GNU Prolog

- `http://www.gprolog.org`
- contient un solutionneur de contraintes sur un domaine fini
- Solutionneur *incomplet* basé sur arc-consistance et bornes-consistance (au choix, voir plus tard).
- Il faut écrire explicitement dans le programme le prédicat pour énumérer les éléments des domaines des variables.
- Dans les contraintes toutes les variables sont des entiers entre 0 et `fd_max_integer`.
- Les contraintes s'écrivent avec des prédicats particuliers (voir les transparents suivants).

Les domaines finis en GNU Prolog

- Attention : que de valeurs entre 0 et `fd_max_integer`.
- Deux représentations :
  - intervals
  - ensembles (Attention: marche uniquement jusqu'à `vector_max`, typiquement 127)
    Réalisé comme un vecteur de bits.
- Si pas de domaine spécifié : interval [0..fd_max_integer].
- Par défaut représentation intervals, changé dynamiquement en représentation ensemble (attention: on risque de perdre des solutions).

Prédicats de base

- `fd_domain(?Vars, +Integer1, +Integer2)`
  définit le domaine d'une variable `Vars` ou d'une liste de variables d'être entre `Integer1` et `Integer2`.
- `fd_domain(?Vars,+ListeValeurs)`
  pareil avec une liste de valeurs
- `fd_all_different(?ListeVars)`
  Ce prédicat décrit la contrainte qui impose que toutes les variables de la liste `ListeVars` prennent des valeurs différentes.

Contraintes arithmétiques

- Les contraintes arithmétiques s'écrivent en utilisant les fonctions habituelles et les prédicats : #=, #\=, #<, #>, #<, #=<, #>=.
  Règles de propagation : bornes-consistance.
- On peut aussi utiliser #=#, #\=#, #<#, #>#, #<#, #=<#, #>=#.
  Règles de propagation : (hyper)-arc-consistance

Intervals et ensembles

```
| ?- X#=<512.
X = _#2(0..512)
yes

| ?- X#=<512, X#\=10.
X = _#2(0..9:11..127@)
yes

| ?- X#=<512, X#\=10, X#=<100.
X = _#2(0..9:11..100)
yes
```

Attention au @ qui indique une perte de valeurs.

Exemple

```
| ?- fd_domain(X,1,8), fd_domain(Y,2,7), X #= 2*Y.

X = _#3(4..8)
Y = _#25(2..4)

yes
| ?- fd_domain(X,1,8), fd_domain(Y,2,7), X #=# 2*Y.

X = _#3(4:6:8)
Y = _#25(2..4)

yes
```


Labeling

- `fd_labeling(?Vars)`
  Ce prédicat est utilisé pour rechercher des solutions de contraintes sur les variables `Vars` avec des options par défaut.
- Peut être appliqué à une seule variable, ou une liste de variables.
- `fd_labeling(?Vars, ?Options)`
  Ce prédicat est utilisé pour rechercher des solutions des contraintes sur les variables `Vars` avec une liste d'options `Options`.
  - par exemple: `[variable_method(most_constrained)]` a comme effet que la variable choisie pendant la résolution est celle qui est la plus contrainte.
- plus de détails plus tard.

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

Exemple

```
probleme([X,Y,Z]) :- fd_domain(X,0,5),
                     fd_domain([Y,Z],3,7),
                     X+Y #< 2*Z,
                     fd_labeling([X,Y,Z],[]).

| ?- probleme(L).
L = [0,3,3] ? ;
L = [0,3,4] ? ;
L = [0,3,5] ? ;
etc.
```


Labeling

- Prédicat extra-logique : `fd_min(?fd_variable,+integer)` : Donne la valeur minimale du domaine de la variable.
- On peut imaginer `fd_labeling` (pour une variable) défini comme :

```
fd_labelling(X) :- fd_min(X,N), X=N.
fd_labelling(X) :- fd_min(X,N), X #\= N,
                   fd_labelling(X).
```

- Utilise la stratégie de recherche de Prolog.

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 GNU Prolog

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

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 Prolog

```
prix(A,B,C,D) :-
     fd_domain([A,B,C,D],1,708),
     A+B+C+D #= 711,
     A*B*C*D #= 711 * 100 * 100 * 100,
     fd_labeling([A,B,C,D]).

| ?- prix(A,B,C,D).
A = 120
B = 125
C = 150
D = 316 ?
(112 ms) yes
```


Éliminer les symétries

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

```
prixa(A,B,C,D) :-
     fd_domain([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,
     fd_labeling([A,B,C,D]).
```

Trouve la solution en 40 milli-secondes.

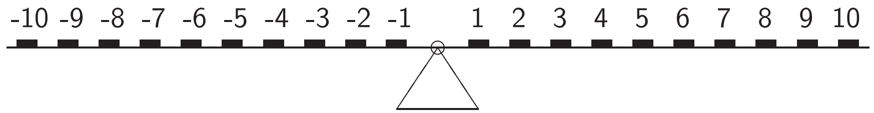
Encore mieux

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

```
prixb(A,B,C,D) :-
   fd_domain([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,
   fd_labeling([A,B,C,D]).
```

Trouve la solution en 4 milli-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 est en équilibre.
- Pour l'exemple : $[-4, 3]$.

Solution balançoire

- Attention : en GNU Prolog, les domaines finis ne contiennent que des valeurs non négatives !
- Décaler toutes les places dans le positif : $0 \dots 20$, le pivot est donc sur la position 10.

```
placement(Poids,Places) :-
     length(Poids,N),
     length(Places,N),
     fd_domain(Places,0,20),
     fd_all_different([10|Places]),
     gauche(Poids,Places,M),
     droite(Poids,Places,M),
     fd_labeling(Places,[]).
```


Solution balançoire

```
gauche([],[],0).
gauche([_|Poids],[Place|Places],M) :-
     Place #> 10, gauche(Poids,Places,M).
gauche([Poid|Poids],[Place|Places],M) :-
     Place #< 10,
     M #= MM + Poid*(10-Place),
     gauche(Poids,Places,MM).
droite([],[],0).
droite([_|Poids],[Place|Places],M) :-
     Place #< 10, droite(Poids,Places,M).
droite([Poid|Poids],[Place|Places],M) :-
     Place #> 10,
     M #= MM + Poid*(Place-10),
     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)

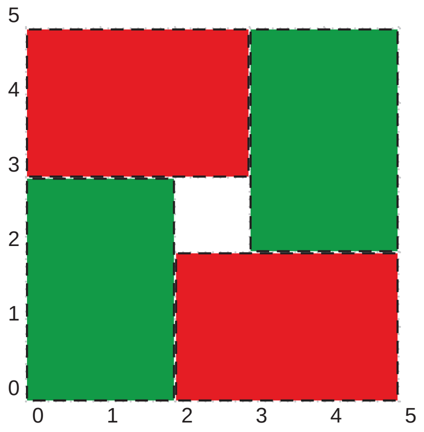
```
p([],_,_,[],[]).
p([(X,Y)|Rest], Width, Height, [(X1,Y1,X2,Y2)|Plan],
  [X1,Y1,X2,Y2|Vars]) :-
    fd_domain([X1,X2],0,Width),
    fd_domain([Y1,Y2],0,Height),
    X2 #= X1 + X,
    Y2 #= Y1 + Y,
    p(Rest,Width, Height, Plan, Vars),
    alldisjoint((X1,Y1,X2,Y2), Plan).

paving(In,Width,Height,Plan) :-
    p(In,Width,Height,Plan,Vars),
    fd_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

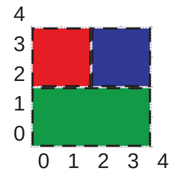


Spécification du problème

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)) :-
 fd_domain(Coupe,1,Largeur),
 RestLargeur #= Largeur-Coupe,
 SurfaceA #= Coupe*Hauteur,
 SurfaceB #= RestLargeur*Hauteur,
 partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
 fd_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

```
  | ?- coupe([(2,2),(2,2),(2,4)],4,3,P).
no

| ?- coupe([(2,2),(2,2),(2,4)],4,4,P).
Fatal Error: cstr 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).

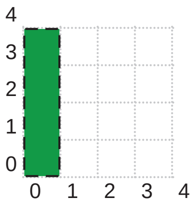
Le programme corrigé

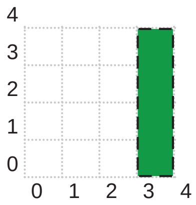
```
coupe([(X,Y)],L,H,piece(X,Y)) :- X #=< L, Y #=< H.
coupe([(X,Y)],L,H,piece(Y,X)) :- Y #=< L, X #=< H.
coupe(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-
 fd_domain(Coupe,1,Largeur),
 Coupe#<Largeur,
 RestLargeur #= Largeur-Coupe,
 SurfaceA #= Coupe*Hauteur,
 SurfaceB #= RestLargeur*Hauteur,
 partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
 fd_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:

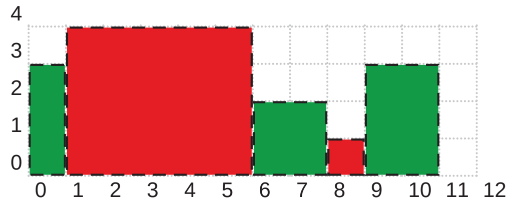
```
2*Coupe #=< Largeur,
```





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)) :-
     fd_domain(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),
     fd_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).`