Using A*-GAC to Solve Nonograms

Johannes Omberg Lier   Per-Christian Berg

October 5, 2015

1 Representation

We have chosen to represent this problem by making a variable for every row and column in the matrix (nonogram). Every row and column variable has its own domain, which is every permutation of colored and not colored cells that satisfies the constraint that belongs to the variable. The constraints are simply that for the cell that is shared between one row and one column, must be either colored or not colored in both variables. Notice from figure 1 that the domain for the row shown in the figure is generated based only on the constraint on that row, the column contraints are ignored so far.

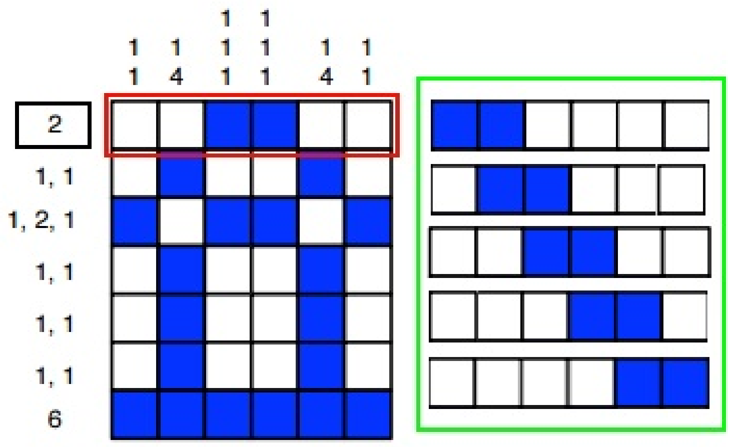

Figure 1: Representation of our problem. **Red: Variable, Green: Domain, Black: Constraint**

2 Heuristics

2.1 A*

The heuristic in the A*-algorithm is based on the total amount of variables in all the domains in the search node. The fewer total variables the closer the search node is to a solution. In other words, the search nodes with the least amount of variables in the domains, is concidered the better choice to expand.

2.2 GAC

In the GAC-algorithm we need a way to determine which search node that should be expanded. The way we do this is to expand the first variable that has more than one variable in its domain. Then we assume the value of the variable. This is done by making a copy of the search node and setting the domain of the variable to a single value. This way we avoid the situation where the algorithm reduces a domain to an empty set.

3 Specifically nonogram

This problem differs from module 2 because the constraints are based on the coordinates of the focus cell. Therefore, we needed to find a different situation where we could make use of the *makefunc*-function. Since we generate all the possible values of each row and column, we can then extract a column-row pair and check if the shared cell has the same color (black or white in our case). If the focal variable is the column and none of its values fits with any of the values for the rows, the column's domain can be reduced. We needed to implement methods for initiating the data structures needed to solve the task. Also, methods for generating neighboring search nodes was spefically created for this task.

4 Design decisions

To store all the variables we used a dictionary with a specific key. The key is a string defining if the variable is a row or a column variable and its index. This way, we can easily modify the indices throughout the search to adjust the y-axis (the list of row variables) such that the bottom row is the first. Notice from listing 1 that the domain of the first row is accessed by changing the key to

```
('row',0)
```

```
print variables[('col',0)].domain[0]
# Will print the first value in the domain of the first column variable.
```

Listing 1: How the possible variables are stored in a dictionary

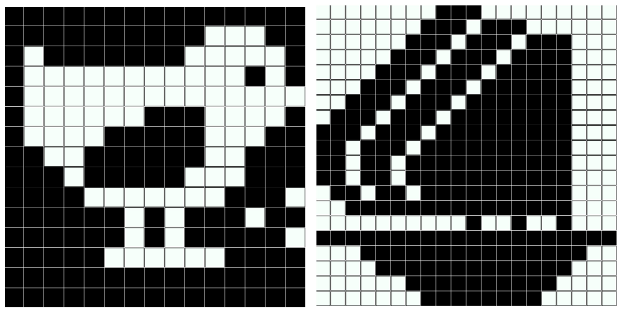

Figure 2: The result of scenario 3 and scenario 7 given in the assignment