Today

See Russell and Norvig, chapter 5

- Constraint satisfaction problems (CSPs)
- Heuristics for CSPs
- Constraint propagation
- Local search for CSPs

Reminder: Constraint satisfaction problems

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

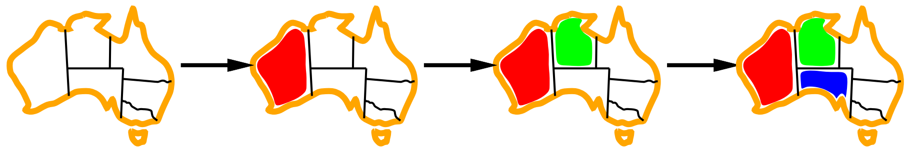
Improving backtracking efficiency

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?

Most constrained variable

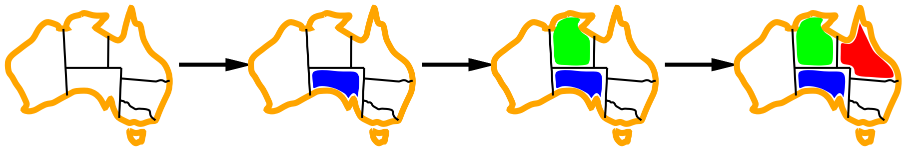
Most constrained variable:
choose the variable with the fewest legal values



Most constraining variable

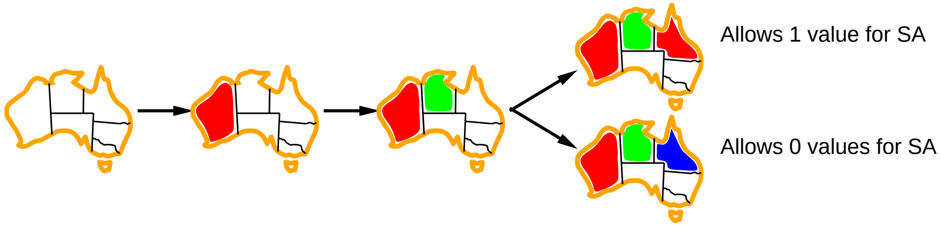
Tie-breaker among most constrained variables

Most constraining variable:
choose the variable with the most constraints on remaining variables



Least constraining value

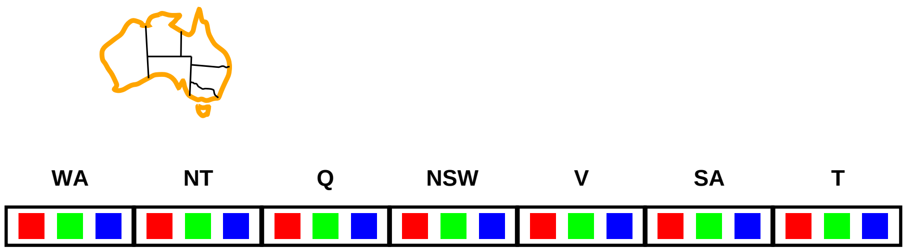
Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible;
recall that straight backtracking search can only deal with 25 queens!

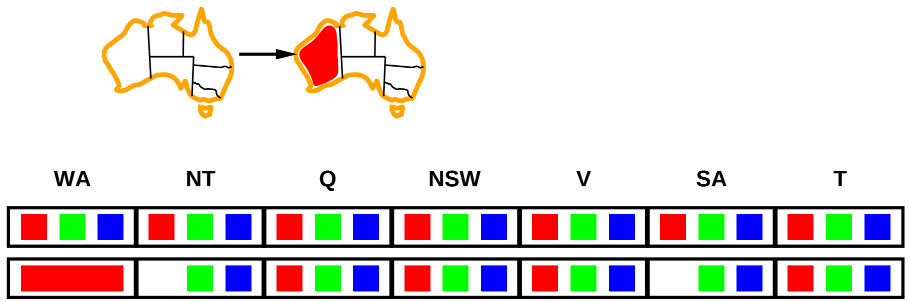
Forward checking

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



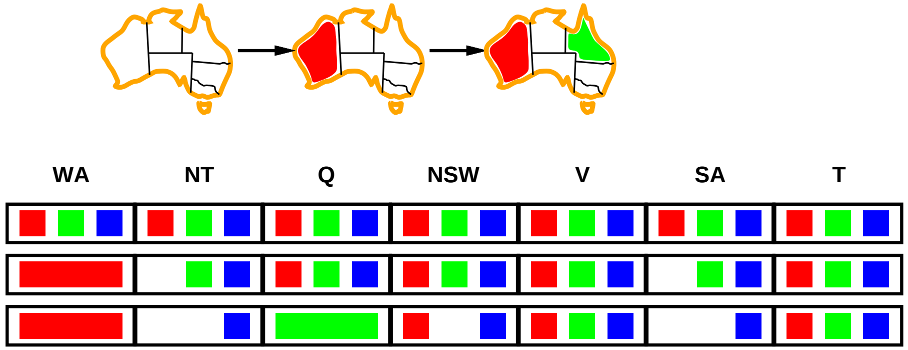
Forward checking

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



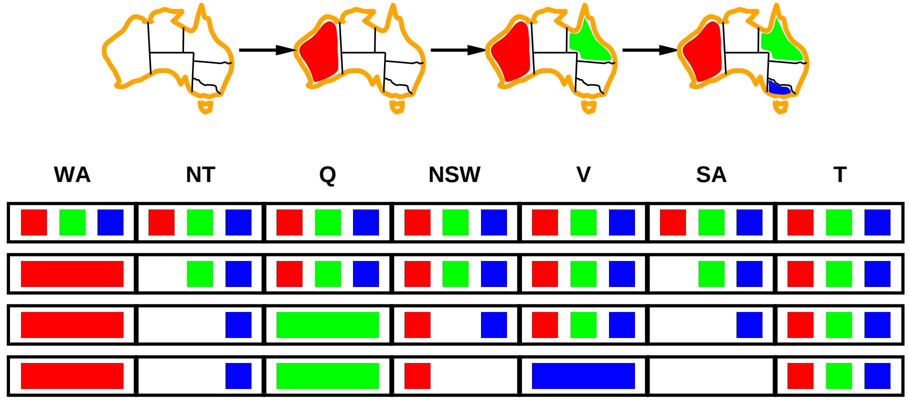
Forward checking

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



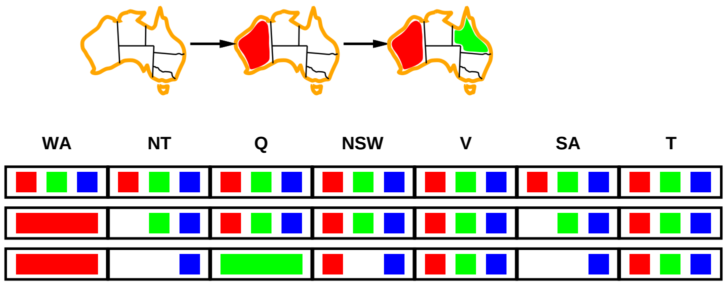
Forward checking

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:



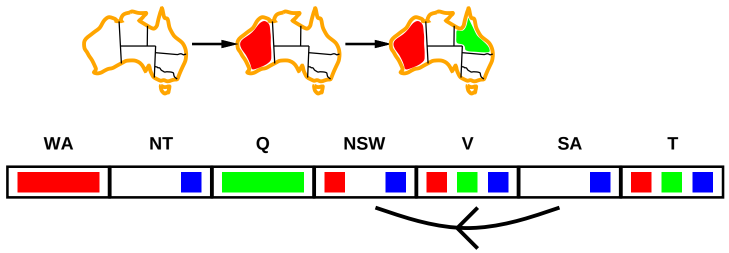
NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency

Simplest form of propagation makes each arc consistent:

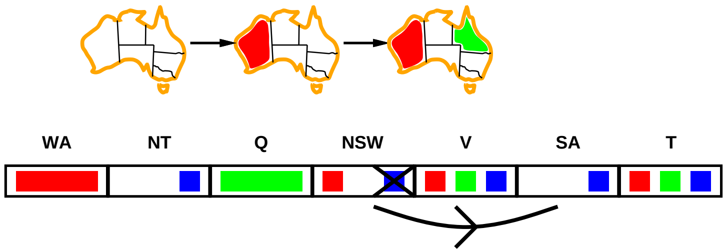
$X \rightarrow Y$ is consistent iff for **every** value x of X there is **some** allowed y



Arc consistency

Simplest form of propagation makes each arc consistent

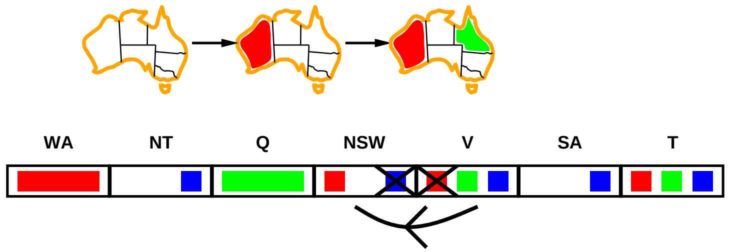
$X \rightarrow Y$ is consistent iff for **every** value x of X there is **some** allowed y



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

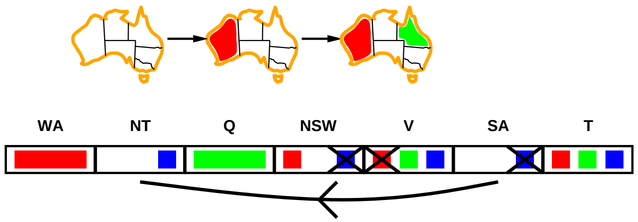


If X loses a value, neighbours of X need to be rechecked

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, neighbours 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

Subsidiary function:

```
function Remove-Inconsistent-Values( X_i, X_j) returns true iff we remove a value
   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 between X_i and X_j
         then delete x from Domain[X_i];  removed ← true
   return removed
```

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*

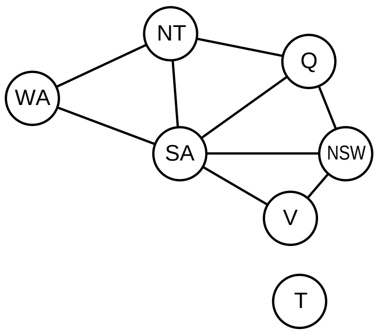
This runs in $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
but cannot detect all failures in poly time!

Recall: d is the max domain size, n the number of variables.

Example: n-queens

Using a combination of constraint propagation and heuristics we have seen so far, we can find solutions for the n-queens problem.

Problem structure



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

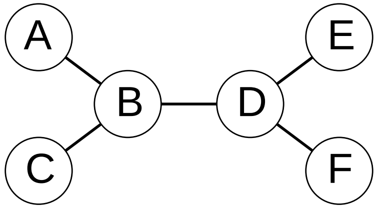
Problem structure contd.

Suppose divide problem into independent subproblems, where 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

Loop-free CSPs



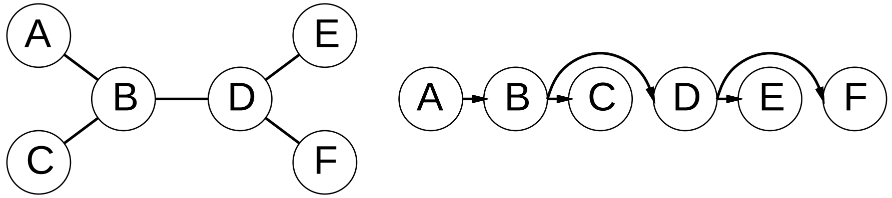
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

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 $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Iterative algorithms for CSPs

Hill-climbing typically works 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

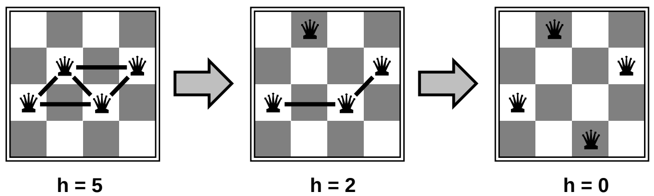
Example: 4-Queens

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



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)

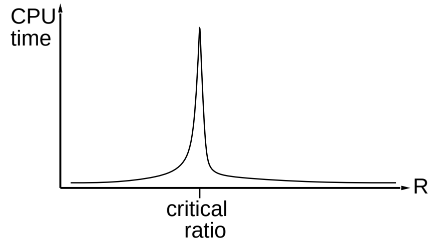
Translate each constraint into set of allowable values for its variables

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

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



Summary

- General purpose CSP heuristics
- Constraint propagation
- Arc consistency algorithm
- Local search for CSPs