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
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.

A bit of history

1979: first published in New York under the name „Number Place“
1986: became popular in Japan
Sudoku – from Japanese "Suji wa dokushin ni kagiru" “the numbers must be single” or "the numbers must occur once"
2005: became popular in the western world
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,\ldots,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.
- **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
**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 ≤ start ≤ 9)
- **a finite set of constraints**
  - constraint is an arbitrary relation over the set of variables, like startA + 15 ≤ 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
Floating point variables

Integer variables

Linear constraints

Logical constraints

• various domains
• arbitrary constraints
• heterogeneous problems

Constraint Programming

Linear Programming

Mixed Integer Programming

Discrete Mathematics

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:**

\[
\begin{array}{c|c|c}
A & 3..7 & \leq \text{B} \\
\hline
\text{no arc is consistent}
\end{array}
\quad
\begin{array}{c|c|c}
A & 3..4 & \leq \text{B} \\
\hline
\text{(A,B) is consistent}
\end{array}
\quad
\begin{array}{c|c|c}
A & 3..4 & \leq \text{B} \\
\hline
\text{(A,B) and (B,A) are consistent}
\end{array}
\]

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 constraints on \(V_i, V_j\)).

**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
```

The procedure also reports the deletion of some value.
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

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
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).
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 $i \leftarrow 1$ to $n$ do
  
  $Q \leftarrow \{(i,j) | (i,j) \in \text{arcs}(G), j<i\}$ % arcs for the base revision
  
  $Q' \leftarrow \{(j,i) | (i,j) \in \text{arcs}(G), j<i\}$ % 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
        
        $Q' \leftarrow Q' \cup \{(p,k) | (p,k) \in \text{arcs}(G), p \leq i, p \neq k\}$
        
      end if
      
    end while
  
  $Q \leftarrow Q'$
  
  $Q' \leftarrow$ empty
  
  end while
  
  end for

end AC-2

**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)

- **procedure** AC-3(G)
  
  $Q \leftarrow \{(i,j) | (i,j) \in \text{arcs}(G), i<j\}$ % queue of arcs for revision
  
  while $Q$ non empty do
    
    select and delete $(k,m)$ from $Q$
    
    if REVISE($(k,m)$) then
      
      $Q \leftarrow Q \cup \{(i,k) | (i,k) \in \text{arcs}(G), i \neq k, i \neq m\}$
      
    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 $O(ed^3)$).
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 V₂,V₁ is revised, the value a is removed from domain of V₂.
2. Now the domain of V₃ should be explored to find out if any value a,b,c,d loses the support in V₂.

Observation:
The values a,b,c need not be checked again because they still have a support in V₂ different from a.

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

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 ← {}, S ← {} % emptying the data structures
    for each arc (Vi,Vj) in arcs(G) do
        for each a in Di do
            total ← 0
            for each b in Dj do
                if (a,b) is consistent according to the constraint C_{ij} then
                    total ← total + 1
                    S_{j,b} ← S_{j,b} ∪ {<i,a>}
                end if
            end for
            counter[(i,j),a] ← total
            if counter[(i,j),a] = 0 then
                delete a from Di
                Q ← Q ∪ {<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 \( V_i \)
Using support sets

Situation:
we have just processed the arc (i,j) in INITIALIZE

Using the support sets:
1. Let $b_3$ is deleted from the domain of $j$ (for some reason).
2. Look at $S_{j,b_3}$ to find out the values that were supported by $b_3$
   (i.e. $<i,a_2>,<i,a_3>$).
3. Decrease the counter for these values (i.e. tell them that they lost one
   support).
4. If any counter becomes zero ($a_3$) then delete the value and repeat the
   procedure with the respective value (i.e., go to 1).

Algorithm AC-4

The algorithm AC-4 has optimal worst case time complexity $O(ed^2)$!

Unfortunately the average time complexity is not so good
... plus there is a big memory consumption!

Mohr, Henderson (1986)
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, Régis, 1999)
  - improves AC-6 by exploiting symmetry of the constraint
- **AC-2000** (Bessière & Régis, 2001)
  - an adaptive version of AC-3 that either looks for a support or propagates deletions
- **AC-2001** (Bessière & Régis, 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)

...
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:

```
X
≠
Y
≠
Z

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 \(x \in D_0\) and \(y \in D_m\) 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:
- \(X \in \{1,2\}, Y \in \{1,2\}, Z \in \{1,2\}, \ X \neq Z, X \neq Y, Y \neq Z\)
- 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 (\(A<B, B<C \implies A+1<C\))
- a unary constraint = a variable’s domain

Path revision

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

Example:
- \(A,B,C \in \{1,2,3\}, B>1\)
- \(A<C, A=B, B>C-2\)
How to make the path \((i,k,j)\) consistent?

\[ R_{ij} \leftarrow R_{ij} \& (R_{ik} \ast R_{kk} \ast R_{kj}) \]

How to make a CSP path consistent?

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

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

Mackworth (1977)

If we use

\[ Y^k_{ii} ← Y^{k-1}_{ii} \& (Y^{k-1}_{ik} \ast Y^{k-1}_{kk} \ast Y^{k-1}_{ki}) \]

then we get AC-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 \((Y_j = Y^j_j)\)
- the grand problem
  - after domain change all the paths are re-revised
    but it is enough to revise just the influenced paths

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

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

Because $Y_j = Y_i$, it is enough to revise only the paths $(i,k,j)$ where $i \leq j$.
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)

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)

$S_a = \{(i,j,m) | i \leq m \leq n & m \neq j\} \cup \{(m,i,j) | 1 \leq m \leq j & m \neq i\} \cup \{(j,i,m) | j < m \leq n\} \cup \{(m,j,i) | 1 \leq m < i\}$

$|S_a| = 2n-2$

**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

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

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

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 if
    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
**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.

**Strong k-consistency**

**Definition:**

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

Visibly: \[ \text{strong } k\text{-consistency } \Rightarrow \text{k-consistency} \]

Moreover: \[ \text{strong } k\text{-consistency } \Rightarrow \text{j-consistency } \forall j \leq k \]

In general: \[ \neg \text{k-consistency } \Rightarrow \text{strong } k\text{-consistency} \]

- **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

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

And what about this graph?

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
  \(\Rightarrow\) weaker domain filtering

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

Example:

\[
\begin{align*}
X_1 &\quad a \quad b \\
X_2 &\quad a \quad b \\
X_3 &\quad c
\end{align*}
\]

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

Régin (1994)

Inside all-different

- a set of binary inequality constraints among all variables
  \(X_1 \neq X_2, X_1 \neq X_3, \ldots, X_{k-1} \neq X_k\)
- \(\text{all\_different}(\{X_1, \ldots, X_k\}) = \{(d_1, \ldots, d_k) \mid \forall i \ d_i \in D_i \text{ and } \forall i \neq j \ d_i \neq d_j\}\)
- 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
A set of disjunctions \((A+PA \leq B \lor B+PB \leq A)\) modeling an exclusive resource

\[
\min(\text{start}(\Omega)) + p(\Omega) + p(A) > \max(\text{end}(\Omega \cup \{A\})) \Rightarrow A \ll \Omega
\]

**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 \(\min(A)+1\ldots\sup\)
     - \(\max(B)\): \(A\) in \(\inf\ldots\max(B)-1\)
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)

- 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)]).
```

**Example**

(Definition of filtering rule(s) for A<B)

```
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]).
```
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)
Search is combined with filtering techniques that prune the search space.

**Look-ahead technique (MAC)**

```plaintext
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 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 supports**
  - 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!**
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.

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*

---

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

- **Constraint optimization problem (COP)**
  \[ \text{CSP + objective function} \]

- Objective function is encoded in a constraint \( v = \text{obj}(Xs) \) 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
  Middle ← (UpperBound+LowerBound) / 2
  NewSolution ← Solve(Variables, Constraints ∪ \{V ≤ Middle\})
  if NewSolution=false then
    LowerBound ← Middle+1
    Constraints ← Constraints ∪ \{LowerBound ≤ V\})
  else
    UpperBound ← Middle
  until LowerBound = UpperBound
```
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
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 6 3 9
```

---

**Seesaw problem**

The problem:

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

A CSP model:

- $A, B, C$ in $-5..5$ position
- $36A + 32B + 16C = 0$ equilibrium state
- $|A-B|>2$, $|A-C|>2$, $|B-C|>2$ minimal distances
Seesaw problem
implementation

?- 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

?- seesaw(X).
X = [-4,2,5] ;
X = [-4,4,1] ;
X = [-4,5,-1] ;
X = [4,-5,1] ;
no


Symmetry breaking
- important to reduce search space

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.
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.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>W2</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>W3</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>W4</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

A CSP model:

- $W_1, W_2, W_3, W_4$ in $1..4$ implies a product per worker
- all_different([W1,W2,W3,W4]) implies different products
- $T_1, W_1 + T_2, W_2 + T_3, W_3 + T_4, W_4 \geq 19$ implies total efficiency

Optimization using B&B

find first feasible instantiation of variables
find better instantiation of variables
repeat until some instantiation of variables exists
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.

Number of choice points

- Primal model: 15
- Dual model: 11

Which model is better?

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

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

- a primal model
  - W1 in \(1 \ldots 2\) \(\backslash\) \{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
A ruler with M marks such that distances between any two marks are different.

The shortest ruler is the optimal ruler.

Hard for \( M \geq 16 \), no exact algorithm for \( M \geq 24 \! \). Applied in radioastronomy.

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

---

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

\( \forall i < j \, D_{ij} = X_j - X_i \)  

difference variables

\( \text{all}\_\text{different}(\{D_{1,2}, D_{1,3}, \ldots D_{1,M}, D_{2,3}, \ldots D_{M,M-1}\}) \)

Model extensions:

\( D_{1,2} < D_{M-1,M} \)  

symmetry breaking

better bounds (implied constraints) for \( D_{ij} \)

\( D_{ij} = D_{i+1} + D_{i+1+j+2} + \ldots + D_{j+1,j} \)

so \( D_{ij} \geq \frac{\sum_{j=1}^{i} D_{ij}}{2} = \frac{(j-i)(j-i+1)}{2} \)  

lower bound

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

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

so \( D_{ij} \leq X_M - \frac{(M-1-j+i)(M-j+i)}{2} \)  

upper bound
What is the effect of different constraint models?

<table>
<thead>
<tr>
<th>size</th>
<th>base model</th>
<th>base model + symmetry</th>
<th>base model + symmetry + implied constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>220</td>
<td>80</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>1462</td>
<td>611</td>
<td>190</td>
</tr>
<tr>
<td>9</td>
<td>13690</td>
<td>5438</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>120363</td>
<td>49971</td>
<td>7011</td>
</tr>
<tr>
<td>11</td>
<td>2480216</td>
<td>985237</td>
<td>170495</td>
</tr>
</tbody>
</table>

What is the effect of different search strategies?

<table>
<thead>
<tr>
<th>size</th>
<th>fail first</th>
<th>leftmost first</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>enum</td>
<td>step</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>390</td>
<td>370</td>
</tr>
<tr>
<td>9</td>
<td>2664</td>
<td>2384</td>
</tr>
<tr>
<td>10</td>
<td>20870</td>
<td>17545</td>
</tr>
<tr>
<td>11</td>
<td>1004515</td>
<td>906323</td>
</tr>
</tbody>
</table>

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**

- **Journals**
  - Constraint Programming Letters, free electronic journal

- **On-line materials**
  - On-line Guide to Constraint Programming (tutorial)
  - Constraints Archive (archive and links)
    - [http://4c.ucc.ie/web/archive/index.jsp](http://4c.ucc.ie/web/archive/index.jsp)
  - Constraint Programming online (community web)
    - [http://www.cp-online.org/](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?

Theory and Practice of Constraint Propagation

Modelling Soft Constraints: A Survey

Incomplete Depth-First Search Techniques: A Short Survey

Constraint Logic Programming – A Survey

Algorithms for Constraint Satisfaction Problems: A Survey

A Tutorial on Constraint Programming

The Origins

The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory

Logic Programming: Further Developments

Constraint Logic Programming

Networks of constraints fundamental properties and applications to picture processing

Sketchpad: a man-machine graphical communication system

Understanding line drawings of scenes with shadows

Consistency techniques

Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling

Improving Domain Filtering using Restricted Path Consistency

Arc-consistency and arc-consistency again

Using constraint metaknowledge to reduce arc consistency computation

Refining the Basic Constraint Propagation Algorithm

Some practicable filtering techniques for the constraint satisfaction problem
Neighborhood inverse consistency preprocessing

Comments on Mohr and Henderson's path consistency algorithm

Consistency in networks of relations

The complexity of some polynomial network consistency algorithms for constraint satisfaction problems

Arc and path consistency revised

Arc consistency for factorable relations

Singleton Consistencies

A filtering algorithm for constraints of difference in CSPs

Path Consistency Revised

A generic customizable framework for inverse local consistency

A generic arc-consistency algorithm and its specializations

Making AC-3 an Optimal Algorithm

Search

Partial Search Strategy in CHIP

ECLiPSe: An Introduction

Backtracking algorithms for constraint satisfaction problems; a survey

Performance Measurement and Analysis of Certain Search Algorithms

Iterative Broadening

Increasing tree search efficiency for constraint satisfaction problems

Limited Discrepancy Search

Nonsystematic backtracking search

Improved Limited Discrepancy Search

Interleaved Depth-First Search

Interleaved and Discrepancy Based Search

Depth-bounded Discrepancy Search