IT3105 Module 3

Iver Jordal

Representations

Domains and variables

The representation I chose is mostly based on the second suggested method in the assignment (2.2 Using an Aggregate Representation)[0]. Each domain is either a row or a column, and each value in the domain is a possible combination of values in that row or column. The segment numbers as specified by the nonogram problem determine what a possible combination for a row or column is.

So I asked myself: How do I find all the possible combinations for a row or column? First I thought generating all possible combinations of zeroes and ones in an array with the size of the row or column. Then I figured that it would be too slow for even medium-sized nonograms. Then I figured that I could be smarter about it and reduce the domains in two steps. Let me explain:

For each row and for each column I first compute the domains for start index possibilities. Allow me to use the same example as on page 3 in the assignment[0]:

Size: 10 Segments: 2, 1, 3	Start(first segment): { 0, 1, 2 } Start(second segment): { 3, 4, 5 } Start(third segment): { 5, 6, 7 }

Then I use the cartesian product of those start index domains to find possible combinations of start indexes. Obviously, not all combinations that come directly from the cartesian product are valid as per the nonogram spec. Therefore, I apply some filtering so that only valid combinations are inserted to the domain of the row or the column. It seems very efficient, at least for the sample problems.

After the filtering, the domain of possible combinations ends up looking like this:

```
{
    (1, 1, 0, 1, 0, 1, 1, 1, 0, 0),
    (1, 1, 0, 1, 0, 0, 1, 1, 1, 0),
    (1, 1, 0, 1, 0, 0, 0, 1, 1, 1),
    (1, 1, 0, 0, 1, 0, 1, 1, 1, 0),
    (1, 1, 0, 0, 1, 0, 0, 1, 1, 1),
    (1, 1, 0, 0, 0, 1, 0, 1, 1, 1),
    (0, 1, 1, 0, 1, 0, 1, 1, 1, 0),
    (0, 1, 1, 0, 1, 0, 0, 1, 1, 1),
    (0, 1, 1, 0, 0, 1, 0, 1, 1, 1),
    (0, 0, 1, 1, 0, 1, 0, 1, 1, 1)
}
```

There exists one variable for each row and one variable for each column. They are named “r” + index and “c” + index respectively. For example, for a nonogram with dimensions equal to 3x3, there would be six variables: r0, r1, r2, c0, c1 and c2.

Constraints

Based on figure 3 in the assignment[0], I figured (no pun intended!) that there should be one constraint for each cell. For example, in a 3x3 nonogram, there are be 9 cells, and thus 9 constraints. The concept of a constraint is that the row and the column for that cell should agree on what the value of the cell should be (either 0 or 1). For example “the sixth in the first row must be equal to the first element of the sixth column”. For that example, the NgConstraintNetwork class would generate a constraint expression that looks like this:

“r0[5] == c5[0]”

This kind of python-compatible expression is easily interpreted by the Constraint class, which compiles them to swift functions.

Heuristics

A* heuristic function

Let's begin with the heuristic function in the A* sense. First, I tried to implement an admissible heuristic function. It was based on tanh, so it would never overestimate. It was horribly slow, and almost equivalent to breadth first search. After that I assumed that, because a CSP problem is not primarily about finding the shortest path, it is okay to implement a heuristic that is not admissible. My heuristic function is based on a suggestion in the module 2 assignment. It simply sums the sizes of the domains in the search state. I tried subtracting one for each domain, but that led to worse execution times, so I dropped that idea. I also tried to multiply by 0.7 and 2. Multiplying by 0.7 worsened the execution time, and the factor of 2 didn't help much either, so I dropped those ideas as well. I've implemented one exception, though. If a search state has at least one empty domain, that means the search state is inconsistent, i.e. it's a dead end and should not be popped. In that case the heuristic function assigns a very large number to the node.

Choice of variable

Next, there's the heuristic for assuming values when generating children. Generating a child node is costly, because one has to call rerun on it immediately after it is created. Therefore, a bad heuristic for generating children can lead to a lot of child nodes, which can worsen the execution time significantly. The basic concept that I've used is: Select the variable with the smallest domain, where the domain size is greater than 1. Let me explain why this is a good idea: When one randomly assumes a value in a small domain, the probability for that assumption being correct/viable is going to be higher compared to an assumption in a larger domain.

Subclasses

For this problem, I had to implement input parsing, graphics and a subclass of ConstraintNetwork. The main work was figuring out the representation and calculating the possible start indexes and viable value combinations. This is done in the NgConstraintNetwork, which is a subclass of ConstraintNetwork. When initializing NgConstraintNetwork with the parsed nonogram spec, it calculates the variables, domains and constraints, and initializes its superclass with them.

When it comes to the Node class I've used the general CspNode class, because the general implementation with heuristics as described above works well for this nonogram problem. In fact, most of the methods in the CspNode class are also used in module 2, where the problem is vertex coloring.

Making it perform well

- I've spent time choosing my data structures carefully. For example a domain is a set. That makes inserting and removing values quick: O(1). Using an inappropriate data structure would significantly worsen the execution time.
- I've used memoization techniques to return cached results instead of doing expensive computations for every call. This makes the program roughly 20% faster.
- The fact that the domains for the rows and columns are pretty well filtered from the beginning makes it easy for the GAC to reduce the domains pretty much down to one or a few values. Typically, when running the sample nonogram problems, a solution is found

without even making any assumptions. This is like a simple sudoku, where all values fall into place by logical deduction.

References

[0] http://www.idi.ntnu.no/emner/it3105/assignments/csp-astar-nonograms.pdf