323
ARTIFICIAL INTELLIGENCE & EXPERT SYSTEMS

Constraint Satisfaction Problems (CSPs)

Chapter 5

Dr. Zeynep ORMAN

Outline

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

Constraint satisfaction problems (CSPs)

- Problems can be solved by searching in a space of states.
- Standard search problem:
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test.
- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints C_m specifying allowable combinations of values for subsets of variables

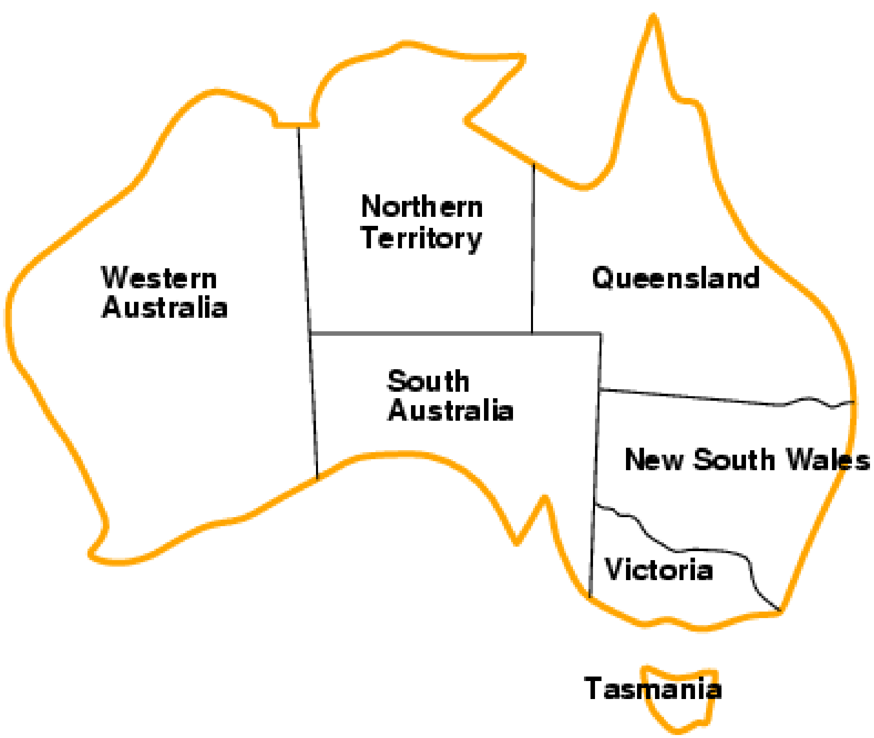
- Allows useful *general-purpose* algorithms with more power than standard search algorithms – *problem specific*.

Constraint satisfaction problems (CSPs)

- A state of the problem is defined by an assignment of values to some or all of the variables.$\{X_i=v_i, X_j=v_j, \ldots\}$
- An assignment that does not violate any constraints is called a consistent (legal) assignment.
- In a complete assignment every variable is mentioned.
- A solution to a CSP is a complete assignment that satisfies all the constraints.
- Some CSPs also require a solution that maximizes an objective function.

Example: Map-Coloring

A map of Australia showing each of its states and territories:

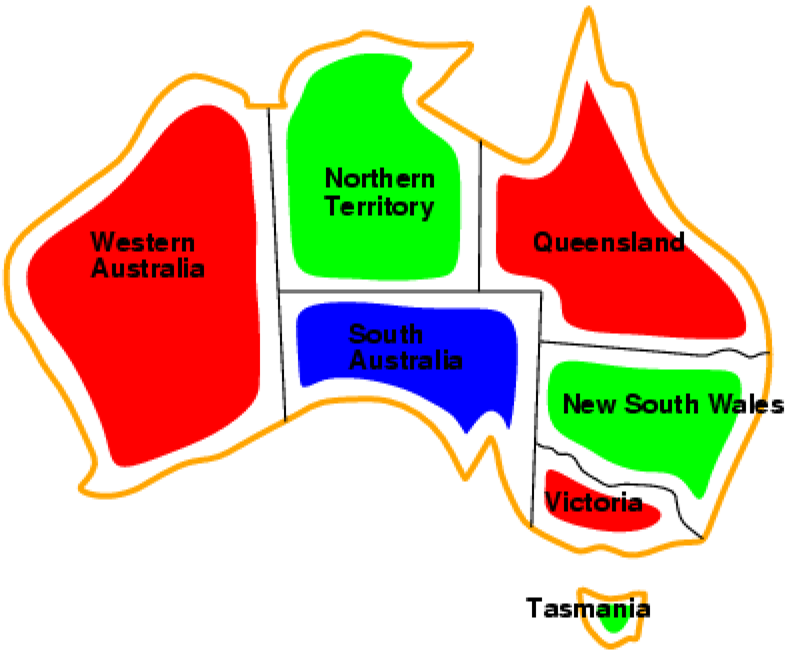


- Suppose that we are given the task of coloring each region either red,green or blue in such a way that no neighboring regions have the same color.
- To formulate this problem as a CSP, we first define
  - the variables,
  - the domain of each variable and
  - the constraints.

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),(blue,red),(blue,green)}

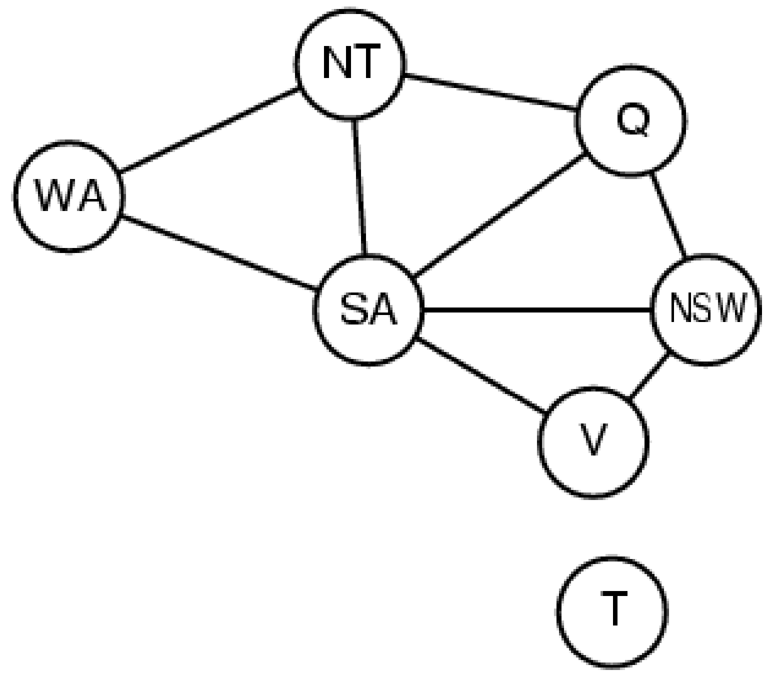
Example: Map-Coloring



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

Constraint graph

- Binary CSP: each constraint relates two variables.
- Constraint graph: nodes are variables, arcs are constraints.
- General-purpose CSP algorithms use the graph structure to speed up search – an exp. reduction in complexity.
  - e.g., Tasmania is an independent subproblem!



Varieties of CSPs

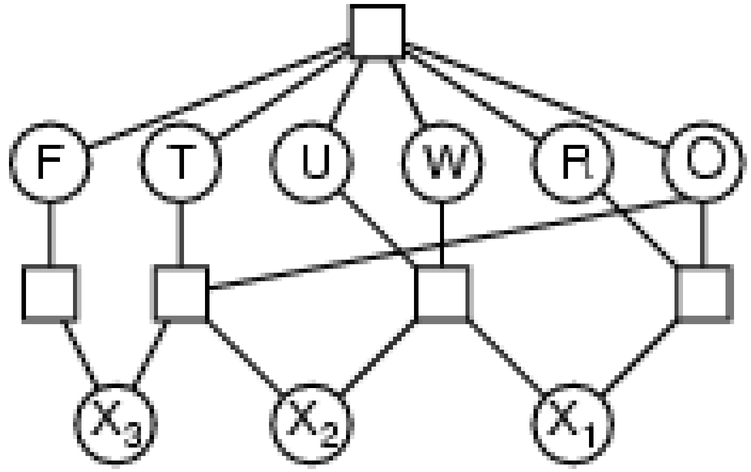
- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* → $O(d^n)$ complete assignments
    - exponential in the number of variables.
    - e.g., map coloring, 8-queens problem, Boolean CSPs.
  - infinite domains:
    - set of integers, set of strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- Continuous domains– common in the real world
  - e.g., start/end times for Hubble Space Telescope observations.
  - **linear programming** problems can be solved in time polynomial in the number of variables.

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
  - it can be represented as a constraint graph.
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic puzzles.
  - each letter represents a different digit.
  - The aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct.

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

Varieties of constraints

- Preferences (soft constraints),
  - e.g., red is better than green.
  - often representable by a cost for each variable assignment.
  - CSPs with preferences can be solved using optimization search methods.
    - e.g. in a university timetabling problem :
    - Prof. X might prefer teaching in the morning whereas Prof. Y prefers teaching in the afternoon.
    - A timetable that has Prof. X teaching at 2 p.m. would still be a solution but would not be an optimal one.
      - assigning an afternoon slot for Prof. X costs 2 points, where as a morning slot costs 1.

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- 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

Standard search formulation (incremental)

- Initial state: the empty assignment { }, all variables are unassigned.
- 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 – every state is a complete assignment that might or might not satisfy the constraints.
4. Local search methods work well with this formulation.

Standard search formulation (incremental)

- We gave a formulation of CSPs as search problems.
- Using this formulation, any of the search algorithms can solve CSPs.
  - Suppose we apply breath-first search.
  - The branching factor at the top level is nd – any of d values can be assigned to any of n variables.
  - At the next level, the branching factor is $(n-1)d$, and so on for n levels.
  - We generate a tree with $n!d^n$ leaves.
  - However, there are only d^n possible complete assignments!!!

Backtracking search

- Variable assignments are commutative.
  - A problem is commutative if the order of application of any given set of actions has no effect on the outcome.

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.
  - e.g. we might have a choice between *SA=red*, *SA=green* and *SA=blue*.
  - but, we would never choose between *SA=red* and *WA=blue*

→ b = d and there are d^n leaves.

Backtracking search

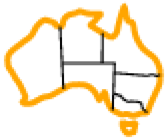
- Depth-first search for CSPs with single-variable assignments is called backtracking search.
- Backtracking search is the basic uninformed algorithm for CSPs.
- Can solve n-queens for $n \approx 25$.

Backtracking search **

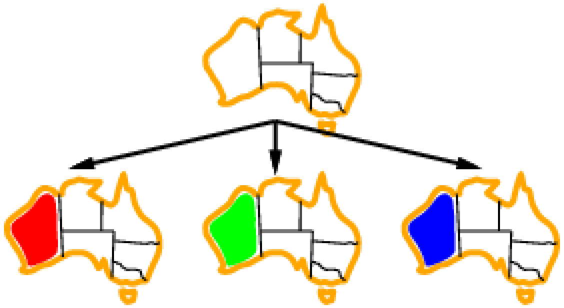
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

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
```

The algorithm is modeled on the recursive depth-first search.

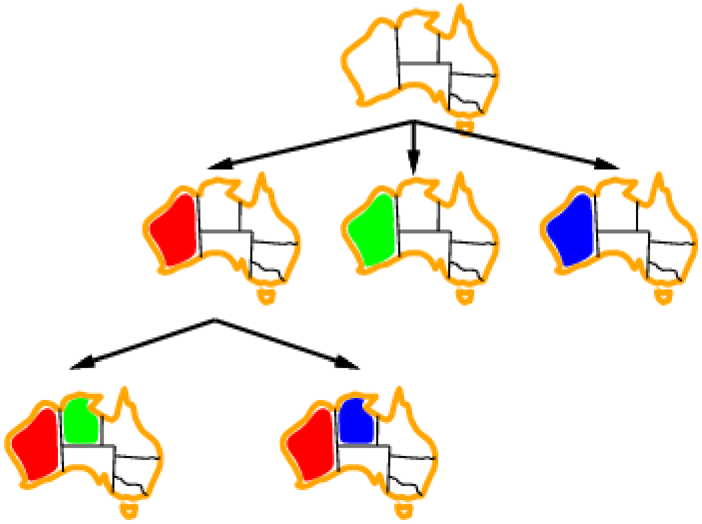
Backtracking example



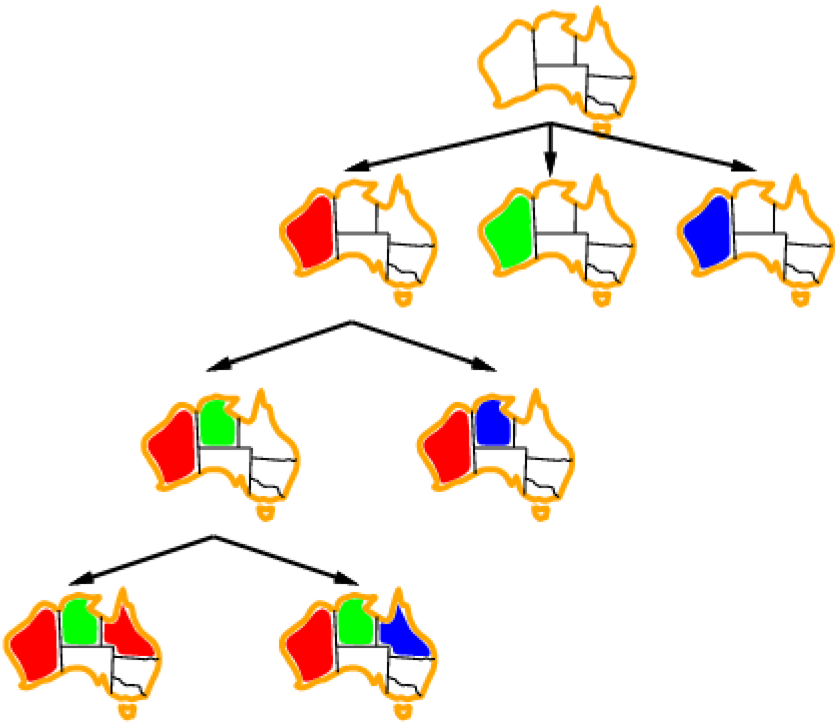
Backtracking example



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?
  - When a path fails – it means, a state is reached in which a variable has no legal values – can the search avoid repeating this failure in the subsequent paths?

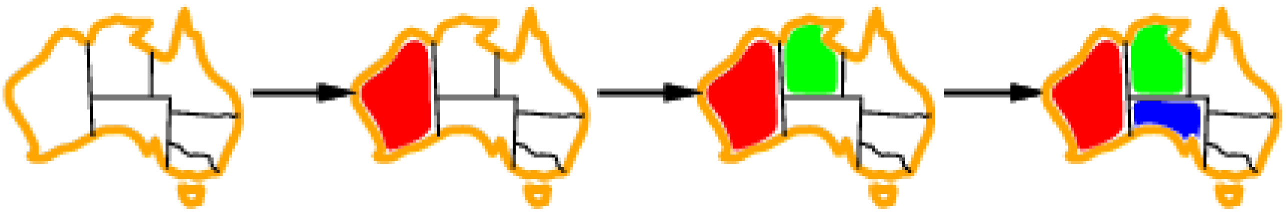
Most constrained variable

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(Variables[csp], assignment, csp)$

- Selects the next unassigned variable in the order given by the list *Variables[csp]*.
- This static variable ordering seldom results in the most efficient search.
  - e.g. after the assignments for *WA=red* and *NT=green*, there is only one possible value for *SA*.
  - So it makes sense to assign *SA=blue* next rather than assigning *Q*.
  - After *SA* is assigned, the choices for *Q*, *NSW*, and *V* are all forced.

Most constrained variable

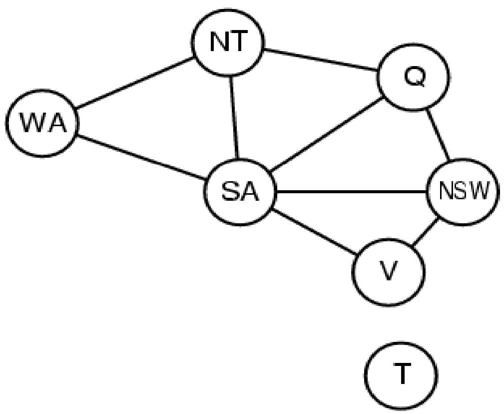
- Most constrained variable:
  choose the variable with the fewest legal values.



- minimum remaining values (MRV) heuristic.
  - It picks a variable that is most likely to cause a failure soon.
  - If there is a variable X with zero legal values remaining, the MRV will select X and failure will be detected immediately.

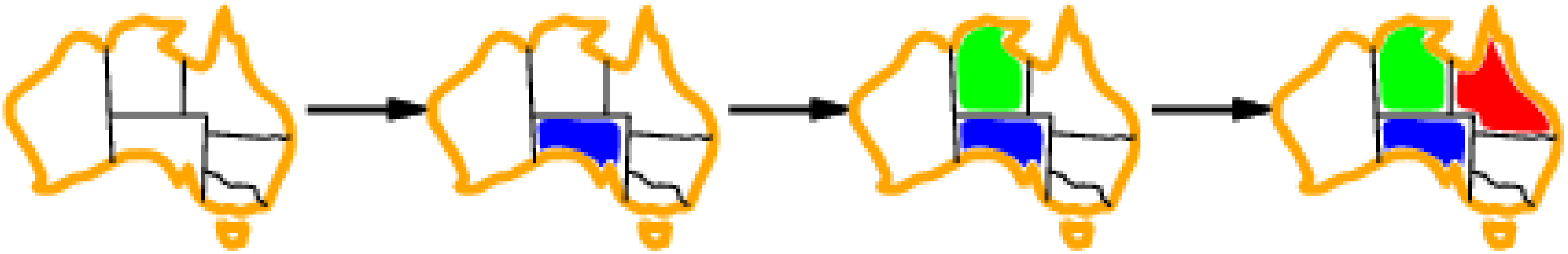
MRV heuristic

- The MRV heuristic doesn't help in choosing the first region to color in Australia because every region has 3 legal colors – degree heuristic.
- Degree heuristic tries to reduce the branching factor by selecting the variable that has the largest number of constraints on.
  - SA has the highest degree, 5; other variables have degree 2 or 3 and T has 0.



Most constraining variable

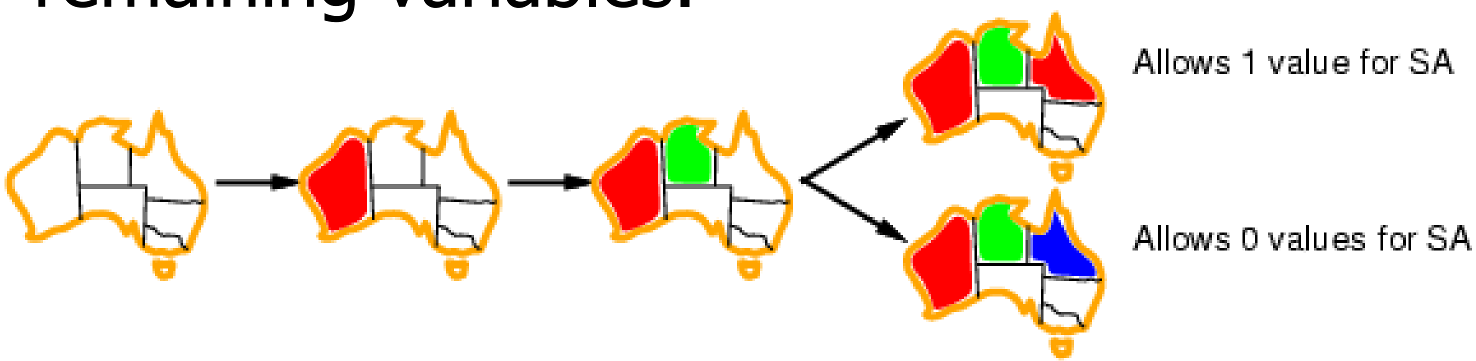
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables



- The MRV heuristic is a more powerful guide, but the degree heuristic can be useful as tie-breaker.

Least constraining value (LCV)

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



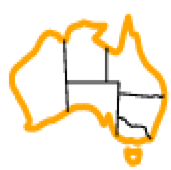
- In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Forward checking

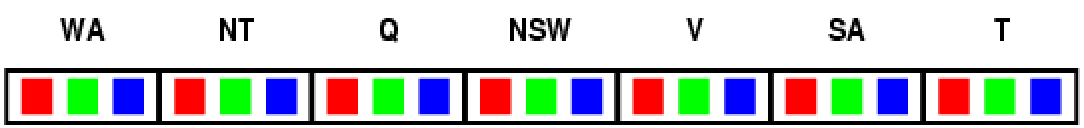
- So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by *SELECT-UNASSIGNED-VARIABLE*.
- However, by looking at some of the constraints earlier in the search or before the search has started, we can reduce the search space.
- One way to make better use of constraints during search is called forward checking.

Forward checking

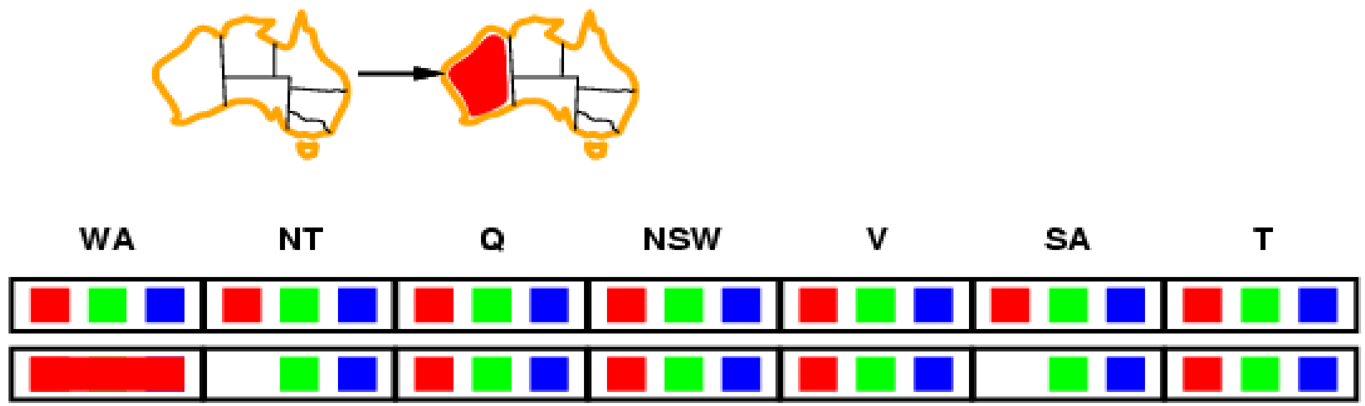
- Idea:
  - Keep track of remaining legal values for unassigned variables.
  - Terminate search when any variable has no legal values.



Initial domains



Forward checking

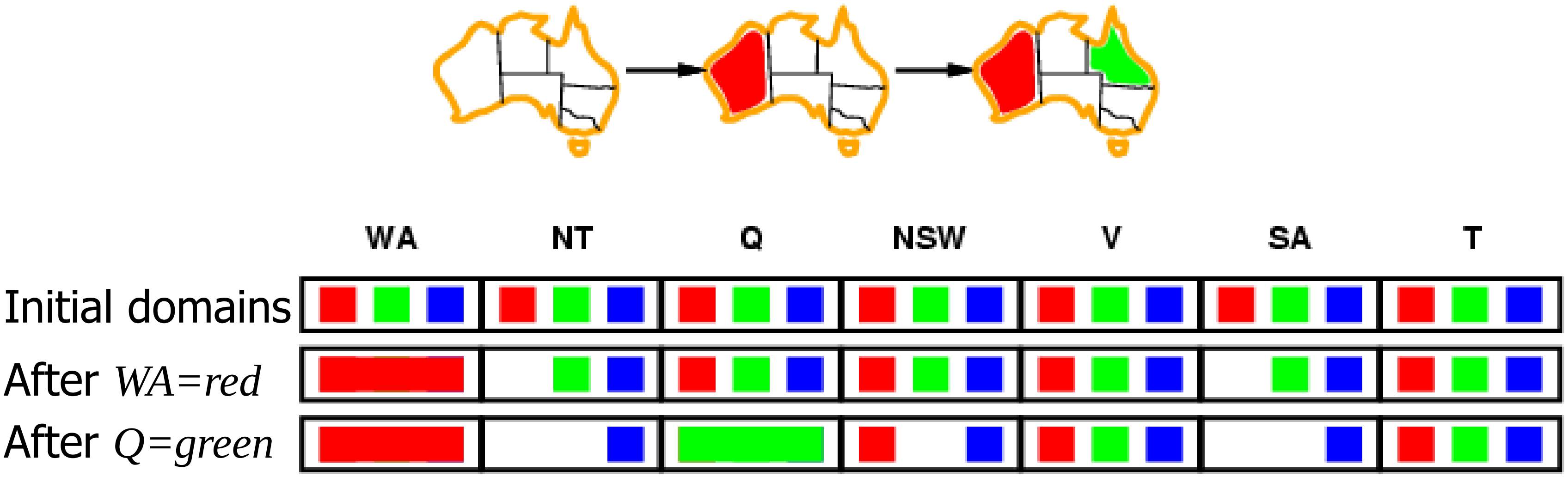


Initial domains

After *WA=red*

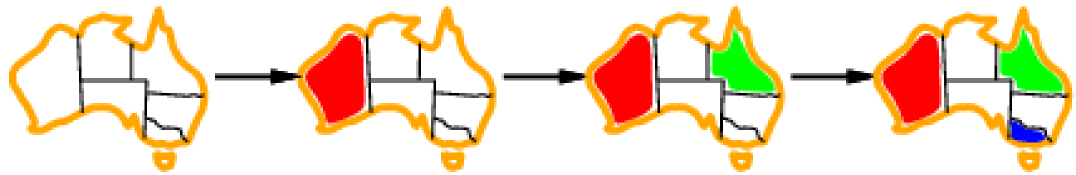
WA=red is assigned first; then forward checking deletes red from the domains of the neighboring variables *NT* and *SA*.

Forward checking



After *Q=green*, green is deleted from the domains of *NT*, *SA* and *NSW*.

Forward checking



Initial domains

After $WA=red$

After $Q=green$

After $V=blue$

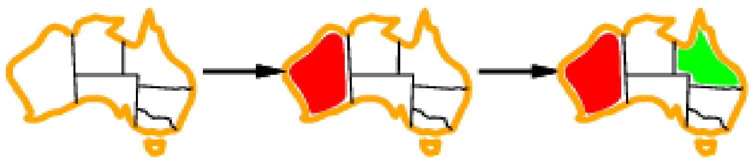
After $V=blue$, blue is deleted from the domains of NSW and SA, leaving SA with no legal values.

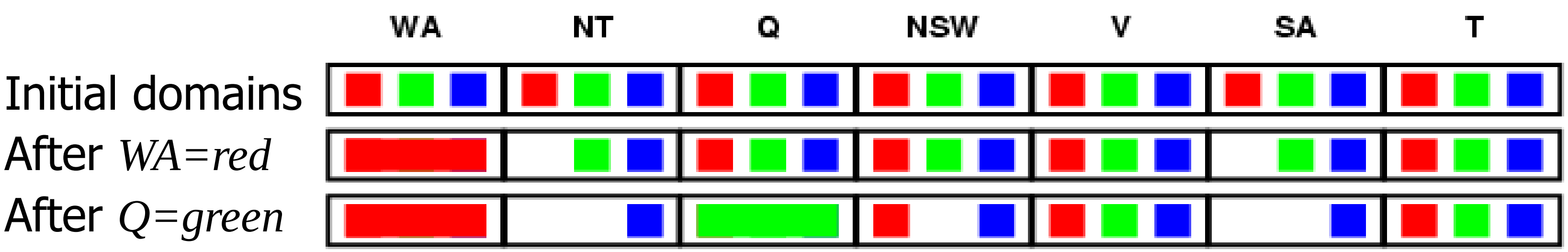
Forward checking

- Forward checking has detected that the partial assignment *{WA=red, Q=green, V=blue}* is inconsistent with the constraints of the problem.
- The algorithm will therefore backtrack immediately.
- Although forward checking detects many inconsistencies, it does NOT detect all of them.
  - e.g. when *WA=red* and *Q=green*, both *NT* and *SA* are forced to be blue.
  - but they are adjacent and they can not have the same value.
  - forward checking does not detect this as an inconsistency.

Constraint propagation

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



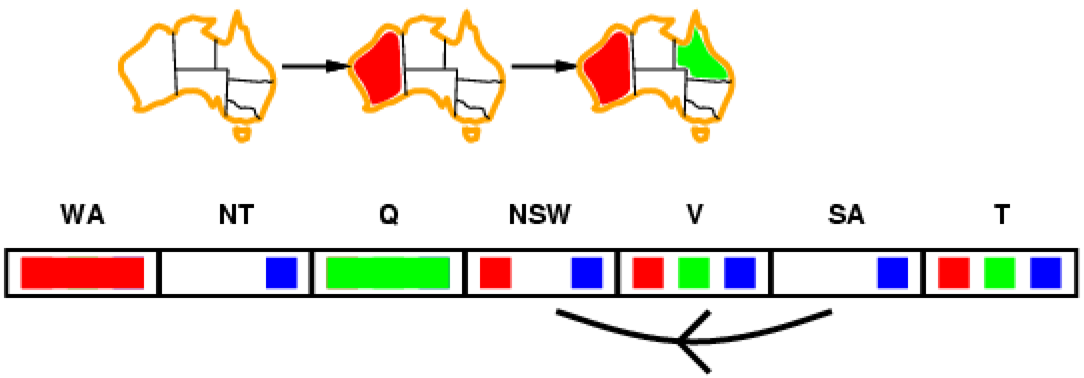


- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally.

Arc consistency

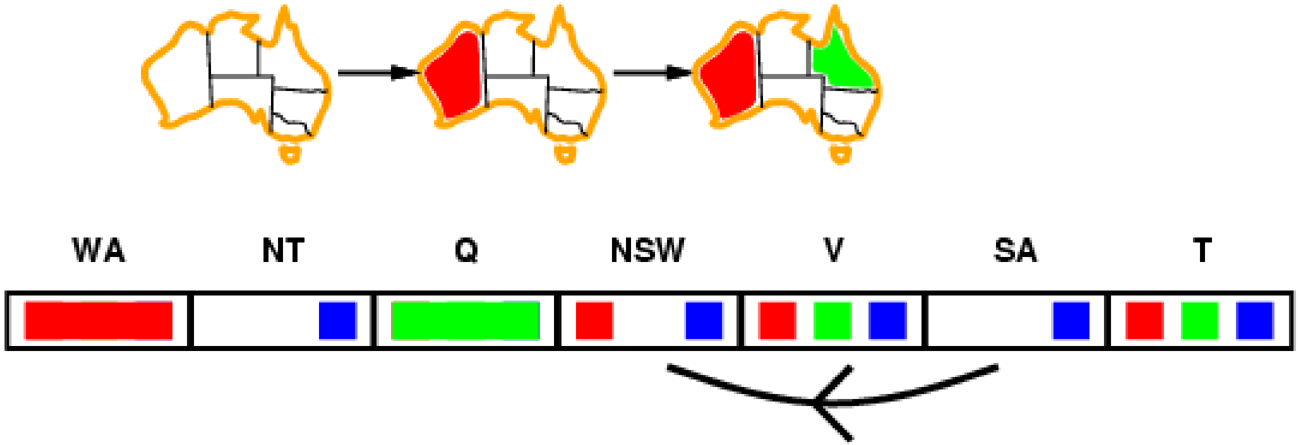
- Simplest form of propagation makes each arc consistent.
- $X \rightarrow Y$ is consistent iff

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



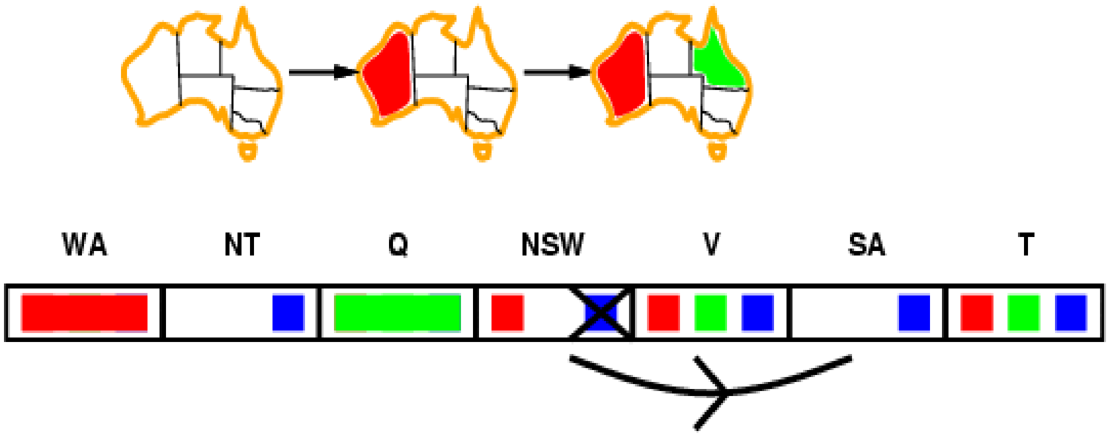
- e.g. the arc from SA to NSW.
  - the arc is consistent if for every value x of SA, there is some value y of NSW that is consistent with x.

Arc consistency



- The current domains: $SA=\{blue\}$ and $NSW=\{red,blue\}$.
- For $SA=\{blue\}$, there is a consistent assignment for NSW. It is $NSW=\{red\}$.
- Therefore, the arc from SA to NSW is consistent.

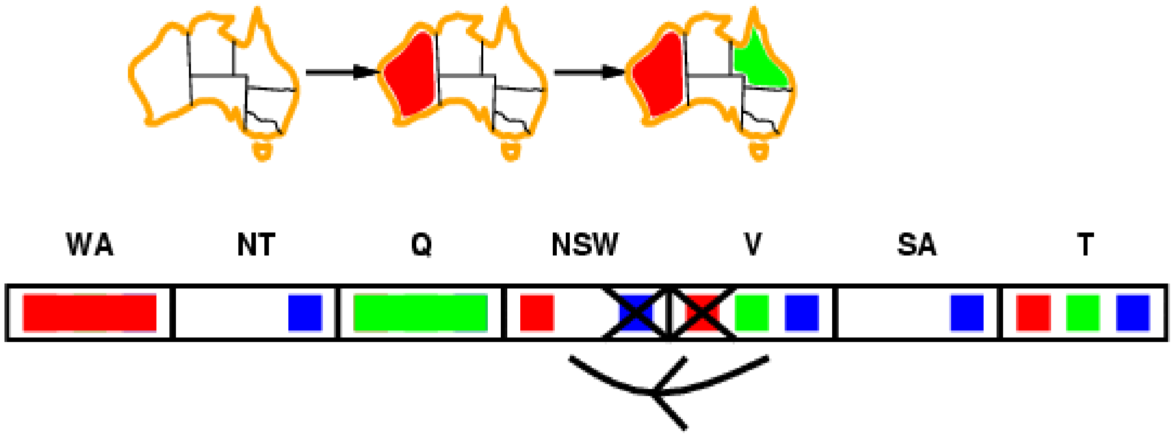
Arc consistency



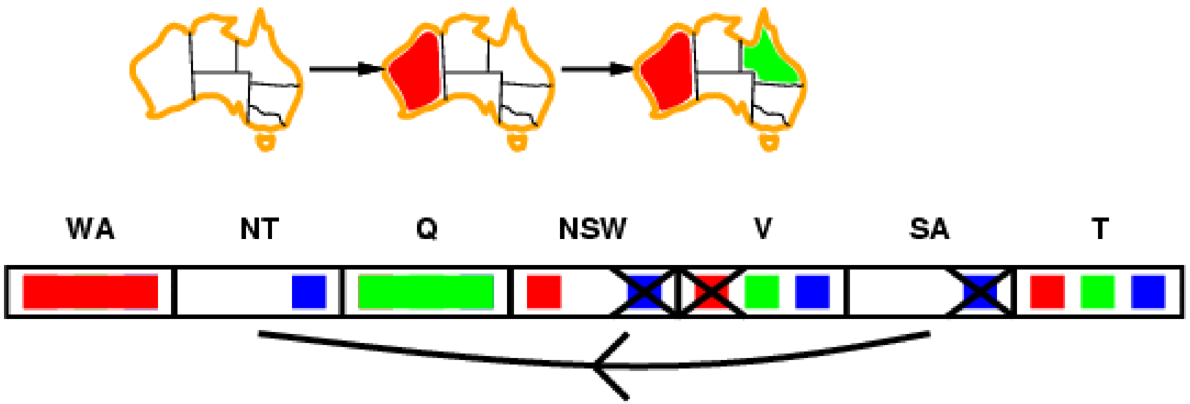
- The reverse arc from *NSW* to *SA* is NOT consistent.
- For the assignment *NSW=blue*, there is no consistent assignment for *SA*.
- The arc can be made consistent by deleting the value blue from the domain of *NSW*.

Arc consistency

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



Arc consistency



- Apply arc consistency to the arc from SA to NT.
- Both variables have the domain $\{blue\}$.
- The result is that blue must be deleted from the domain of SA, leaving the domain empty.
- 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
```

- AC-3 uses a queue to keep track of the arcs that need to be checked for inconsistency.
- After applying AC-3, either every arc is arc-consistent or some variable has an empty domain (thus the CSP cannot be solved).

Local search for CSPs

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

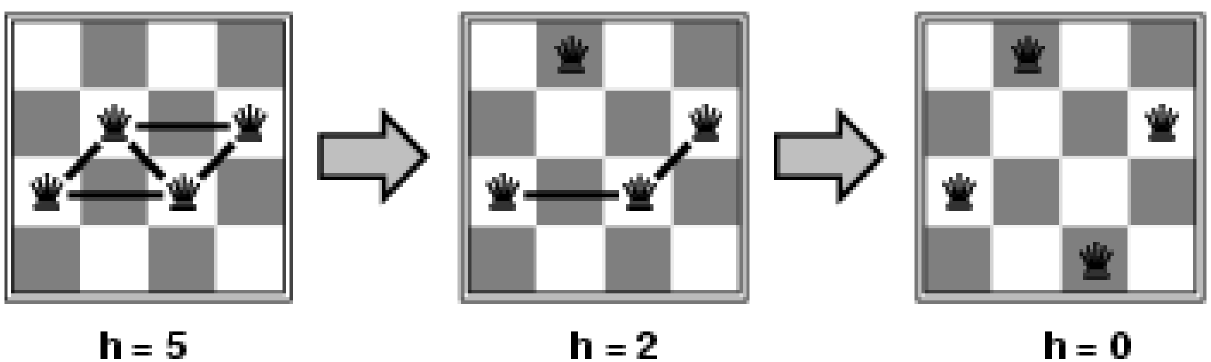
- To apply to CSPs:
  - allow states with unsatisfied constraints.
  - operators reassign variable values.

- 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.
  - CSP can be represented by a constraint graph.
- 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.
- Local search using the min-conflicts heuristic is usually effective in practice.