Nonograms
Lab rapport IT3105 - Module 2

Sigve Skaugvoll & Thomas Wold

September 2017

1 Introduction

The assignments was to solve nonogram puzzles - cells in a grid must be colored/blank in grid to reveal picture, using A*-GAC. We had to implement everything CSP related from scratch and reuse our generic A* from the previous lab exercise. Part of the problem was to find a good representation for nonograms.

2 Nonogram

In this report, we will give answers to the following questions:

1. What representation is used for variables, domains and constraints to represent the nonogram.
2. Explains both the heuristics used for solving the nonogram.
3. Overview of the subclasses and methods needed to specialize our general-purpose A*-GAC system to handle nonograms
4. What are some critical design decisions, to getting a performance system.

2.1 Q1: Representation

2.1.1 Nodes

In our implementation we have a subclass for nodes, which represent a fully representation of a nonongram board. the Node holds variables. Row variables and Column variables are maintained in separate lists.

Nodes maintain a heuristic we called sumDomains, which just loops through all the domains and sums up the values. That is, it sums total domains (number of row domains + number of column domains).

2.1.2 Variables

Variables in our implementation are whole rows, or columns. Our variables is a own object / class, which maintains all the possible domains for the variable. The variables them selves are responsible for generating all their possible and legal permutations.

The variables also maintains a dictionary of all the common colored or non-colored cells. if i.e cell 5 in the row is always colored in all the legal permutations and cell 3 is always blank, the dictionary will look like this : 5: 1, 3: 0

2.1.3 Domains

Out domain is a list of lists, representing all legal permutations of that row or column. The domain length corresponds to the given board size. For each list, if index is 1, that represents a colored cell on the board and 0 a blank cell.
To make assumptions about domains, all we have to do is traverse:
node $\rightarrow$ variables for node $\rightarrow$ for each variable.domain $\rightarrow$
create a new node equal to the current node, with only one (a guessed) domain for this var.

2.1.4 Constraints

As mentioned above, the variables them selves are responsible for generating domains, thus handling the constraints.

The implemented constraints are

1. Find all common colored / blank cells for variable domains.
2. When we have found common for i.e rows, then reduce the domains for each row variable. This is possible since if a column has to have a certain cell filled/blank, then for a valid row domain, the domain must also have the corresponding cell filled/blank. If not, we can remove the domain from that row. The same goes for columns, find common cells for rows, then reduce column domains.
3. A variable must have x segments in y size, $y! = \forall x$
4. Segments must appear in the same order given.
5. There must be at least one blank cell between two segments in a domain

If the CPS part does not solve the nonogram / node, A*-GAC then finds all possible assumptions for the node. That is for each variable (both row and column) in the node, create a new node for each possible domain for that variable. Then use the heuristic to find the best assumption out of those, and try to reduce domain to solve.

2.2 Q2: Heuristic function

There are two heursitics in our implementation; The nodes have one heuristic, which is the number of domains in that node, This just add number of domains for all rows and all columns.
The second heuristic is for our A*-GAC. The size of each domain minus one. The sum of all those give a bad but admissable estiamtion of distance left to reach the goal state.

2.3 Q3: Classes involved

1. AstarClass.py - Generic A* implementation.
2. Problem specific classes
   (a) NonogramBFS.py - inherits from abstract BFSclass.py ; This handles alot of the problem specific logic, such as generating successors, calculating heuristic for A*-GAC, etc.
   (b) NonogramNode.py - see section 2.1.1 - Methods such as; reduce domain, Find new common cells, after reducing domains - calculate node heuristics - filter own domains
   (c) NonogramVariable.py - see section 2.1.2 - Methods such as; generate all possible permutations of own domain. Find common cells
   (d) NonogramGui.py - (optional) Creates a GUI which shows steps and solution found.

2.4 Q4: Critical Design Decision WRT Performance

Critical design decision is to have good dynamic constraint functions, to reduce domains of variables. Thus A* generates a lot less nodes / assumptions. This improves the speed up! Generating only legal domains and not all possible and then having to filter out. This saves memory and allows for faster iterations.

References

1. Essentials of the A* Algorithm - Author: Keith L. Downing - Pub: ? - Accessed : September 20, 2018 - Online.
2. AI Programming (IT-3105): Combining Constraint-Satisfaction Problem-Solving with Best-First Search - Author: ? - Pub: August 28, 2017 - Accessed : September 20, 2018 - Online