Les contraintes sur domaines finis en Eclipse CLP

- On utilise la librairie `ic` avec `lib(ic).` (ou `:- lib(ic).` dans le programme).
- Attention: cette librairie fait plus que les constraintes à domaines finis
- Nous utiliserons uniquement des variables entières.
- Leurs domaines doivent être définis.
  par exemple `L = [X,Y], L #:: [-3..10].` ou `X #:: [1,2,5,6]`
- `alldifferent(L)` impose que toutes les variables de `L` ont des valeurs différentes.
- Les contraintes arithmétiques s'écrivent en utilisant les fonctions habituelles et les prédicats : `#=`, `#\=`, `#<`, `#>`, `#<`, `#=<`, `#>=`.
  Règles de propagation : bornes-consistance sauf pour `#\=` où l'arc-consistance est utilisée.
- `ac_eq(X,Y,C)` est l'implémentation avec arc-consistance de `X #= Y+C`.

Exemples

```
[eclipse]: X #:: [1..8], Y #::[2..7], X #= 2*Y.
X = X{4 .. 8}
Y = Y{2 .. 4} ...
[eclipse]: [X,Y] #:: [0..9], X #\= 5, X #= Y+2.
X = X{[2 .. 4, 6 .. 9]}
Y = Y{0 .. 7} ...
[eclipse]: [X,Y] #:: [0..9], X #\= 5, ac_eq(X,Y,2).
X = X{[2 .. 4, 6 .. 9]}
Y = Y{[0 .. 2, 4 .. 7]} ...
```

Si on veux construire une expression et l'utiliser dans une contrainte à l'éxécution, il faut l'encapsuler avec `eval/1`

```
[eclipse]: [X, Y] #:: 0..10, Expr = X + Y, Sum #= Expr.
number expected in set_up_ic_con(7, 1, [0 * 1, 1 *
Sum{-1.0Inf .. 1.0Inf}, -1 * (X{0 .. 10} + Y{0 .. 10})])
[eclipse]: [X, Y] #:: 0..10, Expr = X + Y, Sum #= eval(Expr).
X = X{0 .. 10}  Y = Y{0 .. 10}
Expr = X{0 .. 10} + Y{0 .. 10}
Sum = Sum{0 .. 20} ...
```

Chercher une solution

- `indomain(+Dvar)`
  instantie la variable `Dvar` avec une valeur de son domaine, en revenant par retour en arrière une nouvelle valeur est prise.
- `labeling(?Vars)`
  Ce prédicat est utilisé pour rechercher des solutions de contraintes sur les variables `Vars`. Il utilise `indomain`.
- `search(+L, ++Arg, ++Select, +Choice, ++Method, +Option)`
  permet de chercher une solution pour les variables `L` avec plein d'options (voir `http://eclipseclp.org/doc/bips/lib/ic/search-6.html`).
  Typiquement `Arg` est 0, `Select` est `first_fail`, `Choice` est `indomain_min`, `Method` est `complete` et `Option` est `[]`.

Structure générale d'un programme simple

- Dans le cas le plus simple, un programme pour résoudre une contrainte s'écrit en trois parties :
  - définir les domaines des variables
  - décrire la contrainte
  - labeling ou search
- Exemple:

```
probleme([X,Y,Z]) :- X #:: [0..5],
                     [Y,Z] #:: [3..7],
                     X+Y #< 2*Z,
                     labeling([X,Y,Z]).
[eclipse]: probleme(L).
L = [0,3,3] ? ;
L = [0,3,4] ? ;
L = [0,3,5] ? ;
...
```

Exemple: sac du contrebandier

- Contrebandier avec un sac de capacité 9.
- Il doit choisir des objets pour faire un profit d'au moins 30
objet	profit	poids
whisky	15	4
parfum	10	3
cigarettes	7	2

  $4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$

Le contrebandier en Eclipse CLP

```
[eclipse]: [W,P,C] #:: [0..9],
    4*W + 3*P + 2*C #=< 9,
   15*W + 10*P + 7*C #>= 30,
   labeling([W,P,C]).
W = 0
P = 1
C = 3 ? ;
W = 0
P = 3
C = 0 ? ;
W = 1
P = 1
C = 1 ? ;
W = 2
P = 0
C = 0
```

Éliminer des symétries

- Des symétries dans l'arbre de recherche peuvent faire un programme naïf très inefficace.
- On peut avoir beaucoup de façons symétriques d'écrire une partie de l'arbre de recherche.
- C'est un problème quand il s'agit d'une partie de l'arbre de recherche dont tous les nœuds sont des nœuds d'échec.
- Si on élimine des symétries : attention quand on essaye de trouver *toutes* les solutions.

Exemple : élimination de symétries

- Dans un magasin : le client veut payer quatre articles.
- Le caissier lui annonce un prix de 7.11 €.
- Quand le client lui demande les prix des articles séparés, le caissier lui répond simplement que le produit des prix est également 7.11 €.
- Quels sont les prix des quatre articles ?

Exemple 7.11 € en ECLIPSE CLP

```
prix(A,B,C,D) :-
        [A,B,C,D] #:: [1..708],
        A+B+C+D #= 711,
        A*B*C*D #= 711 * 100 * 100 * 100,
        labeling([A,B,C,D]).
[eclipse] : prix(A,B,C,D).
A = 120
B = 125
C = 150
D = 316 ?
Yes (25.70s cpu,...)
```

Est-ce qu'il y a d'autres solutions ?

Éliminer les symétries

Idée : on impose l'ordre sur les valeurs des variables.

```
prixa(A,B,C,D) :-
      [A,B,C,D]#::[1..708],
      A+B+C+D #= 711,
      A*B*C*D #= 711 * 100 * 100 * 100,
      A #=< B, B #=< C, C #=< D,
      labeling([A,B,C,D]).
```

Trouve **la** solution en 3.47 secondes.

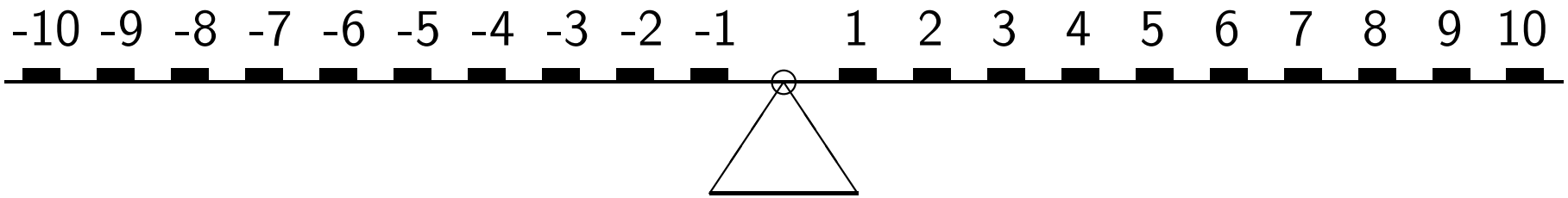
Encore mieux

Idée : 79 est un facteur premier de 711. Choisir la variable qui contient ce facteur.

```
prixb(A,B,C,D) :-
    [A,B,C,D]#::[1..708],
    A+B+C+D #= 711,
    A*B*C*D #= 711 * 100 * 100 * 100,
    A #= 79*X,
    B #=< C, C #=< D,
    labeling([A,B,C,D]).
```

Trouve la solution en 0.62 secondes.

Exemple: Programmer la génération des contraintes



- Donné : liste des poids des personnes.
- Exemple : $[30, 40]$
- Placer toutes les personnes de sorte que la balançoire soit en équilibre.
- Pour l'exemple : $[-4, 3]$.

Solution balançoire

```
placement(Poids,Places) :-
        length(Poids,N),
        length(Places,N),
        Places #:: [-10..10],
        alldifferent([0|Places]),
        gauche(Poids,Places,M),
        droite(Poids,Places,M),
        labeling(Places).
```

Solution balançoire

```
gauche([],[],0).
gauche([_|Poids],[Place|Places],M) :-
     Place #> 0, gauche(Poids,Places,M).
gauche([Poid|Poids],[Place|Places],M) :-
     Place #< 0,
     M #= MM - Poid*Place,
     gauche(Poids,Places,MM).
droite([],[],0).
droite([_|Poids],[Place|Places],M) :-
     Place #< 0, droite(Poids,Places,M).
droite([Poid|Poids],[Place|Places],M) :-
     Place #> 0,
     M #= MM + Poid*Place,
     droite(Poids,Places,MM).
```

Exemple: Pavage

- Donnée: une liste de pavés (exemple: $[(2,2),(3,4)]$) et les dimensions du plan.
- Déterminer si on peut placer les pavés sur le plan sans recouvrement, et si oui donner le plan (les coordonnées des coins des pavés placés).
- Ici : utiliser une contrainte définie par l'utilisateur qui exprime que deux pavés ne se recouvrent pas.
- Ici pour simplifier : on n'autorise pas de tourner les pavés.
- La position du pavé est donnée par quatres valeurs (les coordonnées du point le plus à gauche et le plus bas et les coordonnées du coin opposé).

Exemple : pavage (1)

```
disjoint((_,_,X2,_),(X1,_,_,_)) :- X2 #=< X1.
disjoint((X1,_,_,_),(_,_,X2,_)) :- X1 #>= X2.
disjoint((_,_,_,Y2),(_,Y1,_,_)) :- Y2 #=< Y1.
disjoint((_,Y1,_,_),(_,_,_,Y2)) :- Y1 #>= Y2.

alldisjoint(_,[]).
alldisjoint(Q,[H|T]) :- disjoint(Q,H), alldisjoint(Q,T).
```

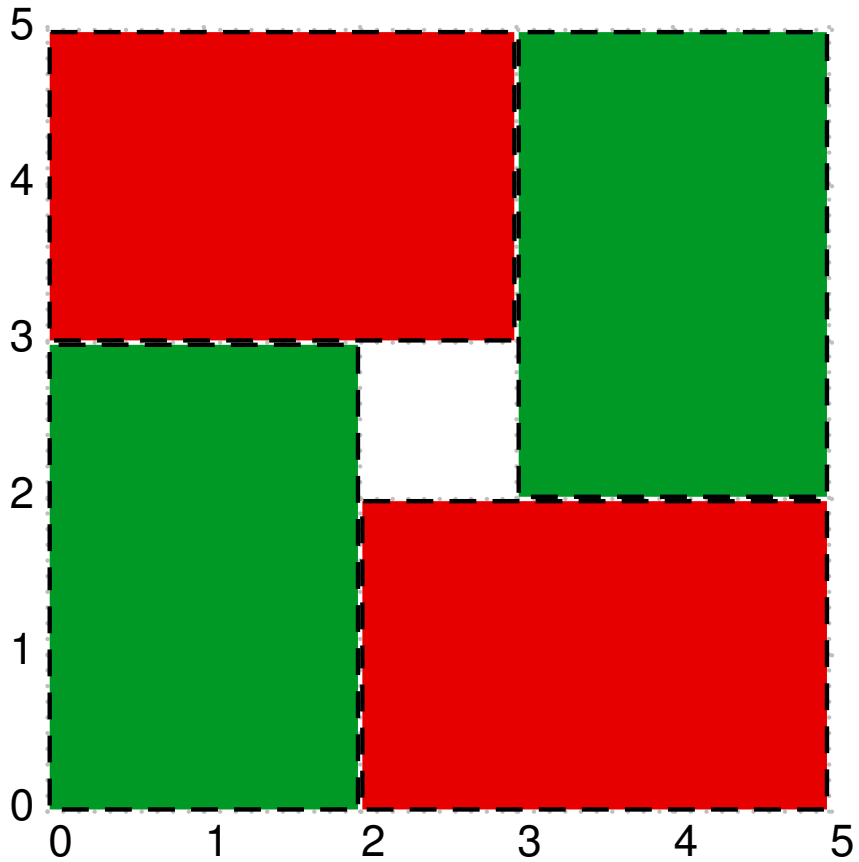
Exemple : pavage (2)

```
p([],_,_,[],[]).
p([(X,Y)|Rest], Width, Height, [(X1,Y1,X2,Y2)|Plan],
  [X1,Y1,X2,Y2|Vars]) :-
    [X1,X2]#::[0..Width],
    [Y1,Y2]#::[0..Height],
    X2 #= X1 + X,
    Y2 #= Y1 + Y,
    p(Rest,Width, Height, Plan, Vars),
    alldisjoint((X1,Y1,X2,Y2), Plan).

paving(In,Width,Height,Plan) :-
    p(In,Width,Height,Plan,Vars),
    labeling(Vars).
```

Exemple : le problème de la guillotine

- On veut couper une feuille de papier en plusieurs morceaux.
- On a seulement une guillotine pour découper la feuille.
- Étant donné les dimensions de la feuille et des morceaux, est-ce possible ?

Différence avec le problème de pavage

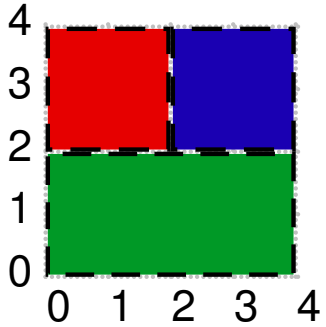


Spécification du problème

Écrire un prédicat `coupe(?Pieces,?Hauteur,?Largeur,+Plan)` où

- `Pieces` est la liste des dimensions des morceaux souhaités,
- `Largeur` et `Hauteur` sont les dimensions du plan à découper
- `Plan` est le plan de découpage.

```
| : coupe([(2,2),(2,2),(2,4)],4,4,P).
P = horizontal(2,vertical(2,piece(2,2),piece(2,2)),
               piece(4,2))
```



L'idée de l'algorithme

- Construction d'un arbre.
- Si on veut un seul morceaux, et s'il tient dans les dimensions : créer une feuille de l'arbre.
- Si on veut plusieurs morceaux :
  - choisir la ligne de découpage.
  - choisir une partition des morceaux telle que les sommes de surfaces tiennent dans les deux moitiés du plan.
  - appeler coupe récursivement sur les deux sous-problèmes, combiner les plans obtenus pour construire le plan entier.

Première solution naïve

```
coupe([(X,Y)],L,H,piece(X,Y)) :- X #=< L, Y #=< H.
coupe([(X,Y)],L,H,piece(Y,X)) :- Y #=< L, X #=< H.

coupe(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-
 Coupe#::[1..Largeur],
 RestLargeur #= Largeur-Coupe,
 SurfaceA #= Coupe*Hauteur,
 SurfaceB #= RestLargeur*Hauteur,
 partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
 labeling([Coupe]),
 coupe(ResultatA,Coupe,Hauteur,PlanA),
 coupe(ResultatB,RestLargeur,Hauteur,PlanB).

coupe(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-
    ...
```

Solution naïve suite

```
partition([],[],[],_,_).

partition([(X,Y)|T],[(X,Y)|R],ResultatB,SurfaceA,SurfaceB) :-
      X*Y #=< SurfaceA,
      SurfaceA #= X*Y + NewSurfaceA,
      partition(T,R,ResultatB,NewSurfaceA,SurfaceB).

partition([(X,Y)|T],ResultatA,[(X,Y)|R],SurfaceA,SurfaceB) :-
      X*Y #=< SurfaceB,
      SurfaceB #= X*Y + NewSurfaceB,
      partition(T,ResultatA,R,SurfaceA,NewSurfaceB).
```

Testons la solution naïve

```
[eclipse] : coupe([(2,2),(2,2),(2,4)],4,3,P).
No

[eclipse] : coupe([(2,2),(2,2),(2,4)],4,4,P).
Stack Overflow
```

L'erreur de la solution naïve

- La récurrence ne s'arrête pas :
  - On permet de couper tel qu'un plan a la surface 0.
  - On permet de partitionner la liste vide en deux listes vides.
- Deux solutions possibles :
  - Renforcer la contrainte sur la variable `Coupe`
  - Ne pas permettre des partitions triviales (une liste vide).

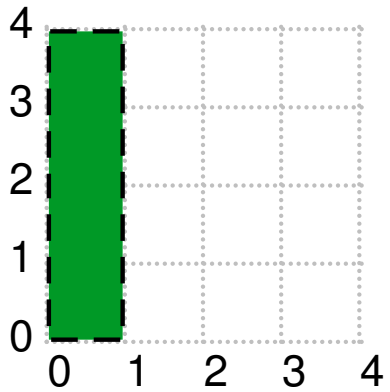
Le programme corrigé

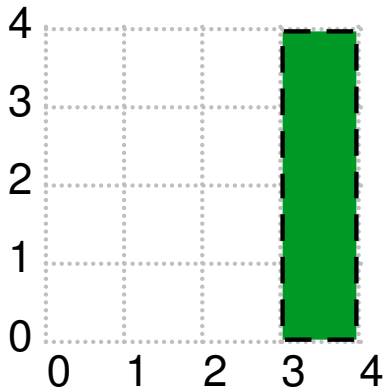
```
coupe([(X,Y)],L,H,piece(X,Y)) :- X #=< L, Y #=< H.
coupe([(X,Y)],L,H,piece(Y,X)) :- Y #=< L, X #=< H.

coupe(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-
 Coupe #:: [1..Largeur],
 Coupe#<Largeur,
 RestLargeur #= Largeur-Coupe,
 SurfaceA #= Coupe*Hauteur,
 SurfaceB #= RestLargeur*Hauteur,
 partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
 labeling([Coupe]),
 coupe(ResultatA,Coupe,Hauteur,PlanA),
 coupe(ResultatB,RestLargeur,Hauteur,PlanB).

coupe(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-
    ...
```

Éviter des symétries

- On peut éviter la symétrie qui consiste en inverser la gauche et la droite, resp. le haut et le bas:
- Il suffit de renforcer la contrainte sur `Coupe`:

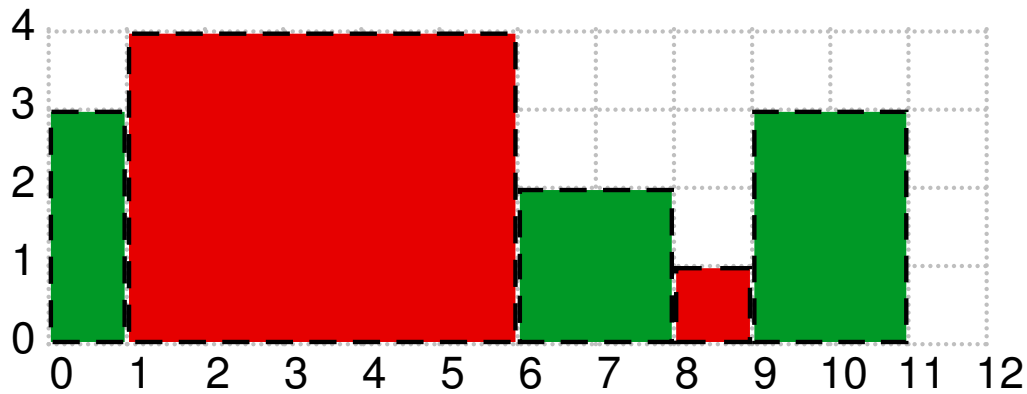
```
2*Coupe #=< Largeur,
```





Une optimisation

- Idée : on peut avoir un cas de base plus général
- Nouveau cas de base : on essaye de placer tous les morceaux côte à côte (ou empilés).



Le programme complet (1)

```
% calcule la surface totale d'une liste de morceaux
sommesurfaces([],0).
sommesurfaces([(X,Y)|R],Somme) :- sommesurfaces(R,RSomme),
                                  Somme is (X*Y) + RSomme.

% ordonne les morceaux
ordonne([],[]).
ordonne([(X,Y)|T],[(X,Y)|RT]) :- X=<Y, ordonne(T,RT).
ordonne([(X,Y)|T],[(Y,X)|RT]) :- X>Y,  ordonne(T,RT).

coupe(Pieces,Largeur,Hauteur,Plan) :-
     sommesurfaces(Pieces,S),
     S =< Hauteur*Largeur,
     ordonne(Pieces,PiecesOnEnd),
     c(PiecesOnEnd,Largeur,Hauteur,Plan).
```

Le programme complet (2)

```
c(Pieces,Largeur,Hauteur,Plan) :-
    listcoupe(Pieces,Largeur,Hauteur,Plan),!.

c(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-
    Coupe #:: [1..Largeur],
    2*Coupe #=< Largeur,
    RestLargeur #= Largeur-Coupe,
    SurfaceA #= Coupe*Hauteur,
    SurfaceB #= RestLargeur*Hauteur,
    partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
    nonempty(ResultatA), nonempty(ResultatB),
    labeling([Coupe]),
    c(ResultatA,Coupe,Hauteur,PlanA),
    c(ResultatB,RestLargeur,Hauteur,PlanB).

c(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-
    ...
```

Le programme complet (3)

```
listcoupe(Pieces, Largeur, Hauteur, list(Plan)) :-
Largeur #>= Hauteur, listcoupe_l(Pieces, Largeur, Hauteur, Plan).
listcoupe(Pieces, Largeur, Hauteur, stack(Plan)) :-
Largeur #< Hauteur, listcoupe_l(Pieces, Hauteur, Largeur, Plan1),
renversepieces(Plan1,Plan).

renversepieces([],[]).
renversepieces([(X,Y)|L],[(Y,X)|L1]) :- renversepieces(L,L1).

listcoupe_l([],_,_,[]).
listcoupe_l([(X,Y)|R], Largeur, Hauteur, [(X,Y)|PlanR]) :-
    Y #=< Hauteur,
    RestLargeur #= Largeur - X, RestLargeur #>= 0,
    listcoupe_l(R,RestLargeur,Hauteur,PlanR).
listcoupe_l([(X,Y)|R], Largeur, Hauteur, [(Y,X)|PlanR]) :-
    Y #> Hauteur,
    X #=< Hauteur,
    RestLargeur #= Largeur - Y, RestLargeur #>= 0,
    listcoupe_l(R,RestLargeur,Hauteur,PlanR).
```

Est-ce qu'on peut faire encore mieux?

- Il y a des symétries quand on a des morceaux identiques
  Exemple : `plan([(2,2),(2,2),(2,2)],4,4,Plan)`.
- Pour les éviter il faudrait changer la représentation du problème :
  Lister des morceaux différents, avec un entier qui donne le nombre de copies souhaitées.
  Exemple : `plan([(2,2):3],4,4,Plan)`.