AI Programming (IT-3105) Project Modules #3-4:
Combining Best-First Search and Constraint-Satisfaction to Solve Complex Puzzles.

Purpose: Learn to frame tasks as constraint-satisfaction problems (CSPs) and then solve them automatically using the A*-GAC program (created for an earlier assignment).

1 Introduction

A wide variety of tasks **can** be posed as constraint-satisfaction problems (CSPs) - not all, but a good many. When a problem does seem amenable to formalization in terms of the CSP's three primary components (variables, domains and constraints), then the full power of tools such as GAC and MIN-CONFLICTS can be unleashed upon them. Your only job is to convert the task formulation into those components such that when the CSP solver returns a set of values for the variables, your system can translate those variable assignments into a solution to the original task.

This project is predominantly a representation task: you must determine how to represent your problems as CSPs. The field of AI is full of representation problems; just about every AI project begins with one. By putting proper emphasis on the critical essence of a problem, a good representations can make it much easier to solve, while a bad representation often produces search spaces that are unnecessarily large and/or biased against the discovery of good solutions.

A good deal of your grade on this assignment will be based on the ability of your system to solve moderately difficult puzzles, and this could prove challenging without proper attention to representational issues. You will solve two types of puzzles using your A*-GAC system: flow puzzles and nonograms. Both can be solved as CSPs, although it is not necessarily the optimal approach to either problem.

For each puzzle type, you will be given a few hints and pointers, but nothing more. Your job is to build two different modules, both of which work with your A*-GAC system, to solve flow and nonogram puzzles.

2 Flow Puzzles

Flow puzzles require the formation of multiple paths (a.k.a. *flows*) on a square grid (of variable size), with the initial conditions consisting of nothing more than the two endpoints of each flow.

Figure 1 shows one such grid along with the specification of its initial configuration. The solution illustrates the complexity of solutions, even for small grids.

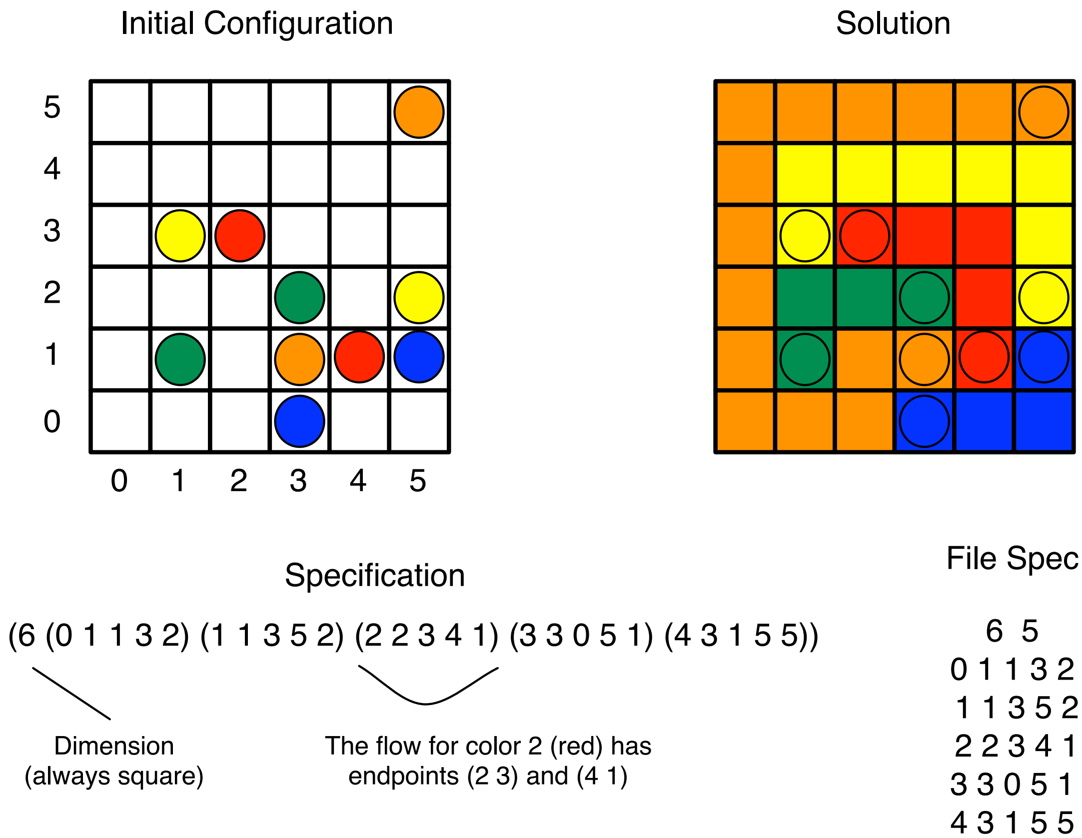

Figure 1: A simple 6 x 6 Flow Puzzle. (Upper left) The scenario involves 5 different pairs of endpoints (colored circles), each of which must be connected by a path/flow. (Upper right) The solution – note that it covers every cell in the grid. (Lower left) The scenario specification consists of the dimension along with K quintuples, one for each flow, indicating the color/index of the flow followed by the Cartesian coordinates of its two endpoints. (Lower right) The listing of a flow scenario in a file, where the first line gives the size of the grid and the number of colored flows, while quintuples appear on the succeeding lines.

A solution to a flow puzzle must satisfying the following conditions:

1. It connects each pair of endpoints using paths that combine vertical and/or horizontal (but no diagonal) cell-cell links.
2. It has no intersecting paths/flows.
3. EVERY cell of the grid is filled with exactly one color: every cell participates in one and only one flow.

Solutions cannot involve loops, but they may involve many neighboring cells. Though the directionality of a flow has no meaningful significance, there are situations when flows make tight 180-degree turns, making the flow pattern difficult to visualize without the help of a few arrows that show the sequence of cells in the flow. Also, it is possible for a flow to make a wider turn but eventually come back and flow parallel with itself. Figure 2 provides one example of a 180-degree turn. These flows, though somewhat rare in typical puzzles, should make it clear that **the following rule is overly restrictive** as a defining condition for a flow-puzzle solution:

> All cells, except endpoints, in a flow solution should have exactly two neighboring cells with the same color. Endpoints should have exactly one.

Here, the neighborhood refers to only 4 cells: those to the immediate north, south, east and west of a given cell. Again note that this is an overly-restrictive condition. By including it as a constraint type in a CSP-solver, you will rule out those legal solutions that include tight 180-degree turns and flows that turn around and run parallel to themselves.

The proper constraint is more flexible:

> All cells, except endpoints, in a flow solution should have **two or more** neighboring cells with the same color. Endpoints should have **one or more**.

It is important to note that:

1. A great many flow-puzzle solutions satisfy the first, overly-restrictive, constraint. When applied to every cell in the grid, this one constraint type is sufficient to produce solutions to many flow puzzles.
2. However, there are some puzzles whose solutions require tight 180-degree turns, and these cannot be found unless the second, more flexible, constraint is in effect.
3. Unfortunately, this second constraint is typically not enough to solve flow puzzles on its own. It weakens the restrictions on cells to the point that many illegal solutions are found (and viewed as legal by the system). Additional constraints are therefore needed to re-restrict flows in order to insure the legality of solutions.
4. One way to re-restrict flows is to associate more than just a color with each cell. In addition, variables and constraints that reflect the legal ways of connecting one cell to another as part of a flow can be defined. For example, it should not be possible for a cell to receive flow input from more than one cell.

2.1 Solving Flow Puzzles with A*-GAC

The task is to employ your A*-GAC system to solve flow puzzles. As input, your system will receive a flow scenario, specified in the format of Figure 1. It must then convert that scenario into a large set of variables and constraints, which formally define the CSP that A*-GAC will then solve.

When all variables have values and all constraints are satisfied, your system should graphically display the solution, as in Figure 1 (upper right), along with the following counts:

1. The total number of search nodes generated
2. The total number of search nodes expanded
3. The total number of search nodes on the path from the root to the solution state.

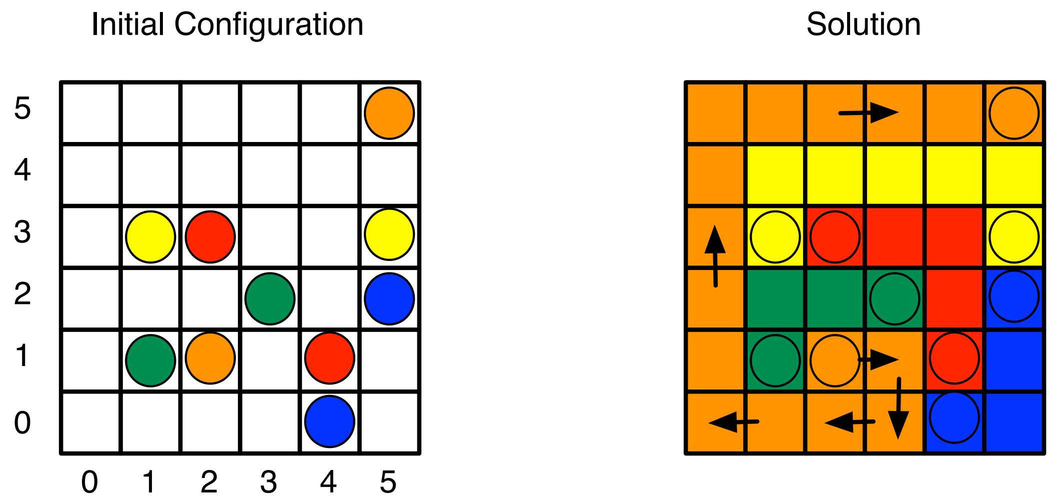

Figure 2: A flow puzzle solution that involves a 180-degree bend (in the orange flow), thus making the precise flow pattern less straightforward to visualize without the help of a few guiding arrows.

Your GUI does NOT need to show arrows or connectors between cells, but it must clearly show the COLOR of the flow that passes through each cell. You can assign colors to color indices in any way, as long as each index has a unique color.

In addition, your system must display the partial flow solution (again using your GUI) associated with each search node as it is popped from the agenda. This will give a rough indicator of search progress.

2.2 Flow Puzzle Scenarios

Table 1 provides several puzzles that your A*-GAC should be able to solve. Though puzzles larger than size 10 certainly exist, you will not be required to solve them for this project. However, if you feel like giving your system a (more) serious challenge, feel free to download one of the free flow-puzzle apps and dig in!

The scenarios in Table 1 are also provided as files on the course web page, where the scenario file's index corresponds to the index in the left column of the table. It is strongly advised that you write a file reader for these scenarios, since that will save time during the demo session.

3 Nonograms

Nonograms (a.k.a. *Griddlers*) are also puzzles involving colored tiles on a 2-dimensional grid. In this case, the patterns typically represent common images, such as that of a cat, car, bird or building. The puzzle solver receives only 1-dimensional information about the *segments* (i.e., continuous blocks of filled cells) in each row and column. From that, he, she or it must discover the complete 2-d image.

Figure 3 displays both the image and the clues of a nonogram that vaguely resembles an open-air structure such as a picnic shelter. On the left of the image, each short list of numbers denotes the sizes, in order, of

Table 1: Several flow-puzzle scenarios for testing A*-GAC. All boards are square, so a size of K indicates a K x K board. Each quintuple consists of the color index followed by the Cartesian coordinates of the flow's two endpoints, as shown in Figure 1

Flow Scenarios		
Index	Size	Flow quintuples
0	6	(0 2 2 4 5) (1 0 1 0 5) (2 0 0 1 5) (3 2 0 5 5) (4 2 3 2 5) (5 2 1 4 4)
1	6	(0 3 0 3 2) (1 1 0 2 4) (2 0 0 2 0) (3 4 4 5 0) (4 4 5 5 1)
2	8	(0 0 7 3 7)(1 1 1 2 5)(2 2 6 3 2)(3 2 4 3 1)(4 3 6 4 3) (5 4 7 7 1)(6 4 6 6 6)(7 4 4 6 1)(8 5 4 7 0)
3	9	(0 0 5 7 8) (1 0 8 7 5) (2 1 1 1 3) (3 2 0 4 3) (4 2 1 3 5) (5 3 0 3 4) (6 5 1 7 4) (7 5 3 5 5)
4	5	(0 0 4 2 3) (1 1 0 1 3) (2 1 1 4 2) (3 3 1 4 3)
5	10	(0 0 0 1 8)(1 0 1 1 3)(2 1 4 6 7)(3 2 1 7 0)(4 2 6 5 8)(5 3 4 8 3) (6 4 2 4 6)(7 4 5 5 7)(8 5 2 8 0)(9 5 3 6 6)(10 6 8 8 6)
6	10	(0 0 1 6 9) (1 1 1 2 8) (2 1 2 3 4) (3 1 4 4 4) (4 1 8 3 7) (5 2 2 4 7) (6 2 6 5 9) (7 3 6 5 4) (8 7 9 8 0)

the segments in that row. For example, the bottom row has the simple clue, 6, indicating that the entire row is filled. The next row up has the specification (1,1), indicating that there are two single-celled segments. Row 4 (0-based indexing) has three segments, of lengths 1, 2 and 1.

Similarly, along the top of the image, the segment counts for each column appear. For example, in row 1 (0-based indexing), there are segments of lengths 1 and 4.

In all nonogram puzzles, the row and column segments must be separated by at least one unfilled cell. Also, in most nonograms, the ordering of the segment sizes is important. For example, (1,2,1) entails that there is a 1-group followed by a 2-group followed by another 1-group, where *followed by* means *separated by one or more unfilled cells*. This ordering condition holds in all the nonograms of this project.

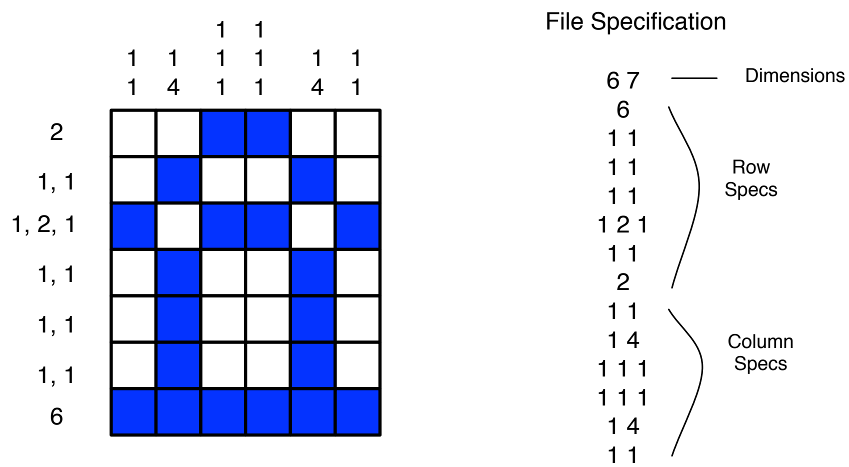

Figure 3: (Left) A simple nonogram illustrating both the row and column clues and the target image. (Right) The file format for nonograms used in this project.

3.1 Nonograms as CSPs

Nonograms admit many possible representations as CSPs, though framing all of the important logic as constraints can be difficult. What follows are several suggestions for possible CSP formulations.

As shown in Figure 4, the segment sizes for a row (or column) provide the basis for variables, domains and constraints. In this row of size 10, there are three segments (A, B and C) of sizes 2, 1 and 3, respectively – in that order going left to right. Adding in the mandatory blank(s) between each segment, the distance (in terms of the total number of tiles) from the start of A to the end of B is a minimum of 8 (2 + 1 + 1 + 1 + 3). Hence, A must begin in column 2 or earlier, and C must end in column 7 or later (using zero-based indexing).

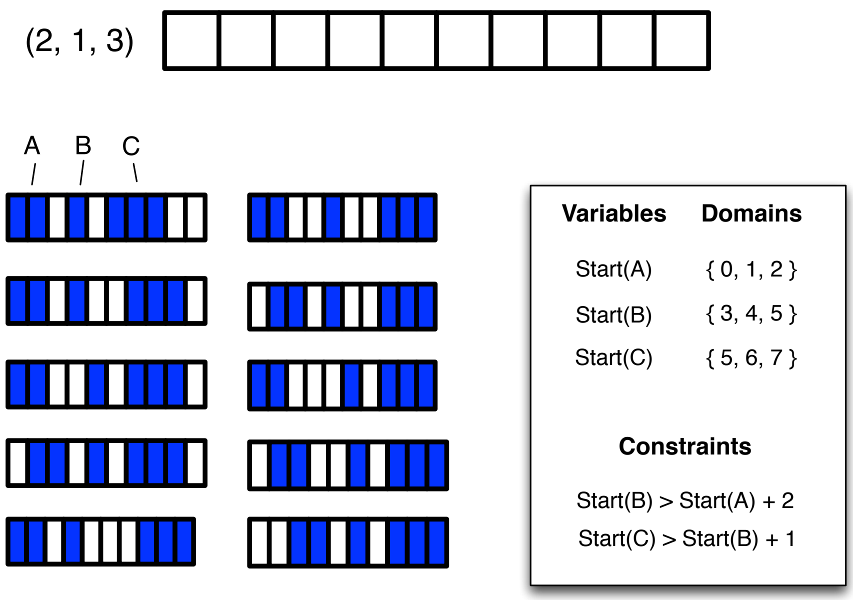

Figure 4: (Top) A 10-cell row with 3 segment clues. (Bottom Left) 10 possible ways, sanctioned by the clues, of positioning the 3 segments in the row. (Bottom Right) The variables, domains and constraints generated for these particular clues.

This type of analysis hints at one potential CSP formulation:

- Variables are the start cells of row (and column) segments.
- Domains are the feasible values for those variables, with some simple arithmetic providing very helpful initial restrictions: very few variables will have domains containing all integers from 0 to row/column size minus 1.
- Constraints embody relationships between adjacent segments, capturing the fact that segment S+1 cannot start until at least L+1 spots after segment S begins, where L is the length of segment S.

By creating variables, domains and constraints for each set of row and column segments, you can capture a good deal of the vital information in a nonogram in the explicit components of a CSP. Unfortunately, this does not capture all of the essential information. The interactions between row segments and column segments are much more difficult to represent as formal, explicit constraints, though accounting for these interactions implicitly requires only minor additional work.

The advantage of explicit constraints is that they allow the system to mechanically reduce variable domains using the standard CSP tools, such as GAC. The disadvantage is that some tasks require a very large number of constraints to incorporate all of the critical information. For example, to capture the interactions between row and column segments would seem to require a wide variety of constraints, where each row (column) segment could have complex relationships between collections of column (row) segments; simple one-to-one constraints (as in the K-queens problem) between all pairings of row and column segments would not suffice, since it is hard to predict, ahead of time, which column segments a given row segment will interact with.

For example, the CSP components in Figure 4 do not capture the fact that **no other segments** should appear in the row. This exclusive nature of the segments is non-trivial to represent as an explicit constraint. However, it is trivial to simply check each row (column) to make sure that no more than N cells are filled, where N is the sum of the lengths of the segments in that row (column). [1]

One compromise is a hybrid solution in which some constraints are handled explicitly via the normal CSP machinery, while others are handled implicitly by other code. This other code typically cannot easily modify variable domains as GAC does, but it can detect violations that A*-GAC can use to prune search nodes.

For instance, anytime a row-checking routine detects too many filled cells in a partial solution, it can simply flag the search state as a dead-end and remove it from further consideration. This is a valid filter, since any A*-GAC search state represents a partial solution in which some of the variables have values. In the nonogram representation discussed so far, the only variables are the start points of the row and column segments. When a segment-start has a value, this implies that L cells in the row/column will be filled in (where L is the length of that segment). So all of the assigned variables in a state produce filled cells in a grid. If too many row or column cells are filled, they will not be un-filled by child states of the current state. Filling is a monotonic process in this type of search. The only way to unfill cells is to abandon the current state and jump to another state.

So using the CSP representation shown in Figure 4 supports a hybrid solution in which formal CSP algorithms handle the domain pruning, while implicit constraints take care of dead-end state pruning. Together, this combination can solve many nonograms, but, unfortunately, it cannot take advantage of the full power of GAC, and some nonograms can take quite a long time for the computer to solve.

Another alternative involves using ONLY the row variables and constraints in GAC, but then checking for violations of column constraints using other code. Here again, any violations of the column constraints lead to immediate pruning of the search node. This also qualifies as a hybrid solution, since some of the constraints are handled explicitly by GAC, while others are manifest in auxiliary code for checking column violations.

3.2 Getting it all into a CSP

Fortunately, there is at least one possibility for incorporating everything into a formal CSP framework, and allowing the full power of A*-GAC to do most of our work. What follows is a (substantial) hint as to how to do so.

Returning to Figure 4, note that, when given a set of segment sizes, we can easily calculate all possible linear

[1] This sum-of-filled-row-cells is inappropriate as a CSP variable, since we already know exactly what value it should have: N. Hence, we cannot incorporate it directly into the CSP representation.

patterns that fill the row and satisfy the segment specification. Similarly, all viable patterns for any column can also be calculated. This extra little bit of pre-processing has huge benefits.

If each row and column is a variable in a CSP, then these sets of viable patterns constitute their domains. Now all we need are some constraints.

Figure 5 hints at the origins of some useful constraints. Every pairing of a row and a column will have one cell in common, and the row and column patterns must agree on the value of that cell: filled or unfilled. Figure 5 shows how the possible patterns of the row variable are filtered against those of the column variable. This leads to a 60 % reduction in the row patterns due to the simple fact that in ALL of the 6 column patterns, the shared cell (C) is filled. So any row pattern with an unfilled cell C can be removed. Note that the opposite filtering (of the column patterns based on the row patterns) would fail to constrict the column patterns, since at least one row pattern fills the shared cell; and thus, all column patterns are consistent with at least one row pattern.

If you want to implement this approach, figuring out the remaining details is your job. Doing so will allow A*-GAC to, quite easily, solve a wide variety of nonograms, and often with only a couple dozen nodes in the A* search space. In contrast, the approaches discussed earlier tend to require much greater computational resources.

As long as you use A*-GAC as the basis for solving nonograms, you are free to use any of the representations mentioned above, or another of your own invention.

3.3 Solving Nonograms with A*-GAC

Your program must accept nonogram scenario specifications from a file (using the format shown in Figure 3). From that information, it should create the appropriate variables, domains and constraints and then employ A*-GAC to discover the target images. The discovered images must be visualized in a simple 2-dimensional grid, as in Figure 3. You do NOT need to display the segment sizes (beside rows and above columns) in this GUI; the image itself is sufficient.

In addition, you must display the following (standard) counts with each solution:

1. The total number of search nodes generated
2. The total number of search nodes expanded
3. The total number of search nodes on the path from the root to the solution state.

In addition, your system must display the partially-solved nonogram (again using your GUI) associated with each search node as it is popped from the agenda. This will give a rough indicator of search progress. Depending upon your choice of representation, many of these images may represent dead-ends (that are simply popped from the agenda and not expanded).

3.4 Nonogram Scenarios

A set of nonogram files will be provided with this project, on the course web page. Your A*-GAC system should be able to handle all of them. None are larger than 20 x 20 grids, and nothing larger than 20 x 20

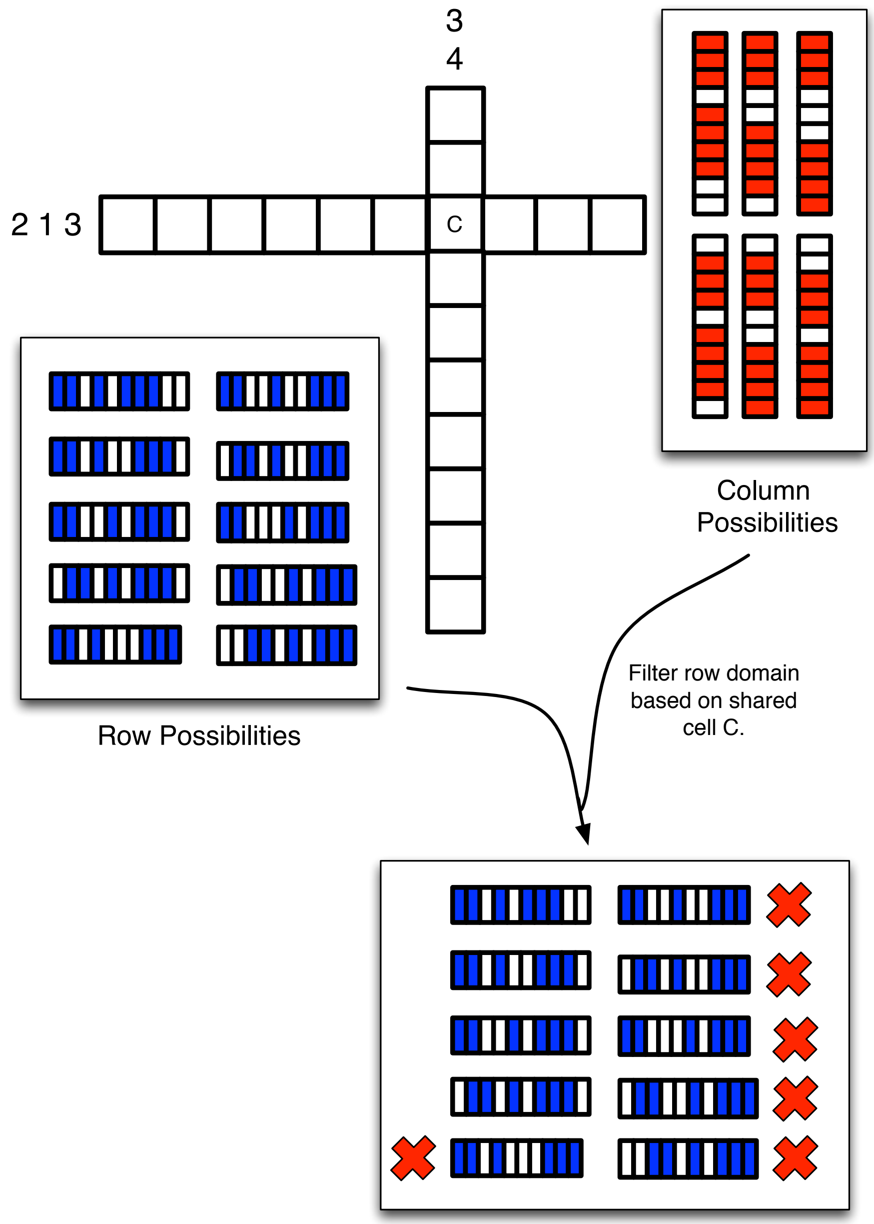

Figure 5: When each variable is an entire row or column, and the domains are all possible fill patterns of the row or column (that are compatible with the segment clues), then the shared cell between a row and column provides the basis for domain filtering. Red crosses denote row patterns that are incompatible with ALL column patterns and can therefore be removed. Note that in all column patterns, the 3rd cell from the top (i.e., the shared cell, C) is filled. The use of blue and red to indicate filled cells only aids in visualization; both colors indicate the same thing: a filled cell.

will be given to you at the demo session.

4 Deliverables

1. A 4-page report that:
   - Clearly documents the two different representations (variables, domains and constraints) that you devised for solving flow puzzles and nonograms.
   - Explains the heuristics used for each problem. Note that heuristics appear in at least two places in A*-GSP: a) in A*'s traditional h function, and b) in the choice of a variable on which to base the next assumption. Both (and others, if relevant) should be mentioned in the report.
   - Briefly overviews the primary subclasses and methods needed to handle these two tasks using A*-GAC.
   - Mentions any other design decisions that are, in your mind, critical to getting the system to perform well.

   During the demo session, questions addressing topics similar to those above may be asked. The report must not exceed 4 pages. (**10 points**)
2. You must demonstrate that your system solves THREE of the flow puzzles provided with this project, along with TWO others that will be given to you at the demo session. (**15 points**)
3. You must demonstrate that your system solves THREE of the nonograms provided with this project, along with TWO others that will be given to you at the demo session. (**15 points**)

At the demo session, all test scenarios will be chosen by the instructor. All of the scenarios provided to you ahead of time should be easily available to your system during the demo. For example, you should be able to enter an index and have any of these cases loaded automatically.

A zip file containing your report along with the commented code must be uploaded to It's Learning prior to the demo session in which this project module is evaluated. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 40 total points for these two modules are 40 of the 100 points that are available for the entire semester.

The due date for these two modules is the 2nd demo session.