Linguaggi per problemi di vincoli

SKETCHPAD - Sutherland 1963- Vincoli per grafica
CONSTRAINTS - Sussman e Steele 1980 - circuiti
Linguaggi matematici - algebra simbolica

CLP- Constraint logic programming
famiglia di linguaggi - uno per ogni dominio di vincoli
il constraint solver è built-in nel linguaggio

CONSTRAINT LOGIC PROGRAMMING

Regole estese:

$A :- B_1, B_2, ..., B_n$

B_i può essere o un **atomo** o un **vincolo**.

I vincoli si riferiscono ad un dominio:

- interi
- reali, razionali
- booleani
- domini finiti

Su un insieme di vincoli si possono eseguire operazioni (dipendenti dal dominio) di:

- propagazione di vincoli (semplificazione):
  da X>Y e Y>Z ==> X>Z

- verifica di consistenza (esiste una soluzione?) (Spesso è incompleta)
  X=Y+1, Y>=3, X<=5

Il fattoriale in Prolog:

```
fatt(0,1).
fatt(N,L) :- N1 is N-1, fatt(N1,L1), L is N*L1.
```

Il fattoriale in CLP (Sicstus):

```
:- use_module(library(clpfd)).
fatt(0,1).
fatt(N,L) :- N1 #= N-1, fatt(N1,L1), L #= N*L1.
```

Di solito conviene mettere prima i vincoli:

```
:- use_module(library(clpfd)).
fatt(0,1).
fatt(N,L) :- N1 #= N-1, L #= N*L1, fatt(N1,L1).
```


VALUTAZIONE DI UN GOAL IN CSP

L'interprete fa uso di un **constraint store (CS)**, inizialmente vuoto.

Un passo dell'interprete (nondeterministico):

? G_1, G_2, ..., G_n

- se G_1 è un **atomo**, si procede al solito modo (sostituire G_1 con il corpo di una clausola la cui testa unifica con G_1)

- se G_1 è un **vincolo**:
  - inserire il vincolo nel CS
  - propagare i vincoli nel CS
  - verificare la consistenza del CS
  - se il CS è consistente continuare
    altrimenti fallimento

L'interprete itera finché il goal è vuoto.

La risposta è data dal contenuto del CS, limitato alle variabili presenti nel goal iniziale.

```
1) fatt(0,1).
2) fatt(N,L) :- N1 #= N-1,  L #= N*L1,  fatt(N1,L1).
```

		Constraint Store
?- fatt(N, 6)	regola 2	L=6
?- N1 #= N-1, L #= N*L1, fatt(N1,L1)		N1=N-1
?- L #= N*L1, fatt(N1,L1)		L=N*L1
?- fatt(N1,L1)		


```
1) fatt(0,1).
2) fatt(N,L) :- N1 #= N-1,  L #= N*L1,  fatt(N1,L1).
```

		Constraint Store
?- fatt(N, 6)	regola 2	L=6
?- N1 #= N-1, L #= N*L1, fatt(N1,L1)		N1=N-1
?- L #= N*L1, fatt(N1,L1)		L=N*L1
?- fatt(N1,L1)	regola 1	N1=0, L1=1
?- []		propagazione dei vincoli
		N=1 6=1*L1, 6=1 incons. BACKTRACKING

```
1) fatt(0,1).
2) fatt(N,L) :- N1 #= N-1,  L #= N*L1,  fatt(N1,L1).
```

	Constraint Store
?- fatt(N, 6) regola 2	L=6
?- N1 #= N-1, L #= N*L1, fatt(N1,L1)	N1=N-1
?- L #= N*L1, fatt(N1,L1)	L=N*L1
?- fatt(N1,L1) regola 2	N1=N', L1=L'
?- N1'#=N'-1, L'#=N'*L1', fatt(N1',L1')	N1'=N'-1 L'=N'*L1'
?- ...... ?- fatt(N1',L1')	N1'=0, L1'=1
?- []	SUCCESSO N=3

La risposta dell'interprete è data dal contenuto del CS, limitato alle variabili presenti nel goal iniziale.
Queste variabili non sono necessariamente istanziate.

```
:- use_module(library(clpfd)).

prova(X):- X #=< 5, Y #>= 3, X #>= Y.
prova2(X):- X #=< 6, Y #>= 3, X #>= Y, X #\= 4.
```

Il goal

```
?- prova(X)
```

dà risposta:

```
X in 3..5
```

e

```
?- prova2(X)

X in {3} \/ (5..6)
```

La programmazione logica standard come CLP

Dominio: termini
Vincoli: uguaglianza

L'applicazione di una regola crea dei vincoli di uguaglianza nel CS, che vengono via via semplificati.

1) p(W, [b,c], Z)
2) p(W, Z, W) :- q(a, [W], g(Z))

	CS
?- p(a, [b|X], g(Y))	i) W=a
	ii) [b,c]=[b|X]
regola 1	iii) Z=g(Y)
?- []	propagaz. da ii)
	b=b, X=[c] **successo**

La programmazione logica standard come CLP

Dominio: termini
Vincoli: uguaglianza

L'applicazione di una regola crea dei vincoli di uguaglianza nel CS, che vengono via via semplificati.

1) p(W, [b,c], Z)
2) p(W, Z, W) :- q(a, [W], g(Z))

	CS
?- p(a, [b|X], g(Y))	i) W=a
	ii) Z=[b|X]
regola 2	iii) W=g(Y)
?- q(a, [W], g(Z))	propagaz. da i) e iii)
	a=g(Y) **inconsist**.

Come strutturare un programma per risolvere un CSP (Constraint Satisfaction Problem) in Domini Finiti

1) dichiarare il dominio delle variabili
2) inserire i vincoli
3) cercare una soluzione via backtracking, o una soluzione ottima via branch and bound

Metodologia **constrain and generate** - molto più efficiente di *generate and test.*

In Sicstus il punto 1) può essere realizzato con

domain(L,N,M) - le variabili nella lista L hanno come dominio gli interi compresi fra N e M

Non si possono usare valori simbolici.

Il punto 3 può essere realizzato con il predicato

labeling(*opzioni*, *lista_di_variabili*)

che ha successo se si trova un assegnamento di valori alle variabili che soddisfa i vincoli. *opzioni* è una lista di opzioni come ad es.:

leftmost	si seleziona la variabile leftmost
min	si seleziona la variabile con il minimo lower bound
ff	si seleziona la variabile con il dominio più piccolo
up	si esplora il dominio in ordine ascendente
minimize(X)	usare un algoritmo branch and bound per trovare un assegnamento che minimizza X

```
mm([S,E,N,D,M,O,R,Y]) :-
     domain([S,E,N,D,M,O,R,Y], 0, 9),      % step 1
     S#>0, M#>0,
     all_different([S,E,N,D,M,O,R,Y]),     % step 2
     sum(S,E,N,D,M,O,R,Y),
     labeling([], [S,E,N,D,M,O,R,Y]).    % step 3

sum(S, E, N, D, M, O, R, Y) :-
           1000*S + 100*E + 10*N + D +
      1000*M + 100*O + 10*R + E #=
      10000*M + 1000*O + 100*N + 10*E + Y.
```

```
?- mm([S,E,N,D,M,O,R,Y]).
D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2
```


N regine

```
queens(N, L) :-
     length(L, N),
     domain(L, 1, N),
     constrain_all(L),
     labeling([], L).

constrain_all([]).
constrain_all([X|Xs]) :-
     constrain_between(X, Xs, 1),
     constrain_all(Xs).

constrain_between(_X, [], _N).
constrain_between(X, [Y|Ys], N) :-
     no_threat(X, Y, N),
     N1 is N+1,
     constrain_between(X, Ys, N1).

no_threat(X, Y, I) :-
        X   #\= Y,
        X+I #\= Y,
        X-I #\= Y.
```

DOMINI REALI

```
:- use_module(library(clpr)).

circuit(resistor(R), V, I) :- {V = I*R}.

circuit(series(N1,N2), V, I) :-
        {I=I1},
        {I=I2},
        {V = V1+V2},
        circuit(N1, V1, I1),
        circuit(N2, V2, I2).

circuit(parallel(N1,N2), V, I) :-
        {V=V1},
        {V=V2},
        {I = I1+I2},
        circuit(N1, V1, I1),
        circuit(N2, V2, I2).
```



```
?- circuit(parallel(resistor(0.4),
           series(resistor(0.2),resistor(0.1))), 12, I).

I = 70.0 ?

yes


?- circuit(parallel(resistor(0.4),
           series(resistor(0.2),resistor(0.1))), V, I).

{I=5.83333333333333*V} ?

yes
```

Vincoli su numeri reali

```
mutuo(P,T,I,R,B):-
        {T=0},
        {B=P}.
mutuo(P,T,I,R,B):-
        {T>=1},
        {NT=T-1},
        {NP=P+P*I-R},
        mutuo(NP,NT,I,R,B).
```

P: ammontare del prestito

T: numero rate

I: interesse

R: ammontare rata

B: prestito rimanente al termine

?- mutuo(10000, 10, 10/100, R, 0).
R = 1627.454

?- mutuo(P, 15, 10/100, 1000, 0).
P = 7606.0795

?- mutuo(P, 10, 10/100, R, 0).
R = 0.1627 * P

Vincoli booleani - FULL ADDER

```
:- use_module(library(clpb)).

full_adder(X,Y,Cin, Sum,Cout) :-
    sat((U1 =:= X * Cin) *
        (U2 =:= Y * U3) *
        (Cout =:= U1 + U2) *
        (U3 =:= X # Cin) *
        (Sum =:= Y # U3)).
```

* AND
\+ OR
\# OR ESCLUSIVO

```
?- full_adder(1,1,0,X,Y).
X = 0,
Y = 1

?- full_adder(1,1,1,X,Y).
X = 1,
Y = 1

?- full_adder(a,b,0,X,Y).
sat(Y=:=a*b),
sat(X=:=a#b)
```

FAULT DETECTION

Al massimo un componente non funzionante

```
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
        sat(card([0-1],[F1,F2,F3,F4,F5]) *
            (F1 + (U1 =:= X * Cin)) *
            (F2 + (U2 =:= Y * U3)) *
            (F3 + (Cout =:= U1 + U2)) *
            (F4 + (U3 =:= X # Cin)) *
             (F5 + (Sum =:= Y # U3))).


?- fault(L,[1,1,0], [1,0]), labeling(L).

L = [0,0,0,1,0]

?- fault(L,[1,1,0], [1,1]), labeling(L).

L = [0,0,0,0,1]
```


DOMINI FINITI: SCHEDULING

```
:- use_module(library(clpfd)).

schedule([S1,S2,S3,S4]) :-
  domain([S1,S2,S3,S4], 1, 20),
  (S1 #= 1) #\/ (S3 #= 1),
  S1+5 #=< S2,
  S3+8 #=< S4,
  mutex(S1,5,S3,8),
  mutex(S2,3,S4,2),
  labeling([], [S1,S2,S3,S4]).
```

	durata
task1	5
task2	3
task3	8
task4	2

task1 prima di **task2**
task3 prima di **task4**

task1 e **task3** mutuamente esclusivi
task2 e **task4** mutuamente esclusivi

```
mutex(S1,D1,S2,D2) :- (S1+D1 #=< S2) #\/ (S2+D2 #=< S1).

    ?- schedule(L).
    L = [1,6,6,14] ? ;
    L = [1,6,6,15] ? ;
    ..............
    L = [9,14,1,10] ? ;
    L = [9,14,1,11] ?
```

Trovare la soluzione ottima

```
schedule([S1,S2,S3,S4], Cost) :-
       domain([S1,S2,S3,S4], 1, 20),
       S1+5 #=< S2,
       S3+8 #=< S4,
       mutex(S1,5,S3,8),
       mutex(S2,3,S4,2),
       Cost #= max(S4+2, S2+3),
       labeling([minimize(Cost)], [S1,S2,S3,S4]).

mutex(S1,D1,S2,D2) :- (S1+D1 #=< S2) #\/ (S2+D2 #=< S1).



?- schedule(L,C).

C = 16,
L = [1,6,6,14]
```