American University of Armenia

College of Science and Engineering

Artificial Intelligence Project

Nonograms: Solving Japanese Puzzle using Machine language

Team:
Darbinyan L.
Hovhannisyan H.
Yeranosyan V.

Fall, 2022

Abstract

The Nonogram is a picture logic puzzle. The task is to generate an image on a grid by filling initially empty cells with unbroken lines of filled-in squares corresponding to the numbers predefined for the columns and rows. The grids are filled with black and white pixels while adhering to a description that specifies the lengths of consecutive black pixel segments for each row and column. After the successful completion, a hidden pixel-art picture is revealed. In this paper, we discussed several approaches to solve this problem, referring to topics such as tree search and local search. We will implement two approaches to solving this problem. As one approach, we defined our problem as Constraint Satisfaction Problem, and the other way we used is Genetic Algorithms to achieve a solution. Although Nonograms in puzzle books ordinarily are possible to solve by hand, however, in computational informatics, they are generally considered as NP-hard problems that usually require exponential time to find a possible solution.

Keywords:
Nonogram, NP-complete, Constraint Satisfactory Problem, Genetic Algorithms, Depth First Search, Brute Force, Fitness function, Tree Search, Local Search, Simulated annealing

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Setting and Description

Nonogram is a Japanese crossword that involves combining rows and columns in such a way that makes it possible to get a final image. It is a grid-based logic puzzle where the user is given an $n \times m$ grid with numbers marked on each row and column. The goal of the problem is to determine all the squares that should be filled in order to discover the ”hidden picture” as described by the row/column constraints. The general problem goes like this: Given the projection of the image (i.e. the numbers on the edges of the board), derive the image itself.

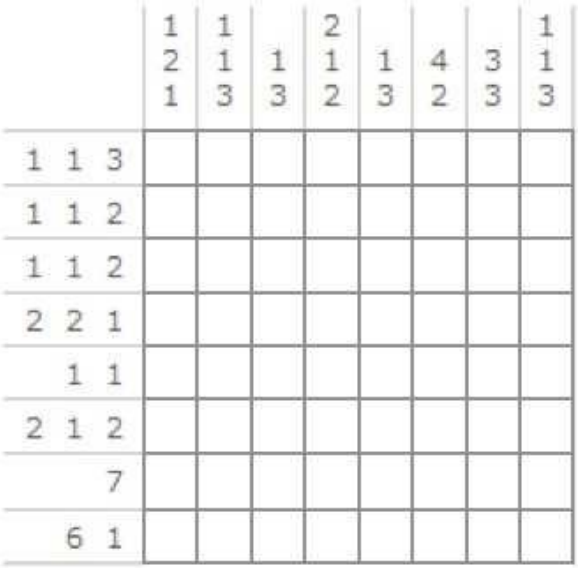


Figure 1.1: Example of an empty nonogram board

In figure 1.1 it is depicted the example of an empty nonogram board.

Before diving into the problem definition, we derive some rules which should always be applied no matter the solving method. Those are:

1. The constraints, i.e the numbers on the edges for each row and column, dictate how many continuous squares in a grid must be filled respectively. Henceforth we would call those sequences black runs.
2. Runs can neither overlap nor cross outside the range of the lengths of rows and columns (borders).
3. There should be at least one empty square between two consecutive runs in order to distinguish between two different constraints.
4. Runs follow constraints in order, e.g. if we have the constraint 2, 3, 6, we can not fill first 3, then 6, then 2.
5. Above mentioned rules are both applied to rows and columns.

Figure 1.2: Example of a Row Satisfying ”2 3 2” Constraint

The puzzle is solved once all the constraints in each row and column are satisfied. Figure 1.2 shows an example of satisfied row constraint. It is worth noting that the solution may not be unique in general. To reduce the complexity, we would only consider puzzles with unique solutions.

Traditionally, to build a Nonogram, the artist starts with an image, then generates its projection on 2 axes and defines those as the projection of those images.

Chapter 2

Literature Review

2.1 P vs NP [1]

The Nonogram is an NP-Complete problem, which makes it a perfect candidate for doing AI on it. A problem is defined as NP-complete when it belongs to a problem of NP-Complete class. The NP-Complete class has the following characteristics:

1. It is a problem for which a Brute-Force search method may discover a solution by trying every conceivable answer and for which each solution can be confirmed fast (i.e., in polynomial time). And the Brute-Force method must have exponential complexity.

2. All problems in NP-Complete class are connected in such a way that a solution in one NP-Complete problem can be converted to a solution to another NP-Complete problem. And in general, if we could quickly solve certain NP-complete problems, we could do the same for all other problems where a given answer is simple to verify.

Since the Nonogram problem itself is NP-Complete, the process of finding a solution is not straightforward, and a general solution for Nonograms probably does not even exist. The solving agents that generate a solution on an inputted puzzle (or indicate that no solution exists for a given problem) differ not only by the implementations but also by the approaches: either solving the puzzle with a unique solution or handling cases with multiple solutions. In such cases, several heuristic-based solving agents and approaches to the problem are proposed. [5] A lot of heuristic solvers and reasoning frameworks are proposed in. Some

solving agents use evolutionary algorithms, that operate on multiple solutions to find unique ones. There are also other solutions available on the internet; however, most of the solving agents convert the problem to a Constraint Satisfaction Problem, such as Integer Programming or Boolean Satisfiability Problem (SAT). In this context, there are several assumptions that we make to simplify our paper further and to have a reasonable goal to aim for. We would make the following assumptions on Nonograms.

- The Nonograms are derived from a common understandable image that can mostly be computed easily by the human hand.
- Nonograms should not have more than one solution.
- The algorithms should exploit the patterns and figures that are commonly associated with Nonograms.
- The algorithms should perform better than exponential time complexity. Or, at least, run on the computer.

2.2 Dataset

To have a diverse set of nonogram, we have searched for a dataset. This dataset includes 8151 Nonograms, each with difficulty ranging from 1-9 on a scale of 10. Some Nonograms are unsolvable by the human hand. We have chosen the dataset because it fits reasonably well with the standards and the assumptions that we have made about the Nonograms in general. Figures 2.1, 2.2, 2.3 show solutions to some of the examples from our dataset.

These are the sample Nonogram samples from our dataset. The images have been labeled to have 5/10, 6/10, and 9/10 difficulty from left to right respectively.

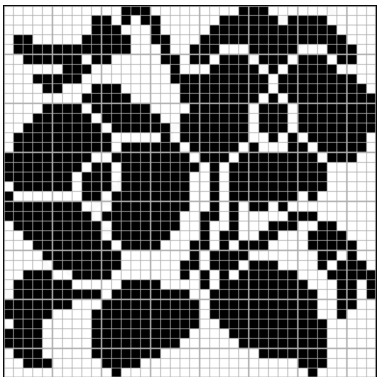


Figure 2.1: Flower

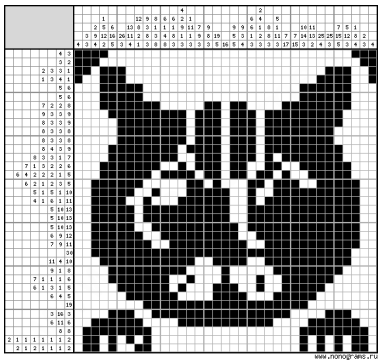


Figure 2.2: Cat

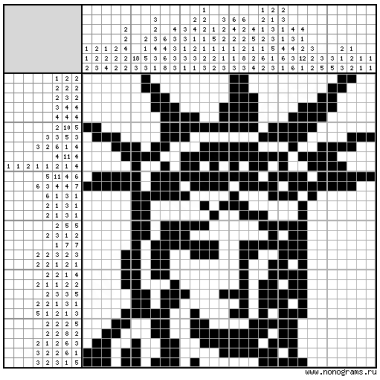


Figure 2.3: Statue of Liberty

2.3 Approaches [2]

It is believed that there are a lot of approaches to solving Nonogram puzzles. Some articles state that two algorithms are solving Nonograms effectively, each of which suggests a logic for a single solution problem. One of them is suggested by Yen et al., a method that is based on the idea of intersection. Another method is iteratively reducing search space, implemented by an online solver that changes segments. A more advanced version of the online solution allows for guessing. However, it still has to contend with the difficulty of producing a solution for puzzles with several solutions in a finite amount of time.

Several scholars have already investigated ways to solve nonogram difficulties by translating them into other issues. In the approach presented by Robert A. Bosch in 2001 [3], he addressed a nonogram problem by converting it to an Integer Linear Programming (IPL) problem: the code converts the definition of a puzzle in a program that can be used by CPLEX to solve the puzzle. However, this program only works for puzzles that have clues for all lines, and this approach only solves black-and-white nonograms. The approach introduced by Faase included the conversion of the nonogram problem into an exact cover problem, which we can then solve using Knuth's dancing-links approach. Unfortunately, here, the translated issues are frequently too vast to tackle efficiently. To tackle nonogram problems, Batenburg developed an evolutionary method for Discrete Tomography (DT), whereas Wiggers employed a Genetic approach. However, neither strategy guarantees perfect results. There is also a local reasoning framework that, when given a

partial filling, the values of some puzzle pixels can be utilized to infer. Often it is possible to solve the puzzle completely by repeating this process from a blank grid. Some strategies are based on concepts from network flows, dynamic programming, and 2-satisfiability issues [4]. Certain experimental findings show that the approach is capable of solving a variety of Nonograms without the need for branching operations, even those that cannot be resolved by straightforward logical reasoning within individual rows and columns. There are experiments conducted to solve a number of multi-solution puzzles with numerous kinds of results showing different levels of effectiveness.

Simple Depth First Search (Brute Force)

Depth First Search (DFS) algorithm iterates across a puzzle grid, filling and emptying cells progressively until a solution is found.

The algorithm begins by filling every cell on the board. In the next step grid's initial open cell is then left empty. The algorithm checks to see if a solution has been found. If no solution can be found, empty the next neighboring cell. If every available cell is unfilled, the algorithm backtracks and fills the most recent cell. Given the new configuration, the algorithm unfills all available cells. Retrace your steps and repeat the steps until a solution is found. This method uses recursive brute force to try every conceivable configuration of cells in an effort to find a solution. When all of the nonogram's restrictions are met, a solution is identified.

This algorithm is complete since it considers all possible configurations of cells, and theoretically, it is likely to find a solution, but the downside is it is a recursive algorithm and which is costly to implement as this DFS requires exponential time to solve ($O(2^{rows \times columns}$).

Simulated Annealing

One of the ways to solve Nonograms is developing a heuristic using the idea of Simulated Annealing (SA) to learn to explore various search spaces by creating a heuristic that is then going to be used in the informed search. All of the row constraints are inherently met in the board's initial random configuration. The algorithm's aim would be to fulfill each and every column constraint. In order to allow the system to transform more freely at first, the starting temperature is set to a fixed, relatively

high value. It makes sense to have a constant beginning temperature to keep things simple.

The algorithm picks a row, then transfers the blocks in that row to a different random configuration. Starting with the first row, a row is picked cyclically from the rest. By simply adding up all of the constraint violations in each row and column, the error is determined. The new answer is acceptable if the computed error is smaller than the prior error. In order to allow the system to have some sufficient time to attain ”equilibrium,” the temperature is kept constant at each stage for a period of time that is proportional to the number of columns in the puzzle. The temperature is reduced to 95% of its original value at each stage once the ”equilibrium” is attained. The process then goes back to the step of selecting neighbors. The pros of this algorithm are that it runs fast when there are not a lot of local maxima since it quickly minimizes the error. In general, the running time is independent of puzzle size and is proportional to the time it takes for the temperature to reach the lower limit. The obvious downside is that the algorithm is not complete, and it is highly probable to get stuck at some local minimum when the temperature could be not high enough for it to be able to exit.

Constraint Satisfaction Problem: Backtracking

This complete approach builds valid configurations of each row and tries to combine them in order to meet the column restrictions. In order to meet the column restrictions, this approach builds valid configurations of each row and tries to combine them. Additionally, it pre-processes the data to remove subtrees that are incorrect since they include black cells already determined from previous iterations. So each row and column represents one variable, hence if we have $n \times m$ matrix, we have $n + m$ variables in total. The algorithm assigns values only for the row variables. Each row constraint may have $\binom{g+e}{g}$ possible combinations, where g is the number of groups defining the row constraint (e.g. row constraint $(4,4)$ would mean $g = 2$) and e shows the number of empty cells left after satisfying the current row constraint. This algorithm consists of three parts, the first of which reduces the future state space by filling the cells that are accurate, meaning that in any configuration those cells are always filled. The second step involves of creating the search tree

where siblings at any depth represent all the different configurations for the same row. Hence the depth of any tree is equal to the size of the board column. The last step performs backtracking by assigning different configurations for each row constraint and checking whether the column constraints are satisfied or not. In the negative case, the subtree is trimmed and the algorithm moves to the next sibling. Algorithm iterates over the tree until a solution is found.

Chapter 3

Methods

3.1 Constraint Satisfaction Problem (CSP)

As discussed, this approach builds valid configurations of each row and tries to combine them in order to meet the column restrictions. In order to meet the column restrictions, this approach builds valid configurations of each row and tries to combine them. Additionally, it pre-processes the data to remove subtrees that are incorrect since they include black cells already determined from previous iterations. The approach is complete and utilizes $O(bm)$ memory and $O(b^m)$ processing time, where b is the branching factor and m is the maximum tree depth (i.e. number of rows). When trimmed subtrees are taken into consideration, so it typically performs better than $O(b^m)$.

3.1.1 Finding the solution

We can define three steps to achieve the solution.

1. This stage involves analyzing the layout of the board to identify the cells that are filled accurately. This is achieved by simulating the traditional human method of solving any row or column. We consider the conjunction of the leftmost complete run and the rightmost complete run. The leftmost complete run is defined in the following way. If we have j constraints: $c_1, c_2, ..., c_j$, we start from the leftmost square, and by moving right we fill the squares with the length of c_1 then proceed to c_2, c_3 leaving only one space gap in between until all constraints are satisfied. The rightmost complete run would use the same approach and instead from the left we start from the right-

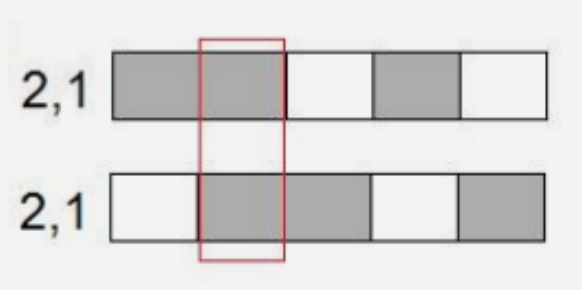


Figure 3.1: Example of leftmost complete (above) and rightmost complete (below) runs

most square and move to the left. The figure 3.1 shows the example of rightmost and leftmost runs. The intersection of these two runs defines those cells which will be filled in all cases.

2. The second step is to create all eligible row configurations. It is accomplished by gradually increasing the number of white spaces between runs until the final run reaches the very last cell. All white space combinations are calculated. For example, the figure 3.2 shows all of the combinations for a 5-cell row/column with (2, 1) constraints. The whole state space of the problem is built by com-

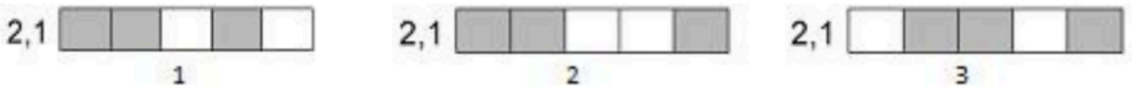


Figure 3.2: Valid combinations of a row

bining the permutations of configurations for each row, where the layer of the tree corresponds to a row's combination. The figure 3.3 depicts this, where $S_{i,j}$ is the j'th viable configuration solution that adheres to the requirements of row i. A further check is performed to see whether a node violates any cells identified to be filled by the pre-processing stage. If one is identified, that node and its subtree are removed from the state space.

3. We created a search tree with the first two stages. In order to identify a working solution, the third step employs a DFS-style backtracking search algorithm. Since any j'th layer in a tree defines a set of possible configurations for the row j, the last leaf nodes define configurations for the last row. This implies that the leaf nodes

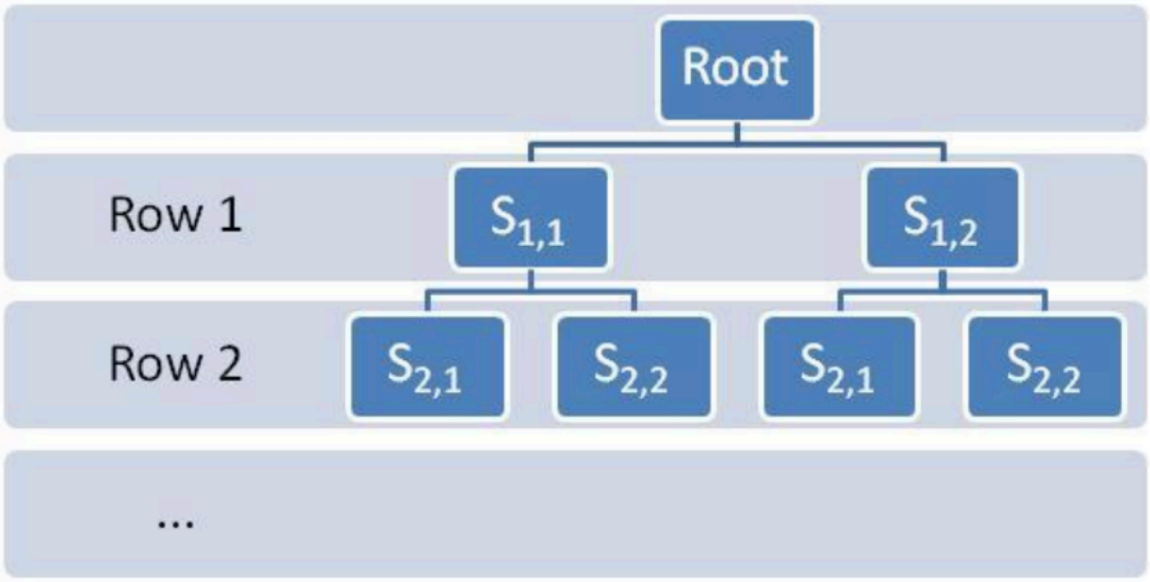


Figure 3.3: CSP tree

contain the goal state of our problem. During the search, the algorithm keeps the number of column constraints. At any step, while selecting the next node that algorithm evaluates the combination of the current node (i.e. current row) and the next node (i.e. the next row). The algorithm will prune the subtree if any column constraints aren't met, and proceed to consider different configurations. See the example in the figure 3.4.

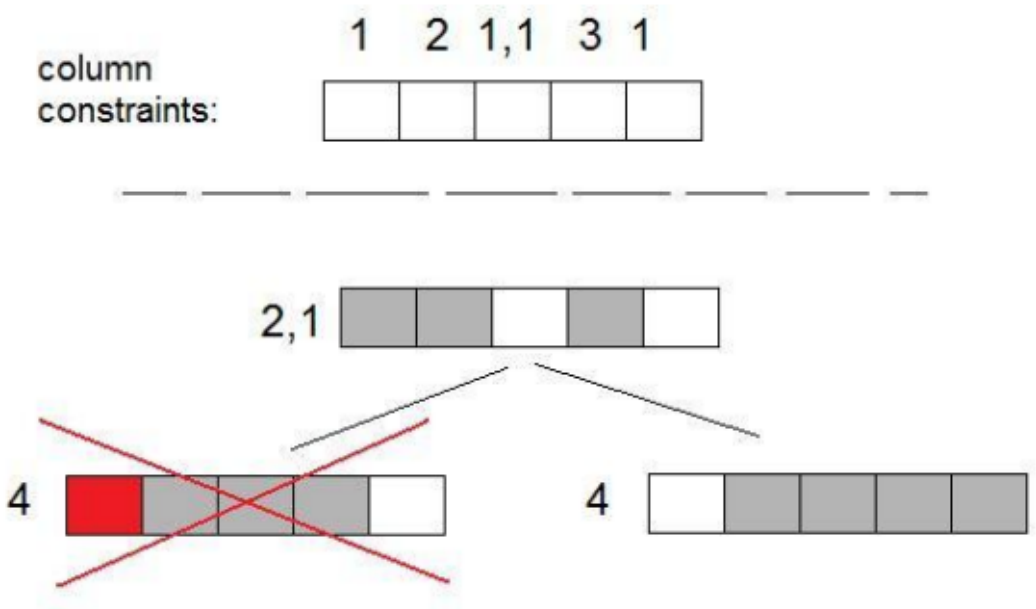


Figure 3.4: Example of pruning during a search. The algorithm pruned the left child subtree because the first column constraint is violated.

This algorithm is complete, thus, it will always find a solution. It is also effective for boards where many cells can be predetermined, as it drastically prunes the tree. When configurations violate the column constraints in the early rows rather than later rows, the tree can be pruned entirely. A drawback of this algorithm is that when the tree cannot be pruned completely using a column constraint or predetermined cells, this approach becomes exceedingly slow. Then, it mostly does a straight DFS search throughout the entire state space, which is exponentially large.

3.2 Genetic Algorithm(GA)

This algorithm is a bit tricky because without a good fitness function and low resources the algorithm would fail miserably. However, this approach favors in hard cases of nonogram, as those could not be solved with CSP. The main difference between GA and CSP is that GA starts with a random image and tries to optimize it through a fitness function. To build a GA, we need to define the following beforehand.

- Fitness function/Heuristic: This is the function that determines how close is the agent to its goal.
- Mutation: How would the agents mutate and give to potential new offsprings.
- Cross over: How 2 agents would crossover to generate a child agent.

In this article, we would implement 2 genetic algorithms and compare them to see which one performs better.

3.2.1 First Genetic Algorithm

Fitness Function

For GA, we implement the Fitness function described below.

$$BDistance(A,B) = |\sum(A) - \sum(B)| + (length(A) - length(B))^2 \tag{3.1}$$

$$EDistance(A,B) = \frac{\sum_{a \in A b \in B}(|a-b|)}{\sum(A)} \tag{3.2}$$

$$Distance(A,B) = \begin{cases} EDistance(A,B) & \text{if } BDistance(A,B) = 0 \\ BDistance(A,B) & \text{otherwise} \end{cases}$$

$$Heuristic(Image, Constraint) = \sum_{C \in Image\_Columns,\ c \in Constraint\_columns} Distance(c,C) + \sum_{R \in Constraint\_Rows,\ r \in Constraints\_rows} Distance(r,R)$$

As you can see (3.2.1) function is 0 if and only if the constraint does not violate the actual image. And so (3.2.1) will have the value of 0 if

the image matches the constraints exactly. So, this heuristic is a perfect candidate for the negative fitness function.

Next is the process of iteration. We have optimized the algorithm so that this GA runs 100s of iterations from 100 environments (meaning 100 samples have been run independently from each other) each with 30 agents, selected from a generation of 100. During each iteration, 30 agents are being selected and 2 random agents are selected and crossed over. This happens 70 times, so that the number of agents would become 100. Then 30 agents with the biggest fitness function would be selected again in the next iteration.

As you can see the Traditional GA performs awfully on the Statue of Liberty Nonogram 3.5 The 2nd column represents the heatmap of 30 agents in the 1st environment. 3rd column represents the 30 agents in the 2nd environment. The 1st column represents the sum of all 100 generations.

Mutation

The mutation is simple, we pick the agent and with mutation-rate = 0.05 probability and add random noise to the image.

Crossover

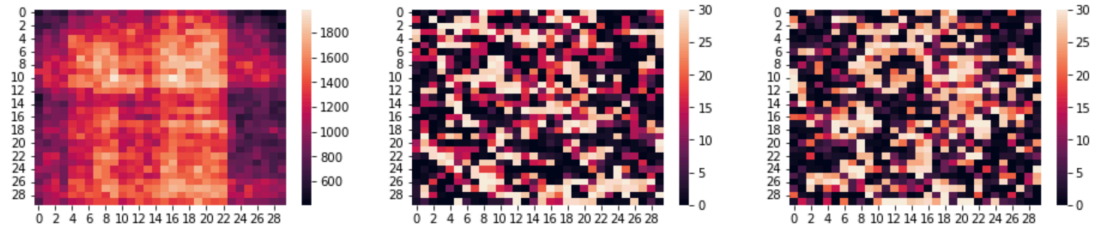
There are several ways we can define the crossover

- Crossover by the vertical and horizontal line(i.e. for 2 image and a line, left side is taken from 1st parent and 2nd from the second)
- Crossover by subsquare: The subsquare is cut from the parent and put in the child offspring, the rest comes from the 2nd parent

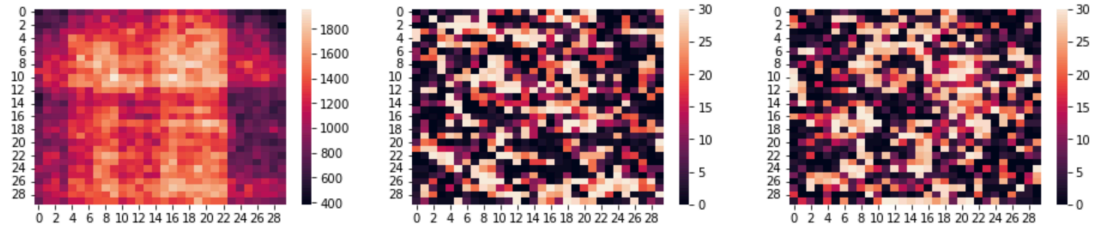
We have shown an example, which uses the 2nd method

3.2.2 Creative Genetic Algorithm (CGA)

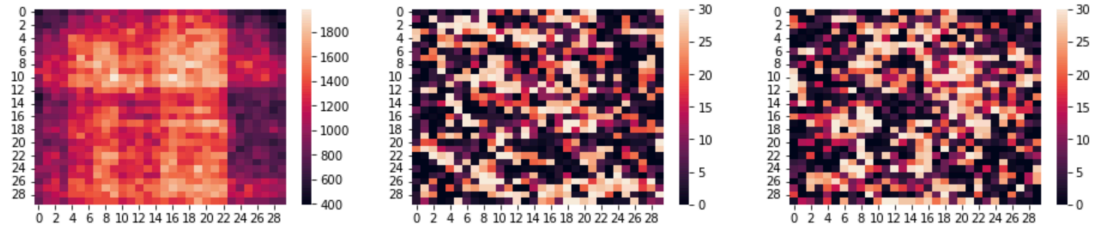
In this example, we have an agent consisting of 2 parts: column subagent and row subagent. The column subagent is constructed and chosen in such a way that it does not violate any column restrictions. Likewise, the row subagent: with row restrictions.



```
461.5333333333333
```



```
455.0
```



```
457.0
```

Figure 3.5: Genetic algorithm on Statue of Liberty nonogram (Fig. 2.3)

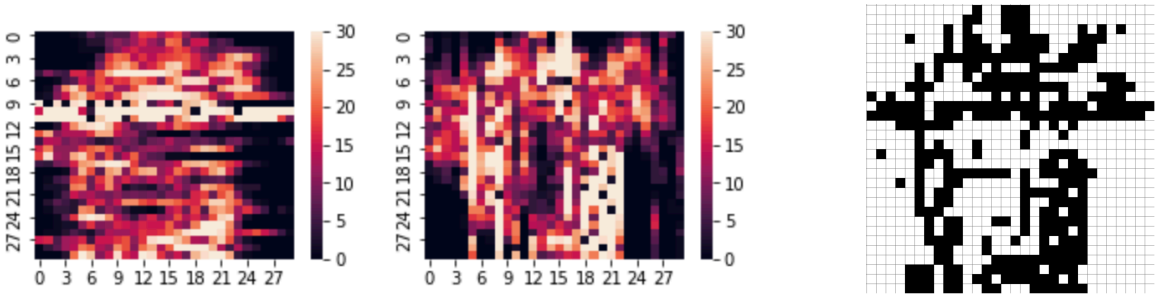


Figure 3.6: Creative genetic algorithm on Statue of Liberty nonogram (Fig. 2.3)

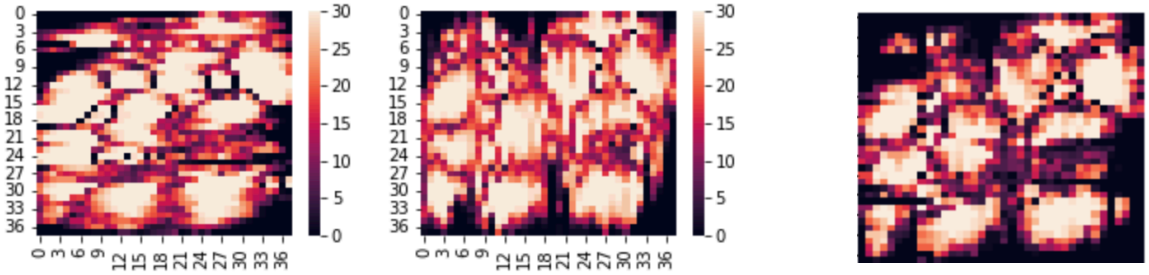


Figure 3.7: Creative Genetic algorithm on Flower nonogram (Fig. 2.1)

Fitness Function

The overlap of the Row subagent and Column subagent (i.e. how much of those 2 parts of the agent have common elements)

Mutation

During each crossover, a randomly chosen constraint will change its position to a new place.

Cross over

Crossover Rows of Row subagents and Columns with Column subagents.

In comparison with GA, CGA has fewer iterations and only one environment. 3.6 and 3.7 are examples of CGA implemented on those difficult Nonograms. In those Images, the left represents the Row SubAgent, the middle: Column agent, and the right: is the post-processed output of those agents.

Chapter 4

Results, Comparisons Conclusion, and Further work

4.1 Results

Constraint Satisfaction Problems

This algorithm will always find a solution since it is complete. Additionally, because it it is expected to significantly prune the tree, it is highly effective with boards where many cells may be accurately predetermined. Since it may also prune whole subtrees, it can solve boards more effectively when configurations do not satisfy the column constraints in the early rows as opposed to later rows. This means that is desirable to fail in the early rows than later deep in the tree. but on the other hand, when the full tree cannot be pruned using predetermined cells or column constraints, this approach becomes exceedingly slow. Then, it mostly does a straight DFS search throughout the whole state space (which is our worst-case scenario and is exponentially large). We run the algorithm on approximately 60 problems each with an increasing number of total cells.

Figures 4.1, 4.2, 4.3, 4.4 show some of the images that the algorithm has generated and we got complete solutions.

The figure 4.5 shows the runtime complexity of each part of the CSP algorithm. We don't have a smooth curve, which means that the algorithm does not always run fast on small-sized boards. Algorithm runs on 300 cell board much faster than on 50 cell board.

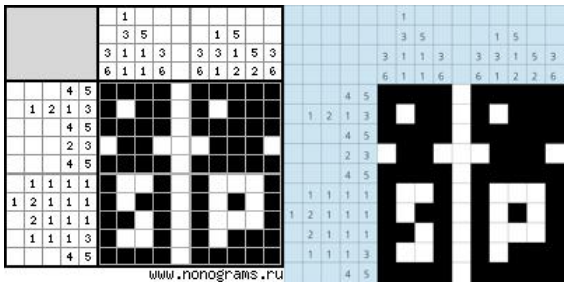


Figure 4.1: Salt and Pepper

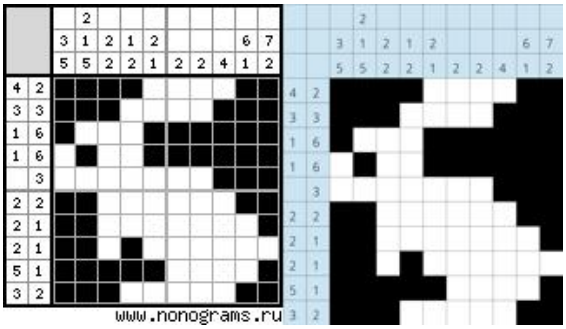


Figure 4.2: Hare

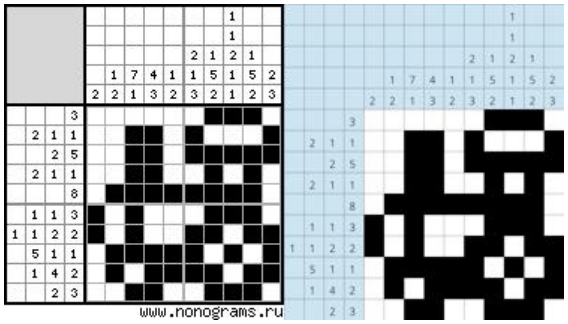


Figure 4.3: Train

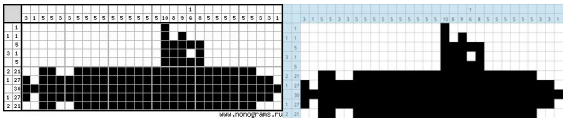


Figure 4.4: Submarine

Figure 4.5: Runtime complexity of each part of CSP algorithm

Genetic Algorithms

We can confirm that the GAs we used need further improvement and analysis. It tries to converge to a solution but fails to do so. As we have progressed, we have come up with a different Genetic Algorithm approach which is very promising, considering it almost solved the Statue of Liberty example, which rates as one of the most challenging Nonograms in the dataset. CGA is really a simple genetic algorithm, so it combines the CSP and GA concepts in a creative way. As you can see, the column agents capture column constraints pretty well and row agents cover the rows in a good way. Even though GAs did not give us any solution in those two cases, the CGA far outperformed GA as you can see in the post-processing of the Statue for Liberty, for example **??** and 3.5. By visual comparison, we can see that the Genetic Algorithm tries to get the center before moving to the edges, in contrast, the Creative Genetic Algorithm tries to understand rows and columns all on its own.

4.2 Conclusion

As we can see, there are a lot of ways to approach NP-complete problems. All the algorithms have both pros and cons as discussed in the paper. All algorithms implemented so far require resource-intensive tasks for computers to compute. In general, we considered two approaches. The CSP solves all easy problems, but may potentially require a lot of time or even fail to solve puzzles of higher difficulties (e.g. problems with 8-10/10 level of difficulty from our dataset). We have demonstrated a lot of potential in solving Nonograms with all types of difficulties. There may potentially be other advanced algorithms (CSPs with advanced logic or Neural Networks) that vastly complicate the algorithms, however considering that this problem is NP-Complete, it can be the case that no Complete and Polynomial solution will not even exist.

Bibliography

[1] J. Hartmanis. “Computers and intractability: A guide to the theory of NP-completeness”. In: *SIAM Review* 1 (1982), pp. 90–91.

[2] Hui-Lung Lee Ling-Hwei Chen Min-Quan Jing Chiung-Hsueh Yu. “Solving Japanese Puzzles with Logical Rules and Depth First Search Algorithm”. In: *Proceedings of the Eighth International Conference on Machine Learning and Cybernetics* 5 (July 2009), pp. 2962–2967.

[3] amp; Azevedo-F. Mingote L. “Colored nonograms: An integer linear programming approach. Progress in Artificial Intelligence”. In: (2009), pp. 213–224.

[4] I.-Chen Wu; Der-Johng Sun; Lung-Ping Chen; Kan-Yueh Chen; Ching-Hua Kuo; Hao-Hua Kang; Hung-Hsuan Lin. “An Efficient Approach to Solving Nonograms”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 5 (March 2013), pp. 255–262.

[5] Pomeranz D.-Rabani R. Raziel B. Berend D. “Nonograms: Combinatorial questions and algorithms”. In: *Discrete Applied Mathematics* 169 (2014), 30–42. DOI: https://doi.org/10.1016/j.dam.2014.01.004.