Artificial Intelligence

Constraint Satisfaction Problems

Summary

- ♢ CSP RN Chapter 5
- ♢ Constraint satisfaction problems
- ♢ Backtracking search for CSP
- ♢ Constraint propagation
- ♢ Local search for CSP
- ♢ Problem Structure

Constraint satisfaction problems (CSPs)

Standard search problem:

state is a "black box"— any data structure that supports goal test, eval, successor

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 v_i for subsets of variables

Solution is an *assignment* $\{X_i = v_j, \ldots\}$ that does not violate the constraints

Key feature: *general-purpose* algorithms with more power than standard search algorithms

Constraint satisfaction problems (CSPs) contd.

A CSP is a triple $\langle X, D, C \rangle$, where:

- $X = \{X_1, \ldots, X_n\}$ is the set of variables;
- $D = \{D_1, \ldots, D_n\}$ is a set of domains, specifying the allowed values for the variables;
- $C = \{C_1, \ldots, C_n\}$ is a set of constraints,
  where $C_i = \langle X_i^k, R_i^k \rangle$
  X_i^k is a subset of k elements of X
  R_i^k is a k-order relation over X_i^k

Varieties of CSPs

Discrete variables

◇ finite domains, are the most common
size $d \Rightarrow O(d^n)$ complete assignments
e.g., Boolean CSPs,(i.e., variables in $\{true, false\}$)
includes Boolean satisfiability (NP-complete)

Other kinds of CSPs

◇ infinite domains (integers, strings, etc.)

CSPs cannot be solved by enumerating assignments

e.g., job scheduling, var = start/end days for each job

need a constraint language,

e.g., $StartJob_1 + 5 \leq StartJob_3$

linear constraints solvable, **nonlinear** undecidable

Infinite domains can be reduced to finite by putting an upper bound to values.

Other kinds of of CSPs contd.

◇ Continuous variables

e.g., start/end times for Hubble Telescope observations

linear constraints solvable in poly time by
Linear Programming methods

Continuous domains are not considered here

$\Rightarrow$ Operation Research

Arity of constraints

Unary constraints involve a single variable,
e.g., $SA \neq green$

Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
e.g. $alldiff(X_1, \ldots, X_n)$

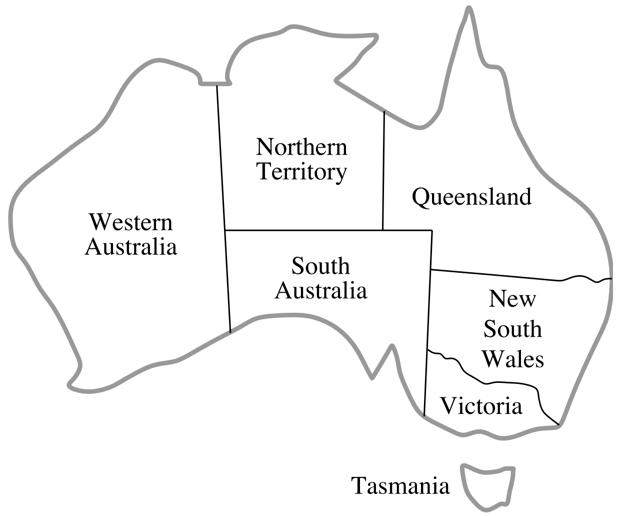
Higher-order constraints can be reduced to binary constraint (by increasing the number of variables and of constraints)

Unary constraints can be treated as domain restrictions

Preference constraints

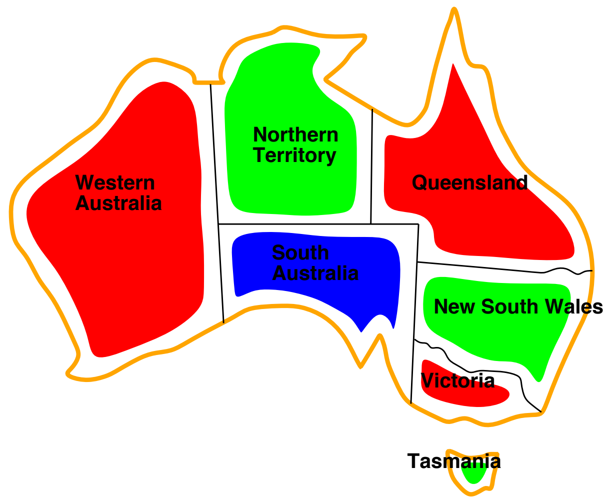
Preferences or soft constraints (versus absolute constraints),
e.g., red is better than $green$,
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems (COP)

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
$(WA, NT) \in \{(red, green), (red, blue), (green, red),$
$(green, blue), \ldots\}$

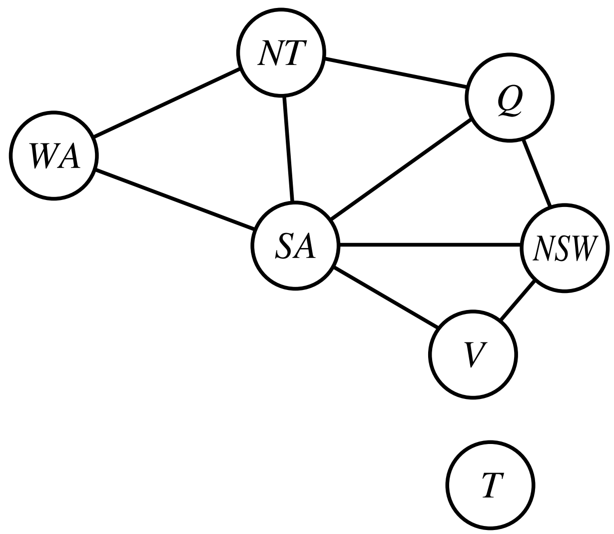
Example: Map-Coloring contd.



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

Constraint graph

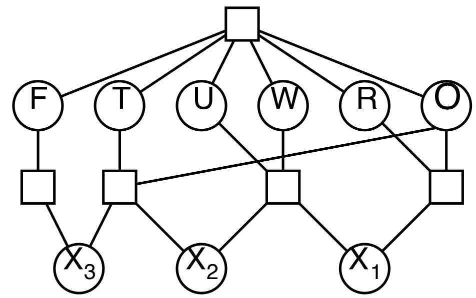
Binary CSP: each constraint relates at most two variables
Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure
to speed up search. E.g., Tasmania is an independent subprob-
lem!

Example: Cryptarithmetic

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



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

$\mathit{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, ... etc.

Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Variables: $A_1 \ldots A_9, \ldots, I_1 \ldots\ I_9$

Domains: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

alldiff$(A_1 \ldots A_9)$, ... etc. *alldiff*$(A_1 \ldots I_1)$, ... etc.

alldiff$(A_1 \ldots A_3, B_1 \ldots B_3, C_1 \ldots C_3)$, ... etc.

Real-world CSPs

Assignment problems (e.g., who teaches what class)

Timetabling problems (e.g., which class when and where)

Hardware configuration

Vehicle routing

Transportation scheduling

Factory scheduling

Floorplanning

Several real-world problems involve real-valued variables

Standard search formulation (incremental)

Straightforward approach:
States are defined by the values assigned so far

♢ Initial state: the empty assignment, $\{\ \}$

♢ Successor function: assign value to unassigned variable that does not conflict with current assignment.
$\Rightarrow$ fail if no legal assignments (not fixable!)

♢ Goal test: the current assignment is complete

Standard search formulation consequences

1) This is the same for all CSPs!

2) Solutions are all at depth n with n variables $\Rightarrow$ use DFS

3) Path is irrelevant, so can also use complete-state formulation

4) branching $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

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

$\Rightarrow b = d$ 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 $n \approx 25$

Backtracking search contd.

Backtracking = depth search

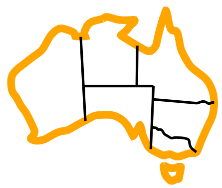
1) fixed variable order
2) only legal successors

Backtracking search

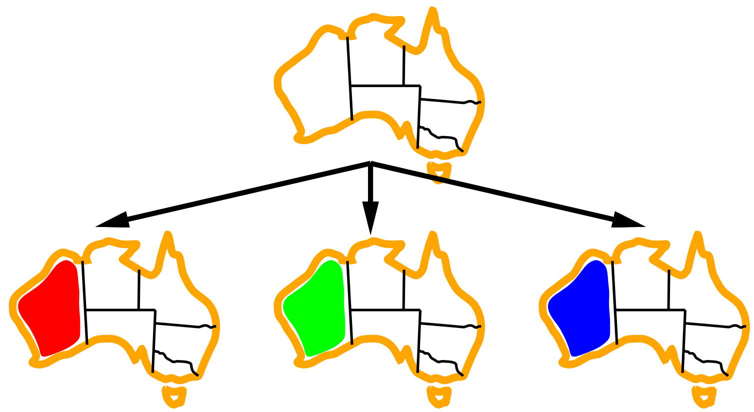
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

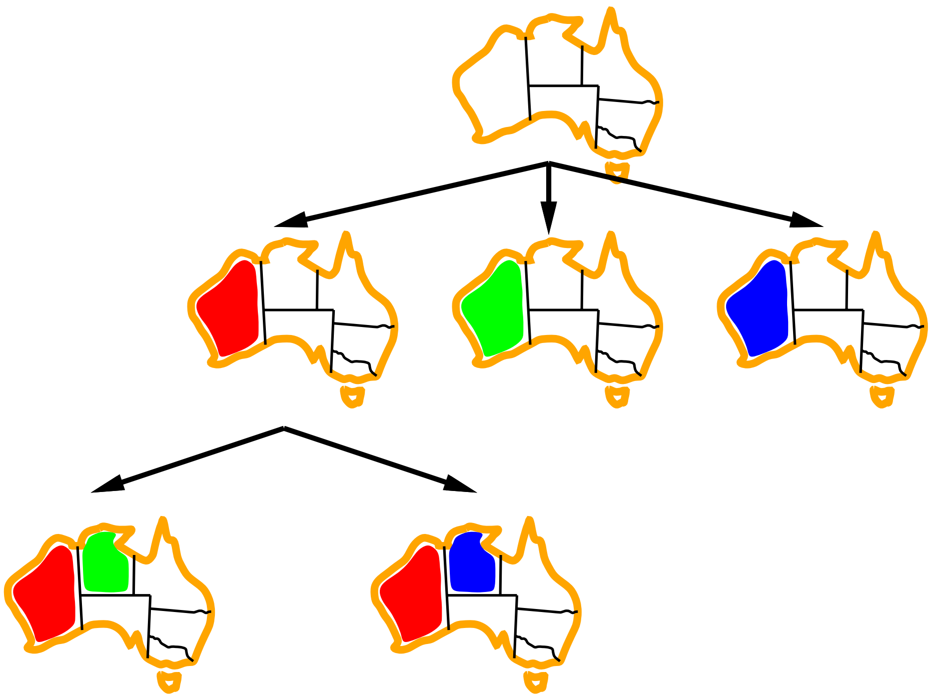
Backtracking example



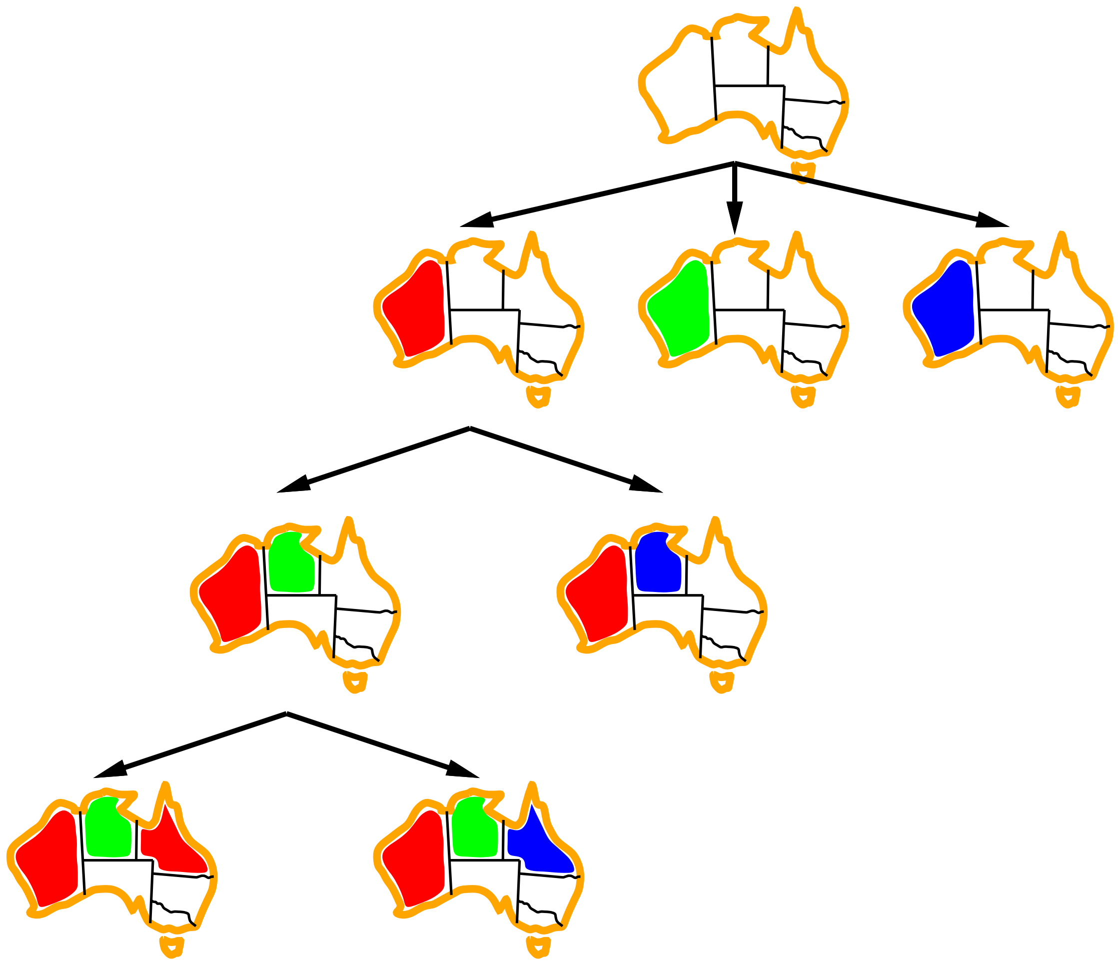
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

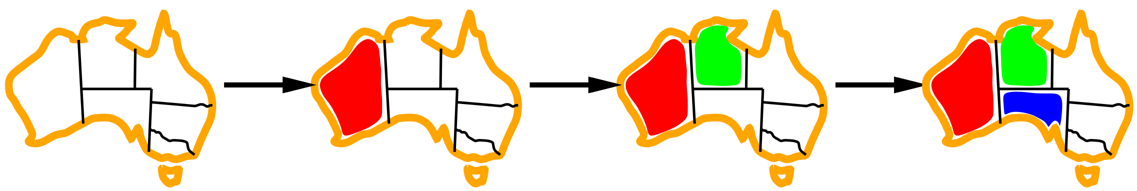
1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we save or reuse partial results of the search?

◇ Can we take advantage of problem structure?

Choosing the variable: MRV

Minimum remaining values (MRV):

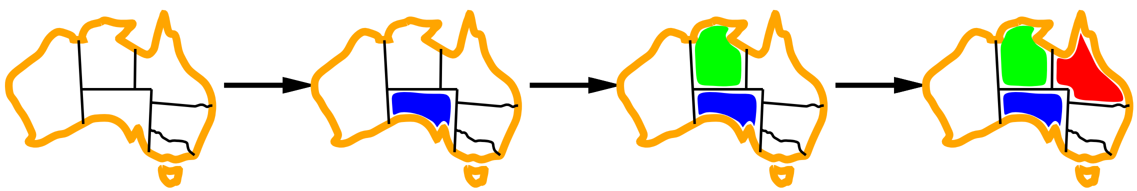
choose the variable with the fewest legal values



Choosing the variable: Degree heuristic

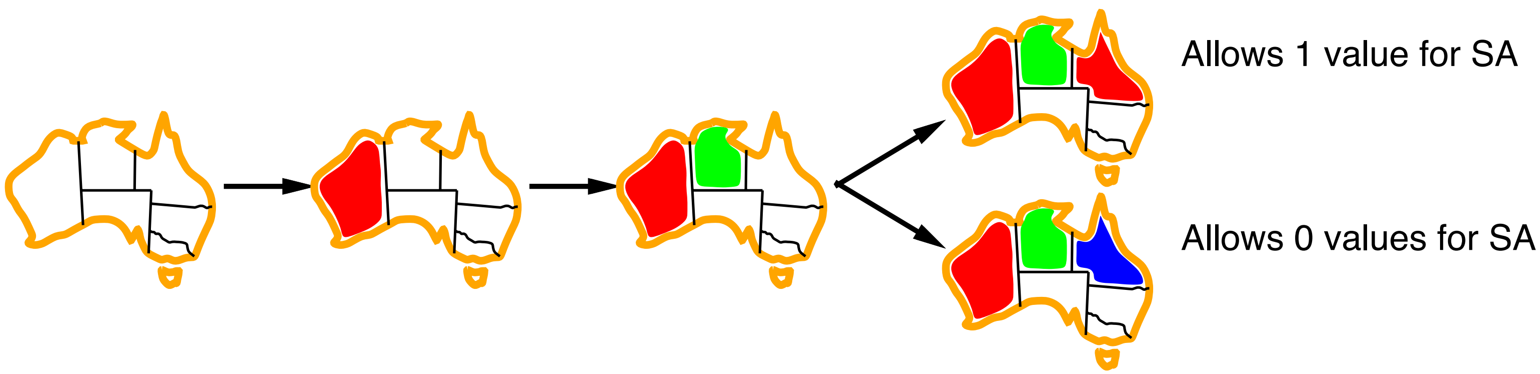
Tie-breaker among MRV variables

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



Choosing the value: 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

Improving backtracking

♢ Intelligent backtracking (vs chronological),

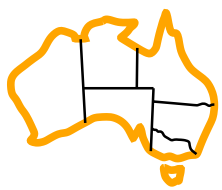
e.g., conflict directed backjumping

Keep track of the sets of conflicting variables

Forward checking

Idea: Keep track of remaining legal values for unassigned variables

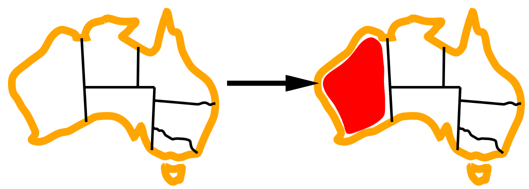
Terminate search when any variable has no legal values

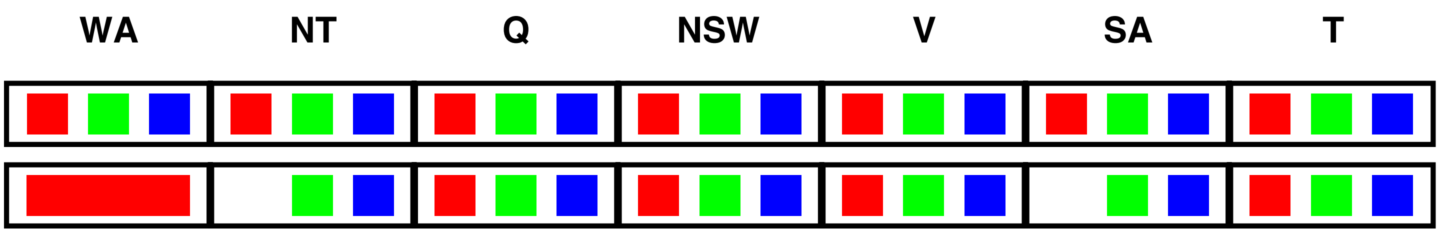


Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values

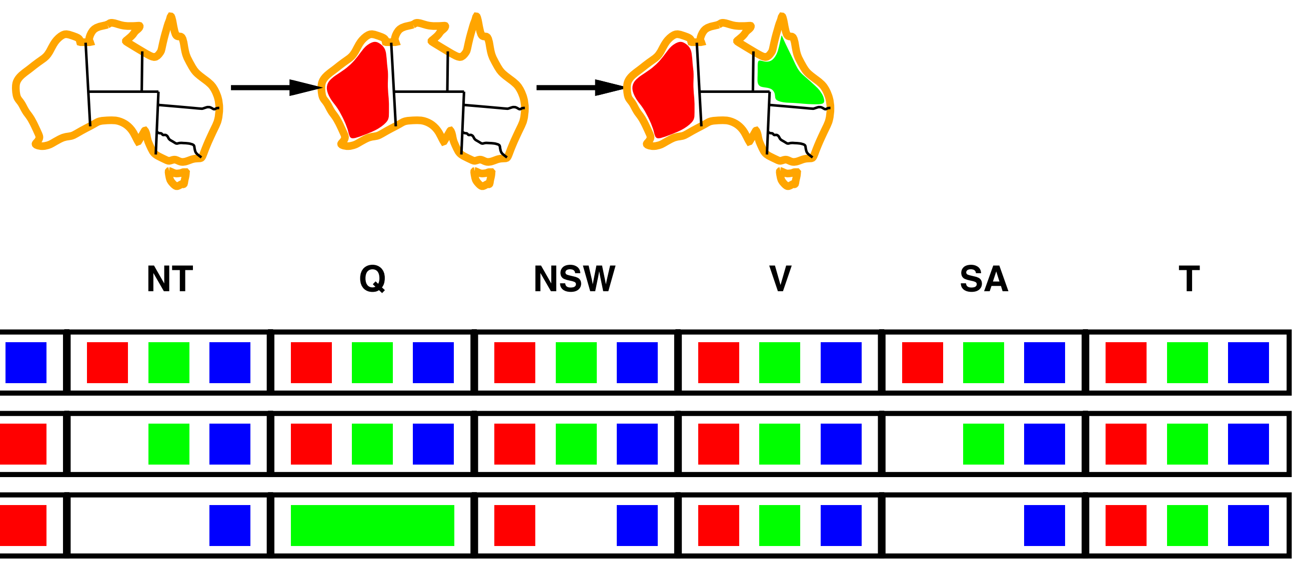




Forward checking

Idea: Keep track of remaining legal values for unassigned variables

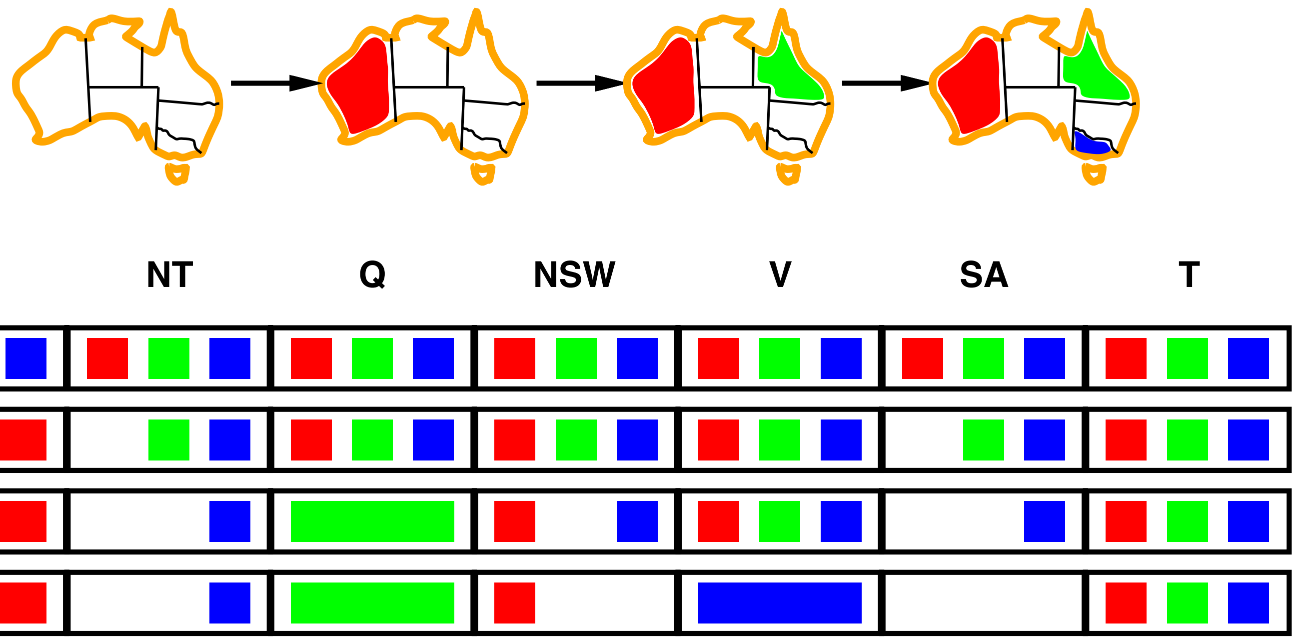
Terminate search when any variable has no legal values



Forward checking

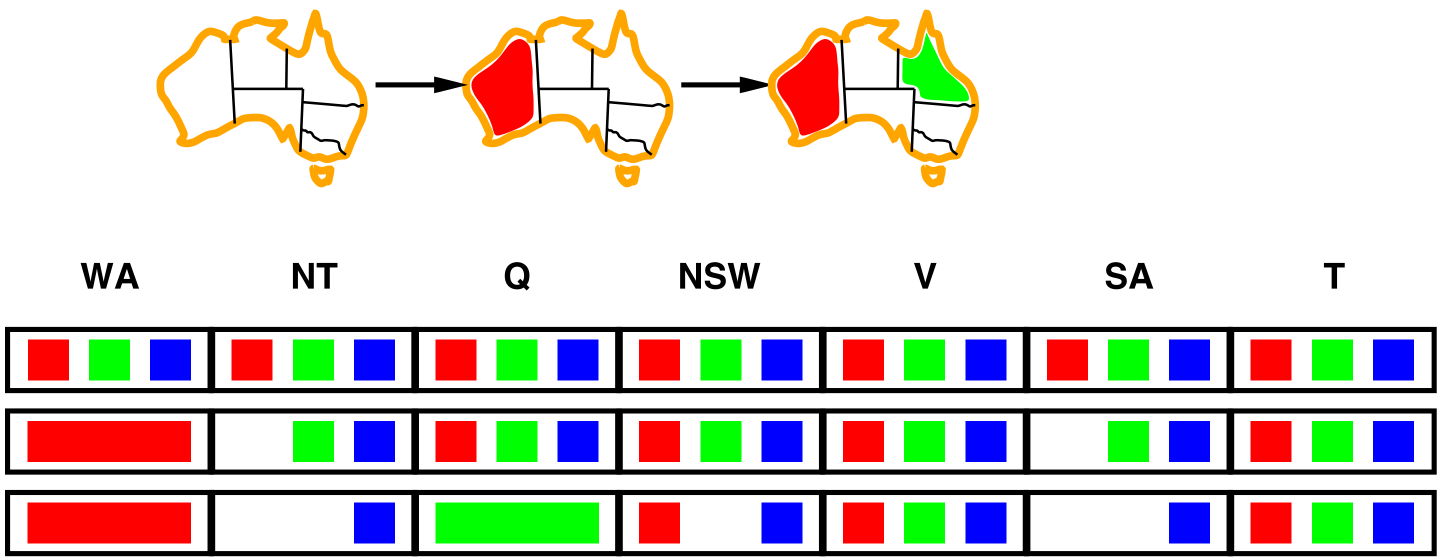
Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



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!

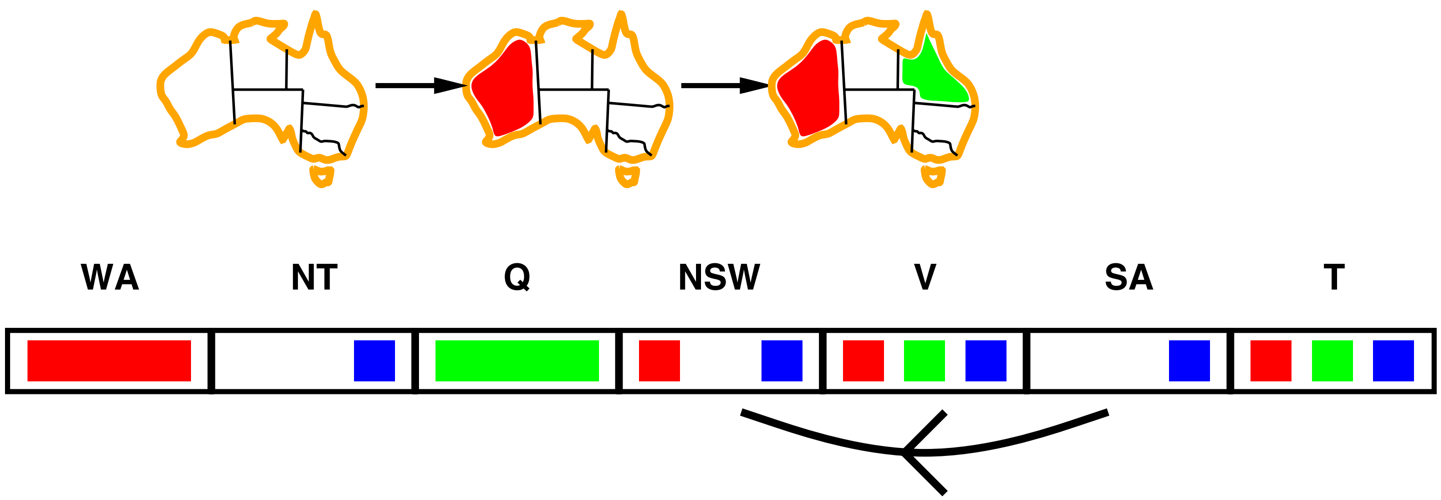
Constraint propagation repeatedly enforces constraints locally

Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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

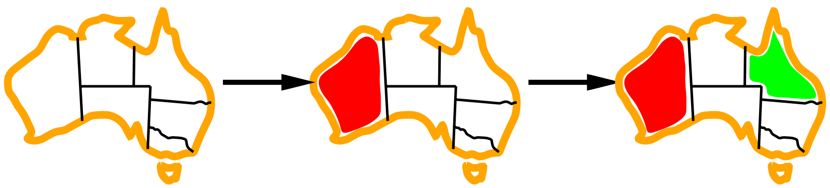


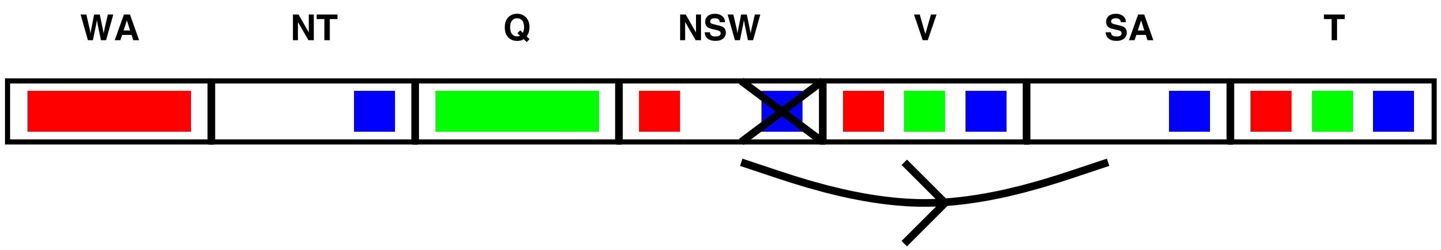
Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \to Y$ is consistent iff

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

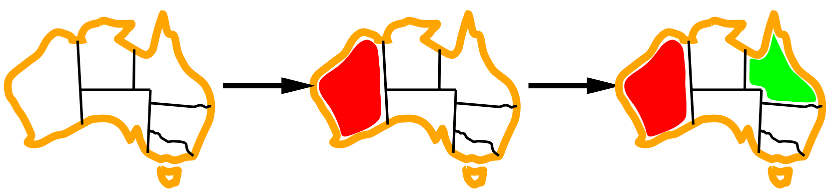


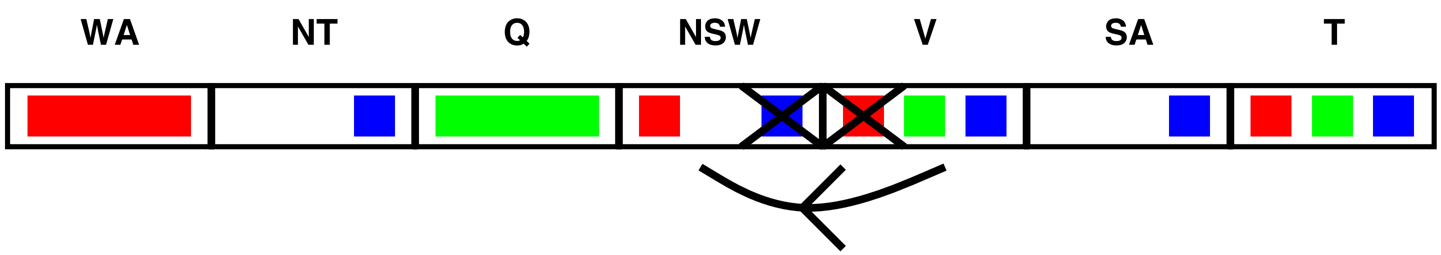


Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff
for **every** value x of X there is **some** allowed y





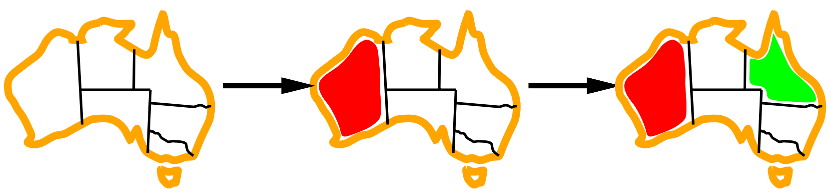
If X loses a value, neighbors of X need to be rechecked

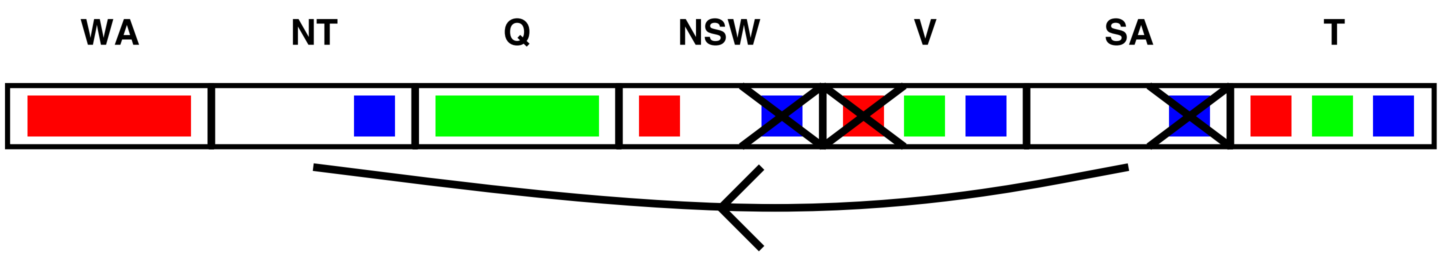
Arc consistency

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

Arc consistency algorithm

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Arc consistency

Arc consistency:

- $O(cd^3)$ (with c number of constraints)
- Can be run as a preprocessor or
- after each assignment: in `Recursive-Backtracking` after adding a new value an inference step is performed
  - forward checking (removing values from the nodes constrained by the assigned var)
  - arc consistency (starting with the arcs that connect the assigned variable with unassigned ones)

Including inference in backtracking search

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

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

Generalizing

- ♢ path-consistency (3-consistency)
- ♢ k-consistency

Constraints that involve k variables.

with $k = n$ it is possible to obtain a complete inference!!

but computational complexity becomes exponential
(as the problem has an exponential complexity–boolean csp)

Local search algorithms for CSPs

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

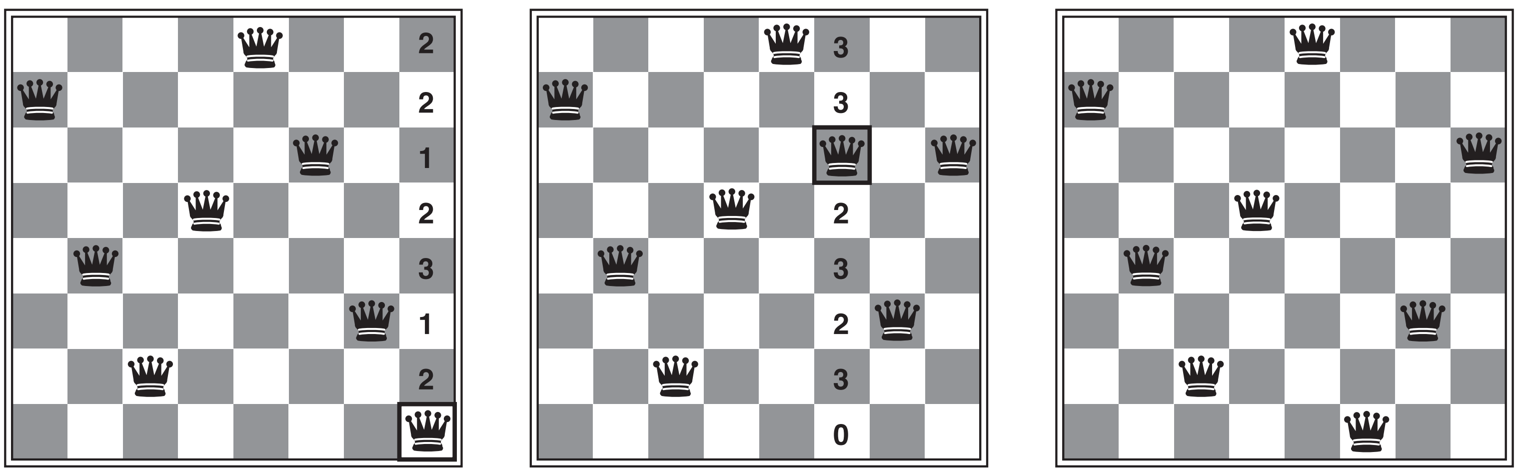
To apply to CSPs:
- allow states with unsatisfied constraints
- operators *reassign* variable values

Variable selection: randomly select any conflicted variable

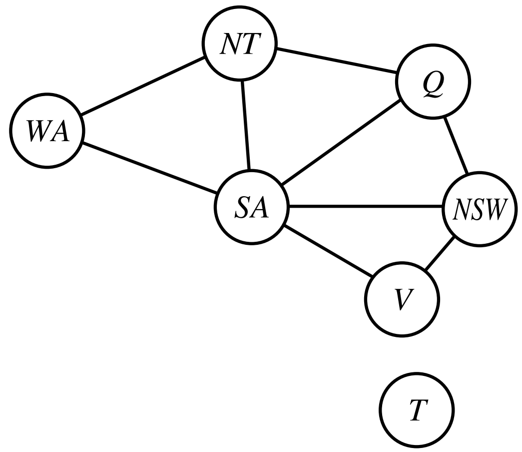
Value selection by *min-conflicts* heuristic:
- choose value that violates the fewest constraints
- i.e., hillclimb with $h(n)$ = total number of violated constraints

Example: 8-Queens

Variables: $Y_1, \ldots, Y_8$
Domains: $\{1, 2, 3, 4, 5, 6, 7, 8\}$
Constraints
$alldiff(Y_1 \ldots, Y_8)$, ...



Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

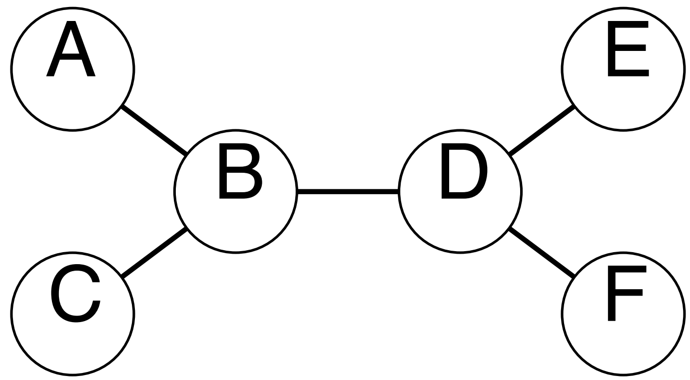
Problem structure contd.

Suppose each subproblem has c variables out of n total

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



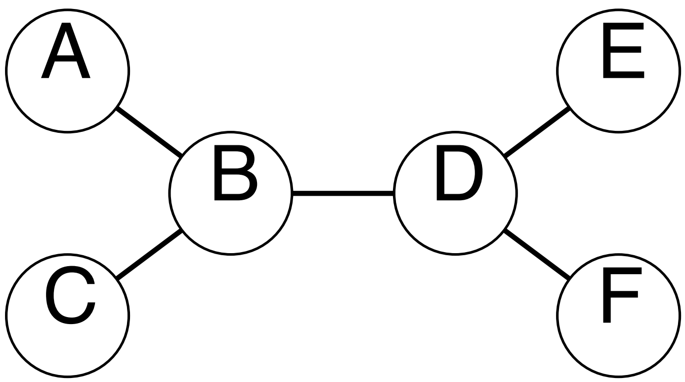
If the constraint graph is a tree, the CSP can be solved in $O(n\,d^2)$ time

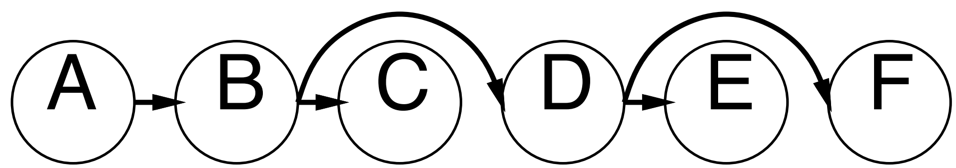
Compare to general CSPs, where worst-case time is $O(d^n)$

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 its children nodes



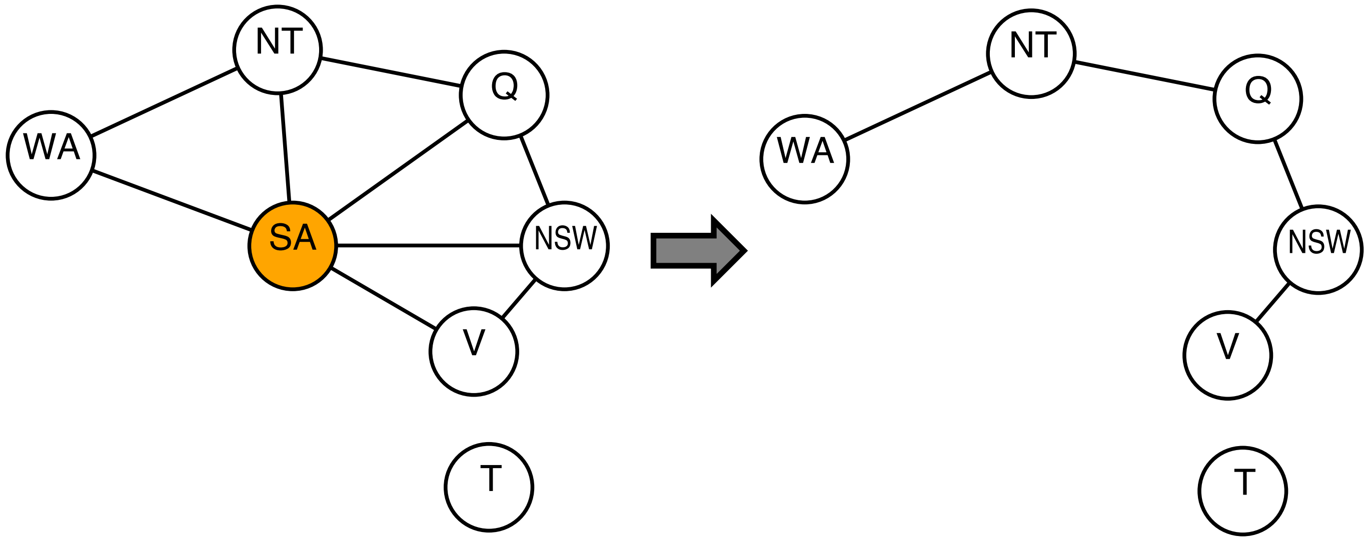


2. For j from n down to 2, apply ArcConsistent$(Parent(X_j), X_j)$

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

Nearly tree-structured CSPs

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

More on problem structure

- Tree decomposition (transformation of graphs into trees)
- Value symmetry (breaking the symmetry to reduce the number of choices)

All problem transformations lead to a polynomial method to solve the problem, but the **exponential** remains in finding the optimal transformation!

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