Constraint Satisfaction Problems

Chapter 6.3-5

Adapted from slides kindly shared by Stuart Russell

Announcements

Deadline for Python Tutorial (P0): Today at 17:00.

Submit P0 via D2L Dropbox, via zip archive exactly as specified

Get started on Project 1 (search), lots of steps, due Thu 9-27 at 5pm

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

Outline

- ♢ Review backtracking, ordering, filtering, forward checking
- ♢ Review arc consistency, do exercise 6.11 in class as a group
- ♢ Problem structure and problem decomposition
- ♢ Local search for CSPs

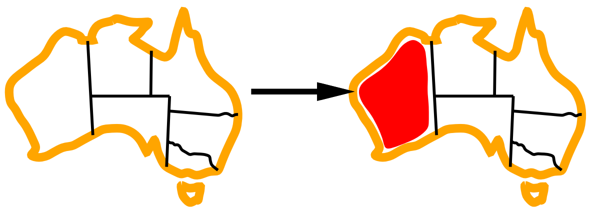
Filtering: Forward checking

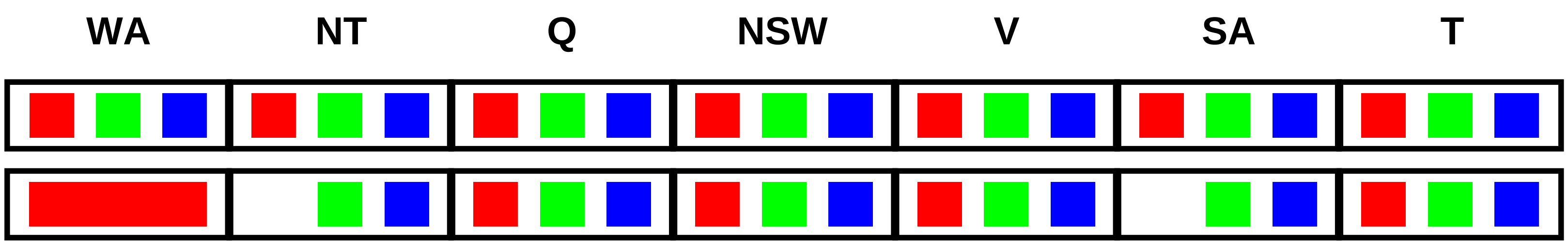
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

How is it different from Arc consistency?

Forward checking

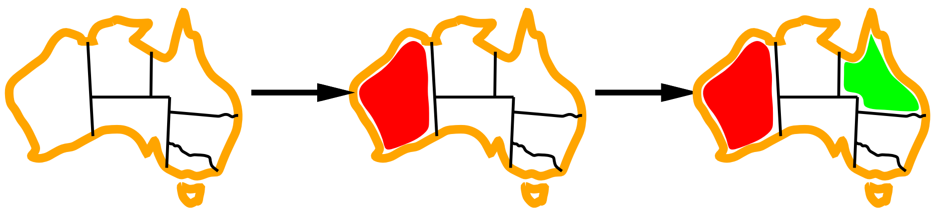
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





Forward checking

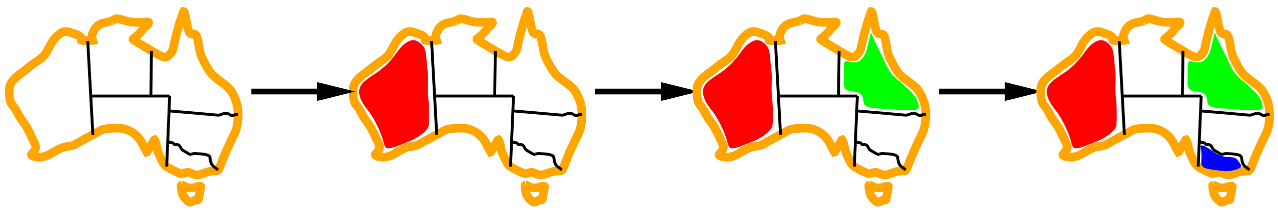
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

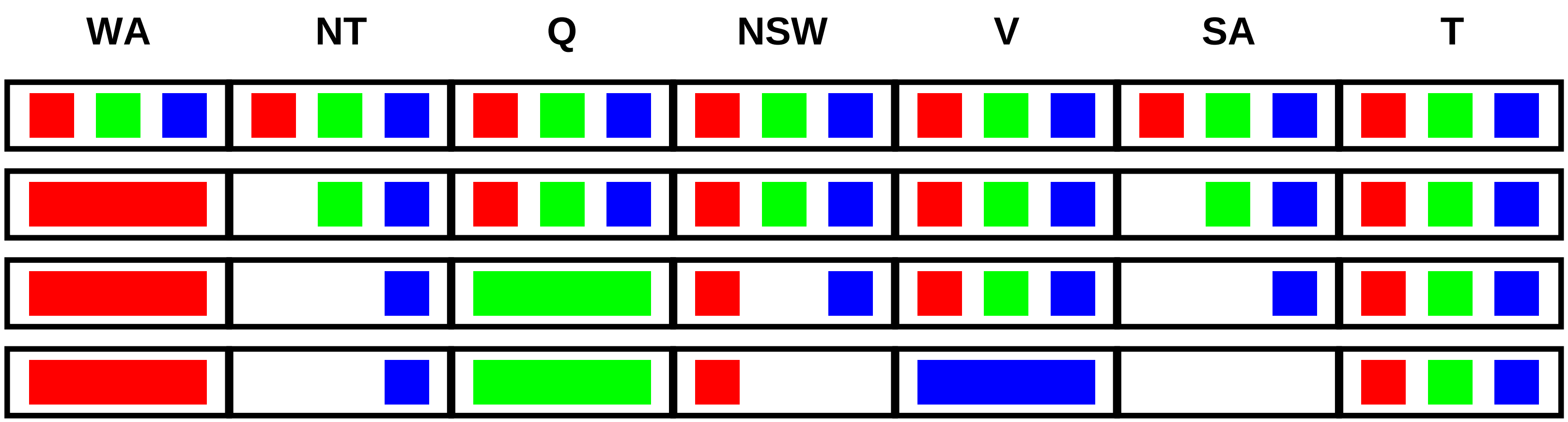


Forward checking

Idea: Keep track of remaining legal values for unassigned variables

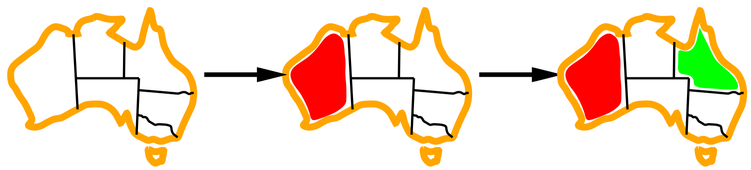
Terminate search when any variable has no legal values





Filtering: Constraint propagation

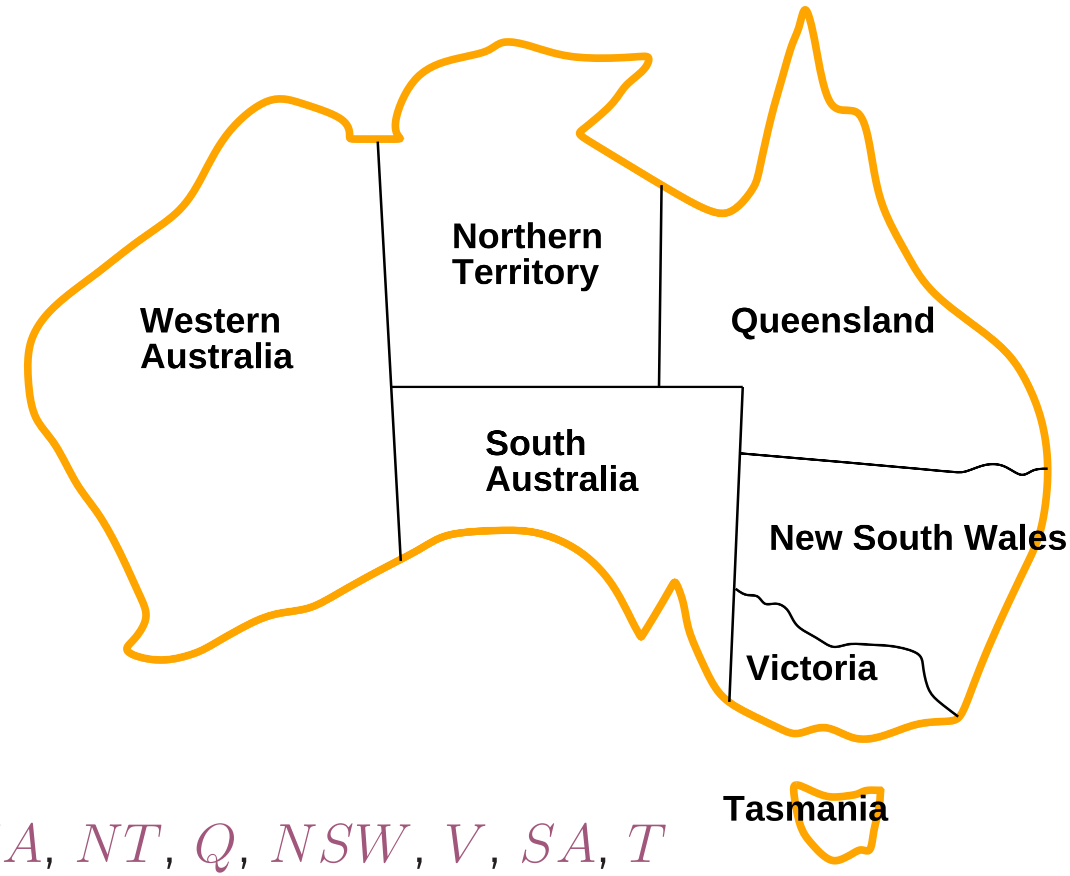
Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

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

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

Arc consistency algorithm

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains, or *false* if inconsistent
  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 size of DOMAIN$[X_i] = 0$ **then return** *false*
    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*

Arc consistency limitations

What are the limitations of arc consistency?

Consider three map regions, each with the same two colors in their domain

K-consistency

Propagating constraints progressively further out

1-consistency (Node Consistency)

2-consistency (Arc Consistency)

k-Consistency: for each k nodes, any consistent assignment to k-1 can be extended to the kth node

How expensive is that? Time? Space?

Exponential in k....

Strong K-consistency

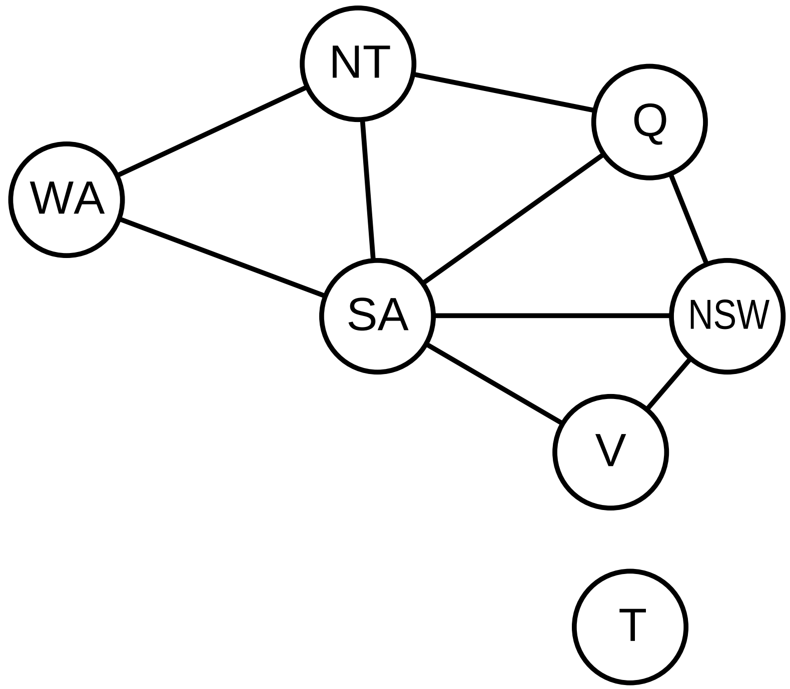
Strong k-consistency just means that not only is the CSP k-consistent, but also k-1, k-2, ... 1 consistent

Strong n-consistency means we can solve without backtracking!

Lots of middle ground

3-consistency the same as path consistency for binary CSPs

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

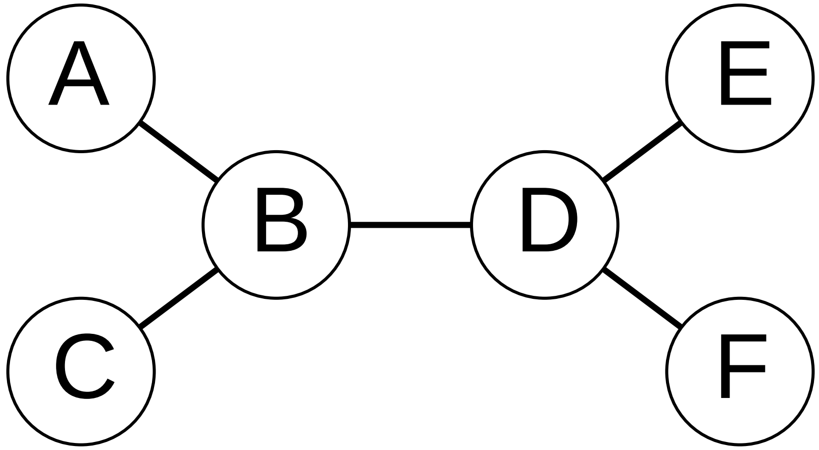
Problem structure contd.

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



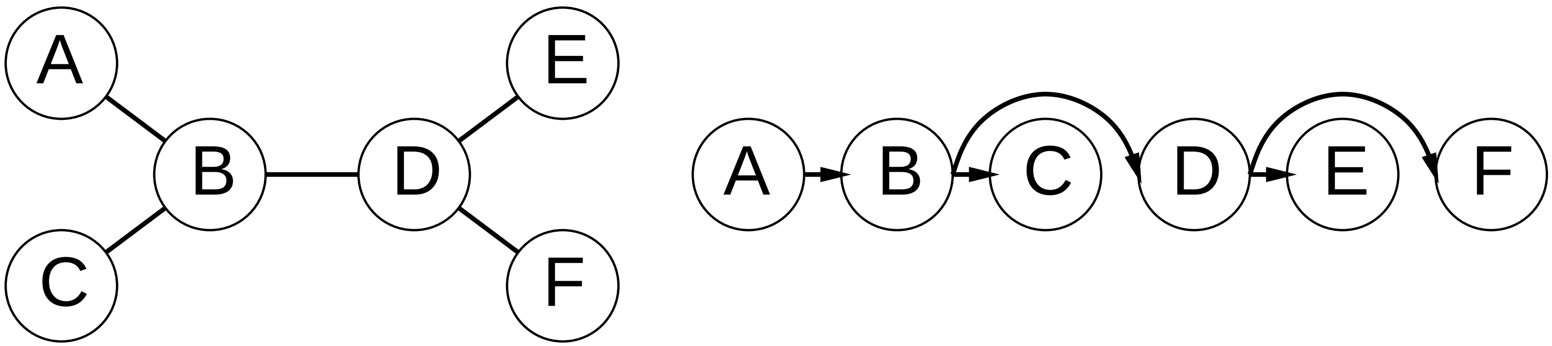
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\, d^2)$ time

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

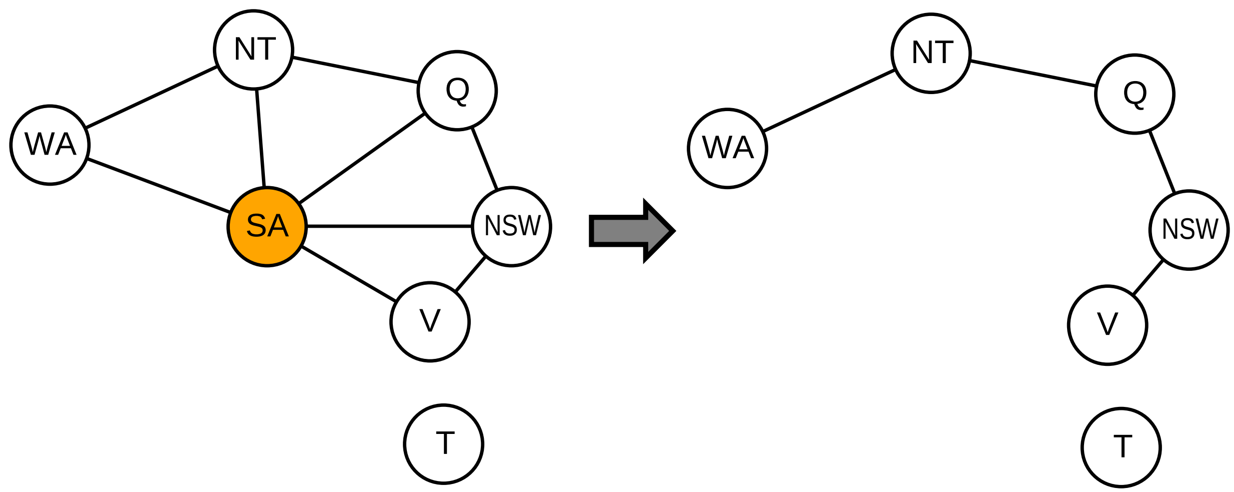


2. For j from n backward to 2, apply $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

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

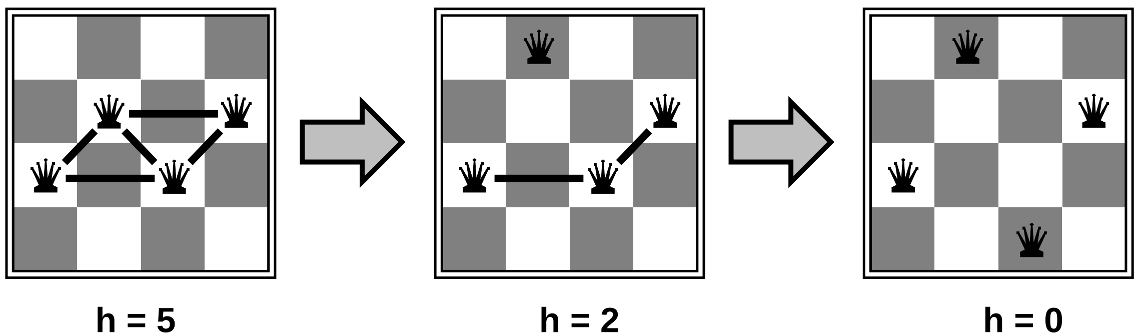
Example: 4-Queens

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

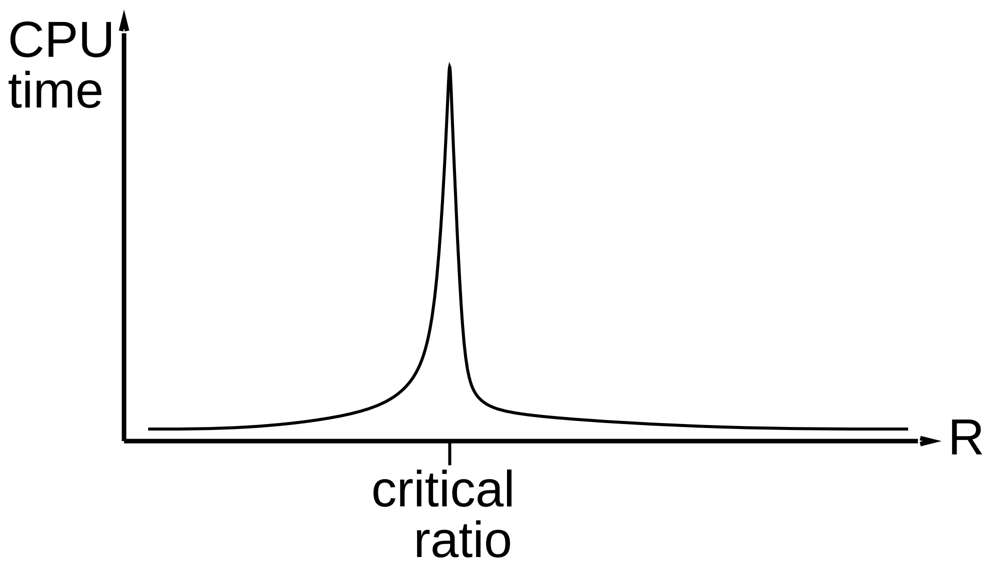


Performance of min-conflicts

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

Tradeoffs between degree / cost of constraint propagation (k-consistency) and of backtracking

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