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

Considerations

While the A*-GAC solver has the benefit of begin general, we found it difficult with copying domaijns

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,

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

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 $e_{i+\alpha,j+\beta}$. 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. This is to use each row or column as a VI, and have their domains as all possible orderings given their rule r. Also, the promise of a "reduction in running time of 60%" was alluring. The disadvantage of this approach, namely the large domain-spaces, were not deemed to be critical to our application under the premise that we would always solve relatively small boards.

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

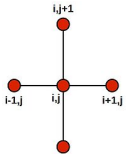

Fig. 1: Stencil operation

Each cell in the cartesian-grid used in a nonogram can be represented by a single number. We did this numbering in the fashion seen in Figure 2 Now the possible domain for each row, column is represented by the numbers in that row or column. For example, given the rule "3", the rows in Figure 2 would have their domain as [1,2,3] or [4,5,6]. Blank cells are representing by using the negative of the coordinate in a cell. Thus, for the second column in Figure 2, given rule "1", would have two possible domain values: [-2,4] and [2,-4].

Now, all constraints can be expressed as matchings between two VI's, a row and a column. Our lambda, in pseudocode looks as follows:

```
len(A.intersection(B)) == 1
```

where A,B can be the VI for a row or column (the function is symmetric).

An intersection will only occur if the coordinate in each domain has the same sign (positive or negative value). Since each row and column only share one cell, this function is able to filter domains successfully.

The disadvantage to our lambda is that the constraints will assess large spaces, while the constraint only depend on one value in each domain. Also, the A*-GAC requires much copying states and domains, so much excess overhead is included. However, the program still runs in under 20 seconds for all the given test inputs, so we have not spent time optimizing the function[4] -

[4] using properties such as that you can determine exactly which cell you need to compare using only the indexes i, j

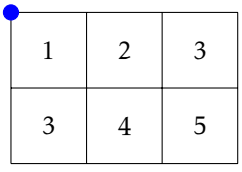

Fig. 2: Our representation of coordinates in a grid

C. Choice of heuristic

After finishing our solver, it was easy to see that few states are ever created and expanded. This is due to the coupling of constraints: each row is paired with every column, and verca visa. In general this means that reducing one domain forces the GAC to filter the entire space of VI's multiple times. With a small problem space for the A*, it became clear that the choise of heuristic is not critical. Therefore we used the same function as in ...

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..
graph labeling