Chapter 6 Constraint Satisfaction Problems

CS5811 - Artificial Intelligence

Nilufer Onder
Department of Computer Science
Michigan Technological University

Outline

Constraint satisfaction problems (CSPs)

A *constraint satisfaction problem* consists of

- a finite set of *variables*, where each variable has a *domain*
  Using a set of variables (features) to represent a domain is called a *factored representation*.
- a set of *constraints* that restrict variables or combinations of variables

CSP example: cryptarithmetic

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

Variables: $F, T, U, W, R, O, X_1, X_2, X_3$
Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (same domain for all)
Sample constraints:

alldif (F, T, U, W, R, O)
or a binary constraint for all, e.g., $F \neq T, F \neq U$.
A unary constraint: $F \neq 0$
An n-ary constraint: $O + O = R + 10 \times X_1$
Can add constraints to restrict the X_i's to 0 or 1.

CSP example: solution

$$\begin{array}{r} \mathrm{T\,W\,O} \\ +\ \ \mathrm{T\,W\,O} \\ \hline \mathrm{F\,O\,U\,R} \end{array} \qquad\qquad \begin{array}{r} 7\,6\,5 \\ +\ \ 7\,6\,5 \\ \hline 1\,5\,3\,0 \end{array}$$

A *solution* is an assignment to all the variables from their domains so that all the constraints are satisfied.
For any CSP, there might be a single solution, multiple solutions, or no solutions at all.

Real-world CSPs

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

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

CSPs with discrete variables

- Finite domains
  $O(d^n)$ complete assignments are possible for
  n variables and domain size d
  e.g., Boolean CSPs, Boolean SATisfiability are *NP-complete*
- Infinite domains (integers, strings, etc.)
  e.g., job scheduling
  variables are start/end days for each job
  $StartJob_1 + 5 \leq StartJob_3$
  linear constraints are solvable,
  nonlinear constraints are *undecidable*

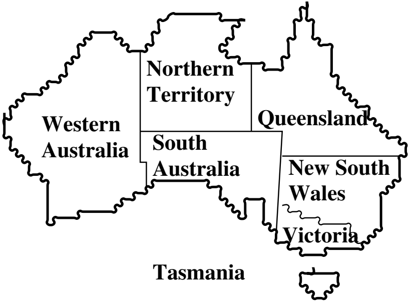
CSPs with continuous variables

- linear constraints solvable in polynomial time by *linear programming (LP)* methods
- e.g., precise start/end times for Hubble Telescope observations with astronomical, precedence, and power constraints

Representing CPSs as canonical search problems

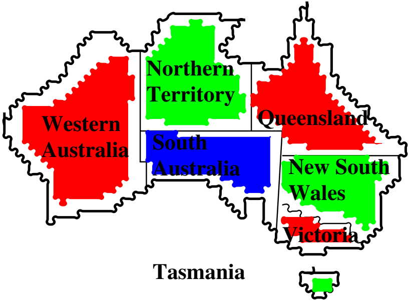
- Standard search problem:
  A *state* is a "black box", i.e, any old data structure
  that supports goal test, actions, result, etc.
- CSP:
  - A *state* is defined by *variables* X_i with *values* from *domains* D_i
    e.g., assigned: $\{F = 1\}$,
    unassigned $\{T, U, W, R, O, X_1, X_2, X_3\}$
  - The *goal test* is that
    all the variables are assigned
    all the constraints are satisfied
- Simple example of a *formal representation language*
- Allows useful *general-purpose algorithms* with more power
  than standard search algorithms:
  Can develop domain-independent heuristics

Working example: map-coloring



Variables: WA, NT, Q, NSW, V, SA, T
Domains: $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

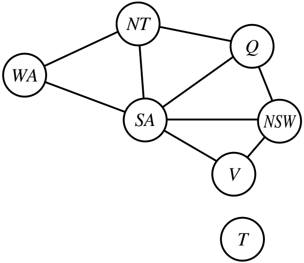
A solution for the map-coloring example



This solution satisfies all the constraints.

$$\{ \quad WA = red, NT = green, Q = red, NSW = green, \\ V = red, SA = blue, T = green\}$$

Constraint graph



- In a *binary CSP*, each constraint relates at most two variables
- A binary CSP can be represented as a *contraint graph*
- In the graph, the nodes are variables, the arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search.
  E.g., Tasmania is an independent subproblem

Working with the standard search process

Start with the straightforward approach, then fix it
States are defined by the values assigned so far

Initial state: the empty assignment, $\emptyset$
Actions: Pick an unassigned variable,
assign a value that does not conflict with the current assignments
If no assignment is possible, the path is a dead end
Goal test: all the variables have assignments

Working with the standard search process (cont'd)

- For a problem with n variables, every solution appears at depth n
- Depth-first search is a good choice
- A node that satisfies the goal test has the complete solution the path is not needed
- However, the branching factor is unnecessarily large ($b = (n - l)d$ at depth l)
- The search tree gets lots of redundant paths that represent the same solution but the order of assignment is different: $n!d^n$ leaves are produced

Backtracking search

- Variable assignments are *commutative*, i.e.,
  $WA = red$ 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
  $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called *backtracking search*
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

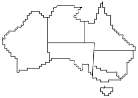
Backtracking search algorithm (1/2)

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

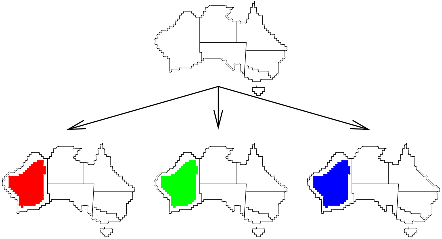
Backtracking search algorithm (2/2)

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

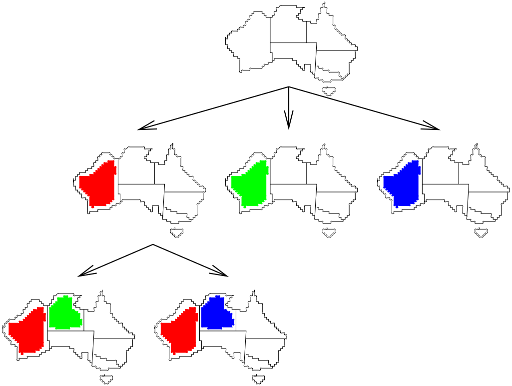
Backtracking example



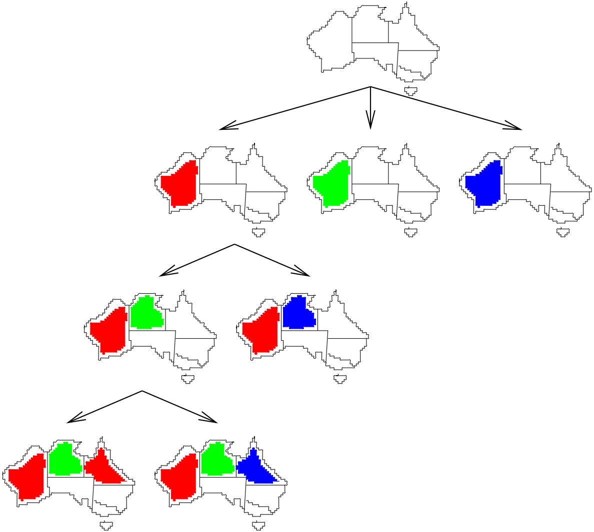
Backtracking example



Backtracking example



Backtracking example



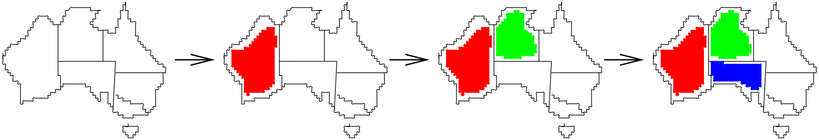
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Most constrained variable strategy

Most constrained variable:
choose the variable with the fewest legal values

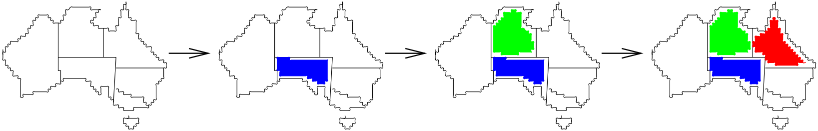


Most constraining variable strategy

Tie-breaker among most constrained variables

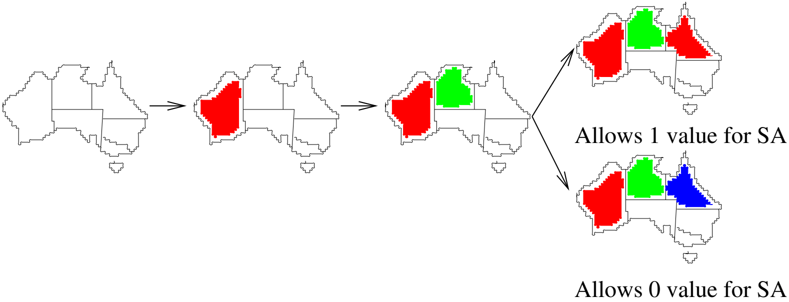
Most constraining variable:

choose the variable with the most constraints on the remaining variables



Least constraining value strategy

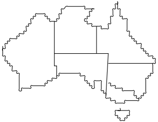
Given a variable, choose the *least constraining value*:
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible

Forward checking

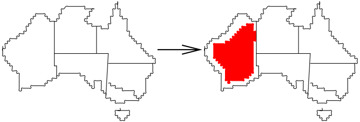
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦

Forward checking

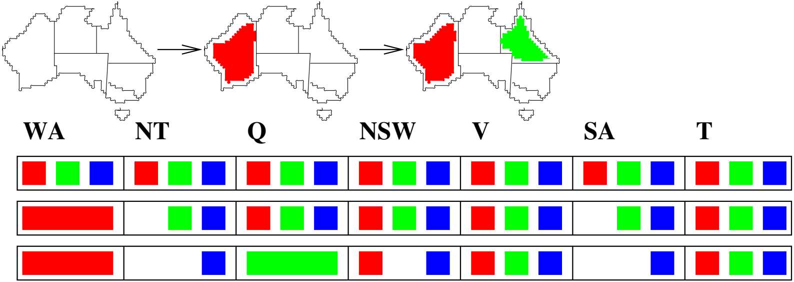
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦

Forward checking

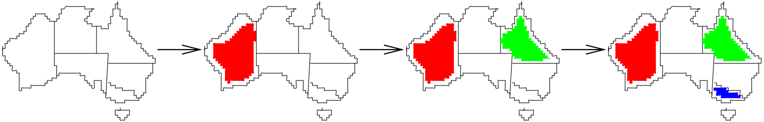
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

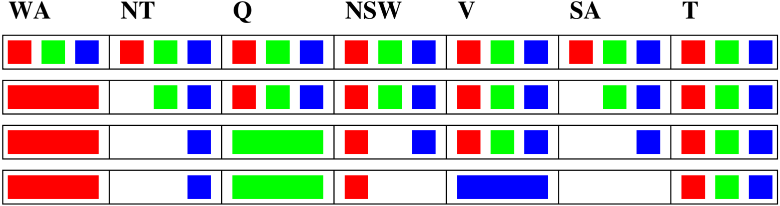


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

Forward checking

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

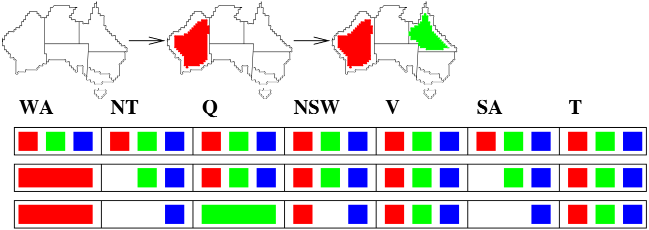




WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥	🟦		🟥 🟩 🟦

Constraint propagation

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



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

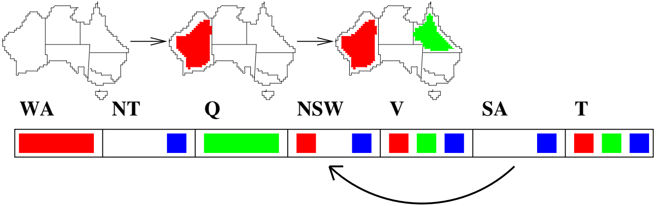
NT and *SA* cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency (1/4)

Simplest form of propagation makes each arc **consistent**

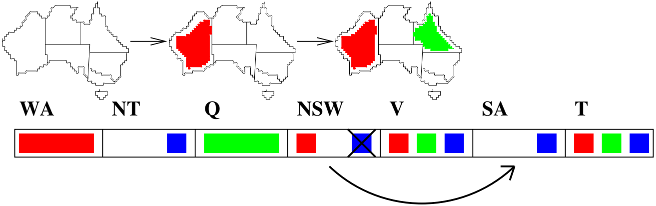
$X \rightarrow Y$ is *consistent* iff
for **every** value x of X there is **some** allowed y from Y



Arc consistency (2/4)

Simplest form of propagation makes each arc **consistent**

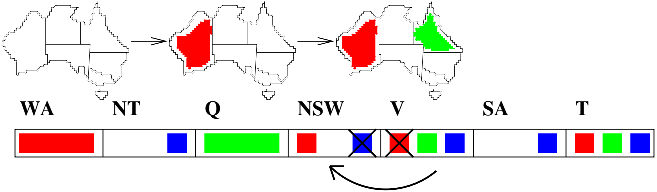
$X \rightarrow Y$ is *consistent* iff
for **every** value x of X there is **some** allowed y from Y



Arc consistency (3/4)

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is *consistent* iff
for **every** value x of X there is **some** allowed y from Y

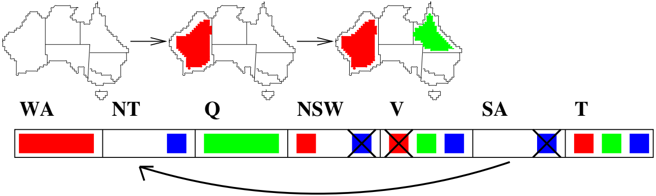


If X loses a value, neighbors of X need to be rechecked

Arc consistency (4/4)

Simplest form of propagation makes each arc **consistent**

$X \to Y$ is *consistent* iff
for **every** value x of X there is **some** allowed y from Y



If X loses a value, neighbors of X need to be rechecked
Arc consistency detects failure earlier than forward checking
Can be run as a preprocessor or after each assignment

Arc consistency algorithm

```
function AC-3 (csp)
returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X, D, C)$
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(queue)
    if REVISE(csp, $X_i, X_j$) then
       if size of $D_i = 0$ then return false
      for each $X_k$ in $X_i$.NEIGHBORS-$\{X_j\}$ do
        add $(X_k, X_i)$ to queue
  return true
```

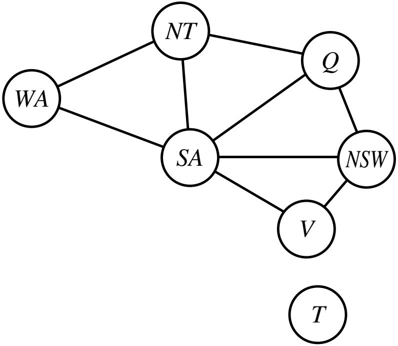
Arc consistency algorithm (cont'd)

```
function REVISE (csp, $X_i, X_j$)
returns true iff we revise the domain of $X_i$

 revised ← false
 for each $x$ in $D_i$ do
   if no value $y$ in $D_j$ allows $(x, y)$ to satisfy the
   constraint between $X_i$ and $X_j$
     then delete $x$ from $D_i$
          revised ← true
 return revised
```

$O(n^2 d^3)$, can be reduced to $O(n^2 d^2)$
But cannot detect all failures in polynomial time

Problem structure



Tasmania and mainland are *independent subproblems*

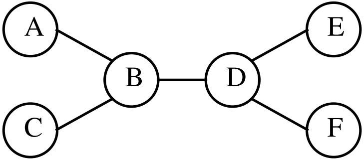
Identifiable as *connected components* of constraint graph

Problem structure (cont'd)

Suppose each subproblem has c variables out of n total
Worst-case solution cost is $n/c \cdot d^c$,**linear** in n
E.g., $n = 80$, $d = 2$, $c = 20$

2^{80} = 4 billion years at 10 million nodes/sec
4×2^{20} = 0.4 seconds at 10 million nodes/sec

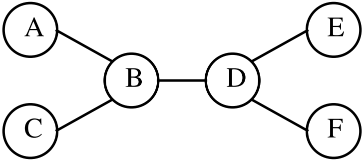
Tree-structured CSPs

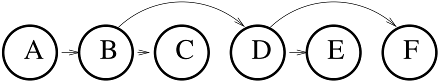


Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time
Compare to general CSPs, where worst-case time is $O(d^n)$
This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm for tree-structured CSPs

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





2. For j from n down to 2, apply
$\textsc{Make-Arc-Consistent}(Parent(X_j), X_j)$
(will remove inconsistent values)
3. For i from 1 to n, assign X_i consistently with $Parent(X_i)$

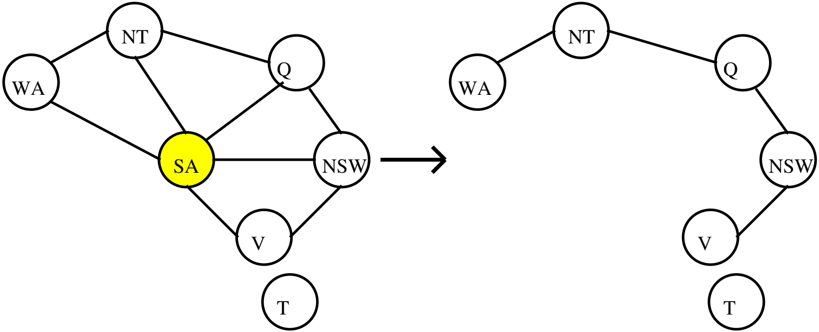
Algorithm for tree-structured CSPs (cont'd)

```
function TREE-CSP-SOLVER (csp)
returns a solution, or failure
  inputs: csp, a binary CSP with components $(X, D, C)$

  $n \leftarrow$ number of variables in $X$
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in $X$
  $X \leftarrow$ TOPOLOGICALSORT($X$, root)
  for $j = n$ down to 2 do
    MAKE-ARC-CONSISTENT(Parent($X_j$), $X_j$)
    if it cannot be made consistent then return failure
  for $i = 1$ to $n$ do
    assignment $[X_i] \leftarrow$ any consistent value from $D_i$
    if there is no consistent value then return failure
  return assignment
```

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Nearly tree-structured CSPs (cont'd)

Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \implies$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small c

Summary

- CSPs are a special kind of problem: states defined by values of a fixed set of variables goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure

Summary (cont'd)

- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- The CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- (Iterative min-conflicts is usually effective in practice)

Sources for the slides

- AIMA textbook (3^{rd} edition)
- AIMA slides (http://aima.cs.berkeley.edu/)
- Bartak, Roman. ICAPS-04 Tutorial on Constraint Satisfaction for Planning and Scheduling. 2004.