# AI Programming (IT-3105): Combining Constraint-Satisfaction Problem-Solving with Best-First Search

August 4, 2017

## 1 Introduction

As the basis of one or more homework modules, you will implement one of the traditional CSP algorithms: General Arc Consistency (GAC), which you will then integrate into the best-first search code used for a previous project module. Together, these provide a system, A*-GAC for solving CSPs in which assumptions (i.e., guesses) are required to reason forward to a viable solution: pure deductive inference alone is insufficient. This document provides a detailed recipe for producing such a hybrid system.

Consider one of the classic CSPs, Sudoku. The variables for Sudoku are simply the cells of the grid, and each has a domain consisting of the integers 1 to 9. The constraints are the standard *rules of the puzzle*: no two cells in the same row, column or 3x3 mini-grid can have the same value.

For easy Sudoku's, a long sequence of relatively simple deductions enables us to gradually and confidently fill in all of the cells, with no real uncertainty as to the validity of a given choice (of value for cell). These Sudokus can be solved by very standard CSP algorithms.

However, more difficult Sudokus require either very complex reasoning or assumptions/guesses (as to the value of a cell) followed by deductions. If those deductions lead to a contradiction, then we go back and retract the assumption (and maybe guess a different value for that same cell). If no contradiction arises, then the guessed value may stand and become part of the final solution. Either way, the nature of the problem requires us to either make very complex deductions or to make guesses and be willing to retract them.

In this assignment, you will build a CSP solver that can handle this combination of simple deductions and assumptions. And though you will not solve Sudokus for this assignment, your system should be general enough to support such a task, with only minimal extra code.

## 2 Constraint Satisfaction Problems (CSPs)

A CSP consists of 3 main components: variables, their domains, and constraints. Domains are simply lists of valid values for the variables. The goal of a CSP is to find one (possibly more) assignments of values to variables such that all of the constraints are satisfied.

For example, consider the following CSP:

$X \in [0,1,2,3]$, $Y \in [0,1,2,3,4,5]$, $Z \in [4,5,6,7]$

C1: $X > Y$

C2: $X + Y > Z$

Variables X, Y and Z have the domains shown beside them, and the two constraints are C1 and C2. Looking at this situation, we can see that X cannot be 0; since there is no value in Y's domain that would satisfy C1 if X were 0. Similarly, Y cannot be 3, 4, nor 5, since none of those values have a corresponding value in X's domain that could allow C1 to be true. For example, if Y is 4, then there is no value in domain(X) that is greater than 4.

Given these reductions in the domains of X and Y, we now have $X \in [1,2,3]$ and $Y \in [0,1,2]$. From this, we can infer that Z cannot be 5, 6, nor 7 due to constraint C2. Thus, Z must be 4. Then, in order for $X + Y$ to exceed Z (thus satisfying C2) and for X and Y to satisfy C1, the only alternative combination is: $X = 3$, $Y = 2$. Thus, the solution to this CSP is [3,2,4].

Note that the logic that we followed to solve this was largely based on a process of elimination, whereby we gradually removed values from the domains of variables until there were very few remaining options. This is the same approach used by computer algorithms for CSP solving: they reduce domains until each variable has only one possible value. The combinations of those possible values then comprise a solution. In some cases, it is not possible to reduce every domain to a single value, so the CSP solver has to try combinations of the variable values and check whether they satisfy all constraints. But either way, the essence of logical inference in CSP solving is the elimination of values from domains.

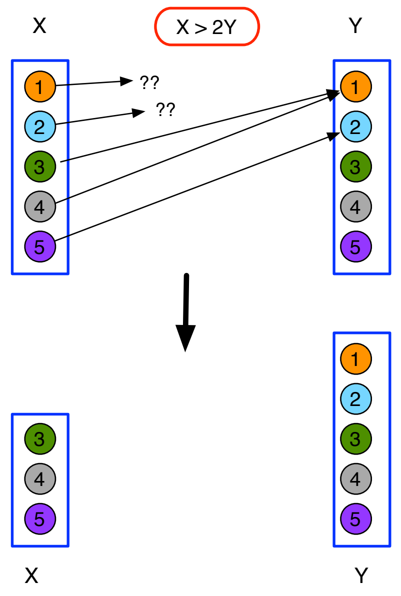

Figure 1: Illustration of the basic REVISE algorithm (pg. 209, Russell and Norvig, 3rd edition). Boxes with numbered circles denote domains of X and Y both before (above) and after (below) filtering.

In most cases, filtering algorithms involve a focal variable, X, a constraint, C, and all other variables involved in C. For example, in Figure 2, the focal variable is X, the constraint is $X > 2Y$, and the other variable is

Y. This figure illustrates the basic REVISE algorithm from our AI textbook. Each call to REVISE has the potential to eliminate options from the domain of ONLY the focal variable, so in this example, only X. The domain of Y can be filtered by a similar call to REVISE, but this time with Y as the focal variable.

The criterion for elimination is simple: if an option for X would make it impossible to satisfy the constraint, no matter which value from domain(Y) were chosen, then that option is removed from X's domain. So in Figure 2, it is obvious that values 1 and 2 need to be removed from domain(X), since neither have a corresponding value in Y that would allow the constraint to be true.

Slightly more formally, we can express the basic result of a call to REVISE as the following:

> Retain all x $\in$ Domain(X) where $\exists$ y $\in$ Domain(Y) such that (x,y) satisfies the constraint. Remove all others.

In the terminology of our AI textbook, constraints involving 2 variables are called *arc constraints*, and the REVISE algorithm performs *arc-consistency filtering*. However, other sources use a more general definition of arcs and arc consistency, such that any constraint involving any number of variables can be handled by a generalized version of REVISE, which performs *general arc-consistencye filtering*. We will refer to this general REVISE algorithm as REVISE*.

Figure 2 shows the essence of REVISE*. Now, instead of a single non-focal variable, there can be many; and a domain value, x, is removed if and only if there does not exist a combination (Q) of values for the non-focal variables such that, together, x and Q satisfy the constraint. To express this more formally:

- Assume constraint C consists of variable X plus variables $Y_1, Y_2 ... Y_n$, and assume that REVISE*(X,C) is called, meaning that X is the focal variable to be tested against the other variables in constraint C.
- Retain all and only those x $\in$ Domain(X) where: $\exists$ $(y_0, y_1, ..y_n)$ with $y_i$ $\in$ Domain($Y_i$) $\forall i$ and $(x, y_0, y_1, ..y_n)$ satisfies C.

Thus, REVISE* enables the filtering of variable domains across a wide variety of constraints. For many types of CSPs, it can handle all necessary domain-filtering chores.

## 3 The General Arc-Consistency Algorithm (GAC)

GAC employs a series of calls to REVISE* in order to solve CSPs. Any call to REVISE* that actually modifies the domain of the focal variable, X, instructs GAC to perform several additional calls involving other focal variables, namely all those that are associated with X via one or more shared constraints.

It is convenient to divide GAC into three processes: **initialization, domain-filtering**, and **rerun**. Each involves the main GAC data structure: a queue of requests to perform REVISE*. These requests are denoted as TODO-REVISE* pairs, with each pair consisting of a focal variable and a constraint.

The essence of each process is described below.

**Initialization**

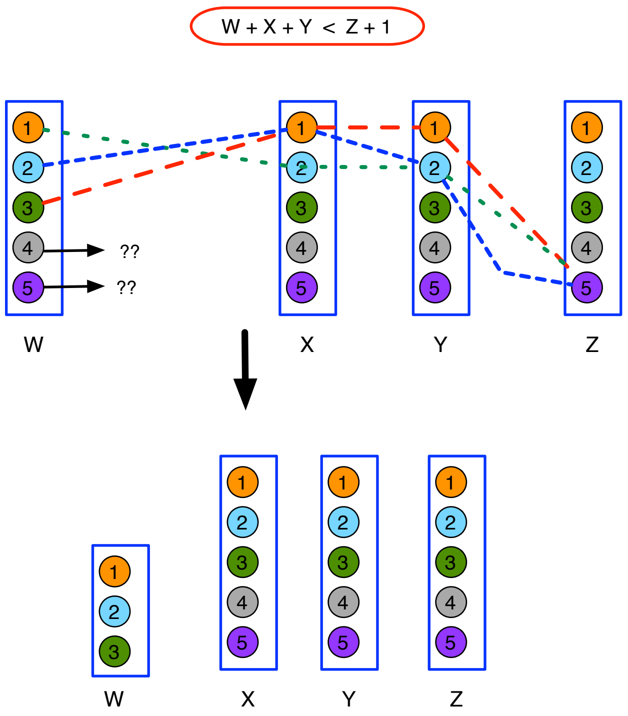

Figure 2: A simple example of general arc-consistency filtering (using REVISE*), wherein the domain of W is reduced because its 2 largest values do not have corresponding combinations of X, Y and Z values that can satisfy the constraint: W + X + Y < Z + 1. For the values 1, 2, and 3 in domain(W), the satisfying combinations of X, Y and Z are shown using dashed lines.

- QUEUE = {TODO-REVISE*($X_{i,j}, C_i$) : $\forall i, j$ } where $X_{i,j}$ = the jth variable in constraint $C_i$.

The queue is loaded up with REVISE* requests, one request for each pair of variable and constraint where the variable actually appears in the constraint. In the domain-filtering loop, these requests are popped from the queue, the calls to REVISE* are made, and more TODO-REVISE* pairs may be added.

**The Domain-Filtering Loop**

While QUEUE $\neq \emptyset$ do:

- TODO-REVISE*(X*,$C_i$) = pop(QUEUE)
- call REVISE*(X*,$C_i$)
- If domain(X*) gets reduced by the call, then:
  - Push TODO-REVISE*($X_{k,m}, C_k$) onto QUEUE for all $C_k$ ($k \neq i$) in which X* appears, and all $X_{k,m} \neq$ X*.

Domain filtering continues until the queue is empty, meaning that, given the current state of the system (as defined by the collective domains of all the variables), no more calls to REVISE* can produce further domain reductions. At this point, a standard constraint-filtering algorithm would have to halt. If all domains were not reduced to a single value, then more advanced techniques would be needed to find variable combinations that solved the CSP.

However, a simpler approach is to use an incremental search procedure, such as A*, to attempt different possibilities of values for variables and then explore the consequences (by re-running the domain-filtering loop). Each such attempt constitutes an assumption (or guess), and each forces more REVISE* requests onto the queue.

The Rerun module is thus a warm reboot of the domain-filtering loop, but as opposed to initialization, rerunning does not load the queue with all variables and all constraints. Instead, it only uses those constraints involving the variable, X*, whose value was most recently assumed/guessed.

**Rerun**(Given that the assumption X* = x was just made)

- Push TODO-REVISE*($X_{k,m}, C_k$) onto QUEUE for all $C_k$ in which X* appears, and all $X_{k,m} \neq X*$.
- Call Domain Filtering Loop

When making assumptions about variables, wherein a variable X* is assumed to have the value x, it is normally convenient to reduce the domain of X* to the singleton set {x}. This allows the GAC and REVISE* algorithms to easy handle all variables, whether they have an assumed (or deduced) value or have domains of size larger than one. [1]

[1] A deduced or inferred value is one that *had to be* in that the domain was reduced to a singleton set via one or more calls to REVISE*.

## 4 Combining GAC with A*

As indicated above, GAC can be fortified with best-first search so that whenever GAC can no longer use the process of elimination to reduce any of the domains, it can make assumptions of values for variables and continue running. Whereas forced domain reductions constitute *deductions*, assumptions are not forced. The search algorithm tries to make them in an intelligent manner, but it is guesswork, not a straightforward logical inference of a value that *has to be*. We enlist incremental search to help manage these assumptions and the alternate CSP states that they produce.

Figure 3 portrays the basic combination of GAC and incremental search. Each node in the search tree represents a state of the CSP, and each state is defined by the domains of each variable. These domains become more reduced as search moves deeper into the tree.

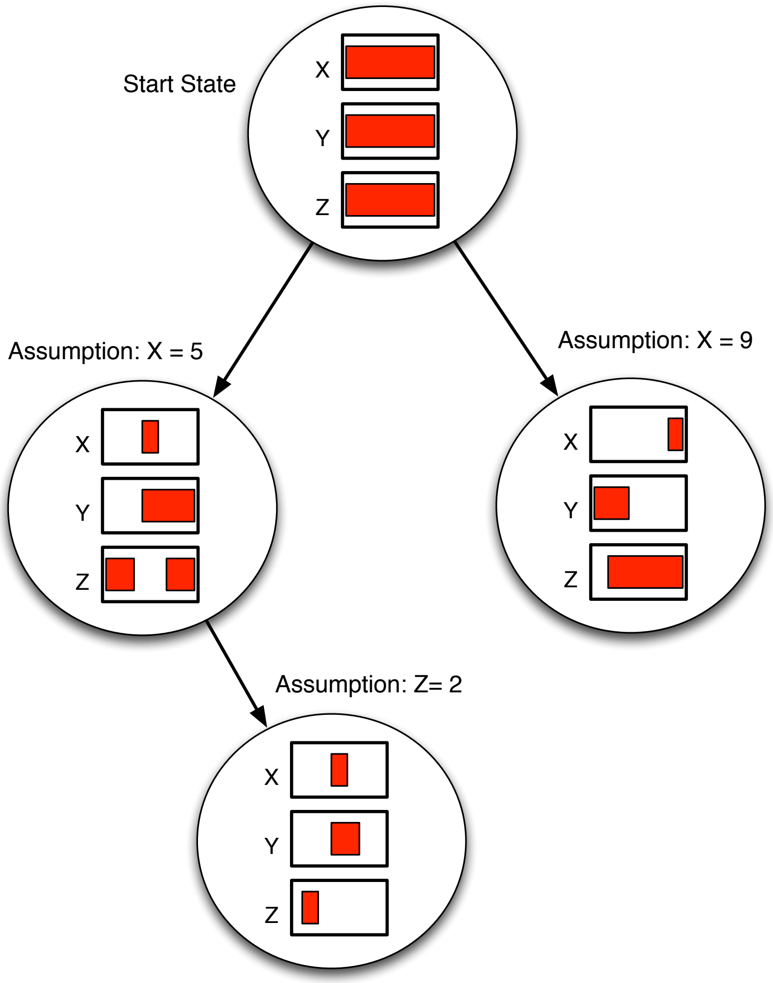

Figure 3: Abstract illustration of GAC allied with incremental search. Each node denotes a state of the CSP, with the boxes in each node representing variable domains. Red markings in each box roughly indicate the contents of each domain, with the key points being that a) they vary between the different states, and b) they become more restricted with distance from the root node. Each assumption automatically reduces the corresponding domain to a singleton.

The combination of GAC and incremental search is an algorithm in which search constitutes the high-level process, while GAC runs to refine each search node. If best-first search (e.g., A*) is used, then GAC runs on each newly-generated state prior to the calculation of a node's h value, which will be based on the apparent proximity of the CSP state to an actual solution.

A simple heuristic involves calculating the size of each domain minus one, then summing all those values to produce a very rough estimate of the distance to the goal. Devising a good, and admissible, heuristic, is more of a challenge, since it is very hard to estimate the extent of domain reduction incurred by any run of the Domain-filtering loop.

The basic algorithm that integrates GAC and A* works as follows:

```
Generate the initial state, S0, in which each variable has its full domain.
Refine S0 by running GAC-Initialize and then GAC-Domain-Filtering-Loop
If S0 is neither a solution nor a contradictory state, then:
        Continue normal A* search (with S0 in the root node) by:
                - Popping search nodes from the agenda
                - Generating their successor states (by making assumptions)
                - Enforcing the assumption in each successor state by reducing
                        the domain of the assumed variable to a singleton set
                - Calling GAC-Rerun on each newly-generated state
                - Computing the f, g and h values for each new state,
                        where h is based on the state of the CSP
                        after the call to GAC-Rerun.
```

S0 is contradictory if any one of the variable domains gets reduced to the empty set. This implies that there is no solution to the problem. S0 is a solution if every variable has a singleton domain.

If contradictory states arise later in A* search (i.e., at a node other than the root node), that only indicates that the sequence of assumptions used to get to that state is invalid, not that the whole problem is unsatisfiable.

When generating successor states, the key point is that each successor is defined by an assumption of a value for a variable. The nature of the CSP determines how many possible assumptions could or should be made from any given state. When solving a problem such as the K-Queens Puzzle via incremental search, it is common to work down the board, one row at a time, with each row constituting a variable in the CSP (e.g. row 7 is where queen 7 will reside), and the list of columns being the domain.

Thus, the successors to a state would be defined by taking the upper-most row in which a queen has not yet been assigned and then generating a child node for each possible column in which that queen could be placed. The assumption associated with each child node would then be its corresponding column choice.

Another alternative is to take the row in which the fewest viable column options still remain: the variable with the smallest domain. Each of those remaining domain values would be the basis for an assumption.

In general, each child node should be based on ONE additional assumption (of a value for a variable) that will separate it from its parent node. The choice of the variable and value is up to the system designer. But in most cases, the expansion of one parent will involve the selection of ONE variable and the formation of a child node for EACH value in that variable's domain. And remember, that domain is the **version** of the domain in the parent node, which is often smaller than the full domain. A key aspect of implementing this A*-GAC combination involves keeping track of all these domain versions and the constraints that bind them.

## 4.1 Implementation Details of A*-GAC

Figure 4 summarizes the key elements of A*-GAC. The key observation is that each node in A* search consists of a state of the CSP. The main difference between any two states is the contents of their variables' domains.[2] Because these domains differ across the states, it is important to maintain separate domain copies in each state. The constraints do not vary [3], so complete copies of them are not necessary. However, constraints are important for maintaining relationships among variables, and since each state contains copies of variables and their domains, it is useful to have copies of some constraint components in each search state.

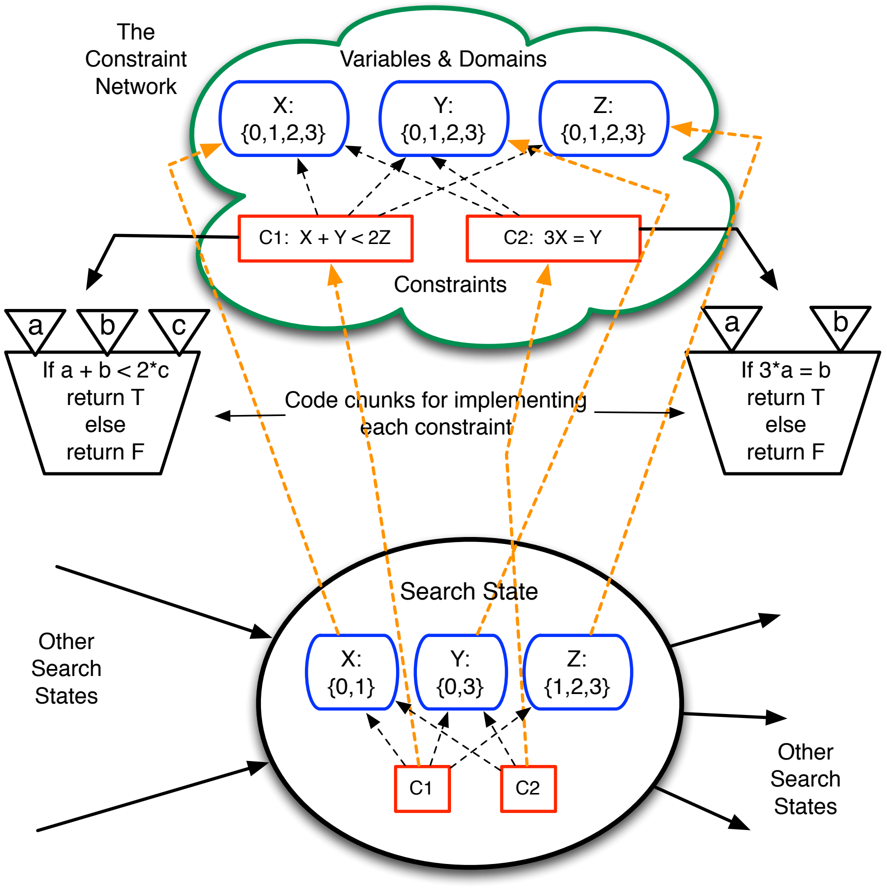

Figure 4: The overall structure of A*-GAC, with the original constraints and variables residing in a constraint network (CNET), while each search state houses instances of variables and constraints, with neither type of instance being a full copy.

At the top of Figure 4, the *constraint network* (CNET) consists of variables, their **full** domains, and detailed information about each constraint. This detailed information includes pointers to all of the variables in the constraint along with a pointer to a chunk of code that actually implements the constraint, i.e., that performs the test to check whether a set of values (one for each variable in the constraint) actually satisfies the constraint. It is this code that will be called from REVISE*.

The copies of constraints and variables will be referred to as *constraint instances* (CIs) and *variable instances*

[2]Although it is certainly possible for two states to have identical domains.

[3]They actually can vary across states in more advanced CSP algorithms, but we will not consider those in this project.

(VIs). Each VI will house the current domain of the variable, along with a pointer to that variable (in the constraint network). Each CI needs a pointer back up to its CNET constraint, along with pointers to each VI that participates in that constraint. The CI follows the pointer to its CNET constraint in order to access the code chunk.

When running the REVISE* algorithm, A*-GAC will be working with CIs and VIs. The domains of VIs become restricted via calls to REVISE*. When each VI in a state has a singleton domain, that constitutes a solution to the CSP. Similarly, if any VI has an empty domain, the state represents a dead-end, and no children should be generated from it.

For example, in the search state of Figure 4, the call REVISE*(X,C2) would use domain(X) = {0,1} and domain(Y) = {0,3}. REVISE* would attempt to filter values from domain(X), but, in this case, both 0 and 1 would remain in the domain.

### 4.1.1 Code Chunks

In order to produce a truly versatile CSP solver, you will need to accept constraint descriptions from the user, at runtime, and convert them into the working code chunks associated with each constraint object. In general, symbolic AI systems need this ability to accept *knowledge*, often in the form of logical rules or constraints, at run-time, and to then use that knowledge during an automated reasoning process. This process is typically hard-coded, but the knowledge is not. The knowledge normally arrives in some sort of standard (a.k.a. *canonical*) form, such as a string or a list of symbols.

Handling user-defined knowledge is facilitated by a programming language that supports the dynamic (i.e, during runtime) definition of functions. Traditional AI languages such as Lisp and Prolog support dynamic function creation, as does Python. In JAVA, C++ and other languages that use strong typing, things are more problematic. However, packages such as BeanShell (http://www.beanshell.org/home.html) for JAVA appear to do the job.

The general requirement is that when the user enters a constraint in canonical form, your system needs to convert that into a working function. In Python, this can be done as follows:

```
def makefunc(var_names, expression, envir=globals()):
    args = ",".join(var_names) # eg ['x','y','z'] => 'x,y,z'
    return eval("(lambda " + args + ": " + expression + ")" , envir)
```

The function *makefunc* takes a list of input variables (each a string), along with a string that uses those variables in a constraint. The third (and optional) argument is the environment in which the function will be created.[4]

First, makefunc simply converts the list of variable-name strings into a single string of all names, each separated by a comma. So ['x','y','z'] is converted into 'x,y,z'. This will be used as the argument list in the function declaration.

Python uses the lambda operator to create unnamed functions. You can use lambda both statically, in normal source code, or dynamically, in code that will produce new functions at runtime. The return line of makefunc puts together a string whose contents constitute an application of the lambda operator. Eval then evaluates that string, which effectively performs the lambda operation and produces the new function.

[4]To understand the third argument, you may need to read up on eval and scoping in a Python manual.

Below is a simple example of a call to makefunc, along with the returned function, which, for readability, is shown as a lambda expression. In reality, the command-line printed representation of the function reveals none of its underlying composition.

```
func = makefunc(['x','y','z'], ' x + y < 2*z')
⟶ (lambda x, y, z: x + y < 2 * z)
```

Now that we have a reference to the code chunk (func), we can easily apply it to any triple of numeric variables to determine if their values satisfy the constraint. For variables a, b and c, either of these calls will work:

```
func(a,b,c)
apply(func, [a,b,c])
```

If a, b and c are variable instances (VIs), then your code would first choose one value from each of their domains before applying the code chunk.

If your language of choice does not support dynamic function creation (at all), then you will probably need to write a limited interpreter that understands all of the legal operators that appear in the constraints of your CSP. When given a constraint such as x + y < z, the interpreter will need to walk through the constraint and perform operations such as fetching the values for x, y and z, adding the values of x and y, and comparing that sum to the value of z. Then, each call to REVISE* will invoke calls (usually many) to your interpreter. Though this is the least attractive alternative, it will insure that your CSP solver has the flexibility to accept user-defined constraints at runtime. However, before spending time writing an interpreter, fully investigate your chosen language for dynamic function possibilities.

## 5 The Vertex Coloring Problem (VC): A Simple Example

Vertex Coloring (VC), a.k.a. *graph coloring*, is a classic NP-Complete problem in which a color (or other parameter assignment) is made to each vertex in a graph such that no two vertices that share an edge have the same color (or parameter value). VC relates to the very practical *map-coloring problem*, in which any two adjacent countries should have different colors, thus visually enhancing their border, as shown in Figure 5.

Instances of VC include the value of a parameter, K, which indicates the size of the color set. The classic version involves K=4, but a general VC problem solver system should be able to handle a wide range of K values, from 3 up to 8 or 10.

As a CSP, Vertex Coloring is very straightforward: every vertex is a variable whose full domain is the numbers 1...K (or 0 ... K-1), and every edge in the graph is the basis for one constraint, which simply indicates that the colors of its two vertices cannot be equal.

The solutions to VC problems typically do not fall into place like an easy Sudoku. Rather, assumptions are required, sometimes several in a row, before a few color assignments are forced by the constraints. It is not unusual for A*-GAC to require a rather deep search tree to solve larger VC problems.

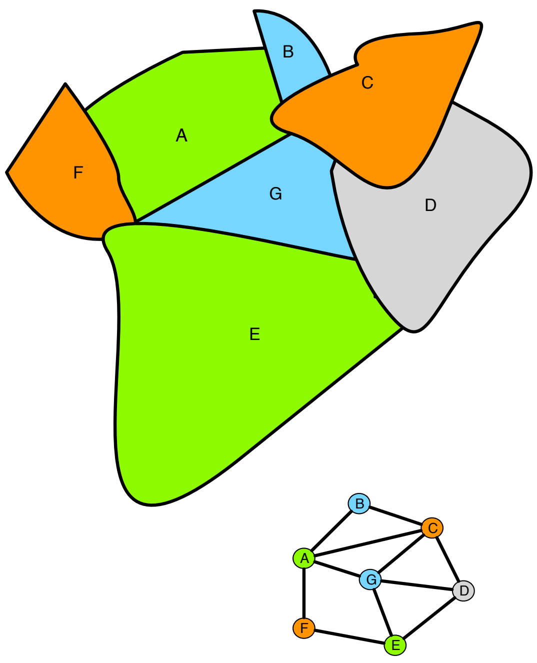

Figure 5: A simple illustration of a vertex-coloring problem (bottom) as an abstraction of map-coloring problem (top), in which any two countries that share a line border a) are represented by vertices with a connecting edge, and b) must have different colors.

Figure 6 displays the use of A*-GAC on a simple VC problem. Initially, each variable begins with a full 4-color domain. The call to GAC-Initialize results in all possible variable-constraint pairs being pushed onto the queue, but when each TODO-REVISE* request is run, no domain changes occur. Hence, all 12 calls produce the same state.

Next, an assumption is made that node C is colored red. This causes all variables related to C by a constraint to be paired with that constraint in a TODO-REVISE* request. Three such requests are placed on the queue, and REVISE* gets called three times, with each call resulting in red being removed from the domain of variables A, D and B.

Each of these domain reductions results in several more TODO-REVISE* requests, but none of these lead to any domain changes, so the queue eventually empties, leaving a CSP state in which node C is red and all other vertices can no longer choose red as a color option. Another assumption and further domain pruning is required to move closer to a solution.

Figure 7 shows several complex graphs that were colored using A*-GAC.

## 5.1 Code Chunks for VC

For this assignment, you will not need to accept user-specified constraints in a canonical form. That may become more important in later assignments in this course. However, you are encouraged to implement dynamic code chunks anyway, on this relatively simple CSP. Learning this technique will improve your general skills as an AI programmer.

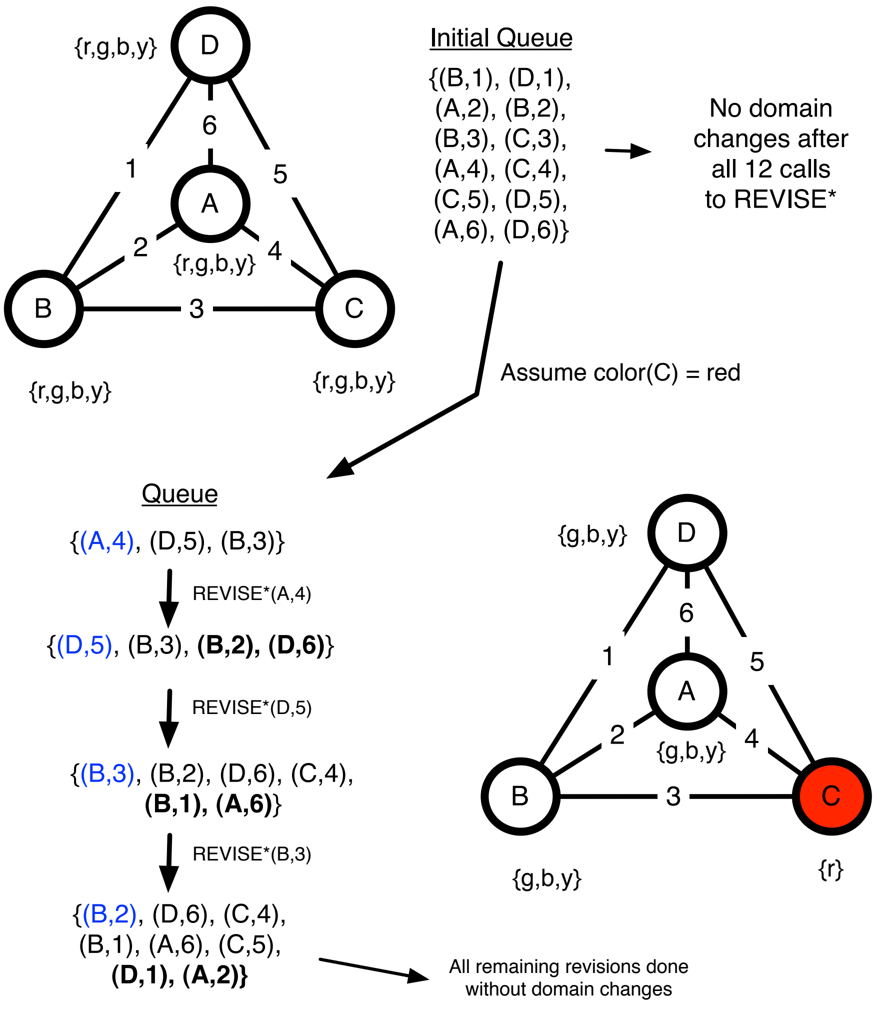
{r,g,b,y}
D
6
1
5
A
2
{r,g,b,y}
4
B
3
C
{r,g,b,y}
{r,g,b,y}
Initial Queue
{(B,1), (D,1), (A,2), (B,2), (B,3), (C,3), (A,4), (C,4), (C,5), (D,5), (A,6), (D,6)}
No domain changes after all 12 calls to REVISE*
Assume color(C) = red
Queue
{(A,4), (D,5), (B,3)}
REVISE*(A,4)
{(D,5), (B,3), (B,2), (D,6)}
REVISE*(D,5)
{(B,3), (B,2), (D,6), (C,4), (B,1), (A,6)}
REVISE*(B,3)
{(B,2), (D,6), (C,4), (B,1), (A,6), (C,5), (D,1), (A,2)}
All remaining revisions done without domain changes
{g,b,y}
D
6
1
5
A
2
{g,b,y}
4
B
3
C
{g,b,y}
{r}

Figure 6: The start of an A*-GAC run on a simple 5-node VC problem (K = 4). The domains of each vertex variable are displayed as small lists (of colors r, g, b and y) near the corresponding vertex in the graph. Elements of the queue are simply pairs, such as (C,4), which represent a TODO-REVISE*(node-C, Constraint-4) request.

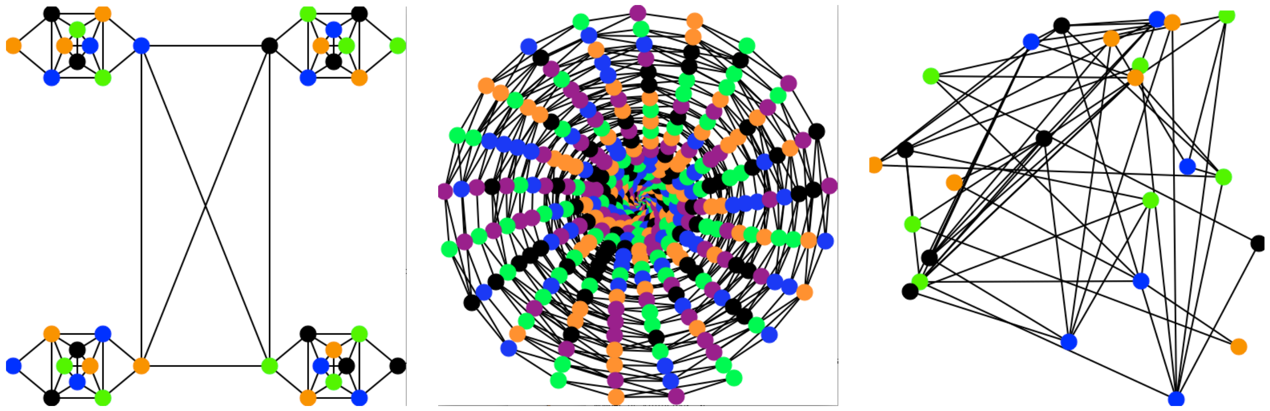

Figure 7: Solutions to vertex-coloring problems on an assortment of graphs, with K = 4, 5 and 4 (moving left to right) for each problem, respectively.

A general VC CSP problem solver needs to create one code chunk (on the fly) **per constraint**, and a graph with E edges will have E constraints. So the **number** of code chunks created will be a function of a runtime user input, although the **form** of each code chunk will be known ahead of time: it is a simple inequality statement.

If the system employs a constraint interpreter instead of creating code chunks, then that interpreter should be able to execute a simple expression (in canonical form), such as " $x \neq y$", and one such expression must be associated with each constraint in the CNET.

## References