20th International Joint Conference on Artificial Intelligence

Hyderabad, India - January 6-12, 2007

IJCAI-07 Tutorial on

Constraint Processing

Roman Barták

Charles University (Czech Republic)

Preface

Solving combinatorial optimization problems like planning, scheduling, design, or configuration is a non-trivial task being attacked by many solving techniques. Constraint satisfaction, that emerged from AI research and nowadays integrates techniques from areas like operations research and discrete mathematics, provides a natural modeling framework for description of such problems supported by general solving technology. Though it is a mature area now, surprisingly many researchers outside the CSP community do not use the full potential of constraint satisfaction and frequently put equality between constraint satisfaction and simple enumeration. A nice example demonstrating the power of constraints are popular Sudoku problems that can be solved almost trivially by means of constraints, if proper technology is known.

The tutorial gives an introduction to mainstream constraint satisfaction techniques available in existing constraint solvers, namely constraint propagation combined with depth-first search, and answers the questions "How does constraint satisfaction work?" and "How to efficiently model problems using constraints?". The tutorial explains methods like arc consistency and shows how filtering algorithms are designed for constraints (this is a way how algorithms from other areas can be easily integrated into constraint solvers). Then it presents how consistency techniques are integrated with depth-first search algorithms and, finally, several modeling examples are given to demonstrate how constraints can be used in problem solving (including the popular Sudoku problems).

The tutorial is targeted to a broad AI community, in particular to everyone who is not familiar with the details of constraint satisfaction technology. It introduces novices as well as expert non-specialists to one of the major topics of AI. The tutorial also provides instructions how to use constraint satisfaction in problem solving. No prior knowledge of constraint satisfaction is required.

Author:

Name: *Roman Barták*
Address: *Charles University in Prague, Faculty of Mathematics and Physics*
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
Phone: *+420 221 914 242*
Fax: *+420 221 914 323*
e-mail: *roman.bartak@mff.cuni.cz*
WWW: *http://ktiml.mff.cuni.cz/~bartak*

Constraint Processing

Roman Barták
Charles University, Prague (CZ)

roman.bartak@mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Sudoku?

- **Logic-based puzzle,** whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

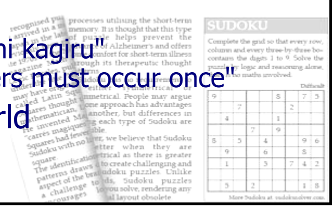
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

A bit of history

1979: first published in New York
under the name „Number Place"

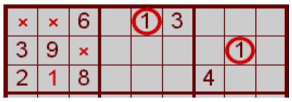
1986: became popular in Japan
Sudoku – from Japanes "**Su**dji wa **doku**shin ni kagiru"
"the numbers must be single" or "the numbers must occur once"

2005: became popular in the western world



Solving Sudoku

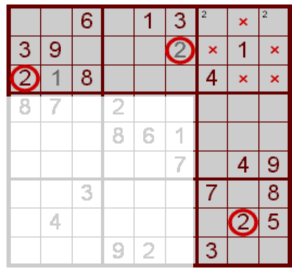
How to find out which digit to fill in?



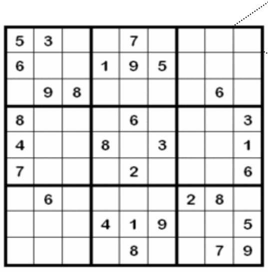
- Use information that each digit appears exactly once in each row and column.

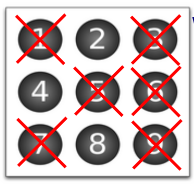
What if it is not enough?

- If rows and columns do not provide enough information then annotate each cell with possible digits that can be filled there.



Sudoku in general





We can see every cell as a **variable** with possible values from **domain** {1,…,9}.

There is a binary inequality **constraint** between all pairs of variables in every row, column, and sub-grid.

Such formulation of the problem is called a **constraint satisfaction problem.**

Tutorial outline

- **Introduction**
  - terminology, history, and application areas
- **Consistency techniques**
  (aka constraint propagation or domain filtering)
  - node, arc, and path consistencies
  - global constraints
  - design of filtering algorithms
- **Search techniques**
  - maintaining consistency during search
  - heuristics and discrepancy search
  - branch-and-bound for optimization
- **Modelling techniques**
  - how to effectively describe problems as CSPs
- **Conclusions**
  - constraint solvers

Introduction

CSP

Constraint satisfaction problem consists of:

- a finite set of **variables**
  - describe decision points of the problem, like the start time of activity
- **domains** - a finite set of values for each variable
  - describe possible options for the decisions, like a time window for the start time of activity
  - sometimes a single super domain is assumed and domains for variables are defined via unary constraints ($0 \leq start \leq 9$)
- a finite set of **constraints**
  - constraint is an arbitrary relation over the set of variables, like $startA + 15 \leq startB$
  - can be defined extensionally (a set of compatible tuples) or intentionally (formula)

Solution of a CSP

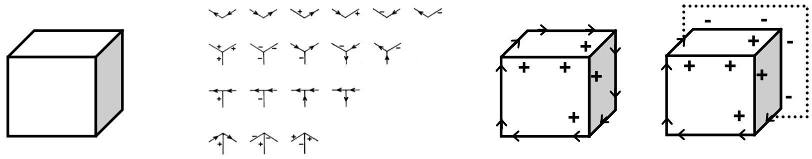
A **feasible solution** of a constraint satisfaction problem is a complete consistent assignment of variables.

- **complete** = a value is assigned to every variable
- **consistent** = all the constraints are satisfied

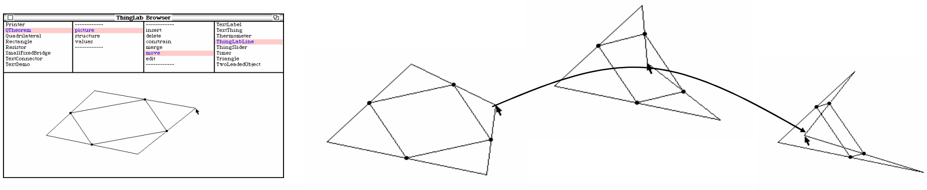
An **optimal solution** of a CSP is a feasible solution minimizing/maximizing a given objective function.

A bit of history

- **Scene labelling** (Waltz 1975)
  - feasible interpretation of 3D lines in a 2D drawing

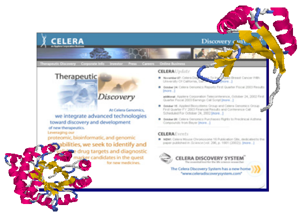


- **Interactive graphics** (Sutherland 1963, Borning 1981)
  - geometrical objects described using constraints



- **Logic programming** (Gallaire 1985, Jaffar, Lassez 1987)
  - from unification to constraint satisfaction

Application areas



Bioinformatics

- DNA sequencing
- 3D protein structures

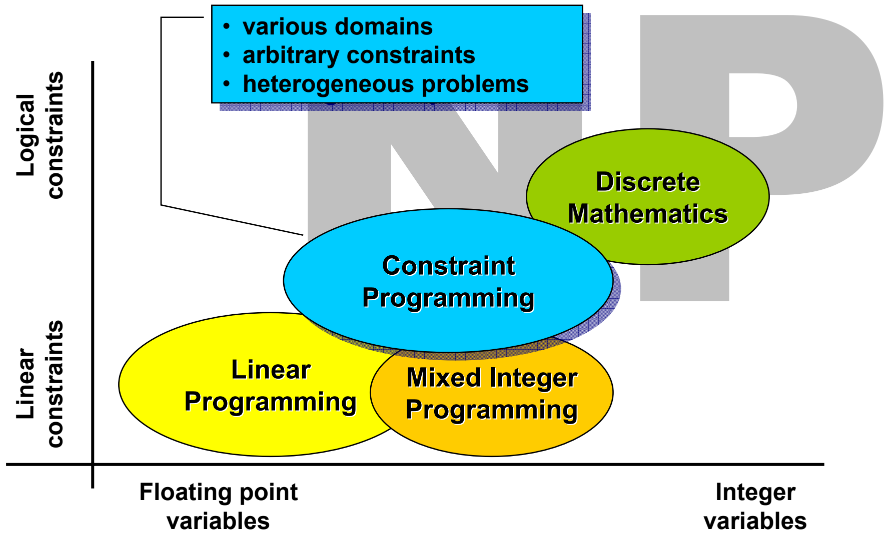
Planning

- Autonomous planning of spacecraft operations (Deep Space 1)

Production scheduling

- Saving after applying CSP: US$ 0.2 to 1 million per day

CP and others



Consistency

Active constraints

How can any constraint contribute to solving a CSP?

By actively removing inconsistencies (values, value tuples) that violate the constraint(s).

Example:

A in 3..7, B in 1..5	the variables' domains
A<B	the constraint

many inconsistent values can be removed

we get A in 3..4, B in 4..5

Note: it does not mean that all the remaining combinations of the values are consistent (for example A=4, B=4 is not consistent)

How to remove the inconsistencies from the constraint network?

Note:
Constraint network is a graph where nodes correspond to variables and (multi) arcs describe constraints.

Node consistency (NC)

Unary constraints are converted into variables' domains.

Definition:

- **The vertex** representing the variable X is **node consistent** iff every value in the variable's domain D_x satisfies all the unary constraints imposed on the variable X.
- **CSP** is **node consistent** iff all the vertices are node consistent.

Algorithm NC

```
procedure NC(G)
    for each variable X in nodes(G) do
        for each value V in the domain D_X do
            if unary constraint on X is inconsistent with V then
                    delete V from D_X
        end for
    end for
end NC
```

Arc consistency (AC)

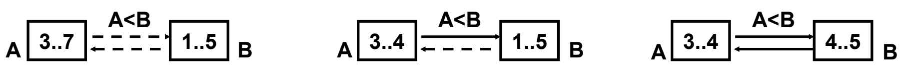
Since now we will assume binary CSPs only

i.e. a constraint corresponds to an arc (edge) in the constraint network.

Definition:

- **The arc (V_i,V_j)** is **arc consistent** iff for each value x from the domain D_i there exists a value y in the domain D_j such that the assignment V_i =x and V_j = y satisfies all the binary constraints on V_i, V_j.
  - **Note:** The concept of arc consistency is directional, i.e., arc consistency of (V_i,V_j) does not guarantee consistency of (V_j,V_i).
- **CSP** is **arc consistent** iff every arc (V_i,V_j) is arc consistent (in both directions).

Example:



no arc is consistent | (A,B) is consistent | (A,B) and (B,A) are consistent

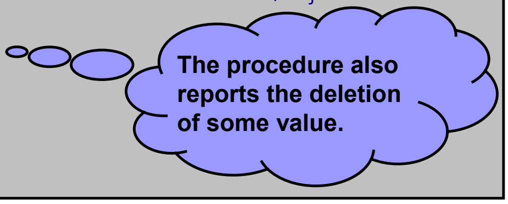
Arc revisions

How to make (V_i,V_j) arc consistent?

- Delete all the values x from the domain D_i that are inconsistent with all the values in D_j (there is no value y in D_j such that the assignment V_i = x, V_j = y satisfies all the binary constrains on V_i a V_j).

Algorithm of arc revision

procedure REVISE((i,j))
    DELETED ← false
    for each X in D_i **do**
        if there is no such Y in D_j such that (X,Y) is consistent, i.e., (X,Y) satisfies all the constraints on V_i, V_j **then**
            delete X from D_i
            DELETED ← true
        end if
    end for
    return DELETED
end REVISE



Mackworth (1977)

Algorithm AC-1

How to establish arc consistency among the constraints?

Doing revision of every arc is not enough!

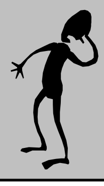
Example: X in [1,..,6], Y in [1,..,6], Z in [1,..,6], X<Y, Z<X-2

X in [1,..,6] Y in [1,..,6] Z in [1,..,6]	$\xrightarrow{X<Y}$	X in [1,..,5] Y in [2,..,6] Z in [1,..,6]	$\xrightarrow{Z<X-2}$	X in [4,5] Y in [2,..,6] Z in [1,2]	$\xrightarrow{X<Y}$	X in [4,5] Y in [5,6] Z in [1,2]

Make all the constraints consistent until any domain is changed.

Algorithm AC-1

```
procedure AC-1(G)
    repeat
        CHANGED ← false
        for each arc (i,j) in G do
            CHANGED ← REVISE((i,j)) or CHANGED
        end for
    until not(CHANGED)
end AC-1
```

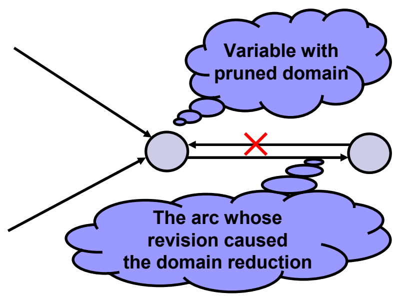


What is wrong with AC-1?

If a single domain is pruned then revisions of all the arcs are repeated even if the pruned domain does not influence most of these arcs.

What arcs should be reconsidered for revisions?

- The arcs whose consistency is affected by the domain pruning, i.e., the **arcs pointing to the changed variable**.
- **We can omit one more arc!**
  - Omit the arc running out of the variable whose domain has been changed (this arc is not affected by the domain change).



Mackworth (1977)

Algorithm AC-2

A generalised version of the **Waltz's labelling algorithm**.

In every step, the arcs going back from a given vertex are processed (i.e. a sub-graph of visited nodes is AC)

Algorithm AC-2

```
procedure AC-2(G)
    for i ← 1 to n do                              % n is a number of variables
        Q ← {(i,j) | (i,j)∈arcs(G), j<i}           % arcs for the base revision
        Q' ← {(j,i) | (i,j)∈arcs(G), j<i}          % arcs for re-revision
        while Q non empty do
            while Q non empty do
                select and delete (k,m) from Q
                if REVISE((k,m)) then
                    Q' ← Q' ∪ {(p,k) | (p,k)∈arcs(G), p≤i, p≠m }
            end while
            Q ← Q'
            Q' ← empty
        end while
    end for
end AC-2
```

Mackworth (1977)

Algorithm AC-3

Re-revisions can be done **more elegantly than in AC-2**.

1) **one queue** of arcs for (re-)revisions is enough

2) only the **arcs affected by domain reduction** are added to the queue (like AC-2)

Algorithm AC-3

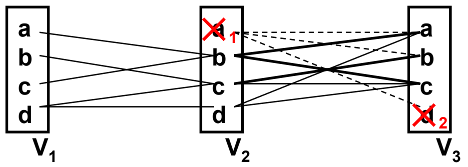
```
procedure AC-3(G)
    Q ← {(i,j) | (i,j)∈arcs(G), i≠j}     % queue of arcs for revision
    while Q non empty do
        select and delete (k,m) from Q
        if REVISE((k,m)) then
            Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
        end if
    end while
end AC-3
```

AC-3 schema is the most widely used consistency algorithm but it is still not optimal (time complexity is $O(ed^3)$).

Looking for a support

Observation (AC-3):

- Many pairs of values are tested for consistency in every arc revision.
- These tests are repeated every time the arc is revised.



1. When the arc V_2,V_1 is revised, the value a is removed from domain of V_2.
2. Now the domain of V_3, should be explored to find out if any value a,b,c,d loses the support in V_2.

Observation:

The values a,b,c need not be checked again because they still have a support in V_2 different from a.

The support set for $a \in D_i$ is the set $\{<j,b> \mid b \in D_j , (a,b) \in C_{i,j}\}$

Cannot we compute the support sets once and then use them during re-revisions?

Computing support sets

A set of values supported by a given value (if the value disappears then these values lost one support), and a number of own supports are kept.

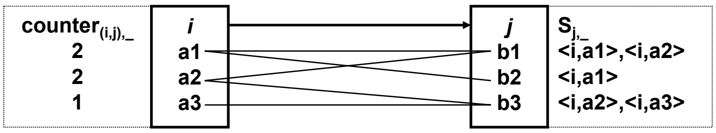
procedure INITIALIZE(G)
    $Q \leftarrow \{\}$, $S \leftarrow \{\}$ % emptying the data structures
    for each arc (V_i,V_j) in arcs(G) **do**
        for each a in D_i **do**
            total $\leftarrow$ 0
            for each b in D_j **do**
                if (a,b) is consistent according to the constraint $C_{i,j}$ **then**
                    total $\leftarrow$ total + 1
                    $S_{j,b} \leftarrow S_{j,b} \cup \{<i,a>\}$
                end if
            end for
            counter[(i,j),a] $\leftarrow$ total
            if counter[(i,j),a] = 0 **then**
                delete a from D_i
                $Q \leftarrow Q \cup \{<i,a>\}$
            end if
        end for
    end for
    return Q
end INITIALIZE

> ***$S_{j,b}$*** **- a set of pairs <i,a> such that <j,b> supports them**
>
> ***counter[(i,j),a]*** **- number of supports for the value *a* from D_i in the variable V_j**

Using support sets

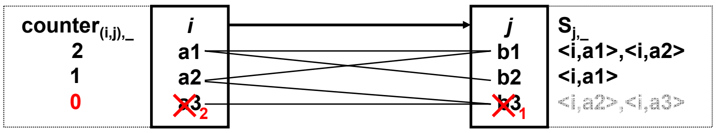
Situation:

we have just processed the arc (i,j) in INITIALIAZE



Using the support sets:

1. Let b3 is deleted from the domain of j (for some reason).
2. Look at $S_{j,b3}$ to find out the values that were supported by b3 (i.e. <i,a2>,<i,a3>).
3. Decrease the counter for these values (i.e. tell them that they lost one support).
4. If any counter becomes zero (a3) then delete the value and repeat the procedure with the respective value (i.e., go to 1).



Mohr, Henderson (1986)

Algorithm AC-4

The algorithm AC-4 has **optimal worst case time complexity O(ed²)**!

Algorithm AC-4

procedure AC-4(G)
  Q ← INITIALIZE(G)
  while Q non empty **do**
    select and delete any pair <j,b> from Q
    for each <i,a> from $S_{j,b}$ **do**
      counter[(i,j),a] ← counter[(i,j),a] - 1
      if counter[(i,j),a] = 0 & "a" is still in D_i **then**
        delete "a" from D_i
        Q ← Q ∪ {<i,a>}
      end if
    end for
  end while
end AC-4

Unfortunately the **average time complexity is not so good**

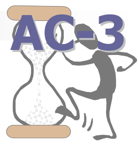
… plus there is a **big memory consumption**!

Other AC algorithms

- **AC-5 (Van Hentenryck, Deville, Teng, 1992)**
  - generic AC algorithm covering both AC-4 and AC-3
- **AC-6 (Bessière, 1994)**
  - improves AC-4 by remembering just one support
- **AC-7 (Bessière, Freuder, Régin, 1999)**
  - improves AC-6 by exploiting symmetry of the constraint
- **AC-2000 (Bessière & Régin, 2001)**
  - an adaptive version of AC-3 that either looks for a support or propagates deletions
- **AC-2001 (Bessière & Régin, 2001)**
  - improvement of AC-3 to get optimality (queue of variables)
- **AC-3.1 (Zhang & Yap, 2001)**
  - improvement of AC-3 to get optimality (queue of constraints)

…

AC-3.1: optimal AC-3



- Some observations:
  - AC-3 is not (theoretically) optimal
  - AC-4 is (theoretically) optimal but (practically) slow
  - AC-6/7 are (practically) faster than AC-4, but quite complicated
- What is inefficient in AC-3?
  - Looking for supports in REVISE starts from scratch!

if „there is no such Y in D_j such that (X,Y) is consistent" then

- AC-3.1 (Zhang, Yap)
  - same run as AC-3
  - but for each value, it remembers the last support in the constraint and the next time, it starts looking for a support at this value

procedure EXIST((i,x),j)
  y ← last((i,x),j)
  if $y \in D_j$ **then** return true
  while $y \leftarrow next(y,D_j)$ & $y \neq nil$ **do**
    if $(x,y) \in C(i,j)$ **then**
      last((i,x),j) ← y
      return true
  end while
  return false
end EXIST

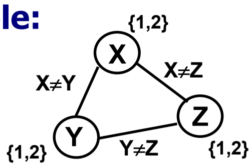
Is arc consistency enough?

By using AC we can remove many incompatible values

- **Do we get a solution?**
- **Do we know that there exists a solution?**

Unfortunately, the answer to both above questions is NO!

Example:



CSP is arc consistent but there is no solution

So what is the benefit of AC?

Sometimes we have a solution after AC

- any domain is empty → no solution exists
- all the domains are singleton → we have a solution

In general, AC prunes the search space.

Path consistency (PC)

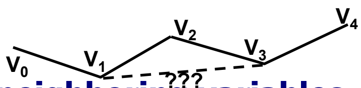
How to strengthen the consistency level?

More constraints are assumed **together**!

Definition:

- **The path** $(V_0, V_1, \ldots, V_m)$ is **path consistent** iff for every pair of values $x \in D_0$ a $y \in D_m$ satisfying all the binary constraints on V_0, V_m there exists an assignment of variables $V_1, \ldots, V_{m-1}$ such that all the binary constraints between the neighbouring variables V_i, V_{i+1} are satisfied.
- CSP is **path consistent** iff every path is consistent.

Some notes:



- only the **constraints between the neighboring variables** must be satisfied
- it is enough to explore **paths of length 2** (Montanary, 1974)

Relation between PC and AC

Does PC subsume AC (i.e. if CSP is PC, is it AC as well)?

- the arc (i, j) is consistent (AC) if the path **(i,j,i)** is consistent (PC)
- thus PC implies AC

Is PC stronger than AC (is there any CSP that is AC but it is not PC)?

Example:

X in {1,2}, Y in {1,2}, Z in {1,2}, X≠Z, X≠Y, Y≠Z

- it is AC, but not PC (X=1, Z=2 cannot be extended to X,Y,Z)

AC removes incompatible values from the domains, what will be done in PC?

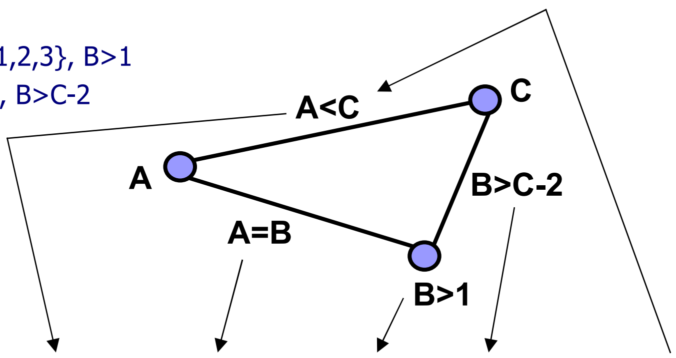
- PC removes pairs of values
- PC makes constraints explicit (A<B,B<C ⇒ A+1<C)
- a unary constraint = a variable's domain

Path revision

Constraints **represented extensionally** via matrixes.
Path consistency is realized via **matrix operations**

Example:

- A,B,C in {1,2,3}, B>1
- A<C, A=B, B>C-2



$$\begin{bmatrix}011\\001\\000\end{bmatrix} \;\&\; \begin{bmatrix}100\\010\\001\end{bmatrix} * \begin{bmatrix}000\\010\\001\end{bmatrix} * \begin{bmatrix}110\\111\\111\end{bmatrix} = \begin{bmatrix}000\\001\\000\end{bmatrix}$$

Mackworth (1977)

Algorithm PC-1

How to make the path (i,k,j) consistent?

$R_{ij} \leftarrow R_{ij}$ & $(R_{ik} * R_{kk} * R_{kj})$

How to make a CSP path consistent?

Repeated revisions of paths (of length 2) while any domain changes.

procedure PC-1(Vars,Constraints)
  $n \leftarrow |Vars|$, $Y^n \leftarrow$ Constraints
  repeat
    $Y^0 \leftarrow Y^n$
    for k = 1 to n **do**
      for i = 1 to n **do**
        for j = 1 to n **do**
          $Y^k_{ij} \leftarrow Y^{k-1}_{ij}$ & $(Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{kj})$
  until $Y^n = Y^0$
  Constraints $\leftarrow Y^0$
end PC-1

If we use
$Y^k_{ii} \leftarrow Y^{k-1}_{ii}$ **&** $(Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{ki})$
then we get AC-1

How to improve PC-1?

Is there any inefficiency in PC-1?

- just a few „bits"
  - it is not necessary to keep all copies of Y^k
    one copy and a bit indicating the change is enough
  - some operations produce no modification ($Y^k_{kk} = Y^{k-1}_{kk}$)
  - half of the operations can be removed ($Y_{ji} = Y^T_{ij}$)
- **the grand problem**
  - after domain change all the paths are re-revised
    but it is enough to revise just the influenced paths

Algorithm of path revision

procedure REVISE_PATH((i,k,j))
  $Z \leftarrow Y_{ij}$ & $(Y_{ik} * Y_{kk} * Y_{kj})$
  if $Z=Y_{ij}$ **then return** false
  $Y_{ij} \leftarrow Z$
  return true
end REVISE_PATH

If the domain is pruned then the influenced paths will be revised.

Influenced paths

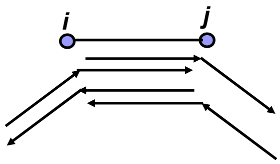
Because $Y_{ji} = Y^t_{ij}$ it is enough to revise only the paths (i,k,j) where i≤j.
Let the domain of the constraint (i,j) be changed when revising (i,k,j):

Situation a: i<j

all the paths containing (i,j) or (j,i) must be re-revised
but the paths (i,j,j), (i,i,j) are not revised again (no change)

S_a = {(i,j,m) | i ≤ m ≤ n & m≠j}
∪ {(m,i,j) | 1 ≤ m ≤ j & m≠i}
∪ {(j,i,m) | j < m ≤ n}
∪ {(m,j,i) | 1 ≤ m < i}

| S_a | = 2n-2



Situation b: i=j

all the paths containing i in the middle of the path are re-revised
but the paths (i,i,i) and (k,i,k) are not revised again

S_b = {(p,i,m) | 1 ≤ m ≤ n & 1 ≤ p ≤ m} - {(i,i,i),(k,i,k)}

| S_b | = n*(n-1)/2 - 2

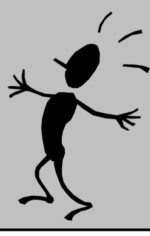
Mackworth (1977)

Algorithm PC-2

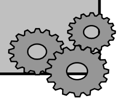
Paths in one direction only (attention, this is not DPC!)
After every revision, the **affected paths are re-revised**

Algorithm PC-2

```
procedure PC-2(G)
    n ← |nodes(G)|
    Q ← {(i,k,j) | 1 ≤ i ≤ j ≤ n & i≠k & j≠k}
    while Q non empty do
        select and delete (i,k,j) from Q
        if REVISE_PATH((i,k,j)) then
            Q ← Q ∪ RELATED_PATHS((i,k,j))
    end while
end PC-2
```



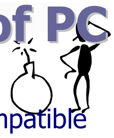
```
procedure RELATED_PATHS((i,k,j))
    if i<j then return S_a else return S_b
end RELATED_PATHS
```



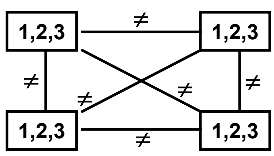
Other PC algorithms

- **PC-3 (Mohr, Henderson 1986)**
  - based on computing supports for a value (like AC-4)
    - If the pair (a,b) at the arc (i,j) is not supported by another variable, then a is removed from D_i and b is removed from D_j.
  - **this algorithm is not sound!**
- **PC-4 (Han, Lee 1988)**
  - correction of the PC-3 algorithm
  - based on computing supports of pairs (b,c) at arc (i,j)
- **PC-5 (Singh 1995)**
  - uses the ideas behind AC-6
  - only one support is kept and a new support is looked for when the current support is lost

Drawbacks of PC



- **memory consumption**
  - because PC eliminates pairs of values, we need to keep all the compatible pairs extensionally, e.g. using {0,1}-matrix
- **bad ratio strength/efficiency**
  - PC removes more (or same) inconsistencies than AC, but the strength/efficiency ratio is much worse than for AC
- **modifies the constraint network**
  - PC adds redundant arcs (constraints) and thus it changes connectivity of the constraint network
  - this complicates using heuristics derived from the structure of the constraint network (like density, graph width etc.)
- **PC is still not a complete technique**
  - A,B,C,D in {1,2,3}
    A≠B, A≠C, A≠D, B≠C, B≠D, C≠D
    is PC but has no solution



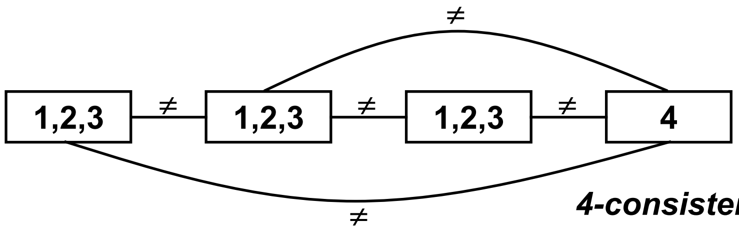
k-consistency

Is there a common formalism for AC and PC?

- AC: a value is extended to another variable
- PC: a pair of values is extended to another variable
- … we can continue

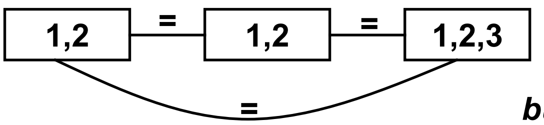
Definition:

CSP is k-consistent iff any consistent assignment of (k-1) different variables can be extended to a consistent assignment of one additional variable.



4-consistent graph

Strong k-consistency



3-consistent graph

but not 2-consistent graph!

Definition:

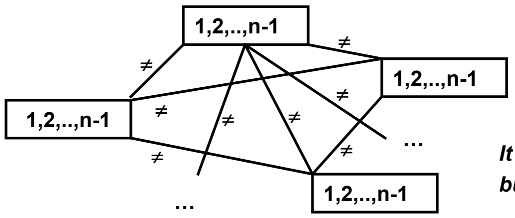
CSP is strongly k-consistent iff it is j-consistent for every $j \leq k$.

Visibly: **strong k-consistency $\Rightarrow$ k-consistency**
Moreover: **strong k-consistency $\Rightarrow$ j-consistency $\forall j \leq k$**
In general: $\neg$ **k-consistency $\Rightarrow$ strong k-consistency**

- NC = strong 1-consistency = 1-consistency
- AC = (strong) 2-consistency
- PC = (strong) 3-consistency
  - sometimes we call NC+AC+PC together **strong path consistency**

What k-consistency is enough?

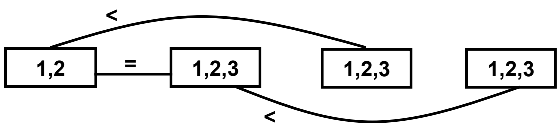
- Assume that the number of vertices is n. What level of consistency do we need to find out the solution?
- **Strong n-consistency for graphs with n vertices!**
  - n-consistency is not enough - see the previous example
  - strong k-consistency where k<n is not enough as well



graph with n vertices
domains 1..(n-1)

It is strongly k-consistent for k<n
but it has no solution

And what about this graph?

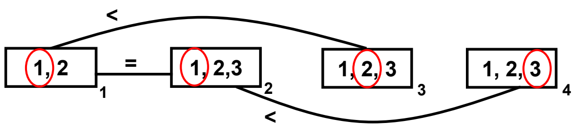


AC is enough!
Because this a tree..

Backtrack-free search

Definition:

CSP is solved using backtrack-free search if for some order of variables we can find a value for each variable compatible with the values of already assigned variables.



How to find out a sufficient consistency level for a given graph?

Some observations:

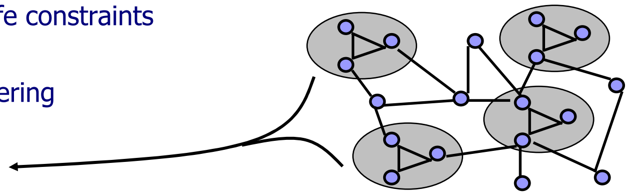
- variable must be compatible with all the "former" variables i.e., across the „backward" edges
- for k „backward" edges we need (k+1)-consistency
- let m be the maximum of backward edges for all the vertices, then strong (m+1)-consistency is enough
- the number of backward edges is different for different variable order
- of course, the order minimising m is looked for

Think globally

CSP describes the problem locally:

the constraints restrict small sets of variables

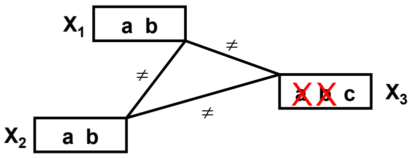
+ heterogeneous real-life constraints
- missing global view
  - ↳ weaker domain filtering



Global constraints

- global reasoning over a local sub-problem
- using semantic information to improve efficiency

Example:

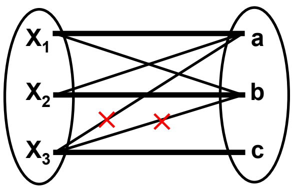


- local (arc) consistency deduces no pruning
- but some values can be removed

Inside all-different

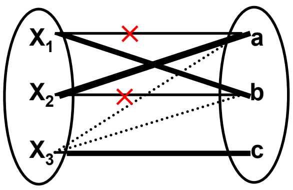
Régin (1994)

- a set of binary inequality constraints among all variables
  $X_1 \neq X_2, X_1 \neq X_3, \ldots, X_{k-1} \neq X_k$
- **all_different($\{X_1,\ldots, X_k\}$)** = $\{(d_1,\ldots, d_k) \mid \forall i \;\; d_i \in D_i \;\; \& \;\; \forall i \neq j \;\; d_i \neq d_j\}$
- better pruning based on **matching theory over bipartite graphs**



Initialisation:

1) compute **maximum matching**
2) remove all **edges** that do not belong to any maximum matching



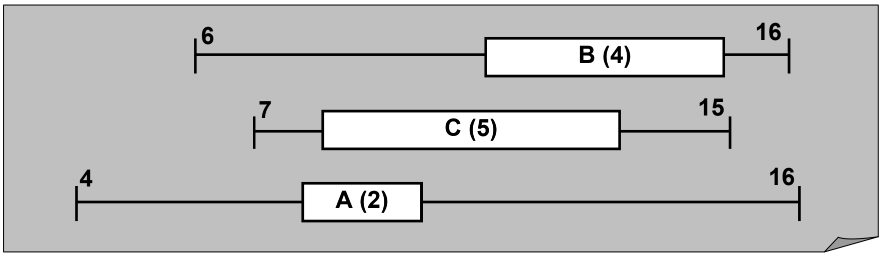
Propagation of deletions ($X_1 \neq a$):

1) remove discharged **edges**
2) compute **new maximum matching**
3) remove all **edges** that do not belong to any maximum matching

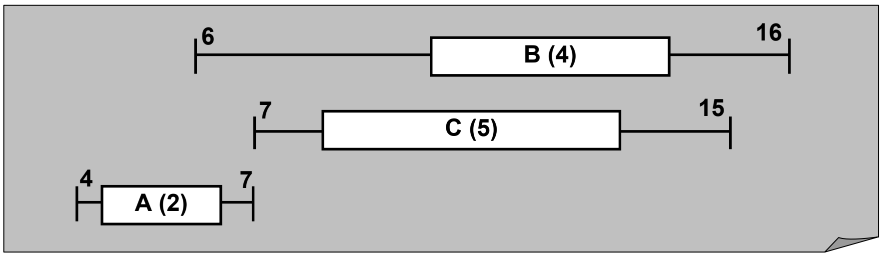
Baptiste, Le Pape (1996)

Inside edge finding

- a set of disjunctions ($A+P_A \leq B \vee B+P_B \leq A$) modeling an exclusive resource



- **min(start(Ω)) + p(Ω) + p(A) > max(end(Ω ∪ {A})) ⇒ A<<Ω**



Schulte (2002)

Design of filters

Users can often define code of the REVISE procedures for new constraints.
How to define new filters and integrate them into solvers?

1) decide about the event to evoke the filtering algorithm

- when the domain of involved variable is changed
  - whenever the domain changes (arc-consistency)
  - when minimum/maximum bound is changed (arc-B-consistency)
  - when the variable becomes singleton (constraint checking)
- different events (suspensions) for different variables

  Example:
  filtering for A<B is evoked after change of min(A) or max(B)

2) design the filtering algorithm for the constraint

- the result of filtering is the change of variables' domains
- more filtering procedures for a single constraint are allowed

Example: A<B

- min(A): B in min(A)+1..sup max(B): A in inf..max(B)-1

Design of filters
integration into solver

It is necessary to specify **when** the filtering algorithm is evoked and **what** global information is available to it.

- Some algorithms are **incremental** – describe how to react to a change in domain of a particular variable.
  - evoke the algorithm after the particular change happens
- Many algorithms for global constraints are proposed as **non-incremental** – filtering is run from scratch independently of the change.
  - evoke the algorithm after any change of constrained variables

Example (installation of filtering rule(s) for A<B)

- arc-B-consistency is identical to full arc-consistency!
- filtering can be realised incrementally

```
less_then(A,B):-
  fd_global(a2b(A,B),no_state,[min(A)]),
  fd_global(b2a(A,B),no_state,[max(B)]).
```

name of the filter with arguments

initial state (local data) of the filter

events when the filter is called

Design of filters
filtering algorithm

The filtering algorithm has access to current domains and it proposes how to restrict the domains (list of changes to domains).

- some solvers provide information about the change that invoked the filter

Example (definition of filtering rule(s) for A<B)

identification of the filter

states before/after running the filtering

list of proposed changes to domains

```
dispatch_global(a2b(A,B),S,S,Actions):-
  fd_min(A,MinA), fd_max(A,MaxA),
  fd_min(B,MinB),
  (MaxA<MinB ->
      Actions = [exit]
  ;   LowerBoundB is MinA+1,
      Actions = [B in LowerBoundB..sup]).

dispatch_global(b2a(A,B),S,S,Actions):-
  fd_max(A,MaxA),
  fd_min(B,MinB), fd_max(B,MaxB),
  (MaxA<MinB ->
      Actions = [exit]
  ;   UpperBoundA is MaxB-1,
      Actions = [A in inf..UpperBoundA]).
```

access to current domains of variables

filtering finished, don't call the filter anymore

Search

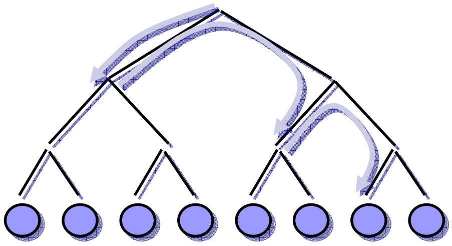
Consistency and Search

Consistency techniques are (usually) incomplete.

↳ **We need a search algorithm to resolve the rest!**

Labeling

- **depth-first search**
  - assign a value to the variable
  - propagate = make the problem locally consistent
  - backtrack upon failure



- **X in 1..3 ≈ X=1 ∨ X=2 ∨ X=3 (enumeration)**

In general, search algorithm resolves remaining disjunctions!

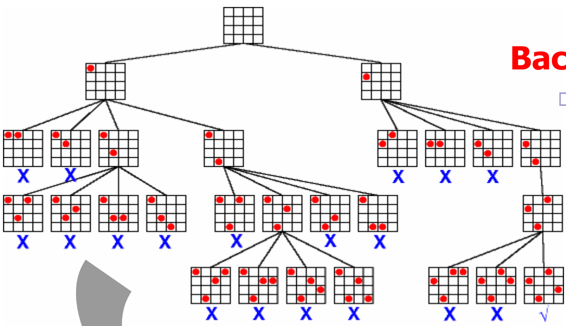
- **X=1 ∨ X≠1 (step labeling)**
- **X<3 ∨ X≥3 (bisection)**
- **X<Y ∨ X≥Y (variable ordering)**

Labeling skeleton

- Search is combined with filtering techniques that prune the search space.
- **Look-ahead technique (MAC)**

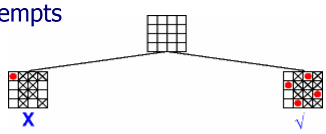
```
procedure labeling(V,D,C)
        if all variables from V are assigned then return V
        select not-yet assigned variable x from V
        for each value v from D_x do
                (TestOK,D') ← consistent(V,D,C∪{x=v})
                if TestOK=true then
                        R ← labeling(V,D',C)
                        if R ≠ fail then return R
        end for
        return fail
end labeling
```


4 queens problem — CP is not enumeration!



Backtracking is not very good
- 19 attempts

MAC combining search and arc consistency
- 2 attempts



Variable ordering

Variable ordering in labelling influence significantly efficiency of solvers (e.g. in a tree-structured CSP).

What variable ordering should be chosen in general?

FAIL FIRST principle

„select the variable whose instantiation will lead to a failure"

it is better to tackle failures earlier, they can be become even harder

- **prefer the variables with smaller domain** (dynamic order)
  - a smaller number of choices ~ lower probability of success
  - the dynamic order is appropriate only when new information appears during solving (e.g., in look ahead algorithms)

„solve the hard cases first, they may become even harder later"

- **prefer the most constrained variables**
  - it is more complicated to label such variables (it is possible to assume complexity of satisfaction of the constraints)
  - this heuristic is used when there is an equal size of the domains
- **prefer the variables with more constraints to past variables**
  - a static heuristic that is useful for look-back techniques

Value ordering

Order of values in labelling influence significantly efficiency (if we choose the right value each time, no backtrack is necessary).

What value ordering for the variable should be chosen in general?

SUCCEED FIRST principle

„prefer the values belonging to the solution"

- if no value is part of the solution then we have to check all values
- if there is a value from the solution then it is better to find it soon

Note: SUCCEED FIRST does not go against FAIL FIRST !

- **prefer the values with more suppors**
  - this information can be found in AC-4
- **prefer the value leading to less domain reduction**
  - this information can be computed using singleton consistency
- **prefer the value simplifying the problem**
  - solve approximation of the problem (e.g. a tree)

Generic heuristics are usually too complex for computation.

It is better to use problem-driven heuristics that propose the value!

Heuristics in search

- **Observation 1:**
  The **search space** for real-life problems is so **huge** that it cannot be fully explored.
- **Heuristics** - a guide of search
  - they recommend a value for assignment
  - quite often lead to a solution
- What to do upon a **failure of the heuristic**?
  - BT cares about the end of search (a bottom part of the search tree)
    so it rather repairs later assignments than the earliest ones
    thus BT assumes that the heuristic guides it well in the top part
- **Observation 2:**
  The **heuristics** are **less reliable in the earlier parts** of the search tree (as search proceeds, more information is available).
- **Observation 3:**
  The number of **heuristic violations** is usually **small**.

Discrepancies

So how to do search better than BT?

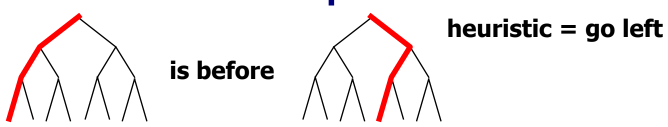
- BT is „heuristics blind"

Discrepancy = the heuristic is not followed

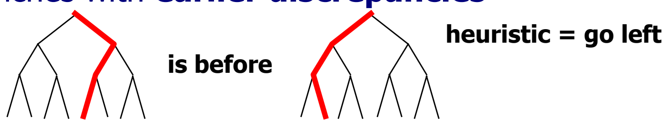
Basic principles of discrepancy search:

change the order of branches to be explored

- prefer branches with **less discrepancies**

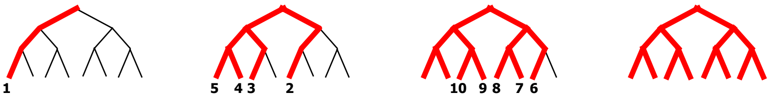


- prefer branches with **earlier discrepancies**

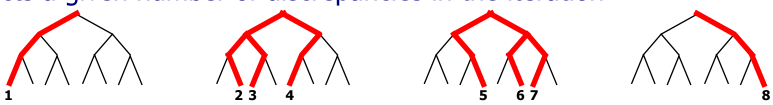


Discrepancy search

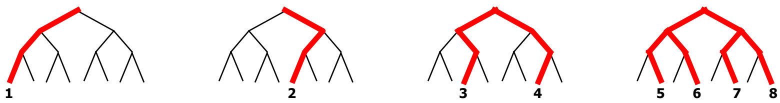
- **Limited Discrepancy Search (Harvey & Ginsberg, 1995)**
  - restricts a maximal number of discrepancies in the iteration



- **Improved LDS (Korf, 1996)**
  - restricts a given number of discrepancies in the iteration



- **Depth-bounded Discrepancy Search (Walsh, 1997)**
  - restricts discrepancies till a given depth in the iteration



- **…**

*** heuristic = go left**

Harvey, Ginsberg (1995)

Algorithm LDS

```
procedure LDS(Variables,Constraints)
      for D=0 to |Variables| do          % D is the number of allowed discrepancies
            R ← LDS-PROBE(Variables,{},Constraints,D)
            if R≠ fail then return R
      end for
      return fail
end LDS

procedure LDS-PROBE(Unlabeled,Labeled,Constraints,D)
      if Unlabelled = {} then return Labeled
      select X in Unlabelled
      Values_X ← D_X - {values inconsistent with Labeled using Constraints}
      if Values_X = {} then return fail
      else select HV in Values_X using heuristic
            if D>0 then                  % some discrepancy still allowed
               for each value V from Values_X -{HV} do
                  R ← LDS-PROBE(Unlabeled-{X}, Labeled∪{X/V}, Constraints, D-1)
                  if R≠ fail then return R
               end for
            return LDS-PROBE(Unlabeled-{X}, Labeled∪{X/HV}, Constraints, D)
      end if
end LDS-PROBE
```

Incomplete search

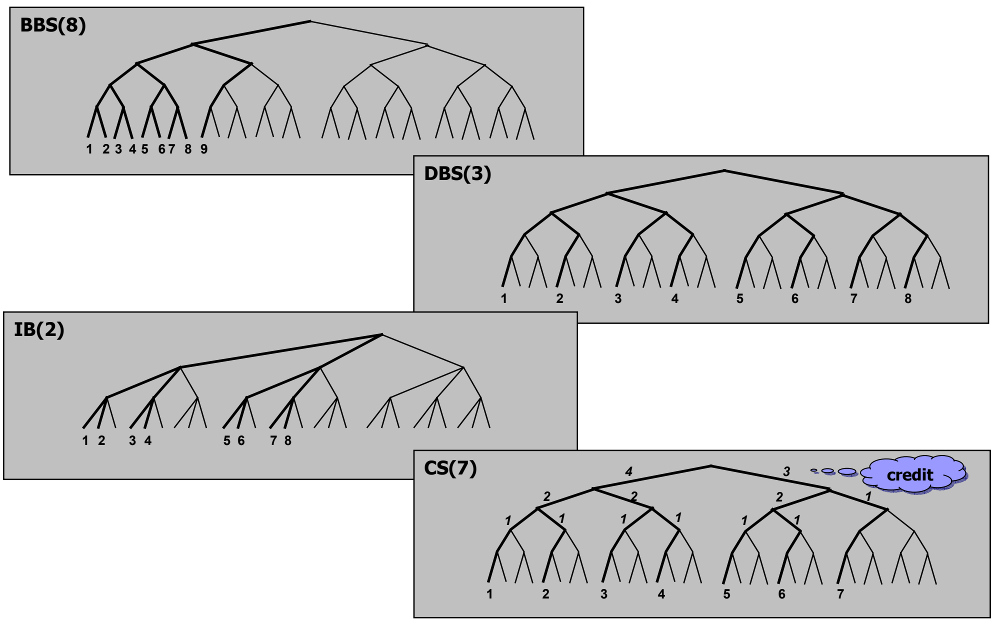
A **cutoff limit** to stop exploring a (sub-)tree

- some branches are skipped → incomplete search

When no solution found, **restart** with enlarged cutoff limit.

- **Bounded Backtrack Search (Harvey, 1995)**
  - restricted number of backtracks
- **Depth-bounded Backtrack Search (Cheadle et al., 2003)**
  - restricted depth where alternatives are explored
- **Iterative Broadening (Ginsberg and Harvey, 1990)**
  - restricted breadth in each node
  - still exponential!
- **Credit Search (Beldiceanu et al., 1997)**
  - limited credit for exploring alternatives
  - credit is split among the alternatives

Incomplete search



Constraint optimization

- **Constraint optimization problem** (COP) = CSP + objective function
- Objective function is encoded in a constraint v=obj(Xs) and the value of v is optimized

Algorithm Branch & Bound

```
procedure BB-Min(Variables, V, Constraints)
    Bound ← sup
    NewSolution ← fail
    repeat
        Solution ← NewSolution
        NewSolution ← Solve(Variables,Constraints ∪ {V<Bound})
        Bound ← value of V in NewSolution (if any)
    until NewSolution = fail
    return Solution
end BB-Min
```


Notes on B&B

- Efficiency dependents on:
  - good propagation of the objective function
  - a good first feasible solution (a good bound)
- The optimal solution can be found fast
  - but proof of optimality can be long
- A good enough solution is frequently OK.
  - BB can stop when it reaches a given limit for objective
  - or when the solution is close to optimum

Dichotomic version of B&B

```
repeat                                          bounds of the "objective" variable v
    Middle ← (UpperBound+LowerBound) / 2
    NewSolution ← Solve(Variables, Constraints ∪ {V ≤ Middle})
    if NewSolution=fail then
        LowerBound ← Middle+1
        Constraints ← Constraints ∪ {LowerBound ≤ V})
    else
        UpperBound ← Middle
until LowerBound = UpperBound
```

Modelling

Constraint modelling

How to describe the problem as a CSP that can be effectively solved?

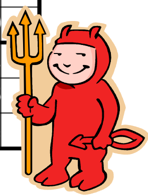
- **Seesaw problem**
  - starting simple with CLP
  - **symmetries** and **global constraints**
- **Assignment problem**
  - almost a real-life problem
  - **optimization** and **dual models**
- **Golomb rulers**
  - small but hard problem
  - **implied constraints**

Back to Sudoku

- Using all-different constraints applied to every row, column, and sub-grid, we can solve most Sudoku problems, but not all.
- There are some **Devil Sudoku Problems**, which we do not know how to solve using logical inference only.

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

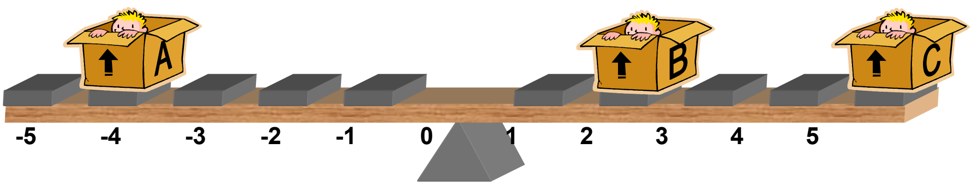
6	8	9	3	4	2	5	1	7
1	5	2	7	9	8	4	3	6
3	4	7	1	6	5	8	9	2
9	1	8	6	2	7	3	4	5
5	7	4	9	3	1	2	6	8
2	6	3	5	8	4	1	7	9
4	9	5	2	7	3	6	8	1
8	2	6	4	1	9	7	5	3
7	3	1	8	5	6	9	2	4



Seesaw problem

The problem:

Adam (36 kg), Boris (32 kg) and Cecil (16 kg) want to sit on a seesaw with the length 10 foots such that the minimal distances between them are more than 2 foots and the seesaw is balanced.



A CSP model:

- A,B,C in -5..5 — position
- 36*A+32*B+16*C = 0 — equilibrium state
- |A-B|>2, |A-C|>2, |B-C|>2 — minimal distances

Seesaw problem
implementation

```
:-use_module(library(clpfd)).

seesaw(Sol):-
    Sol = [A,B,C],

    domain(Sol,-5,5),
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    labeling([ff],Sol).
```

```
?- seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;
X = [4,-5,1] ? ;
X = [4,-4,-1] ? ;
X = [4,-2,-5] ? ;

no
```

Symmetry breaking

- important to reduce search space

```
:-use_module(library(clpfd)).

seesaw(Sol):-
    Sol = [A,B,C],

    domain(Sol,-5,5),
    A #< 0,
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    labeling([ff],Sol).
```

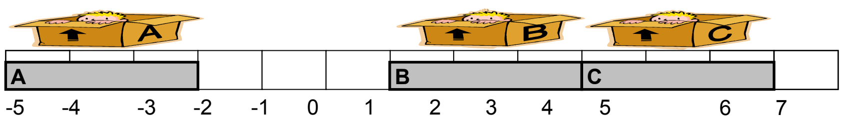
```
?- seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;

no
```


Seesaw problem
a different perspective

```
domain([A,B,C],-5,5),
A #< 0,
36*A+32*B+16*C #= 0,
abs(A-B)#>2,
abs(A-C)#>2,
abs(B-C)#>2
```

```
A in -4..0
B in -1..5
C in -5..5
```

- A set of similar constraints typically indicates a structured sub-problem that can be represented using a **global constraint.**



- We can use a global constraint describing **allocation of activities to exclusive resource.**

```
domain([A,B,C],-5,5),
A #< 0,
36*A+32*B+16*C #= 0,
serialized([A,B,C],[3,3,3],[])
```

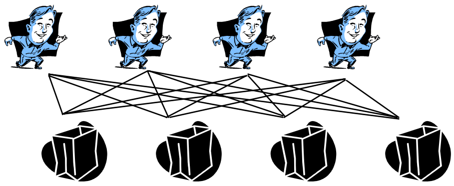
start times
durations
precedences

```
A in -4..0
B in -1..5
C in (-5..-3) \/ (-1..5)
```

Assignment problem

The problem:

There are 4 workers and 4 products and a table describing the efficiency of producing the product by a given worker. The task is assign workers to products (one to one) in such a way that the total efficiency is at least 19.



	P1	P2	P3	P4
W1	7	1	3	4
W2	8	2	5	1
W3	4	3	7	2
W4	3	1	6	3

A CSP model:

- W1,W2,W3,W4 in 1..4 — a product per worker
- all_different([W1,W2,W3,W4]) — different products
- $T_{1,W1}+T_{2,W2}+T_{3,W3}+T_{4,W4} \geq 19$ — total efficiency

Assignment problem
implementation

```
:-use_module(library(clpfd)).

assignment_p(Sol):-
    Sol = [W1,W2,W3,W4],

    domain(Sol,1,4),
    all_different(Sol),
    element(W1,[7,1,3,4],EW1),
    element(W2,[8,2,5,1],EW2),
    element(W3,[4,3,7,2],EW3),
    element(W4,[3,1,6,3],EW4),
    EW1+EW2+EW3+EW4 #>= 19,

    labeling([ff],Sol).
```

```
?- assignment_p(X).

X = [1,2,3,4] ? ;    19
X = [2,1,3,4] ? ;    19
X = [4,1,2,3] ? ;    21
X = [4,1,3,2] ? ;    20

no
```

Optimization using B&B

```
    EW1+EW2+EW3+EW4 #= E,

    maximize(labeling([ff],Sol),E).
```

```
?- assignment_p(X).

X = [4,1,2,3] ? ;

no
```

How does it work?

- **find** first **feasible instantiation** of variables
- **find better instantiation** of variables
- **repeat** until some instantiation of variables exists

Assignment problem
a dual model

Why do we **assign products to workers?**

Cannot we do it in an opposite way, that is, to **assign a worker to a product?**

Of course, we can **swap the role of values and variables!**

- This new model is called a **dual model.**

```
:-use_module(library(clpfd)).

assignment_dual(Products):-
    Products = [P1,P2,P3,P4],

    domain(Products,1,4),
    all_different(Products),
    element(P1,[7,8,4,3],EP1),
    element(P2,[1,2,3,1],EP2),
    element(P3,[3,5,7,6],EP3),
    element(P4,[4,1,2,3],EP4),
    EP1+EP2+EP3+EP4 #>= 19,

    labeling([ff],Products).
```

Number of choice points	
Primal model	15
Dual model	11

```
P1 in 1..2
P2 in 1..4
P3 in 2..4
P4 in 1..4
```

Which model is better?

- In this particular case, the dual model propagates earlier (thus it is assumed to be better).

Assignment problem
composing models

We can combine both primal and dual model in a single model to get better domain pruning.

```
:-use_module(library(clpfd)).

assignment_combined(Workers):-
    Workers= [W1,W2,W3,W4],
    domain(Workers,1,4),
    all_different(Workers),
    element(W1,[7,1,3,4],EW1),
    element(W2,[8,2,5,1],EW2),
    element(W3,[4,3,7,2],EW3),
    element(W4,[3,1,6,3],EW4),
    EW1+EW2+EW3+EW4 #>= 19,

    Products = [P1,P2,P3,P4],
    domain(Products,1,4),
    all_different(Products),
    element(P1,[7,8,4,3],EP1),
    element(P2,[1,2,3,1],EP2),
    element(P3,[3,5,7,6],EP3),
    element(P4,[4,1,2,3],EP4),
    EP1+EP2+EP3+EP4 #>= 19,

    assignment(Workers,Products),

    labeling([ff],Workers).
```

- **a primal model**

```
W1 in (1..2)\/{4}
W2 in 1..4
W3 in 2..4
W4 in 2..4
```

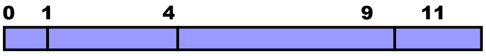
- **a dual model** (redundant)

```
P1 in 1..2
P2 in 1..4
P3 in 2..4
P4 in 1..4
```

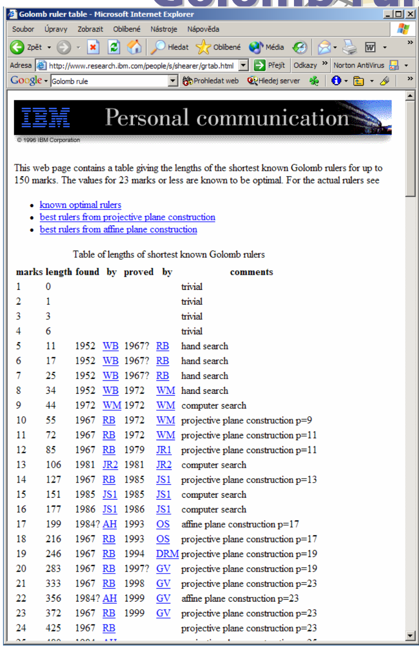
- **a channelling constraint**
- labelling one model is enough

Golomb ruler

- A **ruler with M marks** such that **distances** between any two marks are **different**.
- The **shortest ruler** is the optimal ruler.



- **Hard** for M≥16, no exact algorithm for M ≥ 24!
- Applied in **radioastronomy**.


Golomb ruler table - Microsoft Internet Explorer

Soubor Úpravy Zobrazit Oblíbené Nástroje Nápověda

Adresa http://www.research.ibm.com/people/s/shearer/grtab.html

IBM Personal communication

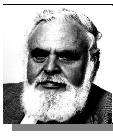
© 1996 IBM Corporation

This web page contains a table giving the lengths of the shortest known Golomb rulers for up to 150 marks. The values for 23 marks or less are known to be optimal. For the actual rulers see

- known optimal rulers
- best rulers from projective plane construction
- best rulers from affine plane construction

Table of lengths of shortest known Golomb rulers

marks	length	found	by	proved	by	comments
1	0					trivial
2	1					trivial
3	3					trivial
4	6					trivial
5	11	1952	WB	1967?	RB	hand search
6	17	1952	WB	1967?	RB	hand search
7	25	1952	WB	1967?	RB	hand search
8	34	1952	WB	1972	WM	hand search
9	44	1972	WM	1972	WM	computer search
10	55	1967	RB	1972	WM	projective plane construction p=9
11	72	1967	RB	1972	WM	projective plane construction p=11
12	85	1967	RB	1979	JR1	projective plane construction p=11
13	106	1981	JR2	1981	JR2	computer search
14	127	1967	RB	1985	JS1	projective plane construction p=13
15	151	1985	JS1	1985	JS1	computer search
16	177	1986	JS1	1986	JS1	computer search
17	199	1984?	AH	1993	OS	affine plane construction p=17
18	216	1967	RB	1993	OS	projective plane construction p=17
19	246	1967	RB	1994	DRM	projective plane construction p=19
20	283	1967	RB	1997?	GV	projective plane construction p=19
21	333	1967	RB	1998	GV	projective plane construction p=23
22	356	1984?	AH	1999	GV	affine plane construction p=23
23	372	1967	RB	1999	GV	projective plane construction p=23
24	425	1967	RB			projective plane construction p=23



Solomon W. Golomb
Professor
University of Southern California
http://csi.usc.edu/faculty/golomb.html

Golomb ruler
CSP model

A base model:

Variables $X_1, \ldots, X_M$ with the domain 0..M*M

$X_1 = 0$ *ruler start*

$X_1 < X_2 < \ldots < X_M$ *no permutations of variables*

$\forall i<j\ D_{i,j} = X_j - X_i$ *difference variables*

all_different($\{D_{1,2}, D_{1,3}, \ldots D_{1,M}, D_{2,3}, \ldots D_{M,M-1}\}$)

Model extensions:

$D_{1,2} < D_{M-1,M}$ *symmetry breaking*

better bounds (**implied constraints**) for $D_{i,j}$

$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$

so $\mathbf{D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2}$ *lower bound*

$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots D_{i-1,i} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$

$D_{i,j} = X_M - (D_{1,2} + \ldots D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$

so $\mathbf{D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2}$ *upper bound*

Golomb ruler
some results

- **What is the effect of different constraint models?**

size	base model	base model + symmetry	base model + symmetry + implied constraints
7	220	80	30
8	1 462	611	190
9	13 690	5 438	1 001
10	120 363	49 971	7 011
11	2 480 216	985 237	170 495

time in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM

- **What is the effect of different search strategies?**

size	fail first			leftmost first		
	enum	*step*	*bisect*	*enum*	*step*	*bisect*
7	40	60	40	30	30	30
8	390	370	350	220	190	200
9	2 664	2 384	2 113	1 182	1 001	921
10	20 870	17 545	14 982	8 782	7 011	6 430
11	1 004 515	906 323	779 851	209 251	170 495	159 559

time in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM

Modeling rules

- Determining the **efficiency of different models** is a difficult problem and one which **relies upon** an **understanding** of the underlying **constraint solver**.
- Usually, the **best model** will be the one in which information is **propagated first.**
- Some **rules of thumb** for constraint modelling:
  - **global constraints**
    - (+) strengthen propagation with good efficiency
  - **symmetry breaking**
    - (+) reduce search space
  - **implied constraints**
    - (+) strengthen propagation
    - (–) but add overhead

Conclusions

Constraint solvers

- It is not necessary to program all the presented techniques from scratch!
- Use existing constraint solvers (packages)!
  - provide **implementation of data structures** for modeling variables' domains and constraints
  - provide a basic **consistency framework**
  - provide **filtering algorithms** for many constraints (including global constraints)
  - provide basic **search strategies**
  - usually **extendible** (new filtering algorithms, new search strategies)

Some systems with **constraint satisfaction packages**:

- **Prolog**: CHIP, ECLiPSe, SICStus Prolog, Prolog IV, GNU Prolog, IF/Prolog
- **C/C++**: CHIP++, ILOG Solver, Gecode
- **Java**: JCK, JCL, Koalog
- **Oz**: Mozart

Resources

- **Books**
  - P. Van Hentenryck: **Constraint Satisfaction in Logic Programming**, MIT Press, 1989
  - E. Tsang: **Foundations of Constraint Satisfaction**, Academic Press, 1993
  - K. Marriott, P.J. Stuckey: **Programming with Constraints: An Introduction**, MIT Press, 1998
  - R. Dechter: **Constraint Processing**, Morgan Kaufmann, 2003
  - **Handbook of Constraint Programming**, Elsevier, 2006
- **Journals**
  - Constraints, An International Journal. Kluwer Academic Publishers
  - Constraint Programming Letters, free electronic journal
- **On-line materials**
  - **On-line Guide to Constraint Programming** (tutorial)
    http://kti.mff.cuni.cz/~bartak/constraints/
  - **Constraints Archive** (archive and links)
    http://4c.ucc.ie/web/archive/index.jsp
  - **Constraint Programming online** (community web)
    http://www.cp-online.org/

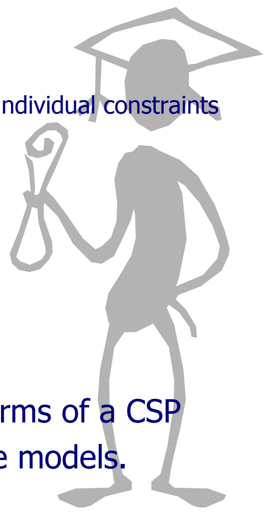
Summary

Constraints
- arbitrary relations over the problem variables
- express partial local information in a declarative way

Basic constraint satisfaction framework:
- **local consistency** connecting filtering algorithms for individual constraints
- **depth-first search** resolves remaining disjunctions
- **local search** can also be used

Problem solving using constraints:
- **declarative modeling** of problems as a CSP
- **dedicated algorithms** can be encoded in constraints
- special **search strategies**

It is easy to state combinatorial problems in terms of a CSP
… but it is more complicated to design solvable models.



References

Surveys

Constraint Programming - What is Behind?
R. Barták, in Proceedings of CPDC99 Workshop, pp. 7-16, Gliwice, 1999.

Theory and Practice of Constraint Propagation
R. Barták, in Proceedings of CPDC2001 Workshop, pp. 7-14, Gliwice, 2001.

Modelling Soft Constraints: A Survey
R. Barták, *Neural Network World,* Vol. 12, Number 5, pp. 421-431, 2002.

Incomplete Depth-First Search Techniques: A Short Survey
R. Barták, in Proceedings of CPDC2004 Workshop, pp. 7-14, Gliwice, 2004.

Constraint Logic Programming – A Survey
J. Jaffar & M.J. Maher, J. Logic Programming, 19/20:503-581, 1996.

Algorithms for Constraint Satisfaction Problems: A Survey
V. Kumar, AI Magazine 13(1): 32-44, 1992.

A Tutorial on Constraint Programming
B.M. Smith, TR 95.14, University of Leeds, 1995.

The Origins

The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory
A. Borning, in ACM Transactions on Programming Languages and Systems 3(4): 252-387, 1981.

Logic Programming: Further Developments
H. Gallaire, in: IEEE Symposium on Logic Programming, Boston, IEEE, 1985.

Constraint Logic Programming
J. Jaffar & J.L. Lassez, in Proc. The ACM Symposium on Principles of Programming Languages, ACM, 1987.

Networks of constraints fundamental properties and applications to picture processing
U. Montanary, in: Information Sciences 7: 95-132, 1974.

Sketchpad: a man-machine graphical communication system
I. Sutherland, in Proc. IFIP Spring Joint Computer Conference, 1963.

Understanding line drawings of scenes with shadows
D.L. Waltz, in Psychology of Computer Vision, McGraw-Hill, New York, 1975.

Consistency techniques

Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling
P. Baptiste, and C. Le Pape. Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group (PLANSIG), 1996.

Improving Domain Filtering using Restricted Path Consistency
P. Berlandier, in Proceedings of the IEEE CAIA-95, Los Angeles CA, 1995.

Arc-consistency and arc-consistency again
C. Bessiere, in Artificial Intelligence 65, pages 179-190, 1994.

Using constraint metaknowledge to reduce arc consistency computation
C. Bessiere, E.C. Freuder, and J.-R. Régin, in Artificial Intelligence 107, pages 125-148, 1999.

Refining the Basic Constraint Propagation Algorithm
Ch. Bessière and J.-Ch. Régin. In Proceedings of IJCAI-01, 309-315, (2001).

Some practicable filtering techniques for the constraint satisfaction problem
R. Debruyne and C. Bessiere, in Proceedings of the 15th IJCAI, pages 412-417, 1997.

Neighborhood inverse consistency preprocessing
E. Freuder and C. D. Elfe, in Proceedings of the AAAI National Conference, pages 202-208, 1996.

Comments on Mohr and Henderson's path consistency algorithm
C. Han and C. Lee, in Artificial Intelligence 36, pages 125-130, 1988.

Consistency in networks of relations
A.K. Mackworth, in Artificial Intelligence 8, pages 99-118, 1977.

The complexity of some polynomial network consistency algorithms for constraint satisfaction problems
A.K. Mackworth and E.C. Freuder, in Artificial Intelligence 25, pages 65-74, 1985.

Arc and path consistency revised
R. Mohr and T.C. Henderson, in Artificial Intelligence 28, pages 225-233, 1986.

Arc consistency for factorable relations
M. Perlin, in Artificial Intelligence 53, pages 329-342, 1992.

Singleton Consistencies
P. Prosser, K. Stergiou, T. Walsh, in Proc Principles and Practice of Constraint Programming (CP2000), pages 353-368, 2000.

A filtering algorithm for constraints of difference in CSPs
J.C. Régin, in AAAI-94, in Proceedings of the Twelfth National Conference on Artificial Intelligence, pages 362-367, 1994.

Path Consistency Revised
M. Singh, in Proceedings of the 7th IEEE International Converence on Tolls with Artificial Intelligence, pages 318-325, 1995.

A generic customizable framework for inverse local consistency
G. Verfaillie, D. Martinez, and C. Bessiere, in Proceedings of the AAAI National Conference, pages 169-174, 1999.

A generic arc-consistency algorithm and its specializations
P. Van Hentenryck, Y. Deville, and C.-M. Teng, in Artificial Intelligence 57, pages 291-321, 1992.

Making AC-3 an Optimal Algorithm.
Y. Zhang and R. Yap. In Proceedings of IJCAI-01, 316-321, (2001).

Search

Partial Search Strategy in CHIP
N. Beldiceanu, E. Bourreau, P. Chan, D. Rivreau, D. In 2nd International Conference on Metaheuristics (MIC 97), 1997.

ECLiPSe: An Introduction
Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen and Mark G. Wallace. IC-Parc, Imperial College London, Technical Report IC-Parc-03-1, 2003.

Backtracking algorithms for constraint satisfaction problems; a survey
R. Dechter, D. Frost, in Constraints, International Journal, 1998.

Performance Measurement and Analysis of Certain Search Algorithms
Gaschnig, J., CMU-CS-79-124, Carnegie-Mellon University, 1979.

Iterative Broadening
M.L. Ginsberg, W.D. Harvey, In AAAI Proceedings, 1990.

Increasing tree search efficiency for constraint satisfaction problems
Haralick, R.M., Elliot, G.L., in: Artificial Intelligence 14:263-314, 1980.

Limited Discrepancy Search
W.D. Harvey and M.L. Ginsberg, in Proceedings of IJCAI95, pages 607-613, 1995.

Nonsystematic backtracking search
W. D. Harvey. PhD thesis, Stanford University, 1995.

Improved Limited Discrepancy Search
Richard E. Korf. In Proceedings of National Conference on Artificial Intelligence (AAAI-96). AAAI Press, pp. 286-291, 1996.

Interleaved Depth-First Search
Pedro Meseguer. In Proceedings of 15th International Joint Conference on Artificial Intelligence, pp. 1382-1387, 1997.

Interleaved and Discrepancy Based Search
Pedro Meseguer and Toby Walsh. In Proceedings of 13th European Conference on Artificial Intelligence, Wiley, pp. 239-243, 1998.

Depth-bounded Discrepancy Search
Toby Walsh. In Proceedings of 15th International Joint Conference on Artificial Intelligence, pp. 1388-1393, 1997.