Last update: October 4, 2012

Constraint Satisfaction Problems

CMSC 421, Chapter 6

Outline

- ◇ CSP examples
- ◇ Backtracking search for CSPs
- ◇ Problem structure and problem decomposition
- ◇ Local search for CSPs

Constraint satisfaction problems (CSPs)

◇ Standard search problem:

- state: any data structure that supports goal test, eval, successor

◇ CSP:

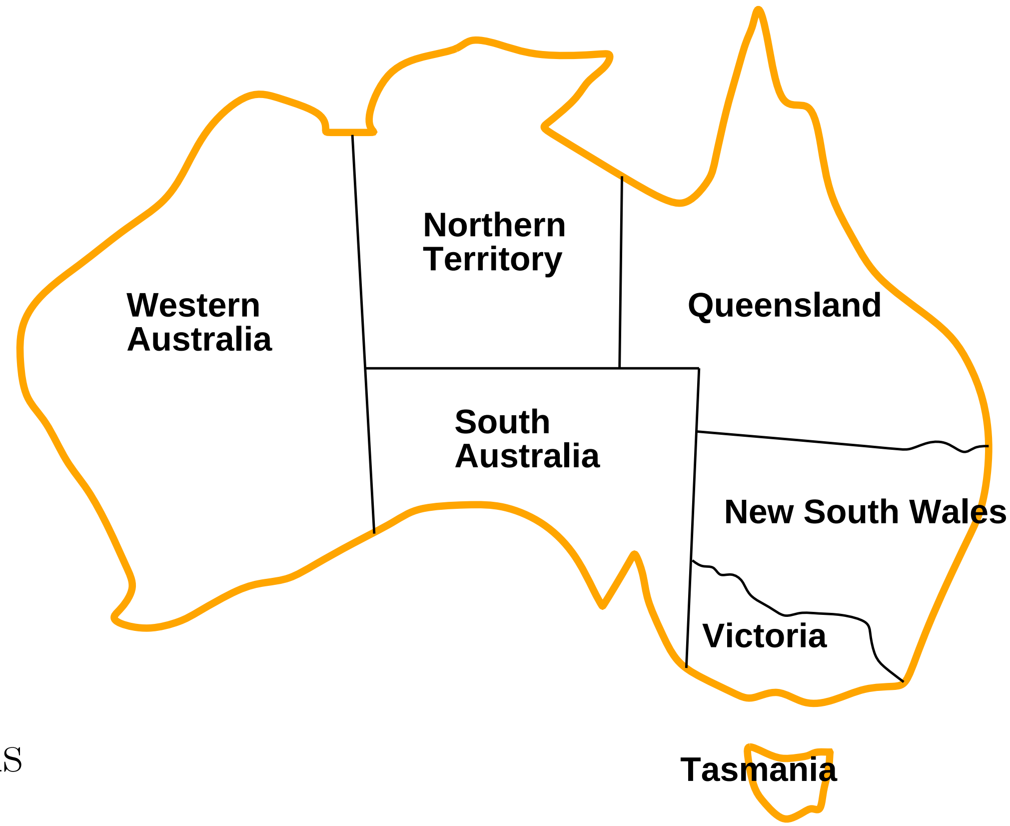
- state is a set of assignments of values to *variables* $\{X_i\}_{i=1}^n$ with *domains* $\{D_i\}_{i=1}^n$
- goal test is a set of *constraints* that specify allowable combinations of values for various sets of variables

◇ Simple example of a **formal representation language**

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

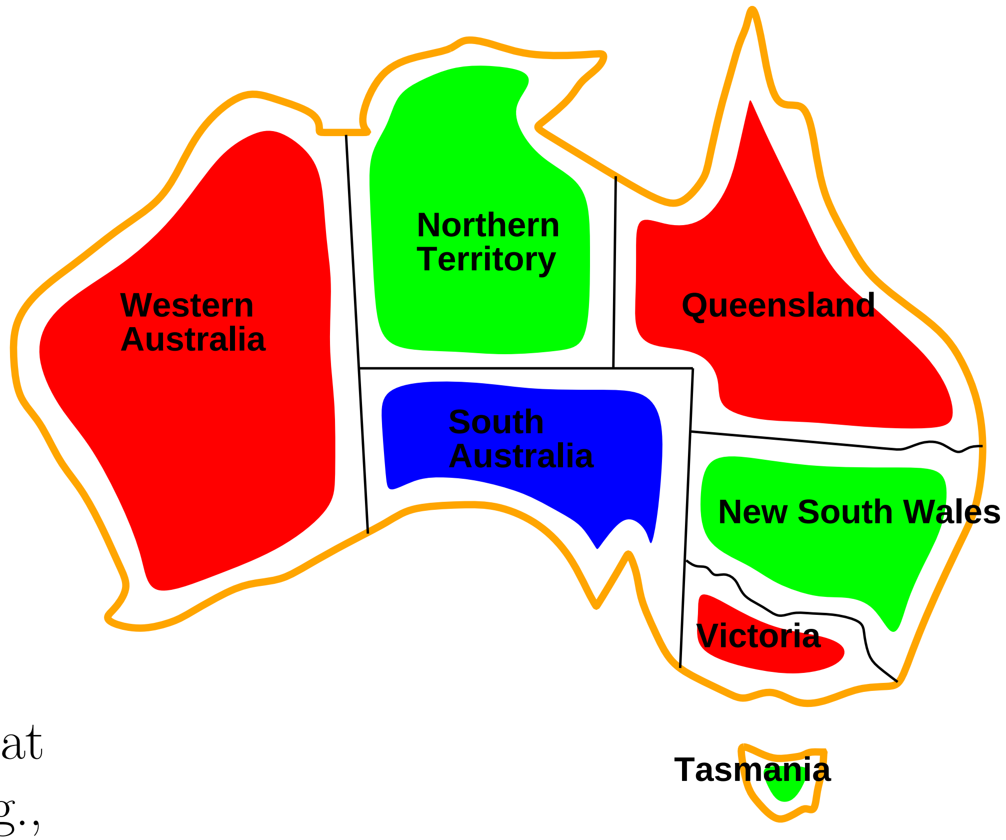
Example: map coloring

- Want to color the map of Australia, using at most three colors



- Variables: WA, NT, Q, NSW, V, SA, T
- Each variable's domain is $\{red, green, blue\}$
- Constraints: adjacent regions must have different colors,
  - e.g., $WA \neq NT$ if the language allows this
  - or else $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$

Example: map coloring, continued



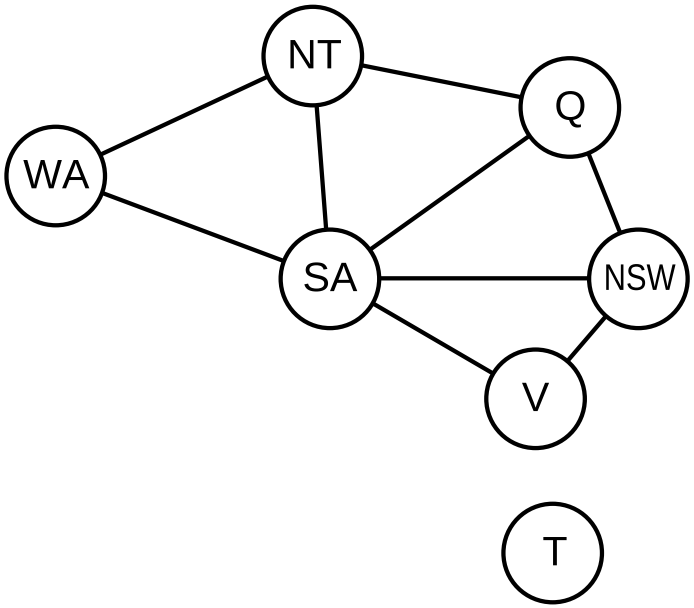
◊ Solutions are assignments that satisfy all the constraints, e.g.,

- $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

◊ *Binary CSP*: each constraint relates at most two variables

◊ *Constraint graph*: nodes are variables, edges represent constraints



◊ General-purpose CSP algorithms use graph structure to speed up search

- E.g., Tasmania is an independent subproblem

Varieties of CSPs

◇ Discrete variables

- if n variables, each with d possible values,
  then $O(d^n)$ complete assignments
- Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
  - ◇ 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 but NP-hard
  - ◇ nonlinear constraints undecidable

◇ Continuous variables

- e.g., start/end times for Hubble Space Telescope observations
- linear constraints solvable using Linear Programming (LP) methods
  - ◇ can be done in polynomial time, but very high overhead
  - ◇ usually use a low-overhead algorithm with exponential worst-case

Varieties of constraints

◇ *Unary* constraints involve a single variable,

- e.g., $SA \neq green$

◇ *Binary* constraints involve pairs of variables

- e.g., $SA \neq WA$

◇ *Preferences* (soft constraints), e.g., *red* is better than *green* often representable by a cost for each variable assignment

- e.g., $cost(red) = 1, cost(green) = 5$
- $\rightarrow$ constrained optimization problems

◇ *Higher-order* constraints involve 3 or more variables,

- e.g., cryptarithmetic (next slide)

Example: Cryptarithmetic

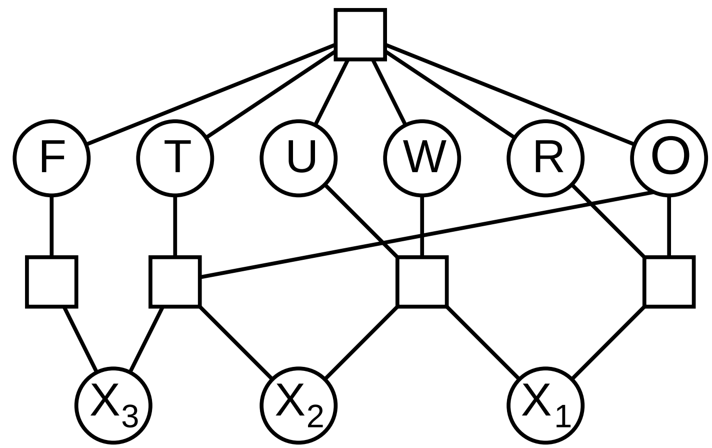
◇ Find distinct digits
F, O, R, T, U, W
such that

```
    T W O
  + T W O
  -------
  F O U R
```

◇ Solution:

```
    7 3 4
  + 7 3 4
  -------
  1 4 6 8
```

- Each square box represents a constraint:



- 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:
  - ◇ $alldiff(F, T, U, W, R, O)$
  - ◇ $O + O = R + 10X_1$
  - ◇ etc.

Real-world CSPs

◇ Assignment problems

- e.g., who teaches what class

◇ Timetabling problems

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

◇ Hardware configuration

◇ Spreadsheets

◇ Transportation scheduling

◇ Factory scheduling

◇ Floorplanning (e.g., factory layouts)

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

Standard search formulation (incremental)

◇ States are defined by the values assigned so far

- Initial state: the empty assignment, $\{\,\}$
- Successor function: choose an unassigned variable x
  - ◇ assign a value to x that doesn't conflict with the other variables
  - ◇ $\Rightarrow$ fail if no legal assignments
- Goal test: the current assignment is complete
- Path is irrelevant

◇ Let's start with the straightforward, dumb approach, then fix it

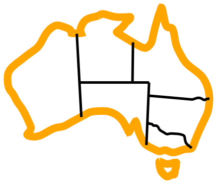
◇ With n variables, every solution is at depth n $\Rightarrow$ use depth-first search

- Suppose there are d possible values for each variable
- Then for $i = 1, \ldots, n$,
  - ◇ at depth i there are $n - i$ unassigned variables
  - ◇ so the branching factor at depth i is $b_i = (n-i)d$
- So the number of leaves is $b_0 b_1 \ldots b_n = n!d^n$

Backtracking search

- Variable assignments are *commutative*
  - e.g., these two:
    - first assign $WA = red$, then $NT = green$
    - first assign $NT = green$, then $WA = red$
- Only need to consider assignments to a single variable at each node
    - $\Rightarrow$ $b = d$ 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 $n \approx 25$

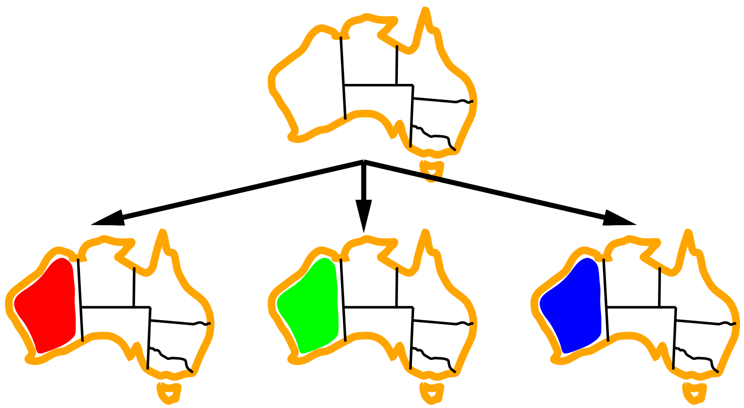
Backtracking search



function BACKTRACKING-SEARCH(*CSP*)
  return BACKTRACK({ }, *CSP*)

function BACKTRACK(*assignment*, *CSP*)
  if *assignment* is complete **then**
    return *assignment*
  select an unassigned variable x in *CSP*
  for each possible value v of x
    new = *assignment* $\cup$ $\{x = v\}$
    if *new* doesn't violate *CSP*'s constraints **then**
      result $\leftarrow$ BACKTRACK(*new*, *CSP*)
      if *result* $\neq$ Failure **then return** *result*
  return Failure

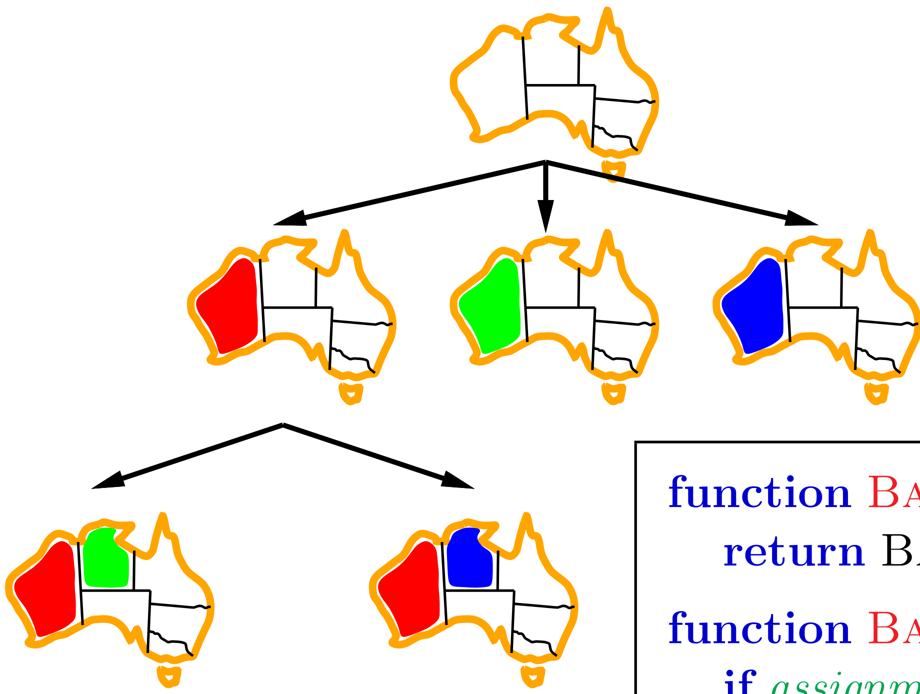
Backtracking search



function BACKTRACKING-SEARCH(*CSP*)
  return BACKTRACK({ }, *CSP*)

function BACKTRACK(*assignment, CSP*)
  if *assignment* is complete **then**
    return *assignment*
  select an unassigned variable x in *CSP*
  for each possible value v of x
    $new = assignment \cup \{x = v\}$
    if *new* doesn't violate *CSP*'s constraints **then**
      result ← BACKTRACK(*new*, *CSP*)
      if *result* ≠ Failure **then return** *result*
  return Failure

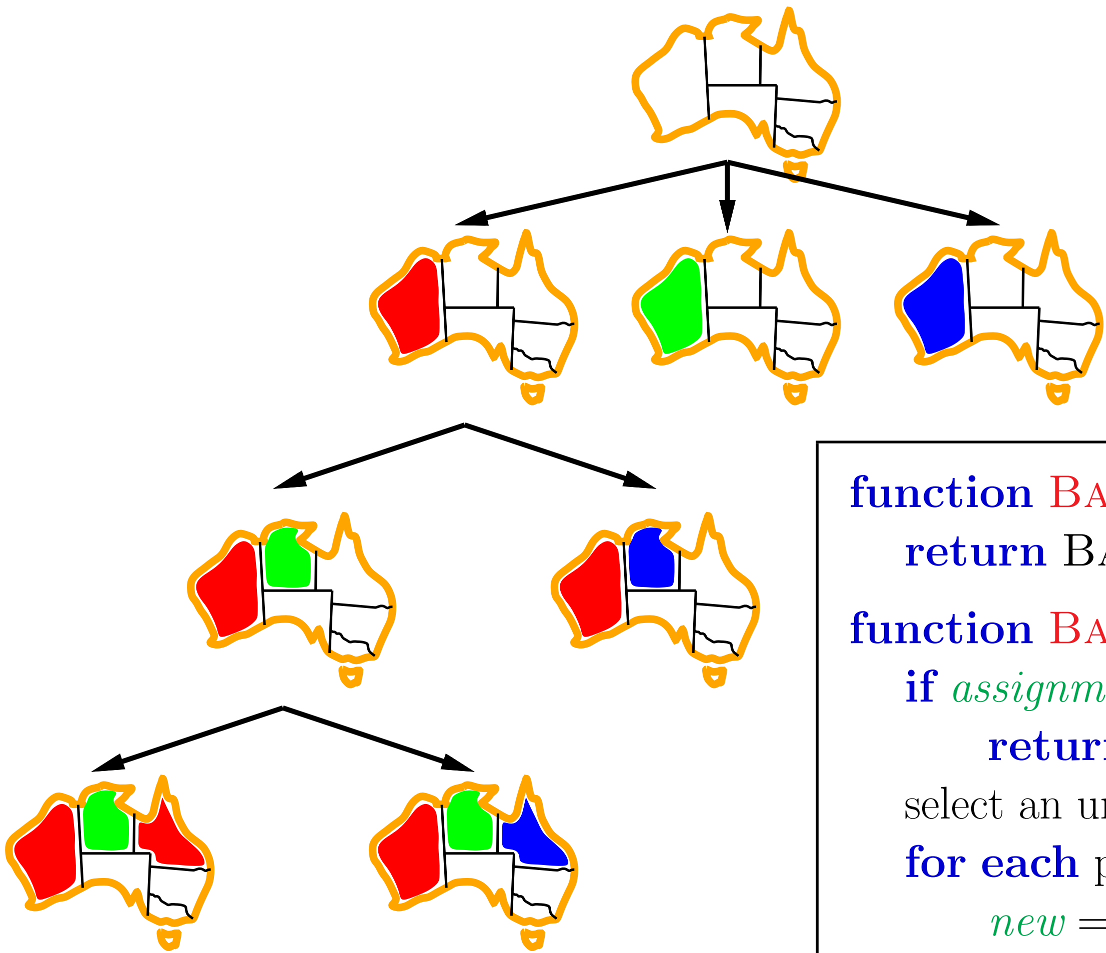
Backtracking search



function BACKTRACKING-SEARCH(*CSP*)
  return BACKTRACK({ }, *CSP*)

function BACKTRACK(*assignment, CSP*)
  if *assignment* is complete **then**
    return *assignment*
  select an unassigned variable x in *CSP*
  for each possible value v of x
    $new = assignment \cup \{x = v\}$
    if *new* doesn't violate *CSP*'s constraints **then**
      result ← BACKTRACK(*new*, *CSP*)
      if *result* ≠ Failure **then return** *result*
  return Failure

Backtracking search



function BACKTRACKING-SEARCH(*CSP*)
  return BACKTRACK({ }, *CSP*)

function BACKTRACK(*assignment, CSP*)
  if *assignment* is complete **then**
    return *assignment*
  select an unassigned variable x in *CSP*
  for each possible value v of x
    $new = assignment \cup \{x = v\}$
    if *new* doesn't violate *CSP*'s constraints **then**
      result ← BACKTRACK(*new*, *CSP*)
      if *result* ≠ Failure **then return** *result*
  return Failure

Improving backtracking efficiency

◊ There are **general-purpose** methods that can give huge gains in speed:

- Deciding which variable to assign next
- In what order to try the variable's values
- Detecting inevitable failures early
- Taking advantage of problem structure

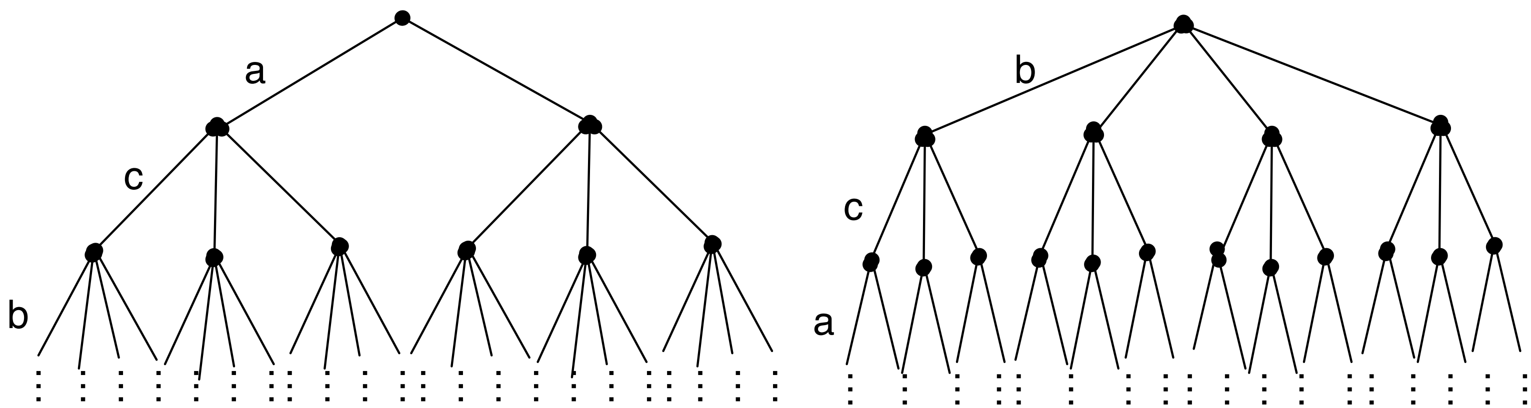
1. Deciding which variable to assign next

◇ **Minimum remaining values (MRV)** heuristic:

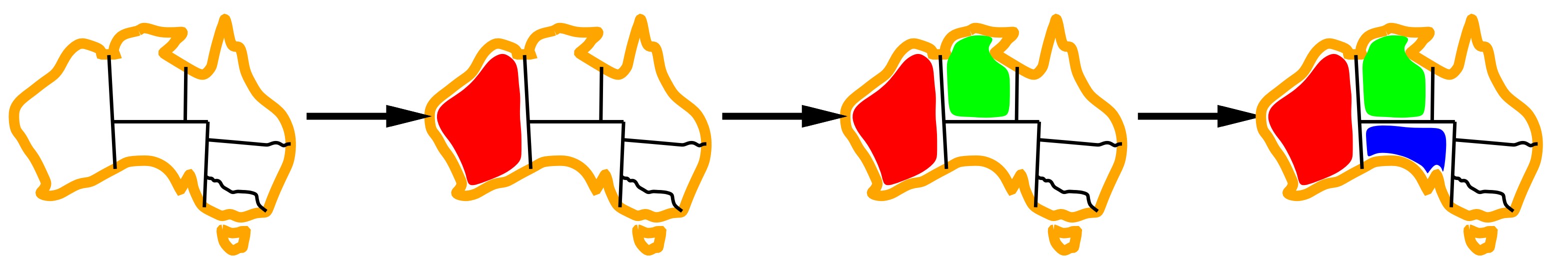
- Choose the variable with the fewest legal values

◇ Example: a has 2 possible values, b has 4, c has 3, ...

- Same number of leaves, but 1st tree has fewer nodes



◇ Australia example:



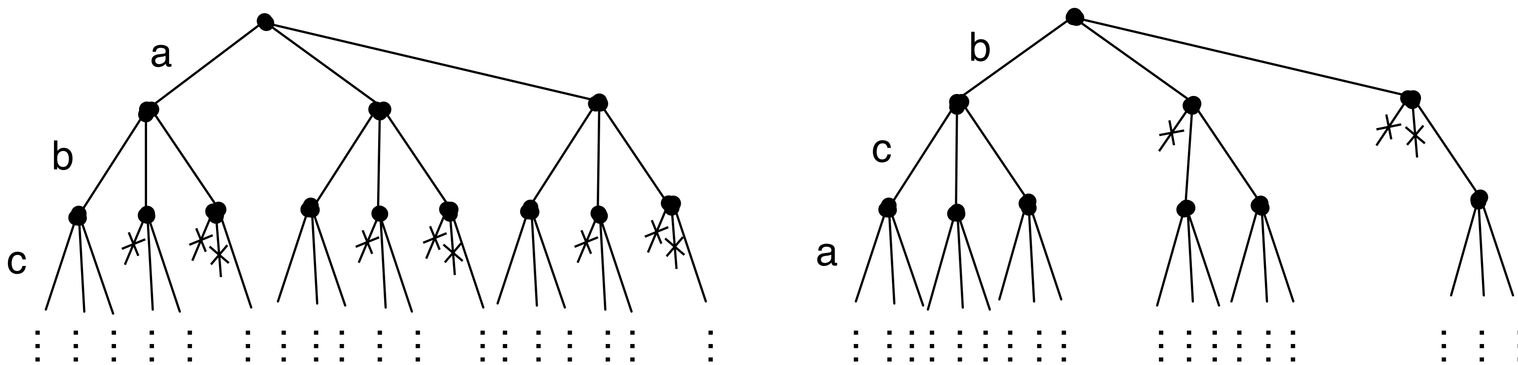
1. Deciding which variable to assign next

◊ **Degree** heuristic – use this as a tie-breaker among MRV variables

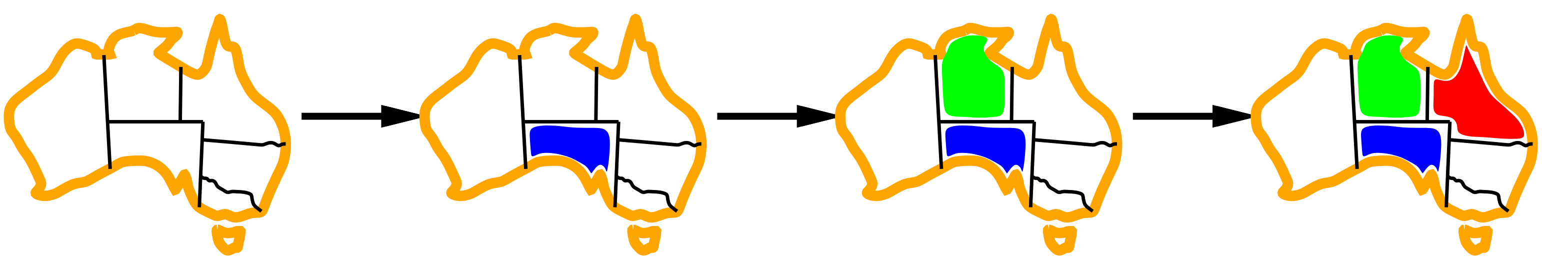
- Choose the variable that's involved in the largest number of constraints on other unassigned variables

◊ Example: $a, b, c, \ldots \in \{1, 2, 3\}$, a unconstrained, constraint $c \geq b$

- Better pruning in the 2nd tree



◊ Australia example: SA instead of WA



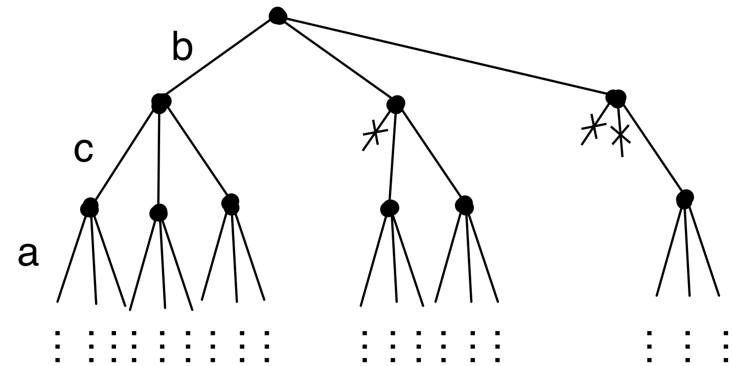
2. In what order to try a variable's values

◇ Once you've selected a variable, what value to choose for it?

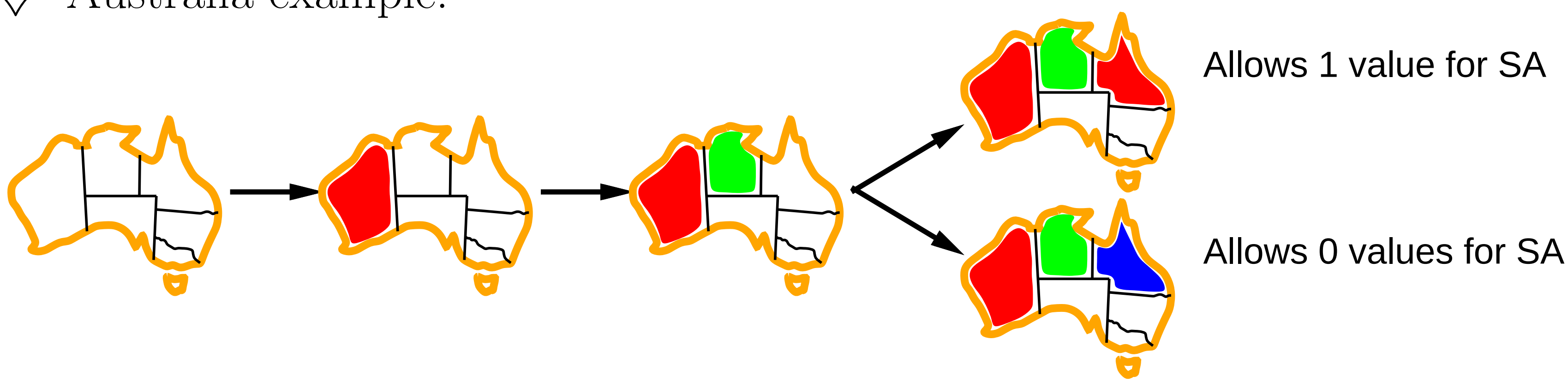
◇ **Least constraining value**: the one that rules out the fewest values in the remaining variables

◇ Example: $a, b, c, \ldots \in \{1, 2, 3\}$, a unconstrained, constraint $c \geq b$

- $b = 1$ is more likely to lead to a solution:



◇ Australia example:

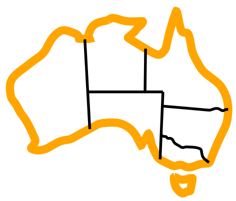


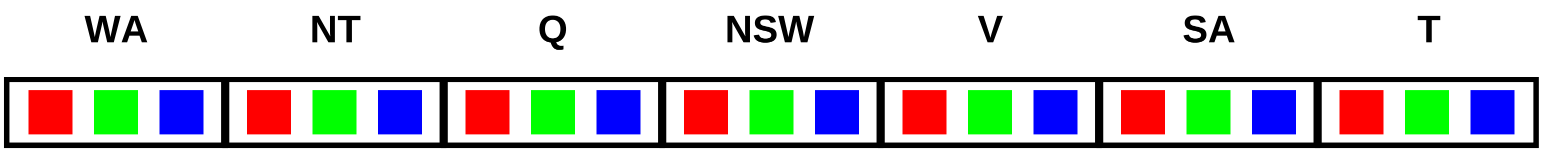
◇ Combining the three heuristics makes 1000 queens feasible

3. Detecting inevitable failures early

♢ Forward checking

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

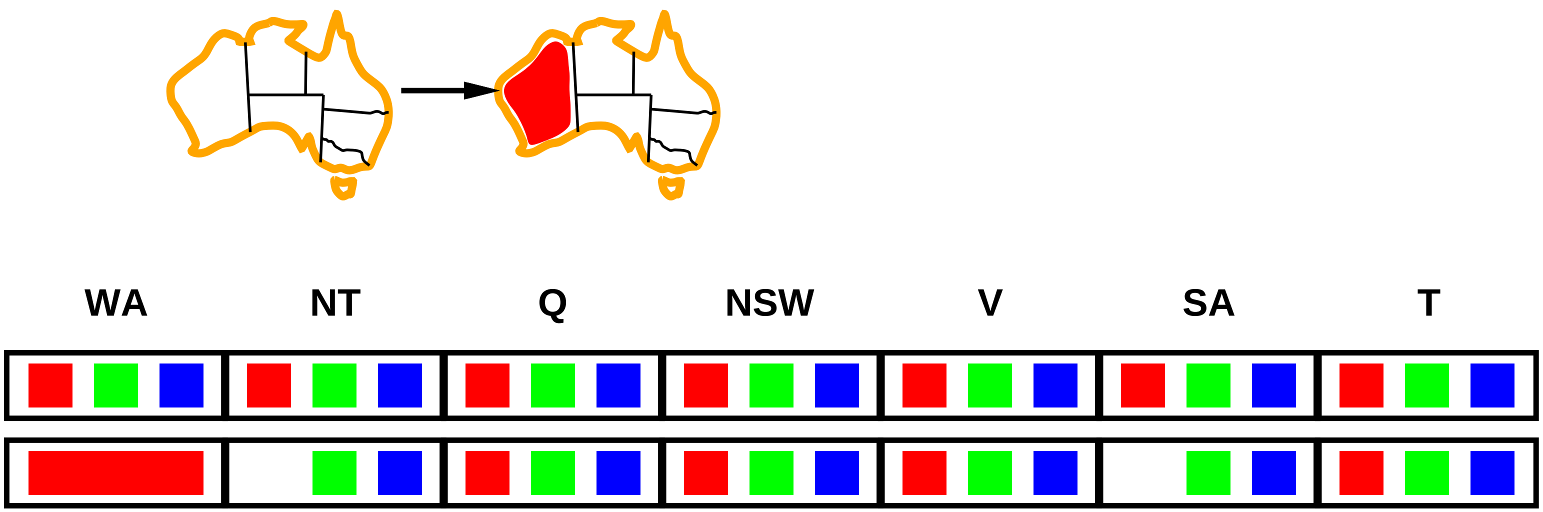




3. Detecting inevitable failures early

◇ Forward checking

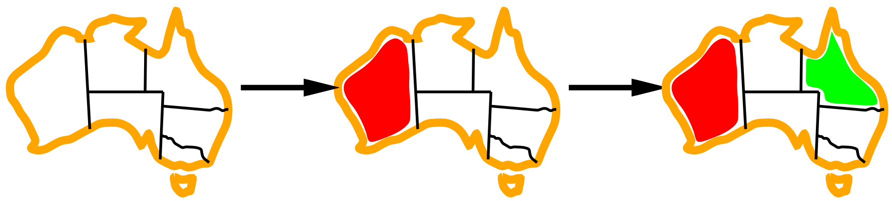
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



3. Detecting inevitable failures early

♢ Forward checking

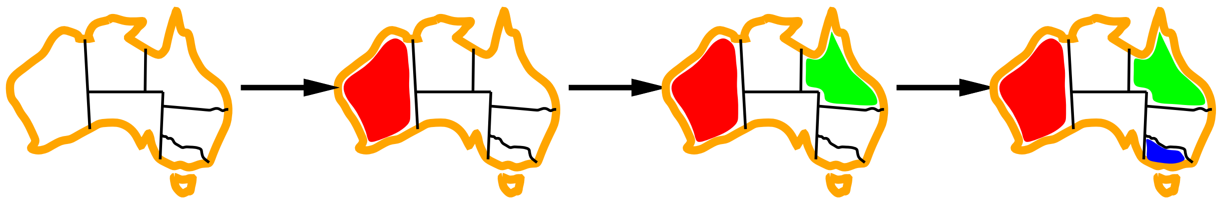
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

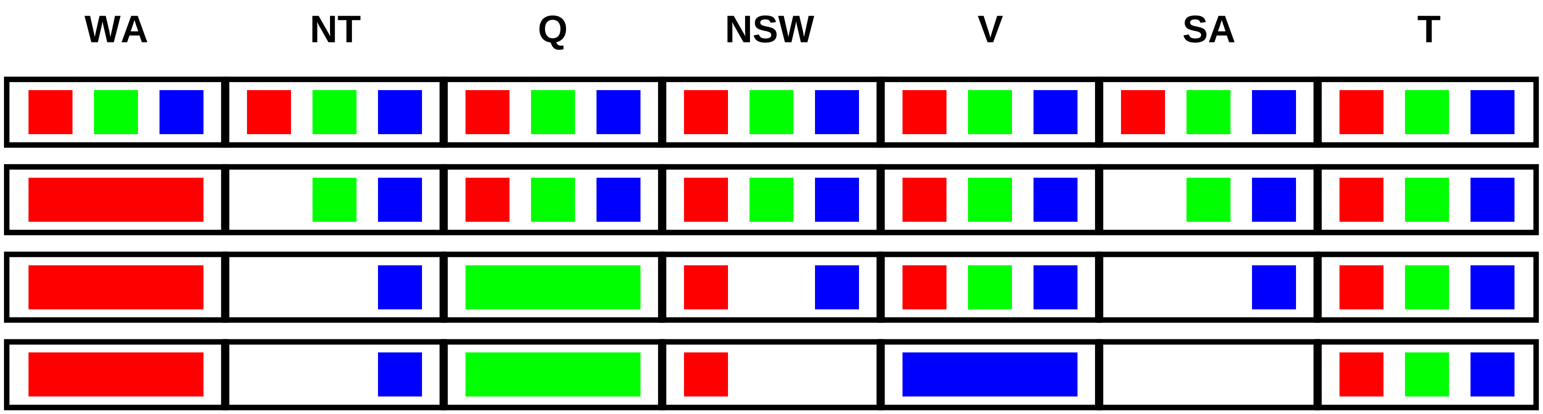


3. Detecting inevitable failures early

♢ Forward checking

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



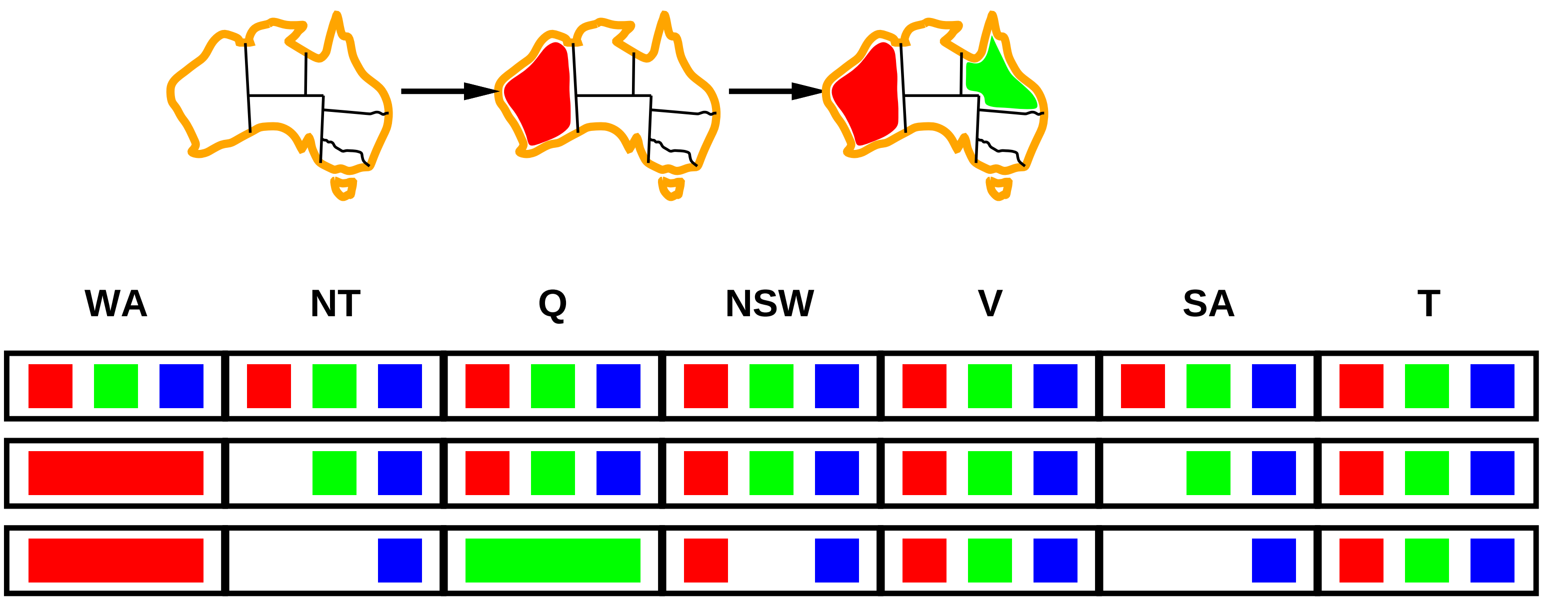


3. Detecting inevitable failures early

◇ Forward checking detects when a variable has no remaining legal values

- But sometimes we can detect even earlier that a failure is inevitable

◇ E.g., *NT* and *SA* cannot both be blue!



◇ To detect this, use *constraint propagation*

- repeated local enforcement of constraints

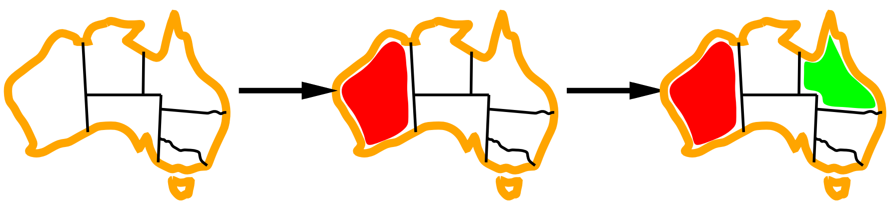
Constraint propagation

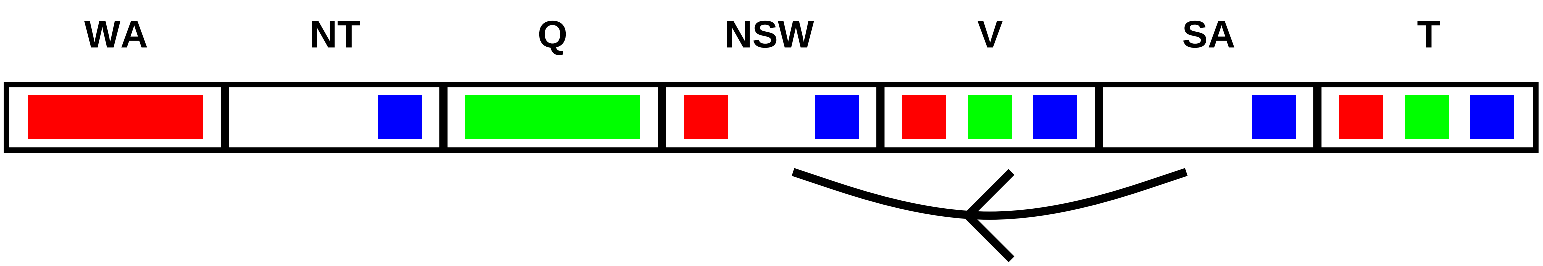
◇ **Arc consistency**: For each constraint on X and Y, consider two arcs:

- ◇ $X \rightarrow Y$ and $Y \rightarrow X$

- $X \rightarrow Y$ is *consistent* iff for **every** value x of X,
  there **exists** an allowed value of Y

- Make $X \rightarrow Y$ consistent by removing the "bad" values of X





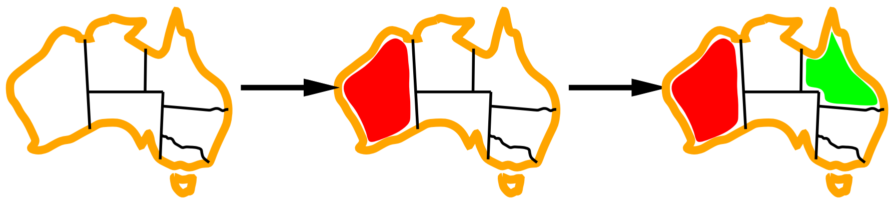
Constraint propagation

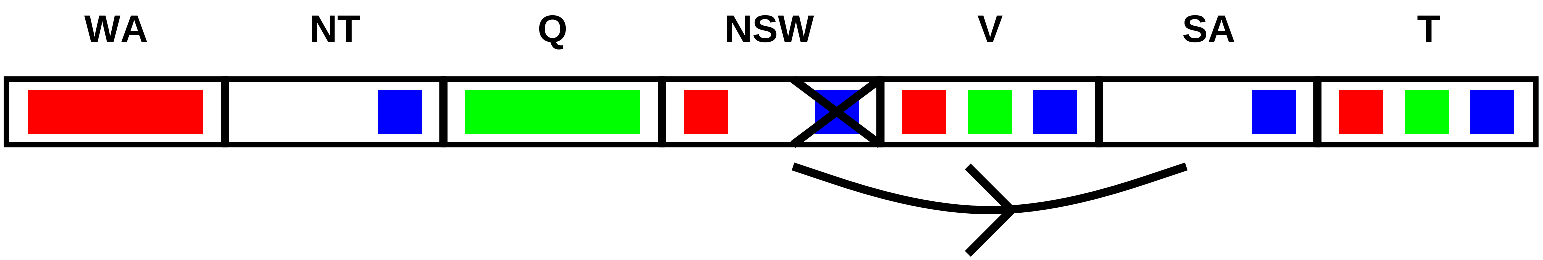
◊ **Arc consistency**: For each constraint on X and Y, consider two arcs:

- ◊ $X \rightarrow Y$ and $Y \rightarrow X$

- $X \rightarrow Y$ is *consistent* iff for **every** value x of X, there **exists** an allowed value of Y

- Make $X \rightarrow Y$ consistent by removing the “bad” values of X



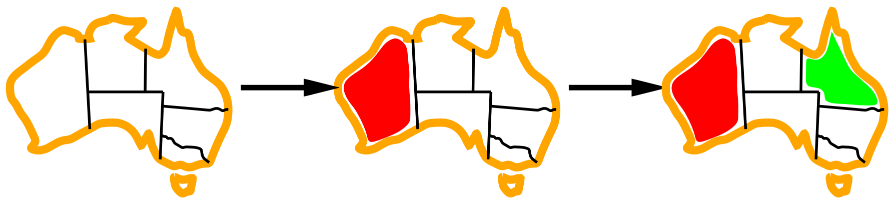


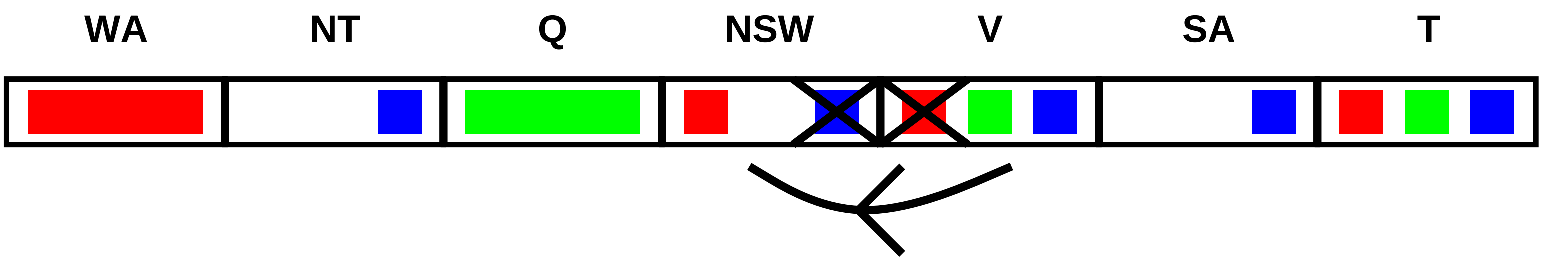
Constraint propagation

♢ **Arc consistency**: For each constraint on X and Y, consider two arcs:

- ⋄ $X \rightarrow Y$ and $Y \rightarrow X$

- $X \rightarrow Y$ is *consistent* iff for **every** value x of X, there **exists** an allowed value of Y
- Make $X \rightarrow Y$ consistent by removing the "bad" values of X





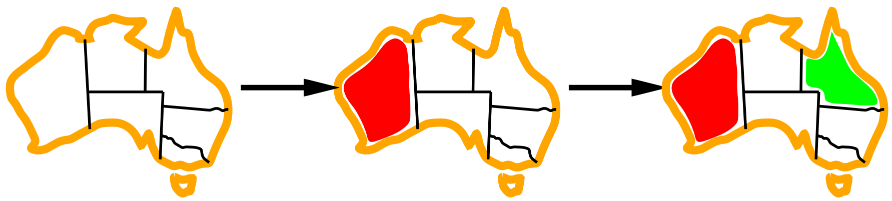
- If X loses a value, every arc $W \rightarrow X$ needs to be rechecked

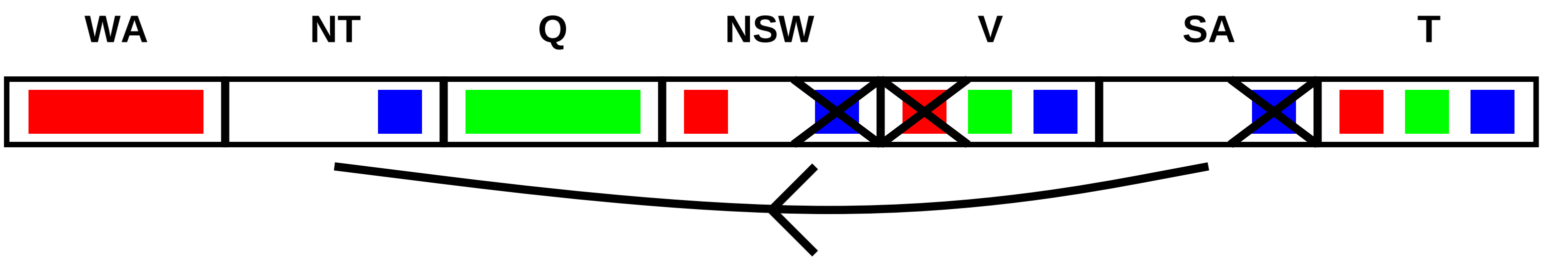
Constraint propagation

◇ **Arc consistency**: For each constraint on X and Y, consider two arcs:

- ◇ $X \to Y$ and $Y \to X$

- $X \to Y$ is *consistent* iff for **every** value x of X, there **exists** an allowed value of Y
- Make $X \to Y$ consistent by removing the "bad" values of X





◇ In general, finds failures earlier than forward-checking

- Finds all the failures forward-checking would find, plus more
- Doesn't find *all* failures – that's NP-hard

Arc consistency algorithm

```
function AC-3( CSP)
    queue ← a queue containing all the arcs in CSP
    while queue is not empty
        remove the first arc (X, Y) from queue
        if Remove-Inconsistent-Values(X, Y) then
            for each neighbor W of X, add (W, X) to queue
```

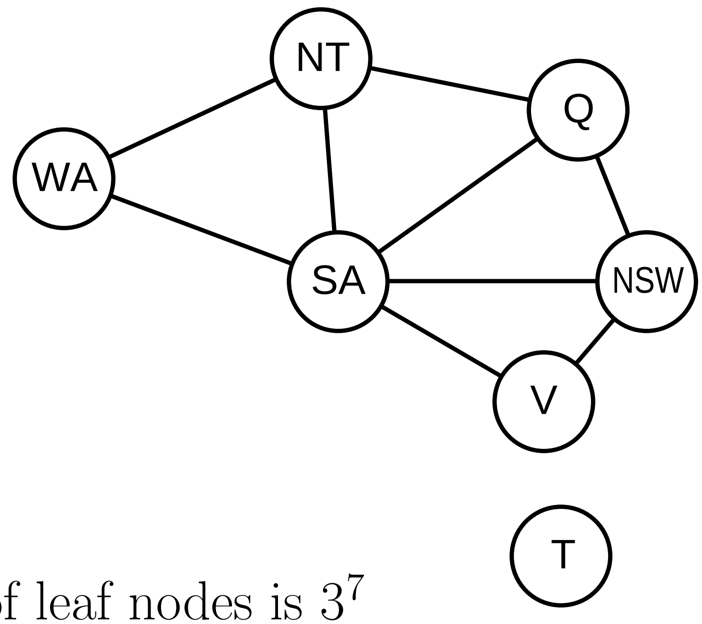
```
function Remove-Inconsistent-Values( X, Y)
    for each x in Domain[X]
        if there's no y in Domain[Y] such that (x,y) satisfies the constraint on (X, Y)
            then delete x from Domain[X]
    if anything was deleted then return true
    else return false
```

- $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- Can run as preprocessor, or after each assignment

◊ Example: $W, X, Y \in \{1, 2, 3\}, \quad X > W, \quad Y > X$

- $queue = \langle (W \rightarrow X), (X \rightarrow Y), \ldots \rangle$

4. Taking advantage of problem structure



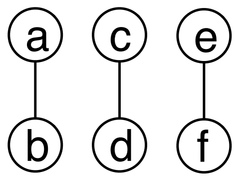
- ♢ Worst-case number of leaf nodes is 3^7
- ♢ But Tasmania and mainland are *independent subproblems*
  - Identifiable as *connected components* of constraint graph
- ♢ Handle them separately ⇒
  - one tree with at most 3^6 leaves, one with at most 3 leaves
- ♢ Can solve this nearly 3 times as fast

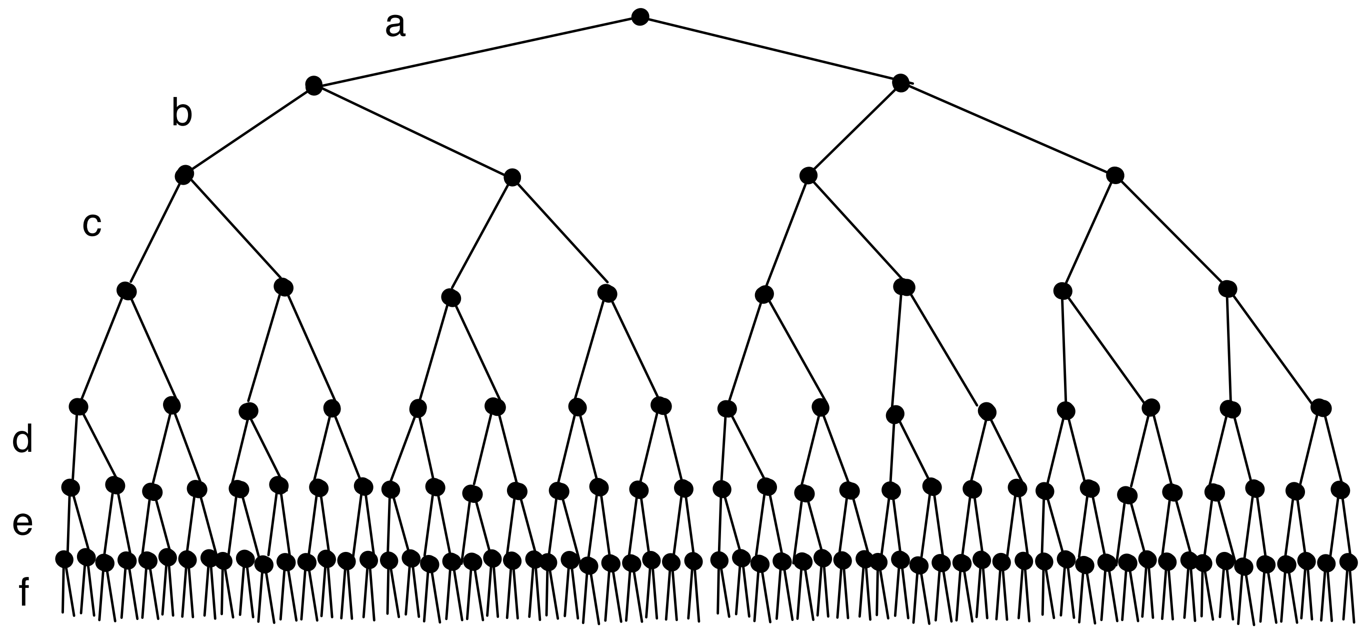
4. Taking advantage of problem structure

◇ Abstract example: 6 binary variables a, b, c, d, e, f

- Worst-case number of leaf nodes is $2^6 = 64$

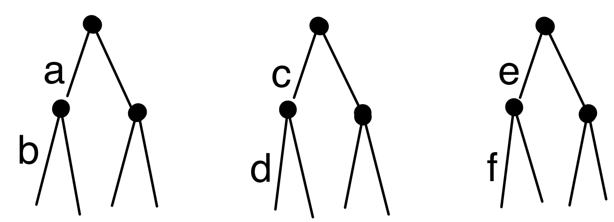
Constraint graph:





◇ Constraint graph shows there are three independent subproblems

- Handle separately $\Rightarrow$ 3 trees, 12 leaf nodes
- Can solve more than 5 times as fast



4. Taking advantage of problem structure

◇ With n variables, each having d possible values,

- worst-case number of leaf nodes is d^n, exponential in n

◇ Suppose we can divide into n/c independent subproblems,

- each with c variables

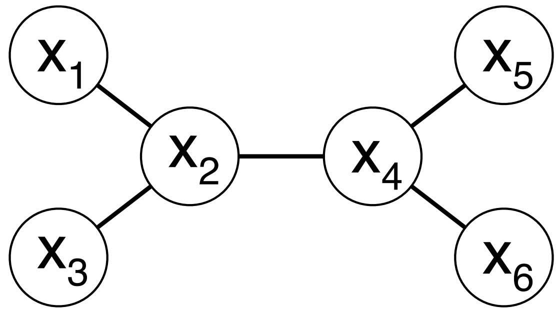
◇ Then the worst-case number of leaf nodes is $(n/c)d^c$

- linear in n

◇ E.g., $n = 80$, $d = 2$, $c = 20$, $n/c = 4$, at 10 million nodes/sec

- 2^{80} = 4 billion years
- 4×2^{20} = 0.4 seconds

Tree-structured CSPs



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

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

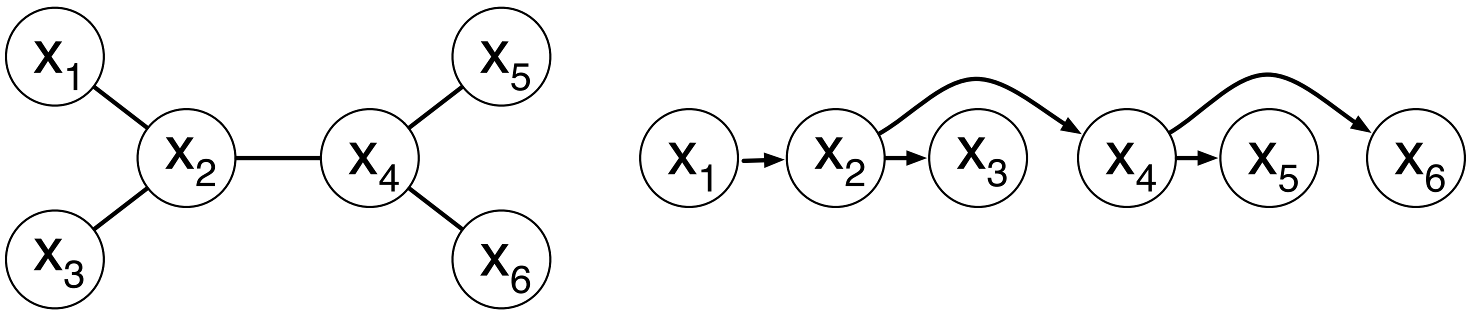
◇ This property also applies to logical and probabilistic reasoning
- good example of the relation between syntactic restrictions
- and the complexity of reasoning.

Algorithm for tree-structured CSPs

◇ Three steps:

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

   ◇ Like a topological sort



   ◇ Now the arcs only point one way

2. For j from n down to 2, apply arc-consistency

   ◇ Remove-Inconsistent-Values$(Parent(X_j), X_j)$

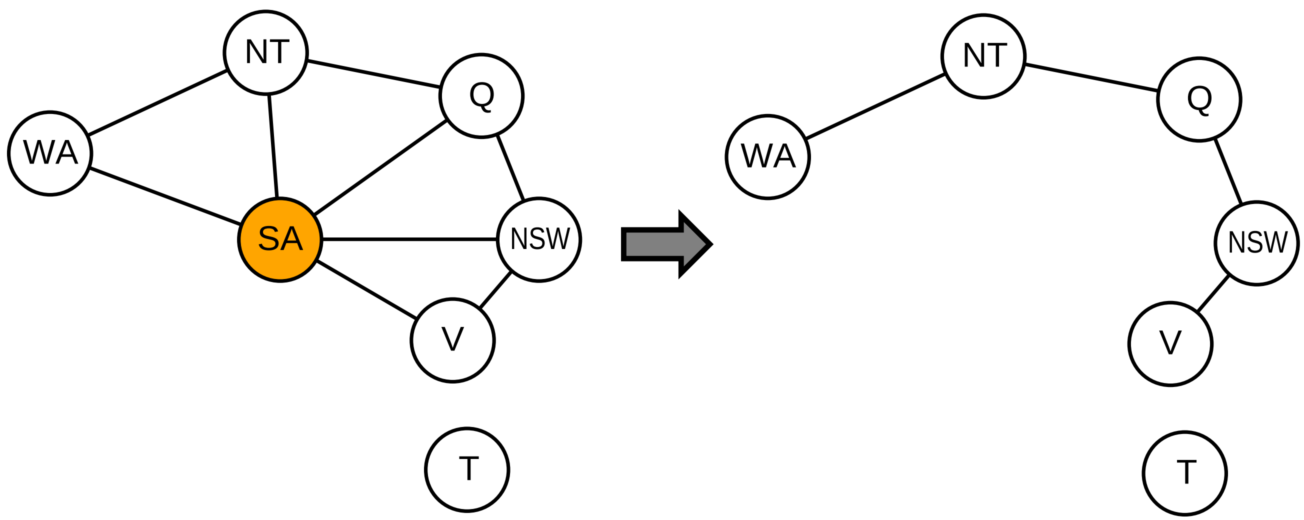
   ◇ Now we know that for each of a node's values, there are consistent values for its children

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

Nearly tree-structured CSPs

◇ *Conditioning*: instantiate a variable (in all possible ways)

- For each instantiation, prune its neighbors' domains



◇ *Cutset conditioning*: instantiate a set of variables such that the remaining constraint graph is a tree

- Then run the algorithm for tree-structured CSPs

◇ Cutset size c $\Rightarrow$ runtime $O(d^c(n-c)d^2)$

- very fast for small c

Iterative algorithms for CSPs

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

◇ To apply them to CSPs,

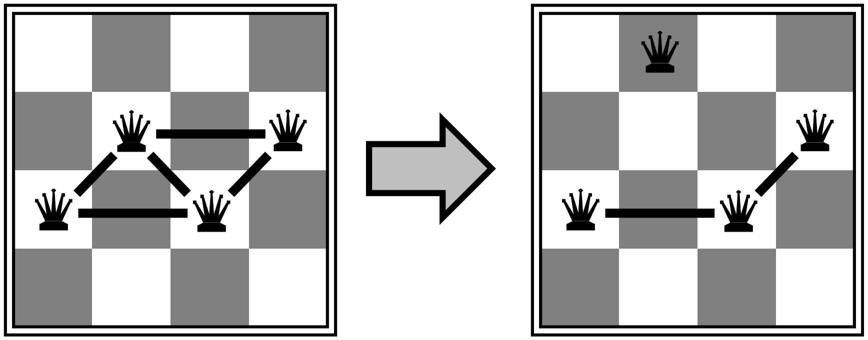
- allow complete states to have unsatisfied constraints

◇ Examples:

- Start with an arbitrary color for each Australian territory
- Start n-queens with each queen in an arbitrary row

◇ Operators **reassign** variable values

- e.g., change what row a queen is in:



Iterative algorithms for CSPs

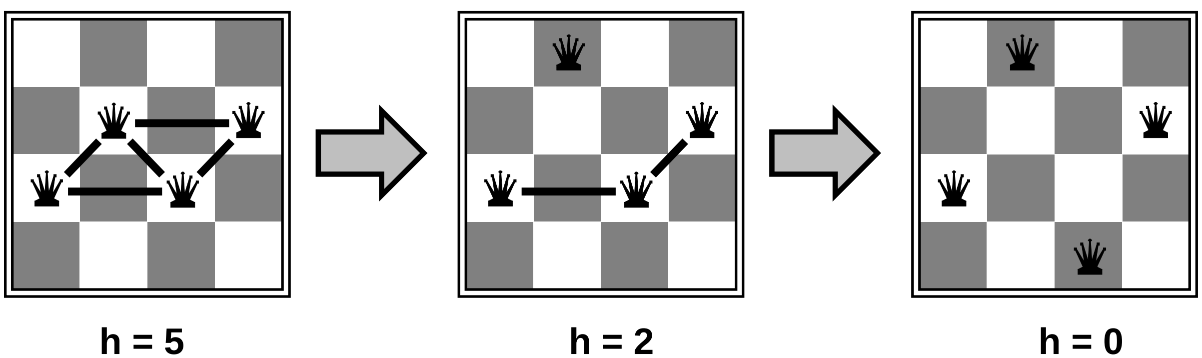
◇ Variable selection: randomly select any conflicted variable

◇ Value selection by *min-conflicts* heuristic:

- choose value that violates the fewest constraints
- i.e., hill-climbing with $h(n)$ = total number of violated constraints

◇ Example: 4-queens problem

- 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

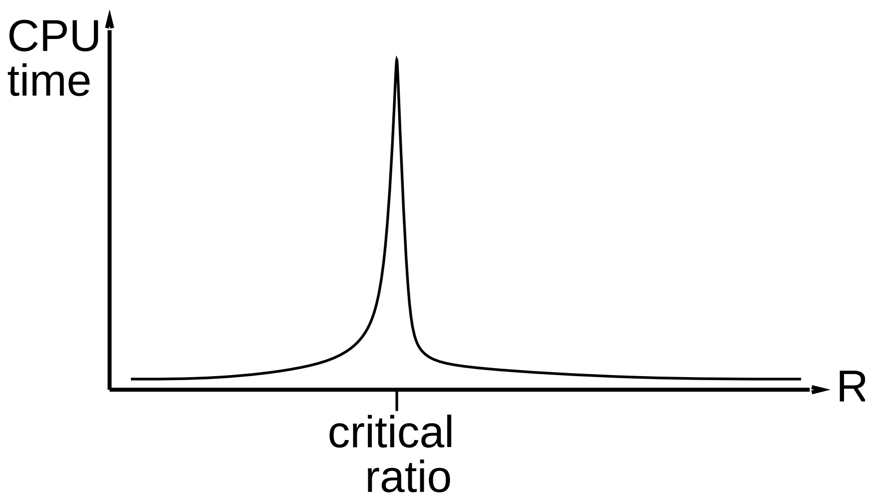


Performance of min-conflicts

◇ Given a 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

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



◇ More information at

- http://www.cs.cornell.edu/selman/papers/pdf/99.nature.phase.pdf

Summary

◇ CSPs: special kind of search problem
- state = set of assignments to a fixed set of variables
- goal test = whether the constraints are satisfied

◇ Backtracking = depth-first search, assign one variable at each node

◇ Ways to improve efficiency:
- Variable ordering and value selection
- Forward checking - detect inconsistencies that guarantee later failure
- Constraint propagation (e.g., arc consistency) - additional work
- to constrain values and detect inconsistencies

◇ Problem structure:
- Independent subproblems
- Tree-structured CSPs can be solved in linear time

◇ Can use iterative algorithms such as hill-climbing
- min-conflicts heuristic often works well

Revisions to Homework 3

- ◇ I had assigned three problems from chapter 5: 5.1, 5.9, 5.16
- ◇ I want to add two more from chapter 6, and extend the due date
  - Additional problems: 6.5, 6.11
  - New due date: Thursday, October 11
  - New late date: Tuesday, October 16