Chapter 6:
Constraint Satisfaction Problems

CS-4365 Artificial Intelligence

Chris Irwin Davis, Ph.D.

Email: chrisirwindavis@utdallas.edu
Phone: (972) 883-3574
Office: ECSS 4.603

- Defining Constraint Satisfaction
- Constraint Propagation: Inference in CSPs
- Backtracking Search for CSPs
- Local Search for CSPs
- The Structure of Problems

Defining CSP

Defining CSP

- A constraint satisfaction problem consists of three components, X, D, and C:
  - X is a set of variables, $\{X_i, \ldots, X_n\}$.
  - D is a set of domains, $\{D_i, \ldots, D_n\}$, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.
- Each domain D_i consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i.
- Each constraint C_i consists of a pair $\langle scope, rel \rangle$, where
  - *scope* is a tuple of variables that participate in the constraint and
  - *rel* is a relation that defines the values that those variables can take on.

Defining CSP

- A relation can be represented as:
  - an explicit list of all tuples of values that satisfy the constraint, or
  - an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.
- For example, if X_1 and X_2 both have the domain $\{A, B\}$, then the constraint saying the two variables must have different values can be written as either:
  - $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ ← explicit list
  - $\langle (X_1, X_2), X_1 \neq X_2 \rangle$ ← abstract relation

Solving CSPs

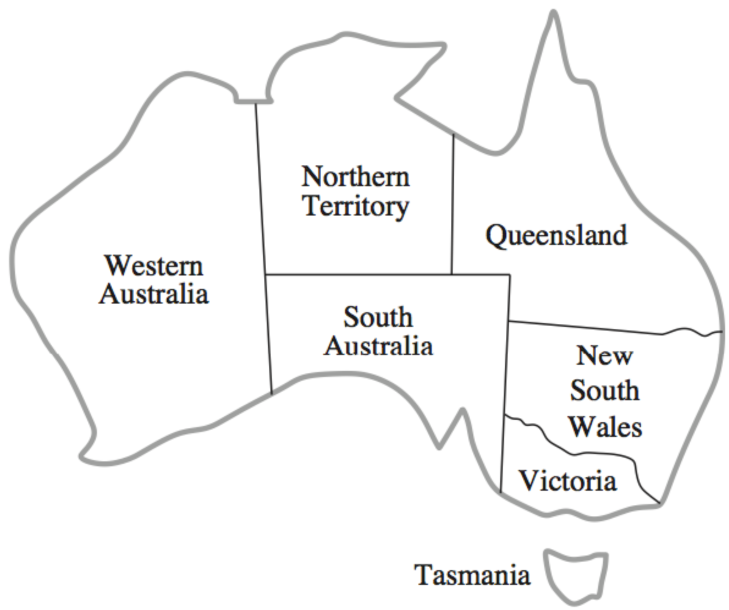
- To solve a CSP, we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$
- An assignment that does not violate any constraints is called a **consistent** or legal assignment.
- A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that assigns values to only some of the variables.

Solving CSPs

- State is defined by a set of variables X_i with values from domain D_i
- Goal test is to satisfy a set of constraints on variables

An Example

- The principal states and territories of Australia.
- Coloring this map can be viewed as a constraint satisfaction problem (CSP).
- The goal is to assign colors to each region so that no neighboring regions have the same color.



An Example

- Variables
  - $X = \{WA, NT, Q, NSW, V, SA, T\}$
- Domain (of each variable)
  - $D_i = \{red, green, blue\}$
- Constraints
  - Adjacent regions must have different colors
    - e.g. WA $\neq$ NT (if the language allows this), or otherwise
    - (WA, NT) $\in$ {(red, green), (red, blue), (green, red), … , etc.}

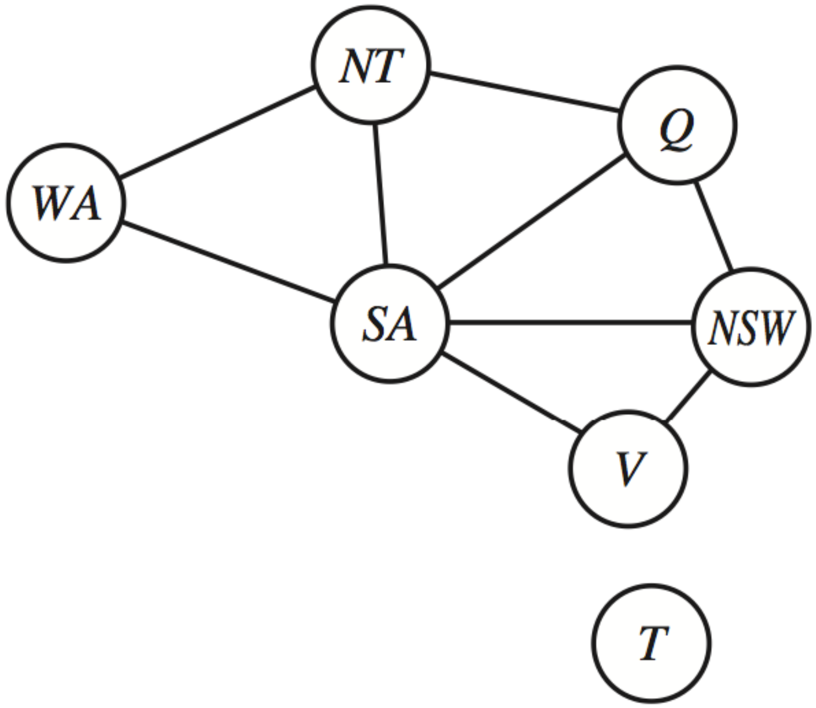
An Example

- Solutions?
  - There are many possible, e.g.
  - { WA = red, NT = green, Q = red, NSW = green, V = red, SA= blue, T = red }
- It can be helpful to visualize a CSP as a *constraint graph*.
- The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

An Example

- The map-coloring problem represented as a constraint graph.

- Binary CSP:
  - Each constraint relates at most two variables
- Constraint Graph
  - Nodes are variables
  - Arcs show constraints



Efficiency of CSP

- Why formulate a problem as a CSP?
- One reason is that the CSPs yield a natural representation for a wide variety of problems;
    - If you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique.

Efficiency of CSP

- In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space.
  - For example, once we have chosen {SA = blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.
  - *Without* taking advantage of constraint propagation, a search procedure would have to consider $3^5 = 243$ assignments for the five neighboring variables;
  - *With* constraint propagation we never have to consider blue as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.

Efficiency of CSP

- In regular state-space search we can only ask: is this specific state a goal?
  - No? What about this one?
- With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.
- Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter.
- As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

Example 2

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques.
- Consider the problem of scheduling the assembly of a car.
  - The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes.
  - Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once.
  - Constraints can also specify that a task takes a certain amount of time to complete.

Example 2

- We consider a small part of the car assembly, consisting of 15 tasks:
  - Install axles (front and back), **2 tasks**
  - Affix all four wheels (right and left, front and back), **4 tasks**
  - Tighten nuts for each wheel, **4 tasks**
  - Affix hubcaps, and **4 tasks**
  - Inspect the final assembly. **1 task**
- We can represent the tasks with 15 variables:
  - 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.
- The value of each variable is the time that the task starts.

Example 2

- Next we represent precedence constraints between individual tasks. Whenever a task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form
  - $T_1 + d_1 \leq T_2$
- In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write:
  - $\text{Axle}_F + 10 \leq \text{Wheel}_{LF}$
  - $\text{Axle}_F + 10 \leq \text{Wheel}_{RF}$
  - $\text{Axle}_B + 10 \leq \text{Wheel}_{LB}$
  - $\text{Axle}_B + 10 \leq \text{Wheel}_{RB}$

Example 2

- Next we say that, for each wheel, we must
  - Affix the wheel (which takes 1 minute), $d_{\text{wheel}} = 1$, then
  - Tighten the nuts (2 minutes), $d_{\text{nuts}} = 2$, and finally
  - Attach the hubcap, (1 minute), $d_{\text{hubcap}} = 1$.

Example 2

- Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjunctive constraint to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:
  - $(Axle_F + 10 \leq Axle_B)$ **or** $(Axle_B + 10 \leq Axle_F)$.
- This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_B$ can take on.

Example 2

- We also need to assert that the inspection comes last and takes 3 minutes. For every variable except Inspect we add a constraint of the form

$$X + d_X \leq Inspect.$$

- Finally, suppose there is a requirement to get the whole assembly done in 30 minutes.
- We can achieve that by limiting the domain of all variables:
  - $D_i = \{1, 2, 3, \ldots, 27\}$

Varieties of Constraints

- Unary constraints involve a single variable
  - SA $\neq$ green
- Binary constraints involve two variables
  - SA $\neq$ WA
  - color(SA) $\neq$ color(WA)
- Higher-order constraints involve 3 or more variables
  - e.g. cryptarithmetic, column constraints, etc.
- Preferences (soft constraints)
  - e.g. *red* is better than *green*
  - Often represented by a cost for each variable assignment

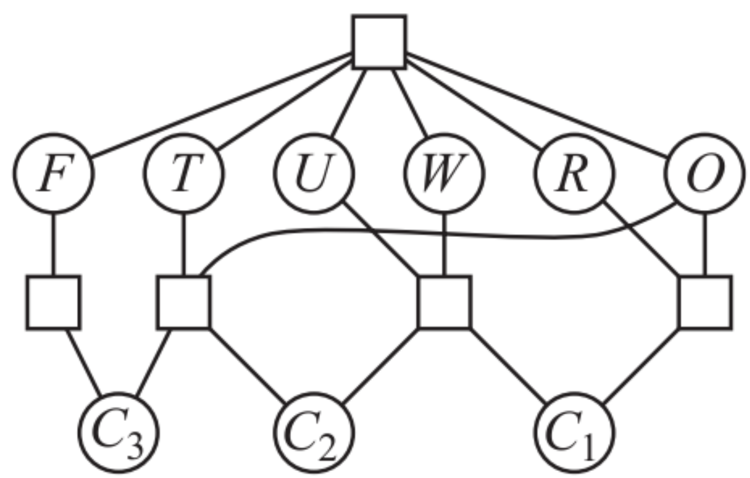
Example: Cryptarithmetic

```
   T W O
 + T W O
 -------
 F O U R
```

Example: Cryptarithmetic Puzzle

- 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
  - ALL-DIFF(F, T, U, W, R, O)
  - $O + O = R + 10 * C_1$,
  - etc.

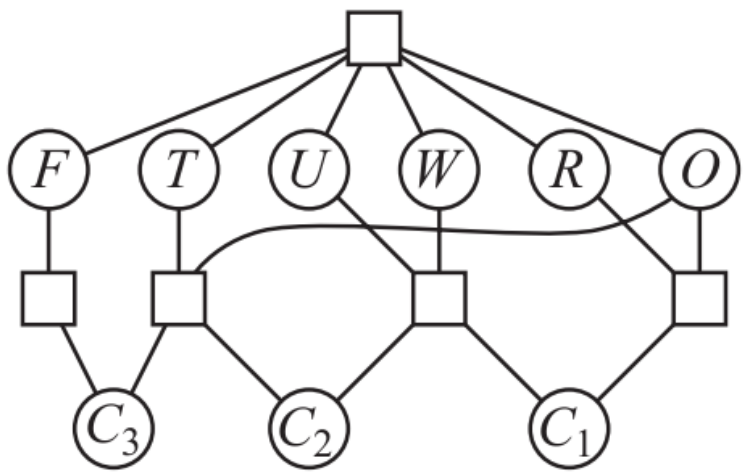
```
    T W O
+   T W O
---------
  F O U R
```



Example: Cryptarithmetic Puzzle

- ALL-DIFF(F, T, U, W, R, O)
- $O + O = R + 10 * C_1$
- $C_1 + W + W = U + 10 * C_2$
- $C_2 + T + T = O + 10 * C_3$
- $C_3 = F$,

```
   T W O
+  T W O
--------
 F O U R
```



§6.2 Constraint Propagation: Inference in CSPs

Real World CSPs

- Assignment problems
  - e.g. "Who teaches that class?"
- Time table problems
  - e.g. which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Floor planning
- *Notice that many real-world problems involve $\mathbb{R}$ valued (i.e. real numbered) variables*

Standard Search Formulation (incremental)

UTD

- Let's start with the straightforward (dumb) 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.
  - $\Rightarrow$ fail if no legal assignments (not fixable!)
  - Goal test: the current assignment is complete

Standard Search Formulation (incremental)

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

Node Consistency

- A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints.
  - For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable SA starts with domain {*red*, *green*, *blue*}, and we can make it node consistent by eliminating *green*, leaving SA with the reduced domain {*red*, *blue*}.
- We say that a **network** is node-consistent if every variable in the network is node-consistent.

Node Consistency

- It is always possible to eliminate all the unary constraints in a CSP by running node consistency.
- It is also possible to transform all *n*-ary constraints into binary ones.
  - ALL-DIFF?
- Because of this, it is common to define CSP solvers that work with only binary constraints;
  - The authors make that assumption for the rest of this chapter, except where noted.

Arc Consistency

- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.
- More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j).
- A network is arc-consistent if every variable is arc consistent with every other variable.

Arc Consistency

- For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of single digits.
- We can write this constraint explicitly as:
  - $(X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9))\}$
- To make X arc-consistent with respect to Y , we reduce X's domain to $\{0, 1, 2, 3\}$.
- If we also make Y arc-consistent with respect to X, then Y's domain becomes $\{0, 1, 4, 9\}$ and the whole CSP is arc-consistent.

Arc Consistency

- On the other hand, arc consistency can do nothing for the Australia map-coloring problem.
- Consider the following inequality constraint on (SA,WA):
  - {(*red*, *green*), (*red*, *blue*), (*green*, *red*), (*green*, *blue*), (*blue*, *red*), (*blue*, *green*)}
- No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

Arc Consistency: AC-3

- The most popular algorithm for arc consistency is called AC-3. To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider.
- Initially, the queue contains all the arcs in the CSP.
- AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j .
- If this leaves D_i unchanged, the algorithm just moves on to the next arc.
- But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i.

Arc Consistency: AC-3

- We need to do that because the change in D_i might enable further reductions in the domains of D_k, even if we have previously considered X_k.
- If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure.
- Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

Path Consistency

- Arc consistency can go a long way toward reducing the domains of variables,
  - sometimes finding a solution (by reducing every domain to size 1) and
  - sometimes finding that the CSP cannot be solved (by reducing some domain to size 0).
- But for other networks, arc consistency fails to make enough inferences.

Path Consistency

- Consider the map-coloring problem on Australia, but with only two colors allowed, *red* and *blue*.
- Arc consistency can do nothing because every variable is already arc consistent: each can be *red* with *blue* at the other end of the arc (or vice versa).
- But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone.

Path Consistency

- Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints).
- To make progress on problems like map coloring, we need a stronger notion of consistency.
- **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

Path Consistency

- A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to 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 $\{X_m, X_j\}$.
  - This is called path consistency because one can think of it as looking at a path from X_i to X_j with X_m in the middle.

K-Consistency

- Stronger forms of propagation can be defined with the notion of 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 k^{th} variable.
- 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called **node consistency**.
- 2-consistency is the same as **arc consistency**.
- For binary constraint networks, 3-consistency is the same as **path consistency**.

§6.3 Backtracking Search

Backtracking Search

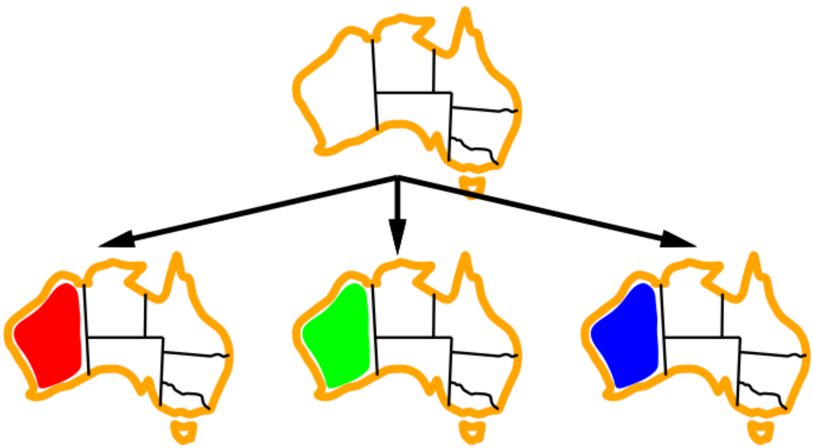
- Variable assignments are commutative, i.e.,
  - [WA=red **then** NT =green] same as [NT =green **then** WA=red]
- Only need to consider assignments to a single variable at each node
  - $\Rightarrow b=d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called backtracking search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

Backtracking Search

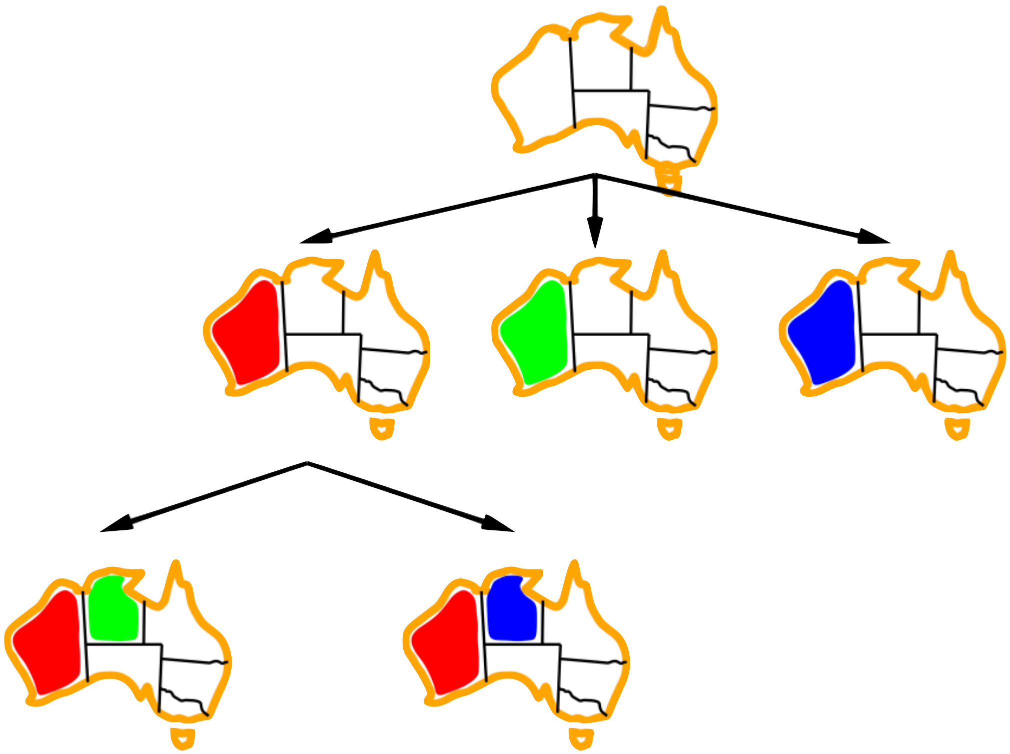
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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

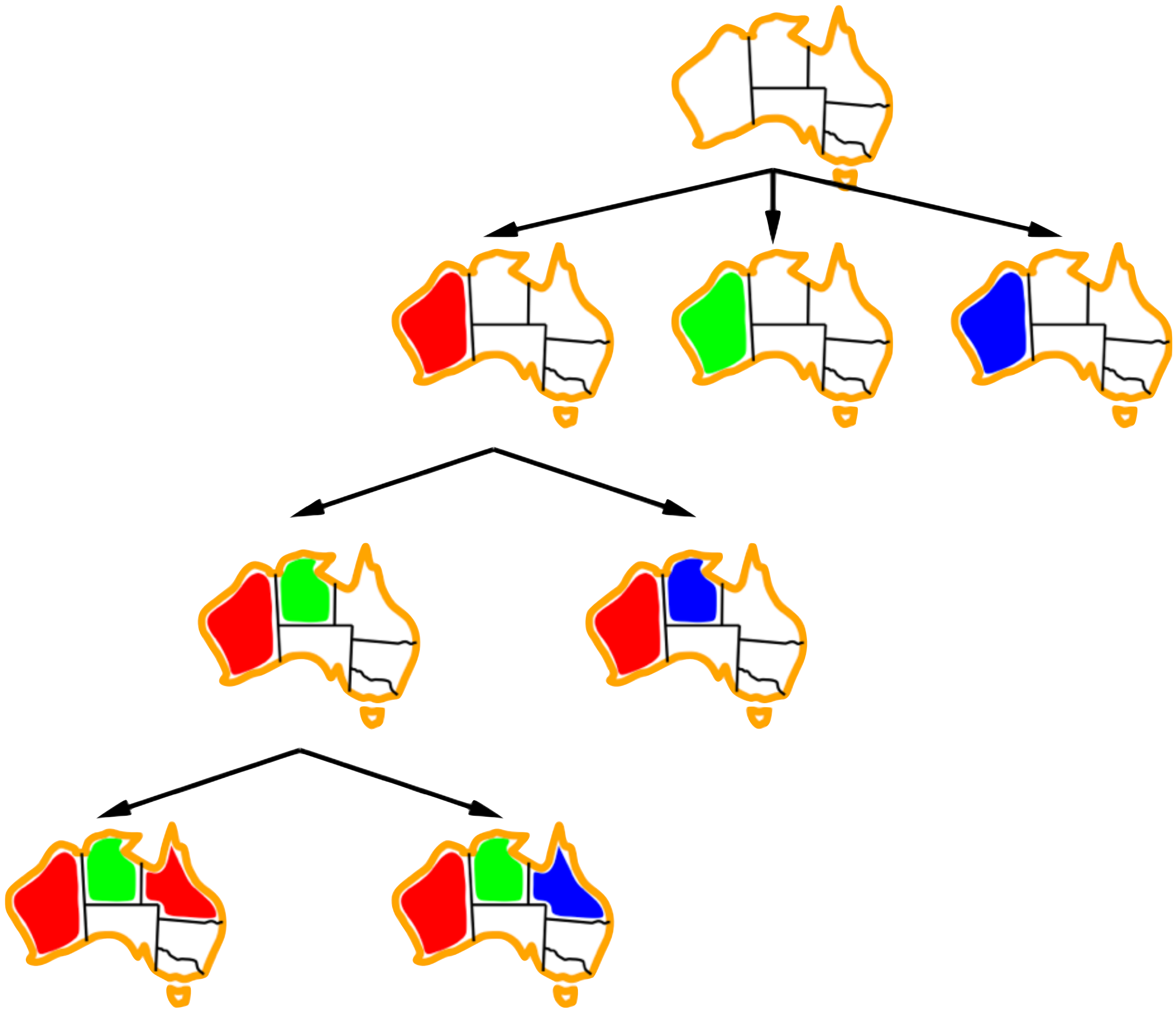
Backtracking Example



Backtracking Example



Backtracking Example

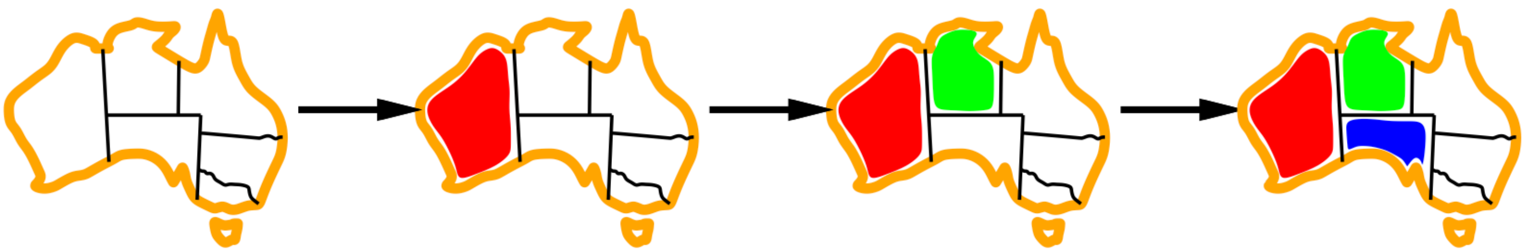


Improving Backtracking Efficiency

- 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 take advantage of problem structure?

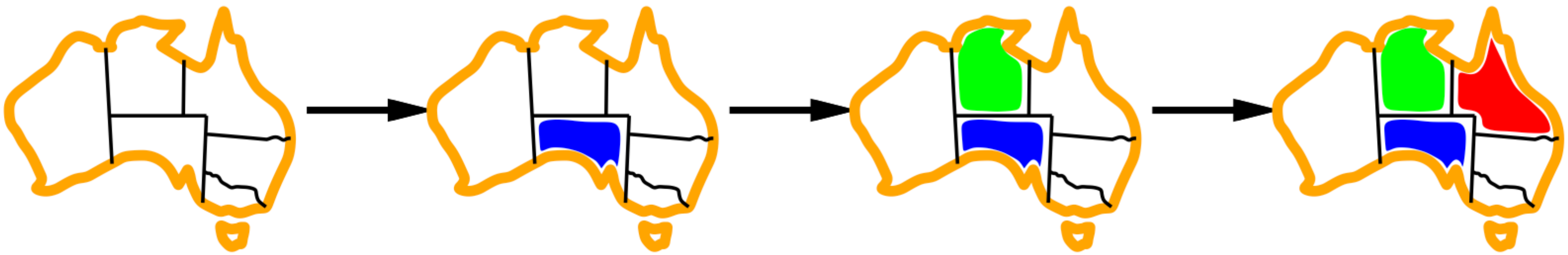
Minimum Remaining Values (MRV)

- Minimum remaining values (MRV):
  - choose the variable with the fewest legal values



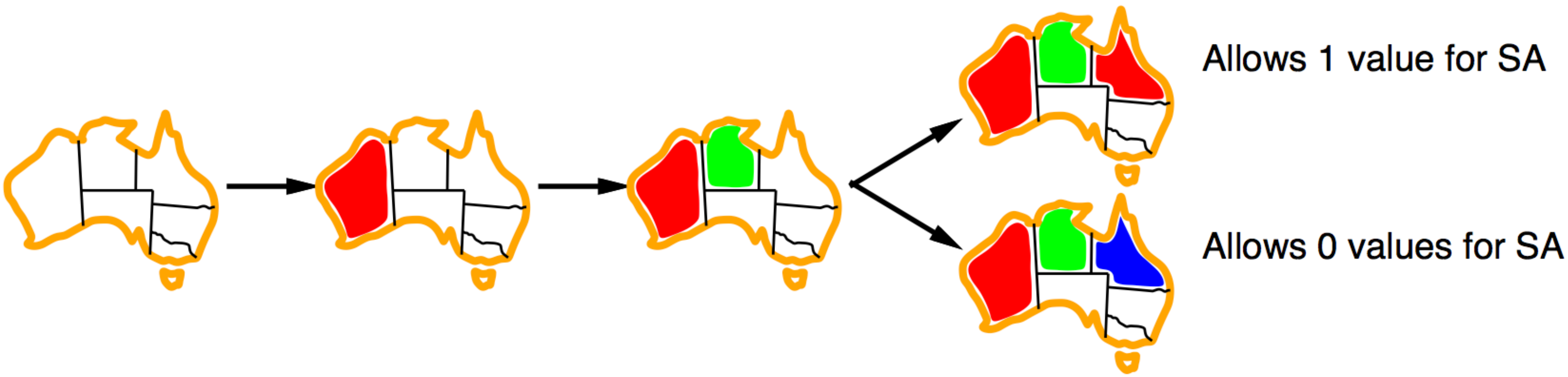
Degree Heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
  - Choose the variable with the most constraints on remaining variable



Least Constraining Value (LCV)

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



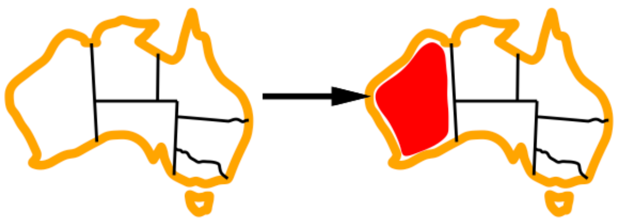
- Combining these heuristics makes 1000 queens feasible

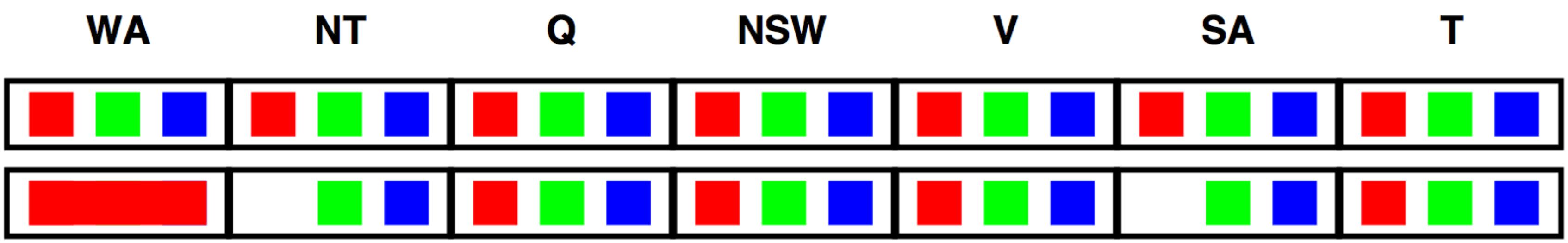
Forward Checking

- 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



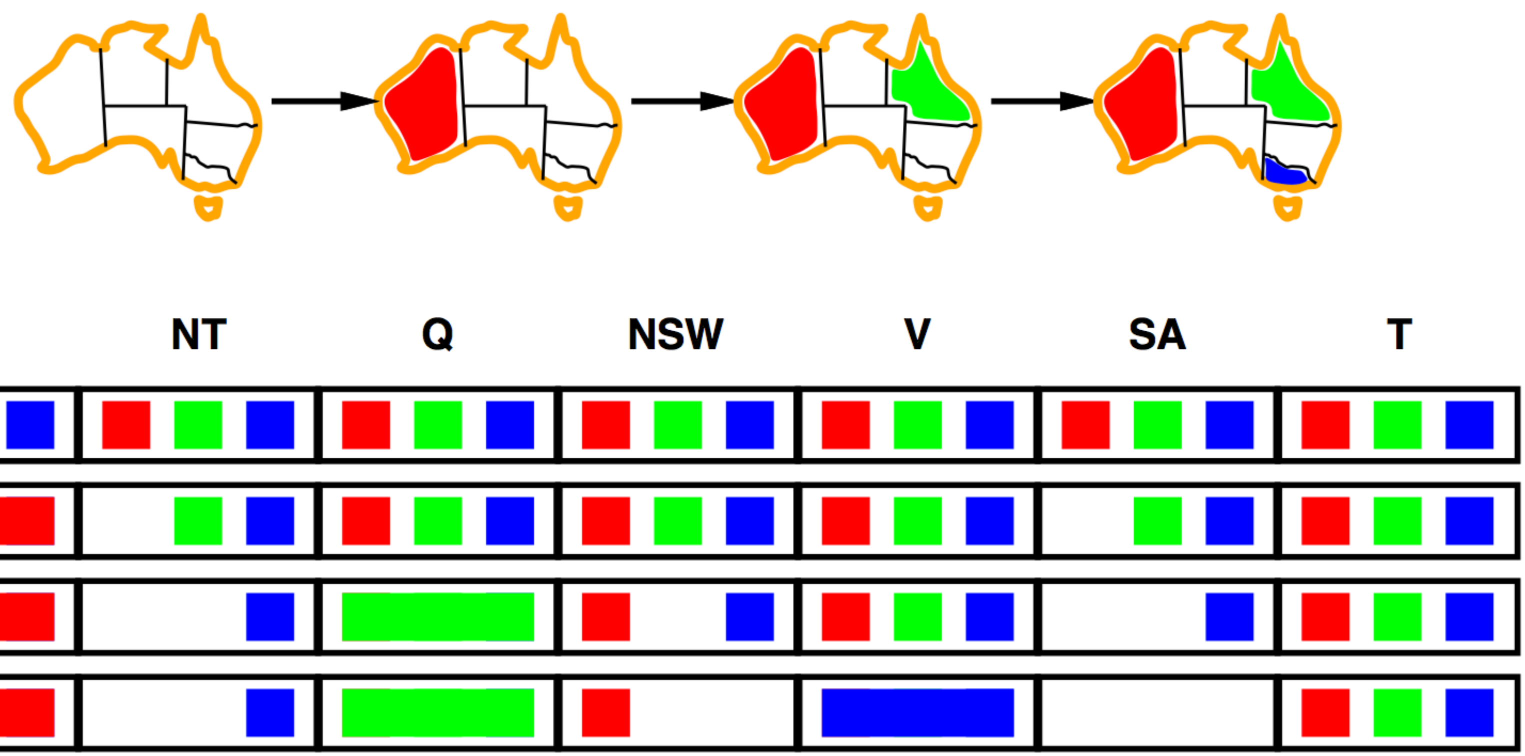


Forward Checking

- 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

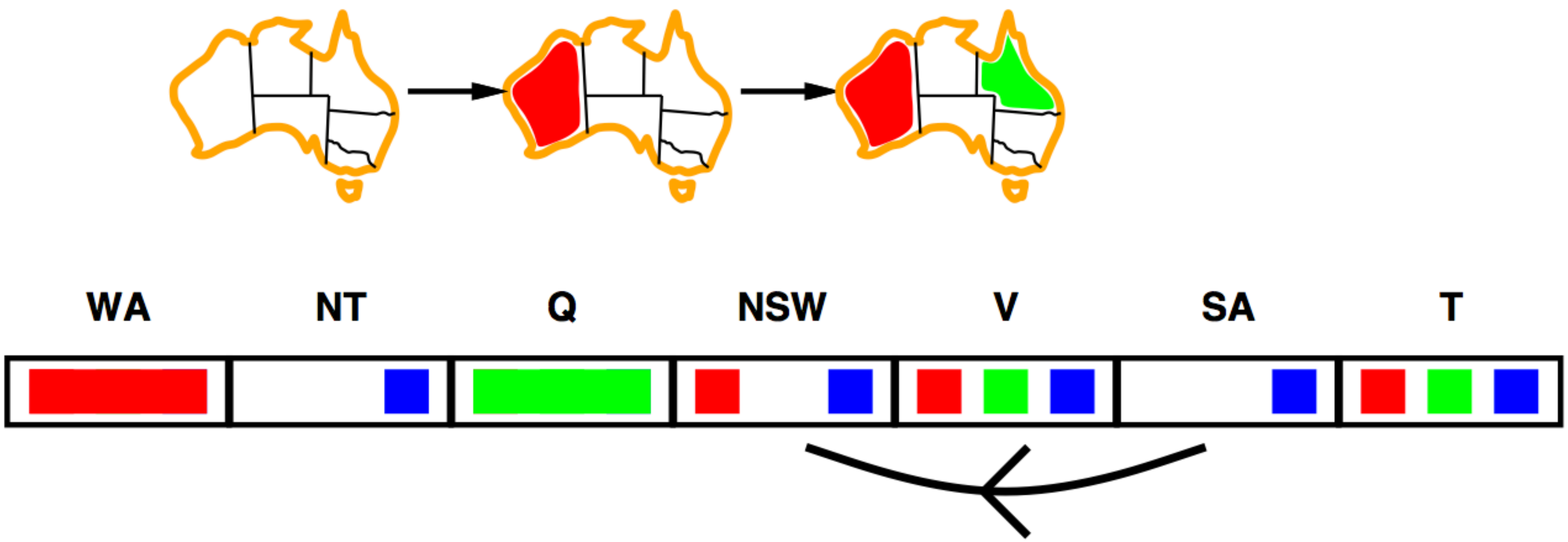


Constraint Propagation

- 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

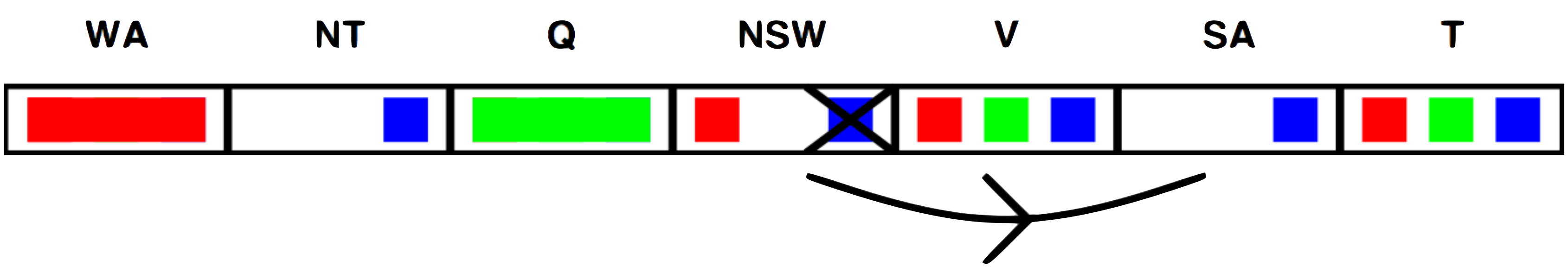
Arc Consistency

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



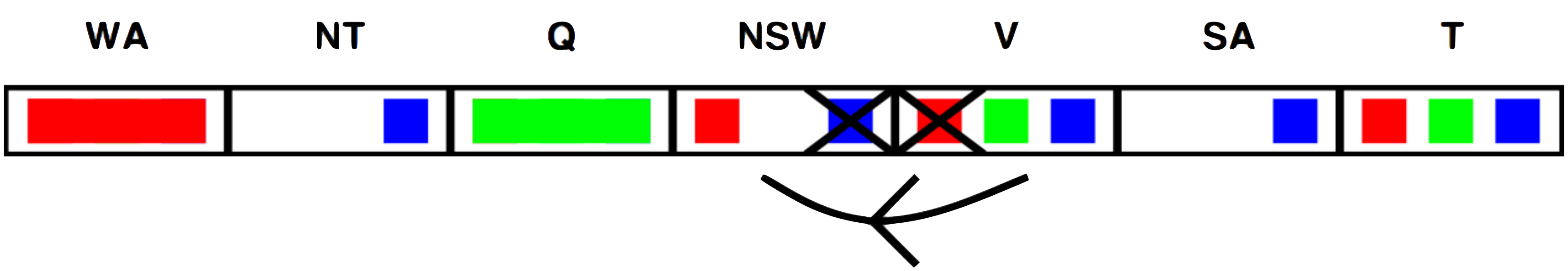
Arc Consistency

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



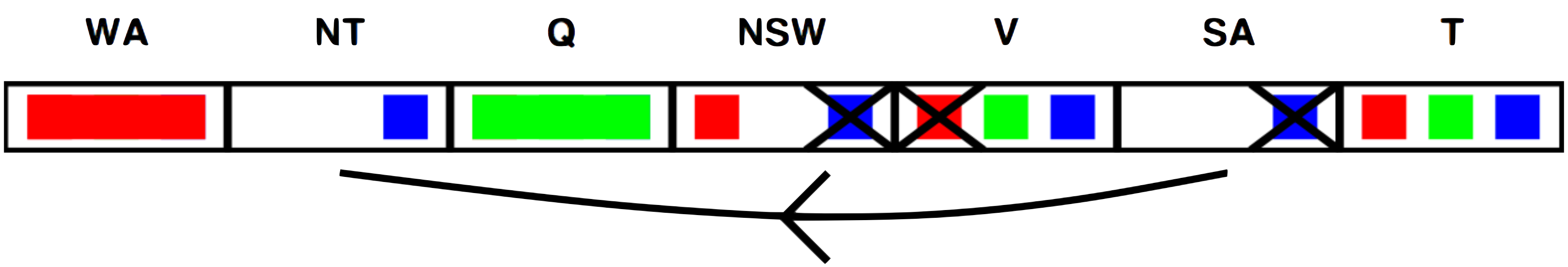
Arc Consistency

- Simplest form of propagation makes each arc consistent
  - X → 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
  - If X loses a value, neighbors of X need to be rechecked for every value *x* of X there is some allowed *y*
  - Arc consistency detects failure earlier than forward checking
  - Can be run as a preprocessor or after each assignment



Arc Consistency Algorithm

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local va

   while q
      (Xi,
      if RE
         fo
```

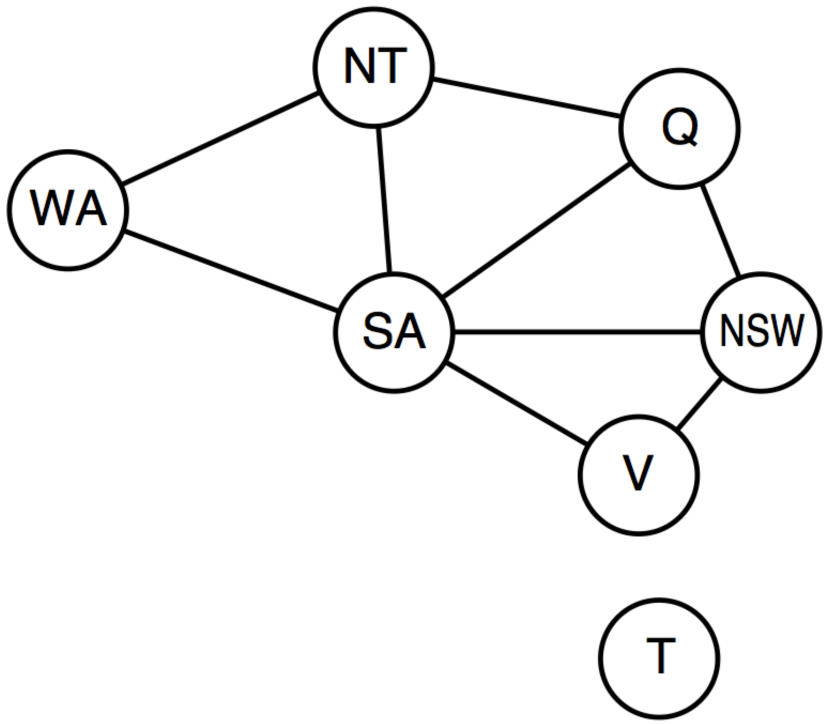
$O(n^2 d^3)$, can be reduced to $O(n^2 d^2)$

(but detecting all is NP-hard)

```
function R                                              succeeds
   removed
   for eac
      if no                                             Xi ↔ Xj
         then delete x from DOMAIN[Xi]; removed ← true
   return removed
```

Problem Structure

- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph



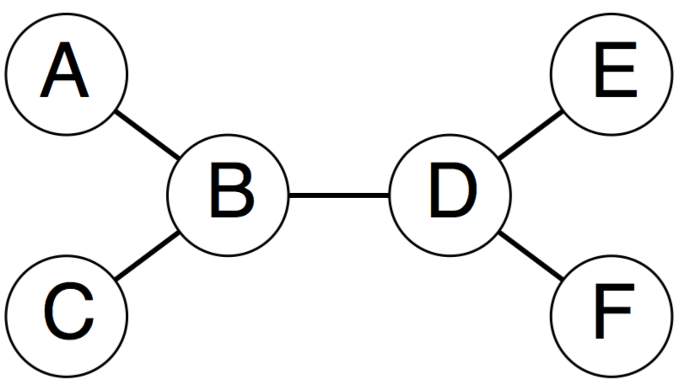
Problem Structure (cont'd)

- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $n/c * d^c$, linear in n

• e.g., n=80, d=2, c=20
• Backtracking search $O(d^n)$
  - 2^{80} = 4 billion years at 10 million nodes/sec
• Constrain variable domain $O(n/c\ d^c)$
  - $4 * 2^{20}$ = 0.4 seconds at 10 million nodes/sec

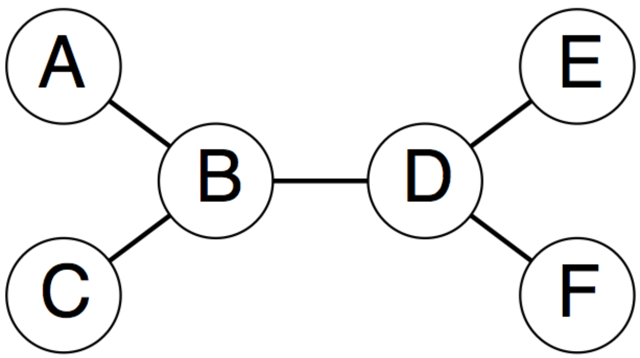
Tree-structured CSPs

- 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:
  - An important example of the relation between syntactic restrictions and the complexity of reasoning.



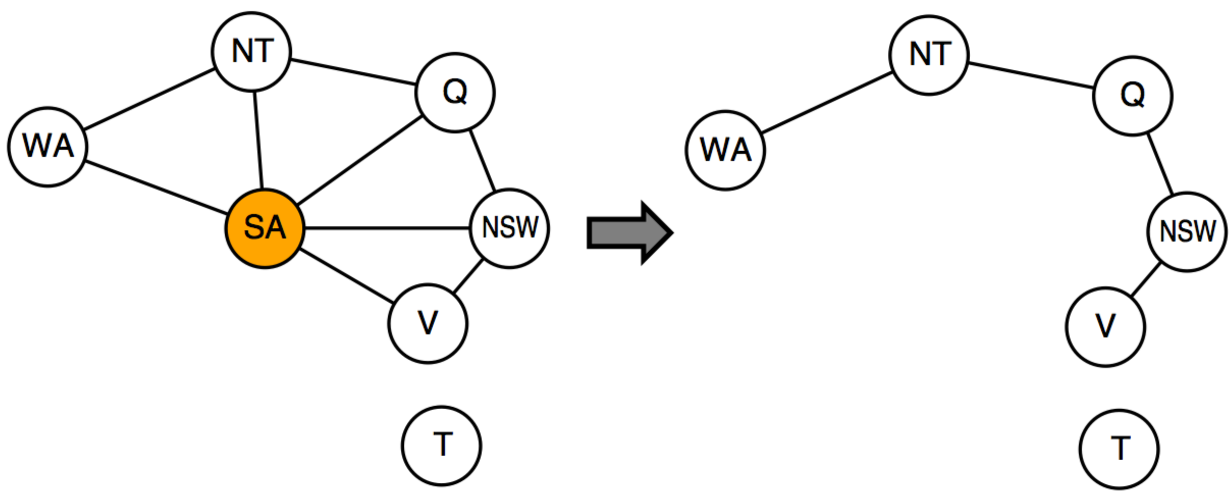
Algorithm for tree-structured CSPs

- Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- For j from n down to 2, apply REMOVEINCONSISTENT($Parent(X_j)$, X_j)
- 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
- Cutset size $c \Rightarrow$
  - runtime $O(d^c \cdot (n - c)\, d^2)$, very fast for small c



Iterative 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: 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