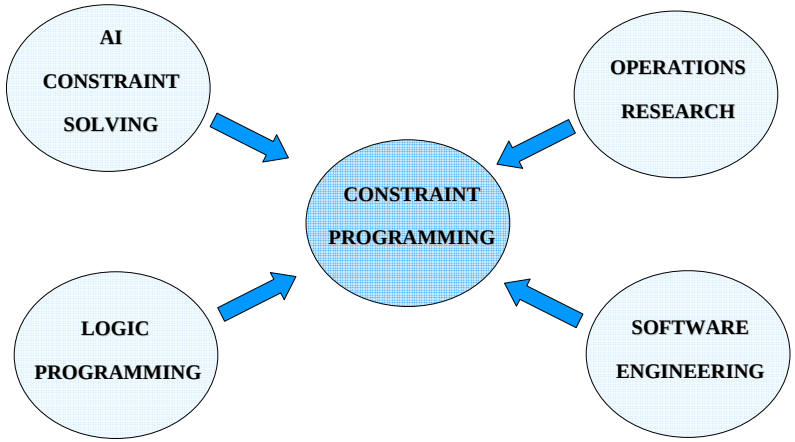
# PROGRAMMAZIONE LOGICA A VINCOLI






# 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* ($V_1$, $V_2$…$V_n$)
– *un dominio discreto per ogni variabile* ($D_1$,$D_2$…$D_n$)
– *un insieme di vincoli su queste variabili*:

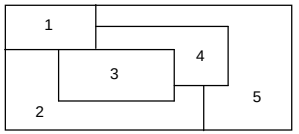
vincolo: una relazione tra variabili che definisce un sottoinsieme del prodotto cartesiano dei domini $D_1xD_2x...xD_n$

**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



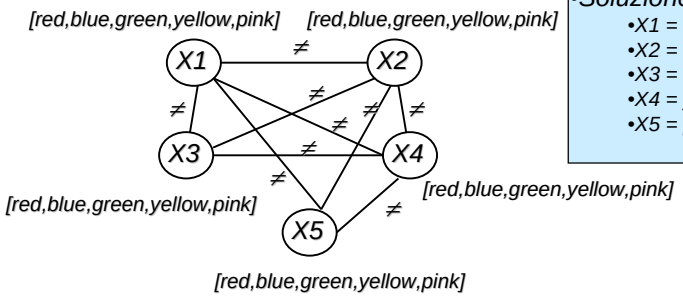

• Trovare un assegnamento di colori alle variabili consistente con i vincoli

– *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*

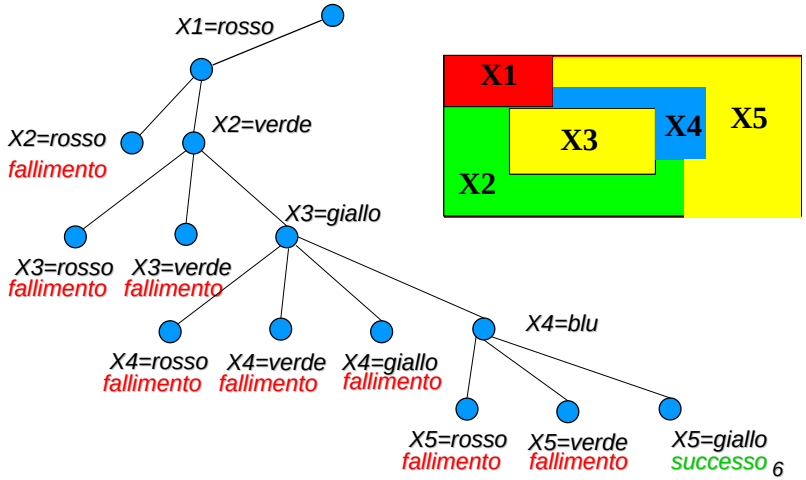



•*Soluzione ammissibile:*
- •*X1 = red*
- •*X2 = green*
- •*X3 = blue*
- •*X4 = yellow*
- •*X5 = pink*



# ESEMPIO: Map Coloring

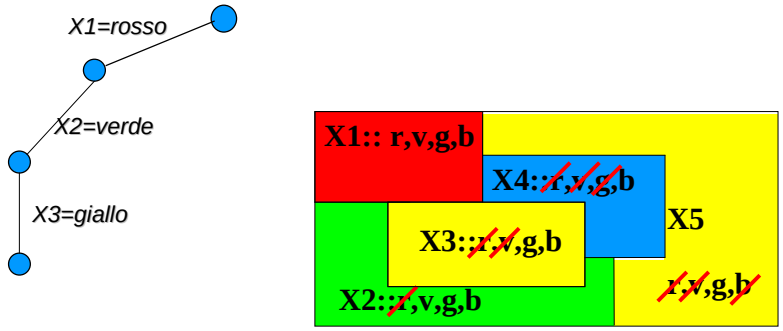
- Algoritmo semplice (ma a volte inefficiente)






# PROPAGAZIONE DI VINCOLI

- Eliminazione a priori dei valori inconsistenti






# PROBLEMI DI OTTIMIZZAZIONE

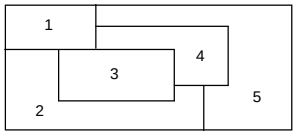
Un problema di ottimizzazione è definito da:

– *un insieme di variabili* ($X_1$, $X_2$…$X_n$)
– *un dominio discreto per ogni variabile* ($D_1$,$D_2$…$D_n$)
– *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...x$D_n$

– *una funzione obiettivo* $f(X_1, X_2…X_n)$

**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

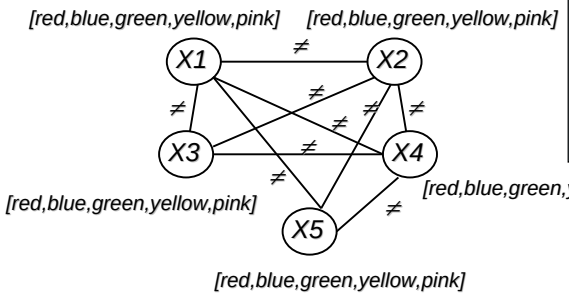
– *variabili V1, V2, V3, V4, V5: zone*
– *domini D1, D2, D3, D4, D5: [red, blue, green, yellow,pink]*
– *vincoli: near(Xi, Vj) $\Rightarrow$ Xi $\neq$ Xj*



## CONSTRAINT GRAPHS

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

– *variabili* $\Leftrightarrow$ *nodi*
– *vincoli* $\Leftrightarrow$ *(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

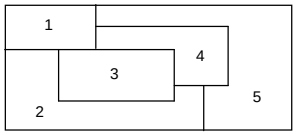
```
permutation([],[]).
permutation([Head|Tail],PermList) :-
  permutation(Tail,PermTail),
  delete(Head,PermList,PermTail).
delete(A, [A|B], B).
delete(A, [B, C|D], [B|E]) :-
  delete(A, [C|D], E).
```

*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

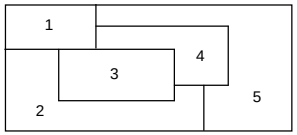



• Trovare un assegnamento di colori alle variabili consistente con i vincoli

– *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.
```



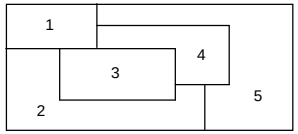

Molto dichiarativo!

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.
```

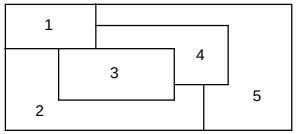



- L'ordine è importante ☹
- un po' meno dichiarativo
- Risolve solo questa istanza



# 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],NewV
   alues).
```

```
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).
```



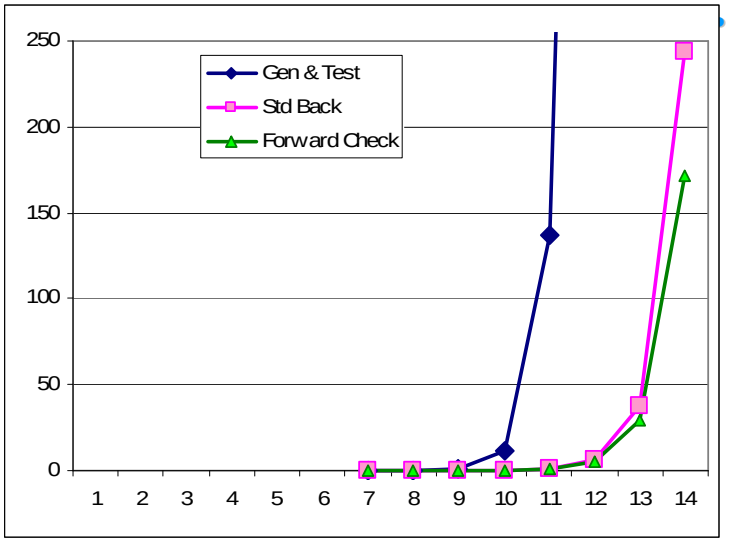
# N-Queens Forward Checking

```
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:*

$$X+\cancel{2}^{1} = Y+\cancel{1}$$

- *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))`**

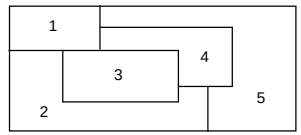
- *Indica a Prolog che l'***`atomo`** *deve essere selezionato (dalla risoluzione SLD) solo quando* **`X`** *non è variabile.*

```
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

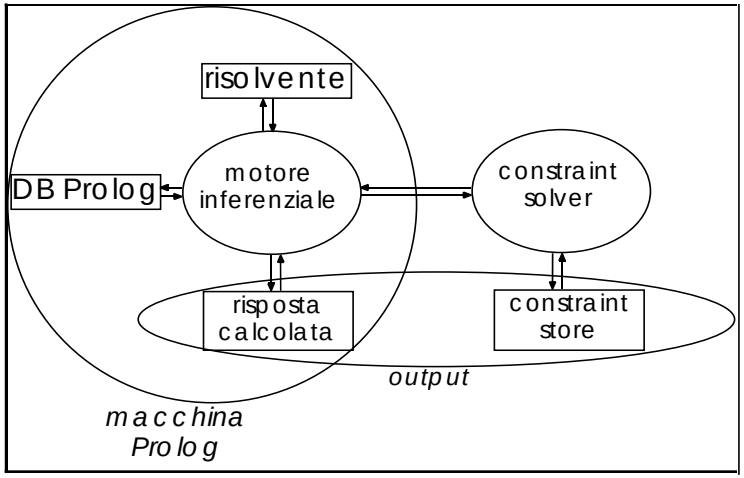
*Constraint Logic Programming (CLP)*

# 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*
  - *…*

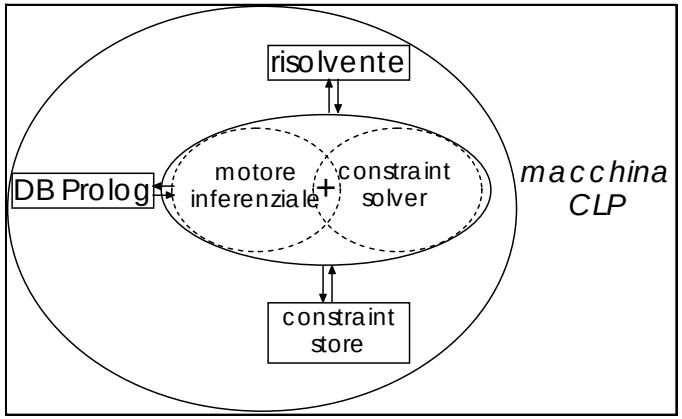


# ESTENDERE LA MACCHINA LOCIGA






# MACCHINA CLP





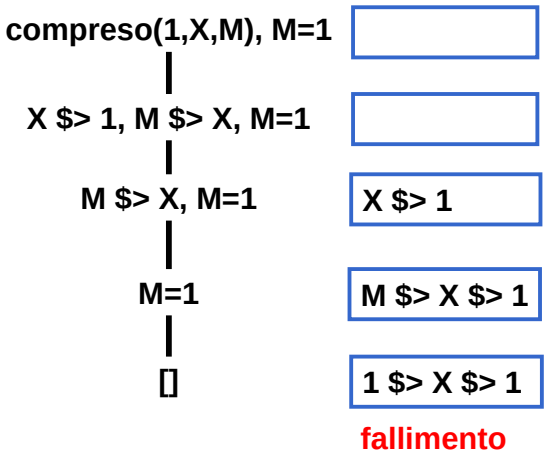

# CLP: sintassi

- *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*
  - *ECL$^i$PS$^e$*
  - *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*



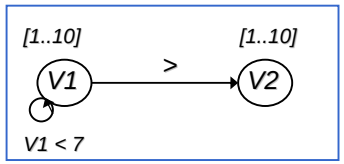
# PROPRIETA' DI CONSISTENZA

• NODE CONSISTENCY

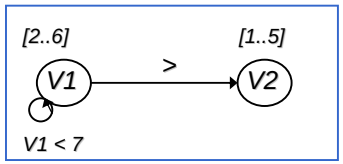
– *una rete è node consistent se in ogni dominio di ogni nodo ogni valore è consistente con i vincoli unari che coinvolgono la variabile*

• 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*

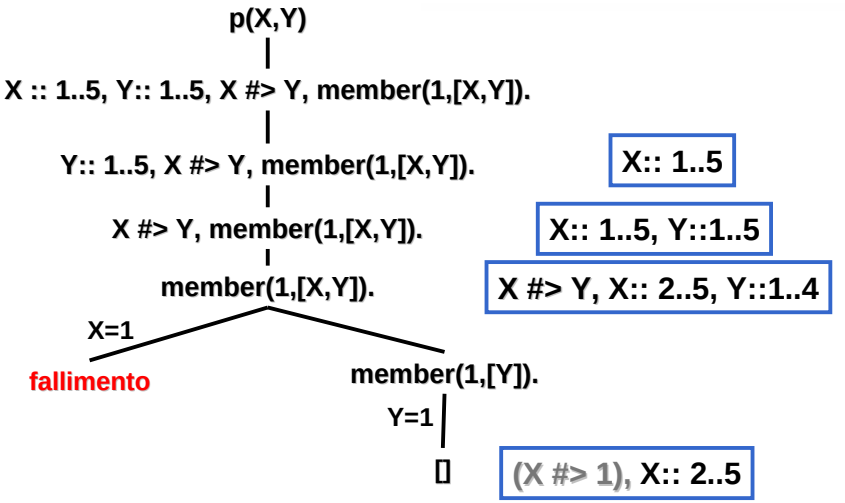
## Esempio:

```
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:

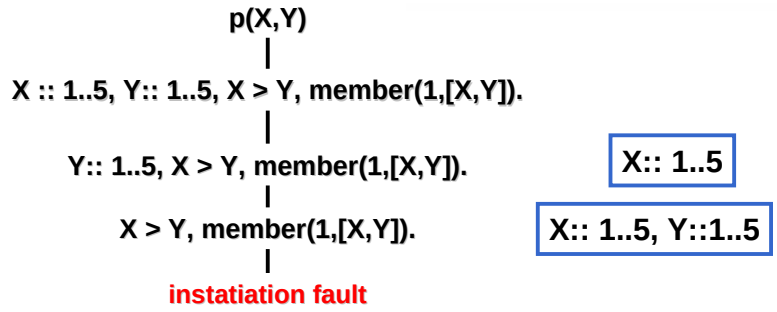
- *Che cosa sarebbe successo se avessi scritto X>Y invece di X #> Y?*

```
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([`$X_1$`,...,`$X_m$`])`
      tutte le variabili devono avere valori differenti
    - `element(N,[`$X_1$`,...`$X_m$`],Value)`
      l'ennesimo elemento della lista deve avere valore uguale a Value
    - `cumulative([`$S_1$`,...`$S_m$`], [`$D_1$`,...`$D_n$`], [`$R_1$`,...`$R_n$`], L)`
      usato per vincoli di capacita'
    - vincoli disgiuntivi



## PROPAGAZIONE DI VINCOLI

- Vincoli matematici:
  - Esempio 1
    - `X::[1..10], Y::[5..15], X#>Y`

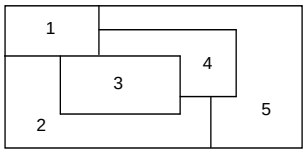
      Arc-consistency: per ogni valore v della variabile X, se non esiste un valore per Y compatibile con v, allora v viene cancellato dal dominio di X e viceversa

      `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*
  - *conjunto* *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

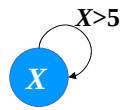


# VINCOLI UNARI E BINARI

*Interpretazione dei vincoli come grafo*

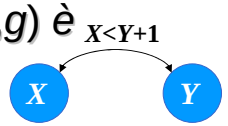
- *Node Consistency: Un vincolo c(X) è node consistente se*

  *∀d ∈ dom(X), c(d) è vero (soddisfatto)*




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

  *∀d ∈ dom(X) ∃g ∈ 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]***

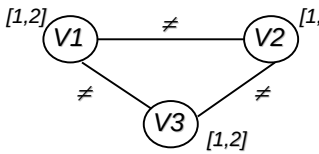


# INCOMPLETENESS of CONSISTENCY ALGORITHMS

- 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*



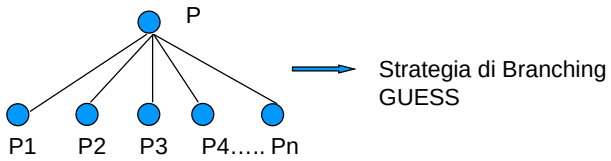
# RICERCA

- 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]).
```

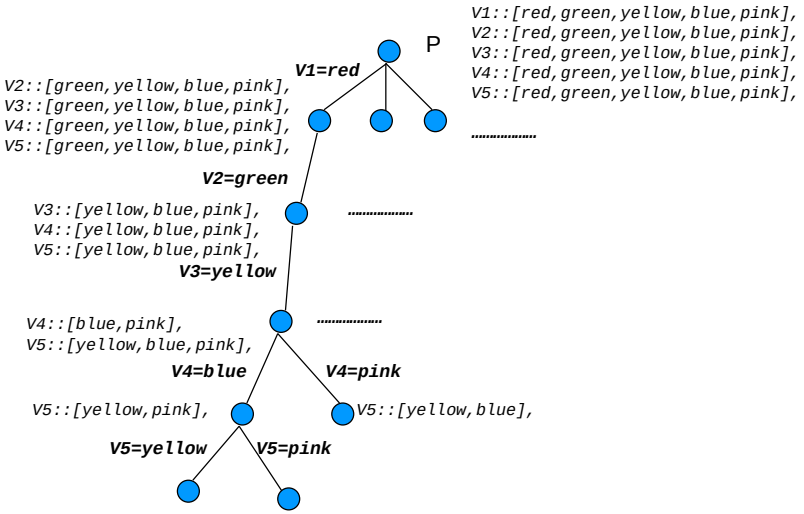
Variabili & domini

vincoli

ricerca

- Separazione fra
  - Modello del problema
  - strategia per risolverlo

## SPAZIO DI RICERCA





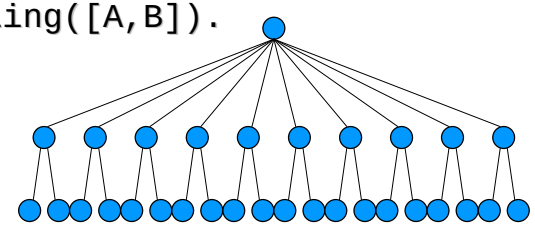

## STRATEGIE DI RICERCA: CRITERI GENERALI

- 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.

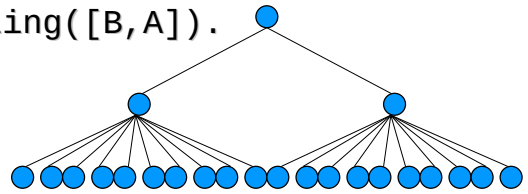


## *First Fail*

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



… `labeling([B,A]).`





## *First Fail in ECLiPSe*

deleteff(X,List,Resto)

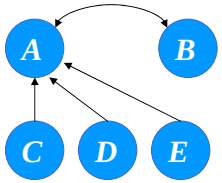
- *Data una lista, fornisce la variabile col dominio più piccolo (ed il resto della lista)*

```
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:
  - 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.

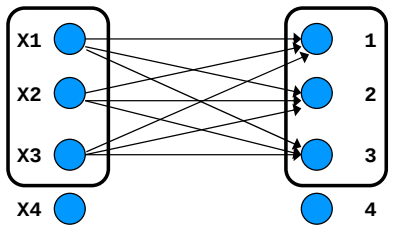
  **X1::[1,2,3],X2::[1,2,3],X3::[1,2,3],X4::[1,2,3,4]**
- *Arc consistency:*
- *non rimuove alcun valore !!*
- *Algoritmo di filtering **[Regin AAAI94]**: rimuove da* **X4** *i valori 1,2,3*



# 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]).
```

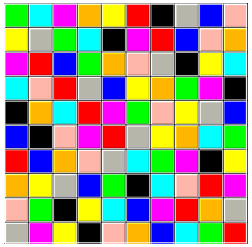



**Insieme di variabili di cardinalità 3 che hanno medesimo dominio di cardinalità 3**

**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*

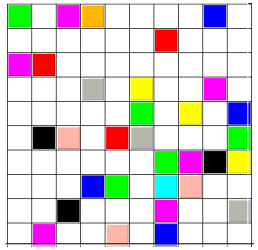
*non se PARZIALMENTE PIENA*

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

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

32% **preassignment**



## *Nota sintattica*

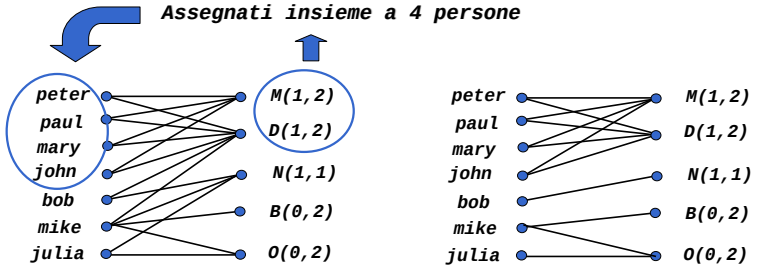
- *In ECLiPSe il vincolo alldifferent è definito in due librerie, con implementazione diversa*
  - **`fd:alldifferent`** *impone n(n-1)/2 vincoli* **`#\=`**
  - **`fd_global:alldifferent`** *è il vincolo globale (in ECLiPSe fa bound consistency)*
- *Per distinguere due predicati con lo stesso nome definiti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

**`libreria:nome_predicato`**



## GLOBAL CARDINALITY CONSTRAINT

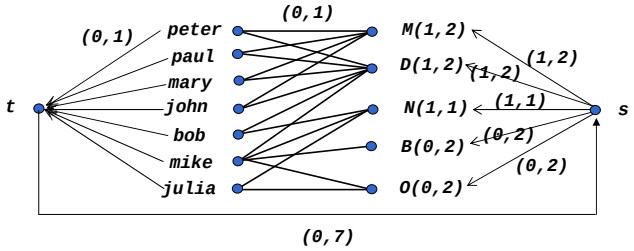
- **`gcc(Var,Val,LB,UB)`** *[Regin AAAI96]* **`Var`** sono variabili, **`Val`** valori **`LB`** e **`UB`** sono il minimo e massimo numero di occorrenze per ogni valore in **`Val`** assegnato a **`Var`**
- Esempio:



# 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 INDEPENDENTI 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,[c`**$_1$**`,…,c`**$_n$**`],u)`** $\Leftrightarrow$ il numero **`k`** di vincoli **`c`**$_i$ (**`1≤i≤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`** $\Leftrightarrow$ **`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

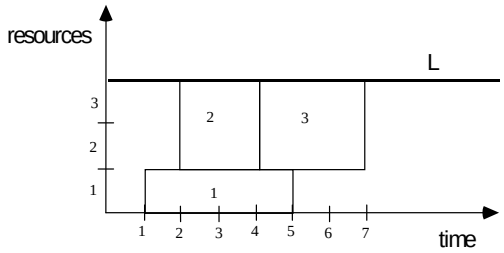
- Vincoli Simbolici: esempio 3
  - **`cumulative([S`**$_1$**`,...S`**$_n$**`], [D`**$_1$**`,...D`**$_n$**`], [R`**$_1$**`,...R`**$_n$**`], L)`**
    - **`S`**$_1$**`,...S`**$_n$ sono istanti di inizio di attività (variabili con dominio)
    - **`D`**$_1$**`,...D`**$_n$ sono durate (variabili con dominio)
    - **`R`**$_1$**`,...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

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

# PROPAGAZIONE DI VINCOLI

- Vincoli Simbolici: esempio 3

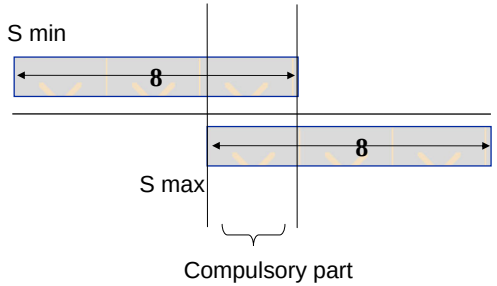
```
lib(edge_finder) o lib(edge_finder3).
cumulative([1,2,4],[4,2,3],[1,2,2],3)
```






# PROPAGAZIONE DI VINCOLI

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

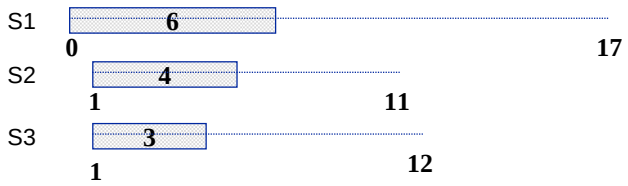





# PROPAGAZIONE DI VINCOLI

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

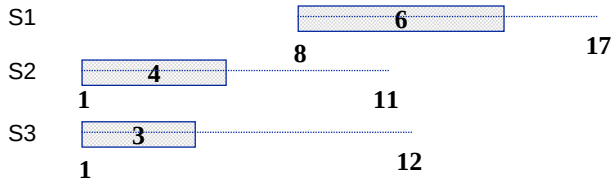
Consideriamo una risorsa unaria e tre attività






# 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).

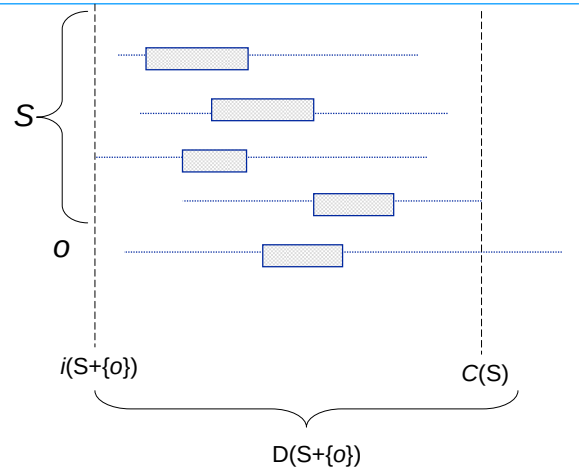
## Teorema: [Carlier, Pinson, Man.Sci.95]

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

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






## 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

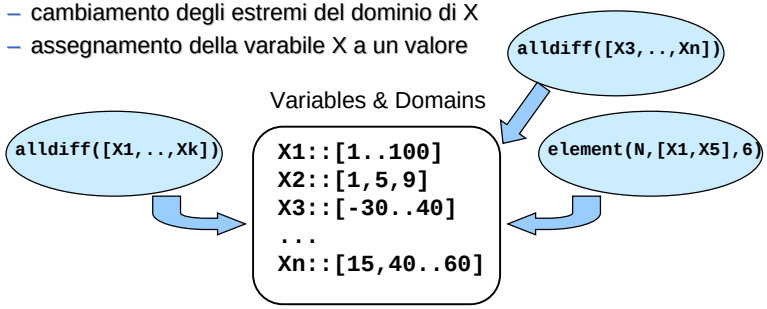
} Tradeoff

- 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

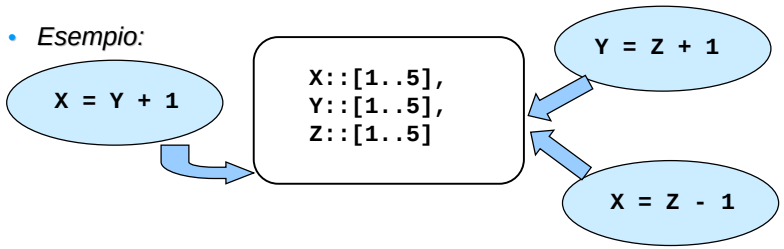
- I vincoli interagiscono attraverso variabili condivise nel constraint store
- La propagazione di un vincolo attivata quando sulla variabile X coinvolta nel vincolo si scatena un evento
  - cambiamento nel dominio di X
  - cambiamento degli estremi del dominio di X
  - assegnamento della varabile X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



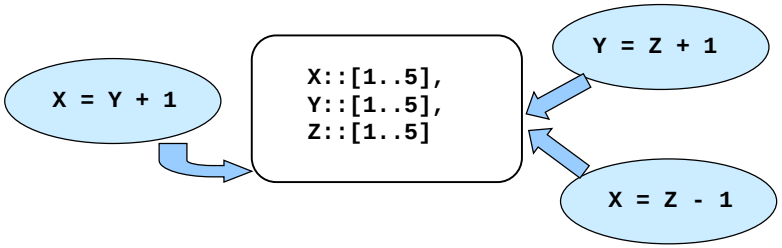



## INTERAZIONE TRA VINCOLI

- 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]
```

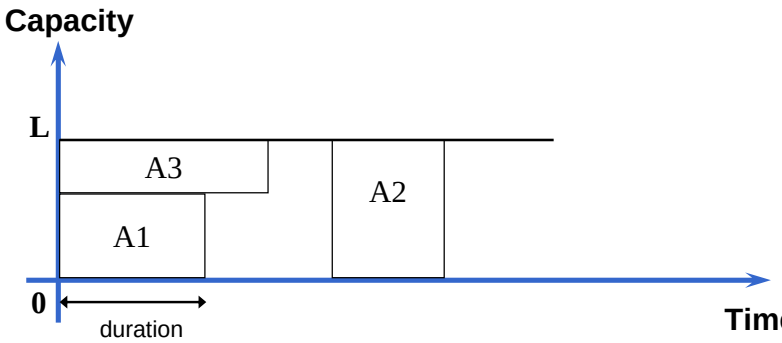
**`FAIL`**

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



# VINCOLI COME COMPONENTI SOFTWARE

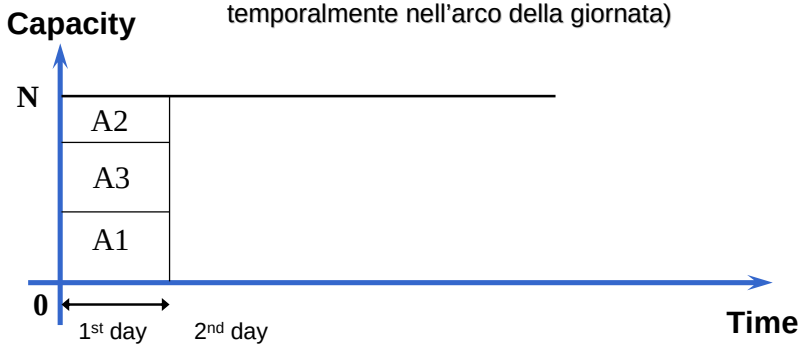
- I vincoli simbolici possono essere visti come componenti software utilizzabili in diverse situazioni. Ad esempio il vincolo cumulative può essere usato in vari modi
  - Scheduling (1): Attività A1, A2, A3 condividono la stessa risorsa con capacità limitata. Durate sull'asse X e uso delle risorse su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

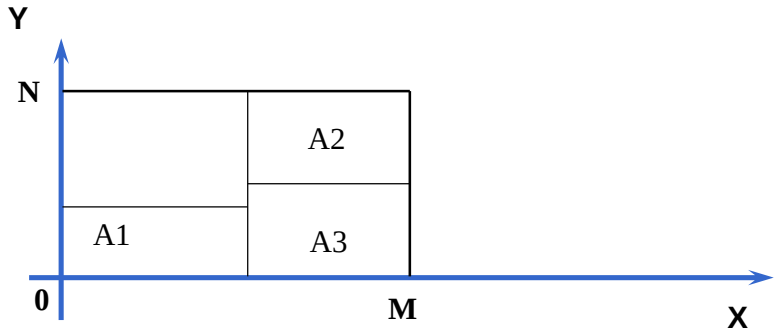
(Non interessa dove le attivita' sono collocate temporalmente nell'arco della giornata)






# 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.*






# *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*

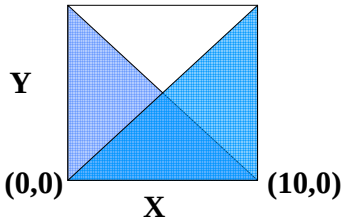


# VINCOLI RIDONDANTI

- 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.

**4, 2, 1, 0, 1, 0, 0, 0**

- Vincolo occurrences(V,L,N) (fd_global) impone che V compaia N volte nella lista L
- **magic_sequence([$X_0$,…,$X_n$]):-**

```
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([$X_0$,…,$X_n$]):-**

```
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**



# OTTIMIZZAZIONE

- In molte applicazioni non siamo interessati a soluzioni ammissibili, ma alla soluzione ottima rispetto a un certo criterio.
- ENUMERAZIONE ⇒ inefficiente
  - trova tutte le soluzioni ammissibili
  - scegli la migliore
- I linguaggi di Programmazione a Vincoli in generale incapsulano una forma di Branch and Bound
  - ogni volta che viene trovata una soluzione di costo C*, viene imposto un vincolo sulla parte rimanente dello spazio di ricerca. Il vincolo afferma che soluzioni successive (il cui costo è C) devono essere migliori della migliore soluzione trovata finora.

**C < C***

# Ottimizzazione in ECLiPSe

*minimize(Goal,Cost) o min_max(Goal,Cost)*

- *invocano Goal; fra le varie soluzioni ne forniscono una per cui Cost è minimo*
- *Usano algoritmi diversi, entrambi basati su branch & bound*



# Ottimizzazione: Esempio

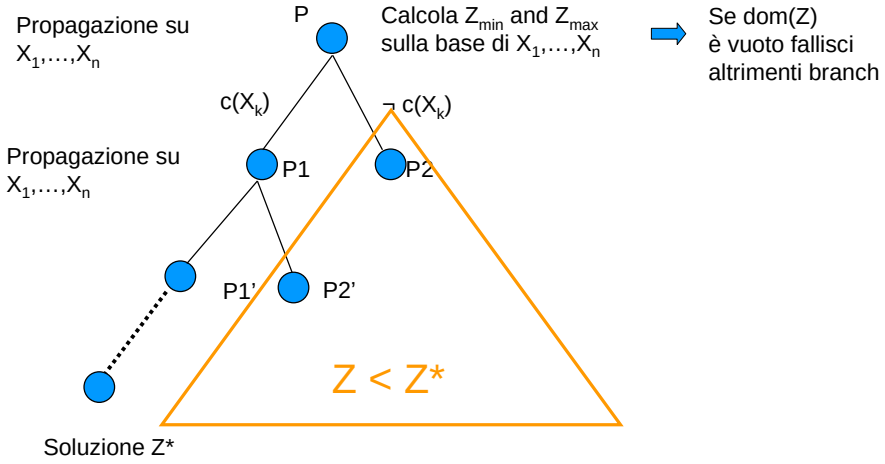
- *Es:* [A,B]::0..5, A+B#<5,A*B#= -C, min_max(labeling([A,B]),C).

1. *Impone i vincoli e propaga:*
   A{[0..4]}, B{[0..4]}, C{[-16..0]}
2. *esegue labeling e trova una soluzione:*
   A=0, B=0, C=0
3. *backtracking: torna alla situazione al passo 1*
4. *impone nuovo vincolo* C #< 0 *e propaga:*
   A{[1..3]}, B{[1..3]}, C{[-9 .. -1]}
5. *esegue labeling e trova una soluzione:*
   A=1, B=1, C=-1
6. *backtracking: torna alla situazione al passo 1*
7. *impone nuovo vincolo C #< -1 e propaga:*
   A{[1..3]}, B{[1..3]}, C{[-9 .. -2]}

- *… etc.*
- *Quando non trova più soluzioni, fornisce l'ultimo risultato ottenuto come soluzione*



# BRANCH & BOUND in PROGRAMMAZIONE A VINCOLI

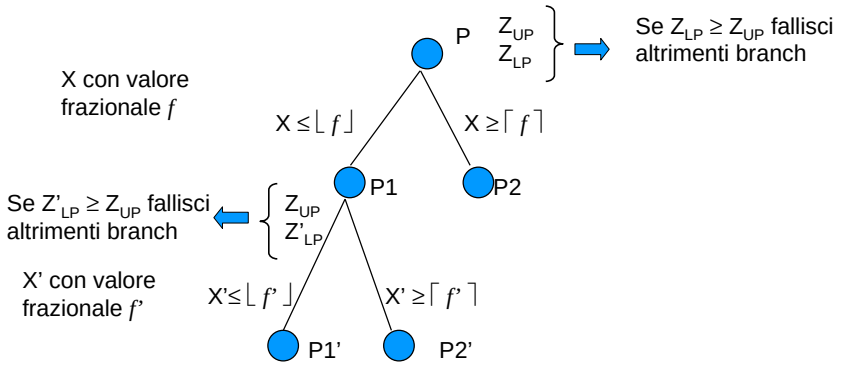





# BRANCH AND BOUND

- Si usa una variabile Z che rappresenta la funzione obiettivo e la si lega alle variabili decisionali, esempio somma di costi
- Se il legame tra Z e tali variabili e' forte, la propagazione elimina molti rami dall'albero, altrimenti, cercare una soluzione ottima è molto difficile
- Il vincolo aggiunto fa normalmente poca propagazione.
- Esempi
  - Per lo scheduling funziona bene (min la lunghezza dello schedule)
  - Per le somme di costi meno bene
  - Per la minimizzazione dei tempi di setup, non funziona affatto.

# BRANCH & BOUND in PROGRAMMAZIONE LINEARE INTERA





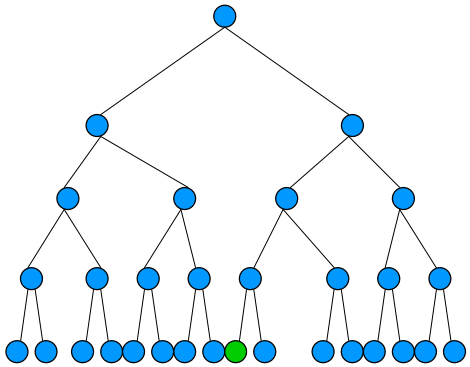

# OTTIMIZZAZIONE

- La Ricerca Operativa ha una lunga tradizione nella soluzione di problemi di ottimizzazione
- I metodi Branch & Bound si basano sulla soluzione ottima di un rilassamento del problema che produce un BOUND, ossia una valutazione ottimistica della funzione obiettivo
  - relaxation: same problem with some constraints relaxed
- Linea di ricerca particolarmente promettente: integrazione di tecniche di Ricerca Operativa nella programmazione a vincoli al fine di migliorarne l'efficienza nella risoluzione di problemi di ottimizzazione



# *Altri algoritmi di Search*

- *Esplorazione depth-1st: in profondità lungo il ramo di sinistra, poi torna indietro 1 passo, ecc.*
- *esplora le foglie di sinistra prima di quelle di destra.*
- *Supponiamo che l'euristica sia molto buona: in tutto il percorso compie solo 1 errore.*
  - *errore all'ultimo nodo => ☺*
  - *errore al primo => ☹*



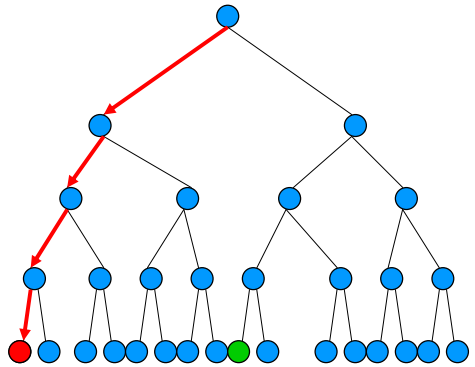
*Soluzione*



# *Limited Discrepancy Search*

- *Discrepanza=0*

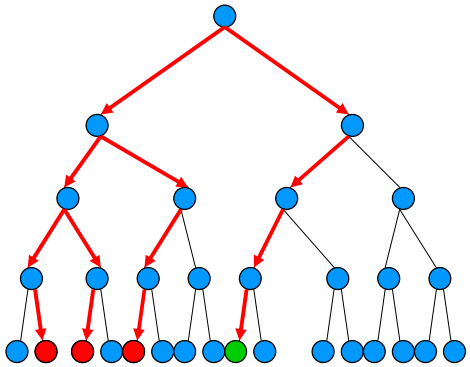
*seguo l'euristica*



*Soluzione*

# Limited Discrepancy Search

- *Discrepanza=1: Permetto che ci sia un errore nell'euristica.*
- *Prendo sempre il ramo di sinistra e prendo solo una volta il ramo di destra*



*Soluzione*



# fd_search

- *Libreria* fd_search*:*
  search(L, Arg, Select, Choice, Method, Opt)
- *L: Lista di variabili.*
- *Arg = 0, Opt=[] (per i nostri usi, V manuale per approfondimenti)*
- *Select: euristica di selezione della variabile.*
  - *input_order, first_fail, smallest, largest, occurrence, most_constrained, max_regret, anti_first_fail*
- *Choice: euristica di selezione del valore*
  - *indomain, indomain_min, indomain_max, indomain_middle, indomain_median, indomain_split, indomain_random, indomain_interval*
- *Method: Algoritmo di search*
  - *complete, bbs(Steps), lds(Disc), credit(Credit, bbs(Steps) or lds(Disc)), dbs(Level, bbs(Steps) or lds(Disc)), sbds*



# LDS in ECLiPSe

```
risolvi(L):-
  crea_domini(L),
  imponi_vincoli(L),
  search(L,0,most_constrained, indomain, lds(5),[]).
ottimizza(L):-
  crea_domini(L),
  imponi_vincoli(L),
  crea_obiettivo(L,F),
  minimize(search(L,0,most_constrained, indomain,
  lds(5),[]), F).
```

*Domanda: è completo così? Come posso renderlo completo?*



# Problemi sovravincolati

- *Alcuni problemi nella vita reale non hanno soluzione: sono sovravincolati*
- *CLP in questo caso dà come risposta semplicemente 'no'*
- *Questo in applicazioni reali non è sufficiente*
- *Come gestire questo problema?*

# Vincoli soft

- *Uno dei metodi è quello di usare vincoli soft (che possono essere rilassati)*
  - *cerchiamo di massimizzare i vincoli soft che sono soddisfatti*
- *I vincoli reificati possono essere usati come vincoli soft: massimizzo il numero di vincoli soddisfatti*
- *Es. Orario delle lezioni:*
  - *cerco di soddisfare le richieste degli insegnanti*

```
Mat #> 10, Fisica #< 14, …
```

  – *Con vincoli Soft:*

```
Mat #> 10 #<=>B1, Fisica#< 14 #<=> B2,
sumlist([B1,B2,…],Sum),
F #= -Sum,
minimize(labeling(…),F).
```



# Ancora sui vincoli reificati

- *I vincoli reificati possono servire per collegare due o più vincoli:*
  - **`X#>Y #<=> B1, X#>Z #<=> B2, X+Y#>0`**
    – *Ci sono altri connettori* `#\/, #/\, #=>`, …
    – *Si può scrivere anche* **`X#>Y #\/ X#>Z`**
- *Poi posso rilassarli insieme:*
  - **`Elettr#<10 #\/ Elettr#>12 #<=> B1, sumlist([B1, …],Sum), F #= -Sum, minimize(labeling(…),Sum).`**



# CONSTRAINT PROGRAMMING TOOLS

– CLP(R) ***[Jaffar et al. Trans. Progr. Lang and Sys. 92]***,
– Prolog III ***[Colmerauer CACM(33) 90]***,
– CHIP ***[Dincbas et al., JICSLP88]***,
– CLP(PB) ***[Bockmayer ICLP95]***,
– Ecl$^i$ps$^e$ ***[Wallace et al.97]***, Conjunto ***[Gervet, Constraints(1), 97]***
– *Oz* ***[Smolka JLP 91]***,
– *AKL* ***[Carlson, S.Haridi, S.Janson ILPS94]***,
– *CIAO* ***[Hermenegildo et al. PPCP94]***
– *ILOG* ***[Puget SPICIS94], [Puget, Leconte ILPS95]***
– *CHARME* ***[Bull Corporation 90]***
– ***SICStus***
– *many others…….*



# CONSTRAINT PROGRAMMING TOOLS

| | *Dichiarativi* | *a oggetti* |
|---|---|---|
| *Open source* | Ecl$^i$ps$^e$ ***[Wallace et al.97]*** ***[Smolka 91]*** ***[Hermenegildo et al. 94]*** | *Gecode* |
| *Commerciali* | ***CHIP*** SICStus | *ILOG* ***[Puget SPICIS94], [Puget, Leconte ILPS95]*** |

***… e molti altri***

# AI APPLICATIONS: MODELLING and SOLVING

- Ci focalizzeremo su CLP(FD)
- Per ogni applicazione, mostriamo la descrizione del problema, il modello CLP e la risoluzione.

- Applicazioni discusse:
  - *Scheduling - Timetabling - Resource Allocation* } optimization
  - *Routing* } optimization
  - *Packing - Cutting* } optimization
  - *Graphics - Vision* } feasibility
  - *Planning* } feasibility



# SCHEDULING - TIMETABLING - RESOURCE ALLOCATION

- Three applications with analogous features/constraints:
  - we will focus on scheduling. Same considerations for timetabling and resource allocation (easier problem)
- Scheduling is probably one of the most successful applications of CP to date
  - flexibility
  - generality
  - easy code
- NP-complete problem studied by the AI community since 80s



# SCHEDULING: problem definition

- Scheduling concerns the assignment of limited resources (machines, money, personnel) to activities (project phases, manufacturing, lessons) over time
- Constraints
  - temporal restrictions
    - ordering among activities
    - due dates - release dates
  - resource capacity
    - different types of resources
    - consumable/renewable resources
- Optimisation Criteria
    - makespan
    - resource balance
    - lateness on due dates
    - resource assignment cost

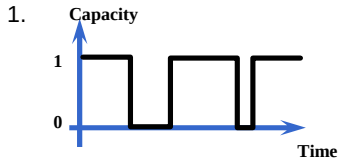


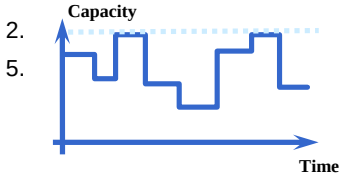
# SCHEDULING: Activities

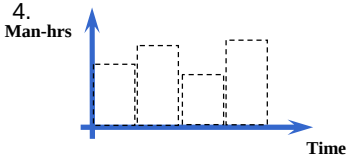
- Decision variables:
  - Activity start times
  - Activity end times
  - Activity resource assignments
  - Alternative activities (alternative routing)
- Activity types:
  - interval activity: cannot be interrupted
  - breakable activity: can be interrupted by breaks
  - preemptable activity: can be interrupted by other activities

# SCHEDULING: Resources

- Resource types:
  - 1. Unary resources
  - 2. Discrete resources
  - 3. State resources
  - 4. Energy resources
  - 5. Stock




# SCHEDULING: Simple Example

- 6 activities: each activity described by a predicate

  *task(NAME,DURATION,LISTofPRECEDINGTASKS,MACHINE).*

```
task(j1,3,[],m1).
task(j2,8,[],m1).
task(j3,8,[j4,j5],m1).
task(j4,6,[],m2).
task(j5,3,[j1],m2).
task(j6,4,[j1],m2).
```

- Machines are unary resources.
- Minimising the maximum ending time End is required.



# SCHEDULING: Simple Example

```
schedule(Data, End, TaskList):-
    makeTaskVariables(Data,End,TaskList),
    precedence(Data, TaskList),
    machines(Data, TaskList),
    minimize(labeling(TaskList),End).
makeTaskVariables([],_,[]).
makeTaskVariables([task(N,D,_,_)|T],End,[Start|Var]):-
    Start #<= End-D, Start #>=0,
    makeTaskVariables(T,End,Var).
precedence([],[]).
precedence([task(N,_,Prec,_)|T], [Start|Var]):-
   select_preceding_tasks(Prec,T,Var,PrecVars,PrecDurations),
   impose_constraints(Start,PrecVars,PrecDurations),
   precedence(T,Var).
impose_constraints(_,[],[]).
impose_constraints(Start,[Var|Other],[Dur|OtherDur]):-
   Var + Dur #<= Start,
   impose_constraints(Start,Other,OtherDur).
```



# SCHEDULING: Simple Example

```
schedule(Data, End, TaskList):-
    makeTaskVariables(Data,End,TaskList),
    precedence(Data, TaskList),
    machines(Data, TaskList),
    minimize(labeling(TaskList),End).
```

```
makeTaskVariables([],_,[]).
makeTaskVariables([task(N,D,_,_)|T],End,[Start|Var]):-
    Start #<= End-D, Start #>=0,
    makeTaskVariables(T,End,Var).
```

```
End :: 8..10000
```

```
[task(j1,3,[],m1),
 task(j2,8,[],m1),
 task(j3,8,[j4,j5],m1),
 task(j4,6,[],m2),
 task(j5,3,[j1],m2),
 task(j6,4,[j1],m2)]
```

```
[Start1 :: 0..+10000,
 Start2 :: 0..+10000,
 Start3 :: 0..+10000,
 Start4 :: 0..+10000,
 Start5 :: 0..+10000,
 Start6 :: 0..+10000]
```

## SCHEDULING: Simple Example

```
schedule(Data, End, TaskList):-
    makeTaskVariables(Data,End,TaskList),
    precedence(Data, TaskList),
    machines(Data, TaskList),
    minimize(labeling(TaskList),End).
```

```
precedence([],[]).
precedence([task(N,_,Prec,_)|T], [Start|Var]):-
  select_preceding_tasks(Prec,T,Var,PrecVars,PrecDurations),
  impose_constraints(Start,PrecVars,PrecDurations),
  precedence(T,Var).
impose_constraints(_,[],[]).
impose_constraints(Start,[Var|Other],[Dur|OtherDur]):-
   Var + Dur #<= Start,
   impose_constraints(Start,Other,OtherDur).
```



## SCHEDULING: Simple Example

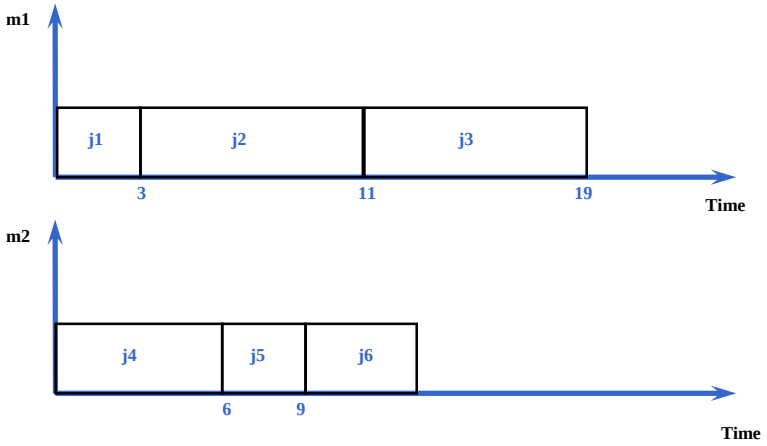
```
machines(Data, TaskList):-
 tasks_sharing_resource(Data,TaskList,SameResource,Durations),
 impose_cumulative(SameResource,Durations,Use).

impose_cumulative([],[],_).
impose_cumulative([ListSameRes|LSR],[Dur|D],[Use|U]):-
  cumulative(ListSameRes,Dur,Use,1),
impose_cumulative(LSR,D,U).
```

results in

```
cumulative([Start1,Start2,Start3],[3,8,8],[1,1,1],1)
cumulative([Start4,Start5,Start6],[6,3,4],[1,1,1],1)
```



## SCHEDULING: Optimal Solution






## SCHEDULING: Optimality

- **`minimize`**: finds the optimal solution (simple Branch & Bound)
- Minimization of the makespan: a heuristic which always selects the task which can be assigned first and assigns to the task the minimal bound is in general a good heuristics. As a choice point, delay the task.

```
labelTasks([]).
labelTasks(TaskList):-
    find_min_start(TaskList,First,MinStart,Others),
    label_earliest(TaskList,First,MinStart,Others).

label_earliest(TaskList,First,Min,Others):- % schedule the task
    First = Min,
    labelTasks(Others).
label_earliest(TaskList,First,Min,Others):- % delay the task
    First ≠ Min,
    labelTasks(TaskList).
```

## TIMETABLING: problem definition

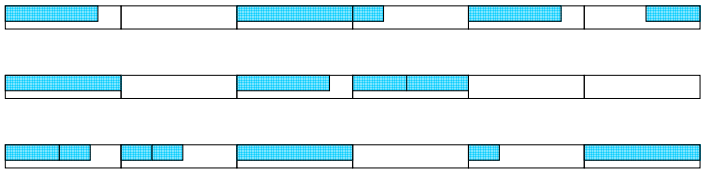
- Timetabling concerns the definitions of agenda (similar to scheduling)

- Constraints
  - temporal restrictions
    - ordering among activities
    - due dates - release dates
  - resource capacity
    - discrete resources
- Optimization Criteria
  - cost/preferences
  - resource balance



## TIMETABLING: simple example

- 4-Hours Slots - 1 to 4 Hours Courses
- Two courses cannot overlap
- A course must be contained in a single slot
- Preferences are associated with
  - Course-Slot assignments
  - Maximize Sum of preferences





## TIMETABLING: code with redundant constraints

```
timetable(Data,Tasks,MaxTime,Costs):-
    define_variable_start(Tasks,MaxTime),
    define_variable_singleHours(Data,SingleHours),
    define_variable_courses3_4Hours(Data, Courses34Hours),
    impose_cumulative(Tasks),
    alldifferent(SingleHours),
    alldifferent(Courses34Hours),
    minimize(labeling(Tasks),Cost).
```

*redundant constraints*

- Redundant variables:
  - start times
  - single hours
  - courses lasting 3 or 4 hours

***linked each other:***

***exchange propagation results***



## TIMETABLING: optimality & search

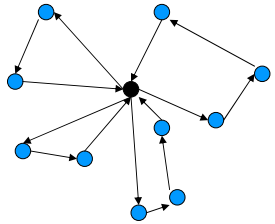
- Search strategy: when coping with objective functions, we can exploit information on costs for defining a good search strategy.

- Example:
  - Choose the variable with max value of regret
    - Regret: difference between the first and the second best on each row of the cost matrix.
    - Combination of regret and first-fail
  - Choose the value associated with the minimum cost
- Example: based on the optimal solution of a relaxation
  - Choose the variable with max value of regret
  - Choose the solution of the relaxation

## ROUTING: problem definition

- Routing concerns the problem of finding a set of routes to be covered by a set of vehicles visiting a set of cities/customers once starting and ending at one (n) depot(s).

- Constraints
  - temporal restrictions:
    - time windows
    - maximal duration of a travel
  - vehicle capacity
  - customer demands
- Optimization Criteria
  - number of vehicles
  - travel cost
  - travel time





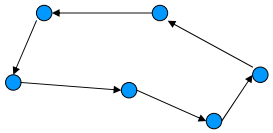
## ROUTING: problem definition

- Routing has been solved within OR community by using
  - Branch & Bound approaches
  - Dynamic Programming
  - Local Search techniques
  - Branch & Cut
  - Column generation
- Routing has been solved within CP community by using
  - Branch & Bound approaches
  - Local Search techniques embedded in CP
- Basic component: Travelling Salesman Problem (TSP) and its time constrained variant.



## TSP: problem definition

- TSP is the problem of finding a minimum cost tour covering a set of nodes once.



- No subtours are allowed
- TSPTW: Time windows are associated to each node. Early arrival is allowed. Late arrival is not permitted
- Even finding an Hamiltonian Circuit (no costs) is NP-complete



## TSP: CP model

- Variable associated to each node. The domain of each variable contains possible next nodes to be visited
- N nodes ⟶ N + 1 variables $\mathbf{Next_i}$ (duplicate the depot)
- For all **i**: $\mathbf{Next_i \neq i}$
- $\mathbf{nocycle([Next_0,\ldots Next_n])}$
- $\mathbf{alldifferent([Next_0,\ldots Next_n])}$
- Cost $\mathbf{c_{ij}}$ if $\mathbf{Next_i = j}$
- In some models, we can find the redundant variables **Prev** indicating a node predecessor.

## TSP: code

```
tsp(Data,Next,Costs):-
      remove_arcs_to_self(Next),
      nocycle(Next),
      alldifferent(Next),
      create_objective(Next,Costs,Z),
      minimize(labeling(Next),Z).
```

- **`nocycle`**: symbolic constraint that ensures that no subtour is present in the solution.
- **`create_objective`**: creates **`Costs`** variables, imposes an **`element`** constraint between the set of **`Next`** variables and **`Costs`** variables, and creates a variable **`Z`** representing the objective function summing costs corresponding to assignments



## TSP: results

- Pure CP implementations: still far from the state of the art OR approaches.
- Integration of OR techniques in CP: better results
  - local search
  - optimal solution of relaxations
    - Lagrangean relaxation
    - Assignment Problem
    - Minimum Spanning Arborescence
  - search strategies based on these relaxations
    - subtour elimination
- Addition of Time Windows in OR approaches requires to re-write major code parts while in CP comes for free.



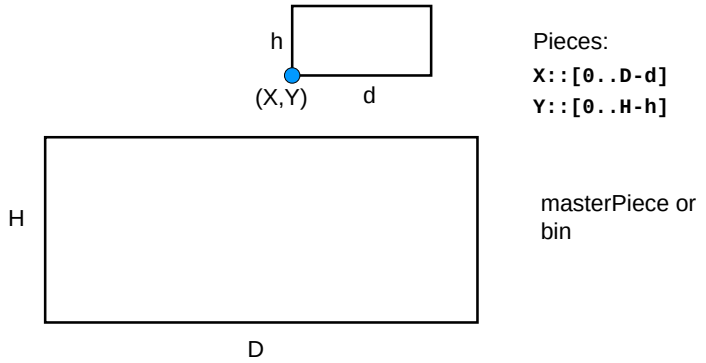
## CUTTING & PACKING: problem definition

- Packing concerns the placement of a number of squares (of different sizes) in one or more larger boxes in such a way that squares do not overlap and minimizing the empty space
- Cutting is the problem of finding cuts of a master piece in order to obtain a given number of pieces with fixed dimensions, minimizing residues
- Many variants:
  - strip packing
  - guillottine cuts
  - rotations allowed
  - 1 dimension - 2 dimensions - 3 dimensions - 4 dimensions



## 2-D PACKING: CP model

- For each square to be packed, we have a couple of variables representing the coordinates of the bottom-left point of the square



## 2-D PACKING: CP model

- Constraints:
  - non overlapping constraints: given two squares whose coordinates are (**X1**, **Y1**) and (**X2**, **Y2**) and dimensions **D1**, **H1** and **D2**, **H2** respectively
- **X1+D1 ≤ X2 OR Y1+H1 ≤ Y2 OR X2+D2 ≤ X1 OR Y2+H2 ≤ Y1**
- Very hard form of disjunction: no propagation even after instantiation
- Redundant constraints can help:
- **cumulative(Xcoordinates,XDimension,Ydimension,H)**
  **cumulative(Ycoordinates,YDimension,Xdimension,D)**



## PACKING: code

```
packing(Data,Xcoords,Ycoords,D,H):-
        create_variables(Data,Xcoords,Ycoords,D,H),
        state_disjunctive(Data,Xcoords,Ycoords),
        state_cumulatives(Data,Xcoords,Ycoords,D,H),

        create_objective(Xcoords,Ycoords,D,H,Z),
        minimize(label_squares(Xcoords,Ycoords),Z).
```

- **create_objective**: creates a variable representing the spare space (or the number of bins if more than one is present)
- **label_squares** selects bigger squares first and assigns the coordinates in order to minimize spare space.



## MODEL BASED VISION
## VISUAL SEARCH: definition

- Visual Search in model based vision concerns the problem of recognizing an object in a scene given its model

- How to describe the model
- How to perform the mapping

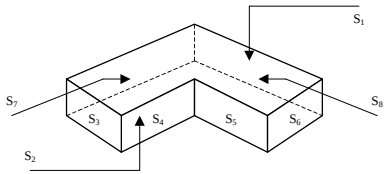
} *Both problems can be solved with constraints*

- MODEL: Constraint graph
  - Nodes: object parts
  - Arcs (constraints): their relations
- RECOGNITION: Constraint satisfaction

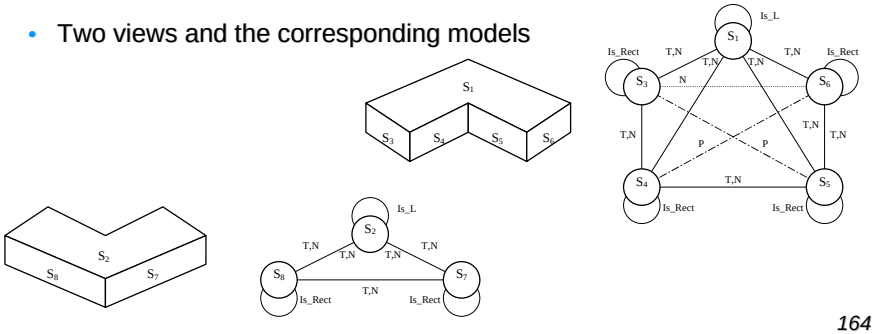


## OBJECT RECOGNITION

- 3-D OBJECT




- Two views and the corresponding models



# CONSTRAINT-BASED OBJECT RECOGNITION

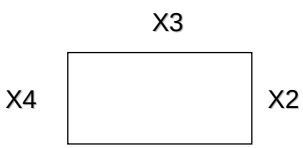
- In order to recognize an object, a low level vision system should extract from the image some visual features (surfaces/edges)
- Constraint satisfaction techniques can be applied in order to recognize the object in the scene.
- The object is recognized if the extracted features satisfy the constraints contained in the model.
- Constraints allow to reduce the search space to be explored.

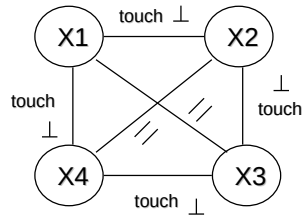


# EXAMPLE

- Rectangle shape

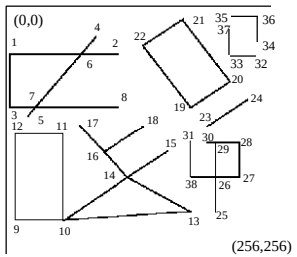
  *Four straight edges (variables), parallel two by two, which mutually touch themselves with a 90 degree angle ...*

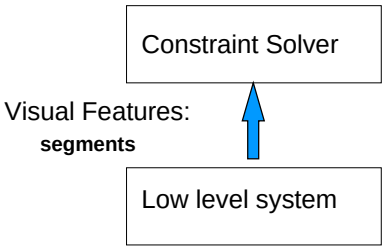





# EXAMPLE (2)




*rectangle CP model*

```
touch(X1,X2), touch(X2,X3), touch(X3,X4), touch(X1,X4),
perpend(X1,X2), perpend(X2,X3), perpend(X3,X4), perpend(X1,X4),
same_len(X2,X4), same_len(X1,X3), parallel(X2,X4), parallel(X1,X3)
```



# CP model: USER DEFINED CONSTRAINTS

*rectangle CP model*

```
recognize([X1,X2,X3,X4]):-
    X1,X2,X3,X4::[s1,s2,…,sn],
    touch(X1,X2), touch(X2,X3), touch(X3,X4), touch(X1,X4),
    perpend(X1,X2), perpend(X2,X3), perpend(X3,X4), perpend(X1,X4),
    same_len(X2,X4), same_len(X1,X3),
    parallel(X2,X4), parallel(X1,X3),
    labeling([X1,X2,X3,X4]).
```

- Give the declarative and operational semantics of the constraints: segments are described as facts: `segment(name,X1,Y1,X2,Y2)`
- In all CP languages there are tools that allow new constraints to be defined.
- An example in the CLP(FD) library of ECL$^i$PS$^e$

# CP model: USER DEFINED CONSTRAINTS

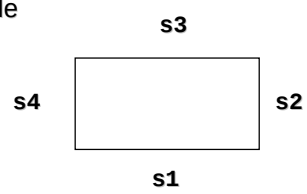
```
touch(X1,X2) :-
      dvar_domain(X1,D1),
      dvar_domain(X2,D2),
      arc_cons_1(D1,D2,D1new),  % user defined propagation
      (dom_compare(>,D1,D1new) -> dvar_update(X1,D1new); true),
      arc_cons_2(D1new,D2,D2new), % user defined propagation
      (dom_compare(>,D2,D2new) -> dvar_update(X2,D2new); true),
      (var(X1),var(X2)
        ->   suspend(touch(X1,X2),3,[X1,X2]->fd:any)
        ;    true),
      wake.
```

- After the propagation, the constraint if not solved is suspended and awaked each time an event ***any of fd*** on one of the variables **(X1,X2)** happens.



# CP model: SYMMETRIES

- Problem symmetries: arise when some permutations of the variables map a solution onto another solution.
- Four segments forming a rectangle
- One solution:
  - **X1 = s1**
  - **X2 = s2**
  - **X3 = s3**
  - **X4 = s4**




- Other identical solutions:
  - **X1 = s2  X1 = s3  X1 = s4**
  - **X2 = s3  X2 = s4  X2 = s1**
  - **X3 = s4  X3 = s1  X3 = s2**
  - **X4 = s1  X4 = s2  X4 = s3**

*Time lost to look for already found solutions. Remove symmetries by imposing additional constraints*

***[Freuder AAAI91], [Puget ISMIS93], [Crawford et al. KR96], [Meseguer, Torras IJCAI99].***



# Modello CP: SIMMETRIE

- Problema: si perde tempo per esplorare stati equivalenti che non aggiungono alcuna informazione.
- Metodi per rimuovere le simmetrie:
  - aggiungere vincoli al modello (esempio ordinamenti tra variabili)
  - aggiungere vincoli dinamicamente nel corso della ricerca
  - modificare la strategia di ricerca per rimuovere le simmetrie.
- Argomento su cui vi è una ricerca molto attiva



# ESEMPIO SEMPLICE

- Pigeonhole problem: Abbiamo *n*-1 gabbie in cui devono essere collocati *n* piccioni uno per gabbia.
- Problema chiaramente insolubile ma abbiamo simmetrie di permutazione

| | *Con simmetrie* | | *NO simmetrie* | |
|---|---|---|---|---|
| *N piccioni* | *Size Tree* | *Tempo (s)* | *Size Tree* | *Tempo (s)* |
| *6* | *119* | *0.213* | *15* | *0.042* |
| *7* | *719* | *1.031* | *31* | *0.064* |
| *8* | *5039* | *7.619* | *63* | *0.102* |
| *9* | *40319* | *61.902* | *127* | *0.228* |

# COME RIMUOVERE LE SIMMETRIE

- Abbiamo *n* variabili $P_1$, … $P_n$ che rappresentano i piccioni e un dominio contenente le gabbie da 1 a *n*-1
- Usiamo tutti vincoli di diverso binari (per vedere l'impatto delle simmetrie)
- Simmetrie di permutazione *n* variabili *n*! stati simmetrici
- Si può imporre $P_1 < P_2$, $P_2 < P_3$ ….$P_{n-1} < P_n$ e si rimuovono tutte le simmetrie



# SPORT SCHEDULING

- Dobbiamo allocare delle partite in diverse settimane
- Ci sono *n* squadre (nell'esempio da 0 a 7). Tutte le squadre devono giocare contro tutte le altre, quindi in totale ho *n**(*n*-1)/2 partite. Devo allocare una partita per ogni cella in modo che in ogni settimana una squadra giochi una sola volta.

| | *Week 1* | *Week 2* | *Week 3* | *Week 4* | *Week 5* | *Week 6* | *Week 7* |
|---|---|---|---|---|---|---|---|
| *Game 1* | *0 vs 1* | *0 vs 2* | *4 vs 7* | *3 vs 6* | *3 vs 7* | *1 vs 5* | *2 vs 4* |
| *Game 2* | *2 vs 3* | *1 vs 7* | *0 vs 3* | *5 vs 7* | *1 vs 4* | *0 vs 6* | *5 vs 6* |
| *Game 3* | *4 vs 5* | *3 vs 5* | *1 vs 6* | *0 vs 4* | *2 vs 6* | *2 vs 7* | *0 vs 7* |
| *Game 4* | *6 vs 7* | *4 vs 6* | *2 vs 5* | *1 vs 2* | *0 vs 5* | *3 vs 4* | *1 vs 3* |



# SPORT SCHEDULING: simmetrie

- *Le settimane sono simmetriche*
- *I Game sono simmetrici*

| | *Week 1* | *Week 2* | *Week 3* | *Week 4* | *Week 5* | *Week 6* | *Week 7* |
|---|---|---|---|---|---|---|---|
| *Game 1* | *0 vs 1* | *0 vs 2* | *4 vs 7* | *3 vs 6* | *3 vs 7* | *1 vs 5* | *2 vs 4* |
| *Game 2* | *2 vs 3* | *1 vs 7* | *0 vs 3* | *5 vs 7* | *1 vs 4* | *0 vs 6* | *5 vs 6* |
| *Game 3* | *4 vs 5* | *3 vs 5* | *1 vs 6* | *0 vs 4* | *2 vs 6* | *2 vs 7* | *0 vs 7* |
| *Game 4* | *6 vs 7* | *4 vs 6* | *2 vs 5* | *1 vs 2* | *0 vs 5* | *3 vs 4* | *1 vs 3* |



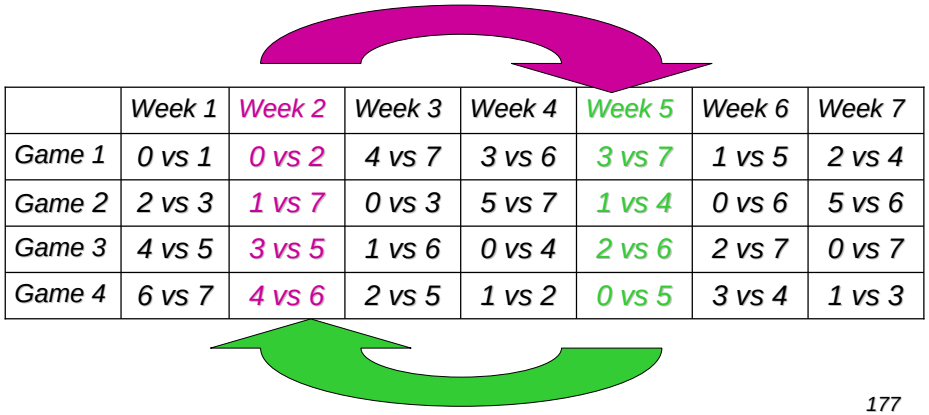
# SPORT SCHEDULING: simmetrie

- *Le settimane sono simmetriche*
- *I Game sono simmetrici*

| | *Week 1* | *Week 2* | *Week 3* | *Week 4* | *Week 5* | *Week 6* | *Week 7* |
|---|---|---|---|---|---|---|---|
| *Game 1* | *0 vs 1* | *0 vs 2* | *4 vs 7* | *3 vs 6* | *3 vs 7* | *1 vs 5* | *2 vs 4* |
| *Game 2* | *2 vs 3* | *1 vs 7* | *0 vs 3* | *5 vs 7* | *1 vs 4* | *0 vs 6* | *5 vs 6* |
| *Game 3* | *4 vs 5* | *3 vs 5* | *1 vs 6* | *0 vs 4* | *2 vs 6* | *2 vs 7* | *0 vs 7* |
| *Game 4* | *6 vs 7* | *4 vs 6* | *2 vs 5* | *1 vs 2* | *0 vs 5* | *3 vs 4* | *1 vs 3* |

## SPORT SCHEDULING: simmetrie

- *Cambiando l'ordine di due settimane trovo una soluzione equivalente*



| | Week 1 | Week 2 | Week 3 | Week 4 | Week 5 | Week 6 | Week 7 |
|---|---|---|---|---|---|---|---|
| Game 1 | 0 vs 1 | 0 vs 2 | 4 vs 7 | 3 vs 6 | 3 vs 7 | 1 vs 5 | 2 vs 4 |
| Game 2 | 2 vs 3 | 1 vs 7 | 0 vs 3 | 5 vs 7 | 1 vs 4 | 0 vs 6 | 5 vs 6 |
| Game 3 | 4 vs 5 | 3 vs 5 | 1 vs 6 | 0 vs 4 | 2 vs 6 | 2 vs 7 | 0 vs 7 |
| Game 4 | 6 vs 7 | 4 vs 6 | 2 vs 5 | 1 vs 2 | 0 vs 5 | 3 vs 4 | 1 vs 3 |



## SPORT SCHEDULING: simmetrie

- *Cambiando l'ordine di due game trovo una soluzione equivalente*

| | Week 1 | Week 2 | Week 3 | Week 4 | Week 5 | Week 6 | Week 7 |
|---|---|---|---|---|---|---|---|
| Game 1 | 0 vs 1 | 3 vs 7 | 4 vs 7 | 3 vs 6 | 0 vs 2 | 1 vs 5 | 2 vs 4 |
| Game 2 | 2 vs 3 | 1 vs 4 | 0 vs 3 | 5 vs 7 | 1 vs 7 | 0 vs 6 | 5 vs 6 |
| Game 3 | 4 vs 5 | 2 vs 6 | 1 vs 6 | 0 vs 4 | 3 vs 5 | 2 vs 7 | 0 vs 7 |
| Game 4 | 6 vs 7 | 0 vs 5 | 2 vs 5 | 1 vs 2 | 4 vs 6 | 3 vs 4 | 1 vs 3 |



## PERCHE' PREOCCUPARSENE

- Per una matrice ***nxm*** che ha simmetrie di riga e di colonna ci sono ***n!*m!*** simmetrie (super-esponenziale)
- Per ogni soluzione (totale o parziale) ce ne sono ***n!*m!*** simmetriche, ma anche per ogni fallimento
- Eliminare tutte le simmetrie non è facile
- Ci si accontenta spesso di eliminarne alcune, in modo da ottenere un albero ridotto.



## COME ELIMINARLE: VINCOLI AGGIUNTIVI NEL MODELLO

- Solo simmetrie di riga, posso eliminarle con un ordinamento totale sulle righe:
- Forzare le righe ad essere lessicograficamente ordinate rompe tutte le simmetrie di riga
- In ECLiPSe: vincolo **`lexico_le`**

lexicographic ordering

$[A\ B\ C] \leq_{lex} [D\ E\ F] \leq_{lex} [G\ H\ I]$

| | | |
|---|---|---|
| *A* | *B* | *C* |
| *D* | *E* | *F* |
| *G* | *H* | *I* |

anti-lexicographic ordering

$[G\ H\ I] \leq_{lex} [D\ E\ F] \leq_{lex} [A\ B\ C]$

## SIMMETRIE DI RIGA E COLONNA

- Riusciamo a eliminare le simmetrie di riga e colonna singolarmente, ma se compaiono insieme, imporre gli ordinamenti su righe e colonne non elimina tutte le simmetrie.



## Good News ☺

- Una simmetria definisce una classe di equivalenza
  - Due assegnamenti sono equivalenti se una simmetria mappa un assegnamento in un altro.
- Ogni simmetria ha ALMENO UN elemento in cui SIA le righe SIA le colonne sono ordinate in senso lessicografico
  - Puo' non esistere un elemento con le righe ordinate in senso lessicografico e le colonne in senso antilessicografico
- Per eliminare le simmetrie di riga e colonna possiamo ordinare lessicograficamente righe e colonne *(double-lex)*



## Bad news ☹

- Una simmetria puo' avere piu' di un elemento con righe e colonne ordinate lessicograficamente
- Double-lex non elimina tutte le simmetrie

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |
| 1 | 0 |

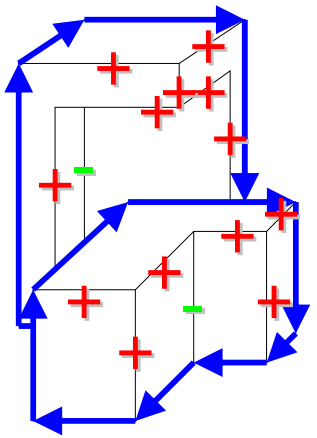
⇐ swap the columns
swap row 1 and row 3 ⇒

| | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 1 | 0 |



## *MODEL BASED VISION (2) OBJECT RECOGNITION*

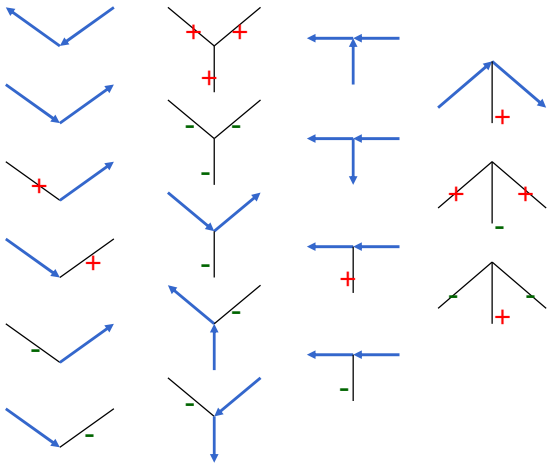
- *Viceversa, può essere richiesto di dare un significato ad un oggetto qualunque.*
- *Es, spiegare gli spigoli di una figura 3D*
  - *quali sono concavi - o convessi +*
  - *quali delimitano una figura →*



# OBJECT RECOGNITION

*Waltz notò che le combinazioni sono in numero limitato*

- *Variabili: segmenti nella figura*
- *Domini*: {+,-,←,→}
- *Vincoli: combinazioni possibili*





# AI PLANNING: definition

- Planning concerns the problem of finding a chain of actions that achieve a given goal starting from a well known initial state. Actions are described with a set of preconditions (requirements) and postconditions (effects)

- Constraints
  - Plan constraints
    - temporal constraints (ordering)
    - designation and co-designation constraints
    - resource constraints
    - domain dependent constraints
  - Plan construction constraints
    - threats
    - open condition achievement



# CONSTRAINTS ON THE PLAN

- Temporal constraints
  - qualitative → action A before action B
  - quantitative → action A in [10..30]
- Resource constraints
  - consumable resources
  - shared resources
  - renewable resources
- Domain-dependent constraints
  - priority among resources
  - destructive actions on object A always after any other action on A

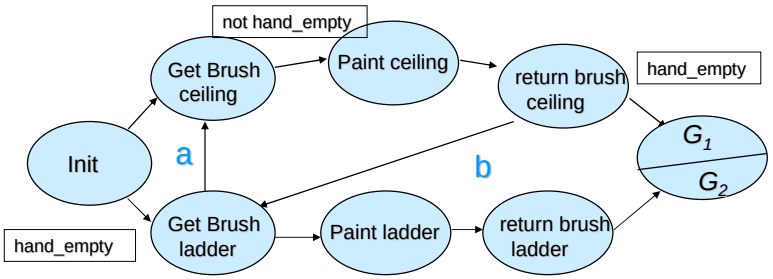


# CONSTRAINTS ON PLAN CONSTRUCTION

- During the plan construction a set of decisions should be taken:
  - threat resolution
  - open condition achievement
- Good technique: least commitment. Decisions are delayed as much as possible in order to perform only consistent choices.
- Passive postponement vs. active postponement
- Each decision is represented by a variable whose domain contains possible solution to the pending decision. Constraints take into account interaction among decisions.

## THREAT RESOLUTION

- Threat: conflict occurring when the effects of an action A threat the preconditions of an action B




Robot with one hand: threat

a promotion: getBrushLadder < getBrushCeiling

b demotion: returnBrushCeiling < getBrushLadder



## THREAT RESOLUTION

- Variable associated with the threat:
  - T1 :: [ *getBrushLadder* < *getBrushCeiling*, *returnBrushCeiling* < *getBrushLadder* ]
  - Threat variables linked by:
    - *incompatibility constraints*
    - *subsumption constraints*
  - Example: if a threat T2 can be solved only by the constraint *getBrushCeiling* < *getBrushLadder*, the only way of solving T1 becomes *returnBrushCeiling* < *getBrushLadder*
- Propagation of consequences of the decisions. Reduction of the search space and trashing avoided
- More complex solver: user defined constraints.



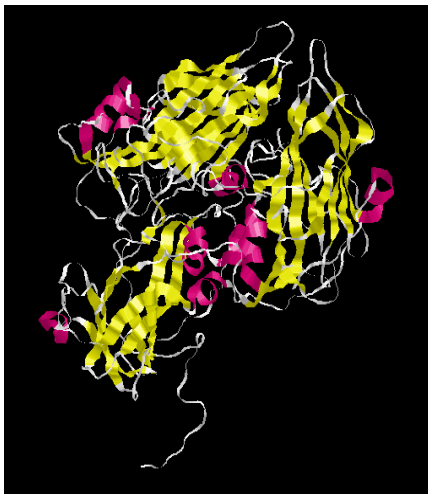
## PLANNERS based on CP/CSPs

- ParcPlan ***[Lever, Richards, ISMIS94]***
- PlaNet ***[Barruffi, Milano ECAI98], [Barruffi et. Al. ECP99]***
- O-Plan ***[Tate, Drabble, Dalton, TR Univ. Edinburg 95]***
- Molgen ***[Stefik AIJ(16), 81]***
- DEVISER ***[Vere IJCAI85]***
- Descartes ***[Joslin PhD, 95], [Joslin, Pollac, EWSP96]***
- WatPlan ***[Yang, AIJ(58), 92]***
- SLNP ***[McAllester, Rosenblitt Nat.Conf AI 91]***
- FSNLP ***[Yang, Chan AIPS94]***
- Kambampati ***[TR Arizona State Univ. 96]***
- SIPE ***[Wilkins AI (22) 84]***



## *PROTEIN FOLDING*

- *Una proteina può essere vista come una sequenza di amminoacidi*
- *Coppie di amminoacidi si attraggono o respingono*
- *La proteina si avvolge in una forma con energia minima*
- *Per trovare il modo in cui si avvolge una proteina, si devono svolgere analisi chimiche*



# PROTEIN FOLDING

- *Le posizioni in cui un amminoacido si può trovare sono discrete*
- *Se due amminoacidi sono vicini, si attraggono o respingono a seconda dei valori di una tabella data*

| | CYS | MET | PHE | ILE | LEU |
|---|---|---|---|---|---|
| CYS | -3.477 | -2.240 | -2.424 | -2.410 | -2.343 |
| MET | -2.240 | -1.901 | -2.304 | -2.286 | -2.208 |
| PHE | -2.424 | -2.304 | -2.467 | -2.530 | -2.491 |
| ILE | -2.410 | -2.286 | -2.530 | -2.691 | -2.647 |
| LEU | -2.343 | -2.208 | -2.491 | -2.647 | -2.501 |
| … | | | | | |



# Esempio in ECLiPSe

- *Consideriamo il caso 2D: gli amminoacidi si possono disporre in una griglia*
- *2 tipi di amminoacidi: 1 e 0. Gli 0 sono inerti, gli 1 si attirano, se sono in posizioni vicine*
- *Prendiamo in ingresso la lista che descrive la proteina (es. [1,0,0,1,0,1,1,0,1]).*
- *Forniamo in uscita una lista di uguale lunghezza che descrive la posizione di ogni amminoacido nello spazio*
  - *Ogni amminoacido ha una X ed una Y (intere)*
  - *Si può riassumere in un'unico valore K=Y*P+X (se P=Xmax+1)*

| *jP* | *jP*+1 | | (*j*+1)*P*-1 |
|---|---|---|---|
| … | … | … | … |
| 2*P* | 2*P*+1 | … | 3*P*-1 |
| *P* | *P*+1 | … | 2*P*-1 |
| 0 | 1 | … | *P*-1 |



# Esempio di codice

```
protein(Lin,Lout) :-
   length(Lin,N), P is N+1,
   length(Lout,N),
   connected(Lout,Steps,P),    /*Un elemento è connesso
   al successivo. Steps=lista di differenze fra le pos
   degli amminoacidi */
   alldifferent(Lout),   /*Una pos occupata da 1 solo
   amminoacido*/
   func(Lout,Lin,Lbool,P),     /*Creo una lista di var
   bool per ogni coppia di amminoacidi: se gli
   amminoacidi sono vicini, allora il bool=1*/
   sumlist(Lbool,F),     F1 #= -F,
   minimize(labeling(Steps),F1).
```



# Altri predicati …

```
% func(Lout,Lin,Lbool,P)
func([_,_],_,[0],_) :- !.
func([H,H1,H2|T],[1,Xin1,Xin2|Tin],Lbool,P)
   :- !,
% Non serve mettere nella funzione obiettivo
   due successivi. Anche il primo ed il terzo
   non possono essere vicini
   cal_fun(H,T,1,Tin,L1,P),
   func([H1,H2|T],[Xin1,Xin2|Tin],L2,P),
   append(L1,L2,Lbool).
func([_|T],[0|Tin],Lbool,P) :-
   func(T,Tin,Lbool,P).
cal_fun(_,[],_,[],[],_).
cal_fun(X,[H|T],1,[1|Tin],[Bool|R],P) :-!,
   X #= H+1#<=>B1, X #= H-1 #<=> B2,
   X #= H+P#<=>B3, X #= H-P #<=> B4,
   sumlist([B1,B2,B3,B4],Bool), Bool::0..1,
   cal_fun(X,T,1,Tin,R,P).
cal_fun(X,[_|T],1,[0|Tin],R,P) :-
   cal_fun(X,T,1,Tin,R,P).
```

```
connected([_],[],_):- !.
connected([A,B|C],[Step|Steps],P):-
   MP is -P,
   Step :: [MP,-1,1,P],
   A #= B+Step,
   connected([B|C],Steps,P).
```

## OTHER APPLICATIONS of CP

- Constraint Databases
- Spreadsheet
- Robotics/Control
- Diagnosis
- Test data generation
- Circuit Verification
- Natural Language
- Graphical Interfaces
- Graphical Editors
- Biology (DNA sequencing)
- Qualitative reasoning
- Temporal reasoning
- SAT
- Other LSCO problems



## *Timetabling*

- *Si devono assegnare ad ogni corso un'aula ed un orario*
- *Ho aule di diversa capacità, con servizi (videoproiettore fisso, laboratori, …)*
- *Ogni corso ha un docente e viene seguito da diversi gruppi di studenti.*
- *Non si devono sovrapporre i corsi di uno stesso docente, seguiti dagli stessi studenti o che si trovano nella stessa aula*
- *Per ogni corso so quante ore fa alla settimana*
- *I docenti possono esprimere preferenze sugli orari (nel limite del possibile). Alta priorità ai docenti a contratto*



## *Specifiche*

- *Corsi*
  - *corso(sistemioperativi,stefanelli,130,[[info,2],[info,4],[ele,3],[tlc,3],[auto,4],[ele,5]],7,_,3,2,3,"Sistemi Operativi","http://www.ing.unife.it/informatica/SO-2/").*
  - *corso(strument_misure_elettr,corticelli,180,[[info,2],[auto,2],[tlc,2],[ele,2]],3,_,1,2,3,"Strumentazione e misure elettroniche",_).*
  - *corso(strument_misure_elettr_lab,corticelli,120,[[info,2],[auto,2],[tlc,2],[ele,2]],4,ele,1,4,4,"Strumentazione e misure elettroniche",_).*
  - *corso(inglese_turno1,inlingua,50,[[info,1],[auto,1]],4,_,2,1,2,"Inglese",_).*
  - *corso(inglese_turno2,inlingua,50,[[tlc,1],[ele,1]], 4,_,2,1,2,"Inglese",_).*
- *Aule*
  - *aula(1,250,n,_,s,s,_,_).*
  - *aula(lab_info,64, s,info,n,s, "http://www.ing.unife.it/sidi/cs_lab/cs_lab.htm","Laboratorio di Informatica Grande").*



## *Domini delle variabili Start*

- *Attività fittizie*
  - *pausa pranzo*
  - *fine giornata*

| | | Lun | Mar | Mer | Gio | Ven |
|---|---|---|---|---|---|---|
| 08:30 | 09:30 | 1 | 13 | 25 | 37 | 49 |
| 09:30 | 10:30 | 2 | 14 | 26 | 38 | 50 |
| 10:30 | 11:30 | 3 | 15 | 27 | 39 | 51 |
| 11:30 | 12:30 | 4 | 16 | 28 | 40 | 52 |
| 12:30 | 13:30 | 5 | 17 | 29 | 41 | 53 |
| 13:30 | 14:00 | 6 | 18 | 30 | 42 | 54 |
| 14:00 | 15:00 | 7 | 19 | 31 | 43 | 55 |
| 15:00 | 16:00 | 8 | 20 | 32 | 44 | 56 |
| 16:00 | 17:00 | 9 | 21 | 33 | 45 | 57 |
| 17:00 | 18:00 | 10 | 22 | 34 | 46 | 58 |
| 18:00 | 19:00 | 11 | 23 | 35 | 47 | 59 |
| | | 12 | 24 | 36 | 48 | 60 |

# Specifiche

| | Lunedì | Martedì | Mercoledì | Giovedì | Venerdì |
|---|---|---|---|---|---|
| 8.30 – 9.30 | A | B | A | A | D |
| 9.30 – 10.30 | A | B | A | A | D |
| 10.30 – 11.30 | A | B | C | B | G |
| 11.30 – 12.30 | B | C | C | B | G |
| 12.30 – 13.30 | B | C | C | D | F |
| | | | | | |
| 14 – 15 | C | F | G | D | F |
| 15 – 16 | C | F | G | G | E |
| 16 - 17 | D | F | G | G | E |
| 17 - 18 | D | E | E | F | E |
| 18 - 19 | D | E | E | F | |

- *Normalmente, i corsi devono stare in un "turno"*
- *In questo modo o due corsi si sovrappongono totalmente o non si sovrappongono*
- *Alcuni corsi sono in comunanza con altri CdL (Informatica a scienze, meccanica)*



# Implementazione vincolo turni

- *Propia:*

```
turni_def(Turno,Ora1,
   O2,O3) infers fd.

turni_def(a,25,37,1).
turni_def(b,4,39,13).
turni_def(c,7,16,27).
turni_def(d,41,49,9).
turni_def(e,22,34,56)
turni_def(f,46,53,19)
turni_def(g,44,51,31)
```

| | | Lun | Mar | Mer | Gio | Ven |
|---|---|---|---|---|---|---|
| 08:30 | 09:30 | 1 | 13 | 25 | 37 | 49 |
| 09:30 | 10:30 | 2 | 14 | 26 | 38 | 50 |
| 10:30 | 11:30 | 3 | 15 | 27 | 39 | 51 |
| 11:30 | 12:30 | 4 | 16 | 28 | 40 | 52 |
| 12:30 | 13:30 | 5 | 17 | 29 | 41 | 53 |
| 13:30 | 14:00 | 6 | 18 | 30 | 42 | 54 |
| 14:00 | 15:00 | 7 | 19 | 31 | 43 | 55 |
| 15:00 | 16:00 | 8 | 20 | 32 | 44 | 56 |
| 16:00 | 17:00 | 9 | 21 | 33 | 45 | 57 |
| 17:00 | 18:00 | 10 | 22 | 34 | 46 | 58 |
| 18:00 | 19:00 | 11 | 23 | 35 | 47 | 59 |
| | | 12 | 24 | 36 | 48 | 60 |



# Timetabling

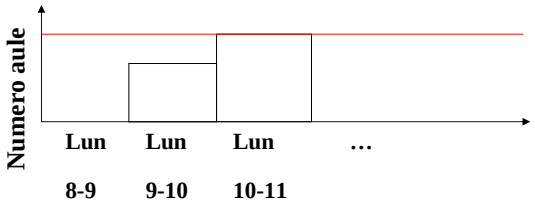
```
solve_ttturni(LSlot,N,LCosts):-
   crea_slot_turni(LSlot),
   imponi_vincoli(LSlot),
   break_symmetries(LSlot),
   obj_function(LSlot,N,LCosts),    N #< 10000,
   min_max(
    (search(LSlot,start of
slot,most_constrained,indomain_random,complete,[]),
   aula_assignment(LSlot),
    save_solution(LSlot,N), print_solution(LSlot)
     )
   ,N).
```



# Imposizione vincoli

```
imponi_vincoli(LSlot):-
   no_overlap_docente(LSlot),
   no_overlap_studente(LSlot),
   impose_cumulative(LSlot),
   other_constraints_courses(LSlot).
```

- *impose_cumulative impone il vincolo cumulativo in cui le aule sono risorse*



# Vincoli sulle aule

- *Per tutti i possibili sottoinsiemi di aule*
  - *Seleziona i corsi che possono stare **solo** in quel sottoinsieme di aule*
  - *Ciascun corso consuma 1 aula (1 risorsa)*
  - *Imponi il vincolo cumulativo su quelle risorse*
- *Es*
  - *Info1 → 130 stud* — *Aula 1 → 250 posti*
  - *Analisi1 → 160 stud* — *Aula 5 → 157 posti*
  - *Fisica 2 → 100 stud* — *Aula 7 → 120 posti*
  - *Reti → 100 stud*
- *Imponiamo:*
  - *Aula 1 →* `cumulative([Analisi1],…,1).`
  - *Aula 5 →* `cumulative([],…,1).`
  - *Aula 7 →* `cumulative([],…,1).`
  - *Aule 1,5 →* `cumulative([Info1,Analisi1],…,2).`
  - *Aule 1,5,7 →* `cumulative([Info1,Analisi1,Fisica2,Reti],…,3).`
  - *Aule 1,7 →* `[] …`



# Euristiche

- *Ho vincolo di non sovrapposizione sulle aule, sugli studenti, sui docenti. Se decido che un gruppo di studenti stanno tutti nella stessa aula, soddisfacendo il no_overlap sull'aula, soddisfo automaticamente il no_overlap sugli studenti (può essere visto come least constraining principle)*
- *Quindi seleziono il gruppo di studenti che hanno più corsi ed assegno loro un'aula. Tolgo dal dominio degli altri corsi l'aula in questione.*
- *Alcune aule sono uguali dal punto di vista del modello. Questo introduce simmetrie: il labeling evita di assegnare l'aula simmetrica.*
- *Problema: io voglio imporre il vincolo cumulativo sui corsi assegnati ad una stessa aula. Però posso imporlo solo dopo che ho assegnato i corsi alle aule!*
- *Esistono dimostrazioni formali che ripartire daccapo è utile se si usa una selezione casuale (min_max va d'accordo con indomain_random)*



# Funzione obiettivo

- *Minimizzare il numero di "buchi" di orario per gli studenti*
- *Allo stesso tempo, bisogna evitare che gli studenti abbiano giornate troppo "piene"*
- *Un giorno libero deve essere considerato positivo!*
- *Contano di più i gruppi di studenti che sono più numerosi*
- *Gli studenti che hanno molte scelte non vengono considerati*

*Differenza fra la fine dell'ultima ora e l'inizio della prima*

*•Aggiungo un valore negativo per ogni giorno libero*

*•Minimizzo la somma pesata (considerando il numero di studenti)*



# Risultati

- *Minimizzazione delle sovrapposizioni:*
  - *AA 2003/04: 9 sovrapposizioni fra corsi obbligatori e facoltativi (dato calcolato su 2 trimestri)*
  - *AA 2004/05: 0 sovrapposizioni (su 3 trim)*
  - *AA 2003/04: 20 sovrapposizioni per recupero (su 2 trim)*
  - *AA 2004/05: 0 sovrapposizioni per recupero (su 3 trim)*

# Creazione di nuovi vincoli

- In certi casi può essere utile creare nuovi vincoli
  - se il linguaggio non è abbastanza espressivo
  - se ho trovato un algoritmo efficiente per implementare un vincolo globale
- Metodi per implementare nuovi vincoli
  - implementazione sospensioni
  - vincolo element/3
  - libreria propia
  - Constraint Handling Rules



# Sospensioni

- Vincoli gestiti internamente tramite il concetto di Sospensione. Una sospensione è un goal che può essere addormentato in attesa che avvenga un certo evento.
- Ricordate l'algoritmo AC3?
  - Due liste di vincoli: Lista dei vincoli attivi e lista di vincoli addormentati
  - Operazioni di: inserimento in lista, estrazione dalla lista, spostare vincoli da una lista all'altra (addormentare vincoli e riscegliare vincoli)
- ECLiPSe gestisce automaticamente le liste tramite uno scheduler.
- Noi dobbiamo solo addormentare un vincolo, in attesa di un evento. Quando l'evento si verifica, ECLiPSe mette il vincolo (goal) nel risolvente.
- Gli eventi nella libreria FD sono
  - min cancellazione del minimo nel dominio
  - max cancellazione del massimo
  - any cancellazione di un elemento qualsiasi del dominio
  - Ci sono altri eventi in altre librerie. Ad esempio, inst nella libreria suspend corrisponde all'istanziazione della variabile



# 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



# Propagazione e sospensioni

- per implementare un nuovo vincolo (a basso livello) si
  - definisce un predicato
  - il predicato elabora i domini delle variabili coinvolte
  - se il vincolo non è completamente risolto, si sospende
- In pratica,
  - ECLiPSe implementa l'algoritmo AC3, con la lista dei vincoli sospesi (addormentati).
  - Noi possiamo implementare nuovi vincoli implementando la parte di propagazione (eliminazione di valori inconsistenti).

# Primitive per gestire i domini

- *dvar_domain(X,D) fornisce il dominio (dato astratto) della variabile X*

  *Es X::1..10, dvar_domain(X,D).*

  *yes, D=1..10*

- *dom_to_list(D,L) trasforma il dominio in lista*

  *Es X::[1..3, 100..102], dvar_domain(X, D),*

  *dom_to_list(D,L).*

  *yes, D = [1..3, 100..102]*

  *List = [1, 2, 3, 100, 101, 102]*

  *Passa da una rappresentazione compatta ad una estesa: spesso sconsigliato*



# Primitive per gestire i domini

- *dom_check_in(+Element, +Dom)*
  - *Verifica che l'intero Element sia nel dominio Dom.*
- *dom_compare(?Res, +Dom1, +Dom2)*
  - *Confronta due domini. Res viene unificato con*
    - *= sse Dom1 = Dom2,*
    - *< sse Dom1 ⊂ Dom2,*
    - *> sse Dom2 ⊂ Dom1.*
  - *Fallisce se nessuno è sottoinsieme dell'altro*
- *dom_member(?Element, +Dom)*
  - *istanzia nondeterministicamente Element ad uno dei valori in Dom*
- *dom_range(+Dom, ?Min, ?Max)*
  - *Fornisce il minimo ed il massimo del dominio*
- *dom_size(+Dom, ?Size)*
  - *Fornisce il numero di elementi nel dominio*



# Primitive per gestire i domini

- *dvar_remove_element(+DVar, +El)*
  - *Elimina El dal dominio della variabile DVar*
- *dvar_remove_smaller(+DVar, +El)*
  - *Elimina dal dominio di DVar gli elementi < El*
- *dvar_remove_greater(+DVar, +El)*
  - *Elimina dal dominio di DVar gli elementi > El*



# (meta)predicato Suspend

*suspend(+Goal, +Prio, +CondList)*

- *Sospende il Goal in attesa che si verifichi una delle condizioni nella CondList*
- *CondList è una lista che contiene elementi del tipo*

  *Variabili -> libreria:evento*
- *Prio è una priorità da 1 a 12: vengono risvegliati prima i vincoli con priorità bassa*
- *Es:* suspend(c(A,B),3,[A->fd:min,B->suspend:inst])

  *sospende il goal c(A,B) e lo risveglia quando o viene eliminato il minimo nel dominio di A o viene istanziato B.*

# Esempio: implementazione di un vincolo

```
minimo(A,B,C):-
 dvar_domain(A,DomA),
 dom_range(DomA,MinA,MaxA),
 dvar_domain(B,DomB),
 dom_range(DomB,MinB,MaxB),
 min_int(MinA,MinB,MinMin),
 min_int(MaxA,MaxB,MaxMin),
 dvar_remove_smaller(C,MinMin),
 dvar_remove_greater(C,MaxMin),
 ( nonvar(A), nonvar(B), nonvar(C)
   ->true
   ; suspend(minimo(A,B,C),3,[A->fd:min,
        A->fd:max,B->fd:min,B->fd:max])
 ), wake.

min_int(A,B,B):- A>=B,!.
min_int(A,B,A):- A<B.
```

Estraggo i domini

Calcolo i nuovi domini

Elimino i val inconsistenti

Se vincolo risolto -> fine

Altrimenti sospendo

Risveglio altri vincoli



# Esempio

- A::3..5,B::2..6, C::0..9, minimo(A,B,C).

  A = A{[3..5]}

  B = B{[2..6]}

  C = C{[2..5]}

  Delayed goals:

  minimo(A{[3..5]}, B{[2..6]}, C{[2..5]})



# Miglioramenti?

- ✗ *Non faccio propagazione da C verso A e B*

  A :: 3..5, B :: 2..6, C :: 1..2, minimo(A, B, C).
  A = A{[3..5]},B = B{[2..6]},C = 2
  Delayed goals:
  minimo(A{[3..5]}, B{[2..6]}, 2)

- ✗ *Risveglio il vincolo troppo spesso*
  - *Se MinA<MinB non c'è bisogno di svegliarsi su B->fd:min*
- *Se MaxA<MinB, so già che A=C:*
  - *potrei evitare di sospendere il vincolo minore(A,B,C) ed imporre il vincolo A#=C.*

*Domanda: che tipo di consistenza ottengo? GAC? GBC?*



# Vincolo element/3

- *Un vincolo è una relazione, quindi può essere scritto come tabella in cui elenco le coppie consistenti e quelle inconsistenti*

| X | Y | c(X,Y) |
|---|---|---|
| 0 | 0 | ✓ |
| 0 | 1 | ✗ |
| 0 | 2 | ✓ |
| 1 | 0 | ✓ |
| 1 | 1 | ✗ |
| 1 | 2 | ✓ |

# Vincolo element/3

- Oppure con una tabella in cui elenco solo le consistenti
- In questo caso, il vincolo è soddisfatto solo se viene selezionata una riga

| X | Y |
|---|---|
| 0 | 0 |
| 0 | 2 |
| 1 | 0 |
| 1 | 2 |

```
c(X,Y):-
  element(I,[0,0,1,1],X),
  element(I,[0,2,0,2],Y).
```



# PROPIA

- Propia è una libreria per definire vincoli a partire da predicati (o trasformare predicati in vincoli)
- Considera le soluzioni possibili ed effettua la minima generalizzazione dei domini

```
c(0,0).
c(0,2).
c(1,0).
c(1,2).
C(2,4).
```

```
?- c(X,Y) infers most.
X = X{[0..2]}
Y = Y{[0, 2, 4]}
Delayed goals:
     c(X{[0..2]}, Y{[0, 2, 4]}) infers most
```



# PROPIA

- Propia fornisce il metapredicato infers, che trasforma un predicato in vincolo. Dopo infers si possono usare varie parole chiave, che indicano il tipo di propagazione richiesto (AC è infers most o infers fd)
- Per effettuare la generalizzazione, utilizza il dominio delle variabili di un certo solver.
  - Si possono usare diversi solver (FD, Herbrand, …)
  - Bisogna aver caricato il solver corrispondente prima

```
lib(fd).
lib(propia).
```



# PROPIA

- Si può usare anche con predicati non ground o ricorsivi

```
no_overlap(Start1,Dur1,Start2,Dur2):-
  Start1 #>= Start2+Dur2.
no_overlap(Start1,Dur1,Start2,Dur2):-
  Start2 #>= Start1+Dur1.

?- S::1..10, no_overlap(S,2,5,3) infers fd.
   S = S{[1..3, 8..10]}
   yes
```

*Simile a disgiunzione costruttiva.*

## Compito 13 set 2007

- *Un commesso viaggiatore deve passare per un insieme di città e poi tornare alla città iniziale, percorrendo meno chilometri possibile.*
- *Per ogni coppia di città collegate da una strada, è riportata la distanza in un insieme di fatti dista/3*

```
dista(bologna,ferrara,50).
dista(ferrara,bologna,50).
dista(bologna,ravenna,84).
```

…



## Modello semplice

- *Lista di variabili:*
  - *La prima var è la prima città visitata*
  - *La seconda var è la seconda città visitata*
  - *…*
- *Domini: le varie città:*
  - `[A,B,C,D…] :: [ferrara,ravenna,bologna…]`
- *Vincoli: ci deve essere un arco fra una città e la successiva. Il vincolo è proprio il predicato dista: devo trasformare il predicato in vincolo*
- *Funzione obiettivo: min somma delle distanze*



## Consideriamo questo problema:

```
?- X :: 1..10000000,
   Y :: 1..10000000, X#>Y, Y#>X.
```



## Soluzione?

- *Noi ci accorgiamo immediatamente del fallimento, perché 'vediamo' i vincoli dall'esterno, non dall'interno*
- *Sappiamo che quando ci sono alcune combinazioni di vincoli, non ci possono essere soluzioni*

# Ordinamenti

- *In matematica, i vincoli <, ≤, >, … sono associati agli ordinamenti*
- *Un ordinamento (parziale) è una relazione binaria che gode delle proprietà:*
  - *Riflessività: a ≤ a.*
  - *Antisimmetria: a ≤ b, b ≤ a ➔ a=b*
  - *Transitività: a ≤ b, b ≤ c ➔ a ≤ c.*
- *Quindi il vincolo in matematica è definito in base alle sue proprietà. Possiamo definire anche noi un vincolo basandoci sulle proprietà?*



# Constraint Handling Rules (CHR)

- *Un modo per definire la propagazione di nuovi vincoli.*
- *I vincoli sono definiti tramite delle regole, che possono essere di tre tipi: simplification, propagation e simpagation (caso particolare di simplification)*



# Propagation rules

- *Una Propagation rule ha la sintassi:*

  **`c1, c2, … ==> guardia | body.`**

  *e significa che se i vincoli* **`c1, c2, …`** *sono nel constraint store e la* **`guardia`** *è vera, allora anche il* **`body`** *deve essere vero.*
- *La* **`guardia`** *è opzionale.*
- *Nella testa (antecedente) possono comparire solo vincoli. Questi sono i nuovi vincoli che vogliamo definire (non possono comparire vincoli predefiniti, come #<, alldifferent, …)*
- *Operazionalmente, se i vincoli* **`c1, c2, …`** *sono nello store, verifica se la* **`guardia`** *è vera, poi esegue il* **`body`**.



# Simplification rules

- *Una Simplification rule ha la sintassi:*

  **`c1, c2, … <=> guardia | body.`**

  *e significa che se la* **`guardia`** *è vera, allora i vincoli* **`c1, c2, …`** *sono equivalenti al* **`body`**.
- *Nella testa (antecedente) possono comparire solo vincoli. Questi sono i nuovi vincoli che vogliamo definire (non possono comparire vincoli predefiniti, come #<, alldifferent, …)*
- *Operazionalmente, se i vincoli* **`c1, c2, …`** *sono nello store, verifica se la* **`guardia`** *è vera, poi toglie dal constraint store* **`c1, c2, …`** *ed esegue il* **`body`**.

# Esempio: vincolo leq (less or equal)

```
reflexivity@  leq(X,X) <=> true.
antisymmetry@ leq(X,Y), leq(Y,X) <=> X=Y.
transitivity@ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

```
leq(A,B), leq(B,C), leq(C,A)

leq(A,B), leq(B,C), leq(C,A), leq(A,C)

leq(A,B), leq(B,A), A=C

A=B, A=C
```



# Note

- *CHR non utilizza l'unificazione, ma il pattern-matching (fa l'unificazione solo in una direzione). Ad es,* **`leq(A,B)`** *non attiva la regola*

  ```
  leq(X,X) <=> true.
  ```

  *(la testa della regola deve essere più specifica)*
- *La guardia deve essere un semplice test (ad es, var, ground, …) non deve unificare variabili che compaiono nella testa. Se si effettuano unificazioni, la guardia fallisce. Ad es:*

  ```
  leq(X,Y) <=> X=Y|true.
  ```

  *è equivalente a* **`leq(X,X) <=> true.`**
- *Non posso usare nella testa delle regole dei vincoli predefiniti, ma solo vincoli definiti con CHR. Se voglio, posso scrivere delle regole che "trasformano" un vincolo CHR in un altro vincolo, tipo:*

  ```
  leq(X,Y) ==> X#=<Y
  ```

  *oppure*

  ```
  leq(X,Y) <=> nonvar(X), nonvar(Y) | X =< Y.
  ```



# CHR in ECLiPSe

- *Per usare CHR, si deve caricare la libreria chr, oppure la nuova implementazione ech (V. sul manuale le differenze).*



# CLP(R) in ECLiPSe

- *ECLiPSe ha una libreria per l'uso di CLP(R), chiamata* **`eplex`**
- *Utilizzo di un risolutore esterno (Cplex, Xpress-MP oppure solver free), basato sull'algoritmo del simplesso*
- *Creazione di un'istanza del solver, in cui si stabilisce una funzione obiettivo*

  ```
  eplex:eplex_solver_setup(min(X))
  eplex:eplex_solver_setup(max(X))
  ```
- *Definizione di domini:*

  **`$::`** *oppure* **`eplex:(L :: Dom)`**
- *Vincoli lineari:*

  **`$>=, $=<, $=`**
- *risoluzione*

  **`eplex_solve(Cost)`**

# Esempio

```
:- lib(eplex).
lp_example(X,Y,Cost) :-
     eplex_solver_setup(min(X)),
     [X,Y] $:: 0..10,
     X+Y $>= 3,
     X-Y $= 0,
     eplex_solve(Cost).
```

- *Risultato:*

```
:- lp_example(X,Y,C).
X = X{0 .. 10 @ 1.5}
Y = Y{0 .. 10 @ 1.5}
C = 1.5
```

Soluzione ottima

Valore della soluzione ottima



# Integrazione dei due solver

- *In molte applicazioni, fare cooperare i due solver porta a soluzioni più velocemente di ciascuno dei due*
- *Nell'esempio precedente, la propagazione Arc-Consistency non restringe i domini:*

```
[X,Y] #:: 0..10,X+Y #>= 3,X-Y #= 0.
X = X{[0..10]}
Y = X{[0..10]}
Delayed goals:  X{[0..10]} + X#>=3
```

- *eppure il valore ottimo del rilassamento lineare è X=1.5 (quindi non ci sono soluzioni con X=0, X=1)*
- *Posso calcolare il valore ottimo del rilassamento lineare e imporre X #>= C.*
- *E` anche possibile creare un vincolo che esegue il simplesso*



# OVERVIEW

- Constraint Satisfaction (Optimization) Problems
- Constraint (Logic) Programming
  - *language and tools*
- AI Applications: modelling and solving
  - *Scheduling - Timetabling - Resource Allocation*
  - *Routing*
  - *Packing - Cutting*
  - *Graphics - Vision*
  - *Planning*
- Advantages and Limitations of CP: extensions



# ADVANTAGES OF CP

- Easy problem modelling
- Constraints provide a natural way of implementing propagation rules
- Flexible treatment of variants of original problems:
  - easy introduction of new constraints
  - transparent interaction with other constraints
- Easy control of the search

## LIMITATIONS of PURE CP

- Optimization side not very effective
- Over-Constrained problems:
  - no effective way of relaxing constraints
  - hard/soft constraints
- Dynamic Changes:
  - addition/deletion of variables
  - addition/deletion of domain values
  - addition/deletion of constraints



## CP EXTENSION FOR OPTIMIZATION

- Integration of OR techniques in CP tools:
  - MP based solvers: 2LP ***[McAloon, Tretkoof PPCP94],*** OPL ***[Van Hentenryck, 99],*** Planner ***[ILOG Planner Manual]***
    - Integration of CPLEX and XPRESS in FD solvers
  - Integration of specialized algorithms for:
    - computing bounds
    - using reduced costs

    } ***[Caseau, Laburthe ICLP97 and CP97], [Focacci, Lodi, Milano ICLP99 and CP99],***
  - Improvement of CP branch and bound
    - ***[Rodosek, Wallace, Hajian Annals OR 97], [Caseau, Laburthe ICLP94 and JICSLP96], [Beringer, DeBacker, LP Formal Method and Pract. Appl. 95]***
  - Integration of local search techniques
  - ***[DeBacker, Furnon,Shaw CPAIOR99], [Caseau, Laburthe CPAIOR99], [Gendreau, Pesant, Rousseau, Transp. Sci. 98]***
  - Integration of branch and cut in a logical setting
    - ***[Bockmayr ICLP95], [Kasper PhD, 99]***



## CP EXTENSION FOR OVER-CONSTRAINED PROBLEMS

- HCSP:
  - Implementation of CP solvers exploiting Hierarchical CSP framework
  - Meta programming

***[A. Borning OOPSLA87], [A. Borning et al. ICLP89], [M.Jamper PhD, 96]***

## CP EXTENSION FOR DYNAMIC CHANGES

- DCSP
- ATMS-based solvers
- Interactive Constraint Satisfaction

} Complex data-structures

***[R.Dechter, A.Dechter, AAAI88], [Verfaillie, Schiex AAAI94], [Bessiere, AAAI91], [Lamma et al. IJCAI99]***



## TO KNOW MORE…..

- *Conferences:*
  - *International Conference on Principles and Practice of Constraint Programming (CP)*
  - *Logic programming conferences (ICLP - ILPS - JICSLP)*
  - *AI Conferences (ECAI - AAAI - IJCAI)*
  - *Operations research conferences (INFORMS - IFORS)*
  - *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*
- *Books:*
  - *Krzysztof R. Apt and Mark Wallace,* Constraint Logic Programming using ECL$^i$PS$^e$, *Cambridge*
  - *K. Marriott and P. Stuckey,* Programming with constraints: An Introduction, *MIT Press*
  - *Rina Dechter,* Constraint Processing, *Morgan Kaufmann*

# TO KNOW MORE…..

- *Journals:*
  - *Constraint - An International Journal*
  - *AI - LP - OR Journals*
- *Industrial Applications:*
  - *COSYTEC, ILOG, CrosscoreOptimization, SIEMENS, BULL*
- *News group:* **`comp.constraints`**
- *Mailing lists:* **`CPWORLD@gmu.edu`**
- *Constraint Archive:* **`http://4c.ucc.ie/web/archive/`**