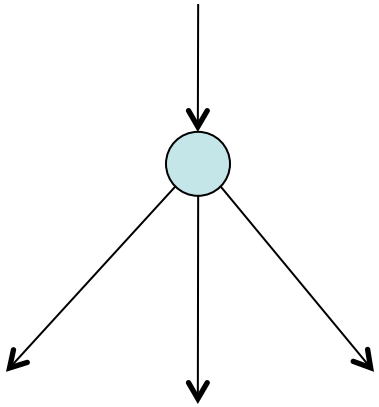
Constraint Satisfaction

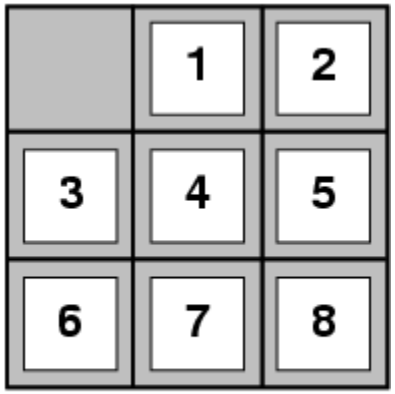
CMPSCI 383
October 4, 2008

Today's lecture

- Defining and representing constraint satisfaction problems (CSPs)
- CSPs as search
- Being smart about search in CSPs
  - Variable and value ordering
  - Propagating information through constraints
  - Intelligent backtracking

States: Black Boxes or Things with Structure?

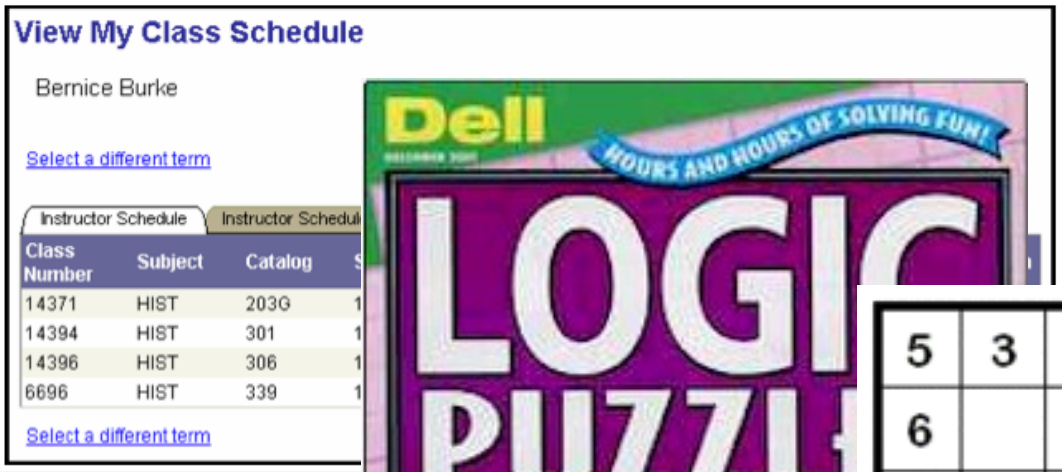




What is constraint satisfaction?

- A family of methods for solving problems where the internal structure of solutions must satisfy a set of specified constraints
- Big ideas
  - Variable and value ordering
    Choose next states wisely
  - Propagate information
    Provide good information to your heuristics
  - Intelligent backtracking
    Backtrack in a smart way

Have you done constraint satisfaction?

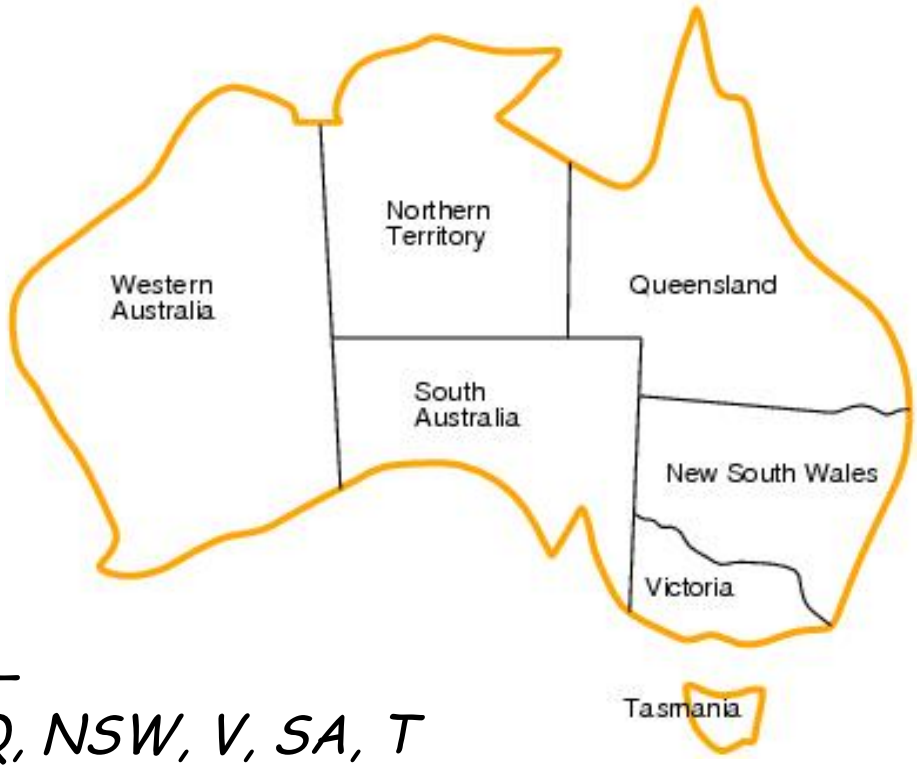


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

What defines a CSP?

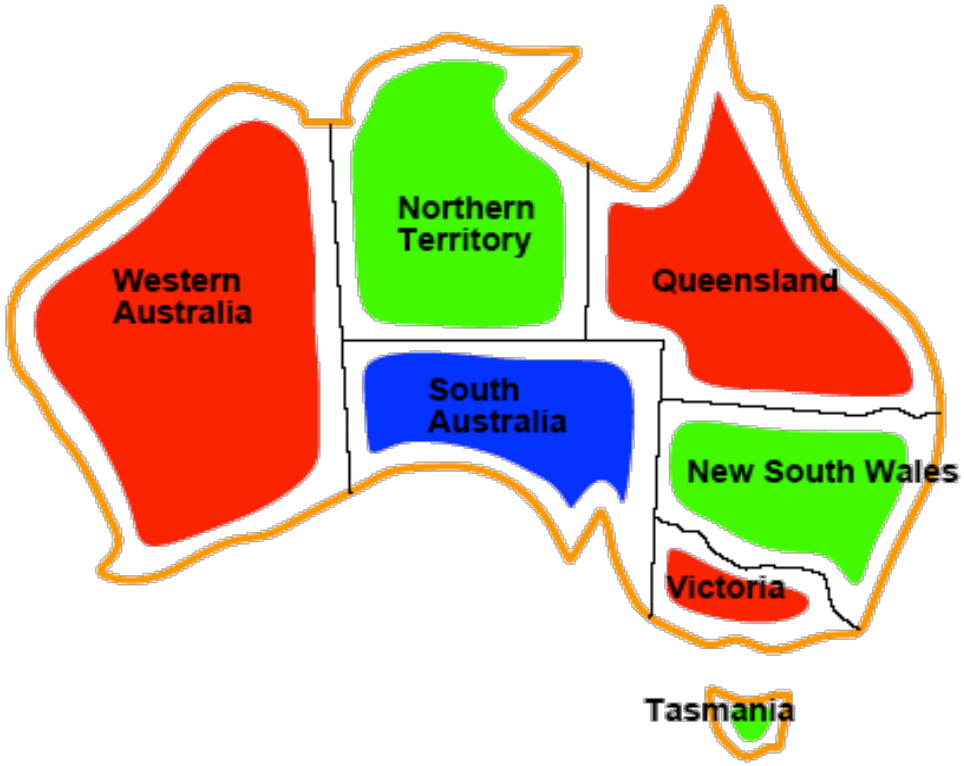
- In CSPs, states are defined by assignments of values to a set of variables $X_1 \ldots X_n$. Each variable X_i has a domain D_i of possible values.
- States are evaluated based on their consistency with a set of constraints $C_1 \ldots C_m$ over the values of the variables.
- A goal state is a complete assignment to all variables that satisfies all the constraints.

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* ≠ *NT* (if the language allows this)
  - E.g. ((*WA,NT*), [(*red,green),(red,blue),(green,red),…*])

Example: Map coloring



- Solutions are **complete** and **consistent** assignments:
  every variable assigned, all assignments legal, e.g.:
  {WA=red,NT=green,Q=red,NSW=green,V=red,SA=blue,T=green}

Varieties of CSPs

- Discrete variables
  - finite domains:
    - n variables, domain 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., $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Real-world CSPs

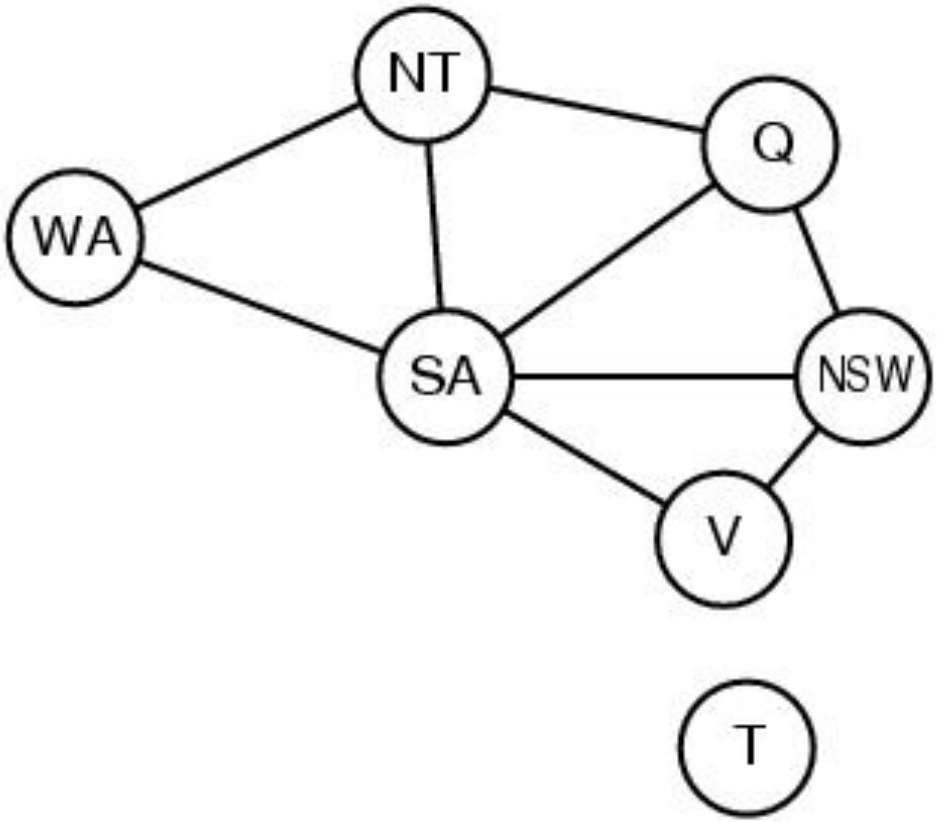
- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

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

Types of Constraints

- Unary constraint: concerns only a single value; e.g., $SA \neq green$
- Binary constraint: concerns the relative values of two variables
- Global constraint: concerns an arbitrary number of variables, e.g., *Alldiff*

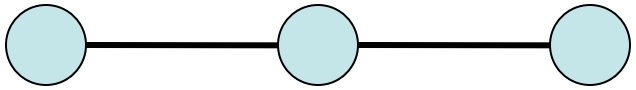
Constraint graph



Example: Cryptarithmetic

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

Local Consistency

- Node Consistency: satisfies all unary constraints
- Arc Consistency: satisfies all binary constraints
- Path Consistency: 
- n-consistency: for any consistent assignment to any set of n-1 variables, a consistent value can be found for any n-th variable.

Arc Consistency (slightly different from the book)

```
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 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 ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

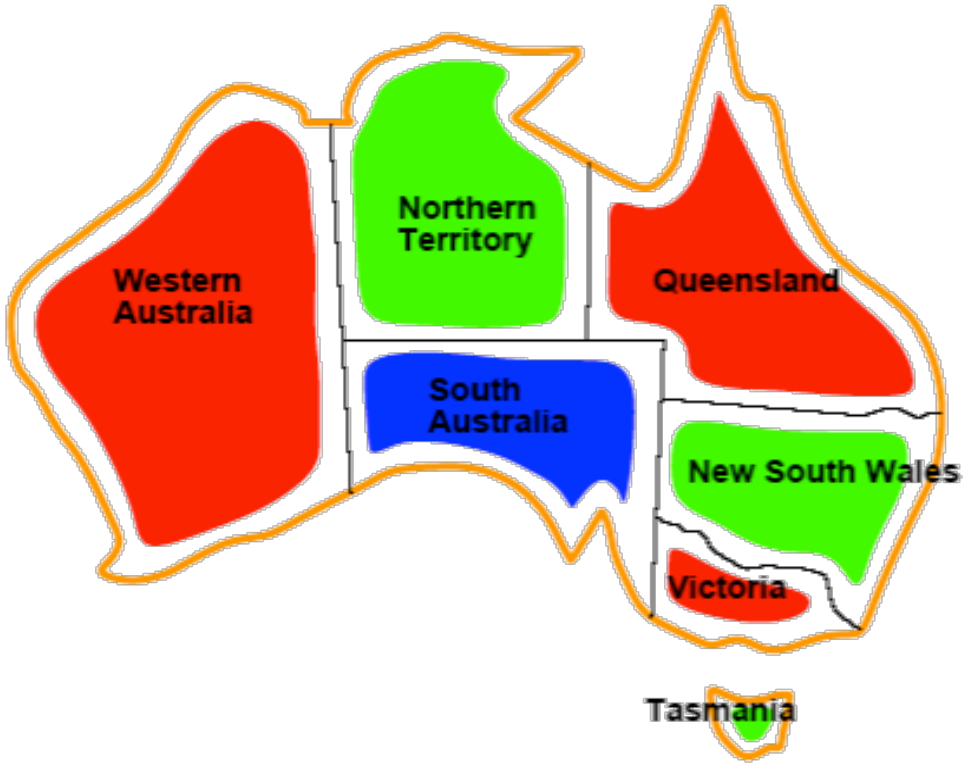
	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Constraints?

Alldif for each row, each column, and each 3x3 box

Can we tackle this as a search problem?

- What does the search tree look like?
  - What are the states?
  - What is the successor function?
- What is the branching factor?
- What search method would you apply?
- What are some good heuristics?



Standard Search Formulation

Let's start with the straightforward 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

- 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 - k)d at depth k, hence n! · d^n leaves (d is domain size)

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

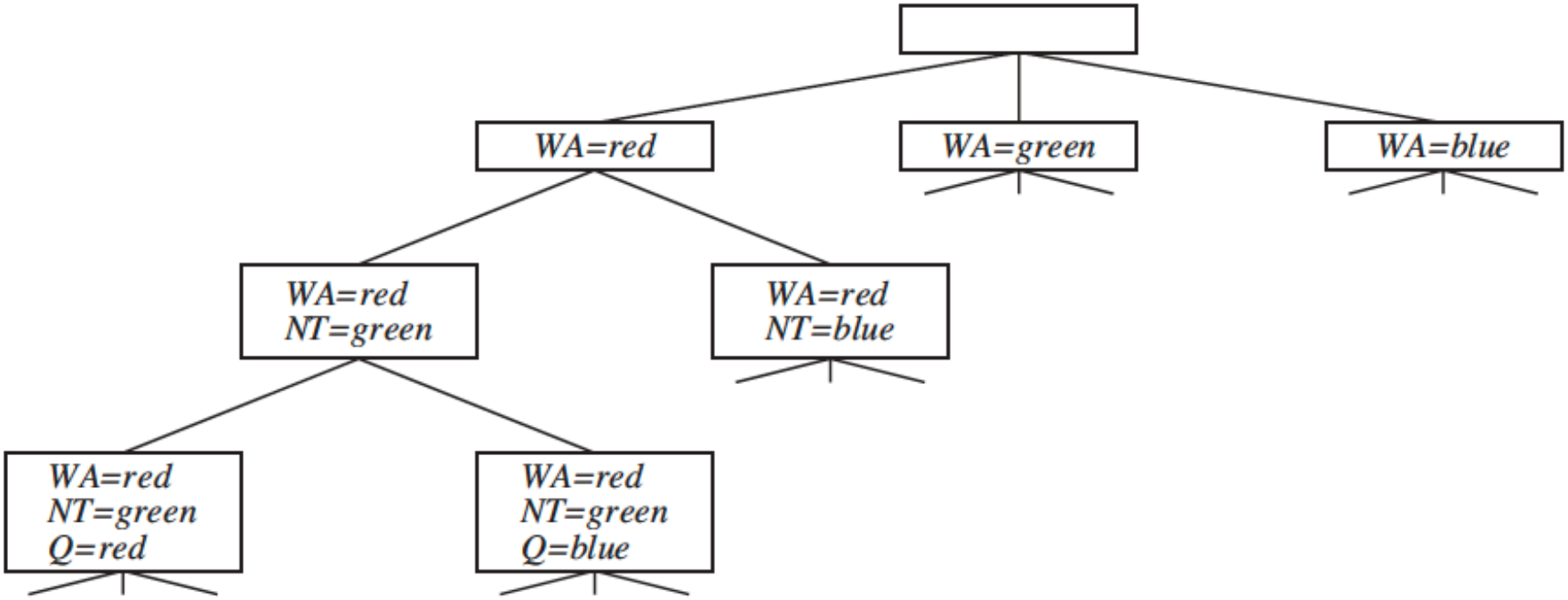
  → b = d and there are d^n leaves

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

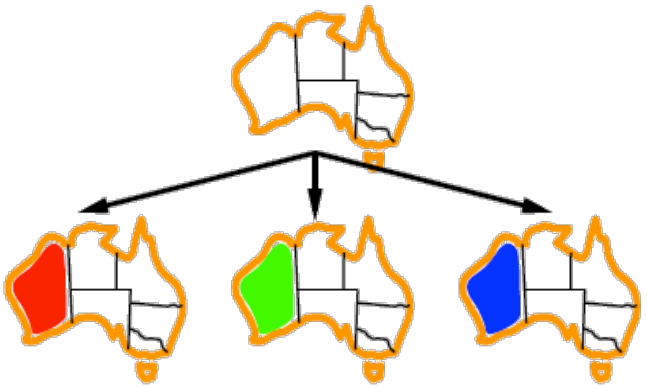
- Backtracking search is the basic uninformed algorithm for CSPs

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

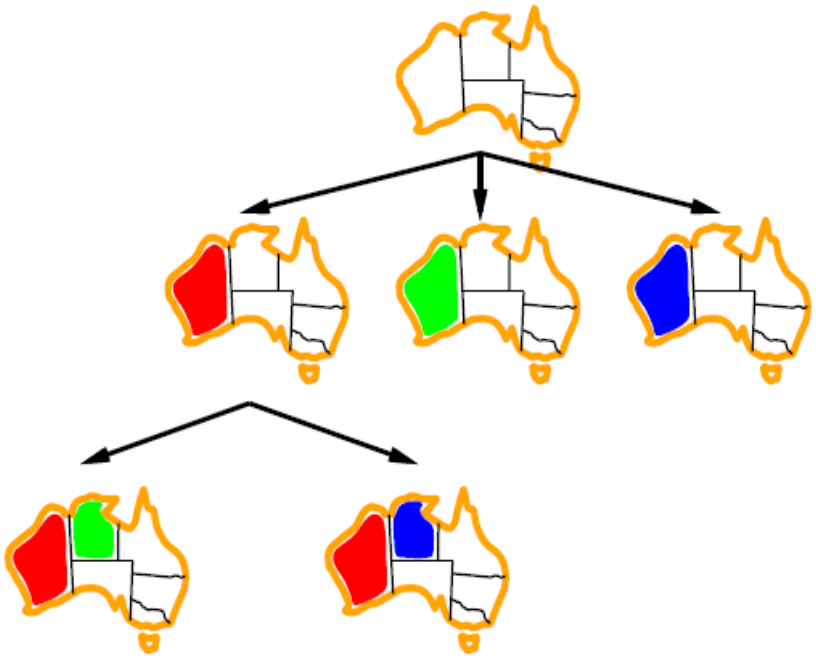
Backtracking Search for coloring Austrailia



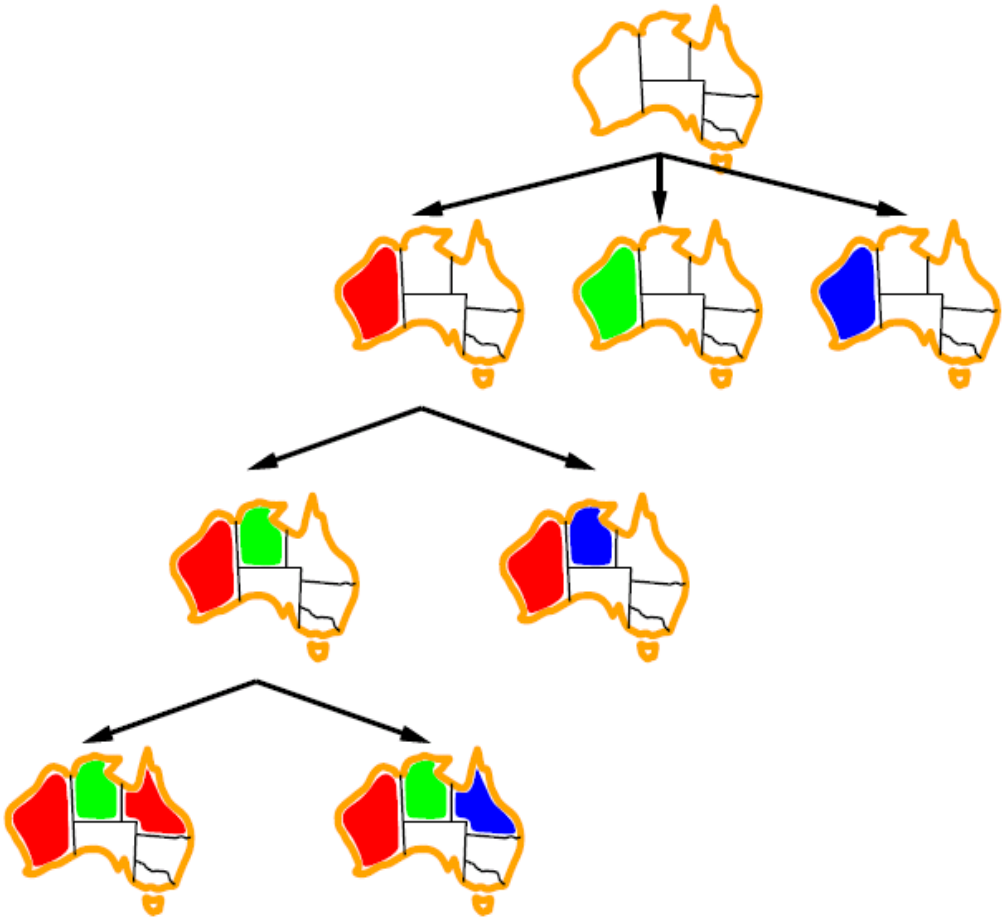
Search example



Search example



Search example



Simple backtracking search

- Depth-first search
- Choose values for one variable at a time
- Backtrack when a variable has no legal values left to assign.
- If search is uninformed, then general performance is relatively poor

Backtracking Search (somewhat different from the book)

```
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 Search (the book's)

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

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

How could we do better?

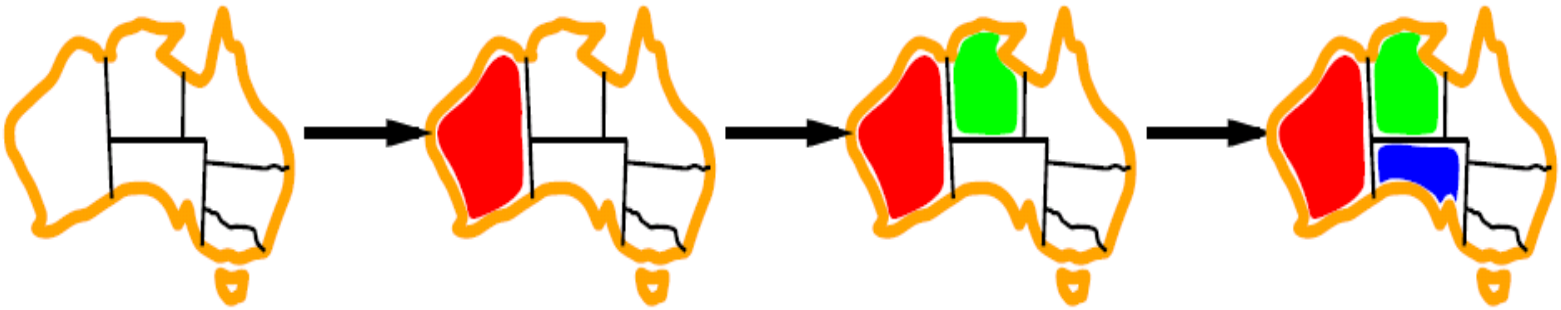
Improving backtracking efficiency

- Basic question: What next step should our search procedure take?
- Approaches
  - Minimum remaining values heuristic
  - Degree heuristic
  - Least-constraining value heuristic

Improving Backtracking Efficiency

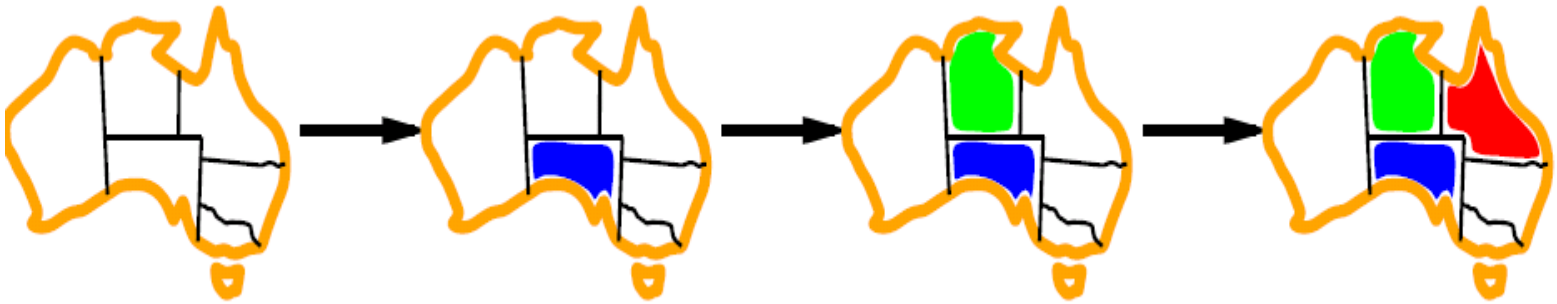
- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Minimum remaining values (MRV) heuristic



- Select the *most constrained* variable (the variable with the smallest number of remaining values)

Degree heuristic



- Select the variable that is involved in the largest number of constraints with other unassigned variables
- The most constraining variable.
- A useful tie breaker (and guide to starting).
- In what order should its values be tried?

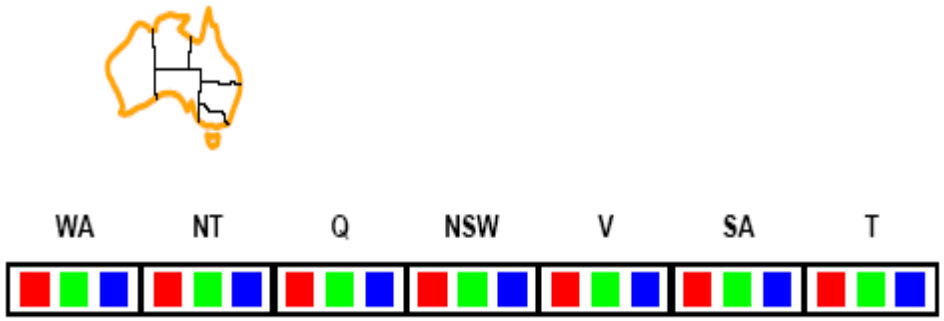
Least constraining value

- Given a variable, choose the *least constraining value* — the value that leaves the maximum flexibility for subsequent variable assignments.
- Combining these makes 1000 Queens possible.

Combining Search with Inference

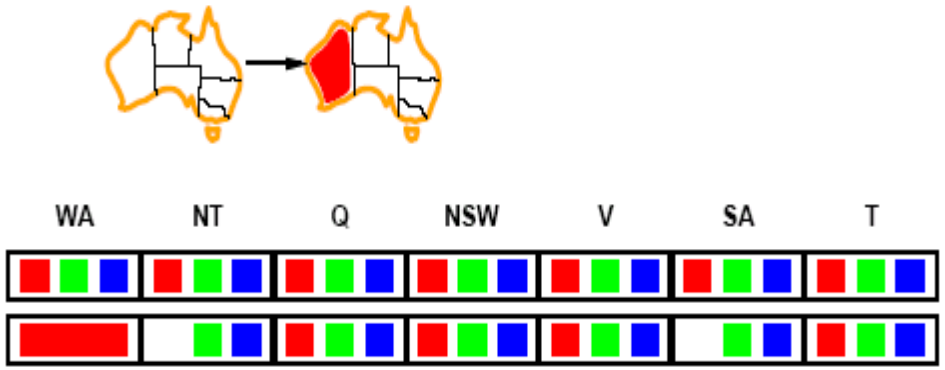
- Basic question — Can we provide better information to these heuristics?
- Forward checking
  - Precomputing information needed by MRV
  - Early stopping
- Constraint propagation
  - Arc consistency (2-consistency)
  - n-consistency

Forward checking



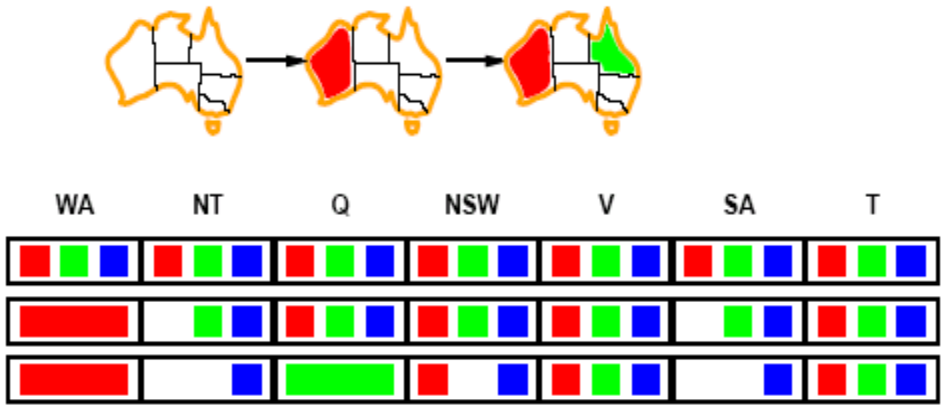
- Can we detect inevitable failure early?
  - *And avoid it later?*
- Yes — track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values.

Forward checking



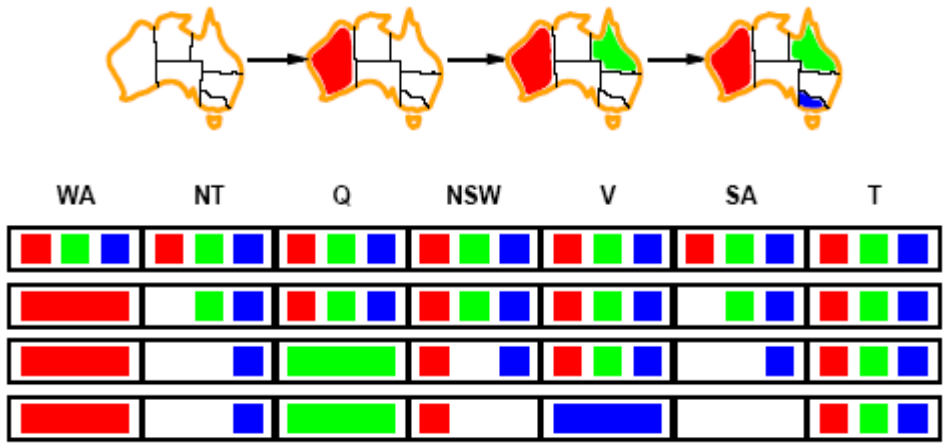
- *Assign {WA=red}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be red*
  - *SA can no longer be red*

Forward checking



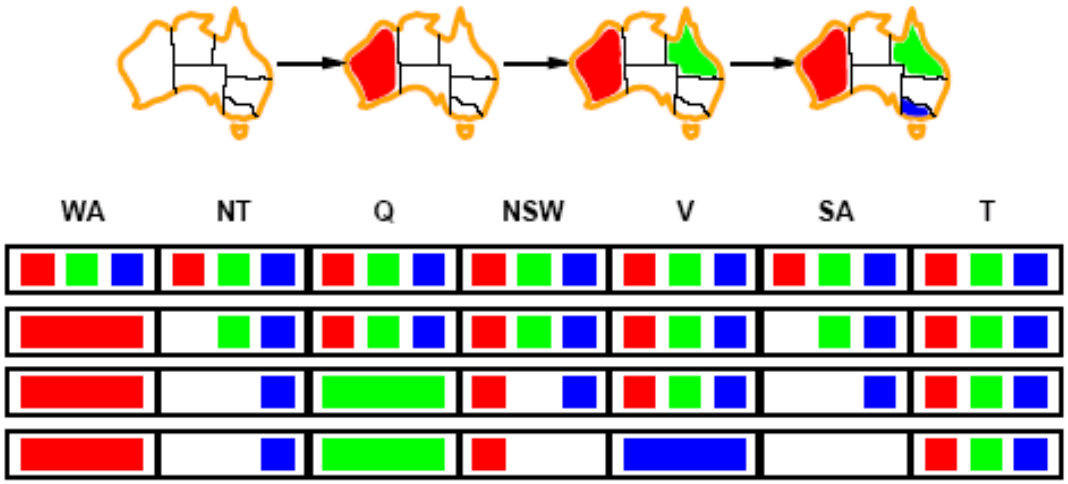
- *Assign {Q=green}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

Forward checking



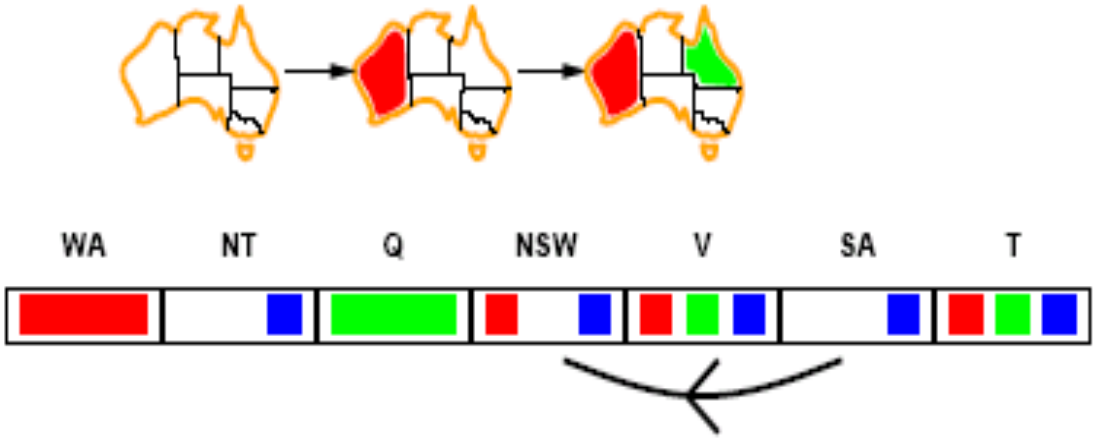
- If *V* is assigned *blue*
- Effects on other variables connected to WA
  - *SA is empty*
  - *NSW can no longer be blue*
- FC has detected a partial assignment that is *inconsistent* with the constraints.

Forward checking



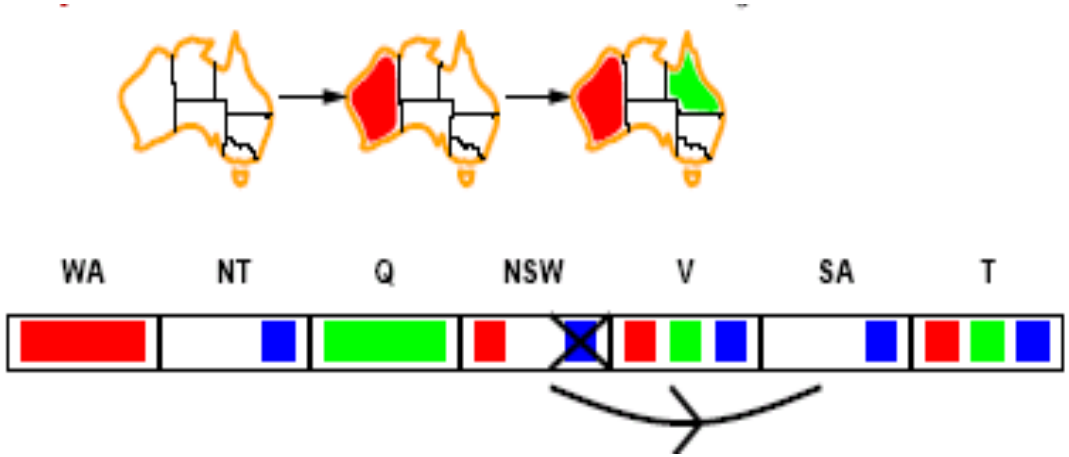
- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC checking propagates information from assigned to unassigned variables but does not provide detection for all failures.
  - NT and SA cannot be blue!
- Makes each current variable assignment arc consistent, but does not look far enough ahead to detect all inconsistencies (as AC-3 would)

Arc consistency



- $X \rightarrow Y$ is consistent iff
  for *every* value x of X there is some allowed y
- $SA \rightarrow NSW$ is consistent iff
  SA=blue and *NSW=red*

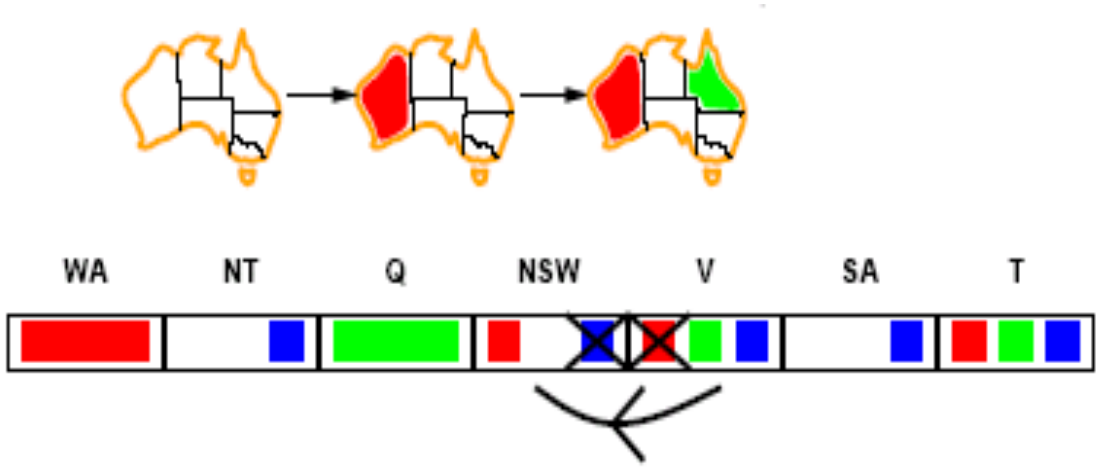
Arc consistency



- *X* → *Y* is consistent iff
  for *every* value *x* of *X* there is some allowed *y*
- *NSW* → *SA* is consistent iff
  NSW=red and *SA=blue*
  NSW=blue and SA=???

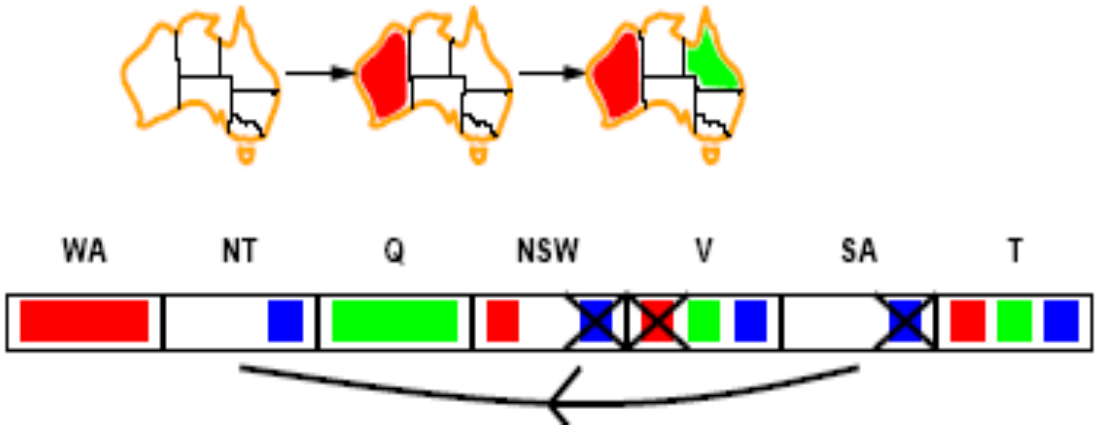
Arc can be made consistent by removing *blue* from *NSW*

Arc consistency



- Arc can be made consistent by removing *blue* from *NSW*
- Recheck *neighbours*
  - Remove red from *V*

Arc consistency



- Arc can be made consistent by removing *blue* from *NSW*
- Recheck *neighbours*
  - Remove red from *V*
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment.
  - Repeated until no inconsistency remains

n-consistency

- Arc consistency does not detect all inconsistencies:
  - Partial assignment *{WA=red, NSW=red}* is inconsistent.
- Stronger forms of propagation can be defined using the notion of n-consistency.
- A CSP is n-consistent if for any set of n-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any nth variable.
  - E.g. 1-consistency or node-consistency
  - E.g. 2-consistency or arc-consistency
  - E.g. 3-consistency or path-consistency

Further improvements

- Checking special constraints
  - Checking *Alldif*(…) constraint
  - Checking *Atmost*(…) constraint
    - *Bounds propagation for larger value domains*
- Intelligent backtracking
  - Standard form is chronological backtracking i.e. try different value for preceding variable.
  - More intelligent, backtrack to conflict set.
    - Set of variables that caused the failure or set of previously assigned variables that are connected to X by constraints.
    - Backjumping moves back to most recent element of the conflict set.
    - Forward checking can be used to determine conflict set.

Key Ideas

- Basic form of a CSP
- Different types of CSPs
- Types of constraints
- Consistent assignment
- Complete assignment
- Constraint graph
- Constraint propagation
- Backtracking search for CSPs
- Heuristics to improve backtracking search
  - MRV
  - Degree heuristic
  - Least-constraining value
- Interleaving search and inference
  - Forward checking
  - Arc consistency
  - Smart backtracking

Next Class

- Next Thursday (Tuesday is Monday's schedule)
- More on Constraint Satisfaction
- Secs. 6.3 – 6.6