Constraint Satisfaction Problems

Xiaojin Zhu

jerryzhu@cs.wisc.edu

Computer Sciences Department

University of Wisconsin, Madison

[Partly based on slides from Andrew Moore http://www.cs.cmu.edu/~awm/tutorials , and Russell & Norvig]

sudoku

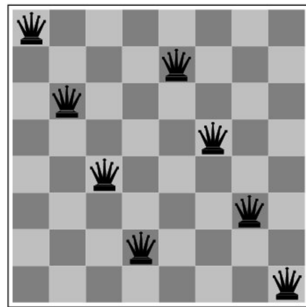
2	1	4						3
		3			4		5	
	7							
7	3		1		9	8		
			2		8			
		2	3		7		9	5
							1	
	6		9			5		
4						9	2	7

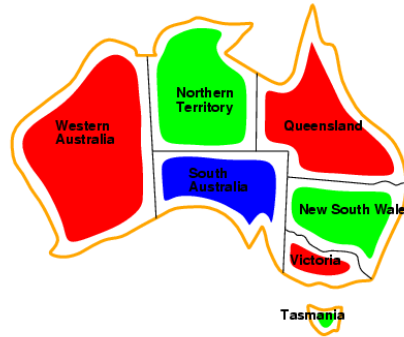
Daily SuDoku: Tue 20-Sep-2005 hard

(c) Daily Sudoku Ltd 2005. All rights reserved.

We've seen CSP before!

- Constraint satisfaction problem (CSP) is a special class of search problem





$A \vee \neg B \vee C$
$\neg A \vee C \vee D$
$B \vee D \vee \neg E$
$\neg C \vee \neg D \vee \neg E$
$\neg A \vee \neg C \vee E$

- Each problem has a set of variables (e.g. A,B,C,D,E)
- Each variable take a value from a domain (e.g. {T,F})
- Each problem has a set of constraints (e.g $A \vee \neg B \vee C$=T)
- Objective: find a complete assignment of variables that satisfies all the constraints.
- What are v/v/d/c of 8-queen? Map coloring?

CSP definition

- CSP is a triplet $\{V, D, C\}$
- $V = \{V_1, V_2, \ldots, V_n\}$ a finite set of variables
- Each variable may be assigned a value from domain D_i
- Each member of C is a pair
  - First member: a subset of variables
  - Second member: a set of valid values
- Example:

$V = \{V_1, V_2, \ldots, V_7\}$

$D = \{R, G, B\}$

$C = \{ (V_1, V_2):\{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)\},$

$(V_1, V_3):\{(R,G), (R,B), (G,B), (G,R), (B,G), (B,R)\},$

…

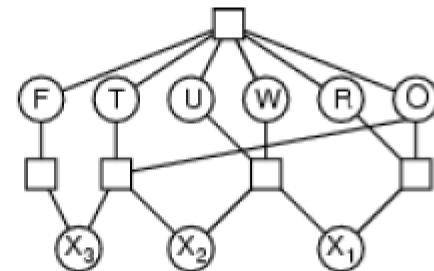
$\}$ (obvious point: C is often represented as a function)

- How did we solve this?

CSP examples

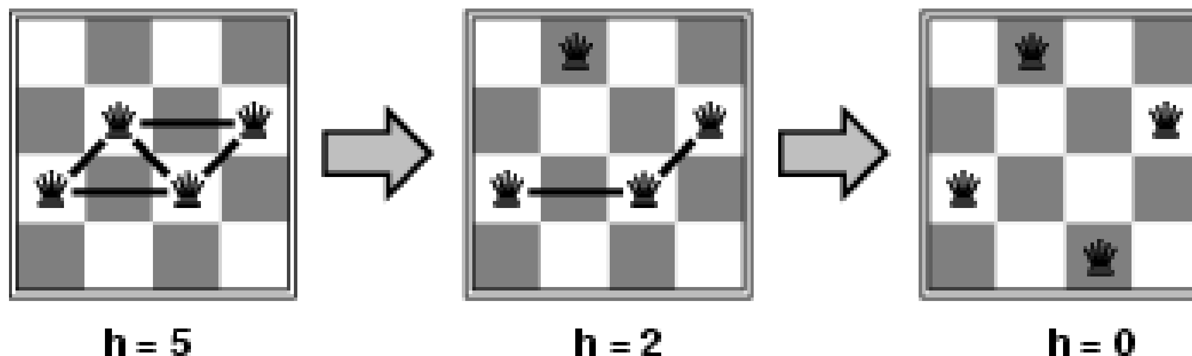
- Timetabling problems (which classes to take?)
- Hubble Telescope / Airline scheduling
- VLSI layout
- Boolean satisfiability
- Graph coloring
- Crosswords, minesweeper
- Cryptarithmetic
  - Variables: F,T,U,W,R,O,X1,X2,X3
  - Domains: {0,1,2,3,4,5,6,7,8,9}
  - Constraints:
    - AllDifferent(F,T,U,W,R,O)
    - O+O=R+10*X1
    - X1+W+W=U+10*X2
    - X2+T+T=O+10*X3
    - X3=F, T>0, F>0

T W O
\+ T W O
F O U R



Old solution #1: stochastic search

- State = a complete (maybe bad) assignment of variables
- Score = # of violated constraints
- Do hill-climbing (downhill!) and variations, or simulated annealing, or genetic algorithm etc.
- Hope to find a state with score=0
- Conceptually simple, empirically can be quite successful



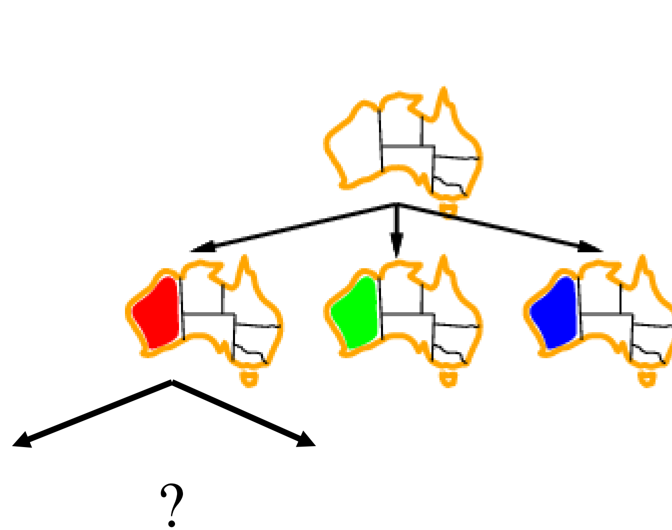
Can solve 10,000,000-queen in constant time. Worth having in your toolbox.

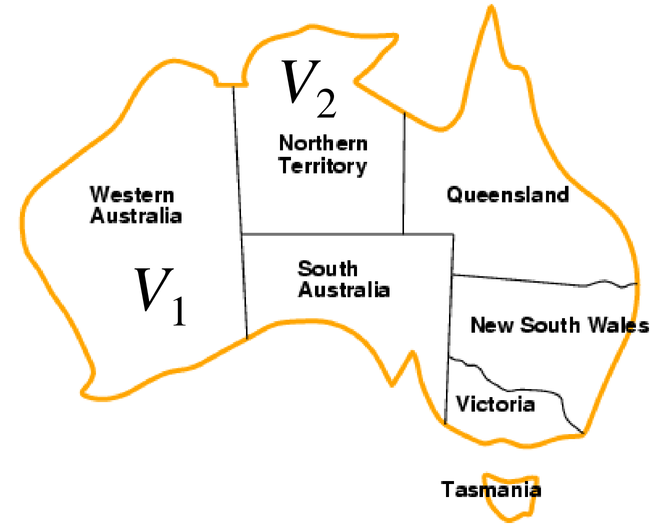
- So why are we still having this lecture?
  - Because there are interesting general structures in CSP that you should know

Old solution #2: BFS, DFS, …

- State: partial assignment. ($V_1 \ldots V_{k-1}$ assigned, $V_k \ldots V_n$ not yet).
- Start state: all variables unassigned
- Goal state: all assigned, constraints satisfied
- Successor of ($V_1 \ldots V_{k-1}$ assigned, $V_k \ldots V_n$ not yet): assign V_k with a value from D_k
- Cost on transitions: 0 is fine. We don't care. We just want any solution.

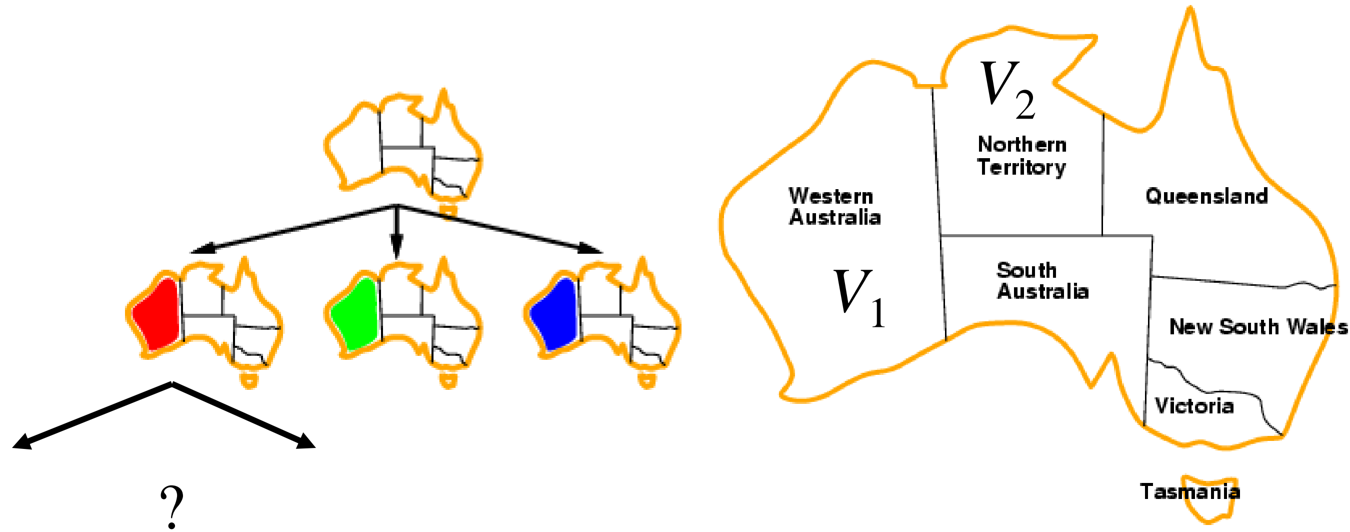
Map coloring example





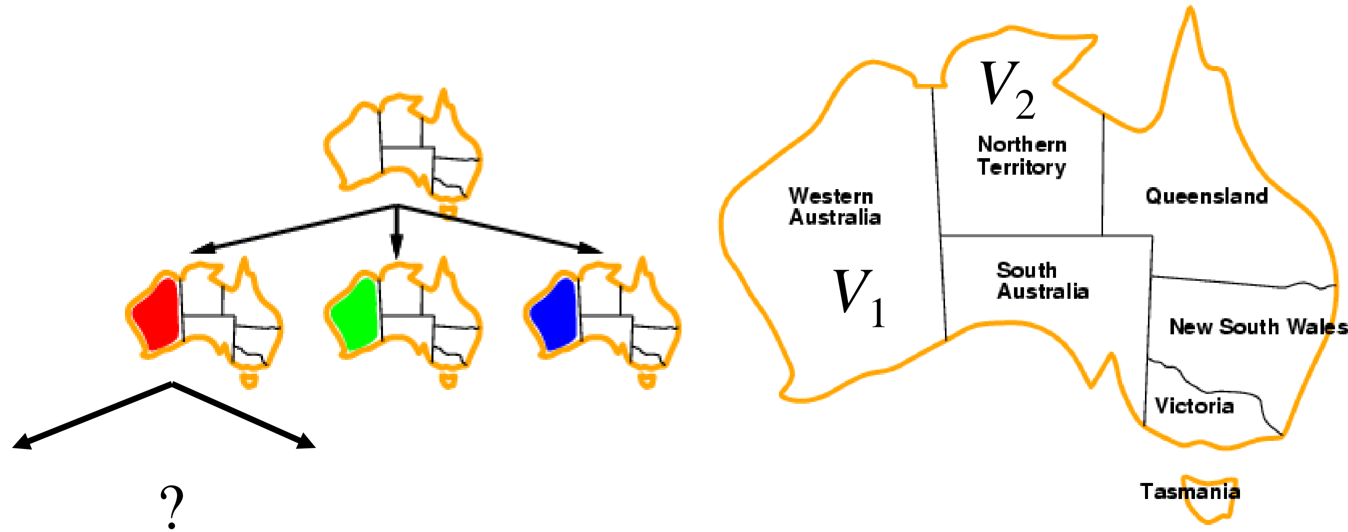
- It turns out BFS is bad. Why?

Map coloring example

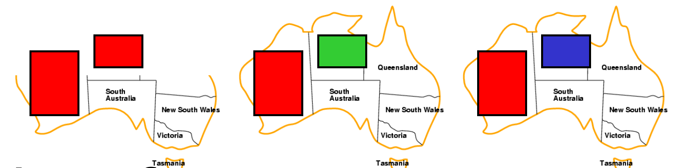


- It turns out BFS is bad. Why? Goal @ search tree leaf level.
- What are the successors above?

Map coloring example



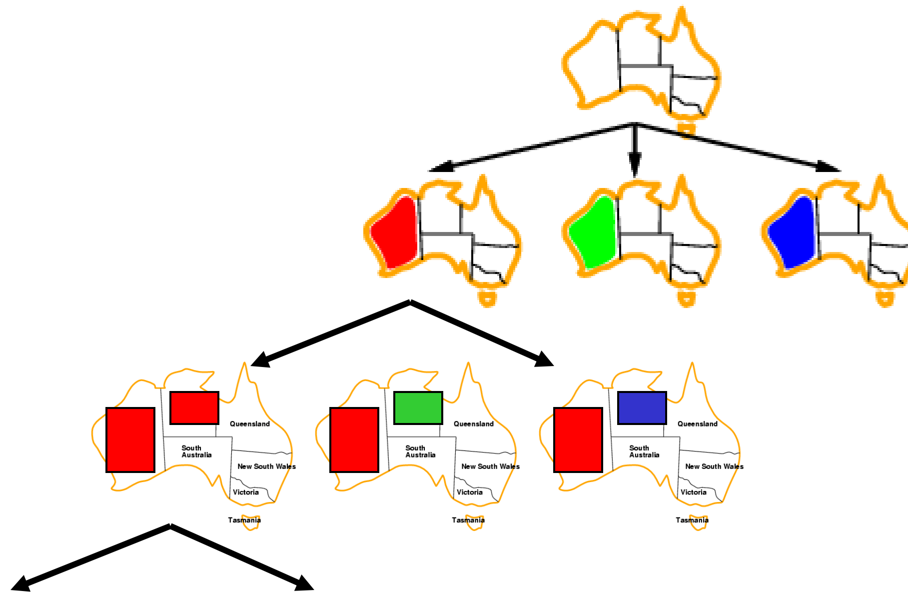
- It turns out BFS is bad. Why? Goal @ search tree leaf level.
- What are the successors above?



- Let's say for every variable the order of assignment is R, G, B. There's something wrong with DFS, can you see why?

Map coloring example

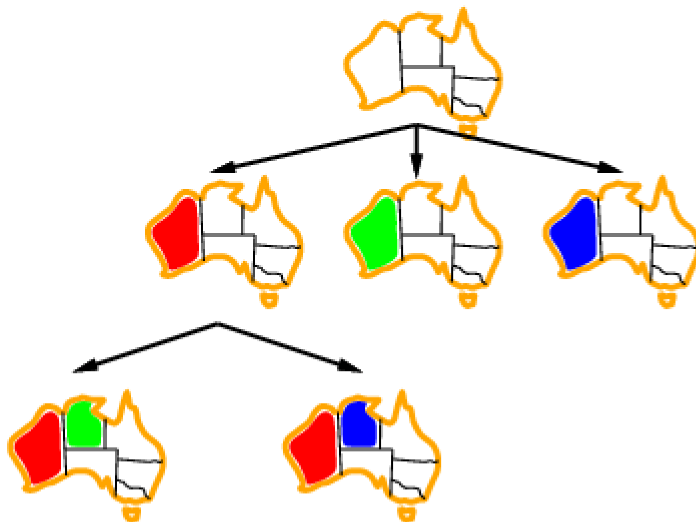
There's something wrong with DFS, can you see why?



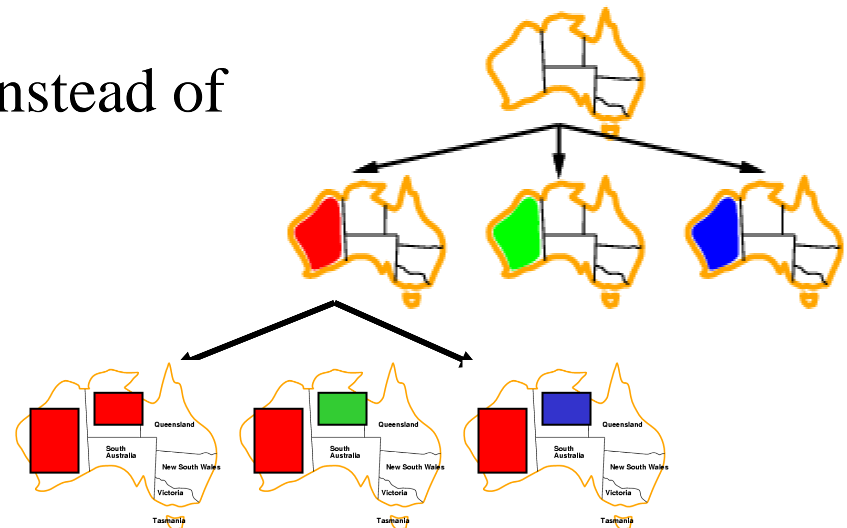
Shouldn't search anything down here!

#1 Obvious improvement: backtracking search

- Succs() should check the constraints and not propose a successor assignment that conflicts with other already-assigned variables.
- ‘backtracking’ happens when no value is valid for that successor.



instead of



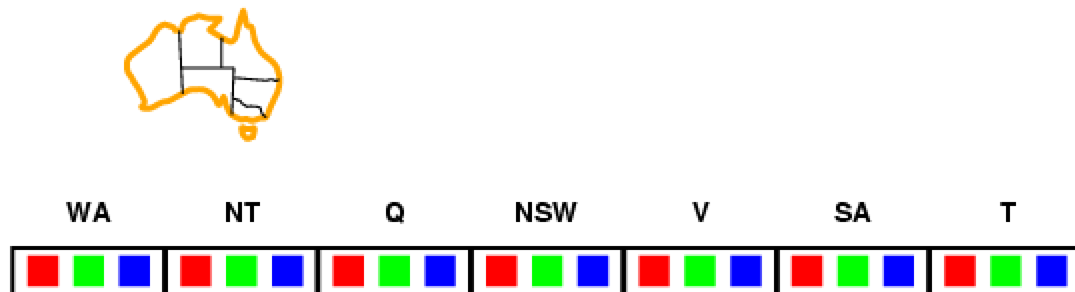
Backtracking search

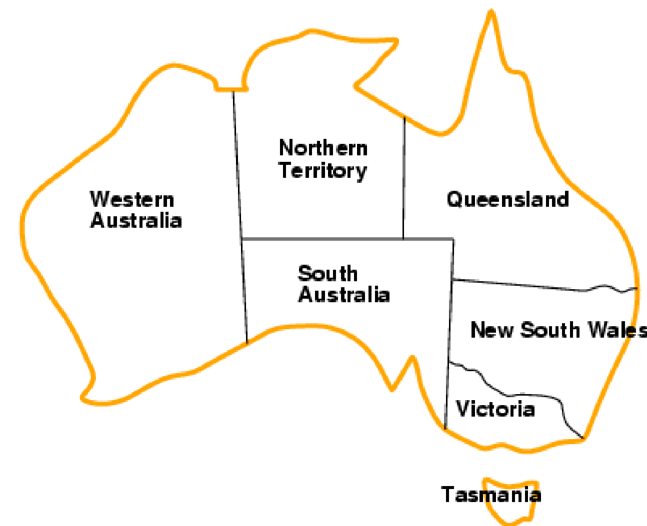
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or
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 according to Constraints[csp] then
         add { var = value } to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failue then return result
         remove { var = value } from assignment
   return failure
```

#2 Less obvious improvement: forward checking

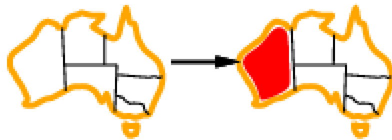
- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i=v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.

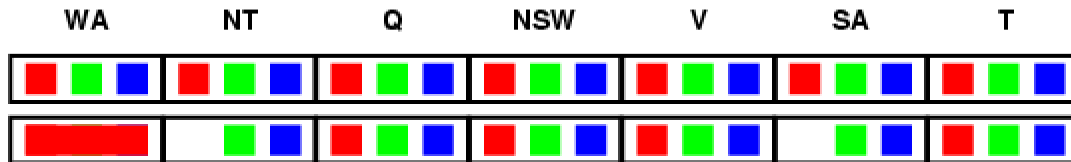




Less obvious: forward checking

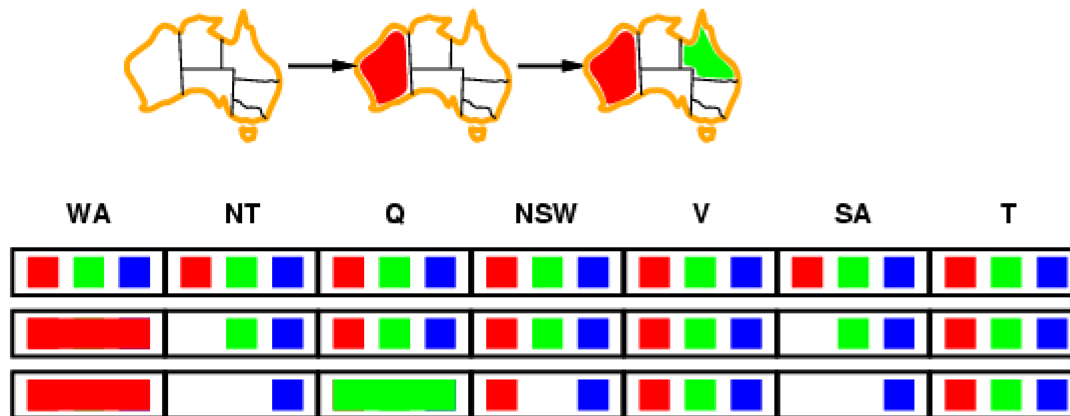
- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i=v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.

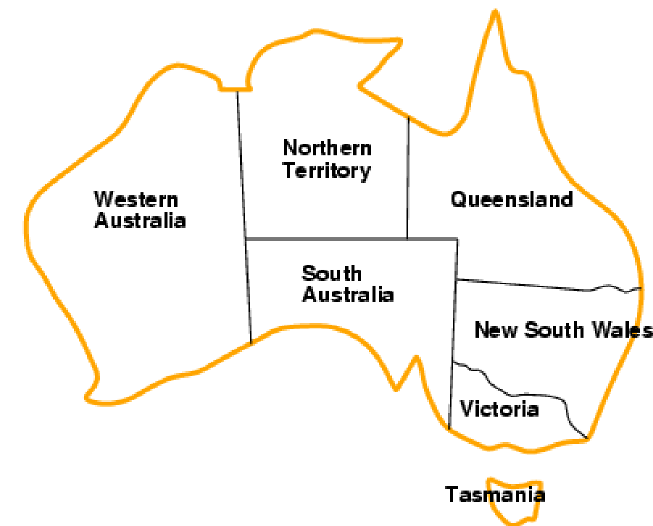




Less obvious: forward checking

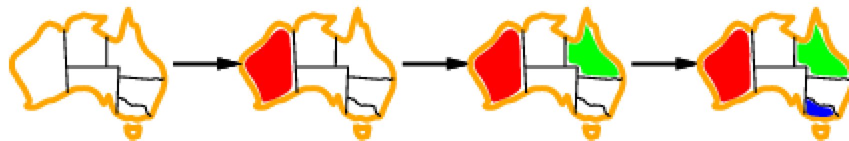
- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i=v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.





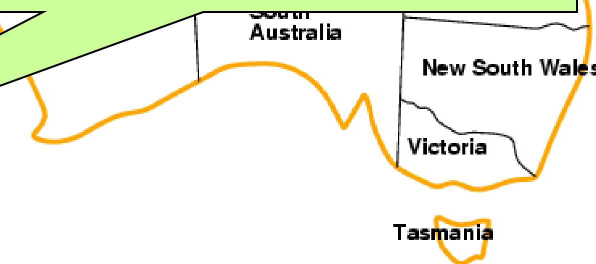
Less obvious: forward checking

- Keep a list of candidate values for each unassigned variable.
- After assigning $V_i = v$, cross out conflicting candidates in other unassigned variables.
- If any unassigned variable's candidate list becomes empty, backtrack immediately.



WA NT Q NSW V SA T

SA may not be the next variable we assign. Thus backtracking search has slower response than forward checking.



Variable ordering heuristics

- When expanding, choose to expand the most-constrained variable first, i.e. the variable with the fewest legal values
- a.k.a. minimum remaining value (MRV) heuristic
- The hope: minimize the search sub-tree. if things are going to fail, let it fail fast

- When there's a tie: choose the most constraining variable to break the tie, i.e. the variable with the most constraints on remaining variables
- The hope: minimize the search sub-tree. if things are going to fail, let it fail fast

Value ordering heuristic

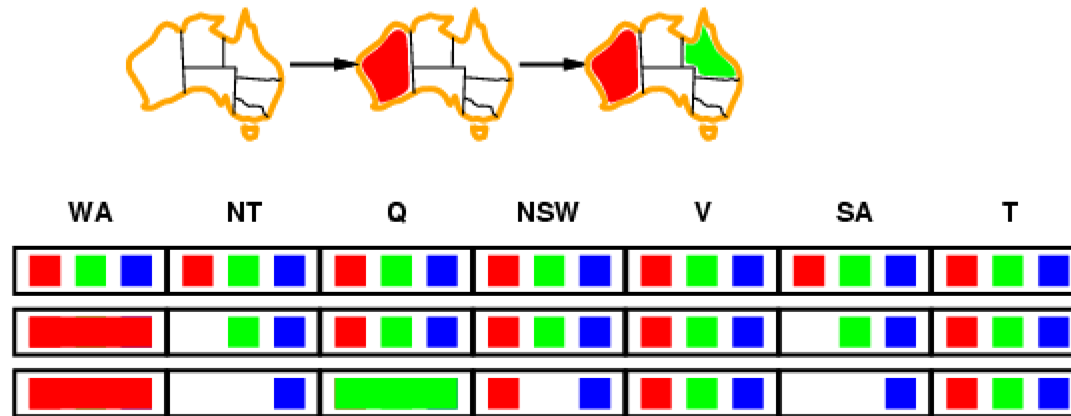
- Given a variable, try the least constraining value first, i.e. the one that rules out the fewest values in the remaining variables
- Why do we prefer the most constrained/constraining variable, but the least constraining value?

Value ordering heuristic

- Given a variable, try the least constraining value first, i.e. the one that rules out the fewest values in the remaining variables
- Why do we prefer the most constrained/constraining variable, but the least constraining value?
- The reason: the failure of a value will not reduce the sub-tree size at all – if one value fails, the remaining values will still be tried. In light of this, we prefer a value that most likely leads to success first.

#3 Not obvious: Constraint propagation

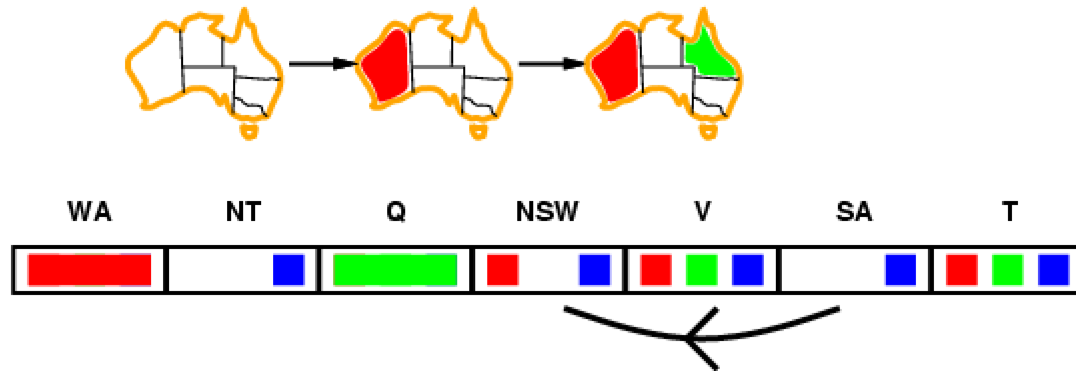
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



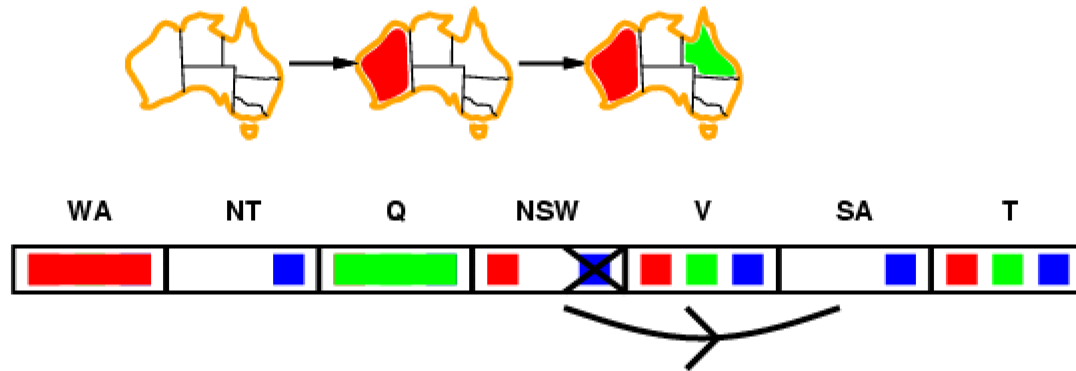
- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

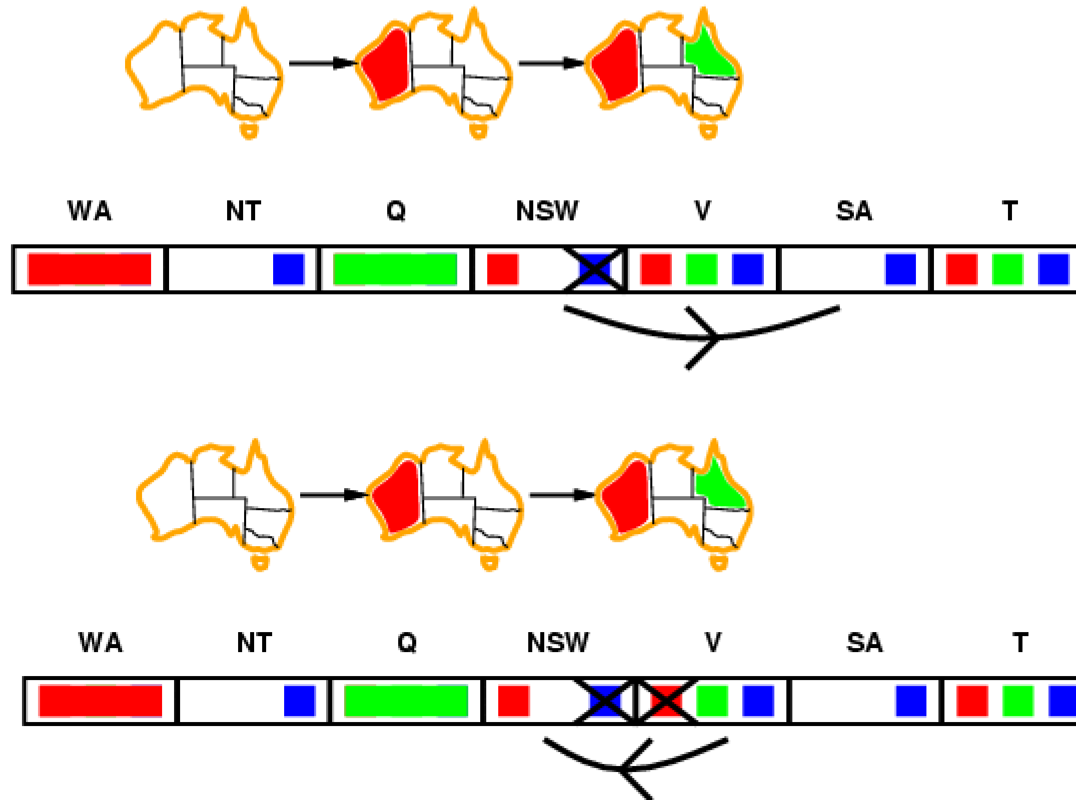
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff for every value x of X there is some allowed y



Arc consistency

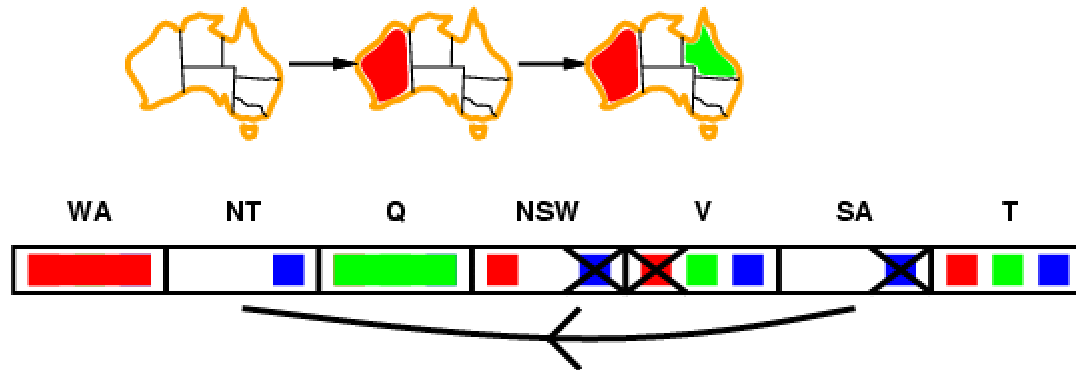


Arc consistency



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

Arc consistency



Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X_1, X_2, ..., 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) ← REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

```
function RM-INCONSISTENT-VALUES( X_i, X_j) returns true iff 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 constraint(X_i, X_j)
         then delete x from DOMAIN[X_i];   removed ← true
   return removed
```

What you should know

- How to formalize problems as CSP
- Backtracking search, forward checking, constraint propagation
- Variable ordering and value ordering
- Stochastic search for CSP