Constraint Satisfaction Problems

Chapter 6.1-3

Adapted from slides kindly shared by Stuart Russell

Announcements

Revised deadline for Python Tutorial (P0): This Wednesday at 17:00.

Submit Project 0 via D2L Dropbox while we work on real-time autograding

Project 1 (search) assigned, due Thu 9-27 at 5pm (change from syllabus)

Office hours for me on Thursday 2:30-3:30, ECST 121

Outline

◇ Constraint Satisfaction Problems (CSP)

◇ Backtracking search for CSPs

Search: Planning vs Identification

We've focussed on using search for (planning) so far:
planning a path to a goal

Can also be used for just (identifying) a goal,
viewed as assignments to variables

CSPs are specialized for identification problems

Constraint satisfaction problems (CSPs)

Standard search problem:

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

CSP:

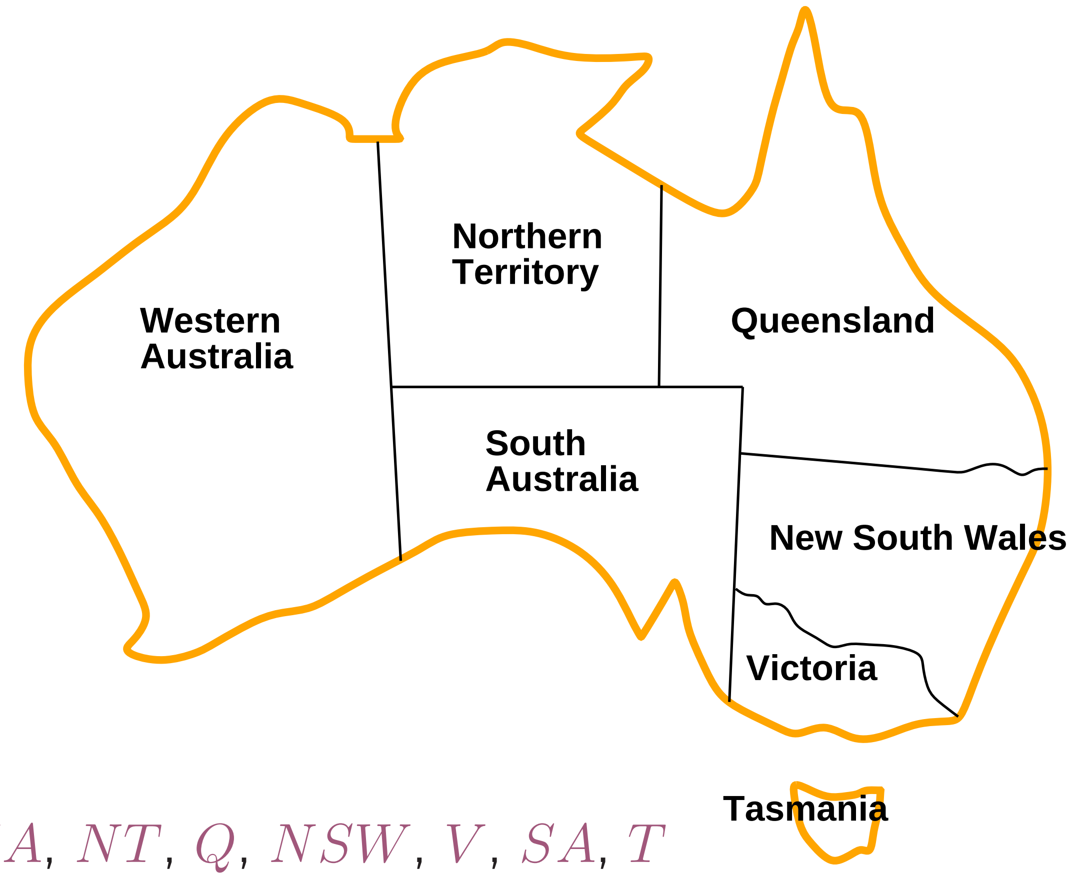
state is defined by variables X_i with values from domain D_i

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

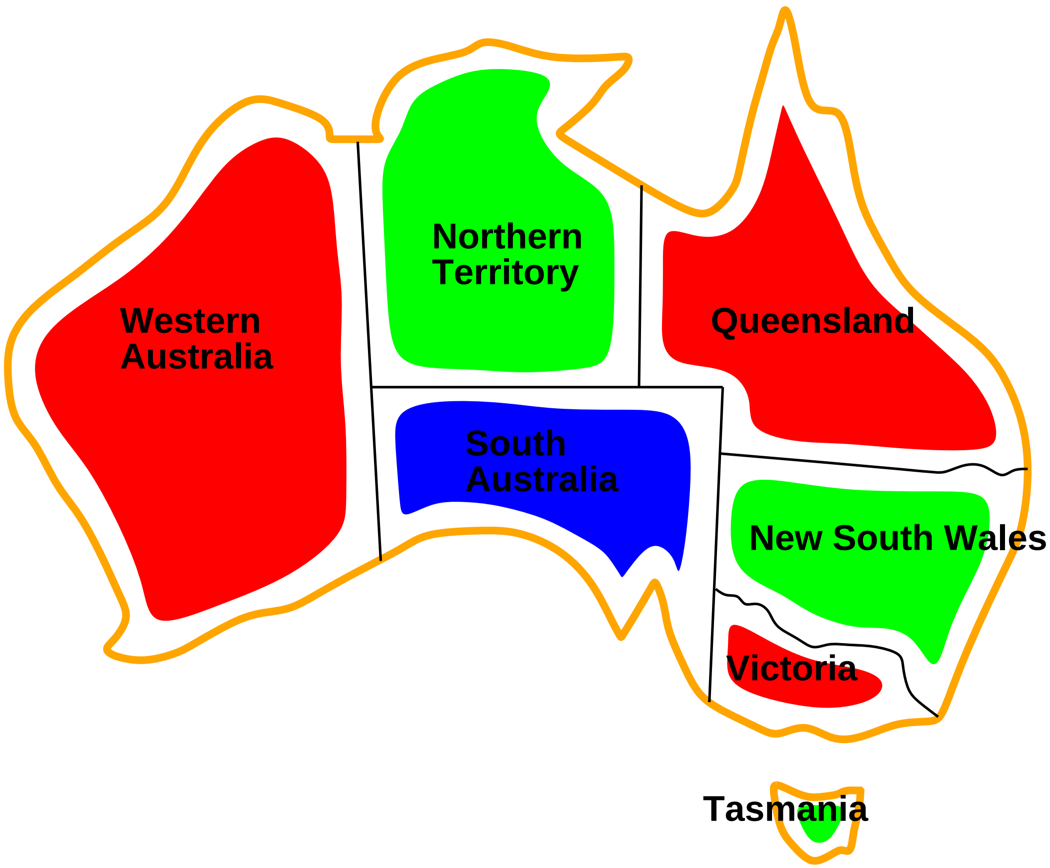
Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Example: Map-Coloring contd.

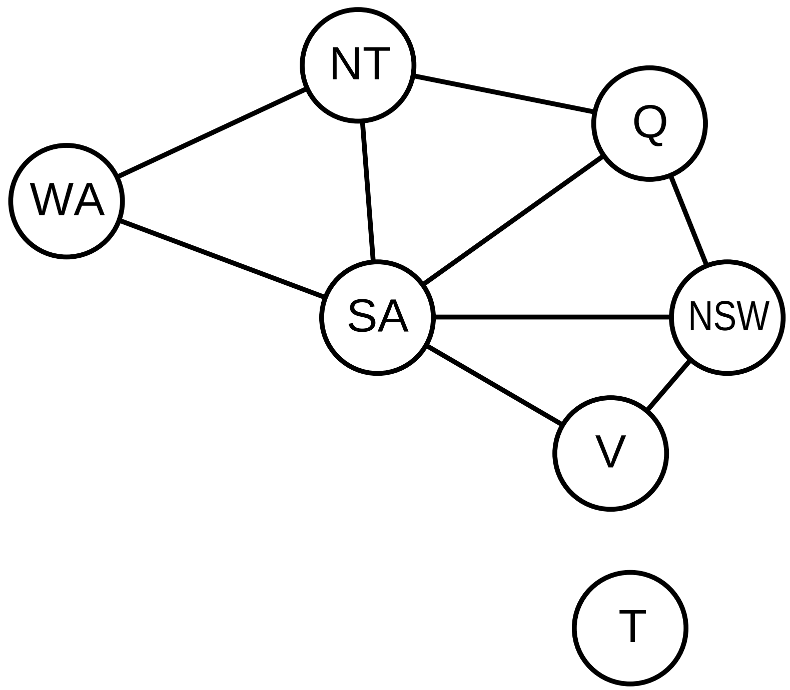


Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

Binary CSP: each constraint relates at most two variables

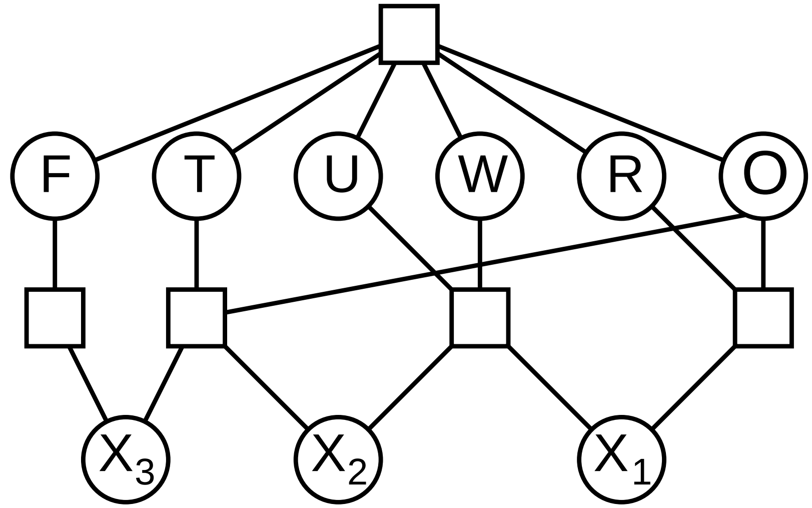
Constraint graph: nodes are variables, arcs show constraints



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

Example: Cryptarithmetic

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



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

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

Constraints

alldiff(F, T, U, W, R, O)

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

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., 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 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

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

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.
⇒ 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
⇒ 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!!!!

where there are n variables, d possible values per variable

Search Methods

Consider Simplest CSP ever: two bits, constrained to be equal

Consider Australian map

What would BFS do?

What would DFS do?

What problems does this approach have?

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

Check for conflicts as you make variable assignments

"incremental goal test"

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

Backtracking search is the basic uninformed algorithm for CSPs

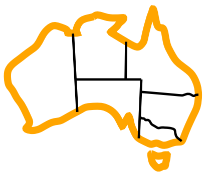
Can solve n-queens for $n \approx 25$

Backtracking search

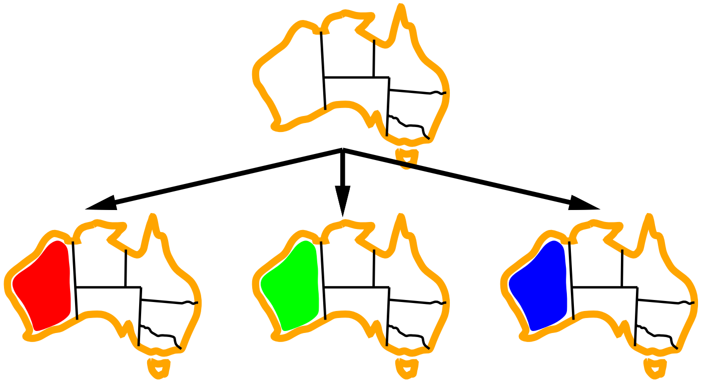
```
function Backtracking-Search(csp) returns solution/failure
   return Recursive-Backtracking({ }, csp)

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
```

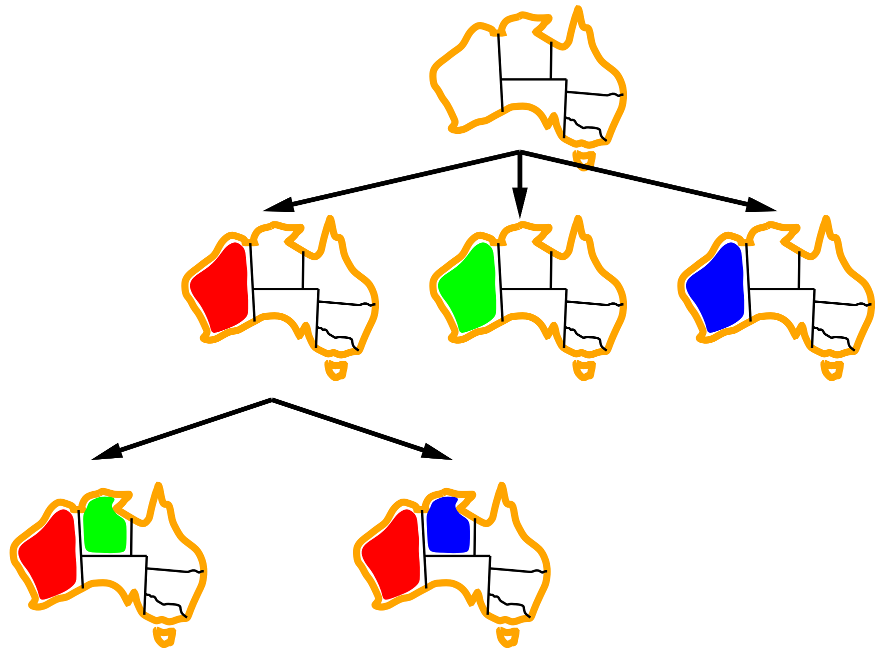
Backtracking example



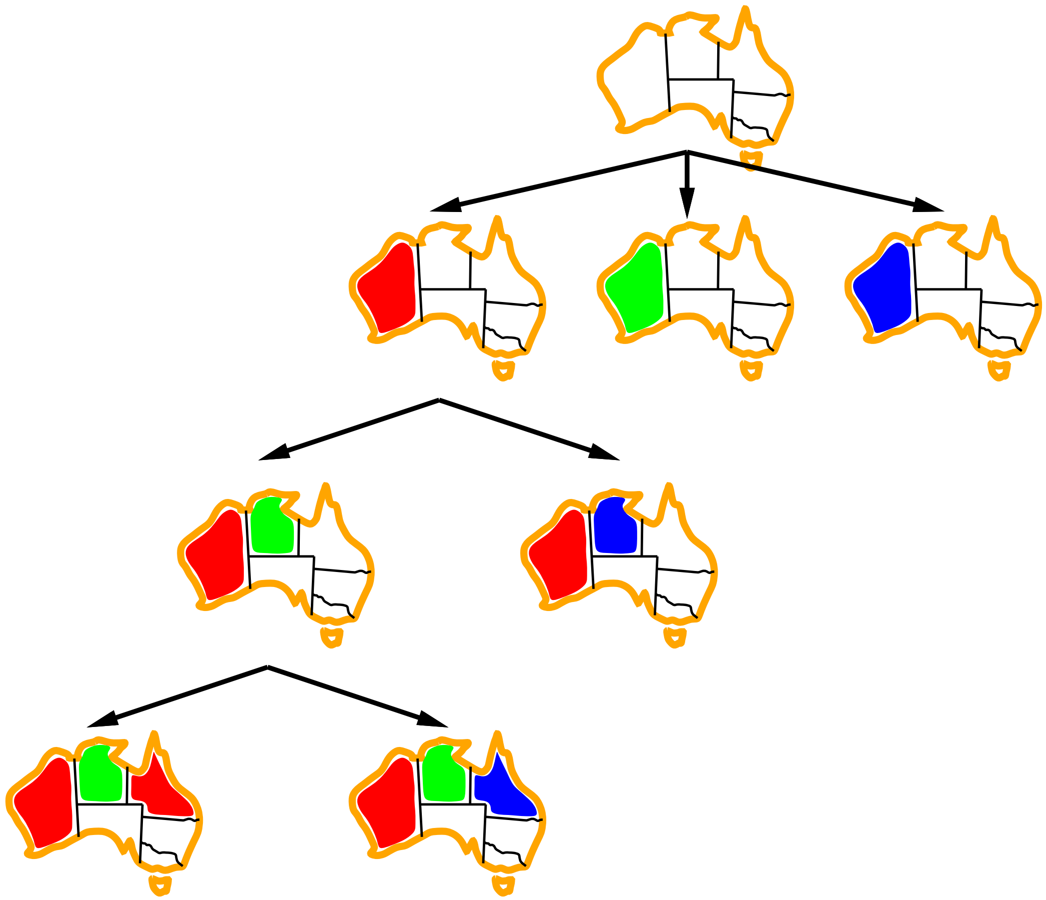
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

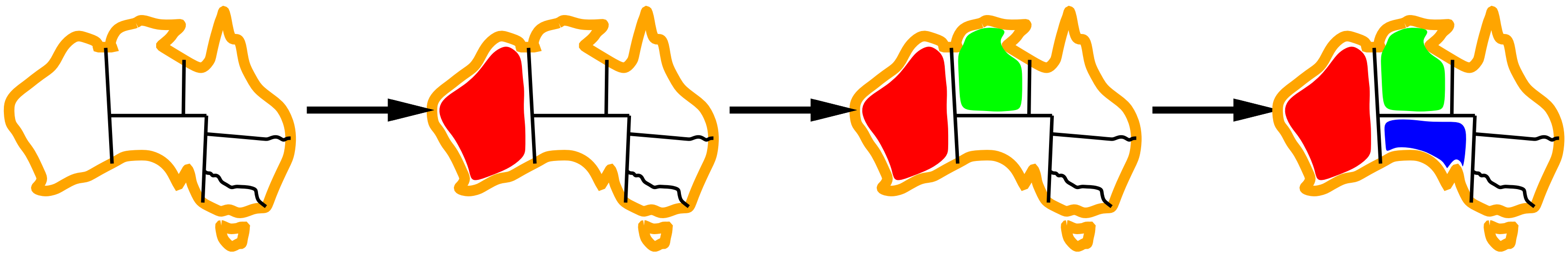
1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

Minimum remaining values (MRV):

choose the variable with the fewest legal values

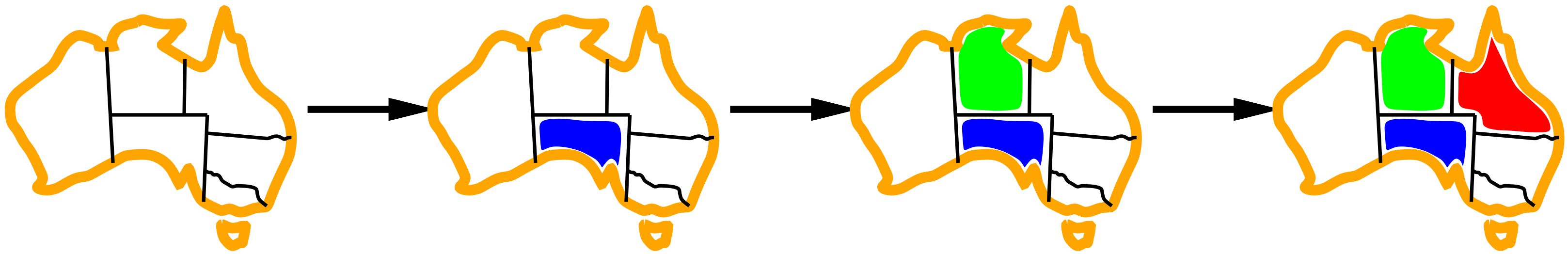
AKA "most constrained variable" or "fail-fast" ordering



Degree heuristic

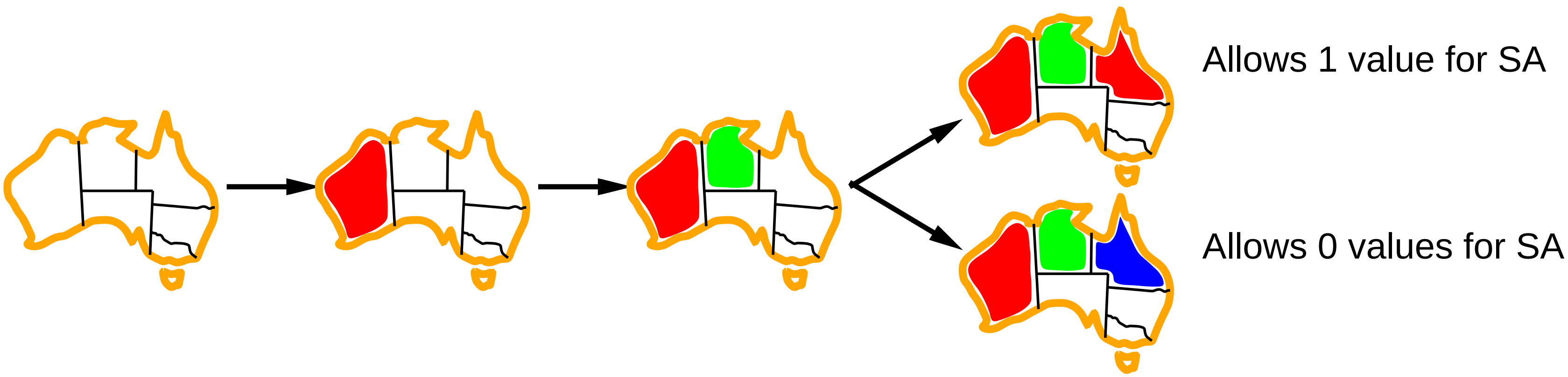
Tie-breaker among MRV variables

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



Least constraining value

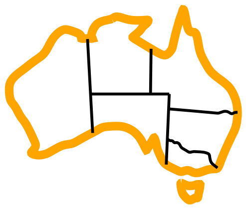
Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible

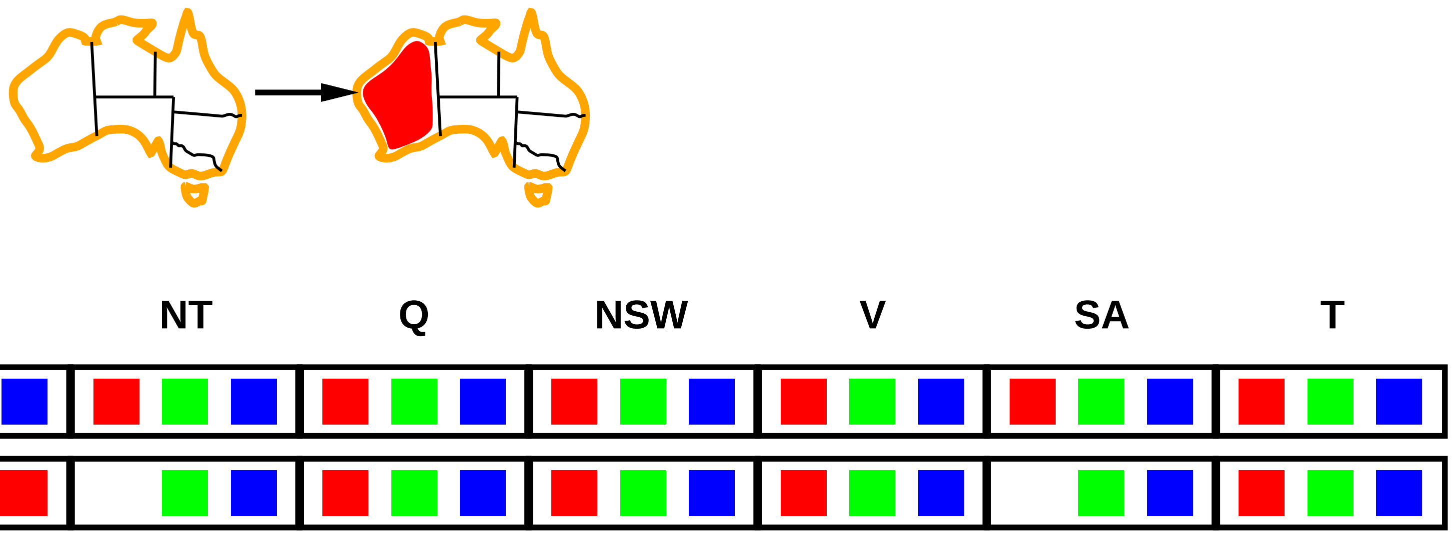
Filtering: Forward checking

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



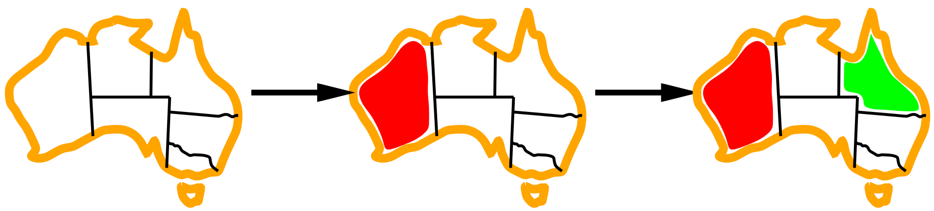
Forward checking

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



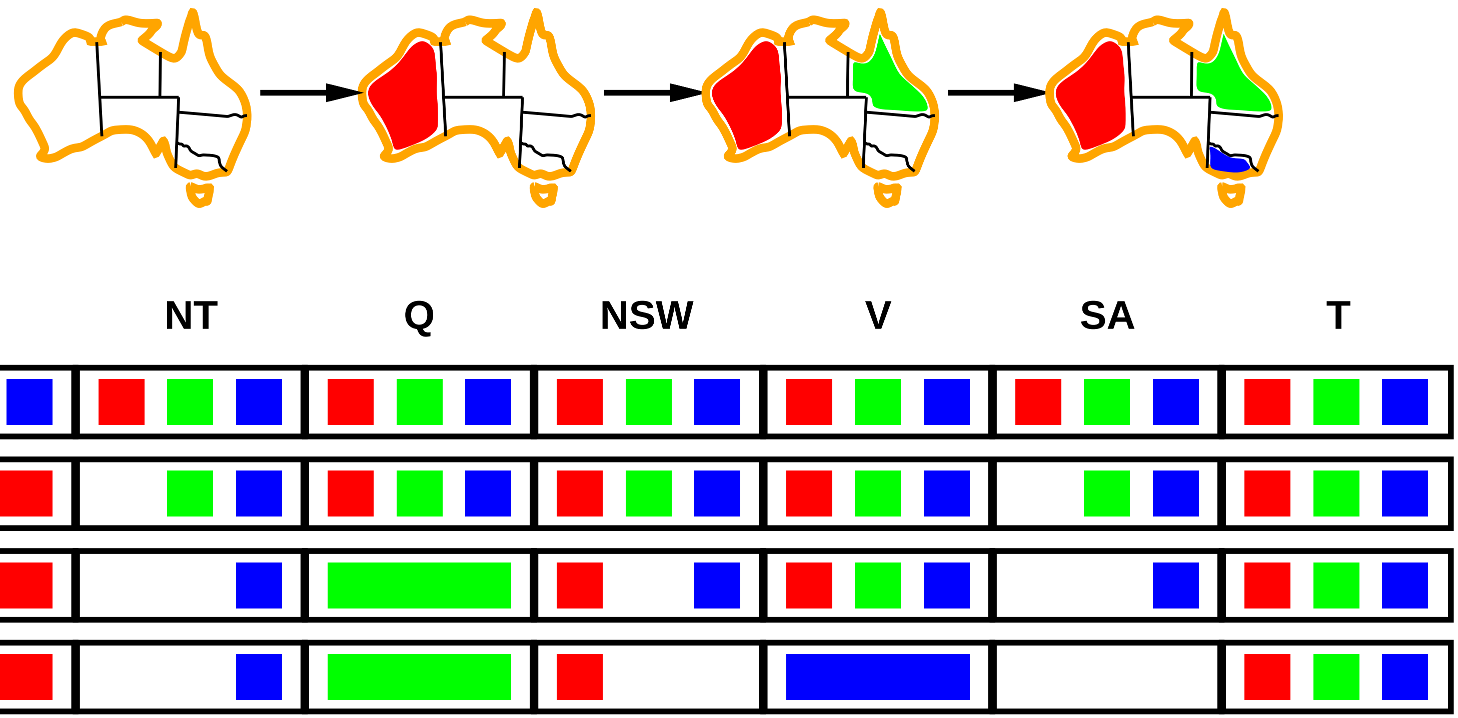
Forward checking

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



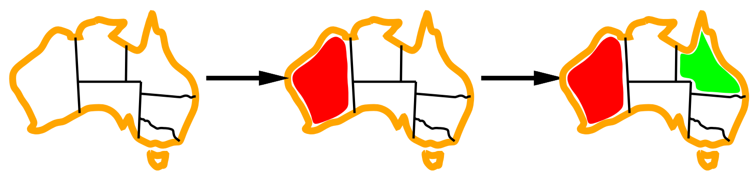
Forward checking

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



Filtering: 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!

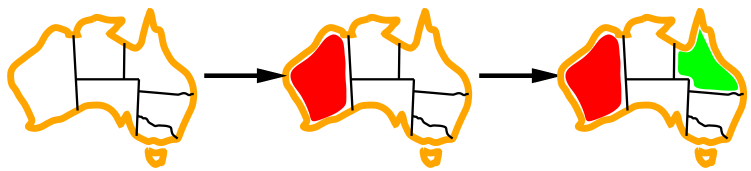
Constraint propagation repeatedly enforces constraints locally

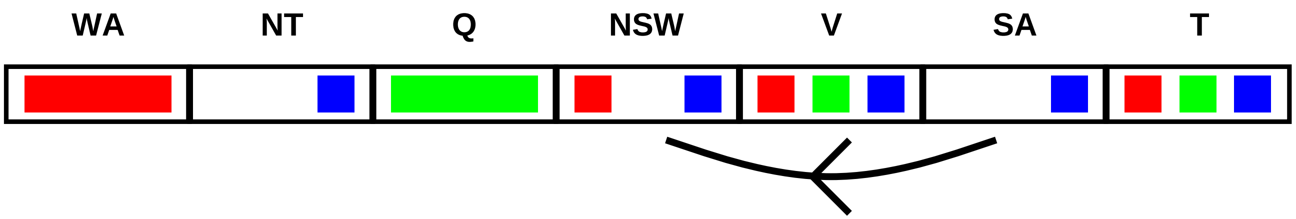
Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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



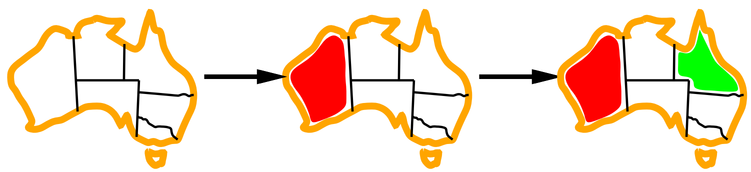


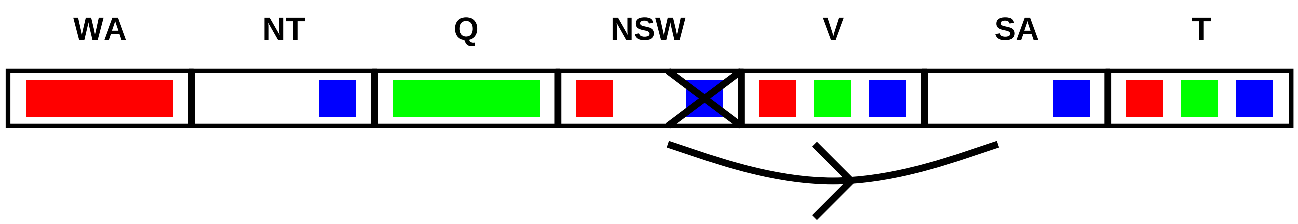
Arc consistency

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



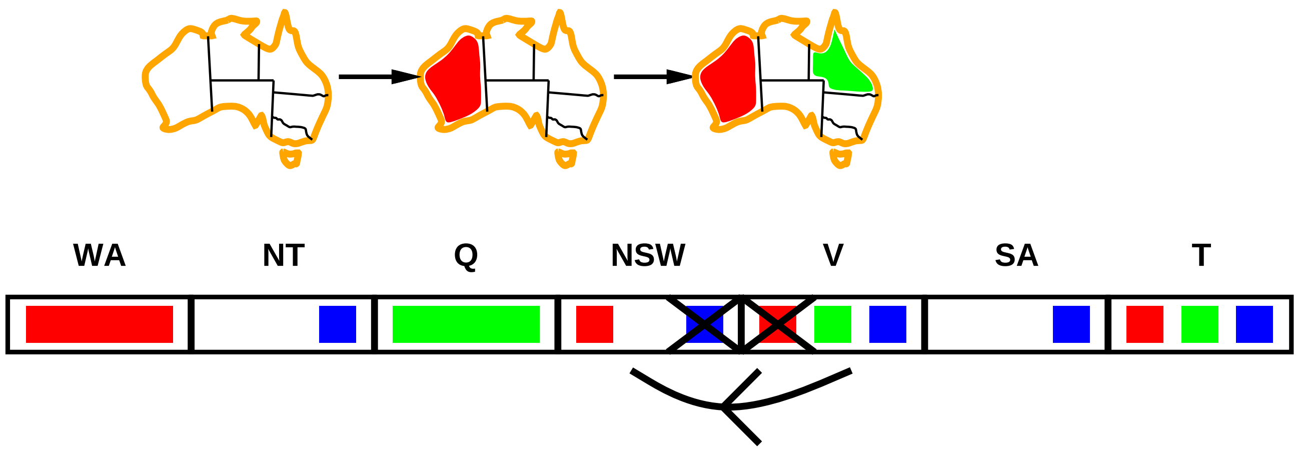


Arc consistency

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

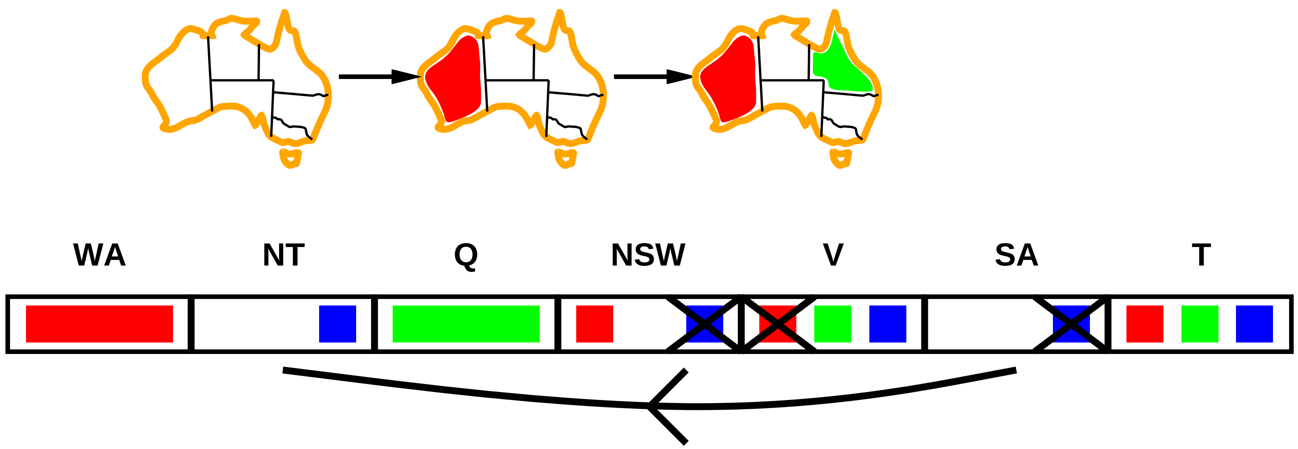


If X loses a value, neighbors of X need to be rechecked

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 ← *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* ← *true*
  return *removed*

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting **all** is NP-hard)

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 legally 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