Last update: February 25, 2010

Constraint Satisfaction Problems

CMSC 421, Chapter 5

Outline

◇ CSP examples

◇ Backtracking search for CSPs

◇ Problem structure and problem decomposition

◇ Local search for CSPs

Constraint satisfaction problems (CSPs)

Standard search problem:
state is *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* $\{D_i\}_{i=1}^{n}$

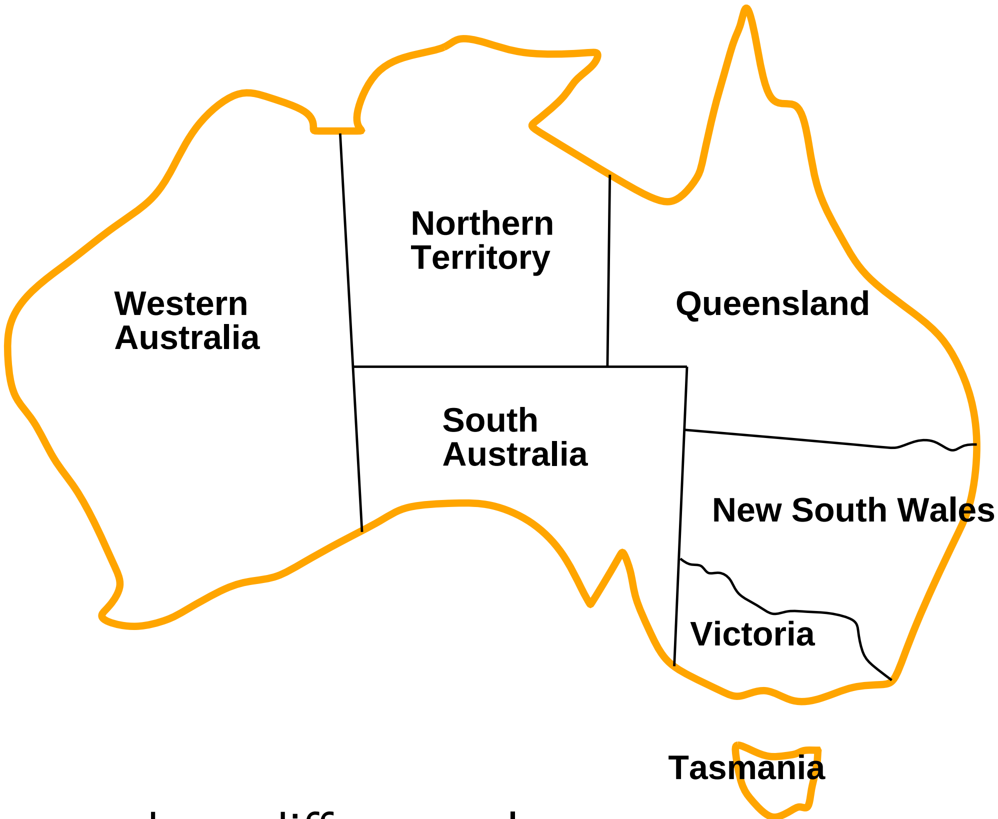
goal test is a set of *constraints* specifying
allowable combinations of values for various sets of variables

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



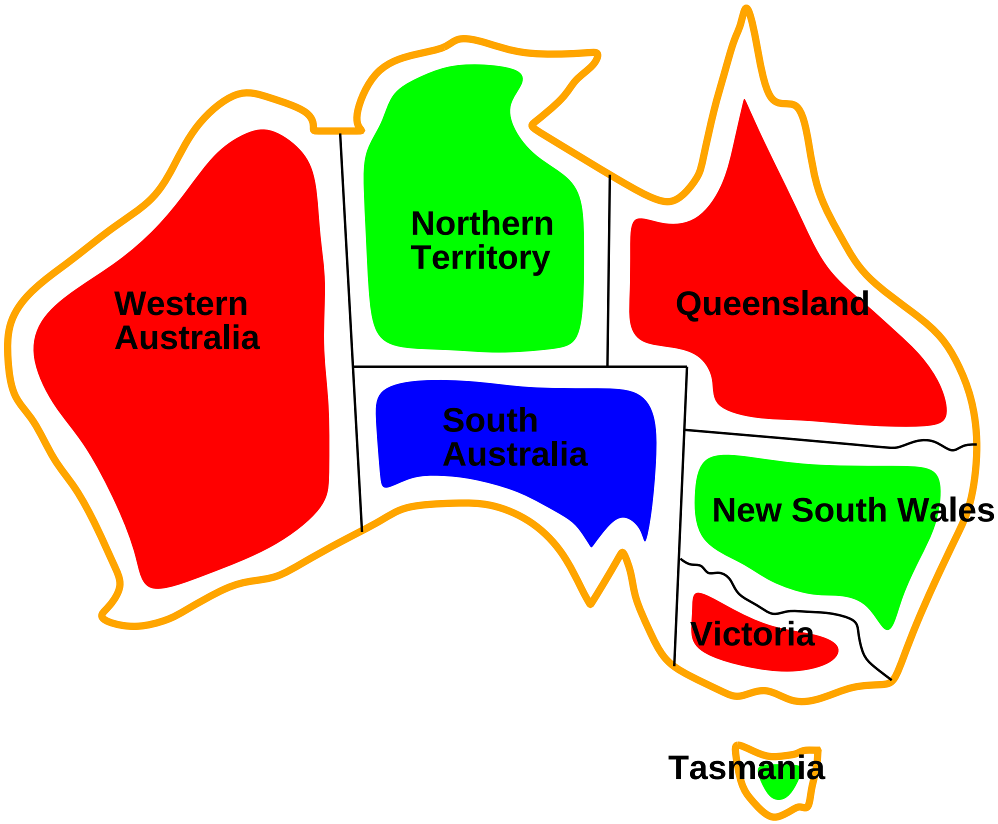
Variables: WA, NT, Q, NSW, V, SA, T

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

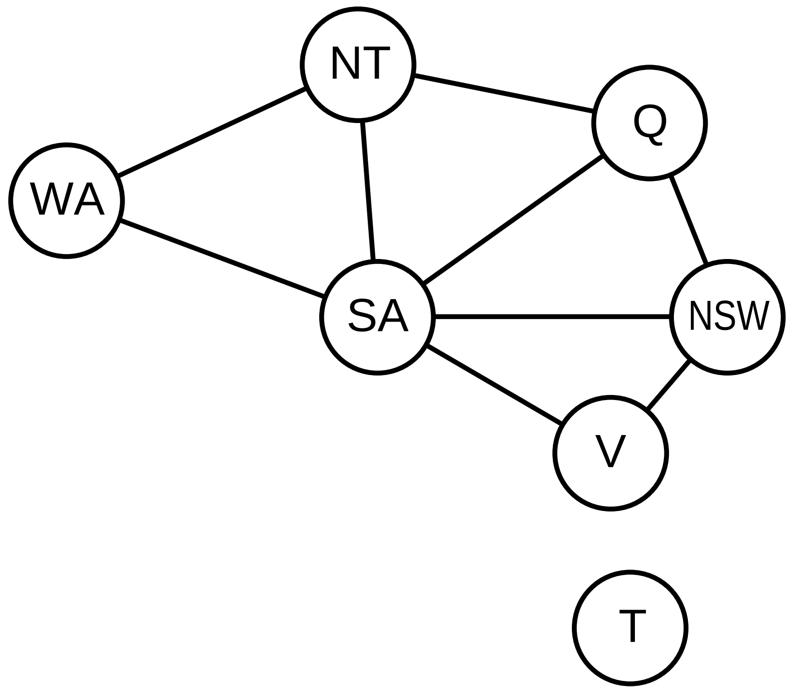


Solutions are assignments that satisfy all the constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, edges represent constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem

Varieties of CSPs

Discrete variables

finite domains of size d $\Rightarrow$ $O(d^n)$ complete assignments for n variables

- ♢ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

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 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., $cost(red) = 1, cost(green) = 5$
$\longrightarrow$ constrained optimization problems

Higher-order constraints involve 3 or more variables,
e.g., cryptarithmetic (next slide)

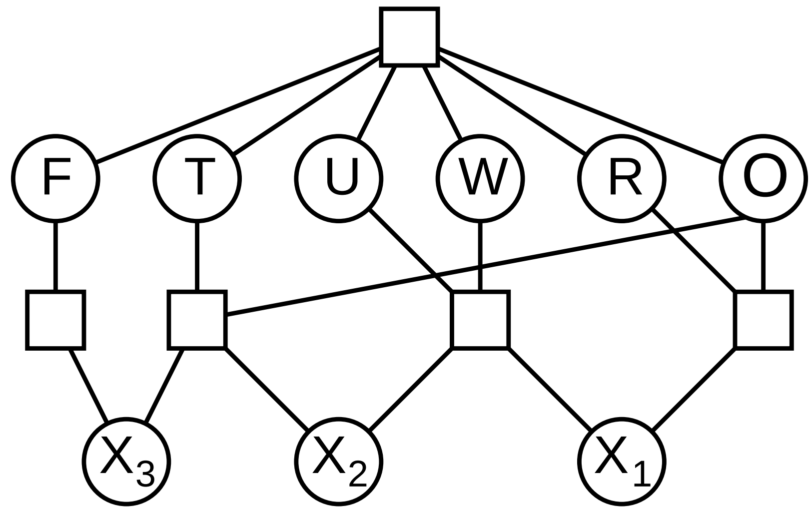
Example: Cryptarithmetic

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

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

Solution: 469+469=0938

Each 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)
$O + O = R + 10 \cdot X_1$,
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)

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

States are defined by the values assigned so far

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

◊ Successor function: choose an unassigned variable v
assign a value to v that doesn't conflict with the other variables
$\Rightarrow$ fail if no legal assignments

◊ Goal test: the current assignment is complete

1) This is the same for all CSPs
2) With n variables, every solution is at depth n $\Rightarrow$ use depth-first search
3) Path is irrelevant
4) If there are d possible values for each variable, then for $i = 1, \ldots, n$,
branching factor at depth i is $b_i = (n - i)d$,
so there are $b_0 b_1 \ldots b_n = n! d^n$ leaves!

Backtracking search

Variable assignments are *commutative*

$[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

Only need to consider assignments to a single variable at each node

$\Rightarrow \quad b = d$ 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

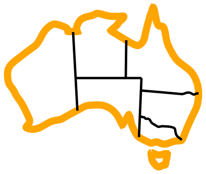
Can solve n-queens for $n \approx 25$

Backtracking search

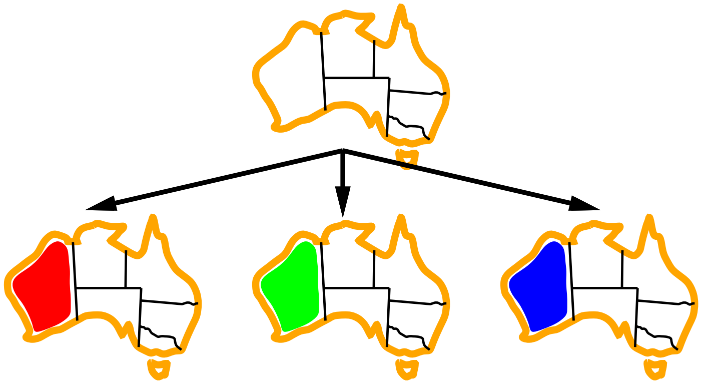
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(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
```

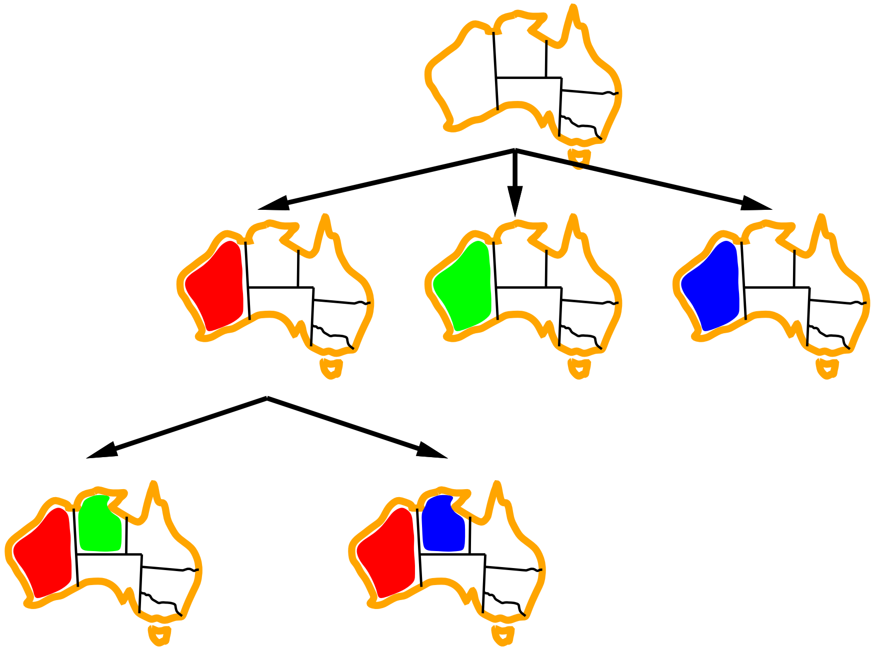
Backtracking example



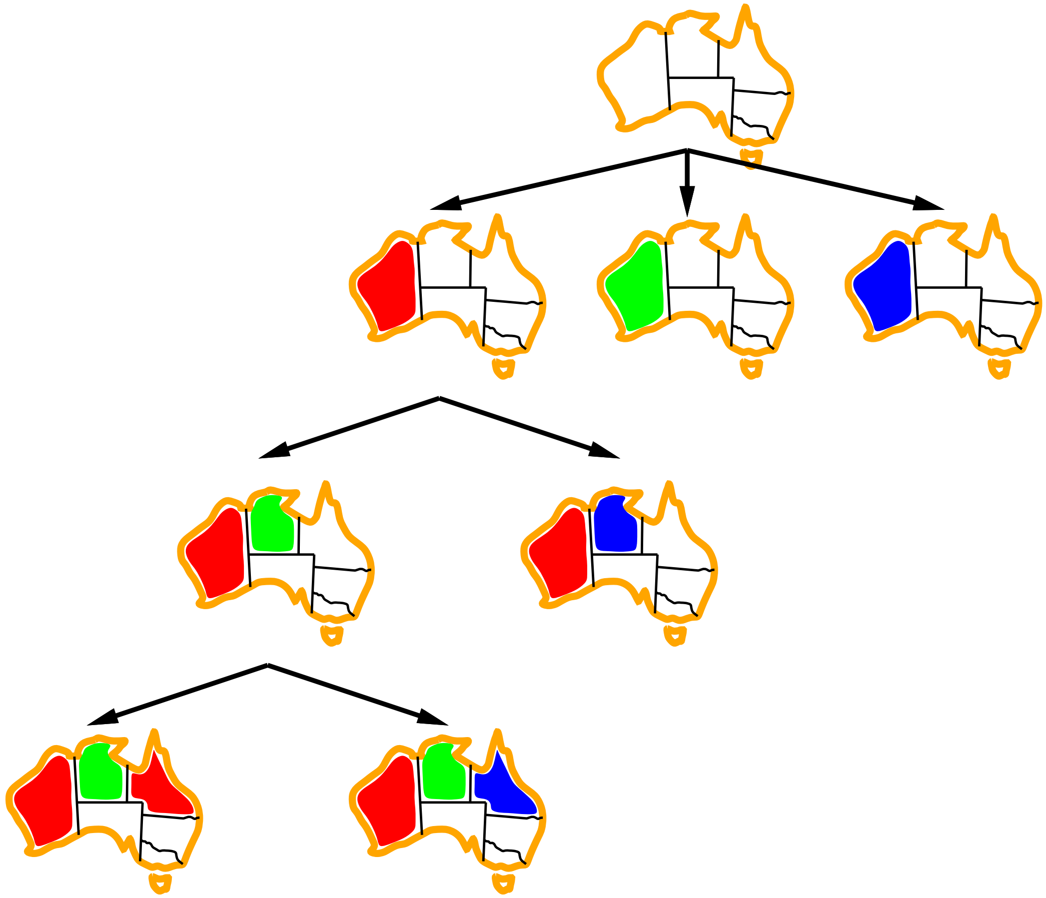
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

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

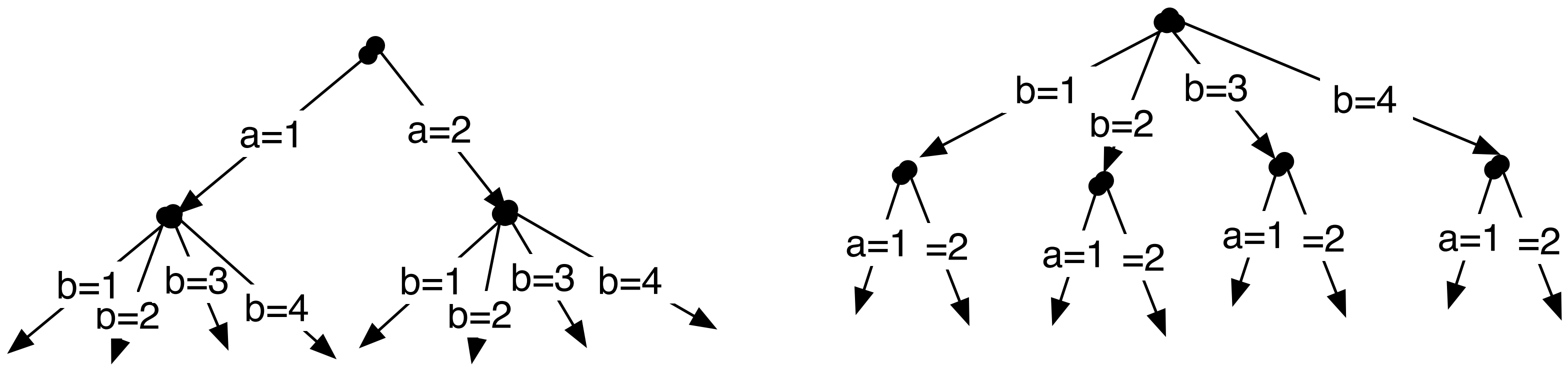
1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. How to take advantage of problem structure?

1. Which variable to assign next?

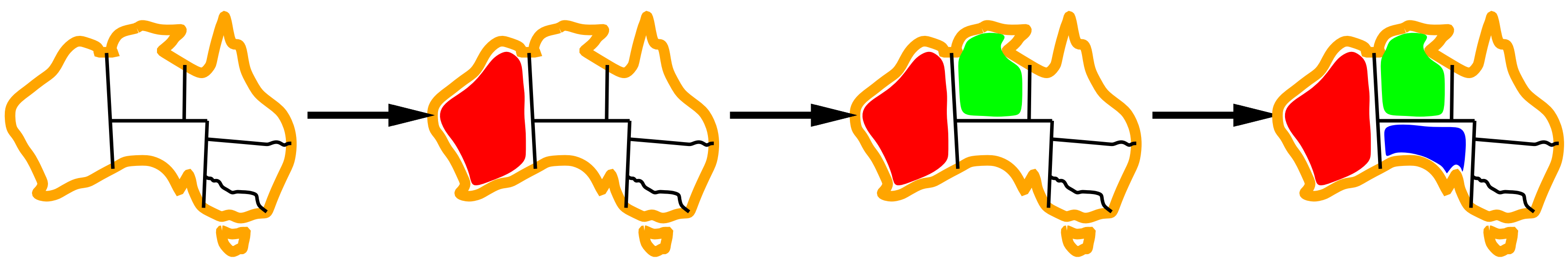
Minimum remaining values (MRV) heuristic:

◇ choose the variable with the fewest legal values

An abstract example: a has 2 possible values and b has 4 possible values:



Australia example:

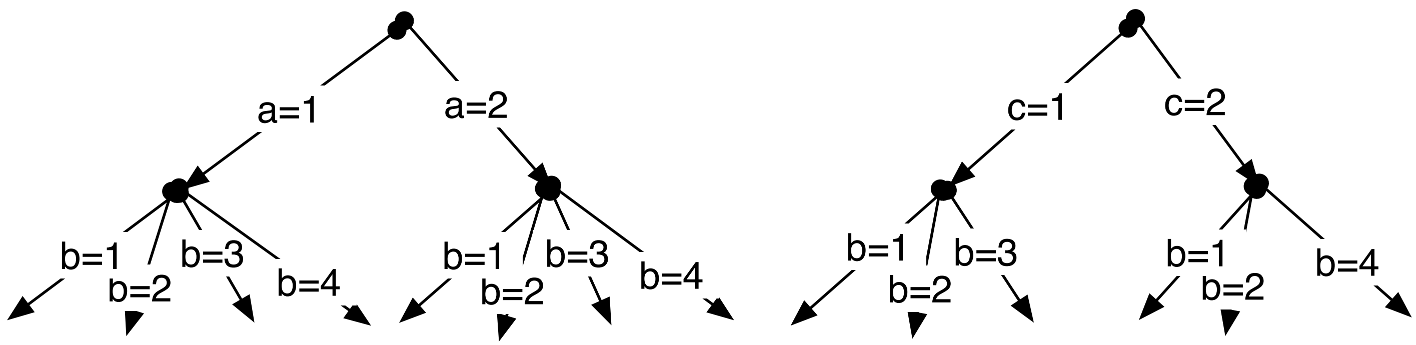


1. Which variable to assign next?

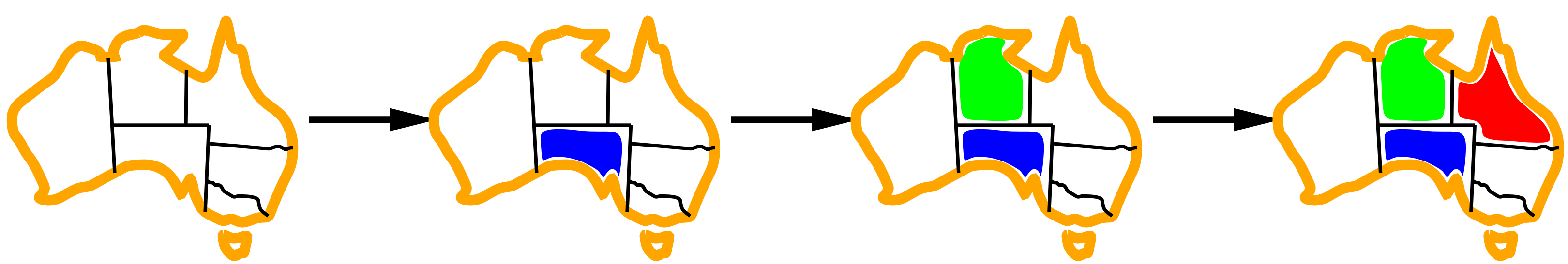
Degree heuristic: $\Rightarrow$ Use this as a tie-breaker among MRV variables

$\diamondsuit$ Choose the variable with the most constraints on remaining variables

Abstract example: a and c both have 2 possible values,
and c constrains b but a doesn't:



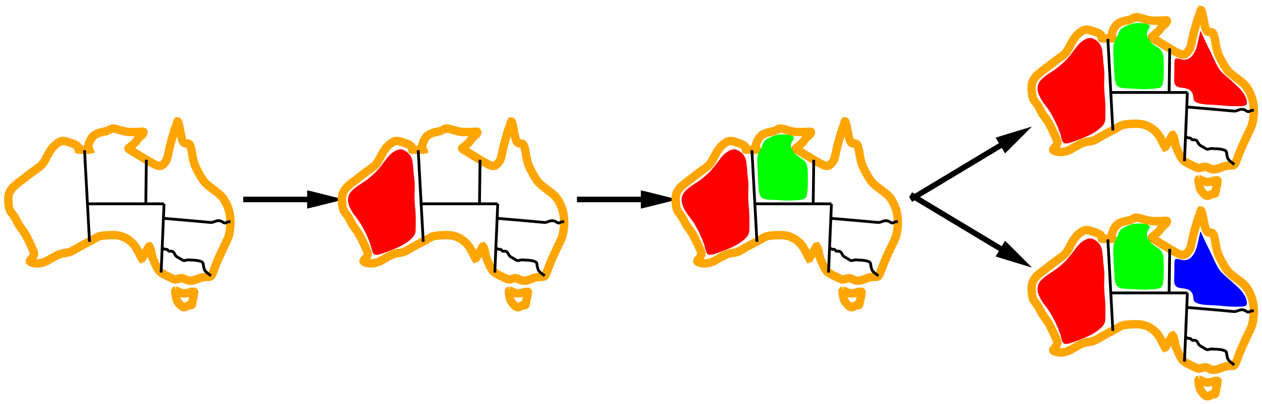
Australia example:



2. In what order should its values be tried?

Least constraining value heuristic:

◇ *Once you've selected a variable*, choose the least constraining *value*: the value that rules out the fewest values in the remaining variables

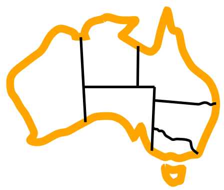


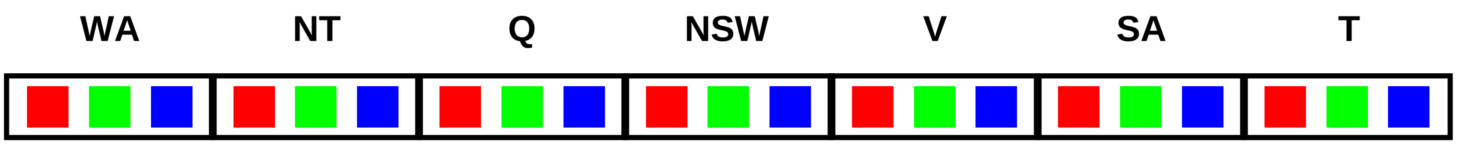
Combining these heuristics makes 1000 queens feasible

3. Can we detect inevitable failure early?

Forward checking

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

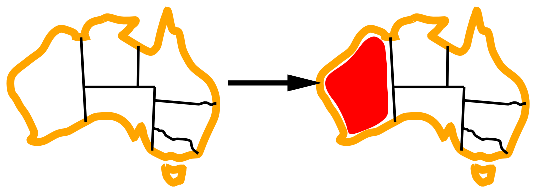


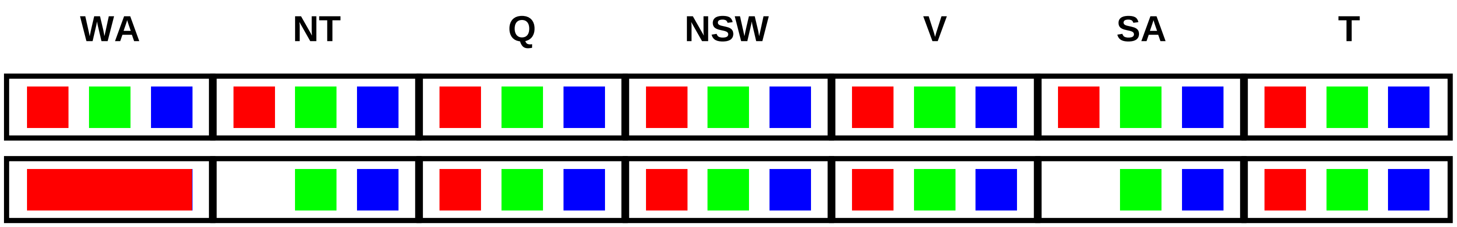


3. Can we detect inevitable failure early?

Forward checking

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

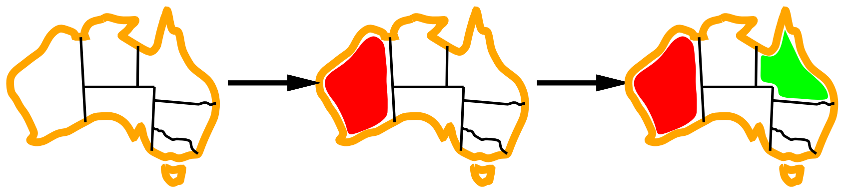




3. Can we detect inevitable failure early?

Forward checking

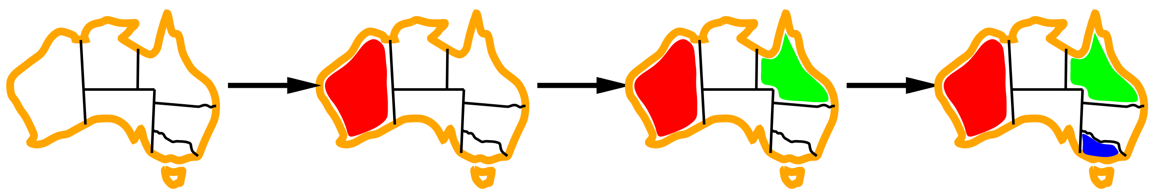
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

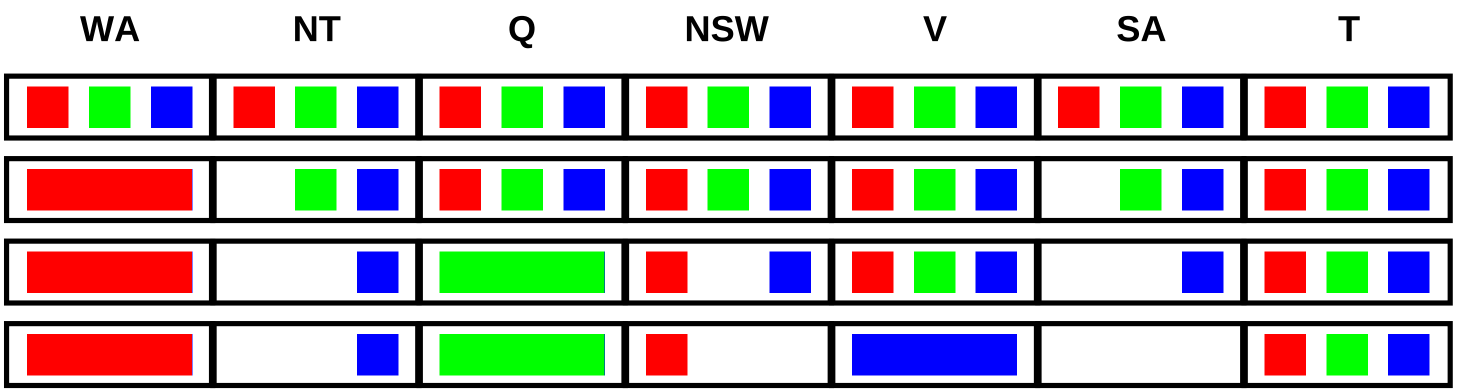


3. Can we detect inevitable failure early?

Forward checking

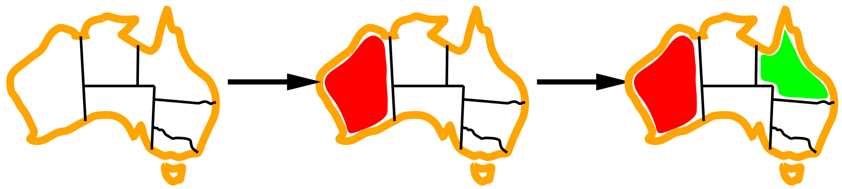
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





3. Can we detect inevitable failure early?

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

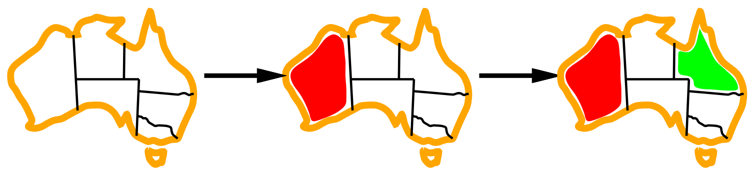
Constraint propagation

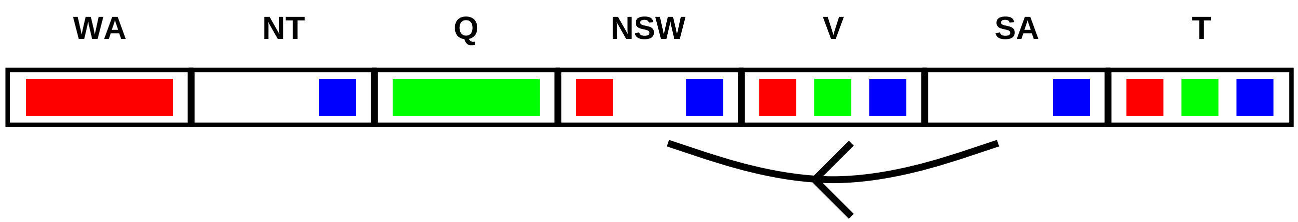
Arc consistency

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 is **some** allowed y

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





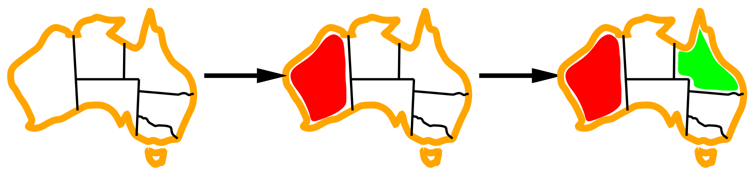
Constraint propagation

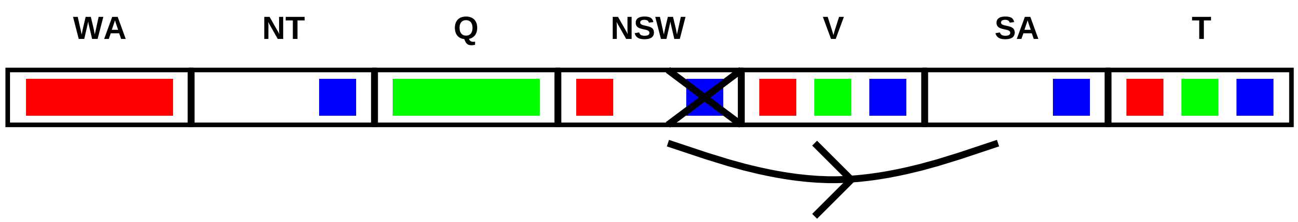
Arc consistency

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 is **some** allowed y

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





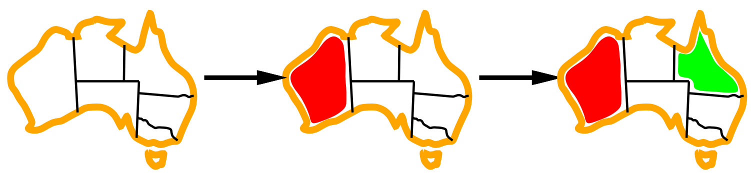
Constraint propagation

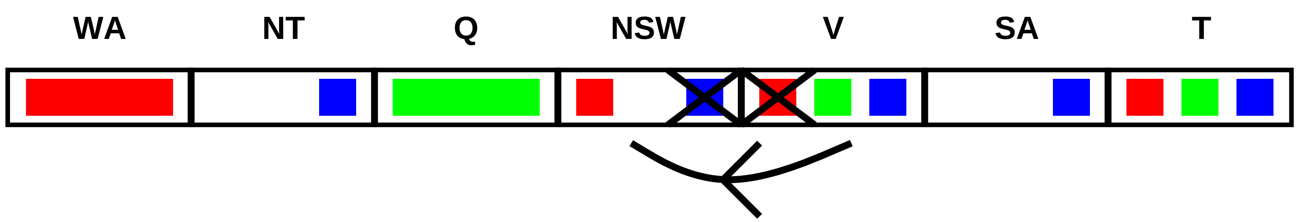
Arc consistency

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 is **some** allowed 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

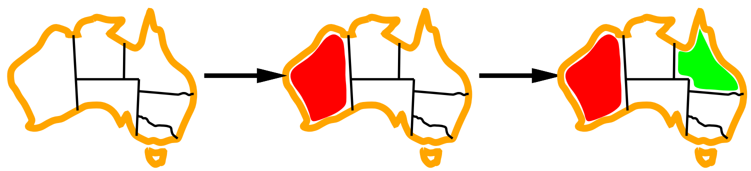
Constraint propagation

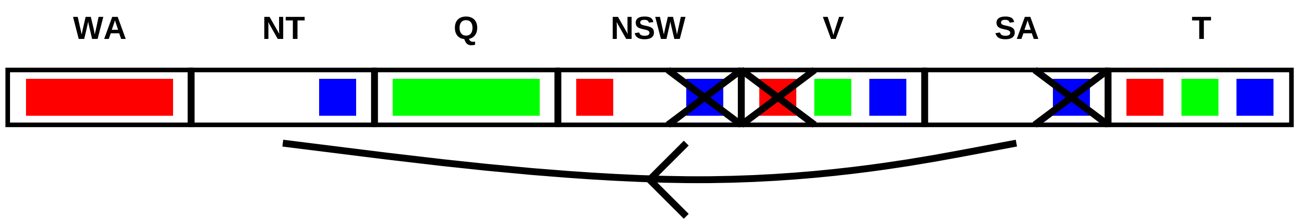
Arc consistency

For each constraint on X and Y, consider two arcs: $X \rightarrow Y$ and $Y \rightarrow X$

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

Make $X \rightarrow Y$ 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*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

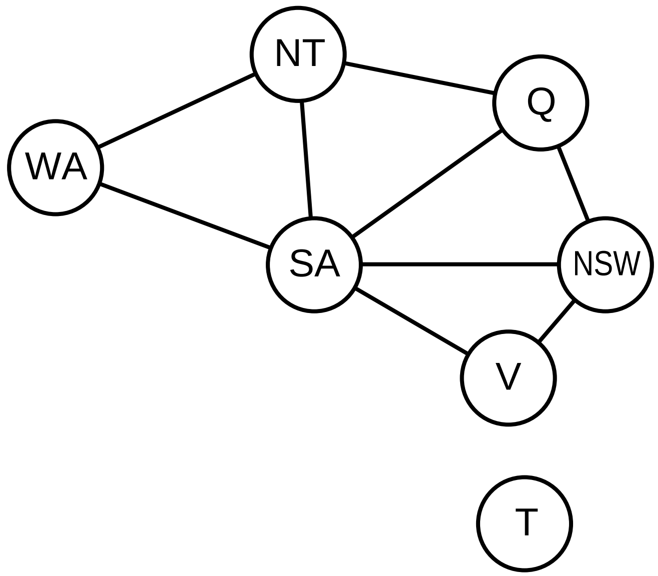
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ Remove-First(*queue*)
    if Remove-Inconsistent-Values(X_i, X_j) **then**
      for each X_k **in** Neighbors[X_i] **do**
        add (X_k, X_i) to *queue*

---

function Remove-Inconsistent-Values(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** Domain[X_i] **do**
    if no value y in Domain[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from Domain[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

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

4. How to take advantage of problem structure?



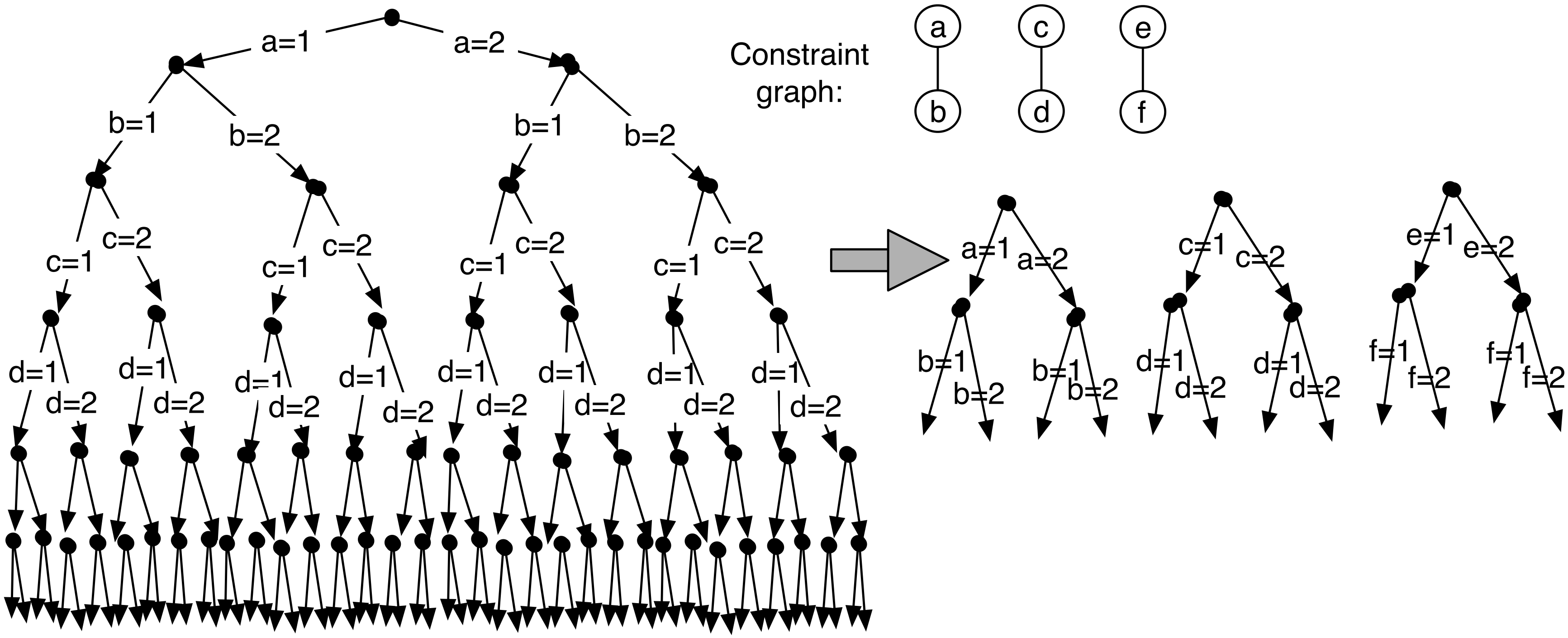
Worst-case number of leaf nodes is 3^7

But Tasmania and mainland are *independent subproblems*
Identifiable as *connected components* of constraint graph

Handle them separately $\Rightarrow$
one tree with at most 3^6 leaves, one with at most 3 leaves
Can solve this nearly 3 times as fast

4. How to take advantage of problem structure?

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



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

If we divide into 3 independent subproblems of equal size, then
worst-case number of leaf nodes is 12
$\Rightarrow$ more than 5 times as fast

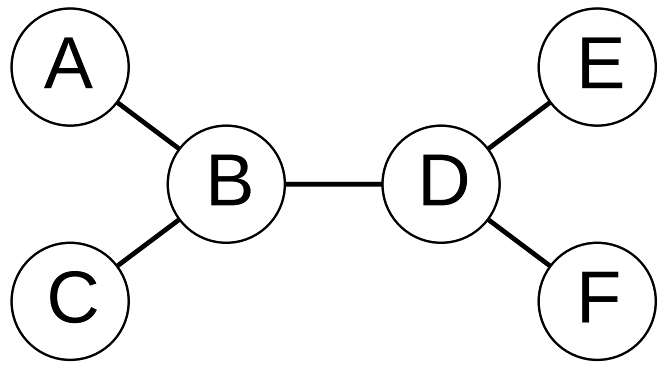
4. How to take advantage of problem structure?

- 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 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

  $4 \cdot 2^{20}$ = 0.4 seconds

Tree-structured CSPs



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

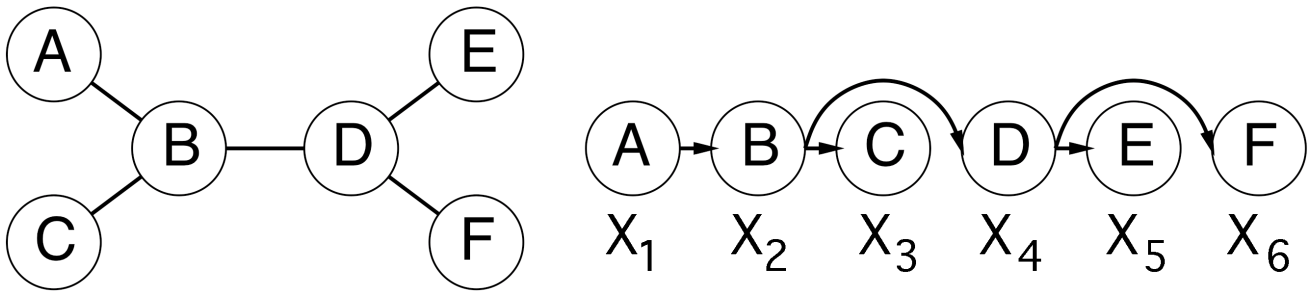
Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning
good 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 such that every node's parent precedes it in the ordering

 ♢ like a topological sort



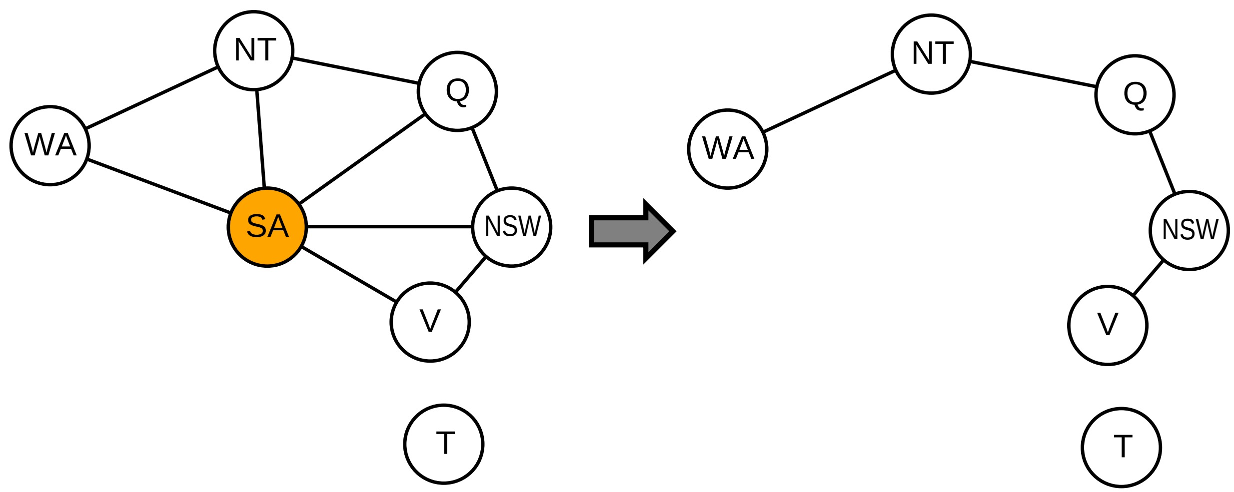
2. For j from n down to 2, apply arc-consistency
 Note that the arcs only point one way
  Remove-Inconsistent-Values$(Parent(X_j), X_j)$

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

Iterative algorithms for CSPs

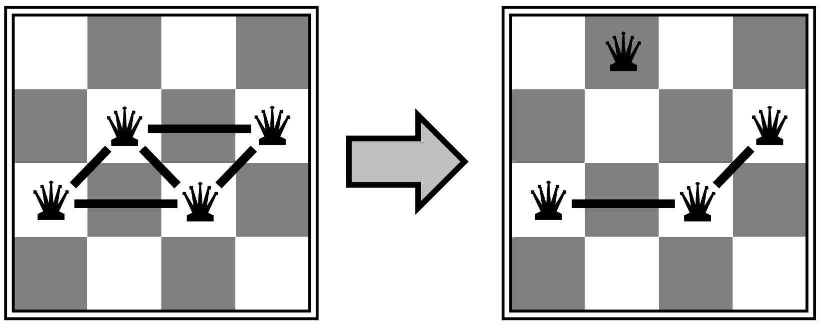
Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

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:

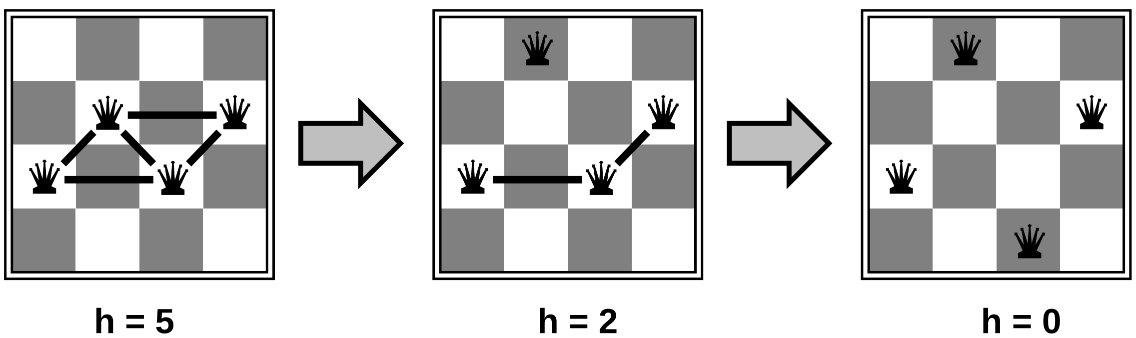


Iterative algorithms for CSPs

Variable selection: randomly select any conflicted variable

Value selection by *min-conflicts* heuristic:
choose value that violates the fewest constraints
i.e., 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

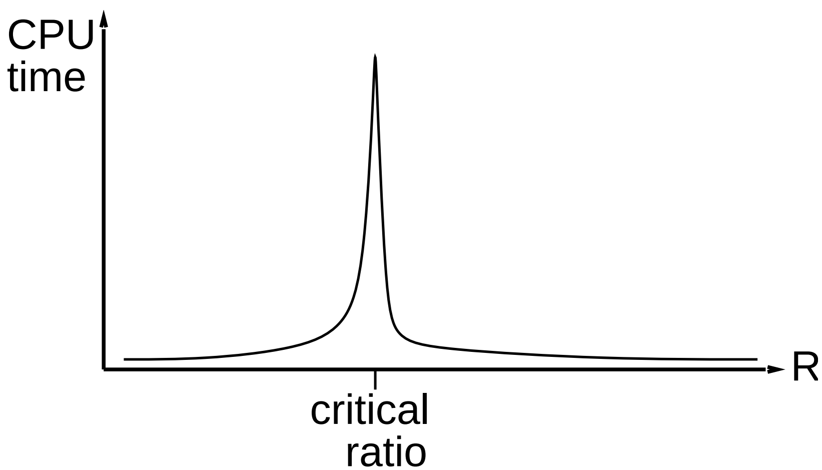


Performance of min-conflicts

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

$R =$ 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

Homework

Problems 5.2, 5.6, 5.8, 6.1(b,c,d,e)
10 points each, 40 points total

Due on March 4

March 4 was the “late date” (5-point penalty) for Project 1
I’ll change the “late date” to March 6,
to make it different from the homework’s due date

I’m not changing Project 1’s due date
It still is due on March 2