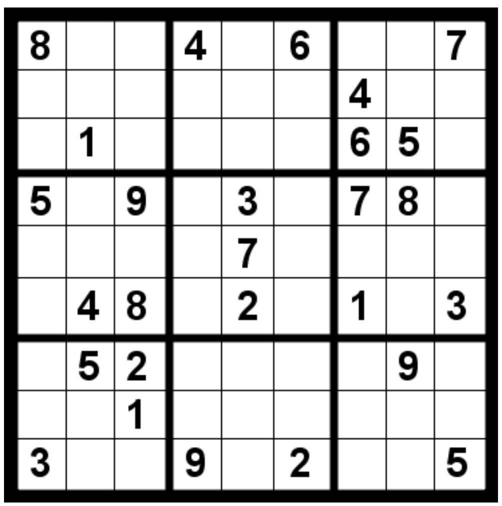
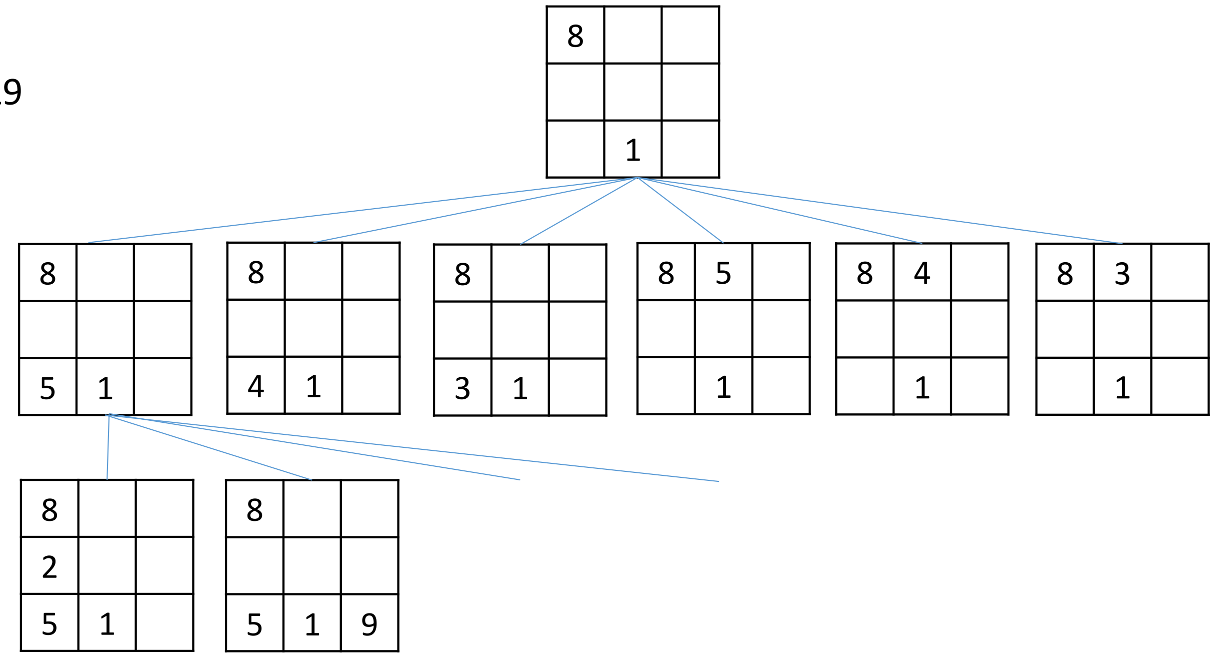
CS440/ECE 448, Lecture 6: Constraint Satisfaction Problems

Slides by Svetlana Lazebnik, 9/2016

Modified by Mark Hasegawa-Johnson, 1/2019



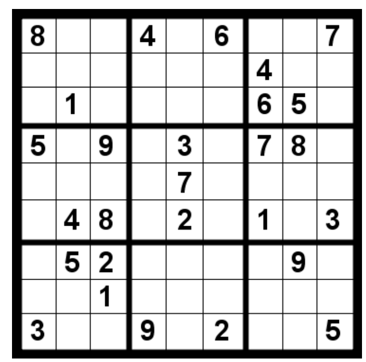


Content

- What is a CSP? Why is it search? Why is it special?
- Examples: Map Task, N-Queens, Crytparithmetic, Classroom Assignment
- Formulation as a standard search
- Backtracking Search
- Heuristics to improve backtracking search
- Tree-structured CSPs
- NP-completeness of CSP in general; the SAT problem
- Local search, e.g., hill-climbing

What is search for?

- Assumptions: single agent, deterministic, fully observable, discrete environment
- **Search for *planning***
  - The path to the goal is the important thing
  - Paths have various costs, depths
- **Search for *assignment***
  - Assign values to variables while respecting certain constraints
  - The goal (complete, consistent assignment) is the important thing



Constraint satisfaction problems (CSPs)

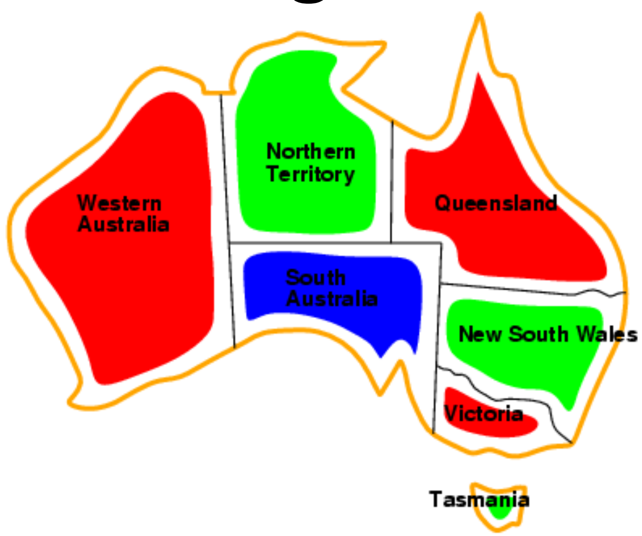
- Definition:
  - **State** is defined by N variables X_i with values from domain D_i
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables.
  - **Solution** is a complete, consistent assignment
  - True path costs are all N or ∞. Any path that works is exactly as good as any other.
- How does this compare to the "generic" tree search formulation?
  - Far more states than usual. BFS and A* are almost never computationally feasible.
  - (Hopefully) many different paths to the same solution, therefore DFS might work.
  - Structured state space allows us to use greedy search with really good heuristics.

Examples

Example: Map Coloring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** {red, green, blue}
- **Constraints:** adjacent regions must have different colors
  - Logical representation: WA ≠ NT
  - Set representation: (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

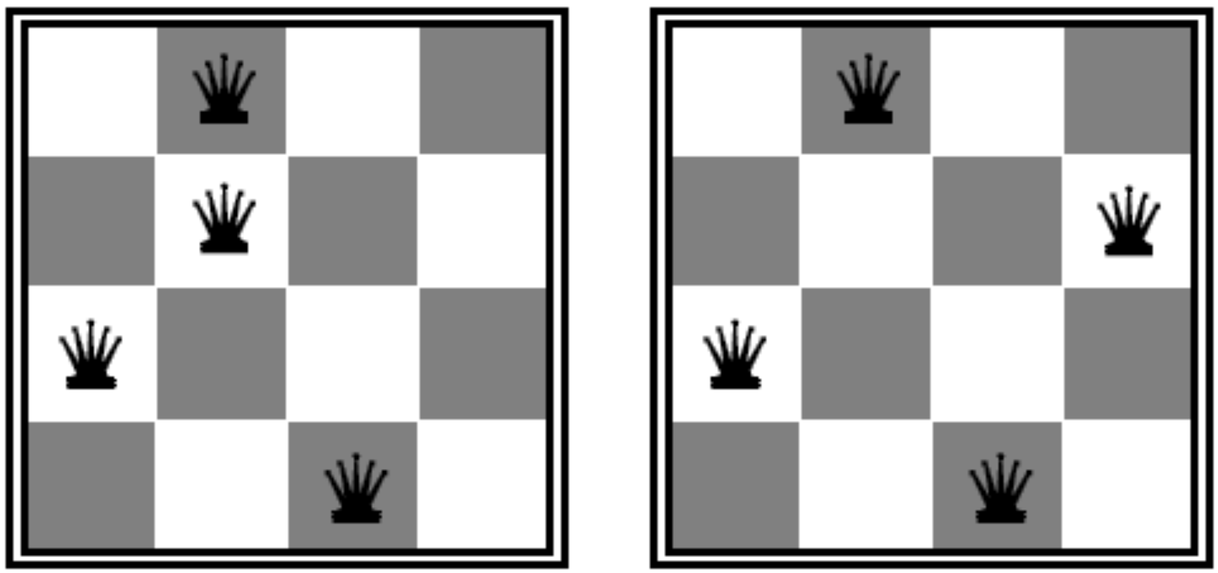
Example: Map Coloring



- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

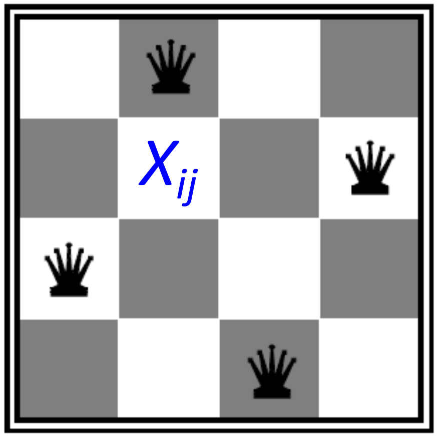
Example: *n*-queens problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



Example: N-Queens

- **Variables:** X_{ij}
- **Domains:** {0, 1}
- **Constraints:**

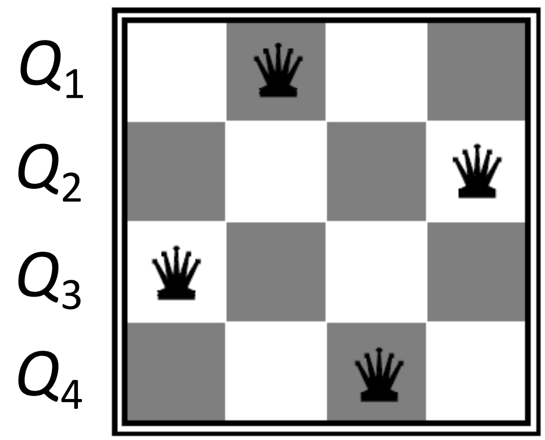


<u>**Logic**</u>	<u>**Set**</u>
$\Sigma_{i,j} X_{ij} = N$	*(??)*
$X_{ij} \wedge X_{ik} = 0$	$(X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$
$X_{ij} \wedge X_{kj} = 0$	$(X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$
$X_{ij} \wedge X_{i+k,j+k} = 0$	$(X_{ij}, X_{i+k,\, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$
$X_{ij} \wedge X_{i+k,j-k} = 0$	$(X_{ij}, X_{i+k,\, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$

N-Queens: Alternative formulation

- **Variables:** Q_i
- **Domains:** $\{1, \ldots, N\}$
- **Constraints:**

  $\forall$ i, j non-threatening (Q_i, Q_j)

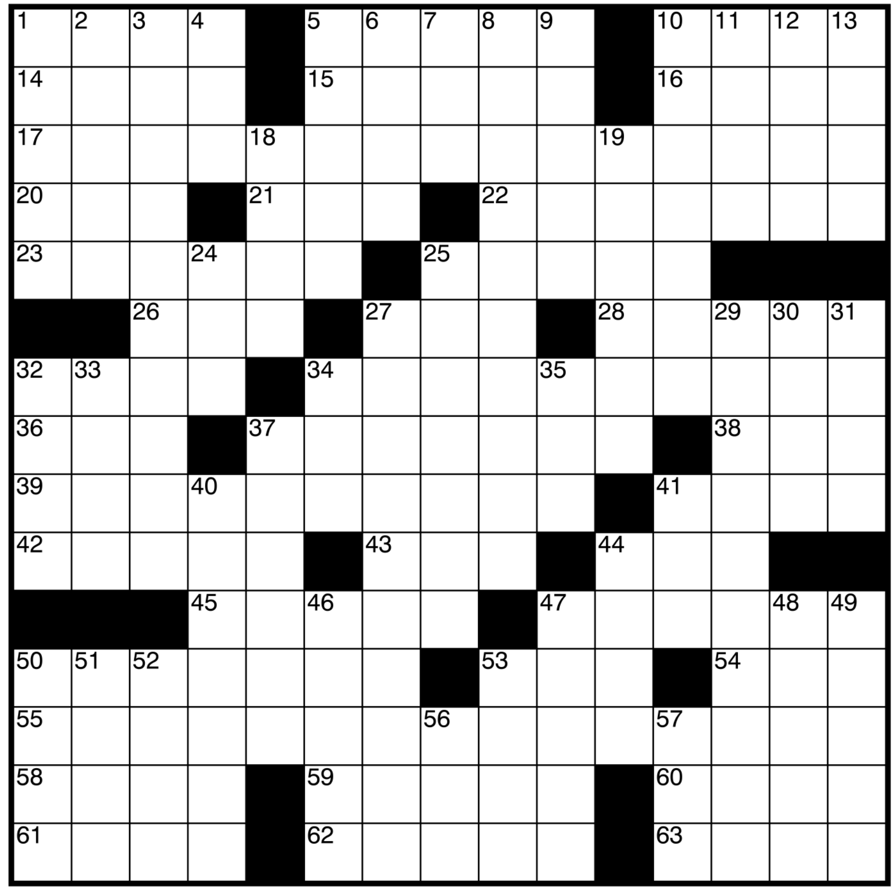


Example: Crossword Puzzle

- **Variables:** 193 squares
- **Domains**: {a,b,…,z}
- **Constraints:**

  Each row-segment is a word from the dictionary.

  Each column-segment is a word from the dictionary.



Example: Cryptarithmetic

- **Variables:** T, W, O, F, U, R, X, Y
- **Domains**: {0, 1, 2, …, 9}
- **Constraints:**

  O + O = R + 10 * Y

  W + W + Y = U + 10 * X

  T + T + X = 10 * F

  Alldiff(T, W, O, F, U, R, X, Y)

  T ≠ 0, F ≠ 0, X ≠ 0

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

Real-world CSPs

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

- More examples of CSPs: http://www.csplib.org/

Formulation as a standard search

Standard search formulation (incremental)

- **States:**
  - Variables and values assigned so far
- **Initial state:**
  - The empty assignment
- **Action:**
  - Choose any unassigned variable and assign to it a value that does not violate any constraints
    - Fail if no legal assignments
- **Goal test:**
  - The current assignment is complete and satisfies all constraints

Standard search formulation (incremental)

- What is the depth of any solution (assuming ***N*** variables)?
  Answer: *N* (this is good)
- Given that there are ***D*** possible values for any variable, how many paths are there in the search tree?
  Answer: $N!\ D^N$ (this is bad)
- All paths have the same depth, so complexity of DFS and BFS are the same (both O{$N!\ D^N$})
- Other reasons to use DFS:
  - There are usually many paths to the solution (at least ***N!***)
  - Often, if a path fails, we can detect this early
- Today's goal: develop heuristics to reduce the branching factor

Backtracking search

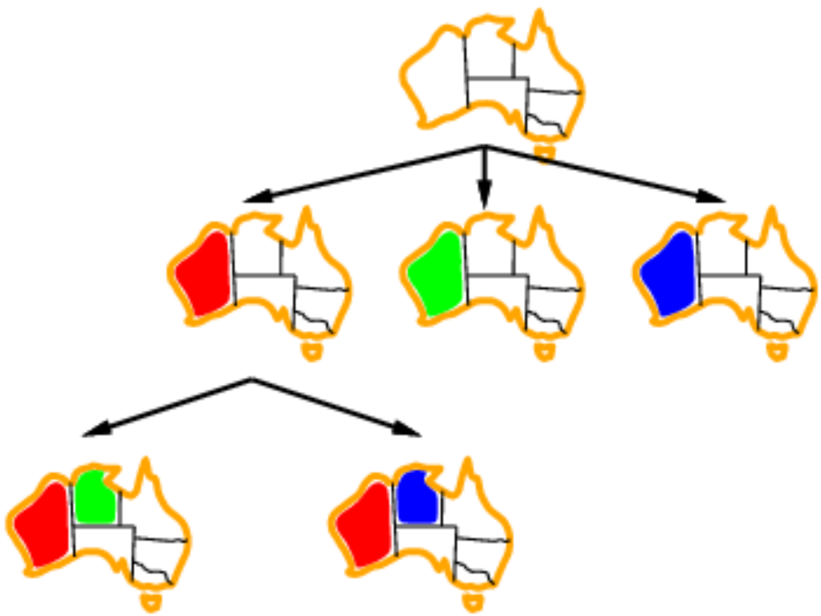
Backtracking search

- In CSP's, variable assignments are **commutative**
  - For example, *[WA = red then NT = green]* is the same as *[NT = green then WA = red]*
- We only need to consider assignments to a single variable at each level (i.e., we fix the order of assignments)
  - Then there are only D^N paths. We have eliminated the ***N!*** redundancy by arbitrarily choosing an order in which to assign variables.
- Depth-first search for CSPs with single-variable assignments is called **backtracking search**

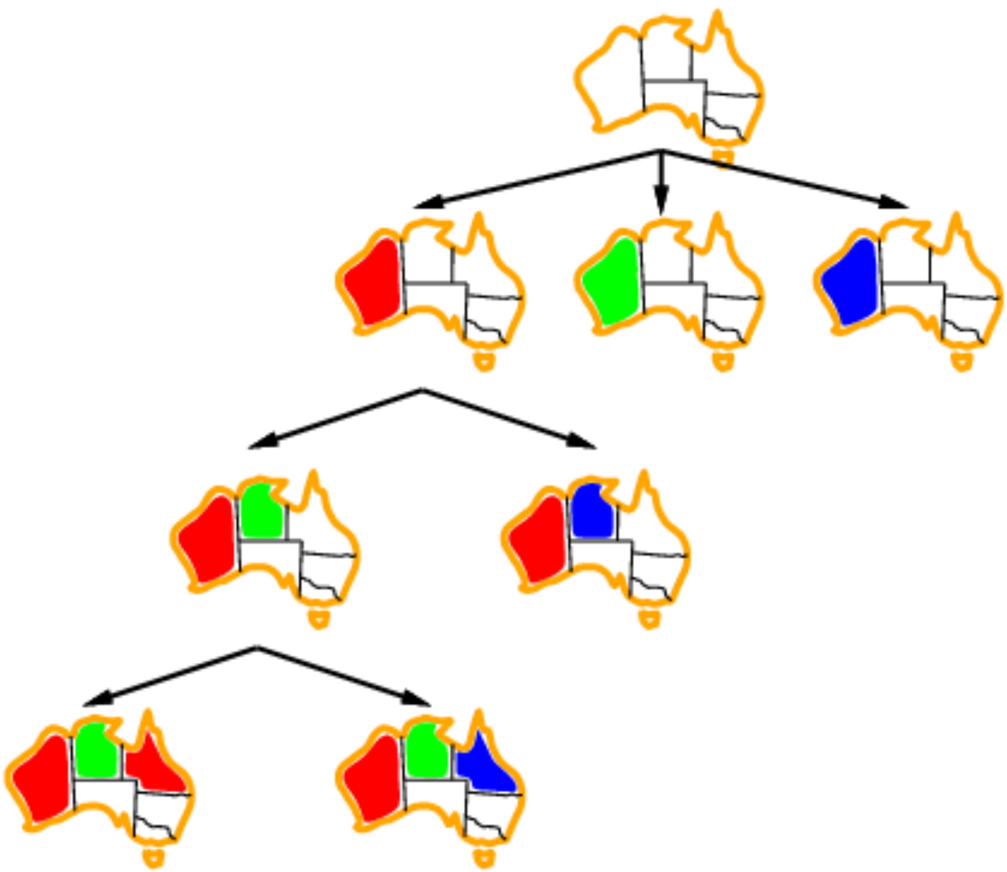
Example

Example

Example



Example



Backtracking search algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  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)
     if value is consistent with assignment given CONSTRAINTS[csp]
        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result ≠ failure then return result
        remove {var = value} from assignment
  return failure
```

- Making backtracking search efficient:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Heuristics for making backtracking search more efficient

Heuristics for making backtracking search more efficient

Still DFS, but we use heuristics to decide which child to expand first.
You could call it GDFS…

- Heuristics that choose the next variable to assign:
  - Least Remaining Values (LRV)
  - Most Constraining Variable (MCV)
- Heuristic that chooses a value for that variable:
  - Least Constraining Assignment (LCA)
- Early detection of failure:
  - Forward Checking
  - Arc Consistency

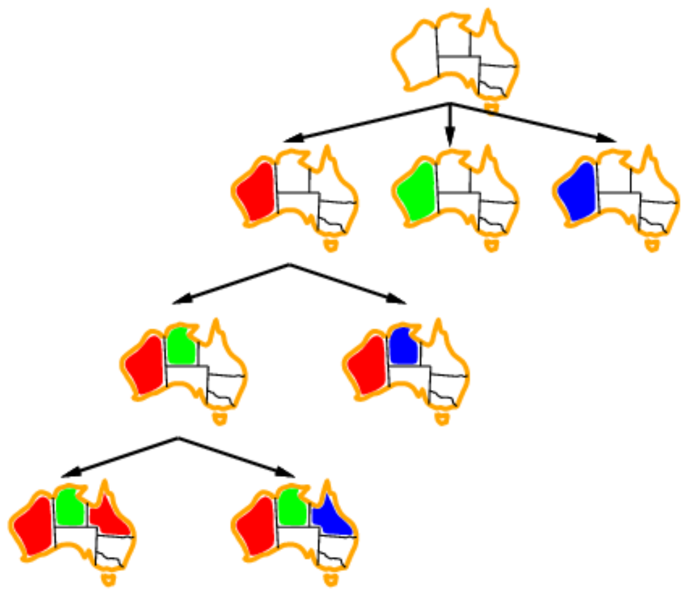
Which variable should be assigned next?

- **Least Remaining Values (LRV) Heuristic:**
  - Choose the variable with the fewest legal values

Which variable should be assigned next?

- **Least Remaining Values (LRV) Heuristic:**
  - Choose the variable with the fewest legal values

??



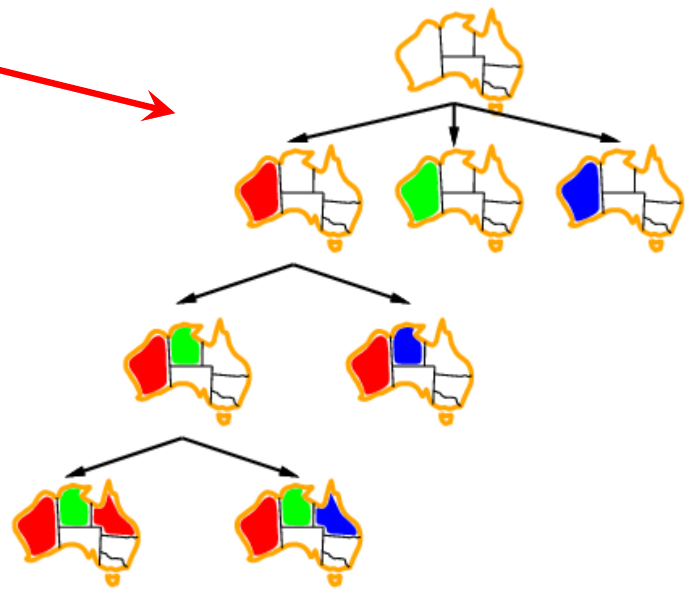
Which variable should be assigned next?

- **Most Constraining Variable (MCV) Heuristic:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among variables that have equal numbers of LRV

Which variable should be assigned next?

- **Most Constraining Variable (MCV) Heuristic:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among variables that have equal numbers of MRV

??



Given a variable, in which order should its values be tried?

- **Least Constraining Assignment (LCA) Heurstic**:
  - Try the following assignment first: to the variable you're studying, the value that rules out the fewest values in the remaining variables

Given a variable, in which order should its values be tried?

- **Least Constraining Assignment (LCA) Heurstic**:
    - Try the following assignment first: to the variable you're studying, the value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?

Early detection of failure

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  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)
     if value is consistent with assignment given CONSTRAINTS[csp]
        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result ≠ failure then return result
        remove {var = value} from assignment
  return failure
```

Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure: O{N} checking

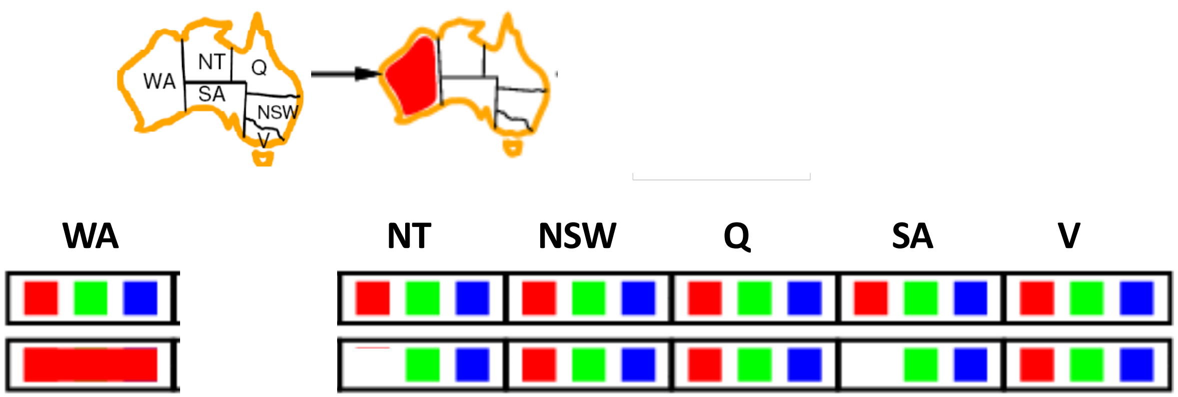
- **Forward Checking:**
  - Check to make sure that every variable still has at least one possible assignment

Early detection of failure: O{N} checking
Forward checking

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

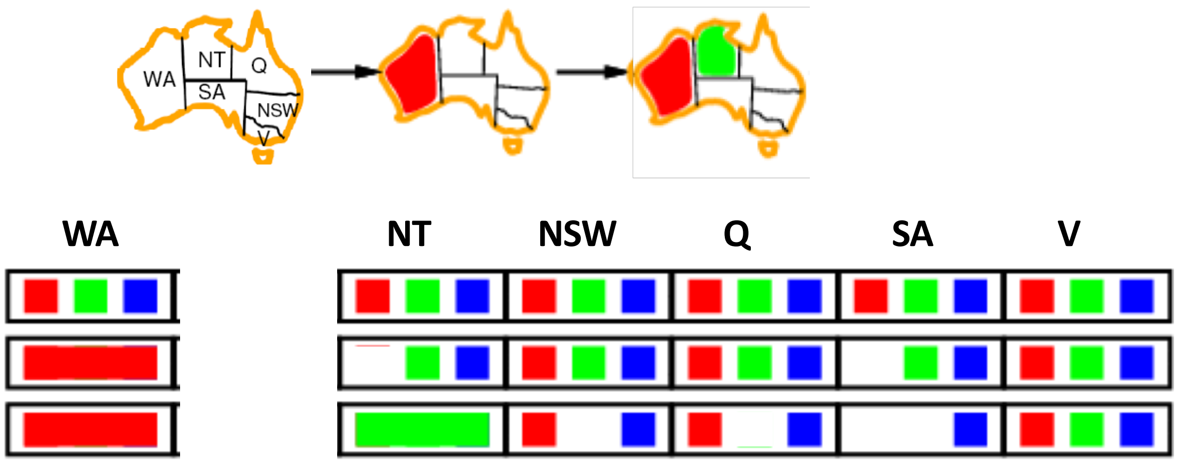
Early detection of failure: O{N} checking
Forward checking

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



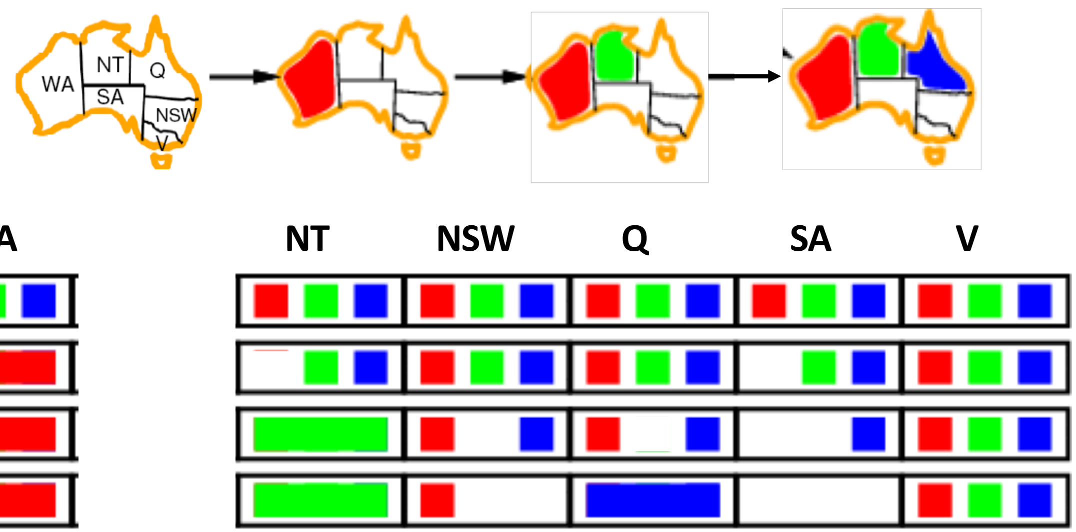
Early detection of failure: O{N} checking
Forward checking

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



Early detection of failure: O{N} checking
Forward checking

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



Early detection of failure: O{N^2} checking

- **Constraint propagation:**
  - Check to make sure that every PAIR of variables still has a pair-wise assignment that satisfies all constraints

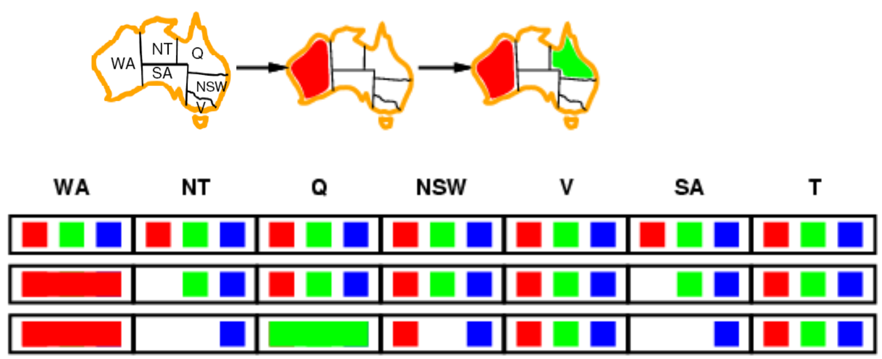
Early detection of failure: O{N^2} checking

Apply *inference* to reduce the space of possible assignments and detect failure early

(Reminder: there are only three colors, RGB…)

Constraint propagation

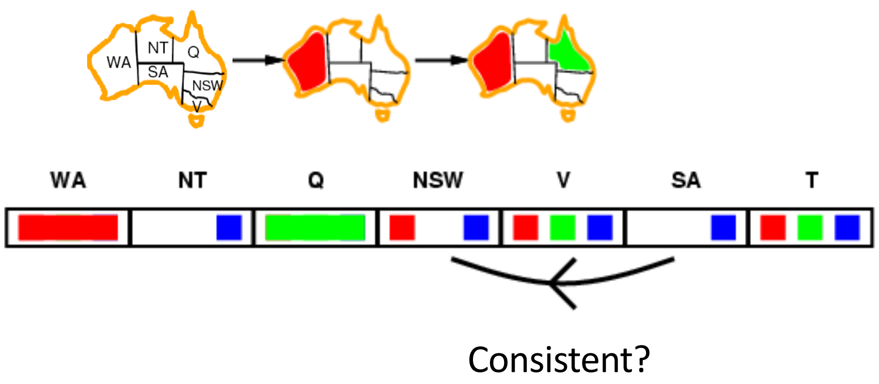
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



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

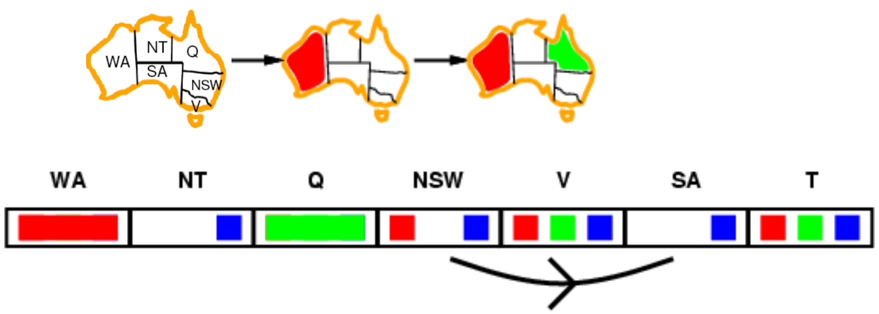
Constraint propagation algorithm: Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Constraint propagation algorithm: Arc consistency

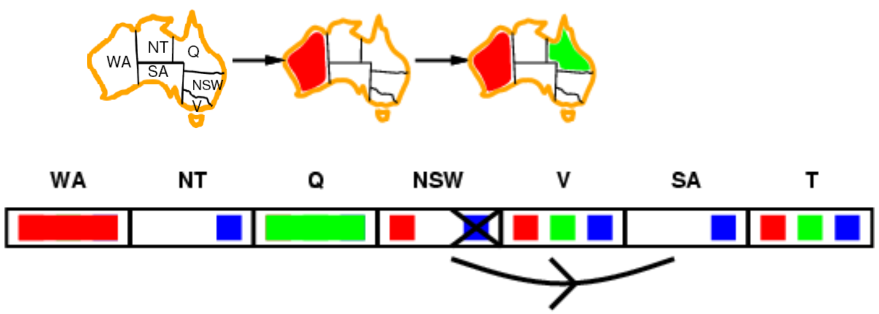
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*



Consistent?

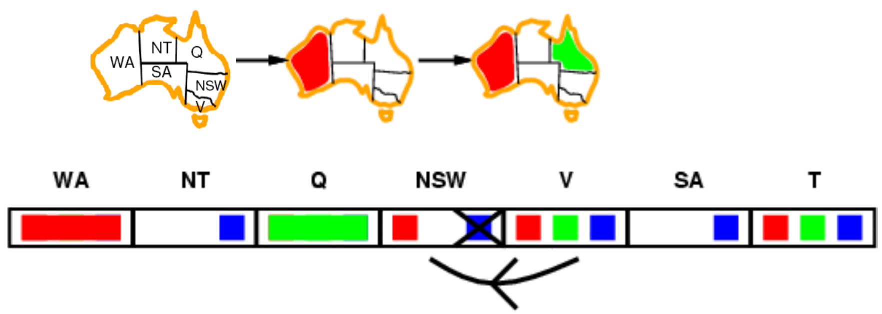
Constraint propagation algorithm: Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



Constraint propagation algorithm: Arc consistency

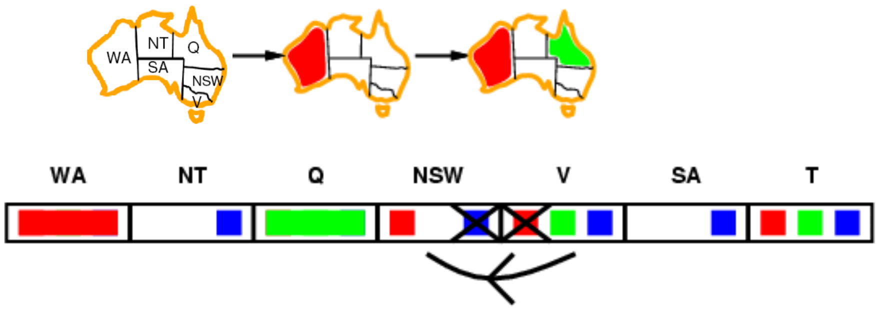
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Constraint propagation algorithm: Arc consistency

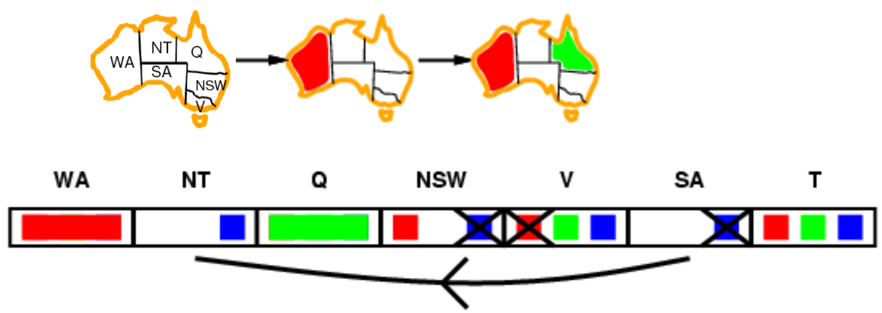
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- If *X* loses a value, all pairs *Z* → *X* need to be rechecked

Constraint propagation algorithm: Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

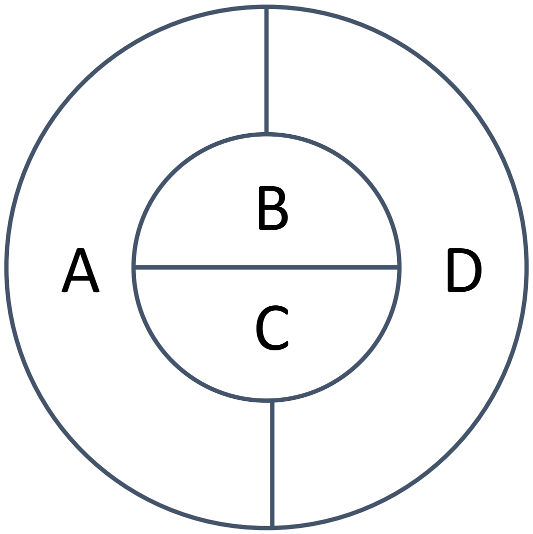
Arc consistency algorithm AC-3

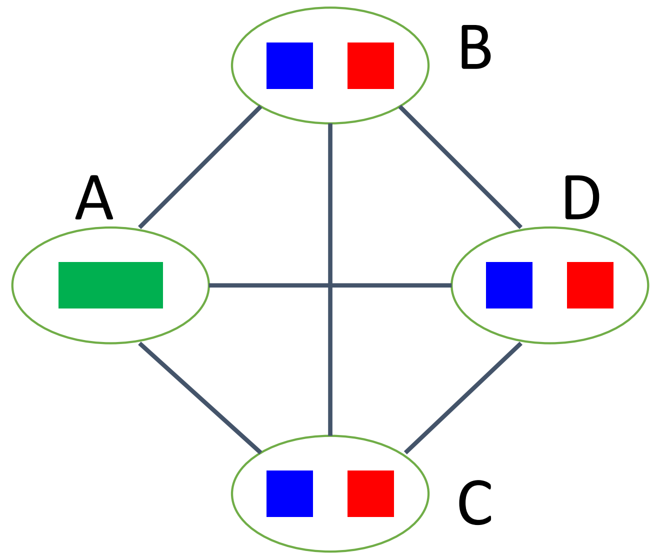
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi]
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

Does arc consistency always detect the lack of a solution?





- There exist stronger notions of consistency (path consistency, k-consistency), but we won’t worry about them

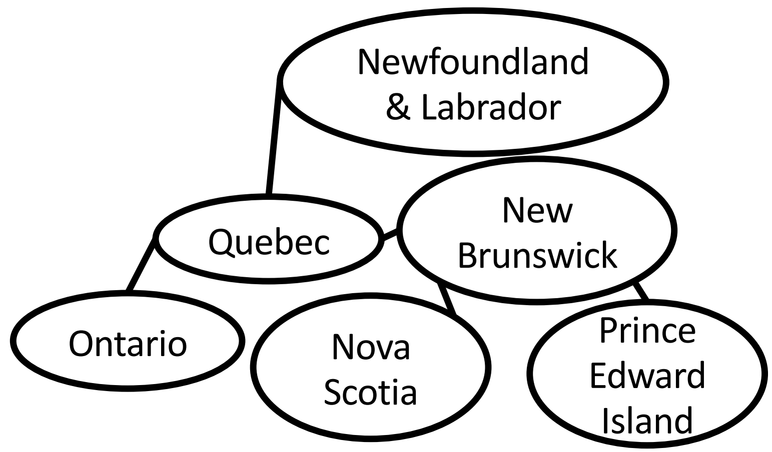
Tree-structured CSPs

Tree-structured CSPs

- Certain kinds of CSPs can be solved without resorting to backtracking search!

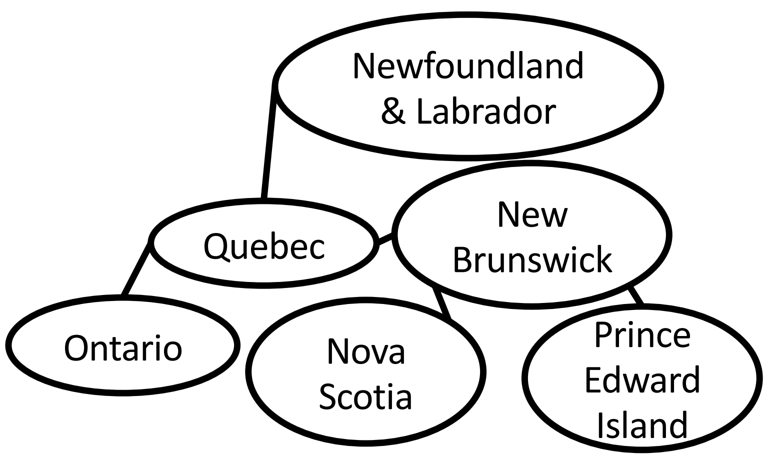
- *Tree-structured CSP*: constraint graph does not have any loops

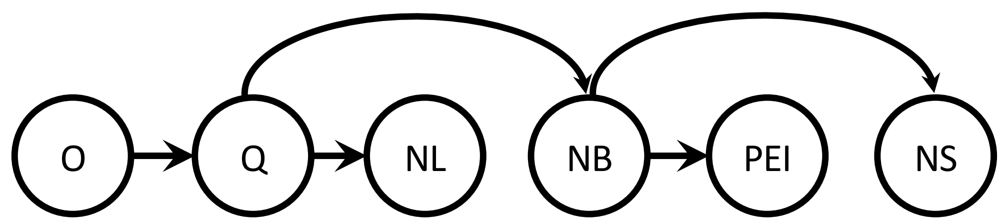
Map released to public domain by E Pluribus Anthony



Algorithm for tree-structured CSPs

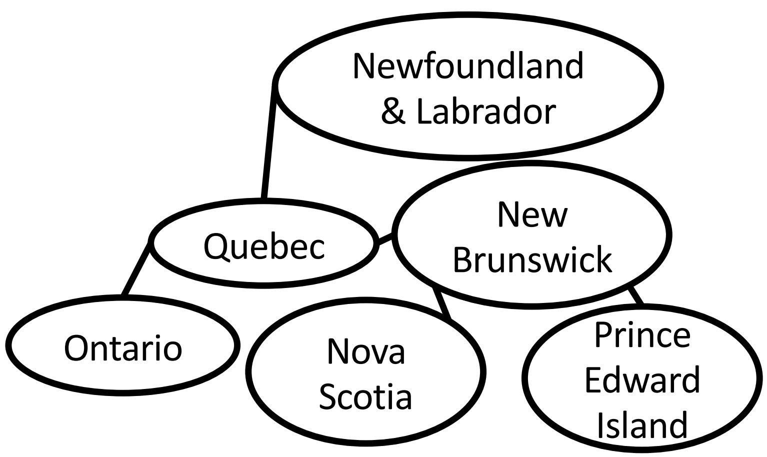
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering.

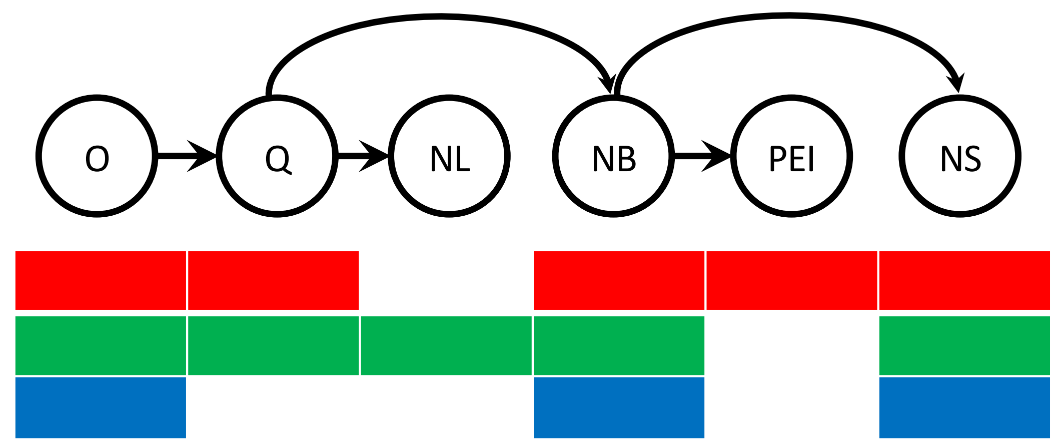




Algorithm for tree-structured CSPs

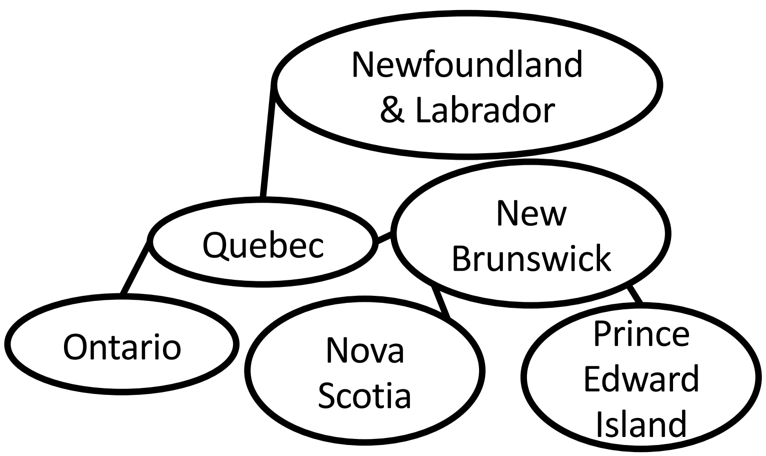
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Create a graph listing all of the values that can be assigned to each variable
  - SUPPOSE: Newfoundland wants to be green
  - Quebec doesn’t want to be blue
  - PEI wants to be red

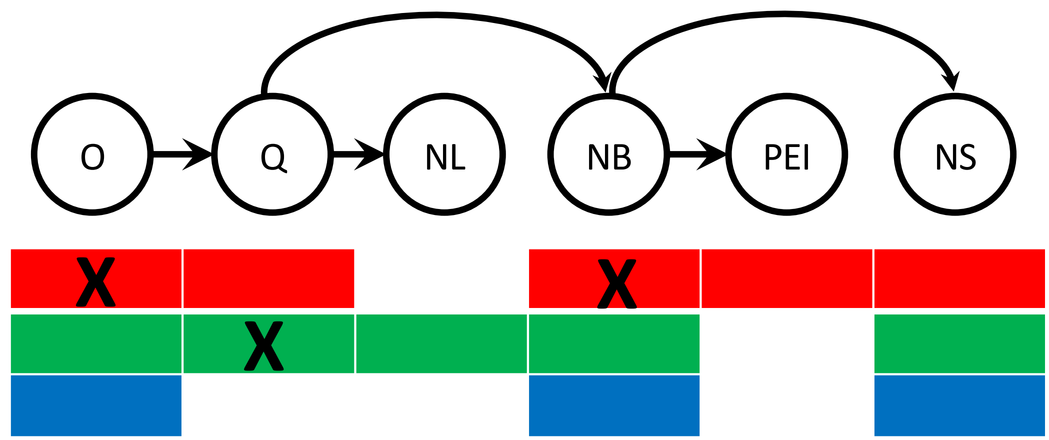




Algorithm for tree-structured CSPs

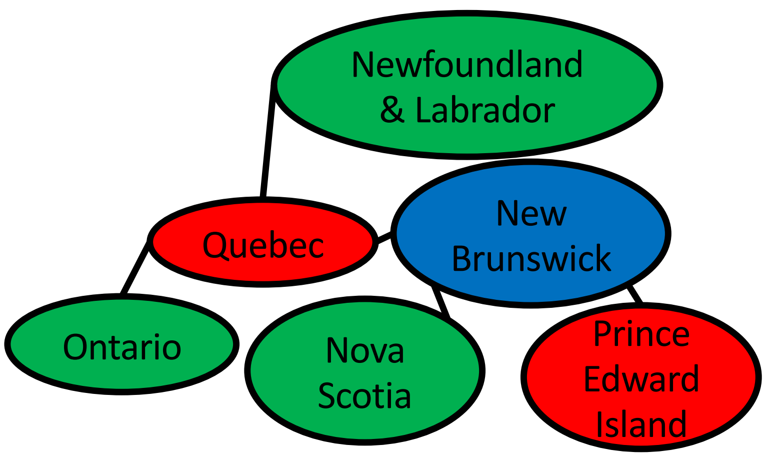
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Create a graph listing all of the values that can be assigned to each variable
- BACKWARD ARC CONSISTENCY: check arc consistency starting from the rightmost node and going backwards

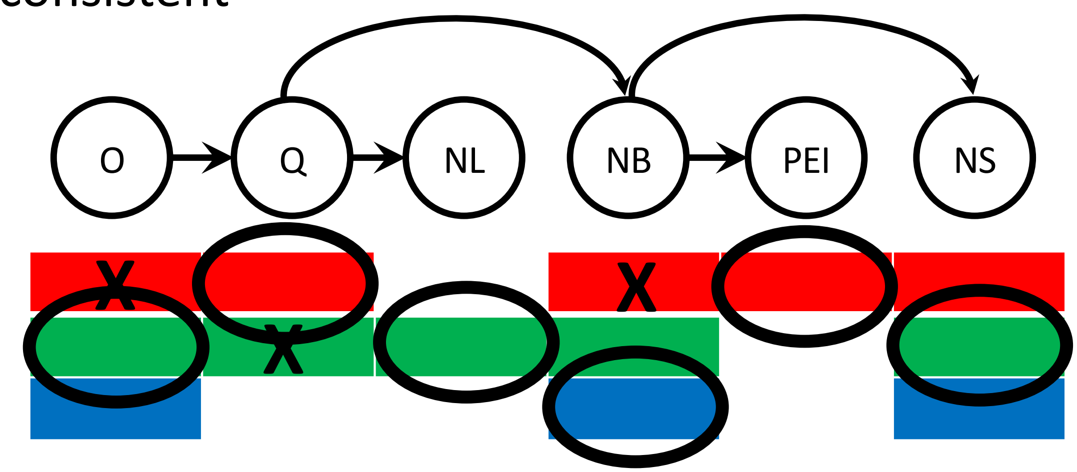




Algorithm for tree-structured CSPs

- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Create a graph listing all of the values that can be assigned to each variable
- BACKWARD ARC CONSISTENCY: check arc consistency starting from the rightmost node and going backwards
- FORWARD ASSIGNMENT PHASE: select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is consistent

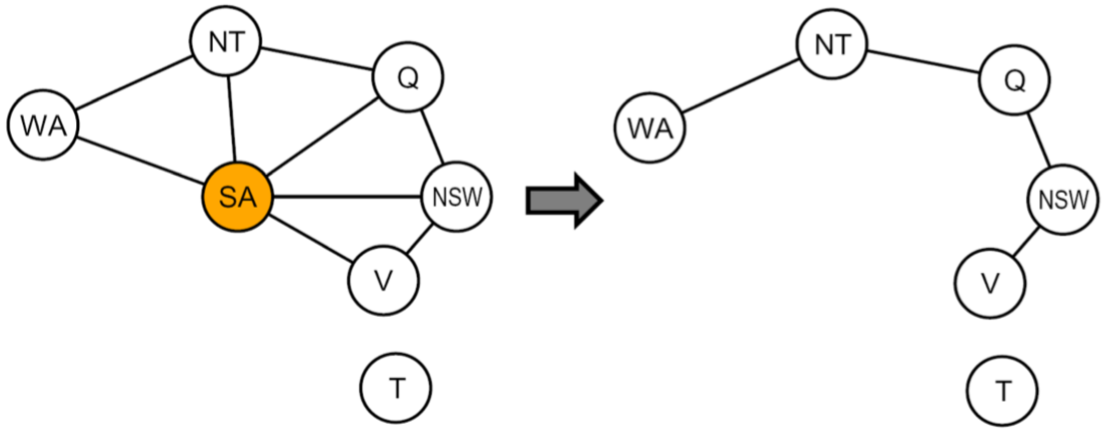




Algorithm for tree-structured CSPs

- If N is the number of variables and D is the domain size, what is the running time of this algorithm?
  - O(ND^2): we have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values

Nearly tree-structured CSPs



- **Cutset conditioning:** find a subset of variables whose removal makes the graph a tree, instantiate that set in all possible ways, prune the domains of the remaining variables and try to solve the resulting tree-structured CSP
- Cutset size c gives runtime $O(D^c (N - c) D^2)$

Source: P. Abbeel, D. Klein, L. Zettlemoyer

NP-Completeness and the SAT Problem

Algorithm for tree-structured CSPs

- Running time is $O(ND^2)$
  (N is the number of variables, D is the domain size)
  - We have to check arc consistency once for every node in the graph (every node has one parent), which involves looking at pairs of domain values
- What about backtracking search for general CSPs?
  - Worst case $O(D^N)$
- Can we do better?

Computational complexity of CSPs

- The satisfiability (SAT) problem:
  - Given a Boolean formula, is there an assignment of the variables that makes it evaluate to true?

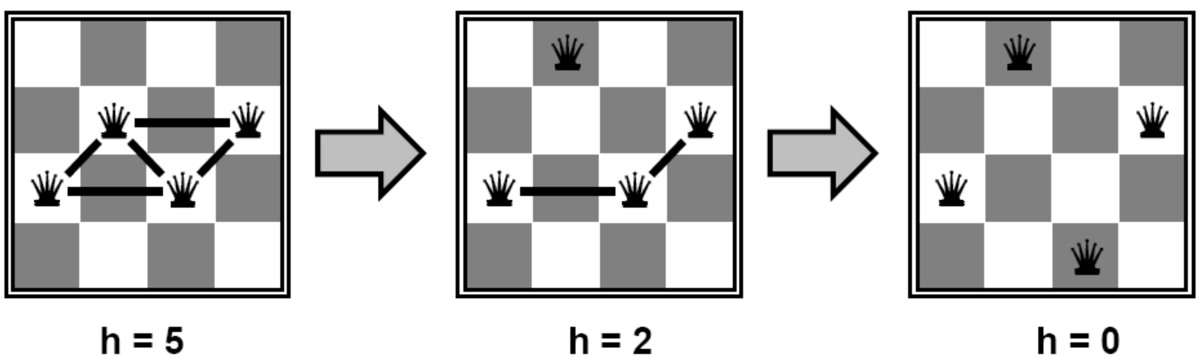
$$(X_1 \vee \overline{X}_7 \vee X_{13}) \wedge (\overline{X}_2 \vee X_{12} \vee X_{25}) \wedge \ldots$$

- SAT is *NP-complete*
  - NP: a class of decision problems for which
    - the "yes" answer can be verified in polynomial time
    - no known algorithm can find a "yes" answer, from scratch, in polynomial time
  - An NP-complete problem is in NP and every other problem in NP can be efficiently reduced to it (Cook, 1971)
  - Other NP-complete problems: graph coloring, n-puzzle, generalized sudoku
  - It is not known whether P = NP, i.e., no efficient algorithms for solving SAT in general are known

Local search, e.g., hill climbing

Local search for CSPs

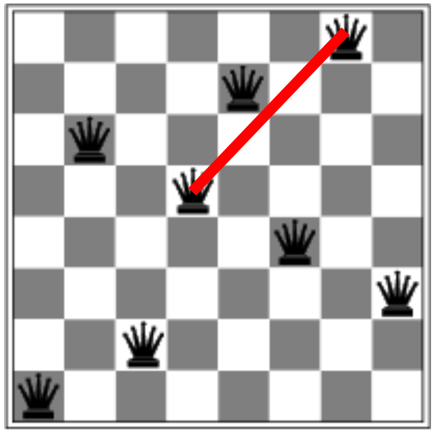
- Start with "complete" states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints



h = number of conflicts

Local search for CSPs

- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints
  - I.e., attempt to greedily minimize total number of violated constraints
  - Problem: *local minima*

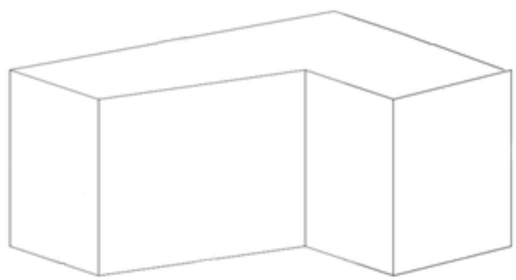


$h = 1$

Applications that look a lot like intelligence…

CSP in computer vision: Line drawing interpretation

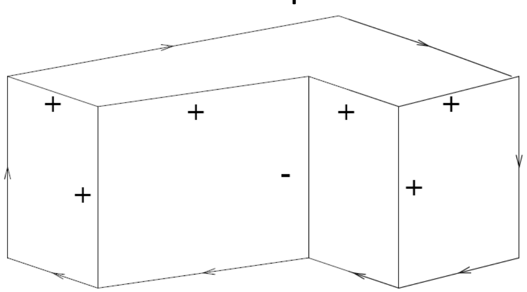
An example polyhedron:



Variables: edges

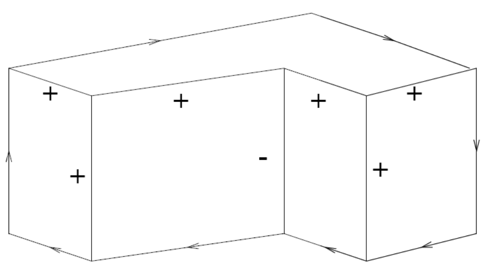
Domains: +, −, →, ←

Desired output:

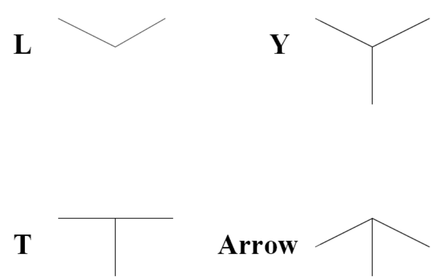


David Waltz, 1975

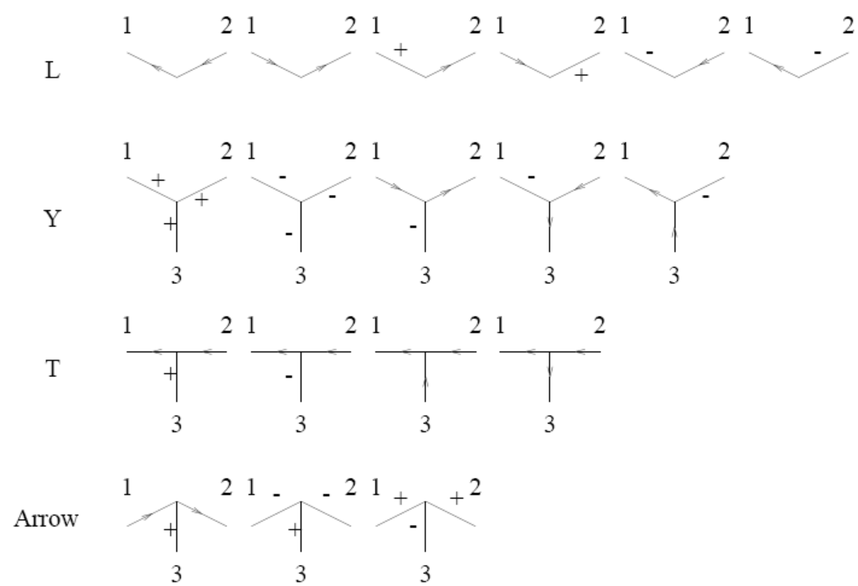
CSP in computer vision: Line drawing interpretation



Four vertex types:



Constraints imposed by each vertex type:



David Waltz, 1975

CSP in computer vision: 4D Cities

1. When was each photograph taken?
2. When did each building first appear?
3. When was each building removed?

Set of Photographs:

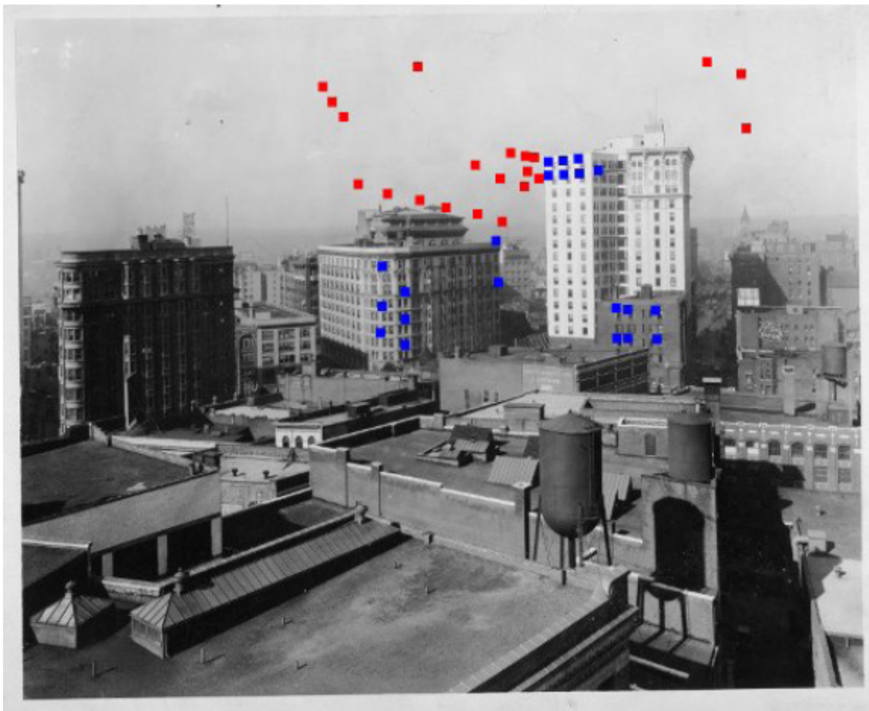
Set of Objects: Buildings

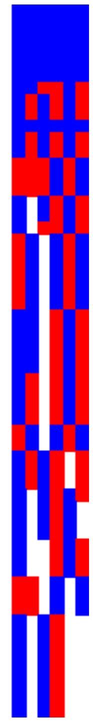
G. Schindler, F. Dellaert, and S.B. Kang, Inferring Temporal Order of Images From 3D Structure, IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), 2007.

http://www.cc.gatech.edu/~phlosoft/

CSP in computer vision: 4D Cities

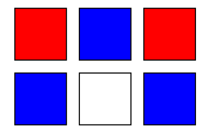
observed missing occluded



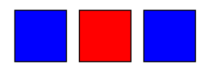


Columns: images
Rows: points

Satisfies constraints:



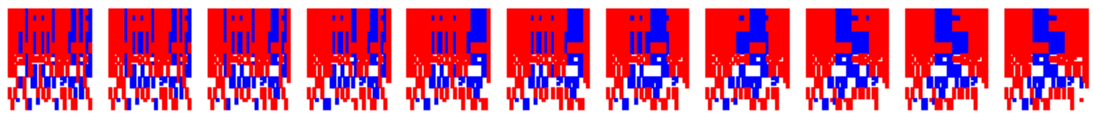
Violates constraints:



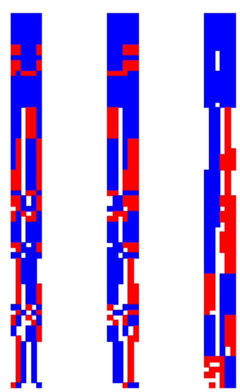
- Goal: reorder images (columns) to have as few violations as possible

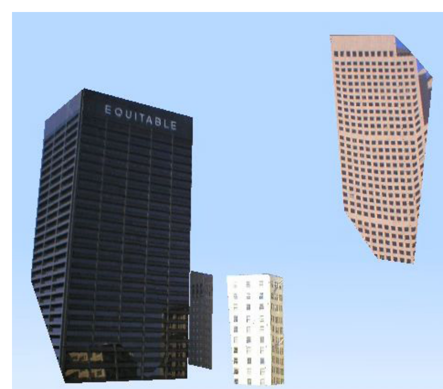
CSP in computer vision: 4D Cities

- **Goal:** reorder images (columns) to have as few violations as possible
- **Local search:** start with random ordering of columns, swap columns or groups of columns to reduce the number of conflicts



- Can also reorder the rows to group together points that appear and disappear at the same time – that gives you buildings





Summary

- CSPs are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- **Backtracking** = depth-first search where successor states are generated by considering assignments to a single variable
  - **Variable ordering** and **value selection** heuristics can 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
- Complexity of CSPs
  - NP-complete in general (exponential worst-case running time)
  - Efficient solutions possible for special cases (e.g., tree-structured CSPs)
- Alternatives to backtracking search: local search