# Constraint Satisfaction Problems

# Constraint satisfaction problems (CSPs)

- Standard search problem: state is a "black box" – any data structure that supports successor 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 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

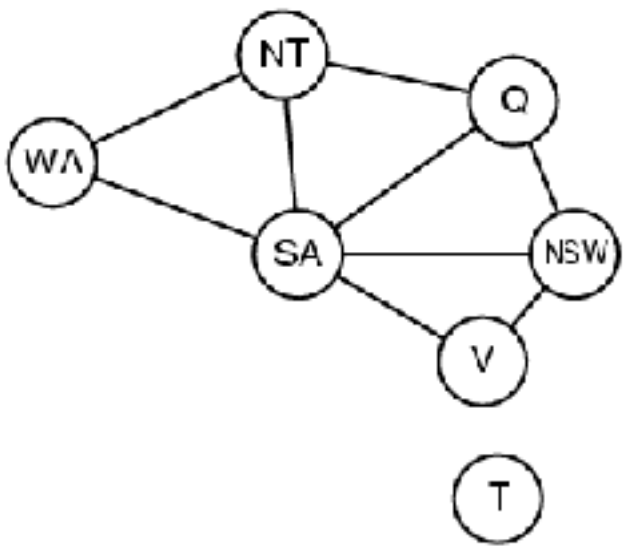
» e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

# 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 $d \rightarrow O(d^n)$ 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 LP

# Varieties 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., cryptarithmetic column constraints

# 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)

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

2. Goal test: the current assignment is complete

1. This is the same for all CSPs

» Every solution appears at depth *n* with *n* variables
→ use depth-first search

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

» b = (n - $\ell$)d at depth $\ell$, hence $n! \cdot d^n$ 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

- Depth-first search for CSPs with single-variable assignments is called backtracking search

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

# Backtracking search

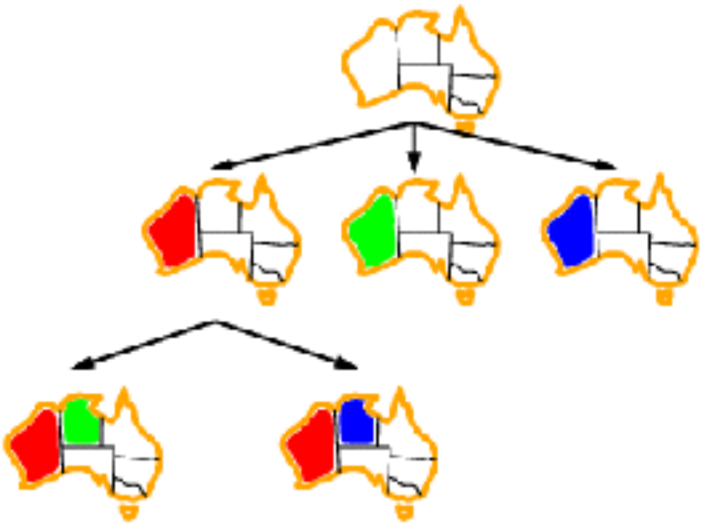
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

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
```

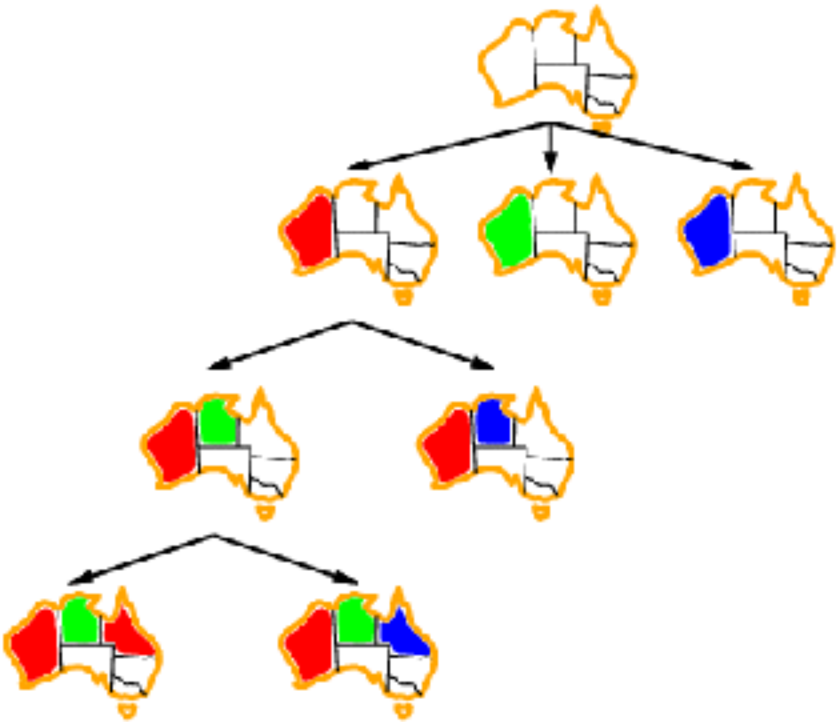
# Backtracking example

# Backtracking example

# Backtracking example



# Backtracking example

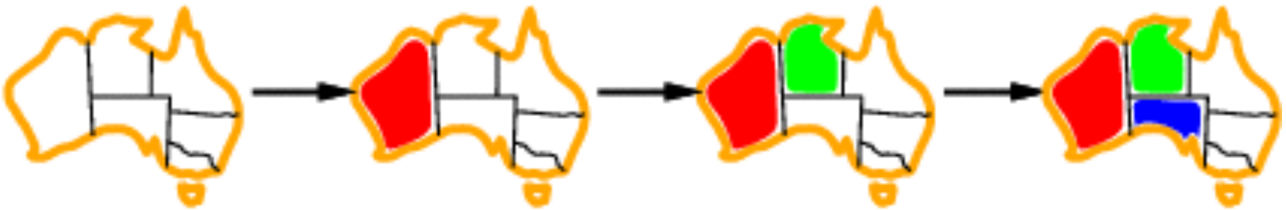


# Improving backtracking efficiency

- 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

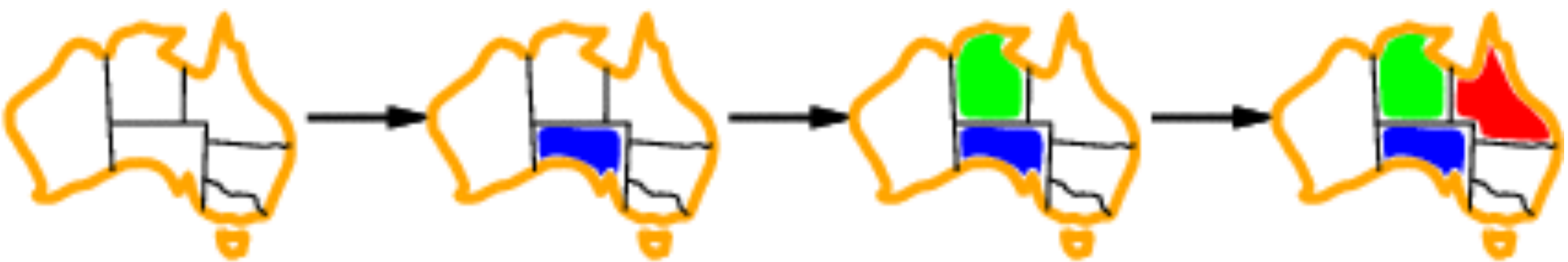
- Most constrained variable:
  choose the variable with the fewest legal values



» minimum remaining values (MRV) heuristic

# Most constraining variable

- A good idea is to use it as a 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 to assign, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

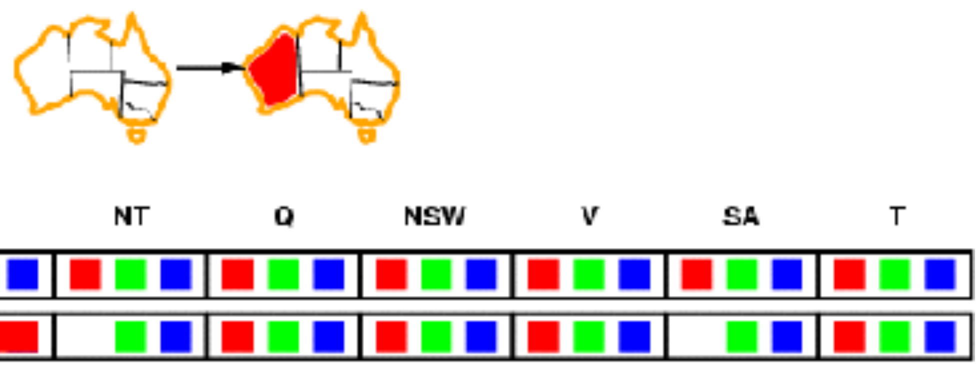
» Combining these heuristics makes 1000 queens feasible

# Forward checking

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

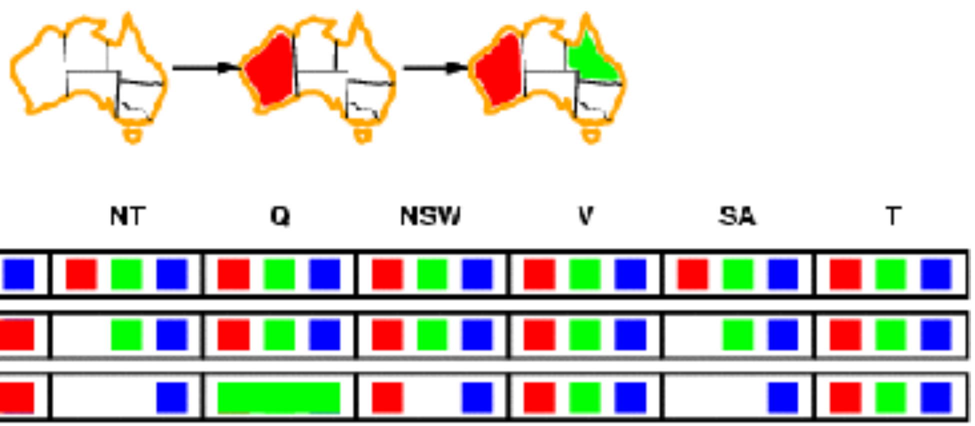
# Forward checking

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



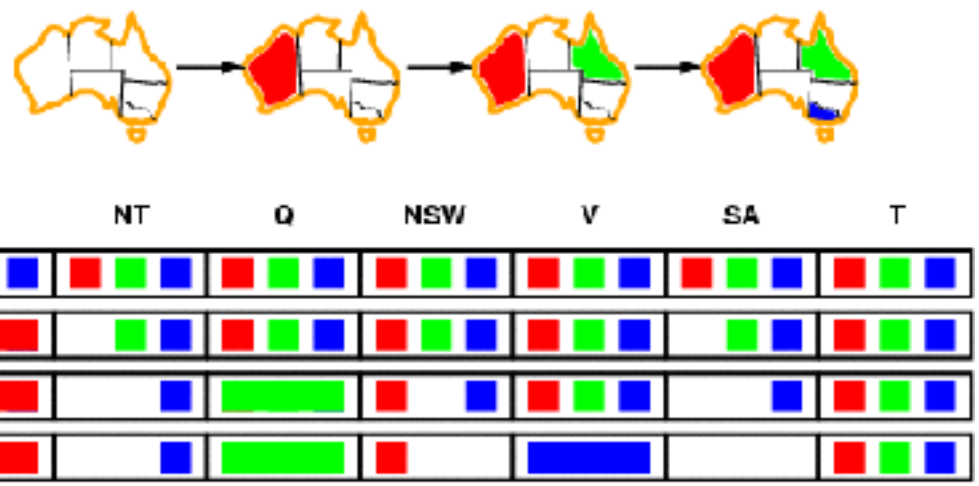
# Forward checking

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



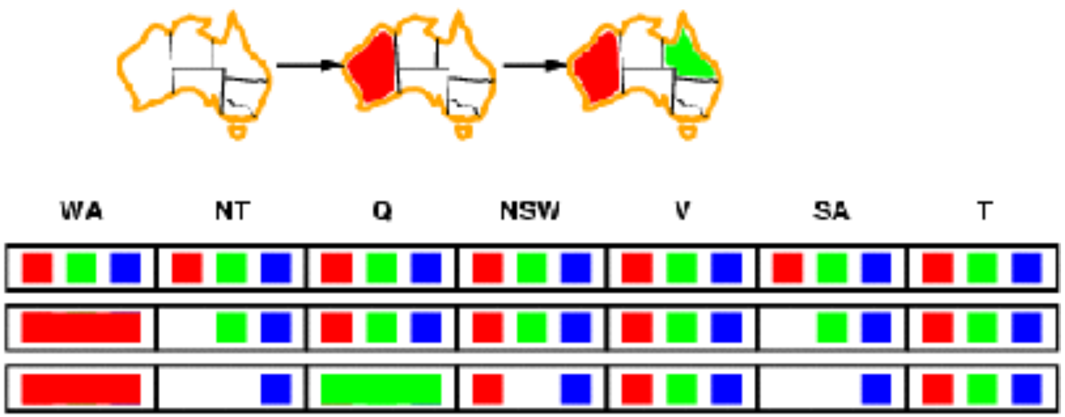
# Forward checking

- 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:

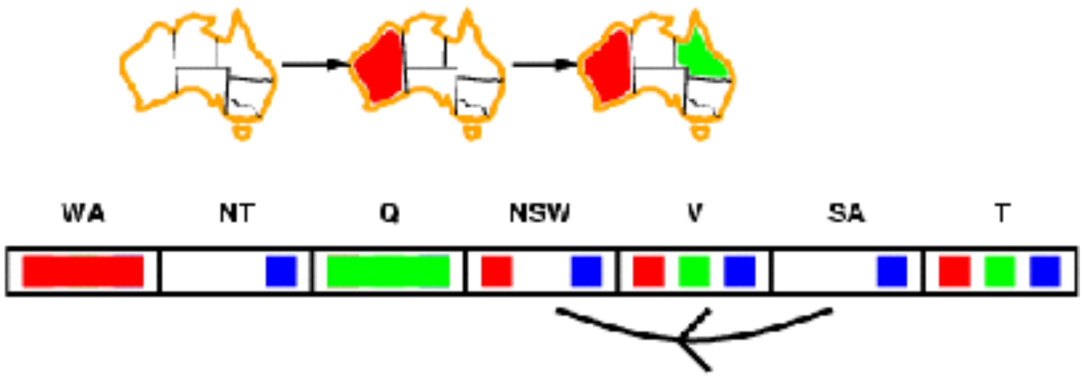



- NT and SA cannot both be blue!

» Constraint propagation algorithms repeatedly enforce constraints locally…

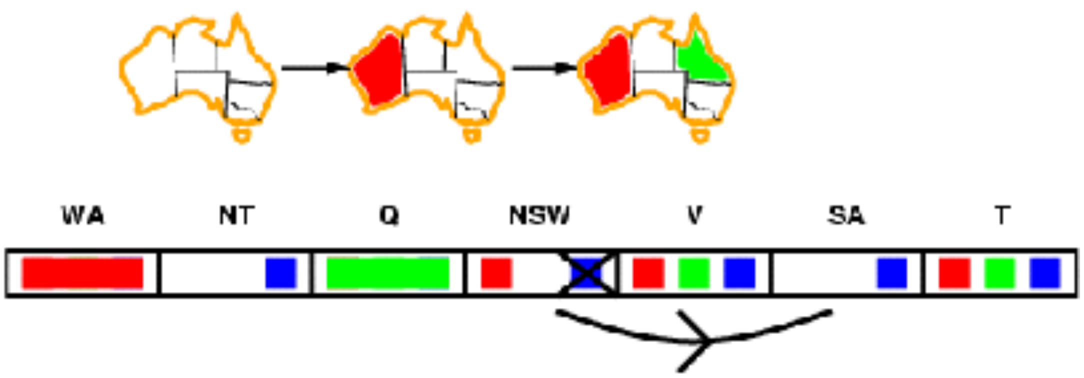
# Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  - for every value $x$ of $X$ there is some allowed $y$



# Arc consistency

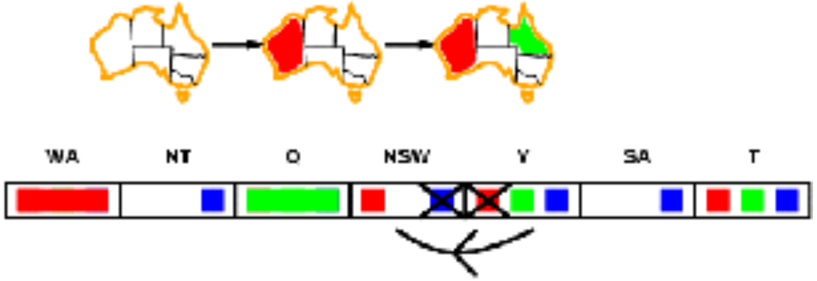
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  - for every value $x$ of $X$ there is some allowed $y$



# 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 $y$

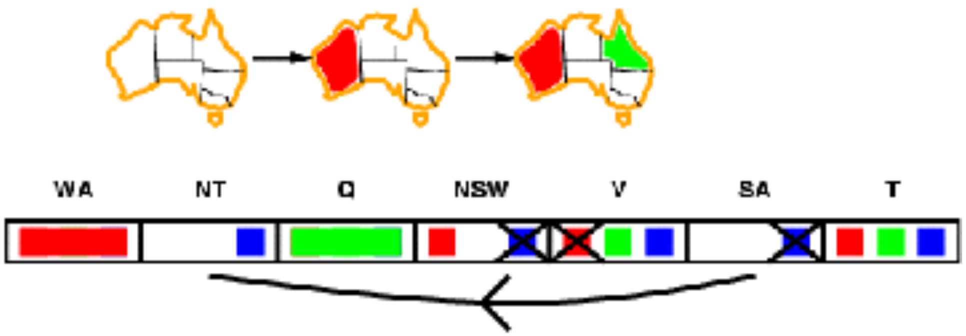



» If $X$ loses a value, neighbors of $X$ need to be rechecked

# 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 {X_1, X_2, ..., X_n}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

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

- Time complexity: O(#constraints $|\text{domain}|^3$)

Checking consistency of an arc is O($|\text{domain}|^2$)

# k-consistency

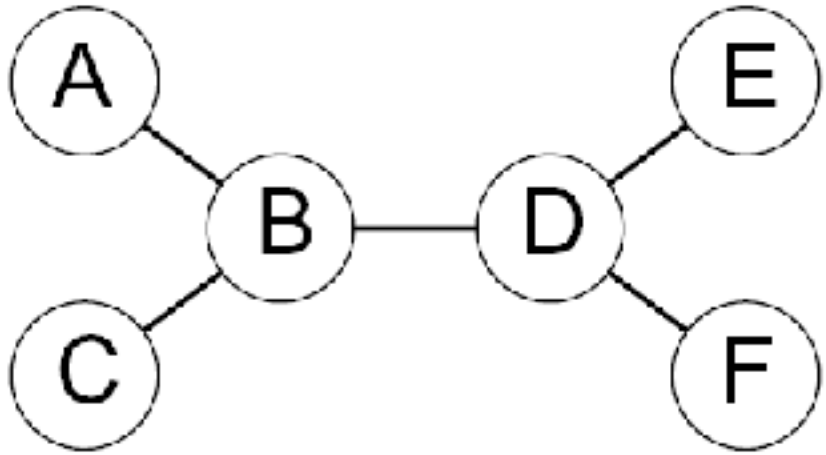
- A CSP is *k-consistent* if, for any set of k-1 variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable
- 1-consistency is node consistency
- 2-consistency is arc consistency
- For binary constraint networks, 3-consistency is the same as *path consistency*
- Getting k-consistency requires time and space exponential in k
- *Strong k-consistency* means k'-consistency for all k' from 1 to k
  - Once strong k-consistency for k=#variables has been obtained, solution can be constructed trivially
- Tradeoff between propagation and branching
- Practitioners usually use 2-consistency and less commonly 3-consistency

# Other techniques for CSPs

- Global constraints
  - E.g., Alldiff
  - E.g., Atmost(10,P1,P2,P3), i.e., sum of the 3 vars ≤ 10
  - Special propagation algorithms
    - Bounds propagation
      - E.g., number of people on two flight D1 = [0, 165] and D2 = [0, 385]
      - Constraint that the total number of people has to be at least 420
      - Propagating bounds constraints yields D1 = [35, 165] and D2 = [255, 385]
- Symmetry breaking

# Structured CSPs

# Tree-structured CSPs



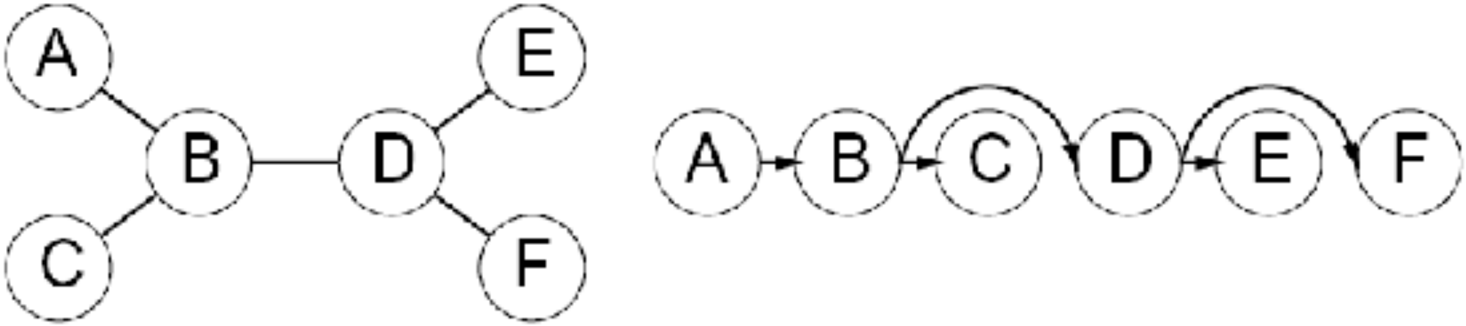

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

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 it in the ordering

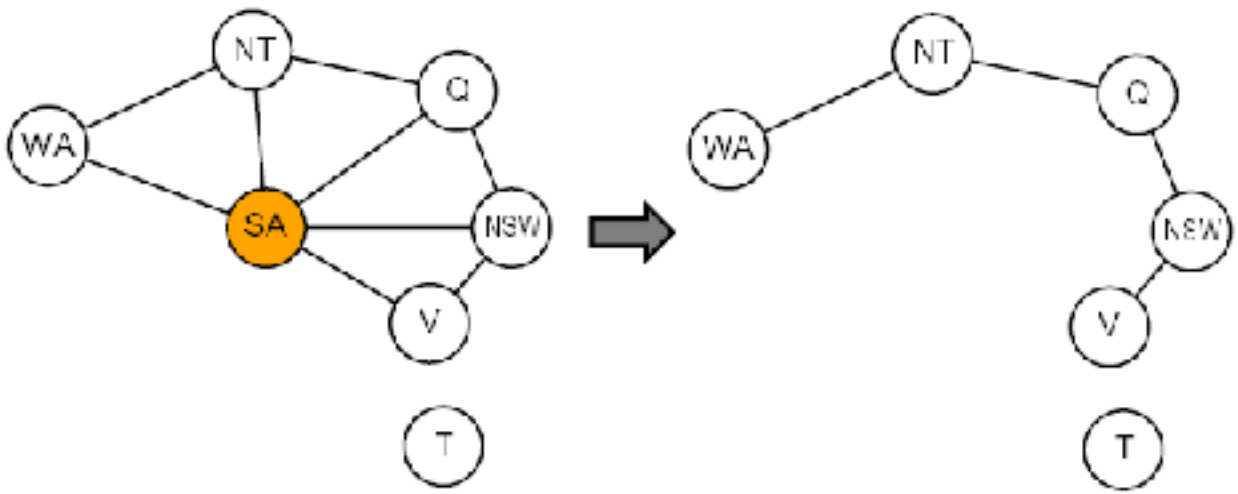



2. For $j$ from $n$ down to $2$, apply RemoveInconsistent$(Parent(X_j), X_j)$

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

# Nearly tree-structured CSPs

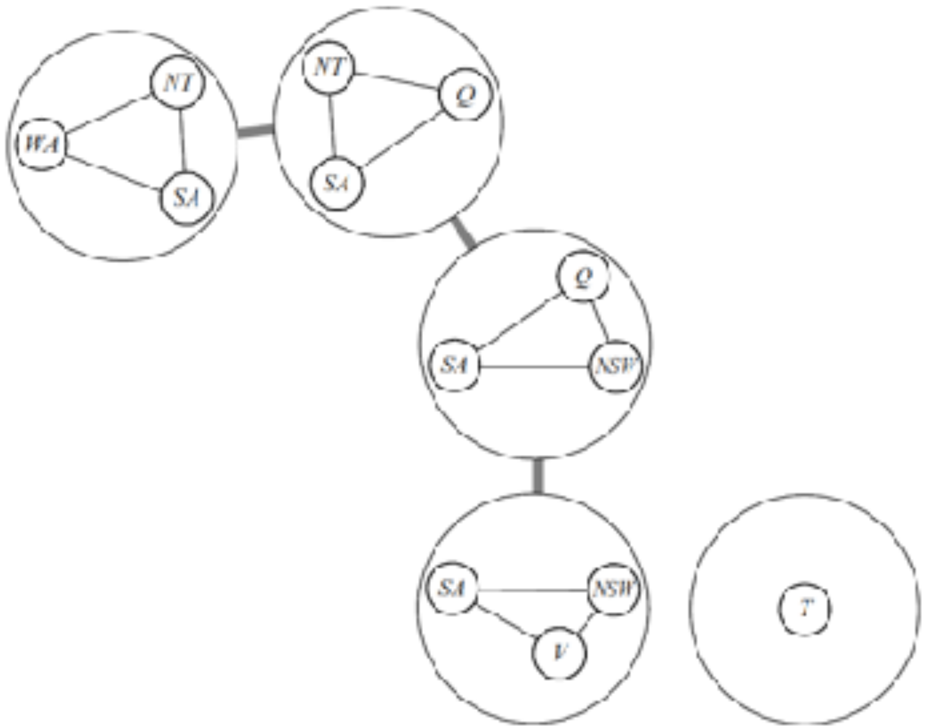
Conditioning: instantiate a variable, prune its neighbors' domains




Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree (Finding the minimum cutset is NP-complete.)

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

# Tree decomposition




- Every variable in original problem must appear in at least one subproblem
- If two variables are connected in the original problem, they must appear together (along with the constraint) in at least one subproblem
- If a variable occurs in two subproblems in the tree, it must appear in every subproblem on the path that connects the two

- Algorithm: solve for all solutions of each subproblem. Then, use the tree-structured algorithm, treating the subproblem solutions as variables for those subproblems.
- $O(nd^{w+1})$ where w is the *treewidth* (= one less than size of largest subproblem)
- Finding a tree decomposition of smallest treewidth is NP-complete, but good heuristic methods exists

# 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

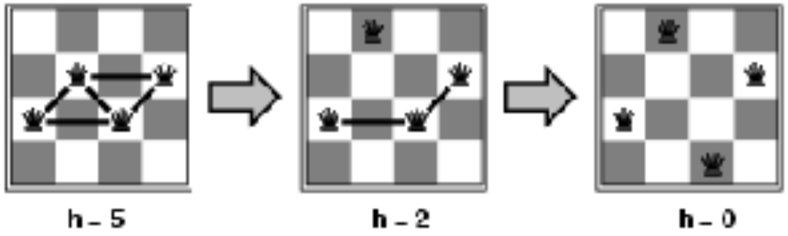
- Variable selection: randomly select any conflicted variable

» 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

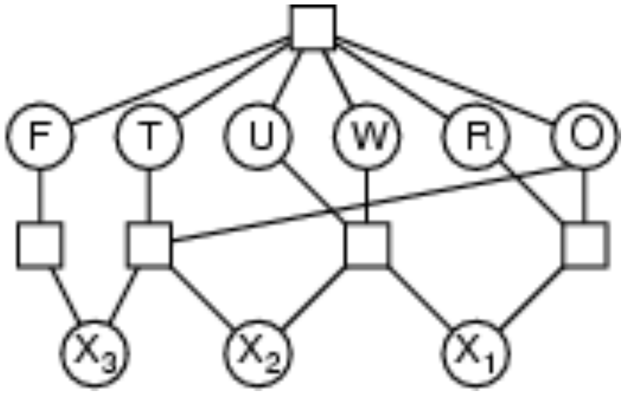
- States: 4 queens in 4 columns ($4^4$ = 256 states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: *h(n)* = number of attacks




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

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

# 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