Project Proposal

Team Members: Dorothy Nelson (dpn2111), Jittisa (Jane) Kraprayoon (jjk2239)

Nonogram Problem Description

A Nonogram is a logic puzzle where players fill in a grid with black or white squares to reveal a hidden image. Every group of black cells must be separated by at least one white cell. You are given clues which tell you the length of the groups of black cells in that row or column. For example, a clue like "3 2" means there will be a group three and then a group of two black cells in that row or column, separated by at least one blank space. When solving the logic puzzle manually, players are generally advised to start at rows or columns with large blocks of black cells.

The difficulties of these puzzles vary greatly depending on the size of the grid and the clue complexity. Puzzles designed for humans generally have one solution and reveal an image. However, it is also possible for Nonograms to have multiple solutions with no discernible pictures.

Nonograms also present interesting computational questions. Nonograms have been shown to be NP-hard and thus there are a multitude of possible approaches to solving one that balance correctness with computational complexity.

Algorithm Description

The most straightforward way to solve a nonogram is an iterative line solver algorithm that focuses on solving nonogram puzzles by analyzing individual rows or columns sequentially. It calculates all the possible block configurations for a line, storing them in an *n* x *k* matrix, where *n* is the line's length and k is the number of black cell blocks. As it processes new lines, the algorithm updates these matrices[1]. This approach is efficient

[1] https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4

for simpler puzzles but is slow for puzzles with large grid sizes or those with multiple solutions.

Instead of proceeding through the lines or rows sequentially, you can use a priority queue data structure to order lines based on their potential value. This adds a heuristic component to an iterative search. Lines with longer blocks are prioritized as having more potential information. When solving a line provides new data, related lines are reprioritized[2]. This strategy works well for randomly generated puzzles with multiple solutions, significantly better than an iterative search. However, this approach can stall if there are no more new pieces of information available.

To ensure that a priority data queue approach does not stall, you can use a breadth-first search (BFS). This algorithm identifies the least probable configurations for unknown blocks with the goal of creating contradictions quickly in order to prune the solution space. If a contradiction is found, the algorithm backtracks and updates the grid with the opposite assumption. This combination of iterative approach and probabilistic guessing enables the algorithm to solve even the most complex puzzles, it is more computationally complex.

Our Parallelization Strategy

For the preprocessing step, we iteratively pick cells that must be filled based on combinations of configurations. This step is independent to each row and column and can be broken down into 1) calculating all possible line configurations, 2) identifying definitive squares, and 3) updating the possibilities for the next iteration. Hence, we plan to use **Control.Parallel.Strategies** to parallelize the computation for these steps. For example, using **parMap** and **rseq** to independently find the configurations and definitive squares.

[2] https://medium.com/smith-hcv/solving-hard-instances-of-nonograms-35c68e4a26df

As nonograms can be represented as grids, we can also consider using **Repa** for array-based computations. For example, **ComputeP** can be used for parallel computation. However, combining Strategies and Repa might be counterproductive and we most likely plan to use only the Strategies approach.

For heuristics (priority queue), to select rows and columns to prioritize based on calculated heuristic scores, we can divide the grid into chunks and rank them. Then, we can keep merging these results for a global ranking result. Again Strategies can be utilized to parallelize this operation. The last BFS component can also be parallelized, although this might be trivial as it's definitely been implemented.

Input Data for the Algorithm

For the input, we plan on constructing a **.txt file** to be read and parsed by the program. A Nonogram can be defined by grid dimensions, row clues, and column clues and these can be stored in a .txt file as line-separated values. Instead of creating Nonograms from scratch, there are publicly available github repositories that supply sample Nonograms which we can use to create this input data[3].

[3] https://github.com/mikix/nonogram-db, https://github.com/monkeyArms/nonogram