# Constraint Satisfaction Problems

Chapter 6 in AIMA

(NOTE: we are covering Ch. 6 before Ch. 5)

# Outline

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

# Constraint satisfaction problems (CSPs)

- Standard search problem:
  - state is a "black box“ – any data structure that supports successor function, heuristic function, and goal test
  - one or more specified goal states (or optimal states)

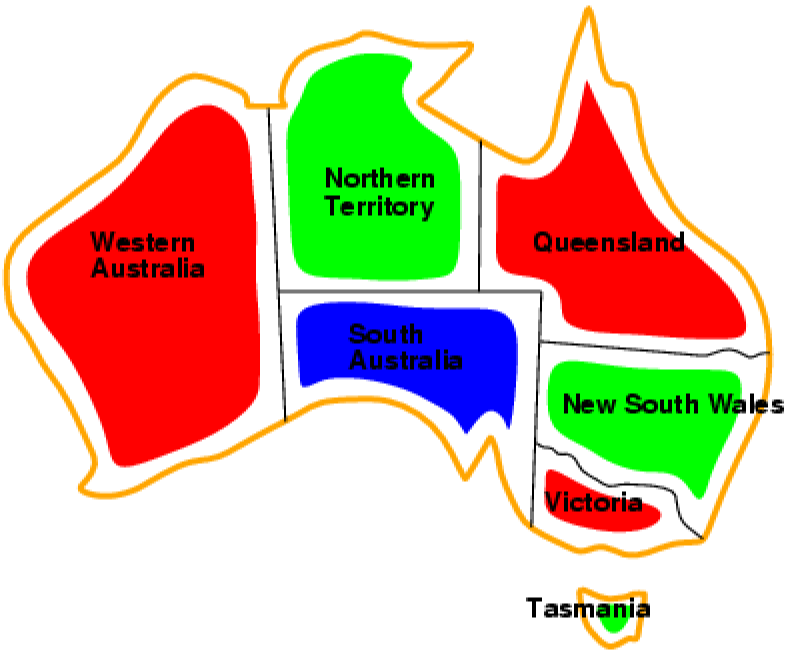
- CSP:
  - state is defined by variables $X_i$ with values from domain $D_i$
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

# 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)}

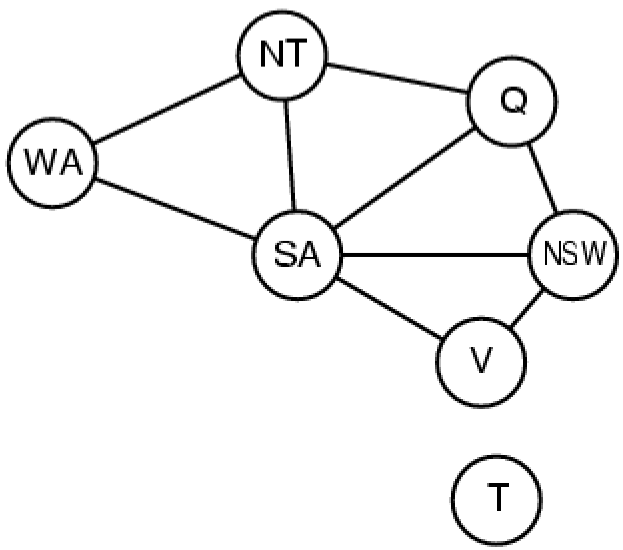
# Example: Map-Coloring




- 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



# Varieties of CSPs

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

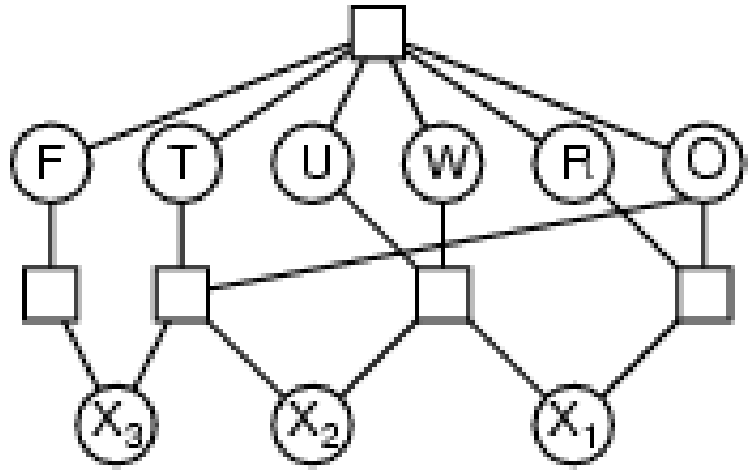
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in poly time (continuous linear programming!)
  - Note: Simplex method for LP usually works very well (and is much simpler than poly-time elliptic algorithm), but is exponential in the worst-case!

# 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

# Example: Cryptarithmetic

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




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

# Real-world CSPs

- Assignment problems
  - e.g., who teaches what class? Which taxi cab gets to pick which passenger?
- Timetabling/scheduling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve *real-valued variables*

# Standard search formulation (incremental)

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

States are defined by the values assigned so far

- Initial state: the empty assignment { }
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment
  → fail if no legal assignments

- Goal test: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth *n* with *n* variables
   → use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. b = (n - L)*d at depth L, hence $n! \cdot d^n$ leaves
5. 

# 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
  → 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

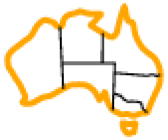
- Can solve *n*-queens for $n \approx 25$
  (strictly, can solve n-queens eventually for any n; but within reasonable time, for only very modest values of n – we'll see shortly how to improve on basic backtracking so we can solve for really big values of *n* )

# Backtracking Search

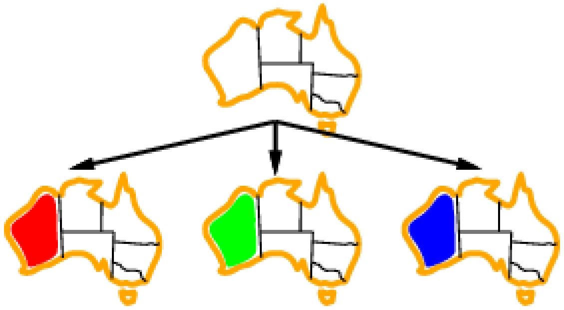
**function** BACKTRACKING-SEARCH( *csp*) **returns** a solution, or failure
  **return** RECURSIVE-BACKTRACKING({}, *csp*)

**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*

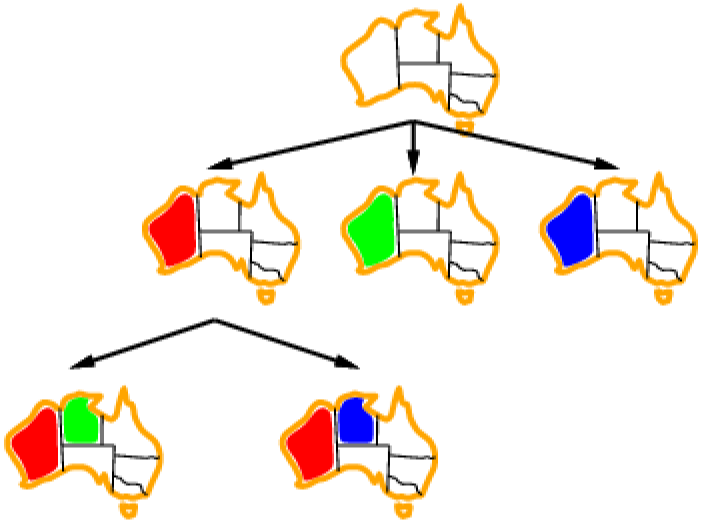
# Backtracking Example



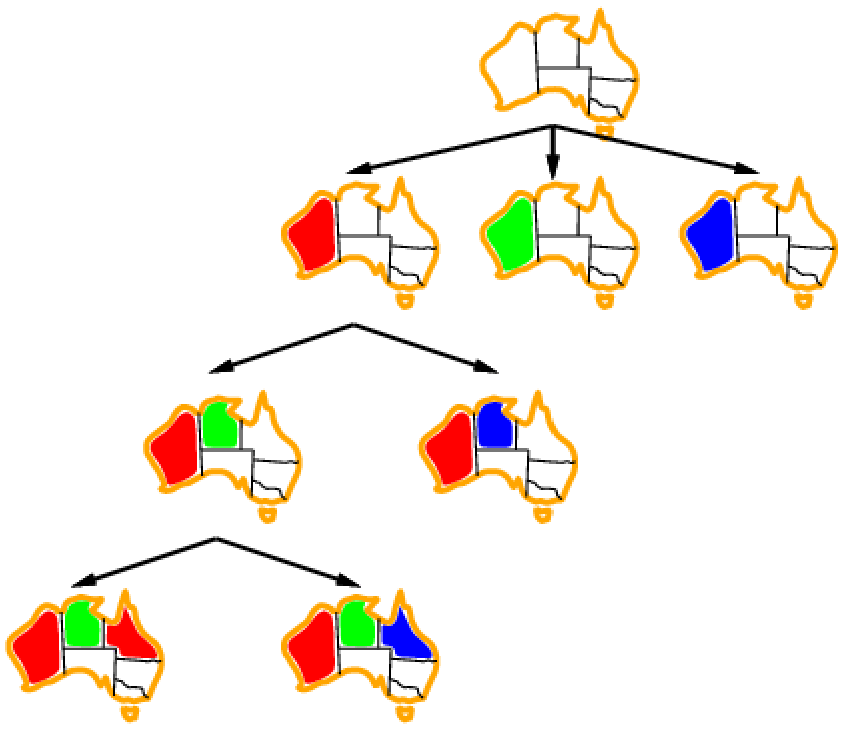
# Backtracking example



# Backtracking example



# Backtracking example



# Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?

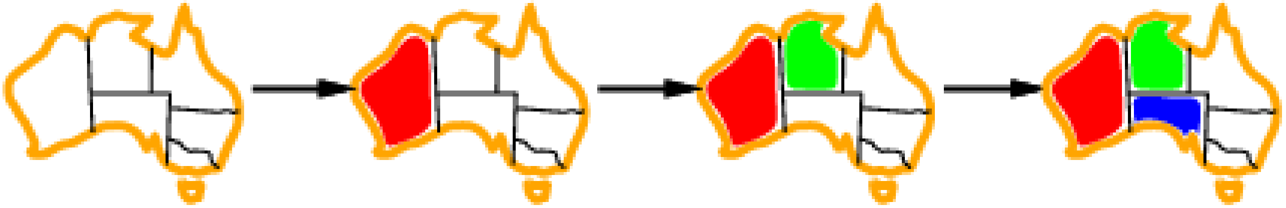
    -- left-to-right? more constrained before less constrained?
  - In what order should its values be tried?

    -- sometimes it does not matter, but often times it does

    -- use less used colors first? or more used colors first?
  - Can we detect inevitable failure early?

# Most constrained variable

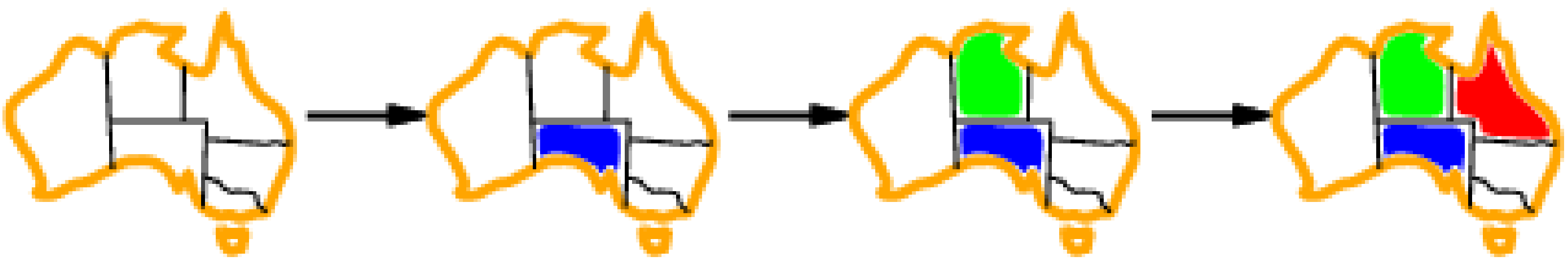
- Most constrained variable:
  choose the variable with the fewest permissible values



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

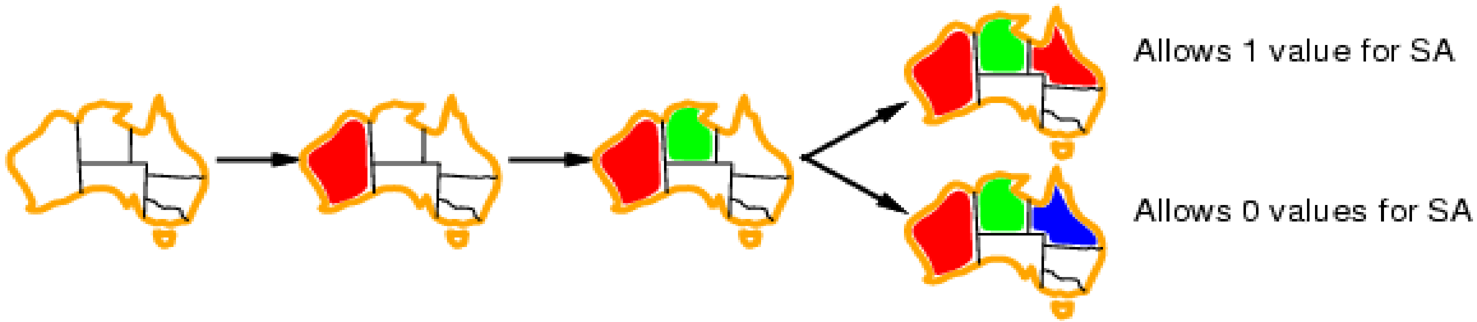
# Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables



# Least constraining value

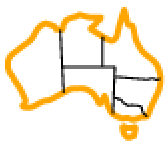
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

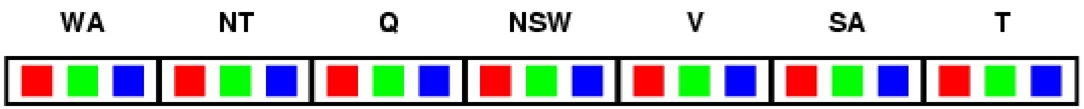



- Combining these heuristics makes 1000 queens feasible

# Forward checking

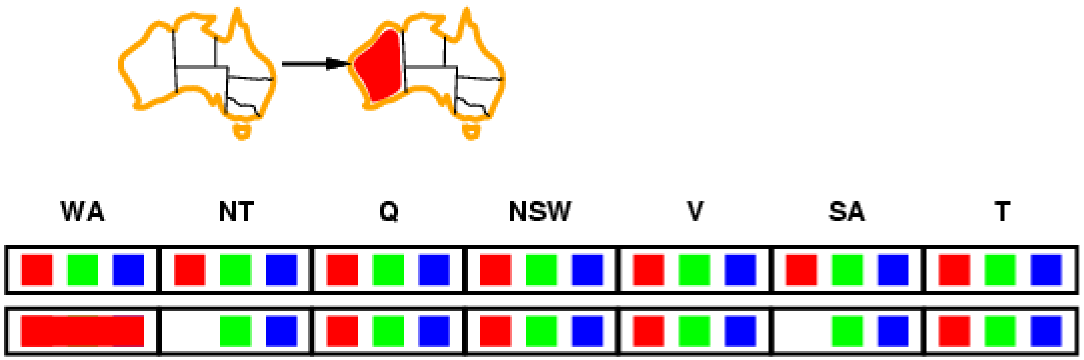
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values





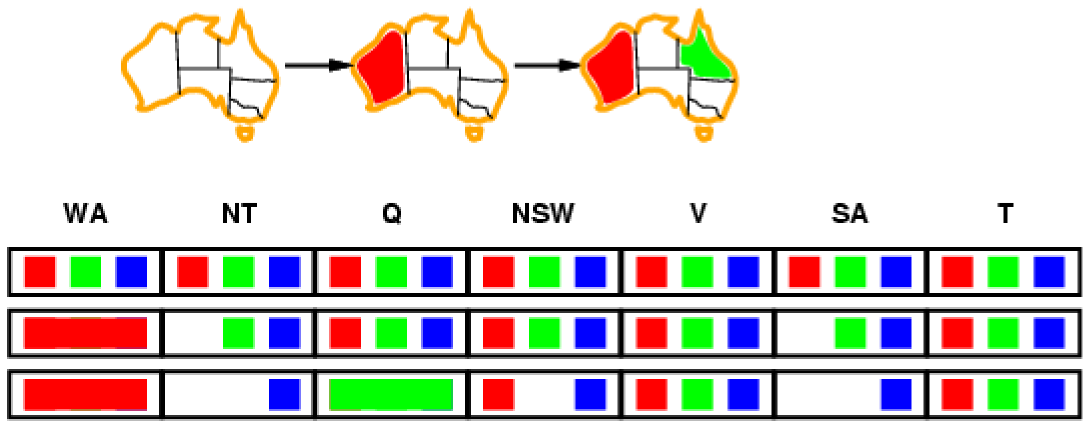
# Forward checking

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



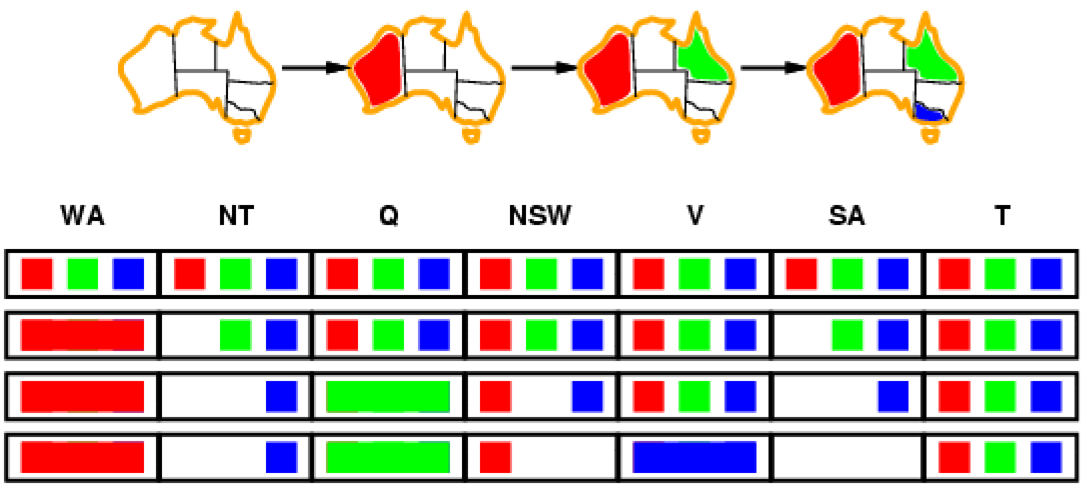
# Forward checking

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

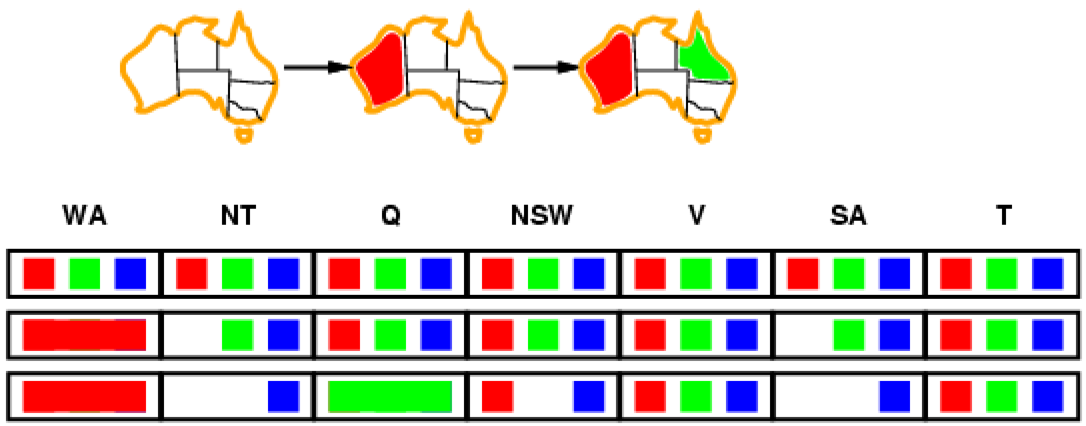


# Forward checking

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



# Constraint propagation

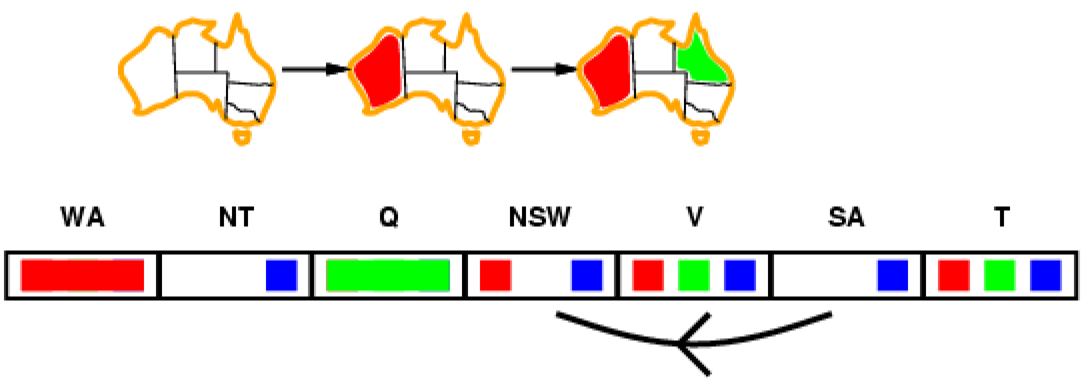
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:
- 

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

# Arc consistency

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

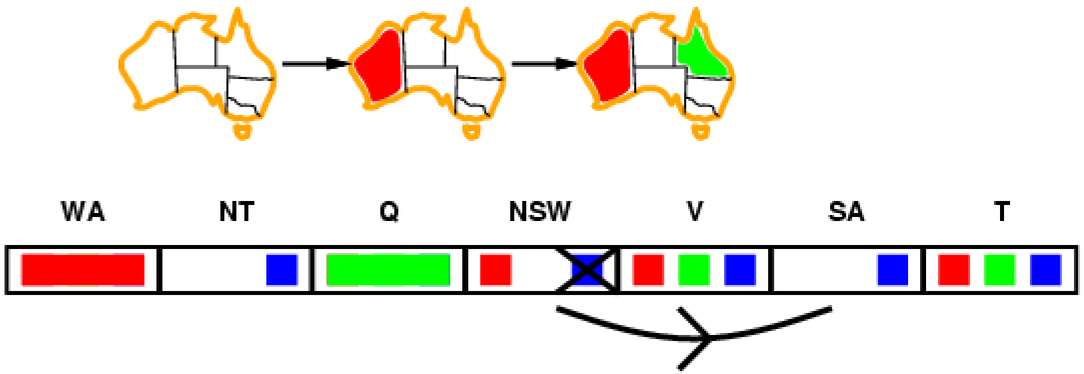
  for every value $x$ for variable $X$ there is some allowed $y$ for var. Y



# Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent if and only if

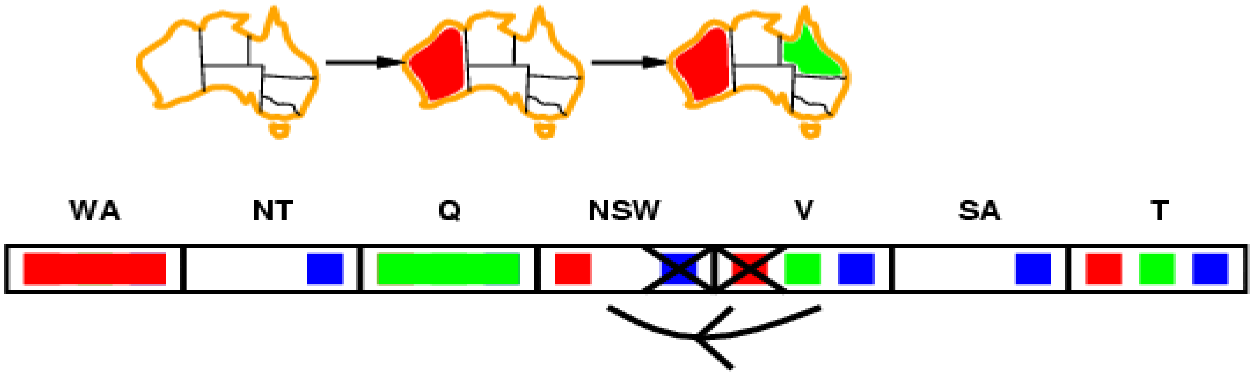
  for every value $x$ of $X$ there is some allowed $y$ for $Y$



# 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 $y$ for $Y$

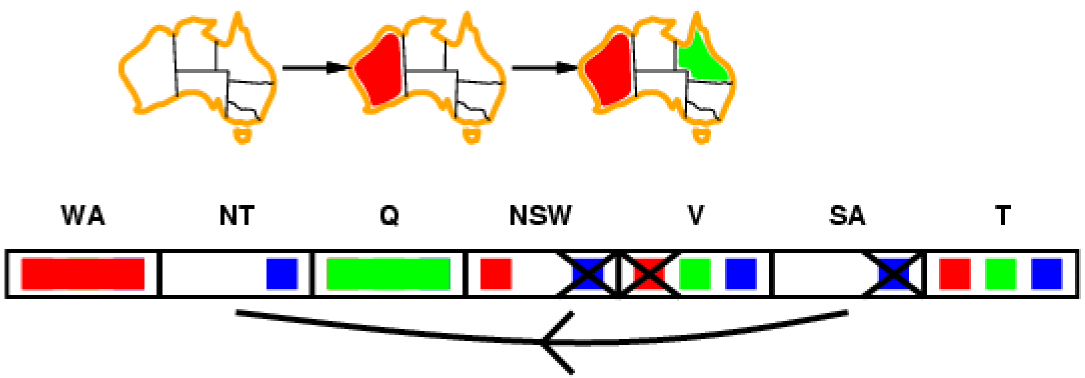



- If $X$ loses a value, neighbors of $X$ need to be rechecked

# 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 $y$ for $Y$




- 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

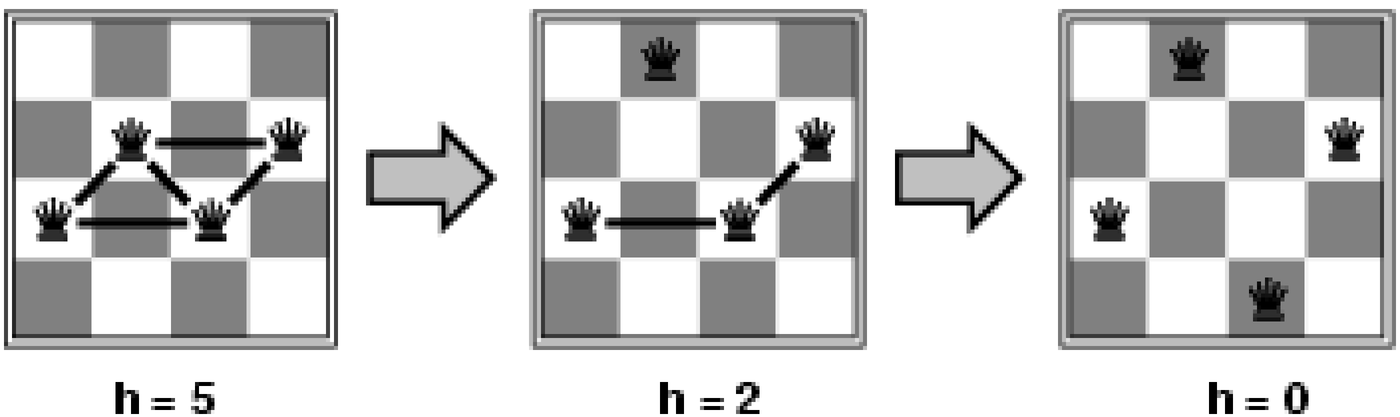
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- 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 a random initial state, can solve $n$-queens very quickly for (practically) 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
  - no objective fn. per se: any state satisfying *all* constraints is a goal state!
- 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
  (but: no guarantee of *always* finding a goal state in reasonable time)