Module 3
IT3105

Simon Borøy-Johnsen
MTDT

October 5, 2015

Representations

Variables

The variables in my solution are the rows and columns. The rows and columns are treated in the same way, there is no distinction between them other than their ids. Row ids are of the form *r[INDEX]*, while the columns' are of the form *c[INDEX]*. Zero-indexing is used, with the upper left corner being indexed *[0, 0]*.

Domains

The domains of the variables are lists containing all possible segment combinations. The combinations are represented as strings with the same length as there are cells in the object (row or column). The strings contain only zeros and ones, where a zero represents an empty cell and the ones represents filled cells.

For example if there is a column (index 0) of width 5 with segments [2, 1], the domain would be:

```
variables['c0'] = ['11010', '11001', '01101']
```

Constraints

The constraints are made up of all possible combinations of rows and columns. For every row there is a constraint connected to every column. A row and column share one cell, so checking for illegal values is simply done by iterating all cells in the row permutation, removing those that doesn't match the same cell in the column permutation.

Heuristics

There are two heuristics used solving this problem; the regular A* heuristic, and when generating new neighbour nodes.

- *A**: The heuristic used in the A* algorithm is pretty simple. It iterates over all the variables' domains and sums the number of values yet to be removed. For example, if there are two variables in the problem, with domains

  ```
  variables['r0'] = ['0', '1']
  variables['c0'] = ['0']
  ```

  the heuristic value would be 1. This heuristic is not admissible, but when solving nonograms it really doesn't matter for the end result. Only one solution is possible, but it will take longer time to find it.

- *Generating neighbours*: The heuristic used when choosing what variable to base the next assumption on is also pretty simple. It simply picks one of the variables with the smallest domain. This is accomplished using Python's *min* function with key

  ```
  lambda variable: len(variables[variable])
  ```

Subclasses

As mentioned in the section about representations, no sub classing is needed for the CSP part. The variables and constraints have to be generated, but this is done in a separate class, the class that reads the input and runs the algorithm.

To implement the A* part, the four problem specific node functions have to be implemented;

- *generate_neighbours*: When generating neighbours, an undetermined variable in the CSP is picked using the heuristic mentioned above. For all the possible values in the variable, a new CSP is run where the variable's domain has been reduced to only contain the current value. If reducing the domain to the given value does not cause any contradictions, the CSP is added as a neighbour.

- *generate_id*: The id of a node is generated by parsing the variable domains as a string. The variable ids are sorted, then a list containing all the domains are built. The list itself is fed into the built in Python function *str*;

  ```
  str([self.state.variables[variable]
  for variable in sorted(self.state.variables.keys())])
  ```

this results in very large ids, which might take a while to compare. I ignored this, because the time it takes to compare ids will still be far smaller than for other parts of the algorithm.

- *heuristic*: The heuristic function is described above.
- *is_solution*: To check whether a node is a goal is simply done by checking if all of its domains have been reduced to singleton values or not. This is done very quickly by checking if the node's heuristic value equals zero.

Design decisions

I believe the design decisions worth mentioning are already mentioned above.