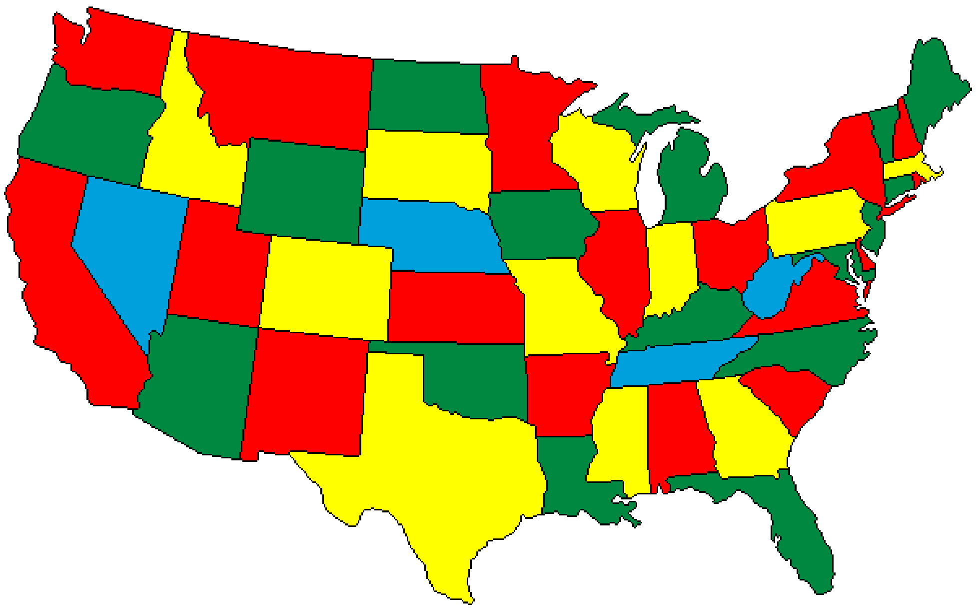
Chapter 5
Constraint Satisfaction Problems

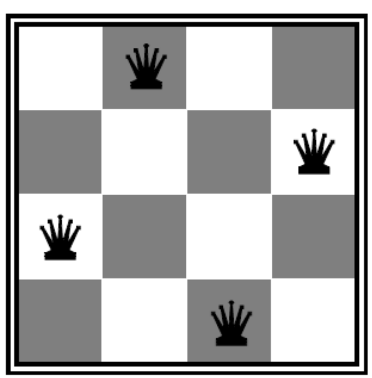
BBM 405 – Fundamentals of Artificial Intelligence

Pinar Duygulu

Slides are mostly adapted from AIMA and MIT Open Courseware

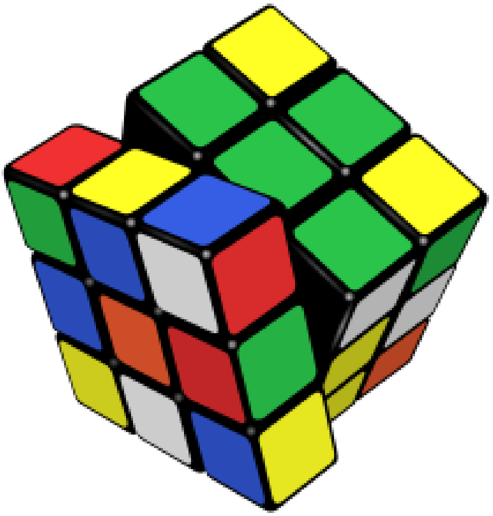


8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5



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



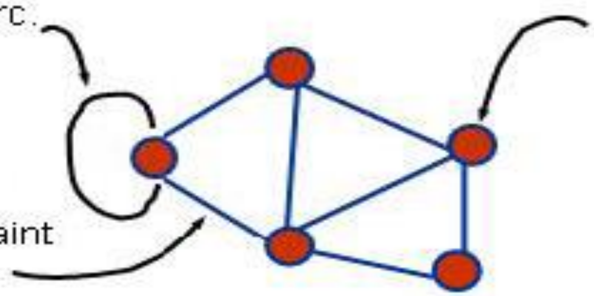
8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

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

Constraint Satisfaction Problems

General class of Problems: **Binary CSP**



[Unary constraints just cut down domains]

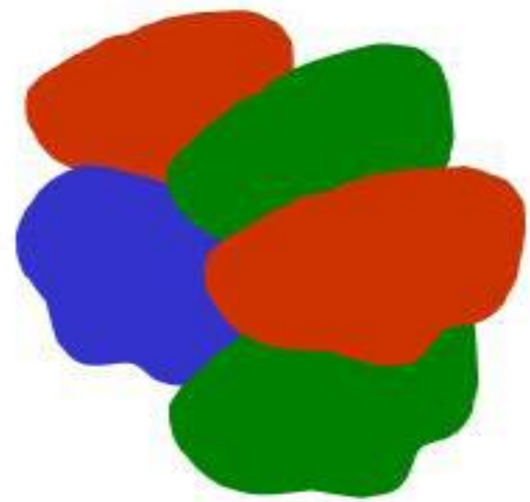
This diagram is called a constraint graph

Basic problem:

Find a $d_j \in D_i$ for each V_i s.t. all constraints satisfied (finding consistent labeling for variables)

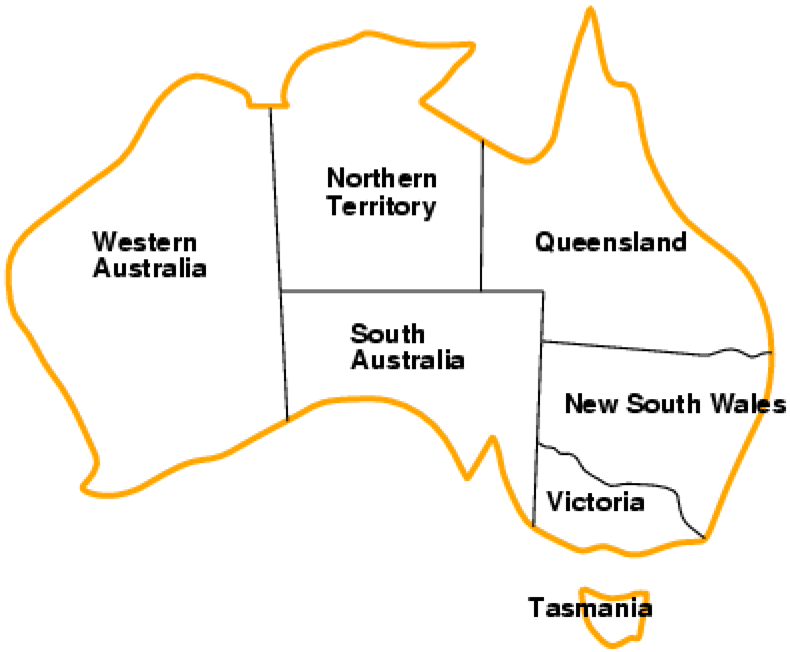
Graph Coloring as CSP

Pick colors for map regions, avoiding coloring adjacent regions with the same color



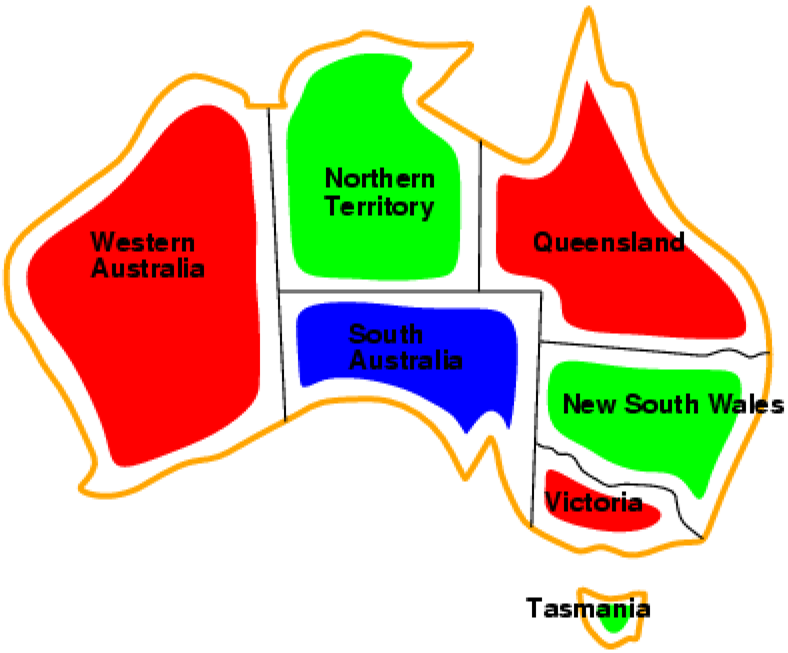
Variables	regions
Domains	colors allowed
Constraints	adjacent regions must have different colors

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)}

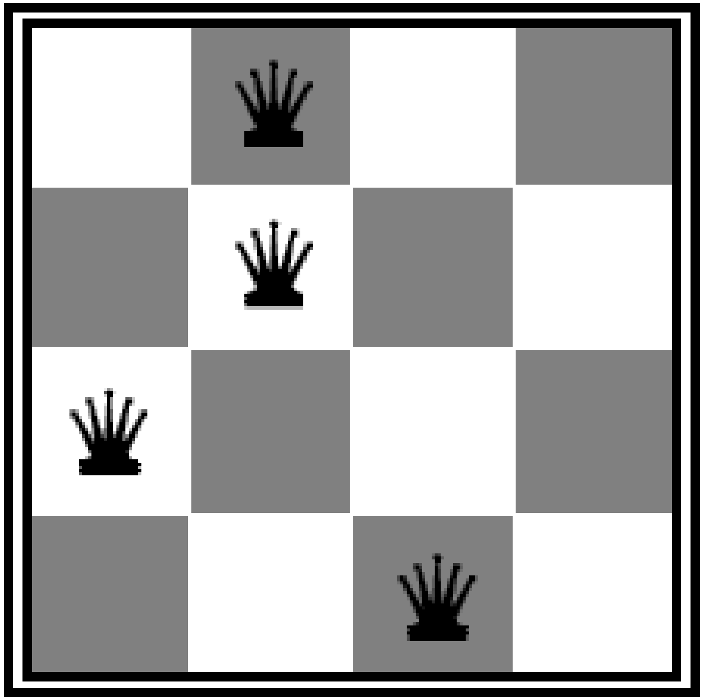
Example: Map Coloring

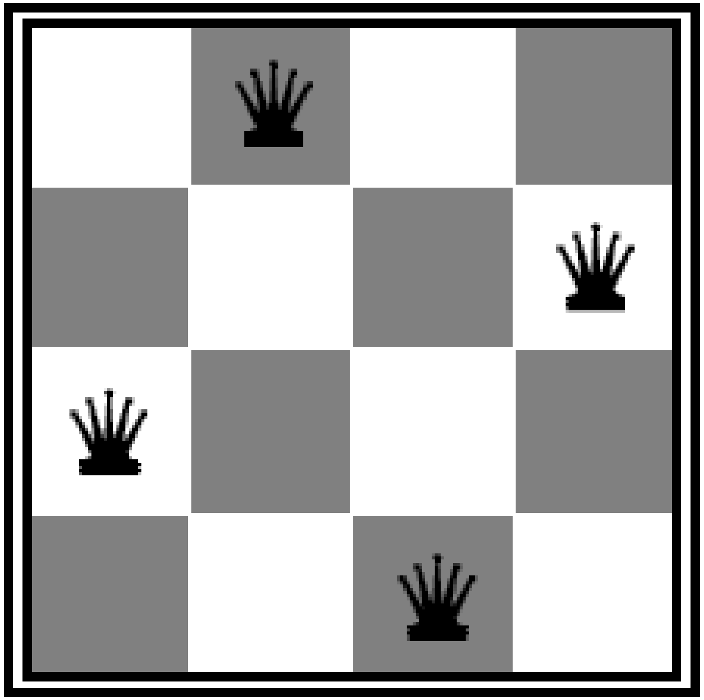


- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Example: *n*-queens problem

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

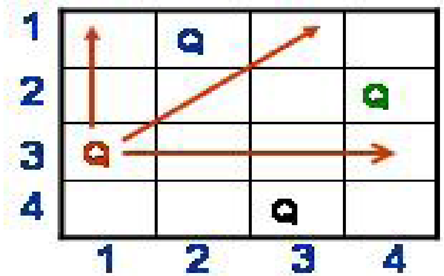




N-Queens as CSP

Classic "benchmark" problem

Place N queens on an NxN chessboard so that none can attack the other.



Variables are board positions in NxN chessboard

Domains Queen or blank

Constraints Two positions on a line (vertical, horizontal, diagonal) cannot both be Q

Example: N-Queens

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

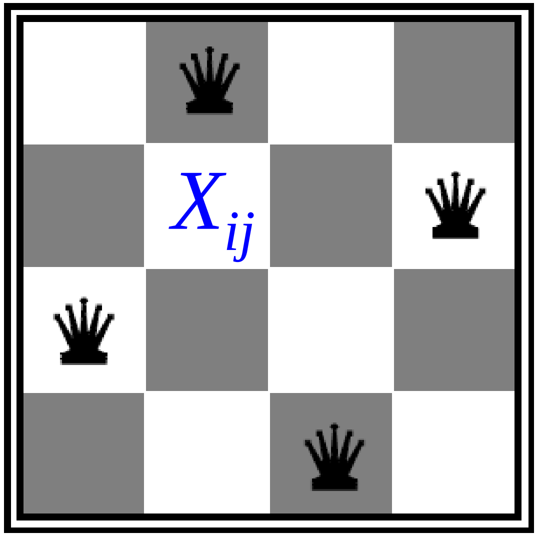
$$\Sigma_{i,j} X_{ij} = N$$

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

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

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

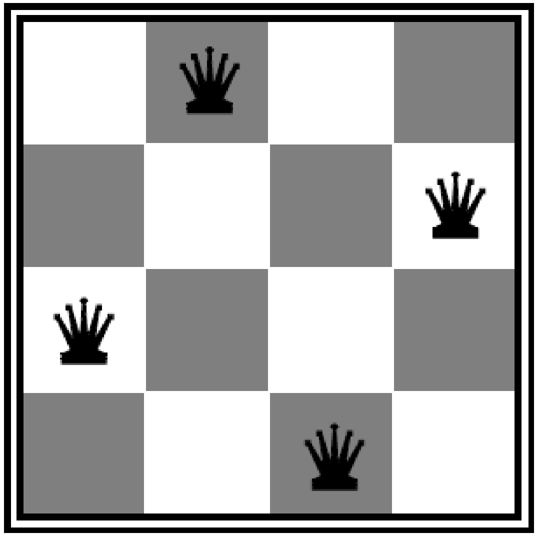
$$(X_{ij}, X_{i+k,\, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$



N-Queens: Alternative formulation

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

  $O + O = R + 10 * X_1$

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

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

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

  $T \neq 0$, $F \neq 0$

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

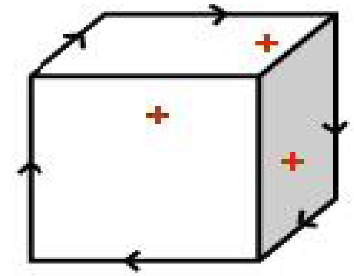
Example: Sudoku

- **Variables:** X_{ij}
- **Domains:** $\{1, 2, \ldots, 9\}$
- **Constraints:**

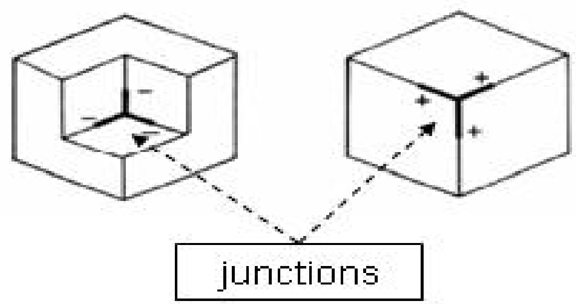
  Alldiff(X_{ij} in the same *unit*)

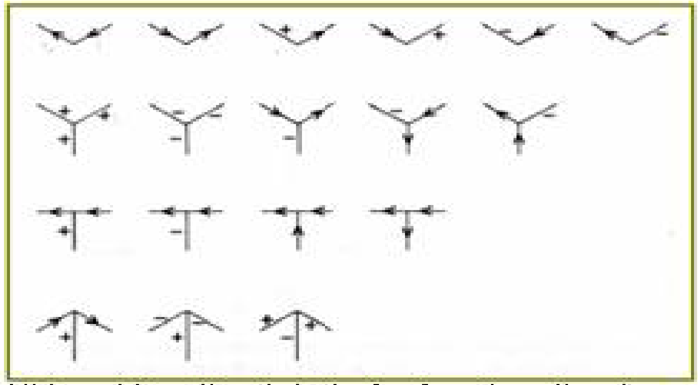
					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8		X_{ij}		4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

Line labelings as CSP



Label lines in drawing as
convex (+), concave (-), or
boundary (>).





All legal junction labels for four junction types

Variables are line junctions

Domains are set of legal labels for that junction type

Constraints shared lines between adjacent junctions must have same label.

3-SAT as CSP

The original NP-complete problem

(A or B or !C) and (!A or C or B) ...

Find values for boolean variables A,B,C,... that satisfy the formula.

Variables	clauses
Domains	boolean variable assignments that make clause true
Constraints	clauses with shared boolean variables must agree on value of variable

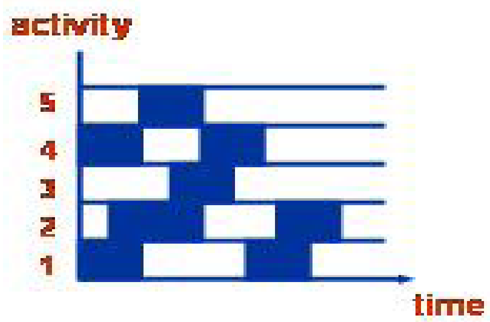
Real-world CSPs

- 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/

Scheduling as CSP

Choose time for activities e.g. observations on Hubble telescope, or terms to take required classes.



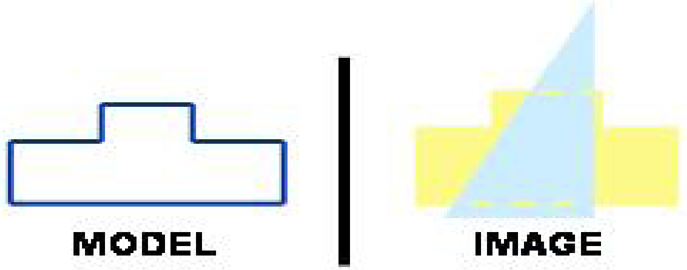
Variables are activities

Domains sets of start times (or "chunks" of time)

Constraints
1. Activities that use same resource cannot overlap in time
2. Preconditions satisfied

Model-based recognition as CSP

Find given model in edge image, with rotation and translation allowed.



Variables	edges in model
Domains	set of edges in image
Constraints	angle between model & image edges must match

Good News / Bad News

Good News - very general & interesting class problems

Bad News - includes NP-Hard (intractable) problems

So, good behavior is a function of domain not the formulation as CSP.

CSP Example

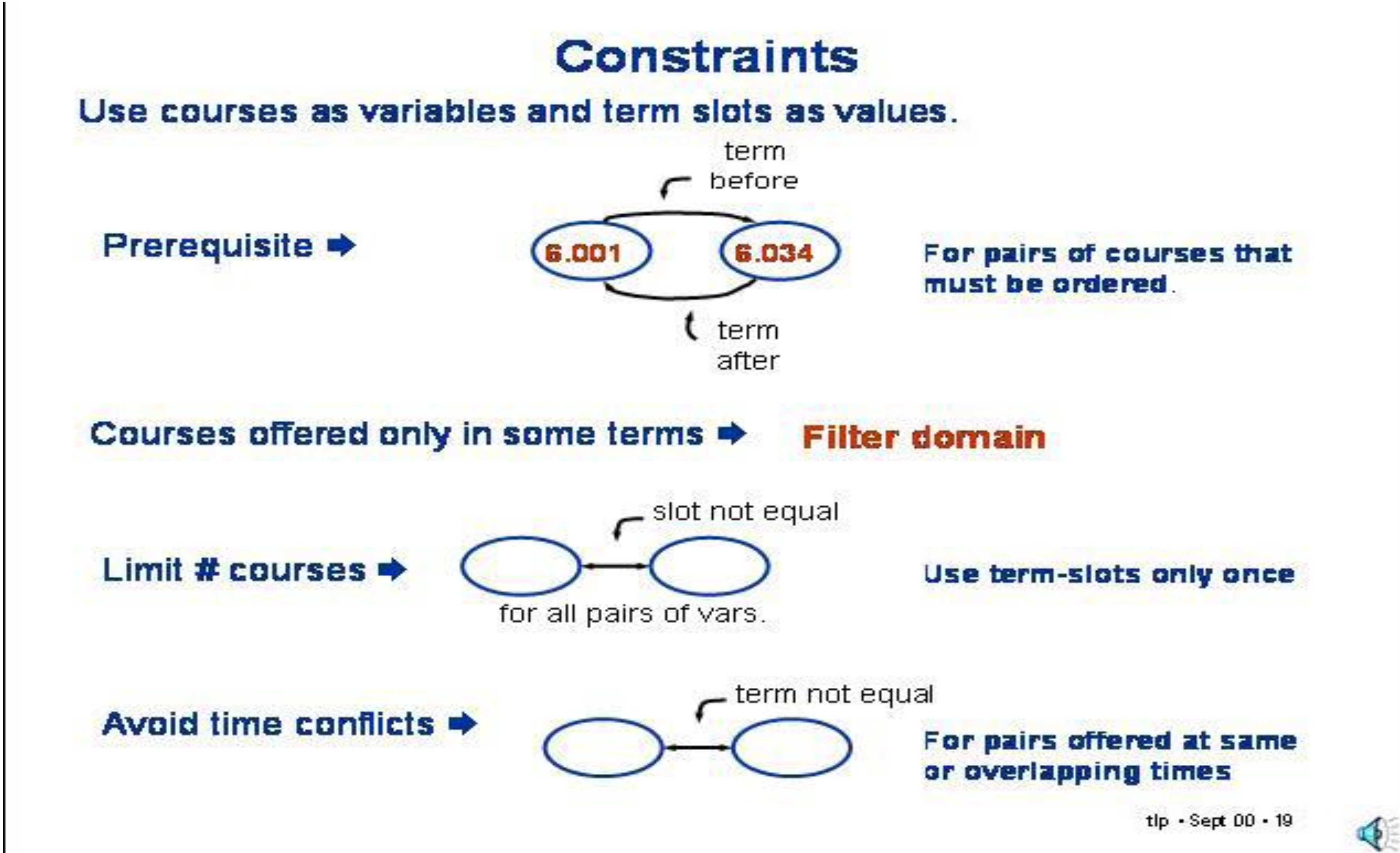
Given 40 courses (8.01, 8.02, ... 6.840) & 10 terms (Fall 1, Spring 1, ... , Spring 5). Find a legal schedule.

Constraints
- Pre-requisites
- Courses offered on limited terms
- Limited number of courses per term
- Avoid time conflicts

Note, CSPs are not for expressing (soft) preferences e.g., minimize difficulty, balance subject areas, etc.

Choice of variables & values

VARIABLES	DOMAINS
A. Terms?	Legal combinations of for example 4 courses (but this is huge set of values).
B. Term Slots? subdivide terms into slots e.g. 4 of them (Fall 1,1) (Fall 1,2) (Fall1,3) (Fall 1,4)	Courses offered during that term
C. Courses?	Terms or term slots (Term slots allow expressing constraint on limited number of of courses / term.)


Constraints
Use courses as variables and term slots as values.
Prerequisite ➡
term before
6.001
6.034
term after
For pairs of courses that must be ordered.
Courses offered only in some terms ➡
Filter domain
Limit # courses ➡
slot not equal
for all pairs of vars.
Use term-slots only once
Avoid time conflicts ➡
term not equal
For pairs offered at same or overlapping times
tlp • Sept 00 • 19

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)?
  n (this is good)

- 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 bad)

- How can we reduce the branching factor?

Solving CSPs

Solving CSPs involves some combination of:

1. Constraint propagation, to eliminate values that could not be part of any solution
2. Search, to explore valid assignments

Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

$$V_i \rightarrow V_j$$

Directed arc (V_i, V_j) is arc consistent if
$\forall x \in D_i \; \exists y \in D_j$ such that (x,y) is allowed by the constraint on the arc

We can achieve consistency on arc by deleting values form D_i (domain of variable at tail of constraint arc) that fail this condition.

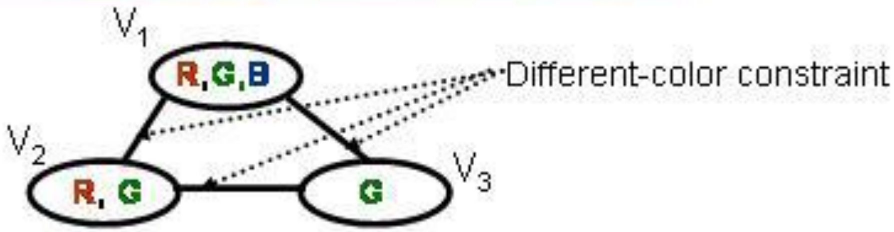
Assume domains are size at most <u>d</u> and there are <u>e</u> binary constraints.

A simple algorithm for arc consistency is $O(ed^3)$ – note that just verifying arc consistency takes $O(d^2)$ for each arc.

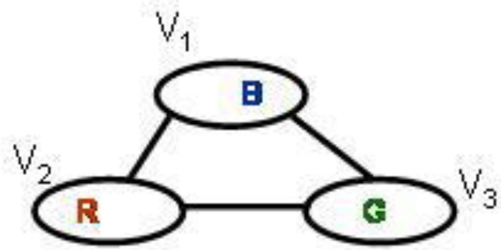
Constraint Propagation Example

Graph Coloring

Initial Domains are indicated

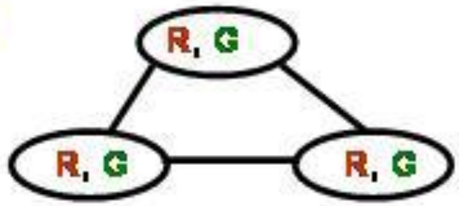


Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	V_1(G)
$V_2 - V_3$	V_2(G)
$V_1 - V_2$	V_1(R)
$V_1 - V_3$	none
$V_2 - V_3$	none

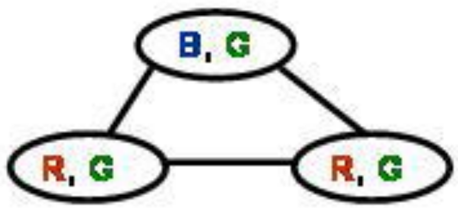


But, arc consistency is not enough in general

Graph Coloring

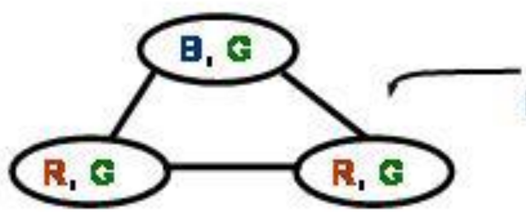


arc consistent but <u>no</u> solutions



arc consistent but <u>2</u> solutions B,R,G ; B,G,R .

arc consistent but <u>1</u> solution

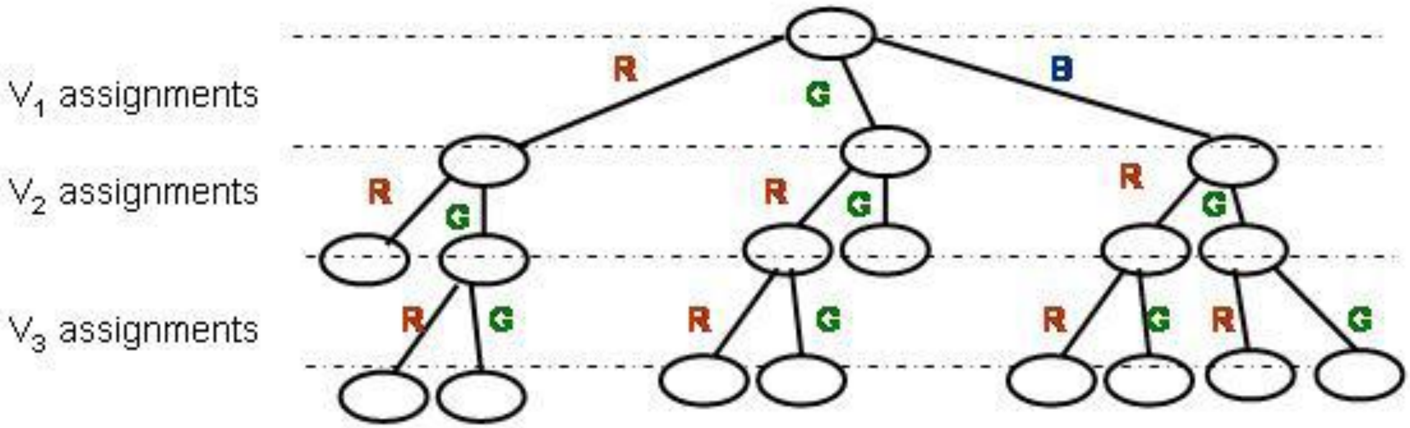


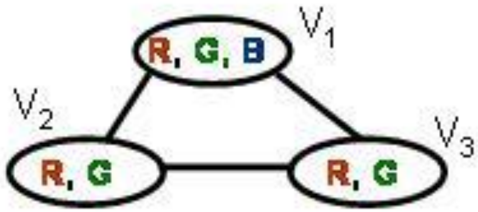
B, R not allowed

Need to do search to find solutions (if any)

Searching for solutions – backtracking (BT)

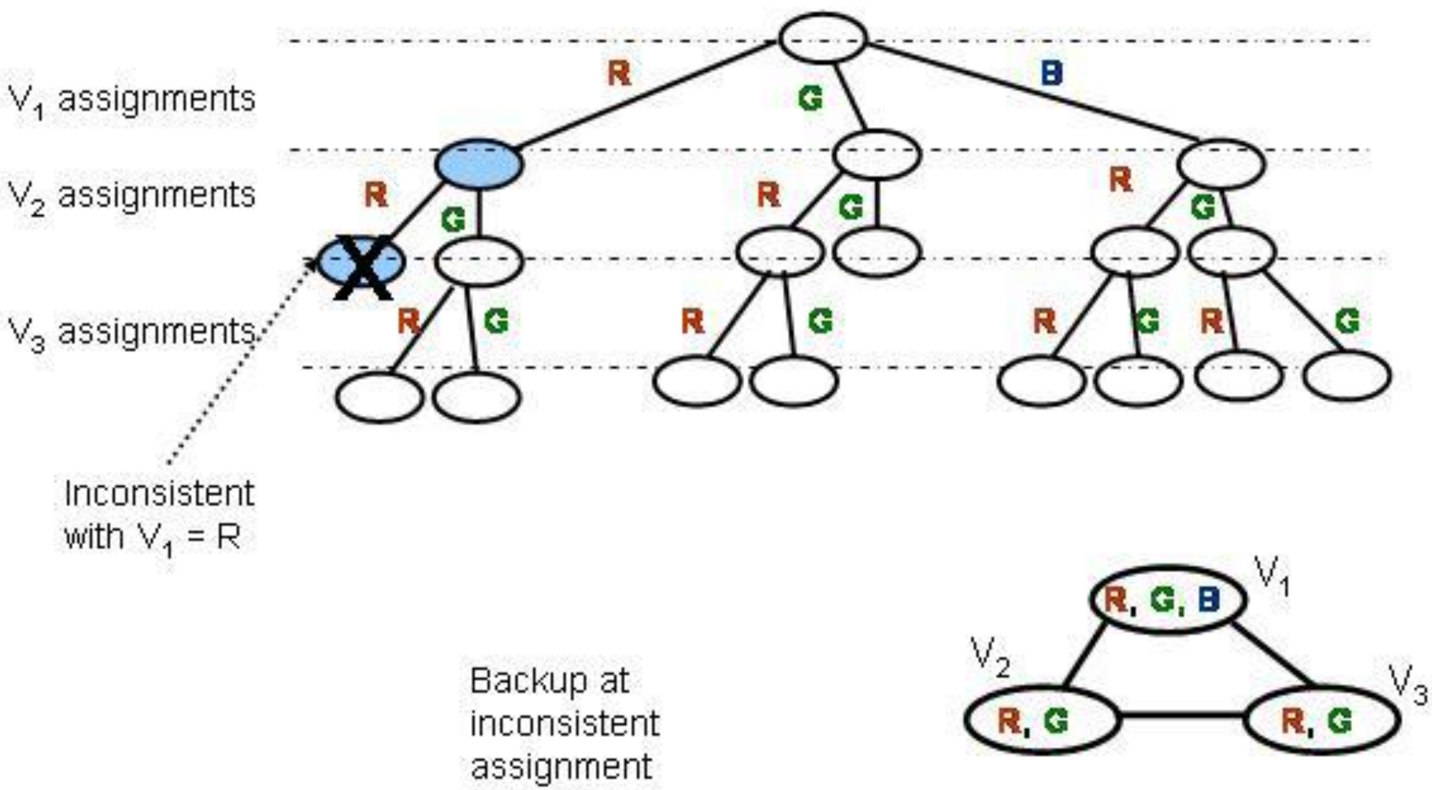
When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).





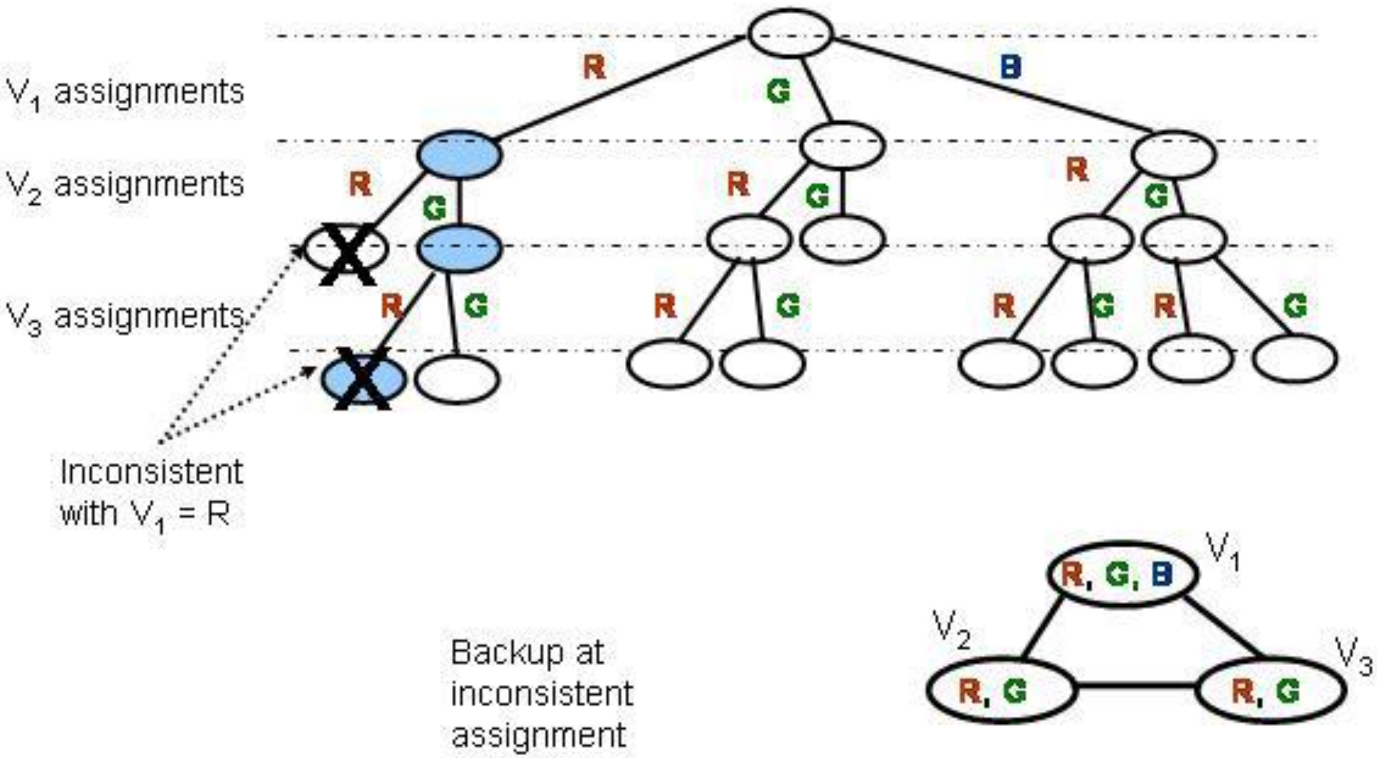
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



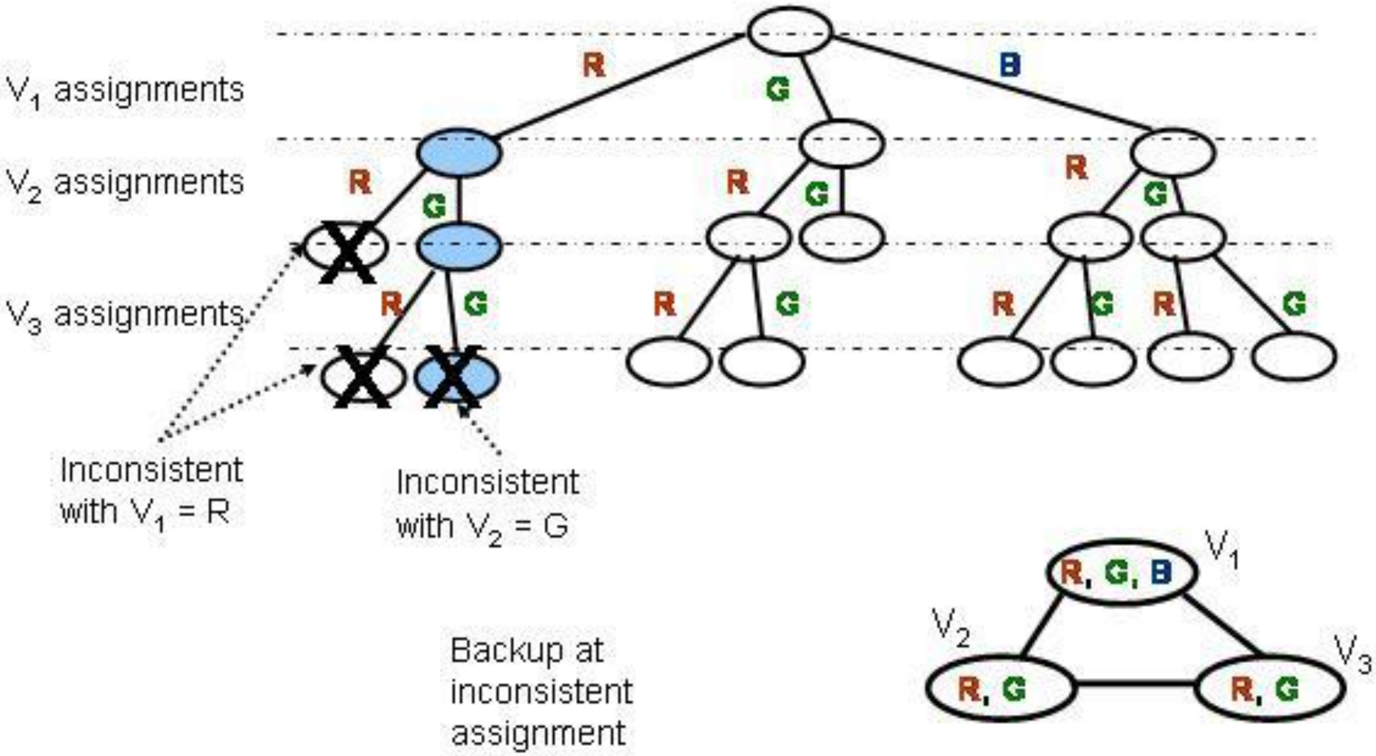
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



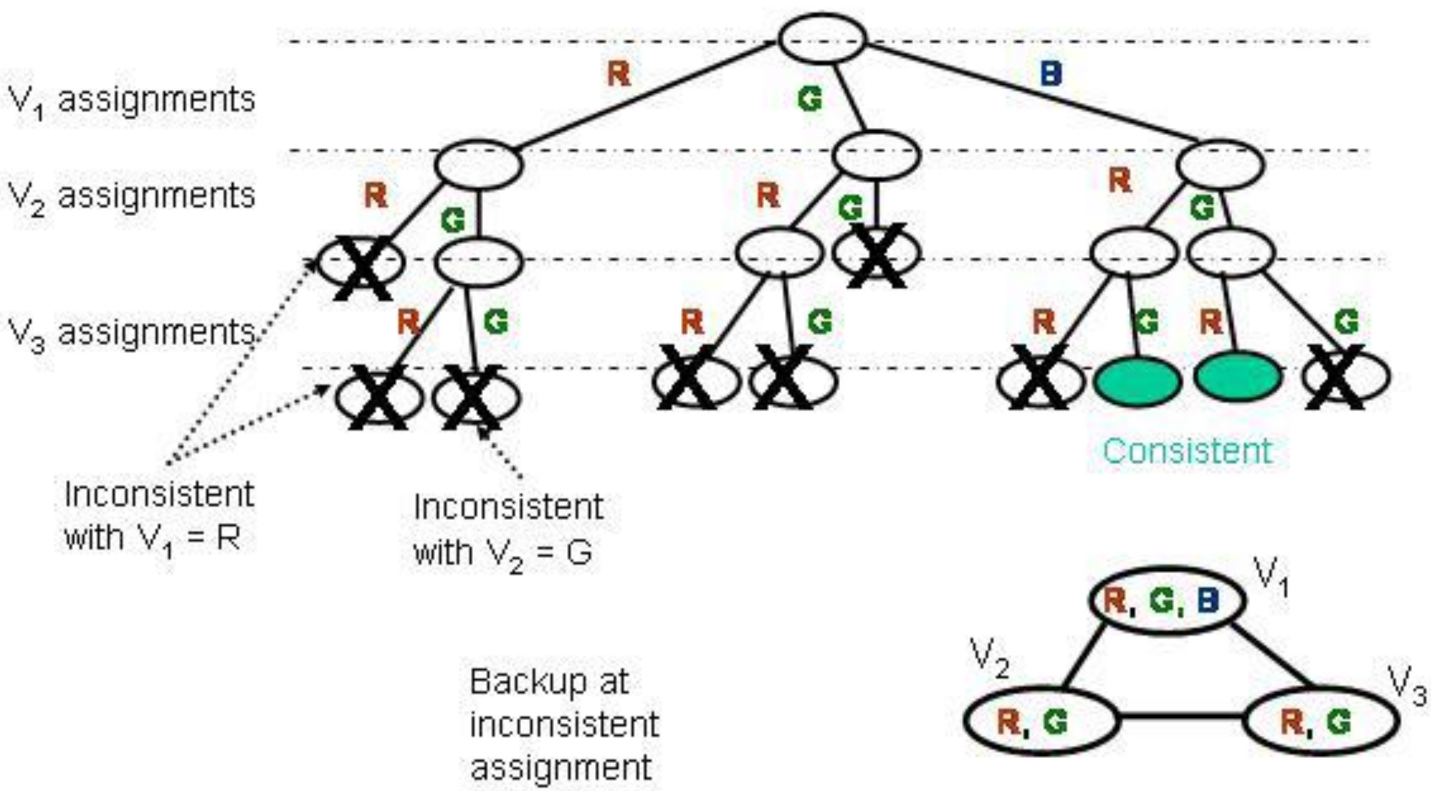
Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



Combine Backtracking & Constraint Propagation

A node in BT tree is partial assignment in which the domain of each variable has been set (tentatively) to singleton set.

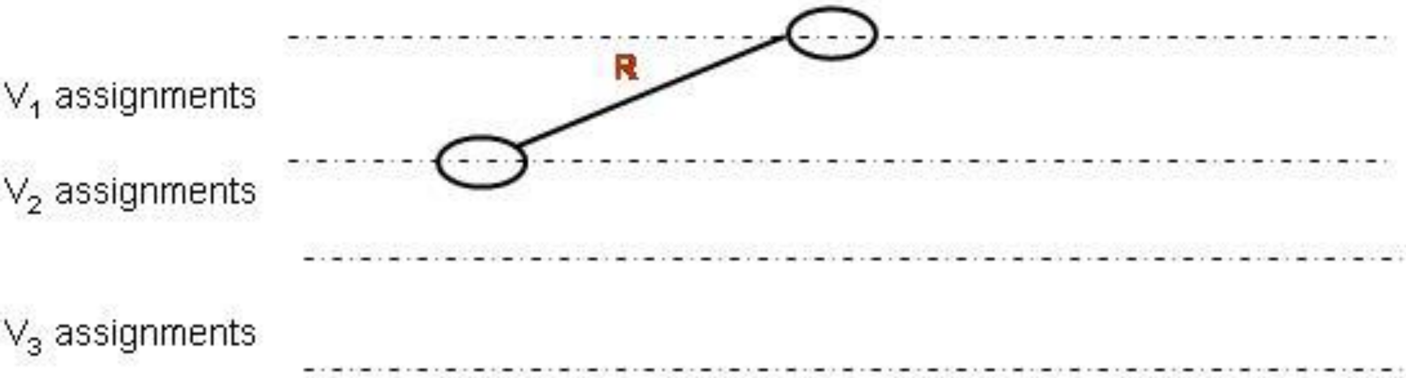
Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

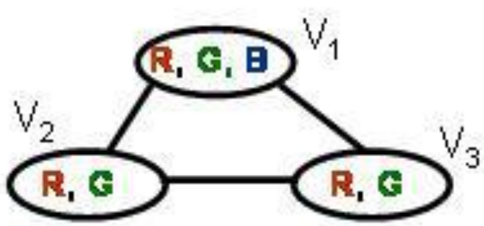
Question: How much propagation to do?

Answer: Not much, just local propagation from domains with unique assignments, which is called forward checking (FC). This conclusion is not necessarily obvious, but it generally holds in practice.

Backtracking with Forward Checking (BT-FC)

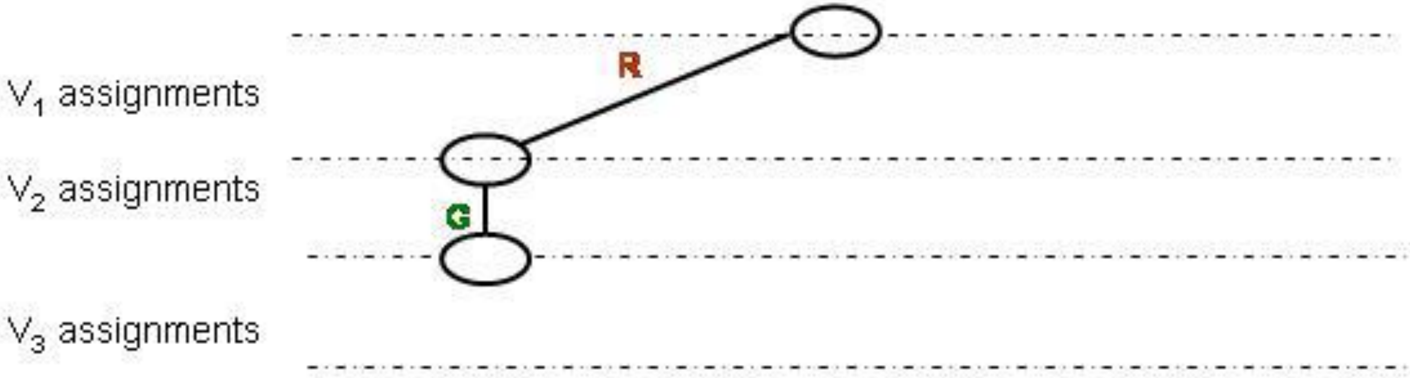
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

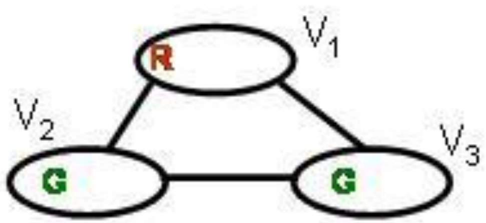




Backtracking with Forward Checking (BT-FC)

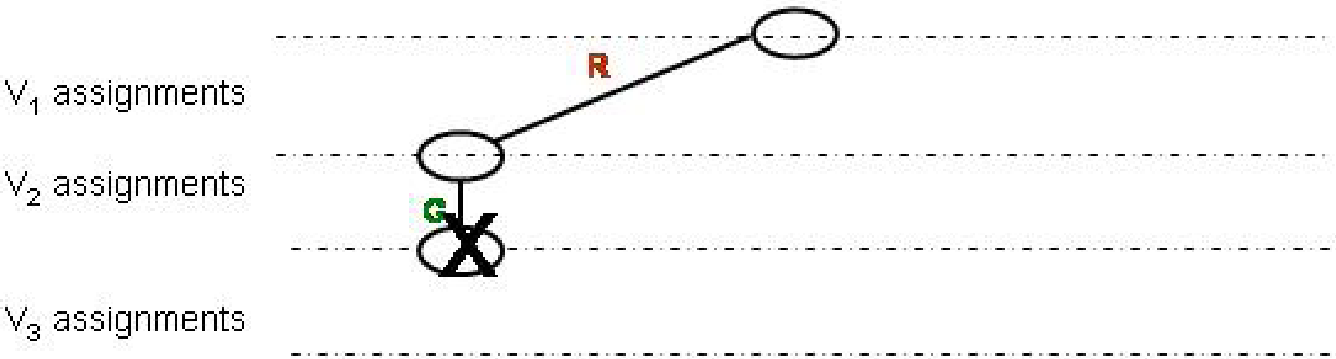
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



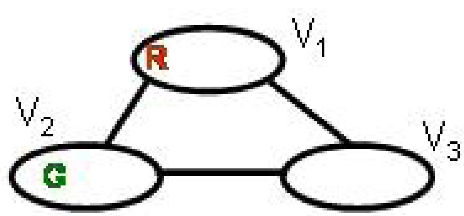


Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

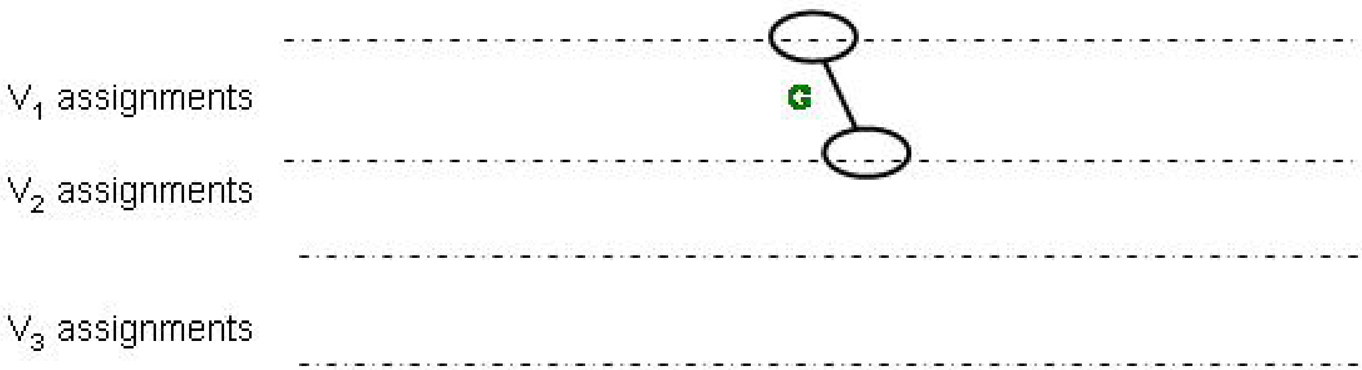


We have a conflict whenever a domain becomes empty.

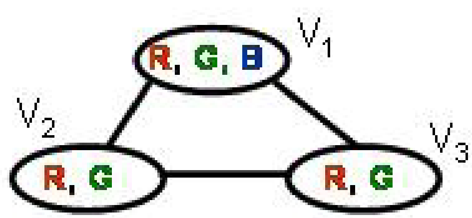


Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

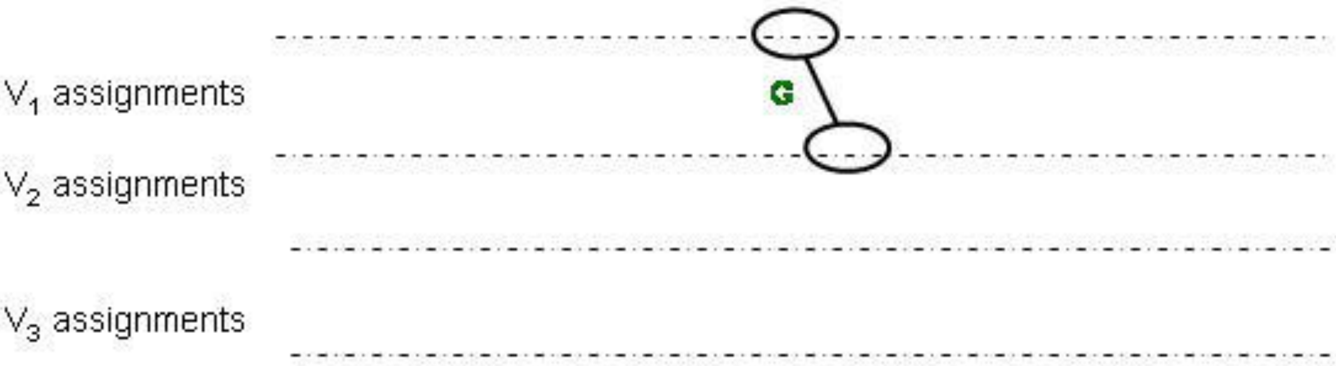


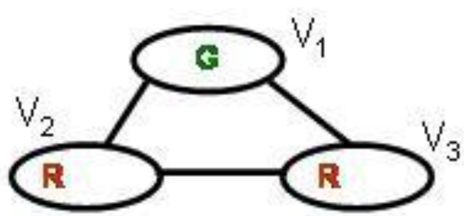
When backing up, need to restore domain values, since deletions were done to reach consistency with tentative assignments considered during search.



Backtracking with Forward Checking (BT-FC)

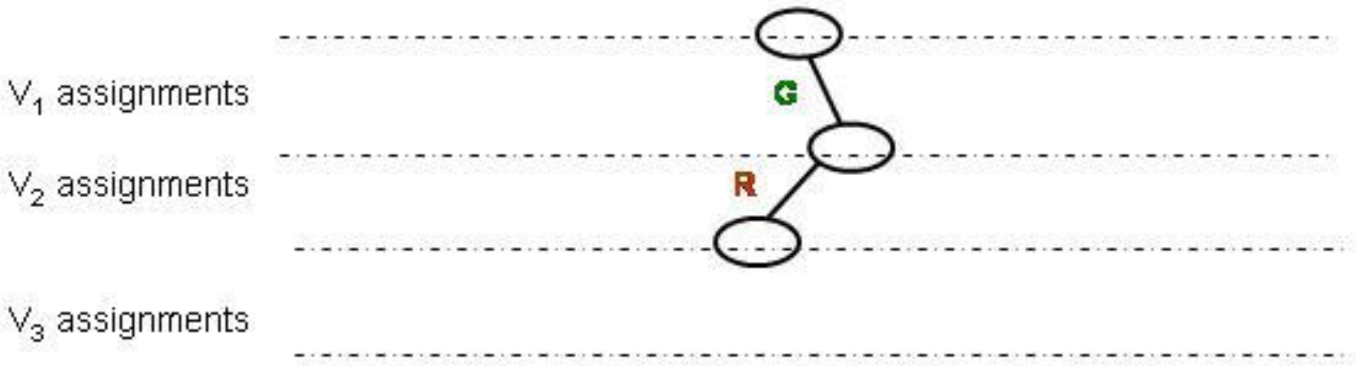
When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

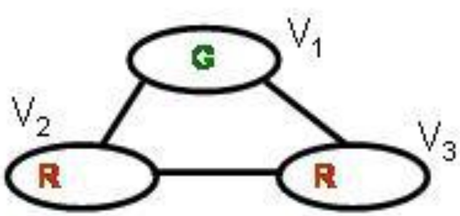




Backtracking with Forward Checking (BT-FC)

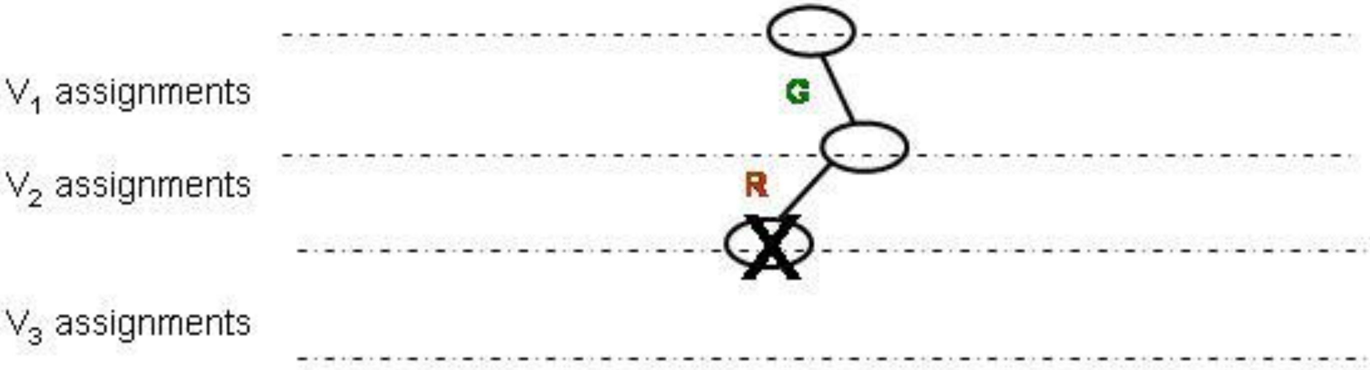
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

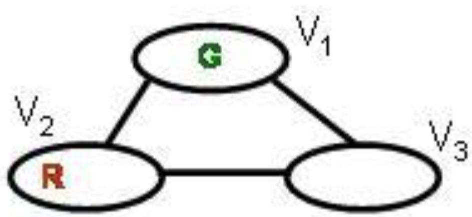




Backtracking with Forward Checking (BT-FC)

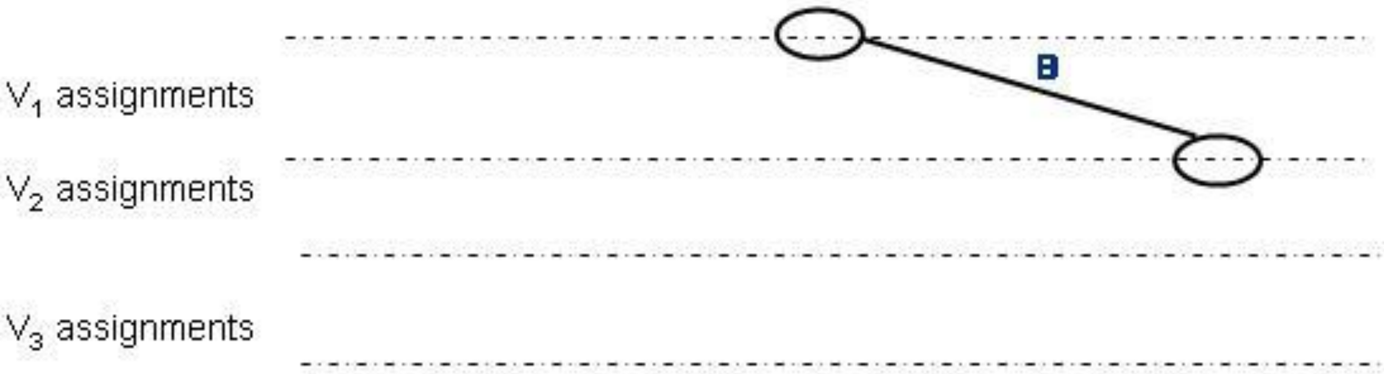
When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

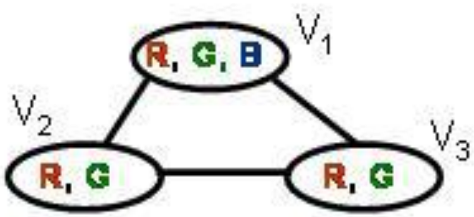




Backtracking with Forward Checking (BT-FC)

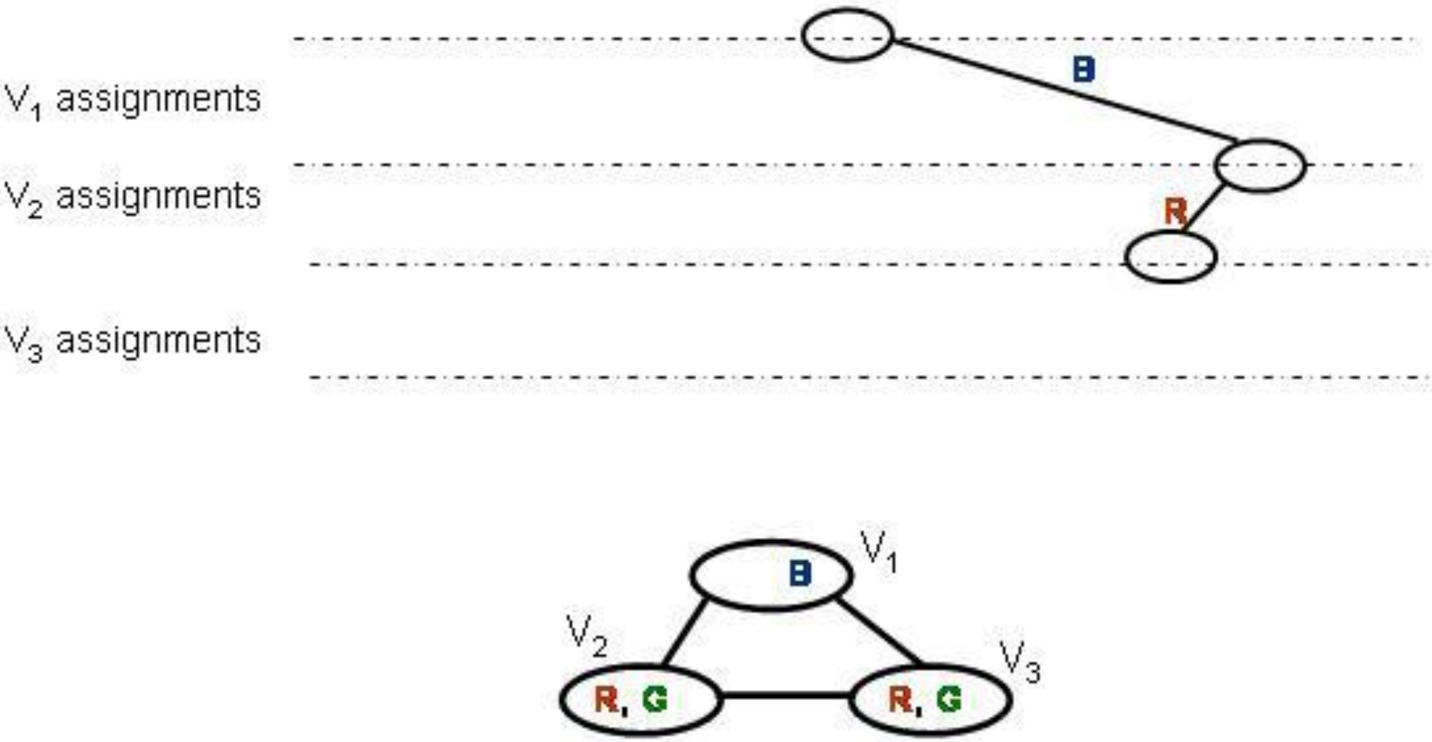
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.





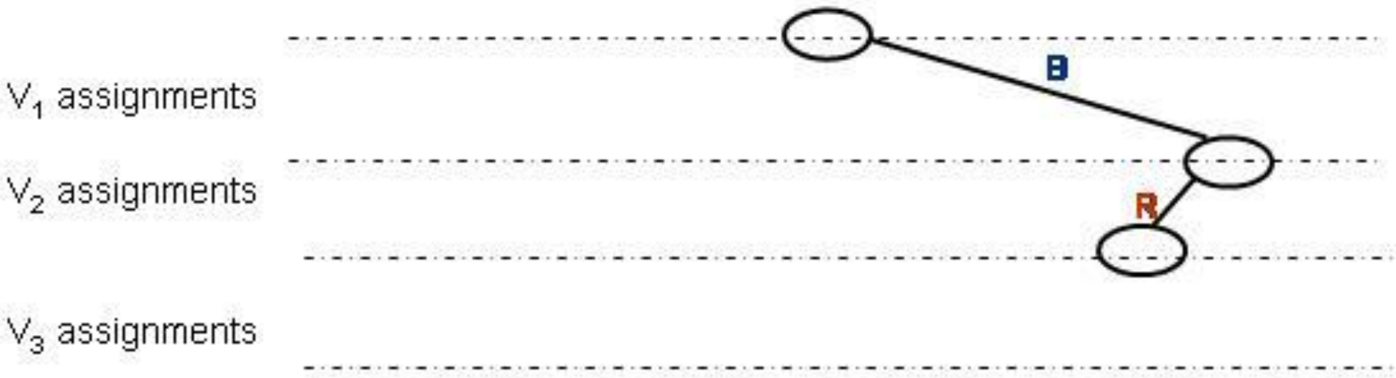
Backtracking with Forward Checking (BT-FC)

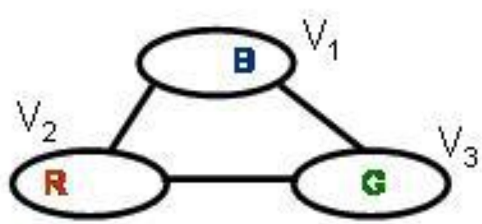
When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



Backtracking with Forward Checking (BT-FC)

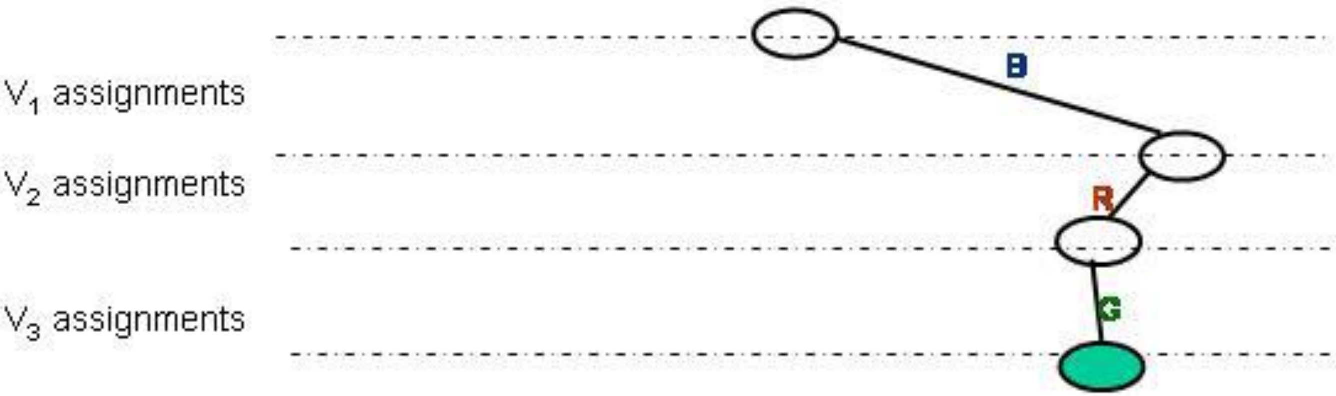
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

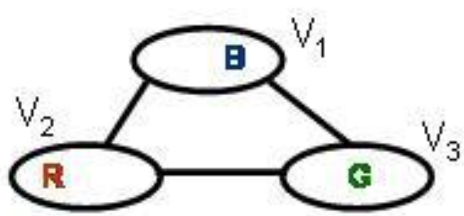




Backtracking with Forward Checking (BT-FC)

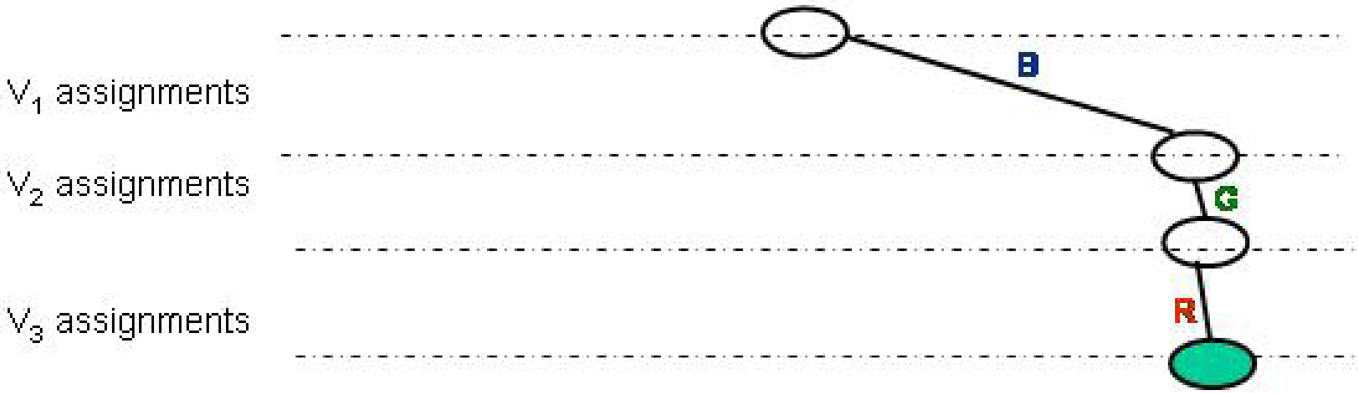
When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

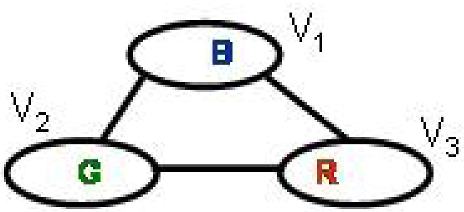




Backtracking with Forward Checking (BT-FC)

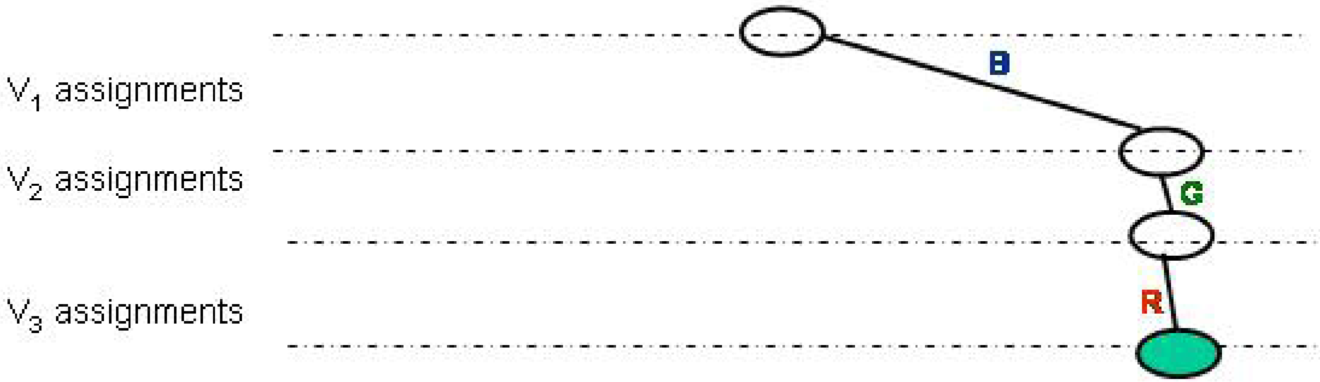
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



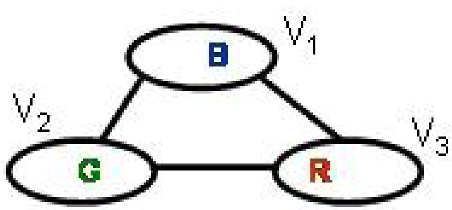


Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i=d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



No need to check previous assignments



Generally preferable to pure BT

BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

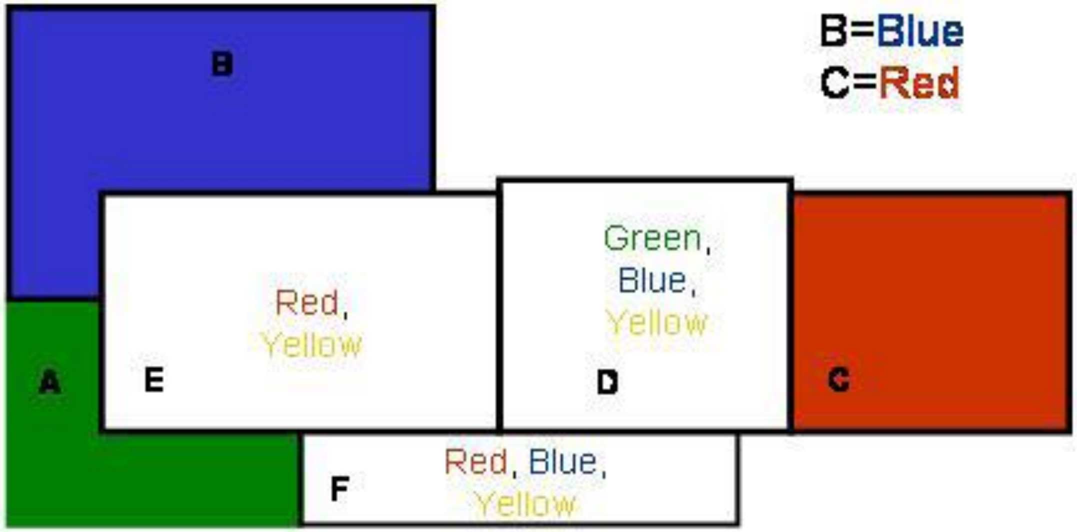
- **Most constrained variable**
  when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)

- **Least constraining value**
  choose value that rules out the fewest values from neighboring domains

E.g. this combination improves feasible n-queens performance from about n = 30 with just FC to about n = 1000 with FC & ordering.

Colors: R, G, B, Y

A=Green
B=Blue
C=Red

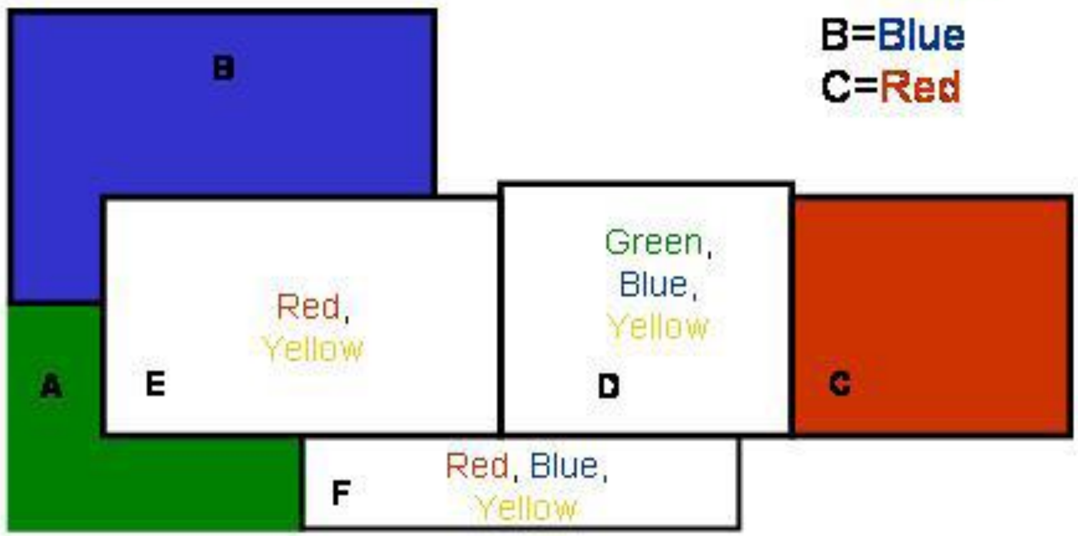


Which country should we color next →

What color should we pick for it? →

Colors: R, G, B, Y

A=Green
B=Blue
C=Red



Which country should we color next → E most-constrained variable (smallest domain)

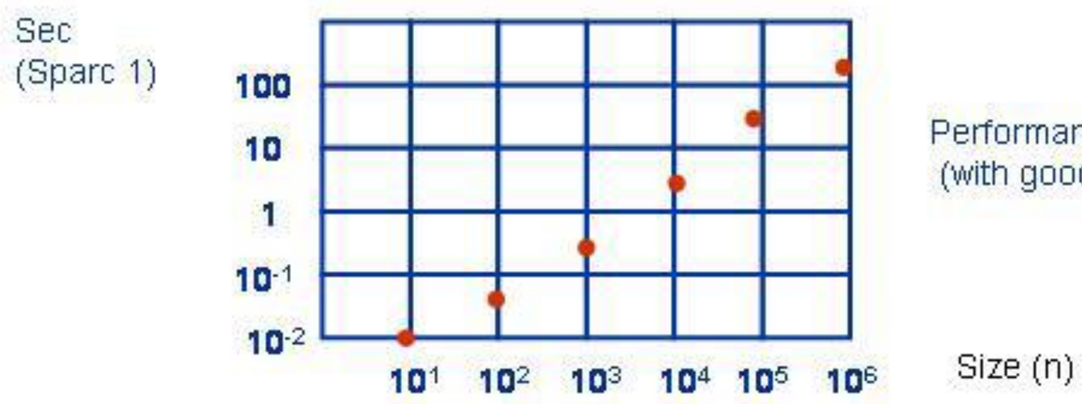
What color should we pick for it? → RED least-constraining value (eliminates fewest values from neighboring domains)

Incremental Repair (min-conflict heuristic)

1. Initialize a candidate solution using "greedy" heuristic – get solution "near" correct one.

2. Select a variable in conflict and assign it a value that minimizes the number of conflicts (beak ties randomly).

Can use this heuristic as part of systematic backtracker that uses heuristics to do value ordering or in a local hill-climber (without backup).



Performance on n-queens.
(with good initial guesses)

Min-conflict heuristic

The pure hill climber (without backtracking) can get stuck in local minima. Can add random moves to attempt getting out of minima – generally quite effective. Can also use weights on violated constraints & increase weight every cycle it remains violated.

GSAT

Randomized hill climber used to solve SAT problems. One of the most effective methods ever found for this problem

GSAT as Heuristic Search

- State space: Space of all full assignments to variables
- Initial state: A random full assignment
- Goal state: A satisfying assignment
- Actions: Flip value of one variable in current assignment
- Heuristic: The number of satisfied clauses (constraints); we want to maximize this. Alternatively, minimize the number of unsatisfied clauses (constraints).

GSAT(F)

- For i=1 to Maxtries
    - Select a complete random assignment A
    - Score = number of satisfied clauses
    - For j=1 to Maxflips
        - If (A satisfies all clauses in F) return A
        - Else flip a variable that maximizes score
        - Flip a randomly chosen variable if no variable flip increases the score.

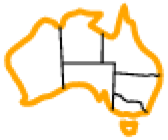
WALKSAT(F)

- For i=1 to Maxtries
  - Select a complete random assignment A
  - Score = number of satisfied clauses
  - For j=1 to Maxflips
    - If (A satisfies all clauses in F) return A
    - Else
      - With probability p /* GSAT */
        - » flip a variable that maximizes score
        - » Flip a randomly chosen variable if no variable flip increases the score.
      - With probability 1-p /* Random Walk */
        - » Pick a random unsatisfied clause C
        - » Flip a randomly chosen variable in C

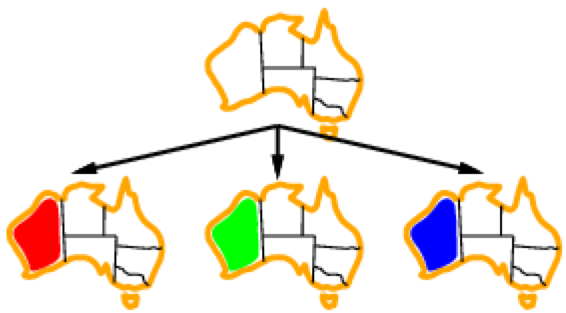
Backtracking search

- 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**

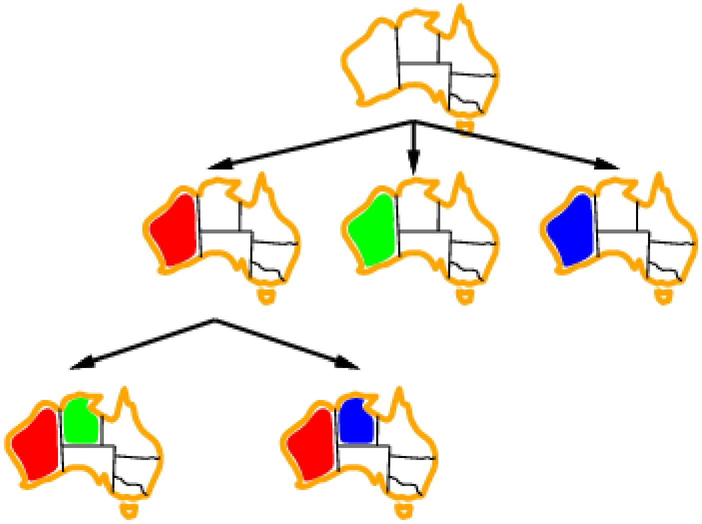
Example



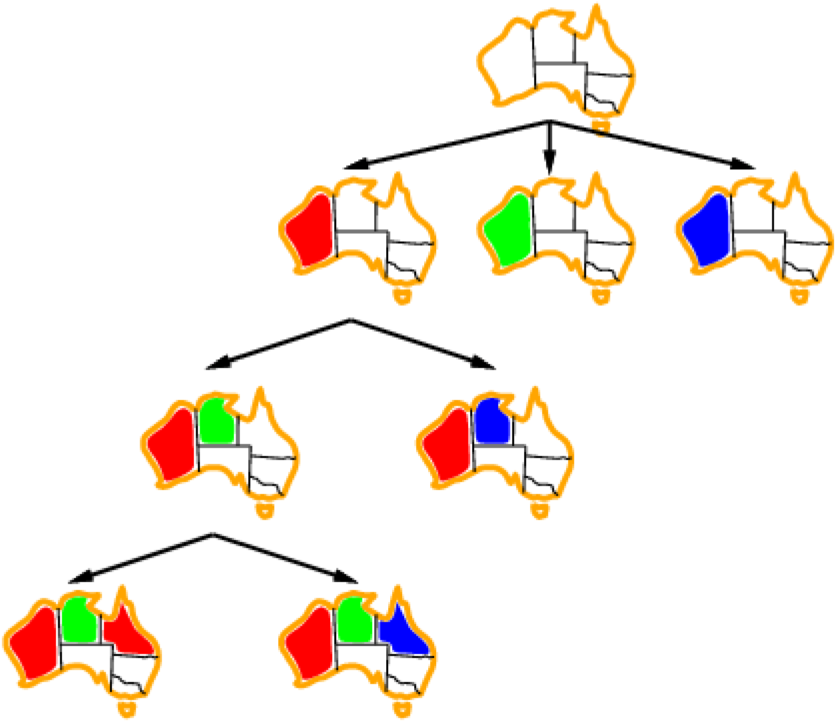
Example



Example



Example



Backtracking search algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
   if assignment is complete then return assignment
   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?

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?

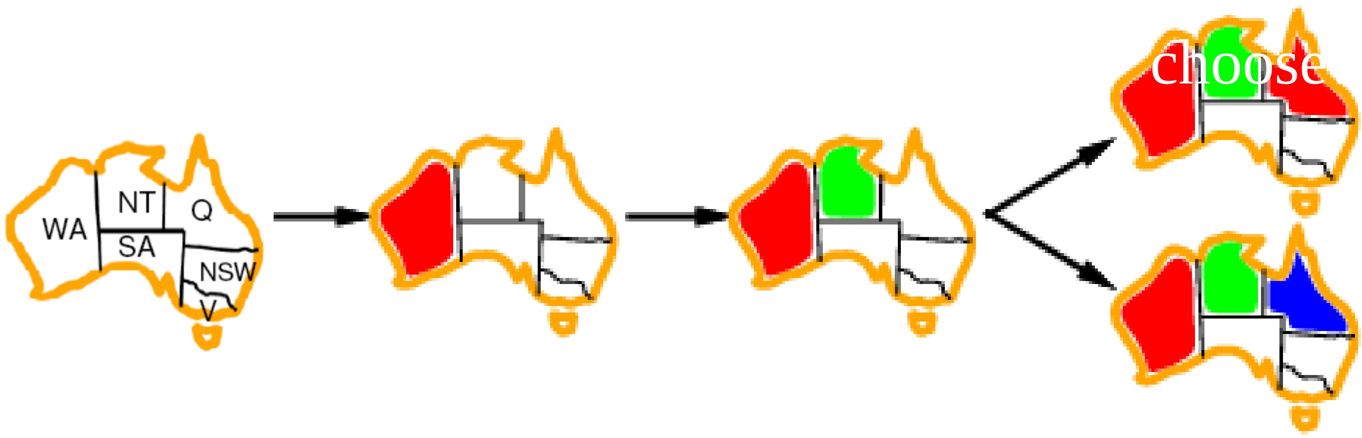
- **Most constraining variable:**
  - Choose the variable that imposes the most constraints on the remaining variables
  - Tie-breaker among most constrained variables

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

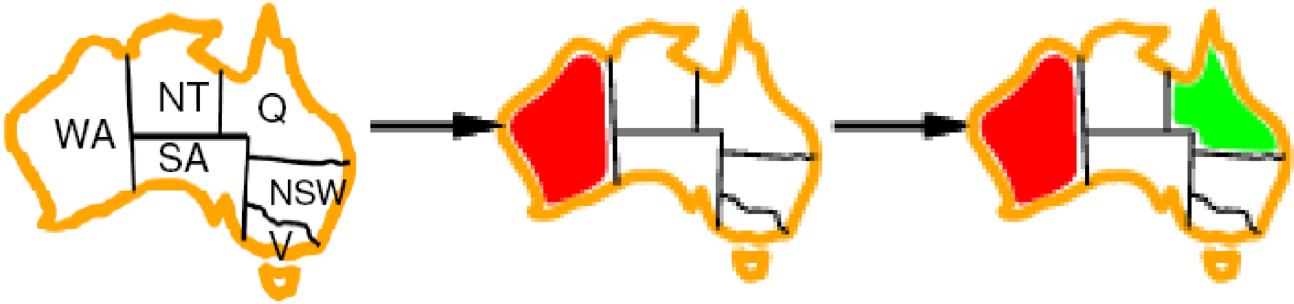


Early detection of failure

```
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



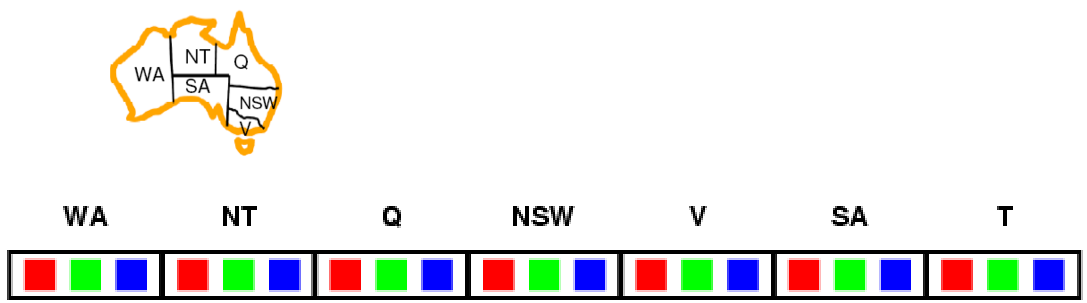
Apply *inference* to reduce the space of possible assignments and detect failure early

Early detection of failure: Forward checking

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

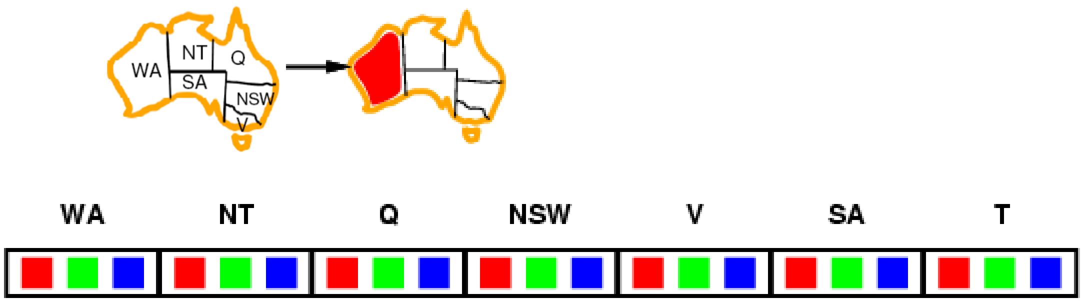
Early detection of failure: Forward checking

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



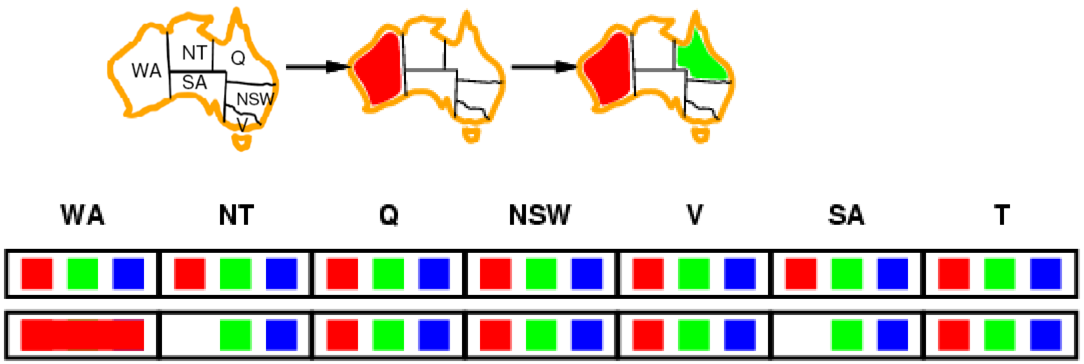
Early detection of failure: Forward checking

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



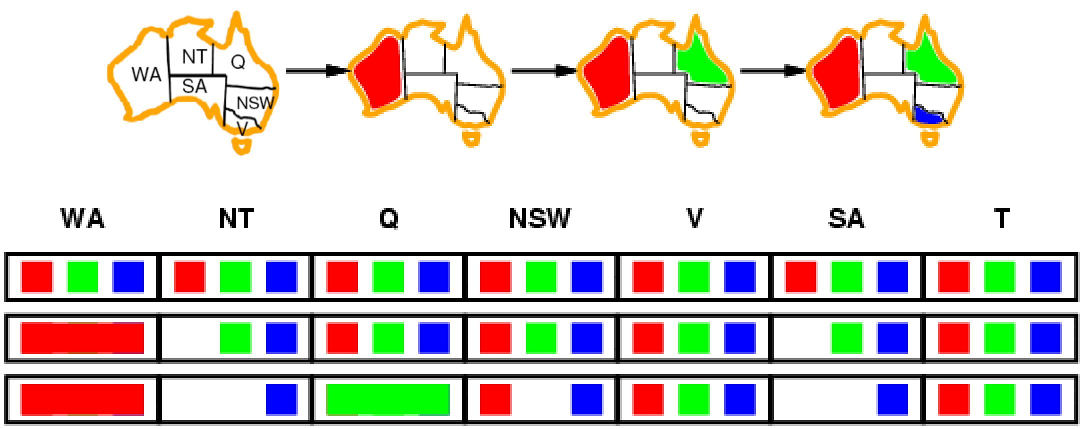
Early detection of failure: Forward checking

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



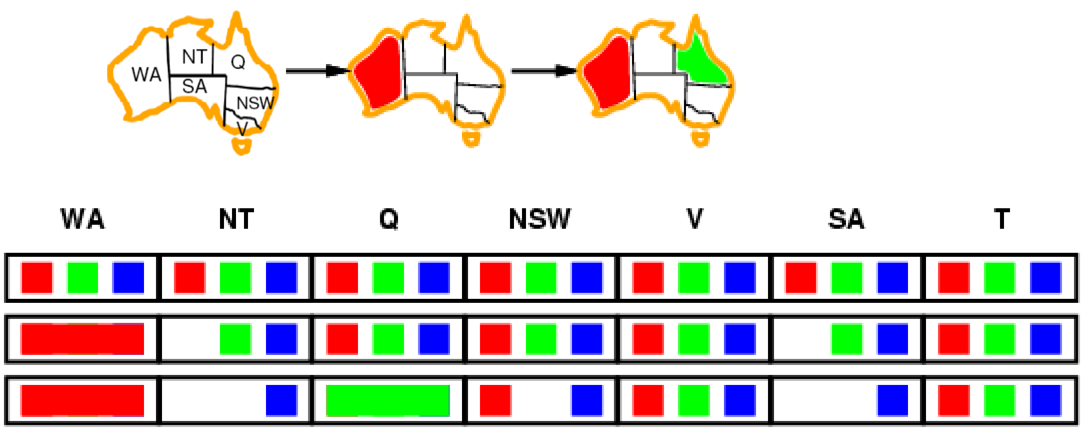
Early detection of failure: Forward checking

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



Constraint propagation

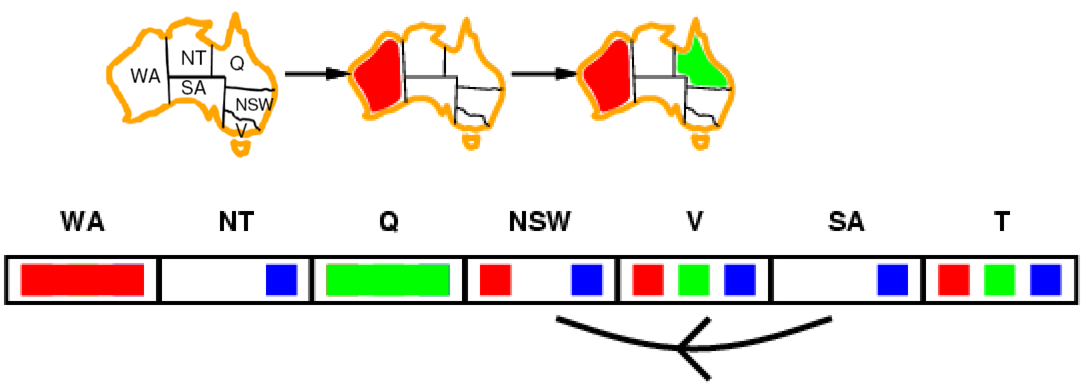
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures



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

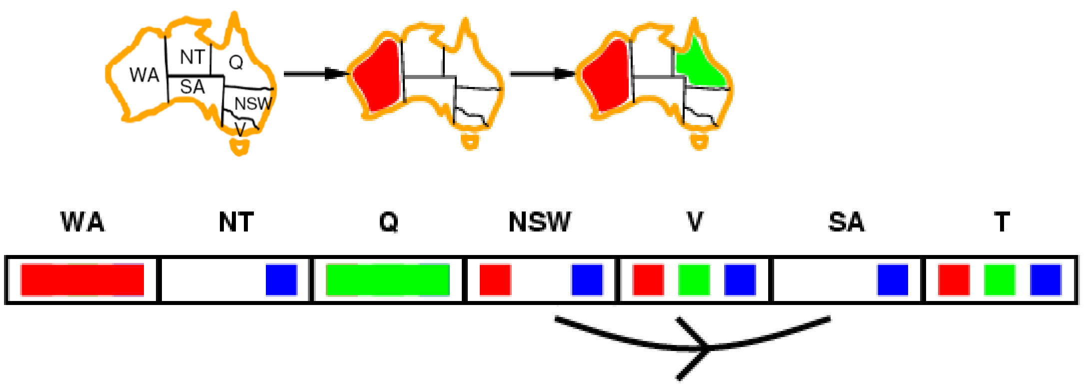
Arc consistency

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



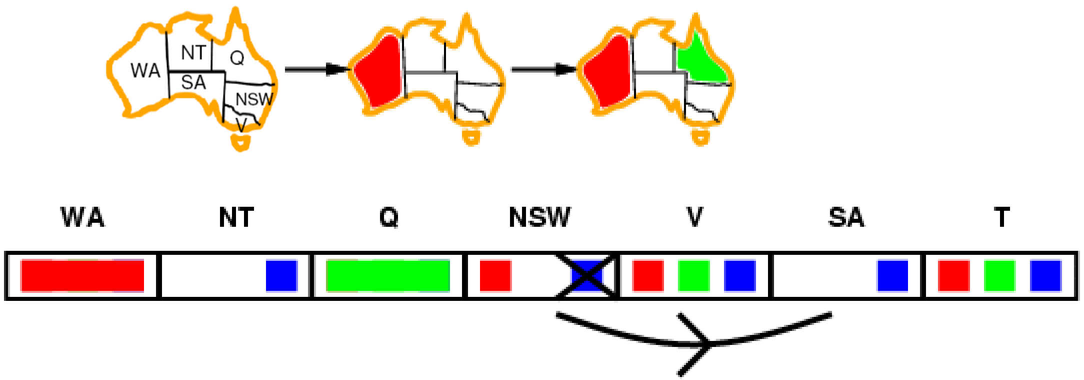
Arc consistency

- 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



Arc consistency

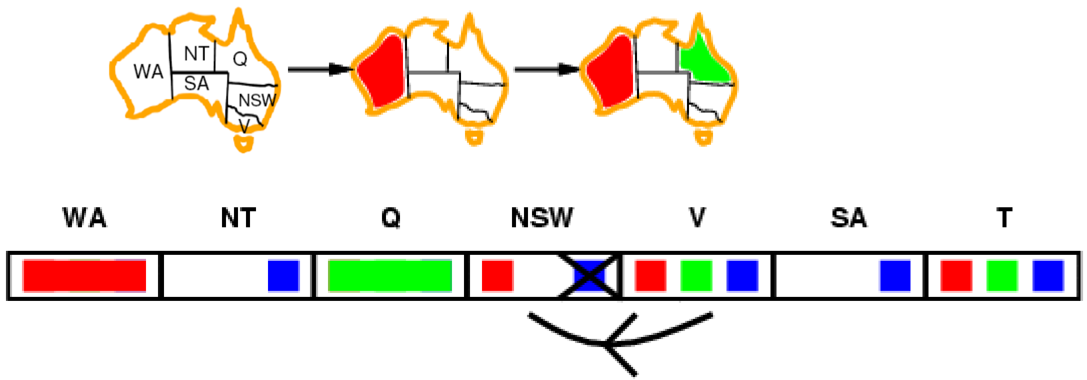
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

Arc consistency

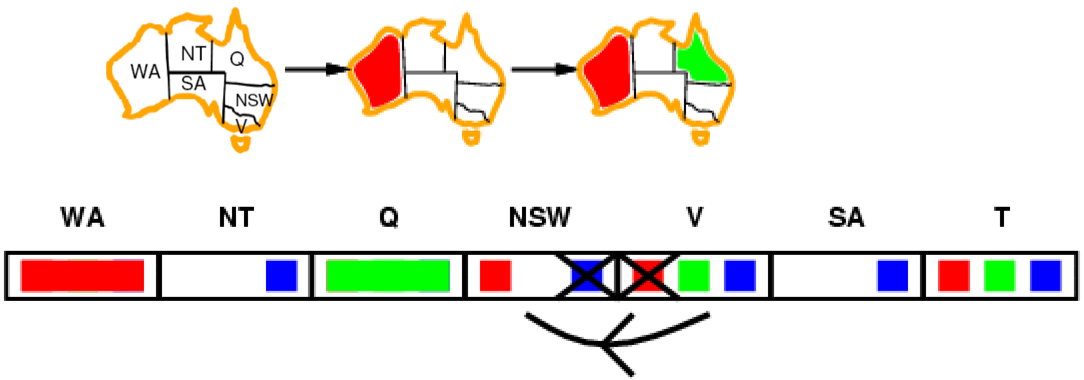
- Simplest form of propagation makes each pair of variables **consistent:**
  - *X* → *Y* is consistent iff for every value of *X* there is some allowed value of *Y*
  - When checking *X* → *Y*, throw out any values of X for which there isn't an allowed value of Y



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

Arc consistency

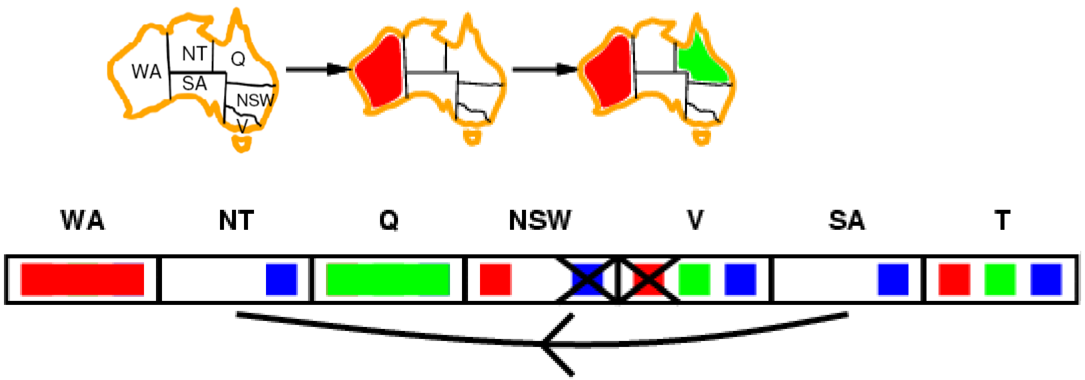
- Simplest form of propagation makes each pair of variables **consistent:**
  - $X \rightarrow Y$ is consistent iff for every value of X there is some allowed value of Y
  - When checking $X \rightarrow Y$, throw out any values of X for which there isn't an allowed value of Y



- If X loses a value, all pairs $Z \rightarrow X$ need to be rechecked

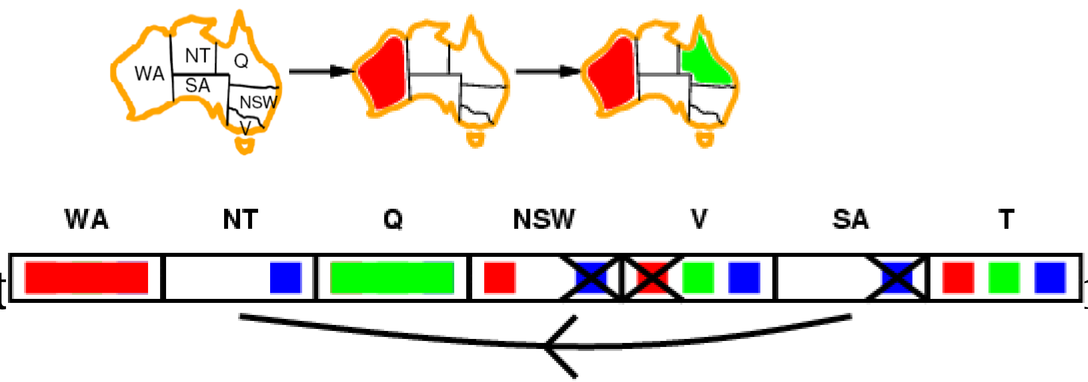
Arc consistency

- 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

- 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 consist hecking
- Can be run

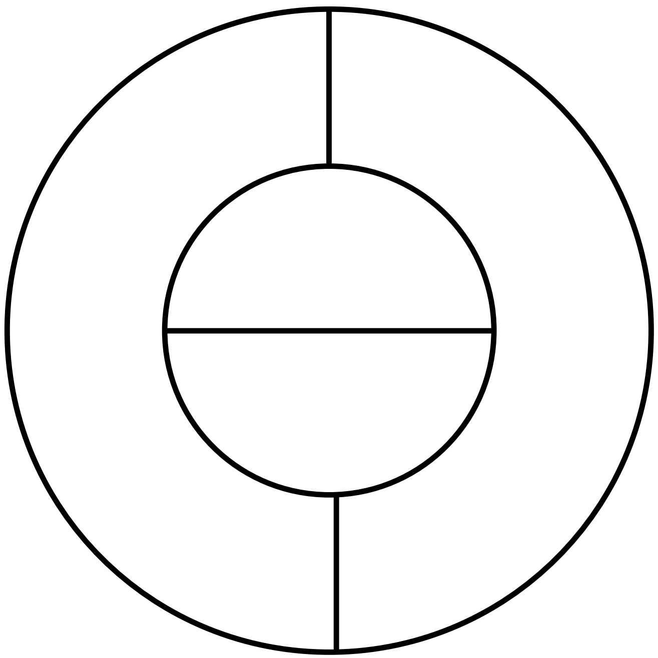
Arc consistency algorithm AC-3

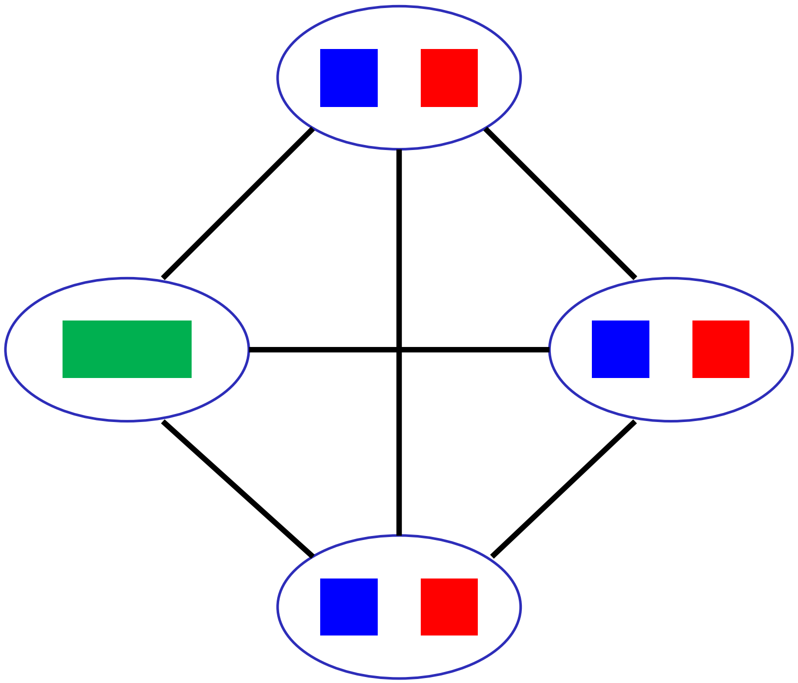
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty
      (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]
      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
```

Does arc consistency always detect the lack of a solution?

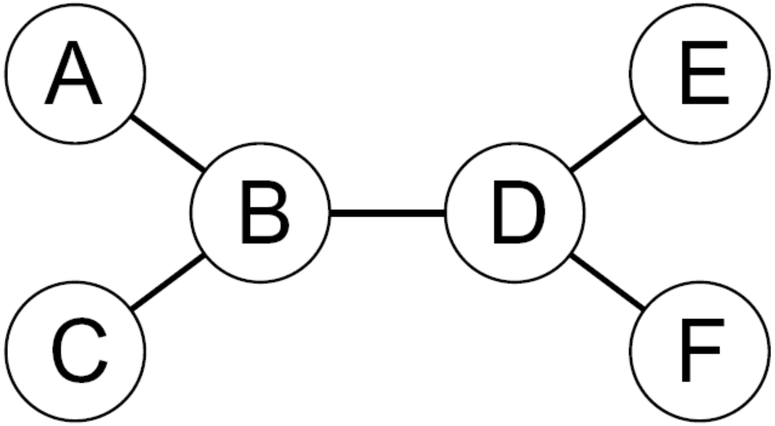




- There exist stronger notions of consistency (path consistency, k-consistency), but we won't worry about them

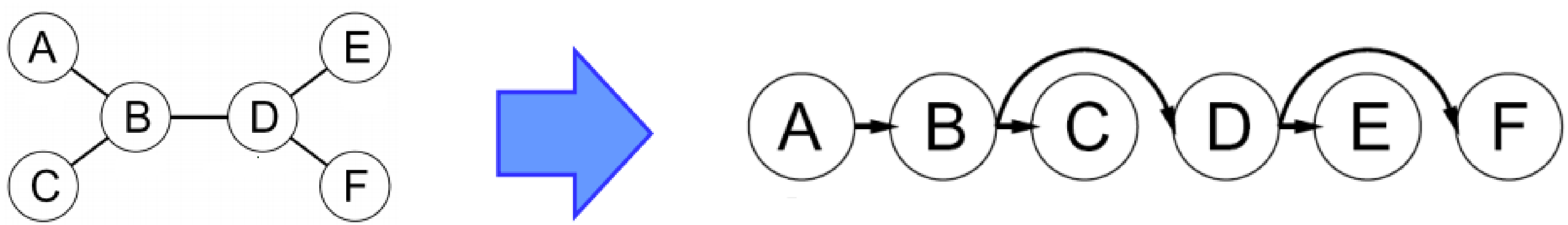
Tree-structured CSPs

- Certain kinds of CSPs can be solved without resorting to backtracking search!
- *Tree-structured CSP*: constraint graph does not have any loops



Algorithm for tree-structured CSPs

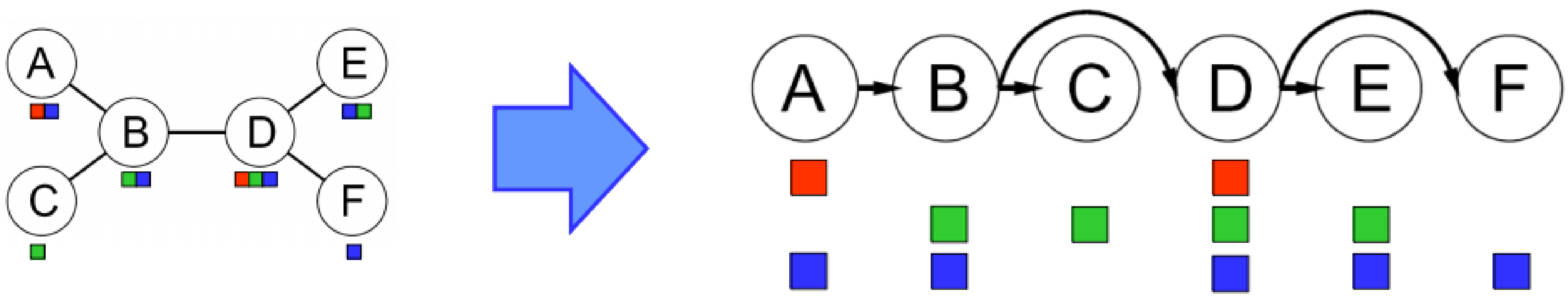
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

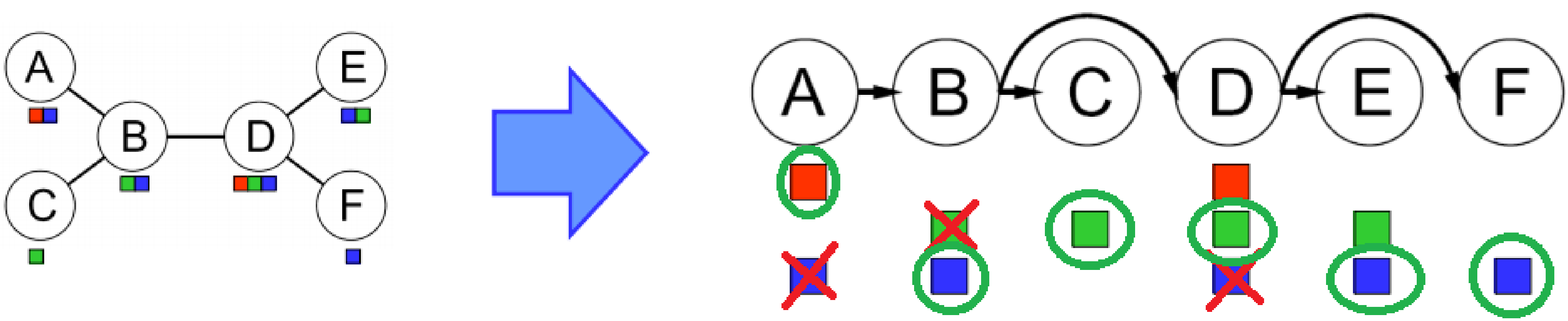
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards



http://cs188ai.wikia.com/wiki/Tree_Structure_CSPs

Algorithm for tree-structured CSPs

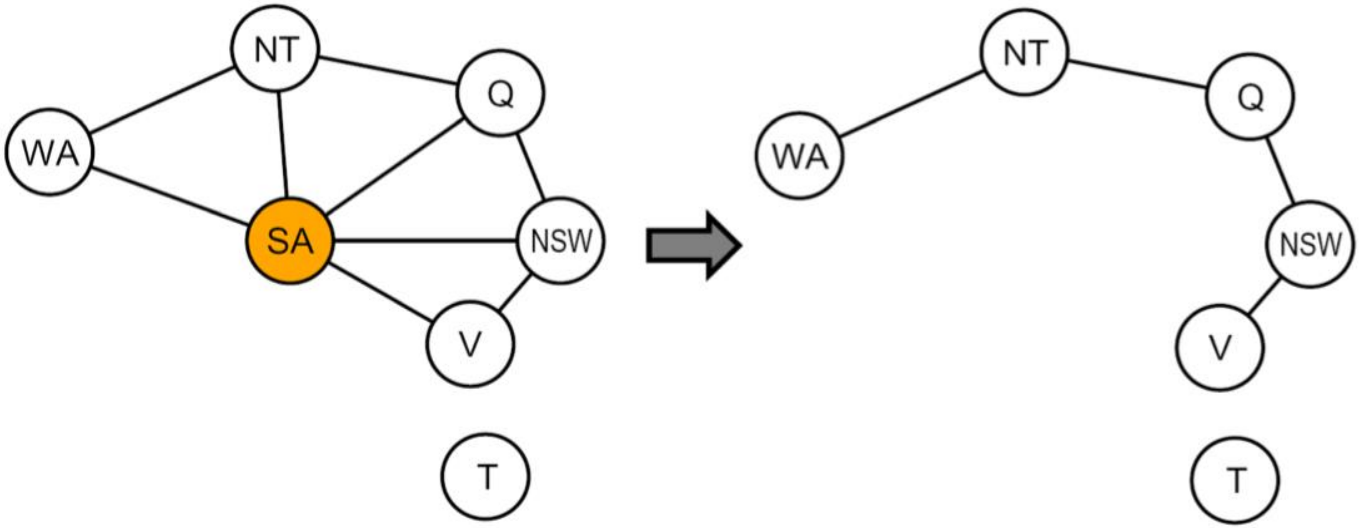
- Choose one variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
- Backward removal phase: check arc consistency starting from the rightmost node and going backwards
- Forward assignment phase: select an element from the domain of each variable 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 numebr of variables and m is the domain size, what is the running time of this algorithm?
  - O(nm^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 $O(m^c (n - c) m^2)$

Algorithm for tree-structured CSPs

- Running time is $O(nm^2)$
  (n is the number of variables, m 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 $O(m^n)$
- Can we do better?

Computational complexity of CSPs

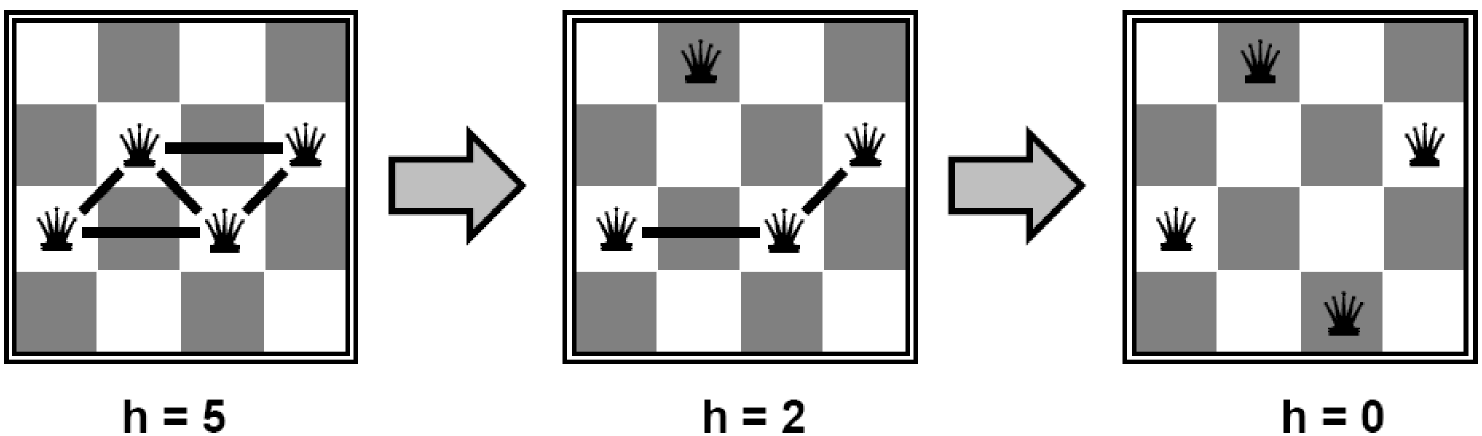
- The satisfiability (SAT) problem:
  - Given a Boolean formula, is there an assignment of the variables that makes it evaluate to true?

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

- SAT is *NP-complete*
  - NP: class of decision problems for which the "yes" answer can be verified 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

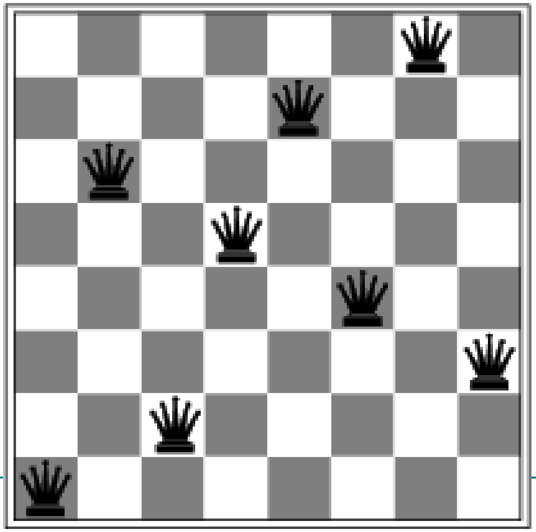
Local search for CSPs

- 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



Local search for CSPs

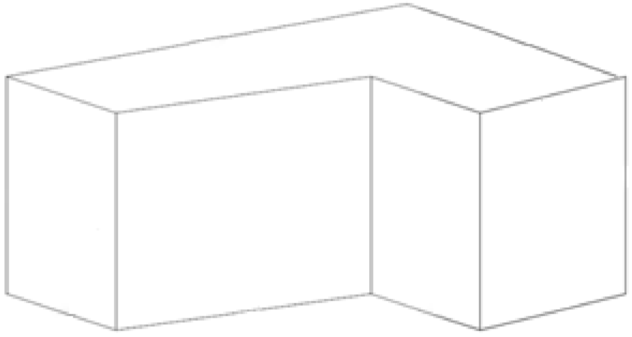
- Start with “complete” states, i.e., all variables assigned
- Allow states with unsatisfied constraints
- Attempt to improve states by reassigning variable values
- Hill-climbing search:
  - In each iteration, randomly select any conflicted variable and choose value that violates the fewest constraints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

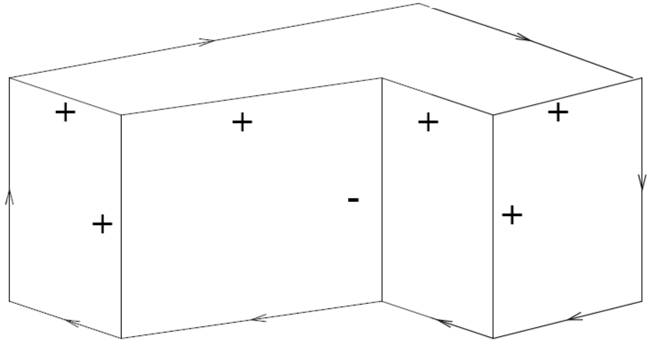


Local search for CSPs

- 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*
- For more on local search, see ch. 4

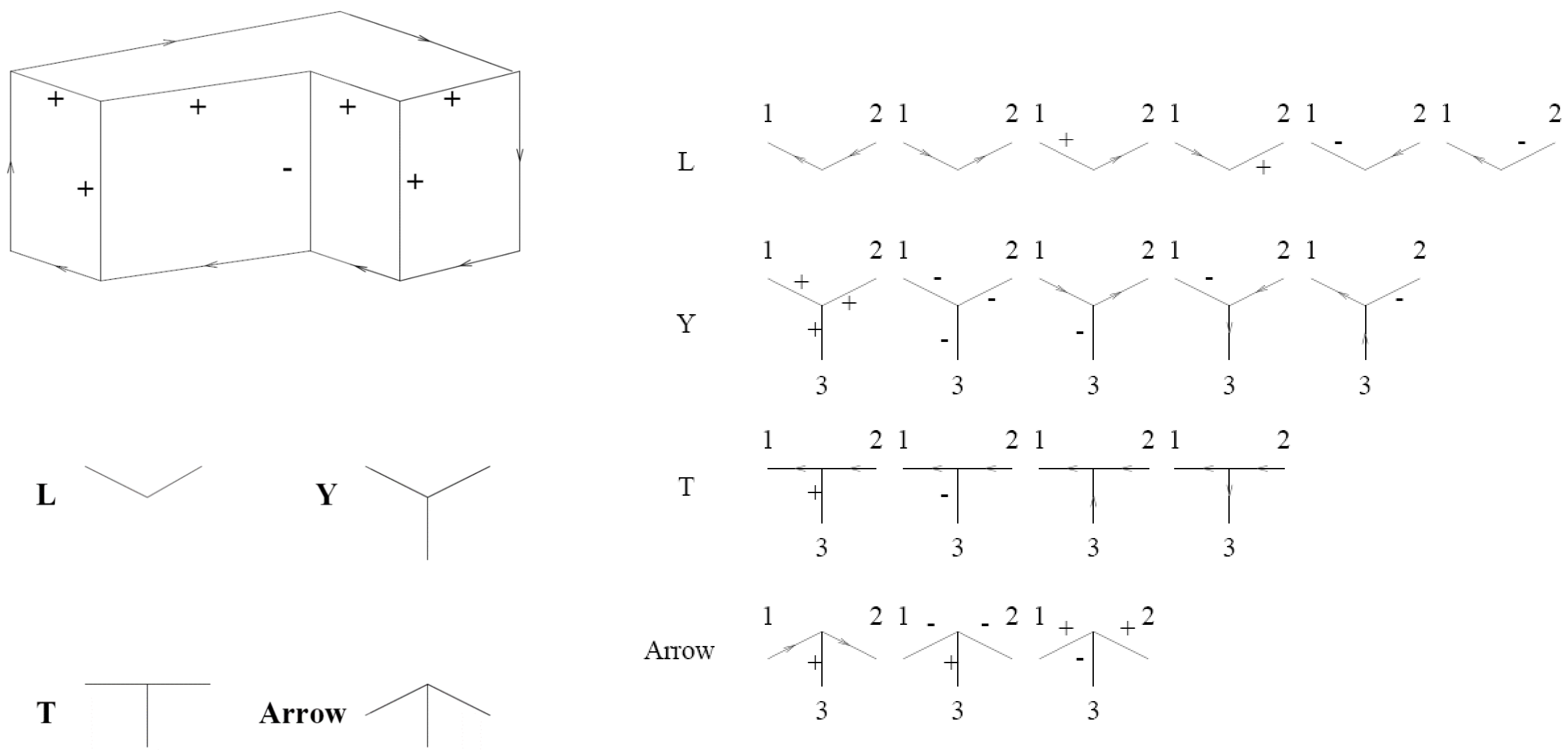
CSP in computer vision: Line drawing interpretation





David Waltz

CSP in computer vision: Line drawing interpretation

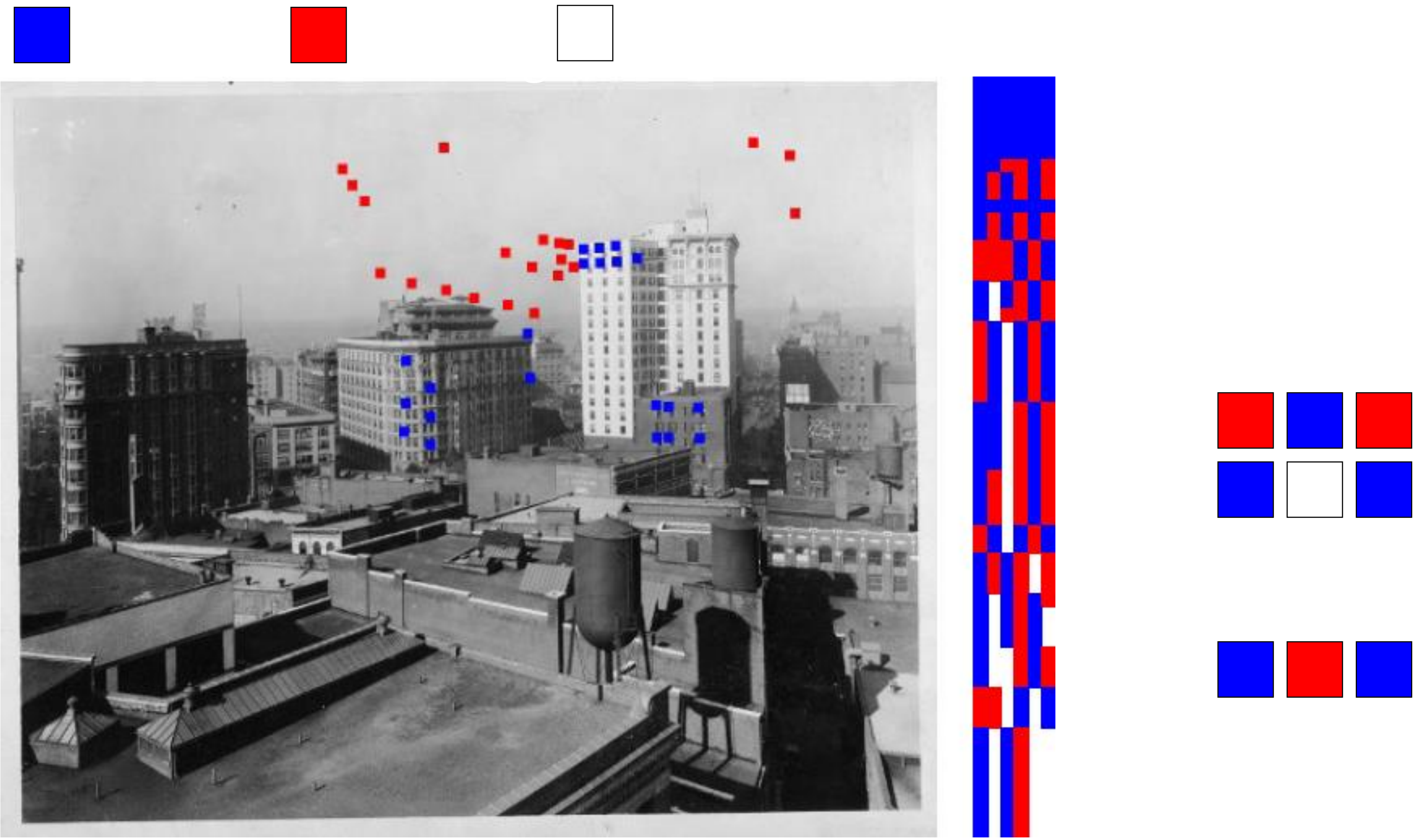


CSP in computer vision: 4D Cities

[Inferring Temporal Order of Images From 3D Structure](http://www.cc.gatech.edu/~phlosoft/)

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

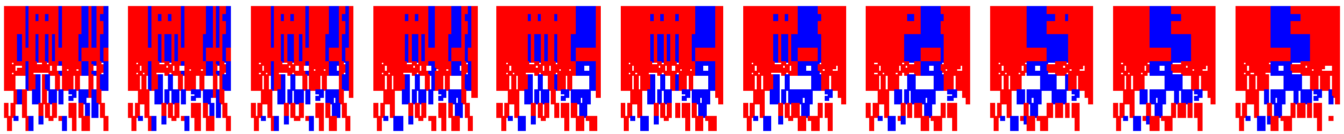
CSP in computer vision: 4D Cities



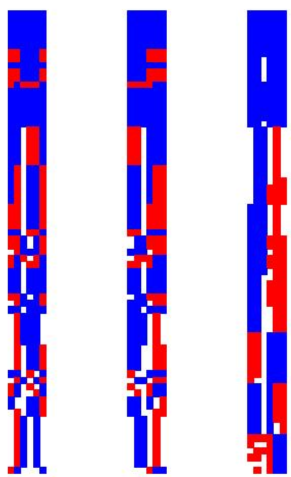
- Goal: reorder images (columns) to have as few violations as possible

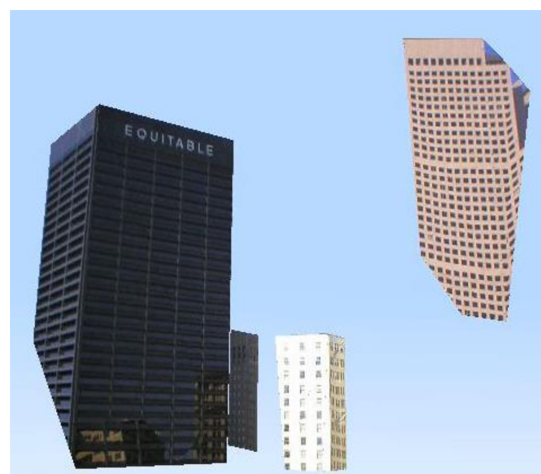
CSP in computer vision: 4D Cities

- **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