What is a constraint?

A constraint is an arbitrary relation over the set of variables. Every variable has a set of possible values - a domain. This course covers discrete finite domains only. The constraint restricts the possible combinations of values.

Some examples:
- The circle C is inside a square S
- The length of the word W is 10 characters
- X is less than Y
- A sum of angles in the triangle is 180°
- The temperature in the warehouse must be in the range 0-5°C
- John can attend the lecture on Wednesday after 14:00

Constraint can be described:
- Intentionally (as a mathematical/logical formula)
- Extensionally (as a table describing compatible tuples)

CSP (Constraint Satisfaction Problem) consists of:
- A finite set of variables
- Domains - a finite set of values for each variable
- A finite set of constraints

A solution to CSP is a complete assignment of variables satisfying all the constraints.

CSP is often represented as a (hyper)graph.

Example:
- Variables x₁,...,x₆
- Domain {0,1}
- Constraints:
  1. x₁ + x₂ + x₆ = 1
  2. x₁ - x₃ + x₄ = 1
  3. x₄ + x₅ - x₆ > 0
  4. x₂ + x₅ - x₆ = 0

Some toy problems

SEND + MORE = MONEY
Assign different numerals to different letters S and M are not zero
A constraint model (with a carry bit):
- E, M, D, O, R, Y in 0..9, S, M in 1..9, P₁, P₂, P₃ :: 0..1
- All different (S, E, N, D, M, O, R, Y)
- S + O = 10*P₁ + Y
- P₁ + N + R = 10*P₂ + E
- P₂ + E + O = 10*P₃ + N
- P₃ + S + M = 10*M + O

N-queens problem
Allocate N queens to the chessboard the queens do not attack each other
A constraint model:
- Queens in columns \( v_i x(j) \) in 1..N
- No conflict
  \[ \forall i,j \quad x(i) - x(j) \& |i-j| = |x(i) - x(j)| \]

A bit of history

Artificial Intelligence
Scene labelling (Waltz 1975)
Interactive graphics
Sketchpad (Sutherland 1963)
ThingLab (Borning 1981)
Logic programming
Unification \(\rightarrow\) Constraint solving (Gallaire 1985, Jaffar, Lassez 1987)

Operations research and discrete mathematics
NP-hard combinatorial problems
**Constraints in scene labelling (Waltz 1975)**

Looking for feasible interpretation of 3D lines in 2D drawing
First usage of constraint propagation techniques

**Constraints in interactive graphics**

How to manipulate a graphical object described by constraints?

```
http://www.cs.washington.edu/research/constraints/
```

**Constraints in A.I. planning and scheduling**

Scheduling problem = a set of activities has to be processed by a limited number of resources in a limited amount of time.
Combinatorial optimisation

Planning problem = find a set of activities to achieve a given goal
Deep Space One – autonomous planning of spacecraft activities

**Constraints in bioinformatics**

Design of a 3D protein structure from the sequence of amino-acids (3D structure determines features of proteins)

Analysing a sequence of DNA, estimating a distance between DNAs, comparing DNAs

```
http://www.soi.city.ac.uk/~drg/bioinformatics/
```

**Solving constraints by enumeration**

Constraints are used only as a test assign values to variables ...
... and see what happens

systematic search explores the space of all assignments systematically
GT, BT, BJ, BM, DB, LDS

non-systematic search some assignments may be skipped during search
Credit Search, Bounded Backtrack

local search explore the search space by small steps
HC, MC, RW, Tabu, GSAT, Genet, simulated annealing

**Systematic search**

Explore systematically the space of all assignments systematic = every valuation will be explored sometime

Features:
+ complete (if there is a solution, the method finds it)
- it could take a lot of time to find the solution

Basic classification:
Explore complete assignments generate and test such search space is used by local search (non-systematic)

Extending partial assignments

tree search
**Hill Climbing**

Hill climbing is perhaps the most known technique of local search. Start at randomly generated state. Look for the best state in the neighbourhood of the current state. Neighbourhood size = \( S_{\text{neighbourhood}} = (D-1) = n^*(d-1) \)

```
procedure hill-climbing(Max_Flips)
    restart := random assignment of variables;
    for j:=1 to Max_Flips do % restricted number of steps
        if eval(s)>0 then return s
        if s is a strict local minimum then
            go to restart
        else
            s := neighbourhood with the smallest evaluation value
        end if
        end for
    end while
    go to restart
end hill-climbing
```

**Algorithm Hill Climbing**

**Generate and test (GT)**

The most general problem solving method

1. generate a candidate for solution
2. test if the candidate is really a solution

**How to apply GT to CSP?**

1. assign values to all variables
2. test whether all the constraints are satisfied

GT explores complete but inconsistent assignments until a (complete) consistent assignment is found.

```
procedure GT(X:variables, C:constraints)
    V := construct a first complete assignment of X
    while V does not satisfy all the constraints C do
        V := construct systematically a complete assignment next to V
        end while
    end while
    return V
```

**Local search - Terminology**

- **state** - a complete assignment of values to variables
- **evaluation** - a value of the objective function (# violated constraints)
- **neighbourhood** - a set of states locally different from the current state
- **local optimum** - a state that is not optimal and there is no state with better evaluation in its neighbourhood
- **strict local optimum** - a state that is not optimal and there is no state with better evaluation in its neighbourhood
- **non-strict local optimum** - local optimum that is not strict
- **global optimum** - the state with the best evaluation
- **plateau** - a set of neighbouring states with the same evaluation

**Min-Conflicts (Minton, Johnston, Laird 1997)**

Observation:

- the hill climbing neighbourhood is pretty large (\( n^*(d-1) \))
- only change of a conflicting variable may improve the valuation

**Min-conflicts method**

select randomly a variable in conflict and try to improve it

```
procedure MC(Max_Moves)
    s := random assignment of variables
    nb_moves := 0
    while eval(s)>0 & nb_moves<Max_Moves do
        choose randomly a variable V in conflict
        choose a value v' that minimises the number of conflicts for V
        if v' = current value of V then
            assign v' to V
            nb_moves := nb_moves+1
        end if
        end while
    end if
    return s
end while
```

**Algorithm Min-Conflicts**

**Generate and test (GT)**

The most general problem solving method

1. generate a candidate for solution
2. test if the candidate is really a solution

**How to apply GT to CSP?**

1. assign values to all variables
2. test whether all the constraints are satisfied

GT explores complete but inconsistent assignments until a (complete) consistent assignment is found.

```
procedure GT(X:variables, C:constraints)
    V := construct a first complete assignment of X
    while V does not satisfy all the constraints C do
        V := construct systematically a complete assignment next to V
        end while
    end while
    return V
```

**How to improve generate and test?**

- **smart generator**
  - smart (perhaps non-systematic) generator that uses result of test on earlier detection of clash
  - constraints are tested as soon as the involved variables are instantiated → backtracking-based search

**Weaknesses and improvements of GT**

The greatest weakness of GT is exploring too many “visibly” wrong assignments.

**Example:**

\[
\begin{array}{cccccccc}
X & Y & Z & \{1,2\} & X = Y, X = Z, Y > Z
\end{array}
\]

```
1 & 1 & 1 & 2 & 2 & 2 & 1 & 2
1 & 1 & 2 & 2 & 1 & 1 & 2 & 1
1 & 2 & 1 & 2 & 1 & 1 & 2 & 1
\]
```

**How to apply GT to CSP?**

1. assign values to all variables
2. test whether all the constraints are satisfied

GT explores complete but inconsistent assignments until a (complete) consistent assignment is found.

```
procedure GT(X:variables, C:constraints)
    V := construct a first complete assignment of X
    while V does not satisfy all the constraints C do
        V := construct systematically a complete assignment next to V
        end while
    end while
    return V
```

**How to improve generate and test?**

- **smart generator**
  - smart (perhaps non-systematic) generator that uses result of test on earlier detection of clash
  - constraints are tested as soon as the involved variables are instantiated → backtracking-based search

**Weaknesses and improvements of GT**

The greatest weakness of GT is exploring too many “visibly” wrong assignments.

**Example:**

\[
\begin{array}{cccccccc}
X & Y & Z & \{1,2\} & X = Y, X = Z, Y > Z
\end{array}
\]

```
1 & 1 & 1 & 2 & 2 & 2 & 1 & 2
1 & 1 & 2 & 2 & 1 & 1 & 2 & 1
1 & 2 & 1 & 2 & 1 & 1 & 2 & 1
\]
```

**How to improve generate and test?**

- **smart generator**
  - smart (perhaps non-systematic) generator that uses result of test on earlier detection of clash
  - constraints are tested as soon as the involved variables are instantiated → backtracking-based search

**Weaknesses and improvements of GT**

The greatest weakness of GT is exploring too many “visibly” wrong assignments.

**Example:**

\[
\begin{array}{cccccccc}
X & Y & Z & \{1,2\} & X = Y, X = Z, Y > Z
\end{array}
\]

```
1 & 1 & 1 & 2 & 2 & 2 & 1 & 2
1 & 1 & 2 & 2 & 1 & 1 & 2 & 1
1 & 2 & 1 & 2 & 1 & 1 & 2 & 1
\]
```

**How to improve generate and test?**

- **smart generator**
  - smart (perhaps non-systematic) generator that uses result of test on earlier detection of clash
  - constraints are tested as soon as the involved variables are instantiated → backtracking-based search
Random walk

How to leave the local optimum without a restart (i.e. via a local step)?
By adding some “noise” to the algorithm!

Random walk

A state from the neighborhood is selected randomly
(e.g., the value is chosen randomly)
such technique can hardly find a solution
so it needs some guide

Random walk can be combined with the heuristic guiding
the search via probability distribution:

\[ p \] - probability of using the random walk
\[ (1-p) \] - probability of using the heuristic guide

---

Min-Conflicts Random Walk

MC guides the search (i.e., satisfaction of all the constraints) and RW allows us to leave the local optima.

Algorithm Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
    s <- random assignment of variables
    nb_moves <- 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose randomly a variable V in conflict
            choose a value v' that minimises the number of conflicts for V
        end if
        if v' != current value of V then
            assign v' to V
            nb_moves <- nb_moves+1
        end if
    end while
    return s
end MCRW
```

---

Steepest Descent Random Walk

Random walk can be combined with the hill climbing heuristic too.
Then, no restart is necessary.

Algorithm Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
    s <- random assignment of variables
    nb_moves <- 0
    while eval(s)>0 & nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose a move <V,v'> with the best performance
        end if
        if v' != current value of V then
            assign v' to V
            nb_moves <- nb_moves+1
        end if
    end while
    return s
end SDRW
```

---

Tabu list

Observation:

- Being trapped in local optimum is a special case of cycling.
- How to avoid cycles in general?
  - Remember already visited states and do not visit them again.
  - Prevents “short” cycles

Tabu list

- A list of forbidden states
- The state can be represented by a selected attribute
- The change of the state (a previous value)
- The tabu list has a fixed length k (tabu tenure)
- “Old” states are removed from the list when a new state is added
- It is possible to remember just few last states
- Aspiration criterion
  - Enables states that are tabu
  - Example: the state is better than any state visited so far

---

Tabu search (Galinier, Hao 1997)

The tabu list prevents short cycles.
It allows only the moves out of the tabu list or the moves satisfying the aspiration criterion.

Algorithm Tabu Search

```
procedure tabu-search(Max_Iter)
    s <- random assignment of variables
    nb_iter <- 0
    initial tabu list
    while eval(s)>0 & nb_iter<Max_Iter do
        choose a move <V,v'> with the best performance
        introduce <V,v> in the tabu list
        remove the oldest move from the tabu list
        select n in neighbourhood(s)
        assign v' to V
        nb_iter <- nb_iter+1
    end while
    return best s
end tabu-search
```

---

Localizer (Michel, Van Hentenryck 1997)

The local search algorithms have a similar structure that can be encoded in the common skeleton. This skeleton is filled by procedures implementing a particular technique.

Algorithm Local Search

```
procedure local-search(Max_Tries,Max_Moves)
    s <- random assignment of variables
    for i:=1 to Max_Tries while Gcondition do
        for j:=1 to Max_Moves while Lcondition do
            if eval(s)=0 then
                return s
            end if
            select n in neighbourhood(s)
            if acceptable(n) then
                assign v' to V
            end if
            remove the oldest move from the tabu list
        end for
    end for
    return best s
end local-search
```

---
Hybrid encoding

Transformation between dual and hidden variable encoding contains parts of both encodings

Double encoding

Hidden and original variables are included

Constraints from both encodings are used

Improved propagation

Binary constraints

World is not binary ...
but it could be transformed to a binary one!

Each CSP can be transformed to an equivalent binary CSP
– many CSP algorithms designed for binary problems
– still open efficiency issues

Projection technique (Montanary 1974):
• straightforward but
• does not give an equivalent problem
• bound consistency
• better efficiency
• weaker pruning

Foundations of constraint satisfaction, Roman Barták

Backtracking

Probably the most widely used systematic search algorithm
basically it is depth-first search

Using backtracking to solve CSP
1) assign values gradually to variables
2) after each assignment test the constraints over the assigned variables (and backtrack upon failure)

Extends a partial consistent assignment until a complete consistent assignment is found.

Open questions:
• what is the order of variables?
  • variables with a smaller domain first
  • variables participating in more constraints first
  • “key” variables first
• what is the order of values?
  • problem dependent
Algorithm chronological backtracking
A recursive definition

```
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 constraints C are consistent with V+{x/h} then
        R := BT(X-x, V+{x/h}, C)
        if R != fail then return R
end for
return fail
```
call BT(X, {}, C)
```

Weaknesses of backtracking

- **Thronsh**
  - 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 (Gaschnig 1979)
Backjumping is used to remove thrashing.

**How?**
1) identify the source of the conflict (impossible to assign a value)
2) jump to the past variable in conflict

The same run like in backtracking, only the back-jump can be longer, i.e. 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

**Consistency check for backjumping**
In addition to the test of satisfaction of the constraints, the closest conflicting level is computed

```
procedure consistent(Labelled, 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 Labelled then
        if C is not satisfied by Labelled then
            NoConflict := false
            J := min (J, max{L | X in C & X/V/L in Labelled & L < Level})
        end if
    end if
end for
if NoConflict then return true
else return fail(J)
```
```
Identification of the conflicting variable
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)
```

```
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 variable from the above chosen variables for each value (°).
3. Select the closest variable from the conflicting variables selected for each value and jump to it.

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

Consistency check for backjumping
In addition to the test of satisfaction of the constraints, the closest conflicting level is computed

```
procedure consistent(Labelled, 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 Labelled then
        if C is not satisfied by Labelled then
            NoConflict := false
            J := min (J, max{L | X in C & X/V/L in Labelled & L-Level})
        end if
    end if
end for
if NoConflict then return true
else return fail(J)
```
```
Identification of the conflicting variable
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)
```

```
Order of assignment

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>*</td>
<td>•</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

Neither 0 nor 1 can be assigned to the seventh variable!
1. Find the closest variable in each violated constraint (o).
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.
```

```
Algorithm backjumping

procedure BJ(Variables, Labelled, Constraints, PreviousLevel)
  if Unlabelled = {} then return Labelled
  pick first X from Unlabelled
  Level ← PreviousLevel+1
  Jump ← X
  for each value V from DX do
    if C = fail(J) then
      Jump ← max (Jump, J)
      return fail(J)
    C = V
    if ValuesX = {} then
      return fail(J)
    select X in Unlabelled
    if X is the farthest variable to which we backtracked since
    where Jump to the conflicting variable
    end if
    return fail(J)
  end for
  return BJ(Unlabelled-{X},{X/V/Level},Constraints, PreviousLevel+1)
call BJ(Variables,{},Constraints,0)

Dynamic backtracking - example

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

Algorithm dynamic backtracking (Ginsberg 1993)

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
      return failure
    let E be an explanation of the conflict (set of conflicting variables)
    if E = {} then failure
    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
    select V in ValuesX
    Labelled ← Labelled - {X}
    Labelled ← Labelled + {X/V/Level}
    end if
  end while
  return Labelled
end DB

Redundant work in backtracking

What is redundant work?

Example:

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

Backmarking (Haralick, Elliot 1980)

Removes redundant constraint checks by memorising 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:

\[ X_a \]
Algorithm backmarking

procedure BM(Unlabelled, Labelled, Constraints, Level)
    if Unlabelled = {} then return Labelled
    pick first X from Unlabelled % fix order of variables
    for each value V from Values(X) do
        if Values(X) = BackTo(X) then % re-check the value
            if consistent(X/V, Labelled, Constraints, Level) then
                R ← BM(Unlabelled-{X}, Labelled-{X/V}, Constraints, Level+1)
                if R = fail then return R % solution found
                end if
            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(Lower-1, BackTo(Y))
    end for
    return fail % return to the previous variable
end BM

Consistency check for backmarking

procedure consistent(X/V, Labelled, Constraints, Level)
    for each Y/VY/LY in Labelled such that LY = heuristic is not followed
        if consistent(X/V, Labelled, Constraints, Level+1) then
            if R = fail then return R % solution found
            end if
            R ← LDS-PROBE(Unlabelled-{X}, Labelled-{X/V}, Constraints, D-1)
            if R = fail then return R % solution found
            end if
            return LDS-PROBE(Unlabelled-{X}, Labelled-{X/V}, Constraints, D)
        end if
    end for
    return true
end consistent

Limited Discrepancy Search

Discrepancy = heuristic is not followed
(a value different from the heuristic is chosen)

Idea of Limited Discrepancy Search (LDS):
- first, follow the heuristic
- when a failure occurs then explore the paths when the heuristic is not followed maximally once (start with earlier violations)
- after next failure occurs then explore the paths when the heuristic is not followed maximally twice...

Example:
the heuristic proposes to use the left branches

Algorithm LDS (Harvey, Ginsberg 1995)

procedure LDS-PROBE(Unlabelled, Labelled, Constraints, D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    select V in Values(X) % D is a number of allowed discrepancies
    if Values(X) = {} then return fail
    else select HV in Values(X) using heuristic
        if for each value V from Values(X) do % the order of violations
            R ← LDS-PROBE(Unlabelled-{X}, Labelled-{X/V}, Constraints, D-1)
            if R = fail then return R
            end if
            if R = fail then return R
            end if
            return LDS-PROBE(Unlabelled-{X}, Labelled-{X/V}, Constraints, D)
        end if
    end if
end LDS-PROBE

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 leads to solution

What to do upon a failure of the heuristics?
- BT cares about the end of search (a bottom part of the search tree)
- so it rather repairs later assignments than the earliest ones
- it 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 (as search proceeds, more information for better decision is available).

Observation 3:
The number of heuristic violations is usually small.
Introduction to consistency techniques

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)
        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 (i,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 valuation V_i=x a V_j=y satisfies all the binary constraints on V_i, V_j.
- CSP is arc consistent iff every arc (i,j) is arc consistent (in both directions).

Example:
- A in 3..7, B in 1..5
- (A,B) and (B,A) are consistent
  
  no arc is consistent

Example:
- A in 3..7, B in 1..5
- (A,B) and (B,A) are consistent
  
  no arc is consistent

Algorithm AC-1 (Mackworth 1977)

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

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 valuation V_j=x, V_i=y satisfies all the binary constraints on V_i, V_j).

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

The procedure also reports the deletion of some value.

```
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 X, Y, then delete x from D_i
        if DELETED = true then
            end if
        end for
    end for
    return DELETED
end REVISE
```

```
procedure AC-1(G)
    repeat
        CHANGED = false
        for each arc (i,j) in G do
            if (i,j) is inconsistent then
                CHANGED = true
                REVISE((i,j))
            end if
        end for
        if CHANGED then
            offer AJ(?)
        end if
    until not(CHANGED)
end AC-1
```

Note:
- The concept of arc consistency is directional, i.e., arc consistency of (i,j) does not guarantee consistency of (j,i).
- CSP is arc consistent iff every arc (i,j) is arc consistent (in both directions).

Example:
- A in 3..7, B in 1..5
- (A,B) and (B,A) are consistent
  
  no arc is consistent

Foundations of constraint satisfaction, Roman Barták
**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).

**Algorithm AC-3 (Mackworth 1977)**

Re-revisions can be done more elegant 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-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)

**Looking for (and remembering of) the 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 $V_i V_j$ is revised, the value $a$ is removed from domain of $V_i$.
2. Now the domain of $V_j$ should be explored to find out if any value $a, b, c, d$ loses the support in $V_j$.

**The support set for $a$:**

$D_a = \{ b | (b, a) \in C \}$

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 supporters are kept.

**Computing supports and how to use them**

**Situation:**

we have just processed the arc $(V_i,V_j)$ in INITIALIZE

Using the support sets:

1. Let $b_3$ is deleted from the domain of $j$ (for some reason).
2. Look at $S_{b_3,j}$ to find out the values that were supported by $b_3$ (i.e. $a, b_2, b_3$).
3. Decrease the counter for these values (i.e. tell them that they lost one support).
4. If any counter is zero ($a_3$) then delete the value and repeat the procedure with the respective value (i.e. go to 1).
Algorithm AC-4 (Mohr, Henderson 1986)

The algorithm AC-4 has the optimal worst case!

```
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 Di
            delete "a" from Di
            Q ← Q ∪ {<i,a>}
    end if
end for
end while
end AC-4
```

Unfortunately the average efficiency is not so good … plus there is a big memory consumption!

Other arc consistency algorithms

AC-5 (Hentenryck, Deville, Teng 1992)
- a generic arc-consistency algorithm
- can be reduced both to AC-3 and AC-4
- exploits semantic of the constraint
  functional, anti-functional, and monotonic constraints

AC-6 (Bessiere 1994)
- improves memory complexity and average time complexity of AC-4
- keeps one support only, the next support is looked for when the current support is lost

AC-7 (Bessiere, Freuder, Regin 1999)
- based on computing supports (like AC-4 and AC-6)
- exploits symmetry of the constraint

Directional arc consistency (DAC)

Observation 1: AC has a directional character but CSP is not directional.
Observation 2: AC has to repeat arc revisions; the total number of revisions depends on the number of arcs but also on the size of domains (while cycle).

Is it possible to weaken AC in such a way that every arc is revised just once?

Definition: CSP is directional arc consistent using a given order of variables iff every arc (i,j) such that i<j is arc consistent.

Again, every arc has to be revised, but revision in one direction is enough now.

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
```

Algorithm DAC-1

If the arc are explored in a “good” order, no revision has to be repeated!

How to use DAC

AC visibly covers DAC (if CSP is AC then it is DAC as well)
So, is DAC useful?
- DAC-1 is surely much faster than any AC-x
- there exist problems where DAC is enough

Example: If the constraint graph forms a tree then DAC is enough to solve the problem without backtracks.

How to order the vertices for DAC?

1. Apply DAC in the order from the root to the leaf nodes.
2. Label vertices starting from the root.

DAC guarantees that there is a value for the child node compatible with all the parents.

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:

\[ X \in \{1,2\}, Y \in \{1\}, Z \in \{1,2\}, \]
\[ X \neq Z, Y \neq Z \]
using the order \(X,Y,Z\)
there is no domain change
\[ X \neq Z, Y \neq Z \]
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 then we get AC!
Constraint semantics is used!

It is possible to use different levels of consistency for different constraints!

The constraint $C_i$ is arc consistent iff for every variable $i$ constrained by $C_i$, and for every value $v_i$ of $C_i$, there is an assignment of the remaining variables in $C_i$ such that the constraint is satisfied.

Example: $A + B = C$, $A$ in $1..3$, $B$ in $2..4$, $C$ in $3..7$ is AC

Interval consistency
- working with intervals rather than with individual values
- interval arithmetic

Example: after change of $A$ we compute $A + B \to C$, $C - A \to B$

Bound consistency
- only lower and upper bound of the domain are propagated
- Such techniques do not provide full arc consistency!

It is possible to use different levels of consistency for different constraints!

—

From DAC to AC for tree-structured CSP

If we apply DAC to tree-structured CSP first using the order from the root to the leaf nodes and second in the reverse direction then we get (full) arc consistency.

Proof:
- the first run of DAC ensures that any value in the parent node has a support (a compatible value) in all the child nodes.
- If any value is deleted during the second run of DAC (in the reverse direction) then this value does not support any value in the parent node (the values in the parent node does not lose any support)
- together: every value has some support in the child nodes (the first run) as well as in the parent node (the second run), i.e., we have AC

—

Consistency techniques in practice

N-ary constraints are processed directly!

- The constraint $C_i$ is arc consistent when the domain of involved variable is changed
- when minimum/maximum bound is changed
- when the variable becomes singleton

- different suspensions for different variables
  - Example: $A \leq B$ filtering evoked after change of min($A$) or max($B$)
  - directional consistency

- 2) Design the filtering algorithm for the constraint
- the result of filtering is the change of domains
- more filtering procedures for a single constraint are allowed
- Example: $A \leq B$
  - $\min(A) \in [\min(A) \ldots \sup]$
  - $\max(B) \in [\inf \ldots \max(B) - 1]$

—

Base propagation algorithm

Based on generalisation of AC-3.

Repeat constraint revisions until any domain is changed.

```
procedure AC-3(C)
  Q % a list of constraints for revision
  while Q non empty do
    select and delete c from Q
    for all (X,Y) in C-A
      fd_max(X,MaxX), fd_min(Y,MinY), fd_max(Y,MaxY), Y-A \to B, X-A \to B
      Actions = \[X in LowerBoundX..\sup\], Actions = \[B in LowerBoundB..\sup\]
      if any value is deleted during the second run of DAC
        reverse direction then we get (full) arc consistency.
      order from the root to the leaf nodes and second in the reverse direction then this value does not support any value in the parent node (the values in the parent node does not lose any support)
      together: every value has some support in the child nodes (the first run) as well as in the parent node (the second run), i.e., we have 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.

Definition of a constraint (SICStus Prolog)

How to describe propagation through $A \leq B$?
- bound consistency is enough for full consistency!

- Example:
  - $\min(A) \in [\min(A) \ldots \sup]$
  - $\max(B) \in [\inf \ldots \max(B) - 1]$

- Example:
  - $\min(A) \in [\min(A) \ldots \sup]$
  - $\max(B) \in [\inf \ldots \max(B) - 1]$

—

Design of consistency algorithms

The user can often define the code of REVISE procedure.

How to do it?

1) Decide about the event to evoke the filtering
- when the domain of involved variable is changed
  - whenever the domain changes
  - when minimum/maximum bound is changed
  - when the variable becomes singleton

- different suspensions for different variables
  - Example: $A \leq B$ filtering evoked after change of min($A$) or max($B$)
  - directional consistency

2) Design the filtering algorithm for the constraint
- the result of filtering is the change of domains
- more filtering procedures for a single constraint are allowed

- Example: $A \leq B$
  - $\min(A) \in [\min(A) \ldots \sup]$
  - $\max(B) \in [\inf \ldots \max(B) - 1]$

—

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.

Definition of a constraint (SICStus Prolog)

How to describe propagation through $A \leq B$?
- bound consistency is enough for full consistency!

- Example:
  - $\min(A) \in [\min(A) \ldots \sup]$
  - $\max(B) \in [\inf \ldots \max(B) - 1]$

PC and paths of length 2 (Montanari)

It is not very practical to ensure consistency of all paths—fortunately, only the paths of length 2 can be explored!

Theorem: CSP is PC iff every path of length 2 is PC.

Proof:
1) PC → paths of length 2 are PC.
2) (paths of length 2 are PC) → ∀N paths of length N are PC.
   - induction using the path length

   a) N=1 (proposition already holds for N)
   - take arbitrary N+1 vertices V_{i0}, V_{i1},..., V_{in-1}
   - take arbitrary pair of compatible values x, y
   - from a) we can find x_{i0}, D_x, y_{i0}, D_y
   - iii) from a) we can find x_{i0}, D_x, and constraints C_{i01}, a C_{i0n}, hold
   - iv) from the induction we can find the values for V_{i0}, V_{i1},..., V_{in-1}

   b) N+1 (proposition already holds for N)
   - take arbitrary N+1 vertices V
   - from a) we can find x
   - iii) from a) we can find x_{i0}, D_x, and constraints C_{i01}, a C_{i0n}, hold
   - iv) from the induction we can find the values for V_{i0}, V_{i1},..., V_{in-1}

   CSP is PC iff every path of length 2 is PC.

Path consistency (PC)

How to strengthen the consistency level?

- More constraints are assumed together!

Definition:
- The path (V_{i0}, V_{i1},..., V_{in-1}) is path consistent if for every pair of values x_{i0}, y_{i1} satisfying all the binary constraints on V_{i0}, V_{i1}, there exists an assignment of variables V_{i1},..., V_{in-1}, such that all the binary constraints between the neighbouring variables V_{i0}, V_{i1}, are satisfied.

Attention!

Path consistency does not guarantee that all the constraints among the variables on the path are satisfied; only the constraints between the neighbouring variables must be satisfied.

A matrix representation of constraints

In PC we need to exclude the pairs of values

≥ the constraints must be represented in explicit form

0 - the values are incompatible
1 - the values are compatible

Example:

5-queens problem

the constraint between queens i, j: r(i)=r(j) & |i-j| = |r(i)-r(j)|

A matrix for queens A(1), B(2):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A matrix for queens A(1), C(3):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Operations over the constraints

Intersection R_A & R_B

A ∩ B = A \cap B = B \cap A

Composition R_A ∘ R_B

A \circ B = B \circ A

The induced constraint is joined with the original constraint

R_A \circ (R_B \circ R_A) = R_A

Notes:

R_A = R_A^T, R_A is a diagonal matrix representing the domain
REVISE((i,j)) from AC is equivalent to R_i \leftarrow R_i \circ (R_i^* \circ R_i)

Notes:

- The induced constraint is joined with the original constraint
- R_A \circ (R_B \circ R_A) = R_A

Notes:

- REVISE((i,j)) from AC is equivalent to R_i \leftarrow R_i \circ (R_i^* \circ R_i)
How to make a CSP consistent?

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

\[ R_k \leftarrow R_k \land (R_{k-1} \land R_k) \]

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

- All the paths containing \((i,j)\) or \((j,i)\) must be re-revised
- The paths \((i,i,i)\) and \((k,i,k)\) are not revised again (no change)
- The grand problem: After domain change all the paths are re-revised.
- It is enough to revise the influenced paths.

Procedure: \( \text{REVISE\_PATH}(i,k,j) \)

- If the domain of the constraint \((i,j)\) is changed when revising \((i,k,j)\):
  - \( Y_{ij} \leftarrow Y_{ij} \land \text{(other constraints)} \)
  - \( Y_{ik} \leftarrow Y_{ik} \land \text{(other constraints)} \)
  - \( Y_{kj} \leftarrow Y_{kj} \land \text{(other constraints)} \)
  - The influenced paths will be revised.

Which paths are influenced by the revision?

Because \( Y_{ij} = Y_{ij} \) it is enough to revise only the paths \((i,k,j)\) where \( i < j \).

Let the domain of the constraint \((i,j)\) is changed when revising \((i,k,j)\):

\[ S_a = \{(i,j,m) | i \leq m < n \land m < j \} \]

- Situation 1:
  - \( a \): all the paths containing \((i,j)\) or \((j,i)\) must be re-revised.
  - The paths \((i,j,j)\), \((i,i,j)\) are not revised again (no change).
  - \( S_b = \{(p,i,m) | 1 \leq m < p \land m < n \} \) and \( S_b = \{(p,i,m) | 1 \leq m < p \land 1 \leq m < p \} \)
  - \( | S_a | = n(n+1)/2 \)
  - \( | S_b | = n(n-1)/2 + 2 \)

- Situation 2:
  - \( b \): all the paths containing \( i \) in the middle of the path are re-revised.
  - The paths \((i,i,i)\) and \((k,i,k)\) are not revised again.
  - \( S_a = \{(p,i,m) | 1 \leq m < p \land 1 \leq p < n \} \) and \( S_b = \{(i,j),(k,k)\} \)
  - \( | S_a | = n^2(n-1)/2 \)

- Situation 3:
  - \( c \): all the paths containing \((i,j,m)\) are re-revised.
  - The paths \((i,j,j)\), \((i,i,j)\) are not revised.
  - \( S_a = \{(i,j,m) | 1 \leq m < n \land 1 \leq m < n \} \)
  - \( S_b = \{(i,j,m) | 1 \leq m < n \land 1 \leq m < n \} \)

Other path consistency algorithms

- **PC-3 (Mohr, Henderson 1988)**
  - based on computing supports for a value (like AC-4)
  - this algorithm is not sound!
    - 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 \( D_j \).

- **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 path consistency**

**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 tightness, graph width etc.)

**PC is still not a complete technique**
- A, B, C, D in \([1,2,3]\)
- A \(\neq\) B, A \(\neq\) C, A \(\neq\) D, B \(\neq\) C, B \(\neq\) D, C \(\neq\) D
- is PC but has not solution

**Half way between AC and PC**

Can we make an algorithm:
- stronger than AC,
- without drawbacks of PC (memory consumption, changing the constraint network)?

**Restricted path consistency** (Berlandier 1995)
- based on AC-4 (uses the support sets)
- as soon as a value has only one support in another variable, PC is evoked for this pair of values

**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 valuation of \((k-1)\) different variables can be extended to a consistent valuation of one additional variable.

**Strong k-consistency**

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

Visibly: \(k\)-consistent \(\Rightarrow\) \(k\)-consistency

Moreover: \(j\)-consistent \(\Rightarrow\) \((j+1)\)-consistent for all \(j\)

In general: \(k\)-consistent \(\Rightarrow\) \(j\)-consistent for all \(j\)

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

\[ 1,2,\ldots,n \]
- graph with \(n\) vertices
- domains \(1\ldots(n-1)\)
- It is strongly \(k\)-consistent for \(k>n\) but it has no solution

And what about this graph?

\[ 1,2,\ldots,n \]
- \((D)\)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.

\[ 0,2 \]
- \[ \rightarrow \]
- \[ 1\overline{0}\overline{2} \]
- \[ \overline{1}0\overline{2} \]
- \[ \overline{1}2\overline{0} \]

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 \(\ldots\) 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
Graph width

Ordered graph is a graph with a given total order of vertices. Vertex width in the ordered graph is the number of edges going back from this vertex. Width of the ordered graph is maximum among the width of vertices. Graph width is the maximum among the widths of its ordered graphs.

```plaintext
Graph width is 1.
```

**Procedure MinWidthOrdering(V,E)**

1. Initialize `Q` to an empty set.
2. While `V` is not empty, do the following:
   1. Select and delete node `N` with the smallest number of edges from `(V,E)`
   2. Enqueue `N` to `Q`
3. Return `Q`

Graph width and consistency level

Theorem: Let `w` be the width of the constraint graph. If the constraint graph is strongly `k`-consistent for any `k > w` then there exists an order of variables giving backtrack-free solution.

Proof:
- `w` is a graph width, i.e., there is some ordered graph of this width.
- Thus the max. number of backward edges for each vertex is `w`.
- Let us assign the variables in the order given by this ordered graph.
- Now, if the variable is being labelled:
  - We must find a value compatible with the labelled variables connected with the current variable.
  - Let there is `m` such variables, then `m - w`.
  - The graph is `(m+1)`-consistent, thus a compatible value must exist.

Inverse consistencies

Worst case time and space complexity of `(i,j)`-consistency is exponential in `i`, moreover we need to record forbidden `i`-tuples extensionally (see `PC`).

What about keeping `i=1` and increasing `j`? We already have such an example:
- `RPC` is `(1,1)`-consistency and sometimes `(1,2)`-consistency.

Definition: `(1,k-1)`-consistency is called `k-inverse consistency`.

We remove values of `v` that cannot be consistently extended to additional `(k-1)` variables.

Singleton consistencies

Can we strengthen any consistency technique? YES! Let’s assign a value and make the rest of the problem consistent.

Definition: CSP `P` is singleton `A`-consistent for some notion of `A`-consistency if for every value `h` of any variable `X` the problem `P|X=h` is `A`-consistent.

Features:
- We remove only values from variable’s domain - like `NIC` and `RPC`
- Easy implementation (meta-programming)
- Not so good time complexity (be careful when using `SC`)

1. Singletons `A`-consistent ⇒ `A`-consistent
2. `A`-consistent ⇒ `B`-consistent ⇒ `singleton A`-consistent ≥ `singleton B`-consistent
3. `singleton (i,j)`-consistent > `(i+1,j)`-consistent (`SAC > PIC`)
4. Strong `(i+1,j)`-consistent > `singleton (i,j)`-consistent (`PC > SAC`)

Consistency techniques at glance

NC = 1-consistency
AC = 2-consistency = `(1,1)`-consistency
PC = 3-consistency = `(2,1)`-consistency
PIC = `(1,2)`-consistency
RPC = `(1,1)`-consistency and sometimes `(1,2)`-consistency
SAC = `strong PC`
PIC = `singleton AC`
Foundations of constraint satisfaction

Roman Barták
Charles University in Prague
bartak@ktiml.mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

How to solve the constraint problems?
In addition to local search we have two other 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!

Core search procedure - depth-first search
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

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 variables and the variables connected to it via constraints.

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

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

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

```
procedure AC3-LA(G, cv)
Q := (V, cv) in arcs(Q) \ cv 
consistent := true
while consistent & Q non empty do
    select and delete any arc (V, cv) from Q
    if REVISE(V, cv) then
        Q := Q \ (V, cv) in arcs(Q) \ cv
        consistent := not empty S
    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).

**Constraint optimisation**

So far we have looked for feasible assignments only.
In many cases the users require optimal assignments where optimality is defined by an objective function.

**Definition:** Constraint Satisfaction Optimisation Problem (CSOP) consists of the standard CSP P and an objective function $f$ mapping feasible solutions of P to numbers.

Solution to CSOP is a solution of P minimising / maximising the value of the objective function $f$. To find a solution of CSOP we need in general to explore all the feasible valuations. Thus, the techniques capable to provide all the solutions of CSP are used.

**Variable ordering**

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

**FIRST-FAIL principle**

- 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)

- prefer the variables with more constraints to past 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 supporters
  - a static heuristic that is useful for look-back techniques

**Constraint propagation at glance**

```
Past (already labelled) variables
Future (free) variables
```

- 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).

**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?

- 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
  - SUCCEED FIRST does not go against FIRST-FAIL!
  - prefer the values with more supporters
  - 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)
- prefer the values with more constraints to past variables
- a static heuristic that is useful for look-back techniques

**Comparison of solving methods (4 queens)**

- Backtracking is not very good
  - 19 attempts
- Forward checking is better
  - 3 attempts
- Full look ahead consistency checks
  - And the winner is Look Ahead
  - 2 attempts
**Branch and bound**

Branch and bound is perhaps the most widely used optimisation technique based on cutting sub-trees where there is no optimal (better) solution. It is based on the heuristic function \( h \) that approximates the objective function.

A sound heuristic for minimisation satisfies \( h(x) \leq f(x) \) (in case of maximisation \( f(x) \leq h(x) \)) a function closer to the objective function is better.

During search, the sub-tree is cut if
- there is no feasible solution in the sub-tree
- there is no optimal solution in the sub-tree

\[ \text{bound} \leq h(x), \text{where bound is max. value of feasible solution} \]

How to get the bound?

It could be an objective value of the best solution so far.

---

**Some notes on branch and bound**

Heuristic \( h \) is hidden in propagation through the constraint \( v = f(x) \).

Efficiency is dependent on:
- a good heuristic (good propagation of the objective function)
- a good first feasible solution (a good bound)

The initial bound can be given by the user to filter bad valuations.

The optimal solution can be found fast.

\[ \text{proof of optimality can be long} \text{ (exploring of the rest part of tree)} \]

The optimality is often not required, a good enough solution is OK.

- BB can stop when reach a given limit of the objective function

**Speed-up of BB:** both lower and upper bounds are used

---

**A motivation - robot dressing problem**

Dress a robot using minimal wardrobe and fashion rules. Variables and domains:
- **shirt:** red, white
- **footwear:** cordovans, sneakers
- **trousers:** blue, denim, grey

Constraints:
- shirt \( \times \) trousers: red-grey, white-blue, white-denim
- footwear \( \times \) trousers: sneakers-denim, cordovans-grey
- shirt \( \times \) footwear: white-cordovans

We call the problems where no feasible solution exists over-constrained problems.

---

**Partial constraint satisfaction**

First let us define a **problem space** as a partially ordered set of CSPs \( (P_1,\ldots,P_n) \), where \( P_1 \) \( \subseteq \ldots \subseteq \) \( P_n \) and the solution set of \( P_i \) is a subset of the solution set of \( P_{i+1} \).

The problem space can be obtained by weakening the original problem.

**Partial Constraint Satisfaction Problem (PCSP)** is a quadruple \( (P,(P_1,\ldots,P_n),M,(N,S)) \):
- \( P \) is the original problem
- \( (P_1,\ldots,P_n) \) is a problem space containing \( P \)
- \( M \) is a metric on the problem space defining the problem distance \( M(P,P') \)
- \( (N,S) \) is a pair of a number of different solutions of \( P \) a \( P' \)

We call the problems where no feasible solution exists over-constrained problems.

**Solution to PCSP** is a problem \( P' \) and its solution set satisfying all the constraints.
Second solution of the robot dressing problem

It is possible to assign a preference to each constraint to describe priorities of satisfaction of the constraints. The preference describes a strict priority. A stronger constraint is preferred to arbitrary number of weaker constraints.

shirt x trousers @ required
footwear x trousers @ strong
shirt x footwear @ weak

Constraints marked by a preference make a hierarchy, thus we are speaking about constraint hierarchies.

Third solution of the robot dressing problem

It is possible to assign a value to each constraint to describe the weight of the constraint. The task is to minimise the sum of weights of violated constraints.

shirt x trousers @ 4
footwear x trousers @ 5
shirt x footwear @ 4

This Weighted CSP can be generalised into Valued CSP

Fourth solution of the robot dressing problem

It is possible to assign a preference to each pair (tuple) in the constraint. The task is to maximise the product of preferences for the assignment projections into all constraints.

shirt x trousers: red-grey (1), white-blue (1), white-denim (0.9)
footwear x trousers: sneakers-denim (1), cordovans-grey (1)
shirt x footwear: white-cordovans (0.8)
all other pairs have the value 0.1

This Probabilistic CSP can be generalised into Semiring-based CSP

Constraint hierarchies

Every constraint is labelled by a preference (the set of preferences is totally ordered)
- there is a special preference required, marking constraints that must be satisfied (hard constraints)
- the other constraints are preferential, their satisfaction is not required (soft constraints)

Constraint hierarchy $H$ is a finite (multi)set of labelled constraints.
- $H_r$ is a set of the required constraints (the label is removed)
- $H_s$ is a set of the most preferred soft constraints

A solution to the hierarchy is an assignment satisfying all the required constraints and satisfying best the preferential constraints.

Valued Constraint Satisfaction

Basic idea:
- some valuation is associated to each constraint
- valuations of violated constraints are aggregated
- the assignment with the best aggregated valuation is chosen

Valuation structure $(E, @, P, \ll, T)$, where
- $E$ is a set of valuations totally ordered by $\ll$, with a minimum element $\bot$ and a maximum element $T$
- $@$ is a commutative, associative binary operation on $E$ with the unit element $\bot$ and a maximum element $T$
- $P$ is a commutative, associative binary operation on $E$ with the unit element $\bot$, the absorbing element $T$ and preserving monotonicity ($aPb \Rightarrow aPc \leq bPc$)

Constraints $C$ are mapped to $E$ via function $\phi: C \rightarrow E$

The assignment $A$ with the smallest aggregated valuation $v(A)$ is the solution:

Semiring-based Constraint Satisfaction

Basic idea:
- each tuple in the constraint is marked by a preference level expressing how good the tuple satisfies the constraint
- preference levels of tuple projections are aggregated
- the assignment with the best aggregated valuation is chosen

C-semiring structure $(A, +, \times, 0, 1)$, where
- $A$ is a set of preferences
- $+$ is a commutative, associative, idempotent $(a+a= a)$ binary operation on $A$ with the unit element $0$ and the absorbing element $1$ (1+1=1)
- $\times$ is a commutative, associative binary operation on $A$ with the unit element $1$ and the absorbing element $0$

The assignment $V$ with the largest aggregated preference $p(V)$ is the solution:
### Why should we use CP?

**Close to real-life (combinatorial) problems**
- everyone uses constraints to specify problem properties
- real-life restriction can be naturally described using constraints

**A declarative character**
- concentrate on problem description rather than on solving

**Co-operative problem solving**
- unified framework for integration of various solving techniques
- simple (search) and sophisticated (propagation) techniques

**Semantically pure**
- clean and elegant programming languages
- roots in logic programming

**Applications**
- CP is not another academic framework, it is already used in many applications

### Final notes

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

**Solution technology**
- search combined with constraint propagation
- local search

It is easy to state combinatorial problems in terms of 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.

Constraint Programming is one of the closest approaches to the Holy Grail of programming!

---

“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