# Constraint Satisfaction Problems

## Chapter 5

# Constraint satisfaction problems (CSPs)

Standard search problem:
state is a "black box"—any old data structure
that supports goal test, eval, successor

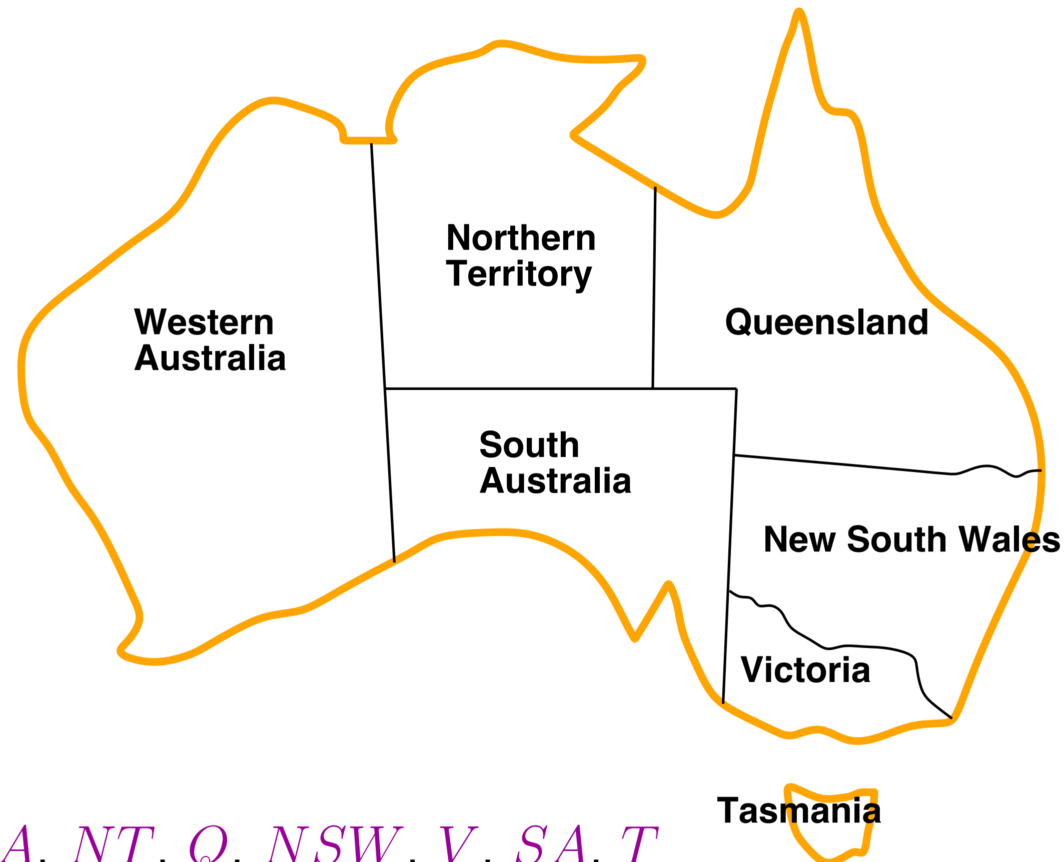
CSP:
state is defined by variables $X_i$ with values from domain $D_i$

goal test is a set of constraints specifying
allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power
than standard search algorithms

# Example: Map-Coloring




Variables $WA$, $NT$, $Q$, $NSW$, $V$, $SA$, $T$

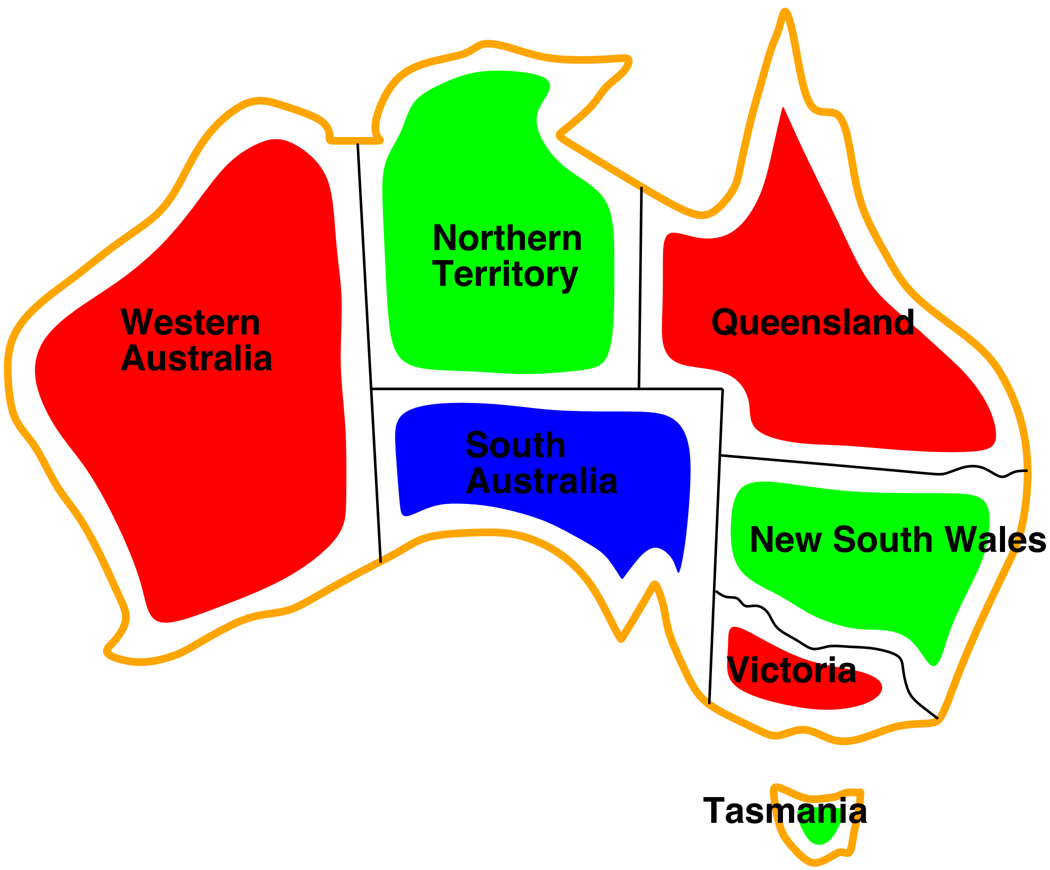
Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

# Example: Map-Coloring contd.




Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

# Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

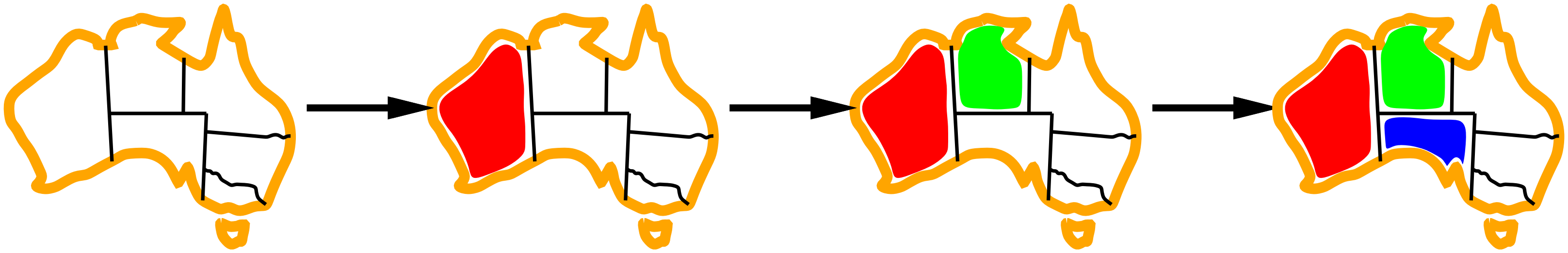
# Improving backtracking efficiency

**General-purpose** methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

# Minimum remaining values

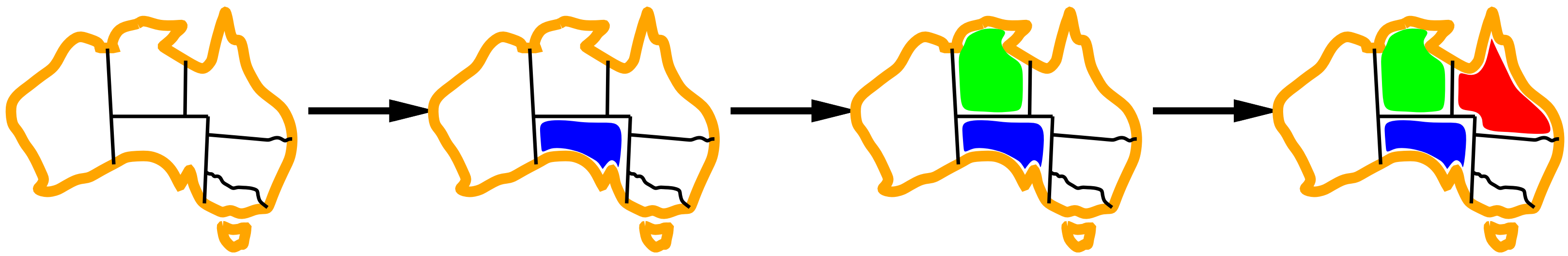
Minimum remaining values (MRV):
choose the variable with the fewest legal values



# Degree heuristic

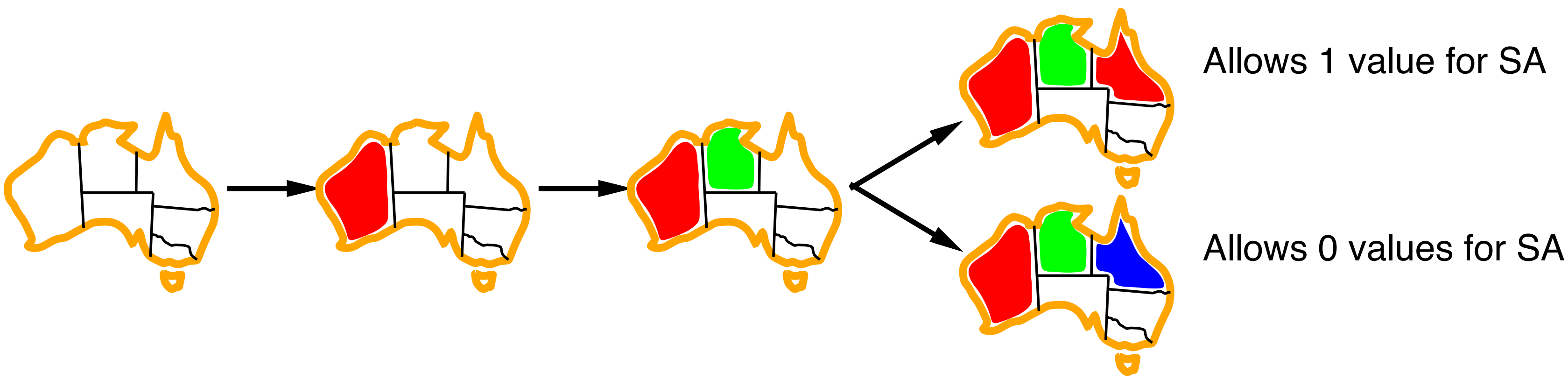
Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables



# Least constraining value

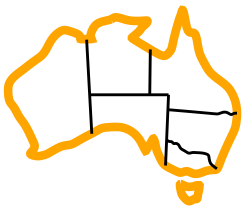
Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables




Combining these heuristics makes 1000 queens feasible

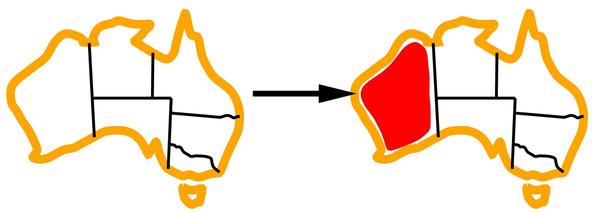
# Forward checking

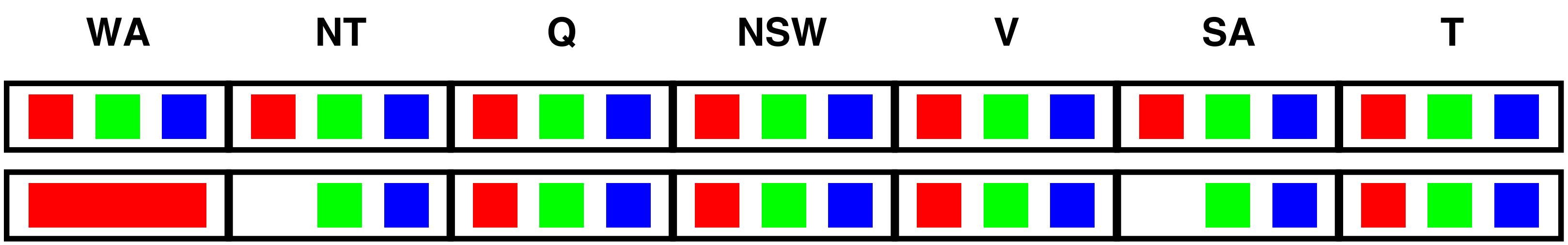
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



# Forward checking

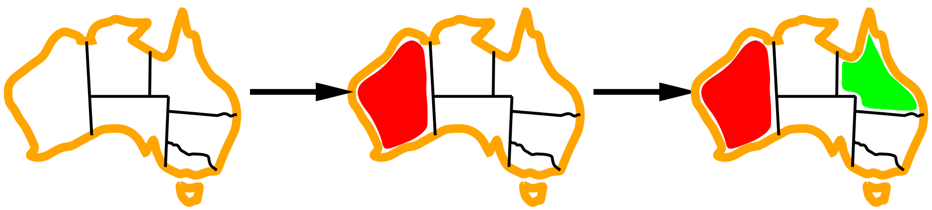
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





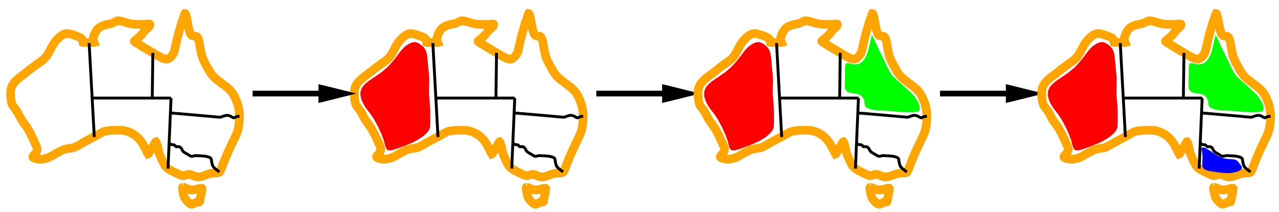
# Forward checking

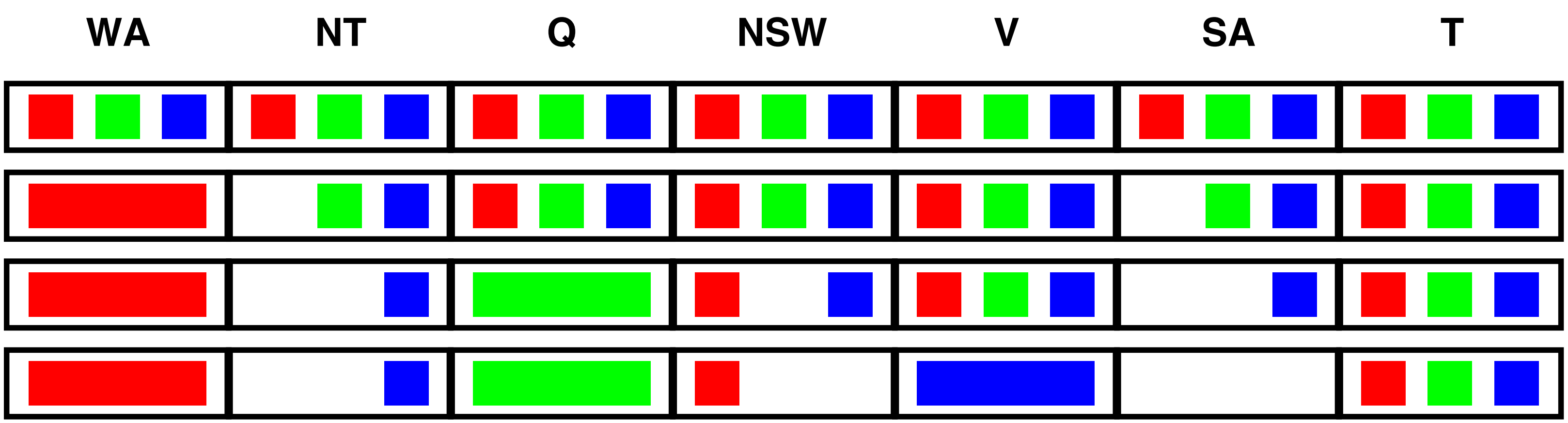
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



# Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



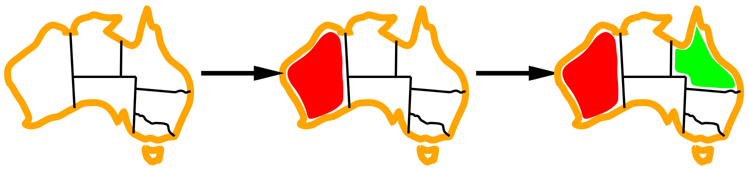


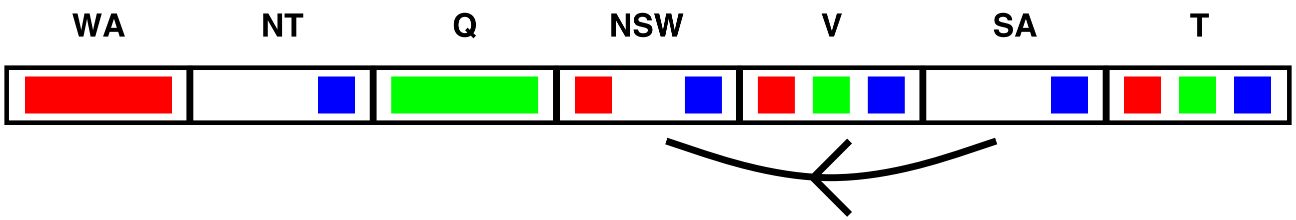
# Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

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



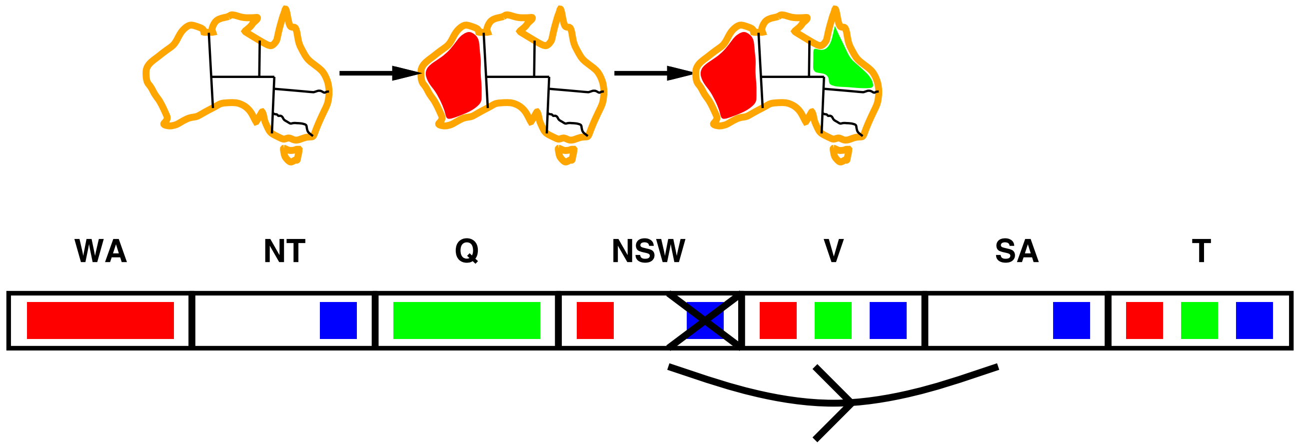


# Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

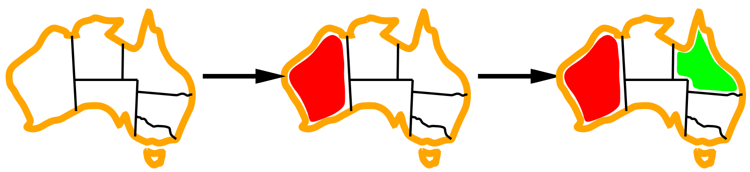
for **every** value $x$ of $X$ there is **some** allowed $y$

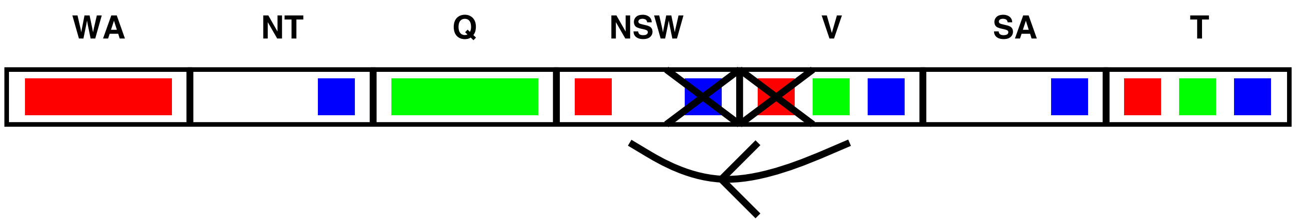


# Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
for **every** value $x$ of $X$ there is **some** allowed $y$



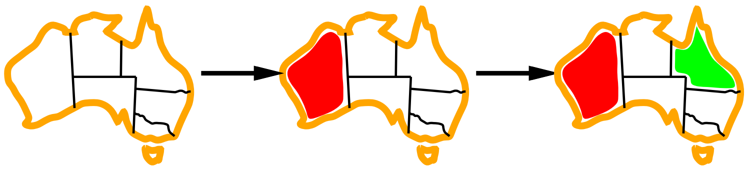


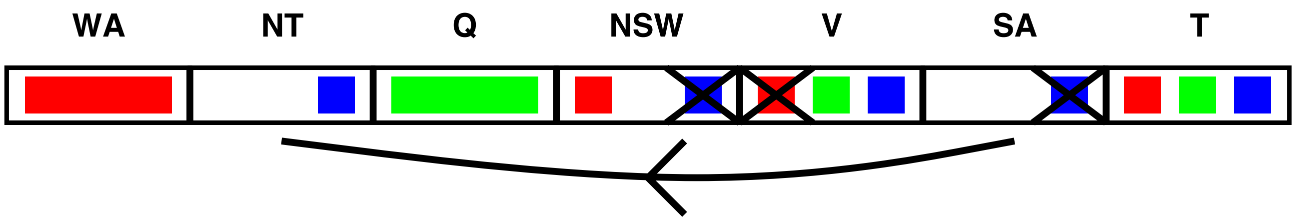

If $X$ loses a value, neighbors of $X$ need to be rechecked

# 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 algorithm

```
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)

# Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with
"complete" states, i.e., all variables assigned

To apply to CSPs:
allow states with unsatisfied constraints
operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by min-conflicts heuristic:
choose value that violates the fewest constraints
i.e., hillclimb with $h(n) =$ total number of violated constraints

# Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n)$ = number of attacks