# CS:4420 Artificial Intelligence

## Spring 2018

# Constraint Satisfaction Problems


Cesare Tinelli

The University of Iowa

# Constraint Satisfaction Problems (CSPs)

Standard search problem:

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

CSP:

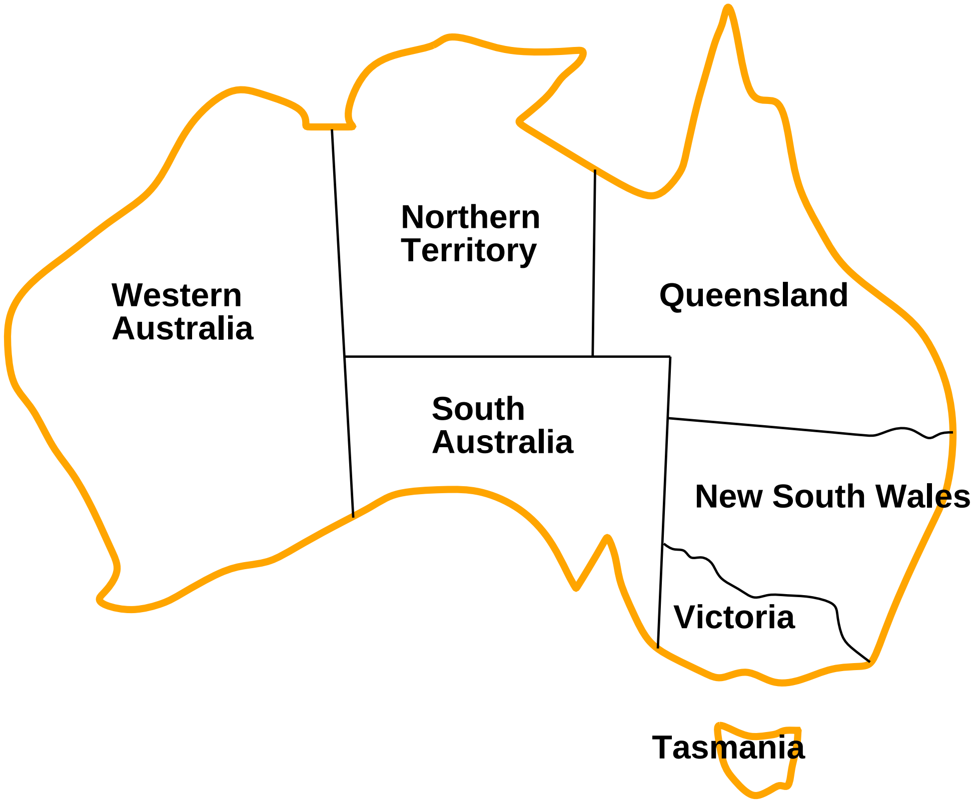
state is defined by variables $X_i$ with values from domain $D_i$

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



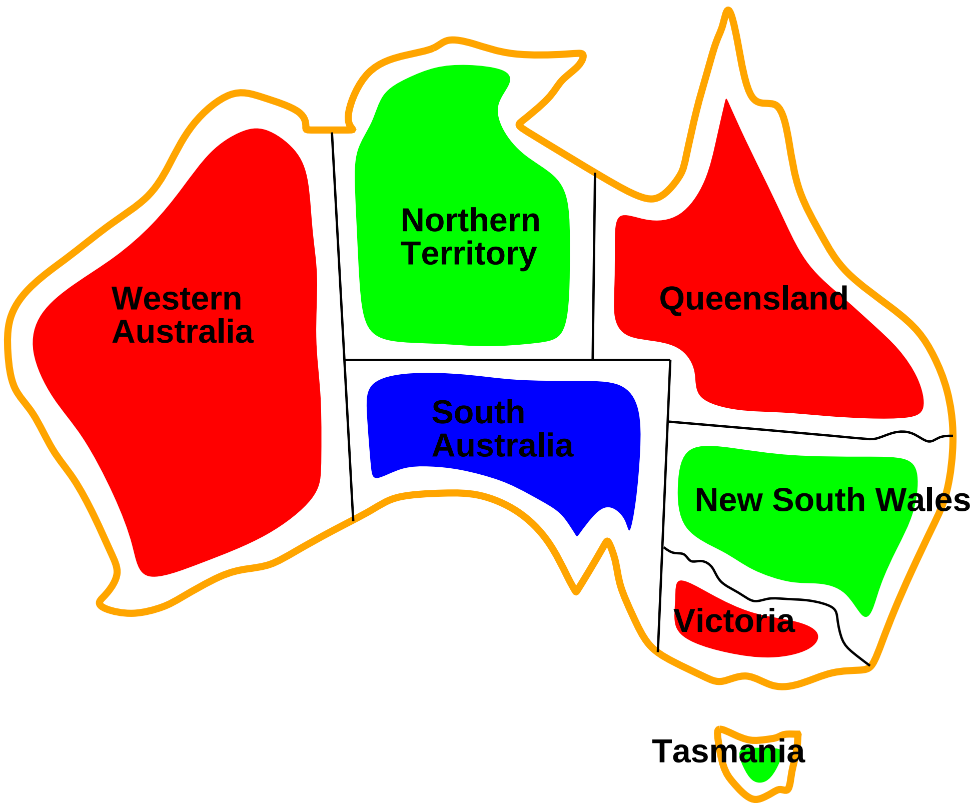

Variables: $MA$, $NT$, $Q$, $NSW$, $V$, $SA$, $T$

Domains: $D_i = \{r(ed), g(reen), b(lue)\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(r, g), (r, b), (g, r), (g, b), \ldots\}$

# Example: Map coloring contd.



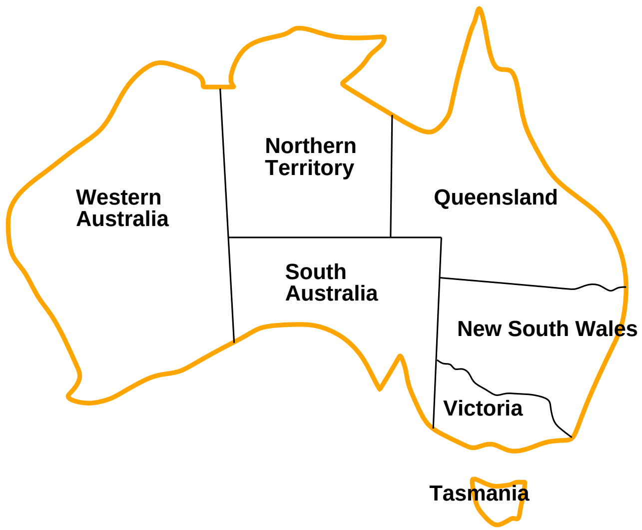

Solutions are assignments satisfying all constraints,

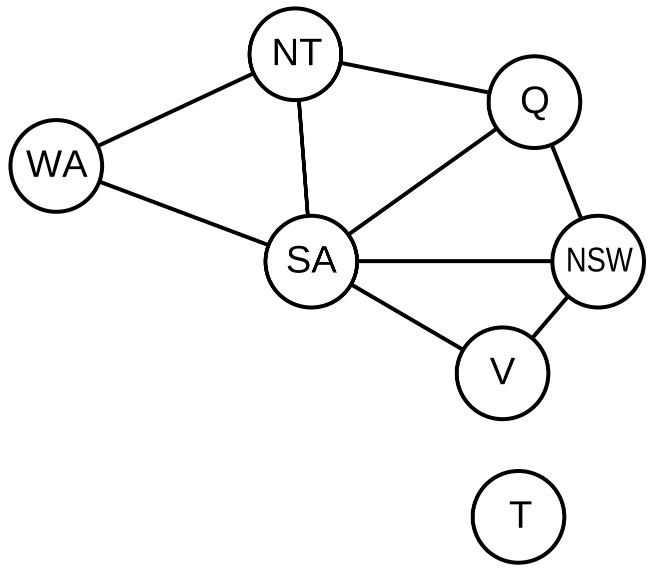
e.g., $\{WA = r, NT = g, Q = r, NSW = g, V = r, SA = b, T = g\}$

# Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints




General-purpose CSP methods use the graph structure to speed up search

e.g., Tasmania is an independent subproblem!

# Varieties of CSPs

Discrete variables

finite domains (size $d$)

- e.g., Boolean CSPs, incl. Boolean SAT (NP-complete)
- $O(d^n)$ complete assignments

infinite domains (integers, strings, etc.)

- e.g., job scheduling, variables are start/end days for each job
- need a constraint language, e.g., $startJob_1 + 5 \leq startJob_3$
- linear constraints solvable, nonlinear undecidable

Continuous variables

- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in polynolmial time by linear programming methods

# Varieties of constraints

Unary constraints involve a single variable
e.g., $SA \neq g$

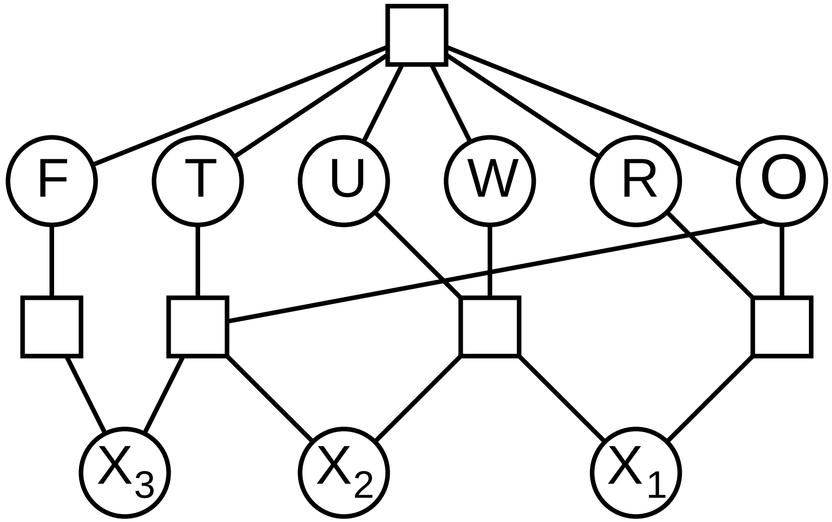
Binary constraints involve pairs of variables
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables
e.g., cryptarithmetic column constraints

Preferences are soft constraints
e.g., $red$ is better than $green$
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems

# Example: Cryptarithmetic

$$\begin{array}{rcccc} & & T & W & O \\ + & & T & W & O \\ \hline F & & O & U & R \end{array}$$




Variables: $F,\ T,\ U,\ W,\ R,\ O,\ X_1,\ X_2,\ X_3$

Domain: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints: $\mathit{alldiff}\,(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$

$\ldots$

# Real-world CSPs

Assignment problems
e.g., who teaches what class

Timetabling problems
e.g., which class is offered when and where?

Hardware configuration

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

# Standard search formulation (incremental)

Let's start with a basic, naive approach and then improve it

States are defined by the values assigned so far

Initial state: the empty assignment, $\{\,\}$

Successor function: assign a value to an unassigned variable that does not conflict with current assignment.
fail if no legal assignments (not fixable!)

Goal test: the current assignment is complete

Note:

1. This is the same for all CSPs!
2. Every solution appears at depth $n$ with $n$ variables $\Longrightarrow$ use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. However, with domain of size $d$, branching factor $b = (n - \ell)d$ at depth $\ell$, hence $n!d^n$ leaves!

# Backtracking search

Variable assignments are commutative
i.e., $[WA = r$ then $NT = g]$ same as $[NT = g$ then $WA = r]$

Only need to consider assignments to a single variable at each node
$\Longrightarrow$ $b = d$ and there are $d^n$ leaves

Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

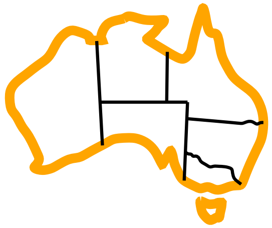
Can solve $n$-queens for $n \approx 25$

# Backtracking search

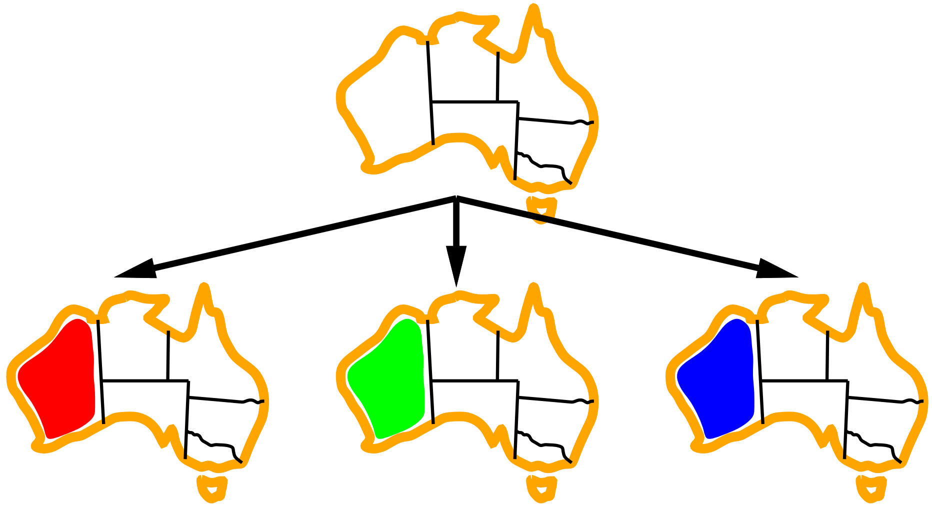
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING([ ], csp)

function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
    if assigned is complete then return assigned
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
        if value is consistent with assigned according to CONSTRAINTS[csp] then
            result ← RECURSIVE-BACKTRACKING([var = value|assigned], csp)
            if result ≠ failure then return result
    end
    return failure
```

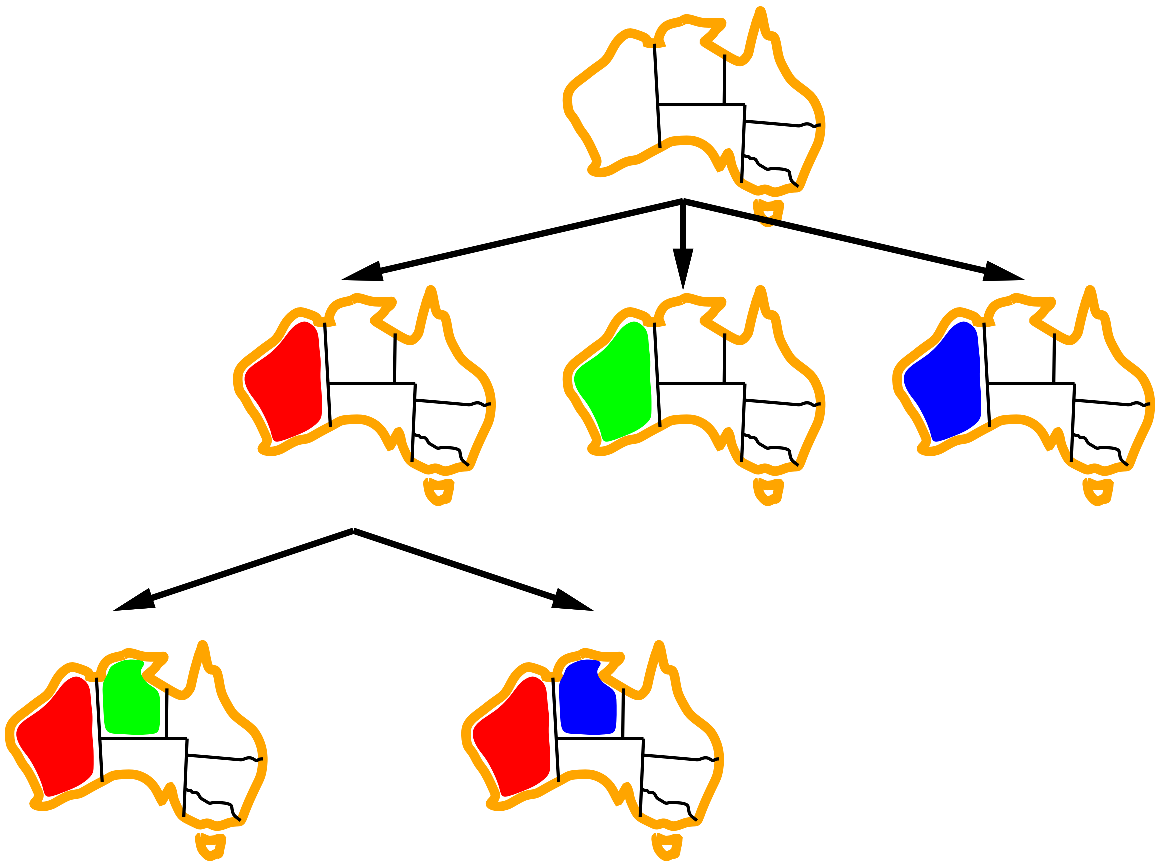
# Backtracking example



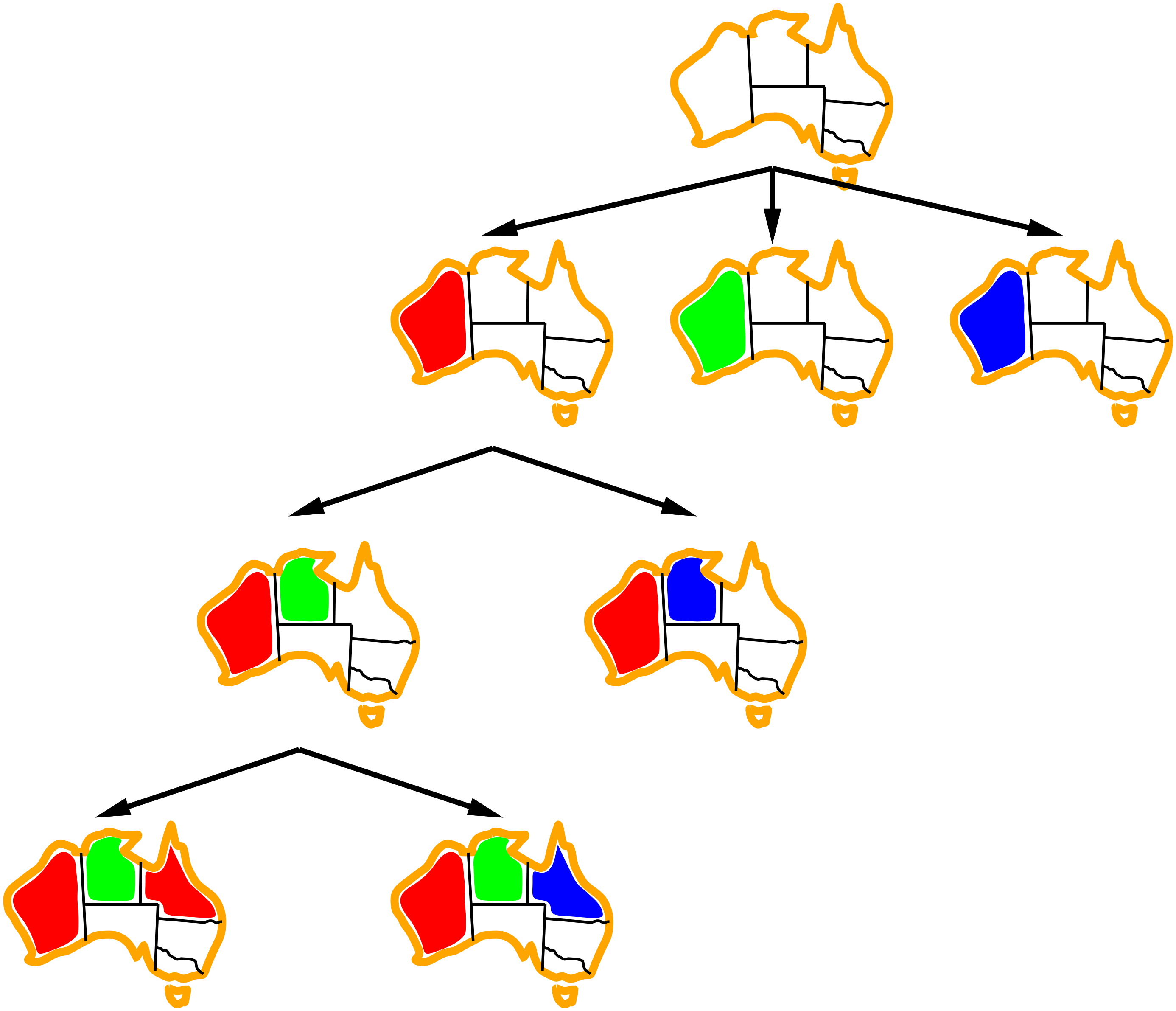
# Backtracking example



# Backtracking example



# Backtracking example



# Improving backtracking efficiency

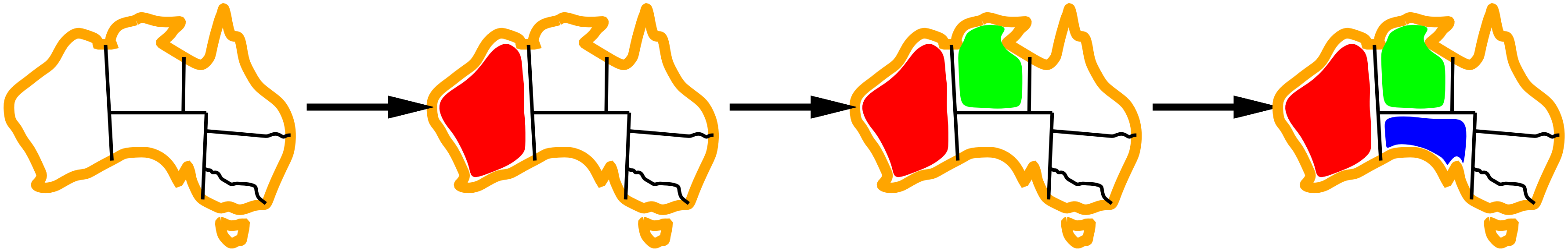
General-purpose methods can give huge gains in speed:

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?

# Variable choice heuristics

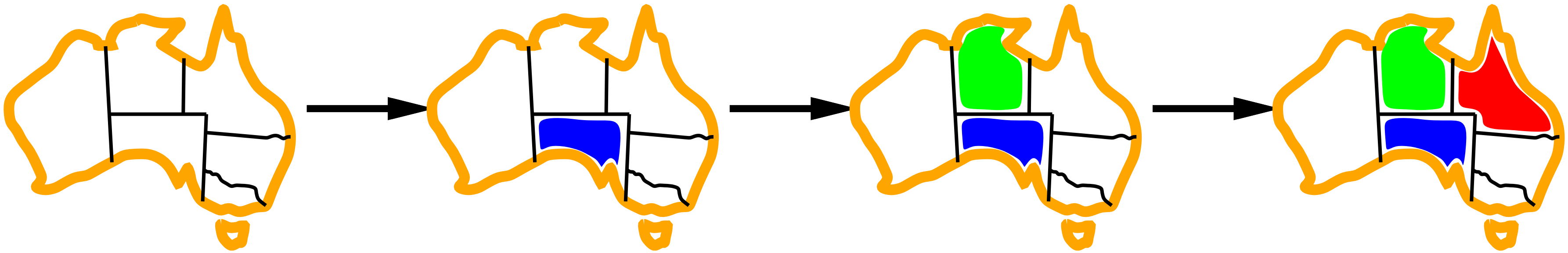
Minimum remaining values (MRV):

choose the variable with the fewest legal values



Degree heuristic:

choose the variable with the most constraints on remaining vars

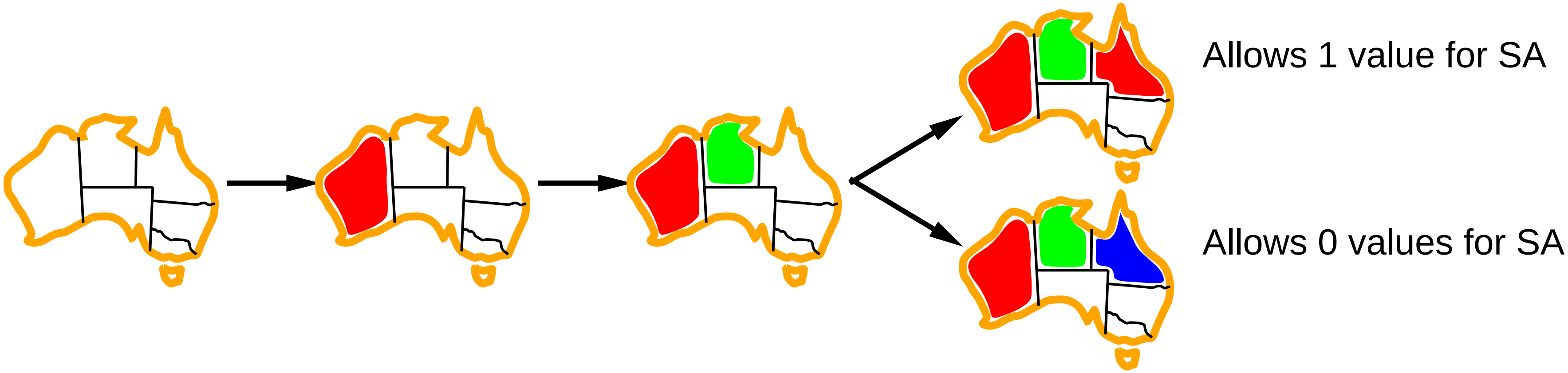


Latter ofter used as a tie-breaker for former

# Value choice heuristics

Least constraining value:

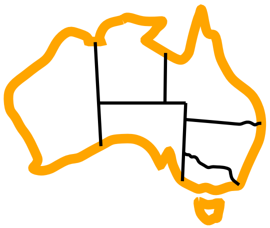
- for a given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables




Combining these heuristics makes 1000-queens feasible

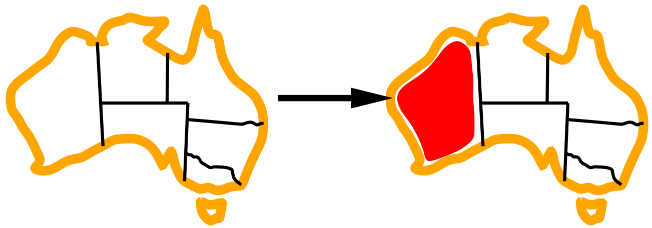
# Forward checking

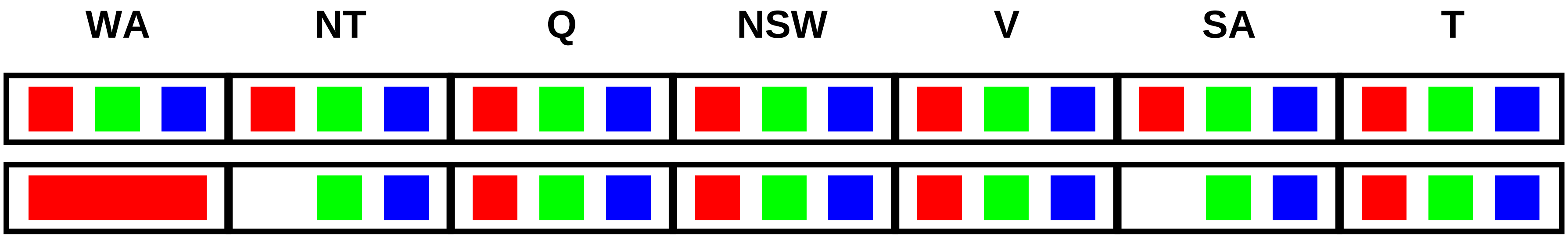
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



# Forward checking

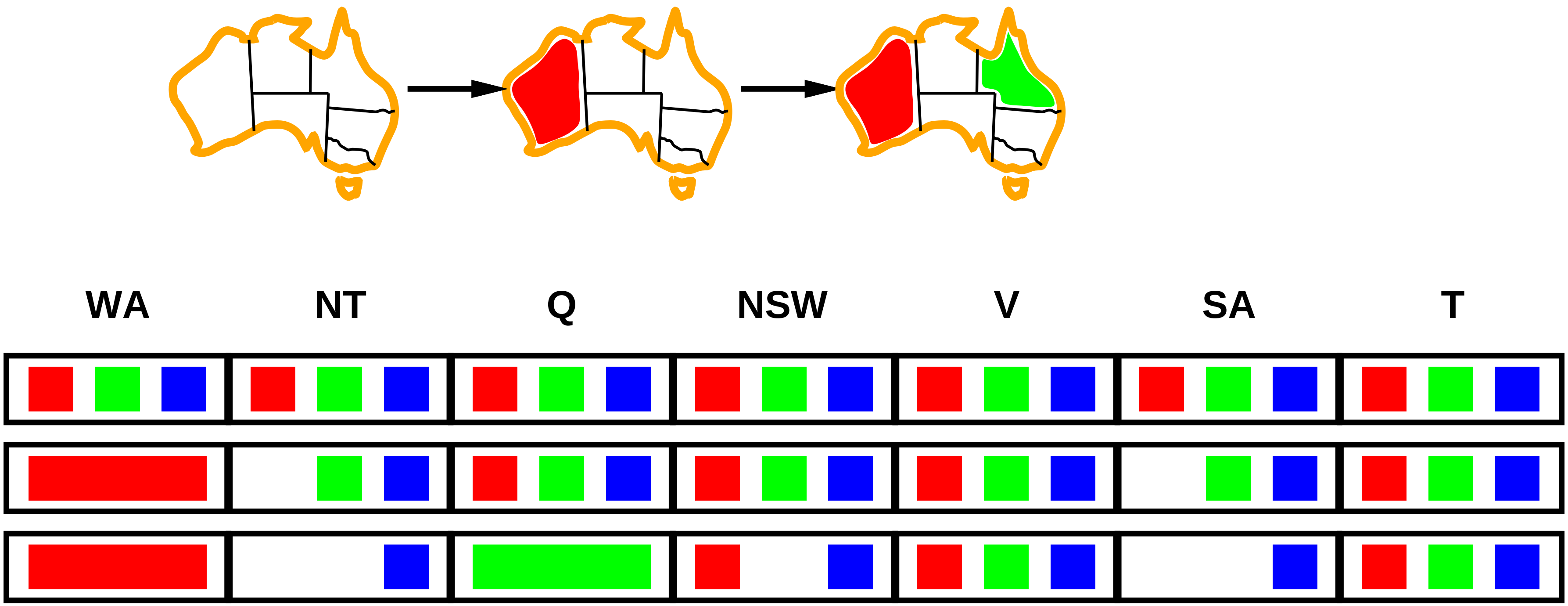
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





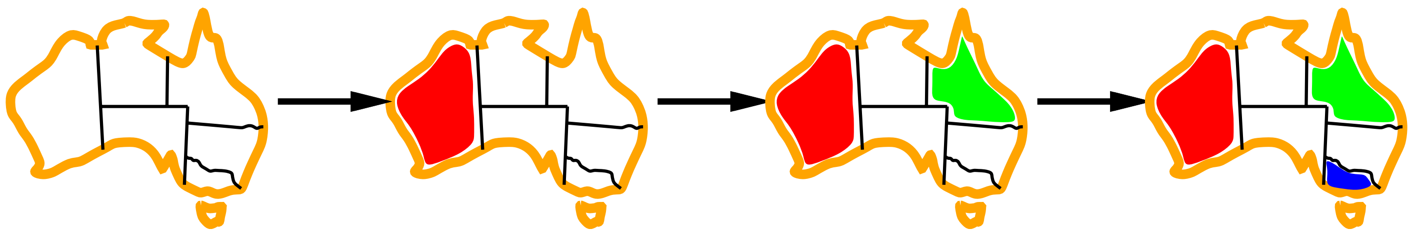
# Forward checking

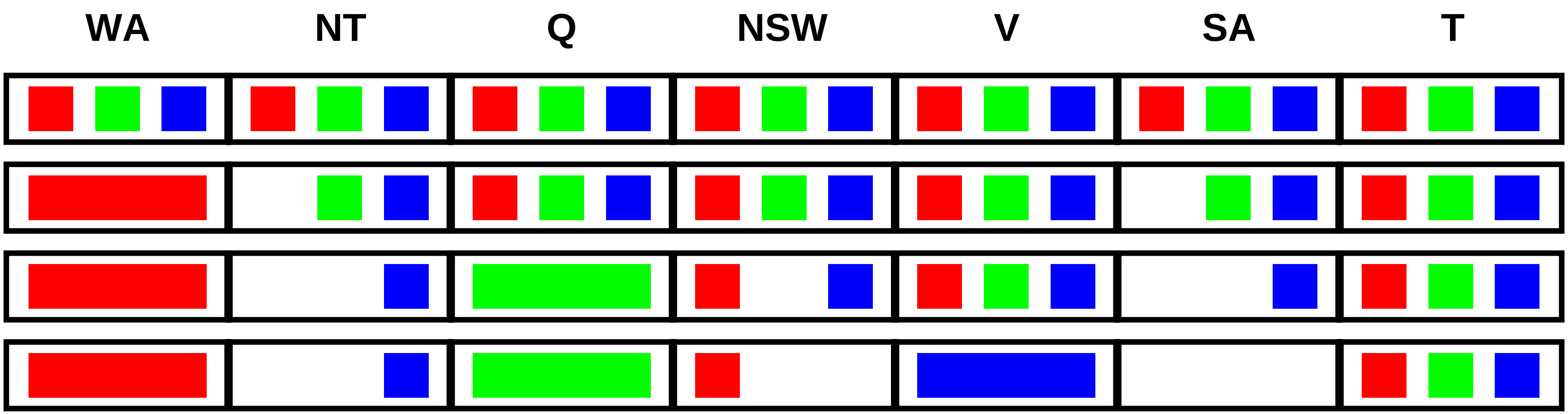
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



# Forward checking

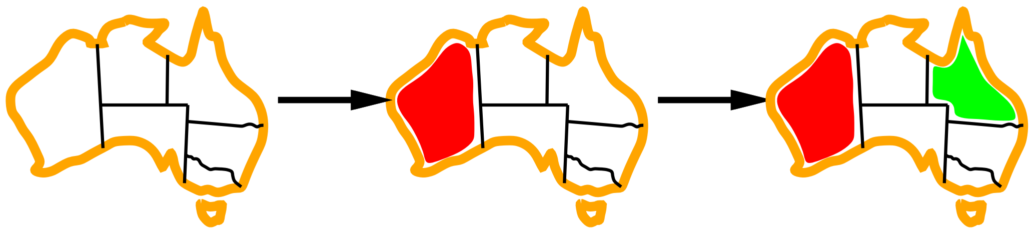
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





# Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:




$NT$ and $SA$ cannot both be blue!

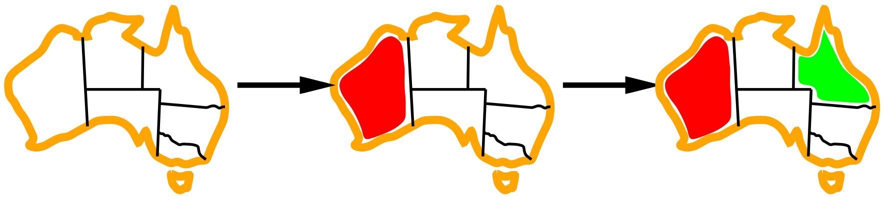
Constraint propagation repeatedly enforces constraints locally

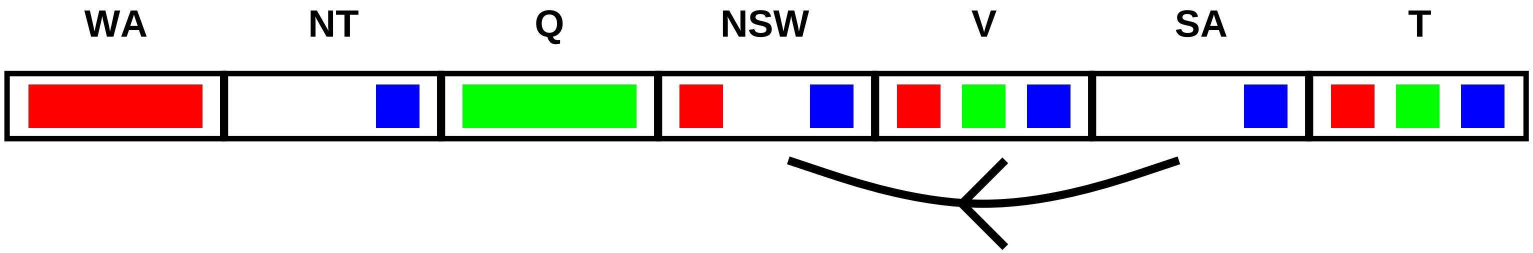
# Arc consistency

Simplest form of propagation, makes each arc consistent

Arc $X \rightarrow Y$ is consistent iff

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



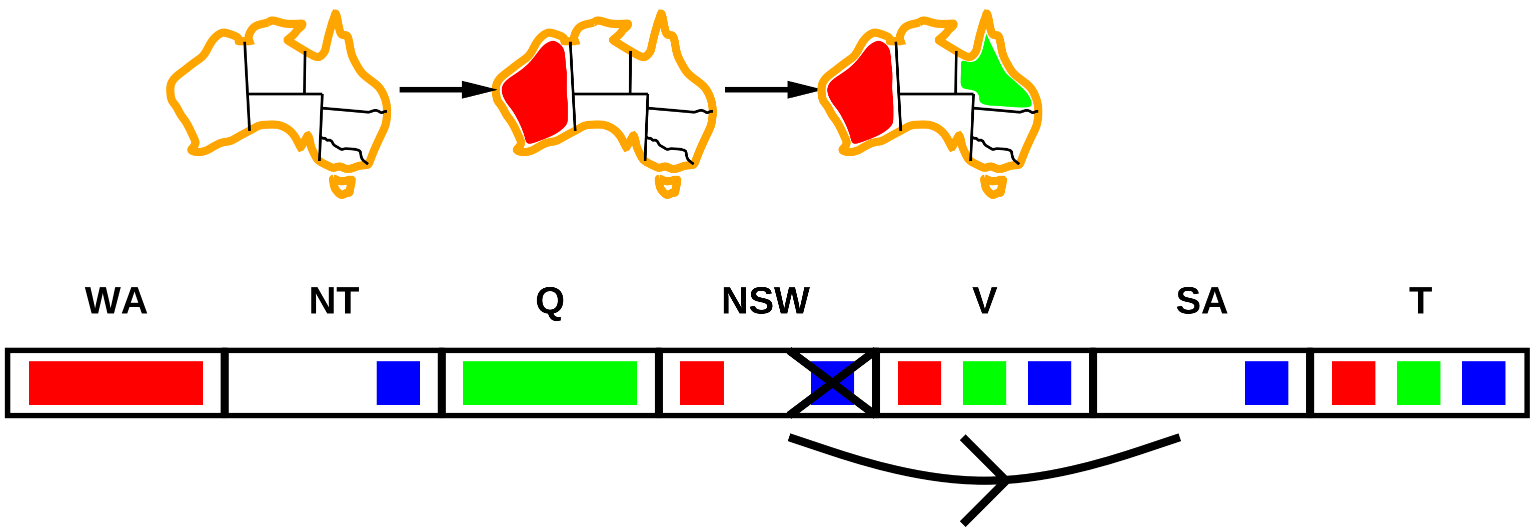


# Arc consistency

Simplest form of propagation, makes each arc consistent

Arc $X \rightarrow Y$ is consistent iff

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

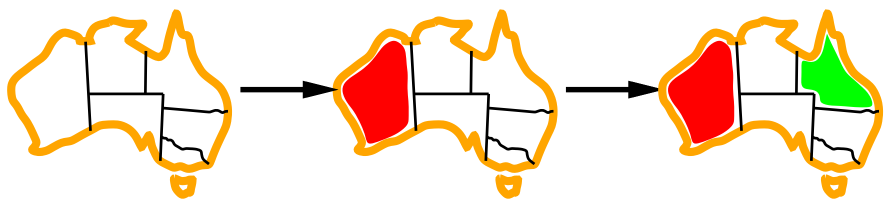


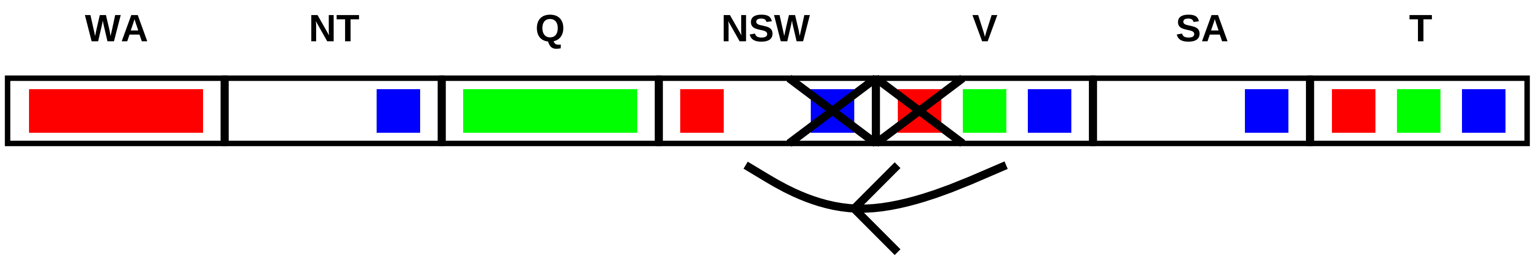
# Arc consistency

Simplest form of propagation, makes each arc consistent

Arc $X \to Y$ is consistent iff

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





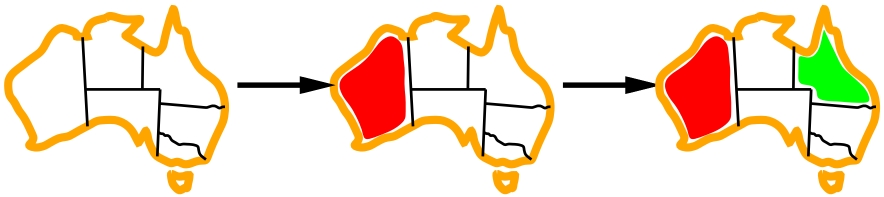

If $X$ loses a value, neighbors of $X$ need to be rechecked

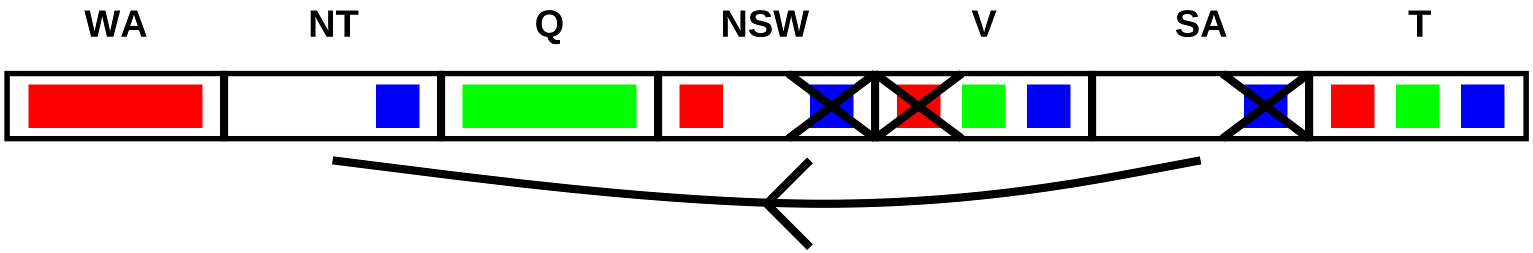
# Arc consistency

Simplest form of propagation, makes each arc consistent

Arc $X \rightarrow Y$ is consistent iff

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






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 and/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 vars: 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 we remove a value
    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 between X_i and X_j
            then delete x from DOMAIN[X_i]; removed ← true
    return removed
```

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

# Further notions of consistency

Node consistency: A single variable $X$ is node-consistent if all the values in $X$'s domain $D(X)$ satisfy the unary constraints on $X$

Ex.

$D(X) = \{1, 2, 3\} \quad C_1 = (X > 0) \quad X$ node-consist. with $C_1$

$D(X) = \{1, 2, 3\} \quad C_2 = (X > 5) \quad X$ not node-consist. with $C_2$

# Further notions of consistency

Arc-consistency for $n$-constraints

Generalized arc consistency: A variable $X_i$ is generalized arc-consistent wrt an $n$-ary constraint $C(X_1, \ldots, X_i, \ldots, X_n)$ if,
for every $v \in D(X_i)$,
there is a $(v_1, \ldots, v, \ldots, v_n) \in D(X_1) \times \cdots \times D(X_i) \times \cdots \times D(X_n)$
that satisfies $C$

Ex.

$D(X) = D(Y) = D(Z) = \{1, 2, 3\}$

$C_1 = (X + Y > Z)$ $\quad$ $Y$ generalized arc-consist. with $C_1$

$C_2 = (X + Y < Z)$ $\quad$ $Z$ not generalized arc-consist. with $C_2$

# Further notions of consistency

*Chained* arc-consistency

Path consistency: A two-variable set $\{X, Z\}$ is path-consistent wrt a third variable $Y$ if,
for every assignment satisfying the constraints on $\{X, Z\}$,
there is an assignment to $Y$ that satisfies the constraints on $\{X, Y\}$ and $\{Y, Z\}$

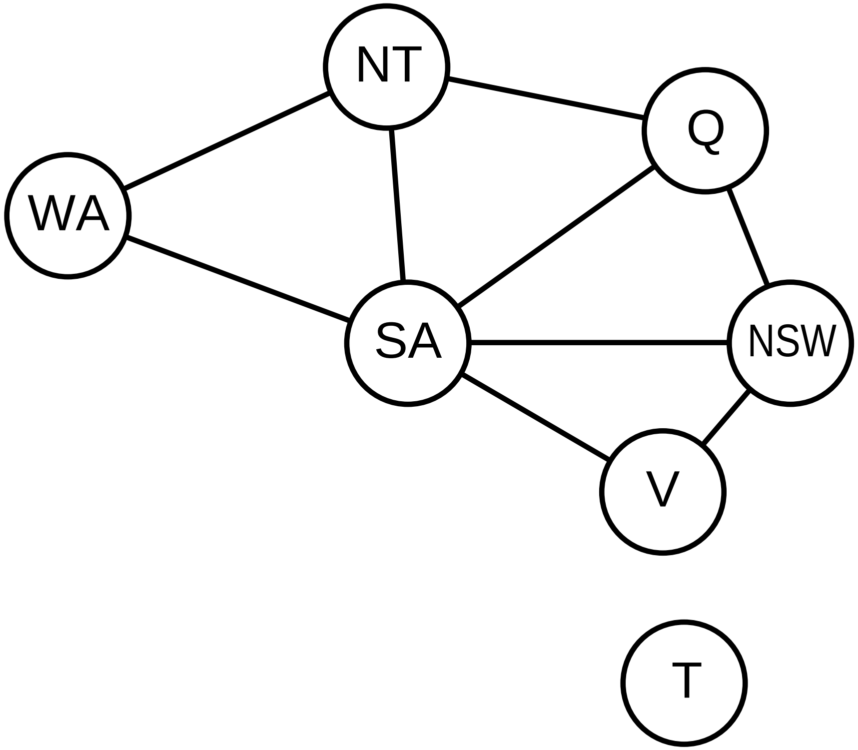
Ex.

$D(X) = D(Y) = D(Z) = \{1, 2, 3, 4\}$

$\{X > 2 \cdot Z,\ X > Y,\ Y = Z + 1\}$ $\{X, Z\}$ path-consistent wrt $Y$

$\{X > 2 \cdot Z,\ X < Y,\ Y = Z + 1\}$ not $\{X, Z\}$ path-consistent wrt $Y$

# Problem structure




Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

# Problem structure

Suppose each subproblem has $c$ variables out of $n$ total

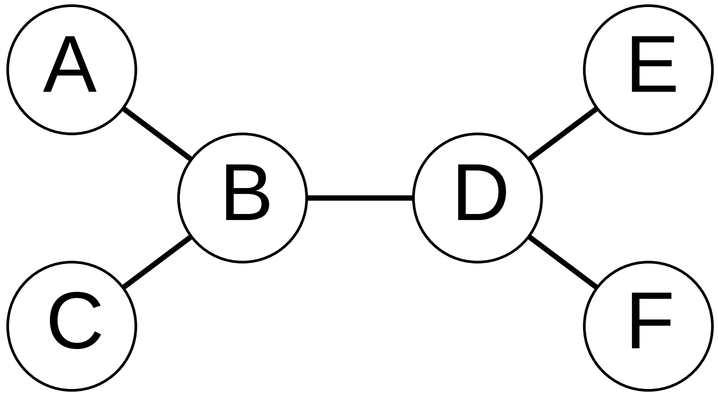
Worst-case solution cost is $n/c \cdot d^c$, linear in $n$

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80}$ = 4 billion years at 10 million nodes/sec

$4 \cdot 2^{20}$ = 0.4 seconds at 10 million nodes/sec

# Tree-structured CSPs




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

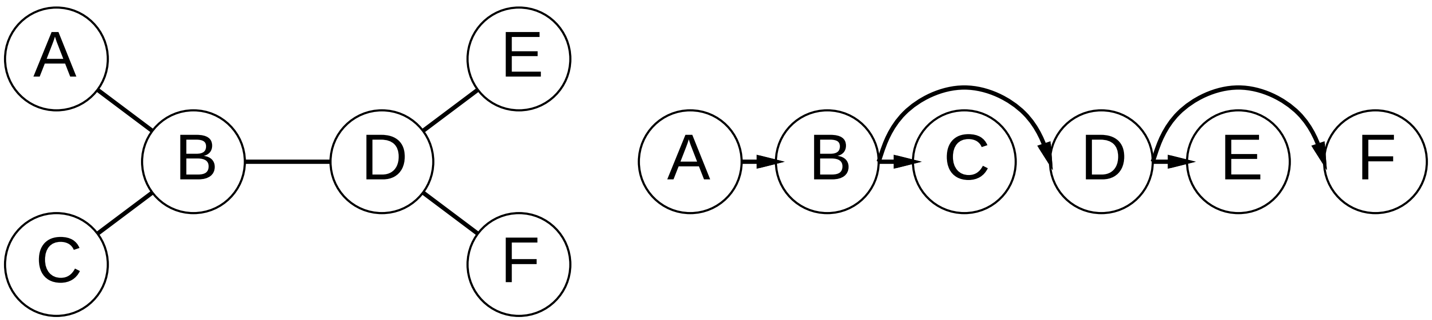
Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning: an important example of the relation between

- syntactic restrictions and
- the complexity of reasoning

# Algorithm for tree-structured CSPs

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

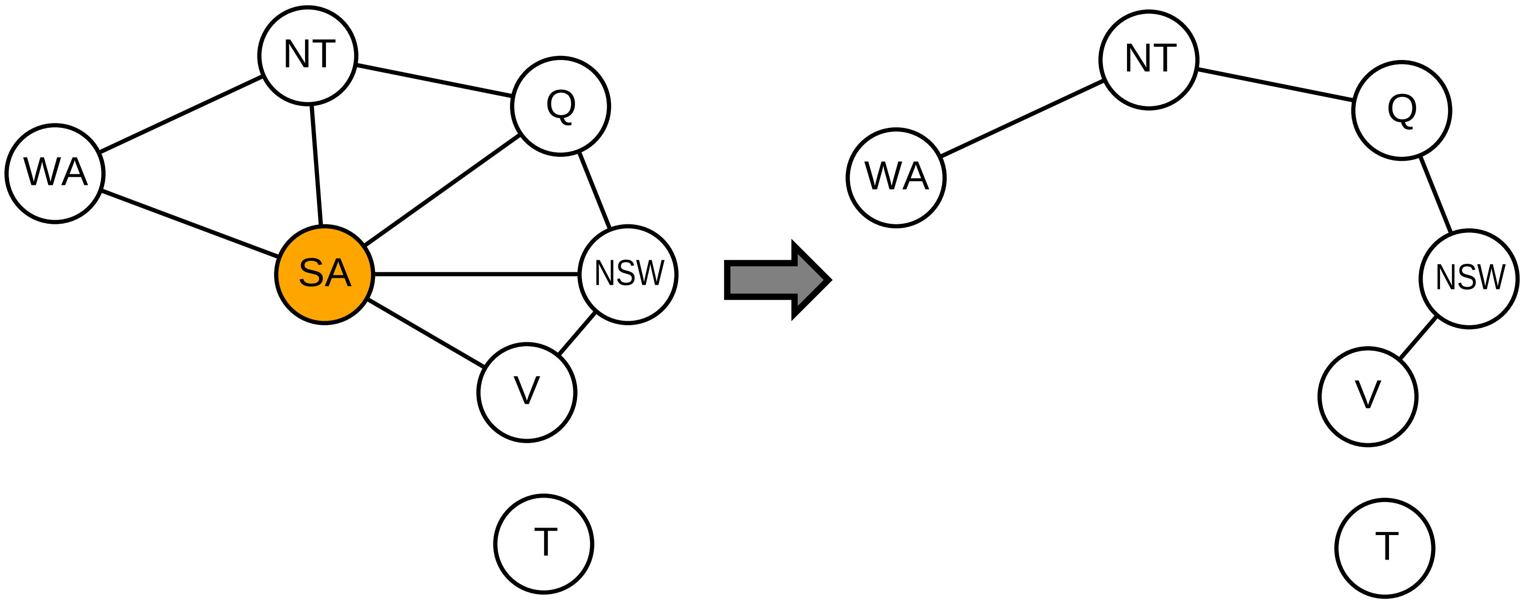



2. For $j$ from $n$ down to 2, apply $\textsc{RemoveInconsistentValues}(Parent(X_j), X_j)$

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

# Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains




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

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

# Further Optimizations

- Tree decomposition
- Symmetry breaking

# 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

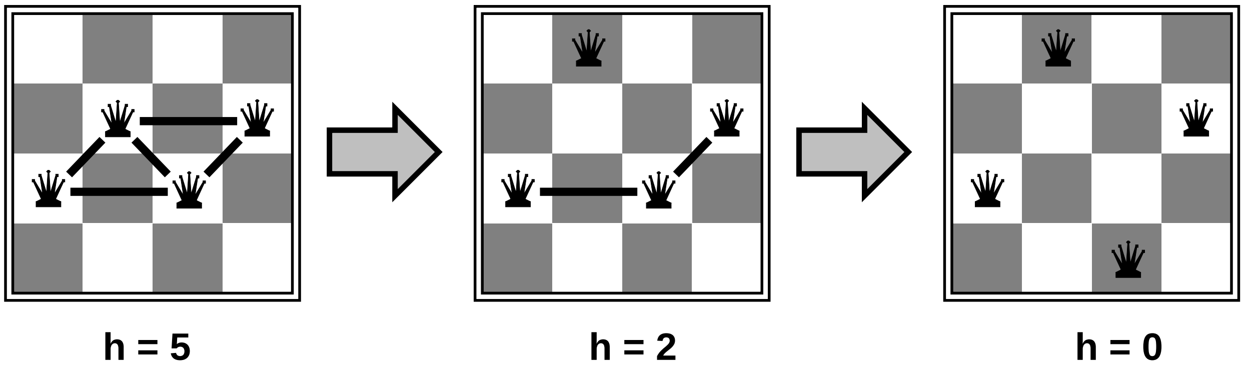
# Example: 4-Queens

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

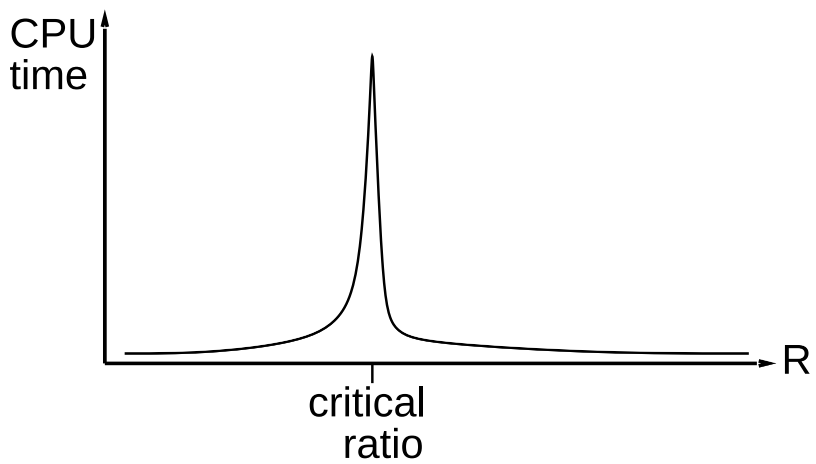


# Performance of min-conflicts

Given random initial state, can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 10{,}000{,}000$)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$




The critical ration corresponds to a phase transition for the problems, from satisfiable to unsatisfiable