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 module using simple extensions and subclassing.**

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 between VI 1 and 2, where the indexes are mapped into the string in order (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 abstract methods[2] . Also, you will need to parse the input for each problem. This is done in our "main.py". As a reminder, the abstract methods of the "Problem" class defines:

Drawing - how each state should be painted. The method receives both the previous state and the current one. By comparing the states, you can avoid repainting the whole GUI, or use a lazy updating scheme. In this assignement, with either few states or nodes, we chose the latter.

Destructor - what to do when the A* queue is exhausted. In this assigment we simply print the total number of nodes used and expanded in the A*-search, and do a last repainting of the GUI.

Generating successor states - defines how, given a state, to generate all its successors.

Generating initial state - creates the initial queue for A*.

Considerations

The biggest problem we found related to using A*-GAC was that we could not implement a solution that relied on states. For example, apart from running time, it should not matter if a constraint c_1 is run before constraint c_2 - they should both prune their domain as much as needed. Therefore we could not implement thoughts such as "if check(A) then othercheck (B)".

Both problems could be fully expressed as SAT, so we found both problems to be NP-hard. Therefore we knew that to find exact solutions, we would have non-polyomial problem-sizes.

[1]ConstraintNETwork, explained in project 1

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

It was therefore tempting to use local search, but we felt that the inputs given were small enough that an iterative solution would do well.

Modelling Nonograms

While nonograms certainly 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. 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.

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 1. 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 1 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 1, 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. In pseudocode, our generated lambda is 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 shared coordinate in each domain has the same sign

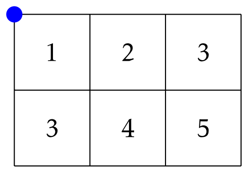


Fig. 1: Our representation of a cartesian grid

(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[3].

A. 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 vice versa. 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*-algorithm, it became clear that the choise of heuristic is not critical. We used "min-domain", which works OK.

B. Choice of next VI

We tried experimenting with the choice of each VI. We initially postulated that choosing a center row or column would yield would be faster. However, it is important to note that to the constraints only binary relationships matter, and pruning one row will likely invoke constraints from all other rows. Therefore the starting VI is chosen randomly.

For successor states we found that selecting the VI with the smallest domain reduced the running time. The idea is that this VI first of all has fewer constraints that needs to be checked. Also, assuming our heuristic has chosen a state that is closer to the goal, the VI with the smallest domain will likely force a feasible solution faster.

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

Modeling Flow Puzzles

We chose to use SAT 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 [4] from an endpoint $e_{i,j}$ to another endpoint $e_{i+\alpha,j+\beta}$. Therefore our constraints are modelled as follows:

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

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

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

C. Choice of next cell

Clearly, random is bad..

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

D. Choice of heurstic

We scanned through

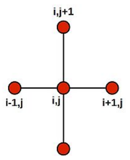
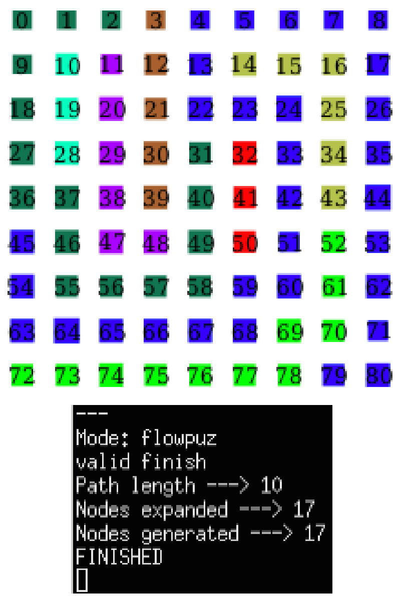
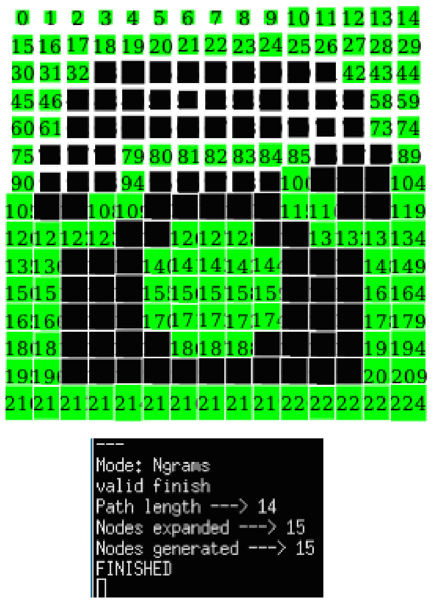


Fig. 2: Stencil operation



```
---
Mode: flowpuz
valid finish
Path length ---> 10
Nodes expanded ---> 17
Nodes generated ---> 17
FINISHED
```

Fig. 3: Flowpuzzle example



```
---
Mode: Ngrams
valid finish
Path length ---> 14
Nodes expanded ---> 15
Nodes generated ---> 15
FINISHED
```

Fig. 4: Nonogram example