Chapter 6: Constraint Satisfaction Problems

Components of a CSP

1. A set of variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$
2. A domain for each variable $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$
   - D_i is a set of allowable values $\{v_1, v_2, \ldots, v_k\}$ for variable X_i.
3. A collection of constraints $\mathcal{C}$
   - A constraint C_j is the pair $\langle \texttt{scope}, \texttt{rel} \rangle$ where the `scope` is the tuple of variables involved in the constraint, and `rel` is the function that checks whether a tuple of values satisfies C_j.

Variable **assignments** are consistent if the values assigned to the variables don't violate any constraints. A **complete assignment** provides a value for each variable. A **CSP solution** is a consistent, complete assignment. A consistent, partial assignment is a partial solution. Partial solutions allow us to eliminate large groups of variable values from further consideration, and in this way CSP formulations can help produce computationally efficient searches.

A simple constraint

Suppose you have variables (X_1, X_2) that for which $D_1 = D_2 = \{1, 2, 3\}$.

The constraint $X_1 > X_2$ is written formally as either

$$\langle (X_1, X_2), X_1 > X_2 \rangle$$

or

$$\langle (X_1, X_2), \{(3, 1), (3, 2), (2, 1)\} \rangle$$

A map coloring example

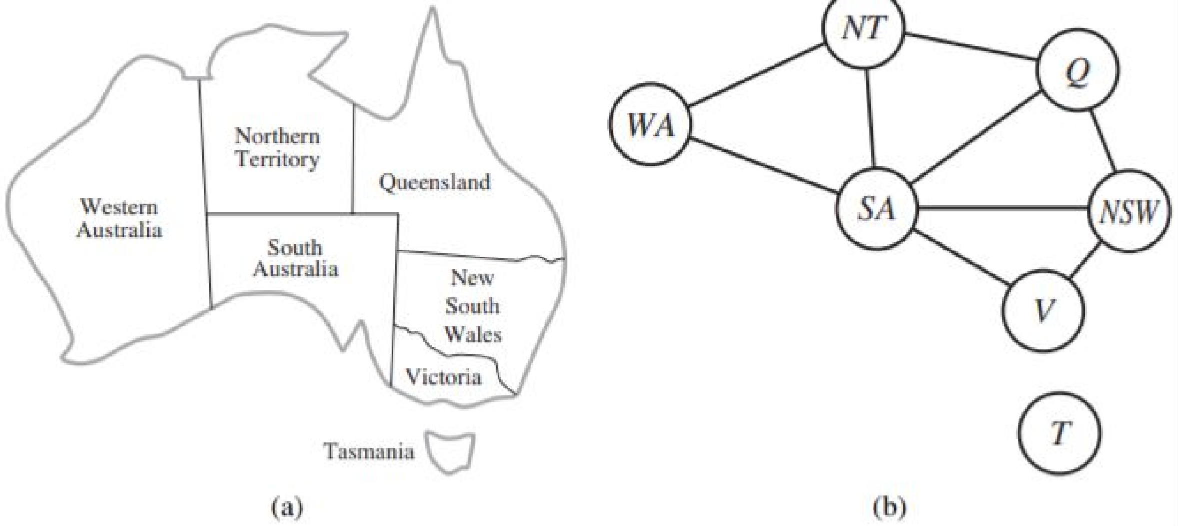


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

A map coloring example as a CSP

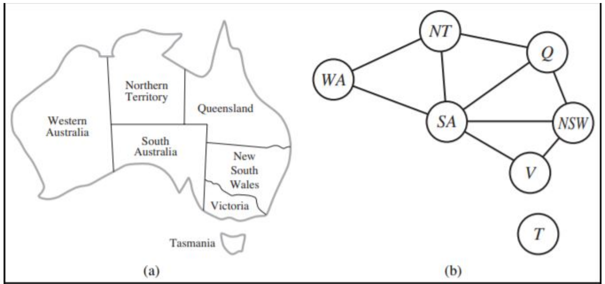


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

The variables are

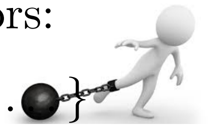
$$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}$$

and each one can assume the colors

$$D = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$$

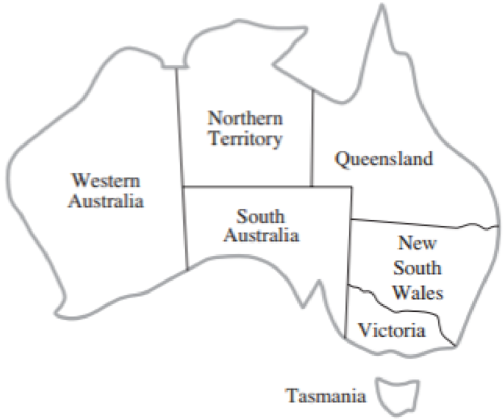
with the constraint that neighbors must have unique colors:

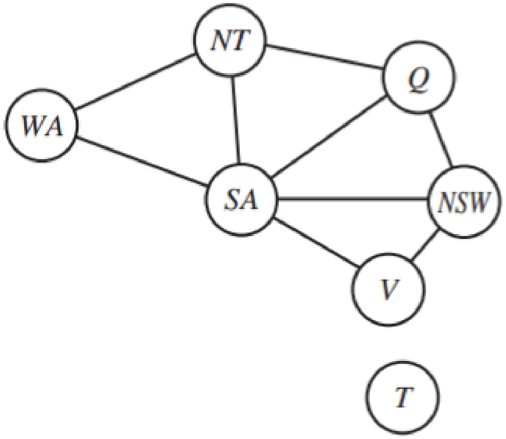
$$\mathcal{C} = \{\langle\{SA, WA\}, SA \neq WA\rangle, \langle\{SA, NT\}, SA \neq NT\rangle, \ldots\}$$



A map coloring exercise from Chegg

1) How many solutions are there for the map-colouring problem in the Australia map (see below)? (tip: Start with SA, which can have any of three colours. Then moving clockwise, WA can have either of the other two colours, and everything else is strictly determined; that makes ... possibilities for the mainland, times ... for Tasmania yields ... solutions). Please provide all possible solutions for the main land (all states except Tasmania). You can either use graphics, or simply state the solution as {WA=red, NT=green, ...}.





2) Explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining in a CSP search.

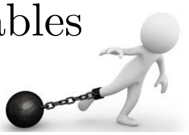
CSP flavors

Types of domains

- Discrete, finite domains
- Discrete, infinite domains (e.g. constraints on integer variables is **integer programming**)
- Linear constraints on continuous domains (**linear programming**)

Types of constraints

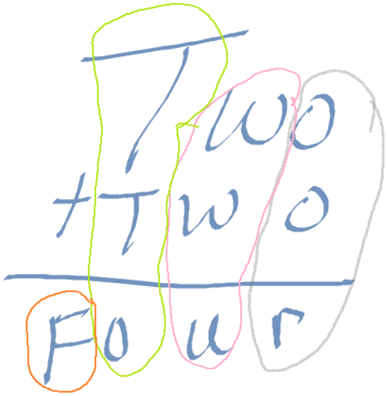
- **unary** constraints act on one variable
- **binary** constraints act on two variables
- **ternary** constraints . . .
- **global** constraints act on an arbitrary subset of variables



Examples of global constraints

`Alldiff` is the constraint that all variable values must differ

Cryptarithmetic puzzles use an `Alldiff` constraint



$O + O = r + 10 * C_{10}$ carry digit

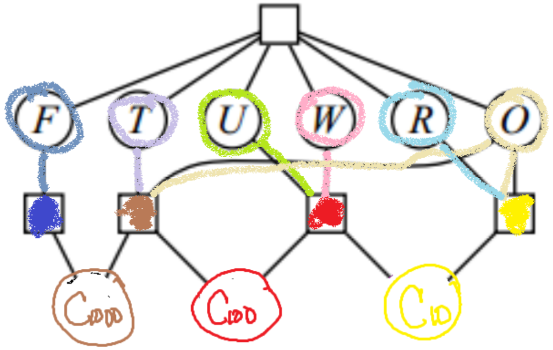
$w + w = C_{10} + u + 10 * C_{100}$

$T + T = C_{100} + O + 10 * C_{1000}$

$C_{1000} = F$

The constraint hypergraph

$$O + O = R + 10 \cdot C_{10}$$
$$C_{10} + W + W = U + 10 \cdot C_{100}$$
$$C_{100} + T + T = O + 10 \cdot C_{1000}$$
$$C_{1000} = F$$



The top square is the initial `Alldiff` constraint, and each additional square represents another constraint. The carry digits become extra variables that had to be introduced (and do not have to be distinct).

Every finite-domain constaint can be reduced to a set of binary constraints

One non-intuitive way to do this is to create a new **dual** problem that has the constraints as variables

$$\begin{aligned} \mathcal{X} &= \{X, Y, Z\} \\ D_X = D_Y = D_Z &= \{1,2,3,4,5\} \\ C_1 &= \langle (X,Y,Z), X+Y=Z \rangle \\ C_2 &= \langle (X,Y), X+1=Y \rangle \end{aligned}$$

$$\begin{aligned} \mathcal{X} &= \{C_1, C_2\} \\ D_{C_1} &= \{(X,Y,Z) \in \{1,2,3,4,5\} \,|\, X+Y=Z\} \\ D_{C_2} &= \{(X,Y) \in \{1,2,3,4,5\} \,|\, X+1=Y\} \\ C &= \langle (C_1, C_2), R \rangle \end{aligned}$$

The relation R contains the variables that co-satisfy C_1 and C_2

Every finite-domain constaint can be reduced to a set of binary constraints

How would you choose to turn this into only binary constraints?

$$
\begin{aligned}
\mathcal{X} &= \{X, Y, Z\} \\
D_X = D_Y = D_Z &= \{1, 2, 3, 4, 5\} \\
C_1 &= \langle (X, Y, Z), X + Y = Z \rangle \\
C_2 &= \langle (X, Y), X + 1 = Y \rangle
\end{aligned}
$$

Constraint Propagation

Again, the power of a CSP is that every time you enforce one constraint, you reduce the number of legal values for at least some of the variables. Propagation through constraints could solve the whole problem, or perhaps reduce the variables that are still needed to be considered if there are multiple domain values remaining for each variable. (Note: a solution may not be unique.)

One strategy is **local consistency** in which *each variable is a node in a graph* and *each binary constraint is an edge in the graph.*

On the previous slide, draw the associated graph.

Node consistency

Each node's domain must satisfy any unary constraints.

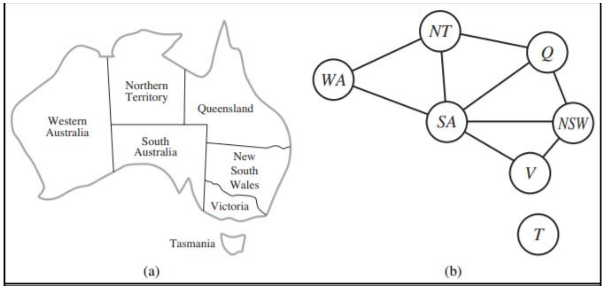


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

The variables are

$$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}$$

$$D = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$$

As a silly example, suppose we know SA hates **`green`**. Then we could start with

$$D_{SA} = \{\texttt{red}, \texttt{blue}\}$$

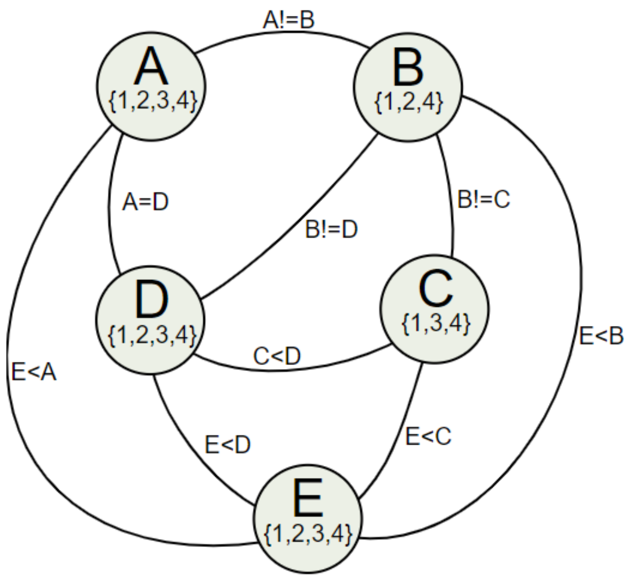
Arc Consistency

The variable X_i is **arc-consistent** with variable X_j if for each binary constraint C with scope (X_i, X_j) and each value $v_i \in D_i$ there exists $v_j \in D_j$ such that C is satisfied.

By applying the rule of arc consistency, we can reduce the domain for a problem. For example, return to the map coloring problem in the previous slide. When we applied the unary constraint to SA, does arc consistency allow any other domain reductions?

Arc consistency Example

Let's think through what can be deduced using arc consistency.



from boristhebrave.com

AC3: Popular Arc Consistency Algorithm

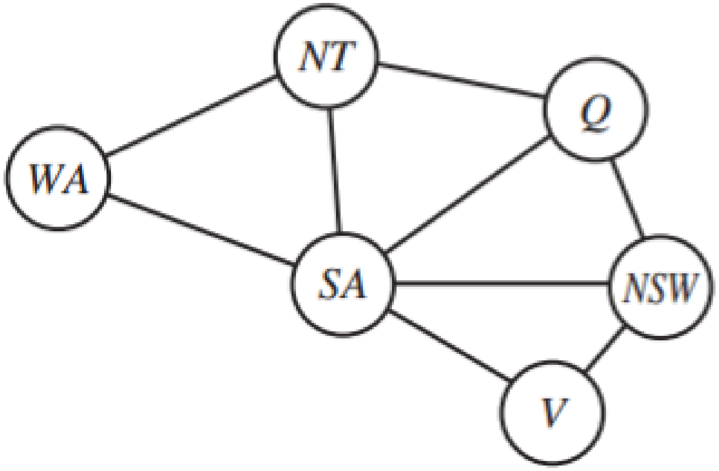
`Queue`: Two arcs (bidirectional) for each binary constraint

Choose an arc to consider.

- If the arc doesn't change any domains, move on to the next arc.
- If the arc *does* change a domain D_i, then add all arcs to adjacent nodes back in (if they aren't already there). AC3's worst-case complexity is $\mathcal{O}(cd^3)$ where c is the number of constraints and d is the maximum domain size.

Path consistency

Rather than just considering binary constraints, pairs of binary constraints are considered by looking at triples of variables. The pair $\{X_i, X_j\}$ is **path consistent** with X_m if for any pair (a, b) that satisfies the binary constraint(s) for $\{X_i, X_j\}$ there exists a value $c \in D_m$ that satisfies any binary constraint(s) between X_i and X_m as well as between X_j and X_m.
How would this help us with the coloring problem if we assume there are only two colors?



k-consistency

This is an extension of the idea of path consistency. A CSP is k-consistent if for any consistent assignment for $k - 1$ variables, a consistent assignment can always be made for a kth variable.

A CSP is **strongly k-consistent** if it $k, (k-1)-, (k-2)-, \cdots 1-$consistent.

In practice, going for more than path consistency is a very expensive approach.

Another global constraint: Resource constraints

An example of a resource constraint is `Atmost`.

Application: Suppose P_1, P_2, P_3, and P_4 people are assigned to four tasks, but that we're not allowed to hire more than 10 people in total. This global resource constraint would be written as

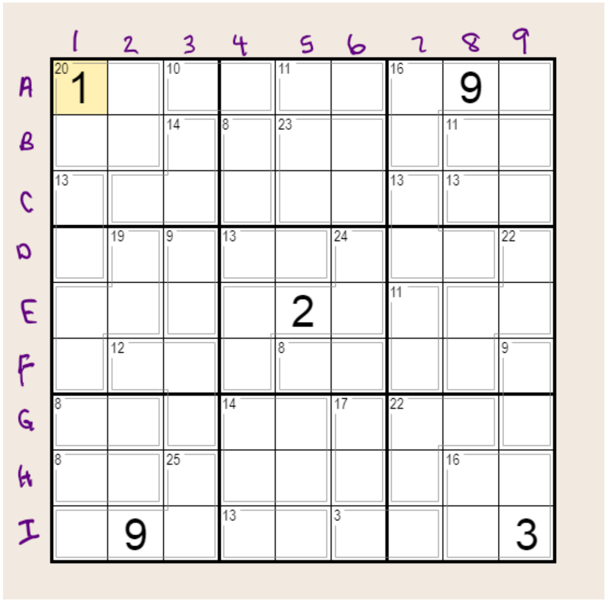
$$\texttt{Atmost}(10, P_1, P_2, P_3, P_4)$$

Given the `Atmost` constraint, how would you edit the shared domain $\{2, 3, 4, 5, 6\}$?

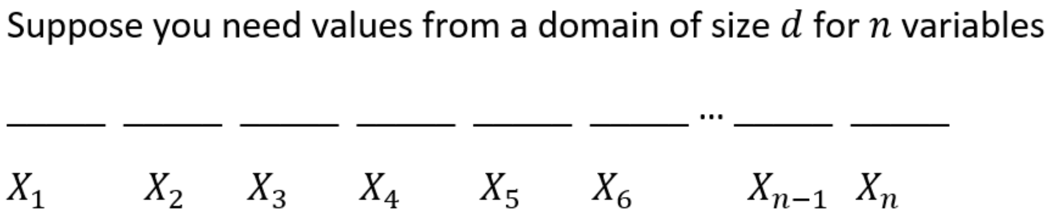
Large problem domains and bounds propagation

If we extended the above problem to a large company, we can't list every integer value for the number of personnel, and we would instead enumerate domains using interval notation. As we apply consistency rules and adjust the domains, we apply **bounds propagation** and perhaps raise the lower bounds and decrease the upper bounds for domains to make them **bounds consistent**.

Let's list some constraints for a Sudoku puzzle



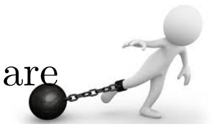
How big is a search?


Suppose you need values from a domain of size d for n variables

If you were to create a search tree, you'd want to first choose one value. So you'd choose one variable (n choices) to assign, and you'd give it one of the d domain values. There are therefore kn ways to start this tree.

At the next tree level, there are only $n-1$ choices remaining, so there are $k(n-1)$ choices for the next level. Continuing this logic, we see that the size of our search tree is $n!d^n$.

However, look at the blanks. How many possible assignments are there? Clearly this is a terrible search tree idea.



Why do we teach CS majors combinatorics?

How many 4-letter “words” can we create from the letters in FLOWERS?

How many collections of 4 letters can we choose from the letters in FLOWERS?

The moral is, no one cares which variable I pick first, second, and so on. So no one cares which of the n variables I start with. Just pick

Backtracking-search

1. Choose an unassigned variable.
2. For each consistent value in its domain, try to extend this to a solution by calling backtracking-search on the solution that uses this value.
   - If the call succeeds, the solution is returned.
   - If the call fails, we try the next value.
3. If we run out of values with no solution we return failure.

Variable order matters in a search

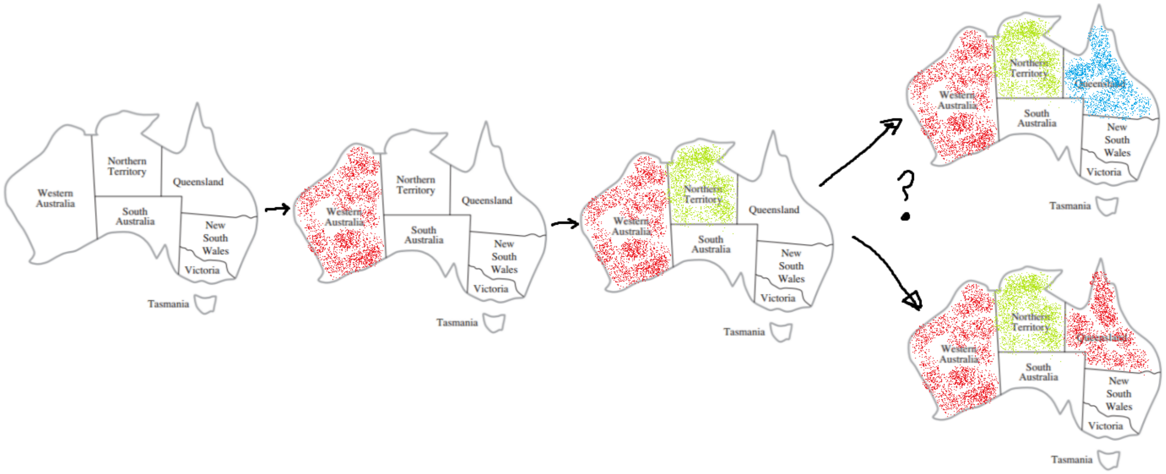
Unassigned variable ordering approaches

- **Minimum-remaining-values (MRV)**: This is the most effective way to prune the search tree early on by using this bottleneck to identify failures down the line.
- **Degree heuristic**: If all of the consistent domains are the same size, MRV doesn't help. Start with the variable that is involved in the largest number of constraints involving other variables.

In what order should we explore values?

- **Least-constraining value heuristic**: Test all values against the the neighbors – variables that share constraints – and work further down the tree by using the value that eliminated the fewest number of values from its neighbors.

Which choice is the least constraining value?



Making CSP search faster

- **Forward checking**: Once you choose a variable value, perform consistency checking for each variable connected to the originating variable by a constraint. Note that without this inference, we'd only check the *next* variable we chose, and many possible eliminations would be missed.
- **Maintaining Arc Consistency (MAC)**: In the previous graph, only the red choice for Queensland allows arc consistency moving forward. MAC alters the AC3 algorithm by searching constraint-neighbor arcs first after a value has been assigned. For example, on the previous slide if I put blue in Queensland, I wouldn't jump to color Victoria next. Rather, I'd want to check whether that would work for my constraint neighbors.

Is there a smarter way to back up?

When we fail in AC3, we just back up and try the next variable. But what if the reason this failed was that the variable *before* this one was a terrible choice? Then I'll do a zillion failures, exhausting all possible choices for the current variable before I back up further.

Other approches:

- **Backjumping** figures out a smarter "jump" to a higher part of the tree that is most likely to be the culprit of the lack of consistency. There are three flavors of this:
  - Gaschnig's Backjumping: focus on the earliest chosen value that conflicts with the current proposed value and jump to that choice in the tree
  - Graph-based Backjumping: focus on the parent variables (not values) in the constraint graph to decide where to jump
  - Conflict-directed Backjumping: combine information from the constraint graph with the minimal prefix conflict set from Gaschnig.

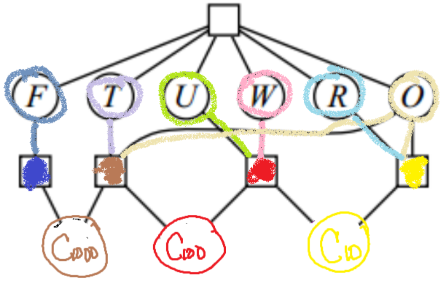
Structuring your problem for a quick solution

In the coloring problem, each **connected component** has a coloring scheme that is independent of other components. So one way to reduce a problem is to split it into its connected components.

But how often is that going to happen?

Tree constraint graphs

$$O + O = R + 10 \cdot C_{10}$$
$$C_{10} + W + W = U + 10 \cdot C_{100}$$
$$C_{100} + T + T = O + 10 \cdot C_{1000}$$
$$C_{1000} = F$$



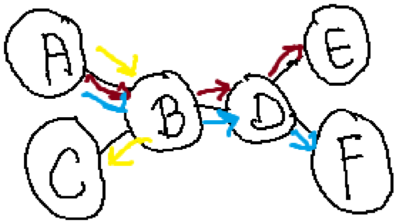
Remember our constraint graph for the Cryptarithmetic example? Look how "O" has two constraints, and thus paths to T and R, as well as our invented variables.

A constraint graph is a **tree** when any two variables are connected by only one path. Such CSPs can be solved in $\mathcal{O}(\# \text{ vars})$.

Why are tree graphs better?

A CSP has **directional arc consistency (DAC)** under an ordering of the variables $X_1, X_2, \ldots X_n$ if and only if every X_i is arc-consistent with each X_j that follows it.

The tree is created by performing a topological sort on its constraint graph – this sort should preserve order, allowing skips.



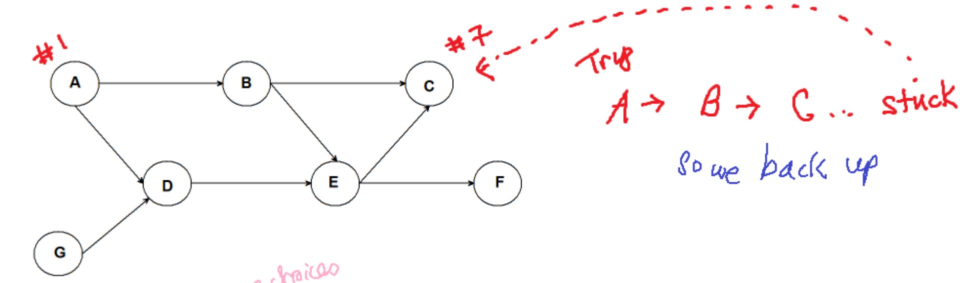
$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & D \arrow[r] & E \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & D \arrow[r] & F \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] \arrow[rr, bend left] & C & D \arrow[r] \arrow[rr, bend left] & E & F \end{tikzcd}$$

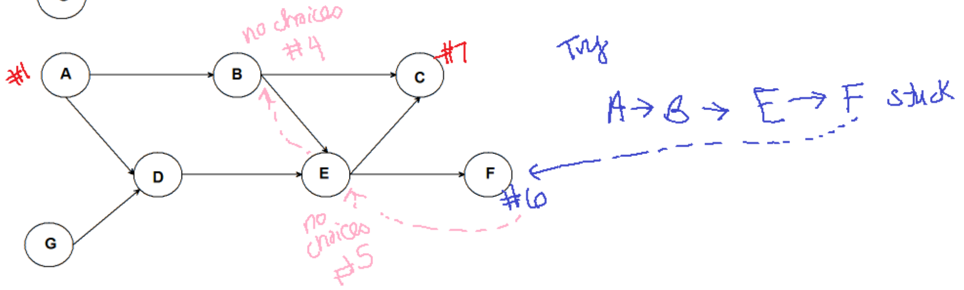
Another topological sort



Try

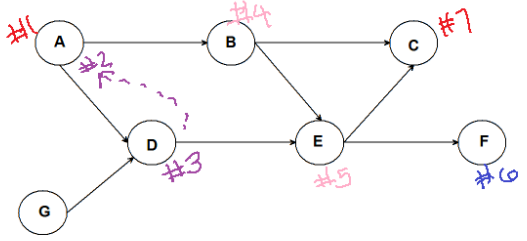
A → B → C … stuck

So we back up



Try

A → B → E → F stuck



Try

A → D → E

but E's top num is 5

Final order is

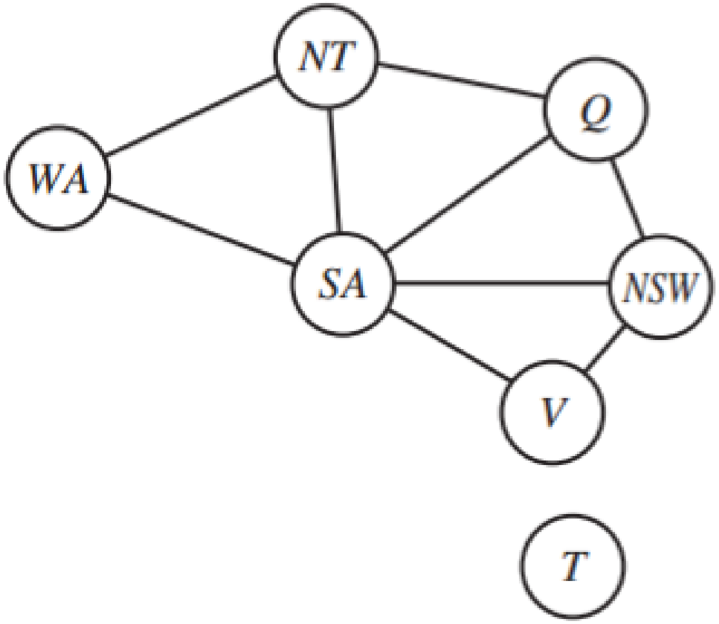
G A D B E F C

How does the topological sort help?

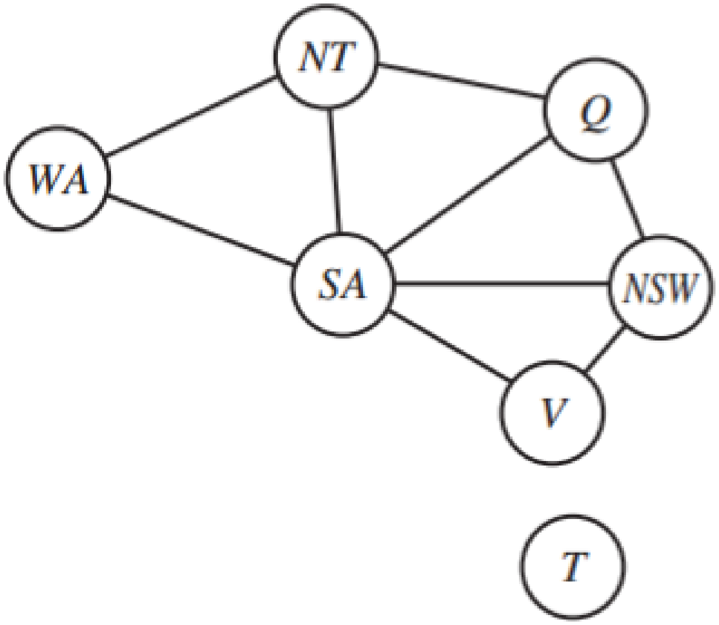
Now the tree can obtain DAC in $\mathcal{O}(n)$ steps. Each step compares up to d possible values for a pair of variables, so the overall computation is $\mathcal{O}(nd^2)$.

Because the graph is arc consistent, we now just choose the solution by choosing any remaining value as we walk down the tree. We never have to backtrack.

Is our Australia problem a tree?



What would it take to make our Australia problem a tree?



Cutset conditioning

- Choose a subset S of CSP's variables so that the graph with those variables removed is a tree. S is called a **cycle subset.**
- Create a list of assignments within S that achieve consistency within S. For set assignment set
  - Remove inconsistent values from the variables in $CSP \setminus S$.
  - If the remaining graph has a solution, return that solution along with the assignments for S.