Programmation de contraintes
ou programmation automatique ?

Constraint propagation or automatic programming?

Jean-Louis Laurière

Résumé : Après avoir effectué un bilan sur les problèmes NP-complets et les résolveurs de problèmes disponibles, nous nous intéressons aux cas où le schéma propagation+choix n'est pas efficace.
Soit parce que le système ne trouve pas les symétries de l'énoncé et répète inlassablement l'étude de situations équivalentes, ou bien parce qu'il a déterminé une solution quasi-optimale mais perd son temps sur la preuve de l'optimalité ou encore parce que, après avoir effectivement pris en compte la plupart des contraintes, il est devant un problème très peu contraint et perd son temps à propager des informations sans intérêt.
Nous pensons que dans ces conditions la solution est la programmation automatique.
Des exemples concrets sont explicités. Le nouveau logiciel RABBIT est présenté. Il est effectivement capable de générer des programmes de plusieurs milliers d'instructions qui permettent de conclure en un temps CPU jusqu'à cent fois inférieur à un pur CSP.

Mots-Clés : Résolution de C.S.P., optimalité, programmation automatique, propagation.

Abstract : After an initial assessment of NP-complete problems and existing problem solvers, the author goes on to consider all the cases where a propagation + choice approach is not efficient. The system is incapable of finding the symmetries of the problem and either it endlessly studies equivalent situations or it determines a quasi-optimal solution and then wastes time proving this optimality, or else it takes most of the constraints into account but then, when having to deal with an under-constrained problem, wastes time propagating useless information.
To overcome these problems,, automatic programming would seem to be an ideal solution for the author.
Concrete examples are given and RABBIT, a new software deriving from ALICE, is described. This software can generate programs containing thousands of instructions which can be run up to one hundred times faster than a pure CSP.

Key words : Propagation, optimality, automatic programming.

Présentation du rapport

Le système ALICE date d'il y a vingt ans et n'a toujours rien perdu de son actualité Nous rencontrons dans la littérature récente nombre de papiers qui sont en train d'en redécouvrir petit à petit les principes sans même s'en rendre compte ; ces systèmes sont loin d'avoir la puissance d'ALICE. Cela vient de ce que ce système est si original que le lecteur, gêné par ce qu'il sait déjà, n'arrive pas à concevoir qu'un système fonctionne de façon si différente de ce qu'il connaît. Aussi, quand j'ai appris que Jean-Louis Laurière avait apporté des améliorations importantes à son système, j'ai pensé qu'il était essentiel que la communauté scientifique puisse en prendre connaissance. C'est pourquoi cette description rapide des dernières améliorations d'ALICE est publiée sous forme de rapport du Laforia.

Les deux ouvrages les plus connus de Lewis Carroll, *Alice au pays des merveilles* et *A travers le miroir*, ont fait les délices de nombreux lecteurs. Pourtant bien des beautés de ces livres leur ont échappé ; en effet, Lewis Carroll, qui a été aussi un éminent logicien, était un esprit brillant, mais il ne se rendait pas toujours compte que ce qui était pour lui une évidence n'était pas toujours accessible à ses lecteurs. C'est une des raisons pour lesquelles, un siècle après la parution de ces ouvrages, Martin Gardner, qui a longtemps tenu la rubrique de récréations mathématiques du Scientific American, nous a donné *The Annotated Alice*. De nombreuses notes accompagent le texte original et nous permettent de mieux apprécier les deux ouvrages qu'elles complètent.

Les textes existant sur ALICE, système d'IA, sont eux aussi parfois difficiles d'accès pour le lecteur qui n'a pas expérimenté le système et qui a de la peine à en comprendre toute l'originalité. C'est pourquoi je participe en tant que commentateur à un nouvel "Annotated Alice" pour faciliter la compréhension de cette version complémentaire du système ALICE. Pour bien distinguer dans ce qui suit mes commentaires du texte de J.-L. Laurière, tout ce que j'ai inséré est en corps 10 alors que le texte original est en corps 12. C'est aussi la solution choisie par Gardner qui écrit ses notes dans un corps plus petit que celui dans lequel est composé le texte de Carroll.

Ce papier n'a pas été conçu pour être autonome ; il présuppose que le lecteur a déjà connaissance du système ALICE. Mes commentaires supposent également une connaissance préalable d'ALICE. Plusieurs textes donnent une description complète de ce système ou précisent certaines de ses caractéristiques :

J.-L. Laurière, Un langage et un programme pour énoncer et résoudre des problèmes combinatoires, Thèse de l'Université Paris 6, 1976.

Ce texte contient une description détaillée du système et la présentation, souvent succinte, des problèmes résolus par ALICE.

J.-L. Laurière, A language and a program for stating and solving combinatorial problems, Artificial Intelligence 10, 1978, 29-127.

Ce texte donne une description complète du système avec certains exemples plus détaillés que dans la thèse.

J.-L. Laurière, Toward efficiency through generality, Proceedings of the sixth IJCAI, 1979, 619-621.

Ce papier explique les raisons du succès d'ALICE qui, bien que général, a obtenu pour certains problèmes des résultats meilleurs que des programmes spécifiquement écrits pour un de ces problèmes.

J.-L. Laurière, Intelligence artificielle, résolution de problèmes par l'homme et la machine, Eyrolles, 1986.

Le chapitre 8 de ce livre donne une excellente description d'ALICE, moins complète que les précédentes pour ce qui est des exemples, mais plus accessible pour le lecteur pressé.

J. Pitrat, Penser autrement l'informatique, Hermès, 1993.

ALICE est un système qui fonctionne et il est parfaitement possible d'en reproduire les qualités. Cela vaut la peine d'être remarqué, car il n'existe pas beaucoup de systèmes d'IA (en dehors de ceux qui font systématiquement appel au combinatoire) que l'on a pu refaire en retrouvant les performances du système original. Dans le cadre du système MACISTE, j'ai incorporé les métaconnaissances qui sont à la base d'ALICE et j'ai obtenu des résultats comparables. Le chapitre 12 décrit les caractéristiques de cette nouvelle implémentation.

Jacques Pitrat

Le dernier précepte est de faire partout des dénombrements si entiers et des revues si générales, que je sois assuré de ne rien omettre.

Descartes, *Discours de la méthode II.21.*

Plusieurs logiciels résolvant, par propagation de contraintes, des problèmes industriels de type combinatoire, (ordonnancements, découpe de bois ou de verre, architecture, construction de plans d'expériences, planification optimale d'investissements, horaires de personnes (public ou privé), génération de jeux de clés multi-fonctions, câblage de circuits imprimés, conception de puces, rotations des équipages dans les entreprises de transports. chargements de conteneurs, organisation des stationnements d'avions) sont aujourd'hui sur le marché : CHIP, PROLOG III, PECOS, THINKLAB, ConstraintLisp, cc(FD), CLP, MULTI-TAC Compiler, parmi d'autres.

Dans le même temps, paraissent des articles dans les revues de recherche pour parler de puzzles comme SEND+MORE= MONEY (Puget 1993) ou d'autres tout aussi simples (GPS a été fait pourtant en 1958...) et, si de nombreux papiers théoriques sur les méthodes de consistance ou les variantes du backtrack programming, peu nombreux, me semble-t-il, sont ceux qui décrivent les solutions de problèmes concrets.

RABBIT, fils d'ALICE (Laurière 1978) est un logiciel qui s'efforce de s'attaquer à de "vrais" problèmes issus du monde réel.

Il utilise :

– la propagation de contraintes qui permet par calcul formel de réduire l'espace de recherche.

– des heuristiques de choix (choisies automatiquement) qui limitent, par ailleurs, la taille de l'arbre des essais.

Toutefois, le caractère NP-complet de ces problèmes ne disparaît pas pour autant et de nombreux énoncés restent revêches à toute résolution par ce type de méthodes.

RABBIT fait allusion au lapin qui apparaît au début de *Alice aux pays des merveilles*. Les aventures d'Alice commencent quand elle le suit dans son terrier.

Aussi RABBIT utilise-t-il pour tous les cas où le schéma **"propagation + choix" n'est pas efficace**, une approche qui donne localement des résultats bien meilleurs que la pure propagation.

Dans tous les cas où la pure propagation se révèle inefficace, deux caractéristiques de l'état du problème sont présentes :

1) La résolution est bien avancée et le problème notablement réduit (grosso modo, plus la taille du problème est importante plus il faut faire preuve d'intelligence et donc propager, alors que plus elle devient faible, plus le combinatoire stupide est approprié).

2) La propagation mène souvent à résoudre les mêmes problèmes résiduels.

Si le système est capable de détecter une telle situation, la meilleure solution pour lui est alors d'engendrer un programme spécifique énumératif par compilation des contraintes.

L'expérience montre que les programmes ainsi générés, qui comportent de mille à dix mille instructions, permettent souvent de conclure en un temps jusqu'à cent fois inférieur à un CSP classique.

<u>PLAN</u>

I. PHILOSOPHIES DE RESOLUTION DES C.S.P.

Historiquement la résolution de problèmes concrets rencontrés dans de très nombreuses entreprises en planification et optimisation a d'abord été le sujet de prédilection des **Graphes** et de **la Recherche Opérationnelle** : méthode PERT, diagramme de Gantt, algorithme hongrois, plus courts chemins, couplages, flots et réseaux de transports, méthode de Little.

Une des approches les plus efficaces a longtemps consisté à transformer le problème de départ en introduisant force variables imaginaires pour rendre le problème linéaire et le traiter par la célèbre méthode du *Simplexe*. La solution n'étant pas, en général, constituée d'entiers, les études sur les contraintes diophantiennes et les troncatures fleurirent (R. Gomory 1975, E. Johnson 1981). Dans les années soixante-dix, ce type d'approche fut violemment critiqué (cf par exemple Gondran 74). Il ne permettait, en effet et au mieux, que de résoudre des problèmes de dimension moyenne et le modèle lui-même explosait en taille des données (matrice de milliards d'éléments) avant toute résolution, pour des cas réels.

Les chercheurs en **Intelligence Artificielle**, sans méthodes et sans outils adaptés, se sont ensuite attaqués à cette question à coup d'arborescences et d'heuristiques. (Colmerauer 71, Selz 76, Lemaire 77, Slabodsky 78, Dincbas 79). Si ces programmes résolvent effectivement des problèmes de taille importante, ils sont très difficiles à juger car les problèmes traités sont tous artificiels et chaque auteur repart de zéro.

Les choses ont bien évoluées aujourd'hui et l'apport de l'Intelligence Artificielle nous paraît net sur les quatre points suivants :

1– ***Traitement formel*** des contraintes en les combinant entre elles.

2– ***Algorithmes de consistance*** pour réduire à tout instant les domaines au plus serré.

3– Gestion de l'arbre de recherche à l'aide d'heuristiques externes au programme proprement dit, mises ***sous forme déclaratives de règles elles-mêmes contrôlées par des méta-règles déclaratives*** (J Pitrat 86, 78, 90)

4– ***Fonctions d'évaluation*** inspirées de la programmation des jeux (Algorithme B* de H. Berliner 84) pour encadrer la valeur des solutions en toute feuille de l'arbre.

Le résultat fondamental sur les problèmes qui nous intéressent (C.S.P.) est par ailleurs dû à Stockmeyer (1973) ; cette classe de problèmes se divise en deux familles :

- **celle des problèmes polynômiaux** (pour lesquels un algorithme est connu qui s'exécute en un temps borné par un polynôme fonction de la taille de l'énoncé) ;

- **les autres** qui sont, de loin, les plus nombreux. Pour résoudre ces derniers, des choix sont absolument indispensables : ces problèmes sont dits Non-déterministes (le N de NP). Il a pu être mathématiquement prouvé (Karp 1972) que, parmi ceux-ci, il existait une vaste famille de problèmes équivalents : d'une part, ils peuvent tous être résolus en temps polynômial mais par un méthode non-déterministe (l'appelation correcte est NDP et non NP).

Le <u>point clé</u> est le suivant :

– Si **la théorie dit que les problèmes NP-complets sont difficiles,** c'est à propos des problèmes au sens mathématique général, c'est à dire quelles que soient les données.

– **Dans la réalité, on rencontre des instances de ces problèmes.**

Ce sont, par construction, des cas particuliers du problème général et, dès lors, rien n'interdit d'obtenir une solution, sur ces données particulières, avec un minimum de choix, et éventuellement aucun par exemple : SEND + MORE = MONEY.

Pour la solution ALICE de cette cryptaddition, voir les pages 80 et 81 de l'article du AI Journal.

La propagation de contraintes prend alors sa pleine valeur (en laissant loin derrière la Recherche Opérationnelle, qui ne traite que la partie numérique des problèmes).

PROPAGER LES CONTRAINTES signifie pour RABBIT que le système cherche à réduire l'espace de recherche à l'aide de calcul formel sur ces contraintes (en particulier, via ***l'algorithme d'unification*** (Pitrat , Robinson) :

– **filtrage** par tests de compatibilité et consistance (arcs, chemins, inter-, k-),

– **algorithmes de chemins** et étude de graphes,

– **réécriture** pour simplifier et normaliser,

– **algèbre d'intervalles,**

– **arithmétique,**

– **agrégation de contraintes,**

mais également, ***construction automatique de programmes étroitement adaptés au stade précis de la résolution.***

Nous donnons d'abord deux exemples caractéristiques de résolution de tels problèmes NP-complets.

Le premier porte sur des variables entières.

Le second porte sur des variables continues et des fonctions trigonométriques. RABBIT n'est en rien gêné par les domaines continus et parvient ici à la solution exacte alors que l'auteur du problème n'obtient, par des méthodes numériques d'approximation d'intervalles, qu'une solution approchée (L'homme 94).

RABBIT contient deux améliorations essentielles par rapport à ALICE. La première lui donne une plus grande puissance en calcul algébrique et la possibilité de considérer des variables réelles dont la valeur appartient à un intervalle continu. La deuxième le transforme en informaticien en lui donnant la possibilité d'écrire, puis d'exécuter des programmes au cours d'une résolution. Les deux exemples qui suivent ne font intervenir que les améliorations en calcul formel. Les exemples où l'on utilise la création de programmes intermédiaires seront présentés dans la section V.

<u>Exemple 1 :</u>

Le problème suivant fut posé par l'Académie des Sciences à Paris en 1840, au calculateur prodige Henri Mondeux alors âgé de 24 ans :

Trouver les solutions entières positives de l'équation :

$$\boxed{(E) \qquad x^3 + 119 = 66 * x}$$

Le résolveur dispose d'une *famille de méthodes* pour extraire de cette équation des *relations plus simples*. Il trouve ainsi, d'abord, un majorant pour x en réécrivant (E) sous la forme :

$$x^3 < 66 * x$$

donc $\quad x^2 < 66 \quad$ d'où : $x \leq 8$

Un raisonnement par congruences, à partir de la factorisation de 119 = 7*17 fournit par ailleurs :

$$x^3 \equiv 15 * x \pmod{17}$$

puisque 66 = (3*17) + 15

soit $\quad x^2 \equiv 15 \pmod{17}$

car x = 17 n'est pas admissible.

RABBIT applique la règle de simplification : si on a la contrainte a*b ≡ 0 (mod k) et si k est premier, alors on a la nouvelle contrainte : a ≡ 0 (mod k) OU b ≡ 0 (mod k). Cette règle de simplification n'existait pas dans ALICE initial ; comme plusieurs autres, c'est une amélioration de RABBIT. Notons qu'il est très facile d'ajouter de nouvelles règles, puisqu'elles sont traitées à l'aide de l'algorithme d'unification : il suffit de les ajouter formellement. La section VIII.5.1.2 du livre [Laurière 86] contient l'énoncé des règles de simplification d'ALICE.

Suivant les méthodes d'ALICE, le système peut traiter des contraintes conjonctives efficacement. En effet, quand le système décide de faire un choix, ce choix n'est pas uniquement de considérer successivement toutes les valeurs possibles d'une variable V. Il peut prendre une contrainte disjonctive et considérer autant de branches qu'il y a d'éléments dans la disjonction ; le problème du zèbre [Laurière 76 et 78] en donne un bon exemple. Si l'on a la contrainte A OU B OU C, on enlèvera cette contrainte et on considèrera la contrainte A dans la première branche, puis les contraintes B et ¬A dans la deuxième branche et enfin les trois contraintes C, ¬A et ¬B dans la troisième branche. On ajoute les négations des contraintes déjà envisagées pour ne pas considérer deux fois la même solution si elle satisfait à la fois les contraintes A et B par exemple. Dans le cas précédent, comme $x*(x^2–15) \equiv 0$ (mod 17), on envisagera d'abord la contrainte $x \equiv 0$ (mod 17) qui donne immédiatement lieu à une contradiction puisqu'aucune des huit valeurs possibles de x n'est divisible par 17, puis les contraintes $x^2–15 \equiv 0$ (mod 17) et $\neg\, x \equiv 0$ (mod 17) ; cette dernière contrainte est immédiatement éliminée car elle est vraie pour toutes les valeurs possibles de x. Nous reverrons ce traitement du OU dans l'exemple suivant.

Les solutions vérifient donc :

$$x^2 = 15 + k*17$$

avec x au plus égal à 8, soit k = 1 ou 2.

$k*17 = x^2$-15, donc k ^{2} (64–15) / 17 qui est inférieur à 3.

Seul k = 2 convient pour l'unique solution : x = 7.

Cette résolution est caractéristique d'ALICE. Il s'agit de résoudre une équation qui est considérée ici comme un problème avec une contrainte unique. Le système commence par restreindre l'étendue des valeurs possibles de la variable, puis il élimine des valeurs en utilisant des méthodes diverses, ici le modulo. Il finit par une combinatoire restreinte, puisqu'il ne reste plus que deux valeurs possibles. Il aurait été possible de commencer la combinatoire dès que l'on avait restreint à huit valeurs, la solution aurait été atteinte plus rapidement, mais aurait été moins satisfaisante pour un humain où l'on privilégie la limitation intelligente des possibilités. Le modulo est une technique très souvent utilisée lors de la résolution de problèmes par ALICE. Dans le cas présent, un humain y penserait naturellement, mais ALICE a souvent obtenu d'excellentes solutions qui nous surprennent en l'utilisant pour des problèmes où nous ne pensons pas, à tort, à nous en servir ; c'est par exemple le cas pour le problème de Mike et John au tennis (Laurière 78, page 85). C'est un des avantages d'avoir un système général : pour un problème particulier, il est évident qu'une certaine méthode est efficace. On l'incorpore donc au système, mais à partir du moment où elle est présente, celui-ci va s'en servir à juste titre pour d'autres problèmes où nous n'envisageons pas de l'utiliser.

Exemple 2 : C.S.P. SUR DOMAINES CONTINUS sur un cas concret en astrophysique.

Les variables de nature continue se rencontrent dans de nombreux champs de recherche pour la modélisation des erreurs de mesure et la prise en compte de probabilités.

$$x \in [0,1000],\ y \in [0,1000],\ z \in [0,\pi],\ t \in [0,\pi]$$

Nous voyons ici une nouveauté importante de RABBIT. Alors qu'ALICE ne pouvait gérer que des variables qui prenaient des valeurs entières, les variables z et t peuvent prendre ici n'importe quelle valeur comprise entre 0 et ^{1}. Par contre, les variables x et y ne prennent que des valeurs entières.

(1)	$x*y*z = 2*t$
(2)	$x*y + t - 2*z = 4$
(3)	$x - y + \cos^2 z = \sin^2 t$
(4)	$x*\sin z + y*\cos t = 0$

RABBIT détecte d'abord que la variable t peut être calculée exactement à partir de (1) et (2) :

z*(2) donne (5) $x*y*z + z*t - 2z2 = 4z$

(1) dans (5) (6) $(2 + z)* t = 2z*(2 + z)$

comme 2 + z > 0, (6) donne (7) $t = 2z$

il vient par (1) (8) $x*y = 4$

ainsi (1) et (2) sont équivalentes à (7) et (8).

Les calculs présentés dans cet exemple donnent les résultats obtenus par RABBIT, sans que le détail de ces calculs fait par RABBIT soit exactement le même que ce qui est indiqué, ceci dans le but d'en faciliter la compréhension par le lecteur. RABBIT ne va pas décider de porter (1) dans le produit de (2) par z, mais il décide d'éliminer x de (2). Il va donc y remplacer x par 2*t / (y*z), tiré de (1), ce qui donne

$$y*2*t / (y*z) + t - 2*z = 4$$

A ce moment le module de simplification et de normalisation intervient et donne effectivement (6). De la même façon, l'obtention de (12) dans le calcul qui suit se fait en réalité par l'élimination dans (11) de y calculé en fonction de x grâce à (9).

Pour résoudre (3) et (4), RABBIT introduit les inconnues auxiliaires :

$$a = \sin z \quad \text{et } b = \cos t$$

Nous rencontrons une autre nouveauté de RABBIT liée à l'augmentation de ses capacités mathématiques, la possibilité de définir des variables auxiliaires. Il choisit sin z parce que z n'apparaît que sous la forme sin z et $\cos^2 z$.

(3) devient (9) $a*x + b*y = 0$

(4) devient (10) $x - y + 1 - a^2 = 1 - b^2$

d'où (11) $x - y = a^2 - b^2$

b*(11) + (9) (12) $(a + b)*x = b*(a + b)*(a - b)$

soit si a+ b _ 0 (13) $x = b*(a - b)$

x*(11) + (8)　　(14)　$x^2 - 4 = x^* (a + b)^*(a - b)$

(13) dans (14) donne (15)　$b^2{}^*(a - b)^2 - 4 = b^*(a - b)^2{}^*(a + b)$

soit　　(16)　$(a - b)^2 * (b^2 - a^* b - b^2) = 4$

d'où　　(17)　$-a^* b^* (a - b)^2 = 4$

RABBIT a mené ce calcul de main de maître, en substituant (pour obtenir 12, 14 et 15), en mettant en facteurs et en simplifiant quand c'était nécessaire. Pour arriver à (13), on a appliqué la règle : Si M*N = 0, alors M = 0 OU N = 0, qui a été décomposé en deux cas, M = 0 d'une part et l'ensemble des deux contraintes N = 0 et M _ 0 d'autre part. On a examiné ici la deuxième partie de la disjonction.

Or a et b sont inférieurs ou égaux à 1 par construction, donc a*b également. L'équation (17) n'admet donc que les solutions a = 1 , b = –1 d'une part et a = –1 , b = 1 d'autre part, solutions exclues par la condition a + b _ 0.

On a une contradiction puisque la contrainte M _ 0 est ici a + b _ 0, donc fausse pour les deux groupes de solution de la forme (17) de la contrainte N=0 : $-a^* b^* (a - b)^2 = 4$. Le traitement de cette dernière contrainte est délicat du fait que les variables qui y apparaissent sont continues : les possibilités de restriction des valeurs de telles variables sont moins complètes que celles des variables discrètes. Par exemple, quand une relation lie deux variables discrètes, une méthode efficace est de vérifier que certaines valeurs d'une de ces variables mène à une contradiction pour toutes les valeurs de l'autre variable et donc d'éliminer ces valeurs de la première variable. Le fait d'avoir des variables continues ne permet plus cela ; RABBIT ne peut alors plus que manipuler des inégalités. D'après (17), on a $|a|^*|b|^*(a-b)^2=4$; mais $|a|^{2}1$, donc $|b|^*(a-b)^{2}{}^{3}4$. De même, $|a-b|^{2}2$, donc $(a-b)^{2}{}^{2}4$ et, par les deux inéquations précédentes, $|b|^{3}1$. Rapproché de $|b|^{2}1$, cela indique que b vaut 1 ou –1. Si b vaut 1, (17) devient $a^*(a-1)^2 = -4$. Comme $(a-1)^2\ {}^{2}\ 4$, on a $|a|^{2}1$, et la valeur a=1 n'est pas solution de l'équation précédente ; il reste donc a=–1. On voit de même que si b vaut–1, alors a vaut 1. On va examiner maintenant la première partie de la disjonction.

Il reste le cas particulier a + b = 0

avec　　$b = \cos t$

(7)　　$t = 2^*z$

et　　$\cos 2z = 1 - 2a^2$

on a finalement :　　$a + 1 - 2a^2 = 0$

équation qui admet pour racines 1 et –1/2.

RABBIT est capable d'appliquer à bon escient les formules trigonométriques classiques. Il est également capable de résoudre une équation du second degré. Si elle n'a pas de racines, on a une contradiction ; si elle a deux racines x_1 et x_2, elle est remplacée par la nouvelle contrainte : $x=x_1$ OU $x=x_2$.

$$a = -1/2 \text{ et } b = 1/2$$

mais sin $z = -1/2$ n'a pas de solution quand $z \in [0,\pi]$.

RABBIT est également capable de résoudre des équations trigonométriques simples.

$$a = 1 \text{ et } b = -1$$

donne par contre :

$$z = \pi / 2 \qquad \text{d'où } t = \pi \text{ et l'on a bien } \cos t = -1$$

ce qui constitue l'unique solution avec $x = 2$ et $y = 2$.

Les valeurs de x et de y sont facilement obtenues à partir de (8) et de (9) qui est devenu : $x = y$.

La solution $x = -2$, $y = -2$ est éliminée puisque x et y sont positifs ou nuls.

Une remarque importante s'impose ici sur les résolveurs construits sur PROLOG :

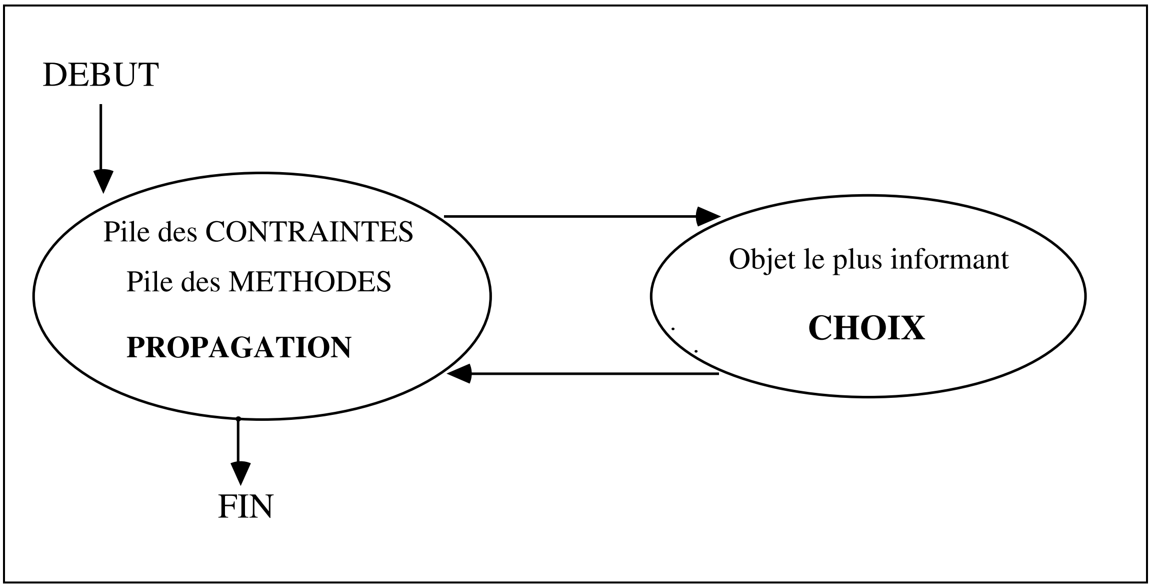
PROLOG se présente a priori comme un excellent langage pour ces problèmes combinatoires : il contient un algorithme d'unification câblé et sait gérer efficacement une arborescence.

Mais en réalité, l'interpréteur PROLOG conserve, à l'exécution, l'ordre donné dans l'énoncé des buts et des clauses. Il s'ensuit que **PROLOG *est bel et bien PROCEDURAL et non DECLARATIF*** : le concepteur du programme guide, volontairement ou non, la recherche.

Ceci est classique dans CHIP (Dincbas 88), par exemple, qui est construit sur PROLOG, et les ingénieurs affinent longtemps à la main les heuristiques *id est* **l'ordre des clauses**.

Un autre exemple est donné par les auteurs de PROLOG III eux-mêmes (Colmerauer, Benhamou 1993) pour le problème du remplissage d'un rectangle par des carrés.

II. SCHEMA GENERAL DES METHODES PAR PROPAGATION



II.1) PROPAGATION

Les algorithmes retenus en phase de propagation sont, en général, *polynômiau*x :

– une étude d'une ou plusieurs équations par congruences dépend par exemple du plus fort coefficient y figurant et de sa décomposition en facteurs premiers : tout est polynômial.

– la méthode de propagation des intervalles de variation des variables à l'intérieur d'une même contrainte (Macworth 77, Davis 87 sur une idée de Waltz 72) est également polynômiale.

– quelques méthodes d'énumération non-polynômiales peuvent cependant être utiles localement : combinatoire pure sur une équation ne comportant que trois variables par exemple.

En fait, il faut souligner que ***la propagation entraîne déjà des choix*** **:**

Quelle contrainte étudier à un instant donné ?
Quelle méthode retenir pour la traiter ?
Quand arrêter la propagation et faire un choix ?

Ces trois questions sur la conduite globale du système sont très convenablement réglées par Méta-règles.

II.2) LES CHOIX

Un choix correspond à l'instanciation d'une variable (arbre de recherche binaire).
Tout choix est de nature heuristique :
le but du jeu est de faire le moins de choix possible (l'idéal est aucun choix).

Deux heuristiques sont utilisées de manière basique :
– (h1) Variable **la plus contrainte** (celle dont le domaine est le plus petit à cet instant),
– (h2) Variable **la plus contraignante** (variable liée au plus grand nombre d'autres variables).
Ces deux heuristiques qui sont, la plupart du temps, en forte compétition pour décider du choix, dérivent d'ailleurs d'une seule **Méta-heuristique** qui dit qu'il faut faire les choix les plus informants.

Malgré tous ces efforts en algorithmes de propagation et en qualité des heuristiques *in fine* **le caractère NP-complet ne disparaît pas pour autant !...**

Et il existe des situations pathologiques où la propagation est hors de prix et au total, la qualité d'un résolveur dépend de trois facteurs :
– richesse et efficacité des méthodes algorithmiques de propagation ;
– qualité des Méta-règles de contrôle sur ces méthodes ;
– adaptation fine à l'état courant du problème pour effectuer un choix.

Mais il existe un quatrième point, essentiel, qui, à notre connaissance, n'a pas été abordé jusqu'à présent : il s'agit de toutes les situations où la propagation donne de mauvais résultats.

III. CAS PATHOLOGIQUES EN PROPAGATION DANS LES C.S.P.

Nous nous intéressons dans la suite aux **cas où les méthodes par propagation cessent d'être efficaces.**

Ceci correspond essentiellement à trois situations types :
1) Le système ne trouve pas les **diverses symétries du problème** et répète inlassablement l'étude de situations équivalentes.
2) Le système est devant un **problème peu contraint** : ayant effectivement traité la plupart des contraintes, il ne fait plus que propager des informations pauvres après chaque choix.

3) Le système perd son temps sur la **preuve de l'optimalité** bien qu'il ait déjà déterminé un (quasi-)optimum.

Donnons quelques exemples concrets de ces différentes situations pathologiques :

III.1. SYMETRIES DU PROBLEME

Considérons les problèmes d'emploi du temps.

A) Cas simple Découpe ou Coloriage :

Les variables sont mono-dimensionnelles, leurs valeurs ne sont pas ordonnées et la contrainte principale s'énonce :

Pour tout couple de variables x_i et x_j telles que :

$$\text{dimension}(x_i) + \text{dimension}(x_j) > \text{taille alors } x_i \neq x_j$$

Tout résolveur devrait alors être capable de détecter les solutions équivalentes qui naissent de la formation d'ensembles de ces couples x_i, x_j : s'il existe m variables telles que

$\forall i \in M$, $\forall j \in M$ avec $M = [1, m]$ on a :

$$\text{dimension}(x_i) + \text{dimension}(x_j) > \text{taille}$$

Alors il existe une **clique** sur ces éléments x_i, $i \in M$, c'est-à-dire qu'**il faut fixer définitivement cet ensemble**.

Si l'on doit découper cinq barres b de six mètres dans des barres B de dix mètres, il ne faut pas considérer que l'on met la première barre b dans la première barre B, puis dans la seconde,…Il faut considérer que la longueur de cinq des barres B n'est plus que de quatre mètres et éliminer les cinq barres b de l'ensemble des barres à obtenir. Sans cette possibilité, le nombre de solutions équivalentes est 5! soit 120 s'il y a aussi cinq barres B et beaucoup plus si ce nombre est supérieur à 5. Quand le problème est peu contraint, le premier choix mènera à la solution et ce ne sera pas gênant. Mais si l'on a fait déjà des choix qui font que le sous-problème n'a plus de solution, un système qui n'est pas capable de tenir compte de cette symétrie considérera tous ces choix et, sauf pour le premier, ce sera inutile. Ce cas est simple à traiter, mais c'est plus délicat s'il y a des grandes barres de dix mètres et d'autres de onze mètres.

B) Emploi du temps général :

Les variables sont multi-dimensionnelles.

Un cours x_i est de la forme { professeur_i, classe_i, salle_i, durée_i }

Cette fois, la symétrie suivante doit être trouvée :

S'il existe deux cours identiques leurs images sont interchangeables.

Si le professeur X fait cinq heures de maths par semaine à la classe de première dans la même salle, il ne s'agit pas de placer successivement ces cinq cours à 14h le lundi ; il suffit d'examiner une seule de ces affectations. Comme dans l'exemple précédent, ce phénomène peut augmenter énormément le nombre de situations équivalentes si le système ne sait pas le prendre en compte.

Et cette symétrie est tout sauf évidente. Il existe de nombreux autres exemples. De plus, de telles "symétries" n'existent pas nécessairement d'entrée de jeu et doivent aussi être détectées *à tout stade de la recherche*.

III.2. RESOLUTION REDUITE A UN ESPACE FAIBLEMENT CONTRAINT

Exemple : *Ordonnancement d'atelier*.

A) Contraintes de type potentiel : $x_i > x_j + d(x_i,x_j)$

Elles sont bien traitées par propagation : algorithmes de plus court chemin et réduction des intervalles.

B) Contraintes disjonctives et cumulatives : limitation des ressources disponibles. C'est un type de contraintes bien connu dans tous les problèmes d'emploi du temps ; il existe aussi de bonnes méthodes pour les prendre en compte, mais elles sont chères car non polynômiales.

C) Autres contraintes plus difficiles, par exemple *ressources croisées* :

$$\Sigma_{i,j}\ C(x_i,x_j) * x_i * x_j < \text{Limite}$$

où les $C(x_i,x_j)$ et Limite sont donnés.

Dans le cas général, une telle contrainte ne donne lieu à aucune information directe, il faut faire un choix sur les x_i et essayer de propager à nouveau, ce qui coûte cher à nouveau.

III.3. PREUVE DE L'OPTIMALITE

Toutes les contraintes sont satisfaites, il ne reste qu'à trouver le minimum d'une fonctionnelle, comme la suivante :

$$\text{minimum de } \Sigma\ \text{Distance}(x_i, x_{i+1})$$

C'est le cas, par exemple, de la détermination du trajet d'un robot mobile dans une usine qui doit collecter des pièces à différentes places en minimisant la distance totale parcourue.

Si les coûts de la matrice *Distance* sont proches les uns des autres (en particulier si les coûts réduits sont faibles, cf Voyageur de Commerce), les choix, suivis de l'interprétation, ne conduisent à une déduction que lorsque le sigma dépasse la meilleure valeur connue, ce qui nécessite en général de nombreux essais.

L'exemple de la section VI donne un bon exemple de ce phénomène.

Aucune méthode rigoureuse n'est connue hors du cas linéaire dans Q (et la Recherche Opérationnelle propose ici des méthodes numériques approchées rapides : *gradient* ou *recuit*).

III.4. BILAN

Le système est dans une situation où :

*** La résolution après propagation est bien avancée et le problème notablement réduit.

*** Il reste peu de contraintes mais elles sont difficiles à analyser.

*** Le domaine de recherche — nombre de points dans l'espace des valeurs — est encore grand (disons $>10^6$).

donc

***** Continuer à propager est cher et inefficace,** car cela implique ***l'interprétation*** du problème résiduel.

Si le système est capable de détecter cette situation, la meilleure solution pour lui est d'engendrer un programme énumératif spécifique qui effectuera une énumération pure sur l'espace restant en profitant de la vitesse d'exécution intrinsèque d'un programme stupide mais compilé.

Tout le contexte courant sera transmis à ce programme et les contraintes restantes seront ***compilées***. C'est le modeste apport de RABBIT.

Nous abordons ici la deuxième amélioration importante par rapport à ALICE. Quand cela ne sert à rien d'être intelligent, le mieux est de faire un programme combinatoire le plus rapide possible. Notons qu'il ne s'agit pas de transformer l'énoncé d'un problème en un programme combinatoire au moment où l'on reçoit un problème. Cette solution, la première qui vient à l'esprit, n'est valable que s'il n'est pas possible de gagner du temps en utilisant les caractéristiques des sous-problèmes engendrés, ce qui est rarement le cas. Une meilleure solution a été mise en œuvre par J.-Y. Lucas dans sa thèse préparée sous la direction de J.-L. Laurière (Génération automatique de programmes par règles et compilation de bases de règles. Application à un système expert de diagnostic de signaux courants de Foucault, Paris 6, 1989). Un des deux systèmes réalisés dans le cadre de cette thèse, SIREN, commence par travailler sur l'énoncé et les données d'un problème particulier comme ALICE. Mais quand la propagation n'apporte plus rien, au lieu de faire des choix et éventuellement de backtracker, SIREN engendre un programme qu'il compile et exécute. L'avantage par rapport à la compilation immédiate vient de l'existence d'une phase préliminaire. Elle est basée sur les méthodes d'ALICE et peut restreindre considérablement l'espace de recherche ; dans certains cas, elle trouvera même directement la solution, par exemple pour résoudre SEND + MORE =MONEY. Un exemple intéressant est un problème proposé par M. Schützenberger et décrit dans (Laurière 86, sections VII.8.2 et VII.8.3). L'arborescence du problème engendrée systématiquement a près de 400 millions de feuilles. SIREN propage les contraintes et les ordonne pour le programme qu'il crée ensuite. Celui-ci développe alors une arborescence de 3822 feuilles. ALICE fait encore mieux puisqu'elle ne considère qu'une arborescence de 64 feuilles qui trouve les 42 solutions de ce problème. Signalons au passage que

les deux sections du livre qui traitent de ce dernier problème expliquent clairement les avantages d'ALICE sur un programme spécifique, même écrit par un excellent informaticien.
Dans cet exemple, SIREN obtient un bon résultat, quoique moins élégant que celui obtenu par ALICE, mais l'exécution du programme combinatoire ne demande que quelques centièmes de seconde. Pour d'autres problèmes, comme la cryptaddition DONALD + GERALD = ROBERT, le programme combinatoire engendré par SIREN doit faire plus de vingt millions d'essais alors que ALICE trouve la solution et en montre l'unicité avec des arborescences comprenant de 3 à 5 feuilles. Il existe donc des cas où faire des choix statiques conduit à des résultats bien inférieurs à ce que donnent des choix dynamiques.

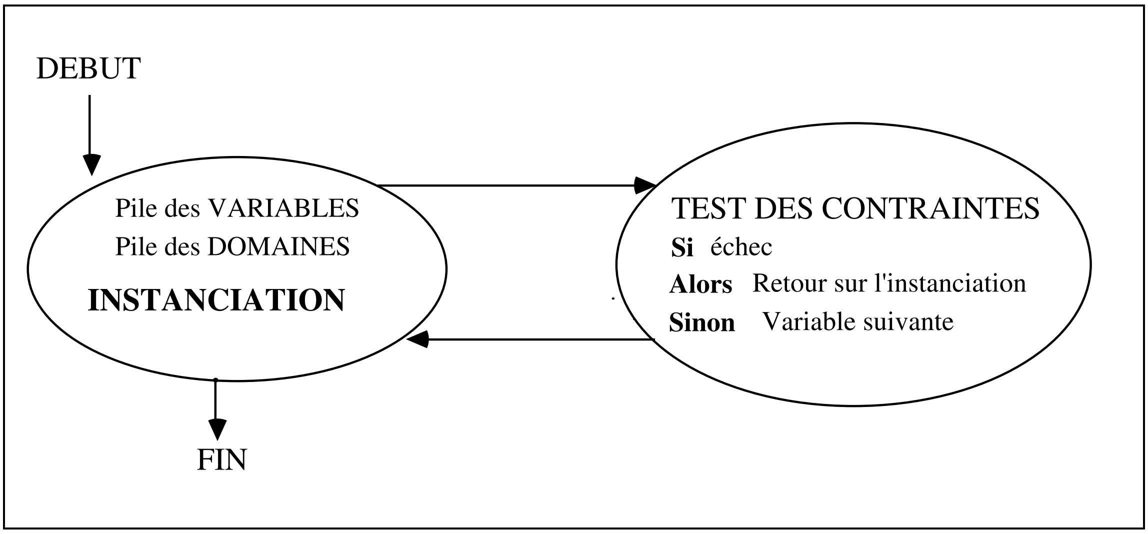
L'idée à la base de RABBIT est de cumuler les avantages des deux méthodes. Quand il est important de faire des choix dynamiques, la système utilise les méthodes d'ALICE, mais quand la force brute est la meilleure approche, il crée un programme combinatoire qui va développer l'arborescence le plus rapidement qu'il peut. A la différence de SIREN, on ne crée plus un seul programme, mais autant de programmes, tous différents qu'il sera nécessaire. En effet, on développe une arborescence comme avec ALICE, mais quand RABBIT estime qu'il est arrivé à une situation où il y a beaucoup de combinatoire à faire et que les méthodes d'ALICE n'apporteraient qu'un faible avantage par rapport à une recherche systématique, il écrit un programme combinatoire, le compile et l'exécute. Puis il revient en mode ALICE en tenant compte de ce qu'il a obtenu dans cette phase combinatoire pure ; par exemple si cela lui a permis de mieux cerner la valeur de l'optimum, il va être plus sélectif. Cette génération de programme peut se faire plusieurs fois au cours de la résolution du même problème et les programmes engendrés seront tous différents, car ils tiennent compte des contraintes existant dans chaque situation.

Un problème délicat est l'estimation par le système du sous-problème qu'il doit résoudre. Il doit tenir compte de la taille de la combinatoire, de l'espoir de gagner encore avec les méthodes d'ALICE et du coût de l'écriture, de la compilation et de l'exécution du programme combinatoire qui résolverait ce sous-problème. Nous avons ici un bon exemple de l'intérêt pour un système de travailler au niveau méta : il compare diverses méthodes possibles pour résoudre un problème, puis il choisit la méthode la mieux adaptée aux caractéristiques du problème qu'il a observées tout en tenant compte de ses propres possibilités en interprétation des connaissances et en écriture de programmes.
Trois éléments déterminent le temps de résolution combinatoire : écriture d'un programme, compilation de ce programme, exécution de ce programme. Le temps d'écriture d'un programme est en général très court. Le temps de compilation est de plusieurs secondes même pour un petit programme. Le temps d'exécution dépend de l'ampleur de la combinatoire, mais s'il est élevé, il le serait souvent encore plus pour la propagation. L'élément qui empêche une utilisation plus fréquente de la création d'un programme spécifique est donc le temps de compilation. Mais ce temps pourrait être annulé. En effet, il ne serait pas beaucoup plus long d'écrire directement un

programme en langage machine plutôt qu'un programme dans un langage de programmation comme le C. Le système n'utilise qu'un sous-ensemble réduit du C dans les programmes qu'il crée et il est facile de produire les instructions machines correspondantes à ces éléments. La plus grande partie du travail du compilateur est de retrouver des informations dont le système dispose quand il a créé le programme. Si le seul temps à prendre en compte est celui de l'écriture du programme, il devient rentable d'appliquer la méthode pour des espaces de recherche bien plus petits. Le système créerait alors des dizaines de programmes, faits sur mesure, au cours de la résolution d'un problème, ce qui rendrait la méthode proposée encore plus intéressante.

IV. STRUCTURE D'UN PROGRAMME COMBINATOIRE SOUS CONTRAINTES

Cette structure est en fait fort semblable au schéma de propagation :



La stratégie de gestion des contraintes et les heuristiques de choix en diffèrent toutefois remarquablement : les données sont ici les variables, les domaines et les contraintes sous forme mathématique qui sont figées cette fois (elles ne seront plus modifiées par le programme).

Les choix s'effectuent sur les variables ordonnées de façon **statique** en fonction des contraintes.

Ainsi, dans une simple inégalité comme : $5 * z < 7 * x + 3 * y$, dans laquelle x et y sont par exemple connues, il n'est pas question de *calculer* les valeurs possibles de z et de les propager. Le programme ne fait qu'énumérer et ne conclura que lorsque z sera elle aussi connue.

Contrairement à un résolveur par propagation cherchant à instancier une variable qui donnera le maximum d'information parce qu'elle intervient, globalement, le plus fortement dans les contraintes, **pour un programme combinatoire**, l'instanciation partielle des variables ne présente ***aucun intérêt***, puisque ce programme ne progresse que lorsqu'une contrainte est ***totalement instanciée***…

L'ordre sur les variables sera donc ici construit en partant de la ***contrainte la plus simple*** (déterminée à partir du nombre de variables et du type de l'opérateur principal), de façon à parvenir le plus vite possible à une évaluation. Les variables figurant dans cette contrainte seront les premières instanciées.

Détermination de l'ordre d'instanciation des variables du programme engendré :

Tant qu'il existe des contraintes

Associer un poids à chaque opérateur de chacune des contraintes restantes

=	<=	>=	!=	∀	∃	==>	autres
100	30	30	2	20	10	5	1

Diviser ce poids par le nombre de variables de la contrainte.

L'idée ici est qu'une égalité a moins de chance d'être vérifiée que toute autre contrainte. Que toute contrainte, outre qu'elle est plus difficile à évaluer, a moins de chance d'être satisfaite quand elle comporte de nombreuses variables.

Associer une masse à chaque inconnue :

masse de x_i = nombre d'occurrences de x_i dans l'ensemble des contraintes affectées de leur poids et du coefficient multiplicateur de x_i quand il existe.

Déterminer la contrainte dont la masse est la plus forte.
Empiler cette contrainte et les variables qu'elle concerne dans l'ordre de leurs masses et ignorer dorénavant cette contrainte.

Fin Tantque.

La masse d'une contrainte est le poids de son opérateur principal. Une fois la contrainte choisie, on affecte d'abord une valeur à la variable de cette contrainte dont la masse est la plus élevée et l'on continue jusqu'à ce que l'on ait affecté toutes les variables de cette contrainte. Si la contrainte où tout est instancié n'est pas satisfaite, on bactracke. Le cas de l'égalité est une exception, on affecte toutes les variables sauf la variable de plus forte masse, dont la valeur est alors obtenue par cette égalité. Quand la masse est élevée parce que le coefficient multiplicateur de la variable est grand, il y a peu de chances que le résultat soit acceptable : le résultat est alors un quotient et plus le dénominateur est grand, moins il y a de chances qu'il divise exactement le numérateur. Cela permet de bactracker souvent sans avoir besoin de considérer toutes les valeurs possibles de la dernière variable.

Nous n'avons ici que la partie principale du calcul du poids des contraintes et de la masse des variables. En réalité, le système prend aussi en compte les dimensions des ensembles, les complexités des contraintes et les intervalles connus pour les valeurs possibles des inconnues. Une description complète des heuristiques de RABBIT ne ferait qu'alourdir inutilement l'exposé.

V. COMPILATION DES CONTRAINTES

Cas des égalités

Toute égalité permet, en fait, au programme engendré, d'éliminer une variable et de reporter son expression formelle partout ailleurs : à l'intérieur de toute égalité la variable de plus forte masse est ainsi sortie. Si elle est affectée d'un facteur multiplicatif, la valeur de cette inconnue sera calculée selon une division entière et la concordance numérique n'en sera que moins probable.

Remarquons ici que de telles éliminations pénaliseraient, en revanche, un programme de pure propagation : elles alourdiraient le système sans apporter d'informations et en compliquant l'analyse des contraintes. En énumération, au contraire, elles font gagner significativement en efficacité car le nombre de variables à énumérer diminue d'autant.

Exemple : *Architecture dans N^2*

Pour tout module i, à placer dans le domaine plan considéré, la relation suivante relie ses différentes caractéristiques :

(1) Haut(i) – Bas(i) = Longueur(i) + Orientation(i) * (Largeur(i) – Longueur(i))

Largeur et Longueur sont des données, tandis que les autres entités sont les variables à déterminer. Parmi celles-ci, Orientation n'a que 2 valeurs possibles 0 et 1. De plus cette variable figure dans toutes les contraintes avec une masse forte à cause de la présence du signe de multiplication. Comme le domaine d'Orientation est de cardinalité 2, son élimination via la relation fondamentale ci-dessus, va la contraindre encore plus.

L'égalité (1) est traduite en :

$\forall i \in$ Modules   O = (Haut(i) – Bas(i) – Longueur(i)) / (Largeur(i) – Longueur(i))

Si O est une valeur inacceptable pour Orientation(i), il y aura retour-arrière.

Il n'est pas nécessaire d'instancier O(i), on vérifie que O ne prend que la valeur 0 ou 1. Plus le nombre en dénominateur est grand, et plus il est rare qu'il divise exactement le numérateur. Dans les rares cas où cela est vrai, on a alors la valeur de O(i). Si l'objet 3 a pour longueur 10 et pour largeur 3, la contrainte sera :

$$\text{Haut}(3) - \text{Bas}(3) = 10 - 7*\text{Orientation } (3)$$

et la plupart des valeurs pour Haut(3) et Bas(3) donneront bien un échec. Par exemple, si Haut(3) vaut 25, l'équation ne sera satisfaite que pour les valeurs 15 et 22 de Bas(3).

La programmation des inégalités (>, ³) ou inéquations (!=) ne peut donner lieu à un test qu'une fois toutes les variables concernées instanciées.

Exemple : La contrainte $\text{Haut}(i_1) + \text{Haut}(i_2) + \text{Haut}(i_3) + 6 < \text{hauteur}$

est traduite en : Si ($\text{Haut}(i_1) + \text{Haut}(i_2) + \text{Haut}(i_3) + 6 < \text{hauteur}$).

Seulement une fois tout connu , la détection d'un échec provoque un retour.

Les contraintes en ∀ et ∃ sont compilées par une boucle sur l'ensemble désigné par le quantificateur.

Un test sur les contraintes qui figurent à l'intérieur de la boucle.doit alors prendre place avec un résultat qui dépend du quantificateur.

Exemples :

Cas du "Quel que soit" $\forall i \in [4, K]$ P(i) donne simplement :

Pour (i= 4; i<= K; i++) Si !P(i) Alors Echec.

Cas du "Il existe" $\exists i \in [K, K+4]$ P(i) donne :

ok= 0;

Pour (i= K; i<= K+4; i++) Si P(i) Alors ok= 1.

Si !ok Alors Echec.

Les contraintes en ==> et <=> sont réécrites, par normalisation, en tests logiques.

Exemple : La contrainte V(i) < n ==> W(V(i)) = V(i+1)

est traduite en :

Si V(i) < n
Alors
Si W(V(i)) != V(i+1)
Alors Echec.

La fonction économique à optimiser est mise à jour après chaque instanciation et chaque retour.

Elle ne peut être évaluée que lorsque toutes les variables qui la composent ont été déterminées. A cet instant, son seul effet est de provoquer un échec si sa valeur dépasse la meilleure valeur obtenue auparavant.

VI. COMPARAISON PROGRAMME GENERE VERSUS PROPAGATION

Si la taille du problème de départ est faible (inférieure à 10^{10}), écrire un programme spécifique ne sert à rien : le temps de *la compilation en langage C* sur SUN3 est *toujours supérieur à 5 secondes*, même pour un programme de cent lignes.

En revanche, pour des tailles supérieures, l'expérience montre que les programmes générés, qui comportent de mille à dix mille instructions, permettent de conclure en un temps CPU qui peut être de *cent fois inférieur* à une pure propagation.

Deux raisons expliquent ce fait :

La première est due au fait que dans le programme combinatoire, toute égalité donne lieu à une élimination de variables (relations fonctionnelles de la littérature).

En propagation de telles éliminations systématiques seraient catastrophiques : elles entraînent, en effet, une augmentation notable de la complexité des contraintes.

La seconde raison tient à ce que : **propager = interpréter**.

Lors de toute propagation, le contexte et les expressions formelles ne cessent de changer et aucune compilation en dur n'est envisageable. Par contre, le fait de figer un état du problème fait gagner du temps par rapport à l'interprétation.

Nous donnons ci-après un exemple simple, mais nous semble-t-il significatif de l'intérêt d'écrire automatiquement un programme spécifique.

ROBOT ATELIER

****

Un robot mobile a pour tâche de collecter périodiquement des matériaux et des produits finis dans différents locaux d'un atelier. Il doit en même temps déposer chacun d'eux à un endroit prévu à l'avance, tout en emprutant au total le chemin le plus court.

Une formulation possible de ce problème part directement des distances entre objets sans se préoccuper des différents locaux :

Soit	constantes	N , P	// nombre de tâches, poids total
Soit	ensemble	EVT = [1, 2*N+1]	// évènement : prise ou pose d'objet
Soit	ensemble	EV2 = [1, 2*N]	
Soit	coefficient Poids sur EVT		// poids des objets à transporter
Soit	coefficient Distance sur EVT x EVT		
			// matrice des distances entre objets
Trouver	bijection f : EVT -> EVT		// action du robot au temps t
Avec	minimum Σ i $\in$ EV2 DIS (f(i),f(i+1))		
			// minimum du trajet parcouru
Avec $\forall$i $\in$ EVT (f(i) <= N+1) ==> $\forall$k $\in$ [1, i-1] f(k) != f(i)+N			
		// le robot ne peut saisir un objet qu'il a déjà posé	
Avec	$\forall$i $\in$ EVT Σ j $\in$ [1,i] Poids(f(j)) <= P		// poids limite
Avec	f(1) = 1		// position initiale
FIN			

A la suite de l'énoncé du problème, on donne les données numériques particulières à ce problème. Mais nous allons d'abord reprendre tous les éléments de cet énoncé pour en préciser le sens.

Soit constantes N , P

N est le nombre d'objets à prendre (et à poser ensuite) ; P est le poids maximum que le robot peut porter.

Soit ensemble EVT = [1, 2*N+1]

La principale difficulté pour comprendre cet énoncé vient de ce que EVT recouvre deux ensembles différents qui ont été fusionnés parce qu'ils ont le même cardinal. Le premier ensemble indique la succession du temps : 1 correspond à l'instant initial, 2 à l'instant suivant et 2*N+1 au dernier instant où le robot a terminé son travail. On peut interpréter i comme "à la i-ième minute". Le deuxième ensemble contient comme premier élément la position initiale du robot, les N éléments suivants indiquent les N tâches où le robot prend des objets, l'élément j+1 correspondant à la tâche "prendre l'objet j". Les N derniers éléments indiquent les N tâches où l'on dépose des objets, l'élément N+j+1 correspondant à la tâche "poser l'objet j".

Soit ensemble EV2 = [1, 2*N]

Il s'agit d'instants, EV2 correpond au premier sens de EVT. EV2 est indispensable pour exprimer la contrainte du trajet à minimiser qui fait intervenir la position à l'instant initial et celle à l'instant final de chaque étape ; il y a un intervalle de moins que d'instants dans la suite des temps.

Soit coefficient Poids sur EVT

Nous avons le deuxième sens de EVT, il s'agit de tâches. La tâche 1 indique la situation de départ, le robot ne portant aucun objet. Les tâches suivantes sont celles des objets à prendre, les poids de 2 à N+1 sont positifs ; les dernières tâches sont celles où l'on pose les objets, donc les poids de N+2 à 2*N+1 sont négatifs. De plus, comme l'objet i est pris en i+1 et posé en N+i+1, on aura Poids(N+i)=–Poids(i) pour i variant de 2 à N+1. Ceci n'est pas une convention arbitraire, il faut la respecter dans l'établissement des données, car elle est essentielle dans la définition de la contrainte que nous verrons plus bas qui assure que le robot ne peut saisir un objet qu'il a déjà posé.

Soit coefficient Distance sur EVT x EVT

Dans les deux cas, il s'agit du deuxième sens de EVT. DIS(a,b) indique la distance entre le point où il faut prendre (ou poser si a>N+1) l'objet correspondant à a et celui où il faut prendre (ou poser) l'objet correspondant à b.

Trouver bijection f : EVT -> EVT

En réalité, on a une bijection entre le premier sens de EVT et le second. Cette bijection ordonne les tâches: a=f(b) signifie que la tâche effectuée à l'instant b est la tâche a. En indiquant que l'on a une bijection, on tient compte automatiquement de deux familles de contraintes : on ne peut prendre deux fois la même pièce et on ne peut poser deux fois la même pièce.

Avec minimum Σ $i \in$ EV2 DIS (f(i),f(i+1))

Contrainte classique de la minimisation du chemin total parcouru. Cette contrainte est développée pour ALICE (et pour RABBIT fonctionnant en mode ALICE) en une somme de 2*N termes qu'il faut minimiser. Par contre, quand RABBIT crée un programme, il ne développe pas cette somme, qui est traduite de façon plus compacte par une boucle "Pour".

Avec $\forall i \in$ EVT (f(i) <= N+1) ==> $\forall k \in$ [1, i-1] f(k) != f(i)+N

f(i) <= N+1 assure que la tâche exécutée à l'instant i consiste à saisir un objet. Dans ce cas,

$\forall k \in$ [1, i-1] f(k) != f(i)+N

assure qu'au moment où l'on saisit cet objet, on ne l'a pas déjà posé, k correspondant à un instant précédant i et f(i)+N étant la tâche consistant à déposer l'objet f(i) saisi à l'instant i. On aurait pu aussi bien imposer qu'un robot ne peut poser un objet qu'il n'a pas encore saisi, il suffirait d'écrire :

$$\forall i \in \text{EVT} \ (f(i) > N+1) \implies \exists k \in [1, i-1] \ f(k) = f(i)-N.$$

On commence par s'assurer que l'on a bien une pose, puis on vérifie qu'à un instant k précédent on a bien pris ce même objet. Parce qu'il y a une bijection, on se rend compte que ces deux formulations donnent le même résultat ; il y a de nombreuses façons d'écrire un énoncé.

Avec $\forall i \in \text{EVT} \sum j \in [1,i] \ \text{Poids}(f(j)) <= P$

La charge du robot à l'instant i est obtenue en cumulant tout ce qu'il a pris et en enlevant tout ce qu'il a posé depuis le début jusqu'à (et y compris) cet instant. A chaque instant, cette charge doit être inférieure à la charge maximale autorisée.

Avec $f(1) = 1$

A l'instant initial, le robot est dans sa position de départ.

Le formalisme de définition des problèmes dans RABBIT est le même que celui d'ALICE. Ce formalisme est totalement déclaratif, mais il n'est pas toujours facile d'y définir un problème. Il existe de nombreuses façons de le faire, et elles ne sont pas forcément équivalentes au point de vue efficacité. De façon générale, il est préférable d'utiliser au mieux les ordres "trouver" et d'utiliser les contraintes aussi peu que possible. En effet, le traitement formel des contraintes ralentit considérablement ALICE. Comme RABBIT compile les contraintes, cette objection est moins valable pour les phases où il exécute des programmes qu'il a créés, mais elle est toujours valable pour les phase en fonctionnement de type ALICE qui les précèdent. Nous venons de voir qu'en choisissant de chercher une bijection, on avait du même coup pris en compte deux familles de contraintes.

RABBIT, dans sa phase ALICE, développe, comme ALICE, les contraintes contenant $\forall$ ou Σ. C'est ainsi que la contrainte

$$\forall i \in \text{EVT} \ (f(i) <= N+1) \implies \forall k \in [1, i-1] \ f(k) \mathrel{!=} f(i)+N$$

va donner lieu à 1+2+3+…+2N, soit N*(2N+1), contraintes plus simples. Si N vaut 10, cela fait 210 contraintes. Des exemples de ces contraintes, où N devra être remplacé par sa valeur, sont :

$$f(2) <= N+1 \implies f(1) \mathrel{!=} f(2)+N$$

$$f(3) <= N+1 \implies f(1) \mathrel{!=} f(3)+N$$

$$f(3) <= N+1 \implies f(2) \mathrel{!=} f(3)+N$$

$$f(4) <= N+1 \implies f(1) \mathrel{!=} f(4)+N$$

Pour sa part, la contrainte $\forall i \in \text{EVT} \sum j \in [1,i] \ \text{Poids}(f(j)) <= P$ va donner naissance à 2N+1 contraintes où P sera remplacé par sa valeur, dont les quatre premières sont :

Poids(f(1)) <= P

Poids(f(1)) + Poids(f(2)) <= P

Poids(f(1)) + Poids(f(2)) + Poids(f(3)) <= P

Poids(f(1)) + Poids(f(2)) + Poids(f(3)) + Poids(f(4)) <= P

Ce sont sur ces contraintes que RABBIT travaille dans sa phase ALICE.

Remarquons que RABBIT a deux façons de traiter le quantificateur "quel que soit" quand il crée un programme. Il peut d'abord le développer comme ALICE. Dans l'exemple donné plus loin pour un des programmes engendrés à partir de cet énoncé, nous verrons qu'il le fait pour la deuxième contrainte, celle qui assure que l'on n'a pas déjà posé l'objet que l'on saisit. Par contre, dans certains cas il ne développe pas une contrainte quantifiée universellement, mais il la remplace par une boucle "Pour". Cela se produit évidemment quand l'ensemble sur lequel porte le quantificateur n'est pas connu au moment où l'on écrit le programme. Il n'y a pas d'autre solution que d'écrire une boucle qui balayera chaque fois des ensembles différents. Mais c'est aussi le cas quand il serait maladroit de développer la contrainte. C'est le cas pour la troisième contrainte, celle qui vérifie que le poids porté par le robot est inférieur à une certaine limite. Si on la développait, le programme engendré recalculerait pour chaque étape la somme des prises ou dépôts d'objets depuis le début. Il est préférable de la vérifier itérativement, le poids étant égal au poids précédent plus (ou moins) le poids du dernier objet pris (ou déposé). A chaque fois, on vérifie naturellement que la condition est satisfaite pour ce poids.

Selon le formalisme d'ALICE, les données numériques d'un problème particulier suivent l'énoncé général du problème. Les variables sont définies d'après l'ordre dans lequel elles apparaissent dans les ordres "**Soit**" de l'énoncé. Nous avons donc successivement N, P et les coefficient Poids puis Distance. EVT et EV2 n'apparaissent pas puisqu'ils sont définis par la donnée de N.

```
10 60                                                       // données de N et P
0 14 30 4 30 40 4 4 2 20 2 -14 -30 -4 -30 -40 -4 -4 -2 -20 -2     // données des poids
```

Un poids positif correspond à la prise d'un objet et un poids négatif à la pose de cet objet. Nous voyons que l'on a bien Poids (i+N) = –Poids(i) pour i>1. La plupart des poids sont faibles devant le poids total, 60 kilos, que le robot peut porter ; les sept objets les moins lourds ne pèsent ensemble que 50 kilos. Il peut donc transporter plusieurs objets simultanément, et nous pouvons nous attendre à un trajet total de longeur assez faible.

0 0 1 2 5 4 0 3 4 1 4 2 3 5 0 1 5 0 1 2 5// données des distances
0 0 1 2 5 4 0 3 4 1 4 2 3 5 0 1 5 0 1 2 5
1 1 0 2 5 4 1 3 4 0 4 2 3 5 1 0 5 1 0 2 5
2 2 2 0 3 2 2 1 2 2 2 0 1 3 2 2 3 2 2 0 3
5 5 5 3 0 1 5 2 1 5 1 3 2 0 5 5 0 5 5 3 0
4 4 4 2 1 0 4 1 0 4 0 2 1 1 4 4 1 4 4 2 1
0 0 1 2 5 4 0 3 4 1 4 2 3 5 0 1 5 0 1 2 5
3 3 3 1 2 1 3 0 1 3 1 1 0 2 3 3 2 3 3 1 2
4 4 4 2 1 0 4 1 0 4 0 2 1 1 4 4 1 4 4 2 1
1 1 0 2 5 4 1 3 4 0 4 2 3 5 1 0 5 1 0 2 5
4 4 4 2 1 0 4 1 0 4 0 2 1 1 4 4 1 4 4 2 1
2 2 2 0 3 2 2 1 2 2 2 0 1 3 2 2 3 2 2 0 3
3 3 3 1 2 1 3 0 1 3 1 1 0 2 3 3 2 3 3 1 2
5 5 5 3 0 1 5 2 1 5 1 3 2 0 5 5 0 5 5 3 0
0 0 1 2 5 4 0 3 4 1 4 2 3 5 0 1 5 0 1 2 5
1 1 0 2 5 4 1 3 4 0 4 2 3 5 1 0 5 1 0 2 5
5 5 5 3 0 1 5 2 1 5 1 3 2 0 5 5 0 5 5 3 0
0 0 1 2 5 4 0 3 4 1 4 2 3 5 0 1 5 0 1 2 5
1 1 0 2 5 4 1 3 4 0 4 2 3 5 1 0 5 1 0 2 5
2 2 2 0 3 2 2 1 2 2 2 0 1 3 2 2 3 2 2 0 3
5 5 5 3 0 1 5 2 1 5 1 3 2 0 5 5 0 5 5 3 0

Cette matrice de distances est symétrique par rapport à la diagonale. La diagonale principale est évidemment nulle. Remarquons que le robot se trouve au départ à l'emplacement où se trouvent les objets 1 et 6 et où il faudra déposer les objets 4 et 7.
La fonction économique à minimiser est la somme des distances. Comme les distances sont en général faibles, au plus cinq, il faudra attendre d'avoir défini la valeur de la plupart des f(i) pour avoir une valeur supérieure ou égale à la valeur la plus faible déjà rencontrée. Ceci fait que ce problème est très difficile, malgré le nombre limité d'objets à transporter.

N dépend du nombre d'évènements, c'est-à-dire du nombre d'objets à saisir ou poser, ici 21. La première contrainte en $\forall i \in$ EVT ($f(i) <= N+1$) est facile à évaluer. A tout instant, le robot ne peut porter un poids plus lourd que P= 60 kg. Enfin, il faut minimiser la distance totale parcourue. Mais la matrice des distances, donnée à la fin de l'énoncé, est *déjà réduite* elle possède au moins un zéro par ligne et par colonne. Il sera très difficile avant d'avoir choisi une valeur pour presque toutes les variables, d'éliminer une branche à l'aide de cette seule fonctionnelle.

Le programme engendré a la structure suivante :

```
        Lire les masses des variables, le poids des contraintes et le contexte
(coefficients,   domaines)

        Tant qu'il existe une variable non instanciée
                Prendre la valeur suivante pour la variable de plus forte masse.

                Tant qu'aucun échec n'est rencontré

                q= 0;                    // poids maximum que peut supporter le robot
                Pour (k= 1; k <= 21; k++)
                Si connu [k] Alors q+= Poids  [f [k] ] Sinon break
                Si (q> P) Alors Echec.
                                         // fonction économique
                feco= 0;
                Pour ( i= 1; i< nombre de variables; i++)
                Si connu [i] et connu [i+1]
                Alors feco+= Distance [f [i], f [i+1]]
                Si feco> fsup Alors Echec.
                                         // contrainte sur les évènements, développée :
                                         // pour l'évènement numéro 9, il vient :
                Si connu [9]
                Si f [9] ² N+1
                Alors
                Pour( k= 1; k<= 8; k++)
                Si connu [k]
                Alors                    // aucun k antérieur prise du même objet
                Si  f [k]= f [9] + 10 Alors Echec
                FinPour
                FinSi
```

La condition "connu [k]" signifie que l'on a déjà attribué une valeur à f[k]. Le traitement de la première contrainte pose un problème, puisqu'elle ne peut être évaluée à un instant que si l'on connaît tous les événements qui précèdent cet instant. En effet, si l'on n'a pas encore défini dans une étape qui précède l'instant k la pose d'un objet que l'on sait que le robot a pris, on peut parfois croire à une surcharge qui n'existe pas puisque l'on n'a pas déduit le poids de l'objet posé du poids total porté par le robot. C'est pourquoi RABBIT a inclus "Sinon break" pour sortir de la boucle dès que l'on rencontre une valeur inconnue.

Pour que cette contrainte soit utile, il faut donc que l'on affecte les f[i] dans l'ordre des i croissants. Mais c'est ce qui se produit puisque les variables d'indice faible apparaissent dans d'autant plus de contraintes de poids qu'ils sont faibles, et surtout dans les contraintes qui ont peu de variables. f[i] aura donc une masse d'autant plus importante que i sera petit puisqu'il sera dans beaucoup (on va donc accumuler le poids de beaucoup de contraintes) de contraintes d'inégalité (donc à poids important) ayant peu de variables (donc on divise la masse 30 affectée à l'inégalité par un entier petit). Les contraintes issues du deuxième "**Avec**" ont une implication dont le poids n'est que de 5 et leur influence sera insuffisante pour renverser la tendance précédente. Ce phénomène explique d'ailleurs pourquoi le programme engendré par RABBIT commence par vérifier que le robot ne supporte jamais une charge trop lourde : ce sont celles qui ont le poids le plus élevé. Ordonner les contraintes de façon à commencer par les plus contraignantes est important pour que le programme engendré développe rapidement l'arborescence. A la différence d'ALICE, il ne peut plus changer cet ordre une fois qu'un programme est créé, l'ordre est alors imposé. Mais cela est meilleur que pour SIREN qui ne créait qu'un seul programme : avec RABBIT, chacun des programmes engendrés peut avoir un ordre des contraintes différent.

La variable fsup est la valeur minimale déjà rencontrée pour la fonction économique. Cette variable a été initialisée dans la phase ALICE de l'optimisation où l'on commence par trouver une solution satisfaisant toutes les contraintes sauf la contrainte d'optimisation. Dans le cas présent, cette valeur initiale de la distance totale parcourue était de 21 pour une valeur optimale de 18. Il n'est pas nécessaire d'ajouter des sorties de boucle avec "Sinon break" pour la fonction économique. En effet, toutes les quantités que l'on ajoute sont positives ou nulles et si une évaluation partielle dépasse le précédent minimum, l'évaluation totale sera au moins supérieure ou égale à celui-ci ; dans ce cas, on backtracke.

L'ensemble des huit dernières lignes est répété au plus vingt fois, à chaque instant on regarde si l'on n'a pas déjà posé l'objet que l'on prend. Remarquons que c'est ici que les programmes engendrés à divers endroits de l'arborescence vont différer. D'une part, il n'est pas nécessaire de mettre la condition "Si connu[9]" si f[9] est déjà connue au moment où l'on va lancer le programme. D'autre part, dans ce cas où f[9] est connu, il est également inutile de mettre la condition suivante "Si f [9] ² N+1" quand elle est vraie, c'est à dire quand on a pris un objet à l'instant 9. Quand cette condition est fausse, si l'on a posé un objet à l'instant 9, on ne met pas du tout les huit lignes de test correspondant à une contrainte qui porte sur le cas où l'on prend un objet à l'instant 9. On ne passera pas son temps à faire des tests inutiles. Le gain obtenu en engendrant des programmes différents à différents emplacements de l'arborescence n'est ici pas très élevé, mais il peut être considérable dans d'autres problèmes. D'une part, on peut avoir moins d'instructions et d'autre part elles sont mieux ordonnées.

Ce programme est complété par des modules standards de lecture, d'édition, de gestion de la pile des variables et de celle de leurs valeurs (continuellement mise à jour ici puisque l'on cherche une bijection).

Dix pages de code sont générées pour cet exemple simple.

RABBIT sans programme spécifique trouve la solution optimale de valeur 18 en 38 minutes 17 secondes (sur SUN 3) après 38 643 choix : les faibles coûts de la matrice Distance le ralentissent considérablement.

Si RABBIT est autorisé à programmer, il écrit 5 programmes différents liés à 5 nœuds de son arbre de recherche, pour lesquels de 2 à 5 variables sont déjà fixées. En outre, une solution de valeur 21 a été obtenue d'emblée par des heuristiques de faisabilité (cf Laurière 1976).
La compilation de ces 5 programmes prend en moyenne 5 secondes. Leurs temps d'exécution sont 104, 51, 109, 26 et 20 secondes. RABBIT utilise 208 secondes (propagation + génération) pour 1782 choix. Le temps total est maintenant de 543 secondes.

Le gain est d'un facteur quatre et si ces cinq programmes sont engendrés après quelques choix seulement, le travail en propagation pure de RABBIT n'en a pas moins notablement dégrossi le problème en amont.

Quelques résultats sur d'autres cas typiques sont présentés dans le tableau suivant.

EXEMPLES	**Nb variables résiduelles**	**Type Situation**	**Temps Propagation**	**Temps Pgs générés**	**Nb Pgs**
Ordo. Atelier	50	peu contraint	5mn	8s	2
Architecture N^3	630	contraintes difficiles	265mn	12s	6
Rotation Avions	1624	fonction économique	11 heures	323s	9
Couplage Min Max (contraintes quadratiques)	4047	contraintes difficiles + optimalité	27 heures	420s	12

VII. CONCLUSION

PROPAGATION ET PROGRAMMATION AUTOMATIQUE

1) Tant que la propagation est efficace, ce qu'évalue le système par la mesure de la restriction des domaines des variables après chaque choix,

Alors continuer le schéma : **propagation + choix**.

2) Dès que ce n'est plus le cas, **engendrer automatiquement un programme spécifique** adapté au problème et au nœud courant de l'arbre de recherche ;

Transmettre les résultats au Résolveur.

Il existe des problèmes où il ne sert à rien de perdre beaucoup de temps pour en gagner très peu en essayant de restreindre les essais à faire ; cette méthode a parfois un trop faible rendement. RABBIT est assez métaintelligent pour se rendre compte des situations où il ne lui sert à rien de se servir de son intelligence. Il préfère alors utiliser son intelligence pour créer un programme spécifique bien adapté au sous-problème en cours. Cela correspond à la démarche humaine, où nous écrivons bien des programmes pour résoudre les problèmes pour lesquels nous ne voyons pas de solution intelligente à notre portée. Mais la programmation est pour un humain une activité lente qui entraîne de nombreuses erreurs. Aussi, faisons-nous très peu de programmes, nous essayons d'en écrire un seul pour toutes les situations d'un problème ; il en résulte qu'il sera mal adapté. RABBIT n'a pas cette limitation, il écrit des programmes vite et bien, ce qui lui permet d'en écrire plusieurs dans la résolution d'un seul problème. La principale restriction actuelle est le temps de compilation, mais nous avons vu qu'elle pourrait être levée en produisant directement des programmes en langage machine.

VOIES DE RECHERCHE

Nous pensons qu'**avant toute résolution de problème, il y a la formulation du problème lui-même**. Parvenir à un énoncé mathématique rigoureux à partir d'un discours en français n'est certes pas chose aisée et beaucoup trichent à ce niveau.

Ensuite il ne s'agit pas tant de résolution de problèmes, que de **démonstration automatique**. [Pastre 78, 82, 88, Pitrat 66 (!...)] car la résolution, elle-même, doit être guidée par ces résultats sur l'énoncé, qui décident des **méthodes**, des **méta-choix** et des **heuristiques**.

L'heuristique (h2) par exemple, appliquée à l'énoncé formel d'un problème de couplage dans un graphe va donner (MULTI-TAC de Minton 93):

– (h2) *Sommet n'appartenant pas encore au couplage de degré le plus fort*

Les heuristiques générales, données au programme, ne peuvent être améliorées automatiquement en les particularisant pour le problème en cours, QUE par un raisonnement au niveau META.

REFERENCES BIBLIOGRAPHIQUES

P. Albert, P. Puget 1991-1994 : élaboration du système PECOS manipulant des contraintes et des objets, ILOG.

J.F. Allen 1983 : Maintaining knowledge about temporal intervals, CACM 26-11, pp 823-843.

F. Benhamou 1993 : calcul booléen, pivot de Gauss en rationnels (application de PROLOG III)

H. Berliner 1982 : The B* search, A.I. 17, pp 24-56.

A. Borning 1979 : A constraint oriented Simulation Laboratory (description du langage THINKLAB)

A. Colmerauer 1990 : An introduction to PROLOG III, CACM 33-7, pp 70-80.

E. Davis 1987 : Constraint propagation with Interval Labels, AI 32 , pp 281-331.

M. Dincbas, P. Van Hentenrick, H. Simonis, A. Aggoun, T. Graf, F. Berthier 1988 : The Constraint Logic Programming Language CHIP, Proc. ICFGCS Japan.

H. Gallaire, M. Dincbas 1988 : élaboration du langage CHIP de programmation logique sous contraintes, COSYTECH, ECRC.

M. Gondran 1974 : Programmation entière, E.D.F. série C 2, pp 67-78.

E. Hyvönen 1989 : Constraint Reasoning Based on Interval Arithmetic, A.I. 58, pp 72-112

J. Jaffar et J.-L. Lassez 1987 et W. Older et F. Benhamou 1993 : élaboration de CLP (Constraint Logic Programming).

E. Johnson 1981 à 1993 : élaboration de OSL, système d'optimisation linéaire en nombres entiers à partir du Simplexe, IBM.

R. Karp 1972 : Reductibility among Combinatorial Problems, C.C.C. Plenum.

J.-L. Laurière 1976 : Un langage et un programme pour énoncer et résoudre des problèmes combinatoires, Thèse Paris VI.

J.-L. Laurière 1978 : ALICE : A Language and a program for stating and solving combinatorial problems, A.I. 10, pp 29-117.

B. Lemaire 1974 : Résolution de problèmes de tournées par arborescence, Thèse Paris 6.

O. L'homme 1994 : Contribution à la résolution de contraintes sur les réels par propagation d'intervalles, (description de INTERLOG II), Thèse Paris XI.

B. Liu et Y. Ku 1992 : élaboration de ConstraintLisp, Stanford.

A. Mackworth 1977 : Consistency in a Network of Relations, A.I. 8, pp 99-118.

J. Marcovich, J. Rohmer 1989 : élaboration de CHARME, BULL.

S. Minton : An analytic learning system for specializing heuristics, (présentation de MULTI-TAC), IJCAI 93, pp 922-928.

A. Newell, J. Shaw, H. Simon 1958 : GPS a General Problem Solving program, Inf Proc. 256.

D. Pastre 1978 : Automatic theorem proving in set theory, A.I. 10, pp 1-27.

J. Pitrat 1966 : Réalisation de programmes de démonstration de théorèmes utilisant des méthodes heuristiques Thèse d'état Paris.

J. Pitrat 1990 : Métaconnaissance : futur de l'intelligence artificielle, HERMES.

J.-F. Puget 1993 : A C++ implementation of CLP, ILOG

M. Selz-Laurière 1978 : Fast Algorithm for Knapsack, Mathematical Programming 14, pp 1-16

G. Slabodsky et al. 1972 : Théorie des Graphes et Emplois du temps, Thèse 3e cycle Université Paris Dauphine.

M. Stockmeyer 1973 : Word Problems requiring exponential time, ACM 5th SIGACT.

P. Taillibert 1993 : Programmer en PROLOG sans effet de bord, Dassault El., N 631 304

P. Van Hentenrick 1989 : Constraint satisfaction in Logic Programming, MIT Press.

P. Van Hentenrick et al. 1993 : élaboration de cc(FD), version CHIP de Brown University.

D. Waltz 1972 : Generating semantic descriptions from drawings of scenes with shadows, M.I.T.