Constraint Satisfaction

Russell & Norvig Ch. 6.1-6.4

Informal Definition of CSP

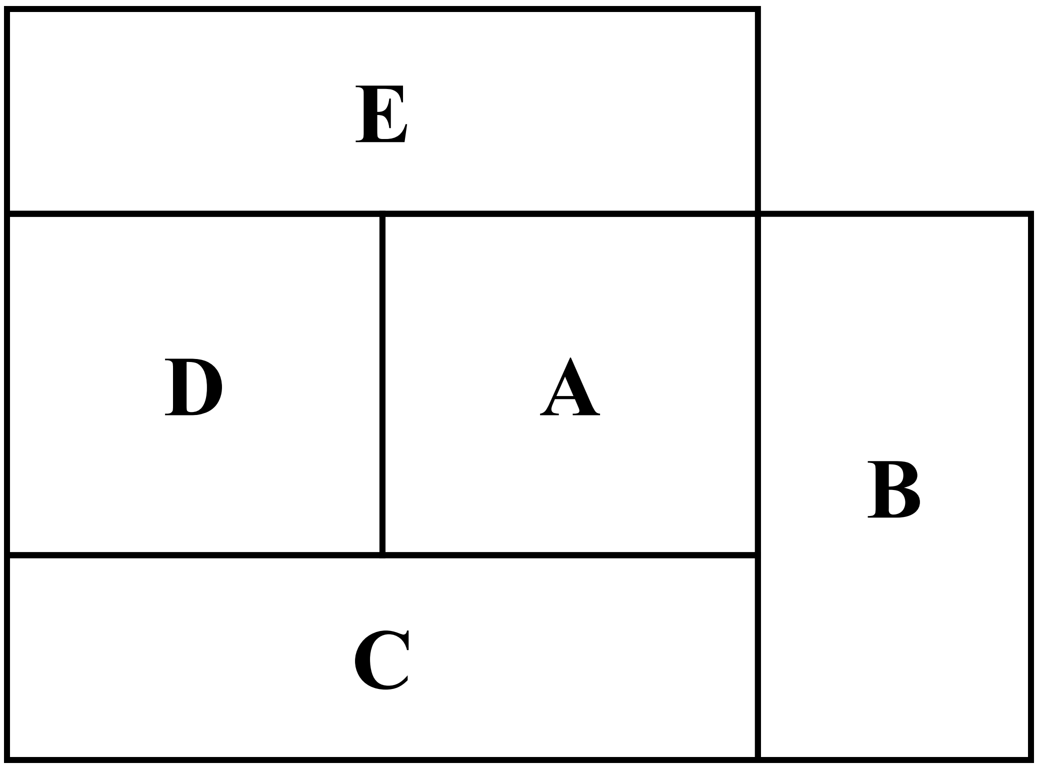
- CSP = Constraint Satisfaction Problem
- Given
  - (1) a finite set of variables
  - (2) each with a domain of possible values (often finite)
  - (3) a set of constraints that limit the values the variables can take on
- A **solution** is an assignment of a value to each variable such that the constraints are all satisfied.
- Tasks might be to decide if a solution exists, to find a solution, to find all solutions, or to find the "best solution" according to some metric (objective function).

Today's Class

- Constraint Processing / Constraint Satisfaction Problem (CSP) paradigm
- Algorithms for CSPs
  - Backtracking (systematic search)
  - Constraint propagation (k-consistency)
  - Variable and value ordering heuristics
  - Intelligent backtracking

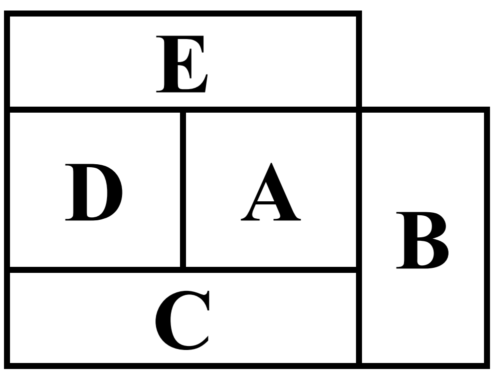
Informal Example: Map Coloring

- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.

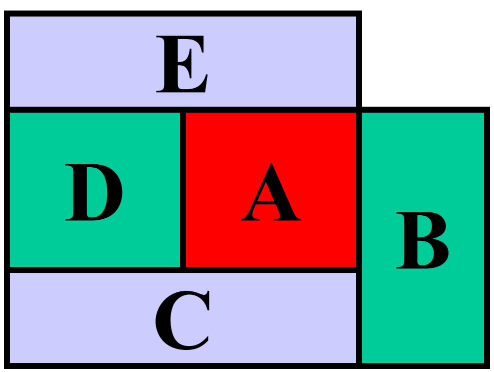


Map Coloring II

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: A≠B, A≠C,A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
- One solution: A=red, B=green, C=blue, D=green, E=blue



=>



Formal Definition of a Constraint Network (CN)

A constraint network (CN) consists of

- a set of variables $X = \{x_1, x_2, \ldots x_n\}$
  - each with an associated domain of values $\{d_1, d_2, \ldots d_n\}$.
  - the domains are typically finite
- a set of constraints $\{c_1, c_2 \ldots c_m\}$ where
  - each constraint defines a predicate which is a relation over a particular subset of X.
  - e.g., C_i involves variables $\{X_{i1}, X_{i2}, \ldots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots D_{ik}$
- **Unary** constraint: only involves one variable
- **Binary** constraint: only involves two variables

Example (Class Scheduling)

- Given a list of courses to be taught, classrooms available, time slots, and professors who can teach certain courses, can classes be scheduled?

- Variables: Courses offered ($C_1, \ldots, C_i$), classrooms ($R_1, \ldots, R_j$), time ($T_1, \ldots, T_k$).

- Domains:
  - DC_i = {professors who can teach course i}
  - DR_j = {room numbers}
  - DT_k = {time slots}

- Constraints:
  - Maximum 1 class per room in each time slot.
  - A professor cannot teach 2 classes in the same time slot.
  - A professor cannot teach more than 2 classes.

Typical Tasks for CSP

- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CN into an equivalent CN that is easier to solve.

Solving Constraint Problems

- Systematic search
  - Generate and test
  - Backtracking
- Constraint propagation (consistency)
- Variable ordering heuristics
- Value ordering heuristics
- Backjumping and dependency-directed backtracking

Systematic Search: Backtracking

(a.k.a. depth-first search!)

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we've reached a dead end and need to backtrack to the previous variable
- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values

Problems with Backtracking

- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can explore areas of the search space that aren't likely to succeed
  - Variable ordering can help

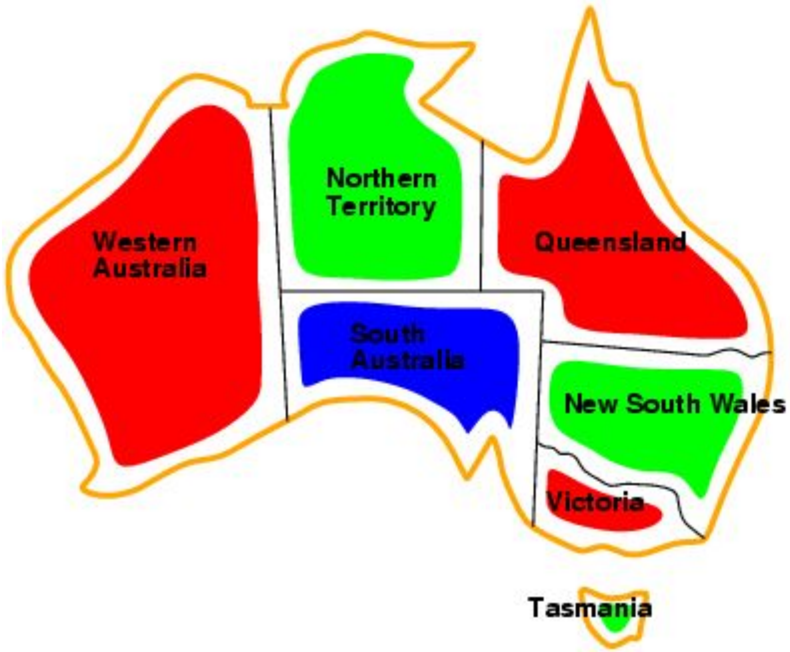
Consistency

- Node consistency
  - A node X is **node-consistent** if every value in the domain of X is consistent with X's unary constraints
  - A graph is node-consistent if all nodes are node-consistent
- Arc consistency
  - An arc (X, Y) is **arc-consistent** if, for every value x of X, there is a value y for Y that satisfies the constraint represented by the arc.
  - A graph is arc-consistent if all arcs are arc-consistent.
- To create arc consistency, we perform **constraint propagation**: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs

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 ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

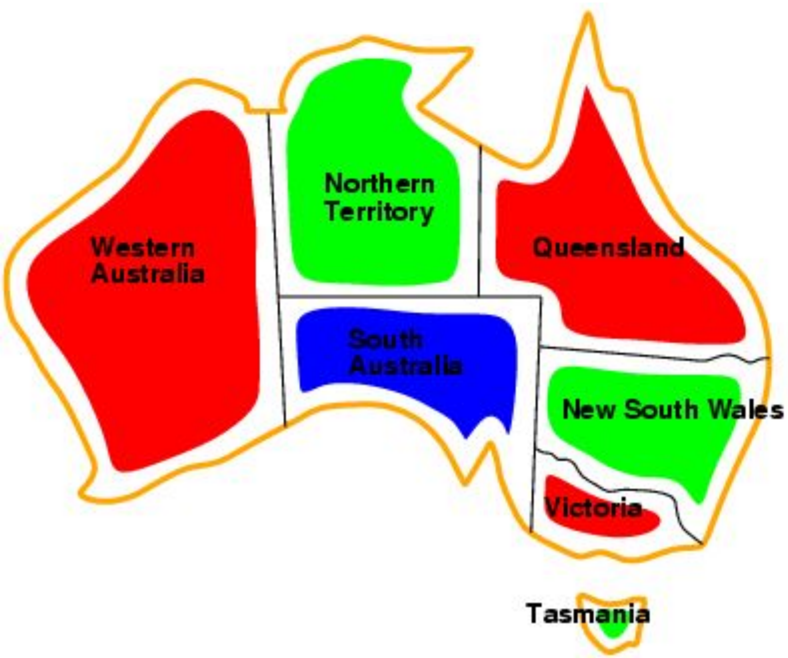
Example: Map-Coloring

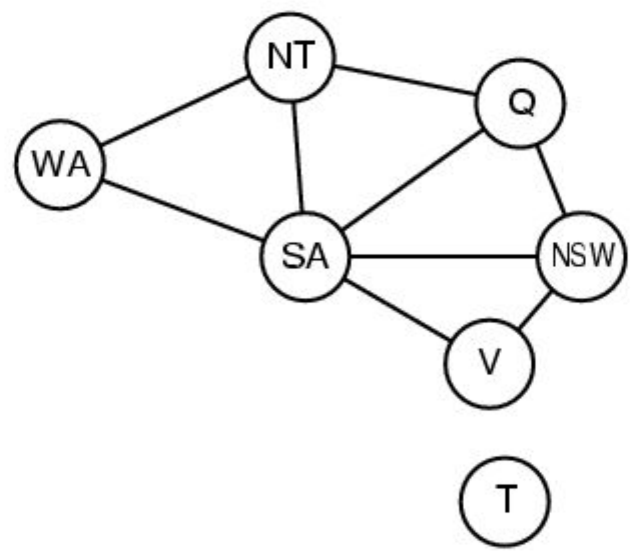


- Solutions are complete and consistent assignments
- e.g., WA = red, NT = green, Q = red, NSW = green,V = red,SA = blue,T = green

Constraint graph

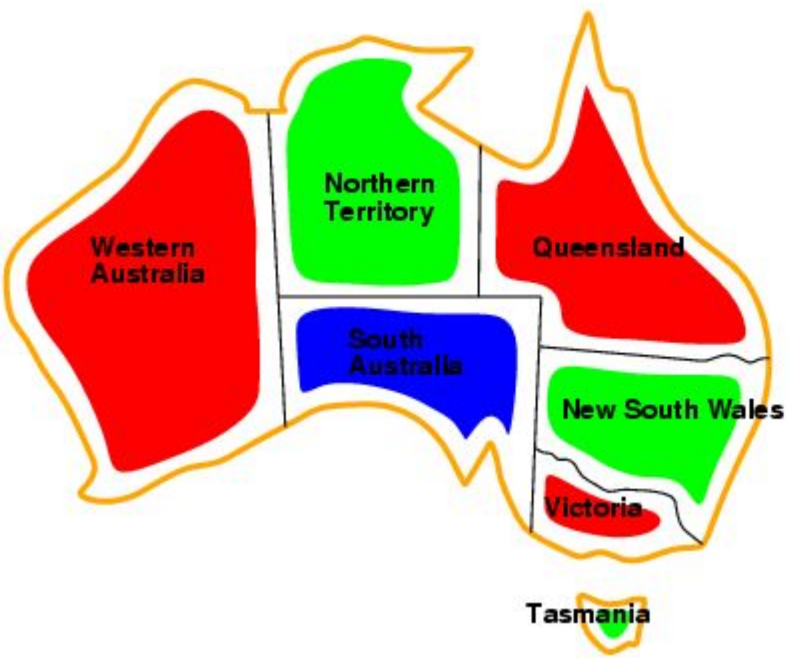
- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints

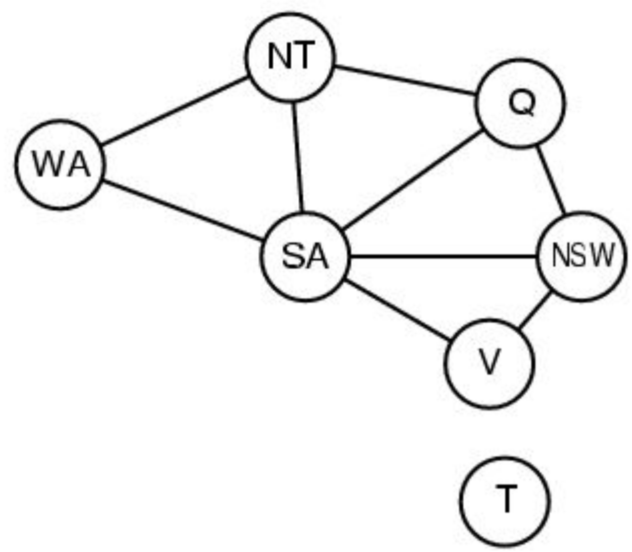




Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints





Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

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

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

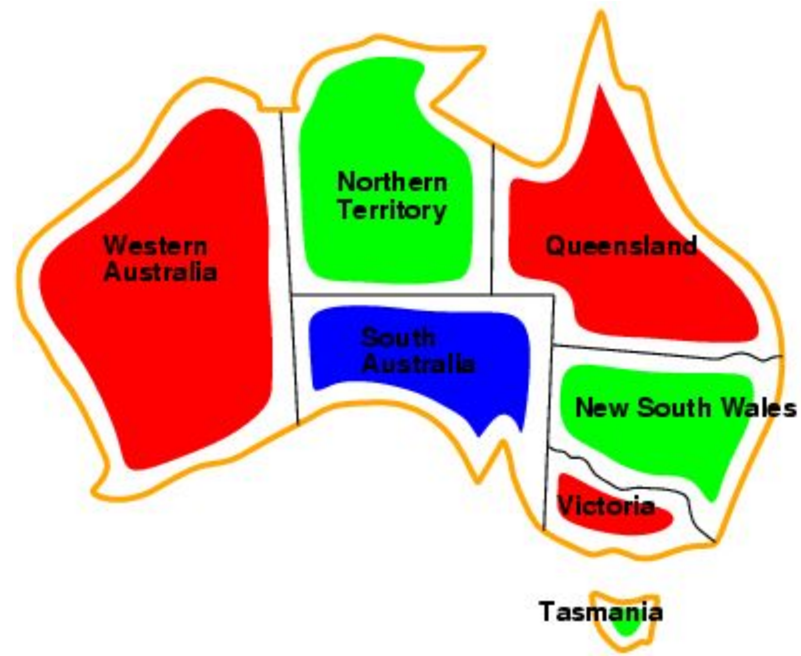
- Can solve n-queens for $n \approx 25$

Backtracking search

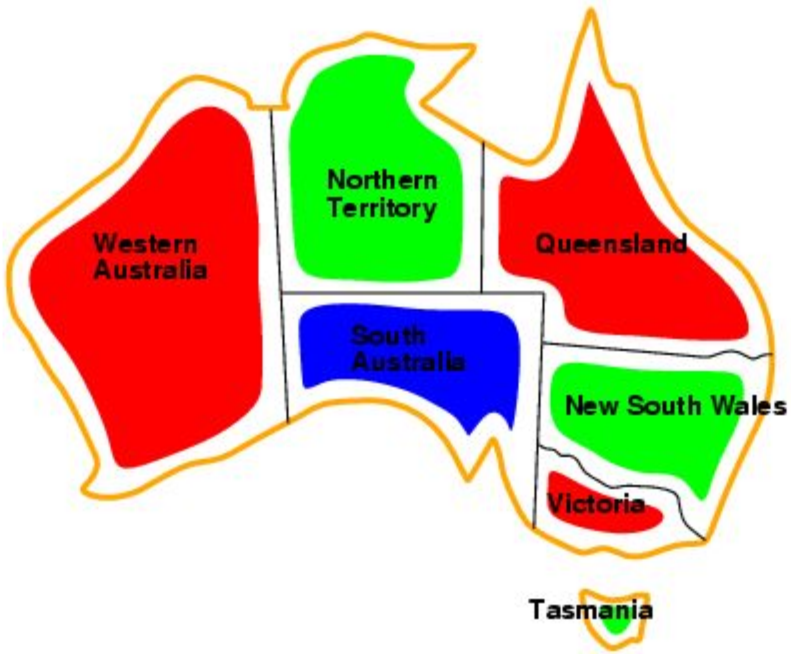
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or
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 according to Constraints[csp] then
         add { var = value } to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failue then return result
         remove { var = value } from assignment
   return failure
```

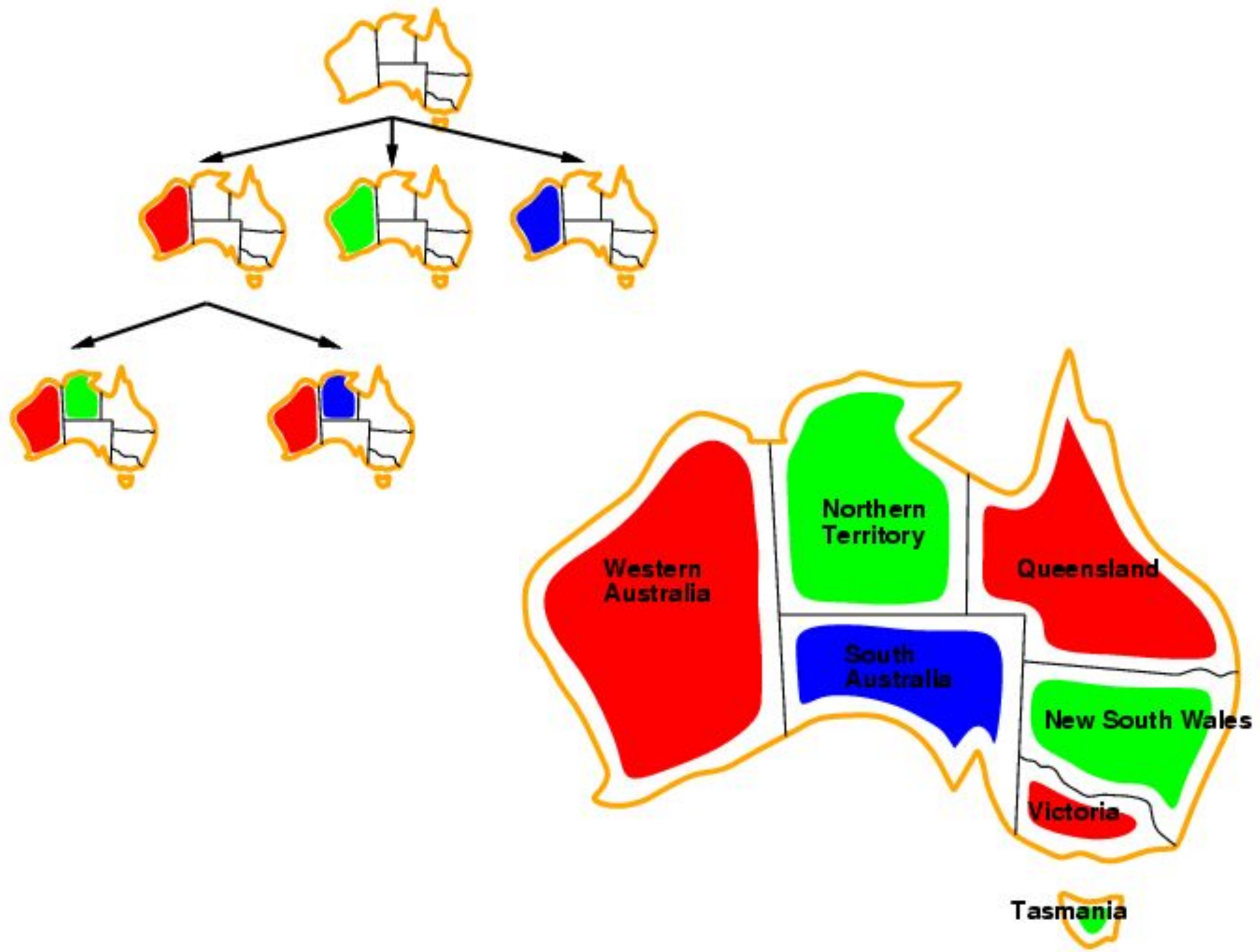
Backtracking example



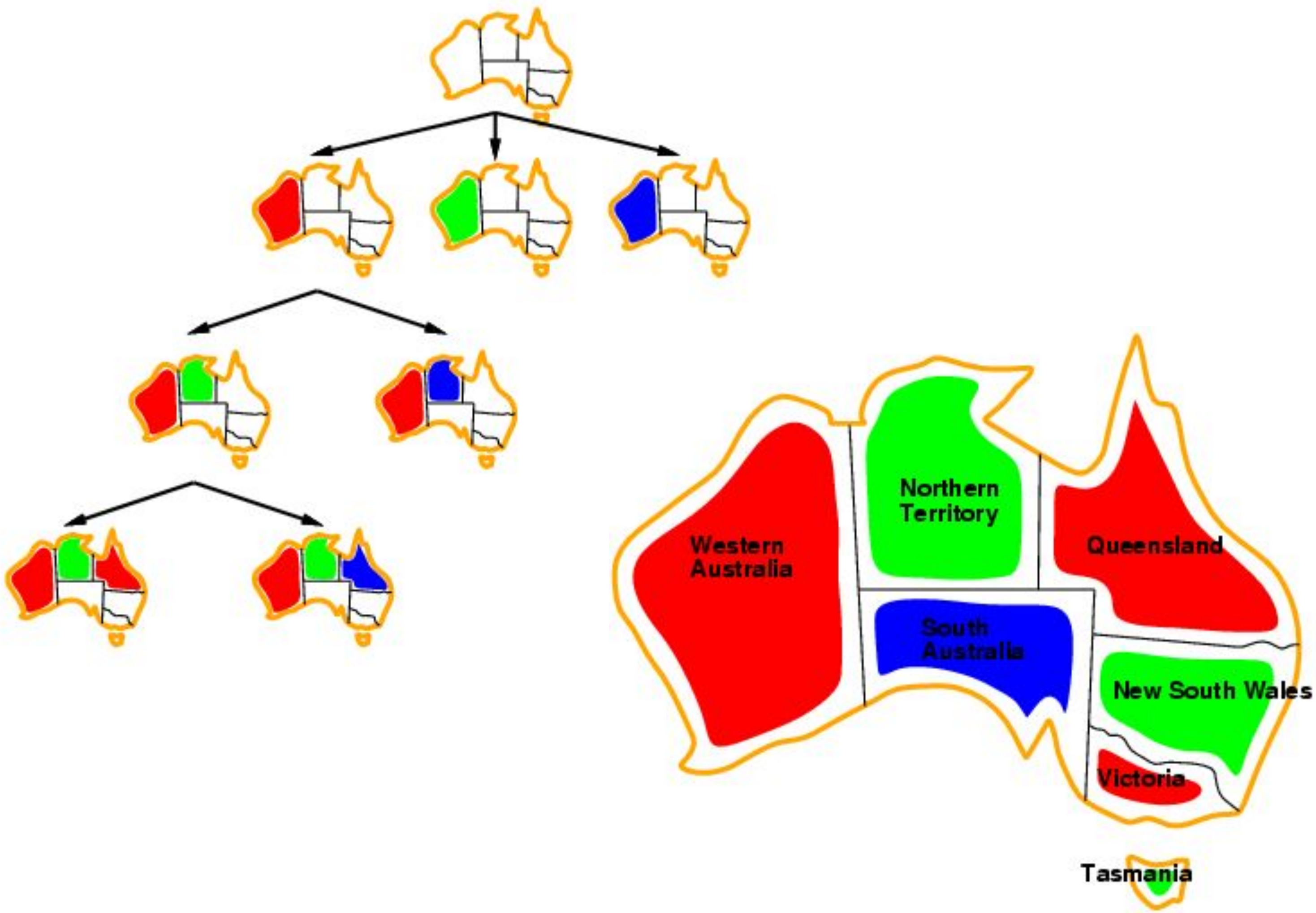
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Variable and Value Selection

- Selecting variables and assigning values using a static list is not always the most efficient approach.
  - Difficult to make the "right" choice for picking and setting the next variable.

HEURISTICS can help here, e.g.,

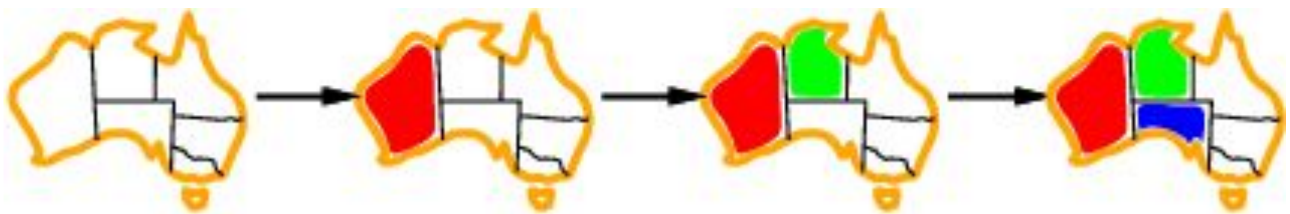
- "Minimum remaining values" heuristic – choose the variable with the smallest number of remaining values in its domain.
  - Also called "most constrained variable" heuristic

- "Degree heuristic" – choose the variable that is part of the most remaining unsatisfied constraints.
  - Useful to select first variable to assign.

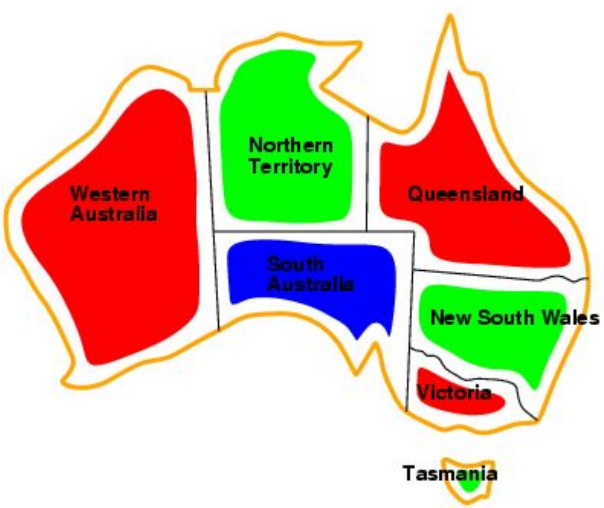
- "Least-constraining-value" heuristic – once a variable is chosen, choose its value as the one that rules out the fewest choices for neighboring variables.
  - Keeps maximum flexibility for future variable assignments.

Most constrained variable

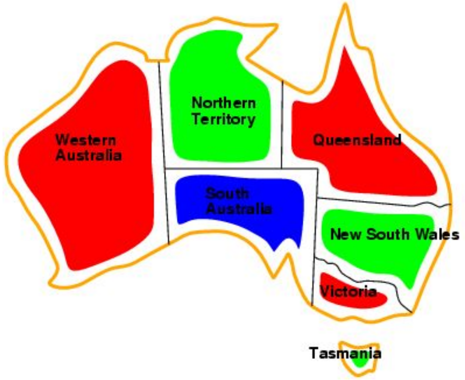
- Most constrained variable:
  choose the variable with the fewest legal values



- a.k.a. minimum remaining values (MRV) heuristic



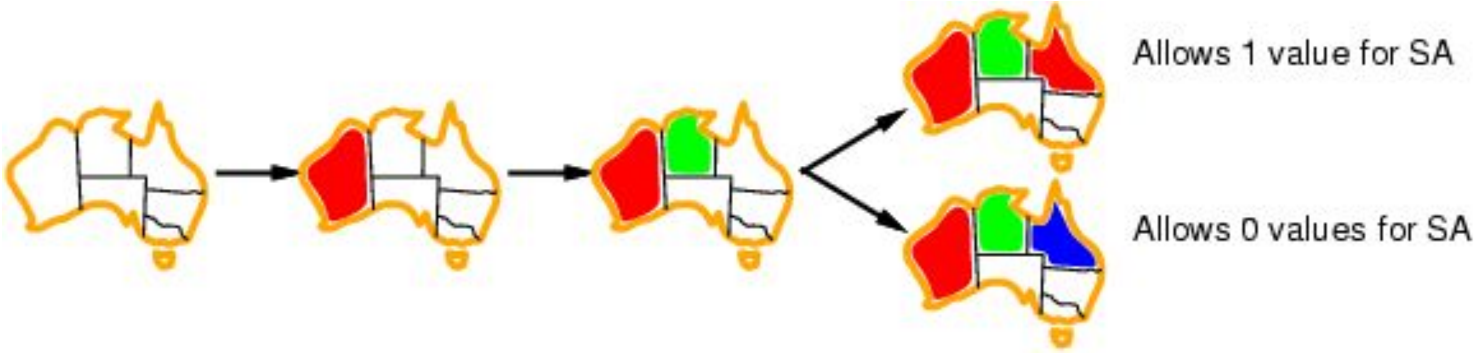
Degree Heuristic



- A good idea is to use it as a tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables

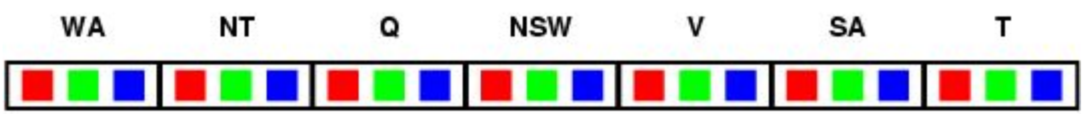
Least constraining value

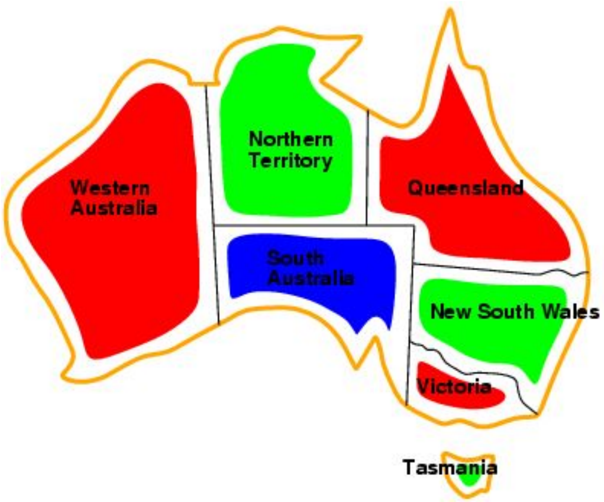
- Given a variable to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

- 

Forward checking

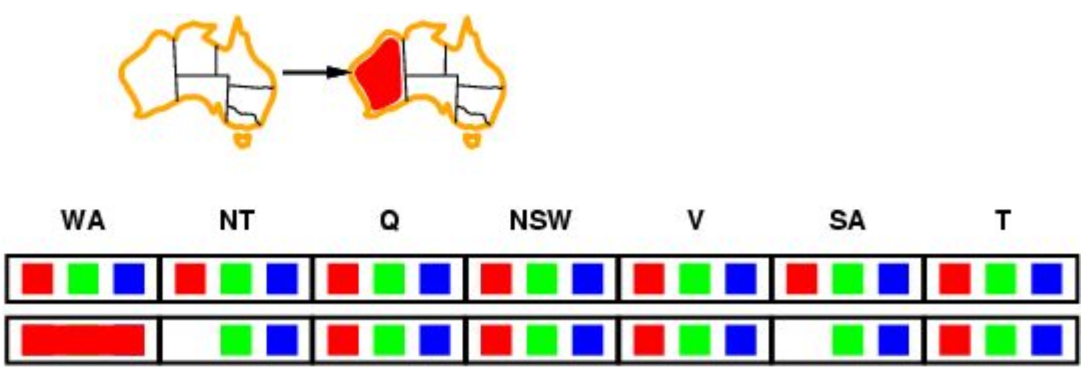
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

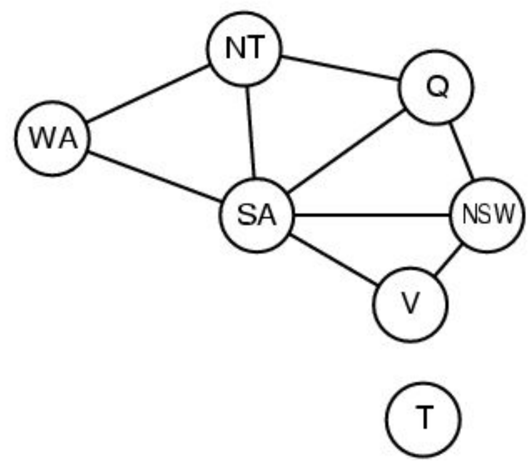




Forward checking

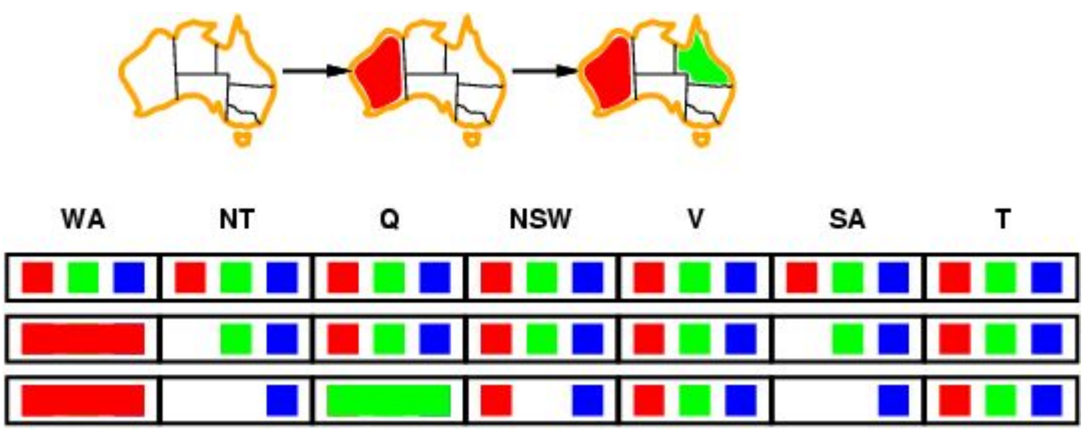
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

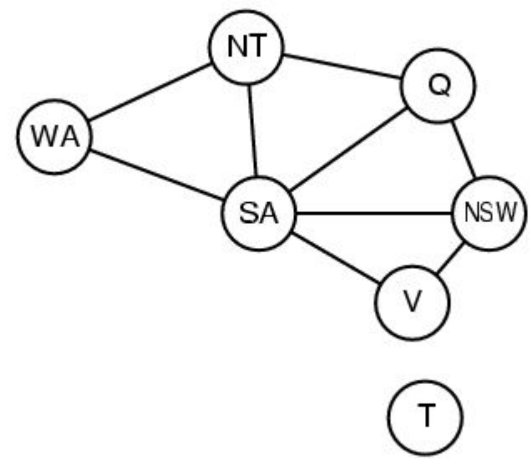




Forward checking

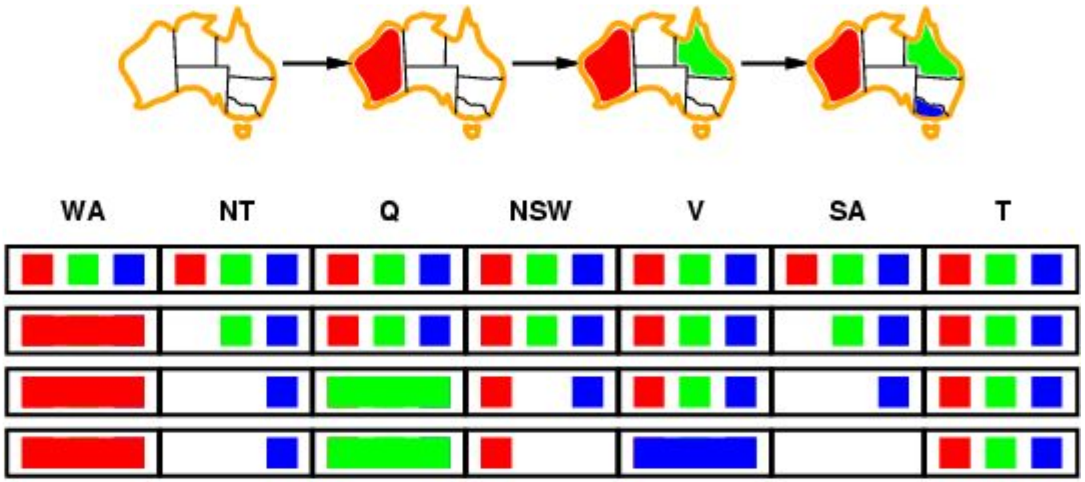
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

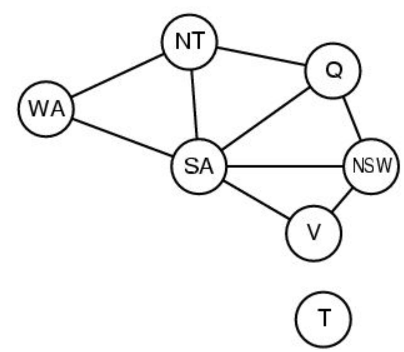




Forward checking

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

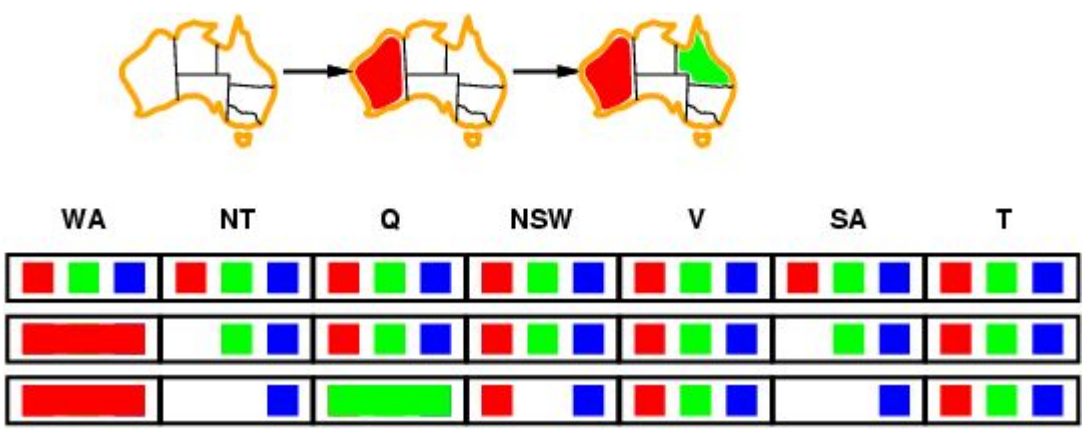


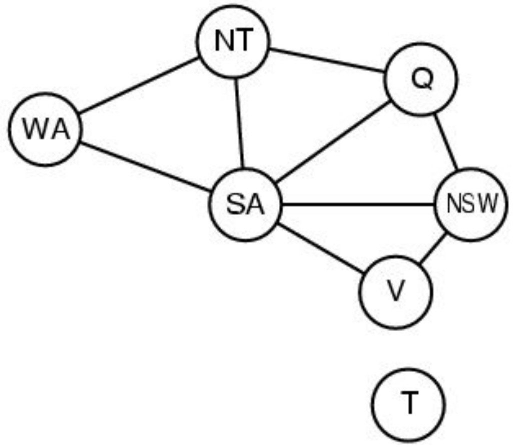


	WA	NT	Q	NSW	V	SA	T
Domains	RGB	RGB	RGB	RGB	RGB	RGB	RGB
After WA	R	GB	RGB	RGB	RGB	GB	RGB
After Q	R	B	G	RB	RGB	B	RGB
After V	R	B	G	R	B		RGB

Constraint propagation

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



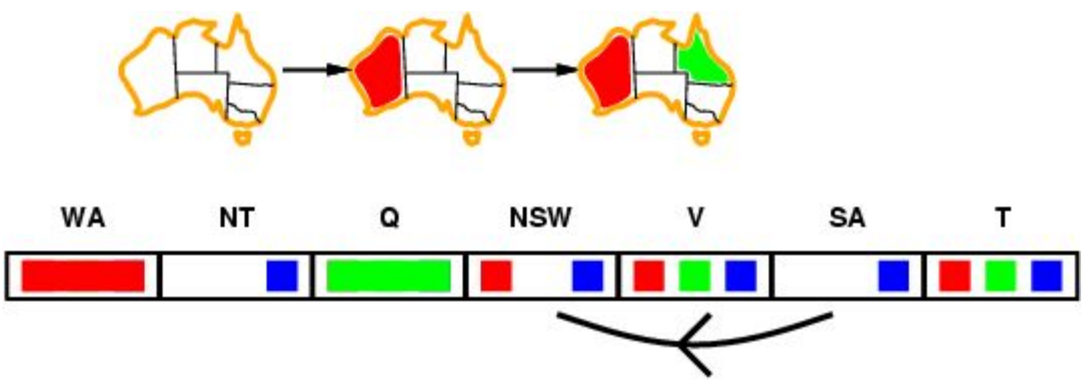


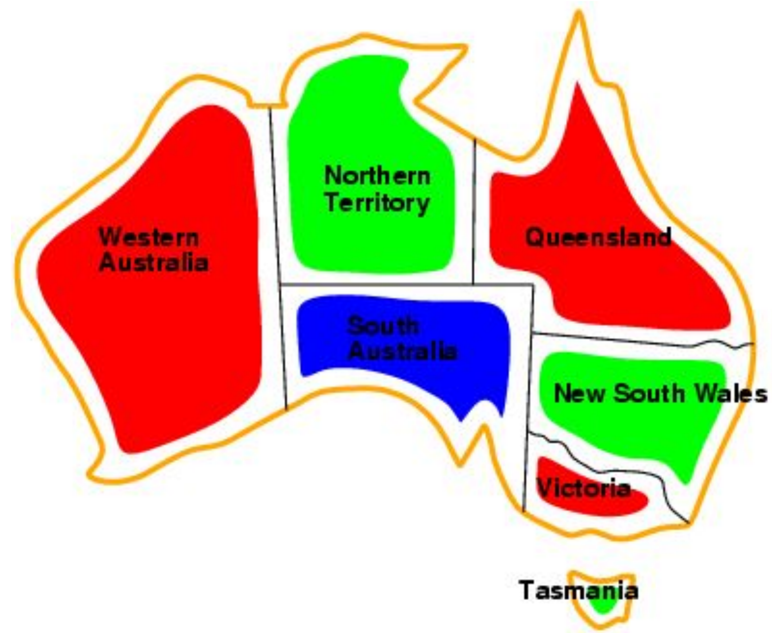
- NT and SA cannot both be blue!
- Constraint propagation algorithms repeatedly enforce constraints locally…

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

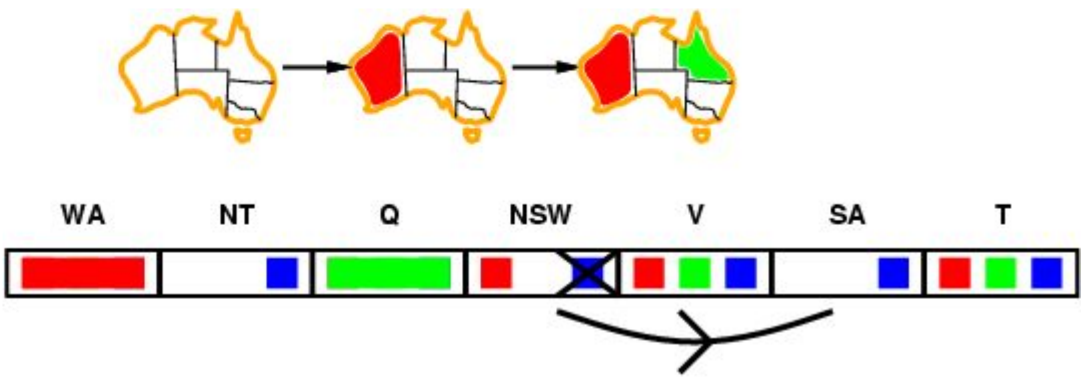


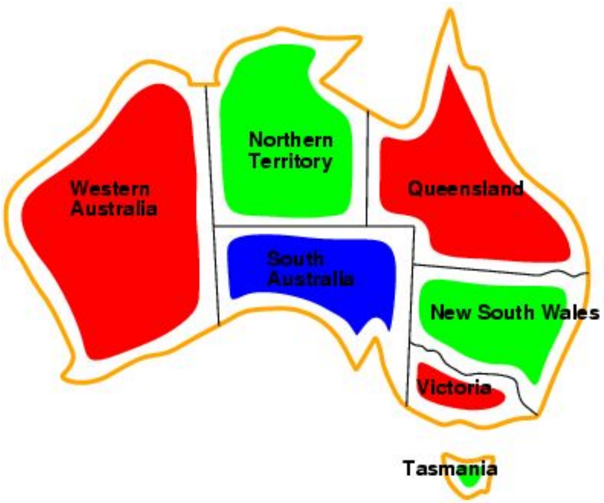


Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* □ *Y* is consistent iff

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

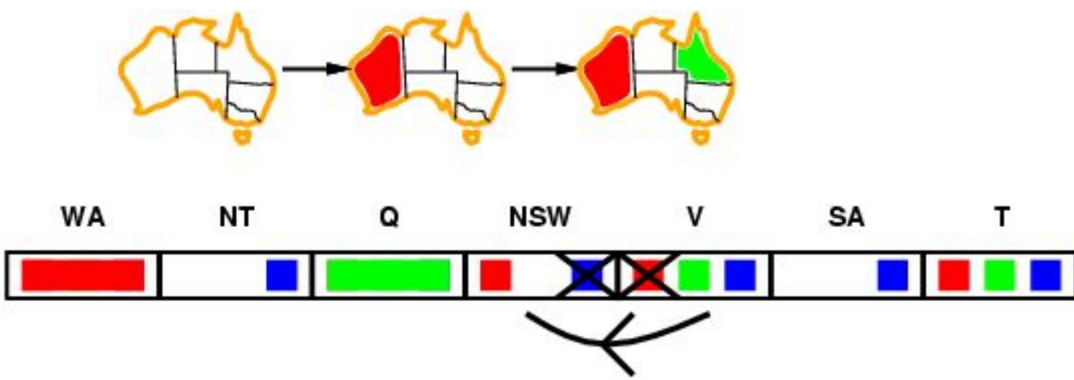


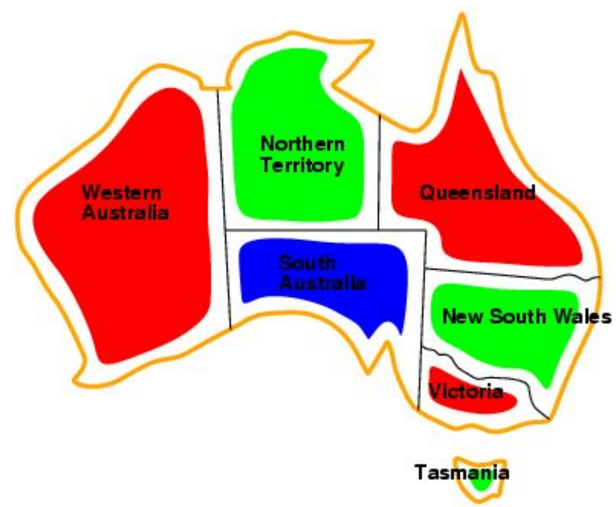


Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

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



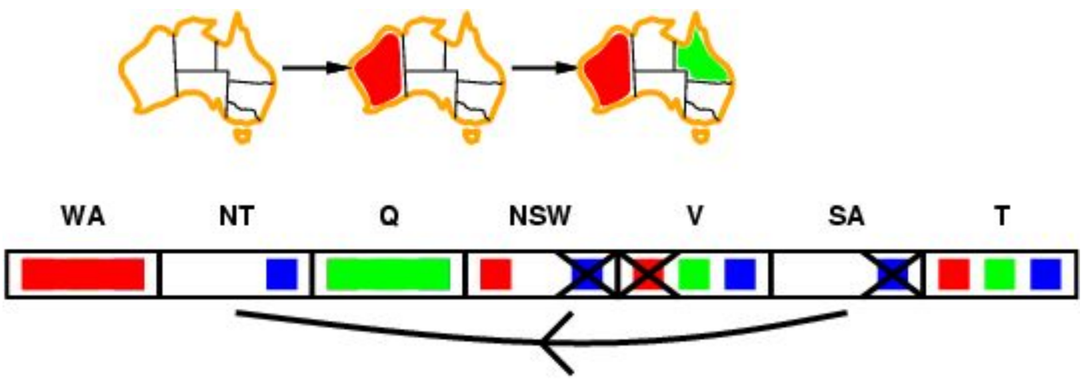


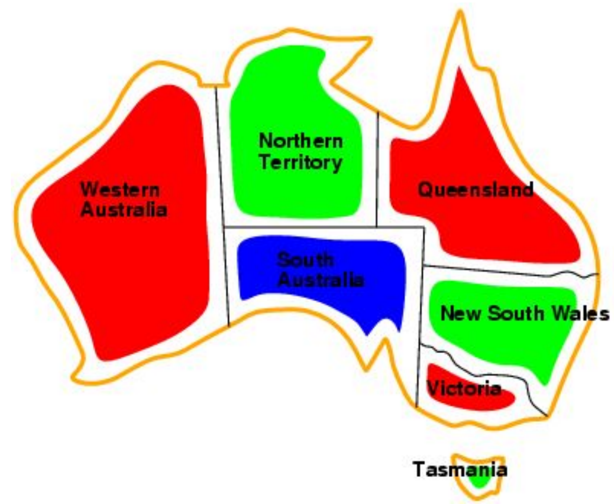
If *X* loses a value, neighbors of *X* need to be rechecked

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 AC-3

```
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 RM-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

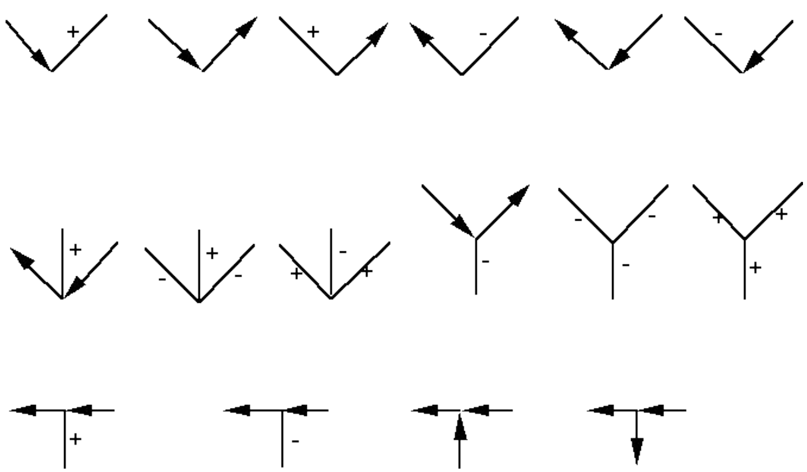
```
function RM-INCONSISTENT-VALUES( X_i, X_j) returns true iff remove a value
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint(X_i, X_j)
         then delete x from DOMAIN[X_i];  removed ← true
   return removed
```

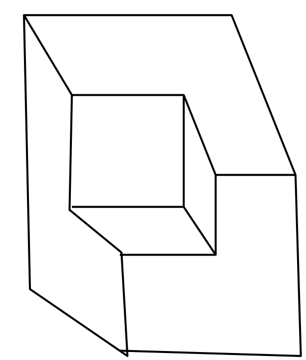
- Time complexity: O(#constraints |domain|^3)

Checking consistency of an arc is O(|domain|^2)

A Famous Example: Labelling Line Drawings

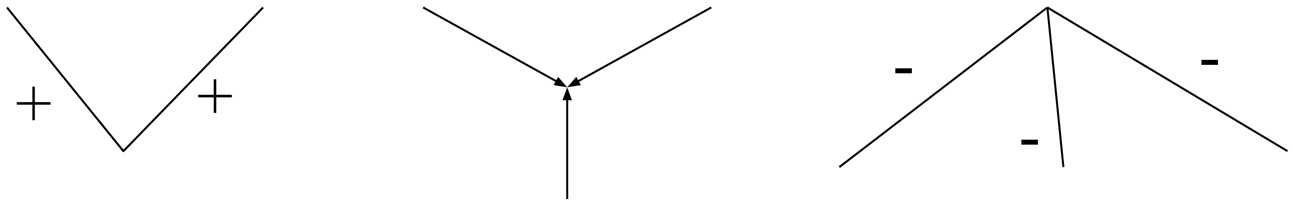
- Waltz labelling algorithm – one of the earliest CSP applications
  - Convex interior lines are labelled as +
  - Concave interior lines are labeled as –
  - Boundary lines are labeled as →
- There are 208 labellings (most of which are impossible)
- Here are the 18 legal labellings:





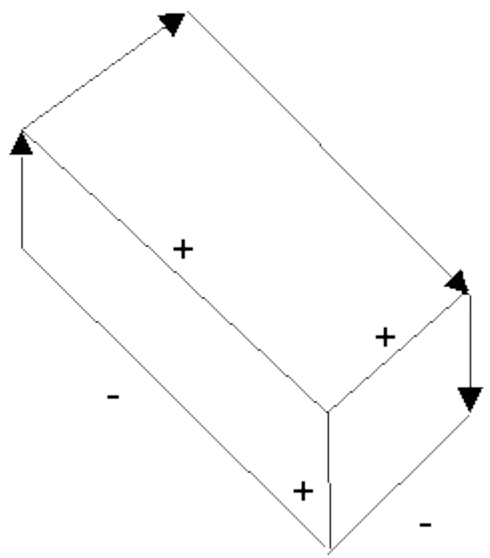
Labelling Line Drawings II

- Here are some illegal labelings:

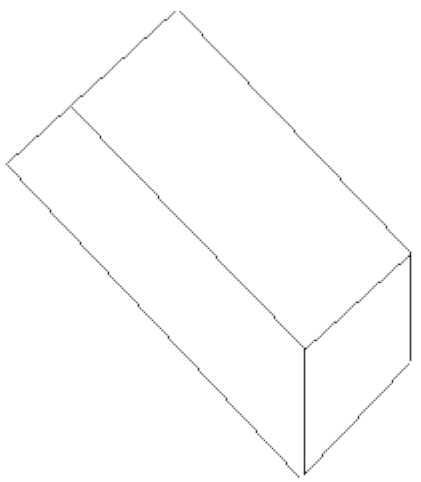


Labelling Line Drawings (cont.)

- Waltz labelling algorithm: Propagate constraints repeatedly until a solution is found



A solution for one labelling problem



A labelling problem with no solution

K-consistency

- K- consistency generalizes the notion of arc consistency to sets of more than two variables.
  - A graph is K-consistent if, for legal values of any K-1 variables in the graph, and for any Kth variable V_k, there is a legal value for V_k
- Strong K-consistency = J-consistency for all J<=K
- **Node consistency = strong 1-consistency**
- **Arc consistency = strong 2-consistency**
- Path consistency = strong 3-consistency

Why Do We Care?

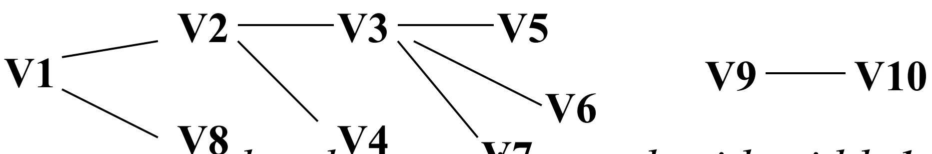
1. If we have a CSP with N variables that is known to be **strongly N-consistent**, we can solve it **without backtracking**
2. For any CSP that is **strongly K-consistent**, if we find an **appropriate variable ordering** (one with “small enough” branching factor), we can solve the CSP **without backtracking**

Ordered Constraint Graphs

- Select a variable ordering, $V_1, \ldots, V_n$
- **Width of a node** in this OCG is the number of arcs leading to earlier variables:
  - $w(V_i) = \text{Count}\ (\ (V_i, V_k) \mid k < i)$
- **Width of the OCG** is the maximum width of any node:
  - $w(G) = \text{Max}\ (w\ (V_i)),\ 1 <= i <= N$
- **Width of an unordered CG** is the minimum width of all orderings of that graph ("best you can do")

Tree-Structured Constraint Graph

- A **constraint tree** rooted at V_1 satisfies the following property:
  - There exists an ordering V1, …, Vn such that every node has zero or one parents (i.e., each node only has constraints with at most one "earlier" node in the ordering)



  - Also known as an *ordered constraint graph with width 1*
- If this constraint tree is also node- and arc-consistent (a.k.a. *strongly 2-consistent*), then it can be solved without backtracking
- (More generally, if the ordered graph is strongly k-consistent, and has width w < k, then it can be solved without backtracking.)

Proof Sketch for Constraint Trees

- Perform backtracking search in the order that satisfies the constraint tree condition
- Every node, when instantiated, is constrained only by at most one previous node
- Arc consistency tells us that there must be at least one legal instantiation in this case
  - (If there are no legal solutions, the arc consistency procedure will collapse the graph – some node will have no legal instantiations)
- Keep doing this for all *n* nodes, and you have a legal solution – without backtracking!

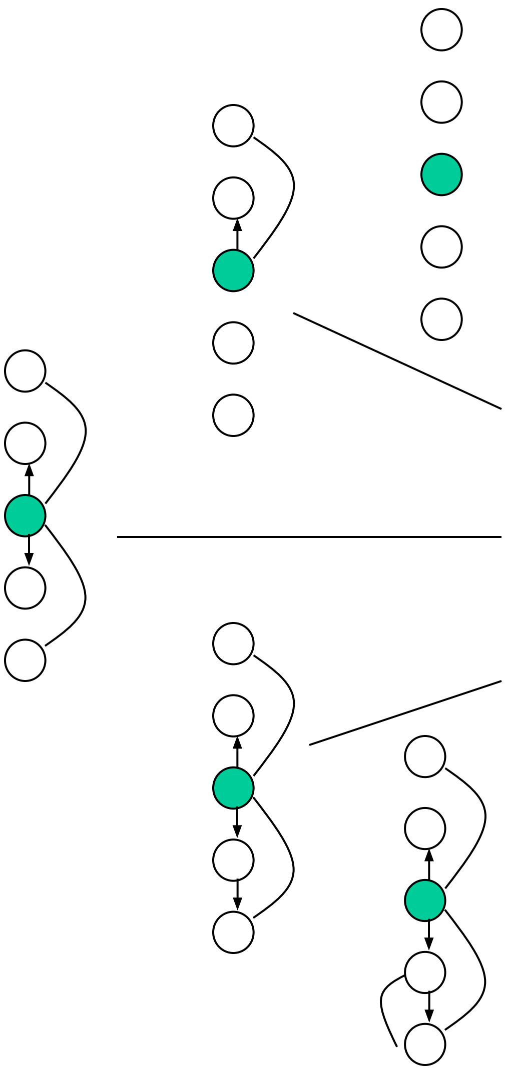
Backtrack-Free CSPs: Proof Sketch

- Given a strongly k-consistent OCG, G, with width w < k:
  - Instantiate variables in order, choosing values that are consistent with the constraints between V_i and its parents
  - Each variable has at most w parents, and k-consistency tells us we can find a legal value consistent with the values of those w parents
- *Unfortunately*, achieving k-consistency is hard (and can increase the width of the graph in the process!)
- *Fortunately*, 2-consistency is relatively easy to achieve, so constraint trees are easy to solve
- *Unfortunately*, many CGs have width greater than one (that is, no equivalent tree), so we still need to improve search

So What If We Don't Have a Tree?

- Answer #1: Try **interleaving** constraint propagation and backtracking
- Answer #2: Try using **variable-ordering** heuristics to improve search
- Answer #3: Try using **value-ordering** heuristics during variable instantiation
- Answer #4: See if **iterative repair** works better
- Answer #5: Try using **intelligent backtracking** methods

Interleaving Constraint Propagation and Search



Generate and Test	No constraint propagation: assign all variable values, **then** test constraints
Simple Backtracking	Check constraints only for variables "up the tree"
Forward Checking	Check constraints for immediate neighbors "down the tree"
Partial Lookahead	Propagate constraints forward "down the tree"
Full Lookahead	Ensure complete arc consistency after each instantiation (AC-3)

Variable Ordering

- **Intuition**: choose variables that are highly constrained early in the search process; leave easy ones for later
- **Minimum width ordering** (MWO): identify OCG with minimum width
- **Maximum cardinality ordering**: approximation of MWO that's cheaper to compute: order variables by decreasing cardinality (a.k.a. **degree heuristic**)
- **Fail first principle** (FFP): choose variable with the fewest values (a.k.a. **minimum remaining values** (MRV))
  - **Static** FFP: use domain size of variables
  - **Dynamic** FFP (**search rearrangement method**): At each point in the search, select the variable with the fewest remaining values

Variable Ordering II

- **Maximal stable set**: find largest set of variables with no constraints between them and save these for last
- **Cycle-cutset tree creation**: Find a set of variables that, once instantiated, leave a tree of uninstantiated variables; solve these, then solve the tree without backtracking
- **Tree decomposition**: Construct a tree-structured set of connected subproblems

Value Ordering

- **Intuition**: Choose values that are the least constrained early on, leaving the most legal values in later variables
- **Maximal options method** (a.k.a. **least-constraining-value** heuristic): Choose the value that leaves the most legal values in uninstantiated variables
- **Min-conflicts**: Used in iterative repair search (see below)

- Symmetry: Introduce **symmetry-breaking constraints** to constrain search space to "useful" solutions (don't examine more than one symmetric/isomorphic solution)

Iterative Repair

- Start with an initial complete (but invalid) assignment
- Hill climbing, simulated annealing
- Min-conflicts: Select new values that minimally conflict with the other variables
  - Use in conjunction with hill climbing or simulated annealing or…
- Local maxima strategies
  - Random restart
  - Random walk
  - Tabu search: don't try recently attempted values

Min-Conflicts Heuristic

- Iterative repair method
  1. Find some “reasonably good” initial solution
     - E.g., in N-queens problem, use greedy search through rows, putting each queen where it conflicts with the smallest number of previously placed queens, breaking ties *randomly*
  2. Find a variable in conflict (randomly)
  3. Select a new value that minimizes the number of constraint violations
     - O(N) time and space
  4. Repeat steps 2 and 3 until done
- Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution

Intelligent Backtracking

- **Backjumping**: if V_j fails, jump back to the variable V_i with greatest i such that the constraint (V_i, V_j) fails (i.e., most recently instantiated variable in conflict with V_i)
- **Backchecking**: keep track of incompatible value assignments computed during backjumping
- **Backmarking**: keep track of which variables led to the incompatible variable assignments for improved backchecking

Some Challenges for Constraint Reasoning

- What if not all constraints can be satisfied?
  - Hard vs. soft constraints
  - Degree of constraint satisfaction
  - Cost of violating constraints
- What if constraints are of different forms?
  - Symbolic constraints
  - Numerical constraints [constraint solving]
  - Temporal constraints
  - Mixed constraints

Some More Challenges for Constraint Reasoning

- What if constraints are represented intensionally?
  - Cost of evaluating constraints (time, memory, resources)
- What if constraints, variables, and/or values change over time?
  - Dynamic constraint networks
  - Temporal constraint networks
  - Constraint repair
- What if you have multiple agents or systems involved in constraint satisfaction?
  - Distributed CSPs
  - Localization techniques

Real-World Problems

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design