Constraint Satisfaction

Philipp Koehn

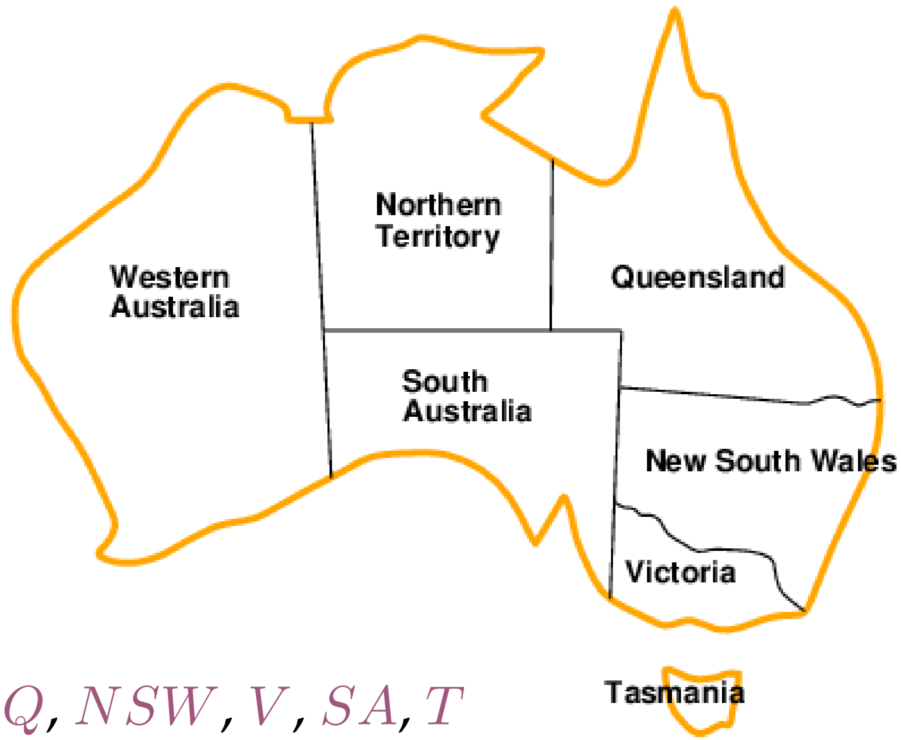
22 February 2024

Outline

- Constraint satisfaction problems (CSP) examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs

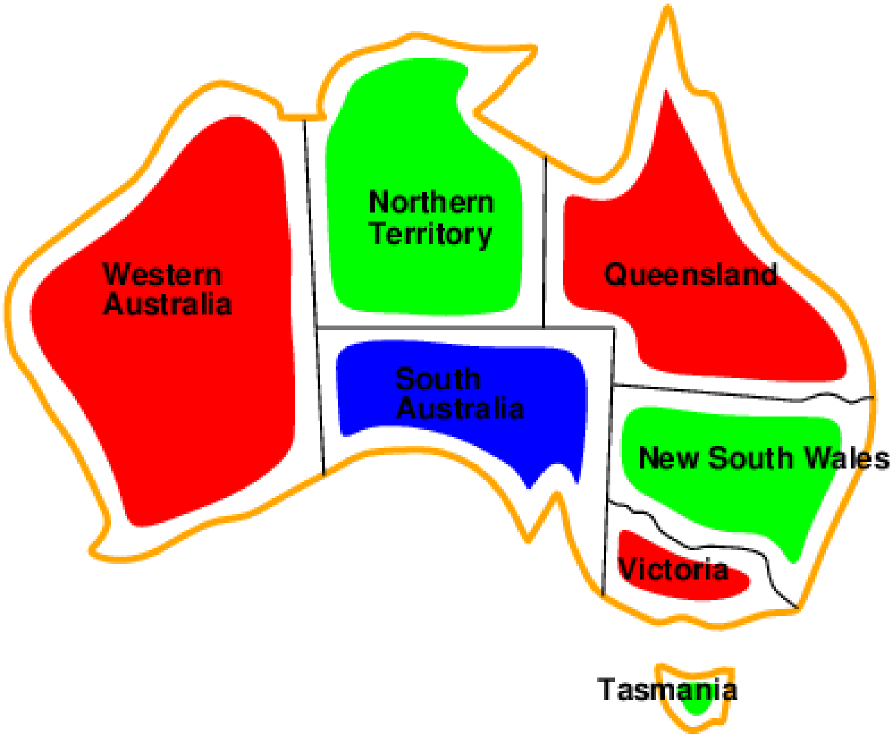
examples

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



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

Constraint Satisfaction Problems (CSPs)

- Previously: generic search
  - state is a "black box"
  - state must support 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**

- We will look at useful **general-purpose** algorithms with more power than standard search algorithms

Varieties of CSPs

- Discrete variables
  - finite domains; size $d \implies O(d^n)$ complete assignments
    - 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, nonlinear undecidable
- Continuous variables
  - e.g., start/end times for Hubble Telescope observations
  - linear constraints solvable in poly time by LP methods

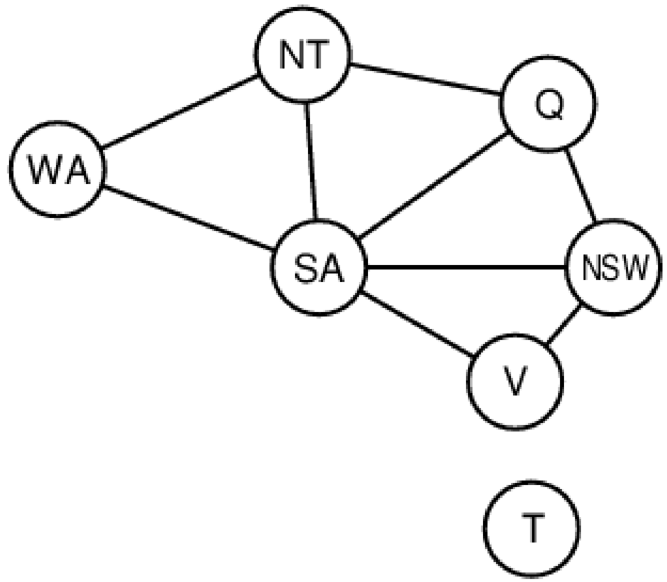
Varieties of Constraints

- Unary constraints involve a single variable,
  e.g., $SA \neq green$
- Binary constraints involve pairs of variables,
  e.g., $SA \neq WA$
- Higher-order constraints involve 3 or more variables,
  e.g., cryptarithmetic column constraints
- Preferences (soft constraints), e.g., red is better than $green$
  often representable by a cost for each variable assignment
  → constrained optimization problems

Map Coloring Constraint Graph

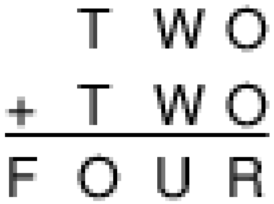
- Binary CSP: each constraint relates at most 2 variables (i.e., colors of 2 states)

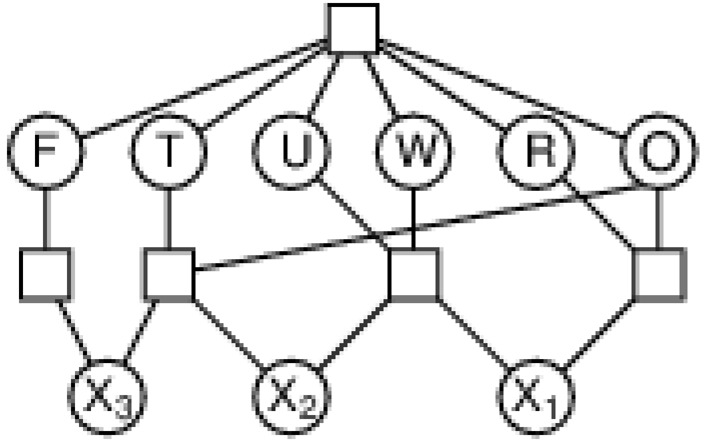
- Constraint graph: nodes are variables, arcs show constraints



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

Example: Cryptarithmetic





- Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
- Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints
  $alldiff(F, T, U, W, R, O)$
  $O + O = R + 10 \cdot X_1$, etc.

Example: Sudoku

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

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- No same number in row, column, small square
- Easily formulated as CSP with *alldiff* constraints
- Can be quickly solved with standard CSP solvers

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

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

backtracking search

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, $\emptyset$
  - Successor function: assign a value to an unassigned variable that does not conflict with current assignment.
    $\Longrightarrow$ fail if no legal assignments (not fixable!)
  - Goal test: the current assignment is complete

- Note
  - This is the same for all CSPs! ☺
  - Every solution appears at depth n with n variables
    $\Longrightarrow$ use depth-first search
  - $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves ☹

Backtracking Search

- Variable assignments are commutative, i.e.,
  $[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

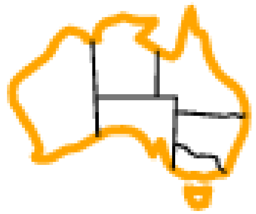
- Only need to consider assignments to a single variable at each node
  $\Longrightarrow$ $b = d$ and there are d^n leaves

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

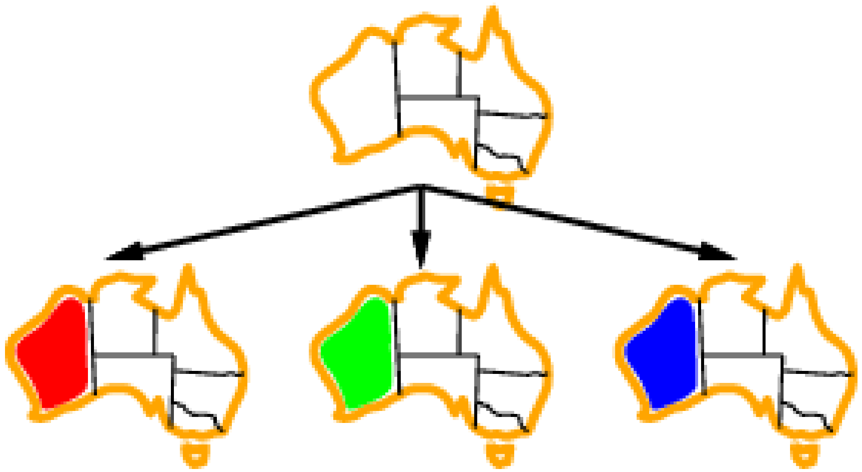
- Backtracking search is the basic uninformed algorithm for CSPs

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

Backtracking Example

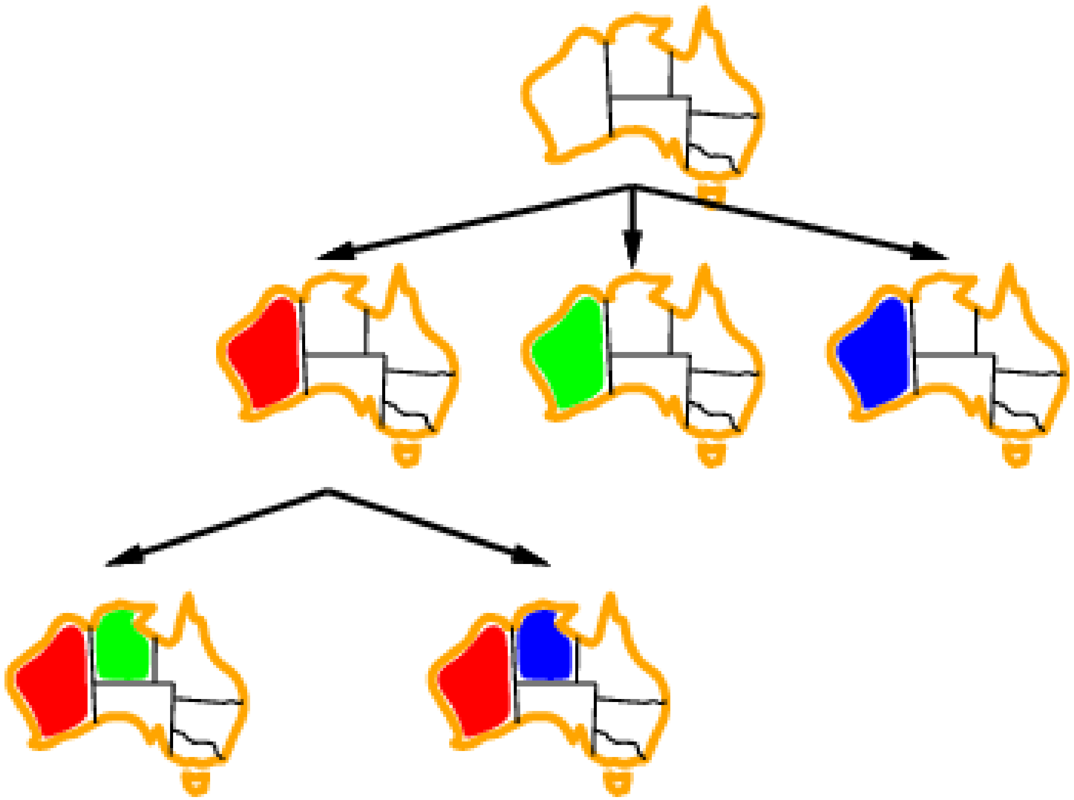


Backtracking Example



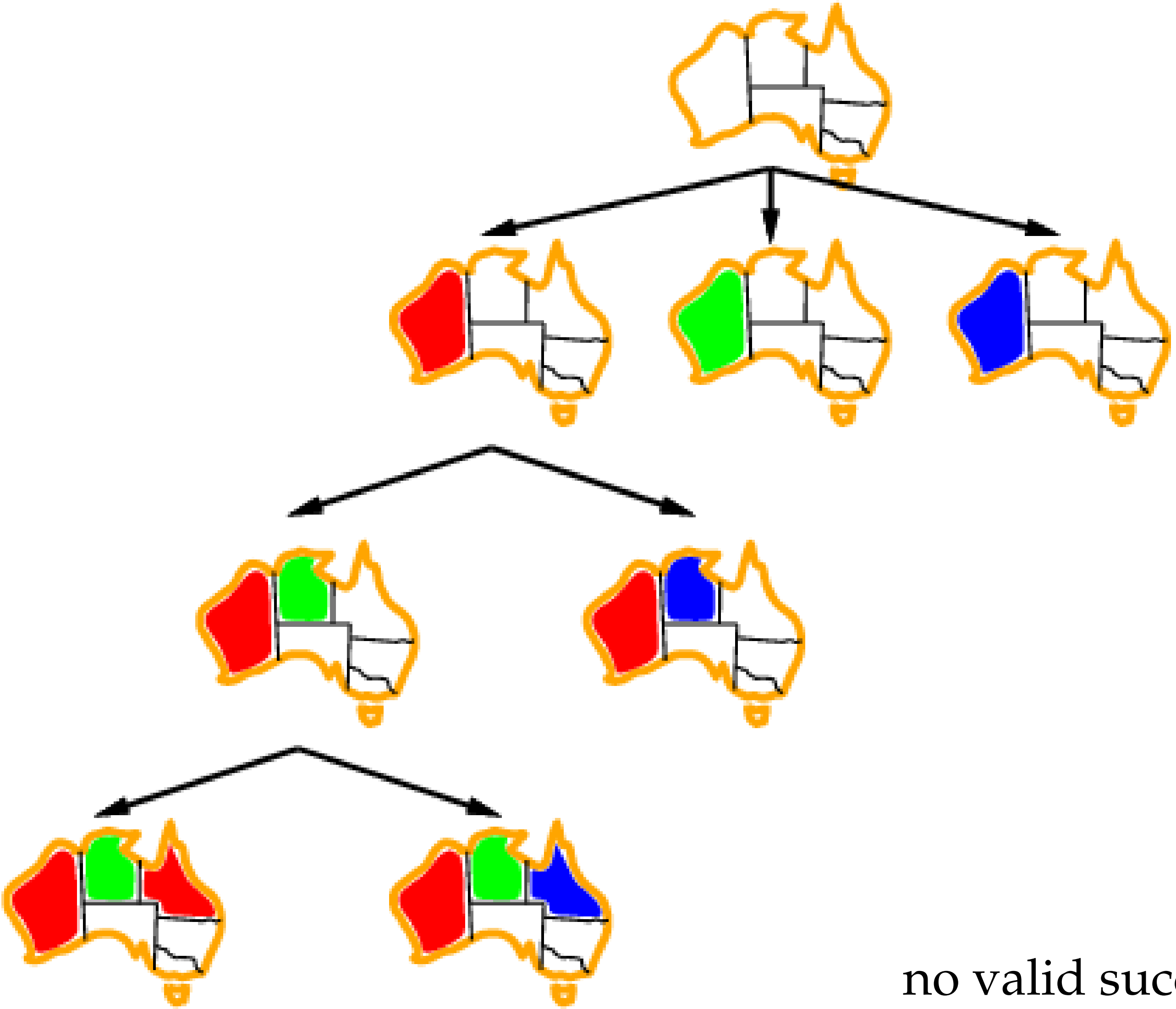
Recall: assign variables in fixed order

Backtracking Example



Only two valid choices (red violates constraint)

Backtracking Example



And so it continues...

full assignmen: done

no valid successor: fail → backtrack

Backtracking Search

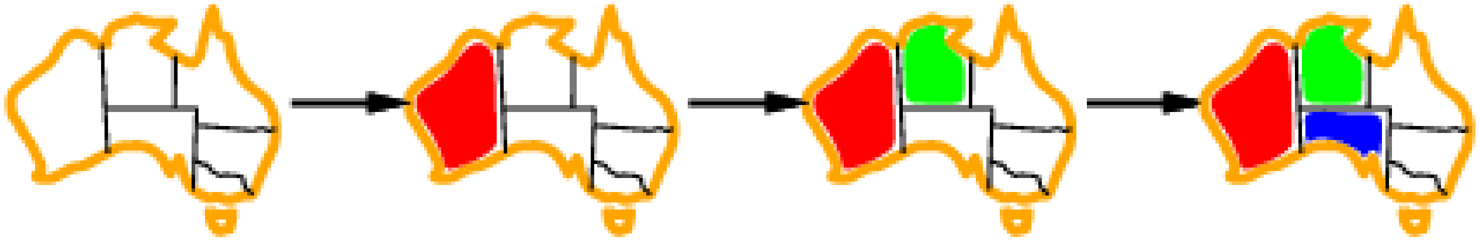

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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

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

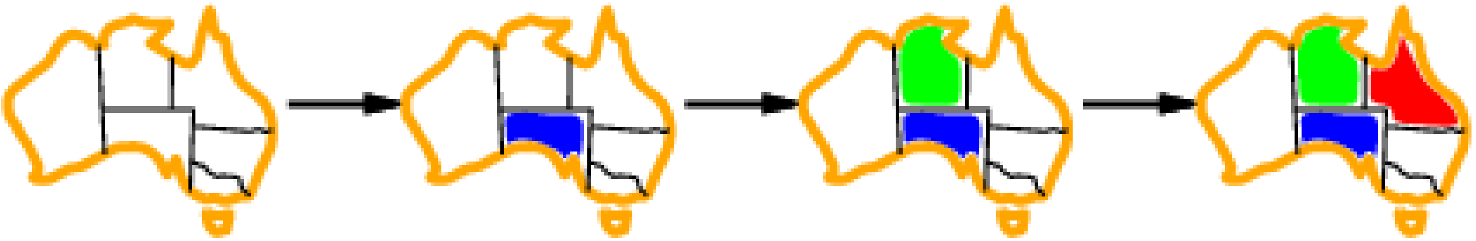


3 choices | 2 choices | 1 choice | ...

Degree Heuristic

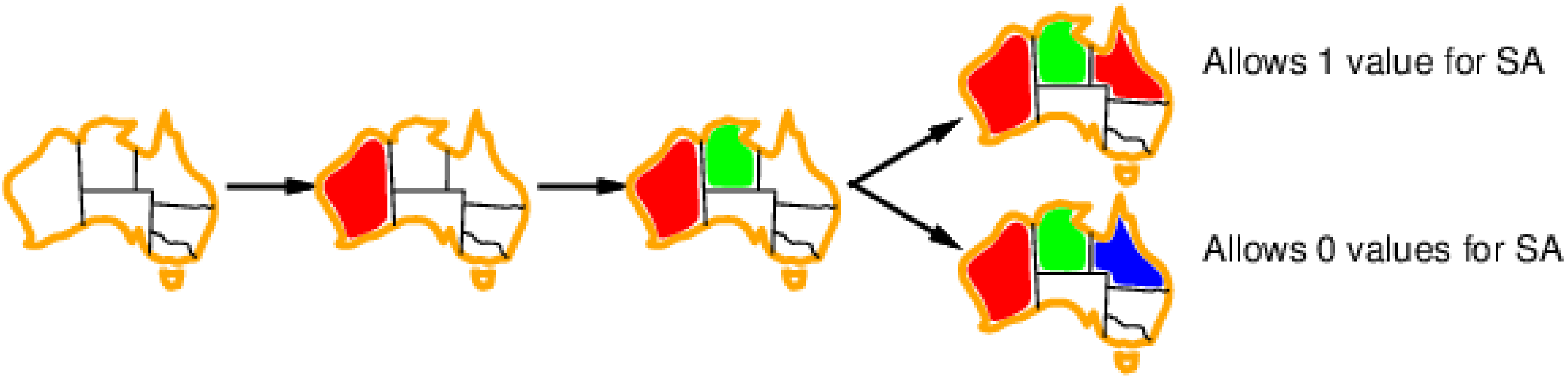
- Tie-breaker among MRV variables

- 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

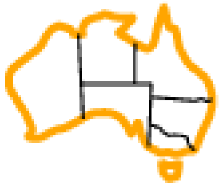


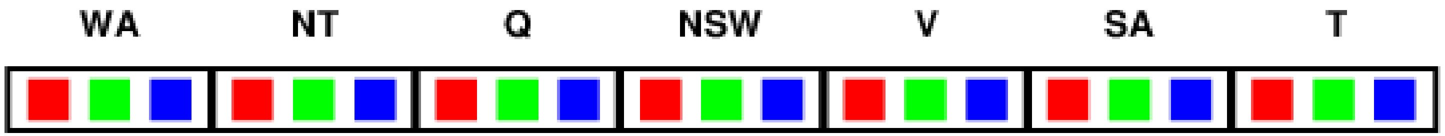
- Combining these heuristics makes 1000 queens feasible

constraint propagation

Forward Checking

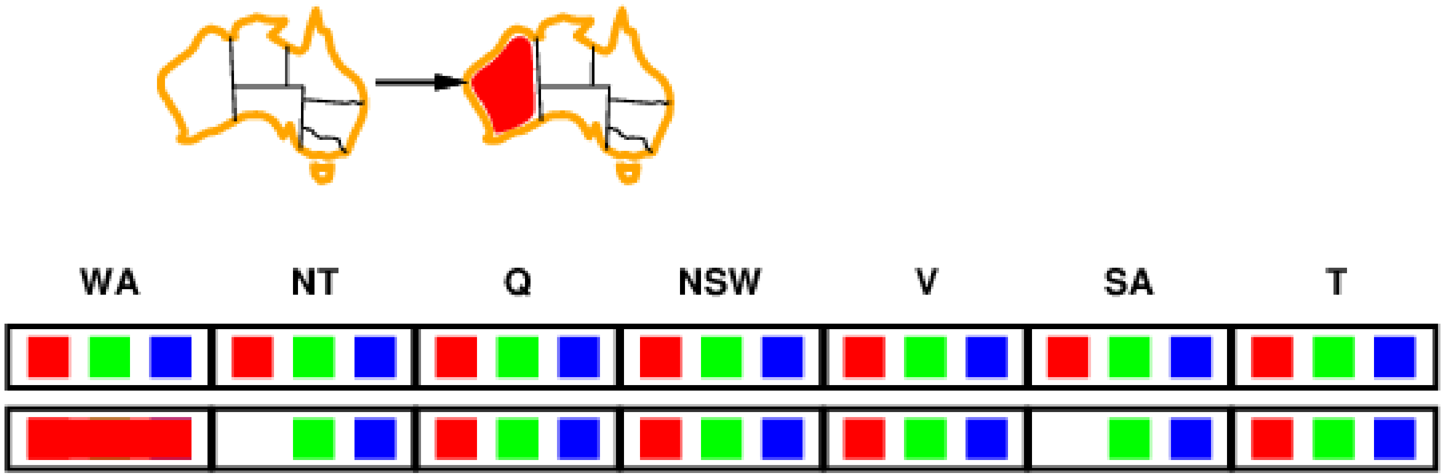
- Idea: Keep track of remaining legal values for unassigned variables
  Terminate search when any variable has no legal values





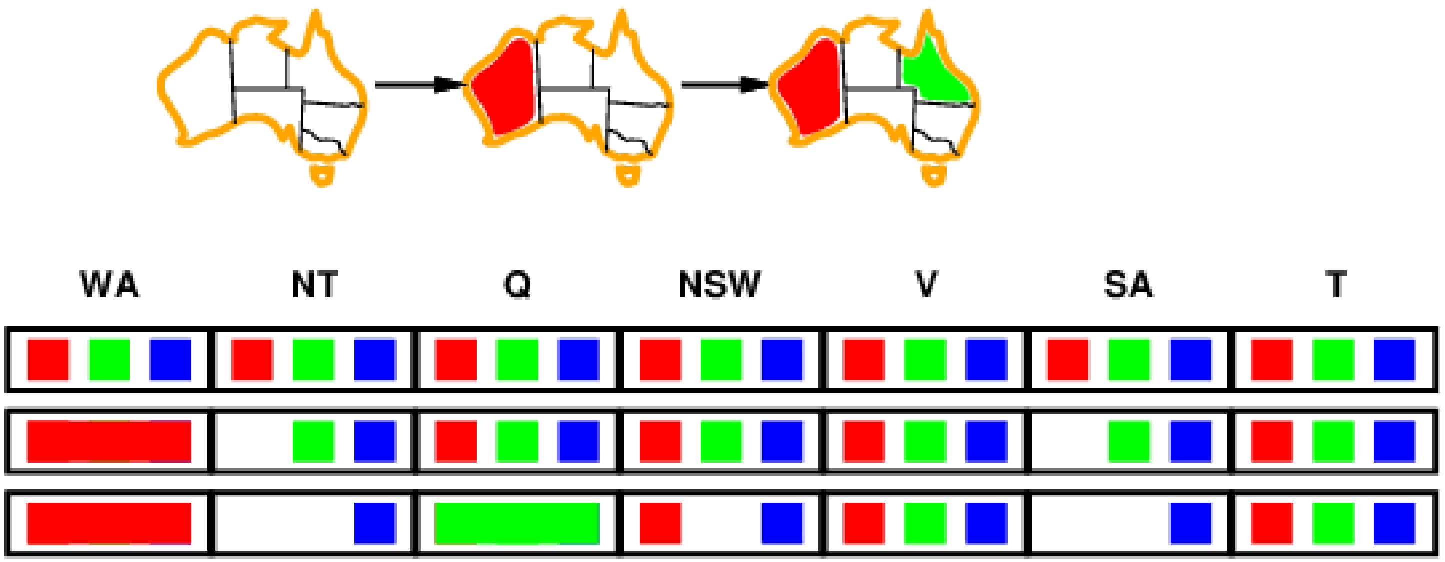
Forward Checking

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



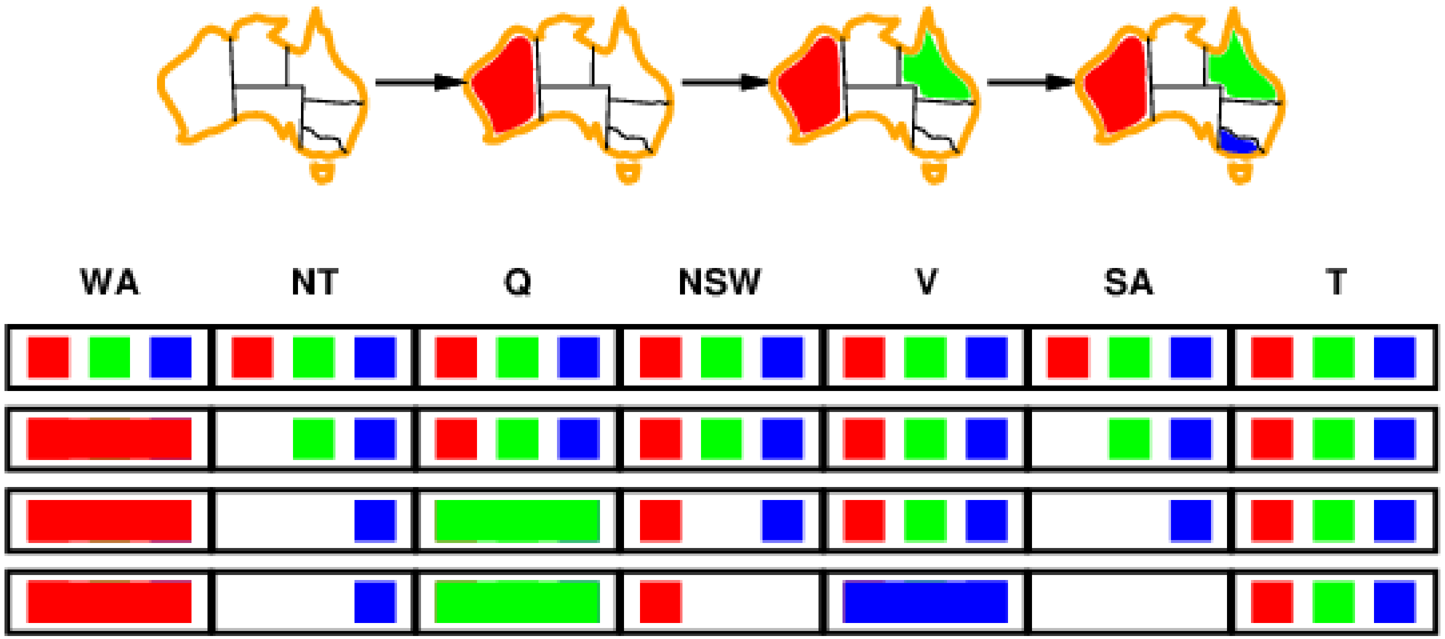
Forward Checking

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



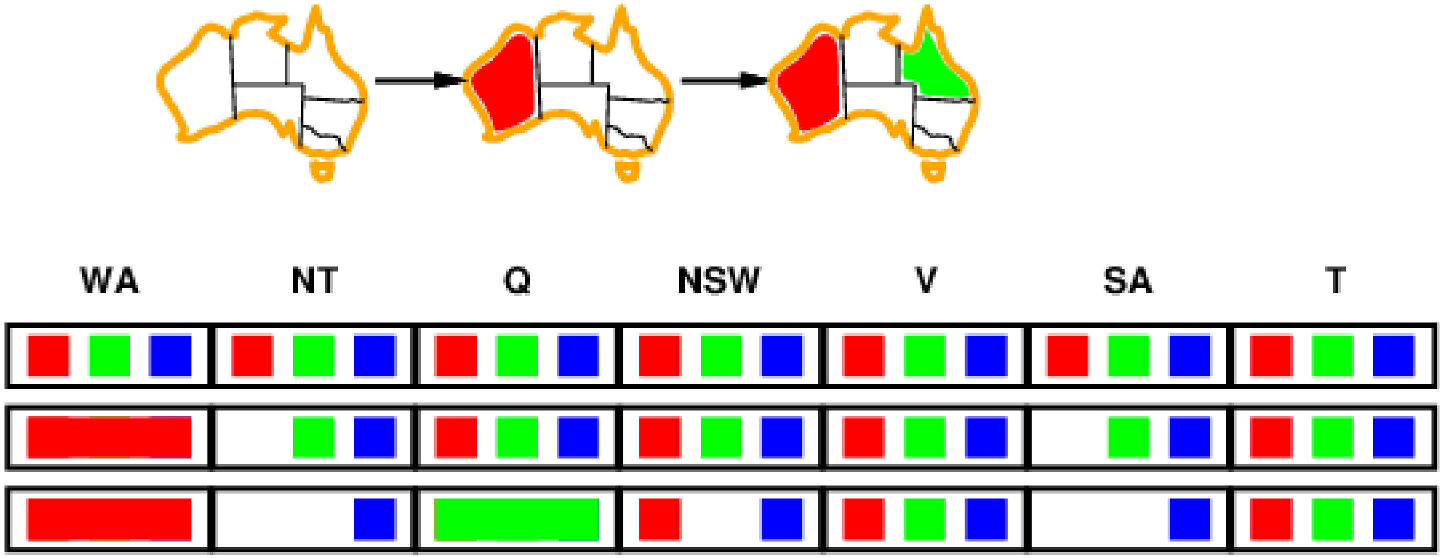
Forward Checking

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



Constraint Propagation

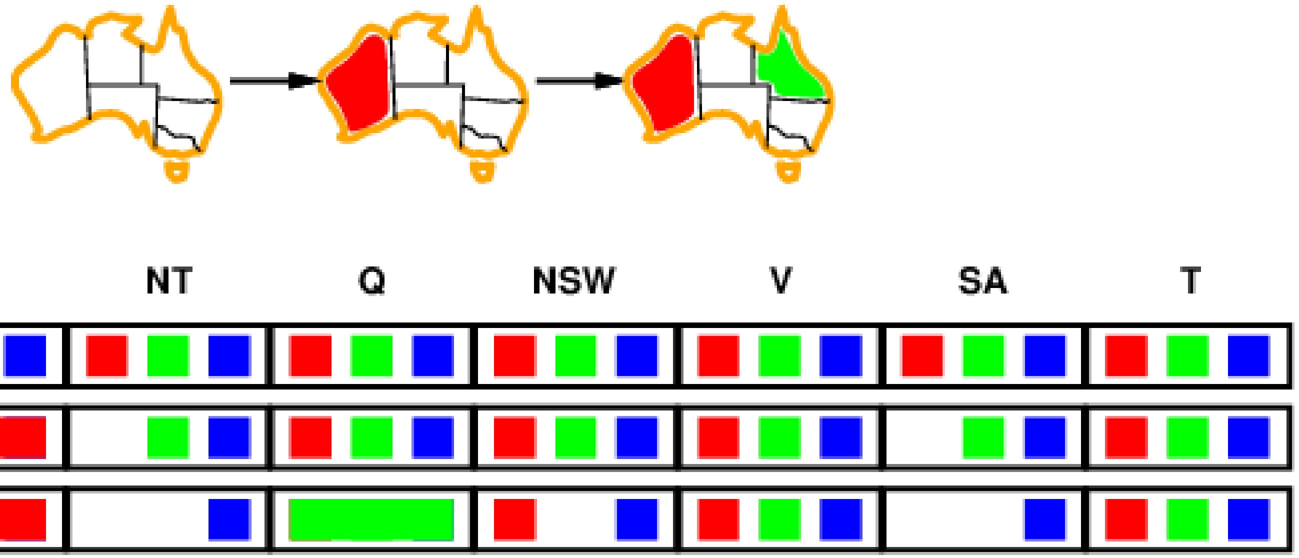
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

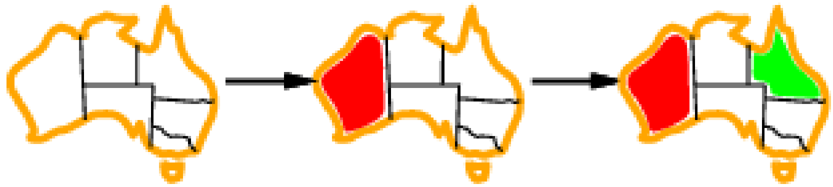
Arc Consistency

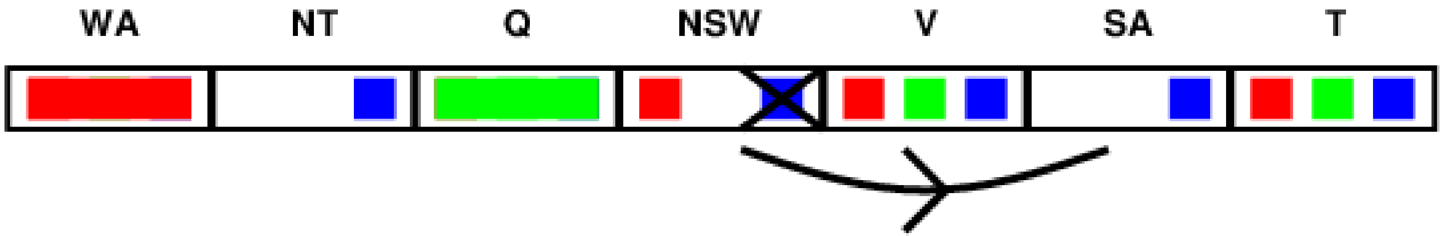
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for **every** value x of X there is **some** allowed y



Arc Consistency

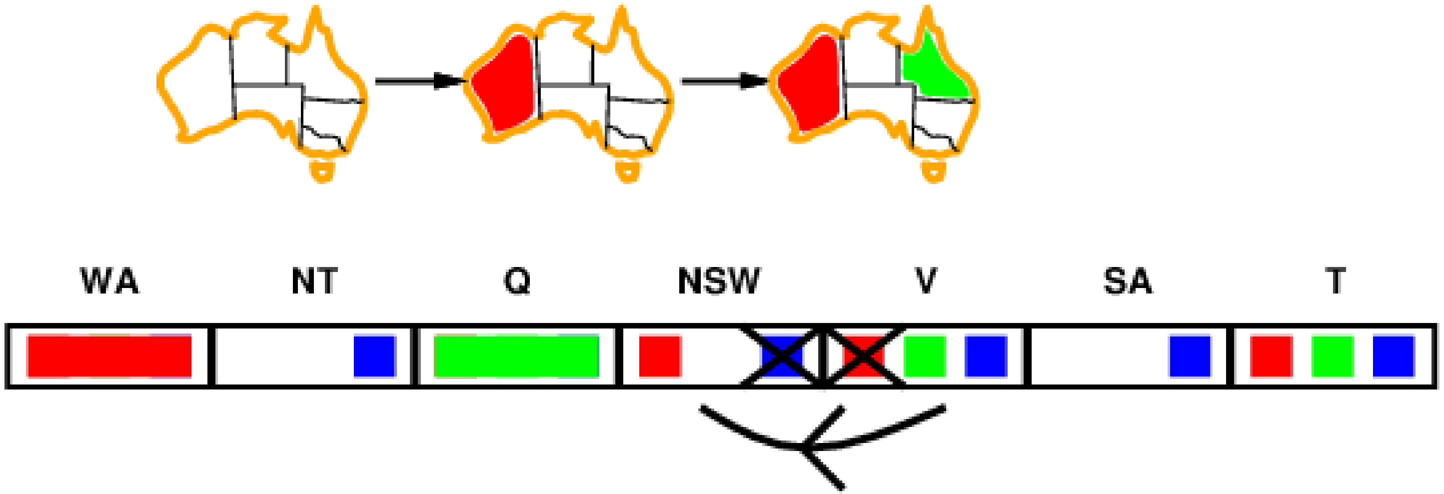
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for **every** value x of X there is **some** allowed y





Arc Consistency

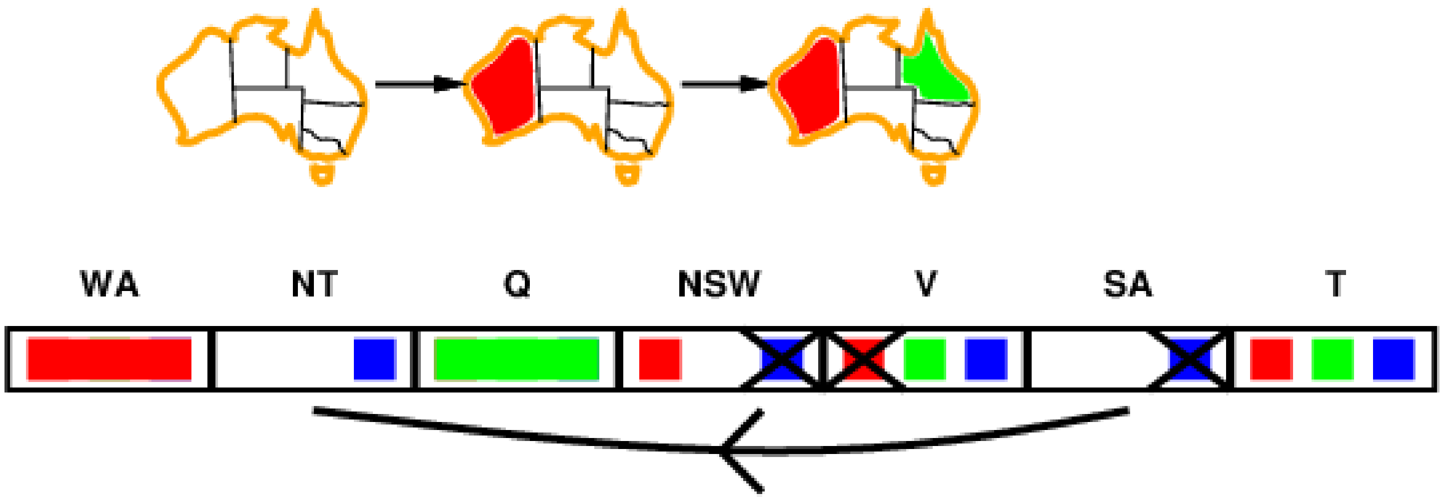
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for **every** value x of X there is **some** allowed y



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

Arc Consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ 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 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 {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

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

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

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

Path Consistency

- Arc consistency check removes some possible values
  - reduces search space
  - may already solve problem (each variable one value)
  - may already eliminate search state (one variable no value)

- One step further: path consistency

- Any two variable set $\{X_i, X_j\}$ is **path consistent** with third variable X_k
  if any assignment $\{X_i = a, X_j = b\}$ there is an assignment for X_k
  that fulfills constraints for $\{X_i, X_k\}$ and $\{X_j, X_k\}$

k-Consistency

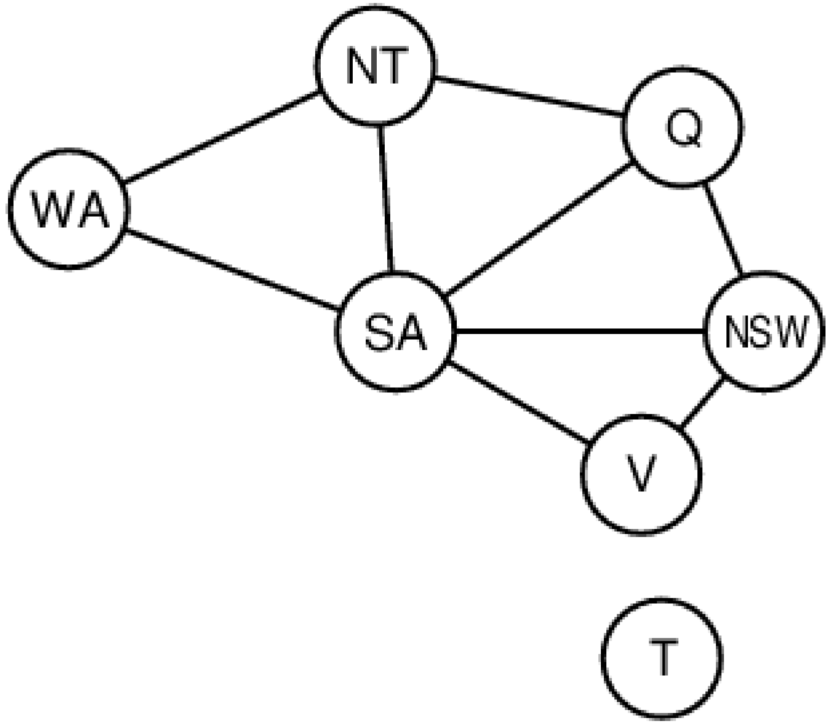
- Node consistency = check all unary constraints
- Arc consistency = check all binary constraints
- Path consistency = check all constraints for each 3-variable subset
- k-consistency = check all constraints for each k-variable subset

- But: checking all subsets for high k increasing computationally expensive

⇒ not done in practice

problem structure

Problem Structure



- Tasmania and mainland are independent subproblems

- Identifiable as connected components of constraint graph

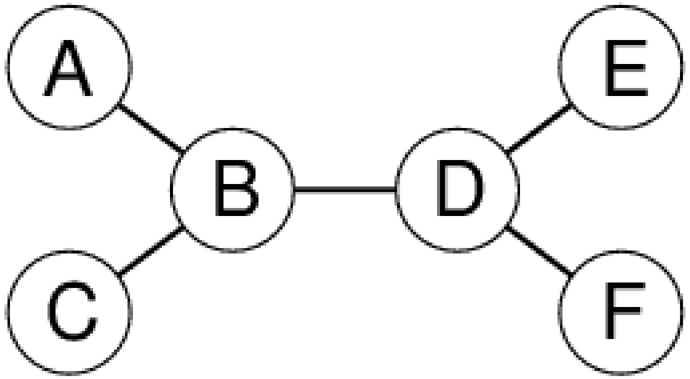
Problem Structure

- 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

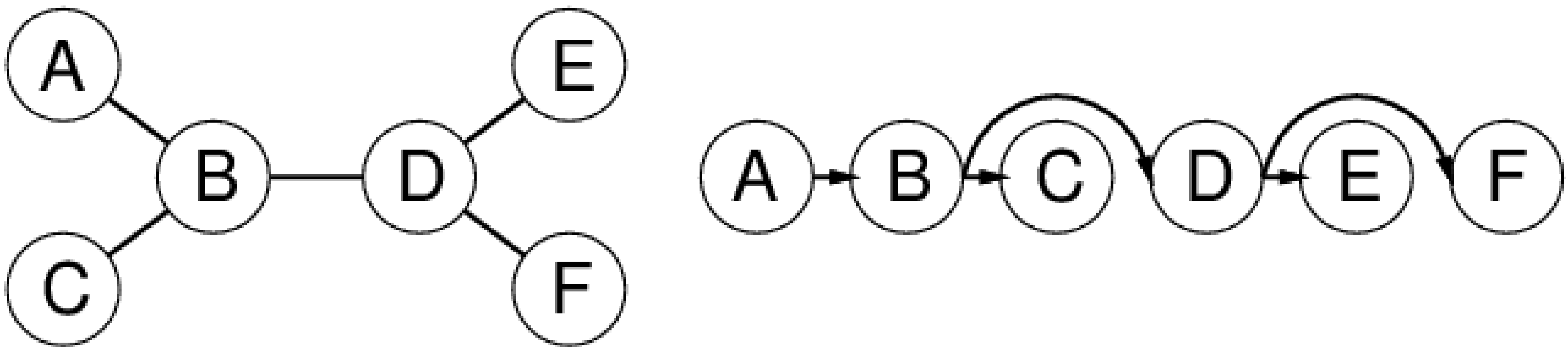
Tree-Structured CSPs



- Theorem: if constraint graph has no loops, 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 such that every node's parent precedes it in the ordering

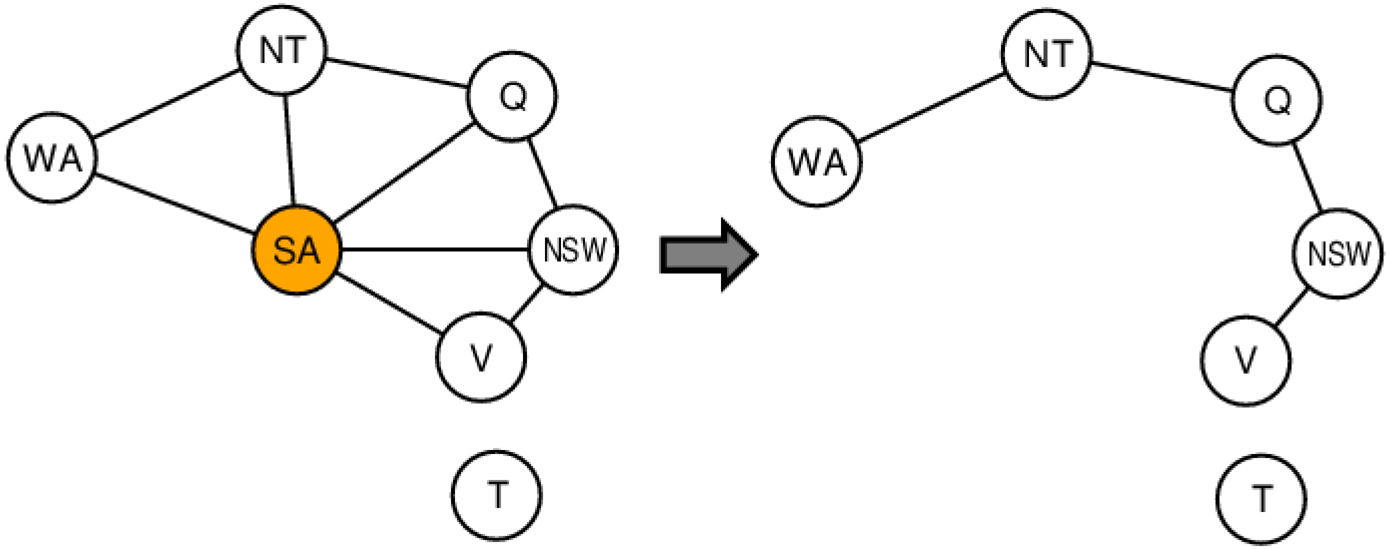


2. For j from n down to 2, apply REMOVEINCONSISTENT$(Parent(X_j), X_j)$

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

Nearly Tree-Structured CSPs

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

local search

Iterative Algorithms for CSPs

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

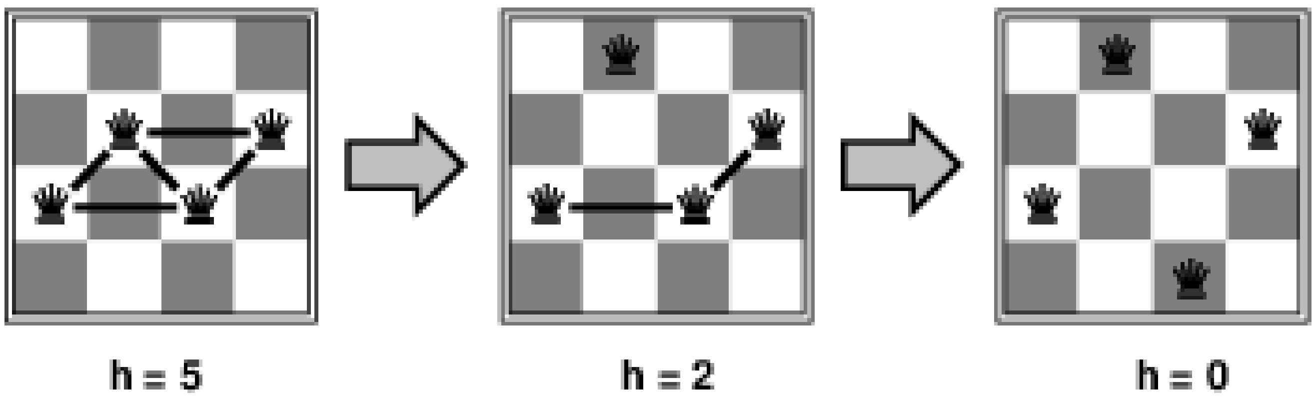
- To apply to CSPs
  - allow states with unsatisfied constraints
  - operators **reassign** variable values

- 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

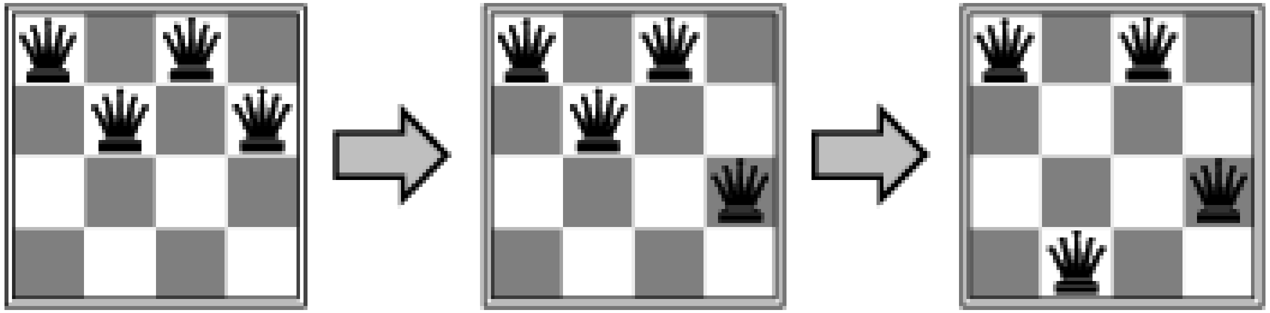
Example: 4-Queens

- 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



Example: 4-Queens as a CSP

- Assume one queen in each column. Which row does each one go in?
- Variables Q_1, Q_2, Q_3, Q_4
- Domains $D_i = \{1, 2, 3, 4\}$



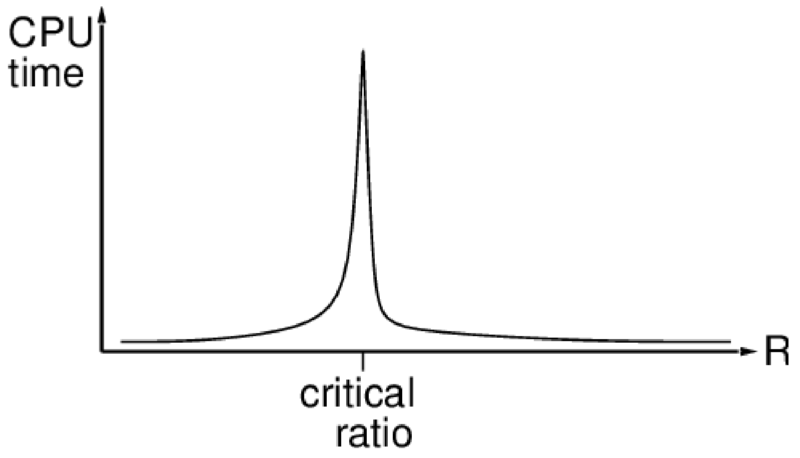
- Constraints
  $Q_i \neq Q_j$ (cannot be in same row)
  $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)
- Translate each constraint into set of allowable values for its variables
- E.g., values for (Q_1, Q_2) are $(1,3)$ $(1,4)$ $(2,4)$ $(3,1)$ $(4,1)$ $(4,2)$

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

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



Summary

- CSPs are a special kind of problem:
  states defined by values of a fixed set of variables
  goal test defined by constraints on variable values

- **Backtracking** = depth-first search with one variable assigned per node

- **Variable ordering** and **value selection** heuristics help significantly

- **Forward checking** prevents assignments that guarantee later failure

- Constraint propagation (e.g., **arc consistency**) does additional work to constrain values and detect inconsistencies

- The CSP representation allows analysis of **problem structure**

- **Tree-structured CSPs** can be solved in linear time

- **Iterative min-conflicts** is usually effective in practice