system which produces all possible valid instances. These instances are tested with the Savile Row and based on that answers it was possible to check whether the instance has one solution or does not.

6.1.2 Secondary Objectives

- (a) To accomplish this objective the valid instance was executed by the Savile Row. Based on the produced solution file(s) it was possible to implement the algorithm which tries to minimise the number of solutions of those instances, which has more than one solution. To get the one solution instance we implemented 2 algorithms: (1) checking only 2 first solutions and based on those pixels which are different in these two solutions decides which pixels should be add and might help in reduction of the solutions; (2) another algorithm takes all solutions, where the amount is less than 10 and based on all the differences between all of them has more iterations, consequently more chances to get one solution instance. To compare these algorithms it was decided to run 100 iterations through both of them and collect the data to the .csv file and do the analysis on R. The testing was made in the following way with the results:

```
# calculates how many instances did not have a solution
nrow(filtered_data[filtered_data$SolverSolutionsFound == 0,])/2

# removes instances without solutions
filtered_data <- filtered_data[filtered_data$SolverSolutionsFound != 0,]

# separates data by algorithm used
alg1<-subset(filtered_data, filtered_data$UUID==filtered_data$UUID[1])
alg2<-subset(filtered_data, filtered_data$UUID==filtered_data$UUID[nrow(filtered_data)])

# shows how many times each algorithm had to run before all solutions for solvable instance
nrow(alg1)
# [1] 126
nrow(alg2)
# [1] 169

# splits data by instances
alg1 <- split(alg1, alg1$FileNames)
alg2 <- split(alg2, alg2$FileNames)

# finds the mean number of iteration on an instance each algorithm required to finish
mean(unlist(lapply(alg1, function(x) length(x$SolverSolutionsFound))))
# [1] 1.26
mean(unlist(lapply(alg2, function(x) length(x$SolverSolutionsFound))))
# [1] 1.69
```

```
# amount of instances that each algorithm was not able to solve
length(Filter(function(x) min(x$SolverSolutionsFound) != 1, alg1))
# [1] 39
length(Filter(function(x) min(x$SolverSolutionsFound) != 1, alg2))
# [1] 30
```

It can be concluded that the algorithms which compares only 2 solution is faster but produces less instances with one solutions than the second algorithm which checks all possible solutions (<10).

- (b) After checking that the instance has only one solution it was decided to implement a function which displays the instance in the classic representation. The user can save an print it on the paper later and play it. That is how the instances were printed for the participants of the user study. As well as the user of the system was allowed to decide which parameters is important to be inserted (see Figure 6.2)

			3	1		1	
		3	1	1	2	3	
1	1						
2	1						
	5						
1	1						
2	1						

Figure 6.2: One solution instance that is generated by the system and displayed

6.1.3 Tertiary Objectives

- (a) Based on the big amount of testing it became more and more apparent with possible properties of the game that could be used to implement the algorithm that will make the level of the instance harder to solve.
- (b) However it is not necessarily that computer and human uses the same logic to solve it. Therefore, they might rate the level of the Nonogram differently. This hypothesis was checked by the user study. This part was considered in more details in the User Study chapters.

6.1.4 Conclusion

The



User Manual

The project has several parts which can be run separately or could be run once by the system. In order to run the constraint model in Essence' firstly it is required to install Savile Row from here [11].

The constraint model file "*picross_solver.eprime*" is located in the main directory called "*PicrossGeneratorSolver*". To run and compile this model it is needed to provide a parameter file, which should have a top matrix and left matrix's values and the needed sizes of the solution board and two more parameters that specifies the number of rows of the top matrix and number of columns in the left matrix.

There are several ways how to run the same parameter file with the savile row here it will be tried to show all of them. In addition for more details it is possible to read it in the manuals of the savilerow [12]. These list of commands which could be useful to run when we are deadling with a lot of solutions or with difficults instances to solve. The following commands were used on Mac OS using `zsh` terminal.

1. `savilerow picross_solver.eprime some_file_name.param -run-solver` - in this case the name of the file for the `.param` will be changing. Instead of `some_file_name.param` put any other file name with the inside format given in the example of Figure A.1. It is essential that the name of the variables kept the same, since the model itself has the same variables as well. Otherwise it is required to change it everywhere. After running this command you should get one solution in the directory of the `.param` file.
2. `savilerow picross_solver.eprime some_param_file.param -run-solver -num-solutions`

10 - in this case the if the instance has more than one solution, then maximum 10 of them will be shown the others will be dropped. It is possible any needed number instead of 10. In order to get all possible solution the `-all-solutions` could be used instead. But it is important to be careful in case of if instance has tremendous amount of solutions may run for too long producing many solution files. Thus better to use `-num-solutions <n>`

3. `savilerow picross_solver.eprime instance.param -run-solver -num-solutions 10 -solver-options '-cpulimit 10 -varorder conflict -prop-node SACBounds'` - is the command that

```

1  language ESSENCE' 1.0
2
3  letting mainRow be 11
4  letting mainCol be 13
5  letting topRows be 4
6  letting leftCols be 4
7
8  letting leftMatrix be [
9  [6, 0, 0, 0],
10 [1, 1, 0, 0],
11 [1, 1, 1, 1],
12 [1, 1, 1, 2],
13 [1, 3, 0, 0],
14
15 [1, 1, 1, 3],
16 [1, 1, 1, 3],
17 [1, 4, 2, 0],
18 [1, 2, 0, 0],
19 [7, 0, 0, 0],
20 [0, 0, 0, 0]
21 ]
22
23 letting topMatrix be [
24 [0, 6, 1, 1, 1, 1, 1, 1, 1, 1, 7, 5, 3],
25 [0, 0, 1, 2, 2, 1, 1, 2, 2, 2, 0, 0, 0],
26 [0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
27 [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
28 ]

```

Figure A.1: Parameter file format example

Bibliography

- [1] F Rossi, P Van Beek, and T Walsh Elsevier.
Handbook of Constraint Programming.
2006.
- [2] Peter Nightingale, Ozgür Akgün, Ian P Gent, Christopher Jefferson, and Ian Miguel.
Automatically Improving Constraint Models in Savile Row through Associative - Commutative Common Subexpression Elimination.
pages 590–605, 2014.
- [3] Luis Quesada Alejandro Arbelaez, Deepak Mehta, Barry O’Sullivan.
A Constraint-Based Local Search for Edge Disjoint Rooted Distance-Constrained Minimum Spanning Tree Problem.
pages 31–46, 2015.
- [4] Frédéric Dandurand, Denis Cousineau, and Thomas R Shultz.
Solving nonogram puzzles by reinforcement learning.
- [5] Helmut Simonis.
Sudoku as a Constraint Problem.
Fourth International Workshop Sitges, pages 13– 27, 2005.
- [6] Ramas Zakarias Kavistavičius.
Nonogram solving algorithms analysis and implementation for augmented reality system.
page 54, 2012.
- [7] Mei-Huei Tang and Wen-Li Wasng.
Simulated annealing approach to solve nonogram puzzles with multiple solutions.
http://ac.els-cdn.com/S1877050914013015/1-s2.0-S1877050914013015-main.pdf?_tid=ebb57068-836e-11e7-a358-00000aacb35e&acdnat=1502989988_b75a1c16427c3ead3395e24b30d39ec8, 2014.
(Accessed on 08/17/2017).
- [8] Krzysztof R Apt.

Principles of Constraint Programming.

Annals of Physics, page 420, 2003.

- [9] Ian P Gent, Christopher Jefferson, and Ian Miguel.
Minion: A Fast Scalable Constraint Solver.
The 17th European Conference on Artificial Intelligence, 141:98–102, 2006.
- [10] Peter Nightingale Ian P. Gent & , Ian Miguel.
Generalised arc consistency for the AllDifferent constraint: An empirical survey.
2008.
- [11] Savile row constraint modelling assistant.
<http://savilerow.cs.st-andrews.ac.uk/>.
(Accessed on 08/16/2017).
- [12] Peter Nightingale.
Savile Row Manual 1.6.4.
pages 1–9.