# Solving a Nonogram Game Using Artificial Intelligence

Using Constraint Specified Problems to Determine Game Solutions.

Abrar Hussain c3hussaj
Matt Kim g5kimmat
Yu-Ching Chen g3yuch

December 6, 2015

## 1 Problem Specification

Nonogram, also known as Picross, is a logic puzzle game in which cells in a grid must be colored or left blank according to the numbers at the sides of the grid to reveal a hidden picture. It originates from Japan where Tetsuya Nishio invented it. The idea of using it for pictures came when a graphics editor created grid art out of skyscraper lights being on or off.[2]

In this puzzle, the numbers measure how many unbroken lines of filled-in squares there are in any given row or column. For example, a clue of ”3 2 4” would mean that there are sets of three, two, and four filled squares (in that order), with at least one blank square between each successive group. The images produced are often black-and-white on the grid structure and aren’t limited to square layouts. The following is a simple Nonogram with its solution:

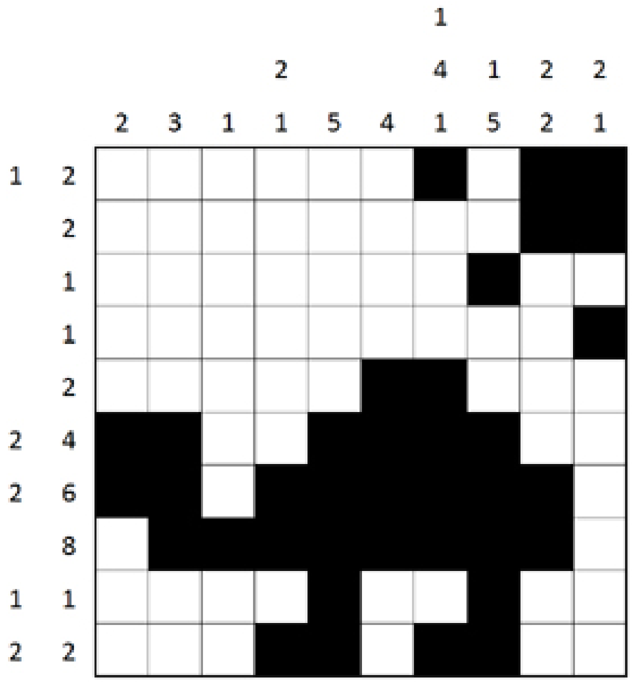
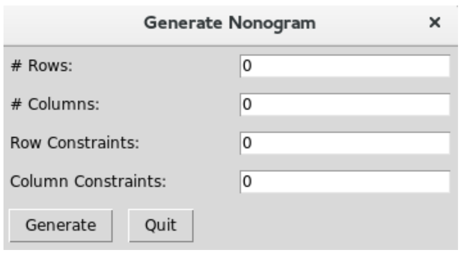


Figure 1: Example Nonogram

Here, we can see that the problem is specified as a 10x10 Nonogram (based on the number of rows and columns in the grid). Alongside the left side of the grid, we have the specification for each row. Alongside the top of the grid, we have the specification for each column. The specifications for all of the rows and columns are met to form an image of a camel with the sun shining on top. Although Nonograms are generally meant to be solved by humans (and produce relevant images), the complexity of a Nonogram can increase with specifications that do not produce aesthetic results.

## 2 Program Application

In order to solve the Nonogram problem, we have decided to create an application that would allow users to specify Nonograms to solve. The application takes the Nonogram board parameters as input and returns a visual display to the user of the Nonogram instance solution.



# Rows: 0
# Columns: 0
Row Constraints: 0
Column Constraints: 0

Figure 2: Nonogram Input

In order to run this, one needs to run `python3 input_nanogram.py`. Alternatviely, a Nonogram could also be specified as a text file with the format included further in the paper, with the command `python3 nonogram_csp.py <file>`. There is separate functionality provided for users that want to test many Nonograms at the same time.

## 3 Problem Approach

The problem of solving a Nonogram can be reduced to a similar Constraint Specification Problem. Here, we see that each row and column has some constraints due to it's specification. Therefore, we can specify constraints at the level of each row or column. At a higher level, for each row or column, we have that:

1. A number n means that n squares must be filled in a group.
2. For each number group, there must be one or more empty squares between the number group and its adjacent number group.

Although this restates the original problem, it does help us transition into thinking about this problem in terms of constraints and variables.

**Domain**

Each square within the grid would contain it's own variable. Based on the fact that a square can be either empty or colored, this gives way to a binary specification. So, the individual squares' domains are the set {0, 1}, where one specifies that the square is colored, and zero specifies that the square is empty.

**Choosing Constraint Specification Problems**

Choosing to reduce the Nonogram problem to a Constraint Specification Problem is an appropriate choice. We have already discussed that the specification for each row or column can be reduced to a constraint that we propagate over. The domain choice for each variable is obvious (zero or one). The alternative options for our solution approach would be to reduce our problem to search or something involving Baye's Nets.

A Bayesian network is not an appropriate choice due to the fact that the problem doesn't represent a set of random variables and their conditional dependencies well in a DAG. While there are relationships that exist between the different row and column specifications (such as the fact that the sum of row and column numbers equals the number of filled squares in the solution), this would not be enough to come with a DAG that can represent this problem well.

At the same time, reducing the problem to a search problem would also not be an appropriate choice. Although our game of Nonogram could be solved by performing DFS on a graph representing the different states of a Nonogram, this problem approach would be very slow [1].

**Constraint Representation**

We considered a few different approaches to representing our constraints. Initially, we considered creating a constraint for each number constraint in the row and columns. We realized that this created variables that satisfied multiple constraints at a time, giving a false output. Our final implementation was to create a constraint for every row and column. We construct the constraint recursively as a permutation of the different combinations possible given the numbers for the row or column.

**Constraint Construction**

Satisfying tuples were computed by this pseudocode:

**Algorithm 1** Generates Satisfying tuples

```
Require: space is the amount of space left to fill all of nums, nums is
    the list of numbers that constrain the row/column, used is backend
    information the function uses to recursively call itself
1: function $get\_sat\_tuples(space, nums, used = [])$
2:     $num \leftarrow$ first element in nums
3:     for pos as each position that num can fill do
4:         append pos to used
5:         if $len(nums) == 1$ then return used
6:         $space\_1 \leftarrow$ number of position num can't fill
7:         $get\_sat\_tuples(space\_1, nums[1:], used)$
```

## 4 Evaluation of Approach

Our Python application can be used for specifying and solvingn nxm Nonograms. The program starts by allowing users to specify the size and then specification alongside each row or column, and then takes some time to process the results. After processing the solution, we get a small graphical user interface with a nxm grid corresponding to the Nonogram solution. The following is an example:

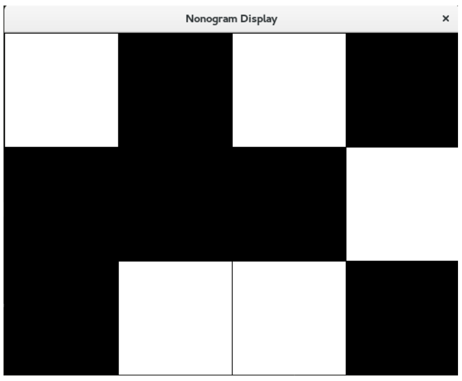
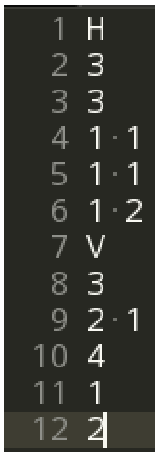


Figure 3: Example Solution

Here, we can see that the solution would have the matrix form [[0, 1, 0, 1], [1,1, 1, 0], [1, 0, 0, 1]], with each inner matrix corresponding to the rows of the grid. A one, according to our previously specified domain, indicates the square is colored, which we can see holds in the grid image.

**Format**

When defining a specific Nonogram, we represent it in the form of a text file with the the letter H at the top of the file, followed by the row specifications, and then the letter V and the column specifications. Here's an example Nonogram problem specification:



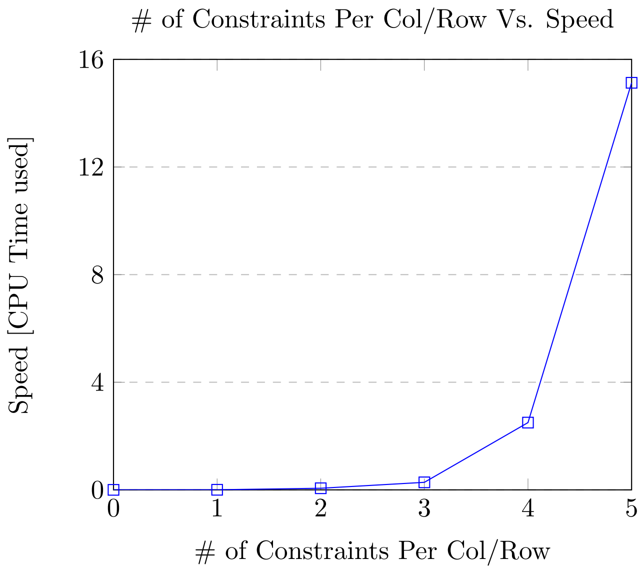
```
H
3
3
1·1
1·1
1·2
V
3
2·1
4
1
2
```

Figure 4: Example Text Input

The input from a text file is then parsed to generate a tuple of lists, with the first list containing the row specifications, and the second list containing the column specifications. From this intermediate representation we go on to create the Nonogram CSP Model, which returns to us the Constraint Model as well as the Variable array representing the problem. This process is similar to what was done for solving Sudoku as a constraint specification problem. Similarly to what was done in Sudoku, we create a solver in the form of a BT class for our CSP, and then perform `bt_search` using one of our available propagators (`prop_BT`, `prop_FC`, or `prop_GAC`). We print out the results to a Window with a grid, as shown previously in Figure 3. If you would like to see it in action, `'python3 ./run_nonograms.py'` will run our algorithm over a few examples.

We've tested our algorithm over a variety inputs and have observed sufficient results to provide as evidence of correctness. In terms of speed, our CSP approach scales with the size of the nanogram board. At about 25 x 25 our CSP begins to slow down from the initial speed of a few seconds.

Below is a graph of our algorithm's computing speed based on the average number of number constraints per row or column. An underlying factor here which may affect the speed is the board size increases as we increase the number of constraints per row as well. However, due to the nature in which we construct the list of satisfying tuples for each constraint it is unlikely that the size of the board significantly affects the runtime.

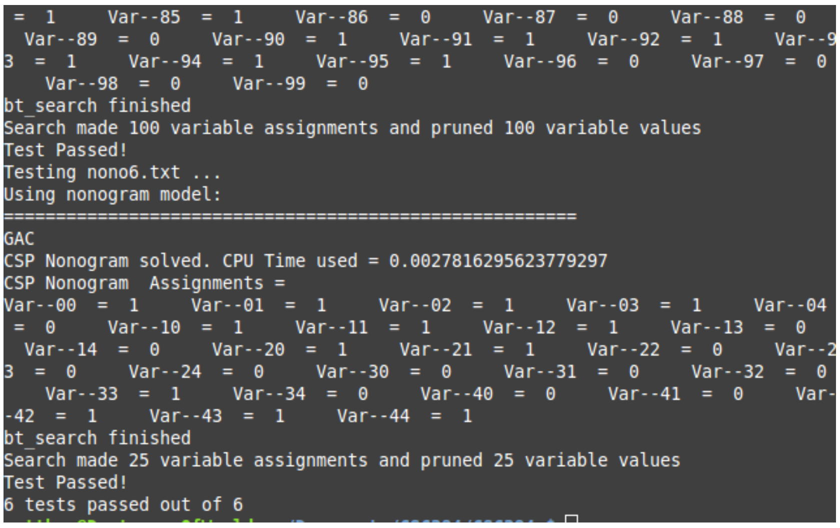


CSP may not be the greatest solution if we want it to scale well with the size of the problem. Fortunately, nonograms usually don't go beyond the size of 25 x 25. Similar to Sudoku, there is an average size of the problem. For example, the first result for nonograms on google has a size limit of 25 x 25 [3]

## Automated Testing Setup

In order to ensure that our solution approach holds for different Nonograms, we decided to create automated tests to run on various Nonogram arrangements (up to 9x9). Although this is not a proof of correctness, we believe that it would be sufficient for most Nonogram problems (if not all).

In order to test multiple instances of Nonogram at once, we send in a file called `full_test.txt`, which lists one Nonogram instance text file per line. Each instance of Nonogram that we use comes with an available online solution, so we can check for consistency between our solution and the available solution online. The specifics of testing can be found in the file `test_nonograms.py`. We provide a solution for each test board and run our CSP search over the board. Then, we check if the solution provided matches the result. At the end of the script we display the total number of passes and fails.



```
 =  1     Var--85  =  1     Var--86  =  0     Var--87  =  0     Var--88  =  0
  Var--89  =  0     Var--90  =  1     Var--91  =  1     Var--92  =  1     Var--9
3  =  1     Var--94  =  1     Var--95  =  1     Var--96  =  0     Var--97  =  0
    Var--98  =  0     Var--99  =  0
bt_search finished
Search made 100 variable assignments and pruned 100 variable values
Test Passed!
Testing nono6.txt ...
Using nonogram model:
=======================================================
GAC
CSP Nonogram solved. CPU Time used = 0.0027816295623779297
CSP Nonogram  Assignments =
Var--00  =  1     Var--01  =  1     Var--02  =  1     Var--03  =  1     Var--04
 =  0     Var--10  =  1     Var--11  =  1     Var--12  =  1     Var--13  =  0
  Var--14  =  0     Var--20  =  1     Var--21  =  1     Var--22  =  0     Var--2
3  =  0     Var--24  =  0     Var--30  =  0     Var--31  =  0     Var--32  =  0
    Var--33  =  1     Var--34  =  0     Var--40  =  0     Var--41  =  0     Var-
-42  =  1     Var--43  =  1     Var--44  =  1
bt_search finished
Search made 25 variable assignments and pruned 25 variable values
Test Passed!
6 tests passed out of 6
```

Figure 5: Test Output

## 5 Conclusion

We can conclude that our Nonogram game problem can be solved by representing the problem as a CSP. Although we haven't proven correctness for our CSP solution, we have strong evidence to believe that a CSP would result in a correct solution most of the time. The evidence to support this is from our test cases and speed results. Our final product is complete with a GUI to specify new problem space inputs.

## References

[1] Wiggers WA *A comparison of a genetic algorithm and a depth first search algorithm applied to Japanese nonograms.* Twente student conference on IT, Jun 2004

[2] Dalgety, James, History of Grid Puzzles http://puzzlemuseum.com/griddler/gridhist.htm Retrieved 2013-12-18

[3] "Nonograms." - Online Puzzle Game. Web. 4 Dec. 2015.