3 Foundations of constraint satisfaction

Roman Barták
Charles University in Prague

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

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)
        for each value V in the domain D_X
            if unary constraint on X is inconsistent with V then
                    delete V from D_X
        end for
    end for
end NC
```

Arc consistency (AC)

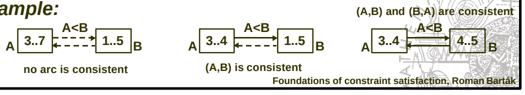
Since now we will assume binary CSP only
i.e. a constraint corresponds to an arc (edge) in the constraint network.

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 valuation $V_i = x$ a $V_j = y$ 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:



Algorithm for 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 valuation $V_i = x$, $V_j = y$ satisfies all the binary constrains on V_i a V_j).

Algorithm of arc revision

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

The procedure also reports the deletion of some value.

Algorithm AC-1 (Mackworth 1977)

How to make CSP arc consistent?

Do revision of every arc.

But this is not enough! Pruning the domain may make some already revised arcs inconsistent again.

A<B, B<C: (3..7,*1..5*,1..5) (*3..4*,1..5,1..5) (3..4,4..5,*1..5*) (3..4,*4*,1..5) (3..4,*4*,5) (3,4,5)

Thus the arc revisions will be repeated until any domain is changed.

Algorithm AC-1

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

What is wrong with AC-1?

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

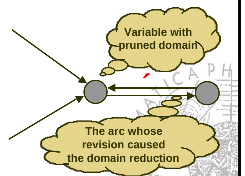
What arcs should be reconsidered for revisions?

The arcs whose consistency is affected by the domain pruning

i.e., the arcs pointing to the changed variable.

We can omit one more arc!

Omit the arc running out of the variable whose domain has been changed (this arc is not affected by the domain change).


Variable with pruned domain
The arc whose revision caused the domain reduction

Algorithm AC-2 (Mackworth 1977)

A generalised version of the Waltz's labelling algorithm.

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

Algorithm AC-2

```
procedure AC-2(G)
    for i ¬ 1 to n do                              % n is a number of variables
        Q ¬ {(i,j) | (i,j) Î arcs(G), j<i}     % arcs for the base revision
        Q' ¬ {(j,i) | (i,j) Î 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' È {(p,k) | (p,k) Î arcs(G), p£i, p¹m }
            end while
            Q ¬ Q'
            Q' ¬ empty
        end while
    end for
end AC-2
```

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-3

```
procedure AC-3(G)
    Q ¬ {(i,j) | (i,j) Î arcs(G), i¹j}     % queue of arcs for revision
    while Q non empty do
        select and delete (k,m) from Q
        if REVISE((k,m)) then
            Q ¬ Q È {(i,k) | (i,k) Î arcs(G), i¹k, i¹m}
        end if
    end while
end AC-3
```

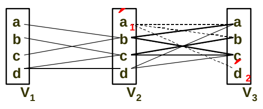
AC-3 is the most widely used consistency algorithm but it is still not optimal.

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_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 Î D_i is the set {<j,b> | b Î D_j , (a,b) Î $C_{i,j}$}

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 supporters are kept.

Computing and counting supporters

```
procedure INITIALIZE(G)
    Q ¬ {} , S ¬ {}                          % emptying the data structures
    for each arc (Vi,Vj) in arcs(G) do
        for each a in Di do
            total ¬ 0
            for each b in Dj do
                if (a,b) is consistent according to the constraint Ci,j then
                        total ¬ total + 1
                        Sj,b ¬ Sj,b È {<i,a>}
                end if
            end for
            counter[(i,j),a] ¬ total
            if counter[(i,j),a] = 0 then
                delete a from Di
                Q ¬ Q È {<i,a>}
            end if
        end for
    end for
    return Q
end INITIALIZE
```

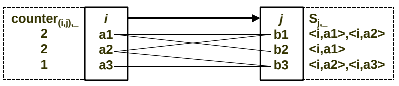
Sj,b - a set of pairs <i,a> such that <j,b> supports them

counter[(i,j),a] - number of supports for the value *a* from Di in the variable Vj

Computing supports and how to use them

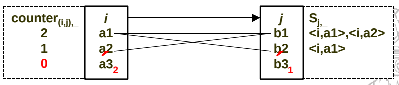
Situation:

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



Using the support sets:

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

Algorithm AC-4

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

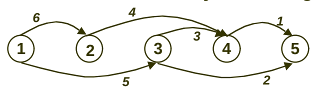
Again, every arc has to be revised, but revision in one direction is enough now.

Algorithm DAC-1

1) Consistency of the arc is required just in one direction.

2) Variables are ordered

➥ there is no directed cycle in the graph!



If the arc 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
```

How to use DAC

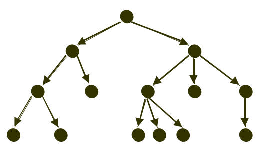
AC visibly covers DAC (if CSP is AC then it is DAC as well)

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?

How to order the vertices for search?



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.

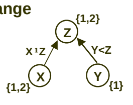
Is it possible to achieve AC by applying DAC in both primal and reverse direction?

In general NO, but …

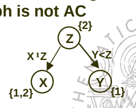
Example:

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



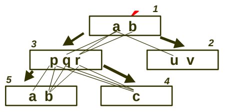
However if the order Z,Y,X is used then we get AC!

From DAC to AC for tree-structured CSP

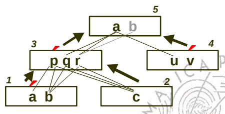
If we apply DAC to tree-structured CSP first using the order from the root to the leaf nodes and second in the reverse direction then we get (full) arc consistency.

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

Is arc consistency enough?

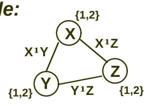
By using AC we can remove many incompatible values

– Do we get a solution?

– Do we know that there exists a solution?

Unfortunately, the answer to both above questions is NO!

Example:



CSP is arc consistent
but there is no solution

So what is the benefit of AC?

Sometimes we have a solution after AC

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

In general, AC prunes the search space.

Consistency techniques in practice

N-ary constraints are processed directly!

The constraint C_Y is arc consistent iff for every variable *i* constrained by C_Y and for every value v Î D_i there is an assignment of the remaining variables in C_Y such that the constraint is satisfied.

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

Constraint semantics is used!

Interval consistency

working with intervals rather than with individual values

interval arithmetic

Example: after change of A we compute A+B ® C, C-A ® B

bounded 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!

Base propagation algorithm

Based on generalisation of AC-3.

Repeat constraint revisions until any domain is changed.

```
procedure AC-3(C)
    Q ¬ C                  % a list of constraints for revision
    while Q non empty do
        select and delete c from Q
        REVISE(c,Q)
    end while
end AC-3
```

The REVISE procedure is customised for each constraint.

we get algorithms with various consistency levels

Constraint planning

How to choose the order of constraints for revisions (a queue Q)?

Event driven programming

event = domain change

REVISE generates new events that evoke further filtering

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<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<B

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

Definition of a constraint (SICStus Prolog)

How to describe propagation through A<B?

bound consistency is enough for full consistency!

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

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

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