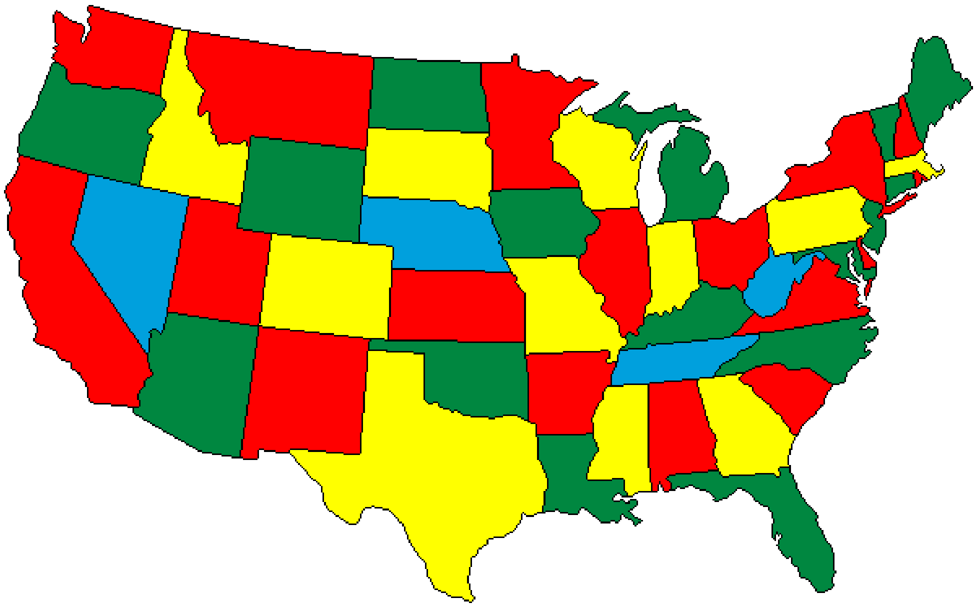
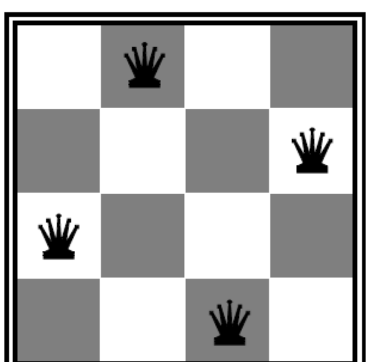
Constraint Satisfaction Problems



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



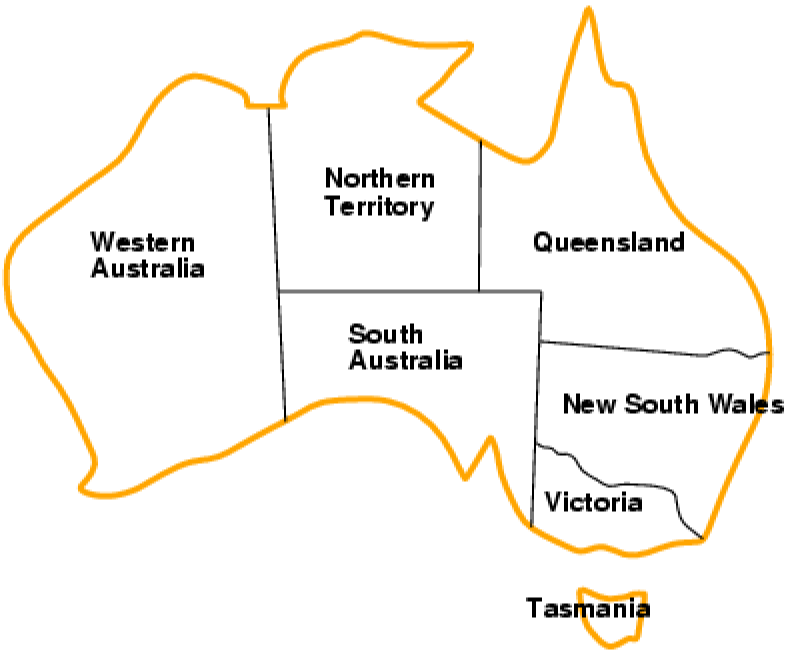
Dr. Shikha Mehta

Constraint satisfaction problems (CSPs)

- Definition:
  - **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
  - **Solution** is a complete, consistent assignment

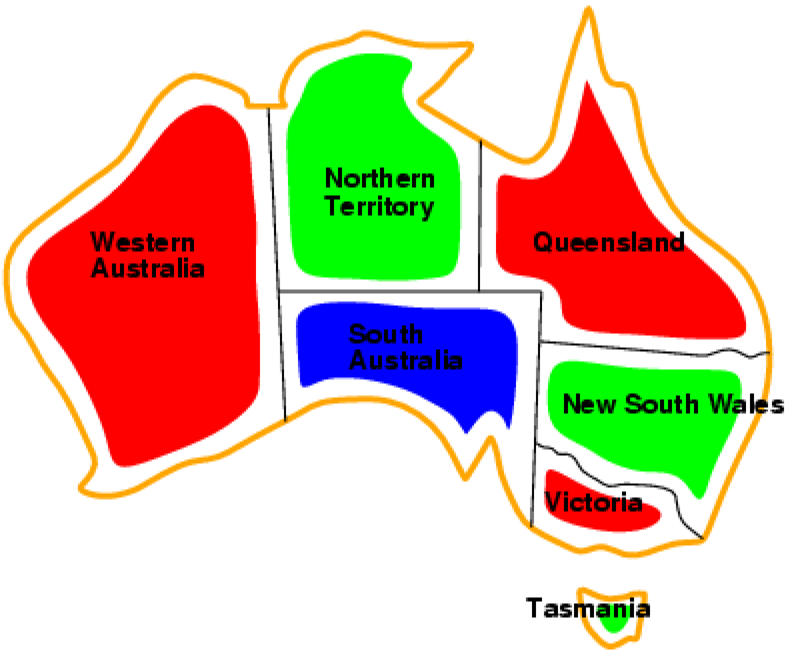
- How does this compare to the "generic" tree search formulation?
  - A more structured representation for states, expressed in a formal representation language
  - Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map Coloring



- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** {red, green, blue}
- **Constraints:** adjacent regions must have different colors
  e.g., WA ≠ NT, or (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

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

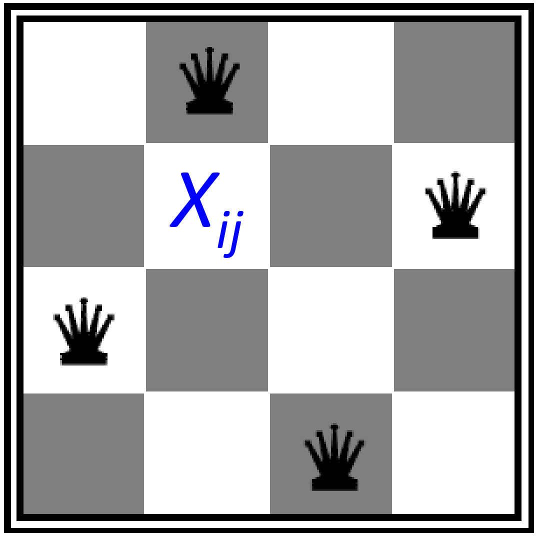
$\sum_{i,j} X_{ij} = N$

$(X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$

$(X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$

$(X_{ij}, X_{i+k,\, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$

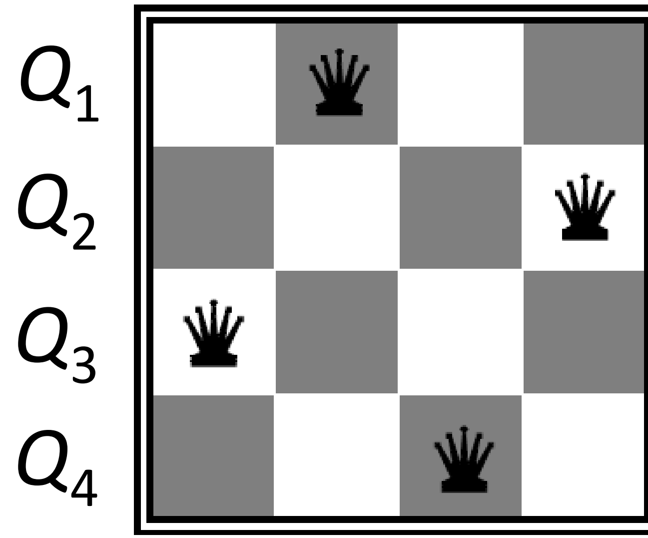
$(X_{ij}, X_{i+k,\, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$



N-Queens: Alternative formulation

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

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



Example: Cryptarithmetic

- **Variables:** T, W, O, F, U, R
  X_1, X_2
- **Domains**: {0, 1, 2, …, 9}
- **Constraints:**

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

  $O + O = R + 10 * X_1$

  $W + W + X_1 = U + 10 * X_2$

  $T + T + X_2 = O + 10 * F$

  $T \neq 0, F \neq 0$

$$\begin{array}{cccc} & X_2 & X_1 & \\ & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

Example: Sudoku

- **Variables:** X_{ij}
- **Domains:** {1, 2, …, 9}
- **Constraints:**

  Alldiff(X_{ij} in the same *unit*)

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8		X_{ij}		4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

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/

Standard search formulation (incremental)

- **States**:
  - Values assigned so far
- **Initial state:**
  - The empty assignment { }
- **Successor function:**
  - 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)?

  n – This is the good news

- Given that there are m possible values for any variable, how many paths are there in the search tree?

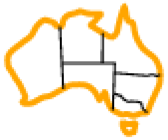
  $n! \cdot m^n$ – This is the bad news

- How can we reduce the branching factor?

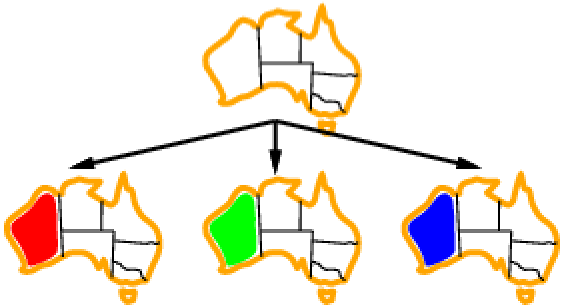
Backtracking search

- 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 m^n leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking search**

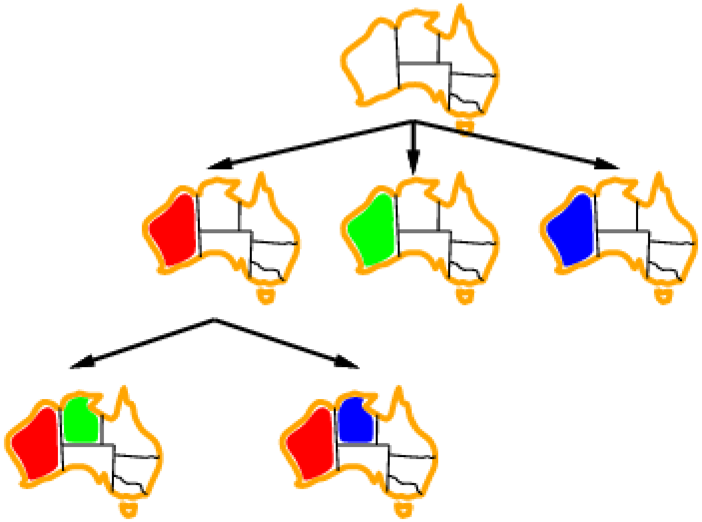
Example



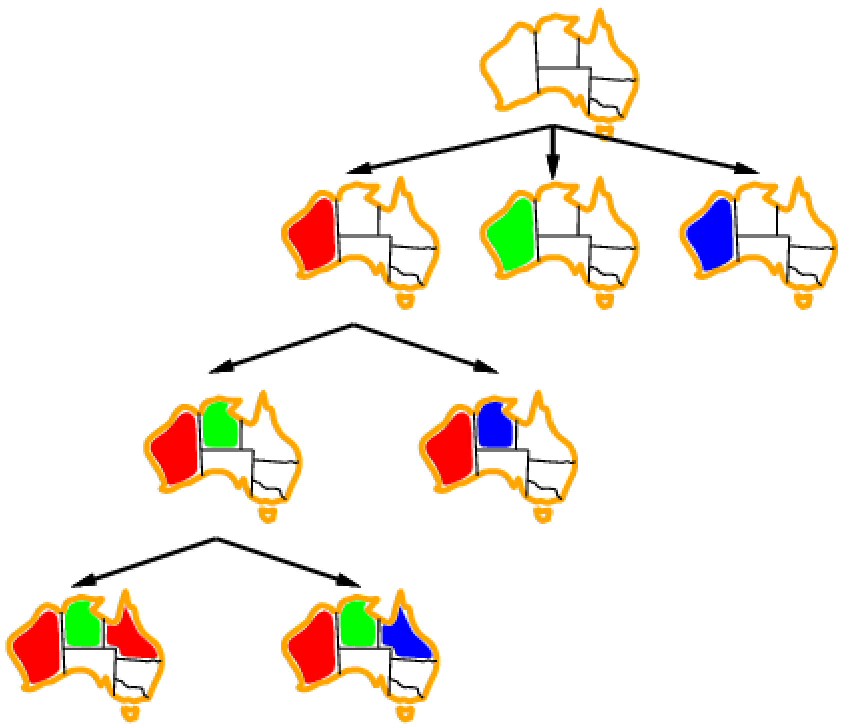
Example



Example



Example



Backtracking search algorithm

function RECURSIVE-BACKTRACKING($assignment, csp$)
    if $assignment$ is complete **then return** $assignment$
    $\boxed{var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\text{VARIABLES}[csp], assignment, csp)}$
    for each $value$ **in** $\boxed{\text{ORDER-DOMAIN-VALUES}(var, assignment, csp)}$
        if $value$ is consistent with $assignment$ given CONSTRAINTS[csp]
            add $\{var = value\}$ to $assignment$
            $result \leftarrow$ RECURSIVE-BACKTRACKING($assignment, csp$)
            if $result \neq failure$ **then return** $result$
            remove $\{var = value\}$ from $assignment$
    return $failure$

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

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constrained variable:**
  - Choose the variable with the fewest legal values
  - A.k.a. **minimum remaining values** (MRV) heuristic

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

Which variable should be assigned next?

- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

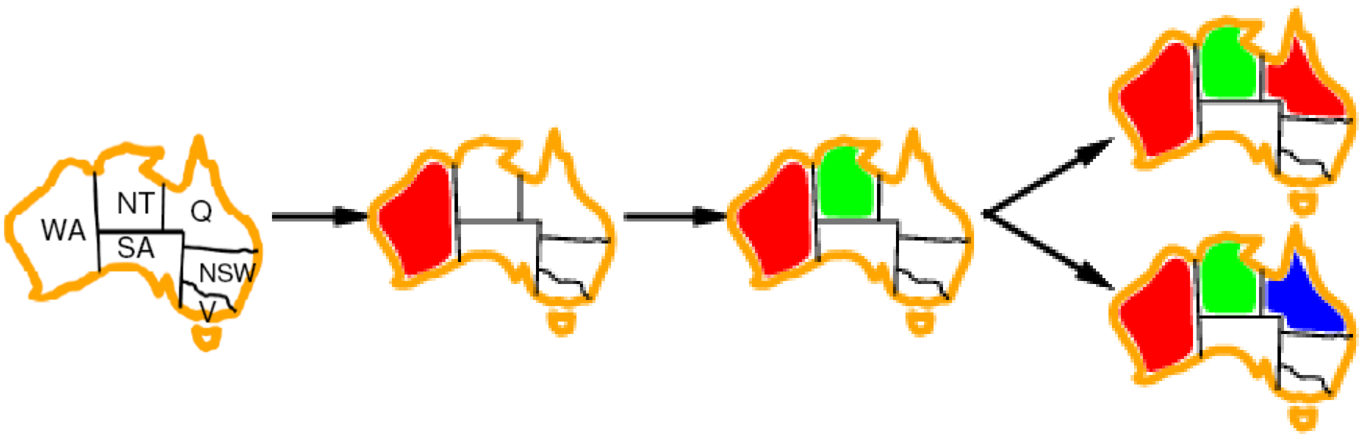
Given a variable, in which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

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

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables

Which assignment for Q should we choose?

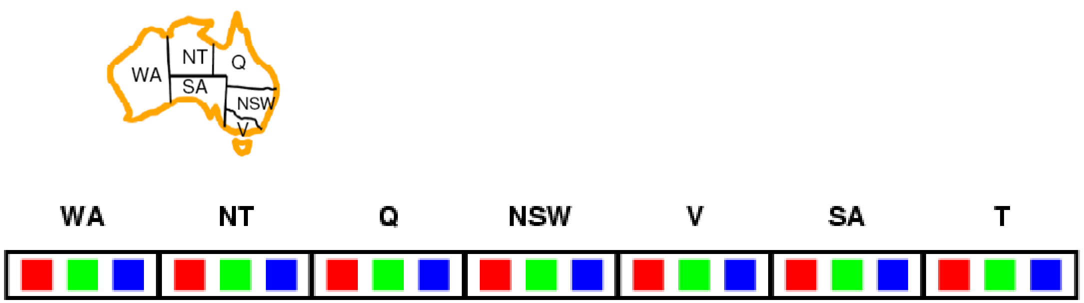


Early detection of failure: Forward checking

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

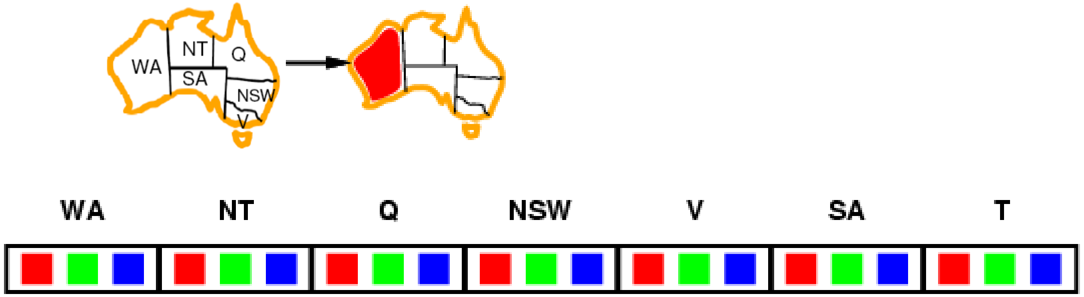
Early detection of failure: Forward checking

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



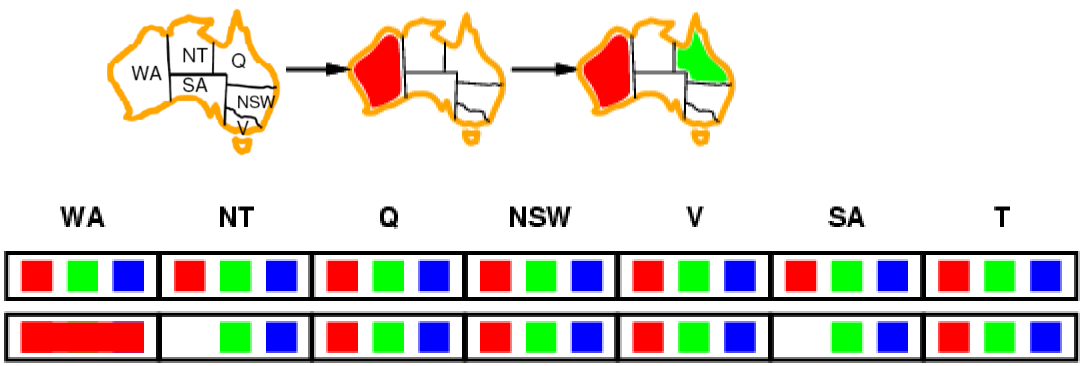
Early detection of failure: Forward checking

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



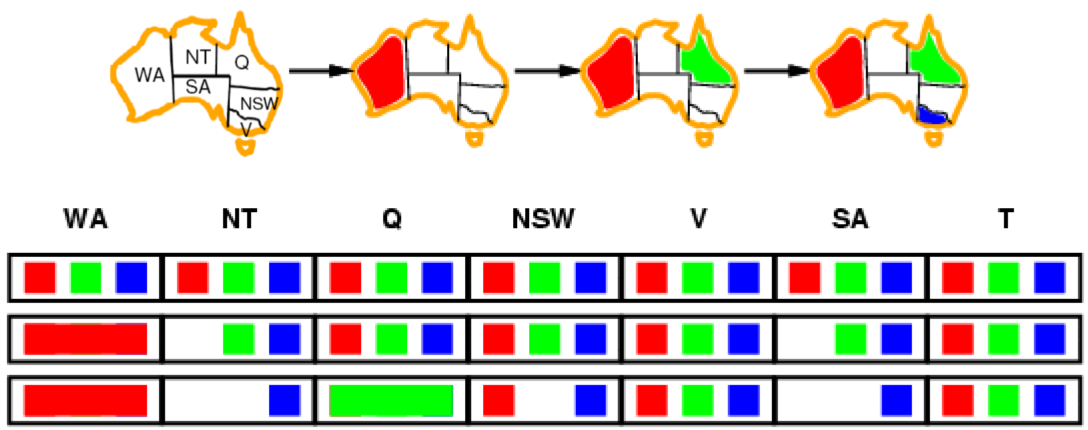
Early detection of failure: Forward checking

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



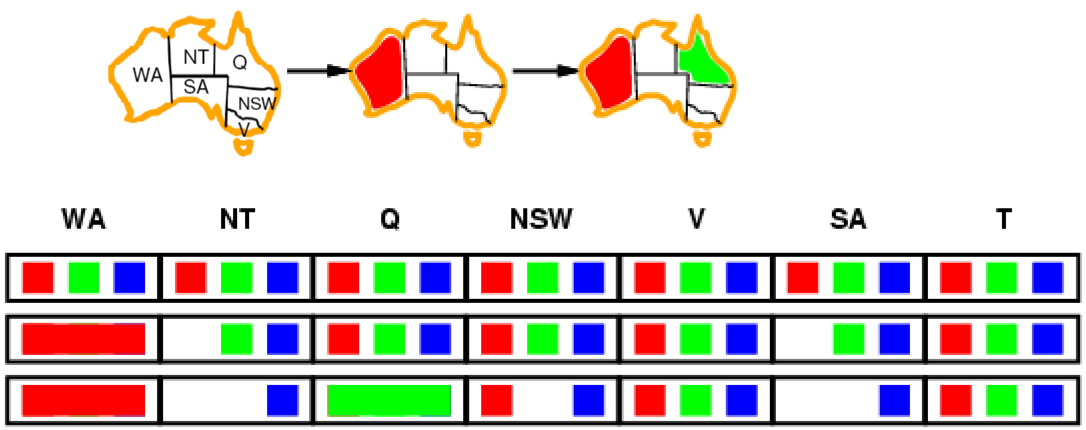
Early detection of failure: Forward checking

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



Constraint propagation

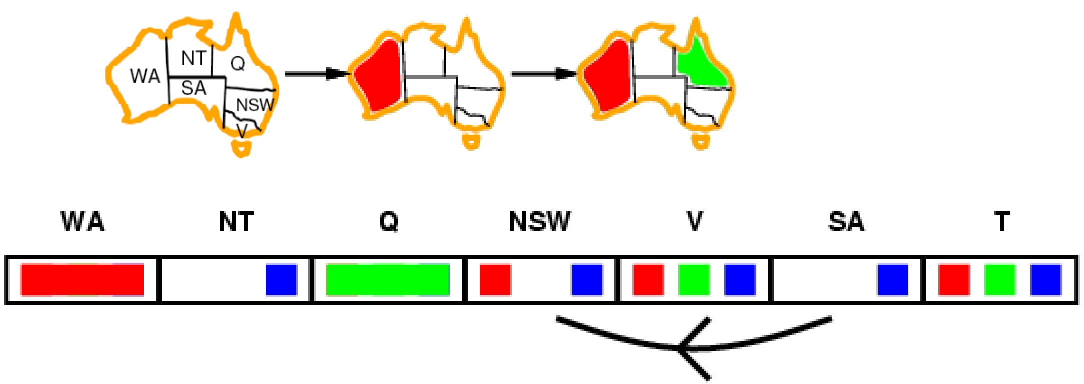
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



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

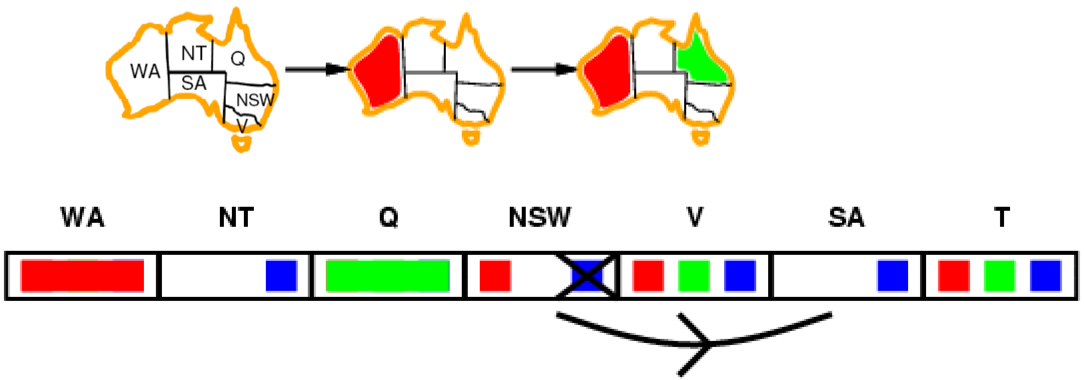
Arc consistency

- Simplest form of propagation makes each 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

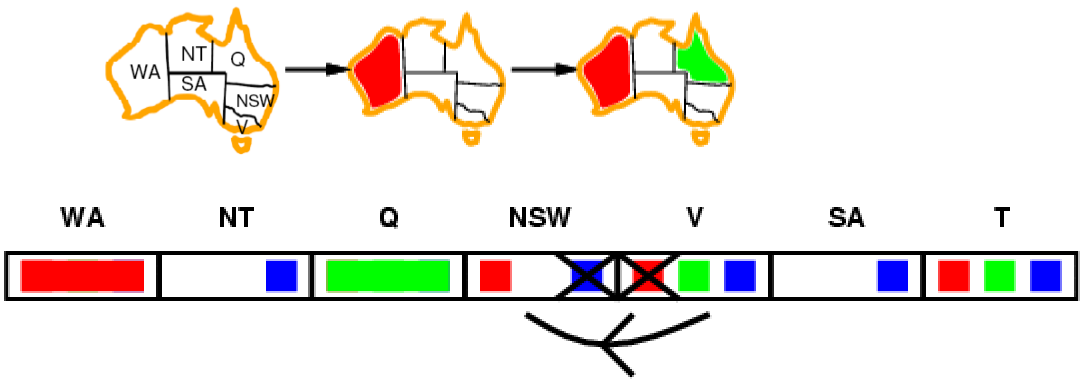
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



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

Arc consistency

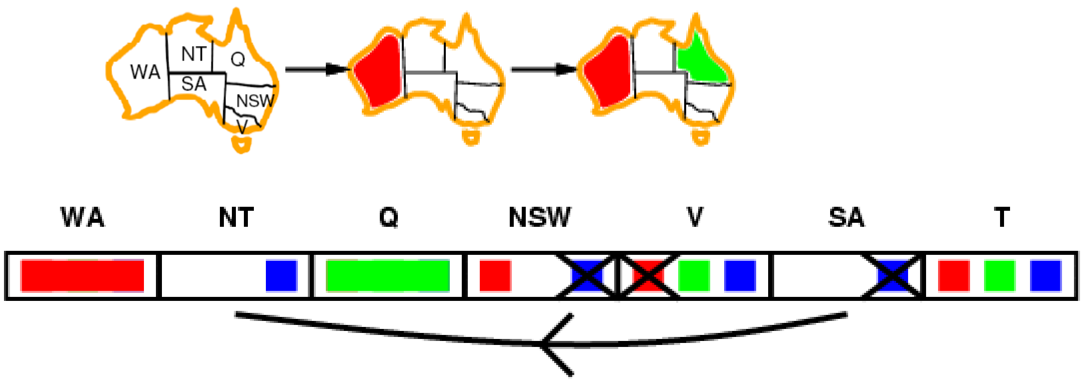
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



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

Arc consistency

- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, 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 as a preprocessor or after each assignment

Arc consistency algorithm AC-3

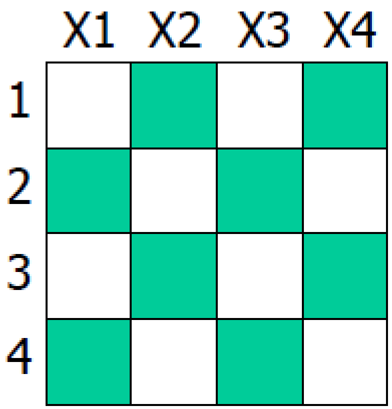
function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

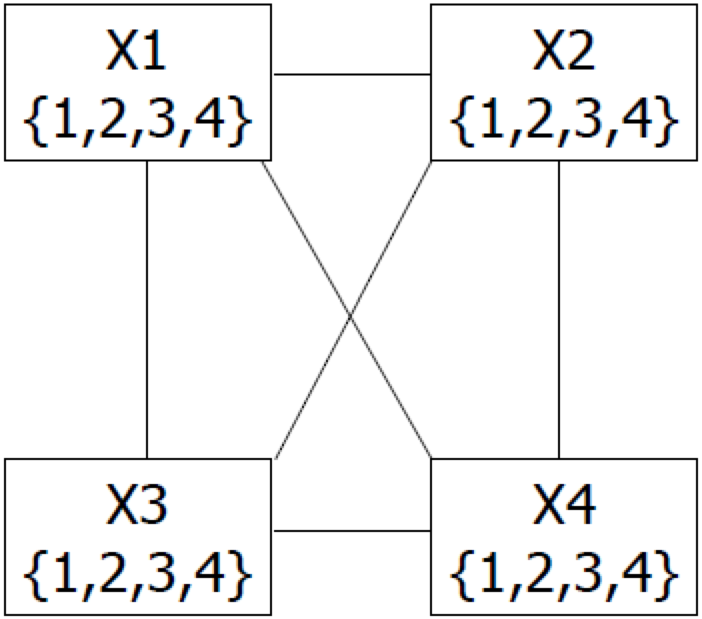
  while *queue* is not empty
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

---

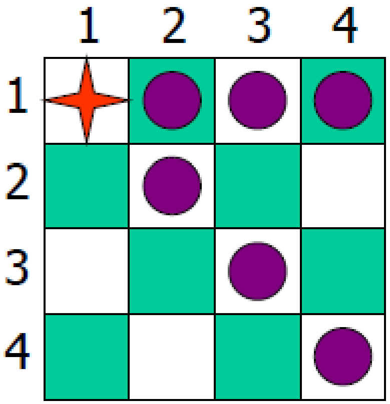
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i]
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

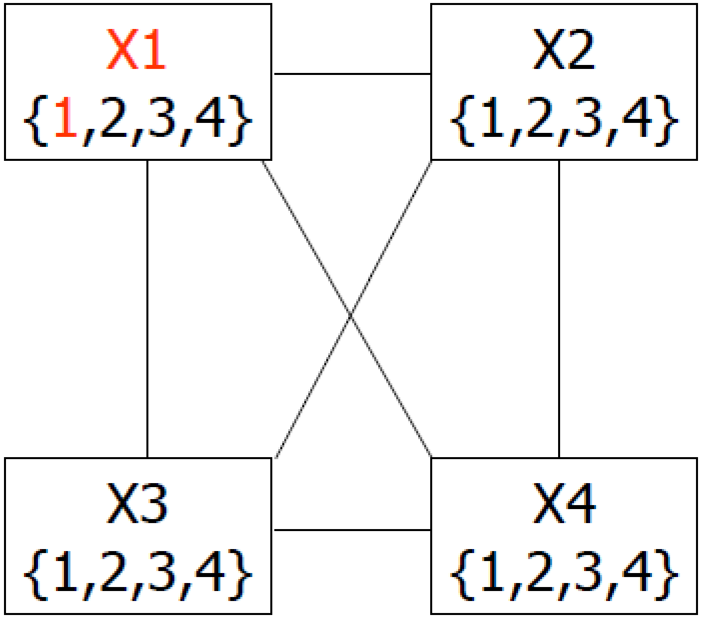
4 Queens problem

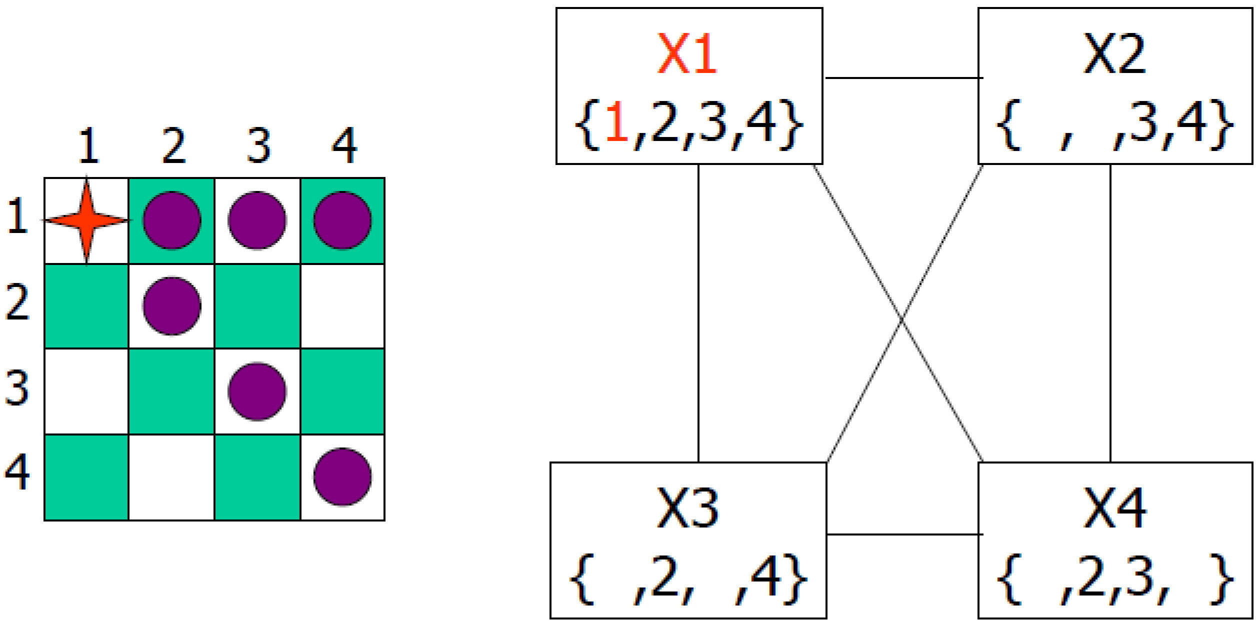


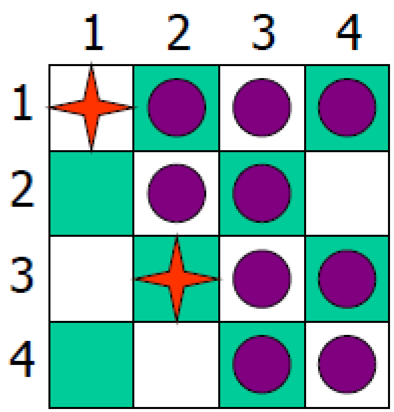


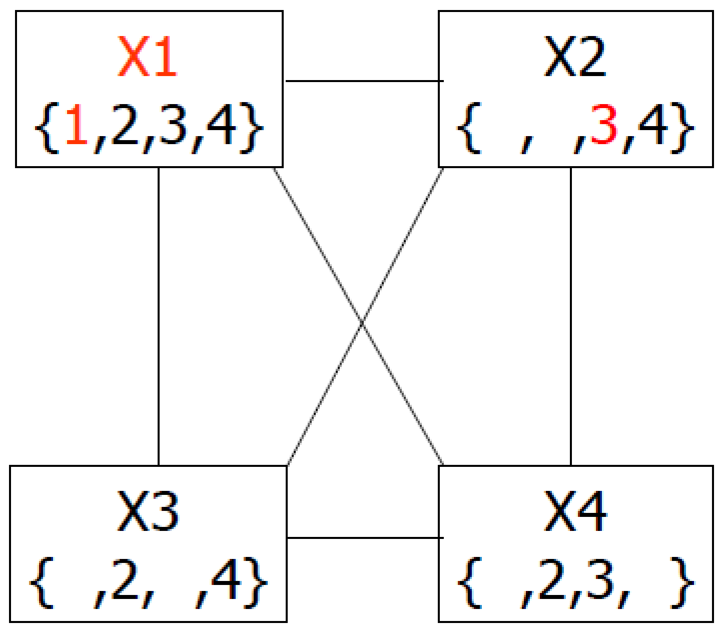
(From Bonnie Dorr, U of Md, CMSC 421)

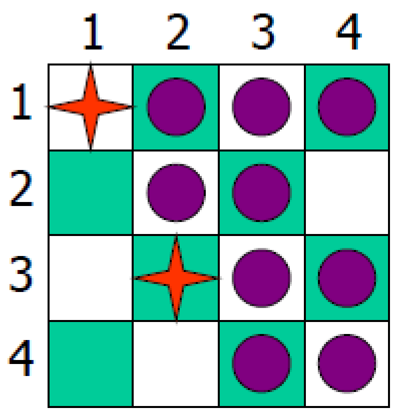

1
2
3
4
1
2
3
4

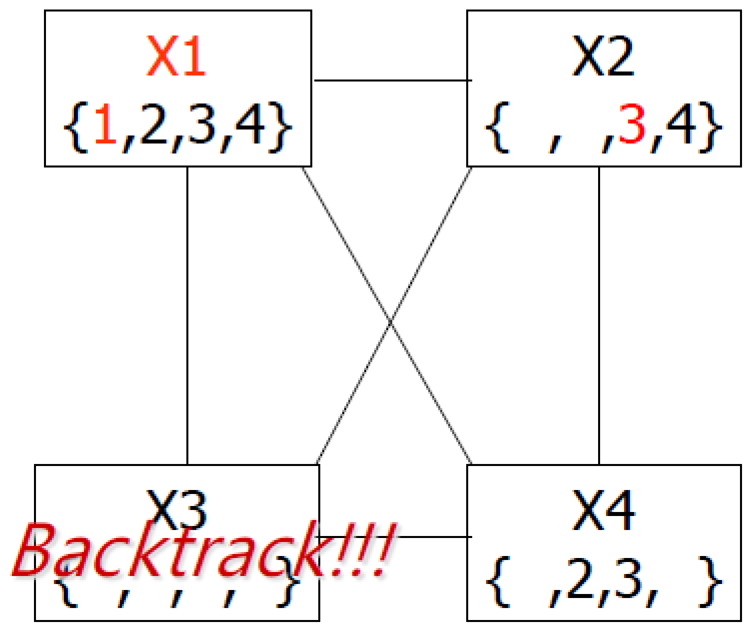

X1
{1,2,3,4}
X2
{1,2,3,4}
X3
{1,2,3,4}
X4
{1,2,3,4}


1 2 3 4
1
2
3
4
X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ ,2, ,4}
X4
{ ,2,3, }

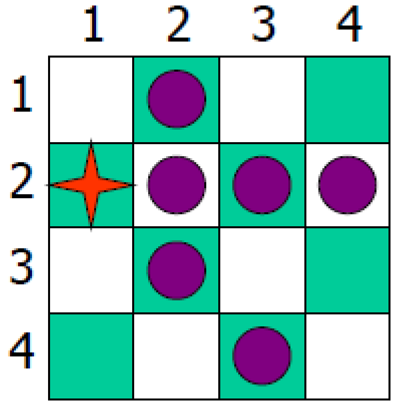

1 2 3 4
1
2
3
4

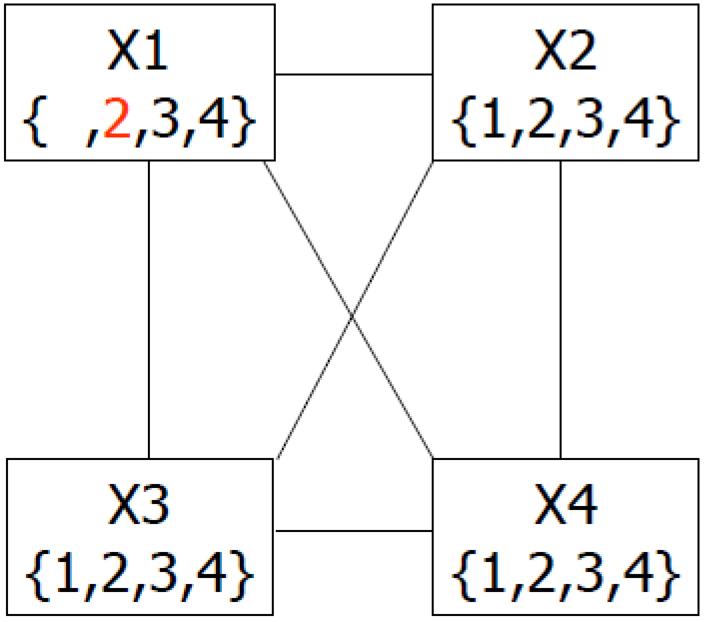

X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ ,2, ,4}
X4
{ ,2,3, }

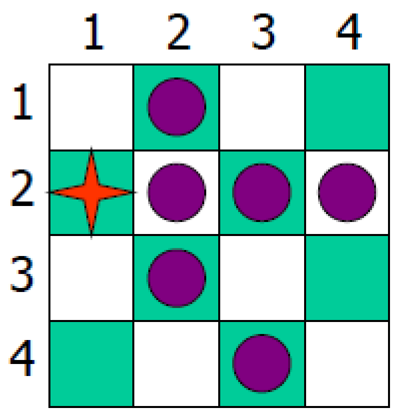

1
2
3
4
1
2
3
4

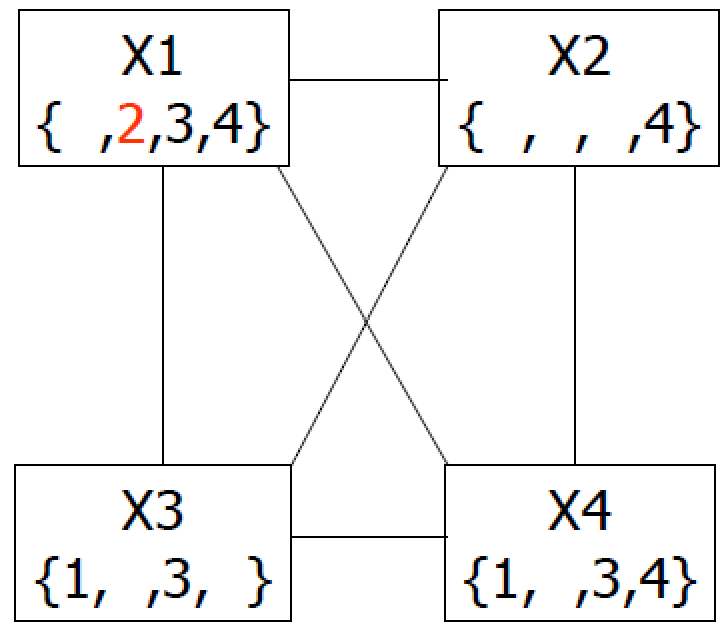

X1
{1,2,3,4}
X2
{ , ,3,4}
X3
{ , , , }
X4
{ ,2,3, }
Backtrack!!!

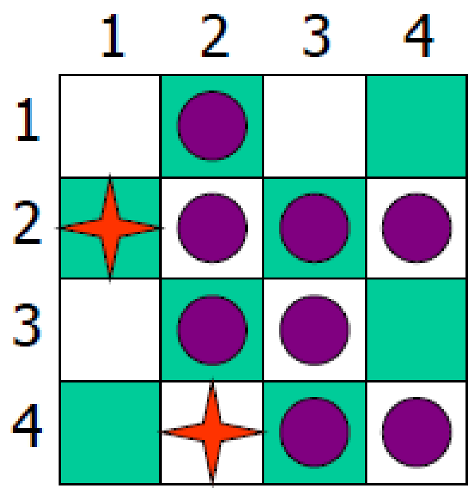
Picking up a little later after two steps of backtracking….

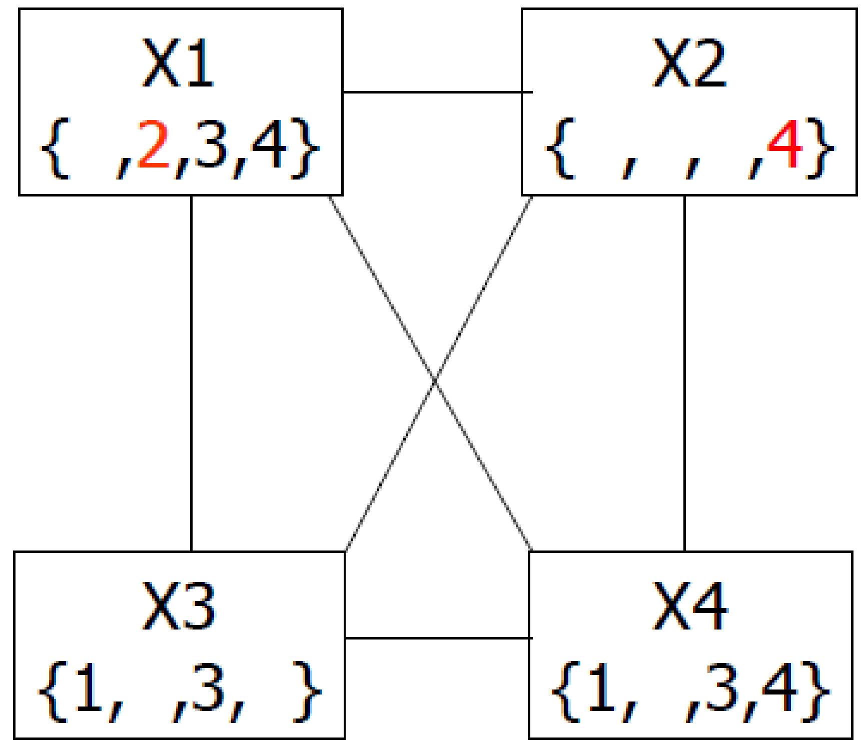


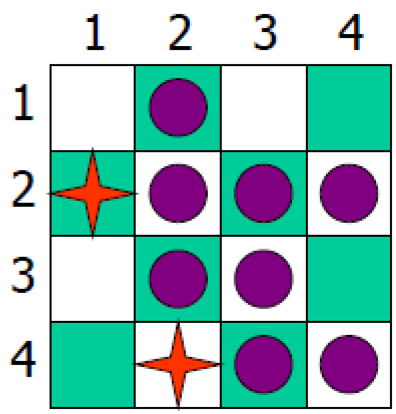


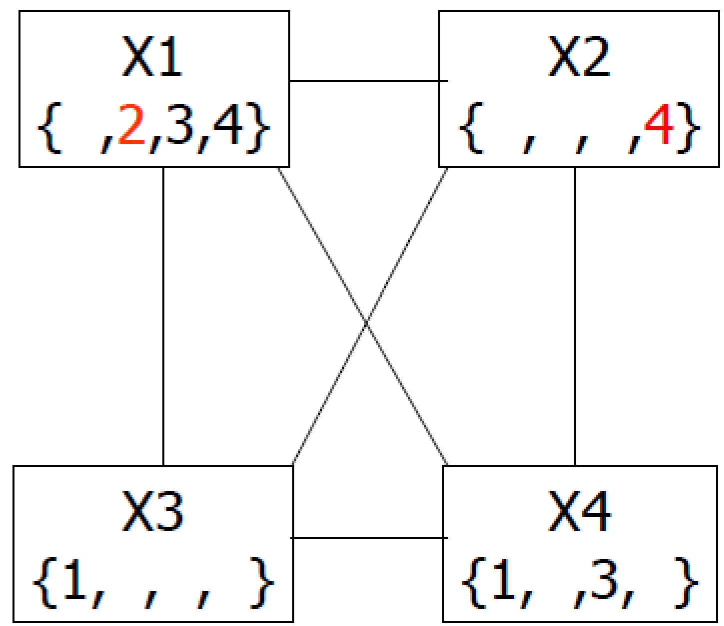

1 2 3 4
1
2
3
4

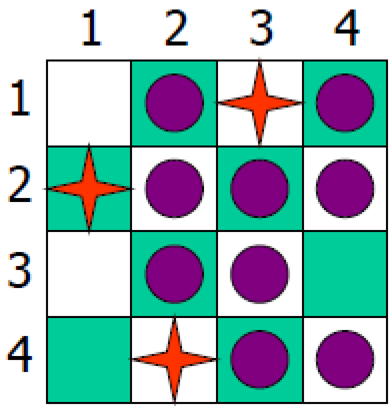

X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, ,3, }
X4
{1, ,3,4}

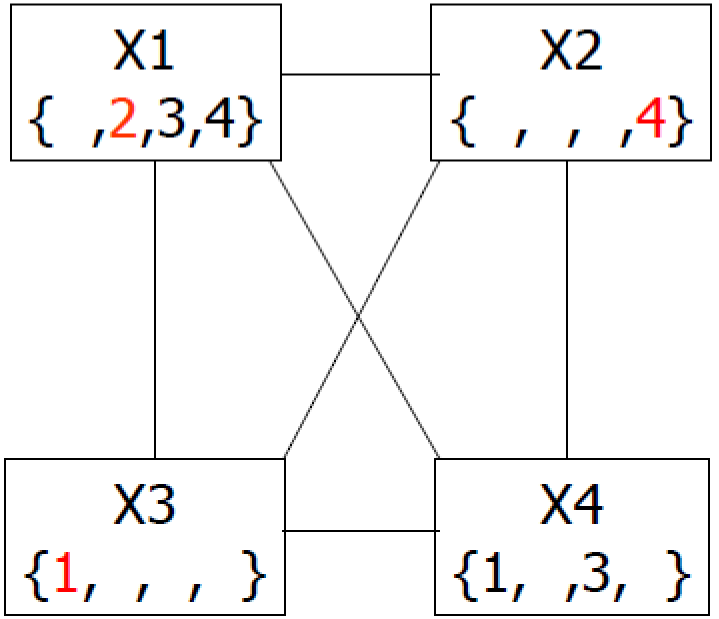

1 2 3 4
1
2
3
4

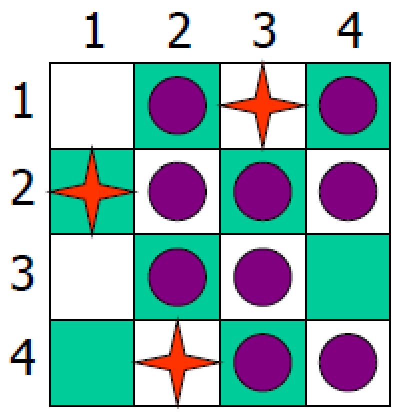

X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, ,3, }
X4
{1, ,3,4}

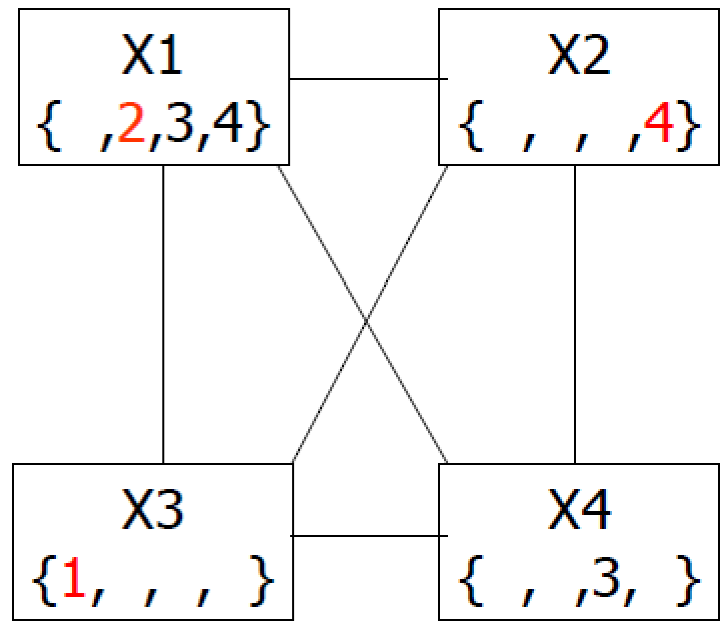

1 2 3 4
1
2
3
4

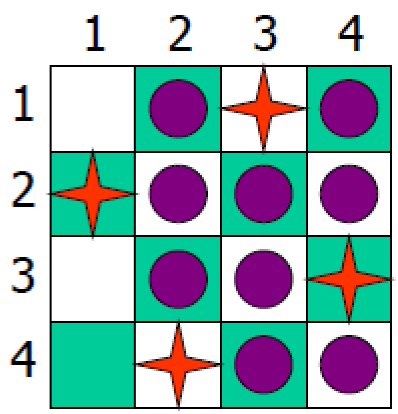

X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, , , }
X4
{1, ,3, }

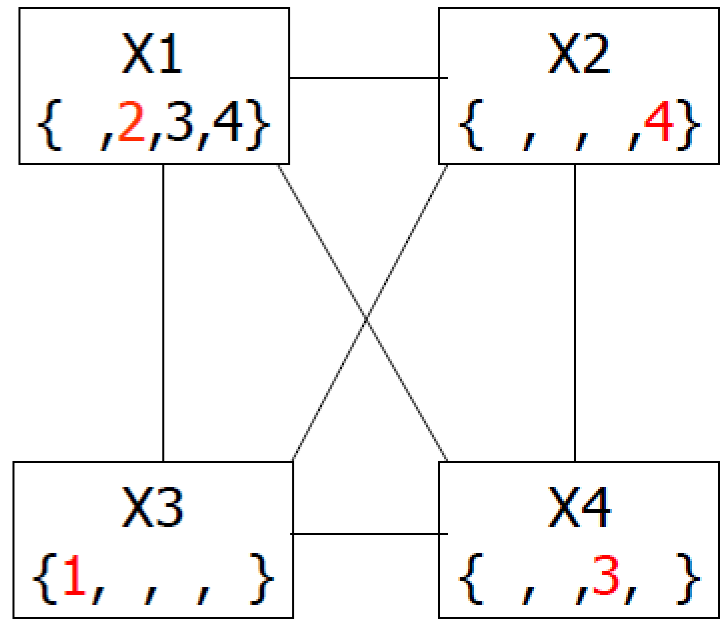

1 2 3 4
1
2
3
4


X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, , , }
X4
{1, ,3, }


1 2 3 4
1
2
3
4


X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, , , }
X4
{ , ,3, }


1 2 3 4
1
2
3
4


X1
{ ,2,3,4}
X2
{ , , ,4}
X3
{1, , , }
X4
{ , ,3, }

Exercise

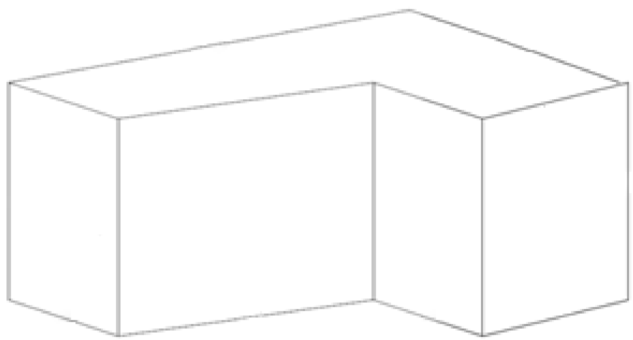
```
   CROSS
+ ROADS
----------------
  DANGER
```

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
- Local search can be done by iterative min-conflicts

CSP in computer vision: Line drawing interpretation

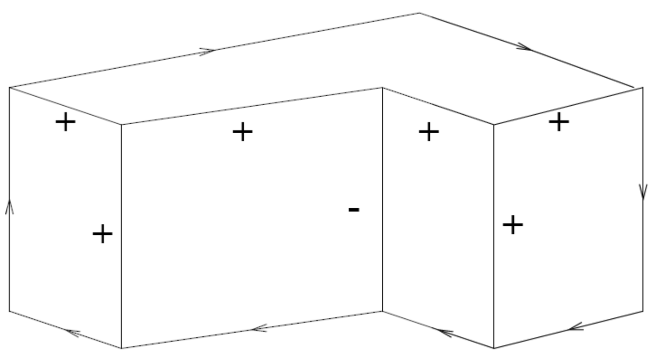
An example polyhedron:



Variables: edges

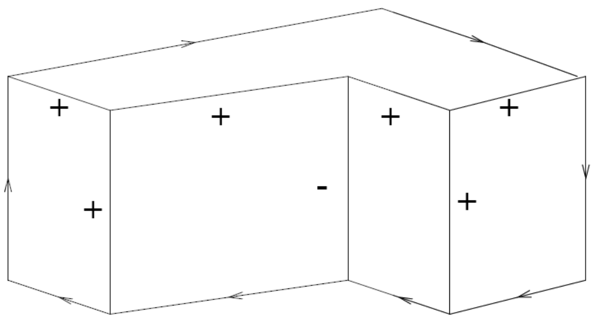
Domains: +, −, →, ←

Desired output:

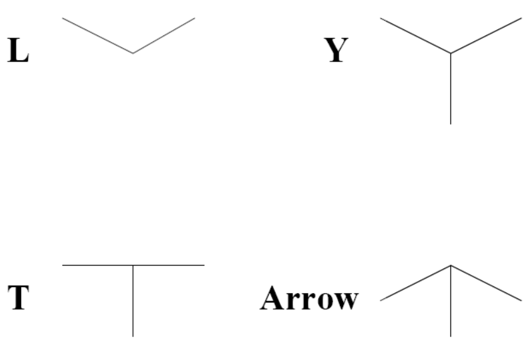


D. Waltz, 1975

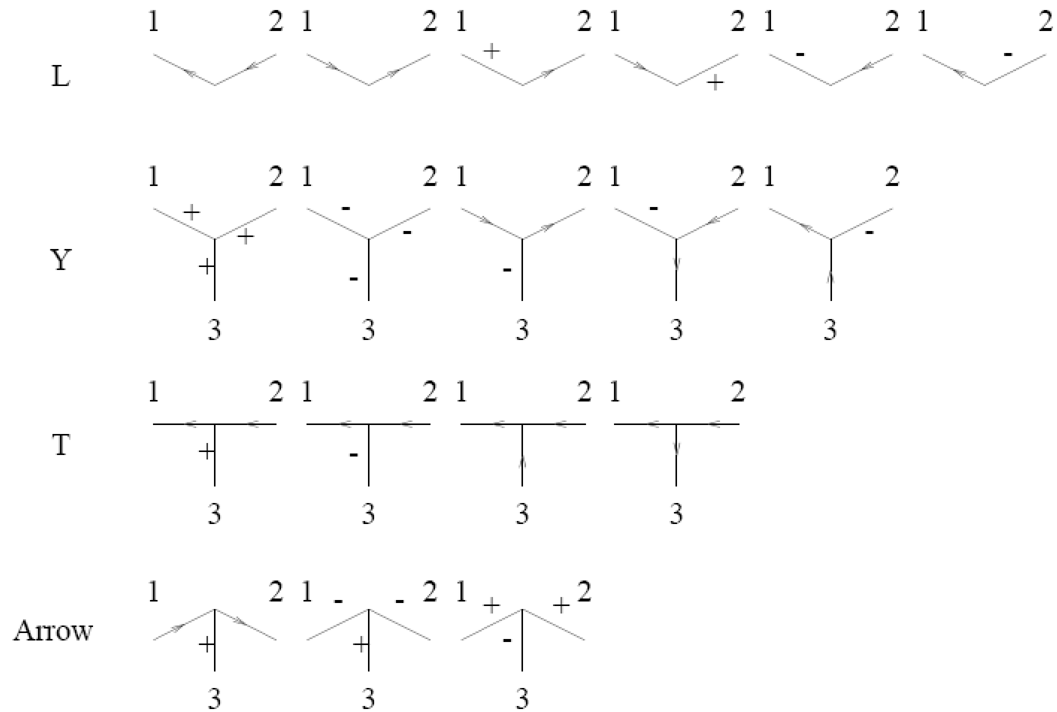
CSP in computer vision: Line drawing interpretation



Four vertex types:



Constraints imposed by each vertex type:



D. 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:

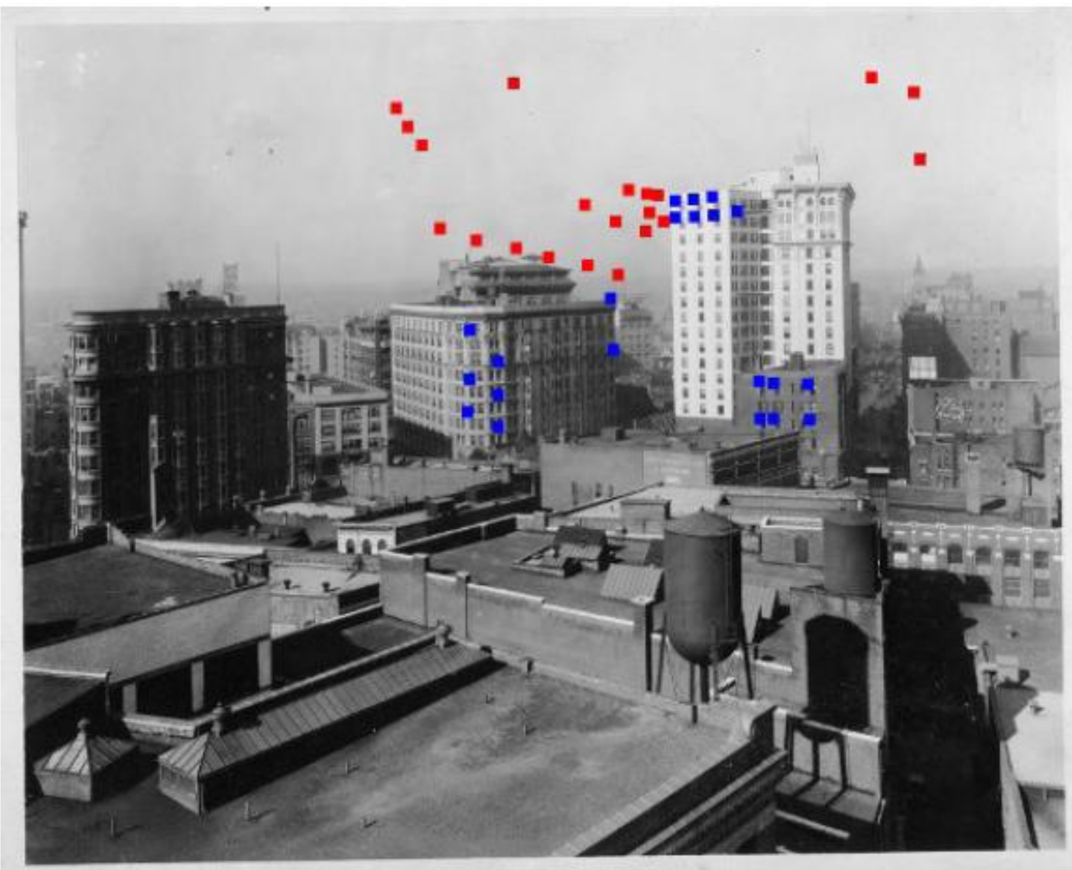
Set of Objects: Buildings

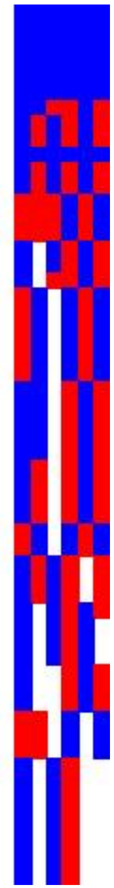
G. Schindler, F. Dellaert, and S.B. Kang, [Inferring Temporal Order of Images From 3D Structure](), IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR) , 2007.

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

CSP in computer vision: 4D Cities

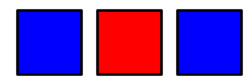
observed missing occluded



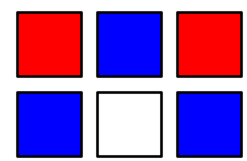


Columns: images
Rows: points

Violates constraints:



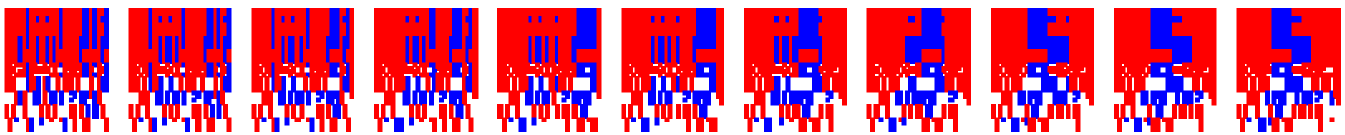
Satisfies constraints:



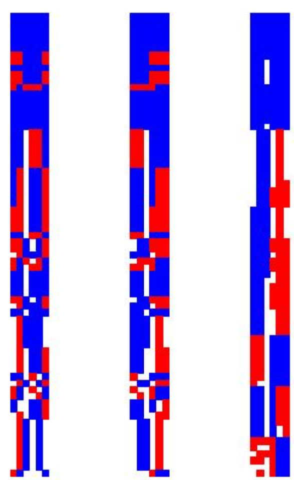
- Goal: reorder images (columns) to have as few violations as possible

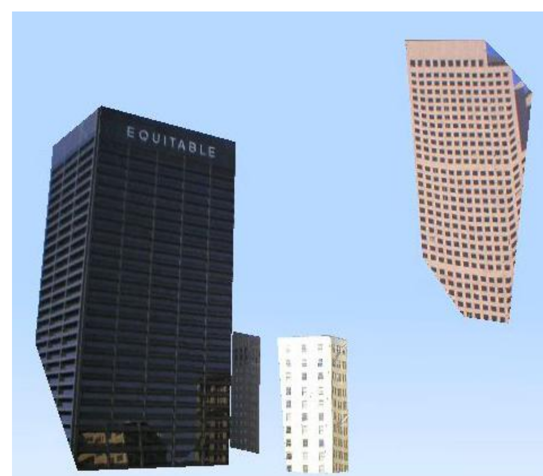
CSP in computer vision: 4D Cities

- **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





CSPs and NP-completeness

- The satisfiability (SAT) problem:
  - Given a Boolean formula, find out whether there exists an assignment of the variables that makes it evaluate to true, e.g.:

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

- SAT is *NP-complete* (Cook, 1971)
  - It's in NP and every other problem in NP can be reduced to it
  - So are graph coloring, n-puzzle, generalized sudoku, and the traveling salesman problem