Programmation logique et contraintes

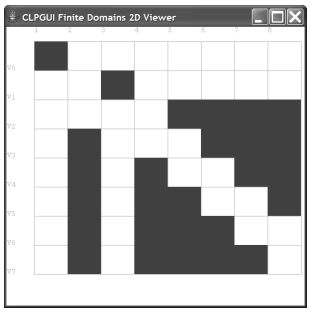
François Fages, INRIA Rocquencourt

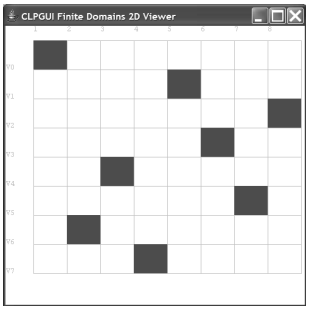
Mots-clés : programmation par contraintes, résolution de contraintes, propagation de contraintes, logique, calcul symbolique, arithmétique d'intervalles, optimisation combinatoire, ordonnancement, procédures de recherche, heuristiques.

Résumé : la programmation logique avec contraintes est porteuse d'un grande ambition pour la programmation : celle d'en faire essentiellement une tâche de modélisation, par des équations, des contraintes, des formules logiques. À la suite de différents travaux pionniers menés dans les années 70, ce courant de programmation est né au milieu des années 80 d'un rapprochement de la programmation logique, des techniques de propagation de contraintes issues de l'intelligence artificielle, et de celles de programmation linéaire issues de la recherche opérationnelle. Les succès rencontrés très rapidement en résolution de problèmes combinatoires, y compris dans des problèmes classiques de recherche opérationnelle comme en ordonnancement par exemple, ont induit une forte vitalité industrielle et académique dans ce domaine. Les enjeux sont aujourd'hui d'explorer de nouvelles applications et d'aller vers une plus grande automatisation, comme cela a été fait pour les outils de programmation linéaire en nombres entiers par exemple.

1 Introduction

Il est possible de présenter les idées essentielles de la programmation par contraintes à travers un petit exemple combinatoire, introduit par Bezzel en 1848 : celui de placer N reines sur un échiquier $N \times N$ sans qu'elles soient en prise, c'est-à-dire placées sur une même colonne, ligne ou diagonale. La figure suivante montre une telle solution pour un échiquier 8×8.





L'idée est d'utiliser les contraintes du problème de façon active pour réduire la taille de l'espace de recherche, en retirant les valeurs impossibles du domaine des variables *avant* leur énumération. La figure suivante montre l'effet des contraintes sur le domaine des variables après le placement des deux premières reines (en lignes) :

L'énumération des valeurs possibles des variables se fait en développant un arbre de recherche. Les calculs symboliques effectués par les contraintes ont pour effet de réduire le nombre de choix à explorer et de stopper immédiatement l'exploration dès que le domaine d'une des variables devient vide. Ils permettent également de guider les choix par des heuristiques en choisissant d'énumérer en premier les variables qui ont le plus petit domaine par exemple. La figure 1.1 montre l'arbre de recherche exploré pour énumérer les 92 solutions au problème des huit reines, ainsi que l'arbre exploré en ajoutant la contrainte $V0 < V7$ entre la première et la dernière variable de la liste, de façon à casser une symétrie. L'utilisation active des contraintes pour élaguer et diriger l'exploration de l'arbre de recherche permet d'obtenir des temps d'exécution qui sont inférieurs de plusieurs ordres de grandeur à ceux obtenus par des algorithmes classiques d'énumération et de test *a pos-*

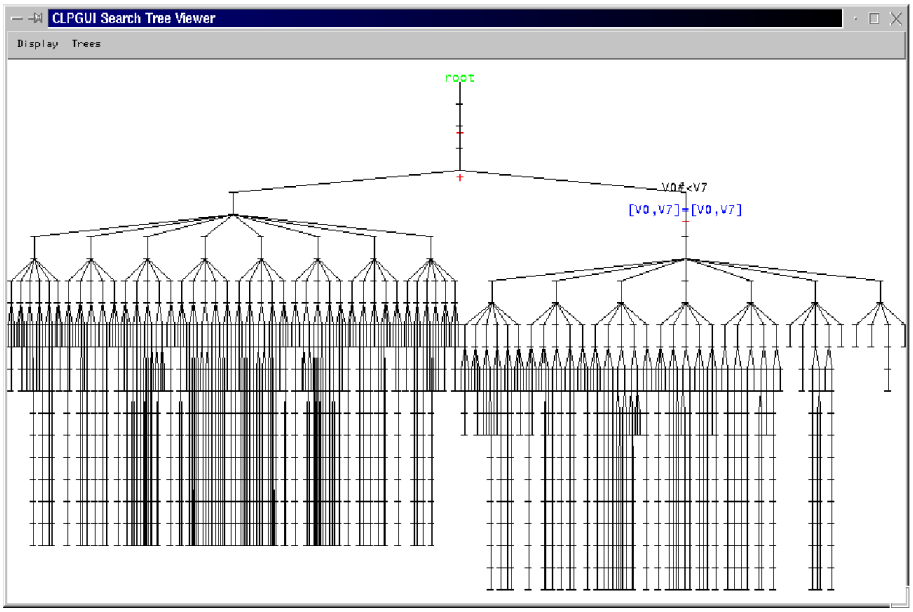


FIG. 1.1 – Arbres de recherche explorés pour trouver toutes les solutions au problème des huit reines, sans et avec la contrainte d'élimination de symétrie $V0 < V7$

teriori des contraintes. Le programme (donné ci-dessous) permet par exemple de placer deux cents reines sur un échiquier 200×200 en 80 millisecondes, alors qu'un programme classique vérifiant les contraintes après chaque énumération serait limité à des problèmes de taille inférieure à 15.

La beauté de la programmation par contraintes vient également de l'extrême concision des programmes qui se concentrent sur la modélisation du problème. Les langages de programmation par contraintes offrent des bibliothèques de contraintes prédéfinies sur différents domaines de calcul, et des moyens de définir le problème à résoudre en introduisant (éventuellement de façon récursive) les variables du problème et les contraintes qui les lient. Le langage de programmation logique Prolog, introduit au début des années 70 par A. Colmerauer et R. Kowalski, est adapté à ce cadre car il ne manipule que des relations et permet de définir de nouvelles relations à partir de relations prédéfinies (les contraintes de base) par des clauses logiques. Le problème des N reines peut ainsi être modélisé par le programme Prolog avec contraintes sur les nombres entiers et les listes suivant :

```
queens(N,L):- length(L,N), domain(L,1,N),
              safe(L), labeling(L).
length([],0).
length([X|L],N):- N=N1+1, length(N1,L).
safe([]).
safe([X|L]):- noattack(L,X,1), safe(L).
noattack([],_,_).
noattack([Y|L],X,D):-
              X=/=Y, X=/=Y+D, X+D=/=Y,
              D1=D+1, noattack(L,X,D1).
? queens(8,L).
L=[1,5,8,6,3,7,2,4]
```

À la requête `queens(8,L)`, l'interpréteur fournit une première solution sous forme d'une contrainte réponse et énumère à la demande les autres solutions. La première clause du programme spécifie que la relation `queens(N,L)` est vraie si L est une liste de longueur N (prédicat `length(L,N)`), chaque élément de la liste prend ses valeurs entre 1 et N (contrainte `domain(L,1,N)`), les contraintes de non prise sont posées (prédicat `safe(L)`) et les valeurs des variables sont énumérées (prédicat `labeling(L)`). Le prédicat `length` est défini par un fait pour le cas de la liste vide, et une règle récursive pour le cas d'une liste à au moins un élément. Une telle définition Prolog est réversible dans le sens où elle sert aussi bien à calculer la longueur de la liste si la liste est connue, qu'à construire une liste (de variables) de longueur N si la liste en argument est une inconnue. Le prédicat `safe` pose les contraintes de non prise entre chaque reine et les suivantes à partir de la distance 1 (prédicat `noattack(L,X,1)`). Les contraintes de non prise s'expriment par des contraintes d'inéquation notées `X=/=Y`.

D'un point de vue général, l'essence de la programmation par contraintes consiste donc à modéliser le problème à résoudre par des relations sur des variables mathématiques, et à effectuer des calculs symboliques sur ces structures d'information partielle, sans nécessairement fixer de valeurs précises. On peut alors s'intéresser à la programmation par contraintes de différents points de vue : fondements en logique mathématique, algorithmes de résolution, procédures de recherche, langages de programmation, applications, historique, ... Les sections suivantes traitent des quatre premiers points dans cet ordre, les applications et certaines références historiques étant mentionnées au cours du texte.

2 Qu'est-ce qu'une contrainte ?

Une contrainte est une formule logique du premier ordre que l'on interprète dans une structure mathématique fixée. Ce domaine d'interprétation, noté $\mathcal{D}$, qui représente le "domaine du discours", est en général une combinaison de structures élémentaires, numériques : nombres entiers naturels, nombres réels,..., ou symboliques : mots, listes, termes (arbres étiquetés), structures de données, etc. Dans un système de contraintes $\mathcal{C} = (\mathcal{L}, \mathcal{D})$, on fixe également le langage des contraintes $\mathcal{L}$, qui détermine l'ensemble des formules logiques autorisées. Ce langage est supposé clos par conjonction, c'est-à-dire que la conjonction d'un nombre fini de contraintes est toujours une contrainte. En revanche l'utilisation des autres connecteurs logiques et des quantificateurs est optionnelle et peut être limitée à des formes simples, comme par exemple $\forall y\ x \neq f(y)$ qui contraint x à ne pas être de la forme f.

Le problème de décision auquel on s'intéresse est celui de la satisfiabilité d'une contrainte, c'est-à-dire de l'existence d'une valuation dans $\mathcal{D}$ des variables pour laquelle la formule est vraie. Supposer que la satisfiabilité des contraintes est un problème décidable dans une interprétation I revient, d'un point de vue logique, à supposer que la structure I peut être axiomatisée par une théorie $\mathcal{T}$ complète pour le langage des contraintes. Ce point de vue est fructueux car il permet, d'une part, d'étudier la décidabilité des langages de contraintes, et d'autre part, de déduire d'une axiomatisation de la structure certains algorithmes de satisfaction de contraintes comme nous le verrons dans la section suivante.

L'arithmétique de Presburger est un premier exemple de théorie complète qui fut donné en 1929. Cette théorie concerne la structure additive des nombres entiers naturels avec zro et successeur $(\mathbb{N}, 0, s, +, =)$. La complétude de cette théorie signifie que la satisfiabilité des contraintes linéaires avec imbrication quelconque de quantificateurs est décidable sur $(\mathbb{N}, 0, s, +, =)$. La complexité en temps de ce problème est cependant rédhibitoire, doublement exponentielle dans le pire cas. L'arithmétique de Peano ajoute à cette théorie deux axiomes pour définir la multiplication. Le fameux théorème d'incomplétude de Gödel (1931) montre que cette théorie n'est pas complète et que toute extension cohérente de l'arithmétique de Peano est nécessairement incomplète. La structure des nombres entiers avec multiplication $(\mathbb{N}, 0, 1, +, *, =)$ n'est donc pas axiomatisable, et la satisfiabilité des contraintes arithmétiques entières avec quantificateurs est un problème indécidable.

De façon quelque peu surprenante, la situation est différente dans les nombres réels $(\mathbb{R}, 0, 1, +, *, =, \leqslant)$, et on pourra s'étonner du résultat de Tarski de 1948 qui montre la complétude de la théorie du corps ordonné des réels. Ce résultat montre la décidabilité de la géométrie élémentaire, et les travaux ultérieurs de Collins (1975) fournissent une procédure complète de décision de complexité doublement exponentielle en le nombre de variables. Les fragments des contraintes linéaires et des programmes linéaires (incluant la minimisation d'une fonction linéaire) ont quant à eux une complexité polynomiale et sont d'une grande importance pratique.

3 Résolution de contraintes

Dans une situation idéale en programmation par contraintes, le problème de satisfiabilité des contraintes est non seulement décidable, mais le test incrémental de satisfiabilité, qui est l'opération élémentaire effectuée à chaque étape d'exécution, a une complexité algorithmique amortie (c'est-à-dire sommée sur une exécution et divisée par le nombre d'étapes) constante ou sous-linéaire en le nombre de variables et de contraintes. En pratique, on peut aussi considérer des langages de contraintes indécidables, dès lors que l'on dispose d'algorithmes de déduction d'informations partielles de faible complexité algorithmique, qui, combinés à une procédure de recherche énumérative pour les domaines finis, ou dichotomique pour les domaines infinis, permettent de déterminer partiellement les cas d'insatisfiabilité, d'élaguer l'arbre de recherche, et de cerner l'ensemble des solutions.

3.1 Résolution complète par simplifications

Unification

Un domaine de contraintes important pour les *structures de données*, est celui de l'algèbre des termes du premier ordre, aussi appelé domaine de Herbrand. Dans l'algèbre des termes, l'égalité est l'égalité syntaxique et exclut toute autre rela-

tion d'équivalence entre termes. C'était à l'origine le seul domaine de contraintes considéré dans le langage de programmation logique Prolog, illustré dans le programme des N reines par les opérations sur les listes.

Cet exemple illustre la méthode générale de résolution de contraintes qui consiste à *normaliser* un système de contraintes par des *règles de simplification* préservant l'ensemble des solutions. Un système de contraintes est ici une conjonction d'égalités entre termes, $M_1 = N_1 \wedge ... \wedge M_n = N_n$, que l'on cherche à résoudre par substitution des variables (problème d'*unification*). La méthode consiste à définir des formes dites résolues des systèmes de contraintes, pour lesquelles la question de la satisfiabilité est triviale, et ensuite à trouver une orientation des axiomes qui permette de transformer tout système de contraintes soit en une forme résolue, soit en échec. On choisit ici comme formes résolues, les systèmes de la forme $x_1 = M_1 \wedge ... \wedge x_n = M_n$, où $n \geqslant 0$ et $\{x_1, ..., x_n\} \cap (V(M_1) \cup ... \cup V(M_n)) = \emptyset$, $V(M)$ désignant l'ensemble des variables apparaissant dans le terme M. En effet dans ces cas, la substitution des variables x_i par le terme M_i, notée $[M_i/x_i]$, fournit une solution triviale.

Le système de règles suivant a été donné quasiment sous cette forme par Herbrand dans sa thèse en 1928, puis redécouvert en 1965 par Robinson dans l'algorithme d'unification à la base de son principe de résolution en démonstration automatique.

1. $f(M_1, ..., M_n) = f(N_1, ..., N_n) \wedge \Gamma \longrightarrow M_1 = N_1 \wedge ... \wedge M_n = N_n \wedge \Gamma$,
2. $f(M_1, ..., M_n) = g(N_1, ..., N_m) \wedge \Gamma \longrightarrow faux$ si $f \neq g$,
3. $x = x \wedge \Gamma \longrightarrow \Gamma$,
4. $x = M \wedge \Gamma \longrightarrow x = M \wedge \Gamma[M/x]$ si $x \notin V(M)$, $x \in V(\Gamma)$,
5. $x = M \wedge \Gamma \longrightarrow faux$ si $x \in V(M)$, $x \neq M$.

Il est facile de vérifier que chaque règle de simplification préserve l'ensemble des solutions du système. Une fois montrée la terminaison de ce système (en montrant ici que le couple formé du nombre de variables en forme non résolue suivi du nombre de symboles dans le système de contraintes, décroît strictement dans l'ordre lexicographique, à chaque application d'une règle), il suffit de vérifier, en raisonnant par cas, que les formes irréductibles sont forcément soit $faux$ soit des formes résolues. L'application des règles de simplification fournit donc un algorithme de décision de la satisfiabilité du système de contraintes initial. Les différentes stratégies d'application des règles influent sur la complexité algorithmique. Il est possible de donner un algorithme de complexité linéaire en représentant les termes par des graphes permettant de partager les sous-termes dupliqués.

Contraintes linéaires sur $\mathbb{R}$

Les principes de résolution de contraintes par règles de simplification s'appliquent à de nombreux autres domaines. L'algorithme d'élimination de Fourier (1824) en est un autre exemple sur les nombres réels. Cet algorithme permet de décider de la satisfiabilité d'un ensemble d'inégalités linéaires sur $\mathbb{R}$, c'est-à-dire géométriquement, de la non vacuité d'un polyhèdre dans $\mathbb{R}^n$. Dans cet algorithme, on dit qu'une inégalité (ou par extension un système d'inégalités) est en *forme normale* si elle est de l'une des trois formes $t \leqslant x$, $x \leqslant t$, $t \leqslant 0$, où t est une expression linéaire et x une variable n'apparaissant pas dans t. Pour une contrainte $s \leqslant t$ on note $\overline{s \leqslant t}^x$ sa forme normale par rapport à x, que l'on obtient par simple manipulation algébrique. L'algorithme de Fourier consiste à appliquer les régles suivantes :

1. $\Gamma \longrightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} s_i \leqslant t_j \wedge \Gamma'$ si $\overline{\Gamma}^x = \bigwedge_{i=1}^{n} s_i \leqslant x \wedge \bigwedge_{j=1}^{m} x \leqslant t_j \wedge \Gamma'$ où $x \notin V(\Gamma')$,
2. $s \leqslant t \wedge \Gamma \longrightarrow \Gamma$ si $s, t \in \mathbb{R}$ et $s \leqslant t$,
3. $s \leqslant t \wedge \Gamma \longrightarrow faux$ si $s, t \in \mathbb{R}$ et $s > t$.

La première règle élimine une variable et préserve la satisfiabilité du système. Les deux autres règles préservent l'ensemble des solutions. Clairement, les règles terminent et un système est satisfiable si et seulement s'il se simplifie en le système vide. De cet algorithme général, on peut déduire d'autres algorithmes de projection des contraintes, de détection des contraintes d'égalité implicites, ainsi que plusieurs théorèmes de la programmation linéaire dont le théorème fondamental de dualité.

Contraintes booléennes

Dans le cas des contraintes booléennes, il est possible de définir des *formes canoniques*, c'est-à-dire des représentants uniques des classes d'équivalence des contraintes modulo l'équivalence logique. Les formes canoniques permettent donc de décider non seulement de la satisfiabilité des contraintes, en vérifiant que la forme canonique n'est pas $faux$, mais aussi de l'équivalence logique entre deux contraintes en testant simplement l'égalité de leurs formes canoniques. Les diagrammes de décision binaires (BDD, *Binary Decision Diagrams*) introduits par Bryant en 1985, sont des formes canoniques pour les contraintes booléennes. Il s'agit de graphes obtenus en fixant un ordre d'énumération des variables et en effectuant un partage maximum

des sous-graphes. Les sommets sont étiquetés par des contraintes booléennes, et les sommets du niveau i ont pour successeurs deux sommets correspondants à la simplification de la contrainte par la valuation $x_i = 0$ $(faux)$ et $x_i = 1$ $(vrai)$ respectivement. La racine est étiquetée par la contrainte booléenne d'origine, et les deux sommets du dernier niveau sont étiquetés par $faux$ et $vrai$. Les BDD sont très utilisés pour représenter les contraintes booléennes dans des domaines aussi variés que la conception de circuits, la vérification de programmes ou la programmation par contraintes.

3.2 Résolution incomplète par réductions de domaines

Si l'on abandonne l'idée de fournir une méthode de résolution complète pour le langage des contraintes, il demeure néanmoins possible de déduire des informations partielles sur le domaine des variables, en examinant les contraintes indépendamment les unes des autres. Cette idée, introduite en intelligence artificielle au milieu des années 70 par Mackworth notamment, a été reprise avec un grand succès en programmation par contraintes car elle s'applique à des contraintes de natures très diverses.

Notons x^d la variable x avec son domaine d. Pour définir les algorithmes de *filtrage*, aussi appelés algorithmes de *propagation* de contraintes, on introduit la notion de *projection* d'une contrainte $c(x_1^{d_1}, ..., x_n^{d_1})$ sur une variable x_1, comme étant le *domaine réduit* $r(x_1^{d_1}, c) = \{v \in d_1 \mid \mathcal{D} \models \exists v_2 \in d_2 \ \ldots \ \exists v_n \in d_n \ c(v, v_2, ..., v_n)\}$, formé de l'ensemble des valeurs de x_1 pour lesquelles la contrainte a des solutions. Cette notion s'applique à des domaines quelconques, finis, discrets ou infinis, représentés par des ensembles ou bien par des intervalles seulement.

Par exemple, pour une contrainte sur $\mathbb{N}$ de la forme : $aX^{[k,l]} \geqslant bY^{[m,n]} + d,\ a > 0,\ b > 0,\ d \geqslant 0$ où les domaines des variables sont représentés par des intervalles finis, le calcul du domaine réduit s'effectue en *temps constant* : on a $r(X^{[k,l]}, c) = [max(k, k'), l]$ et $r(Y^{[m,n]}, c) = [m, min(n, n')]$ où $k' = \lceil \frac{bm+d}{a} \rceil$ and $n' = \lfloor \frac{al-d}{b} \rfloor$.

Dans un système de contraintes, si les domaines des variables sont égaux à leur domaine réduit par rapport à toutes les contraintes, on dit que le système est *arc-cohérent* (pour des raisons historiques de référence au graphe des contraintes, où les sommets sont les variables et les arêtes les contraintes binaires, et à l'hypergraphe des contraintes dans le cas n-aire). Bien sûr, la cohérence d'arc n'implique en rien la cohérence du système de contraintes. Par exemple si $x, y, z \in \{1, 2\}$, le système $(x \neq y) \wedge (x \neq z) \wedge (y \neq z)$ est arc-cohérent et n'admet pourtant pas de solution. Cette notion permet cependant de déterminer la correction des algorithmes de filtrage, car elle définit les réductions maximum que l'on peut effectuer en raisonnant sur chaque contrainte indépendamment les unes des autres.

Les algorithmes de filtrage appliquent les régles générales de réduction de domaine suivantes :

Faux : $c \wedge \Gamma \longrightarrow faux$
si $r(x^d, c) = \emptyset$ pour un $x^d \in V(c)$;

FC : $c \wedge \Gamma \longrightarrow \Gamma[x^{d'}/x^d]$
si $V(c) = \{x^d\}$, $d' = r(x^d, c)$, $d' \neq \emptyset$;

LA : $c \wedge \Gamma \longrightarrow c[x^{d'}/x^d] \wedge \Gamma[x^{d'}/x^d]$
si $|V(c)| > 1$, $x^d \in V(c)$, $d' = r(x^d, c), d' \neq \emptyset, d' \neq d$;

PLA : $c \wedge \Gamma \longrightarrow c[x^{d'}/x^d] \wedge \Gamma[x^{d'}/x^d]$
si $|V(c)| > 1$, $x^d \in V(c)$, $r(x^d, c) \subseteq d' \subset d$, $d' \neq \emptyset$;

EL : $c \wedge \Gamma \longrightarrow \Gamma$ si $\mathcal{D} \models c\rho$ pour toute valuation ρ des variables de c.

La régle Faux détecte l'insatisfiabilité lorsqu'une variable a un domaine vide. La règle FC (*forward checking*) s'applique lorsqu'une contrainte ne contient plus qu'une seule variable. Dans ce cas la contrainte peut être éliminée en remplaçant simplement le domaine de la variable par son domaine réduit. La règle LA (*look-ahead*) applique le calcul du domaine réduit aux contraintes contenant plusieurs variables, la contrainte n'est pas éliminée. La règle PLA (*partial look-ahead*) est une variante de la règle LA qui sert simplement à distinguer les algorithmes qui calculent seulement une approximation supérieure du domaine réduit. La règle d'*élimination* EL élimine les contraintes satisfaites en effectuant un test approximatif d'implication de contraintes fondé sur le domaine des variables. Un algorithme de filtrage peut appliquer tout ou partie de ces règles à chaque modification du domaine des variables suivant plusieurs stratégies de réveil possibles (retrait d'une valeur quelconque, d'une borne uniquement, affectation, etc.).

Dans le cas très particulier d'un système de contraintes sur $\mathbb{N}$ de la forme $aX^{[k,l]} \geqslant bY^{[m,n]} + d$, $a > 0$, $b > 0$, $d \geqslant 0$, signalons un résultat de complétude : un tel système est satisfiable si et seulement s'il est arc-cohérent. Pour le voir il suffit de remarquer que dans un tel système arc-cohérent, chaque inégalité est satisfaite en prenant la plus petite valeur des variables, et que cette valuation est donc une solution du système entier. Ce résultat est important pour les problèmes d'ordonnancement. En effet une contrainte de la forme $X \geqslant Y + d$ permet de modéliser la précédence d'une tâche de durée d démarrant à la date Y sur une autre tâche démarrant en X. La complétude de l'algorithme de filtrage pour ce type de contraintes signifie

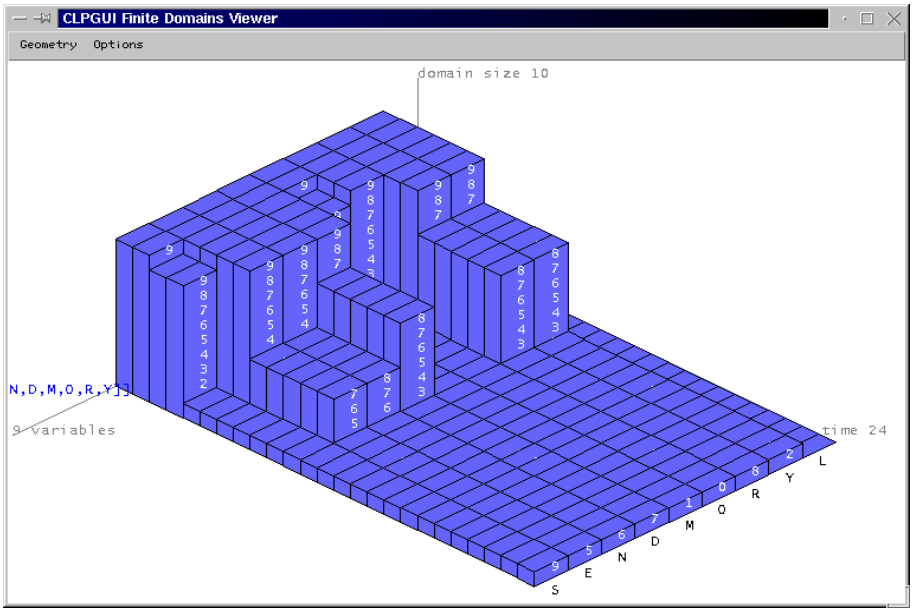


FIG. 1.2 – Effets de la pose des contraintes sur le domaine des variables dans le puzzle SEND + MORE = MONEY

qu'il est inutile d'énumérer les dates de début des tâches pour connaître les plus petites valeurs réalisables. Il s'ensuit que les temps d'exécution obtenus en programmation par contraintes sur des problèmes (NP-difficiles) d'ordonnancement sont indépendants de la discrétisation du temps (voir section 4.3).

En général, les algorithmes de filtrage ne sont pas complets et sont combinés à une procédure d'énumération des valeurs possibles pour trouver les solutions. C'est le cas dans l'exemple simple du puzzle SEND + MORE = MONEY qui consiste à trouver une valuation des lettres par des chiffres tous différents vérifiant l'équation :

```
send(L):-
     L=[S,E,N,D,M,O,R,Y], domain(L,[0,9]),
     alldifferent(L), S=/=0, M=/=0,
               1000*S+100*E+10*N+D
              +1000*M+100*O+10*R+E
     = 10000*M+1000*O+100*N+10*E+Y,
     labeling(L).
```

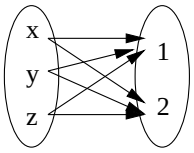
Dans ce programme, les inéquations notées `=/=`, ainsi que la contrainte `alldifferent` décomposée en inéquations, sont propagées avec la règle FC, et la contrainte linéaire avec la règle PLA (effectuant un calcul sur les bornes uniquement). Cela suffit dans cet exemple à trouver la solution de façon déterministe sans échec. La figure 1.2 montre l'effet de la pose des contraintes et de l'énumération sur le domaine des variables, qui conduisent directement à la solution.

3.3 Contraintes globales

La terminologie de contrainte globale a été introduite par opposition à la décomposition d'une contrainte n-aire, comme par exemple `alldifferent([X1,...,Xn])`, en une conjonction de contraintes binaires, $\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} Xi \neq Xj$. Une contrainte globale est donc tout simplement une contrainte n-aire, qui permet de modéliser un sous-problème d'un problème de satisfaction de contraintes, mais pour qu'elle soit intéressante, elle doit être soit indécomposable, soit pourvue d'algorithmes de résolution (complets ou incomplets) plus efficaces que ceux obtenus par décomposition.

Dans l'exemple de la contrainte ternaire `alldifferent([X,Y,Z])`, on a vu que les réductions de domaine ne permettent pas de détecter l'insatisfiabilité du système décomposé lorsque les variables ont un même domaine formé de deux éléments, alors qu'un simple raisonnement sur la cardinalité de l'ensemble des valeurs possibles permettrait de détecter l'insatisfiabilité. En 1994, Régin a proposé un algorithme de filtrage pour la contrainte `alldifferent` qui consiste à représenter la contrainte par un graphe biparti, ayant pour sommets les variables V et l'ensemble des valeurs des domaines D, et pour arêtes les as-

sociations permises par les domaines :



L'adaptation incrémentale de l'algorithme de couplage de Hopcroft et Karp de 1973 fournit alors un algorithme de filtrage de complexité linéaire en le nombre d'arêtes du graphe.

Le fort succès de la programmation par contraintes en résolution de problèmes d'optimisation combinatoire industriels, a conduit au développement de nombreuses contraintes globales depuis les années 90. On en compte aujourd'hui de l'ordre de deux cents, elles reposent sur divers *changements de représentation*, faisant souvent appel à la théorie des graphes, et à l'adaptation d'algorithmes connus, en des algorithmes *incrémentaux* de test de satisfiabilité et de réduction des domaines.

3.4 Contraintes réifiées

Les contraintes réifiées sont des contraintes d'ordre supérieur qui consistent à associer un booléen à la satisfaction d'une contrainte dans un autre domaine. Il s'agit donc de contraintes de la forme générale `B <=> contrainte`, que l'on peut voir comme des contraintes n-aires multi-domaines, et donc comme des contraintes globales particulières. Par exemple la contrainte ternaire `B <=> X<Y` exprime que le booléen B est *vrai* si et seulement si $X < Y$. Cette contrainte pose la contrainte $X < Y$ (resp. $X \geqslant Y$) si B est *vrai* (resp. $faux$) et inversement, positionne B à *vrai* (resp. $faux$) si la contrainte $X < Y$ est satisfaite (resp. insatisfiable).

Les contraintes réifiées nécessitent donc, en plus de tester l'insatisfiabilité des contraintes de base de façon à positionner le booléen à $faux$, de détecter l'*implication* des contraintes de base, de façon à positionner le booléen à *vrai* dans ce cas. Ce test d'implication de contrainte est l'opération fondamentale réalisée par les contraintes réifiées. On retrouve par ailleurs ce test d'implication de contraintes ajouté de façon systématique comme principe de synchronisation dans les langages concurrents avec contraintes (voir section 5). Le test d'implication de contraintes est un problème généralement plus difficile que le test de satisfiabilité (il est par exemple co-NP pour les clauses booléennes alors que la satisfiabilité est NP-complet). On se contente donc généralement de tests incomplets, reposant sur des conditions suffisantes sur le domaine des variables, comme dans la règle d'élimination EL de la section 3.2. Par exemple, on teste que la contrainte $X < Y$ est impliquée par les autres contraintes (resp. est insatisfiable), en testant simplement la condition sur le domaine des variables $max(X) < min(Y)$ (resp. $min(X) \geqslant max(Y)$).

Les contraintes réifiées permettent de définir très simplement, par récursion, des contraintes de cardinalité. Par exemple, la contrainte $card(N, [C1, ..., Cm])$ qui est vraie si et seulement si exactement N contraintes sont vraies dans la liste $[C1, ..., Cm]$, se définit en programmation logique (voir section 5) par récursion sur N avec un fait et une règle récursive :

```
card(0,[]).
card(N,[C|L]):- domain(B,0,1), B<=>C,
                N=B+M, card(M,L).
```

Ces contraintes offrent une puissance de modélisation très grande, qui n'a pas d'équivalent dans les outils, par ailleurs très puissants, de programmation linéaire en nombres entiers. En ordonnancement disjonctif par exemple, une contrainte d'exclusion mutuelle entre deux tâches, l'une de durée dx démarrant à la date X, l'autre de durée dy démarrant en Y, peut s'exprimer par la contrainte `card(1,[X>=Y+dy,Y>=X+dx])` dont l'effet est d'automatiquement poser la bonne contrainte de précédence dès que la contrainte alternative devient insatisfiable.

4 Procédures de recherche

4.1 Arbre de recherche

Un arbre de recherche est un arbre dans lequel chaque noeud est étiqueté par une conjonction de buts (*parallélisme "et"*) dont un sous-but est distingué. Chaque noeud a autant de successeurs qu'il y a de façons de transformer le sous-but distingué. Chaque successeur est étiqueté par la conjonction de buts obtenue en appliquant une règle de transformation au sous-but sélectionné. Les successeurs d'un noeud représentent les différentes façons de transformer un but et représentent donc une disjonction (*parallélisme "ou"*).

En programmation par contraintes, l'arbre de recherche permet de traiter les *buts disjonctifs*, chaque branche de l'arbre correspondant au calcul déterministe des contraintes accumulées sur les nœuds. Ainsi les contraintes d'appartenance à un domaine fini peuvent être traitées comme des disjonctions avec le système non déterministe d'énumération suivant :

$$x \in \{v_1, ..., v_n\} \longrightarrow x = v_1$$

$$...$$

$$x \in \{v_1, ..., v_n\} \longrightarrow x = v_n$$

Les contraintes d'intervalles peuvent être traitées avec le système de recherche dichotomique :

$$x \in [v_1, v_2] \longrightarrow x \in [v_1, (v_1 + v_2)/2]$$

$$x \in [v_1, v_2] \longrightarrow x \in](v_1 + v_2)/2, v_2]$$

Plus généralement des contraintes disjonctives quelconques se traitent avec le système non déterministe :

$$c_1 \vee c_2 \longrightarrow c_1$$

$$c_1 \vee c_2 \longrightarrow c_2$$

Ce traitement des contraintes disjonctives permet par exemple d'énumérer tous les choix d'ordonnancement des contraintes d'exclusion mutuelle (en complément de leur expression éventuelle par la contrainte de cardinalité décrite dans la section 3.4). De façon générale, on modélise très facilement des algorithmes de complexité NP par l'exploration d'un arbre de recherche avec un système de contraintes de complexité polynomiale.

L'exploration de l'arbre de recherche associé aux domaines des variables permet de pallier l'incomplétude des algorithmes de filtrage des contraintes en effectuant une énumération des valeurs possibles des variables. Dans ce cas, les contraintes sont utilisées de façon *active* pour détecter en chaque noeud de l'arbre les cas d'insatisfiabilité, et donc élaguer les branches de l'arbre de recherche le plus tôt possible avant énumération complète.

Dans l'exemple des N reines donné en introduction, l'arbre de recherche est formé par l'énumération des variables de domaines finis. Les contraintes d'inéquations sont propagées simplement avec la règle FC en éliminant la valeur correspondante à chaque instanciation d'une variable. La figure 1.1 montre l'effet de ces contraintes sur l'arbre de recherche. On remarque en particulier que les branches qui ne conduisent pas à une solution sont stoppées relativement tôt dans l'arbre.

4.2 Heuristiques

Les heuristiques sont des guides pour effectuer des choix, et sont d'une grande importance dans les résultats obtenus aujourd'hui en programmation par contraintes. Dans les arbres de recherche, aux deux formes de parallélisme "et" et "ou", correspondent deux principes heuristiques qu'il convient de distinguer.

En parallélisme "et", le choix du sous-but à résoudre, par exemple le choix de la variable à instancier d'abord, ou le choix de la contrainte disjonctive à traiter en premier, consiste généralement à suivre une heuristique d'*échec d'abord* (*first fail*). Cette heuristique consiste à choisir en chaque nœud le sous-but le plus difficile à résoudre, celui qui apporte le plus d'information, risque le plus de conduire à un échec et donc à s'arrêter plus tôt dans l'exploration, et limiter ainsi le nombre de sous-arbres développés. Pour le choix d'une variable à instancier, cela peut consister à choisir la variable de plus petit domaine, ou la variable la plus contrainte par exemple. De façon générale, le choix d'un sous-but ayant le plus petit nombre d'alternatives conduit à minimiser la taille de l'arbre de recherche. En particulier, il est important de détecter le cas extrême des buts devenus déterministes pour les traiter en premier, comme cela a été mentionné a propos des contraintes d'exclusion mutuelle avec la contrainte de cardinalité.

En parallélisme "ou", le choix de la transformation à essayer d'abord, par exemple le choix de la valeur à essayer en premier pour une variable, ou le choix de la précédence à essayer en premier dans une contrainte d'exclusion mutuelle, se fait généralement suivant l'heuristique du *meilleur d'abord* (*best-first*). L'objectif est de trouver en priorité une solution, ou une bonne solution dans un problème d'optimisation.

Il est important de noter que les heuristiques de parallélisme "et" influent sur la *forme de l'arbre* de recherche et ont donc une influence directe sur les performances de la recherche des solutions et de la preuve d'optimalité. En revanche, les heuristiques de parallélisme "ou" influent uniquement sur l'*ordre d'exploration* de l'arbre de recherche. Elles influent donc sur le temps de recherche et la qualité des premières solutions trouvées, mais n'ont aucun effet sur l'énumération de toutes les solutions, ni sur les preuves d'insatisfiabilité tout comme sur les preuves d'optimalité.

La notion d'heuristique est souvent associée à tort à celle d'algorithme incomplet. Dans le cas d'un algorithme "glouton", une heuristique est utilisée pour effectuer des choix à chaque étape de façon irrévocable. Un tel algorithme est bien sûr incomplet, il correspond à l'exploration de la seule branche la plus à gauche de l'arbre de recherche heuristique. En programmation par contraintes, les choix heuristiques n'affectent en rien la complétude de la procédure de recherche qui peut explorer complètement ou non l'arbre de recherche quelles que soient les heuristiques.

À cet égard, les heuristiques de parallélisme "ou" peuvent être utilisées pour explorer l'arbre de recherche de façon incomplète avec une bonne répartition des différentes branches explorées. L'idée de la recherche à écarts limités est d'autoriser un nombre maximum de fois où l'heuristique de parallélisme "ou" n'est pas suivie le long d'une branche. L'effet de la recherche à écarts limités est d'explorer une partie seulement de l'en-

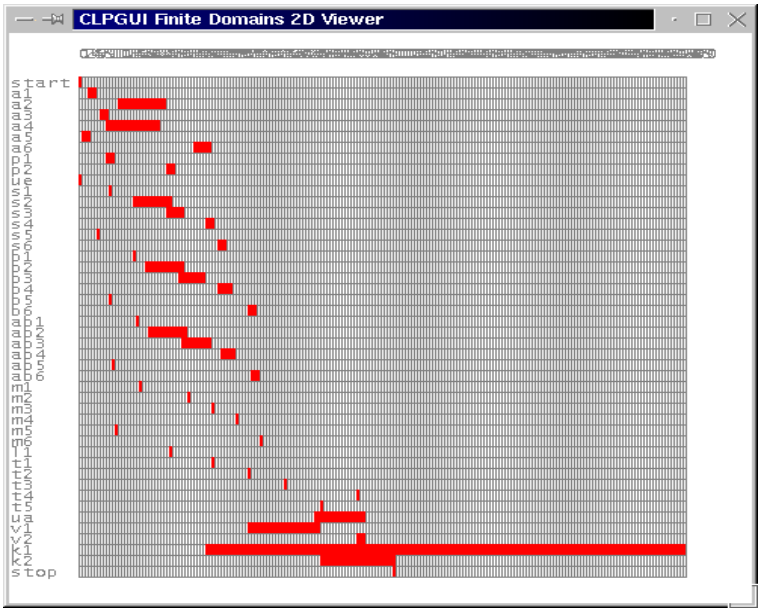


FIG. 1.3 – Dates possibles de début des tâches dans une solution prouvée optimale du problème d'ordonnancement disjonctif du pont

semble des branches de l'arbre mais avec une bonne répartition dans les différents sous-arbres.

4.3 Optimisation par séparation-évaluation

Dans un problème d'optimisation, on ne cherche pas seulement à satisfaire des contraintes, mais à trouver des solutions qui minimisent (ou maximisent) la valeur d'une fonction objectif F. La procédure générale de séparation-évaluation (*branch-and-bound*) se prête parfaitement bien à la programmation par contraintes. Elle consiste à chercher une première solution en parcourant l'arbre de recherche, à évaluer son coût v donné par la fonction objectif, et à continuer (ou à redémarrer) la recherche en ajoutant la contrainte $F < v$. La contrainte de coût $F < v$ est propagée comme n'importe quelle autre contrainte pour élaguer l'arbre de recherche en chaque nœud. L'échec à trouver de nouvelles solutions en explorant l'arbre de recherche constitue la *preuve d'optimalité* du dernier coût trouvé. L'ensemble des solutions optimales peut être énuméré, si besoin, en relançant une recherche avec le coût fixé à sa valeur optimale.

Dans un problème d'ordonnancement disjonctif on cherche à planifier des tâches de façon à terminer la dernière tâche du projet au plus tôt, ceci en présence de contraintes de dates possibles de démarrage de chaque tâche, de contraintes de précédence entre tâches, et de contraintes d'exclusion mutuelle qui rendent le problème NP-difficile. Si l'on modélise les précédences par des contraintes de la forme $X \geqslant Y + dy$ et les contraintes d'exclusion mutuelle par des disjonctions $X \geqslant Y + dy \vee Y \geqslant X + dx$ traitées par des points de choix dans l'arbre de recherche, les contraintes accumulées le long d'une branche demeurent de cette forme simple et la méthode de réduction des domaines est complète (voir section 3.2). Il n'est donc pas nécessaire d'énumérer les valeurs des dates pour garantir que la plus petite valeur du domaine de toute variable est réalisable. En particulier le coût d'une solution trouvée en une feuille de l'arbre de recherche est donné par la plus petite valeur du domaine de la variable de fin du projet, qui constitue la fonction objectif. Une solution prouvée optimale peut ainsi contenir des dates flexibles pour certaines tâches ne se trouvant pas sur un chemin critique dans le graphe de précédence. Une telle solution est visualisée dans la figure 1.3, d'autres solutions optimales existent correspondant à d'autres choix de précédences dans les contraintes disjonctives. La figure 1.4 montre l'arbre de recherche qui est exploré pour ce problème. Elle est typique d'une procédure d'optimisation par séparation-évaluation avec redémarrage : les solutions trouvées sont successivement de coût 110, 106, 104, puis un arbre d'échec est développé montrant ainsi l'optimalité du coût 104 pour la

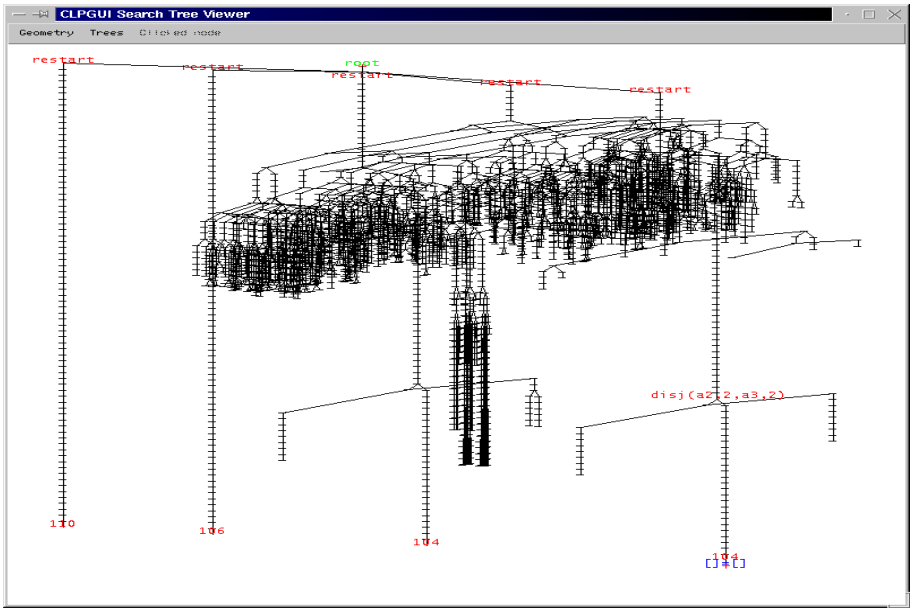


FIG. 1.4 – Vue de l'arbre de recherche exploré dans le problème d'optimisation du pont

fonction objectif. Dans cet exemple la solution optimale est recalculée ensuite en fixant le coût égal à 104, de façon à énumérer éventuellement toutes les solutions optimales. Ce petit exemple contient 44 variables et est résolu en 10ms, preuve d'optimalité comprise. Des exemples d'ordonnancement proposés comme challenges en recherche opérationnelle ont été résolus en programmation par contraintes par ce type de méthode, augmentée de contraintes globales pour les ensembles de tâches en exclusion mutuelle, ainsi que d'heuristiques de recherche pouvant consacrer un temps substantiel à déterminer la bonne disjonction à explorer en premier.

4.4 Recherche locale

Les procédures de recherche locale ne font pas partie à proprement parler de la programmation par contraintes, car elles ne travaillent pas sur des structures d'information partielle (variables) mais sur des valuations (valeurs fixées). De plus elles ne permettent jamais de prouver l'insatisfiabilité d'un système de contraintes, ni l'optimalité d'une solution. Cependant les procédures de recherche locale peuvent s'appliquer à des problèmes de satisfaction de contraintes, avec parfois une efficacité redoutable sur des problèmes de très grande taille, en calculant des solutions en fait quasi-optimales. Ces procédures permettent de plus de traiter des problèmes *surcontraints*, ou contenant des contraintes de *préférence* qui peuvent être insatisfaites, en calculant des valuations qui maximisent le nombre de contraintes satisfaites.

Le principe d'une procédure de recherche locale est de modifier itérativement une valuation initiale, qui peut être aléatoire, de façon à se rapprocher d'une solution, tout en combinant un mécanisme d'*intensification*, d'amélioration continue de la valuation, à un mécanisme de *diversification*, permettant d'échapper aux minima locaux. À chaque étape de la phase d'intensification, la procédure de recherche locale calcule le voisinage de la valuation courante, c'est-à-dire l'ensemble des valuations atteignables à partir de la valuation courante en un mouvement, de façon à choisir la meilleure au sens de la satisfaction des contraintes, et à itérer la recherche à partir de cette nouvelle valuation. L'ordre de comparaison des valuations vis-à-vis de la satisfaction des contraintes ne consiste en général pas simplement à compter le nombre de contraintes satisfaites, car ceci ne permettrait pas de guider finement la recherche, mais à sommer des fonctions d'erreur attachées à chaque contrainte. Les fonctions d'erreur doivent prendre leurs valeurs dans un intervalle normalisé $[0, Emax]$, où la valeur $Emax$ représente l'erreur maximum, et la valeur 0 représente la satisfaction de la contrainte. Par exemple, pour la contrainte $x \geqslant y + 5$, on pourra prendre comme fonction d'erreur avant normalisation $max(y + 5 - x, 0)$. Dans les problèmes de satisfaction de contraintes, on peut utiliser la structuration du problème en le

couple variables-contraintes pour, à chaque étape, focaliser l'intensification sur la variable qui a le plus grand nombre de contraintes insatisfaites.

Le mécanisme de diversification peut consister simplement à repartir d'une nouvelle configuration aléatoire lorsqu'un minimum local est atteint, c'est-à-dire lorsque plus aucune modification ne permet d'améliorer la configuration courante. On peut également s'autoriser à dégrader la qualité de la configuration courante à un certain degré, qui est par exemple une fonction décroissante du temps dans les procédures de recuit simulé. Dans les procédures de recherche Tabu, une liste de variables est gelée pendant un certain nombre de mouvements dès lors qu'un minimum local a été atteint pour cette variable. Dans les algorithmes génétiques, on fait évoluer une population de configurations, auxquelles on applique des opérations de croisement. Ces différentes stratégies de diversification s'appellent des méta-heuristiques, car elles permettent de définir différentes familles de procédures de recherche locale.

Les méthodes de recherche locale peuvent permettre de trouver des solutions, ou des solutions approchées, à des problèmes de très grande taille, pour lesquels les autres méthodes de résolution échouent. Elles permettent de plus de traiter des contraintes très variées puisqu'il suffit de définir une fonction d'erreur pour chaque contrainte. Cependant ces méthodes trouvent rapidement leurs limites sur les problèmes combinant des contraintes hétérogènes, car il devient alors difficile de trouver des fonctions d'erreur équitables pour les différentes contraintes, or le moindre biais a pour effet d'égarer la recherche des meilleures solutions.

5 Modèles d'exécution

On peut distinguer trois grands modèles d'exécution de la programmation par contraintes. Le premier niveau consiste à résoudre un problème de satisfaction de contraintes (CSP, *Constraint Satisfaction Problem*), défini par la donnée d'un ensemble de contraintes entièrement fournies en extension au départ. C'est le cas par exemple du puzzle $SEND + MORE = MONEY$ ou d'un programme linéaire : $Ax \leqslant b, max\ cx$.

Le second niveau est celui des langages de programmation logique avec contraintes (CLP, *Constraint Logic Programs*). Ces langages introduisent des définitions inductives de relations. Ces définitions peuvent être utilisées pour, dans le cas déterministe, poser les contraintes du problème, exprimées en intention, ou, dans le cas non déterministe, définir les points de choix et la procédure de recherche. Les programmes donnés en exemple dans les sections précédentes sont dans cette classe. Le programme CLP($\mathbb{R}$) suivant exprime la formule des intérêts composés par récursion sur le premier argument du prédicat *int* qui lie le nombre de mois T et le montant du capital emprunté E au taux d'interêt I et aux remboursements mensuels M :

```
int(T,E,I,M):- T>0, T<=1, M=E*(1+I).
int(T,E,I,M):- T>1, int(T-1,E*(1+I)-M,I,M).
? int(120,120000,0.01,M).
M=1721.651381
? int(120,E,0.01,1721.651381).
E=120000
? int(120,E,0.01,M).
E=69.700522*M
? int(3,999,Int,400).
400=(-400 +(599+999*Int)*(1+Int))*(1+Int)
```

Les réponses calculées ne sont pas nécessairement des valuations comme dans les deux premières requêtes, mais plus généralement des contraintes, accumulées le long de la branche d'exécution et projetées sur les variables du but initial. Dans le dernier exemple, la réponse est une contrainte non linéaire dont la satisfiabilité n'a pas été testée par l'interpréteur. Ce modèle d'exécution correspond essentiellement à celui que l'on trouve (avec une syntaxe différente) dans les bibliothèques de programmation par contraintes en C, C++, Java etc.

Le troisième niveau est celui des langages concurrents avec contraintes (CC, *Concurrent Constraint languages*), introduits par V. Saraswat en 1989, suivant un modèle d'exécution concurrent utilisant le test d'implication de contraintes comme mécanisme de synchronisation. Ces langages permettent d'une part, de programmer certains algorithmes de résolution de contraintes, et d'autre part, d'exprimer par des contraintes le contrôle de l'exécution du programme. Si l'on conserve la syntaxe des programmes CLP, la seule opération ajoutée est celle, notée `ask(c,G)`, de synchronisation d'un sous-but G sur la satisfaction de la contrainte de garde c. Tant que la contrainte c n'est pas impliquée par l'ensemble des contraintes accumulées, l'exécution du but G est suspendue. Lorsque la contrainte c est satisfaite, le but G est ajouté au but courant. Dans les branches d'exécution qui terminent, on distingue alors les succès, qui terminent avec une contrainte comme réponse, des suspensions qui terminent avec des buts *ask*.

Dans l'exemple CLP($\mathbb{R}$) précédent, une contrainte non linéaire c reste inactive jusqu'à ce qu'elle devienne linéaire. Cela peut être modélisé par un but `ask(l,c)` où l est une contrainte exprimant la condition de linéarité pour c. Les algorithmes simples de réduction de domaines de la section 3.2 se prêtent également bien à une modélisation des propagateurs de contraintes par

des buts de la forme `ask(c,R)`, où c est une condition indiquant la modification du domaine d'une variable et R est le but effectuant le calcul des domaines réduits. Les contraintes réifiées de la section 3.4 peuvent être définies par des programmes CC de la même manière. Enfin, les langages CC permettent d'exprimer simplement la combinaison et la synchronisation de plusieurs modélisations d'un même problème combinatoire, ainsi que la coopération de différentes méthodes de résolution.

Bibliographie

Apt K., (2003), *Principles of Constraint Programming*, Cambridge University Press.

Colmerauer A., (1984), Equations and Inequations on Finite and Infinite Trees, in Institute for New Generation Computer Technology, éditeur, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo.

Fages F., (1996), *Programmation logique par contraintes*. Collection cours de l'Ecole Polytechnique, Ellipses, Paris.

Herbrand J., (1968), Recherches sur la théorie de la démonstration. Thèse de l'Université de Paris, (1928). Dans *Écrits logiques*, Presses Universitaires de France, Paris.

Huet G., (1972), Constrained resolution : a complete method for higher-order logic. Ph.D. thesis, Case Western Reserve Univ., USA.

Jaffar J. et Lassez J.L., (1987), Constraint logic programming, *ACM Symp. Principles of Programming Languages*, ACM, pages 111–119.

Kowalski R., (1979), Algorithm = Logic + Control. *Communications of the ACM* **22** (7), pages 424–436.

Mackworth A.K., (1977), Consistency in networks of relations. *Artificial Intelligence* **8** (1), pages 99–118.

Régin J.C., (1994), A Filtering Algorithm for Constraints of Difference in CSPs. *Actes de AAAI'94*, pages 362–367. AAAI Press.

Saraswat V., (1993), *Concurrent constraint programming*. MIT Press.

Van Hentenryck P., (1999), *The OPL Optimization Programming Language*. MIT Press.

Van Hentenryck P., et Michel L., (2005), *Constraint-based Local Search*. MIT Press.

Index