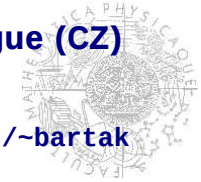
3

Programming with Logic and Constraints

Roman Barták
Charles University, Prague (CZ)

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



Unification?

Recall:

```
?-3=1+2.
no
?-X=1+2
X=1+2;
no
?-3=X+1
no
```

What is the problem?

Term has no meaning (even if it consists of numbers), it is just a syntactic structure!

We would like to have:

```
?-X=1+2.
X=3

?-3=X+1.
X=2

?-3=X+Y,Y=2.
X=1

?-3=X+Y,Y>=2,X>=1.
X=1
Y=2
```

Constraints

- We can go **from unification** (a syntactic equality over terms) **to constraint satisfaction**.
- **Constraint is a relation** (so it has a semantics).
  - relation is a subset of the Cartesian product of domains of constrained variables
  - domain is a set of possible values for the variable

Constraint satisfaction

- For each variable we define its **domain.**
  - we will be using discrete finite domains only
  - such domains can be mapped to integers
- We define **constraints / relations** between the variables.

```
?-domain([X,Y],0,100),3#=X+Y,Y#>=2,X#>=1.
```

- This is called a **constraint satisfaction problem.**
- We want the system to find the values for the variables in such a way that all the constraints are satisfied.

```
X=1, Y=2
```

Domain filtering

How is constraint satisfaction realized?

- For each variable the system keeps its actual domain.
- When a constraint is added, the inconsistent values are removed from the domain.

Example:

	X	Y
	inf..sup	inf..sup
domain([X,Y],0,100)	0..100	0..100
3#=X+Y	0..3	0..3
Y#>=2	0..1	2..3
X#>=1	1	2

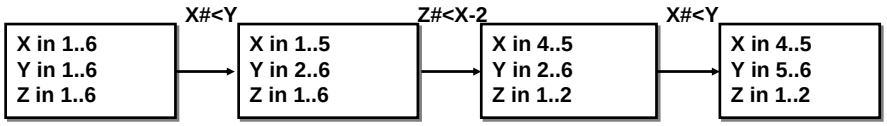
Arc consistency

- We say that a constraint is **arc consistent** (AC) if for any value of the variable in the constraint there exists a value for the other variable(s) in such a way that the constraint is satisfied (we say that the value is supported).
- A **CSP is arc consistent** if all the constraints are arc consistent.

Making problems AC

- How to establish arc consistency in CSP?
- Every constraint must be revised!

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



↳ Doing revision of every constraint just once is not enough!

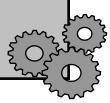
- Revisions must be repeated until any domain is changed (AC-1).

Mackworth (1977)

Algorithm AC-3

- Uses a **queue of constraints** that should be revised.
- When a domain of variable is changed, only the constraints over this variable are added back to the queue for re-revision.

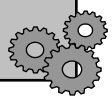
```
procedure AC-3(V,D,C)
    Q ← C
    while non-empty Q do
        select c from Q
        D' ← c.REVISE(D)
        if any domain in D' is empty then return (fail,D')
        Q ← Q ∪ {c'∈C | ∃x∈var(c') D'x≠Dx} – {c}
        D ← D'
    end while
    return (true,D)
end AC-3
```



AC-3 in practice

- Uses a **queue of variables** with changed domains.
  - Users may specify for each constraint when the constraint revision should be done depending on the domain change.
- The algorithm is sometimes called AC-8.

```
procedure AC-8(V,D,C)
    Q ← V
    while non-empty Q do
        select v from Q
        for c∈C such that v is constrained by c do
            D' ← c.REVISE(D)
            if any domain in D' is empty then return (fail,D')
            Q ← Q ∪ {u∈V | D'_u≠D_u}
            D ← D'
        end for
    end while
    return (true,D)
end AC-8
```



Realization

- Constraint solvers typically contain the AC-8 schema realized using event-driven programming (event=domain change).
- Users may add own filtering algorithms for dedicated constraints (REVISE procedure).

Note:

- In CLP, constraints are added incrementally as search proceeds (in Prolog rules) and constraints are removed upon backtracking (domains are restored from the stack in the same way as Prolog variables are restored).

Example (naïve)

SEND+MORE=MONEY

Assign different digits to letters such that SEND+MORE=MONEY holds and S≠0 and M≠0.

Idea:

generate assignments with different digits and check the constraint

```
solve_naive([S,E,N,D,M,O,R,Y]):-
   Digits1_9 = [1,2,3,4,5,6,7,8,9],
   Digits0_9 = [0|Digits1_9],
   member(S, Digits1_9),
   member(E, Digits0_9), E\=S,
   member(N, Digits0_9), N\=S, N\=E,
   member(D, Digits0_9), D\=S, D\=E, D\=N,
   member(M, Digits1_9), M\=S, M\=E, M\=N, M\=D,
   member(O, Digits0_9), O\=S, O\=E, O\=N, O\=D, O\=M,
   member(R, Digits0_9), R\=S, R\=E, R\=N, R\=D, R\=M, R\=O,
   member(Y, Digits0_9), Y\=S, Y\=E, Y\=N, Y\=D, Y\=M, Y\=O, Y\=R,
             1000*S + 100*E + 10*N + D +
             1000*M + 100*O + 10*R + E =:=
   10000*M + 1000*O + 100*N + 10*E + Y.
```

6.8 s

equality of arithmetic expressions

Example (better)

SEND+MORE=MONEY

```
solve_better([S,E,N,D,M,O,R,Y]):-
   Digits1_9 = [1,2,3,4,5,6,7,8,9],
   Digits0_9 = [0|Digits1_9],
   % D+E = 10*P1+Y
   member(D, Digits0_9),
   member(E, Digits0_9), E\=D,
   Y is (D+E) mod 10, Y\=D, Y\=E,
   P1 is (D+E) // 10, % carry bit

   % N+R+P1 = 10*P2+E
   member(N, Digits0_9), N\=D, N\=E, N\=Y,
   R is (10+E-N-P1) mod 10, R\=D, R\=E, R\=Y, R\=N,
   P2 is (N+R+P1) // 10,

   % E+O+P2 = 10*P3+N
   O is (10+N-E-P2) mod 10, O\=D, O\=E, O\=Y, O\=N, O\=R,
   P3 is (E+O+P2) // 10,

   % S+M+P3 = 10*M+O
   member(M, Digits1_9), M\=D, M\=E, M\=Y, M\=N, M\=R, M\=O,
   S is 9*M+O-P3,
   S>0,S<10, S\=D, S\=E, S\=Y, S\=N, S\=R, S\=O, S\=M.
```

Some letters can be computed from other letters and invalidity of the constraint can be checked before all letters are know

0 s

Example (CLP)

SEND + MORE = MONEY

Domain filtering can take care about computing values for letters that depend on other letters.

```
:-use_module(library(clpfd)).
solve(Sol):-
  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_different([S,E,N,D,M,O,R,Y]),
  labeling([],Sol).
```

0 s

assign values (from domains) to variables – depth first search

- Note: It is also possible to use a model with carry bits.

Example

N-queens

Place N queens into a chessboard of size NxN in such a way that no two queens attack each other

```
queensBT(N,Queens):-
   length(Queens,N),
   gen_list(1,N,Positions),
   gen_queens(Queens,[],Positions).

gen_queens([],_,_).
gen_queens([Q|Rest],Assigned,Positions):-
   member(Q,Positions),
   no_attack(Assigned,Q,1),
   gen_queens(Rest,[Q|Assigned],Positions).

gen_list(N,N,[N]).
gen_list(I,N,[I|Rest]):-
   I<N, NextI is I+1,
   gen_list(NextI,N,Rest).

no_attack([],_,_).
no_attack([Q1|Rest],Q,Dist):-
   Q1\=Q, Q1+Dist=\=Q, Q1-Dist=\=Q,
   NextDist is Dist+1,
   no_attack(Rest,Q,NextDist).
```

20 queens=11 s

```
queensCLP(N,Queens):-
   length(Queens,N),
   domain(Queens,1,N),
   all_different(Queens),
   constraint_all(Queens),
   labeling([ff],Queens).

constraint_all([]).
constraint_all([Q|Qs]):-
   constraint_queens(Qs,Q,1),
   constraint_all(Qs).

constraint_queens([],_,_).
constraint_queens([Q2|Qs],Q1,I):-
   Q1#\=Q2+I,
   Q1#\=Q2-I,
   I1 is I+1,
   constraint_queens(Qs,Q1,I1).
```

20 queens=0.01 s

Backtracking and filtering

Why simple backtracking is so bad in comparison with domain filtering?

- **Backtracking**
  - discovers problems late and hence explores more branches

- **Filtering**
  - prunes wrong branches earlier

Design of filters

The user can often define the code of REVISE procedure (filtering code).

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

Filter example
less then

How to describe propagation through A<B?

Note: bound consistency is enough for full arc consistency!

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

:-multifile clpfd:dispatch_global/4.
clpfd: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]).
clpfd:dispatch_global(b2a(A,B),S,S,Actions):-
   fd_max(A,MaxA), fd_min(B,MinB), fd_max(B,MaxB),
    (MaxA<MinB ->
       Actions = [exit]
   ;   UpperBoundA is MaxB-1,
       Actions = [A in inf..UpperBoundA]).
```

A#<B

Filter example
diff

How to describe propagation through A≠B?

Idea: Constraint is **consistent** if **domains** of both variables contain **at least two values**! Hence, propagation is called only when domain becomes singleton.

```
diff(A,B):-
   fd_global(diff(A,B),no_state,[val(A)]),
   fd_global(diff(B,A),no_state,[val(B)]).

:-multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(diff(X,Y),S,S,Actions):-
   (ground(X) ->
       fd_set(Y,SetY),
       fdset_del_element(SetY,X,NewSetY),
       Actions = [exit, Y in_set NewSetY]
   ;
       Actions = []
   ).
```

A#\=B

Filter example
all-diff

How to ensure that different values are assigned to variables in a list?

Idea: If a value is assigned to a variable then remove this value from domains of all other variables in the list.

all_different(List)

```
all_diff(List):-
   start_all_diff(List,List).
start_all_diff([],_).
start_all_diff([H|T],List):-
   fd_global(all_diff(H,T,List),no_state,[val(H)]),
   start_all_diff(T,List).

:-multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(all_diff(X,Pointer,List),S,S,Actions):-
   (ground(X) -> % value has been assigned to X
        filter_diff(List,X,Pointer, Actions)
   ;
        Actions = []
   ).

filter_diff([],_X,_Pointer, [exit]).
filter_diff([Y|T],X,Pointer, Actions):-
   (T==Pointer -> % identical objects
        Actions = RestActions
   ;
        fd_set(Y,SetY),
        fdset_del_element(SetY,X,NewSetY),
        Actions = [Y in_set NewSetY | RestActions]
   ),!,
   filter_diff(T,X,Pointer, RestActions).
```

Diff vs all-diff

- All-diff among N variables can also be modeled using N.(N-1)/2 diff constraints.
- Which approach is better?
  - **Propagation power**
    - Both models filter exactly the **same values** from domains.
  - **Efficiency**
    - **all-diff is faster** than a set of diff constraints
    - Example:
      completing partial Latin squares of order 20 with 8 pre-filled cells
    - all-diff 0.68s, diff 1.43 s

Latin Square of order N is a N×N matrix filled by values {1,...,N} such that in each row and in each column each element occurs exactly once.
Partial Latin Square has only some cells filled.

4	1	3	2
1	4	2	3
2	3	4	1
3	2	1	4

Think globally

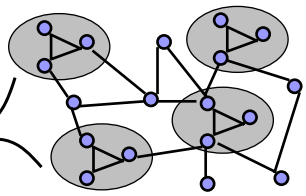
CSP describes the problem locally:

the constraints restrict small sets of variables

\+ heterogeneous real-life constraints

\- missing global view

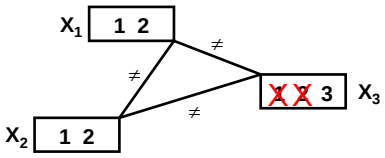
↳ weaker domain filtering



Global constraints

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

Example:

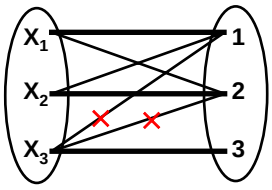


- local (arc) consistency deduces no pruning
- but some values can be removed

Régin (1994)

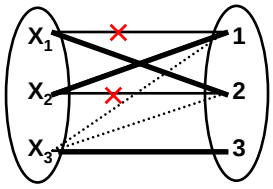
Inside all-distinct

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



Initialisation:

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



Propagation of deletions ($X_1 \neq 1$):

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

Reification

- we can set/find satisfiability of some constraints
- realized via logical constraints (equivalence)
  - **`Constraint #<=> B`**

  Example:
  - **`X#>5 #<=> B`** // the domain of X and B do not change
  - adding **`X#<3`** leads to X in inf..2 and B=0
  - adding **`X#>8`** leads to X in 9..sup and B=1
  - setting **`B=1`** leads to X in 6..sup
  - Only reifiable constraints can participate in logical "meta-constraints" (arithmetic constraints are usually reifiable but most global constraints aren't).

Constraint "exactly"

`exactly(N,List,X)`

- **N** is a FD variable, **List** is a list of FD variables and **X** is a FD variable
- Semantics: exactly N elements from List equals X

implementation via reification:

```
exactly(0, [], _X).
exactly(N, [Y|L], X) :-
  X #= Y #<=> B,
  N #= M+B,
  exactly(M, L, X).
```

Filter "exactly"

`exactly(N,List,X)`

Like all-diff, it is possible to define "exactly" as a single constraint with a dedicated filtering algorithm.

- **Basic idea of the filter:**
  - Undecided ← {Y∈List | Y=X is not decided yet}
  - NumX ← |{Y∈List | Y=X}|
  - max(N)=NumX ⇒ N=NumX & ∀Y∈Undecided Y≠X
  - max(N)<NumX ⇒ fail
  - max(N)>NumX
    - MaxNumX ← NumX+|Undecided|
    - MaxNumX=min(N) ⇒ N=MaxNumX & ∀Y∈Undecided Y=X
    - MaxNumX<min(N) ⇒ fail
    - MaxNumX>min(N) ⇒ N in NumX..MaxNumX
      - NumX<min(N) ⇒ X in domain_union(Undecided)

Homework

- Write a Prolog program for completing a **partial Latin Square**, i.e., finding values for empty cells such that values in each row and in each column are different:
  - **Tips**:
    - think about representation of the matrix, e.g. as a list of list
    - write a procedure for checking whether a given value can be used for a cell
    - use Prolog search to explore alternative filling of cells
- Solve the same problem using Prolog with constraints (CLP).
  - **Tips**:
    - use all-distinct (all-different) constraints to encode the feature of Latin squares
    - procedure for transposition of Matrix might be useful
    - use labeling([ff],ListOfVariables) for value assignment