Etude et réalisation d'un ensemble de primitives pour la satisfaction de contraintes en domaines finis

Pierre Berlandier

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1389

Programme 2
Calcul symbolique, Programmation et Génie logiciel

ETUDE ET REALISATION D'UN ENSEMBLE DE PRIMITIVES POUR LA SATISFACTION DE CONTRAINTES EN DOMAINES FINIS

Pierre BERLANDIER

Janvier 1991

RR_1389

Etude et réalisation d'un ensemble de primitives pour la satisfaction de contraintes en domaines finis

Pierre BERLANDIER

berlandi@mirsa.inria.fr

Etude et réalisation d'un ensemble de primitives pour la satisfaction de contraintes en domaines finis

Résumé

Ce rapport présente, dans une première partie, un état de l'art détaillé sur le problème de satisfaction de contraintes en domaines finis. On se place dans le cadre de contraintes d'arité quelconque et on étudie en particulier des techniques qui exploitent le caractère actif de certaines contraintes, la structure arborescente de certains problèmes et le parcours exhaustif de l'espace de recherche. Une seconde partie décrit et évalue un ensemble de primitives qui fonctionnent au dessus du langage LE_LISP et qui permettent d'exprimer et de résoudre de tels problèmes. Ce module, qui a été initialement écrit pour intégrer des outils de satisfaction dans le générateur de système expert SMECI, est implanté à partir d'un modèle abstrait de contraintes. De ce fait, il est rendu indépendant de leur représentation interne, qu'il manipule à travers une interface fonctionnelle restreinte.

Mots clés

Problème de satisfaction de contraintes, techniques de cohérence.

Study and Implementation of a Set of Primitives for Constraint Satisfaction in Finite Domains

Abstract

The goal of this paper is twofold: first, it provides a survey on the state of the art in the field of constraint satisfaction. Within the framework of nary constraints, we especially study some technics that take advantage of the active nature of some constraints, the tree structure of some problems and the exhaustive traversal of the search space. Second, the paper describes and evaluates a module that operates over the LE_LISP language to state and solve constraint problems. The motivation for writing this module was the integration of constraint satisfaction tools in the SMECI expert system shell. It is thus implemented using an abstract model for constraints. As a consequence, it is independant from the constraint representation which is handled via functional interface

Key words

CSP, consistency technics.

1 Satisfaction de contraintes

Satisfaire un problème de contraintes n'est pas forcément une tâche complexe. Ainsi, pour certains problèmes algébriques, l'arsenal mathématique propose des outils de satisfaction efficaces. Par exemple, un système d'équations linéaires est un problème classique qui, dans le domaine des nombres réels ou rationnels, se résoud facilement par la méthode d'élimination de Gauss [52,39]. Pour des problèmes impliquant des inéquations, on dispose des résultats de la programmation linéaire et, notamment, de l'algorithme du simplexe dont la complexité reste encore faible dans la plupart des cas [52,31].

Malheureusement, dès qu'on s'attaque à des domaines discrets, ces méthodes ne sont plus applicables et les problèmes deviennent instantanément très complexes (NP-complet, en général [41,37]). La stratégie de résolution la plus naïve et la plus typiquement exponentielle consiste à passer en revue toutes les substitutions engendrées à partir des n-uplets du produit cartésien des domaines des variables non-valuées du problème et à retenir celles qui satisfont l'ensemble des contraintes. Il est clair que le coût d'une telle stratégie devient rapidement prohibitif au fur et à mesure qu'augmente le nombre des variables et le cardinal de leur domaines.

Refusant d'utiliser une méthode de recherche aveugle, l'alternative pour gagner en efficacité naît de la connaissance qu'on a initialement sur le problème ou bien de celle qu'on acquiert au cours de sa résolution. Certains algorithmes sont donc dédiés à des types de problèmes particuliers pour lesquels on sait formuler des heuristiques guidant la génération des candidats à la solution. On gagne alors en efficacité ce qu'on perd en généralité. C'est le cas, par exemple, des systèmes d'équations diophantiennes [1]. Néanmoins, en l'absence de connaissances relatives au domaine manipulé, il est impossible d'adopter une telle attitude *constructive* vis-à-vis de la solution. On peut en revanche utiliser une approche *destructive* qui permet de réduire a priori ou au cours du processus de génération, l'espace potentiel des candidats, et ce, en utilisant pour seule information l'échec résultant de l'évaluation d'une contrainte.

Les recherches et la littérature liées à ce dernier type d'approches sont sur-abondantes. Dans ce contexte, plutôt qu'une contribution fondamentale, notre étude est une investigation approfondie des méthodes existantes. Cette investigation s'accompagne d'un souci d'abstraction, d'unification et de paramétrage qui se reflète dans la formulation que nous donnons des méthodes étudiées. Par ailleurs, elle est menée de façon à accommoder plus particulièrement les points suivants :

1. *Les contraintes peuvent être d'arité quelconque.* La plupart des travaux qui ont été menés sur le problème de satisfaction de contraintes restreignent leur cadre de travail aux contraintes *binaires* [45,40,35,13]. Cette limitation a souvent pour but de clarifier l'exposé des algorithmes et surtout, de simplifier l'étude de leur complexité [47]. Cependant, rares sont

les problèmes qui, de façon naturelle, ne s'expriment qu'à l'aide de contraintes binaires. Une première possibilité est de se livrer à une petite gymnastique constante pour transformer les contraintes n-aires en contraintes binaires. La faisabilité de cette transformation est montrée dans [51]. La seconde solution, celle que nous avons choisie, consiste à élaborer un algorithme qui manipule directement des contraintes d'arité quelconque en essayant de tirer le meilleur parti des résultats acquis sur les contraintes binaires.

2. *Les contraintes peuvent avoir un caractère déductif.* De nombreux algorithmes n'exploitent les contraintes qu'à travers leur évaluation par rapport à une substitution. Ils se contentent, par exemple, de vérifier que deux régions adjacentes d'une carte n'ont pas une coloration identique. Or, certaines contraintes peuvent être utilisées de façon active, afin de déduire la valeur d'un de leurs attributs. Par exemple, sachant que la tension aux bornes d'un dipôle est égale à l'intensité qui le traverse multiplié par sa résistance et connaissant la valeur de la tension et de l'intensité, on déduit facilement la valeur de la résistance qui satisfait la contrainte. Ces déductions permettent d'une part de réduire la complexité de la recherche et d'autre part, de s'attaquer à des problèmes comportant des variables dont on ne connaît pas le domaine. Il est donc important de pouvoir mettre en œuvre ces déductions.

3. *Les problèmes peuvent avoir une faible densité de contraintes.* Certains problèmes de contraintes possèdent un graphe très dense. Par exemple, dans le fameux problème des huit reines, chaque variable représentant une colonne de l'échiquier est liée aux sept autres par trois contrainte d'inégalité, une pour interdire les mêmes horizontales et les deux autres pour interdire les mêmes diagonales. Le graphe résultant est donc plus que complet. C'est rarement le cas des systèmes physiques complexes où une variable n'a jamais d'influence *directe* sur toutes les autres variables. Les graphes associés à ces systèmes ont donc une connexité assez faible. Ces problèmes se rapprochent alors de problèmes dits faciles [12,13] pour lesquels on connaît des algorithmes efficaces.

4. *On recherche l'ensemble des solutions du problème.* La résolution d'un problème de satisfaction de contraintes découvre souvent plusieurs solutions. Certaines problématiques se contentent parfaitement d'une seule d'entre elles. Dans notre cas, on veut pouvoir les proposer dans leur ensemble afin qu'elles puissent être effeuillées et comparées par un système extérieur tel qu'un système expert[1]. Or, le fait de chercher toutes les solutions ou bien une seule influe directement sur la construction de l'algorithme. Typiquement, dans le premier cas, les tests sont effectués *a priori* sur tous les éléments des domaines des variables afin d'éliminer rapidement le maximum d'alternatives infructueuses en prévision des recherches futures. Inversement dans l'autre cas, on a plutôt tendance à compter sur l'existence d'une solution facile. C'est pourquoi on préfère se limiter aux contrôles *a posteriori*, suffisants à la seule validation de l'alternative en construction.

   D'autre part, nous verrons que certains traitements, basés sur l'analyse des échecs, permettent d'améliorer les performances de la recherche des solutions. Or, lorsqu'on souhaite trouver toutes ces solutions, on peut maîtriser la complexité et l'impact de ces traitements, mais il faut les appliquer à l'ensemble de l'espace de recherche. En revanche, pour la recherche d'une seule solution, ces traitements se révèlent souvent plus lourds mais leur nombre reste strictement proportionnel à celui des alternatives explorées. Ceci tient au fait qu'on *provoque* les échecs dans le premier cas, alors qu'on les *subit* dans le second.

[1] On trouve dans [10] un algorithme qui, pour certains graphes de contraintes, permet de trouver une solution optimale par rapport à une fonction objective et ceci sans pratiquer une recherche exhaustive des solutions. La méthode n'est cependant pas assez générale pour nous convenir

1.1 Définitions

Nous présentons ici un modèle de contraintes abstrait, indépendant du domaine ainsi que des définitions sur les problèmes de contraintes et sa satisfaction. Ces définitions sont illustrées à l'aide du petit exemple représenté sur la figure 1.1. Il s'agit d'un circuit électrique comprenant deux dipôles d_1 et d_2 connectés en série ainsi qu'un générateur de tension g. On sait que l'intensité du courant qui traverse le générateur est de 1A, qu'il peut produire à ses bornes des tensions de 100, 200 et 300V, que la résistance des dipôles est 50, 100 ou 200Ω et enfin, que la tension maximum à leurs bornes est de 100V. L'ensemble du circuit doit, bien sûr, être correct du point de vue électrique. Pour cela, il doit obéir à la loi des mailles pour les tensions, à la loi des noeuds pour les courants et à la loi d'Ohm pour les caractéristiques (voltage, résistance, courant) des dipôles. Le problème est de trouver ou de maintenir cohérentes les valeurs des différentes variables du circuit.

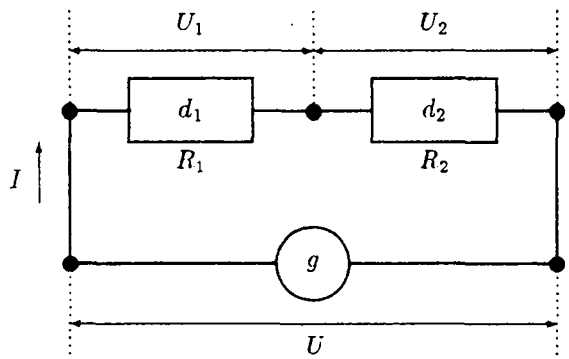

Figure 1.1 : deux dipôles en série

Notations La suite de l'exposé utilise un certain nombre de notations que nous consignons ici. On appelle T l'ensemble des types qui définissent les domaines d'opération du langage de contraintes. Pour tout $\tau \in T$, on note Ω_τ l'ensemble des objets de type τ. On distingue un objet noté $\perp$. dit *indéfini* et tel que $\forall \tau \in T, \perp \notin \Omega_\tau$. Par ailleurs, en marge des ensembles et des opération habituelles qui s'y appliquent, on se donne deux structures de données supplémentaires : les tables et les listes.

Les tables permettent d'associer des valeurs à un ensemble d'index symboliques. On note $[\![(v_1, x_1), \ldots, (v_n, x_n)]\!]$ la table constituée par les couples index-valeur (v_i, x_i) et $S[v]$ la valeur indexée par l'objet v dans la table S. Si v n'est pas présent dans la table S, la référence $S[v]$ retourne la constante $\perp$ et la référence $S[v/x]$ retourne la valeur par défaut x. Enfin, on dispose d'une fonction appelée *objets* qui retourne l'ensemble des index d'une table.

En ce qui concerne les listes, on utilise des opérations de création notées :: et ++ correspondant à la construction et à la concaténation et des opérations de destructuration notées *tete*, *queue* et pr_i qui retournent respectivement la tête, la queue et l'élément de rang i de la liste.

1.1.1 Variables et contraintes

La modélisation d'un problème de contraintes met en jeu deux types d'objets distincts : des *variables* et des *contraintes*. Nous les définissons ici comme des types de données abstraits, par la spécification de propriétés caractéristiques et d'opérations élémentaires qui leur sont applicables. Ces opérations nous permettront en particulier d'expliciter, de façon concise et proche de l'implantation effective, la plupart des mécanismes d'interprétation des contraintes.

Variables

Une variable v quelconque possède les propriétés caractéristiques suivantes :

- *Un type*. Noté $\mathcal{T}(v)$, c'est un élément de l'ensemble des types T. Les résistances R_1 et R_2 des dipôles sont de type *entier*.
- *Une valeur*. Notée $\mathcal{V}(v)$, c'est un objet de $\Omega_{\mathcal{T}(v)} \cup \{\perp\}$. Lorsque $\mathcal{V}(v) = \perp$, on dit que la valeur de v n'est pas définie. Dans notre exemple, on a $\mathcal{V}(I) = 1$.
- *Un domaine*. Noté $\mathcal{D}(v)$, c'est un ensemble d'objets de $\Omega_{\mathcal{T}(v)}$ qui représente l'ensemble des valeurs acceptables par la variable. Il peut être discret fini ou infini ou bien continu. Par exemple, on a $\mathcal{D}(R_1) = \{50, 100, 200\}$, $\mathcal{D}(U) = \{100, 200, 300\}$ ou $\mathcal{D}(U_1) =]0 \ldots 100]$
- *Un ensemble de contraintes*. Noté $\mathcal{C}(v)$, c'est l'ensemble des contraintes auxquelles participe la variable dans le problème considéré. On appelle degré d'une variable le cardinal de cet ensemble.

Afin d'alléger la notation de certaines expressions relatives à la complexité des calculs effectués, on décide de noter $\varpi(E)$ le cardinal du produit cartésien des domaines des variables d'un ensemble E. On a donc :

$$\varpi(E) = |\bigotimes_{v \in E} \mathcal{D}(v)| = \prod_{v \in E} |\mathcal{D}(v)|$$

Contraintes

Une contrainte C quelconque possède les propriétés caractéristiques suivantes :

1. *Une liste d'attributs*. On la note $\mathcal{A}(C)$. C'est la liste des variables sur lesquelles porte la contrainte. Ainsi, soient C_m et C_n les contraintes qui représentant respectivement la loi des mailles et des noeuds et C_1 et C_2 celles qui représentent la loi d'Ohm sur les dipôles D_1 et D_2. Ces contraintes sont telles que :

$$\begin{aligned} \mathcal{A}(C_m) &= (U, U_1, U_2) \\ \mathcal{A}(C_n) &= (I, I_1, I_2) \\ \mathcal{A}(C_1) &= (U_1, R_1, I_1) \\ \mathcal{A}(C_2) &= (U_2, R_2, I_2) \end{aligned}$$

2. *Une relation*. C'est un ensemble de n-uplets, noté $\mathcal{R}(C)$. Cet ensemble peut être fini ou infini. On donnera donc des définitions en extension ou en compréhension. On a, par exemple :

$$\begin{aligned}\mathcal{R}(C_m) &= \{(x,y,z) \mid x = y + z\}\\ \mathcal{R}(C_n) &= \{(x,y,z) \mid x = y = z\}\\ \mathcal{R}(C_1) &= \mathcal{R}(C_2) = \{(x,y,z) \mid x = y * z\}\end{aligned}$$

On définit l'*arité* d'une contrainte comme étant le cardinal de sa liste d'attributs. Ainsi, C_m est ternaire car $|\mathcal{A}(C_m)| = 3$.

1.1.2 Problèmes de contraintes

Un problème de contraintes s'exprime simplement par un ensemble de contraintes. Ainsi, le problème correspondant au circuit électrique est décrit par l'ensemble $P_c = \{C_m, C_n, C_1, C_2\}$. Pour un problème P quelconque, la fonction *variables* retourne l'ensemble des variables du problème :

$$\text{variables}(P) = \bigcup_{C \in P} \mathcal{A}(C)$$

La fonction $var_\top$ (resp. $var_\bot$) retourne l'ensemble des variables de P dont la valeur est (resp. n'est pas) définie :

$$\begin{aligned}\text{var}_\top(P) &= \{v \in \text{variables}(P) \mid \mathcal{V}(v) \neq \bot\}\\ \text{var}_\bot(P) &= \{v \in \text{variables}(P) \mid \mathcal{V}(v) = \bot\}\end{aligned}$$

1.1.3 Graphes de contraintes

La plupart des algorithmes opérant sur des contraintes sont très sensibles à la topologie du problème qu'ils résolvent : ils ne sont applicables ou efficaces que si le problème respecte certaines conditions de connexité comme l'absence de cycles par exemple. Lorsqu'on utilise de tels algorithmes, on préfère regarder le problème comme un graphe biparti qui masque la sémantique des relations entre les variables pour ne conserver que la topologie de ces relations, ce qu'on pourrait apparenter à l'aspect syntaxique du problème. Ce graphe est une paire $\Gamma = (N, A)$ où l'ensemble des noeuds N est l'union des variables et des contraintes du problème et l'ensemble des arêtes A est un sous-ensemble du produit cartésien des variables et des contraintes. Afin de distinguer clairement les variables des contraintes lorsqu'on figure de tels graphes, on convient de les représenter respectivement par des formes rectangulaires et circulaires. Notons pour la petite histoire que [27], un des premier papier à notre connaissance à énumérer et à classifier différents points de vue sur les contraintes, préconise une représentation graphique inverse, arguant que le rectangle fait ressortir la complexité de l'objet contrainte par rapport à l'objet variable. La figure 1.2 offre une représentation du graphe de contrainte correspondant à notre problème de dipôles.

1.1.4 Evaluation

On appelle *substitution* une table σ, indexée par des variables et telle que $\sigma[v] \in \mathcal{D}(v) \cup \{\bot\}$. La valeur $\sigma[v/\mathcal{V}(v)]$ notée aussi $\mathcal{V}_\sigma(v)$ s'appelle la valeur de v par rapport à σ. Evaluer une contrainte C par rapport à une substitution consiste à déterminer si la liste des valeurs des attributs de C par rapport à cette substitution satisfait ou non la contrainte. L'évaluation consiste donc à tester

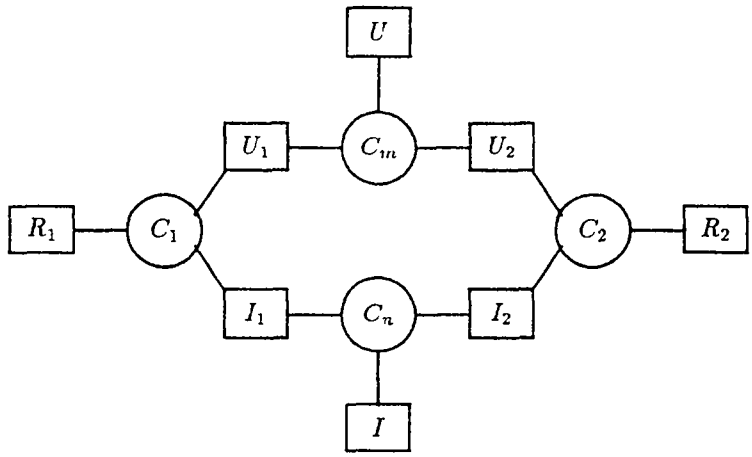

Figure 1.2 : graphe de contraintes

si cette liste est un élément de la relation associée à la contrainte. La sémantique de la fonction *evaluer* qui réalise l'évaluation est alors la suivante :

$$\text{evaluer}(C,\sigma) = \begin{cases} \top & \text{si } (\mathcal{V}_\sigma(\text{pr}_1(\mathcal{A}(C))),\ldots,\mathcal{V}_\sigma(\text{pr}_n(\mathcal{A}(C)))) \in \mathcal{R}(C) \\ \bot & \text{sinon} \end{cases}$$

L'appel à cette fonction constitue généralement le point délicat des algorithmes d'interprétation des contraintes. En effet, le test d'appartenance d'un n-uplet à une relation peut impliquer des calculs coûteux. Les calculs de complexité temporelle s'expriment donc en fonction du nombre de contraintes évaluées par l'algorithme au cours de son exécution.

1.1.5 Satisfaction

Interpréter un problème de contraintes, c'est déterminer une ou plusieurs substitutions d'un sous-ensemble des variables du problème, ces substitutions satisfaisant plus ou moins bien tout ou partie des contraintes du problème. Il y a autant de modes d'interprétations que de façons de préciser cette définition vague et très générale. Nous nous intéressons ici au problème de satisfaction.

Résoudre ce problème, c'est trouver les combinaisons de valeurs des variables *non-valuées* qui permettent de satisfaire *l'ensemble* des contraintes. Une solution est donc une substitution ς, caractérisée par :

$$\text{objets}(\varsigma) = \text{inconnues}(P)$$

$$\forall C \in P,\ \text{evaluer}(C,\varsigma) = \top$$

On appelle Φ_P l'ensemble des solutions de P au problème de satisfaction. Par exemple, considérant que les variables I_1, I_2, R_1, R_2, U_1, U_2 et U du problème P_c ont une valeur indéfinie, on a :

$$\Phi_{P_r} = \{\ [\![(I_1,1),(I_2,1),(R_1,50),(R_2,50),(U_1,50),(U_2,50),(U,100)]\!],$$
$$[\![(I_1,1),(I_2,1),(R_1,100),(R_2,100),(U_1,100),(U_2,100),(U,200)]\!]\ \}$$

Lorsque $\Phi_P = \emptyset$, on dit que P est insatisfiable. C'est le cas de notre exemple, en supposant que $\mathcal{V}(U_2) = 50$ et $\mathcal{V}(R_2) = 200$. L'objectif des algorithmes de satisfaction est de construire tout ou partie de l'ensemble Φ_P et ceci, bien sûr, avec une complexité minimale.

1.2 Enumération avec retour arrière chronologique

La recherche de solutions par un cycle de génération-test grossier oblige à compléter la génération d'une substitution avant d'entamer l'évaluation des contraintes. Il apparaît vite plus astucieux de construire progressivement la substitution et d'entrelacer cette construction avec l'évaluation des contraintes dont tous les attributs ont été substitués. Dès que l'évaluation d'une contrainte révèle un échec, il suffit de remettre en cause la valeur de la *dernière* variable substituée et de reprendre l'évaluation. Ce principe de génération progressive avec retour arrière chronologique (RAC) est utilisé par de nombreux systèmes d'interprétation non-déterministes. Il est notamment au centre de la stratégie de résolution du langage PROLOG [5]. Il constitue aussi la clef de voûte des algorithmes de satisfaction de contraintes.

Le mécanisme est simple à implanter et permet de réduire considérablement l'espace de recherche surtout lorsque l'échec intervient assez tôt dans la construction de la substitution. En effet, un échec intervenant après la substitution de k variables élimine l'examen de $\varpi(E)$ combinaisons potentielles, où E est l'ensemble des $(|\text{vars}_\perp(P)| - k)$ variables non encore substituées. Par ailleurs, pour toute implantation raisonnable d'une recherche avec RAC, la complexité en espace mémoire est identique à celle de la recherche par génération puis test.

1.2.1 Mise en œuvre du RAC

Le schéma de base de l'énumération avec RAC est détaillé sur le tableau 1.1. Il comprend les trois fonctions suivantes :

- *enumerer* est la fonction principale. Elle est chargée de retourner l'ensemble Φ des solutions d'un problème de contraintes P. On suppose que toutes les variables de P ont un domaine fini. On suppose aussi que les contraintes qui ne portent que sur des variables initialement valuées sont satisfaites. En effet, dans le cas contraire, on peut immédiatement conclure sur un ensemble de solutions vide.
- *avancer* procède récursivement à l'énumération des valeurs. A l'aide de la fonction *choisir-variable*, elle sélectionne une variable parmi les inconnues et crée un point de choix pour cette variable. En ce point, elle calcule l'ensemble des contraintes évaluables eu égard à la variable choisie et à celles dont la valeur est déjà connue puis elle déclenche l'expansion de la substitution courante pour chaque valeur du domaine de la variable. Lorsque toutes les variables ont été passées en revue, c'est qu'on a isolé une solution.
- *tester* passe en revue les contraintes évaluables dans un ordre défini par la fonction *choisir-contrainte*, ne permettant de poursuivre la recherche que si aucun échec n'est détecté.

Le schéma proposé omet délibérément tout processus de sélection sur les valeurs à énumérer. En effet, un choix est inutile voire pénalisant lorsqu'on recherche les solutions de façon exhaustive. En revanche, un schéma orienté vers la recherche de la première solution aura l'usage d'une fonction

```
algorithme enumerer(P)
soit Φ = ∅;
avancer(vars⊥(P), [[(v, V(v)) | v ∈ vars⊤(P)]], Φ);
retourner Φ;
fin enumerer
```

```
algorithme avancer(V, σ, Φ)
si V = ∅
alors Φ ← Φ ∪ {σ}
sinon soit v = choisir-variable(V)
      et E = {C ∈ C(v) | (A(C) \ {v}) ∩ V = ∅};
      pour tout x ∈ D(v)
      faire σ[v] ← x;
            si tester(σ, E) = ⊤
            alors avancer(V \ {v}, σ, Φ);
fin avancer
```

```
algorithme tester(σ, E)
tant que E = ∅
faire soit C = choisir-contrainte(E);
      si evaluer(C, σ) = ⊤
      alors E ← E \ {C}
      sinon retourner ⊥;
retourner ⊤;
fin tester
```

Tableau 1.1 : satisfaction par énumération avec RAC

choisir-valeur pour diriger la recherche sur la valeur la plus prometteuse d'un domaine. A cet effet, Dechter et Pearl [13] proposent une fonction qui, à partir d'une version simplifiée du problème initial, estime rapidement le nombre de solutions qui peuvent être engendrées avec chaque valeur du domaine. Les valeurs sont ensuite classées suivant ce résultat et celle qui présage du plus grand nombre de solutions est sélectionnée.

Par ailleurs, quelle que soit la variable substituée, *toutes* les valeurs de son domaine sont passées en revue. Des langages tels que CHIP [36] permettent de ne sélectionner que certaines valeurs d'un domaine pour éviter, par exemple, de générer des solutions redondantes. Un cas typique est le problème de coloration de cartes pour lequel le choix de deux couleurs différentes non encore utilisées pour un sommet engendre deux ensembles de solutions identiques à une permutation des couleurs près (cf. [36], p. 159). Néanmoins, les problèmes qui permettent et nécessitent une telle sélection ne sont pas les plus nombreux ni les plus intéressants pour une résolution par un langage de contraintes. De plus, la programmation de cette sélection n'est envisageable que sur des cas simples et très symétriques comme celui que nous venons d'évoquer.

1.2.2 Premières optimisations

Dans la version la plus simpliste de l'algorithme, les fonctions de sélection *choisir-variable* et *choisir-contrainte* retournent un élément quelconque de l'ensemble sur lequel elles sont appliquées. Un premier raffinement consiste à choisir véritablement ces éléments, c'est à dire, à mettre un peu d'intelligence dans la détermination de leur enchaînement, ceci afin de réduire au maximum le nombre de contraintes à évaluer pour mener la recherche à terme.

Choix des variables

Il y a autant de façons d'énumérer les inconnues d'un problème que de permutations de celles-ci. Ces permutations, ou séquences, sont souvent loin d'être équivalentes du point de vue de la complexité de la recherche. Pour s'en convaincre, regardons l'exemple trivial suivant. Il est composé de deux contraintes C_1 et C_2 telles que :

$$\begin{aligned} \mathcal{A}(C_1) &= (a, b) \\ \mathcal{A}(C_2) &= (a, c) \\ \mathcal{R}(C_1) &= \mathcal{R}(C_2) = \{(x, y) \mid x = y\} \end{aligned}$$

Les variables a, b et c sont telles que $\mathcal{D}(a) = \{1, 2, 3\}$ et $\mathcal{D}(b) = \mathcal{D}(c) = \{1, 2\}$. Le premier arbre de la figure 1.3 est celui parcouru en utilisant l'ordre d'énumération $c \prec b \prec a$ (les traits pointillés caractérisent les feuilles des alternatives infructueuses). Pour cet ordre, l'évaluation des deux contraintes est repoussée aux feuilles de l'arbre. On est alors placé dans le pire des cas pour l'énumération dans lequel l'arbre n'est jamais élagué. On évalue alors un nombre n de contraintes tel que :

$$\varpi(\mathrm{vars}_\perp(P)) \leq n \leq |P| \cdot \varpi(\mathrm{vars}_\perp(P))$$

En revanche, en choisissant l'ordre $a \prec b \prec c$, on peut évaluer C_1 dès que b est substitué ce qui permet d'éliminer rapidement les deux tiers des alternatives. On supprime ainsi autant de tests de C_2, comme le montre le deuxième arbre de recherche de la figure 1.3. L'importance de ce gain se révèle a fortiori sur un problème aux dimensions réelles qui implique un grand nombre de variables. Ainsi, nous avons pu constater, sur une centaine de permutations, que le nombre

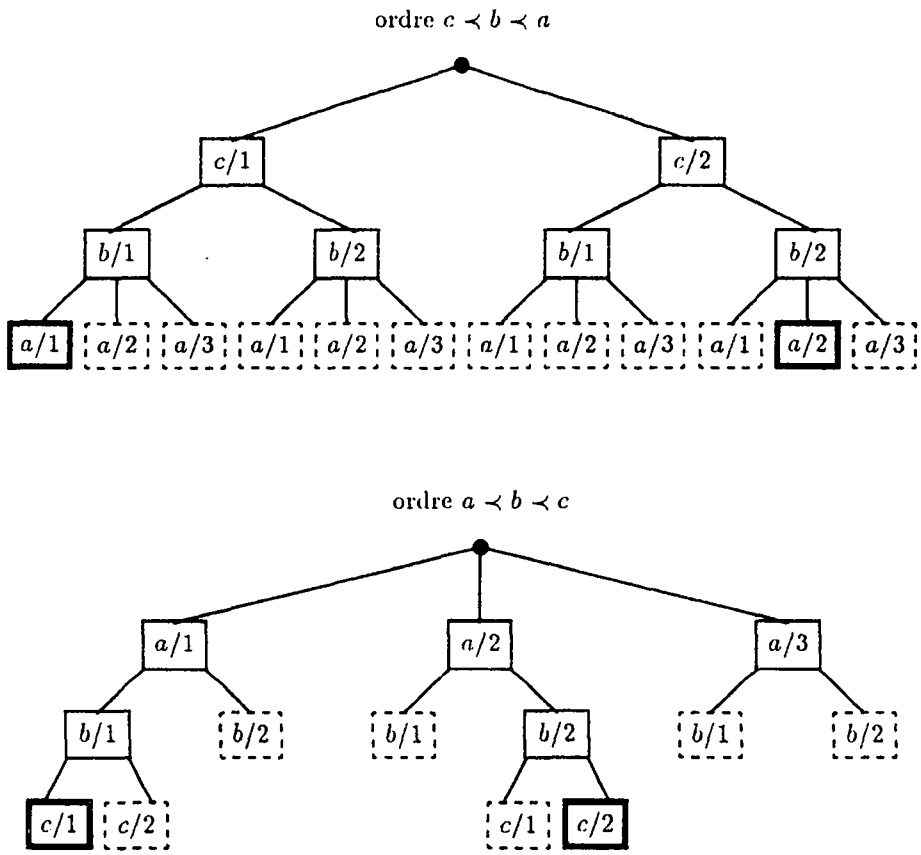

Figure 1.3 : influence de l'ordre d'énumération

d'évaluations nécessaires à la résolution du problème du zèbre (cf. § 2.4) pouvait varier dans un rapport de 1 à 7000.

Effectuer un choix parmi les permutations est donc un problème crucial, rendu difficile par le nombre élevé des alternatives ($|vars_\perp(P)|!$ pour un problème P) et l'absence de critère de comparaison absolu. Le principe intuitif généralement utilisé pour construire une séquence est le principe dit de l'échec d'abord (*first fail principle* [35]) qui recommande de provoquer le plus d'échecs possible, le plus tôt possible. L'effet attendu est l'élimination de grandes parties de l'espace de recherche en détectant au plus tôt les incohérences.

Les heuristiques construites sur ce principe sont peu coûteuses et permettent d'éviter l'utilisation de séquences catastrophiques. Elles induisent une relation d'ordre partielle sur les variables en se référant à des propriétés locales telles que leur degré ou encore le cardinal de leurs domaines. D'autres procédures d'ordonnancement utilisent des critères de nature plus globale, relatifs par exemple à la topologie du problème [25,58].

Lorsque le critère choisi est susceptible d'évoluer entre deux points de choix, il est nécessaire de pouvoir décider dynamiquement de la prochaine variable à énumérer. Ce cas de figure survient, par exemple, lorsque le domaine d'une variable est réduit par l'application d'une procédure de filtrage entre les points de choix. Soulignons à ce propos que le choix de la variable ayant le domaine le plus restreint constitue la meilleure heuristique dynamique connue. Elle est étudiée analytiquement dans [50,47], empiriquement dans [55,11], étendue dans [49] et enfin utilisée dans les systèmes CHIP [36] et CHARME [48].

Inversement, lorsqu'on utilise un critère stable comme le degré des variables, il est préférable d'établir l'ordre de façon statique, avant l'énumération. On évite ainsi, en chaque point de choix, le calcul de la prochaine variable et de l'ensemble des contraintes évaluables. Ce gain n'est pas négligeable puisque ces calculs se répètent potentiellement de façon exponentielle au cours de la recherche. D'après les résultats expérimentaux donnés dans [11], il semble que la meilleure heuristique statique est celle dite de la *cardinalité maximum*. Elle consiste à choisir une première variable au hasard puis à élire itérativement pour suivante la variable non encore placée qui est liée au plus grand nombre de variables parmi celles qui sont déjà placées. Cette heuristique vise à trouver rapidement un ordre dont la largeur de bande [58] est minimum.

Comme toutes heuristiques, celles que nous venons de décrire n'ont pas valeur de lois universelles. L'exemple précédent en est la preuve : c'est l'ordre $a \prec b \prec c$ donne le meilleur résultat alors que la variable a possède le domaine le plus large.

Choix des contraintes

En second lieu, il faut s'intéresser à l'ordre dans lequel on évalue les contraintes relatives à un point de choix. Ici encore, le principe de l'échec d'abord est applicable : dans un ensemble donné, on choisira en priorité la contrainte dont l'évaluation a la plus forte probabilité d'échouer, afin d'éviter une évaluation inutile des autres contraintes de l'ensemble [35]. On peut s'aider ici d'une appréciation qualitative plus ou moins subjective concernant le degré de satisfiabilité des contraintes. Il est, par exemple, plus probable qu'un couple (x, y) quelconque satisfasse la relation $x \neq y$ plutôt que $x = y$. Laurière utilise ce type de classification dans [38]. En l'absence de tout renseignements sur la relation, on peut supposer que la contrainte la plus difficile à satisfaire est celle d'arité maximum. Par ailleurs, lorsque la séquence d'énumération des variables est prédéfinie, on a tout intérêt à ordonner les contraintes à évaluer avant de commencer l'énumération.

1.2.3 Schéma de RAC avec ordre statique

Le tableau 1.2 montre un nouveau schéma qui modifie le précédent afin de pouvoir utiliser un ordre d'énumération prédéfini. Les ensembles de variables et de contraintes utilisés précédemment sont remplacés par des listes qui sont parcourues de façon séquentielle. Ces listes sont construites à l'aide de deux fonctions d'ordonnancement statiques : *ordonner-variables* et *ordonner-contraintes*. En l'état, les schémas des tableaux 1.1 et 1.2 sont loins d'être une panacée. Nous soulignons clairement leurs défauts dans ce qui suit.

1.2.4 Pathologie

Malgré ses attraits, la réputation du RAC est ternie par de trop fréquentes crises de gaspillage aiguës. Son défaut fondamental réside dans sa nature chronologique, c'est à dire dans le fait qu'il incrimine aveuglément la variable la plus récemment substituée. Ainsi, en essayant de corriger un échec sans tenir compte de ses causes, le RAC est amené à remettre en question des choix qui ne devraient pas l'être et à reproduire ce cas d'échec ultérieurement.

Plus formellement, soit $v_1 \prec v_2 \prec \ldots \prec v_n$ la séquence des variables à énumérer. On distingue principalement trois manifestations de l'inefficacité de l'algorithme en observant les phases de génération et de retour arrière :

1. *Le RAC ne recherche pas les causes d'un échec.* Supposons que, pour une substitution donnée, l'évaluation d'une contrainte C échoue pour toutes les valeurs d'un de ses attributs v_j. Ceci amène la remise en cause de la valeur substituée à la variable v_{j-1}. Or, il arrive souvent que $v_{j-1} \notin \mathcal{A}(C)$. Par conséquent, la valeur de v_{j-1} n'intervient pas dans l'échec rencontré et sa remise en cause lors du retour arrière n'est que pure perte. Le nombre de retours arrière qu'on sait a priori être inutiles dépend de l'écart au sein de la séquence entre les deux derniers attributs substitués de C. Il est égal à :

$$\varpi(]\max(\mathcal{A}(C) \setminus \{v_j\}), \ldots, v_{j-1}])$$

2. *Le RAC reproduit ses erreurs.* Lorsque l'évaluation d'une contrainte C échoue pour une substitution σ, cela signifie que la substitution partielle $\varsigma = [\![(v_i, x_i) \in \sigma \mid v_i \in \mathcal{A}(C)]\!]$ ne peut faire partie d'aucune solution. Or, l'algorithme ne garde pas de traces des échecs qu'il rencontre. Lors de la phase de génération, il va donc construire de nouvelles substitutions contenant ς et menant inévitablement à un échec. Ici encore, les essais inutiles sont d'autant plus nombreux que les variables substituées par ς ne sont pas consécutives dans la séquence d'énumération. En effet, au sein de l'arbre de recherche, une substitution ς donnée apparaît un nombre de fois égal à :

$$\varpi(\{v \in]\min(\text{objets}(\varsigma)), \ldots, \max(\text{objets}(\varsigma))[\mid v \notin \text{objets}(\varsigma)\})$$

3. *Le RAC ne sait pas conserver des solutions partielles.* Lorsqu'à partir d'une variable v_j, le retour arrière doit remonter sur v_i pour essayer de valider une contrainte C, la substitution partielle composée des variables comprises entre v_i et v_j est perdue. Or, l'effort qui a été nécessaire à la détermination de cette substitution a pu être important et sera reproduit inutilement lors de la redescente à partir de v_i.

1.2.5 Thérapeutique

Parmi les différentes mesures prophylactiques élaborées pour le retour arrière chronologique, beaucoup sont issues des recherches sur les systèmes de type TMS [20,15,16,17] et sur l'implantation

```
algorithme enumerer(P)
soit Φ = ∅
et V = ordonner-variables(vars⊥(P));
avancer(V, assembler(V), [[(v, V(v)) | v ∈ vars⊤(P)]], Φ);
retourner Φ;
fin enumerer
```

```
algorithme assembler(V)
si V = ∅
alors retourner ∅
sinon soit E = {C ∈ C(tete(V)) | A(C) ∩ queue(V) = ∅};
      retourner ordonner-contraintes(E)::assembler(queue(V));
fin assembler
```

```
algorithme avancer(V, L, σ, Φ)
si V = ∅
alors Φ ← Φ ∪ {σ}
sinon pour tout x ∈ D(tete(V))
      faire σ[v] ← x;
           si tester(σ, tete(L)) = ⊤
           alors avancer(queue(V), queue(L), σ, Φ);
fin avancer
```

```
algorithme tester(σ, L_i)
tant que E ≠ ∅
faire si evaluer(tete(L_i), σ) = ⊤
     alors E ← queue(L_i)
     sinon retourner ⊥;
retourner ⊤;
fin tester
```

Tableau 1.2 : satisfaction par énumération avec ordre fixe et RAC

d'un retour arrière intelligent pour le langage PROLOG [4.6]. Ces traitements s'appliquent au cours de la recherche, au fur et à mesure de l'occurrence des contradictions. Ils aident à mieux y remédier et à éviter qu'elles ne se reproduisent.

D'autres traitements sont plus spécifiques aux problèmes de contraintes. Ils tirent parti du caractère purement local de l'évaluation des contraintes pour détecter des contradictions au sein des parties d'un problème. Ils peuvent ensuite exploiter plus largement les informations recueillies localement en les propageant à l'ensemble du problème. Il est ainsi possible de réduire l'espace de recherche avant l'énumération et de prévenir l'occurrence de nombreux cas d'échec. Dans les deux sections suivantes, on expose des techniques liées respectivement à ces deux approches et on propose de tirer parti des dernières dans notre schéma d'énumération.

1.3 Exploiter le passé

Une démarche générale pour pallier les problèmes du retour arrière chronologique consiste à prendre en compte intelligemment les échecs passés. Les algorithmes issus de cette veine se regroupent sous l'appellation anglophone de *look-back algorithms*. En poussant cette démarche à l'extrême, on est amené à construire un nouveau type de retour arrière qualifié de *dirigé par les dépendances*. Pour chaque maladie du RAC que nous avons isolée dans la section précédente, nous proposons maintenant divers remèdes.

1.3.1 Rechercher les causes d'un échec

Il s'agit de trouver une variable v_i impliquée dans l'échec de la contrainte C et dont la remise en cause directe n'éliminera aucune solution. Cette variable est l'attribut de C qui a le rang maximum dans la séquence, exception faite, bien sûr, de la variable courante v_j. On a donc :

$$v_i = \max(\mathcal{A}(C) \setminus \{v_j\})$$

On peut généraliser cette détermination à un ensemble de contraintes E pour lequel aucune valeur de v_j ne serait à même de satisfaire la totalité des éléments de E, c'est à dire :

$$\forall x \in \mathcal{D}(v_j)\,,\ \exists C \in E\,,\ \text{evaluer}(C, \sigma \cup [\![(v_j, x)]\!]) = \bot$$

Une approximation rapide pour un point de retour est alors [8] :

$$v_i = \max(\bigcup_{C \in E} \mathcal{A}(C) \setminus \{v_j\}) \tag{1.1}$$

Toutefois, il est possible de calculer un meilleur point de retour avec un coût comparable au précédent. A cet effet, on définit une fonction *echec-min* qui détermine, pour un ensemble de contraintes E et une variable v, la variable de rang minimum parmi l'ensemble des variables de rang maximum des attributs de chaque contrainte de E, exception faite de v. De façon plus concise, on a :

$$\text{echec-min}(E, v) = \min_{C \in E}(\max(\mathcal{A}(C) \setminus \{v\}))$$

Muni de cette fonction, on peut calculer, pour chaque valeur du domaine de v_j, la variable de rang minimum sur laquelle on peut remonter et finalement, la variable recherchée est celle de rang maximum parmi celles qu'on a ainsi trouvé. Soit E, l'ensemble des contraintes évaluables au point de choix v_j, on a :

$$v_i = \max_{x \in \mathcal{D}(v_j)} (\text{echec-min}(\{C \in E \mid \text{evaluer}(C, \sigma \cup [\![(v_j, x)]\!]) = \bot\}, v_j)) \tag{1.2}$$

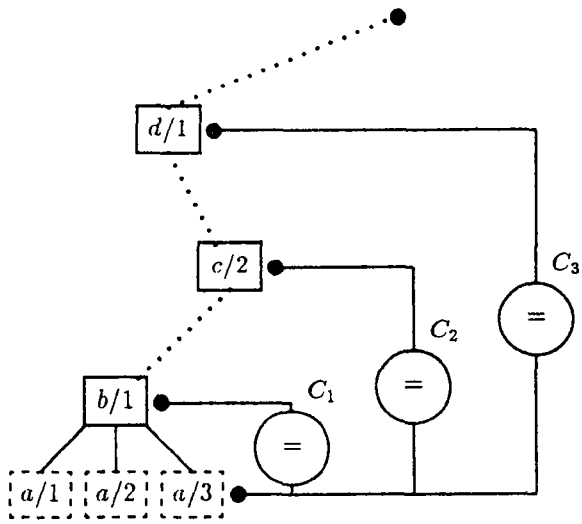

Figure 1.4 : sélection intelligente d'un point de retour

Cette équation est implantée implicitement par l'algorithme de *backjumping* de Gaschnig [29]. Pour illustrer son utilisation, on reprend les données de l'exemple de la figure 1.3 auxquelles on ajoute une variable d et une contrainte C_3 associée à la même relation d'égalité que C_1 et C_2. On s'intéresse à l'alternative représentée sur la figure 1.4. Pour cette alternative, toutes les valeurs de a échouent. Les échecs sont les suivants :

$$\left.\begin{array}{l}\text{evaluer}(C_2, \sigma \cup [\![(a,1)]\!]) = \bot\end{array}\right\} \Rightarrow \text{echec-min}(\{C_2\}, a) = c$$

$$\left.\begin{array}{l}\text{evaluer}(C_1, \sigma \cup [\![(a,2)]\!]) = \bot \\ \text{evaluer}(C_3, \sigma \cup [\![(a,2)]\!]) = \bot\end{array}\right\} \Rightarrow \text{echec-min}(\{C_1, C_3\}, a) = d$$

$$\left.\begin{array}{l}\text{evaluer}(C_1, \sigma \cup [\![(a,3)]\!]) = \bot \\ \text{evaluer}(C_2, \sigma \cup [\![(a,3)]\!]) = \bot \\ \text{evaluer}(C_3, \sigma \cup [\![(a,3)]\!]) = \bot\end{array}\right\} \Rightarrow \text{echec-min}(\{C_1, C_2, C_3\}, a) = d$$

D'après l'équation 1.2, on déduit que le retour arrière peut remonter jusqu'au point de choix mis en place pour la variable $\max(\{c, d, d\}) = c$ au lieu de s'arrêter à b. Notons que, dans ce cas, le calcul du point de retour par l'équation 1.1 n'aurait été d'aucun secours puisque $max(\{d, c, b\}) = b$.

Du point de vue de l'implantation, lorsque l'ordre d'énumération est fixé, la fonction *ordonner-contraintes* peut utiliser l'ordre sur les contraintes introduit par *echec-min* dont le résultat est invariable pour un ordre d'énumération donné. Lors de l'évaluation, la première contrainte qui va échouer pour une valeur x_j va servir à élire le point de retour pour cette valeur. Lorsque toutes les valeurs échouent, il reste seulement à déterminer le point de retour de rang maximum. Un système d'échappement permet alors de revenir rapidement sur un point de choix donné.

Le calcul de point de retour que nous venons d'évoquer est très peu coûteux. Par conséquent, lorsque l'énumération peut rencontrer des échecs dont les causes sont éloignées du point de choix

courant, il est toujours profitable de substituer au retour arrière classique un retour arrière intelligent, ceci quelle que soit la complexité des problèmes qu'on attaque.

1.3.2 Eviter de reproduire des erreurs

Le retour arrière intelligent permet de pallier les effets néfastes du RAC, mais ne supprime pas ses causes. Ainsi, afin de ne pas réitérer l'expansion d'alternatives menant à une impasse, il est nécessaire de mémoriser les ensembles de valeurs incompatibles. Cette mémorisation s'apparente à un processus d'apprentissage simple. Elle répond, en effet, aux caractéristiques suivantes [9] :

- Elle se présente sous forme d'un module indépendant du système de résolution.
- Ce module observe le déroulement de l'algorithme et enregistre des informations découvertes au cours de la recherche.
- Les performances de l'algorithme sont améliorées lorsqu'il s'exécute conjointement avec le module d'apprentissage.
- La connaissance spécifique acquise par le module au cours de la résolution d'un problème permet d'obtenir de meilleures performances lors d'une nouvelle exécution du même problème.

Lorsqu'un échec se présente, le module va enregistrer une substitution partielle conflictuelle (SC) qui caractérise cet échec. Pour sa part, lors de la phase d'expansion, l'algorithme va vérifier qu'il ne reproduit pas une des situations enregistrées. Ce mécanisme permet progressivement d'expliciter les contraintes implicites que recèle le problème. Par exemple, le fait d'enregistrer la substitution $[\![(d,1),(c,2)]\!]$ (cf. figure 1.4) équivaut à ajouter au problème une contrainte C_{d1c2} définie par :

$$\begin{aligned} \mathcal{A}(C_{d1c2}) &= (d,c) \\ \mathcal{R}(C_{d1c2}) &= \{(x,y) \neq (1,2)\} \end{aligned}$$

et explicite ainsi partiellement la contrainte implicite $d = c$, issue de C_3 $(d = a)$ et C_2 $(a = c)$, par transitivité. Steele suggère d'ailleurs dans [54] d'implanter simplement les SC sous forme de contraintes à part entière qu'on ajoute progressivement au graphe. L'inconvénient de cette pratique est double. D'une part, le coût de la représentation et de la manipulation d'une contrainte est lourd. D'autre part, on augmente sans cesse la densité du graphe et, comme nous l'avons déjà mentionné, certains algorithmes efficaces reposent sur la faible connexité du problème.

L'enregistrement des SC doit suivre une politique prudente car d'une part, le processus d'expansion se trouve ralenti linéairement en fonction de leur nombre et d'autre part l'espace mémoire court le risque d'être rapidement saturé. Il faut donc trouver un ratio convenable entre le gain de temps apporté par la quantité de SC enregistrées et le surcroît de complexité en temps et en espace qu'elles imposent au système. Heureusement, la sémantique des SC va dans le bon sens à savoir que, moins elles impliquent de variables, plus elles tendent à réduire l'espace de recherche.

Reprenons l'exemple de la figure 1.4. Après avoir découvert qu'aucune valeur du domaine de a n'était satisfaisante, on peut enregistrer comme conflictuelle l'ensemble de l'alternative jusqu'à b. Toutefois, cela n'est d'aucune utilité puisque la mécanique du retour arrière garantit que cette alternative ne sera pas reproduite. Les seules variables véritablement significatives de l'échec sont celles qui participent avec a à une contrainte explicite. On peut ainsi limiter la SC à $[\![(d,1),(c,2),(b,1)]\!]$. Une étude plus approfondie permet encore de diviser cette substitution en deux sous-substitutions $[\![(d,1),(c,2)]\!]$ et $[\![(c,2),(b,1)]\!]$ qui constituent les *SC minimales* [4] et qui promettent l'élimination d'un espace de recherche maximal. Ici encore, on doit veiller à trouver un bon ratio entre la qualité des SC enregistrées et le temps nécessaire à leur détermination.

Indépendamment de l'énergie qu'on investit à les décomposer, on peut décider de ne prendre en compte que les SC ayant une cardinalité donnée; on se limite le plus souvent à celles qui ne portent que sur *une* ou *deux* variables. Notons que lorsqu'elles sont réduites à une seule variable, les SC peuvent être enregistrées en retirant simplement la valeur prohibée du domaine de la variable. De ce fait, elles n'alourdissent pas la recherche ni ne modifient la connexité du graphe de contraintes[2]. On pourra trouver dans [9] et [8] une évaluation expérimentale des résultats obtenus par de tels mécanismes. On constate que leur apport ne s'avère réellement probant que pour des problèmes difficiles.

On peut reprocher à la technique que nous venons d'évoquer d'accomplir un apprentissage qui soit trop subordonné à la recherche. En effet, il n'y a aucune indépendance du processus d'apprentissage vis à vis du parcours de l'arbre des alternatives. Les SC sont imposées au processus qui, en présence d'une SC de taille supérieure au maximum fixé, peut soit décider de l'ignorer en risquant de perdre le profit d'une SC intéressante, soit de s'attaquer à sa décomposition risquant alors de travailler en pure perte. Une analyse découplée de la recherche permet de générer explicitement les SC désirées au lieu de les rechercher, ce qui s'avère moins coûteux. L'inconvénient d'une telle analyse est qu'elle s'applique sur l'ensemble du problème alors qu'un apprentissage progressif ne va considérer que les SC utiles à la génération de la prochaine solution. Ceci met en valeur l'intérêt spécial de cette dernière technique pour la recherche rapide de la première solution.

1.3.3 Conserver les solutions partielles

On souhaite à présent éviter la destruction des sous-ensembles cohérents d'une substitution lorsqu'on accomplit un retour arrière. Regardons un exemple d'une telle destruction sur la figure 1.5. Aucune substitution de v_n ne satisfait la contrainte d'égalité vis à-vis de v_1. Ceci va forcément entraîner une remise en cause de la valeur de la variable v_1. Ce faisant, on va détruire la substitution partielle ς qui résolvait un sous-problème difficile et on va devoir recommencer sa résolution lors de la redescente depuis v_1. Cette constatation nous conduit à abandonner purement et simplement l'aspect chronologique du retour arrière au profit d'un mécanisme dirigé par les données : le retour arrière dirigé par les dépendances (RADD).

Lorsque l'évaluation d'une contrainte C échoue, le RADD va rechercher les causes de l'échec. Cette recherche peut remonter le long d'une chaîne causale entre les variables, justifiant alors pleinement l'appellation du RADD. Dans notre cas, les variables à incriminer sont simplement les attributs de C. Il suffit alors de changer la valeur substituée à un des attributs et de reprendre la recherche.

Afin d'éviter de reproduire indéfiniment les mêmes substitutions, chose qui était garantie auparavant par la remise en cause chronologique des valeurs, il est maintenant essentiel de mémoriser *toutes* les substitutions conflictuelles rencontrées. Ceci alourdit considérablement la recherche au fur et à mesure de sa progression et rend le RADD totalement inadapté à une recherche exhaustive des solutions. En revanche, il est largement utilisé par les systèmes de contraintes tels que ceux décrits dans [54] ou [42], qui se contentent d'une seule solution. Par ailleurs, il occupe une place centrale dans le fonctionnement des TMS tels que celui de Doyle [20] ou l'ATMS de DeKleer [15,18]. Enfin, d'autres utilisations du RADD sont décrites dans [59] et [53].

[2]Ceci pose néanmoins un problème lorsque le domaine possède une représentation compréhensive comme par exemple un intervalle d'entiers représenté par ses bornes inférieure et supérieure. On est alors amené à ignorer l'élimination ou bien à changer la représentation du domaine.

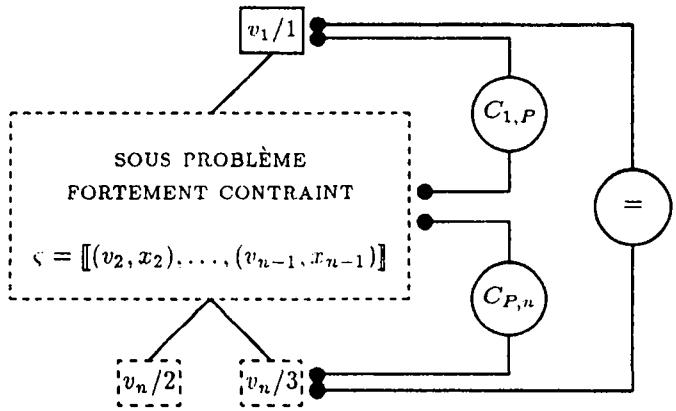

Figure 1.5 : sous-problème résolu n fois

1.3.4 Critique

L'ensemble des techniques que nous venons d'évoquer permettent d'endiguer les errements du RAC. Elles sont éprouvées et utilisées par de nombreux systèmes. Nous leur reprochons pourtant d'être trop particulièrement adaptées à l'accession rapide à la première solution du problème. Pour une recherche exhaustive, nous nous sommes tournés vers une approche plus globale des traitements.

1.4 Lire dans l'avenir

Mieux vaut prévenir que guérir. C'est pourquoi nous nous tournons à présent vers une démarche préventive visant à écarter des cas d'échec qu'on sait être inéluctables. En appliquant une phase de filtrage préliminaire au problème, on va imposer un certain niveau de cohérence partielle avant de chercher à obtenir la cohérence globale. Ce filtrage *a priori* réduit fortement l'espace de recherche et minimise la probabilité de découvrir tardivement des impasses. Lorsqu'en cours d'énumération, la création d'un point de choix alterne avec le filtrage sur le sous-problème restant à résoudre, on dit que l'algorithme est de type *look-ahead*.

1.4.1 Degrés de cohérence d'un problème

Mackworth propose dans [40] des algorithmes qui permettent d'isoler efficacement certaines valeurs ou combinaisons de valeurs qu'on sait être incompatibles avec l'ensemble des contraintes. Les incompatibilités sont détectées à plusieurs niveaux, en considérant des sous-ensembles de plus en plus importants du graphe de contraintes. L'exposé de Mackworth ainsi que certaines révisions de ses algorithmes [44,34] se limite au traitement des contraintes binaires et s'intéresse à trois niveaux de cohérence successifs[3].

[3]Les graphes de contraintes binaires évitent la dualité du type des noeuds. Les seuls noeuds du graphe sont des variables, les contraintes étant représentées par des arcs étiquetés. Ceci justifie les appellations

1. *Cohérence au niveau des noeuds.* Un graphe n'est pas cohérent au niveau de ses noeuds tant qu'il existe une variable dont un des éléments du domaine ne satisfait pas toutes les contraintes unaires auxquelles elle est liée. On établit facilement ce niveau de cohérence en retirant les valeurs indésirables du domaine des variables. La définition des domaines absorbe alors l'ensemble des contraintes unaires du problème dont on se débarrasse ainsi à peu de frais.

2. *Cohérence au niveau des arcs.* Un graphe n'est pas cohérent au niveau de ses arcs tant qu'il existe un attribut d'une contrainte dont le domaine recèle une valeur qui n'entre en jeu dans aucune des substitutions satisfaisant la contrainte. Obtenir l'arc-cohérence consiste donc à trouver les SC de longueur 1 les plus évidentes. Ce niveau de cohérence est le plus utilisé et, de ce fait, il s'approprie presque exclusivement l'appellation de niveau de *cohérence locale* d'un problème [57]. Notons aussi que l'opération qui consiste a établir l'arc-cohérence (AC) pour une contrainte possède un équivalent dans la théorie des bases de données relationnelles qui est l'opération de semi-jointure [28].

3. *Cohérence au niveau des chemins.* C'est l'extension naturelle de la définition précédente pour deux variables quelconques du problème. Ainsi, on dit qu'un graphe n'est pas cohérent au niveau de ses chemins tant qu'il existe un couple de variables et un chemin de contraintes qui les relie tels qu'un des éléments du produit cartésien des domaines de ces variables n'entre en jeu dans aucune des substitutions satisfaisant le chemin. Ce niveau de cohérence implique donc une recherche de SC de longueur 2.

La figure 1.6 représente une carte à colorier comprenant quatre régions. Ces régions disposent chacune de la même palette de couleurs et sont contraintes à être coloriées différemment de leurs régions adjacentes. On peut vérifier que cette carte est cohérente au niveau de ses arcs et des ses chemins. Notons qu'elle n'en a pas pour autant de solutions.

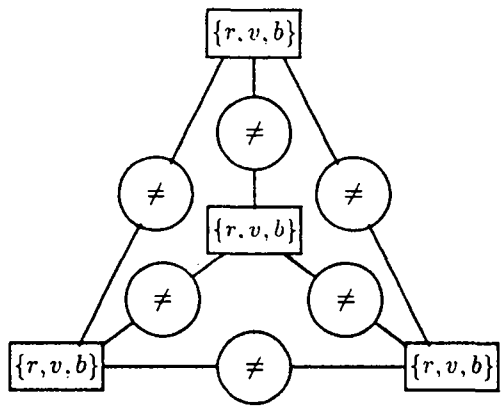

Figure 1.6 : carte cohérente pour ses arcs et ses chemins

subséquentes de *noeud*, *arc* et *chemin*.

La notion de cohérence au niveau des arcs et des chemins se généralise facilement à celle de cohérence au niveau de sous-ensembles de variables d'ordre k. En exploitant successivement cette k-cohérence pour k variant de 1 à $|\text{vars}(P)|$, on peut isoler les n-uplets satisfaisant l'ensemble des contraintes [26]. Bien entendu, la complexité nécessaire pour parvenir à une cohérence de niveau k augmente avec k. Cette complexité reste polynômiale de faible degré pour $k = 2$ et $k = 3$ [41,44].

En ce qui nous concerne, nous nous bornerons à vérifier l'arc-cohérence[4] des problèmes qui se traduit, dans notre formalisme, par la vérification de la proposition :

$$\forall C \in P\,,\ \forall v \in \mathcal{A}(C)\,,\ \forall x \in \mathcal{D}(v)\,.\ \exists \sigma\,,\ \text{evaluer}(C, \sigma \cup [\![(v,x)]\!]) = \top$$

Ce choix se justifie par le fait que l'obtention d'un niveau de cohérence supérieur à 1 requiert de matérialiser les SC découvertes en ajoutant des contraintes au problème. Or, comme nous l'avons évoqué précédemment, cet ajout est fortement susceptible d'alourdir la mémoire et la souplesse du processus de substitution.

1.4.2 Mise en oeuvre de l'arc-cohérence

La trame universelle de l'algorithme qui permet d'obtenir l'arc cohérence est celle donnée par Mackworth dans [40] sous le nom d'AC-3 :

```
begin
  Q ← {(i,j) | (i,j) ∈ arcs(G), i ≠ j}
  while Q ≠ ∅ do
    begin
      select and delete any arc (k,m) from Q;
      if REVISE((k,m))
      then Q ← Q ∪ {(i,k) | (i,k) ∈ arcs(G), i ≠ k, i ≠ m}
    end
end
```

La fonction *REVISE*, qui s'applique sur un arc, a comme effet de bord d'éliminer du domaine d'un noeud les valeurs qui ne sont compatibles avec aucune valeur de l'autre noeud de l'arc. Elle retourne un booléen qui signale si le domaine a été réduit ou non. Cet algorithme est simple et possède surtout un caractère très intuitif. On se figure facilement, la représentation graphique aidant, que la réduction du domaine d'une variable peut entraîner l'élimination de valeurs chez les attributs adjacents. Ces éliminations peuvent, à leur tour, en déclencher d'autres, propageant ainsi les incohérences détectées localement dans l'ensemble du réseau.

Le tableau 1.3 montre notre adaptation de cet algorithme à des contraintes d'arité quelconque. On fournit en paramètre de la fonction *etablir-coherence* l'ensemble P des contraintes pour lesquelles on va vérifier la cohérence. La table Δ, remplie par effet de bord, rapporte les implications de l'arc-cohérence sur le domaine des variables. Notons que si on est amené à réduire un domaine à l'ensemble vide, on peut immédiatement conclure que le problème n'a pas de solutions. Dans ce cas, on retourne la valeur $\bot$. On a recours à une table nommée *origine* pour associer, à chacune des contraintes à examiner, l'attribut dont le domaine a été modifié et qui donc a provoqué l'examen de la contrainte.

Un raffinement supplémentaire de cette fonction consiste à gérer la liste des contraintes à examiner P comme un agenda dont on réarrange le contenu après chaque itération. La contrainte

[4] Nous garderons cette appellation d'arc-cohérence pour les contraintes n-aires, bien qu'il s'agisse là d'un abus de langage. Freuder définit ce type de cohérence dans [24] et le qualifie de $(1,j)$-cohérence, avec $j = |\mathcal{A}(C)| - 1$.

à examiner est alors la première de l'agenda. On obtient bien sûr les stratégies de profondeur d'abord en gérant l'agenda comme une pile et de largeur d'abord en le gérant comme une file. Cette idée apparaît dans [32] mais aucune indication n'est donnée quant à son impact sur les performances de l'algorithme. Il nous paraît préférable du point de vue de l'efficacité et de la simplicité de mettre encore une fois à contribution le principe de l'échec d'abord en choisissant en priorité les contraintes les plus difficiles à satisfaire. On diminue ainsi la probabilité de déclencher plusieurs fois un filtrage à partir de la même contrainte. A cet effet, on peut réutiliser la fonction *choisir-contrainte* élaborée pour l'énumération.

```
algorithme etablir-coherence(P, Δ)
soit origine = [];
et tampon = [];
tant que P ≠ ∅
faire soit C ∈ P
       et V = filtrer(C, origine[C]. Δ. tampon);
       si V = ⊥
       alors retourner ⊥
       sinon P ← P \ {C};
             pour tout v ∈ V
             faire pour tout K ∈ C(v)
                   faire si K ∈ P
                         alors si origine[K] ≠ v
                               alors origine[K] ← ⊥;
                         sinon si K ≠ C
                               alors origine[K] ← v;
                                     P ← P ∪ {K};
retourner ⊤;
fin etablir-coherence
```

Tableau 1.3 : application de l'arc-cohérence

La fonction *filtrer* joue le rôle de la fonction *REVISE*. Comme son nom l'indique, elle établit la cohérence locale d'une contrainte C en filtrant les domaines de ses attributs. Les domaines filtrés sont lus et enregistrés dans la table Δ[5]. Par ailleurs, *filtrer* sauvegarde les valeurs éliminées dans la table *tampon*. Ceci permet de l'utiliser facilement dans des algorithmes non-déterministes qui nécessitent la sauvegarde et la restauration d'environnements. Nous nous intéressons maintenant plus en détail à cette fonction de filtrage qui reçoit généralement peu d'attention dans l'ensemble de la littérature. Sa programmation s'avère pourtant importante à l'égard de la complexité puisqu'elle est responsable de l'évaluation des contraintes.

[5] A l'instar de la notation $\mathcal{V}_\sigma(v)$ qui désigne la valeur d'une variable par rapport à une substitution, on notera $\mathcal{D}_\Delta(v)$ le domaine d'une variable par rapport à une table de domaines Δ.

1.4.3 Filtrage passif des domaines

L'approche la plus brutale pour filtrer les domaines est de tester l'ensemble des substitutions issues du produit cartésien des domaines des attributs de la contrainte et de recueillir les valeurs des substitutions pour lesquelles l'évaluation réussit. La complexité de cette approche, utilisée dans [57] et [32], est donc égale à $\varpi(\mathcal{A}(C))$.

Notre approche (tableau 1.4) consiste au contraire à essayer de trouver une justification pour chaque valeur du domaine de chacun des attributs et à éliminer les valeurs qui ne sont pas justifiables (on dit qu'une valeur x est justifiable pour un attribut v de C si il existe une substitution σ telle que evaluer$(C, \sigma \cup [\![(v,x)]\!]) = \top$. Pour justifier x, on doit donc passer en revue les n-uplets de $\mathcal{A}(C) \setminus \{v\}$ et s'arrêter lorsqu'on en trouve un qui permet de vérifier la proposition précédente. Un majorant de la complexité pour ce processus est $\varpi(\mathcal{A}(C) \setminus \{v\})$. C'est le cas où x n'a, en fait, pas de justifications. La complexité de la justification complète du domaine de v est donc majorée par $\varpi(\mathcal{A}(C))$ et on établit l'arc-cohérence pour C en $k \cdot \varpi(\mathcal{A}(C))$ évaluations, avec $k < |\mathcal{A}(C)|$. Bien sûr, cette approche reste intéressante car, le plus souvent, on a $k < 1$.

A cette méthode, nous avons intégré une optimisation supplémentaire qui permet de réduire fortement le facteur k. Cette optimisation est fondée sur la remarque suivante : lorsqu'on a trouvé une substitution qui passe l'évaluation avec succès, celle-ci justifie non seulement la valeur qu'on cherche à valider mais aussi les valeurs des autres attributs substitués. Dès lors, si on mémorise ces valeurs dans une table, on évitera d'avoir à les justifier par la suite. A cet effet, la table *deja-vus* est nourrie par la fonction *justifier* tandis que son contenu est exploité par la fonction *filtrer-passif*.

L'exemple suivant permet de comparer la complexité du filtrage pour différentes approches et donnent une idée des gains qu'on peut réaliser en utilisant celle que nous venons d'exposer. Soit à appliquer un filtrage avec deux contraintes binaires : une égalité et une non-égalité. On suppose que les variables sur lesquelles elles portent ont un domaine identique de cardinalité n.

1. *Filtrage brutal.* Avec les deux contraintes, le passage en revue de l'ensemble des n-uplets du produit cartésien des domaines conduit à effectuer indifféremment un total de n^2 évaluations.

2. *Filtrage par justification.* Avec la contrainte d'égalité, la justification de la i-ème valeur d'un des domaines entraîne i évaluations. La justification complète du domaine entraîne donc $(n^2 + n)/2$ évaluations. Si l'examen de la contrainte a été provoqué par le filtrage du domaine d'un de ses attributs, on se limite à la seule justification du domaine de l'autre attribut. Le coût total est alors de $(n^2 + n)/2$ évaluations. Si les deux domaines sont à justifier, le coût devient $n^2 + n$ et s'avère être supérieur à celui de la méthode précédente.

   Avec la contrainte de non-égalité, la justification de la première valeur d'un domaine prend 2 évaluations et la justification des suivantes n'en prend qu'une chacune. La totalité du domaine est ainsi justifié en $n+1$ évaluations et le filtrage complet requiert $2n+2$ évaluations. Cette différence de complexité avec la contrainte d'égalité reflète l'écart dans la difficulté de satisfaire les deux contraintes, comme on l'a déjà évoqué en § 1.2.2.

3. *Filtrage par justification avec mémorisation.* Avec la contrainte d'égalité, la justification du premier domaine entraîne $(n^2 + n)/2$ évaluations. A l'issue de cette justification, toutes les valeurs de l'autre domaine figurent dans la table des valeurs déjà vues. A partir de là, aucune évaluation supplémentaire n'est plus nécessaire et la justification complète des deux domaines est accomplie en $(n^2 + n)/2$ évaluations. L'enregistrement des valeurs qui sont justifiées implicitement permet donc d'exploiter le surcroît de travail qu'on a fourni pour la justification explicite d'un domaine.

   En revanche, la justification d'un domaine pour la contrainte de non-égalité effectue un minimum d'évaluations et sera donc peu profitable pour la justification de l'autre. Seules 2 valeurs sont justifiées implicitement pour le domaine restant. Le nombre total d'évaluations

algorithme filtrer($C, f, \Delta, tampon$)
soit *deja-vus* = $[\![]\!]$
et *reduits* = $\emptyset$;
pour tout $v \in \mathcal{A}(C)$ **tel que** $v \neq f$
faire soit δ = filtrer-passif($v, C, \Delta, deja\text{-}vus$)
    si $\delta \neq \emptyset$
    alors soit $D = \mathcal{D}_\Delta(v) \setminus \delta$
        si $D = \emptyset$
        alors retourner $\bot$
        sinon $tampon[v] \leftarrow tampon[v] \cup \delta$;
            $\Delta[v] \leftarrow D$;
            $reduits \leftarrow reduits \cup \{v\}$;
retourner *reduits*;
fin filtrer

algorithme filtrer-passif($v, C, \Delta, deja\text{-}vus$)
soit *exclus* = $\emptyset$;
pour tout $x \in \mathcal{D}_\Delta(v)$ **tel que** $x \notin deja\text{-}vus[v]$
faire si justifier($[\![(v, x)]\!], \mathcal{A}(C) \setminus \{v\}, \Delta, deja\text{-}vus$) = $\bot$
    alors $exclus \leftarrow exclus \cup \{x\}$;
retourner *exclus*;
fin filtrer-passif

algorithme justifier($\varsigma, V, \Delta, deja\text{-}vus$)
pour tout $\sigma \in \{[\![(v_i, x_i) \mid v_i \in V\,.\ (x_1, \ldots, x_n) \in \bigotimes_{v_i \in V} \mathcal{D}_\Delta(v_i)\}$
faire si evaluer($C, \sigma \cup \varsigma$) = $\top$
    alors pour tout $v \in$ objets(σ)
        faire $deja\text{-}vus[v] \leftarrow deja\text{-}vus[v] \cup \sigma[v]$;
        retourner $\top$;
retourner $\bot$;
fin justifier

Tableau 1.4 : filtrage des domaines d'une contrainte

requis est égal à $2n - 1$ pour le filtrage des deux domaines. La complexité reste donc peu différente de celle obtenue sans mémorisation.

Ces deux exemples extrêmes mettent en valeur la complémentarité du filtrage par justification avec le processus de mémorisation. Plus le travail de justification explicite est important, plus le nombre de justifications acquises implicitement a de chance d'être élevé, soulageant ainsi les prochaines recherches. A l'inverse, lorsque la justification est peu coûteuse, la mémorisation est presque inopérante.

1.4.4 Maintien incrémental du niveau de cohérence

Lors de l'énumération, la sélection d'une valeur sur un point de choix revient à réduire le domaine de la variable à un singleton. Cette réduction peut être exploitée pour établir de façon incrémentale un certain niveau de cohérence sur le sous-problème qui reste à résoudre.

Haralick et Eliott ont montré de façon empirique dans [35] que le fait d'établir *incrémentalement* l'arc-cohérence était trop coûteux par rapport au nombre de combinaisons éliminées. Un contrôle de cohérence plus restreint se révèle statistiquement plus efficace. Ce contrôle, que nous appellerons *contrôle avant*[6] (CA), limite la propagation du filtrage aux seules contraintes directement connectées à la variable substituée. Le tableau 1.5 reprend le schéma d'énumération du tableau 1.2 pour intégrer un tel type de contrôle. Les modifications importantes y apparaissent encadrées. La fonction *choisir-variable* reçoit la table des domaines en paramètre supplémentaire, ce qui lui permet de disposer des domaines courants pour effectuer le choix de la variable.

Signalons qu'une version paresseuse de ce schéma est décrite dans [60]. Elle place des filtres sur les domaines au lieu d'exécuter immédiatement l'ensemble des tests de cohérence. Les valeurs filtrées forment un flux de données dont seul l'accès à un élément déclenche des calculs. Le gain réalisé est d'environ 20% des évaluations sur un problème tel que les n reines. Ici encore, cette variante n'offre d'intérêt que lorqu'on ne recherche qu'un nombre limité de solutions puisque, lorsqu'on désire obtenir toutes les solutions, on est en fin de compte amené à filtrer la totalité des domaines.

1.5 Déduire au lieu d'évaluer

Une contrainte peut souvent offrir plus d'informations qu'un simple constat d'échec ou de succès lors de son évaluation. Prenons l'exemple d'une contrainte d'addition C_+ définie par :

$$\begin{aligned} \mathcal{A}(C_+) &= (a, b, c) \\ \mathcal{R}(C_+) &= \{(x, y, z) \mid x = y + z\} \end{aligned}$$

Chaque valeur composant un n-uplet de $\mathcal{R}(C_+)$ se calcule à partir des autres par une expression fonctionnelle. On a en effet :

$$\forall (x, y, z) \in \mathcal{R}(C_+), \begin{cases} x = \lambda yz \,.\, y + z \\ y = \lambda xz \,.\, x - z \\ z = \lambda xy \,.\, x - y \end{cases}$$

Ainsi, lorsqu'on connaît la valeur de deux quelconques des attributs de C_+, il est possible de déduire directement la valeur du troisième telle que le n-uplet de valeurs résultant soit un élément de $\mathcal{R}(C_+)$. On dit alors qu'on utilise la contrainte de façon *active* ou *constructive* [33]. Supposons,

[6] Francisation de l'expression *forward checking*...

```
algorithme enumerer(P)
soit Φ = ∅;
avancer(vars⊥(P), [[(v, {V(v)}) | v ∈ vars⊤(P)]], Φ);
retourner Φ;
fin enumerer
```

```
algorithme avancer(V, Δ, Φ)
si V = ∅
alors Φ ← Φ ∪ {σ}
sinon soit v = choisir-variable(V, Δ)
      et E = {C ∈ C(v) | (A(C) \ {v}) ∩ V ≠ ∅}
      et D = D_Δ(v);
      pour tout x ∈ D
      faire Δ[v] ← {x} ;
            soit tampon = [];
            si tester(v, E, Δ, tampon) = ⊤
            alors avancer(V \ {v}, Δ, Φ);
            pour tout v ∈ objets(tampon)
            faire Δ[v] ← Δ[v] ∪ tampon[v];
      Δ[v] ← D;
fin avancer
```

```
algorithme tester(v, E, Δ, tampon)
tant que E = ∅
faire soit C = choisir-contrainte(E);
      si filtrer(C, v, Δ, tampon) = ⊤
      alors E ← E \ {C}
      sinon retourner ⊥;
retourner ⊤;
fin tester
```

Tableau 1.5 : satisfaction par énumération avec contrôle avant et RAC

par exemple, que a et b soient valués dans l'environnement σ et soit $r = \mathcal{V}_\sigma(a) - \mathcal{V}_\sigma(b)$. On a alors $(\mathcal{V}_\sigma(a), \mathcal{V}_\sigma(b), r) \in \mathcal{R}(C_+)$. Si, de plus, $r \in \mathcal{D}(c)$ alors evaluer$(C_+, \sigma \cup [\![(c,r)]\!]) = \top$

D'une façon très générale, on peut faire une utilisation active d'une contrainte C lorsqu'à partir d'un n-uplet de valeurs $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$, sa relation permet de calculer un ensemble fini X_i de valeurs telles que :

$$\forall x_i \in X_i,\ (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \in \mathcal{R}(C)$$

Le plus souvent, l'ensemble X_i est un singleton car les attributs de la relation sont liés par des correspondances fonctionnelles. C'est le cas de la relation $\mathcal{R}(C_+)$ précédente. En revanche, pour la relation $\{(x,y) \mid y = x^2\}$ qui lie un nombre à son carré, une valeur de y permet de calculer pour x la paire $\{-\sqrt{y}, \sqrt{y}\}$.

Pour d'autres relations, X_i n'est calculable que pour certains n-uplets. Il en est ainsi pour $\{(x,y,z) \mid z = \max(x,y)\}$ qui lie un nombre au maximum des deux autres. y n'est calculable que pour l'ensemble de triplets $\{(x,y,z) \mid z = \max(x,y) \text{ et } x < z\}$. En effet, lorsque $x = z$, on ne peut rien conclure quant à la valeur de y, on sait seulement que $y < x$. L'utilisation active de ce type de contraintes est donc conditionnée par la valeur des attributs. Un test est donc nécessaire avant toute déduction, ce qui alourdit considérablement le processus.

1.5.1 Application à la satisfaction

Pour utiliser les contraintes de façon constructive, nous allons supposer l'existence d'une nouvelle propriété des contraintes. Cette propriété, notée $\mathcal{M}(C)$ pour une contrainte C, s'appelle la liste des méthodes associées à la contrainte. Elle est telle que, si les valeurs possibles de l'attribut $\mathrm{pr}_i(\mathcal{A}(C))$ sont calculables, alors $\mathrm{pr}_i(\mathcal{M}(C))$ est une expression fonctionnelle d'arité $(|\mathcal{A}(C)| - 1)$ qui permet de calculer ces valeurs. Si le calcul n'est pas possible pour *tous* les n-uplets, alors $\mathrm{pr}_i(\mathcal{M}(C)) = \bot$. La liste des méthodes peut être vue comme la sémantique procédurale d'une contrainte par opposition à sa sémantique déclarative, caractérisée par sa relation.

Deux primitives utilisent la liste des méthodes d'une contrainte. La première permet de décider si un attribut donné peut être déduit en fonction des autres. Elle a la sémantique suivante :

$$\text{deductible}(v, C) = \begin{cases} \top & \text{si } \mathrm{pr}_i(\mathcal{M}(C)) \neq \bot \text{ avec } i \text{ tel que } v = \mathrm{pr}_i(\mathcal{A}(C)) \\ \bot & \text{sinon} \end{cases}$$

La seconde primitive effectue le calcul en appliquant l'expression fonctionnelle aux valeurs des attributs connus. Sa sémantique est :

$$\text{deduire}(v, C, \sigma) = \begin{cases} \mathrm{pr}_i(\mathcal{M}(C))\,(\ \mathcal{V}_\sigma(\mathrm{pr}_1(\mathcal{A}(C))), \ldots, \mathcal{V}_\sigma(\mathrm{pr}_{i-1}(\mathcal{A}(C))), \\ \qquad\qquad \mathcal{V}_\sigma(\mathrm{pr}_{i+1}(\mathcal{A}(C))), \ldots, \mathcal{V}_\sigma(\mathrm{pr}_n(\mathcal{A}(C)))\ ) \\ \text{avec } i \text{ tel que } v = \mathrm{pr}_i(\mathcal{A}(C)) \end{cases}$$

L'utilisation de ces deux primitives nous permet d'améliorer la satisfaction de contraintes dans deux directions. En premier lieu, on augmente les performances de la procédure de filtrage des domaines en évitant de passer en revue plusieurs n-uplets pour justifier une seule valeur. La complexité du filtrage du domaine d'un attribut v déductible par une contrainte C est alors $\varpi(\mathcal{A}(C) \setminus \{v\})$.

Cette utilisation active est d'autant plus intéressante que les contraintes concernées sont celles qui coûtent le plus cher à filtrer de façon passive. Ainsi, une contrainte d'égalité binaire dont les variables sont associées à deux domaines identiques de n éléments est filtré en $2n$ déductions alors qu'on a vu en § 1.4.3 qu'un filtrage passif nécessite $n^2 + n$ évaluations. Le tableau 1.6 détaille l'algorithme de filtrage actif. Celui-ci peut être activé par la fonction *filtrer* présentée dans le tableau 1.4, après qu'on s'est assuré que deductible$(v, C) = \top$.

algorithme filtrer-actif(v, C, Δ)
retourner $\mathcal{D}_\Delta(v) \setminus$ generer($\mathcal{A}(C) \setminus \{v\}, v, C, \Delta$);
fin filtrer-actif

algorithme generer(V, f, C, Δ)
soit $\delta = \emptyset$;
pour tout $\sigma \in \{[\![(v_i, x_i) \mid v_i \in V\ ,\ (x_1, \ldots, x_n) \in \bigotimes_{v_i \in V} \mathcal{D}_\Delta(v_i)\}$
faire soit $x =$ deduire(v, C, σ);
    si $x \in \mathcal{D}_\Delta(f)$
    alors $\delta \leftarrow \delta \cup \{x\}$;
fin generer

Tableau 1.6 : filtrage actif d'une contrainte

En second lieu, on élargit, dans une certaine mesure, l'application de l'algorithme de satisfaction à des variables dont le domaine est infini. En effet, on peut mettre en œuvre l'énumération, si à la suite d'un prétraitement, toutes les variables dont le domaine était infini ont pu avoir leur domaine déterminé par déduction. Le rôle de l'algorithme *determiner* (tableau 1.7) est d'essayer de déduire l'ensemble des domaines infinis. Chaque contrainte dont au moins un des attributs possède un domaine infini est considéré comme une règle d'inférence. Une contrainte est activable lorsqu'elle n'a qu'un seul attribut de domaine infini et qu'elle peut déduire cet attribut. Si elle n'est pas activable, la contrainte est temporairement gelée jusqu'a ce qu'elle soit rappelée à l'activation, à la suite de la détermination d'un de ses attributs. L'algorithme fonctionne à saturation sur l'ensemble des contraintes activables. Si certaines contraintes restent encore gelées lorsque l'ensemble des contraintes activables se trouve vide, cela signifie qu'il existe encore des variables dont le domaine est indéterminé. Le problème n'est alors pas soluble par énumération.

Regardons l'exemple de la figure 1.7. Les trois contraintes C_1, C_2 et C_3 sont des contraintes d'addition, similaires à la contrainte C_+ introduite précédemment. Toutes trois sont donc considérées comme des règles d'inférences potentielles pour les domaines. Initialement, seules C_1 et C_2 sont activables. L'activation de C_1 réduit le domaine de e à [2..10]. Cette détermination ne permet pas encore de rendre C_3 activable puisqu'elle possède encore deux attributs de domaines infinis. En revanche, après l'activation de C_2 qui réduit le domaine de f à [2..10], C_3 devient activable et on peut circonscrire le domaine de g qui devient [4..20]. Toutes les variables ont alors un domaine fini et le problème peut être énuméré pour rechercher une solution globale.

Cet algorithme de détermination doit s'appliquer avant d'établir l'arc-cohérence. La séparation de ces deux passes est bénéfique, car elle évite d'alourdir le mécanisme de filtrage standard. En effet, si la détermination des domaines infinis reposait seulement sur l'arc-cohérence, il serait nécessaire d'introduire dans le filtrage, des tests et des traitements exceptionnels selon les domaines. Cette séparation permet aussi de n'effectuer qu'un minimum d'évaluations pour s'assurer qu'on peut traiter un problème.

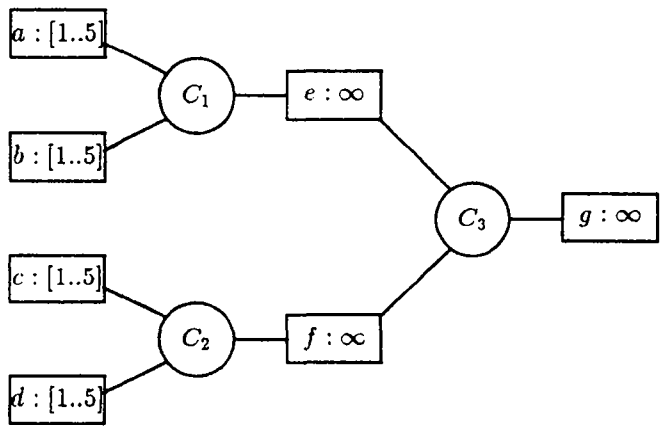

Figure 1.7 : détermination de domaines infinis

algorithme determiner(P, Δ)
soit *geles* = $\emptyset$
et *agenda* = $\{C \in P \mid \exists v \in \mathcal{A}(C),\ |\mathcal{D}(v)| = \infty\}$;
tant que *agenda* $\neq \emptyset$
faire soit $C \in$ *agenda*
    et *infinis* = $\{v \in \mathcal{A}(C), |\mathcal{D}(v)| = \infty\}$;
    agenda $\leftarrow$ *agenda* $\setminus \{C\}$:
    si *infinis* $\neq \emptyset$
    alors si *infinis* = $\{e\}$
        et deductible$(e, C) = \top$
        alors $\Delta[e] \leftarrow$ generer(e, C, Δ);
            pour tout $K \in \mathcal{C}(e) \cap$ *geles*
            faire *geles* $\leftarrow$ *geles* $\setminus \{K\}$;
                agenda $\leftarrow$ *agenda* $\cup \{K\}$
        sinon *geles* $\leftarrow$ *geles* $\cup\ C$;
retourner *geles*;
fin determiner

Tableau 1.7 : détermination des domaines infinis

1.6 Analyse topologique des problèmes

Les travaux rapportés dans [25] et, plus tard, dans [12] révèlent que la complexité de la résolution d'un problème de contraintes est fortement liée à la topologie du graphe qui lui est associé. Comme on s'y attend, le problème s'avère d'autant plus facile que sa structure s'approche d'un arbre. Les résultats de ces travaux ont un intérêt pratique important puisqu'ils donnent les moyens de résoudre certains problèmes avec une complexité polynômiale. Ils permettent, en effet, de calculer le niveau de cohérence à établir ainsi que la séquence d'énumération à utiliser pour que la recherche des solutions soit *déterministe*, c'est à dire qu'elle n'implique aucun retour arrière. Cette section présente tout d'abord ces résultats dans leur cadre originel, c'est à dire pour des problèmes de contraintes binaires. Nous montrons ensuite comment nous les avons adaptés aux contraintes n-aires puis intégrés à notre algorithme de satisfaction.

1.6.1 Problèmes de contraintes binaires

La majorité, sinon l'ensemble, des travaux qui ont été menés sur l'étude d'une instanciation sans retour arrière l'ont été dans le cadre de problèmes de contraintes binaires [25,24,23,12,7,13,8,14,43]. Nous présentons ici les principaux résultats de ces études à l'aide de notre formalisme.

Tout d'abord, on définit une propriété métrique de base qui est la *largeur* d'une variable au sein d'une séquence d'énumération. Cette largeur est égale au nombre de variables qui partagent une contrainte avec la variable donnée et qui la précèdent dans la séquence. Ainsi, soit S une séquence de variables telle que $v_1 \prec v_2 \prec \ldots \prec v_n$; la largeur de v_i dans S, notée $\xi(v_i, S)$ est donnée par :

$$\xi(v_i, S) = |\{v \in S \mid v \prec v_i \text{ et } \mathcal{C}(v_i) \cap \mathcal{C}(v) \neq \emptyset\}|$$

La largeur d'une séquence est égale au maximum des largeurs de ses variables et la largeur d'un graphe de contraintes est égale au minimum des largeurs des séquences qu'il peut engendrer. La figure 1.8 montre deux séquences, S_1 et S_2, parmi les six possibles d'un même graphe de contraintes. Pour S_1, on a $\xi(b) = \xi(c) = 1$ et $\xi(a) = 0$ ce qui donne à la séquence une largeur de $max(\{0, 1\}) = 1$. En revanche, pour S_2, on a $\xi(b) = \xi(c) = 0$ et $\xi(a) = 2$ et donc, sa largeur est $max(\{0, 2\}) = 2$.

Le théorème exposé dans [25] dit que, si le niveau de cohérence forte[7] d'un problème est plus grand que la largeur de son graphe alors, il existe au moins un ordre d'énumération qui n'appelle pas de retour arrière (*backtrack-free search*). On perçoit facilement ce résultat sur le plan intuitif. En effet, avoir un niveau de cohérence forte k signifie qu'on peut instancier jusqu'à k variables quelconques du problème sans risquer de provoquer de contradictions. D'autre part, lorsqu'on instancie une variable de largeur l, on est amené à tester les contraintes qui la lient à l variables précédentes, ce qui revient à tester la cohérence des valeurs de $l+1$ variables. Par conséquent, si la séquence utilisée est de largeur j et que $k > j$, on est assuré qu'il existe toujours un choix cohérent de valeur pour toutes les variables de la séquence. Dans l'exemple de la figure 1.8, la largeur du graphe est de 1. Cela signifie que le fait d'établir l'arc-cohérence (qui correspond à un niveau de cohérence de 2) du problème est suffisant pour assurer une énumération sans retour arrière.

Contrairement aux apparences, trouver la largeur d'un graphe n'est pas un problème de complexité exponentielle, ce qui rend le théorème précédent utilisable. L'algorithme qui trouve la largeur d'un graphe ainsi qu'une séquence de cette largeur est de complexité n^2 où n est le nombre de variables. Le principe de cet algorithme, introduit dans [25], est le suivant : tant que le graphe n'est pas vide, on y cherche une variable dont le degré est inférieur ou égal à une valeur δ, qui vaut initialement 1. Si une telle variable existe, on la place en tête de la séquence à construire

[7] On dit qu'un problème possède une cohérence forte de niveau k si il est cohérent à tout niveaux $i \leq k$.

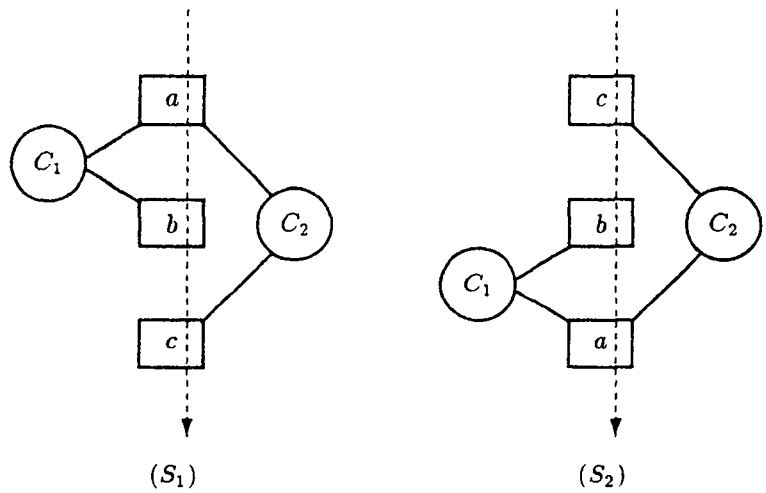

Figure 1.8 : deux séquences issues d'un même graphe

puis on l'élimine du graphe ainsi que toute les contraintes attenantes. Ce faisant, on diminue le degré de certaines variables, qui peuvent ainsi acquérir à leur tour un degré inférieur ou égal à δ. Si toutes les variables sont de degré supérieur à δ, on augmente δ de 1 et on reprend la recherche. Lorsque le graphe est épuisé, on a construit une séquence dont la largeur est minimale, égale à δ. La figure 1.9 montre un exemple du déroulement de cet algorithme. La séquence finale (a c d b) est de largeur 2. Notons que l'algorithme que nous venons de décrire s'apparente, dans son principe, à celui de satisfaction par propagation de degrés de liberté utilisé dans les systèmes SKETCHPAD [56] et THINGLAB [2].

Particularisation des problèmes arborescents

Un arbre de contraintes est un graphe de largeur 1. En effet, en orientant arbitrairement les arêtes et en effectuant un tri topologique de l'arbre obtenu, on produit une séquence dont chaque noeud est précédé au maximum d'un seul de ses noeuds adjacents. Ce tri constitue donc une séquence de largeur 1 ce qui confère à l'arbre une largeur de 1. Résoudre un problème de structure arborescente sans retour arrière implique donc d'établir seulement la cohérence de ses arcs.

Néanmoins, en y regardant de plus près, on s'aperçoit que l'arc-cohérence complète assure un niveau de cohérence encore trop élevé par rapport au minimum requis. Il est, en effet, suffisant de produire une cohérence orientée dans le sens de l'énumération. Cela signifie que pour deux variables v_i et v_j telles que $v_i \prec v_j$, il suffit que pour chaque valeur de v_i, il existe une valeur de v_j compatible. La réciproque n'est pas nécessaire, étant donné que la valeur de v_j est toujours choisie après celle de v_i et qu'il n'existe pas de contrainte implicite entre v_j et v_i. Regardons par exemple la séquence S_1 de la figure 1.8. Afin d'éviter un retour arrière, il faut s'assurer que toute valeur du domaine de a est compatible avec au moins une valeur du domaine de b. En revanche, puisque la valeur de a est connue lorsqu'on cherche à instancier b, il n'y a plus de risque de choisir

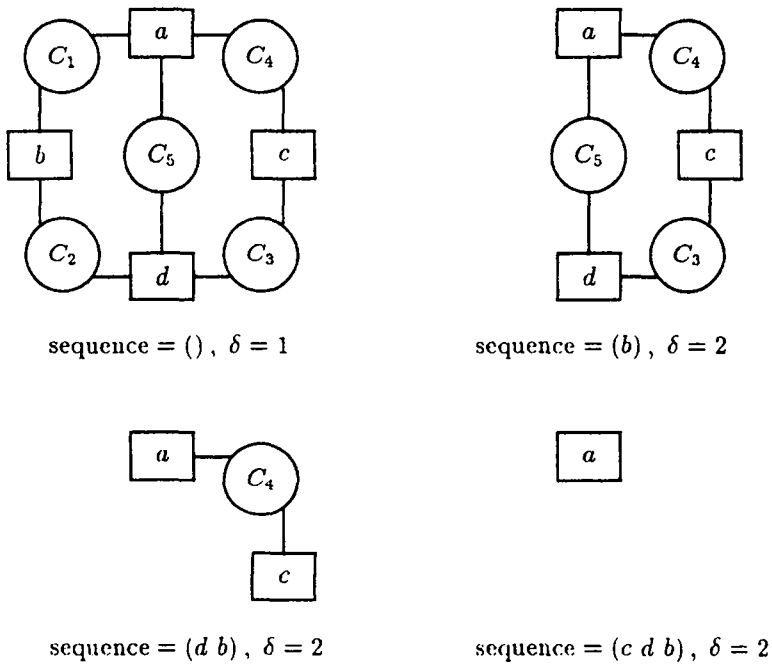

Figure 1.9 : extraction d'une séquence de largeur minimale

une valeur provoquant une contradiction.

L'algorithme qui permet d'assurer l'arc-cohérence orientée d'un problème ne réduit qu'une seule fois chaque contrainte du problème. De ce fait, sa complexité est bien inférieure à celle de l'arc-cohérence complète.

1.6.2 Application aux contraintes n-aires

La définition de la largeur d'une variable implique que l'attribut d'une contrainte d'arité n qui a le plus haut rang dans une séquence d'énumération possède une largeur minimum de $n-1$. En effet, le dernier attribut énuméré de la contrainte est lié au moins aux $n-1$ autres attributs qui le précèdent. Cela signifie par exemple qu'un graphe contenant une contrainte ternaire est au minimum de largeur 2 et nécessite donc d'être cohérent au niveau de ses chemins pour que sa résolution se fasse sans retour arrière. Les occasions d'appliquer ce type de résolution à des problèmes de contraintes n-aires se trouvent donc fortement compromises, sachant qu'on ne souhaite pas produire de niveau de cohérence supérieur à 2. Nous allons donc réduire nos ambitions et formuler une caractérisation des graphes de contraintes qui se limite à assurer que l'amplitude maximum des retours arrière nécessaires à leur résolution est proportionnel au maximum des arités des contraintes.

En guise d'exemple préliminaire, regardons le graphe de la figure 1.10*(a)*. Il comprend des contraintes d'arité 2 à 4. Les attributs partagés par plusieurs contraintes (b, c et h) ont la propriété d'être autant de points d'articulation du graphe. Ce sont donc, à présent, les *ensembles d'attributs* des contraintes qui se trouvent organisés de façon arborescente, ainsi que le montre la figure 1.10*(b)*. Par conséquent, à partir d'un tri topologique de cet arbre, on peut construire une séquence dans laquelle tous les attributs d'une même contrainte se suivent excepté un des points d'articulation qui est séparé et précède les autres (cf. figure 1.10*(c)*). En rendant le graphe arc-cohérent, on prévient les contradictions que pourrait susciter l'instanciation de ce point d'articulation. Ainsi, lors de la résolution, des retours arrière pourront intervenir mais ils seront circonscrits aux groupes consécutifs d'attributs d'une même contrainte. La complexité de la recherche d'une solution est alors dominée par la complexité de résolution de la contrainte de plus forte arité, en l'occurrence C_2.

Dans [24], Freuder étudie formellement les graphes de contraintes binaires dont la résolution nécessite une quantité limitée et mesurable de retours arrière (*backtrack-bounded search*). Il étend la définition de la largeur d'une variable à la j-largeur. La largeur d'un groupe de variables dans une séquence est le nombre de variables qui précèdent le groupe et qui partagent une contrainte avec une des variables du groupe. La j-largeur d'une variable v est le minimum des largeurs des groupes de 1 à j variables consécutives, jusqu'à v inclus. Comme il en est de la largeur simple, la j-largeur d'une séquence est le maximum des j-largeurs des variables et la j-largeur d'un graphe est le minimum des j-largeurs des séquences qu'il peut engendrer. La définition de la k-cohérence est aussi étendue. Un graphe est (i,j)-cohérent si, étant donné i variables pourvues de valeurs cohérentes, on peut trouver des valeurs pour j autres variables quelconques telles que les contraintes entre l'ensemble des $j+i$ variables soient satisfaites. Le graphe est fortement (i,j)-cohérent s'il est (i',j)-cohérent pour tout $i' \leq i$. Le théorème liant la largeur à la k-cohérence forte d'un graphe se retrouve aussi généralisé : si un graphe est fortement (i,j)-cohérent, i étant égal à la j-largeur du graphe, alors il existe un ordre d'énumération tel que la recherche des solutions ne remette en cause qu'au maximum $j-1$ variables instanciées.

Ce résultat s'applique alors très naturellement aux problèmes de contraintes n-aires. Ainsi, soit P un tel problème. Si son graphe ne présente pas de cycles, sa j-largeur est égale à 1 avec $j = max_{C \in P}(|\mathcal{A}(C)|) - 1$. Si P est rendu arc-cohérent dans le sens où nous l'entendons, c'est à dire $(1,j)$-cohérent, alors les retours arrière auront une amplitude maximum de $j-1$ variables. Nous décrivons le mode d'application et l'implantation de ce résultat dans ce qui suit.

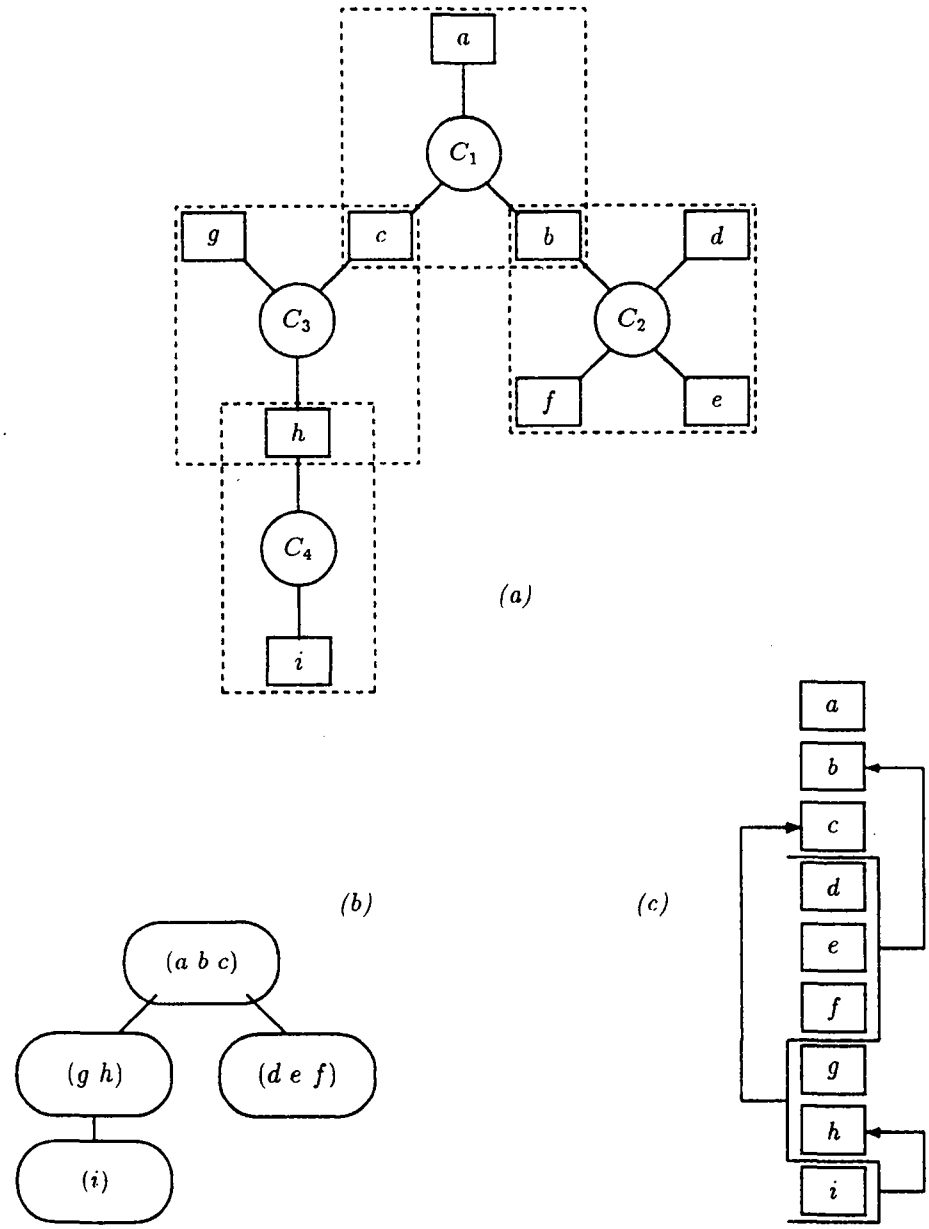

Figure 1.10 : ordonnancement des variables d'un arbre de contraintes n-aires

1.6.3 Mise en œuvre d'une analyse topologique

Un problème de contraintes est rarement arborescent dans son intégralité. Il comporte plutôt des sous-problèmes arborescents pré existants ou révélés au cours de l'instanciation des variables [8]. La figure 1.11 montre la décomposition d'un problème en trois parties arborescentes ceignant une partie centrale cyclique. On peut alors conjuguer un ordonnancement heuristique des variables de la composante cyclique avec un ordonnancement des variables des composantes arborescentes basé sur une analyse topologique.

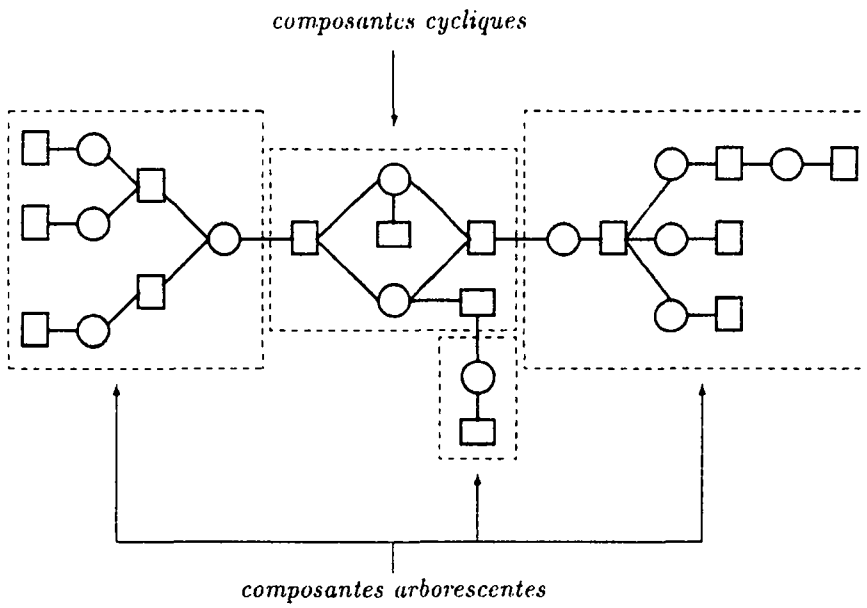

Figure 1.11 : différentes composantes d'un problème

L'algorithme *effeuiller* présenté dans le tableau 1.8 a pour rôle d'isoler les branches du graphe de contraintes et de retourner un tri topologique inverse des contraintes qui les composent. Il fonctionne selon le même principe que celui que nous avons décrit au début de cette section, à la différence qu'il s'arrête dès que le graphe ne comporte plus de variables de degré inférieur ou égal à 1. La différence de l'ensemble des contraintes initiales avec l'ensemble des contraintes retourné par *effeuiller* constitue la composante cyclique du problème.

Lorsqu'on a dégagé les contraintes qui constituent la partie arborescente, on peut établir l'arc-cohérence orientée de cette partie. La fonction *effeuiller* ayant retourné la liste des contraintes ordonnée depuis les feuilles jusqu'à la racine, il suffit de déclencher la réduction de chaque contrainte en suivant l'ordre imposé par cette liste. C'est le rôle de l'algorithme *etablir-coherence-orientee* présenté dans le tableau 1.9.

D'autre part, à partir de la séquence des contraintes, on construit facilement la séquence des variables à énumérer (cf. tableau 1.10). Il suffit pour cela de parcourir la liste des contraintes en ajoutant en tête de la séquence en construction, les attributs de la contrainte qui ne sont liés qu'à

algorithme effeuiller(P)
soit L = trouver-feuilles(P):
si $L = \emptyset$
alors retourner $\emptyset$
sinon retourner effeuiller($P \setminus L$)$+\!\!+L$;
fin effeuiller

algorithme trouver-feuilles(P)
soit *feuilles* $= \emptyset$; **pour tout** $C \in P$
faire si $|\{v \in \mathcal{A}(C) \mid \mathcal{C}(v) \cap P \neq \emptyset\}| \leq 1$
alors *feuilles* $\leftarrow C$::*feuilles*;
retourner *feuilles*;
fin trouver-feuilles

Tableau 1.8 : extraction des parties arborescentes

algorithme etablir-coherence-orientee(P, Δ)
soit *tampon* $= [\![]\!]$;
tant que $P \neq \emptyset$
faire soit V = filtrer(tete(P), $\bot$, Δ, *tampon*);
si $V = \bot$
alors retourner $\bot$;
$P \leftarrow$ queue(P);
retourner $\top$;
fin etablir-coherence-orientee

Tableau 1.9 : application de l'arc-cohérence orientée

des contraintes déjà passées en revue.

```
algorithme sequencer(L)
soit sequence = ∅
et deja-vus = ∅;
tant que L ≠ ∅
faire soit C = tete(L);
       L ← queue(L);
       deja-vus ← deja-vus ∪ {C};
       pour tout v ∈ A(C)
       faire si C(v) \ deja-vus = ∅
              alors sequence ← v::sequence;
retourner sequence;
fin sequencer
```

Tableau 1.10 : séquencement des variables des parties arborescentes

Il nous reste enfin à organiser la conjonction de la résolution de la partie cyclique et de la partie arborescente du problème. Pour ce faire, on concatène simplement les séquences de variables qui en sont respectivement issues. La composante cyclique étant, à priori, la plus complexe à résoudre et celle qui engendre le moins de solutions partielles, il est évidemment avantageux de la résoudre complètement avant l'autre. On évite ainsi de la résoudre plusieurs fois.

1.7 Problèmes mal contraints

La résolution d'un problème de contraintes fait parfois apparaître une situation extrême dans laquelle on trouve *trop* ou *pas* de solutions. On dit alors des problèmes qu'ils sont respectivement *sous-* et *sur*-contraints. Il ne s'agit alors pas d'abandonner l'utilisateur sans explications ni recours, ce qui est trop souvent le cas des systèmes de satisfaction existants.

1.7.1 Problèmes sur-contraints

Lorsqu'un problème de contraintes ne possède pas de solutions, on dit qu'il est sur-contraint. Un système intelligent ne doit pas laisser l'utilisateur démuni face à ce type de problèmes. Comme il est dit dans [23], le fait d'être confronté à des problèmes du monde réel conduit plus souvent à chercher le sous-problème qu'on peut résoudre plutôt que la solution du problème.

Curieusement, les travaux menés sur les problèmes sur-contraints sont relativement peu nombreux. Descotte et Latombe [19] ont élaboré un outil de planification qui établit des compromis lorque les contraintes qu'on fait peser sur la solution deviennent contradictoires. Borning et son équipe étudient depuis 1987 le concept de hiérarchies de contraintes [21,3,22]. Une hiérarchie est un problème dans lequel s'expriment des exigences et des préférences. Une solution est une substitution qui satisfait complètement l'ensemble des exigences et qui satisfait au mieux l'ensemble des préférences selon un comparateur donné. Les résultats de ces travaux ont été appliqués à la

programmation en logique [3] et dans une nouvelle version du langage de contraintes THINGLAB. Freuder définit dans [23] un modèle formel simple pour étudier les problèmes sur-contraints et donne un algorithme à base de séparation et évaluation[8] pour les résoudre. Les deux composantes importantes du modèle sont :

- *un espace de problèmes* $(E, \leq)$ où E est un ensemble de problèmes de contraintes et $\leq$ un ordre partiel tel que :
$$\forall P', P'' \in E, P' \leq P'' \Rightarrow \Phi_{P'} \subset \Phi_{P''}$$
- *une métrique* μ permettant de calculer la distance entre deux problèmes. Un exemple simple (et coûteux) de métrique consiste à compter le nombre de solutions qui séparent les deux problèmes. On peut aussi prendre en compte des indications du programmeur telles que des pondérations sur les contraintes.

La difficulté évidente dans la résolution d'un problème sur-contraint P est alors de trouver un sous-problème soluble $P' \leq P$ dont la sémantique reste la plus proche possible du problème initial. On veut donc que P' vérifie :

$$\mu(P', P) = \min_{P_i \leq P} \mu(P_i, P)$$

Au problème de recherche des solutions s'ajoute donc un problème de recherche dans l'espace des problèmes.

1.7.2 Relaxation d'un problème

Pour relaxer un problème P, c'est à dire trouver P' tel que $\Phi_P \subset \Phi_{P'}$, on peut choisir :

- d'élargir le domaine des variables,
- d'élargir la relation des contraintes,
- de réduire le nombre des variables,
- de réduire le nombre des contraintes.

Ces différentes façons d'agir peuvent toutes se ramener au seul élargissement de la relation des contraintes. En effet, élargir le domaine d'une variable équivaut à ajouter des n-uplets aux relations des contraintes qui impliquent cette variable. Supprimer une contrainte équivaut à la rendre universellement satisfaisable et donc à elargir sa relation. Enfin, supprimer une variable équivaut à supprimer toutes les contraintes qui lui sont attachées.

Pour faire entrer la notion d'élargissement des relations dans notre formalisme, nous introduisons la notion de *coefficients de tolérance*. Ce coefficient agit comme un modérateur des fonctions *evaluer* et *deduire*. La nouvelle syntaxe de ces fonctions est donc respectivement :

$$\text{evaluer}(C, \sigma, k)$$

et

$$\text{deduire}(v, C, \sigma, k)$$

La sémantique du coefficient k est laissé à la discrétion de l'utilisateur lorsqu'il implante les fonctions d'évaluation et de déduction. Le système se charge uniquement de véhiculer ce coefficient à

[8] Branch and bound.

travers les procédures de satisfaction. Ainsi, soit C_+ une contrainte d'addition entre trois attributs a, b et c telle que $a = b + c$.

$$\text{evaluer}(C_+, \sigma, 10)$$

avec $\sigma = [\![(a,8),(b,4),(c,6)]\!]$ peut signifier qu'on tolère une erreur absolue de 10, c'est à dire :

$$|\sigma[a] - \sigma[b] - \sigma[c]| < 10$$

auquel cas l'évaluation retourne $\top$ ou encore qu'on accepte une certaine erreur relative, par exemple :

$$\sigma[a] = \sigma[b] + \sigma[c] \text{ à } 10\% \text{ près}$$

c'est à dire $\sigma[a]/(\sigma[b] + \sigma[c]) = 1 \pm 0,1$, et dans ce cas, l'évaluation échoue et retourne $\bot$.

L'introduction d'un coefficient de tolérance représente une ouverture de notre modèle sur la résolution de problèmes sur-contraints. Néanmoins, l'essentiel, c'est à dire l'algorithme de recherche d'un sous-problème soluble doit encore être écrit.

1.7.3 Problèmes sous-contraints

Lorsqu'un problème est sous-contraint, il est difficile d'aider l'utilisateur. En effet, si on peut supprimer des contraintes pour relaxer un problème, il est impossible d'en générer arbitrairement. En revanche, on peut essayer de mettre en évidence les variables qui sont trop peu contraintes et qui multiplient inutilement le nombre de solutions. Ceci pourra amener l'utilisateur à revoir la spécification de son problème en ajoutant des contraintes sur ces variables.

Lorsqu'on considère l'ensembles des solutions d'un problème, la distribution de valeur des instanciations des variables est d'autant plus uniforme que la variable est peu contrainte. Les extrêmes sont représentés sur la figure 1.12. En abscisse sont portés les différentes valeurs possibles de la variable et en ordonnée, le nombre de fois qu'apparaît une valeur dans une solution. Dans le premier cas, chaque valeur de v apparaît autant de fois qu'il y a de solutions (on suppose qu'il y en a trois). Cette variable n'est donc pas contrainte par le problème. Dans le second cas, la variable prend la même valeur dans toutes les solutions. Elle est donc fortement contrainte.

Il apparaît donc qu'on peut distinguer les variables sous-contraintes en comparant la distribution des instanciations pour l'ensemble des solutions. Au préalable, il nous mesurer cette distribution. On utilise pour cela la théorie des probabilités à travers l'entropie [46]. L'entropie H d'une expérience comportant n alternatives et dont la probabilité de l'alternative i est p_i est définie par :

$$H = -\sum_{i=1}^{n} p_i \ln p_i$$

en prenant la convention $0 \ln 0 = 0$. Plus la valeur de l'entropie est forte, plus les probabilités sont distribuées. Par exemple, pour l'histogramme de la figure 1.12*(a)*, l'entropie est de $-5(\frac{1}{5} \ln \frac{1}{5}) = 1,609$ alors que celle de la figure 1.12*(b)* est minimum, c'est à dire nulle.

On peut constater que lorsqu'un groupe de variables ont la même entropie, elles sont liées de manière fonctionnelles par le problème. Rajouter une contrainte entre ces seules variables donne alors lieu à une contrainte redondante ou contradictoire. Pour ajouter une contrainte, il faut donc choisir des variables d'entropies différentes.

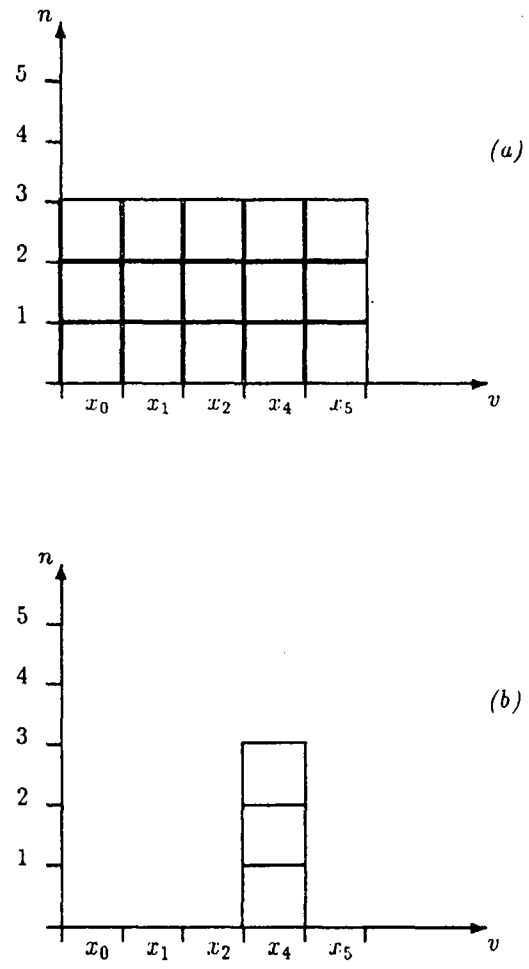

Figure 1.12 : distribution des instanciations

2 Implantation de la satisfaction

A l'issu de l'étude précédente, nous disposons des fondements nécessaires à l'implantation d'outils efficaces et souples pour satisfaire des problèmes de contraintes indépendamment de leur représentation. Nous avons réalisé une telle implantation et ce chapitre est consacré à sa description et à son évaluation.

2.1 Interface sur la représentation du problème

Avant de résoudre un problème de contraintes, il faut le poser. Le programmeur peut définir le problème avec la représentation qui lui convient. Pour que les outils de satisfaction puissent le manipuler, il doit fournir une interface fonctionnelle minimum sur cette représentation. En ce qui concerne les variables et les contraintes, les primitives de cette interface correspondent fidèlement à celles qui sont présentées en § 1.1 et en § 1.5.

2.1.1 Interface sur les variables

Trois primitives sont nécessaires à la manipulation des variables. Elles permettent d'accéder au domaine, à la valeur et aux contraintes d'une variable et doivent être implantées comme des méthodes sur les objets de type variable. Les appels à ces méthodes ont la syntaxe et la sémantique suivante :

- ▷ (send 'V-constraints <*v*>)
  retourne la liste des contraintes de <*v*>.
- ▷ (send 'V-domain <*v*>)
  retourne le domaine de <*v*>.
- ▷ (send 'V-value <*v*>)
  retourne la valeur de <*v*> ou () si la valeur n'est pas définie.

Supposons qu'on veuille représenter les variables par des symboles LE_LISP . Un codage de ces primitives serait alors simplement :

```
(de V-constraints (v)
    (getprop v 'constraints))
```

```
(de V-domain (v)
    (getprop v 'domain))

(de V-value (v)
    (getprop v 'value))
```

2.1.2 Interface sur les contraintes

Les primitives de base sur les contraintes sont au nombre de quatre. Comme on s'en doute, ce sont des méthodes sur des objets de type contrainte :

- ▷ (**send** 'C-attributes <*c*>)
  retourne la liste des attributs de <*c*>.
- ▷ (**send** 'C-evaluate <*c*> <*s*> <*k*>)
  retourne le résultat de l'évaluation de <*c*> par rapport à la substitution <*s*> avec la tolérance <*k*>.
- ▷ (**send** 'C-computable? <*c*> <*v*>)
  ne retourne () que si <*c*> ne permet pas de calculer les valeurs admissibles de la variable <*v*>.
- ▷ (**send** 'C-compute <*c*> <*v*> <*s*> <*k*>)
  retourne le domaine des valeurs admissibles par la variable <*v*> à l'aide de <*c*> par rapport à la substitution <*s*> et avec la tolérance <*k*>. Si les contraintes d'un type donné ne sont jamais utilisées de façon active, il n'est pas nécessaire de définir cette méthode pourvu que C-computable? retourne toujours ().

Les substitutions passées en arguments des appels à ces primitives sont construites par les outils de satisfaction. Leur structure est donc, a priori, inconnue du programmeur, celui-ci peut (et doit) les manipuler à l'aide de la primitive (**SAT-get-value** <*v*> <*s*>) qui retourne la valeur substituée à la variable <*v*> dans la substitution <*s*>.

Supposons qu'on veuille représenter les contraintes par des objets MICROCEYX. On définit à cet effet une classe de contraintes dotée d'un champ pour désigner ses attributs et d'un autre pour sa relation.

```
(deftclass Constraint
    C-attributes
    C-relation)
```

On définit aussi une classe pour les relations, dotée d'un prédicat qui est une lambda-expression et d'une liste de lambda-expressions (et de valeurs () au rang des variables non-calculables) pour les méthodes associées.

```
(deftclass Relation
    R-predicate
    R-methods)
```

Les primitives de manipulation des contraintes sont alors codées de la façon suivante :

```
(de {Constraint}:C-evaluate (C s k)
    (apply (send 'R-predicate (send 'C-relation C))
           (mapcar (lambda (e)
                     (SAT-get-value e s))
                   (send 'C-attributes C))))

(de {Constraint}:C-compute (C v s k)
    (apply (send 'C-computable? C v)
           (mapcar (lambda (e)
                     (SAT-get-value e s))
                   (remq v (send 'C-attributes C)))))

(de {Constraint}:C-computable? (C v)
    (any (lambda (e f)
           (when (eq e v) f))
         (send 'C-attributes C)
         (send 'R-methods (send 'C-relation C))))
```

2.1.3 Interface sur les domaines

Initialement, les différents outils de satisfaction peuvent travailler sur des domaines représentés par de simples listes non ordonnées d'éléments. Cette représentation présente l'avantage d'être applicable à la description de tous les problèmes. Néanmoins, si elle possède l'atout de la généralité, elle ne possède pas celui de l'efficacité. Lorsque le problème travaille sur des nombres entiers, il serait par exemple plus avantageux, en temps de calcul et en espace mémoire, de représenter les domaines par des intervalles (borne inférieure et borne supérieure). La complexité de certaines opérations et de l'occupation mémoire devient constante alors qu'elle était linéaire pour la représentation par listes.

Ainsi, pour autoriser l'utilisation de plusieurs types de domaines, ceux-ci sont manipulés comme des objets répondants aux méthodes suivantes :

- ▷ (send 'D-mapc <*d*> <*f*>)
  applique la fonction <*f*> sur tous les éléments de <*d*>.
- ▷ (send 'D-cons <*d*> <*x*>)
  retourne un domaine composé des éléments de <*d*> et de l'élément <*x*>.
- ▷ (send 'D-remq <*d*> <*x*>)
  retourne un domaine composé des éléments de <*d*> excepté l'élément <*x*>.
- ▷ (send 'D-memq <*d*> <*x*>)
  ne retourne () que si <*x*> n'est pas un élément de <*d*>.
- ▷ (send 'D-card <*d*>)
  retourne le nombre d'éléments de <*d*>.
- ▷ (send 'D-null <*d*>)
  ne retourne () que si <*d*> n'est pas vide.
- ▷ (send 'D-infp <*d*>)
  ne retourne () que si le domaine <*d*> est fini.

Supposons qu'on veuille représenter des intervalles par des objets MICROCEYX de classe `Interval` possédant un champ **inf** et un champ **sup**. l'interface sur cette représentation serait :

```
(de {Interval}:D-mapc (d f)
    (for (i (send 'inf d) 1 (send 'sup d))
         (funcall f i)))

(de {Interval}:D-cons (d x)
    (cond ((eq x (1+ (send 'sup d)))
           (omakeq {Interval} sup x inf (send 'inf d)))
          ((eq x (1- (send 'inf d)))
           (omakeq {Interval} sup (send 'sup d) inf x))
          (t (let (L) (send 'D-mapc d (lambda (y) (newl L y)))
                      (cons x L)))))

(de {Interval}:D-remq (d x)
    (cond ((eq x (send 'sup d))
           (omakeq {Interval} sup (1- x) inf (send 'inf d)))
          ((eq x (send 'inf d))
           (omakeq {Interval} sup (send 'sup d) inf (1+ x)))
          (t (let (L) (send 'D-mapc d (lambda (y) (newl L y)))
                      (remq x L)))))

(de {Interval}:D-memq (d x)
    (and (ge x (send 'inf d))
         (le x (send 'sup d))))

(de {Interval}:D-card (d)
    (1+ (diff (send 'sup d) (send 'inf d))))

(de {Interval}:D-null (d)
    (gt (send 'inf d) (send 'sup d)))

(de {Interval}:D-infp (d)
    (false))
```

La donnée de ces primitives permet à tous les outils de fonctionner correctement. Elle ne permet pas pour autant un fonctionnement optimal. Principalement les opérations de différence et de concaténation de domaines homogènes peuvent être améliorées. Par défaut, ces opérations sont réalisées respectivement par les fonctions :D-excise et :D-append. Leurs complexité est linéaire en fonction des domaines traités. Par exemple la concaténation est implantée de la façon suivante :

```
(de :D-append (d1 d2)
    (send 'D-mapc d2
          (lambda (x) (setq d1 (send 'D-cons d1 x))))
    d1)
```

Or, l'ajout de deux intervalles peut se faire en temps constant lorsque les intervalles ne sont pas disjoints. Pour optimiser la méthode précédente, le programmeur peut définir sa propre méthode de concaténation qui discrimine ses deux arguments :

```
(de #({Interval} . {Interval}):D-append (d1 d2)
    (if (or (< (send 'sup d1) (send 'inf d2))
```

```
    (< (send 'sup d2) (send 'inf d1)))
(omakeq {Interval} inf (min (send 'inf d1) (send 'inf d2))
                       sup (max (send 'sup d1) (send 'sup d2)))
(:D-append d1 d2)))
```

D'une façon générale, le programmeur peut définir des méthodes de différence et de concaténation efficaces dont la syntaxe et la sémantique d'appel sont les suivantes :

- ▷ (send2 'D-excise <*d1*> <*d2*>)
  retourne un domaine composé de tous les éléments de <*d1*> excepté ceux qui figurent dans <*d2*>.
- ▷ (send2 'D-append <*d1*> <*d2*>)
  retourne un domaine composé de l'union des éléments de <*d1*> et des éléments de <*d2*>.

2.2 Méthodes spécifiques

La programmation déclarative permet à l'utilisateur de décrire le *quoi* de ce qu'il veut calculer sans avoir à décrire *comment* accomplir le calcul. Les avantages de ce modèle de programmation sont maintenant largement reconnus (surtout depuis l'avènement du langage PROLOG). Il est aussi reconnu que les interprètes de ces langages ne peuvent pas travailler entièrement comme des boîtes noires et nécessitent parfois des indications d'ordre procédural pour arborer un fonctionnement efficace. Ainsi PROLOG inclut-il des mécanismes de contrôle tels que la coupure ou le gel des évaluations.

Lorsque l'utilisateur connaît suffisamment le comportement de l'interprète, il peut souvent lui communiquer des détails clés qui permettent d'améliorer nettement les performances de la résolution. Il faut donc faciliter l'expression de connaissances procédurales en des points critiques de l'interprétation. Il est aussi essentiel de fournir, pour chacun de ces points, une procédure par défaut afin que l'utilisateur puisse résoudre son problème de façon entièrement déclarative, même si les performances sont moindres. Sur ce dernier point, la programmation par objets va encore nous être d'une aide précieuse.

2.2.1 Spécialisations du filtrage

Comme nous l'avons souligné en § 1.4, le problème du filtrage des domaines est souvent négligé. On s'attache plutôt à optimiser le nombre de déclenchements de contraintes que le nombre d'évaluations par déclenchements. La stratégie brutale qui consiste à évaluer chaque élément du produit cartésien des domaines est généralement adoptée. Nous avons déjà proposé une amélioration de cette technique en § 1.4.3, applicable à tous types de contraintes mais dont le coût reste proportionnel à la taille des domaines examinés. Or, la sémantique de certaines contraintes est telle que le filtrage peut se faire en temps constant. Un exemple typique est la contrainte d'inégalité entre deux attributs. Soit $C_{\neq}$ cette contrainte. On a :

$$\begin{aligned} \mathcal{A}(C_{\neq}) &= (a, b) \\ \mathcal{R}(C_{\neq}) &= \{(x, y) \mid x \neq y\} \end{aligned}$$

Les domaines de a et b peuvent être filtrés en temps constant. En effet, imaginons qu'on effectue le filtrage depuis a; on se trouve alors dans un des deux cas suivants :

- Le domaine de a est un singleton $\{x\}$ auquel cas le domaine cohérent pour b est simplement $\mathcal{D}(b) \setminus \{x\}$.

- Le domaine de a est quelconque, de cardinal supérieur à 1. Quel que soit la valeur y dans $\mathcal{D}(b)$, on est assuré qu'il existe une valeur x dans le domaine de a telle que $x \neq y$. Aucune élimination n'est donc nécessaire.

Le cas ou le filtrage intervient à partir de b est parfaitement symétrique. On voit donc que le nombre d'opérations est indépendant de la taille des domaines. Pour spécialiser le filtrage, l'utilisateur peut coder une méthode de filtrage propre à un type de contrainte. Ce filtrage peut être soit destructif (codage d'un message C-rule-out) soit constructif (codage d'un message C-rule-in). Lorsqu'un algorithme déclenche le filtrage, il fait appel en priorité aux méthodes spécifiques et par défaut, invoque les fonctions standard :C-rule-out et :C-rule-in. L'appel des méthodes a la définition suivante :

- ▷ (send 'C-rule-out <c> <v> <k> <D> <i>)
  retourne un ensemble de domaines qui contiennent des valeurs pour lesquelles la variable <v> ne satisfait pas la contrainte <c> affectée du coefficient de tolérance <k>. Le domaine des variables est pris par rapport à la table <D> . La table <i> sert à enregistrer des valeurs déjà justifiées pour chacune des variables de <c>.
- ▷ (send 'C-rule-in <c> <v> <k> <D> <i>)
  retourne un ensemble de domaines qui contiennent des valeurs pour lesquelles la variable <v> satisfait la contrainte <c> affectée du coefficient de tolérance <k>. Le domaine des variables est pris par rapport à la table <D> . La table <i> des valeurs déjà justifiées figure avant tout parmi la liste des paramètres pour des raisons d'uniformité.

Comme pour les substitutions, la structure des tables de domaines est inconnue de l'utilisateur qui doit les manipuler par l'intermédiaire de la primitive (SAT-get-domain <v> <D>) qui retourne le domaine correspondant à la variable <v> dans la table <D>.

Supposons que la contrainte $C_{\neq}$ soit implanté par un objet de classe `Special`, sous-type de `Constraint`. On codera alors la méthode suivante :

```
(de {Special}:C-rule-out (C e k D I)
    (lets ((trigger (car (remq e (send 'C-attributes C))))
           (domain (SAT-get-domain trigger D)))
          (unless (or (gt (send 'D-card domain) 1)
                      (not (send 'included (SAT-get-domain e D) domain)))
                  (list domain))))
```

2.2.2 Spécialisations du tri

Les algorithmes de satisfaction trient les contraintes par ordre de difficulté décroissante avant de les présenter à l'évaluation. Par défaut, la difficulté d'une contrainte est un entier, égal à l'arité de la contrainte, retourné par la fonction :C-difficulty. Le programmeur peut définir lui même la difficulté d'une contrainte par l'intermédiaire de la méthode C-difficulty sur la contrainte. Cette difficulté peut logiquement varier en fonction du coefficient de tolérance courant associé à la contrainte. La spécification de cette méthode est la suivante :

- ▷ (send 'C-difficulty <c> <k>)
  retourne un entier qui quantifie la difficulté de satisfaire la contrainte <c> avec le coefficient de tolérance <k>. Plus le nombre est grand, plus la contrainte est jugée difficile.

2.3 Primitives pour la satisfaction

En échange des interfaces fonctionnelles sur les objets du problème, on fournit un ensemble de primitives pour la satisfaction. Les principales sont les primitives d'énumération, avec ordonnancement statique ou dynamique des variables, qui recherchent les solutions d'un problème :

- (SAT-dynamic-enumeration <*V*> <*P*> <*D*> <*env*> <*K*>)
  permet de lancer l'énumération avec contrôle avant et ordonnancement dynamique des variables de la liste <*V*>. Les paramètres <*P*>, <*D*>, <*K*> et <*env*> précisent l'environnement de la résolution. La liste <*P*> permet de restreindre le problème aux contraintes qui y figurent. La table <*D*> spécifie les domaines associés aux variables. La table <*K*> est un paramètre optionnel qui donne les coefficients de tolérance associés à chacune des contraintes. Enfin, <*env*> est un objet quelconque par le biais duquel le programmeur exploite et adapte à sa guise le déroulement de l'énumération. Deux méthodes permettent d'implanter ce paramétrage. Leurs appels ont la syntaxe et la sémantique suivante :

  - (send 'E-solution <*env*> <*D*>)
    déclenché lorsqu'une solution (représentée par <*D*>) vient d'être trouvée. On peut alors la stocker dans l'environnement, la comparer aux autres puis reprendre ou arrêter la recherche des solutions suivantes. Par défaut, la fonction :E-solution est invoquée et n'a aucun effet. Le résultat retourné est sans importance; la méthode agit par effet de bord.
  - (send 'E-select-variable <*env*> <*V*> <*P*> <*D*>)
    choisit et retourne la prochaine variable à énumérer parmi celles de la liste <*V*>. Par défaut, la fonction :E-select-variable est invoquée et retourne la variable qui possède le domaine le plus restreint dans <*D*> et parmi les variables dont le domaine a même cardinalité, celle qui est liée au plus grand nombre de contraintes.

- (SAT-static-enumeration <*V*> <*P*> <*D*> <*env*> <*K*>)
  son rôle est identique à la fonction d'énumération précédente mais l'ordre des variables suit strictement celui de la liste <*V*>.

Pour construire certains paramètres des deux fonctions précédentes, on peut utiliser les primitives suivantes :

- (SAT-variables <*P*>)
  retourne la liste des variables impliquées par le problème de contraintes <*P*>.
- (SAT-create-domains-table)
  crée et retourne une table de domaine vide.
- (SAT-fill-domains-table <*D*> <*V*>)
  remplit la table de domaine <*D*> en associant à chaque variable de <*V*> le singleton composé de sa valeur si elle est valuée et son domaine sinon. Retourne <*D*> en valeur.

Dans le cas d'une énumération avec un ordre statique, les variables peuvent être triées à l'aide d'une des méthodes de tri prédéfinies suivante :

- (SAT-order-by-degree <*V*> <*P*> <*D*> <*env*>)
  trie les variables de <*V*> par degré décroissant et, au sein des sous ensembles de même degré, par cardinal de domaine décroissant. Le problème considéré est limité aux contraintes figurant dans <*P*>.

▷ (SAT-order-by-connectivity <*V*> <*P*> <*D*> <*env*>)
trie les variables de <*V*> en prenant la première au hasard, puis les suivantes en comptant et en choisissant celle qui à le plus de contraintes en commun avec les variables déjà triées. Le problème considéré est limité aux contraintes figurant dans <*P*>.

Par ailleurs, le programmeur dispose de plusieurs primitives travaillant sur la topologie du problème. Les deux premières permettent d'extraire et de traiter les parties arborescentes d'un problème. Les deux dernières permettent de traiter le problème en le divisant d'abord en sous-problèmes connexes puis en reconstituant les solutions complètes à partir des solutions partielles.

▷ (SAT-constraints-tree <*P*>)
extrait les contraintes qui forment un arbre pour le problème <*P*> et retourne un tri topologique inverse de ces contraintes.

▷ (SAT-variables-tree <*tree*> <*P*>)
à partir de l'arbre de contraintes <*tree*>, retourne la liste de variables impliquées par <*tree*> dans l'ordre qu'il faut utiliser pour que la recherche effectue un minimum de retours arrière.

▷ (SAT-split <*P*>)
retourne une liste de listes de contraintes appartenant à <*P*> telle que chaque sous-liste représente une partie connexe maximale dans <*P*>. Ces sous-liste forment des sous-problèmes indépendants.

▷ (SAT-combine <*L*>)
à partir de la liste de listes de substitutions partielles <*L*>, retourne la liste de substitutions complètes qui les combinent.

Pour finir, trois prétraitements sont applicables aux problèmes de contraintes. Ils permettent d'appliquer l'arc-cohérence standard ou orientée et d'essayer de déduire les domaines infinis d'un problème. La table de coefficients de tolérance <*K*> est ici encore un paramètre optionnel.

▷ (SAT-standard-consistency <*P*> <*D*> <*bkp*> <*K*>)
filtre les domaines de la table <*D*> en imposant l'arc-cohérence aux contraintes de <*P*>. La table <*bkp*> est une table de domaines qui sauvegarde les valeurs éliminées afin de pouvoir retrouver l'état initial de la table des domaines. La fonction retourne () en valeur si le filtrage s'est déroulé correctement et provoque une sortie de l'échappement :inconsistent lorsque le domaine d'une variable est réduit à ().

▷ (SAT-directed-consistency <*P*> <*D*> <*bkp*> <*K*>)
même rôle que la fonction précédente, mis à part que l'arc-cohérence est orientée et procède suivant l'ordre de la liste <*P*>.

▷ (SAT-determine <*P*> <*D*> <*K*>)
essaye de déterminer les domaines infinis de <*D*> à l'aide des contraintes de <*P*>. La fonction retourne la liste des contraintes de <*P*> qui comptent des attributs dont le domaine n'a pas pu être déterminé. Ainsi, si la fonction retourne (), cela signifie que le problème est énumérable.

2.3.1 Exemples de mise en œuvre

Voici deux morceaux de code qui montrent la mise en œuvre des primitives que nous venons de décrire. Le premier est destiné à résoudre complètement un problème `P` (on suppose que l'environnement `env` que l'on utilise permet d'accumuler les solutions rencontrées dans son champ `solutions`) :

```
(let ((D (SAT-create-domains-table)))
     (SAT-fill-domains-table D (SAT-variables P))
     (unless (SAT-determine P D)
        (tag 'insoluble
             (SAT-combine
                 (mapcar (lambda (Pi)
                           (SAT-dynamic-enumeration
                               (SAT-variables Pi) Pi D env)
                           (or (send 'solutions env)
                               (exit 'insoluble)))
                         (SAT-split P)))))
```

L'organigramme correspondant à ce traitement est donné par la figure 2.1. Le deuxième exemple ci-dessous résout P avec un ordonnancement statique et en extrayant sa partie arborescente. La fonction **complementaire** retourne, comme son nom l'indique, le complémentaire de son premier argument dans le second. L'organigramme de cet exemple est présenté sur la figure 2.2.

```
(lets ((V (SAT-variables P))
       (D (SAT-create-domains-table))
       (O (SAT-extract-tree P))
       (tree (SAT-order-tree O))
       (cycle (complementaire tree V)))
      (SAT-fill-domains-table D V)
      (SAT-directed-consistency O D (SAT-create-domains-table))
      (SAT-standard-consistency
          (complementaire O P) D (SAT-create-domains-table))
      (SAT-static-enumeration
          (append (SAT-order-by-degree cycle P D env) tree) P D env)
      (send 'solutions env))
```

2.3.2 Occupation mémoire

Nous avons particulièrement travaillé à la compacité de l'implantation. Le noyau de base permettant de résoudre un problème avec énumération dynamique occupe 6K dans la zone de code d'une image LE_LISP sur Sun 3.60. Le module de prétraitement établissant l'arc-cohérence et le module de détermination des domaines infinis occupent chacun 1K. Enfin, le module d'énumération statique accompagné des fonctions d'ordonnancement sur les problèmes arborescents occupe 2K. La somme des mécanismes de satisfaction prend donc environ 10K de la mémoire centrale.

2.4 Evaluation expérimentale

Nous essayons maintenant, à l'aide de quelques exemples d'école, de quantifier l'apport des outils développés sur le plan des performances. En premier lieu, on met en valeur le gain obtenu en travaillant successivement sur les notions de cohérence partielle, d'ordonnancement et de définition des problèmes de contraintes. On s'intéresse ensuite aux problèmes à structure arborescente. On mesure l'intérêt d'une analyse topologique de contraintes pour préparer l'énumération. En conclusion, on souligne le fait que les résultats relatifs de différentes expériences varient en fonction des caractéristiques du problème soumis. La meilleure méthode n'est pas toujours celle qu'on attend. Cela suggère l'ouverture des systèmes de satisfaction sur un pilotage intelligent de leurs méthodes.

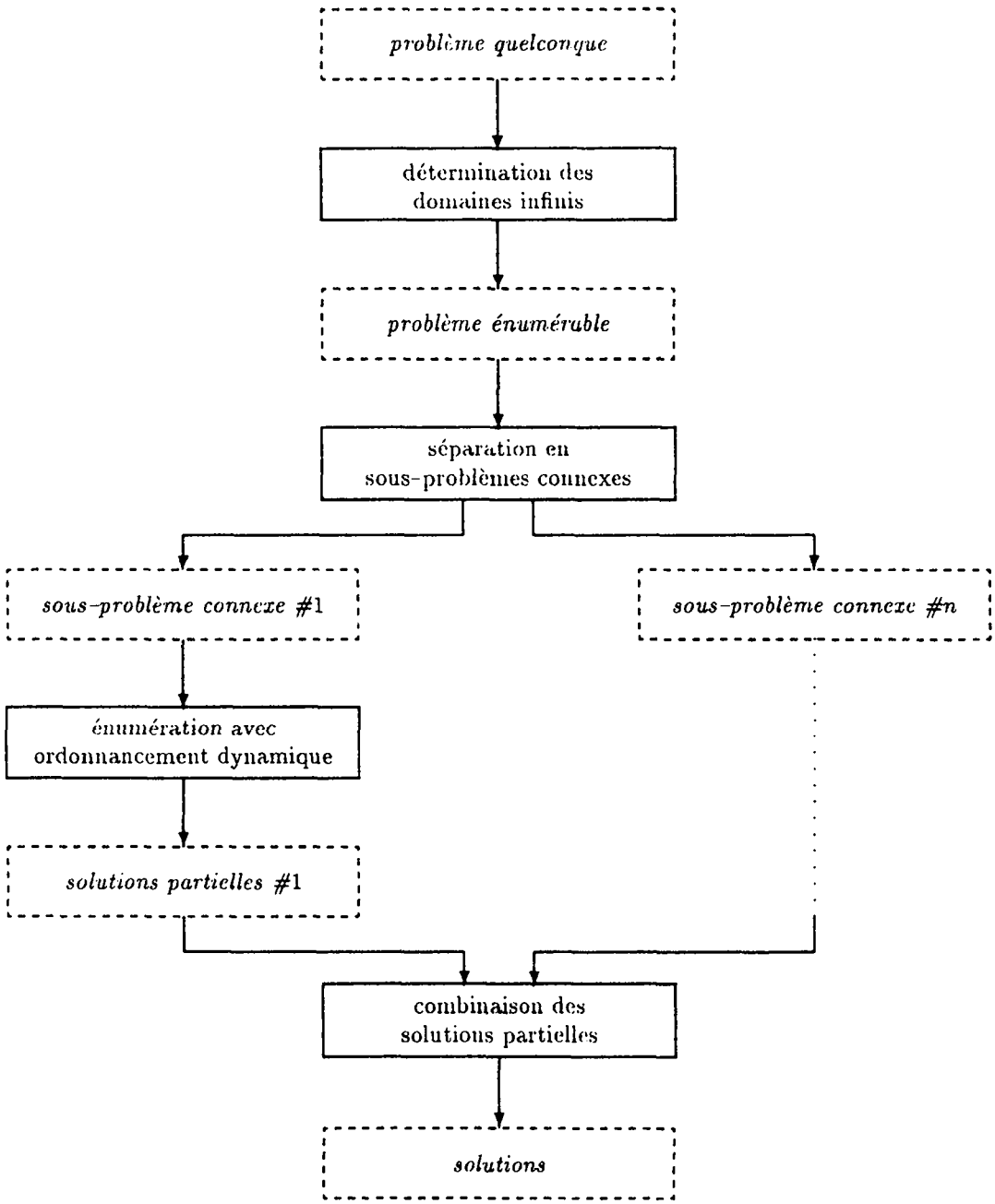
problème quelconque
détermination des domaines infinis
problème énumérable
séparation en sous-problèmes connexes
sous-problème connexe #1
sous-problème connexe #n
énumération avec ordonnancement dynamique
solutions partielles #1
combinaison des solutions partielles
solutions

Figure 2.1

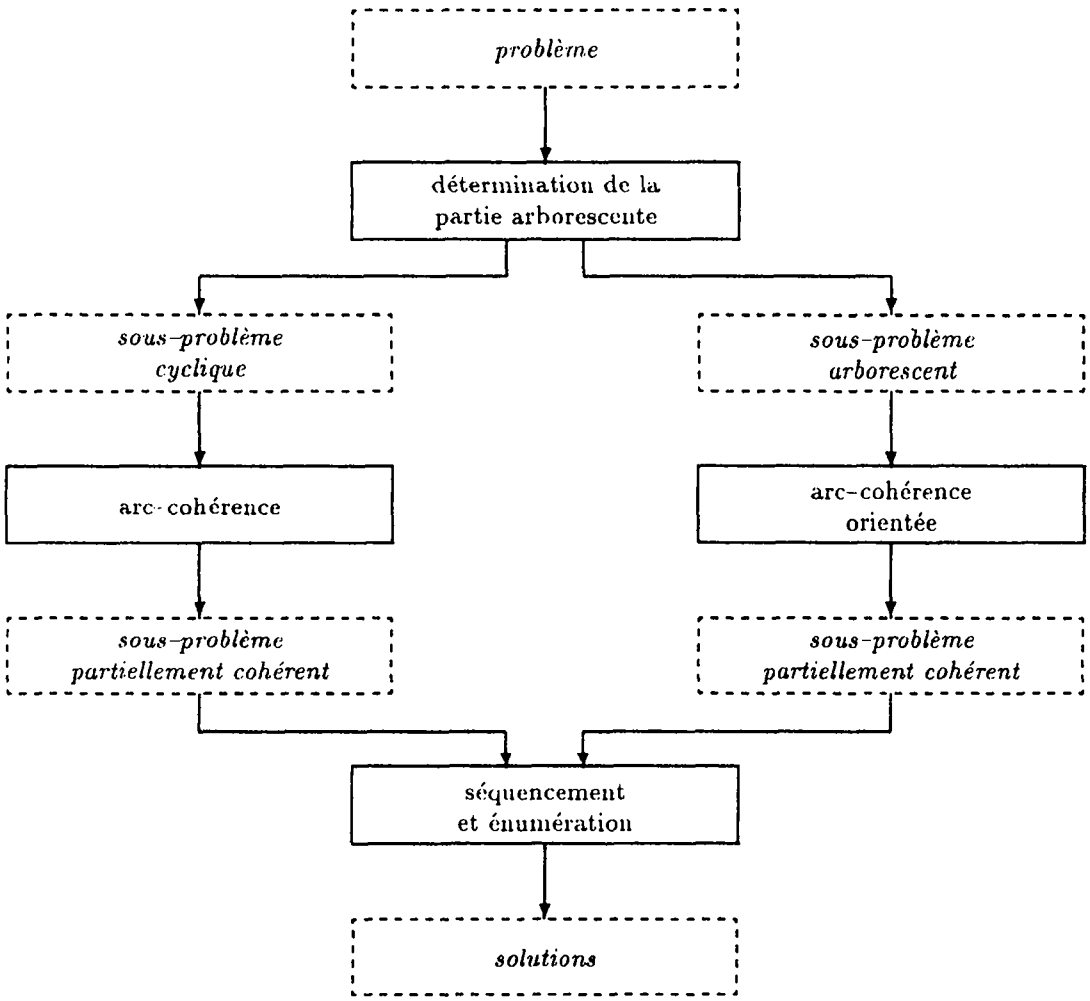
problème
détermination de la
partie arborescente
sous-problème
cyclique
sous-problème
arborescent
arc-cohérence
arc-cohérence
orientée
sous-problème
partiellement cohérent
sous-problème
partiellement cohérent
séquencement
et énumération
solutions

Figure 2.2

Note La réalisation de l'ensemble des tests utilise la représentation des variables et des contraintes décrite dans la section § 2.1. Les temps d'exécution sont ceux d'un Sun 4.60 doté de 16mO de mémoire vive.

2.4.1 Où est le zèbre?

Comme tout problème ayant longuement mûri, le problème de satisfaction des contraintes a engendré un folklore d'exemples qui lui est propre. Le plus célèbre est sans doute celui qui consiste à placer huit reines sur un échiquier de façon à ce qu'aucune d'entre elles ne soit en position de prise. Arrive ensuite le problème que nous avons choisi ici, qui consiste à attribuer un certain nombre de propriétés à un ensemble d'objets, étant donné certaines contraintes sur l'allocation des propriétés. L'énoncé détaillé est le suivant : il y a cinq maisons contiguës, de différentes couleurs, habitées par des personnes de nationalités différentes, possédant des animaux domestiques différents, et ayant chacun une boisson et une marque de cigarette favorite différente. On cherche à localiser le zèbre et le buveur d'eau, sachant que :

1. L'Ukrainien boit du thé
2. L'Espagnol possède un chien
3. Le Japonais fume des Gitanes
4. L'Anglais vit dans la maison rouge
5. On boit du café dans la maison verte
6. On fume des Kools dans la maison jaune
7. On boit du lait dans la maison du milieu
8. Le fumeur de Players possède des termites
9. Le Norvégien vit à coté de la maison bleue
10. Celui qui fume des Lucky boit du jus d'orange
11. La maison verte est à droite de la maison blanche
12. Le Norvegien vit dans la première maison à gauche
13. Le fumeur de Camel vit à coté du propriétaire du renard
14. Le fumeur de Kools vit à coté du propriétaire du cheval

Pour implanter ce problème, on divise les variables en plusieurs ensembles, à savoir : *nationalité*, *animal domestique*, *boisson*, *cigarette* et *couleur de maison*. La valeur d'une variable représente le numéro de la maison dans laquelle elle se trouve. Le domaine de chaque variable est donc l'ensemble $\{1,2,3,4,5\}$. Les variables d'un même ensemble sont contraintes, deux à deux, à avoir une valeur différente. De plus, chaque fait de la liste précédente se traduit par une contrainte. Par exemple, le point 13 se traduit par :

$$\begin{aligned} \mathcal{A}(C_{13}) &= (\mathit{camel}, \mathit{renard}) \\ \mathcal{R}(C_{13}) &= \{(x,y) \mid (x+1=y) \vee (x-1=y)\} \end{aligned}$$

L'ensemble du problème compte 62 contraintes et 25 variables. C'est un problème difficile par rapport à sa taille car il est fortement contraint et ne comporte pas de parties arborescentes.

Pour chaque expérience, nous avons compté le nombre d'évaluations, de retours arrière et le temps d'exécution (en secondes), ceci sur un échantillon de 100 permutations aléatoires de la liste des variables. Les tableaux de résultats consignent la moyenne des mesures (m) et l'écart type (σ). Soulignons que les performances en temps ne sont pas significatives dans l'absolu. Elles dépendent dans une grande mesure de la machine et accessoirement de la représentation du problème. En revanche, il est nécessaire de disposer des temps d'exécution *relatifs* des différentes méthodes. En effet, comme il est précisé dans [30], il est toujours possible de réduire le nombre d'évaluations et de retours arrière en effectuant des traitements approfondis sur le problème, mais ceci à un coût prohibitif en temps CPU. Le temps permet donc de mesurer le caractère *raisonnable* des prétraitements qu'on effectue.

Inefficacité du RAC

Les premières mesures, consignées dans le tableau 2.1, montrent l'inefficacité d'une recherche avec retour arrière chronologique qui demande en moyenne plus de 20 minutes pour résoudre le problème! Il demande aussi 38 fois plus d'évaluations qu'une simple recherche avec contrôle avant. Par ailleurs, notons que, pour toutes les quantités mesurées, l'écart-type est très important par rapport à la moyenne. Ceci laisse déjà présager l'influence de l'ordre d'énumération sur les performances de la recherche. Pour toutes les expériences qui suivent, nous avons utilisé les fonctions d'énumération avec contrôle avant présentées en § 2.3.

méthode	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
RAC	2607555	1709743	385336	253545	1293.12	1073.05
CA	67840	86813	5680	7798	82.55	108.29

Tableau 2.1 : performances du RAC

Influence d'un prétraitement

Nous nous intéressons maintenant à l'influence de l'arc-cohérence sur les performances. Le tableau 2.2 montre les mesures d'une recherche sans puis avec prétraitement. On voit que l'application de l'arc-cohérence fait gagner un facteur 4.5 sur le nombre total d'évaluations. Elle est peu coûteuse : nous avons mesuré qu'elle requiert seulement 14% du total des évaluations. Néanmoins, l'écart-type des mesures reste élevé et promet encore de nettes améliorations.

prétraitement	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
sans	67840	86813	5680	7798	82.55	108.29
avec	14833	16505	1544	2010	19.94	22.84

Tableau 2.2 : influence d'un prétraitement

Influence de l'ordonnancement

Le tableau 2.3 montre l'influence de l'ordre d'énumération des variables. Les deux ordres statiques définis en § 2.3 ainsi qu'un ordre établi dynamiquement sont utilisés. Ce dernier sort nettement gagnant de la comparaison. Non seulement il donne les meilleures moyennes mais il réduit aussi les écarts-type à des valeurs plus raisonnables.

ordonnancement	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
sans	67840	86813	5680	7798	82.55	108.29
par connexité	9901	14722	1236	1985	11.58	17.52
par degré	9259	9155	756	817	11.07	11.08
dynamique	708	494	46	51	0.94	0.67

Tableau 2.3 : influence de l'ordonnancement

Combinaison du prétraitement et de l'ordonnancement

Qu'obtient-on en combinant un prétraitement avec un ordonnancement des variables? Les résultats figurent dans le tableau 2.4. La résolution avec énumération statique (ordre par degré) se trouve nettement améliorée. L'arc-cohérence requiert en moyenne 56% du total des évaluations, c'est-à-dire 835. L'énumération n'en demande donc plus que 655. De plus, les écarts-type se trouvent ramenés à une valeur raisonnable. Le filtrage qu'accomplit l'arc-cohérence aide à construire un ordonnancement plus juste et à éliminer de nombreux tests lors de l'énumération. En revanche, les performances de l'énumération avec ordonancement dynamique sont plus mauvaises avec que sans prétraitement. On est donc tenté d'établir le classement suivant entre les différentes méthodes de résolution :

1. Ordonnancement dynamique seul (OD).
2. Ordonnancement dynamique précédé de l'arc-cohérence (OD+AC).
3. Ordonnancement statique précédé de l'arc cohérence (OS+AC).
4. Ordonnancement statique seul (OS).

ordonnancement	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
statique	1490	68	46	7	1.78	0.19
dynamique	1318	29	35	4	1.64	0.18

Tableau 2.4 : combinaison prétraitement-ordonnancement

Cependant, on ne doit pas généraliser ce classement trop rapidement. En effet, il faut prendre en compte deux faits liés à cette expérience qui diminue son caractère significatif. Le premier est la taille du problème. Elle est assez petite pour que l'arc-cohérence, qui est un processus de complexité polynômiale, demande à elle seule plus d'évaluations que l'énumération du problème, qui est un processus de complexité exponentielle. Si on suppose que le prétraitement d'un problème fait gagner un facteur constant lors de sa résolution, il ne devient profitable qu'à partir d'un certain nombre de contraintes. Il faut aussi prendre en compte le fait que, dans cet exemple, on arrête la recherche dès qu'on a trouvé la solution. Or, l'arc-cohérence s'applique sur la totalité du problème

et son action n'est pleinement amortie que quand on parcours la totalité de l'espace de recherche. Le tableau 2.5 montre les performances des quatre méthodes précédentes sur un parcours exhaustif. On voit qu'à présent, l'ensemble des résultats placent OD+AC devant OD seul. Nous donnerons des résultats encore plus probants dans la section suivante avec un problème arborescent pour lequel l'utilisation de l'arc-cohérence est essentielle.

méthode	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
OD	1377	703	133	73	1.82	0.98
OD+AC	1333	27	61	4	1.66	0.20
OS+AC	1725	79	89	8	1.99	0.22
OS	21218	16723	1804	1589	25.45	20.35

Tableau 2.5 : performances sur un parcours exhaustif

Influence de la définitions des contraintes

Pour mesurer l'influence de la définition des contraintes sur les performances de la résolution, nous avons implanté deux autres versions du problème. Dans la première, les contraintes d'égalité et de proximité sont exploitées de façon active. La seconde reprend l'implantation de la précédente en y ajoutant un filtrage optimisé des contraintes d'inégalité. Les résultats des différentes exécutions figurent dans la table 2.6. La définition de contraintes actives (20% de l'ensemble des contraintes) permet de réduire de 16% le nombre des évaluations effectuées par la définition passive. Le filtrage optimisé, qui concerne 80% des contraintes du problème, permet de gagner près de 50% du temps de résolution. Ces optimisations ne peuvent pas être néfastes car elle agissent localement à chaque contrainte et non pas sur l'ensemble du problème. Elles ne sont donc pas influencées par la structure de ce dernier.

définition	*évaluations*		*temps*	
	m	σ	m	σ
standard	708	494	0.94	0.64
active	594	368	0.93	0.62
optimisée	54	36	0.48	0.35

Tableau 2.6 : influence de la définition du problème

Influence du tri des contraintes

Nous terminons cette revue des différentes facettes de la résolution du problème du zèbre en étudiant l'impact du tri des contraintes avant leur présentation à l'évaluation. Pour cela, nous avons défini la fonction de difficulté des contraintes de façon à ce que, dans un cas *négatif*, les

contraintes d'inégalité, qui sont les plus faciles à satisfaire, soient prioritaires et inversement dans le cas *positif* où les contraintes d'égalité et de proximité sont privilégiées. Rappelons, par ailleurs, que l'ordonnancement par défaut se base sur l'arité des contraintes. Il n'est donc d'aucun secours pour ce problème puisque toutes les contraintes sont binaires. Le tableau 2.7 montre les résultats obtenus pour une énumération avec ordonnancement dynamique parcourant tout l'espace de recherche. On constate un écart d'environ 6% dans les mesures entre le cas négatif et le cas positif.

tri	*évaluations*		*temps*	
	m	σ	m	σ
négatif	1410	709	1.87	0.98
défaut	1377	703	1.82	0.98
positif	1324	676	1.76	0.96

Tableau 2.7 : influence du tri des contraintes

Prudence cependant : lorsque le problème n'est composé que d'un seul type de contraintes, comme c'est le cas de la coloration de graphe ou des n reines, l'opération de tri des contraintes est complètement inutile. Au contraire même, lorsque la taille du problème devient importante (n reines engendrent $n(n-1)/2$ contraintes; 96 reines : 4560 contraintes), cette opération devient très coûteuse. En inhibant cette fonction, les performances de la résolution s'améliorent de façon d'autant plus sensible que la taille du problème grandit. Le tableau 2.8 montre les performances de la recherche d'une solution au problèmes des n reines avec et sans tri des contrainte. Le gain atteint jusqu'à 44% du temps pour 96 reines! La troisième colonne du tableau donne, à titre de comparaison, les temps d'exécution du langage CHIP (extraits de [36]) pour les mêmes problèmes.

n	*avec tri*	*sans tri*	en CHIP
4	0.	0.	0.09
6	0.01	0.01	0.29
8	0.03	0.02	0.77
10	0.05	0.05	0.57
12	0.10	0.07	1.74
14	0.13	0.10	1.73
16	0.19	0.15	1.09
32	1.32	0.83	4.05
64	13.61	8.36	14.05
96	68.80	38.15	36.23

Tableau 2.8 : temps d'exécution pour n reines

2.4.2 Problèmes arborescents

Nous avons étudié en § 1.6 l'intérêt d'une analyse topologique des graphes de contraintes d'arité supérieure à 2. Rappelons que cette analyse permet d'extraire les parties arborescentes des problèmes et de leur appliquer des traitements particuliers en vue d'une résolution efficace. Nous donnons ici quelques résultats sur ce type de résolutions.

Le zèbre et l'arbre

Outre une résolution efficace des parties arborescentes, une analyse topologique éviter d'itérer inutilement la résolution de parties cycliques d'un problème en séparant les deux types de sous-problèmes. Pour illustrer ceci, nous étendons artificiellement le problème du Zèbre en y logeant une famille de Français. Cette famille, abritée par un même toit, comprend treize personnes. On sait seulement que la maison des Français se trouve trois maisons à droite de celle du Japonais et qu'elle porte un numéro compris entre 1 et 8. Ces nouvelles variables ainsi que les contraintes qui les lient sont représentées sur la figure 2.3.

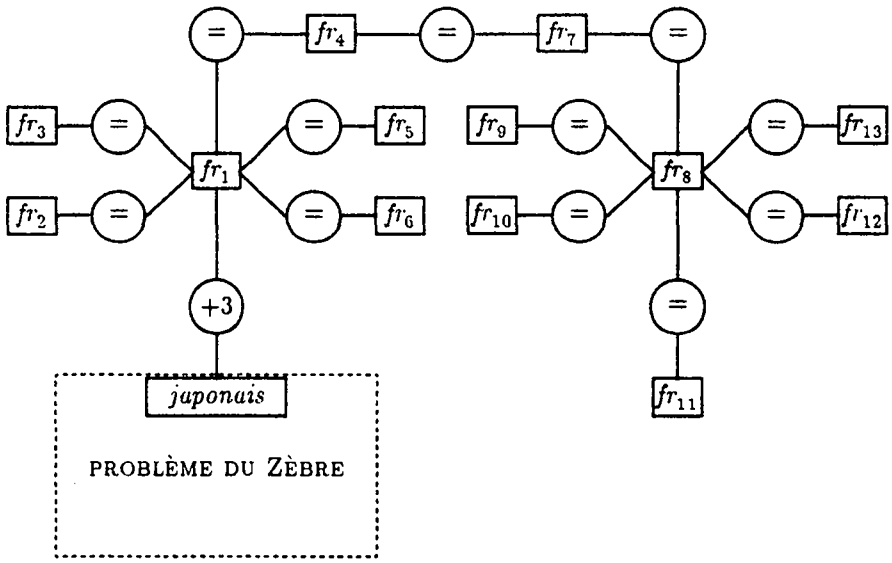

Figure 2.3 : extension arborescente

Comme on peut le voir, cette extension du problème du Zèbre forme un arbre de contraintes. Si on ordonne les variables du problème en fonction de leur degré, les deux variables fr_1 et fr_8 vont se trouver en tête de l'énumération car elles sont liées chacune à six contraintes alors que la plupart des variables du problème du Zèbre ne sont liées qu'à cinq contraintes. Inversement, les autres variables fr_i vont être rejetées en fin de la liste d'énumération. Ainsi, la contrainte implicite $fr_1 = fr_8$ ne pourra être testée qu'après que l'ensemble du problème du Zèbre aura été résolu. Ce sous-problème va donc être résolu plusieurs fois (en l'occurrence, trois) inutilement à cause d'un mauvais ordonnancement des variables. Ici, une analyse topologique du problème permet d'extraire

et d'ordonner les variables appartenant à l'arbre de contraintes issu de la variable *japonais*. On effectue ensuite un tri topologique de cet arbre et on rejette la liste de variables qui en résulte en fin de l'énumération. Le problème du Zèbre proprement dit n'est donc résolu qu'une seule fois. Les résultats d'une résolution respectivement sans et avec analyse topologique sont donnés sur le tableau 2.9.

méthode	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
AC+OS	9632	655	756	55	11.26	0.74
AC+OT	3412	124	192	13	4.14	0.16

Tableau 2.9 : influence de l'analyse topologique

Problèmes purement arborescents

Nous avons effectué des expériences simples sur des problèmes purement arborescents. Nous avons employé un arbre de contraintes ternaires de profondeur 4, ce qui représente $\sum_{i=0}^{3} 2^i = 15$ contraintes et $\sum_{i=1}^{4} 2^i = 31$ variables. Nous avons résolu cet arbre pour des contraintes d'addition puis pour des contraintes d'égalité en parcourant l'espace de recherche de façon exhaustive, c'est à dire en cherchant toutes les solutions. Le domaine de chaque variable est l'intervalle d'entiers $[1,\ldots,16]$, suffisant pour ménager une seule solution aux contraintes d'addition.

Contraintes d'addition L'arbre de contraintes d'addition a une structure telle qu'il peut être complètement résolu par l'application d'un prétraitement. En effet, à l'issu d'un filtrage du problème, les domaines de chaque variable ne comportent plus qu'un seule valeur. L'énumération n'a donc plus rien à énumérer! L'écart entre une résolution sans et avec prétraitement se révèle alors considérable, comme le montre le tableau de résultats 2.10. Ce tableau montre aussi l'avantage de l'arc-cohérence orientée sur l'arc–cohérence classique dans le cas des problèmes arborescents (on gagne ici environ un tiers du temps et du nombre d'évaluations).

méthode	*évaluations*	*retours*	*temps*
DO	2331444	7207	1228.24
AC+DO	74712	31	35.29
DAC+DO	49770	31	23.82

Tableau 2.10 : influence du prétraitement sur un problème arborescent

Contraintes d'égalité Ce cas de figure est orthogonal au précédent. Tout prétraitement est inutile puisque tous les domaines sont initialement égaux et satisfont donc les contraintes. En revanche, le filtrage s'opère lors de l'énumération. En effet, dès qu'une variable est valuée, le contrôle

avant va réduire le domaine des variables voisines à cette valeur. La résolution de ce problème est donc l'occasion de comparer les différents ordres d'énumération de variables dont nous disposons. Le tableau 2.11 donne les résultats respectifs pour un ordonnancement statique (par degré), dynamique et enfin topologique. La différence entre les deux derniers est minime avec un avantage pour le tri topologique. Ce peu de différence est dû au fait que, aidé par le contrôle avant, l'ordonnancement dynamique tend à suivre la structure arborescente d'un problème. En effet, lorsqu'une variable est choisie, ses voisines directes voient généralement leur domaine fortement réduit. Se seront donc les candidates les plus probables lors de la prochaine sélection de l'ordonnancement dynamique. Les variables sont ainsi choisies en s'écartant progressivement de la première variable, suivant l'arbre dont elle est la racine. Un problème dont les variables ont des domaines de taille très variés et comprenant des contraintes qui ont un faible pouvoir de filtrage (contrainte de non-égalité, par exemple) est susceptible d'induire en erreur l'ordonnancement dynamique.

ordonnancement	*évaluations*		*retours*		*temps*	
	m	σ	m	σ	m	σ
OS	1702486	1751062	3215	4359	1099	1782
OD	61893	15	481	0	35.15	0.61
OT	61855	0	481	0	34.52	0.30

Tableau 2.11 : influence de l'ordonnancement sur un problème arborescent

2.4.3 Vers un pilotage intelligent d'algorithmes de satisfaction

Les résultats que nous avons présenté plaident en faveur de la construction d'un système intelligent (pourquoi pas un système expert?) pour l'analyse du problème de contraintes et pour le choix des techniques de résolution à lui appliquer. Parmi les paramètres de l'analyse, il y a :

- *La taille du problème.*
- *La topologie du problème.*
- *Le nombre de solutions cherchées.*
- *La diversité des contraintes.*
- *La disparité des domaines.*

L'élaboration de la base de connaissance d'un tel système nécessite un nombre considérable d'observations variées sur divers problèmes de taille réelle. Dans la conclusion de sa thèse, Gaschnig [29] plaide d'ailleurs en faveur de la mise en oeuvre de tels tests, soutenant que les mesures de performance qu'on peut faire sur des expériences consommant beaucoup de temps CPU permettent d'obtenir des résultats précis et intéressants qui peuvent guider des développements théoriques.

Références biliographiques

[1] H. Bennaceur, G. Plateau, and F. Thomasset. *An Exact Algorithm for the Constraint Satisfaction Problem: Application to Dependance Computing in Automatic Parallelization.* Rapport de Recherche 1246, INRIA Rocquencourt, 1988.

[2] A. Borning. THINGLAB: *A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979.

[3] A. Borning, M. Maher, A. Martindale, and M. Wilson. *Constraint Hierarchies and Logic Programming.* Technical Report 88-11-10, Computer Science Department, University of Washington, 1988.

[4] M. Bruynooghe and L. Pereira. Deduction revision by intelligent backtracking. In J. Campbell, editor, *Implementation of* PROLOG, Ellis Horwood, 1984.

[5] W. Clocksin and C. Mellish. *Programming in* PROLOG, 2^{nd} *ed.* Springer-Verlag, 1984.

[6] P. Cox. Finding backtrack points for intelligent backtracking. In J. Campbell, editor, *Implementation of* PROLOG, Ellis Horwood, 1984.

[7] A. Dechter and R. Dechter. Removing redundancies in constraint networks. In *Proc. AAAI '87*, Seattle, Washington, 1987.

[8] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273-312, 1990.

[9] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proc. AAAI '86*, Philadelphia, Pennsylvania, 1986.

[10] R. Dechter, A. Dechter, and J. Pearl. Optimization in constraint networks. In R. Oliver and J. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis.* pages 411-425, J. Wiley & sons, 1990.

[11] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proc. IJCAI '89*, Detroit, Michigan, 1989.

[12] R. Dechter and J. Pearl. The anatomy of easy problems: a constraint-satisfaction formulation. In *Proc. IJCAI '85*, Los Angeles, California, 1985.

[13] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1-38, 1988.

[14] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353-366, 1989.

[15] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127-162, 1986.

[16] J. deKleer. Choices without backtracking. In *Proc. AAAI '84*, Austin, Texas, 1984.

[17] J. deKleer. A comparison of ATMS and CSP techniques. In *Proc. IJCAI '89*, Detroit, Michigan, 1989.

[18] J. deKleer. Extending the ATMS. *Artificial Intelligence*, 28:163-196, 1986.

[19] Y. Descotte and J-C. Latombe. Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183-217, 1985.

[20] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231-272, 1979.

[21] A. Borning et al. Constraint hierarchies. In *Proc. OOPSLA '87*, 1987.

[22] B. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1), 1990.

[23] E. Freuder. Complexity of K-tree structured constraint satisfaction problems. In *Proc. AAAI '90*. 1990.

[24] E. Freuder. A sufficient condition for backtrack bounded search. *Journal of the ACM*, 32(4):755-761, 1985.

[25] E. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24-32, 1982.

[26] E. Freuder. Synthesizing constraint expression. *Communications of the ACM*, 21(11), 1978.

[27] G. Friedman and C. Leondes. Constraint theory, part I. II & III. *IEEE Transactions on Systems Science and Cybernetics*, 5(2), 1969.

[28] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases.* Addison-Wesley, 1989.

[29] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, 1979.

[30] M. Ginsberg, M. Frank, M. Halpin, and M. Torrance. Search lessons learned from crossword puzzle. In *Proc. AAAI '90*, 1990.

[31] T. Graf. *Extending Constraint Handling in Logic Programming to Rational Arithmetic.* Technical Report, ECRC, 1987.

[32] H. Güsgen. CONSAT: *A System for Constraint Satisfaction.* PhD thesis, GMD Sankt Augustin, 1987.

[33] H. Güsgen. Some fundamental properties of local constraint propagation. *Artificial Intelligence*, 36:237-247, 1988.

[34] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125-130, 1988.

[35] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263-313, 1980.

[36] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[37] J-L. Laurière. *Intelligence Artificielle. Résolution de Problèmes par, l'Homme et la Machine.* Eyrolles, 1986.

[38] J-L. Lauriere. *Un Langage et un Programme pour Énoncer et Résoudre des Problèmes Combinatoires.* Thèse de Doctorat d'État, Université Pierre et Marie Curie, Paris, 1976.

[39] W. Leler. *Specification and Generation of Constraint Satisfaction Systems.* PhD thesis, University of North Carolina at Chapel Hill, 1987.

[40] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[41] A. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[42] J. Malcki. ICONS*traint: A Dependency-Directed Constraint Maintenance System.* Thesis, Linkoping university, 1987.

[43] I. Meiri. R. Dechter, and J. Pearl. Tree decomposition with application to constraint processing. In *Proc. AAAI '90*, 1990.

[44] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[45] U. Montanari. Networks of constraints: fundamental properties and application to picture processing. *Information Science*, 7(3):95–132, 1974.

[46] R. Neapolitan. *Probabilistic Reasoning in Expert Systems. Theory and Algorithms.* J. Wiley & sons, 1989.

[47] B. Nudel. Solving the general consistent labeling problem: two algorithms and their expected complexities. In *Proc. AAAI '83*, Washington, D.C., 1983.

[48] A. Oplobedu, J. Marcovich, and Y. Tourbier. CHARME : un langage industriel de programmation par contraintes. In *Proc. Avignon '89*, 1989.

[49] P. Purdom. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–113, 1981.

[50] P. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[51] F. Rossi, C. Petrie, and V. Dhar. *On the Equivalence of Constraint Satisfaction Problems.* Technical Report ACT-AI-222-89, MCC, 1989.

[52] M. Sakarovitch. *Optimisation Combinatoire : Graphes et programmation linéaire.* Hermann, 1984.

[53] S. Steel. On trying to do dependency-directed backtracking by searching transformed state spaces (and failing). In *Proc. AISB '87*, 1987.

[54] G. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis. Massachusetts Institute of Technology, 1980.

[55] H. Stone and J. Stone. *Efficient Search Techniques. An Empirical Study of the N-Queens Problem.* Technical Report 12057 (#54343), IBM Research Division, 1986.

[56] I. Sutherland. SKETCHPAD*: A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, 1963.

[57] A. Voss and H. Voss. *Formalizing Local Constraint Propagation Methods.* Arbeitspapiere der GMD 248, GMD Sankt Augustin, 1987.

[58] R. Zabih. Some application of graph bandwidth to constraint satisfaction problems. In *Proc. AAAI '90*, 1990.

[59] R. Zabih, D. McAllester, and D. Chapman. Non-deterministic lisp with dependency-directed backtracking. In *Proc. AAAI '87*, Seattle, Washington, 1987.

[60] M. Zweben and M. Eskey. Constraint satisfaction with delayed evaluation. In *Proc. IJCAI '89*, Detroit, Michigan, 1989.

A Implantation des exemples

Nous donnons ici la représentation du problème du zèbre et des n reines. Trois fonctions aident à créer des relation, des contraintes et à initialiser des variables. Ce sont les suivantes :

```
(de :relate (pred . meths)
    (omakeq {Relation}
            R-predicate pred
            R-methods (car meths)))

(de :constrain (attrs rel)
    (let ((C (omakeq {Constraint}
                     C-attributes attrs
                     C-relation rel)))
         (mapc (lambda (e)
                  (putprop e (cons C (getprop e 'constraints))
                             'constraints))
               attrs)
         C))

(de :initvar (lvars dom)
    (mapc (lambda (e)
            (putprop e dom 'domain)
            (remprop e 'constraints))
          lvars))
```

L'invocation de la fonction suivante retourne une instance du problème du zèbre dans sa version la plus simple.

```
(de zebra ()
    (let ((P)
          (R-neq   (:relate (lambda (x y) (<> x y))))
          (R-eq    (:relate (lambda (x y) (= x y))))
          (R-right (:relate (lambda (x y) (= y (1+ x)))))
          (R-next  (:relate (lambda (x y) (or (= x (1+ y))(= x (1- y)))))))
         (mapc (lambda (cluster)
```

Table des matières

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249-6399