Constraint Satisfaction Problems

Juhan Ernits
Institute of Computer Science
Tallinn University of Technology
Juhan.ernits@ttu.ee
2015

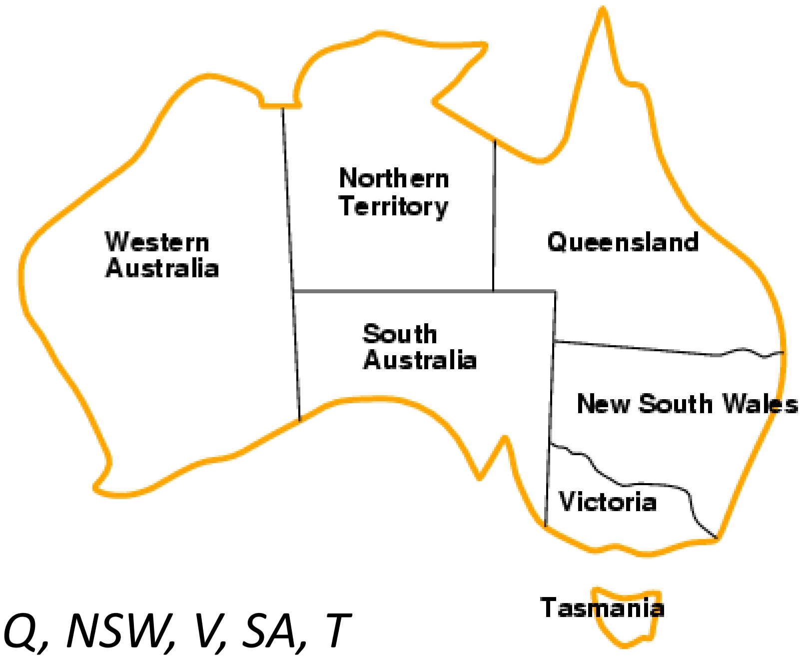
Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs
- Tree search and decomposition of CSPs

Constraint satisfaction problems (CSPs)

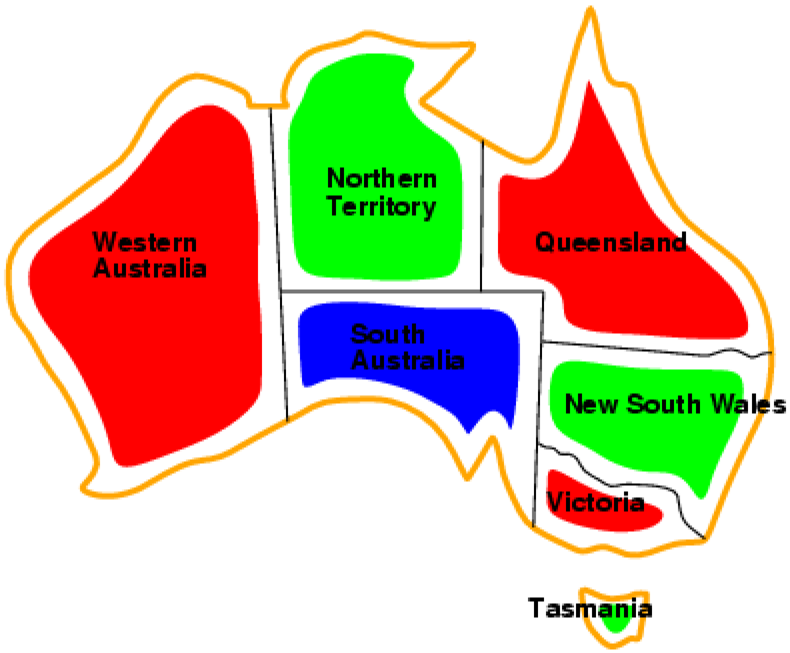
- Standard search problem:
  - state is a "black box“ – any data structure that supports successor function, heuristic function, and goal test
- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for 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)}

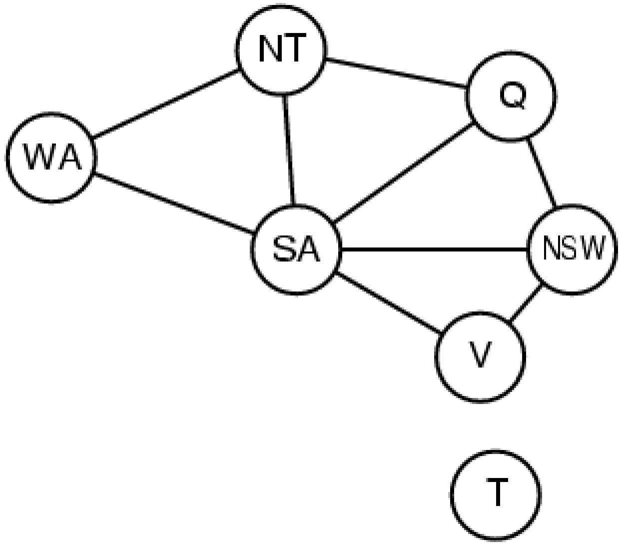
Example: Map-Coloring



- 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 polynomial time by linear programming

Example: Job-shop scheduling

- E.g. schedule day's worth of jobs in a factory

$$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$$

Precedence constraints: $T_1 + d \leq T_2$

$$Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF}$$
$$Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}$$

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF}$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF}$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB}$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}$$

$(Axle_F + 10 \leq Axle_B)$ **or** $(Axle_B + 10 \leq Axle_F)$ (Disjunctive constraints)

$X + d_X \leq Inspect$ (If inspection takes 3 minutes, can all be done in 30 minutes?)

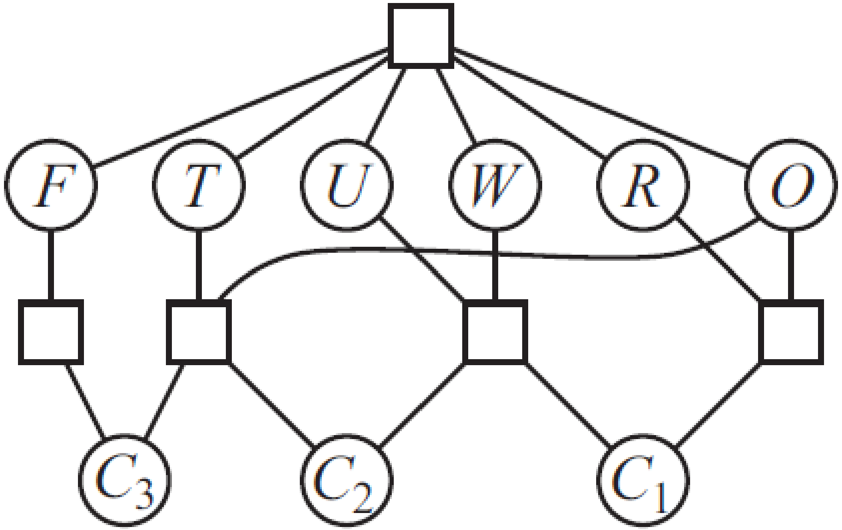
$D_i = \{1, 2, 3, \ldots, 27\}$ (Finite domain)

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Global constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints
  - e.g. allDiff constraints (all values different)

Example: Cryptarithmetic

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



- Variables: *F T U W R O* C_1 C_2 C_3
- Domains: {*0,1,2,3,4,5,6,7,8,9*}
- Constraints: *Alldiff (F,T,U,W,R,O)*
  - $O + O = R + 10 \cdot C_1$
  - $C_1 + W + W = U + 10 \cdot C_2$
  - $C_2 + T + T = O + 10 \cdot C_3$
  - $C_3 = F$

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
  - Involves preference constraints in addition to absolute ones
- Transportation scheduling
- Factory scheduling

- Notice that many real-world problems involve real-valued variables

Solving CSPs

- There are two main approaches for solving CSPs:
  - Inference
  - Search
- Sometimes CSPs can be solved by inference alone.
- In other cases, solving CSP-s involves a combination of inference and search.

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. $b = (n - \ell)d$ at depth ℓ, hence $n! \cdot d^n$ leaves
5. Can be fixed by the observation that variables in CSPs are commutative

Backtracking search

- Variable assignments are commutative}, i.e.,

[WA = red then NT = green] same as [NT = green then WA = red]

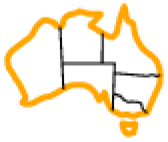
- Only need to consider assignments to a single variable at each node
  → b = d and there are d^n leaves

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

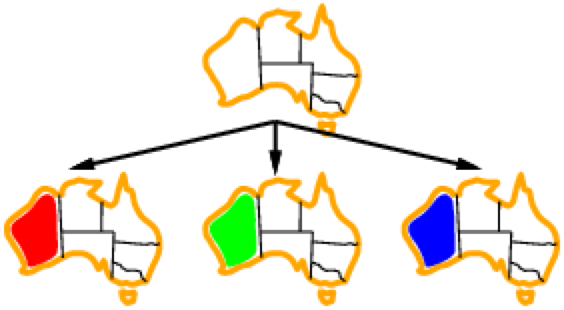
- Backtracking search is the basic uninformed algorithm for CSPs

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

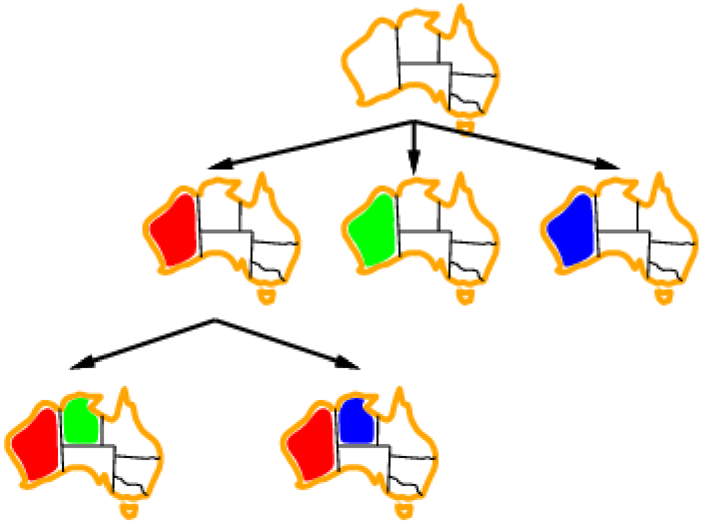
Backtracking example



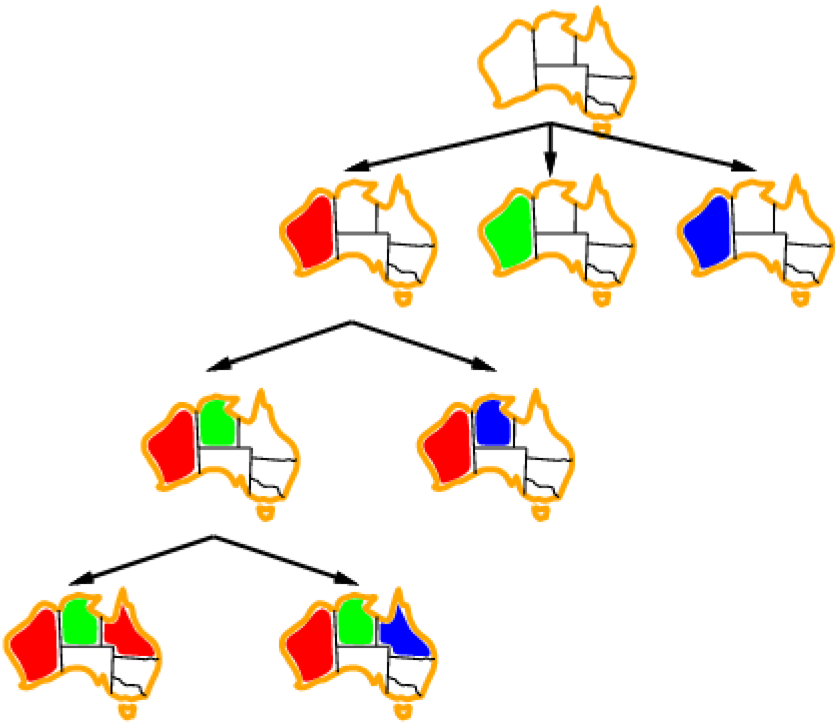
Backtracking example



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?

Backtracking search

```
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
```

Backtracking search

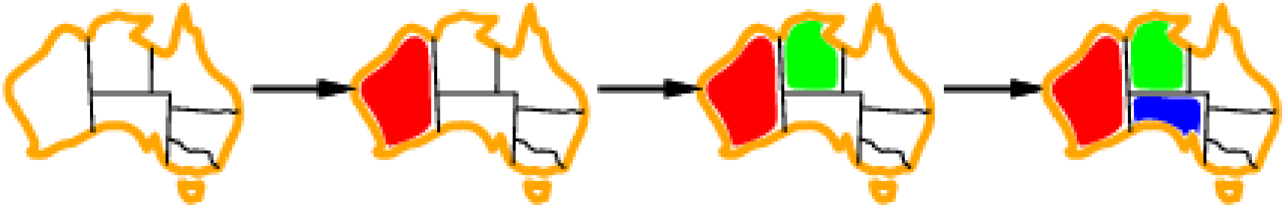
```
def backtracking_search(csp,
                        select_unassigned_variable = first_unassigned_variable,
                        order_domain_values = unordered_domain_values,
                        inference = no_inference):

    def backtrack(assignment):
        if len(assignment) == len(csp.vars):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if 0 == csp.nconflicts(var, value, assignment):
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result
```

Most constrained variable

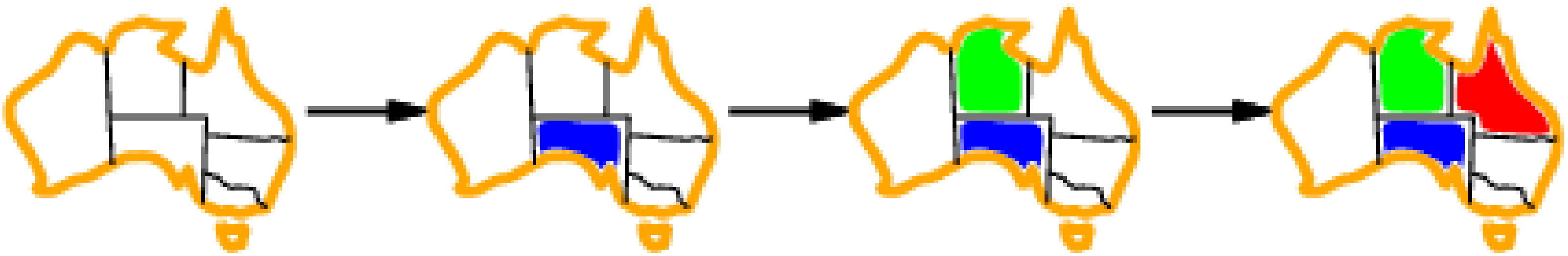
- Most constrained variable:
  choose the variable with the fewest legal values



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

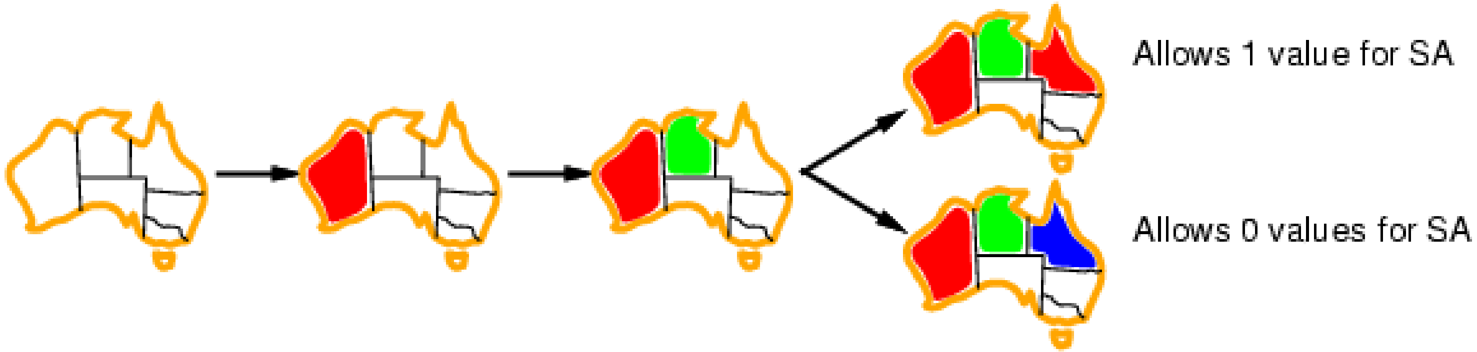
Most constraining variable

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



Least constraining value

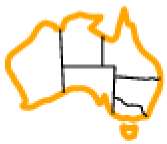
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

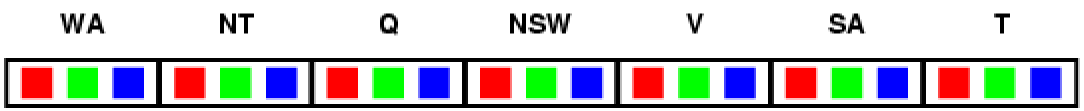


- Combining these heuristics makes 1000 queens feasible

Forward checking

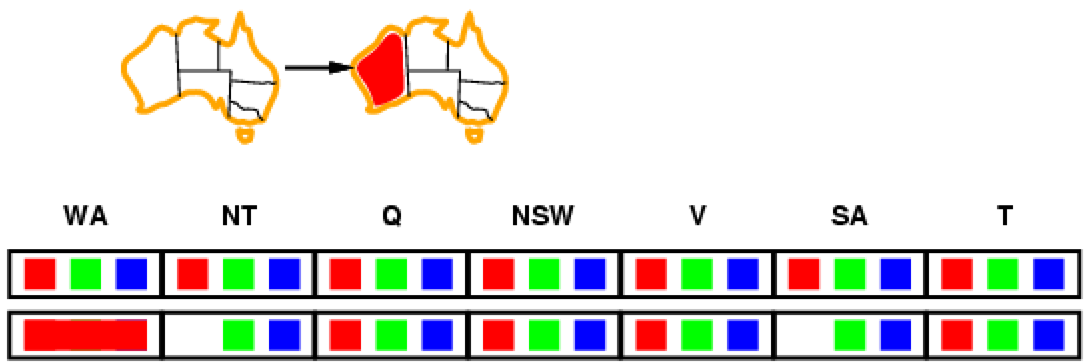
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values





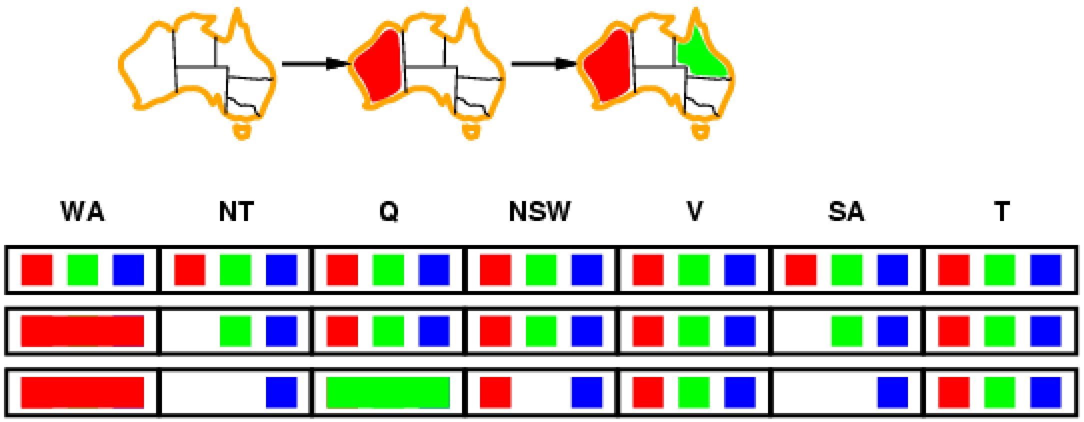
Forward checking

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



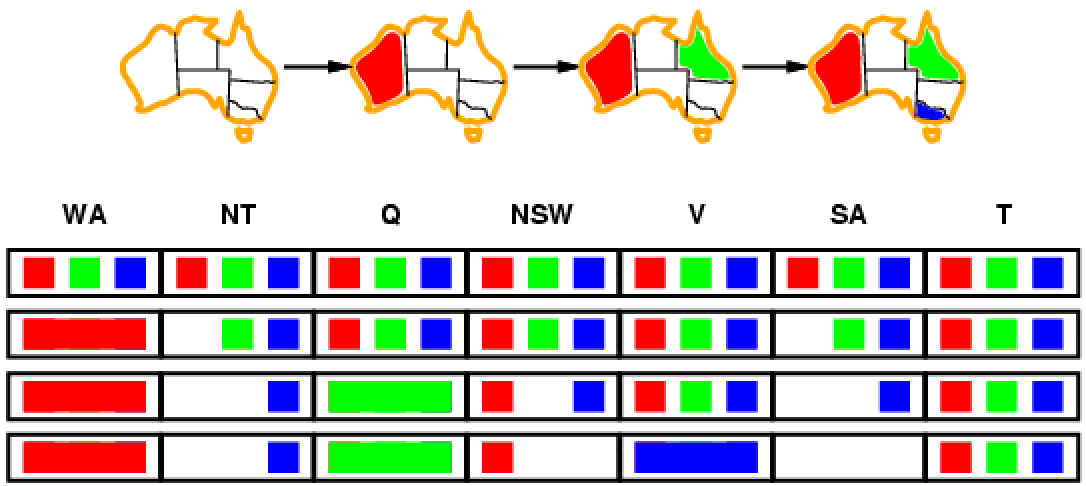
Forward checking

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



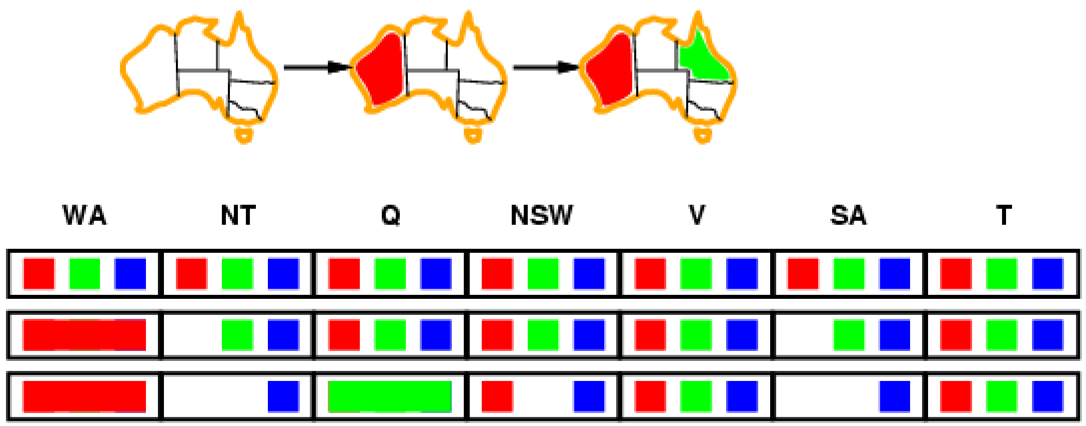
Forward checking

- 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

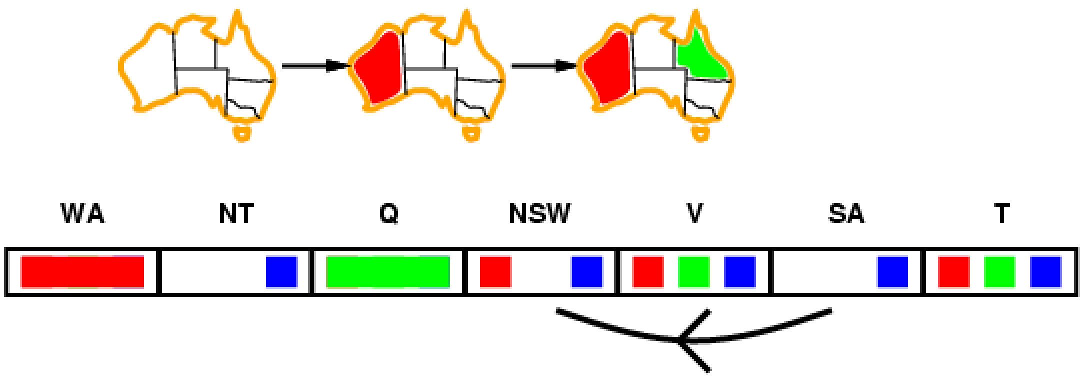
Inference

- Forward checking
- Constraint propagation
  – Node consistency
    - All unary constraints of a variable satisfied
  – Arc consistency
    - Every value in the domain of a variable satisfies the variable's binary constraints
  – Path consistency
  – K-consistency

Arc consistency

- Arc consistency makes each binary constraint (arc) consistent
- *X* → *Y* is consistent iff

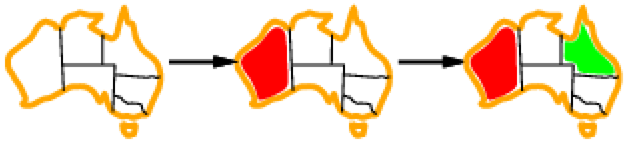
  for every value *x* of *X* there is some allowed *y*

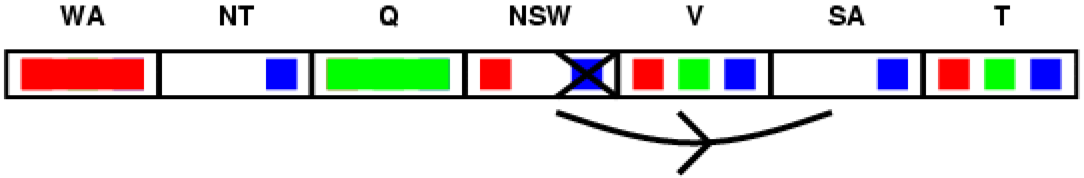


Arc consistency

- Arc consistency makes each binary constraint (arc) consistent
- *X* → *Y* is consistent iff

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

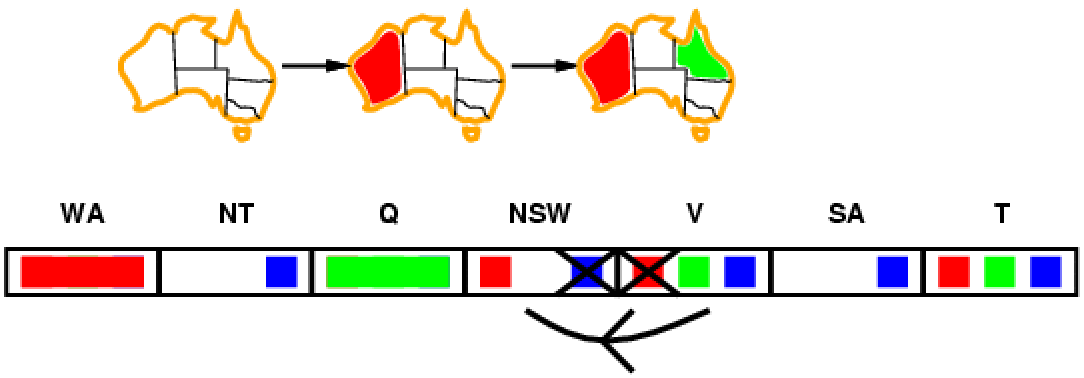




Arc consistency

- Arc consistency makes each binary constraint (arc) consistent
- *X* → *Y* is consistent iff

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

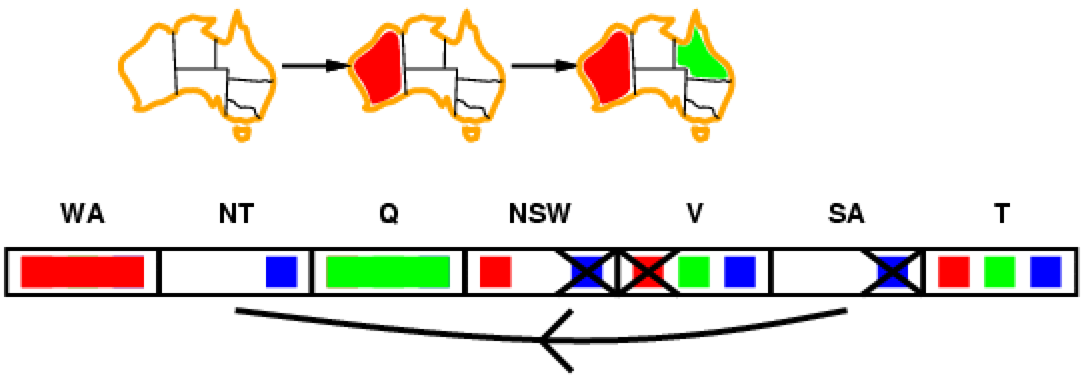


- If *X* loses a value, neighbors of *X* need to be rechecked

Arc consistency

- Arc consistency makes each binary constraint (arc) consistent
- *X* → *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** 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**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*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 $\setminus \{X_j\}$ **do**
        add (X_k, X_i) to *queue*

---

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

- Time complexity: $O(n^2d^3)$

Arc consistency algorithm AC-3

```
def AC3(csp, queue=None, removals=None):
    """[Fig. 6.3]"""
    if queue is None:
        queue = [(Xi, Xk) for Xi in csp.vars for Xk in csp.neighbors[Xi]]
    csp.support_pruning()
    while queue:
        (Xi, Xj) = queue.pop()
        if revise(csp, Xi, Xj, removals):
            if not csp.curr_domains[Xi]:
                return False
            for Xk in csp.neighbors[Xi]:
                if Xk != Xi:
                    queue.append((Xk, Xi))
    return True

def revise(csp, Xi, Xj, removals):
    "Return true if we remove a value."
    revised = False
    for x in csp.curr_domains[Xi][:]:
        # If Xi=x conflicts with Xj=y for every possible y, eliminate Xi=x
        if every(lambda y: not csp.constraints(Xi, x, Xj, y),
                   csp.curr_domains[Xj]):
            csp.prune(Xi, x, removals)
            revised = True
    return revised
```

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		

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		

	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

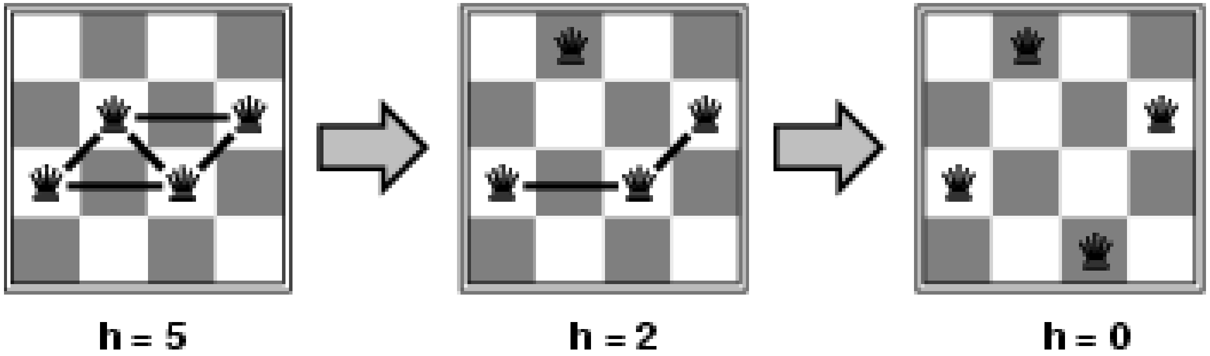
Arc consistency is able to solve some Sudoku puzzles and no classical search is needed!

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

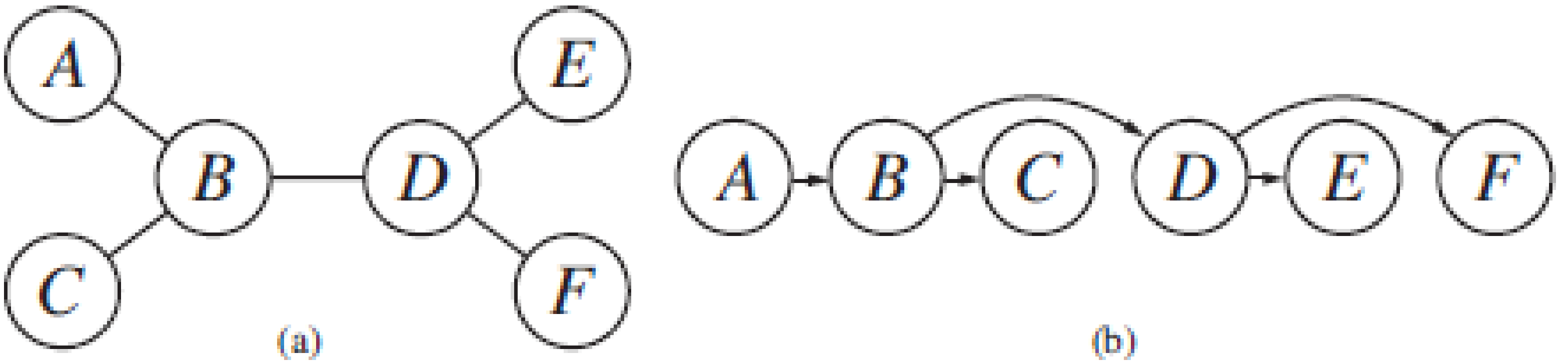
- States: 4 queens in 4 columns (4^4 = 256 states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: *h(n)* = number of attacks



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

Utilising the structure of problems

Topological sorting of nodes



Tree search

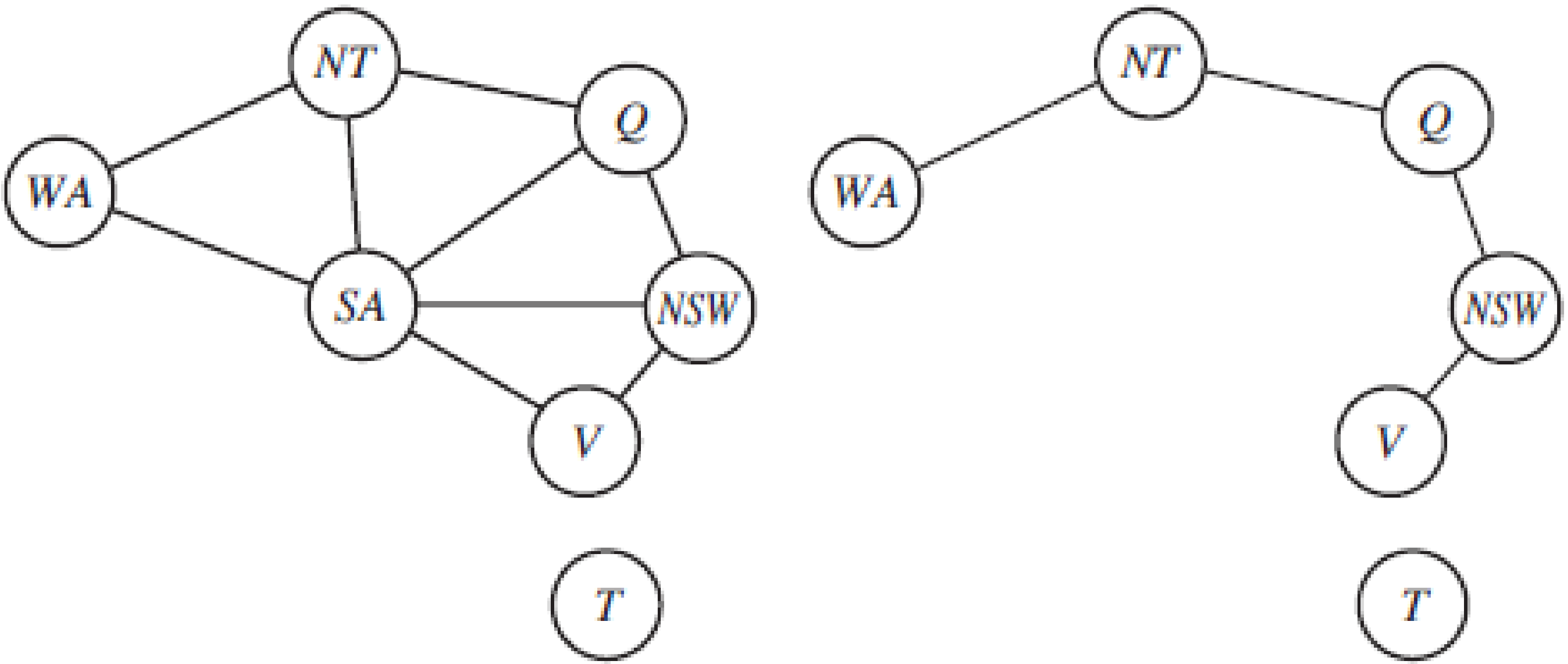
```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
   inputs: csp, a CSP with components X, D, C

   n ← number of variables in X
   assignment ← an empty assignment
   root ← any variable in X
   X ← TOPOLOGICALSORT(X, root)
   for j = n down to 2 do
      MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
      if it cannot be made consistent then return failure
   for i = 1 to n do
      assignment[X_i] ← any consistent value from D_i
      if there is no consistent value then return failure
   return assignment
```

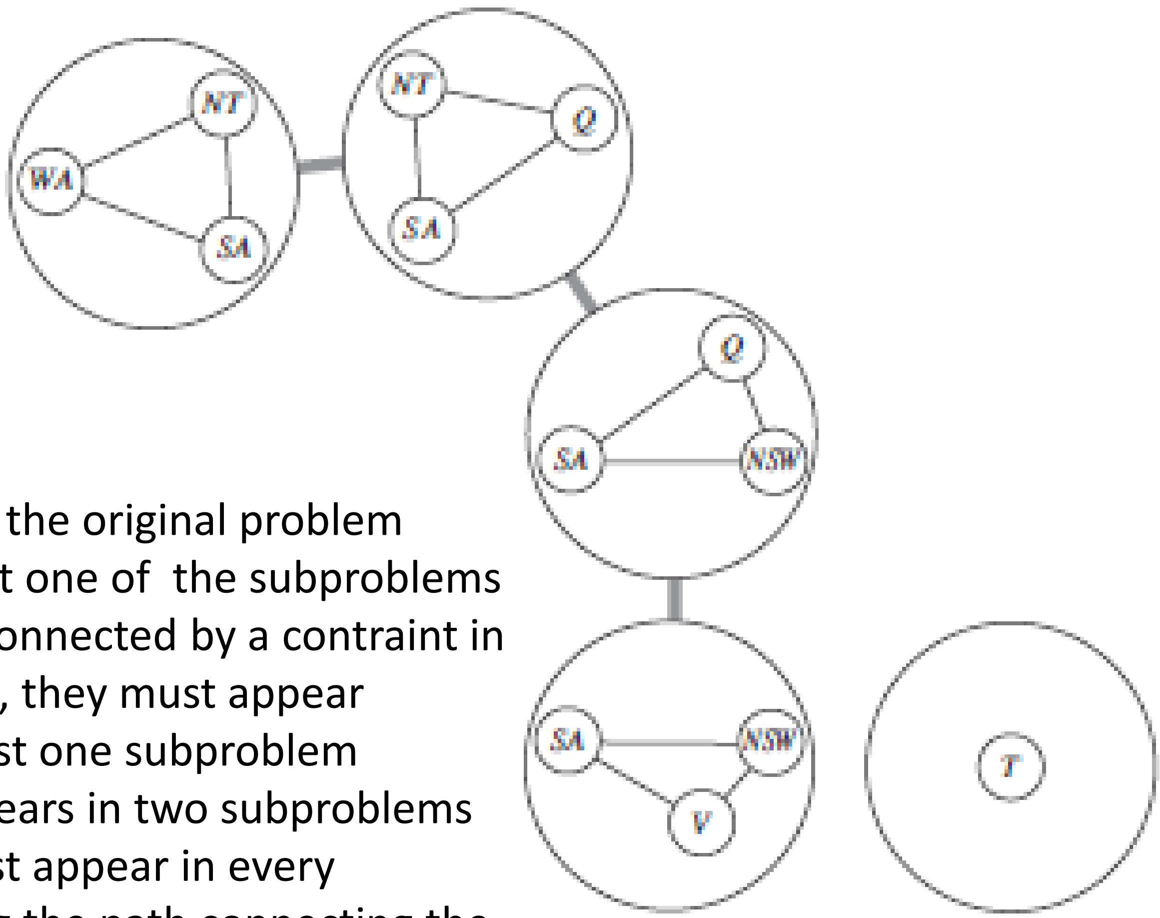
Tree search

```
def tree_csp_solver(csp):
    "[Fig. 6.11]"
    n = len(csp.vars)
    assignment = {}
    root = csp.vars[0]
    X, parent = topological_sort(csp.vars, root)
    for Xj in reversed(X):
        if not make_arc_consistent(parent[Xj], Xj, csp):
            return None
    for Xi in X:
        if not csp.curr_domains[Xi]:
            return None
        assignment[Xi] = csp.curr_domains[Xi][0]
    return assignment
```

Tree search on general graphs



Tree decomposition



•Every variable in the original problem appears in at least one of the subproblems
• If two vars are connected by a contraint in the orig. Problem, they must appear together in at least one subproblem
•IF a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting the subproblems.

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
- If a problem is too hard to solve, break it into pieces and try solving it piece by piece