École Centrale de Lyon
Cours optionnel 3° Année

Concepts et Paradigmes de Programmation

S. Saidi
M. I . S.

1993-1994

CHAPITRE II

Paradigme Contraintes

Ecole Centrale de Lyon
Département Mathématiques,
Informatiques et Systèmes
1993-94
S. SAIDI

Table des matières

Il faut reprendre ce fichier et le mettre en accord avec les transparents (trans-chap-2-fait).
Ca l'est pratiquement jqa la fin de CLP(R).
Ici, il y a rien sur PIII.
Il faut mettre une présentation générale de PIII avec un accent sur les contraintes avec des exemples dans chaque partie (sans tomber dans la partie syntaxique).

Pour le cours sur techniques de parcours de graphes, voir les transparents et recouper avec ce texte.

Introduction

Affectation vs. Equation

- En programmation impérative, le programmeur construit un programme à partir d'un algorithme. Le prgramme est une procédure qui exécute les ordres étape par étap.

Le programmeur doit anticiper les différents problèmes et inclure des points de décision dans son algorithme.

Considérons le problème de conversion de température :

C = (F-32) * 5/9

On doit prévoir par des conditionnels "si" ou "case" pour décider du traitement exact en fonction d'état des variables

F = 32 + 9/5 * C

De même pour Z = X + Y, il faut prévoir tous les cas :
- si X et Y sont connues alors Z := X+Y
- si Z est X sont connues alors Y = Z-X ...

- En programmation avec contraintes, programmer devient *déclarer* où le programmeur spécifie des relations entre les objets et c'est le système qui trouve une solution telle que les relations se vérifient.

- Pour l'exemple, C= (F-32) * 5/9 devient un *programme* qui définit la relation entre F et C.
- Dans ce cadre, le problème de conversion pourrait être spécifié de différentes façons équivalentes. On peut par exemple spécifier simplement les relations entre F et C par :
  - la relation est linéaire : F = M *C + B
  - 212 degrés Fahrenheit est le même que 100 degré Celsius (ébullition)
    212 = M * 100 + B
  - 32 degrés Fahrenheit est le même que 0 degré Celsius (gèle)
    32 = M * 0 + B

Le système nous trouve F = (9/5)C + 32

- De même pour Z = X + Y
- On constate que la modification de tel programme devient facile et on n'est plus obligé de modifier tous les cas que l'on devait énumérer.

- En programmation avec contraintes, X = Y est une relation permanente (pendant l'exécution du programme) entre deux objets alors qu'en paradigme impratif, le seul instant où X=Y est juste après l'exécution de X:=Y.

- Le X := X+1 de l'impératif devient X = X+1 ==> (faux)

Petit exemple d'introduction de système de contraintes:

On a 25 pièces de monnaie de 5, 10 et 25 cents. la valeur du total est 3$45 et il y a 7 pièces de 10 cents de plus que de pièces de 5 cents. Trouver le nombre de chaque pièce.

Le programmation de la la ersion impérative est laissé aux élèves. La version cntraintes est :

{N+D+Q=25, 5N+10D+25Q=345, D=7+N};
==> {N = 5, D = 12, Q = 8}

- On constate que la programmation sous contraintes permet un aspect déclarative par une définition relationnelle et rend la modification et la maintenace des programmes simple et aisé.

- Les langages de programmation contraintes étendent la notion de contrainte pour traiter les contraintes d'inégalité (diséquation $\neq$) et les inéquations ($<, \leq, >, \geq$).

- Etant donné l'aspect relationnel de Prolog, les langages de programmation avec contraintes ont été implanté comme une extension de Prolog.

- Une application particulière de la programmation sous contraintes est le résolution des problème de satisfaction de contraintes appelé CSP (constraint satisfaction problems) introduit par l'exemple ci-dessus.

Problèmes de satisfaction de contraintes (CSP)

Définition

- D'une manière formelle, on définit le CSP par :

-un ensemble fini I de variable $\{X_1,...,X_n\}$ qui prennenet leur valeurs dans les domaines $D_1...D_n$ et

-un ensemble C de contraintes.

- Une contraintes de la forme $\mathbf{c(X_{i1},...,X_{ik})}$ entre k var de **I** est un sous-ensemble du prod cartésien $D_{i1} \times ... \times D_{ik}$ spécifiant les valeurs des vars mutuellement compatibles.
- Une sol° à un CSP est une assignation (afectation) de val aux vars qui satisfont toutes les contraintes. Le but est de trouver une ou toutes les sol°.

- Par exemple, pour le problème des 6-reines où il faut placer 6 pions dans les cases d'un échiquier 6x6, on a
  I = {1 .. 6} et D={A..F}.
  Les contraintes à satisfaire pour aboutir à une solution sont :
  • deux pions ne doivent pas être sur la même ligne. Remarquons qu'en choisissant les variables distinctes dans I, cette contrainte est déjà satisfaite.
  • deux pions ne doivent pas être sur la même colonne.
  • deux pions ne doivent pas être sur la même diagonale.

Caractéristiques des systèmes CSP

- Un système de résolution de CSP est composé de deux parties :
  - Un ensemble de règles
  - Un mécanisme de contrôle

Les règles spécifient le problème et le mécanisme de contrôle conduit l'application des règles.

- La séparation nette des deux parties est souhaitable pour rendre la construction des solutions plus facile

- Prolog est un bon exemple de séparation de ces deux composants où les programmes (règles Prolog) ne prévoient pas en général une manière particulière d'application de la résolution. Le programmeur connaissant la sémantique procédurale de la SLD résolution écrit des règles modélisant un problème donné.

- Les méthodes et techniques de résolution de contraintes doivent être indépendantes des règles spécifiant un problème.

- Si l'on peut résumer les programmes par "**programme = algorithme + structure de données**", R. Kowalski considère, par sa formule bien connue, "**Algorithme = Logique + Contrôle**".

La transposition de cette formule en programmation logique (Prolog) considère :
Algorithme = logique de 1er ordre (partie déclarative) +
Contrôle = la résolution (partie procédurale).

En programmation sous contraintes, en particulier dans dans paradigme lanage déclaratif, la partie contrôle est cachée ou bien spécifiée déclarativement. On peut donc reformuler l'équation de Kowalski par "Algorithme = Logique" où la partie logique peut spécifier (logiquement) du contrôle.

Exemples de CSP :

- Croptoarithmétique : il s'agit de trouver des valeurs pour les variables (S,E,N,D,M,O,R,Y ∈ {0..9}) sans doublon telles que l'équation suivante soit vérifiée :

```
    S E N D +
    M O R E
----------
  M O N E Y
```

- Génération de grilles de mots croisés : produire une grille à partir d'un lexique (un dictionnaire). Ici les contraintes celles imposées par des mots ayant des lettres en intersection avec d'autres mots.

- Construction de ponts

Ordonnancement d'un ensemble de tâches avec des contraintes de précédence et des contraintes disjonctives sur les ressources à utiliser. Le but est de minimiser la durée totale du projet et minimiser l'utilisation des ressources.

- Ordonnancement (Job Shop) : Une entreprise produit N produit. Chaque produit est fabriqué en plusieures étapes utilisant des machines particulières pendant certain temps. L'obkjectif est de minimiser la durée de fabrication des produit en respectant des précédences et en utlisant les machines au maximum.

- Emploi de temps (examen, salles, élèves...)

- .....

Classe de problèmes CSP

- Programmation en général
- Intelligence Artificielle et Recherche Opérationnelle
- En particulier
  - Problèmes d'affectation sous contraintes
    - coloration de graphes
    - ordonnancement,
  - Reconnaissance d'image
  - Isomorphisme et Homomorphisme de graphes,
  - Résolution booléenne
  - Puzzles, ...

Caractéristiques des problème CSP

- Problèmes combinatoires (souvent NP-Complet[1]) où
  ➥
  Arbre de recherche de taille importante
  Très grand nombre de configurations possibles

- Exemples
  - Croptoarithmétique S E N D + M O R E = M O N E Y
  8 variable chcune à valeur dans {0 .. 9}
  ➪ 10^8 combinaisons

  - Composition de mots croisés
  Pour une grille ordinaire (matrice 20 x 30) et un lexique de près de 150 mots avec une répartition équitable selon les longueurs (2 à 9 lettres) des mots et leur fréquence dans dans les dictionnaires

  ➪ $4!^3\ 15!^3\ 30!^3$ solutions potentielles

[1] d'une complexité non polynomiale en temps de calcul, caractéristique des problèmes dont la solution est obtenue d'une manière non déterministe où toutes les combinaisons des valeurs des variables doivent être considérées (générer-tester). Des exemples représentatifs de cette classe de problèmes sont : le problème du circuit hamiltonien, le problème du voyageur de commerce, le problème du sac à dos, la satisfaction booléenne, ...)

- Ces problèmes n'ont pas de solution satisfaisant en Prolog classique et sont souvent traités par des systèmes spécifiques (impératifs). Nous verrons dans les sections suivantes les caractéristiques de Prolog qui en font un bon langage candidat à l'implantation des systèmes de contraintes.

Prolog comme un langage de programmation sous contrainte

- La possibilté de définir, par des énoncés déclaratives, des relations entre les objets procurent un moyen naturel d'expression de problèmes par un ensemble de contraintes.

- Prolog est un bon candidat pour la programmation (logique) sous Contraintes. Les aspects suivants sont déjà présents en Prolog :
  - Relationnel
  - SLD-Résolution et Retour arrière (non déterminisme)
  - La séparation de la logique et le contrôle

- En Prolog, on exprime les contraintes par :
  - Les prédicats qui permettent d'exprimer des relations entre leurs arguments. Il n'y a pas de direction (entrée/sortie) dans les arguments des prédicats; ce qui convient bien à l'aspect équationnel des contraintes.
  - Par l'unification qui crée et résout des contraintes **d'égalité** entre les termes.

  En effet, l'unification de Prolog est déjà un solveur de contraintes d'égalité constructive qui n'attend pas que les termes soient instanciés pour tester leur égalité. Quand une variable prend une valeur, cette valeur est **propagée** vers d'autres variables.

Exemple1 : 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,]).
.....
```

- Ici, le prédicat *mot* impose le choix des mots de taille donnée alors que la variable A5 permet, par l'unification, d'imposer l'égalié des valeurs présentes à certains emplacements.

Méthodes de résolution de CSP en Prolog standard

Nous étudions les différentes techniques de résolution de problèmes (dont CSP) à travers l'exemple suivant :

Exemple :

Le problème de coloration de carte. Il s'agit de dolorier la carte ci-dessous en affectatnt des couleurs rouge, vert et bleu différentes aux régions voisines.

Description de la solution en Prolog

On considère trois approches :

- Constructive

- Générer -Tester
- Tester - Générer

Schéma Constructive

- A partir d'une configuration initiale S_0, on génère la configuration partielle S_1 successeur de S_0 dont la conhérence (les contraintes) est testée dès que possible (dès sa génération). Ainsi, la cohérence de la solution est vérifiée lorsqu'à l'étape n, la configuration partielle S_n (successeur de S_{n-1}) est la configuration finale recherchée.

- Le schéma général de la méthode constructive est donné par :

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

- Pour l'exemple de coloration, on obtient une solution :

```
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.
```

- Réponses obtenues:

C1	C2	C3	C4	C5
r	b	v	b	r
r	v	b	v	r
b	r	v	r	b
b	v	r	v	b
v	r	b	r	v
v	b	r	b	v

Critiques

- Le programme est inefficace. Un échec du test *voisin(C4, C5)* remet en cause les succès précédents et on risque de reconsidérer toutes les combiniasons possibles avant de se rendre compte que la cause des échecs se trouve dans les tests initiaux. On procède à un recouvrement a posteriori de l'échec.

- Pour des problèmes de taille importante, l'arbre est trop grand et développé d'une manière aveugle. Le seule mécanisme utilisé est le retour arrière. L'introduction d'une heuristique confondue avec les autres composants (génération et test) conduit à une perte de la déclarativité des programmes.

- Amélioration possible par le mécanisme de retardement (attendre linstanciation des variables avant d'effectuer les tests) + retour-arrière intelligent.

Schéma générer / tester

- Le schéma général de cette méthode est

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

- Pour ce schéma, on peut envisager un prédicat test qui vérifie si une solution est juste, puis un prédicat de génération. Un schéma de permutation de la liste des variables permet de présenter toutes les combinaisons de valeurs pour les variables.

- Pour l'exemple de coloration, on construit une combinaison de cinq couleurs puis on vérifie les contraintes.

```
coloration(C1,C2,C3,C4,C5) :-
      cinq_couleurs(C1,C2,C3,C4,C5) ,     % génération d'une
configuration
      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).
```

Critique

- Le programme est très inefficace pour les mêmes raisons que le précédent. Il semble évident que certaines configurations ne peut figurer dans la solution finale. La production de ces configurations peut être évitée immédiatement. Cette méthode est cependant utilisé lorsque l'on dispose déjà d'un prédicat qui teste si une solution est acceptable. Dans un perspective de réutilisation, on produit une configuration sur laquelle certaines contraintes (nouvellement ajoutées) sont vérifiées.

- Pour des problèmes de taille importante, l'arbre est trop grand. Le seule mécanisme utilisé est le retour arrière.

- On procède toujours à un recouvrement a posteriori.

Schéma tester / générer

- Dans cette méthode, on pose les contraintes puis on génère les combinaisons. Les contraintes posées ayant un effet global sur l'environnement, les combinaisons invalides seront rejetées immédiatement et ne seront pas générées. De plus, les contraintes étant testées avant toute génération, on évite de procéder à la génération dès lors que les contraintes sont jugées inconsistantes.

- Le schéma général de cette méthode est

  p(...) :-

  Contraintes à vérifier (accompagné de gèle des variables)

Génération des configurations totales.

- Les prolog classiques ne peuvent pas implanter de telles méthodes car la mise en place de cette méthode nécessite la modification de la règle de calcul de Prolog pour disposer du mécanisme de retardement (PrologII, Mu-Prolog, D-Prolog, ...).

- Une solution en PrologII utilisant la contrainte *dif* :

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

- *dif(X,Y)* pose la contrainte suivante : les deux termes X et Y doivent être différents dès que leur valeurs sont connues. Notons que ce test se fait dès que possible mais toujours à posteriori (voir plus loin pour *dif*).

- Comme dans la méthode Générer-Tester, on peut prévoir de un prédicat qui teste si une solution proposée est bonne, puis on écrit un autre prédicat qui génére des configurations. Les Prolog classique échouent sur les tests qui contiennet des variables.

Utilisation de Prolog standard dans les CSP

- La programmation logique exemplifiée par Prolog est un outil de description puissante pour le CSP.

- Pour un CSP donné, il suffit de lui associer un programme logique avec chaque type de contrainte et proposer un générateur de valeur pour les variables. Cependant, les langages de programmation logique actuels sont très inefficaces pour exécuter ce programme déclaratif a cause de la stratégie qu'ils emploient.

- Lorsque le mécanisme de retardement est disponible (IC-Prolog, PrologII, Meta-Log, Mu-prolog, D-prolog...), les contraintes peuvent être utilisées avec les générateurs menant a une stratégie "en-profondeur-d'abord" avec retour arrière chronologique. Nous étudions ce mécanisme plus loin dans ce chapitre.

Insuffisances de Prolog

- Malgré ses aspects, Prolog n'est pas bien adapté pour la résolution de certaine classe de problèmes car :

- Il est peu efficace en arithmétique et en résolution de systèmes d'équations.
- Les termes fonctionnels de l'univers de Herbrand (U_H) ne sont pas interprétés. Un terme tel que X= Y+Z dénote l'égalité purement syntaxique des arbres X et Y+Z. En outre, Y+Z n'a qu'une interprétation de Herbrand. Par exemple, avec l'alphabet {0, succ}, on a :

  U_H = {0,succ(0), succ(succ(0)), ...}

  Pour faire de l'arithmétique en Prolog "pur", on peut écrire les axiomes de Peano :

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

  Bien que Prolog puisse être employé pour décrire n'importe quel problème (on dit alors qu'il a le pouvoir d'expression de la machine de Turing), On peut difficilement envisager faire de l'arithmétique avec le prédicat "add" ci-dessus. Ceci serait particulièrement inefficace.
  Une autre solution pour résoudre par exemple X= Y+Z serait d'écrire tous les prédicats qui testent les états des variables et effectuent l'action appropriée par le prédicat IS de Prolog (c'est ce qui est demandé quand on interface prolog avec un langage impératif). Ce qui entraine une perte de l'aspect déclaratif de Prolog.

- Les termes du 1er ordre employés en Prolog sont trop généraux et ne suffisent pas pour programmer des application réelles. De plus, il est parfois nécessaire de traduire (coder) d'une manière "non naturelle" certains domaines sous forme de clauses de Horn avec les termes de l'U_H. Les problèmes pouvant être résolus en algèbre de bool (tel que A & B <=> C | D & ~E) en sont un exemple.

- Pour faire de l'arithmétique en Prolog, les variables doivent être instanciées. Ce qui nécessite une écriture séquentielle au détriment de la déclarativité et entraine la perte de la réversibilité (les arguments deviennent directionnels).

Considérons le prédicat 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.
```

- Le prédicat "fib" est non réversible : *fib(Y,89)* échoue car dans *N1 is N-1*, N étant une variable, le prédéfini IS échoue.

- Le test N>=2 en tête du prédicat récursif échoue car ce test opère sur des arguements instanciés.

- De plus, la résolution fait une utilisation passive des contraintes. Les échecs sont découverts a posteriori; ce qui entraine une exécution inefficace

- Pour l'exemple de **SEND+MORE=MONEY** et la solution suivante (schéma Générer-Tester), on obtient une réponse après 1/2 heure de calcul sur un MacII :

```
sendmory(S, E, N, D, M, O, R, Y) :-
        M = 1,
        retenue(C1), retenue(C2), retenue(C3),
        chiffre(S),    chiffre(E),    chiffre(N),    chiffre(D),
        chiffre(O),    chiffre(R),    chiffre(Y),
        differents([S, E, N, D, M, O, R, Y]),
        D + E =:= Y + 10 * C1,
        C1 + N + R =:= E + 10 * C2,
        C2 + E + O =:= N + 10 * C3,
        C3 + S + M =:= O + 10 * M.

retenue(0).    retenue(1).
chiffre(0).      .....  chiffre(9).
differents([]).
differents([X|Y]) :- hors_de(X, Y), differents(Y).
hors_de(_,[]).
hors_de(X,[Y|Z]) :- X \== Y, hors_de(X,Z).
```

- Ainsi, le Prolog classique n'est pas efficacement adapté à la résolution de contraintes et au traitement de l'arithmétique. Il utilise les contraintes d'une manière passive et réduit l'espace de recherche a posteriori.

- Une adaptation de Prolog nécessite la modification de la règle d'inférence utilisée en Prolog pour permettre d'implanter des heuristiques, d'intégrer des fonctions interprétées sur différents domaines et permettre la **propagation des contraintes.** De telles techniques permettront de programmer directement dans les domaines appropriés. On peut envisager d'écrire des programmes Prolog qui incorporte ces techniques mais cette approche deviendrait complexe et trahit l'esprit déclaratif de caréactéristique de la programmation logique.

De la programmation Logique au CLP

- Le support de Prolog comme langage de spécification de CSP est intéressant par :
  - Sa forme déclarative et relationnelle

- L'unification (à étendre)
- Sa possibilté de calcul non déter
- La séparation de la logique et le contrôle.

- Afin d'étendre Prolog et d'en faire un langage de programmation sous contraintes, il fudra :

  - Enrichir U_H et intégrer les fonctions interprétées sur un domaine D. Disposer d'un domaine plus riche que l'U_H procure un pouvoir d'expression et de calcul plus grand avec des termes plus expressifs et nécessite une extension de l'unification.
  - Modifier la règle de calcul (nous avons vu au chapitre-I le principe d'indépendance de la règle de calcul)
  - Introduire des contraintes ($<,\leq,...\neq$) pour exprimer des relations telles que :

    $X+Y < 10 +Z$

    $X \neq Y...$
  - Introduire des extensions et les moyens de traitement de contraintes symboliques tels que les intervalles :

    $X \in$ [lundi, mardi, ..., dimanche]

  - Introduire la propagation des contraintes et plus généalement des techniques de résolution de contraintes.
  - Introduire des techniques d'exploitation active des contraintes pour mettre en place le recouvrement a priori des échecs.

Exemple :

- Une version CLP de factorielle :

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

- Calculer le nombre de pattes des chevaux (Y) et des hommes (X), X et Y son 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 d'équation
            {X+Y =8, 2X+4Y = 20}
```

Intérêts d'un CLP

- Grace à ces caractéristiques, la CLP facilite la programmation déclarative. Ce qui entraine une meilleure spécification de l'intuition. Dans une majorité des cas, l'interprétation attendue est clairement visible dans le programme et on travaille directement avec le domaine de discours (par exemple les réels).
  Ainsi, le codage des objets de l'univers de discours en U_H n'est plus obligatoire.
  Les propriétés complexes du problème sont spécifiées d'une façon naturelle (par exemple via des contraintes arithmétiques) tandis que le problème lui même est représenté par un ensemble de règles.

- Dans un CLP, les contraintes sont utilisées pour représenter des relations entre les objets et pour calculer des valeurs basées sur ces relations (cf. fonctions interprétées) :

  p(X, Y, Z) :- q(max(X,Y), Z).

  Ici, les prédicats p et q définissent des contraintes. de plus, max(X,Y) précise une propriété particulière sur le premier paramètre du prédicat q. De même, la contrainte *Z = 2Y +3* spécifie une relation entre Z et Y et permet de calculer des valeurs pour ces variables.

- Les règles récursives permettent l'ajout dynamique de contraintes.

- Dans un CLP, la résolution est Contrôlée par les contrainte. Ainsi, on vérifie à chaque pas de la résolution que le système de contraintes est solvable.

- On peut utiliser les contraintes pour améliorer la "pureté" de Prolog. Ainsi, on peut éviter dans certains cas l'utilisation de *cut* et de *not* Par exemple, l'écriture :

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

Peut être remplacé dans certains cas par :

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

où **q'** est le complément de **q** (par exemple, si q∈ {=,<}, q'∈ {≠,≥} :

```
p(X, Y) :- X ≥ Y , r(X, Y).
p(X, Y) :- X < Y , t(X, Y).
```

- De même, lors de l'utilisation de *not* dans les relations (contraintes) R sur des termes des domaines, on pourra utilisers directement R et *not R*. Par exemple, au lieu d'écrire :

```
p(X, Y) :- X = Y , r(X, Y).
p(X, Y) :- not (X = Y) , t(X, Y).
```

On écrira :

```
p(X, Y) :- X = Y , r(X, Y).
p(X, Y) :- X ≠ Y , t(X, Y).
```

- Dire qu'il y a deux grandes classes (CHIP-PIII)
  - présentation très très courte en énumérant les domaines traités dans chaque
  - mais il y en a d'autres (e.g. generalized propagation)

Domaines d'applications de CLP

- Programmation en général et CSP en particulier où les problème sont spécifiés naturellement par un ensemble de contraintes.

- Traitement non déterministe (point de succès de Prolog) dans les problèmes combinatoires. Jusqu'à présent, la complexité des algorithmes nécessitait un traitement spécifique, un codage des heuristiques complexes confondue avec le traitement. Il arrive fréquement que ces systèmes spécifiques se contentent d'une première réponse obtenue.
  Des exemples typiques sont la classe de problème tel que SENDMORY (problème d'affectation).

- Les programmes CLP sont réversibles par "nature". S'il reste vrai que les programmes non réversibles sont plus efficaces, il est néanmoins exact que rendre les programmes réversibles lors que ceci est nécessaire ne réclame pas un effort trop important.

- Les CLP remplacent d'une façon générale les Prolog classiques, le langage Lisp (et ses applications en Intelligence Artificielle) ainsi que d'autres langages tels que MacSyma (utilisé pour leur capacité de traitement des contraintes numériques).

- Analyse numérique et Recerche Opérationnel :
  - La possibilié de pouvoir exprimer un grand nombre de constriantes sous forme d'égalité, d'inégalité et autres est intéressant pour ceux qui par exemple sont intéressés aux l'approximation des équations différentielles de Laplace ou aux méthodes telles que la méthode Runge-Kutta pour la résolution d'équation différentielle (applications en électronique, Physique,...) et en Analyse numérique en général.

- Applications en Intelligence Artificielle. Les langages CLP tels que Prolog-III et CHIP dans la satisfaction de contraintes booléennes sont déjà utilisées dans les systèmes experts.

- TLN où les nombres et chaines jouent un rôle importants. La possibilité d'avoir des programmes réversible ouvre des domaine d'applications (reverse-xx) comme dans l'exemple de Option-Trading, opérations financières,...

- Application d'ingénieurs : des problème tels que la conception de circuits électroniques avec les lois d'Ohm et Kirchoff pour générer des équations (fonctions) de description du comportement des circuits contenant des composants parallèles ou séries. De même, l'analyse et la détection de pannes ont déjà été programmées en utilisant les langages CLP.

- Infographie, contraintes géométriques, conception d'interface Homme Machine, ...

- Les ingénieurs sont intéressé par la résolution des problème de CSP. Cette classe de problème a une importance majeur en IA et englobe un certains nombre de problèmes tel que la coloration de graphes, isomorphisme et homomorphisme de graphes, satisfaction booléenne, étiquetage d'arbre et les puzzles logiques.

Les domaines de calcul traités dans les CLP

- D'une facon générale, unifier deux termes de premier ordre t_1 et t_2 dans un domaine de calcul D revient à résoudre l'équation $t_1 =_T t_2$ où T est la théorie d'égalité définie dans D.

- Une solution à cette équation est une substitution θ qui rend t_1 et t_2 identiques dans la théorie T.
  Ce qui veut dire que selon la théorie T, la classe des contraintes d'égalité traitées par un langage ediffère.

- En Prolog, cette théorie est vide. Les termes de U_H sont unifiés s'ils sont syntaxiquement identiques (unification de Robinson).

- La caractéristique des CLP est d'étendre la théorie T en définissant un domaine de calcul.
  Il y a différents façons d'étender la théorie T :
  - Par les règles de réécriture des termes ;
  - Par incorporation de T dans l'unification.

- L'intérêt de la première approche utilisant les termes canoniques (ou normales) obtenus par surréduction (une forme de réécriture) est que l'utilisateur peut définir sa propre théo d'égalité. L'inconv est que l'on risque d'avoir une infinité de MGU au lieu d'un. De plus, l'unification dans ce cadre est moins efficace.
- Dans la seconde approche, il faut choisir et figer le domaine. Chaque équation doit être résolue efficacement de facon déter. Dans ce cas, le domaine doit être suffisament utile pour justifier l'extension.

- Les domaine candidats à être incorporer dans les CLP sont :
  • Le domaine des termes de l'arithmétique linéaire :
  Il s'agit des entiers, des réels et des rationnels. Des méthodes bien maîtrisées comme la méthode d'élimination de GAUSS (pour les équations) et Simplex (pour les inéquations sont connues dans ces domaines. Des exemples de CLP implantatnt ces domaines sonts :
  PrologIII : réels, entiers, rationnels
  CLP(R) : réels (et par conséquent les entiers).
  PrologIII : réels, entiers, rationnels

  • Le domaine des Booléens :
  On considère un seul MGU, les algo d'unification sur les anneaux booléens peuvent s'appliquer à ces termes.
  Les langages CLP traitant de ces domaines sont PrologIII et CHIP.

  • Le domaine des chaînes (et les tuples) :
  Une forme restreinte de l'algorithme d'unification de "mots" est appliquée. La concaténation au sens véritable est définie sur ces termes.
  Seul PrologIII implante ces termes.

  • Le domaine des Complexes :

Langage CAL.

- Les domaine finis :
  Employé en CSP, CHIP implante les domaines finis.

- Les ensembles :
  CLP(Σ*)

- Les intervalles réels :
  BNR.
- ...

Le domaine arithmétique linéaire

- Possibilité de disposer des fonctions interprétées
- Termes avec les nombres, les variables, les opérateurs (-, +, *, /, ...).
- Les contraintes de la forme *r1 R t2* avec R∈ {>, ≥, <, ≤, = , ≠}
- Résolution avec l'algorithme d'élimination de Gauss et de Simplex.

1- Une version CLP de **fib**(X,Y) :

```
fib(0,1).
fib(1,1).
fib(N, R1+R2) :-
      N > 1, fib(N-1,R1), fib(N-2, R2).
```

où N>1, N_1, N-2 sont des contraintes

?- fib(10,F). => F=89
?- fib(N,89).=> N=10
?- 80 < F, 90 > F, fib(N,F) , !.==> N=10, F=89.

2- 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

- Une solution en PrologIII :

```
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 };
```

- Question :

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

➥ Il faut 12 configuration-1 et 12 configuration-3
Le total des chutes = 312 cm

Arithmétique non linéaire

Exemples :

1• Déterminer l'équation représentant des cercles qui passent par deux points X et Y.

```
cercle(X,Y, cercle(A,B,(X-A)2 + (Y-B)2)).
```

X et Y sont les coordonnées du point sur le périmètre du cercle
A et B : les coordonnées du centre. On aura $R^2= (X-A)^2 + (Y-B)^2$

- Question :

```
?- cercle(7,1,C), cercle(0,2,C).
```

On obtient le système de contraintes :

```
2*B-14*A+46 = 0
50*A2-350*A+625=R2 = C
```

2• Multiplication des complexes

```
zmul(R1, I1, R2, I2, R3, I3) :-
    R3 = R1 * R2 - I1 * I2,
    I3  = R1 * I2 + R2 * I1.
```

- Question :

```
zmul(1, 2, R2, I2, R3, I3).
```

On obtient

```
I2 = 0.2 * I3 - 0.4 * R3
R2 = 0.4 * I3 + 0.2 * R3
```

- La plupart des systèmes gèlent l'évaluation de ces contraintes.

Le domaine booléen

- Unification (booléenne) dans les anneaux booléens.
- L'intégration des expr bool en programmation logique nécessite l'unifiaction dans l'algèbre de bool.
- La satisfiablité booléenne est NP-complet. Ce qui veut dire que certains problème ne peuvent se résoudre que par des choix initiaux et heuristiques.

- La résolution des contraintes bool est de plus en plus utilisée dans la conception de circuits. L'apport principal est d'aider à prouver les fonctions attendues d'un circuit, détection de pannes, ...
  Lors que le circuit est simple, on peut se contenter de générer toutes les valeurs d'entrées/sorties et de faire des tests. Mais cette technique devient impraticable lorsque celui-ci devient coomplexe et nécessite des programmes spécialisés. L'unification bool est un moyen élégant de résolution de ce genre de problèmes.

Exemple : Le Circuit croisé.

On veut montrer que ce circuit établit la relation A=Y, B=X.

- Une solution en PrologIII :

```
circuit(X,Y,A,B) :-
        {I4 = ~x | I3, I3=x & y, I5=~y|I3,
         I8 = ~I4|I3, I9 = ~I5 | I3,
         a= I4 & I11, I11=I8 | I9, b=I5 & I11};

circuit(X,Y,A,B).    ==> {Y => A,  A => Y, X=> B,  B => X}
circuit(0',1',A,B) .   ==> {A = 1', B = 0'}
```

```
circuit(1',0',A,B) .  ==> {A = 0', B = 1'}
```

- Dans ces cas, le système n'est pas restreint à vérifier des circuits. Par exemple, à partir de la spécification d'un circuit, on peut obtenir un autre circuit plus spécialisé (et moins cher) qui réalise un sous ensemble de la fonction de départ (le cas arrive souvent qd on achète par exemple une bascule pour faire un "et" booléen !). On veut de plus remplacer certaines portes avec d'autres, par exemple, utiliser seulement des portes "nande" et "not". Le but ultime étant de minimiser le nombre de composants dans le circuit simplifié.
- On appliquera l'unification bool pour simplifier des circuits et supprimer des portes redondantes (l'exemple de circuit en est un de simplification).
- D'autres règles peuvent être :
  • si les entrées d'une porte & sont identiques alors éliminer la porte;
  • si les entrées d'un & sont un signal et son inverse, remplacer le & par un 0.
  • si deux portes calculent la mêmes sorties, en supprimer une.
  ....

- On peut remplacer une porte par une autre. Ce traitement relève de la réécriture et peut se mettre en place par des prédicats tels que :

  or(X,Y,Z) :- not(X,X1), not(Y,Y1), nand(X1,Y1,Z).

Les domaines finis

- Lorsqu'un domaine fini est codé dans un langaeg CLP (entier, réel, rationnel, bool, tuples, chaînes), on a la possibilté de raisonner directement dans le domaine du problème en profitant des possibilités du système dans la réslution par des techniques de consistance et d'autres méthodes de parcours de graphes.
- La spécifité des domaines fini est la suivante :
  Les domaines des variables étant connu d'avance (par exemple, on sait qu'une variable $X \in \{1,2,3,4,5\}$), on peut "se permettre" de générer des valeurs pour ces variables, une fois que le système de contraintes aura été simplifié.

Exemple : graphe de précédence de tâches

- Soit un problème d'ordonnancement de tâches. Le diagramme suivant montre les précédences des tâches : les tâches à droite doivent s'effectuer avant celles se trouvant à leur gauche.

où la durée de chaque tâche est d'1 heure. On peut déduire que le début de chaque tâche $\in \{1,2,3,4,5\}$ et la durée totale est < 7.

- On exprime les contraintes par :

avant(T1, T2)	noté par	$D1 < D2$
avant(T1, T3)	noté par	$D1 < D3$
avant(T2, T6)	noté par	$D2 < D6$
avant(T3, T5)	noté par	$D3 < D5$
avant(T4, T5)	noté par	$D4 < D5$
avant(T5, T6)	noté par	$D5 < D6$
diff(T2, T3)	noté par	$D2 \neq D3$ (T2 et T3 disjonctives)

- On utilise les contraintes et les domaines pour restreindre les domaines des variables :
  Les contraintes entre T1, T3, T5 et T6 consuisent à réduire les domaines de ces variables en :
  $T1 \in \{1,2\}$, $T3 \in \{2,3\}$, $T5 \in \{3,4\}$, $T6 \in \{4,5\}$

  - Sachant que $T2 > T1$, on obtient dans un premier temps :
    $T2 \in \{2,3,4,5\}$. De plus, on a $T6 > T2$ d'où $T2 \in \{2,3,4\}$.
    La disjonction entre T2 et T3 n'apporte rien à ce niveau.

- On a T5 > T4, on obtient T4 ∈ {1,2,3}.

- Les propagations successives auront réduit les domaines par :
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 ➪ 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}.

Méthodes et Techniques de résolution de contraintes

- Un CSP est composé de deux parties : un ensemble de règles et un mécanisme de contrôle.

- Les méthodes et techniques de résolution de contraintes doivent être indépendantes des règles spécifiant un problème.

- Les mécanismes décrits ci-dessous montrent l'évolution des techniques de résolution vers cet objectif.

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

- La composition de substitiution est une sorte de propagation de valeur.

- *Propagation* : transmettre la valeur d'une variable aux contraintes dans lesquelles elle figure.

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

- Une autres technique :
  - *Retardement* : retarder l'activation d'un prédicat (contrainte) jusqu'à ce qu'une variable change d'état.

  - *Réveil* : réactiver les contraintes retardées en attente d'une valeur de cette variableles.

- Les méthodes de résolution dans les CLP s'appuient sur des techniques plus élaborées de propagation que l'unification de Robinson.
  Ces langages divergent ensuite selon leur spécificité :

- CHIP et les domaines finis : implante les techniques de consistance et utilise la propagation.
- PrologIII : implante un solveur de résolution générale.

Retardement

- L'unification est le mécanisme de transmission de valeur (ou de domaine) d'une variables aux autres variables du système. On peut combiner l'unification et un **mécanisme de délai** (retardement) pour tester la satisfaction des contraintes. On appelle ce mécanisme **coroutinage** qui fonctionne par le cycle "Retardement/Réveil".

Délai

- Le mécanisme de délai a été proposé dans les dialects de Prolog tels que PrologII (prédicat *freeze*), D-prolog et dans certains Lisp. Mu-prolog implante le même mécanisme avec le prédicat WAIT. Le mécanisme de délai entraine une modification de l'ordre de la règle de calcul.

- En programmation, le retardement est souvent combinée avec énumération. L'idée de base est de retarder les tests (contraintes) jsq'à ce que les var soient suffisament instanciés. Ce qui permet de faire les tests dès que possible sans attendre la fin des énumérations.

Exemple de coroutinage :

1- Pour X = Y+Z, on écrirait :

```
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 Z+X)).
```

- Le coroutinage existe comme outile de programmation dans les langages. C'est le programmeur qui décide de l'employer dans ses programmes. Il a ensuit été incorporé dans PrologII sous certaine forme (e.g. le prédicat dif). Les CLP actuels l'appliquent implicitement.

- Une application de retardement est l'implantation du prédicat *not*.
  Par exemple, dans le prédicat de négation noté *~P* en Mu-Prolog, on attend que les variables de P soient suffisamment instanciées avant de prouver P.
  De même, dans le prédicat différent noté *X ~=Y*, on retarde le test de différence tant que X ou Y contiennent des variables non instanciées.

Dif (≠)

- Considérons l'implantation classique du prédicat
  X ≠ Y :- not(X=Y).

- Une règle correcte de négation par échec ne doit faire le test que si X et Y ont une valeur. Cette utilisation de ≠ est passive (on teste a posteriori si les deux termes sont différents).

- On peut implanter ≠ par *freeze* efficacement. Pour cela, on définit donc X≠Y par :
  • si X est une var et/ou Y une var alors attendre. Ce prédicat sera réveillé dès que les variables de X ou de Y changent d'état. On vérifie ensuite que les deux variables ont une valeur; sinon on regèle (X≠Y).
  • si X et Y sont cst alors X ≠ Y réussit si X est différent de Y.

  ➥ *X ≠ Y :- freeze([X,Y], not(X=Y)).*

- Un application de retardement (implicite) est le prédicat *dif* de PrologII (créé par A.Colmérauer).

  En PrologII, les termes sont des arbres infinis. L'unification dans ce système est ramenée à la résolution d'un système d'équations où les domaines des variables sont des arbres infinis (appelés arbres *rationnels*).

  Pour unifier deux termes t_1 et t_2, on applique des règles de réécriture à l'équation $t_1 = t_2$. Cette application continue jusqu'à ce que plus aucune règle de réécriture ne soit applicable.

Par exemple, pour unifier t_1 = f(X, g(a)) et t_2 = f(Y, g(X)) :

{f(X, g(a)) = f(Y, g(X))} => {X=Y, g(a) = g(Y)}
=> {X=Y, a=Y} => {X=a, Y=a}.

Opérant sur des arbres rationnels, l'algorithme d'unification de PrologII rend le test d'occurrence inutile.

Le retour arrière dans PrologII a lieu quand le système d'équations n'a pas de solution.

- En considérant les équations comme des contraintes, A. Colmérauer a considéré les diséquation (≠) par le prédicat *dif*.

  dif(t1,t2) avec t_1 et t_2 des termes (et donc des variables) impose aux arbres t_1 et t_2 à être différents. Si les deux arbres deviennent identiques à un point de calcul, alors on retourne en arrière.

Critique de coroutinage

- On s'intéresse aux techniques classiques utilisant le mécanisme de retour arrière (de prolog) plus le coroutinage.

- Le schéma de prgrammation avec coroutinage serait :

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

(cf. coloration de carte en PrologII à l'aide du prédicat dif).

- Le mécanisme de délai est néc mais ne suffit pas pour traiter les contraintes. Les contraintes sont encore utilisées de facon passive et l'espace de reche est réduit tjs a posteriori après avoir détecté l'echec lors d'une mauvais instanciation.

- De plus, il n'y a pas d'interaction entre les contraintes. Par exemple, dans

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

Question *p(Z,Z)*. ==> Z≥5, Z≤ 5 retardés (Z=5 est solution)
Question *q(Z,Z)*. ==> Z≥8, Z≤ 2 retardés (inconsistance).

- Le schéma génére-tester est inexploitable dès que la taille du problème devient important. De plus, le coroutinage souffre des problèmes svt :
  - la découverte continuelle des mêmes faits
  - générations inutiles et détection tardive des échecs
  - mauvais points de retour arrière.

- Des exemples de tel comportement pathologiques sont très courants dans la plupart de prog logique appliqué en CSP.

- Bien que coroutinage soit une amélioration du schéma généraer-tester classique, il souffre des mêmes problème dans les cas généraux. On peut envisager des techniques de retour arrière intelligentes à l'aide de graphes de dépendances pour mettre en place des points de choix intéressants.

- Dans ces techniques, on ne remet pas en cause un seul état en retournant en arrière (qui risque de nous redonner le même cas d'échec) mais on remonte plusieurs étapes en arrière. Un exemple typique simple est le problème des n-reines où le premier choix pour le premier pion est de toute façon mauvais mais, le mécanisme de retour arrière le découvrira bien après avoir essayé tous les cas possibles.

Propagation Locale

- Application implicite de délai.
- Le plus simple des mécanismes de résolutions est **la propagation locale** (des valeurs connues). Dans ce mécanisme, les valeurs connues des variables sont propagées le long des arcs du graphe des contraintes. Un noeud (contrainte) est activée lorqu'elle reçoit assez d'informations de ses arcs. Elle calcule alors une ou plusieurs valeurs pour les arcs qui n'en ont

pas et propage ces valeurs. Ces nouvelles valeurs peuvent causer l'activation d'autres noeuds (de la même contrainte).

- Cette technique a permis d'utiliser les contraintes dans les langages graphiques en 1980 par Borning, Steele et Sussman dans un langage appelé "Constraints".

- La propagation locale ne s'intéresse qu'à une seule contrainte (un ensemble de noeds) du graphe des contraintes.

- Les règles de contrôle sont locales aux noeud. Par exemple, le noeud "+" de ce schéma connaît les règles (selon le sens de prrcours) :

  Z <- X+Y, X <- Z-Y, Y <- Z - X

- Le principe de fonctionnement de ce mécanisme est montré par l'exemple ci-dessous:
  Soit la relation F=1.8 * C + 32. On met en place des techniques permettant de calculer F si C est donnée et vice versa en utilisant la propagation locale.

  Pour ce faire, la relation est représentée par une liste chainée. Si F et C restent inconnues, le résultat est la formule elle même.

- Si C est connue, on parcourt la chaîne "normallement" de gauche à droite. Si F est connue :

  x1 <- F - 32.0 règles sur le noeud "+"

C = x1 / 1.8 règles sur le noeud "*"
C = (F-32.0) / 1.8

- La propagation locale attend en général qu'il y ait au plus une variable. Ce qui revient à l'Affectation / Test.

Résolution par propagation généralisée

- On peut généraliser le principe de propagation.
- Dans un système fonctionnant par propagation, on détermine les variables après un nombre fini d'étapes de propagations locales. Une étape de propagation à lieu quand une contrainte a un nombre suffisant de variable déterminées pour déterminer ses autres variables. Les nouvelles variables donnent lieu à d'autres propagations.

- Habituellement, les variables peuvent être interdépendantes d'une façon cycliques. Il faut donc des techniques plus puissantes telles que la relaxation (on devine les valeurs des variables; on vérifie ensuite l'exactitude de ces estimations; on retourne en arrière au besoin) ou la transformation de graphes (réécriture; par exemple : 2*X = X + X).

Résolution par propagation

- La propagation locale tente de transformer un environnement initial qui ne satisfait pas les contraintes en un environnement qui les satisfait.

- Un environnement est constitué des couples variable/domaine dénoté par un ensemble de contraintes C. Un environnement associe un ensemble de valeurs aux variables. Par exemple, pour le système de contraintes :
  C= {X>0, Y = X+2},
  on peut avoir l'environnement :
  E = [(X={1}, Y={3}), (X=2), Y={4}), ...]

- On note par $f_c(\mathbf{E})$ l'application d'une contrainte **c** à un environnement **E** qui transforme E en modifiant l'ensemble de valeurs possibles de certaines variables.

Par exemple, si

E = [(X={1}, Y={3}), (X=2), Y={4}), ...] et

c={Y< 5},

alors f_c(E) produit :

E'= [(X={1}, Y={3}), (X=2), Y={4})]

- L'opération de *fusion* de deux environnement E1 et E2 notée **fusion(E1, E2)** produit un nouvel environnement E' où le domaine des variables communes à E1 et E2 est réduit à l'intersection de leur domaines dans E1 et E2. Ainsi, pour toute variable X commune à E1 et E2, si E1(X) = S1 et E2(X)= S2, alors E'(X) = S1 ∩ S2.

Algorithme de principe de propagation

```
fonction prpoagation(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', f_C(E))   -- M.A.J. des domaines
                  C' = C' ∪ {c}              -- des variables
            Fin pour
            C <- C - C'
            E <- E'
      Fin Tant que
      Retourne(E)
Fin
```

- Exemple de résolution par la propagation locale :

  C= {X>0, Y = X+2, Y < 5}

  E = [(X=1, Y=3), (X=2, Y=4)]

- Notons que si une contrainte n'est pas satisfaite dans l'environnement courant, alors la satisfaction du système de contraintes n'est pas prouvable par la technique de propagation locale.

Par exemple, la satisfiablité du système :

C= {X>0, Y = X+2, Y < 5, Y > 10}

La propagation sur le sous-système C= {X>0, Y = X+2, Y < 5} produit l'environnement E = [(X=1, Y=3), (X=2, Y=4)]. La contrainte {Y>10} n'est pas satisfaite dans E et l'algorithme boucle car C ne sera jamais vide.

Le système C n'est pas prouvable par la propagation locale et l'inconsistance de C doit être détectée par d'autres techniques.
On peut également modifier l'algorithme ci-dessus :

```
fonction prpoagation(C : contraintes; E: environnement ) :
Début                              <contraintes, environnement> =
      modifié = vrai
      Tant que (C ≠ {}) et (modifié) faire
            modifié = faux
            E' <- E
            C' <- {}
            Pour tout c∈ C telle que c est satisfaite dans E
                  modifié = vrai;
                  E' <- fusion(E', f_C(E))   -- M.A.J. des domaines
                  C' = C' ∪ {c}              -- des variables
            Fin pour
            C <- C - C'
            E <- E'
      Fin Tant que
      Retourne(<C, E>)
Fin
```

Résolution dans les domaines finis : anticipation

- Les problèmes rencontrés par le schéma générer-tester avec le coroutinage viennent du fait qu'on utilise les contraintes d'une manière passive (par exemple pour tester des valeurs). Ce qui réduit l'espace **a posteriori**, après avoir découvert un échec.

Il existe d'autres méthodes de recherche pour résoudre les problème de CSP qui sont basées sur l'utilisation active des contraintes pour réduire l'espace de recherche **a priori** (avant la détection de l'échec).
Dans ces méthodes, on supprime les valeurs qui ne peuvenet pas aparaitre ensemble dans la solution . Ainsi, les procédures sont orientées pour éviter les échecs et permettent à la fois une détection assez tôt des échecs et la réduction des retours arriers et de tests des contraintes.

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

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

- Lorsque le domaine des variables est fini, un traitement actif étudie les domaines des valeurs de X et de Y et donne notamment une valeur à celle dont le domaine est réduit à un élément. Ce mécanisme permet d'élaguer l'arbre de recherche a priori et entraine un gain substentiel.

- Exemple de techniques de **consistance** :
  Forward Checking/ Looking Ahead, ...
  Combiné avec **First Fail, best fit, branh & bound.....**

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

Exemple N-reines

Classification des langages CLP

- En fonction des techniques de résolution employées, on peut classer les langages CLP en deux grandes classes :
  1• CLP(X) basées sur des solveurs de contraintes (algorithme de Simplex, Gauss, ...). Ce sont les CLP généraux instance de CLP(X) dont le principe de base de fonctionnement de ces systèmes est la propagation. Ces systèmes ne font pas d'utilisation particulière du domaine éventuellement fini des valeurs des variables. Ainsi, il n'y a pas de génération de valeurs pour les variables (sauf si le domaine est réduit à une seule valeur).
  Ces systèmes acceptent les équations non linéaires et les traitent par le retardement.
  Dans cette classe de langages CLP, on note :
  • **Prolog-III** :
  Traite les contraintes du domaine de l'arithmétique linéaire (Simplex), un solveur de contraintes booléennes basé sur la méthode de saturation et un solveur de contraintes sur chaînes (listes finies).

  • Le langage **CLP(R)**, instance de CLP(X) : implante les contraintes sur l'arithmétique des réels (Simplex).

  2• CHIP : implante des techniques de consistance telles que le Forward Cheching, branch-and-Bond, if-then-else sur les contraintes...

- Hors mis ces deux classes, les recherches continuent pour permettre la prise en compte des contraintes spécifiées par l'utilisateur. Ce point sera abordé dans le chapitre "contraintes généralisées".

Résolution de contraintes par solveur CLP(X)

- Le cadre CLP étend celui de la programmation logique par les aspects suivants :

  • L'univers de Herbrand est sans restriction (non énumérable). Il n'est plus restreint aux termes de l'U_H obtenus associé à un programme Prolog.
  • L'unification est sans restriction. En effet, l'unification classique est un cas particulier de résolution de contrainte et l'obtention d'un MGU devient une notion d'implantation.
  • Il n'y a plus de restriction aux équations : une équation n'est qu'une forme particulière de contrainte.

- De plus, l'interaction entre le cadre logique et le cadre algébrique des contraintes est claire. Un exemple typique est PrologIII.

- Un langage CLP s'obtient par l'instanciation du schéma CLP(X) par :
  • La spécification d'une structure X (au sens mathématique) de calcul.
  • La spécification du domaine, les fonctions et les relations du domaine.

- Exemples :
  CLP(I) : (N, {+,-,..}, {=, ≠, >,<, ...}).
  CLP(Σ*) : (Σ, {∩, ∪,..}, {∈ ,⊂, =, ≠, ...}).

➥ **L'intérêt** : l'ensemble des instances se base sur le même cadre théorique CLP défini une fois pour toute.

- La programmation logique permet de spécifier des tuples de la relation caractérisant le programme (le prédicat principal). Cette notion est étendue en programmation sous contraintes.

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

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

➥ Un programme CLP spécifie des contraintes par des relations (prédicats) pour définir un sous ensemble de valeurs pour les variables vérifiant les contraintes.

➥ Un programme transforme (simplifie, résoud) les contraintes en entrées et produit les contraintes en sortie.

- De telles utilisations de contraintes donnent lieu à un langage déclaratif où, contrairement au langages impératifs (l'affectation), les contraintes ne sont pas directionnelles.

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

  En effet, il est préférable de travailler directement avec un domaine de discours (par exemple R) qu'avec univers de Herbrand dans lequel le données du problème doivent être codés. En CLP, le codage de ces domaines en U_H n'est plus utile.

- Dans un CLP, les contraintes sont utilisées non seulement pour représenter des relations entre objets, mais aussi pour calculer des valeurs basées sur ces relations (cf. function interprétées).
  ➥ $X = 2Y + 5$ établit une relation entre X et Y; calcule les valeurs de X et de Y.

- Dans un CLP, on peut avoir deux types de contraintes : statiques & dynamiques.

  • Contraintes statiques : il n'y a pas d'ajout dynamique pour chaque objet; tout est défini statiquement.
  • Contraintes dynamiques : l'ajout dynamique des contraintes est possibles. Ces contraintes interviennent par exemple dans le cas des règles récursives.

- Dans un langage CLP, le solveur de contraintes contrôle la résolution. Ce solveur doit pouvoir déterminer à chaque étape si une collection de contraintes est solvable (consistant) ou non.
  En fait, la classe de contraintes définies sur une structure est telle que la solvabilité est en général indécidable (ou bien non calculable ou encore non tractable). Par exemple, la classe de l'arithmétique des entiers est indécidable. Pourtant, les entiers sont utilisés dans les langages de programmation.
  Par exemple, dans 3=X*Y, on ne peut pas décider des valeurs de X et de Y. Pour traiter ces cas, les langages CLP retardent en général l'évaluation de la multiplication tant qu'une des variables X ou Y n'est pas connue.

  De même, en PrologIII où la "véritable" concaténation est définie sur les chaînes, la taille des chaînes doit être déterminé pour permettre la décidabilité de la concaténation. Ainsi, dans X=Y.Z (X est le résultat de la concaténation de Y et de Z), on est confronté au même problème que dans Z=X*Y.

  Cependant, le programmeur peut spécifier ces types de contraintes sachant qu'elles seront systématiquement retardées en attente d'une instanciation suffisante.

CLP(R)

- CLP(R) est l'instance réelle de CLP(X), c'est à dire, instance (réelle) de la structure algébrique composée de fonctors non-interprétés sur le domaine des réels lequel domaine est muni des foncteurs interprétés habituels : **+, *, /, -, =, =, >, <,....**

- CLP(R) dispose d'un solveur d'équations et d'inéquations qui procurent un traitement modulaire et multi-phase des contraintes.

- CLP(R) permet de spécifier les propriétés (complexes) du problème d'une façon naturelle (contraintes arithmétiques) pendnat que le problème lui même est représenté par un ensemble de règles.

- Du point de vue d'implantation des langages CLP, le solveur de contraintes doit être incrémentale : l'ajout d'une nouvelle contrainte ne doit pas remettre en cause les traitements faits sur les anciennes contraintes (à moins que le nouvel ensemble de contraintes ne soit démontré insatisfiable).
  Chaque classe de contraintes est prise en main par un algorithme particulier. Ce qui évite d'employer un algorithme trop général qui entrainera un surcout pour les contraintes simples.
  Par exemple, l'algorithme de Simplex sera utilisé pour le cas d'arithmétique linéaire tandis que l'algorithme d'élimination de Gausse sera plus efficace dans le cas où les contraintes sont uniquement des équations. On peut donc avoir un algorithme Simplex plus puissant à utiliser dans le cas des inéquations. Cette approche muti-phases nécessite l'établissement fiable de communication entre modules.

Résolution des contraintes en CLP(R)

- Le mécanisme de base est la **propagation locale.**

- Dans CLP(R), on détermine les variables après un nombre fini d'étapes de propagations locales. Une étape de propagation à lieu quand une contrainte a un nombre suffisant de variable déterminées pour permettre le calcul des autres variables. Par exemple, dans X > Y, on attend que les variables soient suffisamment instanciées pour pouvoir vérifier la contrainte. Les nouvelles valeurs des variables donnent lieu à d'autres propagations.

- La propagation locale attend en général qu'il y ait au plus une variable.

- En fait, les techniques utilisées dans CLP(R) sont plus puissantes et plus élaborées que la simple propagation locale. Ainsi, les contraintes linéaires sont traitées tout de suite et sont mise sous une forme normale alors que les contraintes non linéaires sont prétraitées (mises sous une forme canonique) avant d'être retardées jusqu'à ce que certaines conditions soient remplies.

- La propagation locale nécessite l'emploi du mécanisme de *délai/réveil* implicite (après pré-traitement) :
  - délai : on met de coté certaines contraintes (on teste pas leur solvabilité)
  - réveil : on réveille certaines contraintes (par propagation)

- Cette technique est semblable à celle utilisée dans les Prolog classiques avec retardement (Freeze de PrologII, Wait de Mu-Prolog, ...). Cependant, dans CLP(R), la stratégie de délai est générale et n'est pas précisée par l'utilisateur.

- En CLP(R), les nombres réels sont considérés une précision infinie (représentation en virgule flottante).
  Par conséquence, la résolution de certains problèmes de modélisation en physique est rendue possible.

Exemple : Les N-reines

```
n_reines(L) :- test(L), generer(L).
test([]).
test([X|Xs]) :-
        safe(X,Xs,1) , test(Xs).

safe(X,[],_).
safe(X,[Y|L],N) :-
      ne(X,Y), ne(Y, X-N), ne(Y, X+N), safe(X,L,N+1).
ne(X,Y) :- X > Y.
ne(X,Y) :- X < Y.

generer(L) :- indomaine(L).
indomaine([]).
indomaine([X|Y]) :- indom(X), indomaine(Y).
indom(1).
indom(2).
indom(3).
indom(4).
```

Le model opératoire de CLP(R)

La structure

- Une structure est définie par le triplet :

  <domaine de discours, opérations, relations sur ce domaine>

  Le modèle associé à cette structure doit être solution-compacte. Un modèle est solution-compacte si tout terme du domaine peut être défini par les relations définies dans la structure. Cette spécification peut être éventuellement infinie. Par exemple, la modèle associé à la structure <R, {+, *}, {=, <, ≤, >, ≥}> est solution-compacte et permet de représenter n'importe quel réel. On a par exemple

  $6.1 < 6.2 < 6.3$

  $6.19 < 6.2 < 6.21$ ....

  En revanche, la structure <R, {+, *}, {=}> ne l'est pas car elle ne permet pas de spécifier le réel 6.2.

- De plus, la théorie qui axiomatise les relations de la structure doit être satisfaction-complète : on doit pouvoir décider (et prouver) si une contrainte est solvable ou non.

- La structure R est une structure à 2 sortes :
  - les nombres réels
  - les arbres (sur les foncteurs non interprétés et les réels)

Les termes et les contraintes :

Termes :

- Les termes :
  - Les termes (classiques) comme en Prolog
  - les termes arithmétiques

- Les termes arithmétiques sont construits par:
  - Les nombres réels ,
  - Les variables ,
  - Les constantes ,

• Si t_1 et t_2 sont des termes alors $\mathbf{t_1 + t_2}$, $\mathbf{t_1 - t_2}$, $\mathbf{t_1 * t_2}$ et $\mathbf{t_1 / t_2}$ sont des termes.
(en fait, seuls les opérateurs "+", "*" et le "-" unaires suffisent)

Contraintes :

- Si t_1 et t_2 sont des termes arithmétiques alors $t_1=t_2$, $t_1<t_2$, $t_1\leq t_2$ sont des contraintes arithmétiques.
- Si t_1 et t_2 ne sont pas des termes arithmétiques alors seulement $t_1 = t_2$ est une contrainte (sur arbres) avec son sens habituel. Deux arbres sont égaux de la manière classique : égalité des racines et des sous-arbres récursivement.

- Une contrainte est solvable s'il existe une assignation aux variables telle que la contrainte soit vraie.

Autres définitions :

- Un atome : de la forme $\mathbf{P(t_1, ----t_n)}$ où $P \notin \{=, <, \leq\}$ et les t_i sont des termes.
- Une règle : de la forme $\mathbf{A_0 : - \alpha_1, ..., \alpha_k}$. $k\leq 0$, α_i : atome ou contrainte
- Un Programme : une collection de règles.
- Un but : de la forme $\mathbf{? - \alpha_1, ..., \alpha_k}$. $k\leq 0$, α_i : atome ou contrainte
  Chaque α_i dans le but est *retardé* ou *résolu* .

Exemple :

```
fib (0,1).
fib (1,1).
fib (N, X1 + X2) :- N>1, fib (N-1, X1), fib (N-2, X2)

? - 80 < = B, B < = 90, fib(A,B).      ==> A = 10, B = 89
```

Sémantique Oérationnelle

- soit le but (le résolvant) courant $G = A_1, ..., A_n. , n \geq 0$ avec
  $S = \sigma_1, ..., \sigma_m \quad m \geq 0$: les contraintes *résolues*

$D = \delta_1, ..., \delta_k$ $k \geq 0$: les contraintes *retardées*

- Un pas de dérivation de G donnant G_2 est obtenu par :

1 - La régle de choix sélectionne la contrainte $\boldsymbol{\delta_i}$ dans G

$G_2 = A_1, ..., A_n$. avec

$S = \sigma_1, ..., \sigma_m, \boldsymbol{\delta_i}$ et

$D = \delta_1, ..., \delta_{i-1}, \delta_{i+1}, ..., \delta_k$

tel que dans G_2, la conjonction des contraintes résolues S est solvable.

2- La régle de choix sélectionne A_i et une régle R (renommé) de la forme

$B :- B_1, ... , B_s$. $s \leq 0$ et les contraintes de R sont $\delta'_1, ..., \delta'_t$

On aura :

$G_2 = A_1, ..., A_{i-1}, B_1, ... , B_s, A_{i+1}, ..., A_n$. avec

$S = \sigma_1, ..., \sigma_m$ les contraintes résolues et

$D = \delta_1, ..., \delta_k, (A=B), \delta'_1, ..., \delta'_t$. les contraintes rétardées

(A=B) est une équation sur les foncteurs et les arguments de A et de B.

De plus la conjonction des contraintes résolues S de G2 est solvable.

- Dans un but initial, <u>toutes les contraintes sont retardées</u>.
- Une séquence de dérivation est possiblement infinie commençant par G. Elle est <u>réussie</u> (aboutit à un succès) si le dernier but contient uniquement des contraintes résolues.

- Une séquence est un <u>succès conditionnel</u> si elle est fini et son dernier but contient uniquement des contraintes résolues ou retardées.

- Une séquence à <u>échec fini</u> est finie, ni à succès ni à succès conditionnel telle que l'on ne puisse plus dériver.
- Les contraintes d'une séquence à succès ou à succès conditionnel sont appelés les **contraintes-réponses** qui constituent la sortie (la réponse) du Programme.

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

- Donc, un Programme CLP(R) et un but sont éxécutés par un processus continu de réduction des atomes ou de la résolution des contraintes retardées.

- Comme en SLD-Résolution,eux aspects non déterministes dans la dérivation : choix de A_i et le choix de régle R.

- Exemple

```
Ohm (V, I, R) : - V = I * R.
```

?- ohm(V1,I,R1), ohm(V2,I,R2), V=V1+V2, R1=15,R2=5 .
>>> but courant: ohm(V1, I, R1)
>>> Contrainte courante: V1 = I * R1
>>> but courant: ohm(V2, I, R2)
>>> Contrainte courante: V2 = I * R2
>>> Contrainte courante: V = V1 + V2
>>> Contrainte courante: R1 = 15
>>> Contrainte courante: R2 = 5

```
Réponse :
                V1 = 0.75*V
                I = 0.05*V
                R1 = 15
                V2 = 0.25*V
                R2 = 5
```

- Les réponses sont des contraintes décrivant un sous-ensemble des réponses:

Question :

```
?- ohm(12, I, R).
```

```
12 = I * R
*** Maybe ***
```

La résolution en CLP(R)

- Dans CLP(R), les équations sur les termes non arithmétiques sont traitées par l'unification classique sans le test d'occurrence.

- La technique de traitement des contraintes non triviales est basée sur le mécanisme de délai/réveil. Ainsi, on traite une contrainte quand elle est suffisamment simplifiée. Cette définition dépend de l'Algorithme de résolution de contraintes.

- CLP(R) utilise un schéma descendant/en profondeur d'abord dans les dérivations. Le moteur de CLP(R) fonctionne comme dans Prolog classique : la stratégie de choix de sous-but dans le résolvant est gauche-droite dans laquelle les contraintes non linéaires sont implicitement retardées jusqu'à ce que des solutions aux autres contraintes les rendent linéaires (suffisament de variables instanciées).

- Le solveur ne détermine pas la solvabilité de toutes les contraintes non linéaires.

- Une contrainte non linéaire retardée est réveillée le plus tôt possible. Si une contrainte retardée est réveillée et s'avère non solvable, on retourne en arrière dans le style classique de Prolog et il n'y a pas de recherche du point où la contrainte a été rencontrée la première fois.

Inconvénients de retardement

- Le retardement des contraintes non linéaires :
  a) Peut conduire à une dérivation infinie alors qu'il peut en exister une avec une dérivation à échec fini. Ce même problème est dû à la stratégie "en profondeur d'abord" et existe en Prolog.

Exemple :

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

La question ?- p(A) cause une dérivation infinie.

b) L'ensemble des contraintes-réponses peut être non-solvable.
Dans ce cas, les contraintes-réponses sont correctes car elles représentent un sous-ensemble de l'espace de solutions. En revanche, il n'y a pas de garanti que cet espace n'est pas vide.

Exemple :

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

La réponse est :

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

Méthodologie de programmation avec contraintes

- Schéma général contraindre-générer :

P(----) : -
contraintes,
générateurs.

Exemple SMM :

Puzzle Crypto-arithmétique.

Assigner des entiers 0..9 distincts aux variables S, E, N, D, M, O, R, Y tels que l'èquation suivante soit vérifiée : S et M ≠ 0

```
  S E N D
+ M O R E
----------
M O N E Y
```

```
solve([S, E, N, D, M, O, R, Y]) :-
➠ contraintes([S, E, N, D, M, O, R, Y]),          %CONTRAINTES
➠ gen_chiffres_differents([S, E, N, D, M, O, R, Y]). %GENERATION

contraintes([S, E, N, D, M, O, R, Y]) :-
    S <= 9, E <= 9, N <= 9, D <= 9, M <= 9, O <= 9, R <= 9, Y <= 9,
    S >= 1, E >= 0, N >= 0, D >= 0, M >= 1, O >= 0, R >= 0, Y >= 0,
    C1 >= 0, C2 >= 0, C3 >= 0, C4 >= 0,
```

```
        C1 <= 1, C2 <= 1, C3 <= 1, C4 <= 1,
        M = C1,
        C2 + S + M = O + C1 * 10,
        C3 + E + O = N + 10 * C2,
        C4 + N + R = E + 10 * C3,
        D + E = Y + 10*C4.

gen_chiffres_differents(L) :-
        gen_chiffres_differents(L, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).

gen_chiffres_differents([], _).
gen_chiffres_differents([Une_var | Autres_Vars], Domaine) :-
        gen_un_chiffre(Une_var , Domaine, Reste_Domaine),
        gen_chiffres_differents(Autres_Vars, Reste_Domaine).

gen_un_chiffre(H, [H | T], T).              % prédicat efface
gen_un_chiffre(H, [H2 | T], [H2 | T2]) :-
        gen_un_chiffre(H, T, T2).
```

- Question : ?-solve([S, E, N, D, M, O, R, Y]).
- Réponse : S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2

- L'expression des contraintes peut se faire d'une manière triviale et statique. Elles peuvent également être spécifiées par des règles récursives. On utilise souvent cette technique quand le nombre de composants n'est pas connu d'avance, par exemple, en traversant une structure de données récursive.

  Par exemple, pour contraindre tous les éléments d'une liste à être différents, on peut avoir :

```
        tous_diff([]).
        tous_diff([X|L]) :- hors_de(X,L), tous_diff(L).

        hors_de(X,[]).
        hors_de(X, [Y|L]) :- X ≠ Y , hors_de(X, L).
```

- On peut considérer une contraintes-réponse comme une évaluation partielle du programme. Une telle réponse est une instance plus spécifique du programme.

Exemple

Calcul des mensualités à verser d'un capital emprunté à un taux spécifique :

```
versement(Capital, Mois, Taux, Due, Mensualite) :-
        Mois = 1,
        Due = Capital + (Capital*Taux - Mensualite).

versement(Capital, Mois, Taux, Due, Mensualite) :-
        Mois > 1,
        versement(   Capital*(1 + Taux) - Mensualite,
                     Mois - 1, Taux, Due, Mensualite).
```

- Combien faut-il emprunter et quelle est la mensualité sur 12 mois à un taux annuel de 0.1 pour que l'on ne doive rien à la fin.

  ➥ ?- versement(P, 12, 0.1, 0, M).

  ➥ P = 6.81369*M

  Cette réponse désigne l'ensemble (potentiellement infini) de valeur des capitaux satisfiant les données de la question. Cet ensemble défini un sous-espace des tuples de solution spécifié par le prédicat *versement*.

- Rappelons que les sorties d'un programme CLP sont des systèmes de contraintes dont l'existance de solution ne peut pas toujours être garantie. On sait cependant que la réponse donne une équation correcte.

Raisonnement hiérarchique :

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

- Une contrainte représente une propriété locale du problème traité dans un programme CLP.
  Par exemple, le prédicat "ohm" spécifie les propriétés locales d'un élément quelconque d'un circuit :

```
ohm (V, I, R) : - V = I * R.
```

- Parallèlement, on utilise des contraintes *globales* pour exprimer comment ces contraintes interagissent.

- Dans CLP(R), les propriétés globales sont représentées par les règles.
- De même, les contraintes-réponses expriment les contraintes globales du problème traité.

- Le schéma le plus direct de représentation des propriétés globales est :

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

- Par exemple :

```
resistance (V, I, R) : - V = I * R.
```

décrit une propriété locale alors que

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

décrivent les propriétés globales.

- Egalement, on peut spécifier une hiérarchie multi-niveaux :

```
circuit_parallel_serie(VI, R1, R2, R3, R4) : -
        V1 + V2 = V,
        circuit_parallel (V1, I, R1, R2)
        circuit_parallel(V2, I, R3 R4).
```

CLP dans les problèmes de recherche combinatoire

- CLP est utilisé dans les problèmes combinatoires et les techniques de satisfaction de contraintes représentant la meilleure solution .

  Ceci implique une recherche dans un espace où les contraintes sont utilisées pour **guider** la résolution en générant des valeurs ou en supprimant des branches quand les contraintes sont non satisfiables.

- En toute rigueur, avec un solveur parfait, les contraintes seules sont suffisantes pour définir les réponses (valeur) sans avoir recours aux générateurs. Dans ce cas, c'est le solveur qui cherche les valeurs.

  Mais, le solveur parfait, c'est soit impraticable, soit impossible à obtenir pour un domaine donné.

- Il faut donc une compromis : conjointement avec un solveur partiel, on doit mettre en place une heuristique de génération de valeurs pour les variables.

- Deux approches ont été utilisées pour mettre en place de tels solver partiels :
  • Produire un solveur spécifique en utilisant le schéma délai/révail où la solvabilité des contraintes est vérifiée seulement quand elles sont suffisamment instanciés.

  • Utiliser un solveur sur un domaine différent que celui du problème. Par exemple, utiliser CLP(R) pour résoudre les problèmes dont le domaine est celui des entiers.
  Un autre exemple de telles utilisations est d'employer un solveur de booléens pour spécifier un problème "binaire" sur des entiers ou,

inversement, utiliser un solveur d'entiers pour résoudre des problèmes de l'algèbre de bool. Il faudra alors mettre en place un isomorphisme partiel de structures (sur les domaines, les fonctions interprétées et les relations).

- Pour prendre l'exemple d'utilisation de CLP(R) sur des entiers, le solveur détecte partiellement la non-solvabilité des contraintes sur des entiers. Le solveur sera partiel et représentera les inconvénients énumérés précédement. Ainsi, on peut avoir une réponse positive alors que les contraintes ajoutées peuvent être en contradiction avec les précédentes. Ce cas se produisent lorsque des contraintes sont retardées et non encore traitées :

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

==> X > 0
Y > 0
X = Y * Z

*** Maybe ***

- Le schéma général des spécifications d'un problème combinatoire est :

P(------) : -
Les contraintes primitives (par exemple, un coût à minimiser);
Les contraintes exprimées par les prédicats;
Les générateurs.

- Les générateurs instancient des variables sur un domaine. Cette méthode appelé "TESTER & GENERE" est meilleure que la méthode "GENERER-TESTER" car les contraintes sont testées avant les générateurs. Ceci permet d'éviter de générer des valeurs quand les C° sont inconsistants :
?- X <= 1, X >= 2. ==> non.

Exemple : un problème du domaine des entiers. Problème de localisation d'entrepôts

Il s'agit de 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 respectivement les coûts 18, 10 et 20 pour E1, E2 et E3

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

- Le problème est de trouver une configuration telle que le coût total soit inférieur ou égal à une limite (ici 60).

Solution en CLP(R) :

- Utiliser le domaine R pour modéliser ce problème.
- On installe les contraintes primitives (coût) puis les autres contraintes (par les règles) et finalement, les générateurs.

- 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.

```
approvisionnement(Ts, Ls, Cout, Limite) :-
        Cout <= Limite,
        configuration(Ts, Ls, Cout).

configuration([C11, C12, C21, C23, C32, C33, C43, C51, C53],
                              [E1, E2, E3], Cout) :-
        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,          % voir ci-dessous
        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([C11, C12, C21, C23, C32, C33, C43, C51,
                    C53, E1, E2, E3]).

genere([]).                 % simplifié pour cet exemple car on sait 0,1
genere([1 | Xs]):-  genere(Xs).
genere([0 | Xs]):-  genere(Xs).
```

- Les cinq premières contraintes expriment le fait qu'un client doit être associé à un seul entrepôt
- Les trois autres contraintes évitent qu'un client soit associé à un entrepôt ne figurant pas dans la configuration finale.

- Une question posée est

```
        ?- approvisionnement(A,B,C,60).

        ➥  B = [0, 1, 1]
           A = [0, 1, 0, 1, 1, 0, 1, 0, 1]
           C = 60
```

C'est à dire :

- Les entrepôts E1 et E2 seront implantés.
- Le client C1 est servi par E2
- Le client C2 est servi par E3
- Le client C3 est servi par E2

- Le client C4 est servi par E3
- Le client C5 est servi par E3

PIII

- Le point important dans les CLP : on peut résoudre des problème sans coder une méthode de résolution des contraintes et de parcours de graphe.
- Dans un langage CLP on peut mettre en place différentes méthodes :
  - générer/tester (avec des variantes et heuristiques comme CHIP)
  - constructive

Différences P-III, CLP(R)

- Ces différences concernenet la méthode de résolution de contraintes et le domaine des réels.

Note : en fait, il n'y a pas de réelle différence entre CLP(R) et P-III. PIII est en plus plus riche.

- En général la classe de C° définie sur une structure est telle que la solvabilité est indécidable (ou bien non calculable; non tractable). Par exemple, dans l'arithmétique des entiers, la clase est indécidable et pourtant, il en faut dans le langage de programmation.

Par exemple, en P-III, la taille des chaines doit être donné et la multiplication est abandonnée retardé systématique et non prise en compte immédiatement) (sauf par une constante). Ceci permet la décidabilité.

- Dans CLP(R) l'approche est le mécanisme de *delay/wakeup* automatique (après pré-traitement) :
  - delay : on met de coté certaines C° (on teste pas leur solvabilité)
  - réveil : on réveille certaines C°

Cette technique est semblable à celle utilisée dans les Prolog classiques avec retardement. Cependant, dans CLP, la stratégie de delay est générale et n'est pas précisée par l'utilisateur (par exemple, Freeze dans P-II)

- Une autre différence est qu'en P-III, les nombres réels sont considérés avec une précision dépendante de la machine. En revanche, les nombres rationnels sont codés avec une précision parfaite (avec autant de chiffres que leur expression exacte le requiert).

En CLP, les réels sont considérés avec une précision infinie. Par conséquence, la résolution de certains problèmes la modélisation en physique est traitable alors qu'en P-III, il faut faire ça autrement :

CLP(R) : une implantation approximative de l'arithmétique des réels parfait
P-III : une implantation parfaite de l'arithmétique des réels approximatives

Les autres langages CLP :

CHIP : utilisé en contraintes des entiers dans la recherche des combinatoires (comme en PIII).
CAL : Comme CLP(R) plus l'arithmétique non linéraire sur des nombres complexes

CLP(∑) : contraintes ensemblistes

%%% **LA partie double. à enlever après vérif°**
donné pour permettre la décidabilité de la concaténation de chaînes. Dans X=Y.Z, on est en face du même problème que dans Z=X*Y.

CLP(R)

- CLP(R) : instance réelle de CLP, c'est à dire, instance (réelle) de la structure algébrique composée de fonctor non-interprétés sur le domaine des réels lequel domaine est muni de foncteurs interprétés habituels comme :

  +, *, /, -, =, =, >, <,....

- CLP(R) dispose d'un solveur d'équations et d'inéquations qui procurent un traitement modulaire et multi-phase des contraintes.

- CLP(R) permet de spécifier les propriétés (complexes) du Problème d'une façon naturelle (contraintes arithmétiques) pendnat que le problème lui même est représenté par un ensemble de règles.

- Du point de vue d'implantation, dans un langage CLP, le solveur de contrainte doit être incrémentale : l'ajout d'une nouvelle contrainte ne doit pas remettre en cause les traitements faits sur les anciennes contraintes (à moins que le nouvel ensemble de contraintes se soit démontré insatisfiable.

Chaque classe de contraintes est prise en main par un algo particulier. Ce qui évite d'employer un algo général qui coutera trop cher.
Par exemple, l'algo de Simplex sera utilisé pour le cas d'arithmétique linéaire.
Cependant, l'algo d'élimination de Gausse sera plus efficace dans le cas où les contraintes sont uniquement des équations. On peut donc avoir un algo simplex plus puissant à utiliser dans le cas des inéquations. Cette approche muti-phase peut donner lieu à des problèmes de communication entre modules.

Résolution des contraintes en CLP(R)

- **La Propagation locale.**

- Dans CLP(R), on détermine les variables après un nombre fini d'étapes de propagations locales. Une étape de propagation à lieu quand une contrainte a un nombre suffisant de variable déterminées pour déterminer ses autres variables (cf. retard dans X ~=Y de Mu-Prolog : on attend que les variables soient suffisamment instanciées). Les nouvelles variables donnent lieu à d'autres propagations.

- La propagation locale attend en général qu'il y ait au plus une variable. Ce qui revient à l'Affectation/Test.

- En fait, les techniques utilisées dans CLP(R) sont plus puissantes et plus élaborées que la simple propagation locale. Par exemple, les contraintes linéaires sont traitées tout de suite et sont mise sous une forme normale alors que les contraintes non linéaires sont retardées jusqu'à ce que certaines conditions soient remplies.

Différences P-III, CLP(R)

- Ces différences concernenet la méthode de résolution de contraintes et le domaine des réels.

Note : en fait, il n'y a pas de réelle différence entre CLP(R) et P-III. PIII est en plus plus riche.

- En général la classe de C° définie sur une structure est telle que la solvabilité est indécidable (ou bien non calculable; non tractable). Par exemple, dans l'arithmétique des entiers, la clase est indécidable et pourtant, il en faut dans le langage de programmation.

Par exemple, en P-III, la taille des chaines doit être donné et la multiplication est abandonnée retardé systématique et non prise en compte immédiatement) (sauf par une constante). Ceci permet la décidabilité.

- Dans CLP(R) l'approche est le mécanisme de *delay/wakeup* automatique (après pré-traitement) :
  - delay : on met de coté certaines C° (on teste pas leur solvabilité)
  - réveil : on réveille certaines C°

Cette technique est semblable à celle utilisée dans les Prolog classiques avec retardement. Cependant, dans CLP, la stratégie de delay est générale et n'est pas précisée par l'utilisateur (par exemple, Freeze dans P-II)

- Une autre différence est qu'en P-III, les nombres réels sont considérés avec une précision dépendante de la machine. En revanche, les nombres rationnels sont codés avec une précision parfaite (avec autant de chiffres que leur expression exacte le requiert).

En CLP, les réels sont considérés avec une précision infinie. Par conséquence, la résolution de certains problèmes la modélisation en physique est traitable alors qu'en P-III, il faut faire ça autrement :

CLP(R) : une implantation approximative de l'arithmétique des réels parfait
P-III : une implantation parfaite de l'arithmétique des réels approximatives

%%% **Fin de la partie double**

Méthodologie de programmation en CLP

- Schéma général Contraindre générer
- Méthodes de CSP - guide lines (livre Pascal, p 111)
- des méta outils : Minimize/Maximize/Min-Max ...
- First fail... (l'article de Haralick)
- Papier de Pascal

Techniques de Parcours de graphes dans les CSP

A recouper avec les transparents

- Les heuristiques sont associées aux procédures de recherches.
- Des techniques heuristiques intéressantes de recherche dans les problèmes de 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 pas s'inquiéter pour le passé

- Ces techniques remplacent le back-tracking habituel et sont plus efficaces.
- La classe de problèmes CSP concernée est composé des éléments suivants :
  - N variables (généralement appelé Unité)
  - Chaque variables a un domaine de M valeurs (généralement appelé Label)

  Le problème de CSP consiste à trouver toutes les assignations f de valeurs aux variables telles que toute paire variable/ valeur notée <var,f(var)> satisfasse les C°.

Définitions

- Plus formellement, si U est l'ensemble des variables et L l'ensemble de valeurs, la relation de contrainte binaire R est définie sur (U x L) et on a :

  $R \subseteq (U \times L) \times (U \times L)$.

Si une paire $(<u_1, l_1>,<u_2, l_2>) \in R$, alors on dit que les labels (valeurs) l_1 et l_2 sont consistantes et compatibles.

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

- La classe de problèmes concernés (en IA) sont diverses tels que la coloration de graphes, homorphisme et isomorphisme de graphes, planning, résolution booléenne, démonstration automatique de propositions...

- Pour illustrer le propos, prenons l'exemple de n-reines (N=6 ici). On veut placer N pions sur cet échiquier et deux pions ne doivent pas s'attaquer.

- Dans ce problème, les variables sont les lignes et les valeurs seront les colonnes. Par exemple, la paire (<1,A>, <2,D>) satisfont la Contrainte : ((<1,A>, <2,D>)∈ R.

Présentation informelle par un exemple

- Une trace de l'exécution par un mécanisme de retour arrière est le suivant :

L'entrée "2 A,B" indique que les valeurs A et B ont été infructueuses pour 2, mais C est une bonne valeur. Remarquons que cette partie de l'arbre de recherche ne donnera pas une solution finale.

- Notons que pour la variable 5, les valeurs A,C,E et F se trouvent deux fois dans la trace. Elles ont été testés et ont échouées pour les même raisons à chaque fois : 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.

- Notons que lorsque la variable 3 prend la valeur E, les seules valeurs possibles pour 4 et 6 sont incompatibles. La technique de **Looking_Ahead** permettra de détecter cet état (voir les algorithmes).

- **Forward-Checking** étant une version moins élaborée de Looking_Ahead, Elle ne permet pas de déceler cette situation. Cependant, la première fois que la variable 4 prendra la valeur B, il n'y a pas de toute évidence aucune possibilité pour la variable 6. Ainsi, la recherche pour 5 et 6 seront superflues et Forward-Checking permet de découvrir cette incompatibilité.

Présentation des méthodes

Appelons les *variables passées* celles qui ont déjà une valeur; *variables présentes* celle sans valeur en cours de traitement et *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 de s'assurer que :

1- chaque variable future a au moins une valeur avec les valeurs déjà prises par les variables passées et présente **(Futur x (Passé - Présent))**.
2- chaque variable future a au moins une valeur avec une des valeurs des autres variables futures (**Futur x 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 complet:

Le but de la *procédure LAC(U, F, T)* est d'affecter des valeurs consistantes aux variables représentées par U (et Ui) en respectant les C°. Ces affectations seront conservées et produites à la fin dans le tableau ***F*** *: tableau(1..Nombre_de_Var) de Label*.

Les évolutions des valeurs possibles pour les variables seront dans la matrice **T** de *Nombre_de_Var* lignes et *Nombre_de_Valeurs* colonnes de booléens. On peut considérer l'échiquier précédent comme un exemple de cette matrice. La présence du booléen vrai dans une case <l,c> de la matrice veut dire que la valeur représentée par la colonne <c> est consistante pour la variable représentée par la ligne <l>. Cette matrice est initialisée à vrai au départ.

Au fur et à mesure des calculs, de nouvelles versions de cette matrices seront créées à partir des précédentes représentant les valeurs encore consistantes pour les variables.

La fonction *relation(u1,l1, u2, l2)* vérifie, d'une manière générale que <u1,l1> et <u2,l2> sont compatibles. Ici, l'on vérifie que deux pions ne s'attaquent pas.

La fonction *Check_Forward(U, F1(U),T, New_T, Succès)* construit New_T à partir de T en ne conservant que les paires <future, valeur> consistantes avec le label actuel de U. S'il n'y a aucune paire de ce genre, Succès sera = faux et il faudra envisager une autre valeur pour U. La procédure *Look_future(U, New_T)* vérifie ensuite que chaque paire <future, label> de New_T est compatible avec a au moins un label pour toute autre variable et

supprime les futures qui ne sont pas consistante avec au moins une valeur pour tout autre variable (sauf pour la variable en cours).

Par exemple, pour la variable 1 placée, *Look_future* considère qu'il existe au moins une valeur pour les couples suivants :

U=1, U1 = 2..6, U2 = 2..6 sauf U1 en cours (i.e. si U1=2, U2=3..6)

C'est à dire, à la recherche d'au moins une valeur compatible, toutes les valeurs restantes pour la variable 2 sont considérées avec toutes valeurs restantes pour les variables 3..6. Si pour <U1,x>, aucune valeur pour U2 n'est trouvée alors, on supprime x du domaine de valeurs de U1 et on envisage un autre x pour U1. Les autres couples considérés par *Look_future* selon U sont :

U=2 U1 = 3..6, U2 = 3..6 sauf U1 en cours
U=3 U1 = 4..6, U2 = 4..6 sauf U1 en cours
U=4 U1 = 5..6, U2 = 5..6 sauf U1 en cours

Exemple pour N=6 :

- On place <1,A>. *Forward_Check* produit la matrice T selon la figure-1.
- On active *Look_Future*. <2,C> est placé et il y a au moins une case compatible avec <2,C> pour les autres variables. De plus, pour toutes les autres variables, il y au moins une case compatible les une avec les autres (figure-2).
- <2,C> étant validé, *Look_Future* envisage la variable 3. On place <3,E> qui laisse au moins une valeur aux autres. Mais lorsque *Look_Future* envisage B pour 4, il trouve une incompatibilité car <4,B> n'est pas compatible avec <6,D>; la seule choix possible laissée à 6 étant D dans ce cas. (figure 3).

<3,E> est supprimé de T et l'on revient sur le choix de <3,E>.

- <3,F> est placée (figure 4) et laisse au moins une case aux autres. Mais on constate immédiatement que la seule case pour 4, c'est à dire <4,B> ne sera pas compatible avec la seule case <5,B> restée possible pour 4. Il faut donc remettre en cause la choix de <2,C> et reprendre.

Procédure LAC(↓U, ↓F,↓T) =

```
Début
    Pour tout X élément de T(U)          -- i.e. les valeurs possibles de U
        F(U) := X;
        Si U < Nombre_de_Var Alors
            New_T := Check_Forward(U, F(U), T);
            Look_future(U, New_T);
            Si New_T ≠ VIDE Alors
                LAC(U+1, F, NEW_T);
            Fin Si;
        Sinon F est la solution;
        Fin Si;
    Fin Pour;
Fin LAC;
```

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

```
Début
    New_T := VIDE;
    Pour U2= U+1 .. ..Nombre_de_Var
        Pour tout L2 élément de T(U2)
        Si Relation(U,L,U2,L2) Alors
                Ajouter L2 dans New_T(U2);
            Fin Si;
        Fin Pour;
        Si New_T(U2) = VIDE alors Retourne(VIDE);     -- on abandonne
            Fin Si;                                   -- car pas de valeur
    consistante
    Fin Pour;
    Retourne(New_T);
Fin CHECK_FORWARD;
```

```
Procédure Look_Future(↓U, ↓↑T) =
Début
    Si U+1 ≥ Nombre_de_Var Alors Retourne; Fin Si;
    Pour U1 = U+1 .. Nombre_de_Var
        Pour tout L1 élément de T(U1)
            Pour U2 = U+1 .. Nombre_de_Var sauf U1
                Pour tout L2 élément de 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;
```

Looking-Ahead partiel (LAP)

- LAC ne mémorisera pas la plupart des résultats des tests effectués entre (Futur x Futur) pour les étapes suivantes; ceci demandant beaucoup d'espace.

- Une alternative sera de considérer la technique Looking-Ahead partiel qui teste et mémorise uniquement (Présent x Futur) et abandonne la moitié des tests sur les (Futur x Futur). Dans cet version, chaque variable future n'est testée qu'avec ses propres futures au lieu d'être testée les autres toutes les autres futures (pour elle et pour les autres). Ce qui veut dire que cette

version supprime moins de valeur de la liste des futures. Étant donnée que les test (future x future) détectent peu d'inconsistance pour justifier leur application, et du fait que l'on ne mémorise pas ces tests, LAP est plus efficace que LAC.

Cet algorithme sera identique aux précédent sauf pour la procédure Look_Future qui devra être remplacée par la procédure suivante (les différence sont signalées avec les caractères gras). On remplacera dans LAC les appels correspondants.

Ici, chaque future est testé avec ses propres futures. Par exemple, pour la variable 1 placée, *Look_future_Partiel* considère qu'il existe au moins une valeur pour les couples suivants :

U=1, U1 = 2..5, U2 = 3..6

Les autres couples considérés par *Look_future* selon U sont :

U=2 U1 = 3..5, U2 = 4..6 sauf U1 en cours

U=3 U1 = 4..5, U2 = 5..6 sauf U1 en cours

U=4 U1 = 5, U2 = 6 sauf U1 en cours

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 tout L1 élément de T(U1)
            Pour U2 = U1+1 .. Nombre_de_Var
                Pour tout L2 élément de 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)

Forward_Checking est une sorte de LAP des futures avec passé et présents. Les variables futures ne sont pas testées avec les futures. Forward_Checking commence dans un état où aucune variable future n'est en contradiction avec les variables passées. Forward_Checking vérifie à chaque pas qu'il n'y a pas de future incompatible avec le présent auquel cas il fera échouer la recherche aussi tôt. La matrice des variables x valeurs est au fur et à mesure dans les niveaux successifs. 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. FC n'effectue pas de test (Future x Future); ce qui été fait pas Look_Future dans LAC.

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. Cependant, il teste la variable présente seulement avec les variables passé et non pas avec les futures. Par 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.

Reprenons l'exemple précédent. Par cette méthode, on place <1,A> puis <2,A> qui s'avère être incompatible avec <1,A>. Il en est de même avec <2,B>. On place <2,C> et on vérifie avec <1,A>. Puis on place <3,A>...<3,D> qui seront rejetés car incompatibles avec les précédents. On place <3,E> puis <4,B> . On constate que 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.

En comparaison avec Forward_Checking, tout test fait par Back_Checking vers l'arrière (looking back) depuis une variable u vers une autre variable v, est également fait par la technique Forward_Checking. De plus, Forward_Checking aura effectué des tests entre u et les variable futures. Par conséquent, le Back_Checking fait moins de test de consistance; ce qui pourrait être un avantage. Mais le prix à payer est de faire plus de retour arrières en ayant un arbre aussi large que dans le cas de Forward_Checking. En résumé, 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 avec un supplément. Elle élimine des tests de consistance déjà effectués qui n'ont pas réussi et qui ne réussirons pas si on les refait. De même, elle élimine des tests de consistance déjà effectués qui ont réussi et qui réussirons encore si on les refait

Considérons la variable u. Soit v la plus petite variable à laquelle on est retourné lors de la dernière visite (couronnée par un échec) de u. Back_Marking mémorise ce v. Cela veut dire que nous avons épuisé toutes les valeurs de u en les testant avec 1..v; cela n'a pas donné de succès et nous avons dû revenir à v pour lui choisir une nouvelle valeur. On sait donc à ce stade que les valeurs actuelles des variables 1..v-1 ont déjà été testé avec toutes les valeurs possibles de u.

L'idée de Back_Marking est la suivante. Si on donne une autre valeur à v et on repart avec des valeurs pour u, ce n'est pas la peine de retester la nouvelle valeur de u avec celles de 1..v-1 car cela a déjà été fait; a réussit ou a échoué. Il suffit donc de tester, s'il le faut, cette nouvelle valeur de u avec la nouvelle valeur donnée à v.

Ainsi, si u=v, on procède par Back_Checking. Si u>v, on ne teste la nouvelle valeur de u avec celles de v..u-1 et on se rappelle de tests fait entre u les las valeurs 1..v-1.

Exemple.

On donne la valeur A à la variable 1.

On donne la valeur C à la variable 2. les tests sont fait par Back_Checking.

On donne la valeur E à la variable 3.

On donne la valeur B à la variable 4.

On donne la valeur D à la variable 5.

Aucune valeur possible pour 6, on remet en cause 5 puis 4 puis 3.

On donne F à 3. (3 est le v ci-dessus).

Maintenant, 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. Avoir remis en cause 4 veut dire qu'aucune des valeur de 4 n'a pu être compatible avec <1,A>, <2,C> et <3,E> car on les a toute testé.

Algorithme de BackMarking

Label : domain 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 ete modifiee

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

MARK(u,l) indique le plus bas niveau auquel un test a echoue lorque <u,l> du niveau actuel a ete teste contre les autres paires des niveaux inferieurs.

A tout point d'execution, si MARK(u,l) est < à NIVEAU_INF(u) alors on sait que <u,l> a deja ete teste contre les paires des niveaux inferieurs a NIVEAU_INF(u) et ces tests echoueront au niveau MARK(u,l), doc pas besoin de refaire.
Si mark(u,l) est plus grand que NIVEAU_INF(u) alors tous les tests reussiront sous le niveau (et egal) a NIVEAU_INF(u) et donc seuls les tests a partir de NIVEAU_INF(u) jqa niveau actuel devront etre refaits.

```
Procedure BACK_MARK (↓U , ↓↑F , ↓↑MARK, ↓↑NIVEAU_INF) =
Début
    Pour LAB dans LABEL boucle
        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 echec
                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'echec
            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.

Il faut noter que Back_tracking est très proche de Back_checking. Ce qui montre l'efficacité de Prolog par rapport aux langages de programmation avec C°.

Back_marking a tendance a trop consulter ses tables de mémorisation par rapport à la taille du problème (la valeur de N) alors que les meilleurs résultats sont obtenus par Forward_Cheking (400,000, pour N=10 et 600,000 pour Back_marking)

Des mesures statistiques conséquentes ainsi que l'analyse faite lors de l'exposé des algorithmes montrent que le meilleure méthode est la méthode Forward_checking. Une implantation avec tableaux de bit sera encore plus efficace. Forward_checking a été implanté dans CHIP.

Principe de First-Fail

Une des stratégies employées dans les problèmes CSP. Looking_Ahead et Forward_cheking emploient cette technique pour élaguer rapidement des arbres de recherche. Il existe d'autres méthodes sur le même principe qui consistent à essayer d'échouer le plutôt possible en mémorisant la situation d'échec pour ne pas avoir à la recommencer. D'autres applications de cette stratégie sont les suivantes :

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) 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 de contrainte imposée à une variable par la valeur de la variable K à la variable K+1. Soit P(k) la probabilité pour une valeur L de la variable K à être consistante avec une valeur pour K+1. Si les test de consistance sont indépendants, la probabilité d'avoir un succès sur les tests pour les variables 1..K est $P=\prod_{k=1,K} P(K)$.

Il faut donc maximiser cette valeur, soit minimiser $\prod_{i=1,n} 1-P(Ki)$, Ki le ième variable. Par des calculs plus ou moins complexes, on obtient un nombre de test C à effectuer :

$C= 1+\sum_{i=1,k-1}\prod_{j=1,i} P(K_j)$.

Minimiser cette valeur revient à avoir une permutation de $K_1..K_n$ telle que $P(K_1) \leq P(K_2) \leq ... \leq P(K_n)$.

Ce qui veut dire que pour minimiser C, il faut effectuer d'abord les tests susceptibles d'échouer.

Pour le cas de N-reines par exemple, on peut effectuer les tests de consistance entre la valeur de la variable K_i et les valeurs des variables $K_{i-1}..K_1$ dans un ordre de contrainte décroissant. Puisque K_{i-1} est la variable qui contraint le plus la variable K_i, on effectuera les tests dans l'ordre décroissant des indices. 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. Il est évident que 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. Notons que ce choix est local (au niveau d'une branche). La preuve mathématique pour l'optimum local est laissé aux élèves.

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-checcking 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 Forward-Check

On installe le domaine puis on énumère une valeur. On élimine ensuite des valeurs "chez" les 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-Check sur HP :
première réponses 460 ms et
dernière réponses 174940

- L'apporte de Forward-Check 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 vue 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 sont Np-Complet.
  On considère ici des problème non triviaux (plus de 100 variables) pour lesquels les techniques d'optimisation sont plus "payantes".

Branch & Bound

- 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), tous les problèmes 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 Branc & Bound classique si dans l'arbre de recherche, certaines branches ont un coût nul (traitement n'entraînant pas de modificat du coup à minimiser) ou bien lorsque le coût trouvé doit être modulé.

- Contrôle possible dans les points de choix ou au niveau supérieur.

- Maximize et Minimize de PrologIII sont des branch & bound "locaux".

Exemples

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

Coloration de 5 régions : branch & bound dans sa 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>
```

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 nxm booléens. Dans l'exemple de n-reines, on peut prendre nxn 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.

Intégration des techniques CSP en P.L.

Remarque générales sur la méthodologie :

On peut avoir deux approches :

- écrire des programmes logiques qui implantent le Forward Checking
- Etendre le langage en codant la technique dans la procédure de recherche.

- La première approche conduit à une déclarativité moindre et à des programmes peu efficaces que la second.

- Il semble évident que la seconde approche est souhaitable lors du traitement des proclèmes CSP.
  ==> les programmes logiques peuvent utiliser le forward checking sans perdre de leur déclarativité et de leur pouvoir d'expression.

- Dans ce cas, le programmeur écrit du générer-tester alors que l'interpreteur/compilateur utilise le forward cheking pour résoudre le CSP. Ceci conduit à une efficacité substentielle et rend la programmation logique un outil efficace et puissante pour resoudre ls problème CSP.

- Les techniques telles que Looking-Ahead et Forward-Checking ont été implantées dans le langage CHIP. Ces techniques exploitent activement les contraintes pour élaguer l'arbre de recherche a priori en supprimant les valeurs incompatibles.

- CHIP implante les techniques de parcours de graphes et d'autres méthodes de recherche pour résoudre les problèmes de CSP qui sont basées sur l'utilisation active des contraintes pour réduire l'espace de recherche **a priori** (avant la détection de l'échec).
- CHIP implante d'autres outils tel que les prédéfinis
  Minimize/Maximize/Min-Max ...
  Element, Atmost,...
  La plupart de ces prédéfinis sont prgrammables en PrologIII.

Définitions

- Il est souvent le cas que les variables prennenet leur valeurs dans un domain fini. Cette information ne peut pas être explicitée en programmation logique. La déclaration de domaine est introduite ici pour permettre de limiter les valeurs des variables. Une déclaration de domaine est de la forme :

  domaine p(a_1,...,a_n). où les a_i, $1 \leq i \leq n$ sont
  - soit un objet de l'U_H
  - soit un objet d'un domaine D

Dans ce dernier cas, le domaine est fini et explicité par un ensemble de valeurs. Nous allons considérer en particulier les domaines des entiers et des chaînes.

- La déclaration de domaine peut être considérée comme une sorte de méta-connaissance. L'effet d'une telle déclaration est de réduire la classe des interprétations qui doit être considérée pour un programme logique. Dans ce cas, la sémantique déclarative est une forme particulière de la logique multi-sorte ou encore d'une logique à une-sorte si on considère les variables d'agrégats.

- La résolution est étendue pour tenir compte des domaines de variables. Une variable avec un domaine défini est appélée une **d-var** dont le domaine est D_X. Les autres variables seront notées **h-var**. Pour tenir compte de ces deux sortes de variables, on étend l'unification aux cas des variables :
  - si une h-var et une d-var doivent être unifiées, la h-var sera liée à la d-var.
  - si une constante et une d-var doivent être unifiées : la d-var est liée à la constante si elle est dans le domaine sinon, échec
  - Si deux d-var sont à unifier : soit Dz l'intersection des deux domaines. si Dz est vide alors échec. si Dz={v} alors v est liée aux deux variables

sinon, les deux variabless sont liées à une nouvelle variables Z dont le domaine est Dz.

Les techniques de résolution

Forward Cheking en Programmation Logique

- Le Forward-Checking permet d'élaguer l'espace de recherche **a priori** contrairement aux méthodes **a posteriorie** employé en programmation logique.
- Cette extension permet un gain substentiel et une efficacite importante dans la résolution d'un classe de problème et permet à la prog° logique d'être un outil adapté à la résolution des problème en CSP.

Le forward checking (FC)

- Elle utilise une contrainte dès qu'elle contient une variable pour réduire les valeurs possibles qui peuvent lui être assignées.

- Forward-checking est considéré comme l'une des meilleurs procedures pour résoudre les CSP. intuitivement, une contraintes peut être utilisée dès qu'elle contient au plus une var. Dans ce cas, les valeurs possibles de cette var sont réduites aux valeurs qui satisfont les contraintes sur cette variable.
- Ainsi, un programme basé sur le FC donne des valeurs aux variables, (ce qui n'est pas possible en programmation logique où le U_H est infini), utilise toutes les contraintes contenant au plus une variable, choisit une valeur pour la prochaine variable, utilise des constraintes....
- Si pendant la recherche, une contraintes ne peut pas être satisfaite, (l'ensemble de valeurs réduit à {}), la procédure de recherche donne une autre valeur aux variables précédentes et recommence.
- Pour mettre en place le FC, on introduit le forward déclaration (FD) quelque soit le type des contraintes. La seule condition est que la contraintes doit être une procédure de décision (i.e. étant donnée les valeurs pour les variables, il doit être de déterminer si la C se vérifie ou non).

Exemple simple :

Soit le prédicat

$X \neq Y$:- not(X=Y).

- En Prolog classique, on utilise la négation par échec et donc une utilisation passive des valeurs de X et de Y.
- Une règle juste de négation ne doit faire le test que si X et Y ont une valeur (par exemple *dif* de PII).
- On définit donc X≠Y par :
  - Si X est une d-var et Y une constante alors soit $D_Z = D_X \backslash \{Y\}$.
    Si Dz= {v} alors X est unifié avec v et X≠Y réussit;
    Sinon, X est lié à Z (dont le domaine est Dz) et X≠Y réussit.
  - si X et Y sont constante alors $X \neq Y$ réussit si X est différent de Y.
  - notons qu'aucune décision ne peut être prise si X et Y sont tous deux variables ??

- La contraintes ≠ est traitée quand l'une des vriables est une constante et l'autre une d-variable. Dans ce cas, on peut implanté efficacement ≠ par délai.
- Dans les deux premiers cas, la ≠ est utilisé comme une règle FC en réduisant les valeur des d-vars. Donc la contraintes est utilisée activement et de façon a priori pour élguer. On a un gain substentiel lorsque l'on combine ce traitement avec une règle de calcul qui traite les ≠ dès que possible. Des mesures montrent une facteur de 360 fois plus rapide qu'un programmation utilisant le mécanisme de coroutinage.

- Ainsi, les programmes basés sur les techniques classiques de coroutinage ou sur le schéma *génrer-tester* peuvent utiliser la stratégie FC. Ce mécanisme général peut être spécialisé pour des contraintes qui sont souvent utilisées et peut être implanté de façon plus efficace si l'on tient compte des propriétés particulières telles que les contraintes de non-égalité. Ainsi, utilisée avec le mécanisme de délai, FD améliore la recherche FC en permettant la propagation des contraintes qui contribuent à la réduction de l'espace de recherche et à une détection rapide des échecs.

- Une déclaration Forward pour un prédicat P d'arité n est une expression unique pour le prédicat P :
  forward p(a_1,...,a_n) où a_i est un **c** (clos) ou un **d** (d-variable). Cette déclaration spécifie la forme des arguments attendus pour les contrainte. Elle définit une précondition pur l'utilisation de cette contrainte. Le sens de cette déclaration est :

  1• Un littéral *p(x_1,...,x_n)* dans le résolvant avec une FD est "Forward-Checkable" si et seulement si tous les arguments corresponant aux 'c' dans la FD sont clos et, exactement un argument correspondant à un 'd' est une d-variable; les autres étant closes. Dans ce cas, la d-variable et appelée la *variable forward*.
  (i.e. toutes les c sont closes, une seule des d est encore variable)

  2• Une règle de calcul est sauve par rapport à la FD si un littéral *p(x_1,...x_n)* dans le résolvant avec une FD est choisi si une des conditions suivantes est vérifiée :
  - tous les arguments sont clos
  - p(X_1,...,X_n) est Forward-ckeckable.

  Ainsi, on choisit un littéral s'il a fait l'objet d'une déclaration FD et si ses arguments sont soit tous clos soit, avec au plus une d-variable.

- Lorsqu'un prédicat avec une FD est choisi par la règle de calcul, deux cas peuvent se présenter :
  - si tous les arguments sont clos : on procède à la résolution
  - si le prédicat est Forward-Checkable, une autre règle d'inférence est choisie.

  Cette règle est la suivante :
  - soit G_i un but de la forme
    <-A_1,...,A_{m-1}, p(X_1,...,X_n), A_{m+1}, ... A_k.

  R est la règle de calcul sauve par rapport à FD, PR un programme logique et p(X_1,...,X_n) un prédicat avec une FD.

On dérive G_{i+1} de Gi en utilisant un mgu θ_{i+1} via PR si les conditions suivantes sont vérifiées :

1- $p(X_1,...,X_n)$ est choisi par R
2 -$p(X_1,...,X_n)$ est forward-checkable et X_j est une (la) Forward-variable
3- $D_Z = \{\mathbf{v} \in D_{Xj} \mid$ il existe une SLD-réfutation de $\{P \cup \{<\text{-}p(X_1,...,X_{j-1}, \mathbf{v}, X_{j+1},...,X_n)\}\}$ et $D_Z \neq \{\}$
4- θ_{i+1} $= \{(x_j / e)\}$ si $D_Z = \{e\}$
$= \{(x_j / Z)\}$ si $D_Z = \{e_1...e_k\}$ k>1 où Z est une nouvelle variable du domaine D_Z
5- G_{i+1} est le but (<- $A_1,...,A_{m-1}, A_{m+1}, ... A_k$) θ_{i+1} .

- Intuitivemt, l'ensemble possible des forward-variables est réduit à l'élément qui satisfait la constraint. On remarque que $p(X_1,...,X_n)$ est sortie du résolvant même si Dz possède plusieurs valeurs. La génération de différente valeurs {e1...ek} est donc effectuée hors résolution même si l'on retourne en arrière. Ce retour arrière est fait sur le générateur de valeur plutôt que sur la résolution.
- Ainsi, étant donné qu'une forward-variable peut seulement être instanciée à un élément de D_{xj}, la sémantique procédurale est équivalente à la sémantique déclarartive : quelque soit la valeur prise par la variable Z, elle sera dans D_{xj} et satisfera $p(X_1,...,X_{j-1}, \mathbf{v}, X_{j+1},...,X_n)$.
Ainsi, si une seule valeur satisfait la contrainte, elle peut ^être assignée à la forward-variable. Finallement, si aucune valeur du domaine de X_j satisfait la contrainte, il n'y a pas de moyen de la satisfaire.
Ainsi, la contrainte est résolue **une fois pour toute** et a réduit le domaine de X_j. Le langage sera "complet" si les générateurs sont fournis tel que les prédicats soumis à une FD deviennent soit clos soit forward-checkables dans une des étapes de la résolution.

- Ainsi, ce mécaisme général utilise les contraintes d'une manière active de deux manières:
  - il peut supprimer une ou plusieurs valeurs de l'ensemble pour les d-variable.

- il peut assigner une valeur lorsque une seule valeur consistante reste pour cete variable.

La règle de calcul

- Avoir une règle de calcul sauve vis a vis des FD n'est pas strictement nécessaire. En effet, le programmeur peut s'assurer qu'un littéral est forward-checkable ou clos chaque fois qu'il est choisi pour la résolution. Cette responsabilité et du même type que l'utilisation de la négation sure ou les prédéfinis en Prolog. Cependant, il y a des raisons d'avoir une règle sauve. Dans ce cas, les FD agissent comme des préconditions de sélection des prédicats. Ainsi, les contraintes (sous forme de prédicats peuvent être exprimées avant les générateurs et peuvent être choisis dès que possible. Ceci introduit un calcul orienté données.

- Dans la technique classique de FC, on s'assure qu'à chaque étape de calcul, toutes les d-variable non-instanciées ont au moins une valeur dans leur ensembles possibles de valeurs respectifs. Cependant, losrqu'il y a une seule valeur, il est claire que cette valeur peut être assignée. Ainsi, les nouvelles contraintes peuvent devenir FC et l'espace de recherche sera réduit d'une manière "a priori" et peut conduire à la détection d'échecs. Dans la technique classique de FC, cet échec peut être découvert avant quelques générations inutiles de valeurs. Il faut noter que cette généralisation ne produit aucun surcharge et ne demande pas d'effort de la part du programmeur. C'est en fait la règle de calcul sauve responsable de choisir de nouveaux prédicats "forward-checkable". On conclut que les FD ne sont pas nécessaires et peuvent être généralisés.

- Il est important de noter que l'expression des FD peut être généralisée (implicite) et il ne sera plus nécessaires des les préciser. Ainsi, l'expression des programmes devient plus simple en spécifiant **d'abord** les contraintes et en laissant l'interpreteur les choisir lors des étapes de calcul.
Ainsi, étant donnée un programme selon le schéma **génére-tester**, la présence des informations de coroutinage conduit à une approch coroutinage alors que les FD conduisent à un forward-cheching.

Exemples

- On illustre l'utilsation de forward checking en programmation logique par 3 exemples :
  - le problème de N-reines
  - le puzzle de 5 maisons
  - le problème des mots croisés

Le problème des n-reines

- Dans une approche génére/tester en Prolog, un programme classique pour résoudre ce même problème (avec N=5) consiste à permuter une liste [1,2,3,4,5] (génération) et vérifier si une de ces permutations est une solution (test). Cette approche est très inéfficace. On peut alors envisager une méthode constructive.

- Dans l'approche CLP, les mécanismes de contrôle basés sur les information de contrôle (constraintes) fournies par l'utilisateur peuvent être utilisées dès que possible et ainsi améliorer l'efficacité de la recherche. L'approche coroutinage utilise ces informations pour améliore un programme générer/tester.

- X_i représentent les numéros de lignes des pions.

```
queens([]).
queens([X|Y]) :-
        indomain(X),
        safe(X,Y,1),
        queens(Y).

safe(_,[],_).
safe(X, [F|T], Nb) :-
        noattack(X,F,Nb),
        Newnb is Nb +1,
        safe(X,T,Newnb).
```

```
noattack(X,Y,Nb) :-
        Y≠X Y ≠ X-Nb, Y ≠ X + Nb.
```

- On peut utiliser queens(X) de différentes façons :
  - Si X est une liste d'entiers, on vérifie si elle constitue une solution.
  - Si X est une liste de d-variables, on génère une solution. On peut donc, en fixant n=5, définir :

```
domain cinq_reines({1,2,3,4,5}).
cinq_reines([X1,X2,X3,X4,X5]) :- queens([X1,X2,X3,X4,X5]).
```

- Ici, avec la déclaration **domaine**, le prédicat **indomaine** permet de choisir une valeur pour X_1; étant donné cette valeur, toutes les valeurs inconsistates pour $X_2..X_5$ sont écartées par le prédicat **safe**. Egalement, **noattack** supprime X, X-Nb et X+Nb du domaine de Y.La situation après un et deux choix sont données par (on place colonne par colonne) (une erreure dans fig1):

- Dans la figure à gauche, après le placement de X_1, les cases marquées par 'x' sont les seules à pouvoir être examinées (les domaines des variables ont été réduits). Pour placer le deuxième pion, on va donc considérer uniquement ces cases. Dans la figure à droite, le placement de deux pions réduit largement les cases disponibles. Ainsi, il n'existe qu'une seule valeur possible pour X_3; cette valeur est donnée à X3. X4 prend également sa seule valeur possible, ce qui ne laisse à X5 qu'une seule valeur. La solution est ainsi trouvée avec deux choix (X1, X2) et **sans retour-arrière**.
- Considérons le retour arrière pour une autre solution. Le premier point de choix est X_2 et les valeurs de X_1 ne sera plus comparée avec X_2, X_3, X_4 et X_5. Alors que dans une méthode classique (par exemple coroutinage), à

chaque fois qu'on donne une valeur à X_2, X_3, X_4 et X_5, cette valeur est comparée avec X_1. Ce qui introduit beaucoupe de redondance. Ce traitement accélère grandement l'efficacitéet prévoit "a priori" ces échecs.

- Considérons maintenant une autre question avec N=8.

- X_6 a déjà reçu une valeur (4) car celle-ci est la seule qui reste dans son domaine. Ce qui écarte la valeur 2 pour X_4 (détection d'échec par le prédicat *safe*). De plus, la vraie raison de cet échec (X4) est déterminée. Dans une méthode coroutinage, cet échec est détectée seulement quand une valeur est donnée à X_6 et les retours arrières réexaminent toutes les valeurs (de 1 à 8) pour X_5 et X_6. Ceci montre clairement comment FC peut détecter les échec d'avance et choisir le bon point de choix.

Généralisation de FC

- La solution précédente était une solution *générer-tester* où la génération est faite par *indomain* et les tests sous forme de contraintes par le prédicat safe.

- On peut écrire un programme qui implante la démarche *contraindre-générer* afin de conduire à une propagation plus importante des contraintes .

- Dans cette démarche, on dispose d'un prédicat qui teste si une configuration donnée est une solution (tests sous forme de contraintes). Un schéma possible est de proposer test + permutation.

- Pour l'exemple précédent, on peut donc écrire un prédicat *test(L)* qui vérifie si la liste L=[X_1,...,X_n] est une solution au problème.

```
queens([]).
queens(L) :-
        test(L), generer(L).

test([]).
test([X|Xs]) :-
        safe(X,Xs,1) , test(Xs).

safe(X,[],_).
safe(X,[Y|L],Nn) :-
        Y≠X Y ≠ X-Nb, Y ≠ X + Nb.

generer([]).
generer([X|Y]) :-
        indomaine(X), generer(Y).
```

Une spécialisation pour N= 5 serait :

```
domain cinq_reines({1,2,3,4,5}).
cinq_reines([X1,X2,X3,X4,X5]) :- queens([X1,X2,X3,X4,X5]).
```

- Ainsi, l'ensemble des contraites est d'abord installé puis, on procède à la génération des valeurs. On note que lors d'installation des contraintes, aucune valeur n'intervient et c'est uniquement lors des appels à **indomaine(X)** que ces valeurs sont engendrées. Rappelons que dans la solution précédente, la génération des valeurs avait lieu au début par l'appel à indomaine dans queens.

- Reprenons la figure de l'échiquier pour N=8. Après 3 choix pour X_1, X_2 et X_3, X_6 reçoit la valeur 4. Tous les prédicats qui incluent cette variables sont Forward-checkables. Ce qui permet d'assigner 7 à X_8 (sa seule possibilité) qui, à son tour assigne 2 à X_7. A ce moment, X_4 et X_5 ne

peuvent prendre que la valeur 8, ce qui conduit à une contradiction Ainsi, l'échec est détecté plutot sans aucune génération de valeurs pour X_4 et X_5.

La technique Looking-Ahead

- La technique Looking-Ahead est différente de Forward-Cheching dans la mesure où les contraintes peuvent être utilisées si plusieurs d-variables sont restées sans valeur.
  Le domaine de ces variables est réduit mais les contraintes ne sont pas forcément satisfaites et doivent être reconsidérées plus tard.

- Pour expliquer la technique, prenons un exemple d'équation et d'inéquation linéaire sur les entiers naturels (>0) où on peut raisonner sur les variations des intervalles des valeurs des variables.

- Soit une contraintes d'égalité :
  $a_1X_1,...a_nX_n+c_1 = b_1Y_1+...+b_mY_m+c_2$.

  où X_i Y_j sont des d-variables avec les domaines des entiers positifs et a_i, b_j, c_1, c_2 des entiers positifs.

  Soit $a_1X_1,...a_nX_n+c_1$ variant dans l'intervalle $[min_1,max_1]$ et
  $b_1Y_1+...+b_mY_m+c_2$ variant dans l'intervalle $[min_2,max_2]$.

  Pour satisfaire la contrainte d'égalité, les deux expressions doivent varier [min,max] où min est le <u>maximum</u> de min_1 et min_2 et max est le <u>minimum</u> de max_1 et de max_2. Ce qui introduit les contraintes :

  $a_1X_1,...a_nX_n+c_1$ $\geq$min si $min_1 <$ min
  $b_1Y_1+...+b_mY_m+c_2$ $\leq$ max si max1 > max.

  et de même pour les autres termes.

- En raisonnant de façon analogue sur les variables, on produit de nouvelles contraintes qui réduisent largement les domaines de celles-ci. Les contraintes ne seront pas satisfaites immédiatement (il peut y avoir des

combinaisons inconsistantes) mais elle sera conservée dans le résolvant pour être reconsidérée si plus d'information devient disponible. Par exemple, la suppression ou l'instanciation de valeur de certaines variables réactivera la contrainte.

Ceci est un exemple du paradigme programmation par données (data-driven) où ce sont les données qui dirigenet les calculs.

Le principe first-fail

Le FC est généralement amélioreé par ce que l'on appelle le principe de "fist-fail". Ce principe consiste à dire : "pour réussir, essayer d'abord les choix qui sont susceptibles d'échoer". La motivation de ce principe est que, par ce moyen, les échecs sont découverts plutot et le nombre des retours-arrières sont ainsi réduits. Une application simple de ce principe consiste à choisir à instancier la prochaine variable qui a le plus petit domain. Dans CHIP, un prédicat **label(X)** est prévu qui diffère de labelling (le prédicat générer ci-dessus) en ce sens qu'il choisit un ordre dynamique d'instation des variables.

p La solution suivante du même problème en CLP(R) montre quelques détails importants.

Remarque : CLP(R) n'a pas de contrainte ≠ et il faut utiliser < ou >, ce qui rend la résolution moins efficace)

1- Version avec seule le génération d'une valeur pour X à chauque appel de queens. Par exemple, pour la question queens([X1,X2,X3,X4]), X1 prend une première valeur et les autres variables (X2..X4) prennenet la leur dans les appels successifs à *queens*.

Au retour du premier appel de *safe*, toutes les contraintes (exprimées dans noattack) entre X1 et les autres (X2..X4) sont installées. Au second appel, X2 prend une valeur, si qui permet de décider des contraintes entre X1 et X2, les autres doivenet attendre ....

queens([]).

```
queens([X|Y]) :-
        indom(X),
        safe(X,Y,1),
        queens(Y).
```

2- Version **générer/tester** avec la génération de evaleurs pour toutes les variables. Pour la question queens1([X1,X2,X3,X4]), X1,..,X4 prennent leur valeurs et safe peut éjà metrre X1 en accord avec X2..X4. Les autres appeles à queens1 feront de même entre X2 et (X3,X4) puis X3 et X4.

```
queens1([]).
queens1([X|Y]) :-
        indomaine([X|Y]),
        safe(X,Y,1),
        queens1(Y).
```

Notons qu'à partir du premier appel à queens1 qui donne des valeurs à X1..X4, les autres appels (inutiles) à indomaine seront des tests. On peut donc écrire :

```
queens1bis(L) :-
        indomaine(L), % la génération
        queens1(L).          % les contraintes
```

Cette version est plus efficace que la précédente car on peut décider immédiatement des contraintes (puisque les variables ont des valeurs) et en cas d'échec, générer d'autres valeurs pour les variables.

Remarquons enfin que indomaine ne peut générer de nouvelle valeurs que si l'on retourne en arrière et l'on le réexécute. En revanche, le prédicat indomaine de CHIP installe le domaine de valeur d'une variable comme une contrainte qui échape aux retours arrières.

3- Version avec aucune génération de valeur.

```
queens2([]).
queens2([X|Y]) :-
```

```
    safe(X,Y,1),
    queens2(Y).
```

Pour la question queens1([X1,X2,X3,X4]), le système répond par un système de contrainte sous la forme simplifiée :

X1 < X2 + 1
X1 < X3 + 2
X1 < X4 + 3
X4 < X1
X3 < X1
X2 < X3 + 1
X2 < X1
X2 < X4 + 2
X4 < X2
X3 < X2
X3 < X4 + 1
X4 < X3

Notons que par l'absence de '<>', le prédicat ne installe des contraintes '<' puis '>'. Ces deux possibilités augmentent le nombre de réponses donn"es sous forme de contraintes . Il faut notre qu'il y a **plusieurs niveaux de retours arrières** dans cette solution :

- Les retours arrières classiques de Prolog
- Les retours arrières dans les systèmes de contraintes déjà installé
- *Les retours arrières provoqués par le prédicat ne (qui rentre en fait dans le cadre du premier cas) du à l'absence de ≠.*

4- Version **contraindre-générer** avec la génération de valeurs après l'installation des contraintes . Cette solution montre le problème de l'absence de ≠ car en fait, pour chacun des solutions de la version 3 précédent, les valeurs générées sont testées. Lorsque l'on aura épuisée tous les points de choix sur le prédicat indom, on reviendra sur ne. Ce qui montre l'inefficacité des trois niveaux de contraintes vus dans la version 3.

```
queens3(L) :-
        queens3bis(L),        % les contraintes
         indomaine(L).% la génération
queens3bis([]).
queens3bis([X|Y]) :-
        safe(X,Y,1),
        queens3bis(Y).
```

Les prédicats inchangés pour les quatre versions sont :

```
safe(_,[],_).
safe(X, [F|T], Nb) :-
        noattack(X,F,Nb),
        safe(X,T,Nb+1).

noattack(X,Y,Nb) :-
       ne(X,Y), ne(Y, X-Nb), ne(Y, X+Nb).
ne(X,Y) :- X > Y.
ne(X,Y) :- X < Y.
indomaine([]).
indomaine([X|Y]) :- indom(X), indomaine(Y).
indom(1).
indom(2).
indom(3).
indom(4).
```

D'autres exemples

Le problème Criyparithmétique

L'exemple fameux "SEND+MORE=MONEY" consiste à donner comme valeur un chiffre 0..9 tous différent aux variable (lettres) SENDMORY pour que l'addition se réalise.

```
domaine solve(0..9, 0..1).
solve([S, E, N, D, M, O, R, Y], [C1,C2,C3,C4]) :-
        tous_differents([S, E, N, D, M, O, R, Y]),
        S ~= 0, M ~= 0,
        C1 = M,
        C2 + S + M = O + C1 * 10,
        C3 + E + O = N + 10 * C2,
        C4 + N + R = E + 10 * C3,
        D + E = Y + 10*C4,
        generer([C1,C2,C3,C4], [0,1]),
        generer([S, E, N, D, M, O, R, Y],[0,1,2,3,4,5,6,7,8,9]).

tous_differents([]) .
tous_differents([X|L]) :- hors_de(X,L) tous_differents(L).
hors_de(X,[]).
hors_de(X,[Y|L]) :- X ~= Y, hors_de(X,L) .

generer([],_).
generer([X|Xs], L) :-
        membre(X,L), generer(Xs,L).
```

On constate que ce programme contient deux sorte d'information. La contraintes d''inégalité (~=) exprimée par tous_differents et deux autres contraintes individuelles. Ces contraintes sont choisies par la règle de calcul seulement quand une seule d-var est non instanciée.

Puisque M ~=0 et C1=M, on déduit immédiatement que M=C1=1. Ceci est fait car M=1..9, C1=0..1, M ~=0 et C1=M.

Toutes les inéquation faisant intervenir M sont maintenant considérées et 1 est supprimé de leur domaine. Les contraintes sont ensuite suspendues.

Considérons C2 + S + M = O + C1 * 10. puisque C1=M=1, on obtient C2 + S + 1 = O + 10.

C2+S+1 varie sur [1,11] et O+10 sur [10,19]. On en déduit que C2+S+1 >9 réduisant le domaine de S à [8,9]. Aussi on conclut que O+10 <12 réduisant le domaine de O à [0,1]. Mais sachant que O doit être différent de 1, on donne 0 à O.

A ce stade, il est possible de donner une val à C2 et S tq la contraintes ne soit pas satisfaite. L'espace de recherche a été grandement réduit. Lorsq'on cosidère

C3 + E + O = N + 10 * C2,

on trouvera que C2=0 car C3+E varie sur [2,10] et N + 10 * C2 sur [2,19]. Le programme en déduit que N + 10 * C2 doit être < 11; ce qui implique C2=0.

La CONTRAINTES_ préc est reconsidéré conduisant à affecter 9 à S. En poursuibant le même raisonnement, le programme produira un gain substentiel par rapport par exemple aux technique générer/tester avec ou sans coroutinage (le gain est de l'ordre de 30).

On peut modifier le prédicat principal par :

```
domaine solve(0..9).
solve([S, E, N, D, M, O, R, Y]) :-
        tous_differents([S, E, N, D, M, O, R, Y]),
        S ~= 0, M ~= 0,
        1000*S+100*E*10*N+D+1000*M+100*O+10*R+E=
        10000*M+1000*O+100*N+10*E+Y,
        generer([S, E, N, D, M, O, R, Y],[0,1,2,3,4,5,6,7,8,9]).
```

Dans cette version, avant de commencer l'énumération, et uniquement par la manipulation des contraintes et la propagation, on réduit l'espace en déduisant

S=9, M=1, O=0, E,N=4..8, R,D,Y=2..8.

et avec un seul retour arrière, le programme trouve directement une solution.

Le problème d'approvisionnement

Il s'agit de trouver des locations pour des entrepots à partir desquels on fournit des cleints. On considère ici 3 entrepots noté par E1,E2 et E3. Si la valeur de Ei = 0 alors cet entrepot ne sera paas dans la config finale. On sait de plus que selon la situation géographique :

. Client1 peut être fourni par E1, E2
. Client2 peut être fourni par E1, E3
. Client3 peut être fourni par E2, E3
. Client4 peut être fourni par E3
. Client5 peut être fourni par E1, E3

On associe à chaque client Ci deux variables Ci1 et Ci2 . Ci1 = 1 si le client est fourni par l'entrepot 1 et 0 sinon. De même pour Ci2.

Chaque entrepôt a un coût fixe d'ouverture et 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.

Le problème est de trouver une config° telle que le cout total soit inférieur ou égal à une limite (ici 60).

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 total de moins de 60.

Le programme suivant :

```
approvisionnement(Ts, Ls, Cout, Limite):-
    Cout <= Limite,
    configuration(Ts, Ls, Cout).

configuration([T11, T12, T21, T23, T33, T33, T43, T51, T53],
                        [L1, L2, L3], Cout) :-
```

```
        T11 + T12 = 1,              % l'un ou l'autre des entrepots pour le client1
        T21 + T23 = 1,
        T32 + T33 = 1,
        T43 = 1,                    % client4 est fourni par E3
        T51 + T53 = 1,
        T11 + T21 + T51 <= 3 * L1,  % voir ci-dessous
        T12 + T32 <= 2 * L2,
        T23 + T33 + T43 + T53 <= 4 * L3,
        Cout = 5 * T11 + 7 * T12 + 4 * T21 + T23 + 2 * T32 + 5 * T33 +
                    4 * T43 +3 * T51 + 8 * T53 + 18 * L1 + 10 * L2 + 28 * L3,
        genere([T11, T12, T21, T23, T33, T33, T43, T51, T53, L1, L2, L3]).

genere([]).                     % simplifié pour cet exemple car on sait 0,1.
genere([1 | Xs]):-
        genere(Xs).
genere([0 | Xs]):-
        genere(Xs).
```

Les cinq premières contraintes expriment le fait qu'un client doit être associé à un seul entrepot; les trois autres contraintes évitent qu'un client soit associé à un entrepot ne figurant pas dans la config finale.

Une question posée est

```
?- approvisionnement(A,B,C,60).
```

et on obtient :

```
B = [0, 1, 1]
A = [0, 1, 0, 1, 0, 0, 1, 0, 1]
C = 60
```

Le premier résultat obtenu par le programme est que E3 doit être = 1 (pour le client 4); le troisième entrepot est choisi. Puis on essaie d'affecter le premier client au premier entrepot; T11=1. Ceci mplique que le cout devient :

$4 * T21 + T23 + 2 * T32 + 5 * T33 + 3 * T51 + 8 * T53 + 10 * L2 \leq 5.$

Ce qui conduit à affecter 0 à E2. Mais cela conduit à T12+T32=0 qui est en contradiction avec le rste acr T33 doit être=1 et donc T51 et T53 doivent être =0.

T11 est donc assigné à 0; ce qui donne T12=1 par la propagation des contraintes . Mais maintenant, le cout est

$4 * T21 + T23 + 2 * T32 + 5 * T33 + 3 * T51 + 8 * T53 + 18 * L1 \leq 11$.

Ce qui conduira à L1=0 et donc T23=T53=1. Maintenant, le cout est de 58 laissant E2 pour le client 3 : T3=1.

Cet exemple montre la propagation de contraintes dans le système CLP.

Le problème de 5 maisons

Cinq hommes de natianalités différentes vivent dans les 5 premières maisons d'une rue. Ils ont 5 professions différentes, chacun d'eux a un animal favorit, une boisson favorite. Les animaux et les boissons sont tous différents. Les 5 maisons sont peintes de 5 couleurs différentes. Les faits suivants sont données :

1- l'anglais vit dans une maison rouge
2- l'espagnole a un chien
3- le japonais est peintre
4- L'italien boit du thé
5- Le norvégien vit dans la 1ère maison à gauche
6- le propriétaire de la maison verte boit du café
7- La maison verte est à droite de la maison blanche
8- Le sculpteur élève des escargots
9- Le diplomate vit dans la maison jaune
10- On boit du lait dans la maison du milieu
11- La maison de norvégien est à coté de la maison bleue
12- Le violoniste boit du jus de fruits
13- Le renard est dans la maison à coté de celle du medecin
14- Le cheval est dans la maison à coté de celle du diplomate

Principe de codage et représentation d'informations :

On définit la matrice suivante pour présenter les différentes informations.
Le but du problème est de considérer chaque case de la matrice et de trouver une valeur (1..5) pour ces cases en respectant les contraintes . Ces valeurs représenterons les numéros de maison. Par exemple, 1 dans la case P3 veut dire que le violoniste habite la première maisen (maison N° 1).
Les données initiales sont codées de la manière suivante.
(1) Anglais = Rouge, c'est à dire, le numéro trouvé pour Anglais doit être égal à celui de Rouge pour refléter le fait que l'anglais habite une masion de couleur rouge....
(10) Lait = 3 , c'est à dire, la maison où on boit du lait est la 3ème.
Pour miex comprendre :

Par (6), on peut écrire : Ci = verte => Bi = caffé. On a formulé ceci autrement: (Ci = verte & Bj = caffé) => i=j. C'est à dire, le même N° dans les deux cases.

	1	**2**	**3**	**4**	**5**
Nationa	Anglais	Espagnole	Japonais	Italien	Norvégien
Couleur	Vetre	Rouge	Jaune	Bleu	Blanche
Profess	Peintre	Diplomate	Violoniste	Medecin	Sculpteur
Animal	Chien	Zèbre	Renard	Escargots	Cheval
Boisson	Jus	Eau	The	Caffé	Lait

Pour simplifier les noms des variables, nous allons prendre Les premières lettres de la première colonne combinée avec 1..5. Par exemple, Japonais devient N3.

Une solution en CHIP :

```
domain maison({1,2,3,4,5}).
maison([N1,N2,N3,N4,N5, C1,C2,C3,C4,C5,
     P1,P2,P3,P4,P5, A1,A2,A3,A4,A5, B1,B2,B3,B4,B5]) :-

      N1=C2, N2=A1, N3=P1, N4=B3, N5=1,      %contraintes  1..5
      C1=B4,P5=A4,   P2=C3, B5=3, P3=B1,     % contraintes  6..12 sauf 7
      plusun(C1,C5)                          % contrainte 7

      alldif(<C1,C2,C3,C4,C5>)
      alldif(<P1,P2,P3,P4,P5>)
      alldif(<N1,N2,N3,N4,N5>)
      alldif(<A1,A2,A3,A4,A5>)
      alldif(<B1,B2,B3,B4,B5>)
      plusoumoinsun(A3,P4)                   % contrainte 11
      plusoumoinsun(A5,P2)                   % contrainte 13
      plusoumoinsun(N5,C4)                   % contrainte 14

      labeling([N1,N2,N3,N4,N5, C1,C2,C3,C4,C5,        % énumération
           P1,P2,P3,P4,P5, A1,A2,A3,A4,A5, B1,B2,B3,B4,B5]).
alldif([]) .
alldif([X|L]) :- outof(X,L) alldif(L).
```

outof(X,[]).
outof(X,[Y|L]) :- X ≠ Y, outof(X,L) .

plusoumoinsun(X,Y) :- X is Y+1.
plusoumoinsun(X,Y) :- X is Y-1.
plusun(X,Y) :- X is Y+1.

On a les déclarations :

forward plusun(X,Y).
forward plusoumoinsun(X,Y)

La difficulté du problème est dans les contraintes 7,11,13 et 14 :
plusun(C1,C5)
plusoumoinsun(A3,P4)
plusoumoinsun(A5,P2)
plusoumoinsun(N5,C4)

On note que *plusoumoinsun(X,Y)* est une *contrainte disjonctive*. Ce qui couteux et rare en programmation par contraintes (on l'évite au maximum car il engendre des retour arrières importants).

Comportement du programme :
Après avoir installé les contraintes d'égalité et de non-égalité, la contrainte *plusoumoinsun(N1,C5)* est immédiatement résolue avec C4=2 (car N5=1). C'est parce que plusoumoinsun(X,Y) est forward-checkable t il reste une valeur pour C4 dans son domaine. Cette valeur est immédiatement donnée à C4. Ce qui entraine la propagation et qq non-églités sont activées. Puis, on doit faire un choix pour C1. Les valeurs possibles sont {1,4,5}. Le premier choix est rejeté car il viole la contrainte C1=C5+1, C5 ne pouvant pas être =0. Le choix de 4 est également rejeté car, par la prpagation, C1..C5, D1..D5, N1,N4,N5, P2, P3 et A2 prennenet des valeurs. La prochaine variable assignée est P1 et son ensemle de valeurs possibles est réduit à deux variable qui sont rejetées par la propagation. Ainsi, la seule valeur possible pour C1 sera 5. La prochaine variable sera encore P1 dont

les valeurs possibles sont {2,4,5}. Les deux premières sont rejetées par propagation et la troisième mène à un esolution sans d'autres génération de valeur. Le problème est ainsi résolu avec 2 choix et 5 retours arrières.

Cet exemple montre l'interêt de l'utilisation de la stratégie forwardcheching qq soit les contraites.

Contrairement à la propagation locale (coroutinage), on n'est pas obligé d'avoir les contraintes complètement instanciées pour pouvoir les tester. Ainsi, avec les déclarations de forward,

Une solution en CLP(R) :

Une solution en PIII :

```
gentest(X) -> enumlist(X,5) maison(X);        %generer-tester
testgen(X) -> maison(X) enumlist(X,5);        %contraindre-generer
maison(< N1,N2,N3,N4,N5, C1,C2,C3,C4,C5,
      P1,P2,P3,P4,P5, A1,A2,A3,A4,A5, B1,B2,B3,B4,B5>) ->

      alldif(<C1,C2,C3,C4,C5>)
      alldif(<P1,P2,P3,P4,P5>)
      alldif(<N1,N2,N3,N4,N5>)
      alldif(<A1,A2,A3,A4,A5>)
      alldif(<B1,B2,B3,B4,B5>)
      plusoumoinsun(A3,P4)
      plusoumoinsun(A5,P2)
      plusoumoinsun(N5,C4)
      {N1=C2, N2=A1, N3=P1, N4=B3, N5=1, B5=3, P3=B1, C1=B4,
      P5=A4, P2=C3,
      C1=C5+1
      };
alldif(<>) -> ;
alldif(<X>.L) -> outof(X,L) alldif(L);
outof(X,<>) -> ;
outof(X,<Y>.L) -> outof(X,L) {X # Y};
plusoumoinsun(X,Y) -> {X=Y+1};
plusoumoinsun(X,Y) -> {X=Y-1};
enumlist(<>,N)->;
```

enumlist(<X>.L,N) -> enum(X,N) enumlist(L,N);

la réponse :
testgen(< N1,N2,N3,N4,N5, C1,C2,C3,C4,C5,
P1,P2,P3,P4,P5, A1,A2,A3,A4,A5, B1,B2,B3,B4,B5>);

{N1 = 3, N2 = 4, N3 = 5, N4 = 2, N5 = 1,
C1 = 5, C2 = 3, C3 = 1, C4 = 2, C5 = 4,
P1 = 5, P2 = 1, P3 = 4, P4 = 2, P5 = 3,
A1 = 4,A2 = 5, A3 = 1, A4 = 3, A5 = 2,
B1 = 4, B2 = 1, B3 = 2, B4 = 5, B5 = 3}

Pour la méthode générer-tester, la solution est très très longue à obtenir car il y a 25 variables chacune pouvant avoir 5 valeurs (5^{25}) combinaisons.

Le problème des mots croisés

Ce problème consiste à remplir une grille des mots croisés avec un ensemble prédéfini de mots avec la contrainte qu'un mot ne doit être utilisé qu'une seule fois. Un hmain habitué à ce genre de problèmes peut résoudre ce cas en 15 minutes.

Cet exemple montre l'intêret de disposer une règle d'inférence générale telle que FC : quelque soit les contraintes du problème, FC peut être utilisé pour obtenir un gain très important dans les performances.

De plus, il montre les techniques d'exppression des contraintes , aspect délicat des CSP lorsque le nombr de contraintes est lever et il faut obtenir une granulité fine en prgrammant l'introduction des contraintes (eg. alldif) plutot que de les énumérer soi-même à la main.

La solution avec Forward Checking :

On associe une variable à chaque mot (une suite de case blanches) de la grille; in a ainsi 22 variables. Chacune de ces variables a un domain qui est l'ensemble des mots qui peuvent entrer dans ces cases. Les contraintes du problème sont de deux types :
- contraintes de non égalité pour montrer qu'un mot ne peut figurer qu'une seule fois.
- les contraintes concernant les intersections des mots.

Pour ce problème, il y a :
- 242 contraintes de non-égalié
- 39 contraintes d'intersection

Une solution est trouvée en moins d'une minute.

Expression des contraintes :
Une contraintes d'intersection est précisée par *accord(I, Mot1, J, Mot2)* qui, pour les entiers I et J et les mots Mot1 et Mot2 sera vérifiée si la Ieme lettre de Mot1 est la même que la Jeme lettre de Mot2.

accord(I, Mot1, J, Mot2) :-
lettre(I,Mot1, Lettre), lettre(J, Mot2, Lettre).

Cette contrainte est déclarée forward :
forward lettre(g,d,g,d).

Ce qui veut dire que le test est fait lorsque les deux mots ont déjà une valeur.

Le programme installe les contraintes puis lance un générateur de valeur pour les variables.
Pour améliorer des performances, dans le cas des grandes grilles, on peut donner des valeurs aux variables correspondant aux plus larges places; les autres valeurs sont trouvées par les contraintes .

Pour l'exemple ci-dessus
les variables :
- Les mots à une lettre ne génèrent aucune contrainte.

- les autres mots seront :

mot2(C_1_2) mot2(C_6_7) mot2(E_1_2) mot2(E_6_7)
mot2(w_3_AB) mot2(w_3_FG) mot2(w_5_AB) mot2(w_5_FG)
mot3(A_1_3) mot3(A_5_7) mot3(G_1_3) mot3(G_5_7)
mot3(w_1_AC) mot3(w_1_EG) mot3(w_7_AC) mot3(w_7_EG)
mot5(D_2_6) mot5(w_4_BF)
mot7(B_1_7) mot7(F_1_7) mot7(w_2_AG) mot7(w_6_AG)

mot2(<C1,C2>) mot2(<C6,C7>) mot2(<E1,E2>)
mot2(<E6,E7>) mot2(<A3,B3>) mot2(<F3,G3>)
mot2(<A5,B5>) mot2(<F5,G5>)
mot3(<A1,A2,A3>) mot3(<A5,A6,A7>)
mot3(<G1,G2,G3>) mot3(<G5,G6,G7>)
mot3(<A1,B1,C1>) mot3(<E1,F1,G1>)
mot3(<A7,B7,C7>) mot3(<E7,F7,G7>)
mot5(<D2,D3,D4,D5,D6>) mot5(<D4,C4,D4,E4,F4>)
mot7(<B1,B2,B3,B4,B5,B6,B7>)
mot7(<F1,F2,F3,F4,F5,F6,F7>)
mot7(<A2,B2,C2,D2,E2,F2,G2>)
mot7(<A6,B6,C6,D6,E6,F6,G6>)

Expression des contraintes : meilleure granulité

On exprime les contraintes d'utilisation des mots par un prédicat tel que *alldif(liste_des_mots)* qui introduit les 242 contraintes de cette nature.

simples mais trop longues à écrire.

Pour les problèmes d'intersection, on peut énumérer ces intersections à l'aide du prédicat *accord*.

la complexité :

Si dans le lexique, on a D mots à L lettres et dans la grille, G mots ($G \leq D$) à L lettres, il y a D! possibilités pour les mots à L lettres. La complexité est le produits de ces complexités élémentaires.

Un cas simplifié :

Pour la grille suivante et les mots codés sous forme de lettres :

```
mot6("better") .      mot6("cannon").
mot6("wealth") .    mot6("dearth") .
mot5("brake") .     mot5("bloke") .          mot5("steam") .
mot5("cream").      mot5("patch") .          mot5("pitch").
mot4("bump") .      mot4("play") .
mot4("free") .  mot4("stop") .
```

```
solution(WA,WC,W2,W5):-
      accord(3, W2, 2, WC)   accord(3, W5, 5, WC)
      accord(1, W2, 1, WA)   accord(4, WA, 1, W5)
      alldif([WA,WC,W2,W5]),
      mot4(WA), mot6(WC),  mot5(W2),  mot5(W5).

accord(I,W1,J,W2) :- lettre(I,W1,Lettre),  lettre(J,W2,Lettre);

lettre(1,[X|_],X) .
lettre(n,[X|L],M) :- lettre(n-1,L,M), n>1.
```

Une solution en PrologIII

```
mot6("better") -> ;   mot6("cannon") -> ;
mot6("wealth") -> ; mot6("dearth") -> ;
mot5("brake") -> ;  mot5("bloke") -> ;        mot5("steam") -> ;
mot5("cream") -> ;  mot5("patch") -> ;        mot5("pitch") -> ;
mot4("bump") -> ;  mot4("play") -> ;
mot4("free") -> ;           mot4("stop") -> ;

solution(W_A,W_C,W_2,W_5) ->
      accord(3, W_2, 2, W_C)       accord(3, W_5, 5, W_C)
```

```
        accord(1, W_2, 1, W_A)        accord(4, W_A, 1, W_5)
        alldif(<W_2,W_5,W_A,W_C>)
        mot4(W_A)    mot6(W_C)    mot5(W_2)      mot5(W_5);

accord(I,W1,J,W2) -> lettre(I,W1,l_ettre)  lettre(J,W2,l_ettre);

%les chaines sont des tuples
lettre(1,<X>._,X) -> ;
lettre(n,<X>.L,M) -> lettre(n-1,L,M) {n>1};

alldif(<>) -> ;
alldif(<X>.L) -> outof(X,L) alldif(L);
outof(X,<>) -> ;
outof(X,<Y>.L) -> outof(X,L) {X # Y};

Les réponses :
{A = "stop", B = "wealth", C = "steam", D = "patch"}
{A = "stop", B = "wealth", C = "steam", D = "pitch"}
{A = "stop", B = "dearth", C = "steam", D = "patch"}
{A = "stop", B = "dearth", C = "steam", D = "pitch"}
```

Conclusion des méthodes

Les langages de programmation logiques sont des outiles puissant de description. Cependant, leur stratégies provoque une inefficacité dans certaines classes de problèmes, nottament les CSP.

Nous avons vu que les langages de programmation logiques peuvent utiliserdes efficacement des méthodes telles que FC pour implanter la propagation des contraintes , produire des solutions pour les CSP par une recherche efficace. FC est montrée plus efficace que les coroutinage ans tous les cas de figure.

D'autres techniques de recherche telle que looking-ahead existent pour résoudre les problème de CSP.

Méthodes (guidelines) pour CHIP

- Méthodes de CSP (livre Pascal, p 111)
- Papier de Pascal

Général propagation : une autre classe de contraintes

Cadre Théorique

Théorie : Jaffer/lassez

Unification (booléenne) dans les anneaux booléens.

- Dans le papier de Dincbas (vieux) on a toute la théorie sur l'unification dans l'algèbre de bool & les théorèmes) avec l'algo d'unification.

A partir de la page 59 de Cohen . C'est bien pour CLP.

Titre suivant

Autres Systèmes CLP

- BNR...(CAL)... voir papier ECRC
- Deux mots sur LIFE, on va en parler après

Eléments d'implantation des systèmes classiques

comment résoudre des contraintes :

- CLP(R) = PIII
- Papier codognet pour CHIP

Vers des extensions de CLP : LIFE

- Parallèle avec les extension de PL : Extensions de CLP
  - fonctions
  - objets (LIFE)

Parallèle avec les extension de PL Extensions de CLP

- fonctions
- objets (LIFE)