8 Constraint Satisfaction

In 2004 Google launched a campaign to recruit employees called the Google Labs Aptitude Test (or GLAT).[64] The test was patterned after standardized tests, and its very first question was:

1. Solve this cryptic equation, realizing of course that values for M and E could be interchanged. No leading zeros are allowed. WWWDOT − GOOGLE = DOTCOM

This is a **cryptarithmetic puzzle**. You are meant to replace each letter with a unique digit such that the resulting equation is correct. You may not use the same digit for two different kinds of letters. That is, if you substitute 8 for W, you cannot also substitute 8 for G. It's more helpful, for purposes of discussion here, to rearrange it as addition, that is, GOOGLE + DOTCOM = WWWDOT. Here's the puzzle and its two solutions.[65]

Puzzle

$$\begin{array}{ccccccc} & G & O & O & G & L & E \\ + & D & O & T & C & O & M \\ \hline & W & W & W & D & O & T \end{array}$$

Solution 1

$$\begin{array}{ccccccc} & 1 & 8 & 8 & 1 & 0 & 3 \\ + & 5 & 8 & 9 & 4 & 8 & 6 \\ \hline & 7 & 7 & 7 & 5 & 8 & 9 \end{array}$$

Solution 2 (M and E swapped)

$$\begin{array}{ccccccc} & 1 & 8 & 8 & 1 & 0 & 6 \\ + & 5 & 8 & 9 & 4 & 8 & 3 \\ \hline & 7 & 7 & 7 & 5 & 8 & 9 \end{array}$$

Cryptarithmetic is an example of a **constraint satisfaction problem**, where the objective is to replace a set of **variables** with certain **values** (a process called **variable assignment**) such that those values meet certain **constraints** (in this case, the rules of mathematics). The set of legal values for each variable may vary from variable to variable and is known as the **domain** of the variable. Constraint satisfaction problems differ from, say, the 15-puzzle (or Rubik's Cube) in a fundamental way. In the 15-puzzle we're not interested in the final solution (we *know* what the final solution looks like). Instead, we're interested in how to solve the puzzle (the proof). But in a cryptarithmetic or other constraint satisfaction problem we're usually not interested in the proof at all: we just want to know what the final solution looks like. Although you can solve constraint satisfaction problems using state-space search, this fundamental difference generally leads to a set of approaches to solving the problem in a more efficient manner. I call these techniques **constraint-based search**.

To formulate the variables, values, and constraints for a cryptarithmetic problem, we must first realize that not all variables are spelled out. For example, in the Google puzzle, in addition to the variables G, O, L, E, D, T, C, M, W, and D, there are also **carry variables** $x_1, ..., x_5$ which are set to either 1 or 0, as shown at right. Given this configuration of the puzzle, we can formulate its components as:

$$\begin{array}{ccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & \\ & G & O & O & G & L & E \\ + & D & O & T & C & O & M \\ \hline & W & W & W & D & O & T \end{array}$$

Figure 45 Variables for the Google cryptarithmetic problem.

- **Variables**: G, O, L, E, D, T, C, M, W, D, x_1, x_2, x_3, x_4, x_5

[64] http://googleblog.blogspot.com/2004/09/pencils-down-people.html

[65] Annoyed I gave the answer away? Here are some more for you to try:
SEND + MORE = MONEY (from "Mathematical Puzzles for Beginners and Enthusiasts" by Geoffrey Mott-Smith)
EAT + THAT = APPLE (from "Take a Look at a Good Book" by Steven Kahan)
TAKE + A + CAKE = KATE
BE × BE = MOB (from "Madachy's Mathematical Recreations" by Joseph S. Madachy)
NO + GUN + NO = HUNT (from "Entertaining Mathematical Teasers and How to Solve Them" by J. A. H. Hunter)

- **Domain**: 0,1 for the x_n variables, and 0 ... 9 for the others
- **Constraints**: (E + M) mod 10 = T (“mod” means remainder after integer division)
  (E + M) div 10 = x_5 (“div” means integer division)
  (x_5 + L + O) mod 10 = O
  (x_5 + L + O) div 10 = x_4
  (x_4 + G + C) mod 10 = D
  (x_4 + G + C) div 10 = x_3
  (x_3 + O + T) mod 10 = W
  (x_3 + O + T) div 10 = x_2
  (x_2 + O + O) mod 10 = W
  (x_2 + O + O) div 10 = x_1
  (x_1 + G + D) = W (Notice no mod or div needed)
  G, D, and W may not be 0 (No leading zeros. This could be put in the domain)
  None of G, O, L, E, D, T, C, M, or W may have the same values

Let’s take another problem: the **N-Queens Problem** described in Section 7. The figure at right shows one of multiple solutions to the 5-Queens problem. We can formulate this as a constraint satisfaction problem by realizing that each column must hold exactly one queen. We can thus treat the columns as variables, and the row-location of the queens on those columns as values.

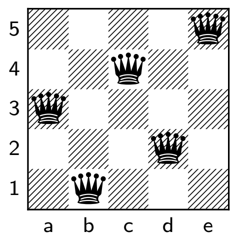


Figure 46 A solution to the 5-Queens problem, showing variables (a, b, c, d, e) and values (1, 2, 3, 4, 5).

- **Variables**: a, b, c, d, e
- **Domain**: 1, 2, 3, 4, 5
- **Constraints**: No two queens may threaten one another

Obviously this single constraint is a doozy: you’ll have to compute a lot about the board to determine it. We could divide it into dozens of smaller constraints (such as “if a = 5, then b may not = 4”). But you get the idea.

Here’s another example. At right is the *Petersen Graph*, a classic 10-node graph structure. The objective is to color each node such that no two nodes of the same color are connected via an edge. You have three colors to work with. The figure shows one possible solution. We can formulate this **graph coloring** problem as:

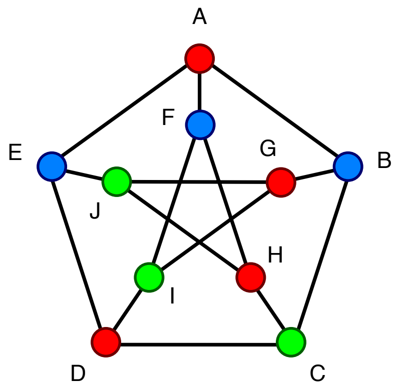


Figure 47 A 3-colored Petersen Graph. Courtesy of Wikipedia.

- **Variables**: A, B, C, D, E, F, G, H, I, J
- **Domain**: red, green, blue
- **Constraints**: A and B may not share the same value
  B and C may not share the same value
  C and D may not share the same value
  D and E may not share the same value
  E and A may not share the same value
  G and I may not share the same value
  H and J may not share the same value
  I and F may not share the same value

J and G may not share the same value
F and H may not share the same value
A and F may not share the same value
B and G may not share the same value
C and H may not share the same value
D and I may not share the same value
E and J may not share the same value

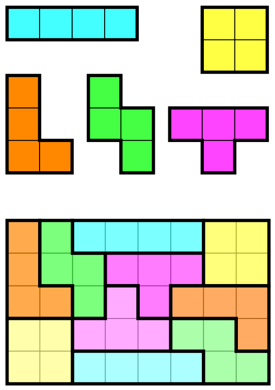


Figure 48 The five tetrominos, and one solution to the 5×8 tetromino tiling problem (using two copies of each tetromino). Courtesy of Wikipedia.

Now consider **Tetromino Tiling**. A tetromino is a shape made out of four adjacent squares, as shown at right. You might recognize these shapes from Tetris.[66]

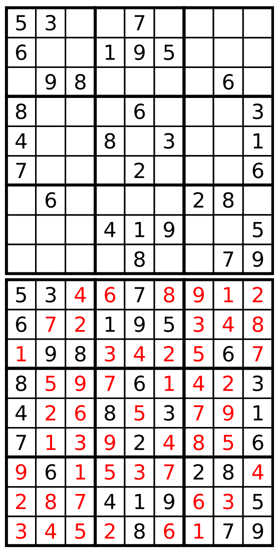
The objective of tetromino tiling is to fill a specific region with a provided set of tetrominos, which may be rotated or flipped as necessary. We can formulate this problem as a constraint satisfaction problem as well:

- **Variables**: Each tetromino in the set
- **Domain**: Possible rotations (4), flips (2), and $\langle x, y \rangle$ positions (40 in the 5×8 problem) for a tetromino, for a total of 320 values.
- **Constraints**: No two tetrominos may overlap
  No tetromino may lie, in part or in whole, outside the problem region (in this case, the 5×8 rectangle)

We presume that the tetromino variables are hard-set to specific tetromino types: else this can be defined in the variable domains or as constraints.

Finally, the classic puzzle craze **Sudoku** is also a straightforward constraint satisfaction problem. In short:

- **Variables**: Each open square location
- **Domain**: 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Constraints**: A value may not appear twice in any column
  A value may not appear twice in any row
  A value may not appear twice in any outlined 9×9 box

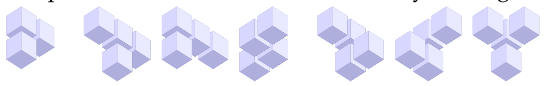


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 49 A Sudoku puzzle and its solution. Courtesy of Wikipedia.

Logistics Constraint satisfaction is particularly common in planning and scheduling problems: everything from determining optimal factory floor plans to scheduling astronomer usage of the Hubble Space Telescope.[67] How might you cast a problem like these as a constraint satisfaction problem? Consider the problem of devising a student schedule for all four years as a computer science major. To keep things simple, let's assume that every course is 3 credits. You might construct the problem like this:

[66]A classic 3-D version of this problem is the **Soma Cube**, where you are given the following **polycubes** consisting of 3 or 4 cubes glued together:  The objective is to put them together to form a 3×3 cube (or certain other shapes). Pictures, as usual, courtesy of Wikipedia. I'll leave it as an exercise to work out the formulation of the Soma Cube as a constraint satisfaction problem.

[67]Seriously. I knew someone at NASA in charge of this.

- **Variables**: 60 courses
- **Domain**: All valid combinations of course numbers, semesters, years, and sections (for example, one value would be CS480/Fall/2011/001), or *nil*
- **Constraints**: The courses must satisfy all general education requirements
  The courses must satisfy all major requirements
  No two courses may overlap in time
  No more than six non-*nil* courses may be taken in a semester
  No fewer than three non-*nil* courses may be taken in a semester
  Courses may only be taken after their prerequisites have been taken
  At least 40 courses must be non-*nil*

I included *nil* to make it easier to take more than 40 credits of courses. Notice that several of these constraints are of a different kind than we've seen so far. In previous examples, the constraints were always $A = B$ or *"foo is not permitted"*. Those kinds of constraints can be checked immediately in the middle of building an incomplete solution. But "The courses must satisfy all general education requirements" can only be checked when the solution is complete (or near enough that it's easy to tell that it's impossible to satisfy the constraint with the remaining unassigned variables). There are several other constraints like this (which ones?).

8.1 Backtracking

The goal of a constraint satisfaction search algorithm is to produce a solution which is both **complete** and **correct**. By *correct* we mean that, for every variable assigned a value, those values collectively do not violate any constraints (that it, it is **consistent** with them). By *complete* we mean that every variable has been assigned a variable. Recall that there are two general ways of searching for a solution: by hunting through the space of *incorrect but complete* states until we find one which is both complete and correct; or by hunting through the space of *incomplete but correct* states until we find one which is both complete and correct. In many cases it's more efficient to do the latter in constraint satisfaction search. So that's what we'll do here.

The general procedure will be to do the following:

1. Select a variable
2. For each possible value you could set that variable to, set the value and recurse to see if this lead to a solution
3. If it does, return it

This procedure is known as **backtracking**: since if trying one value setting doesn't lead to a solution, we backtrack and try the next, and so on. It's a variation on depth-first search.

The first interesting thing is that **correctness doesn't matter what variable you select first**. If there exists a solution, you'll get to it by first working on variable X, then later Y, just as likely as you will if you start on variable Y, then later X. However you can make the process *more efficient* by carefully selecting which variables to go after first, and also by carefully selecting which values to try first. This is where the **heuristic** nature of the search comes into play. We'll get to that in a bit.

The second interesting thing is that if you take this route (incomplete but correct states), your state space has no cycles: since you're always adding another assigned variable, at some point you'll run out of unassigned variables. Thus the depth of your search tree is bounded to the number of variables. So we don't need a "maxdepth" parameter.

Algorithm 22 *Backtracking Search*

```
1: $S \leftarrow$ current state ▷ Including current domains. Some variables may have been assigned to values.
2: $select(...) \leftarrow$ returns an unassigned variable to work on
3: $domain(...) \leftarrow$ returns all possible values which may be assigned to a given variable
4: $infer(...) \leftarrow$ reduces the domains of various variables where possible
5: $expand(...) \leftarrow$ produces a single new state from an old state and the assignment of a variable
6: $consistent(...) \leftarrow$ consistency predicate function
7: $complete(...) \leftarrow$ completeness predicate function

8: if complete($S$) then                                          ▷ Found a solution!
9:     return $S$
10: else
11:     $S \leftarrow$ infer($S$)                                          ▷ Optional.
12:     $var \leftarrow$ select($S$)                                ▷ pick a variable to work on
13:     $D \leftarrow$ domain($var$)
14:     for each variable $v \in D$ do
15:         $C \leftarrow$ expand($S$, "$var = v$")           ▷ Copy $S$ and set $var = v$ in the copy
16:         if consistent($C$) then
17:             $result \leftarrow$ Backtracking Search with $C$, $select, domain, infer, expand, correct,$ and $complete$
18:             if $result \neq$ FAILURE then
19:                 return result
20:     return FAILURE                                      ▷ Couldn't find an answer
```

This algorithm requires you to provide several functions. Let's go through these functions one by one.

- **select(...)** Given a state S, this function selects an unassigned variable to start working on. Technically it doesn't matter what variable you return, but the variable you select has a huge impact on how long it takes to find a solution. We'll get to that in a bit.
- **domain(...)** Given a variable *var*, this function returns all the possible values you could set *var* to (the *domain* of *var*).
- **infer(...)** Given a state S this function reduces the domains of each of the variables in S as much as it can, by removing those values whose setting would make S inconsistent. This is an optional preprocessing step to reduce the search: it's not technically necessary.
- **expand(...)** Given a state S and an assignment "$var = v$", creates a new state C which is a copy of S but where that assignment is now set.
- **consistent(...)** Returns whether or not all variables in S are consistent given the constraints. You don't have to do this if you're weeding out all possible inconsistent variable settings in the infer(...) function.

- **complete(...)** Returns whether or not S is complete, that is, all variables have been assigned.

Figure 50 shows the search tree produced by this algorithm on the 5-Queens problem, where at each level select(S) is defined as just picking the next column, left to right.

Where Do You Check for Inconsistency? Notice that there are **two places** where you could check for consistency: the infer(...) function (Line 11) or the consistent(...) function (Line 16).

The infer(...) function is optional. Sometimes it's cheap to do some checking beforehand to weed out the stupid assignments, as a preprocessing step before doing further recursion. For example, you might have some constraints which only involve a single variable (like "G may not be set to 0" or "Tetromino 4 may not lie outside the problem region". These **unary constraints** are often efficiently checked beforehand in the infer(...) step. This is known as checking for **1-consistency** (sometimes called **node consistency**). It's *possibly* worthwhile to also check for **binary constraints** at this step: constraints involving two variables, such as "J and G may not share the same value" or "No two courses may overlap in time". This is known as **2-consistency** (sometimes called **arc consistency**).

The most well-known preprocessing algorithm for unary and binary constraints is the **AC3 Inference** algorithm, shown here:

Algorithm 23 *AC3 Inference*

```
1:  $S \leftarrow$ Current state
2:  $V \leftarrow \{var_1, ..., var_n\}$ Variables in $S$.
3:  $D \leftarrow \{D_1, ..., D_n\}$ Domains for each variable in $S$.

4:  for each $D_i \in D$ do
5:      Remove from $D_i$ all values $v$ where setting $var_i = v$ would violate a unary constraint in $S$
6:  $W \leftarrow \{\}$
7:  for each $var_i \in V$ do
8:      for each $var_j \in V,\ i < j$ do
9:          if there is a binary constraint involving $var_i$ and $var_j$ then
10:             $W \leftarrow W \cup \{\langle i, j\rangle\}$
11: do
12:     Select some arc $\langle i, j\rangle \in W$
13:     $W \leftarrow W - \{\langle i, j\rangle\}$
14:     if ArcReduce($S, V, D, \langle i, j\rangle$) is $true$ then
15:         if $D_i$ is empty then
16:             return FAILURE
17:         else
18:             for each $var_k \in V,\ k \neq i,\ k \neq j$ do
19:                 if there is a binary constraint involving $var_k$ and $var_i$ then
20:                     $W \leftarrow W \cup \{\langle i, k\rangle\}$
21: while $W$ is nonempty
22: return the revised $S$
```

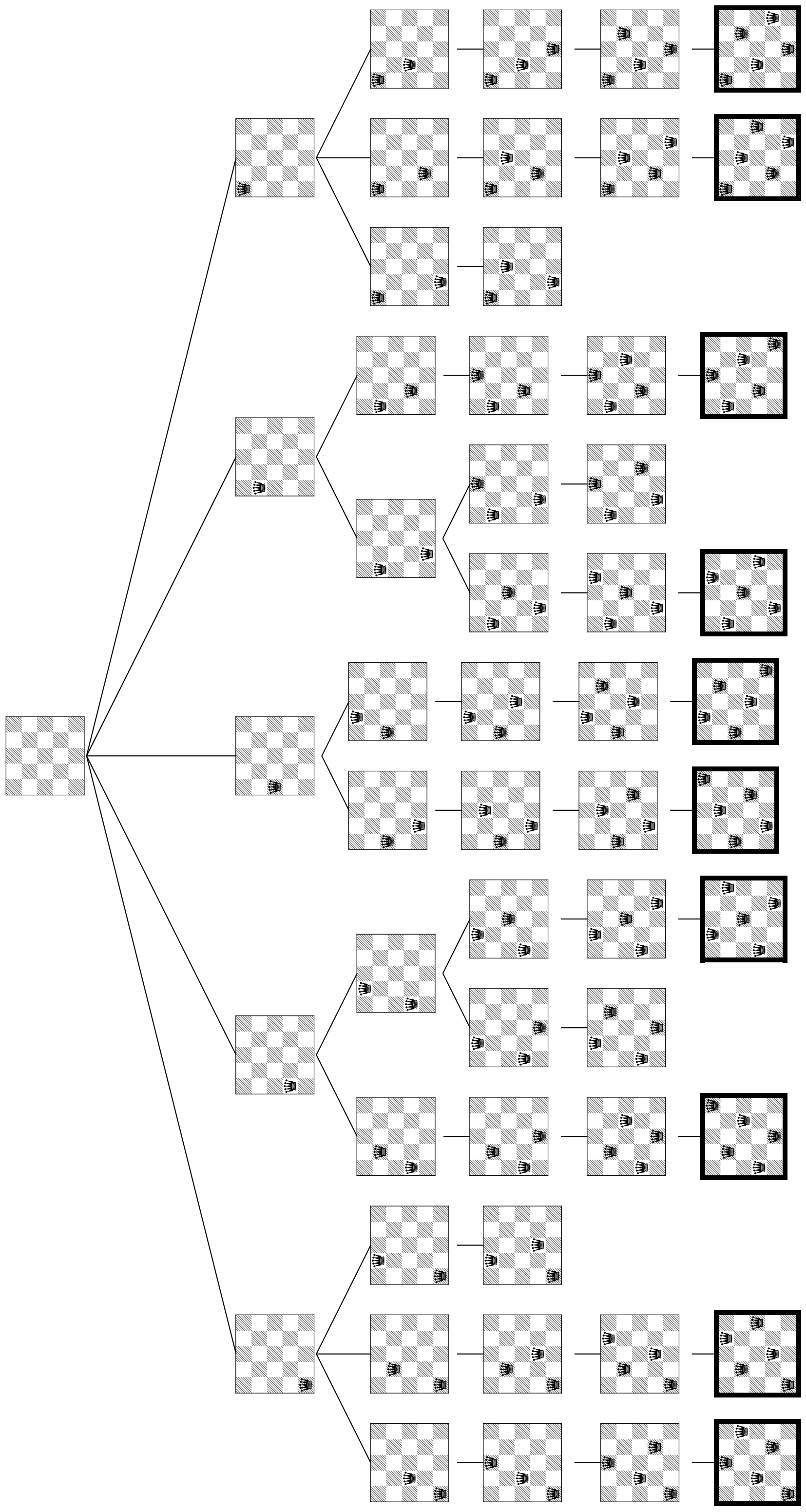


Figure 50 Full constraint satisfaction search tree for the 5-queens problem, where the select(S) is defined as selecting columns left to right. Solutions shown in bold.

Algorithm 24 *ArcReduce*

```
1: $S \leftarrow$ Current state
2: $V \leftarrow \{var_1, ..., var_n\}$ Variables in $S$.
3: $D \leftarrow \{D_1, ..., D_n\}$ Domains for each variable in $S$.
4: $\langle i, j \rangle \leftarrow$ Arc to reduce

5: $change \leftarrow$ false
6: for each $v_x \in D_i$ do
7:     if there does not exist a value $v_y \in D_j$ such that setting $var_i = v_x$ and $var_j = v_y$ will satisfy all
       binary constraints involving $var_k$ and $var_i$ together then
8:         $D_i \leftarrow D_i - \{v_x\}$
9:         $change \leftarrow$ true
10: return change
```

AC3 begins by removing from the domains any values which violate unary constraints. It then uses a queue of binary constraints (the arcs) checks for values in domains which violate binary constraints (via the function **ArcReduce**). If a value violates a binary constraint, it is removed, and any variable in that constraint has all of *its* constraints added back into the queue to be rechecked. When the queue is exhausted, AC3 is done.

AC3 is $O(bd^3)$ in computational complexity, where b is the number of binary constraints (arcs), and d is the size of the largest domain. Is an n^3 algorithm worth your time every step? Maybe, since constraint satisfaction in general is exponential. Or maybe not: it might not be worth the time.

Some larger constraints, involving more variables, (so-called **k-consistency** checking) may be more expensive to check at the infer(...) step and it might make more sense to only whittle down variables with these constraints as you need to, during the consistent(...) step. These are constraints such as " (x_5 + L + O) div 10 = x_4" or "no more than six non-*nil* courses may be taken in a semester". This last constraint example is often called a **resource constraint**: you have constraints on a resource (here, a semester) that may be shared among variables (courses).

Finally, so-called **global constraints**, which involve *all* the variables, are usually worthwhile doing at the infer(...) step because they can be done quickly and you'll need to do them regardless. These include constraints like "None of G, O, L, E, D, T, C, M, W, or D may have the same values". Some other global constraints can only be checked at the very end. An example of these is "The courses must satisfy all general education requirements." You can check for these in either the infer(...) or consistent(...) functions. If you use infer(...), check when S has all but one variable left to assign (the one you're assigning now). If you use consistent(...), check when C has had all its variables assigned.

8.2 Heuristic Backtracking

Besides the consistency checking options discussed above, there are two major places where you can add heuristic knowledge about your problem to help the search go faster than brute-force depth-first search:

- Choose the right variable to work on next

- Choose the right order in which you'll try assigning values to that variable

Like consistency checking, these heuristic decisions are often domain-specific. But here are some good heuristics:

- **Select the variable which has the fewest consistent values left as options.** The idea here is that if there's a variable with very few options left, you want to try it first because if it fails, you'll probably find very soon. That way you can avoid deep searches which all end with this variable failing. As an extreme example: if the variable has *no options left*, then you've already failed! So you'd want to check that first always. This is often called the **Minimum Remaining Values** or **MRV** heuristic. It might be useful here to have handled a lot of variable reduction in the infer(...) function.

- **Select the variable which is involved in as many constraints as possible.** The idea here again is to pick a variable likely to cause early failure, because assigning it to a value will deeply constrain so many other variables (wiping out as many remaining options as possible). This isn't as powerful as MRV, but it's useful to use as a tie-breaker for MRV. It's known as the **Degree** heuristic. Again, it might be useful here to have handled a lot of variable reduction in the infer(...) function.

- **Select the value which reduces the options for the remaining variables as little as possible.** Strangely, this is the opposite of the fail-as-soon-as-possible approach for setting variables. The idea here is that we want to increase our chances of finding an answer, and picking a value which reduces options for other variables will reduce that chance. This heuristic, called the **Least Constraining Value** or **LCV** heuristic, might essentially require you to call the infer(...) function internally as part of its computation. (You might want to store away those results so you don't have to recompute them!)

Why are the first and last heuristics so different? Here's something to ruminate on. In constraint satisfaction search, you are not backtracking on variable selection (it doesn't matter what order you pick variables, as you'll still find the solution if one exists). You are only backtracking on the *value selection* for a variable. If you select a particular variable and it doesn't work out, you're done: no solution exists. But if you try a *value* and it doesn't work out, you have to try the next value and see if that one works out, and so on.

Where to Go From Here? Constraint Satisfaction is a very old, and very well studied problem with an enormous number of algorithms and approaches. Just within the search framework described earlier, there are techniques such as **backjumping** and **forward checking** which trade off preprocessing steps for recursive steps. And decisions regarding variable and value ordering, and when you check for consistency, are domain-specific and require a lot of thought to keep from searching too much to find a solution. There are a lot of algorithm variants here.

8.3 Greedy Search

An alternative is to instead search through the space of **incorrect but complete** solutions until you find one which is both complete and correct. Recall in Figure 36 (page 82) that, for the **N-Queens problem** this might consist of plopping down N random queens, one for each column and then searching though the space of states where you move a queen (within her column) to transition from one state to another. Ordinarily this is a bad way of doing constraint satisfaction: but surprisingly you can often do a great job on some constraint satisfaction problems with a greedy version. Instead of recursively searching the space in a depth-first fashion, simply move forward through the space by changing one queen at a time and never go back. Keep moving queens until you find an answer.

Surprisingly this often works really well. But to do it right you have to *move a queen to the right place* each time. This is again an opportunity for a heuristic. The heuristic will be:

- **The Minimum Conflicts Heuristic** Change the variable's value to the one which violates the fewest constraints. In the N-Queens problem, this means: move the queen to the spot in her column where she attacks the fewest other queens.

The algorithm is simple: we repeatedly pick a random queen which is violating a constraint. We then move the queen to the spot where she violates the fewest constraints. And we repeat until no queen is violating any constraints. And that's it!

Algorithm 25 *Min-Conflicts Greedy Search*

```
1: $S \leftarrow$ initial state                                  ▷ With random assignments to all variables
2: $randomVariable(...) \leftarrow$ returns a random variable violating a constraint in $S$
3: $bestValue(...) \leftarrow$ returns the value for a given variable which violates the fewest constraints in $S$
4: $expand(...) \leftarrow$ produces a single new state from an old state and the (re-)assignment of a variable
5: $consistent(...) \leftarrow$ consistency predicate function

6: loop
7:     if consistent($S$) then
8:         return $S$                                                  ▷ Found a solution!
9:     else
10:        $var \leftarrow$ randomVariable($S$)
11:        $value \leftarrow$ bestValue($var$, $S$)
12:        $S \leftarrow$ expand($S$, "$var = v$")
```

Note that the algorithm, unlike Backtracking, won't terminate if it can't find a solution: it'll just keep on going. You might modify it to run for a limited (if long) period of time.

8.4 Constrained Optimization

So far the constraints we've seen have been either met or unmet: either the states are correct (consistent) or incorrect. Note however that in Figure 50 there were no less than ten solutions to the 5-Queens problem.[68] What if, for example, we **preferred** a solution where a Queen sat directly in the center of the board (that is, at C3)?

Now we've introduced a **soft constraint**: a *quality assessment*[69] we apply to a state rather than simply declaring it right or wrong. We cannot accept wrong solutions: but among the right solutions we'd prefer the ones with the highest quality. We've seen quality assessments before: in

[68]This isn't always the case: often there are very few solutions, or only one (or maybe zero).

[69]The constraints we've seen up to now are usually called **hard constraints**.

metaheuristics. Most metaheuristics are pure optimization (soft constraints), and the constraint search algorithms described so far are pure hard constraints.

There exist quite a number of algorithms which handle both hard and soft constraints in certain situations, such as the **Simplex Algorithm** or the algorithm known as **Branch and Bound**. In the more general context, we need an algorithm which allows for an arbitrary quality function in the context of hard constraints. One easy way to do this is to use an evolutionary algorithm. Here's a simplistic approach. Let's imagine that the soft constraint was a value s from 0 to n, with 0 being most preferred. And let's say that the number of hard constraints violated is h. We could construct a fitness function as $s + h \times n$. Lower fitnesses are preferred. The idea would be that a solution which violated fewer hard constraints would be preferred, but ties would be broken by the solution which had the lowest soft constraint result.