Solving nonograms using A*-GAC

Mikael Kvalvær

September 20, 2017

0.1 Representation of the problem

An aggregate representation was used for this project. Aggregate representation looks at rows and columns as the fundamental units. The preprocessing takes somewhat more time, but the advantage is good computational performance and simpler code. Next follows a short description of the variables, domains and constraints employed in the representation.

variables: Consider the figure of the rabbit. The first row consists of the feet of the rabbit, where the feet touches the ground. The segment lengths are 4 and 5. The variable here is defined as the entries in the row. Therefore, we have a single variable for the first row, with the value $[0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0]$.

In general, we have rows with dimensionality M and columns with dimensionality N. Thus, a variable for a row is a vector $x \in \{0, 1\}^M$ and for a column a vector $y \in \{0, 1\}^N$.

domains: For each variable x the domain $\mathcal{D}(x)$ is defined as all feasible values for the variable. Consider a column with length N and with n segments, each with length $l_1, \ldots, l_n$. The total length is $L = \sum_i^N l_i$. For the first segment, the first index the first cell can be placed is 0. The last index is found by $M - (n-1) - L$ because the expression $L + (n-1)$ represent the minimum space occupied by all cells. For example, we can see in the first row that $M = 20$, $L = 4 + 5 = 9$ and $n - 1 = 1$. From this we can deduce that the last index that the first segment can start on is $20 - 9 - 1 = 10$. A recursive procedure is applied that finds the domains for each variable

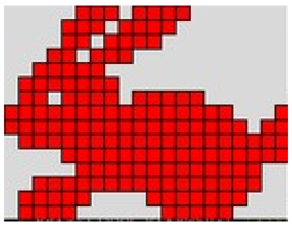

Figure 1: Example of a solved nonogram-puzzle. The red boxes marks cells that are common for both column and row.

constraints: The discussion above showed how the domains of each variable is found. We can restrict the domain size by enforcing the domain size for each row and column. This will generate a list of possible rows for each row and a list of possible columns for each column.

Furthermore, each column and row must agree with each other on which cells are marked and which are not. Any row that marks a cell that is for certain not in the corresponding column must be removed. Similarly, any column that marks a cell that is for certain not in the corresponding row must be removed. Pseudocode of this filtering algorithm is shown below.

0.2 Heuristics used

choice of h function

A state consists of candidates for each row and column. Let nc_i be the number of column candidates for column i. Similarly, we define nr_j as the number of candidate rows for row j. Then, h is defined as

$$h = \sum_{i \in \texttt{cols}} log(nc_i) + \sum_{j \in \texttt{rows}} log(nr_j)$$

Logarithm's are used to prevent numerical overflow. This could occur if there are a large number of rows / columns or if the range is sufficiently large. If any $nc_i = 1$ then they will not contribute to the heuristic function because $log(1) = 0$.

Choice of variable on next assumption

The choice of variable on which to base the next assumption is solely based on the number of candidates (domain cardinality) for that variable. If there are few candidates, then it will be faster to visit all possible states by assuming specific values for that variable. On the other hand, if the domain is large, it'll take mroe time to visit all subsequent nodes. Therefore, the line with fewest candidates (greater than 1) is chosen. The algorithm below implements this and returns the line index with smallest domain.

```
def fewest_candidates(lines):
  lengths = np.array(
    [len(each) for each in lines])

  lengths[lengths == 1] = max(lengths)

  return np.argsort(lengths)[0]
```

Listing 1: Algorithm used to find the row / column with smallest domain.

This heuristic is important because if there are a lot of candidates for a specific line, then it will be computationally expensive to generate assumptions on each one of them. Since only one candidate will be correct for each line, we would like to reduce the number of assumptions made. That is the main reason for implementing the `fewest_candidates` algorithm to reduce the number of assumptions made. This made larger puzzles, such as the *reindeer*-puzzle computationally feasible.

0.3 Specialization of the A* algorithm

Several tweaks of the `A*`-algorithm had to be done in order for the nonogram solver to work. The `A*`-algorithm is implemented as an abstract base class in Python. This means that several methods had to be implemented. The methods that had to be implemented were as follows:

- `goal_fun`: Specifies if a certain state has reached the goal or not. Done by checking if $\forall_i : \|\mathcal{D}(x_i)\| = 1$ holds or not.
- `cost_fun`: Specifies the cost of going from `parent` to `child` where `parent` and `child` are Node-instances. In this case, the cost function returns a 1. The function could also be 0, but this makes it more susceptible to deep nesting of states.
- `generate_children`: This is the heart of the `A*`-algorithm. Specifies how the children are generated. Here also the domain creation + constraint enforcement is done.

0.4 Other design aspects

iterated filtering

Some tweaks were added to the function `generate_children` of the solver. The function takes in a state $S = (X, Y)$ where X represent the row candidates and Y represent the column candidates. For each row $x \in X$, we filter the columns $y_i \in Y$ based on the common indices of x. For example, if we define the entry x_1 as follows:

- $x_1[0] =$`[1,0,0,1]`
- $x_1[1] =$`[1,0,1,0]`

Then we know for sure that any candidate in x_1 must have the entries `[1,0,?,?]` because they are common in all candidates. We filter out all columns in y_0 and y_1 where their value do not agree. This will do the transformation $y_1 \rightarrow y_1'$ where y_1' is the set of candidates in y_1 that satisfies the common cells from x_1. We can next filter the different $x_i \in X$ based on the 'reduced set' $Y' = [y_1', y_2, ...]$. This procedure can be applied iteratively until we have a set Y^* that does not change when filtered on the corresponding set X^* and vice versa. This greatly increased the processing speed of the model.

0.5 Results

An overview of performance is shown in the table at the bottom of the page. The table compares the different puzzles and shows the time taken, path length, node created and nodes visited in a given puzzle.

puzzle	**time taken (s)**	**path length**	**nodes created**	**nodes visited**
cat	0.0022	1	3	2
chick	0.046	1	3	2
clover	0.048	1	3	2
elephant	0.031	1	3	2
fox	0.330	1	3	2
rabbit	0.053	1	3	2
reindeer	12.383	3	20	8
sailboat	0.076	1	3	2
snail	0.188	2	11	4
telephone	0.054	1	3	2