Tai-Long Riddle
Stephen Hutt
Introduction to Artificial Intelligence
19 November 2024

Nonogram Constraint Satisfaction

Introduction

This report tackles solving Nonograms using constraint satisfaction. A nonogram is defined as a logic puzzle or game wherein the player marks squares as "full" or "empty" to form a picture on an n by n grid [5]. The numbers on the left and top borders of the grid denote blocks of consecutive cells, which can also be referred to as "blocks." The numbers on the left side of the grid refer to the blocks in each row, and the numbers on the top of the grid refer to the blocks in each column. Any two "blocks" of cells must have at least one empty cell between them. A zero in either a row or a column means that there is a non-specified amount of filled/empty cells in that line. A human puzzle solver might employ any number of strategies to solve a Nonogram, but this project uses constraint satisfaction.

One must first understand the definition of Constraint Satisfaction Problems (CSP) before attempting to implement them. CSPs are problems that use a certain number of constraints to find a solution. They are often used in artificial intelligence and can be used to solve logic puzzles such as Sudoku. Constraint Satisfaction uses three important components: X (variables), D (domains), and C (constraints). Variables are what need to be found in a CSP, domains represent the range of possible values that a variable could be, and constraints dictate the interactions between variables.

CSPs can be separated into two different types: Binary CSPs and Non-Binary CSPs [4]. In a Binary CSP, each variable must only relate to a singular other variable. In other words, the constraint will only contain two variables. An example of a Binary CSP would be the "map coloring problem," where each region on a map must be colored, and no two touching regions may have the same color. This is a Binary CSP because the constraint specifically refers to two regions touching. On the other hand, a Non-Binary CSP is the direct opposite of Binary CSP, meaning that the constraints will have more than two variables. This could be reflected in a scheduling problem where different students need to be assigned to rooms that have a capacity of more than two, but certain combinations of students cannot be assigned together [4].

There are a multitude of specific techniques used within CSPs, two of which are backtracking and forward checking. Brailsford et al. in "Constraint Satisfaction Problems: Algorithms and Applications" explain both of these techniques thoroughly, stating the functions, faults, and upsides of using both [2]. The article defines backtracking as a function wherein a variable is set as a specific value from its domain and then checked against a set of constraints. If the variable passes, the program continues, and if it fails, the program "backtracks" to the failed variable and tries another value from the domain. The program then continues until a solution is found or ends when the values in each variable's domain have been checked. While this method works, it is often inefficient, as it will check irrelevant paths. On the other hand, forward checking algorithms lend much less redundancy, as they prune search trees by "looking forward" and removing values from the variable domains that conflict with the current attempted variable in the backtracking algorithm. If a variable at any point has no values in the domain, the forward checking algorithm signals that the current assignment is invalid. Conveniently, the two algorithms can work together to iterate through the same search tree, with the backtracking algorithm trying out variables and the forward checking algorithm testing how that variable interacts with other variables in the problem [2].

While Sudoku puzzles have often been solved using constraint satisfaction, I thought it would be more interesting to try to solve a Nonogram, which is less well-known. In order to implement the CSP, I determined how the three components (X, D, C) of a CSP relate to a Nonogram. The variables (X) and domain (D) were easiest to define, as X was simply the number of cells in the grid, and D was whether a cell was empty or full. The variables were represented in the program as a 2D list containing 36 initial values of 0. The domain was not specifically an object in the program, but instead, the variables would be assigned either -1 (empty) or 1 (full) when being checked in the backtracking algorithm. Comparatively, the constraints (C) were much harder to define. I used nested lists for the row constraints and column restraints (represented on the borders of the grid), but the rules of the Nonogram puzzle had to be implemented later as functions, which became quite tricky.

Methods

I went about implementing the functions in the way that I would normally solve a Nonogram. I am aware there is likely a specific pseudocode for this purpose, but I understand my code better when I work through the logic myself. To begin, I created two distinct initial forward propagation functions before backtracking even began. The first function filled "obvious" cells using two rules. The first rule was that if a row or column had a border constraint that was the same as the size n of the board, like a "6" in a 6 x 6 board, it would mean that the entire row or column is filled in. The second rule was that any row or column with a border constraint that was more than half the size n of the board would have a specified number of filled cells starting from the middle of that line. The number was dictated by the equation border constraint - (size / 2)) * 2. For example, a "4" in a 6 x 6 board would mean that the middle two cells in that line are filled, as (4 - 3) * 2 = 2. After that rule was deployed, the puzzle was then subject to the second rule, which stated that if the current filled cells were the same amount as the constraint, the rest of the cells in the row or column must be empty. At this point, much of a 6 x 6 grid would be determined. To complete the CSP, a simple backtracking algorithm was implemented.

It was at this point that a recursive backtracking algorithm was implemented. To begin, the algorithm would check if the puzzle was solved (and return the puzzle if so). The puzzle was considered solved if no zeros existed in the grid. Then, the algorithm would find the first empty cell (containing the value 0) and try filling in the cell with either 1 (full) or -1 (empty). After that, the program would check for consistency with the row and column constraints. The row and column constraints checked for several things. For one, they returned invalid if the number of filled cells in a row or column was greater than the sum of the constraints. They also returned invalid if a block of filled cells was at any point more than any number in the constraint list or if the number of blocks exceeded what was stated in the constraint list. As described in the description of a backtracking algorithm earlier, if the value was invalid this function would backtrack and try another value. If the assigned value was valid, this function would recursively call itself until the puzzle was solved. The program completes by printing the solved puzzle.

In order to test the efficiency of using constraint satisfaction on Nonograms, a brute force solution was also created. A brute force solution is defined as a solution that tries every single possible answer in an attempt to find the correct one [3]. Every solution can be found using brute force - meaning, if a problem cannot be solved with brute force, it cannot be solved at all. It is a good baseline for timing code, as it is often extremely inefficient. For a 6 x 6 Nonogram, a brute force algorithm would explore around 68.7 billion possible configurations. This is extremely computationally expensive and takes a long time to run, so an alternative where only valid rows are checked was implemented.

Results

Before referring to the actual results from timing the two algorithms, it is important that I mention the shortcomings of the final product of this project. There are several logical and technical errors within the forward checking, backtracking, and constraint functions. Additionally, the brute force algorithm only works on one of the four tested Nonograms, which is possibly due to the issues with the constraint functions.

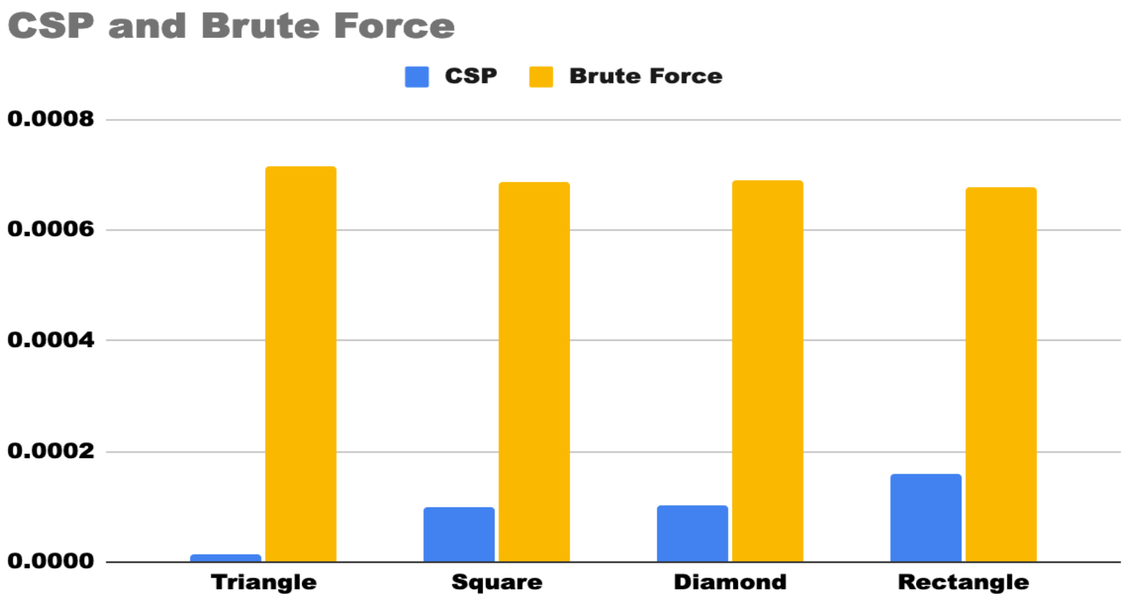
To begin, the solver can only handle the specific Nonograms that I have given it (diamond, triangle, square, rectangle). I believe this is due to several reasons, one of which being the logic in the forward checking functions. These functions cannot handle Nonograms that are not divided by two. This issue is further compounded by the fact that the solver cannot take in Nonograms with more than one constraint on a border (such as a 4 and a 1 attributed to a single column). This is likely a result of the specific logic implemented in the two functions that check if the rows and columns are valid. They do not account for the empty spaces between two blocks of filled cells and how that affects the rest of the logic in the function. I initially thought the number and size of the blocks were adequate, but I believe that these functions actually require an additional rule to work properly. To complicate matters further, these functions also allow variables that may not be correct by not requiring the number of filled cells to be exactly equal to the sum of the constraint blocks, but this is a direct result of how the backtracking algorithm functions.

The largest problem lies within how the backtracking works. Clearly, the backtracking is extremely similar to how it would solve a Sudoku puzzle. The largest difference is how the "variables" should be handled. In Sudoku, the individual cell is checked, and a number 1 through 9 is tested. Conversely, Nonograms only have the two values - filled (1) and not filled (-1). Since the constraints check rows and columns, not cells, the program was initially quitting early with no solution. It wasn't until it was too late that I thought of a better approach, which would be to create a generator of possible row or column values and then check those against the constraints. To remedy my code, I simply made it so that the validator functions let variables slip in that might not be entirely correct. Also, I would have liked to actually implement forward checking correctly within the backtracking algorithm, and I suspect that might require domains of row and column values instead of individual domains for each cell. Generally, this is why the program only works on four specific 6 x 6 Nonograms.

As a direct result of the constraints not being entirely correct, the brute force program isn't right either, and it only works on the diamond Nonogram. I have tried to edit the validator functions to only accept specific cells, but then it returns with no solution again. I actually do feed the brute force algorithm valid row combinations, so I'm not sure what is causing that issue in particular. I also don't know why only the diamond shape works. I have tried altering just one or two of the parameters in the constraints of the shape, but that breaks the code as well.

Overall, my attempts at debugging and restructuring the code were unsuccessful. The reason why the four included Nonograms do work is because they work on the first pass of the backtracking algorithm. I felt the need to include these errors in my results, as they do impact the validity of any conclusions I make.

The average time to solve Nonograms using the constraint satisfaction and brute force algorithms is represented in a graph below (over 1000 trials):



This graph fills in the missing data for the brute force algorithm in the categories triangle, square, and rectangle. Since these are all puzzles of the same size, it can be reasonably assumed that the brute force algorithm will take roughly the same amount of time across the categories. The results of the diamond category are repeated across all four to represent what the data is expected to look like.

Tree pruning, as exercised by the two forward propagation functions in the constraint satisfaction program, is very important to cutting down the time it takes the solvers to find the correct Nonogram [1]. The point of this pruning is to stop irrelevant paths of search before they happen, thus saving computational space and time.

Discussion

Overall, the goal of this project was to solve Nonograms using constraint satisfaction (specifically, backtracking and forward checking). While both techniques were partially implemented, there were multiple issues that hindered the effectiveness of the program. The primary issues were seen in the constraints and backtracking. Specifically, the constraints struggled to consider gaps between blocks of filled cells, and the backtracking algorithm needed to apply the constraints to entire rows or columns instead of individual cells.

After comparing constraint satisfaction in solving the four Nonogram puzzles to that of a brute force algorithm, it is clear that the latter is much slower. Since the brute force algorithm tested every valid possible configuration of the grid, it was computationally expensive. Contrastingly, the constraint satisfaction algorithm was much faster because it pruned irrelevant paths early on.

In the future, it would be interesting to see how much faster the program could be if proper forward checking was implemented along with the other improvements, including the testing of rows and columns.
Conclusively, while the constraint satisfaction algorithm was faster when comparing output times for the diamond Nonogram, the limitations imposed by the improper implementation of constraints and backtracking hurt the specificity of the results gathered.

References

[1] Fahiem Bacchus. 2000. Extending Forward Checking. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 35–51. Retrieved November 19, 2024 from http://dx.doi.org/10.1007/3-540-45349-0_5

[2] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research* 119, 3 (December 1999), 557–581. https://doi.org/10.1016/s0377-2217(98)00364-6

[3] freeCodeCamp. 2020. Brute Force Algorithms Explained. *freeCodeCamp.org*. Retrieved November 19, 2024 from https://www.freecodecamp.org/news/brute-force-algorithms-explained/

[4] GeeksforGeeks. 2023. Constraint Satisfaction Problems (CSP) in Artificial Intelligence. *GeeksforGeeks*. Retrieved November 19, 2024 from https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/

[5] PuzzlyGame. How to play Nonograms. *Puzzly Game*. Retrieved November 19, 2024 from https://puzzlygame.com/pages/how_to_play_nonograms/