Constraint Satisfaction Problems

Outline

- ♢ CSP examples
- ♢ Backtracking search for CSPs
- ♢ Problem structure and problem decomposition
- ♢ Local search for CSPs

Constraint satisfaction problems (CSPs)

Standard search problem:

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

CSP: states and goal test conform a standard, structured and simple representation.

state is defined by variables X_i with values from domain D_i

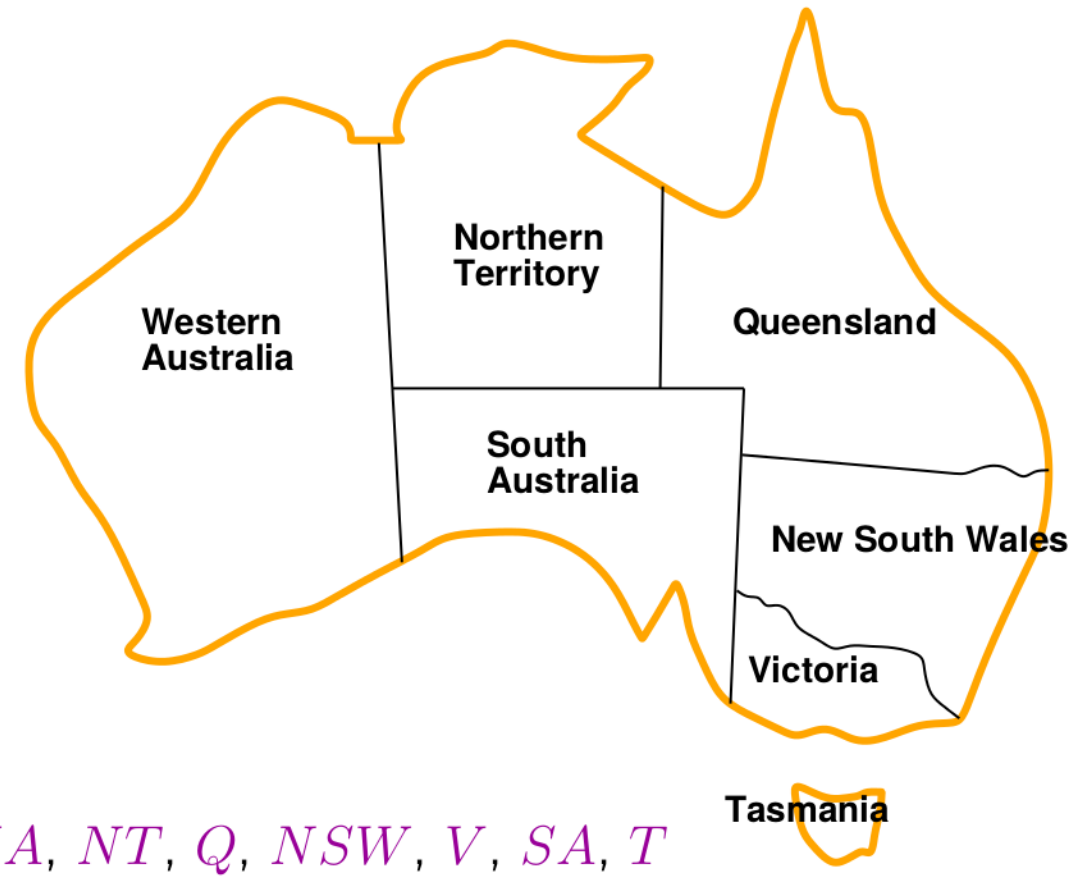
goal test is a set of constraints specifying
allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power than standard search algorithms

- state: assignment of variables {Xi=a, Xj=b..}
- assignment is consistent or legal if not violates constraints
- solution: a complete assignment that satisfies all constraints
- some CFPs require soln that maximize objective function

Example: Map-Coloring



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

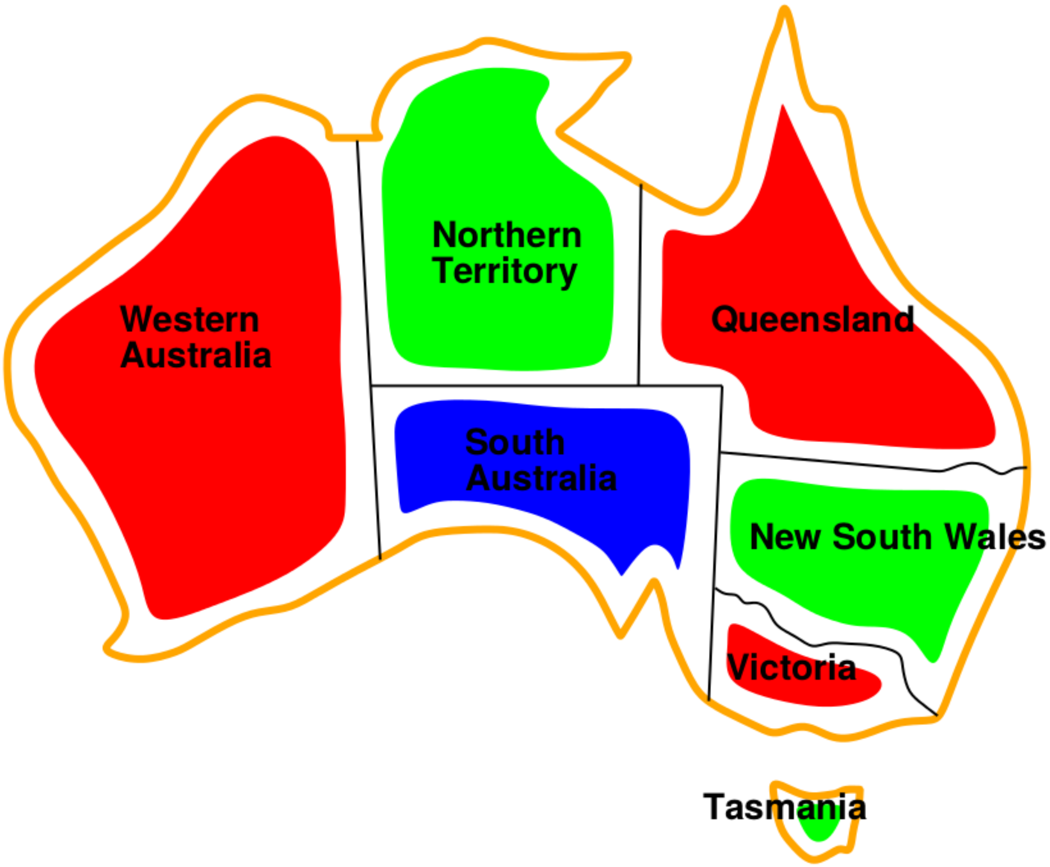
Domains $D_i = \{red, green, blue\}$

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.,

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

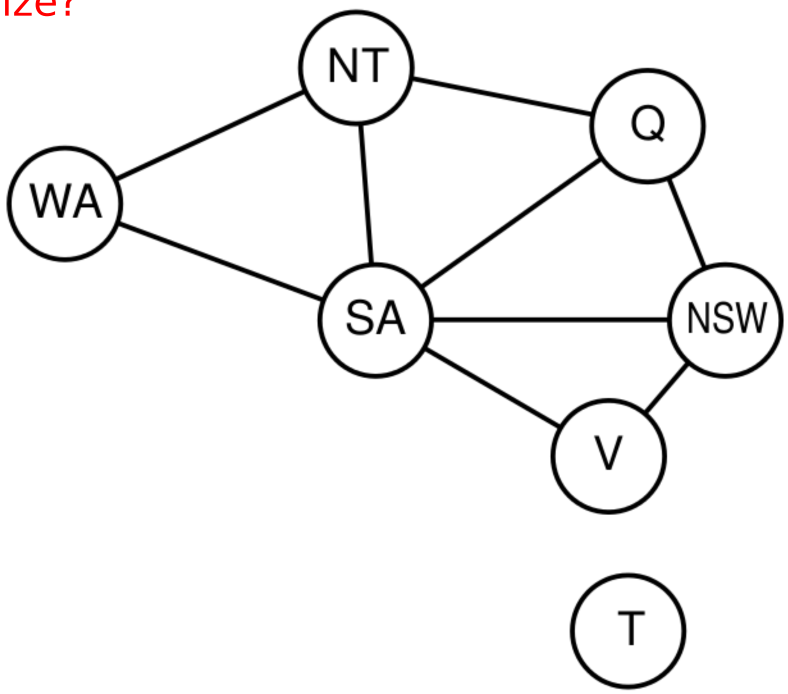
There are different solutions.

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints

Maybe easier to visualize?



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

Start from initial-state={}, assign a value in each step.

Varieties of CSPs

Discrete variables

e.g.? finite domains; size $d \Rightarrow O(d^n)$ complete assignments why? Depth-first-search is popular. Why?

- 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., $StartJob_1 + 5 \leq StartJob_3$
- linear constraints solvable, nonlinear undecidable

enumerating assignments not possible

Continuous variables

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

Varieties of constraints

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

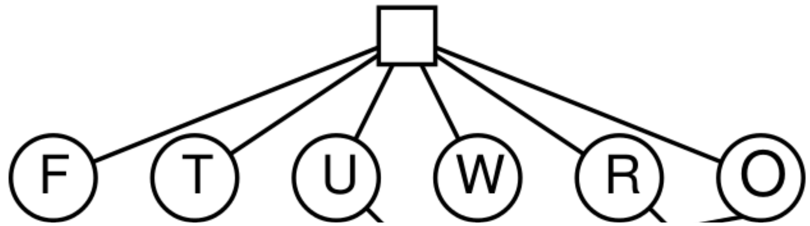
Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
e.g., cryptarithmetic column constraints

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

Example: Cryptarithmetic

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



Variables: $F\ T\ U\ W\ R\ O$

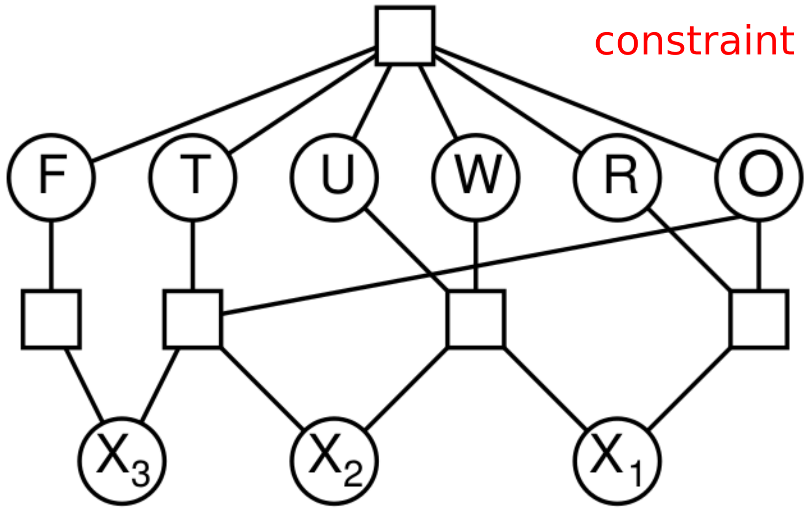
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$

Constraints

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

Example: Cryptarithmetic

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



constraint hypergraph

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)

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

Real-world CSPs

Assignment problems
e.g., who teaches what class

Timetabling problems
e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Absolute vs. preference constraints

Transportation scheduling

Factory scheduling

Floorplanning

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

Standard search formulation (incremental)

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

1) This is the same for all CSPs! 😆
2) Every solution appears at depth n with n variables
$\Rightarrow$ use depth-first search   d values
3) Path is irrelevant, so can also use complete-state formulation
4) How many leaves?

Standard search formulation (incremental)

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

1) This is the same for all CSPs!
2) Every solution appears at depth n 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!!!!

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

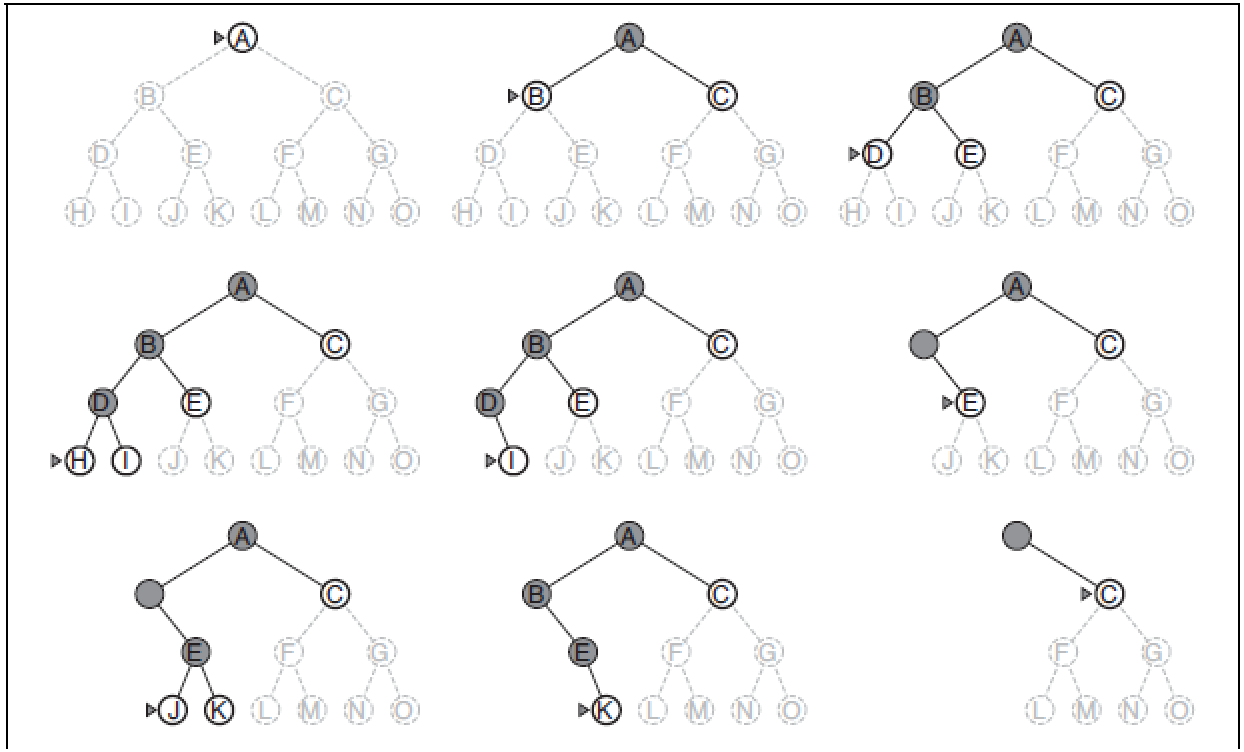
Depth-first search for CSPs with single-variable assignments
is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

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

Depth-first-search: Backtracking

- A variant of depth-first search called backtracking BACKTRACKING search uses still less memory.
- Only one successor is generated at a time rather than all successors;
- Each partially expanded node remembers which successor to generate next.
- In this way, only O(m) memory is needed rather than O(bm).



Backtracking search

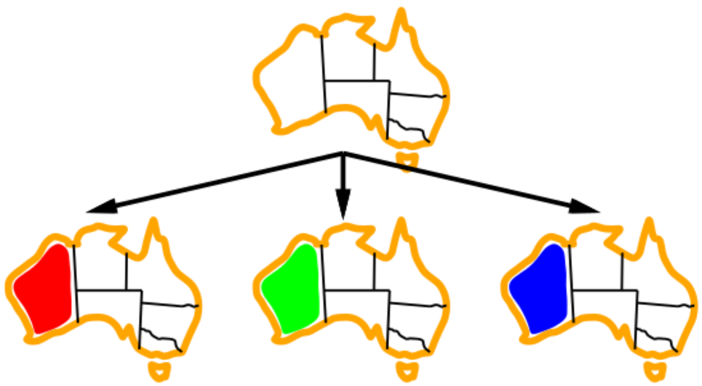
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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
```

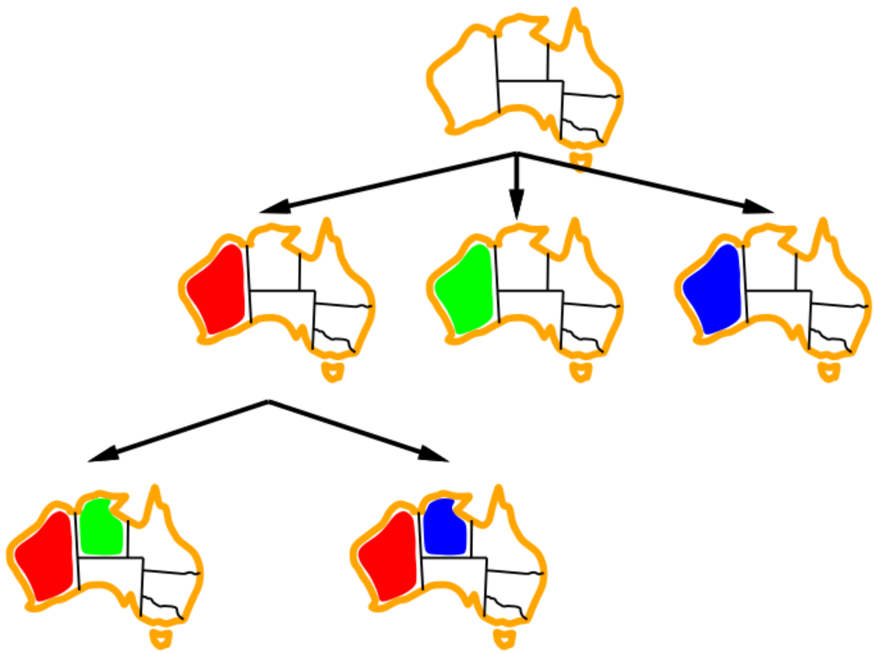
add backtracking search from pg. 76

Backtracking example

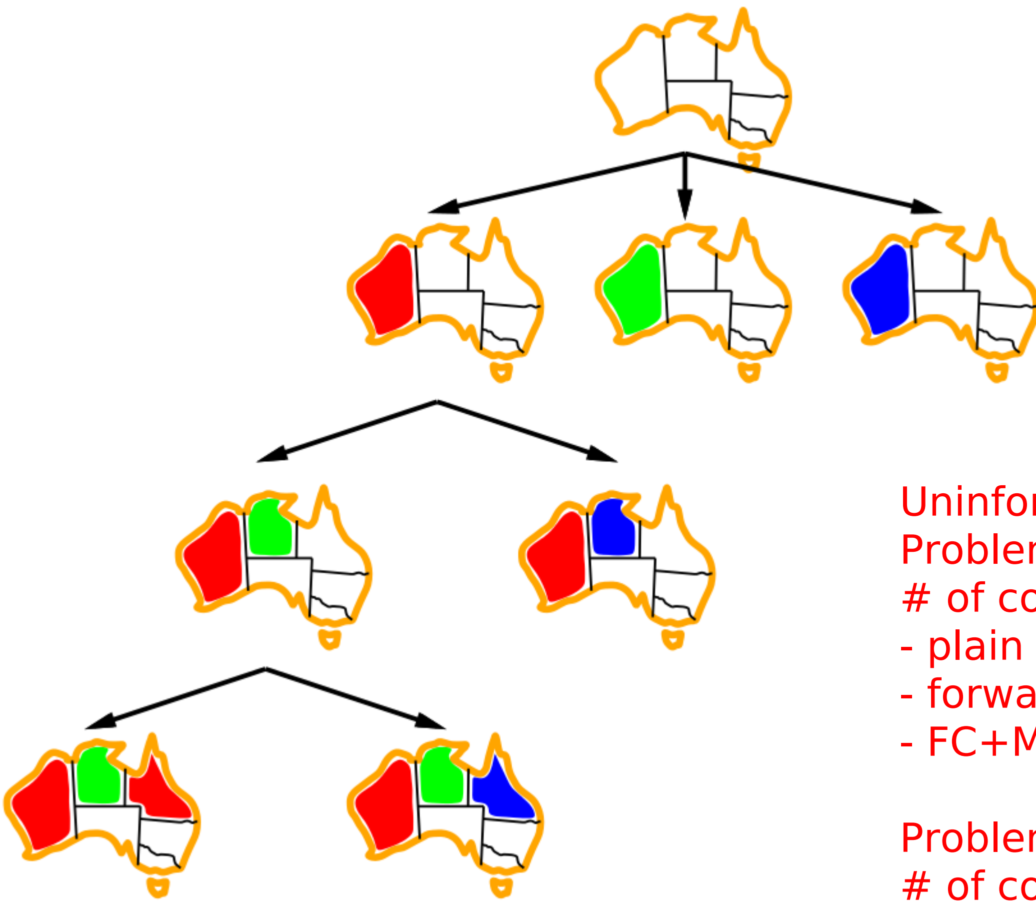
Backtracking example



Backtracking example



Backtracking example



Uninformed algorithm. No big expectations
Problem: 4-coloring of 50 USA states
# of consistency checks:
- plain backtracking: >1,000K
- forward checking: >1,000K
- FC+MRV: 60

Problem: n-queens
# of consistency checks:
- plain backtracking: >40,000K
- forward checking: >40,000K
- FC+MRV: 717K

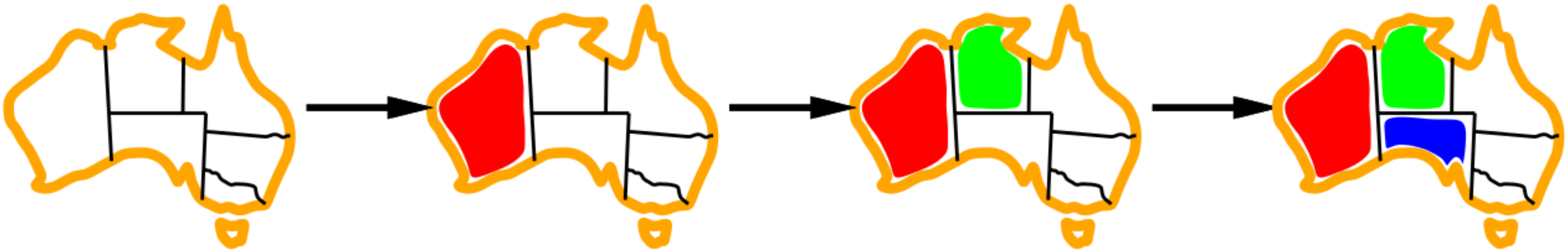
Improving backtracking efficiency

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

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



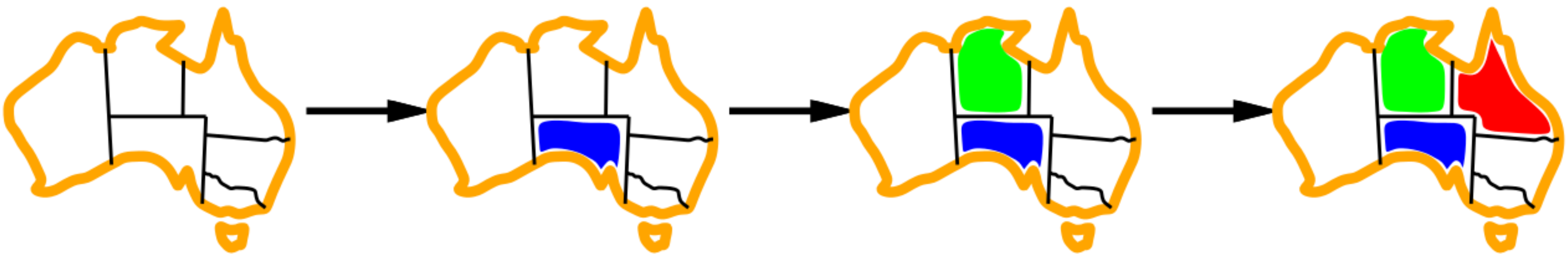
called "the most constrained variable"

Uninformed algorithm. No big expectations
Problem: n-queens
# of consistency checks:
- plain backtracking: >40,000K
- BT+MRV > 13,500K
- forward checking: >40,000K
- FC+MRV: 717K

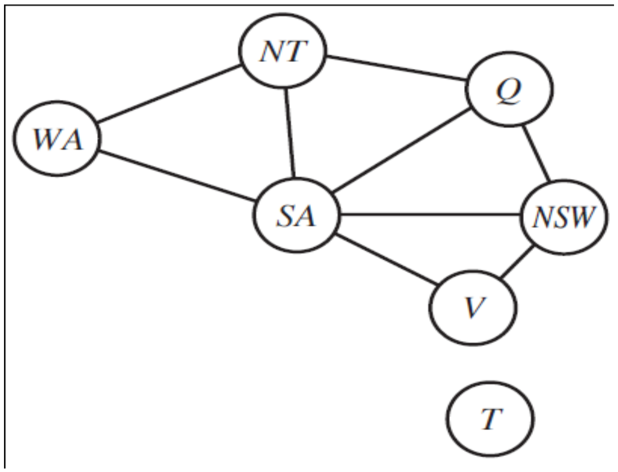
Degree heuristic

Tie-breaker among MRV variables

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



Attempt to reduce branching factor of future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

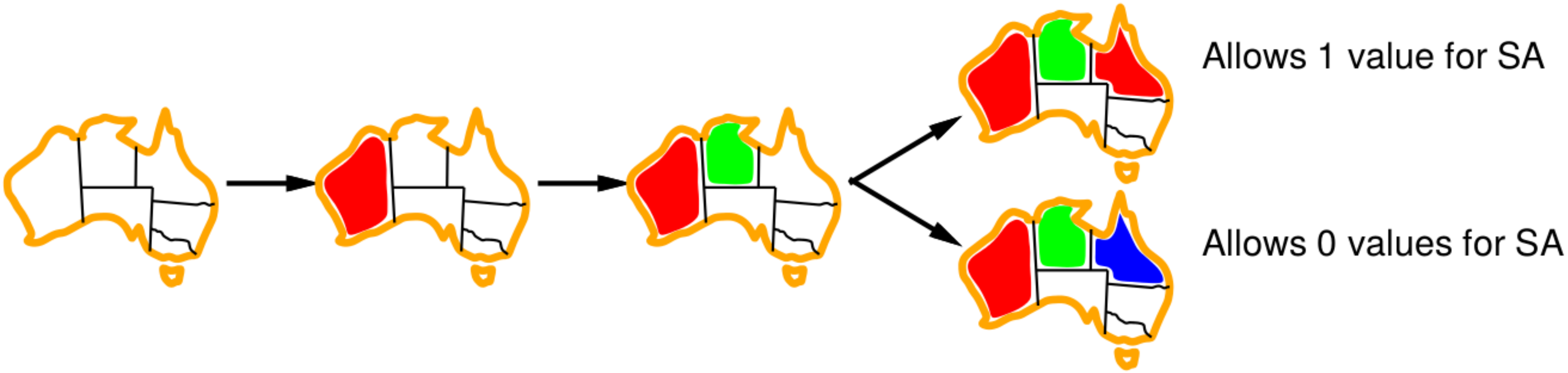


Least constraining value

Given a variable, choose the least constraining value:

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

i.e. leave the max flexibility for subsequent variable assignments.



Combining these heuristics makes 1000 queens feasible

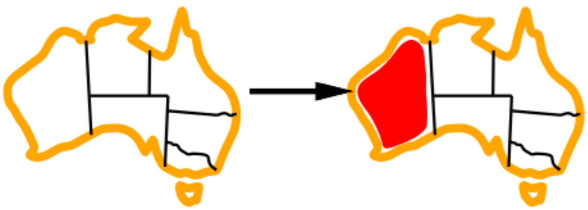
So far, we only considered the constraints on a variable only at a time

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

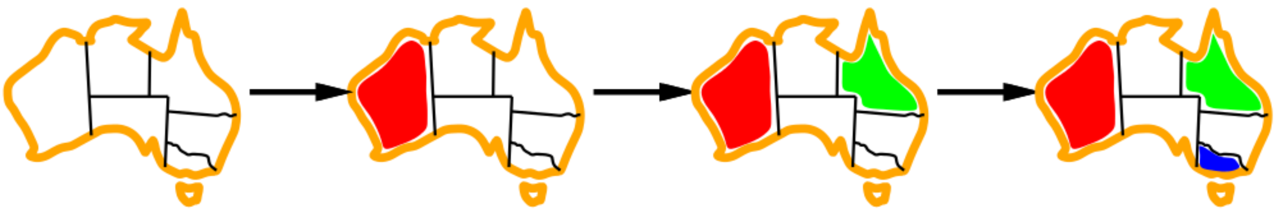


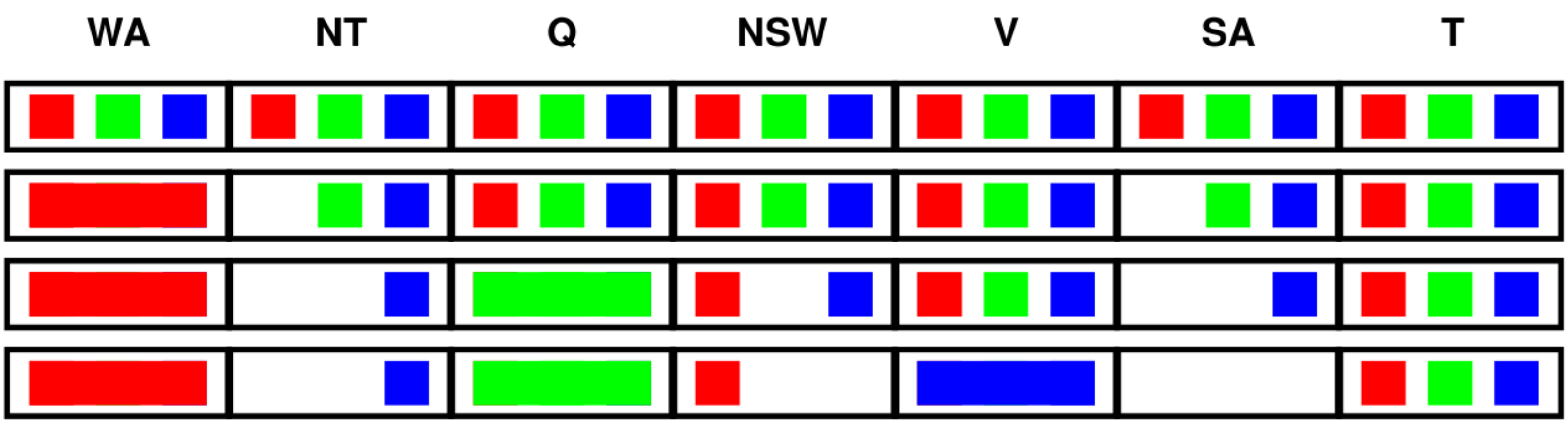
Forward checking

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

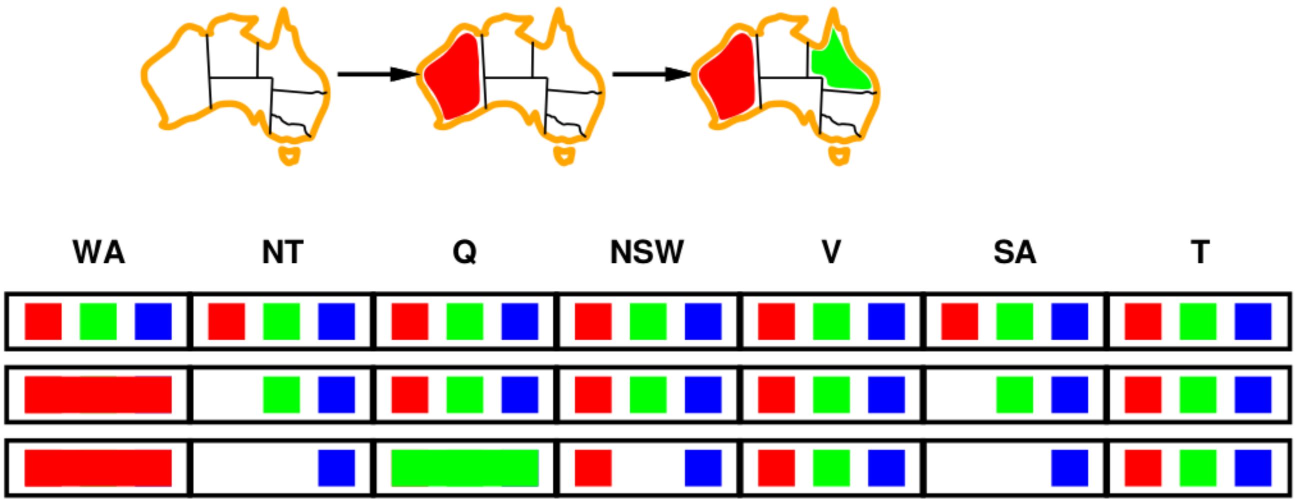




But probably we would select either NT or SA

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

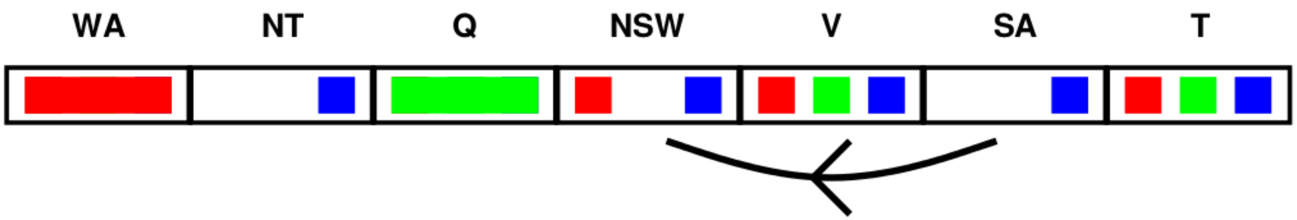
general term for propagating the implications of a constraint on one variable onto other variables.

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

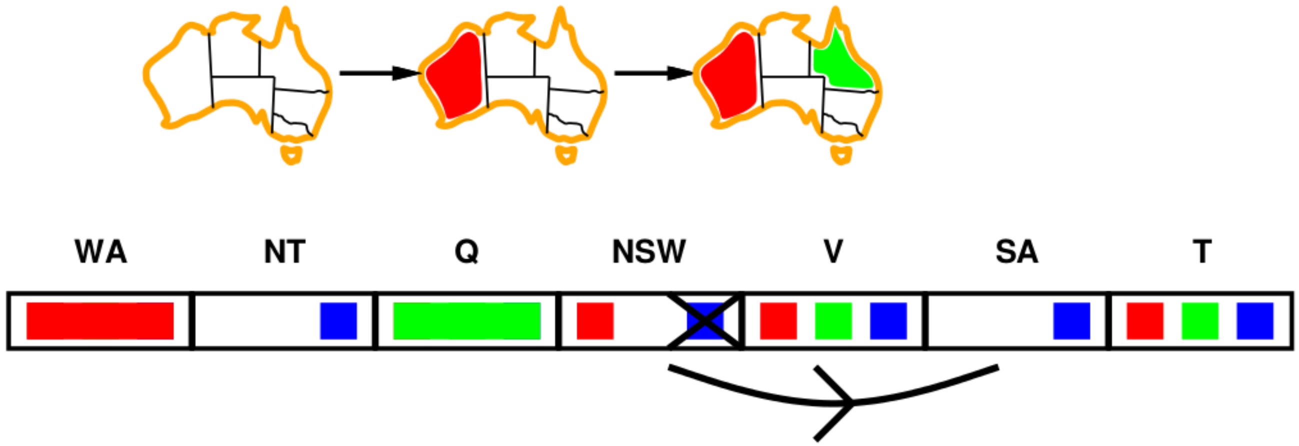


Arc consistency

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

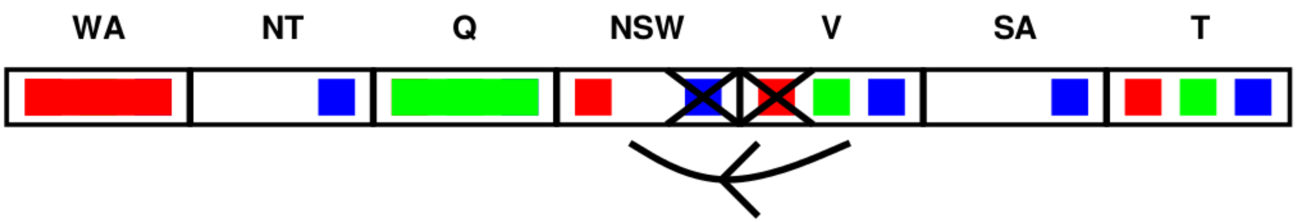


Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

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



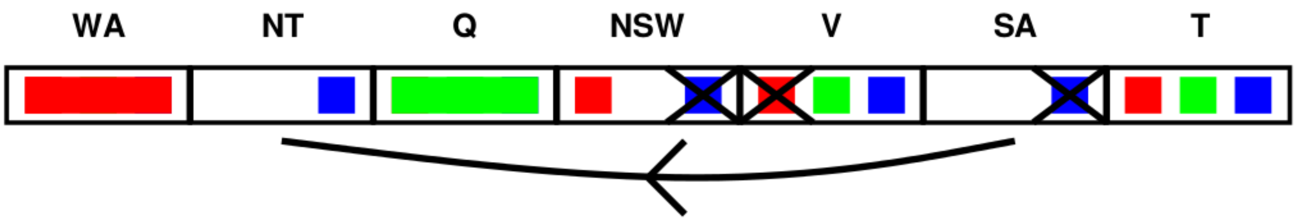
If X loses a value, neighbors of X need to be rechecked

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

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

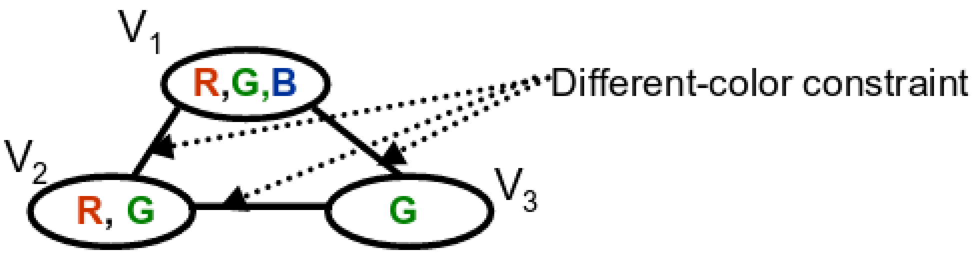
$O(n^2d^3)$ (but detecting **all** is NP-hard)

^ compute!

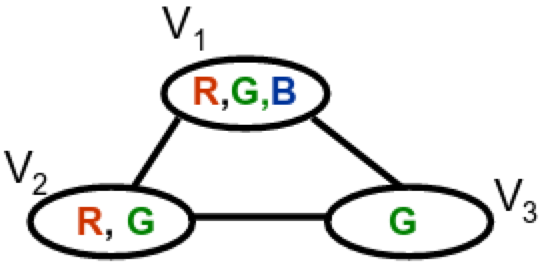
Constraint Propagation Example

Graph Coloring

Initial Domains are indicated



Arc examined	Value deleted



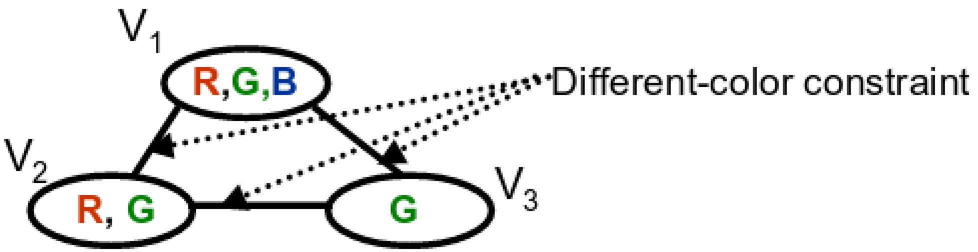
Each undirected constraint arc is really two directed constraint arcs, the effects shown above are from examining BOTH arcs.

http://web.mit.edu/6.034/wwwbob/constraint.pdf

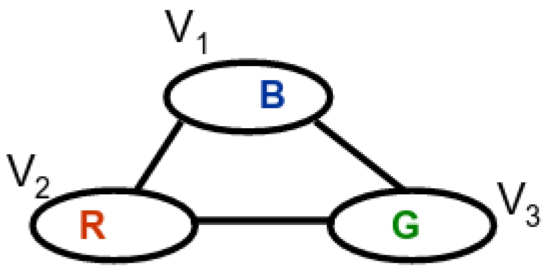
Constraint Propagation Example

Graph Coloring

Initial Domains are indicated



Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	V_1(G)
$V_2 - V_3$	V_2(G)
$V_1 - V_2$	V_1(R)
$V_1 - V_3$	none
$V_2 - V_3$	none

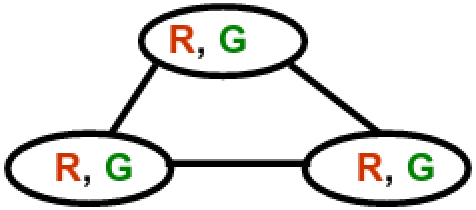


http://web.mit.edu/6.034/wwwbob/constraint.pdf

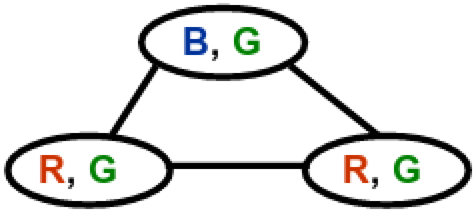
Constraint Propagation Example

But, arc consistency is not enough in general

Graph Coloring

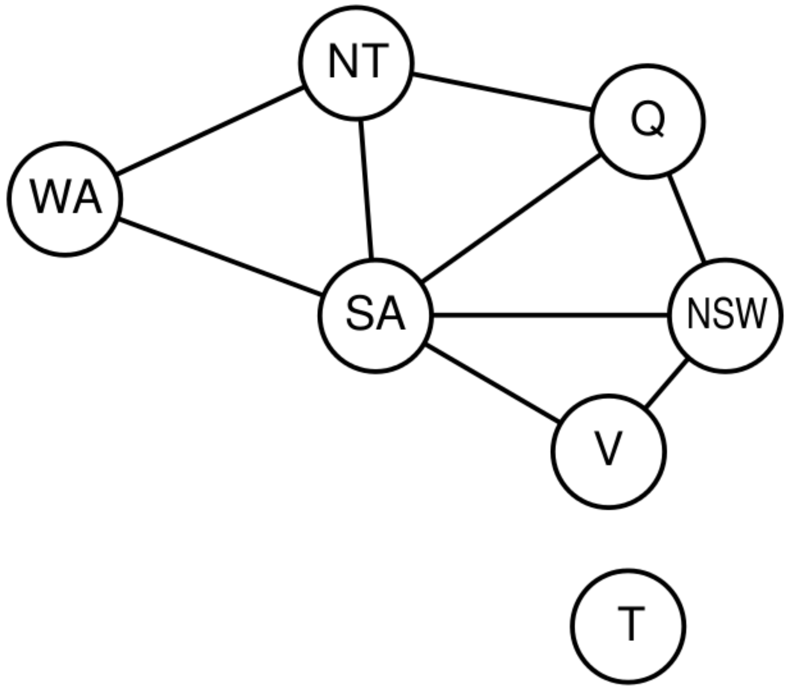


arc consistent but
no solutions



arc consistent but 2
solutions B,R,G ;
B,G,R .

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

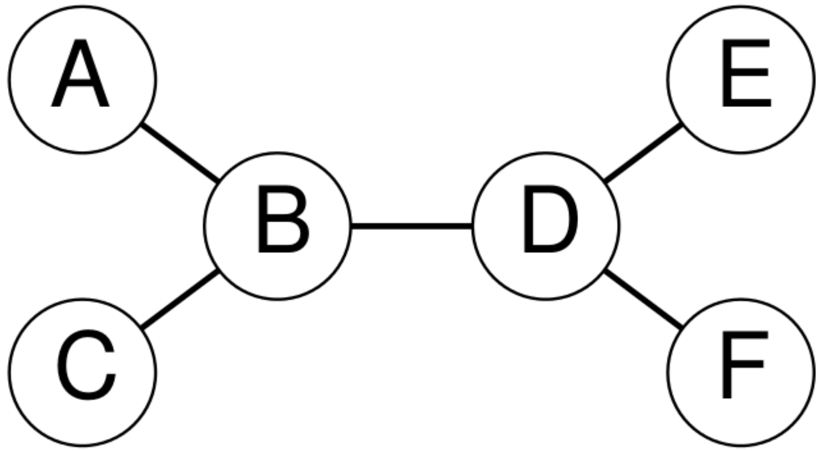
Problem structure contd.

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



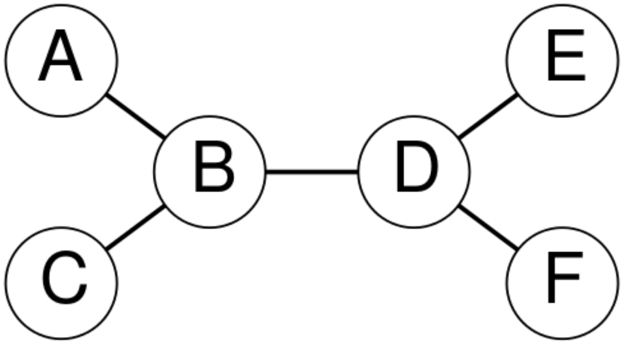
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\, d^2)$ time

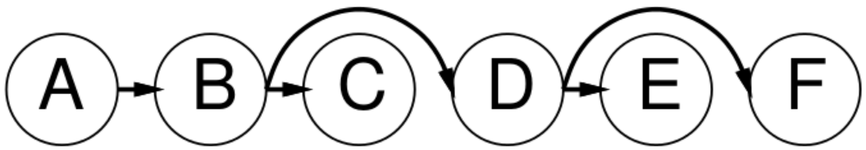
Compare to general CSPs, where worst-case time is $O(d^n)$

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





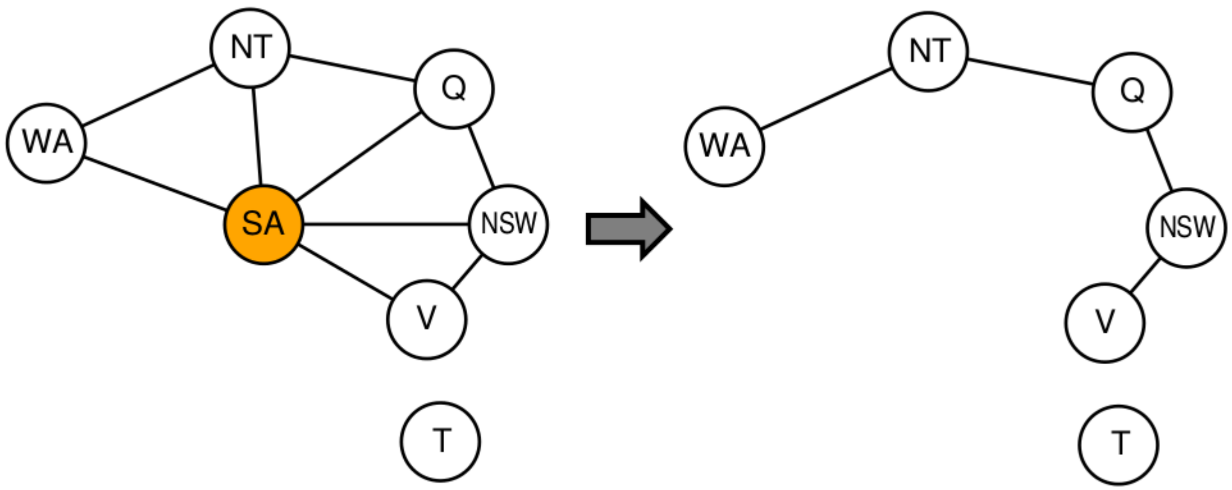
2. For j from n down to 2, apply RemoveInconsistent$(Parent(X_j), X_j)$

why do we remove in backwards order?

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



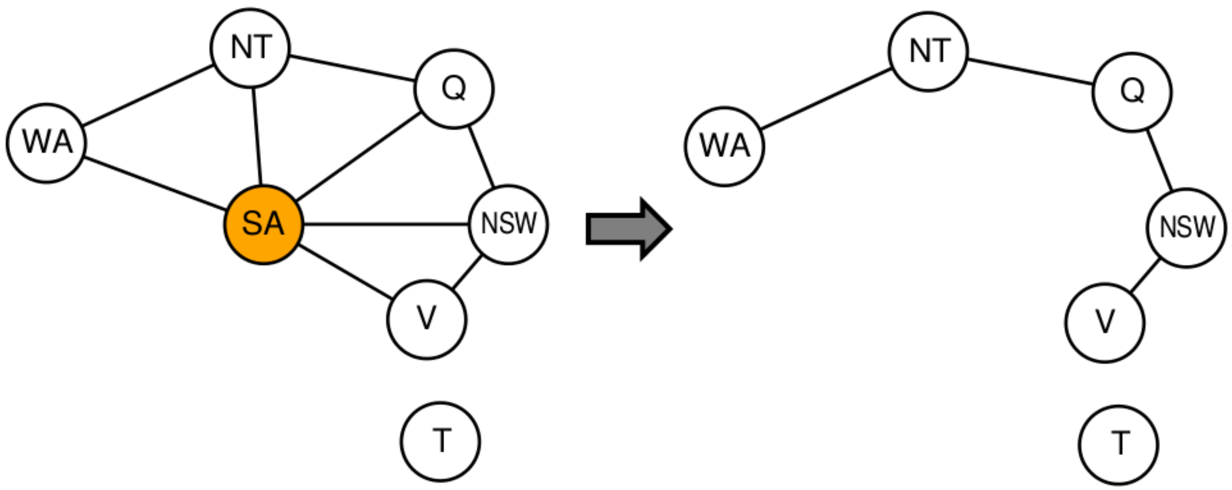
Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c $\Rightarrow$ runtime $O($???? $)$, very fast for small c

1. Choose a subset S of variables such that constraint graph becomes tree.
2. For each possible assignment of S that satisfies all constraints on S
  (a) remove from domains of the remaining variables that are inconsistent with assign. of S
  (b) if remaining CSP has a solution..

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

1. Choose a subset S of variables such that constraint graph becomes tree.
2. For each possible assignment of S that satisfies all constraints on S
   (a) remove from domains of the remaining variables that are inconsistent with assign. of S
   (b) if remaining CSP has a solution..

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

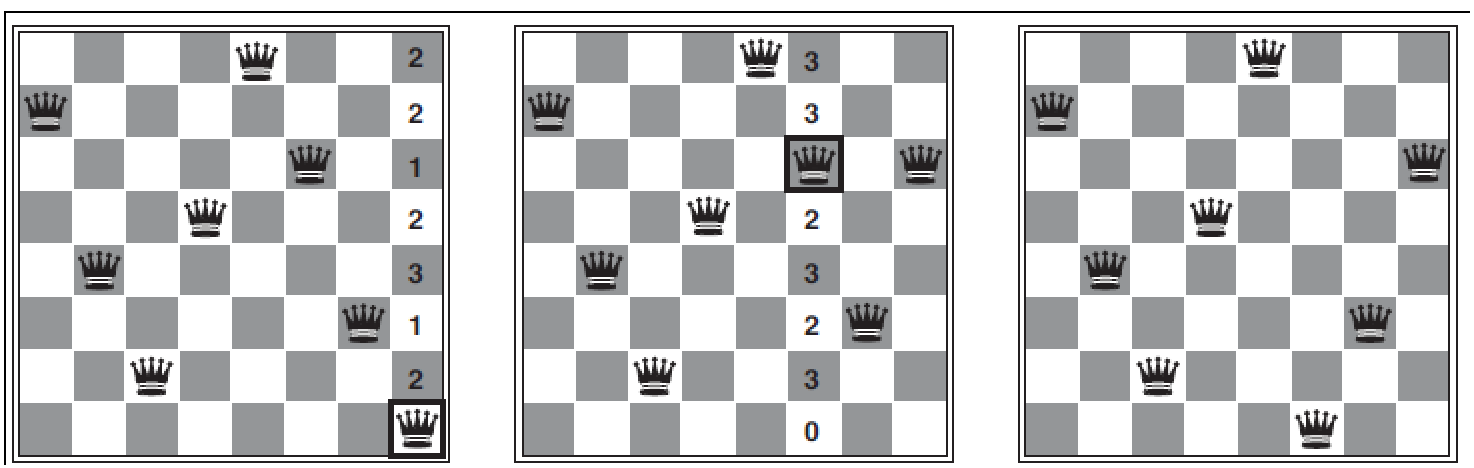
Value selection by min-conflicts heuristic:
choose value that violates the fewest constraints
i.e., hillclimb with $h(n)$ = total number of violated constraints

Local search for CSP

- Min-conflicts heuristic: select the value that results in min number of conflicts with other variables. Surprisingly effective

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          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(var, v, current, csp)
     set var = value in current
  return failure
```



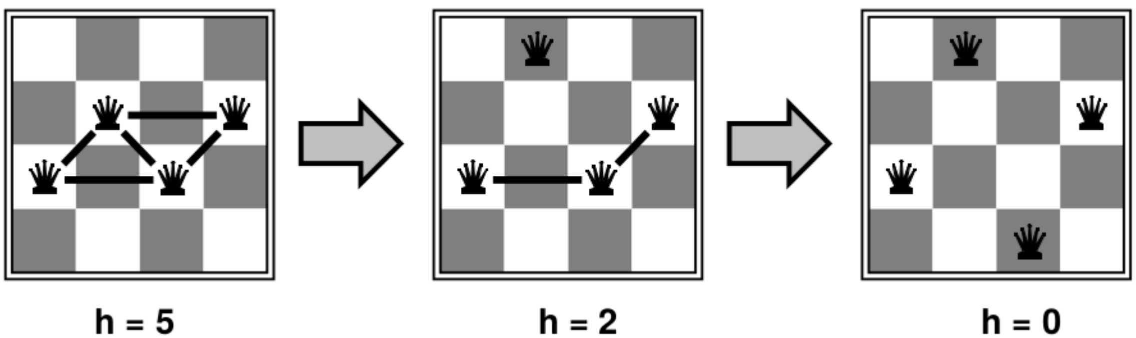
Example: 4-Queens

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



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

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice