AI Programming (IT-3105) Project Module #3:
Combining Best-First Search and Constraint-Satisfaction to Solve Nonograms.

Purpose: Learn to represent puzzles 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 a special type of puzzle known as a *nonogram* using your A*-GAC system. This document gives you a few hints as to how to do so; you will have to figure out the rest.

2 Nonograms

Nonograms (a.k.a. *Griddlers*) are puzzles involving colored tiles on a 2-dimensional grid. In this assignment, the patterns typically represent common images, such as that of a cat, car, bird or building – whereas some nonogram sites use images that appear random. 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 1 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 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.

As an important format detail, the nonogram files for this project contain rows of numbers (segment counts) that correspond to the image in the exact same manner as the correspondence shown in Figure 1. The first line houses the number of columns and rows, and then the segment counts for rows begin, with the **bottom** row first. The column counts begin with the **leftmost** column. In short, the counts begin at the bottom left corner of the image and move clockwise around the image. Any other interpretation of these files can have fatal consequences: a reversed set of row or column specifications may yield an unsolvable puzzle.

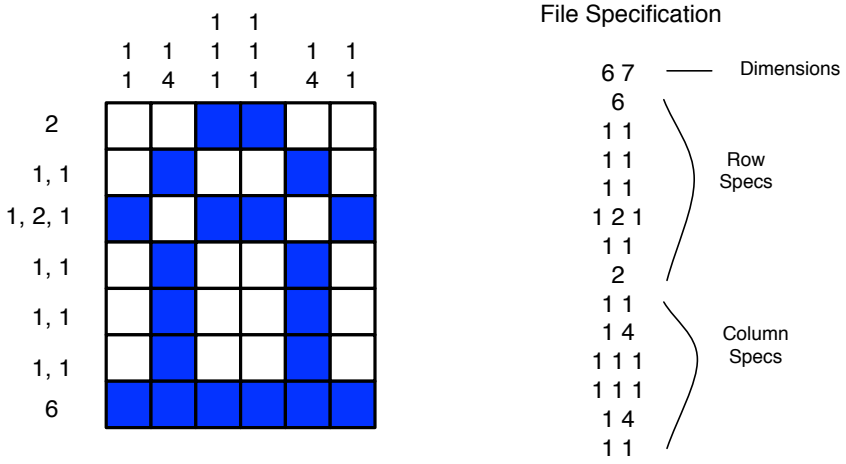

Figure 1: (Left) A simple nonogram illustrating both the row and column clues and the target image. (Right) The file format for nonograms used in this project.

2.1 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 2, 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).

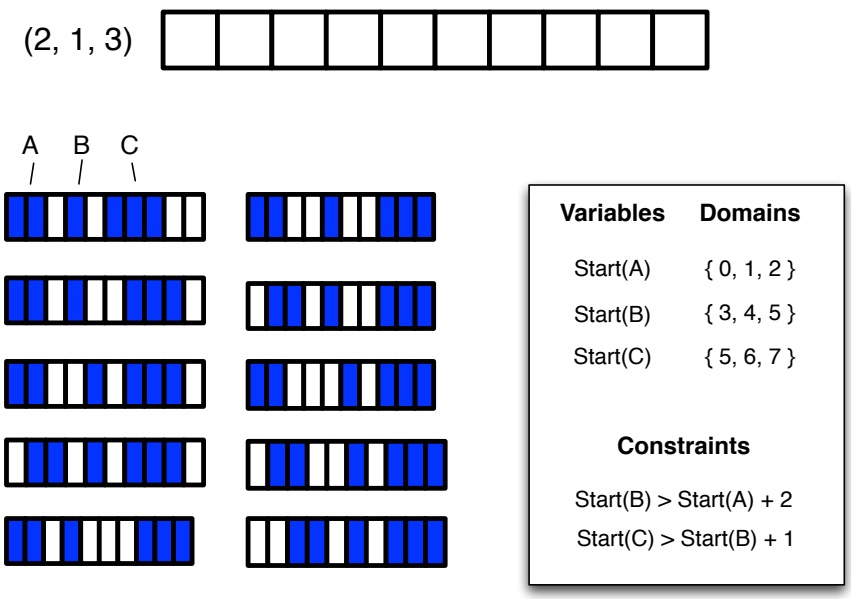

Figure 2: (Top) A 10-cell row with 3 segment clues. (Bottom Left) 10 possible ways, sanctioned by the clues, of positioning the 3 segments in the row. (Bottom Right) The variables, domains and constraints generated for these particular clues.

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 also require constraints. There are probably several ways to do this. One is to include one constraint per cell that embodies the following relationship:

> If any row segment intersects this cell, then a column segment must also intersect it; and, if any column segment intersects it, then a row segment must also intersect it.

A proper implementation of this constraint, one per cell, combined with the basic row and column constraints (discussed above and shown in Figure 2), will allow your system to fully exploit A*-GAC to mechanically reduce variable domains and solve many complex nonograms.

2.2 Using an Aggregate Representation

Another approach to solving nonograms involves variables and constraints that capture a higher level of representation. Instead of looking at segments and how they interact, this one considers whole rows and columns and their relationships. What follows is a substantial hint as to how this representation could be used to solve nonograms using A*-GAC.

Returning to Figure 2, note that, when given a set of segment sizes, we can easily calculate all possible linear 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 3 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 3 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.

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.

2.3 Solving Nonograms with A*-GAC

Your program must accept nonogram scenario specifications from a file (using the format shown in Figure 1). 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 1. 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.

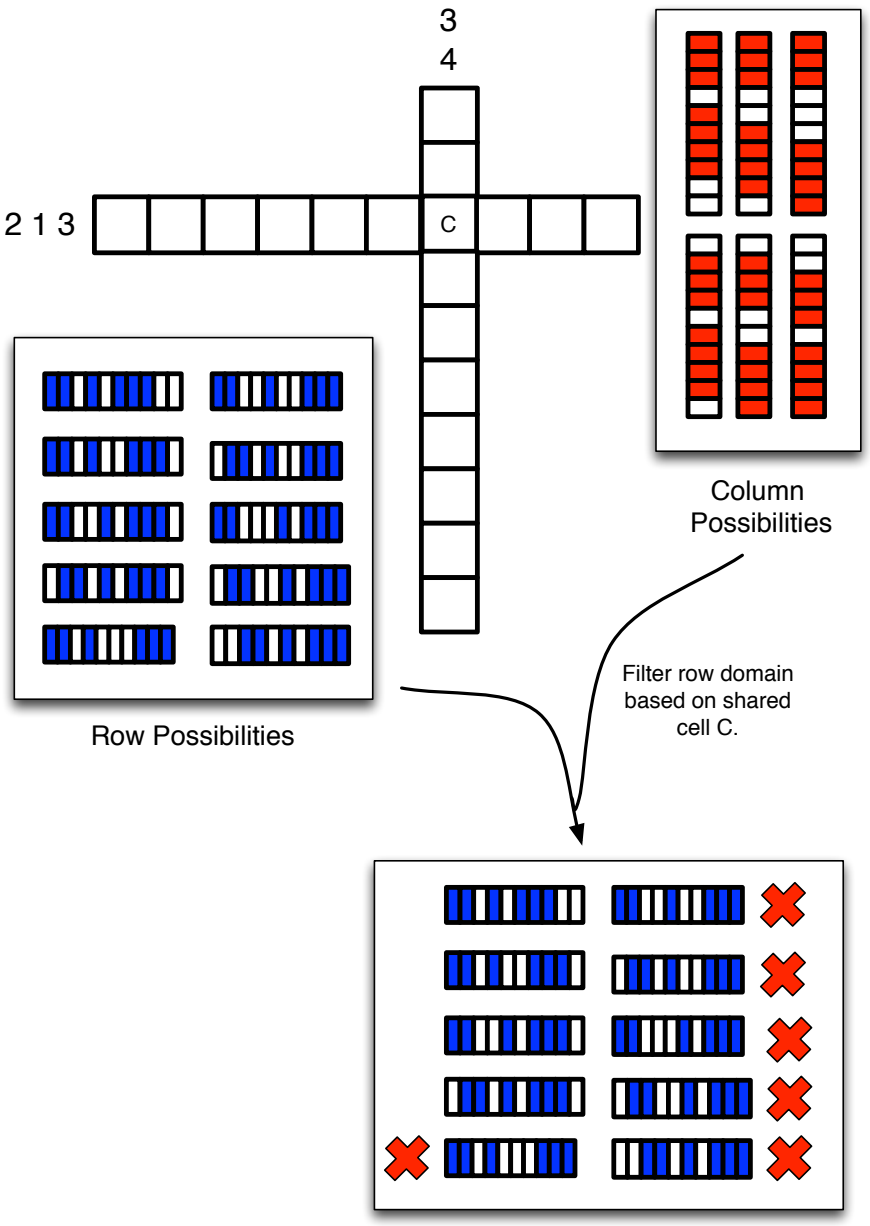

Figure 3: When each variable is an entire row or column, and the domains are all possible fill patterns of the row or column (that are compatible with the segment clues), then the shared cell between a row and column provides the basis for domain filtering. Red crosses denote row patterns that are incompatible 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.

Depending upon your choice of representation, many of these images may represent dead-ends (that are simply popped from the agenda and not expanded).

2.4 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 will be given to you at the demo session.

3 Deliverables

1. A 3-page report (**5 points**) that:
   - Clearly documents the representations (variables, domains and constraints) that you devised for solving nonograms.
   - Explains the heuristics used for this 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 specialize your general-purpose A*-GAC system to handle nonograms.
   - 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, along with any others that the instructor considers relevant. The report must NOT exceed 3 pages.
2. You must demonstrate that your system solves several of the nonograms provided with this project, along with a few 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. Failure to streamline the scenario-entry process can lead to loss of points.

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 20 total points for this module are 20 of the 100 points that are available for the entire semester.

The due date for this module is the first demo session.

4 Appendix: Hybrid Approaches to Nonograms

The two representational approaches discussed in this document manage to formally represent enough constraints so that A*-GAC can solve a wide variety of nonograms. This is normally an advantage. However, for very large problems, the number of variables and constraints, and/or the sizes of variable domains can tax computational resources. The version of GAC described in this and related documents, is not a highly optimized process, so, depending upon your available computing resources, a large nonogram could bring your system to its knees. After all, the number of constraints scales quadratically with the basic dimensions of a puzzle, i.e., the number of rows or columns. Worse than that, the complexity of GAC is O(cd^{k+1}), where c is the number of constraints, d is the domain size of a variable, and k is the number of variables in a constraint.

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 (safely) modify variable domains as GAC does, but it can detect violations that A*-GAC can use to prune search nodes.

For example, assume that you only implement row and column constraints, but not cross-constraints that relate rows to columns. In that case, unwanted cells can easily get filled in a row or column without violating any of the formal constraints. To mitigate this problem, a hybrid approach could include implicit constraints in the form of routines that explicitly check for overfilled rows and columns. Then, 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 segment-based representation discussed above, 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 2 supports a hybrid solution in which formal CSP algorithms handle the domain pruning, while implicit constraints take care of dead-end state pruning. 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.

Unless you write an extremely inefficient A*-GAC or have a very old computer, you should not have any major trouble solving most of the nonograms in this project assignment by using one of the two main approaches described above. You should not need to resort to a hybrid solution, many of which are not that effective anyway.