**Solving Nonograms with AI**

**Final Report**

Hozyfa Mohammed Khair

Supervised by Magnus Wahlström

2023/24

Table of Contents

[Abstract 4](#_Toc163280553)

[1. Aims and Objectives 4](#_Toc163280554)

[1.1 Motivation 4](#_Toc163280555)

[1.2 Objectives 5](#_Toc163280556)

[1.3 Structure of Report 5](#_Toc163280557)

[2. Literature Review 6](#_Toc163280558)

[2.1 Constraint Satisfaction 7](#_Toc163280559)

[2.2 Backtracking 8](#_Toc163280560)

[2.3 Complexity, NP-hardness, and big O. 8](#_Toc163280561)

[3. Human-solving Techniques and Methods 9](#_Toc163280562)

[3.1 Logic Rules 9](#_Toc163280563)

[3.1.1 Numbers greater than half the length 9](#_Toc163280564)

[3.1.2 Pushing Off from Walls 10](#_Toc163280565)

[3.1.3 Inaccessibility 10](#_Toc163280566)

[3.2 Analysis 11](#_Toc163280567)

[3.3 Conclusion 11](#_Toc163280568)

[4. Project Development and Code 11](#_Toc163280569)

[4.1 Initial Approach 11](#_Toc163280570)

[4.1.2 8-queens and N-queens problem 12](#_Toc163280571)

[4.1.3 Initial Nonogram Solver 13](#_Toc163280572)

[4.2 Software Engineering Techniques 13](#_Toc163280573)

[4.2.1 Agile Methodology 13](#_Toc163280574)

[4.2.2 UML diagrams 14](#_Toc163280575)

[4.2.3 Tests and TDD 16](#_Toc163280576)

[4.2.4 Conclusion 19](#_Toc163280577)

[4.3 First Solver 19](#_Toc163280578)

[4.3.1 Pre-processing using Logic Rules 19](#_Toc163280579)

[4.3.2 Backtracking implementation 20](#_Toc163280580)

[4.3.3 Checking constraints 21](#_Toc163280581)

[4.3.4 Solve Puzzle Wrapper 22](#_Toc163280582)

[4.4 Final Solver 22](#_Toc163280583)

[4.4.1 Calculate Permutations 22](#_Toc163280584)

[4.4.2 Combination Backtracking 23](#_Toc163280585)

[4.4.3 Experimental rowMask 23](#_Toc163280586)

[4.5 Solver comparisons and benchmark 24](#_Toc163280587)

3. Designing a User Interface for The Solver

3.1 User Input

3.2 Solving Capabilities

3.3 Layout and Design

3.4 Conclusion

6. Current Code and Method

6.1 Pre-processing using Logic Rules

6.2 Backtracking Implementation

6.3 Checking constraints

6.4 Front-end

6.5 Plan Moving Forward

Project Diary

Bibliography

# Abstract

A nonogram is a puzzle played on a grid of pixels, the aim is to reveal a picture by colouring the pixels black or white. The restrictions are given in the form of row and column constraints, and satisfying these constraints produces the solution. This project aims to develop an algorithm capable of solving nonograms with AI techniques and then allow a user to use this algorithm to help them solve nonograms by providing hints and highlighting mistakes. Solving a nonogram is an NP-complete problem it is relatively easy to very if a solution is correct, while finding a solution from scratch will take exponential time, this project uses backtracking as its main method with constraint satisfaction techniques to increase efficiency. Preprocessing is used to reduce the total search space by finding cells that must be painted, forward checking will confirm what cells the next possible combination must have to be valid, and variable ordering will select rows with the most confirmed cells in order to arrive at the solution earlier. My algorithm can solve nonograms accurately and is capable of run times around 100ms on puzzles lower than 10x10, while past 15x10 the run-time increases exponentially to over 10 minutes, so it cannot be considered.

In Figure 1, a nonogram is depicted as a 7x5 grid. The constraints for each row or column are shown above and to the left of the corresponding row or column. The solution to the nonogram forms a pixel art representation of a cup. The interpretation of constraints is as follows: a single number, n, signifies the requirement for n consecutive pixels in that row or column with no gaps. In the case of multiple numbers, n and m, the pattern involves n consecutive pixels, followed by a gap of at least 1, and then m consecutive pixels, with no interruptions in between, resulting in a total of n + m pixels in that row or column.

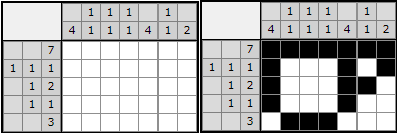


Figure 1 - A nonogram puzzle, and its solution.

# 1. Aims and Objectives

## 1.1 Motivation

Nonograms are a great example of constraint-satisfaction problems. A constraint is a logical relation among several variables, each taking a value in a given domain. A constraint thus restricts the possible values the variables can take [5]. A constraint-satisfaction problem is one where you are given a finite set of variables, a function which maps every variable to a finite domain, and a finite set of constraints. [5] The solution to a constraint-satisfaction problem is found by searching for the combination of values for each variable where no constraint is broken. These problems can be solved by search algorithms.

Many real-life problems can be represented as constraint-satisfaction problems. When considering scheduling and timetabling problems such as students taking an exam, you have a finite set of variables, i.e. students, a function which maps every variable to a finite domain, i.e. which students are registered under which course, and a finite set of constraints, i.e. no student can take two different exams at the same time, or one exam must be taken by all registered students at the same time. Satisfying all constraints will give you a timetable for every student to be able to take their exams without overlapping or clashing assignments. The vehicle routing problem is another good example, you have a finite set of variables, i.e. delivery trucks, a function that maps every variable to a finite domain, i.e. which location is within reach for the truck, and a finite set of constraints, i.e. truck cannot move between paths that are not connected or must take a connecting path to reach certain areas. Satisfying all constraints will give you the combination of paths taken to deliver all packages to customers.

In “Solving Nonograms by Comparing Relaxtions” by K.J Batenburg and W.A. Kosters [17], the comparison between nonograms and a job scheduling problem was made as such: A single row in a nonogram would be an employee or processor, the employees are ordered where better employees have higher row index numbers. Each employee has a series of work shifts and mandatory breaks, the columns correspond with consecutive time slots, and the column constraints define the number of employees needed for the different tasks during that time slot.

Creating a nonogram solver, which solves a constraint-satisfaction problem, will give me crucial skills in using and understanding simple AI techniques. I can extrapolate the knowledge gained and methods used to then solve other constraint-satisfaction problems whether they be in other games or real life.

## 1.2 Objectives

The primary objective of this project is to use AI techniques to develop a nonogram solver capable of accurately solving nonograms, with a secondary objective of building a user-friendly front-end GUI to allow users to interact with the solver. Through this, I will learn to use AI techniques and improve on the ones previously known by using them more efficiently and applying them in different situations.

The first phase of this project involved creating an initial nonogram solver that used simple backtracking. Then constraint-satisfaction techniques and depth-first search techniques were used to reduce the search space and increase efficiency. The first solver used preprocessing inspired by logic rules derived from human solving methods and forward checking by using constraints to calculate what the next cell must be.

The second task was to create a simple GUI which displayed a pre-solved nonogram when a button was pressed, allowing me to build on it later.

Moving on was to improve the first solver or create a better second solver by enhancing the constraint-satisfaction techniques and DFS strategies or implementing more. This solver had forward-checking and used preprocessing to implement variable ordering to prioritise solution paths with a higher chance of being correct, making it more efficient than its previous iteration.

Finally, the last task was to completely develop the front end by allowing users to interact with a puzzle grid to toggle cells and propose a solution to a nonogram, check their mistakes with a button in case they need some assistance while solving, press a button to see the fully solved answer, and load puzzles from a pre-made list.

## 1.3 Structure of Report

// Fill after completion //

# 2. Literature Review

Several researchers have studied how to solve nonogram puzzles, Bosch did so by translating the puzzle into an integer linear programming problem and solving it accordingly [1][11]. Batenburg proposed an evolutionary algorithm for discrete tomography (DT) [9] and modified it to solve nonograms [10]. However, it did not guarantee exact solutions because of the differences between discrete images and nonograms. Discrete images do not have multiple constraints in a single row/column so the algorithm could generate a solution that satisfies the DT but not the nonogram constraints, an example is shown in (figure 2) [1]. In 2004, Wiggers [13] proposed a depth-first search (DFS) algorithm, it worked by generating every possible combination for rows and columns and then matching them until all constraints were satisfied which is the basis behind my second solver. The DFS algorithm was rather inefficient and slow on large puzzles. C-H Yu’s approach to creating an algorithm that solves nonograms is closest to my initial attempt [1], he uses logical rules to deduce painted cells, then uses chronological backtracking to solve undetermined cells and arrive at the solution.

A screenshot of a crossword puzzle

Description automatically generated

Figure 2 A DT problem (a), and a nonogram (b)

Batenburg and Kosters [17] used relaxations of Nonogram problems that can be solved in polynomial time, and often can already determine the value of certain cells, and relations between pairs of certain pixels are also collected and combined into a 2-Satisfiability problem(2-SAT). A 2-SAT is a type of Boolean satisfiability problem where each clause consists of two literals joined by logical OR operators. These can be solved efficiently in polynomial time, unlike general satisfiability, which is NP-complete. They are very useful in problems where logical constraints need to be satisfied efficiently such as Nonograms. Iterating this procedure often allows one to solve the puzzle completely or determine a substantial part of the pixels. Using this method, they can solve simple nonograms, a nonogram that can be solved by hand often involving single rows that can be completely determined individually, in relatively fast times. When regarding complex or randomly generated nonograms it is often only capable of producing partial solutions, then requiring backtracking to find and complete the solution. The idea of reducing search space by using different solving techniques to determine certain pixels is one I have taken also.

Wiggers [13] compared the performance of genetic algorithms and a depth-first search algorithm on solving Nonograms. “A genetic algorithm uses many biological-derived techniques such as inheritance, natural selection, recombination and mutation.” [13], genetic algorithms are optimisation algorithms inspired by the biological processes of natural selection and evolution, used to find approximate solutions to optimisation and search problems. Comparisons showed that on small nonograms, DFS outperformed the genetic algorithm, while on larger puzzles the genetic algorithm sometimes had 10x less evaluations to solve. However, the genetic algorithm often got stuck in local optima requiring a reset to the puzzle. This was the first time I came across genetic algorithms and it provided a very interesting look into mimicking biology to achieve efficient logical constraint solving, it was out of the scope of this project and difficult to implement, however.

In “An Efficient Approach to Solving Nonograms” [2], a fast dynamic programming method is proposed for line solving, along with fully probing methods to determine pixels with similar functions to Batenburg and Kosters, and my own preprocess functions. These fully probing methods are also used to guide the next pixel selection when using backtracking arriving at a solution faster by taking promising routes. The algorithm performed very well, outperforming all programs collected in webpbn.com and won two nonogram tournaments at the 2011 Conference on Technologies and Applications of Artificial Intelligence (TAAI 2011, Taiwan).

Nonogram solvers found on the internet often use the method of calculating all possible rows and then finding a combination of them that satisfies the column constraints using backtracking, therefore arriving at a solution, various techniques are used to increase the efficiency such as the three I have previously mentioned, forward checking, variable ordering, and preprocessing. These are usually capable of solving small simple nonograms but take impractical amounts of time on larger puzzles. These commonly used methods are also what I will use to design my initial solver.

## 2.1 Constraint Satisfaction

A constraint satisfaction problem (CSP) can be defined by a set of variables X1, X2, …, Xn, where each Xi is associated with a domain, Di, which represents its possible values and a set of constraints C1, C2, …, Cn. Solving the CSP is to find any legal solution which assigns values to each variable while satisfying all constraints. [1]

In the context of nonograms, the variables correspond to the individual cells of the grid, and each variable Xi can take on a value from its domain, Di, which is 1 (representing black) or 0 (representing white). The constraints in the nonograms are defined by the row and column restrictions. Which specifies how many cells should be in which row and how many consecutive cells are in each group within a row or column.

Solving CSPs typically involves employing algorithms that explore the space of possible assignments to the variables while checking that each assignment satisfies all constraints. Backtracking search, constraint propagation, and local search are commonly used techniques to tackle CSPs.

Constraint propagation techniques are methods used to enforce constraints in constraint-satisfaction problems, by iteratively updating the possible domains of variables based on the constraints imposed by other variables. This helps reduce the search space and guide the search towards more feasible solutions. Arc consistency ensures that every value in a variable’s domain is consistent with the constraints of its neighbouring variables, i.e. that the potential assignment of black and white cells along a row is consistent with the constraints specified by another row or column. This is the basic method of eliminating assignments that violate the constraints by propagating constraints throughout the grid.

Forward checking involves immediately detecting and eliminating invalid cell assignments from the domain of unassigned cells. When a cell is assigned black, forward checking is to examine the constraints imposed on the cell by the row and column and removes any assignment that conflicts with the constraints from the possible combinations along that specific path. This helps reduce search space by pruning branches as early as possible.

By using these techniques, solvers for nonogram puzzles can effectively navigate the search space of possible cell combinations, gradually arriving towards solutions that satisfy all the row and column constraints. The combination of constraint propagation, backtracking search, and forward checking allows for efficient and systematic exploration of the solution space, allowing the accurate and timely resolution of nonogram puzzles.

## 2.2 Backtracking

Backtracking is a fundamental technique used in solving constraint-satisfaction problems, including nonogram puzzles. It is a type of depth-first search algorithm that systematically explores the space of possible solutions by recursively trying different values for variables until a solution is found or all possible solutions are exhausted.

The core method is to make a series of choices, one at a time, and proceed to explore the path until either a solution is reached, or it becomes clear that the current path cannot lead to a solution. If an assignment violates a constraint it will backtrack to the previous choice and attempt to pick a different option.

In the context of nonogram solving, backtracking involves iteratively assigning values, 0 or 1, to individual cells, starting from a certain point and proceeding row by row, in my case, or even column by column. At each step the algorithm will check whether the constraints are still satisfied, backtracking if any constraint is violated. Alternatively, you could backtrack through possible permutations of rows rather than individual cells, this allows you to confirm row constraints before backtracking is required, greatly reducing the search space, and improving efficiency.

Backtracking often exhibits exponential time complexity in the worst case, especially for large nonogram puzzles. It's necessary to use constraint propagation techniques to allow a solver to efficiently solve nonograms.

In summary, backtracking is a key component of nonogram solving being the main method by which trial and error is used to determine a solution, and when coupled with optimisation techniques can have very fast runtimes.

## 2.3 Complexity, NP-hardness, and big O.

Understanding the complexity of solving nonogram puzzles involves researching NP problems, a class of computational problems. At the core of NP problems is the concept of non-deterministic polynomial time, NP. An NP problem is one where the proposed solution can be verified or checked in polynomial time, given an input size of n [16]. This means with a proposed solution, we can verify whether it is correct efficiently. Nonogram puzzles fall within the class of NP problems, as it is easy to verify whether a proposed solution satisfies the row and column constraints in polynomial time.

Meanwhile, generating a nonogram puzzle that possesses a unique solution is more computationally intensive. This would fall under the class of NP-complete problems, these are all the problems X in NP problems for which it is possible to reduce some other problem Y to X in polynomial time. A problem, A, is NP-hard if there is an NP-complete problem B such that B is reducible to A in polynomial time, these do not have to be in NP. In Oosterman’s paper “Complexity and Solvability of Nonogram Puzzles” [16], he demonstrates the NP-completeness of the uniqueness problem by reducing the 3-dimensional matching problem (3DM) to nonograms. This reduction shows that ensuring a nonogram puzzle possesses only one solution, needed for puzzle creation- falls in the class of NP-complete problems.

Nonogram solving algorithms have time complexity that can be expressed using big O notation. Using the method of testing all possible combinations of a grid, let r = number of rows, c = number of columns, n = r \* c, the number of possible combinations would be 2 n, leading to a run time of O(2n), even on a small grid of 5x5 this would mean 225 = 33,554,432 maximum evaluations needed to find a solution. Of course, we do have row and column constraints and can determine squares that must be filled or can’t be filled greatly reducing the search space. Using the method of generating all row permutations first, we would have, O(2C) permutations generated for a single row and R rows so O(2C \* R), Then let P be the number of permutations for each row, big O (PR \* R) would be required to compare all possible combinations of permutations both cases are exponential but considering the same grid, calculating row permutations method would only need (55 \* 5) = 15625 evaluations, assuming there are 5 possible permutations per row, often there are only 1 or 2 depending on the constraints. Considering this, searching for solutions is a very computationally extensive goal and further highlights the importance of using various techniques to reduce the search space as much as possible and arrive at the solution sooner rather than being forced to compare all possible combinations.

An evaluation of time complexities is also done in “An Efficient Approach to Nonograms” [2]. Firstly, Batenburg and Kosters proposed a dynamic programming (DP) method for line solving, which runs faster and paints more pixels compared to traditional logical rules[17]. The time complexity for solving each line using this DP method is O(N2) in the worst case, where N is the size of the grid. The method to help paint more pixels was estimated as O(L7) where N = L x L. Comparatively the method proposed by Chen-Wu and others [2], their method for line solving had a worst-case O (kL) where the average number of integers in a single constraint is k. The fully probing methods they use achieve O(kl5). These are still before backtracking, but greatly help to reduce the search space and improve efficiency.

In summary, studying the problem size and hardness of nonogram puzzles reveals the inherent challenges associated with solving these logic-based puzzles. The exponential growth in possible configurations as the grid size increases highlights the difficulty of developing efficient algorithms. This understanding is crucial for seeking to create effective algorithms for solving nonogram puzzles.

# 3. Human-solving Techniques and Methods

Nonograms can be difficult puzzles to solve, especially larger nonograms which often have human solvers stuck on where they should start. In this section. In this section, I will discuss methods and strategies commonly used by humans to solve nonograms. These rules can also be adapted and used by my algorithm to preprocess the grid and streamline the solving process, therefore enhancing poo efficiency.

## 3.1 Logic Rules

Looking at the constraints provided for each row and column, you can often determine cells that must have pixels by considering all possible placements of cells in a row or column. The basics include highlighting rows and columns that have constraints that only allow for one possible combination. Additionally, after confirming the placement of a group painted square, if the constraint in its corresponding row or column says that the group is completed you can surround it with inaccessible squares to reduce the possible locations of other painted cells and ensure the gap requirement is kept between groups of cells. Following this after placing all groups of cells in determined places, you can fill the rest of the row or column with inaccessible squares to repeat the process.

### 3.1.1 Numbers greater than half the length

The first rule, numbers greater than half the length, also known as the superposition of extreme positions, is usually used at the beginning to determine as many boxes as possible. If the constraint on the line is single and makes more than half the length, you can paint squares in the middle. You must first place the sequence of squares starting at the extreme left and then starting at the extreme right, the place where the two sequences overlap must have painted squares.

A screenshot of a puzzle

Description automatically generated

Figure 5 Rows depicting numbers greater than half the length rule

If there are several numbers near the line, we can do the same by placing the sequences to start at the extreme left and then start at the extreme right. In this case, however, we must take care to only paint squares that overlap with the same group of squares rather than different constraints. We must also take into account the empty space to be found between multiple restraints.

A screenshot of a crossword puzzle

Description automatically generated

In the case where a single constraint takes up the entire row, this rule will also be able to fill in the appropriate cells relieving us from requiring a separate case where constraint == row length.

Figure 6 - Rows depicting numbers greater than half the length rule on multi-constraints

### 3.1.2 Pushing Off from Walls

The second rule requires a painted square, and for the constraint to be greater than the smallest distance to a wall. Through this, we can determine that any potential painted squares that go past our original painted square must also be painted. You can also use an inaccessible square in replacement of a wall, they serve the same function as long as you are able to determine what the closest constraint to that inaccessible square is and then push off the inaccessible square.

A group of rectangular objects with numbers

Description automatically generated

Figure 7 - Rows depicting pushing off from walls rule

### 3.1.3 Inaccessibility

If in a line there are painted squares and the remaining constraint, when placed at their extremes, cannot reach certain other squares. Then we can determine that there can be no painted squares in those unreachable squares.

A screenshot of a game

Description automatically generated

### 3.2 Analysis

These first three logic rules are crucial to human solvers when tackling a nonogram, the first rule can be used to exhaustion to gain many certain painted squares and can also be made use of after the third rule, inaccessibility, to further confirm the placement of painted squares.

These rules are also useful to my algorithm, the first and second rules can be used to preprocess the grid to reduce the total search space. The third rule can be used to prune branches early without having to attempt them.

Additionally, the third rule is also something that can be useful when automating a mistake checker and highlighter. When the user satisfies the constraints for a line, the program can automatically cross out cells that cannot have a painted cell in them to aid the user and improve their solving efficiency.

Alternatively, human solvers can look for contradictions in the grid which will lead them to areas where there cannot be a painted cell allowing them to more easily use the other rules to deduce painted cells.

## 3.3 Conclusion

In conclusion, solving using human techniques involves using logic rules to deduce the position of painted cells while also finding contradictions to reduce the number of possible cells. These techniques can be implemented into my code in the form of pre-processing or used as a heuristic for the variable ordering technique. Using the rules along with variable ordering will allow my algorithm to select the cells with the highest chance of leading to a possible solution which will increase efficiency as fewer branches will have to be travelled overall.

# 4. Project Development and Code

## 4.1 Initial Approach

The project's initial steps were to decide on the most suitable technologies I would want to use for my project, that are capable of dealing with the specified problem. When reading over the project specification, it was recommended to solve a problem called the 8-queens problem, as an early use of backtracking and recursion. The 8-queens problem which involves placing eight queens on a chessboard without any two queens threatening each other, served as an ideal scenario for exploring the fundamental concepts of backtracking and recursion.

When doing research and looking at existing implementations many were in Python, so I also began to use Python when developing my own 8-queens solver and went on to decide on Python as the main development language for the backend. My prior fluency with Python gave me the advantage of expediting the process of learning a new coding language or syntax, allowing for a rather simple decision.

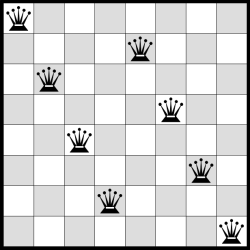
For front-end development, ReactJS was an easy pick since it provides a powerful framework for building a dynamic and interactive user interface. The component-based architecture allows me to modularise and reuse my HTML elements and components, improving the maintainability of my code. Additionally, React uses a virtual DOM and efficient rendering mechanism which is especially useful in the development stages removing the need to reload the page upon every change I make to the HTML, instead, all changes are immediately reflected skipping the need to re-run and reload the code.

Using Flask, a lightweight Python web framework, simplified backend development by giving a process of hosting the solver and enabling seamless communication between the front-end and back-end components. Using Axios, an HTTP client library for JavaScript, allowed me to use asynchronous (async) communication between the front end and back end without interruption.

The selection of Python for the back end, and ReactJS for the front end along with Flask to web host, gave me a good foundation to build up my project step-by-step and ensured maintainability along the entire application stack.

### 4.1.2 8-queens and N-queens problem

To begin my development process, I started by implementing a solution to the 8-queens problem. The 8 queens problem consists of having a grid depicting a chessboard of 8x8, the aim is to place 8 queens on the chessboard in such a way that none of the queens are threatening each other, this means they cannot be on the same row, column, or diagonal. The solution used backtracking and recursion to repeatedly attempt to place a new queen on the board in a square and then perform a check to see if it was safe, if it is safe, it then moves on to place the next queen, if not then the previous queen placement is undone and the code travels down other branches. It is possible for there not to be a solution if the size of the grid is too small for the number of queens to be placed.



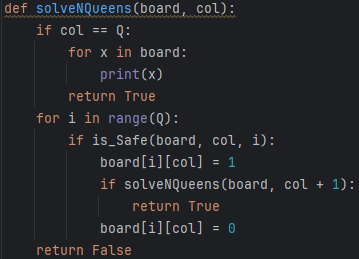


Figure 3 N-queens solving code

Figure 4 a solution to the 8 queens problem

Evaluating the time complexity of this algorithm requires looking at the backtracking and the method used to check the validity of a solution. The core of recursive backtracking which involves attempting possible configurations is as usual exponential, since a square can either have a queen or not, it gives us an approximate O(2N) where N is the number of queens, and the upper bound reduces significantly if the number of columns of the board is larger than the number of queens since there aren’t as many restrictions. The safety checks involve checking the rows, columns, and diagonals, and have a polynomial time of O(N) where N is the size of the board.

### 4.1.3 Initial Nonogram Solver

After implementing the N-queens solver, the next step was to modify it to solve simple nonograms. Initially, I set up the test containing the solution to an example nonogram. Then to modify the code I set the base case to be reaching the end of the grid, I decided to omit the for loop present in the N-queens algorithm and instead assume the square to be painted before checking the constraints, this prevented me from having to call the safe checking method multiple times and also allows easier access to prune branches in the future as the key branches of whether a cell is black and white is easily accessible since they are not embedded in each other. However, it ended up being detrimental as upon needing to backtrack, it would require an entire extra backtracking call rather than simply moving back once and continuing along possible solutions which led to an exponential increase in run-time. This was later fixed when implementing the row permutations method.

I modified the is\_Safe method to check if the constraints were satisfied. At first, the method simply checked if the number of pixels in a row matched the total of the constraints for the row, however, this quickly brought up the problem of multi-constraint rows which needed gaps in between them. Therefore, I decided to convert rows of pixels into their constraint representations to then compare them to the existing constraints allowing me to handle multi-constraint puzzles. The time complexity of this method was O(N) where N is the size of the rows but was relatively efficient and didn’t require major changes.

With the completion of those foundational steps, a basic nonogram solver was successfully created, and the aim from then was to improve its efficiency to allow it to tackle more complex nonogram puzzles.

## 4.2 Software Engineering Techniques

### 4.2.1 Agile Methodology

Agile methodology is an iterative approach to software engineering that has an emphasis on flexibility and focuses on producing a working piece of software in a short amount of time. This involves breaking down the project into smaller tasks and working on them in sprints, time-boxed periods in which a set of tasks will be set to complete. Usually, it is done by a team and at the end of a sprint the team will review their progress and take in any new information or changes to requirements based on feedback. This allows the project to stay flexible and responsive to customer requirements. One of the key parts of agile methodology is producing a prototype and demoing it. By producing a working demo, we can easily identify parts of the software that still require improvement and adapt to any change in requirements brought about by new techniques or changes in planning.

I have chosen to use agile methodology to develop my solver as it’s a project where my knowledge base is constantly expanding, this requires me to often change the requirements as I learn new possibilities and direct my project’s focus into the desired direction. By breaking down the project into small tasks I can ensure that progress is being made while also staying responsive to new information I gather during research. Furthermore, the iterative approach allows me to test and improve my algorithm as I work on the project rather than only benchmarking and testing at the end.

At the end of the first sprint, I had produced a skeleton of my final project in its basic forms, a working solver, and a basic GUI with a working button. With these, I was able to further refine my project until it became an acceptable program that meets all the requirements for the user and efficiency. Additionally, with a visual representation of my code, I could more easily envision what features a user would want to have.

Now at the end of the second sprint, I have a fully functional front end and the back end contains two solvers, although only one is put into use since it is superior in all cases, the front end is interactive and simple to use.

Overall, agile methodology is the best approach for developing my nonogram solver as it allows me to be flexible and respond quickly to new information throughout the development process.

### 4.2.2 UML diagrams

UML diagrams are a great way to define the requirements of the project and begin the design process. Here I’ve made three diagrams, a class diagram, a sequence diagram, and a use case diagram.

**UML Class Diagram**

A class diagram will represent the hierarchy of the code and the layout of methods, attributes, and responsibilities. It’s a useful tool while designing to appropriately assign responsibilities to methods and classes which allows reusability.

The current class diagram is rather simple, made up of three main components: Model, Controller, and View. The model holds the solver which will handle computations, the controller will bridge the gap between the model and the view and handle calls to the solver, and the view will display the results and changes. With ReactJS, the JavaScript file can act as both the controller and the view since it can return the HTML required to make up the view and directly access the data in the controller.

The design of this class diagram has been fully achieved, small changes were made to the Nonogram wrapper and is often just passed as the grid since accessing it is simpler that way.

A screenshot of a computer program

Description automatically generated

**UML Sequence Diagram**

The sequence diagram shows how the actor and classes will interact with each other in a process over time.

The sequence diagram starts with loading the program. The default puzzle is retrieved and displayed for the user to begin solving immediately should they wish to. After this, the user has the options to: Solve the Entire Grid, Solve the Next Step, Toggle Cell, show mistakes, and Reset the Grid.

A diagram of a workflow

Description automatically generated with medium confidence

### 4.2.3 Tests and TDD

Test-driven development (TDD) is a style of development where the developer will first write a test for a method before writing the method itself. Once this test passes, they will write another test and repeat the process until the method has all its required functionality. This allows the developer to clearly define the responsibilities of the method and never write unnecessary code.

The beginning stages of my development did not use TDD due to the nature of the project. Since the problem is a constraint satisfaction problem writing a test case that checks the solution would be considered as solving the problem, or maybe by technicality you can say my entire project is TDD. Moving forward, now that I have made a constraint checker, I can begin to use TDD to ensure that the values returned are correct with boundary inputs or that an appropriate error message is returned with incorrect input types.

The latter stages of my development used unit tests in the form of PyTest, which helped ensure individual parts of the code were producing the desired result and allowed me to zero in on any bugs occurring.

#### Unit Tests

Implementing a robust unit testing framework, such as Pytest, helps streamline the testing process and ensure comprehensive test coverage. Writing unit tests for individual components of the solver can help detect bugs, and regressions and validate functionality.

##### Test ‘preprocessGrid’:

Tests the ‘preprocessGrid’ function correctly preprocesses the grid based on row and column constraints, given an empty row and constraints, can the method find the correct cells that can be determined? First, the constraint and multi-constraint rule of numbers greater than half the length is tested. This method tests the case where there is one constraint for a single row greater than half the length, and the second test compares multiple constraints in a single row, i.e. more than one group of painted cells. Both are checked whether they return the correct row with squares marked by asserting they're equal to a premade row.

A screen shot of a computer program

Description automatically generatedA computer screen shot of a program code

Description automatically generated

##### Test ‘rowToRestriction’

Tests the ‘rowToRestriction’ function that turns a given row into its form as a constraint to allow element-wise comparison with the constraint for the row. The ‘colToRestriction’ method functions the same with only a change in where the temporary row is pulled from so no tests have been written for it.

A screen shot of a computer program

Description automatically generated

##### Test ‘is\_Safe’

The next test tests the ‘is\_Safe’ method which takes a row and column that has been turned into a restriction and compares it with the current row and column restrictions respective to its location. It simply performs comparisons between elements to determine whether the current configuration is safe.

A screen shot of a computer program

Description automatically generated

##### Test ‘solvePuzzle’

The solvePuzzle function was the wrapper for the first solver, this test simply prints the contents since if it reaches the end then it has either found a solution or there does not exist a solution as long as all previous tests have passed.

A computer screen shot of a code

Description automatically generated

The next few tests correspond to the newer solver which uses the permutations of rows as targets of backtracking.

##### test ‘calcAllPermutations’

This test determines whether the correct number of permutations will be generated for a row of a certain size, the correct number is 2N where N is the size of the row. Since there would be N spaces and each space could be 1 or 0.

A computer screen shot of a program

Description automatically generated

##### test ‘calcValidPermutations’

This test determines whether the permutations can be filtered correctly to only contain permutations that will be transformed into the correct constraint. It uses the permToRestriction which is another version of rowToRestriction. The test prints its contents out, as long as they are not empty then the test can be considered to have passed as long as the rowToRestriction test also passes.  
A computer screen shot of numbers and letters

Description automatically generated

##### test ‘findValidCombination’

This test is the final test that uses the previous methods to create the final solver, it calculates the permutations, filters them, and then backtracks the combinations until it finds a combination of rows that satisfies all column constraints. It then prints out the contents since I can determine by eye whether the pattern is correct.  
A computer screen shot of a code

Description automatically generated

#### Manual user interface and functional testing

In addition to unit testing, I performed manual testing by interacting with the front end. By directly interacting with my GUI I can validate its functionality, usability, and adherence to requirements. I get real-time feedback on the application’s behaviour from an end-user’s perspective.

I can evaluate subjective criteria such as colours, alignment, and symmetry. Functional criteria such as ensuring all elements that need to be displayed are displayed, checking buttons perform their correct function, submitting forms such as row and column constraints and seeing the changes take effect in the GUI.

Manual user interface testing complements automated testing by allowing exploratory testing and experiencing subjective features such as user satisfaction and aesthetics.

#### 

### 4.2.4 Conclusion

While developing my nonogram solver, I’ve used various software engineering techniques to shape the project’s direction, ensuring its structural integrity and enhancing its quality. Agile Methodology was the cornerstone, providing the framework for iterative development, allowing me to break down the project into manageable tasks, work on them in focused sprints, and review progress.

UML diagrams provided a visual blueprint for the project’s architecture, enabling me to allocate responsibilities effectively and ensure reusability. Even if the project’s design changed the UML still served as a guiding torch.

Furthermore, adding Test-Driven Development practices, although only in later stages, improved the projects robustness and reliability. Writing unit tests using Pytest ensured individual components of the solver function as intended, detecting bugs early and validating their functionality. Manual user interface and functional testing helped me assess usability, functionality, and aesthetics of the front end.

In conclusion, the application of software engineering techniques has been crucial in driving the development of my nonogram solver. The techniques have facilitated effective planning and design and contributed to the project’s robustness and success. Moving forward, I will continue to leverage techniques like these to further refine the solver itself or other projects in the future.

## 4.3 First Solver

The first solver began by pre-processing the grid using the Logic Rules described previously. The algorithm will then take the pre-processed grid and begin using backtracking. It will start at the first unassigned cell and set it to black, it will then check if the grid still fits the constraints, if it does it will continue to the next variable, if not it will set it to white and check the constraints once again, if the constraints are satisfied it moves on, if not then this branch does not contain the correct solution and is pruned.

### 4.3.1 Pre-processing using Logic Rules

The first rule ‘Superposition of extreme positions’ also referred to as ‘numbers greater than half the length’ applies when the constraints take up more than half the length of a column or row. By assuming the consecutive squares start at the edge we can determine that the squares in the middle must be painted black.

A diagram of a crossword puzzle

Description automatically generated

Figure 8 - a row from a nonogram showing logic rule 1

The method preprocessGrid (shown in Figure 9) takes 3 variables the grid, the row constraints, and the column constraints. Then for each row, if the total of the constraints is higher than half the length of the row we can determine where painted cells will be by looking at the overlap between the end of the sequence starting from the left and the start of the sequence starting from the right. Previously this did not work on multiple constraints such as [3,4] in a row of 10, but with some additions it can now consider multiple constraints.

A computer screen shot of a program code

Description automatically generated

Figure 9 - method preprocessgrid

### 4.3.2 Backtracking implementation

After the grid has been pre-processed, it is then passed on to the main backtracking loop which uses recursion to perform a depth-first search with forward checking. The method starts by checking the base case, that the current row is equal to one more than the number of rows that exist and that the grid satisfies all constraints. If this is not the case, it continues to assign the coordinates of the next cell to be evaluated, then assigns the current cell to be a black square and checks the constraints, if the constraints are not violated the next cell is passed to the recursive call of the method. Otherwise, the current cell is assigned to be a white square and the process repeats. Forward checking is done by consistently checking whether the constraints of the current cell are violated before pruning the entire branch, reducing the need to travel down unsuccessful branches. Through this implementation, the code can also determine whether a nonogram has no solution as the code will return false and empty the grid. One iteration of the code tried to prioritise confirmed cells but the design of the first solver was such that if the cells prior to a confirmed cells are incorrect it would require more backtracking calls to reach the confirmed cell with a suitable configuration, but the original would have reached in the same if not less time. The design of the code was not yet suitable to variable ordering, unlike the second solver.  
  
A computer screen with white text

Description automatically generated

Figure 10 - the solvepic function which handles backtracking

### 4.3.3 Checking constraints

Checking whether constraints have been violated is done by converting the current state of a row or column into its restriction. So, a row with two black squares, a gap of any size, and then two black squares would become [2, 2], this would then be compared against the current constraint to see if they match. If they match the method will return True allowing the main method to continue down that branch, if not it will return false, and that possible branch is pruned. This method has not changed much in the current or previous solver, only that the currentRow[L] count is incremented rather than adding the value in the specified grid cell to prevent adding too much for a confirmed cell with the value of 2.

A computer screen with white text and numbers

Description automatically generated

Figure 11 - a function showing converting a row into its restriction

### 4.3.4 Solve Puzzle Wrapper

The final section of the first solver, is the solvePuzzle wrapper. This function takes the initial inputs containing the grid details, the column and row constraints. Calls the preprocess grid function, and then calls the backtracking function and assigns solved to whether there is a solution or not and solvedGrid to the array that is returned. The solved variable is used to make decisions within the backtracking algorithm and also decides whether to return an empty grid or one filled with 0s which would imply there is no solution.

A computer screen shot of a program code

Description automatically generated

## 4.4 Final Solver

The final solver uses the method of generating all possible permutations of rows, then filtering them to only the rows that satisfy the row constraints. Finally, using backtracking to search for a combination of rows that satisfy all the column constraints.

### 4.4.1 Calculate Permutations

This function takes two inputs, the row constraints, and the size of a row which is the number of columns and returns the valid permutations for each row that satisfies the row constraints. It uses the product function of the itertools library which takes two iterable objects and produces its cartesian product. For example, if we had two lists a = [1, 2], b = [x,y], it would produce [1,x], [1,y], [2,x], [2,y]. Using product, we can calculate all possible permutations for a row of a certain size.

A screen shot of a computer code

Description automatically generated

Then we filter the permutations to only leave the valid permutations, ones that satisfy the row constraints for their respective rows, by using permIsSafe method. This method calls permToRestriction which functions the same as rowToRestriction, converting a permutation into its constraint form. Then the constraint is checked to see if it matches the constraint for the row. If it does, then the permutation is valid and is placed into the valid\_permutations list.

A computer screen shot of a program code

Description automatically generatedA computer screen shot of a code

Description automatically generated

### 4.4.2 Combination Backtracking

The next step is to use backtracking to search for a combination of rows that satisfies all the column constraints, there by arriving at a solution that satisfies all row and column constraints.

The method find\_valid\_combination takes valid permutations, the number of rows, and the column constraints. It consists of two embedded methods, is\_valid\_combination, and backtrack. is\_valid\_combination takes a combination of rows, which does not have to be complete, and determines whether that combination is valid, so far if it is not complete, and returns true or false respectively. It confirms whether a combination is valid using the combinationIsSafe method, this method extracts the columns from each row and turns it into its constraint form using permToRestriction, then checks if the column constraints are satisfied. If the column constraints are satisfied, then the current combination can continue to be used or is the solution if max number of rows has been reached.

Backtrack is the main backtracking method for this solver, its base case of row\_idx == R, is the case where the last row has been passed, therefore it returns the combination of rows. Otherwise for each permutation in a row, it attempts to add that permutation to the current combination. If is\_valid\_combination returns true then that permutation is added to the combination and row\_idx is incremented to go to the next row. If there is no combination that satisfies the column constraints, then the code will backtrack until it returns None.

A screen shot of a computer program

Description automatically generated

### 4.4.3 Experimental rowMask

rowMask is an unfinished function which took the number rows currently in combination, the value of the current column constraint, the column id, and columns. It created a row mask which determined what value the next column would need to be in the next row depending on the value of the previous row. For example, if the constraint said there needed to be a group of 3 consecutive painted cells in the column, and the previous row has that column painted. Then the next row must also have that column painted or the constraint would be violated. This helps reduce the possible search space in a local search and is a form of variable ordering, prioritizing rows with a higher chance of having the solution. Unfortunately, I was unable to implement this function in time and the implementation I had caused an infinite loop or wasn’t able to solve puzzles that it was previously able to solve so it had to be left as incomplete.

A computer screen shot of a program code

Description automatically generated

## 4.5 Solver comparisons and benchmark

Using PyCharm, IntelliJ’s IDE for python, I can profile the run times of each solver and receive a snapshot of how many times each method is called allowing me to compare and analyse the two solvers, focusing on their runtime performance and method call frequency.

Considering a puzzle of size 9x5, titled lock, profiling the initial solver gives us a run-time of 180ms. A large amount of this is spent on calling colToRestriction, rowToRestriction, and is\_Safe. These methods while essential for constraint checking, are currently too inefficient and the number of times they are called on is excessive leading to an inflating of the run time. While 180ms does not seem like a lot, for an algorithm that has exponentially increasing time complexity it sets a bad base line.

In contrast, the second solver only takes 4ms, quite a good runtime, with no excessive calls to functions that I have written, improving past 4ms would not be a matter of making single functions more efficient but improving the backtracking overall. This is quite a good baseline, but it helps that the puzzle contains three rows that only have a single possible permutation, these reduce the search space greatly.

A black and white crossword puzzle

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Considering a puzzle size of size 10x10, titled insect. The first solver takes an entire 10 minutes and 21 seconds to arrive at an answer. Showing how the first solver is much more inefficient than the second. When taking a look at the function call counts, rowToRestriction, and is\_Safe, are each called over 100 million times. The excessive and inefficient calls are highlighted when working on a grid of size 10x10. Further showing the importance of improving the efficiency of method themselves or the frequency at which they are called.

The second solver is capable of solving it within 80ms, a large increase but still instantaneous to a human. When looking at the profiler for the second solver, we see that permToRestriction is being called upwards of 10000 times, this is not because there are 10000 different row permutations but because even after calculating the restriction version of a row or column it is not saved to be used again thus must be recalculated many times, this leaves an area that must be improved upon if the solver is to solve more complex puzzles, this would fall under the technique of memoisation, storing results that are computed frequently.

A crossword puzzle with black squares

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

In summary, the comparative analysis shows the superior efficiency and scalability of the second solver over its predecessor. By minimizing redundant method invocations, the second solver demonstrates significant performance gains, paving the way for tackling more complex puzzles with enhanced speed and accuracy. However, further enhancements such as caching or memoisation are essential to optimize the solver's performance and address scalability issues for larger puzzles.

## 4.6 Front End

ReactJS was used for front end, it’s capabilities in dynamically updating the HTML made for good ease of use in development

The first steps to develop the front end were rather simple, the initial requirement was only to have a grid that displayed empty, then solve a grid that was entered in the source code by pressing the solve button which would send a request to the back end and return with the array for the solved grid. This simple GUI served its purpose but is missing features crucial for the final solver.

A screenshot of a puzzle

Description automatically generatedA grey rectangular object with a black line

Description automatically generated

**6.4 Front-end**

The front end of the code is made using React. This allows me to dynamically generate the grid and solution with ease, also allowing for future implementations of step-by-step solving and providing hints and clues. The current front end is just a simple page with a button ‘solve’ which will cause the solution to the nonogram stored in the code. Currently, you must change the code to affect which nonogram is being solved and displayed. The grid size and restraints will be sent to the Python code, which is hosted locally using Flask, through an Axios request and the response is sent as a list of lists of integers, list[list[int]].

The grid is generated using the grid-container class of CSS and then populated by mapping the solved grid to the cells which are made using the span divider. Depending on whether there is a 1 or a 0, the element is assigned to the style of black-cell or white-cell determining its displayed colour.

A screen shot of a computer code

Description automatically generated

Figure 13 html script for nonogram solver app

**6.5 Plan Moving Forward**

The current progress of the project is behind compared to the project plan; however, the delay can be solved using the buffer week I provided for myself. Immediate objectives are still to improve the efficiency of the solver to quickly handle large puzzles.

The plan for term 2 is to complete the functionality of the solver for users and allow them to interact with the front end by entering their nonograms and being assisted through the process of solving them. Along with implementing an alternate way to input puzzles, such as reading from a file or image. Benchmarking the algorithm and adding a different algorithm to compare is also one of the aims.

To reiterate my project plan, in the second term, I will first begin developing an alternative way for the user to enter their puzzle on top of the capability to type it in. This will either be in the form of machine vision or reading and interpreting from a file. After this, I will begin adding crucial functionality to the solver, such as stepping through puzzles, undoing the previous move, highlighting mistakes, importing your puzzle, and resetting the grid. Following this will be the development of a second algorithm, then benchmarking the two algorithms and comparing them against each other and existing algorithms.

**3. Designing a User Interface for The Solver**

The user interface aims to design a simple but helpful interface that allows the user to interact with the Nonogram solver easily. The UI should have features for inputting the user’s nonogram puzzles by defining the grid size and constraints, solving, and displaying the solution, and providing hints or clues to the user to assist them in solving the puzzle.

**3.1 User Input**

Inputting a user’s own puzzle will begin by inputting the limits of the grid, this will most likely be displayed above the grid in an easily visible position. The constraints will be able to be inputted above the columns and to the left side of the rows.

The user should be able to toggle between painted squares and empty squares, with the added option of placing a cross in a square to help remind them that it is impossible for a cell to be there. The last option could potentially be automated as is found in many other nonogram solvers [14].

In the future the user should also be able to import their own puzzle from an external source, this will most likely be a file or an image.

**3.2 Solving Capabilities**

A solve button should be available which will start the solving algorithm and display the solution. There should also be options to provide step-by-step solving that also allows the user to take control at any point to complete the solution. Additionally, there should be step-back, and step-forward buttons to allow the user to determine their own pace for solving the puzzle.

A clear grid or reset grid button will also be useful to allow the user to quickly clear the grid to start again with the puzzle. It’s important that any modifications the user made to the restraints or grid size, stay so that they do not have to re-enter their own puzzle each time they want to solve.

The solver should also bring up an error if a puzzle that is entered has no solution, or if one of the constraints is violated and there is no solution with the current placement of painted squares.

A user manual containing common human-solving methods will also be useful for first-time solvers or those unfamiliar with Nonograms.

**3.3 Layout and Design**

The elements of the solver should be set out to guide the users along inputting their own puzzle or simply starting to solve the default puzzle displayed. The grid will be found close to the top and centred where many users' eyes will look.

Using recognisable icons can also assist the user in navigating the user interface, such as a plus button to enter a new nonogram, or a back button to undo the previous move.

**3.4 Conclusion**

In conclusion, designing a user interface for a nonogram solver involves combining functionality with design to allow all kinds of users to easily interact with the solver. By implementing intuitive user input, visual feedback and displaying the system status, the user won’t feel lost or stuck while using the solver.

**Project Diary**

**Week 1-2 (October 1 - October 15, 2023)**

* Research nonogram-solving algorithms and AI techniques.
  + Backtracking and constraint satisfaction problems

Oct 14: Read papers ‘An Efficient Approach to Solving Nonograms’ and ‘An Efficient Algorithm for Solving Nonograms’, both are very closely related to my project, especially the second paper which uses methods that seem simplest to implement.

* Set up a development environment.
* Implement a solution for the 8-queens problem using backtracking.

Oct 15: PyCharm was chosen as IDE. JetBrains has many modules that are helpful when coding. Implemented a solution to the 8-queens problem, then modified it to become the N-queens solution.

**Week 3-4 (October 16 - October 31, 2023)**

* Develop a simple GUI to display nonogram puzzles.
* Adapt the backtracking algorithm for solving a simple nonogram.

Oct 25: Implemented nonogram solver using backtracking and recursion, quite basic and requires more methods to improve its efficiency. It takes too much time on large nonograms with many constraints per row/column.

* Research constraint satisfaction techniques.

Oct 27: Techniques such as forward checking, variable ordering and pre-processing have been selected to be implemented first. They are on the simpler side but will greatly improve efficiency.

**Week 5-6 (November 1 - November 15, 2023)**

* Develop a simple GUI to display nonogram puzzles.

Nov 1: Simple UI implemented to display a nonogram when solved, React chosen as front-end and Python code will be hosted locally and then send data using Axios and Flask

**Week 7-8 (November 16 - November 30, 2023)**

* Test the system on various nonogram puzzles.

Nov 30: Tests for different small nonogram puzzles done.

* Prepare for interim report.

**Bibliography:**

1. Yu, CH., Lee, HL. & Chen, LH. An efficient algorithm for solving nonograms. Appl Intell 35, 18–31 (2011). <https://doi.org/10.1007/s10489-009-0200-0>
2. Wu, I. C., Sun, D. J., Chen, L. P., Chen, K. Y., Kuo, C. H., Kang, H. H., & Lin, H. H. (2013). An efficient approach to solving nonograms. IEEE Transactions on Computational Intelligence and AI in Games, 5(3), 251-264.
3. Dandurand, F., Cousineau, D., & Shultz, T. R. (Year). Solving nonogram puzzles by reinforcement learning.
4. . -C. Wu et al., "An Efficient Approach to Solving Nonograms," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 5, no. 3, pp. 251-264, Sept. 2013, doi: 10.1109/TCIAIG.2013.2251884.
5. Guide to constraint programming - <http://ktiml.mff.cuni.cz/~bartak/constraints/intro.html>
6. Nonogram AI solver - <https://chihyulai.com/nonogram-ai/>
7. Brailsford, S. C., Potts, C. N., & Smith, B. M. (1999). Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research, 119(3), 557-581. <https://doi.org/10.1016/S0377-2217(98)00364-6>
8. Dechter, R., & Frost, D. (1999). *Backtracking algorithms for constraint satisfaction problems* (Vol. 56). Technical Report.
9. Batenburg, K. J., & Kosters, W. A. (2012). On the difficulty of Nonograms. *ICGA Journal*, *35*(4), 195-205.
10. Batenburg KJ (2003) An evolutionary algorithm for discrete tomography. Master thesis in computer science, University of Leiden, The Netherlands
11. Batenburg KJ, Kosters WA (2004) A discrete tomography approach to Japanese puzzles. Proceedings of BNAIC, pp 243–250
12. R. A. Bosch, “Painting by numbers,” Optima, vol. 65, pp. 16–17, 2001
13. Wiggers, W. (2004). A comparison of a genetic algorithm and a depth-first search algorithm applied to Japanese nonograms. Faculty of EECMS, University of Twente. Presented at the 1st Twente Student Conference on IT, Enschede, June 14, 2004.
14. Japanese crosswords – [www.nonograms.org](http://www.nonograms.org)
15. Nonogram solving methods - <http://www.pro.or.jp/~fuji/java/puzzle/nonogram/knowhow.html>
16. Oosterman, R.A. (2017) Complexity and solvability of Nonogram puzzles. Master's Thesis / Essay, Science Education and Communication.
17. Batenburg, K.J., & Kosters, W.A. (2008). Solving Nonograms by combining relaxations. Vision Lab, Department of Physics, University of Antwerp; Leiden Institute of Advanced Computer Science, University of Leiden, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands.