

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 puzzles 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 our CNET¹. 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 assignment is solved by subclassing the class “Problem” and

implementing its abstract methods². 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 assignment, with either few states or nodes, we chose the latter.

Destructor - what to do when the A* queue is exhausted. In this assignment 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

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-polynomial problem-sizes. 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

¹ConstraintNetwork, explained in project 1

²in “astar.py”: triggerStart(), genNeighbour(), destructor(), updateStates()

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 (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 over-

head 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 function³.

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 choice 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.

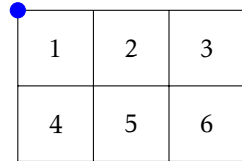
For successor states we found that selecting the VI with the smallest domain reduced the running time. The idea is that this VI first of all has fewer constraints that needs to be checked. Also, assuming our heuristic has chosen a state that is closer to the goal, the VI with the smallest domain will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

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]

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



| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

Fig. 1: Our representation of a cartesian grid

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 the intended 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.

The way we modeled our problem was not strictly True or False values. Instead, each cell had a domain value $d_{i,j}$ that indicated its color. Thus, the scope of all possible domains for a given cell was k , where k is the number of different colors. This can still be used to express similarities in relationships, however, with a reduced set of constraints, which we are about to explain.

We defined the neighbour of a cell to be the similar to a 5-point stencil, except on the borders, where the out-of-bounds values are omitted. The first constraints for a cells are easy: for regular cells:

$$\text{at least two neighbours share color} \quad (1)$$

and for endpoints:

$$\text{at least one neighbour share color} \quad (2)$$

These constraints can be modeled as a nested set of conditional branches, or, to keep the program adherant to numbers only, asserting that the sum, for each neighbour: $\text{abs}(\text{DIFF}(\text{COLOR}_{adj})) - \text{abs}(\text{DIFF}(\text{COLOR}_{adj}) - 1) \leq \alpha$ (given that there are n variables, we can assert how many have the same value). are sufficient for simple problems, but they miss in the case shown in Figure 3.

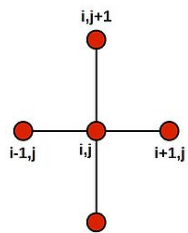


Fig. 2: Stencil operation

Before we present how we solved this problem, we must review our choice of nodes:

Choice of next cell

When iterating over the board, randomly choosing a cell and iterating over its domain puts you at risk for finding a good partial solution that needs to be backtracked many many times. An example would be to make a guess of a path in two separate corners that can never meet. Before this is realized, you would have generated many states with possible permutations in between.

If one is always propagating a path, backtracking begins as soon as *one or several* paths are deemed invalid. To always propagate a path, you can choose a starting node in a corner and iterate over the board. Now, noticing, that a path that starts in a corner propels in *both* directions, you will likely determine if the path is valid if you iterate over the diagonal of the board. Therefore, when choosing the next cell, we use the diagonal of the board. In Figure 1 this would be [1,2,4,3,5,6].

C. Additional constraints

Now, using the fact that we know we are always propagating or starting a path when we make assumption from A*-states, we can always assume that any cell that we expand *has flow incoming*. Since we start in the north west, this means that we can assume that when considering a cell, it has flow coming in from the north or west. If not, this means we are making a new path, and neither NORTH or

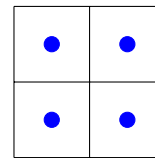


Fig. 3: Failure for constraint 1) and 2): all cells are satisfied, but there is a loss of flow

WEST is set, so must “toggle” both east and south.

These constraints are again, working, but they also fail to see some scenarios: when a SOUTH or EAST neighbour is an endpoint, flow can go as shown in ?? . We dealt with this by asserting that if neighbour $(i, j+1)$, or $(i+1, j)$ is an endpoint, you are allowed to have a node with 3 adjacent neighbours. This is sufficient to solve all the example input problems and some other test cases we designed, but will, if you are unfortunate, make it possible to have loops in the solution.

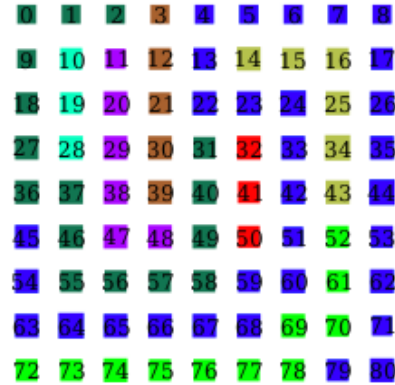
D. Choice of heuristic

While there are several heuristics available, we again feel that “min-conflicts” won away for being the most efficient.

One can early on determine if a solution for the flow puzzle is inefficient. By counting the min cost from each node to each other (using for example Manhattan distance), one can note that if there are not enough available cells to fill in the remaining, unfinished paths for each end-node, then the solution is infeasible. We did, however, feel that the cost of computing this number, when using our diagonal iterations, did not pay off. Trusting our approach, we kept the heuristic function simple.

IN THE END

This was a fun project to work on. We were surprised, and annoyed by how difficult path uniqueness was to settle with SAT. In the end, we would like to post some screenshots that shows the program in use.

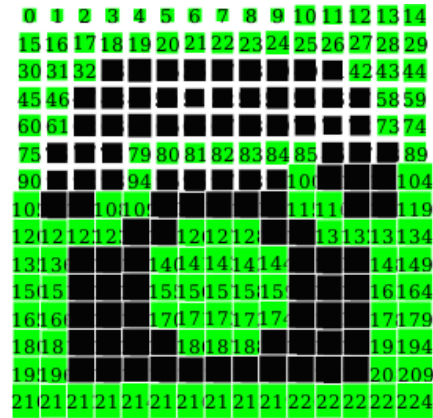


```

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

```

Fig. 4: Flowpuzzle example



```

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

```

Fig. 5: Nonogram example