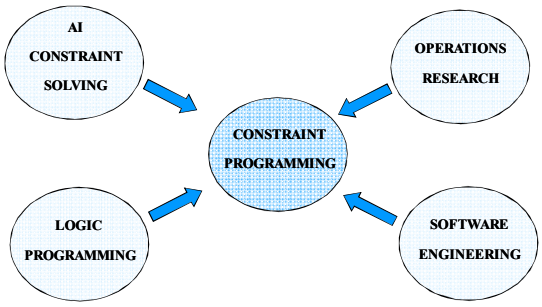
PROGRAMMAZIONE LOGICA A VINCOLI



PROGRAMMAZIONE LOGICA A VINCOLI

- Problemi di soddisfacimento di vincoli:
  - concetti generali
- Programmazione Logica
  - vantaggi
  - limiti
- Programmazione Logica a Vincoli
  - dominio e interpretazione
  - controllo
  - modello computazionale

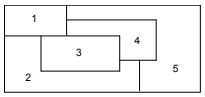
PROBLEMI DI SODDISFACIMENTO DI VINCOLI

- Un problema di soddisfacimento di vincoli è definito da:
  - *un insieme di variabili* (V_1, $V_2 \ldots V_n$)
  - *un dominio discreto per ogni variabile* ($D_1, D_2 \ldots D_n$)
  - *un insieme di vincoli su queste variabili*:
    vincolo: una relazione tra variabili che definisce un sottoinsieme del prodotto cartesiano dei domini $D_1 x D_2 x \ldots x D_n$

Soluzione di un Problema di Soddisfacimento di vincoli: un assegnamento di valori alle variabili consistente con i vincoli

E. Tsang: "Foundations of Constraint Satisfaction" Academic Press, 1992.

ESEMPIO: Map Coloring



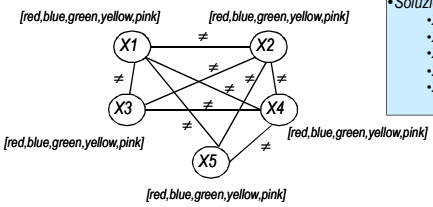
• Trovare un assegnamento di colori alle variabili consistente con i vincoli

- *variabili X1, X2, X3, X4, X5: zone*
- *domini D1, D2, D3, D4, D5: [red, blue, green, yellow,pink]*
- *vincoli : near(Xi, Xj)* $\Rightarrow Xi \neq Xj$

CONSTRAINT GRAPHS

Un problema di soddisfacimento di vincoli si può rappresentare con un grafo detto constraint graph:

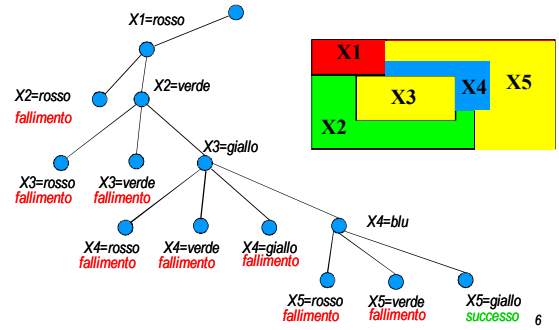
- *variabili* ⟺ *nodi*
- *vincoli* ⟺ *(iper)-archi*



- *Soluzione ammissibile:*
  - *X1 = red*
  - *X2 = green*
  - *X3 = blue*
  - *X4 = yellow*
  - *X5 = pink*

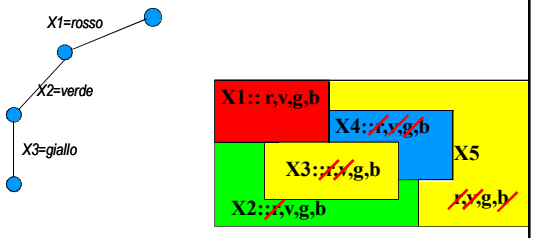
ESEMPIO: Map Coloring

- Algoritmo semplice (ma a volte inefficiente)



PROPAGAZIONE DI VINCOLI

- Eliminazione a priori dei valori inconsistenti



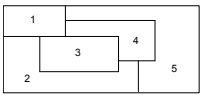
PROBLEMI DI OTTIMIZZAZIONE

Un problema di ottimizzazione è definito da:

- *un insieme di variabili* (X_1, $X_2 \ldots X_n$)
- *un dominio discreto per ogni variabile* ($D_1, D_2 \ldots D_n$)
- *un insieme di vincoli su queste variabili*:
  vincolo: una relazione tra variabili che definisce un sottoinsieme del prodotto cartesiano dei domini $D_1 x D_2 x \ldots x D_n$
- *una funzione obiettivo* $f(X_1, X_2 \ldots X_n)$

Soluzione di un problema di ottimizzazione: un assegnamento di valori alle variabili compatibile con i vincoli del problema che ottimizza la funzione obiettivo

EXAMPLE: Map Coloring



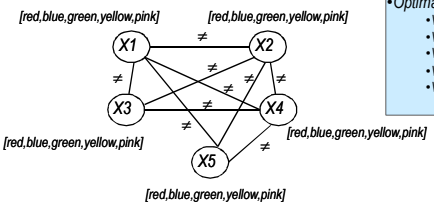
• Trovare un assegnamento di colori alle zone tale che due zone adiacenti sono colorate con colori diversi, e MINIMIZZANDO il numero di colori usati

– *variabili V1, V2, V3, V4, V5: zone*
– *domini D1, D2, D3, D4, D5: [red, blue, green, yellow,pink]*
– *vincoli: near(Xi, Vj) ⇒ Xi ≠ Xj*

CONSTRAINT GRAPHS

Un problema di ottimizzazione si può rappresentare con un grafo detto constraint graph:

– *variabili* ⟸ *nodi*
– *vincoli* ⟸ *(iper)-archi*



•*Optimal Solution:*
- *V1 = red*
- *V2 = green*
- *V3 = yellow*
- *V4 = blue*
- *V5 = yellow*

“Prolog è un buon linguaggio per risolvere CSP?”

Prolog per CSP

Vantaggi

- *Linguaggio dichiarativo*
- *Variabili logiche*
- *Backtracking*
- *Reversibilità: molti predicati per generare combinazioni*

Svantaggi

- *Integrazione con altro software?*

Es reversibilità

```
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```

- *verifica se un elemento appartiene ad una lista*

```
member(1,[4,1,2]).
yes
```

- *Se metto l'elemento variabile, in backtracking gli vengono assegnati i vari elementi della lista*

```
member(X,[4,1,2]).
yes, X=4, more?
yes, X=1, more?
yes, X=2.
```

- *Utile per generare assegnamenti!*

Esempio di CSP

- *Variabili: X, Y - Domini: da 1 a 4 - Vincoli: X>Y*

```
csp(X,Y):-
  member(X,[1,2,3,4]),
  member(Y,[1,2,3,4]),
  X>Y.
```

- *Elenca, in backtracking, tutte le soluzioni del CSP*
- *Molto dichiarativo: dichiaro le variabili, i domini (member) e i vincoli*
- *Quale algoritmo viene usato?*
- *Che cosa devo fare se voglio cambiare euristica?*
  - *selezione del valore*
  - *selezione della variabile*

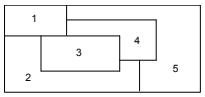
Esempio Reversibilità 2

```
permutation([],[]).
permutation([Head|Tail],PermList) :-
  permutation(Tail,PermTail),
  delete(Head,PermList,PermTail).
delete(A, [A|B], B).
delete(A, [B, C|D], [B|E]) :-
  delete(A, [C|D], E).
```

Può generare le permutazioni di una lista

```
?- permutation([a,b,c],L).
Yes, L = [a, b, c]      more? ;
Yes, L = [b, a, c]      more? ;
Yes, L = [b, c, a]      more?
```


ESEMPIO: Map Coloring

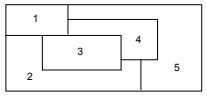


• Trovare un assegnamento di colori alle variabili consistente con i vincoli

– *variabili V1, V2, V3, V4, V5: zone*
– *domini D1, D2, D3, D4, D5: [red, blue, green, yellow]*
– *vincoli : near(Vi, Vj) ⇒ Vi ≠ Vj*

Esempio

```
coloring([X1,X2,X3,X4,X5],Dom):-
    member(X1,Dom),
    member(X2,Dom),
    member(X3,Dom),
    member(X4,Dom),
    member(X5,Dom),
    X2 \= X1, X3 \= X1,
    X4 \= X1, X5 \= X1,
    X3 \= X2, X4 \= X2,
    X5 \= X2, X4 \= X3,
    X4 \= X5.
```

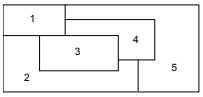


Molto dichiarativo!

Che algoritmo viene usato?

Standard backtracking

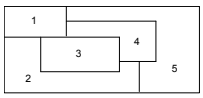
```
coloring([X1,X2,X3,X4,X5],Dom):-
    member(X1,Dom),
    member(X2,Dom),
    X2 \= X1,
    member(X3,Dom),
    X3 \= X1, X3 \= X2,
    member(X4,Dom),
    X4 \= X1, X4 \= X2, X4 \= X3,
    member(X5,Dom),
    X5 \= X1,
    X5 \= X2, X4 \= X5.
```



- L'ordine è importante
- un po' meno dichiarativo
- Risolve solo questa istanza

Standard backtracking: generale

```
% colori(+LStati, -Assegnam, +NodiGiaAssegnati,
  +ValoriGiaAssegnati, Dom)
% ?- colori([1,2,3,4,5], X,[],[],[r,b,g,y]).
colori([],[],_,_,_).
colori([N1|Ntail], [X|Tail], NPlaced, Placed, Values):-
  member(X,Values),
  compatible(X,N1,Placed,NPlaced),
  colori(Ntail, Tail, [N1|NPlaced], [X|Placed], Values).

compatible(_,_,[],[]).
compatible(X,N1,[P|Tail],[NP|NTail]):-
  (near(N1,NP); near(NP,N1)), !,
  X\==P,
  compatible(X,N1,Tail,NTail).
compatible(X,N1,[P|Tail],[NP|NTail]):-
  compatible(X,N1,Tail,NTail).
```




```
near(1,2).
near(1,3).
...
```


Altro esempio: N-queens

```
queens(S,N):-
   domains(1,N,D),
   permutation(D,S),
   safe(S).

safe([]).
safe([Queen|Others]) :-
 safe(Others),
 noattack(Queen,Others,1).
```

```
domains(X,X,[X]):-!.
domains(N,Max,[N|T]):-
     N1 is N+1,
     domains(N1,Max,T).

noattack(_,[],_).
noattack(Y,[Y1|Ylist],Xdist):-
     Y-Y1=\=Xdist,
     Y1-Y=\=Xdist,
     Dist1 is Xdist +1,
     noattack(Y,Ylist,Dist1).
```

Che algoritmo è?

N-Queens Standard Backtracking

```
stdback(X,N):-
   domains(1,N,D),
   stdback(X,[],D).

stdback([], Placed, []).
stdback([X|Xs], Placed, Values):-
   delete(X,Values,NewValues),
   noattack(X,Placed,1),
   stdback(Xs,[X|Placed],NewValues).
```

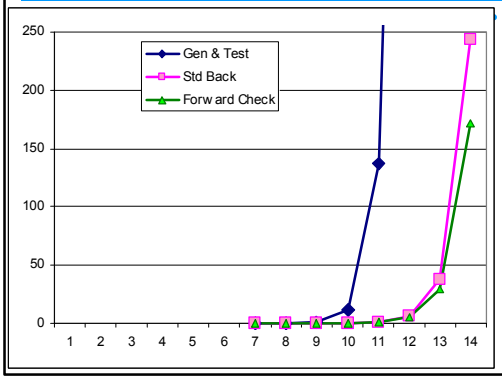
```
domains(X,X,[X]):-!.
domains(N,Max,[N|T]):-
     N1 is N+1,
     domains(N1,Max,T).

noattack(_,[],_).
noattack(Y,[Y1|Ylist],Xdist)
   :-
     Y-Y1=\=Xdist,
     Y1-Y=\=Xdist,
     Dist1 is Xdist +1,
     noattack(Y,Ylist,Dist1).
```


N-Queens Forward Checking

```
fwdcheck(Var, N):-
    domains(1,N,D),
    length(Var,N),
    assegna_dom(VarDom,D,Var),
    queens_aux(VarDom).
queens_aux([]).
queens_aux([[X1,D]|Rest]):-
    member(X1,D), % istanzia X1
    forward(X1,Rest,Newrest),
    %propagazione
    queens_aux(Newrest).
forward(X,Rest,Newrest):-
    forward(X,Rest,1,Newrest).
forward(X,[],Nb,[]).
forward(X,[[Var,Dom]|Rest],Nb,[[Var,[
   F|T]]|Newrest]):-
    remove_value(X,Dom,Nb,[F|T]),
    Nb1 is  Nb +1,
    forward(X,Rest,Nb1,Newrest).
```

```
remove_value(X,[],Nb,[]).
remove_value(X,[Val|Rest],Nb,[Val|New
   rest]):-
    compatible(X,Val,Nb), !,
    remove_value(X,Rest,Nb,Newrest).
remove_value(X,[Val|Rest],Nb,Newrest)
   :-
    remove_value(X,Rest,Nb,Newrest).
domains(X,X,[X]):-!.
domains(N,Max,[N|T]):-
     N1 is N+1,
     domains(N1,Max,T).
compatible(Value1,Value2,Nb):-
    Value1 =\= Value2 +Nb,
    Value1 =\= Value2 - Nb,
    Value1 =\= Value2.
% Crea una lista
   [[Var1,Dom],[Var2,Dom],…]
assegna_dom([],_,[]).
assegna_dom([[V,D]|LVarDom],D,[V|LVar
   ]):-
    assegna_dom(LVarDom,D,LVar).
```


N-Queens: Efficienza



secondi necessari per trovare tutte le soluzioni

Limite n. 1

Prolog spinge a scrivere programmi basati su Standard Backtracking

meno efficienti di algoritmi che applicano il pruning a priori

Rappresentazione dei numeri

- *Risolvere questa equazione:*

$$X+2 = Y+1$$

- *Soluzione:*

$$X+1 = Y$$

- *Possiamo ottenere questo risultato in Prolog?*

Aritmetica di Peano

```
sum(X,0,X).
sum(X,s(Y),s(Z)):-
  sum(X,Y,Z).

?- sum(X,s(s(0)),Z), sum(Y,s(0),Z).
yes, Y=s(X).
```


Possiamo scrivere così?

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


E così?

```
?- Z is X+2, Z is Y+1.
```

Ordine dei goal significativo

- *X is Y+1, Y=3* → *errore*
- *Y=3, X is Y+1* → *X=4*

Lo stesso vale per le relazioni (>, <, ≠, ≥, …):

- *X > 3, X = 5* → *errore*
- *X = 5, X > 3* → *yes*

Limite n. 2

Prolog non interpreta i numeri

Soluzione parziale: freeze, when

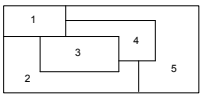
`freeze(X,atomo(X))`

- *Indica a Prolog che l'*`atomo` *deve essere selezionato (dalla risoluzione SLD) solo quando* `X` *non è variabile.*

```
freeze(Y, X is Y+1), Y=3.
        Y/3 |
freeze(3, X is 3+1).
        X/4 |
          []
```


Map Coloring usando `when`

```
diverso(A,B):-
  when((nonvar(A),nonvar(B)),A\=B).
coloring([X1,X2,X3,X4,X5],Dom):-
  diverso(X2,X1), diverso(X3,X1),
  diverso(X4,X1), diverso(X5,X1),
  diverso(X3,X2), diverso(X4,X2),
  diverso(X5,X2), diverso(X4,X3),
  diverso(X4,X5),
  member(X1,Dom),
  member(X2,Dom),
  member(X3,Dom),
  member(X4,Dom),
  member(X5,Dom).
```



Molto dichiarativo! Che algoritmo viene usato?

Ancora su freeze / when

- *E se Y non diventa ground?*

risposta condizionale!

```
?- freeze(Y, X is Y+1).
Delayed goals: X is Y + 1
Yes
```

- *Non semplifica le equazioni, ma è già qualcosa*
- *A carico del programmatore*
- ***Cosa ci serve ?***
  - *Che il risolutore "addormenti" i goal che non è in grado di valutare per selezionarli non appena sono disponibili nuove informazioni*

Altro esempio di CSP

- *Variabili: X, Y - Domini: da 1 a 4 - Vincoli: X>Y*

```
csp(X,Y):-
   when((nonvar(X),nonvar(Y)), X>Y ),
   member(X,[1,2,3,4]),
   member(Y,[1,2,3,4]).
```

- *Prima fa gli assegnamenti, poi verifica i vincoli*
- *Sarebbe più efficiente se, quando invoco X>Y, questo eliminasse già a priori gli elementi inconsistenti dai domini: pruning*

Come fare?

- *Se vogliamo che Prolog faccia anche le semplificazioni, il pruning, bisogna che sappia il tipo della variabile:*
  - *alcune variabili non sono solo variabili logiche (a cui può essere assegnato un termine), ma numeriche*
  - *Devo dire qual è il dominio della variabile*
  - *A questo punto il sistema associa ad ogni variabile il suo dominio e lavora su di esso*

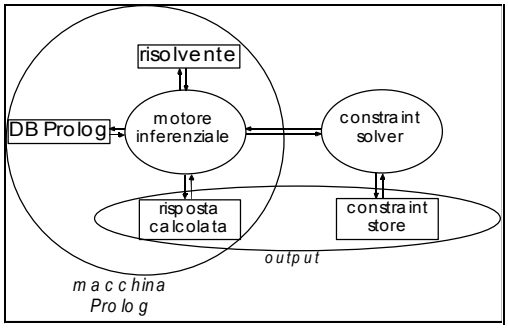
Programmazione Logica a Vincoli

Constraint Logic Programming (CLP)

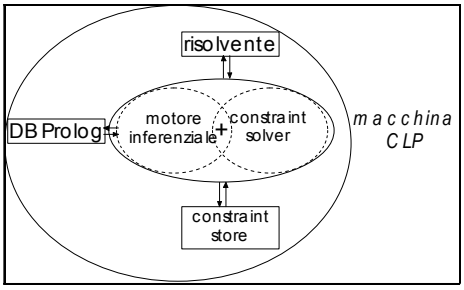
Constraint Logic Programming

- *Nuovo paradigma di programmazione, estensione della Programmazione Logica [Jaffar e Lassez, 1998]*
- *è uno schema per produrre nuovi linguaggi*
- *durante la risoluzione SLD, alcuni atomi speciali, detti vincoli (constraints), non vengono risolti, ma vengono accumulati in un'area di memoria esterna (constraint store) ed elaborati da un risolutore esterno (constraint solver).*
- *Ci possono essere diversi risolutori, basati su tecnologie diverse. Ciascuno di questi dà luogo ad un linguaggio della classe CLP:*
  - *CLP(FD): sui domini finiti basato su consistency*
  - *CLP(R): sui reali basato sul simplesso*
  - *CLP(Bool): sui booleani basato su BDD*
  - *…*

ESTENDERE LA MACCHINA LOCIGA



MACCHINA CLP



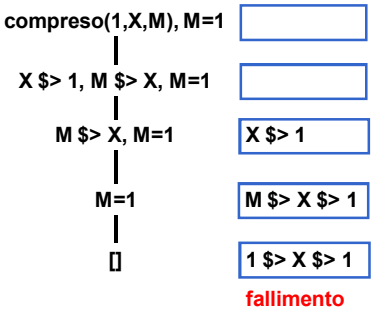
CLP: sintassi

- *In CLP ci sono alcuni predicati, con una sintassi particolare, che non vengono risolti con la risoluzione SLD*
- *Ad esempio, supponiamo che il nostro linguaggio CLP abbia il vincolo* **$>**, *che ha il significato di "maggiore"*

```
compreso(Inf,X,Sup) :- X $> Inf,
  Sup $> X.
```

CLP: semantica operazionale

```
compreso(Inf,X,Sup) :- X $> Inf, Sup $> X.
```



Come fa il constraint solver?

- *Da cosa si accorge il constraint solver che c'è un fallimento?*
- *Dipende dal tipo di solver*
  - *CLP(FD): ho un fallimento quando una variabile ha il dominio vuoto*
  - *CLP(R): ho fallimento quando l'algoritmo del simplesso rileva che non ci sono soluzioni*

MACCHINA CLP: PASSI FONDAMENTALI

- ***Risoluzione r***
  - *selezione di un atomo dal risolvente*
  - *unificazione: aggiunta di vincoli al constraint store*
- ***Constraining c***
  - *selezione di un vincolo dal risolvente*
  - *aggiunta del vincolo al constraint store*
- ***Infer (propagazione) i***
  - *trasformazione del constraint store*
- ***Consistenza s***
  - *verifica della soddisfacibilità del constraint store*

"metodi" che il risolutore deve esportare per poter essere usato in una macchina CLP

CLP(FD)

- ***CLP(FD): Constraint Logic Programming su domini finiti***
  - particolarmente adatta a modellare e risolvere problemi a vincoli
- *Modello del problema*
  - *Le VARIABILI rappresentano le entità del problema*
  - *Definite su DOMINI FINITI di oggetti arbitrari (normalmente interi)*
  - *Legate da VINCOLI (relazioni tra variabili)*
    - *matematici*
    - *simbolici*
  - *Nei problemi di ottimizzazione si ha una FUNZIONE OBIETTIVO*
- *Risoluzione*
  - *Algoritmi di propagazione (incompleti) incapsulati nei vincoli*
  - *Strategie di ricerca*

CLP(FD): Sintassi

- Esistono diversi linguaggi CLP(FD).
  - CHIP
  - SICStus
  - ECLiPSe
  - B-Prolog
  - …
- La sintassi è simile, ma non identica
- Ciascuno ha caratteristiche distintive

CLP(FD): Sintassi

In generale però si hanno

- Un metodo per definire il dominio delle variabili
  - ECLiPSe: `X :: [1..10,13..15].`
  - SICStus: `X in (1..10) \/ (13..15).`
- Una sintassi che contraddistingue i vincoli dagli altri predicati (ECLiPSe e SICStus usano il '#'):
  - `#>, #>=, #=<, #=, #\=`, … sono vincoli

CLP(FD): Semantica Operazionale

- Quando il letterale selezionato dalla risoluzione SLD è un vincolo, questo viene inserito nel constraint store
- A questo punto, si ha una fase di inferenza: in CLP(FD) questa fase è data da una propagazione, tipicamente Arc-Consistency
- Fase di Consistenza: se uno dei domini risulta vuoto, si ha fallimento
- Alla fine viene fornito il constraint store come risposta, insieme al binding

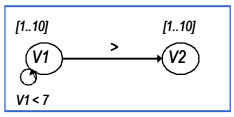
PROPRIETA' DI CONSISTENZA

• NODE CONSISTENCY

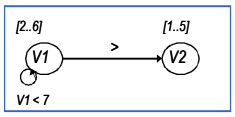
– una rete è node consistent se in ogni dominio di ogni nodo ogni valore è consistente con i vincoli unari che coinvolgono la variabile

• ARC CONSISTENCY

– una rete è arc consistent se per ogni arco (vincolo binario) che connette le variabili Vi e Vj per ogni valore nel dominio di Vi esiste un valore nel dominio di Vj consistente con il vincolo



Non Node consistent
Non Arc consistent



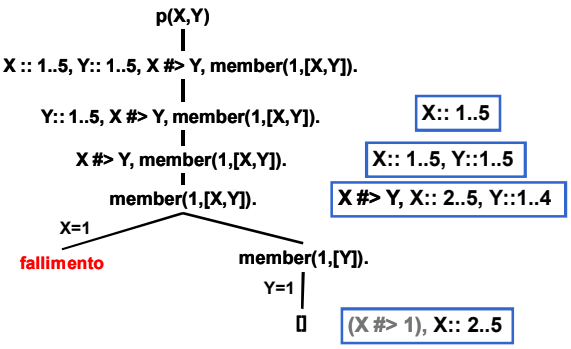
Node consistent
Arc consistent

Esempio:

```
p(X,Y):-
  X :: 1..5, Y :: 1..5,
  X #> Y, member(1,[X,Y]).
```


Esempio:

```
p(X,Y):- X::1..5, Y::1..5, X#>Y,
   member(1,[X,Y]).
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```



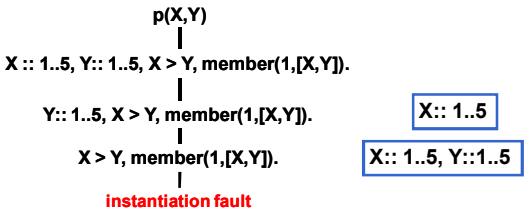
Esempio:

- *Che cosa sarebbe successo se avessi scritto X>Y invece di X #> Y?*

```
p(X,Y):-
  X :: 1..5, Y:: 1..5,
  X > Y, member(1,[X,Y]).
```


Esempio:

```
p(X,Y):- X::1..5, Y::1..5, X>Y,
   member(1,[X,Y]).
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```



- *Il predicato > è sempre il solito predicato di Prolog!!*
- *Quindi vuole avere entrambi gli argomenti istanziati*
- *Può essere usato solo come test, non è un vincolo!*

VINCOLI

In generale, in CLP(FD) si hanno queste tipologie di vincoli:

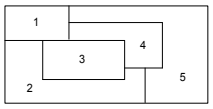
- Vincoli matematici: =, >, <, ≠, ≥, ≤
  - Propagazione: arc-consistency
- Vincoli Simbolici ***[Beldiceanu, Contejean, Math.Comp.Mod. 94]***
  - Incapsulano un metodo di propagazione globale ed efficiente
  - Formulazioni più concise
    - `alldifferent([X1,...,Xm])`
      tutte le variabili devono avere valori differenti
    - `element(N,[X1,...Xm],Value)`
      l'ennesimo elemento della lista deve avere valore uguale a Value
    - `cumulative([S1,...Sm], [D1,...Dn], [R1,...Rn], L)`
      usato per vincoli di capacita'
    - vincoli disgiuntivi

PROPAGAZIONE DI VINCOLI

- Vincoli matematici:
  - Esempio 1
    - `X::[1..10], Y::[5..15], X#>Y`
      Arc-consistency: per ogni valore v della variabile X, se non esiste un valore per Y compatibile con v, allora v viene cancellato dal dominio di X e viceversa
      `X::[6..10], Y::[5..9]` dopo la propagazione
  - Esempio 2
    - `X::[1..10], Y::[5..15], X#=Y`
    - `X::[5..10], Y::[5..10]` dopo la propagazione
  - Esempio 3
    - `X::[1..10], Y::[5..15], X#\=Y`
    - Nessuna propagazione

ESEMPIO DI MODELLO DI UN PROBLEMA

- Map Colouring




```
map_colouring([V1,V2,V3,V4,V5]):-
      V1::[red,green,yellow,blue],
      V2::[red,green,yellow,blue],
      V3::[red,green,yellow,blue],
      V4::[red,green,yellow,blue],
      V5::[red,green,yellow,blue],
      V1#\=V2, V1#\=V3, V1#\=V4, V1#\=V5, V2#\=V3,
      V2#\=V4, V2#\=V5, V3#\=V4, V4#\=V5,
      .....
```

variabili & domini

vincoli

Alternativamente...

```
      alldifferent([V1,V2,V3,V4]),
      alldifferent([V1,V2,V4,V5]).
```


Risolutori (in)completi

- *Problema: l'Arc-Consistency da sola non è in grado di trovare una soluzione*
- *Il risolutore CLP(FD) è un solver incompleto:*
  - *Se dice 'no' non esiste soluzione*
  - *Se dice 'sì' potrebbe esistere o potrebbe non esistere*
  - *Fornisce comunque nella risposta calcolata i vincoli che devono essere soddisfatti*
- *Se vogliamo trovare una soluzione, dovremo fare una ricerca nello spazio degli stati*

CP: PROBLEM SOLVING

- Nozione di consistenza:
  - L'insieme dei vincoli e' consistente ?
  - Esiste una soluzione ?
- Propagazione di vincoli: meccanismo di inferenza
  - Rimuovere dai domini valori inconsistenti
  - Inferire nuovi vincoli
- Ricerca: strategie di branching
  - Selezione di una variabile
  - Selezione del valore

CP: CONSISTENZA

- Un insieme di vincoli è CONSISTENTE (SODDISFACIBILE) se ammette almeno una soluzione
- Risolutori COMPLETI sono in grado di decidere se un insieme di vincoli è soddisfacibile (Es. risolutori sui reali)
- Risolutori INCOMPLETI sono in grado di individuare alcune forme di inconsistenza, ma non di decidere se un insieme di vincoli è soddisfacibile. (Es. Risolutori su domini finiti)
  - l'inconsistenza è identificata quando il dominio di una variabile diventa vuoto.

CP(FD): CONSISTENZA

- *Inconsistenza: dominio di una variabile vuoto*
  - *non esistono valori che possono essere assegnati alla variabile*
- *Se ho una variabile con dominio vuoto sicuramente l'insieme di vincoli è INSODDISFACIBILE*
- *Se l'insieme di vincoli è INSODDISFACIBILE non è detto che una variabile abbia dominio vuoto.*
- *Se riuscissimo a trovare TUTTI i valori inconsistenti e a rimuoverli dal dominio delle variabili, avremmo un risolutore completo. Il problema di trovare tutti i valori inconsistenti ha la stessa complessità del problema originale.*

CP(FD): PROPAGAZIONE

- La propagazione di vincoli è quella forma di inferenza che permette di identificare e rimuovere dai domini i valori inconsistenti
- Riduzione dello spazio di ricerca in cui ogni nodo corrisponde all'istanziazione di una variabile del problema a un valore contenuto nel suo dominio.
- Per ridurre completamente il problema (eliminando tutti i valori inconsistenti) la propagazione avrebbe complessità esponenziale.
- Allora è necessario trovare un compromesso tra costo computazionale della propagazione e riduzione dello spazio di ricerca.

ECLiPSe

- *ECLiPSe è un linguaggio CLP che incorpora varie librerie, che forniscono dei risolutori specifici*
  - *fd* *sui domini finiti*
  - *fd_set* *sugli insiemi*
  - *eplex* *reali, vincoli lineari*
  - *…*
- *Inoltre, è possibile definire nuovi risolutori e vincoli*
  - *propia*
  - *CHR*
  - *…*

ECLiPSe: CLP(FD)

- *La libreria FD viene caricata col comando*

  use_module(library(fd))

  oppure, per semplicità

  lib(fd).
- *A questo punto abbiamo a disposizione:*
  - *::* *operatore per definire il dominio di una variabile, es* A::[0,3,7,10], B::[0..15].
    o di una lista di variabili: [A,B,C] :: [1..10,13].
  - *Vincoli predefiniti:*

    #<, #>, #=, #\=, #<=, #>=

Esempio

Carico la libreria

```
[eclipse 1]: lib(fd).
fd_domain.eco loaded traceable 0 bytes in 0.05 seconds
…
fd.eco     loaded traceable 0 bytes in 0.22 seconds

Yes (0.22s cpu)

[eclipse 2]: A::[0,3,7,10], B::[0..15], A#> B.

A = A{[3, 7, 10]}
B = B{[0..9]}

Delayed goals:
      A{[3, 7, 10]} - B{[0..9]}#>=1
Yes (0.00s cpu)
```

Domini arc-consistenti

Vincoli da soddisfare

VINCOLI UNARI E BINARI

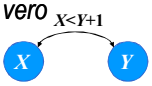
Interpretazione dei vincoli come grafo

- *Node Consistency: Un vincolo c(X) è node consistente se*

  $\forall d \in dom(X)$, $c(d)$ *è vero (soddisfatto)*

- *Arc Consistency (AC): Un vincolo c(X,Y) è arc consistente se*

  $\forall d \in dom(X)\ \exists g \in dom(Y)$ *t.c.* $c(d,g)$ *è vero (soddisfatto) e viceversa*



Algoritmi per ottenere AC

- Vari algoritmi sono stati proposti per rendere una rete AC: (AC1) AC2, AC3, … AC7, AC2000, AC 2001, …
- Ogni algoritmo usa una lista in cui si ricorda "che cosa deve ancora valutare"
- AC3 è uno dei più usati, perché è semplice ed utilizza una lista di vincoli

AC3 (Mackworth)

La (List of active constraints) = lista di tutti i vincoli;
Ls (List of sleeping constraints) = ∅;
while La ≠ ∅ do
  prendi un vincolo c(X,Y) ∈ La e toglilo da La
  se ci sono elementi inconsistenti in dom(X)
    allora eliminali (se dom(X) = ∅, fallisci)
      metti in La tutti i vincoli in Ls che coinvolgono X
  (stesso ragionamento per Y)
  se c(X,Y) non è completamente risolto
    allora mettilo in Ls

Vincolo Completamente Risolto

- Un vincolo è completamente risolto (o entailed dai domini) se per ogni possibile assegnamento esso è vero
- Es

  X:: 0..5, Y::10..20, X#<Y

- Sicuramente se tutte le variabili che coinvolge sono istanziate, il vincolo è risolto (oppure è falso)

Domanda: è la stessa cosa dell'Arc-Consistency?
Domanda: e se solo una delle 2 variabili è istanziata, posso dire che è risolto?

Complessità

- A volte l'Arc-Consistency è troppo costosa
- Quando risveglio un vincolo c(X,Y), per ogni valore in dom(X) cerco un valore consistente in dom(Y)

  intuitivamente, circa d^2 confronti

  ogni volta che risveglio (se d è la cardinalità dei domini)!
- Mi accorgo di un fallimento quando un dominio è vuoto.
- Potrei fermarmi quando ho trovato un valore consistente
- Ragionare per intervalli, invece di verificare tutti gli elementi del dominio

Bound Consistency

- *Un vincolo c(X,Y) è bound consistente se*

  $\forall d \in \{min(X), max(X)\}\ \exists g \in dom(Y)$ *t.c.* $c(d,g)$ *è vero (soddisfatto) e viceversa*

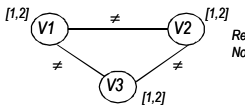
- ✓ *Comodo per vincoli come <, >, …*
- ✓ *Non ho bisogno di tenere una lista di elementi del dominio, ma bastano gli estremi*
- ✓ *Risveglio un vincolo c(X,Y) solo se elimino gli estremi del dom di X o Y*
- × *Faccio meno pruning*

BIBLIOGRAFIA

- NODE CONSISTENCY: banale
- ARC CONSISTENCY
  - *Algoritmi proposti*
    - *AC1 - AC2 - AC3 [Mackworth AIJ (8), 77] [Montanari Inf.Sci (7), 74], AC4 [Mohr, Henderson AIJ(28), 86],* AC5 *[Van Hentenryck, Deville and Teng AIJ(58), 92],* AC6 *[Bessiere AIJ(65), 94],* AC7 *[Bessiere, Freuder, Regin AIJ(107), 99]*
    - *Varianti: DAC [Detcher,Pearl IJCAI85], LAC [Schiex,Régin,Gaspin,Verfaillie AAAI96] MAC [Bessiere, Freuder, Regin, IJCAI95]*
    - Bound Consistency *[Van Hentenryck, Saraswat, Deville TR Brown, CS-93-02, 93]*
    - Complessita': *[Mackworth, Freuder AIJ(25), 85], [Mohr, Henderson AIJ(28), 86], [Detcher,Pearl AIJ (34), 88] [Han, Lee AIJ(36), 88], [Cooper AIJ (41), 89]*
- PATH CONSISTENCY
  - *PC1 - PC2 [Mackworth AIJ (8), 77]*
  - *PC3 [Mohr, Henderson AIJ(28), 86]*
  - *PC4 [Han, Lee AIJ(36), 88]*

INCOMPLETENESS of CONSISTENCY ALGORITHMS

- NODE, ARC CONSISTENCY in generale non sono completi
  - sono complete per particolari problemi che hanno strutture particolari *[Freuder JACM (29), 82], [Freuder JACM (32), 85]*
- Algoritmo completo: N-CONSISTENCY per problemi di N variabili. Complessità esponenziale *[Freuder CACM (21), 78], [Cooper AIJ (41), 89]*
- Esempio:

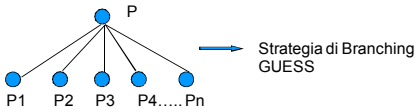


Rete Node + Arc consistent
Non ammette soluzioni ammissibili

RICERCA

- La propagazione non è in generale completa. Dopo la propagazione:
  - Trovo una soluzione ⟹ stop
  - Fallimento ⟹ backtracking
  - I domini contengono alcuni valori ⟹ SEARCH
- Ricerca: idea
  - Divide il problema in sotto-problemi (BRANCHING) e risolve ciascuno di essi indipendentemente
  - I sottoproblemi devono essere una partizione del problema originale
- Scopo: mantenere lo spazio di ricerca il più piccolo possibile
  - per convenzione, i rami di sinistra vengono esplorati per primi.

RICERCA



- Strategie di Branching definiscono il modo di partizionare il problema P in sottoproblemi più facili P1, P2, …, Pn.
- Per ogni sotto problema si applica di nuovo la propagazione. Possono essere rimossi nuovi rami grazie alle nuove informazioni derivate dal branching

RICERCA

- In Programmazione logica a vincoli la tecnica più popolare di branching è detta *labeling*
- LABELING:
  - Seleziona una VARIABILE
  - Seleziona un VALORE nel suo dominio
  - Assegna il VALORE alla VARIABILE
- L'ordine in cui le variabili e i valori vengono scelti (la search strategy) non influenza la completezza dell'algoritmo ma ne influenza pesantemente l'efficienza.
- Attività di ricerca volta a trovare buone strategie.

Labeling in ECLiPSe

`indomain(X)`

- *assegna alla variabile X un valore nel dominio; in backtracking ne seleziona un altro*
- *utile per definire predicati di labeling*

```
labeling([]).
labeling([H|T]):-
  indomain(H),
  labeling(T).
```


ESEMPIO COMPLETO

```
map_colouring([V1,V2,V3,V4,V5]):-
     V1::[red,green,yellow,blue],
     V2::[red,green,yellow,blue],
     V3::[red,green,yellow,blue],
     V4::[red,green,yellow,blue],
     V5::[red,green,yellow,blue],
     V1#\=V2, V1#\=V3, V1#\=V4, V1#\=V5, V2#\=V3,
     V2#\=V4, V2#\=V5, V3#\=V4, V4#\=V5,
     labeling([V1,V2,V3,V4,V5]).
```

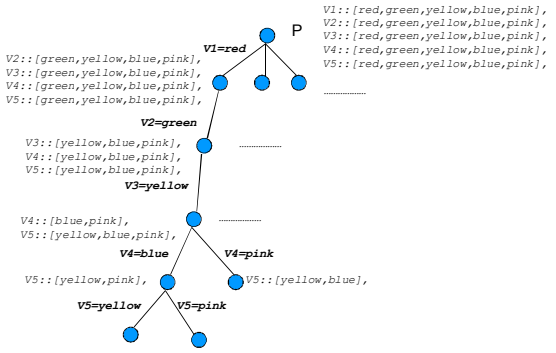
Variabili & domini

vincoli

ricerca

- Separazione fra
  - Modello del problema
  - strategia per risolverlo

SPAZIO DI RICERCA

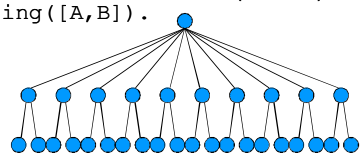


STRATEGIE DI RICERCA: CRITERI GENERALI

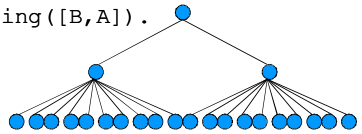
- Scelta della Variabile: prima istanzio le variabili più difficili
  - FIRST FAIL: seleziono prima la variabile con dominio più piccolo
  - MOST CONSTRAINED: seleziono prima la variabile coinvolta nel maggior numero di vincoli
  - APPROCCI IBRIDI: combinazioni dei due
- Scelta del valore: valori più promettenti prima
  - LEAST CONSTRAINING PRINCIPLE.
- Sono poi state definite numerose strategie dipendenti dal problema.

First Fail

- `A::[1..10], B::[1..2],` *(vincoli ...),* `labeling([A,B]).`



... `labeling([B,A]).`



First Fail in ECLiPSe

`deleteff(X,List,Resto)`

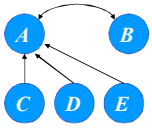
- *Data una lista, fornisce la variabile col dominio più piccolo (ed il resto della lista)*

```
labelingff([]).
labelingff(List):-
  deleteff(X,List,Resto),
  indomain(X),
  labelingff(Resto).
```

Most Constrained

`deleteffc(X,List,Resto)`

- *Data una lista, fornisce la variabile col dominio più piccolo (ed il resto della lista)*
- *A parità di dominio, sceglie quella soggetta a più vincoli*



COME SCEGLIERE UNA STRATEGIA

- Non esistono regole definite per scegliere la migliore strategia di ricerca. Dipende dal problema che dobbiamo risolvere e tipicamente la scelta viene effettuata dopo prove computazionali con diverse strategie
- CRITERIO: una strategia è più efficiente se rimuove prima rami dello spazio delle soluzioni.
- PARAMETRI da considerare
  - tempo computazione
  - numero di fallimenti.

Esercizi

- *Si scriva un predicato CLP(FD) che impone che una lista di variabili FD, date in ingresso, sia ordinata in senso decrescente stretto.*
- *Esempio:*

```
?- [A,B,C]::0..10,  ordina([A,B,C,D]).
  yes, A:: 2..10
  B:: 1..9
  C:: 0..8
```

- *Si scriva un predicato CLP(FD) che impone che i valori di una lista di variabili FD siano tutti diversi fra loro*

VINCOLI N-ari

- *Per vincoli N-ari, non c'è più l'interpretazione come grafo. Es: X+Y=Z ?*
- *Generalized Arc Consistency (GAC) (o Hyper Arc-Consistency): Un vincolo* $c(X_1,X_2,\ldots,X_n)$ *è* arc consistente in senso generalizzato *se*
  - *presa una variabile* X_i *(*i=1*..n)*
  - *per ogni assegnamento delle rimanenti* n-1 *variabili*

    $X_1 \rightarrow v_1,\ldots,X_{i-1} \rightarrow v_{i-1},X_{i+1} \rightarrow v_{i+1},\ldots,X_n \rightarrow v_n$

  $\exists g \in dom(X_i)$ *t.c.* $c(v_1,\ldots,v_{i-1},g,v_{i+1},v_n)$ *è vero (soddisfatto).*

Domanda: Sapete definire Generalized Bound Consistency?

Vincoli N-ari: espressioni

- *Si possono definire vincoli come*

  `somma(A,B,C)`

  vero se A+B=C, con propagazione GAC.
- *Questi vincoli sono già definiti, con zucchero sintattico. Possiamo usare direttamente:*

  `A+B #= C`

  `A#< B*C+D,`

  …

PROPAGAZIONE DI VINCOLI

- Finora abbiamo visto propagazioni generali
- Vincoli Simbolici:
  - In generale, la propagazione ha costo esponenziale nel numero di variabili
  - Per vincoli specifici, si puo` avere propagazione polinomiale
  - Ogni vincolo ha associato un algoritmo di *PROPAGAZIONE* o di *FILTERING*
    - Algoritmo di filtering implementa tecniche complesse di propagazione che derivano da studi effettuati nell'Intelligenza Artificiale e Ricerca Operativa
  - La propagazione termina quando la rete raggiunge uno stato di quiescenza non si possono più cancellare valori
  - Propagazione incrementale

PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: esempio 1
  - `alldifferent([`X_1`,...`X_n`])`
    vero se tutte le variabili assumono valori diversi

  Equivalente da un punto di vista dichiarativo all'insieme di vincoli binari

  `alldifferent([`X_1`,...`X_n`])` $\leftrightarrow X_1 \neq X_2,\ X_1 \neq X_3, \ldots, X_{n-1} \neq X_n$

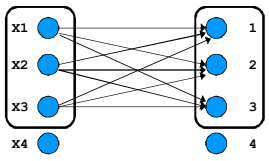
  Operazionalmente permette una propagazione più forte.

  `X1::[1,2,3],X2::[1,2,3],X3::[1,2,3],X4::[1,2,3,4]`
- *Arc consistency:*
- *non rimuove alcun valore !!*
- *Algoritmo di filtering **[Regin AAAI94]**: rimuove da* `X4` *i valori 1,2,3*

PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: esempio 1

```
lib(fd_global).
X1::[1,2,3],X2::[1,2,3],X3::[1,2,3],X4::[1,2,3,4],
alldifferent([X1,X2,X3,X4]).
```

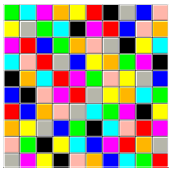


Insieme di variabili di cardinalità 3 che hanno medesimo dominio di cardinalità 3

`X4::[1,2,3,4]`

ALL-DIFFERENT

- Vincoli Simbolici: L'**`alldifferent`** si usa in tantissime applicazioni
- Esempio: Partial Latin Square



Colorare ogni riga e colonna con 10 colori in modo che su ogni riga e colonna ci siano tutti colori diversi

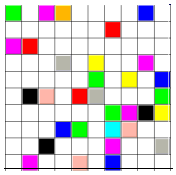
FACILE SE LA GRIGLIA E' VUOTA ma

non se PARZIALMENTE PIENA

35%-45% PERCENTUALE CRITICA

ALL-DIFFERENT NEL PARTIAL LATIN SQUARE

- Problema la cui struttura si trova in molte applicazioni (scheduling e timetabling, routing in fibre ottiche, ecc…)
- Modello del problema: alcune variabili già assegnate, altre hanno come dominio tutti i colori



per ogni riga i=1..n
`alldifferent([`X_{i1}**`,`**X_{i2}**`,….`**X_{in}**`])`**
per ogni colonna j=1..n
`alldifferent([`X_{1j}**`,`**X_{2j}**`,….`**X_{nj}**`])`**

`SI VEDA`
`http://www.cs.cornell.edu/gomes`

32% preassignment

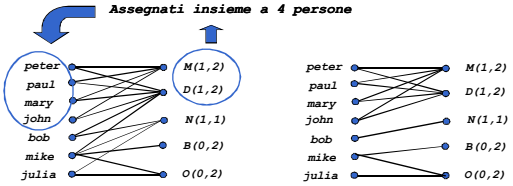
Nota sintattica

- *In ECLiPSe il vincolo alldifferent è definito in due librerie, con implementazione diversa*
  - **`fd:alldifferent`** *impone n(n-1)/2 vincoli* **`#\=`**
  - **`fd_global:alldifferent`** *è il vincolo globale (in ECLiPSe fa bound consistency)*
- *Per distinguere due predicati con lo stesso nome definiti in librerie diverse (se ho caricato entrambe le librerie), uso la notazione*

`libreria:nome_predicato`

GLOBAL CARDINALITY CONSTRAINT

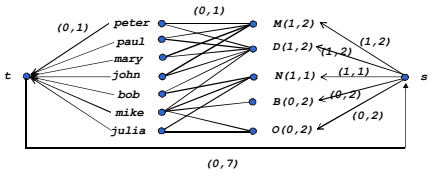
- **`gcc(Var,Val,LB,UB)`** *[Regin AAAI96]* **`Var`** sono variabili, **`Val`** valori **`LB`** e **`UB`** sono il minimo e massimo numero di occorrenze per ogni valore in **`Val`** assegnato a **`Var`**
- Esempio:



GLOBAL CARDINALITY CONSTRAINT

- La nozione di consistenza si basa su un algoritmo di *network maximum flow* sulla value network N(C)

- **gcc su k variabili e' consistente**
- **c'e' un max flow da s a t di valore k**



PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: esempio
  - **element(N,[X_1,...X_m],Value)**
    l'ennesimo elemento della lista è uguale a Value
- propagazione da **N** a **Value**:
  - **N=i → X_i = Value**
- propagazione da **Value** a **N** e **X_i**:
  - **Value = x → N=1 and X_1=x or**
    N=2 and X_2=x or....
    N=m and X_m=x

PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: vincoli disgiuntivi
  - Supponiamo di avere due lezioni che devono essere tenute dallo stesso docente. Abbiamo gli istanti di inizio delle lezioni: **L1Start** e **L2Start** e la loro durata **Duration1** e **Duration2**.
  - Le due lezioni non possono sovrapporsi:

    L1Start + Duration1 ≤ L2Start
    OR
    L2Start + Duration2 ≤ L1Start

  - Due problemi INDIPENDENTI uno per ogni parte della disgiunzione.

PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: vincoli disgiuntivi
  - Due problemi INDIPENDENTI, uno per ogni parte della disgiunzione: una scelta non ha effetto sull'altra Trashing ⇒
  - Numero esponenziale di problemi:
    - N disgiunzioni ⇒ 2^N Problemi
    - Fonte primaria di complessità in problemi reali
  - Soluzioni proposte:
    - *disgiunzione costruttiva*
    - *operatore di cardinalità*
    - *meta-constraint*

DISGIUNZIONE COSTRUTTIVA

- *P. Van Hentenryck, V. Saraswat, Y. Deville, Design, Implementation and Evaluation of the Constraint Language cc(FD), Tech. Rep. Brown University, CS-93-02, 1993.*
- Sfrutta la disgiunzione per ridurre lo spazio di ricerca
- Idea: aggiungere al constraint store vincoli che sono implicati da tutte le parti della disgiunzione
- Esempio: `X::[5..10], Y::[7..11], Z::[1..20], (Z=X OR Z=Y)`
  - `Z=X` ridurrebbe il dominio di `Z` a `[5..10]`
  - `Z=Y` ridurrebbe il dominio di `Z` a `[7..11]`
  - risultato della disgiunzione costruttiva: `Z::[5..11]`
- In ECLiPSe non c'è, ma può essere implementato semplicemente (V. `Propia`)

OPERATORE DI CARDINALITA'

- Symbolic Constraint: operatore di cardinalità
  - `#(l,[c1,…,cn],u)` ⇔ il numero `k` di vincoli c_i $(l \le i \le n)$ soddisfatti non è minore di `l` e non maggiore di `u`
  - Con l'operatore di cardinalità modello i vincoli disgiuntivi nel modo seguente

```
#(1,[L1Start+Duration1 ≤ L2Start, L2Start+Duration2 ≤
L1Start],1)
```


META-CONSTRAINTS

- Vincoli Simbolici: meta-constraints o Vincoli Reificati
  - A ogni vincolo (matematico, del tipo #>, #<, #<=, #\=, …) viene associata una variabile booleana `B`.
    - Se `B=1` il vincolo è verificato e viceversa,
    - se `B=0` il vincolo non è verificato e viceversa.

`c` ⇔ `B`

  - Con i meta-constraints posso modellare la disgiunzione nel modo seguente:

```
B1 ::[0,1], B2::[0,1],
L1Start+Duration1 #<= L2Start #<=> B1,
L2Start+Duration2 #<= L1Start #<=> B2,
B1 + B2 #= 1.
```


PROPAGAZIONE DI VINCOLI

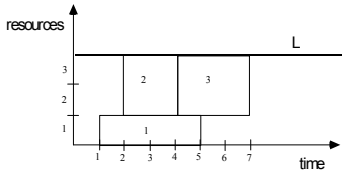
- Vincoli Simbolici: esempio 3
  - `cumulative([S1,...Sn], [D1,...Dn], [R1,...Rn], L)`
    - $S_1,\ldots S_n$ sono istanti di inizio di attività (variabili con dominio)
    - $D_1,\ldots D_n$ sono durate (variabili con dominio)
    - $R_1,\ldots R_n$ sono richieste di risorse (variabili con dominio)
    - `L` limite di capacita' delle risorse (fisso o variabile nel tempo)
- Dato l'intervallo [min,max] dove min = min_i $\{S_i\}$, max = max$\{S_i+D_i\}$ - 1, il vincolo cumulative assicura che

$$\max\{\sum_{j \mid S_j \le i \le S_j+D_j} R_i\} \le L$$

PROPAGAZIONE DI VINCOLI

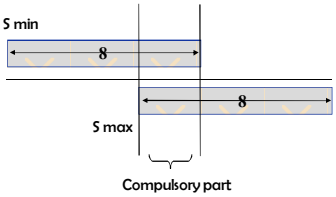
- Vincoli Simbolici: esempio 3

```
lib(edge_finder) o lib(edge_finder3).
cumulative([1,2,4],[4,2,3],[1,2,2],3)
```



PROPAGAZIONE DI VINCOLI

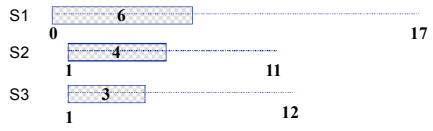
- Vincoli Simbolici: esempio 3
  - un esempio di propagazione usato nei vincoli sulle risorse è quello basato sulle *parti obbligatorie*



PROPAGAZIONE DI VINCOLI

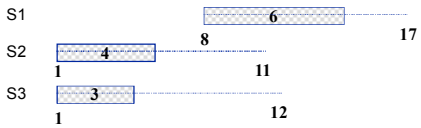
- Vincoli Simbolici: esempio 3
  - un'altra propagazione usata nel vincolo di capacità è quella basata sull' *edge finding* ***[Baptiste, Le Pape, Nuijten, IJCAI95]***

Consideriamo una risorsa unaria e tre attività



PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: esempio 3



Possiamo dedurre che minimo istante di inizio per S1 e' 8.

Considerazione basata sul fatto che S1 deve essere eseguito dopo S2 e S3.

Ragionamento globale: supponiamo che S2 o S3 siano eseguiti dopo S1. Allora l'istante di fine di S2 e S3 e' almeno 13 (elemento non contenuto nel dominio di S2 e S3).

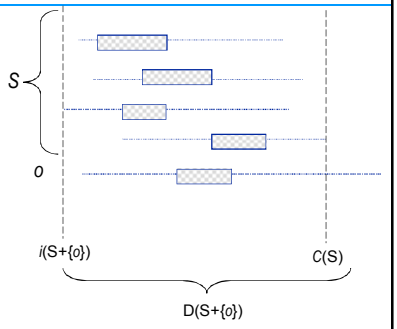
Teorema: [Carlier, Pinson, Man.Sci.95]

Sia *o* un'attività e *S* un insieme di attività che utilizzano la stessa risorsa unaria (*o* non contenuta in *S*). Il minimo istante di inizio è *i*, la somma delle durate è *D* e il massimo istante di terminazione è *C*. Se

$i(S+\{o\}) + D(S+\{o\}) > C(S)$

Allora non è possibile che *o* preceda alcuna operazione in *S*. Questo implica che il minimo istante iniziale per *o* può essere fissato a

$\max_{(S' \subseteq S)} \{i(S') + D(S')\}$.



VINCOLI: CONSIDERAZIONI GENERALI

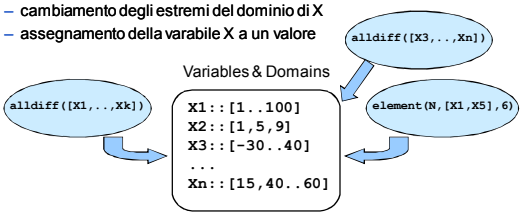
- Vincoli simbolici disponibili nella maggior parte dei linguaggi a vincoli
- Ragionamento locale vs. globale ⇒ propagazione potente
- Ragionamento locale vs. globale ⇒ costo computazionale

  } Tradeoff

- Generalizzazione di vincoli che appaiono frequentemente in applicazioni reali
- Codice conciso e semplice da capire e scrivere
- Vincoli simbolici rappresentano sottoproblemi indipendenti (rilassamenti del problema originale)

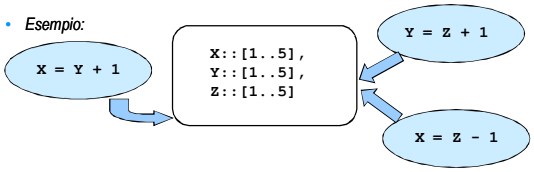
INTERAZIONE TRA VINCOLI

- I vincoli interagiscono attraverso variabili condivise nel constraint store
- La propagazione di un vincolo attivata quando sulla variabile X coinvolta nel vincolo si scatena un evento
  - cambiamento nel dominio di X
  - cambiamento degli estremi del dominio di X
  - assegnamento della varabile X a un valore

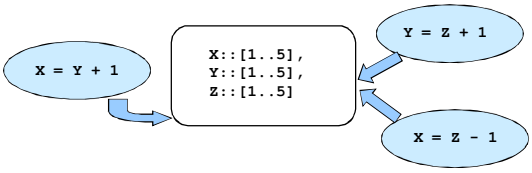


INTERAZIONE TRA VINCOLI

- In generale ogni variabile è coinvolta in molti vincoli. Di conseguenza, ogni cambiamento nel dominio della variabile come risultato di una propagazione può causare la rimozione di altri valori dai domini delle altre variabili.
- Prospettiva ad agenti: durante il loro tempo di vita, i vincoli alternano il loro stato da sospesi e attivi (attivati da eventi)
- *Esempio:*



INTERAZIONE TRA VINCOLI



- Prima propagazione di `X = Y + 1` porta a

```
X::[2..5]
Y::[1..4]
Z::[1..5]
```

`X = Y + 1`
sospeso

INTERAZIONE TRA VINCOLI

- Seconda propagazione di `Y = Z + 1` porta a

```
X::[2..5]
Y::[2..4]
Z::[1..3]
```

`Y = Z + 1`
sospeso

- Il dominio di `Y` e' cambiato `X = Y + 1` viene attivato

```
X::[3..5]
Y::[2..4]
Z::[1..3]
```

`X = Y + 1`
sospeso

INTERAZIONE TRA VINCOLI

- Terza propagazione di `Z = X - 1` porta a

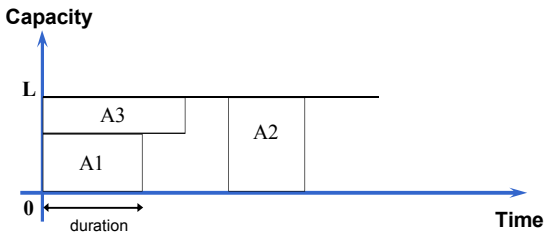
```
X::[]
Y::[2..4]
Z::[1..3]
```

`FAIL`

- L'ordine nel quali i vincoli sono considerati (sospesi/risvegliati) non influenza il risultato MA può influenzare le prestazioni computazionali

VINCOLI COME COMPONENTI SOFTWARE

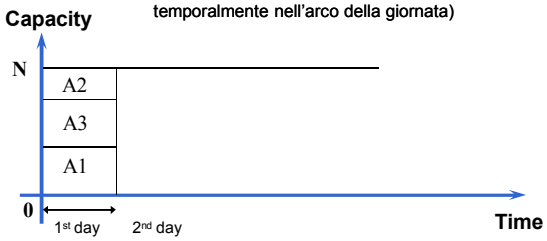
- I vincoli simbolici possono essere visti come componenti software utilizzabili in diverse situazioni. Ad esempio il vincolo cumulative può essere usato in vari modi
  - Scheduling (1): Attività A1, A2, A3 condividono la stessa risorsa con capacità limitata. Durate sull'asse X e uso delle risorse su Y



VINCOLI COME COMPONENTI SOFTWARE

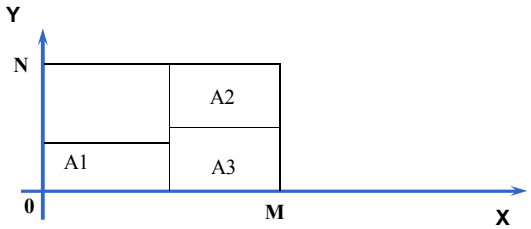
- Altro esempio di uso del vincolo cumulativo
  - Scheduling (2): *Numero limitato di risorse per giorno = N.* Rappresento i giorni sull'asse X e il numero di risorse usate su Y

(Non interessa dove le attivita' sono collocate temporalmente nell'arco della giornata)



VINCOLI COME COMPONENTI SOFTWARE

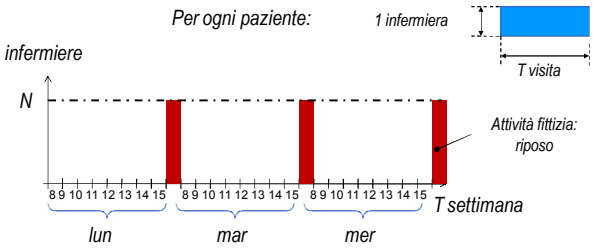
- Altro esempio di vincolo cumulativo
  - Packing: *Data una scatola di dimensioni M x N, è necessario collocare dei pezzi in modo che le dimensioni della scatola siano rispettate.*



Esempio

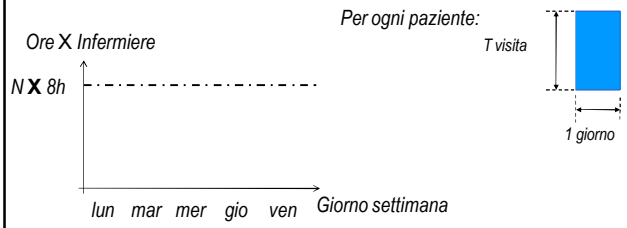
- *N infermiere devono visitare P pazienti nell'arco di una settimana*
- *Per ogni paziente si sa quanto tempo e` richiesto per la visita*
  - *In piu`, possono esserci vincoli sull'orario in cui viene visitato un paziente, alcuni pazienti vanno visitati piu` volte in una settimana, ecc.*
- *Ogni infermiera lavora 8 ore al giorno*
- *Trovare un assegnamento dei pazienti alle infermiere*

Uso del vincolo cumulativo (I)



- *Tempo = ore all'interno della settimana*
- *Risorse = infermiere*
- *Limite risorse = Numero infermiere*

Uso del vincolo cumulativo (II)



- *Tempo = giorno della settimana (da 1 a 7)*
- *Risorse = Tempo della visita*
- *Limite risorse = (Numero infermiere)* **X** *8h*

Alcuni vincoli utili

Vedere il manuale: librerie **`fd`** *e* **`fd_global`**

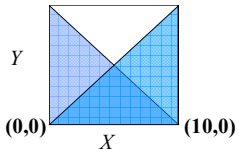
- *atmost(+N, ?List, +V): At most N elements of the list List have the value V.*
- *element(?Index, +List, ?Value): Value is the Index'th element of the integer list List.*
- *alldifferent(+List,++Capacity): The list List contains at most Capacity elements of each value*
- *lexico_le(+List1, +List2): List1 is lexicographically less or equal to List2*
- *maxlist(+List, ?Max): Max is the maximum of the values in List*
- *minlist(+List, ?Min): Min is the minimum of the values in List*
- *occurrences(++Value, +List, ?N): The value Value occurs in List N times*
- *sorted(?List, ?Sorted): Sorted is a sorted permutation of List*
- *sumlist(+List, ?Sum): The sum of the list elements is Sum*

VINCOLI RIDONDANTI

- La propagazione in generale non è completa: alcuni valori inconsistenti possono essere lasciati nei domini
- I vincoli ridondanti possono essere utili per ridurre lo spazio di ricerca anche se introducono un overhead (trade-off).
- Un vincolo ridondante C è un vincolo che è implicato da altri vincoli {C1…Ck}, ma il risolutore non identifica questa implicazione a causa della sua incompletezza

Vincoli logicamente ridondanti

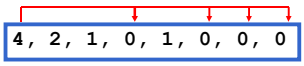
```
[X,Y]::0..10,     X#>=Y,
    Y #=< 10-X
```



- *Nessuna propagazione (entrambi i vincoli già AC).*
- *Se aggiungo il vincolo (logicamente ridondante)* **`Y#=<5`** *ho propagazione*

VINCOLI RIDONDANTI

- Esempio: Sequenza magica
- Dato un insieme di n+1 variabili $X_0,\ldots,X_n$. Ogni X_i deve rispettare i seguenti vincoli :
  - 0 compare X_0 volte in soluzione
  - 1 compare X_1 volte
  - …
  - n appare X_n volte.



- Vincolo `occurrences(V,L,N)` (`fd_global`) impone che V compaia N volte nella lista L
- `magic_sequence([X0,…,Xn]):-`

```
X0,…,Xn ::[0..n],
occurrences(0,[X0,…,Xn], X0),
occurrences(1,[X0,…,Xn], X1),
...,
occurrences(n,[X0,…,Xn], Xn),
...
```


VINCOLI RIDONDANTI

- Vincolo ridondante: si noti che la somma di tutte le variabili moltiplicate per il loro valore è uguale al numero di celle nella sequenza.

`4, 2, 1, 0, 1, 0, 0, 0`

- Quindi, le variabili soddisfano il vincolo:

$$X_1 + 2*X_2 + \ldots + N*X_n = N + 1$$

- `magic_sequence([X0,…,Xn]):-`

```
X0,…,Xn ::[0..n],
occurrences(0,[X0,…,Xn], X0),
occurrences(1,[X0,…,Xn], X1),
...,
occurrences(n,[X0,…,Xn], Xn),
X1+ 2*X2 +…+ N*Xn = N + 1,
...
```

Con P4 2Ghz, N=23

- senza vincolo ridondante

5.88s

- con vincolo ridondante

0.55s