Efficient Constraint Satisfaction

MICHAEL WOLLOWSKI

Forward checking

Inference can be very powerful in the course of a search.

Every time we make a choice of a value for a variable, we have an opportunity to infer new domain reductions on the neighboring variables.

One of the simplest forms of inference is called **forward checking:**

- Whenever a variable *X* is assigned, the forward-checking process establishes consistency for it: for each unassigned variable *Y* that is connected to *X* by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for *X.*

Constraint Graphs

Before looking at forward checking, let's look at a simple example.

Consider the following CSP:

- Four variables: X, Y, Z, T
- Domains for each variable: {1, 2, 3}
- Constraints:
  - X < Y
  - Y = Z
  - T < Z
  - X < T

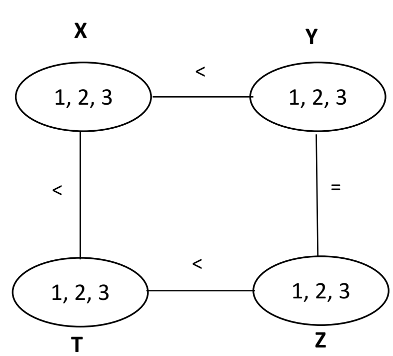
Constraint Graph

Here are the constraints draw in a constraint graph.

Let's begin by solving the problem with plain *backtracking*.

We will solve it in following order:

X, Y, Z, T

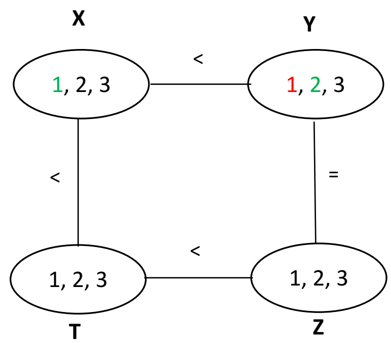


Solving a Constraint Graph with Backtracking

We will pick 1 for X

We then start with 1 for Y but that violates the X<Y constraint.

So we will pick 2

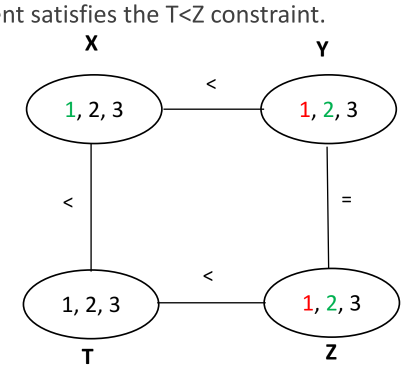


Solving a Constraint Graph with Backtracking

We will try 1 for Z, but that violates the Y=Z constraint.

We will pick 2.

We then try 1 for T, this assignment satisfies the T<Z constraint.

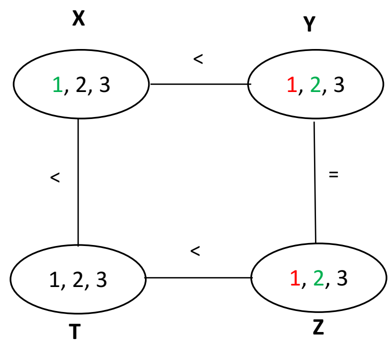


Solving a Constraint Graph with Backtracking

We then check the X<T constraint and realize that it is not satisfied.

We try the value 2 for T but that does not satisfy the T<Z constraint, neither does 3 for T.

We backtrack to Z.



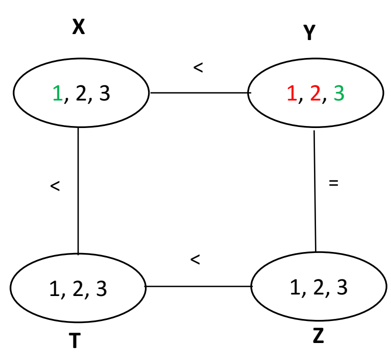
Solving a Constraint Graph with Backtracking

At Z, we try the value 3, but that violates the Y=Z constraint.

We backtrack to Y.

There we select 3.

We then advance to Z.

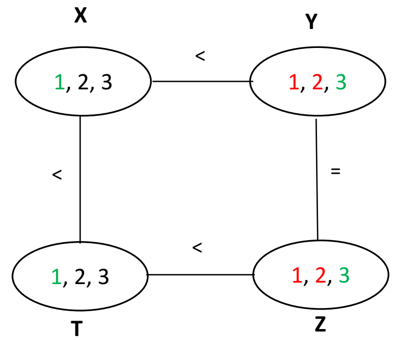


Solving a Constraint Graph with Backtracking

At Z, we try 1 and 2 but they violate the Y=Z constraint.

Hence we assign 3 to Z.

We move to T and pick 1, it satisfies the T<Z constraint.



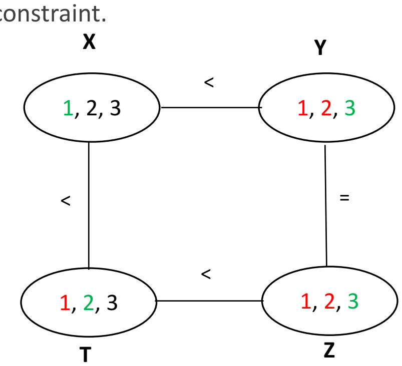
Solving a Constraint Graph with Backtracking

Now we check the X<T constraint and it is violated.

We assign 2 to T.

It satisfies the X<T and the T<Z constraint.

We found a solution.



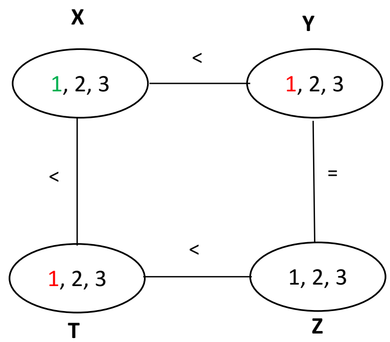
Solving a Constraint Graph with Forward Checking

Now we will solve the CG with forward checking.

We will use the same order of variables.

We begin by selecting 1 for X.

Using forward checking, we will eliminate values from the domains of Y and T as indicated in red.

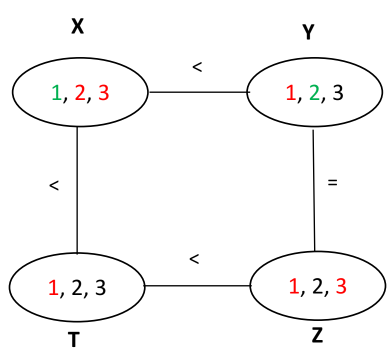


Solving a Constraint Graph with Forward Checking

Next, we assign 2 to Y and apply forward checking.

This eliminates 1 and 3 from the domain of Z.

It eliminates 2 and 3 from X



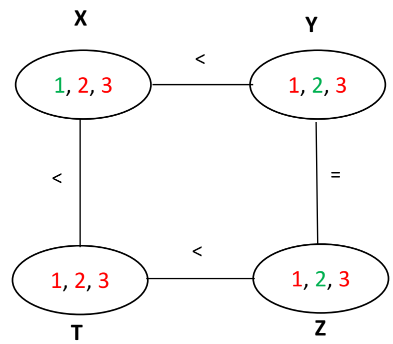
Solving a Constraint Graph with Forward Checking

Next, we select the only remaining value for Z.

We apply forward checking and remove 2 and 3 from the domain of T.

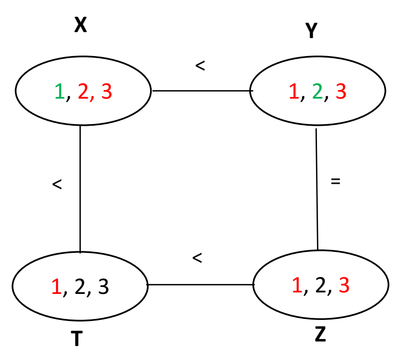
We remove 3 from Y.

At T, we need to backtrack.



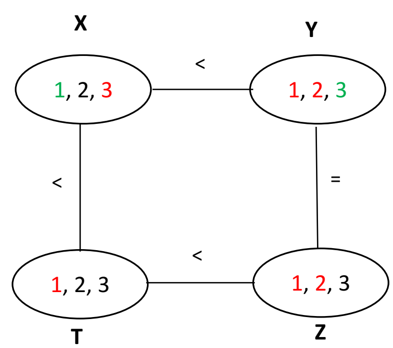
Solving a Constraint Graph with Forward Checking

We also need to backtrack at Z.



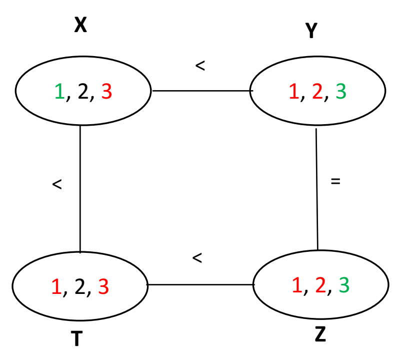
Solving a Constraint Graph with Forward Checking

We pick 3 for Y and perform forward checking on X and Z



Solving a Constraint Graph with Forward Checking

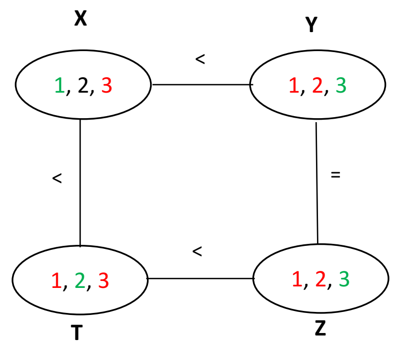
We now select 3 for Z and perform forward checking on T.



Solving a Constraint Graph with Forward Checking

We select 2 for T.

We have found a consistent solution.



Arc Consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.

More formally, an arc (*X, Y*) is *arc-consistent* if, for every value *x* of *X*, there is a value *y* for *Y* that satisfies the constraint represented by the arc.

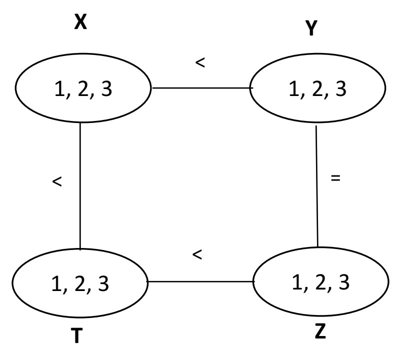
A graph is arc-consistent if all arcs are arc-consistent.

To create arc consistency, we perform *constraint propagation*: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs.

Notice that while in forward checking, we only look at a variables immediate neighbors, constraint propagation looks at the transitive closure of all neighbors.

Arc Consistency

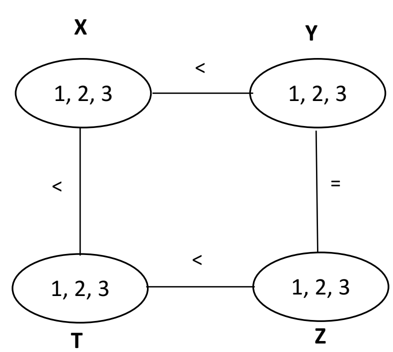
We will now solve the problem below with constraint propagation.



Arc Consistency

Notice that we begin by constraint propagation.

In other words, we do NOT begin by selecting a value for X.



Arc Consistency

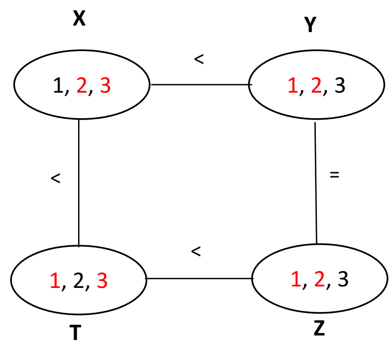
Here is the graph with all constraints enforced.

I will leave it as an exercise to you to figure out how we got here.

For this particular example,

we can now read off a

solution.



Constraint Propagation

The most popular algorithm for arc consistency is called AC-3

To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider.

(Actually, the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue.)

Initially, the queue contains all the arcs in the CSP.

(Each binary constraint becomes two arcs, one in each direction.)

AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j.

Arc Consistency

If this leaves D_i unchanged, the algorithm just moves on to the next arc.

But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i.

We need to do that because the change in D_i might enable further reductions in the domains of D_k even if we have previously considered X_k.

If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure.

Arc Consistency

Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue.

At that point, we are left with a CSP that is equivalent to the original CSP-they both have the same solutions-but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

Arc consistency algorithm AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) <- REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
-----------------------------------------------------------------------
function RM-INCONSISTENT-VALUES( Xi, Xj) returns true iff remove a value
   removed <- false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
         then delete x from DOMAIN[Xi];  removed <- true
   return removed
```

Time complexity: O(nd^3), where *n* is the number of arcs and *d* is the maximum size of a domain.

Algorithm source: Russell and Norvig: AIMA, 2nd Edition, p 146

Constraint Propagation

Interleave constraint propagation and backtracking search.

```
Solve:
  Do constraint propagation until no values change
  If not solved:
    Save state to stack
    Pick a variable and a value, assign it
      Make recursive call to solve
      If success, return
      If not, restore old state and pick the next value
```

Inference

- Consider the Sudoku columns at right.
- Looking at it, we can eliminate some of the numbers from some of the domains.
- This would be accomplished by inference.
- If we look at the 4th and 5th cell from the top, either can only have a value of 1 or 6.
- This means that one of those cells will end up having a value of 1 and the other one will have a value of 6.
- This also means that we can eliminate 1 and 6 from the domains of all other cells.
- This is shown in red in the right-most column.

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

Inference

- Looking at the column, there is only one 8 left, in the top cell.
- We will eliminate 3 and 4 from the top cell, leaving the top cell with just the value 8.

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