Les intervalles de Prolog IV
Exemples et méthodes

Stéphane N'DONG

25 Septembre 1996 - (Version 0.8+++)

Table des matières

1 Introduction

Ce document s'adresse à ceux qui souhaitent exploiter les aspects numériques de Prolog IV, et plus particulièrement les intervalles, que ce soit à des fins de démonstration des mécanismes de CLP/CP, ou pour résoudre des problèmes. Son objet est à la fois pédagogique, explicatif, et démonstratif.

Chacun des points abordés dans ce document l'est de la manière la plus intuitive et la plus simple possible. De nombreux exemples et illustrations contribuent à cette clarté. Cependant, afin de ne pas minimiser la partie théorique, importante dans la philosophie qui sous-tend les systèmes de résolution numériques basés sur les intervalles, un support théorique plus fondamental est également présent. Néanmoins, afin de ne pas perdre le lecteur débutant (ces parties plus théoriques peuvent être abordées dans un deuxième temps), ou les personnes qui ne sont pas intéressées par les développements sémantiques, deux règles sont respectées : les parties théoriques sont isolées dans des sections spécifiques ; et il n'est jamais supposé dans les parties plus pragmatiques que les parties théoriques précédentes ont été lues.

Des exemples continus ou discrets, évidents à programmer ou à résoudre , ou plus délicats serviront à illustrer diverses techniques de programmation et d'énumération.

Ce document ne prétend bien sûr pas représenter une méthodologie complète de programmation avec les intervalles. Il explore néanmoins quelques techniques non-naïves de résolution, particulièrement dans le domaine continu.

Organisation du document

Le cœur du document est constitué de cinq parties :

1. Le chapitre 2 : "Résolution avec les intervalles"
   Il s'agit essentiellement de décrire à différents niveaux les principales facettes des mécanismes de résolution impliqués. Ce chapitre ne contient pas d'exemple en Prolog IV. La lecture des sections 2.1 (introduction), 2.3 (vision intuitive) et 2.7 (Détail des relations) est utile pour comprendre les mécanismes fondamentaux et le fonctionnement du noyau de Prolog IV.

2. Le chapitre 3 : "Expression d'un problème en Prolog IV"
   On a regroupé dans ce chapitre plusieurs aspects des liens existant entre formulation mathématique et programmation. En particulier, les sections 3.1 (quelques techniques de base) et 3.2 (liens entre expression et efficacité) peuvent être utiles au programmeur. La section 3.3 (CLP et mathématiques) présente un point de vue plus "philosophique".

3. Le chapitre 4 : "Techniques d'énumération"
   Les sections 4.1 (principes) et 4.2 (méta-prédicats de Prolog IV) sont très utiles pour aborder le point fondamental qu'est l'énumération. La section 4.3 (optimisation) détaille trois méthodes de base permettant la recherche d'optima.

4. Le chapitre 5 : "Exemples de requêtes et programmes"
   Ce chapitre regroupe un certain nombre d'exemples de natures diverses. Nombre d'entre eux sont classiques, et le programmeur y trouvera des points de repère pour écrire ses propres programmes. La formalisation et le programme sont détaillés pour chaque exemple.

5. Le chapitre 6 : "Résolution de quelques problèmes délicats"
   Ce chapitre ne s'adresse pas aux débutants. Il regroupe un certain nombre de techniques permettant de s'attaquer à des problèmes non triviaux du domaine continu. Sa lecture, à part la section 6.1 (description des problèmes et première résolution naïve) n'est pas évidente, et mélange idées, sémantiques, et résultats expérimentaux.

<u>Lecture du document</u>

Voici enfin quelques remarques préliminaires :

- Pré-requis :
  Il n'est pas nécessaire pour consulter ou lire ce document d'avoir assimilé tous les principes de base, et en particulier les principes fondamentaux de Prolog IV. Par contre, il est préférable de connaître certains aspects du langage, principalement pour comprendre aisément les exemples de programmes et de requêtes. On peut citer :

- Une bonne connaissance de Prolog ISO, de manière à pouvoir déchiffrer la structure des programmes facilement. Notamment l'usage du "cut", de la négation par l'échec, et des prédicats `record` et `recorded`.
- La compréhension des liens unissant relations Prolog IV et pseudo-termes.
- La connaissance des principaux symboles fonctionnels graphiques :
  * Les symboles numériques : `.+.`, `.-.`, `.*.`, `./.`, `.^.`
  * Les symboles booléens : A chaque relation booléenne pure est associé un opérateur infixe : le nom de la relation elle-même (`or`, `bor`, `xor`, ...).
  * Les symboles de comparaison : `le`, `ge`, `lt`, `gt`
  * Les symboles généraux : `=`, `~`, `u`, `n`, `o`, `:`
  * La négation par l'échec : `\+`

- A propos des styles :
  Il existe deux styles spécifiques :

  1. Le style mathématique (par exemple $X = Y + Z$).
  2. Le style Prolog IV (par exemple `plus(X,Y,Z)`).

  Tout ce qui est écrit en utilisant le style Prolog IV répond à la syntaxe et à la sémantique de Prolog IV, et non aux principes mathématiques habituels. Par exemple `X = Y + Z` signifie "Ajouter la contrainte linéaire $X = Y + Z$ dans le système de résolution linéaire rationnel de Prolog IV" ; alors que `X = Y .+. Z` signifie "Ajouter la contrainte linéaire $X = Y + Z$ dans le système de résolution avec les intervalles de Prolog IV".

- A propos des écritures mathématiques :

  - Pour éviter des confusions entre les résultats des programmes Prolog IV et le texte lui-même, toutes les quantités numériques sont écrites avec un point décimal (écriture anglaise), et non une virgule. Dans le texte, pour faciliter la lisibilité, les triplets de chiffres sont séparés par un blanc. Exemple : "333 441.7".

  - La notation mathématique des intervalles est la notation américaine, c'est à dire que les ouverts sont notés par des parenthèses (et non des crochets), et les fermés par des crochets (et non des crochets inversés).

- A propos des programmes :
  Les temps d'exécution de tous les programmes sont indiqués en annexe. Dans certains cas, cependant, pour comparer deux méthodes par exemple, ils apparaissent également dans le texte.

- A propos de la segmentation de ce document.
  Ce document est sectionné en chapitres (i), sections(i.j), et paragraphes(i.j.k).

- A propos de ce document :
  Ce document est encore, et dans l'idéal devrait toujours être, en période d'évolution. Il ne faut donc pas s'etonner que certaines parties soient encore "en chantier", ou que d'autres soient manquantes : elles apparaîtront ou seront complétées dans une prochaine version du document.

2 Résolution avec les intervalles

2.1 Introduction

De nombreux problèmes industriels ou de la vie courante se présentent naturellement sous la forme de systèmes de contraintes. On va se restreindre ici à une sous-catégorie de ces problèmes : ceux composés de contraintes numériques réelles. Cette restriction impose, dans le cas général, une étape de raisonnement bien connue du monde scientifique : la formalisation du problème.

Cet obstacle franchi, on se trouve face à un système de contraintes reposant sur un certain nombre d'opérateurs et de fonctions réelles. Citons entre autres l'addition, la multiplication, les fonctions algébriques (puissances, racines), les fonctions transcendantales (hyperboliques, trigonométriques, exponentielles, etc...). On y trouvera également des fonctions de nature entière (partie entière, PGCD, etc...), les fonctions booléennes, ainsi que bon nombre de fonctions et relations mixtes.

Malheureusement, il n'existe aucun algorithme (du moins, utilisable dans la pratique), susceptible de résoudre de tels systèmes de contraintes. Loin s'en faut !

Il a donc fallu imaginer une stratégie pouvant s'attaquer à de tels systèmes. Comme nous le verrons plus en détail un peu plus loin, il y a deux manières d'envisager cette stratégie :

1. La vision la plus intuitive considère que, ne sachant pas résoudre le problème initial exactement, on va le résoudre de manière approchée, obtenant alors des solutions sous la forme d'intervalles de nombres réels. La caractérisation principale de cet algorithme de résolution approché étant de garantir que toute solution du problème initial est incluse dans l'intervalle-résultat. **On dira que l'on résout de manière approchée un problème exact.**

2. L'autre vision est peut-être moins intuitive mais elle est, d'une certaine manière, plus précise et plus proche de la réalité. On va considérer ici que l'on effectue premièrement une transformation du problème initial en un problème plus faible, que l'on appellera problème approché. On

> va ensuite le résoudre de manière exacte. **On dira que l'on résout de manière exacte un problème approché.**

Même si l'on parle dans les deux cas du même algorithme, le point de vue que l'on en a n'est pas innocent : la deuxième vision laisse clairement entrevoir la rigueur de la méthodologie utilisée, ce qui n'est pas forcément le cas de la première.

Comme on l'a entrevu plus tôt, un élément fondamental du système de résolution par approximation est l'intervalle. Il représente en fait le moyen d'exprimer une approximation de tout sous-ensemble des nombres réels. Dans le cas de Prolog IV, on a choisi de prendre les bornes de ces intervalles parmi les nombres flottants simples de la norme IEEE 754.

Pourquoi ce choix ?

Premièrement, le choix d'un sous-ensemble fini des nombres réels permet de garantir la terminaison de l'algorithme de résolution en un temps fini (cf. propriété 2.15 à la section 2.4.3); deuxièmement, les nombres flottants possèdent de très bonnes propriétés de représentation des nombres réels et de comportement (cf. 2.2), et garantissent de bonnes performances grâce à des implantations généralement réalisées au niveau matériel ; enfin, ils assurent un comportement rigoureusement identique sur toutes les plates-formes.

C'est donc ce système de résolution avec les intervalles que nous allons décrire dans ce chapitre. Dans un premier temps, nous décrirons les nombres flottants et leurs principales propriétés. Nous proposerons ensuite une approche intuitive de ces mécanismes de résolution, que nous compléterons dans un deuxième temps par une description plus précise et plus théorique.

Nous aborderons ensuite rapidement un modèle de résolution dans une extension de $\mathbb{R}$, puis nous détaillerons les mécanismes de Prolog IV permettant de vérifier et d'effectuer des calculs en précision infinie avec des nombres rationnels.

Enfin, nous terminerons en montrant les relations disponibles en Prolog IV, et nous ferons quelques commentaires sur certaines familles de relations.

2.2 Les nombres flottants

Le système de résolution que l'on présente dans ce document repose essentiellement sur l'approximation des ensembles de nombres réels par des intervalles dont les bornes sont les flottants simples de la norme IEEE 754. On va survoler dans cette section cette norme, en insistant sur les points qui nous ont conduits à les utiliser.

Structure des nombres flottants

Les nombres flottants s'opposent aux nombres à virgule décimale fixe. Ces derniers s'articulent autour de la virgule décimale, avec un certain nombre de chiffres à gauche, et un certain nombre de chiffres à droite de cette virgule. Dans ce type de systèmes numériques, la distance entre deux nombres consécutifs est constante. La représentation des nombres entiers obéit en général à ces règles. Les nombres flottants, eux, reposent sur une information divisée en deux : la **mantisse**, qui permet de représenter un certain nombre de chiffres significatifs, et l'**exposant**, qui permet de déplacer la virgule ; d'où le nom de "nombres à virgule flottante", ou "nombres flottants".

Les nombres flottants auxquels on va s'intéresser sont les flottants de la norme IEEE 754 simples. Il sont codés sur 32 bits en machine. Il existe dans cette norme d'autres types de nombres flottants, mais ils ne diffèrent que par leur taille en machine, et donc leur précision ; leurs propriétés étant les mêmes.

Voici deux définitions équivalentes des nombres flottants. La première, très proche de l'implantation, est celle donnée par la norme. La deuxième, qui s'en déduit, est plus claire, et plus intégrée dans les nombres rationnels.

1. Première définition :
   Les nombres flottants IEEE 754 (single) sont les nombres composés :
   - d'un signe s sur 1 bit.
   - d'un exposant e sur 8 bits.
   - d'une mantisse m sur 23 bits.

Il existe alors cinq types de nombres flottants :

Exposant e	Mantisse m	Valeur v	Type de v
$0 < e < 255$	tout m	$v = (-1)^s * 2^{(e-127)} * (1.m)$	Normalisé
$e = 0$	$f \neq 0$	$v = (-1)^s * 2^{-126} * (0.m)$	Dénormalisé
$e = 0$	$f = 0$	$v = (-1)^s * 0$	Zéro
$e = 255$	$f = 0$	$v = (-1)^s * Infini$	Infini
$e = 255$	$f \neq 0$	v est un NaN	NaN

2. Deuxième définition :
   Les nombres flottants IEEE 754 (single) sont les nombres réels v tels que : $v = \pm p * 2^q$ avec $0 \leq p \leq 2^{24} - 1$ et $-149 \leq q \leq 104$, plus les deux infinis, et des "non-nombres" (NaN).

Les flottants sont au nombre de 4 278 190 081 (y compris les deux infinis).

Comme le démontrent ces définitions (la deuxième en particulier), les nombres flottants finis IEEE 754 (single) sont des nombres rationnels. **On les appellera désormais rationnels IEEE 754**.

Propriétés des nombres flottants

Voici les principales propriétés des nombres flottants, qu'elles proviennent de leur structure, ou qu'elles soient imposées par la norme.

1. Propriété d'homogénéité.
   La principale propriété structurelle des nombres flottants, qui est à l'origine de leur création, est leur homogénéité dans le corps des nombres réels :
   - Homogénéité par rapport à l'addition : si f est un rationnel IEEE 754, alors $-f$ en est un. Les rationnels IEEE 754 sont répartis de manière homogène autour de 0.
   - Homogénéité par rapport à la multiplication : si f est un rationnel IEEE 754, alors en général (si l'on exclut les flottants dénormalisés) $1/f$ en est un également. Les rationnels IEEE 754 sont également répartis entre 0 et 1 et entre 1 et $+\infty$.

2. La norme connaît cinq opérations fondamentales : l'addition, la soustraction, la multiplication, la division, la racine carrée. Ces opérations sont effectuées exactement en fonction d'une définition paramétrable du mode d'arrondi. Celui-ci peut être :

   - Au plus proche
   - Au plus grand
   - Au plus petit
   - Au plus proche de 0

   Il va de soi que le mode d'arrondi est indifférent si le résultat est un rationnel IEEE 754.

   Les comparaisons et les conversions avec des nombres entiers sont également effectuées de manière optimale.

   Ces propriétés correspondent exactement aux nécessités de programmation des systèmes basés sur les intervalles. Ce n'est d'ailleurs pas tout à fait un hasard. En effet, l'un des débats autour de la création de cette norme concernait l'inclusion des intervalles eux-mêmes dans les structures de base. Cette option fut écartée, mais ses défenseurs obtinrent que le matériel nécessaire à leur réalisation efficace soit inclus dans la norme.

3. Comme on l'a dit, il existe d'autres types de nombres flottants, plus précis. On a cependant choisi ici de privilégier les flottants simples, plus efficaces et moins encombrants. Le choix de nombres flottants de plus grande précision se justifie en général par l'incapacité des systèmes numériques à gérer les erreurs de calcul correctement. Or, comme on le verra plus loin, les mécanismes mis en jeux ici reposent sur une gestion très précise de ces phénomènes. Dès lors, l'usage de nombres flottants plus précis ne ferait qu'alourdir la machine Prolog IV, sans vraiment apporter de précision supplémentaire. Des nombres flottants doubles (sur 64 bits) implantent d'ailleurs les nombres flottants tels que la norme Prolog les définit.

Ecriture des rationnels IEEE 754 en Prolog IV

Comme le suggère la définition des rationnels IEEE 754, une de leurs caractéristiques principales est d'avoir un nombre de chiffres significatifs stable (environ 6/7).

Se pose alors le problème suivant : comment écrire (affichage des résultats Prolog, par exemple) les rationnels IEEE 754 en respectant le cahier des charges suivant :

1. L'écriture d'un rationnel IEEE 754 ne doit être ambiguë. C'est à dire qu'à une écriture ne doit pouvoir correspondre qu'un seul rationnel IEEE 754.
2. La relecture : un rationnel IEEE 754 qui a été écrit doit pouvoir être relu à l'identique.
3. Son écriture doit être lisible facilement par l'œil humain. C'est à dire que sa valeur approximative doit être très visible.

Il existe en Prolog IV deux possibilités pour écrire un rationnel IEEE 754. La première consiste à le considérer comme un nombre rationnel à part entière. Cette méthode a un inconvénient majeur : le résultat est très peu lisible.

La deuxième méthode est la suivante :
Soit f un rationnel IEEE 754. On va alors écrire f sous la forme d'un couple (S, D) construit de la manière suivante :

- D est un nombre décimal.
- S est soit le symbole $'<'$, soit le symbole $'>'$, soit $\emptyset$ (vide).
- Si $S = \emptyset$, alors $D = f$.
- Si $S = \ '<'$, alors D est le plus petit nombre décimal (en nombre de chiffres) tel que $f < D < f^+$ (f^+ désignant le flottant suivant immédiatement f). Si plusieurs nombres décimaux correspondent à cette propriété, le plus petit, cette fois au sens usuel, est retenu.

- Si $S = ' >'$, alors D est le plus petit nombre décimal (en nombre de chiffres) tel que $f^- < D < f$ (f^- désignant le flottant précédent immédiatement f). Si plusieurs nombres décimaux correspondent à cette propriété, le plus petit, cette fois au sens usuel, est retenu.
- On retiendra celui des trois couples précédent d'écriture la plus courte.

Considérons par exemple la contrainte suivante : $\{X \in [1.1, 2]\}$.
Le domaine résultant sera $X \in (2306867/2097152, 2]$ car 1.1 n'est pas un rationnel IEEE 754. Cette écriture n'est pas très claire. Voici ce que l'on obtient en Prolog IV (qui utilise la technique que l'on vient de décrire pour afficher les domaines des variables) :

```
>> cc(X,1.1,2).

X ~ oc('<1.1',2).
```

Il est clair que la notation '<1.1' est plus explicite que 2306867/2097152 (sous forme rationnelle) , ou 1.099999904632568359375 (sous forme décimale complète).
Cette notation apporte donc la précision et la lisibilité voulue.

2.3 Résolution avec les intervalles : une approche intuitive

Remarque préalable : cette section n'a pas pour objectif une description précise et complète des mécanismes de résolution. Reportez vous pour cela à la sémantique générale de Prolog IV, ou à la section 2.4. L'objectif poursuivi ici est plutôt de faire comprendre les mécanismes de base. L'abord de parties plus théoriques semblerait délicat sans cet apprentissage.

Une première description très simple du système

L'état courant du système de résolution se caractérise par :

1. Une liste de variables, avec un intervalle associé à chacune d'elles, que l'on appellera "domaine".

2. Une liste de contraintes portant sur ces variables. Chacune de ces contraintes correspond à l'une des relations de base de Prolog IV (dont la liste est détaillée à la section 2.7) ; ce sont des contraintes de base.

Le mécanisme de résolution est le suivant :

Chacune des contraintes est réévaluée dans le but de réduire les domaines (intervalles) de ses variables, **en ne considérant que ceux-ci**. Réévaluer une contrainte consiste à calculer l'ensemble des valeurs que peut prendre chacune des variables, en fonction des intervalles qui leur sont associés, et de la définition de la relation correspondante. L'algorithme prend fin lorsqu'aucune réévaluation ne peut plus aboutir à la réduction du domaine d'une des variables. On dit alors que l'on a atteint le point fixe.

C'est la raison pour laquelle le système de résolution approché avec les intervalles pourra également être appelé "algorithme du point fixe", ou "point fixe".

Réévaluation d'une relation

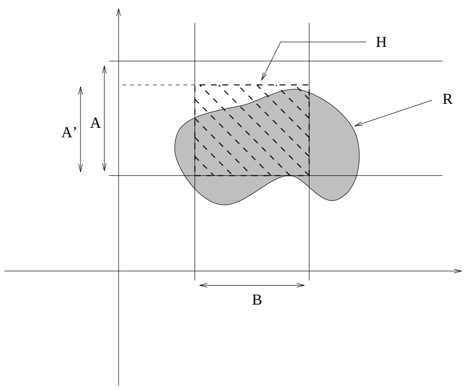
Comme on vient de le voir dans la description de l'algorithme du point fixe, l'opération de base de cet algorithme est la réévaluation des contraintes de base.

Voyons sur un exemple graphique comment cette opération est effectuée.

Sur le schéma suivant :

- La zone grisée R représente le graphe d'une relation binaire.
- Les zones A et B représentent les domaines des deux variables associées.
- La zone hachurée H représente l'approximation par un produit cartésien d'intervalles flottants de l'intersection entre la relation R et le produit cartésien des domaines des variables.

- La zone A' représente le nouveau domaine de la première variable (anciennement A). Ce nouveau domaine est obtenu en projetant la zone H sur l'axe de cette variable.



La réévaluation a donc permis de réduire le domaine de la première variable.

Si on reprend une à une les opérations que l'on a effectuées, on obtient : réévaluer une relation revient, pour chacune des variables associées, à calculer la projection de l'approximation de l'intersection de la relation avec le produit cartésien des domaines des variables. Une définition plus mathématique sera donnée à la section 2.4.

Des exemples simples

Considérons le système de contraintes suivant :

$$\begin{array}{lll} plus(X, Y, C) & (Eq1) & (X = Y + C) \\ times(Y, A, B) & (Eq2) & (Y = A * B) \end{array}$$

Avec les domaines initiaux suivants :

$A \in [1,2]$
$B \in [2,3]$
$C \in [1,2]$
$X \in I\!R$
$Y \in I\!R$

Voici la chaîne des réductions :

$(Eq2) \Longrightarrow \quad Y \in [2,6]$
$(Eq1) \Longrightarrow \quad X \in [3,7]$

On déduit donc que X est dans l'intervalle $[3,7]$.

Dans cet exemple, l'information transite de A, B, C vers Y, puis X, donnant l'impression d'effectuer un simple calcul sur des ensembles de nombres. Modifions un peu cet exemple de telle manière que l'information circule dans les deux sens :

On garde les mêmes relations, mais avec les domaines initiaux suivants :

$A \in [1,2]$
$B \in [2,3]$
$C \in [1,2]$
$X \in [0,3]$
$Y \in I\!R$

Voici la chaîne des réductions :

$$(Eq2) \Longrightarrow \quad Y \in [2,6]$$
$$(Eq1) \Longrightarrow \quad \left\{ \begin{array}{l} X \in [3,3] \\ Y \in [2,2] \\ C \in [1,1] \end{array} \right.$$
$$(Eq2) \Longrightarrow \quad \left\{ \begin{array}{l} B \in [2,2] \\ A \in [1,1] \end{array} \right.$$

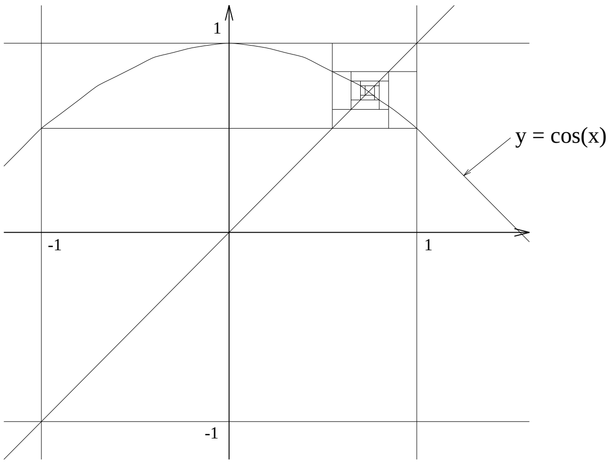
On déduit donc que $A = 1, B = 2, C = 1, Y = 2$, et $X = 3$.

Un autre exemple

Ce transfert circulaire de l'information par le biais des domaines des variables du problème est encore plus flagrant et plus intense sur l'exemple suivant :

$cos(X, Y)$ $(Eq1)$ $(X = cos(Y))$
$eq(X, Y)$ $(Eq2)$ $(X = Y)$

Voici une visualisation graphique de la chaîne de réductions :



On appelle ce type de chaînes de réduction "convergence en escargot" (à cause de la forme de la représentation graphique, et non de sa lenteur!). Le résultat final obtenu est : $X \in (0.739085, 0.7390852)$

Intervalles simples et unions d'intervalles

On a jusque là parlé très grossièrement d'intervalles, laissant entendre qu'ils représentent l'ensemble des nombres réels compris entre deux bornes, inférieurs ou supérieurs à une borne, ou l'ensemble des nombres réels lui-même. Il existe cependant un autre mode d'approximation des ensembles de nombres réels : les unions d'intervalles. Il suffit pour les obtenir de remplacer les intervalles flottants (ainsi dénommés à cause de leurs bornes) par des unions finies d'intervalles flottants : ces unions sont de toute façon forcément finies puisque le nombre d'intervalles flottants est fini (cf. sections 2.2, et 2.4.2). On obtient alors une approximation plus fine, mais plus coûteuse. On verra au fur et à mesure des exemples des cas où elle peut être intéressante.

Il existe deux moyens pour choisir le type d'approximation utilisé couramment par Prolog IV :

1. Sur la ligne de commande, en utilisant l'option `-union` pour lancer une session Prolog IV utilisant les unions d'intervalles (sans cette option, ce sont les intervalles simples qui sont utilisés).

2. Au cours d'une session Prolog IV, en utilisant :

   - `set_prolog_flag(interval_mode,union).` pour passer en unions d'intervalles
   - ou `set_prolog_flag(interval_mode,simple).` pour passer en intervalles simples

   ATTENTION : Ces prédicats sont fragiles, et risquent de provoquer des dysfonctionnements si elles sont utilisées alors que le système courant de contraintes n'est pas vide, ou en corrélation avec les prédicats record/recorded.

Remarques complémentaires

1. Le mécanisme que l'on vient de décrire ne s'appuie que sur un fonctionnement réduit au niveau de chaque relation. Seuls les domaines des variables réalisent un lien entre les différentes relations. C'est cet aspect qui fait souvent parler de **résolution locale** pour ce type d'algorithme.

2. Comme cela a été dit, les variables n'ont aucune existence formelle, puisque le seul "principe actif" de l'algorithme de résolution repose sur les domaines des variables. Une conséquence fâcheuse de cette caractéristique est l'incompétence du système à résoudre des systèmes

de contraintes apparemment triviaux comme $\{X = -X\}$. En effet, dans ce cas, le système ne voit que la contrainte *uminus* appliquée à deux variables dont les domaines dont égaux à $I\!R$, et ne peut donc rien en déduire. On pourra remarquer que, par contre, le système est capable de résoudre les systèmes de contraintes $\{X = -X, X < 0\}$ et $\{X = -X, X \geq 0\}$. On pourra remarquer également que le système de contraintes $\{X = -X\}$ est, du point de vue du système de résolution avec les intervalles, totalement différent du système de contraintes $\{2X = 0\}$, qui est, lui, parfaitement résolu.

3. On va introduire ici rapidement la notion de résolution satisfaisante d'un système de contraintes. Cette notion est purement intuitive, et n'est rien d'autre que l'écho des desiderata de l'utilisateur.
   Par exemple, $\{X = -X, X < 0\}$, qui provoque un échec Prolog IV (dont on devine qu'il est sémantiquement équivalent à des domaines vides pour toutes les variables), est résolu de manière satisfaisante. De manière générale, tout échec constitue une résolution satisfaisante ; on verra même à la section 2.4 que c'est l'une des rares situations où la résolution est parfaitement satisfaisante.
   Un autre exemple de résolution satisfaisante : $\{X = -X, X \geq 0\}$, qui déduit $\{X = 0\}$. De même, $\{X = cos(X)\}$ est résolu de manière satisfaisante, bien que le résultat rendu ne soit pas formellement égal à $\pi/4$, dans la mesure où il constitue un bon encadrement de la solution exacte. Enfin, il est clair que $\{X = -X\}$ n'est pas résolu de manière satisfaisante.

4. La définition intuitive que l'on vient de donner du système de résolution avec les intervalles laisse apparaître la propriété fondamentale suivante (cf. propriété 2.17 à la section 2.4.3) :
   Le mécanisme de résolution, bien qu'approché, ne "perd" jamais de solution. C'est évident, puisque chaque étape de l'algorithme réduit les domaines des variables en en enlevant uniquement des valeurs impossibles.
   On déduit immédiatement de cette propriété que si le point fixe aboutit à des intervalles vides, alors le système de contraintes considéré n'a pas de solution. C'est pourquoi on peut affirmer que la résolution est parfaite dans ce cas.

5. L'algorithme du point fixe repose fondamentalement sur la réévaluation de contraintes correspondantes à des relations de base. Ces relations existent en nombre fini. Elles sont initialement connues du système de résolution. La conséquence de ce mode de fonctionnement est que toute contrainte ne correspondant pas à une relation de base du système sera, si c'est possible, décomposée en de telles contraintes, pour pouvoir être interprétée par l'algorithme du point fixe. Le comportement de cette contrainte au cours de la résolution ne sera donc pas optimum, contrairement à celui des contraintes correspondant à des relations de base. On reviendra sur cet aspect, sur les difficultés qu'il pose, et sur les moyens de les résoudre dans les sections 3.2 et 6.2.

2.4 Résolution avec les intervalles : une définition plus précise

2.4.1 Introduction

On a introduit de manière relativement informelle les concepts fondamentaux de projection, d'approximation, et de point fixe dans la section précédente. On va maintenant s'attacher à donner de ces concepts une définition plus précise, plus mathématique.

L'objectif de cette section est triple. Le premier aspect, le plus évident, est la description précise de la technique de résolution utilisée. Mais le plus important est peut-être la démonstration que l'on essaie de faire au travers de cette description de la rigueur avec laquelle on traite cette résolution approchée. On montrera notamment dans cet esprit que la résolution obtenue par l'algorithme du point fixe est parfaitement définie. Enfin, ce traitement précis et les propriétés qui en découlent nous permettront de parler avec exactitude, et de comparer les mérites et lacunes de différentes techniques d'énumérations ou de résolutions basées sur les intervalles (cf. chapitre 6).

Comme on l'a entrevu dans la section précédente, il existe deux approches possibles de ce type de système de résolution, selon que l'on considère que l'on résout de manière exacte une approximation d'un problème ou que l'on résout directement un problème, mais de manière approchée.

C'est cette double optique qui va conditionner le plan de cette section. On abordera dans un premier temps la notion d'approximation, puis on développera l'approche présentant la résolution exacte d'un problème approché. Ce n'est que par la suite que l'on extrapolera les principales caractéristiques des solutions obtenues, par rapport au problème initial.

Quelques remarques pour finir cette introduction :

- Le lien entre cette section et la description des concepts de base de Prolog IV (chapitre 2 de la documentation générale) passe par la notion de sous-domaines privilégiés, et essentiellement par l'axiome de réduction de domaine. Simplement, on se limite ici à l'aspect numérique des sous-domaines. En fait, pratiquement tout ce qui va être dit dans cette section pourrait être étendu directement à l'approximation de l'ensemble des arbres.

- L'ensemble de ce document est d'orientation principalement pratique; cependant d'autres sections de nature plus théoriques suivront celle-ci. Elles comportent toutes des références à cette section, qui introduit les concepts de base. Il est donc fortement conseillé de lire cette section préalablement. Cette lecture ne s'impose cependant pas pour la compréhension des parties non-théoriques du document. Chaque section à vocation théorique commencera par la liste des sections théoriques précédentes requises à sa lecture.

- L'ensemble des démonstrations se trouve en annexe, de manière à ne pas alourdir ces parties théoriques que l'on a préféré étayer par des explications plus naturelles que le simple énoncé des définitions et propriétés.

2.4.2 Notion d'approximation

S'il est un mot - et un concept - qu'il est difficile de contourner quand on parle de résolution avec les intervalles, c'est bien celui d'approximation.

Il y a cependant un piège dans lequel il ne faut pas tomber : il y a en fait deux niveaux d'approximation. Le premier, qui saute aux yeux, n'est pourtant pas le plus fondamental : il s'agit de l'approximation liée à l'usage des nombres flottants qui ne peuvent qu'approcher imparfaitement les nombres réels. On verra un peu plus loin sur un exemple simple d'approximation que l'absence de ce niveau d'approximation ne change pratiquement rien aux problèmes rencontrés. Par contre, l'approximation des sous-ensembles quelconques de nombres réels par des produits cartésiens d'intervalles flottants est fondamentale. C'est elle qui permet de résoudre facilement n'importe quel problème, mais c'est également elle qui éloigne le problème approché du problème initial.

On va donner dans ce paragraphe les définitions et propriétés fondamentales d'une approximation. On détaillera ensuite celles utilisées par Prolog IV.

Définitions

Définition 2.1 *Soit E un ensemble quelconque.*

α est une **base d'approximation** *sur E si α est un sous-ensemble des parties de E fermé par intersection, et contenant l'ensemble vide et lui-même.*

Soit B un sous-ensemble quelconque de E.

On définit alors **l'approximation de** *B dans la base α comme étant le plus petit élément (au sens de l'inclusion) de α contenant B, s'il existe.*

On note : $appx_\alpha(A) = Min\{B \in \alpha \mid B \supset A\}$

On étend naturellement cette définition à un sous-ensemble A quelconque de E^n :

$appx_\alpha(A) = appx_\alpha(proj_1(A)) \times ... \times appx_\alpha(proj_n(A))$

En d'autres termes, l'approximation de $A \subset E^n$ dans α est exactement l'approximation de A dans α^n.

Définition 2.2 *Une base d'approximation α définit* **une approximation complète** *sur E si tout sous-ensemble de E a une approximation dans α, c'est à dire si :*

$\forall A \subset E$, $\{B \in \alpha \mid B \supset A\}$ possède un minimum.

Une base d'approximation α définit **une approximation finie** *si α est un ensemble fini.*

Définition 2.3 *Soit α une base d'approximation sur E, A un élément non vide de α.*

A est un **élément d'approximation atomique** *si les seuls éléments de α contenus dans A sont l'ensemble vide, et A lui-même.*

Propriétés

Propriété 2.4 (Propriétés de base des approximations) *Soit E un ensemble quelconque, α une base d'approximation sur E, A un sous-ensemble quelconque de E.*

1. *Si $appx_\alpha(A)$ existe, alors $appx_\alpha(A) = \bigcap_{B \in \alpha, B \supset A} B$.*
2. *Si $appx_\alpha(A)$ existe, alors $A \subset appx_\alpha(A)$.*
3. *$appx_\alpha(A) = A$ si et seulement si $A \in \alpha$.*
4. *Si $appx_\alpha(A) = \{s\}$, alors $A = \{s\}$.*
5. *Si $appx_\alpha(A_1) = B_1$, $appx_\alpha(A_2) = B_2$, alors $appx_\alpha(A_1 \cup A_2) = appx_\alpha(B_1 \cup B_2)$.*

Les preuves de ces propriétés se trouvent à l'annexe 9.3.1, page 192.

Propriété 2.5 *Soit E un ensemble, α une approximation sur E.*
Si α définit une approximation finie, alors celle-ci est complète.

La preuve de cette propriété se trouve à l'annexe 9.3.1, page 192.

Quelques exemples d'approximation

1. Dans tout ensemble E, on peut toujours définir la base d'approximation complète et finie, bien que sans grand intérêt : $\alpha = \{\emptyset, E\}$.

2. De même, l'ensemble des parties de E forment toujours une base d'approximation.

3. Un exemple d'approximation sur $\mathbb{Q}$.
Soit $\alpha_{\mathbb{Q}}$ l'ensemble des intervalles de $\mathbb{Q}$ à bornes rationnelles.
Cet exemple d'approximation illustre parfaitement les propos tenus dans l'introduction. En effet, si l'on s'avance un peu sur la suite du document, en supposant que l'on ne travaille qu'avec l'addition, la négation, ou la multiplication par une constante rationnelle non nulle, on imagine facilement que le premier "aspect" de l'approximation telle qu'on la détaillait, c'est à dire l'approximation des nombres eux-mêmes est ici parfaite, puisque les opérations considérées ne peuvent faire apparaître que des nombres rationnels parfaitement gérés par $\alpha_{\mathbb{Q}}$. Pour autant, on sera bien incapable de résoudre à l'aide du point fixe des systèmes linéaires dans $\mathbb{Q}$ en utilisant cette approximation.

Les approximations utilisées par Prolog IV

Notations : on note $\mathbb{F}$ l'ensemble des rationnels IEEE 754.

Définition 2.6 *$A \subset \mathbb{R}$ est* **convexe** *si :*
$\forall x \in A,\ \forall y \in A,\ \forall z \in \mathbb{R},\ x < z < y \Longrightarrow z \in A.$

Les ensembles convexes de $\mathbb{R}$ seront appelés **intervalles** *de $\mathbb{R}$.*

Propriété 2.7 *Il existe 10 formes d'intervalles dans $\mathbb{R}$:*

n^o	*Forme*	*Notation usuelle*
1	$\mathbb{R}$	$\mathbb{R}$ *ou* $(-\infty, +\infty)$
2	$\forall a \in \mathbb{R},\ \{x \in \mathbb{R},\ x > a\}$	$(a, +\infty)$
3	$\forall a \in \mathbb{R},\ \{x \in \mathbb{R},\ x \geq a\}$	$[a, +\infty)$
4	$\forall a \in \mathbb{R},\ \{x \in \mathbb{R},\ x < a\}$	$(-\infty, a)$
5	$\forall a \in \mathbb{R},\ \{x \in \mathbb{R},\ x \leq a\}$	$(-\infty, a]$
6	$\forall a \in \mathbb{R},\ \forall b \in \mathbb{R},\ \{x \in \mathbb{R},\ a \leq x \leq b\}$	$[a, b]$ *ou* $\{a\}$ *si* $a = b$
7	$\forall a \in \mathbb{R},\ \forall b \in \mathbb{R},\ \{x \in \mathbb{R},\ a < x \leq b\}$	$(a, b]$
8	$\forall a \in \mathbb{R},\ \forall b \in \mathbb{R},\ \{x \in \mathbb{R},\ a < x < b\}$	(a, b)
9	$\forall a \in \mathbb{R},\ \forall b \in \mathbb{R},\ \{x \in \mathbb{R},\ a \leq x < b\}$	$[a, b)$
10	$\emptyset$	$\emptyset$

La preuve de cette propriété se trouve à l'annexe 9.3.2, page 194.

On remarquera ici à nouveau, que, pour différentes raisons, on a retenu les notations anglo-saxonnes pour les intervalles.

Notation : Les a et b sont appelés **bornes de l'intervalle**.

Voici les définitions des deux approximations utilisées en Prolog IV :

Définition 2.8 *$\mathbb{I}$ est l'ensemble des intervalles de $\mathbb{R}$ à bornes dans $\mathbb{F}$. On parlera, lorsque l'on utilisera cette approximation, d'approximation par les intervalles simples (flottants ou IEEE).*

$\mathbb{I}_u$ est l'ensemble des unions d'éléments de $\mathbb{I}$. On parlera, lorsque l'on utilisera cette approximation, d'approximation par les unions d'intervalles (flottants ou IEEE).

Quelques remarques :

- Quand on utilisera $\mathbb{I}_u$, on se placera en fait dans le cadre d'une classe d'équivalence définie par la relation "Etre des sous-ensembles égaux". En d'autres termes, les éléments $[1, 2] \cup [2, 3]$ et $[1, 3]$ sont considérés comme le même élément de $\mathbb{I}_u$.

- Le cardinal de $\mathbb{I}$ est 9 151 455 195 278 868 484.
  Le cardinal de $\mathbb{I}_u$ est de l'ordre de $10^{2\,575\,727\,083}$.

- $\mathbb{I}$ et $\mathbb{I}_u$ sont des approximations finies sur $\mathbb{R}$, donc complètes (cf. propriété 2.5).

Propriété 2.9 *Les éléments d'approximation atomiques de $\mathbb{I}$ et $\mathbb{I}_u$ sont les suivants :*

1. *$\{f\}$, f étant un rationnel IEEE 754.*
2. *(f_1, f_2), f_1 et f_2 étant deux rationnels IEEE 754 consécutifs.*
3. *$(-\infty, f)$, f étant le plus petit rationnel IEEE 754.*
4. *$(f, +\infty)$, f étant le plus grand rationnel IEEE 754.*

La preuve de cette propriété se trouve à l'annexe 9.3.3, page 196.

Les éléments d'approximation atomiques de $\mathbb{I}$ et $\mathbb{I}_u$ seront appelés plus simplement **intervalles (flottants ou IEEE) atomiques**.
Ils sont au nombre de 8 556 380 163.

2.4.3 Résolution

Du problème initial au problème approché

Définition 2.10 Une relation *sur un ensemble E est un sous-ensemble quelconque de E.*
Une relation sur E^n sera également appelée relation n-aire sur E ou relation à n variables sur E.

Un problème *à n variables sur E est une famille finie de relations n-aire sur E.*
On dit qu'un tel problème a n variables.

Cette définition est un peu surprenante car elle suggère que tout problème à n variables ne contient que des relations n-aire, ce qui est bien sûr en général faux. L'objectif est en fait ici d'alléger les notations qui suivront. Il est en effet aisé d'étendre une relation p-aire en une relation n-aire (avec $p < n$).

Définition 2.11 *Soit r une relation n-aire sur E.*
Soit α une base d'approximation complète sur E.
On appelle **approximation de la relation** *r dans α la relation n-aire sur α notée $appx_\alpha(r)$ définie de la manière suivante :*
$appx_\alpha(r) = \{A_1 \times ... \times A_n \in \alpha^n \mid \exists B \subset r \text{ et } appx_\alpha(B) = A_1 \times ... \times A_n\}$.

Attention : même si une relation n-aire sur E est un sous-ensemble de E, son approximation dans α **n'est pas** l'approximation du sous-ensemble de E qui la représente.

Définition 2.12 *Soit $(P) = (r_i)_{i \in I}$ un problème sur E.*
Soit α une approximation complète sur E.
On appelle **approximation de** (P) **dans** *α le problème noté $appx_\alpha(P)$ définit de la manière suivante :*
$appx_\alpha(P) = (appx_\alpha(r_i))_{i \in I}$

$appx_\alpha(P)$ est donc un problème sur α. Cette remarque est très importante car elle met le doigt sur la différence fondamentale distinguant un problème réel de son approximation.

Définition 2.13 *Soit $(P) = (r_i)_{i \in I}$ un problème sur E à n variables.*
On appelle **solution de** (P) *tout élément s de E^n tel que :*

$$s \in \bigcap_{i \in I} r_i$$

L'ensemble $S = \bigcap_{i \in I} r_i$ de toutes les solutions de (P) est appelé **ensemble de solutions maximal**.

Si, de plus, S est solution de (P), alors S est appelée **solution maximale** *de (P).*

On pourra remarquer que dans $I\!R$, par exemple, il ne peut exister de solution maximale que si la solution est unique. On verra un peu plus loin qu'il n'en va pas de même lorsqu'on travaille sur des sous-ensembles de $I\!R^n$ comme éléments de base.

Il est de manière générale impossible (du moins pratiquement) de trouver les solutions d'un problème dans $I\!R$, et *a fortiori* l'ensemble de solutions maximal.

On va maintenant exhiber un algorithme dit "du point fixe", qui permet en un temps fini de trouver la solution maximale du problème approché tel qu'on vient de le définir. On montrera ensuite que la solution que l'on trouve ainsi possède de très intéressantes propriétés vis-à-vis du problème initial.

L'algorithme du point fixe

Soit $(P) = (r_i)_{i \in I}$ un problème à n variables sur $I\!R$.
Soit $appx_{I\!I}(P) = (appx_{I\!I}(r_i))_{i \in I}$ l'approximation de ce problème dans $I\!I$.

Soit S la suite de $I\!I^n$ définie récursivement de la manière suivante :

- $S_0 = I\!R^n$
- Supposons la suite S construite jusqu'au terme S_j.
  S'il existe $i \in I$ tel que $S_j \notin appx_{I\!I}(r_i)$,
  alors $S_{j+1} = Max\{A \in appx_{I\!I}(r_i) \mid A \subset S_j\}$,
  sinon S_j est le dernier terme de la suite.

Remarque : on aurait bien sûr le même algorithme en utilisant l'approximation $I\!I_u$.

Remarque : dans Prolog IV, l'opération permettant de calculer le terme suivant de la suite ne peut être effectuée que sur certaines relations connues du système. La liste exhaustive de ces relations, appelées relations de base, se trouve à la section 2.7.

Propriété 2.14 *L'ensemble $\{A \in appx_{I\!I} \mid A \subset S_j\}$ possède un maximum.*

La preuve de cette propriété se trouve à l'annexe 9.3.4, page 198.

Cette propriété permet de justifier la pertinence du passage d'un terme de la suite au suivant, c'est à dire l'existence du lien itératif dans l'algorithme de point fixe.

Propriété 2.15 *La suite S (et donc le nombre d'opérations élémentaires de l'algorithme du point fixe) est finie.*

La preuve de cette propriété se trouve à l'annexe 9.3.5, page 199.

Cette propriété est importante car elle assure la terminaison de l'algorithme du point fixe en un temps fini.

Remarque : comme on vient de le montrer, l'algorithme du point fixe se termine toujours en un temps fini. D'un point de vue pragmatique, il faut tout de même remarquer que pour un problème à n variables et p contraintes de base, le nombre de réévaluations effectuées durant la résolution (qui dépend directement de la taille de la suite S) peut atteindre 8 556 380 163 np, que ce soit avec des intervalles simples ou avec des unions d'intervalles. 8 556 380 163 est en fait la taille de la plus grande suite strictement décroissante de $\mathbb{I}^n$. Dans la pratique, cependant, la convergence du point fixe sera très rapide dans une écrasante majorité des cas.

Propriété 2.16 (Propriété fondamentale du point fixe) *Si S_{j_0} est le dernier terme de la suite S, alors S_{j_0} est la solution maximale du problème $appx_{\mathbb{I}}(P)$.*

La preuve de cette propriété se trouve à l'annexe 9.3.6, page 200.

Cette propriété est la plus importante et la plus fondamentale. Elle donne à la fois la sémantique exacte du résultat du point fixe, et prouve que celui-ci opère **la résolution exacte du problème approché**, comme annoncé en introduction.

On dispose maintenant d'une caractérisation très précise du résultat du point fixe, mais il nous manque encore quelques éléments permettant d'assurer le lien entre cette résolution et les solutions du problème initial.

Les deux propriétés suivantes seront les plus fondamentales dans l'exploitation pratique de l'algorithme du point fixe.

Liens entre solution du problème approché et solutions du problème initial

Soit $(P) = (r_i)_{i \in I}$ un problème sur $\mathbb{R}$ à n variables.
Soit $appx_{\mathbb{I}}(P) = (appx_{\mathbb{I}}(r_i))_{i \in I}$ l'approximation de (P) dans $\mathbb{I}$.

Propriété 2.17 *Soit* $(x_1, ..., x_n)$ *une solution de* (P).
Soit $I_1 \times ... \times I_n$ *la solution maximale de* $appx_{\mathbb{I}}(P)$ *(c'est à dire le résultat de l'algorithme du point fixe).*
Alors $x_1 \in I_1, \; ..., \; x_n \in I_n$.

La preuve de cette propriété se trouve à l'annexe 9.3.7, page 201.

Cette propriété confère à l'algorithme du point fixe sa principale caractéristique dans la résolution pratique de problèmes : toute solution du problème initial est incluse dans la solution maximale du problème approché calculée par l'algorithme du point fixe.

En des termes plus familier, **on a la garantie de ne jamais "perdre" de solution**.

On déduit donc immédiatement le corollaire suivant : si la solution maximale du problème approché est l'ensemble vide (et on peut en profiter pour remarquer que le problème approché a toujours au moins l'ensemble vide pour solution), alors le problème initial dans $\mathbb{R}$ n'a pas de solution. Ce corollaire permet de confirmer qu'un échec est bien une résolution parfaitement satisfaisante.

Propriété 2.18 *Si la solution maximale du problème approché est de la forme* $\{x_1\} \times ... \times \{x_n\}$,
alors $(x_1, ..., x_n)$ *est solution du problème initial.*

La preuve de cette propriété se trouve à l'annexe 9.3.8, page 201.

Cette propriété fait apparaître un des rares cas où la solution maximale du problème approché permet de conclure directement quant à l'existence d'une solution, en la donnant.

On pourra remarquer que ce cas ne peut se produire que si toutes les valeurs de la solution ont elle-même pour approximation dans $\mathbb{I}$. Dans le cas de Prolog IV, cela signifie, que ce soit avec les intervalles simples ou avec les unions d'intervalles, que toutes ces valeurs sont des rationnels IEEE 754.

Dans la pratique, on sera (ou on se ramènera par une énumération) très souvent dans ce cas en traitant des problèmes booléens, de domaines finis, ou entiers. On sera donc en général capable d'exhiber avec certitude les solutions de ces problèmes. Ce ne sera en général pas le cas des problèmes de nature continue.

2.5 Résolution dans une extension de $I\!R$

il y a énormément à dire et à faire sur le sujet que l'on va aborder ici (voir [?]). On ne va cependant pas en faire une présentation complète dans cette section pour plusieurs raisons. Tout d'abord, ce serait bien trop long, mais surtout, cela risquerait de perdre le lecteur ayant suivi les développements précédents.

En effet, cette approche est en grande partie concurrente de celle développée jusqu'ici. On se contentera donc d'en exposer une vision superficielle, et d'ébaucher les avantages que l'on peut tirer d'une telle approche.

Il est clair que la lecture de cette section ne s'impose pour aucune raison. Seule la curiosité peut vous conduire à la parcourir.

Définition de l'extension et première propriété

Soit $x \in I\!R$.
On définit x^+ et x^- de la manière suivante :

$$\forall y < x,\ \forall z > x,\ \ y < x^- < x < x^+ < z$$

On définit alors :

$$\overline{\overline{I\!R}} = I\!R \cup \{x^+, x \in I\!R\} \cup \{x^-, x \in I\!R\} \cup \{+\infty, -\infty\}$$

Intuitivement, on a ajouté des éléments plus proches des nombres réels que tout autre nombre réel, et on a donc, d'une certaine manière, séparé les nombres réels.

On peut alors observer une des propriétés fondamentales de cette extension de $I\!R$: le pouvoir d'expression des intervalles fermés de $\overline{\overline{I\!R}}$ est aussi fort que celui des intervalles quelconques de $I\!R$.

Par exemple, l'intervalle $[1^+, 2^-]$ de $\overline{\overline{I\!R}}$ contient autant d'information que l'intervalle $(1, 2)$ de $I\!R$ (car $[1^+, 2^-] \cap I\!R = (1, 2)$).

Cette propriété permet de ne plus considérer que les intervalles fermés lorsque l'on travaille dans l'extension de $I\!R$.

Autres propriétés

Cette extension de $\mathbb{R}$ possède également de bonnes propriétés dans la connexion des mécanismes de résolution avec la norme IEEE. En effet, celle-ci fait apparaître des concepts similaires au voisinage de 0, permettant de traiter facilement les phénomènes issus des divisions par zéro, par exemple.

Elle permet également de lier plus directement les notions d'arrondi par défaut ou par excès de la norme IEEE aux éléments fondamentaux de la structure sur laquelle on travaille.

Il faut cependant noter que certaines étapes très intuitives de raisonnement perdent de leur clarté car $\overline{\overline{\mathbb{R}}}$ n'est plus un corps ; cet ensemble ne conserve en fait que les propriétés d'ordre (il reste également archimédien et clos).

Influences de cette vision sur Prolog IV

Même s'il n'est pas nécessaire de le savoir, certains prédicats de Prolog IV héritent de la vision des choses que l'on vient d'ébaucher. Il s'agit des prédicats `eglb, elub`, et `ebounds` (le `e` signifie "Extended structure").

Ces prédicats permettent de connaître les bornes du domaine d'une variable dans l'extension de $\mathbb{R}$.

Pour connaître les modalités pratiques d'utilisation de ces prédicats, reportez-vous à la section 4.2.1, page 74.

2.6 Prolog IV: une calculatrice rationnelle

2.6.1 Introduction

L'aspect numérique de Prolog IVque nous allons évoquer dans cette section est décrit dans la sémantique générale de Prolog IV. A l'instar de l'algorithme de résolution avec les intervalles, il est la partie numérique d'un algorithme beaucoup plus général ; plus encore que celui-ci, il est difficile de l'identifier au cœur de l'algorithme sur les arbres qui l'abrite.

L'idée fondamentale de cet algorithme est pourtant assez simple. Comme on l'a vu précédemment, les seuls nombres manipulés directement et produits par le système de résolution avec les intervalles de Prolog IVsont les rationnels IEEE 754.

Or, si ces nombres possèdent diverses propriétés très intéressantes, il n'en demeure pas moins qu'ils sont en nombre fini, et que, par conséquent, une infinité de rationnels ne peuvent être représentés de manière optimale par ceux-ci. Ainsi, si $1, 2$, ou $1/2$ sont représentés exactement par les intervalles flottants, il n'en va pas de même pour $1/3, 1.1$, ou 10^{20}.

Les conséquences de cette inexactitude de la représentation des nombres rationnels par les intervalles flottants sont de plusieurs natures :

1. Comme le suggèrent les quelques exemples de nombres rationnels que l'on vient de citer, le système de résolution avec les intervalles aura un comportement différent suivant que les nombres apparaissant dans les calculs soient des rationnels IEEE 754 ou pas. Cette différence de comportement a toutes les chances de ne pas être comprise par l'utilisateur, à moins que celui-ci ne maîtrise parfaitement la mécanique Prolog IV, et la norme IEEE 754.

2. On souhaite pouvoir utiliser la grande quantité de relations disponibles pour effectuer de simples calculs. Il est alors heureux que ceux-ci soient effectués exactement sur les nombres rationnels, lorsque cela est possible.

3. On souhaite que lorsque toutes les variables d'une relation sont des constantes (rationnelles, donc), le système de résolution provoque un échec Prolog si elle n'est pas satisfaite.

On appellera ces trois propriétés respectivement l'**homogénéité rationnelle**, la **fonction calculatoire rationnelle**, et la **fonction de vérification rationnelle**.

2.6.2 Définition

Plusieurs possibilités de complexités diverses étaient envisageables, satisfaisant plus ou moins ces trois propriétés. L'algorithme que l'on a retenu nous a paru représenter le meilleur rapport qualité-coût possible. Essentiellement, il implante la fonction calculatoire rationnelle, couvrant parfaitement la fonction de vérification rationnelle, et partiellement l'homogénéité rationnelle.

La définition exacte de cet algorithme est simple :

Considérons une relation r, appliquée à des variables $X_1, ..., X_n$. Soit I un sous-ensemble de $1..n$ tel que $\forall i \in I$, X_i est une constante rationnelle.

Soit $(E_i)_{i \in 1..n}$ la famille de sous-ensembles de $\mathbb{R}$ définie par $E_i = X_i$ si $i \in I$, et $E_i = \mathbb{R}$ sinon.

Considérons alors pour tout $i \in 1..n$, l'ensemble $A_i = proj_i(r \cap E_1 \times ... \times E_n)$ (projection sur la i^{ieme} coordonnée).

Alors, pour tout $i \in 1..n$,

1. Si A_i est un nombre rationnel, on fige X_i à A_i.
2. Si $A_i = \emptyset$, alors le système de contraintes courant n'est pas satisfaisable.

Plus intuitivement, on va calculer pour chaque relation et en précision infinie, les valeurs des variables qui peuvent se déduire des variables déjà figées à des constantes rationnelles, en ignorant les domaines des autres variables.

Dans le tableau suivant, on pourra observer les modifications de comportement apportées par cette "calculatrice rationnelle".

Système	Sans calculette	Avec calculette
$\{X = 1/4 + 1/4\}$	$X = 1/2$	$X = 1/2$
$\{X = 1/3 + 1/3\}$	$X \in (0.6666666, 0.6666667)$	$X = 2/3$
$\{2/3 = 1 - 0.333333333\}$	true	false
$\left\{\begin{array}{l} X = Y + 1/3 \\ Y \in [1/3, 2/3] \end{array}\right\}$	$X \in (0.6666666, 1.0000001)$	$X \in (0.6666666, 1.0000001)$
$\{1/2 - 1/2 > 0\}$	false	false
$\{1/3 - 1/3 > 0\}$	true	false

Les deux derniers exemples montrent clairement que la propriété d'homogénéité rationnelle est partiellement respectée, mais l'exemple précédent reste un cas de non homogénéité.

2.6.3 Un traitement optionnel

Comme on vient de le voir, le traitement spécifique des constantes rationnelles que l'on a introduit possède de bonnes propriétés. Il a cependant deux défauts qui nous ont conduit à le rendre optionnel.

1. Dans certains cas, très rares, il peut poser de délicats problèmes de performance. Par exemple, si l'on calcule des puissances à l'aide d'une cascade de multiplications, l'évaluation d'un polynôme tel que X^{256} risque d'être coûteuse, tant en termes de mémoire que de temps. Dès lors que X devient constant, le système calcule une cascade de multiplications en précision infinie pouvant conduire à un résultat sous la forme d'un nombre rationnel dont numérateur et dénominateur comptent plusieurs centaines de chiffres.

2. La "calculatrice rationnelle" peut être vue dans certaines circonstances comme une verrue indésirable du système de résolution avec les intervalles. Ce sera en particulier le cas lorsque l'on s'attache tout particulièrement à étudier les intervalles seuls.

Il existe deux moyens pour activer ou désactiver la calculatrice rationnelle de Prolog IV:

1. Sur la ligne de commande, en utilisant l'option `-Q-calculator=on` pour lancer une session Prolog IVutilisant la calculatrice rationnelle, ou en utilisant l'option `-Q-calculator=off` pour lancer une session Prolog IVn'utilisant pas la calculatrice rationnelle (le mode par défaut est `on`).

2. Au cours d'une session Prolog IV, en utilisant :

   - `set_prolog_flag('Q-calculator',on).` pour activer la calculatrice rationnelle.
   - ou `set_prolog_flag('Q-calculator',off).` pour désactiver la calculatrice rationnelle.

   REMARQUE : Ces commandes n'ont d'influence que sur les contraintes que l'on va ajouter au système. Celles déjà posées continueront à être traitées de la même manière.

2.7 Les relations de Prolog IV

On va évoquer rapidement dans cette section les relations de base qui constituent le cœur de Prolog IV. On va premièrement en donner la liste exhaustive. On ne s'étendra pas sur leur définition précise (détaillée dans la documentation générale de Prolog IV), mais on insistera sur quelques points qui méritent peut-être des explications complémentaires.

2.7.1 La liste complète

Voici la liste des relations faisant intervenir le système de résolution avec les intervalles :

• • • abs/2
• • • and/2
• • • arccos/2
• • • arcsin/2
• • • arctan/2
• • • band/3
• • • bcc/4
• • • bco/4
• • bdif/3
• • beq/3
• • • bequiv/3
• bfinite/2
• • • bge/3
• • • bgt/3
• bidentifier/2
• • • bimpl/3
• binfinite/2
• • binlist/3
• • • bint/2
• • • ble/3
• bleaf/1
• blist/1
• • • blt/3
• bnidentifier/2
• • • bnint/2
• bnleaf/1
• bnlist/1
• • • bnot/2
• bnreal/2
• • • boc/4
• • • boo/4
• • • bor/3
• • • boutcc/4
• • • boutco/4
• • boutlist/3
• • • boutoc/4
• • • boutoo/4
• breal/2
• • • bxor/3
• • • cc/3
• • • ceil/2
• • • co/3
• • • cos/2
• • • cosh/2
• • • cot/2
• • • coth/2
• • dif/2
• • • div/3
• • eq/2
• • • equiv/2
• • • exp/2
• • • floor/2
• • • ge/2
• • • gt/2
• • if/4
• • • impl/2
• • index/3
• • inlist/2
• • • int/1
• • • le/2
• • • ln/2
• • • log/2
• • • lt/2
• • • max/3
• • • min/3
• • • minus/3
• • n/3
• • • nint/1
• • • not/1
• • • oc/3
• • • oo/3
• • • or/2
• • • outcc/3
• • • outco/3
• • outlist/2
• • • outoc/3
• • • outoo/3
• • • pi/1
• • • plus/3
• • • power/3
• • • root/3
• • • sin/2
• • • sinh/2
• size/2
• • • sqrt/2
• • • square/2
• • • tan/2
• • • tanh/2
• • • times/3
• • u/3
• • • uminus/2
• • • uplus/2
• xor/2

2.7.2 Relation sur les arbres et relations numériques

On va commenter ici les marqueurs (•) placés devant les relations. Ils permettent de distinguer trois catégories de relations :

1. Les relations concernant uniquement les intervalles (• • •) :
Elles sont purement numériques. Leurs variables sont immédiatement typées $real$, ou $listofreal$.

2. Les relations mixtes arbres/numérique (••) :
Ces relations sont, elles, *a priori*, gérées par le mécanisme général de réduction des domaines sur les arbres. Cependant, lorsque les domaines des variables sont suffisamment réduits, la relation est traitée comme une relation purement numérique. Considérons par exemple la relation $index(X, L, R)$ qui décrit l'ensemble des triplets (X, L, R) tels que X est le R^{ieme} élément de la liste L. Si tous les éléments de la liste L sont typés $real$, alors il est clair que X doit être $real$ également. De plus, R est naturellement $real$. Cette relation devient donc purement numérique.

3. Les relations partiellement numériques (•) :
Certaines de leur variables sont numériques, les autres sont, *a priori*, des arbres, mais elles ne peuvent devenir purement numériques. Le plus souvent, leur partie numérique est une variable booléenne permettant de qualifier une relation (cf. 2.7.4), comme par exemple $bfinite(B, A)$ qui décrit la liste des couples $(1, A)$, A étant un arbre fini, et des couples $(0, A)$, A étant un arbre infini. La seule exception est la relation $size$ qui fait intervenir une variable numérique (la taille de la liste), et une variable non numérique (la liste elle-même).

Il est clair que ce que l'on vient de dire ici a pour unique but de mieux faire comprendre le fonctionnement de Prolog IV, de la manière la plus intuitive possible. Cela ne s'oppose pas, bien entendu, à la sémantique du langage décrite dans la documentation générale.

Cette distinction entre la partie numérique et la partie plus générale de l'algorithme de résolution par réduction de domaines est cependant importante car le comportement de la partie numérique est en général plus fin que celui sur les arbres. Cette différence est due au fait que l'approximation des nombres réels est plus fine que celle des arbres, puisque l'algorithme de réduction des domaines est exactement le même.

Vérifions-le sur un exemple simple. Considérons le problème suivant :

$inlist(X, [toto, tata, tutu])$ $(Eq1)$ $(X \in \{toto, tata, tutu\})$
$inlist(X, [tata, tete, tyty])$ $(Eq2)$ $(X \in \{tata, tete, tyty\})$

Prolog IVne déduit rien de ces relations. Par contre, si l'on remplace $toto, tata, tutu, tete, tyty$ par $1, 2, 3, 4, 5$, on obtient alors le système de contraintes suivant :

$inlist(X, [1, 2, 3])$ $(Eq1)$ $(X \in \{1, 2, 3\})$
$inlist(X, [2, 4, 5])$ $(Eq2)$ $(X \in \{2, 4, 5\})$

Prolog IVdéduit alors que X est égal à 2.

2.7.3 Entiers et booléens en Prolog IV

Comme on le verra dans la suite de ce document, on va résoudre de nombreux problèmes portant sur des nombres entiers ou des booléens avec Prolog IV. Cependant, il ne faudrait pas croire que les entiers ou les booléens existent en tant que type dans Prolog IV. "Etre un entier" ou "Etre un booléen" sont des relations de base, et rien d'autre. On peut s'en apercevoir très facilement avec le système de contraintes $\{int(X), nint(X)\}$, qui contraint successivement X à être entier, et à ne pas l'être. Prolog IVest cependant incapable de déduire que ce système de contraintes n'a pas de solution. En particulier, "Etre un entier" n'est pas une relation correspondante à un sous-domaine privilégié de Prolog IV (cf. chapitre 2 de la documentation générale de Prolog IV). On peut observer que dans ce cas (par exemple $\{real(X), nreal(X)\}$ ou $\{identifier(X), nindentifier(X)\}$, etc ...) le système déduit l'impossibilité.

Pour être plus précis en ce qui concerne les booléens, il s'agit de variables pouvant prendre uniquement deux valeurs : 0 (sémantiquement équivalent à "faux") et 1 (sémantiquement équivalent à "vrai"). Contraindre une variable B à être booléenne consiste donc simplement à poser les contraintes $\{int(B), cc(B, 0, 1)\}$, ou bien encore $\{inlist(B, [0, 1])\}$. Pour cette raison, on appellera aussi les nombres booléens, ou les variables astreintes à valoir une valeur booléenne des "0/1".

La relation unaire "booléen", elle, n'existe pas explicitement, mais elle est automatiquement insérée dès lors qu'une variable apparaît dans une relation à une position où un booléen est attendu. Comme on le verra à la section 5.1, l'absence de type "booléen" ne gêne en rien l'expression des problème de nature booléenne, ni l'efficacité de leur résolution. On perd cependant l'aspect formel des booléens de Prolog III. Par contre, la capacité des booléens à se mélanger aux nombres réels, et plus généralement aux arbres, permet dans de nombreux cas de simplifier considérablement l'expression de problèmes, qu'ils soient uniquement booléens, ou qu'ils soient mixtes. D'autre part, l'existence des relations pseudo-booléennes (cf. 2.7.4) permet d'exprimer des contraintes impossibles ou très complexes à exprimer autrement.

Enfin, une dernière remarque : la résolution à l'aide d'une énumération des problèmes booléens et entiers (sous réserve que les valeurs soient raisonnablement petites, c'est à dire inférieures au million, grossièrement), est complète. C'est à dire que les solutions obtenues sont les bonnes solutions et qu'il n'y en a pas d'autres. Cette propriété (propriété 2.18 à la section 2.4.3) permet de conclure que dans ces cas particuliers très importants, le problème est totalement et exactement résolu.

2.7.4 Relations et booléens en Prolog IV

On va s'attarder ici sur les relations formées du préfixe b accolé au nom d'une autre relation. On appelle ce type de relations "relations pseudo-booléennes". Elles ne bénéficient d'aucun traitement particulier dans le système de résolution mais il est tout de même important d'insister un peu sur leur signification.

La relation $b < rel > (X_1, X_2, ..., X_n)$ est l'ensemble des n-uplets de l'une des deux formes suivantes :

1. $(1, X_2, ..., X_n)$ si $(X_2, ..., X_n) \in < rel >$.
2. $(0, X_2, ..., X_n)$ si $(X_2, ..., X_n) \notin < rel >$

Cette définition appelle quelques commentaires :

- La première variable des relations pseudo-booléennes est toujours un booléen, ou si l'on préfère un "0/1".

- Pour toute valeur de $X_2, ..., X_n$, il existe une, et une seule valeur de X_1. Cette propriété fait souvent penser à ces relations avant tout comme à une fonction dont la vocation serait de calculer X_1 ; impression confirmée par les formes fonctionnelles de ces relations. Ce sont cependant de vraies relations, et il peut bien sûr arriver que, X_1 étant figé (à 0 ou à 1), les domaines des autres variables soient réduits.

- On ne considère pas que les relations purement booléennes, comme *bxor* soient des relations pseudo-booléennes, même si elles répondent à la définition que l'on vient de donner. Elles appartiennent plutôt aux relations booléennes pures.

Ces relations particulières permettent d'exprimer de nombreuses contraintes, et notamment les systèmes de contraintes exprimés de façon logique, comme $\{X < -1 \textit{ ou } X > 1\}$.

3 Expression d'un problème en Prolog IV

La liste des relations de base disponibles en Prolog IV donne une bonne idée des contraintes que l'on peut exprimer facilement. Cependant, il ne faudrait pas perdre de vue les liens délicats existant entre l'expression d'un problème et la faculté de l'algorithme du point fixe et/ou de l'énumération à le résoudre.

On va survoler dans ce chapitre trois aspects de ces liens :

1. On étudiera tout d'abord quelques techniques usuelles de programmation, même si l'essentiel se trouve dans les exemples traités au chapitre 5.
2. On tentera ensuite de mettre en lumière un certain nombre de situations dans lesquelles l'efficacité de la résolution n'est pas à la hauteur de la facilité d'expression du problème, en évoquant des techniques permettant d'y remédier.
3. Enfin, on s'attardera ensuite sur les multiples facettes sous lesquelles on peut observer la collaboration entre le système de résolution et des mathématiques plus conventionnelles.

3.1 Exprimer un problème en Prolog IV

La conversion d'un problème mathématique en un programme Prolog IV n'impose en général aucune gymnastique particulière. Il existe cependant quelques situations isolées dans lesquelles on peut donner des conseils utiles.

Appartenance d'un vecteur à une liste de vecteurs

Considérons le problème suivant :

$$V \in (V_i)_{i \in 1..p}$$

V étant un vecteur de taille n, et $(V_i)_{i \in 1..p}$ étant une liste de vecteurs de taille n.

Comparons maintenant deux traductions possibles de cette contrainte à l'aide des relations de Prolog IV:

1. Version naïve:
$$inlist(V, [V_1, V_2, ..., V_p])$$

2. Version décomposée:
$$\forall j \in 1..n,\ index(V^j, [V_1^j, V_2^j, ..., V_p^j], R)$$

La deuxième version sera capable de déduire beaucoup plus d'informations que la première. Ce meilleur comportement n'est cependant pas une évidence car on aurait pu être tenté de penser que le fait que la contrainte doive être décomposée en contraintes de base est un handicap important.

Sur un exemple:

```
>> inlist([1,A],[[2,B],[2,C]]).

C ~ tree,
B ~ tree,
A ~ tree.

>> index(1,[2,2],R),index(A,[B,C],R).

false.
```

Si on regarde de plus près la contrainte naïve, on s'aperçoit que le sous-domaine de son deuxième argument est "liste de taille 2"; cette contrainte ne pourra avoir d'effet que si son deuxième argument devient complètement figé.

Le raisonnement que l'on vient de faire ici se reproduit si l'on souhaite exprimer qu'un arbre appartient à une liste d'arbres de même forme, mais avec des feuilles variables et/ou des étiquettes variables sur les nœds. Il est bien sûr particulièrement intéressant si les vecteurs sont numériques.

Cette technique est utilisée pour résoudre le problème du puzzle (cf. 5.2.2).

Expression de conditions

On va maintenant s'intéresser aux expressions attribuant une valeur à une variable en fonction d'une ou de plusieurs conditions dans divers cas :

- "X vaut A si $< Cond >$, 0 sinon".
  Si A est un réel, il y a deux traductions possibles :

  1. `X = if(Cond,A,0)`
  2. `X = Cond .*. A`

  On choisira plutôt la deuxième solution, plus efficace. Les deux possibilités ont le même comportement (toutes deux n'invoquent qu'une seule relation de base).

- "X vaut A si $< Cond >$, B sinon".
  Là aussi, si toutes les variables sont numériques, il y a deux possibilités :

  1. `X = if(Cond,A,B)`
  2. `X = Cond .*. A .+. bnot(Cond) .*. B`

  La situation est maintenant différente puisque la deuxième transcription fait intervenir plusieurs contraintes de base. Le problème qui en découle apparaît clairement sur l'exemple suivant :

```
>> X = if(Cond,1,2).

Cond ~ cc(0,1),
X ~ cc(1,2).

>> X = Cond .*. 1 .+. bnot(Cond) .*. 2 .

Cond ~ cc(0,1),
X ~ cc(0,3).
```

  La deuxième formulation fait des déductions moins intéressantes quant au domaine de la variable X.

- "X vaut A_1 si $< Cond_1 >$, ..., A_n si $< Cond_n >$", les conditions étant exclusives.
  On va proposer trois formulations en Prolog IV :

  1. `X = if(Cond1,A1,if(...,if(Condn,An,_))))...)`
     ou bien encore en exploitant l'exclusivité des relations :
     `X = if(Cond1,A1,if(...,if(Condn-1,An-1,An))))...)`
  2. `X = Cond1 .*. A1 .+. ... .+. Condn .*. An`
  3. `X = Cond1 .*. A1 .+. ... .+. Condn .*. An,`
     `inlist(X,[A1,...,An])`

  On peut montrer (voir le traitement de la relation if à la section 3.2.4) que les premier et troisième systèmes sont équivalents pour le système de résolutions avec les intervalles.

  Intuitivement, il manque à la deuxième formulation l'information qu'indépendamment de la condition qui est valide, X vaut l'une des valeurs A_i, information apportée par la contrainte `inlist`.

3.2 Expression et résolution

Les relations de base de Prolog IV, associées au mécanisme de décomposition du compilateur permettent d'exprimer un grand nombre de contraintes de manière évidente. Cependant, il y a une grande différence entre la capacité à exprimer une contrainte et la capacité à la résoudre de manière satisfaisante ou efficace.

L'étape qui conduit de l'expression mathématique du problème à un programme Prolog IV n'est donc pas innocente.

La localité inhérente à l'algorithme de résolution avec les intervalles, la décomposition des contraintes qui ne correspondent pas à des relations de base, ainsi que l'approximation des nombres réels engendrent ce que l'on pourrait appeler des malentendus. On va ici essayer de les dissiper en les isolant le plus précisément possible. On tentera ensuite d'ébaucher des solutions permettant de contourner les difficultés qu'ils font apparaître.

3.2.1 Les causes de malentendus

Que se cache-t-il sous ce titre étrange ?

Comme on l'a déjà dit à plusieurs reprises, un certain nombre de contraintes ou de systèmes de contraintes d'apparence triviales ne sont pas résolus de manière satisfaisante par l'algorithme du point fixe. Le but de cette section est de détailler d'un point de vue pratique ces cas de faiblesses apparentes. En connaissant leur nature et en sachant les identifier, on sera alors capable d'ouvrir des voies permettant d'y remédier. Les sous-sections suivantes ébaucheront plusieurs méthodes dans ce sens.

Il est clair que le système basé sur le point fixe que l'on vient de définir n'est pas complet. On va distinguer ici trois causes plus spécifiques à l'origine d'insuffisances du point fixe. On montrera sur quelques exemples leurs conséquences sur le comportement du système.

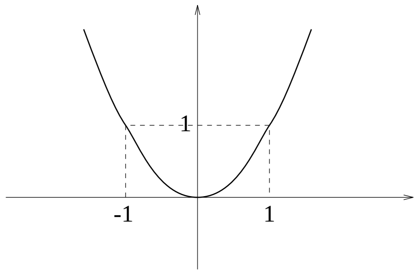
Relations non-convexes

Définition 3.1 *Un sous-ensemble E, de $\mathbb{R}^n$ est dit convexe si pour tout couple de points de E, le segment de droite déterminé par ces deux points est inclus dans E.*

Définition 3.2 *Une relation r, de $\mathbb{R}^n$ est dite à projections convexes si les projections selon tous les axes de son intersection avec tout pavé de $\mathbb{R}^n$ sont convexes.*

L'approximation que réalisent les intervalles flottants simples est convexe, et donc, l'approximation d'ensembles non-convexes est très pauvre. Il apparaît donc que la coexistence de cette approximation et des relations à projections non-convexes sera une source de pertes d'informations. Considérons par exemple le système de contraintes suivant :

$$\begin{cases} X^2 = 1 \\ (X = 0) \text{ } ou \text{ } (A = 1) \end{cases}$$



Fonction $Y = X^2$

Le point fixe s'avère incapable de déduire que A vaut 1. En effet, l'équation $X^2 = 1$ admet pour solutions -1 et 1. Donc la projection de cette relation est $\{-1\} \cup \{1\}$. Mais cet ensemble n'est pas convexe, donc l'approximation ne peut le représenter correctement. En fait, le domaine de X sera réduit à $[-1, 1]$, ce qui ne permet pas de conclure quant au fait que X vaille ou non 0. La valeur de A reste donc indéfinie.

Variables à occurrences multiples

Il convient pour comprendre ce problème de bien garder en mémoire le fait qu'aucune manipulation formelle n'est jamais faite lors du point fixe. Pas même la plus simple : l'identité. Cette faiblesse, inhérente aux mécanismes de réduction de domaines va engendrer un mauvais comportement dans le traitement des expressions où une même variable apparaît plusieurs fois.

Parmi les exemples les plus flagrants, on peut citer les systèmes de contraintes : $\{X \neq X\}$ et $\{Y = X - X\}$. Dans ces deux cas en effet, d'apparence triviale, le point fixe est incapable de déduire quoi que ce soit.

Décomposition de contraintes complexes en relations primaires

On touche ici au cœur du problème. Comme on l'a vu dans la description du point fixe, l'utilisation d'une relation dans le système impose le calcul de son intersection avec les pavés de $I\!R^n$. On n'est cependant pas toujours capable de la calculer. Et on sait encore moins le faire de manière automatique avec une relation initialement inconnue. La procédure classique pour traiter des relations complexes va alors consister à les décomposer en relations de base. Ces relations de base constituent le noyau du système. Bien choisies, elles permettent de décomposer toutes les relations complexes.

On peut donner l'exemple classique suivant :
Dans un système restreint aux nombres entiers, considérons la contrainte : "X_1, X_2, X_3 sont deux à deux différents". Si l'on ne dispose que d'un "dif" binaire, cette contrainte sera décomposée en trois contraintes de base : $\{X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3\}$.
Si les domaines des trois variables sont réduits à $[1, 2]$, il est clair que le système n'aura pas de solution (trois valeurs différentes prises parmi deux). Le point fixe sera cependant incapable de détecter cette insatisfaisabilité.

Enfin, sans être directement lié à la localité de l'algorithme de résolution, certaines relations peuvent poser des difficultés quand on utilise les unions d'intervalles :

Relations infiniment non-convexes

Définition 3.3 *Une relation r, de $I\!R^n$ est dite infiniment non-convexe s'il existe un pavé de $I\!R^n$ dont l'intersection avec la relation une union infinie de sous-ensembles disjoints de $I\!R^n$.*

Parmi les relations de base de Prolog IV, deux types de relations appartiennent à cette catégorie : les relations trigonométriques $(cos, sin, ...)$, et les relations entières $(int, nint, ...)$.
Bien que les unions d'intervalles flottants ne permettent pas l'apparition d'unions infinies d'intervalles, ces relations risquent cependant de créer des unions d'intervalles de très grande taille.

Prenons par exemple la relation suivante : $\{int(X)\}$.

En unions d'intervalles, le domaine de X est :
$(-\infty, -16777216] \cup \{-16777215\} \cup \{-16777214\} \cup ...$
$\cup\{16777214\} \cup \{16777215\} \cup [16777216, +\infty)$
Soit une union d'intervalles composée de quelque 33 554 433 sous-ensembles. On remarquera qu'il n'y a pas de "trous" en deçà de -16 777 216, ni au-delà de 16 777 216. C'est du au fait que dans ces zones, les nombres flottants (qui sont ici tous des entiers) deviennent moins fréquents que les entiers eux-mêmes. Résultat : tout intervalle atomique dans ces zones contient au moins un entier, et aucun ne peut donc être éliminé.

3.2.2 Regroupement des sous-expressions identiques

De quoi s'agit-il ?

Comme on l'a déjà dit et répété, le mécanisme de résolution avec les intervalles ignore tout aspect formel d'un système de contraintes. Cette remarque fondamentale va nous conduire à introduire une technique de programmation très simple et naturelle, dont les conséquences sur l'efficacité de la résolution peuvent être très importantes.

Prolog étant un langage très adapté aux manipulations formelles, les idées que l'on va présenter ici peuvent être intégrées dans un programme Prolog IV de pré-traitement formel automatique des systèmes de contraintes.

Nous allons en voir le principe sur un exemple simple.
Considérons le système de contraintes : $\{X + Y > 0, X + Y < 0\}$.

Voici très exactement les relations de base Prolog IV qui traduisent ce système : $\{plus(A, X, Y), gt(A, 0), plus(B, X, Y), lt(B, 0)\}$.

En supposant que les domaines des variables soient tous égaux à $\mathbb{R}$, les seules déductions que le point fixe pourra faire sont : $A \in (0, +\infty), B \in (-\infty, 0)$.

C'est ici qu'apparaît clairement l'absence de relation formelle dans le système. En effet, rien ne permet de lier l'expression $X + Y$ de la première équation à l'expression $X + Y$ de la deuxième équation. Ce lien manquant explique que le mécanisme du point fixe soit incapable de déduire l'insatisfaisabilité de ce système de contraintes.

C'est justement ce lien formel que l'on se propose ici de recréer. On peut d'ailleurs constater sur l'exemple précédent qu'il suffit de rajouter l'égalité de A et B pour réaliser ce lien.

En revenant au niveau du système de contraintes initial, on va remplacer $\{X+Y>0, X+Y<0\}$ par $\{A=X+Y, A>0, A<0\}$, ce qui permet immédiatement de conclure. On a en fait regroupé deux sous-expressions identiques sous un même label pour réaliser leur égalité formelle dans le système de contraintes.

On va voir dans l'exemple suivant les conséquences pratiques que peut avoir ce genre de manipulations très simples sur la résolution de systèmes de contraintes. D'autres exemples plus complexes viendront appuyer cette démonstration dans les chapitres 5 et 6.

Démonstration sur un petit système de contraintes polynômiales

Le petit système algébrique que nous allons présenter ici apparaît dans plusieurs articles dédiés à la résolution de systèmes de contraintes continus ([10, 16]).

Voici le système de contraintes :

$$\begin{array}{l} X_1+X_2+X_3+X_4=1 \\ X_1+X_2-X_3+X_4=3 \\ X_1^2+X_2^2+X_3^2+X_4^2=4 \\ X_1^2+X_2^2+X_3^2+X_4^2-2X_1-3=0 \\ \forall i \in 1..4,\ X_i \in -10..10 \end{array}$$

Ce problème possède deux solutions :

$$\left\{\begin{array}{l} X_1 = 1/2 \\ X_2 \simeq -0.151388 \\ X_3 = -1 \\ X_4 \simeq 1.65138 \end{array}\right. \text{ et } \left\{\begin{array}{l} X_1 = 1/2 \\ X_2 \simeq 1.65138 \\ X_3 = -1 \\ X_4 \simeq -0.151388 \end{array}\right.$$

Voici sa traduction naïve en Prolog IV ([miniPoly]):

```
constraints([X1,X2,X3,X4]):-
   cc(X1,-10,10),
   cc(X2,-10,10),
   cc(X3,-10,10),
   cc(X4,-10,10),
   X1 .+. X2 .+. X3 .+. X4 = 1,
   X1 .+. X2 .-. X3 .+. X4 = 3,
   square(X1) .+. square(X2) .+. square(X3) .+. square(X4) = 4,
   square(X1) .+. square(X2) .+. square(X3) .+. square(X4)
              .-. 2.*.X1 .-. 3 = 0.

>> constraints(L),realsplit(L,greatest_domain).
```

Cette requête produit 11 pseudo-solutions (cf. 4.1) pour deux solutions "mathématiques" du problème. Il y a donc 9 solutions parasites, néanmoins très proches des vraies solutions. L'option "greatest_domain" est décrite dans la section dédiée aux prédicats d'énumération de Prolog IV (cf. section 4.2).

Transformons maintenant ce problème en regroupant les expressions :

$$\begin{cases} X_1 + X_2 + X_4 \\ X_1^2 + X_2^2 + X_3^2 + X_4^2 \end{cases}$$

On obtient alors le programme suivant ([miniPolyS]) :

```
constraints([X1,X2,X3,X4]):-
   cc(X1,-10,10),
   cc(X2,-10,10),
   cc(X3,-10,10),
   cc(X4,-10,10),
   SX .+. X3 = 1,
   SX .-. X3 = 3,
   SQX = 4,
   SQX .- 2.*X1 .- 3 = 0,
   % Sub-expressions
   SX = X1 .+. X2 .+. X4,
   SQX = square(X1) .+. square(X2) .+.
```

```
        square(X3) .+. square(X4).

>> constraints(L),realsplit(L,greatest_domain).

L ~ [1/2,cc(-'>0.15138792',-'<0.1513877'),
    -1,cc('>1.6513876','<1.651388')];

L ~ [1/2,oo('>1.6513876','<1.651388'),
    -1,oo(-'>0.15138792',-'<0.1513877')].
```

Cette requête ne produit plus que deux pseudo-solutions correspondant aux deux solutions "mathématiques" du problème. Il n'y a donc plus de solutions parasites. De plus, son exécution est environ 18 fois plus rapide.

Cet exemple montre donc bien l'intérêt de ce type de manipulation de nature formelle.

3.2.3 Relations infiniment non-convexes

Cette section ne concerne que le fonctionnement avec les unions d'intervalles.

Les relations infiniment non-convexes de Prolog IV ont toutes une intéressante propriété : si les domaines des variables sont relativement restreints, la taille des unions d'intervalles générés devient raisonnable.

Par exemple, un système de contraintes tel que $\{int(X), cc(X,0,10)\}$ va générer une union de 11 intervalles. Cependant, si la relation $\{int(X)\}$ est évaluée avant la relation $\{cc(X,0,10)\}$, alors une union d'intervalles démesurée aura été créée de manière transitoire. Il convient donc de placer dans ce type de situation la relation cc en premier.

Cette mise en ordre est à la charge du programmeur, sauf si celui-ci a recours aux pseudo-termes de Prolog IV pour exprimer ces contraintes. C'est alors le compilateur Prolog IV qui va placer les relations dans un ordre tel que leur exécution ne génère pas d'unions d'intervalles colossales, si un tel ordre existe.

Voici quelques exemples de requêtes en Prolog IV (avec les unions d'intervalles) :

```
>> int(X),cc(X,0,10).

error: heap_overflow
>> cc(X,0,10),int(X).

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>> X ~ cc(0,10) n int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>> X ~ int n cc(0,10).

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
```

L'erreur de débordement ci-dessus est la conséquence de la tentative de création d'une union d'intervalles trop grande. Ce débordement n'est cependant pas inéluctable. Avec suffisamment de mémoire, cette requête peut aboutir. Un tel travail n'aurait cependant aucun intérêt.

3.2.4 Contraintes redondantes

Soit (P) un problème portant sur des variables réelles. Soit (S) un ensemble de contraintes portant sur les variables de (P) tel que l'ensemble des solutions de $(P \cup S)$ soit égal à l'ensemble des solutions de (P). On dit alors que les contraintes de (S) sont redondantes.

Considérons maintenant la résolution de (P) et de $(P \cup S)$ par le système de résolution avec les intervalles. Il est clair que les contraintes qui sont redondantes pour le problème initial ne le seront pas nécessairement pour le problème approché. On va distinguer trois cas :

1. Les solutions de (P) et de $(P \cup S)$ sont les mêmes, et le comportement du point fixe est similaire. Les contraintes de (S) sont redondantes pour le problème initial, comme pour le problème approché. Elles sont donc inutiles.
   Exemple : soit $(P) = \{X < Y, Y < Z\}$, et $(S) = \{X < Z\}$.

2. Les solutions de (P) et de $(P \cup S)$ sont les mêmes, mais le comportement du point fixe est meilleur (convergence plus rapide). Ce cas est assez rare, et justifie l'emploi des contraintes redondantes, bien qu'elles soient également redondantes pour le système approché.
   Exemple : $(P) = \{int(X)\ n\ nint(X)\ n\ cc(0,1000)\}$, et $(S) = \{false\}$.

3. Les solutions de (P) et de $(P \cup S)$ sont différentes. Les contraintes de (S) ne sont pas redondantes pour le système approché, et leur emploi peut être très utile.
   Exemple : soit $(P) = \{X = -X\}$, et $(S) = \{2X = 0\}$.
   Cet exemple est traité à la section 2.3. D'autres programmes utilisent des contraintes redondantes (cf. 5.2.1, 5.3.4).

L'emploi de contraintes redondantes peut aller jusqu'à conférer au système de résolution avec les intervalles un comportement complet sur certains problèmes. On verra par exemple à la section 6.3.2 que l'ajout de certaines contraintes redondantes peut donner aux intervalles la possibilité de résoudre directement (sans énumération) des systèmes de contraintes purement linéaires, sous réserve que tous les nombres apparaissant dans les calculs soient des rationnels IEEE 754 . Dans tous les cas, le résultat obtenu sera une approximation très satisfaisante du résultat.

Traitement de la relation if

On va illustrer l'emploi des contraintes redondantes sur l'implantation de la relation if définie comme suit :

$$if = \{(X,1,X,Y),\ \forall(X,Y) \in Tree^2\} \cup \{(X,0,Y,X),\ \forall(X,Y) \in Tree^2\}$$

On va essayer d'écrire cette relation en fonction d'autres contraintes. Il est clair que le système de contraintes suivant est mathématiquement équivalent à la contrainte $\{if(X,C,Y,Z)\}$: $\{X = (C * Y) + (bnot(C) * Z)\}$. Cependant, ces deux systèmes sont différents du point de vue du système de résolution avec les intervalles, comme le montrent les deux requêtes suivantes :

```
>> X = if(Cond,1,2).
```

```
Cond ~ cc(0,1),
X ~ cc(1,2).

>> X = Cond .*. 1 .+. bnot(Cond) .*. 2 .

Cond ~ cc(0,1),
X ~ cc(0,3).
```

On va maintenant montrer qu'en ajoutant au deuxième système la contrainte mathématiquement redondante $\{inlist(X,[Y,Z])\}$, on va obtenir un système de contraintes équivalent du point de vue du système de résolution avec les intervalles.

Soit X, C, Y, Z des variables dont les domaines sont tels que le système $\{if(X,C,Y,Z)\}$ n'ait pas de solution dans le système de résolution approché. Montrons que le système $\{X = (C*Y) + (bnot(C)*Z),\ inlist(X,[Y,Z])\}$ n'a pas de solutions dans le système de résolution approché.

Distinguons pour cela trois cas :

1. $D(C) = \{0\}$. Alors, on a :

   $$D(X) \times \{0\} \times D(Y) \times D(Z) \notin \{(X,0,Y,X),\ \forall(X,Y) \in Tree^2\}$$

   Donc $D(X) \cap D(Z) = \emptyset$.
   Mais, dans le deuxième système, la contrainte $\{X = (C*Y)+(bnot(C)*Z)\}$ devient $\{X = Z\}$, donc permet au système de déduire que $D(X) = D(Z)$.
   Donc le deuxième système de contrainte échoue.

2. $D(C) = \{1\}$.
   La démonstration est similaire au cas précédent.

3. $D(C) = [0,1]$. Alors, on a :

   $$D(X) \times D(C) \times D(Y) \times D(Z) \notin \{(X,0,Y,X),\ \forall(X,Y) \in Tree^2\}$$

   et

   $$D(X) \times D(C) \times D(Y) \times D(Z) \notin \{(X,0,Y,X),\ \forall(X,Y) \in Tree^2\}$$

Donc $D(X) \notin D(Y)$ et $D(X) \notin D(Z)$. Mais alors, c'est la contrainte $\{inlist(X, [Y, Z])\}$ qui provoque un échec du système de résolution.

On a donc bien construit un système de contraintes équivalent (du moins, du même niveau de complétude) dans le système de résolution approché en ajoutant une contrainte mathématiquement redondante à un système de contraintes.

3.2.5 Les contraintes globales

Comme on a pu le constater, seule la résolution d'une contrainte correspondant à une relation de base est complète (c'est à dire que tout système composé d'une relation de base et d'inclusions dans des intervalles flottants qui n'a pas de solution provoque un échec Prolog IV). La section 6.2 donne une méthode permettant d'obtenir un comportement complet au niveau d'une contrainte, même si celle-ci ne correspond pas à une relation de base.

Cependant, dans de nombreux cas, on ne veut pas se contenter du comportement, mais on souhaite surtout programmer l'évaluation d'une relation de manière efficace. Si on reprend l'exemple de la contrainte "Etre des entiers tous différents", il est clair que ce n'est pas tant le comportement qui nous intéresse (une énumération donne facilement à cette contrainte un comportement complet), mais bien l'efficacité du traitement. En effet, alors qu'une énumération permettra d'obtenir un traitement de ce problème dans un temps croissant exponentiellement en fonction du nombre de variables, des algorithmes plus fins permettront un traitement polynômial, sous réserve que l'on utilise pour approximation de $\mathbb{R}$ les intervalles simples.

Ces contraintes sont souvent appelées "contraintes globales" car elles ont en général pour objectif de traiter une somme de contraintes primaires sur un ensemble de variables en tirant partie de la globalité de point de vue dont on dispose alors.

Il est donc naturel de souhaiter disposer de certaines relations de ce type, bénéficiant d'une implantation la plus fine possible.

La possibilité d'ajouter des relations de base dans Prolog IV, en décrivant totalement l'algorithme de réévaluation sera probablement disponible dans une prochaine version de Prolog IV, ainsi qu'un certain nombre de contraintes globales d'intérêt général ou destinées à résoudre plus efficacement des problèmes spécifiques (cf. section 7.2).

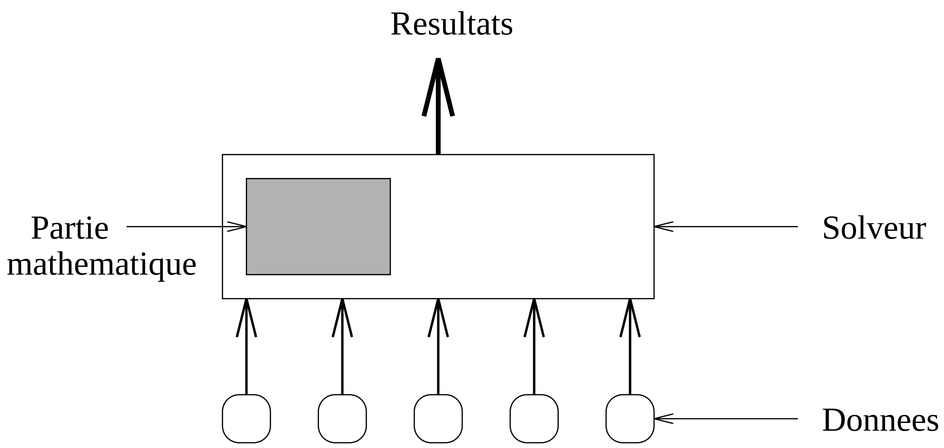
3.3 Mathématiques et CLP

Introduction

L'objet de cette section est plus philosophique que technique. On va ici poser un doigt sur un aspect inhabituel de la programmation (logique) avec contraintes, surtout pour les spécialistes de la programmation d'algorithme mathématiques.

Bien souvent, pour faciliter la résolution de problèmes possédant une structure mathématique sous-jacente importante, on utilise des algorithmes issus de méthodologie mathématiques (dérivation de fonctions, résolution de systèmes linéaires, etc...). Cette utilisation reste cependant "clandestine", en ce sens que l'algorithme fait partie intégrante du système, restant en général invisible du point de vue de l'utilisateur; seul le concepteur du système y a accès.

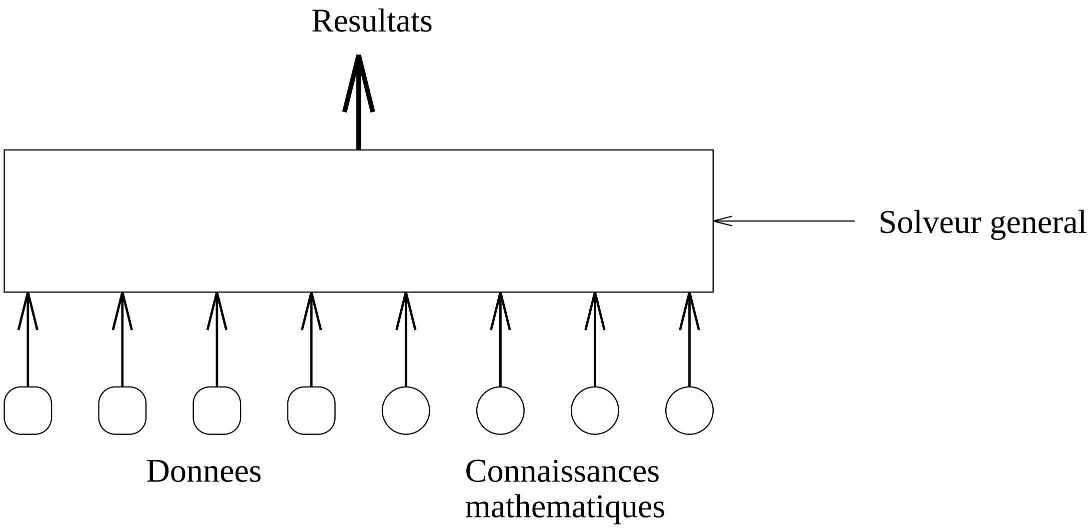
Si l'on symbolise le système de résolution comme une machine qui prend des données en entrée, et produit des résultats en sortie, on peut alors obtenir le fonctionnement schématique suivant :



Une autre vision

Il semble que l'approche "Contraintes" nous permette d'envisager le problème sous un angle légèrement différent : le système de résolution est alors vu comme une machine très générale, capable de gérer l'information au sens large, c'est à dire aussi bien les données elles-mêmes que les connaissances mathématiques dont on dispose. Les méthodes et techniques provenant d'horizons divers sont alors considérées comme une partie des données ; elles ne font plus partie à proprement dire du système de résolution.

En reprenant le schéma précédent, on peut obtenir les modifications suivantes :



4 Techniques d'énumération

Dans cette section, on va introduire la notion d'énumération. Elle est indissociable de la notion d'intervalle, car indispensable à la résolution de pratiquement tous les problèmes. Nous montrerons dans un premier temps son utilité, puis ses principes de base. On détaillera ensuite les prédicats disponibles en Prolog IV, pour énumérer ou pour construire sa propre énumération. Enfin, nous conclurons cette section en montrant comment l'on peut étendre cette notion d'énumération à la recherche de minima et de maxima sur un problème contraint.

4.1 Principe et sémantique

4.1.1 Pourquoi énumérer ?

Reprenons l'un des systèmes de contraintes introduits à la section 2.3 :
$\{X = -X\}$.

Comme on a pu le constater, aucune déduction ne peut être faite par le système de résolution.

Au contraire, les systèmes de contraintes $\{X = -X, X \geq 0\}$ et $\{X = -X, X < 0\}$ sont, eux, résolus de manière satisfaisante. Hors, justement, du point de vue du système de contraintes, la résolution de ces deux systèmes est équivalente à la résolution du système unique $\{X = -X\}$. Plus précisément, l'union des solutions des deux systèmes est égale à l'ensemble des solutions du système de contraintes initial.

Comme on le voit très clairement sur cet exemple, la décomposition du problème initial a permi sa résolution en deux étapes. C'est là le principe fondamental de toute énumération.

4.1.2 Principes de base

Formalisme

On peut formaliser le principe de toute énumération de la manière suivante :

Soit (P) un problème sur un espace E.
Soit $(E_i)_{i \in I}$ une partition de E.
Si $S(P, E)$ représente l'ensemble des solutions de (P) dans E, alors :

$$S(P, E) = \bigcup_{i \in I} S(P, E_i)$$

Un premier algorithme d'énumération

Pour des raisons de performance, et non de principe, il est naturel et très important d'utiliser cette méthode de "découpage" de manière récursive. En effet, sans récursivité, le "découpage" de l'espace le plus fin consisterait à faire intervenir tous les éléments de l'espace considéré dans la "décomposition".

L'intérêt de ce principe récursif est flagrant dans l'exemple que l'on vient de traiter, puisqu'il n'a pas été nécessaire de "décomposer" l'ensemble des nombres réels en autant de sous-ensembles qu'il contient d'éléments (heureusement!). Il a suffi au contraire de partitionner l'espace une seule fois en deux, la récursivité permettant alors de réitérer ce processus de découpage pour obtenir une partition plus fine si les découpages précédents n'ont pas suffi.

Une autre remarque très importante est la suivante :

Certains partitionnements sont parfaitement inutiles étant donné le système de résolution sous-jacent. Plus précisément, tout partitionnement faisant intervenir des nombres autres que des rationnels IEEE 754 sera approché puisque les bornes des domaines des variables ne peuvent être que des rationnels IEEE 754.

Supposons par exemple que l'on travaille sur un problème (P), dont une variable a un domaine réduit à l'intervalle (a, b), et supposons qu'il n'existe pas de rationnel IEEE 754 dans cet intervalle (a et b sont deux rationnels IEEE 754 consécutifs). Soit c tel que $a < c < b$. Alors, $(P) \cup \{X \in (a, c]\}$, et $(P) \cup \{X \in (c, b)\}$ auront les mêmes solutions que (P) puisque l'approximation de ces deux intervalles est égale à (a, b). Ce partitionnement sera donc inutile.

Il est donc clair qu'il est suffisant de choisir des partitionnements ne faisant intervenir que des rationnels IEEE 754.

Voici donc un premier algorithme d'énumération récursif cherchant des pseudo-solutions d'un problème (P) dans un espace E :

$ENUM(P, E)$
    $POINT\_FIXE(P, E)$
    Si le point fixe démontre que (P) est insatisfaisable sur E
        Retourner $\emptyset$
    Sinon
        Si E est partitionnable (utilement)
            Soit $(E_i)_{i \in I}$ une telle partition
            Retourner $\bigcup_{i \in I} ENUM(P, E_i)$
        Sinon
            Retourner E

Cette algorithme décrit un arbre que l'on appellera "arbre de recherche" ou "arbre d'énumération". Les feuilles de cet arbre seront soit des échecs du point fixe (pas de solution), soit un état non-partitionnable (utile).

Un algorithme d'énumération plus sophistiqué

Comme on l'a vu, la notion que l'on utilise pour décider de la "partitionnabilité" est liée à la précision des rationnels IEEE 754. On peut cependant imaginer que l'on ne souhaite pas une telle précision, ou plus généralement, que l'on souhaite piloter plus finement la condition de terminaison de l'algorithme récursif d'énumération.

On obtient alors l'algorithme suivant :

$ENUM(P, E)$
    $POINT\_FIXE(P, E)$
    Si le point fixe démontre que (P) est insatisfaisable sur E
        Retourner $\emptyset$
    Sinon
        Si E est partitionnable (utilement)
        et que la $CONDITION\_D\_ARRET$ n'est pas atteinte
            Soit $(E_i)_{i\in I}$ une telle partition
            Retourner $\bigcup_{i\in I} ENUM(P, E_i)$
        Sinon
            Retourner E

A propos de cette condition d'arrêt

L'usage le plus courant que l'on fait de cette condition d'arrêt concerne la précision. Dans le premier algorithme que l'on a décrit, la précision est nécessairement l'intervalle atomique, lequel est naturellement à l'origine de la notion de non-partitionnabilité. L'utilisateur n'est cependant pas forcément attaché à cette notion de précision optimale du système, et peut, par exemple, se satisfaire d'une solution à 10^{-5} près, ou bien à 100 flottants près.

On gagne ainsi du temps d'exécution, et, dans bien des cas, on limite la croissance du nombre de solutions parasites.

A propos du partitionnement

Il existe deux types particuliers de partitionnement, très couramment utilisés :

1. La dichotomie.
   Comme son nom l'indique, il s'agit de procéder à des découpages en deux parties. Ce découpage s'effectuera donc dans une seule dimension,

sur une seule variable. Il peut s'exprimer par des contraintes de la forme : $\{X < a\}$, et $\{X \geq a\}$.
On distinguera encore deux sous-cas :

(a) La dichotomie centrale, consistant à partitionner le domaine de la variable en deux parties égales (cette notion d'égalité pouvant être variable).

(b) La dichotomie par un bord, consistant à distinguer l'un des bords du reste de l'intervalle. Il est clair que ce type de découpage n'a d'intérêt que si l'on opère dans un domaine discret.

2. L'énumération proprement dite.
Elle consiste en fait à scinder le domaine d'une variable en chacun de ses éléments, ceux-ci étant en nombre fini, ou au plus dénombrable. Cette technique a donné son nom au processus en général car l'on peut toujours s'y ramener dans la mesure où, les intervalles flottants étant en nombre fini, on peut très bien les énumérer.
Dans la pratique cependant, on s'attachera plutôt à énumérer des domaines "petits", comme des domaines finis, ou des entiers raisonnablement bornés.

Comme on l'a dit, ces deux techniques ne travaillent que sur une variable à la fois, et font apparaître un des éléments fondamentaux de ce type de méthodes : le choix de cette variable.

L'énumération standard de Prolog IV

Nous allons maintenant décrire l'énumération standard de Prolog IV. Les énumérations effectuées par les prédicats décrits à la section 4.2, et utilisés aux chapitres 5 et 6 obéissent à ce schéma.

Cette énumération est en fait une dichotomie dont l'algorithme peut s'écrire de la manière suivante :

$ENUM\_ST(P)$
    $POINT\_FIXE(P)$

Si le point fixe démontre que (P) est insatisfaisable
    Retourner $\emptyset$
Sinon
    S'il existe une variable de (P) de domaine non atomique
    et que la CONDITION_D_ARRET ne soit pas atteinte
        $X = CHOIX\_VARIABLE(P)$
        $DECOUPER\_DOMAINE(Dom(X), D_1, D_2)$
        Retourner $ENUM\_ST(\{P, X \in D_1\})$
                $\cup ENUM\_ST(\{P, X \in D_2\})$
    Sinon
        Retourner le produit cartésien des domaines
        des variables de (P).

Cet algorithme est paramétré par trois facteurs :

1. La condition d'arrêt.
2. Le choix de la variable.
3. La façon de couper un domaine en deux.

On choisit le plus souvent la variable dont le domaine est le plus petit. Cette méthode est en fait l'implantation la plus simple dans le cadre des intervalles du concept général souvent appelé "first fail", dont le principe est le suivant : choisir `a priori` la variable la plus contrainte (ici, on assimile cette notion à la taille du domaine), de manière à agir le plus efficacement possible sur l'ensemble du système de contraintes. Plus "philosophiquement", son créateur disait "si tu veux réussir, essaie d'abord là où tu as le plus de chance d'échouer".

Comme on le verra à la section 6.4, il y a beaucoup à dire à propos du choix de l'ordre d'énumération des variables.

Ecriture d'une énumération en Prolog IV

Voici le programme Prolog IV correspondant à l'algorithme précédent, avec les choix suivants :

1. La condition d'arrêt : les domaines de toutes les variables contiennent moins de 1000 intervalles flottants atomiques.
2. Le choix de la variable : la première variable dont le domaine contient plus de 1000 intervalles flottants atomiques.
3. Méthode de découpage du domaine : le domaine de la variable choisie est découpé en une partie gauche et une partie droite de telle sorte que ces deux parties contiennent à peu près autant d'intervalles flottants atomiques.

Les méta-prédicats qui sont utilisés ici sont décrits dans la section 4.2.1. D'autre part, cette énumération est prédéfinie en Prolog IV, il faut utiliser : `realsplit(L,list_order,1000,float_mode)` (cf. 4.2.2).

```
mon_enum(L):-
   condition_d_arret(L),
   !.
mon_enum(L):-
   choix_variable(L,X),
   milieu(X,M),
   decouper(X,M),
   mon_enum(L).

condition_d_arret([]).
condition_d_arret([X|L]):-
   float_size(X,le(1000)),
   condition_d_arret(L).

choix_variable([X|L],X):-
   float_size(X,gt(1000)),
   true,
   !.
choix_variable([X|L],Y):-
   choix_variable(L,Y).

milieu(X,M):-
   bounds(X,Min,Max),
```

```
   float_rank(Min,RMin),
   float_rank(Max,RMax),
   float_rank(M,floor((RMin.+.RMax)./.2)).

decouper(X,M):-
   le(X,M).
decouper(X,M):-
   gt(X,M).
```

Quelques remarques :

- Le point fixe n'apparaît pas explicitement dans le programme Prolog IV. Celui-ci fait partie intégrante du langage, et il est exécuté automatiquement à chaque ajout de contrainte dans le système. Ici, le point fixe est exécuté après `le(X,M)` et `gt(X,M)` dans le prédicat `decouper`.
- A la différence de la description formelle de l'algorithme d'énumération standard, on ne va pas calculer l'union des solutions, mais simplement les afficher toutes par le biais des solutions Prolog.
- Cette énumération ne peut fonctionner correctement que si les variables de la liste L n'ont pas de borne infinie. Ce cas n'est en effet pas traité dans le prédicat `milieu`.

A propos des solutions de l'énumération

Le processus d'énumération consiste à enlever des sous-ensembles de l'espace de recherche dans lesquels le mécanisme du point fixe permet d'affirmer qu'il n'y a pas de solution. A l'instar du point fixe, rien ne garantit en général qu'il y ait une solution du problème dans les produits cartésiens d'intervalles produits par l'énumération. Par contre, bien sûr, toute solution du problème initial est incluse dans la -ou l'une des- solutions de l'énumération.

Les solutions de l'énumération seront appelées **pseudo-solutions**, et celles ne contenant aucune "vraie" solution (solution mathématique du problème initial) **solutions parasites**. Dans la majorité des cas, ces solutions parasites sont proches d'une vraie solution. Il arrive cependant que des solutions parasites soient très éloignées de vraies solutions. Cela arrive notamment dans les deux cas suivants :

1. Les solutions parasites à l'infini.
2. Les solutions parasites consécutives à des "quasi-zéros" de fonctions. Ces quasi-zéros correspondent à des endroits où une fonction est trop proche de 0 pour que, avec la précision des nombres flottants, on puisse dire si, oui ou non, elle vaut effectivement 0.

Un peu de vocabulaire

Le vocabulaire qui entoure les processus d'énumération est riche, et souvent peu clair. On a vu par exemple qu' "énumération" a un sens précis, doublé d'un sens plus général.

Voici donc quelques uns des termes les plus couramment utilisés en français et en anglais :

Nom	Sens
Enumération	Précis : Décomposition du domaine d'une variable en chacun de ses éléments. Généralement utilisé pour des problèmes entiers. Réalisé en Prolog IV par le prédicat *enum*. Générique : Nom donné à l'ensemble des techniques.
Dichotomie	Méthode d'énumération reposant sur le découpage en deux du domaine d'une des variables à chaque pas.
Affinage	Nom donné plus particulièrement aux techniques d'énumération ayant pour objet de réduire le domaine des variables, mais en fournissant un seul résultat (cf. section 6.6, et 6.7).
Split	Dichotomie
Enumerate	Enumération
Shrink Narrow Squeeze	Noms génériques (plutôt affinage)

4.1.3 Sémantique

Cette section a pour but essentiel de démontrer que, à l'instar de l'algorithme du point fixe, et sous certaines conditions, les énumérations produisent des solutions répondant à une caractérisation bien précise.

On en tirera deux déductions importantes : tout d'abord, cette caractérisation nous donne une définition exacte du résultat obtenu, ensuite, elle permet de conclure que toutes les énumérations ont le même résultat, et que seule leur efficacité varie.

Pré-requis : il est conseillé d'avoir lu la section décrivant la sémantique du système de résolution (cf. 2.4).

Intuitivement, la caractérisation des solutions de l'énumération s'appliquera à toutes les formes d'énumérations complètes (c'est à dire n'ignorant aucune partie de l'espace de recherche), et atomique (c'est à dire considérant chaque élément atomique de l'approximation, si nécessaire).

Elle permet alors d'affirmer que les solutions de l'énumération forment l'ensemble des sous-ensembles atomiques tels que la solution maximale de la restriction du problème à ce sous-ensemble n'est pas vide (c'est à dire en fait est lui-même).

Le terme "énumération" recouvre dans le langage courant un ensemble mal défini de techniques diverses.

En reprenant les concepts introduits dans ce chapitre, on va donner une définition précise d'une énumération. On va pour cela se ramener à une vision "à plat", c'est à dire ne prenant pas en considération la technique de programmation utilisée (itérative, récursive, ou autre ...), mais se focalisant sur le résultat de l'ensemble de l'opération d'énumération.

Si (P) est un problème sur E, et A un sous-ensemble de E, on note $(P)_{|A}$ la restriction de (P) à A.

Définition 4.1 *Soit (P) un problème sur E, à n variables.*
On appelle **énumération de** (P) *deux familles :*
$(F_i)_{i \in Fail}$ *et* $(S_i)_{i \in Succeed}$ *telles que :*

1. *Pour tout $i \in Fail$, la solution maximale de $appx_{\Pi}(P)_{|F_i}$ est l'ensemble vide.*

2. *Pour tout $i \in Succeed$, la solution maximale de $appx_{\Pi}(P)_{|S_i}$ n'est pas l'ensemble vide.*

On appelle **solution de l'énumération** *l'union des solutions maximales des problèmes $appx_{\Pi}(P)_{|S_i}$ pour tout $i \in Succeed$.*

Définition 4.2 *Soit $\xi = ((F_i)_{i \in Fail}, (S_i)_{i \in Succeed})$ une énumération d'un problème (P) sur E, à n variables.*

ξ est dite **complète** *si* $\left(\bigcup_{i \in Fail} F_i\right) \cup \left(\bigcup_{i \in Succeed} S_i\right) = E^n$.

ξ est dite **atomique** *si, pour tout $i \in Succeed$, S_i est un élément d'approximation atomique.*

Quelques remarques :

- En termes clairs, une énumération consiste à scinder l'espace de recherche en zones dans lesquelles l'algorithme du point fixe est capable de démontrer qu'il n'y a pas de solution, et en zones dans lesquelles il peut y en avoir.

- Cette partition de l'espace est celle qui est faite de manière récursive dans ce que l'on a appelé "énumération standard de Prolog IV". C'est une énumération complète. Si, de plus, on a choisi une énumération en précision atomique, on obtient alors une énumération complète atomique.

- On remarquera que, dans le cas d'une énumération atomique, la solution de l'énumération est plus simplement $\bigcup_{i \in Succeed} S_i$.

- Enfin, tout ce qui vient d'être dit, et que l'on va, dans le cadre de ce document, utiliser essentiellement dans un cadre numérique, est valable dans n'importe quel espace. Il est donc, entre autre, possible d'utiliser des techniques et raisonnements similaires en travaillant sur les arbres de Prolog IV.

Propriété 4.3 *Soit (P) un problème réel à n variables.*

Toutes les énumérations complètes et atomiques ont exactement pour solution l'union des éléments d'approximation atomiques tels que la restriction de $appx_{\Pi}(P)$ correspondante ait une solution maximale non-vide; c'est-à-dire l'ensemble des solutions atomiques du problème approché.

La preuve de cette propriété se trouve à l'annexe 9.3.9, page 202.

4.2 Méta-prédicats de Prolog IV

On va présenter dans cette section tous les prédicats de Prolog IV relatifs à la méta-programmation des intervalles, ainsi que quelques outils accessoires.

On abordera dans un premier temps les méta-prédicats de bas niveau. Ils permettent d'écrire en Prolog IV de nombreuses méthodes d'énumération, y compris celles qui sont décrites dans un deuxième temps, qui représentent une petite bibliothèque de méthodes standard que l'on aura toujours sous la main, et qu'il ne sera pas nécessaire de re-programmer.

Tous les prédicats qui vont être détaillés ici sont également décrits sommairement dans la documentation générale des prédicats de Prolog IV. On va tâcher ici de leur donner une définition plus précise, liée aux concepts que l'on vient de développer, et de donner pour chacun quelques exemples significatifs.

4.2.1 Méta-prédicats de bas niveau

<u>Bornes du domaine d'une variable</u>

- `lub(Var,Bsup)`
- `glb(Var,Binf)`
- `bounds(Var,Binf,Bsup)`

Ces prédicats permettent de connaître les bornes des domaines des variables numériques. `lub` (Least Upper Bound) rend la borne supérieure, `glb` (Greatest Lower Bound) rend la borne inférieure, et `bounds` rend les deux bornes. Ce sont des rationnels IEEE 754 (cf. 2.2). Ces prédicats échouent si la (ou une) borne n'est pas définie (c'est à dire si le domaine numérique de la variable n'est pas borné). Si la variable `Var` n'est pas compatible avec le type `real`, alors une erreur de type est générée.

Quelques remarques :

1. Ces prédicats ne permettent pas de connaître les types des bornes du domaine des variables (ouvert ou fermé), puisque le résultat rendu ne tient pas compte du fait que la borne soit atteinte. En d'autres termes, le résultat rendu est le plus petit intervalle fermé contenant le domaine de la variable.

2. Si `Var` est une constante rationnelle, ces prédicats rendent la ou les bornes du plus petit intervalle qui la contient.

Exemples et erreurs :

```
>> bounds(cc(0,10),M1,M2).
M2 = 10,
M1 = 0.

>> bounds(1/3,M1,M2).

M2 = 11184811/33554432,
M1 = 5592405/16777216.

>> glb(cc(0,10),M).
M = 0.

>> lub(lt(10),M).
M = 10.

>> glb(X,M).
false.

>> lub(toto,X).
error: error(type_error(real,toto),bounds/3)
```

Bornes du domaine d'une variable dans l'extension des réels

- `elub(Var,Bsup)`
- `eglb(Var,Binf)`
- `ebounds(Var,Binf,Bsup)`

Ces prédicats permettent de connaître les bornes des domaines des variables numériques, dans l'extension des réels (cf. 2.5). `elub` rend la borne supérieure, `eglb` rend la borne inférieure, et `ebounds` rend les deux bornes. Les bornes rendues sont des rationnels IEEE 754 étendus. Leur représentation en Prolog IV se fait de la manière suivante :

- X (un rationnel IEEE 754) représente le réel étendu X.
- `up(X)` et `dn(X)` représentent respectivement les réels étendus qui succèdent et précèdent immédiatement `X` (qui est un rationnel IEEE 754).
- `pinfinity`, et `minfinity` représentent les deux infinis.

Si la variable `Var` n'est pas compatible avec le type `real`, alors une erreur de type est générée.

Quelques remarques :

1. On rappelle que les bornes dans l'extension des réels existent toujours, et qu'elles représentent toujours des intervalles fermés.
2. Ces prédicats permettent indirectement de connaître les types des bornes du domaine d'une variable : `up(X)` ou `dn(X)` permettent de conclure que la borne concernée n'est pas atteinte dans $\mathbb{R}$ (cf. 2.5).
3. Si la variable `Var` est une constante rationnelle, ces prédicats rendent la ou les bornes du plus petit intervalle qui la contient.

Exemples et erreurs :

```
>> ebounds(cc(0,10),M1,M2).
M2 = 10,
M1 = 0.

>> ebounds(oo(0,10),M1,M2).
M2 = dn(10),
M1 = up(0).

>> elub(X,M).
M = pinfinity,
X ~ real.

>> eglb(toto,M).
error: error(type_error(real,toto),bounds/3)
```

A propos des rationnels IEEE 754

Les prédicats suivants sont très utiles pour manipuler les rationnels IEEE 754.

- `max_float(F)`
- `min_positive_float(F)`
- `float_rank(F,R)`

`float_rank` calcule le rang d'un flottant donné, ou le flottant d'un certain rang (cf. 2.2). Si `F` et `R` sont tous les deux libres, alors une erreur d'instanciation est générée; si `F` n'est pas compatible avec un rationnel IEEE 754, ou si `R` n'est pas compatible avec le rang d'un flottant, alors une erreur de type est générée.

`min_positive_float(F)` et `max_float(F)` unifient `F` respectivement avec le plus petit rationnel IEEE 754 positif, et le plus grand rationnel IEEE 754. On pourra noter que ces deux nombres ne sont pas l'inverse l'un de l'autre car la densité des nombres flottants est plus importante au voisinage de 0 qu'au voisinage de l'infini; à cause des nombres flottants dénormalisés (cf. 2.2).

Exemples et erreurs :

```
>> max_float(F),float_rank(F,R).
R = 2139095039,
F = 340282346638528859811704183484516925440.

>> min_positive_float(F),float_rank(F,R).
R = 1,
F = 1/713623846352979940529142984724747568191373312.

>> float_rank(340282346638528859811704183484516925441,R).
error: error(type_error('ieee754(single)',
340282346638528859811704183484516925441), float_rank/2)

>> float_rank(F,1/2).
error: error(type_error('ieee754(single) rank',1/2),
float_rank/2)

>> float_rank(F,R).
error: error(instantiation_error,float_rank/2)

>> float_rank(toto,R).
error: error(type_error('ieee754(single)',toto),float_rank/2)
```

Taille des domaines des variables réelles

Les trois prédicats suivants ont pour vocation de calculer la "dimension" du domaine d'une variable numérique. Ils représentent chacun une manière de calculer cette "dimension".

- `real_size(Var,Size)`
- `int_size(Var,Size)`

- `float_size(Var,Size)`

`real_size` calcule la mesure dans $I\!R$ du domaine de la variable `Var`. Essentiellement, si l'on considère les intervalles simples, c'est la différence entre la borne supérieure et la borne inférieure du domaine de la variable. `int_size` calcule le nombre d'entiers contenus dans le domaine de la variable `Var`. Dans les deux cas précédents, `pinfinity` est rendu si la taille est infinie. Enfin, `float_size` calcule le nombre d'intervalles atomiques composant le domaine de la variable `Var`. Pour mémoire, un intervalle atomique est soit un rationnel IEEE 754, soit un intervalle dont les deux bornes sont deux rationnels IEEE 754 consécutifs, et qui est ouvert des deux côtés (cf. propriété 2.9). Cette taille est toujours finie. Ces trois prédicats rendent des erreurs de type si la variable `Var` n'est pas compatible avec le type `real`.

Exemples et erreurs avec des intervalles simples :

```
>> real_size(cc(1,3),S).
S = 2.

>> int_size(cc(1,3),S).
S = 3.

>> float_size(cc(1,3),S).
S = 25165825.

>> int_size(toto,S).
error: error(type_error(real,toto),int_size/2)
```

Exemples avec des unions d'intervalles :

```
>> real_size(cc(1,3) n int,S).
S = 0.

>> int_size(cc(1,3) n int,S).
S = 3.

>> float_size(cc(1,3) n int,S).
S = 3.
```

4.2.2 Méta-prédicats de haut niveau

Enumération des nombres entiers

- `enum(X)`
- `enum(X,Min,Max)`

Le prédicat `enum` énumère les entiers et les unifie avec la variable `X`. Bien sûr, les entiers n'appartenant pas au domaine de la variable `X` sont éliminés *a priori* de ce processus. Si un `Min` et un `Max` sont précisés, l'énumération est limitée aux entiers compris entre `Min` et `Max`. `Min` et `Max` peuvent être des variables. Dans ce cas, les contraintes $\{X \leq Max, X \geq Min\}$ sont ajoutées au système de contraintes.

Le prédicat `enum` est le pendant dans les intervalles du prédicat `enumlin`, qui prend en compte les renseignements apportés par le solveur linéaire.

Ce prédicat rend une erreur de type si la variable `X` n'est pas compatible avec le type `real`.

Exemples et erreurs :

```
>> enum(X,1,3).
X = 1;

X = 2;

X = 3.

>> cc(X,1,3),enum(X).
X = 1;

X = 2;

X = 3.

>> enum(toto).
```

```
error: error('Real variable expected',enum)
```

Enumération d'une liste de booléens

- `boolsplit(List)`
- `boolsplit(List,First)`

Le prédicat `boolsplit` énumère une liste de variables booléennes, en utilisant la technique d'énumération standard (cf. 4.1.2). Le choix des variables se fait toujours dans l'ordre de la liste `List`. Bien sûr, lorsque le processus prend fin, toutes les variables de la liste `List` sont figées, puisque les variables booléennes ne peuvent prendre que deux valeurs (cf. 2.7.3). Si l'option `First` est précisée, alors la première valeur essayée pour chaque variable booléenne est `First`, sinon, c'est 0 (faux). `First` ne peut donc valoir que 0 ou 1.

Exemples et erreurs :

```
>> boolsplit([X,Y]).
Y = 0,
X = 0;

Y = 1,
X = 0;

Y = 0,
X = 1;

Y = 1,
X = 1.

>> boolsplit([X],0).
X = 0;
```

```
X = 1.

>> boolsplit([X,bnot(X)]).
X = 0;

X = 1.

>> boolsplit(X).
error: error('Real variable list expected',boolsplit)

>> boolsplit([]).
true.

>> boolsplit([toto]).
error: error('Boolean list expected',boolsplit)

>> boolsplit([X],2).
error: error('Second argument can only be 0 or 1',boolsplit)
```

Enumérations de listes d'entiers ou de réels

- `intsplit(L)`
- `intsplit(L,Choice)`
- `intsplit(L,Choice,Stop)`

- `realsplit(L)`
- `realsplit(L,Choice)`
- `realsplit(L,Choice,Stop)`
- `realsplit(L,Choice,Stop,Mode)`

Les prédicats `intsplit` et `realsplit` énumèrent des listes de variables réelles, en utilisant la technique d'énumération standard (cf. 4.1.2). Voici les différents paramètres :

- Choice
  - Méthode de choix de la variable dont on souhaite couper le domaine en deux à chaque étape.
  - Méthodes de choix pré-définies : `smallest_domain` : choix de la variable ayant le plus petit domaine (si la condition d'arrêt est `prec(P)`, parmi les variables dont le domaine est plus grand que la précision requise `P`), `greatest_domain` : choix de la variable ayant le plus grand domaine, `list_order` : choix de la prochaine variable dont le domaine a une dimension plus grande que la précision requise, dans l'ordre de la liste `L`.
  - Utilisation d'une méthode de choix définie par l'utilisateur : on peut utiliser `my_choice`. Le prédicat `my_choice(L,C)` sera appelé durant l'énumération. `L` sera alors la liste des variables que l'on énumère, et le prédicat devra unifier `C` avec l'une des variables de la liste `L`, dont le domaine n'est pas atomique (cf. propriété 2.9, 4.1.2).
  - La méthode choisie par défaut est `smallest_domain`.
- Stop
  - Condition d'arrêt du processus d'énumération.
  - Conditions d'arrêt pré-définies : `depth(N)` : en fonction de la profondeur de l'arbre d'énumération, `prec(P)` ou `P` : précision (taille du plus grand domaine). `P` est alors nécessairement une constante rationnelle.
  - Utilisation d'une condition d'arrêt définie par l'utilisateur : on peut utiliser `my_stop`. Le prédicat `my_stop(L)` sera appelé durant l'énumération. Un échec de ce prédicat signifie que la condition d'arrêt n'est pas atteinte.
  - La condition d'arrêt choisie par défaut est 0.0001 (`real_mode`) ou 1000 (`float_mode`).

- Mode
  - (Uniquement pour `realsplit`): permet de choisir une méthode de calcul de la dimension des domaines. Ce choix conditionne la détermination du plus petit ou du plus grand intervalle, les conditions d'arrêt liées à la précision, ainsi que la manière de couper les intervalles en deux. On pourra remarquer qu'une condition d'arrêt exprimée en `real_mode` sera absolue (c'est à dire indépendante de l'ordre de grandeur des variables), alors qu'en `float_mode`, elle sera relative.
  - Modes possibles: `real_mode` (correspondant à `real_size`), et `float_mode` (correspondant à `float_size`).
  - Le mode par défaut est `real_mode`.

Quelques remarques :

1. Si des variables libres sont utilisées pour les différentes options, elles sont unifiées avec le choix par défaut retenu.
2. Le découpage des intervalles se fait toujours à gauche d'abord. C'est grâce à cette particularité que l'on peut implanter très simplement à partir de ces prédicats des méthodes de recherche d'optimum (cf. 4.3).

Exemples et erreurs :

```
>> cc(X,0,3),intsplit([X]).
X = 0;

X = 1;

X = 2;

X = 3.

>> cc(X,0,10),intsplit([X],_,3).
X ~ cc(0,2);
```

```
X ~ cc(3,5);

X ~ cc(6,8);

X ~ cc(9,10).

>> cc(X,1,2),cc(Y,1,2),intsplit([X,Y],list_order).
Y = 1,
X = 1;

Y = 2,
X = 1;

Y = 1,
X = 2;

Y = 2,
X = 2.

>> cc(X,1,2),cc(Y,1,2),intsplit([X,Y],smallest_domain).
Y = 1,
X = 1;

Y = 2,
X = 1;

Y = 1,
X = 2;

Y = 2,
X = 2.

>> cc(X,0,3),int(X),realsplit([X],_,0).
X = 0;

X = 1;
```

```
X = 2;

X = 3.

>> intsplit([nint n cc(0,100)]).
error: error('Integer list expected',boolsplit)

>> cc(X,0,3),intsplit([X],_,0).
error: error('Stop condition << 0 >> is non-sens',_split)

>> cc(X,0,1),realsplit([X],_,0.5).
X ~ cc(0,0.5);

X ~ oc(0.5,1).

>> cc(X,1,2),realsplit([X],_,5.0e+6,float_mode).
X ~ cc(1,1.5);

X ~ oc(1.5,2).

>> cc(X,1,2),realsplit([X],_,depth(2)).
X ~ cc(1,1.25);

X ~ oc(1.25,1.5);

X ~ oo(1.5,1.75);

X ~ cc(1.75,2).

>> realsplit([X],_,_,toto_mode).
error: error('Mode should be real_mode or float_mode',
realsplit)

>> realsplit([X],_,-1).
error: error('Stop condition << -1 >> is non-sens','_split')
```

4.2.3 Quelques utilitaires

Les prédicats suivants donnent accès à des statistiques spécialisées concernant les intervalles :

- `reset_istats`
- `get_istats(P,E)`
- `get_istats(P,E,M)`

Fonctionnant de manière similaire à `reset_cpu_time`, `reset_istats` remet trois compteurs a zéro :

1. Le nombre de points fixes exécutés : `P`
2. Le nombre de réévaluations de relations primaires effectuées dans le point fixe : `E`
3. La taille de la plus grande union d'intervalles : `M`

Ces prédicats peuvent s'avérer très utiles pour comprendre le fonctionnement du système, et tout particulièrement pour évaluer le comportement des différentes techniques d'énumération. On pourra considérer que `P` est sensiblement égal au nombre de noeuds de l'arbre d'énumération, sous réserve que l'on ne recourt pas à des calculs intermédiaires utilisant les intervalles. Une autre information intéressante est le ratio `E/P` : plus il est important, plus le mécanisme du point fixe joue un rôle prépondérant dans la résolution; réciproquement, plus ce ratio est faible, plus c'est le mécanisme d'énumération qui joue un rôle important.

Exemples et erreurs :

```
>> reset_istats.

true.

>> X.*.Y = 101, intsplit([X,Y]).
```

```
Y = -1,
X = -101;

Y = -101,
X = -1;

Y = 101,
X = 1;

Y = 1,
X = 101.

>> get_istats(P,E).

E = 395,
P = 29.
```

4.3 Optimisation

Introduction

Les procédés d'énumération dont on a décrit le principe, et les divers méta-prédicats de haut niveau que l'on a présentés dans ce chapitre ont principalement pour objet la recherche de solutions isolées.

Il y a cependant deux types de recherche très courants qui ne sont pas traités par cette méthodologie :

1. La recherche de limite de zones de l'espace qui ne se réduisent pas à des points singuliers. Comment obtenir des informations intéressantes sur un problème tel que $\{Y > X^2\}$?

2. La recherche d'un optimum d'une variable dans un système contraint.

On va s'intéresser ici au deuxième problème, qui est mieux défini. Les techniques que l'on va introduire pourront alors servir à certains problèmes de recherche de zones, particulièrement si celles-ci se présentent sous forme de pavés de $\mathbb{R}^n$.

Une méthode simple

On va proposer ici une méthode très simple permettant de calculer une approximation du minimum d'une variable. On va pour cela utiliser la remarque du 4.2.2 à propos des prédicats *intsplit* et *realsplit* : les découpages des variables sont toujours faits de gauche à droite.

Soit (P) un problème portant sur une liste de variables L, X une de ces variables. Pour calculer une approximation de la borne inférieure de X, il suffit d'énumérer X, puis d'énumérer L.

La première solution trouvée en énumérant X correspondant à une solution pour L nous permettra d'exhiber un minorant de la borne inférieure de X.

Voici comment on peut écrire ce prédicat d'énumération en Prolog IV :

```
lower_boundi(X,L,M):-
   record(lower_boundi_sol,0),
   lower_boundi_eval(X,L,M).
lower_boundi(_,_,_):-
   recorded(lower_boundi_sol,0),
   !,
   throw(error("Problem has no solution",lower_boundi)).
lower_boundi(_,_,M):-
   recorded(lower_boundi_val,M).


lower_boundi_eval(X,L,M):-
   realsplit([X]),
   glb(X,MinX),
   record(lower_boundi_val,MinX),
   realsplit(L),
   !,
   record(lower_boundi_sol,1),
   fail.
```

Le prédicat `lower_boundi(X,L,M)` calcule et rend dans `M` un minorant du minimum de X dans le système de contraintes défini par la liste de variables `L`.

Quelques remarques :

- Pour éviter tout malentendu, il faut préciser le sens du résultat obtenu. Il s'agit bien d'un minorant du minimum, puisque dans le cas contraire, cela signifierait qu'il existe une solution avec un `X` inférieur à `M`, donc qu'une des solutions du problème ne serait pas incluse dans une des solutions de l'énumération ; ce qui est impossible (cf. 4.1).
  Attention : si la borne inférieure du domaine de `X` est bien un minorant du minimum de `X` après son énumération par `realsplit`, il y a trois erreurs à ne pas commettre :
  1. La borne supérieure du domaine de `X` N'EST PAS UN MAJORANT du minimum de `X`. Le seul majorant que l'on peut calculer est celui du maximum de `X`, et ce calcul n'a de sens, en ce qui concerne le minimum de `X`, que si le système de contraintes a au moins une solution.
  2. A moins d'utiliser une énumération en précision maximale (atomique), la borne inférieure du domaine de `X` n'a plus aucun sens particulier après énumération de la liste `L`. C'est pour cela que `record(lower_boundi_val,MinX)` est placé avant.
  3. Utiliser le minorant du minimum de `X` calculé par ce type de méthode pour affecter ensuite ce résultat à `X` est susceptible de provoquer un échec.
- Enfin, il est bien sûr possible d'utiliser toutes les options décrites à la section 4.2 pour paramétrer les deux énumérations apparaissant dans ce programme.

Un exemple de requête :

```
>> Y = square(X), cc(X,-100,100), lower_boundi(Y,[X],M).

M = 0,
X ~ cc(-100,100),
Y ~ cc(0,10000).
```

Le "Branch & bound" usuel

La technique que l'on vient d'introduire est assez simple et peut s'avérer efficace, mais on peut bien sûr avoir recours à des techniques plus traditionnelles comme le "branch & bound", dont le principe est le suivant :

On parcourt l'arbre d'énumération de manière usuelle, mais en s'assurant que, sur chaque nœud de l'arbre, la variable que l'on cherche à minimiser est contrainte à valoir moins que la plus petite valeur déjà trouvée.

Une telle énumération peut s'écrire de la manière suivante :

```
minimizei(V,L,M):-
   record(b_and_b_min,1.0e+40),
   record(b_and_b_sol,0),
   b_and_b(V,L),
   fail.
minimizei(V,L,M):-
   recorded(b_and_b_sol,0),
   !,
   throw(error("Problem has no solution",branch_and_bound)).
minimizei(V,L,M):-
   recorded(b_and_b_min,M).


b_and_b(V,L):-
   condition_d_arret(L),
   !,
   record(b_and_b_sol,1),
   glb(V,Min),
   record(b_and_b_min,Min).
b_and_b(V,L):-
   choix_variable(L,X),
   milieu(X,M),
   decouper(X,M),
   recorded(b_and_b_min,Min),
   lt(V,Min),
   b_and_b(V,L).
```

Les prédicats `condition_d_arret`, `choix_variable`, `milieu`, `decouper` sont ceux décrits au début de ce chapitre (cf. page 69).

Un exemple de requête (on cherche à minimiser la variable X dans le système de contraintes $\{X^2 < 50,\ int(X)\}$.

```
>> int(X),le(square(X),50),minimizei(X,[X],M).

M = -7,
X ~ cc(-7,7).
```

A propos de ces deux méthodes

On pourra noter que la première méthode (basée sur une double énumération) peut connaître des difficultés si la variable `X` qui est imposée comme première variable d'énumération n'a que peu d'influence sur le système de contraintes (cf. 6.4).

A l'opposé, une difficulté qui peut survenir en utilisant la méthode classique du "branch and bound" est la convergence lente (ou en escalier), que l'on peut observer sur le système de contraintes réduit à $\{X \in [-1,1], Y = -X\}$, en cherchant le minimum de Y. En effet, la première solution trouvée ($X \in [-1,-0.9999849]$, $Y \in [0.9999849,1]$) permet de poser sur chaque noeud de l'énumération la contrainte $\{Y < 0.9999849\}$, ce qui nous amène à la solution suivante : $X \in (-0.9999849,-0.9999689]$, $Y \in [0.9999689,0.9999849)$, et ainsi de suite. On ne peut donc atteindre -1 qu'après un très grand nombre de pas.

Recherche d'optimum par dichotomie

Enfin, nous allons décrire une dernière méthode permettant d'approcher le minimum d'une variable dans un système de contraintes. Cette méthode a un gros avantage et un inconvénient : elle se montre en général plus rapide que les autres (voire beaucoup plus rapide dans certains cas), et en tout cas beaucoup plus régulière. Elle doit ces bonnes propriétés à la convergence logarithmique, conséquence de la technique dichotomique utilisée. Par contre, cette technique ne peut être utilisée que sous la condition suivante : la variable que l'on veut minimiser doit avoir un ensemble de solutions convexes, c'est à dire qu'entre son minimum et son maximum, toutes les valeurs doivent être solution.

La technique est alors la suivante (minimisation de X dans un système défini par une liste de variable L) :

Soit (Min, Max), respectivement un minorant du minimum de X (par exemple, la borne inférieure du domaine de X), et une valeur possible de X, donc un majorant du minimum de X. Si $Max - Min$ est jugé suffisamment petit, alors Min sera un minorant du minimum de X satisfaisant. Sinon, soit Mid le milieu de (Min, Max) (On prendra le milieu algébrique pour une variable réelle, et un arrondi entier pour une variable réelle). Alors, si le problème a une solution avec $X = Mid$, on réitère le procédé entre Min et Mid, sinon, on réitère le procédé entre Mid et Max.

On obtient donc le programme Prolog IV suivant :

```
minimizei(X,L,M):-
   glb(X,Min),
   search_one_sol(X,L,Max),
   dichotomy(X,L,Min,Max,M).

search_one_sol(X,L,_):-
   record(one_sol,no),
   once(realsplit(L)),
   glb(X,V),
   record(one_sol,V),
   fail.
search_one_sol(_,_,V):-
   recorded(one_sol,no),
   !,
   throw(error("No solution",dichotomy)).
search_one_sol(_,_,V):-
   recorded(one_sol,V).
```

```
dichotomy(X,L,Min,Max,Min):-
   lt(Max.-.Min,0.0001),!.
dichotomy(X,L,Min,Max,Vmin):-
   middle(Min,Max,Mid),
   verify((X=Mid,realsplit(L))),
   !,
   dichotomy(X,L,Min,Mid,Vmin).
dichotomy(X,L,Min,Max,Vmin):-
   middle(Min,Max,Mid),
   dichotomy(X,L,Mid,Max,Vmin).

verify(P):- \+(\+(P)).

% Integer middle
%middle(Min,Max,Mid):-
%   Mid = floor((Min.+.Max)./.2).

% real middle
middle(Min,Max,Mid):-
   Mid = (Min.+.Max)./.2.
```

Un exemple de requête :

```
>> cc(X,-1,1), Y = .-. X, minimizei(Y,[X,Y],M).

M = -1,
Y ~ cc(-1,1),
X ~ cc(-1,1).
```

Cet exemple passe sans problème alors qu'il posait d'importantes difficultés avec la méthode du branch and bound.

Recherche d'un maximum

Dans cette section, on s'est contenté de décrire les calculs de minima. Les calculs de maxima s'en déduisent naturellement par la formule suivante :

$$max(X, S) = -min(-X, S)$$

X étant une variable, et S un système de contraintes.

En Prolog, on obtiendra donc, si `calcule_min(X,P,Min)` est un prédicat qui calcule une approximation du minimum de X dans P :

```
calcule_max(X,P,Max):-
   calcule_min(.-.X,P,.-.Max).
```

5 Exemples de requêtes et de programmes

Les quelques exemples regroupés dans cette section ont avant tout pour but de mettre en scène les différentes relations disponibles, ainsi que les procédures d'énumération de bases décrites dans les chapitres précédents. Quelle que soit leur longueur ou leur nature, ils ne présentent pas de grandes difficultés, que ce soit au niveau de l'expression en Prolog IV, ou au niveau de la résolution. Des exemples de nature continue un peu plus ardus sont abordés dans le chapitre suivant.

Les exemples ont été regroupés en quatre classes, en fonction de la nature de l'espace sur lequel ils sont construits : les problèmes booléens (2 valeurs), les problèmes de domaines finis (un nombre fini de valeurs), les problèmes entiers (une quantité dénombrable de valeurs), et les problèmes de nature continue (une infinité indénombrable de valeurs).

5.1 Problèmes booléens

5.1.1 Un petit problème booléen

L'objectif de cet exemple est de montrer comment on peut travailler sur des nombres booléens en Prolog IV, en reprenant un exemple booléen traditionnel de Prolog III.

On va introduire d'abord un petit programme Prolog IV ayant pour objet de déduire des connaissances d'un système de contraintes booléennes. Le prédicat `valeur_de(X,L,V)` rend dans V la valeur de la variable X dans le système défini par une liste de variables L. Cette valeur peut être :

- "1" si l'on peut démontrer que X est TOUJOURS vrai.
- "0" si l'on peut démontrer que X est TOUJOURS faux.
- "inconnue" si l'on peut démontrer que X peut être vrai ET peut être faux.
- "impossible" si l'on peut démontrer que le système n'a pas de solution.

Voici ce petit programme en Prolog IV :

```
valeur_de(X,L,1):-
   peut_etre(X,1,L),
   ne_peut_etre(X,0,L),
   !.
valeur_de(X,L,0):-
   peut_etre(X,0,L),
   ne_peut_etre(X,1,L),
   !.
valeur_de(X,L,inconnue):-
   peut_etre(X,0,L),
   peut_etre(X,1,L),
   !.
valeur_de(X,_,impossible).

peut_etre(X,Y,L):-
   verifie((X = Y, boolsplit(L))).

ne_peut_etre(X,Y,L):-
   verifie((Y = bnot(X), boolsplit(L))).

verifie(L):-
   \+(\+(L)).
```

On va maintenant introduire un petit exemple classique (de M. Boole lui-même) dont voici l'énoncé :

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

5. Si ce quelque chose existe par la volonté d'un autre être, alors l'hypothèse que les choses qui existent sont sorties du néant est fausse.

Nommons les valeurs logiques :

- A : quelque chose existe
- B : quelque chose a toujours existé
- C : tout ce qui existe maintenant est sorti du néant
- D : quelque chose existe par la nécessité de sa propre nature
- E : quelque chose existe par la volonté d'un autre être

Si l'on souhaite savoir si quelque chose a toujours existé, on peut alors formuler la requête suivante en Prolog IV ([god]) :

```
>> A ex B ex C ex D ex E ex
   A = 1,
   A impl (B bxor C),
   A impl (D bxor E),
   D impl B,
   E impl (bnot C),
   valeur_de(B,[A,B,C,D,E],Valeur).

Valeur = 1.
```

Quelque chose aurait donc toujours existé.
Faut-il en conclure pour autant que Dieu existe ?

5.1.2 Un problème booléen plus complexe

Considérons le problème booléen suivant : "Soit $B_1, ..., B_n$ n variables booléennes. On souhaite les contraindre de telle manière que p d'entre elles exactement soient vraies".

Le programme suivant implante cette définition récursivement : dans une liste $(B_1, ..., B_n)$, p éléments sont vrais si et seulement si, soit le premier est vrai et $p-1$ sont vrais dans le reste de la liste, soit le premier est faux, et p sont vrais dans le reste de la liste.

On obtient donc le programme suivant ([vrais(n,p)]) :

```
vrais(0,[],1):-!.
vrais(P,[],0).
vrais(0,[B|L],bnot(B) band B1):-
  !,
   vrais(0,L,B1).
vrais(P,[B|L],(B band B2) bor (bnot(B) band B1)):-
   vrais(P.-.1,L,B1),
   vrais(P,L,B2).
```

Si on note $N_R(n,p)$ le nombre de relations booléennes nécessaires pour écrire cette contrainte, on obtient :

$$N_R(n,p) \simeq C_n^p = \frac{n!}{p!(n-p)!}$$

Le nombre de relations croît donc très vite (exponentiellement).

Si l'on considère maintenant le statut des booléens dans Prolog IV, on peut écrire cette contrainte de manière beaucoup plus efficace et astucieuse : il suffit de considérer que p variables valent 1 parmi $(B_1, ..., B_n)$ est équivalent à $B_1 + ... + B_n = p$. Cela nous donne donc le programme suivant ([vraisM(n,p)]) :

```
vrais(P,L,beq(P1,P)):-
   somme(L,P1).

somme([],0).
somme([X|L],X.+.SL):-
   somme(L,SL).
```

Cette deuxième formulation est certes moins naturelle d'un point de vue logique, mais elle est nettement plus facile à mettre en œuvre, et nettement plus efficace. Pour rechercher toutes les solutions, avec $n = 8$, et $p = 4$, elle est déjà plus de 30 fois plus rapide.

Cet exemple montre bien que l'immersion des booléens dans les nombres réels possède de nombreux avantages pour exprimer et résoudre certains problèmes.

5.2 Problèmes de domaines finis

5.2.1 Le puzzle auto-référentiel

Définition et résolution en Prolog IV

Il s'agit d'un problème classique consistant à construire une phrase de la forme :

> Il y a dans cette phrase
> x_1 occurrences de "1"
> x_2 occurrences de "2"
> ...
> x_n occurrences de "n"

Voici un exemple de solution : "Il y a dans cette phrase 2 occurrences de "1", 3 occurrences de "2", 2 occurrences de "3", et 1 occurrence de "4"."

On comprend rapidement la provenance du nom de ce problème lorsqu'on essaie de le formaliser à l'aide de contraintes. En effet, si l'on considère que l'on s'intéresse au vecteur $(x_i)_{i \in 1..n}$, on s'aperçoit très vite qu'il se définit à partir de lui-même de la manière suivante :

$$\forall i \in 1..n \; : \; x_i = 1 + \sum_{j \in 1..n} \begin{cases} 1 \; si \; x_j = i \\ 0 \; sinon \end{cases}$$

Or justement, il existe en Prolog IV une forme de contrainte particulièrement adaptée à l'expression de ce type d'expressions (cf. section 2.7.4). Pour être plus précis, on va utiliser ici la relation "beq" dont la définition est :

$$beq/3 = \{(b_1, a_2, a_3) \mid \; si \; (a_2 = a_3) \; alors \; b_1 = 1 \; sinon \; b_1 = 0\}$$

Ce qui nous donne comme expression du problème :

$$\forall i \in 1..n \ : \ x_i = 1 + \sum_{j \in 1..n} beq(x_j, i)$$

On déduit alors immédiatement le programme Prolog IV suivant ([magic(N)]) :

```
magicsequence(L):-
   nfirstconstraints(size(L),L),
   intsplit(L).

nfirstconstraints(0,L).
nfirstconstraints(I,L):-
   gt(I,0),
   nboccurrences(I,L,L:I),
   nfirstconstraints(I.-.1,L).

nboccurrences(_,[],1).
nboccurrences(I,[X|L],N .+. beq(I,X)):-
   nboccurrences(I,L,N).
```

... et voici deux exemples de requête :

```
>> size(L)=4,magicsequence(L).

L = [2,3,2,1];

L = [3,1,3,1].

>> size(L)=10,magicsequence(L).

L = [7,3,2,1,1,1,2,1,1,1].
```

Ajout de contraintes redondantes

La somme des x_i désigne clairement la quantité de nombres présents dans la phrase. Or ce nombre est connu : c'est $2n$.

De même, la somme des ix_i représente la somme de tous les nombres de la phrase. Cette somme est elle-même composée de la somme des x_i, qui vaut $2n$, et de la somme des nombres situés à droite du mot "occurrence" dans la phrase, soit la somme des nombres de 1 à n, donc $n(n+1)/2$.

On obtient donc les deux formules suivantes :

$$\sum_{i \in 1..n} x_i = 2n$$

et

$$\sum_{i \in 1..n} ix_i = n(n+1)/2 + 2n = n(n+5)/2$$

On déduit alors le programme Prolog IV suivant ([magicR(N)]) :

```
magicsequence(L):-
   N = size(L),
   nfirstconstraints(N,L),
   sum_x(L,2.*.N),
   sum_ix(1,L,N.*.(N.+.5)./.2),
   intsplit(L).

nfirstconstraints(0,L).
nfirstconstraints(I,L):-
   gt(I,0),
   nboccurrences(I,L,L:I),
   nfirstconstraints(I.-.1,L).

nboccurrences(_,[],1).
nboccurrences(I,[X|L],N .+. beq(I,X)):-
   nboccurrences(I,L,N).

sum_x([],0).
sum_x([X|L],X.+.S):-
   sum_x(L,S).
```

```
sum_ix(_,[],0).
sum_ix(I,[X|L],I.*.X.+.S):-
    sum_ix(I.+.1,L,S).
```

Comparons maintenant les temps d'exécution selon que l'on utilise ou pas les contraintes redondantes :

achille noopt

n	Sans redondance	Avec redondances	ratio
10	2.8	1.2	2.3
20	32.2	5.1	6.3
30	144.2	12.3	11.7
40	426.7	22.8	18.7
50	1 015.6	40.5	25.1
60	2 026.8	61.8	32.8
70	3 753.9	89.8	41.8
80	6 026.3	100.7	59.8
90	9 611.3	133.4	72.0
100	14 985.6	171.8	87.2

Comme on peut le voir sur ce tableau comparatif, les contraintes redondantes modifient profondément le comportement du programme. Il est cependant important de rappeler (cf. 2.3) qu'elles ne sont redondantes que du point de vue de la résolution du problème initial ; pour le système de résolution par approximation, il n'y a pas redondance. C'est pourquoi le comportement du point fixe est différent.

5.2.2 Un puzzle difficile

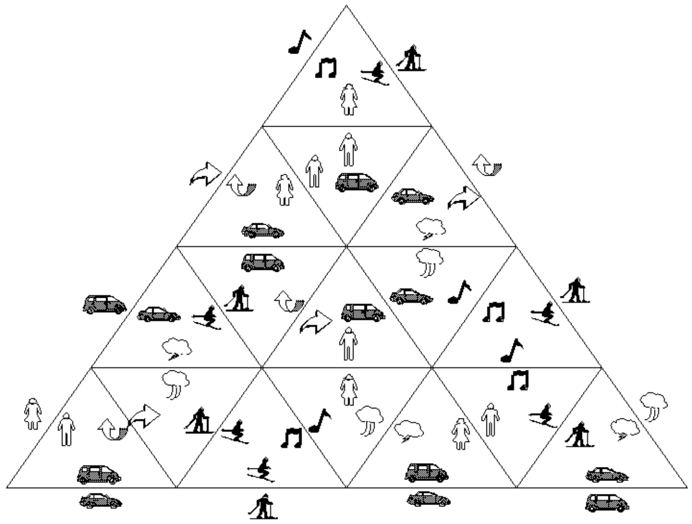
Le problème que l'on va traiter maintenant demande quelque habitude pour être traduit en Prolog IV facilement.

Enoncé

Il s'agit d'un puzzle composé de 16 pièces. Chacune de ces pièces triangulaires porte un symbole sur chaque côté. Ces symboles ne permettent aux pièces de s'apparier que s'ils sont de même nature, mais différents. Il y a 6 paires de symboles. L'objet du puzzle est de reconstituer un grand triangle composé des 16 petits triangles, en respectant la contrainte d'appariement entre les petits triangles, ainsi que par rapport à des symboles situés dans l'encadrement du puzzle.

Si vous avez quelque temps devant vous, construisez donc ce puzzle, vous vous rendrez vite compte qu'il n'est pas aisé à résoudre à la main.

Voici son unique solution :



Toute la difficulté de ce type de problème repose dans le fait que les contraintes que l'on veut exprimer portent sur des éléments inconnus. Les contraintes sont par exemple de la forme :

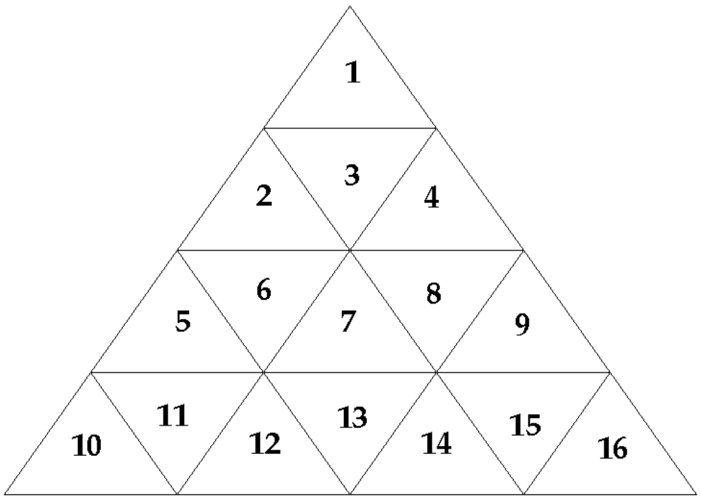
"Le bord gauche de la pièce située à telle place est compatible avec le bord gauche de la pièce située à telle autre place"

On va donc être amené à parler simultanément des 16 pièces du puzzle, que l'on numérotera de 1 à 16, et des 16 positions possibles, que l'on numérotera également de 1 à 16. Le problème se complique encore un peu du fait que les pièces ont la faculté de tourner.

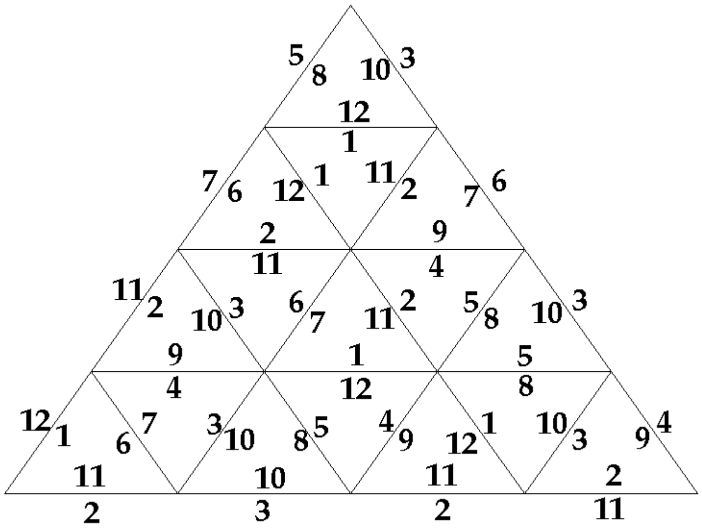
Formalisation du problème

- Tout d'abord, on considère que l'on dispose de 48 pièces, et non plus de 16. Ces 48 pièces sont, respectivement, les 16 pièces initiales, ces mêmes 16 pièces après une rotation, et toujours ces mêmes 16 pièces, après deux rotations.
- On considère alors que choisir une pièce revient à choisir un numéro de pièce P entre 1 et 16, et un numéro de sens S, entre 0 et 2. La pièce correspondante est alors la pièce $16S + P$, et l'on peut imposer que deux pièces (P_1, S_1) et (P_2, S_2) soient différentes tout simplement avec la contrainte : $P_1 \neq P_2$.
- On associe à chacune des 16 positions i du puzzle (cf. schéma ci-après) un doublet (P_i, S_i) désignant la pièce placée à cet endroit. Il ne reste plus qu'à contraindre cette pièce.
- Enfin, on associe à chaque symbole un numéro de 1 à 12 de telle manière que deux côtes de pièces soient compatibles si leur somme vaut 13 (cf. schéma suivant).

Voici les numéros des positions du puzzle :



Voici le puzzle, symboles remplacés par des numéros :



Programmation en Prolog IV

Les données de base dont on dispose sont les 48 pièces, telles que définies plus haut, sous la forme de trois listes : les numéros des symboles en bas, les numéros des symboles à gauche, et les numéros des symboles à droite.

```
gauche([5,12,9,11,7,9,11,11,5,1,2,12,5,6,1,10,
        10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,
        8,8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10]).
droite([8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10,5,
        12,9,11,7,9,11,11,5,1,2,12,5,6,1,10,
        10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,8]).
bas([10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,
     8,8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10,5,
     12,9,11,7,9,11,11,5,1,2,12,5,6,1,10]).
```

On va contraindre les numéros et sens de pièces associés à chaque position à être des entiers respectivement compris entre 1 et 16, et entre 0 et 2. Puis on va calculer les numéros des symboles des côtés gauches, droits et bas des pièces placées à chaque position, construisant ainsi trois listes : celle des gauches des pièces, celle des droites, et celle des bas (LB, LG, et LD sont les listes de numéros de symboles précédentes ; LGa, LDr, et LBa les listes que l'on calcule) :

```
calcul_gdb([[],[]],_,[[],[],[]]).
calcul_gdb([[C|Choix],[S|Sens]],[LB,LG,LD],
                                [[G|LGa],[D|LDr],[B|LBa]]):-
   Pos = C.+.S.*.16,
   B = LB:Pos,
   D = LD:Pos,
   G = LG:Pos,
   calcul_gdb([Choix,Sens],[LB,LG,LD],[LGa,LDr,LBa]).
```

Enfin, voici par exemple, le prédicat permettant de contraindre deux pièces dont les bas se touchent, et celui permettant de préciser quel est le bas d'une pièce située en bas du puzzle (symboles présents sur les bords) :

```
bb(P1,P2,[_,_,LB]):-
   LB:P1 .+. LB:P2 = 13.

b(P1,V,[_,_,LB]):-
   LB:P1 = V.
```

L'essentiel est là. Il ne reste plus qu'à donner l'ensemble du programme ([puzzle]) :

```
contraintes(L):-
   bas(LB), gauche(LG), droite(LD),
   eq(Pieces,[LB,LG,LD]),
   size(Choix) = 16, size(Sens) = 16,
   L = Choix o Sens,
   Lcs = [Choix,Sens],
   val_poss(Choix,Sens),
   tous_dif(Choix),
   calcul_gdb(Lcs,Pieces,Lgdb),
```

```
   % Contraintes entre pièces
   bb(1,3,Lgdb),      dd(2,3,Lgdb),      gg(3,4,Lgdb),
   bb(6,2,Lgdb),      bb(4,8,Lgdb),      dd(5,6,Lgdb),
   gg(6,7,Lgdb),      dd(7,8,Lgdb),      gg(8,9,Lgdb),
   bb(5,11,Lgdb),     bb(7,13,Lgdb),     bb(9,15,Lgdb),
   dd(10,11,Lgdb),    gg(11,12,Lgdb),    dd(12,13,Lgdb),
   gg(13,14,Lgdb),    dd(14,15,Lgdb),    gg(15,16,Lgdb),
   % Contraintes au bord
   g(1,8,Lgdb),       g(2,6,Lgdb),       g(5,2,Lgdb),
   g(10,1,Lgdb),      d(1,10,Lgdb),      d(4,7,Lgdb),
   d(9,10,Lgdb),      d(16,9,Lgdb),     b(10,11,Lgdb),
   b(12,10,Lgdb),     b(14,11,Lgdb),     b(16,2,Lgdb).


val_poss([],[]).
val_poss([C|Choix],[S|Sens]):-
   val_poss(Choix,Sens),
   C ~ cc(1,16) n int,
   S ~ cc(0,2) n int.

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

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

calcul_gdb([[],[]],_,[[],[],[]]).
calcul_gdb([[C|Choix],[S|Sens]],[LB,LG,LD],
                                [[G|LGa],[D|LDr],[B|LBa]]):-
   Pos = C.+.S.*.16,
```

```
   B = LB:Pos,
   D = LD:Pos,
   G = LG:Pos,
   calcul_gdb([Choix,Sens],[LB,LG,LD],[LGa,LDr,LBa]).

bb(P1,P2,[_,_,LB]):-
   LB:P1 .+. LB:P2 = 13.

b(P1,V,[_,_,LB]):-
   LB:P1 = V.

gg(P1,P2,[LG,_,_]):-
   LG:P1 .+. LG:P2 = 13.

g(P1,V,[LG,_,_]):-
   V = LG:P1.

dd(P1,P2,[_,LD,_]):-
   LD:P1 .+. LD:P2 = 13.

d(P1,V,[_,LD,_]):-
   V = LD:P1.

/* Données : */

gauche([5,12,9,11,7,9,11,11,5,1,2,12,5,6,1,10,
        10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,
        8,8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10]).
droite([8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10,5,
        12,9,11,7,9,11,11,5,1,2,12,5,6,1,10,
        10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,8]).
bas([10,6,3,1,3,10,12,7,4,11,9,10,4,11,10,
     8,8,2,2,1,4,2,9,1,2,6,7,8,12,3,8,10,5,
     12,9,11,7,9,11,11,5,1,2,12,5,6,1,10]).
```

Exécution de ce programme :

```
>> contraintes(L),intsplit(L).
```

```
[12,2,4,11,6,14,8,9,1,10,5,16,13,7,15,3,
   2,1,0,0,2,0,1,0,2,0,1,2,1,2,1,1].
```

Unions d'intervalles et intervalles simples

achille noopt

Il est assez important de noter que le comportement du programme est très différent suivant que l'on utilise des unions d'intervalles ou des intervalles simples. Dans le premier cas, en effet, l'énumération n'est pas nécessaire, alors qu'avec les intervalles simples, elle l'est. L'aspect le plus gênant est la très grande dépendance des performances en fonction de la stratégie d'énumération, et ce, de manière quasiment imprévisible. Résultat : le problème se résout en 4.1 secondes avec les unions d'intervalles, quelle que soit la méthode d'énumération employée (puisqu'elle ne sert à rien), au lieu de 48.6 secondes à 53 heures en utilisant les intervalles simples.

Une méthode générale

Contrairement à ce que pourrait laisser croire cet exemple au premier abord, les techniques de programmation utilisées sont relativement générales : elle peuvent servir dans tous les problèmes consistant à choisir des "ingrédients" (ici, les pièces du puzzle) dans une liste, pour ensuite appliquer des contraintes aux choix qui ont été faits.

5.3 Problèmes en nombres entiers

5.3.1 Calcul des nombres premiers

On va montrer ici une requête permettant de calculer l'ensemble des nombres premiers entre 1 et 100 très simplement. On va pour cela se ramener à la définition basique des nombres premiers, adaptée au cas particulier qui nous intéresse :

X est un nombre premier entre 1 et 100 si X est 1,2,3,5, ou 7, ou si X est un entier supérieur à 1 qui n'est multiple ni de 2, ni de 3, ni de 5, ni de

7.

On obtient alors la requête suivante (Prolog IV étant configuré en unions d'intervalles) ([premiers]) :

```
>> X ~ 1 u 2 u 3 u 5 u 7 u
      cc(2,100) n int n 2.*.nint n 3.*.nint n
                          5.*.nint n 7.*.nint.

X ~ 1 u 2 u 3 u 5 u 7 u 11 u 13 u 17 u 19 u
    23 u 29 u 31 u 37 u 41 u 43 u 47 u 53 u
    59 u 61 u 67 u 71 u 73 u 79 u 83 u 89 u 97.
```

Quelques remarques et rappels :

- `n` (qui signifie intersection) réalise l'intersection des pseudo-termes du point de vue ensembliste. D'un point de vue plus "contraintes", cette relation doit être vue comme une conjonction, c'est à dire que la deuxième partie de cette requête peut se lire : "X est compris entre 2 et 100 ET X est entier ET ...".
- De même `u` (qui signifie union) réalise l'union des pseudo-termes du point de vue ensembliste. D'un point de vue plus "contraintes", cette relation doit être vue comme une disjonction, c'est à dire que cette requête peut se lire : "X vaut 1 OU X vaut 2 OU ...".
- `nint` est la relation unaire qui signifie "Etre un nombre réel qui n'est pas un entier", en d'autres termes "1.*.nint" représente l'ensemble des nombres qui ne sont pas multiples de 1. De même "2.*.nint" représente l'ensemble des nombres qui ne sont pas multiples de 2, etc ...

5.3.2 DONALD + GERALD = ROBERT

Là encore, il s'agit d'un exemple très classique dont l'énoncé est le suivant :

"DONALGERBT sont les dix chiffres (ils sont donc tous différents). On cherche alors une solution vérifiant l'équation $DONALD + GERALD = ROBERT$.

Si on note δ le vecteur $[..., 10^n, 10^n-1, ...10^2, 10, 1]$, et $*$ le produit scalaire, on obtient l'équation suivante :

$$[D, O, N, A, L, D] * \delta + [G, E, R, A, L, D] * \delta = [R, O, B, E, R, T] * \delta$$

On en déduit alors immédiatement le programme Prolog IV suivant ([donald]) :

```
poser_contraintes([D,O,N,A,L,G,E,R,B,T]):-
   chiffres_differents([D,O,N,A,L,G,E,R,B,T]),
   produit_scalaire([D,O,N,A,L,D],
                    [100000,10000,1000,100,10,1],S1),
   produit_scalaire([G,E,R,A,L,D],
                    [100000,10000,1000,100,10,1],S2),
   produit_scalaire([R,O,B,E,R,T],
                    [100000,10000,1000,100,10,1],S3),
   S3 = S1.+.S2.

chiffres_differents(L):-
   chiffres(L),
   tous_dif(L).

chiffres([]).
chiffres([X|L]):-
   int(X),
   cc(X,0,9),
   chiffres(L).

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

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

```
produit_scalaire([],[],0).
produit_scalaire([X1|L1],[X2|L2],S .+. X1.*.X2):-
   produit_scalaire(L1,L2,S).
```

On peut alors poser la question suivante :

```
>> poser_contraintes(L),intsplit(L).

L = [5,2,6,4,8,1,9,7,3,0].
```

La solution de ce problème est donc :

$$\begin{array}{rr} & 526485 \\ + & 197485 \\ \hline = & 723970 \end{array}$$

On peut cependant être tenté de modifier légèrement ce programme en effectuant certaines manipulations formelles (des additions et des soustractions terme à terme). On obtient alors le programme suivant ([donaldM]) :

```
poser_contraintes([D,O,N,A,L,G,E,R,B,T]):-
   chiffres_differents([D,O,N,A,L,G,E,R,B,T]),
   produit_scalaire([D,O,N,A,L,G,E,R,B,T],
       [100002,0,1000,200,20,100000,9900,-99010,-1000,-1],0).
```

On peut alors poser la question suivante :

```
>> poser_contraintes(L),intsplit(L).

L = [5,2,6,4,8,1,9,7,3,0].
```

Là encore, une modification formelle mineure entraîne d'importants changements dans le comportement du programme, puisque cette deuxième version est 7 fois plus rapide.

5.3.3 Les reines

Nous allons résoudre ici le problème classique des reines qui se pose de la manière suivante : on cherche à placer n reines sur un échiquier de dimension $n*n$ de sorte qu'elles ne soient pas "en prise". On dit que deux reines sont "en prise" si elles sont sur la même ligne, la même colonne, ou la même diagonale (dans les deux sens).

Plutôt que de considérer l'abscisse et l'ordonnée de chaque reine, on va manipuler le vecteur constitué par les ordonnées des reines placées respectivement sur chacune des lignes. On calcule ensuite les numéros de diagonales, et on contraint les numéros de colonnes et de diagonales à être différents.

Voici le programme que l'on obtient en Prolog IV, suivi de deux requêtes, la première calculant toutes les solutions pour $n = 4$ ([queens(4)]), la deuxième une seule solution pour $n = 10$ ([queensOne(10)]).

```
constraints(N,L):-
   size(L) = N,
   build_diags(1,L,Ld1,Ld2),
   in_1n(N,L),
   all_dif(L),
   all_dif(Ld1),
   all_dif(Ld2).

build_diags(_,[],[],[]).
build_diags(I,[X|L],[X.+.I|Ld1],[X.-.I|Ld2]):-
   build_diags(I.+.1,L,Ld1,Ld2).

all_dif([]).
all_dif([X|L]):-
   out_of_list(X,L),
   all_dif(L).

out_of_list(X,[]).
out_of_list(X,[Y|L]):-
   dif(X,Y),
   out_of_list(X,L).
```

```
in_1n(N,[]).
in_1n(N,[X|L]):-
   cc(X,1,N),int(X),
   in_1n(N,L).


>> constraints(4,L),intsplit(L).

L = [2,4,1,3];

L = [3,1,4,2].

>> constraints(10,L),intsplit(L),!.

L = [1,3,6,9,7,10,4,2,5,8].
```

Intervalles simples et unions d'intervalles

Si l'on compare les intervalles simples et les unions d'intervalles sur ce type de problème, on constate qu'à l'instar du puzzle (cf. 5.2.2), les unions d'intervalles apportent une plus grande régularité dans l'exécution et une plus grande indépendance vis à vis de l'énumération.

Ainsi, si l'on souhaite trouver une solution ([queensOne(n)]) du problème des reines, pour des n assez grands, et en utilisant l'énumération de base des nombres entiers, on obtient les résultats suivants (ce tableau compare les résultats obtenus en fonction du mode d'approximation choisi) :

n	Ratio unions/simples
5	1.0
10	1.2
15	7.0
20	19.7
25	68.3

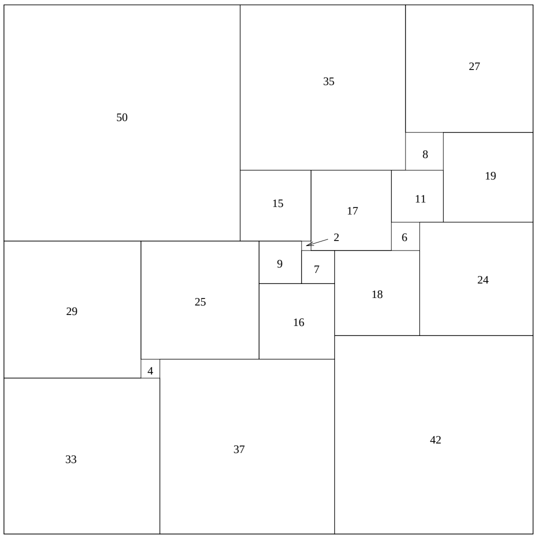
5.3.4 Le problème des carrés

Ici encore, nous allons traiter un problème classique aux allures d'exercice d'école, mais dont les mécanismes fondamentaux sont des techniques classiques, généralement utilisées pour résoudre des problèmes d'ordonnancement.

Enoncé du problème

On dispose de 21 carrés de tailles connues :
50, 42, 37, 35, 33, 29, 27, 25, 24, 19, 18, 17, 16, 15, 11, 9, 8, 7, 6, 4, 2. Le problème consiste à former un grand carré de côté 112 à l'aide de ces 21 carrés. L'énoncé initial de ce problème consistait à trouver les tailles de 21 carrés pouvant former un grand carré ([7]), mais la problématique, et les techniques sont complètement différentes. Le problème tel que nous le présentons ici à été résolu pour la première fois en PLC par N. Beldiceanu ([1]).

Ce problème n'a qu'une seule solution, si l'on prend en considération les 3 axes de symétrie possibles. Là voici :



Formalisation du problème

On va représenter ce problème par un vecteur V de quintuplets de la forme (X_d, Y_d, X_f, Y_f, S), chaque quintuplet représentant les abscisses et ordonnées des points supérieur gauche et inférieur droit d'un des 21 carrés, ainsi que sa taille.

Voici les contraintes :

- Contraintes basiques sur V :
  - Les carrés ... sont des carrés :

    $$\forall (X_d, Y_d, X_f, Y_f, S) \in V, X_f = X_d + S\ ,\ Y_f = Y_d + S$$

  - Les carrés sont dans le grand carré :

    $$\forall (X_d, Y_d, X_f, Y_f, S) \in V, X_d \in [0, 112 - S]\ ,\ Y_d \in [0, 112 - S]$$

  - Les positions des carrés sont nécessairement entières :

    $$\forall (X_d, Y_d, X_f, Y_f, S) \in V, X_d \in I\!N,\ Y_d \in I\!N$$

- Contraintes de non-chevauchement :
  Soit (X_d, Y_d, X_f, Y_f, S) et $(X'_d, Y'_d, X'_f, Y'_f, S')$ deux éléments distincts de V. On a :

  $$(X_d \geq X'_f)\ ou\ (X'_d \geq X_f)\ ou\ (y_d \geq Y'_f)\ ou\ (Y'_d \geq Y_f)$$

  Cette contrainte traduit le fait qu'un carré doit être à gauche, à droite, au dessus, ou au dessous d'un autre carré.

- Contraintes d'ordonnancement :
  On va ici exprimer le fait que pour tout $i \in [0, 111]$, la somme des tailles des carrés qui coupent la droite d'équation $X = i$ vaut 112. Il en va de même de la somme des tailles des carrés coupant la droite d'équation $Y = i$.
  On obtient donc les formules suivantes :

  $$\forall i \in [0, 111]\ , \sum_{(X_d, Y_d, X_f, Y_f, S) \in V | i \in [X_d, X_f)} S = 112$$

$$\forall i \in [0,111]\,, \quad \sum_{(X_d,Y_d,X_f,Y_f,S)\in V | i\in[Y_d,Y_f)} S = 112$$

En renversant le sens des inégalités, on trouve les formules équivalentes suivantes :

$$\forall i \in [0,111]\,, \quad \sum_{(X_d,Y_d,X_f,Y_f,S)\in V | X_d\in(i-S,i]} S = 112$$

$$\forall i \in [0,111]\,, \quad \sum_{(X_d,Y_d,X_f,Y_f,S)\in V | Y_d\in(i-S,i]} S = 112$$

En se ramenant à un formalisme plus "Prolog IV", on obtient :

$$\forall i \in [0,111]\,, \quad \sum_{(X_d,Y_d,X_f,Y_f,S)\in V} boc(X_d, i-S, i) * S = 112$$

$$\forall i \in [0,111]\,, \quad \sum_{(X_d,Y_d,X_f,Y_f,S)\in V} boc(Y_d, i-S, i) * S = 112$$

Il est clair que ces contraintes ne sont pas nécessaires à l'expression mathématique du problème, et sont donc redondantes. Elles apportent cependant des informations supplémentaires au système de résolution.

Ecriture des contraintes en Prolog IV

Voici le prédicat Prolog IV qui pose la contrainte de non-chevauchement entre deux carrés :

```
no_overlape_constraints3([X1d,Y1d,X1f,Y1f,_],
                         [X2d,Y2d,X2f,Y2f,_]):-
   ble(X1f,X2d) or bge(X1d,X2f) or
   ble(Y1f,Y2d) or bge(Y1d,Y2f).
```

Et voici le prédicat Prolog IV qui pose les contraintes d'ordonnancement pour un i donné :

```
schedule([],_,0,0).
schedule([[Xd,Yd,_,_,S]|V],I,SX.+.S1,SY.+.S2):-
   Imin = I .-. S .+. 1,
   SX = S .*. bcc(Xd,Imin,I),
   SY = S .*. bcc(Yd,Imin,I),
   schedule(V,I,S1,S2).
```

Enumération du problème

On va utiliser une énumération particulière pour résoudre ce problème, définie comme suit. On va énumérer successivement les abscisses et les ordonnées des carrés de la manière suivante :

- Choix de la variable : l'abscisse (ou l'ordonnée) du plus gros carré non-encore figé tel que le domaine de son abscisse (ou de son ordonnée) ait pour borne inférieure le minimum des bornes inférieures des abscisses (ou des ordonnées) non-encore figées.

  Plus simplement, on s'intéresse aux carrés non-encore placés pouvant se placer le plus bas (ou le plus à gauche) possible. Parmi ceux-ci, on va choisir le plus gros.

- Découpage du domaine de la variable : le domaine de la variable sélectionnée va être coupé en deux : son premier élément, et le reste de son domaine.

Voici le programme Prolog IV qui effectue cette énumération :

```
enumerate(Squares):-
   split(Squares,Abs,Ord),
   enumerate2(Abs),
   enumerate2(Ord).

split([],[],[]).
split([[Xd,Yd,_,_,_]|L],[Xd|L1],[Yd|L2]):-
   split(L,L1,L2).
```

```
enumerate2(L):-
   all_bounded(L),
   !.
enumerate2(L):-
   min_list_unbound(L,Min),
   bigest_square(L,Min,X),
   choose(X,Min),
   enumerate2(L).

bigest_square([X|L],Min,Y):-
   number(X),
   !,
   bigest_square(L,Min,Y).
bigest_square([X|_],Min,X):-
   glb(X,Min),
   !.
bigest_square([_|L],Min,X):-
   bigest_square(L,Min,X).

choose(Min,Min).
choose(X,Min):-
   gt(X,Min).


all_bounded([]).
all_bounded([X|L]):-
   number(X),
   all_bounded(L).


min_list_unbound([],200).
min_list_unbound([X|L],M):-
   number(X),!,
   min_list_unbound(L,M).
min_list_unbound([X|L],M):-
   min_list_unbound(L,M1),
```

```
glb(X,M2),
M = min(M1,M2).
```

Le programme complet ([squares]) se trouve en annexe (cf. annexe 9.4.1).

Les contraintes d'ordonnancement

Ce problème des carrés peut être vu comme un programme d'ordonnancement : les abscisses correspondant au temps, et les ordonnées à des capacités. Les contraintes redondantes que l'on a nommé "contraintes d'ordonnancement" se lisent alors : "la somme des capacités nécessaires aux tâches s'exécutant à un instant t est égale à la capacité totale (112)".

Ces contraintes ne suffisent cependant pas à résoudre le problème des carrés car il n'y a là aucune notion de consistance géométrique des carrés. En d'autres termes, rien ne dit qu'un carré ne peut pas être découpé pour satisfaire les contraintes.

Si on n'utilise que ces contraintes, on trouve alors des solutions faisant apparaître des carrés qui se trouvent pour partie en bas du grand carré, et pour partie en haut, ce qui ne correspond pas aux solutions que l'on cherche.

Il n'empêche que le plus gros du travail est fait par ces contraintes, et que l'on peut donc considérer que le problème des carrés est un problème d'ordonnancement complété par quelques contraintes particulières.

5.4 Des problèmes continus

5.4.1 De petits systèmes algébriques

Racines d'un polynôme

Considérons le polynôme suivant :

$$y = 35x^{256} - 14x^{17} + x$$

On va chercher ses racines dans l'intervalle $[-100, 100]$. On va pour cela formuler le problème de trois manières différentes :

1. En utilisant uniquement la multiplication pour évaluer les puissances.
2. En utilisant l'exponentielle et le logarithme pour évaluer les puissances, utilisant la formule $x^n = e^{n.ln(X)}$. Les deux règles (voir ci-après) permettent de distinguer le cas "puissance paire" du cas "puissance impaire".
3. En utilisant la contrainte "power" pour évaluer les puissances.

Cette équation possède quatre racines :

$$\begin{cases} X \simeq 0.8479437 \\ X = 0 \\ X \simeq 0.8479437 \\ X \simeq 0.9958424 \end{cases}$$

.

Voici le programme Prolog IV, et les trois requêtes correspondantes :

```
puissance(X,0,1):-! .
puissance(X,N,X .*. Y):- puissance(X,N .-. 1,Y).

my_power(X,N,Y):-
   0 is '^mod'(N,2),
   Y = exp(N .*. ln(abs(X))).
my_power(X,N,Y):-
   1 is '^mod'(N,2),
   if(Y,bge(X,0),exp(N .*. ln(abs(X))),
                 .-.exp(N .*. ln(abs(X)))).


>> puissance(X,256,X256), puissance(X,17,X17),
   35 .*. X256 .-. 14 .*. X17 .+. X = 0,
   X ~ cc(-100,100),
   realsplit([X]).

X ~ cc(-‘>0.8480072‘,-‘>0.8479118‘);

X ~ oc(-‘<7e-44‘,0);
```

```
X ~ oo(0,`<2e-44`);

X ~ oo(`>0.8479118`,`>0.8480072`);

X ~ oo(`>0.9958267`,`<0.9959221`).

>> my_power(X,256,X256), my_power(X,17,X17),
   35 .*. X256 .-. 14 .*. X17 .+. X = 0,
   X ~ cc(-100,100),
   realsplit([X]).

X ~ oo(-`>0.8479437`,-`>0.8479436`);

X ~ oo(-`<7e-44`,0);

X ~ oo(0,`<2e-44`);

X ~ oo(`>0.8479436`,`>0.8479437`);

X ~ oo(`>0.9958424`,`>0.9958425`).

>> 35 .*. power(X,256) .-. 14 .*. power(X,17) .+. X = 0,
   X ~ cc(-100,100),
   realsplit([X]).

X ~ cc(-`>0.8479437`,-`>0.8479436`);

X ~ oc(-`<7e-44`,`<2e-44`);

X ~ oo(`>0.8479436`,`>0.8479437`);

X ~ oo(`>0.9958424`,`>0.9958425`).
```

Tableau comparatif des trois requêtes (Temps, nombre de pseudo-solutions trouvées, et précision de ces pseudo-solutions) :

achille noopt

	Temps	Nb sols.	Précision
Requête I [polyTimes]	27.5 s	5	$1.0e^{-}4$
Requête II [polyExp]	1.6 s	5	$1.0e^{-}7$
Requête III [polyPower]	0.8 s	4	$1.0e^{-}7$

Les trois traitements que l'on a proposés pour ce même système de contraintes montrent bien que, si le traitement d'une relation est effectué de manière parfaite lorsque celle-ci est une des relations de base, le comportement du système de résolution se dégrade lorsqu'elle est décomposée en relations de base.

Un problème de chimie

Considérons le système d'équations suivant qui décrit la combustion du propane dans l'air (ce problème a été introduit dans [14] comme un système d'équations difficile à résoudre). Les variables X_1 à X_n représentent les volumes des gaz en présence. R et P sont des paramètres de combustion.

$$X_1 + X_4 - 3 = 0$$
$$2X_1 + X_2 + X_4 + X_7 + X_8 + X_9 + 2 * X_10 - R = 0$$
$$X_2 + 2 * X_5 + X_6 + X_7 - 8 = 0$$
$$X_3 + X_9 - 4 * R = 0$$
$$X_T = X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + X_8 + X_9 + X_10$$
$$K_5 * X_2 * X_4 - X_1 * X_5 = 0$$
$$K_6 * X_2^{1/2} * X_4^{1/2} - X_1^{1/2} * X_6 * (P/X_T)^{1/2} = 0$$
$$K_7 * X_1^{1/2} * X_2^{1/2} - X_4^{1/2} * X_7 * (P/X_T)^{1/2} = 0$$
$$K_8 * X_1 - X_4 * X_8 * PX_T = 0$$
$$K_9 * X_1 * X_3^{1/2} - X_4 * X_9 * (P/X_T)^{1/2} = 0$$
$$K_10 * X_1^2 - X_4^2 * X_10 * (P/X_T) = 0$$
$$K_5 = 1.93E - 1$$
$$K_6 = 2.597E - 3$$
$$K_7 = 3.448E - 3$$
$$K_8 = 1.799E - 5$$
$$K_9 = 2.155E - 4$$
$$K_10 = 3.846E - 5$$
$$P = 40, R = 10$$
$$\forall i \in 1..10,\ X_i > 0$$

Voici la traduction en Prolog IV ([chemical]) :

```
constraints([X1,X2,X3,X4,X5,X6,X7,X8,X9,X10]):-
   X1 .+. X4 .-. 3 = 0,
   2 .*. X1 .+. X2 .+. X4 .+. X7 .+. X8 .+. X9 .+.
               2 .*. X10 .-. R = 0,
   2 .*. X2 .+. 2 .*. X5 .+. X6 .+. X7 .-. 8 = 0,
   2 .*. X3 .+. X9 .-. 4 .*. R = 0,
   XT = X1 .+. X2 .+. X3 .+. X4 .+. X5 .+. X6 .+.
        X7 .+. X8 .+. X9 .+. X10,
   PXT = P ./. XT,
   SQRTPXT = sqrt(PXT),
   K5 .*. X2 .*. X4 .-. X1 .*. X5 = 0,
   K6 .*. sqrt(X2 .*. X4) .-. sqrt(X1) .*. X6 .*. SQRTPXT = 0,
   K7 .*. sqrt(X1 .*. X2) .-. sqrt(X4) .*. X7 .*. SQRTPXT = 0,
   K8 .*. X1 .-. X4 .*. X8 .*. PXT = 0,
```

```
   K9 .*. X1 .*. sqrt(X3) .-. X4 .*. X9 .*. SQRTPXT = 0,
   K10 .*. square(X1) .-. square(X4) .*. X10 .*. PXT = 0,
   K5 = 1.93E-1,
   K6 = 2.597E-3,
   K7 = 3.448E-3,
   K8 = 1.799E-5,
   K9 = 2.155E-4,
   K10 = 3.846E-5,
   P = 40,
   R = 10,
   gt(X1 , 0),
   gt(X2 , 0),
   gt(X3 , 0),
   gt(X4 , 0),
   gt(X5 , 0),
   gt(X6 , 0),
   gt(X7 , 0),
   gt(X8 , 0),
   gt(X9 , 0),
   gt(X10 , 0).


>> constraints(L),realsplit(L).

L ~ [oo('>2.9157235','<2.915733'),
     oo('>3.9609413','>3.9609453'),
     oc('<19.98629','>19.98629'),
     oo('<0.08426714','>0.08427633'),
     oo('>0.022093582','>0.022096091'),
     oo('>7.227331e-4','<7.227752e-4'),
     oo('<0.03320002','<0.03320196'),
     oo('>4.210902e-4','<4.211385e-4'),
     co('<0.02741623','<0.02741922'),
     oo('>0.0311454','>0.031152473')].
```

5.4.2 Un peu de trigonométrie

Prolog IV dispose d'une large palette de relations de bases. Parmi celles-ci, on trouve les relations trigonométriques. Voici quelques exemples d'utilisation de ces relations.

Quelques requêtes simples

Une équation simple :

```
>> cos(X)=X.

X ~ oo('>0.739085','>0.7390852').
```

Une équation un peu plus complexe :

```
>> X ~ cc(0,pi), cos(X)=sin(X).

X ~ cc(0,'<1.570797').

>> X ~ cc(0,pi), cos(X)=sin(X), realsplit([X]).

X ~ oc('>0.7853977','<0.7853985').
```

Comme on peut le constater, le recours à une énumération est nécessaire pour obtenir une précision satisfaisante.

Résolvons maintenant le système de contraintes $\{X \in [0,10],\ sin(X) = 0\}$ successivement avec les intervalles simples et avec les unions d'intervalles :

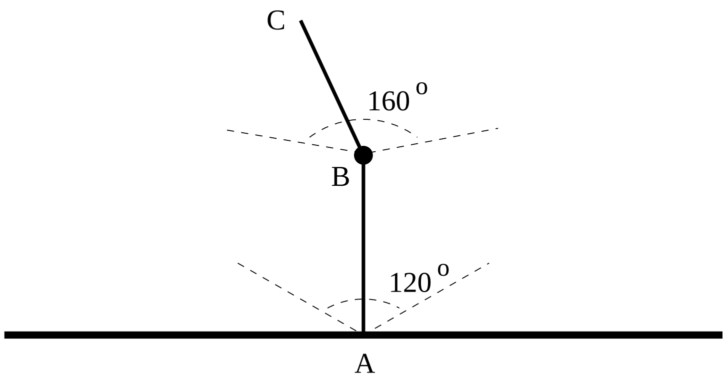
```
>> set_prolog_flag(interval_mode,simple).

true.

>> X ~ cc(0,10), sin(X)=0.

X ~ co(0,'>9.424778').
```

```
>> set_prolog_flag(interval_mode,union).

true.

>> X ~ cc(0,10), sin(X)=0.

X ~ 0 u oo('>3.1415925','>3.1415927')
      u oo('<6.283185','<6.283186')
      u oo('>9.424777','>9.424778').
```

Un petit problème de trigonométrie

On s'intéresse ici à un bras manipulateur composé de deux bras pouvant se déplacer dans le plan :

1. Le premier est fixé en $(0, 0)$, de taille 1, et peut se déplacer suivant un angle de 30 à 150 degrés.
2. Le deuxième est fixé à l'extrémité du premier, peut faire un angle de -80 à 80 avec celui-ci, et mesure également 1.



On se propose de résoudre le problème suivant : quel est le point le plus éloigné de son attache A que le robot peut toucher sur la table.

On va premièrement décrire en Prolog IV les contraintes de fonctionnement du robot. (X_0, Y_0), (X_1, Y_1), et (X_2, Y_2) représentent respectivement les coordonnées des points A, B, C, et θ_1 et θ_2 les angles des deux bras.

```
bras_robot([[0,0],[X1,Y1],[X2,Y2],Teta1,Teta2]):-
   Teta1_degre ~ cc(30,150),
   Teta1 = Teta1_degre.*.pi./.180,
   Teta2_degre ~ cc(0,360),
   le(abs(Teta2_degre .-. Teta1_degre),60),
   Teta2 = Teta2_degre.*.pi./.180,
   X1 = cos(Teta1),
   Y1 = sin(Teta1),
   X2 = X1 .+. cos(Teta2),
   Y2 = Y1 .+. sin(Teta2).
```

On va donc chercher à minimiser X_2, Y_2 étant fixé à 0. Pour cela, on va utiliser la méthode de recherche de minimum par dichotomie introduite à la section 4.3.

On obtient alors la requête et la réponse suivante ([robot]) :

```
>> bras_robot([_,_,[X,0],T1,T2]),minimizei(X,[T1,T2],MinX).

MinX = -14529499/8388608,
T2 ~ oo('<3.665191','<4.014258'),
T1 ~ oo('>2.268927','>2.617994'),
X ~ oo(-'>1.7320512',-'>1.2855738').
```

Le point le plus éloigné qui puisse atteindre le robot sur la table est donc situé à environ 1.73 (14529499/8388608) du point A.

Le programme complet ([robot]) se trouve en annexe (cf. annexe 9.4.2).

6 Résolution de quelques problèmes délicats

L'objectif de cette section est de montrer quelques méthodes qui peuvent être utilisées pour résoudre des problèmes délicats. On considère ces problèmes "délicats" car leur résolution naïve donne des résultats peu satisfaisants (si résultat il y a !). Les problèmes que l'on a retenus sont tous de nature continue (ne faisant pas intervenir de relation partiellement ou totalement discrète, comme "entier", par exemple), la littérature concernant les problèmes discrets étant beaucoup plus fournie.

Les méthodes que l'on va présenter dans ce chapitre sont de deux natures :

1. Celles qui ont pour but de modifier le comportement du système de résolution de contraintes lui-même :
   - Un traitement des contraintes ne correspondant pas à une relation de base autrement que par une simple décomposition en relations de base (cf. 6.2).
   - Deux traitements particuliers des sous-systèmes linéaires (cf. 6.3), l'un utilisant le solveur linéaire (simplex), et l'autre des contraintes redondantes.
2. Celles qui proposent des méthodes d'énumération permettant de résoudre plus efficacement, ou de manière plus "propre" certaines classes de problèmes.
   - Choix statiques et dynamiques de l'ordre d'énumération (cf. 6.4).
   - Enumération sur une grille précise, donnant aux résultats des énumérations en précision non-atomique une sémantique exacte (cf. 6.5).
   - Contraction et énumération par les bords, permettant entre autre une bonne gestion des problèmes dont la solution n'est pas un point, ou des problèmes n'ayant qu'une solution (cf. 6.6 et 6.7).

Rien n'empêche bien sûr de mélanger plusieurs de ces méthodes.

L'objet de cette section n'est d'ailleurs pas de proposer des méthodes infaillibles ou magiques, "prêt à utiliser", mais plutôt de permettre au lecteur intéressé par les techniques plus fines de résolution avec les intervalles d'avoir un aperçu de certaines d'entre elles.

6.1 Définition et analyse de quelques problèmes

Dans une première étape, on va définir quelques problèmes qui serviront de "cobayes" aux techniques de résolution qui seront détaillées par la suite.

Le premier (Le pentagone) regroupe un grand nombre de difficultés inhérentes à la résolution des problèmes continus avec les intervalles. Les deux autres (Broyden-Banded et Kinematics) sont des tests classiques des systèmes de résolution numériques, sur lesquels beaucoup de gens se sont penchés, beaucoup d'expériences ont été menées.

On définira ici ces problèmes et on détaillera leurs solutions. Puis on donnera une première version naïve de programme Prolog IV, et on fera une première analyse des difficultés rencontrées.

6.1.1 Le pentagone

Soit (C) le cercle de centre 0, et de rayon 1. Soit M_0 le point de (C) de coordonnées $(1,0)$.
On cherche quatre autres points M_1, M_2, M_3, M_4 de coordonnées (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) tels que le système d'équations suivant soit vérifié :

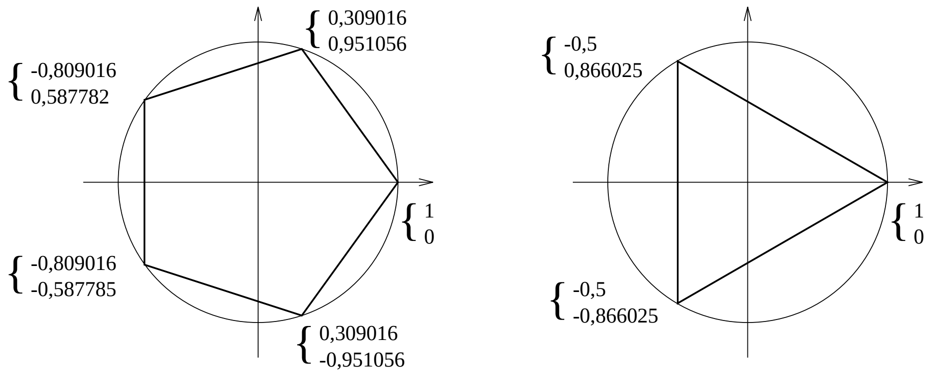
- les points sont sur le cercle : $\forall i \in 1..4,\ x_i^2 + y_i^2 = 1$
- les distances sont égales :

$$\begin{aligned}(x_1-1)^2+(y_1-0)^2 &= (x_2-x_1)^2+(y_2-y_1)^2\\ &= (x_3-x_2)^2+(y_3-y_2)^2\\ &= (x_4-x_3)^2+(y_4-y_1)^2\\ &= (1-x_4)^2+(0-y_4)^2\end{aligned}$$

Les variables du problème sont les deux coordonnées de chaque point M_1, M_2, M_3, M_4. Ce problème a donc huit variables.

Il semble que l'on ait défini là un pentagone. Les solutions sont en fait deux pentagones directs, deux pentagones "en étoiles", dix pentagones dégénérés en triangles, et un pentagone dégénéré en un point : M_0. Soit en tout, 15 solutions.

Sur le schéma suivant, vous pouvez observer la morphologie des pentagones (purs et dégénérés en triangles) :



Le programme Prolog IV suivant ([penta]) est la version la plus intuitive de ce problème, la simple traduction des contraintes mathématiques.

```
pentagone([X2,Y2,X3,Y3,X4,Y4,X5,Y5,D]):-
   square(X2.-.1) .+. square(Y2) = D,
   square(X3.-.X2) .+. square(Y3.-.Y2) = D,
   square(X4.-.X3) .+. square(Y4.-.Y3) = D,
   square(X5.-.X4) .+. square(Y5.-.Y4) = D,
   square(X5.-.1) .+. square(Y5) = D,
   square(X2) .+. square(Y2) = 1,
   square(X3) .+. square(Y3) = 1,
   square(X4) .+. square(Y4) = 1,
   square(X5) .+. square(Y5) = 1.

>> pentagone([X2,Y2,X3,Y3,X4,Y4,X5,Y5,D]),
   realsplit([X2,Y2,X3,Y3,X4,Y4,X5,Y5],_,0.1).
```

Cette requête, dont nous vous épargnerons les 76 pseudo-solutions, s'exécute dans un temps assez long (cf. 9.2), même si la complexité de ce problème est plus importante qu'il n'y parait. Ce résultat est d'autant plus décevant que la précision demandée (et obtenue) n'est pas très importante.

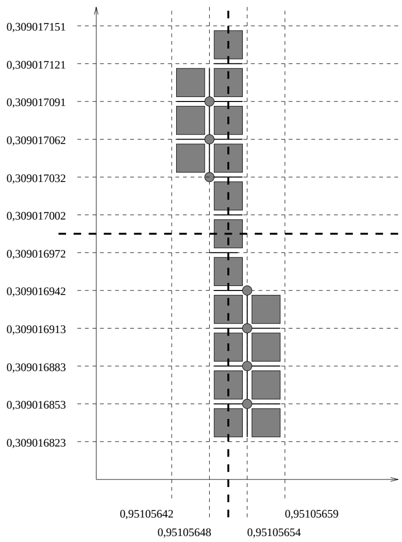
Une première analyse du problème

L'étude de ce problème, apparemment assez simple, recèle plusieurs difficultés et spécificités liées à la résolution des problèmes de nature continue. On va aborder ici trois ce ces aspects :

1. Le problème le plus classique est la multiplicité des pseudo-solutions atomiques correspondant à une seule solution du problème initial. Sur l'exemple de requête précédent, on peut observer que le programme trouve, pour la solution la plus triviale du problème (pentagone direct) quelque 333 443 solutions atomiques (dont 333 442 solutions parasites), et bien plus encore pour les autres solutions.

   En précision atomique, ce très grand nombre de pseudo-solutions correspond tout simplement à la propagation des erreurs de calcul sur les rationnels IEEE 754. On peut d'ailleurs remarquer que si le résultat était un pavé régulier de $I\!R^8$ (puisqu'il y a 8 variables), son côté mesurerait environ l'écart entre 5 flottants, ce qui est faible. La précision obtenue est donc de très bonne qualité.

   Pour la première solution "mathématique" du problème du pentagone (pentagone direct), voici la projection des solutions atomiques sur les variables (X_2, Y_2) (les pointillés représentent la position de la solution mathématique parmi les pseudo-solutions) :



2. La contrainte "Etre deux éléments différents" existe naturellement parmi les relations de base du système de résolution avec les intervalles. Il ne faut cependant pas trop en attendre car elle n'a finalement de conséquence que si l'une des variables au moins est, ou devient un rationnel IEEE 754.

   En effet, $\{X \neq 1/3\}$ (1/3 n'est pas un rationnel IEEE 754) est équivalent à $\{X \neq A, A \in (0.3333333, 0.33333334)\}$. Or, rien ne peut être déduit de ce système de contraintes.

   Il est donc particulièrement délicat d'exprimer de manière efficace que deux variables représentent des valeurs distinctes (sauf dans des cas spécifiques : les problèmes entiers, par exemple). En ce qui concerne le problème du pentagone, cela signifie qu'il sera très difficile de contraindre les cinq points du pentagone à être distincts pour éviter l'apparition de pentagones dégénérés.

   On remarquera cependant que $\{1/3 \neq 1/3\}$ provoque un échec Prolog, ce qui pourrait paraître en contradiction avec ce que l'on vient de dire. Cette déduction n'est en fait pas due à l'algorithme de résolution avec les intervalles, mais à la "calculatrice rationnelle" (cf. 2.6).

3. Enfin, le dernier point est plus spécifique à ce problème : il est lié à l'hyper-densité des rationnels IEEE 754 autour de 0 (cf. 2.2), certaines solutions (les pentagones dégénérés) contenant des 0.

   Observons ce phénomène sur une exemple plus simple :

   Supposons que $X = 0, Y = 1, Z = 1$ soit la solution d'un problème, et que X ne se déduise du système de contraintes que par la relation $\{X = Y - Z\}$. Si Y et Z peuvent varier de 2 ou 3 flottants à cause des erreurs de calculs, cela signifie qu'ils peuvent varier d'environ $3.6 * 10^{-7}$. La variation de X est donc de l'ordre de $7.2 * 10^{-7}$. Mais cette variation de X, au voisinage de 0, représente 1 769 996 288 rationnels IEEE 754. Cette disparité fait alors apparaître un nombre de solutions parasites tellement grand qu'il devient impossible de les calculer toutes.

6.1.2 Broyden-Banded

Soit H_n le problème suivant (n équations, n variables) [8, 9] :

$$\forall i \in 1..n, x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) = 0$$

avec $J_i = \{j \in I\!N \mid j \neq i, MAX(1, i-5) \leq j \leq MIN(n, i+1)\}$.

Ce problème n'a, pour tout n, qu'une seule solution.

Voici maintenant deux interprétations en Prolog IV de ce problème. La première est totalement candide ([broyden(N)]), et la deuxième tire parti des méthodes de regroupement de sous-expressions développées dans la section 3.2.2 ([broydenSE(N)]).

```
bound_list([]).
bound_list([X|L]):-
   cc(X,-1,1),
   bound_list(L).

put_constraints(V,N.+.1,N):-!.
put_constraints(V,I,N):-
   build_sum(V,N,I,S),
   V:I .*. (2 .+. 5 .*. square(V:I)) .+. 1 .-. S = 0,
   put_constraints(V,I.+.1,N).

build_sum(V,N,I,S):-
   build_sum2(V,I,max(1,I.-.5),min(N,I.+.1),S).


build_sum2(V,I,J.+.1,J,0):-!.
build_sum2(V,I,I,F,S):-
  !,
   build_sum2(V,I,I.+.1,F,S).
build_sum2(V,I,J,F,S):-
   build_sum2(V,I,J.+.1,F,S1),
   S = S1 .+. V:J .*. (1 .+. V:J).
```

```
>> size(L)=5,bound_list(L),
   put_constraints(L,1,5),realsplit(L,_,0.01).

L ~ [co(-‘>0.42913693‘,-‘>0.4271791‘),
     co(-‘<0.4778505‘,-‘>0.4743888‘),
     co(-‘>0.5224821‘,-‘<0.5162722‘),
     oc(-‘<0.56054‘,-‘>0.5590075‘),
     oo(-‘>0.5605235‘,-‘<0.5590331‘)];

L ~ [co(-‘>0.42913693‘,-‘>0.4271783‘),
     co(-‘<0.4778505‘,-‘>0.47438415‘),
     co(-‘>0.5224821‘,-‘<0.5162429‘),
     oc(-‘<0.5607114‘,-‘>0.5590075‘),
     co(-‘<0.5590331‘,-‘>0.5567972‘)];

L ~ [co(-‘>0.42913693‘,-‘<0.4271784‘),
     co(-‘<0.4778505‘,-‘>0.47438454‘),
     co(-‘>0.5224821‘,-‘>0.5162453‘),
     oo(-‘>0.5590075‘,-‘>0.5568656‘),
     oo(-‘<0.5605509‘,-‘>0.5591736‘)];

L ~ [co(-‘>0.42913693‘,-‘>0.427178‘),
     co(-‘<0.4778505‘,-‘<0.4743825‘),
     co(-‘>0.5224821‘,-‘>0.5162322‘),
     oo(-‘>0.5590075‘,-‘>0.556769‘),
     co(-‘>0.5591736‘,-‘<0.556663‘)].
```

Dans cette deuxième interprétation en Prolog IV, l'expression $x(x+1)$ qui est très fortement redondante dans le problème, est calculée une seule fois, et stockée dans une liste.

```
bound_list([]).
bound_list([X|L]):-
   cc(X,-1,1),
   bound_list(L).
```

```
put_constraints(V,N):-
   build_expression(V,Lexpr),
   build_constraints(V,Lexpr,N,1).

build_expression([],[]).
build_expression([X|L],[E | LL]):-
   E = X .*. (1 .+. X),
   build_expression(L,LL).


build_constraints(_,_,N,N+1):-!.
build_constraints(V,Le,N,I):-
   build_sum(Le,N,I,S),
   V:I .*. (2 .+. 5 .*. square(V:I)) .+. 1 .-. S = 0,
   build_constraints(V,Le,N,I+1).

build_sum(L,N,I,S):-
   build_sum2(L,I,max(1,I.-.5),min(N,I.+.1),S).


build_sum2(_,_,J.+.1,J,0):-!.
build_sum2(L,I,I,F,S):-
  !,
   build_sum2(L,I,I.+.1,F,S).
build_sum2(L,I,J,F,S):-
   build_sum2(L,I,J.+.1,F,S1),
   S = S1 .+. L:J.

>> size(L)=5,bound_list(L),
   put_constraints(L,5),realsplit(L,_,0.01).

L ~ [co(-‘>0.42913693‘,-‘<0.4271871‘),
     co(-‘<0.4778505‘,-‘>0.47443315‘),
     co(-‘<0.5223216‘,-‘>0.5163808‘),
     oo(-‘>0.5608002‘,-‘<0.559122‘),
     co(-‘<0.559978‘,-‘>0.5560226‘)];
```

```
L ~ [co(-`>0.42913693`,-`>0.4271852`),
     co(-`<0.4778505`,-`>0.4744226`),
     co(-`<0.5223216`,-`<0.5163138`),
     co(-`<0.559122`,-`<0.5564967`),
     co(-`<0.559978`,-`<0.5559505`)].
```

On observe immédiatement une diminution sensible du temps d'exécution (environ deux fois plus rapide : cf. 9.2), accompagnée d'une diminution du nombres de pseudo-solutions approchées trouvées.

Dans la suite, c'est à partir de cette deuxième version, mieux structurée, et plus efficace, que l'on travaillera.

Il n'est pas aisé de dire ici *a priori* (sans références à d'autres systèmes ou méthodes de résolution) si les résultats obtenus sont satisfaisants.

Le tableau suivant permet d'avoir une idée de l'évolution des temps d'exécution en fonction de n :

achille noopt

n	Temps T_n	Nb pseudo-sols	$T_n/T_{n/2}$
5	7.9	2	-
10	40.3	9	5.1
20	597.6	150	14.8
40	> 110 326	-	> 184.6

L'évolution du temps d'exécution en fonction de n laisse penser que la complexité de la résolution est exponentielle. On est donc dans le plus mauvais cas.

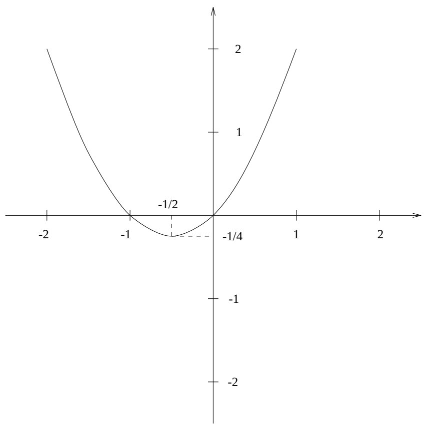
Une première analyse du problème

Il y a manifestement deux causes prépondérantes aux difficultés que l'on rencontre en résolvant ce problème :

1. Tout d'abord, les polynômes $f(x) = x(x+1)$ et $g(x) = x(2+5x^2)$ ne sont pas traités de manière satisfaisante par le système de résolution du fait des occurrences multiples de la variable x dans ces deux polynômes (cf. 3.2.1).

Prenons par exemple le polynôme f. On peut aisément calculer que $f([-1,0])$ est égal à $[-1/4,0]$.

Voici le graphe de f :



Or le système de résolution n'est capable que d'indiquer que $f([-1,0])$ est inclus dans $[-1,0]$.

On étudiera un moyen de contourner cette difficulté directement à la section 6.2.

2. D'autre part, on retrouve un problème similaire à celui rencontré avec le pentagone quant au nombre de pseudo-solutions produites par le système de résolution avec les intervalles. A la vue du tableau de résultat précédent, il semblerait que la seule croissance de leur nombre soit un handicap suffisant pour interdire la résolution de ce problème pour de plus grandes valeurs de n.

   Comme on le verra à la section 6.2, résoudre le problème des variables multi-occurrentes permet de résoudre ce système d'équations pour des valeurs de n bien plus grandes. On peut cependant suivre une autre voie en s'attaquant directement au problème du nombre de pseudo-solutions produites par l'énumération elle-même, comme on le verra aux sections 6.6 et 6.7.

6.1.3 Kinematics

Le problème que l'on va décrire maintenant représente les équations de contraintes d'un bras manipulateur de robot.

$$\begin{array}{l}
-S_2C_5S_6 - S_3C_5S_6 - S_4C_5S_6 + C_2C_6 + C_3C_6 + C_4C_6 - 0.4077 = 0 \\
C_1C_2S_5 + C_1C_3S_5 + C_1C_4S_5 + S_1C_5 - 1.9115 = 0 \\
S_2S_5 + S_3S_5 + S_4S_5 - 1.9791 = 0 \\
C_1C_2 + C_1C_3 + C_1C_4 + C_1C_2 + C_1C_3 + C_1C_2 - 4.0616 = 0 \\
S_1C_2 + S_1C_3 + S_1C_4 + S_1C_2 + S_1C_3 + S_1C_2 - 1.7172 = 0 \\
S_2 + S_3 + S_4 + S_2 + S_3 + S_2 - 3.9701 = 0 \\
S_1^2 + C_1^2 = 1 \\
S_2^2 + C_2^2 = 1 \\
S_3^2 + C_3^2 = 1 \\
S_4^2 + C_4^2 = 1 \\
S_5^2 + C_5^2 = 1 \\
S_6^2 + C_6^2 = 1
\end{array}$$

Ce problème possède seize solutions distinctes. Le programme Prolog IV ci-après est la transcription exacte des contraintes précédentes ([kine]) :

```
constraints([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6]):-
   .-.S2.*.C5.*.S6 .-. S3.*.C5.*.S6 .-. S4.*.C5.*.S6
      .+. C2.*.C6 .+. C3.*.C6 .+. C4.*.C6 .-. 0.4077 = 0,
   C1.*.C2.*.S5 .+. C1.*.C3.*.S5 .+. C1.*.C4.*.S5
      .+. S1.*.C5 .-.1.9115 = 0,
   S2.*.S5 .+. S3.*.S5 .+. S4.*.S5 .-. 1.9791 = 0,
   C1.*.C2 .+. C1.*.C3 .+. C1.*.C4
      .+. C1.*.C2 .+. C1.*.C3 .+. C1.*.C2 .-. 4.0616 = 0,
   S1.*.C2 .+. S1.*.C3 .+. S1.*.C4
      .+. S1.*.C2 .+. S1.*.C3 .+. S1.*.C2 .-. 1.7172 = 0,
   S2 .+. S3 .+. S4 .+. S2 .+. S3 .+. S2 .-. 3.9701 = 0,
   square(S1).+.square(C1)=1,
   square(S2).+.square(C2)=1,
   square(S3).+.square(C3)=1,
   square(S4).+.square(C4)=1,
   square(S5).+.square(C5)=1,
   square(S6).+.square(C6)=1.
```

```
>> constraints(L),realsplit(L,_,0.01).
```

Prolog IV calcule 5 073 pseudo-solutions approchées après un très long temps de calcul (cf. annexe 9.2). Cela fait beaucoup de temps, et beaucoup de solutions. Ce résultat n'est manifestement pas satisfaisant.

Comme on l'a déjà fait pour l'exemple précédent, on va proposer une simplification formelle de l'énoncé du problème. On ira ici un peu plus loin, puisqu'on va factoriser certains produits de manière à faire apparaître les sous-expressions communes plus aisément. On obtient alors le programme suivant ([kineS]) :

```
constraints([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6]):-
   C5.*.S6.*. uminus(Aux2) .+. C6.*. Aux3 .-. 0.4077 = 0,
   S5.*.C1.*. Aux3 .+. S1.*.C5 .-. 1.9115 = 0,
   S5 .*. Aux2 .-. 1.9791 = 0,
   C1.*. Aux1 .-. 4.0616 = 0,
   S1.*. Aux1 .-. 1.7172 = 0,
   Aux4 .-. 3.9701 = 0,
   Aux1 = 3.*.C2 .+. 2.*.C3 .+. C4,
   Aux4 = 3.*.S2 .+. 2.*.S3 .+. S4,
   Aux2 = S2.+.S3.+.S4,
   Aux3 = C2.+.C3.+.C4,
   square(S1).+.square(C1)=1,
   square(S2).+.square(C2)=1,
   square(S3).+.square(C3)=1,
   square(S4).+.square(C4)=1,
   square(S5).+.square(C5)=1,
   square(S6).+.square(C6)=1.

>> constraints(L),realsplit(L,_,0.01).
```

Prolog IV ne calcule plus que 168 pseudo-solutions, et l'exécution est un peu plus de 2 fois plus rapide. Comme on peut le voir une nouvelle fois, des manipulations formelles très simples aident considérablement le système de résolution, en lui apportant certaines informations formelles qu'il ne peut déduire seul.

Une première analyse du problème

La simplification du problème fait apparaître clairement un sous-système totalement linéaire dans l'expression du problème. Ce type de système reposant sur des liens formels entre les variables, les mécanismes de propagation de domaines sont relativement impuissants pour les résoudre.

On verra à la section 6.3 comment on peut traiter ce type de problèmes.

6.2 Traitement global des expressions complexes

6.2.1 Introduction

Comme on l'a vu au paragraphe 3.2.1, certaines expressions ne correspondant pas à des contraintes de base du système de résolution ne sont pas traitées de manière satisfaisante. Plusieurs solutions sont alors envisageables :

1. Les solutions que nous appellerons extérieures, faisant intervenir des mécanismes étrangers au système de résolution avec les intervalles. Il peut s'agir de solveurs linéaires (dont on verra un exemple à la section 6.3.3), bases de Groëbner (pour les polynômes), manipulation formelles diverses, Newton, etc...

2. Les solutions reposant exclusivement sur les deux outils dont on dispose : l'algorithme du point fixe et l'énumération.

C'est cette deuxième solution que l'on va présenter dans cette section. Après en avoir décrit plus précisément le principe (également décrit dans [18, 19]), on montrera que cette technique peut être implantée directement en Prolog IV, et on détaillera son implantation. On montrera ensuite sur des exemples (notamment sur Broyden-Banded) les améliorations que l'on peut obtenir. Enfin, on détaillera les aspects sémantiques et théoriques de cette méthode.

6.2.2 Principe et implantation

Principe

Revenons tout d'abord sur le fonctionnement du point fixe : un ensemble de relations de bases portant sur un ensemble de variables est réévalué pour tenter de réduire les domaines des variables. Cette réévaluation consiste essentiellement à calculer l'approximation de la projection de l'intersection d'une relation avec un produit cartésien d'intervalles (voir le chapitre 2). Si une contrainte ne correspond pas à l'une des relations de base du système de résolution, celle-ci est décomposée en relations de bases, en faisant intervenir des variables intermédiaires.

Supposons maintenant que l'on souhaite obtenir un comportement optimal du système de résolution sur certaines contraintes (ne correspondant pas aux relations de base). On souhaite en fait que celui-ci se comporte comme s'il connaissait une nouvelle relation de base.

On va donner le statut de "relations spéciales" à ces relations. Le mécanisme du point fixe connaîtra alors deux types de relations :

1. Les relations de base

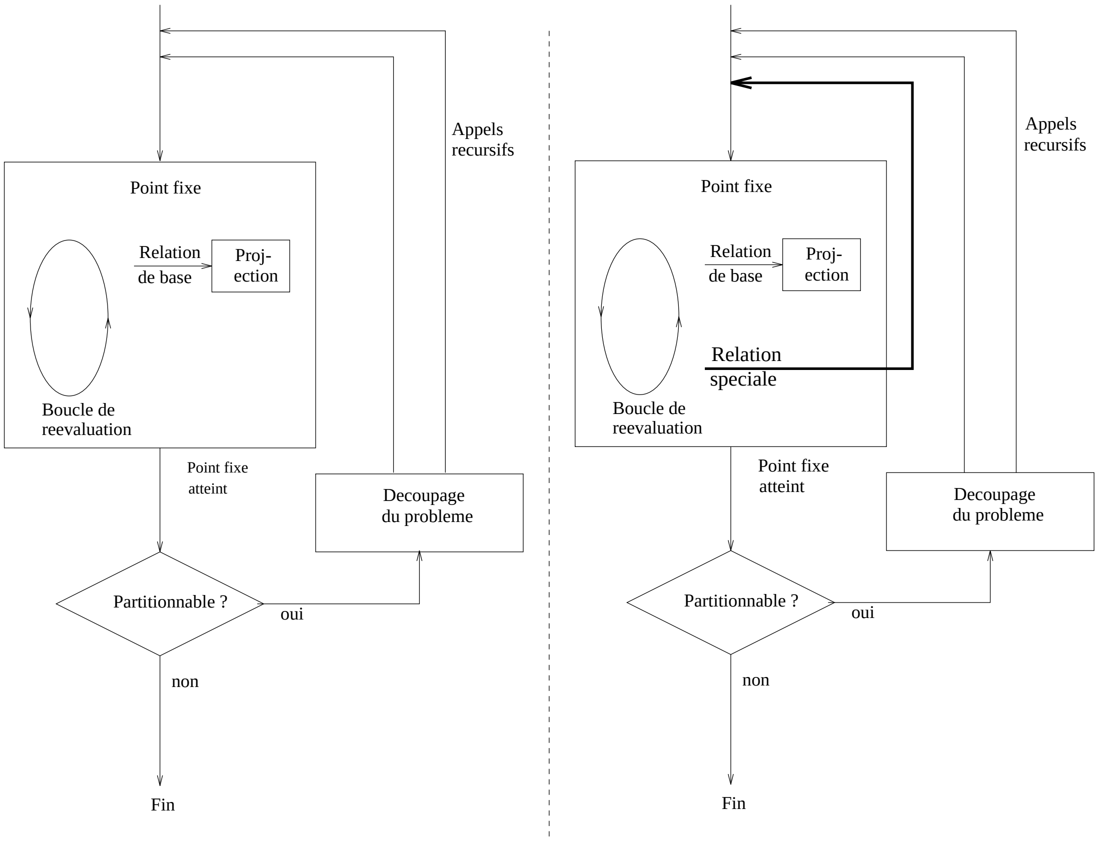
2. Les relations spéciales

Le traitement de la réévaluation des relations de base reste inchangé, et celui des relations spéciales se fait de la manière suivante :

Les relations spéciales sont décomposées en relations de base (comme c'est déjà le cas des autres relations intervenant dans un problème). Lors de chaque réévaluation d'une relation spéciale, on va chercher les minimum et maximum de chacune des variables de la relation spéciale (et pas des variables auxiliaires apparues lors de la décomposition de la relation spéciale en relations de base). Ces recherches d'optima se font grâce aux techniques décrites à la section 4.3. Les domaines des variables de la relation spéciale sont alors réduits en utilisant les optima calculés.

Il est aisé d'imaginer que les calculs nécessaires à la réévaluation d'une relation spéciale sont beaucoup plus lourds que pour une relation de base.

Le schéma suivant permet de comparer le fonctionnement du système de résolution initialement décrit (cf. chapitres 2 et 4) et celui que l'on vient d'introduire :



Comme on peut le constater sur ce schéma, la différence fondamentale entre l'algorithme initial et cet algorithme modifié tient dans la récursion mutuelle qui apparaît entre point fixe et énumération.

Quelques remarques :

1. On peut remarquer (et on le démontrera à la section 6.2.4) que l'algorithme n'est pas de nature à modifier les résultats produits par le couple Point fixe / Enumération. Intuitivement, on peut dire que l'on

ne fait que déplacer une partie de l'énumération à un niveau plus local ; les calculs ne sont pas faits au même moment, mais au bout du compte, ce sont les mêmes.

2. Comme on l'a sous-entendu, il est naturel que ce traitement spécifique soit optionnel car son coût peut être très important, et il n'est bien souvent d'aucune utilité.

Implantation en Prolog IV

L'algorithme, tel qu'on l'a décrit, suppose une modification du fonctionnement du point fixe qui n'est naturellement pas possible en Prolog IV.

On va donc introduire un algorithme que l'on peut écrire en Prolog IV, et qui permet de simuler très efficacement des modifications de comportement de l'algorithme du point fixe. On verra même un peu plus loin que cette méthode s'avèrera généralement plus efficace qu'une modification du point fixe au niveau de l'implantation.

Cette technique, que l'on nomme "méta-Point fixe" permet en fait de donner un comportement nouveau à n'importe quelle contrainte. On va s'en servir ici pour implanter l'algorithme que l'on vient de décrire, mais on s'en servira également dans la section 6.3.

Le "méta-Point fixe" est en fait un point fixe géré explicitement en Prolog IV, fonctionnant de la manière suivante :

Soit (S) un système de contraintes décomposé de la manière suivante :

1. Un ensemble (S_1) de contraintes de base. On ne détaillera pas sa structure puisque sa résolution sera effectuée de manière standard par le système de résolution avec les intervalles.

2. Un ensemble (S_2) de contraintes spéciales. Chaque contrainte spéciale est représentée par un triplet $[type, V_1, V_2]$:

   (a) *type* est le type de la relation spéciale associée. Plus précisément, c'est le nom d'un prédicat Prolog IV qui sera chargé de réévaluer cette relation spéciale.

(b) V_1 est la liste des variables propres à la relation spéciale. Ces variables ne doivent pas apparaître ailleurs dans le système de contraintes.

(c) V_2 est une liste de variables du système (S_1). Les variables de V_1 et V_2 doivent se correspondre.

L'algorithme du "méta-Point fixe" est alors le suivant :

```
META_PF(S)
    Répéter
        POINT_FIXE(S1)
        Pour tout triplet [type, V1, V2] de (S2)
            Si l'une des variables de V2 a un domaine plus
            réduit que son homologue de V1
                Transférer les domaines de V2 vers V1 (TR1)
                Appeler le prédicat type(V1)
                Transférer les domaines de V1 vers V2 (TR2)
    Tant que l'un des transferts (TR2) a provoqué la réduction
    du domaine d'une des variables de (S1)
```

Le programme complet du méta-Point fixe écrit en Prolog IV se trouve à l'annexe 9.4.3.

Comme on l'a déjà dit, cet algorithme est, en général, plus efficace qu'une modification du point fixe lui-même.

Pourquoi?

Parce que la réévaluation des relations spéciales étant beaucoup plus complexe que celle des relations de base, elle est beaucoup plus coûteuse en temps d'exécution. Il est donc logique de chercher à ne traiter les relations spéciales qu'en dernier lieu, et donc, potentiellement, le moins souvent possible. Dans la majorité des programmes utilisant cette technique, le gain par rapport à une modification du point fixe est important : de 2 à 5 fois plus rapide.

Cependant, ce type de comportement peut être implanté à bas niveau grâce à des techniques de point fixe à plusieurs files, de priorité de réévaluation diverses.

6.2.3 Résultats expérimentaux

Broyden-Banded

Comme on l'a remarqué en introduisant le problème de Broyden-Banded, une des difficultés rencontrées provient du traitement peu satisfaisant des polynômes $f(x) = x(x+1)$ et $g(x) = x(2+5x^2)$. On va donc ici leur conférer un comportement complet, en utilisant la technique que l'on vient de développer.

On va pour cela créer deux vecteurs contenant ces deux expressions pour chaque x_i, $i \in 1..n$. Les variables de ces relations spéciales seront réduites explicitement en Prolog par le prédicat `shrink_box` qui calcule les minimum et maximum d'une variable en utilisant la méthode simple de calcul d'optimum décrite à la section 4.3.

Le type des contraintes spéciales sera donc partagé :

1. L'identificateur `to_shrink` décrivant le mécanisme général de réduction des domaines par calculs d'optima.

2. Les contraintes (de base, celles-ci) portant sur les variables propres à la contrainte spéciale, lui servant de définition.

On aurait bien sur pu avoir un identificateur de type pour chaque type de relation spéciale (ici, deux différentes). Mais cela nous aurait conduit à reposer à chaque réévaluation d'une contrainte spéciale l'ensemble des contraintes de base la décomposant (ce qui est plus inutile que coûteux).

Voici le programme Prolog IV qui, étant donnée une liste de variables liées par un système de contraintes de base, réduit leur domaines par calculs d'optima :

```
to_shrink(L):-
   shrink_box(L,L,10000).

shrink_box([],Lc,P).
```

```
shrink_box([X|L],Lc,P):-
   % Appels aux prédicats d'optimisation
   lower_boundi(X,Lc,Min,smallest_domain,
                          prec(P),float_mode),
   upper_boundi(X,Lc,Max,smallest_domain,
                          prec(P),float_mode),
   cc(X,Min,Max),
   shrink_box(L,Lc,P).
```

Il ne reste plus qu'à construire les listes de contraintes spéciales, et à utiliser une énumération construite autour du méta-Point fixe que l'on vient de décrire. Le programme complet ([BrydenG(n)]) se trouve à l'annexe 9.4.4.

Voyons maintenant quels résultats on peut obtenir en utilisant ce type de techniques. Le tableau suivant compare les résultats obtenus avec la méthode naturelle (ou presque) (cf. 6.1.2) et la version que l'on vient de détailler, qui utilise le traitement global de certaines contraintes complexes et la notion de méta-Point fixe : achille noopt

n	Temps T_n (naturel)	$T_n/T_{n/2}$	Temps T'_n (méta)	$T'_n/T'_{n/2}$
5	7.9	-	8.7	-
10	40.3	5.1	16.8	1.9
20	597.6	14.8	35.4	2.1
40	$>$ 110 326	$>$ 184.6	76.2	2.2
80	-	-	159.5	2.1

Ces résultats montrent que cette technique de traitement global est parfaitement adaptée à la résolution de ce type de problème. De plus, le temps de résolution semble, à la vue des expérimentations, être quasiment linéaire, ce qui est manifestement ce que l'on peut faire de mieux.

6.2.4 Sémantique

On va s'attacher dans ce paragraphe à donner une définition plus précise de l'algorithme de résolution que l'on a introduit.

On en déduira alors la propriété fondamentale de cet algorithme qui atteste du fait que le résultat obtenu après énumération est identique à celui obtenu avec une énumération et un point fixe classiques.

Pré-requis : il est conseillé pour aborder ce paragraphe d'avoir lu les sections décrivant la sémantique du système de résolution (cf. 2.4), et la sémantique de l'énumération (cf. 4.1.3).

Définition de la méthode

Soit $(P) = (r_i)_{i \in I_1} \cup (r_i)_{i \in I_2}$ un problème sur n variables dans $\mathbb{R}$,
où les $(r_i)_{i \in I_1}$ sont des relations normales, c'est à dire des relations de base, et les $(r_i)_{i \in I_2}$ sont des relations spéciales.

On définit l'approximation d'une relation spéciale de la manière suivante :

Soit r une relation spéciale.
Soit $(r_i)_{i \in I_r}$ une famille de relation de base sémantiquement équivalente à r.
On dit que $(r_i)_{i \in I_r}$ est la décomposition de r en relations de base.

Soit $(S_i)_{i \in E_r}$ l'ensemble des solutions atomiques de $appx_{\mathbb{I}}((r_i)_{i \in I_r})$.
On définit alors l'approximation d'une relation spéciale de la manière suivante :

$$appx_{\mathbb{I}}(r) = \left(appx_{\mathbb{I}}\left(\bigcup_{i \in K} S_i\right) \right)_{K \subset E_r}$$

L'approximation de r est donc définie comme étant la relation sur $\mathbb{I}$ formée des approximations d'unions de solutions atomiques de $appx_{\mathbb{I}}((r_i)_{i \in I_r})$.

Propriété fondamentale

Soit $(r_i)_{i \in I_1}$ une famille de relations de base.
Soit $(r_i)_{i \in I_2}$ une famille de relations spéciales.
Soit $(r_i)_{i \in I_3}$ une famille de relations de base, sémantiquement équivalente à

$(r_i)_{i \in I_2}$, c'est à dire décomposant les $(r_i)_{i \in I_2}$ en relations de base.

On pourra considérer, en les étendant éventuellement, que toutes les relations sont n-aires, les deux premières portant initialement sur sur les p premières dimensions.

Soit $(S_i)_{i \in E}$ la projection sur les p premières dimensions des solutions atomiques du problème $appx_{\mathbb{I}}((r_i)_{i \in I_1} \cup (r_i)_{i \in I_3})$.
Soit $(S'_i)_{i \in E'}$ les solutions atomiques (dans $\mathbb{I}^p$) du problème $appx_{\mathbb{I}}((r_i)_{i \in I_1} \cup (r_i)_{i \in I_2})$.

Propriété 6.1 $(S_i)_{i \in E} = (S'_i)_{i \in E'}$.

La preuve de cette propriété se trouve à l'annexe 9.3.10, page 203.

En d'autres termes, les solutions atomiques, si l'on se concentre uniquement sur les variables explicites su problème, sont les mêmes.

D'après la caractérisation de l'énumération (cf. propriété 4.3), on peut donc en déduire que les deux systèmes de résolutions (avec ou sans traitement spécifique des expressions complexes) sont, d'un point de vue sémantique, identiques.

On peut donc en conclure que cette technique est parfaitement intégrée dans le système de résolution tel qu'on l'a défini.

6.3 Traitement spécifique des sous-systèmes linéaires

6.3.1 Introduction

Si l'on observe les techniques traditionnelles de résolution des systèmes linéaires, on peut voir qu'ils reposent essentiellement sur des manipulations formelles : multiplications d'équations par des constantes, additions d'équations, etc ...

Comme on l'a déjà fait remarquer à de nombreuses reprises, le système de résolution avec les intervalles ne fait justement aucune manipulation formelle. Il sera donc particulièrement maladroit dans la résolution des problèmes linéaires, ou contenant des sous-systèmes linéaires important (quantitativement ou qualitativement).

L'idée naturelle qui vient à l'esprit pour contourner cette difficulté est la suivante : si les intervalles ne sont pas bien adaptés à la résolution des problèmes linéaires et formels, il y a dans Prolog IV deux outils, qui, eux, sont particulièrement adéquats : tout d'abord, il y a Prolog lui-même, langage de programmation particulièrement adapté pour faire du calcul formel, et ensuite, Prolog IV fournit, parallèlement aux intervalles, un mécanisme de résolution des systèmes linéaires par des méthodes de Gauss/Simplex.

On va donc dans cette section faire collaborer ces deux possibilités avec les intervalles à la résolution de systèmes de contraintes partiellement linéaires.

La première méthode que l'on va présenter reprend la philosophie que l'on a introduite à la section 3.3, en ajoutant des contraintes redondantes dans le système de contraintes. Celles-ci permettent de gérer des petits sous-systèmes linéaires de manière très efficace.

La seconde introduit une technique de collaboration entre le solveur linéaire et les intervalles plus profonde que celle existant originellement dans Prolog IV, uniquement basée sur les transmissions de constantes d'un solveur à l'autre.

6.3.2 La méthode de Fourier

Principe

Il y a deux manières d'envisager la méthode que l'on va introduire ici :

1. La première, la plus "noble", considère que l'on va combler les lacunes du système de résolution avec les intervalles, en matière de raisonnement formel. En effet, un système linéaire peut être considéré comme une somme de liens formels entre des variables. Ces liens en impliquent d'autres, et ce sont ces implications qui ne sont pas faites correctement par les intervalles. On va donc tout simplement rajouter les liens manquant dans le système de résolution avec les intervalles de manière explicite.

2. La deuxième, plus pragmatique, considère que l'on va ajouter au système de contraintes un certain nombre de contraintes redondantes. On va cependant disposer d'une méthode rigoureuse et systématique pour trouver ces contraintes redondantes, et décrire leur conséquence sur la résolution des systèmes de contraintes.

Ces contraintes redondantes seront obtenues en effectuant des éliminations de variables dans le système de contraintes linéaires. Dans l'absolu, toutes les éliminations possibles doivent être faites. Apparaît alors le point faible de cette méthode : le nombre de contraintes redondantes nécessaires croît exponentiellement en fonction du nombre de contraintes et de variables. Elle ne pourra donc être utilisée que pour résoudre des systèmes linéaires de petites tailles, ou possédant une structure simple (des systèmes linéaires de grandes tailles, mais très peu liés - c'est à dire ayant peu de contraintes par rapport au nombre de variables - ou au contraire, très liés - ayant pratiquement autant de contraintes indépendantes que de variables - peuvent néanmoins être traités facilement, c'est à dire en générant un nombre raisonnable de contraintes redondantes).

On pourra également, dans certains cas, concevoir des méthodes partielles, ne posant pas toutes les contraintes redondantes, notamment en fonction des variables auxquelles ont s'intéresse directement.

Cette génération de contraintes redondantes peut bien sûr être faite automatiquement par un programme Prolog.

Il est important de préciser que de telles contraintes redondantes donnent au système de résolution avec les intervalles un comportement complet vis-à-vis du problème initial, et ce, modulo les imprécisions dues aux calculs effectués grâce aux rationnels IEEE 754.

Un petit exemple

Considérons le problème suivant :

Le point M, de coordonnées (X, Y) est équidistant des points de coordonnées $(2, 0)$, $(0, 1)$, et $(-1/2, 0)$.

On obtient alors le système de contraintes suivant :
$(X-2)^2 + Y^2 = X^2 + (Y-1)^2 = (X+1/2)^2 + Y^2$

En développant les carrés, et en produisant les contraintes redondantes du système linéaire, on obtient alors le programme suivant :

```
c_system(X,Y):-
   % Basic system
   SqXm2.+.SqY=D,
   SqX.+.SqYm1=D,
   SqXpdemi.+.SqY=D,
   SqX = square(X),
   SqY = square(Y),
   SqXm2 = square(X.-.2),
   SqXpdemi = square(X.+.1/2),
   SqYm1 = square(Y.-.1),
   % Square decomposition
   SqXm2 = SqX .+. 4 .-. 4 .*. X,
   SqXpdemi = SqX .+. 1/4 .+. X,
   SqYm1 = SqY .+. 1 .-. 2 .*. Y,
   % Linear substitutions
   3 .-. 4 .*. X .+. 2 .*. Y = 0,
   15/4 .-. 5 .*. X = 0,
   3/4 .-. 2 .*. Y .-. X = 0.
```

On a alors la requête et la réponse suivante :

```
>> c_system(X,Y).

Y = 0,
X = 3/4.
```

Cette solution est obtenue après un temps de calcul non mesurable, alors qu'il est particulièrement délicat de résoudre ce problème sans traitement spécifique du système linéaire qu'il contient. Dans ce cas, il est d'ailleurs nécessaire de borner les variables du problème, sinon, un très grand nombre de pseudo-solutions parasites sera produit lors de l'énumération.

Traitement de kinematics

On va reprendre ici la deuxième version du problème introduit dans le paragraphe 6.1.3, et ajouter les contraintes redondantes permettant de traiter correctement le sous-système linéaire apparaissant dans ce problème.

Le programme que l'on obtient ainsi (qui se trouve à l'annexe 9.4.5, page 221) donne des résultats bien meilleurs en terme de vitesse d'exécution, de 2 à 5 fois plus rapides suivant la technique d'énumération (les temps d'exécution sont précisés à l'annexe 9.2, page 189).

On peut donc, en utilisant les contraintes redondantes linéaires, obtenir des résultats satisfaisants dans des temps corrects.

6.3.3 Le mélange Intervalles/Simplex

Collaboration Linéaire - Intervalles

Le principe de la collaboration entre un solveur linéaire et un solveur à base d'intervalles est intéressant *a priori* pour résoudre des problèmes contenant une partie linéaire non triviale (difficilement résolue par les intervalles, mais parfaitement traitée par des algorithmes spécialisés), et une partie non linéaire (ou discrète, etc...), qui ne peut être traitée par les algorithmes de traitement linéaire.

On a étudié dans le paragraphe précédent une première stratégie pour résoudre de tels problèmes, avec l'adjonction de contraintes redondantes dans le système de contraintes produites par des méthodes dérivées des techniques de Fourier. Ces techniques sont très simples à appliquer, et très efficaces, pour peu que la partie linéaire du système de contraintes soit peu importante. Les deux techniques que l'on va présenter ici complètent cette première approche dans la mesure où elles s'appliqueront avec plus de succès sur des systèmes essentiellement linéaires, et contenant une part non linéaire.

Ces deux méthodes représentent en fait la même idée très simple : faire communiquer les deux solveur par les bornes des intervalles. Seule l'approche diffère :

1. L'approche la plus naturelle du point de vue d'un système de résolution

avec les intervalles, et qui s'avère également la plus efficace, consiste à introduire dans ce système une nouvelle contrainte, liant une matrice constante A, un vecteur constant C, et un vecteur de variables X par l'ensemble de contraintes linéaires $C = A.X$. L'implantation d'une telle contrainte faisant appel à des technologies basées sur des simplex bornés.

2. L'autre approche consiste à faire collaborer les deux solveurs sur un pied d'égalité en leur faisant produire successivement des déductions sur les bornes des variables.

La technique que l'on va présenter ici est une implantation explicite en Prolog IV de la première approche, dans la mesure où la résolution de l'ensemble des contraintes linéaires est traitée d'un seul tenant, comme une contrainte primaire.

Implantation

L'implantation va donc reprendre le principe du méta-Point fixe, introduit à la section 6.2, mais en modifiant le mécanisme de réévaluation de la contrainte. Le programme Prolog IV implantant le méta-Point fixe se trouve à l'annexe 9.4.3.

La réévaluation de la contrainte linéaire globale sera assurée par le prédicat `to_lin` dont voici le contenu en Prolog IV:

```
to_lin(L):-
   create_box(L,LB),
   copy_symbolic_domains(LB,L).

copy_symbolic_domains([],[]).
copy_symbolic_domains([[Min,Max]|LB],[X|L]):-
   cc(X,Min,Max),
   copy_symbolic_domains(LB,L).

create_box(L,_):-
   record(list_box,[]),
```

```
   domains_to_lin(L),
   narrow_lin_box2(L),
   fail.
create_box(_,LB):-
   recorded(list_box,LB),
   dif(LB,no_sol).

domains_to_lin([]).
domains_to_lin([X|L]):-
   bounds(X,Min,Max),
   gelin(X,Min),
   lelin(X,Max),
   domains_to_lin(L).


narrow_lin_box2([]):-!.
narrow_lin_box2([X|L]):-
   lower_bound(X,Min),
   upper_bound(X,Max),
   !,
   narrow_lin_box2(L),
   recorda(list_box,[Min,Max]).
narrow_lin_box2(_):-
   record(list_box,no_sol).
```

Quelques commentaires :

- Le prédicat `to_lin` réduit les domaines (solveur intervalles) d'une liste de variables sur lesquelles portent des contraintes linéaires, en fonction de celles-ci.

- Le prédicat `create_box`, après avoir posé dans le système de contraintes linéaires les contraintes correspondant aux bornes des domaines des variables (prédicat `domains_to_lin`), crée une liste d'intervalles symboliques (prédicat `narrow_lin_box2`), sous la forme d'une liste de couples de rationnels construite sous l'identificateur `list_box`.

  Cette gestion sous forme symbolique permet de provoquer un échec systématique à la fin du travail sur le linéaire (`fail` dans `create_box`).

Cela permet d'éviter l'accumulation de contraintes redondantes dans le système de résolution linéaire, le solveur attaché subissant lourdement de telles répétitions.

- Enfin, le prédicat `copy_symbolic_domains` reprend la liste symbolique, et contraint les variables en appliquant les réductions de domaines obtenues.

Traitement de Kinematics

Cette méthode permet, tout comme celle développée au paragraphe précédent, de traiter la partie linéaire du problème Kinematics. Il est clair que le traitement effectué ici sur cette partie linéaire va être complet. On s'aperçoit en comparant les arbres d'énumération obtenus que le comportement est le même, que l'on utilise l'une ou l'autre des méthodes développées dans cette section. Cette égalité de comportement confirme donc bien le fait que le traitement obtenu en ajoutant les contraintes redondantes est optimal.

Par contre, la lourdeur des mécanismes mis en place pour traiter les contraintes linéaires (appels multiples du solveur linéaire, méta-Point fixe, etc...) ne peut être compensée par la taille du système linéaire impliqué. Les temps d'exécution sont donc nettement à l'avantage de la méthode par ajout de contraintes redondantes (cf. 9.2). Cet état de fait dépendra du rapport de force existant entre les différents types de contraintes présentes dans le système.

Le programme complet ([kineL]) permettant de résoudre Kinematics, en utilisant la technique que l'on vient de décrire se trouve à l'annexe 9.4.6, page 222.

Comparons la taille de l'arbre d'énumération (nombre de points de choix), et le comportement (nombre de pseudo-solutions) obtenus, selon que l'on utilise le système de contraintes de base simplifié ([kineS]), ou le traitement avec le méta-Point fixe et le solveur linéaire ([kineL]).

	[kineS]	[kineL]
Nb. nœuds	1 656	272
Nb pseudo-sols	182	42

6.4 A propos de l'ordre d'énumération

6.4.1 Introduction

Comme on a pu le voir dans le chapitre 4, de nombreux algorithmes d'énumération sont récursifs, et chaque étape est en grande partie conditionnée par le choix de la variable sur laquelle on va travailler. On ne va, dans cette section, s'intéresser qu'aux énumérations relativement naïves, c'est à dire n'étant pas orientées par une étude *a priori* du problème à résoudre (on trouve un exemple d'une telle énumération dans l'exemple des carrés, traité à la section 5.3.4).

On a vu également que plusieurs méthodologies sont disponibles parmi les options préprogrammées des prédicats d'énumération prédéfinis de Prolog IV, et que ces différentes options sont utilisées pour résoudre les exemples présentés au chapitre 5.

Il est alors légitime de se demander si ce choix a une grande influence sur l'efficacité de la résolution, et si oui, comment on peut décider d'un choix *a priori* pour résoudre un problème particulier.

L'objet de cette section est d'apporter un début de réponse à ces questions.

6.4.2 De l'importance du choix de la variable

Choix statique ou choix dynamique

L'unique but que l'on poursuit est de trouver la méthode produisant l'exécution la plus rapide. Cette volonté n'est cependant pas exprimable directement sous cette forme. On est donc amené à tenter de la formuler autrement avec tout ce que cela comporte d'approximation (une de plus!).

On approchera donc souvent cette notion par celle de "choix de la variable ayant le plus d'influence" sur le reste du système ; espérant que cela aboutisse à un arbre d'énumération le plus petit possible.

On va distinguer ici deux grandes catégories, parmi les techniques de choix :

1. Les techniques de choix statiques.

L'idée est ici de définir préalablement au processus d'énumération lui-même, un ordre des variables que l'on va juger approprié. A chaque étape de l'énumération, la variable choisie sera tout simplement la suivante dont le domaine peut encore être scindé (avec éventuellement une restriction éliminant les variables dont le domaine est déjà suffisamment petit).
Ce type de méthodes donne souvent de bons résultats, et se programme très simplement (c'est le cas de l'énumération de base programmée à la section 4.1.2, page 69). Il a cependant un inconvénient majeur : à moins de disposer d'une méthode systématique pour choisir cet ordre statique, il est nécessaire de se livrer à une série de tests ou une analyse du système de contraintes à chaque nouveau problème.

2. Les techniques de choix dynamiques.
   L'idée consiste ici à effectuer le choix, et en fonction de l'état courant du système de contraintes et des domaines des variables.
   Il semble cependant difficile d'exhiber une méthode de choix dynamique donnant de bons résultats pour la quasi-totalité des problèmes. Il faut donc à nouveau jongler avec différentes possibilités.

Les mauvais choix

On va essayer de montrer ici quelques mauvais cas de choix de variable à énumérer, espérant à la fois répondre à l'objet de ce paragraphe (c'est à dire que si les choix sont mauvais, l'efficacité peut en pâtir), et mettre en évidence les pièges qui menacent le procédé d'énumération (car qui sait ce qu'il ne faut pas faire n'est pas loin de savoir ce qu'il faut faire).

- Variable sans influence.
  Choisir une variable qui a peu, voire pas d'influence sur l'ensemble du système conduit à augmenter le temps de l'ensemble de la résolution sans apporter d'information intéressante. Considérons par exemple le système de contrainte suivant :

$$\begin{cases} X + Y = 2 \\ X - Y = 0 \\ Z = X * Y \end{cases}$$

qui a pour unique solution $\{X = Y = Z = 1\}$.
Essayons maintenant de résoudre ce système de contraintes en Prolog IV, en utilisant un ordre d'énumération statique :

```
>> X.+.Y=2, X.-.Y=0,Z=X.*.Y,realsplit([X,Y,Z],list_order).

Z = 1,
Y = 1,
X = 1.
```

(l'arbre d'énumération compte 4 nœuds)
On peut remarquer que si l'on énumère uniquement sur la variable X ou sur la variable Y, on obtient le même résultat. Par contre, si la première variable de la liste d'énumération est Z, l'énumération va probablement être plus longue que votre patience ne pourra le supporter.

- Enumération trop isolée.
  Soit (P) un problème. Soit X et Y deux variables de (P). Si l'énumération de X isolée, ainsi que l'énumération de Y isolée ne permet pas de résoudre aisément (P) alors qu'une énumération alternée de X et Y est capable de déduire des informations plus efficacement, on dira alors que le choix statique de la variable X ou de la variable Y est trop isolé. Ce type de problème assez courant surgit en général dans des systèmes de contraintes assez volumineux.

  L'exemple que l'on va donner ici est petit (2 variables), mais en contrepartie, il n'a pas une signification très intéressante.

  Voici cet exemple : soit une variable booléenne B vraie si et seulement si $X \geq 0$ ou $X < 0$, et si d'autre part $Y \geq 0$ ou $Y < 0$. S'il nous apparaît évident que B est toujours vraie, ce n'est pas une trivialité pour le système de résolution avec les intervalles dans la mesure où, tant que X est inconnu, il est impossible de déterminer la disjonction $X \geq 0$ ou $X < 0$, et donc sa valeur. On ajoute enfin la contrainte B implique non B, qui a la propriété d'échouer si B vaut 1, sans pour autant être capable de figer B à 0.

  En Prolog IV, on obtient donc :

```
>> B = (bge(X,0) bor blt(X,0))
       band (bge(Y,0) bor blt(Y,0)),
B impl bnot(B) , realsplit([X,Y],greatest_domain).


false.
```

L'expérience nous montre que toute autre méthode d'énumération (`list_order` ou `smallest_domain`) conduit à un temps d'exécution multiplié par un facteur de plusieurs millions.
Pourquoi ?
Ce comportement est dû au fait que la première contrainte se résout parfaitement si l'on énumère à la fois sur X et Y, par exemple en alternant le choix de la variable, ou en choisissant celle dont le domaine est le plus grand (ce qui ici revient au même), mais reste non résolu si l'on énumère une seule des deux variables.

6.5 Enumération sur une grille

6.5.1 Introduction

La propriété de caractérisation des solutions de l'énumération développée au paragraphe 4.1.3 nous apporte un résultat très important du point de vue théorique, et rassurant du point de vue pratique : la technique d'énumération est sans influence sur le résultat que l'on obtient, sous réserve que l'énumération sois totale, c'est à dire concerne toutes les variables et tout l'espace, et soit en précision optimale (atomique).

Bien souvent, cependant, une énumération atomique est beaucoup plus lente qu'une énumération en précision relâchée, pour un résultat dont l'intérêt pratique ne justifie pas le coût. Pour cette raison, les énumérations des problèmes de nature réelle ne sont en général pas atomiques, et la propriété de caractérisation ne peut donc s'appliquer. Si cette restriction d'ordre théorique ne gênera probablement pas ceux qui sont plus intéressés par un résultat satisfaisant que par les propriétés théoriques de ce résultat, elle peut toutefois amener à mettre en doute le sens des solutions obtenues.

Illustrons ce problème sur un exemple pratique. Considérons le problème défini par les deux contraintes suivantes : X est le R^{ieme} élément de la liste $[1, 5, 10, 10, 10, 10, 3, 6]$, et R est le R^{ieme} élément de la liste $[7, 8, 10, 10, 10, 10, 1, 2]$.

On va énumérer les variables de ce problème à l'aide du prédicat `intsplit`, avec les options `list_order` (qui indique que l'on énumère les variables dans leur ordre d'apparition dans la liste), et `prec(3)` (qui signifie que l'énumération d'une variable s'arrête lorsque celle-ci contient 3 entiers ou moins (donc, non atomique).

On va alors poser deux requêtes, en mode "unions d'intervalles", en changeant l'ordre de la liste d'énumération :

```
>> X = index([1,5,10,10,10,10,3,6],R),
   R = index([7,8,10,10,10,10,1,2],R),
   intsplit([X,R],list_order,prec(3)).

R ~ 1 u 7,
X ~ 1 u 3;

R ~ 2 u 8,
X ~ 5 u 6.

>> X = index([1,5,10,10,10,10,3,6],R),
   R = index([7,8,10,10,10,10,1,2],R),
   intsplit([R,X],list_order,prec(3)).

false.
```

Comme on peut le constater, alors que dans le deuxième cas, le système s'avère capable de déduire l'inconsistance du système, dans le premier cas, il propose deux pseudo-solutions. Ce petit exemple montre donc bien que, alors que la précision demandée et la méthode d'énumération utilisée sont les mêmes, un simple changement d'ordre des variables a entraîné un comportement totalement différent.

Naturellement, si l'on remplace `prec(3)` par `prec(1)`, c'est à dire si l'on utilise une énumération atomique, alors, quel que soit l'ordre de la liste, l'énumération est capable de conclure à l'inconsistance du système de contraintes.

On va présenter dans cette section une technique d'énumération en précision relâchée, qui, moyennant un surcoût en temps d'exécution modique, permet de retrouver une propriété de caractérisation équivalente à celle donnée pour la précision atomique. Cette technique peut être mélangée à toutes les autres méthodes introduites dans ce chapitre, leur conférant cette "bonne" propriété.

6.5.2 Principe et Implantation

Principe de base

Le principe que l'on va appliquer est assez simple. On va en fait isoler la propriété fondamentale des énumérations atomiques qui leur permet de posséder cette caractérisation des solutions, et la généraliser.

Le principe de base de la caractérisation des solutions des énumérations atomiques repose sur le traitement de chaque produit cartésien d'intervalles atomiques. Bien entendu, on n'effectue jamais un tel travail, mais on montre que toute énumération atomique complète s'y ramène.

On va reproduire ici ce même raisonnement, mais au lieu de se concentrer pour toutes les variables sur le découpage qu'imposent les rationnels IEEE 754 (c'est à dire les intervalles atomiques), on va s'intéresser à un autre découpage plus grossier et **fixe**, des valeurs possibles, pour chaque variable du problème. On va ainsi définir, pour chacune des variables, un **grille fixe** sur laquelle on énumérera, et ce, à l'exclusion de toute autre valeur. Pour des raisons de simplicités de programmation et de notation, on va supposer ici que l'on applique la même grille à toutes les variables, mais on pourrait avoir une grille différente pour chaque variable.

Insistons encore une fois sur l'aspect le plus important : on va choisir ici une grille **fixe**, cela veut dire que l'on est capable de dire à l'avance quelles sont les valeurs qui permettront de procéder au découpage du domaine d'une variable, alors que dans une énumération en précision relâchée plus ordinaire, cette prédiction est impossible. C'est cette spécificité qui va nous permettre de retrouver une propriété de caractérisation satisfaisante.

Implantation

On va maintenant voir qu'il est simple d'implanter une telle méthode en Prolog IV. Reprenons pour cela l'énumération que l'on a décrite au paragraphe 4.1.2, page 69. On va la modifier de telle manière qu'elle se conforme, pour toutes les variables, à une énumération sur la grille constituée de tous les rationnels IEEE 754 dont le rang est un multiple de 1 000. On obtiendra alors une énumération comparable, puisque la condition d'arrêt s'appuyait sur une taille pour chaque domaine inférieure à 1 000 intervalles flottants [1].

Pour alléger le programme, on n'a pas tenu compte des domaines numériques infinis.

```
% Une énumération sur une grille
mon_enum(L):-
   condition_d_arret(L),
   !.
mon_enum(L):-
   choix_variable(L,X),
   milieu(X,M),
   decouper(X,M),
   mon_enum(L).


% Condition de terminaison
condition_d_arret([]).
condition_d_arret([X|L]):-
   \+(domaine_coupe_grille(X)),
   condition_d_arret(L).

% Choix de la variable
choix_variable([X|L],X):-
   domaine_coupe_grille(X),
```

[1] En toute rigueur, il aurait fallu choisir une grille basée sur les rationnels IEEE 754 dont l'ordre est multiple de 500 pour être vraiment équivalent, car entre deux rationnels IEEE 754 encadrant 500 rationnels IEEE 754, il y a environ 1 000 intervalles flottants atomiques, et non 500.

```
   !.
choix_variable([X|L],Y):-
   choix_variable(L,Y).

% Détermination du milieu (sur la grille)
milieu(X,M):-
   bounds(X,Min,Max),
   float_rank(Min,RMin),
   float_rank(Max,RMax),
   (
      gt(RMax .-. RMin,2000),
      !,
      float_rank(M,1000 .*. floor((RMin.+.RMax)./.2000))
   ;
      float_rank(M,1000 .*. (floor(RMin./.1000) .+. 1))
   ).


% Découpage du domaine avec backtracking
decouper(X,M):-
   le(X,M).
decouper(X,M):-
   gt(X,M).

% Y a-t-il un élément de la grille dans le domaine de X
domaine_coupe_grille(X):-
   bounds(X,Min,Max),
   float_rank(Min,RMin),
   float_rank(Max,RMax),
   suivant_sur_grille(RMin,RGrille),
   lt(RGrille,RMax).

% Rang du prochain flottant sur la grille
suivant_sur_grille(R,Rsuivant):-
   Rsuivant = 1000 .*. (floor(R./.1000).+.1).
```

Expérimentation pratique

Reprenons l'exemple donné en introduction, mais en utilisant maintenant une énumération entière sur une grille comprenant tous les entiers multiples de 3, de manière à obtenir une précision de même nature. Voici ce que l'on obtient en posant les mêmes requêtes en Prolog IV (le programme Prolog IV qui implante cette énumération se trouve à l'annexe 9.4.7, page 227).

```
>> X = index([1,5,10,10,10,10,3,6],R),
   R = index([7,8,10,10,10,10,1,2],R),
mon_enum_ent([X,R]).

false.

>> X = index([1,5,10,10,10,10,3,6],R),
   R = index([7,8,10,10,10,10,1,2],R),
mon_enum_ent([R,X]).

false.
```

6.5.3 Caractérisation des solutions obtenues

On va s'attacher dans ce paragraphe à montrer que l'on obtient avec la méthode que l'on vient de présenter une caractérisation des solutions similaire à celle obtenue avec les énumérations complètes et atomiques, tout en bénéficiant des avantages pratiques d'une énumération en précision relâchée.

On déduira principalement de cette caractérisation l'assurance que les résultats obtenus sont indépendants de la technique de dichotomie des domaines des variables, ou de l'ordre d'énumération des variables.

Pré-requis : il est conseillé pour aborder ce paragraphe d'avoir lu les sections décrivant la sémantique du système de résolution (cf. 2.4), et la sémantique de l'énumération (cf. 4.1.3).

Une définition plus générale

On va, dans cette définition de l'énumération en zones définies, étendre quelque peu la notion d'énumération sur une grille que l'on a introduite dans les paragraphes précédents. L'énumération sur une grille est en fait le cas particulier le plus simple et le plus utile de l'énumération en zones définies que l'on va présenter.

Définition 6.2 *Soit* (P) *un problème à* n *variables.*
Soit $(Z_i)_{i\in I}$ *une partition de* $\mathbb{R}^n$*.*
Soit $E = ((F_i)_{i\in Fail}, (S_i)_{i\in Succeed})$ *une énumération de* (P)*.*
E *est une* **énumération en zones définies** *sur* $(Z_i)_{i\in I}$ *si :*

$$\begin{cases} \forall j \in Fail,\ \exists K \subset I,\ F_j = \bigcup_{i\in K} Z_i \\ \forall j \in Succeed,\ S_j \in (Z_i)_{i\in I} \end{cases}$$

Remarque : une énumération sur une grille est tout simplement une énumération en zones définies déterminées par les pavés constituant la grille.

Caractérisation des énumérations en zones définies

Soit (P) un problème à n variables.
Soit $(Z_i)_{i\in I}$ une partition de $\mathbb{R}^n$.
Soit $E = ((F_i)_{i\in Fail}, (S_i)_{i\in Succeed})$ une énumération complète en zones définies sur $(Z_i)_{i\in I}$ de (P).

Propriété 6.3 $(S_i)_{i\in Succeed} = \{Z_i, i \in I \mid appx_{\Pi}(P)_{|Z_i}$ *n'a pas pour solution maximale* $\emptyset\}$

La preuve de cette propriété se trouve à l' annexe *9.3.11, page 203.*

Tout se passe en fait comme si les éléments de zone définis jouaient le rôle des éléments d'approximation atomiques pour les énumérations atomiques complètes.

On retrouve alors naturellement une propriété similaire : le résultat d'une telle énumération de (P) sur $(Z_i)_{i\in I}$ est l'ensemble des zones pré-définies Z_i telles que la restriction de l'approximation de (P) à Z_i a une solution maximale non-vide.

6.6 Contraction par les bords

6.6.1 Introduction

Cette section, ainsi que la section suivante, abordent deux méthodes poursuivant le même but, mais en utilisant deux approches différentes.

Les techniques d'énumération que l'on a présentées et utilisées dans les chapitres 4 et 5 ont essentiellement pour mission la recherche de toutes les solutions d'un problème. Cet objectif porte en lui d'importantes limites dans deux cas :

1. Limitations dues au système de résolution.
   Comme on l'a vu, à une seule solution "mathématique" du problème initial peuvent correspondre plusieurs pseudo-solutions après énumération. Ces pseudo-solutions peuvent même être en nombre tel que les énumérer toutes devient impossible. On se retrouve alors dans une situation où l'on cherche en fait à circonscrire une zone de l'espace.

2. Limitations dues au problème lui-même.
   Il s'agit particulièrement des problèmes où l'ensemble des solutions est infini. De façon plus pragmatique, il s'agit des cas ou les solutions du problème ne sont pas isolées, mais forment une zone de l'espace. Les intervalles de $I\!R$ qui ne sont pas réduits à un point forment des exemples de telles zones. Par exemple, si l'on cherche les solutions du système de contraintes $\{X \in [1, 2]\}$ à l'aide des méthodes précédemment décrites, on va énumérer tous les intervalles atomiques compris dans cet intervalle. Ils sont au nombre de 16 777 217.

Les différences entre les deux méthodes que l'on va présenter sont de plusieurs ordres. La première semble plus adaptée aux cas des problèmes dont la solution est une zone de l'espace, mais sa sémantique est distante de celle de l'ensemble du système (cf. 6.6.3). A l'opposé, la méthode qui sera présentée dans la section suivante est probablement plus adaptée à la circonscription des zones contenant des pseudo-solutions, d'autant que la sémantique du résultat obtenu est très proche de celle d'une énumération classique.

On verra dans la section suivante des exemples comparant les deux méthodes.

Cette première méthode, que nous appellerons "technique de contraction par les bords", a été introduite par Olivier Lhomme [12]. Ce n'est pas à proprement parler une technique d'énumération, dans la mesure où il n'est pas question d'énumérer tous les éléments d'un domaine, de quelque manière que ce soit. Ses principales qualités sont sa facilité d'implantation, sa compréhension relativement naturelle, et les bons résultats que l'on peut obtenir sur certains problèmes.

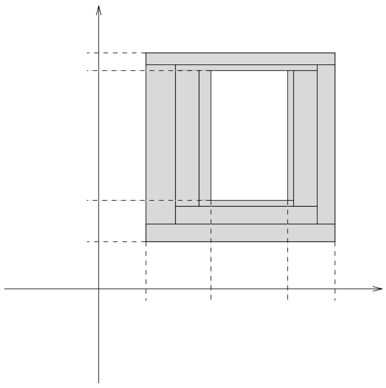
6.6.2 Principe et Implantation

Principe

Le principe de la méthode de contraction par les bords est très simple.

Pour toute variable X, de domaine $< A, B >$, d'un problème (P), on va tenter, par le biais du point fixe de trouver des sous-problèmes de (P) de la forme $(P) \cup \{X \in < A, A+\epsilon >\}$ ou $(P) \cup \{X \in < B-\epsilon, B >\}$ qui n'ont pas de solution. En d'autres termes, on va chercher à éliminer les parties du domaine des variables situées aux bords, en ne considérant qu'une seule dimension à la fois.

On peut donc, en dimension 2, schématiser cet algorithme de la manière suivante :



Comme on peut s'en rendre compte sur ce schéma, cet algorithme est nécessairement itératif, car la contraction du domaine d'une variable oblige à reconsidérer toutes les autres variables. L'algorithme s'arrête donc lorsque plus aucune réduction ne peut être faite.

A l'image des techniques d'énumération, on pourra affaiblir l'algorithme en considérant que, si plus aucune réduction de taille ϵ n'est possible, on n'essaie aucune réduction de taille inférieure à ϵ, et l'algorithme s'arrête.

Enfin, pour améliorer l'efficacité pratique et la complexité théorique (cf. 6.6.3), on va tenter des éliminations successives de dimensions de plus en plus petites suivant le principe suivant : au premier pas de l'algorithme, on cherche à éliminer des zones en bordure dont la taille correspond à la moitié de la taille du domaine le plus grand, puis on divise cette taille par deux à chaque pas, jusqu'à ce que celui-ci soit inférieur à la précision demandée. On pourra noter que la précision demandée n'a aucun lien *a priori* avec la précision obtenue (cf. exemples au paragraphe 6.7.3).

Il est à noter que cette technique peut être totalement incapable de résoudre certains problèmes, ce qui n'est jamais le cas des vraies techniques d'énumération.

En particulier, cette méthode sera inopérante sur tous les problèmes 1-insensibles pour le système de résolution avec les intervalles.

Un système portant sur des variables $X_1, ..., X_n$ est dit p-insensible si pour tout $X_{i_1}, ..., X_{i_p}$, et pour tout $V_{i_1}, ..., V_{i_p}$ appartenant respectivement aux domaines de $X_{i_1}, ..., X_{i_p}$, l'ajout des contraintes $\{X_{i_1} = V_{i_1}\}$, ..., $\{X_{i_p} = V_{i_p}\}$ ne permet pas de réduire les domaines des autres variables.

Bien entendu, cette condition est suffisante pour rendre l'algorithme de contraction par les bords inopérant, mais pas nécessaire.

Implantation

Voici quelques commentaires sur le programme implantant cette technique d'affinage des domaines (le programme se trouve à l'annexe 9.4.8, page 229) :

- On amorce le programme en choisissant une valeur de tentative de contraction par les bords : la moitié de la taille du domaine le plus grand.

- Le programme se compose de trois boucles imbriquées. De la plus centrale à la plus extérieure :

  1. Une boucle parcourant la liste des variables en essayant, pour chacune, de contracter son domaine par ses deux bords de la valeur courante de tentative de contraction.
  2. Une boucle répétant la boucle précédente tant que celle-ci a permis de contracter au moins un domaine.
  3. Une boucle répétant la boucle précédente en divisant la valeur courante de tentative de contraction par les bords par deux, jusqu'à ce que celle-ci soit inférieure à la précision demandée en paramètre du programme.

Enumération et contraction

Il est clair (cf. exemples au paragraphe 6.7.3) que la contraction obtenue grâce à cet algorithme ne fournit pas un résultat de nature comparable à celui d'une énumération traditionnelle. En particulier, il est inopérant sur les systèmes possédant plusieurs solutions séparées, ou de manière plus adaptée à cette méthode, plusieurs zones séparées contenant des solutions. Il peut donc s'avérer intéressant, voire nécessaire, de l'inclure dans une énumération.

6.6.3 Sémantique

On va reprendre dans cette section la sémantique de la contraction par les bords telle que décrite par O. Lhomme [12]. Essentiellement, on va décrire la $3 - B(w) - consistence$, qui est directement implantée par l'algorithme que l'on a donné dans le paragraphe précédent.

Cette description sémantique a deux intérêts. Le premier, évident, est de mieux comprendre le sens du résultat de la contraction par les bords. Le deuxième est de montrer que ce résultat n'est pas en corrélation directe avec la sémantique du système, telle qu'on l'a décrite dans les chapitres 2 et 4.

Définition 6.4 ($3 - B - consistence$) *Soit* (P) *un problème sur* $\mathbb{R}$.
(P) *est* $3 - B - consistent$ *si, pour toute variable* X *de* (P), *de domaine* $< a, b >$, *les solutions maximales de* $appx_{\Pi}(P \cup \{X = a\})$ *et* $appx_{\Pi}(P \cup \{X = b\})$ *sont non vides.*

En termes clairs, un problème est $3 - B - consistent$ si, en affectant chacune de ses variables à l'une des deux bornes de son domaines, aucune inconsistance n'apparaît dans le point fixe.

Comme pour les autres algorithmes, il existe une version dégénérée de cet algorithme, faisant intervenir un niveau d'imprécision toléré. La principale différence avec la version pure tient alors dans le fait que la solution trouvée n'est pas unique, mais dépend de l'implantation de l'algorithme. Ce phénomène est similaire à celui obtenu avec les énumérations.

Notation :

Soit f un rationnel IEEE 754. Soit w un entier positif.
On note f^{+w} (resp. f^{-w}) le w^{ieme} nombre suivant (resp. précédent) f. Si ce nombre n'existe pas, c'est $+\infty$ (resp. $-\infty$). La non densité des rationnels IEEE 754 dans les réels donne un sens à cette définition.

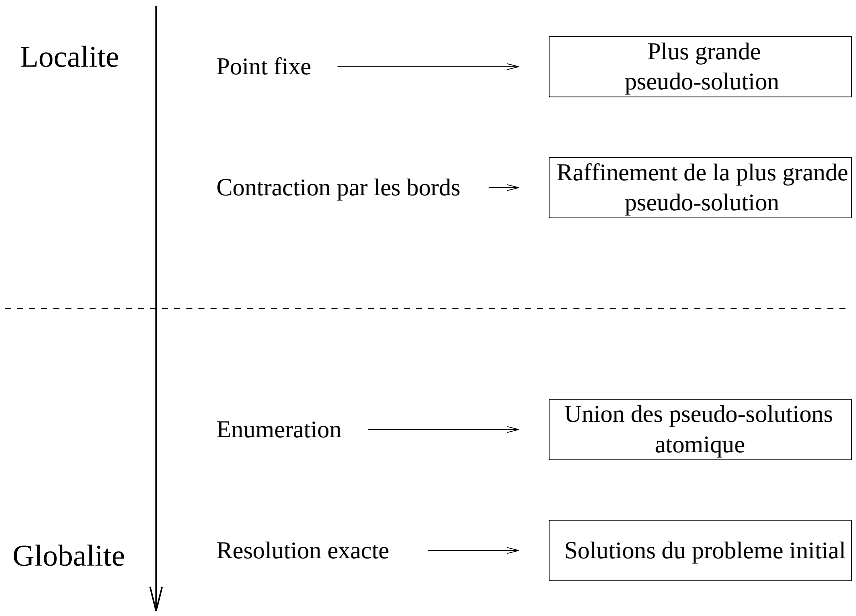
Définition 6.5 ($3 - B(w) - consistence$) *Soit* (P) *un problème sur* $\mathbb{R}$.
(P) *est* $3 - B(w) - consistent$ *si, pour toute variable* X *de* (P), *de domaine* $< a, b >$, *les solutions maximales de* $appx_{\Pi}(P \cup \{X \in [a, a^{+w}]\})$ *et* $appx_{\Pi}(P \cup \{X = [b^{-w}, b]\})$ *sont non vides.*

Propriété 6.6 *L'algorithme décrit au paragraphe précédent calcule une fermeture par* $3 - B(w) - consistence$.

Un point important est de noter que la contraction par les bords possède d'intéressantes bornes de complexité : soit m le nombre de contraintes, et n le nombre de variables, soit A la taille du plus grand domaine. Alors la complexité de la contraction par les bornes est encadrée par $nlogA$ et mn^2A^2.

La principale différence avec les techniques d'énumération décrites au chapitre 4 tient au fait que, avec la version pure ou avec la version dégénérée, il est impossible de lier les solutions du problème initial à l'encadrement produit. En fait, alors qu'une énumération a l'effet d'une globalisation du traitement effectué par le point fixe, la $3 - B - consistence$, appelée ici contraction par les bords, n'effectue qu'un traitement local un peu plus fort que la consistance locale pure du point fixe. Ceci explique les différences de comportement entre cet algorithme d'une part, et les algorithmes d'énumération et leurs dérivés d'autre part.

On peut visualiser ces différents niveaux de raisonnement sur le schéma suivant :



6.7 Enumération par les bords

6.7.1 Introduction

La technique que l'on va décrire dans cette section (également décrite dans [17]) répond au même cahier des charges que celle décrite dans la section précédente, mais avec quelques différences fonctionnelles et sémantiques :

- Elle est un peu plus délicate à implanter.
- Elle est en général moins rapide à l'exécution, mais son résultat est toujours de qualité supérieure ou égale.
- Sa sémantique est beaucoup plus proche de celle de la vraie énumération générale décrite au chapitre 4.

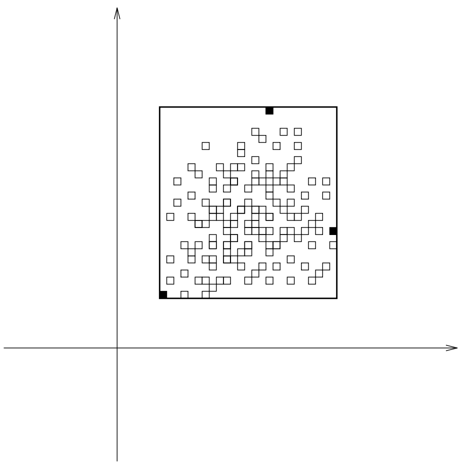
On appellera cette technique "énumération par les bords", par opposition à celle précédemment évoquée de "contraction par les bords", à cause de la technique sur laquelle elle repose et du sens du résultat qu'elle calcule, tous deux fortement liés à la notion d'énumération.

6.7.2 Principe et implantation

Principe

Le principe de base de la technique d'énumération par les bords est simple. L'idée de base est la suivante : les pseudo-solutions produites par une énumération forment un ensemble, qui peut être de cardinal élevé, et qui, en général, est relativement "informe". L'information la plus pertinente est en général, pour chaque variable du problème, l'intervalle qui contient toutes les pseudo-solutions.

Considérons par exemple un problème en dimension 2, dont les pseudo-solutions forment l'ensemble suivant :



Les petits carrés représentent les pseudo-solutions, le grand carré est le plus petit pavé les contenant. L'idée de l'algorithme d'énumération par les bords est que, sur cet exemple schématique, par exemple, les 3 petits carrés noirs, c'est à dire trois pseudo-solutions, suffisent à construire le grand carré, c'est à dire le plus petit pavé contenant toutes les pseudo-solutions.

L'algorithme est donc construit de la manière suivante : on va chercher des pseudo-solutions, mais en dehors du plus petit pavé contenant toutes les pseudo-solutions déjà trouvées. On a donc ainsi toutes les chances de trouver le plus petit pavé contenant toutes les pseudo-solutions sans pour autant les avoir énumérées toutes. Comme on le montrera au 6.7.4, le résultat obtenu est exactement le plus petit pavé contenant toutes les pseudo-solutions.

Comme pour les autres méthodes d'énumération, on peut relâcher la précision, en se concentrant sur des pseudo-solutions non atomiques. Une approche basée sur les zones définies (cf. section 6.5) permettra de retrouver une caractérisation plus rigoureuse de la solution obtenue.

Il existe une autre forme de relâchement de la précision. Plutôt que de chercher le plus petit pavé contenant toutes les pseudo-solutions du problème approché, on peut se contenter de chercher un pavé les contenant toutes, mais pas forcément le plus petit, en tolérant une marge d'encadrement.

Pour des raisons liées à la topologie des pseudo-solutions dans l'espace de recherche (plus on est proche de la solution "mathématique", plus il est difficile de montrer qu'un pavé d'une certaine taille ne contient pas de solution), on pourra ainsi gagner beaucoup de temps, en perdant peu de précision.

Implantation

Voici quelques commentaires sur le programme implantant cette technique d'énumération par les bords (le programme se trouve à l'annexe 9.4.9, page 232) :

- Structure du programme.
  Le programme implante fidèlement l'algorithme : il cherche tout d' abord une première solution au problème (prédicat `find_solution`), et, s'il l'a trouvée, il en cherche une autre, hors de la zone où il a trouvé la première (à un epsilon près, argument du prédicat d'énumération) (prédicat `find_solution_outbox`). S'il en trouve à nouveau une, il calcule le nouveau pavé contenant toutes les solutions trouvées jusqu'alors (prédicat `compute_new_box`), et poursuit jusqu'à ce qu'aucune nouvelle

solution ne puisse être trouvée (prédicat `check_out_and_complete`).

- La recherche "gauche-droite".
  Il est intuitivement logique de chercher successivement des solutions les plus éloignées de manière à construire le pavé contenant toutes les pseudo-solutions avec un minimum de pseudo-solutions effectivement calculées. Le programme donné en annexe implante une version naïve de ce concept en alternant recherche à gauche d'abord, et recherche à droite d'abord dans la recherche des pseudo-solutions successives. C'est le rôle de la variable `sens` dans le programme.

- La manipulation des pavés de $I\!R^n$.
  Ce programme manipule des pavés de $I\!R^n$. Cette manipulation est faite explicitement par des listes de paires de nombres rationnels, appelées dans le programme `SymbolicSol`, par exemple.

6.7.3 Résultats expérimentaux

On a regroupé dans ce paragraphe les exemples d'application des techniques de contraction par les bords et d'énumération par les bords pour deux raisons :

1. Tout d'abord, ces techniques étant de même nature opérationnelle, bien que différentes dans leur manière d'aborder le problème comme dans leur sémantique, elles s'attaquent aux mêmes exemples.

2. Ensuite, cela permettra de les comparer directement.

Traitement de Broyden-Banded

Voici le rapport des temps entre ces deux méthodes que l'on obtient sur le problème de Broyden-Banded (problème décrit dans le paragraphe 6.1.2.

Quelques points à noter à propos de ces résultats :

- Les résultats de chacune des deux méthodes sont détaillés dans l'annexe 9.2, page 189.

- Ils doivent être également comparés aux résultats obtenus en utilisant la technique d'évaluation globale des expressions complexes (cf. 6.2), ainsi qu'à la version la plus naïve. Il semble que, pour les tailles du problème que l'on est capable de traiter (plus pour des raisons de mémoire que de vitesse d'exécution), les méthodes de contraction par les bords et de traitement global des expressions complexes (qui semblent linéaires) donnent des résultats similaires, alors que la méthode d'énumération par les bords est légèrement en retrait.

- Pour obtenir des résultats d'une précision constante en fonction de n, et de même nature que pour les autres méthodes, il convient de faire croître la précision demandée pour la méthode de contraction par les bords. En effet (cf. 6.6), la précision demandée, qui commande la fin de cet algorithme itératif n'est pas liée à la précision obtenue. On verra plus précisément un exemple de tel comportement un peu plus loin.

- On pourra noter que dans le cas des deux algorithmes dont nous traitons ici, aucune énumération supplémentaire n'est nécessaire pour résoudre correctement le problème. C'est naturel pour la méthode d'énumération par les bords (puisqu'il n'y a qu'une solution), mais ce n'était pas acquis pour la méthode de contraction par les bords.

- Dans le tableau suivant, le rapport exprime la vitesse relative des deux méthodes : un rapport T_n^e/T_n^c supérieur à 1 signifie que la méthode de contraction par les bords est plus rapide. Ce tableau fait également apparaître la croissance des temps d'exécution en fonction de n, pour la méthode de contraction (T_n^c/T_{n-1}^c) et pour la méthode d'énumération (T_n^e/T_{n-1}^e).

n	T_n^c/T_{n-1}^c	T_n^e/T_{n-1}^e	T_n^e/T_n^c
5	-	-	1.3
10	2.6	2.1	1.1
20	1.3	1.7	1.4
40	1.3	1.8	2.0
80	1.5	2.0	2.6
160	1.6	2.1	3.5

Les limites de la contraction par les bords

Comme on l'a dit dans les paragraphes et sections précédents, la méthode de contraction par les bords ne fournit aucune garantie quant à la qualité (en terme de précision) de la solution obtenue. En voici un exemple sur un petit système de contraintes non linéaires. Les deux requêtes qui suivent le programme Prolog IV de pose des contraintes exécutent successivement la contraction par les bords (qui ne donne aucun renseignement), et l'énumération par les bords qui fournit (comme elle le doit) un résultat à la précision demandée).

```
constraints(X,Y,Z,T):-
   Z = square(X) .+. 2 .*. X .+. 1,
   T = square(Y) .+. 2 .*. Y,
   square(X) .+. square(Y) = Z .*. T,
   square(X) .*. square(Y) = Z .+. 3 .*. T,
   X ge T.-. 10,
   Y ge Z.-. 10,
   X le .-. Y.


>> constraints(X,Y,Z,T),narrow_box([X,Y,Z,T],1000).

T ~ real,
Z ~ real,
Y ~ real,
X ~ real.

>> constraints(X,Y,Z,T),englobing_box([X,Y,Z,T],1000).

T ~ cc('>1.7705974','>1.7709496'),
Z ~ cc('>4.814734','>4.815387'),
Y ~ cc(-'>2.6646173',-'<2.664511'),
X ~ cc('>1.1942503','<1.194399').
```

6.7.4 Sémantique

On va s'attacher dans ce paragraphe à démontrer que la technique que l'on a introduite dans cette section produit un résultat fortement lié au résultat d'une énumération complète et atomique.

Pré-requis : il est conseillé pour aborder ce paragraphe d'avoir lu les sections décrivant la sémantique du système de résolution (cf. 2.4), et la sémantique de l'énumération (cf. 4.1.3).

Définition de l'algorithme

L'algorithme dit "d'énumération par les bords" que l'on vient d'introduire construit une famille $(A_i)_{i\in I}$ de solutions atomiques, et une partition $(B_j)_{j\in J}$ de $I\!R^n \setminus appx_{I\!I}(\bigcup_{i\in I} A_i)$ telle que :
pour tout $j \in J$, $appx_{I\!I}(P)_{|B_j}$ a une solution maximale égale à l'ensemble vide.

Ces deux familles se construisent naturellement au cours de l'algorithme : chaque étape ajoute une solution atomique à la famille A, et la dernière étape construit la famille B.

Propriété de caractérisation

Soit $(S_i)_{i\in I_s}$ l'ensemble des solutions atomiques de $appx_{I\!I}(P)$ (et donc l'ensemble des solutions d'une énumération complète atomique (cf. propriété 4.3).

Propriété 6.7

$$appx_{I\!I}(\bigcup_{i\in I_s} S_i) = appx_{I\!I}(\bigcup_{i\in I} A_i)$$

La preuve de cette propriété se trouve à l'annexe 9.3.12, page 204.

Là encore, on peut en déduire que cette technique est très bien intégrée à l'ensemble du système de résolution tel qu'il est décrit dans ce document, puisque le résultat produit est très fortement lié à celui, plus classique, mais plus délicat à atteindre, et moins aisément manipulable, des énumérations atomiques complètes.

6.8 En résumé

Quelques méthodes ...

On a donc présenté dans ce chapitre un certain nombre de méthodes permettant de résoudre de manière satisfaisante trois problèmes délicats introduits dans la première section. Le champ d'application de ces méthodes ne s'arrête bien sûr pas à ces trois exemples, mais ils permettent de comprendre quelles difficultés on peut rencontrer, et comment on peut tenter de les résoudre.

On a abordé dans ce chapitre que des problèmes liés à des systèmes de contraintes essentiellement continus, et fortement non-linéaires. Il pourrait être complété par tout un ensemble de techniques connues concernant les domaines finis, par exemples; et il y aurait encore beaucoup à dire quant aux mélanges de contraintes de natures différentes.

En fait, on a essayé de mettre en évidence les aspects essentiels de la programmation par contraintes avec les intervalles :

- De tels systèmes proposent **un cadre unifié** pour exprimer des problèmes divers avec **une très grande puissance d'expression**.

- Leur **utilisation est très naturelle**, et permet une **formalisation très proche des problèmes initiaux**.

- La résolution de problèmes délicats laisse apparaître **un manque de méthodologie** : il est encore difficile de proposer à un non spécialiste de résoudre de tels problèmes.

  Les techniques et outils dont on dispose forment encore un tas mal organisé, dans lequel il est difficile *a priori* de faire le bon choix.

Compilation de résultats

De manière plus pragmatique, on a regroupé dans le tableau suivant les diverses techniques décrites dans ce chapitre, en indiquant l'intérêt de leur utilisation pour nos trois problèmes témoins.

	Pentagone	Broyden-Banded	Kinematics
Traitement naturel (cf. 6.1)	[Penta] Mauvais	[Broyden] Mauvais	[Kine] Très mauvais
Avec traitement global (cf. 6.2)	???	[BroydenG] **Très bon**	???
Avec traitement par Fourier (cf. 6.3.2)	[PentaF] [2] **Très Bon**	???	[KineF] **Très bon**
Avec traitement linéaire (cf. 6.3.3)	???	???	[KineL] Assez Bon
Avec une bonne énumération (cf. 6.4)	[Penta] **Bon**	???	[KineS] **Bon**
Avec contraction par les bords (cf. 6.6)	[1] Inopérant	[BroydenN] **Très bon**	[1] Inopérant
Avec énumération par les bords (cf. 6.7)	???	[BroydenE] **Bon**	???

Quelques remarques propos de ce tableau :

- Explication des notes apparaissant dans cette table :
  1 : L'utilisation des techniques de contraction par les bords et d'énumération par les bords peut nécessiter le recours à une énumération externe (cf. sections 6.6 et 6.7). Dans ce cas, cela est signalé par cette note.
  2 : Couplé avec une bonne stratégie d'énumération.
- A l'image de Kinematics, l'utilisation des méthodes de Fourier, et de collaboration linéaire nécessite quelques manipulations formelles préalables simples du type décomposition des carrés, ou factorisations.

- Temps d'exécution des programmes en annexe (cf. 9.2). La dénomination du programme est indiquée dans le tableau précédent.

Deux points crutiaux

Comme on l'a vu tout au long de ce chapitre, et du chapitre précédent, la résolution d'un problème se divise en deux parties :

1. Le système de contraintes, et son solveur, ou ensemble de solveurs. Son efficacité peut se mesurer en puissance de déduction, ou puissance de contraction des intervalles. Cette information correspond à la capacité à déduire des réductions d'intervalles dans un système de contraintes, en fonction des domaines des variables. On a vu plusieurs méthodes pour améliorer cette compétence (traitement global, méthodes basées sur Fourier, collaboration avec le solveur linéaire).

   Un bon indicateur de la puissance de déduction du couple système de contraintes / système de résolution est le nombre de pseudo-solutions produites par une énumération. Dans l'exemple de Kinematics, par exemple, alors que la version naïve produit 5 073 pseudo-solutions, la version avec les simplifications formelles n'en produit plus que 168, et la version avec contraintes redondantes (Fourier), plus que 40 (ces trois résultats étant bien sûr obtenus avec la même technique d'énumération, et la même précision).

2. La technique d'énumération elle-même, qui peut parfois, à elle seule, pallier les insuffisances du système de contraintes, ou au contraire, pénaliser la résolution.

   Par exemple, toutes choses égales par ailleurs, et toujours sur le problème Kinematics, la méthode avec les contraintes redondantes (Fourier) associée à une énumération dynamique par le plus petit domaine s'avère 15 fois plus lente que la méthode simplifiée (pourtant moins puissante), si celle-ci est associée à une énumération en ordre statique (dans l'ordre initial des variables).

Il ressort de cette étude qu'il est raisonnable de penser résoudre des problèmes intéressants basés -ou contenant- des contraintes non-linéaires, en associant le bon couple système de contraintes / solveur(s) à la bonne technique d'énumération.

7 Conclusion

7.1 Un aperçu

Comme vous avez pu le constater, ce document est assez volumineux. Il faut y voir deux causes :

1. Il est "touche-à-tout".

   Il aborde en effet aussi bien les aspects théoriques que les aspects pratiques; il traite aussi bien de ce qui a été fait que de ce qui aurait pu l'être, ou de ce qui le sera peut-être.

2. Il est énumératif.

   Comme on le dit souvent, les grandes idées ne prennent que peu de place. Faut-il en déduire que les intervalles ne sont pas une grande idée ?
   Non.
   Pour autant, l'aspect "énumération" de ce document témoigne du fait que l'ensemble des techniques relatives aux intervalles n'a pas encore effectué sa convergence, ne forme pas encore une technologie achevée, unifiée, et donc simple.

   C'est probablement cet objectif qui doit conduire nos travaux et recherches à venir.

7.2 Le futur

<u>De ce document en particulier, de la P(L)C en général</u>

Il est clair que l'avenir de ce document est étroitement lié à l'avenir de la P(L)C, et plus particulièrement de Prolog IV.

Il doit donc continuer à refléter une somme d'expériences, mais également converger de plus en plus nettement vers la définition d'une méthodologie plus uniforme.

Pourquoi écrire **P(L)C** ?

La Programmation Logique par Contraintes (PLC), que représente Prolog IV a aujourd'hui pris une orientation qui, comme le montre ce document, la classe de manière plus évidente parmi la Programmation par Contraintes (PC), que parmi la Programmation Logique (PL).

Il nous a en effet très nettement semblé que Prolog représente un terrain très favorable à l'expression et à la manipulation des contraintes. Pour autant, peut-on vraiment dire que l'on fait encore du Prolog, si l'on n'en utilise plus que quelques aspects, plus pratiques que fondamentaux (backtracking gratuit, manipulations formelles aisées, facilité de programmation, ...) ?

C'est l'une des questions auxquelles il nous faudra répondre pour prédire l'avenir de Prolog IV et de la PLC.

De Prolog IV en particulier

De manière plus pragmatique, il est clair que Prolog IV devra recevoir dans un avenir proche quelques compléments lui permettant de s'attaquer plus efficacement à toutes sortes de problèmes de nature "industrielle" :

1. **Relations globales** permettant de traiter avec succès certains problèmes spécifiques à grande échelle (planification) ou de nature plus génériques (la contrainte "tous différents", par exemple).

2. **Relation globale linéaire**, permettant de gérer plus efficacement les problèmes présentant une réelle mixité linéaire / non-linéaire.

3. La possibilité de programmer complètement **ses propres relations globales** pour les ajouter aux relations de base.

4. Des techniques de **gestion globale systématique des expressions continues non-linéaires** dérivables (par le biais de méthodes de Newton, par exemple) permettant de gérer plus efficacement les équations de modélisation physique faisant souvent intervenir des polynômes ou autres exponentielles.

5. D'autres **méthodes d'énumération**, générales, ou au contraire plus spécifiques.

8 Références

[1] N. Beldiceanu, E. Contejean. *Including Global Constraints in CHIP.* Mathl. Comput. Modelling Vol 20, No 12, pp. 97-123, 1994.

[2] F. Benhamou, D. McAllester, P. Van Hentenryck. *CLP(Intervals) revisited.* ILPS, 1994.

[3] F. Benhamou, Touraïvane. *Prolog IV : langage et algorithmes.* JFPL, 1995.

[4] J.G. Cleary. Logical arithmetic. *Future generation computing systems.* 1987.

[5] A. Colmerauer. *An introduction to Prolog III.* CACM, 28(4),1990.

[6] A. Colmerauer. *Résolution approchée de contraintes par des produits cartésiens d'ensembles privilégiés.* Technical documentation. PrologIA. 1994.

[7] A. J. W. Duijvestijn. *Simple Perfect squared square of lowest order.* Journal of combinatorial theory, Series B25, p 240. 1978.

[8] E.R. Hansen, R.I. Greenberg. *An interval Newton method.* Appl. Math. Comput, 1983.

[9] E.R. Hansen, S. Sengupta. *Bounding solutions of systems of equations using interval analysis.* BIT, 1981.

[10] H. Hong, V. Stahl. *Safe starting regions by fixed points and tightening.* 1993.

[11] A. Hyvönen. *Constraint reasoning based on interval arithmetic: the tolerance propagation approach.* Artificial intelligence 58, p 71 -112, 1992.

[12] O. Lhomme. *Consistency techniques for numeric CSP's.* IJCAI, 1993.

[13] A.K. Mackworth. *Consistency in Networks of Relations.* Artificial Intelligence, Vol 8, No 1, 1977.

[14] K. Meintjes, A. P. Morgan. *Chimical equilibrium systems as numerical test problems.* ACM Transactions on mathematical software, 1990, No. 2, pp 143-151.

[15] R.E. Moore. *Interval analysis.* Prentice-Hall, Englewood Cliffs, 1966.

[16] A. Morgan. *Solving polynomial systems using continuation for engineering and scientific problems.* Prentice-Hall, Inc., 1987.

[17] S. N'Dong, M. van Caneghem. *Problèmes continus en PLC avec les intervalles.* JFPL, 1995.

[18] S. N'Dong, M. van Caneghem. *Global behaviour for complex constraints.* CP96, LNCS 1118, 1996.

[19] S. N'Dong. *Traitement global des expressions complexes en CLP(Intervalles).* RJC-IA, Proceedings, 1996.

[20] S. N'Dong. *Les intervalles de Prolog IV : Méthodes et exemples.* Technical documentation. PrologIA. 1996.

[21] W. Older, A.J. Velino. *Constraint arithmetic on real interval in CLP.* Selected research. F. Benhamou and A. Colmerauer, eds MIT Press, 1993.

[22] W. Older, F. Benhamou. *Programming in CLP(BNR).* PPCP, 1993.

[23] Pascal Van Hentenryck. *Scheduling and Packing in the Constraint Language cc(FD).*

[24] Pascal Van Hentenryck. *Constraint Satisfaction on Logic Programming.* MIT Press, Cambridge, Mass. USA. 1989.

[25] A. Wierthoff. *C-XSC : A C++ Class Library for Extended Scientific Computing.* Institute for Applied Mathematics, 1993.

[26] American National Standard Institute. *IEEE Standard for Binary Floating-Point Arithmetic.* 1985

9 Annexes

9.1 La norme IEEE 754

Partie non-encore développée
[*Description de la norme IEEE*] A développer

9.2 Temps d'éxécution

On a regroupé dans cette section tous les temps d'exécution des programmes presentés. Il s'agit de temps obtenus sur une station SUN Sparc Classic, sans compilation optimisée. Si des options particulières sont utilisées, elles sont précisées. D'autres informations relatives à l'exécution peuvent également être précisèes dans la rubrique "Remarques".

Nom du programme	Temps (s)	Remarques
[broyden(5)]	15.3	
[broyden(10)]	71.0	
[broydenSE(5)]	7.9	
[broydenSE(10)]	40.3	
[broydenSE(20)]	597.6	
[broydenSE(40)]	> 110 326	
[broydenG(5)]	8.7	
[broydenG(10)]	16.8	
[broydenG(20)]	35.4	
[broydenG(40)]	76.2	
[broydenG(80)]	159.5	
[broydenG(160)]	316.9	
[broydenG(320)]	647.7	
[broydenG(640)]	1 325.9	
[broydenE(5)]	42.3	
[broydenE(10)]	87.9	
[broydenE(20)]	150.6	
[broydenE(40)]	273.2	
[broydenE(80)]	536.4	
[broydenE(160)]	1150.3	

Nom du programme	Temps (s)	Remarques
[broydenN(5)]	31.5	prec = 10 000
[broydenN(10)]	82.6	prec = 6 000
[broydenN(20)]	110.7	prec = 3 000
[broydenN(40)]	139.7	prec = 3 000
[broydenN(80)]	205.5	prec = 3 000
[broydenN(160)]	333.4	prec = 3 000
[broydenN(320)]	619.6	prec = 3 000
[chemical]	7.2	
[donald]	31.0	
[donaldM]	4.3	
[god]	n.m.	
[kine]	14 021	
[kineS]	6 322	
[kineS]	393.4	avec greatest_domain
[kineF]	1 488	
[kineF]	163.3	avec greatest_domain
[kineL]	1 776	avec greatest_domain
[magic(10)]	2.8	
[magic(20)]	32.2	
[magic(30)]	144.2	
[magic(40)]	426.7	
[magic(50)]	1 015.6	
[magic(60)]	2 026.8	
[magic(70)]	3 753.9	
[magic(80)]	6 026.3	
[magic(90)]	9 611.3	
[magic(100)]	14 985.6	

Nom du programme	Temps (s)	Remarques
[magicR(10)]	1.2	
[magicR(20)]	5.1	
[magicR(30)]	12.3	
[magicR(40)]	22.8	
[magicR(50)]	40.5	
[magicR(60)]	61.8	
[magicR(70)]	89.8	
[magicR(80)]	100.7	
[magicR(90)]	133.4	
[magicR(100)]	171.8	
[miniPoly]	16.0	
[miniPolyS]	0.9	
[penta]	528.0	enum : list_order
[penta]	240.0	enum : greatest_domain
[pentaF]	129.0	enum : greatest_domain
[polyExp]	1.6	
[polyPower]	0.8	
[polyTimes]	27.5	
[premiers]	n.m.	Unions d'intervalles
[puzzle]	48.6	Intervalles simples
[puzzle]	4.1	Unions d'intervalles
[queens(4)]	0.6	
[queensOne(5)]	0.1	Intervalles simples
[queensOne(10)]	1.0	Intervalles simples
[queensOne(15)]	9.8	Intervalles simples
[queensOne(20)]	84.6	Intervalles simples
[queensOne(25)]	628.0	Intervalles simples

[queensOne(5)]	0.1	Unions d'intervalles
[queensOne(10)]	0.8	Unions d'intervalles
[queensOne(15)]	1.4	Unions d'intervalles
[queensOne(20)]	4.3	Unions d'intervalles
[queensOne(25)]	9.2	Unions d'intervalles
[queensOne(30)]	9.3	Unions d'intervalles
[queensOne(40)]	16.7	Unions d'intervalles
[queensOne(50)]	92.4	Unions d'intervalles
[robot]	0.7	
[squares]	7 406.0	
[vrais(8,4)]	21.7	
[vraisM(8,4)]	0.7	

9.3 Démonstrations

9.3.1 Propriétés des approximations

Preuve de la propriété 2.4

1. (a) Montrons que $appx_\alpha(A) \subset \bigcap_{B\in\alpha, B\supset A} B$.

   $appx_\alpha(A) = Min\{B \in \alpha \mid B \supset A\}$
   donc $\forall B \in \alpha,\ B \supset A$, on a $appx_\alpha(A) \subset B$,
   donc $appx_\alpha(A) \subset \bigcap_{B\in\alpha, B\supset A} B$.

   (b) Montrons que $\bigcap_{B\in\alpha, B\supset A} B \subset appx_\alpha(A)$.

   $\bigcap_{B\in\alpha, B\supset A} B \subset B,\ \forall B \in \alpha,\ B \supset A$,
   donc $\bigcap_{B\in\alpha, B\supset A} B \subset Min\{B \in \alpha,\ B \supset A\}$,

donc $\bigcap_{B \in \alpha, B \supset A} B \subset appx_\alpha(A)$.

2. Découle immédiatement de la définition.

3. (a) Montrons que $A \in \alpha$ implique que $appx_\alpha(A) = A$.
   $appx_\alpha(A) = Min\{B \in \alpha \mid B \supset A\}$
   or A lui-même appartient à $\{B \in \alpha \mid B \supset A\}$, et A est forcément le plus petit élément de cet ensemble.
   donc $appx_\alpha(A) \subset A$,
   or $appx_\alpha(A) \supset A$ (d'après le point 2),
   donc $appx_\alpha(A) = A$.

   (b) Montrons que $appx_\alpha(A) = A$ implique que $A \in \alpha$.
   C'est immédiat, puisque, par définition, l'approximation d'un sous-ensemble de E, si elle existe, appartient à α, et que A est le plus grand élément inclus dans A.

4. Supposons que $appx_\alpha(A) = \{s\}$.
   D'après le point 2 de la propriété, on a : $A \subset \{s\}$.
   Donc :

   - Soit A est l'ensemble vide, mais alors $appx_\alpha(A) = \emptyset$, puisque $\emptyset \in \alpha$ (point 3 de la propriété). C'est impossible.
   - Soit $A = \{s\}$.

   On a donc bien $A = \{s\}$.

5. (a) Montrons que $appx_\alpha(A_1 \cup A_2) \subset appx_\alpha(B_1 \cup B_2)$.
   $B_1 = appx_\alpha(A_1)$, donc $A_1 \subset B_1$ (point 2 de la propriété).
   De même, on a $A_2 \subset B_2$.
   Donc, on a $A_1 \cup A_2 \subset B_1 \cup B_2$,
   et donc $appx_\alpha(A_1 \cup A_2) \subset appx_\alpha(B_1 \cup B_2)$ (puisqu'il est clair que $appx_\alpha$ est une fonction croissante).

   (b) Montrons que $appx_\alpha(B_1 \cup B_2) \subset appx_\alpha(A_1 \cup A_2)$.
   Soit $x \in appx_\alpha(B_1 \cup B_2)$.
   On a $x \in \bigcap_{C \in \alpha, C \supset B_1 \cup B_2} C$,
   Or $\{C \in \alpha \mid C \supset B_1 \cup B_2\}$

est égal à $\{C \in \alpha \mid C \supset A_1 \; et \; C \supset A_2\}$
(car B_1 est justement (par définition) le plus petit élément de α contenant A_1, et B_2 le plus petit élément de α contenant A_2).
Donc $x \in \bigcap_{C \in \alpha, C \supset A_1 \cup A_2} C$, donc $x \in appx_\alpha(A_1 \cup A_2)$.
Donc $appx_\alpha(B_1 \cup B_2) \subset appx_\alpha(A_1 \cup A_2)$.

Donc $appx_\alpha(A_1 \cup A_2) = appx_\alpha(B_1 \cup B_2)$.

Preuve de la propriété 2.5

Soit α une approximation finie sur E.
Soit A un sous-ensemble quelconque de E.
Considérons l'ensemble : $\bigcap_{B \in \alpha, B \supset A} B$.
Cette intersection existe puisqu'elle est finie. De plus, elle appartient à α puisqu'elle est l'intersection d'éléments de α, et que α est fermé par intersection.
Donc $\bigcap_{B \in \alpha, B \supset A} B = Min\{B \in \alpha \mid B \supset A\} = appx_\alpha(A)$ existe.

9.3.2 Propriétés des intervalles

Preuve de la propriété 2.7

Soit C un sous-ensemble convexe de $I\!R$.
C est convexe, donc, d'après la définition, on a :
$\forall a, b$ dans C, l'intervalle fermé $[a, b]$ est inclus dans C.

Distinguons alors 4 cas :

1. Soit C est vide, alors $C = \emptyset$ **(cas 10)**.

2. Soit C n'a pas de borne inférieure.
   Distinguons alors 3 cas :

   (a) Soit C n'a pas de borne supérieure,
   mais alors $C = I\!R = (-\infty, +\infty)$ **(cas 1)**.

   (b) Soit C a une borne supérieure atteinte : a.
   Montrons alors que $C = (-\infty, a]$.
   Il est évident que $C \subset (-\infty, a]$; et il reste donc à montrer que $(-\infty, a] \subset C$.
   Soit $x \in (-\infty, a]$ (donc $x \leq a$).
   Alors il existe $y \in C,\ y < x$ (puisque C est non-borné inférieurement).
   Donc $x \in [y, a]$.
   Donc $x \in C$ (puisque C est convexe).
   Donc $C = (-\infty, a]$ **(cas 5)**.

   (c) Soit C a une borne inférieure non atteinte : a.
   Montrons alors que $C = (-\infty, a)$.
   Il est clair que $C \subset (-\infty, a)$; et il reste donc à montrer que $(-\infty, a) \subset C$.
   Soit $x \in (-\infty, a)$.
   Alors il existe $y \in C$ tel que $y < x$ (puisque C est non-borné inférieurement),
   et il existe $z \in C$ tel que $z > x$ (puisque $x < a$ et que a est la borne supérieure de C).
   Donc $x \in [y, z]$.
   Donc $x \in C$.
   Donc $C = (-\infty, a)$ **(cas 4)**.

3. Soit C a une borne inférieure atteinte : a.
   Avec des raisonnements similaires, on fait apparaître les cas :

   (a) Si C n'a pas de borne supérieure : $C = [a, +\infty)$ **(cas 3)**.

   (b) Si C a une borne inférieure atteinte b : $C = [a, b]$ **(cas 6)**.

(c) Si C a une borne inférieure non-atteinte b : $C = [a, b)$ **(cas 9)**.

4. Soit C a une borne inférieure non-atteinte : a.
   Avec des raisonnements similaires, on fait apparaître les cas :

   (a) Si C n'a pas de borne supérieure : $C = (a, +\infty)$ **(cas 2)**.

   (b) Si C a une borne inférieure atteinte b : $C = (a, b]$ **(cas 7)**.

   (c) Si C a une borne inférieure non-atteinte b : $C = (a, b)$ **(cas 8)**.

9.3.3 Les intervalles atomiques

Preuve de la propriété 2.9

Bien que ce résultat soit tout à fait intuitif, en voici la démonstration.

Reprenons les dix formes possibles d'intervalles flottants (cf. 2.7), et pour chacun des cas, recherchons les éléments atomiques d'approximation.

1. $I\!R$
   $(-\infty, 0] \subset I\!R$, donc $I\!R$ n'est pas atomique.

2. $(f, +\infty)$
   Distinguons deux cas :

   (a) f n'est pas le plus grand rationnel IEEE 754.
   Soit alors f' un rationnel IEEE 754supérieur à f.
   $(f', +\infty) \subset (f, +\infty)$, donc $(f, +\infty)$ n'est pas atomique.

   (b) f est le plus grand rationnel IEEE 754.
   Alors $(f, +\infty)$ est atomique (**cas 4**).

3. $[f, +\infty)$
   $(f, +\infty) \subset [f, +\infty)$, donc $[f, +\infty)$ n'est pas atomique.

4. $(-\infty, f)$
   Comme pour le point 2, $(-\infty, f)$ est atomique si et seulement si f est le plus petit rationnel IEEE 754(**cas 3**).

5. $(-\infty, f]$
   $(-\infty, f) \subset (-\infty, f]$, donc $(-\infty, f]$ n'est pas atomique.

6. $[f_1, f_2]$
   Distinguons deux cas :

   (a) $f_1 = f_2$, alors $[f_1, f_2] = \{f_1\}$ est atomique (**cas 1**).

   (b) $f_1 \neq f_2$
   $(f_1, f_2) \subset [f_1, f_2]$, donc $[f_1, f_2]$ n'est pas atomique.

7. $(f_1, f_2]$
   $(f_1, f_2) \subset (f_1, f_2]$, donc $(f_1, f_2]$ n'est pas atomique.

8. (f_1, f_2)
   Distinguons deux cas :

   (a) f_1 et f_2 sont deux rationnels IEEE 754consécutifs.
   Alors (f_1, f_2) est atomique (**cas 2**).

   (b) f_1 et f_2 ne sont pas deux rationnels IEEE 754consécutifs.
   Soit alors f' un rationnel IEEE 754tel que $f_1 < f' < f_2$.
   $(f', f_2) \subset (f_1, f_2)$, donc (f_1, f_2) n'est pas atomique.

9. $[f_1, f_2)$
   $(f_1, f_2) \subset [f_1, f_2)$, donc $[f_1, f_2)$ n'est pas atomique.

10. $\emptyset$
    Ne peut être atomique par définition.

9.3.4 Justification de l'algorithme du point fixe

Preuve de la propriété 2.14

Soit r une relation n-aire sur $I\!R$, $S \in I\!I^n$.
Considérons l'ensemble $E = \{A \in appx_{I\!I}(r),\ A \subset S\}$.

Montrons alors que E possède un maximum (au sens de l'inclusion).

$I\!I$ est fini, donc $I\!I^n$ est fini, donc $appx_{I\!I}(r)$, qui est un sous-ensemble de $I\!I^n$ est fini.

Considérons alors $B = appx_{I\!I}\left(\bigcup_{A \in appx_{I\!I}(r),\ A \subset S} A\right) = appx_{I\!I}\left(\bigcup_{A \in E} A\right)$.

Montrons que B est le maximum de E.

1. Il est clair qu'uncun élément de E ne peut être plus grand que B puisque B est plus grand (cf. propriété 2.4) que l'union des éléments de E.
2. Il reste donc à montrer que $B \in E$.
   Il est clair que $B \subset S$ puisque $S \in I\!I^n$,et que B est l'approximation d'un ensemble inclus dans S (cf. propriété 2.4).

Il reste donc à montrer que :
si $A_1 \in appx_{I\!I}(r)$, $A_2 \in appx_{I\!I}(r)$, alors :
$appx_{I\!I}(A_1 \cup A_2) \in appx_{I\!I}(r)$,
ce qui, par extension à une union finie, implique que :
$B = appx_{I\!I}\left(\bigcup_{A \in appx_{I\!I}(r), A \subset S} A\right) \in appx_{I\!I}(r)$.
Alors, comme $B \subset S$, on en déduit que $B \in E$.

Soit donc $A_1 \in appx_{I\!I}(r)$, $A_2 \in appx_{I\!I}(r)$.
Soit B_1 le sous-ensemble de $I\!R^n$ défini par :
$B_1 = \{(x_1, ..., x_n) \in A_1 \ | \ (x_1, ..., x_n) \in r\}$.
De même, soit $B_2 = \{(x_1, ..., x_n) \in A_2 \ | \ (x_1, ..., x_n) \in r\}$.
Considérons l'ensemble $B_1 \cup B_2$, et montrons que :
$appx_{I\!I}(B_1 \cup B_2) = appx_{I\!I}(A_1 \cup A_2) \in appx_{I\!I}(r)$.

Il est clair que $appx_{I\!I}(B_1) = A_1$ et $appx_{I\!I}(B_2) = A_2$.
Donc, d'après le point 5 de la propriété 2.4, on a :
$appx_{I\!I}(B_1 \cup B_2) = appx_{I\!I}(A_1 \cup A_2)$,
or $appx_{I\!I}(B_1 \cup B_2) \in appx_{I\!I}(r)$ (par définition),
donc $appx_{I\!I}(A_1 \cup A_2) \in appx_{I\!I}(r)$.

9.3.5 Finitude du point fixe

Preuve de la propriété 2.15

Rappelons la transition de S_j à S_{j+1} :
s'il existe une relation r_i de (P) telle que :
$S_j \notin appx_{I\!I}(r_i)$, alors
$S_{j+1} = Max\{A \in appx_{I\!I}(r_i) \ | \ A \subset S_j\}$.
Or, $S_j \neq S_{j+1}$, puisque, dans le cas contraire, il en découlerait que $S_j \in appx_{I\!I}(r_i)$, ce qui est contraire à la définition de la transition.

Donc (S_j) est une suite strictement décroissante dans un ensemble fini, puisque c'est une suite d'éléments de $I\!I^n$, qui est un ensemble fini.
Donc cette suite est finie.

9.3.6 Propriété fondamentale du point fixe

Preuve de la propriété 2.16

On veut montrer ici que S_{j_0} est une solution du problème approché, et que c'est la plus grande.
La définition même de l'algorithme du point fixe que l'on a donnée prouve que S_{j_0} est une solution du problème approché car la condition d'arrêt est équivalente à :
$\forall i \in I,\ S_{j_0} \in appx_{\Pi}(r_i)$.

Soit maintenant B une solution de $appx_{\Pi}(P)$.
Montrons par récurrence que $B \subset S_{j_0}$.
Il est immédiat que $B \subset S_0$.

Soit $j < j_0$; supposons (hypothèse de récurrence) que $B \subset S_j$,
et montrons que $B \subset S_{j+1}$.
Soit r_i la relation du problème (P) telle que $S_j \notin appx_{\Pi}(r_i)$, qui assure la transition de S_j à S_{j+1}.
On a $S_{j+1} = Max\{A \in appx_{\Pi}(r_i) \mid A \subset S_j\}$,
or $B \in appx_{\Pi}(r_i)$ (par définition, puisque B est solution de $appx_{\Pi}(P)$).
De plus, $B \subset S_j$ (hypothèse de récurrence).
Donc $B \in \{A \in appx_{\Pi}(r_i) \mid A \subset S_j\}$,
donc $B \subset Max\{A \in appx_{\Pi}(r_i) \mid A \subset S_j\}$,
donc $B \subset S_{j+1}$.
La propriété de récurrence est donc bien vérifiée, pour tout $j \in 0..j_0$.

Donc $B \subset S_{j_0}$.

9.3.7 Inclusion des solutions du problème initial

Preuve de la propriété 2.17

Soit $(x_1, ..., x_n)$ une solution de $(P) = (r_i)_{i \in I}$.
On a donc $(x_1, ..., x_n) \in r_i, \ \forall i \in I$.
Donc $appx_{\mathit{II}}(\{x_1\} \times ... \times \{x_n\}) \in appx_{\mathit{II}}(r_i) \ \forall i \in I$ (par définition de l'approximation d'une relation).
Donc $appx_{\mathit{II}}(\{x_1\} \times ... \times \{x_n\})$ est solution du problème $appx_{\mathit{II}}(P)$.
Donc $appx_{\mathit{II}}(\{x_1\} \times ... \times \{x_n\}) \subset I_1 \times ... \times I_n$ (puisque $I_1 \times ... \times I_n$ est la plus grande solution du problème approché).
Donc $\{x_1\} \times ... \times \{x_n\} \subset I_1 \times ... \times I_n$.
Donc $(x_1, ..., x_n) \in I_1 \times ... \times I_n$.

9.3.8 Solutions réduites à un point

Preuve de la propriété 2.18

Soit $(P) = (r_i)_{i \in I}$ un problème sur $I\!R$ à n variables.
Supposons que la solution maximale du problème approché soit de la forme : $\{x_1\} \times ... \times \{x_n\}$.
On a donc $\forall i \in I, \ \{x_1\} \times ... \times \{x_n\} \in appx_{\mathit{II}}(r_i)$.
Donc $\forall i \in I, \ (x_1, ..., x_n) \in r_i$
(puisque c'est le seul sous-sensemble de $I\!R^n$ dont l'approximation peut être $\{x_1\} \times ... \times \{x_n\}$: cf. propriété 2.4).

Donc $(x_1, ..., x_n)$ est solution de (P).

9.3.9 Caractérisation des solutions de l'énumération

Preuve de la propriété 4.3

Soit $E = ((F_i)_{i \in Fail}, (S_i)_{i \in Succeed})$ une énumération complète.
Soit $(A_i)_{i \in AS}$ l'ensemble des éléments d'approximation atomiques tels que :
$\forall i \in AS$, la solution maximale de $appx_{\Pi}(P)_{|A_i}$ est A_i.

Il faut donc démontrer que $(A_i)_{i \in AS} = (S_i)_{i \in Succeed}$.

Il est immédiat que $(S_i)_{i \in Succeed} \subset (A_i)_{i \in AS}$ par définition de l'énumération.

Soit $i \in AS$; montrons par l'absurde que $A_i \in (S_i)_{i \in Succeed}$.
Supposons que $A_i \notin (S_i)_{i \in Succeed}$.
Alors, $A_i \subset \bigcup_{i \in Fail} Fi$ puisque l'énumération est complète.
Donc, il existe $j \in Fail$ tel que $A_i \subset F_j$.
Or la solution maximale de $appx_{\Pi}(P)_{|F_j}$ est l'ensemble vide, par définition de l'énumération.
Donc la solution maximale de $appx_{\Pi}(P)_{|A_i}$ est également vide puisque $A_i \in F_j$.
Or ceci est impossible, car en contradiction avec la définition de l'énumération.

Donc $(A_i)_{i \in AS} = (S_i)_{i \in Succeed}$

9.3.10 Caractérisation des solutions avec traitement global

Preuve de la propriété 6.1

Soit $A_1 \times ... \times A_p \in (S_i)_{i \in E}$.
C'est équivalent à : il existe $A_{p+1}, ..., A_n$, éléments d'approximation atomiques de Π tels que :
$A_1 \times ... \times A_n$ est solution atomique de $appx_{\Pi}((r_i)_{i \in I_1} \cup (r_i)_{i \in I_3})$.

$$\Longleftrightarrow \begin{cases} \forall i \in I_1,\ A_1 \times ... \times A_n \in appx_{\Pi}(r_i) \\ \forall i \in I_3,\ A_1 \times ... \times A_n \in appx_{\Pi}(r_i) \end{cases}$$

$$\Longleftrightarrow \begin{cases} \forall i \in I_1,\ A_1 \times ... \times A_n \in appx_{\Pi}(r_i) \\ \forall i \in I_2,\ A_1 \times ... \times A_n \in appx_{\Pi}(r_i) \end{cases}$$

En effet, un élément atomique appartient à l'approximation d'une relation spéciale si et seulement s'il est solution atomique de ses relations de base de décomposition (définition), et donc si et seulement s'il appartient à chacune de ses relations de base de décomposition.

Donc, c'est équivalent à : $A_1 \times ... \times A_p \in appx_{\Pi}((r_i)_{i \in I_1} \cup (r_i)_{i \in I_3})$.

9.3.11 Caractérisation des solutions sur une grille

Preuve de la propriété 6.3

1. Soit $i \in Succeed$.
   $S_i \in (Z_i)_{i \in I}$ par définition d'une énumération en zones définies,

et la solution maximale de $appx_{\Pi}(P)_{|S_i}$ n'est pas vide par définition générale d'une énumération.
Donc $S_i \in \{Z_i, i \in I \mid appx_{\Pi}(P)_{|Z_i}$ a une solution maximale non vide $\}$.

2. Soit $Z_i \in \{Z_i, i \in I \mid appx_{\Pi}(P)_{|Z_i}$ a une solution maximale non vide $\}$.
Montrons par l'absurde que $Z_i \in (S_i)_{i \in Succeed}$.
Supposons que ce ne soit pas le cas.
Alors, comme E est une énumération complète (et réalise donc une partition), on a :
$\exists j \in Fail$ tel que $Z_i \in F_j$ (par définition d'une énumération en zones définies).
Mais alors, $appx_{\Pi}(P)_{|Z_i}$ a une solution maximale non vide, ce qui est contraire à la définition d'une énumération.
Donc, $Z_i \in (S_i)_{i \in Succeed}$.

Donc $(S_i)_{i \in Succeed} = \{Z_i, i \in I \mid appx_{\Pi}(P)_{|Z_i}$ a une solution maximale non vide $\}$.

9.3.12 Caractérisation des solutions des énumérations par les bords

Preuve de la propriété 6.7

1. Par définition : $(A_i)_{i \in I} \subset (S_i)_{i \in I_s}$.
Donc, $appx_{\Pi}\left(\bigcup_{i \in I} A_i\right) \subset appx_{\Pi}\left(\bigcup_{i \in I_s} S_i\right)$.
(propriétés de base des approximations)

2. Soit $i \in I_s$.
Il est clair que pour tout $j \in J$, $S_i \notin B_j$.

puisque $\left\{ \begin{array}{l} appx_{\Pi}(P)_{|S_i} \ a\ pour\ solution\ maximale\ S_i \\ appx_{\Pi}(P)_{|B_j} \ a\ pour\ solution\ maximale\ \emptyset \end{array} \right.$

Donc $S_i \notin \bigcup_{j \in J} B_j$.

Donc $S_i \in I\!R^n \setminus \bigcup_{j \in J} B_j$.

Donc $S_i \in appx_{\Pi}\left(\bigcup_{i \in I} A_i\right)$.

(d'après la définition de la famille $(B_j)_{j \in J}$.

Donc, $\bigcup_{i \in I_s} S_i \subset appx_{\Pi}\left(\bigcup_{i \in I} A_i\right)$.

Donc $appx_{\Pi}\left(\bigcup_{i \in I_s} S_i\right) \subset appx_{\Pi}\left(\bigcup_{i \in I} A_i\right)$.

Donc $appx_{\Pi}\left(\bigcup_{i \in I_s} S_i\right) = appx_{\Pi}\left(\bigcup_{i \in I} A_i\right)$.

9.4 Programmes

9.4.1 Programme des carrés

```
/********/
/* Main */
go:-
   make_point_vector(_Vector),
   no_overlape_constraints(_Vector),
   scheduling_constraints(_Vector),

   write("Enumerating ..."),nl,
   enumerate(_Vector),

   write("Placement : "),write(_Vector),nl,nl.



/*****************************/
/* Basic constraints and data */

size_list([50,42,37,35,33,29,27,25,24,19,18,17,
           16,15,11,9,8,7,6,4,2]).


make_point_vector(V):-
   size_list(SL),
   make_point_vector2(SL,V).

make_point_vector2([S|L],[[Xd,Yd,Xf,Yf,S]|L2]):-
   Xd ~ int n cc(0,112 .-. S),
   Yd ~ int n cc(0,112 .-. S),
   Xf = plus(Xd,S),
   Yf = plus(Yd,S),
   make_point_vector2(L,L2).
make_point_vector2([],[]).
```

```
no_overlape_constraints([X|V]):-
   no_overlape_constraints2(X,V),
   no_overlape_constraints(V).
no_overlape_constraints([]).

no_overlape_constraints2(X,[Y|V]):-
   no_overlape_constraints3(X,Y),
   no_overlape_constraints2(X,V).
no_overlape_constraints2(_,[]).

no_overlape_constraints3([X1d,Y1d,X1f,Y1f,_],
                         [X2d,Y2d,X2f,Y2f,_]):-
   ble(X1f,X2d) or bge(X1d,X2f) or
                   ble(Y1f,Y2d) or bge(Y1d,Y2f).


/****************/
/* Redondancies */

scheduling_constraints(V):-
   loop(V,0).

loop(_,112):-!.
loop(V,I):-
   schedule(V,I,112,112),
   loop(V,I.+.1).

schedule([],_,0,0).
schedule([[Xd,Yd,_,_,S]|V],I,SX.+.S1,SY.+.S2):-
   Imin = I .-. S .+. 1,
   SX = S .*. bcc(Xd,Imin,I),
   SY = S .*. bcc(Yd,Imin,I),
   schedule(V,I,S1,S2).
```

```
/***************/
/* Enumeration */

enumerate(Squares):-
   split(Squares,Abs,Ord),
   enumerate2(Abs),
   enumerate2(Ord).

split([],[],[]).
split([[Xd,Yd,_,_,_]|L],[Xd|L1],[Yd|L2]):-
   split(L,L1,L2).

enumerate2(L):-
   all_bounded(L),
   !.
enumerate2(L):-
   min_list_unbound(L,Min),
   bigest_square(L,Min,X),
   choose(X,Min),
   enumerate2(L).

bigest_square([X|L],Min,Y):-
   number(X),
   !,
   bigest_square(L,Min,Y).
bigest_square([X|_],Min,X):-
   glb(X,Min),
   !.
bigest_square([_|L],Min,X):-
   bigest_square(L,Min,X).

choose(Min,Min).
choose(X,Min):-
   gt(X,Min).


all_bounded([]).
```

```
all_bounded([X|L]):-
   number(X),
   all_bounded(L).


min_list_unbound([],200).
min_list_unbound([X|L],M):-
   number(X),!,
   min_list_unbound(L,M).
min_list_unbound([X|L],M):-
   min_list_unbound(L,M1),
   glb(X,M2),
   M = min(M1,M2).
```

9.4.2 Programme du robot

```
% Main

go(MinX):-
   bras_robot([_,_,[X,0],T1,T2]),
   minimizei(X,[T1,T2],MinX).


% Contraintes

bras_robot([[0,0],[X1,Y1],[X2,Y2],Teta1,Teta2]):-
   Teta1_degre ~ cc(30,150),
   Teta1 = Teta1_degre.*.pi./.180,
   Teta2_degre ~ cc(0,360),
   le(abs(Teta2_degre .-. Teta1_degre),80),
   Teta2 = Teta2_degre.*.pi./.180,
   X1 = cos(Teta1),
   Y1 = sin(Teta1),
   X2 = X1 .+. cos(Teta2),
   Y2 = Y1 .+. sin(Teta2).


% Minimisation

minimizei(X,L,M):-
   glb(X,Min),
   search_one_sol(X,L,Max),
   dichotomy(X,L,Min,Max,M).

search_one_sol(X,L,_):-
   record(one_sol,no),
   once(realsplit(L)),
   glb(X,V),
   record(one_sol,V),
   fail.
```

```
search_one_sol(_,_,V):-
   recorded(one_sol,no),
  !,
   throw(error("No solution",dichotomy)).
search_one_sol(_,_,V):-
   recorded(one_sol,V).

dichotomy(X,L,Min,Max,Min):-
   lt(Max.-.Min,0.0001),!.
dichotomy(X,L,Min,Max,Vmin):-
   middle(Min,Max,Mid),
   verify((X=Mid,realsplit(L))),
  !,
   dichotomy(X,L,Min,Mid,Vmin).
dichotomy(X,L,Min,Max,Vmin):-
   middle(Min,Max,Mid),
   dichotomy(X,L,Mid,Max,Vmin).

middle(Min,Max,Mid):-
   Mid = (Min.+.Max)./.2.

verify(P):- \+(\+(P)).
```

9.4.3 Implantation en Prolog IV du méta-Point fixe

```
/* Le prédicat suivant exécute le méta-Point fixe. */

execute_calls(ListCalls):-
   execute_calls(ListCalls,Modif),
   (
      Modif=no_modif,
      !
   ;
      execute_calls(ListCalls)
   ).

execute_calls([],no_modif).
execute_calls([[ToCall,Lbase,Lref]|Lcalls],Modif):-
   copy_domains(Lref,Lbase,Modif1),
   (
      Modif1=no_modif,
      !,
      execute_calls(Lcalls,Modif)
   ;
      Call =.. [ToCall,Lbase],
      call(Call),
      copy_domains(Lbase,Lref,Modif2),
      execute_calls(Lcalls,Modif3),
      or_modif(Modif2,Modif3,Modif)
   ).


or_modif(no_modif,no_modif,no_modif):-!.
or_modif(_,_,modif).


/* Le prédicat suivant copie les domaines, en signalant
   si au moins un des domaines-cible a été réduit. */
```

```
copy_domains([],[],no_modif).
copy_domains([X1|L1],[X2|L2],Modif):-
   appx_bounds(X1,Bi1,Bs1),
   appx_bounds(X2,Bi2,Bs2),
   (
      Bi1=Bi2,
      Bs1=Bs2,
      !,
      copy_domains(L1,L2,Modif)
   ;
      cc(X2,Bi1,Bs1),
      Modif=modif,
      copy_domains(L1,L2,_)
   ).


/* Interface du prédicat bounds/3 qui rend un très grand
   nombre en cas d'infini */

appx_bounds(X,M,N):-
   appx_min(X,M),
   appx_max(X,N).

appx_min(X,M):-
   glb(X,M),!.
appx_min(X,-1.0e+100).

appx_max(X,M):-
   lub(X,M),!.
appx_max(X,1.0e+100).
```

9.4.4 Broyden-Banded avec méta-Point fixe

```
/*
   Programme : [BroydenG]
   Résolution du problème de Broyden-Banded
      en utilisant le traitement global des expressions
      complexes par la méthode du méta-Point fixe.
   N : numéro du problème
   V : Vecteur résultat
 */


/**************************************/
/*
   Prédicat d'entrée dans le programme.
   Après avoir posé les contraintes de base,
       il construit la liste de contraintes spéciales
       destinée au méta-Point fixe, puis appelle
       une énumération construite sur celui-ci.
*/

go(N,V):-
   % Contraintes de base
   size(V)=N,
   bound_list(V),
   put_constraints(V,N,Lexpr1,Lexpr2),
   % Création de la liste des contraintes spéciales
   size(V1)=N,
   size(V2)=N,
   build_expression1(V1,Lexpr1b),
   build_expression2(V2,Lexpr2b),
   split_to_call_list(V o V,Lexpr1 o Lexpr2,V1 o V2,
                      Lexpr1b o Lexpr2b,to_shrink,Lcall),
   % Appel de l'énumération avec méta-Point fixe
   enum_with_calls(V,Lcall).
```

```
/*********************************/
/*
   Prédicats assurant la pose des contraintes
      de base.
*/

bound_list([]).
bound_list([X|L]):-
   cc(X,-1,1),
   bound_list(L).

put_constraints(V,N,Lexpr1,Lexpr2):-
   build_expression1(V,Lexpr1),
   build_expression2(V,Lexpr2),
   build_constraints(Lexpr1,Lexpr2,N,1).

build_expression1([],[]).
build_expression1([X|L],[E | LL]):-
   E = X .*. (2 .+. 5 .*. square(X)),
   build_expression1(L,LL).

build_expression2([],[]).
build_expression2([X|L],[E | LL]):-
   E = X .*. (1 .+. X),
   build_expression2(L,LL).

build_constraints(_,_,N,N+1):-!.
build_constraints([E1|L1],L2,N,I):-
   build_sum(L2,N,I,S),
   E1 .+. 1 .-. S = 0,
   build_constraints(L1,L2,N,I+1).

build_sum(L,N,I,S):-
   build_sum2(L,I,max(1,I.-.5),min(N,I.+.1),S).
```

```
build_sum2(_,_,J.+.1,J,0):-!.
build_sum2(L,I,I,F,S):-
   !,
   build_sum2(L,I,I.+.1,F,S).
build_sum2(L,I,J,F,S):-
   build_sum2(L,I,J.+.1,F,S1),
   S = S1 .+. L:J.


/**************************************/
/*
   Formation de la liste des contraintes spéciales .
*/

split_to_call_list([],[],[],[],_,[]).
split_to_call_list([X|V],[E|Lexpr],[X2|V2],[E2|Lexpr2],
                        Pred,[[Pred,[X2,E2],[X,E]]|Lcall]):-
   split_to_call_list(V,Lexpr,V2,Lexpr2,Pred,Lcall).


/***************************************/
/*
   Prédicat assurant la réduction des domaines
      des contraintes spéciales par calculs d'optima .
*/

to_shrink(L):-
   shrink_box(L,L).

shrink_box([],Lc).
shrink_box([X|L],Lc):-
   lower_boundi(X,Lc,Min),
   upper_boundi(X,Lc,Max),
   cc(X,Min,Max),
   shrink_box(L,Lc).
```

```
/************************************/
/*
   Prédicats assurant le calcul des optima
      d'après la méthode simple.
*/

lower_boundi(X,L,_):-
   record(lower_boundi_sol,0),
   lower_boundi_eval(X,L).
lower_boundi(_,_,M):-
   recorded(lower_boundi_val,M).

lower_boundi_eval(X,L):-
   realsplit([X],smallest_domain,10000,float_mode),
   glb(X,MinX),
   record(lower_boundi_val,MinX),
   realsplit(L,smallest_domain,10000,float_mode),
   !,
   record(lower_boundi_sol,1),
   fail.


upper_boundi(X,L,_):-
   record(upper_boundi_sol,0),
   upper_boundi_eval(X,L).
upper_boundi(_,_,M):-
   recorded(upper_boundi_val,M).

upper_boundi_eval(X,L):-
   realsplit([.-.X],smallest_domain,10000,float_mode),
   lub(X,MaxX),
   record(upper_boundi_val,MaxX),
   realsplit(L,smallest_domain,10000,float_mode),
   !,
   record(upper_boundi_sol,1),
   fail.
```

```
/****************************************/
/*
   Enumération avec sur chaque noeud execution
      d'un meta-point fixe en Prolog
   Condition d'arrêt : Toutes les variables ont
      des domaines composés de moins de 100000
      intervalles flottants atomiques.
    Choix de la variable : Première variable dans
       l'ordre de la liste dont le domaine contient
       plus de 100000 intervalles flottants atomiques.
    Découpage du domaine : En deux parties
       contenant autant d'intervalles flottants atomiques
*/

enum_with_calls(List,ListCalls):-
   execute_calls(ListCalls),
   enum_with_calls2(List,ListCalls).

enum_with_calls2(List,ListCalls):-
   stop_condition(List),
   !.
enum_with_calls2(List,ListCalls):-
   choose_variable(List,Var),
   middle(Var,Mid),
   cut_domain(Var,Mid),
   enum_with_calls(List,ListCalls).

stop_condition([]).
stop_condition([X|L]):-
   float_size(X,le(100000)),
   stop_condition(L).

choose_variable([X|L],X):-
   float_size(X,gt(100000)),
   !.
```

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

middle(X,M):-
   bounds(X,Min,Max),
   float_rank(Min,RMin),
   float_rank(Max,RMax),
   float_rank(M,floor((RMin.+.RMax)./.2)).

cut_domain(X,M):-
   le(X,M).
cut_domain(X,M):-
   gt(X,M).


/*********************************/
/*
   Méta-Point fixe
*/

execute_calls(ListCalls):-
   execute_calls(ListCalls,Modif),
   (
      Modif=no_modif,
      !
   ;
      execute_calls(ListCalls)
   ).

execute_calls([],no_modif).
execute_calls([[ToCall,Lbase,Lref]|Lcalls],Modif):-
   copy_domains(Lref,Lbase,Modif1),
   (
      Modif1=no_modif,
      !,
      execute_calls(Lcalls,Modif)
   ;
```

```
      Call =.. [ToCall,Lbase],
      call(Call),
      copy_domains(Lbase,Lref,Modif2),
      execute_calls(Lcalls,Modif3),
      or_modif(Modif2,Modif3,Modif)
   ).

or_modif(no_modif,no_modif,no_modif):-!.
or_modif(_,_,modif).

copy_domains([],[],no_modif).
copy_domains([X1|L1],[X2|L2],Modif):-
   appx_bounds(X1,Bi1,Bs1),
   appx_bounds(X2,Bi2,Bs2),
   (
      Bi1=Bi2,
      Bs1=Bs2,
      !,
      copy_domains(L1,L2,Modif)
   ;
      cc(X2,Bi1,Bs1),
      Modif=modif,
      copy_domains(L1,L2,_)
   ).

appx_bounds(X,M,N):-
   appx_min(X,M),
   appx_max(X,N).

appx_min(X,M):-
   glb(X,M),!.
appx_min(X,-1.0e+100).

appx_max(X,M):-
   lub(X,M),!.
appx_max(X,1.0e+100).
```

9.4.5 Kinematics avec traitement redondant linéaire

```
/* Nom du programme : [kineF] */

go(L):-
   L = [S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6],
   constraints(L),
   realsplit(L,_,0.01).

constraints([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6]):-
   % Factorized primary constraints
   C5.*.S6.*. uminus(Aux2) .+. C6.*. Aux3 .-. 0.4077 = 0,
   S5.*.C1.*. Aux3 .+. S1.*.C5 .-. 1.9115 = 0,
   S5.*. Aux2 .-. 1.9791 = 0,
   C1.*. Aux1 .-. 4.0616 = 0,
   S1.*. Aux1 .-. 1.7172 = 0,
   Aux4 .-. 3.9701 = 0,
   % Common sub-expressions
   Aux1 = 3.*.C2 .+. 2.*.C3 .+. C4,
   Aux4 = 3.*.S2 .+. 2.*.S3 .+. S4,
   Aux2 = S2.+.S3.+.S4,
   Aux3 = C2.+.C3.+.C4,
   % Linear combination of common sub-expressions
   Aux1 .-. Aux3 = 2.*.C2 .+. C3,
   Aux4 .-. Aux2 = 2.*.S2 .+. S3,
   Aux1 .-. 2.*.Aux3 = C2 .-. C4,
   Aux1 .-. 3.*.Aux3 = 0 .-. C3 .-. 2.*.C4,
   Aux4 .-. 2.*.Aux2 = S2 .-. S4,
   Aux4 .-. 3.*.Aux2 = 0 .-. S3 .-. 2.*.S4,
   % Circle constraints
   square(S1) .+. square(C1) = 1,
   square(S2) .+. square(C2) = 1,
   square(S3) .+. square(C3) = 1,
   square(S4) .+. square(C4) = 1,
   square(S5) .+. square(C5) = 1,
   square(S6) .+. square(C6) = 1.
```

9.4.6 Kinematics et collaboration linéaire/Intervalles

```
go:-
   constraints([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6,
                A1,A2,A3,A4]),
   lin_system([SS2,CC2,SS3,CC3,SS4,CC4,AA1,AA2,AA3,AA4]),
   set_primary_domains([SS2,CC2,SS3,CC3,SS4,CC4,
                        AA1,AA2,AA3,AA4],-100,100),
   Lcall = [[to_lin,[SS2,CC2,SS3,CC3,SS4,CC4,AA1,AA2,AA3,AA4],
                   [S2,C2,S3,C3,S4,C4,A1,A2,A3,A4]]],
   enum_with_calls([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6],
                  greatest_domain,prec(0.01),real_mode,
                  1,Lcall).


set_primary_domains([],Min,Max).
set_primary_domains([X|L],Min,Max):-
   gelin(X,Min),lelin(X,Max),
   set_primary_domains(L,Min,Max).


lin_system([S2,C2,S3,C3,S4,C4,Aux1,Aux2,Aux3,Aux4]):-
   Aux1 = 3*C2 + 2*C3 + C4,
   Aux4 = 3*S2 + 2*S3 + S4,
   Aux2 = S2 + S3 + S4,
   Aux3 = C2 + C3 + C4.


constraints([S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6,
             Aux1,Aux2,Aux3,Aux4]):-
   % Factorized primary constraints
   C5.*.S6.*. uminus(Aux2) .+. C6.*. Aux3 .-. 0.4077 = 0,
   S5.*.C1.*. Aux3 .+. S1.*.C5 .-. 1.9115 = 0,
   S5.*. Aux2 .-. 1.9791 = 0,
```

```
   C1.*. Aux1 .-. 4.0616 = 0,
   S1.*. Aux1 .-. 1.7172 = 0,
   Aux4 .-. 3.9701 = 0,
   % Common sub-expressions
   Aux1 = 3.*.C2 .+. 2.*.C3 .+. C4,
   Aux4 = 3.*.S2 .+. 2.*.S3 .+. S4,
   Aux2 = S2.+.S3.+.S4,
   Aux3 = C2.+.C3.+.C4,
   % Circle constraints
   square(S1) .+. square(C1) = 1,
   square(S2) .+. square(C2) = 1,
   square(S3) .+. square(C3) = 1,
   square(S4) .+. square(C4) = 1,
   square(S5) .+. square(C5) = 1,
   square(S6) .+. square(C6) = 1.



% Enumération avec sur chaque noeud execution
% d'un meta-point fixe en Prolog


enum_with_calls(List,Choice,Stop,Mode,Depth,ListCalls):-
   execute_calls(ListCalls),
   enum_with_calls2(List,Choice,Stop,Mode,Depth,ListCalls).

enum_with_calls2(List,Choice,Stop,Mode,Depth,ListCalls):-
   e__stop_function(List,Stop,Mode,Depth),
   !.
enum_with_calls2(List,Choice,Stop,Mode,Depth,ListCalls):-
   e__choice_function(List,Choice,Mode,Stop,Var),
   e__cut_function(Mode,Var,Mid,Dir),
   e__cut_domain(Var,Mid,Dir),
   enum_with_calls(List,Choice,Stop,Mode,Depth.+.1,ListCalls).
```

```
execute_calls(ListCalls):-
   execute_calls(ListCalls,Modif),
   (
      Modif=no_modif,
      !
   ;
      execute_calls(ListCalls)
   ),
  true.

execute_calls([],no_modif).
execute_calls([[ToCall,Lbase,Lref]|Lcalls],Modif):-
   copy_domains(Lref,Lbase,Modif1),
   (
      Modif1=no_modif,
      !,
      execute_calls(Lcalls,Modif)
   ;
      Call =.. [ToCall,Lbase],
      call(Call),
      copy_domains(Lbase,Lref,Modif2),
      execute_calls(Lcalls,Modif3),
      or_modif(Modif2,Modif3,Modif)
   ).


or_modif(no_modif,no_modif,no_modif):-!.
or_modif(_,_,modif).



copy_domains([],[],no_modif).
copy_domains([X1|L1],[X2|L2],Modif):-
   appx_bounds(X1,Bi1,Bs1),
   appx_bounds(X2,Bi2,Bs2),
   (
      Bi1=Bi2,
```

```
      Bs1=Bs2,
      !,
      copy_domains(L1,L2,Modif)
   ;
      cc(X2,Bi1,Bs1),
      (
         Taille1 = Bs1 .-. Bi1,
         Taille2 = Bs2 .-. Bi2,
         le(abs(Taille2 .-. Taille1), 0.00001),
         !,
         copy_domains(L1,L2,Modif)
      ;
         Modif=modif,
         copy_domains(L1,L2,_)
      )
   ).


appx_bounds(X,M,N):-
   appx_min(X,M),
   appx_max(X,N).

appx_min(X,M):-
   glb(X,M),!.
appx_min(X,-1.0e+100).

appx_max(X,M):-
   lub(X,M),!.
appx_max(X,1.0e+100).


/* Réévaluation des contraintes linéaires */

to_lin(L):-
   create_box(L,LB),
   copy_symbolic_domains(LB,L).
```

```
copy_symbolic_domains([],[]).
copy_symbolic_domains([[Min,Max]|LB],[X|L]):-
   cc(X,Min,Max),
   copy_symbolic_domains(LB,L).

create_box(L,_):-
   record(list_box,[]),
   domains_to_lin(L),
   narrow_lin_box2(L),
   fail.
create_box(_,LB):-
   recorded(list_box,LB),
   dif(LB,no_sol).

domains_to_lin([]).
domains_to_lin([X|L]):-
   bounds(X,Min,Max),
   gelin(X,Min),
   lelin(X,Max),
   domains_to_lin(L).


narrow_lin_box2([]):-!.
narrow_lin_box2([X|L]):-
   lower_bound(X,Min),
   upper_bound(X,Max),
  !,
   narrow_lin_box2(L),
   recorda(list_box,[Min,Max]).
narrow_lin_box2(_):-
   record(list_box,no_sol).
```

9.4.7 Enumération entière sur une grille

```
% Une énumération sur une grille
mon_enum_ent(L):-
   condition_d_arret(L),
   !.
mon_enum_ent(L):-
   choix_variable(L,X),
   milieu(X,M),
   decouper(X,M),
   mon_enum_int(L).


% Condition de terminaison
condition_d_arret([]).
condition_d_arret([X|L]):-
   \+(domaine_coupe_grille(X)),
   condition_d_arret(L).

% Choix de la variable
choix_variable([X|L],X):-
   domaine_coupe_grille(X),
   !.
choix_variable([X|L],Y):-
   choix_variable(L,Y).

% Détermination du milieu (sur la grille)
milieu(X,M):-
   bounds(X,Min,Max),
   (
      gt(Max .-. Min,6),
      !,
      M = 3 .*. floor((Min.+.Max)./.6)
   ;
      M = 3 .*. (floor(Min./.3) .+. 1)
   ).
```

```
% Découpage du domaine avec backtracking
decouper(X,M):-
   le(X,M).
decouper(X,M):-
   gt(X,M).

% Y a-t-il un élément de la grille dans le domaine de X
domaine_coupe_grille(X):-
   bounds(X,Min,Max),
   suivant_sur_grille(Min,Grille),
   lt(Grille,Max).

% Rang du prochain flottant sur la grille
suivant_sur_grille(I,Isuivant):-
   Isuivant = 3 .*. (floor(I./.3).+.1).
```

9.4.8 Contraction par les bords

```
/*
   Prédicat d'énumération par les bords
   selon la 3-B arc-consistence de O. Lhomme.
   Officie systematiquement en float_mode
   pour obeir a la sementique.
*/


narrow_box(List,Prec):-
   greatest_domain_size(List,MaxSize),
   narrow_box(List,Prec,floor(MaxSize./.4)).


% Recherche du plus grand domaine.
greatest_domain_size([X],S):-
   !,
   float_size(X,S).
greatest_domain_size([X|L],max(S,M)):-
   float_size(X,S),
   greatest_domain_size(L,M).

% narrow_box/3: boucle principale
% avec réduction de la taille
narrow_box(_,Prec,Size):-
   Prec >= Size,
   !.
narrow_box(List,Prec,Size):-
   narrow_box2(List,Prec,Size,modif),
   NewSize = floor(Size ./. 2),
   narrow_box(List,Prec,NewSize).

% narrow_box2/4: boucle de tentative
% de réduction jusqu'à impossibilité
```

```
narrow_box2(_,_,_,no_modif):-!.
narrow_box2(List,Prec,Size,modif):-
   narrow_box3(List,Prec,Size,Modif),
   narrow_box2(List,Prec,Size,Modif).

% narrow_box3/4 : boucle sur les variables
% pour tentative de reduction
narrow_box3([],_,_,no_modif).
narrow_box3([V|L],Prec,Size,Modif):-
   narrow_box4(V,Prec,Size,Modif1),
   narrow_box3(L,Prec,Size,Modif2),
   or_modif(Modif1,Modif2,Modif).

or_modif(no_modif,no_modif,no_modif):-!.
or_modif(_,_,modif).

% narrow_box4/4 : tentative de réduction
% du domaine d'une variable
narrow_box4(V,Prec,Size,no_modif):-
   float_size(V,SizeV),
   (SizeV =< Prec ; SizeV =< 2.*.Size),
   !.
narrow_box4(V,_,Size,Modif):-
   shrink_left(V,Size,Modif1),
   shrink_right(V,Size,Modif2),
   or_modif(Modif1,Modif2,Modif).

% Les deux shrinks modifient eventuellement
% une des bornes de la variable
shrink_left(V,Size,modif):-
   glb(V,Min),
   float_rank(Min,RMin),
   float_rank(NewMin,RMin.+.Size),
   \+(le(V,NewMin)),
   !,
   ge(V,NewMin).
shrink_left(_,_,no_modif).
```

```
shrink_right(V,Size,modif):-
   lub(V,Max),
   float_rank(Max,RMax),
   float_rank(NewMax,RMax.-.Size),
   \+(ge(V,NewMax)),
   !,
   le(V,NewMax).
shrink_right(_,_,no_modif).
```

9.4.9 Enumération par les bords

```
/* Enumération par les bords */


englobing_box(List,Prec):-
   find_solution(no_box,List,Prec,0,SymbolicSol),
   check_out_and_complete(List,Prec,SymbolicSol,1,
                          EnglobingSol),
   apply_box(EnglobingSol,Prec,List).



% find_solution/5 trouve une solution au probleme
% et en rend une version symbolique sous la forme
% d'une liste de couples.
find_solution(Lbool,List,Prec,Sens,_):-
   record(symb,[]),
   find_one_solution(Lbool,List,Prec,Sens),
   domains_to_symbolic(List,SymbolicSol),
   record(symb,SymbolicSol),
   fail.
find_solution(_,_,_,_,SymbolicSol):-
   recorded(symb,SymbolicSol),
   dif(SymbolicSol,[]).

find_one_solution(Lbool,List,Prec,0):-
   outside_box(Lbool,List,ListIn,ListOut),
   realsplit(ListOut,_,Prec,float_mode),
   realsplit(ListIn,_,Prec,float_mode),
   !.
find_one_solution(Lbool,List,Prec,1):-
   negate_list(List,NList),
   outside_box(Lbool,List,ListIn,ListOut),
   realsplit(ListOut,_,Prec,float_mode),
   realsplit(ListIn,_,Prec,float_mode),
   !.
```

```
outside_box(no_box,Lv,[],Lv):-!.
outside_box([1,0|_],Lv,[],Lv).
outside_box([0,1|_],Lv,[],Lv).
outside_box([0,0|L],[V|Lv],[V|LvIn],LvOut):-
   outside_box(L,Lv,LvIn,LvOut).

negate_list([],[]).
negate_list([V|L],[.-.V|LL]):-
   negate_list(L,LL).


% domains_to_symbolic/2 convertit une liste de
% variables en une liste de doublets formés des
% bornes des domaines des variables.
domains_to_symbolic([],[]).
domains_to_symbolic([X|L],[[Bi,Bs]|LD]):-
   bounds(X,Bi,Bs),
   domains_to_symbolic(L,LD).


% check_out_and_complete/5 cherche une solution
% a plus de Prec de la boite symbolique. S'il en
% trouve une, il constitue une nouvelle boite
% symbolique, et recommence. Sens represente
% le sens d'énumération (gauche a droite ou le contraire).
check_out_and_complete(List,Prec,Box,Sens,EnglobingBox):-
   find_solution_outbox(List,Prec,Box,Sens,SymbolicSol),
   !,
   compute_new_box(Box,SymbolicSol,NewBox),
   check_out_and_complete(List,Prec,NewBox,bnot(Sens),
                          EnglobingBox).
check_out_and_complete(_,_,Box,_,Box).


% compute_new_box/3 calcule la plus petite boite
% contenant deux boites.
```

```
compute_new_box([],[],[]).
compute_new_box([[Bi1,Bs1]|LD1],[[Bi2,Bs2]|LD2],
                [[min(Bi1,Bi2),max(Bs1,Bs2)]|LD]):-
   compute_new_box(LD1,LD2,LD).


% find_solution_outbox/4 cherche une solution
% a plus de Prec de Box.
find_solution_outbox(List,Prec,Box,Sens,_):-
   record(symb2,[]),
   exclude_box(List,Prec,Box,Lbool),
   find_solution(Lbool,List,Prec,Sens,SymbolicSol),
   record(symb2,SymbolicSol),
   fail.
find_solution_outbox(_,_,_,_,SymbolicSol):-
   recorded(symb2,SymbolicSol),
   dif(SymbolicSol,[]).

exclude_box([],_,[],[]).
exclude_box([V|L],Prec,[[Bi,Bs]|LD],[Bool1,Bool2|Lbool]):-
   float_rank(Bi,RBi),
   float_rank(Bs,RBs),
   float_rank(BBi,RBi.-.Prec),
   float_rank(BBs,RBs.+.Prec),
   Bool1 = ble(V,BBi),
   Bool2 = bge(V,BBs),
   exclude_box(L,Prec,LD,Lbool).

% apply_box/2 est la reciproque de domains_to_symbolic,
% a Prec pres.
apply_box([],_,[]).
apply_box([[Bi,Bs]|LD],Prec,[V|LV]):-
   float_rank(Bi,RBi),
   float_rank(Bs,RBs),
   float_rank(BBi,RBi.-.Prec),
   float_rank(BBs,RBs.+.Prec),
   cc(V,BBi,BBs),
```

```
apply_box(LD,Prec,LV).
```