Solving Puzzles with A*-GAC

Anders Sildnes, Andrej Leitner *students*

***Abstract*—This text answers assignment 3, 4: Combining Best-First Search and Constraint-Satisfaction to Solve Complex Puzzles. In this document, we explain representations, heuristics and design decisions we used to handle flow puzzels and nonograms tasks. The assignment is built on previous modul through several extensions.**

CSP solutions defines variables with domains. A valid solution is one where the domain of all variables has been reduced to the singleton domain. This occurs by making sure that a) no constraints are violated and b) all variables are iterated over, and been locked to a single value in their domain.

Generality of A*-GAC Solver

The solvers for each puzzle is implemented using the same source code as in the previous project. *Essentially*, we use the exact same code for out CNET[1]. The changes we have made are:

1) Added methods "addCons(...)" and "addLambda(...)", which can take either a lambda, or a string that is parsable as a lambda, and translate that into a constraint. Example: addCons([1,2], "A <B") adds a constraint for vertex instance 1 and 2, and maps them onto A, B respectively (the first index maps to the first capital letter in the string).
2) Special case handling for when the domains consist of *sets* rather than single numbers, as used in the previous assignment.

Apart from item 1 and 2, this assignement is solved by subclassing the class "Problem" and implementing its methods[2] and also a parser for the input files.

[1]ConstraintNETwork, explained in project 1

Modeling Numberlink

Both problems could be fully expressed as SAT, so we found both problems to be NP-hard. We chose to use this representation for the flow puzzle. While the benchmark of Michael Spivey

```
[http://spivey.oriel.ox.ac.uk
/corner/Programming_competition_results]
```

found that it can be faster to represent point in the graph by its incoming/outgoing direction of flow (NE, NW, NS, WE, WS ...), we chose SAT because of its simplicity and ease of implementing it in our model, and also with the knowledge that the input files were no bigger than 10×10, and that small differences in performance is not critical to our evaluation. Also, the SAT formulation meant that we had small domains, which is important when using our CNET because domains are copied quite often... So therefore, even if the winning "numberlink" from the benchmark[ibid] modeled his project using directions, it might be that it is inefficient in our model when using GAC and constraint propagation.

In our model, we represent the board as a 2D-grid of cells, each representing one vertex instance $v_{i,j}$ for position i, j in the cartesian plane. For each v, you can assign k colors, where k is given by the number of endpoints / 2. Each cell $v_{i,j}$ must be a part of the path [3] from an endpoint $e_{i,j}$ to another endpoint

[2]in "astar.py": triggerStart(), genNeighbour(), destructor(), updateStates()

[3]I will assume the reader is has a mutual understanding of "path"

$e_{i+\alpha,j+\beta}$. Therefore our constraints are modelled as follows:

$$\forall i,j : leastTwoOf(v_{i+1,j}, v_{i+1,j}, v_{i+1,j}, v_{i+1,j},) == v_{i,j} \quad (1)$$

$$\forall i,j : leastOneOf(v_{i+1,j}, v_{i+1,j}, v_{i+1,j}, v_{i+1,j},) == e_{i,j} \quad (2)$$

this can also be thought of as applying a 5-point stencil to the grid and validating each neighbour value. This constraint only applies to the entir LOOPS, deffered

A. Choice of next cell

Clearly, random is bad..

B. Choice of heurstic

We scanned through

Modelling Nonograms

While nonograms certainly also can be expressed as SAT, we found it easier to follow the model suggested to us in the assignment text. In this approach firstly from given input we need to calculate all possible linear patterns that fill the row and satisfy the segment specification. Same for columns. We do this with recursive function *generatePos* inspired by this project:

```
[https://github.com/coolbutuseless
/nonogram-solver]
```

Here from given rule r, we generate all possible sequences for row or column where "filled" cells are represented with positive value of mapped coordinates of the cell and "unfilled" conversely negative. We use the same representation of cells as in previous module - vertex instances with two coordinates in 2D-grid board, however mapped in single value in row or column pattern representation.

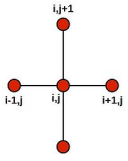

Fig. 1: Stencil operation

Here we are able to build the CNET. In this step we had to create a new subclass of CNET with overridden constructor. Just because we use variables with constant domain size in previous modules and this case we work with variable domain size. In our approach we use each row and each column as variable. Then all viable patterns for specific rows and columns are their domains. As the contraint there is only one function applied overall:

```
func = lambda A,B: len(A.intersection(B))
```

Here we can assume A variable for rows and B variable for columns. Firstly we pair each row with all columns and then we pass through these pairs with intersection checking. In particular, we check for the length of intersection between possible row and column pattern and accept only value 1 as valid solution.

C. Choice of heuristic

In this approach we can find heuristics in some places:

1) in A* we used the same $h-function$ as in previous graph coloring module where an index value to each node in the search tree is given determined by:

   ```
   f = depth + h(domains)
   ```

   In $h-function$ we count the size of all domains to each variable and then the state with lowest value is chosen.

2) New vertex is generated in specific state as first chosen from vertexes where the domain size where not reduced to 1. Subsequently for all values in its domain assumptions are made domain revised and new states generated.

D. Structure and other

In the nonogram solver module we use new subclass of our general *Problem* again.

As usually, we needed to implement basic abstract methods for triggering initial queue in A*, generating neighbours, update method for painting, final destructor and also unique constructer as we mentioned. These are quite simple functions, therefore we do not consider it necessary to discuss them in more details.

In order to create fast solution we decided to use sets instead of lists for preserving and comparing all values in domains of our variables. We also use indexing of values in variables and we try eliminate all redundant copies. This decision forced us to create modified version of *revise* method.

PICTURES?

I. MIN-CONFLICTS?

Using MIN-conflicts never occured..