Constraint Satisfaction Problems

4

4 Constraint Satisfaction Problems

4.1 Constraint satisfaction problems

4.2 Constraint propagation

4.3 Backtracking search

4.4 Local search

4.5 Structure and decomposition^{+}

Constraint satisfaction problems

Standard search problem

state is a "black box" — any data structure (atomic representation)

that supports goal test, evaluation, successor operators

CSP: a simple formal representation language

– a factored representation for each state

– a vector of variables and their (attribute) values

Allows useful **general-purpose** algorithms with more power than standard (problem-specific) search algorithms

CSP

CSP=(X, D, C)

- $X = \{X_1, \cdots, X_n\}$: a set of variables
- $D = \{D_1, \cdots, D_n\}$: a set of domains
- $C = \{C_1, \cdots, C_n\}$: a set of constraints

Each domain $D_i \in D$ consists of a set of allowable values $\{v_1, \cdots, v_k\}$ for variable $X_i \in X$

Each constrain $C_i = (scope, rel)$

- $scope$: a tuple of variables that participate in the constraint
- rel: a relation that defines the allowable values for $scope$

C specifies allowable combinations of values for subsets of X

CSP

Each state in a CSP is defined by an assignment of values to some (partial assignment) or all (complete assignment) variables

$$\{X_i = v_i, X_j = v_j, \cdots\}$$

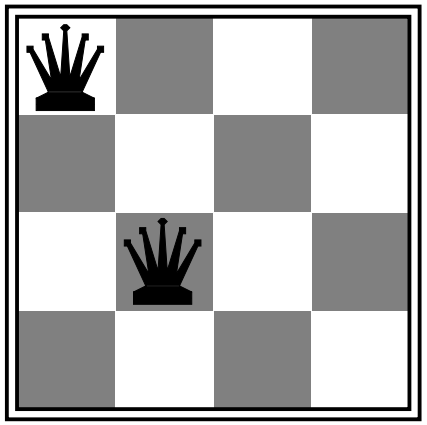
Consistent assignment: an assignment that does not violate any constraints

A solution to a CSP is a **consistent** and **complete** assignment

Example: 4-Queens as a CSP

Assume one queen in each column. Which row does each one go in?

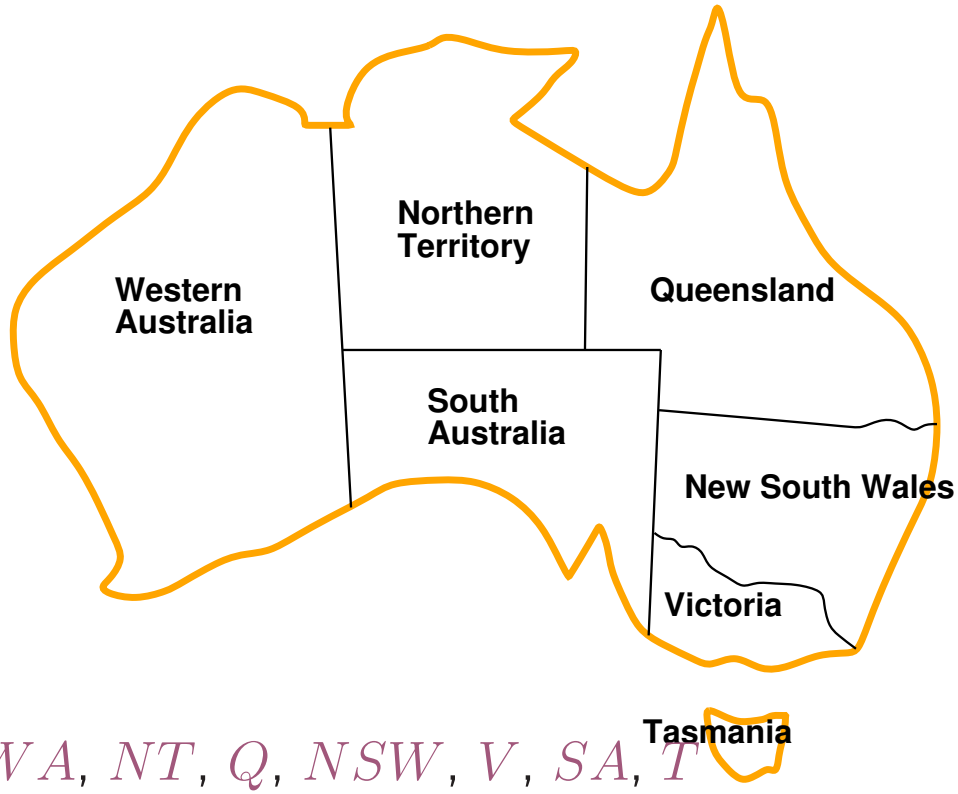
Variables Q_1, Q_2, Q_3, Q_4
Domains $D_i = \{1, 2, 3, 4\}$
Constraints
$Q_i \neq Q_j$ (cannot be in same row)
$|Q_i - Q_j| \neq |i - j|$ (or same diagonal)



Translate each constraint into a set of allowable values for its variables

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

Example: Map coloring



Variables: WA, NT, Q, NSW, V, SA, T

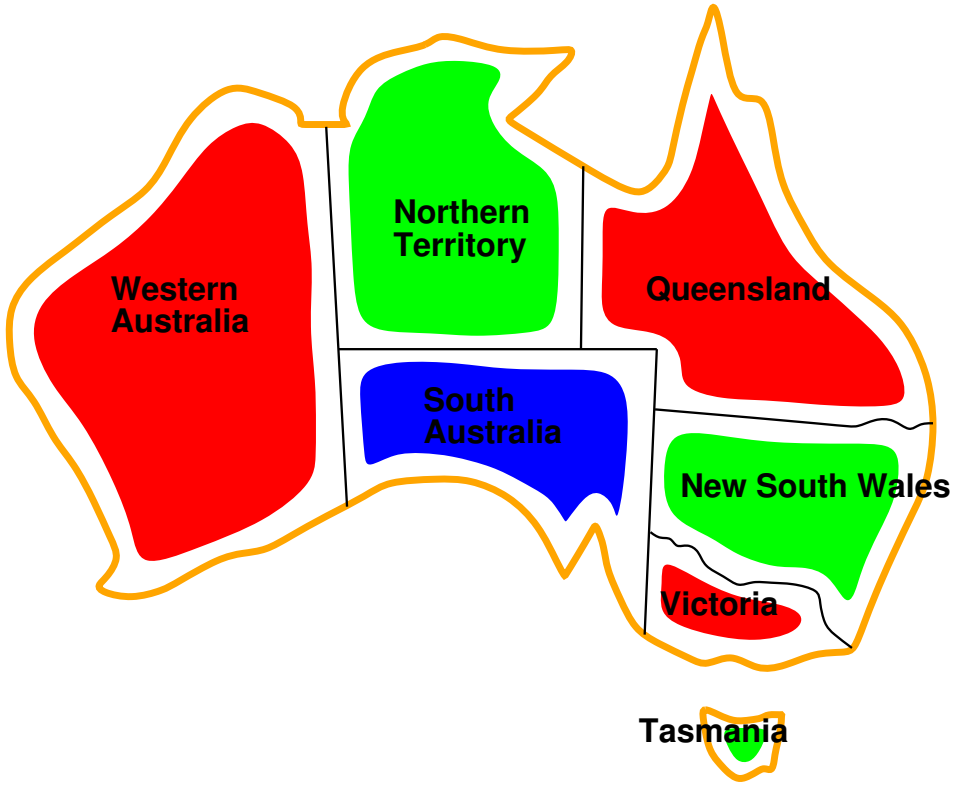
Domains: $D_i = \{red, green, blue\}$

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

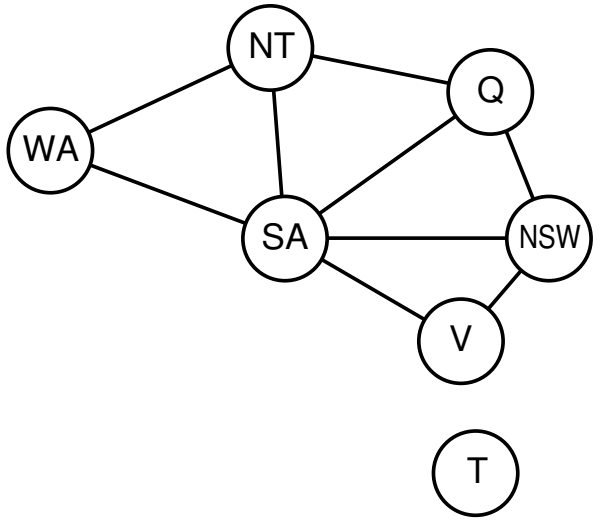


Solutions are assignments satisfying all constraints, e.g.,

$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue,$
$T = green\}$

Constraint graph

Constraint graph: nodes are variables, arcs show constraints
Constraint hypergraph: adding hypermodes for n-ary constraint



CSP algorithms use the graph structure to speed up search
E.g., Tasmania is an independent subgraph

Varieties of CSPs

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

e.g., disjunctive constraint $Red(x) \vee Blue(x)$

- e.g., job scheduling, variables are start/end days for each job
- need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- linear constraints solvable, nonlinear undecidable

Continuous variables

- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in poly time by LP (linear programming) methods

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

Dual graph: an n-ary CSP can be converted to a binary one

— one variable for each constraint in the original graph, and one binary constraint for each pair of constraints in the original graph that share variables

Preferences (precedence constraints, soft constraints), e.g., red is better than $green$

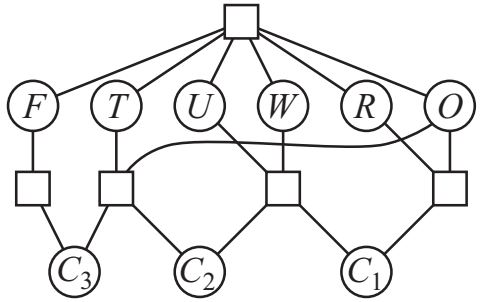
often representable by a cost for each variable assignment

$\rightarrow$ **constrained optimization** problems

Example: Cryptarithmetic

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

(a)



(b)

(a) a substitution of digits for letters s.t. the resulting sum is arithmetically correct

(b) constraint hypergraph with the squares for hypernodes

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) (square at the top, global constraint)

$O + O = R + 10 \cdot C_{10}$ (C_i for carrying digits, square at the most right), etc.

Inference in CSPs

CSPs

- Search - choose new variable assignment for several possibilities
- Inference — using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on — called constraint propagation

Constraint propagation

- may have variables with multiple possible values — have to search for a solution
- may be intertwined with search
- may be done as a preprocessing step before search starts

(Sometimes the preprocessing can solve the whole problem)

CSPs as search problems

CSP as search: states are defined by the values assigned so far

INITIAL: the empty assignment $\{\}$, in which all variables are unassigned

ACTION: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables

IS-GOAL: the current assignment is complete

PATH-COST: a constant cost (say, 1) for every step

Start with the straightforward, dumb approach, then fix it

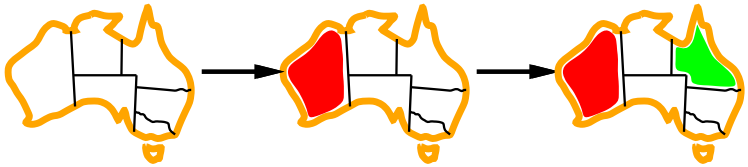
1) This is the same for all CSPs 
2) Every solution appears at depth d with n variables
   $\Rightarrow$ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves 

Constraint propagation

Constraint propagation (INFERENCE): using the constrains to reduce the number of legal values for a neighbor variable

repeatedly enforcing constraints locally $\Rightarrow$ local consistency

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

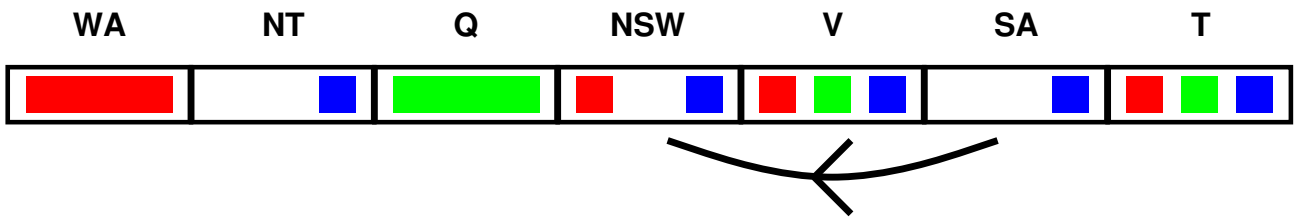


NT and SA cannot both be blue

Arc consistency

Basic form of propagation makes each arc consistent
— node consistency as special situation

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

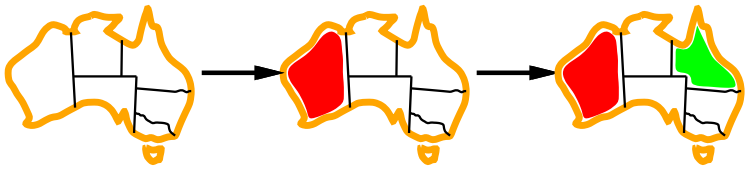


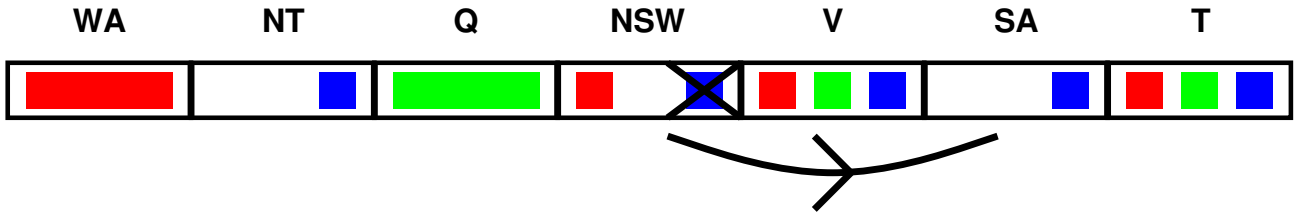
Arc consistency

Basic form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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



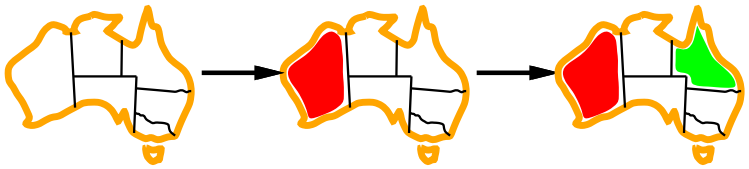


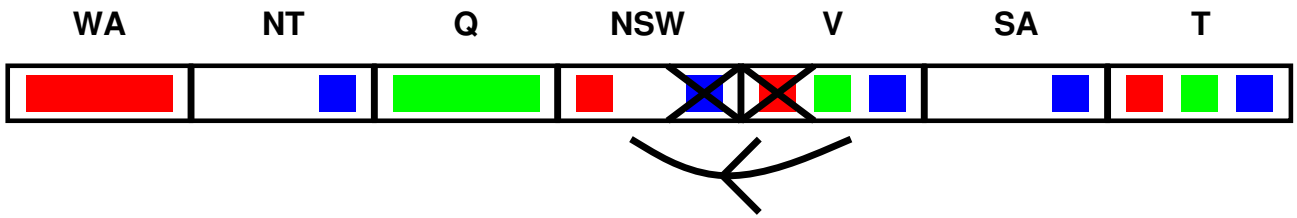
Arc consistency

Basic form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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





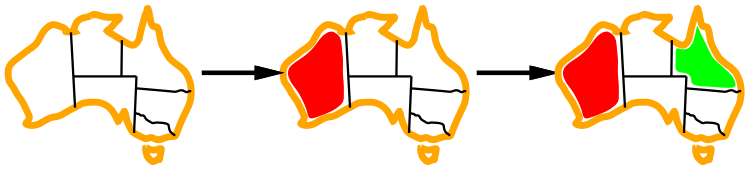
If X loses a value, neighbors of X need to be rechecked

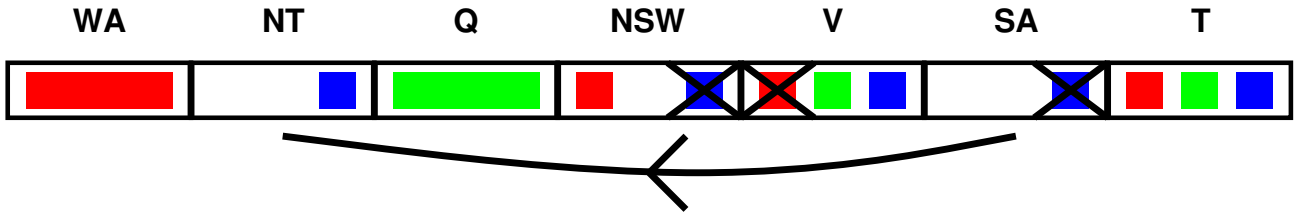
Arc consistency

Basic form of propagation makes each arc consistent (AC)

$X \to Y$ 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 detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

```
def AC3( csp)
    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 // making the domain smaller
            if size of D_i = 0 then return false // an inconsistency is found
            for each X_k in X_i.NEIGHBORS−{X_i} do
                add (X_k, X_i) to queue
    return true

def REVISE(csp, X_i, X_j)
    revised ← false
    for each x in D_i do
        if no value y in D_j allows (x,y) to satisfy the constraint X_i ↔ X_j
            then delete x from D_i
            revised ← true
    return revised // true iff revising the domain of X_i
```

Arc consistency algorithm

The complexity of AC3 is $O(cd^3)$

– Assume n variables, each with domain size at most d, and with c binary arcs

– Each arc (X_k, X_i) can be inserted in the queue only d times (because X_i has at most d values to delete)

– Checking consistency of an arc can be done in $O(d^2)$, and hence $O(cd^3)$ at total worst time

But detecting **all** is NP-hard (combination without arc consistency)

A variable X_i is generalized arc consistent w.r.t. an n-ary constraint if for every value v in the domain of X_i there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its X_i component equal to v

Path consistency

Arc consistency fails to make enough inferences

– e.g., the map-coloring problem with only two colors (say, red and blue)

AC3 does nothing for every variable is already arc consistent

there is no solution

Path consistency (PC): a two-variable set $\{X_i, X_j\}$ is path-consistent w.r.t. a third variable X_m if

for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$,

there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$

The PC algorithm achieves path consistency in much the same way that AC3 does

K-consistency

K-consistency: for any set of $k-1$ variables and for any consistent assignment to those variable,

a consistent value can always be assigned to any kth variable

Special cases:

– 1-consistency: given the empty set, one variable consistent, called node consistency

– 2-consistency: arc consistency

– 3-consistency: path consistency (for binary constraint networks)

A CSP is strongly k-consistent if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, $\cdots$ all the way down to 1-consistent

E.g., assume that a CSP with n nodes and make it strongly n-consistent

How to solve the problem??

Constraint inconsistency

Global constraint: involving an arbitrary number of variables (but not necessarily all variables)

Inconsistency checking: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint is inconsistent

Resource constraint: *atmost* constraint are assigned in total

Inconsistency checking: detecting the sum of the minimum values of the current domains

Bounds constraint: domains are represented by upper and lower bounds and are managed by bounds propagation

Bounds consistent: if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y

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

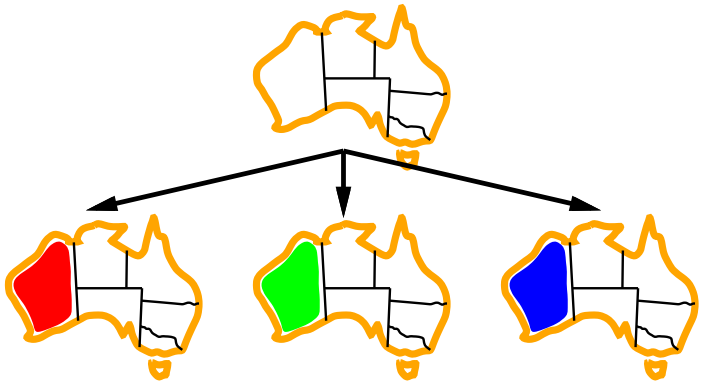
Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

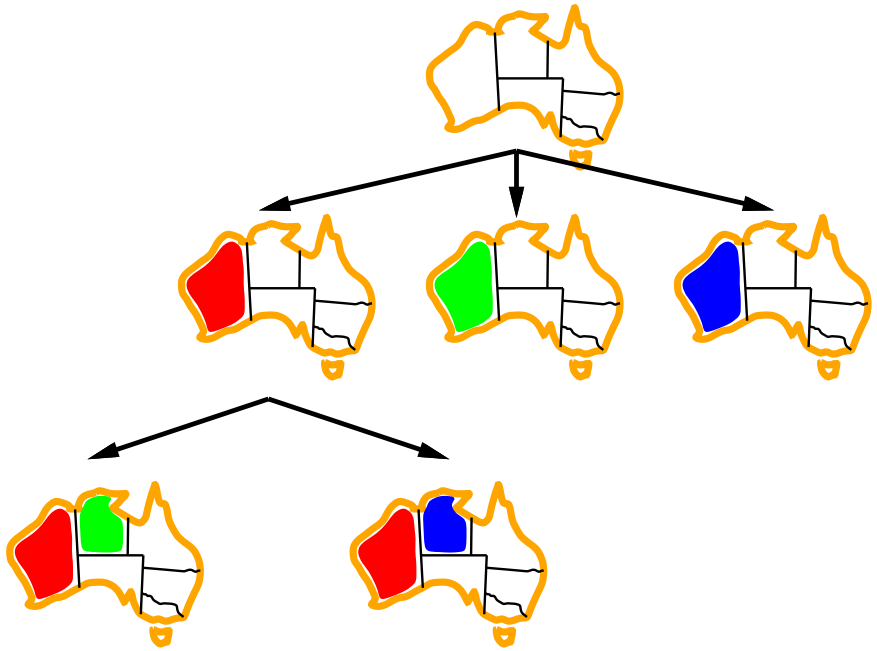
Can solve n-queens for $n \approx 25$

Backtracking example

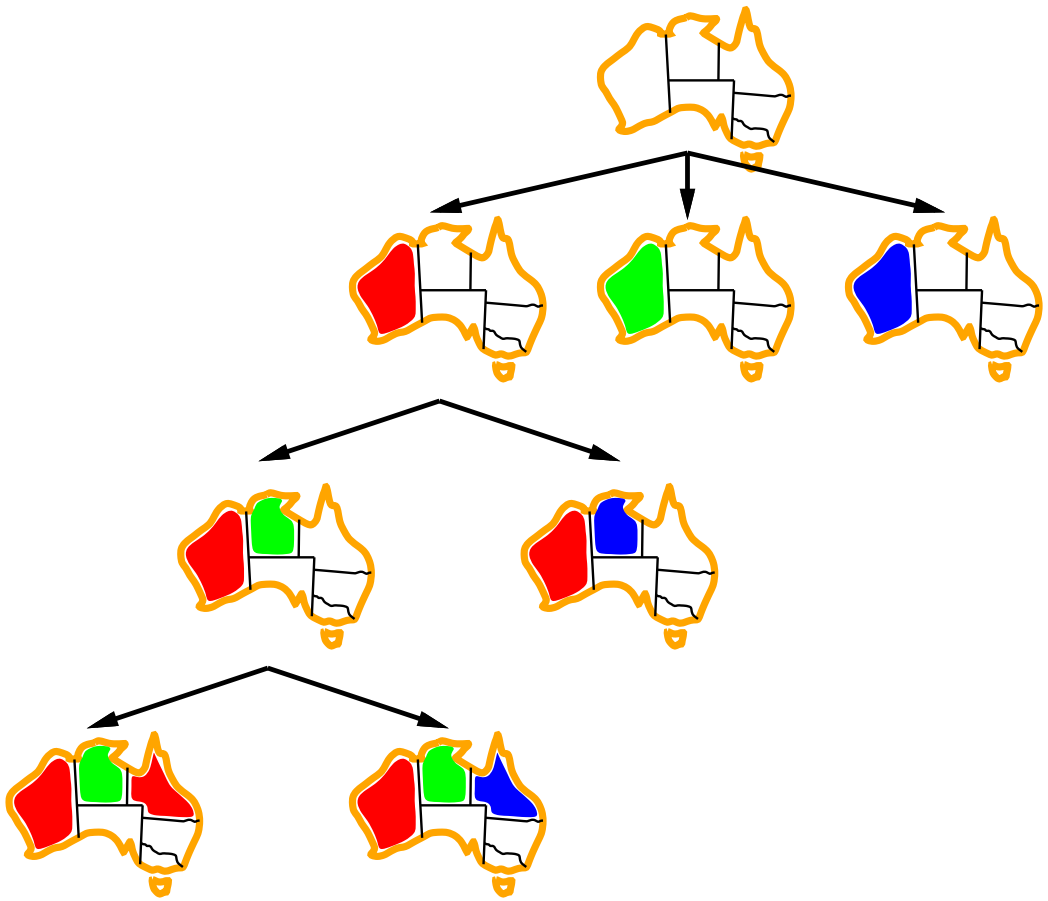
Backtracking example



Backtracking example



Backtracking example



Backtracking search

```
def BACKTRACKING-SEARCH(csp)
    return BACKTRACKING(csp, { })

def BACKTRACKING(csp, assignment)
    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 assignment
                result ← BACKTRACKING(csp, assignment)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
```

Improving backtracking efficiency

General-purpose methods can give huge gains in speed

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE)?
   (heuristic: minimum remaining values)
2. In what order should its values be tried (ORDER-DOMAIN-VALUES)?
   (heuristic: least constraining value)
3. Can we detect inevitable failure early? What inferences should be performed at each step (INFERENCE)?
   (i.e., constraint propagation, e.g., forward checking)
4. Can we take advantage of problem structure?
   (graph theory)

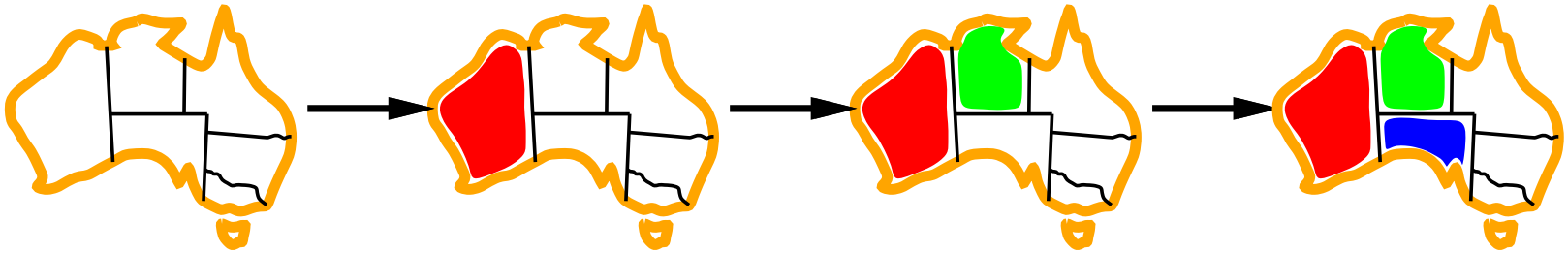
Minimum remaining values

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

MRV (minimum remaining values, or most constrained variable)

Choose the variable with the fewest legal values

– "fail-first" (for pruning) heuristic, better than random ordering



Doesn't help in choosing the first variable (region to color)

Degree heuristic

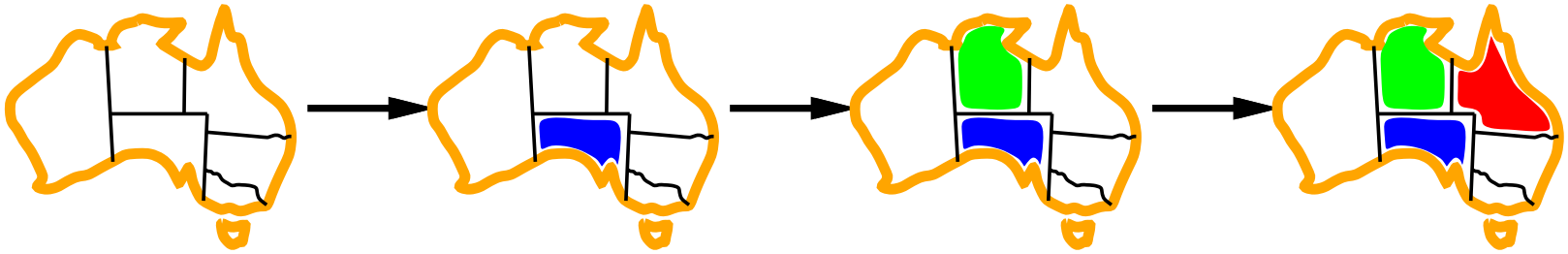
$var \leftarrow$ SELECT-UNASSIGNED-VARIABLE($csp, assignment$)

DH (degree heuristic)

Choose the variable with the most constraints on remaining variables

e.g., SA (blue) with highest degree 5

– choose the first variable to assign



Least constraining value

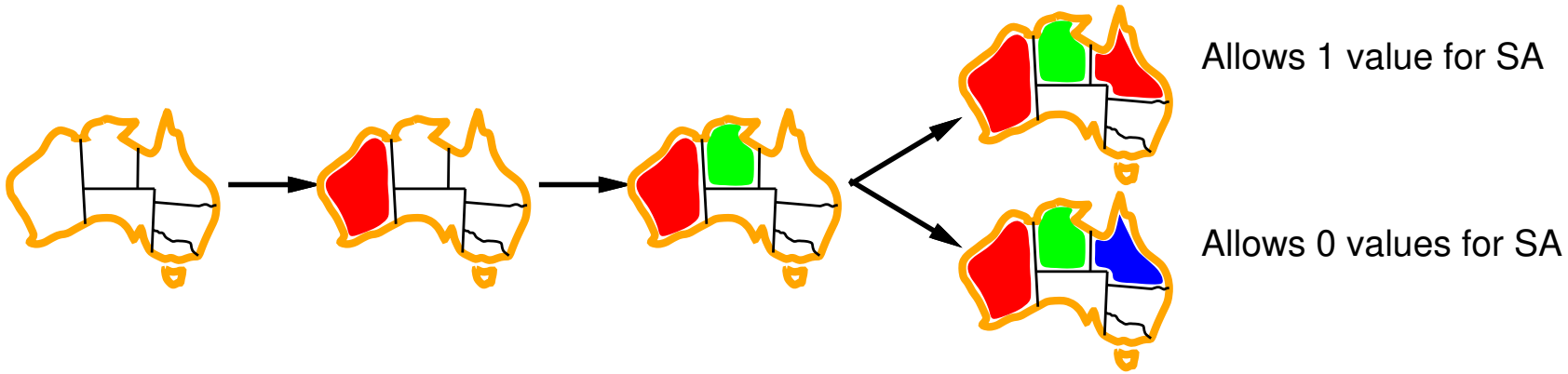
ORDER-DOMAIN-VALUES(*csp*, *var*, *assignment*)

LCV (least constraining value)

Given a variable, choose the least constraining value

(the one that rules out the fewest values in the remaining variables)

– "fail-last" (for one solution) heuristic, irrelevant to all solutions



Combining these heuristics makes 1000 queens feasible

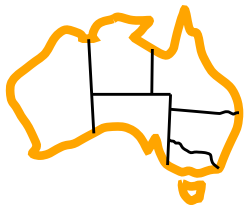
Forward checking

$inferences \leftarrow$ INFERENCE(csp, var, $assignment$)

FC (forward checking)

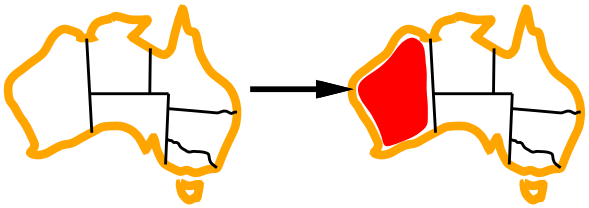
Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



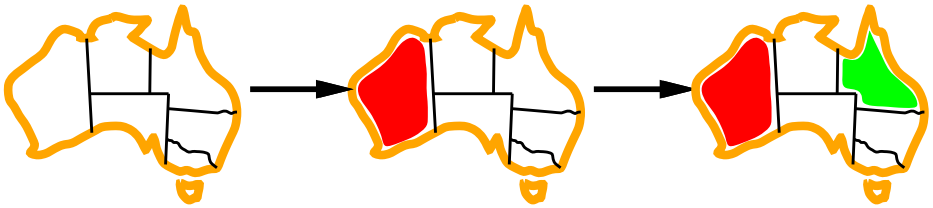
Forward checking

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



Forward checking

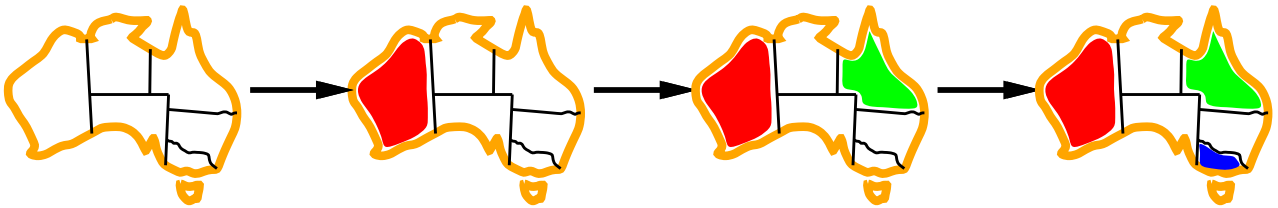
FC: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

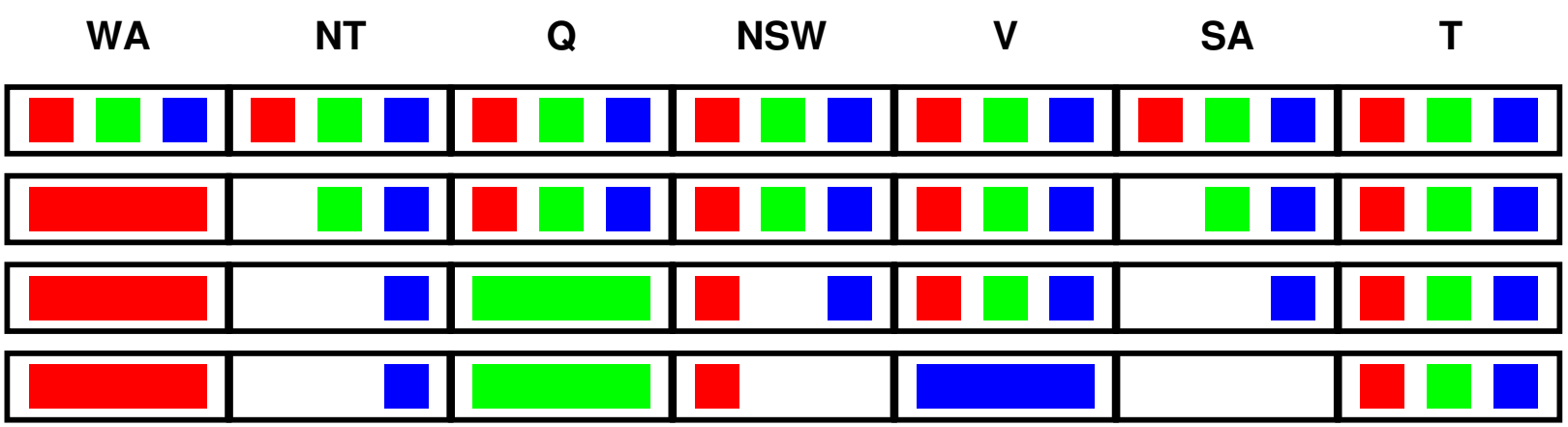


Forward checking

FC: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values





Forward checking inference has detected that the partial assignment is inconsistent with the constraints $\Rightarrow$ BACKTRACKING

Forward and backward

$inferences \leftarrow$ INFERENCE(csp, var, $assignment$)

FC doesn't detect all of the inconsistency, because it doesn't look ahead far enough

MAC (maintaining arc consistency): after a variable is assigned a value, the INFERENCE procedure calls AC3 $\Rightarrow$ more powerful than FC

BACKTRACKING is chronological because the most recent decision point is backed up $\Rightarrow$ looking backward

$\Leftarrow$ BACKJUMPING (modified BACKTRACKING)

— backtracks to the most recent conflict assignment

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 the value that violates the fewest constraints
- i.e., hillclimb with $h(n)$ = total number of violated constraints

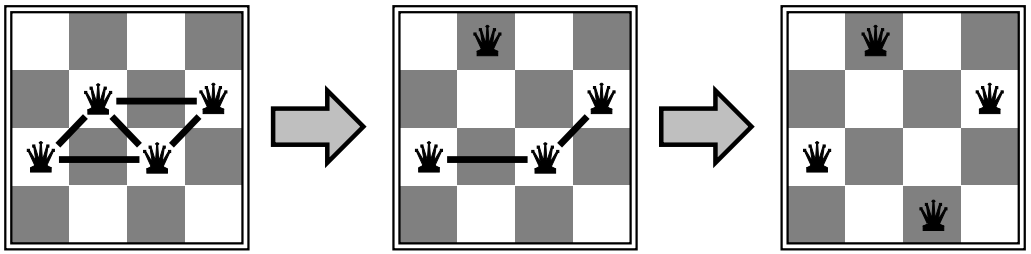
Example: 4-Queens

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



Min-conflicts algorithm

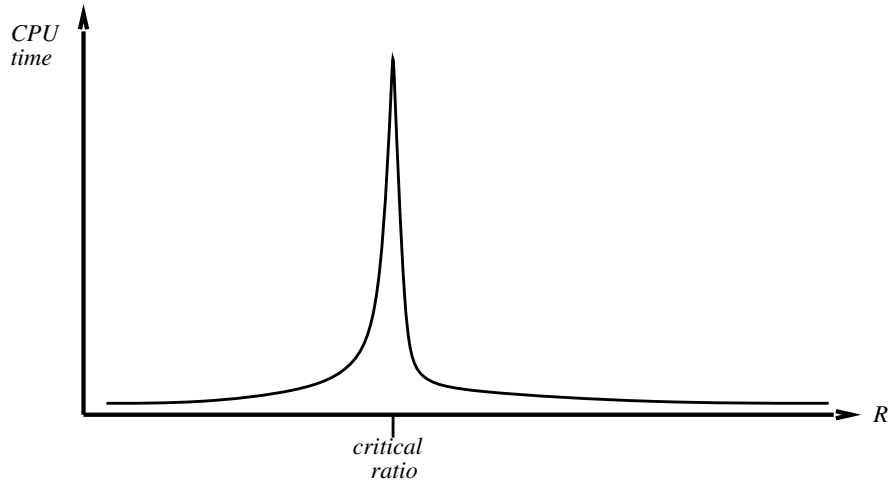
```
def MIN-CONFLICTS(csp, max-steps)
        max-steps, the number of steps allowed before giving up
  current ← an initial complete assignment for csp
  for i = 1 to max-steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
      set var=value in current
  end
  return failure
```

Performance of min-conflicts

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

The same appears to be true for any randomly-generated CSP **except** in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



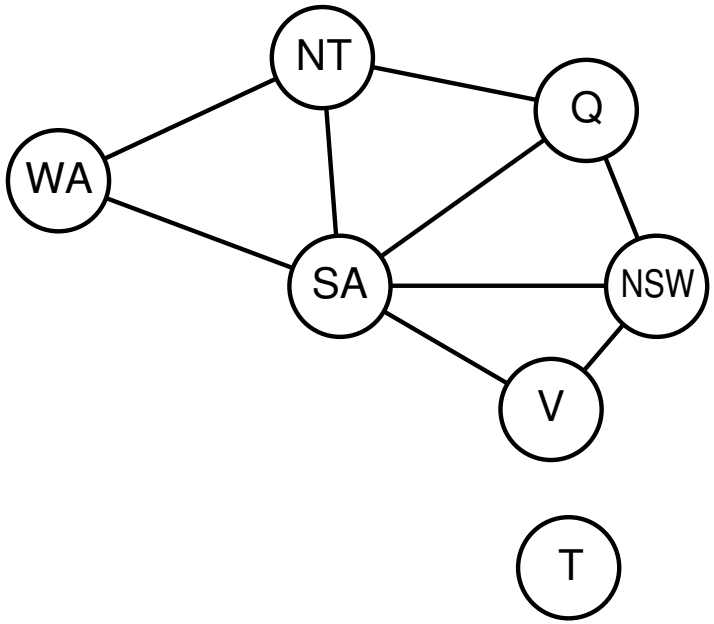
Structure and decomposition$^+$

The structure of a problem, represented as a constraint graph, can be used to find solutions quickly, and the only way to deal with real world problem is to decompose it into many subproblems

Suppose each subproblem has c variables out of n total

What is the worst-case solution cost?

Problem Structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Problem structure

Suppose each subproblem has c variables out of n total

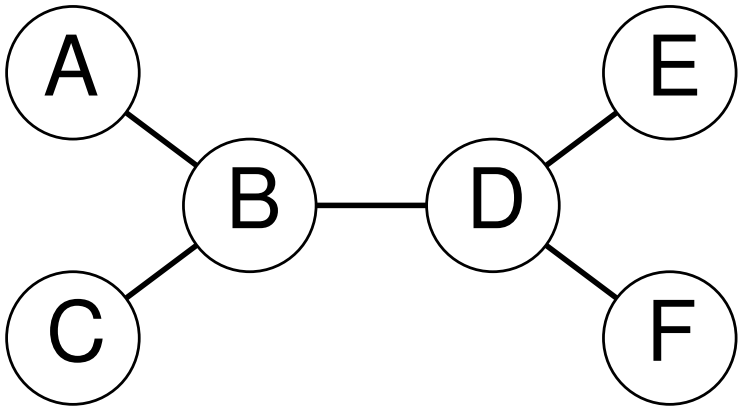
Worst-case solution cost is $n/c \cdot d^c$, **linear** in n

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



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

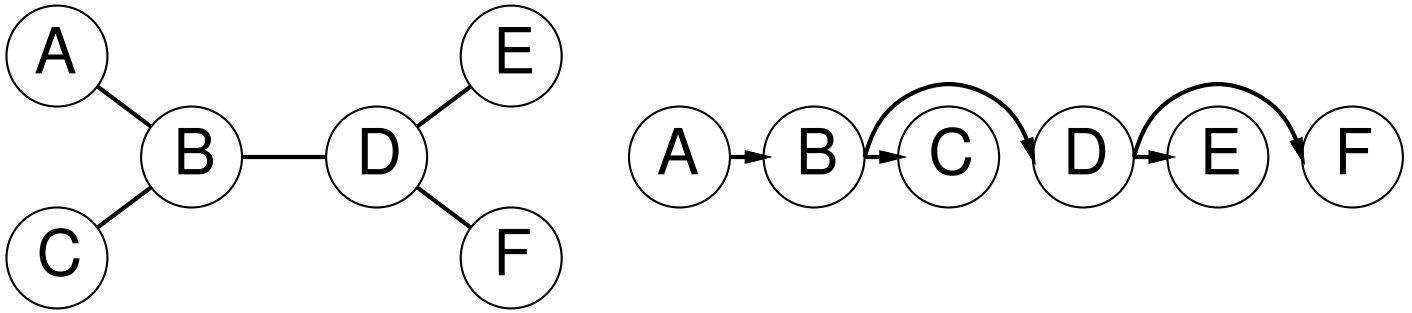
Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning (notice that the relation between syntactic restrictions and the complexity of reasoning)

Algorithm for tree-structured CSPs

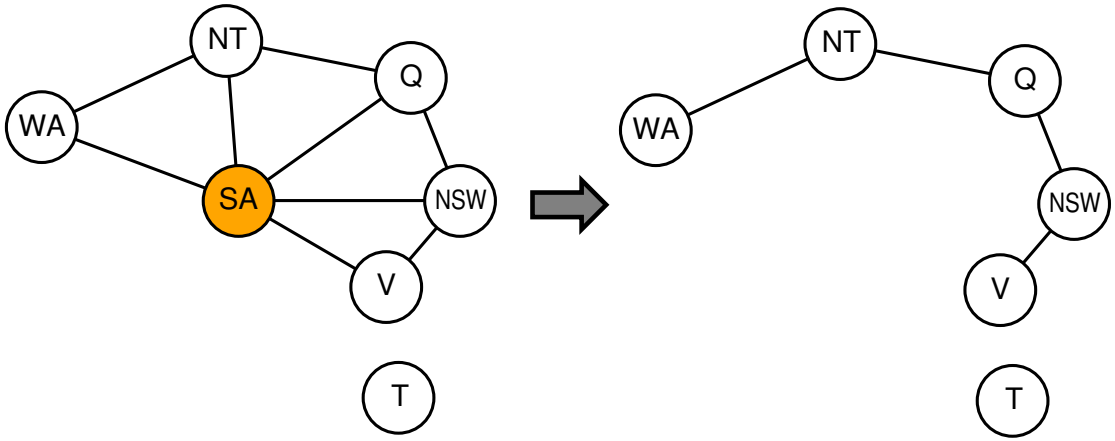
Procedure of tree CSPs

1. Choose a variable as root, order variables from root to leaves s.t. every node's parent precedes it in the ordering
   // see the following figure
2. For j from n down to 2, apply RemoveInconsistent$(Parent(X_j), X_j)$
3. For i from 1 to n, assign X_i consistently with $Parent(X_i)$



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

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

Real-world CSPs*

- Timetabling problems
  e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling, etc.

Many real-world problems involve real-valued variables

Constraint programming*

Constraint programming: modeling and solving problems by CSPs

– describing the problems by constraint programming language, such as ESSENCE/ZINC, in high-level modeling

– mapping the description into a set of constraints in low-level CSP format, which is then solved

Example: Course scheduling

Problem: Given a university with a set of courses, teachers, classrooms, and time slots, the objective is to schedule each course to a specific teacher, classroom, and time slot

Variables

- Each course c is assigned to a teacher t
- Each course c is assigned to a classroom r that meets the requirements of the course (e.g., capacity, equipment)
- Each course c is assigned to a time slot s

Example: Course scheduling

Constraints

- Teacher availability: Each teacher t can only teach one course at a given time slot s

- Classroom availability: Each classroom r can only be used for one course at a given time slot s

- No overlapping courses: No two courses can be scheduled for the same class at the same time

- Teacher qualification: Each course c requires a teacher t with the appropriate qualifications

- Classroom capacity: The capacity of each classroom r must be sufficient to accommodate the number of students enrolled in the scheduled course

- Student schedule conflicts: Minimize conflicts in the student schedules, ensuring that they do not have overlapping courses

Example: Course scheduling

Objective

Find a valid schedule that satisfies all the constraints while minimizing conflicts and ensuring an efficient use of resources

Solving the course schedule as a CSP??

- using algorithms mentioned above

Evaluation criteria

- Constraint satisfaction
- Resource utilization
- Student convenience
- Teacher workload balance