Origines de la programmation sous contraintes

Prolog, un langage CLP

- Prolog est un bon candidat pour la programmation (logique) sous Contraintes :
  - Relationnel
  - SLD-Résolution et retour arrière (non déterminisme)

- En Prolog, on exprime (déjà) les contraintes par :
  - Les prédicats : relations entre leurs arguments.
  - L'unification : contraintes **d'égalité** entre les termes.

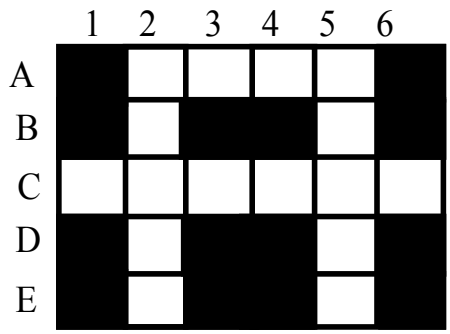
- L'unification est un solveur de contraintes d'égalité
  Unification > filtrage > test

- L'algorithme d'unification se charge de la **Propagation** des valeurs des variables vers les autres.

Exemple :
dans X=Y, Y=Z, Y=125, '**=**' est l'opérateur *d'unification* :
X est lié à Y et Y à Z. Quand Y=125,
il y a propagation vers X et Z.

Exemple-1 : génération de grille de mots croisés

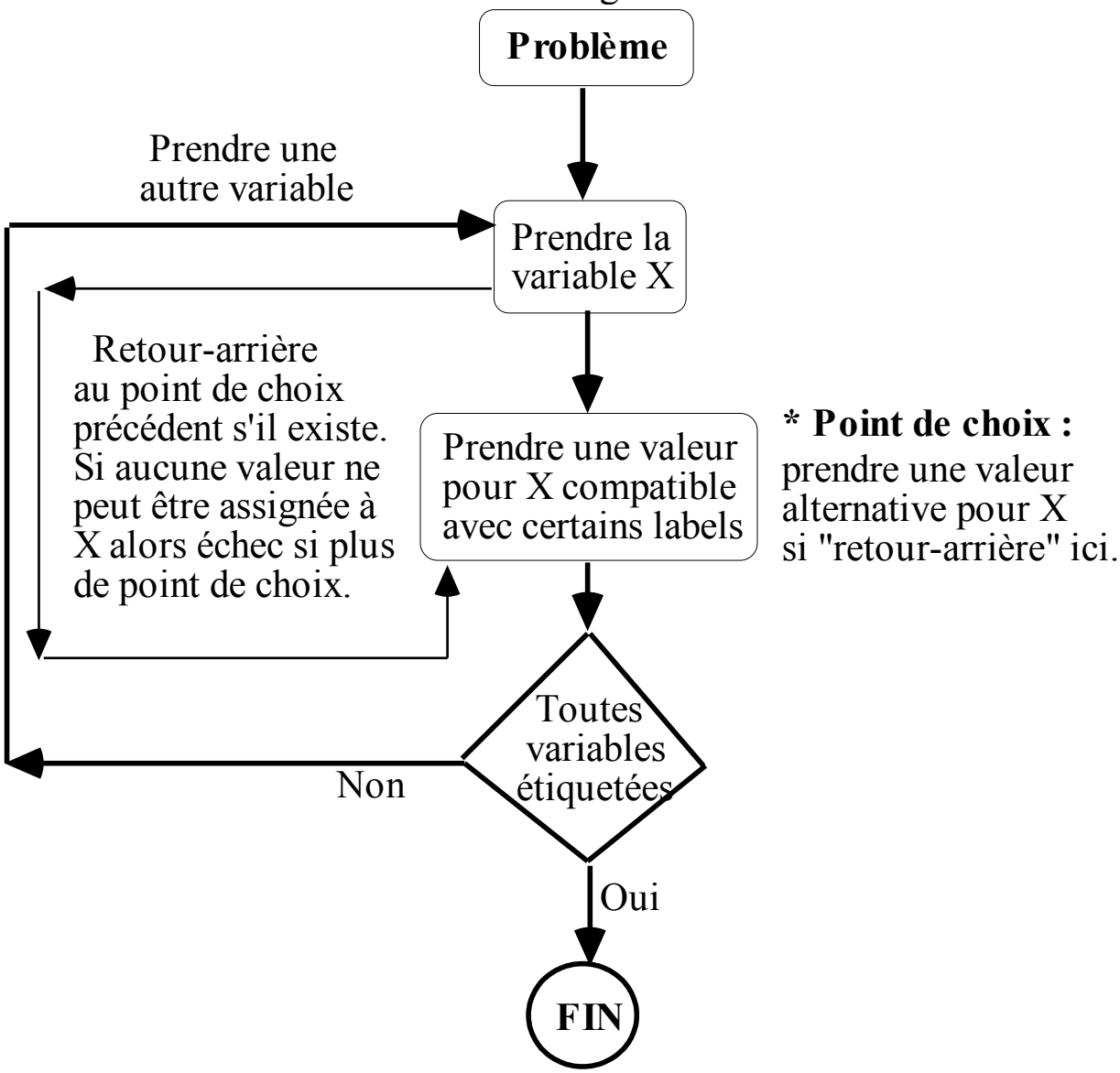
```
Solution([A2,A3,..., E5]) :-
     mot([A2,A3,A4,A5]),
          .....
     mot([A5,B5,C5,D5,E5]).

mot([b,e,t,t,e,r]) .
mot([c,a,n,n,o,n]).
   ..........
```



Algorithme BackTracking (BT)

- L'opération de base dans la résolution de CSP :
  - prendre une variable à la fois
  - lui donner une valeur à la fois : étiquetage (labeling)
  - s'assurer que le label ainsi constitué est compatible avec tous les autres labels construits jusqu'à ce stade.

- L'algorithme "retour-arrière chronologique"(chronological back-tracking **BT**) est une stratégie générale de recherche largement utilisée dans la résolution des problèmes.
- Le moteur de recherche de Prolog est à base de BT.



Contrôle de l'algorithme de retour-arrière chronologique

```
Procédure Retour_arrière_chronologique(Z, D, C)
Début
 BT_1(Z, {}, D, C);
Fin Retour_arrière_chronologique;

Procédure BT_1(Libre, Label_composé, D, C) : Labels =
% Libre : ensemble de variables à étiqueter (Unlabelled)
% Label_composé : ensemble de labels déjà réalisés
Début
 Si (Libre={}) alors retourne (Label_composé)
 Sinon
    Prendre une variable X dans Libre
    Répéter
       Prendre une valeur V dans Dx
       Si (Label_composé + {<X,V>} ne viole pas C alors
          Résultat ←
              BT_1(Libre-{X}, Label_composé+{<X,V>}, D, C);
          Si (Résultat ≠ NIL) alors retourne(Résultat); Fin si;
       Fin si;
    Jusqu'à (Dx = {});
    Retourne (NIL);
 Fin si;
Fin BT_1;
```

- Soit **n** : le nombre de variables, **e** : le nombre de contraintes et **a** la taille des domaines des variables du CSP. Il y a $\mathbf{a^n}$ combinaisons possibles de n-uplets (candidats : solutions potentielles). Pour chaque candidat, toutes les contraintes doivent être vérifiées, une fois dans le cas pire. La complexité en temps de l'algorithme BT-1 est alors $\mathbf{O(a^n e)}$.

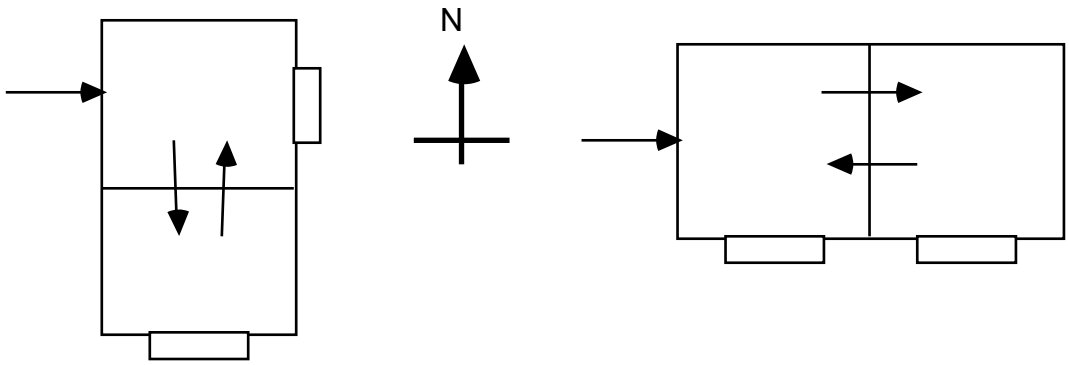
- Le stockage des domaines demande **O(na)** espaces. Ce qui donne la complexité en espace de l'algorithme sachant que le stockage des labels_composés est de l'ordre de O(n) mémoire temporaire.
- La complexité en temps montre qu'une baisse de **a** permet un gain des performances. Cette réduction est effectuée par les techniques de réduction.

Exemple-2 : Planification en Prolog

- Conception d'une unité de construction en suivant les spécifications (contraintes) suivantes :

  - L'unité comporte deux pièces rectangulaires;
  - Les pièces auront chacune une fenêtre et une porte intérieure;
  - Les pièces communiquent par les portes intérieures;
  - Une des chambres aura une porte vers l'extérieur;
  - Un mur ne doit avoir qu'une seule fenêtre ou une seule porte;
  - Aucune fenêtre ne sera face au nord;
  - Les fenêtres ne doivent pas être sur les cotés opposés de l'unité.

- Exemples de solutions (l'entrée, l'orientation de la Porte1, l'orientation de la Fenêtre1, l'orientation de la Porte2 et l'orientation de la Fenêtre2 de chaque plan sont donnés sous le dessin du plan):



ouest, sud, est, nord, sud

ouest, est, sud, ouest, sud

Analyse :

Emploi du schéma générer-tester spécifique :
- construire l'entrée (constructive)
- construire la chambre (constructive)
- vérifier que les deux chambres communiquent (test)
- vérifier la contrainte "non opposé"

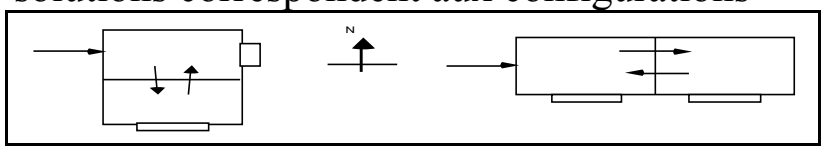
Une solution en Prolog classique (le schéma générer-tester)

```
plan(P_Entree, P_Ch1, F_Ch1, P_Ch2, F_Ch2) :-
 entree(P_Entree, P_Ch1,  F_Ch1),
    chambre(P_Ch2, F_Ch2),
       communique(P_Ch1,P_Ch2),   % elles communiquent
    non_oppose(F_Ch1, F_Ch2).      % la dernière contrainte
```

```
% choisir une porte intérieure, une fenêtre et la porte extérieure tels
% que cette dernière soit dans une direction différente des deux autres
(schéma générer-tester)
entree(P_Entree,P_Ch, F_Ch) :-
 chambre(P_Ch, F_Ch), direction(P_Entree),
 P_Ch \== P_Entree, F_Ch \== P_Entree.
```

```
% prendre une fenêtre et une porte qui ne sont pas sur le même mur
chambre(D, W) :- direction(D), direction(W), D \== W.
communique(X,Y) :- oppose(X,Y).
non_oppose(X,Y) :- oppose(X,Z), Z \== Y.

oppose(nord, sud).    oppose(sud, nord).
oppose(est, ouest).   oppose(ouest, est).

direction(nord).      direction(sud).
direction(est).       direction(ouest).
```

Requête : les solutions correspondent aux configurations




```
?-   P_Entree=ouest, F_Ch2 = sud,
     plan(P_Entree, P_Ch1, F_Ch1,F_Ch2, sud).
=>
   P_Ch1= sud, F_Ch1= est, F_Ch2= nord
   P_Ch1= est, F_Ch1= sud, F_Ch2= ouest
```

Méthodes de résolution de CSP en Prolog

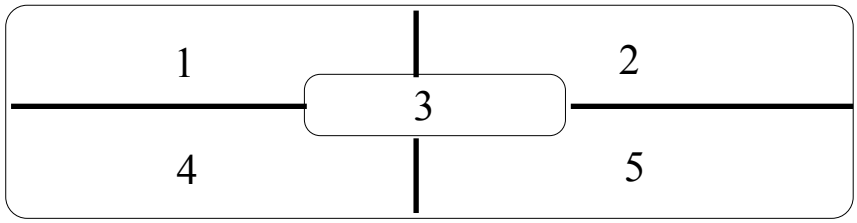


Schéma Constructive

p(...) :-
générer puis vérifier une succession de configurations partielles jusqu'à la configuration finale

```
coloration(C1, C2, C3, C4, C5) :-
   voisin(C1, C2) ,
   voisin(C1, C3) ,
   voisin(C1, C4) ,
   voisin(C2, C3) , voisin(C2, C5) ,
   voisin(C3, C4) , voisin(C3, C5) ,
   voisin(C4, C5) .

couleur(r).
couleur(b).
couleur(v).

voisin(X,Y) :-
 couleur(X) , couleur(Y) , X \==Y.
```

Critiques

- Inefficace (classique)
- Recouvrement a posteriori
- Développement aveugle d'arbre de recherche
- Heuristique, recherche et génération toutes confondues
  => Perte de la déclarativité

Schéma générer/tester

p(...) :-
Génération des configurations totales suivie des
Conditions à vérifier.

On dispose en général d'un prédicat qui teste une configuration

```
coloration(C1,C2,C3,C4,C5) :-
 cinq_couleurs(C1,C2,C3,C4,C5) ,
 different(C1,C2) ,
 different(C1,C3) ,
 different(C1,C4),
 different(C2,C3) , different(C2,C5),
 different(C3,C4) , different(C3,C5) ,
    different(C4,C5) .

couleur(r).
couleur(b).
couleur(v).

different(X,Y) :- X \== Y.

cinq_couleurs(C1,C2,C3,C4,C5) :-
    couleur(C1) , couleur(C2) ,
    couleur(C3) ,
    couleur(C4) , couleur(C5).
```

Critiques

- Inefficace (moins efficace que le précédent)
- Recouvrement a posteriori
- Développement aveugle d'arbre

- Peut être utilisé pour définir un sous cas du problème traité (réutilisation)

Schéma tester-générer (contraindre-générer simple)

```
p(...) :-
    Conditions à vérifier
    Génération des configurations totales.
```

Critiques

- Non disponible en Prolog standard

- Pour les contraintes d'égalité (unification) et d'inégalité, PrologII avait apporté une solution (*eq, dif*).

```
color(C1,C2,C3,C4,C5) :-
 dif(C1,C2) ,            % imposer à C1 et C2 d'être
 dif(C1,C3) ,            % toujours différents
 dif(C1,C4) ,
 dif(C2,C3) ,
 dif(C2,C5) , dif(C3,C4) ,
 dif(C3,C5) , dif(C4,C5),
 couleur(C1) ,     % donner des valeurs aux variables
 couleur(C2) ,
 couleur(C3) ,
 couleur(C4) , couleur(C5).
```

- Recouvrement a posteriori des échecs

Un échec est provoqué par le non respect des contraintes

Ce qui peut être fait : anticipation

C1=r ⇒ C2={b,v}, C3={b,v}, C4={b,v}, C5={r,b,v}
C2=b ⇒ C3={v} ⇒ C4={b} ⇒ C5={r}

Deux choix ont suffisent pour affecter une valeur à toutes les variables.

Insuffisances de Prolog

- U_H et l'interprétation de *Herbrand* (fonctions non interprétées)
  {0, succ(0), succ(succ(0)), ...}

```
add(0, X, X).
add(succ(X), Y, succ(Z)) :- add(X, Y, Z).
```

- Les termes de l'U_H ne suffisent pas dans des applications réelles.

- Certains problèmes sont codés de façon arbitraire en clauses de Horn avec les termes de l'U_H (complétude) :
  (A & B <=> C | D & ~E)

- Peu efficace en arithmétique et en résolution de systèmes d'équations.

- L'arithmétique en Prolog : les variables doivent être instanciées.

=> Ecriture séquentielle,
Perte de la déclarativité
Programmes non réversibles (arguments directionnels)

Exemple : fib en Prolog

```
fib(0,1).
fib(1,1).
fib(N,R) :-
   N>=2, N1 is N-1, fib(N1,R1),
   N2 is N-2, fib(N2, R2),
   R is R1 + R2.
```

- Non réversible : *fib(Y,89)* échoue car dans *N1 is N-1*, N étant une variable, le prédéfini *IS* échoue.
- On échoue sur *N>=2* car ce test réclame des arguments instanciés.
- Utilisation passive des contraintes $\Rightarrow$ inefficace

De Prolog vers les CLP

Conserver

- Aspects déclaratif et relationnel de Prolog
- L'unification (à étendre)
- Possibilité de calcul non déterministe de Prolog
- La séparation de la logique et du contrôle

Apporter

- Enrichir U_H, intégrer les fonctions interprétées sur un domaine
- Modifier la règle de calcul (indépendance de la règle de calcul)
- Introduire des contraintes ($<, \leq, ... \neq$)

  $X+Y < 10 +Z$

  $X \neq Y$
- Traiter des contraintes symboliques par exemple

  $X \in$ [lundi, mardi, ..., dimanche]
- Introduire des techniques de résolution de contraintes

  => La propagation des contraintes

  => Exploitation active des contraintes

  => Recouvrement a priori des échecs

Exemples :

- Version CLP de factorielle :

```
fact(0,1).
fact(N,M*N) :-
   N > 0, fact(N-1, M) .
```

- Calculer le nombre de pattes des lapins (Y) et des pigeons (X), X et Y sont des entiers, + et * ont leur sens arithmétique.

```
nombre_de_tetes_et_de_pattes(X, Y, X+Y, 2*X + 4* Y).
?- nombre_de_tetes_et_de_pattes(6,2,U,V).
=> {U=8, V=20}
?- nombre_de_tetes_et_de_pattes(X,Y,8,20);
=> {X = 6, Y = 2} solution au système {X+Y =8,   2X+4Y = 20}
```

Le problème de planification en PrologIII (contreindre-générer)

• La modélisation (formalisation) en CLP :

Z ={P1,P2,F1,F2,E} les deux portes internes, les 2 fenêtres, l'entrée.

D = {nord, est, sud, ouest} et **C** = {C1..C5}

• On décide que la porte externe est dans la pièce-1 (avec P1,F1).

• Les contraintes :

- L'unité comporte deux pièces rectangulaires;
- Les pièces auront chacune une fenêtre et une porte intérieure;

sont réglées dans le choix de Z et de D.

C1 : - Les pièces communiquent par les portes intérieures;

C2 : - Un mur ne doit avoir qu'une seule fenêtre ou une seule porte;

C3 : - la contrainte "Une des chambres aura une porte vers l'extérieur". Cette porte sera dans pièce-1 et E doit être différente de P1 et F1.

C4 : - Aucune fenêtre ne sera face au nord;

C5 : - Les fenêtres ne doivent pas être sur les cotés opposés de l'unité.

```
maison([E,P1,F1,P2,F2]) ->
 oppose(P1,P2)              % C1
 dif(P1,F1) dif(P2,F2)      %C2
 dif(E,P1) dif(E,F1)              % C3
 non_oppose(F1,F2)          %C5
 domaines([P1,F1,P2,F2,E])     % Les domaines
 {  F1 # nord,  F2 # nord};   % C4

non_oppose(X,Y) -> oppose(X,Z) {Z # Y};
oppose(X,Y) -> cotes_opposes(X,Y);
oppose(X,Y) -> cotes_opposes(Y,X);

cotes_opposes(nord,sud) -> ;
cotes_opposes(est,ouest) -> ;

domaines([]) ->;
domaines([X|Y]) -> membre(X,[nord,sud,est,ouest]) domaines(Y);
```

```
maison([ouest,P1,F1,P2,sud]);        %Les 2 cas du dessin
=> {P1 = est, F1 = sud, P2 = ouest} % Pas besoin d'énumérer
    {{P1 = sud, F1 = est, P2 = nord} % pour cette question.
```

Trace de l'exécution (dif transformé en #) :

```
maison([E,P1,F1,P2,F2]) ->
 oppose(P1,P2) non_oppose(F1,F2)
 domaines([P1,F1,P2,F2,E])     % Les domaines
 {  F1 # nord,  F2 # nord, P1 # F1, P2 # F2, E # P1, E # F1 };

non_oppose(X,Y) -> oppose(X,Z) {Z # Y};
oppose(X,Y) -> cotes_opposes(X,Y);
oppose(X,Y) -> cotes_opposes(Y,X);

cotes_opposes(nord,sud) -> ;   cotes_opposes(est,ouest) -> ;

domaines([]) ->;
domaines([X|Y]) -> membre(X,[nord,sud,est,ouest]) domaines(Y);
```

Trace :

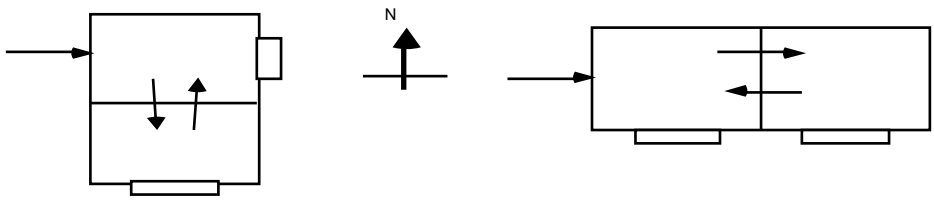
- maison([ouest,P1,F1,P2,sud]);
  => *{E= ouest, F2=sud, F1 # nord,* % dès qu'on rentre
  F2 # nord, P1 # F1, P2 # F2, E # P1, E # F1}
- oppose(P1,P2) => cotes_opposes(P1,P2) => {P1=nord, P2=sud}
  échec car P2=sud et F2=sud mais P2#F2 => **back-track**
  cotes_opposes(P1,P2) => {P1=est, P2=ouest}
- non_oppose(F1,F2) => ¬oppose(F1,X) {F2 # X}
  => cotes_opposes(F1,X) {F2 # X}
  => *F1=nord* **échec** *F1#nord* => **back-track**
  cotes_opposes(F1,X){F2 # X} => {F1=est , F2#ouest}
  échec car *P1=F1=est* mais on a *P1#F1*=> **back-track**
- cotes_opposes(F2,F1) => cotes_opposes(F2,X) {F1 # X}
  => F2=nord => **échec** car F2 =sud déjà => **back-track**
  => **échec** car F2 =sud déjà =>**back-track**
- cotes_opposes(P2,P1) => {P2=nord, P1=sud}
- non_oppose(F1,F2) => ∆oppose(F1,X) {F2 # X} =>
  cotes_opposes(F1,X) {F2 # X}
  => F1=nord **échec** F1#nord => **back-track**
  cotes_opposes(F1,X){F2 # X} => {F1=est , F2#ouest}
  => {E=ouest, P1=sud, F1=est , P2=nord, F2=sud} **est une solution**
- L'autre solution sera trouvée en activant les points de choix.
  => {P1 = est, F1 = sud, P2 = ouest}

On constate que ces solutions seront produites sans l'énumération (du domaine).

Une autre solution utilisant des entiers :

Le domaine **D** = {nord, est, sud, ouest} est représenté par **D** = {1,2,3,4}. Ce qui simplifie les expressions de contraintes.

```
maison([E,P1,F1,P2,F2]) ->
 oppose(P1,P2)              % C1
 dif(P1,F1) dif(P2,F2)      %C2
 dif(E,P1) dif(E,F1)        % C3     domaine([P1,F1,P2,F2,E],1,4)   %
Les domaines
 non_oppose(F1,F2)          % C5
 { F1 # 1, F2 # 1};         % C4 : aucune fenêtre face au Nord

non_oppose(X,Y) -> {X#Y, X+Y # 4, X+Y # 6};
non_oppose(X,Y) -> {X=Y};

oppose(X,Y) -> {X#Y, X+Y = 4};
oppose(X,Y) -> {X#Y, X+Y = 6};

domaine([],_,_) ->;
domaine([X|Y],I,S) -> domaine(Y,I,S) {I<=X <=S };

enum_liste([]) ->;
enum_liste([X|Y]) -> enum(X) enum_liste(Y);
```



Question :

```
maison([4,P1,F1,P2,3]) enum_liste([P1,F1,P2]);
=>   {P1 = 3, F1 = 2, P2 = 1}
   {P1 = 2, F1 = 3, P2 = 4}
```

Intérêts d'un CLP

- L'environnement CLP facilite la programmation déclarative

=> meilleure spécification de l'intuition (l'interprétation attendue)
=> prise en compte directe du domaine de discours (e.g. $\Re$)
=> Le codage en U_H n'est plus utile
=> Les propriétés complexes du problème sont spécifiées d'une façon naturelle (e.g. contraintes arithmétiques, ensemblistes)

Le problème lui même est représenté par un ensemble de règles.

- Les contraintes sont utilisées

- Pour représenter des relations entre objets,
- Pour calculer des valeurs basées sur ces relations (via les fonctions interprétées). Z = 2Y +3 définit une relation entre Z et Y. Elle permet également de calculer la valeur de Z et de Y.

- Règles récursives ⇒ ajout dynamique de contraintes.

- Contrôle de la résolution par les contrainte.

=> On vérifie à chaque pas de la résolution que le système est solvable.

- Utilisation des contraintes pour améliorer la "pureté" de Prolog :

=> Eviter dans certains cas l'utilisation de *cut* et de la *not*

```
p:- q, !, r.
p :- t.
```

Peut être remplacé dans certains cas par :

```
p :- q , r.
p :- q' , t.
```

q' : le complément de q (e.g. $q \in \{=,<\}$, $q' \in \{\neq,\geq\}$).

- Idem pour *not*

=> cf. *pas_dans* au lieu de *not membre*, # au lieu de *not* =, ...

Les domaines X de CLP(X)

- Soit t_1 et t_2 deux termes de premier ordre
- Unifier t_1 et t_2 dans un domaine de calcul D

=> Résoudre l'équation $t_1 =_T t_2$ où T est la théorie d'égalité définie dans D.

Exemple : des règles de l'algèbre de Bool (cf. PIII) employées par un système de réécriture (ordre, confluence, ...):

$(A \Rightarrow B) => \neg A \vee B$ **=>** : se réécrit en

$\neg(A \wedge B) => \neg A \vee \neg B$

$\neg(A \vee B) => \neg A \wedge \neg B$

$(A \Leftrightarrow B)$ -> $(A \Rightarrow B) \wedge (B \Rightarrow A)$

- Une solution à cette équation est une substitution θ qui rend t_1 et t_2 identiques dans la théorie T.

=> La classe des contraintes d'égalité traitées par un langage varie selon la théorie T.

- En Prolog, cette théorie est vide (unification syntaxique des termes U_H)

- Chaque équation doit avoir une seule solution déterministe dans le domaine choisi avec des performances acceptables.

- Deux approches :
  - prédéfinie
  - user-defined (réécriture)

Avantages et inconvénients

Les domaine candidats :

- Domaine des termes de l'arithmétique linéaire (réels, entiers, rationnels).
- Domaine des Booléens
- Chaînes (et tuples finis)
- Nombres Complexes (CAL)
- Domaine finis (problèmes CSP)
- Les ensembles et intervalles (CLP(Σ), BNR)
- Domaine des termes de l'arithmétique **non** linéaire ....

Présentation brève de Prolog-III

N.B. : à Adapter à Gprolog

Calcul bancaire (le taux d'intérêt = 10%)

```
Versement_capital(<>, 0) ->;    % pas de versement pour
                                % un capital nul
Versement_capital(<i>.x, c) -> Versement_capital(x, c+(10/100)c - i);
```

- La seconde règle : on verse **i** francs, puis il faut verser une suite de **x** versements pour rembourser le capital **c** augmenté de 10% d'intérêt mais diminué du versement **i** effectué.

- *Questions :*

```
Versement_capital(<i, 2i, 3i>, 10000);    =>   {i=2076.443}
Versement_capital(<3i, 2i, i>, 10000);    =>   {i = 1948.755}
Versement_capital(<i, i, i, i, i, i>, 10000);=>  {i =  2296.074}
```

Exemples :

```
repas(x,y,z) -> entree(x) plat(y) dessert(z);
repas(x,y,z) -> entree(x) plat(y) dessert(z) {x # salade};
circuit(A,B,C) -> / {C | A & ~B = 0'} ;
somme(<x>.l, s) -> somme(l,s0) , {s = s0 +x};
somme(<>, 0) -> ;
```

Signification d'un programme PrologIII

- Un programme est la définition récursive d'un ensemble d'arbres. Ces arbres sont considérés comme des faits vrais par le programmeur (*sémantique déclarative*).

- Un programme se traduit par une suite d'actions à exécuter : résolution des systèmes de contraintes et les appels de procédures (*sémantique procédurale*).

- Un programme est construit par des règles :

t_0 -> t_1 t_2 ... t_n , S;

t_i est un terme et **S** un ensemble de contraintes.

- Une réponse est une affectation A qui transforme
- Les termes t_i en des arbres $a_i = t_i/_A$ (application de A à t_i)
- L'ensemble S en un ensemble de conditions $C=S/_A$ sur des arbres.

- Une règle engendre une infinité de règles sans variable de la forme

a_0 -> a_1 a_2 ... a_n .

où si a_1... a_n sont des faits alors a_0 est un fait.

- Un programme définit le plus petit ensemble de fait satisfaisant à toutes les propriétés logiques engendrées par les règles du programme.

Exemple : p(x, y) -> q(x) r(y) {x + y < 15};
q(1) -> ;
q(2) -> ;
r(10) ->;
r(20) -> ;

On a
p(1, 10) -> q(1) r(10);
p(2, 10) -> q(2) r(10);

Puisque q(1), q(2) et r(10) sont des faits, alors p(1,10) et p(2,10) seront des faits. Les deux derniers faits n'étaient pas explicites.

Exécution d'un programme Prolog

- L'objectif d'exécution de la suite de termes $\mathbf{t_1\ t_2\ ...\ t_n\ ,\ S}$ est de trouver une affectation qui rend :
  - Les termes t_i des faits et
  - Les contraintes de S en conditions vérifiées.

- Dans la requête $\mathbf{t_1\ t_2\ ...\ t_n\ ,\ S}$
- Si n=0, la requête se résume à demander la résolution du système S;
- Si S est vide et n=1, la requête se résume à la question :

  "pour quelles valeurs des variables, t_1 se transforme en un fait défini par le programme ?".

Notion de machine abstraite

- Instructions de base de la machine non déterministe PrologIII :

(1)	$(W, t_0\ t_1\ ...\ t_n, S)$
(2)	$s_0 \rightarrow s_1\ ...\ s_m\ ,\ R$
(3)	$(W, s_1\ ...\ s_m\ t_1\ ...\ t_n, S \cup R \cup \{s_0 = t_0\}$

(1) : état de la machine à un instant donné.

(2) : une règle du programme permettant de changer d'état.

(3) : le nouvel état de la machine après l'application de (2). Le passage dans cet état n'est possible que si $S \cup R \cup \{s_0 = t_0\}$ possède une solution.

- Départ : pour une requête initiale $t_0\ ...\ t_n$, S, on part de l'état $(W, t_0\ ...\ t_n, S)$ où W est l'ensemble des variables de la requête.

La machine passera dans tous les états permettant d'examiner toutes les règles du programme en (2). L'ordre d'application des règles est défini par le programmeur (l'ordre d'entrée des règles).

- La machine PrologIII est en fait la machine déterministe qui produit toutes les exécutions possibles de la machine non-déterministe.

- Chaque fois que cette machine parviendra à (W, , S), PrologIII fournit comme réponse la solution du système S sur l'ensemble W.

- Exemple :

```
p(x, y) -> q(x) r(y) {x + y < 15};
q(1) -> ;
q(2) -> ;
r(10) ->;
r(20) -> ;
```

La requête p(x,y).

- **Les états :**
- {x,y}, p(x,y), {}
- {x,y} , q(x') r(y'), {x' + y' < 15, p(x',y') = p(x,y)}

simplifié en :

{x,y} , q(x') r(y'), {x' + y' < 15, x=x', y=y'}

- {x,y} , r(y'), {x' + y' < 15, x=x', y=y', q(x') = q(1)}

simplifié en :

{x,y} , r(y'), {x' + y' < 15, x=x', y=y', x'=1}

- {x,y} , , {x' + y'<15, x=x', y=y', x'=1, r(y')=r(10)}

simplifié en :

{x,y} , , {x' + y' < 15, x=x', y=y', x'=1, y' = 10}

- La solution du système S (projetée sur {x,y})

{x=1, y=10}

- Les autres réponses s'obtiennent par l'exploration des autres exécutions possibles de la machine non déterministe (1), (2) et (3).

Exemples d'utilisation de contraintes numériques

PERT en PrologIII

Tâche	durée	précédence	
a	7		
b	3	a	maçonnerie
c	1	b	charpenterie
d	8	a	toiture
e	2	d,c	sanitaire + électricité
f	1	d,c	façade
g	1	d,c	fenêtres
h	3	f	jardin
j	2	h	plafonnage
k	1	e,g,j	emménagement

```
pert ->
 test(<S_a,S_b, S_c, S_d, S_e, S_f, S_g, S_h, S_j, S_k, S_fin>)
 minimize(S_fin)
 affiche(<S_a,S_b,S_c,S_d,S_e,S_f,S_g,S_h,S_j,S_k, S_fin>) ;

test(<S_a, S_b, S_c, S_d, S_e, S_f, S_g, S_h, S_j, S_k, S_fin>) ->
{ 30>= S_a >=0, S_b >= S_a + 7,   S_c >= S_b + 3,
  S_d >= S_a + 7,       S_e >= S_c +1,   S_e >= S_d + 8,
  S_f >= S_d + 8, S_f >= S_c + 1,
  S_g >= S_d + 8,       S_g >= S_c + 1,
  S_h >= S_f +1,  S_j >= S_h + 3,
  S_k >= S_e +2, S_k >= S_g +1, S_k >= S_j +2,
 30>=S_fin >= S_k +1 };

affiche(<>) -> ;
affiche(<x>.l) ->
 outl(x) affiche(l) ;
```

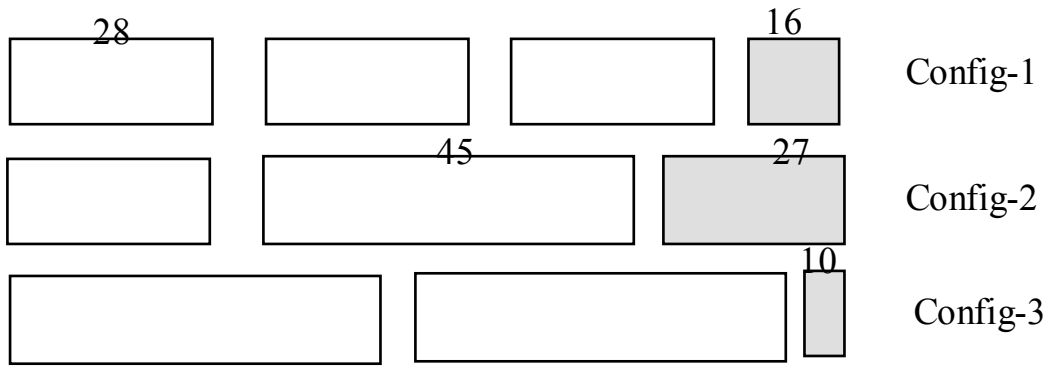
Question :

```
> pert ;
{   S_a = 0, S_b = -A - B - C + 16,   S_c = -B - C + 19,
 S_d = 7,    S_e = -D + 19,S_f = 15,S_g = -C + 20,     S_h = 16,
 S_j = 19,   S_k = 21, S_fin = 22,

 -A - B - C + 9 >= 0, B + C-5 >= 0, -C + 5 >= 0, B - D + C-1 >= 0,
 -D + 4 >= 0, A >= 0, B >= 0, C >= 0,  D >= 0
}
```

=> Pour certains variables, on connaît la valeur exacte.
Pour d'autres, elles varient dans un intervalle.
Pour les connaître, il faut les énumérer : l'énumération des variables 'b', 'e' et 'g' donne une première solution optimale :
=> {**S_b** =11, **S_e** =19, **S_g** = 15}

Découpe de barres de 28 et 45 dans une barre d'un mètre :

Demande à satisfaire : 36 barres de 28 cm et 24 barres de 45 cm




```
decoupe(Chutes, X, Y, Z) :-
 minimize(Chutes) ,          % minimiser les chutes
{  Chutes=16X+27Y+10Z, % système de contraintes
   3X + Y = 36,
   Y + 2Z = 24,
   X>=0, Y>=0, Z>=0 };
```

- decoupe(C,X,Y,Z) ; => {C = 312, X = 12, Y = 0, Z = 12}

A propos de minimize (et maximize)

- Dans

```
membre(X,[2, 9, 1, 4, 5]) minimize(X);
=>  {X = 2},   {X = 9},  {X = 1},  {X = 4},  {X = 5}
```

X prend toutes les valeurs possibles dans [2, 9, 1, 4, 5].

- Cette valeur ne sera pas minimisée. *minimize* intervient après avoir choisi la valeur de X dans [1,2,9,4,5]. Cette valeur est déjà minimum car elle est la seule en jeu .

- *minimize* n'a aucune mémoire des réponses. Il ne fonctionne pas comme un schéma "Branch & Bound" (à voir plus loin).

- Dans

```
minimize(N) { N>= 1, N <= 10, N>=2};=>  {N = 2}
minimize(N) {1<=N<=10};          =>  {N = 1}
```

minimize unifie N avec la plus petite valeur du domaine de N.

- Dans *minimize(X)*, X doit être une expression bornée.

- minimize(N);

Err 59: Expression non bornée lors d'un calcul d'extremum

- minimize(N) {N **>=**1}; => {N = 1}
- minimize(N) {N **>**1}; => échec

- *minimize* est déterministe.

Lorsque *minimize(N)* échoue, N ne prend pas une autre valeur.
Lors d'un "retour-arrière", on ne *re-satisfait pas minimize*.
On peut comparer le comportement du *minimize* avec un ordre d'écriture déterministe comme *out(N)*.

- Lorsqu'une question avec *minimize* échoue, cela veut dire que pour la valeur choisie par minimize, il n'y a pas de solution.

Lors de cet échec, l'on doit prévoir la possibilité d'essayer d'autres valeurs.

Par conséquent, *minimize* est souvent précédé dans ce cas de figure, d'une énumération (une génération) de valeurs.

- **Tests** :

```
fait(1)->; fait(2) -> ;    fait(3) ->;fait(0) -> ;

fait(X)  minimize(X)  {1 <= X <= 3};  % 0 hors intervalle
=>   X=1,   X=2,  X=3
```

Le retour-arrière remonte à fait(X) indépendamment de minimize.

```
fait(X)  minimize(X) {-1 <= X <= 3};
      % X=-1 non confirmé par fait.
=>    {X = 1}  {X = 2}  {X = 3}   {X = 0}
```

- **Par contre (minimize au début) :**

```
minimize(X)  fait(X) {0 <= X <= 3};
=>   X=0
```

Ici, c'est *minimize* qui choisit X=0, *fait(0)* n'est autre qu'une confirmation : "0 est bien un fait".

```
minimize(X)  fait(X) {-1 <= X <= 3};
=>   Echec car minimize choisit X=-1 mais fait ne confirme pas.
```

Résumé (minimize-maximize):
- *minimize* n'a pas de mémoire.
- *minimize* est déterministe.
- X doit être borné dans *minimize(X)*
- *minimize(X) enum(X)* donne X=la première valeur d'enum(X).
- *enum(X) minimize(X)* considère toutes les valeurs d'enum(X).

Un autre exemple de minimize

On considère l'exemple de découpe des barres d'un mètre dans deux versions.

1- version avec "3X + Y = 36, Y + 2Z = 24"

```
decoupe1(Chutes, X, Y, Z) :-
minimize(Chutes) ,
{  Chutes=16X+27Y+10Z,
   3X + Y = 36, Y + 2Z = 24,
   X>=0, Y>=0, Z>=0 };
```

- Avec l'égalité, les contraintes sont plus fortes et on peut voir la forme simplifiée de la variable *Chutes = -50X + 912* obtenue par la simplification {Y = -3X + 36, Z = (3/2)X-6}

- C'est donc "-50X + 912" qui doit être minimisé. Etant donné les bornes inférieures de X, Y et de Z (=0), on a :
  3X+Y=36 ✝ X=12-Y/3
  min(Y) = 0 => min(X) = 12 => min(Chutes) = 312

- La valeur *Chutes = 312* sera **imposée** et donne la solution :

decoupe1(C,X,Y,Z) ; => {C = 312, X = 12, Y = 0, Z = 12}

- **decoupe1**(C, **13**, Y, Z) ; => ECHEC.

car il n'y a pas de solution pour {X=13, 312 = -50X + 912}.

2- Considérons la seconde version **decoupe2** avec :
3X + Y >= 36 , Y + 2Z >= 24

- Ici, les contraintes sont plus souples et on peut voir la forme simplifiée de la variable *Chutes = 10Z + 27Y + 16X*.

- Avec l'expérience de l'exemple précédent, si l'on ajoute la contrainte *Chutes < 312*, on obtient un système inconsistant (programme non chargeable).

- Les questions :

decoupe2(C,X,Y,Z); => {C = 312, X = 12, Y = 0, Z = 12}
decoupe2(C,**13**,Y,Z); => {C = 328, Y = 0, Z = 12}.....

Booléans

Les opérateurs booléennes

~X X & Y X | Y X => Y X <=> Y

Les contraintes booléenne

- **!boolt** : imposer à un arbre d'être étiqueté par une valeur booléenne.
- **!bool** : imposer à un arbre d'être réduit à une feuille et étiqueté par une valeur booléenne (équivalent à **b !boolt** et **b::0**).

- = : égalité
- # : différence
- => : implication '=>' est à la fois relation et opération

- Une contrainte est vérifiée ou non; mais ne renvoie pas de valeur. On ne pourra donc pas écrire {a | b, b#a} car '|' n'est pas une relation et 'a|b' n'est pas une contrainte mais une expression. De même pour (a>0) & (b>0) = 1'.

- **PrologIII ne permet pas de mélanger les algèbres.**

- Une contrainte booléenne n'est valide que si elle contient exactement un symbole relationnel :

{a => b => c, a # b} contient 2 symboles relationnels => 2 contraintes => est équivalent à **{a => (b => c), a # b}**
=> est simplifié en **{c!bool , a&b = 0', a|b = 1'}**

Dans {a=>b=>c}, le premier '=>' est une relation, le deuxième un opérateur.

Par contre *{(a=b) => (b=a)}* n'est pas valide car elle contient deux symboles relationnels.

- Pour imposer à une variable d'être booléenne :

- La contrainte *{b !bool}*
- La contrainte *{b !boolt, b::0}*
- En faisant figurer b dans une expression booléenne toujours vérifiée (par exemple {b | ~b =1'})

- En faisant figurer b dans une contraintes booléennes de type '=>' toujours vérifiée (par exemple {0' => b})

- Les autres symboles de contraintes booléennes (=, #) ne suffiront pas car ils sont partagés par les autres domaines.

- Dans un ensemble de contraintes, la virgule est une conjonction. Ainsi, si l'ensemble ne contient que des contraintes booléennes, la virgule peut être supprimée dans un ensemble de contraintes équivalent utilisant le connecteur '&' :

{a=1', b=1'} ≡ {a&b=1'} mais pas à {(a=1') & (b=1')} qui n'est pas une forme booléenne valide car un connecteur ne peut pas connecter des contraintes.

Exemples

1- Faire un Ou booléen sur une liste

```
ou(<>,0') -> ;
ou(<x>.l,  x|y) -> ou(l, y);

ou(<1', b1, 0'>,a);          =>  {a = 1', b1 !bool }
ou(<1', b1, 0'>,1');     =>  {b1  !bool }
ou(<x,x,x>,1');          =>  {x = 1'}
ou(<x,y,z>,1');          =>  {x|y|z= 1'}
ou(<x,y,z>,k);           =>  {k => x|y|z, x => k, y => k, z => k}
```

2- Construire un prédicat vrais(K, L, B) où B est la valeur de l'expression*" la liste L contient exactement K éléments vrais"*

- Le paramètre B permet de rendre le programme déterministe si K et L sont connus.

Le raisonnement :

Il y a exactement K éléments vrais dans la liste <x>.L si :

- x est vrai et il y a exactement K-1 éléments vrais dans la liste L
- x est faux et il y a exactement K éléments vrais dans la liste L

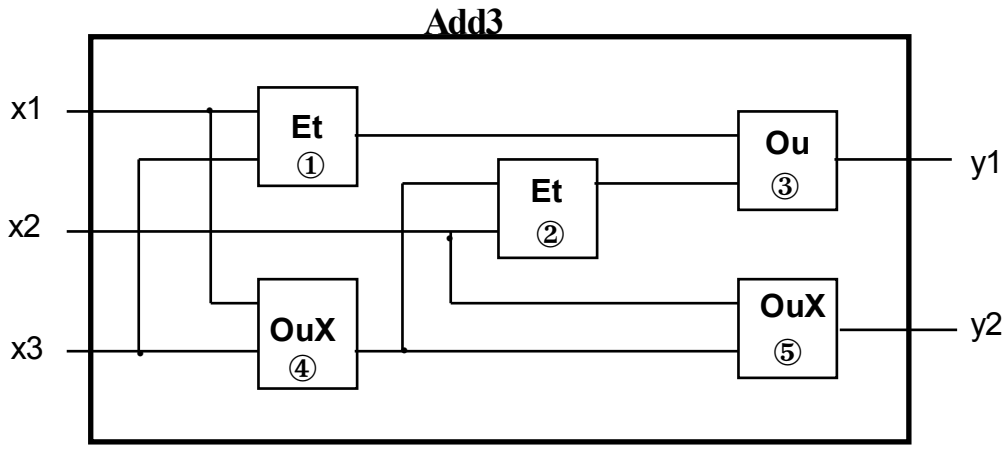
```
vrais(k,<>,0') -> {k#0};
vrais(0,<>,1') ->;
vrais(0,<x>.L, ~x&b) ->
 vrais(0,L, b);
vrais(k,<0'>.L, b) ->
 vrais(k,L, b)
 { k #0};
vrais(k,<1'>.L, b') ->
 vrais(k-1,L, b')
 { k #0};
```

Questions :

```
vrais(3, L, 1') {L::n, n < 5};
 =>   {L = <0',1',1',1'>, n = 4}
 =>   {L = <1',0',1',1'>, n = 4}
 =>   {L = <1',1',0',1'>, n = 4}
 =>   {L = <1',1',1'>, n = 3}
 =>   {L = <1',1',1',0'>, n = 4}
```

2- Panne dans un additionneur binaire

- Soit un additionneur 3 bits :



- On désire écrire un programme qui détecte une panne dans ce circuit selon l'hypothèse de *panne unique* : un seul composant est en panne à la fois.

- Cette hypothèse est vérifiée par le prédicat *AuPlusUnVrai*. Notons que ce prédicat permet d'installer une contrainte avant de décrire le circuit (méthode contraindre/générer).

- Une valeur booléenne 1' sur les porte p1..p5 veut dire que celle-ci est en panne.

- Une Contrainte comme *~p1 => (u1 <=> x1 & x3)* veut dire : si P1 n'est pas en panne alors p1 vérifie la relation (électronique) qui lui est attribuée.

```
Circuit(<x1,x2,x3>, <y1,y2>, <p1,p2,p3,p4,p5>) ->
 AuPlusUnVrai(<p1,p2,p3,p4,p5>) ,
    { ~p1 => ( u1 <=> x1 & x3 ),
      ~p2 => ( u2 <=> x2 & u3 ),
      ~p3 => ( y1 <=> u1 | u2 ),
      ~p4 => ( u3 <=> ~(x1 <=> x3)),
      ~p5 => ( y2 <=> ~(x2 <=> u3))  };

AuPlusUnVrai(P) -> OuSurAuPlusUnVrai(P,p);

OuSurAuPlusUnVrai(<>, 0') ->;
OuSurAuPlusUnVrai(<p>.P,  p|q) ->
 OuSurAuPlusUnVrai(P, q) ,
 {p&q = 0'};
```

- **Questions :**

```
Circuit(<1',1',0'>,<0',1'>,<p1,p2,p3,p4,p5>);
 =>   {p1 = 0', p2 = 0', p3 = 0', p4 = 1', p5 = 0'}
```

Panne dans la porte 4.

```
Circuit(<1',0',1'>,<0',0'>,<p1,p2,p3,p4,p5>);

 =>   {p2 = 0', p4 = 0', p5 = 0', p1&p3 = 0', p1|p3 = 1'}
```

Une seule des portes parmi 1 ou 3 est déficiente

```
Circuit(<0',0',1'>,<0',1'>,<p1,p2,p3,p4,p5>);
{   p4_1&p5_1 = 0',
 p3_1&p5_1 = 0',
 p3_1&p4_1 = 0',
    p2_1&p5_1 = 0',
 p2_1&p4_1 = 0',
 p2_1&p3_1 = 0',
    p1_1&p5_1 = 0',
 p1_1&p4_1 = 0',
    p1_1&p3_1 = 0',
 p1_1&p2_1 = 0'}
```

Aucune porte en panne !

Casse-tête logiques

1- Lewis Caroll

- La casse-tête logique suivante est une suite de phrases proposée par Lewis Caroll. On peut présenter ces phrases sous forme de proposition par une système de contraintes. On peut ensuite s'amuser à poser des questions pour rechercher des liens qui existent entre ces propositions. C'est donc un exemple typique pour lequel la solution recherchée ne peut se présenter que sous la forme d'une simplification sur variables.

- Ces propositions sont : <photocopie>

On considère maintenant les 18 phrases d'un puzzle de **Lewis** Carroll [7] que nous reproduisons ci-dessous. Il s'agit de répondre à des questions du genre : « quel lien existe-t-il entre avoir l'esprit clair, être populaire et être apte à être député ? » ou « quel lien existe-t-il entre savoir garder un secret, être apte à être député et valoir son pesant d'or ?».

1. Tout individu apte à être député et qui ne passe pas son temps à faire des discours, est un bienfaiteur du peuple.
2. Les gens à l'esprit clair, et qui s'expriment bien, ont reçu une éducation convenable.
3. Une femme qui est digne d'éloges est une femme qui sait garder un secret.
4. Les gens qui rendent des services au peuple, mais n'emploient pas leur influence à des fins méritoires, ne sont pas aptes à être députés.
5. Les gens qui valent leur pesant d'or et qui sont dignes d'éloges, sont toujours sans prétention.
6. Les bienfaiteurs du peuple qui emploient leur influence à des fins méritoires sont dignes d'éloges.
7. Les gens qui sont impopulaires et qui ne valent pas leur pesant d'or, ne savent pas garder un secret.
8. Les gens qui savent parler pendant des heures et des heures et qui sont aptes à être députés, sont dignes d'éloges.
9. Tout individu qui sait garder un secret et qui est sans prétention, est un bienfaiteur du peuple dont le souvenir restera impérissable.
10. Une femme qui rend des services au peuple est toujours populaire.

- Le prédicat *CasPossible* suivant associe les phrases proposées aux variables booléennes sur lesquelles il pose les contraintes :

```
CasPossible(<
 <a,"avoir l'esprit clair">,
 <b,"avoir reçu une bonne éducation">,
 <c,"discourir sans cesse">,            %affirmation 1
 <d,"employer son influence a des fins méritoires">,
 <e,"être affiche dans les vitrines">,
 <f,"être apte a être député">,         %affirmation 1
 <g,"être un bienfaiteur du peuple">,           %affirmation 1
 <h,"être digne d'éloges">,
 <i,"être populaire">,
 <j,"être sans prétention">,
 <k,"être une femme">,
 <l,"laisser un souvenir impérissable">,
 <m,"posséder une influence">,
 <n,"savoir garder un secret">,
 <o,"s'exprimer bien">,
 <p,"valoir son pesant d'or">>) ->
{ (f&~c) => g,                          %affirmation 1
 (a&o) => b,
 (k&h) => n,
 (g&~d)=> ~f,
 (p&h) => j,
 (g&d) => h,
 (~i&~p) => ~n,
 (c&f) => h,
 (n&j) => (g&l),
 (k&g) => i,
 (p&c&l) => e,
 (k&~a&~b) => ~f,
 (n&~c) => ~i,
 (a&m&d) => g,
 (g&j) => ~e,
 (n&d) => p,
 (~o&~m) => ~k,
 (i&h) => (g|j) };
```

- Le reste du programme vérifie qu'une liste fournie en entrée est un sous ensemble de la liste donnée dans *CasPossible*. Les contraintes sont alors installées et la simplification nous permet de voir les liens entre les variables du sous-ensemble considéré dans la question.

```
Possibilite(x) -> CasPossible(y) SousEnsemble(x,y);

SousEnsemble(<>,y) ->;
SousEnsemble(<e>.x,y) -> ElementDe(e,y) SousEnsemble(x,y);

ElementDe(e,<e>.y) ->;
ElementDe(e,<f>.y) -> ElementDe(e,y) {e#f};
```

Questions :

- Etablissons les rapports éventuels qui existent entre “avoir l'esprit clair”, “être populaire” et “savoir garder un secret" :

```
Possibilite(<<p,"avoir l'esprit clair">,
   <q,"être populaire">,
   <r,"savoir garder un secret">>);   =>  {}
```

La réponse est l'ensemble vide et signifie que selon Lewis Caroll il n'y a aucun lien entre ces trois vertus.

- Quel sont les liens qui unissent les propositions "savoir garder un secret", "être apte a être député" et "valoir son pesant d'or" ?

```
Possibilite(<<p,"savoir garder un secret">,
      <q,"être apte a être député">,
      <r, "valoir son pesant d'or">>); =>  {p & q => r}
```

La réponse exprime le fait que si l'on sait garder un secret et que l'on est apte à être député alors on vaut son pesant d'or.

- Quel sont les liens entre les propositions "être une femme" et "être apte a être député" selon Lewis Caroll?

```
Possibilite(<<p,”être une femme">, <q,"être apte a être député">>);
   =>   {p & q =  0'}
```

Les deux semble **incompatibles** !!

2- Début de preuve d'existence de Dieu

Il s'agit de montrer que "quelque chose a toujours existé" à partir des 5 propositions de George Boole.

1- Quelque chose existe;
2- Si quelque chose existe alors, soit quelque chose a toujours existé, soit les choses qui existent maintenant sont sorties du néant;
3- Si quelque chose existe alors, soit ce quelque chose existe par la nécessité de sa propre nature, soit ce quelque chose existe par la volonté d'un autre être;
4- Si quelque chose existe par la nécessité de sa propre nature alors quelque chose a toujours existé;
5- Si quelque chose existe par la volonté d'un autre être alors l'hypothèse que les choses qui existent maintenant sont sorties du néant, est fausse.

L'ensemble de ces faits sera l'ensemble des arbres de la forme *ValeurDeQuelqueChoseAToujoursExiste(x)* où x désigne la valeur de vérité pour la proposition "quelque chose a toujours existé".

Les proposition de George Boole sont codées par l'introduction de 5 variables qui sont les valeurs de vérité possibles des 5 propositions :
a : Quelque chose existe
b : Quelque chose a toujours existé
c : Tout ce qui existe maintenant est sorti du néant
d : Quelque chose existe par la nécessité de sa propre nature
e : Quelque chose existe par la volonté d'un autre être

```
ValeurDeQuelqueChoseAToujoursExiste(b) ->
  { a = 1',
    a => (b|c)&~(b&c),
    a => (d|e)&~(d&e),
    d => b,
    e => ~c };
```

Pour résoudre le problème, on pose la question :

```
ValeurDeQuelqueChoseAToujoursExiste(b);=>{b=1'}
```

Caractéristiques de la CLP

- Spécification d'un programme ≡ expression des contraintes

- En CLP, les contraintes sont utilisées pour spécifier les entrées et les sorties des programmes :

*Contraintes en entrées => **Programme** => Contraintes en sortie*

- Un programme transforme (simplifie, résout) les contraintes en entrées et produit les contraintes en sortie.

- Le langage est déclaratif
  => les contraintes non directionnelles

- L'utilisation de CLP facilite la programmation déclarative grâce à une meilleure spécification de l'intuition.

- On peut travailler directement dans le domaine du problème (e.g. R) sans devoir coder les termes par ceux de U_H.

- **Le solveur de contraintes contrôle la résolution**.

Il doit pouvoir déterminer à chaque étape si une collection de contraintes est solvable (consistant) ou non.

- Dans un programme CLP :

- On spécifie les propriétés (complexes) du problème d'une façon naturelle par des contraintes.
- Le problème lui même est représenté par un ensemble de règles.

Exemple :

```
circuit_parallele (V, I, R1, R2) ->
   resistance (V, I1, R1)
   resistance (V, I2, R2)
   {I1 + I2 = I};

resistance(V, I, R) -> {V = I * R};
```

- Les règles récursives permettent d'ajouter dynamiquement des contraintes.

```
Chiffres_differents(<>) ->;
Chiffres_differents(<x>.s) ->
 Hors_de(x, s)
 Chiffres_differents(s),
 {0 <= x <= 9};

Hors_de(x, <>) ->;
Hors_de(x, <y>.s) -> Hors_de(x, s), {x#y};
```

- Le solveur incrémental de CLP vérifie la consistance du système de contraintes à toute étape de la résolution.
  (Simplex, Gauss, ... incrémental)

=> Ce comportement est illustré par la *machine abstraite* PrologIII.

CLP et les problèmes combinatoire

- CLP est utilisé pour la recherche d'une meilleure solution dans le CSP.

- Pour un solveur idéal (parfait), les contraintes suffisent pour définir les réponses sans avoir recours aux générateurs :

=> Il n'y a pas de solveur parfait

=> Il faut mettre en place une heuristique conjointement avec un solveur partiel.

=> Générer des valeurs si nécessaire

- Dans les problèmes combinatoires, on utilise les CLP pour traiter les problèmes sur les entiers, booléens, ... par un solveur partiel.

- Le schéma général est le schéma **contraindre-générer**

  P(----) : -

  contraintes,

  générateurs.

=> Les contraintes sont installées avant les générateurs.

=> Les contraintes sont utilisées pour **guider** la résolution en générant des valeurs et en supprimant des branches quand les contraintes sont non satisfiables.

=> Les valeurs générées sont sous le contrôle du système de contraintes.

=> On évite de générer des valeurs quand les contraintes sont inconsistantes

=> PIII refuse de charger une règle sont les contraintes sont inconsistantes. Par contre, le message affiché est déroutant :

- Lecture d'un fichier contenant :

```
pp(x) -> {x>0, x<0} ;
```

=> Erreur dans insert ou *assert* ou *in_term* .

- Même règle tapée au clavier donne lieu au message

=> Le système de contraintes de la règle est insatisfaisable.

Remarque : Mettre les contraintes à la fin d'une règle en PIII n'est qu'une question syntaxique. Elle sont prises en compte dès le choix de la règle (voir la machine abstraite PrologIII).

Notion de contrainte-réponse

- Un programme transforme (simplifie, résout) les contraintes en entrées et produit les contraintes en sortie.

Les sortie sont appelées **contraintes-réponses**

- Une contrainte-réponse est une évaluation partielle du programme : une instance plus spécifique du programme.

Exemple :

```
versement(_Capital, _Mois, _Taux, _Due, _Mensualite) ->
   {_Mois = 1,
_Due = _Capital + (_Capital *_Taux - _Mensualite) };

versement(_Capital, _Mois, _Taux, _Due, _Mensualite) ->
   versement(_Capital*(1 + _Taux) - _Mensualite,
            _Mois - 1, _Taux, _Due, _Mensualite)
{_Mois > 1};
```

Question : Combien emprunter et quelle est la mensualité sur 12 mois à un taux annuel de 10% pour que l'on ne doive rien à la fin.

versement(P, 12, 0.1, 0, M);

{P = _Capital, M = 0.147 * _Capital}

=> P = 6.814*M

=> Emprunter **6,8** fois la somme remboursée chaque mois (mais rembourser **12** mois) : le prêteur double son capital.

Changement de domaine

- Codage d'un domaine par un autre : booléen par {0,1}, symbolique par entiers, ...

- Le changement de domaine peut être nécessaire dans les cas suivants :

- Le langage ne permet pas un codage directe du domaine.

Exemple : comment dire X ∈ {lundi mardi, ... , dimanche} en PIII en évitant les contraintes disjonctives.

- La solution doit être optimisée (en évitant par exemple les contraintes disjonctives).

Exemple : Exemple d'entrepôts

- Le changement de domaine est une technique pratique courante.

Quelques schémas de CLP

- Plusieurs schéma dérivés du schéma général

- Schéma général contraindre-générer :
  P(----) : -
  contraintes,
  générateurs,
  vérification des propriétés de la solution.

Exemple- Crypto-arithmétique :
GERALD + DONALD =ROBERT

```
Solution(i, j, k) ->
  Chiffres_tous_differents(<g,e,r,a,l,d,o,n,b,t>)     Contraintes
  Tous_entiers(<g,e,r,a,l,d,o,n,b,t>),              Génération
  { g#0, d#0, r#0,
    i = 100000g + 10000e + 1000r + 100a+ 10l + d ,        Propriétés
    j = 100000d + 10000o + 1000n + 100a +10l + d ,      de la
   k = 100000r + 10000o + 1000b + 100e + 10r + t ,     solution
    i+j=k };

Tous_entiers(<>) ->;                        Génération
Tous_entiers(<e>.r) -> enum(e) Tous_entiers(r);  récursive

Chiffres_tous_differents(<>) ->;            Contraintes
Chiffres_tous_differents(<x>.s) ->           installées
 Hors_de(x, s)                             par des
 Chiffres_tous_differents(s),               règles
 {0 <= x <= 9};                           récursives

Hors_de(x, <>) ->;
Hors_de(x, <y>.s) -> Hors_de(x, s), {x#y};
```

Question :

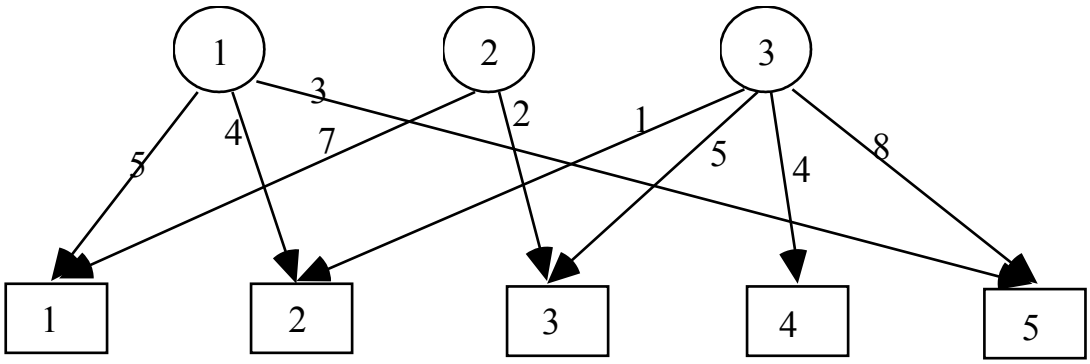
```
Solution(i,j,k); => {i = 197485, j = 526485, k = 723970}
```

Le schéma général des spécifications d'un problème combinatoire :

P(------) : -
les contraintes primitives (un coût à minimiser, ...)
les contraintes exprimées avec les prédicats;
les générateurs de valeurs.
vérification des propriétés de la solution.

Exemple : Problème de localisation d'entrepôts

Objectif : Satisfaire les demandes des clients au moindre coût en implantant des entrepôts de marchandises dans diverses régions.



- Chaque entrepôt a un coût fixe d'ouverture et de maintenance. On a les coûts 18, 10 et 28 pour les entrepôts E1, E2 et E3
- Un client n'est servi que par un seul entrepôt.
- Servir tous les clients

BUT :

Trouver une configuration telle que le coût total soit inférieur ou égal à une limite (ici 60).

Modélisation :

- On associe à chaque client C_i (I = 1..5) les variables C_{ij} (j = 1..3). C_{ij} = 1 si le client i est fourni par l'entrepôt j et 0 sinon.
  => entier utilisé au lieu de booléen

- $E_{j,\ i=1..3}$ représentent les entrepôts

Contraintes :

• Un client n'est servi que par un seul entrepôt :

soit $C_{ij,\ j=1..3}$ les entrepôts qui peuvent servir $C_{i,\ i\in 1..5}$.

=> $\sum C_{ij} = 1$ pour un i donné.

=> Chaque client est forcément servi par un et un seul entrepôt.

Propriétés de la solution :

• La solution doit être telle qu'un entrepôt non sélectionné ne doit pas figurer comme fournisseur d'un client.

Si $E_{j,\ j=1..3} = 0$ pour un j donné alors il ne doit pas y avoir un client pour le même entrepôt.

Ej=1 => la somme des K clients servis par l'entrepôt j $\leq$ K

Ej=0 => la somme des K clients servis par l'entrepôt j = 0

=> $\sum C_{ij} \leq k*Ej$ pour un j donné, k $\leq$ 5 est le nombre des clients effectivement servi (ici, la solution est connue) par Ej.

On a :

Ej=0 => $\sum C_{ij} = 0$ et

Ej=1 => $\sum C_{ij} \leq k$

"$\leq$ " veut dire : Ej ne servira peut-être pas tous ses clients potentiels.)

- Le Cout_total = le coût de transport pour chaque couple Cij + le coût fixe d'ouverture et de maintenance de Ej. Cout_total $\leq$ Limite.

Si Cij=0 ou Ej=0 alors le coût correspondant sera = 0.

Solution :

```
approvisionnement(_Clients, _EntrePot , _Cout, _Limite) ->
   configuration(_Clients, _EntrePot , _Cout)
{_Cout <=_Limite};                    C. Primitive

configuration(_Clients, _EntrePot , _Cout) ->
   eq(_Clients,<C11, C12, C21, C23, C32, C33, C43, C51, C53>)
eq(_EntrePot , <E1, E2, E3>)
genere(_Clients . _EntrePot )         Génération

      {C11 + C12 = 1,        % l'un ou l'autre des
      C21 + C23 = 1,         %entrepôts pour le
      C32 + C33 = 1,         % client 1, deux et 3    Contraintes
      C43 = 1,               % client4 est fourni par E3
      C51 + C53 = 1,

      C11 + C21 + C51 <= 3 * E1,          Propriétés
      C12 + C32 <= 2 * E2,                de la
      C23 + C33 + C43 + C53 <= 4 * E3,          solution

      _Cout = 5 * C11 + 7 * C12 + 4 * C21 + C23 +      suite des
   2 * C32 + 5 * C33 + 4 * C43 +3 * C51 +     Contraintes.
   8 * C53 + 18 * E1 + 10 * E2 + 28 * E3} ;   en PIII, à la fin

genere(<>) -> ;          % simplifié pour cet exemple car on sait 0,1
genere(<1>. L) ->        genere(L) ;
genere(<0 >. L) ->       genere(L);
```

Exemple

```
approvisionnement(_Ts, _Ls, _Cout, 60);
{_Ts = <0,1,0,1,1,0,1,0,1>, _Ls = <0,1,1>, _Cout = 60}
```

C'est à dire :
E2 et E3 implantés
E2 sert C1 et C3, E3 sert C2, C4 et C5
Le coût = 60

Une autre solution (booléenne avec minimize) :

- Choix des variables C_{ij} et E_j booléennes
- C_{ij} = vrai => C_i servi par E_j => E_j=vrai
- Un client n'est servi que par un seul entrepôt :
  Ou-exclusif sur les mêmes clients ($C_{11} \oplus C_{12}$)
- Le coût total = le même que dans la version précédente.

```
entrepots(_Clients, _Entrepots, _Limite, _Cout) ->
 configuration(_Clients,_Entrepots, _Cout) {_Cout <= L};

configuration([C11, C12, C21, C23, C32, C33, C43, C51, C53],
                                [E1, E2, E3], _Cout) ->

 seulement_un(C11, C12)        % C1 servi par un seul
 seulement_un(C21, C23)  % des E1 ou E2.
 seulement_un(C32, C33)  % C'est le OU-exclusif
 seulement_un(C51, C53)
 cout_transport([cc11(C11),cc12(C12),cc21(C21),
          cc23(C23), cc32(C32), cc33(C33), cc43(C43),
          cc51(C51), cc53(C53)],  _Cout1)

 cout_fixe([ee1(E1), ee2(E2), ee3(E3)] , _Cout2)

 % le système booléen : C4 est servi par E3
 % Cij => Ej     mais l'inverse n'est pas vrai.

 {  C43=1',  E3=1',              % E3 est vrai car C43 est vrai.
    C11 => E1, C21 => E1, C51 => E1,    % mais pas E1 => C51
    C12 => E2, C32 => E2,  % car C5 peut être servi
    C23 => E3, C33 => E3,  % par un autre     entrepôt   C53 => E3,

    _Cout = _Cout1 + _Cout2};
```

```
seulement_un(X,Y) ->  {X&Y=0', X|Y=1'};       % OuX

cout_transport([],0) ->;
cout_transport([X|Y],S1+S) ->  cout(X,S1)  cout_transport(Y,S);

cout_fixe([],0) -> ;
cout_fixe([X|Y],S1+S) ->  cout(X,S1)  cout_fixe(Y,S);

cout(ee1(1'),18) ->  ;
cout(ee2(1'), 10) ->  ;
cout(ee3(1'), 28) ->  ;

cout(cc11(1'),5) ->  ;
cout(cc12(1'),7) ->  ;
cout(cc21(1'),4) ->  ;
cout(cc23(1'),1) ->  ;
cout(cc32(1'),2) ->  ;
cout(cc33(1'),5) ->  ;
cout(cc43(1'),4) ->  ;
cout(cc51(1'),3) ->  ;
cout(cc53(1'),8) ->  ;

cout(X(0'),0) -> ;  % attention: racine X variable
;
```

Question :

```
go(C, E, 60, _cout);
{C = [0',1',0',1',1',0',1',0',1'],  E = [0',1',1'], _cout=60}
```

=> E2 et E3 implantés, E2 sert C1 et C3, E3 sert C2, C4 et C5
Le coût = 60

```
go(C, E, 65, _cout);
{C = [1',0',0',1',0',1',1',1',0'], E = [1',0',1'], _cout= 64}
{C = [0',1',0',1',1',0',1',0',1'], E = [0',1',1'], _cout= 60}
{C = [0',1',0',1',0',1',1',0',1'], E = [0',1',1'], _cout= 63}
```

Raisonnement hiérarchique

- Un contrainte représente une propriété locale du problème à résoudre.

- Les contraintes représentent les relations entre les différents objets du problème.

```
resistance(V, I, R) -> {V = I * R};
```

- Les contraintes globales décrivent comment les contraintes locales interagissent.

- Dans CLP, les propriétés globales sont représentées par les règles.

=> Les contraintes-réponses expriment des contraintes globales.

- Le schéma de représentation des propriétés globales d'un CLP:

  P(----) :-
  contraintes,
  P1(----), ..., Pn(----)

```
circuit_parallele (V, I, R1, R2) ->
   resistance (V, I1, R1)
   resistance (V, I2, R2)
   {I1 + I2 = I};

circuit_serie (V, I, R1, R2) : -
   resistance (V, I1, R1)
   resistance (V, I2, R2)
   {V1 + V2 = V};
```

- Expression d'une **hiérarchie** multi-niveaux :

```
circuit_parallele_serie(VI, R1, R2, R3, R4) ->
   circuit_parallele (V1, I, R1, R2)
   circuit_parallele(V2, I, R3 R4)
   {V1 + V2 = V};
```

Conclusion

- CLP permet une spécification déclarative des problèmes
- Le programmeur est libéré des considérations opératoires; il se concentre sur la définition des relations et les contraintes
- Le système de programmation s'occupe de la résolution dirigée par les contraintes.
- Il n'y a plus de système de résolution spécifique à inventer pour chaque problème.
- Le programmeur formalise le problème dans le langage CLP en suivant les schémas de spécification.
- La tâche de la spécification d'une solution :

- Faire un découpage hiérarchique ;
- Trouver une définition par contraintes;
- Spécifier les relations;
- Spécifier les générations de valeurs;
  - Vérifier (éventuellement) les réponses.

NB : La partie qui vient du fichier Trans-cours-7-et-8-98 ./..

Méthodes et Techniques de résolution de contraintes

Spécification de Programme = Expression de Contraintes

- L'énoncé (le texte) d'un problème de contraintes CSP correspond à deux sortes de besoins :

  ⓾ **La recherche de solutions (satisfaction de contraintes)**

  - Trouver une solution
  - Trouver toutes les solutions
  - Trouver une solution optimale pour un critère donné

  ③ Approche appliquée en Recherche Opérationnelle.

  ❡ **La modification de solutions (propagation de valeur)**

  En partant d'une solution, le solveur en cherche de nouvelles en modifiant différentes valeurs de variables et/ou de champs d'objets (par exemple : Branch & Bound)

- Les CP utilisent en général plusieurs techniques de résolution appelées *techniques de satisfaction de contraintes.*

- Ces techniques sont (ordre chronologique) :

L'unification (une propagation "primitive")
Le retardement
La propagation locale (propagation de valeurs)
La relaxation
La résolution d'équation/inéquation
La propagation de contraintes
.....

Exploitation de contraintes : passive et active

- L'exploitation des contraintes peut être faite d'une façon active ou passive (comme un simple test de Prolog).

Exemple : Coloration d'une carte :

```
Color(A,B,C,D) :-   nonegal(A,B)  nonegal(A,C)  nonegal(B,C)
        nonegal(B,D)  nonegal(A,D)  nonegal(C,D)
        enumere([A,B,C,D],1,5);
```

Code passif (à la Prolog classique)
nonegal(X,Y) :- Si X et Y sont connues alors tester X≠Y
Sinon échouer/erreur...

Code actif (à la CLP)
nonegal(X,Y) :- Si X est connu alors **ne**(Y,X). % Y ≠ X
nonegal(X,Y) :- Si Y est connu alors **ne**(X,Y). % X ≠ Y

Et quand on affirme X≠Y, on en tire des conclusions (propagation)
ne(X,1) :- Si X ∈ 2..5 alors succès.
ne(X,1) :- Si X =1 alors échec.
ne(X,1) :- Si X ∈ 1..2 alors X=2.
ne(X,1) :- Si X ∈ 1..3 alors X ∈ 2..3....
idem pour ne(X,2) .. ne(X,5).

☞ De cette manière, on évitera l'énumération exhaustive à chaque étape et par endroit, on fait des assertions directes.

- La plupart des CLP actuelles exploitent activement les contraintes, en particulier en domaine fini.

Mécanisme de bas niveau : certains systèmes permettent de *vérifier* une contrainte (par **ask**) ou d'en *ajouter* une (par **tell**).

Exemple : Combinaison de couleurs

Soit X,Y∈ {jaune, bleu, rouge, vert, orange} et
la relation **compatible(X,Y)** est vraie si X et Y "s'accordent".

```
compatible(jaune, bleu).
compatible(jaune, rouge).
compatible(bleu, jaune).
compatible(rouge, jaune).
compatible(vert, orange).
compatible(orange, vert).
```

On veut définir la contrainte **paire(X,Y)** pour attribuer des valeurs compatibles à X et à Y.
Cette contrainte peut être définie de différentes manières :

1- Résoudre (à la prolog) en faisant des appels à compatible/2.

```
paire1(X, Y) -> compatible(X, Y);     % exploitation passive
```

Question : combinaison tricolores

```
dif(X,orange) paire1(X,Y) paire1(Y,Z);
③ {X = jaune, Y = bleu, Z = jaune}
{X = jaune, Y = rouge, Z = jaune}
{X = bleu, Y = jaune, Z = bleu}
{X = bleu, Y = jaune, Z = rouge}
{X = rouge, Y = jaune, Z = bleu}
{X = rouge, Y = jaune, Z = rouge}
{X = vert, Y = orange, Z = vert}
```

- On peut également se servir du prédicat *findall* (ou setof) pour constituer les ensembles de valeurs de X et de Y :

```
findall([X,Y],compatible(X,Y), L);

  {L =[[jaune, bleu],[jaune, rouge],[bleu, jaune],
       [rouge,jaune], [vert, orange],[orange, vert]]}
```

2- **paire2(X,Y)** avec propagation après toute modification des variables. Nécessite l'accès de bas niveau aux contraintes, domaines, ...

Le principe :

- Extraire le domaine actuel (**Dx** et **Dy**) des variables X et Y;

- Construire **Lx** la liste des Y compatibles avec le X (par compatible/2);
- De même, construire **Ly** la liste des couleurs **X** compatibles avec le **Y**;
- Poser Dx1 = Lx ∩ Dx et Dy1 = Ly ∩ Dy
- Mettre à jour les domaines de X par Dx1 et de Y par Dy1
- Si l'une des variables X et/ou Y est connue alors trouver l'autre
- Sinon retarder paire2(X,Y) en attendant une modification de X et/ou de Y

Simulation en PrologIII (absence d'accès de bas niveau en PIII) :

```
paire2(X,Y) -> paire2'(      X, [ jaune, bleu, rouge, vert, orange],
                        Y, [ jaune, bleu, rouge, vert, orange]);
paire2'(X,_Dx, Y,_Dy) ->
 findall(A, [compatible(A,Y), membre(Y,_Dy)], _Lx)
    findall(B, [compatible(X,B), membre(X,_Dx) ], _Ly)
    intersec(_Lx,_Dx,_Dx')
 intersec(_Ly,_Dy,_Dy')
 if([free(X),free(Y)], [freeze(X,paire2'(X,_Dx', Y, _Dy')),
                        freeze(Y,paire2'(X,_Dx', Y, _Dy'))]
                ,[ok(X,Y) ,membre(X,_Dx'), membre(Y,_Dy')]);
```

Question :

```
membre(X, [jaune, bleu, rouge, vert, orange]) dif(X,orange) paire2(X,Y) ;
{X = jaune, Y = bleu}        {X = jaune, Y = rouge}  {X = bleu, Y = jaune}
{X = rouge, Y = jaune}       {X = vert, Y = orange}

dif(X,orange) paire2(X,Y) membre(X,[jaune, bleu, rouge, vert, orange]) ;
{X = jaune, Y = bleu}        {X = jaune, Y = rouge}  {X = bleu, Y = jaune}
{X = rouge, Y = jaune}       {X = vert, Y = orange}
```

3- Utilisation des contraintes de plus haut niveau (telle que **element/3**).

```
paire3(X,Y) :-      %sans utiliser compatible/2
element(I, [jaune, jaune, bleu, rouge, vert, orange], X),
element(I, [bleu, rouge, jaune, jaune, orange, vert], Y).
```

4- Manipulation des contraintes en **EcliPse**

```
paire4(A,B) :-
   % extraire les domaines de A et de B
dvar_domain(A,DA),          % DA= le domaine de A
dvar_domain(B,DB),

   %trouver la liste BL des Y compatibles avec ∀X∈ DA
setof(Y, X^ (dom_member(X,DA), compatible(X,Y)), BL),
setof(X, Y^ (dom_member(Y,DB), compatible(X,Y)), AL),

   % transformer ces listes en domaines
sorted_list_to_dom(AL, DA1), sorted_list_to_dom(BL, DB1),

   % Poser DA_New = DA1 ∩ DA. idem pour DB_New
dom_intersection(DA, DA1, DA_New, _),
dom_intersection(DB, DB1, DB_New, _),

   % Imposer ces nouveaux domaines à A et B
dvar_update(A, DA_New), dvar_update(B, DB_New),

   % Si une parmi A,B est connue alors calculer l'autre
   % Sinon suspendre jusqu'une modification des domaines
(var(A), var(B) ->
   suspend(paire4(A,B), 2, [A,B]->any)
;  true
).

% Récupération des domaines : déclaration des domaines :
couleur(A) :- findall(X, compatible(X,_), L), A :: L.
```

Question :

```
?- couleur(A), couleur(B), couleur(C), p4(A, B), paire4(B, C), A##vert.
⇒ B= B{[bleu,vert,rouge,jaune]}
 C= C{[bleu, orange, rouge, jaune]}
 A= A{[bleu, orange, rouge, jaune]}
```

Exploitation et résolution des contraintes : mécanisme de base (**Retardement et Propagation**)

- *Retardement* : retarder l'activation d'un prédicat jusqu'à ce que l'état d'une (ou plusieurs) variable se modifie.

- *Propagation* : transmettre la valeur/domaine d'une variable aux contraintes dans lesquelles elle figure (provoque le *réveil*).

Retardement : Délai

- Mécanisme de *Retardement/Réveil* (**coroutinage**) :

dès que l'état (pas seulement la valeur) d'une variable est modifié, les buts dépendants de cette variable sont réactivés.

- *Freeze* de PrologII/III, *Wait* de Mu-Prolog
- Entraîne une modification de la règle de calcul.

- En CLP, le retardement est souvent combinée avec énumération.

Exemple de coroutinage

Pour implanter "X = Y+Z", on écrit (pseudo-équationnelle) :

```
plus(X,Y,Z) :-
   freeze(X, freeze(Y, Z is X-Y)),
   freeze(Y, freeze(Z, X is Z+Y)),
   freeze(Z, freeze(X, Y is X-Z)).
```

✌ Utilisation de freeze par le programmeur ou *implicite*

Inconvénients du retardement implicite

- Le retardement peut causer une boucle infinie (problème classique en Prolog).

Exemple (CLP(R)):

```
p(X) :- X = Y * Y , p(Y).
p(4).
```

③ La question *?- p(A)* conduit à une dérivation infinie alors que X=4 est une solution.

- Dans le cas de retardement systématique, le solveur ne détermine pas la solvabilité de toutes les contraintes non linéaires.

Exemple (CLP(R)):

```
?- X = Y * Y , X < 0.
```

La réponse de CLP(R)) :

```
0 > X
X = Y * Y
*** Maybe ***
```

Remarques sur la Propagation et le Coroutinage

- *Propagation* : transmettre la valeur/domaine d'une variable aux contraintes dans lesquelles elle figure (pour simplifier les contraintes).

- L'unification est le mécanisme de base transmission de valeur (ou de domaine) d'une variable pour la contrainte d'égalité syntaxique.

La composition de substitutions est une sorte de propagation.

- Il y a **Propagation Locale** et **Propagation de Contraintes**.

- La propagation locale attend qu'il y ait au plus une variable dans la contrainte.

- Les valeurs des variables sont déterminées après un nombre fini d'étapes de propagations locales.

- La propagation est mise en place par *délai/réveil* systématique.

③ Pré traitement des contraintes linéaires (et non linéaires) avant le retardement.

NB: On ne vérifie pas toujours la satisfiabilité des contraintes retardées.

- *Freeze* est un mécanisme de base de *délai/réveil* explicite.

- On peut propager la valeur d'une variable en scrutant les contraintes du système.

- Le retardement et la propagation sont liés (**coroutinage**) : dès que l'état (et non la valeur) d'une variable est modifié, les buts dépendants de cette variable sont réactivés, des valeurs sont calculées puis propagées.

- Les CLP actuels étendent la propagation aux contraintes : transmission de la valeur/domaine d'une variable aux autres variables du système de contraintes.

- Les moteurs de résolution dans les CLP s'appuient sur la propagation. Elles divergent ensuite selon leur spécificité :

③ l'exploitation des domaines finis (cf. CHIP) ou
③ la résolution rationnelle (générale, Simplex) en CLP(R) et PrologIII.

Une application de freeze : Dif (≠)

- Retardement explicite : on peut simplement implanter *dif* par *freeze*

en remplaçant $X \neq Y$:- *not(X=Y).*
par $X \neq Y$:- *freeze([X,Y], not(X=Y)).*

- *dif(t1,t2)* est plus élaboré. Il ajoute la contrainte (t1≠t2) au système de contraintes. Retardement implicite à la simplification de *dif*.

Critique de coroutinage en CSP

- Mécanisme passif : traitement *a posteriori* (réduction d'arbre de recherche a posteriori).

- Pas d'interaction entre les contraintes :

```
p(X,Y) :- X ≥ 5, Y≤ 5.
q(X,Y) :- X ≥ 8, Y ≤ 2.
```

• Question *p(Z,Z)*. fl $Z \geq 5$, $Z \leq 5$ retardés ($Z=5$ est solution)
• Question *q(Z,Z)*. fl $Z \geq 8$, $Z \leq 2$ retardés (inconsistance).

- Le coroutinage permet l'utilisation du schéma générer-tester :

```
p(...) :-
   Retarder les tests  : geler(vars, contraintes)
   Générer des valeurs pour les variables
```

③ Ce schéma est inexploitable dès que la taille du problème devient importante.

- Les problèmes de coroutinage :
  - La découverte continuelle des mêmes faits
  - Générations inutiles et détection tardive des échecs
  - Mauvais points de retours arrière

- Lorsque les domaines sont finis, il faut **éviter** les échecs.

Genèse de la propagation locale

- Technique simple et rapide de résolution de contraintes
- La propagation locale traite une seule contrainte
- Les valeurs connues sont propagées le long des arcs du graphe de contraintes. Lorsqu'un noeud reçoit suffisamment d'information des arcs incidents, il s'active, calcule une ou plusieurs valeurs et propage ses résultats.
- On peut stocker les règles de propagation séparément du mécanisme lui-même.

Les règles de contrôle du noeud "+" :
Z := X+Y, X := Z-Y, Y := Z - X

- **Exemple** : F=1.8 * C + 32

Calcul par la propagation locale:

- Si C est connue, on parcourt la chaîne "normalement" de gauche à droite.
- Si F est connue :
  x1 := F - 32.0 règles sur le noeud "+"
  C := x1 / 1.8 règles sur le noeud "*"
  ③ C := (F - 32.0) / 1.8

- Faiblesse : les variables peuvent être interdépendantes d'une façon cycliques : $x = 2x + 1$ ne peut pas être prise en compte par ce graphe.

Résolution des contraintes statiques

- Stratégies de résolution (planification de satisfaction de contraintes) par la

Propagation d'états connus et la *Propagation de degré de liberté*.

- **Propagation d'états connus :**

Rechercher les parties (variables) qui seront connues à l'exécution. En déduire de proche en proche les parties qui seront connues à l'aide des précédentes, ...

Exemple : dans F=32+9/5 C, on peut déduire facilement que si C est connues alors F l'est aussi et vice versa. Les autres opérandes sont des constantes.

- **Propagation de degré de liberté :**

Degré de liberté : Une constante n'a aucun degré de liberté. Pour une variable X, le degré de liberté se définit en fonction du nombre de variables à connaître pour pouvoir trouver la valeur de X.

Exemple : dans F=32+9/5 C, degré(C)=degré(F)=1 : Pour connaître la valeur d'une variable F, il faut C et vice versa.

Propagation de degré de liberté : au lieu de scruter les valeurs connues, la stratégie est de chercher des entités avec peu de degré de liberté pour satisfaire toutes ses contraintes. On procède ensuite de proche en proche pour aboutir à des parties sans degré de liberté (connues).

Exemple : soient trois vues A, B et C. La hauteur de A est fixée par la taille d'un texte, B est du double de A et C du double de B.

Pour résoudre ces contraintes, on calcule d'abord A par propagation d'états connus, puis B puis C par propagation de degrés de libertés.

- Lorsque le gestionnaire ne peut pas satisfaire les contraintes en une passe, il est souvent possible d'ajouter des contraintes redondantes. Sinon, on utilise l'algorithme de relaxation.

Relaxation : (approximation numérique itérative)

- Par une hypothèse de départ sur les valeurs des variables, on calcule l'erreur et recommence jusqu'une erreur minimum.
- Facile à implanter
- Est utilisé pour résoudre des cycles dans les graphes de contraintes à l'exécution.
- Pour limiter la relaxation à résoudre les cycles sans s'étendre au-delà, on emploie la technique de *propagation de degré de liberté*.

  Propagation de degré de liberté en Relaxation : au lieu de scruter les valeurs connues, la stratégie est de chercher des entités avec suffisamment peu de contraintes pour pouvoir changer leur valeur.

On modifie leur valeur et on les retire du graphe des contraintes avec toutes les contraintes qui leur sont associées.

Exemple : voir la page précédente.

- Faiblesse :

- Limité au domaine numérique et aux contraintes linéaires
- Lente, donne une seule solution,

Transformation de graphes (réécriture)

- X+X =6 ==> petit cycle

On a X+X = 2X ==> 2X =6 plus de cycle

- Puissant : utilisation de contraintes d'ordre supérieur (contraintes sur des contraintes)

- Faiblesse : Difficile à implanter, peu utilisé

Résolution d'équations

- Outil de résolution de systèmes linéaires et non linéaires
- Faiblesse : Difficile à implanter, lente en général.

Dans les systèmes linéaires, les variantes (incrémentales) du Simplexe donnent des résultats intéressants.

Propagation de cohérence (propagation de contraintes)

- Techniques plus récente
- Au lieu de propager des valeurs, on propage des domaines de variation (e.g. sur le maximum et le minimum) et on vérifie la cohérence de ces informations

- La propagation de contraintes + résolution d'équations :

On utilise l'existence de la solution pour vérifier la cohérence à tout moment.

③ La CP devient alors un *outil de preuve*.

Exemple 1: 4-reines

V={V1,...,V4}, Di={A...D} Une reine par ligne

Le placement de V1 réduit le domaine de V2 à {C,D}, V3 à ....

Exemple 2 : SEND + MORE = MONEY

On a R1 R2 R3 R4 R5

S E N D +
M O R E

M O N E Y

• V= {S, E, N, D, M, O, R, Y} ∈ {0..9} et
{R1, R2, R3, R4, R5 } ∈ {0..1}

• Les contraintes M≠0 et S≠0 réduisent leur domaine à{1..9}
R1=M ∈ {0..1} ➢ **M=R1=1**.
On déduit : S≠1, E et O ≠1, R et ≠1, D ≠1 ➢ S ∈ {2..9}
La contrainte R2+S+M=O+10*R1 ➢ R2+S=O+9
En raisonnant sur les bornes, on a : R2(0..1)+S(2..9) ≤ 10
Donc O+9 ≤10. Sachant O≠1, ➢ **O = 0**
et comme R2(0..1)+S(2..9) =9 ➢ [S,E,N,D,R,Y] ≠ 0
La contrainte R3+E=N+10*R2 ➢ R3(0..1)+E2(2..9) ≤ 10
et ➢ N(2..9)+10*R2(0..1) ≤ 10
Puisque 2≤N ➢ **R2 = 0**
Les propagations successives donnent :
S = 9, [E,N,D,R,Y] ≠ 9 ➢ 2≤ [E,N,D,R,Y] ≤ 8

Juste avant les énumérations, on a :
S=9, M=1, O=0, E ∈ {4..7}, [N,R] ∈ {5..8}, Y∈ {2..5}

La stratégie d'énumération

Les variables étant aussi contraintes les une que les autres, la première énumération donne E=4 qui aboutit à un échec.
Avec E=5, la première solution trouvée est :
S=9, M=1, O=0, E=5, N=6, D=7, R=8 et Y=2.

Les retours arrières ne donneront aucune autre solution. Le système a alors *trouvé* une solution, et a *prouvé* qu'elle était unique.

Résolution par propagation généralisée

- Résolution de systèmes de contraintes par des étapes successives de propagation.

- Une étape de propagation locale a lieu quand une contrainte a un nombre suffisant de variables instanciées (généralement, quand il reste une seule variable). Ces variables peuvent participer à d'autres propagations.

Algorithme de principe de la propagation

(pour les systèmes consistants)

```
fonction propagation(C : contraintes; E: environnement ) :
Début                                   environnement =
 Tant que C ≠ {} faire
    E' <- E
    C' <- {}
    Pour tout c∈C telle que c est satisfaite dans E
       E' <- fusion(E', fc(E))        % M.A.J. des domaines
       C' = C' ∪ {c}                  % des variables
    Fin pour
    C <- C - C'                       % Recommencer car
    E <- E'                           % E a changé
 Fin Tant que
 Retourne(E)
Fin
```

- Exemple de résolution par la propagation locale :
  C= {X>0, Y = X+2, Y < 5}

❺ E = [(X=1, Y=3), (X=2, Y=4)]

- Exemple de non satisfiabilité par la propagation locale :
  C= {X>0, Y = X+2, Y < 5, Y > 10}

❺ On **boucle** (inconsistance).

Résolution dans les domaines finis : anticipation

- Lorsque les domaines sont finis, on peut **éviter** les échecs.

③ Eviter les échecs : réduire a priori de l'espace de recherche

- L'anticipation dans les calculs est mise en place par des techniques de **consistance** : méthodes de réduction d'espace de recherche *a priori* (avant la détection de l'échec).

③ On supprime les valeurs qui ne peuvent pas apparaître ensemble dans la solution.

- Exemple de techniques de **consistance** :
  Forward Checking/ Looking Ahead, ...

Combiné avec **First Fail, Best Fit, Branch & Bound.....**

③ Utiliser une contrainte dès qu'elle contient une variable pour réduire l'espace de recherche.

Exemple1 : N-reines

Quand on travaille en domaine fini

- Application plus directe en CSP;
- La possibilité de raisonner directement dans le domaine du problème;
- Techniques de consistance et de propagation;
- Câblage des algorithmes d'anticipation.

Exemple2 : graphe de précédence de tâches

- Une tâche à droite s'effectue après celles à sa gauche.

- Durée de chaque tâche = 1 heure
- Début de chaque tâche ∈ {1,2,3,4,5}

- Les contraintes :

avant(T1, T2).
avant(T1, T3).
avant(T2, T6).
avant(T3, T5).
avant(T4, T5).
avant(T5, T6).
dif(T2, T3). T2 et T3 disjonctives

- Les propagations successives donneront
  T1 ∈ {1,2}, T2∈ {2,3,4}, T3∈ {2,3},
  T4 ∈ {1,2,3}, T5∈ {3,4}, T6∈ {4,5}

- On procède ensuite par énumération (les domaines sont fixes) :

Par exemple : T1 = 2 ❹ T2=4, T3=3, T4∈ {1,2,3}, T5=4, T6=5.

- Si l'on est contraint à terminer toutes les tâches en 4 heures, on aura :

T1 = 1, T3 = 2, T5 = 3, T6 = 4, T2 ∈ {3,4} et T4 ∈ {1,2}.

Exemple

Ordonnancement avec contrainte de budget en EcliPse (CHIP)

Dxx : Début tâche xx
Fxx : Fin tâche xx
Lxx : Durée tâche xx
Cxx : Coût de xx = durée de xx * 1000 F à payer d'avance

- Le coût total = somme des durées = 29 => 29000 F.
- Les taches : maçonnerie, charpente, toiture, plomberie, plafond, fenêtres, façades, jardin, pointure , emménagement

Il faut minimiser la total des durées pour emménager le plus vite

1• On propose une solution sans la contrainte de budget. On transforme ensuite celle-ci en solution finale.

```
go (L) :-
 L= [Dmassonerie ,Dcharpent,Dtoiture,Dplomberie,Dplafond,
          Dfenetres,Dfacades,Djardin,Dpeinture,Demenagement],

 L :: 0..30,

 Lmassonerie = 7, Lcharpent= 3, Ltoiture=1, Lplomberie=8,
 Lplafond=3, Lfenetres=1, Lfacades=2, Ljardin=1,
 Lpeinture=2 , Lemenagement=1,

 % les contraintes
 Fmassonerie #= Dmassonerie + Lmassonerie,
 Fcharpent #= Dcharpent + Lcharpent,
 Ftoiture #= Dtoiture +Ltoiture,
 Fplomberie #= Dplomberie +Lplomberie,
 Fplafond #= Dplafond +Lplafond,
 Ffenetres #= Dfenetres +Lfenetres,
 Ffacades #= Dfacades +Lfacades,
 Fjardin #= Djardin +Ljardin,
 Fpeinture #= Dpeinture +Lpeinture,
 Femenagement #= Demenagement +Lemenagement,

 avant(Fmassonerie,[ Dcharpent, Dplomberie,Dplafond]),
         Fcharpent #<= Dtoiture,
         avant(Ftoiture,[Dfenetres,Dfacades,Djardin]),
         avant(Fplomberie,[Dfacades,Djardin]),
         Fplafond #<= Dpeinture,
         Ffenetres#<=Demenagement,Ffacades#<=Demenagement,
         Fjardin#<=Demenagement,Fpeinture#<=Demenagement,

  min_max(labeling(L),L).  % B & B

avant(X,[]) :- !.
avant(X,[Y|L]) :- X #<= Y , avant(X,L).
```

```
%Enumération avec la stratégie "most constrained"

labeling([]).
labeling([X|Y]) :-
 deleteffc(Var,[X|Y], Reste),
 indomain(Var),              % generates values in domain
 labeling(Reste).

% Question :
go(L).

Found a solution with cost 17
L = [0, 7, 10, 7, 7, 11, 15, 15, 10, 17]
```

2• On modifie cette solution pour tenir compte du budget :

```
go (Ldebuts,L1,L2) :-
 L=[   maconnerie(LMac,CMac, DMac,FMac),
    charpente(Lcha,Ccha,Dcha,Fcha),
    toiture(Ltoi,Ctoi,Dtoi,Ftoi), plomberie(Lplo,Cplo,Dplo,Fplo),
    plafond(Lpla,Cpla,Dpla,Fpla), fenetres(Lfen,Cfen,Dfen,Ffen),
    facades(Lfac,Cfac,Dfac,Ffac), jardin(Ljar,Cjar,Djar,Fjar),
    peinture(Lpei,Cpei,Dpei,Fpei),
    emmenagement(Lemme,Cemme,Demme,Femme)],
 Ldebuts = [Dmac,Dcha,Dtoi,Dplo,Dpla,Dfen,Dfac,Djar,Dpei,Demme]Ldebuts ::
0..30,

 Lmac = 7, Lcha= 3, Ltoi=1, Lplo=8, Lpla=3, Lfen=1,
 Lfac=2, Ljar=1, Lpei=2 , Lemme=1,
 CMac = 7, Ccha= 3, Ctoi=1, Cplo=8, Cpla=3, Cfen=1,
 Cfac=2, Cjar=1, Cpei=2 , Cemme=1,

 somme([CMac,Ccha,Ctoi,Cplo,Cpla,Cfen,Cfac,Cjar,Cpei,Cemme],
                                        Cout_total),
 somme([LMac,Lcha,Ltoi,Lplo,Lpla,Lfen,Lfac,Ljar,Lpei,Lemme],
                                        Duree_total),
```

```
% les contraintes
 FMac #= DMac + LMac,
 Fcha #= Dcha + Lcha,
 Ftoi #= Dtoi +Ltoi,
 Fplo #= Dplo +Lplo,
 Fpla #= Dpla +Lpla,
 Ffen #= Dfen +Lfen,
 Ffac #= Dfac +Lfac,
 Fjar #= Djar +Ljar,
 Fpei #= Dpei +Lpei,
 Femme #= Demme +Lemme,

 avant(FMac,[Dcha,Dplo,Dpla]),
 Fcha #<= Dtoi,
 avant(Ftoi,[Dfen,Dfac,Djar]),
 avant(Fplo,[Dfac,Djar]),
 Fpla #<= Dpei,
 Ffen #<= Demme,Ffac#<=Demme,
 Fjar #<= Demme,Fpei#<=Demme,

/* Pour tenir compte des contraintes de budget
 faire 2 tranches de travaux T1 et T2 (listes L1 et L2) :
 Durée de T1 : au moins 15 jours (à maximiser),
 Coût de T1 : <=20000 à maximiser
 Durée de T2 = à minimiser, Coût de T2 >= 9000
*/
 append(L1,L2,L),   % attention a l'ordre
 Cout_L1 :: 0..20,
 somme_couts(L1, Cout_L1) ,
 Duree_L1 :: 15 .. 30,       % pourquoi, car 9000 dans 15 jours ?
 somme_durees(L1, Duree_L1) ,
 liste_debuts(L1,Ld1),
 Ld1 :: 0..20,

 somme_couts(L2,Cout_L2) , Cout_L2 + Cout_L1 #= Cout_total,
 liste_debuts(L2,Ld2),
 Ld2 :: 15..30,
```

```
min_max(labeling(L1), Ld1),
min_max(labeling(L2), Ld2).

somme([],0).
somme([X|L],Z) :- Z #= X+Y, somme(L,Y).

somme_couts([],0).
somme_couts([T|L],Z) :-
 T =.. [Nomt,Lt,Ct,Dt,Ft], Z #= Ct+Y, somme_couts(L,Y).

somme_durees([],0).
somme_durees([T|L],Z) :-
 T =.. [Nomt,Lt,Ct,Dt,Ft], Z #= Lt+Y, somme_durees(L,Y).

liste_debuts([],[]).
liste_debuts([T|L], [Dt|Z]) :-
 T =.. [Nomt,Lt,Ct,Dt,Ft], liste_debuts(L,Z).

labeling([]).            % La version simple.
labeling([T|Y]) :-
   T =.. [Nomt,Lt,Ct,Dt,Ft],        % peut etre fait avec liste_Dt
   indomain(Dt),              %  generer des valeur dans le domaine
   labeling(Y).
```

%Question :

[eclipse 71]: go(A,B,C).

Found a solution with cost 10
Found a solution with cost 20

A = [0, 7, 10, 7, 15, 15, 15, 15, 18, 20]
B = [maçonnerie(7, 7, 0, 7), charpente(3, 3, 7, 10),
toiture(1, 1, 10, 11), plomberie(8, 8, 7, 15)]
C = [plafond(3, 3, 15, 18), fenetres(1, 1, 15, 16),
facades(2, 2, 15, 17), jardin(1, 1, 15, 16),
peintre(2, 2, 18, 20),emménagement(1, 1, 20, 21)]

Classification des langages CLP :

Deux grandes classes : PrologIII (CLP(X)) et CHIP.

Prolog-III, CLP(R) :

- Instance de CLP(X) basées sur des solveurs de contraintes.
- Pas d'exploitation des domaines finis.
- Pas de génération implicite de valeur.
- Fonctionnent par retardement/réveil et propagation.
- Implantent des algorithmes GAUSS et SIMPLEX pour l'arithmétique.

- PrologIII traite également les booléens et les chaînes

CHIP / Eclipse / ...:

- Implante des techniques de consistance, Forward- Checking, ...

- Génération de valeurs pour les variables à domaine fini.

D'autres voies telles que la généralisation des contraintes sont à l'étude.

Techniques de Parcours de graphes dans les CSP

- En l'absence d'un solveur parfait, il faut mettre en place des heuristiques.

- Les heuristiques sont associées aux procédures de recherches.

- Des techniques heuristiques intéressantes en CSP :

• **Look-Ahead** : Anticiper le future pour réussir le présent
• **First-Fail** : Pour réussir, essayer d'abord où vous avez la chance d'échouer;
• **Back-Checking** : Se rappeler ce qui a été fait pour ne pas recommencer les même erreurs
• **Back-Marking :** Regarder en avant pour ne pas s'inquiéter pour le passé

- Ces techniques remplacent le back-tracking (voire, en suppriment la nécessité). Elles sont plus efficaces que le back-tracking habituel.

- La classe de problèmes CSP concernée :
• N variables
• Chaque variables a un domaine de M valeurs
Trouver toutes les assignations f de valeurs aux variables telles que toute paire <variable/valeur> notée <var,f(var)> satisfasse les contraintes.

Définitions

- Si U est l'ensemble des variables et L l'ensemble de valeurs, la relation de contrainte binaire R est définie sur $(U \times L)$ et on a :
$R \subseteq (U \times L) \times (U \times L)$.

Si une paire (<u1, l1>,<u2, l2>) $\in$ R, alors on dit que les valeurs l1 et l2 sont consistantes et compatibles.

- Une fonction d'assignation f satisfait les contraintes si pour toute paire u1,u2 de variables, (<u1,f(u1)>, <u2, f(u2)>)$\in$ R

Présentation informelle par un exemple

- L'exemple de l'échiquier n-reines (N=6 ici).
- Les variables sont les lignes et les valeurs seront les colonnes.

Par exemple, la paire (<1,A>, <2,D>) satisfait la contrainte :

(<1,A>, <2,D>)$\in$ R.

Exemple de retours en arrières :

- Pour la variable 5, les valeurs A,C,E et F se trouvent deux fois dans la trace. Elles ont été testées et ont échouées à chaque fois pour les même raisons : incompatibilité avec les variables 1 et 2.

- Ce type de redondance peut être évité si l'on se rappelle des échecs ou bien, si les variables 1 et 2 peuvent regarder en avant et prévenir 5 de prendre les valeurs A,C,E ou F : $\Rightarrow$ stratégie **Looking Ahead**

- La technique **Looking-Ahead** permet de détecter :

Lorsque la variable 3 prend la valeur E, les seules valeurs possibles pour 4 et 6 sont incompatibles.

- **Forward-Checking** =une version plus simple de Looking-Ahead.

La première fois que la variable 4 prendra la valeur B, il y a de toute évidence aucune possibilité pour la variable 6. Ainsi, la recherche pour 5 et 6 seront superflues.

Présentation des méthodes

- Notation :

variables passées : celles qui ont déjà une valeur;
variables présentes : celles en cours de traitement
variables futures : celles sans valeurs suivantes.

On se donne une table permettant de savoir quelles sont les valeurs encore possibles pour une variables donnée.

Cette table représente en fait l'état (réduit) du domaine des valeurs de chaque variable.

L'objectifs des techniques étudiées est de réduire d'avance ce domaine et éviter des tests inutiles.

Looking-Ahead complet (LAC)

- Son rôle est d'assurer que :

1- toute variable future a au moins une valeur compatible avec les valeurs déjà prises par les variables passées et présente :

(Futur × (Passé & Présent)).

2- toute variable future a au moins une valeur compatible avec une des valeurs des autres variables futures (**Futur × Futur**).

- LAC permet d'éviter un développement de l'arbre en allant en avant et revenir en arrière entre deux variables u et v, $v < u$ pour finalement découvrir que les valeurs des variables 1..v sont la cause des incompatibilités entre une variable w, $w > u$ et une variable passée, présente ou future.

Algorithme de LAC

- Soient
  Variables : 1.. Nombre_de_Var (e.g. 1..6)
  Domaine : le domaine des variables (e.g. 'A' .. 'F')
  ***F** : tableau(Variables) de Domaine*
  ***T** : tableau(Variables, Domaine) de booléen* init vrai

T[l,c] = vrai ==> la valeur représentée par la colonne <c> est consistante pour la variable représentée par la ligne <l>.

Procédure LAC(↓VAR, ↓F, ↓T) =

Début

Pour ∀ VAL ∈ T(VAR) -- les valeurs possibles de VAR

F(VAR) := VAL ;

Si VAR < Nombre_de_VAR Alors

New_T := **CHECK_FORWARD**(Var, F(VAR), T);

LOOK_FUTURE(VAR, New_T);

Si New_T ≠ VIDE Alors **LAC**(VAR+1, F, NEW_T); Finsi;

Sinon F est la solution;

Fin si;

Fin pour;

Fin LAC;

Fonction CHECK_FORWARD(↓VAR, ↓L, ↓T) : matrice =

Début

New_T := VIDE;

Pour VAR2= VAR+1 .. ..Nombre_de_Var

Pour ∀L2 ∈ T(VAR2)

Si Relation(VAR,L,VAR2,L2) Alors

Ajouter L2 dans New_T(VAR2);

Fin si;

Fin Pour;

Si New_T(VAR2) = VIDE alors Retourne(VIDE); Fin si;

-- on abandonne car pas de valeur consistante pour VAR2

Fin Pour;

Retourne(New_T);

Fin CHECK_FORWARD;

```
Procédure LOOK_FUTURE(↓VAR, ↓↑ T) =
Début
 Si VAR+1 ≥ Nombre_de_Var Alors Retourne; Fin si;
 Pour VAR1 = VAR+1 .. Nombre_de_Var
    Pour ∀L1 ∈ T(VAR1)
       Pour VAR2 = VAR+1 .. Nombre_de_Var sauf VAR1
          Pour ∀L2 ∈ T(VAR2)
             Si Relation(VAR1,L1, VAR2,L2) Alors
                Exit la boucle Pour tout L2;  -- élément
             Fin si;                          -- compatible trouvé
          Fin Pour;
          Si aucun élément compatible trouvé pour L2
          Alors                  -VAR2 n'a pas de valeur compatible
                Supprimer L1 de la liste T(VAR1);
             Exit la boucle Pour VAR2 = ...  -pas de valeur
          Fin si;                          -compatible avec <VAR1,L1>
       Fin Pour;
    Fin Pour;
    Si T(VAR1) = VIDE Alors  T := VIDE;  Retourne; Finsi;
 Fin Pour;
 Retourne;
Fin Look_Future;
```

Dans l'exemple: Var=3 ==> Var_1 = 4..6, Var_2 = 4..6 sauf Var1 en cours

(i.e. si Var_1=4, Var_2=5..6)

Suppression de B du domaine de 5 car aucune possibilité entre <5,B> et les possibilités de 4 (i.e. <4,B>). LAP n'aurait pas fait cette élimination.

- La fonction *relation(V_1,Val_1, V_2, Val_2)* teste les contraintes
- Exemple n-reines (N=6)

Looking-Ahead Partiel (LAP)

- LAC ne mémorise pas la plupart des résultats des tests (Future × Future) pour les étapes suivantes; ceci demandant beaucoup d'espace.

Les tests (Future x Future) détectent peu d'inconsistance pour justifier leur application (beaucoup de redondances dans les tests).

- **Looking-Ahead partiel** teste et mémorise uniquement (Présent × Futur) et abandonne la moitié des tests sur les (Future x Future).

③ Toute variable Future n'est testée qu'avec ses propres futures au lieu d'être testée avec toutes les autres Futures (pour elle et pour les autres).

③ Looking-Ahead partiel supprime moins de valeur de la liste des futures. Elle est plus efficace que LAC.

- Exemple : U=3, U_1 = 4..5, U_2 = U_1+1..6 ==> <4,5>, <4,6>, <5,6>

```
Procédure Look_Future_Partiel(↓U, ↓↑ T) =
Début
 Si U+1 ≥ Nombre_de_Var Alors Retourne; Fin Si;
 Pour U1 = U+1 .. Nombre_de_Var - 1
    Pour ∀L1 ∈ T(U1)
       Pour U2 = U1+1 .. Nombre_de_Var
          Pour ∀L2 ∈ T(U2)
             Si Relation(U1,L1, U2,L2) Alors-- élément
                Exit la boucle Pour tout L2;  -- compatible
             Fin si;                          -- trouvé
          Fin Pour;
          Si aucun élément compatible trouvé pour L2 Alors
             Supprimer L1 de la liste T(U1); -- U2 n'a pas
             Exit la boucle Pour U2 = ...    -- de valeur
          Fin si;                  -- compatible avec <U1,L1>
       Fin Pour;
    Fin Pour;
    Si T(U1) = VIDE Alors  T := VIDE;  Retourne; Fin si;
 Fin Pour;
 Retourne;
Fin Look_Future_Partiel;
```

Forward-Checking (FC)

- Dans Forward-Checking, on vérifie à chaque pas qu'il n'y a pas de future incompatible avec le présent.

- Les variables futures ne sont pas testées avec les futures.

- Si pour une variable donnée U, on ne trouve pas de future compatible, on passe à la valeur suivante pour U.

- Pour obtenir l'algorithme Forward-Checking, il suffit de supprimer dans LAC, l'appel à la procédure Look_Future.

- *Forward_Checking* attribue <1,A>, <2,C>, <3,E>, puis <4,B> (figures 1..4). On teste ensuite <4,B> avec ses futures et on conclut que 6 n'a plus de valeur possible (figure 5). On remet en case <3,E>, on redonne <4,B> mais cette fois, c'est 5 qui n'aura plus de valeur. Il faudra remettre en cause <2,C>.

Back_Checking (BC)

- *Back_Checking* est similaire à Forward_Checking dans la mesure où cette technique se rappelle les paires <var,valeur> inconsistantes avec le présent où les précédents.
- *Back_Checking* teste la variable présente seulement avec les variables passé et non pas avec les futures.

Exemple : si dans un problème donné, les valeurs A, B et C pour la variable 5 sont incompatibles avec la variable <2,B>, la prochaine fois que 5 doit choisir une valeur, celle ci ne sera jamais égale à A,B ou C tant que 2 a la valeur B.

- Cette technique est très proche de ce qui est fait dans les programmes classiques de Prolog : les précédents ont des valeurs et les suivant sont compatibles avec les précédent; on ne regarde jamais vers le future.
- Back_Checking seul n'est pas aussi bien que Forward_Checking. Il sera cependant utilisé dans la technique suivante.

Back_Marking (BM)

- Back_Marking est une variante de la technique Back_Checking :
- Elle élimine des tests de consistance déjà effectués qui n'ont pas réussi et qui ne réussirons pas si on les refait.
- Elle élimine des tests de consistance déjà effectués qui ont réussi et qui réussirons encore si on les refait

Exemple :

- On donne la valeur A à la variable 1; C à la variable 2 (les tests sont fait par Back_Checking).
- On donne la valeur E à la variable 3; B à la variable 4 et D à la variable 5.
- Aucune valeur possible pour 6, on remet en cause 5 puis 4 puis 3.
- On donne F à 3.
- Puisque toutes les valeurs pour 4,5 et 6 on été épuisées lors du précédent passage; si on choisi une valeur pour 4 plus petit ou égale à l'ancienne valeur de 4, on sait que celle-ci sera compatible avec la valeur de 1 et de 2.

Algorithme de BackMarking

Label : domaine des variables (e.g. 'A' .. 'H')
UNIT : 1..NB_UNITS;
F : tableau (1 .. NB_UNITS) de LABEL;
NIVEAU_INF : tableau (UNIT) de UNIT init 1;

NIVEAU_INF(i) :
le niveau inférieur auquel un changement de label a eu lieu depuis la dernière fois que la matrice MARK a été modifiée

MARK : MATRICE (LABEL) (UNIT) d'entier init 1;

MARK(u,l) indique le plus bas niveau auquel un test a échoue lorsque <u,l> du niveau actuel a été testé contre les autres paires des niveaux inférieurs.
A tout point d'exécution, si MARK(u,l) est < à NIVEAU_INF(u) alors on sait que <u,l> a déjà été testé contre les paires des niveaux inférieurs a NIVEAU_INF(u) et ces tests échoueront au niveau MARK(u,l), donc pas besoin de refaire.
Si mark(u,l) est plus grand que NIVEAU_INF(u) alors tous les tests réussiront sous le niveau (et égal) a NIVEAU_INF(u) et donc seuls les tests a partir de NIVEAU_INF(u) jusqu'à niveau actuel devront être refaits.

```
Procédure BACK_MARK(↓ U, ↓↑ F, ↓↑ MARK, ↓↑ NIVEAU_INF) =
Début
Pour LAB dans LABEL
   F (U) := LAB;
      Si MARK (U) (F (U)) >= NIVEAU_INF (U) Alors
               TESTFLAG := TRUE;
      L := NIVEAU_INF (U);
      TantQue (L < U)          -- trouver le plus bas échec
         TESTFLAG := RELATION (L, F (L), U, F (U));
         Exit si non TESTFLAG;
         L := L + 1;
      Fin TQ;
      MARK (U) (F (U)) := L;      -- marquer avec le plus bas
                                  -- niveau  d'échec
      Si TESTFLAG Alors
         Si U < NB_UNITS Alors
            BACK_MARK (U + 1, F, MARK, NIVEAU_INF);
         Sinon
            F est une solution
         Fin Si;
      Fin Si;
         Fin Si;
Fin Pour;
Si U = 1 Alors retourne;
Sinon
   NIVEAU_INF (U) := U - 1;
   Pour I in U + 1 .. NB_UNITS
      NIVEAU_INF (I) := MIN (NIVEAU_INF (I), U - 1);
   Fin Pour;
Fin Si;
Fin BACK_MARK;
```

Performances

- Dans l'ordre , Forward_Checking et Back-marking sont les meilleurs (moins de 250,000 tests pour n-reines, N=10).
  • Looking-Ahead partiel : 600,000
  • Back-Checking : 750,000
  • Back-Tracking : 1,100,000.

- Back-Tracking est très proche de Back-Checking. Ce qui montre l'efficacité de Prolog par rapport aux langages de programmation avec contraintes.

- Back-Marking a tendance a trop consulter ses tables de mémorisation par rapport à la taille du problème (la valeur de N). Les meilleurs résultats sont obtenus par Forward_Checking (400,000, pour N=10 et 600,000 pour Back-Marking)

- Le meilleur rapport qualité performances est Forward-Checking. Forward-Checking a été implanté dans CHIP.

- Looking-Ahead et Forward-Checking élaguent rapidement des arbres de recherche.

Principe de First-Fail

- Une des stratégies employées dans les CSP.

- D'autres méthodes sur le même principe :

- Essayer d'échouer le plutôt possible en mémorisant la situation d'échec pour ne pas avoir à la recommencer.

Applications de cette stratégie :

1. Optimiser l'ordre dans lequel les tests de consistances sont faits. Parmi un ensemble de test de consistance, le principe First-Fail encourage à effectuer d'abord ceux davantage susceptibles d'échouer (ont plus de chance d'échec) et ainsi éviter d'effectuer les autres tests de l'ensemble.

2• Choix dynamique d'un ordre optimum de traitement (affectation, génération) des variables. Un choix même local permet de réduire le nombre de tests de consistance.

Un tel choix pourrait être celui de sélectionner la variable la plus contrainte (celle qui a le minimum de choix de valeur possible).

Ordre des test

- Pour mettre en place une politique de test, il faut avoir une idée du degré des contraintes imposée par la valeur de la variable K à la variable K+1.

- Pour le cas de N-reines, on peut effectuer les tests de consistance entre la valeur de la variable Ki et les valeurs des variables Ki-1..K1 dans un ordre décroissant de contraintes (puisque Ki-1 est la variable qui contraint le plus la variable Ki). Une expérimentation avec le Back-Tracking et N=10 montre une différence de 200,000 tests de consistance.

Choix des variables

- Optimisation d'ordre de recherche (du niveau de l'arbre de recherche).
- On choisira d'affecter la variable la plus contrainte d'abord.
- Plus on avance en profondeur dans l'arbre (pour une branche donnée), moins les variables ont un choix de valeur. Ainsi, en choisissant le noeud (la variable) qui a un minimum de chance de se trouver une valeur (car son domaine est de plus en plus réduit), on augment la probabilité d'avoir un échec dans les test de consistance et ainsi, on réduit le niveau de l'arbre de recherche.
- Ce choix est local (au niveau d'une branche).
- Pour N=10, un choix normal pour le Forward_Checking effectue 242174 tests alors qu'avec l'optimum local, on a 205,970 test.

Forward-Checking en PrologIII

- PrologIII ne propose pas d'outil intégré et il faut mettre en ouvre la technique selon le problème traité.

Exemple : n-reines

1- la version contraindre-générer

```
queens(n,l) -> queens'(l) enumere(n,l);  % contraindre/générer
enumere(n,<>) -> ;
enumere(n,<x>.l) -> enum(x,n) enumere(n,l);
queens'(<>) ->;
queens'(<X>.Y) ->
 safe(X,Y,1)
 queens'(Y) ;
```

```
safe(_,<>,_) -> ;
safe(X, <F>.T, N) ->
 safe(X,T,N+1)
{F#X, X #F-N, X#F+N};
```

2- Simulation de Forward-Checking

On installe le domaine puis on énumère une valeur. On élimine ensuite des valeurs des domaines des autres variables par NOATTACK.

```
nreines(n,<>) ->;
nreines(n,l) -> nreines'(n,l) fc(l);

fc(<>) ->;
fc(<x>.l) -> enum(x) noattack(x,l) fc(l);

nreines'(n,<>) ->;
nreines'(n,<x>.l) -> nreines'(n,l) {n>=x>=1};

noattack(x,l) -> safe(x,l,1);
```

Résultats des tests avec n=10

- Contraindre/générer sur HP :
  première réponses 780 ms et
  dernière réponses 282110

- Forward-Checking sur HP :
  première réponses 460 ms et
  dernière réponses 174940

- L'apport de Forward-Checking est évident.
- Pour implanter le principe First Fail, une étude plus détaillée permettant de désigner les variables les plus contraintes est nécessaire.

Techniques de résolution

- Les techniques vues précédemment permettent de réduire l'espace de recherche a priori.

- Elles sont bien adaptées aux CSP car les CSP sont avant tout des problème de recherche.

- Les CSP sont Np-complets avec un grand nombre de choix pour obtenir une solution et le but de ces techniques est de réduire le choix.

- Les Algorithmes de résolution des CSP procèdent en deux étapes :
  - raisonner à propos des contraintes
  - faire un choix (de valeur pour les variables)

- Raisonner à propos des contraintes n'est pas résoudre le problème. C'est pourquoi on est parfois amené à faire des choix ou des hypothèses sur les valeurs des variables.

- Ce choix peut considérablement influencer le comportement de la solution. Il faut donc contrôler le choix de valeurs des variables.

- Une technique simple est le principe Firs-Fail. Ce principe consiste à choisir d'abord la variable la plus contrainte pour provoquer le plus rapidement des échecs.

La variable dont le domaine est le plus réduit est la meilleure candidate.

- Inversement, pousser au plus tard le choix de la variable la plus contrainte risque de provoquer un échec tardif.

- Pour deux variables dont les domaines sont également réduits, on choisira celle participant aux plus de contraintes. Celle-ci devant satisfaire plus de contraintes que les autres. De plus, on espère qu'il sera plus difficile de satisfaire ses contraintes et donc de lui trouver une valeur.

- PrologIII propose quelques primitives :
  frozen_goals
  delayed_constraints
  les métatermes (par outc).

- Contraintes comme choix :

P :- C1 , A1.
P :- C2 , A2.
.....
P :- Cn , An.

Ci des contraintes, Ai des conjonctions d'atomes.

- Chaque satisfaction de P est conditionnée par la satisfaction des contraintes Ci.
- En programmation logique classique, les contraintes seront des tests à évaluer.
- En CLP, les contraintes vont dirigent les choix.

- **Exemple** (PrologIII):
  pp(X, Y, Z) :- {X < Y + Z};
  pp(X, Y, Z) :- {X > Y - Z};

En programmation logique, X, Y et Z doivent avoir des valeurs pour pouvoir faire les tests. En CLP, il s'agit **d'ajouter** une contrainte au système actuel de contraintes et vérifier la cohérence du système résultant.

Ces contraintes peuvent réduire les domaines des valeurs de telle sorte que l'on puisse déduire d'autres information du système actuel.

Si plus tard, le système devient inconsistant, on retourne en arrière pour reprendre une autre possibilité.

- **En CLP, le calcul est guidé par les contraintes.**

- Une autre technique est de découper le domaine (large) d'une variable. Ainsi, on pourra rejeter une partie des valeurs au lieu d'en rejeter qu'une valeur à la fois.

Exemple : (Mid = (min(X) + max(X)) /2)

```
split (X, _Mid) -> { X <= _Mid};
split (X, _Mid) -> { X < _Mid};
```

Le choix de l'un ou l'autre permet de rejeter 1/2 des valeurs de X. Notons que dans ce cas (cf. CSP), les domaines doivent être connus.

Techniques d'optimisation

- Cas typique :

- Un ensemble de variables
- Le domaine entier (discret)
- Un ensemble de contraintes sur les variables
- Une fonction objective à optimiser (coût à minimiser, gain à maximiser....)

- Le problème dans le domaine des entiers est plus difficile que dans les autres domaines.
  - Simplex est de complexité exponentielle mais la programmation linéaire est polynomiale.
  - Il n'y a pas d'algorithme Simplex sur (seulement) des entiers.

Ce type de problèmes est Np-Complet.

On considère ici des problème non triviaux (plus de 100 variables) pour lesquels les techniques d'optimisation sont plus "payantes".

Branch & Bound

ZZ : en AR-07-8 (BE4 et 5), j'ai dit que B&B se réalise en Gprolog par fd_minimize Non appliqué à fd_labeling mais plutôt un fd_minimize sur un prédicat qui lui contient son fd_labeling.
L'ex typique a été la découpe (voir mon fichier .pl de 06-07.).

Tirer ca au claire avec une trace.
D'ailleurs, pourquoi fd_minimize ne marche pas sur fd_labeling lorsqu'on a l'op #>= dans Gprol comme on peut le voir dans decoupe.pl).

- Pour les problème CSP sur les entiers, une recherche exhaustive est impraticable.

- Branch & Bound est une technique intelligente d'instanciation.

- Branch & Bound consiste en

- une phase de séparation du problème en sous problèmes
- une phase d'évaluation de borne (coût)

- Dès qu'une solution est trouvée (de n'importe quelle façon), toutes les solutions ayant un coût supérieur (ou inférieur selon le cas) au coût de cette solution n'auront plus à être considérés pour les phases de séparation et d'évaluation. Ce qui réduit largement l'espace.

- Pour éviter des retours en arrières (produits dès la première solution), une alternative à Branch & Bound est de **recommencer dès le début** à chaque fois qu'une solution est trouvée. Le travail refait est largement compensé par la suppression des points de retour arrière. Cette technique est plus intéressante que le Branch & Bound classique si dans l'arbre de recherche, certaines branches ont un coût nul (traitement n'entraînant pas de modification du coût à minimiser) ou bien lorsque le coût trouvé doit être modulé.

Aussi, si l'environnement est modifié pendant la recherche d'une solution (par exemple avec des asserts ou avec des suppressions), on est obligé de recommencer les calculs depuis le début.

- Contrôle possible dans les points de choix ou au niveau supérieur.
- Maximize et Minimize de PrologIII sont des Branch & Bound "locaux".

Exemples en PIII

1 : Contrôle chaque alternative d'un choix donné. On conserve la meilleure alternative ainsi que son coût, qu'on répercute à la prochaine alternative, afin de la faire échouer si elle est moins bonne. Ceci est correct si on a qu'une seule disjonction, comme dans le cas suivant.

Appel: go(X, C);
Réponse :
"meilleure solution, une seule disjonction"
{X = seconde_reponse, C = 1}

```
go(_,_) ->
 assign(rep, rep)              % rien trouve au départ.
 assign(cout, 1000000)         % valeur 'bidon' de départ.
 disjonction
 fail;

go(_rep, _cout) ->
 val(rep, _rep)
 val(cout, _cout)
 {_rep # rep};       % on échoue si aucune réponse possible                %
(meilleure que 1000000)
            %on veut pas de réponse rep (bidon)
%
% disjonctions
%
disjonction ->
 val(cout, M)
 eq(C,2)
          % franchie, donc meilleure,
          % franchi car eq affecte et C<M teste
 assign(cout, C)
 assign(rep, premiere_reponse)%ordre important
 { C < M};
```

```
disjonction ->
 val(cout, M)
 eq(C,1)
          % franchie, donc meilleure
 assign(cout, C)
 assign(rep, seconde_reponse)
 { C < M};

disjonction ->
 val(cout, M)
 eq(C,3)
             % franchie, donc meilleure
 assign(cout, C)
 assign(rep, troisieme_reponse)
 { C < M};
```

2 : On suppose que l'on cherche la plus petite somme parmi les couples. On suppose que chacun des membres doit être positif par exemple : {1,4,2} x {3,1,2}

Le coût n'est affecté qu'une fois les deux disjonctions passées. Ici, on NE contrôle PLUS chaque alternative d'un choix donné. On se contente de contrôler chaque choix (on vérifie qu'après, il n'est pas impossible de faire mieux que la solution précédente).

Appel: go2(X, C);
Réponse :
{X = [un,deux], C = 2}
"meilleure solution, plusieurs disjonctions"

```
go2(_,_) ->
 assign(repx, repx)          % rien trouve au départ.
 assign(cout, 1000000)       % valeur 'bidon' de départ.
 val(cout, C0)               % on installe la contrainte  disjonction1(x,
_nx)    % "meilleure  qu'avant"
 val(cout, C1)               % on installe la contrainte  disjonction2(y,
_ny)    "meilleure  qu'avant"
 val(cout, C2)               % on installe la contrainte
                    % "meilleure  qu'avant"
% on arrive ici: c'est donc meilleur qu'avant (on conserve les  infos)
 assign(cout, C)
 assign(repx, _nx)
 assign(repy, _ny)
 fail
 {x >= 0, y >= 0, C = x+y, C0 >= C, C1 >= C, C2 >= C};

go2([_repx, _repy], _cout) ->
 val(repx, _repx)
 val(repy, _repy)
 val(cout, _cout)
 {_repx # repx};   % on échoue si aucune réponse possible
              %(meilleure que  1000000)
```

```
%
% Les disjonctions
%
disjonction1(1, un) -> ;
disjonction1(4, deux) -> ;
disjonction1(2, trois) -> ;

disjonction2(3, un) -> ;
disjonction2(1, deux) -> ;
disjonction2(2, trois) -> ;
```

Programmation de la variante de Branch & Bound en PrologIII

3- Coloration de 5 régions : Branch & Bound dans la version où on trouve un coût et on recommence tout (meilleurs résultats)

```
col(n) ->
 assign(cout,15)  % coût maximum
 col'(n)          % pas besoin des Xi
 /                % couper et recommencer
 col1(_,n);

col(n) -> outml("pas de solution");

col1(_,n) ->
 col'(n) /
 col1(_,n);

col1(<x1,x2,x3,x4,x5>, n) ->
 val(cout,R)
    diff(x1, <x2,x3,x4>) diff(x2, <x3,x5>)
    diff(x3, <x4,x5>)          diff(x4, <x5>)
    enum(x1) enum(x2) enum(x3) enum(x4) enum(x5)
 outml("la meilleure  solution est ")
 outl(<x1,x2,x3,x4,x5>)
    {0<x1<n,0<x2<n, n>x3>0, n>x4>0, 0<x5<n, x1+x2+x3+x4+x5 =R};

col'(n) ->
 val(cout,C)
 diff(x1, <x2,x3,x4>)    diff(x2, <x3,x5>)
 diff(x3, <x4,x5>)       diff(x4, <x5>)
 enum(x1) enum(x2) enum(x3) enum(x4) enum(x5)
 minimize(R)
 assign(cout, R)
 {0<x1<n,0<x2<n, n>x3>0, n>x4>0, 0<x5<n,
    x1+x2+x3+x4+x5 = R,  5< R < C};

diff(_,<>) ->;
diff(x,<y>.l) -> diff(x,l) {x # y};
```

```
col(5);
la meilleure  solution est
<1,2,3,2,1>
la meilleure  solution est
<2,1,3,1,2>
```

4- Exemple d'entrepôts avec Branch & Bound

Enoncé :

Une compagnie a des entrepôts et des clients.

-Il y a 3 entrepôts (E1, E2, E3)

-Il y a 5 clients (C1..C5)

-Selon la situation géographique, on sait d'avance que :

. C1 peut être fourni par E1, E2

. C2 peut être fourni par E1, E3

. C3 peut être fourni par E2, E3

. C4 peut être fourni par E3

. C5 peut être fourni par E1, E3

- Chaque entrepôt a un coût fixe de maintenance.

On a respectivement les coûts 18, 10 et 20 pour E1, E2 et E3

- Il y a des frais de transport (variables fonction de distance et autres) pour les livraisons. On prend les variable Cij (l'entrepôt i vers le client j)

- Le tableau des frais de transport est :

E1 fournit C1,C2 et C5 avec les coûts respectifs 5,4 et 3

E2 fournit C1,C3 avec les coûts respectifs 7 et 2

E3 fournit C2,C3,C4 et C5 avec les coûts respectifs 1,5,4 et 8

- Un client n'est servi que par un seul entrepôt

On veut savoir quel entrepôt fournit quel client avec un coût minimal (pas besoin de connaître le seuil 60 car on est en B&B)

Différentes solutions

Solution 1 :

Le programme ci-dessous s'intéresse uniquement au meilleur COUT.

Pour récupérer les autre paramètres, étant donne que assign n'accepte que des données simples, on peut :

- se servir d'assert et de suppress (i.e., faire notre propre assign)
- transférer les listes vers les tableaux (on peut manipuler des tableaux en PIII)
- une fois le meilleur coût trouvé, relancer une exécution avec ce coût et récupérer les autres paramètres. L'inconvénient serait qu'il y ait plusieurs solutions avec ce coût minimum (est-ce important?)

```
app(_Ts, _Ls, _Cout) ->
 assign(cout,100)
 approvisionnement(_Ts, _Ls, _Cout)
 outm("le cout de la solution ")  out(_Ts) outm(" est : ") outl(_Cout)
 val(cout,C1)
 outm("et le meilleur cout actuel qui va eter remplace par ")
 out(_Cout) outm(" est ")  outl(C1)
    assign(cout,_Cout)
 fail
 {_Cout < C1};

app(_Ts, _Ls, _Cout) ->
 val(cout,_Cout)
 outm(" le meilleur cout = ")  outl(_Cout);

approvisionnement(_Ts, _Ls, _Cout) ->
    configuration(_Ts, _Ls, _Cout) ;

configuration([C11, C12, C21, C23, C32, C33, C43, C51, C53],
        [E1, E2, E3], _Cout) ->

% **** dans cette solution, générer est obligatoire ****

    genere([C11, C12, C21, C23, C32, C33, C43, C51,C53,
              E1, E2, E3]) ,
       {C11 + C12 = 1,  % l'un ou l'autre des entrepôts pour le client1
       C21 + C23 = 1,
       C32 + C33 = 1,
       C43 = 1,             % client4 est fourni par E3
       C51 + C53 = 1,
       C11 + C21 + C51 <= 3 * E1,
       % éviter un client d'être assigné a un entrepôt 0
       C12 + C32 <= 2 * E2,
       C23 + C33 + C43 + C53 <= 4 * E3,
       _Cout = 5 * C11 + 7 * C12 + 4 * C21 + C23 + 2 * C32 + 5 * C33 +
    4 * C43 +3 * C51 + 8 * C53 + 18 * E1 + 10 * E2 + 28 * E3} ;
```

```
genere([]) -> ;      % simplifié pour cet exemple car on sait 0,1
genere([1 | _Xs]) ->    genere(_Xs) ;
genere([0 | _Xs]) ->    genere(_Xs);
```

Exemple d'exécution :

```
cpu_time(X)  app(a,b,v) cpu_time(Y)  {Z=Y-X};
le cout de la solution [1,0,1,0,1,0,1,1,0] est : 74
et le meilleur cout actuel qui va être remplace par 74 est 100
le cout de la solution [1,0,1,0,1,0,1,0,1] est : 79
le cout de la solution [1,0,1,0,0,1,1,1,0] est : 77
le cout de la solution [1,0,1,0,0,1,1,1,0] est : 67
et le meilleur cout actuel qui va être remplace par 67 est 74
le cout de la solution [1,0,1,0,0,1,1,0,1] est : 82
le cout de la solution [1,0,1,0,0,1,1,0,1] est : 72
le cout de la solution [1,0,0,1,1,0,1,1,0] est : 71
le cout de la solution [1,0,0,1,1,0,1,0,1] est : 76
le cout de la solution [1,0,0,1,0,1,1,1,0] est : 74
le cout de la solution [1,0,0,1,0,1,1,1,0] est : 64
et le meilleur cout actuel qui va être remplace par 64 est 67
le cout de la solution [1,0,0,1,0,1,1,0,1] est : 79
le cout de la solution [1,0,0,1,0,1,1,0,1] est : 69
le cout de la solution [0,1,1,0,1,0,1,1,0] est : 76
le cout de la solution [0,1,1,0,1,0,1,0,1] est : 81
le cout de la solution [0,1,1,0,0,1,1,1,0] est : 79
le cout de la solution [0,1,1,0,0,1,1,0,1] est : 84
le cout de la solution [0,1,0,1,1,0,1,1,0] est : 73
le cout de la solution [0,1,0,1,1,0,1,0,1] est : 78
le cout de la solution [0,1,0,1,1,0,1,0,1] est : 60
et le meilleur cout actuel qui va être remplace par 60 est 64
le cout de la solution [0,1,0,1,0,1,1,1,0] est : 76
le cout de la solution [0,1,0,1,0,1,1,0,1] est : 81
le cout de la solution [0,1,0,1,0,1,1,0,1] est : 63
 le meilleur cout = 60
{v = 60}
TEMPS = 1290 ms. environ
```

Solution 2 : gain important de performances

La solution précédente va jusqu'au bout de chaque solution.

Dans "configuration1", on met "val(cout,X)" puis "_Cout < X" dans les contraintes pour élaguer le plus possible.

NB : la place de "val(cout,X)" après "genere" est très importante car le FAIL provoqué plus haut impose un retour arrière sur "genere". Si val(cout,X) est mis avant "genere", ca va faire du n'importe quoi et toute nouvelle solution écrase la précédente car **X EST RESTE FIGE** à sa toute première valeur initiale, c'est a dire : 100.

Pour constater cet effet néfaste, il suffit de permuter "val" et "genere1" dans configuration1 et mettre une trace après val pour voir que l'on ne passe qu'une seule fois dans ce VAL et X reste fixé à 100 et les générations successives de "genere1" donne des valeurs.

Il faut remarquer que le "fail" provoqué en haut ne NOUS FAIT PAS REPASSER dans "approvisionnement1" mais retourne en arrière sur la dernière disjonction (i.e. genere).

```
app1(_Ts, _Ls, _Cout) ->
 assign(cout,100)
 approvisionnement1(_Ts, _Ls, _Cout)
 outm("le cout de la solution ")  out(_Ts) outm(" est : ") outl(_Cout)
 val(cout,C1)
 outm("et le meilleur cout actuel qui va être remplace par ")
 out(_Cout) outm(" est ")  outl(C1)
    assign(cout,_Cout)
 fail;              % FAIL ICI


app1(_Ts, _Ls, _Cout) ->
 val(cout,_Cout)
 outm(" le meilleur cout = ")  outl(_Cout);

approvisionnement1(_Ts, _Ls, _Cout) ->
   configuration1(_Ts, _Ls, _Cout) ;
```

```
configuration1([C11, C12, C21, C23, C32, C33, C43, C51, C53],
            [E1, E2, E3], _Cout) ->
   genere([C11, C12, C21, C23, C32, C33, C43, C51,C53,
               E1, E2, E3])
   val(cout,X)

      {C11 + C12 = 1,  % l'un ou l'autre des entrepôts pour le client1
      C21 + C23 = 1,
      C32 + C33 = 1,
      C43 = 1,            % client4 est fourni par E3
    C51 + C53 = 1,
 C11 + C21 + C51 <= 3 * E1,
         % eviter un client d'être assigné a un entrepôt 0
 C12 + C32 <= 2 * E2,
 C23 + C33 + C43 + C53 <= 4 * E3,
 Cout = 5 * C11 + 7 * C12 + 4 * C21 + C23 + 2 * C32 + 5 * C33 +
   4 * C43 +3 * C51 + 8 * C53 + 18 * E1 + 10 * E2 + 28 * E3 ,
   _Cout < X } ;
```

Exemple (On ne peut faire mieux pour élaguer)

```
cpu_time(X)  app1(a,b,v) cpu_time(Y)  {Z=Y-X};
le cout de la solution [0,1,0,1,0,1,1,0,1] est : 63
et le meilleur cout actuel qui va être remplace par 63 est 100
le cout de la solution [0,1,0,1,1,0,1,0,1] est : 60
et le meilleur cout actuel qui va être remplace par 60 est 63
 le meilleur cout = 60
{X = 14628866, v = 60, Y = 14629483, Z = 617}
```

Solution 3 :

"val" et "genere" permutés. On est sauvé par {_Cout < C1} dans app3.

X dans "configuration3" reste figé à 100 donc conceptuelle ment c'est une mauvais solution et sans trop de gain mais on élague un peu mieux.

```
app3(_Ts, _Ls, _Cout) ->
 assign(cout,100)
 approvisionnement3(_Ts, _Ls, _Cout)
 outm("le cout de la solution ")  out(_Ts) outm(" est : ") outl(_Cout)
 val(cout,C1)
 outm("et le meilleur cout actuel qui va être remplace par ")
 out(_Cout) outm(" est ")  outl(C1)
    assign(cout,_Cout)
 fail                % FAIL ICI
 {_Cout < C1};


app3(_Ts, _Ls, _Cout) ->
 val(cout,_Cout)
 outm(" le meilleur cout = ")  outl(_Cout);


approvisionnement3(_Ts, _Ls, _Cout) ->
   configuration3(_Ts, _Ls, _Cout) ;

configuration3([C11, C12, C21, C23, C32, C33, C43, C51, C53],
       [E1, E2, E3], _Cout) ->
   val(cout,X)
   genere([C11, C12, C21, C23, C32, C33, C43, C51,C53,
                          E1, E2, E3])

       {C11 + C12 = 1,  % l'un ou l'autre des entrepôts pour le client1
       C21 + C23 = 1,
       C32 + C33 = 1,
       C43 = 1,            % client4 est fourni par E3
       C51 + C53 = 1,
       C11 + C21 + C51 <= 3 * E1,
       % eviter un client d'être assigné a un entrepôt 0
```

```
    C12 + C32 <= 2 * E2,
    C23 + C33 + C43 + C53 <= 4 * E3,
    _Cout = 5 * C11 + 7 * C12 + 4 * C21 + C23 + 2 * C32 + 5 * C33 +
  4 * C43 +3 * C51 + 8 * C53 + 18 * E1 + 10 * E2 + 28 * E3 ,
   _Cout < X } ;
```

Exemple

```
cpu_time(X)  app3(a,b,v) cpu_time(Y)  {Z=Y-X};
le cout de la solution [0,1,0,1,0,1,1,0,1] est : 63
et le meilleur cout actuel qui va être remplace par 63 est 100
le cout de la solution [0,1,0,1,0,1,1,0,1] est : 81
le cout de la solution [0,1,0,1,0,1,1,1,0] est : 76
le cout de la solution [0,1,0,1,1,0,1,0,1] est : 60
et le meilleur cout actuel qui va être remplace par 60 est 63
le cout de la solution [0,1,0,1,1,0,1,0,1] est : 78
le cout de la solution [0,1,0,1,1,0,1,1,0] est : 73
le cout de la solution [0,1,1,0,0,1,1,0,1] est : 84
le cout de la solution [0,1,1,0,0,1,1,1,0] est : 79
le cout de la solution [0,1,1,0,1,0,1,0,1] est : 81
le cout de la solution [0,1,1,0,1,0,1,1,0] est : 76
le cout de la solution [1,0,0,1,0,1,1,0,1] est : 69
le cout de la solution [1,0,0,1,0,1,1,0,1] est : 79
le cout de la solution [1,0,0,1,0,1,1,1,0] est : 64
le cout de la solution [1,0,0,1,0,1,1,1,0] est : 74
le cout de la solution [1,0,0,1,1,0,1,0,1] est : 76
le cout de la solution [1,0,0,1,1,0,1,1,0] est : 71
le cout de la solution [1,0,1,0,0,1,1,0,1] est : 72
le cout de la solution [1,0,1,0,0,1,1,0,1] est : 82
le cout de la solution [1,0,1,0,0,1,1,1,0] est : 67
le cout de la solution [1,0,1,0,0,1,1,1,0] est : 77
le cout de la solution [1,0,1,0,1,0,1,0,1] est : 79
le cout de la solution [1,0,1,0,1,0,1,1,0] est : 74
 le meilleur cout = 60
{X = 14725833, v = 60, Y = 14727550, Z = 1717}
```

Compléments sur les techniques de résolution

- Méthodes de programmation sous contraintes.

- La complexité du programme et l'efficacité de la solution dépendent grandement du respect des points ci-dessus.

- **définir d'abord les variables et leur domaines.** La construction d'une liste de variable de taille importante peut se faire par des prédicats.

- **Imposer des contraintes aux variables.** Ces contraintes peuvent être élémentaires ou être installées par des prédicats. Par exemples des prédicats récursives (cf. **tous_dif**) ou des prédicats qui simplement installent une collection de contraintes.

- Si l'expression seule des contraintes ne peut pas résoudre le problème (l'exemple PERT suffisait mais pas tous les autres) alors il faut **assigner des valeurs aux variables** (solveurs partiels).

Chaque instanciation réveille immédiatement des contraintes associées aux variables et les modifications sont propagées.
==> l'espace de recherche est réduit rapidement et un premier échec peut éventuellement être détecté (e.g. domaine de variable réduit à vide). Sinon, les autres variables verront leur domaine réduit (par fois) à une seule valeur ==> le système pourra affecter cette valeur unique.

Cette politique d'affectation (appelée **labeling**) est d'une importance majeure et sera l'élément principal du schéma tester/générer.

- Il est souvent possible d'exprimer la même solution avec différents ensembles de variables. Il vaudra mieux conserver un espace de recherche aussi réduit que possible. Par exemple, l'on préférera 5 variables dans 1..10 plutôt que 15 variables dans 0..1.

==> Les solutions booléennes aux problèmes d'entiers sont rarement préférables.
Dans l'exemple de pigeons (n pigeons à placer dans m trous), on peut prendre n variable dont les domaines sont 1..m ou bien prendre n

x m booléens. Dans l'exemple de n-reines, on peut prendre n x n booléens ou n entier dans 1..n.

- Le choix des contraintes est très important.

Parfois, une contrainte redondante (i.e. l'un déductibles des autres) peut accélérer considérablement les performances.

La raison en est que le système ne propage pas toute les informations qu'il possède concernant les variables car la plupart du temps, ceci n'apporte rien et réduira la recherche.

Les contraintes doivent être spécifiées de telle sorte que le système puisse propager celles qui sont importantes (essentiellement concernant les mise à jours des domaines des variables).

==> Donner un maximum de contraintes éventuellement redondantes

- Une autre règle d'or est d'éviter les points de choix et de les repousser au plus tard possible. Les contraintes disjonctives doivent être traitées le plus tard possible.

- Le choix de labeling est l'un des plus sensibles. On remarque souvent qu'une modification mineure de l'ordre d'instanciation modifie grandement la vitesse. (cf. coloration de graphes).

Si l'espace est grand, il serait intéressant de passer du temps à sélectionner la prochaine variable à instancier.

PrologIII permet d'accéder partiellement aux contraintes par *delayed_constraints/2*.

Il est parfois nécessaire d'essayer différentes méthodes et observer le comportement à l'exécution.