“Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.“

Eugene C. Freuder, Constraints, April 1997
Holly Grail of Programming

> Computer, solve the SEND, MORE, MONEY problem!
> Here you are. The solution is
> \[9,5,6,7]+[1,0,8,5]=[1,0,6,5,2]\]

a Star Trek view

> Sol=[S,E,N,D,M,O,R,Y],
domain([E,N,D,O,R,Y],0,9), domain([S,M],1,9),
1000*S + 100*E + 10*N + D +
1000*M + 100*O + 10*R + E #=
10000*M + 1000*O + 100*N + 10*E + Y,
all_differe(Sol),
labeling([ff],Sol).
> Sol = [9,5,6,7,1,0,8,2]

today reality

Tutorial outline

- **Introduction**
  - history, applications, a constraint satisfaction problem

- **Depth-first search**
  - backtracking, backjumping, backmarking
  - incomplete and discrepancy search

- **Consistency**
  - node, arc, and path consistencies
  - \(k\)-consistency and global constraints

- **Combining search and consistency**
  - look-back and look-ahead schemes
  - variable and value ordering

- **Conclusions**
  - constraint solvers, resources
Introduction

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

All types of hard combinatorial problems:

- molecular biology
  - DNA sequencing
  - determining protein structures
- interactive graphic
  - web layout
- network configuration
- assignment problems
  - personal assignment
  - stand allocation
- timetabling
- scheduling
- planning

Constraint technology

based on declarative problem description via:

- variables with domains (sets of possible values)
  - e.g. start of activity with time windows
- constraints restricting combinations of variables
  - e.g. endA < startB

constraint optimization via objective function
  - e.g. minimize makespan

Why to use constraint technology?

- understandable
- open and extendible
- proof of concept
Constraint satisfaction problem consists of:
- a finite set of variables
- domains - a finite set of values for each variable
- a finite set of constraints
  - constraint is an arbitrary relation over the set of variables
  - can be defined extensionally (a set of compatible tuples) or intentionally (formula)

A solution to a CSP is a complete consistent assignment of variables.
- complete = a value is assigned to every variable
- consistent = all the constraints are satisfied

**Binary constraint satisfaction**
- only binary constraints
- any CSP is convertible to a binary CSP
  - dual encoding (Stergiou & Walsh, 1990)
    swapping the role of variables and constraints

**Boolean constraint satisfaction**
- on domains {0, 1}
- any CSP is convertible to a Boolean CSP
  - SAT encoding
    - Boolean variable indicates whether a given value is assigned to the variable

variables $x_1, \ldots, x_6$ with domain $\{0, 1\}$
- $c_1: x_1 + x_2 + x_6 = 1$
- $c_2: x_1 - x_3 + x_4 = 1$
- $c_3: x_4 + x_5 - x_6 > 0$
- $c_4: x_2 + x_5 - x_6 = 0$

Possible assignments:
- $(0, 0, 1), (0, 1, 0), (1, 0, 0)$
- $(0, 0, 0), (0, 1, 1), (1, 0, 1)$
- $(0, 0, 1), (1, 0, 0), (1, 1, 1)$
- $(0, 1, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1)$
Two or more?

- **Binary constraint satisfaction**
  - only binary constraints
  - any CSP is convertible to a binary CSP
    - **dual encoding** (Stergiou & Walsh, 1990)
      - swapping the role of variables and constraints

- **Boolean constraint satisfaction**
  - only Boolean (two valued) domains
  - any CSP is convertible to a Boolean CSP
    - **SAT encoding**
      - Boolean variable indicates whether a given value is assigned to the variable

**Depth-first search**
Basic Idea

- We are looking for a **complete consistent assignment**.
  - start with a **consistent assignment** (for example, empty one)
  - **extend the assignment** towards a complete assignment

- **Depth-first search** is a technique of searching solution by extending a partial consistent assignment towards a complete consistent assignment.
  - **assign values** gradually to variables
  - after each assignment **test consistency** of the constraints over the assigned variables
  - and **backtrack** upon failure

- **Backtracking** is probably the most widely used complete systematic search algorithm.
  - complete = guarantees finding a solution or proving its non-existence

---

Chronological backtracking

**A recursive definition**

```plaintext
procedure BT(X:variables, V:assignment, C:constraints)
  if X={} then return V
  x ← select a not-yet assigned variable from X
  for each value h from the domain of x do
    if consistent(V+{x|h}, C) then
      R ← BT(X-{x}, V+{x|h}, C)
      if R≠fail then return R
  end for
  return fail
end BT
```

call BT(X, {}, C)

Consistency procedure checks satisfaction of constraints whose variables are already assigned.
Weaknesses of backtracking

- **thrashing**
  - throws away the reason of the conflict
  - **Example:** A,B,C,D,E:: 1..10, A>E
    - BT tries all the assignments for B,C,D before finding that A≠1
  - **Solution:** backjumping (jump to the source of the failure)

- **redundant work**
  - unnecessary constraint checks are repeated
  - **Example:** A,B,C,D,E:: 1..10, B+8<D, C=5*E
    - when labelling C,E the values 1,...,9 are repeatedly checked for D
  - **Solution:** backmarking, backchecking (remember (no-)good assignments)

- **late detection of the conflict**
  - constraint violation is discovered only when the values are known
  - **Example:** A,B,C,D,E:: 1..10, A=3*E
    - the fact that A>2 is discovered when labelling E
  - **Solution:** forward checking (forward check of constraints)

Backjumping

Backjumping is a technique for removing thrashing from backtracking. **How?**

1. identify the source of the conflict (impossibility to assign a value)
2. jump to the past variable in conflict

- The same forward run like in backtracking, only the back-jump can be longer, and hence irrelevant assignments are skipped!

**How to find a jump position? What is the source of the conflict?**

- select the constraints containing just the currently assigned variable and the past variables
- select the closest variable participating in the selected constraints

**Graph-directed backjumping**

**Enhancement:** use only the violated constraints

© conflict-directed backjumping
Conflict-directed backjumping

in practice

N-queens problem

Queens in rows are allocated to columns.

6th queen cannot be allocated!

1. Write a number of conflicting queens to each position.
2. Select the farthest conflicting queen for each position.
3. Select the closest conflicting queen among positions.

Note:
Graph-directed backjumping has no effect here (due to the complete graph)!

How to find out the conflicting variable?

Situation:
- Assume that the variable no. 7 is being assigned (values are 0, 1)
- The symbol • marks the variables participating the violated constraints (two constraints for each value)

Neither 0 nor 1 can be assigned to the seventh variable!

1. Find the closest variable in each violated constraint (○).
2. Select the farthest variable from the above chosen variables for each value (×).
3. Choose the closest variable from the conflicting variables selected for each value and jump to it.
In addition to the test of satisfaction of the constraints, the closest conflicting level is computed.

**Consistency check (BJ)**

```plaintext
procedure consistent(Labeled, Constraints, Level)
    J ← Level % the level to which we will jump
    NoConflict ← true % indicator of a conflict
    for each C in Constraints do
        if all variables from C are Labeled then
            if C is not satisfied by Labeled then
                NoConflict ← false
                J ← min {J, max{L | X in C & X/V/L in Labeled & L<Level}}
            end if
        end if
    end for
    if NoConflict then return true
    else return fail(J)
end consistent
```

**Algorithm backjumping**

```plaintext
procedure BJ(Unlabeled, Labeled, Constraints, PreviousLevel)
    if Unlabelled = {} then return Labeled
    pick first X from Unlabelled
    Level ← PreviousLevel+1
    Jump ← 0
    for each value V from DX do
        C ← consistent({X/V/Level} ∪ Labeled, Constraints, Level)
        if C = fail() then
            Jump ← max {Jump, J}
        else
            Jump ← PreviousLevel
            R ← BJ(Unlabeled-{X}, {X/V/Level} ∪ Labeled, Constraints, Level)
            if R ≠ fail( Level) then return R % success or backjump
        end if
    end for
    return fail(Jump) % jump to the conflicting variable
end BJ
```

Gaschnig (1979)
Weakness of Backjumping

When jumping back the in-between assignment is lost!

**Example:**
- colour the graph in such a way that the connected vertices have different colours

<table>
<thead>
<tr>
<th>node</th>
<th>vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1 2</td>
</tr>
<tr>
<td>D</td>
<td>1 2 3</td>
</tr>
<tr>
<td>E</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

During the second attempt to label C superfluous work is done - it is enough to leave there the original value 2, the change of B does not influence C.

Dynamic Backtracking

The same graph (A,B,C,D,E), the same colours (1,2,3) but a different approach.

**Backjumping**
- remember the source of the conflict
- carry the source of the conflict
- change the order of variables

= **Dynamic Backtracking**

The vertex C (and the possible sub-graph connected to C) is not re-coloured.
**Algorithm dynamic BT**

```plaintext
procedure DB(Variables, Constraints)
    Labelled ← {}; Unlabelled ← Variables
    while Unlabelled ≠ {} do
        select X in Unlabelled
        ValuesX ← DX - {values inconsistent with Labelled using Constraints}
        if ValuesX = {} then
            let E be an explanation of the conflict (set of conflicting variables)
            if E = {} then failure
            else let Y be the most recent variable in E
                unassign Y (from Labelled) with eliminating explanation E-{Y}
                remove all the explanations involving Y
            end if
        else
            select V in ValuesX
            Unlabelled ← Unlabelled - {X}
            Labelled ← Labelled \cup \{X/V\}
        end if
    end while
    return Labelled
end DB
```

---

**Redundant work**

**What is redundant work in chronological backtracking?**

- repeated computation whose result has already been obtained

**Example:**

A,B,C,D :: 1..10, A+8<C, B=5*D

Redundant computations:

it is not necessary to repeat them because the change of B does not influence C.
Backmarking

Removes redundant constraint checks by memorizing negative and positive tests:

- **Mark**(X,V) is the farthest (instantiated) variable in conflict with the assignment X=V
- **BackTo**(X) is the farthest variable to which we backtracked since the last attempt to instantiate X

Now, some constraint checks can be omitted:

### N-queens problem

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="queen" alt="1" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td><img src="queen" alt="2" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td><img src="queen" alt="3" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td><img src="queen" alt="4" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td><img src="queen" alt="5" /></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:
Backmarking can be combined with backjumping (for free).
**Consistency check (BM)**

Only the constraints where any value is changed are re-checked, and the farthest conflicting level is computed.

```plaintext
procedure consistent(X/V, Labeled, Constraints, Level)
    for each Y/VY/LY in Labeled such that LY ≥ BackTo(X) do
        % only possible changed variables Y are explored
        % in the increasing order of LY (first the oldest one)
        if X/V is not compatible with Y/VY using Constraints then
            Mark(X,V) ← LY
            return fail
        end if
    end for
    Mark(X,V) ← Level-1
    return true
end consistent
```

**Algorithm backmarking**

```plaintext
procedure BM(Unlabeled, Labeled, Constraints, Level)
    if Unlabelled = {} then return Labeled
    pick first X from Unlabelled % fix order of variables
    for each value V from DX do
        if Mark(X,V) ≥ BackTo(X) then % re-check the value
            if consistent(X/V, Labeled, Constraints, Level) then
                R ← BM(Unlabeled-{X}, Labeled ∪ {X/V/Level}, Constraints, Level+1)
                if R ≠ fail then return R % solution found
            end if
        end if
    end for
    BackTo(X) ← Level-1 % jump will be to the previous variable
    for each Y in Unlabelled do % tell everyone about the jump
        BackTo(Y) ← min {Level-1, BackTo(Y)}
    end for
    return fail % return to the previous variable
end BM
```
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

---

**BBS(8)**

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

**DBS(3)**

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

**IB(2)**

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

**CS(7)**

```
1 2 3 4 5 6 7
```
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

**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
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
So far we used constraints in a **passive way** (as a test) ... 
...in the best case we analysed the reason of the conflict.

Cannot we use the constraints in a more active way?

**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 inconsistent values from the variables’ domains in the constraint network?**

---

**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 CSP 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:
- \(3..7\) \begin{array}{c|c|c|c|c|c} A & 3 & B & 4 & 5 \end{array}\)
- \(1..5\) \begin{array}{c|c|c|c|c|c} A & B \end{array}\)
- \(A < B\)
- \((A,B)\) 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 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

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X in ([1,..,6])</td>
<td>X in ([1,..,5])</td>
<td>X in ([4,5])</td>
</tr>
<tr>
<td>Y in ([1,..,6])</td>
<td>Y in ([2,..,6])</td>
<td>Y in ([5,6])</td>
</tr>
<tr>
<td>Z in ([1,..,6])</td>
<td>Z in ([1,..,6])</td>
<td>Z in ([1,2])</td>
</tr>
</tbody>
</table>

Make all the constraints consistent until any domain is changed.

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 ← 1 to n do
    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' ∪ {(i,k) | (i,k) ∈ arcs(G), p≤i, p≠m }
            end if
        end while
        Q ← Q' ∪ {(i,k) | (i,k) ∈ arcs(G), p≠i, p≠m }
    end while
end for
end AC-2

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

**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 if
end while
end 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)
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 $\{<j, b> | b \in D_j, (a, b) \in C_{i,j}\}$

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

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

<table>
<thead>
<tr>
<th>counter (i,j)</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>&lt;i,a1&gt;,&lt;i,a2&gt;</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>&lt;i,a1&gt;</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>&lt;i,a2&gt;,&lt;i,a3&gt;</td>
</tr>
</tbody>
</table>

**Using the support sets:**
1. Let b3 is deleted from the domain of j (for some reason).
2. Look at Sj,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).

![Algorithm AC-4](image)

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

```
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 Sj,b do
            counter[(i,j),a] ← counter[(i,j),a] - 1
            if counter[(i,j),a] = 0 & "a" is still in Di then
                delete "a" from Di
                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)

...
1) Consistency of an arc is required just in one direction.
2) Variables are ordered

there is no directed cycle in the graph!

If arcs are explored in a „good“ order, no revision has to be repeated!

Algorithm DAC-1

```
procedure DAC-1(G)
    for j = |nodes(G)| to 1 by -1 do
        for each arc (i,j) in G such that i<j do
            REVISE((i,j))
        end for
    end for
end DAC-1
```
Relation between DAC and AC

Observation:
CSP is arc consistent iff for some order of the variables, the problem is directional arc consistent in both directions.

Is it possible to achieve AC by applying DAC in both primal and reverse direction?
In general NO, but ...

Example:
X in \{1,2\}, Y in \{1\}, Z in \{1,2\}, \quad X \neq Z, Y < Z

- using the order X, Y, Z, there is no domain change
- using the order Z, Y, X, the domain of Z is changed but the graph is not AC

However if the order Z, Y, X is used first then we get AC!

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:

So what is the benefit of AC?

- Sometimes we have a solution after AC
  - any domain is empty \(\Rightarrow\) no solution exists
  - all the domains are singleton \(\Rightarrow\) 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 $(V_0, V_1, ..., V_m)$ is path consistent iff for every pair of values $x \in D_0, y \in D_m$ satisfying all the binary constraints on $V_0, V_m$ there exists an assignment of variables $V_1, ..., V_{m-1}$ 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 ($A<B, B<C \Rightarrow A+1<C$)
- 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

![Path Diagram]

\[
\begin{pmatrix}
011 \\
001 \\
000
\end{pmatrix}
& \times
\begin{pmatrix}
100 \\
010 \\
001
\end{pmatrix}
= \begin{pmatrix}
000 \\
001 \\
000
\end{pmatrix}
\]

Mackworth (1977)

**Algorithm PC-1**

How to make the path \((i,k,j)\) consistent?
\[ R_{ij} \leftarrow R_{ij} \& \left( R_{ik} \ast R_{kk} \ast R_{kj} \right) \]

How to make a CSP path consistent?
Repeated revisions of paths (of length 2) while any domain changes.

**procedure** PC-1(Vars,Constraints)

\[
\begin{align*}
n & \leftarrow |Vars|, \ Y^0 \leftarrow \text{Constraints} \\
& \text{repeat} \\
\ Y^0 & \leftarrow Y^0 \\
& \text{for } k = 1 \text{ to } n \text{ do} \\
& \quad \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad \quad Y_{ij} \leftarrow Y_{ij} \& \left( Y_{ik}^{k-1} \ast Y_{kk}^{k-1} \ast Y_{kj}^{k-1} \right) \\
& \quad \text{until } Y^n = Y^0 \\
\ \text{Constraints} & \leftarrow Y^0 \\
& \text{end PC-1}
\end{align*}
\]
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

```plaintext
procedure REVISE_PATH((i,k,j))

Z ← $Y_{ij} \land (Y_{ik} \land Y_{kk} \land Y_{kj})$

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 $Y_{ji} = Y^\prime_{ij}$ 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)

$S_a = \{(i,j,m) | i ≤ m ≤ n \land m≠j\} \cup \{(m,i,j) | 1 ≤ m ≤ j \land m≠i\} \cup \{(j,i,m) | j < m ≤ n\} \cup \{(m,j,i) | 1 ≤ m < i\}$

$| S_a | = 2n-2$

**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

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

$| S_b | = n^2(n-1)/2 - 2$
Algorithm PC-2

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

```plaintext
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

-记忆消耗
  - 因为PC消除了值对，我们需要保存所有兼容的值对，例如使用{0,1}-矩阵

- 坏的比例 强度/效率
  - PC消除了更多的（或相同的）不一致，但强度/效率比例比AC差得多

- 修改约束网络
  - PC添加了冗余弧（约束）并且改变了约束网络的连通性
  - 这复杂化了从结构推断的启发式的使用（如密度，图宽等）

- PC仍然是不完整的技术

A, B, C, D in {1,2,3}
A ≠ B, A ≠ C, A ≠ D, B ≠ C, B ≠ D, C ≠ D
是PC但没有解决方案

Half way between AC and PC

Can we make a consistency algorithm:
- 强于AC，
- 无PC的缺点（记忆消耗，改变约束网络）？

Restricted path consistency (Berlandier 1995)
- 基于AC-4（使用支持集）
- 一旦一个值在另一个变量中只有一个支持，PC被触发为这个值对
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**

**Definition:**

CSP is strongly k-consistent iff it is j-consistent for every j ≤ k.

Visibly: strong k-consistency ⇒ k-consistency
Moreover: strong k-consistency ⇒ j-consistency ∀j ≤ k
In general: ¬ k-consistency ⇒ 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

<table>
<thead>
<tr>
<th>1,2,\ldots,n-1</th>
<th>1,2,\ldots,n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\neq</td>
<td>\neq</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>1,2,\ldots,n-1</td>
<td>1,2,\ldots,n-1</td>
</tr>
</tbody>
</table>

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

And what about this graph?

1,2 \equiv 1,2,3 1,2,3 1,2,3

(D)AC is enough! Because this a tree.

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
a set of binary inequality constraints among all variables
\[ X_1 \neq X_2, \ X_1 \neq X_3, \ldots, \ X_{k-1} \neq X_k \]
\[ \text{all\_different}(\{X_1, \ldots, X_k\}) = \{(d_1, \ldots, d_k) | \forall i \ d_i \in D_i \ \& \ \forall i \neq j \ d_i \neq d_j\} \]
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
How to solve CSPs?

So far we have two separate methods:

- **depth-first search**
  - complete (finds a solution or proves its non-existence)
  - too slow (exponential)
  - explores “visibly” wrong valuations

- **consistency techniques**
  - usually incomplete (inconsistent values stay in domains)
  - pretty fast (polynomial)

Share advantages of both approaches - combine them!

- label the variables step by step (backtracking)
- maintain consistency after assigning a value

Do not forget about **traditional solving techniques**!

- Linear equality solvers, simplex ...
- such techniques can be integrated to global constraints!

There is also **local search**.

---

Core search procedure - DFS

The basic constraint satisfaction technology:

- label the variables step by step
  - the variables are marked by numbers and labelled in a given order
- ensure consistency after variable assignment

A skeleton of search procedure

```
procedure Labelling(G)
  return LBL(G,1)
end Labelling

procedure LBL(G,cv)
  if cv>|nodes(G)| then return nodes(G)
  for each value V from Dcv do
    if consistent(G,cv) then
      R ← LBL(G,cv+1)
      if R ≠ fail then return R
    end if
  end for
  return fail
end LBL
```

A "hook" for consistency procedure
Look back techniques

“Maintain” consistency among the already labelled variables.

„look back“ = look to already labelled variables

What’s result of consistency maintenance among labelled variables?

a conflict (and/or its source - a violated constraint)

Backtracking is the basic look back method.

Backward consistency checks

procedure AC-BT(G,cv)
Q ← \{(Vi,Vcv) \in arcs(G), i < cv\} % arcs to labelled variables.
consistent ← true
while consistent & Q non empty do
    select and delete any arc \((V_k,V_m)\) from Q
    consistent ← not REVISE\((V_k,V_m)\)
end while
return consistent
end AC-BT

Backjumping & comp. uses information about the violated constraints.

Forward checking

It is better to prevent failures than to detect them only!

Consistency techniques can remove incompatible values for future (=not yet labelled) variables.

Forward checking ensures consistency between the currently labelled variable and the variables connected to it via constraints.

Forward consistency checks

procedure AC-FC(G,cv)
Q ← \{(V_i,V_{cv}) \in arcs(G), i > cv\} % arcs to future variables
consistent ← true
while consistent & Q non empty do
    select and delete any arc \((V_k,V_m)\) from Q
    if REVISE\((V_k,V_m)\) then
        consistent ← not empty \(D_k\)
    end if
end while
return consistent
end AC-FC

Empty domain implies inconsistency

When a value is deleted, the domain is empty
Partial look ahead

We can extend the consistency checks to more future variables!
The value assigned to the current variable can be propagated to all future variables.

Partial lookahead consistency checks

```plaintext
procedure DAC-LA(G,cv)
    for i=cv+1 to n do
        for each arc (Vi,Vj) in arcs(G) such that i>j & j≥cv do
            if REVISE(Vi,Vj) then
                if empty Di then return fail
            end if
        end for
    end for
    return true
end DAC-LA
```

Notes:
- In fact DAC is maintained (in the order reverse to the labelling order).
- Partial Look Ahead or DAC - Look Ahead
- It is not necessary to check consistency of arcs between the future variables and the past variables (different from the current variable)!

Full look ahead

Knowing more about far future is an advantage!
Instead of DAC we can use a full AC (e.g. AC-3).

Full look ahead consistency checks

```plaintext
procedure AC3-LA(G,cv)
    Q ← {(Vi,Vcv) in arcs(G),i>cv}               % start with arcs going to cv
    consistent ← true
    while consistent & Q non empty do
        select and delete any arc (Vk,Vm) from Q
        if REVISE(Vk,Vm) then
            Q ← Q ∪ {(Vi,Vk) | (Vi,Vk) in arcs(G), i≠k,i≠m,i>cv}
            consistent ← not empty Dk
        end if
    end while
    return consistent
end AC3-LA
```

Notes:
- The arcs going to the current variable are checked exactly once.
- The arcs to past variables are not checked at all.
- It is possible to use other than AC-3 algorithms (e.g. AC-4)
Comparison
4 queens

Backtracking is not very good
19 attempts

Forward checking is better
3 attempts

And the winner is Look Ahead
2 attempts

Constraint propagation at glance

- Propagating through more constraints remove more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the step.
- Forward Checking does no increase complexity of backtracking, the constraint is just checked earlier in FC (BT tests it later).
- When using AC-4 in LA, the initialisation is done just once.
- Consistency can be ensured before starting search
  - **Algorithm MAC (Maintaining Arc Consistency)**
    - AC is checked before search and after each assignment
    - It is possible to use stronger consistency techniques (e.g. use them once before starting 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 \( 1..3 \) \( \Rightarrow \) \( X=1 \lor X=2 \lor X=3 \) (enumeration)

In general, search algorithm resolves remaining disjunctions!
- \( X=1 \lor X=1 \) (step labeling)
- \( X<3 \lor X\geq 3 \) (bisection)
- \( X<Y \lor X\geq Y \) (variable ordering)

**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-FI RST 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
**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.

![Diagram showing backtrack-free search]

**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 minimizing m is looked for.

---

**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 FIRST-FAIL!
- Prefer the values with more support.
  - 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!
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 \( 0..M \times M \)

- \( X_1 = 0 \) \( \text{ruler start} \)
- \( X_1 < X_2 < \ldots < X_M \) \( \text{no permutations of variables} \)
- \( \forall i < j \) \( D_{ij} = X_j - X_i \) \( \text{difference variables} \)
- \( \text{all different}(\{D_{1,2}, D_{1,3}, \ldots, D_{1,M}, D_{2,3}, \ldots, D_{M,M-1}\}) \)

Model extensions:

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

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

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

\[ \text{so } D_{ij} \geq \frac{1}{2} \sum_{j=i}^{j} (j-i)(j-i+1) \] \( \text{lower bound} \)

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

\[ \text{so } D_{ij} \leq X_M - (M-1-j+i)(M-j+i)/2 \] \( \text{upper bound} \)

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>13 690</td>
<td>5 438</td>
<td>1 001</td>
</tr>
<tr>
<td>10</td>
<td>120 363</td>
<td>49 971</td>
<td>7 011</td>
</tr>
<tr>
<td>11</td>
<td>2 480 216</td>
<td>985 237</td>
<td>170 495</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>2 664</td>
<td>2 384</td>
</tr>
<tr>
<td>10</td>
<td>20 870</td>
<td>17 545</td>
</tr>
<tr>
<td>11</td>
<td>1 004 515</td>
<td>906 323</td>
</tr>
</tbody>
</table>
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 (AC-3)
  - 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
  - **Java**: JCK, JCL, Koalog
  - Mozart
**Resources**

- **Books**

- **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.

We still did not reach the Holy Grail of computer programming (the user states the problem, the computer solves it) but CP is close.