AI programming: Module 3

Solving Nonograms using Best-First search and Constraint-Satisfaction

Bjørnar Walle Alvestad

2015

Norwegian University of Science and Technology

NTNU

Introduction

This report will describe the central aspects of the nonogram solving algorithm implemented in module 3 of the AI programming course. This module sought to represent a nonogram puzzle system as a constraint satisfaction problem, and later solve it using the A*-GAC algorithm implemented in module 2 of this course.

This module is implemented using the Java programming language. It utilizes code from module 1 and 2 of this programming course.

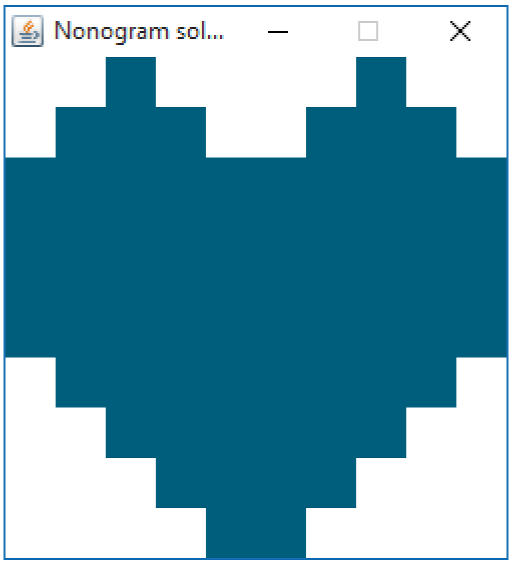

Figure 1: A solved nonogram puzzle resembling a heart.

CSP representation

Variables and domains

For A*-GAC to be able to solve nonogram puzzles, the problem needs to be represented as a CSP. In this implementation, whole rows and columns are considered variables in a CSP. The domains will be filled with all legal permutations of the specific row/column. Consider the nonogram in figure 2. In figure 3, all legal permutations of rows and columns are listed. These will be added to the appropriate variable domain. Columns are represented by the characters A-C. Rows are similarly represented by D-F.

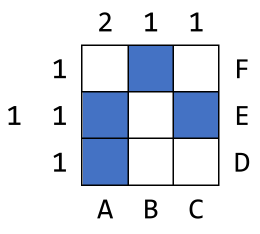

Figure 2: A simple nonogram with row and column specifications. The correct solution is marked in blue.

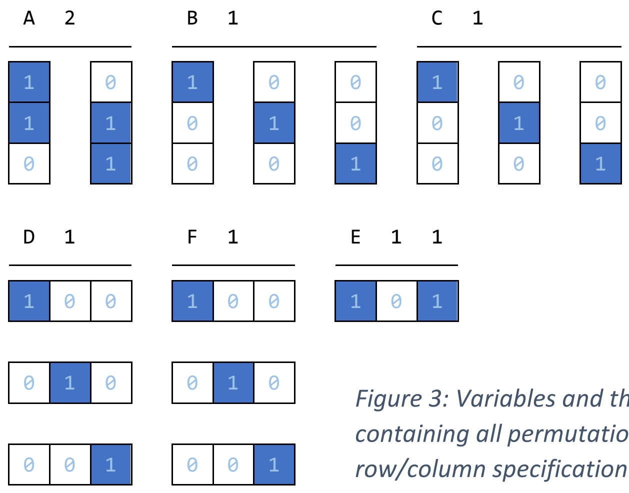

Figure 3: Variables and their domain, containing all permutations of the row/column specification.

Because of the way A*-GAC was implemented in module 2, only integer values can actually reside in the domains. An actual row/column permutation can't be stored as an object. This is solved by converting the permutation into a bit sequence, where each set bit designates a filled cell. This will effectively represent a row/column permutation as an integer, which can successfully be stored in a domain. Note that the java primitive datatype `int`, used in this implementation is a 32-bit integer. This will put a row/column size cap of 32, making this implementation support nonograms of size up to and including 32 by 32.[1]

[1] This can be extended to 64 by using the `long` primitive datatype.

Constraints

In this implementation, one constraint is generated for each cell in the nonogram. A cell represents a row/column intersection. Every constraint will check for equality between the row and column that intersects it. The constraint in cell (0, 0) will look like this: C_1`:  A  ==  D`. It will check for equality between column A and row D. Because rows and columns are converted into bit sequences and stored as integers in the domains, this evaluation can be unfavorable because it takes the whole row and column into consideration. The constraint should only check equality in the respective cell it is located. This can be accomplished by bit shifting the row and column permutation, so that the bit in question is located at the lowest significant position.

C_7	C_8	C_9	F
C_4	C_5	C_6	E
C_1	C_2	C_3	D
A	B	C	

Figure 4: Constraints are created for every cell in the nonogram.

A constraint is satisfied if the intersecting row and column agrees on whether the cell should be filled or not. This can be evaluated using the bitwise XOR operation. It returns true when the inputs differ. After shifting the correct bit in both bit sequences to position 0, the XOR operation is applied. If the least significant bit in the result is 0, the row and column agrees on the cell state. It the bit is 1, they disagree. The least significant bit can be obtained by applying the bitwise AND operation on the result and the bitmask 1.

A	**B**	**A XOR B**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Figure 5: Truth table for the XOR operation.

Consider constraint C_8. It intersects column B and row F, and should check equality at position (1, 2). B is the second column, and has index 1. F is the third row, and has index 2. By right shifting the column permutation value 2 places, and the row permutation value 1 place, the bits in question are located at bit index 0. Constraint C_8 will look like this:

$$C_8\texttt{: (B >> 2 XOR F >> 1) AND 1 == 0}$$

The general formula is:

$$C_{A,B}\texttt{: (A >> }B_{index}\texttt{ XOR B >> }A_{index}\texttt{) AND 1 == 0}$$

When a solution is proposed with only singleton sets, all constraints must still be satisfied in order for the solution to be valid. This is because rows and columns can still disagree on the filled state of a cell.

Heuristics

The A* heuristic function used in this implementation is the same as in module 2, namely the sum of all domain sizes minus 1. This will produce the heuristic value 0 when all domains are singleton sets.

The second heuristic is not specified in this implementation, as it was not necessary for the algorithm to choose a variable to base the next assumption. After the initial GAC filtering, most of the domains are already singleton sets. The default successor generation method from A*-GAC is used, which generate successors for all variables and domain values.

Subclasses and specializations

No sub-classing of the original A*-GAC code was necessary in this implementation. The class *Strip* represents a row/column permutation. It is used to generate all permutations of a row/column specification, as well as conversion to bit sequences. Rows and columns are represented in the *Strip* class by specification, segment start indices, and strip length. All possible permutations are generated as following:

```
function recursive_permutation(strip, segment_to_push)
   while strip.can_push_segment(segment_to_push)
      if segment_to_push + 1 < strip.number_of_segments then
         recursive_permutation(strip.make_copy, segment_to_push + 1)
      end if
      strip.push_segment(segment_to_push)
      permutation ← strip.make_copy
      store permutation in permutations list
   end while
end function
```

The function `can_push_segment(int i)` checks if segment at index `i` can be moved one unit to the right without causing it to move out of the strip boundaries.
The function `push_segment(int i)` moves all segments with index `i` or more, one unit to the right.

References

Downing, Keith L., *AI Programming (IT-3105) Project Module #3: Combining Best-First Search and Constraint-Satisfaction to Solve Nonograms.* [Online]