AI Programming: Assignment 3

Eirik Vågeskar

October 6, 2015

1 Representations

The domains, variables, and constraints are represented in the following ways:

- **Variables**: Each row and column is referred to by a string. The string starts with a letter followed by an integer in base 10. The letter is either "x" or "y": An X denotes a column, and a Y denotes a row. This is because the specifications of the columns are found along the X-axis and the rows along the Y-axis.

  The integer denotes the index of the row or column, with the origin in the top left corner. The choice of of origin has been made because it is simpler to iterate over a two dimensional Python structure starting from the top left corner when printing to the console or drawing in a GUI. This is in opposition to the specification, which places the origin in the bottom left corner.

  As a consequence, the specification for the rows are read starting with the lastmost row specification first, which is denoted by "y0". The column specifications are read in the same sequence as they appear in the specification.

- **Domains**: The domain of a variable is a set of bit vectors. The bit vectors are tuples of 0's and 1's, with 1's representing a filled square. The length of the bit vectors are the width and height of the nonogram for the rows and columns respectively.

  One bit vector of a variable's domain is one of the possible legal configurations that fulfill that row or column's specification, as specified on p. 2 of the task description.

- **Constraints**: A constraint is placed on each cell on the board. The constraint specifies that the cell's value should be the same in the row and column that corresponds to this row. A constraint for the cell with coordinates (2, 3) would be specified by `Constraint(['x', 'y'], 'x[3] == y[2]', ['x2', 'y3'])`.

2 Heuristics

The heuristics of the NonogramProblem class is the same as in CSProblem, and these are also described in the report for Assignment 2.

- **Search heuristic**: The search heuristic is the static method *domain_sizes_minus_one*, which sums the sizes of the domains of all variables and decrements the value by 1 for every variable in the problem. This number represents the number of domain values that have to be pruned away before every variable has a domain of size one, implying the problem has been solved.

  A minor tweak to this heuristic is that $+\infty$ is returned when a variable has a domain of size 0. This is because a variable with a domain of size 0 is a sign of a state in which a constraint has been validated. It is not possible to generate successors from it. This puts the node at the end of the agenda, postponing its evaluation until we have exhausted all productive states.

- **Domain splitting heuristic**: The variable selected for having its domain guessed is the variable with the lowest number of remaining domain values (except if the number is one). The reasoning behind this is that this makes for the smallest number of remaining guesses: The probability of randomly picking the right domain value for a variable with n remaining values is $\frac{1}{n}$. It is clear that the smaller n is, the higher the probability that the right value is picked, even if the value is selected at random. If a wrong choice is made, there are fewer remaining choices.

  The reason why a variable with size 1 can not be chosen for splitting, is that this variable has already had its true value determined. Selecting it for a guess runs the algorithm into a corner. The reason why a variable with domain size 0 can be selected for splitting, is that a state with a variable with no remaining domain values will yield no successors when it is selected for successor generation. The only consequences of its evaluation is that the state is pruned from the agenda.

3 Subclasses for Nonograms

A nonogram CSP state is stored in an instance of the NonogramProblem class, which is a subclass of the CSPproblem class from module 2. This contains an initialization method, which takes a nonogram specification in a list format. The format is a list of lists, each inner list representing one line from the specification. The dimensions of the board are read from the first entry, and this is used to determine which lines to interpret as rows and columns.

For each subsequent entry in the specification, all possible bit vectors that can satisfy this entry's specification is generated. This is done by the static method *create_all_possible_combinations_from_line_spec*, which takes the line specification and the line's length as arguments. This utilizes a helper method for placing a segment in every possible way after a given starting position. After all bit vectors have been created, the set of these bit vectors are assigned to this column or row's entry in the NonogramProblem's *domains* dictionary.

After domains for all rows and columns have been created, a Constraint for every cell on the board is initialized and added to the list of constraints. This is not affected by any part of the specification other than the dimensions of the board.

Other than what is described above, no further subclassing has been needed.

4 Design Decisions for Performance

The tweak of the search heuristic for returning $+\infty$ when a domain has size 0 (mentioned in Section 2) is the only decision that has been made for the sake of performance. This makes the A* implementation pop fewer nodes from the agenda. Only scenario 0 of the nonograms require any successors to be generated, but in this case, the number of nodes expanded is reduced from 2 to 1 before a solution is found.