Constraint Satisfaction

Chapter 6
Part 2

Slides courtesy of Andrea Thomaz

Improving Backtracking

* What variable to assign next?
* What order to try the domain values?
* What inference can be made to detect failure early?
* Can we take advantage of the problem structure?

Improving Backtracking

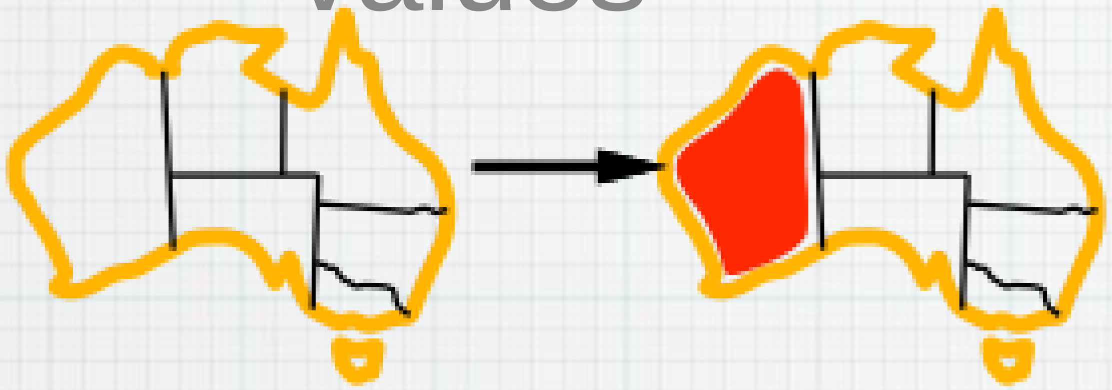
* What variable to assign next?
* What order to try the domain values?
* What inference can be made to detect failure early?
* Can we take advantage of the problem structure?

What var to do next?

* Simplest: some fixed order, or random
* Better idea: choose the most constrained variable as the next one to assign so it doesn't run out of options

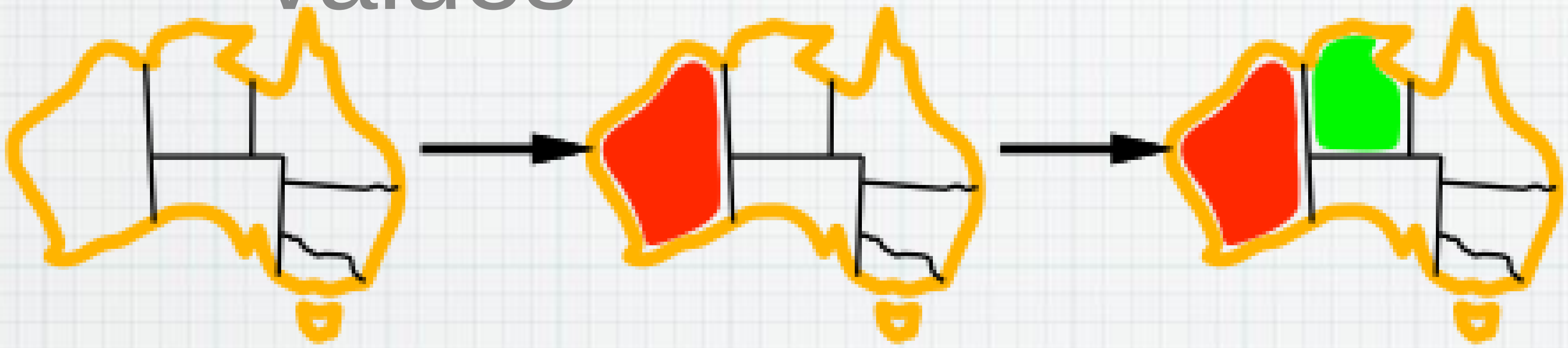
Minimum Remaining Values (MRV)

* Choose var with the fewest legal values



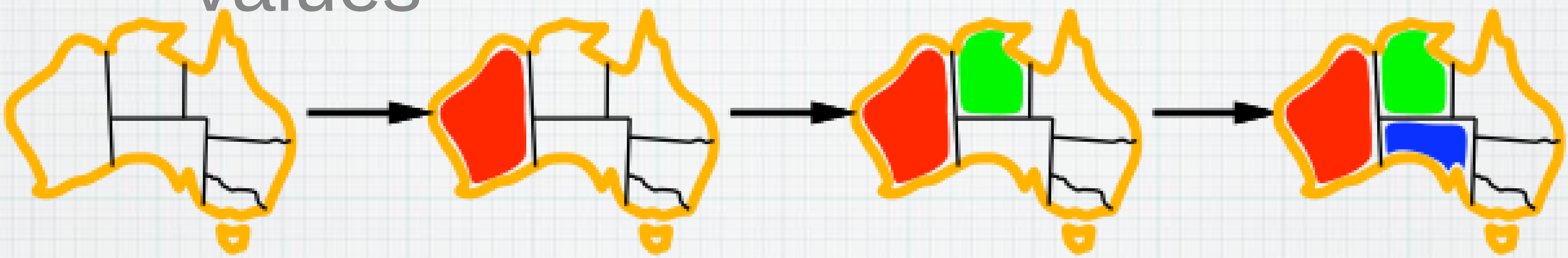
Minimum Remaining Values (MRV)

- Choose var with the fewest legal values



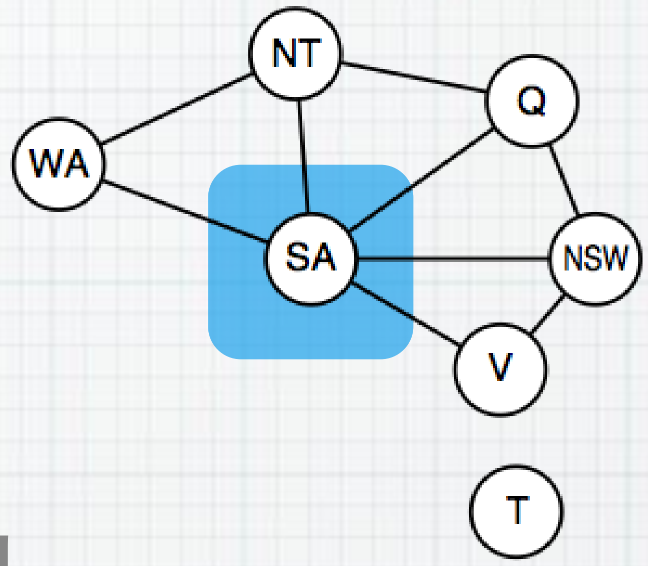
Minimum Remaining Values (MRV)

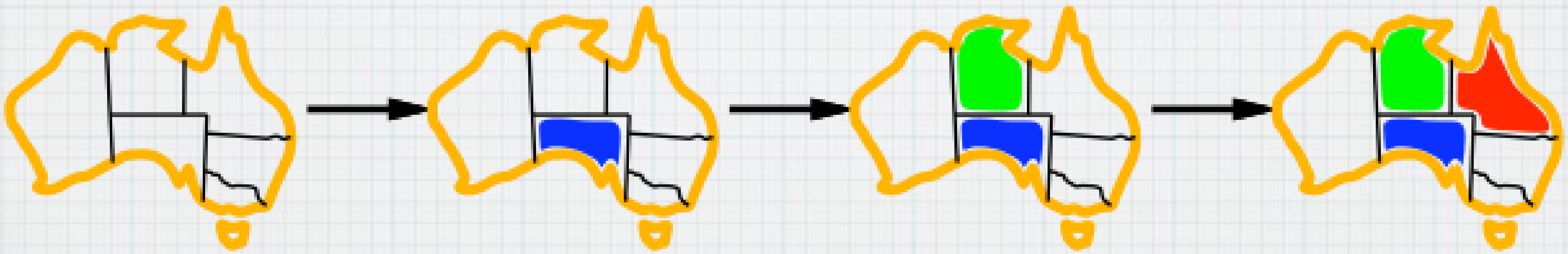
- Choose var with the fewest legal values



Degree Heuristic

- Tie breaker for MRV
- Choose var with most constraints on remaining variables



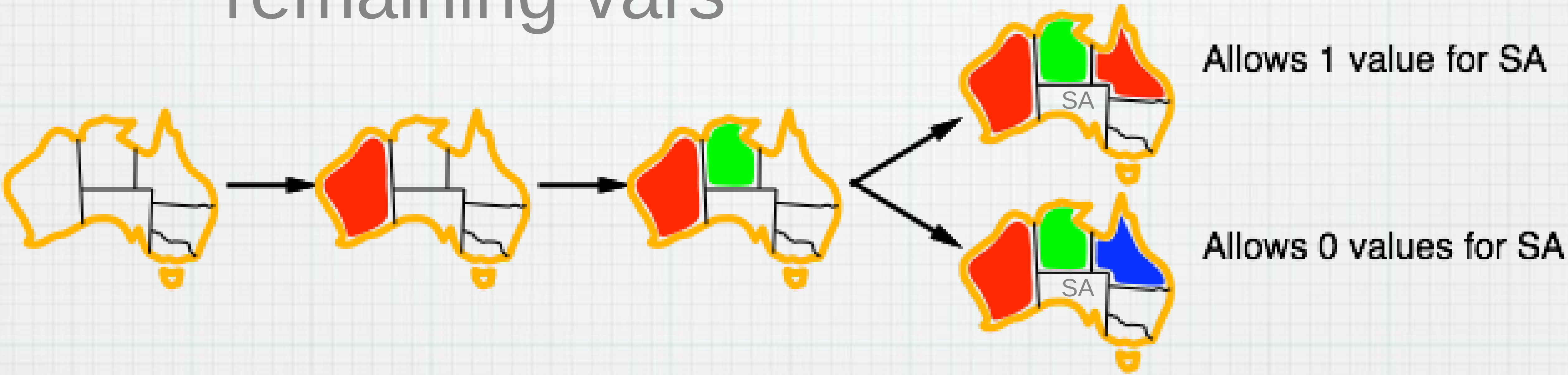


Improving Backtracking

* What variable to assign next?
* What order to try the domain values?
* What inference can be made to detect failure early?
* Can we take advantage of the problem structure?

Least Constraining Value

* Given a variable, choose value that rules out the least values in remaining vars



Improving Backtracking

* Variable ordering -- fail first, to minimize nodes in the search tree
* Value ordering -- fail last, we only need one solution so keep options open
* Can we interleave search and inference to narrow our choices even further?

Improving Backtracking

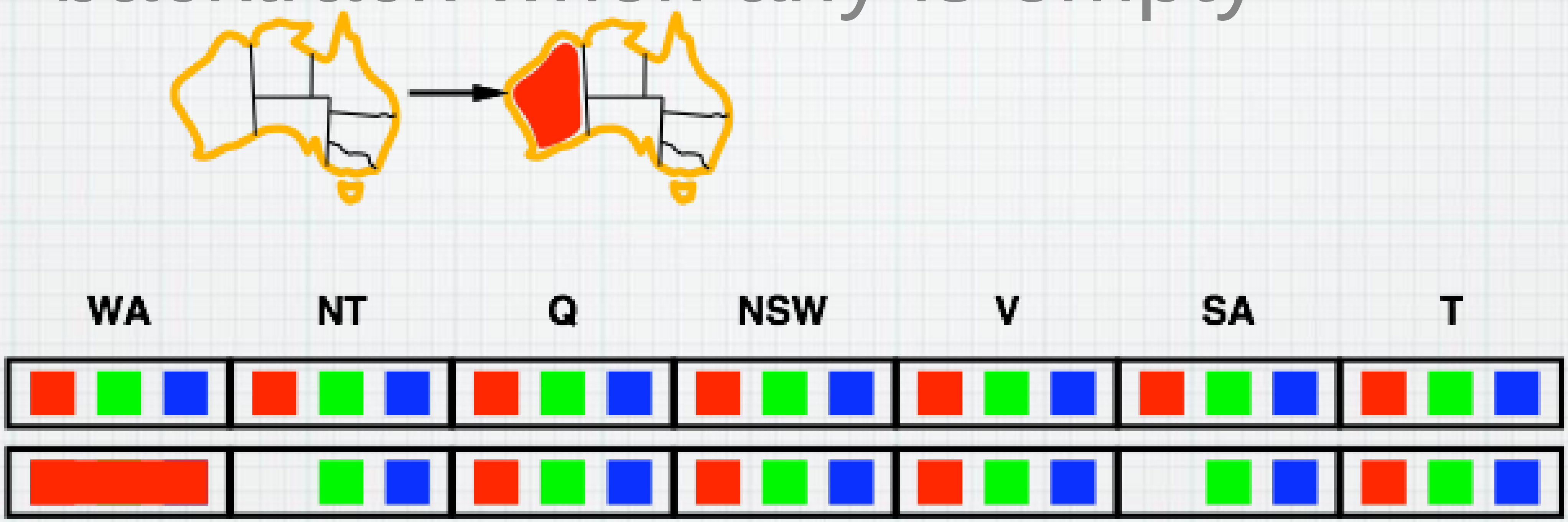
* What variable to assign next?
* What order to try the domain values?
* What inference can be made to detect failure early?
* Can we take advantage of the problem structure?

Forward Checking

* Track remaining legal vals for vars, backtrack when any is empty

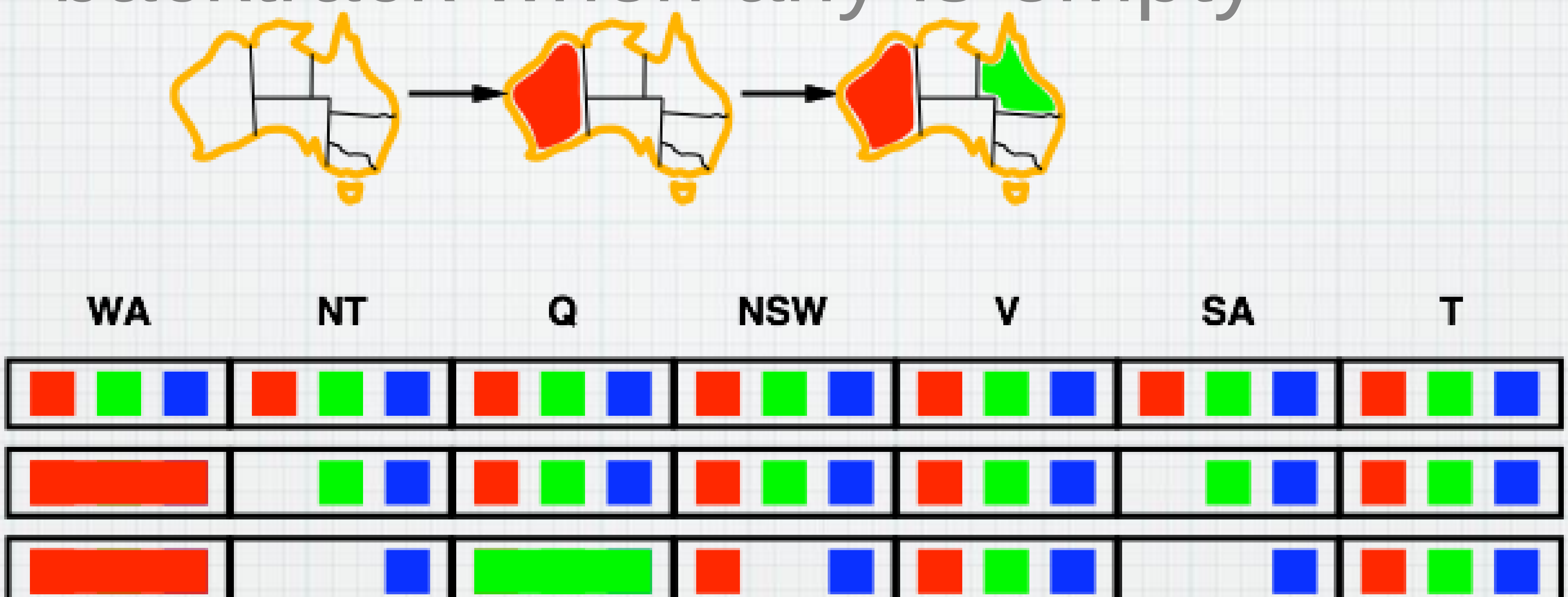
Forward Checking

* Track remaining legal vals for vars, backtrack when any is empty



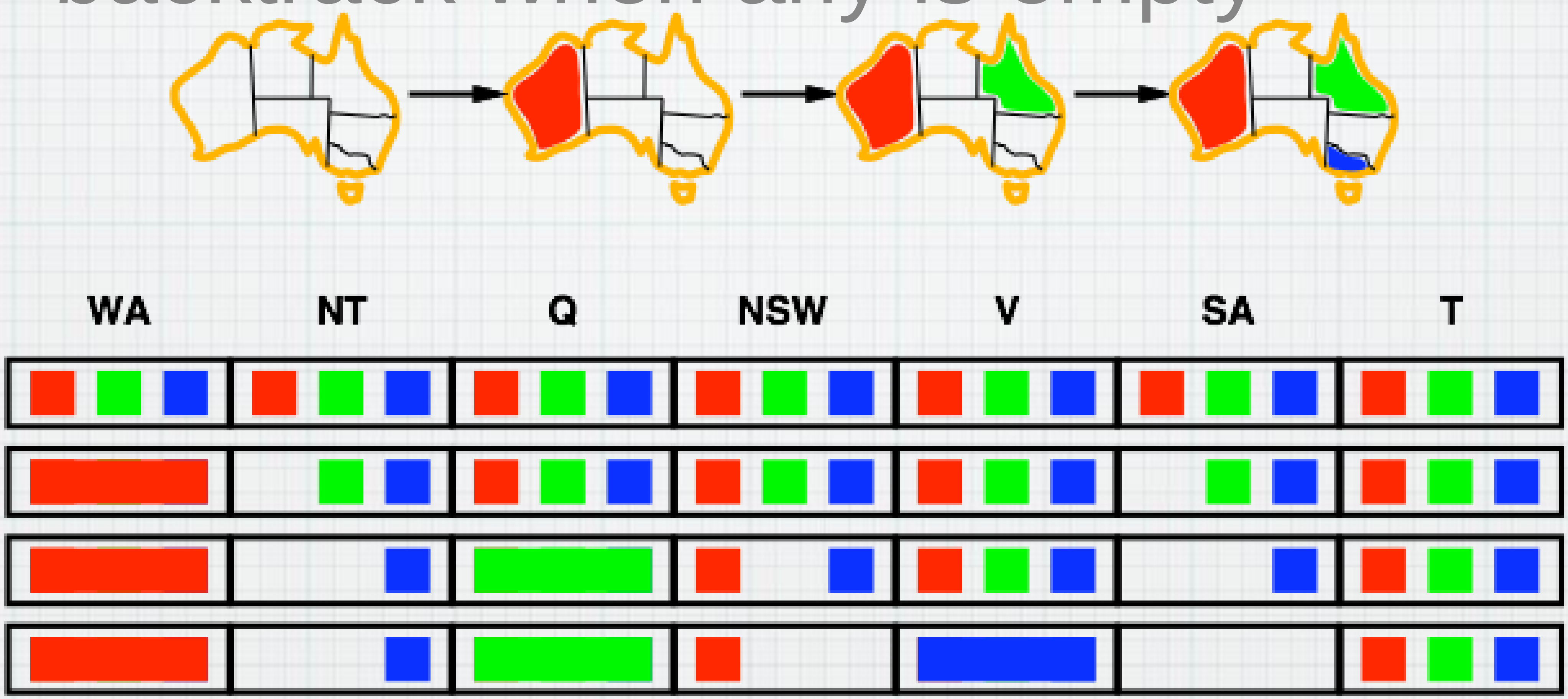
Forward Checking

* Track remaining legal vals for vars, backtrack when any is empty



Forward Checking

* Track remaining legal vals for vars, backtrack when any is empty

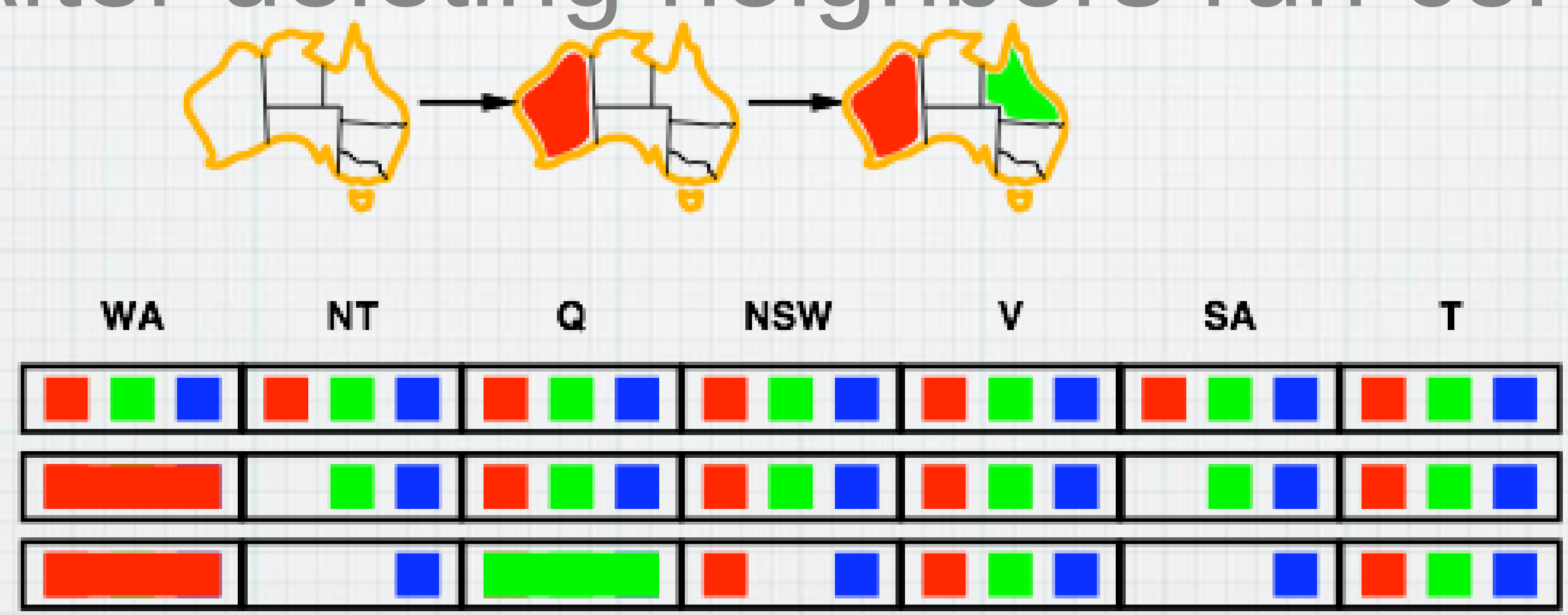


Forward Checking

* Keep track of remaining legal vals for vars, terminate and backtrack when any is empty
* MRV + Forward Checking...
    * FC is efficient way to compute info that the MRV heuristic needs

Constraint Propagation

* Can stop a branch even earlier by propagating constraints and values
* After deleting neighbors run constraints

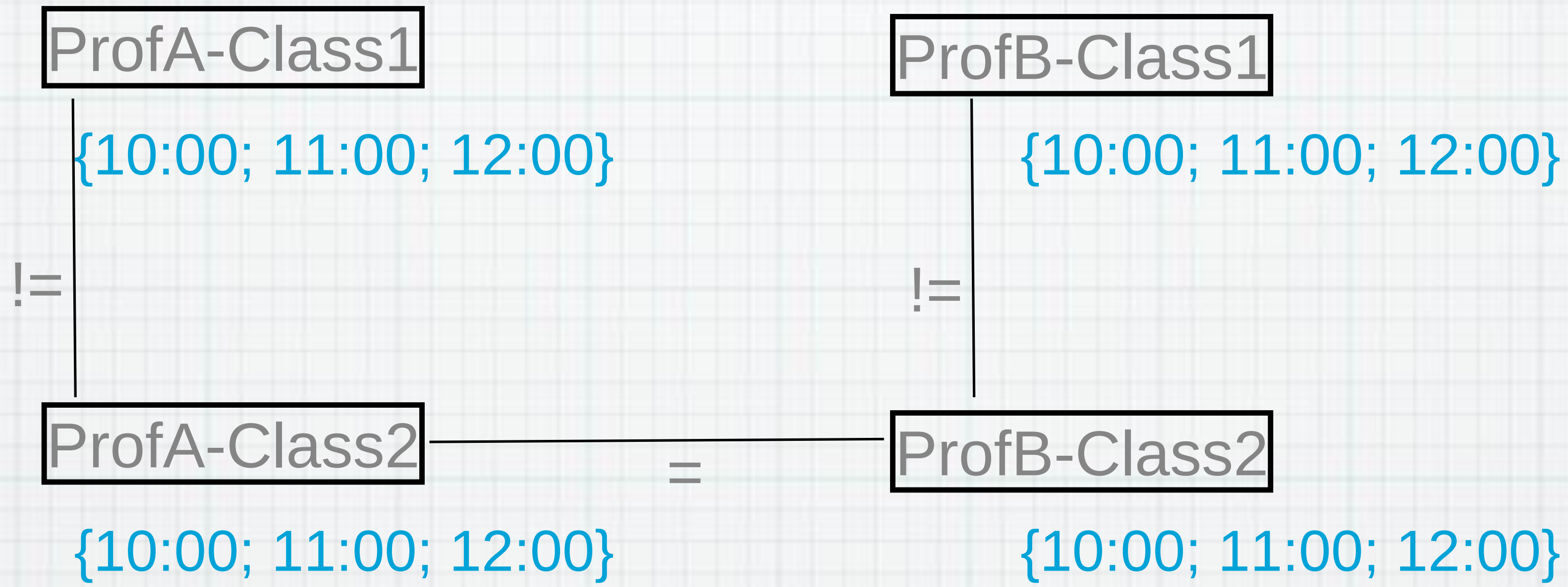


NT and SA cannot both be blue!

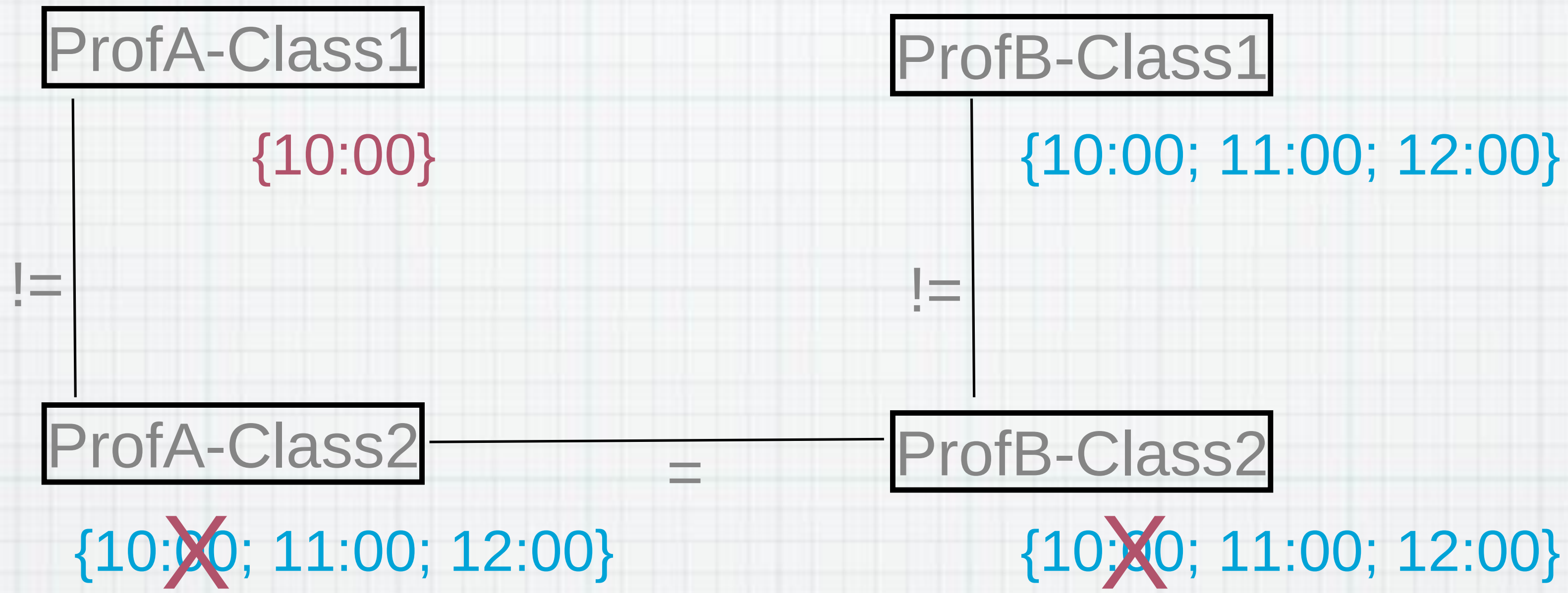
Constraint Propagation

* Use the constraints to limit the domain values of your variables
* Any time you set a variable, propagate that decision to the domains of all your other variables

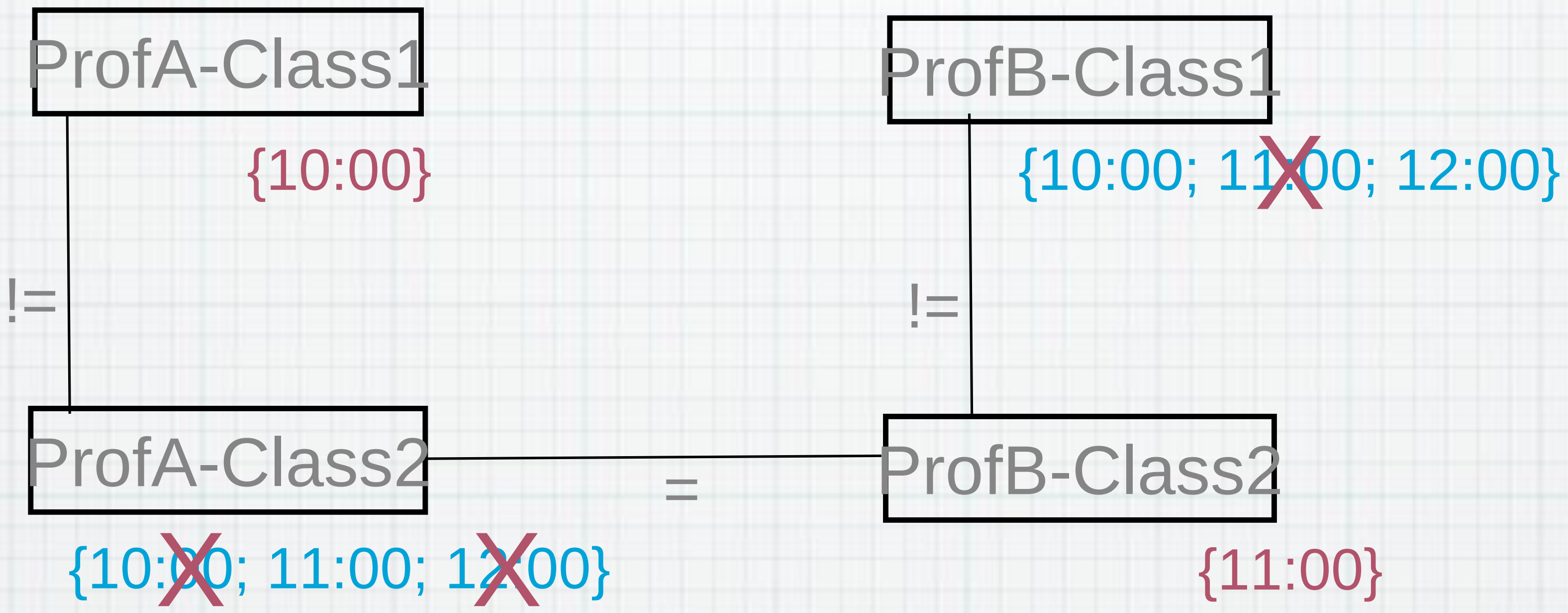
Constraint Propagation



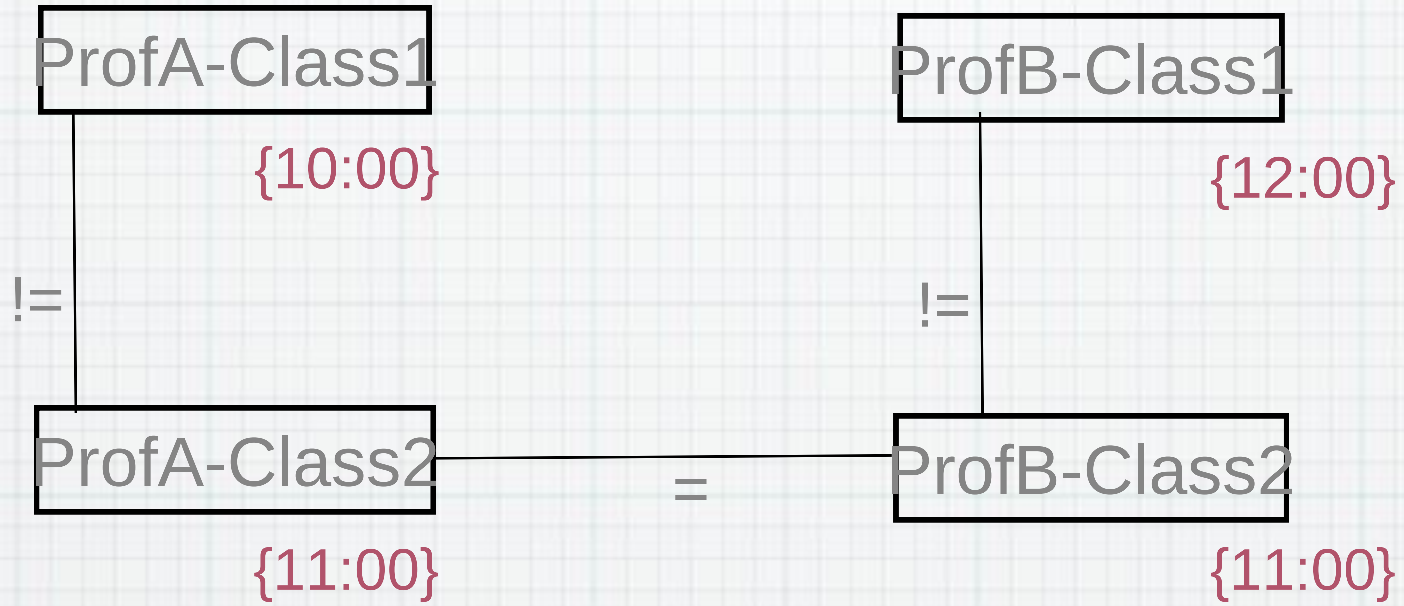
Constraint Propagation



Constraint Propagation

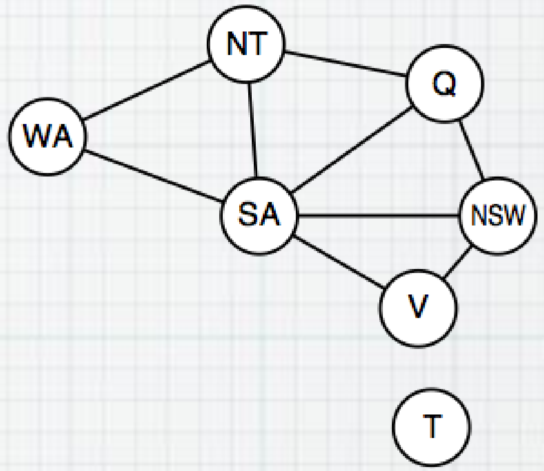


Constraint Propagation



Arc Consistency

Simplest form of propagation makes each arc consistent



$X \rightarrow Y$ is consistent iff

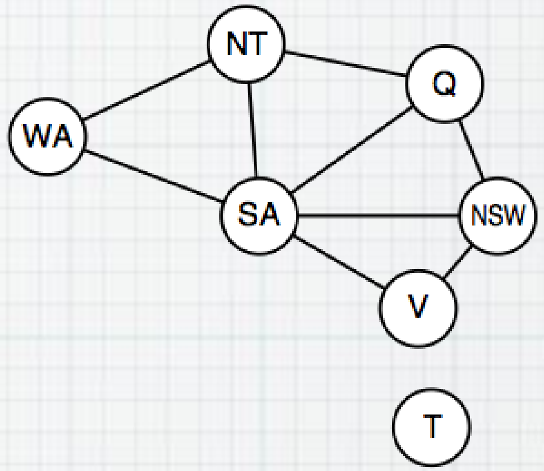
for **every** value x of X there is **some** allowed y

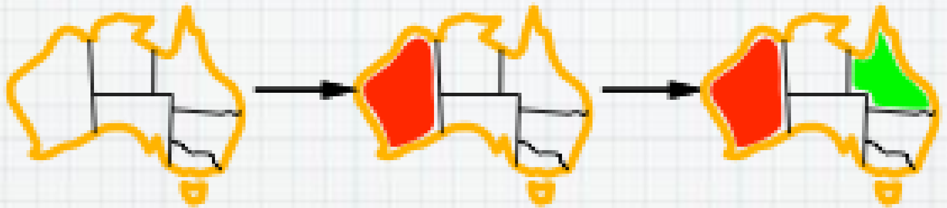
Arc Consistency

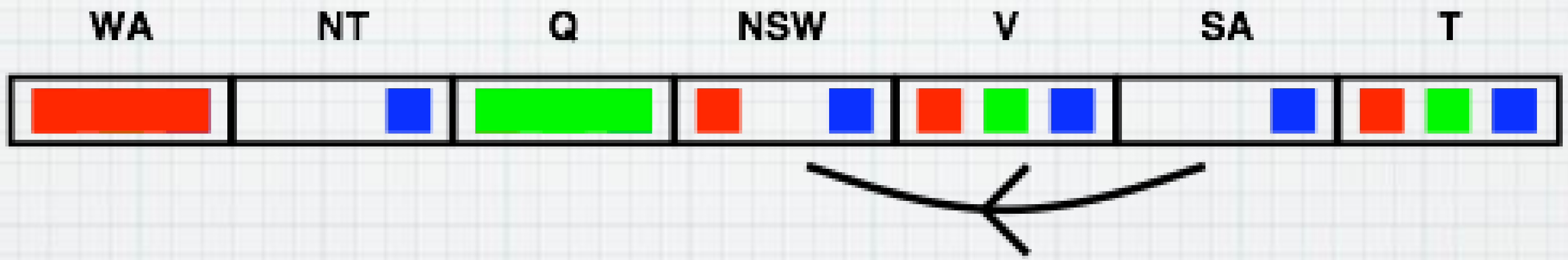
Simplest form of propagation makes each arc **consistent**

$X \to Y$ is consistent iff

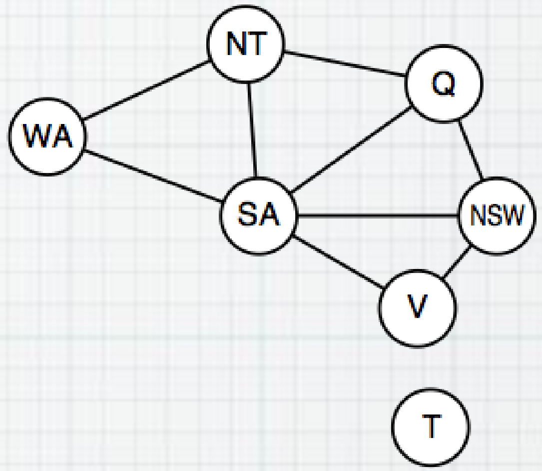
for **every** value x of X there is **some** allowed y







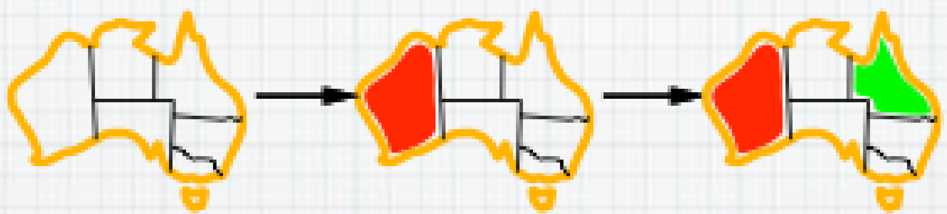
Arc Consistency

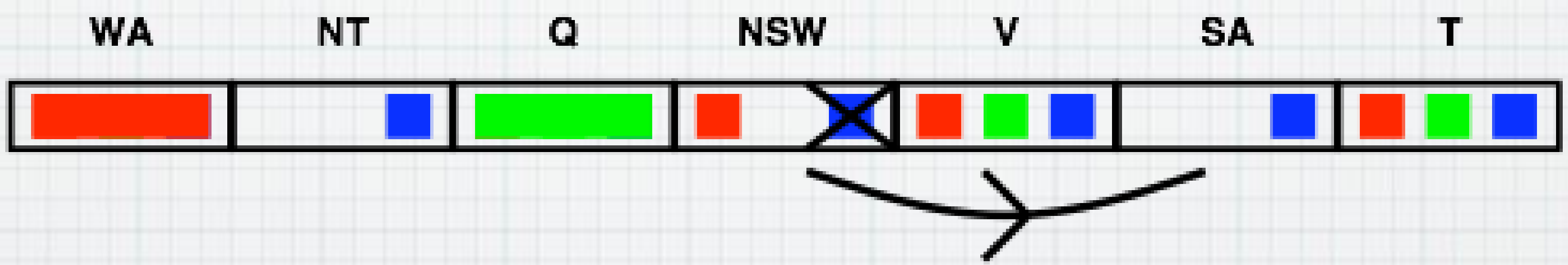


Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

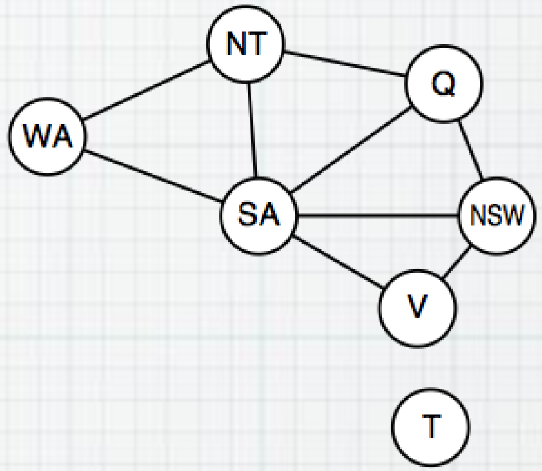
for **every** value x of X there is **some** allowed y





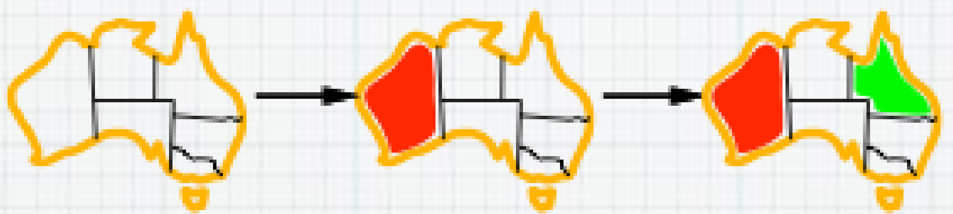
Arc Consistency

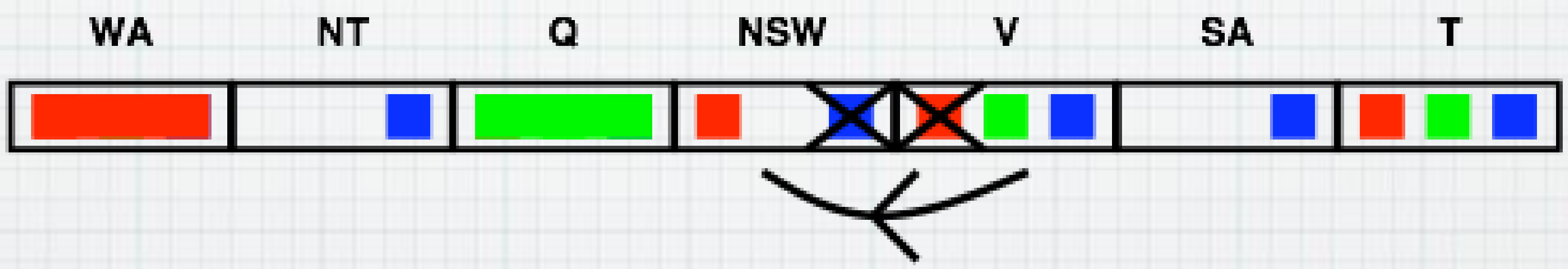
Simplest form of propagation makes each arc consistent



$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y

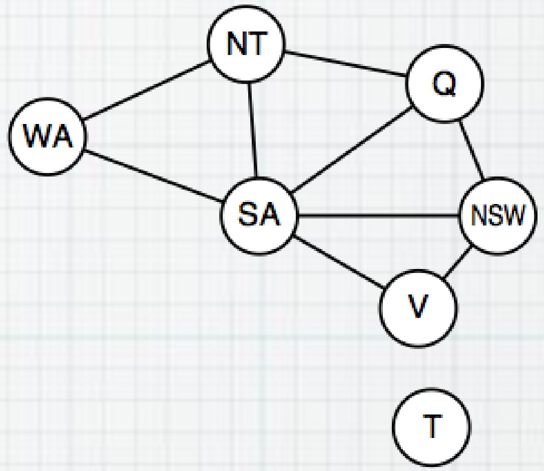




If X loses a value, neighbors of X need to be rechecked

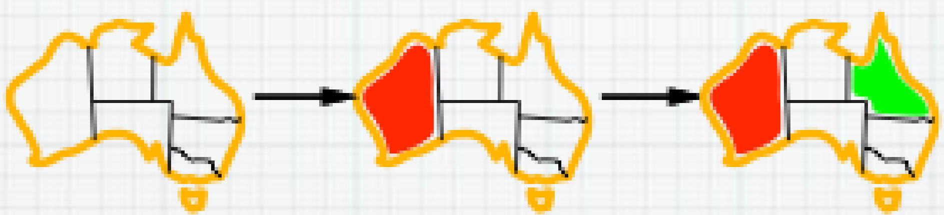
Arc Consistency

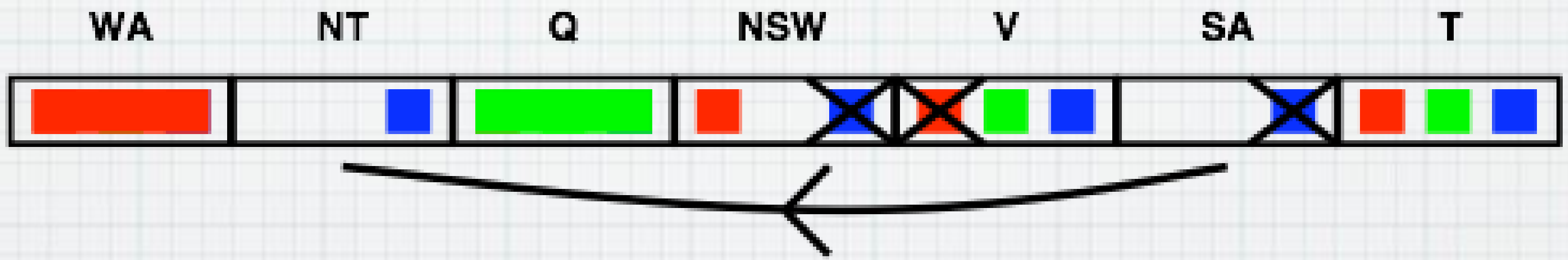
Simplest form of propagation makes each arc consistent



$X \to Y$ is consistent iff

for **every** value x of X there is **some** allowed y





If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc Consistency

```
function AC-3( csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
    local variables: queue, a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (X_i, X_j) ← REMOVE-FIRST(queue)
        if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
            then delete x from DOMAIN[X_i];  removed ← true
    return removed
```

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting **all** is NP-hard)

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

Domain:{1,2,3,4,5,6,7,8,9}

Propagate Box constraint

{1,2,3,4,5 7,8,9}

Propagate Row constraint

{ 2,3 7,8 }

Propagate Col constraint

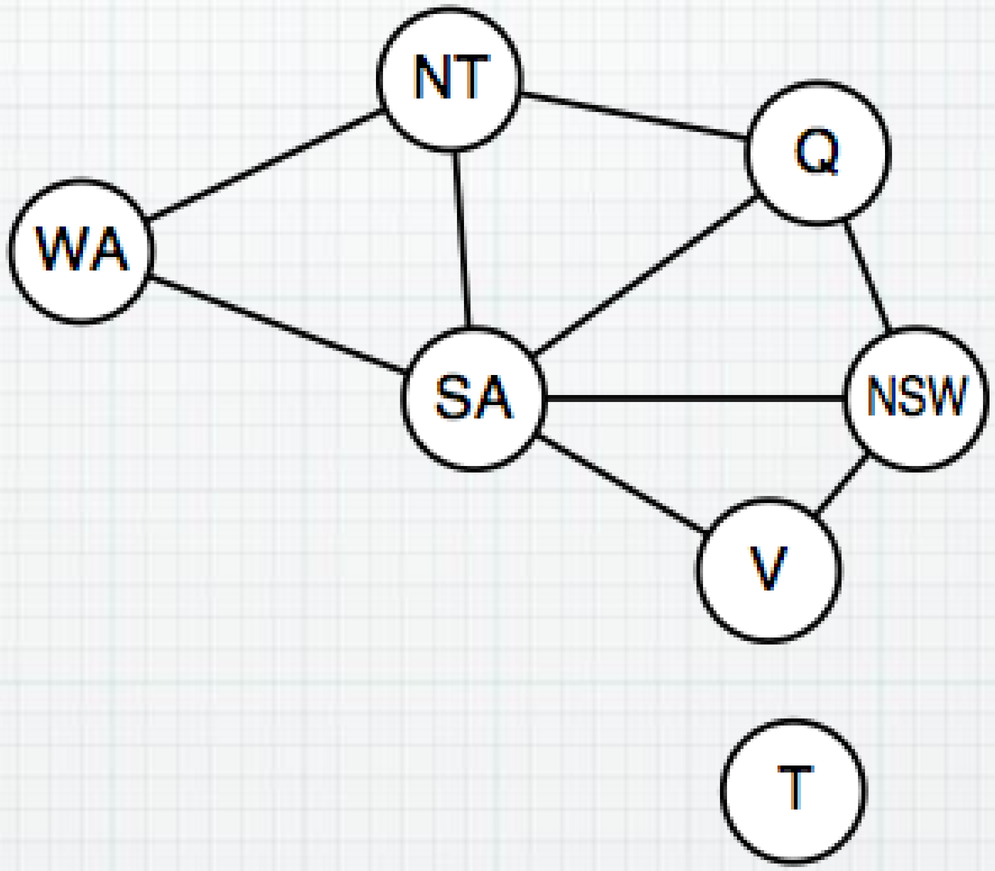
{ 2,3 7 }

For Sodoku, you just propagate constraints, setting values as their domains have only one option

Improving Backtracking

- What variable to assign next?
- What order to try the domain values?
- What inference can be made to detect failure early?
- Can we take advantage of the problem structure?

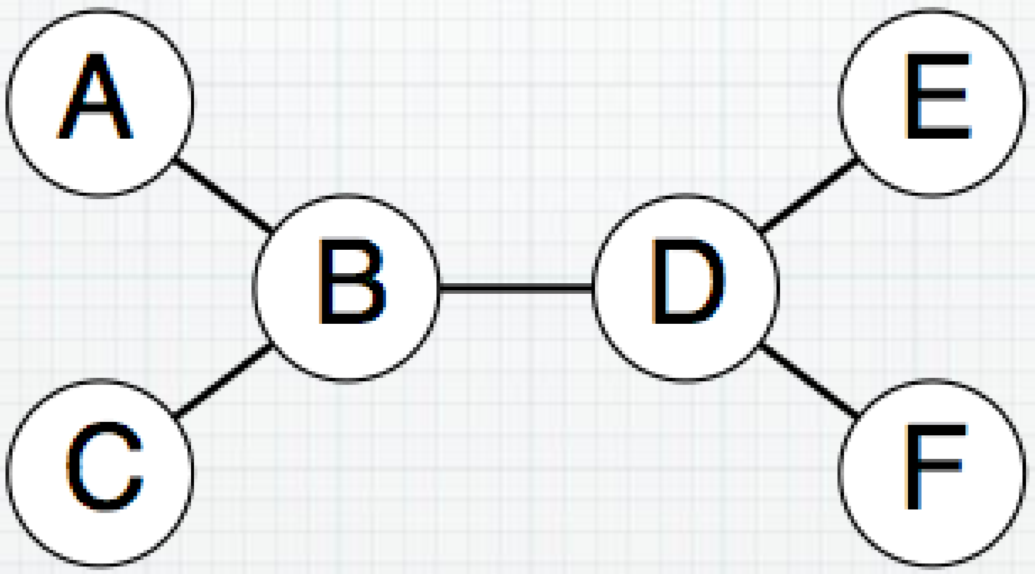
Problem Structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Tree-structured CSPs

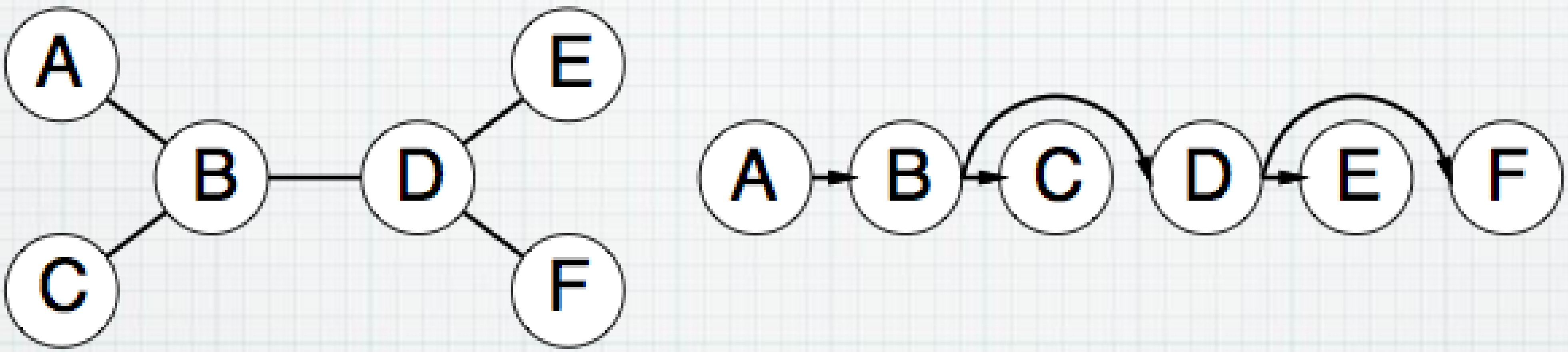


Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

Solving Tree CPSs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



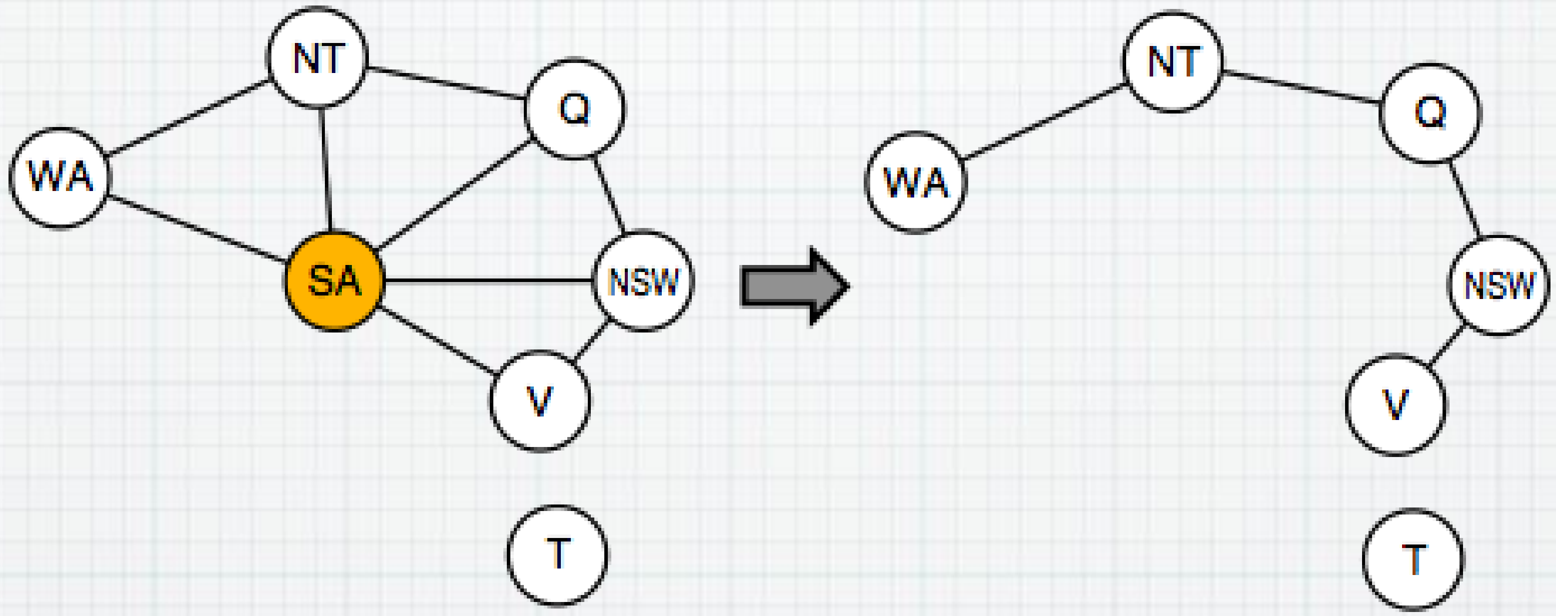
2. For j from n down to 2, apply REMOVEINCONSISTENT$(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Wow, trees are easy! Let's make everything a tree...

Nearly Trees

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c $\Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Summary

* CSP are special problem representing states=variables, goals=constraints
* Backtracking = DFS assign 1 var/node
* Variable ordering and Value selection heuristics help significantly
* Constraint propagation prevents assignments that lead to later failure
* Problem structure helps, Tree CSPs can be solved in linear time.