Constraint Satisfaction Problems

CS4881 Artificial Intelligence

Jay Urbain, Ph.D.

Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

Constraint satisfaction problems (CSPs)

- Standard search problem:
  - From the point of view of standard search algorithm, each state is atomic (indivisible, no partial states).
  - Treated as a "black box” with no internal structure.

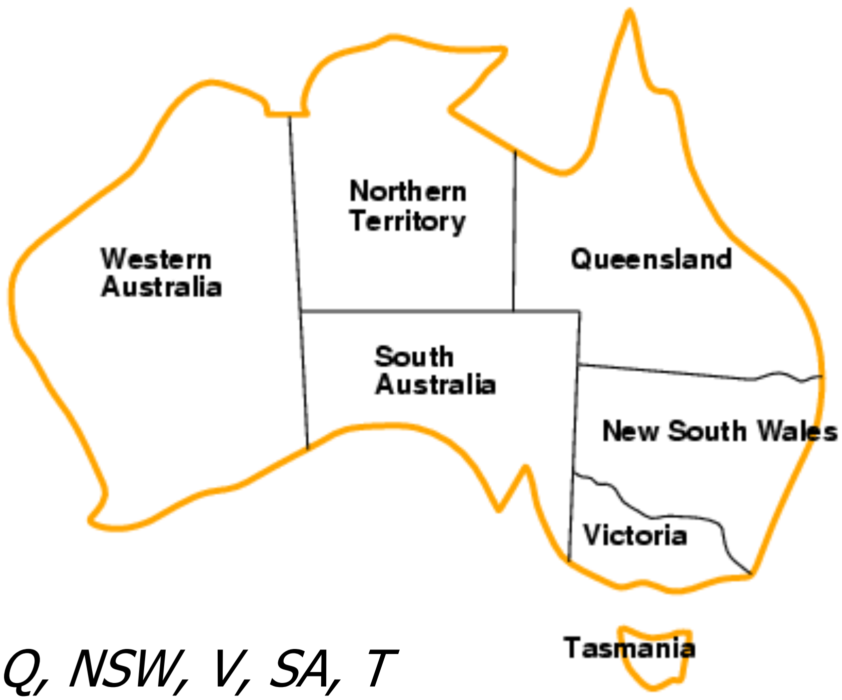
- CSP:
  - Take advantage of the structure of states, and use *general-purpose* rather than *problem-specific* heuristics.
  - Use a factored representation of states defined by *variables* X_i with *values* from domain D_i
  - Eliminate large portions of the search space all at once by identifying variable/value combinations that violate constraints.
  - CSP: Goal test is a set of *constraints* specifying allowable combinations of values for subsets of variables.

Constraint satisfaction problems (CSPs)

- CSP Problem {X,D,C}:
  - X is set of variables, $\{X_1, X_2, \ldots, X_n\}$
  - D is a set of domains, $\{D_1, D_2, \ldots, D_n\}$, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.

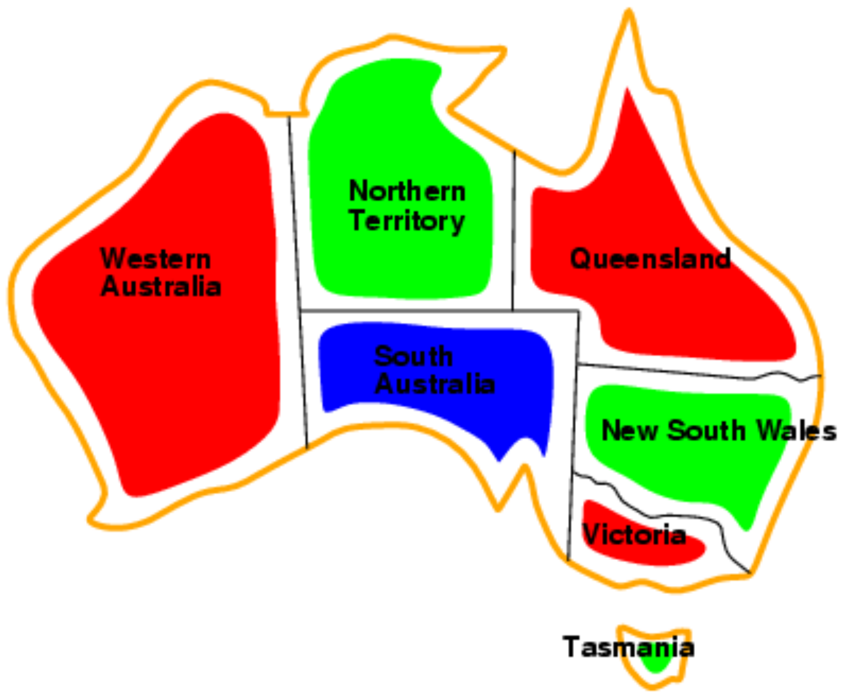
- To solve a CSP, we need to define a state space and allowable solutions.
  - Consistent assignment: assignment (of values to variables) that does not violate any constraint.
  - Complete assignment: Every variable is assigned.
  - *Solution: Consistent and complete assignment.*
  - Partial assignment: assign values to only some of the variables.

Example: Map-Coloring



- Variables *WA, NT, Q, NSW, V, SA, T*
- Domain (of each variable) 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)}
  - Set of constraints define a simple formal language

Example: Map-Coloring



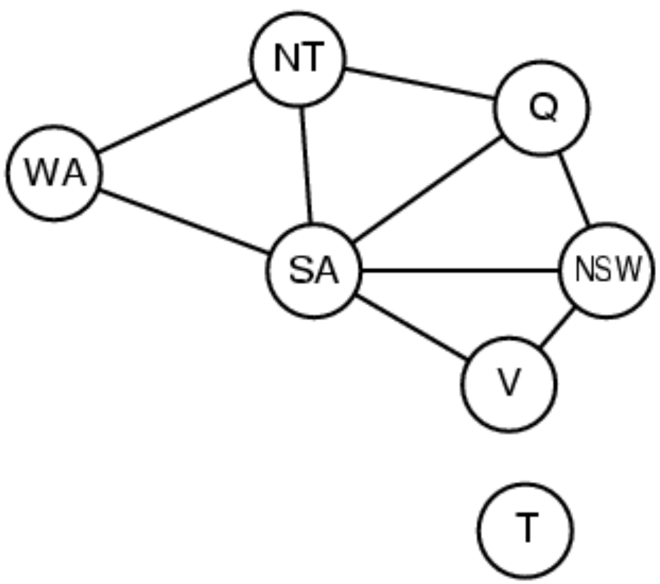
- Solutions are complete and consistent assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

CSP

- Many problems are naturally represented by constraints: scheduling, assignment, etc.
- By using constraints, search space can be significantly reduced.
- By casting a problem as a CSP, can use existing CSP solver.

Constraint graph

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



Varieties of CSPs

- Discrete variables
  - finite domains:
    - n variables, domain size $d \rightarrow O(d^n)$ 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., $StartJob_1 + 5 \leq StartJob_3$

- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

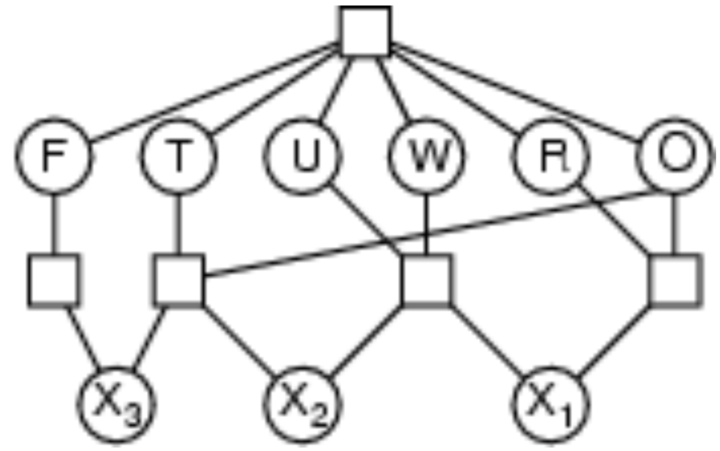
Varieties of constraints

- 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, sudoku, rubics cube, etc.

Example: Cryptarithmetic Puzzle

- *Each letter stands for a distinct digit.*
- *Aim: find submission of digits for letters such that resulting sum is arithmetically correct!*
- *No leading zeros allowed.*

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

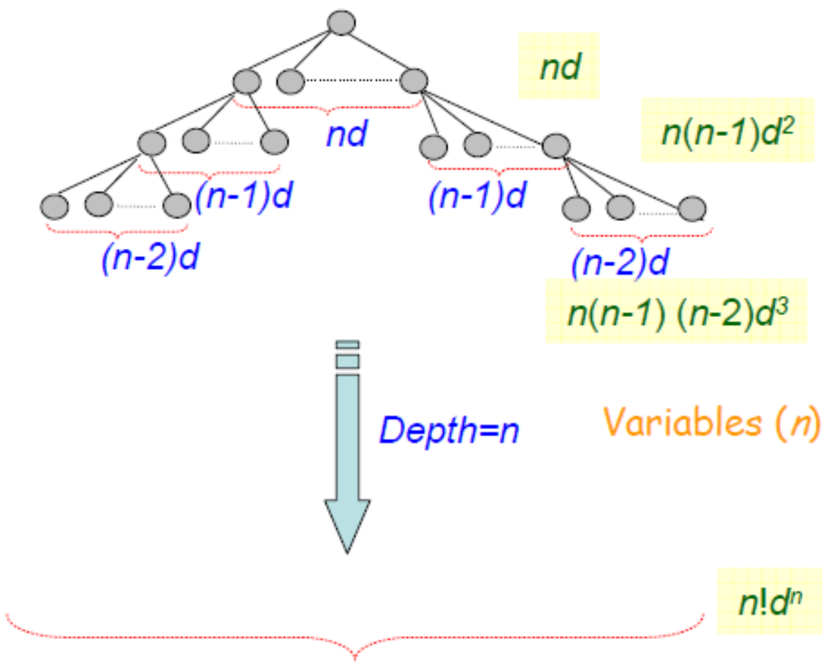


- Constraint Hypergraph – middle squares are add const's, *X_1 X_2 X_3 caries*
- Variables:

  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)*
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F$, $T \neq 0$, $F \neq 0$

Standard Search Approach

- Can formulate as standard search problem – but order not important!
- If **incremental formulation** is used
- Breadth-first search with search tree with depth limit *n*



Initial state: empty assignment {}

Successor function: a value can be assigned to any unassigned variables, provided that no conflict occurs

Goal test: the assignment is complete

Variables (*n*) and Values (*d*)

Totally, d^n distinct leaf nodes (because of commutativity)

  - Every solution appears at depth *n* with *n* variable assigned
  - DFS (or depth-limited search) also can be applied (smaller space requirement)

Real-world CSPs

- Assignment problems
  - e.g., games: Sudoku, who teaches what, verification problems
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

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

- Backtracking search is the basic uninformed algorithm for CSPs

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

Backtracking search

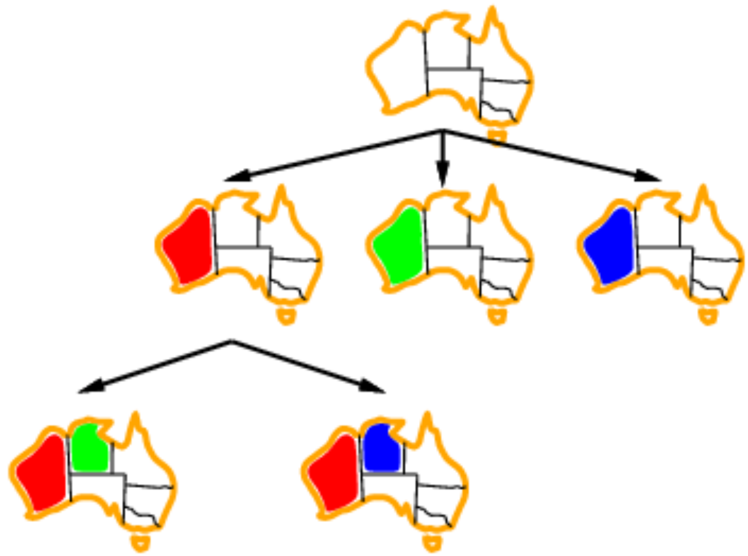
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

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
```

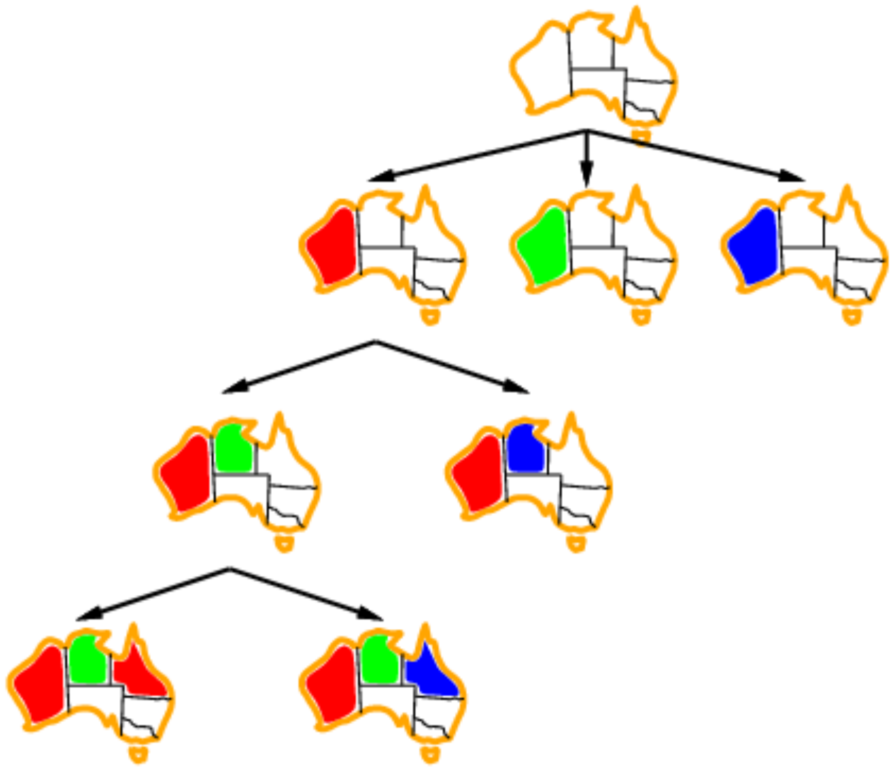
Backtracking example

Backtracking example

Backtracking example



Backtracking example

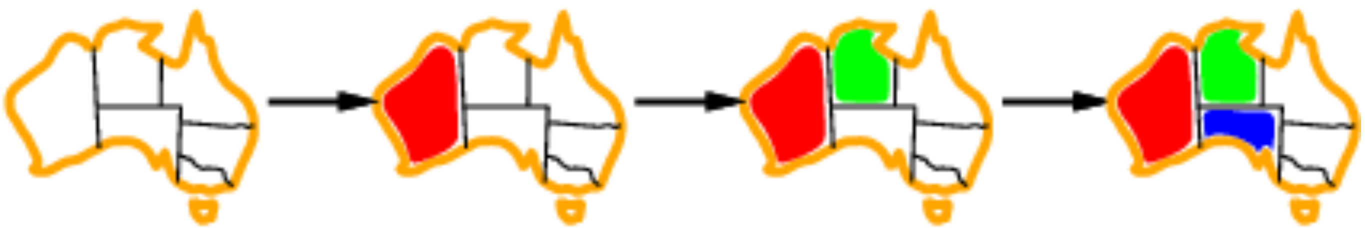


Improving backtracking efficiency

- 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?

Most constrained variable

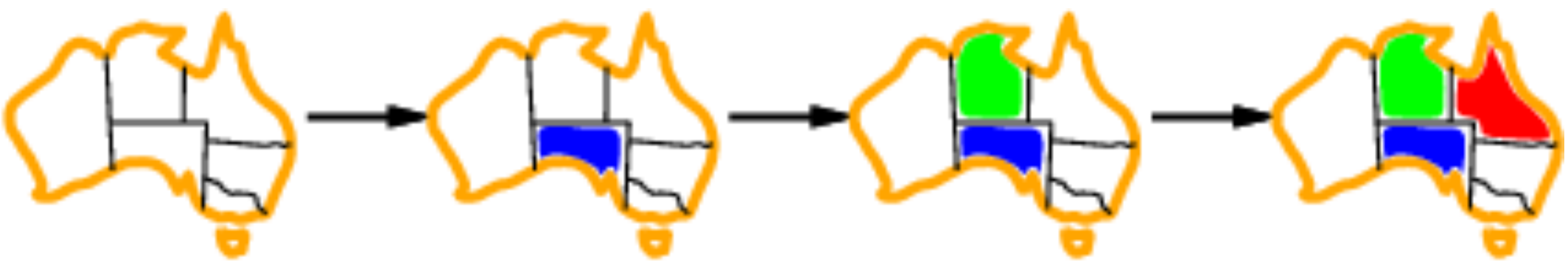
- *Most constrained variable*:
  choose the variable with the fewest legal values



- Also known as *minimum remaining values (MRV)* heuristic

Least *constraining* value

- Tie-breaker among most constrained variables
  - Assign value to variable that places the least constraints on *remaining* variables.



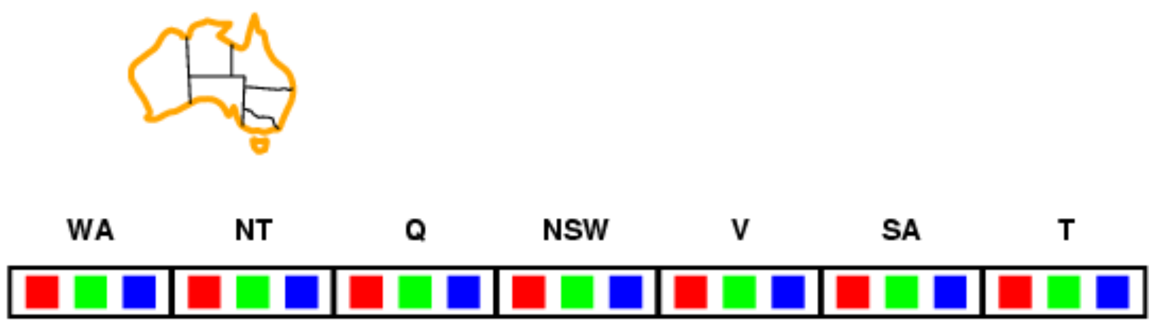
Least constraining value

- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables – reserve the right for other variables to play.

- Combining these heuristics makes 1000 queens feasible

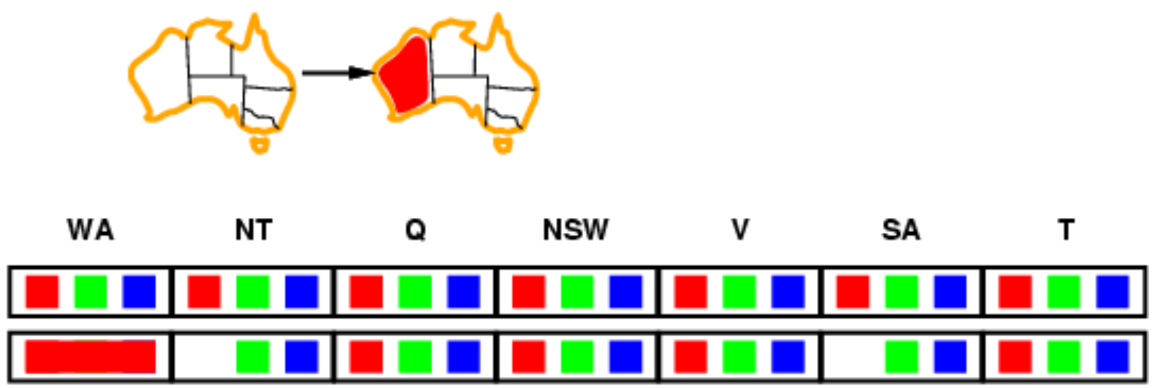
Forward checking

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



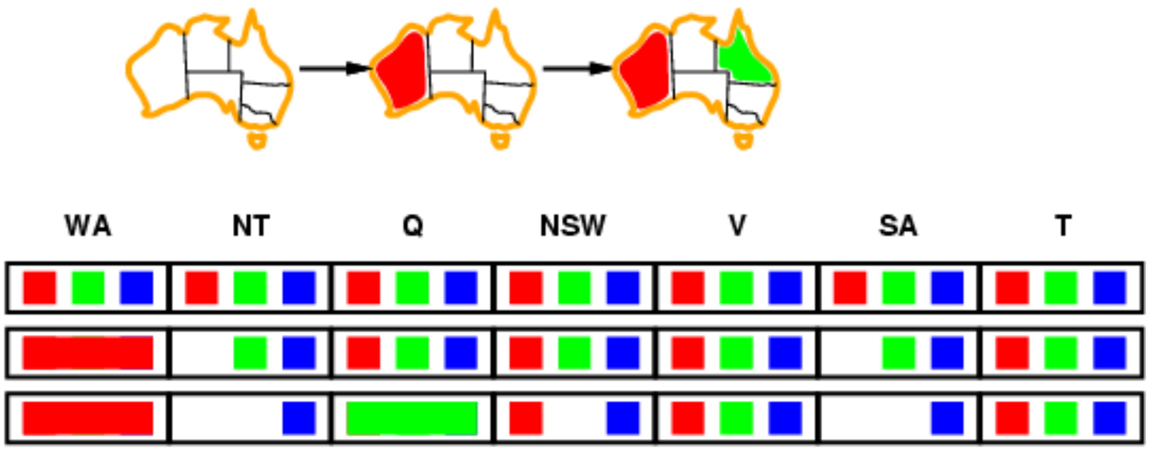
Forward checking

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



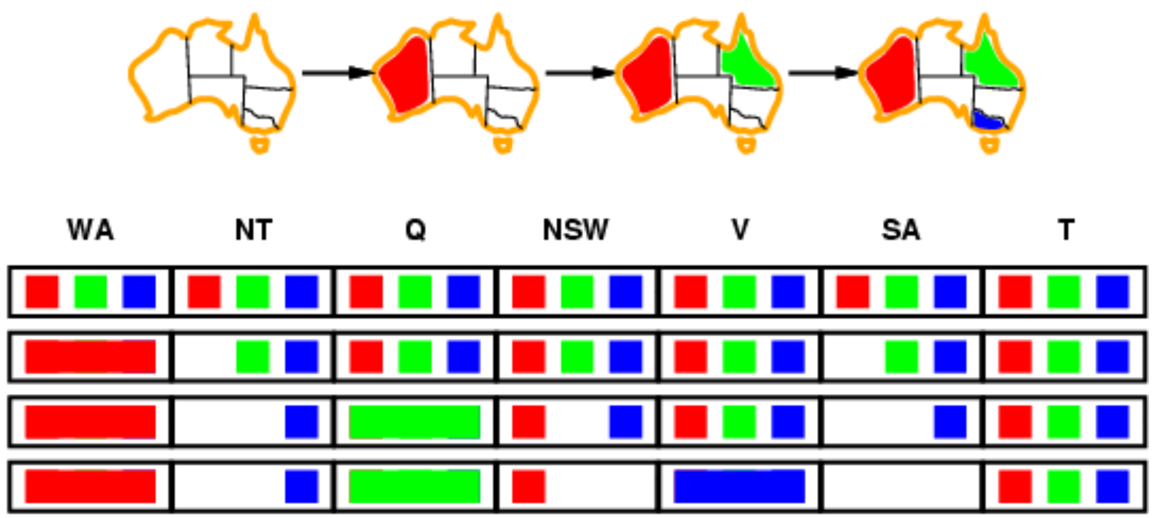
Forward checking

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



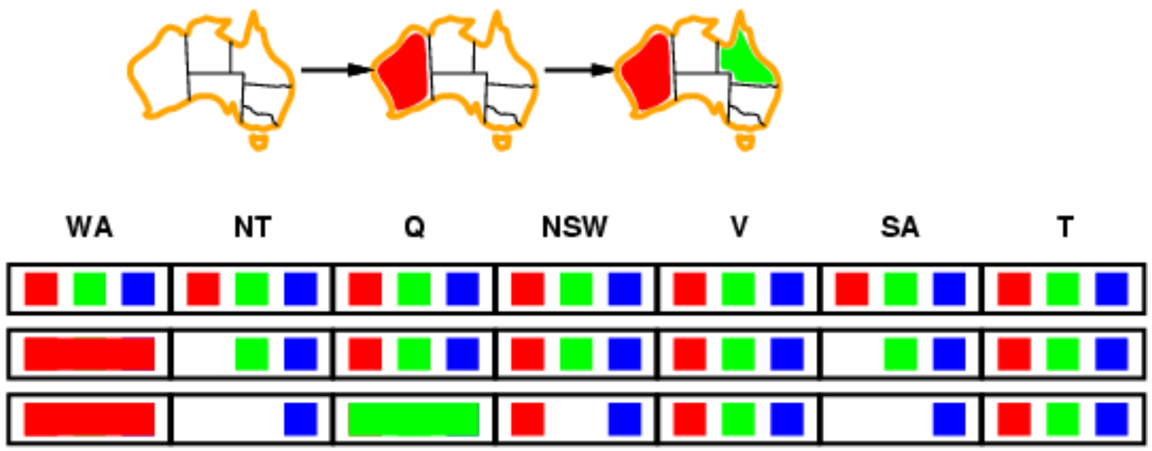
Forward checking

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



Constraint propagation

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



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

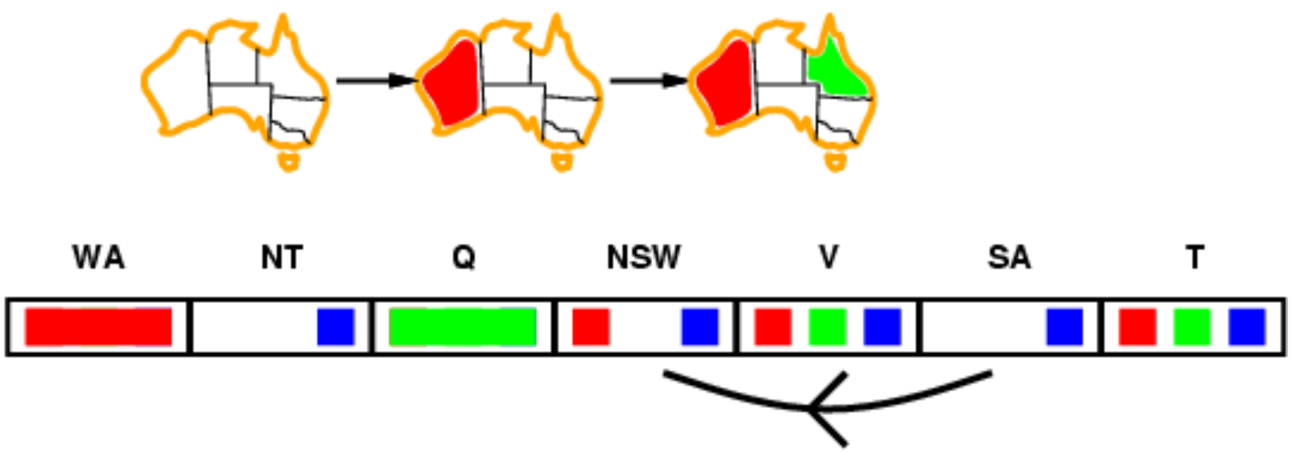
Node consistency

- A single variable is node-consistent if all the values in the variable's domain satisfy the variables' *unary* constraints.
- E.g.:
  - SA starts with domain {red, green, blue}, but Australian's don't like green.
  - We can make it node consistent by eliminating green {red, blue}.

Arc consistency

- A variable is arc-consistent if every value in its domain satisfies the variable's *binary* constraints.
- Simplest form of propagation makes each arc *consistent*
- $X \rightarrow Y$ is consistent iff

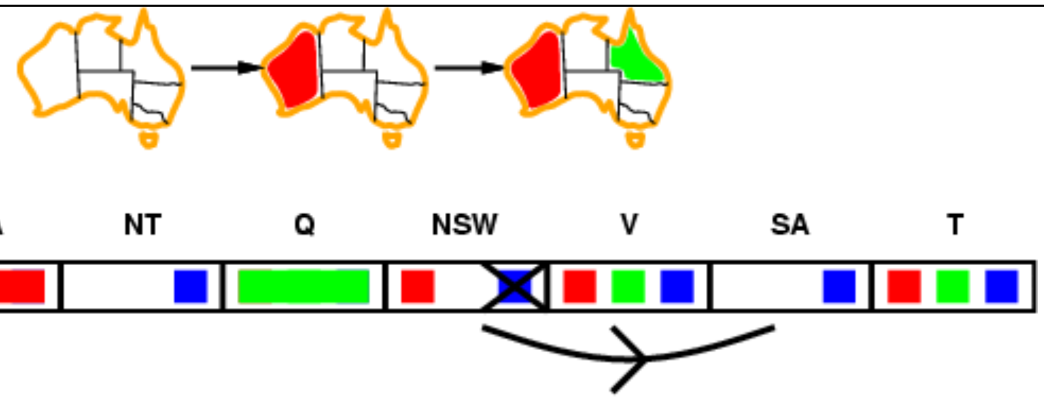
  for every value x of X there is some allowed y



Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

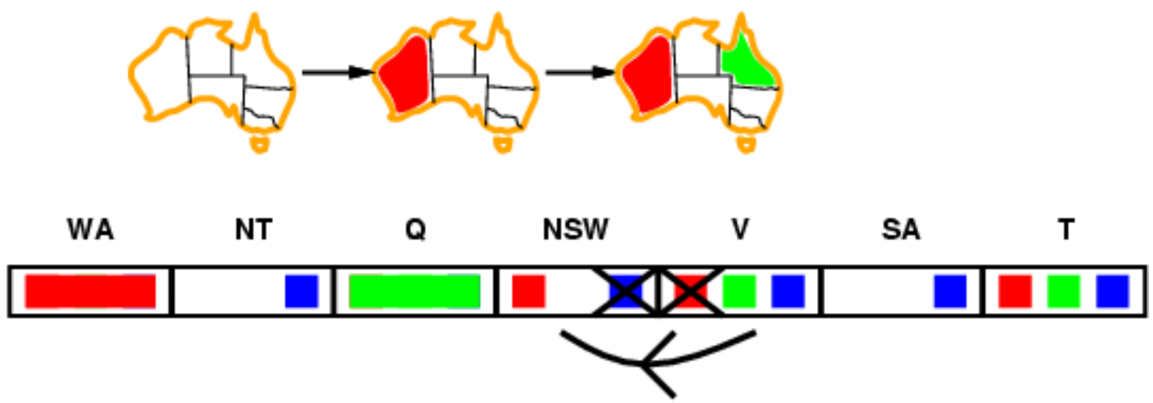
  for every value x of X there is some allowed y



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

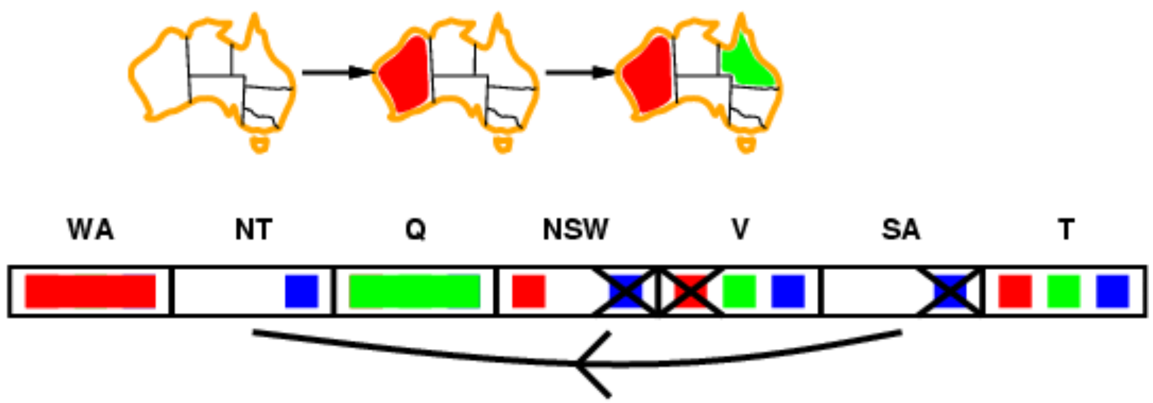


- If X loses a value, neighbors of X need to be rechecked

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 AC-3

```
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 RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

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

- Time complexity: $O(n^2d^3)$

Local search for CSPs

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

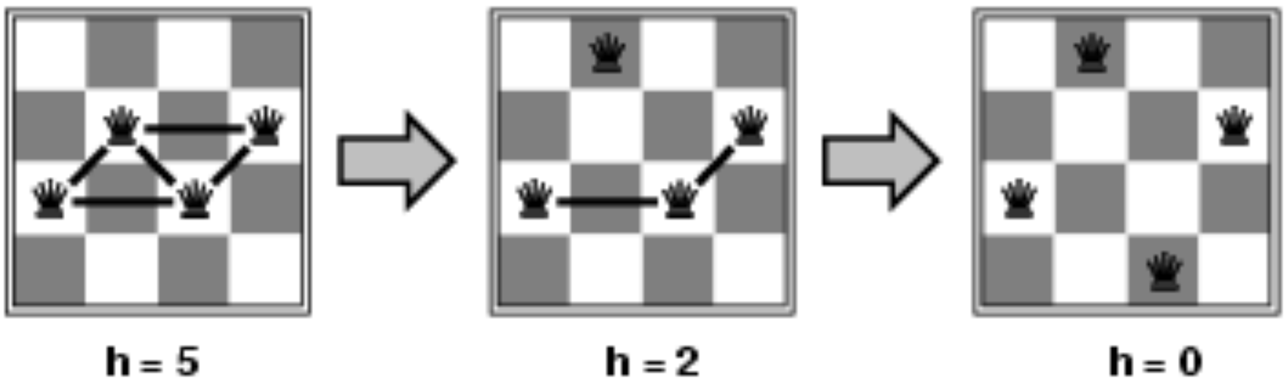
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- Variable selection: randomly select any conflicted variable.

- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with $h(n)$ = total number of violated constraints

Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks



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

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
- Iterative min-conflicts is usually effective in practice