Solving Nonograms with AI
Final Report

Hozyfa Mohammed Khair

Supervised by Magnus Wahlström

2023/24

Word Count: 12543

Solving Nonograms with AI

Table of Contents

Solving Nonograms with AI

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.

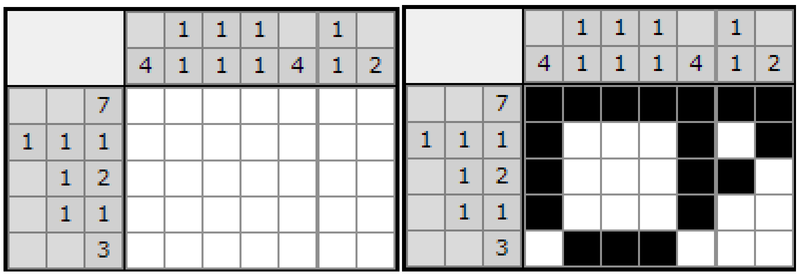


FIGURE 1 - A NONOGRAM PUZZLE, AND ITS SOLUTION.

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." ("Constraint Programming Based Algorithm for Solving Large ... - Springer") 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

This report will first perform a literature review on previous nonogram solvers, assessing their time complexity, difficulty, and success. Then we will look at how a human would solve a nonogram to gain better insight into what the algorithm is achieving and providing an introduction to any beginners. Assessing the project’s development process, initial approach, software engineering techniques, and key algorithms will be done next followed by the development of the front end. There will be a critical analysis and discussion on the overall project’s success. Finally, there is a section on professional issues at the end.

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.

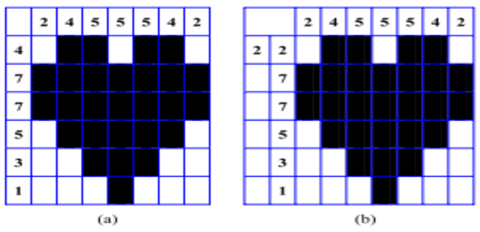


FIGURE 2 A DT PROBLEM (A), AND A NONOGRAM (B)

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 $X_1, X_2, \ldots, X_n$, where each X_i is associated with a domain, D_i, which represents its possible values and a set of constraints $C_1, C_2, \ldots, C_n$. 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 X_i can take on a value from its domain, D_i, 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 $O(2^n)$, even on a small grid of 5x5 this would mean 2^{25} = 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, $O(2^C)$ permutations generated for a single row and R rows so $O(2^C * R)$, Then let P be the number of permutations for each row, big O $(P^R * R)$ 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 $(5^5 * 5)$ = 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 $O(N^2)$ in the worst case, where N is the size of the grid. The method to help paint more pixels was estimated as $O(L^7)$ 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(kl^5). 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.

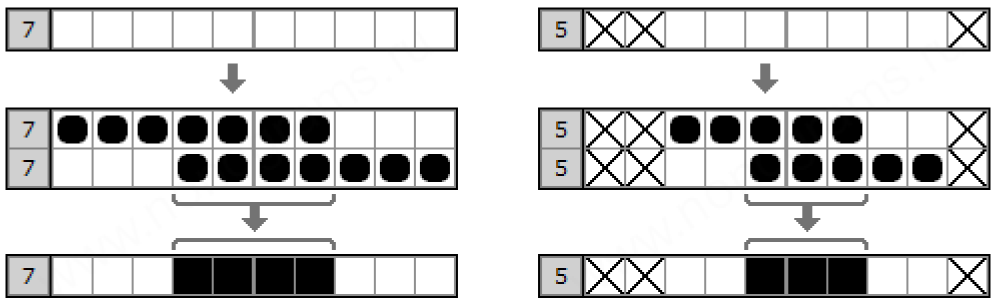
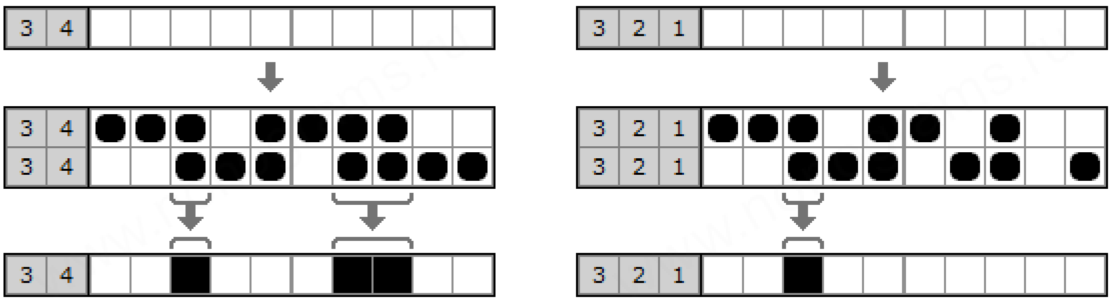


FIGURE 3 ROWS DEPICTING NUMBERS GREATER THAN HALF THE LENGTH RULE

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.



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 4 - 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.

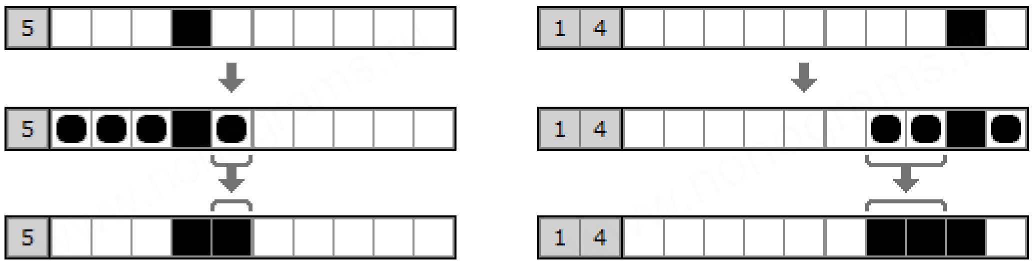
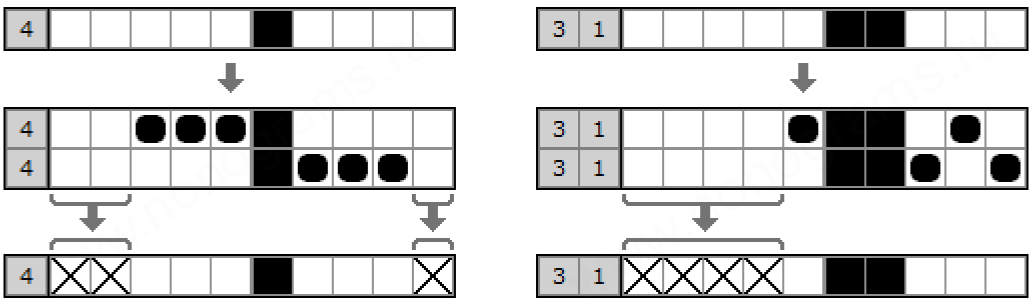


FIGURE 5 - ROWS DEPICTING PUSHING OFF FROM WALLS RULE

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.



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.

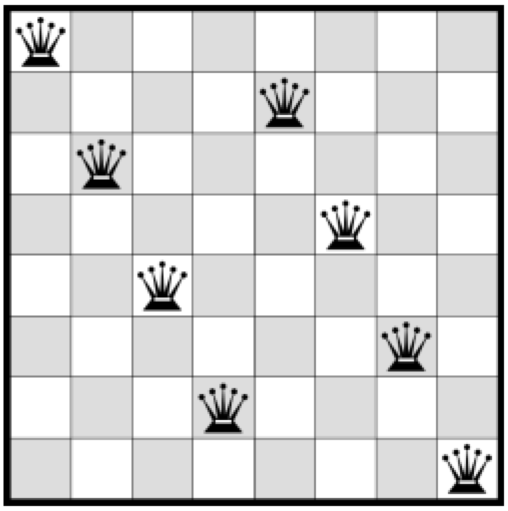
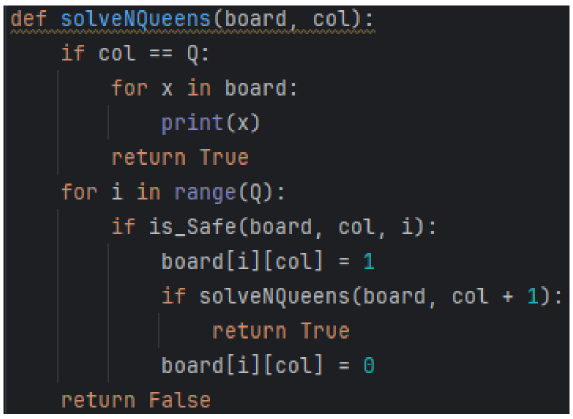


FIGURE 7 A SOLUTION TO THE 8 QUEENS PROBLEM



```
def solveNQueens(board, col):
    if col == Q:
        for x in board:
            print(x)
        return True
    for i in range(Q):
        if is_Safe(board, col, i):
            board[i][col] = 1
            if solveNQueens(board, col + 1):
                return True
            board[i][col] = 0
    return False
```

FIGURE 6 N-QUEENS SOLVING CODE

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(2^N) 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

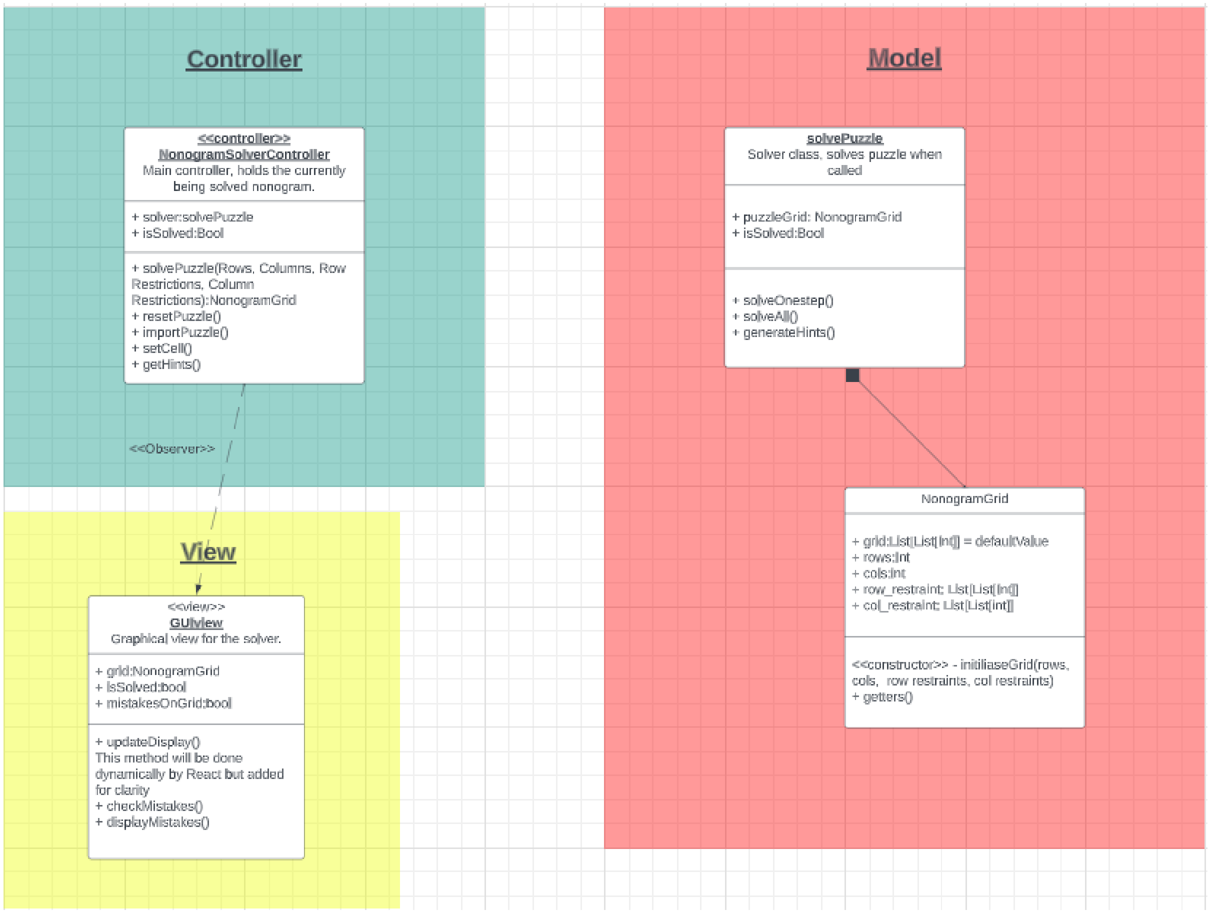
A class diagram will represent the hierarchy of the code and the layout of methods, attributes, and responsibilities. It's a useful tool while designing to appropriately assign responsibilities to methods and classes which allows reusability.

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

Solving Nonograms with AI

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.

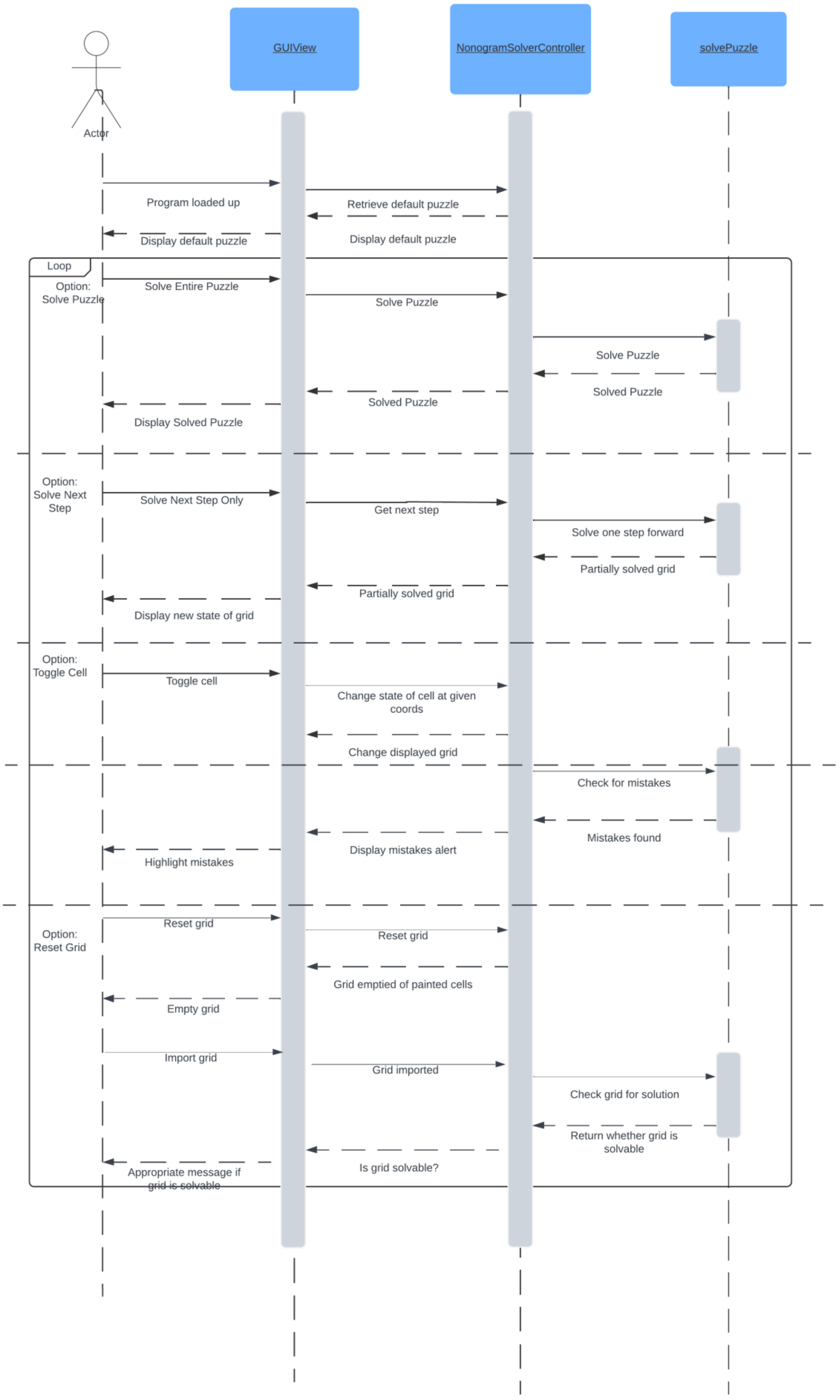

Controller
<<controller>>
NonogramSolverController
Main controller, holds the currently being solved nonogram.
+ solver:solvePuzzle
+ isSolved:Bool
+ solvePuzzle(Rows, Columns, Row Restrictions, Column Restrictions):NonogramGrid
+ resetPuzzle()
+ importPuzzle()
+ setCell()
+ getHints()
<<Observer>>
View
<<view>>
GUIview
Graphical view for the solver.
+ grid:NonogramGrid
+ isSolved:bool
+ mistakesOnGrid:bool
+ updateDisplay()
This method will be done dynamically by React but added for clarity
+ checkMistakes()
+ displayMistakes()
Model
solvePuzzle
Solver class, solves puzzle when called
+ puzzleGrid: NonogramGrid
+ isSolved:Bool
+ solveOnestep()
+ solveAll()
+ generateHints()
NonogramGrid
+ grid:List[List[int]] = defaultValue
+ rows:Int
+ cols:Int
+ row_restraint: List[List[int]]
+ col_restraint: List[List[int]]
<<constructor>> - initiliaseGrid(rows, cols, row restraints, col restraints)
+ getters()

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.

Solving Nonograms with AI



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.

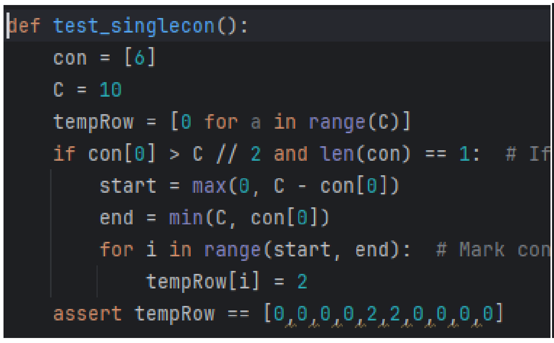
The latter stages of my development used unit tests in the form of PyTest, which helped ensure individual parts of the code were producing the desired result and allowed me to zero in on any bugs occurring.

Unit Tests

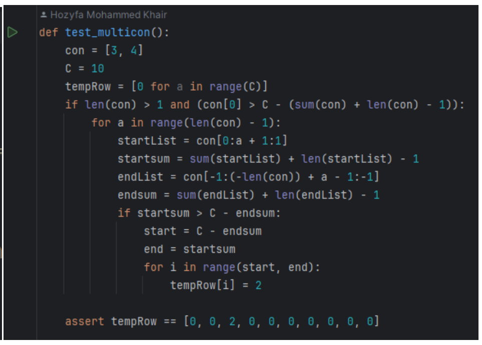
Implementing a robust unit testing framework, such as Pytest, helps streamline the testing process and ensure comprehensive test coverage. Writing unit tests for individual components of the solver can help detect bugs, and regressions and validate functionality.

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.



```python
def test_singlecon():
    con = [6]
    C = 10
    tempRow = [0 for a in range(C)]
    if con[0] > C // 2 and len(con) == 1:  # If
        start = max(0, C - con[0])
        end = min(C, con[0])
        for i in range(start, end):  # Mark con
            tempRow[i] = 2
    assert tempRow == [0,0,0,0,2,2,0,0,0,0]
```

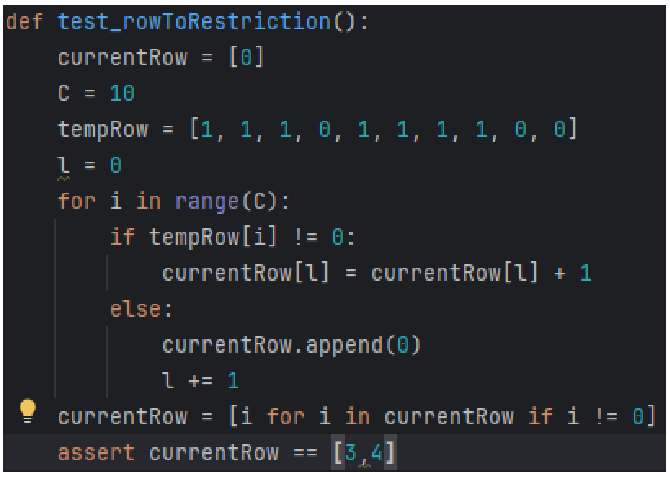


Hozyfa Mohammed Khair

```python
def test_multicon():
    con = [3, 4]
    C = 10
    tempRow = [0 for a in range(C)]
    if len(con) > 1 and (con[0] > C - (sum(con) + len(con) - 1)):
        for a in range(len(con) - 1):
            startList = con[0:a + 1:1]
            startsum = sum(startList) + len(startList) - 1
            endList = con[-1:(-len(con)) + a - 1:-1]
            endsum = sum(endList) + len(endList) - 1
            if startsum > C - endsum:
                start = C - endsum
                end = startsum
                for i in range(start, end):
                    tempRow[i] = 2

    assert tempRow == [0, 0, 2, 0, 0, 0, 0, 0, 0, 0]
```

Test ‘rowToRestriction’

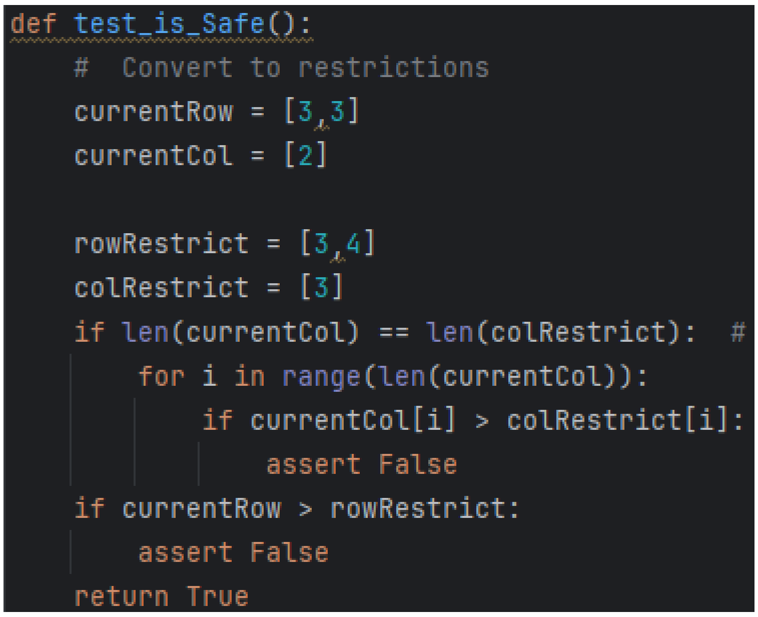
Tests the ‘rowToRestriction’ function that turns a given row into its form as a constraint to allow element-wise comparison with the constraint for the row. The ‘colToRestriction’ method functions the same with only a change in where the temporary row is pulled from so no tests have been written for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



```python
def test_rowToRestriction():
    currentRow = [0]
    C = 10
    tempRow = [1, 1, 1, 0, 1, 1, 1, 1, 0, 0]
    l = 0
    for i in range(C):
        if tempRow[i] != 0:
            currentRow[l] = currentRow[l] + 1
        else:
            currentRow.append(0)
            l += 1
    currentRow = [i for i in currentRow if i != 0]
    assert currentRow == [3,4]
```

Test 'is_Safe'

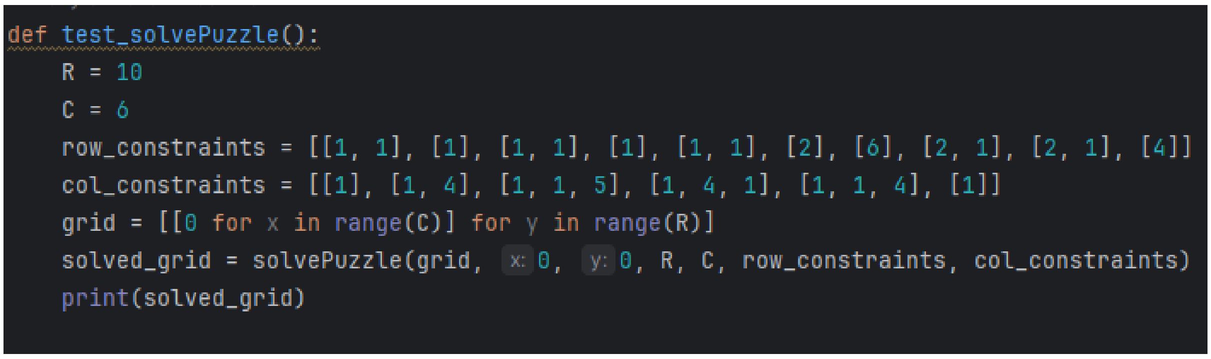
The next test tests the 'is_Safe' method which takes a row and column that has been turned into a restriction and compares it with the current row and column restrictions respective to its location. It simply performs comparisons between elements to determine whether the current configuration is safe.



```python
def test_is_Safe():
    #  Convert to restrictions
    currentRow = [3,3]
    currentCol = [2]

    rowRestrict = [3,4]
    colRestrict = [3]
    if len(currentCol) == len(colRestrict):  #
        for i in range(len(currentCol)):
            if currentCol[i] > colRestrict[i]:
                assert False
    if currentRow > rowRestrict:
        assert False
    return True
```

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.

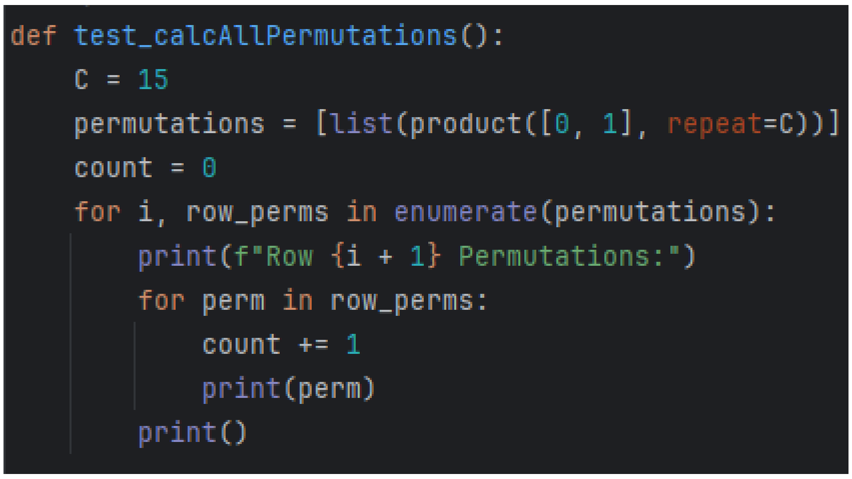


```python
def test_solvePuzzle():
    R = 10
    C = 6
    row_constraints = [[1, 1], [1], [1, 1], [1], [1, 1], [2], [6], [2, 1], [2, 1], [4]]
    col_constraints = [[1], [1, 4], [1, 1, 5], [1, 4, 1], [1, 1, 4], [1]]
    grid = [[0 for x in range(C)] for y in range(R)]
    solved_grid = solvePuzzle(grid, x: 0, y: 0, R, C, row_constraints, col_constraints)
    print(solved_grid)
```

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

test ‘calcAllPermutations’

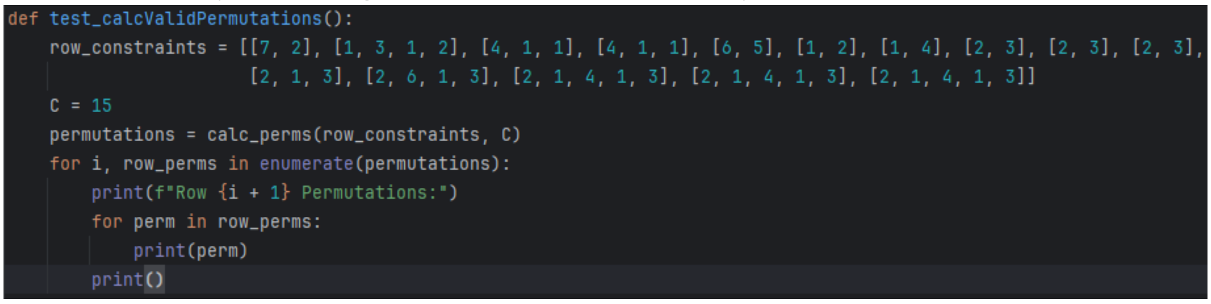
This test determines whether the correct number of permutations will be generated for a row of a certain size, the correct number is 2^N where N is the size of the row. Since there would be N spaces and each space could be 1 or 0.



```python
def test_calcAllPermutations():
    C = 15
    permutations = [list(product([0, 1], repeat=C))]
    count = 0
    for i, row_perms in enumerate(permutations):
        print(f"Row {i + 1} Permutations:")
        for perm in row_perms:
            count += 1
            print(perm)
        print()
```

test ‘calcValidPermutations’

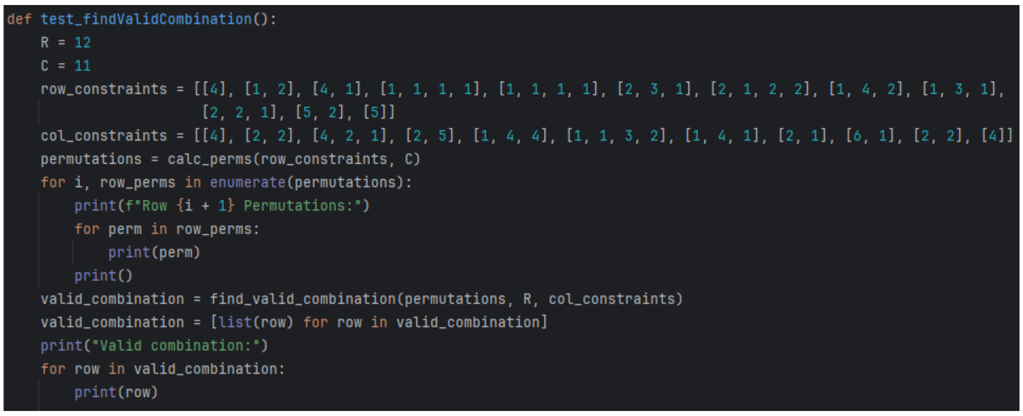
This test determines whether the permutations can be filtered correctly to only contain permutations that will be transformed into the correct constraint. It uses the permToRestriction which is another version of rowToRestriction. The test prints its contents out, as long as they are not empty then the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



```python
def test_calcValidPermutations():
    row_constraints = [[7, 2], [1, 3, 1, 2], [4, 1, 1], [4, 1, 1], [6, 5], [1, 2], [1, 4], [2, 3], [2, 3], [2, 3],
                       [2, 1, 3], [2, 6, 1, 3], [2, 1, 4, 1, 3], [2, 1, 4, 1, 3], [2, 1, 4, 1, 3]]
    C = 15
    permutations = calc_perms(row_constraints, C)
    for i, row_perms in enumerate(permutations):
        print(f"Row {i + 1} Permutations:")
        for perm in row_perms:
            print(perm)
        print()
```

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.



```python
def test_findValidCombination():
    R = 12
    C = 11
    row_constraints = [[4], [1, 2], [4, 1], [1, 1, 1, 1], [1, 1, 1, 1], [2, 3, 1], [2, 1, 2, 2], [1, 4, 2], [1, 3, 1],
                       [2, 2, 1], [5, 2], [5]]
    col_constraints = [[4], [2, 2], [4, 2, 1], [2, 5], [1, 4, 4], [1, 1, 3, 2], [1, 4, 1], [2, 1], [6, 1], [2, 2], [4]]
    permutations = calc_perms(row_constraints, C)
    for i, row_perms in enumerate(permutations):
        print(f"Row {i + 1} Permutations:")
        for perm in row_perms:
            print(perm)
        print()
    valid_combination = find_valid_combination(permutations, R, col_constraints)
    valid_combination = [list(row) for row in valid_combination]
    print("Valid combination:")
    for row in valid_combination:
        print(row)
```

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 project's 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 the 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.

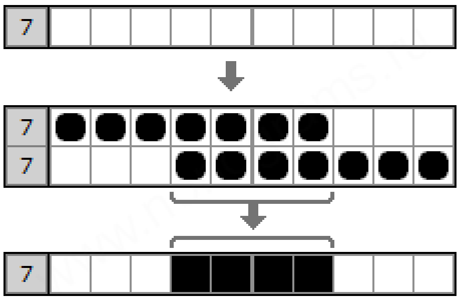
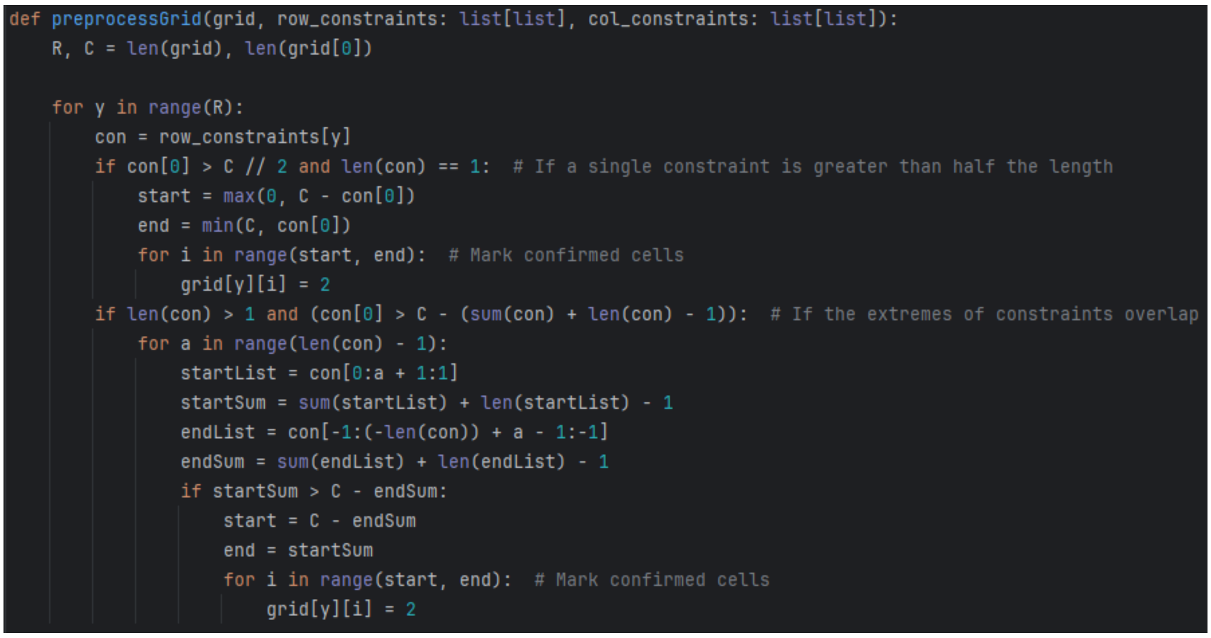


FIGURE 8 - A ROW FROM A NONOGRAM SHOWING LOGIC RULE 1

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.

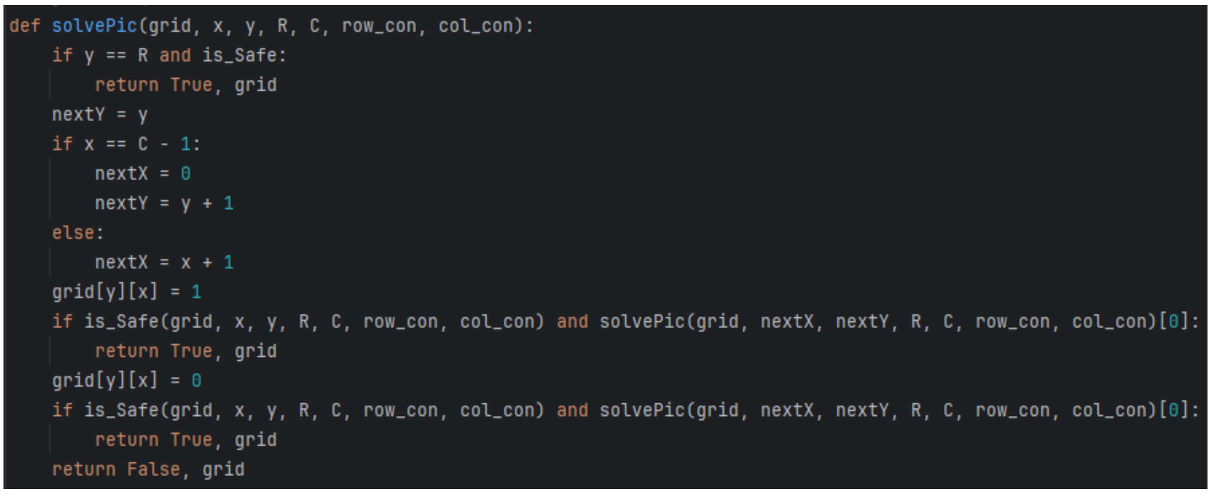


```python
def preprocessGrid(grid, row_constraints: list[list], col_constraints: list[list]):
    R, C = len(grid), len(grid[0])

    for y in range(R):
        con = row_constraints[y]
        if con[0] > C // 2 and len(con) == 1:  # If a single constraint is greater than half the length
            start = max(0, C - con[0])
            end = min(C, con[0])
            for i in range(start, end):  # Mark confirmed cells
                grid[y][i] = 2
        if len(con) > 1 and (con[0] > C - (sum(con) + len(con) - 1)):  # If the extremes of constraints overlap
            for a in range(len(con) - 1):
                startList = con[0:a + 1:1]
                startSum = sum(startList) + len(startList) - 1
                endList = con[-1:(-len(con)) + a - 1:-1]
                endSum = sum(endList) + len(endList) - 1
                if startSum > C - endSum:
                    start = C - endSum
                    end = startSum
                    for i in range(start, end):  # Mark confirmed cells
                        grid[y][i] = 2
```

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 were 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 for variable ordering, unlike the second solver.

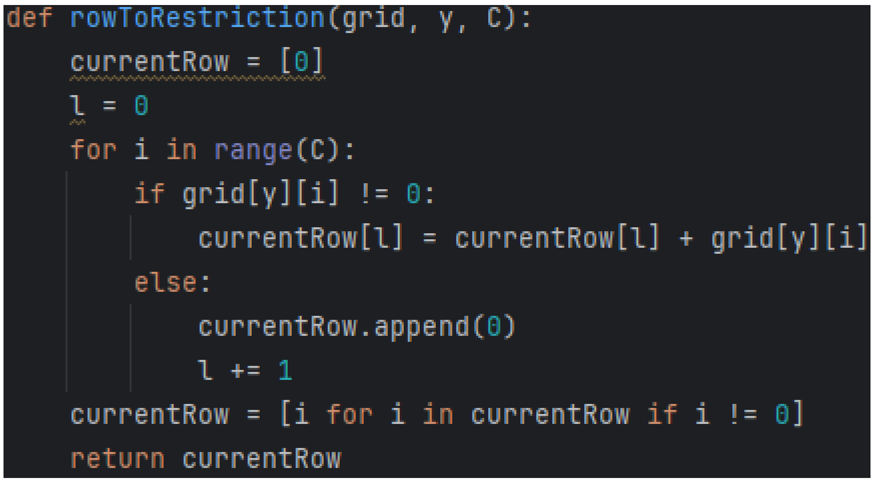


```python
def solvePic(grid, x, y, R, C, row_con, col_con):
    if y == R and is_Safe:
        return True, grid
    nextY = y
    if x == C - 1:
        nextX = 0
        nextY = y + 1
    else:
        nextX = x + 1
    grid[y][x] = 1
    if is_Safe(grid, x, y, R, C, row_con, col_con) and solvePic(grid, nextX, nextY, R, C, row_con, col_con)[0]:
        return True, grid
    grid[y][x] = 0
    if is_Safe(grid, x, y, R, C, row_con, col_con) and solvePic(grid, nextX, nextY, R, C, row_con, col_con)[0]:
        return True, grid
    return False, grid
```

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.

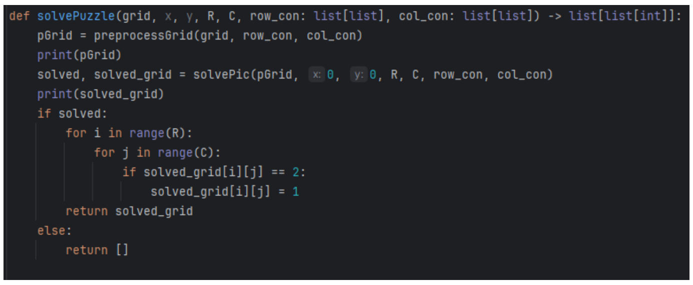


```python
def rowToRestriction(grid, y, C):
    currentRow = [0]
    l = 0
    for i in range(C):
        if grid[y][i] != 0:
            currentRow[l] = currentRow[l] + grid[y][i]
        else:
            currentRow.append(0)
            l += 1
    currentRow = [i for i in currentRow if i != 0]
    return currentRow
```

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.



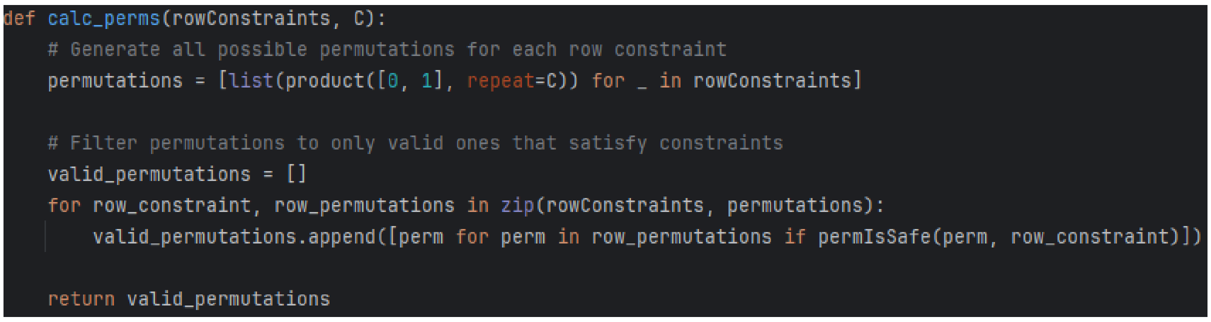
```python
def solvePuzzle(grid, x, y, R, C, row_con: list[list], col_con: list[list]) -> list[list[int]]:
    pGrid = preprocessGrid(grid, row_con, col_con)
    print(pGrid)
    solved, solved_grid = solvePic(pGrid, x: 0, y: 0, R, C, row_con, col_con)
    print(solved_grid)
    if solved:
        for i in range(R):
            for j in range(C):
                if solved_grid[i][j] == 2:
                    solved_grid[i][j] = 1
        return solved_grid
    else:
        return []
```

4.4 Final Solver

The final solver uses the method of generating all possible permutations of rows, and 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.

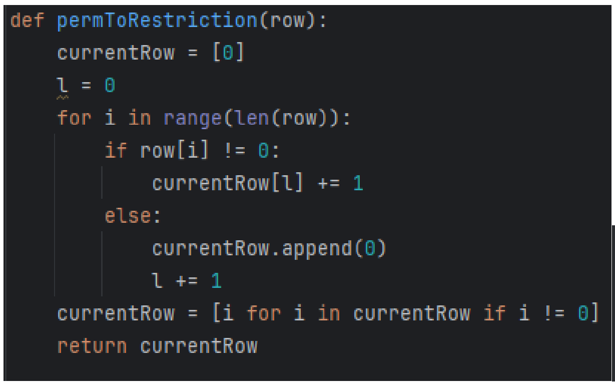
4.4.1 Calculate Permutations

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 the product, we can calculate all possible permutations for a row of a certain size.

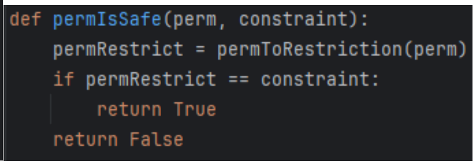


```python
def calc_perms(rowConstraints, C):
    # Generate all possible permutations for each row constraint
    permutations = [list(product([0, 1], repeat=C)) for _ in rowConstraints]

    # Filter permutations to only valid ones that satisfy constraints
    valid_permutations = []
    for row_constraint, row_permutations in zip(rowConstraints, permutations):
        valid_permutations.append([perm for perm in row_permutations if permIsSafe(perm, row_constraint)])

    return valid_permutations
```

Then we filter the permutations to only leave the valid permutations, ones that satisfy the row constraints for their respective rows, by using the 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.



```python
def permToRestriction(row):
    currentRow = [0]
    l = 0
    for i in range(len(row)):
        if row[i] != 0:
            currentRow[l] += 1
        else:
            currentRow.append(0)
            l += 1
    currentRow = [i for i in currentRow if i != 0]
    return currentRow
```



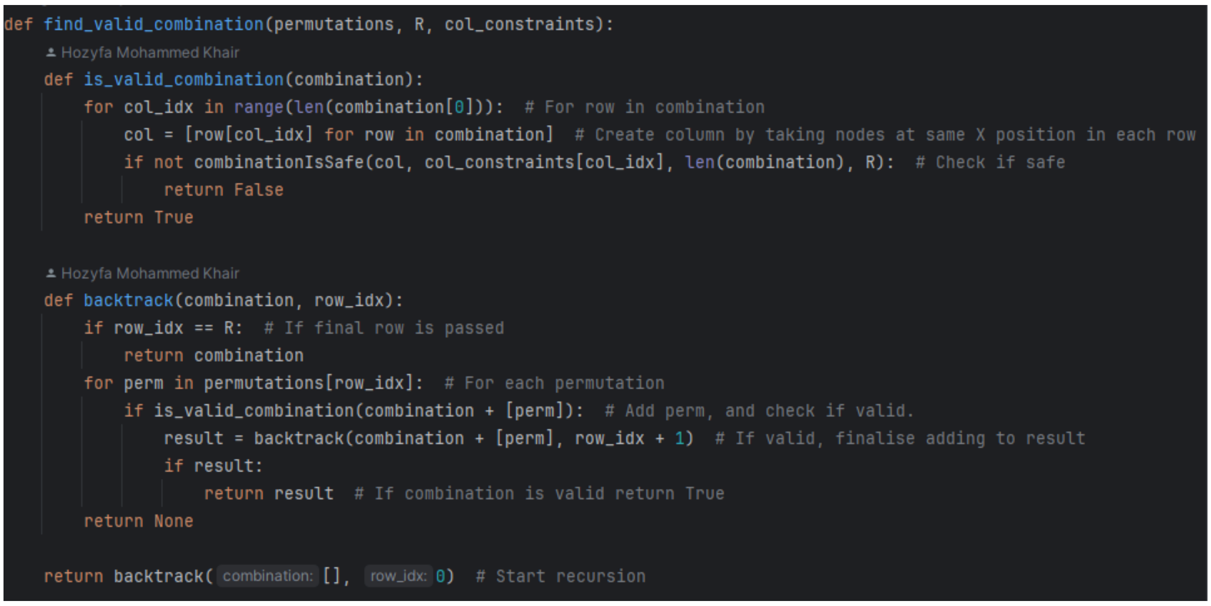
```python
def permIsSafe(perm, constraint):
    permRestrict = permToRestriction(perm)
    if permRestrict == constraint:
        return True
    return False
```

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, thereby 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.

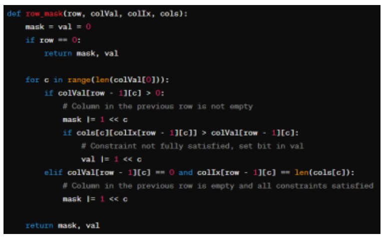
Backtrack is the main backtracking method for this solver, its base case of row_idx == R, is the case where the last row has been passed, therefore it returns the combination of rows. Otherwise, for each permutation in a row, it attempts to add that permutation to the current combination. If is_valid_combination 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.



```python
def find_valid_combination(permutations, R, col_constraints):
    Hozyfa Mohammed Khair
    def is_valid_combination(combination):
        for col_idx in range(len(combination[0])):  # For row in combination
            col = [row[col_idx] for row in combination]  # Create column by taking nodes at same X position in each row
            if not combinationIsSafe(col, col_constraints[col_idx], len(combination), R):  # Check if safe
                return False
        return True

    Hozyfa Mohammed Khair
    def backtrack(combination, row_idx):
        if row_idx == R:  # If final row is passed
            return combination
        for perm in permutations[row_idx]:  # For each permutation
            if is_valid_combination(combination + [perm]):  # Add perm, and check if valid.
                result = backtrack(combination + [perm], row_idx + 1)  # If valid, finalise adding to result
                if result:
                    return result  # If combination is valid return True
        return None

    return backtrack( combination: [],  row_idx: 0)  # Start recursion
```

4.4.3 Experimental rowMask

rowMask is an unfinished function which takes the number of 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 incomplete.



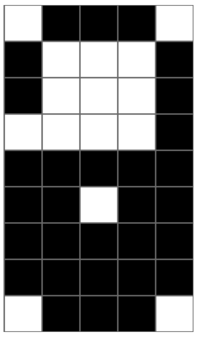
```python
def row_mask(row, colVal, colIx, cols):
    mask = val = 0
    if row == 0:
        return mask, val

    for c in range(len(colVal[0])):
        if colVal[row - 1][c] > 0:
            # Column in the previous row is not empty
            mask |= 1 << c
            if cols[c][colIx[row - 1][c]] > colVal[row - 1][c]:
                # Constraint not fully satisfied, set bit in val
                val |= 1 << c
        elif colVal[row - 1][c] == 0 and colIx[row - 1][c] == len(cols[c]):
            # Column in the previous row is empty and all constraints satisfied
            mask |= 1 << c

    return mask, val
```

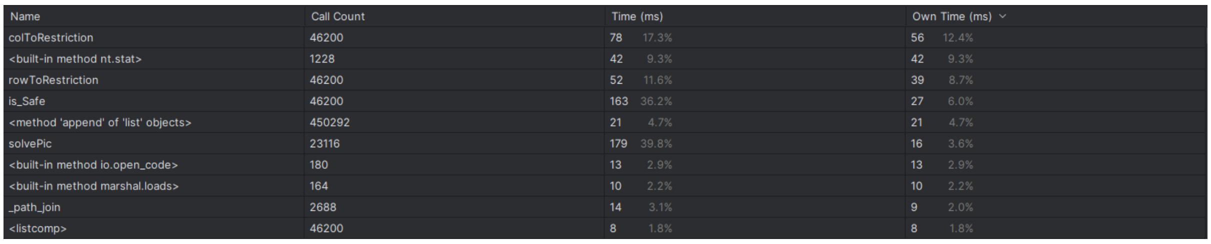
4.5 Solver comparisons and benchmark

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.

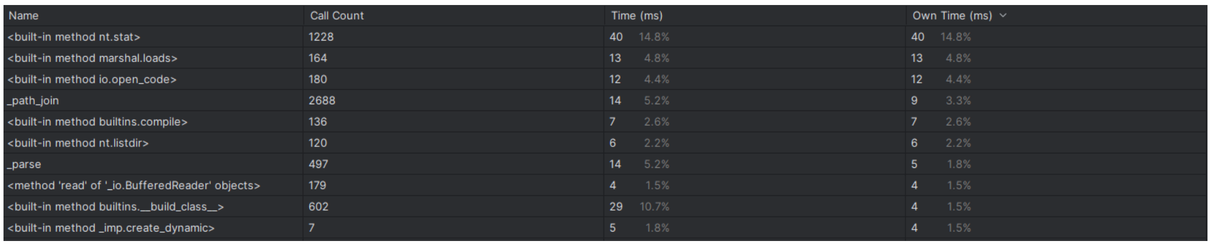
Considering a puzzle of size 9x5, titled lock, profiling the initial solver gives us a run-time of 180ms. A large amount of this is spent on calling colToRestriction, rowToRestriction, and is_Safe. These methods while essential for constraint checking, are currently too inefficient and the number of 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 baseline.

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.





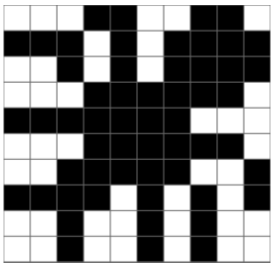
Name	Call Count	Time (ms)	Own Time (ms) ⌄
colToRestriction	46200	78 17.3%	56 12.4%
<built-in method nt.stat>	1228	42 9.3%	42 9.3%
rowToRestriction	46200	52 11.6%	39 8.7%
is_Safe	46200	163 36.2%	27 6.0%
<method 'append' of 'list' objects>	450292	21 4.7%	21 4.7%
solvePic	23116	179 39.8%	16 3.6%
<built-in method io.open_code>	180	13 2.9%	13 2.9%
<built-in method marshal.loads>	164	10 2.2%	10 2.2%
_path_join	2688	14 3.1%	9 2.0%
<listcomp>	46200	8 1.8%	8 1.8%

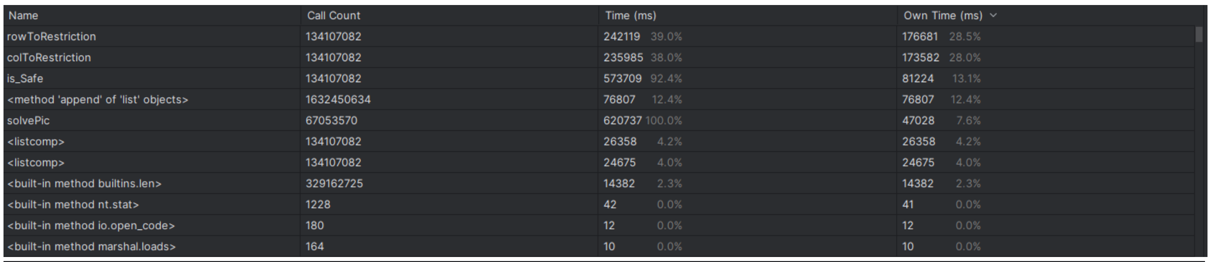


Name	Call Count	Time (ms)	Own Time (ms) ⌄
<built-in method nt.stat>	1228	40 14.8%	40 14.8%
<built-in method marshal.loads>	164	13 4.8%	13 4.8%
<built-in method io.open_code>	180	12 4.4%	12 4.4%
_path_join	2688	14 5.2%	9 3.3%
<built-in method builtins.compile>	136	7 2.6%	7 2.6%
<built-in method nt.listdir>	120	6 2.2%	6 2.2%
_parse	497	14 5.2%	5 1.8%
<method 'read' of '_io.BufferedReader' objects>	179	4 1.5%	4 1.5%
<built-in method builtins.__build_class__>	602	29 10.7%	4 1.5%
<built-in method _imp.create_dynamic>	7	5 1.8%	4 1.5%

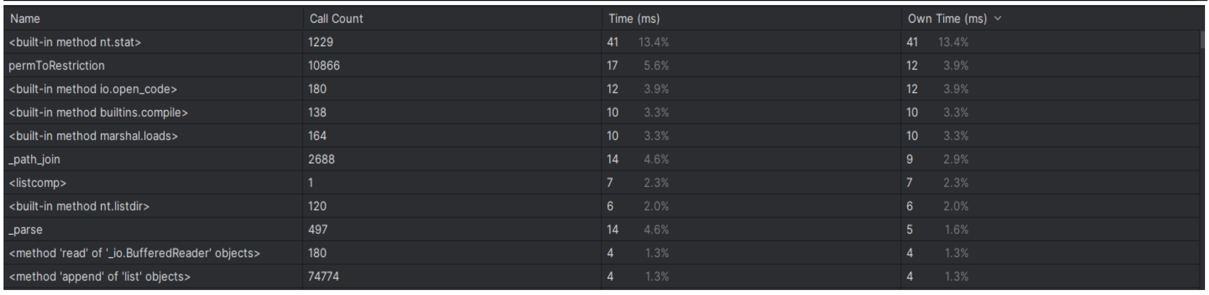
Considering a puzzle size of size 10x10, titled insect. The first solver takes an entire 10 minutes and 21 seconds to arrive at an answer. Showing how the first solver is much more inefficient than the second. When taking a look at the function call counts, rowToRestriction, and is_Safe, are each called 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 methods 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.





Name	Call Count	Time (ms)	Own Time (ms)
rowToRestriction	134107082	242119 39.0%	176681 28.5%
colToRestriction	134107082	235985 38.0%	173582 28.0%
is_Safe	134107082	573709 92.4%	81224 13.1%
<method 'append' of 'list' objects>	1632450634	76807 12.4%	76807 12.4%
solvePic	67053570	620737 100.0%	47028 7.6%
<listcomp>	134107082	26358 4.2%	26358 4.2%
<listcomp>	134107082	24675 4.0%	24675 4.0%
<built-in method builtins.len>	329162725	14382 2.3%	14382 2.3%
<built-in method nt.stat>	1228	42 0.0%	41 0.0%
<built-in method io.open_code>	180	12 0.0%	12 0.0%
<built-in method marshal.loads>	164	10 0.0%	10 0.0%



Name	Call Count	Time (ms)	Own Time (ms)
<built-in method nt.stat>	1229	41 13.4%	41 13.4%
permToRestriction	10866	17 5.6%	12 3.9%
<built-in method io.open_code>	180	12 3.9%	12 3.9%
<built-in method builtins.compile>	138	10 3.3%	10 3.3%
<built-in method marshal.loads>	164	10 3.3%	10 3.3%
_path_join	2688	14 4.6%	9 2.9%
<listcomp>	1	7 2.3%	7 2.3%
<built-in method nt.listdir>	120	6 2.0%	6 2.0%
_parse	497	14 4.6%	5 1.6%
<method 'read' of '_io.BufferedReader' objects>	180	4 1.3%	4 1.3%
<method 'append' of 'list' objects>	74774	4 1.3%	4 1.3%

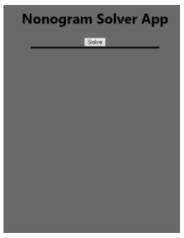
In summary, the comparative analysis shows the superior efficiency and scalability of the second solver over its predecessor. By minimizing redundant method invocations, the second solver demonstrates significant performance gains, paving the way for tackling more complex puzzles with enhanced speed 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.

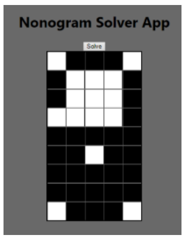
4.6 Front End

ReactJS was used for the front end, its capabilities in dynamically updating the HTML made for good ease of use in development and allowed for smooth chances when put in practice. This allows me to dynamically generate the grid and solution with ease, also allowing for future implementations of solving features.

4.6.1 Initial UI

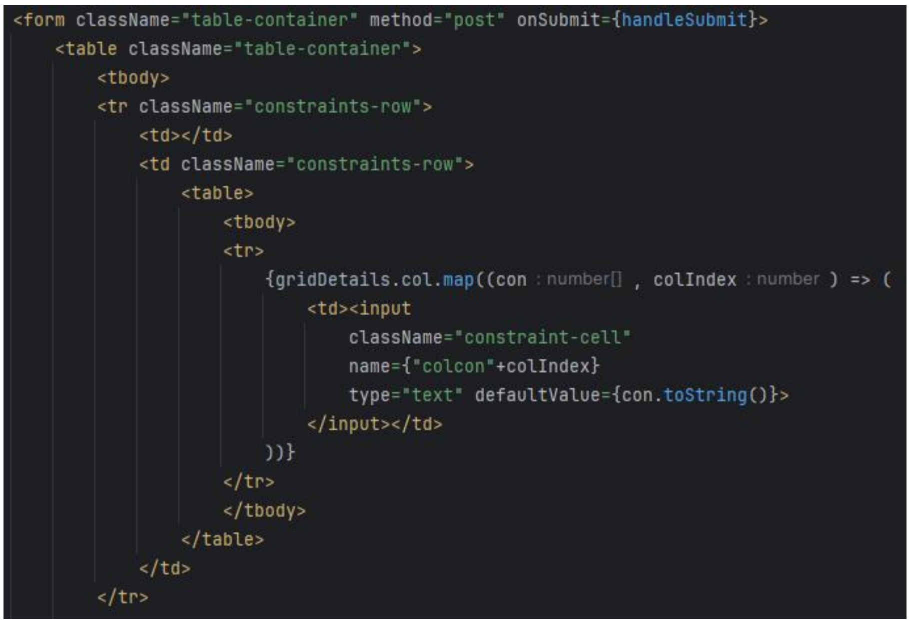
The first steps to develop the front end were rather simple, the initial requirement was only to have a grid that displayed empty, then solve a grid that was entered in the source code by pressing the solve button which would send a request to the back end and return with the array for the solved grid.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



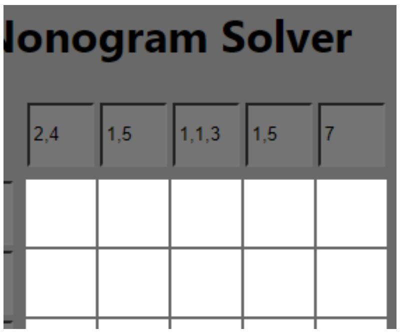


4.6.2 HTML

The HTML used to display the Nonogram grid has gone through major changes. Originally, I attempted to use the grid layout feature of CSS to display the Nonograms, but this proved slightly problematic. It was difficult to then add a constraint row and constraint column without disrupting the order of the nonogram cells, eventually, I switched the HTML to use the table element instead. This was easier since I felt I could find more examples of people using tables to display grids and thus draw inspiration. In hindsight, the real reason I struggled to add those constraint rows and columns was a lack of embedding elements, had I embedded grid dividers in each other the elements would align themselves and be easier to style individually. I ended up using this method for tables instead, which overall would have been an improvement anyway since individual cells of data in a table are easier to manage.



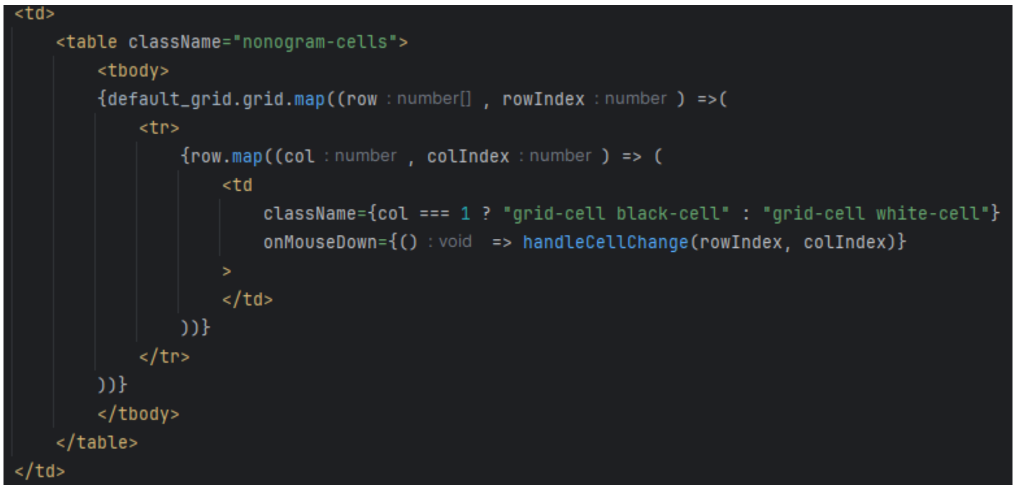
```
<form className="table-container" method="post" onSubmit={handleSubmit}>
    <table className="table-container">
        <tbody>
        <tr className="constraints-row">
            <td></td>
            <td className="constraints-row">
                <table>
                    <tbody>
                    <tr>
                        {gridDetails.col.map((con : number[] , colIndex : number ) => (
                            <td><input
                                className="constraint-cell"
                                name={"colcon"+colIndex}
                                type="text" defaultValue={con.toString()}>
                            </input></td>
                        ))}
                    </tr>
                    </tbody>
                </table>
            </td>
        </tr>
```



This image shows the outer wrapper of the form which is used to submit the contents of the constraints and grid size. Within the form is a table, with an empty data space for the top left of the grid, then another table which is used to hold the column constraints row. This layout was meant to eventually allow me to display every number in a constraint for a single column in its own cell, similar to the layout used at nonograms.org. This method would make the constraints more intuitive to read for the user, rather than a list separated by commas. Currently, each column holds its own input element of type text which holds the default value of whichever puzzle is loaded in. You can also change the constraints yourself by simply typing them in.

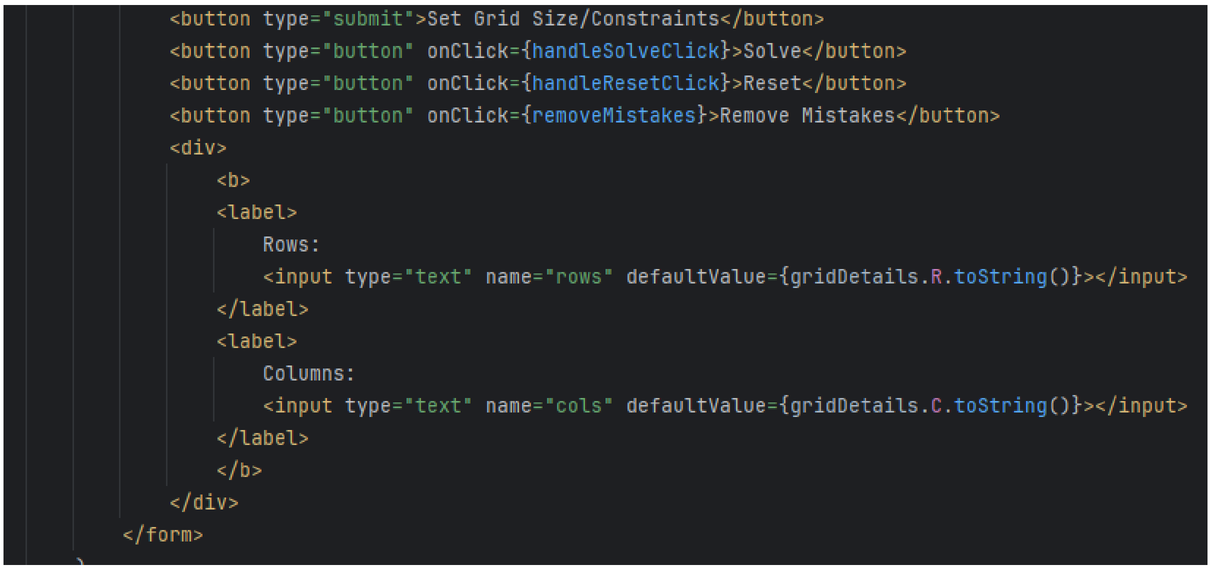
The layout for the row constraints is the same, except it is displayed on the left of the nonogram grid instead.

The nonogram grid itself is another table embedded in a table data element. This allows me to individually style the cells of the nonogram without affecting the constraints and ensures that the constraints stay aligned with the row and column that they are responsible for. The cells of the nonogram change their class name depending on whether there is a 1 or a 0 in the array representing their position, this change in class name causes the styling to switch from a white background to a black background and vice versa. The mouse-down event will allow the user themselves to toggle the cell on and off, using mouseDown instead of mouseClick will allow me to make it so the user can click down on one cell and then drag across the remaining grid cells they want to paint, however, this feature has not been added yet.

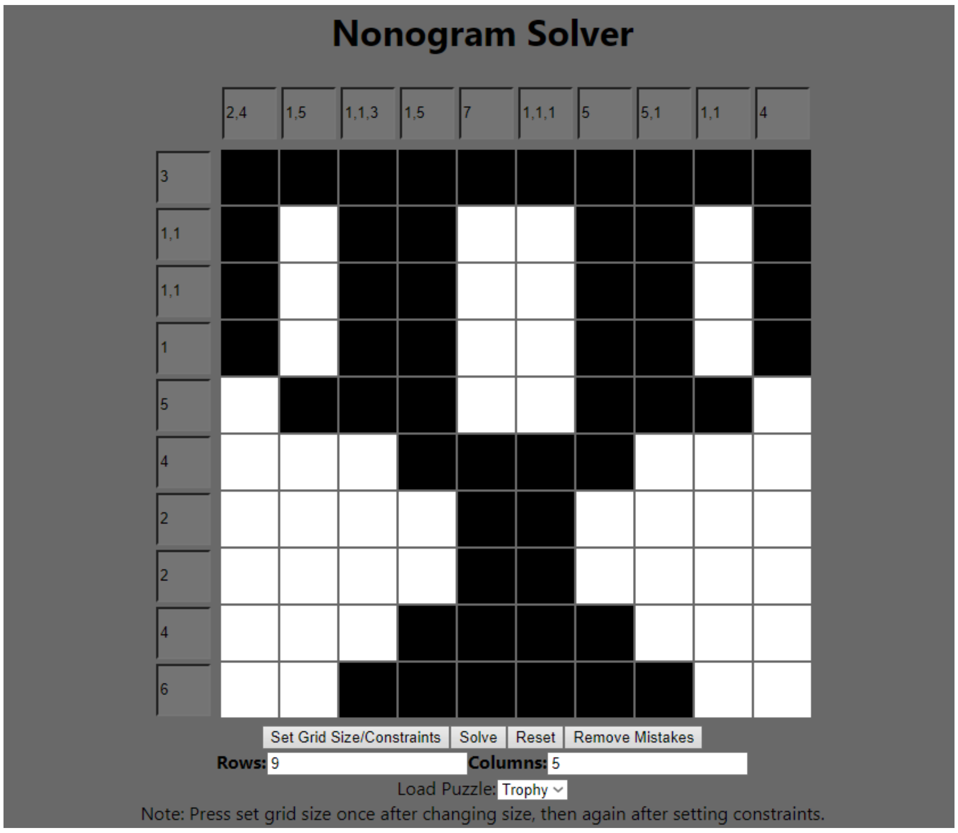


```
<td>
    <table className="nonogram-cells">
        <tbody>
        {default_grid.grid.map((row : number[] , rowIndex : number ) =>(
            <tr>
                {row.map((col : number , colIndex : number ) => (
                    <td
                        className={col === 1 ? "grid-cell black-cell" : "grid-cell white-cell"}
                        onMouseDown={() : void  => handleCellChange(rowIndex, colIndex)}
                    >
                    </td>
                ))}
            </tr>
        ))}
        </tbody>
    </table>
</td>
```

The remaining part of my HTML is the buttons used to confirm changes to the grid, reset the grid, solve the nonogram, and remove any mistakes. Solve and Reset buttons are basic requirements. The remove mistakes button is a rudimentary form of hints, the user can compare their partial solution of the puzzle to the final solution by comparing the arrays. Eventually, the plan was to have a notification when there are mistakes made and an option to show where the mistakes are. Currently, the user can use the remove mistakes button as a substitute for the hint button, if they select a random square and press remove mistakes they can confirm whether there will be a painted cell there or not.



```
                <button type="submit">Set Grid Size/Constraints</button>
                <button type="button" onClick={handleSolveClick}>Solve</button>
                <button type="button" onClick={handleResetClick}>Reset</button>
                <button type="button" onClick={removeMistakes}>Remove Mistakes</button>
                <div>
                    <b>
                    <label>
                        Rows:
                        <input type="text" name="rows" defaultValue={gridDetails.R.toString()}></input>
                    </label>
                    <label>
                        Columns:
                        <input type="text" name="cols" defaultValue={gridDetails.C.toString()}></input>
                    </label>
                    </b>
                </div>
            </form>
```

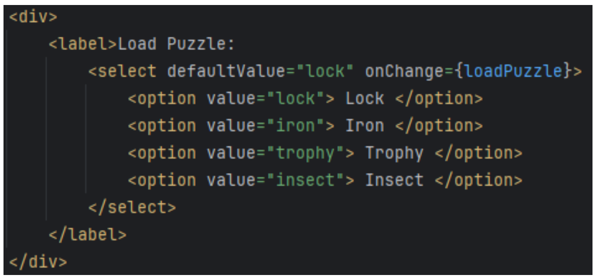


Rows: 9 Columns: 5

Load Puzzle: Trophy

Note: Press set grid size once after changing size, then again after setting constraints.

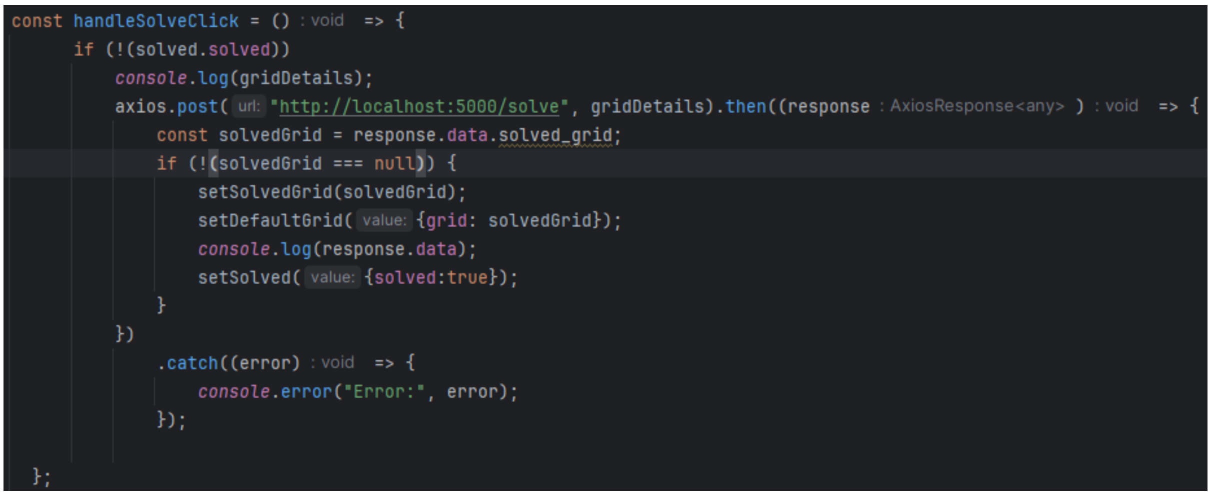
The final appearance is a simple page, showing a nonogram grid with the constraints aligned above and to the left. Crucial buttons are found directly under the grid in clear view, and the fields to change the size of the grid itself are labelled in bold. There is also a drop-down menu to allow users to cycle through four pre-loaded puzzles.


```
<div>
    <label>Load Puzzle:
        <select defaultValue="lock" onChange={loadPuzzle}>
            <option value="lock"> Lock </option>
            <option value="iron"> Iron </option>
            <option value="trophy"> Trophy </option>
            <option value="insect"> Insect </option>
        </select>
    </label>
</div>
```

4.6.3 JavaScript

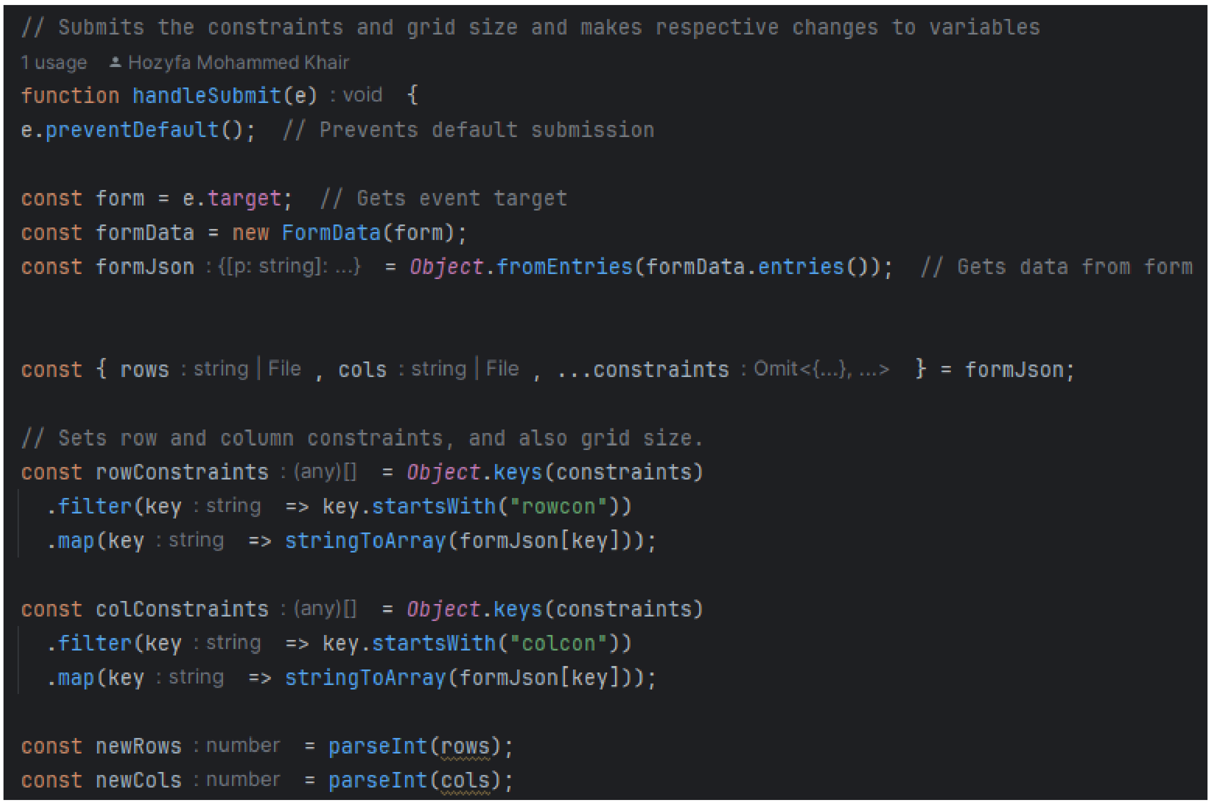
JavaScript is used to add functionality and process the logic of my code.

The first method is the handleSolveClick method which sends an Axios request after it has been called with the details of the grid needed to be solved. If the response is not null or an empty array the changes are then made to the front end, setting the required variables to their desired values.



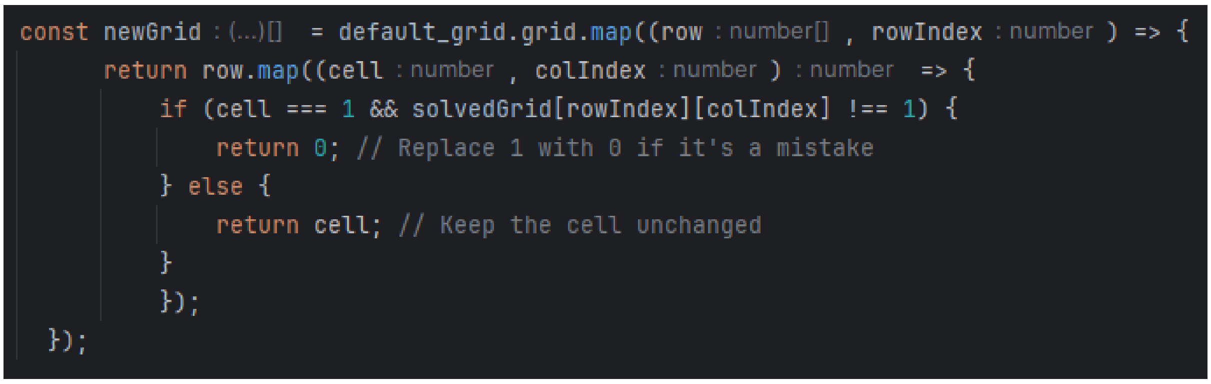
```
const handleSolveClick = () : void  => {
      if (!(solved.solved))
          console.log(gridDetails);
          axios.post( url: "http://localhost:5000/solve", gridDetails).then((response : AxiosResponse<any> ) : void  => {
              const solvedGrid = response.data.solved_grid;
              if (!(solvedGrid === null)) {
                  setSolvedGrid(solvedGrid);
                  setDefaultGrid( value: {grid: solvedGrid});
                  console.log(response.data);
                  setSolved( value: {solved:true});
              }
          })
              .catch((error) : void  => {
                  console.error("Error:", error);
              });

  };
```

handleSubmit is the next function, this handles the changes to the grid size and constraint by submitting the form that is wrapped around the entire nonogram grid. This will allow me to easily access the data and reflect the changes in the variables.



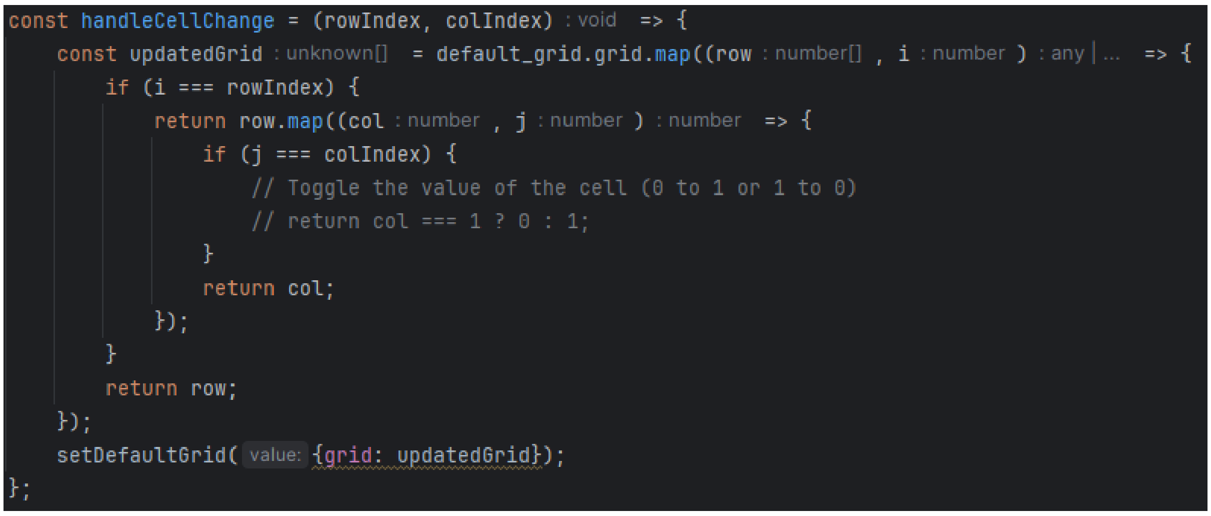
```
// Submits the constraints and grid size and makes respective changes to variables
1 usage   Hozyfa Mohammed Khair
function handleSubmit(e) : void  {
e.preventDefault();  // Prevents default submission

const form = e.target;  // Gets event target
const formData = new FormData(form);
const formJson : {[p: string]: ...}  = Object.fromEntries(formData.entries());  // Gets data from form


const { rows : string | File , cols : string | File , ...constraints : Omit<{...}, ...>  } = formJson;

// Sets row and column constraints, and also grid size.
const rowConstraints : (any)[]  = Object.keys(constraints)
  .filter(key : string  => key.startsWith("rowcon"))
  .map(key : string  => stringToArray(formJson[key]));

const colConstraints : (any)[]  = Object.keys(constraints)
  .filter(key : string  => key.startsWith("colcon"))
  .map(key : string  => stringToArray(formJson[key]));

const newRows : number  = parseInt(rows);
const newCols : number  = parseInt(cols);
```

The removeMistakes function is quite simple, it compares the current grid that the user has been interacting with and making changes to, with the solved grid of their puzzle and removes any mistakes. If the puzzle they are solving has not yet been solved it will also make a call to the solve method to retrieve the solution.



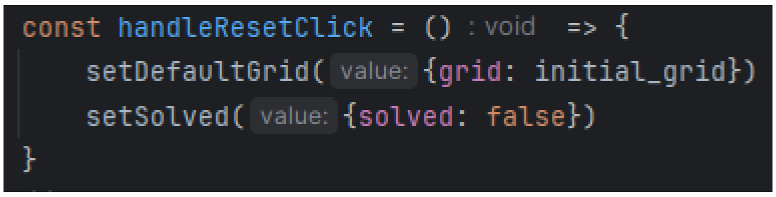
```
const newGrid : (...)[]  = default_grid.grid.map((row : number[] , rowIndex : number ) => {
      return row.map((cell : number , colIndex : number ) : number  => {
          if (cell === 1 && solvedGrid[rowIndex][colIndex] !== 1) {
              return 0; // Replace 1 with 0 if it's a mistake
          } else {
              return cell; // Keep the cell unchanged
          }
          });
  });
```

handleCellChange is another simple method, it refers to the parameters of row index and column index to determine the location of where a cell was clicked on and then toggles between 1 and 0 using an if statement.



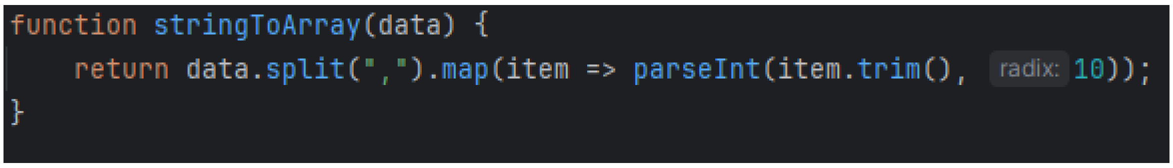
```
const handleCellChange = (rowIndex, colIndex) : void  => {
    const updatedGrid : unknown[]  = default_grid.grid.map((row : number[] , i : number ) : any | ...  => {
        if (i === rowIndex) {
            return row.map((col : number , j : number ) : number  => {
                if (j === colIndex) {
                    // Toggle the value of the cell (0 to 1 or 1 to 0)
                    // return col === 1 ? 0 : 1;
                }
                return col;
            });
        }
        return row;
    });
    setDefaultGrid( value: {grid: updatedGrid});
};
```

handleResetClick simply applies the initial empty grid to the current grid and sets solved to false



```
const handleResetClick = () : void  => {
    setDefaultGrid( value: {grid: initial_grid})
    setSolved( value: {solved: false})
}
```

stringToArray is an auxiliary function used to convert the inputs of the constraints into arrays so they can be passed to the back end.



```
function stringToArray(data) {
    return data.split(",").map(item => parseInt(item.trim(),  radix: 10));
}
```

These methods handle the functionality of the front-end well, and are modularised to help isolate and find bugs, using React's useState function allows seamless switching between different states for the user.

Expanding these basic elements, any future iterations of this project would focus on adding more quality-of-life features. Such as highlighting the corresponding constraints when hovering over a grid cell. Refining the UI aesthetics, and adding a back button to undo the last action.

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.

4.6.4 Designing a User Interface for The Solver

The user interface aimed 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. These goals have all been met, except the feature of providing hints could do with some improvements, as currently, the initiative is all on the user themselves to decide whether they need to check for a mistake.

Inputting a user's puzzle will begin by inputting the limits of the grid, this is displayed at the bottom of the grid with all the other buttons and options. The original plan was to have it above the grid, but when isolating only the grid size inputs it becomes counter-intuitive for a user who wants to change the settings. The constraints are inputted above and to the left of the rows and columns they correspond to, which is standard for nonograms and aligns with common sense.

The user is able to toggle between painted squares and empty squares by simply clicking on the desired cell, there was also the intent to allow the user to place Xs in a cell where they believe it is impossible for a painted cell to be, but this has not yet been implemented.

The user can press the 'Set Grid Size / Constraints' button to confirm the changes they have made to the grid details and begin the solving process. There is the main solve button which will produce the solution to the nonogram if it exists. There is also the Reset button which empties the grid of all painted squares, and the remove mistakes which removes any painted cells that are incorrect. The placement of these buttons is bottom and centre in a clearly visible place, each is labelled aptly and since they are gathered together it is easy for the user to determine where to look when they want to change any of the settings.

Additionally, 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. 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. Icons are an improvement I still need to make for the UI, using icons can help improve understanding of the function of a button at first glance and allows for people who may not read English to be able to use the UI.

Another improvement is letting the system status be visible. Currently, there are puzzles for which the solver will take a while and most users would just assume that it is not working since there is a lack of a loading bar or circle to indicate the puzzle is being processed.

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 will not feel lost or stuck while using the solver.

4.7 Project Conclusion

In conclusion, the development of the nonogram solver has used various software engineering techniques, a robust coding style, and thoughtful UI considerations. Through the implementation of Agile Methodology, the project maintained a flexible approach, allowing for iterative improvements and responsiveness to new information or changes in requirements. Using UML diagrams to determine the architecture of the code aided in efficient design and development.

The code itself was tested rigorously to ensure that additional features and aspects of the code would not be difficult to add. Using unit testing with PyTest along with Test-driven development principles ensured the code was robust, and profiling the code's run time performance and method calls allowed me to identify areas that needed to be optimized.

The front end was focused on producing a user interface with good usability and intuitive user interaction, design choices aimed at simplicity and functionality allowing a user to input, solve, and interact with nonogram puzzles seamlessly.

In summary, the project development and code implementation phase successfully integrated software engineering techniques. The result is a decent nonogram solver, with a practical UI, which can be used to quickly solve most simple nonograms. There are many improvements necessary, these will be detailed in the next section of the report, as well as an overview of the entire process.

5. Critical Analysis and Discussion

Overall, the project can be considered a success, it managed to complete the basics set out in the requirements and project plan producing an algorithm capable of solving a nonogram, a web app that allows users to solve their own nonograms and check their answers, and a partial loading system that allows a user to load a hard-coded nonogram. However, when compared to the potential for innovation and extension, the project falls short, displaying a lack of creativity and differentiation.

When considering the algorithms used, the first solver which attempted to solve cell by cell was a reasonable choice as a starting point for the project. However, reflection reveals that I missed many opportunities for optimisation and efficiency improvements. I hesitated to just take improvements from other solvers and learn and then credit their creations, for fear of not having enough original code or getting flagged for plagiarism. This hindered the project's progress, as I was often attempting to improve the efficiency from scratch on my own. The method of calculating row permutations is much more efficient and would also have been a good baseline for the project, with the same amount of time to improve this one larger puzzles would have been able to be solved.

The final deliverable was sufficient for the requirements, the program has a full object-oriented design and is implemented using modern software engineering principles (agile methodology). The program could have added a splash screen rather than jumping straight into the solver, but I felt for a web app a title screen or something similar would be unnecessary. The web app itself allows you to solve, load, and provide some assistance with puzzle solving. The program can solve simple nonograms very quickly and the target areas of improvement have already been identified.

Profiling the algorithms alone provided good insights into where their shortcomings lay, unfortunately, I only began profiling them for run time and method calls when I needed to compare and benchmark. Benchmarking did help identify the clear differences between the methods and why one was better than the other, these findings helped not only in the project’s future development but also in any projects I may undertake from here on.

A large shortfall of the project was the poor time management, since this falls into professional issues I will analyse it more there, a glance at the project diary will give one an understanding of the process. While the first term was good, the second term needed more effort or hours to improve the project.

In conclusion, while the project achieves its primary goals and adheres to modern software engineering principles, there are many opportunities for enhancement and refinement. A more proactive approach to research and development would greatly elevate the process and deliverables.

6. Professional Issues

In this section, I will discuss professional issues in general and those I encountered during my project, professionalism in computing is concerned with the societal impact of computer technology [Final Year Project Specification]. I will be focusing on Management and Plagiarism, as these have tie-ins to my own project.

Management is the process of handling resources such as time, money, and even effort by planning and directing to achieve some sort of goal. In an organization, this could be assigning senior employees to complete crucial tasks and giving new employees simpler tasks to learn and develop with. It could be funding a promising project that has produced satisfactory results, or defunding one that has taken too much time or effort in comparison to its potential return. Poor management can lead to many problems, the primary is not completing the goal or project within the specified time or not meeting all requirements set by the stakeholders. This could cause the effort placed in since to go to waste, or in the eyes of a business cause them losses in profit. Furthermore, if poor management results in employees having to work overtime or put in unreasonable amounts of effort for prolonged times, they could become dissatisfied and potentially leave their job requiring the leaders of that business to go through the hiring process again. This leaves the management, developers, and stakeholders all required to put in more effort in order to recoup losses, or simply cut their losses and move on.

Since this is an individual project, the only thing I have to manage is my own time and effort. Regarding the project, I started out quite well on time. Despite having a module that was 100% coursework and spanned the first term, and various other assignments, I completed the goals for the first term only being a few days behind schedule but it would not have much effect since I simply caught up during the Christmas holiday which was not required to be in the plan. The complications arose during the second term. For several reasons, be it other responsibilities or simply a bad work ethic, progress during the project was not made until a few weeks before the deadline. This led to an overall decline in the project’s quality, and the extensions selected in the plan were unable to be completed. There are a few things I could’ve done to prevent this, specifying a time in the project plan to review the progress of the project would’ve helped me understand the overall completion of the project, and more research when making the plan would’ve helped as I’d know which parts of the project were more intensive and would’ve been able to detail more specific goals for each week or sprint. Finally, setting aside hours specifically for working on the project every day rather than goals to achieve over a week would have prompted me to achieve more progress. Thankfully, I still managed to meet the basic requirements, but this did require a lot of effort in a brief period of time rather than small amounts of effort throughout the entirety of the second term.

Plagiarism is the act of taking another’s piece of work or intellectual effort and passing it off as your own, whether it be intentional or accidental. In computer science, this could be using another’s code, algorithms, or ideas without providing proper citation and acknowledgement. In a non-academic field, this is problematic to the rightful owner as there could be others using their work without acknowledging them as the creators, even competing against their product with stolen code. In an academic field, this is problematic to both the creator and the one passing their work off as their own. The creator would not get their rightful recognition as the developer of that algorithm or code. If the plagiariser is a student, they would only be worse off if it was intentional since they would be expected to be able to complete this in a professional situation in the future or even exams. If they are a researcher the institution could be illusioned that they are more capable than they are causing funding to be wasted.

On the other hand, in my project, I was hesitant to use other's code and ideas lest it be taken as plagiarism or lack of original code. While this did give me the experience of developing “new” algorithms partially on my own, it would have been best for the project as a whole if I had drawn inspiration from other solvers earlier and implemented and credited their ideas. Had I used other’s code as a basis for my own I would have been able to make innovative changes and overall improve the efficiency of my code since I was not solving problems that had already been solved or done by others. Of course, in the scope of the project it is not necessary to do things at the researcher level, but selecting interesting features from various solvers to make one capable of solving complex nonograms within a relatively short amount of time would have improved the overall quality of the final deliverables.

In conclusion, it’s important for a computer science professional to keep in mind professional issues that stem from the impact programs and technology have on society. Understanding the consequences, and why there are rules in place for computer professionals will help me navigate in future businesses, industries, or even research occupations.

Solving Nonograms with AI

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.

Term 2

Mar 23: Implemented multi-constraint preprocess and started unit tests

Mar 27: Fixed preprocessGrid function, implemented into the solver

Apr 1: Added push from walls rule for efficiency into preprocessGrid

Apr 3: Research ReactJS features to allow alignment of nonogram table

Apr 4: Changed front-end, added change constraints and reset button.

Solving Nonograms with AI

Apr 5:

- Allowed grid-size changes and fixed incorrect restraints bug
- Merged react_UI starting to finalise features
- Implemented row mask to reduce search space
- Added comments

Apr 6: Removed rowMask due to bug, will attempt to fix until final submission

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
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.

Appendix

User Manual and Installation

Requires python 3.9+, Node.js node package manager (npm in command line).

Flask and flask_cors need to be installed both can be done by running “npm install flask” and “npm install flask_cors”.

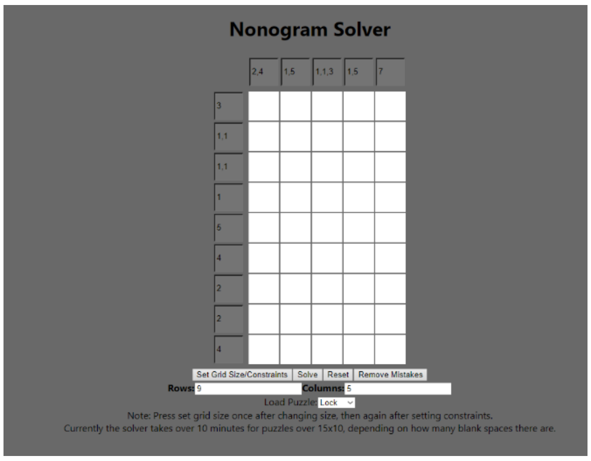
React can also be installed in the same way “npm install react”

Run the file nonogram_solver.py this can be done by any modern IDE

In command line go to the directory nonogram-solver-app and run npm start

The code should automatically open a web-browser page where the app is launched.

You’ll be met with the default page, you can attempt to solve this by just clicking on the grid, and pressing solve will reveal the correct answer. Pressing remove mistakes will remove any painted cells not in the solution.



Note: Press set grid size once after changing size, then again after setting constraints.
Currently the solver takes over 10 minutes for puzzles over 15x10, depending on how many blank spaces there are.

When trying to change the constraints, set the grid size in the rows and columns section first then press set grid size. Then add, change or remove constraints depending on the puzzle. The constraints must be separated by commas, whitespace does not matter. Then when finished setting constraints press “Set Grid Size/Constraints” again, and solve the nonogram.

Link to Video:
https://youtu.be/221si7waVYo