Group 6 - IT1360E Course Project Report: Nonogram Solver

Phan Duc Hung · Nguyen Viet Minh · Truong Gia Bach · Pham Quang Trung

January 2023

1 Introduction

Nonograms are Japanese picture logical puzzles in which cells in the grid must be colored or left empty according to the numbers at the sides of the grid to disclose a hidden art-style picture. In this puzzle, the number of numbers at the side indicates how many blocks of consecutive shaded cells are, whereas the numbers measure the length of the blocks. For example, the clue '1 3 2' would mean there are 3 blocks of shaded cells with length 1, 3, and 2, respectively, and there is at least 1 empty cell between successive blocks. This puzzle can be modeled to be a constraint satisfaction problem (CSP). The CSP is composed of a finite set of variables, each variable is associated with a domain, and a set of constraints. The solution of CSP is a legal assignment in which each variable is assigned a value without violating any constraints. Nonogram is proved to be an NP-Complete problem[4]

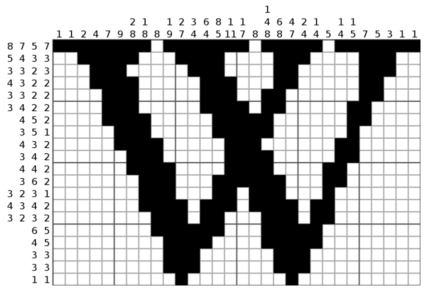


Figure 1: An example of a nonogram puzzle (source: wikipedia.com)

2 Solutions description

2.1 Search Problem Formulation

2.1.1 Components of a state

Each state is composed of 4 parts:

- size: Size of the puzzle ($m \times n$), m is the number of rows in the puzzle, n is the number of columns, the size is defined and constant through out the execution of the algorithm
- row_constraint: the linked list that contains the information of all the rows in the puzzle
- column_constraint: the linked list that contains the information of all the columns in the puzzle
- current_matrix: the matrix that contain the determined (as empty or colored) cells in the puzzle grid, with -1,0,1 denotes empty, undetermined, and colored, respectively.

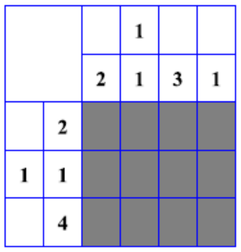
2.1.2 Initial state

: At the beginning, current_matrix is the ($m \times n$) matrix with all elements equal 0. The DFS tree of the puzzle is created based on the row information of each row.

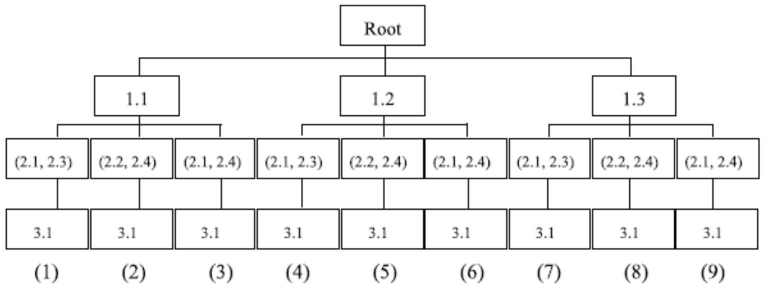
2.1.3 Conversion to Graph Theory Problem

A $(m \times n)$ Nonogram puzzle with m rows and n columns can be viewed as a Graph Theory Problem by constructing a tree using the constrains of the rows as follow:

1. Generate all possible solutions of each row, each node of depth i $(i \leq m)$ represents one valid configuration of i^{th} row
2. The root is the initial condition of the grid, or the grid with all empty cells
3. Each node of depth i $(i < m)$ has the nodes representing all possible configuration of $(i+1)^{th}$ row as its children
4. Each possible solution corresponds to a path from root to a leaf node



(a) A puzzle problem.



(b) The DFS tree of (a).

Figure 2: An example to illustrate the DFS algorithm for solving nonogram[1]

2.2 The intuition of the solution

To solve the puzzle, one needs to determine which cell needs coloring and which cell should be left empty. The solvers often cross mark or color the cell which they are certain to be empty or shaded, respectively. The puzzle can be solved by reasoning on one single row or column at a time, then try the next row or column, repeating the process until a solution is found. The more difficult puzzles may require reasoning on several rows or columns at a time.

2.3 DFS (Depth First Search)

2.3.1 DFS vs BFS

Being exhaustive searches, BFS and DFS will traverse the tree representing the puzzle eventually. Hence, they will certainly find the solution to the puzzle. BFS will visit the all the nodes of the same depth first, then move to the deeper nodes. Therefore, before reaching a solution (a leaf), BFS will have to visit almost all the nodes whose depth is less than m. In the worst-case scenario, DFS will have to visit almost every node in the tree. However, the worst case does not happen frequently. Moreover, BFS will need the extra memory to store currently valid solution, while DFS will abandon the incorrect solution it encounters and free the need for this extra memory. In conclusion, DFS is the more appropriate search technique.

2.3.2 Chronological Backtracking

Chronological backtracking (CB) is a depth first search-based algorithm and commonly used for solving CSP. The algorithm chooses one variable at a time and backtracks to the last decision when it becomes unable to proceed.
We use row information to build a DFS tree, and the column information is used immediately to do verification when a row is used to create a layer. It means that every layer of the DFS tree is composed of row information, and all nodes in each layer are the possible solutions for the corresponding row[1]

For optimizing the running time and searching space, we will have functions:

- Finish the row with white blocks when all the black blocks are filled in
- Filled in consecutive black blocks based on the information of the row given
- Filled in with a white block when finish filling in consecutive black blocks

Running time: $\mathcal{O}\left(2^{\frac{m\times n}{2}}\right)$

2.4 Chronological Backtracking with Logical Rules filter

For this algorithm, we use the results from [1]
Although chronological backtracking will find the solution to a nonogram puzzle eventually, it is time-consuming, here we use the LR to raise the processing speed In a general nonogram, we will usually paint those cells which can be determined immediately at first. After that, the rest of the cell will be solved by guess. The algorithm is based on this fact. At first, a puzzle will be solved by logical rules until logical rules cannot be applied. Then, CB will be used to visit a possible solution (an internal node) in a row and the column constraints are used for verification. All logical rules will be applied first before we used CB to visit the next node. LR and CB will run alternately and recurrently until a solution is obtained.

2.4.1 Logical Rules

In this phase, eleven rules with a concept of range of black run are used. These rules are used to determined which cells should be colored or left empty, and to refine the ranges of the black runs. In the beginning, all cells in a puzzle are considered as unknown. The eleven rules are executed sequentially and iteratively. In some iterations, some unknown cells will be determined or the ranges of some black runs are refined. If no unknown cell is determined and no black run's range is changed, we will stop using LR and start to apply CB algorithm.

Definition 2.1 (Run range)**.** The position where the black run may be placed. In this report,the range of the black run j will be written in the form (r_{js}, r_{je}), where r_{js} stands for the left-most possible starting position of run j and r_{je} are the right-most possible ending position. This means the black run j can only placed between r_{js} and r_{je}

Initial run range estimation In each row of the puzzle, let k be the number of the black run and n be the number of cells. The cells in the row are indexed from 0 to $n-1$. The following formulas are used to determine the initial run range of each black run in the row:

- $r_{1s} = 0$
- $r_{js} = \sum_{i=1}^{j-1}(LB_i + 1), \forall j = 2, ..., k$
- $r_{je} = (n-1) - \sum_{i=j+1}^{k}(LB_i + 1), \forall j = 1, ..., k-1$
- $r_{ke} = n-1$

where LB_i is the length of the black run j.

Rule 1 For each black run, those cells in the intersection of all possible solutions of the black run must be colored. Those cells can also be viewed as the intersection of the left-most solution and the right-most solution of the black run.

Rule 2 When a cell is not contained in the run range of any black runs, it should be left empty.

Rule 3 For each black run j, when the first cell $c_{r_{js}}$ of its run range is colored and covered by the run ranges of other black runs, if the length of those black runs are all one, cell $c_{r_{js}-1}$ will be left empty. Similarly, when the last cell $c_{r_{je}}$ is colored and covered by the ranges of other black runs, and the length of those black runs are all one, cell $c_{r_{je}+1}$ will be left empty

Rule 4 There may be some black segments in a row. If two consecutive black segments with an unknown cell between them are combined into a new black segment with length larger than the maximal length of all black runs containing part of the two segments, the unknown cell will be left empty.

Rule 5 Empty cells may obstruct the expansion of some black segments. For a black segment covered by several black runs having the same length and overlapping ranges, if the length of the black segment equals to the length of those black runs, the two cells next to two end of the segment will be empty.

Rule 6 For 2 consecutive black runs j and $j+1$, the start (end) point of run j should be in front of the start (end) point of run $j+1$

Rule 7 There should be at least one empty cell between two consecutive black runs, we will update the range of black run j if cell $c_{r_{js}-1}$ or $c_{r_{je}+1}$ is colored

Rule 8 For each black run j, find out all black segments in (r_{js}, r_{je}). Denote the set of the black segments by B.
For each black segment i in B with start point i_s and endpoint i_e. If $(i_e - i_s + 1)$ is larger than LB_j, set

$$\begin{cases} r_{js} = i_e + 2, \text{if black segment i only belongs to the former black runs of j} \\ r_{je} = i_s - 2, \text{if black segment i only belongs to the later black runs of j} \end{cases}$$

Rule 9 For each black run j, find the first colored cell c_m after $r_{(j-1)e}$ and the last colored cell c_n before $r_{(j+1)s}$, color all cells between c_m and c_n, and set

$$\begin{cases} r_{js} = m - u \\ r_{je} = n + u \end{cases}$$

where $u = LB_j - (n - m + 1)$

Rule 10 For each black run j, find out all segments bounded by empty cells in (r_{js}, r_{je}). Denote the number of these segments to be b and index them as $0, ..., b-1$

Step 1 Set $i = 0$

Step 2 If the length of segment j is less than $LB_j, i = i + 1$ and go to step 2. Otherwise, set r_{js} equal to the start index of segment i, stop and go to step 3.

Step 3 Set $i = b - 1$

Step 4 If the length of segment i is less than $LB_j, i = i - 1$ and go to step 4. Otherwise, set r_{je} equal the end index of segment i and go to step 5.

Step 5 If there still remain some segments with length less than LB_j, for each of this kind of segment, if the segment does not belong to other black runs, all cells in this segment should be left empty.

Rule 11 This rule is designed for solving the situations that the range of black run j does not overlap the range of black run $j-1$ or $j+1$

2.4.2 Chronological Backtracking with Logical Rules filter

As mentioned in section 2.3.2, we will use row information to create a DFS tree, and the column information to verify. Using LR as a filter, we can prune a considerable numbers of node in DFS tree, because we won't have to consider all configurations if some cells in the row are determined. For example, for a clue '1 3 2' in the row with 10 cells, if the first cell is determined as empty, there will be only 35 configurations of the row instead of 56 configurations.

2.5 Constraint Programming

Constraint programming is an algorithm using logical rules to solve constraint satisfaction problems(CSP). The algorithm uses the row information to reduce the search space.

In this problem, the algorithm, at the beginning, finds all the possible solution of each row and column. Then, it will decide the order in which the of the rows and columns to be handled next with the row or column with least possible solutions be handled first. For those row or column with less solutions will has the higher possibility of finding the celled which can be colored immediately than those with more solutions. After each iteration, it will fill more undetermined cells. The priority of the row or column will be cleared after each iteration and updated at the top of the loop. The updating process will guarantee the number of cases to be solved in each iteration be the least.

For many cases that the problem has multiple solutions, using only constraint programming will not be able to solve. For example, when there is not enough information to remove possibilities that happen on that row or column, the constraint programming can get stuck and run forever.

Applying backtracking will help to add information of the block that is uncompleted, given possibilities of that row or column so that the program can continue to run and find the solution that satisfies. If the solution is wrong, backtrack and add the contradictory condition to that block (white to black or black to white).

This is our improvement for the problem of dealing with multiple solutions Nonogram problems, when mostly all the algorithms recently can only apply on solving single solution problems (the problems that information is given enough that we can solve logically).

2.6 Simulated Annealing

Since the puzzle is a CSP, different local search algorithms are considered in order to solve the puzzle. In our project, we tried to use the simulated annealing (SA) algorithm to solve the problem. With the desired goal condition is a board with rows and columns that are mostly similar to the constraints, the SA algorithm attempts to find the global optimum without getting stuck in the local optimum. The SA algorithm takes into account three main factors: the initial and neighbor state, the cost function and the cooling schedule. We have to carefully choose the suitable ones for each of these factors for the program to have the best probability to converge to the global optimum:

2.6.1 The initial and the neighbor state:

In this problem, every state is a board with size $m \times n$ and every cell has value 1 for being black and -1 for being blank. We consider 3 ways to choose the initial board and its neighbor region respectively.

In the first method, the initial board automatically has the row constraint satisfied, which means we just put all the black blocks with correct length to every that satisfy two consecutive blocks separated by at least one blank cell. In this way the cost function only takes into account the column constraints. Next, we consider the neighboring states of this initial one. All neighbor boards still need to maintain the initial condition. The way to do this is that we just choose a random black block at a random row and shift it to the left or to the right a number of positions at random.

In the second way, all the states only have one condition to maintain: the number of black cells (and also the number of blank cells of course). Therefore, the initial state is a board with all the back cells placed randomly into it. Then, this condition can be maintained in all the neighbor boards by choosing a random black cell and a random blank cell and then swap position the two. Note: we can do this easily by using a list for all the black cells and a list for all the white cells.

In the final way, literally no additional conditions need to be considered. A random board is chosen for the initial state with a cell being black or blank at equal probability. The neighbor board is obtained by choosing a random cell and changing its value (i.e. black cells become blank and blank cells become black).

The last one performed best in our experimental running.

2.6.2 The cost function:

The cost function calculates how "good" the current state is. In a typical CSP, the cost function represents how different the current state is against the goal state which stands for the row constraint and the column constraint. If the different function arrives at 0, that means we found the global optimum (i.e. no difference found between the current state and the constraint). We proposed two cost functions which base on Manhattan distance and Euclidean distance. In our experimental results, the Manhattan distance works better as the cost function.

2.6.3 Cooling Schedule:

In order to escape the local optimum and find the global optimum, SA algorithm accepts many next states with worse value according to the cost function in the early stage although these situations appear less and less toward the end. This comes with an acceptance probability of a worse state that converges to 0 as T (temperature) converges to 0. The choice of cooling schedule is the most crucial factor against the probability of correctness in a SA algorithm. In our work, we tried three different cooling schedule.This includes: logarithmic cooling schedule, geometrical cooling schedule and linear cooling schedule. In the end, the linear cooling schedule with temperature T decreases at a constant rate from 1 to 0.

3 Problems/Issues/Difficulties occurring during the execution of the project and the solutions

3.1 Using inappropriate approach to the problem

At first, we proposed using 3 algorithms for solving this kind of puzzle: BFS, DFS and A* algorithms. However, as mentioned in section 2.3.1, BFS is not a suitable approach to this problem. On the other hand, A* algorithm is a graph traversal and path search algorithm. A* algorithm is used to find a path to the given goal with smallest cost, it is mostly used for searching graph problems. In our problem, because the solution is unique and there is only a global minimum that satisfies the condition, so that all the local minimums will be unacceptable. Also, the heuristic function to estimate the path to the goal is very hard to evaluate (the solution is kept unknown until it is found). So that we can not apply A* to solve this problem. Therefore, we decided to find different approaches to this problem. The alternative approaches are Chronological Backtracking, Constraint Propagation, and Stimulated Annealing.

3.2 Executing the Simulated Annealing

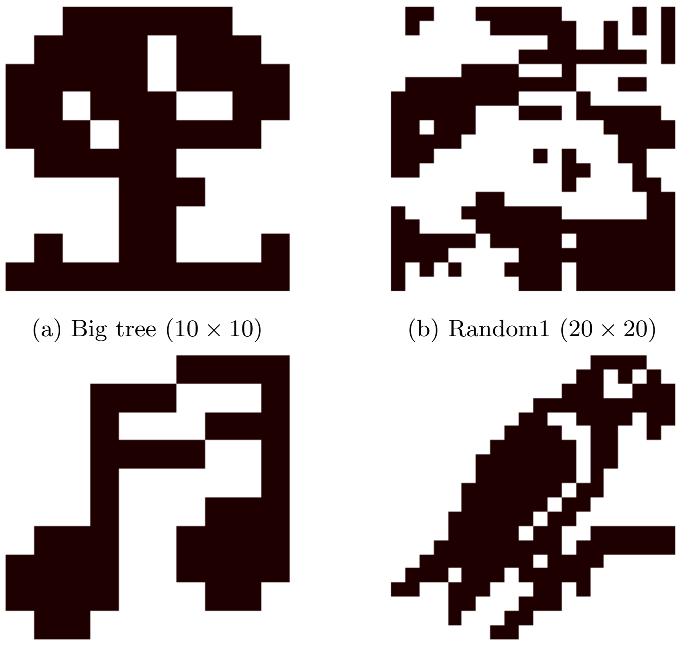
A good simulated annealing algorithm requires a good combination of choosing the suitable cooling schedule, the initial state, the neighbor states and a good cost function. This can only be improved through two factors: the understanding of the problem faced and the experience in using and evaluating different types of cooling schedule, neighbor states, and cost function. To overcome this obstacle, we decided to try different types of all three factors (as described above), recording the running time to determine the optimal one. However, this remains what we need to work on in the future. As for now, we find it hard to come up with the right one to optimize our SA algorithm to solve bigger puzzles at faster running time.

4 Experimental results

To do experiment, we collect 50 puzzles. They come from [3] and our test generate file: test_generator.py. The experimental results are shown for DFS, Chronological BackTracking with Logical Rule filter (CBLR), Simulated Annealing (SA), and Constraint Programming (CP).

Testing condition and methodology

- ROG Rephyzus R14 (AMD Ryzen 9 5900HS, 3301 MHz, 8 Cores, 16 Logical Processors, RAM 16 GB, Windows 11, SSD)



(a) Big tree (10×10) (b) Random1 (20×20)

(c) Musical note(10×10) (d) Parrot (20×20)

Figure 3: Test images

- All algorithms are run on Jupyter Notebook 6.4.5 with Python 3.10.9
- Only start the algorithm when OS is stabilized (CPU usage is less than 5%, RAM usage 6.7 GB, Disk usage less than 2%). Airplane mode is turned on to prevent random disruption.
- All test file are created prior to testing to reduce randomness in I/O activities
- Step of each run:
  1. Restart the computer
  2. Initialize Jupyter Notebook enviroment, open Task Manager, wait about 5 minutes(for the OS to stabalize)
  3. Execute algorithm on 50 pre-generated puzzles.

Puzzle size	DFS	CP	CBLR	SA
5×5	31.54	2.01	7.86	$4.76 - 56.20$
10×10	30426.24	5.00	10.70	$159.00 - 397.40$
15×15	$>$ 1 hour	28.00	28.00	X
20×20	$>$ 1 hour	52.00	32.01	X
25×25	$>$ 1 hour	448.84	96.17	X
50×50	$>$ 1 hour	163563.57	308.41	X

Table 1: The comparison of the experimental results among our methods (unit: milisecond)

Did the algorithms find a solution ?

For small puzzles ($5 \times 5, 10 \times 10$), all the algorithms find the correct solutions. As we expected, DFS performs the worst on running time. Simulated Annealing's performance is unclear on the large puzzle as it quits before find a solution because the maximal operations of SA depends on the decreasing speed of temperature on the cooling schedule. Therefore, the maximal number of operations is difficult to adjust.

5 Conclusions and discussion

5.1 Analytic conclusions

In our project, we used 4 different methods for solving nonogram puzzle. Of the four, CP and CBLR appear to be the most efficient algorithms. For small puzzle ($5 \times 5, 10 \times 10$), CP seems to solve the puzzle faster. When the puzzle is larger, CBLS is more efficient than CP. One possible explanation is that most nonogram puzzles are meaningful, therefore they can be solved quickly only by using logical rules. With small puzzle, however, the search space is narrow, the pruning scheme takes more time to execute than simply visiting invalid node and backtracking. For puzzle with multiple solutions, only DFS and CBLR can determined all the solution while CP gets stuck in a loop. However, DFS is time-consuming because it will have to consider every possible solution of the problem. In conclusion, CBLR is the most suitable algorithm for this problem.

Due to randomize process, the running time of Simulated Annealing lies in a larger domain. In some experiments, the running time is less than that of CBLR. The reverse is true in some other experiments. We can see that SA algorithm we built is not stable. Therefore, we make no conclusion about the efficiency of SA algorithm.

5.2 Proposals for improvement

1. The Simulated Annealing algorithm still has some problems with the cooling schedule. We think in the future we can come up with some kind of an adaptive cooling schedule that adjusts properly with different problem sizes and different situations of evaluating the cost function. It seems that these different scenarios require different step sizes in the decreasing of the temperature(T).

2. The Chronological Backtracking using Logical Rules is still not optimized. The technique to find which rules can be adapted first can still be improved to get faster running time. Moreover, we can still optimize the code of each rule to achieve the algorithm's running time presented in[1].

Reference

1. *Chiung-Hsueh Yu, Hui-Lung Lee, Ling-Hwei Chen (2011) An efficient algorithm for solving nonogram, Applied Intelligence, 2011 - Springer*

2. *Wen-Li Wang, Mei Huei Tang (2014) Simulated Annealing Approach to Solve Nonogram Puzzles with Multiple Solutions, Complex Adaptive Systems, Publication 4*

3. *For data test: https://nonogramskatana.wordpress.com/*

4. *Ueda N, Nagao T (1996) NP-completeness results for NONOGRAM via parsimonious reductions. Technical report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, May 1996*

Contributors

1. Phan Duc Hung 20214903
   email:hung.pd214903@sis.hust.edu.vn

2. Nguyen Viet Minh 20214917
   email: minh.nv214917@sis.hust.edu.vn

3. Truong Gia Bach 20210087
   email: bach.tg210087@sis.hust.edu.vn

4. Pham Quang Trung 20214935
   email: trung.pq214935@sis.hust.edu.vn