Constraint Satisfaction Problems

AIMA-3rd Chapter 6

Backtracking search algorithm

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

function BACKTRACK(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment then
           add {var = value} to assignment
           inferences ← INFERENCE(csp, var, value)
           if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                 return result
       remove {var = value} and inferences from assignment
   return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter **??**. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

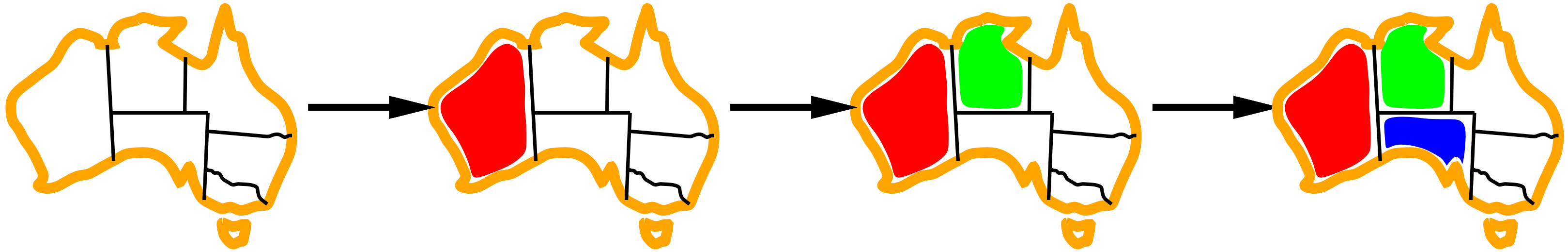
1. Which variable should be assigned next?
   $var \leftarrow \textsc{Select-Unassigned-Variable}(csp)$
2. In what order should its values be tried?
   $value \leftarrow \textsc{Order-Domain-Values}(var, assignment, csp)$
3. What inference should be performed at each step?
   $inferences \leftarrow \textsc{Inference}(csp, var, value)$
4. Can we detect inevitable failure early?
5. Can we take advantage of problem structure?

Variable ordering: Minimum remaining values

$var \leftarrow$ SELECT-UNASSIGNED-VARIABLE(csp)

Simplest: Static ordering

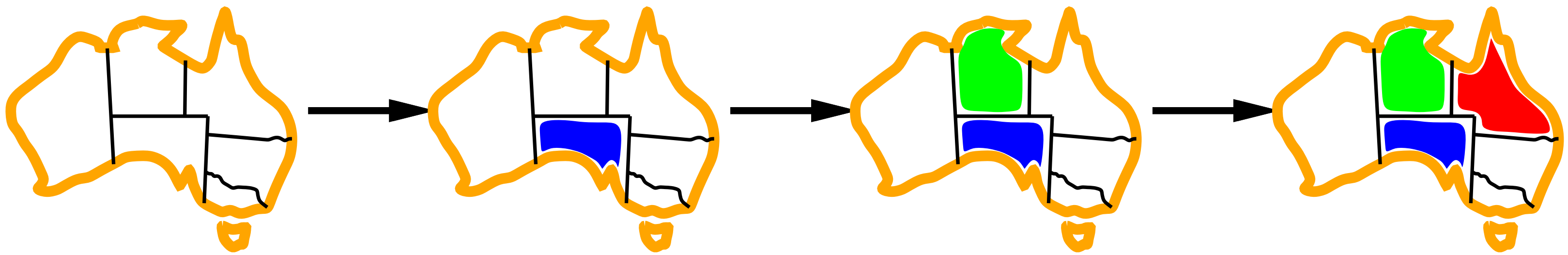
Better: Order by *minimum remaining values (MRV)*:
choose the variable with the fewest legal values



Variable ordering: Degree heuristic

How do we breat a tie among MRV variables?

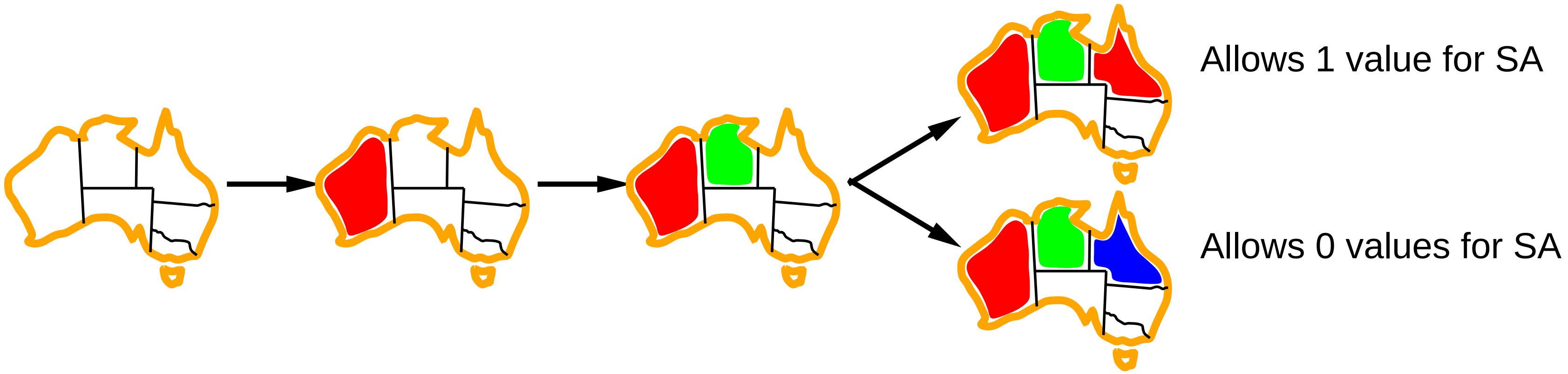
Degree heuristic:
    choose the *variable with the most constraints* on remaining variables



Value ordering: Least constraining value

$value \leftarrow \textsc{Order-Domain-Values}(var, assignment, csp)$

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



* If we want to enumerate all solutions, the value ordering is irrelevant.

* Combining these heuristics makes 1000 queens feasible

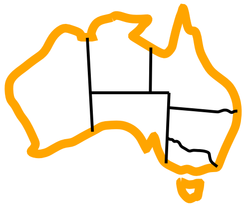
Inference: forward checking

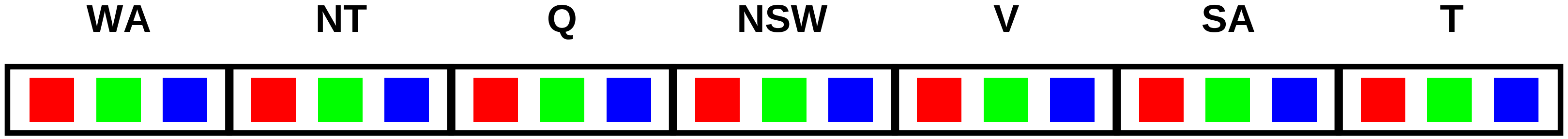
$inferences \leftarrow \textsc{Inference}(csp, var, value)$

A simplest type of inference that can be used with search is **Forward checking**.

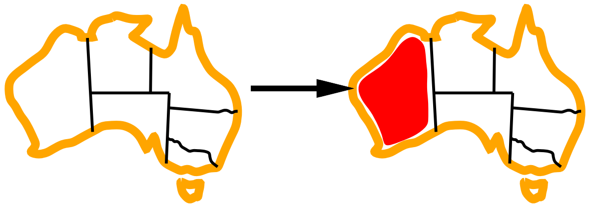
Idea: Whenever a variable X is assigned, establish *arc consistency* for it:
– for each unassigned variable Y that is connected to X by a constraint,
– delete from Y's domain any value that is inconsistent with the value chosen for X.
– terminate search when any variable has no legal values

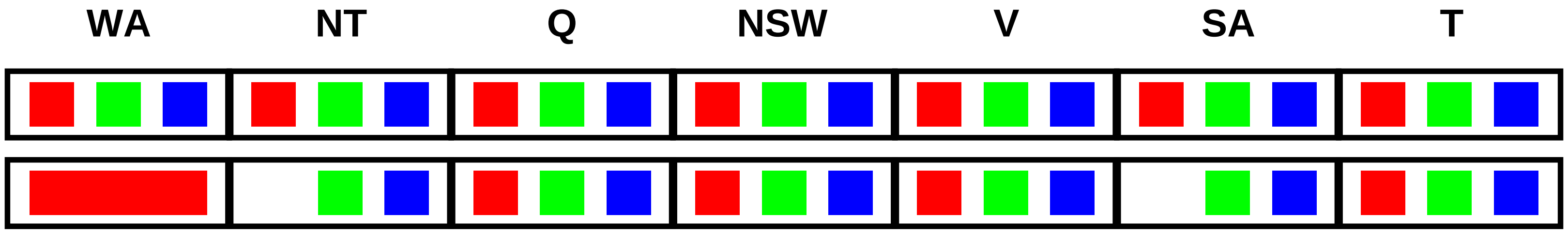
Forward checking example



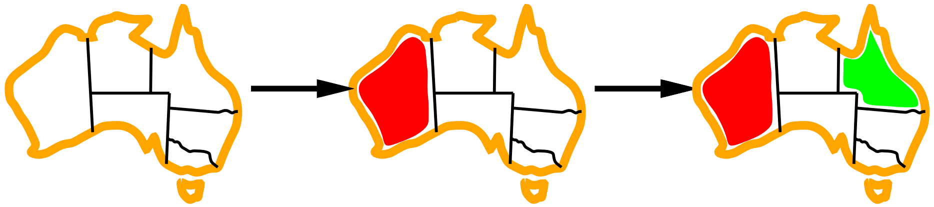


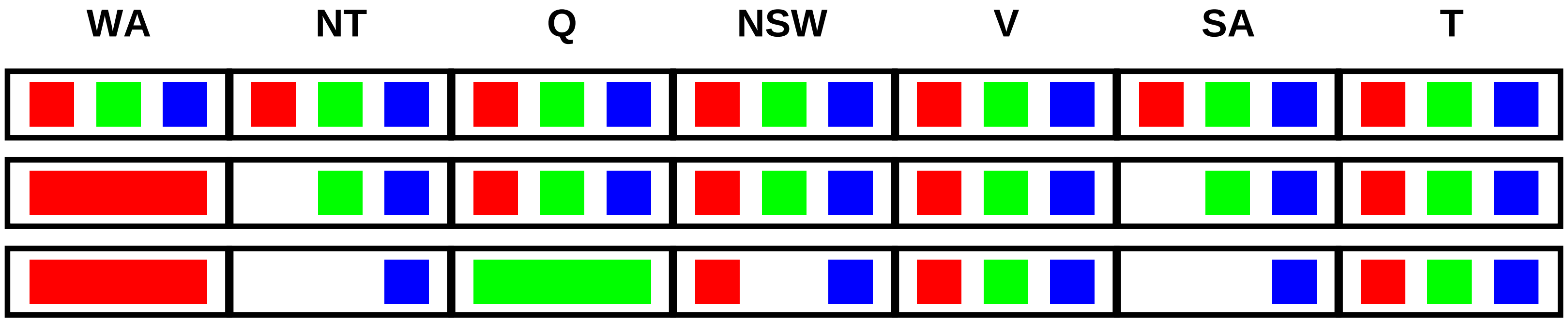
Forward checking example



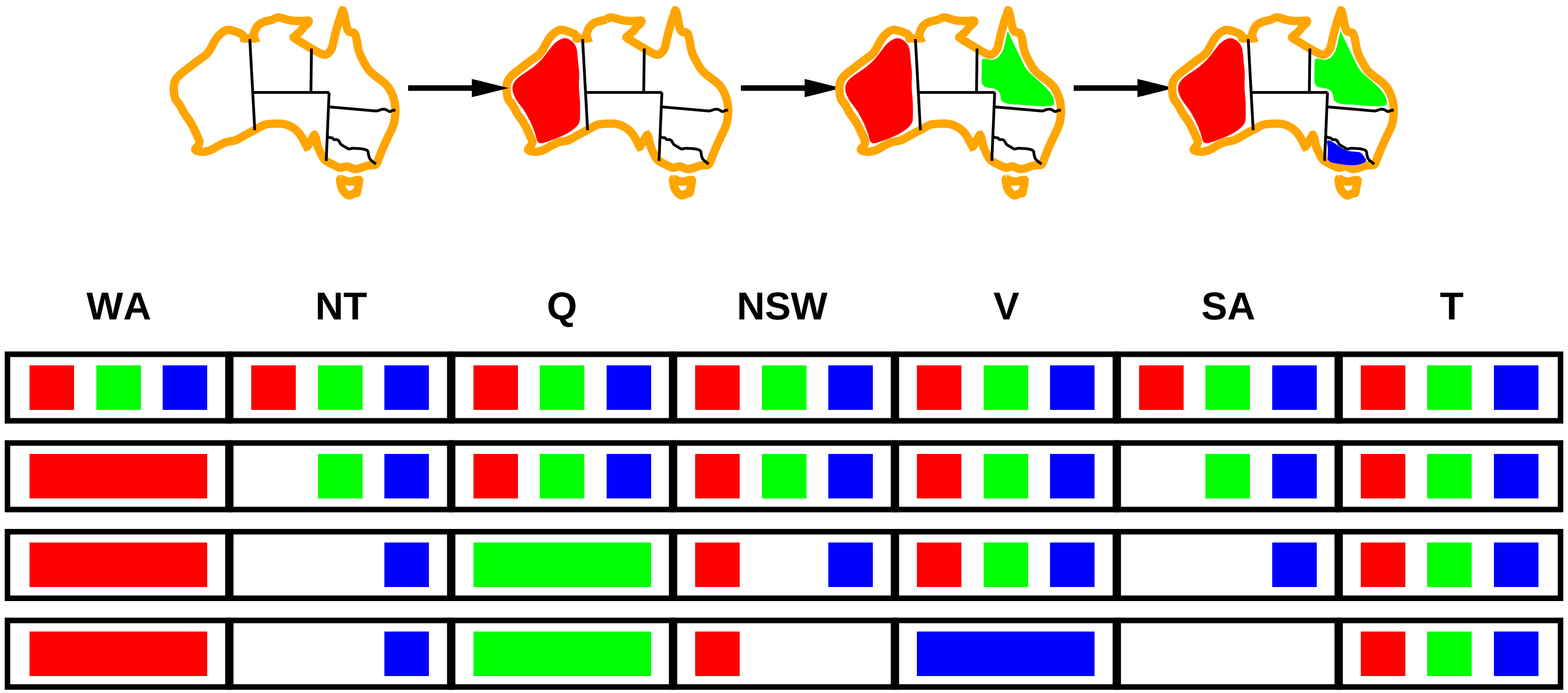


Forward checking example





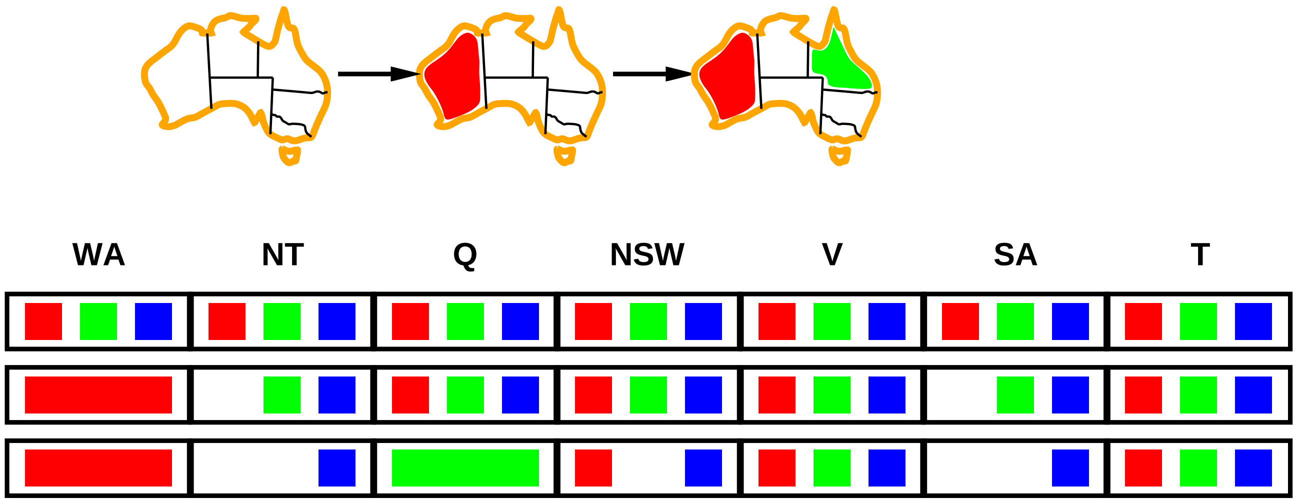
Forward checking example



Forward checking problem

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

- ◇ makes the current variable arc-consistent,
- ◇ but doesnt look ahead and make all the other variables arc-consistent.



NT and SA cannot both be blue!

Algorithm AC-3 (reminder)

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with components (X, D, C)
   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 REVISE(csp, Xi, Xj) then
         if size of Di = 0 then return false
         for each Xk in Xi.NEIGHBORS - {Xj} do
            add (Xk, Xi) to queue
   return true
---------------------------------------------------------------------------
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
   revised ← false
   for each x in Di do
      if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
         delete x from Di
         revised ← true
   return revised
```

Inference: Maintaining Arc Consistency (MAC)

A slightly better inference can be done via MAC.

Idea: After a variable X_i is assigend a value, the INFERENCE procedure calls AC-3 algo,

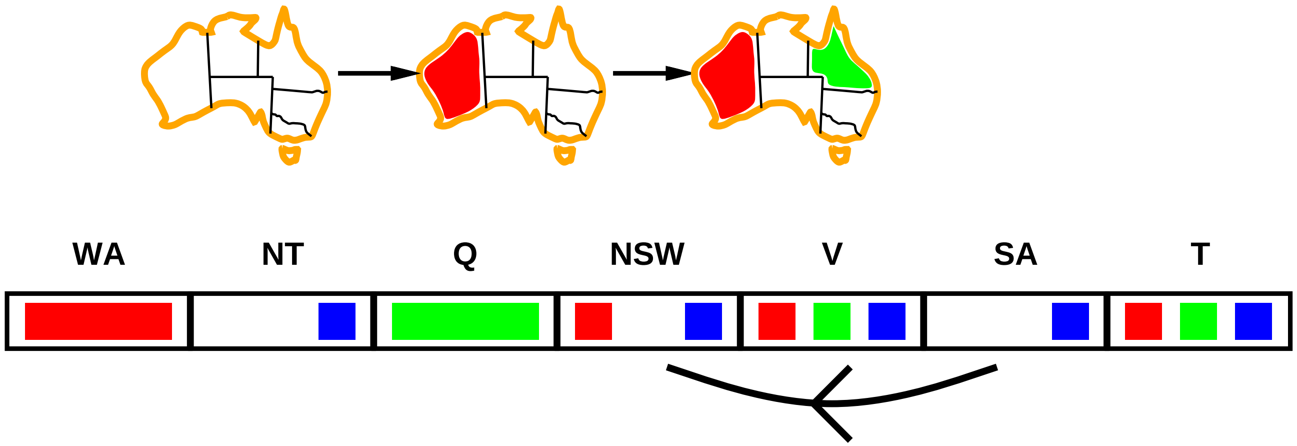
– but instead of a queue of all arcs in the CSP, start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.

– AC-3 then perform constraint propagation – if any variabvle has its domain reduce to a empty set, then AC-3 fails and we know to backtrack.

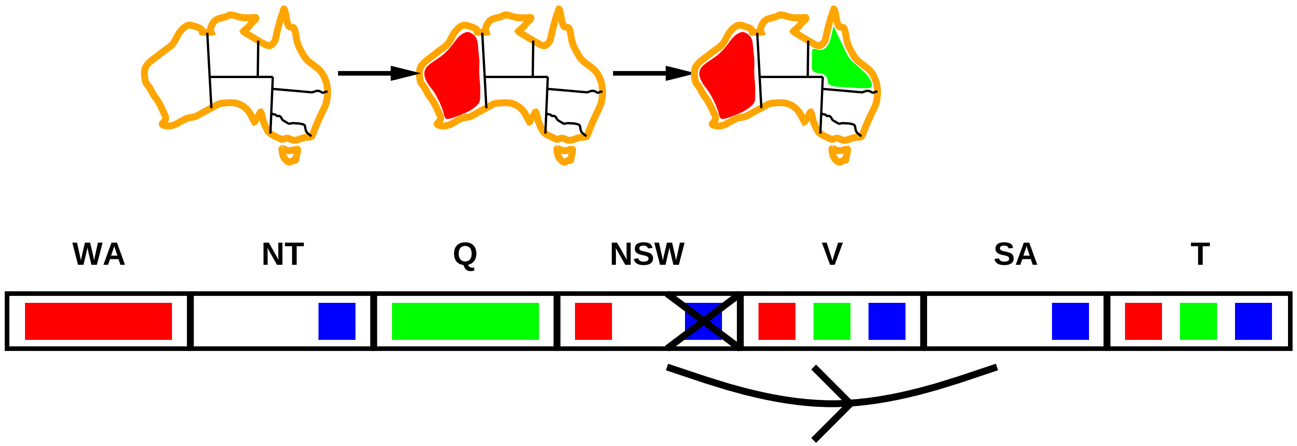
$X \rightarrow Y$ is consistent iff

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

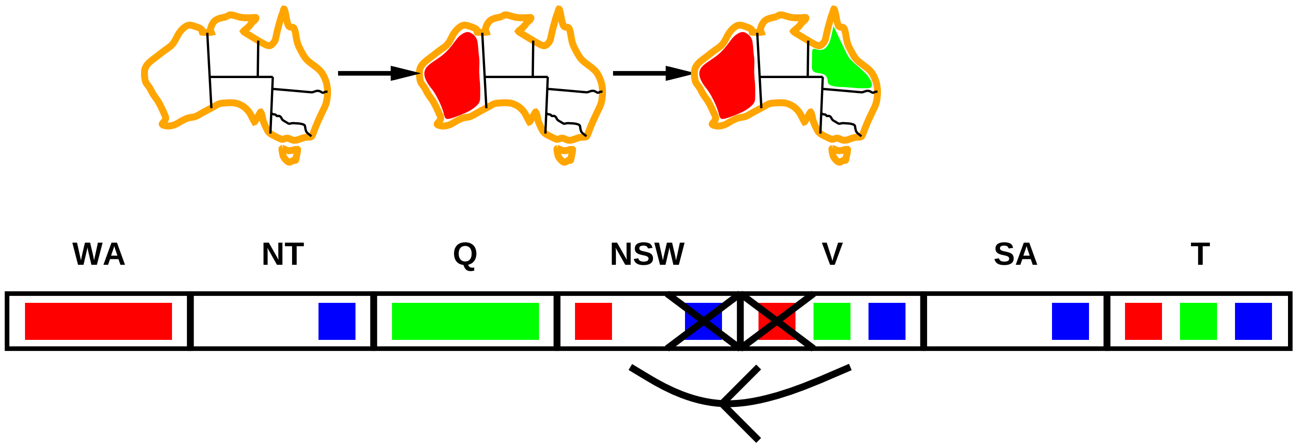
MAC example



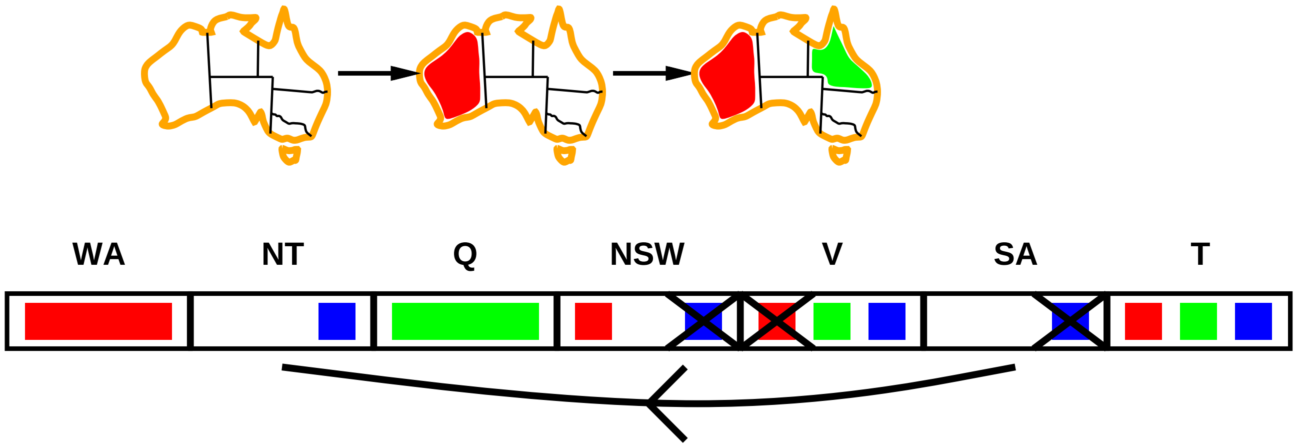
MAC example



MAC example

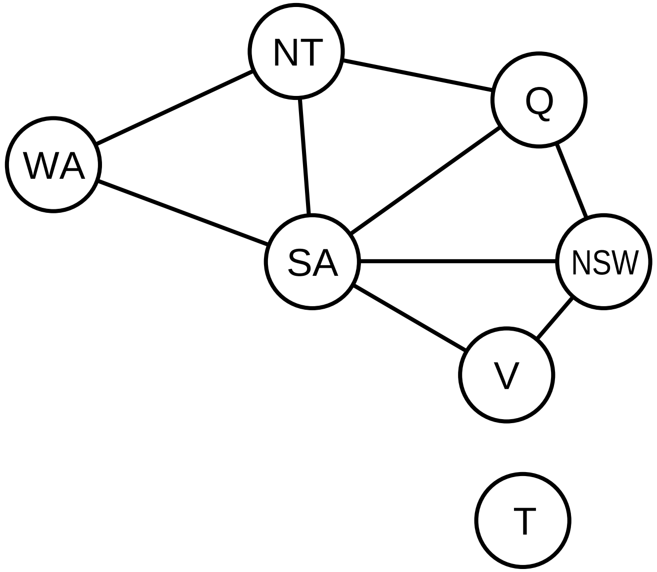


MAC example



Problem structure

Can we utilize the problem structure to find the solution faster?



Tasmania and mainland are **independent subproblems**

Identifiable as **connected components** of constraint graph

Problem structure contd.

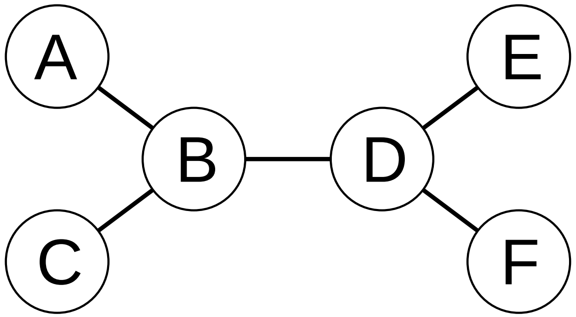
Suppose each subproblem has c variables out of n total variables

Then there are n/c subproblems. Each subproblem takes at most d^c work to solve, where d is the size of the domain.

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



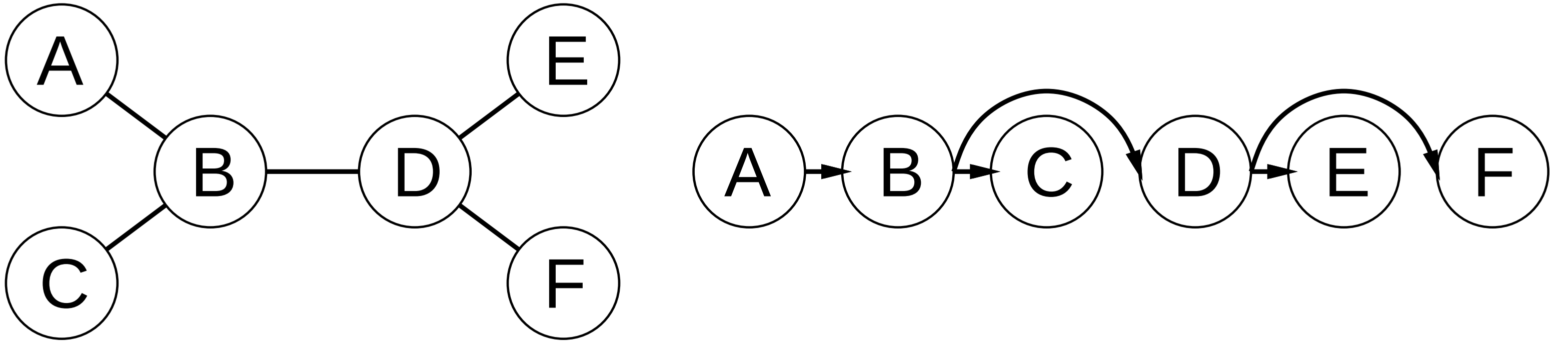
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time. Any tree with n nodes has $n-1$ arcs, make this graph in to directed tree in $O(n)$ steps, each of wich must compare up to d possible domain values for two variables.

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

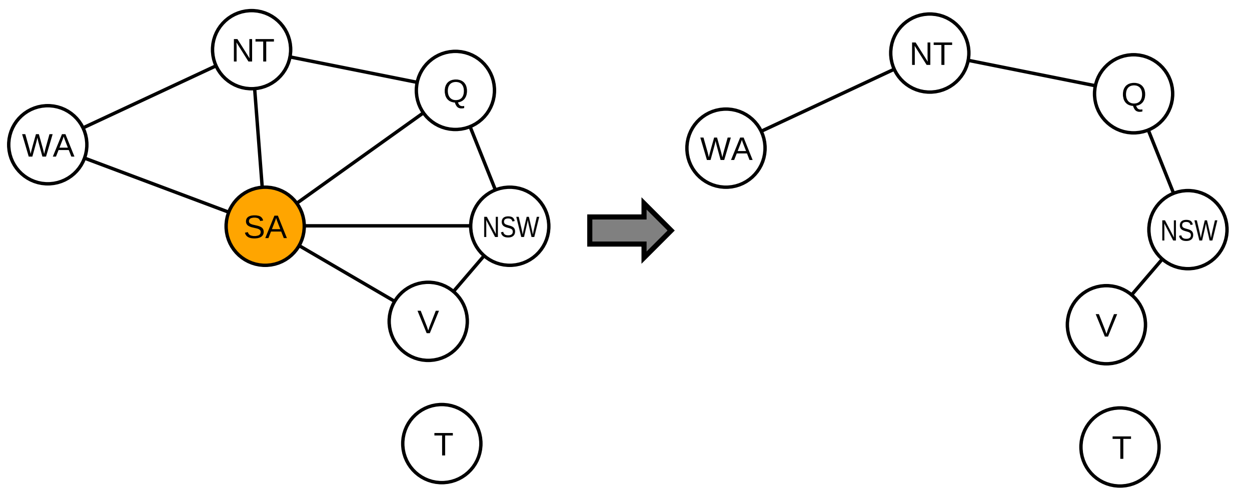


2. For j from n down to 2, apply REMOVEINCONSISTENT$(Parent(X_j), X_j)$

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

Reducing graphs to trees: Removing nodes

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

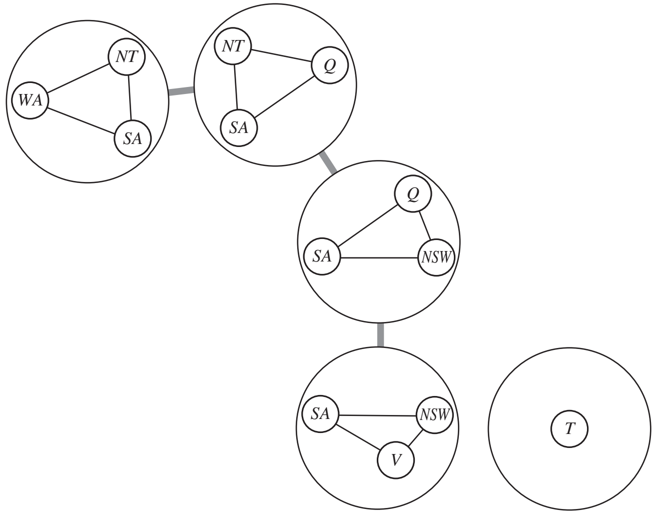
Reducing graphs to trees: Tree decomposition

Each subproblem is solved independently, and resulting solutions are then combined.
Conditions:

◊ Every variable in the original problem appears in at least one of the subproblems.

◊ If two variables are connected by a constraint in the original problem, they must appear together in at least one of the subproblems.

◊ If a variablea apears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblem.



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

Simple: Variable selection: randomly select any conflicted variable

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

Local Search for CSPs

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
   inputs: csp, a constraint satisfaction problem
           max-steps, the number of steps allowed before giving up
   local variables: current, a complete assignment
                    var, a variable
                    value, a value for a variable

   current ← an initial complete assignment for csp
   for i = 1 to max-steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var=value in current
   return failure
```

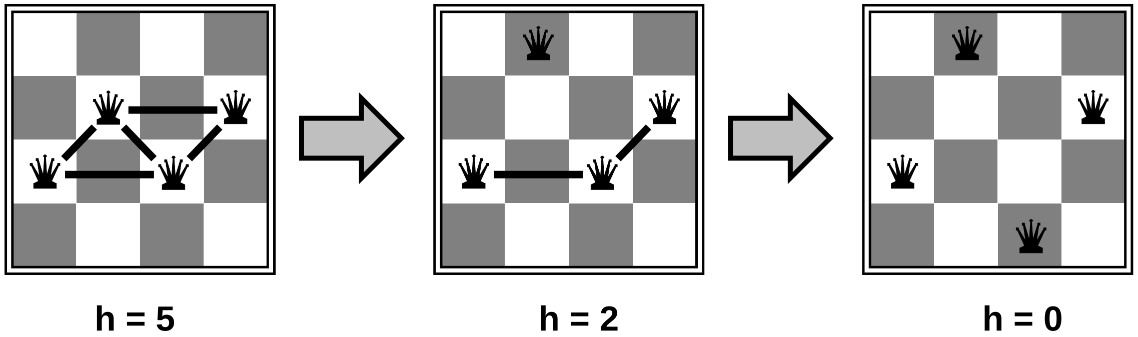
Example: 4-Queens

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



CSP 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

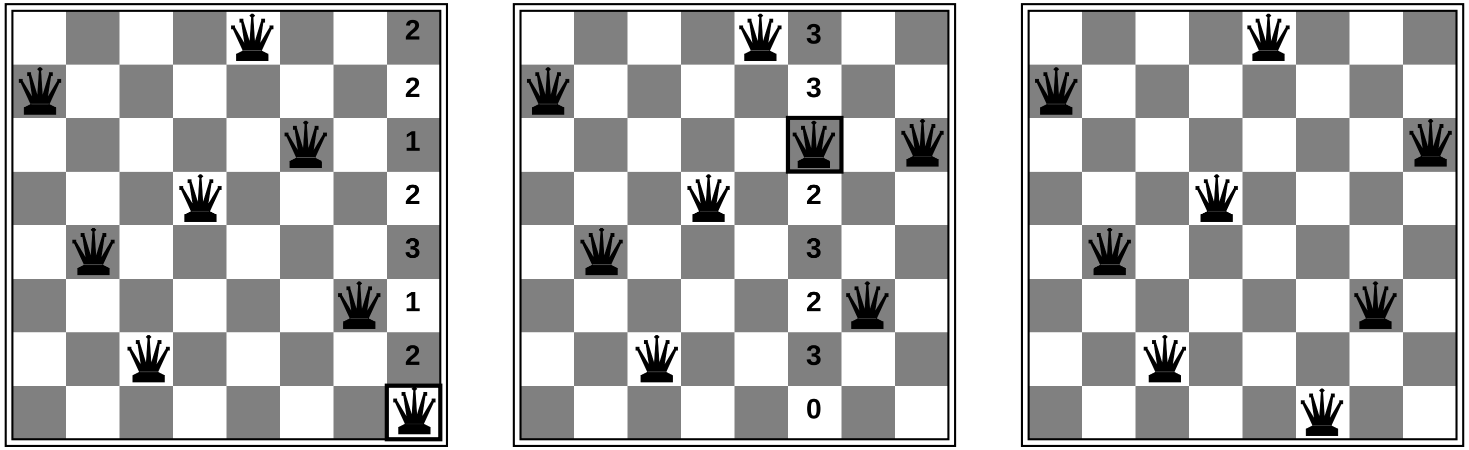
$Q_i \neq Q_j$ (cannot be in same row)

$|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

Translate each constraint into set of allowable values for its variables

E.g., values for (Q_1, Q_2) are $(1, 3)$ $(1, 4)$ $(2, 4)$ $(3, 1)$ $(4, 1)$ $(4, 2)$

Min-conflicts example: 8-Queens



At each stage, a queen is chosen for reassignment in its column. The number of conflicts (the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Min-conicts is surprisingly effective for many CSPs.

n-queens problem, the run time of min-conicts is roughly independent of problem size.

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