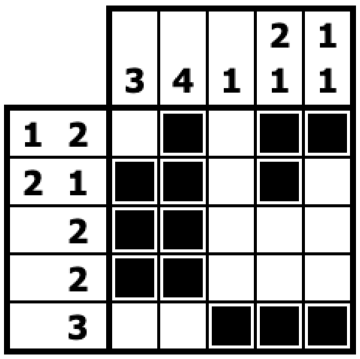
# CSE 202 Project: Nonogram Solver

Brian Tsao, Darren Yeung, Zhichao Liu, Jihu Mun, Shihui Wang

## 1 Project Summary

The game we will be attempting to solve computationally will be Nonogram. Nonogram is a picture logic puzzle game with squares that must be either filled or left blank according to the numbers at the side and top of the grid.

Example of a solved Nonogram puzzle:



The numbers next to each column and row are clues to the puzzle. For example, for the first row, there is "1 2". This means that from left to right, the first row must have 1 consecutive filled squares, one or more blank squares, and then 2 consecutive filled squares. The puzzle is solved when all of these clues are satisfied. We will refer to a single cell as a **square**, and a consecutive sequence of filled squares as a **block**.

## 2 Problem Specification

**Input:**

- The dimensions of the puzzle $(n, m)$, where $n$ is the number of rows and $m$ is the number of columns
- The clues for the puzzle $(R_1, \dots, R_n)$ and $(C_1, \dots, C_m)$
  - For each row clue $R_i = (r_{i1}, \dots, r_{ik})$, $\left(\sum_{j=1}^{k} r_{ij}\right) + k - 1 \leq m$.
  - For each column clue $C_i = (c_{i1}, \dots, c_{ik})$, $\left(\sum_{j=1}^{k} c_{ij}\right) + k - 1 \leq n$.

**Output:**

- A binary matrix ($n \times m$) that represents a solution for the puzzle. 1 represents a filled square and 0 represents a blank square. The matrix must satisfy the following:
  - For each row $i$ and the row clue $(r_{i1}, \ldots, r_{ik})$, from left to right, there must be $r_{i1}$ consecutive filled squares, one or more blank squares, $r_{i2}$ consecutive filled squares, and all the way until $r_{ik}$ consecutive filled squares.
  - For each column $i$ and the column clue $(c_{i1}, \ldots, c_{ik})$, from top to bottom, there must be $c_{i1}$ consecutive filled squares, one or more blank squares, $c_{i2}$ consecutive filled squares, and all the way until $c_{ik}$ consecutive filled squares.
- If there is no valid solution for the puzzle, the algorithm will output "unsolvable".

## 3 Algorithm

We use a backtracking algorithm for this problem. The main idea is that we try to fill each row according to the row clues by placing a horizontal block at a specific position, and then check whether the column clues are satisfied. If they are satisfied, we move on to the next row clue. If the column clues are violated, we try another position for the block. We return the board as a solution as soon as all rows are filled.

### 3.1 Checking if a column's clues are satisfied

An O represents a filled square and an X represents an empty square.

**Input:**

- List of clues in the column, such as [1, 2]
- List of lengths of blocks in the column, top to bottom (For example, OXOOX becomes [1, 2])

**Output:**

- True, if the solution (so far) for the column does not violate the clues
- False, otherwise

**Possible Violations:**

1. There are more blocks in the column than the number of clues
2. There exists a block (other than the last one) that has a different length than the clue corresponding to the block
3. The last block has a greater length than its corresponding clue
4. There are not enough squares left to satisfy all of the clues

If none of these violations are detected, we can say that the given column is valid and return true. Otherwise, we return false.

### 3.2 Listing the possible positions of the next block

**Input:**

- The current state of the board
- The length of the next block
- The list of remaining clues for the current row, as well as all column clues

**Output:**

- The list of valid starting columns for the next block

Given a partially filled board, we try to place the next horizontal block according to the next row clue. For the minimum starting column of the block, we need to leave one blank square after the end of the previous block. For the maximum starting column of the block, we have to consider the length of the block and length of the remaining clues. Then, we set the maximum starting column such that there are just enough squares to finish the remaining clues for the current row.

After getting the minimum and maximum starting columns, we now have a range of the columns $L, \ldots, R$ that can potentially be filled by the block. We will now check which of these columns would still be valid if one more filled square were appended to it. To accomplish this, we temporarily append one filled square to each column from $L$ to $R$ and check if the column is still valid.

For a block's starting column $i$ to be valid, the columns $i, i+1, \ldots, i+l-1$ must be valid. We return the list of such valid starting columns.

### 3.3 Backtracking

We start from the top-left square of the board, and we fill the board from top to bottom and from left to right by placing horizontal blocks, one at a time, according to the row clues.

#### 3.3.1 Subproblems

Let Solution$(B, C)$ be the solution of the board you can produce given the current state of the board $B$ and the list of remaining clues $C$. The function returns a valid solution if one exists, and it returns "No solution" otherwise.

#### 3.3.2 Base case

The base case is Solution$(B, \emptyset)$, which returns $B$ if $B$ satisfies all of the clues, or "No solution" otherwise.

### 3.3.3 Recursion

Given the current state of the board and the next horizontal block we have to place, there are several options of where we can place the block. We know which row we need to place the block, but there are multiple options for the starting column of the block, which is determined by the algorithm in section 3.2.

Given the board $B$ and the next clue $x$, we call the algorithm in section 3.2 to get the list of possible starting columns $c_1, c_2, \ldots c_n$ for $x$. Let $B_i$ be the modified board after placing the next block in the corresponding row and starting from column $c_i$. Then, we compute Solution$(B_i, C - x)$ for each $i$. If any of these return a valid solution, we return any one of those solutions. Otherwise, we return "No solution". If there are no valid starting columns for the next block, we also return "No solution".

The cases we consider are the starting columns $c_1, c_2, \ldots c_n$ for the next block. Assuming that the algorithm in section 3.2 is correct, these cases cover all possible cases. When we prune a case, it is because it violates the column clues according to the algorithm in section 3.1.

### 3.3.4 Form of Output

The final answer is Solution$(B, C)$, where $B$ is the empty board and $C$ is the list of all clues, sorted from the top row to the bottom row, and from left to right.

### 3.3.5 Pseudocode

```
Solution(B, C):
    if C is empty:
        return B

    x = next clue in C
    R = row of x
    L = length of x

    # List the valid starting columns for the next block
    starting_cols = list_starting_cols(B, x)

    for col in starting_cols:
        B’ = B
        Starting from col, place L filled squares in Row R of B’, left to right
        possible_sol = Solution(B’, C - x)

        if possible_sol is not Null:
            return possible_sol

    return Null
```

## 4 Proof of Correctness

#### 4.0.1 Proof for 3.1 (Checking if column is valid)

Given a column as lists of lengths of blocks, we try to determine whether the blocks satisfy the column clues. In the algorithm, we give 4 cases of where the cases are violated, and we prove that violating each of them will result in invalidity for the column.

1. If there are more blocks in the column than the number of clues, it is obvious that the column cannot meet the clues.
2. Since the order of our algorithm is from left-top to right-bottom, we cannot change the previous blocks. Thus, other than the last blocks, we have to make sure each block has the same length as each clues' block. Otherwise the column will violate the clues.
3. For the last (ongoing) block, it is possible to further extend it. However, it is impossible to shorten the block. Thus, if the last block is longer than the corresponding clue, it is impossible to satisfy the clues.
4. We have to make sure that there is enough unfilled space left in the column to satisfy the clues. Otherwise, it is impossible to finish the column.

If cases 1 and 2 are not violated, that means all the previous blocks (except the last block) satisfied the clues. Not violating cases 3 and 4 makes sure that it is possible to further extend the last block to meet the rest of the clues. Thus, the column will be valid if not violating the above 4 cases.

#### 4.0.2 Proof for 3.2 (Listing the possible positions of the next block)

We prove that:

1. Our selection will make the row valid according to the row clues, and keep the columns valid.
2. Our selection will cover every possible case.

**Statement 1**

We assume that the current state of the board is valid before we place the next block. Then, according to our strategy, we place the next horizontal block according to the next row clue. We calculate the minimum and maximum starting columns, to make sure that there is at least one blank space between the next block and the previous block and there are enough spaces left for the rest of row clues after placing the next block. By doing this, we make sure that the row is valid and that it is possible to finish the row in a valid manner.

Our selection also keeps the columns valid, since for every possible position of the next block we are checking whether the column clues will be violated if we place the block at that position.

**Statement 2**

We prove that the minimum and the maximum starting columns are actually the minimum and the maximum.

The minimum starting column is the column leaving one empty square after the previous block. If we move one more column to the left, it will occupy the empty square and connect with the previous block, violating the previous clue. Thus, the minimum starting column is the minimum we can go.

The maximum starting column is calculated by leaving just enough spaces for the remaining clues to the right of block. If we move that starting column one more space right, there will not be enough spaces for the remaining clues, violating the row clues. Thus, the maximum starting column is the maximum we can go.

Since the minimum and maximum starting columns are actually the minimum and the maximum, placing the next block within this range covers every possible cases while keeping the board valid.

## 5 Runtime Analysis

### 5.1 Checking if each column is valid

Let the board be an $n \times n$ matrix. There are at most $n$ clues in a single column and at most $n$ blocks. Since we only need to iterate through the clues and the blocks once, this algorithm is $O(n)$.

### 5.2 Listing the possible positions of the next block

It takes $O(n)$ time to figure out the minimum and the maximum starting columns, and checking if the columns are valid takes $O(n^2)$ time since there are at most $n$ columns and checking each column takes $O(n)$ time.

### 5.3 Backtracking

The number of branches we make at each step is at most $n$, since the list of possible starting columns for each block is at most $n$. Let $C$ be the number of clues. We have the following recurrence relation:

$$\begin{aligned}
T(C) &= nT(C-1) + O(n^2) \\
&= n^2T(C-2) + O(n^3) \\
&= n^C + O(n^{C+1}) \\
&= O(n^{C+1})
\end{aligned}$$

Since $C$ is at most $\frac{n^2}{2}$, this algorithm seems very inefficient. However, in the vast majority of cases there are much less than $n$ possible starting columns. Given $C$ clues, the worst case to maximize the number of branches is the case where all of the clues are length 1. This means that there are around $\frac{n^2}{C}$ choices for each block, which gives us an upper bound of $O\left(\left(\frac{n^2}{C}\right)^C\right)$. This is a better upper bound as $C$ increases. In practice, there will be a lot of times where there will only be 1 or 2 valid cases because of the column clue constraints, so the actual runtime is much lower.

## Appendix A: NP-Completeness Proof

Nonogram is in NP since we only need to check each row and each column linearly to verify the solution, and we could show that it is NP-Complete by reduction from Constraint Graph Satisfiability [1], only using two initially undirected gadgets: AND and OR. [2]

The global layout of the construction is shown in Figure 1.

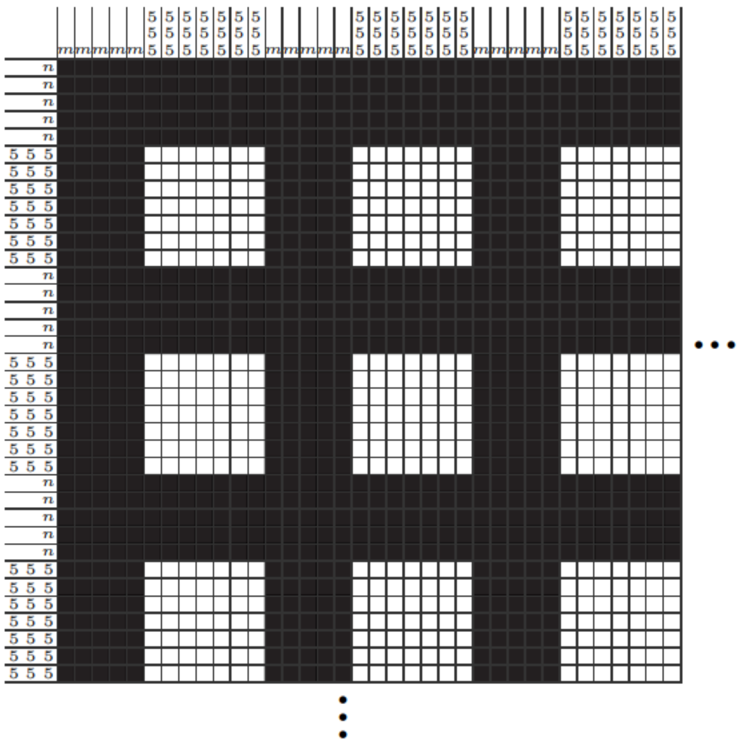


Figure 1: Global layout.

There are groups of $D$ adjacent columns and rows which are called separation lines. Between each group of separation lines, there are $G$ other lines. In this case (figure 1), $D = 5$ and $G = 7$. The descriptions and the width of the delimiters will not interfere with those of the single elements in between, so we can specify disjoint subnonograms between the separation lines.

We could send a signal between two orthogonal adjacent subnonograms by slightly adjusting delimiters between them as shown in Figure 2.

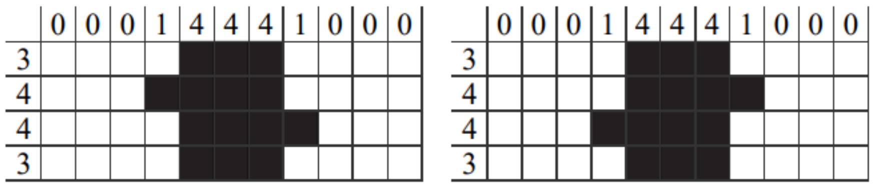


Figure 2: The two solutions of a Nonogram featuring two subnonograms horizontally separated by 3 separation lines.

We will use this property to construct gadgets within a subnonogram, and propagate signals between them, so we can embed a constraint graph on a Nonogram grid.

Figure 3 is the template of the gadgets. The black cells must be filled and the dotted cells must be left blank.

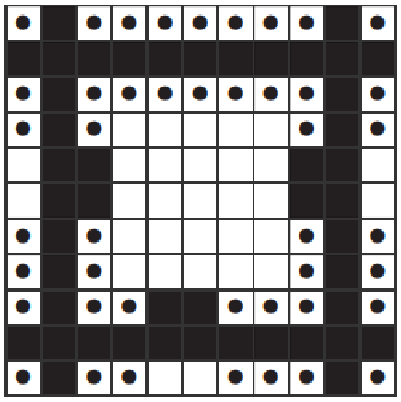
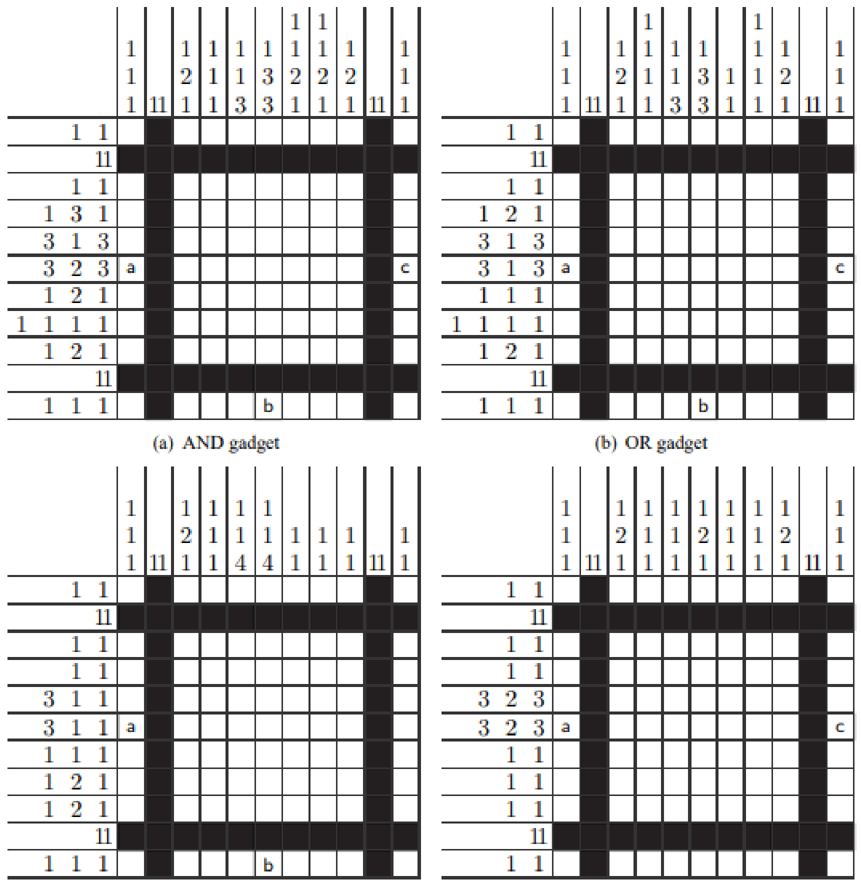


Figure 3: Template for Nonogram gadgets.

The gadgets shown in Figure 4 and have G = 7 and D = 1 since the width of separation line does not influence the functionality and the three cells marked with a, b and c correspond with the edges.

We can simulate a planar constraint graph with only AND and OR nodes on a Nonogram grid using the global layout of Figure 1 and the two top gadgets shown in Figure 4. The two bottom gadgets from Figure 4 can be used for wires, in a straight line or as a corner. Now the arrows can be inserted in a legal way if and only if the resulting Nonogram can be solved.

In this way we could reduce Constraint Graph Satisfiability to Nonogram and thus we prove that Nonogram is NP-Complete.



(a) AND gadget (b) OR gadget (c) Wire, turn (d) Wire, straight

Figure 4: Nonogram gadgets.

## Appendix B: References

[1] Hearn, R. A., & Demaine, E. D. (2009). *Games, puzzles, and computation.* CRC Press.

[2] Hoogeboom, H. J., Kosters, W. A., van Rijn, J. N., & Vis, J. K. (2014). Acyclic constraint logic and games. *ICGA Journal, 37*(1), 3-16.