Numéro d'identification : 95 - MON 2 - 250

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUE DU LANGUEDOC —

THÈSE

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

SPÉCIALITÉ : **INFORMATIQUE**
Formation Doctorale : **Informatique**
École Doctorale : **Sciences Pour l'Ingénieur**

Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.

par

Jean-Charles RÉGIN

Soutenue le 21 Décembre 1995 devant le Jury composé de :

M. Michel HABIB, Professeur, LIRMM, Université de Montpellier II, ... Président
M. Yves DEVILLE, Professeur, Université Catholique de Louvain (Belgique), ... Rapporteur
M. Amedeo NAPOLI, Chargé de Recherche, CNRS/CRIN, Nancy, ... Rapporteur
M. Michel RUEHER, Professeur, Université de Nice Sophia Antipolis, ... Rapporteur
M. Christian BESSIÈRE, Chargé de Recherche, CNRS/LIRMM, Montpellier, ... Examinateur
M. Bertrand CASTRO, Directeur D.A.C., SANOFI-CHIMIE, Gentilly, ... Examinateur
M. Claude LAURENÇO, Directeur de Recherche, CNRS/CCIPE, Montpellier, ... Examinateur
M. Jean-François PUGET, Chef de projet, ILOG S.A., Gentilly, ... Examinateur
M. Joël QUINQUETON, Directeur de Recherche, INRIA/LIRMM, Montpellier, ... Examinateur
M. Olivier GASCUEL, Chargé de Recherche, CNRS/LIRMM, Montpellier, ... Directeur de Thèse

Table des matières

«Good ideas aren't good unless they can be implemented. That's one of the lessons of AI today:
There are *lots* of good ideas. But the only way to tell what is worth pursuing is to code it up and see what happens. AI is awash in great ideas. What it needs is more substance.»

Matthew L. GINSBERG

Remerciements

En premier lieu, je remercie vivement Bertrand Castro, directeur du département des activités chimiques de la société SANOFI-CHIMIE, pour la confiance sans faille qu'il m'a accordée pendant de nombreuses années. Sans le financement qu'il m'a apporté je n'aurais jamais pu faire cette thèse. J'espère que dans l'avenir d'autres thésards bénéficierons d'un tel avantage.

Je remercie Claude Laurenço, directeur de recherche au CNRS, pour l'intérêt qu'il a porté à mes travaux depuis le DEA et pour toutes sortes de choses qu'il m'a enseignées aussi bien sur la nature humaine que sur la synthèse en chimie organique. Merci, Claude.

Je voudrais remercier ensuite ceux qui ont accepté d'être rapporteur de «mon pavé de 380 pages» dans un délai légal, mais plutôt court :

- Yves Deville, professeur à l'Université Catholique de Louvain (Belgique). Je le remercie pour l'intérêt qu'il a porté à mon travail et pour l'évaluation qu'il en a faite.

- Michel Rueher, professeur à l'Université de Nice Sophia Antipolis. Je le remercie de s'être intéressé à ma thèse, de l'avoir appréciée et de s'être investi personnellement pour défendre mon travail.

- Amedeo Napoli, chargé de recherche au CNRS. Je le remercie pour m'avoir toujours soutenu et aussi parce qu'il n'a jamais hésité à m'aider ou à me proposer son aide. Je le remercie également pour la lecture très approfondie qu'il a faite de mon manuscrit, pour ses remarques judicieuses et pour le jugement qu'il a porté sur mon travail.

Je remercie Joël Quinqueton, directeur de recherche à l'INRIA, pour avoir accepté d'être membre de mon jury de thèse. Discuter avec lui a toujours été un plaisir pour moi.

Je remercie Christian Bessière pour m'avoir toujours accepté dans son bureau (après que j'ai frappé) pour écouter ma dernière «idée». Je le remercie aussi pour son honnêteté et sa franchise scientifique. Je tiens à lui exprimer toute mon amitié.

Je remercie Jean-Francois Puget, chef de projet chez ILOG S.A., pour l'intérêt qu'il porte à mes travaux et pour avoir accepté de participer à mon jury de thèse. J'espère que la collaboration que nous avons entreprise, puisqu'il est actuellement mon «chef», sera longue et profitable.

Je remercie Michel Habib, professeur à l'Université de Montpellier II, pour m'avoir fait l'honneur de présider le jury de cette thèse.

Je remercie Olivier Gascuel, chargé de recherche au CNRS, pour avoir accepté d'être mon directeur de thèse et pour la très grande liberté qu'il m'a laissée pendant plusieurs années.

Je remercie Eugène Freuder, professeur à l'Université du New Hampshire, pour la confiance qu'il m'a accordée en acceptant de partager la présentation de notre papier à l'IJCAI'95. Je lui exprime aussi toute ma gratitude pour m'avoir écouté plusieurs fois malgré mon très mauvais anglais.

Je remercie Pascal van Hentenryck, professeur à l'Université de Brown, pour la sympathie qu'il montre à mon égard. Je souhaite sincèrement que nous ayons un jour l'occasion de travailler ensemble.

Je remercie Christophe Fagot pour m'avoir réconcilié avec l'apprentissage.

Je remercie Daniel Sabin, thésard de l'Université du New Hampshire, pour avoir relu avec intérêt certains chapitres de cette thèse.

Je remercie particulièrement Christophe Fiorio et Richard Nock, deux thésards du LIRMM, avec qui j'ai eu des discussions très intéressantes sur beaucoup de domaines. Je tiens à leur exprimer ici toute ma sympathie.

Je remercie Philippe Vismara, mon voisin de bureau, pour m'avoir supporté pendant plusieurs années.

Je remercie tous les autres membres du GDR chimie «Traitement Informatique de la Connaissance en Chimie Organique», pour les bons moments que nous avons passés ensemble.

Je remercie Sandrine pour m'avoir toujours soutenu et encouragé dans les moments difficiles.

Je remercie mes parents pour m'avoir toujours fait confiance et aussi pour le soutien qu'ils m'ont toujours apporté.

Que ceux que j'apprécie et que j'ai oubliés m'excusent. Je leur témoigne toute mon affection.

Enfin, je terminerai ces remerciements par un souhait. Pendant les 5 années que j'ai passées en thèse, j'ai côtoyé dans le milieu universitaire et notamment au LIRMM, toutes sortes de gens. Certains particulièrement intelligents, brillants et passionnés; d'autres plutôt insignifiants et incapables. Aussi pendant que les premiers essayaient de me montrer la beauté des sciences et tentaient de me communiquer leur enthousiasme, les seconds m'apprenaient le sens profond des termes incompétence et mauvaise fois. Je souhaite simplement que, dans l'avenir, les personnes de la première catégorie soient plus représentées à l'Université que ceux de la seconde.

Avant Propos

Les travaux que nous présentons constituent l'un des thèmes de recherche du GDR-Chimie T.I.C.C.O. (Traitement Informatique de la Connaissance en Chimie Organique). Ils ont été réalisés en collaboration avec la société SANOFI-CHIMIE, et s'intègrent dans un projet qui a pour objectif de montrer les performances de techniques de l'Intelligence Artificielle comme l'apprentissage par similitude dans un domaine d'application tel que la chimie organique et plus particulièrement la synthèse des molécules organiques.

Le thème principal de cette thèse était, à l'origine, la création d'un système capable de déterminer et d'expliquer le caractère stratégique de certaines liaisons en synthèse organique. L'explication devait obligatoirement être compréhensible par un expert de la chimie organique. Le langage employé par ce dernier pour décrire des objets de son domaine repose sur les formules moléculaires qui , très souvent, peuvent être représentées par des graphes. Aussi était-il indispensable que le système manipule les données sous une forme symbolique et plus particulièrement qu'il utilise des graphes étiquetés. L'une des opérations de base d'un tel système est la comparaison de deux éléments e_1 et e_2 pour savoir si l'un est plus général que l'autre. Cette opération se traduit en théorie des graphes par la recherche d'un isomorphisme entre e_1 et un sous-graphe partiel de e_2. Non seulement il est indispensable de pouvoir répondre à cette question, mais il faut aussi que cette réponse soit la plus rapide possible. En effet, c'est cette opération qui prend globalement le plus de temps car elle est appelée plusieurs millions de fois par le système. Face à un tel nombre, la résolution du problème de l'isomorphisme de sous-graphes partiels (Iso-ssgrpartiel en abrégé) est devenu l'un des problèmes majeurs à résoudre. C'est pourquoi j'ai consacré une partie importante de mon temps à ce problème. Cela explique que la moitié de ma thèse lui

soit, en partie, dédiée. Cette première partie concerne aussi les réseaux de contraintes car de nombreux algorithmes de ce domaine permettent de résoudre efficacement Iso-ssgrpartiel.

Dans la seconde partie de ce document je propose un nouveau système d'apprentissage conceptuel pour des données symboliques. Je présente aussi comment j'ai modélisé le problème d'origine. Ces deux tâches furent assez longues et difficiles. En effet, il est délicat lorsque l'on traite un problème réel pour lequel il n'existe pas de modélisation naturelle et quand on développe parallèlement un système d'apprentissage basé sur la modélisation du problème, de bien gérer les interactions entre la modélisation et la conception du système.

Par ailleurs, je dois mentionner un grand absent de cette thèse: le système ReSyn. Ce système a reçu en 1993 un premier prix de l'innovation et de la recherche délivré par l'A.D.E.R. L.R. (Association pour le Développement de l'Enseignement et de la Recherche du Languedoc-Roussillon). J'ai eu le plaisir de participer à son développement, tant pour la modélisation que pour la programmation. Notamment j'ai implémenté son mécanisme de classification et bien sûr l'algorithme d'appariement de graphes. ReSyn est détaillé dans la thèse de P. Vismara [Vismara, 1995]. Cependant, pour ne pas décevoir le lecteur particulièrement intéressé, j'ai placé en Annexe un article en cours de soumission qui détaille le fonctionnement de ce système. Mon travail a donné lieu a d'autres publications [Régin, 1994; Régin *et al.*, 1994; Bessière and Régin, 1994a; Bessière and Régin, 1994b; Bessière and Régin, 1995; Bessière *et al.*, 1995] qui ne sont pas données car elles sont intégrées dans le corps du texte.

Pour terminer cet avant-propos, je tiens à préciser quelques conventions d'écriture que j'ai choisies. J'ai employé, sauf dans les introductions et les conclusions des deux parties, uniquement la première personne du pluriel afin de traduire un travail d'équipe, mais aussi pour respecter une certaine homogénéité. En ce qui concerne les algorithmes, le style employé est proche de ALGOL et n'est lié à aucun langage particulier. Les paramètres en entrée sont précédés d'un «i» et ceux en entrée/sortie d'un «i/o». Les commentaires sont encadrés par /* et */ comme en C.

Première partie

Isomorphisme de sous-graphes

Chapitre 1

Introduction

Pour présenter cette partie, j'ai décidé d'en raconter l'histoire afin de faire mieux comprendre au lecteur pourquoi je ne me suis pas limité uniquement à l'étude du problème de l'isomorphisme de sous-graphes partiels (Iso-ssgrpartiel en abrégé) pour les graphes moléculaires.

Dans un premier temps, je me suis, bien entendu, intéressé aux graphes moléculaires. Ces graphes sont assez particuliers puisqu'ils sont le plus souvent planaires et puisque leurs sommets possèdent des degrés très faibles (la valence[1] du carbone vaut 4). Devant de tels graphes, la première question que l'on doit se poser concerne la complexité théorique de Iso-ssgrpartiel. Je montre dans le deuxième chapitre que la frontière entre instances polynômiales et instances NP-Complètes est maintenant assez bien définie, mais que, malheureusement, le problème est très souvent NP-Complet pour les graphes moléculaires. Sans rentrer dans le détail, c'est presque toujours vrai dès l'instant où l'un des deux graphes possède un cycle. Autant dire qu'en chimie organique le problème n'est pas facile (au sens polynômial) à résoudre.

Il n'empêche que je restais persuadé, à ce stade, que bien que le problème soit NP-Complet dans le pire des cas, on devait avoir développé des algorithmes très efficaces en pratique pour les données de la chimie organique. En effet, les graphes considérés ont une structure assez proche de celle des arbres et l'on connaît des algorithmes polynômiaux pour résoudre Iso-ssgrpartiel pour deux arbres. D'ailleurs, j'en présente un

1. La valence correspond, très grossièrement, au nombre maximum de voisins que peut avoir un sommet dans la formule moléculaire

dans le troisième chapitre de cette thèse. J'ai donc étudié les articles de la littérature pour essayer de trouver un algorithme me satisfaisant. Bien que le problème considéré soit un problème très important de la théorie des graphes, il n'existe en fait que peu d'algorithmes se proposant de le résoudre. Le premier que j'ai employé provenait de la thèse d'un chimiste ([Tonnelier, 1990]) et s'est très vite avéré inutilisable. Il fallait parfois attendre plus d'une demi-heure pour avoir la réponse pour des molécules courantes ayant une quarantaine de sommets ! Dans ces conditions il n'était pas question de l'utiliser dans mon système. Ceux que j'ai par la suite programmés ([Sussenguth, Jr., 1965; Ullmann, 1976]) étaient bien meilleurs. En voulant les améliorer, j'ai compris que ces algorithmes étaient d'une part équivalents et d'autre part qu'ils utilisaient, sans le savoir, certaines techniques développées pour résoudre les problèmes de satisfaction de contraintes (CSP). Cela avait déjà été mis en évidence par J.-J. McGregor [McGregor, 1979]. Je ne découvris ce papier, malheureusement, qu'après avoir fait ce constat. Ces algorithmes ainsi que leur équivalence sont exposés dans le chapitre 3.

Comme il existe au LIRMM[2] une équipe étudiant les réseaux de contraintes, qui sont la base des CSP, et puisque je trouve ce domaine passionnant, j'ai décidé d'y consacrer une partie de mon temps. De plus, en améliorant les techniques développées pour résoudre les CSP, j'étais assuré de pouvoir résoudre mon problème plus rapidement. Je présente dans le chapitre 4 un bref état de l'art sur les réseaux de contraintes.

Très souvent, le système d'apprentissage demande s'il existe une solution à Iso-ssgrpartiel pour deux graphes alors qu'il n'en existe pas. Je pouvais donc améliorer les performances en temps du système en détectant plus rapidement l'inconsistance du réseau de contraintes obtenu en modélisant Iso-ssgrpartiel. Dans ce cas la démarche légitime est de s'intéresser aux procédures de filtrage dont le rôle est de supprimer des valeurs du domaine des variables qui ne peuvent apparaître dans aucune solution. La plus connue et la plus utilisée de ces méthodes est la consistance d'arc. Je me suis donc plus particulièrement intéressé à celle-ci. Alors que j'essayais de particulariser l'algorithme optimal AC-4 [Mohr and Henderson, 1986] pour mon problème, C. Bessière et M-O. Cordier ont découvert un algorithme beaucoup plus performant en pratique : AC-6 [Bessière and Cordier, 1993; Bessière, 1994]. Convaincu de l'intérêt d'AC-6 par

2. Laboratoire d'Informatique de Robotique et de Micro-électronique de Montpellier.

rapport aux algorithmes précédents j'ai collaboré avec C. Bessière[3]. De cette collaboration, nous avons obtenu plusieurs résultats que je présente dans le chapitre 5. Nous avons notamment réussi à prendre en compte la bidirectionnalité des contraintes dans AC-6 sans modifier sa remarquable complexité en espace; c'est ainsi qu'est né l'algorithme AC-7. Ce résultat n'était pas évident puisque E. Freuder dans un article, présenté au même Workshop que celui où a été exposé AC-7, pensait que c'était impossible [Freuder, 1994; Freuder, 1995]. Plusieurs versions furent nécessaires pour simplifier notre algorithme [Bessière and Régin, 1994a; Bessière and Régin, 1994b; Bessière and Régin, 1995]. Je ne présente dans cette thèse que la dernière version.

Parallèlement à nos travaux, E. Freuder proposait de prendre en compte d'autres connaissances sur les contraintes, comme la commutativité et la réflexivité des contraintes, afin d'améliorer les algorithmes de fermeture par consistance d'arc. Les travaux que nous avions entrepris indépendemment étant très proches nous avons décidé de collaborer. Encore une fois cette collaboration fut fructueuse puisqu'elle a abouti à une publication à l'IJCAI [Bessière *et al.*, 1995]. Dans ce papier, dont les idées sont détaillées dans le chapitre 5, nous proposons un algorithme général achevant la consistance d'arc en tenant compte de propriétés des contraintes : AC-Inférence. L'un des inconvénients d'AC-Inférence est qu'il perd la faible complexité en espace d'AC-7. Pour remédier en partie à ce problème, nous avons introduit la notion de contraintes identiques. AC-7 et AC-Inférence remettent en cause le schéma habituel des algorithmes précédents, à savoir deux phases distinctes : une d'initialisation et une de propagation. En effet nous montrons, de manière expérimentale, qu'il est préférable d'étudier immédiatement les conséquences de la disparition d'une valeur quand on s'en aperçoit pendant la phase d'initialisation. Pour finir le chapitre 5, je présente une heuristique permettant de détecter les inconsistances plus rapidement.

L'idée communément admise pour résoudre les CSP est qu'il faut utiliser un filtrage puis un algorithme de type Forward-Checking, c'est-à-dire un backtrack avec un filtrage très réduit pour chaque niveau. Or je disposais d'algorithmes de consistance d'arc performants, j'ai donc décidé d'aller à l'encontre de cette idée et d'employer systématiquement la consistance d'arc pendant la procédure de recherche. Peu de temps après, D. Sabin et E. Freuder publiait deux articles [Sabin and Freuder, 1994a;

3. Son bureau étant situé à 10 mètres du mien, cette collaboration n'en fut que plus facile.

Sabin and Freuder, 1994b] renforçant ma démarche puisqu'ils montraient que MAC-4 (pour maintien de la consistance d'arc avec AC-4) est souvent meilleur que le Forward-Checking. Pour obtenir de bien meilleures performances, il fallait que j'arrive à utiliser, pendant la recherche, AC-6, AC-7 ou AC-Inférence plutôt qu'AC-4. Ces algorithmes sont plus difficiles à intégrer dans une procédure de recherche qu'AC-4 si l'on veut profiter pleinement de leurs performances. Je pense avoir réussi à atteindre ce but comme le montre le chapitre 6 dans lequel je détaille les algorithmes MAC-6, MAC-7 et MAC-Inférence.

Par ailleurs, une des contraintes qui pose le plus de problèmes pour des cas réels est la contrainte de différence. Elle exprime que les valeurs affectées à un ensemble de variables doivent être deux à deux différentes. Le célèbre problème des pigeons est un exemple d'une telle contrainte. On retrouve ce type de contraintes dans Iso-ssgrpartiel, puisque dans toute solution un sommet du «grand» graphe ne peut pas être apparié avec deux sommets du «petit» graphe. J'avais remarqué lors de l'étude de Iso-ssgrpartiel pour deux arbres qu'il existait un lien étroit entre la notion d'injection et la notion de couplage en théorie des graphes. L'étude de cette dernière notion m'a permis de trouver un algorithme de complexité polynomiales faible achevant la consistance d'arc généralisée pour les contraintes de différence. J'ai eu l'honneur de présenter ces travaux au congrès AAAI en 1994. Dans le chapitre 7, je les détaille et les étend en proposant un algorithme qui s'avère très efficace pour résoudre Iso-ssgrpartiel.

Pour finir cette partie je présente dans le chapitre 8 deux approches pour résoudre Iso-ssgrpartiel pour des graphes étiquetés.

Enfin, une annexe me semble particulièrement intéressante. Elle concerne les ordres d'instanciation pendant la recherche de solution dans un CSP. Ce domaine est très difficile à appréhender. Néanmoins, je propose un nouvel ordre d'instanciation; je donne un nouvel algorithme de calcul de la largeur minimum d'un graphe et j'essaie de remettre en cause l'ordre le plus communément admis : le domaine minimum. L'ordre présenté est dérivé de considérations générales et personnelles sur la difficulté des CSP. Comme je n'ai pas trop voulu rentrer dans le détail et puisque je me base essentiellement sur des expérimentations, j'ai préféré mettre cet exposé en annexe.

Finalement et pour terminer cette introduction sur une satisfaction personnelle, je

dirai que le résultat que j'obtiens pour un benchmark classique de CSP (l'instance 11 des problèmes d'affectation de fréquences à des noeuds de communication), à savoir trouver une solution sans aucun backtrack[4], me fais très plaisir car j'ai gagné le «concours» entrepris avec T. Schiex de l'INRA de Toulouse.

4. C'est le meilleur résultat obtenu pour ce problème.

Chapitre 2

Présentation du problème

L'isomorphisme de sous-graphes est un problème fondamental en informatique. C'est le premier problème présenté dans le livre sur la complexité algorithmique de Garey et Johnson [Garey and Johnson, 1979]. On le rencontre dans de nombreux domaines comme la reconnaissance des formes [Freuder, 1976; Cheng and Huang, 1981; Wong, 1992; Vosselman, 1992], la chimie organique [Sussenguth, Jr., 1965; Figueras, 1972; Barrow and Burstall, 1976; O'Korn, 1977; Tarjan, 1977; Wipke and Rogers, 1984a; Wipke and Rogers, 1984b; von Scholley, 1984; Willet, 1985; Willet *et al.*, 1986; Willet, 1987; Nicholson *et al.*, 1987; Downs *et al.*, 1988; Barnard, 1988; Armstrong and Hibbert, 1989; Dengler and Ugi, 1991; Barnard, 1991; Barnard, 1993], la classification [Baader *et al.*, 1992; Levinson, 1984; Levinson, 1992; Vismara *et al.*, 1992; Wilcox and Levinson, 1986]. Avec l'apparition de bases de données de graphes, on peut sans risque penser qu'il continuera à être étudié longtemps.

Depuis quelques années, on essaie de plus en plus d'utiliser une représentation symbolique des données. Ce type de représentation s'appuie sur la théorie des graphes. Très souvent les relations d'ordres ou de préordres que l'on a besoin de définir entre les objets tirent parti de la représentation symbolique des ces objets. En général, ces relations s'apparentent au problème de l'isomorphisme de sous-graphes. Résoudre ce problème devient donc une nécessité si l'on veut pouvoir manipuler et organiser des données sous une forme symbolique.

Les graphes que nous considérerons pourront avoir des sommets et des arêtes étiquetés. Cependant, afin de simplifier l'exposé, nous nous intéresserons dans cette

partie uniquement aux graphes non étiquetés. Car l'étiquetage de sommets ou d'arêtes ne modifie pas en théorie la complexité de notre problème. Nous introduirons au chapitre 8 l'étiquetage des sommets et des arêtes.

Comme le problème de l'isomorphisme a fait l'objet de nombreux travaux dans des disciplines très différentes il n'y a pas d'unification des termes. C'est pourquoi nous commencerons par donner une définition formelle de ce problème en distinguant particulièrement les notions de graphe partiel, sous-graphe et sous-graphe partiel. Nous montrerons ensuite que ce problème est NP-Complet dans le cas général et nous présenterons quelques cas connus pour lesquels il existe des algorithmes permettant de le résoudre en temps polynômial.

2.1 Graphes et sous-graphes

Tous les graphes considérés seront, sauf mention particulière, des graphes simples non orientés. Cela ne constitue pas une restriction car la plupart des notions présentées peuvent être facilement utilisées pour des graphes orientés.

Les définitions qui suivent sont extraites des livres de [Berge, 1970], [Harary, 1969] et [Gondran and Minoux, 1985].

Définition 1 *Un **graphe** $G = (X, E)$ est déterminé par la donnée de :*

- *un ensemble X dont les éléments sont appelés **sommets**.*
- *un ensemble E dont les éléments sont des paires de sommets appelées **arêtes**.*

*Un graphe **simple** est sans boucle (arête dont les extrémités coïncident) et ne possède jamais plus d'une arête entre deux sommets quelconques.*
*Deux sommets sont **voisins** ou **adjacents** s'ils appartiennent à la même arête. Les deux sommets d'une arête sont appelés **extrémités** de l'arête. L'ensemble des voisins d'un sommet x est noté $\Gamma(x)$. Le **degré** d'un sommet x, noté $deg(x)$, est le nombre de ses voisins.*

Un graphe simple non orienté est parfaitement déterminé par la donnée de l'ensemble X de ses sommets et par l'application Γ, qui à tout élément de X fait correspondre l'ensemble de ses voisins. On pourra noter un graphe simple non orienté aussi bien $G = (X, \Gamma)$ que $G = (X, E)$. Dans la suite, et par abus de langage, nous

emploierons le terme graphe pour graphe simple.

Définition 2

• *Une* **chaîne** *est une séquence* $(u_1, ..., u_q)$ *d'arêtes de* G *telles que chaque arête de la séquence ait une extrémité en commun avec l'arête qui la précède et l'autre extrémité en commun avec l'arête suivante. On peut également décrire une chaîne par une séquence de sommets* $(x_{1_1}, x_{1_2}, ..., x_{q_1}, x_{q_2})$ *ordonnée suivant la séquence d'arêtes*[1] *définissant la chaîne. Une chaîne est* **simple** *si elle ne contient pas deux fois la même arête. Une chaîne qui n'utilise pas deux fois le même sommet est dite* **élémentaire**.

• *Un* **cycle** *est une chaîne ne contenant pas deux fois la même arête dans la séquence et telle que les deux sommets aux extrémités de la chaîne coïncident*[2].

• *Un graphe* G *est* **connexe** *si pour toute paire de sommets* x, y *il existe au moins une chaîne dans* G *qui relie* x *et* y. *L'existence d'une chaîne entre deux sommets quelconques d'un graphe définit une relation d'équivalence sur les sommets de ce graphe. Les classes d'équivalence de cette relation constituent les* **composantes connexes** *du graphe.*

• *Un graphe connexe sans cycle est appelé un* **arbre**. *Un graphe non connexe sans cycle est appelé* **forêt**.

Les notions de graphe partiel, sous-graphe et sous-graphe partiel ne sont pas toujours bien distinguées dans la littérature. Le terme anglais *subgraph* désigne selon les auteurs un sous-graphe ou bien un sous-graphe partiel. Afin qu'il n'y ait pas de confusion, nous rappelons les définitions de ces termes.

Définition 3 *Soient* $G = (X, E)$ *un graphe,* Y *un sous-ensemble de* X *et* F *un sous-ensemble de* E.

• *Le* **sous-graphe**[3] *engendré par* Y *est le graphe* G_Y *dont les sommets sont les éléments de* Y *et dont les arêtes sont celles de* G *qui ont leurs deux extrémités dans* Y.

• *Le* **graphe partiel**[4] *engendré par* F *est le graphe ayant le même ensemble de*

1. x_{i_1} et x_{i_2} représentent les deux extrémités d'une arête.

2. Cette définition s'entend à une permutation près. En effet, si (a, b, c, d, a) est un cycle (b, c, d, a, b) décrit le même cycle.

3. *subgraph induced by a given subset of vertices* en anglais.

4. *subgraph spanned by a given subset of edges* en anglais.

sommets X que G, et dont les arêtes sont celles de F (on élimine de G les arêtes de E - F).

• *Le* ***sous-graphe partiel***[5] *engendré par Y et F est le graphe partiel de G_Y engendré par F.*

Il est indispensable de bien faire la distinction entre ces trois notions. La figure 2.1 illustre ces différences.

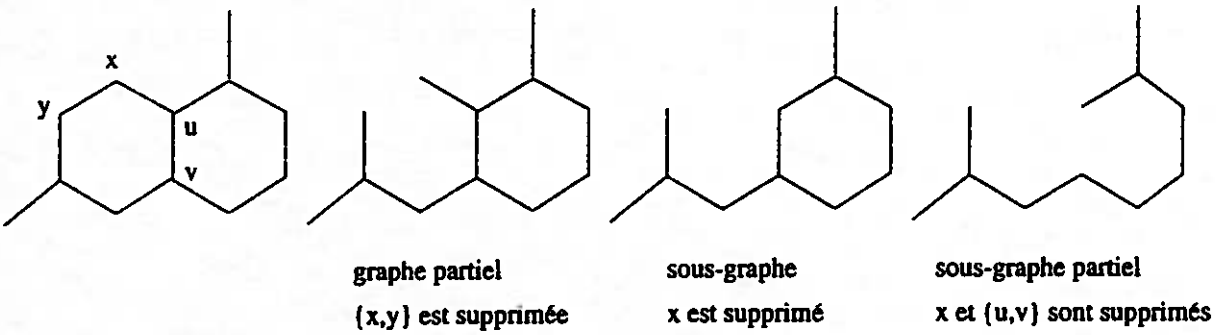


FIG. 2.1 - *Graphe partiel, sous-graphe et sous-graphe partiel d'un graphe*

Intuitivement, à partir d'un graphe $G = (X, E)$, on peut les voir de la manière suivante :

• un sous-graphe de G s'obtient en supprimant des sommets de G. Lorsque l'on enlève un sommet x on retire aussi toutes les arêtes dont l'une des extrémités est x.

• un graphe partiel de G s'obtient en supprimant des arêtes de G. On ne supprime aucun sommet.

• un sous-graphe partiel de G s'obtient en supprimant des sommets et des arêtes de G. On peut retirer du graphe ce que l'on veut.

Tout sous-graphe est un sous-graphe partiel particulier, et tout graphe partiel est un sous-graphe partiel particulier. La notion de sous-graphe partiel est donc plus générale que les deux autres. Ce qui distingue fondamentalement ces concepts c'est la modification du rapport d'adjacence entre les sommets (existence ou non d'une arête entre les sommets). Cela s'exprime par la propriété suivante :

Propriété 1 *Soient $G = (X, E)$ un graphe, $G_{sg} = (Y, E_Y)$ le sous graphe de G engendré par Y et $G_p = (X, F)$ le graphe partiel de G engendré par un sous-ensemble*

5. *subgraph* ou *partial subgraph* en anglais.

F non trivial[6] *de E. Quels que soient x et y de Y, les deux propriétés suivantes sont vérifiées :*

1 $(x,y) \in E \Leftrightarrow (x,y) \in E_Y$

2 $(x,y) \in F \Rightarrow (x,y) \in E$

Dans le premier cas, on dit que le rapport d'adjacence est ***fortement*** *préservé. Alors que dans le second cas il est dit* ***faiblement*** *préservé.*

Définition 4 *Soient $H = (X_H, E_H)$ et $G = (X_G, E_G)$ deux graphes :*

• *On appelle* ***fonction qui préserve fortement l'adjacence*** *de H dans G, une fonction $f : X_H \to X_G$ telle que : $\forall a, b \in X_H : (a,b) \in E_H \Leftrightarrow (f(a), f(b)) \in E_G$.*

• *On appelle* ***fonction qui préserve faiblement l'adjacence, ou homomorphisme, ou morphisme,*** *de H dans G, une fonction $f : X_H \to X_G$ telle que : $\forall a, b \in X_H : (a,b) \in E_H \Rightarrow (f(a), f(b)) \in E_G$.*

Un morphisme bijectif est appelé ***isomorphisme***. Il est important de remarquer que pour un homomorphisme on ne peut rien déduire de la présence d'une arête dans E_G pour la présence d'une arête dans E_H. Par contre dans le cas de l'isomorphisme, l'application est une bijection donc sa réciproque l'est aussi et par conséquent l'adjacence est fortement préservée. La définition de l'isomorphisme se déduit de celle du morphisme :

Définition 5 *Deux graphes $H = (X_H, E_H)$ et $G = (X_G, E_G)$ sont* ***isomorphes*** *si et seulement s'il existe une bijection $f : X_H \to X_G$ telle que :*

$$\forall a, b \in X_H : (a,b) \in E_H \Leftrightarrow (f(a), f(b)) \in E_G.$$

On peut maintenant chercher s'il existe entre deux graphes donnés une relation de sous-graphe, graphe partiel ou sous-graphe partiel. La recherche d'une de ces relations peut être vue selon deux points de vue différents. Soit on recherche une relation directe entre G et H. Soit on recherche une relation entre l'un des graphes et un graphe que contient l'autre. La manière de poser ce problème variant suivant le domaine d'application notre présentation se fera selon ces deux points de vue.

6. Un sous-ensemble F d'un ensemble E est non trivial s'il est différent de l'ensemble vide et de E.

2.1.1 Isomorphisme de sous-graphes partiels

Si l'on recherche une relation directe entre H et G, le problème est celui de la recherche d'un morphisme injectif de H dans G :

Iso-ssgrpartiel(H, G) :

Données : $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes.
Question : existe-t-il un morphisme injectif de H dans G?

L'autre possibilité consiste à rechercher une relation entre H et un sous-graphe de G. Le problème se ramène à celui de l'isomorphisme de sous-graphes partiels :

Iso-ssgrpartiel(H, G) :

Données : $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes.
Question : G contient-il un sous-graphe partiel isomorphe à H, c'est-à-dire un sous-ensemble X'_G de sommets et un sous-ensemble E'_G d'arêtes tels que $|X'_G| = |X_H|, |E'_G| = |E_H|$ et il existe une bijection $f : X_H \to X'_G$ satisfaisant $\{u, v\} \in E_H \Leftrightarrow \{f(u), f(v)\} \in E'_G$?

Dans la suite nous travaillerons avec le premier point de vue.

2.1.2 Isomorphisme de sous-graphes

Dans ce cas nous auront le problème de la recherche d'une injection qui préserve fortement l'adjacence de H dans G et celui de l'isomorphisme de sous-graphes.

Iso-ssgr(H, G) :

Données : $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes.
Question : existe-t-il une injection qui préserve fortement l'adjacence de H dans G?

Iso-ssgr(H, G) :

Données : $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes.
Question : G contient-il un sous-graphe isomorphe à H, c'est-à-dire un

sous-ensemble X'_G de sommets tel que $|X'_G| = |X_H|$ et $|E'_G| = |E_H|$, où E'_G est l'ensemble des arêtes de E_G ayant leurs deux extrémités dans X'_G, et tel qu'il existe une bijection $f : X_H \rightarrow X'_G$ satisfaisant $\{u, v\} \in E_H \Leftrightarrow \{f(u), f(v)\} \in E'_G$?

Les problèmes que nous avons considérés sont des problèmes de décision. La réponse à ces problèmes appartient à {OUI,NON} ou {VRAI,FAUX}. L'objectif de ces problèmes est la recherche de l'existence d'une fonction particulière entre deux graphes. Si par contre on désire connaître cette fonction, on dira que l'on recherche un ***plongement*** de H dans G:

Définition 6 *Soient G et H deux graphes, si H est isomorphe à un sous- graphe partiel G' de G, alors l'isomorphisme $\phi(H) = G'$ est appelé **plongement**[7] de H dans G et l'image d'un sommet h de H par l'isomorphisme est appelé **appariement** de h.*

2.2 Complexité du problème

Nous allons rappeler que le problème est NP-Complet dans le cas général. Ensuite nous essayerons d'estimer la frontière entre les classes d'instances polynomiales et celles NP-Complètes. Nous présenterons, alors, un type de graphe très particulier pour lequel Iso-ssgrpartiel est NP-Complet tandis que Iso-ssgr est polynomial.

2.2.1 NP-Complétude du problème

D'après [Garey and Johnson, 1979] p.105, le problème de l'isomorphisme de sous-graphe, partiel ou non, est classé dans le cas général comme étant NP-Complet. En effet le problème de l'isomorphisme de sous-graphes est équivalent au problème de la recherche d'une clique de taille k dans un graphe. Nous allons détailler cette équivalence car certains auteurs utilisent la recherche de clique pour résoudre le problème de l'isomorphisme de sous-graphe.

7. Plongement est une traduction du terme anglais *embedding*.

Avant tout nous rappelons quelques définitions (le lecteur trouvera un excellent exposé sur les problèmes NP-Complets dans le livre de [Garey and Johnson, 1979], et notamment page 35 les définitions suivantes) :

Définition 7 *Soient P et Q deux problèmes de décision, Q se* ***réduit polynomialement*** *en P s'il existe une fonction g calculable en un temps polynomial telle que pour toute solution x de P, $y = g(x)$ soit solution de Q.*
On appelle g ***fonction de réduction.***

Si Q se réduit polynomialement à P, alors si l'on sait résoudre P on sait aussi résoudre Q. Cela signifie donc que P est au moins aussi difficile que Q.

Propriété 2 *Si P est réductible à Q et si Q est réductible à P alors les deux problèmes sont dits* ***équivalents.***

Le problème de la recherche d'une clique (càd un sous-graphe complet) de taille k dans un graphe est le suivant :

Clique(G, k) :

Données : Soient G un graphe et k un entier.
Question : G contient-il une clique de taille k ?

Il apparait clairement que le problème de la recherche d'une clique de taille k se réduit polynomialement au problème de l'isomorphisme de sous-graphe, partiel ou non. Comme Clique est NP-Complet, Iso-ssgr l'est aussi. Pour les raisons que nous avons exposées précédemment, nous allons présenter une réduction de Iso-ssgr en Clique, c'est-à-dire exhiber une fonction calculable en temps polynomial qui permet de passer d'une solution de Clique à une solution Iso-ssgr. Que ce soit pour le cas partiel ou pour le cas non partiel, la démonstration s'appuie sur la construction d'un graphe produit des deux graphes. La définition de ce graphe diffère légèrement selon le problème traité.

2.2.2 Cas non partiel

À partir d'une instance du problème de l'isomorphisme de sous-graphes partiels,

on peut construire un graphe particulier nommé graphe produit d'adjacence forte[8].

Définition 8 *Le **graphe produit d'adjacence forte** de deux graphes $H = (X_H, E_H)$ et $G = (X_G, E_G)$ est noté $H \times G$ et il est défini par le graphe $H \times G = (X_{H\times G}, E_{H\times G})$ où :*

- *l'ensemble de sommets $X_{H\times G}$ est égal au produit cartésien de X_H par X_G : $X_{H\times G} = X_H \times X_G$.*
- *l'ensemble d'arêtes $E_{H\times G}$, où $\{(x_H, x_G), (y_H, y_G)\} \in E_{H\times G}$ si et seulement si $x_H \neq y_H$ et $x_G \neq y_G$ et l'une des deux propriétés suivantes est vraie :*
  1 $\{x_H, y_H\} \in E_H$ *et* $\{x_G, y_G\} \in E_G$
  2 $\{x_H, y_H\} \notin E_H$ *et* $\{x_G, y_G\} \notin E_G$.

La proposition suivante nous fournit l'équivalence recherchée :

Proposition 1 *Soient $G = (X_G, E_G)$, $H = (X_H, E_H)$ deux graphes et $H \times G$ le graphe produit d'adjacence forte de H par G, alors :*
à toute clique de taille $|X_H|$ du graphe $H \times G$ correspond une injection de H dans G qui préserve fortement l'adjacence, et réciproquement.

preuve :

Considérons $K_{|X_H|} = \{(h_{i_1}, g_{j_1}), ..., (h_{i_{|X_H|}}, g_{j_{|X_H|}})\}$ une clique de taille $|X_H|$ du graphe $H \times G$. D'après la construction de ce graphe : $\forall p, q$ de 1 à $|X_H|$ et $p \neq q$ on a : $h_{i_p} \neq h_{i_q}$ et $g_{j_p} \neq g_{j_q}$. Aussi la fonction de X_H dans X_G qui fait correspondre g_{j_p} à h_{i_p} est une injection. D'après les propriétés 1 et 2 de construction du graphe produit, elle préserve fortement l'adjacence. La réciproque est évidente. ⊙

2.2.3 Cas partiel

On construit de manière analogue au cas non partiel un graphe produit d'adjacence faible.

Définition 9 *Le **graphe produit d'adjacence faible** de deux graphes $H = (X_H, E_H)$ et $G = (X_G, E_G)$ est noté $H \otimes G$ et définit par le graphe $H \times G = (X_{H\otimes G}, E_{H\otimes G})$ où :*

8. Ce graphe est parfois appelé graphe de compatibilité.

• *l'ensemble de sommets* $X_{H\otimes G}$ *est égal au produit cartésien de* X_H *par* X_G *:* $X_{H\otimes G} = X_H \times X_G$.

• *l'ensemble d'arêtes* $E_{H\otimes G}$, *où* $\{(x_H, x_G), (y_H, y_G)\} \in E_{H\otimes G}$ *si et seulement si* $x_H \neq y_H$ *et* $x_G \neq y_G$ *et l'une des deux propriétés suivantes est vraie :*

1 $\{x_H, y_H\} \in E_H$ *et* $\{x_G, y_G\} \in E_G$

2 $\{x_H, y_H\} \notin E_H$.

La proposition suivante nous fournit l'équivalence recherchée :

Proposition 2 *Soient* $G = (X_G, E_G)$, $H = (X_H, E_H)$ *deux graphes et* $H \otimes G$ *le graphe produit d'adjacence faible de* H *par* G, *alors :*
à toute clique de taille $|X_H|$ *du graphe* $H \otimes G$ *correspond un monomorphisme de* H *dans* G, *et réciproquement.*

Le problème est donc NP-Complet dans le cas général. Nous allons maintenant nous intéresser aux classes d'instances pour lesquelles le problème est polynomial. Nous donnerons ainsi une idée de la frontière entre les classes d'instances «faciles» et celles «difficiles».

2.2.4 Frontière entre classes d'instances polynomiales et classes d'instances NP-Complètes

La difficulté du problème peut être caractérisée en fonction des propriétés de G et de H.

G et H sont des arbres

Si G et H sont des arbres (graphes connexes ne contenant pas de cycle), alors le problème est polynomial. Matula [Matula, 1978] et Moon Jung Chung [Moon Jung Chung, 1987] ont proposé un algorithme en $O(n_H^{1,5} n_G)$. Si l'on modifie légèrement la structure de G et de H nous allons avoir une première idée de la frontière entre les classes d'instances polynomiales et celles NP-Complètes.

Qu'en est-il de complexité du problème si l'on impose qu'un seul des deux graphes soit un arbre?

Seul G est un arbre

Considérons, dans un premier temps, que G est un arbre. Le seul cas non trivial et non encore traité est celui où H est une forêt (graphe non connexe dont chaque composante est un arbre). [Garey and Johnson, 1979] montrent, page 105, que ce problème appelé isomorphisme de sous-forêts est NP-Complet.

Seul H est un arbre

Considérons maintenant que H est un arbre. Dans ce cas Iso-ssgrpartiel contient un problème classique de la théorie des graphes: la recherche d'une chaine hamiltonienne dans un graphe.

Chaine-hamiltonienne(G):

> Données: $G = (X, E)$ un graphe.
> Question: G contient-il une chaîne hamiltonienne, c'est-à-dire une chaîne qui passe une et une seule fois par chaque sommet de X?

[Garey and Johnson, 1979] montrent (pages 56–60) que ce problème est NP-Complet. Donc si G est un graphe et si H est une chaîne, alors Iso-ssgrpartiel reste NP-Complet.

G et H sont planaires

Une classe particulière de graphe pour laquelle de nombreux problèmes se résolvent en temps polynomiaux et qui se rencontre souvent dans les applications réelles a fait l'objet de nombreux travaux. Elle est constituée des graphes planaires. Le problème de l'isomorphisme de graphes, non classé dans le cas général, est polynomial pour les graphes planaires [Hopcroft and Tarjan, 1980]. Malheureusement, Iso-ssgrpartiel est NP-Complet pour ces graphes. En effet le problème de la recherche d'une chaîne hamiltonienne reste NP-Complet pour ce type de graphe [Garey *et al.*, 1976]. Néanmoins, pour certains graphes planaires particuliers, Iso-ssgr devient polynomial. Aussi surprenant que cela puisse paraître, Iso-ssgrpartiel demeure NP-Complet pour ces graphes. L'un des thèmes de recherche actuel est de déterminer la famille maximale de graphes planaires pour lesquels Iso-ssgr est polynomial

[Lingas and Syslo, 1988]. Deux types de graphes planaires pour lesquels le problème est polynomial sont connus : les graphes 2-connexes outerplanar[9][Syslo, 1982; Lingas, 1986] et les graphes 2-connexes serie-parallèle [Lingas and Syslo, 1988]. Le premier type est en fait une spécialisation du second.

La frontière entre classes d'instances polynomiales et classes d'instances NP-Complètes peut donc être déterminée en fonction de la connexité, de la présence de cycle et de la planarité.

Pour Iso-ssgrpartiel nous avons la proposition :

Proposition 3 *Soient G et H deux graphes non orientés qui ne sont pas des chaînes. Si les trois propriétés suivantes sont vraies alors* Iso-ssgrpartiel(H, G) *se résout en temps polynomial, sinon* Iso-ssgrpartiel(H, G) *est NP-Complet.*

- *H est connexe,*
- *H est sans cycle,*
- *G est sans cycle.*

Cette frontière est différente pour Iso-ssgr comme le montre la proposition suivante :

Proposition 4 *Soient G et H deux graphes non orientés qui ne sont pas des chaînes.* Iso-ssgr(H, G) *est NP-Complet si l'un des deux cas suivants est vrai :*

- *H n'est pas connexe,*
- *G n'est pas planaire*[10].

Iso-ssgr(H, G) *se résout en temps polynomial si l'une des propriétés suivantes est vrai :*

- Iso-ssgrpartiel(H, G) *se résout en temps polynomial,*
- *H est 2-connexe serie-parallèle et G est 2-connexe serie-parallèle.*

2.3 Conclusion

Iso-ssgrpartiel est le plus général des problèmes d'isomorphisme de sous-graphes. C'est aussi le plus difficile. En effet même pour des graphes connexes, il suffit, dans le cas général, que l'un des deux graphes possède un cycle pour que le problème soit NP-Complet.

9. Les graphes outerplanars et serie-parallèles sont définis en annexe.

10. Si G n'est pas planaire, alors il possède au moins un cycle.

Chapitre 3

Rapide survol des différentes techniques de résolution

En reconnaissance des formes, l'isomorphisme de sous-graphes a été abondamment étudié dans les années 1970 et au début des années 1980. Ces études étaient motivées par le désir de retrouver des objets dans des images. Malheureusement les temps de calculs se sont avérés trop longs pour que ces méthodes puissent être employées en pratique. C'est pourquoi certains se sont orientés vers des méthodes approximatives [Shapiro and Haralick, 1981]. Cependant l'idée d'une recherche exacte n'a jamais été totalement abandonnée, on trouve encore récemment des publications sur ce sujet [Wong, 1992].

En apprentissage, les données de type symbolique sont souvent représentées par des graphes. De plus, il est nécessaire de disposer d'une fonction qui permette de comparer deux objets, afin de savoir si l'un est plus général que l'autre. Autrement dit, il faut résoudre Iso-ssgrpartiel.

Par ailleurs ce problème a toujours intéressé les chimistes. En effet, disposant d'énormes bases de données de molécules (5 à 6 millions d'éléments), ils souhaitent pouvoir connaître l'ensemble des molécules qui possèdent une sous-structure particulière. Comme les formules moléculaires peuvent être représentées par des graphes dont les sommets et les arêtes sont étiquetés, on voit facilement le lien avec notre sujet.

Nous distinguerons deux parties importantes, l'une sera consacrée aux principes

des algorithmes qui permettent de résoudre les cas polynômiaux, tandis que l'autre étudiera les méthodes de résolution du problème dans le cas général, c'est-à-dire lorsqu'il est NP-Complet. Dans la première partie de ce chapitre et dans le chapitre 7 de cette thèse, nous utiliserons la théorie des couplages; c'est pourquoi nous commencerons par introduire quelques définitions de ce domaine.

3.1 Présentation de la théorie des couplages

La théorie des couplages a été introduite pour résoudre les problèmes d'affectation [Berge, 1970] :

> "Dans une entreprise il y a p personnes et t tâches à effectuer. Une personne est qualifiée pour exécuter certaines tâches. Le problème est d'affecter chaque personne à une tâche, de telle façon que deux personnes ne soient pas affectées à la même tâche."

Définition 10 *Étant donné un graphe simple $G = (V, E)$ non orienté, un* ***couplage*** *est un sous-ensemble d'arêtes $M \subseteq E$ tel que deux arêtes quelconques de M ne soient pas adjacentes.*

Le nombre d'arête d'un couplage est appelé ***cardinalité*** du couplage. Un couplage maximum est un couplage de cardinalité maximum. Un couplage M ***couvre*** ou ***sature*** un ensemble X de sommets, si tout sommet de X est l'extrémité d'une arête de M. Les sommets qui ne sont pas une extrémité d'une arête d'un couplage M sont dits ***insaturés*** par M ou ***M-insaturés***.

Nous limiterons notre étude aux couplages dans les graphes bipartis.

Définition 11 *Un graphe est biparti si l'ensemble de ses sommets V peut être partitionné en deux sous-ensembles X et Y de telle sorte que, pour toute arête $(i, j) \in E$ on ait :*

(i) $i \in X \Rightarrow j \in Y$

(ii) $i \in Y \Rightarrow j \in X$.

Pour un graphe biparti $G = (X, Y, E)$, un couplage couvrant X est appelé ***couplage de X dans Y***. La figure 3.1 présente deux graphes bipartis, pour le premier le

couplage maximum couvre X tandis que pour le second le couplage maximum laisse des sommets insaturés dans chacune des partitions.

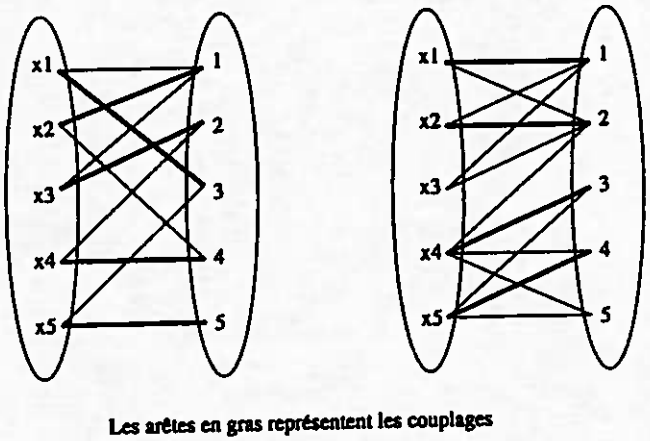


FIG. 3.1 - *Exemples de couplages maximum pour des graphes bipartis.*

Pour résoudre les problèmes d'affectation, on construit un graphe biparti $G = (P, T, E)$, où P représente l'ensemble des personnes, T l'ensemble des tâches et où il existe une arête (p, t) si et seulement si la personne p est qualifiée pour la tâche t. Résoudre le problème revient à rechercher dans G un couplage de cardinalité p.

Si l'on imagine des variables à la place des personnes et des valeurs à la place des tâches, alors nous pourrons résoudre les problèmes d'affectation de variables avec des valeurs différentes. La cardinalité du couplage maximum nous donnera le nombre maximum de variables qui peuvent être affectées simultanément à des valeurs différentes. Si le couplage couvre l'ensemble X des variables, cela signifie que l'on peut affecter une valeur différente à chaque variable du problème. Il y a donc un ***rapport étroit entre la notion de couplage couvrant un ensemble et une injection.*** Comme le problème de l'isomorphisme de sous-graphe est équivalent à la recherche d'une injection possédant certaines propriétés, on voit donc bien le lien avec notre problème.

La notion de couplage apporte donc une autre vision pour la modélisation du problème. Elle permet aussi une résolution plus aisée car le calcul d'un couplage maximum pour un graphe s'effectue en un temps polynômial. En effet la recherche d'un couplage maximum dans un graphe biparti est une application particulière du problème du flot maximum. Des algorithmes indépendants ont été trouvés mais Even et Tarjan [Even and Tarjan, 1975] ont montré que tous les algorithmes développés

spécialement pour ce problème se déduisaient des algorithmes sur les flots soit de Ford et Fulkerson [Ford and Fulkerson, 1962], soit de Dinic [Dinic, 1970]. L'algorithme le plus connu et certainement le plus utilisé est l'algorithme de Hopcroft et Karp [Hopcroft and Karp, 1973], dont la complexité dans le pire des cas est $O(m\sqrt{n})$, pour un graphe biparti de m arêtes et n sommets. Mais le meilleur algorithme est dû à Alt, Blum, Melhorn et Paul [Alt *et al.*, 1991] qui est en $O(n^{1,5}\sqrt{m/\log n})$. Cet algorithme utilise une structure de données astucieuse qui permet de réduire la complexité dans le pire des cas de l'algorithme précédent. La réduction peut aller jusqu'à un facteur $\sqrt{\log n}$ pour les graphes denses. Mais il n'est pas certain qu'en moyenne cela soit aussi vrai, c'est pourquoi nous nous baserons uniquement sur celui de Hopcroft et Karp. Nous allons nous attarder un peu sur la complexité de ce dernier algorithme, car elle est souvent présentée sous des formes différentes ce qui prête à confusion.

La complexité C dans le pire des cas s'exprime de manière plus précise par $O(m\sqrt{s})$ où m est le nombre d'arêtes du graphe et s la cardinalité du couplage maximum. Pour un graphe biparti $G = (X, Y, E)$, C sera en $O(|E| \times \sqrt{min(|X|, |Y|)})$. Supposons que X ait moins d'éléments que Y et notons leurs cardinaux respectivement par n_X et n_Y, alors C pourra être exprimé par :

$$O(m\sqrt{n_X}) \leq O(n_X^{1,5} n_Y).$$

Nous considérerons dans la suite que la fonction CALCULCOUPLAGEMAX(G) retourne un couplage maximum du graphe G.

3.2 Principe des algorithmes polynômiaux

Nous avons décidé de présenter de manière détaillée l'algorithme de Iso-ssarbre pour permettre à des utilisateurs non familiers du domaine de les implémenter facilement. Ce choix se justifie d'autant plus que les deux articles sur lesquels nous nous sommes appuyés sont d'un abord assez difficile. En effet l'article de Matula [Matula, 1978] est très complexe et celui de Moon Jung Chung [Moon Jung Chung, 1987] présente l'algorithme de résolution de Iso-ssarbre comme une dérivation possible de celui qu'il propose pour résoudre le problème de l'homéomorphisme d'arbre.

Que ce soit pour les arbres ou les graphes outerplanars ou serie-parallèles, les algorithmes font appel aux mêmes techniques. La méthode de résolution utilisée est

toujours la programmation dynamique.

Le problème Iso-ssarbre initial se décompose récursivement en une collection de sous-problèmes Iso-ssarbre pour des arbres plus petits. Ces sous-problèmes, pris dans un ordre approprié, peuvent être résolus par la recherche d'un couplage maximum dans un graphe biparti.

Après une présentation détaillée l'algorithme pour les arbres possédant des racines, nous donnerons les principes de l'algorithme de même complexité qui permet de traiter Iso-ssarbre pour des arbres sans racines.

3.2.1 Arbres avec racines

L'algorithme exposé est inspiré de [Matula, 1978] et [Moon Jung Chung, 1987]. Il détermine si un arbre P_r muni d'une racine r est un sous-arbre d'un arbre $G_{r'}$ muni d'une racine r' ***sans nécessairement faire coïncider*** les racines. Sur la figure 3.2, toute fonction de collage de P_a dans $G_{a'}$ associe a à a', tandis que toute fonction de collage de $Q_{a''}$ dans $G_{a'}$ n'associe jamais a'' à a'.

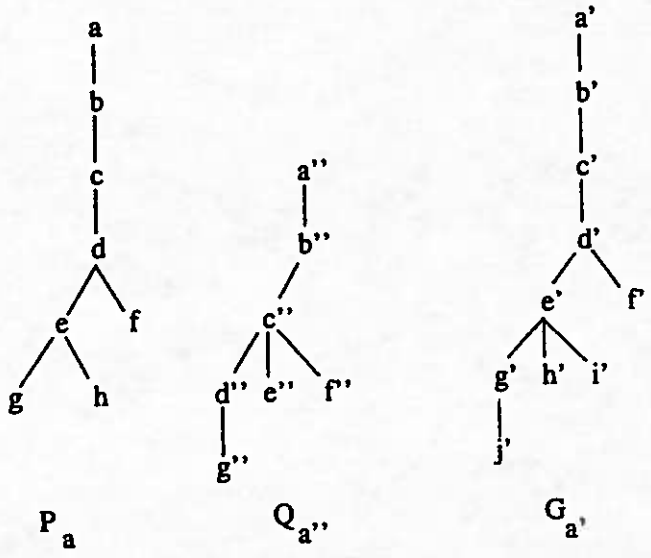


FIG. 3.2 - *Deux sous-arbres d'un arbre.*

Par cohérence avec les algorithmes qui suivront dans cette thèse, nous avons fait le choix de présenter un algorithme qui cherche à déterminer les sommets de P_r qui peuvent être appariés avec les sommets de $G_{r'}$ contrairement à tous les algorithmes proposés jusqu'à maintenant qui raisonnent à partir des sommets de $G_{r'}$.

De manière informelle nous pouvons donner les principes de l'algorithme à partir des deux arbres $G_{a'}$ et P_a de la figure 3.3.

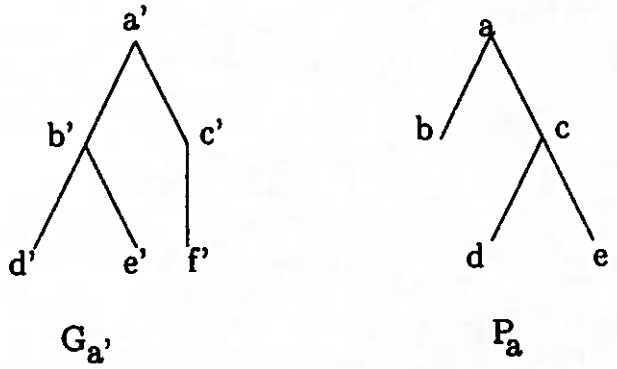


FIG. 3.3 - *Un exemple du problème de l'isomorphisme de sous-arbres.*

Les sommets de P_a sont étudiés «en largeur» des feuilles vers la racine. Un noeud p de P_a peut être apparié avec un noeud g de $G_{a'}$ si les fils de p peuvent s'apparier simultanément avec les fils de g, c'est-à-dire que l'on doit pouvoir apparier chaque fils de p avec un fils différent de g. Le but est donc de pouvoir apparier la racine de P_a avec un sommet de $G_{a'}$.

Les feuilles de P_a, puisqu'elles n'ont pas de fils, peuvent s'apparier avec n'importe quel sommet de $G_{a'}$. Ainsi b, d et e peuvent s'apparier avec les valeurs a', b', c', d', e', f'. On étudie ensuite les sommets dont les appariements sont connus pour l'ensemble de leurs fils. Dans notre exemple seul c remplit cette condition. c peut être apparié avec un sommet g de $G_{a'}$ s'il est possible d'apparier simultanément les fils de c avec les fils de g. Pour résoudre ce problème de simultanéité, on construit un graphe biparti dont les ensembles de sommets sont les fils de c et les fils de g. Un fils f_c de c est relié à un fils f_g de g si et seulement si f_c peut être apparié avec f_g. Un couplage de ce graphe biparti saturant l'ensemble des fils de c garantit qu'on peut apparier les fils de c, chacun avec un fils de g différent. Si un tel couplage existe, alors c pourra être apparié avec g, sinon il ne le pourra pas. Dans notre exemple nous trouvons que c peut être apparié avec b' ou a', mais pas avec c'. La figure 3.4 présente les graphes bipartis obtenus lors de cette recherche.

Si l'on répète l'opération précédente pour tous les sommets dont les fils ont été étudiés, alors on rencontrera l'une des deux possibilités suivantes : soit un sommet de

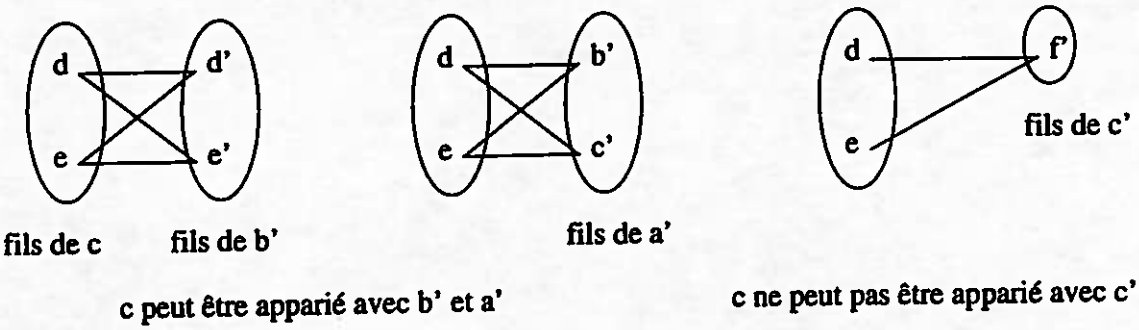


FIG. 3.4 - *Appariements possibles et impossibles.*

P_a ne peut être apparié avec aucun sommet de $G_{a'}$ et dans ce cas P_a n'est pas un sous-arbre de $G_{a'}$, soit on peut apparier la racine de P_a avec un sommet de $G_{a'}$ (pas nécessairement la racine) et P_a est un sous-arbre de $G_{a'}$.

Dans notre exemple nous arriverons à apparier a avec a', donc P_a est un sous-arbre de $G_{a'}$.

La présentation qui suit maintenant est plus formelle. Nous considérerons que $G_{r'} = (X_G, E_G, r')$ et $P_r = (X_P, E_P, r)$ sont deux arbres orientés à partir de leur racine.

Un sommet y est un ***descendant*** d'un sommet x s'il existe un chemin de x à y. L'ensemble des descendants d'un sommet x est noté $D(x)$. $F(x)$ désigne l'ensemble des fils d'un sommet x. $G_{r'}(g)$ représente le sous-arbre de $G_{r'}$ engendré par l'ensemble de sommets $\{g\} \cup D(g)$. Pour chaque sommet p de P_r on définit par $S^r(p)$ l'ensemble des sommets g de $G_{r'}$ tel que le sous-arbre de P_r engendré par p est un sous-arbre du sous-arbre de $G_{r'}$ engendré par g. Formellement nous avons :

$$S^r(p) = \{g \in X_G \text{ tel que } \exists \text{ un morphisme injectif } m : P_r(p) \to G_{r'}(g) \text{ avec } m(p) = g\}$$

Si $S^r(r)$ est non vide alors $P_r(r)$ est isomorphe à un sous-arbre de $G_{r'}$. Le but de notre algorithme est donc de calculer $S^r(r)$.

Etant donné deux arbres $P_r(p)$ et $G_{r'}(g)$ on peut leur associer un graphe biparti $B_{p,g} = (F(p) \cup F(g), E)$ où deux fils f_p et f_g sont reliés s'il existe un morphisme m de $P_r(f_p)$ dans $G_{r'}(f_g)$ qui associe ces deux fils $(m(f_p) = f_g)$. De manière plus formelle cela s'écrit :

$$(f_p, f_g) \in E \Leftrightarrow f_g \in S^r(f_p)$$

Nous pouvons maintenant introduire la proposition centrale sur laquelle s'appuie l'algorithme et qui va nous permettre de construire les ensembles $S^r(p)$:

Proposition 5 (Moon Jung Chung, 1987) $g \in S^r(p) \Leftrightarrow$ *il existe un couplage saturant* $F(p)$ *dans* $B_{p,g}$.

preuve :

$\Rightarrow$: évident.

$\Leftarrow$: D'après la construction des arêtes de $B_{p,g}$ un couplage saturant $F(p)$ définit une injection i de $F(p)$ dans $F(g)$ avec $i(f_p) = f_g \Leftrightarrow f_g \in S^r(f_p)$. Or $f_g \in S^r(f_p)$ signifie qu'il existe un morphisme m_{f_p,f_g} de $P_r(f_p)$ dans $G_{r'}(f_g)$ tel que $m(f_p) = f_g$. Comme P_r et $G_{r'}$ sont deux arbres, on peut définir une fonction h de $P_r(p)$ dans $G_{r'}(g)$ telle que :

- $h(p) = g$
- $\forall s \in F(p) : h(s) = i(s)$
- $\forall s \in D(f_p)$ avec $f_p \in F(p) : h(s) = m_{f_p,i(f_g)}(s)$

h un morphisme, donc $g \in S^r(p)$ $\odot$

D'après la construction du graphe biparti, nous avons immédiatement un algorithme récursif. Il est plus intéressant de le présenter de manière itérative en commençant par les feuilles (cf. algorithme 1.

La complexité de CALCULS repose sur la complexité de la construction d'un graphe biparti et du calcul d'un couplage maximum dans ce graphe. Pour deux sommets p et g le coût de la construction d'un graphe $B_{p,g}$ est en $O(F(p)F(g))$. La recherche d'un couplage saturant $F(p)$ dans ce graphe est en $O(|F(g)||F(p)|^{1,5})$. La complexité de CALCULS est donc en $O(\sum_{g \in X_g} |F(g)||F(p)|^{1,5})$.

La complexité de SOUSARBREAVECRACINE est donc en :

$$O(\sum_{p \in X_P} \sum_{g \in X_g} |F(g)||F(p)|^{1,5}) \leq O(n_G n_P^{1,5})$$

3.2.2 Arbres sans racines

À partir de l'algorithme précédent, on peut résoudre facilement le problème pour les arbres sans racines en fixant une racine arbitraire pour G et en essayant succes-

Algorithme 1 Algorithme de résolution du problème de l'isomorphisme de sous-arbres.

SOUSARBREAVECRACINE(i $G_{r'}, P_r$) : **booléen**
  $\forall p \in X_P : S^r(p) \leftarrow \varnothing$; *marque*$(p) \leftarrow$ **faux**
  $\forall p$ feuille de $X_P : S^r(p) \leftarrow X_G$; *marque*$(p) \leftarrow$ **vrai**
  pour *chaque* $p \in X_P$ *tel que* $\neg$ *marque*(p) *et* $\forall f_p \in F(p)$: *marque*(f_p) **faire**
    CALCULS$(S^r(p), r, p, G_{r'})$
    marque$(p) \leftarrow$ **vrai**
  si $S^r(p) = \varnothing$ **alors retourner faux**
  sinon retourner vrai

CALCULS(**io** $S^r(p)$, **i** $r, p, G_{r'}$)
  pour *chaque* $g \in X_G$ **faire**
    CONSTRUIRE$(B_{p,g})$
    $m \leftarrow$ CALCULCOUPLAGEMAX$(B_{p,g})$
    si $|m| = |F(p)|$ **alors** AJOUTER$(g, S^r(p))$

sivement tous les sommets de P comme racine. L'idée de Matula et de Moon Jung Chung est de choisir intelligemment une nouvelle racine en prenant un sommet voisin de la racine courante.

Supposons que l'algorithme précédent ait été appelé pour l'arbre P avec a comme racine (cf figure 3.5). Considérons maintenant b comme nouvelle racine.

On remarque que seule l'orientation d'une arête a été changée. Celle de l'arête entre a et b, c'est-à-dire entre l'ancienne et la nouvelle racine. Par rapport à l'algorithme des arbres avec racine cela signifie que si une nouvelle racine b est choisie parmi les voisins de la précédente racine a alors on a besoin de recalculer uniquement $S^b(a)$ et $S^b(b)$ car pour tous les autres sommets x on a $S^b(x) = S^a(x)$. On peut garantir le choix d'une nouvelle racine voisine de la racine précédente par un parcours en profondeur de l'arbre. Cette jolie idée va permettre d'obtenir la même complexité dans le pire des cas pour les arbres sans racines qu'avec les arbres avec racines. Matula et Moon Jung Chung montrent en utilisant une propriété des couplages[1] qu'il est possible de

1. Nous ne détaillerons pas cette propriété ici car il nous a semblé qu'elle sortait du cadre de cette thèse. Elle est clairement expliquée dans [Moon Jung Chung, 1987].

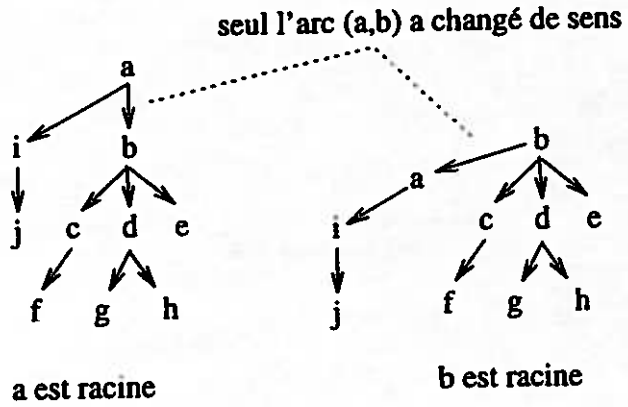


FIG. 3.5 - *Deux orientations possibles pour le même arbre.*

savoir si un sommet g de X_G appartient aux ensembles $S^x(r)$ pour tous les voisins x de r avec une complexité dans le pire des cas en $O(|\Gamma(r) \cup \{r\}|^{1,5}|F(g)|)$. Donc les calculs des ensembles $S^x(r)$ pour tous les voisins x de r se feront en $O(|\Gamma(r) \cup \{r\}|^{1,5} m_G)$ qui est équivalent à $O(|\Gamma(r) \cup \{r\}|^{1,5} n_G)$. Comme cette opération peut être répétée avec tous les sommets de P comme racine, la complexité de l'algorithme pour Iso-ssarbre sans racines est donc en $O(n_P^{1,5} n_G)$.

Cette complexité est remarquable car elle est semblable à celle de la recherche d'un couplage maximum dans un graphe biparti ayant des ensembles de sommets de taille n_X et n_Y. D'autre part on peut remarquer que toute amélioration de la complexité pour le calcul d'un couplage maximum dans un graphe biparti entrainera une réduction de la complexité de cet algorithme.

3.3 Résolution de l'isomorphisme de sous-graphe dans le cas général

On peut considérer que le problème a été traité selon deux approches différentes :

1. Une approche basée sur la recherche de cliques maximales.
2. Une approche centrée sur l'emploi de méthodes de filtrage encore appelée relaxation, combinées à des algorithmes d'énumération de l'espace de solution.

Nous ne désirons pas faire un état de l'art, mais plutôt présenter les idées les plus marquantes du domaine. Le lecteur particulièrement intéressé par ce sujet pour

les graphes moléculaires en chimie organique peut consulter les travaux récents de Barnard [Barnard, 1988; Barnard, 1991; Barnard, 1993].

En ce qui concerne la première approche nous présenterons les travaux de Tonnelier [Tonnelier, 1990]. Il utilise la réduction que nous avons présentée au chapitre précédent (cf. paragraphes 2.2.2 et 2.2.3) mais en utilisant le graphe représentatif des arêtes du graphe d'origine.

Pour la seconde approche, nous respecterons l'ordre chronologique en présentant les travaux de Sussenguth [Sussenguth, Jr., 1965] et ceux de Ullmann [Ullmann, 1976]. Nous montrerons l'équivalence entre la procédure de filtrage de l'algorithme de Ullmann et celle de Sussenguth, qui a pourtant été écrite 12 ans plus tôt.

L'article de McGregor [McGregor, 1979] est peut être le plus important car il montre que le problème que l'on tente de résoudre peut se mettre très facilement sous la forme d'un problème de satisfaction de contraintes, ce qui va permettre d'utiliser les techniques très performantes qui sont disponibles dans ce domaine. La présentation de cet article et des problèmes de satisfaction de contraintes sera le thème du chapitre suivant.

3.3.1 Résolution par recherche de cliques maximales

Tonnelier propose dans sa thèse [Tonnelier, 1990] une méthode de résolution qui utilise la réduction de Iso-ssgrpartiel à Clique. L'originalité de son approche est l'utilisation de graphes aux arêtes[2].

Définition 12 *Le **graphe aux arêtes ou graphe représentatif des arêtes d'un** graphe $G = (X, E)$ est un graphe noté $L(G)$, dont les sommets représentent les arêtes de G, deux de ces sommets étant joints si les deux arêtes qu'ils représentent ont une extrémité commune dans G.*

La figure 3.6 donne un exemple de graphes aux arêtes.

Comme une arête d'un graphe correspond à un sommet de son graphe aux arêtes, on peut envisager de transformer le problème Iso-ssgrpartiel(H, G) en Iso-ssgr$(L(H), L(G))$. Mais le passage d'un graphe au graphe représentatif de ses arêtes n'est pas bijectif. De ce fait un isomorphisme de sous-graphe pour le graphe

2. Le graphe aux arêtes est une notion classique de la théorie des graphes.

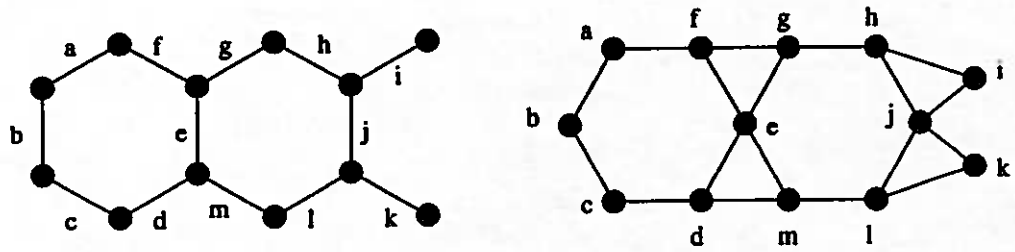


FIG. 3.6 - *Un graphe et le graphe représentatif de ses arêtes.*

aux arêtes ne correspond pas toujours à un isomorphisme de sous-graphe partiel pour les graphes initiaux. Nicholson [Nicholson *et al.*, 1987] a exhibé le théorème suivant qui va permettre de lever cette ambiguïté.

Théorème 1 *Soient H et G deux graphes et m une injection préservant fortement l'adjacence de $L(H)$ dans $L(G)$. Si m ne contient pas d'échange triode-triangle, alors on peut déduire de m un morphisme injectif m' de H dans G.*

La figure 3.7 donne un exemple de triode et de triangle.

Définition 13 *On dit qu'une injection qui préserve fortement l'adjacence de $L(H)$ dans $L(G)$ contient un* ***échange triode-triangle****, s'il existe un sous-ensemble de 3 arêtes $\{x, y, z\}$ dans H qui forme un triode (resp. triangle) tel que l'ensemble des arêtes $\{m(x), m(y), m(z)\}$ forme un triangle (resp. triode) dans G.*

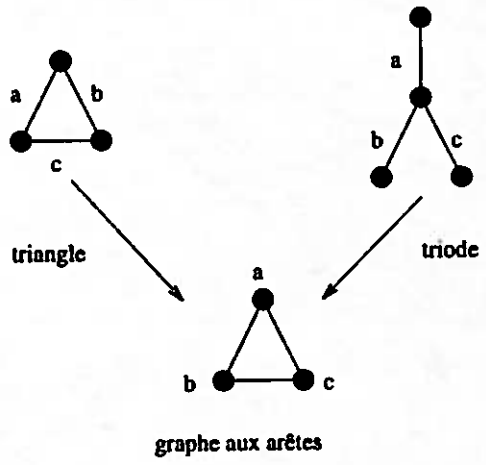


FIG. 3.7 - *Un triangle, un triode et leur graphe aux arêtes.*

Comme le travail de Tonnelier s'est fait dans le cadre de la chimie organique et que les graphes moléculaires possèdent très rarement des triangles, il a négligé ce problème.

Après avoir construit les graphes aux arêtes $L(H)$ et $L(G)$, Tonnelier construit le graphe produit d'adjacence forte puis recherche une clique de la taille du nombre de sommets de $L(H)$ dans ce graphe produit.

Tonnelier voit deux avantages dans son approche :

1. «L'information contenue dans chaque sommet du graphe aux arêtes est plus riche que celle contenue dans le graphe initial. Il y aura donc moins d'appariements locaux possibles entre deux graphes aux arêtes qu'entre les deux graphes initiaux. Le nombre de combinaisons à envisager est moins important ce qui permet de réduire la complexité du problème.»

2. «Si un graphe possède des sommets et des arêtes étiquetés, le graphe représentatif de ses arêtes n'a pas d'arêtes étiquetées ce qui simplifie l'appariement.»

Il nous semble difficile de pouvoir conclure que la combinatoire du problème est réduite en utilisant le graphe aux arêtes. Car d'une part la diminution du nombre d'appariements locaux possibles n'est pas garantie et d'autre part l'utilisation des graphes aux arêtes s'accompagne d'une augmentation du coût de manipulation des objets du graphe. Aussi le coût global semble ne pas être changé. Par ailleurs certaines propriétés comme l'absence de cycles, ne sont pas stables pour le passage aux graphes représentatifs des arêtes. C'est pourquoi nous n'approuvons pas ce premier argument. D'autant plus que si l'on observait réellement une réduction en utilisant les graphes aux arêtes on ne voit pas pourquoi cette idée ne pourrait pas être poussée plus loin en prenant les graphes aux arêtes des graphes aux arêtes et ainsi de suite.

Le second argument est plus intéressant, mais Tonnelier montre que l'étiquetage des sommets et des arêtes pose certains problèmes lorsque l'on veut conserver et utiliser cette information avec le graphe aux arêtes.

Nous avons utilisé cette méthode dans les premières versions du système RESYN [Vismara *et al.*, 1992]. De cette expérimentation nous pouvons conclure qu'il est nécessaire d'utiliser un algorithme extrêmement performant de recherche de cliques maximales, comme celui de Balas et Yu [Balas and Yu, 1986] ou celui de Bron et

Kerbosh [Bron and Kerbosh, 1973]. L'algorithme proposé par Tonnelier pour résoudre Clique est inutilisable en pratique. Mais malheureusement, même avec des algorithmes plus performants, cette approche ne nous a pas convaincu. En effet pour un problème moyen soumis au système, il fallait parfois attendre une demie heure pour obtenir une réponse alors que les temps sont de l'ordre de la seconde avec les méthodes qui vont suivre. Enfin il faut signaler que Tonnelier a depuis abandonné cette méthode.

3.3.2 Résolution par satisfaction de contraintes

Comme nous l'avons déjà dit, résoudre Iso-ssgrpartiel(H, G) c'est établir ou montrer que l'on ne peut pas établir, une correspondance entre les sommets de H et ceux de G qui préserve l'adjacence.

Pour calculer si une telle fonction existe, pour chaque sommet h de H est déterminée la liste des sommets de G avec qui h peut être a priori apparié, c'est-à-dire ceux qui remplissent certaines conditions comme par exemple posséder un degré supérieur au égal à $deg(h)$. Puis ces listes sont réduites dans le but d'obtenir un seul élément par liste de façon à obtenir une injection qui préserve l'adjacence. Cette réduction se fait en employant une technique de filtrage, puis un algorithme de backtrack. Ce dernier peut éventuellement utiliser lui aussi le filtrage. Le chapitre suivant étudie en détail ce mécanisme et ses diverses variantes.

L'algorithme 2 donne le principe des méthodes de résolution qui vont être présentées. Toute liste d'appariements sera appelée LAP.

La procédure PRÉTRAITEMENT vérifie des conditions nécessaires simples, comme la supériorité des degrés, de connaissances liées au domaine, comme par exemple le nombre de cycles auquels appartiennent les sommets pour les graphes moléculaires, et du filtrage qui est employé par la suite. On retrouve souvent le terme prétraitement sous la dénomination «screen» ou «prescreen» dans la littérature.

Algorithme de Sussenguth

Sussenguth [Sussenguth, Jr., 1965] utilise comme filtrage deux propriétés, la connectivité et le partitionnement.

La connectivité exprime le fait que si h peut être apparié avec un ensemble de sommets de X_G (c'est-à-dire sa LAP) alors les voisins de h ne peuvent être appariés

Algorithme 2 Algorithme de résolution du problème de l'isomorphisme de sous-graphe à l'aide de la propagation de contraintes

ISO-SSGRPARTIEL(i H, G) : **booléen**
  $\forall h \in X_H : LAP(h) \leftarrow X_G$
  solution $\leftarrow$ **faux**; *continue* $\leftarrow$ **vrai**
  PRÉTRAITEMENT(LAP, *continue*, *solution*)
  si *continue et* $\neg$*solution* **alors** BACKTRACK(LAP, *continue*, *solution*)
  retourner *solution*

BACKTRACK(**io** LAP, *continue*, *solution*)
  si $\forall h \in X_H |LAP(h)| = 1$ **alors** *solution* $\leftarrow$ **vrai**
  sinon
    choisir h tel que $|LAP(h)| > 1$
    pour *chaque* $g \in LAP(h)$ **tant que** $\neg$*solution* **faire**
      $LAP' \leftarrow LAP$
      $LAP'(h) \leftarrow \{g\}$
      FILTRAGE(LAP', *continue*, *solution*) /* continue est en écriture */
      si *continue* **alors** BACKTRACK(LAP', *continue*, *solution*)

qu'avec les sommets appartenant à l'union des ensembles de voisins de chaque sommet de sa LAP. D'où la fonction de réduction suivante qui assure la cohérence des contraintes d'adjacence :

$$\forall v_h \in \Gamma_H(h) : LAP(v_h) \leftarrow LAP(v_h) \cap (\cup_{g \in LAP(h)} \Gamma_G(g))$$

S'il existe un morphisme injectif de H dans G, alors deux sommets de H ne seront pas appariés avec le même sommet de G. Sussenguth généralise cette idée pour n sommets afin d'obtenir la fonction suivante, nommée partionnement :

si $\exists A \subset X_H$ tel que $\forall a, a' \in A : LAP(a) = LAP(a')$ et $|A| = |LAP(A)|$,
avec $LAP(A) = LAP(a \in A)$ alors :

$$\forall h \in X_H - A : LAP(h) \leftarrow LAP(h) \cap LAP(A)$$

La mise en oeuvre de cette idée est relativement facile puisque toutes les LAP d'un ensemble A sont égales.

Le filtrage proposé par Sussenguth enchaîne la propriété de connectivité avec celle de partionnement. Le filtrage s'arrête dès qu'une LAP devient vide ou bien lorsque plus aucune LAP n'est modifiée.

D'un point de vue pratique les résultats sont assez bons pour la plupart des graphes moléculaires que l'on trouve en chimie organique. Cet article est particulièrement intéressant puisqu'il emploie deux types de traitement. Dans la suite de ce chapitre mais aussi dans cette partie de la thèse, nous verrons des méthodes qui améliorent ces idées, mais leurs principes ne sont pas fondamentalement différents.

Algorithme de Ullmann

L'algorithme de Ullmann [Ullmann, 1976] est certainement le plus connu et le plus utilisé pour résoudre Iso-ssgrpartiel. Il est célèbre pour sa procédure de filtrage (encore appelée relaxation). Nous allons montrer que pour des graphes non orientés et non étiquetés, ce filtrage parvient au même résultat que celui proposé par Sussenguth. Cette observation est originale car personne n'avait encore établi cette égalité.

En plus d'un partionnement très simple qui consiste à supprimer l'unique valeur d'une LAP de toutes les autres LAP afin de garantir que deux sommets de H ne soient pas appariés avec le même sommet de G, Ullmann remarque que si un sommet h est apparié avec un sommet g alors les voisins de h doivent aussi pouvoir être apparié avec les voisins de g. Dans notre formalisme nous obtenons la condition nécessaire suivante :

$$\forall v_h \in \Gamma_H(h) : g \in LAP(h) \Rightarrow \exists v_g \in \Gamma_G(g) \text{ tel que } v_g \in LAP(v_h)$$

La nouvelle proposition est :

Proposition 6 *Les deux propriétés suivantes sont équivalentes.*

(1) $\forall v_h \in \Gamma_H(h) : LAP(v_h) \subseteq (\cup_{g \in LAP(h)} \Gamma_G(g))$

(2) $\forall v_h \in \Gamma_H(h) : g \in LAP(h) \Rightarrow \exists v_g \in \Gamma_G(g)$ *tel que* $v_g \in LAP(v_h)$.

preuve :

(1) $\Leftrightarrow \forall v_h \in \Gamma_H(h) : g \in LAP(v_h) \Rightarrow g \in (\cup_{g' \in LAP(h)} \Gamma_G(g'))$.
(1) $\Leftrightarrow \forall v_h \in \Gamma_H(h) : g \in LAP(v_h) \Rightarrow \exists g''$ tel que $g \in \Gamma_G(g'')$ et $g'' \in LAP(h)$.
or $g \in \Gamma_G(g'') \Leftrightarrow g'' \in \Gamma_G(g)$ puisque le graphe est non orienté. D'où
(1) $\Leftrightarrow \forall v_h \in \Gamma_H(h) : g \in LAP(v_h) \Rightarrow \exists g''$ tel que $g'' \in \Gamma_G(g)$ et $g'' \in LAP(h)$.
Donc (1) $\Leftrightarrow$ (2).

La procédure de Ullmann est plus facile à implémenter que celle de Sussenguth car elle n'utilise pas les structures ensemblistes.

3.4 Conclusion

En nous inspirant des travaux de [Matula, 1978] et [Moon Jung Chung, 1987] nous avons détaillé un algorithme résolvant Iso-ssarbre en temps polynômial. Nous avons ensuite présenté deux approches possibles pour résoudre Iso-ssgrpartiel dans le cas général. La première est basée sur la recherche de cliques maximales et ne nous a pas convaincu. La seconde a été proposée indépendamment par Sussenguth [Sussenguth, Jr., 1965] et Ullmann [Ullmann, 1976]. Elle utilise les techniques dites «de propagation de contraintes» et semble beaucoup plus prometteuse. C'est pourquoi nous allons dans la suite de cette thèse détailler ces techniques.

Chapitre 4

Les réseaux de contraintes

McGregor [McGregor, 1979] a formalisé le problème de l'isomorphisme de sous-graphe par un réseau de contraintes. Puis il a montré l'efficacité des méthodes de ce domaine pour résoudre les problèmes d'isomorphisme. Comme ces méthodes sont nombreuses nous avons jugé nécessaire de leur consacrer un chapitre.

Un réseau de contraintes est défini par la donnée d'un ensemble de variables, chacune associée à un domaine de valeurs, et par la donnée d'un ensemble de contraintes qui mettent en relation ces variables et définissent l'ensemble des combinaisons de valeurs qui satisfont la contrainte. Rechercher une instanciation des variables qui satisfait toutes les contraintes du réseau est le problème principal que l'on se pose habituellement.

De nombreux problèmes se codent naturellement en un réseau de contraintes et se résolvent efficacement grâce aux techniques développées depuis vingt ans notamment par Mackworth et Freuder [Mackworth and Freuder, 1985], Haralick et Elliot [Haralick and Elliot, 1980] et Dechter et Pearl [Dechter and Pearl, 1988]. Le champ des applications qui utilisent les réseaux de contraintes (abrégé CN pour Constraint Networks) va de l'analyse de scènes en vision à l'analyse grammaticale en langage naturel en passant par la conception, l'ordonnancement, le raisonnement temporel et la planification d'expériences génétiques. En biologie et en chimie, les applications portant sur l'étude de grosses molécules (protéines, ARN ...) ou les applications d'aide à la conception de plans de synthèse de peptides [Janssen *et al.*, 1990] sont de grandes consommatrices de CN.

Résoudre un CN, c'est-à-dire savoir s'il a une solution, est un problème NP-Complet. Aussi un grand nombre de méthodes ont été développées pour essayer de minimiser le coût de la résolution d'un CN. Comme nous le verrons par la suite le réseau de contraintes que l'on construit pour résoudre les problèmes d'isomorphisme est complet, c'est-à-dire qu'il existe une contrainte entre toutes les paires de variables. Nous limiterons donc notre étude à ce type de réseau. Néanmoins, comme certains des résultats que nous avons obtenus ne sont pas liés à cette propriété, nous présenterons les réseaux de contraintes dans leur forme la plus générale.

Le but de ce chapitre n'est pas de faire un état de l'art complet (pour cela on peut se reporter à [Jégou, 1991; Tsang, 1993]) mais de donner une idée des différents types d'approches possibles pour résoudre un CN et de situer dans leur contexte les travaux présentés dans les chapitres suivants. Après avoir donné les définitions de base des CN et montré un exemple de codage du problème de l'isomorphisme de sous-graphe partiel par un réseau de contraintes, nous présenterons l'algorithme de recherche systématique, encore appelé recherche par «essai et erreur» et plus connu sous le nom de backtrack chronologique. Cet algorithme est inefficace car il parcourt presque la totalité de l'espace de recherche, c'est pourquoi les travaux sur les CN se sont orientés vers l'élaboration de techniques qui tentent de réduire l'espace de recherche parcouru. Plusieurs approches sont possibles, certaines détectent ***au cours de leur exécution*** des zones de l'espace de recherche qui ne contiennent pas de solution et évitent de les parcourir, tandis que d'autres réduisent la taille de l'espace de recherche ***avant*** de lancer n'importe quelle procédure. Parmi les premières on trouve les algorithmes de recherche, comme le Forward-Checking (FC) ou le Really-Full-Lookahead. Les secondes sont appelées techniques de filtrage ou prétraitement, la méthode la plus connue étant la consistance d'arc. Il existe aussi des méthodes qui essaient de déterminer les cause d'un échec afin d'économiser plus tard des essais de valeurs ou bien de sélectionner un meilleur point de retour. Nous présenterons ces différentes approches et nous montrerons aussi comment l'ordre d'instanciation des variables et des valeurs peut jouer un grand rôle sur l'efficacité des recherches.

4.1 Définitions

Nous ne considérerons dans la suite que les réseaux de contraintes possédant des domaines finis.

Définition 14 *Un **réseau de contraintes fini** $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ est défini comme:*

- *un ensemble de n **variables** $X = \{x_1, ..., x_n\}$,*
- *un ensemble de **domaines** finis $\mathcal{D} = \{D_1, ..., D_n\}$ où D_i est l'ensemble des **valeurs** possibles pour la variable x_i,*
- *un ensemble de **contraintes** entre les variables $\mathcal{C} = \{C_1, ..., C_e\}$. Une contrainte C_i est définie sur un sous-ensemble de variables $X_{C_i} = \{x_{i_1}, ..., x_{i_j}\}$ de X par un sous-ensemble du produit cartésien $D_{i_1} \times ... \times D_{i_j}$, qui spécifie quelles valeurs des variables de $\{x_{i_1}, ..., x_{i_j}\}$ sont compatibles entre elles. L'arité de C_i est j.*

Notations 1

- $n = |X|$
- $d = \max_{i \in 1..n}(|D_i|)$
- $e = |\mathcal{C}|$
- *$C_{/X'}$, avec $X' \subseteq X$, désigne la restriction aux variables de X' de la contrainte $C \in \mathcal{C}$. Si C est définie sur X_C, alors $C_{/X'}$ est la contrainte définie sur $X' \cap X_C$[1], par l'ensemble correspondant à la restriction aux variables de $X' \cap X_C$ des n-uplets de C. $C_{/X'}$ représente la restriction aux variables de X' de toutes les contraintes du réseau.*
- *(i, a) désigne la valeur a de la variable x_i.*

Définition 15 *Un réseau de contraintes est **binaire** si toutes ses contraintes sont d'arité 2. Dans ce cas, une contrainte entre deux variables x_i et x_j est notée C_{ij}. L'ensemble des contraintes contraignant x_i est noté C_{i*}. L'ensemble des variables x_j telles que C_{ij} existe est noté $\Gamma(x_i)$.*

Un réseau de contraintes qui ne contient pas que des contraintes binaires est appelé réseau de contraintes ***général***.

Définition 16 *On appelle **hypergraphe associé** à un réseau de contraintes $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$, l'hypergraphe $H_{\mathcal{R}}$ obtenu en représentant chaque variable du réseau par*

1. Si $X' \cap X_C = \emptyset$ alors $C_{/X'}$ est la contrainte universelle.

un sommet et chaque contrainte C_i définie sur un ensemble $\{x_{i_1}, ..., x_{i_j}\}$ par une hyperarête $\{x_{i_1}, ..., x_{i_j}\}$.

Dans le cas d'un réseau de contraintes binaire on parlera de ***graphe associé*** au réseau ou de ***graphe des contraintes***. Le terme de graphe est aussi souvent employé par abus de langage à la place d'hypergraphe.

Définition 17 *La recherche de solution dans un réseau de contraintes est appelé* ***problème de satisfaction de contraintes*** *et noté* ***CSP***.

4.2 Représentation de Iso-ssgr et de Iso-ssgrpartiel par un réseau de contraintes

4.2.1 Iso-ssgrpartiel

Considérons $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes et supposons que l'on veuille savoir si H est isomorphe à un sous-graphe partiel de G. McGregor [McGregor, 1979] propose de représenter ce problème par le réseau de contraintes binaires suivant[2] :
$\mathcal{R} = (X_H, \mathcal{D}, \mathcal{C})$ où :

- X_H est l'ensemble des sommets de H.
- Le domaine de chaque variable h est constitué par l'ensemble des sommets de G dont le degré est supérieur ou égal à celui de h.
- Il existe une contrainte entre toute paire de variables, définie de la manière suivante : $C_{ij}(a, b)$ est vrai si l'une des deux conditions est vérifiée :
  (1) $\{i, j\} \in E_H$ et $\{a, b\} \in E_G$.
  (2) $\{i, j\} \notin E_H$ et $a \neq b$.

La première condition de validité d'une contrainte exprime que si deux sommets sont reliés dans le graphe H, alors ils ne peuvent s'apparier qu'avec deux sommets

2. Nous verrons par la suite une représentation différente qui exprime mieux que les variables doivent prendre des valeurs différentes.

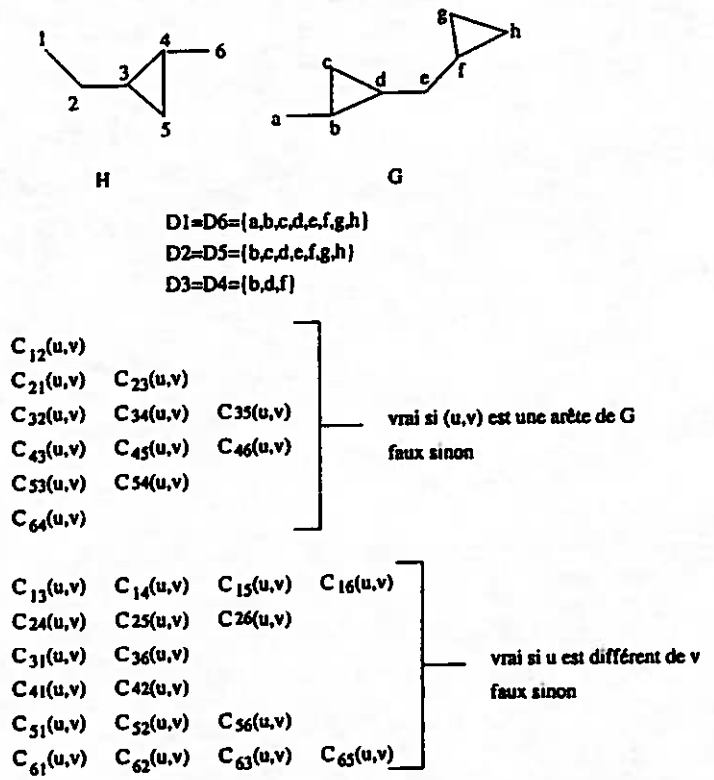


FIG. 4.1 - *Exemple de codage de* lso-ssgrpartiel *par un réseau de contraintes.*

reliés dans le graphe G. La seconde condition correspond au fait que deux sommets de H ne peuvent pas être appariés avec le même sommet de G. Le graphe des contraintes est donc un graphe complet.

Une contrainte C_{ij} est ***une contrainte d'adjacence*** si $\{i,j\} \in E_H$ sinon c'est ***une contrainte de différence.***

La figure 4.1 donne les domaines obtenus après codage du problème de l'isomorphisme de sous-graphe partiel par un réseau de contraintes. Le graphe des contraintes étant complet il n'est pas représenté.

À chaque solution du CSP correspond un plongement de H dans G et récirpoquement.

4.2.2 lso-ssgr

Ce problème se différencie du précédent par l'introduction d'une condition supplémentaire: deux sommets non reliés dans le graphe H peuvent être appariés uni-

quement avec des sommets non reliés dans G. Le réseau de contraintes modélisant ce problème se déduit de celui utilisé pour Iso-ssgrpartiel en modifiant la définition d'une contrainte de la façon suivante :
$C_{ij}(a,b)$ est vrai si l'une des deux conditions est vérifiée :

(1) $\{i,j\} \in E_H$ et $\{a,b\} \in E_G$.

(2) $\{i,j\} \notin E_H$ et $\{a,b\} \notin E_G$ et $a \neq b$.

4.3 Recherche de solutions

Définition 18 ([Dechter, 1992]) *Soit un sous-ensemble de variables $X' = \{x_{i_1}, ..., x_{i_k}\}$ de X, une **instanciation partielle** sur X', notée $I_{X'}$, associe à chaque variable de X' une valeur de son domaine.*

Une instanciation partielle est donc un ensemble de couples (variable, valeur), autrement dit d'affectations, ne contenant jamais deux fois la même variable.

Définition 19 ([Dechter, 1992]) *Une instanciation partielle sur X' est **localement consistante** si et seulement si elle satisfait toutes les contraintes de $C_{/X'}$.*

Définition 20 *Soit $I_{X'}$ une instanciation partielle :*

- *une affectation (y,a) d'une valeur a à une variable y n'appartenant pas à X' est appelée **affectation augmentante** de $I_{X'}$.*
- *l'instanciation partielle obtenue en ajoutant une et une seule affectation augmentante de $I_{X'}$ à $I_{X'}$ est appelée **augmentation** de $I_{X'}$.*
- *un **ensemble prolongeant** de $I_{X'}$ est un ensemble A d'affectations augmentantes de $I_{X'}$ tel que $I_{X'} \cup A$ soit une instanciation partielle.*
- *l'instanciation partielle obtenue en ajoutant un ensemble prolongeant de $I_{X'}$ à $I_{X'}$ est appelée **prolongement** de $I_{X'}$.*
- *l'instanciation partielle obtenue en supprimant une ou plusieurs affectations de $I_{X'}$ est appelée **réduction** de $I_{X'}$.*

Définition 21 ([Dechter, 1992]) *Une **solution** d'un CSP est une instanciation sur X localement consistante c'est-à-dire une affectation de valeurs à toutes les variables qui satisfait toutes les contraintes.*

Définition 22 ([Dechter, 1992]) *Une instanciation partielle est* ***globalement consistante*** *si et seulement si elle admet un prolongement qui est une solution.*

Définition 23 ***L'espace de recherche*** *d'un CSP correspond à l'ensemble de toutes les instanciations possibles sur X. Sa taille est donc égale à $\prod_{1 \leq i \leq n} |D_i|$, soit exactement d^n pour un réseau de n variables ayant toutes un domaine de taille d.*

Dans l'exemple donné par la figure 4.1, l'instanciation $I = \{(1, a), (2, b), (3, d)\}$ est localement consistante, $(4, f)$ est une affectation augmentante de I, $I \cup (4, f)$ est une augmentation de I mais elle n'est pas localement consistante (3 et 4 sont reliés dans H alors que d et f ne le sont pas dans G). $\{(5, c), (6, h)\}$ est un ensemble prolongeant de I et l'instanciation $\{(1, a), (2, b), (3, d), (5, c), (6, h)\}$ est un prolongement de I. Le lecteur attentif remarquera que cette dernière instanciation, qui est localement consistante, n'admet pas d'augmentation conduisant à une solution. Par conséquent, elle n'est pas globalement consistante.

Le principe général des procédures de recherche de solutions consiste à instancier les variables du problème successivement dans l'espoir d'obtenir une instanciation sur X consistante. De manière informelle, cela signifie que les algorithmes cherchent à augmenter petit à petit les instanciations partielles pour obtenir une instanciation consistante de toutes les variables. Ces procédures peuvent prendre diverses formes, mais toutes ont comme base l'algorithme ***backtrack*** [Golomg and Baumert, 1965], BT en abrégé.

Nous commencerons par présenter cet algorithme, puis nous étudierons ses inconvénients. Cela nous amènera à proposer un nouvel algorithme plus général remédiant à certains défauts de BT. Ensuite nous montrerons que l'application d'une procédure de recherche de solutions est équivalente à un parcours en profondeur dans un arbre. Cela nous permettra de valider facilement diverses variantes de l'algorithme général. Finalement nous donnerons quelques outils pour caractériser et comparer le comportement des algorithmes de recherche.

4.3.1 L'algorithme backtrack

La procédure backtrack, encore appelé backtrack chronologique, consiste à instancier les variables du problème successivement dans un ordre prédéfini : $x_1, ..., x_n$. A

chaque fois qu'une valeur est affectée à une variable, la consistance de l'instanciation partielle obtenue est testée. Si le test est positif, une autre variable est instanciée et ce jusqu'à ce que toutes les variables le soient, sinon une autre valeur pour la variable est choisie. S'il n'existe plus de valeurs pour la variable (toutes les valeurs ont été essayées), un retour-arrière (ou backtrack) est réalisé sur l'instanciation de la variable qui précède la variable courante dans l'ordre afin d'essayer une nouvelle valeur. S'il n'en existe pas on procède de nouveau à un retour-arrière.

Les structures de données employées sont les suivantes :

- k représente la taille de l'ensemble sur lequel est défini l'instanciation partielle courante. Elle est appelée niveau.

- I est un tableau déterminant l'instanciation partielle courante : $I[j]$, pour $j \leq k$, contient un couple (i, a) signifiant que la valeur a a été affectée à la variable x_i. L'instanciation partielle obtenue après k affectations est représentée par l'ensemble des $I[j]$ pour j variant de 1 à k. Aussi une augmentation de l'instanciation partielle courante, se traduira par l'incrémentation de k.

- La procédure SUPPRIMER(a, D_i, k) supprime la valeur a du domaine de D_i et mémorise le fait que cette valeur a été supprimée de son domaine au niveau k. La procédure RESTAURERCSP(k) replace dans leur domaine les valeurs qui ont été supprimées par la procédure précédente au niveau k.

Si les variables sont choisies selon l'ordre fixé a priori : $x_1, ..., x_n$, alors la $k^{ème}$ variable instanciée sera toujours x_k et l'ensemble sur lequel est défini une instanciation partielle de k éléments sera toujours $\{x_1, ..., x_k\}$.

La fonction LOCALEMENTCONSISTANTE (cf. algorithme 3) vérifie la consistance locale d'une augmentation par rapport à l'instanciation partielle dont elle est issue. La procédure BACKTRACK (cf. algorithme 4) recherche toutes les solutions du CSP.

Principe d'empilement des appels :
L'instanciation partielle courante est successivement augmentée tant que les augmentations obtenues sont localement consistantes. Ce processus se termine dès qu'une solution est atteinte ou bien quand une instanciation partielle non localement consistante est rencontrée.

Algorithme 3 Test de la consistance d'un choix avec les choix passés

```
LOCALEMENTCONSISTANTE(i (I_k, k)) : booléen
    consistant ← vrai
    (i, a) ← I[k]
    l ← 1
    tant que l < k et consistant faire
    |   (j, b) ← I[l]
    |   si ∃C_ji ∈ C alors
    |   |   si ¬C_ji(b, a) alors consistant ← faux
    |   |   sinon l ← l + 1
    retourner consistant
```

Algorithme 4 Algorithme backtrack

```
BACKTRACK(i I, k)
    faire
    |   a ← CHOISIRVALEUR(D_k)
    |   I[k] ← (x_k, a)
    |   si LOCALEMENTCONSISTANT(I, k) alors
    |   |   si k = n alors AFFICHERSOLUTION(I)
    |   |   sinon BACKTRACK(I, k + 1)
    |   SUPPRIMER(a, D_k, k)
    tant que D_k ≠ ∅
    RESTAURERCSP(k)
```

Principe de dépilement des appels :

L'instanciation partielle courante est successivement réduite tant que la dernière variable instanciée par l'instanciation partielle courante ne possède pas d'autre valeur dans son domaine que celle qui lui avait été affectée. Si la procédure précédente s'arrête à un niveau $k-1 > 0$, alors la valeur affectée à x_k dans $I[k]$ est supprimée du domaine de x_k. Avant de réduire une instanciation partielle $I_{X' \cup \{x_k\}}$ définie sur k variables, le CSP est rétabli dans l'état où il se trouvait avant d'augmenter pour la première fois $I_{X'}$. Ce processus de réduction se termine au niveau k, soit parce que k

vaut 0 et dans ce cas toutes les solutions ont été trouvées, soit parce qu'une variable possède d'autres valeurs dans son domaine que celles qui lui avait été affectées.

4.3.2 Arbres de recherche

L'application d'une procédure de recherche peut toujours se représenter par un arbre muni d'une racine. Un noeud de cet arbre correspond à l'affectation d'une valeur à une variable. Si cette affectation a lieu au niveau k, alors le noeud apparaîtra à la profondeur k dans l'arbre. A tout noeud n_k de niveau k est donc associé une instanciation partielle I_k définie sur un ensemble de k variables. Tous les noeuds associés aux augmentations de I_k seront les fils de n_k dans l'arbre. La racine de cet arbre est un noeud particulier qui ne correspond à aucune instanciation mais qui a pour fils tous les noeuds de la profondeur 1.

Le nombre maximum de feuilles d'un arbre associé à une procédure de recherche est égal au nombre d'instanciations possibles.

L'affectation de valeurs à des variables s'apparente à une ***descente*** dans l'arbre de recherche, tandis que la remise en cause d'une affectation est vue comme une ***remontée***. Une procédure de recherche correspond donc à un parcours en profondeur d'un arbre de recherche.

Les arbres de recherche que nous présenterons par la suite seront souvent simplifiés. En effet, nous ne représenterons qu'une seule arête entre un noeud et ses fils, car dans la configuration actuelle, tous les fils d'un noeud correspondent à des affectations impliquant la même variable. Ainsi pour Iso-ssgrpartiel présenté en figure 4.1, on obtient l'arbre donné en figure 4.2.

4.3.3 Inconvénients du backtrack

Cette méthode peut faire parcourir de manière exhaustive la totalité de l'espace de recherche et comme la taille de cet espace est bornée par d^n, son emploi peut s'avérer très coûteux. En effet, sa complexité est en $O(ed^n)$, si l'on considère que les tests de consistance peuvent être faits en temps constant. Cependant cette complexité n'est pas étonnante à cause de la nature exponentielle du problème. Néanmoins, on peut remarquer que cette procédure parcourt aveuglément l'espace de recherche sans tenir

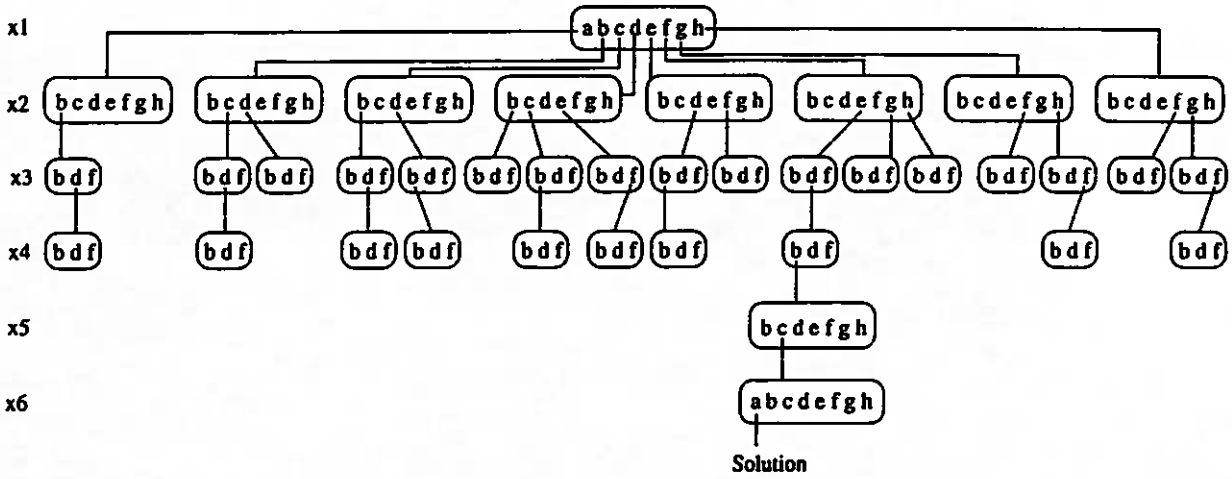


FIG. 4.2 - *Arbre de recherche engendré par l'algorithme backtrack*

compte d'informations faciles à extraire du CSP et qui économiseraient du temps de calcul. On peut considérer que cette procédure présente quatre types de défauts :

1. Elle n'applique aucun prétraitement pour essayer de détecter des inconsistances locales afin d'éliminer d'emblée des valeurs qui ne peuvent pas appartenir à une solution.

2. Elle choisit aveuglément les valeurs qu'elle affecte aux variables sans vérifier qu'il existe des valeurs compatibles avec l'instanciation partielle courante pour toutes les autres variables non instanciées. Elle parcourt donc des zones de l'espace dont on pourrait déterminer qu'il ne peut pas s'y trouver de solutions.

3. Elle choisit les variables à instancier selon un ordre arbitraire qui ne tient pas compte des particularités du problème, comme par exemple l'adjacence entre les sommets du petit graphe pour Iso-ssgrpartiel.

4. Elle engendre souvent des prolongements qui sont identiques à des prolongements précédemment étudiés et qui ne conduisent à aucune solution. Car le retour-arrière chronologique ne tient pas compte des interactions qui existent entre les variables. Elle peut donc refaire plusieurs fois les mêmes erreurs.

À partir de ces quatre types d'inconvénients, on peut définir quatre approches d'améliorations :

• Réduction explicite de l'espace de recherche avant l'exécution de la procédure de recherche

Cette réduction peut être réalisée soit en employant un filtrage, soit en décomposant le CSP d'origine. Dans les deux cas, le but recherché est le même: modifier le CSP d'origine pour obtenir un CSP comportant le même ensemble de solutions et plus facile à résoudre.

Une procédure de filtrage ne résout pas complètement le CSP mais élimine une fois pour toutes des inconsistances qui autrement auraient pu être découvertes plusieurs fois par la procédure de recherche de solutions. Les procédures de filtrage s'appuient sur les ***consistances locales*** définies sur le CSP, qui expriment des conditions locales suffisantes pour qu'une instanciation ne soit pas globalement consistante. Par la suite (cf 4.4), nous limiterons notre étude à la plus ancienne des techniques de consistance locale: la ***consistance d'arc***, car elle est simple, efficace et son coût est peu élevé, contrairement aux autres consistances qui ne présentent, à l'heure actuelle, qu'un intérêt théorique, notamment à cause de leur complexité en espace.

Les méthodes de décomposition ont pour but de découper le problème initial en sous-problèmes plus faciles à résoudre. Leur raisonnement est généralement basé sur les propriétés structurelles du réseau de contraintes. Comme le réseau modélisant Isossgrpartiel est complet, ces techniques ne peuvent pas s'appliquer pour ce problème. Par conséquent nous ne les étudierons pas.

• Réduction explicite de l'espace de recherche pendant l'exécution de la procédure de recherche

Cette technique est prospective («*look-ahead*») et consiste à examiner les valeurs des variables non encore instanciées et à supprimer par filtrage celles dont on est assuré que leur emploi, compte tenu de l'instanciation partielle courante, conduira à un échec. Elle va donc anticiper des échecs en introduisant un traitement supplémentaire après chaque instanciation. Ce traitement élimine des inconsistances locales liées à l'instanciation partielle courante. Divers filtrages peuvent être employés. La seule condition qui leur est imposée est de garantir que toute augmentation d'une instanciation partielle localement consistante soit aussi localement consistante. Nous

étudierons (cf 4.6) deux algorithmes de ce type: le Forward-Checking et le Really Full Lookhead.

• Guidage de la procédure de recherche

L'ordre d'instanciation selon lequel les variables sont choisies est très important. L'ordonnancement peut être statique ou bien dynamique. Dans le premier cas, il est réalisé une fois pour toutes avant la recherche, tandis que dans le second, la prochaine variable à instancier est déterminée en fonction d'un critère dynamique. Ces ordres s'appuient sur la structure du réseau de contraintes. Par exemple, ils peuvent tenir compte du nombre de valeurs dans les domaines ou bien du nombre de voisins dans le graphe des contraintes. Nous étudierons (cf 4.7) ces deux types d'ordres.

• Détermination des raisons d'un échec

Cette technique d'amélioration est rétrospective (*«look-back»*). Les méthodes employées sont souvent qualifiées d'intelligentes, car elles essaient de déterminer les causes d'un échec. Elles ont pour but soit d'économiser plus tard des essais de valeurs (backchecking [Haralick and Elliot, 1980], backmarking [Gaschnig, 1977]), soit de sélectionner un meilleur point de retour (backjumping [Gaschnig, 1978], graph-based backjumping [Dechter, 1990], conflict-directed backjumping [Prosser, 1993a; Prosser, 1993b]). Nous ne les étudierons pas, car nous verrons que le maintien de la consistance d'arc pendant une procédure de recherche est très efficace pour résoudre lso-ssgrpartiel et aussi parce qu'il n'existe pas encore de méthode combinant cette approche avec les méthodes de détermination d'un échec.

Ces différentes approches peuvent être utilisées simultanément pour résoudre un problème.

Nous sommes maintenant en mesure de proposer un nouvel algorithme de recherche qui va permettre d'introduire différentes méthodes d'améliorations de l'algorithme classique.

4.3.4 Algorithme générique de recherche

L'algorithme proposé est très général. Les structures de données utilisées sont les même que pour le backtrack. Il est important de noter que cette fois-ci l'ordre d'instanciation des variables n'est pas nécessairement fixé a priori. Aussi la variable instanciée au niveau k ne sera pas obligatoirement x_k. La fonction CHOISIRVARIABLE a été introduite pour choisir une variable parmi les variables non instanciées.

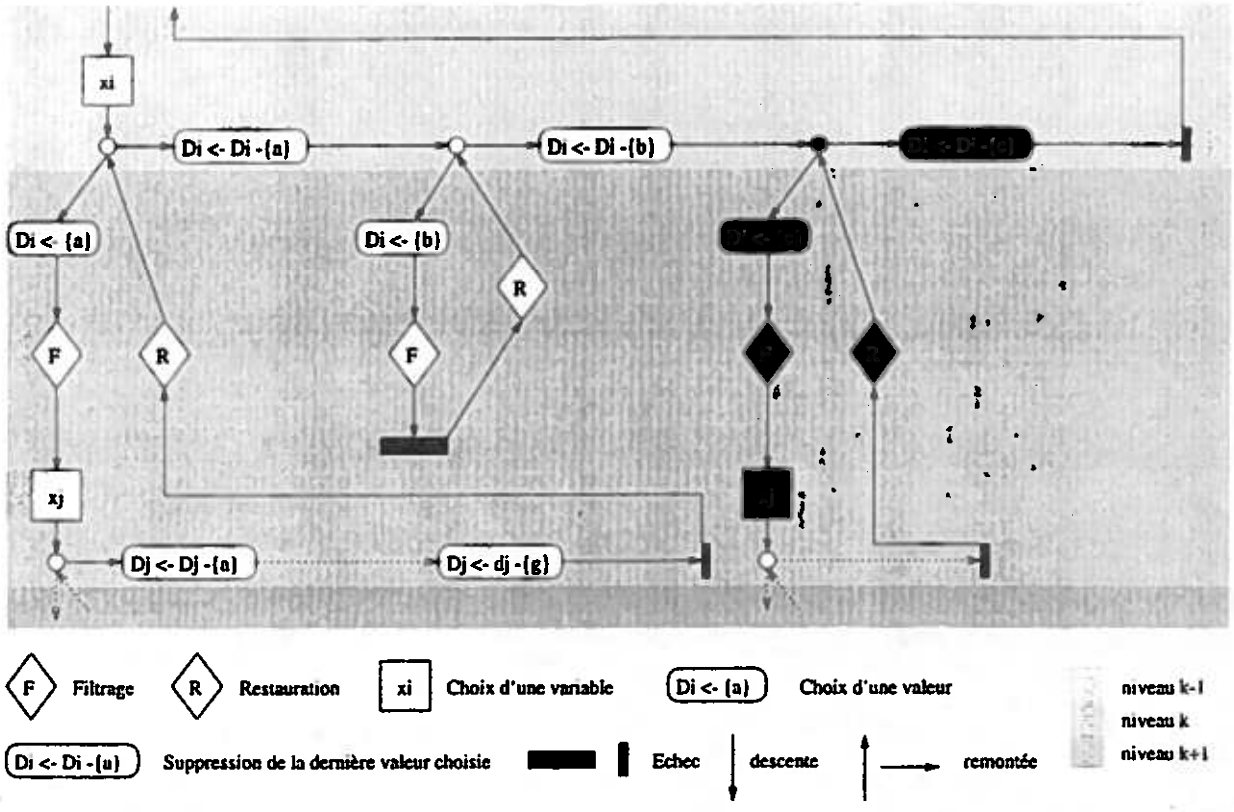


FIG. 4.3 - *Fonctionnement de l'algorithme générique pour un niveau de l'arbre de recherche.*

La figure 4.3 donne une partie de l'arbre binaire équivalent à l'arbre engendré par l'algorithme générique de recherche et montre l'enchaînement des choix de valeurs, des tests de consistance (appelé filtrage sur le dessin) et des restaurations.

Algorithme 5 Algorithme générique de recherche de toutes les solutions.

```
RECHERCHERTOUTESSOLUTIONS()
    consistant ← PRÉTRAITEMENT()
    si consistant alors RECHERCHERSOLUTIONS(I, 1)
RECHERCHERSOLUTIONS(i I, k)
    x_i ← CHOISIRVARIABLE()
    faire
        a ← CHOISIRVALEUR(D_i)
        I[k] ← (i, a)
        si k = n alors AFFICHERSOLUTION(I)
        sinon
            si ÉLIMINERINCONSISTANCESLOCALES(I, k) alors
                RECHERCHERSOLUTIONS(I, k + 1)
        SUPPRIMER(a, D_i, k)
    tant que D_i ≠ ∅
    RESTAURERCSP(k)
```

4.3.5 Caractérisation des algorithmes de recherche

Nous avons comme objectif d'essayer de caractériser les algorithmes de recherche ***sans tenir compte de l'ordre d'instanciations des variables.*** Pour mettre en oeuvre cette approche originale nous introduisons la notion suivante :

Définition 24 *Une instanciation partielle $I_{X'}$ est dite* ***atteinte*** *par une procédure de recherche si et seulement si à un moment donné de l'algorithme on a :* $I_{X'} = \cup_{i=1}^{|X'|} I[i]$.

Pour l'algorithme backtrack, nous savons qu'une instanciation partielle atteinte est augmentée si et seulement si elle est localement consistante. On en déduit la propriété suivante :

Propriété 3 *Une instanciation partielle est atteinte si et seulement si toutes ses réductions sont localement consistantes.*

4.3.6 Outils de comparaison des algorithmes de recherche

Définition 25 *Lors de l'exécution d'une procédure de recherche :*

• *Une augmentation est dite* ***libre*** *si la dernière variable affectée possède plusieurs valeurs dans son domaine; elle est dite* ***forcée*** *si le domaine de cette variable est un singleton.*

• *Il y a* ***échec*** *quand une instanciation partielle atteinte par une procédure de recherche doit finalement être écartée parce qu'elle n'admet* ***pas*** *d'augmentation.*

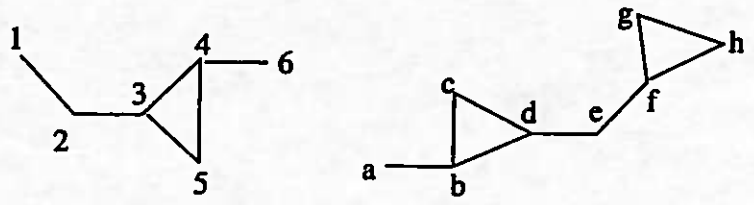
• *Il y a* ***retour-arrière*** *quand une instanciation partielle atteinte par une procédure de recherche doit finalement être écartée parce qu'elle n'admet* ***plus*** *d'augmentation.*

Il est important de bien faire la différence entre un échec et un retour-arrière. Intuitivement, un retour-arrière correspond à l'abandon du niveau courant pour le niveau précédent. L'arbre de recherche donné en figure 4.2 comprend exactement $1+8+17+10+1+1=38$ noeuds donc 38 retours-arrière, $1\times 8+8\times 7+17\times 3+10\times 3+1\times 7+1\times 8=160$ augmentations dont 122 libres et 38 forcées, et 122 échecs.

4.4 La consistance d'arc

La consistance d'arc est la technique de filtrage la plus employée. Elle est habituellement attribuée à Waltz [Waltz, 1975]. Elle exprime une condition de consistance locale qui impose que chaque valeur (i,a) soit compatible avec toute contrainte portant sur x_i. Si (i,a) n'apparaît dans aucun n-uplet d'une contrainte contraignant x_i, alors elle n'appartiendra à aucune solution, on peut donc la supprimer du domaine (càd a de D_i). Si un CSP ne contient pas de telles valeurs, on dira qu'il vérifie la consistance d'arc.

La figure 4.4 reprend l'exemple proposé en figure 4.1. La valeur f de D_3 peut être supprimée car f n'est compatible avec aucune valeur de x_4 sur la contrainte C_{34}. Après le filtrage par consistance d'arc on obtient un réseau équivalent au premier, c'est-à-dire possédant le même ensemble de solutions, mais dont les domaines sont réduits. Ainsi dans notre exemple les domaines après filtrage sont :



H G

D1=D6={a,b,c,d,e,f,g,h}
D2=D5={b,c,d,e,f,g,h}
D3=D4={b,d,f}

FIG. 4.4 - *Exemple de codage de* lso-ssgrpartiel *par un réseau de contraintes*

$$D_3 = D_4 = \{b, d\}$$
$$D_2 = D_5 = \{b, c, d, e\}$$
$$D_6 = \{a, b, c, d, e\}$$
$$D_1 = \{a, b, c, d, e, f\}$$

La définition de la consistance d'arc que nous donnons est celle proposée par [Jégou, 1991] qui est équivalente à celle que l'on trouve dans [Mohr and Masini, 1988a], elle est appelée consistance d'arc généralisée car elle s'applique aux CSP n-aires.

Définition 26 *Etant donné $\mathcal{R} = (X, \mathcal{D}, C)$, un domaine D_i vérifie la* ***consistance d'arc*** *si et seulement si l'on a: $D_i \neq \varnothing$ et $\forall a \in D_i, \forall C_j \in C$ contraignant x_i, $a \in C_{j/\{x_i\}}$.*

Le réseau de contraintes obtenu par suppression des valeurs non viables d'un réseau de contraintes est la ***fermeture*** par consistance d'arc du réseau d'origine. Dans le cas des réseaux de contraintes binaires on peut formuler la consistance d'arc de la manière suivante:

Définition 27 *Un réseau de contraintes binaires $\mathcal{R} = (X, \mathcal{D}, C)$ vérifie la consistance d'arc si et seulement si:*
$\forall x_i \in X : D_i \neq \varnothing$ et $\forall a \in D_i, \forall x_j$ avec $C_{ij} \in C, \exists b \in D_j$ tel que $C_{ij}(a, b)$.

Cette définition nous permet d'introduire logiquement les notions de support et de valeur viable [Mohr and Henderson, 1986; Bessière, 1992] qui vont permettre de présenter simplement le principe des algorithmes de fermeture par consistance d'arc.

Définition 28 *Soit C_{ij} une contrainte, si $C_{ij}(a,b)$ est vrai, on dira indifféremment que :*

- (j,b) ***supporte*** (i,a);
- (j,b) *est un* ***support*** *de* (i,a).

Définition 29 *Une valeur a d'une variable x_i est* ***viable*** *si pour toute variable x_j telle que C_{ij} existe, a possède un support dans D_j.*

La consistance d'arc peut alors se définir de la manière suivante :

Définition 30 *Un réseau de contraintes binaires $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ vérifie la consistance d'arc si et seulement si aucun domaine n'est vide et toutes les valeurs de tous les domaines sont viables.*

Comme ce filtrage supprime des valeurs des domaines des variables, il permet une meilleure efficacité de la procédure de recherche. Ainsi pour notre exemple (cf figure 4.1 on obtient l'arbre donné en figure 4.5.

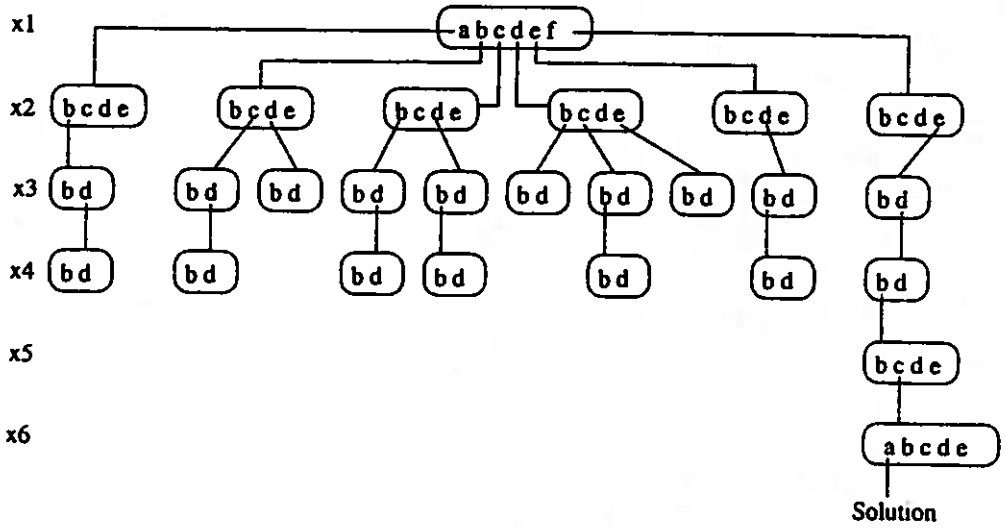


FIG. 4.5 - *Arbre de recherche engendré par l'algorithme backtrack après filtrage par consistance d'arc.*

Afin de simplifier l'exposé nous introduisons la notion d'inconsistance d'arc :

Définition 31 *Un réseau de contrainte $\mathcal{R} = (X, \mathcal{D}, C)$ vérifie* ***l'inconsistance d'arc*** *si et seulement si $\mathcal{R}$ ne contient pas de sous-domaine $\mathcal{D}'$ (càd $\forall D'_i \in \mathcal{D}'$, on a $D'_i \subseteq D_i \in \mathcal{D}$) tel que le réseau $\mathcal{R}' = (X, \mathcal{D}', C)$ vérifie la consistance d'arc.*

Un réseau peut ne vérifier ni la consistance d'arc ni l'inconsistance d'arc. Par contre, tout réseau obtenu après fermeture par consistance d'arc vérifie l'une de ces deux propriétés.

Pourr simplifier notre exposé, nous appelerons dans la suite «algorithme d'AC» un algorithme calculant la fermeture par consistance d'arc d'un réseau.

Equivalence entre le filtrage de Ullman et la consistance d'arc pour Iso-ssgrpartiel

McGregor [McGregor, 1979] affirme que le filtrage de Ullmann (cf 2.3.2) est équivalent à la consistance d'arc pour les contraintes exprimant que deux sommets reliés dans H ne peuvent s'apparier qu'avec deux sommets reliés dans G (condition (1) de la validité d'une contrainte cf 3.2). Nous allons montrer cette équivalence. Le filtrage proposé par Ullmann est :

$$\forall v_h \in \Gamma_H(h) : g \in LAP(h) \Rightarrow \exists v_g \text{ tel que } v_g \in \Gamma_G(g) \text{ et } v_g \in LAP(v_h)$$

$\forall v_h \in \Gamma_H(h)$ peut se mettre sous la forme $\forall (v_h, h) \in E_H$.
De la même manière $v_g \in \Gamma_G(g)$ correspond à $(v_g, g) \in E_G$.
$g \in LAP(h)$ et $v_g \in LAP(v_h)$ sont respectivement équivalents dans le formalisme des CSP à $g \in D_h$ et à $v_g \in D_{v_h}$.
La condition de Ullmann est donc équivalente à :
$\forall (v_h, h) \in E_H : g \in D_h \Rightarrow \exists v_g$ tel que $v_g \in D_{v_h}$ et $(v_g, g) \in E_g$.
Ou encore :
$g \in D_h \Rightarrow \forall (v_h, h) \in E_H, \exists v_g$ tel que $v_g \in D_{v_h}$ et $(v_g, g) \in E_g$.
Or la première condition de satisfaisabilité d'une contrainte $C_{ij}(a, b)$ est $(i, j) \in E_H$ et $(a, b) \in E_G$.
Si C_1 représente l'ensemble dont la définition repose sur l'adjacence, nous pouvons reformuler la condition de Ullmann à l'aide des CSP de la manière suivante : $g \in$

$D_h \Rightarrow \forall C_{v_h h} \in \mathcal{C}_1, \exists v_g \in D_{v_h}$ tel que $C_{v_h h}(v_g, g)$ soit vrai.
On retrouve bien la définition de la consistance d'arc (cf. définition 27).

La recherche de support pour les valeurs est le coeur des algorithmes que nous allons maintenant étudier.

4.4.1 Algorithmes de fermeture par consistance d'arc

Mohr et Masini [Mohr and Masini, 1988a] ont développé GAC-4 un algorithme de fermeture par consistance d'arc pour les réseaux de contraintes générales. Sa complexité est en $O(K)$ où K est la somme des longueurs de tous les n-uplets de toutes les contraintes. Cette complexité empêche l'emploi en pratique de cet algorithme dès que l'arité des contraintes est élevée. C'est pourquoi nous ne considérons dans cette partie que les réseaux de contraintes binaires.

Le premier algorithme d'AC que nous présentons, pour ce type de réseaux et sans avoir d'information sur la sémantique des contraintes, a été proposé par Mackworth [Mackworth, 1977] sous le nom de AC-3, sa complexité en temps est en $O(ed^3)$[Mackworth and Freuder, 1985]. Mohr et Henderson [Mohr and Henderson, 1986] ont donné une nouvelle version de ce filtrage, référencée sous la dénomination AC-4, dont la complexité en temps ($O(ed^2)$) est optimale. Mais cet algorithme présente deux inconvénients : sa complexité en espace ($O(ed^2)$) et sa complexité en moyenne, qui font, comme l'ont montré Wallace et Freuder de manière empirique, que l'algorithme AC-3 est presque toujours meilleur en pratique que AC-4 [Wallace, 1993]. Cette discussion a pris fin avec l'arrivée d'un nouvel algorithme, AC-6, proposé par Bessière et Cordier[3] [Bessière and Cordier, 1993; Bessière, 1994]. En effet, si la complexité d'AC-6 en temps dans le pire des cas est semblable à celle d'AC-4, sa complexité en espace est en $O(ed)$ et sa complexité expérimentale est inférieure à celle d'AC-3.

Mackworth a montré que la consistance d'arc est basée sur la notion de support (cf. définition 28). Aussi longtemps qu'une valeur (i, a) est supportée par une valeur b de chaque variable x_j voisine de x_i dans le graphe des contraintes, (i, a) est viable. Mais s'il existe une variable x_j dont aucune des valeurs ne supporte a, alors a peut être

3. Nous ne nous intéresserons pas ici à AC-5 [Van Hentenryck *et al.*, 1992], qui n'améliore pas l'algorithme AC-4, de manière générale. Cet algorithme permet une adaptation aisée des algorithmes de fermeture par consistance d'arc pour tenir compte des propriétés sémantiques des contraintes.

supprimée de D_i. La suppression de cette valeur peut remettre en cause la viabilité des valeurs des variables voisines de x_i dans le graphe des contraintes. Il est alors nécessaire de leur rechercher de nouveaux supports dans D_i afin de savoir si ces valeurs de x_i demeurent ou non viables. Ce processus est répété dans le cas où de nouvelles valeurs viennent à leur tour à être supprimées. L'algorithme s'arrête soit si un domaine devient vide, soit si l'on ne supprime plus de valeur, et dans ce cas, le réseau de contraintes vérifie la consistance d'arc. Ce principe est propre à tous les algorithmes d'AC. Ceux-ci diffèrent essentiellement par le nombre de valeurs qui sont réétudiées après la suppression d'une valeur.

Pour mettre en évidence leurs similitudes et leurs différences, nous avons choisi de présenter les algorithmes avec le même modèle, c'est-à-dire avec :

- une phase d'initialisation, qui recherche les valeurs non viables,
- une phase de propagation, qui étudie les conséquences de l'élimination de ces valeurs en déterminant notamment les valeurs dont la viabilité doit être remise en cause,
- une fonction qui détermine si une valeur d'une variable possède un support sur une contrainte donnée et qui éventuellement ajoute cette valeur dans la liste des valeurs supportées du support.

D'une manière générale, lorsque la consistance d'arc est utilisée comme prétraitement, l'algorithme 6 réalise ce filtrage et détermine si le réseau vérifie l'inconsistance d'arc.

Nous considérerons qu'une valeur d'une variable x_i possède un drapeau qui permet de savoir en temps constant si elle appartient ou non au domaine courant D_i. Il est aussi possible de supprimer une valeur d'un domaine en temps constant grâce à la fonction SUPPRIMER(a, D_i). *listeAttente* est une liste contenant les éléments à propager.

AC-3

Lorsqu'une valeur (i, a) n'a pas de support sur une contrainte C_{ij}, AC-3 ***réétudie systématiquement la viabilité des valeurs de toutes les variables sur les***

Algorithme 6 La consistance d'arc comme prétraitement

```
PRÉTRAITEMENT() : booléen
    listeAttente ← ∅
    si INITIALISATION(listeAttente) alors retourner PROPAGATION(listeAttente)
    retourner faux
```

contraintes C_{*i} (sauf C_{ji}). Ces dernières sont mises dans *listeAttente* (cf algorithme 7). Supposons que la valeur (i, a) ait été supprimée parce qu'elle ne possède pas (ou plus) de support sur la contrainte C_{ij} et considérons une autre contrainte C_{ik}. AC-3 recherche un support dans D_i pour toutes les valeurs de D_k. Si une autre valeur de x_i disparaît, alors AC-3 recherche de nouveau un support dans D_i pour toutes les valeurs de D_k.

AC-4

Le principe d'AC-4 est d'éviter de faire plusieurs fois les mêmes calculs, lors de l'étude des conséquences d'une suppression. Reprenons l'exemple précédent. Il ne sert à rien de rechercher si toutes les valeurs de D_k possèdent un support sur x_i, car les valeurs de D_k viables avant la suppression de (i, a) et qui ne sont pas consistantes avec a restent toujours viables après la disparition de a. Si elles sont viables alors elles possèdent un support sur C_{ik} qui est différent de a, donc l'élimination de a ne remet pas directement en cause leur viabilité.

AC-4 ***réalise, une fois pour toutes, tous les tests de consistance et mémorise ceux qui sont positifs.*** Pour ce faire, AC-4 utilise deux structures de données :

- $S_{ia} = \{(j, b) | b \in D_j \text{ et } C_{ji}(b, a)\}$. C'est l'ensemble S_{ia} des valeurs que a supporte dans D_j sur la contrainte C_{ij}.
- un compteur $cpt[(i, j), a]$ qui indique le nombre de supports de la valeur a de x_i sur la contrainte C_{ij}.

Ainsi lorsqu'une valeur est supprimée, on connaît l'ensemble des valeurs qu'elle supportait sur chaque contrainte et on peut savoir si ces valeurs possèdent un autre support en utilisant les compteurs. La fonction EXISTESUPPORT devient donc triviale.

Algorithme 7 L'algorithme AC-3

```
INITIALISATION(o listeAttente) : booléen
    listeAttente ← C
    retourner vrai
PROPAGATION(io listeAttente) : booléen
    tant que listeAttente ≠ ∅ faire
        prendre C_ij dans listeAttente et la supprimer
        changement ← faux
        pour chaque a ∈ D_i faire
            si ¬ EXISTESUPPORT(x_i, a, C_ij) alors
                changement ← vrai
                SUPPRIMER(a, D_i)
                si D_i = ∅ alors retourner faux
        si changement alors
            /* on ajoute les contraintes C_*i sauf C_ji dans listeAttente */
            listeAttente ← listeAttente ∪ {C_ki ∈ C | k ≠ i, k ≠ j}
    retourner vrai
EXISTESUPPORT(i x_i, a, C_ij) : booléen
    trouvé ← faux
    pour chaque b ∈ D_j tant que ¬trouvé faire
        si C_ij(a, b) alors trouvé ← vrai
    retourner trouvé
```

Par exemple, soient $S_{ia} = \{(j,a),(j,c),(k,b)\}$, l'ensemble des valeurs que a supporte pour les contraintes C_{ij} et C_{ik} et $cpt[(j,i),a] = 3, cpt[(j,i),c] = 1, cpt[(k,i),b] = 2$, les valeurs des compteurs des valeurs que a supporte. Supposons que a soit supprimée de D_i, alors la viabilité des valeurs a, c de D_j et b de D_k est réexaminée. Chacune de ces valeurs ayant un support en moins, il faut décrémenter leur compteur. Si le compteur d'une valeur devient nul alors on doit supprimer cette valeur de son domaine. Dans notre exemple on doit éliminer c de D_j. Cette étape correspond à la phase de propagation.

La phase d'initialisation recherche systématiquement tous les supports pour toutes

les valeurs et détecte les valeurs qui ne sont pas viables.

Algorithme 8 AC-4 : fonctions d'initialisation et d'existence d'un support

```
INITIALISATION(o listeAttente) : booléen
    listeAttente ← ∅
    pour chaque x_i ∈ X faire
        pour chaque a ∈ D_i faire S_ia ← ∅
    pour chaque x_i ∈ X faire
        pour chaque a ∈ D_i faire
            pour chaque x_j ∈ Γ(i) tant que a ∈ D_i faire
                total ← 0
                pour chaque b ∈ D_j faire
                    si C_ij(a, b) alors
                        total ← total + 1
                        AJOUTER((i, a), S_jb)
                si total = 0 alors
                    SUPPRIMER(a, D_i)
                    AJOUTER((i, a), listeAttente)
                sinon cpt[(i, j), a] ← total
        si D_i = ∅ alors retourner faux
    retourner vrai
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
    /* b est inutile dans AC-4 */
    cpt[(i, j), a] ← cpt[(i, j), a] − 1
    si cpt[(i, j), a] = 0 alors retourner faux
    sinon retourner vrai
```

Les algorithmes que nous proposons diffèrent légèrement de ceux que l'on trouve dans [Mohr and Henderson, 1986], afin de pouvoir utiliser le code de la fonction de propagation pour AC-6. Mais cela entraîne aussi l'apparition d'un paramètre supplémentaire pour la fonction EXISTESUPPORT qui est inutile pour AC-4.

La complexité dans le pire des cas est réduite à $O(ed^2)$. Mais cette amélioration dans le pire des cas ne reflète pas la complexité réelle. Ainsi, le coût du calcul systéma-

Algorithme 9 Fonction de propagation commune à AC-4 et AC-6

```
PROPAGATION(io listeAttente) : booléen
  tant que listeAttente ≠ ∅ faire
    prendre (j, b) dans listeAttente et le supprimer
    pour chaque (i, a) ∈ S_jb faire
      si a ∈ D_i alors
        si ¬ EXISTESUPPORT(x_i, a, C_ij, b) alors
          SUPPRIMER(a, D_i)
          si D_i = ∅ alors retourner faux
          AJOUTER((i, a), listeAttente)
  retourner vrai
```

tique de tous les supports lors de l'étape d'initialisation s'avère élevé en pratique, c'est d'ailleurs pourquoi AC-3 est souvent meilleur. Pour éviter cela, Bessière et Cordier ont développé l'algorithme AC-6.

AC-6

L'idée essentielle d'AC-6 repose sur le fait qu'il n'est pas nécessaire de connaître pour une valeur (i, a) toutes les valeurs de x_j supportant a sur une contrainte C_{ij}. La connaissance d'un seul support sur x_j est suffisante. En effet la définition nous impose ***l'existence d'un support et non pas la connaissance de tous les supports.***

Afin de pouvoir étudier les conséquences de la disparition d'une valeur, l'algorithme tient à jour pour chaque valeur (i, a) la liste S_{ia} des valeurs dont le support dans D_i est a. Nous rappelons qu'une valeur (i, a) n'a besoin que d'un seul support dans le domaine de chaque variable voisine de x_i dans le graphe des contraintes. S_{ia} ne contient donc pas toutes les valeurs supportées, à l'inverse de AC-4. Comme on peut être amené à rechercher plusieurs fois un nouveau support pour une valeur, il est important de trouver efficacement, c'est-à-dire sans refaire des calculs déjà faits, si la valeur possède un autre support. Pour y parvenir, l'algorithme utilise un ordre total sur les valeurs de chaque domaine et, lorsqu'il recherche un nouveau support, il lui suffit d'étudier les valeurs qui suivent dans l'ordre le support supprimé, d'où le

code de la fonction EXISTESUPPORT.

Plus formellement l'algorithme utilise les structures de données suivantes :

- Chaque domaine initial D_i est muni d'un ordre total et de deux fonctions en temps constant (en plus de SUPPRIMER) permettant d'accéder à certaines informations du domaine :
  - *dernier*(D_i) retourne la valeur de D_i dont l'indice est le plus élevé si $D_i \neq \varnothing$, sinon elle retourne -1.
  - si $a \in D_i$ et $a \neq$ *dernier*(D_i), *suivant*(a) retourne la valeur de D_i de plus petit indice supérieur à celui de a.
- $S_{jb} = \{(i,a) | (j,b)$ est la valeur de D_j de plus petit indice qui supporte $(i,a)\}$.
- *listeAttente* est identique à celle d'AC-4.

Nous pouvons étudier maintenant plus précisément le comportement de l'algorithme. Les conséquences de la suppression d'une valeur (j,b) se traduisent par la recherche de nouveaux supports dans D_j pour toutes les valeurs (i,a) de S_{jb}. L'ordre sur les domaines permet de rechercher un nouveau support pour (i,a) uniquement dans l'ensemble des valeurs du domaine D_j dont les indices sont strictement supérieurs à celui de b. On sait, en effet, que toute valeur de D_j dont l'indice est inférieur à b ne peut pas être un support. Lorsque l'on trouve un nouveau support c pour (i,a) sur C_{ij}, on supprime (i,a) de la liste des valeurs supportées de son précédent support, pour garantir qu'une valeur n'appartient jamais à plus d'une seule liste de valeurs supportées sur chaque contrainte, et on ajoute (i,a) dans la liste S_{jc}, puisque (j,c) est le support de plus petit indice dans D_j. Si l'on ne trouve aucun support, alors on supprime a du domaine de D_i et on place (i,a) dans *listeAttente*.

La preuve de cet algorithme se trouve dans [Bessière and Cordier, 1993] ou dans [Bessière, 1994].

La complexité en espace dépend de la taille des différentes listes des valeurs supportées. Considérons une arête (x_i, x_j) du graphe des contraintes. Une valeur a de D_i appartient au plus à une seule liste S_{jb}. La taille totale de tous les S_{jb} pour cette contrainte est en $O(d)$. Puisqu'il y a e arêtes dans le graphe des contraintes la complexité en espace est en $O(ed)$.

Algorithme 10 AC-6 : fonctions d'initialisation et de recherche d'un support. La fonction de propagation est la même que celle d'AC-4.

```
INITIALISATION(o listeAttente) : booléen
    listeAttente ← ∅
    pour chaque x_i ∈ X faire
        pour chaque a ∈ D_i faire S_ia ← ∅
    pour chaque x_i ∈ X faire
        pour chaque a ∈ D_i faire
            pour chaque x_j ∈ Γ(x_i) faire
                si ¬ EXISTESUPPORT(x_i, a, C_ij, nil) alors
                    SUPPRIMER(a, D_i)
                    AJOUTER((i, a), listeAttente)
        si D_i = ∅ alors retourner faux
    retourner vrai
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
    trouvé ← faux
    c ← SUIVANT(b, D_j)
    tant que c ≤ DERNIER(D_j) et ¬trouvé faire
        si C_ij(a, c) alors
            si b ≠ nil alors SUPPRIMER((i, a), S_jb)
            AJOUTER((i, a), S_jc)
            trouvé ← vrai
        sinon c ← SUIVANT(c, D_j)
    retourner trouvé
```

Pour une arête (x_i, x_j) du graphe des contraintes et pour une valeur a de D_i, la somme des calculs faits à l'intérieur de la fonction EXISTESUPPORT est en $O(d)$. La complexité en temps de la procédure INITIALISATION qui fait ed appels à cette fonction est donc en $O(ed^2)$. Pour calculer la complexité en temps de la procédure PROPAGATION, il faut s'intéresser au nombre de fois où l'on peut être amené à calculer la valeur de $C_{ij}(a, b)$. Cette valeur est calculée par la fonction EXISTESUPPORT, qui ne peut faire ce calcul qu'une seule fois, grâce à l'ordre total utilisé. La complexité

en temps de la procédure PROPAGATION est semblable au nombre de vérifications possibles de $C_{ij}(a,b)$, soit $O(ed^2)$. De cela Bessière et Cordier ont déduit que : «Si l'on ne dispose pas d'informations sur la sémantique des contraintes, alors l'algorithme AC-6 a une complexité optimale en temps en $(O(ed^2))$ et en espace en $(O(ed))$.»

Nous verrons dans le chapitre suivant que l'optimalité énoncée n'est pas exacte.

L'avantage d'AC-6 sur AC-4 ne se résume pas seulement à une complexité en espace réduite, ni à une diminution du nombre de tests de consistance réalisés. L'un des avantages majeurs est la réduction des calculs pendant la phase de propagation (moins de valeurs supportées à réexaminer). C'est-à-dire que non seulement la phase d'initialisation est plus coûteuse dans AC-4 mais aussi la phase de propagation. En effet la propriété suivante est un invariant de l'algorithme AC-6 :

Propriété 4 $(i,a) \in S_{jc} \Leftrightarrow C_{ij}(a,c)$ *et* (j,c) *est le plus petit support de* (i,a) *dans* D_j *selon l'ordre de* D_j.

Cette propriété signifie que si (i,a) est supportée par (j,c) alors il n'existe pas de valeur b dans D_j qui supporte aussi (i,a) et qui soit inférieure à c. La différence entre les deux algorithmes s'exprime de la façon suivante :

Propriété 5 *Soit* C_{ij} *une contrainte. Si une valeur* c *de* x_j *est supprimée de* D_j, *alors l'étude de la viabilité des éléments* (i,a) *compatible avec* (j,c) *et possédant un support inférieur à* c *selon l'ordre de* D_j *au moment où* c *disparaît ne sera pas faite par AC-6 alors qu'elle le sera par AC-4.*

Le gain réalisé par AC-6 par rapport à AC-4 peut-être de l'ordre de $O(d^2)$ pour une contrainte, comme le montre la figure 4.6.

Dans cet exemple, nous pouvons voir que la liste S_{jd} des valeurs supportées par (j,d) contient les valeurs a,b,c,d de D_i pour AC-4 et aucune de ces valeurs pour AC-6. Si (j,d) est supprimée, alors AC-4 étudiera la viabilité de toutes les valeurs de x_i tandis que AC-6 ne fera aucune de ces études car toutes ces valeurs ont a comme support sur C_{ij}, qui est inférieure à d dans l'ordre de D_j.

4.4.2 Prise en compte de la sémantique des contraintes

Seuls quelques travaux prenant en compte la sémantique des contraintes ont été menés à bien : Mohr et Masini ont décrit une amélioration de la complexité en temps

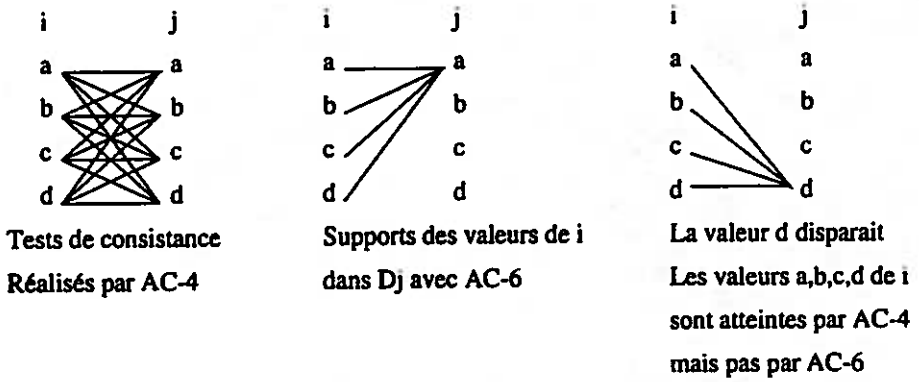


FIG. 4.6 - *Comparaison des algorithmes AC-4 et AC-6.*

de l'algorithme de fermeture par consistance d'arc pour des contraintes binaires particulières introduites par un problème de reconnaissance des formes [Mohr and Masini, 1988b]; van Hentenryck, Deville et Teng [Van Hentenryck, 1989] ont étudié les contraintes binaires fonctionnelles et les contraintes binaires monotones. Puisque dans Iso-ssgrpartiel, les seules contraintes particulières que l'on rencontre sont des contraintes de différence, nous avons décidé de limiter notre présentation à ce type de contraintes.

Définition 32 *Une contrainte C_{ij} est dite contrainte de **différence** si et seulement si elle est de la forme :*

(1) $\forall a \in D_i \cap D_j$ *on a* $\neg C_{ij}(a,a)$.

(2) $\forall a \in D_i$ *et* $\forall b \in D_j, b \neq a$, *on a* $C_{ij}(a,b)$.

Mohr et Masini ont remarqué que pour ce type de contrainte l'absence de support ne peut se produire que dans le cas où un domaine se réduit à un singleton. En effet, considérons par exemple une contrainte de différence C_{ij} et supposons que le domaine D_j contienne au moins deux valeurs a et b, alors toute valeur de D_i est supportée par a ou bien par b. Il ne sert donc à rien d'étudier pour ce type de contraintes les conséquences de la suppression de valeur d'un domaine tant que ce domaine n'est pas réduit à une seule valeur. Dans ce dernier cas la propagation devient évidente puisqu'il suffit de supprimer cette valeur de l'autre domaine. Pour ces contraintes la complexité en temps du calcul de la fermeture par consistance d'arc est donc réduite d'un facteur d.

4.5 Consistance d'arc et Iso-ssgrpartiel

Dans cette section nous allons montrer comment l'on peut tirer parti des spécificités de Iso-ssgrpartiel pour accélérer les algorithmes d'AC.

Reprenons la définition des contraintes pour Iso-ssgrpartiel(H, G). Il existe deux types de contraintes :

1. les contraintes de différence : $\forall \{i, j\} \notin E_H : C_{ij}(a, b)$ est vrai ssi $a \neq b$.

2. les contraintes d'adjacence : $\forall \{i, j\} \in E_H : C_{ij}(a, b)$ est vrai si $\{a, b\} \in E_G$;

Nous avons vu en section 4.4.2 qu'il était possible d'utiliser un traitement particulier pour les contraintes de différence. Nous allons donc nous intéresser au second type de contraintes : les contraintes d'adjacence. La matrice d'adjacence de G, que l'on peut considérer comme une donnée du problème, nous permet immédiatement de savoir si un couple satisfait une contrainte d'adjacence. De plus l'ensemble des supports d'une valeur (i, a) sur une contrainte d'adjacence est égale à $\Gamma_G(a) \cap D_j$, où $\Gamma_G(a)$ désigne le voisinage du sommet a dans le graphe G. Pour que AC-4 et AC-6 tiennent compte de cette particularité il suffit de modifier la fonction EXISTESUPPORT d'AC-6 (algorithme 11) et la phase d'initialisation d'AC-4 (algorithme 12).

Algorithme 11 Fonction de recherche d'un support d'AC-6 pour Iso-ssgrpartiel.

```
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
  /* PREMIER(E) retourne le premier élément de E si E n'est pas vide et nil sinon */
  /* Γ_G(a) est le voisinage d'un sommet a dans le graphe G */
  c ← PREMIER(Γ_G(a) ∩ D_j)
  si c = nil alors retourner faux
  sinon
    SUPPRIMER((i, a), S_jb)
    AJOUTER((i, a), S_jc)
  retourner vrai
```

Algorithme 12 Fonction d'initialisation d'AC-4 pour Iso-ssgrpartiel.

```
INITIALISATION(o listeAttente) : booléen
  /* Γ_G(a) est le voisinage d'un sommet a dans le graphe G */
  listeAttente ← ∅
  pour chaque x_i ∈ X faire
    pour chaque a ∈ D_i faire S_ia ← ∅
  pour chaque x_i ∈ X faire
    pour chaque a ∈ D_i faire
      pour chaque x_j ∈ Γ(i) tant que a ∈ D_i faire
        total ← |Γ_G(a) ∩ D_j|
        si total = 0 alors
          SUPPRIMER(a, D_i)
          AJOUTER((i, a), listeAttente)
        sinon cpt[(i, j), a] ← total
      si D_i = ∅ alors retourner faux
    retourner vrai
```

4.6 Algorithmes de recherche

Haralick et Elliot [Haralick and Elliot, 1980] ont présenté des procédures qui fonctionnent avec un filtrage pendant la recherche. Elles se distinguent par le type de filtrage qu'elles utilisent. Nadel [Nadel, 1988] a établit le rapport entre ces procédures et le type de consistance qu'ils utilisent. Il propose quatre algorithmes qu'il note $TS + AC_i$, pour *Tree-Search + Arc-Consistency*, l'indice i correspond au type de consistance effectivement utilisée. Les deux plus connus sont $TS + AC_{1/4}$ ou Forward-Checking et $TS + AC_1$ ou Really Full Lookahead.

Jusqu'à présent, on a considéré que le meilleur algorithme en pratique était le Forward-Checking. Mais les travaux récents de Sabin et Freuder [Sabin and Freuder, 1994a; Sabin and Freuder, 1994b] montrent qu'il faut se méfier de ce genre d'a priori. De plus, l'amélioration des performances des algorithmes d'AC permettent d'envisager une utilisation encore plus efficace de l'algorithme Really Full Lookahead.

Il est évident que plus le filtrage utilisé sera fort plus le nombre de noeuds étudiés par la procédure de recherche sera réduit. Un filtrage appliqué de manière systé-

matique permet d'élaguer l'arbre de recherche. Cependant l'utilisation d'un filtrage entraîne un coût supplémentaire en temps. Une méthode sera donc efficace si le compromis entre le surcoût dû au filtrage et l'élaguage réalisé est bon.

4.6.1 Forward-Checking (FC)

Nous abrégerons dans la suite Forward-Checking par FC.

D'après Nadel, cet algorithme a été proposé pour la première fois par McGregor [McGregor, 1979]. Ce dernier le préconise pour résoudre Iso-ssgrpartiel.

Cet algorithme analyse le voisinage immédiat de la dernière variable instanciée, pour éliminer dans les domaines des variables non encore instanciées toutes les valeurs qui sont directement incompatibles avec l'instanciation courante. La nouvelle fonction ÉLIMINERINCONSISTANCESLOCALES (cf algorithme 13) réalise cette propagation. Une valeur ne peut plus être incompatible avec des réductions de l'instanciation courante, puisque les inconsistances sont éliminées immédiatement après chaque augmentation. Le principe de l'algorithme de recherche de toutes les solutions est inchangé. Mais comme une instanciation peut entraîner la suppression de certaines valeurs, il est nécessaire de mémoriser ces suppressions afin de pouvoir restaurer le réseau dans l'état où il se trouvait avant l'instanciation. C'est la fonction RESTAURERCSP qui est chargée de cette tâche.

Algorithme 13 Forward-Checking

```
ÉLIMINERINCONSISTANCESLOCALES(i I, k) : booléen
    consistant ← vrai
    (x_i, a) ← I[k]
    pour chaque x_j ∈ Γ(x_i) tant que consistant faire
    |   pour chaque b ∈ D_j faire
    |   |   si ¬C_ij(a, b) alors SUPPRIMER(b, D_j, k)
    |   si D_j = ∅ alors consistant ← faux
    retourner consistant
```

L'utilisation de FC sur notre exemple donne l'arbre de la figure 4.7. Cet arbre comprend 22 noeuds, 22 retours-arrière, 16 échecs, 38 instanciations dont 16 libres et

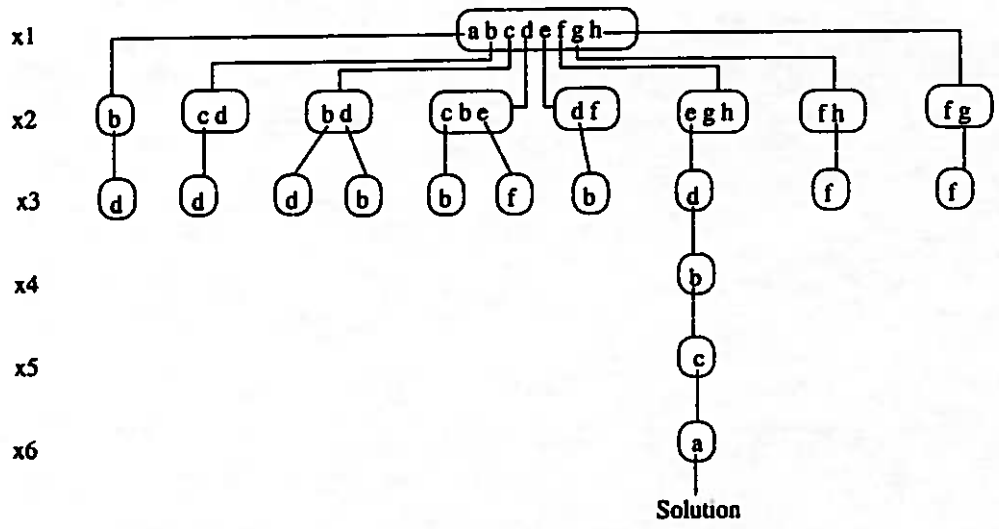


FIG. 4.7 - *Arbre de recherche engendré par l'algorithme Forward-Checking*

22 forcées.

Comparativement à la procédure de recherche systématique (backtrack) cette procédure est beaucoup plus efficace du point de vue du nombre de noeuds engendrés, du nombre d'échecs et du nombre d'instanciations. Cependant, pour comparer ces deux méthodes il faut aussi prendre en compte le nombre de tests de consistance effectués. En général même si l'on fait localement plus de tests, le nombre de noeuds engendrés par FC est nettement inférieur à celui engendré par le backtrack, si bien que globalement FC fait aussi moins de tests de consistance. Le coût du filtrage après chaque instanciation est en $O(deg(x_i)d)$. Cette majoration est forte; plus finement elle s'exprime par: $\sum_{x_j \in \Gamma(x_i)} |D_j|$. Pour une branche de l'arbre de recherche, la complexité sera en $\sum_{x_i \in X} \sum_{x_j \in \Gamma(x_i)} |D_j|$, c'est-à-dire en $O(ed)$.

Comme pour l'algorithme backtrack, nous pouvons caractériser le FC:

Propriété 6 *Une instanciation partielle $I_{X'}$ est atteinte si et seulement si: $\forall I_{X''}$ une réduction de $I_{X'}$, $\forall x_j \in X - X'', \exists b \in D_j$ tel que $I_{X''} \cup \{(x_j, b)\}$ soit localement consistante.*

4.6.2 Really Full Lookahead (RFL)

Damns la suite nous abrégerons Really Full Lookahead par RFL.

Cet algorithme réalise un filtrage par consistance d'arc après chaque instanciation.

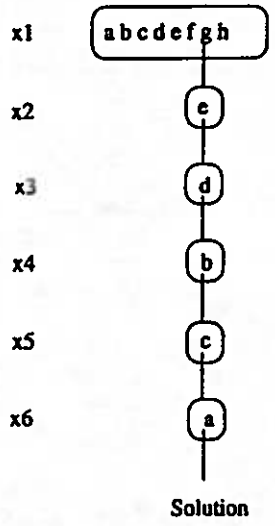


FIG. 4.8 - *Arbre de recherche engendré par l'algorithme Really Full Lookahead.*

Nous verrons dans le chapitre 6 de la thèse que l'on ne peut pas considérer qu'il y a maintien de la consistance d'arc pendant toute la procédure de recherche. Après chaque augmentation, on étudie la consistance du réseau et comme pour FC, toutes les modifications liées à une instanciation et au filtrage qui s'en suivent doivent être mémorisées pour rétablir le réseau dans le même état qu'avant. L'algorithme 14 propose une implémentation possible de la fonction ÉLIMINERINCONSISTANCESLOCALES pour mettre en oeuvre l'algorithme RFL.

Algorithme 14 Really Full Lookahead avec AC-3.

ÉLIMINERINCONSISTANCESLOCALES(i $(a, x_i), I, k$) : **booléen**
  pour *chaque* $b \in D_i$ *tel que* $b \neq a$ **faire** SUPPRIMER(b, D_i, k)
  AJOUTER(C_{*i}, *listeAttente*)
  /* appel de la propagation de la consistance d'arc */
  retourner AC-3(*listeAttente*)

Pour notre exemple, l'arbre de recherche est extrêmement réduit (cf figure 4.8).

On a seulement étudié 6 noeuds, 6 retours-arrière, fait 13 instantiations dont 7 libres et 6 forcés et on a rencontré 7 échecs. Mais cette fois le nombre de tests de consistance risque d'être plus élevé que pour FC, car on filtre tous les domaines des variables non instanciées par la consistance d'arc, dont le coût est en $O(ed^3)$, si l'on

emploie AC-3. Mais on obtient la même complexité pour une branche de l'arbre de recherche.

La caractérisation du cet algorithme est résumée par la propriété suivante :

Propriété 7 *Une instanciation partielle $I_{X'}$ est atteinte si et seulement si $I_{X'}$ est localement consistante et si le sous-réseau $\mathcal{R}' = (X, \mathcal{D}', C)$, où $\mathcal{D}' = \{D_i \in \mathcal{D} | x_i \in X - X'\} \cup \{D_i = \{a\} | (x_i, a) \in I_{X'}\}$ ne vérifie pas l'inconsistance d'arc.*

4.7 Heuristiques sur l'ordre des instanciations

Le nombre de noeuds parcourus par l'algorithme de recherche dépend de l'ordre choisi pour l'instanciation des variables. Étudions, par exemple, l'arbre de recherche de la procédure backtrack pour l'ordre $(x_3, x_4, x_5, x_2, x_6, x_1)$ qui est donné par la figure 4.9.

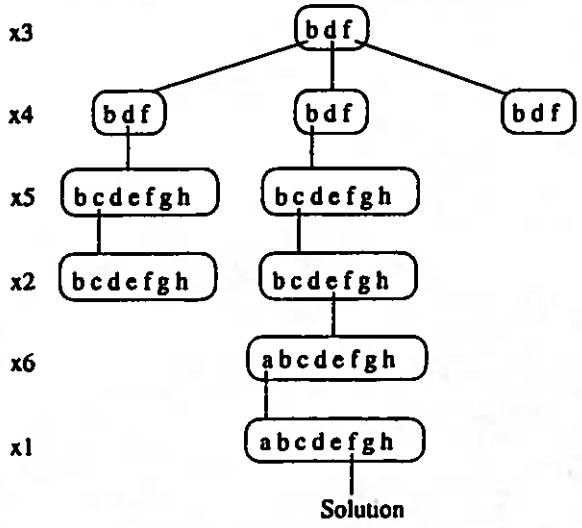


FIG. 4.9 - *Arbre de recherche engendré par l'algorithme backtrack pour l'ordre $(x_3, x_4, x_5, x_2, x_6, x_1)$.*

Cet arbre est très réduit par rapport à celui engendré par l'ordre $(x_1, x_2, x_3, x_4, x_5, x_6)$ Il comprend seulement 10 noeuds et 56 augmentations. L'importance d'un bon choix est donc très clairement exprimée.

Suivant qu'on s'intéresse à la recherche d'une seule ou de toutes les solutions, les heuristiques sur les ordres d'instanciation ne seront pas les mêmes. Si une seule solu-

tion est recherchée, il vaut mieux choisir d'abord les variables les moins contraintes. Par contre, si l'on recherche toutes les solutions, il est préférable de choisir en premier les variables les plus contraintes. Le fait qu'une variable est contrainte peut être vu de plusieurs manière différentes. Cette notion peut correspondre au degré de la variable dans le graphe des contraintes, mais aussi à l'inverse du nombre de n-uplets contenus dans les contraintes.

Les travaux les plus importants dans ce domaine sont dus à Haralick et Elliot [Haralick and Elliot, 1980]. Ils ont notamment établi le ***first fail principle*** qui suggère de commencer par instancier la variable qui a le plus de chance de conduire à une impasse, car le retour-arrière sera moins coûteux que si on instancie cette variable plus tard dans l'arbre de recherche.

Les variables peuvent être ordonnées soit ***une fois pour toute et avant de commencer la recherche***, on parle d'ordre statique sur les variables, soit ***pendant la recherche*** et l'ordonnancement est dit dynamique.

4.7.1 Ordonnancement statique des variables

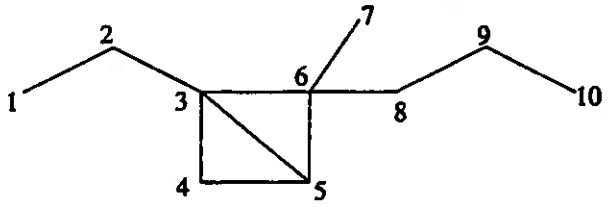
Avec un ordonnancement statique la variable choisie à un niveau k est toujours la même. ***Toutes les augmentations de n'importe quelle instanciation partielle I atteinte au niveau k impliquent toujours la même variable.***

Dechter et Meiri [Dechter and Meiri, 1989] proposent les trois ordonnancement suivants :

- ***la cardinalité maximale***, la première variable est sélectionnée arbitrairement; à chaque étape, la variable choisie est celle qui est connectée avec le plus grand nombre de variables déjà sélectionnées.
- ***le degré maximum***, c'est l'ordre décroissant des degrés des variables dans le graphe des contraintes.
- ***la largeur minimum***, l'ordre est établi de la dernière variable vers la première, en sélectionnant à chaque étape le sommet de degré minimal dans le sous- graphe des contraintes induit par la suppression des variables déjà sélectionnées.

Si l'on considère le réseau de contraintes construit pour Iso-ssgrpartiel, le même résultat est obtenu par les trois ordres précédents car le graphe des contraintes est

complet. Comme l'a remarqué McGregor, il faut dans ce cas définir l'ordre en ne s'intéressant uniquement qu'aux contraintes d'adjacence. Pour certains graphes les ordres ne seront plus les mêmes. Ainsi pour le problème de la figure 4.10 les trois ordres seront différents.



Ordres :

Cardinalité maximale	:	1 2 3 4 5 6 7 8 9 10
Degré maximum	:	1 7 10 2 4 8 9 5 6 3
Largeur minimum	:	1 7 10 2 9 8 6 4 5 3

FIG. 4.10 - *Un réseau de contraintes et différents ordres d'instanciation possibles.*

On peut choisir les variables selon des critères plus ou moins complexes. Bessière propose dans sa thèse l'ordre obtenu à partir du critère croissant suivant :

$$critère(x_i) = |D_i|^2 / \sum poids(C_{ij}), \forall C_{ij} \in \mathcal{C} \text{ contraignant } x_i$$
$$\text{avec } poids(C_{ij}) = |D_i \times D_j| / |C_{ij}|.$$

Pour l'exemple 4.1 on obtient :

$poids(C_{12}) = 56/17; poids(C_{23}) = poids(C_{45}) = 21/6;$
$poids(C_{34}) = 9/2; poids(C_{35}) = 21/8; poids(C_{46}) = 24/9.$
D'où
$critère(x_1) = 8,11; critère(x_2) = 4,77; critère(x_3) = 0,79;$
$critère(x_4) = 0,69; critère(x_5) = 6,03; critère(x_6) = 8,82.$

Ce qui donne l'ordre croissant : $x_4, x_3, x_2, x_5, x_1, x_6$.

Cet ordre donne un arbre de recherche un peu moins bon que le précédent. On remarquera toutefois que cette méthode présente l'avantage de ne pas seulement tenir

compte de la structure du graphe des contraintes, ou bien du nombre d'éléments présents dans les domaines, mais des deux à la fois.

4.7.2 Ordonnancement dynamique des variables

Cette fois-ci l'ordre n'est plus déterminé une fois pour toutes avant la recherche, mais après chaque instanciation. L'heuristique la plus connue a été proposée par Haralick et Elliot [Haralick and Elliot, 1980]. Elle consiste à choisir, après chaque instanciation, la variable qui a le moins de valeurs dans son domaine. Remarquons, cependant, que toutes les augmentations d'une instanciation partielle impliquent toujours la même variable. Ce qui ne signifie par nécessairement que toutes les augmentations des instanciations partielles atteintes au niveau k impliquent la même variable, comme cela était le cas avec un ordre statique.

Pour que cette idée présente de l'intérêt il faut la coupler avec une procédure look-ahead, sinon l'ordre ne sera pas réellement dynamique. Dans l'exemple, nous obtiendrons l'arbre de recherche suivant :

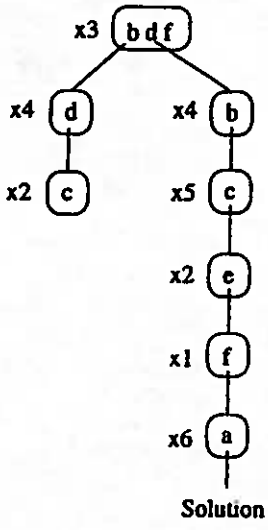


FIG. 4.11 - *Arbre de recherche engendré par l'algorithme Forward-Checking avec ordre dynamique.*

Au niveau 3, la variable choisie n'est pas toujours la même, l'ordre est donc bien dynamique.

4.7.3 Ordonnancement des valeurs

Si l'on cherche une solution, cet ordre a particulièrement de l'importance. En effet, si le CSP admet une solution et que l'on choisit pour chaque variable la valeur qu'elle prend dans la solution, alors on trouvera cette solution sans aucun échec et sans aucun retour-arrière. Il existe le même type d'heuristiques pour le choix des valeurs que pour l'ordonnancement des variables.

4.8 Évaluation des méthodes

Dans les chapitres suivants, nous allons proposer des améliorations de certains algorithmes utilisés pour résoudre les CSP. Ces algorithmes vont permettre d'accélérer la recherche de solutions pour Iso-ssgrpartiel, mais ils ne sont pas particulièrement dédiés à ce problème et peuvent être aussi utilisés dans le cas général. Aussi nous avons eu l'ambition de mettre en oeuvre un processus expérimental nous permettant d'évaluer les performances de ces nouveaux algorithmes et même de ceux précédemment proposés dans le cas général. Une telle évaluation est difficile pour au moins deux raisons. D'une part, parce que les CSP issus de problèmes réels se résolvent souvent en utilisant des heuristiques simples, mais malheureusement trop dépendantes du domaine pour être généralisées. D'autre part, parce qu'il est difficile d'établir un processus expérimental qui permette d'affirmer que les comparaisons faites relèvent du cas général. Jusqu'à présent, la plupart des tentatives de comparaisons ont été faites sur des CSP particuliers, tels que : le problème des dames (il s'agit de placer n dames sur un échiquier $n \times n$, sans que deux dames puissent se prendre selon les règles du jeu d'échec), le problème du zèbre (voir annexe 2), ou bien des CSP engendrés de manière aléatoire. Les travaux les plus marquants sont ceux de Haralick et Elliot et ceux de Nadel. Leurs résultats ont conduit aux conclusions suivantes :

- La procédure de prétraitement ayant le meilleur rapport entre l'efficacité et le coût est la consistance d'arc, et il est intéressant de l'utiliser.

- La procédure de recherche la plus efficace est FC. Il semble préférable de ne pas utiliser de filtrage trop puissant lors de la procédure de recherche.

- Les heuristiques basées sur les choix les plus contraints donnent les meilleurs résultats.

Cependant ces conclusions ont été récemment remises en cause par Sabin et Freuder [Sabin and Freuder, 1994a; Sabin and Freuder, 1994b]. Après de nombreuses expérimentations, ils ont constaté les deux faits suivants :

1. L'idée communément admise qui affirme que l'utilisation d'un filtrage comme prétraitement réduit l'espace de recherche n'est pas toujours vérifiée en pratique.
2. Il vaut mieux utiliser un algorithme de maintien de la consistance d'arc pendant la recherche plutôt que FC.

Pour établir le premier fait, Sabin et Freuder ont comparé sur des CSP aléatoires de 50 variables, 8 valeurs avec un taux de satisfaisabilité de 50%, FC avec l'ordre dynamique sur les variables qui consiste à prendre la variable ayant le plus petit domaine, que nous noterons FCdmin, et le même processus précédé du filtrage par consistance d'arc AC-4, que nous noterons AC-FC. Ils ont observé trois phénomènes pour le nombre de tests de consistance réalisé :

- AC-FC est moins bon en moyenne que FCdmin.
- Le filtrage par consistance d'arc prend parfois plus de temps que la résolution du problème par FCdmin.
- Les efforts de recherche de FC sont quelquefois plus importants après un prétraitement que lorsque celui-ci n'a pas été effectué.

Les conclusions auxquelles ils aboutissent sont établies à partir du nombre de tests de consistance réalisés et elles se vérifient nettement moins si l'on prend plutôt en compte le temps. On peut raisonnablement conclure de cette expérimentation que l'ordre dynamique basé sur la minimalité des domaines ne semble pas être le meilleur ordre pour AC-FC. Il ressort d'ailleurs de travaux récents de certains auteurs [Baker, 1994] que les performances des ordres sont tout autant dépendantes de l'algorithme employé que du domaine d'application. Nous partageons entièrement ce point de vue.

Processus expérimental

Nous allons brièvement présenter le processus expérimental permettant d'évaluer les méthodes que nous proposerons dans les chapitres suivants pour différents types de CSP.

Pour tester la résolution de Iso-ssgrpartiel nous avons choisi de ne considérer que des exemples réels plutôt que des exemples générés aléatoirement. À partir d'une base de données de graphes moléculaires contenant plus de 16 000 molécules, nous avons sélectionné les squelettes[4] des molécules ayant plus de 20 sommets. Nous avons ensuite constitué une base de tests en choisissant aléatoirement 10 000 couples de graphes tels qu'a priori il soit possible qu'il existe un morphisme injectif du premier élément du couple dans le second, autrement dit que tout couple (H, G) vérifie: $X_H \leq X_G$ et $E_H \leq E_G$ et $degrémax(X_H) \leq degrémax(X_G)$. 17% des tests vérifient l'inconsistance d'arc et dans 4% des cas il existe un morphisme injectif entre les deux graphes.

Les autres problèmes réels utilisés sont les problèmes d'affectation de fréquences (*Radio Link Frequency Assignement Problem*, RLFAP en abrégé). Leurs instances sont disponibles par internet via une messagerie électronique automatique (csp-list@saturne.cert.fr)[5]. Ce type de problème consiste à affecter des fréquences à des noeuds de communication de telle sorte qu'aucune interférence n'intervienne. Il s'agit de rechercher si les réseaux admettent une solution, et dans le cas où ils n'en possèdent pas, de maximiser une fonction. Nous nous sommes limités à la recherche de solution. L'intérêt de ces problèmes est leur diversité, certains sont très faciles et d'autres très difficiles, et leur taille qui est importante (900 variables possédant environ 30 valeurs et plus de 4000 contraintes pour certains). Nous présentons leurs caractéristiques dans le tableau donné en figure 4.12.

Nous présenterons aussi des résultats pour les CSP aléatoires. Les expérimentations faites pour ce type de CSP sont très discutables d'un point de vue théorique, mais elles présentent l'avantage d'être utilisées par de nombreux auteurs. Pour tenter de comparer de manière générale les algorithmes, l'idée est de construire des réseaux de contraintes de manière aléatoire, puis de tester les différentes méthodes de résolu-

4. Le squelette d'une molécule est le graphe moléculaire sans ses étiquettes, c'est-à-dire que l'on ne tient compte ni des types des atomes ni de l'ordre des liaisons.

5. Comme cela nous est explicitement demandé, nous remercions le Centre Électronique de l'Armement pour nous avoir fourni ces problèmes.

	RLFAP-1	RLFAP-2	RLFAP-3	RLFAP-4	RLFAP-5	RLFAP-6
variables	916	200	400	680	400	200
contraintes	5548	1235	2670	3967	2598	1322

	RLFAP-7	RLFAP-8	RLFAP-9	RLFAP-10	RLFAP-11
variables	400	916	680	680	680
contraintes	2865	5744	4103	4103	4103

Pour tous les problèmes les variables peuvent prendre environ 30 valeurs.

FIG. 4.12 - *Caractéristiques des problèmes d'affectation de fréquences.*

tion sur ces réseaux.

Nous avons construit les réseaux de contraintes aléatoires à partir des paramètres suivants :

- le nombre n de variables.
- la taille d de chaque domaine. Tous les domaines ont la même taille.
- la densité pc du graphe des contraintes, c'est-à-dire que le graphe possède $pc \times n(n-1)/2$ arêtes.
- la probabilité pu qu'un n-uplet d'une contrainte soit admissible.

Même si la construction aléatoire a comme objectif de prendre en compte les différents types de problèmes que l'on peut rencontrer, elle ne permet pas de mettre en évidence certains cas particuliers, comme par exemple les réseaux dont le graphe des contraintes est régulier. De plus il est quasiment impossible de simuler avec cette méthode certains problèmes réels en respectant leurs caractéristiques. Ainsi, certains problèmes d'affectation de fréquences ont une densité de l'ordre de 1% et sont très difficiles à résoudre, et nous n'avons pas réussi à reproduire ce type de problème avec des réseaux de contraintes aléatoires.

Nous insistons sur le fait qu'il ne s'agit pas ici d'utiliser les problèmes aléatoires afin de déterminer une méthode générale de résolution. Notre but est seulement de comparer les performances de plusieurs algorithmes à l'aide de ce type de problèmes.

Tous les algorithmes présents dans cette thèse ont été implémentés en C++ et compilé avec le compilateur Solaris 2.0 avec l'option -O3. Tous les temps ont été mesurés sur une SPARC-5.

Chapitre 5

Amélioration de la consistance d'arc

Ce chapitre réunit et étend des travaux réalisés dans un premier temps de manière indépendante par Bessière et Régin [Bessière and Régin, 1995] et par Freuder [Freuder, 1994; Freuder, 1995], puis ensemble [Bessière *et al.*, 1995] dans un second temps.

Disposer de tous les couples admissibles des contraintes, sans avoir besoin de les calculer, dès la construction du réseau de contraintes est un avantage important pour filtrer un réseau par la consistance d'arc. Or, de manière générale, cette information n'est pas disponible et le coût d'un test de consistance n'est pas nécessairement négligeable. Il est donc important que les algorithmes d'AC effectuent le moins possible de tests de consistance. C'est pourquoi nous proposons dans la première partie de ce chapitre deux algorithmes qui utilisent les propriétés des contraintes afin de déduire le résultat de certains tests de consistance. Le premier, appelé AC-7, tire parti de la bidirectionnalité des contraintes, propriété simple commune à toutes les contraintes; l'autre, dénommé AC-Inférence, généralise l'idée de déduction de la présence ou de l'absence de support utilisé par AC-7 pour les contraintes bidirectionnelles à des contraintes présentant des propriétés plus complexes. Nous identifions plusieurs propriétés des contraintes permettant de telles déductions.

AC-7 est un algorithme général de consistance d'arc, car il ne dépend pas de propriétés particulières liées à une classe limitée de contraintes. Il utilise simplement la bidirectionnalité des supports : si (i, a) supporte (j, b) alors (j, b) supporte aussi (i, a).

L'exploitation de cette propriété confère à AC-7 un avantage par rapport aux autres algorithmes d'AC. De plus limiter l'attention d'AC-7 à la bidirectionnalité autorise une implémentation efficace en espace qui est comparable aux meilleurs algorithmes, ce qui n'est pas possible avec AC-Inférence. D'un autre côté, AC-Inférence permet d'obtenir une meilleure efficacité en temps en tirant avantage de certaines connaissances sur des classes de contraintes réduites. Par ailleurs, une forme particulière de déduction peut être faite en regroupant les contraintes identiques, c'est-à-dire celles dont la sémantique est la même et portant sur les mêmes domaines. L'idée est de mettre en commun le travail réalisé pour ces contraintes afin d'éviter des répétitions inutiles. Le regroupement de ces contraintes permet également de réduire la complexité en espace d'AC-Inférence.

Les travaux que nous venons de mentionner ne sont pas utilisables pour Iso-ssgrpartiel puisque les contraintes sont données en extension. La seconde partie de ce chapitre propose certaines heuristiques permettant d'accélérer très souvent la fermeture par consistance d'arc. Notamment, nous remettons en cause le schéma classique des algorithmes en abandonnant l'emploi de deux phases successives et distinctes, c'est-à-dire une d'initialisation et une de propagation, pour adopter un nouveau schéma dans lequel toute valeur sans support est immédiatement propagée.

Nous commencerons par présenter une expérimentation montrant l'avantage que procure la connaissance de tous les couples admissibles des contraintes. Puis, nous détaillerons les algorithmes AC-7 et AC-Inférence. Nous nous intéresserons alors au principe de déduction des supports et au regroupement des contraintes identiques. Ensuite nous proposerons deux heuristiques permettant d'accélérer le processus de fermeture par consistance d'arc. Enfin, nous donnerons un certain nombre de résultats expérimentaux mettant en évidence l'intérêt de notre approche.

5.1 Connaissance des couples admissibles

Disposer de cette connaissance est un avantage très important. Pour nous en convaincre nous avons comparé deux versions d'AC-4 et deux versions d'AC-6 pour Iso-ssgrpartiel. La première version de chaque algorithme n'utilise pas cette connaissance tandis que l'autre la prend en compte. Dans ce dernier cas, la fonction d'initia-

lisation d'AC-4 et la fonction existeSupport d'AC-6 sont celles qui ont été présentées dans le chapitre précédent. Les contraintes de différence ont été implémentées selon la méthode proposée par Mohr et Masini.

Le tableau suivant résume les résultats que nous avons obtenus. Les temps donnés sont les temps totaux pour filtrer 10 000 réseaux.

AC-4	AC-4 modifié	AC-6	AC-6 modifié
640 170 ms	68 410 ms	434 674 ms	47 676 ms

Il apparaît clairement que la connaissance de tous les couples autorisés est un avantage énorme, puisque l'on gagne presque un facteur 10. Notons aussi que chaque version d'AC-6 est meilleure que la version correspondante d'AC-4, mais qu'AC-4 modifié est plus performante que la version classique d'AC-6.

5.2 AC-7

L'algorithme, appelé AC-7 dans ce chapitre et AC-6++ dans ses premières versions [Bessière and Régin, 1994a; Bessière and Régin, 1994b; Bessière and Régin, 1995], qui est présenté ici calcule la fermuture par consistance d'arc après un nombre optimal de vérifications de tests de consistance en utilisant l'indirection des contraintes, $C_{ij}(a,b) = C_{ji}(b,a)$, tout en conservant les propriétés d'AC-6.

La redondance potentielle dans la recherche de support bidirectionnel a été reconnue auparavant. En effet, quand AC-3 supprime une valeur du domaine d'une variable x_i car elle n'a plus de support sur C_{ij}, il réalise que cette suppression ne peut pas priver une valeur de D_j de support sur C_{ji}. DEEB [Gaschnig, 1978] utilise une procédure "*revise-both*" qui anticipait plus directement AC-7. Après avoir recherché un support pour les valeurs de D_i sur une contrainte C_{ij}, il recherchait immédiatement un support sur C_{ji} pour les valeurs de D_j, mais seulement pour celles qui ne sont pas déjà des supports pour les valeurs de D_i. Gaschnig met en évidence que cela permet d'éviter des vérifications non nécessaires effectuées par AC-3 la première fois que la viabilité des valeurs de D_i et D_j sont vérifiées entre elles. Mais Gaschnig conclut, de manière incorrecte selon nous, que DEEB évite aussi tous les tests de consistance que AC-3 évite en utilisant la bidirectionnalité.

Aucun algorithme jusqu'à présent n'a utilisé de déduction de supports basé sur la bidirectionnalité. Par exemple supposons que la première valeur, d_1, et la dernière valeur, d_{100}, d'une variable x_i soient supportées par la valeur (i, a). Si plus tard d_1 est supprimée durant la phase de propagation, ni AC-3, ni DEEB ne se souviendront que a est supportée par d_{100}. En effet, ils auront besoin d'étudier les 98 autres valeurs de x_i, avant de redécouvrir que a est supportée par d_{100}.

L'algorithme que nous proposons est une amélioration de AC-6 qui tient compte de l'indirection des contraintes. AC-7 est optimal dans le nombre de tests de consistance qu'il effectue, et conserve les complexités d'AC-6. Nous allons dans les sections suivantes détailler le fonctionnement de cet algorithme.

5.2.1 Principes d'AC-7

La consistance d'arc est basée sur la notion de support. Aussi longtemps qu'une valeur (i, a) a un support dans le domaine de chaque autre variable x_j reliée à x_i dans le graphe des contraintes, (i, a) est viable. Mais s'il existe une variable pour laquelle a ne satisfait pas la contrainte, alors a doit être éliminé de D_i. AC-6, qui assigne un ordre sur les valeurs de chaque domaine D_i, recherche un support (le premier) pour chaque valeur (i, a) sur chaque contrainte C_{ij} pour prouver que (i, a) est viable à un instant donné. Lorsque (j, b) est trouvé comme étant le plus petit support de (i, a) sur C_{ij}, (i, a) est ajouté à S_{jb}, la liste des valeurs qui ont (j, b) à l'instant courant comme plus petit support. Mais AC-6 n'utilise pas l'égalité entre $C_{ij}(a, b)$ et $C_{ji}(b, a)$, ainsi lorsqu'il recherche un support pour (j, b) sur C_{ji}, il ne se sert pas du fait que $C_{ij}(a, b)$ a peut être déjà été vérifié pour de nombreuses valeurs a de x_i lorsqu'on leur recherchait des supports.

Considérons par exemple deux variables x_i et x_j ayant pour domaines $D_i = \{a, b, c, d\}$ et $D_j = \{e, f, g, h\}$ munis de l'ordre lexicographique et la contrainte C_{ij} entre ces deux variables dont les couples admissibles sont $(a, g), (b, h), (c, e), (c, h), (d, e), (d, f), (d, g)$. Cette contrainte peut se représenter à l'aide du schéma donné en figure 5.1, où une arête entre deux sommets indique l'admissibilité du couple.

Supposons que l'on ait recherché d'abord si les valeurs de x_i sont viables sur C_{ij}. À ce moment nous aurons les listes de supports suivantes, si l'on ne considère que la contrainte C_{ij} : $S_{je} = \{(i, c), (i, d)\}$, $S_{jf} = \varnothing$, $S_{jg} = \{(i, a)\}$, $S_{jh} = \{(i, b)\}$

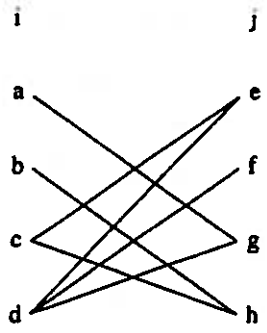


FIG. 5.1 - *Une contrainte et ses couples admissibles.*

Pour construire ces listes AC-6 a vérifié successivement pour la contrainte C_{ij} la consistance des couples: $(a,e),(a,f),(a,g),(b,e),(b,f),(b,g),(b,h),(c,e),(d,e)$. Graphiquement, nous pouvons exprimer cela par le dessin de la figure 5.2, où une flèche en pointillée exprime un test de consistance négatif, tandis qu'une flèche en trait plein représente un test positif.

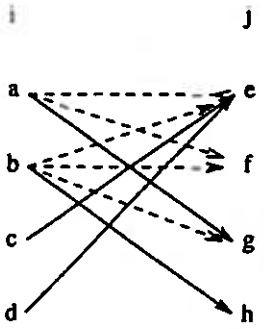


FIG. 5.2 - *AC-6: recherche de la viabilité des valeurs de D_i.*

Étudions maintenant le comportement d'AC-6 lorsque l'on recherche si les valeurs de x_j sont viables sur la contrainte C_{ji}. AC-6 étudie successivement les couples de valeurs: $(e,a),(e,b),(e,c),(f,a),(f,b),(f,c),(f,d),(g,a),(h,a),(h,b)$.

Or, on peut immédiatement remarquer qu'avant la recherche de support sur C_{ji} pour la valeur e, la liste des supports S_{je} réduite à la contrainte C_{ij} n'est pas vide. Si la contrainte n'est pas orientée alors n'importe quelle valeur que e supporte sur la contrainte C_{ij}, supporte également e sur la contrainte C_{ji}. On peut se servir de ce résultat pour trouver plus rapidement un support. L'idée consiste donc à ne pas rechercher systématiquement les supports en suivant l'ordre des domaines, mais plu-

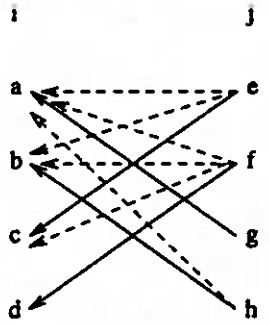


FIG. 5.3 - *AC-6 : recherche de la viabilité des valeurs de D_j.*

tôt à commencer par rechercher si la valeur n'est pas elle-même un support. Cela signifie que, ***si les contraintes ne sont pas orientées, on peut rechercher un support pour une valeur sur une contrainte dans la liste des valeurs supportées par cette valeur sur cette contrainte.*** Ainsi pour notre exemple, on trouve un support pour e sur la contrainte C_{ji} grâce à la liste S_{je} réduite à la contrainte C_{ij}, ce support est (i, c).

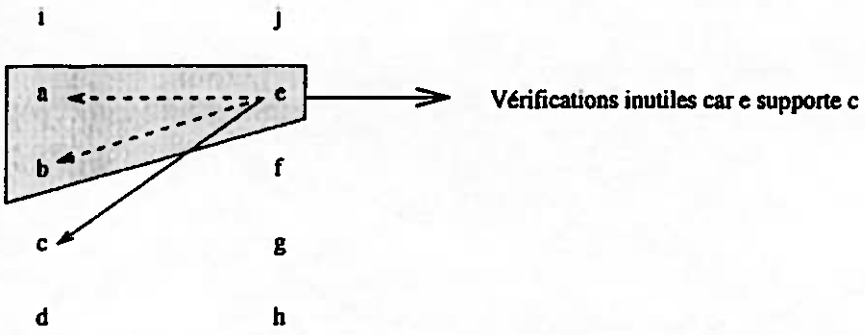


FIG. 5.4 - *Tests de consistance évités lorsque la liste des valeurs supportées n'est pas vide.*

On peut appliquer le même raisonnement pour la valeur g de x_j, pour trouver le support (i, a). En procédant ainsi nous éviterons l'étude de la consistance des couples : $(e, a), (e, b), (e, c), (g, a), (h, a), (h, b)$. Cette idée repose sur le fait que si l'on a été amené à vérifier positivement la consistance d'un couple, cette information est mémorisée dans une liste de supports. Nous pouvons donc éviter de revérifier la consistance d'un couple (e, a) sur une contrainte C_{ji}, si l'on a vérifié positivement une consistance $C_{ij}(c, e)$.

Le problème qui se pose maintenant est :

«Comment éviter de revérifier la consistance d'un couple (f, a) sur une contrainte C_{ji} si l'on a jamais vérifié positivement une consistance $C_{ij}(b, f)$?»

Globalement, il s'agit ici d'exploiter de l'information négative. On est confronté à ce problème lorsque l'on cherche un support pour la valeur f de x_j sur la contrainte C_{ji}. En effet on a déjà étudié la consistance des couples (a, f) et (b, f) pour C_{ij} et le résultat a été négatif. Comme précédemment, nous devons considérer uniquement lors de l'étude de C_{ji} ce qui c'est passé lorsque AC-6 a étudié la contrainte C_{ij}. Considérons pour chaque valeur de x_i la dernière valeur dans l'ordre du domaine de D_j pour laquelle une consistance a été vérifiée. Deux cas sont alors possibles, soit aucun support n'a été trouvé et dans ce cas la valeur a été supprimée de D_i, soit une valeur consistante a été trouvée et c'est la dernière valeur étudiée. Aucune des valeurs précédant (dans l'ordre de D_j) la dernière valeur étudiée n'est consistante avec la valeur courante. Comme les contraintes ne sont pas orientées, nous pouvons en déduire qu'une valeur p de D_j n'est pas consistante avec les valeurs de D_i dont la dernière valeur étudiée sur C_{ij} suit p dans l'ordre de D_j. C'est ce résultat que nous allons utiliser. Pour notre exemple, la dernière valeur étudiée pour a sur la contrainte C_{ij} est g, a n'est donc pas consistante avec toutes les valeurs du domaines de D_j qui précèdent g dans l'ordre de D_j. Comme f précède g dans l'ordre de D_j, nous savons que f n'est pas consistante avec a et sans avoir à tester cette consistance. On peut appliquer le même raisonnement avec la valeur b dont la dernière valeur étudiée est h qui est supérieure à f.

Si l'on applique les deux principes précédents à notre exemple nous étudierons la compatibilité des couples : $(a, e), (a, f), (a, g), (b, e), (b, f), (b, g), (c, e), (d, e)$ pour déterminer la viabilité des valeurs de D_i pour la contrainte C_{ij}, puis la compatibilité des couples : $(f, c), (f, d)$ pour déterminer les valeurs viables de D_j par rapport à la contrainte C_{ji}. On remarque un gain important par rapport à AC-6.

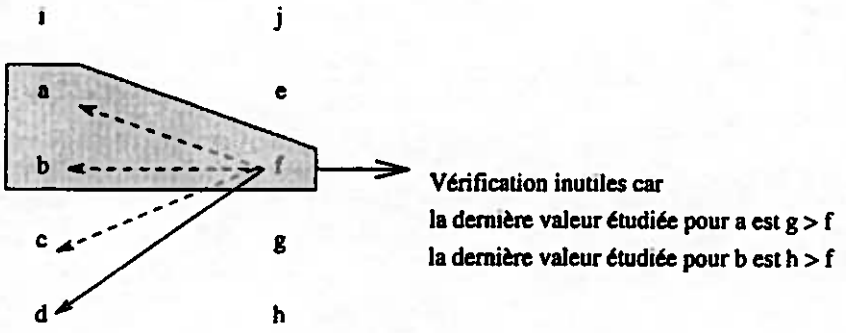


FIG. 5.5 - *Tests de consistance évités grâce à la mémorisation de la dernière valeur étudiée.*

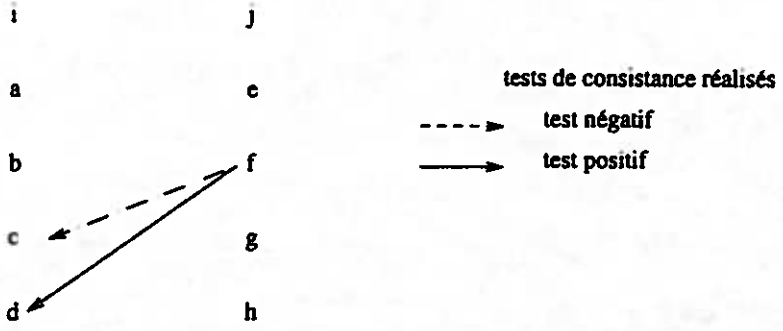


FIG. 5.6 - *AC-7: recherche de la viabilité des valeurs de D_j.*

5.2.2 L'algorithme

AC-7 est construit pour maintenir les quatres propriétés suivantes tout au long de l'algorithme :

1. ne jamais vérifier $C_{ij}(a, b)$ s'il existe c encore dans D_j tel que $C_{ij}(a, c)$ ait déjà été vérifié positivement.

2. ne jamais vérifier $C_{ij}(a, b)$ s'il existe c encore dans D_j tel que $C_{ji}(c, a)$ ait déjà été vérifié positivement.

3. ne jamais vérifier $C_{ij}(a, b)$ si :
   (3a) $C_{ij}(a, b)$ a déjà été vérifié.
   (3b) $C_{ji}(b, a)$ a déjà été vérifié.

4. maintenir une complexité en espace en $O(ed)$.

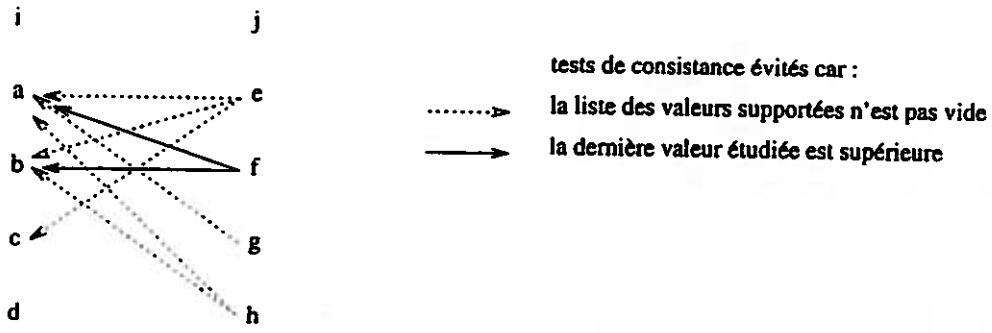


FIG. 5.7 - *Tests de consistance évités par AC-7 par rapport à AC-6.*

Les propriétés 1) et 3a) sont garanties par AC-6. Les propriétés 1), 2) et 3) définissent ce que l'on appelle l'optimalité d'AC-7. Cette optimalité est donc définie selon un ordre pour les variables, un ordre pour les valeurs et un ordre pour les arêtes. Elle n'implique pas ***le*** nombre minimum de tests de consistance, mais un nombre minimal selon l'ordre utilisé [Wallace and Freuder, 1992].

Sur la figure 5.7, la propriété 2) permet d'éviter la vérification des tests de consistance $C_{ji}(e,a), C_{ji}(e,b), C_{ji}(h,a)$ et la propriété 3b) les tests $C_{ji}(g,a), C_{ji}(h,b), C_{ji}(e,c)$ qui sont positifs et les tests $C_{ji}(f,a), C_{ji}(f,b)$ qui sont négatifs.

Le fonctionnement d'AC-7 est semblable à celui d'AC-6, le principe des phases d'initialisation et de propagation est inchangé, aussi AC-7 impose comme AC-6 un ordre total sur les domaines et des fonctions d'accès à certaines informations de ces domaines (DERNIER, SUIVANT, SUPPRIMER), et manipule une liste (*listeAttente*) contenant les valeurs supprimées des domaines et dont les conséquences de leur suppression n'ont pas encore été étudiées.

Comme nous l'avons vu précédemment, AC-7 particularise les structures de données de AC-6 par rapport à chaque contrainte. Ainsi pour n'importe qu'elle valeur (i,a), AC-7 a besoin de connaître les valeurs qu'elle supporte pour chaque contrainte et non plus pour l'ensemble des contraintes comme c'était le cas avec AC-6. Les listes des supports sont donc spécialisées pour chaque contrainte. Nous noterons $S_{ia}[j]$ l'ensemble des valeurs de D_j que a supporte pour la contrainte C_{ji}. On remarque immédiatement l'égalité suivante : $S_{ia} = \cup_{j \in \Gamma(i)} S_{ia}[j]$.

La dernière valeur étudiée pour une valeur a d'une variable x_i sur une contrainte C_{ij} est mémorisée à l'aide d'un tableau $L_{ia}[j]$. Ce tableau représente la plus petite

valeur de D_j qui peut être compatible avec (i, a). Ces tableaux sont nécessaires afin de garantir la propriété 3a) et la propriété 3b) pour les tests de consistance négatifs. La propriété 3a) est vraie dans AC-6 sans utiliser de tableaux car (j, b) est toujours le plus petit support des valeurs contenues dans S_{jb}. Dans AC-7, cette propriété forte n'est plus assurée.

Les nouvelles structures de données utilisées par l'algorithme sont donc :

- $S_{jb}[i] = \{(i, a) | (j, b)$ supporte (i, a) pour la contrainte $C_{ij}\}$. C'est l'ensemble des valeurs de D_i pour lesquelles (j, b) est le support courant. Contrairement à AC-6, le support courant n'est pas nécessairement le plus petit support.

- $L_{ia}[j]$ indique la dernière valeur de D_j qui a été étudiée pour la valeur a de la variable x_i sur la contrainte C_{ij}. Ces tableaux sont mis-à-jour par AC-7 afin de garantir que tout b de D_j compatible avec (i, a) est plus grand que $L_{ia}[j]$.

On considérera, pour ne pas réécrire inutilement les fonctions d'initialisation et de propagation, que les structures de données sont correctement initialisées.

La fonction de recherche d'un nouveau support diffère par rapport à AC-6. En effet elle recherche tout d'abord un support dans la liste des valeurs supportées, puis éventuellement dans le domaine, et dans ce cas, elle modifie le tableau des dernières valeurs étudiées. Lorsqu'une valeur atteinte n'appartient plus au domaine, elle est supprimée de la liste des valeurs supportées, pour qu'elle ne puisse pas être atteinte plusieurs fois. Comme pour AC-6, lorsqu'un nouveau support est trouvé pour une valeur sur une contrainte, cette dernière est supprimée de la liste des valeurs supportées de son précédent support afin qu'elle ne puisse apparaître au plus qu'une seule fois dans une de ces listes pour chaque contrainte.

5.2.3 Implémentation des structures de données

La particularisation des listes de supports en fonction des contraintes pose 2 problèmes, car elles sont utilisées de deux manières différentes :

1. Par la phase de propagation pour connaître l'ensemble des valeurs que supporte une valeur b d'une variable x_j qui est supprimée de D_j. Pour ne pas perdre de temps, notamment par rapport à AC-6, cet accès doit pouvoir être effectué

Algorithme 15 AC-7 : la fonction EXISTESUPPORT

```
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
  trouvé ← faux
  /* recherche dans la liste des valeurs supportées */
  tant que ¬trouvé et S_ia[j] ≠ ∅ faire
  |  c ← PREMIER(S_ia[j])
  |  si c ∉ D_j alors SUPPRIMER(c, S_ia[j])
  |  sinon trouvé ← vrai
  /* recherche dans le domaine */
  tant que ¬trouvé et L_ia[j] ≤ DERNIER(D_j) faire
  |  c ← SUIVANT(L_ia[j], D_j)
  |  L_ia[j] ← c
  |  si L_jc[i] < a alors
  |  |  si C_ij(a, c) alors trouvé ← vrai
  si trouvé alors
  |  si b ≠ nil alors SUPPRIMER((i, a), S_jb)
  |  AJOUTER((i, a), S_jc[i])
  retourner trouvé
```

rapidement et surtout sans forcément accéder à chacune des listes $S_{jb}[i]$ pour chaque voisin de x_j, car il est possible que b ne supporte aucune valeur sur certaines contraintes, c'est-à-dire que certaines de ces listes soient vides.

2. Par la fonction EXISTESUPPORT pour rechercher si une valeur a d'une variable x_i possède un support sur une contrainte C_{ij}. Cela se produit lorsque l'on ne connaît encore aucun support ou bien quand une valeur b de x_j qui supportait a vient à disparaître et que l'on étudie les conséquences de cette suppression. Comme on commence par rechercher un support dans la liste des valeurs que a supporte sur C_{ji} c'est-à-dire dans la liste $S_{ia}[j]$, l'algorithme doit pouvoir rapidement accéder à une de ces listes. On retrouve ce type de problème pour les tableaux des dernières valeurs étudiées.

Il faut, de plus, trouver une implémentation qui respecte une complexité en espace en $O(ed)$ (cf propriété 4).

On peut remédier au premier problème en construisant la liste S_{jb} de l'ensemble des valeurs supportées par b sur toutes les contraintes C_{*j} à partir de la concaténation de toutes les listes $S_{jb}[i]$ qui ne sont pas vides. Chaque sous-liste $S_{jb}[i]$ connaît sa position dans la liste S_{jb}. Supposons par exemple que (x_j, b) supporte :

sur C_{ij} les valeurs : a, b, c.
sur C_{kj} aucune valeur.
sur C_{lj} les valeurs a, f.
On aura la liste $Sjb = \{(x_i, a), (x_i, b), (x_i, c), (x_l, a), (x_l, f)\}$.

Pour retrouver les sous-listes correspondant à $S_{jb}[i], S_{jb}[k], S_{jb}[l]$ on peut imaginer un mécanisme de pointeurs. Ainsi une liste $S_{jb}[i]$ possédera deux pointeurs, l'un référençant le début de la liste $S_{jb}[i]$ dans S_{jb} et l'autre la fin.

Deux types de suppression peuvent intervenir. L'une se produit pendant la phase de propagation et est totale, ce qui ne pose pas de problème car plus aucun accès à ces listes ne sera fait. L'autre intervient lorsque l'on recherche un nouveau support à une valeur dans une sous-liste. Les éléments rencontrés qui n'appartiennent plus au domaine doivent être supprimés pour qu'ils ne puissent pas être atteint plusieurs fois. Comme l'accès à une sous-liste se fait toujours par l'intermédiaire d'un pointeur on pourra le modifier aisément.

L'ajout d'un élément se fait toujours par rapport à une sous-liste donnée dont les pointeurs associés sont connus. Il n'y a donc pas de problème particulier liés aux mises à jour.

Pour résoudre le second problème il faut prendre quelques précautions afin d'éviter une complexité en espace de l'ordre de n^2 qui peut être supérieure à ed pour certains réseaux. On doit pouvoir accéder à une liste de supports $S_{ia}[j]$ soit en sachant que (i, a) était supportée par un (j, b), soit en passant par l'intermédiaire d'une contrainte C_{ij}. Pour que ces accès se fassent rapidement, il faut que l'on puisse retrouver les pointeurs d'une sous-liste à partir des contraintes. L'ensemble des pointeurs pour une valeur doit donc être codé sous la forme d'un tableau. Ce tableau ***ne doit pas être indicé directement*** par le numéro de la variable voisine, mais plutôt à l'aide d'une indirection, sinon sa dimension serait n pour chaque variable. Par exemple, si la variable x_i est reliée aux variables 3,5,12,8 dans le graphe des contraintes, la dimension

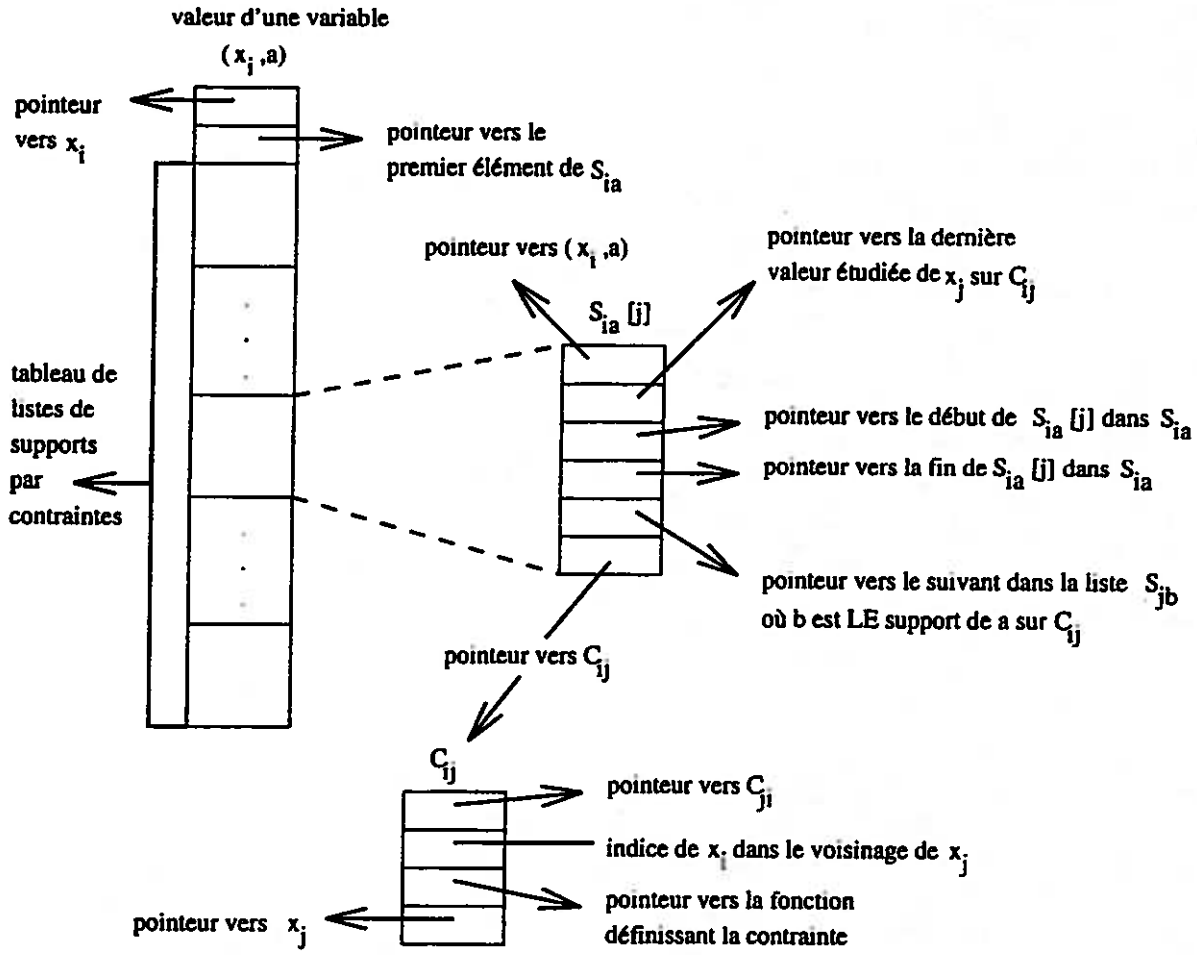


FIG. 5.8 - *AC-7: structures de données*

du tableau doit être 4 puisque x_i a quatre voisins et non pas 12 qui est le numéro de voisins le plus élevé. Le problème réside dans la mise en oeuvre de ce mécanisme d'indirection. Considérons une contrainte C_{ij} nous avons besoin de connaître l'indice p par lequel on accède aux tableaux des valeurs de x_j supportées par les valeurs de x_i, c'est-à-dire aux tableaux $S_{i*}[p]$ et aussi à l'indice q permettant d'accéder aux valeurs de x_i supportées par les valeurs de x_j, c'est-à-dire aux tableaux $S_{j*}[q]$ [1]. Nous pouvons mémoriser ces informations au moment de la construction du graphe des contraintes. Par exemple supposons que la variable 4 soit reliée aux variables 3,5,12 et 8 dans cet ordre et que la variable 5 soit reliée aux variables 6,9,4 dans cet ordre, alors la

1. Les tableaux $S_{i*}[p]$ et $S_{j*}[q]$ étaient respectivement notés, jusqu'à maintenant, $S_{i*}[j]$ et $S_{j*}[i]$ par abus de langage.

contrainte C_{45} contiendra deux indices : l'un correspond à la position de 5 dans la liste des voisins de 4 (ici 2) et l'autre à l'indice de 4 dans la liste des voisins de 5 (ici 3). De manière analogue, la contrainte C_{54} aura 3 comme premier indice et 2 comme second. Si l'on attache maintenant la contrainte qui a été à l'origine du calcul d'un support sur chaque support, on pourra toujours passer par son intermédiaire. En procédant ainsi, le passage d'une liste de supports $S_{ia}[j]$ à une autre liste de supports $S_{jb}[i]$ peut donc se faire en $O(1)$.

Un exemple de réalisation sur machine des structures de données est résumé par la figure 5.8.

5.2.4 Preuves

Preuve d'AC-7

AC-7 est similaire à AC-6 excepté qu'il ne garantit pas la connaissance du plus petit support pour une valeur sur une contrainte, mais seulement l'existence d'un support et d'une borne inférieure avant laquelle aucun support n'existe.

Montrons que lorsqu'une valeur (j, b) n'a pas de support sur une contrainte C_{ji}, tous les tests de consistance entre b et les valeurs de D_i ont été effectués. C'est la fonction EXISTESUPPORT qui réalise les tests de consistance. Cette fonction retourne *nil* uniquement si la dernière valeur étudiée est la dernière valeur du domaine et si le dernier test possible s'est avéré négatif. Par conséquent, toutes les valeurs du domaine ont été testées, et l'algorithme ne peut donc pas omettre un support.

A chaque fois qu'une valeur (j, b) est supprimée, elle est introduite dans la *listeAttente*. Elle reste dans cette liste tant qu'un nouveau support n'a pas été recherché pour toutes les valeurs que b supportait sur chaque contrainte contraignant x_j. A chaque fois qu'une valeur n'a pas de support sur une contrainte, elle est supprimée du domaine. Donc chaque valeur de chaque domaine a au moins un support dans $\mathcal{D} \cup$ *listeAttente* pour chaque contrainte C_{ij}. AC-7 se termine avec une *listeAttente* vide. Aussi après AC-7 chaque valeur de $\mathcal{D}$ a un support dans $\mathcal{D}$ sur chaque contrainte. Donc $\mathcal{D}$ vérifie la consistance d'arc.

Preuve d'optimalité

L'optimalité d'AC-7 est définie par rapport aux propriétés 1), 2), 3a) et 3b) précédemment présentées. Dans ce paragraphe nous montrons que ces propriétés sont vraies tout au long de l'algorithme.

• **propriété 1) et 3a)** :

Le raisonnement est le même que pour AC-6. En effet, AC-7 arrête de rechercher un support pour une valeur (i, a) sur une contrainte C_{ij} dès qu'il a trouvé le premier support. Un test $C_{ij}(a, b)$ ne peut être vérifié deux fois car AC-7 commence à rechercher un nouveau support pour (i, a) sur C_{ij} à partir de l'endroit où il s'était arrêté lors de la précédente recherche.

• **propriété 2)** :

A chaque fois qu'un test de consistance $C_{ij}(a, b)$ est positif, AC-7 mémorise b dans $S_{ia}[j]$. AC-7 recherche un support pour (i, a) sur C_{ij} uniquement si $S_{ia}[j] \cap D_j = \varnothing$, donc nous sommes sûrs que $C_{ij}(a, b)$ n'est vérifié que dans le cas où aucun autre test précédemment effectué sur C_{ji} ne peut prouver que (i, a) a un support dans D_j.

• **propriété 3b)** :

Si $C_{ij}(a, b)$ a déjà été vérifié nous devons montrer que $C_{ji}(b, a)$ ne peut pas l'être. Si $C_{ij}(a, b)$ a été positif, alors a est mémorisée dans la liste $S_{jb}[i]$. AC-7 commence par rechercher un support dans la liste des valeurs supportées, aussi on est certain que $C_{ji}(b, a)$ ne sera jamais testé car a sera trouvé dans la liste $S_{jb}[i]$. Si $C_{ij}(a, b)$ a été testé négativement alors $L_{ia}[j]$ est strictement supérieur à b, ce qui interdit le test de $C_{ji}(b, a)$ lors de la procédure de recherche d'un nouveau support.

5.2.5 Complexités en temps et en espace

Un test de consistance n'étant effectué qu'une seule fois, la complexité en temps due aux vérifications de consistance est bornée par le nombre de tests possibles, c'est-à-dire par $O(ed^2)$. La fonction EXISTESUPPORT, pour une valeur a d'une variable x_i sur une contrainte C_{ij}, a un coût global maximal en $O(d)$. Le coût global, c'est-à-dire quelque soit le nombre de fois où la fonction est appelée, de EXISTESUPPORT est en : $\sum_{x_i \in X} \sum_{a \in D_i} \sum_{j \in \Gamma(x_i)} O(d) = O(ed^2)$. La complexité en temps d'AC-7 dans le pire des cas est donc en $O(ed^2)$.

La complexité en espace d'AC-6 est dans le pire des cas en $O(ed)$ à cause de la

taille des listes S_{ia}. Dans AC-7 chaque valeur a de chaque variable x_i est supportée par une seule valeur pour chaque contrainte, ainsi pour une contrainte C_{ij}, a ne peut appartenir qu'à une seule liste $S_{jb}[i]$. Le nombre de valeurs de D_i supportées sur la contrainte C_{ij} par toutes les valeurs de D_j est égal au cardinal de D_i. Pour une contrainte C_{ij}, le nombre de supports mémorisés est au maximum en $O(d)$. Pour toutes les contraintes, la complexité est donc en $2e \times O(d) = O(ed)$. Nous devons ajouter à cette complexité celle des valeurs mémorisées qui est trivialement en $O(ed)$ pour obtenir la complexité en espace d'AC-7 qui est donc en $O(ed)$.

5.2.6 Intérêt d'AC-7 par rapport à AC-6

Peut-on affirmer qu'AC-7 est meilleur qu'AC-6 car il effectue moins de tests de consistance? La réponse est évidente si ces tests sont coûteux, comme dans le cas d'un accès disque ou d'un calcul qui ne s'effectue pas en $O(1)$. Mais il ne faut pas croire que lorsque les tests se font très rapidement AC-7, devienne inintéressant. Nous allons montrer que le travail réalisé par AC-7 n'est pas équivalent à celui fait par AC-6, même si l'on considère qu'un test de consistance est équivalent en temps à n'importe quelle évaluation d'une expression conditionnelle du programme.

Des quatres propriétés qui sont maintenues par AC-7, deux ne le sont pas par AC-6. Ce sont les propriétés 2) et 3b):

2) ne jamais vérifier $C_{ij}(a, b)$ s'il existe c encore dans D_j tel que $C_{ji}(c, a)$ ait déjà été vérifié positivement.

3b) ne jamais vérifier $C_{ij}(a, b)$ si $C_{ji}(b, a)$ a déjà été vérifié.

Si l'on voit bien les conséquences de la propriété 3b), il n'en est pas de même pour la propriété 2). Cette dernière permet à AC-7 de gagner effectivement du temps par rapport à AC-6. Ainsi dans l'exemple de la figure 5.1, lorsque l'on recherche un support pour la valeur e sur la contrainte C_{ji}, AC-7 trouvera la valeur c beaucoup plus rapidement que AC-6 grâce à la propriété 2). En effet AC-7 utilise le fait que $C_{ij}(c, e)$ a déjà été vérifié positivement, pour trouver un support à e sur la contrainte C_{ji}, tandis qu'AC-6 vérifiera $C_{ji}(e, a)$ puis $C_{ji}(e, b)$ négativement avant de vérifier $C_{ji}(e, c)$ positivement.

D'autre part, cette propriété permet à AC-7 de réaliser moins de tests de consistance que AC-6, même si l'on compte les tests $C_{ij}(a,b)$ et $C_{ji}(b,a)$ pour un seul test. AC-7 peut éviter, par rapport à AC-6, la vérification de la consistance d'un couple (a,b) pour une contrainte C_{ij} prise dans n'importe quel sens, c'est-à-dire éviter aussi bien le test $C_{ij}(a,b)$ que le test $C_{ji}(b,a)$. Sur l'exemple précédent, AC-7 trouve très rapidement le support (i,b) pour (j,h) sur la contrainte C_{ji}, car (j,h) est le support de (i,b) sur C_{ij}, tandis qu'AC-6 trouvera ce support après avoir testé la consistance de $C_{ji}(h,a)$, puis de $C_{ji}(h,b)$. Le test $C_{ji}(h,a)$ n'a été réalisé dans aucun sens par AC-7.

Ce gain est réel même si l'on considère qu'un test de consistance est équivalent en temps à n'importe quelle évaluation d'une expression conditionnelle du programme. Pour mettre cela en évidence, étudions le comportement des algorithmes pour une contrainte d'égalité. Dans ce cas, les seuls couples admissibles sont ceux pour lesquels les deux valeurs sont égales.

Si les deux variables ont le même domaine de taille d, il faudra aux deux algorithmes $\sum_{i=1..d} i$ tests de consistance pour déterminer les valeurs viables de D_i. AC-7 utilisera en plus d tests pour vérifier que les listes des valeurs supportées sont vides. Par contre, pour déterminer les valeurs viables de D_j, AC-6 refera $\sum_{i=1..d} i$ tests de consistance tandis que AC-7 ne fera que d tests pour trouver les supports dans les listes des valeurs supportées. Le bilan, tous tests confondus, est donc largement en faveur de AC-7. Cet exemple nous permet d'établir la propriété suivante :

Propriété 8 *AC-7 peut éviter, par rapport à AC-6, $O(d^2)$ tests de consistance pour des domaines de taille d.*

Il est important de remarquer que nous avons établi cette propriété sans considérer la phase de propagation où d'autres économies peuvent être réalisées. Toutefois, nous devons nuancer ce résultat par le fait que la recherche systématique dans la liste des valeurs supportées et la mémorisation des dernières valeurs étudiées représentent des instructions supplémentaires par rapport à AC-6. Nous verrons dans la partie consacrée au comportement expérimental des algorithmes que cela ne pénalise pas beaucoup AC-7.

5.2.7 Quelques petites améliorations d'AC-7

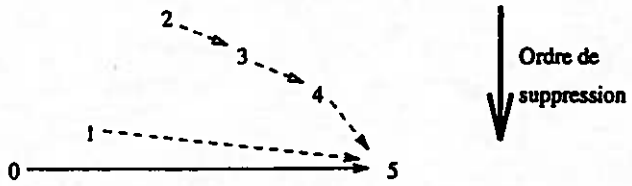
La fonction de recherche d'un suivant dans le domaine

La fonction SUIVANT(a, D_i) mérite une étude approfondie. Elle retourne la valeur de D_i immédiatement supérieure à a. La première manière pour obtenir ce résultat consiste à balayer successivement les valeurs supérieures à a dans le domaine initial tant qu'elles n'appartiennent pas au domaine courant. Le code trivial est :

$$a \leftarrow a+1 \textbf{ tant que } a \notin D_i \textbf{ faire } a \leftarrow a+1$$

Avec cette méthode on risque d'atteindre un grand nombre de valeurs du domaine initial qui n'appartiennent plus au domaine courant. Bessière [Bessière, 1994] propose une amélioration de cette recherche.

Le nouveau principe consiste à remplacer $a \leftarrow a+1$ par $a \leftarrow$ DERNIERSUIVANT(a, D_i), où DERNIERSUIVANT(a, D_i) est la valeur a' qui était égale à SUIVANT(a, D_i) quand a a été supprimée de D_i (si a n'est pas la dernière valeur du domaine). Si les domaines sont représentés avec une liste chaînée, ce qui est généralement le cas, alors on peut obtenir le résultat précédent en ne modifiant pas le chaînage des valeurs qui sont supprimées du domaine. Illustrons cela sur le domaine initial $D_i = \{0, 1, 2, 3, 4, 5\}$ auquel on a supprimé les valeurs 2 puis 3 puis 4 et enfin 1.



Si l'on recherche le suivant de 1 on trouve immédiatement 5, tandis que pour trouver la valeur 5 à partir de 2, on étudiera 3 puis 4. Cette méthode n'est pas coûteuse et est toujours meilleure que la précédente.

On peut l'améliorer en théorie en utilisant une technique inspirée de *l'union-find* de Tarjan [Tarjan, 1975]. Elle consiste à mettre à jour le chaînage lorsque l'accès n'est pas direct. Dans notre exemple, lorsque l'on cherche le suivant de 2 on parcourt le chemin 2,3,4,5. Comme on peut être amené à faire plusieurs fois cette requête, on risque de parcourir plusieurs fois ce chemin. L'idée consiste à chaîner toutes les valeurs du chemin hormis l'extrémité, avec la valeur finale du chemin. Si l'on obtient

un gain en théorie ce n'est malheureusement pas forcément vrai en pratique. En effet, si une seule requête pour les valeurs 2,3 et 4 est faite alors la mise à jour est inutile. C'est pourquoi, nous avons décidé de ne pas implémenter cette dernière idée.

Insertion ou ajout dans les listes de supports?

Le nombre d'opérations effectuées par l'algorithme n'est pas équivalent si l'on insère en tête ou ajoute en fin les nouvelles valeurs supportées dans les listes de supports. Cette différence apparaît dans la fonction EXISTESUPPORT. En effet, on recherche tout d'abord dans les listes de valeurs supportées. L'idéal serait de ne jamais rencontrer de valeurs qui n'appartiennent pas au domaine courant. On pourrait obtenir ce résultat en mémorisant pour chaque valeur ses justifications, c'est-à-dire les valeurs qui la font vivre sur chaque contrainte. Ainsi, lors de sa disparition, une valeur pourrait se déconnecter des listes auxquelles elle appartient. Malheureusement, en procédant ainsi on perd du temps par rapport à la méthode proposée car on réalise systématiquement, à chaque suppression de valeur, le travail éventuellement effectué lors de la recherche d'un nouveau support. En effet des listes de supports peuvent contenir des valeurs qui ne sont plus dans les domaines courant à la fin de l'algorithme de fermeture par consistance d'arc. De plus, si le réseau vérifie l'inconsistance d'arc, toute mise à jour non effectuée avant la détection de cette inconsistance se traduit par du temps gagné.

On peut éviter de rencontrer certaines valeurs qui n'appartiennent pas au domaine courant, ou plus exactement essayer de favoriser les cas pour lesquels les valeurs rencontrées appartiennent aux domaines courants. Intuitivement on s'aperçoit que plus une valeur est introduite tardivement dans une liste de supports par la phase de propagation, plus elle a de chance de ne pas être supprimée. Aussi l'insertion en tête est préférable à l'ajout en fin, car en insérant on a plus de chance de ne pas trouver de valeurs éliminées.

5.3 AC-Inférence

5.3.1 Déduction de supports

Nous illustrons le principe de déduction de supports avec un exemple simple de problème de coloration. Le problème consiste à assigner une couleur a (aquamarine), b (bleu), c (corail) à chacun des deux pays x_i et x_j tels qu'ils aient des couleurs différentes. Ce problème est trivial mais il va nous permettre d'illustrer notre propos. Le réseau modélisant ce problème a pour variables les deux pays et pour valeurs les couleurs. La contrainte entre deux variables spécifie que deux pays ne peuvent pas prendre la même couleur.

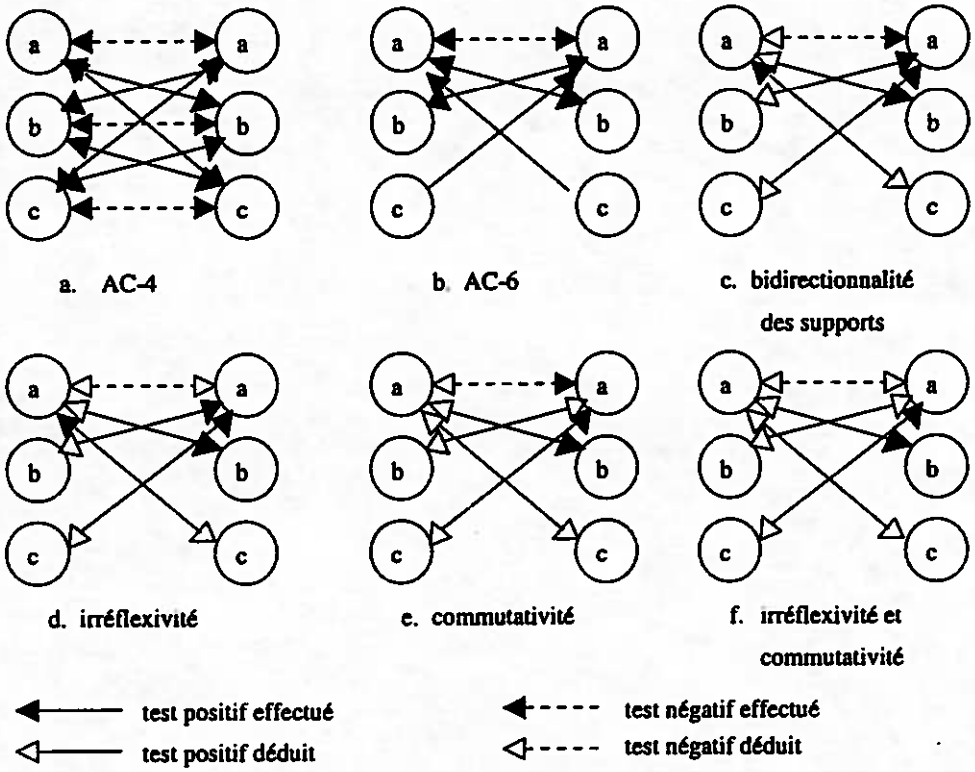


FIG. 5.9 - *Mise en évidence de l'intérêt de la déduction de supports.*

AC-4 vérifie chaque paire exactement deux fois, il fera donc d'après la figure 5.9a. 18 tests de consistance pour réaliser la fermeture par consistance d'arc. AC-6 qui a été dit comme étant optimal dans le sens où il effectue seulement les vérifications nécessaires réalise 8 tests de consistance (cf figure 5.9b.). Cependant, la déduction de support basée sur la bidirectionnalité peut réduire davantage le nombre de tests

réalisés. AC-6, AC-3 et AC-4 vérifient par exemple que (i, a) est supportée par (j, b) et séparément étudie la consistance de $C_{ji}(b, a)$. Ces algorithmes ne peuvent pas dire : «Oh je sais déjà que (j, b) est supportée, j'ai trouvé cela lorsque je cherchais un support pour (i, a)». L'algorithme AC-7 que nous avons proposé peut effectivement dire cela. Après avoir testé (i, a) avec (j, b) et trouvé que (j, b) supporte (i, a) ***il déduit*** que (i, a) supporte (j, b). Cette déduction est simplement basé sur le fait que les supports sont bidirectionnels. En utilisant de telles déductions AC-7 ne requiert seulement que 5 tests de consistance pour achever la consistance d'arc. Cela est montré en figure 5.9c. AC-7 utilise une connaissance particulière sur les contraintes. Il est possible de prendre en compte d'autres types de propriétés comme la réflexivité et la commutativité :

Définition 33 *Soit C_{ij} une contrainte binaire d'un réseau de contraintes :*

- *C_{ij} est **commutative** ssi $\forall a, b \in D_i \cap D_j : C_{ij}(a, b) \Leftrightarrow C_{ij}(b, a)$.*
- *C_{ij} est **anti-commutative** ssi $\forall a, b \in D_i \cap D_j : C_{ij}(a, b) \Leftrightarrow \neg C_{ij}(b, a)$.*
- *C_{ij} est **réflexive** ssi $\forall a \in D_i \cap D_j : C_{ij}(a, a)$ est vrai.*
- *C_{ij} est **anti-réflexive** ssi $\forall a \in D_i \cap D_j : C_{ij}(a, a)$ est faux.*

En tenant compte de l'anti-réflexivité des contraintes, le nombre de tests de consistance est réduit à 4 (cf figure 5.9d). En effet cette connaissance nous permet de déduire immédiatement que $C_{ij}(a, a)$ est faux, rendant par la même inutile ce test de consistance. Supposons que nous sachions que les contraintes sont aussi commutatives. Cela permet, encore une fois, de réduire le nombre de tests de consistance : il n'en faut plus que 3 (cf figure 5.9e). Par exemple après avoir tester $C_{ij}(a, b)$ nous pouvons déduire $C_{ij}(b, a)$. Si l'on combine les deux propriétés précédentes le nombre de tests est réduit à 2 (cf figure 5.9f).

5.3.2 Algorithme

Le fonctionnement d'AC-Inférence est semblable à celui d'AC-7. Le principe des phases d'initialisation et de propagation reste inchangé. Seule la fonction de recherche d'une nouveau support diffère par rapport à AC-7. AC-Inférence abandonne la notion de dernière valeur étudiée. En contrepartie il a besoin de deux nouvelles structures

de données :

- $P_{ia}[j] = \{(j,b)|C_{ij}(a,b)$ est connue comme étant vrai $\}$. C'est l'ensemble des valeurs de D_j dont on sait qu'elles supportent (i,a), après avoir déduit ce résultat.

- $U_{ia}[j] = \{(j,b)|C_{ij}(a,b)$ est inconnu $\}$. C'est l'ensemble des valeurs de D_j dont on ne sait pas si elles sont ou non des supports pour (i,a).

La fonction EXISTESUPPORT d'AC-Inférence est donnée par l'algorithme 16. Elle appelle la fonction DÉDUIRESUPPORT qui utilise les propriétés des contraintes pour déduire à partir du résultat d'un test de consistance explicitement réalisé le résultat d'autres tests. La recherche d'un nouveau support se fait en deux étapes. Tout d'abord, on recherche la présence d'un support dans la liste P. Si aucun support n'est trouvé, soit parce que la liste est vide, soit car tous les éléments n'appartiennent plus au domaine, on teste la consistance des valeurs contenues dans la liste U jusqu'à trouver un nouveau support. Supposons, par exemple, que l'on recherche un support pour la valeur (i,a) sur la contrainte C_{ij}. L'algorithme recherche une valeur de $P_{ia}[j]$ appartenant au domaine courant D_j puis, s'il n'en a pas trouvé, recherche dans $U_{ia}[j]$ la première valeur consistante. Pour chaque valeur b atteinte, l'algorithme étudie la consistance de $C_{ij}(a,b)$ et supprime cette valeur de $U_{ia}[j]$ et la valeur a de $U_{jb}[i]$. Ensuite, il déduit grâce aux propriétés de la contrainte, le résultat d'autres tests et, éventuellement, met à jour d'autres listes U et des listes P.

Il est important de noter que AC-Inférence n'est pas similaire à AC-4. Ce n'est pas une implémentation astucieuse de ce dernier algorithme. L'idée du calcul du premier support est présente dans AC-Inférence comme elle l'est dans AC-6 ou AC-7, alors qu'elle ne l'est pas dans AC-4. Un tel algorithme permet donc d'envisager le même type de gain par rapport à AC-4 que ceux réalisés par AC-6.

5.3.3 Problèmes liés à l'implémentation

Nous avons dit précédemment que pour chaque valeur b atteinte d'une liste $U_{ia}[j]$, l'algorithme étudie la consistance de $C_{ij}(a,b)$ et supprime cette valeur de $U_{ia}[j]$ et de $U_{jb}[i]$. Si la suppression de (j,b) de la liste $U_{ia}[j]$ ne pose pas de problème, il n'en est pas de même pour la suppression de (i,a) de $U_{jb}[i]$. En effet, comment peut-on

Algorithme 16 AC-Inférence : la fonction EXISTESUPPORT

```
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
    trouvé ← faux
    /* recherche dans la liste des supports connus */
    tant que ¬trouvé et P_ia[j] ≠ ∅ faire
        c ← PREMIER(P_ia[j])
        si c ∉ D_j alors SUPPRIMER(c, P_ia[j])
        sinon trouvé ← vrai
    /* recherche dans la liste des valeurs pouvant être des supports */
    tant que ¬trouvé et U_ia[j] ≠ ∅ faire
        c ← PREMIER(U_ia[j], D_j)
        SUPPRIMER(c, U_ia[j])
        si c ∈ D_j alors
            trouvé ← C_ij(a, c)
            DÉDUIRESUPPORT((i, a), (j, c), trouvé)
    si trouvé alors
        si b ≠ nil alors SUPPRIMER((i, a), S_jb)
        AJOUTER((i, a), S_jc[i])
    retourner trouvé
```

accéder à cette valeur? Pour parvenir à ce résultat il suffit simplement de relier entre eux les éléments des différentes listes. Ainsi on relie n'importe quelle valeur b de $U_{ia}[j]$ avec la valeur a de $U_{jb}[i]$. La figure 5.10 résume ce mécanisme.

Un autre problème se pose pour certaines propriétés des contraintes comme la commutativité : $C_{ij}(a,b) = C_{ij}(b,a)$: a et b sont ils des indices du domaine ou des valeurs?

Si a et b représentent des indices, alors si l'on dispose d'un accès direct aux valeurs du domaine il n'y aura pas de problème. En effet, nous pourrons très facilement accéder aux valeurs correspondant aux indices a et b de chacun des domaines.

Par contre si a et b sont des valeurs, le problème reste entier. En effet, comment pourrons nous, après avoir calculé $C_{ij}(a,b)$, accéder à la valeur a de x_j et à la valeur b de x_i?

Supposons, dans un premier temps, que les domaines soient égaux $D_i = D_j =$

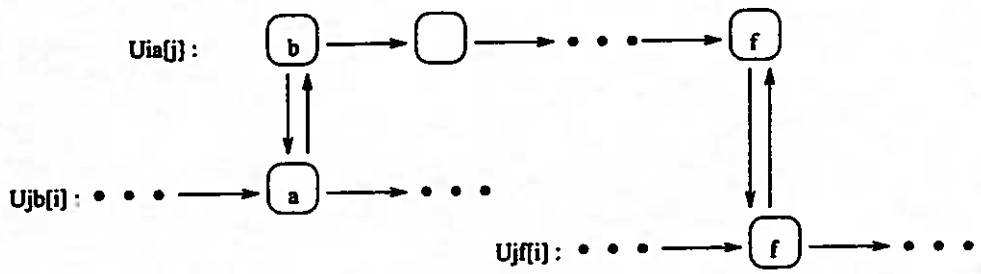


FIG. 5.10 - *Accès direct entre éléments de listes différentes.*

$\{12, 22, 45, 66\}$. Dans ce cas les indices attribuées aux valeurs sont les mêmes pour les deux domaines. Donc nous pouvons travailler uniquement sur les indices et utiliser un tableau attribuant à chaque indice une valeur réelle. Le problème est donc résolu.

Supposons, maintenant, que les domaines ne soient pas égaux, $D_i = \{12, 22, 45, 65\}$ et $D_j = \{22, 45, 65, 87\}$ par exemple. Les valeurs ne correspondent plus aux mêmes indices dans les deux domaines. A partir d'une valeur de D_i il devient difficile d'accéder à la même valeur dans D_j. Pour remédier à cela il est possible d'envisager deux solutions :

1. On utilise un seul domaine égal à l'union des deux domaines précédents afin qu'un indice corresponde toujours à la même valeur pour les deux domaines. Puis on supprime de chacun des domaines les valeurs qui n'étaient pas présentes à l'origine. Comme une suppression ne modifie pas les indices nous pourrons ainsi régler notre problème. Dans notre exemple nous obtiendrons comme correspondance entre les indices et les valeurs : $c(0) = 12; c(1) = 22; c(2) = 45; c(3) = 65; c(4) = 87$. Cependant, cette méthode présente un inconvénient majeur. Il est nécessaire pour déterminer le domaine d'une variable x_i de faire l'union de tous les domaines des variables contraignant x_i par une contrainte commutative. Cela revient très souvent à faire l'union de tous les domaines de toutes les variables du réseau. Pour certains réseaux cela peut devenir très gênant surtout si les domaines sont importants et ont peu de valeurs communes. Toutefois il faut réaliser, aussi, que la commutativité de valeurs n'est intéressante que si les deux valeurs sont présentes dans les deux domaines.

2. On utilise un mécanisme d'indirection qui exprime la correspondance entre les indices et les valeurs de chaque domaine. Il ne s'agit pas d'utiliser un tableau

indexé par les valeurs, mais plutôt d'attacher à chaque indice d'un domaine l'indice de l'autre domaine correspondant à la même valeur. Pour les deux domaines D_i et D_j exprimés ci-dessus nous obtiendrons :

$$c_{i\to j}(0) = nil; c_{i\to j}(1) = 0; c_{i\to j}(2) = 1; c_{i\to j}(3) = 2.$$
$$c_{j\to i}(0) = 1; c_{j\to i}(1) = 2; c_{j\to i}(2) = 3; c_{j\to i}(3) = nil.$$

Ce mécanisme doit être introduit sur chaque contrainte commutative.

5.4 Regroupement des contraintes identiques

Deux contraintes sont identiques si elles possèdent la même sémantique, c'est-à-dire la même déclaration en intention. On peut alors être tenté d'utiliser le travail déjà effectué pour rechercher des supports sur une contrainte, pour une autre contrainte équivalente. Considérons par exemple le problème des n-dames. Toutes les contraintes du type $C_{(i+j)i}$ avec j fixé sont identiques. Représentons une contrainte $C_{(i+1)i}$ par deux lignes de l'échiquier :

x_i	a	b	c	d	...
x_{i+1}		b			...

Si l'ordre des domaines est l'ordre lexicographique, alors, pour trouver un support à la valeur b de x_{i+1} sur la contrainte $C_{(i+1)i}$ il faudra étudier successivement la consistance de $C_{(i+1)i}(b,a), C_{(i+1)i}(b,b), C_{(i+1)i}(b,c)$ et $C_{(i+1)i}(b,d)$. Sur l'ensemble du réseau nous ferons le même type de tests pour n'importe quelle valeur de i afin de trouver un support aux valeurs b des domaines. L'identité entre contraintes permet d'éviter cette répétition en recherchant lorsque l'on essaie de trouver un support pour une valeur sur une contrainte, par exemple b sur $C_{(k+1)k}$, si une telle requête n'a pas déjà été faite pour une autre contrainte identique, par exemple pour $C_{(i+1)i}$. Bien sûr, il faut s'assurer que le support trouvé dans D_i est aussi présent dans D_k.

On retrouve très souvent des contraintes identiques dans des applications réelles. D'ailleurs, lors de la définition du problème, les contraintes sont souvent regroupées lorsqu'elles sont identiques.

Pour les problèmes d'affectation de fréquences les contraintes sont obtenues à

partir d'une des deux équations suivantes:

1. $|x_i - x_j| > k$, où k est une constante;

2. $|x_i - x_j| = k$, où k est une constante.

Comme le nombre de valeurs de k est limité, il y a donc de nombreuses contraintes identiques. Le tableau suivant présente le nombre de types différents de contraintes pour chaque problème. Le première ligne correspond au numéro du problème, la deuxième au nombre de contraintes et la troisième au nombre de contrainte de type différent.

1	2	3	4	5	6	7	8	11
5548	1235	2670	3967	2598	1322	2865	5744	4103
57	35	48	429	388	282	401	462	53

L'identité de contraintes permet donc d'introduire un nouveau mécanisme de déduction. Mais elles présentent aussi un autre avantage: l'économie de place en mémoire. En effet à quoi bon représenter autant de fois les listes $P_{i*}[j]$ et $U_{i*}[j]$ qu'il y a de contraintes identiques à C_{ij}? ***On peut se contenter de n'utiliser qu'une seule représentation pour un ensemble de contraintes identiques.*** Il faut alors munir chaque valeur pour chaque contrainte d'un pointeur vers le dernier élément étudié dans la liste P, car les domaines des variables contraintes ne sont pas tous égaux. On introduit donc une nouvelle structure de données dont la complexité en espace est en $O(ed)$. Par contre, le regroupement d'un ensemble de q variables identiques permet d'obtenir une représentation en mémoire des listes P et U en $O(d^2)$, au lieu de $O(qd^2)$.

5.5 Quelques heuristiques améliorant la consistance d'arc

Afin de ne pas surcharger inutilement notre exposé, nous illustrerons notre approche essentiellement à l'aide d'AC-7, car ce dernier n'est pas dépendant des propriétés des contraintes.

5.5.1 Ordonnancement des valeurs propagées

Wallace et Freuder ont testé, de manière expérimentale, de nombreux ordres pour les valeurs et les variables [Wallace and Freuder, 1992] pour l'algorithme AC-3. Nous ne pouvons malheureusement pas utiliser leurs résultats pour AC-6, AC-7 ou AC-Inférence car ces algorithmes sont trop différents d'AC-3. Si les tests de consistance sont peu coûteux, alors le calcul de l'ordre suivant lequel les valeurs seront propagées doit être fait rapidement. Par contre si ces tests prennent beaucoup de temps, on peut utiliser des heuristiques plus complexes. La qualité d'un ordre de propagation des valeurs varie en fonction de l'inconsistance d'un réseau. En effet, lorsque le réseau vérifie l'inconsistance d'arc, un bon ordre doit détecter cette inconsistance le plus rapidement possible. Ce qui n'est pas le but recherché si le réseau ne présente pas la même caractéristique. Le problème majeur est que cette inconsistance n'est pas connue a priori. On ne peut donc pas spécialiser les algorithmes en fonction de la consistance du réseau.

Nous ne proposons pas une heuristique mais une manière différente de voir la propagation donnant des résultats semblables aux résultats classiques et nous permettant d'écrire plus facilement certaines améliorations beaucoup plus fortes que nous proposerons dans la suite. Cette méthode consiste à propager les valeurs en fonction de leur variable. C'est-à-dire que l'on ne s'intéresse plus aux valeurs supprimées indépendemment les unes des autres mais que l'on propage toutes les valeurs en attente d'une variable avant d'étudier celles d'une autre variable. Pour obtenir ce résultat il suffit de munir chaque variable d'une liste d'attente et de transformer les listes d'attente de valeurs des phases d'initialisation et de propagation en listes d'attente de variables. C'est la fonction de suppression d'une valeur d'un domaine qui déterminera si la variable doit ou non être ajoutée à la liste des variables en attente. Nous noterons $listeAttente[i]$ la liste des valeurs en attente de la variable x_i. $listeAttente$ devient une liste de variables.

D'un point de vue général nous avons observé des variations inférieures à 0.01% par rapport à la méthode précédente.

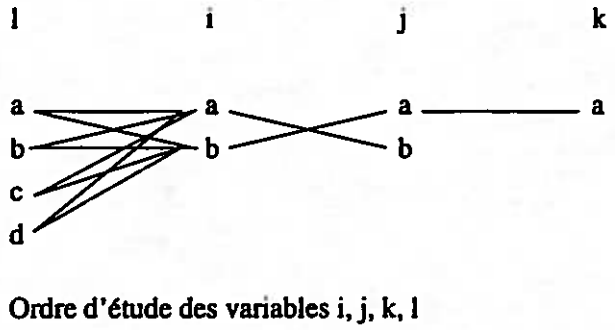


FIG. 5.11 - *Graphe de consistance du réseau de contraintes.*

5.5.2 Amélioration de la phase d'initialisation

Lors de la phase d'initialisation, on s'intéresse successivement à la viabilité des valeurs de chaque variable. Puis dans la phase de propagation, on étudie les conséquences de la suppression des valeurs non viables. Si toutes les valeurs sont viables après l'initialisation, alors la phase de propagation n'est pas appelée. Il est très difficile dans ce cas d'améliorer les algorithmes d'AC. D'autant plus que ce type de prétraitement est alors inutile puisqu'il ne supprime aucune valeur des domaines. Par contre, dès l'instant où certaines valeurs disparaissent, on peut s'intéresser à l'ordre suivant lequel les conséquences de leurs éliminations sont étudiées par AC-7.

Nous allons montrer à l'aide d'un exemple qu'il n'est pas toujours judicieux de séparer la phase d'initialisation de la phase de propagation.

Considérons le réseau de contraintes dont le graphe de consistance est donné par la figure 5.11.

La phase d'initialisation d'AC-7 réalise les tests de consistance donnés en figure 5.12.

Ensuite, AC-7 étudie les conséquences de la suppression de la valeur b de D_j. On obtient les résultats fournis par le tableau 5.13.

Or, on peut très bien envisager d'appeler la propagation dès qu'une valeur est supprimée dans l'initialisation. Le déroulement de l'algorithme est présenté par la figure 5.14.

On remarque que cette idée permet d'éviter les tests de consistance entre toutes les valeurs de x_l et la valeur (i, a). Grâce à elle, on pourra parfois, lors de la phase d'initialisation, ne pas trouver comme support une valeur qui sera supprimée par la

variables	contraintes	tests de consistance	résultats des tests	
i	C_{ij}	$C_{ij}(a,a)$	-	
		$C_{ij}(a,b)$	+	
		$C_{ij}(b,a)$	+	
	C_{il}	$C_{il}(a,a)$	+	
		$C_{il}(b,a)$	+	
j	C_{ji}			
	C_{jk}	$C_{jk}(a,a)$	+	
		$C_{jk}(b,a)$	-	b est supprimée de D_j
k	C_{kj}			
l	C_{li}	$C_{li}(a,a)$	+	
		$C_{li}(b,a)$	+	
		$C_{li}(c,a)$	+	
		$C_{li}(d,a)$	+	

FIG. 5.12 - *Phase d'initialisation d'AC-7.*

valeurs propagées	valeurs supportées	tests de consistance	
(j,b)	(i,a)		a est supprimée de D_i
(i,a)	(l,a)	$C_{li}(a,b)$	
	(l,b)	$C_{li}(b,b)$	
	(l,c)	$C_{li}(c,b)$	
	(l,d)	$C_{li}(d,b)$	

FIG. 5.13 - *Phase de propagation d'AC-7.*

Initialisation

variables	contraintes	tests de consistance	résultats des tests
i	C_{ij}	$C_{ij}(a,a)$	-
		$C_{ij}(a,b)$	+
		$C_{ij}(b,a)$	+
	C_{il}	$C_{il}(a,a)$	+
		$C_{il}(b,a)$	+
j	C_{ji}		
	C_{jk}	$C_{jk}(a,a)$	+
		$C_{jk}(b,a)$	-

b est supprimée de D_j

Propagation Immédiate

valeurs propagées	valeurs supportées	tests de consistance
(j,b)	(i,a)	
(i,a)	aucune	

a est supprimée de D_i

Retour à l'initialisation

variables	contraintes	tests de consistance	résultats des tests
k	C_{kj}		
l	C_{li}	$C_{li}(a,b)$	+
		$C_{li}(b,b)$	+
		$C_{li}(c,b)$	+
		$C_{li}(d,b)$	+

FIG. 5.14 - *AC-7 avec appel immédiat de la propagation dans l'initialisation.*

propagation.

Il est possible de donner une idée du gain réalisé par cette variante d'AC-7. Représentons par le symbole $<$ l'ordre suivant lequel les variables sont traitées par la procédure d'initialisation , $x_i < x_j$ signifiant que la viabilité des valeurs de x_i est étudiée avant celle des valeurs de x_j. Supposons qu'une valeur (j, a) ne soit pas viable, alors si l'on étudie immédiatement les conséquences de cette suppression, on évitera tous les éventuels tests de consistance entre les valeurs des variables x_k suivant x_j dans l'ordre $(x_k > x_j)$ et les valeurs des variables x_i précédant x_j dans l'ordre $(x_i < x_j)$ et qui sont supprimées par la phase de propagation issue de la disparition de (j, a). Ainsi, dans notre exemple, l'étude immédiate des conséquences de la suppression de (j, b) permet de ne pas tester la consistance entre les valeurs de x_l $(x_l > x_j)$ et la valeur a de x_i $(x_i < x_j)$ qui est éliminée par la phase de propagation.

Cependant, un problème se pose pour les valeurs de x_j et les valeurs des variables qui précèdent x_j dans l'ordre. Le moment où la propagation est appelée n'est pas indifférent. Si on l'introduit immédiatement après la suppression d'une valeur alors il se peut que l'on fasse plus de tests de consistance qu'avec la méthode classique qui traite de manière indépendante l'initialisation et la propagation. Cela se vérifie sur l'exemple de la figure 5.15.

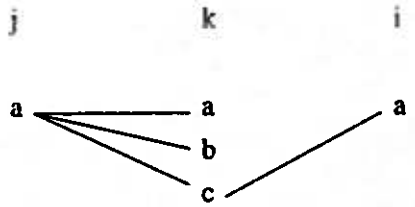


FIG. 5.15 - *Graphe de consistance.*

Considérons que les variables sont initialisées en suivant l'ordre x_i, x_j, x_k, et les contraintes en suivant l'ordre $C_{ik}, C_{jk}, C_{ki}, C_{kj}$. Pour trouver que la valeur a de x_i est viable, AC-7 étudie la consistance des couples $(a, a), (a, b), (a, c)$ sur la contrainte C_{ik}, puis il montre que (j, a) possède le support (k, a) sur la contrainte C_{jk}. AC-7 ne trouve pas de support pour (k, a) sur la contrainte C_{ki}, aussi il supprime cette valeur. Si l'on étudie immédiatement les conséquences de cette suppression, on va être amené à rechercher un nouveau support pour (j, a) sur la contrainte C_{jk}. Après avoir fait le test de consistance $C_{jk}(a, b)$, (k, b) est le nouveau support pour (j, a)

et la phase de propagation se termine. Or la phase d'initialisation qui reprend après l'étude des conséquences de la suppression de (k, a) élimine la valeur b de x_k car elle n'a pas de support sur C_{ki}. Cette suppression remet en cause la viabilité de (j, a). Un nouvel appel à la propagation lui trouvera le support (k, c). Par contre, si l'on emploie la phase de propagation après avoir étudié la viabilité de toutes les valeurs de x_j, on évitera le test de consistance $C_{jk}(a, b)$. Les expérimentations donnent raison à cette seconde approche, bien qu'elle ne soit pas systématiquement meilleure que la première. Le lecteur particulièrement intéressé pourra s'en convaincre avec le graphe de consistance de la figure 5.16, où le test de consistance $C_{ki}(c, a)$ n'est effectué dans aucun sens par la première méthode.

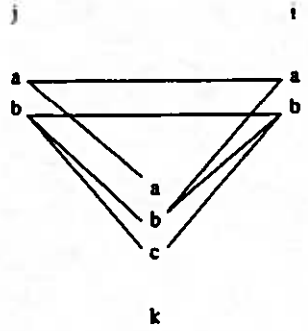


FIG. 5.16 - *Graphe de consistance.*

Cette heuristique ne garantit pas toujours un nombre de tests de consistance inférieur à ceux nécessaires à AC-7. En effet si le réseau vérifie l'inconsistance d'arc et si cette inconsistance est détectée lors de la phase d'initialisation par AC-7, il se peut que l'on fasse moins de tests de consistance avec la version classique d'AC-7. On trouve notamment ce cas lorsqu'aucune valeur d'une variable n'est consistante sur une contrainte. Dans le cas où le réseau ne vérifie pas l'inconsistance d'arc, l'heuristique présentée fait pratiquement toujours moins de tests.

La phase d'initialisation appelant immédiatement la phase de propagation est donnée par l'algorithme 17.

5.5.3 Amélioration de la phase de propagation

Lorsque le domaine d'une variable se réduit à une seule valeur pendant la propagation, on a intérêt à étudier différemment la conséquence de la suppression des valeurs

Algorithme 17 Propagation appelée par l'initialisation

```
INITIALISATIONV2() : booléen
    listeAttente ← ∅ /* liste d'attente de VARIABLES */
    pour chaque x_i ∈ X faire
        listeAttente[i] ← ∅
        pour chaque a ∈ D_i faire S_ia[j] ← ∅; L_ia[j] ← −1
    pour chaque x_i ∈ X faire
        pour chaque a ∈ D_i faire
            pour chaque x_j ∈ Γ(x_i) tant que a ∈ D_i faire
                si ¬ EXISTESUPPORT(x_i, a, C_ij) alors
                    SUPPRIMERETMETTREENATTENTE(a, D_i, listeAttente)
        si D_i = ∅ ou ¬ PROPAGATION(listeAttente) alors retourner faux
    retourner vrai
```

en attente de cette variable. En effet, seules les valeurs compatibles avec l'unique valeur restante de cette variable seront conservées; toutes les autres pourront être éliminées. Ce cas se rencontre forcément si le réseau de contraintes vérifie l'inconsistance d'arc. Il ne s'agit pas dans cette section de gagner des tests de consistance mais plutôt d'éviter certaines instructions afin de gagner du temps. Cependant, si on combine cette heuristique avec la précédente, on détectera plus rapidement l'inconsistance d'arc d'un réseau.

L'idée est de déterminer ces valeurs supprimées à partir de la variable x_i dont le domaine est réduit à un singleton, puis de déconnecter cette variable du réseau afin d'éviter d'étudier les contraintes dans le sens C_{*i}. La suppression d'une telle variable du graphe des contraintes est possible si l'on sait que son unique valeur est consistante avec toutes les valeurs des domaines de ses variables voisines, car si le réseau est inconsistant, cela ne peut pas provenir de cette variable. En effet, comme l'unique valeur supporte toutes les valeurs des variables voisines elle a donc elle-même toujours un support si ces domaines ne sont pas vides. Il faut donc étudier la consistance entre la seule valeur du domaine et les autres domaines, afin de supprimer les valeurs non compatibles, puis déconnecter la variable du réseau. Cette déconnexion est d'ailleurs équivalente à une instanciation forcée.

On peut aussi remarquer qu'il ne sert à rien de mettre à jour les listes de supports ou le tableau des dernières valeurs étudiées, pour AC-7, puisque la variable sera déconnectée du réseau.

Afin de simplifier notre exposé nous noterons x_s la variable dont le domaine est réduit à un singleton et u l'unique valeur de son domaine.

Pour mettre en oeuvre ce nouveau mécanisme, nous introduisons une seconde liste de variables en attente, notée *listeSingleton*. Nous considérerons que la fonction de suppression d'une valeur d'un domaine et de mise en attente de cette valeur détermine la liste à laquelle la variable doit appartenir, celle-ci pouvant en effet changer de liste si son domaine se réduit à un singleton. On obtient l'algorithme 18.

Algorithme 18 Étude des variables dont le domaine se réduit à un singleton

```
PROPAGATIONSINGLETON(i x_s, u i/o listeAttente, listeSingleton) : booléen
    pour chaque x_j ∈ Γ(x_s) faire
        VALEURSCOMPATIBLESAVECSINGLETON(C_sj, u, listeAttente, listeSingleton)
        si D_j = ∅ alors retourner faux
        SUPPRIMER(C_js)
    SUPPRIMER(x_s, listeSingleton)
    listeAttente[s] ← ∅
    retourner vrai
```

Le lecteur attentif remarquera que ce traitement n'est pas équivalent à une insertion des variables dans la liste d'attente.

Trois questions se posent :

1. Comment tester les valeurs compatibles avec l'unique valeur ?
2. Quand doit-on appeler la fonction PROPAGATIONSINGLETON ?
3. Comment éviter d'étudier les contraintes C_{*s} ?

Nous allons montrer qu'il existe deux réponses à la première question. Puis, nous verrons qu'il est particulièrement intéressant d'essayer de déclencher cette propagation particulière le plus tôt possible après sa détection. Enfin, nous montrerons qu'avec un test supplémentaire on peut répondre à la troisième question.

Calcul des valeurs compatibles avec l'unique valeur d'un domaine

Les algorithmes que nous allons présenter par la suite sont destinés à AC-7. Il est possible de les adapter pour AC-Inférence.

Pour commencer, nous donnons deux conditions nécessaires à l'existence d'au moins une valeur compatible avec u sur une contrainte C_{sj} :

Propriété 9

- *Si* $L_{su}[j] >$DERNIER(D_j) *alors le réseau est inconsistant.*
- *Si* $\forall a \in D_j : L_{ja}[s] > u$ *alors le réseau est inconsistant.*

Il est possible d'implémenter facilement (en $O(1)$) la première possibilité, par contre la seconde nécessite de parcourir l'ensemble des valeurs du domaine de D_j. C'est pourquoi nous avons choisi de ne pas nous en servir.

Deux approches sont possibles pour déterminer les valeurs non compatibles avec u: la première consiste à employer le même mécanisme que pour la propagation, tandis que la seconde procède directement aux vérifications.

Dans le premier cas, on recherche si les valeurs qui étaient supportées sur la contraintes par les valeurs en attente de x_s sont compatibles avec u. Cette compatibilité n'est pas faite en testant systématiquement la consistance entre u et les valeurs supportées. Pour garantir un nombre de tests de consistance optimal nous devons utiliser le tableau des dernières valeurs étudiées afin de savoir si la consistance entre u et les valeurs supportées n'a pas déjà été testée négativement. Il n'est pas possible qu'un test positif ait déjà été fait car a supporterait alors cette valeur. Cette dernière ne pourrait en aucun cas être présente dans une liste des valeurs supportées pour une autre valeur de x_s, c'est à dire par exemple pour une valeur de la liste d'attente de x_s, puisqu'une valeur appartient à seulement une seule de ces listes pour une contrainte donnée. On obtient la fonction ETUDIESINGLETONPARPROPAGATION (cf algorithme 19).

Nous attirons l'attention du lecteur sur le fait que cette fonction ne peut pas être appelée si les contraintes C_{sj} et C_{js} n'ont pas été étudiées par la phase d'initialisation d'AC-7, car toutes les valeurs de D_j non supportées par u doivent pouvoir être atteintes par la propagation des valeurs en attente de x_s.

Algorithme 19 AC-7 : calcul des valeurs compatibles avec l'unique valeur d'un domaine en procédant par propagation

```
ÉTUDIESINGLETONPARPROPAGATION(i C_sj, u io listeAttente, listeSingleton)
  pour chaque w ∈listeAttente[s] faire
    pour chaque a ∈ S_sw[j] faire
      si a ∈ D_j alors
        si L_ja[s] > u ou L_su[j] > a alors
          SUPPRIMERETMETTREENATTENTE(a, D_j, listeAttente, listeSingleton)
        sinon
          si ¬C_sj(u, a) alors
            SUPPRIMERETMETTREENATTENTE(a, D_j, listeAttente, listeSingleton)
```

Cette méthode peut atteindre des valeurs qui n'appartiennent plus au domaine courant D_j. Pour nous en convaincre étudions la contrainte C_{sj} de la figure 5.17.

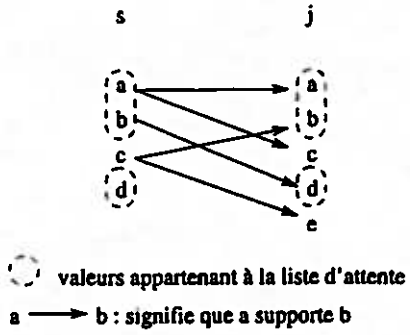


FIG. 5.17 - *AC-7: étude particulière d'une variable dont le domaine se réduit à un singleton.*

Dans ce cas nous atteindrons les valeurs a et d de x_j qui n'appartiennent plus à D_j. Si l'unique valeur du domaine ne supporte aucune valeur, ce cas a beaucoup de chance de se rencontrer en pratique. C'est pourquoi nous proposons une seconde approche qui procède directement aux vérifications entre u et les autres valeurs du domaine sans utiliser les listes des valeurs supportées. Nous n'utiliserons pas non plus la liste des valeurs que u supporte, car on risquerait d'atteindre aussi des valeurs n'appartenant plus au domaine, comme la valeur b de l'exemple précédent. De plus,

on peut savoir si une valeur a d'une variable x_j appartient à $S_{su}[j]$ en utilisant le tableau des dernières valeurs étudiées. Deux cas sont possibles si $a \in S_{su}[j]$: soit $L_{su}[j] = a$, soit $L_{ja}[s] = u$. La vérification de ces possibilités nous permettra de ne pas refaire des tests de consistance positifs. Pour éviter une revérification négative on utilise les tableaux de la même façon que précédemment. Cela nous assurera donc toujours un algorithme optimal pour un ordre donné. Nous proposons sous le nom ÉTUDIESINGLETONPARVÉRIFICATION le nouvel algorithme.

Algorithme 20 AC-7 : calcul des valeurs compatibles avec l'unique valeur d'un domaine en procédant par vérification

```
ÉTUDIESINGLETONPARVÉRIFICATION(i C_sj, u io listeAttente, listeSingleton)
  pour chaque a ∈ D_j faire
    si L_su[j] > a ou L_ja[i] > u alors
      SUPPRIMERETMETTREENATTENTE(a, D_j, listeAttente, listeSingleton)
    sinon
      si L_su[j] ≠ a et L_ja[i] ≠ u alors
        si ¬C_sj(u, a) alors
          SUPPRIMERETMETTREENATTENTE(a, D_j, listeAttente, listeSingleton)
```

Cette étude particulière, pour une variable dont le domaine est réduit à un singleton, ne modifie pas la complexité d'AC-7 dans le pire des cas, car on ne peut faire qu'une seule fois un tel traitement pour une variable, la variable étant ensuite déconnectée du réseau. Cela ne peut multiplier l'accès aux tableaux des dernières valeurs étudiées que par un facteur deux.

Il est important aussi d'essayer de déterminer les conditions pour lesquelles l'un ou l'autre des deux traitements proposés est meilleur. Il est évident que si la liste des valeurs supportées par u est vide, alors il est préférable d'employer le traitement par vérification. Par contre, si u supporte de nombreuses valeurs ou bien si une variable voisine a peu de valeur en attente, on a plutôt intérêt à utiliser la version faisant appel à la propagation.

Appel du traitement particulier pour les singletons

Comme le but des traitements que nous avons présentés est de déconnecter une variable du réseau afin d'éviter d'étudier les contraintes contraignant cette variable, il est intéressant de pouvoir le faire le plus tôt possible. C'est-à-dire dès que le domaine d'une variable se réduit à un singleton. Pour réaliser cela, on peut imaginer d'utiliser une liste d'attente particulière pour ces variables et de commencer par propager les variables présentes dans cette liste avant celle de l'autre liste d'attente. C'est la fonction de suppression d'une valeur d'un domaine qui déterminera à quelle liste la variable doit appartenir. Comme nous le verrons un peu plus loin, et surtout dans le chapitre suivant qui emploie systématiquement le filtrage par consistance d'arc lors de la recherche, cette méthode donne de bons résultats en pratique. Cependant, comme l'ordre de propagation des valeurs est changé par rapport à la version de base d'AC-7, on ne peut plus garantir un nombre de tests toujours inférieur par rapport à cette dernière. Dans le cas où le réseau vérifie l'inconsistance d'arc, elle permet très souvent de faire moins de tests car cette inconsistance est détectée plus rapidement qu'avec une méthode aveugle.

Suppression des contraintes C_{*s}

Après l'emploi de la fonction PROPAGATIONSINGLETON toutes les valeurs des domaines des variables voisines de x_s dans le graphe des contraintes sont compatibles avec u. Lorsqu'un des supports de u disparaît, il ne sert à rien de rechercher un nouveau support pour u car, soit le domaine auquel appartenait ce support devient vide et le réseau est inconsistant, soit ce domaine contient d'autres valeurs qui sont des supports pour u. Pour éviter ces recherches inutiles, il faut d'une part déconnecter la variable x_s du graphe des contraintes, et d'autre part, modifier la phase de propagation en ne vérifiant plus seulement si une valeur supportée appartient encore au domaine, mais aussi si ce dernier n'est pas un singleton. Ainsi, dès lors que l'on atteint une sous-liste $S_{ia}[j]$, on regarde si D_j n'est pas un singleton, et, si c'est le cas, on passe immédiatement à la sous-liste suivante. Nous pouvons maintenant donner un nouvel algorithme pour la phase de propagation (cf algorithme 21).

Algorithme 21 Nouvelle propagation

```
PROPAGATIONV2(io listeAttente, listeSingleton) : booléen
  tant que listeSingleton ≠ ∅ et listeAttente ≠ ∅ faire
    si listeSingleton ≠ ∅ alors
      prendre x_s dans listeSingleton et le supprimer
      u ← PREMIER(D_s)
      si ¬ PROPAGATIONSINGLETON(x_s, u, listeAttente, listeSingleton) alors
        retourner faux
    sinon
      prendre x_j dans listeAttente et le supprimer
      tant que listeAttente[j] ≠ ∅ faire
        prendre b dans listeAttente[j] et le supprimer
        pour chaque x_i ∈ Γ(x_j) tel que S_jb[i] ≠ ∅ faire
          si |D_i| > 1 alors
            pour chaque (i, a) ∈ S_jb[i] tant que |D_i| > 1 faire
              si a ∈ D_i alors
                si ¬ EXISTESUPPORT(x_i, a, C_ij) alors
                  SUPPRIMERETMETTREENATTENTE(a, D_i, listeAttente, listeS
  retourner vrai
```

Intérêt de ce traitement pour les contraintes de différences

Comme nous l'avons vu en 4.4.2, Mohr et Masini ont remarqué que pour les contraintes de différences (cf définition 32) l'absence de support ne peut se produire que dans le cas où un domaine se réduit à un singleton. Grâce au traitement particulier que nous venons de proposer, nous pouvons donc donner une méthode simple et efficace pour calculer la fermeture par consistance d'arc avec ces contraintes. Elle consiste à prendre en compte les contraintes de différence ***uniquement*** lors de l'étude d'un singleton. Pour parvenir à ce résultat on munit chaque variable de la liste *listeContraintesDiff* des contraintes de différence qui la contraignent. Quand le domaine d'une variable x_s se réduit à une seule valeur u, on supprime u de tous les domaines

Algorithme 22 Étude des variables dont le domaine se réduit à un singleton avec traitement particulier pour les contraintes de différence

```
PROPAGATIONSINGLETON(i x_s, u i/o listeAttente, listeSingleton)
  pour chaque x_j ∈ Γ(x_s) faire
  |  VALEURSCOMPATIBLESAVECSINGLETON(C_sj, u, listeAttente, listeSingleton)
  |  si D_j = ∅ alors retourner faux
  |  SUPPRIMER(C_js)
  pour chaque C_js ∈ listeContraintesDiff[s] faire
  |  si u ∈ D_j alors
  |  |  SUPPRIMERETMETTREENATTENTE(u, D_j, listeAttente, listeSingleton)
  |  |  si D_j = ∅ alors retourner faux
  |  |  SUPPRIMER(C_sj, listeContraintesDiff[j])
  SUPPRIMER(x_s, listeSingleton)
  listeAttente[s] ← ∅
  SUPPRIMER(x_s, X)
  retourner vrai
```

des variables qui présentent une contrainte de différence entre elles et x_s. La nouvelle fonction PROPAGATIONSINGLETON est fournie par l'algorithme 22.

5.6 Expérimentations

Nous avons réalisé trois types d'expérimentation pour mettre en évidence l'intérêt du mécanisme de déduction, des heuristiques et de la combinaison des heuristiques et du mécanisme de déduction.

Les différents algorithmes ont été comparés selon le modèle expérimental que nous avons défini dans le chapitre précédent. Nous donnerons, en général, trois types de résultats. Le premier donne les tests de consistance réalisés par les algorithmes, le second le nombre d'appel à la fonction EXISTESUPPORT et enfin le dernier type concerne le temps.

Nous utiliserons toujours AC-4 et AC-6 comme référence. et nous noterons AC-I, AC-Inférence avec la bidirectionnalité, la commutativité et la réflexivité et AC-Iid

la version précédente d'AC-Inférence qui regroupe les contraintes identiques. Nous ajouterons au nom des algorithmes «p» s'ils appellent la propagation dans l'initialisation et «ps» s'ils utilisent en plus le traitement particulier pour les variables dont le domaine est réduit à un singleton.

Nous avons utilisé des réseaux aléatoires ayant des caractéristiques très différentes. Ainsi, nous proposons trois types de problèmes: un pour les réseaux ayant plus de variables que de valeurs par domaine (nous avons choisit 60 variables et 30 valeurs par variables), un pour les réseaux ayant autant de variables que de valeurs par domaine (dans notre cas il y aura 45 variables), et enfin un troisième type pour les réseaux qui ont moins de variables que de valeurs par domaine (30 variables et 60 valeurs). Après de nombreux essais, nous avons choisi une densité de 60% pour le premier et le troisième type et une densité de 30% pour le second. Ces choix n'ont pas pour but de masquer certains résultats, mais plutôt d'éviter d'avoir une majorité de cas pour lesquels soit le réseau vérifie inconsistance d'arc soit aucune valeur n'est supprimée d'un domaine. Les bornes entre lesquelles le taux de satisfaisabilité des contraintes évolue ont été ajustées avec le même esprit. Une flèche verticale sur les figures indique la limite de l'inconsistance d'arc des réseaux. Chaque résultat donné pour un problème aléatoire correspond à la moyenne des résultats obtenus pour 30 problèmes ayant les mêmes paramètres.

Pour les problèmes d'affectation de fréquences, le nombre de valeurs supprimées par une fermeture par consistance d'arc est résumée dans le tableau suivant:

1	2	3	4	5	6	7	8	11
0	0	0	13868	12046	toutes	toutes	toutes	0

Les problèmes 1, 2, 3 et 11 vérifient donc la consistance d'arc et les problèmes 6, 7 et 8 vérifient l'inconsistance d'arc.

5.6.1 Intérêt de la déduction de supports

Intéressons nous tout d'abord aux problèmes d'affectation de fréquences (figure 5.21 et 5.22).

Il apparaît clairement qu'AC-7, AC-I et AC-Iid réalisent nettement moins de tests de consistance qu'AC-6. La bidirectionnalité économise plus de 20% des tests, l'introduction d'autres propriétés porte ce chiffre à plus de 50%. On remarque aussi qu'il y

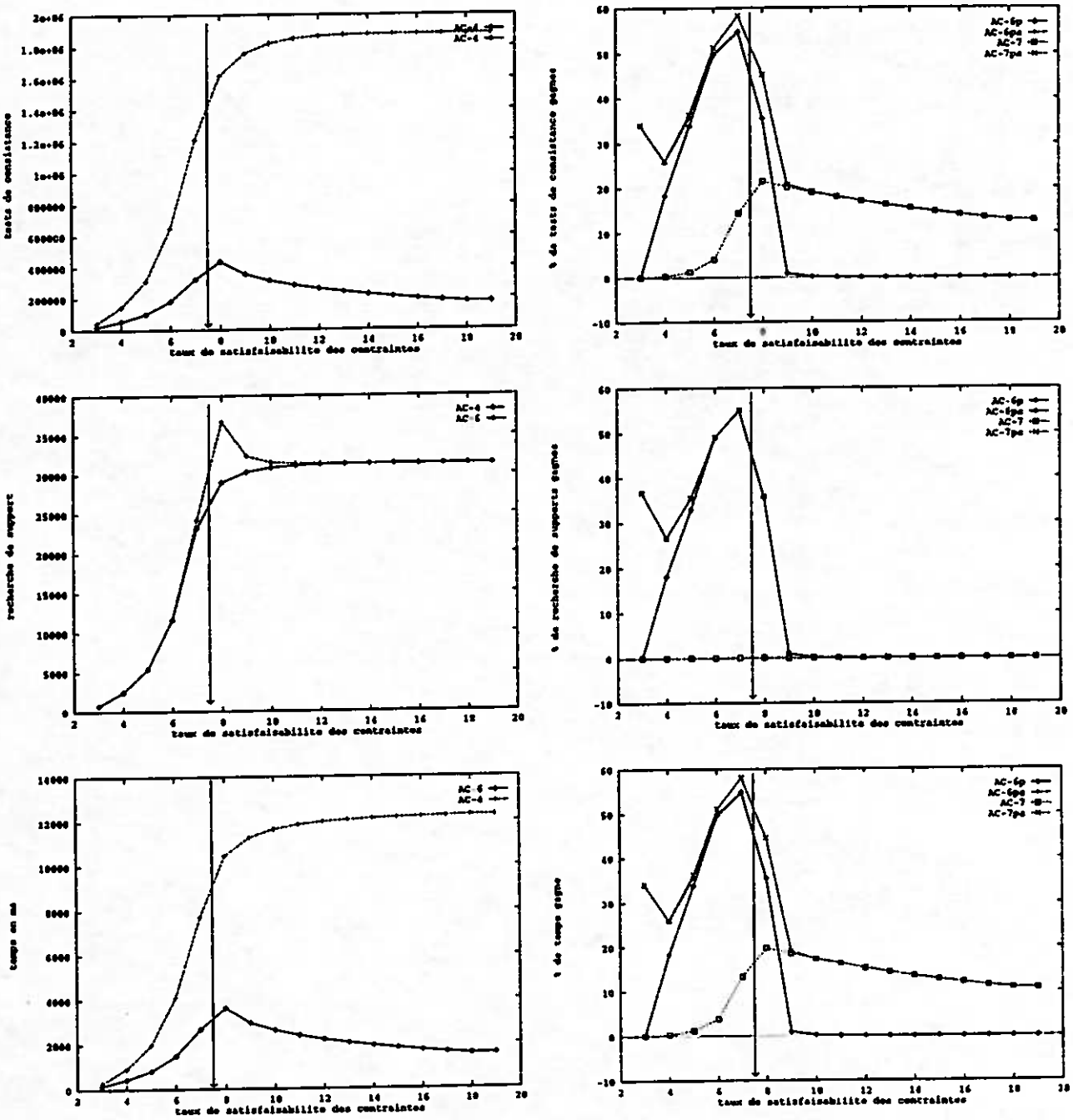


FIG. 5.18 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6p, AC-6ps, AC-7 et AC-7ps pour des réseaux aléatoires ayant 30 variables 60 valeurs et une densité de contraintes égale à 60%.*

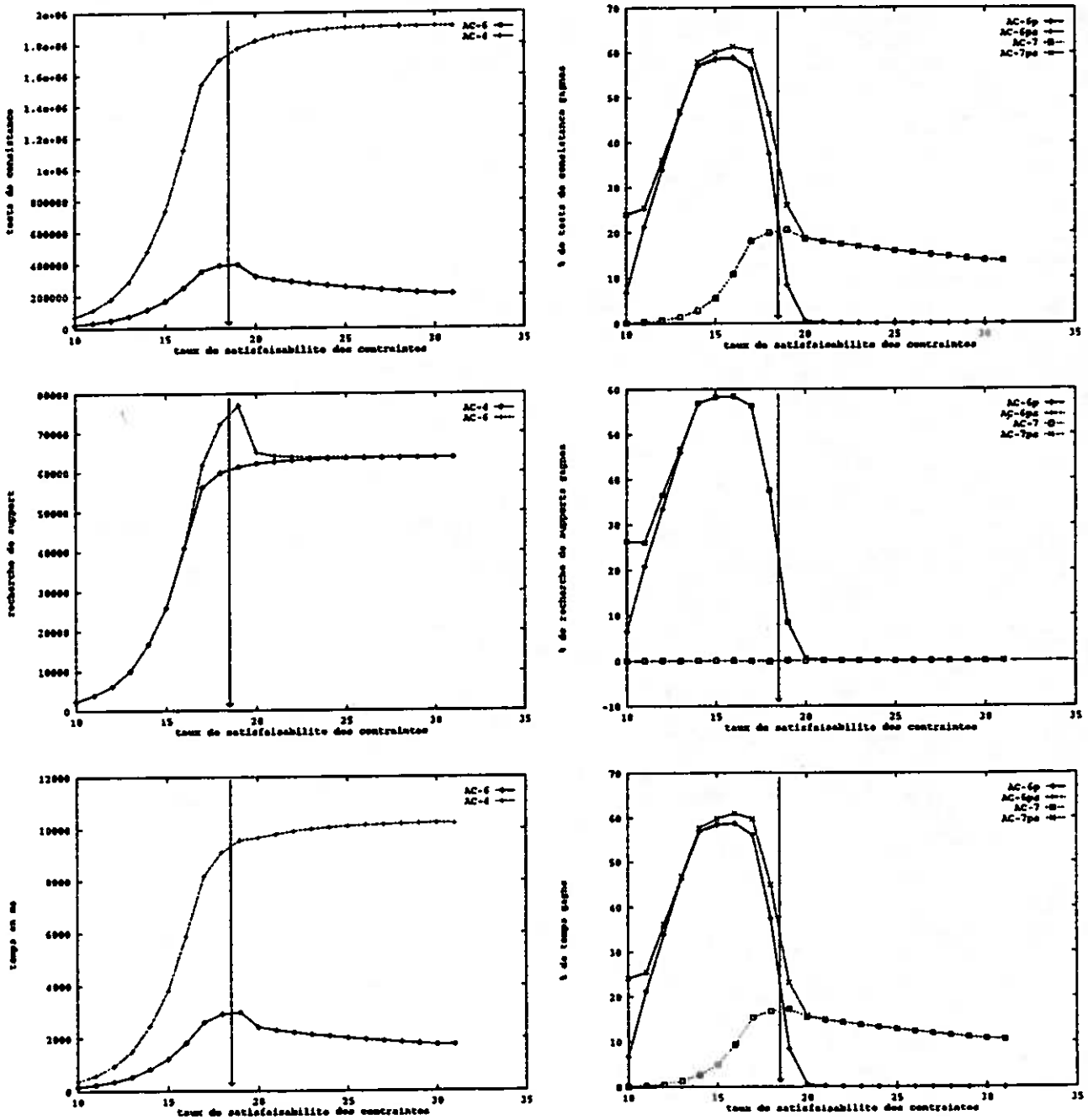


FIG. 5.19 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6p, AC-6ps, AC-7 et AC-7 pour des réseaux aléatoires ayant 60 variables 30 valeurs et une densité de contraintes égale à 60%.*

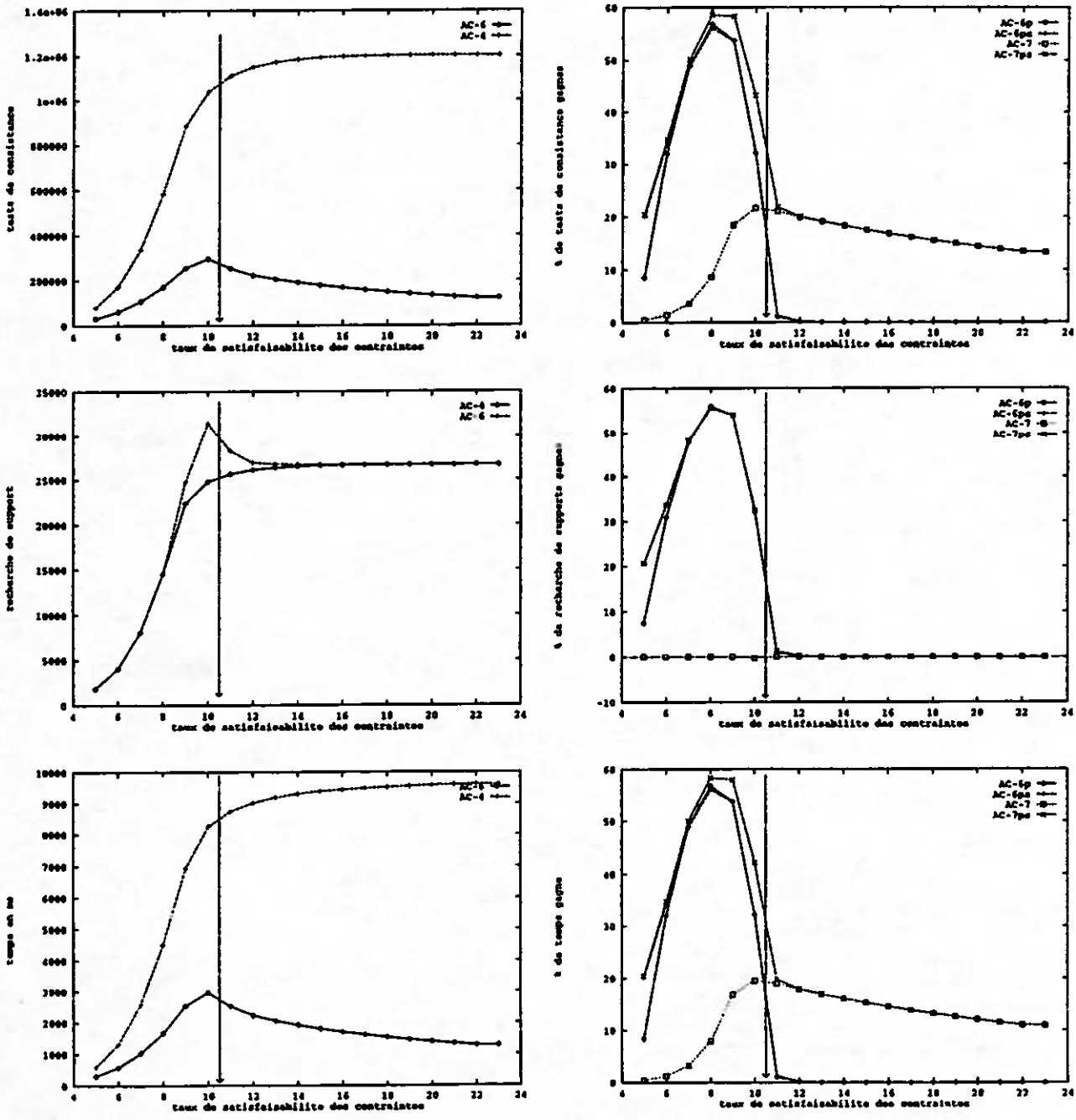


FIG. 5.20 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6p, AC-6ps, AC-7 et AC-7ps pour des réseaux aléatoires ayant 45 variables 45 valeurs et une densité de contraintes égale à 30%.*

	RLFAP-1		RLFAP-2		RLFAP-3		RLFAP-4			RLFAP-5		
	cc	tps	cc	tps	cc	tps	cc	rs	tps	cc	rs	tps
4	16 996	47,8	4 074	11,0	8 922	24,1	1 822	252	4,7	5 610	1 121	16,1
6	1 294	7,8	299	1,7	615	3,5	490	89	2,2	1 237	213	5,4
7	849	7,0	197	1,5	413	3,2	432	89	2,2	968	215	5,2
I	447	5,2	101	1,5	214	2,4	287	89	1,9	607	215	4,3
Iid	4	1,5	3	0,4	4	0,8	61	70	0,7	133	196	1,7

	RLFAP-6			RLFAP-7			RLFAP-8			RLFAP-11	
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	tps
4	3 310	762	9,7	6 508	252	15,9	12 925	552	32,7	13 051	35,9
6	694	124	3,1	1 261	186	5,4	2 512	372	11,4	972	5,5
7	516	124	2,8	963	186	5,1	1 859	372	10,5	639	5,0
I	317	127	2,3	584	186	4,0	1 114	373	8,6	331	3,7
Iid	90	118	1,0	110	171	1,3	146	345	2,2	4	1,1

FIG. 5.21 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-7, AC-I et AC-Iid pour les problèmes d'affectation de fréquences. Le nombre de tests de consistance réalisés (cc) et le nombre d'appels à la fonction* EXISTESUPPORT *(rs) sont exprimés en milliers, les temps (tps) sont donnés en seconde.*

	RLFAP-1		RLFAP-2		RLFAP-3		RLFAP-4			RLFAP-5		
	cc	tps	cc	tps	cc	tps	cc	rs	tps	cc	rs	tps
7	34,4	10,3	34,1	11,8	32,9	8,6	11,0	0,0	0,0	21,8	-1,0	3,7
I	65,5	33,3	66,1	11,8	65,2	31,4	41,4	0,0	13,6	51,0	-1,0	20,4
Iid	99,7	80,8	99,1	76,5	99,4	77,1	87,6	21,3	68,2	89,2	8,0	68,5

	RLFAP-6			RLFAP-7			RLFAP-8			RLFAP-11	
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	tps
7	25,7	0,0	9,7	23,7	0,0	5,6	26,0	0,0	7,9	34,3	9,1
I	54,3	-2,4	25,8	53,7	0,0	25,9	55,6	-0,3	24,6	66,0	32,7
Iid	87,0	4,8	67,7	91,2	8,1	75,9	94,2	7,3	80,7	99,6	80,0

FIG. 5.22 - *Pourcentages de gain réalisés par les algorithmes AC-7, AC-I et AC-Iid par rapport à AC-6 pour les problèmes d'affectation de fréquences.*

	RLFAP-4			RLFAP-5			RLFAP-6			RLFAP-7			RLFAP-8		
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps
4	1 822	252	4,7	5 610	1 121	16,1	3 310	762	9,7	6 508	252	15,9	12 925	552	32,7
6	490	89	2,2	1 237	213	5,4	694	124	3,1	1 261	186	5,4	2 512	372	11,4
6p	180	42	0,8	1 220	268	5,6	673	158	3,1	537	101	2,4	886	186	4,2
6ps	143	21	0,6	1 025	176	4,4	611	107	2,6	503	75	2,1	431	70	1,9

FIG. 5.23 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6p et AC-6ps pour les problèmes d'affectation de fréquences. Le nombre de tests de consistance réalisés (cc) et le nombre d'appels à la fonction* EXISTESUPPORT *(rs) sont exprimés en milliers, les temps (tps) sont donnés en seconde.*

a une forte corrélation entre le nombre de tests de consistance et le temps mis par les algorithmes. Toutefois on ne retrouve pas le même pourcentage de gain. AC-7 gagne environ 10% et AC-I 25%.

En ce qui concerne le nombre d'appels à la fonction EXISTESUPPORT, il n'y a pratiquement pas de différence entre AC-6, AC-7 et AC-I. Par contre on observe une diminution de ce chiffre lorsque l'on regroupe les contraintes identiques. Nous pensons que l'on peut expliquer ce phénomène par le fait qu'il y a de grandes chances pour que de nombreuses valeurs partagent le même support. De plus, si ce support vient à disparaître, alors il est fort possible qu'un seul calcul soit nécessaire pour leur redonner à toutes le même support.

Pour les problèmes aléatoires (cf figure 5.18, 5.19 et 5.20), nous ne pouvons comparer qu'AC-4, AC-6 et AC-7. AC-4 est très largement battu par AC-6. Comme précédemment le nombre de tests de consistance et le temps sont fortement liés. AC-7 améliore AC-6 d'environ 10 à 15 %. Les résultats sont très semblables pour les 3 types de réseaux, pourtant différents, que nous avons testés.

5.6.2 Intérêt des heuristiques

L'introduction de la propagation immédiate et du traitement des variables dont le domaine est un singleton n'est intéressante que si le réseau ne vérifie pas la consistance d'arc. Aussi nous ne considérerons que les problèmes d'assignement de fréquence 4 à 8

	RLFAP-4			RLFAP-5			RLFAP-6			RLFAP-7			RLFAP-8		
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps
6p	63,2	52,8	63,6	1,4	-25,8	-3,7	3,0	-27,4	0,0	57,4	45,7	55,6	64,7	50,0	63,2
6ps	70,8	76,4	72,7	14,7	17,4	18,5	12,0	13,7	16,1	60,1	59,7	61,1	82,8	81,2	83,4

FIG. 5.24 - *Pourcentages de gain réalisés par les algorithmes AC-6p, AC-6ps par rapport à AC-6 pour les problèmes d'affectation de fréquences.*

	tps (en s)	rs (en millier)	% gain tps	% gain rs
AC-4	68,4	1 425	-	-
AC-6	47,7	1 000	30,3	29,8
AC-6p	45,6	952	4,4	4,8
AC-6ps	44,2	948	7,4	5,2

FIG. 5.25 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6p et AC-6ps pour* lso-ssgrpartiel. *Les gains d'AC-6 sont exprimés par rapport à AC-4, ceux d'AC-6 et d'AC-6ps par rapport à AC-6.*

(cf figure 5.23 et 5.24). Pour ces derniers on observe qu'AC-6p fait toujours moins de tests de consistance qu'AC-6. Par contre il se produit parfois (RLFAP-5 et RLFAP-6) une augmentation du nombre d'appel de la fonction EXISTESUPPORT. Dans ce cas il n'y a aucun gain de temps. Pour les autres instances le gain en temps réalisé par AC-6ps par rapport à AC-6 est du même ordre de grandeur que celui du nombre de tests de consistance, c'est-à-dire autour de 60%.

Pour les réseaux aléatoires on observe qu'AC-6p gagne par rapport à AC-6 entre 20 et 60% pour tous les réseaux vérifiant l'inconsistance d'arc et pour tous les critères mesurés.

Malheureusement, nous ne retrouvons pas ces chiffres pour lso-ssgrpartiel (cf figure 5.25). Cependant, on remarque que le nombre d'appels de la fonction EXISTESUPPORT pour AC-6p et AC-6ps est plus faible que pour AC-6. Les gains en temps par rapport à AC-6 sont relativement faibles puisqu'AC-6p économise 4,4% du temps total et AC-6ps 7,4%.

Nous proposons en figure 5.26 une étude comparative des performances d'AC-6p et d'AC-6ps par rapport à AC-6, en fonction du nombre de valeurs supprimées. À

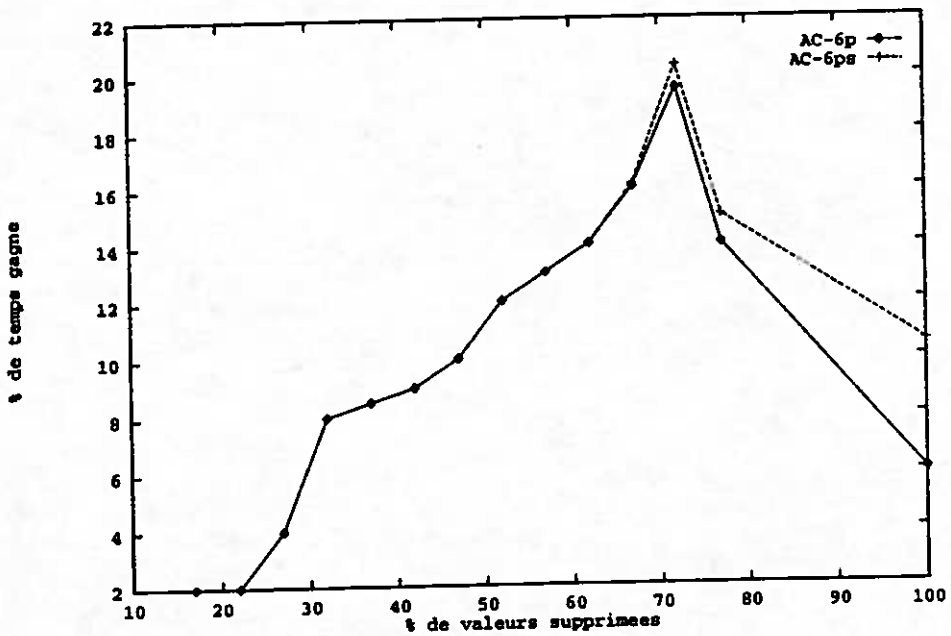


FIG. 5.26 - *Comparaison des performances des algorithmes AC-6, AC-6p, et AC-6ps pour* lso-ssgrpartiel *en fonction du nombre de valeurs supprimées par la fermeture par consistance d'arc.*

	RLFAP-4			RLFAP-5			RLFAP-6			RLFAP-7			RLFAP-8	
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs
4	1 822	252	4,7	5 610	1 121	16,1	3 310	762	9,7	6 508	252	15,9	12 925	552
6	490	89	2,2	1 237	213	5,4	694	124	3,1	1 261	186	5,4	2 512	372
6ps	143	21	0,6	1 055	176	4,4	611	107	2,6	503	75	2,1	431	70
7ps	128	21	0,6	862	176	4,4	474	107	2,5	443	75	2,2	369	71
Iidps	39	17	0,4	118	154	1,5	85	103	1,0	80	67	0,8	85	63

FIG. 5.27 - *Comparaison des performances des algorithmes AC-4, AC-6, AC-6ps, AC-7ps et AC-Iidps pour les problèmes d'affectation de fréquences. Le nombre de tests de consistance réalisés (cc) et le nombre d'appels à la fonction* EXISTESUPPORT *(rs) sont exprimés en milliers, les temps (tps) sont donnés en seconde.*

partir de 20% de valeurs supprimées, AC-6p est intéressant. Il arrive à gagner 20% du temps si 75% des valeurs sont supprimées. C'est d'ailleurs à partir de cette valeur qu'AC-6ps se détache d'AC-6p. On obtient pour ce dernier plus de 10% de gain pour les réseaux vérifiant l'inconsistance d'arc.

Pour tous les réseaux aléatoires que nous avons testés le pourcentage de gain pour tous les critères atteint 60% lorsqu'il y a inconsistance d'arc. La propagation dans l'initialisation est donc particulièrement intéressante pour ce type de réseaux. Notons que la seconde heuristique permet de ne pas descendre en dessous du seuil de 20% de gain.

5.6.3 Combinaison du mécanisme de déduction et des heuristiques

On observe pour les problèmes d'affectation de fréquences (cf figure 5.27 et 5.28) et pour les réseaux aléatoires que l'introduction du mécanisme de déduction et des heuristiques se combinent parfaitement bien.

Les gains mesurés pour tous les critères sont toujours meilleurs que ceux que nous avons précédemment observés. Les deux types d'améliorations sont donc complémentaires.

Pour terminer notre étude expérimentale, nous proposons de comparer AC-6 avec

	RLFAP-4			RLFAP-5			RLFAP-6			RLFAP-7			RLFAP-8		
	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps	cc	rs	tps
6ps	70,8	76,4	72,7	14,7	17,4	18,5	12,0	13,7	16,1	60,1	59,7	61,1	82,8	81,2	83,4
7ps	73,8	76,4	72,7	30,3	17,4	18,5	31,7	42,5	19,3	64,9	59,7	59,2	85,3	80,9	82,5
Iidps	92,0	80,9	81,8	90,4	27,7	72,2	87,8	16,9	67,7	93,6	64,0	85,2	96,6	83,1	93,0

FIG. 5.28 - *Pourcentages de gain réalisés par les algorithmes AC-6ps, AC-7ps et AC-Iidps par rapport à AC-6 pour les problèmes d'affectation de fréquences.*

RLFAP	1	2	3	4	5	6	7	8	11	total
6/4	97,9	84,5	85,5	53,2	66,5	68,0	66,0	65,1	84,7	76,7
Iidps/6	80,8	76,5	77,1	81,8	72,2	67,7	85,2	93,0	80,0	82,0

FIG. 5.29 - *Pourcentages de gain de temps réalisés par l'algorithme AC-Iidps par rapport à AC-6 et par AC-6 par rapport à AC-4 pour les problèmes d'affectation de fréquences.*

AC-4 d'une part et AC-Iidps avec AC-6 d'autre part pour les problèmes d'affectation de fréquences (cf figure 5.29). De cette comparaison nous pouvons déduire que le nouvel algorithme proposé améliore presqu'autant AC-6, qu'AC-6 a amélioré AC-4.

5.7 Conclusion et Perspectives

Nous avons proposé dans ce chapitre deux nouveaux algorithmes AC-7 et AC-Inférence. Le premier possède une complexité semblable à celle d'AC-6 et fait environ 20% de tests de consistance en moins et améliore les performances en temps d'AC-6. AC-Inférence perd la complexité en espace d'AC-6 mais améliore d'un facteur 2 les performances d'AC-7. De plus le regroupement des contraintes identiques procure un avantage certain à AC-Iid.

Par ailleurs, nous avons proposé deux heuristiques améliorant n'importe quel algorithme de fermeture par consistance d'arc. L'une remet en cause le schéma classique des algorithmes en étudiant immédiatement les conséquences de la disparition d'une valeur pendant la phase d'initialisation. L'autre propose de propager différemment les variables dont le domaine est un singleton. Ces heuristiques se combinent aisément

avec les algorithmes AC-7 et AC-Inférence et nous ont permis d'obtenir des algorithmes performants. Pour tous les RLFAP, nous avons amélioré les temps d'AC-6 de plus de 65%.

«Est-il encore possible d'améliorer les algorithmes de fermeture par consistance d'arc?»

Nous pensons que l'on peut répondre de manière affirmative à cette question. En effet, tous les algorithmes que nous avons présentés supposent que l'on dispose a priori d'un ordre sur les valeurs des domaines. Cet ordre est utilisé pour rechercher la présence de supports. Il pourrait être intéressant de commencer par chercher si le réseau $R = (X, \mathcal{D}, \mathcal{C})$ considéré admet un sous-domaine $\mathcal{D}'$ (ie $\forall D_i \in \mathcal{D}', D'_i \subseteq D_i \in \mathcal{D}$) vérifiant la consistance d'arc. Puis de rechercher l'existence d'un support du sous-domaine. Car si le sous-domaine vérifie la consistance d'arc, alors aucune de ses valeurs ne pourra disparaître pendant une procédure de filtrage. On aura ainsi de grande chance de réduire la phase de propagation. On peut voir cette idée comme la construction par l'algorithme d'un bon ordre des valeurs des domaines.

Chapitre 6

Maintien de la consistance d'arc pendant la procédure de recherche

La combinaison d'un algorithme de filtrage avec l'algorithme de recherche backtrack a une longue histoire. Plusieurs degrés de consistance ont été étudiés expérimentalement [Haralick and Elliot, 1980; Nadel, 1988; McGregor, 1979]. Bien que les limites de ces expérimentations aient été reconnues par la communauté, les succès répétés de FC ont fait penser qu'il valait mieux employer le minimum de filtrage pendant la procédure de recherche pour obtenir les meilleurs résultats en pratique. Cette idée était tout à fait compréhensible à l'époque où le meilleur algorithme de filtrage par consistance d'arc était AC-3 et où l'algorithme utilisant ce dernier pendant la recherche était RFL. Aussi de nombreux chercheurs se sont tournés vers l'algorithme de recherche pour essayer de le rendre plus intelligent [Gaschnig, 1977; Dechter, 1988; Dechter, 1990; Ginsberg, 1993; Prosser, 1994]. Mais l'approche de la consistance d'arc a évolué, et de nouveaux algorithmes plus performants sont apparus notamment en 1986 avec AC-4. Car même si AC-4 a été critiqué comme prétraitement, sa phase de propagation est souvent plus performante que celle d'AC-3, or c'est elle qui est employée comme filtrage pendant la recherche. Il est donc surprenant de lire dans des articles récents comme [Kumar, 1992] que le filtrage employé pendant la recherche doit être limité. Ce n'est que très récemment que Sabin et Freuder [Sabin and Freuder, 1994a; Sabin and Freuder, 1994b] ont réintroduit le maintien de la consistance d'arc pendant la procédure de recherche et montré qu'il est très souvent meilleur que

le FC. L'objet de ce chapitre est de développer et d'enrichir leurs idées.

Nous commencerons par montrer que l'algorithme RFL ne peut être en toute objectivité considéré comme étant un algorithme de maintien d'un filtrage pendant une procédure de recherche, ce qui nous amènera à proposer un nouvel algorithme, mais aussi à particulariser la notion d'ordre dynamique. Puis nous proposerons d'utiliser la consistance d'arc comme filtrage et nous essayerons de déterminer quel algorithme d'AC doit être employé. Ensuite nous nous intéresserons aux comportements expérimentaux de ces différents algorithmes lorsqu'ils sont employés par une procédure de recherche. Enfin nous conclurons et nous donnerons quelques perspectives.

6.1 Maintien d'un filtrage pendant la procédure de recherche

Nous allons montrer dans cette section que l'algorithme RFL qui utilise systématiquement une procédure de filtrage après chaque instanciation doit être modifié pour obtenir un algorithme de maintien d'un filtrage pendant la recherche. Cela nous amènera à redéfinir la notion d'ordre dynamique d'instanciation.

6.1.1 Filtrage et remontée

D'une manière générale un algorithme de recherche de solution dans un CSP instancie successivement les variables dans le but d'obtenir une instanciation de toutes les variables localement consistante, c'est-à-dire une solution. Afin de réduire l'espace de recherche, des algorithmes détectant au cours de leur exécution des zones de cet espace qui ne comportent pas de solution et donc qui évitent de les parcourir, ont été développés. Pour ce faire, après chaque affectation ils vérifient la consistance de l'instanciation partielle puis examinent les valeurs des variables non encore instanciées et suppriment par filtrage celles dont on est assuré que leur emploi, compte tenu de l'instanciation partielle courante, conduira à un échec. Le degré de filtrage employé influe directement sur la diminution de l'espace de recherche, mais plus le filtrage est puissant plus il est coûteux. L'objectif est donc de déterminer le meilleur compromis entre le coût du filtrage et l'élagage réalisé. Le filtrage le plus simple est

celui employé par FC : il supprime les valeurs futures non consistantes avec la dernière valeur choisie. On peut en imaginer de nombreux autres, comme la consistance d'arc ou la consistance de chemin, mais quel que soit celui que l'on utilise les principes de l'algorithme de recherche ne sont pas remis en cause. Nous les avons présentés dans le chapitre 4 (cf algorithme 5). La figure 4.3 du chapitre 4 montre l'enchaînement des choix de valeurs, des filtrages et des restaurations après le choix d'une affectation augmentante.

L'idée directrice de cet algorithme, que nous qualifierons par la suite de classique, est donc d'étudier en filtrant les conséquences d'un choix d'une affectation augmentante. Or ce n'est pas semblable d'une manière générale au maintien d'un filtrage pendant une procédure de recherche si celui-ci se définit comme suit :

Définition 34 *Un algorithme de recherche de solution dans un CSP* ***maintient un filtrage*** *si et seulement si le CSP est systématiquement filtré avant n'importe quelle augmentation.*

L'algorithme classique ne procède au filtrage du CSP que lors d'une de descente dans l'arbre de recherche et uniquement après avoir augmenté l'instanciation courante. En fait, il considère que des valeurs futures ne peuvent être éliminées qu'après une instanciation. Si cela est vrai avec un filtrage trivial comme celui de FC[1], cela ne l'est plus avec des filtrages «plus forts» comme la consistance d'arc. En effet, avec ces derniers des modifications peuvent intervenir dans le CSP après la suppression d'une valeur précédemment choisie. Si l'on nomme ***réfutation*** le fait de supprimer la dernière valeur choisie de son domaine, nous pouvons établir le principe suivant :

> Pour maintenir un filtrage pendant la recherche, ***il faut aussi filtrer le CSP après une réfutation***, c'est-à-dire aussi bien pendant la descente que pendant la remontée dans l'arbre de recherche.

On obtient alors le schéma donné par la figure 6.1.

Cette nouvelle approche peut réduire le nombre d'échecs et n'est pas globalement plus coûteuse que la précédente.

1. A condition de considérer que le filtrage de FC élimine les valeurs non compatibles avec l'instanciation partielle et non pas celles qui n'ont pas de supports dans le domaine de la variable à instancier.

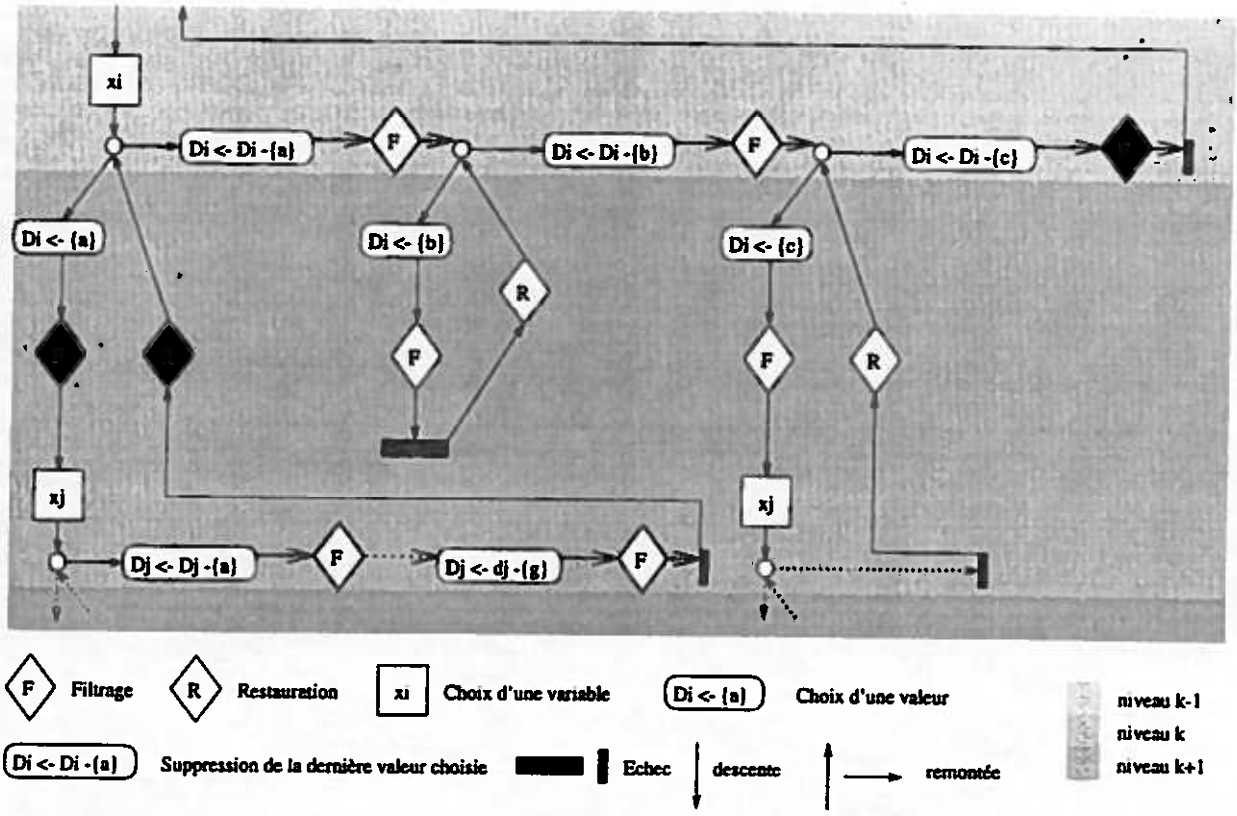


FIG. 6.1 - *Fonctionnement de l'algorithme de maintien d'un filtrage entre deux niveaux de l'arbre de recherche.*

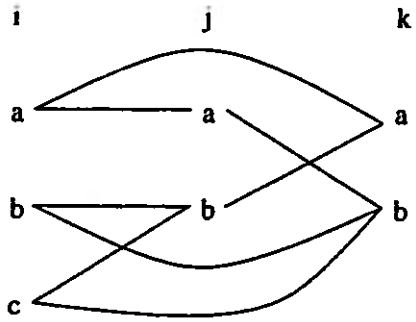


FIG. 6.2 - *Graphe de consistance d'un CSP vérifiant la consistance d'arc.*

Pour démontrer la première affirmation, plaçons nous à un niveau k arbitraire et étudions le comportement de l'algorithme classique. Son but est de faire un choix puis de filtrer le CSP afin d'une part de supprimer des valeurs dans les domaines qui ne pourront pas prolonger l'instanciation partielle en une instanciation consistante et d'autre part de transmettre un CSP non-inconsistant selon le filtrage au niveau $k+1$. Si le CSP transmis ne possède pas de solution ou bien si le filtrage détecte l'inconsistance du CSP, alors le choix est remis en cause en rétablissant le CSP dans l'état où il était avant ce choix puis en supprimant la dernière valeur choisie. À partir de là, un autre choix sera essayé. Entre le moment où une valeur précédemment choisie est supprimée et où une autre valeur est désignée, le CSP n'a pas été filtré, certaines valeurs auraient pu être éliminées par le filtrage et notamment des valeurs du domaine de la variable à instancier. En filtrant le CSP après une réfutation on pourra donc éviter des échecs. Pour illustrer cela, considérons la fermeture par consistance d'arc comme procédure de filtrage et le CSP dont le graphe de consistance est donné par la figure 6.2.

Supposons que les choix portent sur la variable x_i. Si l'on choisit la valeur a, on aboutit à un CSP vérifiant l'inconsistance d'arc, cette valeur est donc remise en cause. Si l'on étudie immédiatement les conséquences de la suppression de a de D_i on s'aperçoit que le CSP vérifie l'inconsistance d'arc, il est donc inutile d'étudier les autres choix, b et c, pour x_i.

L'utilisation du filtrage après une réfutation permet aussi d'éviter de supprimer plusieurs fois des valeurs du CSP qui ne sont viables que grâce à la présence d'une ou plusieurs valeurs du domaine de la variable à instancier. À partir de la figure 6.3,

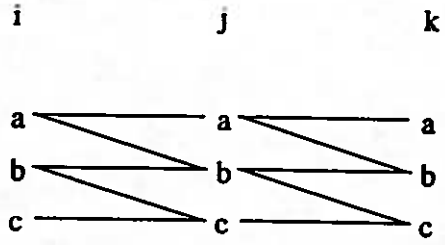


FIG. 6.3 - *Graphe de consistance d'un CSP vérifiant la consistance d'arc.*

nous allons mettre en évidence son intérêt.

Supposons que x_i soit la variable à instancier, et que les valeurs soient choisies selon l'ordre lexicographique. L'algorithme classique avec la consistance d'arc comme filtrage, autrement dit RFL, supprimera les valeurs a de D_j et a de D_k deux fois tandis qu'avec notre approche, elle ne sont éliminées qu'une seule fois.

6.1.2 Filtrage et déconnexion de variable

Nous avons vu dans le chapitre 3 que dès l'instant où une procédure de filtrage réduit le domaine d'une variable à une seule valeur, la variable est déconnectée du réseau par la phase de propagation particulière liée aux singletons. Cette idée, comme nous l'avons remarqué dans le chapitre précédent, peut aussi rendre plus efficace le filtrage, car d'une part elle permet souvent une détection plus rapide de l'inconsistance d'un réseau de contraintes et d'autre part parce qu'une variable déconnectée du réseau n'est plus prise en compte par les filtrages futurs.

6.1.3 Ordres d'instanciation

Le fait d'employer une procédure de filtrage après une réfutation permet une approche différente des ordres d'instanciation.

Nous proposons de distinguer trois types d'ordres d'instanciation, plutôt que les deux types classiques, dynamique et statique :

1. Ordre statique.
2. Ordre dynamique classique.
3. Ordre fortement dynamique.

Ordre statique

L'idée générale est de toujours choisir la même variable pour un niveau donné, que ce niveau soit atteint par la procédure de descente ou bien après la procédure de remontée. Puis d'essayer successivement toutes les valeurs de cette variable. Cet ordre est donc un ordre statique, si l'on considère que ce qualificatif s'accompagne de l'établissement avant toute recherche de l'ordre d'instanciation des variables et qu'au niveau k la $k^{ème}$ variable de cet ordre est choisie. L'ordre sur les variables est donc donné à l'avance.

Ordre dynamique classique

L'idée directrice est de ne faire un nouveau choix de variable que lorsqu'il y a un changement de niveau pendant la procédure de descente. Un nouveau choix est fait par rapport à un critère dynamique pour un niveau k et n'est pas remis en cause quand ce niveau est réétudié après avoir exploré les niveaux supérieurs à k. Une fois qu'une variable est désignée, on choisit successivement comme affectation augmentante toutes les valeurs de son domaine.

Ordre fortement dynamique

Cet ordre a été appelé «really dynamic» par D. Sabin lors d'une présentation orale à l'ECAI'94. Le processus de choix d'une variable est relancé à chaque fois qu'une modification intervient dans le réseau, après n'importe quel filtrage, soit après une augmentation, soit après une réfutation. Pour le même niveau, la variable à instancier peut donc changer avant que toutes les valeurs de la variable aient été essayées, c'est pourquoi nous avons qualifié cet ordre comme étant fortement dynamique. On peut même favoriser sa dynamicité en donnant la préférence à une variable qui n'a pas encore été choisie en cas d'ex-aequo des critères de choix. Cette approche est très différente des précédentes car ***il n'y a plus de dissociation entre le choix des variables et le choix des valeurs***. À chaque fois que l'on veut augmenter une instanciation partielle, on choisit réellement une nouvelle affectation augmentante.

La figure 6.4 met en évidence ce nouveau mécanisme de choix.

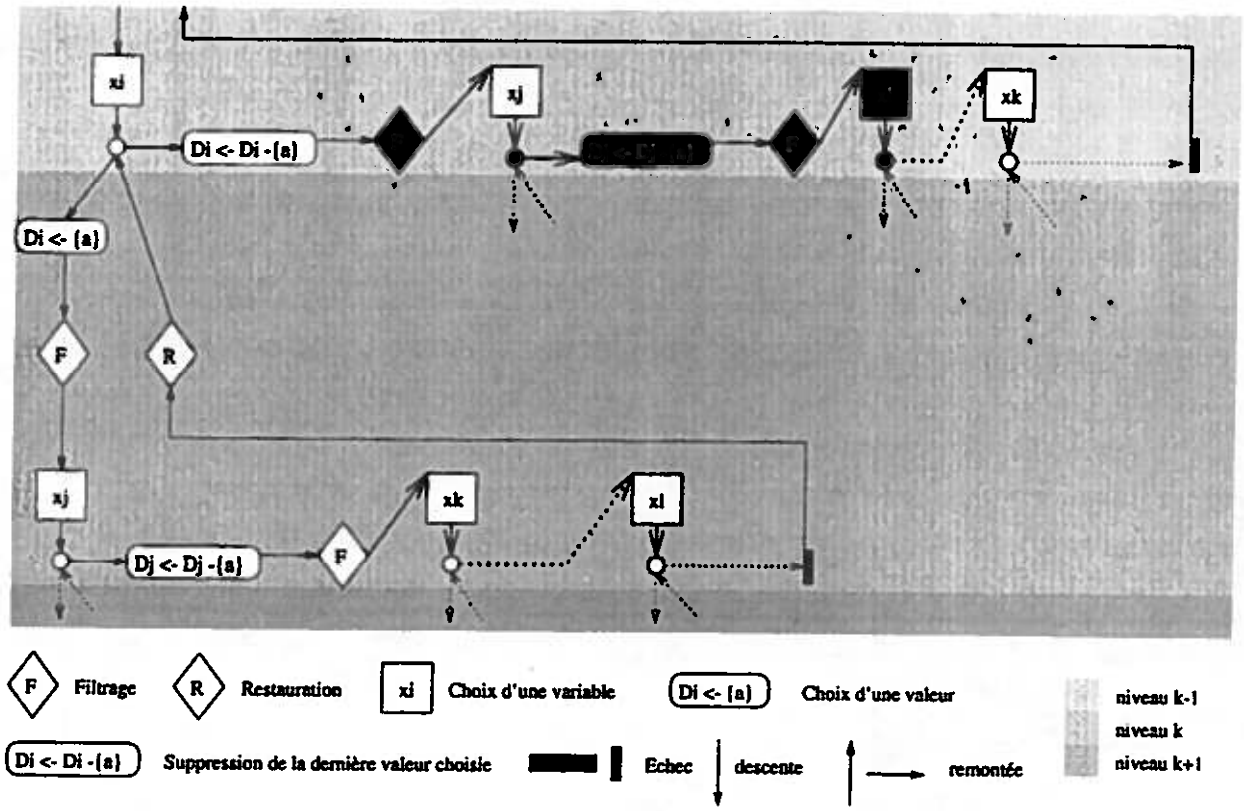


FIG. 6.4 - *Fonctionnement de l'algorithme classique entre deux niveaux de l'arbre de recherche avec un ordre fortement dynamique.*

6.1.4 L'algorithme

La version que nous présentons recherche toutes les solutions d'un CSP.

Algorithme 23 Algorithme de maintien d'un filtrage pendant la procédure de recherche

```
RECHERCHERTOUTESSOLUTIONS()
  consistant ← PRÉTRAITEMENT()
  tant que consistant faire
    (x_i, a) ← CHOISIRAFFECTATIONAUGMENTANTE(I, 0)
    RECHERCHERSOLUTIONS((x_i, a), I, 1)
    SUPPRIMER(a, D_i, 1)
    si D_i = ∅ alors consistant ← faux
    sinon consistant ← ÉLIMINERINCONSISTANCESLOCALESAPRÈSRÉFUTATION(1)
  RESTAURERCSP(1)
RECHERCHERSOLUTIONS(i (x_i, a), I, k)
  AFFECTER((x_i, a), I, k)
  si k = n alors AFFICHERSOLUTION(I)
  sinon
    consistant ← ÉLIMINERINCONSISTANCESLOCALESAPRÈSAUGMENTATION(I, k)
    tant que consistant faire
      (x_j, b) ← CHOISIRAFFECTATIONAUGMENTANTE(I, k)
      RECHERCHERSOLUTIONS((x_j, b), I, k + 1)
      SUPPRIMER(b, D_j, k)
      si D_j = ∅ alors consistant ← faux
      sinon consistant ← ÉLIMINERINCONSISTANCESLOCALESAPRÈSRÉFUTATION(k)
  RESTAURERCSP(k)
```

Par rapport à l'algorithme classique, la principale différence est le test de la consistance du CSP après la suppression de la dernière valeur choisie : fonction ÉLIMINERINCONSISTANCESLOCALESAPRÈSRÉFUTATION. Cette fonction comme ÉLIMINERINCONSISTANCESLOCALESAPRÈSAUGMENTATION est chargée d'appeler le filtrage.

C'est la fonction CHOISIRAFFECTATIONAUGMENTANTE qui détermine si une nouvelle variable doit être ou non choisie en accord avec le type d'ordre désiré. Pour

parvenir à ce résultat il faut qu'elle puisse savoir si elle est appelée après une réfutation ou bien après le passage à un niveau supérieur. En fait, il suffit qu'elle soit capable de comparer le niveau auquel elle est appelée avec le niveau de son précédent appel. Comme cela peut se mettre en oeuvre facilement avec la plupart des langages informatiques[2], nous avons décidé de ne pas surcharger le code inutilement et de considérer que l'on dispose de cette information à l'intérieur de la fonction sans avoir besoin de la passer en paramètre.

Algorithme 24 Fonction d'affectation.

```
AFFECTER(i (x_i, a), I, niveau)
    I[niveau] ← (x_i, a)
    SUPPRIMER(x_i, X)
    si |D_i| > 1 alors
        pour chaque b ∈ D_i tel que b ≠ a faire SUPPRIMER(b, D_i, niveau)
        AJOUTER(x_i, listeSingleton)
```

La fonction AFFECTER, dont le code est donné par l'algorithme 24, mérite une explication particulière. Lorsque l'on doit affecter la valeur a à la variable x_i, deux cas peuvent se présenter :

1. a est l'unique valeur de D_i;
2. D_i contient plusieurs valeurs.

Dans le premier cas, d'après le fonctionnement de l'algorithme on est assuré que x_i a été placé avant l'affectation dans la liste *listeSingleton* et traité par la fonction PROPAGATIONSINGLETON. Il ne sert donc à rien de faire d'autres tests pour a. Dans ce cas, aucune valeur n'est mise dans les listes d'attente et donc aucun filtrage n'aura lieu immédiatement après l'affectation. Par contre, dans le second cas, il est nécessaire d'étudier les conséquences du choix de a. Les autres valeurs de d_i doivent être supprimées et x_i doit être placé dans *listeSingleton*.

L'ordre fortement dynamique ne pose aucun problème puisqu'à chaque appel de la fonction CHOISIRAFFECTATIONAUGMENTANTE une variable est choisie selon un

2. En C ou en C++, on utilise des variables de type *static* locales à la procédure.

critère dynamique, sans se soucier du moment où la fonction est appelée. Pour les autres ordres, une nouvelle variable n'est choisie que si le niveau courant est supérieur au niveau précédent et si la variable dont l'instanciation vient d'être remise en cause n'a pas été instanciée par le processus de filtrage qui a suivi la réfutation née de cette remise en cause. Si l'on nomme *niveauPrec* le précédent niveau on obtient la fonction donnée par l'algorithme 25.

Algorithme 25 Fonction de choix d'une nouvelle valeur en accord avec un type d'ordre donné

```
CHOISIRAFFECTATIONAUGMENTANTE(i I, niveau)
  selon typeOrdre faire
    cas ordre statique
      si niveau > niveauPrec alors
        x_i ← PREMIER(listeVariableOrdonnée)
      sinon (x_i, tmp) ← I[niveau]
    cas ordre dynamique classique
      si niveau > niveauPrec alors
        choisir x_i parmi X selon le critère dynamique
      sinon (x_i, tmp) ← I[niveau]
    cas ordre fortement dynamique
      choisir x_i parmi X selon le critère dynamique
  niveauPrec ← niveau
  a ← CHOISIRVALEUR(x_i)
  retourner (x_i, a)
```

6.2 MAC-X = MAC + AC-X

Nous proposons dans la suite d'utiliser la consistance d'arc comme filtrage car son coût est faible et son efficacité démontrée.

L'objectif principal de cette section est de déterminer d'une part les modifications que l'on doit apporter à un algorithme d'AC pour pouvoir l'utiliser pendant

une procédure de recherche et d'autre part l'algorithme qui donne les meilleurs résultats dans ce contexte. Nous nous intéresserons essentiellement, dans la suite, à deux algorithmes d'AC : AC-7 et AC-Inférence[3].

6.2.1 Modification des algorithmes d'AC

Lors du maintien de la consistance d'arc pendant la procédure de recherche la phase d'initialisation des algorithmes d'AC n'est pas employée, si l'on excepte, bien sûr, le prétraitement. Cette phase ne subit donc pas de changement. Par contre la phase de propagation doit être modifiée afin de procéder aux mémorisations nécessaires au rétablissement de l'état du réseau avant le filtrage.

Cas général

Il est possible de dégager certaines modifications communes aux différents algorithmes d'AC possibles. Il est clair que les valeurs qui sont supprimées des domaines ainsi que les variables déconnectées et les contraintes qui les contraignent doivent être mémorisées. Il faut également procéder à la ***mémorisation de certains supports.*** En effet, supposons que l'on ne procède à aucune mémorisation pour les supports, alors les valeurs supprimées apparaîtront à tout instant dans les listes des valeurs supportées de leurs supports. Comme les modifications du réseau de contraintes se font par niveau, une valeur peut disparaître au niveau k et l'un de ses supports à des niveaux supérieurs à k. À chaque fois qu'une valeur (i, a) est éliminée, la liste S_{ia} des valeurs qu'elle supportait est parcourue, et comme (i, a) peut être supprimée de nombreuses fois lors d'une procédure de recherche, on risque d'atteindre de nombreuses fois les valeurs éliminées de S_{ia} à des niveaux inférieurs. Il est donc nécessaire d'éviter cela.

Une solution envisageable consiste lors de la suppression d'une valeur à la déconnecter systématiquement de toutes les listes de supports auxquelles elle appartient. On peut parvenir à ce résultat en mémorisant pour chaque valeur la liste des valeurs qui la supportent. Cela présente le même inconvénient que celui que nous avons présenté dans le chapitre précédent. À savoir que l'on réalise de manière systématique

3. ce qui s'applique pour AC-7, est également vrai pour AC-6.

un travail qui peut s'avérer inutile dans le cas où le CSP vérifie l'inconsistance d'arc, donc à chaque fois que l'on rencontre un échec. Comme ce cas se produit assez souvent quand le CSP est difficile à résoudre, nous ne pouvons pas accepter cette solution.

La méthode que nous proposons est un peu plus complexe et nécessite une étude plus approfondie de la procédure de restauration d'un état précédent. D'une manière générale cette étape consiste à rétablir le réseau de contraintes dans l'état où il se trouvait précédemment[4], c'est-à-dire que toute valeur possède le même ensemble de supports. Lorsqu'une valeur est remise dans un domaine, elle doit être viable, donc posséder un support sur toutes les contraintes. Si l'on est amené à supprimer des éléments des listes de valeurs supportées, il faut donc les mémoriser pour pouvoir les remettre dans ces listes. Nous proposons de supprimer une valeur (i, a) d'une liste S_{jb} quand (i, a) est atteinte lors du parcours de S_{jb} alors que a n'appartient plus à D_i. On introduit alors (j, b), c'est-à-dire le support (i, a), dans une liste particulière liée à (i, a). Ainsi, on pourra correctement rétablir les supports de (i, a) au moment où celle-ci est replacée dans le domaine. ***C'est la valeur elle-même qui va rétablir ses supports.*** Pour réaliser cela une nouvelle structure de donnée est introduite, il s'agit pour chaque valeur a supprimée du domaine D_i de la liste RS_{ia}. Lorsque la valeur a sera remise dans D_i, elle devra aussi être remise dans la liste des valeurs supportées des supports contenus dans RS_{ia}.

De plus on peut remarquer qu'il ne sert à rien de procéder à ce type de suppression si une valeur et la valeur qu'elle supporte sont supprimées de leurs domaines respectifs pendant la même phase de fermeture par consistance d'arc, car elles seront restaurées toutes les deux en même temps. En comparant le niveau auquelle une valeur a été supprimée avec le niveau courant, nous éviterons donc des modifications «intempestives».

4. Nous verrons un peu plus loin que l'on peut se contenter d'un état équivalent.

Algorithme 26 Filtrage par consistance d'arc employé pendant une procédure de recherche

```
fonction PROPAGATION(i/o listeAttente, listeSingleton i k) : booléen
  /* k est le niveau courant */
  tant que listeSingleton ≠ ∅ et listeAttente ≠ ∅ faire
    si listeSingleton ≠ ∅ alors
      prendre x_s dans listeSingleton et le supprimer
      u ← PREMIER(D_s)
      si ¬ PROPAGATIONSINGLETON(x_s, u, listeAttente, listeSingleton, k) alors
        retourner faux
    sinon
      prendre x_j dans listeAttente et le supprimer
      tant que listeAttente[j] ≠ ∅ faire
        prendre b dans listeAttente[j] et le supprimer
        pour chaque x_i ∈ Γ(x_j) tel que S_jb[i] ≠ ∅ faire
          si |D_i| > 1 alors
            pour chaque (i, a) ∈ S_jb[i] faire
              si a ∈ D_i alors
                si ¬ EXISTESUPPORT(x_i, a, C_ij, b, k) alors
                  SUPPRIMERETMETTREENATTENTE(a, D_i, listeAttente, listeSingleton, k)
                sinon
                  si NIVEAUSUPPRESSION((x_i, a)) ≠ k alors
                    SUPPRIMER((i, a), S_jb)
                    AJOUTER((j, b), RS_ia)
          sinon
            si NIVEAUDÉCONNEXION(x_i) ≠ k alors
              DÉCONNECTER(S_jb[i], S_jb)
              AJOUTER((j, b), RX[C_ji])
    retourner vrai
```

Donc ***une valeur*** a ***de*** x_i ***sera ôtée d'une liste*** $S_{jb}[i]$ ***contenant*** (i, a) ***si*** (j, b)

a été supprimé à un niveau k et que la conséquence de la disparition de (j,b) a entraîné l'étude de la viabilité de (i,a) , alors que a n'appartenait plus à D_i depuis un niveau strictement inférieur à k.

Comme toute suppression est attachée à un niveau, tous les tableaux de mémorisation seront indicés par des niveaux et dès lors qu'une valeur sera supprimée d'un domaine, elle sera marquée par le niveau de sa suppression, qui pourra être accédé par la fonction NIVEAUSUPPRESSION.

Un problème particulier se pose pour les variables déconnectées qui sont atteintes par la propagation d'une valeur supprimée. Nous avons proposé en 4.11.3 un mécanisme permettant de supprimer les contraintes C_{*s} où x_s est une variable instanciée, ou plus exactement d'éviter d'étudier les valeurs de x_s lors de la propagation. Le maintien de la consistance d'arc nous oblige à avoir une vision globale de l'évolution du CSP pendant toute la recherche. Aussi pour les mêmes raisons que précédemment avec les supports, si une variable est instanciée à un niveau k elle ne doit pas pouvoir être atteinte plusieurs fois à cause de la suppression d'une même valeur à des niveaux supérieurs à k. Pour y parvenir il suffit, par exemple, d'éclater les listes de supports en fonction des contraintes, comme pour AC-7, afin de pouvoir utiliser la méthode proposée en 4.11.3, et lorsqu'on rencontre x_i déconnectée, en étudiant les conséquences de la disparition d'une valeur b d'un domaine D_j, de déconnecter la liste $S_{jb}[i]$ de la liste S_{jb} et d'ajouter (j,b) dans une liste particulière que nous noterons RX de la contrainte C_{ji}. Ainsi quand C_{ji} sera replacée dans le graphe des contraintes elle donnera l'ordre à la valeur b de x_j d'ajouter la liste $S_{jb}[i]$ dans S_{jb}; cet ajout se fera sans avoir besoin de parcourir tous les éléments de la liste.

Pour résumer les nouvelles structures de données sont :

- RS_{ia} : un ensemble de supports. Pour (i,a) supprimée du domaine D_i, cette liste est constituée de l'ensemble des supports (j,b) pour lesquels (i,a) devra, lorsque a sera remis dans D_i, être remis dans leur liste de valeur supportées $S_{jb}[i]$.

- $RX[C_{ji}]$: une liste de valeurs. Pour une contrainte C_{ji}, avec x_i déconnectée à un niveau k, cette liste contient l'ensemble des valeurs de x_j qui ont été supprimées à des niveaux supérieurs à k et pour lesquels l'étude des conséquences de leur disparition les a amenés à vouloir étudier une valeur de x_i alors que celle-ci était

instanciée.

Si l'on considère que la fonction EXISTESUPPORT, comme dans le chapitre 3, mémorise le nouveau support trouvé si l'on emploie AC-6, AC-7 ou AC-Inférence alors l'algorithme 26 donne la structure commune à tous ces algorithmes utilisés pendant la procédure de recherche.

Il est possible de donner aussi une procédure de restauration générique (cf algorithme 27).

Algorithme 27 Restauration du CSP de l'instant précédent.

```
procédure RESTAURERCSP(i k)
    /* k est un niveau */
    (x_p, a) ← I[k]
    /* on remet x_p dans la liste des variables */
    AJOUTER(x_p, X)
    pour chaque variable x_i déconnectée au niveau k faire
        pour chaque x_j ∈ Γ(x_i) faire
            /* on remet la contrainte C_ji *
            AJOUTER(x_i, Γ(x_j))
            /* on reconnecte les listes de valeurs supportées déconnectées */
            pour chaque (j, b) ∈ RX[C_ji] faire AJOUTER(S_jb[i], S_jb)
            RX[C_ji] ← ∅
    pour chaque valeur (x_i, a) supprimée au niveau k faire
        /* on remet a dans D_i */
        AJOUTER(a, D_i)
        /* on remet (x_i, a) dans certaines listes de ses supports */
        pour chaque (x_j, b) ∈ RS_ia faire AJOUTER((x_i, a), S_jb)
        RS_ia ← ∅
```

A partir des considérations précédentes on peut dégager trois caractéristiques que devraient posséder un "bon" algorithme de filtrage par consistance d'arc employé pendant une procédure de recherche :

1. Pouvoir trouver rapidement un nouveau support pour une valeur sur une contrainte.

2. Limiter l'étude de la conséquence de la suppression d'une valeur sur une contrainte, et atteindre le moins possible de valeurs supprimées à l'instant courant ou supprimées à un instant précédent ou encore qui possèdent un autre support sur la contrainte.

3. Minimiser le nombre de restaurations.

Les deux dernières caractéristiques sont très liées. Plus on atteindra de valeurs supprimées, plus on mémorisera des supports et donc plus les restaurations seront nombreuses. Nous allons dans les sections suivantes étudier le comportement des algorithmes AC-7 et AC-Inférence lorsqu'ils sont employés pendant une procédure de recherche. Cependant, on peut remarquer immédiatement que pour AC-7 et AC-Inférence, qui n'ont besoin de connaître qu'un seul support par valeur sur n'importe quelle contrainte, le nombre de valeurs atteintes suite à la disparition d'une valeur est faible. La deuxième caractéristique d'un bon algorithme de filtrage par consistance d'arc employé pendant la procédure de recherche est donc possédée par ces deux algorithmes.

6.2.2 MAC-7

AC-7 commence par rechercher un nouveau support pour une valeur (i, a) dans la liste S_{ia} des valeurs qu'elle supporte. Or il est possible que (j, b), une valeur supportée, n'appartienne plus au domaine de sa variable. (j, b) doit donc être momentanément supprimée de S_{ia} afin que ce constat ne puisse pas être fait plusieurs fois pour un même sous-arbre de l'arbre de recherche. La valeur pour laquelle on recherche un nouveau support appartient toujours au CSP. On pourrait donc attendre pour savoir si (i, a) va ou non posséder un nouveau support, car si elle n'a plus de support elle sera éliminée. De plus, dans le cas où le niveau de suppression de (j, b) est le niveau courant on pourrait ne pas supprimer (j, b) de S_{ia}. Mais cela ne présente pas d'intérêt car de toute façon il faudrait mémoriser les valeurs atteintes n'appartenant plus au domaine de leur variable; on ne peut plus se contenter d'un seul test. Aussi nous utiliserons les listes RS selon le même principe que précédemment.

Pour résumer nous pouvons donner, pour AC-7, une seconde raison justifiant la suppression d'une valeur d'une liste des valeurs supportées: ***une valeur*** (j, b) ***sera***

supprimée d'une liste $S_{ia}[j]$ ***des valeurs supportées d'un de ses supports si*** (j,b) ***a été atteinte pendant la procédure de recherche d'un nouveau support de*** (i,a) ***alors que*** b ***n'appartenait plus à*** D_j.

Donc, quand AC-7 est employé pendant une procédure de recherche, un élément (x_j,b) pourra être supprimé d'une liste S_{ia} pour trois raisons:

1. si a a été supprimée de D_i et si un nouveau support a été trouvé pour b de x_j sur la contrainte C_{ji} (le cas normal dans AC-7).

2. si le niveau de sa disparition est inférieur à celui pendant lequel la liste a été parcourue pour réétudier la viabilité des valeurs appartenant à la liste.

3. si pendant la recherche d'un nouveau support pour (i,a), (j,b) est atteint et b n'appartient plus à D_j

Dans les deux derniers cas, on mémorise (i,a) dans RS_{jb}, pour pouvoir remettre (j,b) dans S_{ia} lors de la restauration de b dans D_j. Par contre, pour la première éventualité, aucune mémorisation ne sera réalisée et donc aucune restauration. Aussi, avec MAC-7, ***un CSP peut être restauré dans un état équivalent et non plus exactement dans l'état où il se trouvait avant une augmentation.*** Le support d'une valeur sur une contrainte peut donc évoluer dans le temps. La figure 6.5 montre sur un exemple le changement possible de support pour certaines valeurs après restauration, ici a et b de x_k sur C_{kj}.

Par ailleurs, AC-7 a besoin de connaître, pendant la recherche d'un nouveau support pour (i,a) sur C_{ij}, la dernière valeur étudiée dans le domaine de D_j par a sur C_{ij}, notée $L_{ia}[j]$. Comme les $L_{ia}[j]$ peuvent être modifiées pendant la procédure de filtrage il faudra pouvoir rétablir ces tableaux dans l'état où ils se trouvaient avant le filtrage si l'on veut être sûr de ne pas oublier de supports. Ce qui importe ce n'est pas de mémoriser toutes les modifications pouvant survenir pendant une phase de filtrage, mais de pouvoir restaurer la dernière valeur étudiée d'une valeur avant cette phase. Une mémorisation n'est donc nécessaire que lors de la première modification. Il faut donc pouvoir savoir si pour un instant donné $L_{ia}[j]$ a déjà été mémorisée avant cet instant. Pour parvenir à ce résultat on associe à tous les $L_{ia}[j]$ le niveau de leur dernière modification. Nous appellerons NIVEAUDERNIÈREMODIFICATION la fonction qui permet d'accéder à cette information. Aussi lors de la descente dans l'arbre

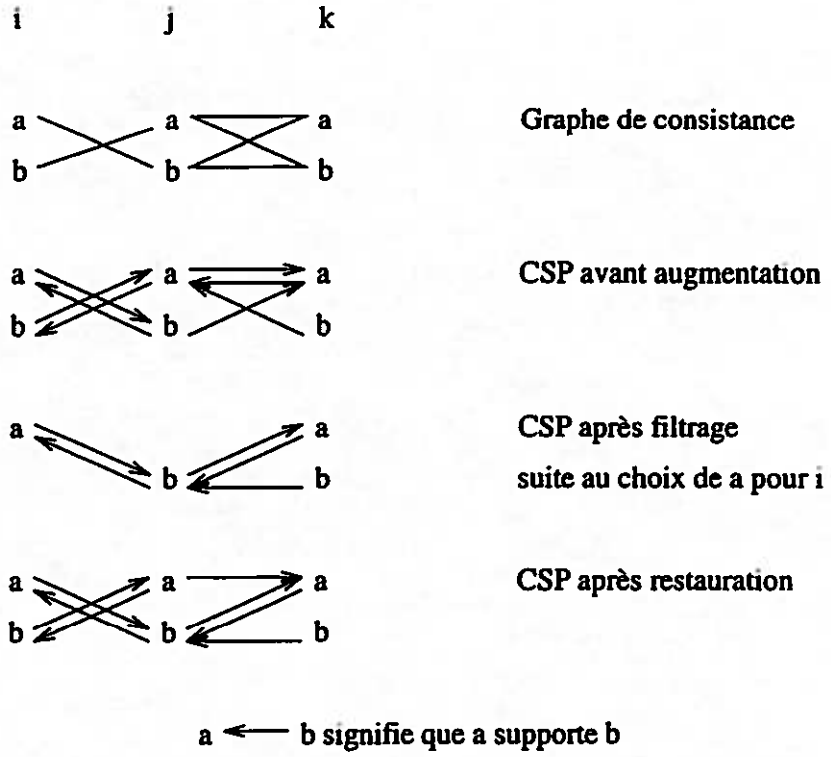


FIG. 6.5 - *Évolution des supports après filtrage et restauration.*

de recherche, pour savoir si $L_{ia}[j]$ a déjà été modifiée, il suffira de comparer le niveau courant avec celui retourné par la fonction NIVEAUDERNIÈREMODIFICATION($L_{ia}[j]$). Cette dernière valeur doit être correctement remise à jour lors des remontées. Pour ce faire on mémorise ensemble la valeur de $L_{ia}[j]$ et le niveau de la précédente modification, pour qu'une restauration rétablisse correctement ces deux valeurs.

D'autre part, la nouvelle possibilité de suppression d'éléments des listes de valeurs supportées impose une vision différente de la notion de dernière valeur étudiée. Comme un CSP peut être rétabli dans un état équivalent, il est possible pour une valeur (i, a) que la dernière valeur qu'elle ait étudiée sur une contrainte ne soit plus explicitement un des ses supports, c'est-à-dire que (i, a) n'apparaisse plus dans la liste des valeurs supportées de sa dernière valeur étudiée. Considérons l'exemple donné par la figure 6.6 et supposons que la valeur (j, a) soit éliminée pendant le processus de filtrage. Cette valeur correspond à $L_{ib}[j]$. La valeur (j, b) devient le nouveau support pour (i, b) et $L_{ib}[j]$ est modifiée en conséquence puisque ce support a été trouvé en parcourant le domaine de D_j. La restauration rétablit la précédente valeur de $L_{ib}[j]$,

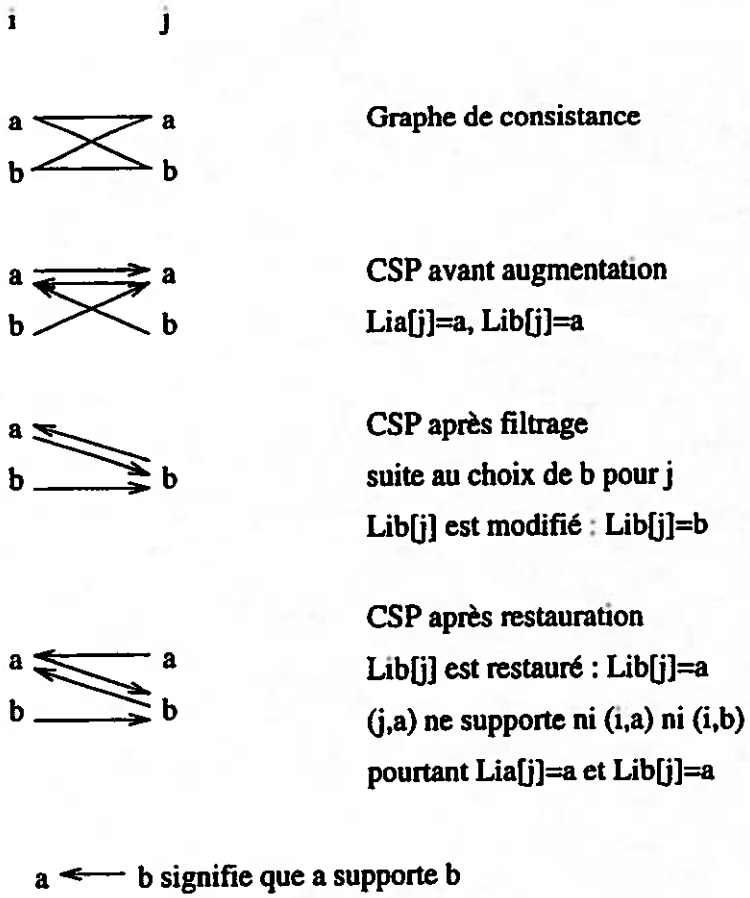


FIG. 6.6 - *Modification et restauration des dernières valeur étudiées.*

c'est-à-dire a, qui ne supporte plus (i, b). Et bien que $L_{ia}[j]$ ne soit pas modifiée, on retrouve le même résultat pour (i, a).

La dernière valeur étudiée est en fait une limite, on est assuré qu'il n'existe pas de valeurs inférieures dans le domaine qui puisse être un support. Si cette valeur n'apparaît plus explicitement comme un support, elle le reste de manière implicite. Quand on recherchera un nouveau support pour une valeur (i, a) sur une contrainte C_{ij} dans le domaine D_j ***il faudra donc tester si*** $L_{ia}[j]$ ***appartient à*** D_j ***pour rendre explicite ce support.*** Ce problème apparaît aussi quand on veut éviter des tests négatifs. En effet une liste S_{ia} peut ne plus contenir une valeur (j, b) telle que $L_{jb}[i] = a$. (i, a) ne pourra donc pas trouver (j, b) comme support pendant la première étape de la fonction EXISTESUPPORT soit pendant la recherche dans la liste $S_{ia}[j]$. Ensuite elle ne l'acceptera pas non plus pendant la recherche dans le

Algorithme 28 Modifications d'AC-7 pour pouvoir être employé pendant une procédure de recherche.

```
fonction EXISTESUPPORT(i x_i, a, C_ij, b, k) : booléen
    /* k est le niveau courant */
    trouvé ← faux
    tant que ¬trouvé et S_ia[j] ≠ ∅ faire
        c ← PREMIER(S_ia[j])
        si c ∉ D_j alors
            SUPPRIMER(c, S_ia[j])
            MÉMORISER((i, a), RS_jc)
        sinon trouvé ← vrai
    si ¬trouvé et L_ia[j] ∈ D_j alors
        c ← L_ia[j]
        trouvé ← vrai
    tant que ¬trouvé et c ≤ DERNIER(D_j) faire
        c ← SUIVANT(c, D_j)
        si L_jc[i] <= a alors
            si L_jc[i] = a ou C_ij(a, c) alors
                trouvé ← vrai
                /* on mémorise L_ia[j] si c'est la première fois */
                /* qu'elle est modifiée au niveau k */
                dernièreMémo ← NIVEAUDERNIÈREMÉMORISATION(L_ia[j])
                si dernièreMémo ≠ k alors
                    MÉMORISER(L_ia[j], dernièreMémo)
                    NIVEAUDERNIÈREMÉMORISATION(L_ia[j]) ← k
                L_ia[j] ← c
    si trouvé alors
        si b ≠ nil alors SUPPRIMER((i, a), S_jb)
        AJOUTER((i, a), S_jc[i])
    retourner trouvé
```

domaine puisque $L_{jb}[i]$ n'est pas strictement inférieur à a. Il faut donc modifier cette dernière condition afin qu'elle accepte des valeurs inférieures ou égales à a (et non plus strictement inférieures). Comme cette recherche ne considère que des valeurs du domaine, on pourra donc se passer du test pour savoir si elles appartiennent au domaine. Ces modifications ne changent pas la complexité dans le pire des cas d'AC-7. L'algorithme 28 résume les modifications nécessaires à l'emploi d'AC-7 pendant une procédure de recherche.

L'inconvénient majeur de MAC-7 est le coût des mémorisations des tableaux L dans le pire des cas, qui, heureusement, est très loin de la complexité réelle. Comme la dernière valeur étudiée $L_{ia}[j]$ pour une valeur (i, a) sur une contrainte C_{ij} peut être modifiée par la disparition de chaque valeur de x_j, la complexité globale est bornée par $O(ed^2)$, mais on peut remarquer aussi que $L_{ia}[j]$ ne peut être mémorisée qu'une seule fois par niveau et comme la profondeur maximale de l'arbre de recherche est bornée par n (le nombre de variable du CSP à l'origine), le nombre de mémorisations est également borné par $O(ned)$. La complexité en espace de MAC-7 est donc en $O(min(ed^2, ned))$. La complexité en temps pour une branche de l'arbre de recherche reste en $O(ed^2)$. Mais intuitivement on devine que MAC-7 risque de parcourir plusieurs fois les mêmes valeurs de certains domaines pour trouver des supports, pour un sous-arbre de l'arbre de recherche. Aussi on peut s'attendre à ce que ses performances diminuent en même temps que le taux de satisfaisabilité des contraintes. Toutefois nous verrons dans la partie expérimentale que MAC-7 est rarement pénalisé pour des CSP ayant, à l'origine, des contraintes de ce type, car le taux de satisfaisabilité des contraintes évolue pendant la recherche.

6.2.3 MAC-Inférence

Pour des raisons similaires à celles que nous avons précédemment exposées, certaines mémorisations vont être nécessaires afin de rétablir, lors de remontées, le réseau dans un état équivalent. Rappelons, tout d'abord, les structures de données d'AC-Inférence. Pour chaque valeur sur chaque contrainte, AC-Inférence utilise trois types de listes : une liste de type S ($S_{ia}[j]$ par exemple) qui contient les valeurs supportées, une liste de type P contenant les valeurs qui sont connues comme étant consistantes et une liste de type U contenant les valeurs dont on ne sait pas si elles sont des

supports.

Lorsqu'AC-Inférence est utilisé comme prétraitement, il existe trois types de transfert entre éléments de listes de types différents:

1. de U vers S;
2. de U vers P.
3. de P vers S;

Le premier et le troisième types de transfert sont réalisés par la fonction EXISTESUPPORT tandis que le deuxième est effectué par la fonction DÉDUIRESUPPORT.

Si l'on respecte les principes de restauration et de mémorisation précédemment exprimés nous aurons besoin de 3 nouveaux types de listes pour pouvoir utiliser AC-Inférence pendant une procédure de recherche. Ce sont pour une valeur (i, a) sur une contrainte C_{ij} :

- RS_{jb} : cette liste contient les valeurs (i, a) tels que (j, b) appartenait la liste $S_{ia}[j]$ et a été atteint alors que b a été supprimé de D_j à un niveau inférieur au niveau courant.
- RP_{jb} : idem ci-dessus pour les listes $P_{i*}[j]$.
- RU_{jb} : idem ci-dessus pour les listes $U_{i*}[j]$.

Bien entendu, lorsqu'une valeur (j, b) sera remise dans D_j, on replacera aussi (j, b) dans les listes de type P, U et S correspondant respectivement aux valeurs appartenant à RP, RU et RS.

Maintenant, si on utilise AC-Inférence pendant une procédure de recherche, d'autres types de transfert vont intervenir :

1. de S vers RS;
2. de P vers RP;
3. de U vers RU;
4. de S vers RP;

5. de RS vers S;
6. de RP vers P;
7. de RU vers U;
8. de RU vers RP;

type de transfert 1 :
Ce type de transfert est similaire à celui précédemment présenté pour tous les algorithme de consistance d'arc procédant par maintien de support. Il est effectué par la fonction PROPAGATION.

types de transfert 2 à 4 :
Ces types de transfert sont réalisés par la fonction EXISTESUPPORT. Ils permettent d'éviter d'atteindre plusieurs fois pendant la procédure de recherche une valeur qui n'appartient plus au domaine de sa variable. Leur principe est semblable à celui du type précédent.

types de transfert 5 à 7 :
Ces transferts permettent de rétablir le CSP dans un état équivalent lors du processus de remontée. Aussi c'est la fonction RESTAURERCSP qui en a la charge.

type de transfert 8 :
C'est un transfert qui est dû au principe de déduction employé par l'algorithme. Il sera donc réaliser par la fonction DÉDUIRESUPPORT. Pour mettre en évidence l'intérêt de ce transfert considérons une contrainte commutative C_{ij} et les valeurs a et b de D_i munies des listes de type U suivantes :

$$U_{ia}[j] = \{b, c\}$$
$$U_{ib}[j] = \{a, d\}$$

Supposons que l'on recherche un support pour la valeur (i, a) sur C_{ij} dans la liste $U_{ia}[j]$. Supposons aussi que la valeur b n'appartienne plus à D_j. La valeur b de $U_{ia}[j]$ est atteinte en premier, comme elle n'appartient plus à D_j, on mémorise (i, a) dans RU_{jb} et on supprime b de $U_{ia}[j]$. Supposons maintenant que l'on trouve a comme

support pour (i, b) après avoir parcouru la liste $U_{ib}[j]$. $C_{ij}(b, a)$ est vrai donc d'après la commutativité $C_{ij}(a, b)$ aussi; par conséquent la valeur (i, a) de RU_{jb} doit être transférée dans RP_{jb} afin de prendre en compte ce résultat quand b sera remis dans D_j. Si le test de consistance s'était avéré faux alors la valeur (i, a) aurait dû être supprimé de RU_{jb}.

L'algorithme 29 montre les modifications qu'il faut apporter à AC-Inférence pour pouvoir l'utiliser pendant une procédure de recherche.

Contraintes Identiques

Le regroupement des contraintes identiques entraîne l'ajout de nouvelles structures de données pour AC-Inférence si les listes de type U et P pour un ensemble de contraintes identiques sont partagées. Par exemple si C_{ij} et C_{kl} sont deux contraintes identiques alors les listes $P_{ia}[j]$ et $P_{ka}[j]$ ne sont représentées qu'une seule fois. Afin que ce partage de ressources puisse se faire, on munit chaque valeur sur chaque contrainte d'un pointeur sur la dernière valeur étudiée pour la liste P et la liste U rattachée à cette valeur. Pour que ce mécanisme fonctionne correctement lors d'une procédure de recherche il faut mémoriser les variations de ces pointeurs pour chaque niveau. Le principe de mémorisation et de restauration de ces pointeurs est semblable à celui que nous avons décrit pour l'indice de la dernière valeur étudiée avec AC-7. Par contre, on ne procédera plus à aucune suppression dans les liste de type P. Les listes de type RP et RU ne seront plus utiles.

En utilisant cette méthode on pourra être amené à atteindre plusieurs fois des valeurs n'appartenant plus au domaine. Mais on ne pourra pas réaliser plusieurs fois le même test de consistance.

6.3 Représentation Ensembliste

Si l'on suppose que l'on sait réaliser efficacement certaines opérations de base sur les ensembles, comme par exemple l'intersection, alors il est possible de simplifier les algorithmes. L'implémentation de ces opérations peut se faire, par exemple, à l'aide de vecteurs de bits.

Nous supposerons que les domaines des variables et les listes P, qui contiennent les

Algorithme 29 Fonction EXISTESUPPORT d'AC-Inférence employé pendant une procédure de recherche

```
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
  trouvé ← faux
  /* recherche dans la liste des supports connus */
  tant que ¬trouvé et P_ia[j] ≠ ∅ faire
    c ← PREMIER(P_ia[j])
    si c ∉ D_j alors
      SUPPRIMER(c, P_ia[j])
      MÉMORISER((i, a), RP_jc)
    sinon
      trouvé ← vrai
  /* recherche dans la liste des valeurs pouvant être des supports */
  tant que ¬trouvé et U_ia[j] ≠ ∅ faire
    c ← PREMIER(U_ia[j], D_j)
    SUPPRIMER(c, U_ia[j])
    si c ∉ D_j alors MÉMORISER((i, a), RU_jc)
    sinon
      trouvé ← C_ij(a, c)
      DÉDUIRESUPPORT((i, a), (j, c), trouvé)
  si trouvé alors
    si b ≠ nil alors
      SUPPRIMER((i, a), S_jb)
      MÉMORISER((i, a), RP_jb)
    AJOUTER((i, a), S_jc[i])
  retourner trouvé
```

valeurs qui sont connues comme étant consistantes, et U, qui contiennent les valeurs dont on ne sait pas si elles sont des supports, sont représentées par des ensembles. La nouvelle fonction EXISTESUPPORT de MAC-Inférence se simplifie comme le montre l'algorithme 30. En effet, contrairement à la précédente, elle ne contient plus de mécanisme de mémorisation. La fonction PREMIER(E) retourne le premier élément d'un ensemble E si E n'est pas vide et *nil* sinon.

Algorithme 30 Fonction EXISTESUPPORT d'AC-Inférence utilisant une représentation ensembliste des listes P et U et des domaines.

```
EXISTESUPPORT(i x_i, a, C_ij, b) : booléen
  trouvé ← vrai
  c ← PREMIER(D_j ∩ P_ia[j])
  si c = nil alors
  |  trouvé ← faux
  |  pour chaque c ∈ (D_j ∩ U_ia[j]) tant que ¬trouvé faire
  |  |  SUPPRIMER(c, U_ia[j])
  |  |  trouvé ← C_ij(a, c)
  |  |  si trouvé alors AJOUTER(c, P_ia[j]) DÉDUIRESUPPORT((i, a), (j, c), trouvé)
  si trouvé alors
  |  SUPPRIMER((i, a), S_jb)
  |  AJOUTER((i, a), S_jc[i])
  retourner trouvé
```

6.4 Expérimentations

Nous rappelons que le suffixe «ps» ajouté au nom des algorithmes signifie qu'ils utilisent le traitement particulier pour les variables dont le domaine est un singleton.

Dans un premier temps, nous détaillerons les résultats obtenus pour Iso-ssgrpartiel, puis nous nous intéresserons aux problèmes aléatoires et enfin aux problèmes d'affectation de fréquences.

La version de MAC-4 que nous avons utilisée a été très fortement optimisée. Elle améliore au moins de 50% les versions plus classiques.

	tps (en s)	rs (en millier)	% gain tps	% gain rs
MAC-4	373,3	4 534	-	-
MAC-6	256,3	2 791	31,3	38,4
MAC-6ps	222,0	2 510	13,2	10,1

FIG. 6.7 - *Comparaison des performances des algorithmes MAC-4, MAC-6 et MAC-6ps pour* lso-ssgrpartiel. *Les gains de MAC-6 sont exprimés par rapport à MAC-4, ceux de MAC-6ps par rapport à MAC-6.*

Pour lso-ssgrpartiel on obtient les résultats donnés en figure 6.7.

Les temps donnés sont ceux pour pour résoudre les 10 000 instances définies selon le protocole précédemment exposé. MAC-6 met 30% de temps de moins que MAC-4 ce qui est mieux qu'AC-6 par rapport à AC-4. Les améliorations des algorithmes d'AC conduisent donc à un gain de temps lorsque la consistance d'arc est maintenue pendant la recherche.

Pour le nombre d'appels à la fonction EXISTESUPPORT, la différence entre MAC-4 et MAC-6 est très importante. Aussi AC-6, même quand il est employé pendant la recherche, a une phase de propagation moins coûteuse que celle AC-4. La viabilité des valeurs est nettement moins souvent remise en cause par MAC-6 que par MAC-4.

MAC-6ps améliore en temps MAC-6 d'environ 13%. Ce chiffre est supérieur au gain d'AC-6ps sur AC-6.

Les différences que nous venons d'observer sont relativement modestes car les problèmes testés sont faciles. En effet, MAC-6 n'a besoin, en moyenne, que de 25 ms pour résoudre une instance. C'est pourquoi nous proposons d'étudier le comportement des algorithmes pour des problèmes plus difficiles.

Nous avons généré 3 ensembles de réseaux aléatoires difficiles. Le premier ensemble comprend des réseaux ayant 20 variables et 10 valeurs. Le deuxième est constitué des réseaux ayant plus valeurs que le précédent, nous avons choisi 30 valeurs. Le troisième a pour élément des réseaux ayant plus de variables que le premier, ici 30 variables. Afin d'obtenir des réseaux difficiles à résoudre, nous avons calculé grâce à [Prosser, 1994], pour chaque densité du graphe, un taux de satisfaisabilité de façon à placer le problème sur la phase de transition. Chaque point des courbes données en figure 6.8, 6.9 et 6.10 représente la moyenne obtenue pour 30 instances.

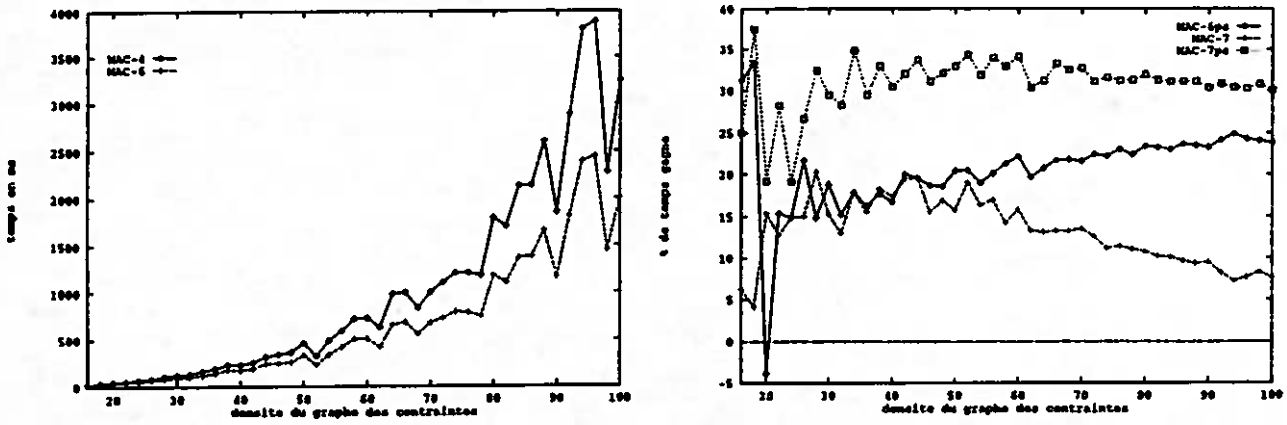


FIG. 6.8 - *Comparaison des performances des algorithmes MAC-4, MAC-6, MAC-6ps, MAC-7 et MAC-7ps pour des réseaux aléatoires ayant 20 variables 10 valeurs.*

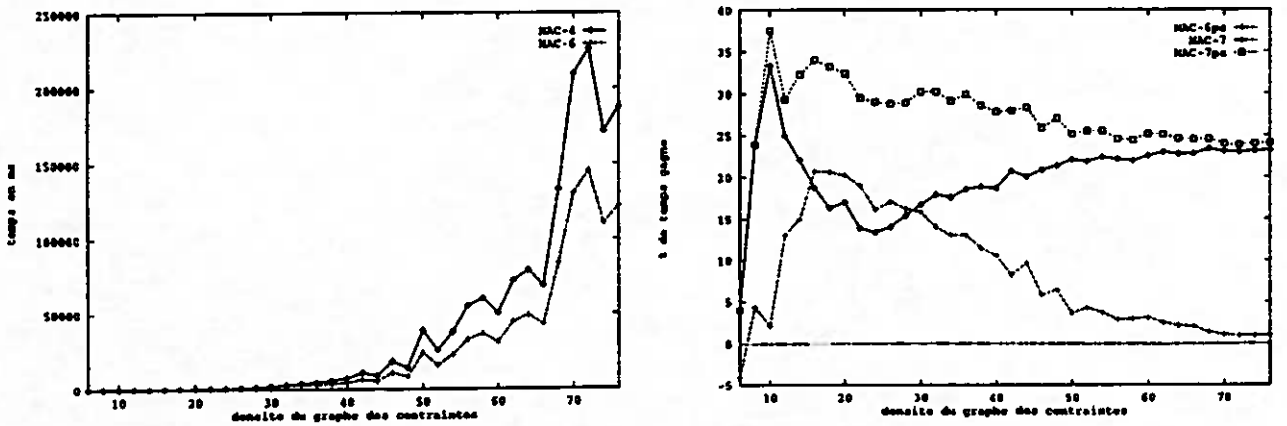


FIG. 6.9 - *Comparaison des performances des algorithmes MAC-4, MAC-6, MAC-6ps, MAC-7 et MAC-7ps pour des réseaux aléatoires ayant 20 variables 30 valeurs.*

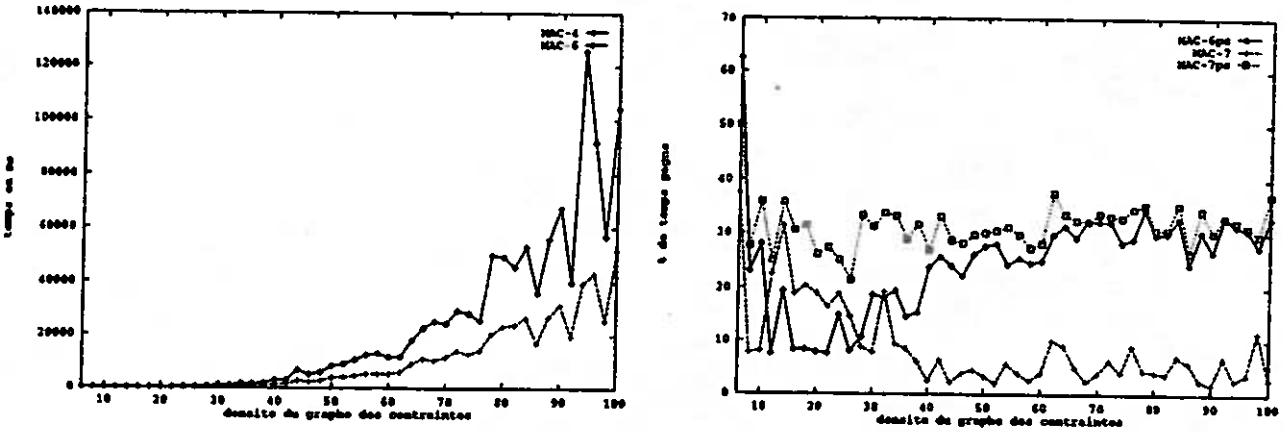


FIG. 6.10 - *Comparaison des performances des algorithmes MAC-4, MAC-6, MAC-6ps, MAC-7 et MAC-7ps pour des réseaux aléatoires ayant 30 variables 10 valeurs.*

Il apparaît clairement sur ces figures que MAC-6 est meilleur que MAC-4. Plus le nombre de valeurs augmente ou plus le nombre de variables croit et plus la différence (pour une même densité) est importante.

MAC-6ps économise 20 % du temps mis par MAC-6 pour des réseaux ayant 20 variables. Lorsque les réseaux ont plus de variables, l'économie est plus importante (cf figure 6.10), ce qui montre l'intérêt du traitement particulier pour les domaines qui deviennent des singletons.

Le comportement de MAC-7 par rapport à MAC-6 est assez difficile à étudier. Pour le premier ensemble de réseaux le gain réalisé par MAC-7 est d'environ 12%. Pour les réseaux ayant une faible densité l'utilisation de la bidirectionnalité des contraintes est intéressante, mais elle le devient beaucoup moins lorsque la densité augmente. On ne retrouve pas ce résultat pour MAC-7ps car plus la densité augmente et plus le traitement particulier pour les domaines qui deviennent des singletons améliore les performances de MAC-7. Avec MAC-7ps, on est pratiquement assuré de résoudre les problèmes en utilisant 25% de temps en moins que MAC-6.

Considérons maintenant les problèmes d'affectation de fréquences. Seules les instances 1, 2, 3, 4, 5 et 11 des RLFAP ne vérifient pas l'inconsistance d'arc, et parmi elles, seule la 11 est difficile à résoudre (plus de 2 500 backtracks). Les autres se résolvent sans retour-arrière. Les résultats que nous avons obtenus sont donnés en figure 6.11 et 6.12.

	RLFAP-1			RLFAP-2			RLFAP-3		
	cc	rs	tps	cc	rs	tps	cc	rs	tps
4	16 996	5 032	65,3	4 074	1 185	13,8	8 922	2 687	30,8
6	1 933	839	12,4	450	192	2,6	997	476	6,0
7	1 341	837	12,5	312	192	2,4	715	474	5,6
Iid	24	818	4,8	11	193	0,9	20	452	2,0
6ps	1 877	768	11,5	434	178	2,4	961	436	5,4
7ps	1 291	766	10,7	299	178	2,2	682	434	5,1
Iidps	26	736	4,0	11	176	0,9	20	412	2,0

	RLFAP-4			RLFAP-5			RLFAP-11		
	cc	rs	tps	cc	rs	tps	cc	rs	tps
4	1 822	261	5,0	5 610	1 206	16,7	13 051	37 038	138,8
6	492	90	2,4	1 261	223	5,7	2 667	2 258	22,7
7	433	90	2,5	988	226	5,6	2 165	2 298	21,8
Iid	62	72	1,0	139	207	2,0	25	2 429	15,5
6ps	147	22	0,8	1 080	184	4,7	2 587	1 571	16,9
7ps	129	22	0,8	881	185	4,7	2 096	1 585	16,2
Iidps	40	18	0,5	124	162	1,7	24	1 602	10,1

FIG. 6.11 - *Comparaison des performances des algorithmes MAC-4, MAC-6, MAC-7, MAC-Iid, MAC-6ps, MAC-7ps, MAC-Iidps pour les problèmes d'affectation de fréquences. Le nombre de tests de consistance réalisés (cc) et le nombre d'appels à la fonction* EXISTESUPPORT *(rs) sont exprimés en milliers, les temps (tps) sont donnés en seconde.*

	RLFAP-1			RLFAP-2			RLFAP-3		
	cc	rs	tps	cc	rs	tps	cc	rs	tps
6	88,6	83,2	81,0	89,0	83,8	81,2	88,8	82,3	80,5
7	30,6	0,2	-0,8	30,7	0,0	7,7	28,3	0,4	6,7
Iid	98,8	2,5	61,3	97,6	-0,5	65,4	98,0	5,0	66,7
6ps	2,9	8,5	7,3	3,6	7,3	7,7	3,7	8,4	10,0
7ps	33,2	8,7	13,7	33,6	7,3	15,4	31,6	8,8	15,0
Iidps	98,7	12,3	67,8	97,6	8,3	65,4	98,0	13,4	66,7

	RLFAP-4			RLFAP-5			RLFAP-11		
	cc	rs	tps	cc	rs	tps	cc	rs	tps
6	73,0	65,5	48,0	77,5	81,5	65,9	79,6	93,9	83,7
7	12,0	0,0	-4,2	21,7	-1,3	1,8	18,8	-1,8	4,0
Iid	87,4	20,0	58,4	89,0	7,2	64,9	99,1	-7,6	31,7
6ps	70,1	75,6	66,7	14,4	17,5	17,5	3,0	30,4	25,6
7ps	73,8	75,6	66,7	30,1	17,0	17,5	21,4	29,8	28,6
Iidps	91,9	80,0	79,2	90,2	27,4	70,2	99,1	29,1	55,5

FIG. 6.12 - *Pourcentages de gain réalisés par MAC-6 par rapport à MAC-4 et par les algorithmes MAC-7, MAC-Iid, MAC-6ps, MAC-7ps, MAC-Iidps par rapport à MAC-6 pour les problèmes d'affectation de fréquences.*

Cette fois-ci MAC-4 est très largement battu. On peut également voir que le regroupement des contraintes identiques permet d'améliorer fortement les algorithmes. Par ailleurs, dès que le problème devient difficile l'introduction du traitement particulier pour les domaines qui deviennent des singletons s'avère très utile, car cela amène une très forte réduction du nombre d'appel à la fonction EXISTESUPPORT. De plus, la combinaison de cette heuristique avec la prise en compte des propriétés des contraintes (MAC-7ps, MAC-Iidps) permet d'obtenir des gains remarquables.

Jusqu'à présent MAC-4 était le seul algorithme maintenant la consistance d'arc pendant la recherche publié. Pour les RLFAP, MAC-Iidps l'améliore[5] pour les instances 1, 2, 3, 4 ,5 et 11 respectivement de 94%, 93%, 94%, 90%, 90% et 93%. Ces gains sont considérables.

6.5 Conclusion et Perspectives

Dans ce chapitre nous avons défini les principes d'un algorithme maintenant une procédure de filtrage pendant une procédure de recherche. Nous avons montré qu'un tel algorithme doit filtrer le réseau après toute augmentation d'une instanciation partielle mais aussi après n'importe quelle réfutation. Cela nous a conduit à redéfinir les différents types d'ordres d'instanciation et à en définir un nouveau. Ensuite nous avons détaillé les modifications qu'il faut apporter aux algorithmes achevant la consistance d'arc afin qu'ils puissent être utilisés pendant une procédure de recherche. Puis nous avons proposé deux nouveaux algorithmes: MAC-7 et MAC-Inférence.

Les expérimentations que nous avons entreprises montrent d'une part que l'algorithme MAC-6, qui n'avait encore jamais été présenté, est bien meilleur que MAC-4, et d'autre part, que plus on introduit des connaissances sur le réseau (prise en compte des propriétés des contraintes et regroupement des contraintes identiques) et plus les résultats obtenus sont bons. L'intérêt du traitement particulier pour les variables dont le domaine devient un singleton a été également mis en évidence.

Les algorithmes que nous avons présentés reviennent lors d'un échec à la dernière variable instanciée. Nous avons mentionné, dans le chapitre 4, l'existence d'algo-

5. Nous rappelons que la version de MAC-4 testée est une version optimisée gagnant plus de 50% sur les versions classiques.

rithmes tentant de déterminer les variables qui sont la cause d'un échec. Ces algorithmes, dits intelligents, sont généralement qualifiés par le terme «backjump». Or, ils ont été testés essentiellement avec FC. Nous pensons qu'il est temps de les coupler avec un algorithme de type MAC. D'ailleurs, très récemment, Prosser a proposé MAC-CBJ [Prosser, 1995]. Mais son algorithme possède plusieurs inconvénients et peut être amélioré. Le lecteur intéressé pourra consulter [Régin, 1995] qui tente de remédier à certains défauts de l'algorithme de Prosser et qui propose quelques résultats expérimentaux.

Chapitre 7

Traitement des contraintes de différence

Comme nous l'avons vu dans les chapitres précédents, plusieurs algorithmes d'AC que ce soit pour des réseaux de contraintes binaires ou n-aires ont été développés. Mais seulement peu d'entre eux tiennent compte du type des contraintes. Mohr et Masini [Mohr and Masini, 1988b] semblent avoir été les premiers à présenter des traitements particuliers réalisant la fermeture par consistance d'arc de réseaux en utilisant la sémantique des contraintes. Récemment Van Hentenryck, Deville et Teng [Van Hentenryck *et al.*, 1992] ont présenté un algorithme de fermeture par consistance d'arc générique (AC-5) qui tire parti des spécificités de certaines contraintes. Ils ont notamment étudié les contraintes binaires monotones et fonctionnelles. Dans ce chapitre nous nous intéresserons à un type particulier de contraintes n-aires. Nous proposons une implémentation efficace de la consistance d'arc généralisée pour ces contraintes et une nouvelle consistance qui combine la consistance d'arc pour les contraintes binaires et la consistance d'arc généralisée pour les contraintes de différence.

Ce chapitre présente et étend les travaux exposés au congrès AAAI [Régin, 1994].

7.1 Introduction

Une contrainte est appelée contrainte de différence si elle est définie sur un sous-ensemble de variables par l'ensemble des n-uplets pour lesquels les valeurs apparais-

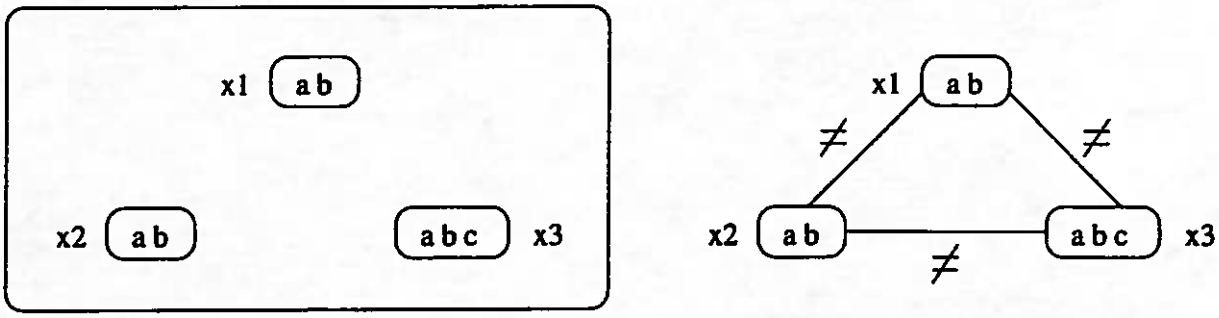
sant dans un même n-uplet sont toutes différentes. Elles sont présentes dans de nombreux problèmes réels et notamment comme nous l'avons vu dans les chapitres 3 et 4, les réseaux de contraintes modélisant les problèmes d'isomorphismes de sous-graphe les impliquent.

Ces contraintes peuvent être représentées comme des contraintes n-aires et filtrées par l'algorithme de fermeture par consistance d'arc généralisée GAC-4 [Mohr and Masini, 1988a]. Ce filtrage réduit efficacement les domaines mais sa complexité est très élevée, car elle dépend de la longueur et du nombre de n-uplets admissibles des contraintes. Pour une contrainte de différence définie sur un ensemble de p variables prenant leurs valeurs dans un ensemble de cardinalité d, ce nombre correspond au nombre d'arrangements de p éléments parmi d: $A_p^d = \frac{d!}{(d-p)!}$. C'est pourquoi de nombreux systèmes de résolution de contraintes, comme CHIP [Van Hentenryck, 1989], représentent ces contraintes n-aires par un ensemble de contraintes binaires. C'est d'ailleurs la méthode que nous avions proposée en 4.2 pour coder Iso-ssgrpartiel par un réseau de contraintes. Dans ce cas, une contrainte de différence binaire est construite pour chaque paire de variables appartenant à la même contrainte de différence. Mais comme nous l'avons remarqué en 4.4.2, l'efficacité de la fermeture par consistance d'arc, en terme de valeurs supprimées des domaines, est faible pour ce type de contrainte. En fait, pour une contrainte de différence entre deux variables x_i et x_j, la fermeture par consistance d'arc supprime une valeur du domaine de x_i uniquement dans le cas où le domaine de x_j se réduit à une seule valeur. Nous avons d'ailleurs présenté en 4.4.2 une version de l'algorithme d'AC tenant compte de cette particularité.

Par ailleurs, l'efficacité des filtrages n'est pas la même selon que l'on emploie une représentation binaire ou n-aire. Pour mettre cela en évidence, supposons que nous ayons un réseau de contraintes comportant 3 variables x_1, x_2 et x_3 et une contrainte de différence entre ces variables (voir figure 7.1).

Les domaines des variables sont $D_1 = \{a, b\}$, $D_2 = \{a, b\}$ et $D_3 = \{a, b, c\}$. Le filtrage GAC-4 avec la contrainte de différence représentée par une unique contrainte ternaire supprime les valeurs a et b du domaine de x_3, tandis que la fermeture par consistance d'arc avec la contrainte de différence représentée par trois contraintes binaires n'élimine aucune valeur.

Dans ce chapitre, nous allons présenter une implémentation efficace, pour les



Représentation par une contrainte ternaire

Représentation par 3 contraintes binaires

FIG. 7.1 - *Deux représentations possibles pour une contrainte de différence.*

contraintes de différence, de la fermeture par consistance d'arc généralisée pour bénéficier de ses performances d'élagage. Sa complexité en espace est en $O(pd)$ et sa complexité en temps en $O(p^2d^2)$ pour une contrainte de différence définie sur p variables prenant leurs valeurs dans un ensemble de cardinalité d.

Cette implémentation utilise le lien étroit, dont nous avons parlé dans le chapitre 2, entre la notion de couplage couvrant un ensemble et la notion d'injection. Brièvement nous pouvons rappeler ce dont il s'agit. Considérons une contrainte de différence définie sur un ensemble $X = \{x_1, ..., x_6\}$, dont les domaines sont donnés par la figure 7.2 et appelons $D(X)$ l'union de ces domaines, puis construisons le graphe biparti $GV = (X, D(X), E)$, que nous nommerons graphe des valeurs de la contrainte, où (x_i, a) est une arête si et seulement si a appartient à D_i.

Nous savons (cf 3.1) que les variables ne peuvent prendre dans leur ensemble des valeurs deux à deux différentes que s'il existe un couplage couvrant X dans GV. A tout n-uplet d'une contrainte de différence correspond donc un couplage et réciproquement. Aussi, une contrainte de différence sera consistante s'il existe au moins un couplage X dans $D(X)$ pour son graphe des valeurs. Nous montrerons, par la suite, que la présence d'une valeur dans un n-uplet de la contrainte correspond à l'appartenance d'une arête à au moins un couplage de X dans $D(X)$. La fermeture par consistance d'arc généralisée se traduira alors par la suppression des arêtes qui n'appartiennent à aucun couplage de ce type.

La section 2 donne certaines définitions et propose une version réduite de la consistance d'arc généralisée qui ne concerne que les contraintes de différence: la

X={x1,x2,x3,x4,x5,x6}
Dx1={1,2}
Dx2={2,3}
Dx3={1,3}
Dx4={2,4}
Dx5={3,4,5,6}
Dx6={6,7}

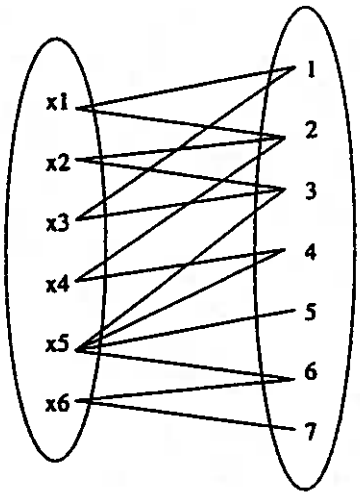


FIG. 7.2 - *Une contrainte de différence définie sur un ensemble X et son graphe des valeurs.*

diff-arc-consistance. La section 3 présente une nouvelle condition garantissant la diff-arc-consistance d'un réseau de contraintes. La section 4 exhibe une méthode efficace permettant de calculer la fermeture par diff-arc-consistance. Dans la section 5 nous montrons comment cette consistance peut être combinée avec d'autres consistances. En section 6 nous étudions sa complexité. Nous donnons, en section 7, un point de vue mathématique de notre filtrage. Puis, en section 8, nous présentons quelques applications possibles de ce filtrage. En section 9, nous introduisons une nouvelle consistance qui est une combinaison de la diff-arc-consistance et de la consistance d'arc pour les contraintes binaires. Certains résultats expérimentaux obtenus pour résoudre lso-ssgrpartiel sont exposés en section 10. Finalement, en section 11 nous concluons.

7.2 Définitions

Rappelons tout d'abord la définition de la consistance d'arc généralisée, c'est-à-dire quelque soit l'arité des contraintes (cf [Jégou, 1991; Mohr and Masini, 1988a]) :

Définition 35 *Étant donné $\mathcal{R} = (X, \mathcal{D}, C)$, un domaine D_i vérifie la* ***consistance d'arc*** *si et seulement si l'on a: $D_i \neq \varnothing$ et $\forall a \in D_i, \forall C_j \in C$ contraignant x_i, $a \in C_{j/\{x_i\}}$.*

Une valeur a dans le domaine d'une variable x_i est donc consistante avec une contrainte n-aire donnée s'il existe une valeur pour toutes les autres variables de la contrainte telle que ces valeurs et a ***satisfassent ensemble et simultanément*** la contrainte.

Nous avons donné la définition d'une contrainte binaire de différence (cf. définition 32), nous présentons cette fois celle pour les contraintes n-aires :

Définition 36 *Une contrainte C est appelée* ***contrainte de différence*** *si et seulement si elle est définie sur un sous-ensemble de variables $X_C = \{x_{i_1}, ..., x_{i_k}\}$ par l'ensemble de n-uplets, noté n-uplet(C) tel que :*
n-uplet(C) $= D_{i_1} \times ... \times D_{i_k} \backslash \{(a_1, ..., a_k) \in D_{i_1} \times ... \times D_{i_k}$ tel que $\exists u, v$ avec $a_u = a_v\}$.

A partir des deux précédentes définitions nous proposons une consistance d'arc restreinte aux contraintes de différences.

Définition 37 *Soient un réseau de contraintes $\mathcal{R} = (X, \mathcal{D}, C)$, avec $C_{\neq} \subseteq C$ l'ensemble des contraintes de différence qu'il contient. $\mathcal{R}$ est* ***diff-arc-consistant*** *si et seulement si le réseau de contraintes de différence $\mathcal{R}_{\neq} = (X, \mathcal{D}, C_{\neq})$ vérifie la consistance d'arc.*

Pour pouvoir utiliser la théorie des couplages nous avons besoin d'une nouvelle représentation (cf. [Laurière, 1976]) :

Définition 38 *Soient une contrainte de différence $C_{\neq}$ définie sur l'ensemble de variables X_C, et $D(X_C)$ l'union des domaines des variables de $X_{C_{\neq}}$, le graphe biparti $GV(C_{\neq}) = (X_{C_{\neq}}, D(X_{C_{\neq}}), E)$ où $\{x_i, a\} \in E \Leftrightarrow a \in D_i$, est appelé* ***graphe des valeurs*** *de $C_{\neq}$.*

7.3 Fermeture par diff-arc-consistance

Le théorème suivant établit un lien entre la diff-arc-consistance et la notion de couplage dans le graphe des valeurs des contraintes de différence.

Théorème 2 *Un réseau de contrainte $\mathcal{R} = (X, \mathcal{D}, C)$ est diff-arc-consistant si et seulement si pour chaque contrainte de différence $C_{\neq}$ de C toute arête de $GV(C_{\neq})$ appartient à un couplage couvrant $X_{C_{\neq}}$ dans $GV(C_{\neq})$.*

preuve :

$\Rightarrow$: Considérons une contrainte de différence $C_{\neq}$ et son graphe des valeurs $GV(C_{\neq})$. A partir de chaque n-uplet admissible de $C_{\neq}$ un ensemble de paires peut être construit, une paire est constituée de la variable et de la valeur qui lui est assignée dans le n-uplet. Pour un n-uplet donné, une variable appartient à une paire. De plus, il existe une paire et une seule contenant chaque variable. Cet ensemble correspond à un ensemble A d'arêtes de $GV(C_{\neq})$ tel que n'importe quel sommet de $X_{C_{\neq}}$ soit l'extrémité d'une et une seule de ces arêtes. Comme $\mathcal{R}$ est diff-arc-consistant, les valeurs de chaque n-uplet sont deux à deux différentes, donc deux arêtes de A ne peuvent pas avoir une extrémité commune et A est un couplage couvrant $X_{C_{\neq}}$ dans $GV(C_{\neq})$. De plus chaque valeur de chaque variable d'une contrainte appartient au moins à un n-uplet. Par conséquent, tout arête de $GV(C_{\neq})$ appartient au moins à un couplage couvrant $X_{C_{\neq}}$.
$\Leftarrow$: Considérons une variable x_i et une valeur a de son domaine. Pour chaque contrainte de différence $C_{\neq}$, la paire $\{x_i, a\}$ appartient au moins à un couplage couvrant $X_{C_{\neq}}$ dans $GV(C_{\neq})$. Comme dans un couplage les arêtes sont deux à deux disjointes, il existe des valeurs pour toutes les autres variables de la contrainte telle que ces valeurs et a satisfassent ensemble et simultanément la contrainte, donc $\mathcal{R}$ est diff-arc-consistant.

L'utilisation de la théorie des couplages est intéressante car, comme nous l'avons vu en 3.1, le calcul d'un couplage maximum peut être réalisé en temps polynômial.

Le théorème 2 nous donne un moyen efficace pour manipuler une contrainte de différence dans un réseau de contraintes. Ainsi, une telle contrainte peut être représentée par son graphe des valeurs, soit avec une complexité en espace de l'ordre de $O(pd)$, si la contrainte est définie sur p variables pouvant prendre leurs valeurs dans un ensemble de d éléments.

Il permet aussi de définir une implémentation efficace de la fermeture par diff-arc-consistance :

Proposition 7 *Soient un CN $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ et $C_{\neq}$ une contrainte de différence. La fermeture par diff-arc-consistance de $\mathcal{R}$ est équivalente à la suppression de toutes*

les arêtes de $GV(C_{\neq})$ qui n'appartiennent à aucun couplage de $X_{C_{\neq}}$ dans $D(X_{C_{\neq}})$. La suppression d'une arête (x_i, a) se traduisant par la suppression de la valeur a du domaine D_i.

Nous pouvons donc proposer un nouvel algorithme (algorithme 31) pour filtrer les domaines des variables de l'ensemble sur lequel une contrainte de différence est définie. Nous considérerons dans la suite que la fonction CALCULCOUPLAGEMAX(G) retourne un couplage maximum du graphe G.

Algorithme 31 Filtrage d'une contrainte de différence.

```
FILTRECONTRAINTEDIFFÉRENCE(i C≠ o AI) : booléen
  construire GV(C≠) le graphe des valeurs de la contrainte C≠)
  M ← CALCULCOUPLAGEMAX(GV(C≠))
  si |M| < |X_C≠| alors retourner faux
  sinon
  | SUPPRIMERARÊTESINUTILISABLES(GV(C≠), M, AI)
  └ retourner vrai
```

Cet algorithme construit le graphe des valeurs de la contrainte de différence, puis recherche un couplage maximum dans ce graphe. Si ce couplage ne couvre pas l'ensemble $X_{C_{\neq}}$ des variables sur lequel la contrainte est définie alors la contrainte ne peut pas être satisfaite et le réseau n'a pas de solution, sinon il élimine les arêtes du graphe des valeurs qui n'appartiennent à aucun couplage couvrant $X_{C_{\neq}}$. La figure 7.3 montre l'application de ce nouveau filtrage pour une contrainte de différence.

Nous allons maintenant expliquer comment de telles arêtes peuvent être supprimées.

7.4 Suppression d'arêtes n'appartenant à aucun couplage couvrant X

Nous considérerons dans la suite, pour simplifier l'exposé, que l'on a un graphe biparti $G = (X, Y, E)$, plutôt que le graphe $GV(C_{\neq}) = (X_{C_{\neq}}, D(X_{C_{\neq}}), E)$, et un couplage M de X dans Y.

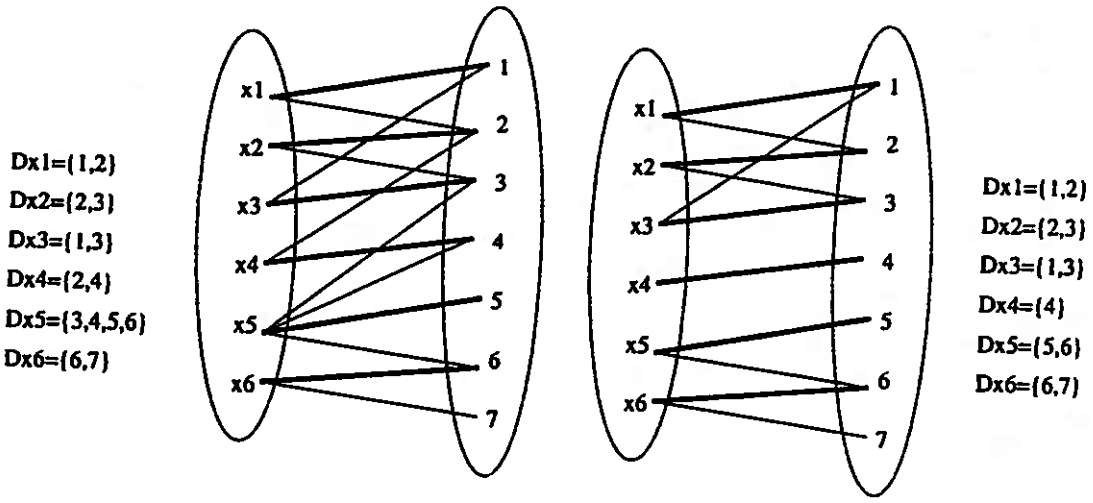


FIG. 7.3 - *Un couplage couvrant X et le résultat du filtrage.*

Afin de comprendre comment on peut supprimer efficacement toutes les arêtes qui n'appartiennent à aucun couplage, nous présentons quelques définitions de la théorie des couplages. Le lecteur particulièrement intéressé pourra consulter le livre de [Berge, 1970] ou celui de Lovász et Plummer [Lovász and Plummer, 1986].

Définition 39 *Soit M un couplage d'un graphe :*

• *un sommet x est* ***saturé*** *par le couplage M ou* ***M-saturé*** *s'il existe une arête (x, a) appartenant à M, sinon x est* ***insaturé*** *par M ou* ***M-insaturé.***

• *une chaîne simple (c'est à dire n'utilisant pas deux fois la même arête) (resp. un cycle simple) dont les arêtes sont alternativement dans M et dans $E - M$ est appelée* ***chaîne alternée*** *(resp.* ***cycle alterné****) relative à M.*

La figure 7.4 donne un exemple d'un couplage de X dans Y pour un graphe biparti. Les arêtes en gras sont celles du couplage. Le sommet 7 est insaturé par le couplage. La chaîne (7,x6,6,x5,5) est une chaîne alternée commençant par un sommet insaturé. Le cycle (1,x3,3,x2,2,x1,1) est un cycle alterné. Les arêtes en pointillés très fins n'appartiennent à aucun couplage couvrant X.

Berge a introduit un théorème très important qui a permis de rechercher en un temps polynômial un couplage maximum dans un graphe.

Théorème 3 ([Berge, 1957]) *Un couplage M est maximum si et seulement si il n'existe pas de chaîne alternée reliant un sommet insaturé à un autre sommet insaturé*

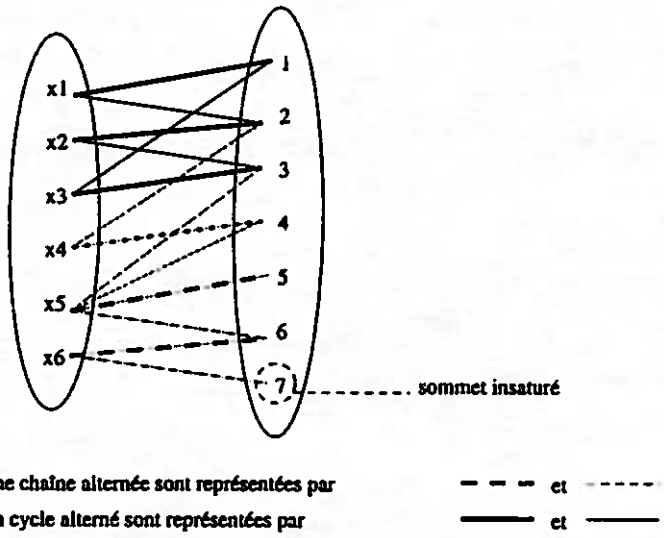


FIG. 7.4 - *Un couplage couvrant X, un cycle alterné et une chaîne alternée.*

Nous pouvons donner les idées d'un premier algorithme pour rechercher les arêtes qui n'appartiennent pas à un couplage couvrant X.

1. On commence par rechercher un couplage M couvrant X. S'il n'en existe pas alors aucune arête n'appartient à un couplage couvrant X et on met toutes les arêtes dans une liste LS; sinon on place toutes les arêtes du graphe dans une liste LA et on initialise LS à vide.

2. On supprime de LA toutes les arêtes qui appartiennent à M.

3. Tant que LA n'est pas vide, on choisit une arête $e = \{x_i, a\}$ de LA, on supprime du graphe initial ses deux extrémités puis on recherche s'il existe un couplage couvrant $X - \{x_i\}$ pour le nouveau graphe. Si c'est le cas on retourne en 2. Sinon on place e dans la liste LS.

À la fin de cet algorithme, la liste LS contient toutes les arêtes qui n'appartiennent à aucun couplage couvrant X. Mis à part l'étape 1, l'algorithme aura toujours la possibilité de calculer l'existence d'un couplage couvrant X avec un couplage de cardinalité $|X| - 1$. Ce calcul peut-être réalisé en $O(m)$, où m désigne le nombre d'arêtes du graphe biparti. Dans le pire des cas la complexité de cet algorithme est donc en $O(m^2)$. Dans la suite de cette section nous allons proposer un algorithme ayant une complexité en $O(m)$.

À partir du théorème 3, Berge a démontré le corollaire suivant :

Corollaire 1 ([Berge, 1970]) *Appelons **libre** une arête qui appartient à un couplage maximum mais non à tous. Une arête e est libre si et seulement si, pour un couplage maximum M arbitraire, elle appartient soit à une chaîne alternée paire commençant par un sommet insaturé, soit à un cycle alterné pair.*

Ce corollaire nous intéresse particulièrement puisqu'il permet, à partir de M, de rechercher toutes les arêtes qui n'appartiennent à aucun couplage de X dans Y. En effet ce sont celles qui n'appartiennent ni à M, ni à une chaîne alternée paire commençant par un sommet insaturé, ni à un cycle alterné pair. Nous nommerons ***vitale*** une arête qui appartient à tous les couplages de X dans Y. L'arête $(x4, 4)$ du graphe de la figure 7.4 est une arête vitale.

Proposition 8 *Soit $G_O = (X, Y, Succ)$ le graphe biparti orienté défini à partir de G et de l'orientation :*

- $\forall x \in X : Succ(x) = \{y \in Y \,/\, (x, y) \in M\}$
- $\forall y \in Y : Succ(y) = \{x \in X \,/\, (x, y) \notin M\}$

Alors on a les deux propriétés suivantes :

1) *A tout circuit élémentaire de G_O correspond un cycle alterné pair de G et réciproquement.*

2) *A tout chemin élémentaire commençant par un sommet M-insaturé de G_O correspond une chaîne alternée paire commençant par un sommet M-insaturé de G et réciproquement.*

preuve :

Il est évident que si l'on ne tient pas compte de la parité la proposition est vraie. Dans le premier cas comme G est biparti il n'admet pas de cycle impair. Dans le second cas nous devons montrer qu'à tout chemin élémentaire commençant par un sommet M-insaturé de G_O correspond une chaîne alternée paire commençant par un sommet M-insaturé de G. Considérons un tel chemin élémentaire. Si ce chemin est de longueur pair alors il lui correspond de manière évidente une chaîne alternée paire commençant par un sommet M-insaturé. Supposons que ce chemin soit de

longueur impaire. M est un couplage de X dans Y il n'existe donc pas de sommets de X insaturés. Comme le graphe est biparti et puisque l'on débute avec un sommet insaturé, donc de Y, ce chemin élémentaire se termine par un sommet de X. Le chemin étant élémentaire son extrémité dans X possède un successeur qui n'appartient pas au chemin, car tout sommet de X a un successeur et un sommet de Y ne peut être atteint que par un et un seul sommet puisqu'il ne possède qu'un seul prédécesseur. Donc à partir d'un chemin élémentaire de longueur impaire on pourra toujours construire un chemin élémentaire de longueur paire, auquel il correspond une chaîne alternée de longueur paire.⊙

De cette proposition nous pouvons déduire l'algorithme recherché (cf. algorithme 32).

Algorithme 32 Suppression des arêtes n'appartenant à aucun couplage couvrant X

SUPPRIMERARÊTESINUTILISABLES(i $GV(C_{\neq}), M$ o AI)

1 Marquer tous les arcs de G_O comme "non atteints". Initialiser AI à l'ensemble vide

2 Parcourir à partir des sommets M-insaturés G_O en largeur d'abord, en n'empruntant uniquement que des arcs marqués "non atteints". Marquer un arc par "atteint" quand il est traversé

3 Calculer les composantes fortement connexes de G_O en n'empruntant que les arcs qui sont marqués "non atteints". Marquer "atteints" tout arc appartenant à une composante fortement connexe.

4 **pour** *chaque arc* e_O *marqué "non atteint"* **faire**
  Mettre dans e l'arête correspondant à e_O
  si $e \in M$ **alors** marquer e comme "vital" **sinon**
    SUPPRIMER(e, G)
    AJOUTER(AI, e)

L'étape 2 correspond à la seconde éventualité de la proposition 8. Elle peut en plus atteindre des arêtes qui appartiennent à des cycles alternés pairs. L'étape 3 calcule les composantes fortement connexes de G_O car un arc reliant deux sommets appartenant à la même composante fortement connexe, appartient à un circuit et

réciproquement; et d'après l'éventualité 1 de la proposition 8 l'arête correspondante appartient à un cycle alterné de G. Après cette étape, l'ensemble A des arêtes qui appartiennent à un couplage couvrant X mais non à tous est connu. L'ensemble AI des arêtes à supprimer de G est donc: $E - (A \cup M)$, d'où le code de l'étape 4.

Nous avons montré comment pour ***une*** contrainte de différence $C_{\neq}$ toute arête qui n'appartient à aucun couplage couvrant $X_{C_{\neq}}$ peut être supprimée. Mais une variable peut être contrainte par plusieurs contraintes, toute suppression doit donc être propagée. Considérons une variable x_i de $X_{C_{\neq}}$, x_i peut appartenir à plusieurs ensembles sur lesquels des contraintes de différence sont définies, aussi une valeur a de D_i peut être supprimée car elle ne possède pas de support sur une contrainte différente de $C_{\neq}$. Cette suppression nécessite de supprimer l'arête (x_i, a) du graphe $GV(C_{\neq})$. Donc les conséquences de la modification de la structure de ce graphe doivent être déterminées.

7.5 Propagation des suppressions

La suppression de valeurs pour une contrainte de différence peut entraîner de nombreuses modifications pour d'autres contraintes. Parmi ces dernières il peut y avoir des contraintes de différence, on peut pour elles faire mieux que répéter le premier algorithme en utilisant le fait qu'avant les suppressions un couplage couvrant l'ensemble des variables était connu.

L'algorithme de propagation (cf algorithme 33) que nous proposons a deux ensembles comme paramètres (S et AI). Le premier (S) représente l'ensemble des arêtes qui doivent être supprimées du graphe biparti et le second (AI) l'ensemble des arêtes qui seront supprimées par le filtrage. L'algorithme a aussi besoin d'une fonction CALCULCOUPLAGEMAX($GV(C_{\neq}), M$) qui calcule un couplage maximum $GV(C_{\neq})$ à partir d'un couplage M qui ne l'est pas forcément.

L'algorithme est divisé en trois parties. La première supprime les arêtes du graphe biparti et éventuellement du précédent couplage, la seconde calcule éventuellement un nouveau couplage couvrant $X_{C_{\neq}}$ à partir du couplage précédent et la troisième procède au nouveau filtrage. Il retourne faux si une arête vitale appartient à S ou bien s'il n'existe plus de couplage couvrant $X_{C_{\neq}}$, sinon il renvoie vrai.

Algorithme 33 Propagation des suppressions.

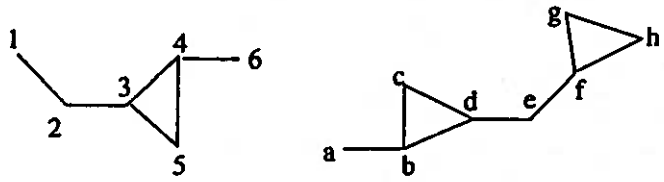
```
DIFFPROPAGATION( i/o GV(C≠), M, S, AI) : booléen
    calculCouplage ← faux
1   pour chaque arête e ∈ S faire
        si e ∈ M alors
            SUPPRIMER(e, M)
            si e est marqué "vital" alors retourner faux
            sinon calculCouplage ← vrai
2   si calculCouplage alors
        M ← CALCULCOUPLAGEMAX(GV(C≠), M)
        si |M| = |X_C≠| alors retourner faux
3   SUPPRIMERARÊTESINUTILISABLES(GV(C≠), M, AI)
    retourner vrai
```

L'annexe B détaille l'association de la consistance d'arc généralisée pour les contraintes de différence et de la consistance d'arc pour les contraintes binaires pour le problème du zèbre, aussi nous nous contenterons, dans cette section, d'illustrer cette association pour une instance de lso-ssgrpartiel.

Reprenons l'exemple utilisé dans le chapitre 3.

La figure 7.5 donne les domaines après application de la fermeture par consistance d'arc pour les contraintes binaires. La figure 7.6 présente le graphe des valeurs immédiatement après la fermeture précédente, un couplage maximum dans ce graphe et le résultat de la suppression des arêtes n'appartenant à aucun couplage saturant toutes les variables.

Puisque certaines valeurs ont été supprimées par le nouveau filtrage, on doit relancer la fermeture par consistance d'arc pour les contraintes binaires. Celle-ci permet d'obtenir l'unique solution du problème: $D_1 = \{f\}, D_2 = \{e\}, D_3 = \{d\}, D_4 = \{b\}, D_5 = \{c\}, D_6 = \{a\}$.



H G

D1=D6={a,b,c,d,e,f,g,h}
D2=D5={b,c,d,e,f,g,h}
D3=D4={b,d,f}

Domaines après fermeture par consistance d'arc pour les contraintes binaires :

D3=D4={b,d}
D2=D5={b,c,d,e}
D6={a,b,c,d,e}
D1={a,b,c,d,e,f}

FIG. 7.5 - lso-ssgrpartiel : *un exemple.*

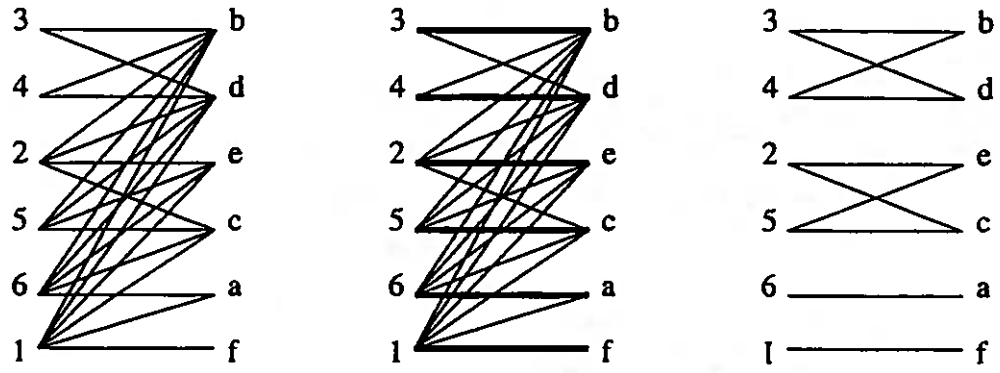


FIG. 7.6 - lso-ssgrpartiel : *graphe des valeurs.*

7.6 Complexité

Suppression des arêtes n'appartenant à aucun couplage couvrant $X_{C_{\neq}}$

La complexité de cet algorithme est la même que celle de la recherche de composantes fortement connexes, soit $O(m+n)$ [Tarjan, 1972], pour un graphe de m arêtes et n sommets. Cette complexité est donc très faible.

Initialisation et Filtrage

La complexité de l'étape 1 est $O(d|X_{C_{\neq}}| + |X_{C_{\neq}}| + |D(X_{C_{\neq}})|)$. L'étape 2 coûte $O(d|X_{C_{\neq}}|\sqrt{|X_{C_{\neq}}|})$. Et nous avons vu précédemment que l'étape 3 pouvait être faite de manière linéaire c'est-à-dire avec la même complexité que l'étape 1. La complexité globale du filtrage pour ***une*** contrainte de différence est donc en $O(d|X_{C_{\neq}}|\sqrt{|X_{C_{\neq}}|})$.

Propagation des suppressions

Nous considérerons dans la suite que G a m arêtes et n sommets.
Supposons que l'on doivent supprimer k arêtes de G ($|S| = k$). L'étape 1 a une complexité en $O(k)$. Pour l'étape 2 on peut être amené à calculer un couplage de X dans Y à partir d'un couplage de cardinalité $|M - k|$. Ce calcul nécessite au plus k étapes[1] chacune en $O(m + n)$. L'étape 3 se fait en $O(m + n)$. Si l'on supprime k arêtes, le nombre d'opérations effectuées par l'algorithme sera donc en $O(k(m + n))$.
On peut être amené à supprimer une à une toutes les arêtes du graphe biparti. La complexité globale du filtrage est $O(m^2)$. Si p est l'arité d'une contrainte de différence et si le domaine de chaque variable contient toutes les valeurs du CSP, alors la complexité du filtrage pour une contrainte de différence est $O(p^2d^2)$. Cette complexité est relativement faible.

1. Il semblerait que l'on puisse atteindre une complexité en $O(\sqrt{k}(m + n))$. Mais cela reste au conditionnel.

7.7 Point de vue mathématique

Nous allons dans cette section montrer d'un point de vue ensembliste à quoi correspond la suppression d'arêtes n'appartenant à aucun couplage. Cela dans l'espoir de donner une piste éventuelle pour obtenir un algorithme ayant une meilleure complexité. Avant tout, nous rappelons le célèbre théorème de König-Hall.

Théorème 4 (König-Hall) *Dans un graphe biparti $B = (X, Y, E)$, on peut coupler X dans Y si et seulement si l'on a :*

$$\forall A \subset X : |\Gamma(A)_B| \geq |A|$$

Ce théorème implique une inégalité au sens large. Nous allons nous intéresser à l'égalité.

Définition 40 *Dans un graphe biparti $B = (X, Y, E)$, nous appellerons **gourmand** un ensemble $A \subseteq X$ tel que $|\Gamma(A)_B| = |A|$.*

A partir de cette définition nous pouvons énoncé la propriété suivante :

Propriété 10 *Soient un graphe biparti $B = (X, Y, E)$ et A un ensemble gourmand, alors aucune arête de $X - A$ dans $\Gamma_B(A)$ ne peut appartenir à un couplage de X dans Y.*

preuve :

Le raisonnement se fait par l'absurde.
Supposons qu'il existe un couplage de X dans Y qui contienne une arête entre un sommet x de $X - A$ et un sommet a appartenant à $\Gamma_B(A)$, alors il doit aussi exister un couplage de $X - \{x\}$ dans $Y - \{a\}$ pour le sous-graphe biparti B' de B engendré par les sous-ensembles de sommets $X - \{x\}$ et $Y - \{a\}$. Or pour ce sous-graphe biparti B' on a $\Gamma_{B'}(A) = \Gamma_B(A) - \{a\}$ puisque x n'appartient pas à A et a appartient à $\Gamma_B(A)$. Donc pour B', il existe un ensemble A de sommets tel que $|\Gamma_{B'}(A)| = |A| + 1$ c'est-à-dire $|\Gamma_{B'}(A)| < |A|$, il ne peut donc pas exister de couplage de $X - \{x\}$ dans $Y - \{a\}$ pour ce graphe. Par conséquent aucun couplage de X dans Y ne peut contenir une arête entre un sommet x de $X - A$ et un sommet a appartenant à $\Gamma_B(A)$.

L'avantage de la propriété énoncée ci-dessus est qu'elle ne nécessite pas la présence d'un couplage maximum. Il peut être intéressant de s'assurer que certains ensembles particuliers vérifient cette inéquation. Trivialement, on peut vérifier si X satisfait cette inégalité. D'autres ensembles particuliers peuvent aussi être considérés. Il s'agit des variables qui ne sont pas reliées à des sommets pendants.

Définition 41 *Soit $G = (X, Y, E)$ un graphe biparti, une variable $x \in X$* ***fait vivre une valeur*** *$a \in Y$ si et seulement si a est un sommet pendant ($\Gamma(a) = \{x\}$). Une telle valeur a est dite unique, dans le cas contraire on dira qu'elle est multiple.*

Nous noterons X_{FV} l'ensemble des variables qui font vivre au moins une valeur et Y_M l'ensemble des valeurs multiples.

Lemme 1 *Soit $G = (X, Y, E)$ un graphe biparti, s'il existe un couplage de X dans Y alors $|X| \leq |Y_M| + |X_{FV}|$.*

preuve :

Supposons qu'il existe un couplage de X dans Y vérifiant $|X| > |Y_M| + |X_{FV}|$. On a alors $|X - X_{FV}| > |Y_M|$, or $|Y_M| \geq |\Gamma(X - X_{FV})|$ puisque les variables de $X - X_{FV}$ ne sont reliées à aucune valeur unique. D'où $|X - X_{FV}| > |\Gamma(X - X_{FV})|$ ce qui est en contradiction avec le théorème de König-Hall.

Proposition 9 *Soit $G = (X, Y, E)$ un graphe biparti, si $|X| = |Y_M| + |X_{FV}|$ alors les arêtes dont une extrémité appartient à X_{FV} et l'autre à Y_M ne peuvent pas appartenir à un couplage de X dans Y.*

preuve :

Soit (x, a) une arête de G avec $x \in X_{FV}$ et $a \in Y_M$, considérons $G' = (X', Y', E')$ le sous graphe de G induit par $X' = X - \{x\}$ et $Y' = Y - \{a\}$. Notons k le nombre de voisins de x dans G, hormis a, de degré 2. Dans G' on a $|X'| = |X| - 1$, $|Y'_M| = |Y_m - k - 1$ et $|X_{FV}| \leq |X_{FV}| - 1 + k$ (c'est une inégalité car certains sommets liés à un voisin de x de degré peuvent déjà faire vivre des valeurs dans G). Dans G nous avons $|X| = |Y_M| + |X_{FV}|$

aussi pour G' on a $|X'| + 1 \geq |Y'_M| + k + 1 + |X'_{FV}| + 1 - k$, c'est-à-dire $|X'| \geq |Y'_M| + |X'_{FV}| + 1$ et donc $|X'| > |Y'_M| + |X'_{FV}|$. D'après le lemme 1 cela signifie qu'il n'existe pas de couplage de X' dans Y' pour le graphe G'. L'arête (x, a) n'appartient donc à aucun couplage de X dans Y pour G.

Cette proposition s'applique pour chaque composante connexe du graphe biparti. Elle permet d'obtenir un algorithme de préfiltrage qui s'avère souvent efficace en pratique car il est très peu coûteux. De plus les variables qui font vivre des valeurs peuvent ne pas être prises en considération pour la recherche de couplages maximum. La mise en évidence de telles variables est donc bénéfique pour la fermeture par consistance d'arc généralisée.

7.8 Autres applications possibles de la consistance d'arc généralisée

Nous allons voir dans cette section d'autres raisons pouvant justifier l'utilisation de ce filtrage et ce même pour des contraintes qui ne sont pas exactement des contraintes de différences. Tout d'abord, rappelons que la recherche d'une injection[2] définie sur un ensemble de variables peut être vue comme une contrainte de différence (cf chapitre 2).

7.8.1 Contraintes plus spécifiques que la différence

Nous avons proposé une implémentation efficace de la consistance d'arc généralisée pour les contraintes de différence. Cette implémentation peut être utilisée pour des contraintes qui sont plus spécifiques que les contraintes de différence :

Définition 42 *Une contrainte C est **plus spécifique** qu'une contrainte de différence $C_{\neq}$ si et seulement si elle est définie sur le même sous-ensemble de variables $X_{\neq} = \{x_{i_1}, ..., x_{i_k}\}$ que $C_{\neq}$ par l'ensemble de n-uplets, noté n-uplet(C) tel que : n-uplet(C) $\subseteq$ n-uplet($C_{\neq}$).*

2. Attention, il ne s'agit pas ici de contraintes fonctionnelles entre les domaines des variables.

Dans ce cas nous ne pouvons plus garantir que l'algorithme que nous avons proposé calcule la fermeture par consistance d'arc généralisée pour ces contraintes. Cependant, il peut toujours être appliqué.

7.8.2 Nouvelle modélisation de certaines problèmes

L'annexe 2 présente en détail une nouvelle modélisation pour le problème du zèbre. Dans cette section nous nous limiterons à deux problèmes: celui des n-dames et Iso-ssgrpartiel.

Par la suite nous noterons: TOUSDIFFÉRENTS($\{x_{i_1}, ..., x_{i_j}\}$) une contrainte de différence définie sur l'ensemble de variables $\{x_{i_1}, ..., x_{i_j}\}$.

Iso-ssgrpartiel

Nous avons présenté en 4.2.1 une modélisation possible de ce problème. Nous sommes maintenant en mesure d'en présenter une autre plus concise. Soient $G = (X_G, E_G)$ et $H = (X_H, E_H)$ deux graphes et supposons que l'on veuille résoudre Iso-ssgrpartiel(H, G). On utilise pour ce faire le réseau de contraintes $\mathcal{R} = (X_H, \mathcal{D}, \mathcal{C})$ avec:

- X_H est l'ensemble des sommets de H.
- Le domaine de chaque variable h est constitué par l'ensemble des sommets de G dont le degré est supérieur ou égal à celui de h
- A chaque arête (i, j) de H correspond une contrainte binaire C_{ij} telle que $C_{ij}(a, b)$ est vrai si et seulement si $(a, b) \in E_G$.
- il existe une contrainte de différence impliquant tous les sommets de X_H: TOUSDIFFÉRENTS(X_H).

Le problème des n-dames

Le problème des n-dames consiste à placer n dames sur un échiquier $n \times n$ sans qu'aucune d'elles ne puisse en prendre une autre. Rappelons que selon les règles du jeu

d'échecs, une dame placée sur case contrôle la ligne, la colonne et les deux diagonales auxquelles appartient la case. On peut modéliser ce problème de différentes façons.

La plus classique consiste à utiliser autant de variables qu'il y a de dames. Une variable représente une ligne de l'échiquier et peut prendre n valeurs qui correspondent aux colonnes de l'échiquier. $n(n-1)/2$ contraintes binaires sont créées. Elles expriment que les valeurs prises, pour tout couple de variables, doivent correspondre à des colonnes différentes mais également à des diagonales différentes :

$$C_{ij}(a,b) \text{ est vrai} \Leftrightarrow a \neq b \text{ et } i+a \neq j+b \text{ et } i-a \neq j-b$$

L'inconvénient de cette méthode est que le nombre de contraintes impliquées est très important.

Une autre solution est d'utiliser $3n$ variables réparties en trois groupes $\{x1_1, ..., x1_n\}$, $\{x2_1, ..., x2_n\}$, $\{x3_1, ..., x3_n\}$. Les variables des deux derniers groupes sont reliées au premier groupe de la manière suivante : $\forall i : x2_i = x1_i + i, x3_i = x1_i - i$. Cette représentation induit $2n$ contraintes. Les variables du premier groupe prennent leurs valeurs dans l'intervalle $[1, n]$, celles du second dans $[2, 2n]$ et celles du troisième dans $[1-n, n-1]$. Ensuite on utilise 3 contraintes de différences :

• TOUSDIFFÉRENTS($\{x1_1, ..., x1_n\}$). Cette contrainte exprime que deux dames ne peuvent pas être placées sur la même colonne.

• TOUSDIFFÉRENTS($\{x2_1, ..., x2_n\}$). Cette contrainte signifie que deux dames ne peuvent pas appartenir à la même diagonale montant vers la droite.

• TOUSDIFFÉRENTS($\{x2_1, ..., x2_n\}$). Cette contrainte implique que deux dames ne peuvent pas appartenir à la même diagonale montant vers la gauche.

Cette représentation n'implique que $2n$ contraintes binaires et 3 contraintes n-aires. Elle donne en pratique de meilleurs résultats que la précédente [Leconte, 1995].

Le problème Golomb

Leconte [Leconte, 1995] a proposé de tester l'algorithme achevant la consistance d'arc généralisé pour le problème Golomb.

Il s'agit de trouver un ensemble de variables représentant les graduations d'une règle tel que les différences entre deux graduations soient toutes distinctes et tel que la longueur de la règle soit minimale. Le problème peut se modéliser de la façon suivante :

	temps sans ACGDiff	temps avec ACGDiff	gain
Golomb 7	0,53	0,57	0%
Golomb 8	4,69	3,66	22%
Golomb 9	45,36	26,51	42%
Golomb 10	404,4	141,70	53%
Golomb 11	8511	3606	58%

FIG. 7.7 - *Résultats obtenus par M. Leconte pour le problème Golomb avec ILOG Solver. ACGDiff signifie consistance d'arc généralisée pour les contraintes de différence.*

Étant donné n variables $0 = x_1 < ... < x_i < ... < x_n$ telles que TOUSDIFFÉRENTS($\{x_i - x_j; 0 < j < i \leq n\}$), trouver la valeur minimale de x_n.

Ce problème se rencontre en astronomie, lorsqu'il faut calculer les positions relatives de plusieurs radio-téléscopes. Une solution optimale pour 5 variables est: 0,1,4,9,11. Pour améliorer la résolution du problème on introduit généralement la contrainte suivante: $x_2 - x_1 < x_n - x_{n-1}$. Pour être complet, mentionnons qu'à notre connaissance Golomb 17 n'est toujours pas connu. La solution optimale[3] de Golomb 16 est 0,1,4,11,26,32,56,68,76,115,117,134,150,163,168,177.

Leconte a implémenté dans ILOG Solver [Puget, 1994] le traitement que nous proposons. Les résultats qu'il a obtenus sont donnés en figure 7.7.

Le gain réalisé par notre algorithme croit en fonction de n. Plus le problème est difficile et meilleures sont ses performances par rapport à un algorithme ne calculant pas la fermeture par consistance d'arc généralisée.

Contraintes du type «au moins k valeurs différentes»

La contrainte «au moins k valeurs différentes» est une contrainte n-aire exprimant que parmi les valeurs prises par un ensemble de n variables il doit y en avoir au moins k différentes. Si k vaut n alors la contrainte est une contrainte de différence.

3. Cette solution nous a été communiquée par J. Shearer du «Department of Mathematical Sciences, IBM T.J. Watson Research Center».

La théorie des couplages va encore une fois nous permettre de réaliser la fermeture par consistance d'arc généralisée pour ce type de contraintes. A l'aide d'un raisonnement similaire à celui que nous avons fait pour les contraintes de différence, nous pouvons énoncer le théorème suivant :

Théorème 5 *Une contrainte C du type «au moins k valeurs différentes» définie sur un ensemble X vérifie la consistance d'arc généralisée si et seulement si toute arête de $GV(C)$ appartient à un couplage de taille k dans $GV(C)$.*

Le calcul de la fermeture par consistance d'arc généralisée pour ce type de contraintes se réalise efficacement grâce à la proposition suivante :

Proposition 10 *Soit $G = (X, Y, E)$ un graphe biparti, s'il existe un couplage M de cardinalité k alors toutes les arêtes de G appartiennent à au moins un couplage de taille $k-1$.*

preuve :

Considérons une arête $\{x_i, a\}$ de G et M un couplage avec $|M| = k$. Nous allons examiner tous les cas possibles :

a) $\{x_i, a\} \in M$: la proposition est évidemment vérifiée.

b) $\nexists b, \{x_i, b\} \in M$ et $\nexists x_j, \{x_j, a\} \in M$: $M \cup \{\{x_i, a\}\}$ est un couplage de taille $k+1$, la proposition est donc vérifiée.

c) $\exists b, \{x_i, b\} \in M$ et $\nexists x_j, \{x_j, a\} \in M$: $M - \{\{x_i, b\}\} \cup \{\{x_i, a\}\}$ est un couplage de taille k, la proposition est donc vérifiée.

d) $\nexists b, \{x_i, b\} \in M$ et $\exists x_j, \{x_j, a\} \in M$: $M - \{\{x_j, a\}\} \cup \{\{x_i, a\}\}$ est un couplage de taille k, la proposition est donc vérifiée.

e) $\exists b, \{x_i, b\} \in M$ et $\exists x_j, \{x_j, a\} \in M$: $M - \{\{x_i, b\}, \{x_j, a\}\} \cup \{\{x_i, a\}\}$ est un couplage de taille $k-1$, la proposition est donc vérifiée.

Donc, pour supprimer les arêtes qui n'appartiennent à aucun couplage de taille k, il suffit de rechercher s'il existe un couplage dont la taille est strictement supérieure à k. Si c'est le cas, alors la contrainte vérifie la consistance d'arc généralisée. S'il n'existe pas de couplage dont la taille est supérieure ou égale à k, alors la contrainte n'est

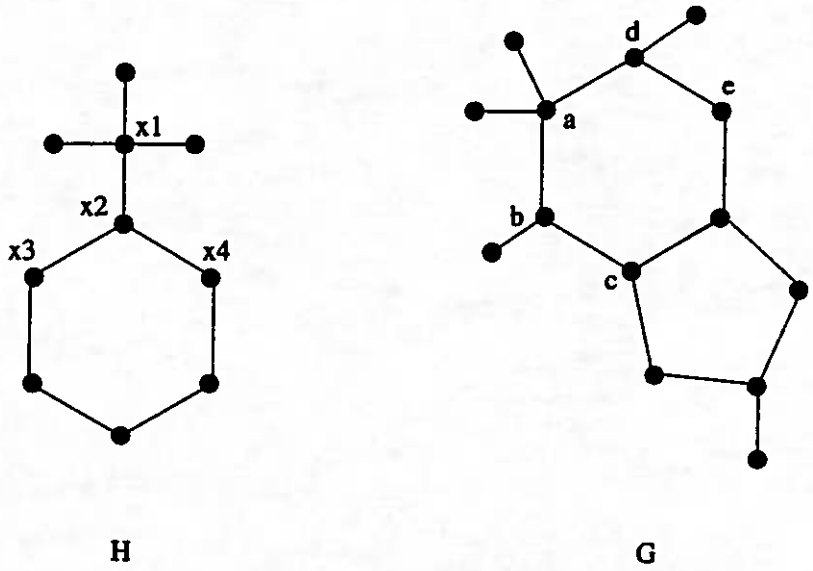


FIG. 7.8 - *Recherche d'un morphisme injectif de H dans G.*

pas consistante. Enfin si le couplage maximum a une cardinalité égale à k alors on applique le même raisonnement que celui que nous avons employé pour les contraintes de différence. Il est important de remarquer que certains sommets correspondant à des variables peuvent être insaturés.

La complexité du traitement lié à ces variables est dans le pire des cas la même que celle pour les contraintes de différences.

7.9 Combinaison des contraintes de différence et des contraintes binaires

Nous avons présenté en 7.4 l'association d'un algorithme de fermeture par consistance d'arc généralisée pour les contraintes de différence avec un algorithme réalisant la fermeture par consistance d'arc des contraintes binaires. Nous proposons, dans cette section, de combiner, plutôt que d'associer, ces deux algorithmes.

Avant une présentation plus formelle, nous montrons sur un exemple l'intérêt d'une telle approche.

Supposons que l'on veuille résoudre Iso-ssgrpartiel pour le graphe H et le graphe G donnés en figure 7.8. On s'aperçoit immédiatement que ce problème n'a pas de

solution. Or l'association de la consistance d'arc généralisée pour les contraintes de différence et de la consistance d'arc pour les contraintes d'adjacence ne permet pas de détecter cette inconsistance. Pourtant il apparaît clairement que le domaine de x_1 ne contient que la valeur a et que x_2 ne peut pas être placé sur b car sinon x_3 et x_4 devraient tous les deux être appariés avec c, ce qui est impossible. La consistance d'arc généralisée considère les variables et les valeurs dans leur ensemble, elle ne permet pas de simuler un tel raisonnement. La consistance d'arc pour les contraintes d'adjacence trouve que (x_2, b) est supportée par (x_3, c) et (x_4, c) et s'en contente. Pour améliorer la procédure de filtrage pour Iso-ssgrpartiel il faut prendre en compte la condition suivante :

> «Une valeur a peut appartenir au domaine d'une variable x_i si elle possède sur chaque contrainte d'adjacence reliée à x_i un support qui soit différent de celui qu'elle a sur n'importe quelle autre contrainte d'adjacence impliquant x_i».

Ce type de traitement est très proche de celui utilisé par l'algorithme de résolution de Iso-ssarbre, présenté au chapitre 3.

7.9.1 Une nouvelle consistance d'arc

Nous allons maintenant généraliser cette idée pour des réseaux ne correspondant pas nécessairement à ceux obtenus pour résoudre Iso-ssgrpartiel. Puis nous montrerons comment elle peut, à faible coût, être mise en oeuvre.

Dans la suite nous limiterons notre étude aux réseaux de contraintes $R = (X, \mathcal{D}, \mathcal{C}_b \cup \mathcal{C}_{\neq})$, où :

- $\mathcal{C}_b$ est un ensemble de contraintes binaires;
- $\mathcal{C}_{\neq}$ est un ensemble de contraintes de différence.

Nous noterons $\Gamma_b(x_i)$ l'ensemble des variables x_j telles qu'il existe une contrainte binaire C_{ij} entre x_i et x_j. Nous supposerons dans la suite que s'il existe une contrainte binaire entre C_{ij} et une contrainte de différence impliquant x_i et x_j, alors quelque soit la valeur a, $C_{ij}(a, a)$ est toujours faux.

Définition 43 *Soient un réseau de contraintes $R(X, \mathcal{D}, \mathcal{C}_b \cup \mathcal{C}_{\neq})$ défini comme ci-dessus, une valeur a d'une variable x_i et $D(\Gamma_b(x_i))$ l'union des domaines des variables*

de $\Gamma_b(x_i)$, le graphe biparti $GVS((x_i,a)) = (\Gamma_b(x_i), D(\Gamma_b(x_i)), E)$ où $\{x_j,b\} \in E$ si et seulement si $b \in D_j$ et $C_{ij}(a,b)$ est vrai, est appelé ***graphe des valeurs des supports*** *de (x_i,a).*

Définition 44 *On dit que la valeur b de x_j* ***supporte*** *la valeur a de x_i* ***en respectant une contrainte de différence*** *$C_{\neq}$ définie sur l'ensemble $X_{\neq}$ si et seulement si les deux conditions suivantes sont vérifiées:*

- *$C_{ij}(a,b)$ est vrai;*
- *$x_j \notin X_{\neq}$ ou l'arête $\{x_j,b\}$ appartient à un couplage de $\Gamma_b(x_i) \cap X_{\neq}$ dans $GVS((x_i,a))$ couvrant $\Gamma_b(x_i) \cap X_{\neq}$.*

Définition 45 *Une valeur a d'une variable x_i est* ***viable en respectant une contrainte de différence*** *$C_{\neq}$ si pour toute variable $x_j \in \Gamma_b(x_i)$, a possède un support dans D_j qui respecte $C_{\neq}$.*

On remarque que si une valeur est viable en respectant n'importe quelle contrainte de différence elle est viable selon le sens de ce terme pour la consistance d'arc.

Nous pouvons maintenant définir une nouvelle consistance.

Définition 46 *Un réseau de contraintes $R(X,\mathcal{D},C_b \cup C_{\neq})$ vérifie* ***la consistance d'arc en respectant les contraintes de différence*** *si et seulement si aucun domaine n'est vide et toutes les valeurs de tous les domaines sont viables en respectant toutes les contraintes de différence.*

Propriété 11 *Un réseau vérifiant la consistance d'arc en respectant les contraintes de différence vérifie également la consistance d'arc.*

7.9.2 Algorithme

Nous proposons un algorithme (ACRDIFF) calculant la fermeture de cette consistance pour des réseaux ne comportant qu'une seule contrainte de différence portant sur l'ensemble des variables, comme c'est le cas pour les réseaux modélisant Isossgrpartiel. Nous supposons, aussi, que l'***on connaît tous les couples autorisés de toutes les contraintes binaires.***

Algorithme 34 Phase d'initialisation d'ACRDIFF.

```
INITIALISATION(o listeAttente) : booléen
  pour chaque x_i ∈ X faire
  |  pour chaque a ∈ D_i faire
  |  |  S_ia ← ∅
  |  |  M_ia ← ∅
  pour chaque x_i ∈ X faire
  |  pour chaque a ∈ D_i faire
  |  |  si ¬ EXISTESUPPORT(x_i, a, nil) alors
  |  |  |  SUPPRIMER(a, D_i)
  |  |  |  AJOUTER((i, a), listeAttente)
  |  si D_i = ∅ alors retourner faux
  retourner vrai
```

Les principes de cet algorithme sont proches de ceux d'AC-6. En effet on peut se contenter sur chaque contrainte de la présence d'un seul support respectant la contrainte de différence. Pour simplifier l'algorithme, nous appellerons M_{ia} l'ensemble des valeurs supportant la valeur de (i, a) en respectant la contrainte de différence. Cette information est généralement facile à obtenir lorsque l'on implémente les algorithmes de fermeture par consistance d'arc. La recherche de support d'une valeur d'une variable x_i ne se fait plus par rapport à une seule contrainte mais par rapport à l'ensemble des contraintes contraignant x_i. Ainsi la phase d'initialisation est légèrement modifiée comme le montre l'algorithme 34 et on peut très facilement, si on le désire, appeler la phase de propagation à l'intérieur de la phase de propagation (cf chapitre 5).

La phase de propagation est semblable à celle d'AC-6.

La fonction EXISTESUPPORT est le coeur d'ACRDIFF. Pour une valeur (i, a), EXISTESUPPORT construit le graphe des valeurs des supports puis recherche un couplage maximum M dans ce graphe. Si M ne sature pas tous les sommets de $\Gamma_b(x_i)$ alors (i, a) n'est plus viable, sinon *les* supports de (i, a) sont mis à jour. En effet plusieurs supports peuvent avoir été modifiés car ils appartenaient au couplage maximum précédant l'appel de la fonction EXISTESUPPORT et parce qu'ils n'appartiennent pas

Algorithme 35 La fonction de recherche d'un nouveau support d'ACRDIFF.

```
EXISTESUPPORT(i x_i, a, b) : booléen
    trouvé ← faux
    GVS((x_i, a)) ← CONSTRUIREGVS(x_i, a)
    si b = nil alors
        nvM_ia ← CALCULERCOUPLAGEMAX(∅, GVS((x_i, a)))
        si |nvM_ia| = |Γ_b(x_i)| alors
            trouvé ← vrai
            pour chaque x_j ∈ Γ_b(x_i) faire
                {x_j, c} ← AUTREEXTRÉMITÉ(x_j, nvM_ia)
                AJOUTER((i, a), S_jc)
    sinon
        nvM_ia ← CALCULERCOUPLAGEMAX(M_ia − {x_j, b}, GVS((x_i, a)))
        si |nvM_ia| = |Γ_b(x_i)| alors
            trouvé ← vrai
            pour chaque x_j ∈ Γ_b(x_i) faire
                {x_j, b} ← AUTREEXTRÉMITÉ(x_j, M_ia)
                {x_j, c} ← AUTREEXTRÉMITÉ(x_j, nvM_ia)
                si b ≠ c alors
                    SUPPRIMER((i, a), S_jb)
                    AJOUTER((i, a), S_jc)
            M_ia ← nvM_ia
    retourner trouvé
```

au nouveau couplage maximum calculé par la fonction CALCULERCOUPLAGEMAX.

Une implémentation possible de la fonction EXISTESUPPORT est donnée par l'algorithme 35. Trois fonctions sont appelées :

- CONSTRUIREGVS(x_i, a) qui construit le graphe $GVS((x_i, a))$;
- CALCULERCOUPLAGEMAX$(GVS((x_i, a)), M)$ qui calcule un couplage maximum dans $GVS((x_i, a))$ à partir du couplage M;
- AUTREEXTRÉMITÉ(x_j, M) qui retourne la valeur de l'arête du couplage M ayant x_j comme autre extrémité, si M est vide elle renvoie *nil*.

7.9.3 Complexité

Nous noterons e le nombre de contraintes binaires du réseau et nous supposerons que $d = |\cup_{x_i \in X} D_i|$ est proche de $\max_{x_i \in X}(|D_i|)$.

La graphe des valeurs des supports de (x_i, a) a un nombre de sommets en $O(|\Gamma_b(x_i)| + d)$ et un nombre d'arêtes en $O(d|\Gamma_b(x_i)|))$.

Étudions tout d'abord la complexité en espace :

La taille des listes de supports ne change pas par rapport à AC-6, elle est en $O(ed)$.

Si le graphe des valeurs des supports est recalculé à appel de la fonction EXISTESUPPORT alors la complexité en espace sera en $O(ed + |\Gamma_b(x_i)| + d + d|\Gamma_b(x_i)|)$, autrement dit elle reste en $O(ed)$. Par contre si l'on mémorise pour chaque valeur le dernier graphe des valeurs des supports calculé alors nous aurons besoin, pour mémoriser ces graphes, d'un espace mémoire de l'ordre de $O(\sum_{x_i \in X} \sum_{a \in D_i}(|\Gamma_b(x_i)| + d + d|\Gamma_b(x_i)|))$ soit $O(ed^2)$.

La complexité en temps d'ACRDIFF dépend essentiellement de celle de la fonction EXISTESUPPORT et la complexité de cette dernière dépend principalement de celle de la fonction CALCULERCOUPLAGEMAX(GVS, M). Si M est vide alors le coût de cette fonction est $O(\sqrt{|\Gamma_b(x_i)|}d|\Gamma_b(x_i)|)$, et si $M = |\Gamma_b(x_i)| - 1$ il est en $O(d|\Gamma_b(x_i)|)$.

La fonction EXISTESUPPORT, quand elle est appelée par la phase d'initialisation, appelle la fonction CALCULERCOUPLAGEMAX à partir d'un couplage vide. D'après ce qui précède son coût sera $O(\sqrt{|\Gamma_b(x_i)|}d|\Gamma_b(x_i)|)$ pour chaque appel. Or, dans le pire

des cas, toutes les valeurs de toutes les variables sont initialisées. Aussi la complexité de la phase d'initialisation est $O(\sum_{x_i \in X} \sum_{a \in D_i} \sqrt{|\Gamma_b(x_i)|} d |\Gamma_b(x_i)|)$ qui est majoré par $O(\sqrt{n} d^2 \sum_{x_i \in X} |\Gamma_b(x_i)|) = O(\sqrt{n} e d^2)$.

Lorsque la fonction EXISTESUPPORT est appelée par la phase de propagation, elle appelle la fonction CALCULERCOUPLAGEMAX avec un couplage de taille $|\Gamma_b(x_i)| - 1$. Cette dernière s'exécute, alors, en $O(d|\Gamma_b(x_i)|)$. Comme n'importe quelle valeur (i, a) peut supporter sur une contrainte binaire toutes les valeurs de D_j, la complexité de la phase de propagation est en $O(\sum_{x_i \in X} \sum_{a \in D_i} \sum_{x_j \in \Gamma_b(x_i)} \sum_{c \in D_j} d|\Gamma_b(x_i)|)$ qui est majorée par $O(\sum_{x_i \in X} \sum_{a \in D_i} \sum_{x_j \in \Gamma_b(x_i)} \sum_{c \in D_j} nd) = O(ned^3)$.

La complexité en temps de ACRDIFF est donc $O(ned^3)$.

7.9.4 Perspectives

Lorsque ACRDIFF est employé comme prétraitement, il est possible, après chaque calcul de couplage maximum, de supprimer, sans coût supplémentaire, toutes les arêtes qui n'appartiennent à aucun couplage couvrant le sous-ensemble de variables considéré. La suppression d'une telle arête se traduit par une modification de la contrainte. Par exemple, si l'arête $\{x_j, b\}$ est supprimée après avoir étudié la viabilité de (i, a) alors $C_{ij}(a, b)$ devient faux. Cette modification nécessite de réétudier la viabilité de certaines valeurs. Soit parce que (i, a) est le support courant de $\{x_j, b\}$ sur C_{ji} et la viabilité de (j, b) est remise en cause. Cette valeur peut donc être directement supprimée. Soit parce que la suppression de l'arête $\{x_i, a\}$ dans le graphe $GVS((x_j, b))$ peut entraîner la suppression d'autres arêtes de ce graphe qui peuvent entraîner l'étude de la viabilité d'autres valeurs. On peut alors assister à la suppression de valeurs n'appartenant pas à D_j. L'implémentation de ce mécanisme de propagation est plutôt délicate. Cette idée est, à l'heure actuelle, une perspective possible d'amélioration de notre filtrage.

Nous pouvons aussi donner les principes d'un algorithme calculant la fermeture par consistance d'arc qui respecte les contraintes de différence, lorsque le réseau comporte plusieurs contraintes de différence. Il suffit, par exemple, d'utiliser autant de listes de supports pour une valeur quelconque (x_i, a) que le nombre de fois où x_i est impliqué dans des contraintes de différence.

type d'instance	IAC	IACrdiff	$IAC_{\neq}$	autres	toutes
gain	50%	38%	31%	-73%	20,5%

FIG. 7.9 - *Pourcentage de gain réalisé par $AC_{\neq}$ par rapport à AC-6ps pour résoudre* Iso-ssgrpartiel.

Par ailleurs, mentionnons qu'il doit être possible de définir un algorithme ACRDIFF n'ayant besoin ni de connaître ni de calculer systématiquement tous les supports de toutes les valeurs. Un tel algorithme n'est certainement pas très facile à mettre en oeuvre car il nécessite un algorithme incrémental de calcul d'un couplage maximum.

Enfin, une question reste en suspend :
«ACRDIFF permet-il de résoudre le problème de l'isomorphisme de sous-arbre en temps polynômial?»

7.10 Expérimentations

Nous avons testé, pour 10 000 instances de Iso-ssgrpartiel, un algorithme combinant ACRDIFF et celui réalisant la fermeture par diff-arc-consistance des réseaux. Nous appellerons $AC_{\neq}$ ce nouvel algorithme.

Nous noterons IAC les instances vérifiant l'inconsistance d'arc, IACrdiff celles qui vérifiant l'inconsistance d'arc en respectant les contraintes de différence, et $IAC_{\neq}$ celles qui sont IACrdiff et qui en plus vérifient l'inconsistance d'arc généralisée pour les contraintes de différence.

16,7 % des instances sont IAC, 75 % sont IACrdiff et 90 % sont $IAC_{\neq}$. Le filtrage est donc très intéressant pour résoudre notre problème.

Étudions maintenant le temps mis par les différents algorithmes pour filtrer les 10 000 instances. Nous avons résumé en figure 7.9 les pourcentages de gain obtenus par $AC_{\neq}$ par rapport à AC-6ps.

On remarque que $AC_{\neq}$ gagne presque toujours du temps pour filtrer les problèmes. Cela est dû au fait qu'un grand nombre d'entre eux sont inconsistants et que cette inconsistance est détectée plus tôt dans l'algorithme par $AC_{\neq}$. D'ailleurs dès que le filtrage ne permet plus de conclure à l'absence de solution, AC-6ps devient beaucoup plus rapide.

type d'instance	AC et $IAC_{\neq}$	$AC_{\neq}$	toutes
gain	94%	18%	77%

FIG. 7.10 - *Pourcentages de gain obtenus par $MAC_{\neq}$ par rapport à MAC-6ps.*

Intéressons nous maintenant à la recherche d'une solution. Nous avons comparé les performances de MAC-6ps et de $MAC_{\neq}$ (maintien de $AC_{\neq}$ pendant la recherche). La figure 7.10 donne les pourcentages de gain obtenus.

La gain réalisé par $MAC_{\neq}$ pour les instances que le prétraitement a détectées comme étant inconsistantes est très important. Cela montre clairement que la puissance du filtrage réalisé par $AC_{\neq}$. Pour les autres instances le gain est plus faible car l'arbre de recherche de $MAC_{\neq}$ est réduit par rapport à celui de MAC-6ps mais le temps passé pour étudier chaque noeud est plus important. Enfin le temps global gagné (77%) montre l'intérêt de notre approche pour résoudre lso-ssgrpartiel.

Il est important de bien noter que tous les problèmes considérés sont assez particuliers, puisque que les graphes considérés sont les squelettes de molécules organiques. Les degrés des sommets de ces graphes sont presque toujours inférieur à 5. Ce qui explique, peut-être, de telles différences entre les algorithmes MAC-6ps et $MAC_{\neq}$.

7.11 Conclusion et Perspectives

Dans ce chapitre, nous nous sommes intéressés aux contraintes de différence. Nous avons présenté une implémentation efficace de la consistance d'arc généralisée pour ces contraintes. Le coût du nouvel algorithme calculant la fermeture de cette consistance pour une contrainte est semblable à la vérification de l'existence d'au moins un tuple satisfaisant la contrainte. Nous avons montré l'intérêt de disposer d'un tel traitement pour modéliser certains problèmes. Par ailleurs, nous avons proposé une nouvelle consistance pour les contraintes binaires: la consistance d'arc en respectant les contraintes de différence, et un algorithme la mettant en oeuvre. La puissance du nouveau filtrage obtenu en combinant cet algorithme avec celui achevant la consistance d'arc généralisée pour les contraintes de différence a été mis en évidence pour lso-ssgrpartiel où l'on obtient 77% de gain en temps par rapport aux méthodes développées dans les chapitres précédents.

Ce chapitre offre plusieurs perspectives.

À la vue des résultats obtenus, nous pensons qu'il est temps de s'intéresser à la consistance d'arc généralisée pour d'autres types de contraintes n-aires.

Il serait certainement aussi intéressant d'essayer de trouver de nouveaux ordres d'instanciation basés sur la structure du graphe des valeurs. On pourrait par exemple choisir l'arête entraînant le plus de suppression.

On pourrait également se demander à quelles conditions un réseau de contraintes qui comporte des contraintes de différence peut se résoudre en un temps polynômial.

Chapitre 8

Les graphes étiquetés

Les graphes étiquetés se rencontrent très fréquemment en pratique. Par exemple, les graphes moléculaires, qui sont la base du langage du chimiste, peuvent être représentés par des graphes étiquetés, où l'étiquette des sommets contient certaines informations relatives aux propriétés chimiques des atomes, et où l'étiquette des arêtes correspond aux caractères chimiques des liaisons.

Le propos de ce chapitre n'est pas seulement de montrer comment l'on peut résoudre Iso-ssgrpartiel pour des graphes dont les sommets et les arêtes sont étiquetés. Il explique aussi comment certains problèmes peuvent se modéliser facilement grâce à l'étiquetage des sommets et des arêtes. Nous distinguerons deux parties, l'une consacrée à la modélisation à l'aide d'étiquettes de certains problèmes et l'autre à la résolution de Iso-ssgrpartiel pour des graphes étiquetés. Nous verrons alors que les étiquettes sur les sommets ne posent pas de problèmes particuliers car elles sont prises en compte en introduisant des contraintes unaires. En ce qui concerne les étiquettes liées aux arêtes, il existe deux méthodes pour construire un réseau de contraintes prenant en compte ces étiquettes. La première les intègre dans la définition des contraintes, tandis que la seconde introduit de nouvelles variables et des contraintes unaires portant sur ces variables.

8.1 Graphes étiquetés

Un graphe étiqueté est un graphe dont les sommets et éventuellement les arêtes sont munis d'étiquettes. Il peut être modélisé par un quadruplet (X, E, etq_S, etq_A) où X est l'ensemble de sommets du graphe, E son ensemble d'arêtes et :

- etq_S est une fonction de l'ensemble des sommets dans un sous-ensemble de l'alphabet $A_S = \{\epsilon_1, ..., \epsilon_p\}$.
- etq_A est une fonction de l'ensemble des arêtes dans un sous-ensemble de l'alphabet $A_A = \{\eta_1, ..., \eta_q\}$.

Nous allons avoir besoin de comparer deux étiquettes. Pour ce faire nous définissons deux fonctions booléennes permettant de savoir si deux étiquettes sont compatibles.

- $\sigma(x, y)$ est vrai si $etq_S(x)$ est compatible avec $etq_S(y)$, et faux sinon.
- $\alpha(e, f)$ est vrai si $etq_A(e)$ est compatible avec $etq_A(f)$, et faux sinon.

La notion de compatibilité est beaucoup plus forte que la notion d'égalité. Ces deux fonctions vont nous permettre de modéliser certains problèmes beaucoup plus facilement mais aussi d'obtenir des algorithmes plus performants.

8.2 Modélisation de certains problèmes à l'aide de graphes étiquetés

Certains problèmes consistent à rechercher des morphismes injectifs entre deux graphes vérifiant certaines propriétés. Il est possible, pour trouver une solution à ce type de problème, d'énumérer tous les morphismes injectifs entre les deux graphes puis de ne conserver que ceux qui répondent à certains critères. Nous allons illustrer cela sur un cas d'école. En chimie organique, les molécules sont classées en fonction de certains groupement d'atomes qu'elles possèdent. Ainsi la propriété «acide-carboxylique» d'une molécule dépend de la présence du motif donné par la figure 8.1 dans le graphe moléculaire.

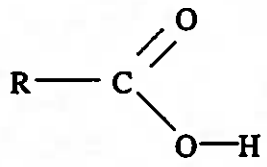


FIG. 8.1 - *Le groupement caractérisant un acide carboxylique.*

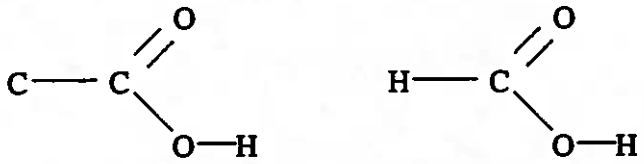


FIG. 8.2 - *Deux groupements représentant un acide carboxylique.*

Dans cette figure C représente un atome de carbone, O un atome d'oxygène, H un atome d'hydrogène et R est un symbole générique pouvant représenter soit un carbone, soit un hydrogène. En fait le motif donné sans la figure 8.1 correspond aux deux graphes de la figure 8.2.

Une molécule[1] peut être représentée par un graphe étiqueté dont les alphabets sont définis de la façon suivante :

- A_S est constitué des types atomiques c'est-à-dire des symboles des atomes tels qu'ils apparaissent dans la classification de Mendeleïev et certains symboles génériques comme R.

- A_A est l'ensemble des multiplicités possibles des liaisons (liaison simple, double, triple, aromatique).

Pour savoir si une molécule présente un ou plusieurs groupements acides carboxiliques, on peut utiliser au moins trois méthodes :

1. On recherche les sous-graphes du graphe moléculaire isomorphe au groupement sans tenir compte des étiquettes (des types atomiques pour les sommets et la multiplicité des arêtes pour les liaisons). Puis on sélectionne ceux qui possèdent des étiquettes compatibles.

1. Nous verrons dans la seconde partie de cette thèse que cette modélisation peut être plus complexe.

2. On recherche en tenant compte des étiquettes, si le graphe moléculaire contient un sous-graphe isomorphe à l'un des deux graphes donné en figure 8.2. La fonction de compatibilité entre étiquette correspond à l'égalité.

3. On recherche en tenant des étiquettes, si le graphe moléculaire contient un sous-graphe isomorphe au groupement moléculaire de la figure 8.1. Cette fois, la fonction de compatibilité entre étiquette prend en considération certaines connaissances du domaine, par exemple R est compatible avec C ou H.

La dernière méthode est la plus puissante, car elle permet de modéliser assez facilement des problèmes complexes. De plus, elle donne, en pratique, de bien meilleurs résultats.

Dans la section suivante, nous allons montrer comment une telle technique peut être mise en oeuvre.

8.3 Résolution de Iso-ssgrpartiel pour des graphes étiquetés

Nous considérerons dans cette section que l'on dispose des fonctions α et σ qui permettent de comparer deux étiquettes.

Dans un premier temps nous considérerons que seuls les sommets des graphes sont étiquetés. Puis nous verrons comment le problème peut être résolu pour des graphes dont leurs arêtes sont aussi étiquetées.

8.3.1 Graphes dont les sommets sont étiquetés

Comme nous l'avons vu dans les chapitres précédents, Iso-ssgrpartiel peut être modélisé par un réseau de contraintes. Il en est de même si les sommets sont étiquetés. Les étiquettes induisent simplement des contraintes unaires, les contraintes binaires et l'unique contrainte n-aire restent inchangées. Les contraintes unaires définissent l'appartenance d'une valeur au domaine d'une variable.

Considérons $G = (X_G, E_G, etq_S)$ et $H = (X_H, E_H, etq_S)$ deux graphes, σ une fonction de compatibilité entre étiquettes des sommets et supposons que l'on veuille savoir

si H est isomorphe à un sous-graphe partiel de G en tenant compte des étiquettes. Pour résoudre ce problème nous utiliserons le réseau de contraintes suivant :

$\mathcal{R} = (X_H, \mathcal{D}, \mathcal{C})$ où :

- X_H est l'ensemble des sommets de H.
- Le domaine de chaque variable h est constitué par l'ensemble des sommets de G dont le degré est supérieur ou égal à celui de h.
- A chaque variable h correspond une contrainte unaire définie par la fonction σ : $C_h(g) = \sigma(h, g)$.
- A chaque arête $\{i, j\}$ de H correspond une contrainte binaire C_{ij} telle que $C_{ij}(a, b)$ est vrai si et seulement si $\{a, b\} \in E_G$.
- il existe une contrainte de différence impliquant tous les sommets de X_H : TOUSDIFFÉRENTS(X_H).

Pour résoudre le CSP associé à ce réseau il suffit de résoudre dans un premier temps les contraintes unaires puis d'appliquer les techniques qui ont été développées dans les chapitres précédents.

8.3.2 Graphes dont les sommets et les arêtes sont étiquetés

Lorsque les arêtes sont étiquetées, il est possible d'envisager deux modélisations. La première intègre ces étiquettes dans la définition des contraintes binaires. La seconde consiste à construire un nouveau graphe en plaçant un sommet étiqueté sur chaque arête.

Nous considérerons dans la suite deux graphes $G = (X_G, E_G, etq_S, etq_A)$ et $H = (X_H, E_H, etq_S, etq_A)$ dont les sommets et les arêtes sont étiquetés, σ une fonction de compatibilité entre étiquettes des sommets et α une fonction de compatibilité entre étiquettes des arêtes.

Prise en compte des étiquettes des arêtes par les contraintes

Pour résoudre Iso-ssgrpartiel, on peut construire le réseau suivant :

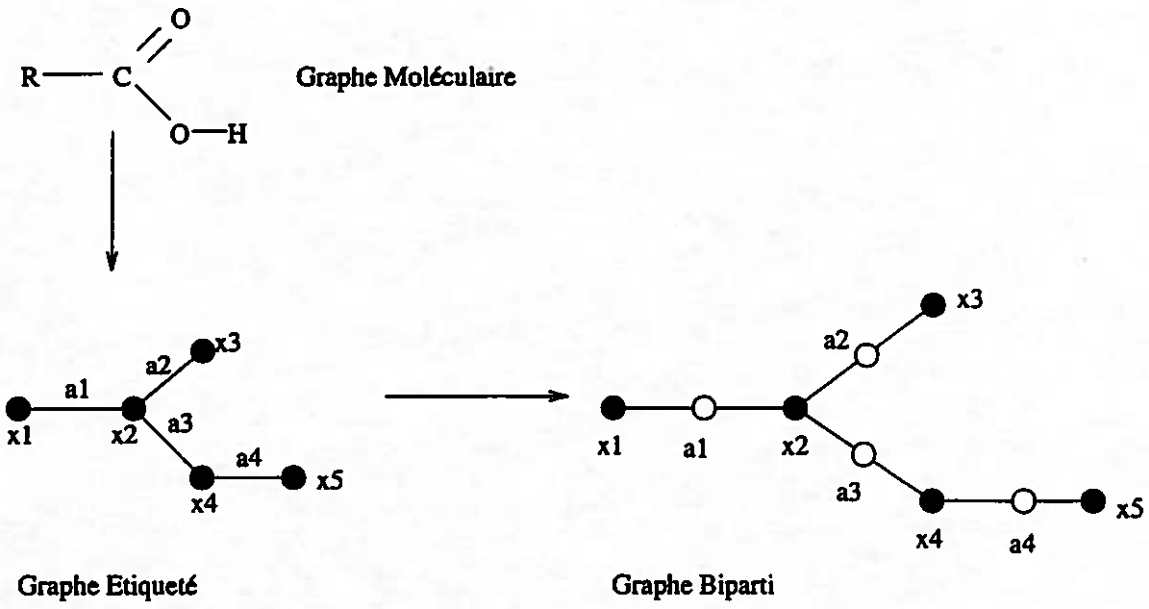
$\mathcal{R} = (X_H, \mathcal{D}, \mathcal{C})$ où :

- X_H est l'ensemble des sommets de H.
- Le domaine de chaque variable h est constitué par l'ensemble des sommets de G dont le degré est supérieur ou égal à celui de h.
- A chaque variable h correspond une contrainte unaire définie par la fonction σ : $C_h(g) = \sigma(h, g)$.
- A chaque arête $\{i, j\}$ de H correspond une contrainte binaire C_{ij} telle que $C_{ij}(a, b)$ est vrai si et seulement si $\{a, b\} \in E_G$ et $\alpha(\{i, j\}, \{a, b\})$ est vrai.
- il existe une contrainte de différence impliquant tous les sommets de X_H : TOUSDIFFÉRENTS(X_H).

Avec cette représentation, on remarquera que l'on obtient autant de types de contraintes différents qu'il y a d'étiquettes différentes dans H. Comme toutes les contraintes ne sont plus identiques, la complexité en espace ne sera plus celle des graphes ayant des sommets étiquetés. En outre cette méthode risque d'entraîner davantage de calculs.

Prise en compte des étiquettes des arêtes à l'aide de graphes bipartis

Un graphe dont les arêtes sont étiquetées peut toujours être vu comme un graphe biparti ayant uniquement des sommets étiquetés. Nous allons montrer comment un tel graphe peut être construit pour le graphe H. Appelons B le graphe que l'on veut obtenir. L'ensemble des sommets de B est composé de l'ensemble X_H des sommets de H et d'un ensemble A de sommets comportant autant d'éléments que H possède d'arêtes. A chaque sommet a de A correspond une arête e de E_H, a est muni de l'étiquette de e. Un sommet x de X_H dans B est relié à un sommet a de A ssi x est relié dans H par l'arête de E_H correspondant au sommet a. La figure 8.3 donne un exemple de tel graphe.



etqS(x1)=R	etqA(a1)=1
etqS(x2)=C	etqA(a2)=2
etqS(x3)=O	etqA(a3)=I
etqS(x4)=O	etqA(a4)=1
etqS(x5)=H	

etqS(x1)=R	etqS(aI)=1
etqS(x2)=C	etqS(a2)=2
etqS(x3)=O	etqS(a3)=1
etqS(x4)=O	etqS(a4)=1
etqS(x5)=H	

FIG. 8.3 - *Un graphe ayant des sommets et des arêtes étiquetés et un graphe biparti équivalent sans arêtes étiquetées.*

En construisant un graphe biparti pour chacun des graphes H et G, on peut résoudre lso-ssgrpartiel pour deux graphes dont seuls les sommets sont étiquetés. En procédant ainsi, on retrouve une excellente complexité en espace.

Chapitre 9

Conclusion

Dans cette partie, nous nous sommes intéressés au problème de l'isomorphisme de sous-graphe. Nous avons d'abord positionné le problème et défini la frontière entre les classes d'instances polynomiales et celles NP-Complètes. Nous avons ensuite présenté quelques méthodes pour résoudre les classes d'instances polynomiales. Puis nous avons montré l'intérêt des réseaux de contraintes pour modéliser et résoudre les autres classes.

Après un état de l'art concernant les problèmes de satisfaction de contraintes, nous avons présenté deux nouveaux algorithmes calculant la fermeture par consistance d'arc. L'un, appelé AC-7, utilise la bidirectionnalité des contraintes afin de réaliser moins de tests de consistance que l'algorithme AC-6, tout en préservant les complexités en temps et en espace de ce dernier. L'autre, AC-Inférence, tire profit de certaines propriétés des contraintes pour déduire des tests effectués le résultat d'autres tests de consistance. Nous avons aussi proposé deux heuristiques permettant très souvent d'améliorer les performances d'algorithmes achevant la consistance d'arc. La première remet en cause le schéma classique des algorithmes AC-4 et AC-6. Au lieu de diviser ces algorithmes en deux phases distinctes, une d'initialisation et une de propagation, nous préconisons de regrouper ces phases en une seule, autrement dit d'étudier immédiatement les conséquences de la disparition d'une valeur lors de la phase d'initialisation. La seconde heuristique est un traitement particulier pour les variables dont le domaine se réduit à un singleton.

Nous avons ensuite donné des algorithmes maintenant la consistance d'arc pendant

une procédure de recherche. Pour ce faire nous avons défini les caractéristiques de tels algorithmes et expliqué les modifications qu'il faut apporter aux algorithmes calculant la fermeture par consistance d'arc pour qu'ils puissent être utilisés pendant une procédure de recherche. Cela nous a amené à redéfinir les différents types d'ordre d'instanciation en proposant un nouveau ordre: l'ordre fortement dynamique. Pour la première fois nous avons testé les algorithmes MAC-6, MAC-7 et MAC-Inférence.

Enfin, nous nous sommes intéressés aux contraintes de différence. Nous avons défini deux nouvelles consistances et deux algorithmes calculant leur fermeture. La première correspond à la consistance d'arc généralisée pour les contraintes de différence. La seconde a pour but de calculer la fermeture par consistance d'arc pour les contraintes binaires en respectant les contraintes de différence.

Tous ces algorithmes ont été testé avec succès pour lso-ssgrpartiel, les problèmes d'assignement de fréquence et des réseaux aléatoires.

Les perspectives de cette partie sont nombreuses, nous donnons les 3 principales:

Tout d'abord nous pensons qu'il est encore possible d'améliorer les algorithmes calculant la fermeture par consistance d'arc en recherchant en premier lieu si le réseau contient un sous-domaine qui vérifie cette consistance.

Ensuite, il semble nécessaire, maintenant, d'étudier la combinaison d'un algorithme MAC avec un algorithme de recherche intelligent, comme le conflict-directed backjumping.

Enfin, il faudrait s'intéresser à la consistance d'arc généralisée pour d'autres contraintes n-aires et améliorer l'algorithme ACRDIFF.

Annexe A

Quelques définitions peu usuelles de la théorie des graphes

Les définitions de cette annexe sont empruntés aux livres de [Berge, 1970], [Harary, 1969], [Gondran and Minoux, 1985].

A.1 Les graphes planaires

On dit qu'un graphe est ***planaire*** s'il est possible de le représenter sur un plan de sorte que les sommets soient des points distincts, les arêtes des courbes simples, et que deux arêtes ne se rencontrent pas en dehors de leurs extrémités[1] La représentation d'un graphe sur un plan conformément aux conditions imposées s'appelle ***graphe planaire topologique***. Considérons un graphe planaire topologique G Une ***face*** de G est une région de plan limitée par des arêtes et telle que deux points arbitraires dans cette région peuvent toujours être reliés par un trait continu ne rencontrant ni sommets ni arêtes. La ***frontière*** d'une face est l'ensemble des arêtes qui touchent la face. Tout graphe planaire topologique contient une ***face extérieure*** illimitée.

Afin de présenter une définition plus formelle d'un graphe planaire, nous introduisons la notion d'homéomorphisme :

Définition 47 *Un graphe G est **homéomorphe** à un graphe H, si et seulement G*

1. en anglais une telle représentation d'un graphe sur une surface est appelée *embedding*

peut être converti en un graphe isomorphe à H en répétant l'opération qui consiste à remplacer un sommet de degré deux par une arête qui relie ses deux voisins.

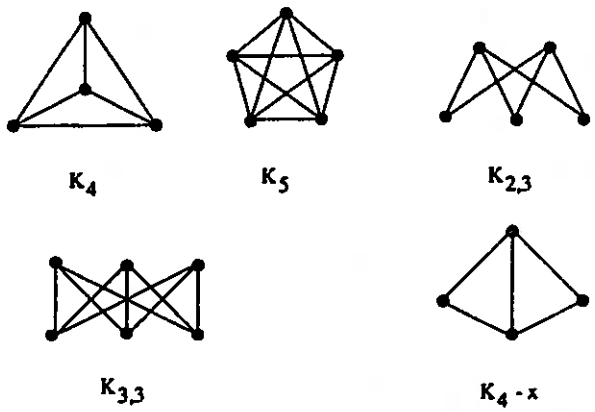


FIG. A.1 - *Les graphes $K_4, K_5, K_{2,3}, K_{3,3}$ et $K_4 - x$.*

On peut alors définir de manière plus formelle un graphe planaire :

Définition 48 (Kuratowski 1930) *Un graphe est planaire si et seulement si il ne contient pas un sous-graphe partiel homéomorphe à $K_{3,3}$ ou à K_5.*

Définition 49 *Un graphe est* ***outerplanar***[2] *si et seulement si il admet une représentation topologique planaire telle que tous les sommets du graphe appartiennent à la face extérieure.*

Tous les graphes outerplanar sauf $K_4 - x$, ne contiennent pas de sous-graphe homéomorphe à K_4 ou à $K_{2,3}$.

Définition 50 *Un graphe est* ***serie-parallèle*** *si et seulement si quelque soit un cycle simple C du graphe, il existe une représentation topologique planaire de G, tel que C soit la frontière de la face extérieure.*

On peut aussi définir un graphe série-parallèle comme étant un graphe qui ne possède pas de sous-graphe partiel homéomorphe à K_4.

2. Comme nous n'avons pas trouvé de traduction satisfaisante, nous conserverons le terme anglais

Annexe B

Le problème du zèbre

Le problème du zèbre présente de nombreuses similarités avec des problèmes réels. Il a été proposé par Dechter [Laurière, 1976]. Le problème est donné en figure B.1.

Ce problème peut être représenté par un réseau de contraintes impliquant 25 variables, une pour chacune des 5 couleurs, des 5 boissons, des 5 marques de cigarettes

1. il y a 5 maisons, chacune a une couleur différente, un habitant de nationalité différente, un animal différent; chaque habitant boit une boisson différente et fume des cigarettes différentes.
2. l'anglais habite la maison rouge.
3. l'espagnol possède un chien.
4. le café est bu dans la maison verte.
5. l'ukrainien boit du thé.
6. la maison verte est immédiatement à droite de la maison ivoire.
7. le fumeur de Old-Gold possède des escargots.
8. les Kools sont fumées dans la maison jaune.
9. le lait est bu dans la maison du milieu.
10. le norvégien habite dans la première maison à gauche.
11. le fumeur de Chesterfield habite à côté de la maison où se trouve le renard.
12. les Kools sont fumées dans la maison à côté de la maison où se trouve le cheval.
13. le fumeur de Lucky-Strike boit du jus d'orange.
14. le japonais fume des Parliament.
15. le norvégien habite à côté de la maison bleue.

La question est : Qui boit de l'eau et où habite le zèbre?

FIG. B.1 - *Le problème du zèbre.*

C_1	rouge	N_1	anglais	A_1	chien	B_1	café	T_1	Old Gold
C_2	vert	N_2	espagnol	A_2	escargot	B_2	thé	T_2	Chesterfield
C_3	ivoire	N_3	ukrainien	A_3	renard	B_3	lait	T_3	Kool
C_4	jaune	N_4	norvégien	A_4	cheval	B_4	jus d'orange	T_4	Lucky Strike
C_5	bleu	N_5	japonais	A_5	zèbre	B_5	eau	T_5	Parliament

FIG. B.2 - *Définition des variables pour le problème du zèbre.*

et des 5 animaux. Chaque variable a comme domaine $\{1, 2, 3, 4, 5\}$. Chaque numéro correspond à la position d'une maison (affecter la valeur 2 à la variable A_4 signifie que le possesseur du cheval habite dans la seconde maison). Les correspondances entre les mots et les variables sont données par la figure B.2.

Les affirmations 2 à 15 sont traduites en contraintes binaires et unaires. La figure B.3 représente en extension les contraintes correspondant aux affirmations 2 à 15. Les contraintes de différence entre variables du même groupe sont omises pour des raisons de clarté. À la création, il y a 68 couples de valeurs, 125 valeurs au total et 2 variables sont instanciées soit 8%.

Par ailleurs, il y a trois manières de représenter la première affirmation qui signifie que les variables du même groupe doivent prendre des valeurs deux à deux différentes.

1. Une contrainte binaire est construite entre toutes les paires de variables du même groupe assurant que deux variables ne seront pas instanciées avec la même valeur. Dans ce cas nous obtenons un réseau de contraintes binaires.

2. 5 contraintes 5-aires sont construites (une pour chaque groupe). Le réseau n'est pas binaire.

3. Les 5 contraintes de différence sont représentées par leur graphe des valeurs (cf Chapitre 7).

La première représentation est celle qui est généralement utilisée pour résoudre le problème [Dechter, 1990; Bessière and Cordier, 1993].

À partir de ces trois représentations, nous pouvons étudié les différents résultats obtenus par la consistance d'arc.

Pour la première représentation, les résultats du filtrage par consistance d'arc sont donnés en figure B.4.

2 (=)	
N_1	C_1
1	1
2	2
3	3
4	4
5	5

3 (=)	
N_2	A_1
1	1
2	2
3	3
4	4
5	5

4(=)	
B_1	C_2
1	1
2	2
3	3
4	4
5	5

5(=)	
N_3	B_2
1	1
2	2
3	3
4	4
5	5

6 (-1)	
C_2	C_3
2	1
3	2
4	3
5	4

7(=)	
T_1	A_2
1	1
2	2
3	3
4	4
5	5

8(=)	
T_3	C_4
1	1
2	2
3	3
4	4
5	5

9
B_3
3

10
N_4
1

11(+1;-1)	
T_2	A_3
1	2
2	1
2	3
3	2
3	4
4	3
4	5
5	4

12(+1;-1)	
A_4	T_3
1	2
2	1
2	3
3	2
3	4
4	3
4	5
5	4

13 (=)	
T_4	B_4
1	1
2	2
3	3
4	4
5	5

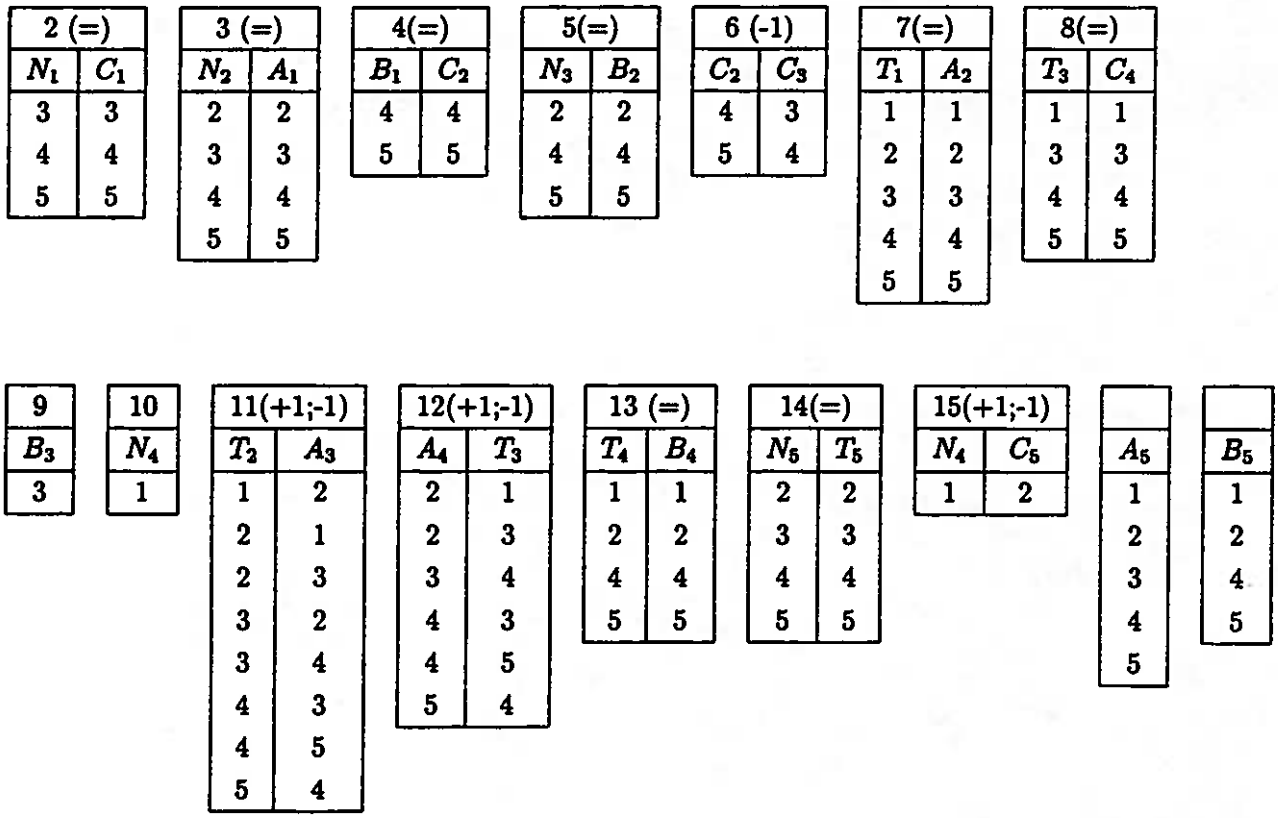
14(=)	
N_5	T_5
1	1
2	2
3	3
4	4
5	5

15(+1;-1)	
N_4	C_5
1	2
2	1
2	3
3	2
3	4
4	3
4	5
5	4

A_5	
1	
2	
3	
4	
5	

B_5	
1	
2	
3	
4	
5	

FIG. B.3 - *Représentation en extension des contraintes correspondant aux affirmations 2 à 15.*



2 (=)	
N_1	C_1
3	3
4	4
5	5

3 (=)	
N_2	A_1
2	2
3	3
4	4
5	5

4(=)	
B_1	C_2
4	4
5	5

5(=)	
N_3	B_2
2	2
4	4
5	5

6 (-1)	
C_2	C_3
4	3
5	4

7(=)	
T_1	A_2
1	1
2	2
3	3
4	4
5	5

8(=)	
T_3	C_4
1	1
3	3
4	4
5	5

9
B_3
3

10
N_4
1

11(+1;-1)	
T_2	A_3
1	2
2	1
2	3
3	2
3	4
4	3
4	5
5	4

12(+1;-1)	
A_4	T_3
2	1
2	3
3	4
4	3
4	5
5	4

13 (=)	
T_4	B_4
1	1
2	2
4	4
5	5

14(=)	
N_5	T_5
2	2
3	3
4	4
5	5

15(+1;-1)	
N_4	C_5
1	2

A_5
1
2
3
4
5

B_5
1
2
4
5

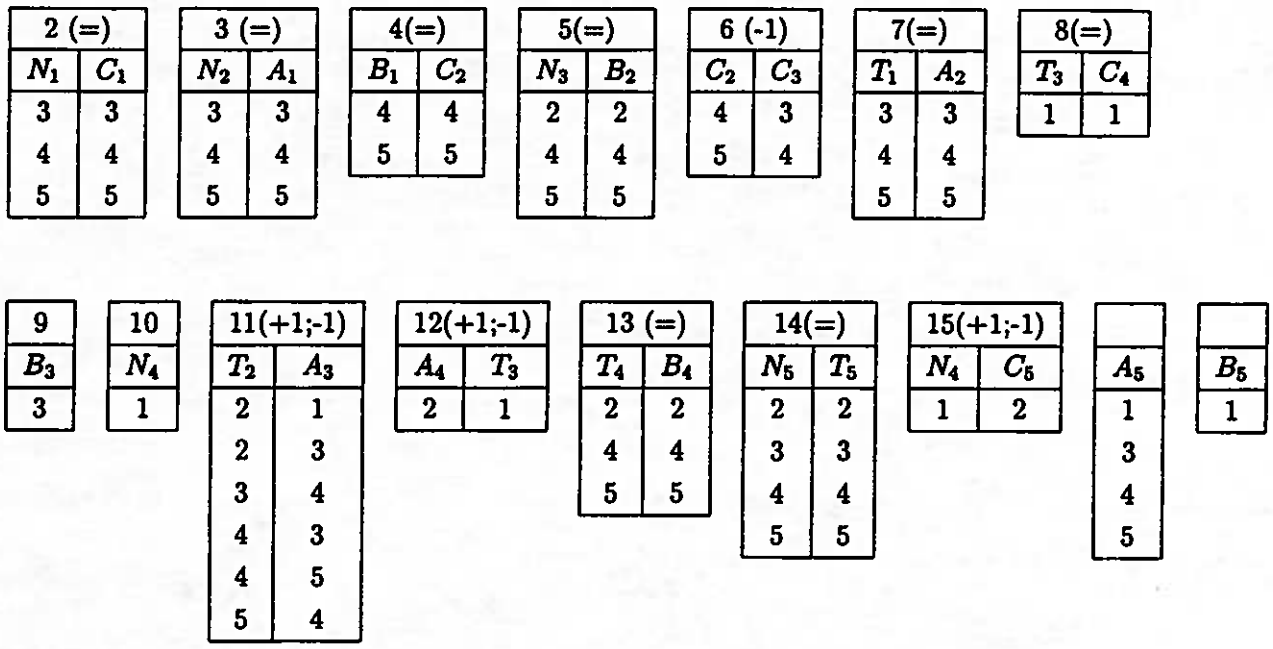
FIG. B.4 - *Première représentation après application de la consistance d'arc.*

Les contraintes qui ne sont pas des contraintes de différence sont représentées en figure B.3.

La consistance d'arc permet l'élimination de 22 couples de valeurs, soit une réduction de 32,4 %. Le nombre total de valeurs est égal à 86, soit 31,2% valeurs en moins. Et une variable en plus est instanciée.

Pour la seconde représentation, le filtrage employé est la consistance d'arc généralisée. La figure B.5 montre les nouveaux résultats obtenus. Ce filtrage supprime nettement plus de valeurs que le précédent. En effet, il reste 32 couples, soit un facteur d'élimination de 53%, et 62 valeurs total soit 51% de suppressions et on a instancié 8 variables, presqu'un tiers du problème est résolu.

Pour la troisième représentation le filtrage employé est la consistance d'arc pour les contraintes binaires et la fermeture par consistance d'arc généralisée pour les contraintes de différence. Les résultats obtenus sont les mêmes qu'avec la précédente méthode. Nous pouvons détailler un peu le fonctionnement du nouveau filtrage.



2 (=)	
N_1	C_1
3	3
4	4
5	5

3 (=)	
N_2	A_1
3	3
4	4
5	5

4(=)	
B_1	C_2
4	4
5	5

5(=)	
N_3	B_2
2	2
4	4
5	5

6 (-1)	
C_2	C_3
4	3
5	4

7(=)	
T_1	A_2
3	3
4	4
5	5

8(=)	
T_3	C_4
1	1

9
B_3
3

10
N_4
1

11(+1;-1)	
T_2	A_3
2	1
2	3
3	4
4	3
4	5
5	4

12(+1;-1)	
A_4	T_3
2	1

13 (=)	
T_4	B_4
2	2
4	4
5	5

14(=)	
N_5	T_5
2	2
3	3
4	4
5	5

15(+1;-1)	
N_4	C_5
1	2

A_5	
1	
3	
4	
5	

B_5	
1	

FIG. B.5 - *Représentation 2 et 3 : résultats obtenus après filtrage par consistance d'arc.*

Le traitement particulier lié aux contraintes de différence s'applique de la manière suivante. Par la définition du problème nous savons qu'il existe 5 contraintes de différence, chacune regroupant les variables de même type. On a, par exemple la contrainte de différence définie sur l'ensemble de variables $\{C_1, C_2, C_3, C_4, C_5\}$. Dessinons le graphe biparti correspondant aux variables de cette contrainte et aux valeurs de leur domaines. On obtient le graphe donné par la figure B.6.

On remarque que s'il existe un couplage de l'ensemble des variables dans les valeurs alors nécessairement ce couplage contiendra la valeur 1 pour la variable C_4. En effet les trois variables C_1, C_2 et C_3 prennent leurs valeurs dans $\{3, 4, 5\}$ donc C_4 ne peut pas être apparié avec une de ces valeurs, car on a trois valeurs pour trois variables. On pourra ensuite appliquer le même raisonnement pour la contrainte de différence définie sur l'ensemble de variables $\{B_1, B_2, B_3, B_4, B_5\}$, ce qui permettra d'instancier la variable B_5 avec la valeur 1.

Notons a le nombre de contraintes binaires correspondant aux affirmations 2 à 15, p la taille d'un groupe de variable, c le nombre de groupe et d le nombre de valeur

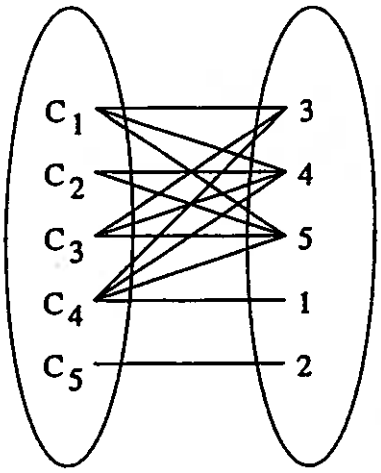


FIG. B.6 - *Graphe des valeurs de la contrainte de différence définie sur* $\{C_1, C_2, C_3, C_4, C_5\}$.

des domaines. Nous pouvons maintenant calculer la complexité des trois méthodes:

1. Pour la première représentation, le nombre de contraintes binaires ajoutées est en $O(cp^2)$. Aussi, la complexité du filtrage est en $O((a+cp^2)d^2)$.

2. Dans le second cas nous pouvons considérer que la complexité est la somme des longueurs de toutes les 5-uplets admissibles pour les 5 contraintes 5-aires. Elle est en $O(\frac{d!}{(d-p)!}p)$.

3. Pour la troisième méthode la consistance d'arc pour les contraintes binaires est en $O(ad^2)$ et le filtrage pour les contraintes de différence est en $O(cp^2d^2)$. La complexité totale est en $O(ad^2)+O(cp^2d^2)$. Ce qui est équivalent à la première.

Le second filtrage élimine plus de valeurs que le premier. Mais sa complexité est nettement plus élevée. La représentation et l'algorithme que nous avons proposé au chapitre 6 donne des résultats équivalent à la seconde méthode avec une complexité équivalente à celle de la première méthode. Nous pouvons donc conclure que ce type d'approche est excellent pour des problèmes ressemblant à celui du zèbre.

Cependant, on peut obtenir encore un meilleur résultat. Il est possible de formuler le problème du zèbre différemment pour rendre le traitement des contraintes de différence encore plus efficace. On regroupe les variables qui sont liées par une contrainte d'égalité en une seule variable dont le nom est la concaténation des deux variables.

6(-1)	
B_1C_2	C_3
4	3
5	4

11(+1;-1)	
T_2	A_3
2	1
2	3
3	4
4	3
4	5
5	4

12(+1;-1)	
A_4	T_3C_4
2	1

15(+1;-1)	
N_4	C_5
1	2

FIG. B.7 - *Résultats obtenus après regroupement des variables liées par une contrainte d'égalité et après application du filtrage pour les nouvelles contraintes de différence.*

Cette fois-ci le réseau comprend 17 variables:

$X = \{N_1C_1, N_2A_1, B_1C_2, N_3B_2, T_1A_2, T_3C_4, T_4B_4, N_5T_5, N_4, A_3, A_4, A_5, B_3, B_5, C_3, C_5, T_2\}$

On retrouve les 5 contraintes de différence du réseau précédent qui sont avec les nouvelles variables:

- $(N_1C_1, N_2A_1, N_3B_2, N_4, N_5T_5)$
- $(A_1N_2, A_2T_1, A_3, A_4, A_5)$
- $(B_1C_2, B_2N_3, B_3, B_4T_4, B_5)$
- $(C_1N_1, C_2B_1, C_3, C_4T_3, C_5)$
- $(T_1A_2, T_2, T_3C_4, T_4B_4, T_5N_5)$

On trouve en plus 8 triangles:

- (N_1C_1, B_1C_2, N_3B_2)
- (N_1C_1, T_3C_4, N_5T_5)
- (N_2A_1, N_5T_5, T_1A_2)
- (B_1C_2, T_3C_4, T_4B_4)
- (N_3B_2, T_4B_4, N_5T_5)
- (T_2, A_3, T_1A_2)
- (A_4, T_3C_4, T_1A_2)
- (N_4, C_5, N_1C_1)

Les contraintes qui ne sont pas des contraintes de différence sont au nombre de 4 , au lieu de 12. Après le traitement lié aux contraintes de différence et associé à la consistance d'arc, on obtient la figure B.7.

En procédant ainsi on n'a pas de meilleurs résultats qu'avec la méthode précédente pour la première application de l'algorithme. Si l'on emploie de manière systématique

ce filtrage, comme le font les algorithmes MAC, alors si l'on choisit la valeur 5 pour la variable B_1C_2, qui est une des variables non instanciées qui a le plus petit domaine (l'autre est C_3 et le même raisonnement que celui qui va suivre donnerait le même résultat), le filtrage va donner la solution du CSP, sans faire de retour-arrière. Et si l'on supprime la valeur 5 pour la variable B_1C_2 le filtrage va trouver, sans avoir besoin de faire d'instanciation (donc de retour-arrière), que le CSP résultant de cette suppression n'a pas de solution[1]. Avec la précédente formulation du problème il était impossible d'arriver au même résultat sans faire de retour-arrière.

Deux choix seulement après le premier filtrage par consistance d'arc peuvent engendrer un retour-arrière. Ce sont 5 pour T_1A_2 et 1 pour A_5. Alors qu'il y a 35 choix possible à ce niveau. Donc dans le pire des cas notre algorithme trouvera la solution et montrera qu'elle est unique avec un seul retour-arrière. Il devient même possible de résoudre le problème du zèbre à la main, en représentant les 4 contraintes binaires et les 10 graphes bipartis associés aux 13 contraintes de différence. Et de façon plus aisée que la méthode décrite par [Smith, 1992].

Ce résultat, trouver la solution du zèbre et montrer qu'elle est unique, en faisant dans le pire des cas un seul retour arrière est le meilleur résultat qui ait été trouvé jusqu'à maintenant. En effet le système CHIP résout le problème avec trois choix et cinq retours-arrière. Et [Smith, 1992] propose une méthode particulière au problème du zèbre qui nécessite, dans le meilleur cas, un retour-arrière. Mais la probabilité d'avoir un retour-arrière est beaucoup plus faible pour notre algorithme ($2/35 = 0.058$, soit moins de 6%).

1. Si on avait choisit la valeur 4 pour B_1C_2, qui est l'autre possibilité, cela aurait conduit immédiatement à un échec. Mais le choix suivant qui aurait résulté de sa suppression nous aurait donné la solution du problème et prouvé qu'elle est unique

Annexe C

Deux nouveaux ordres d'instanciations

Dans cette annexe nous présentons un nouvel ordre d'instanciation qui donne de bons résultats en pratique. Cet ordre essaie de combiner plusieurs types d'informations comme la taille des domaines et la structure du graphe des contraintes. Nous proposons aussi un nouvel algorithme calculant la largeur d'un graphe.

C.1 Association de la cardinalité du domaine et du degré des variables

Ce premier ordre a pour but de prendre en compte simultanément la taille des domaines et le degré des variables dans le graphe des contraintes.

Dans le chapitre 3 nous avons présenté l'ordre domaine-min qui consiste à choisir comme prochaine variable à instancier celle dont le domaine est le plus petit. Cet ordre est dynamique. Il donne, en général, de bons résultats. Mais il peut également en donner de très mauvais. D'une certaine manière nous ne pouvons pas considérer que c'est un ordre aux performances stables. Il permet essentiellement de limiter la largeur[1] de l'arbre de recherche. Le reproche essentiel que l'on peut lui faire est qu'il ne tient absolument pas compte de la structure du graphe des contraintes. Pour

1. La largeur exprime ici une notion purement visuelle.

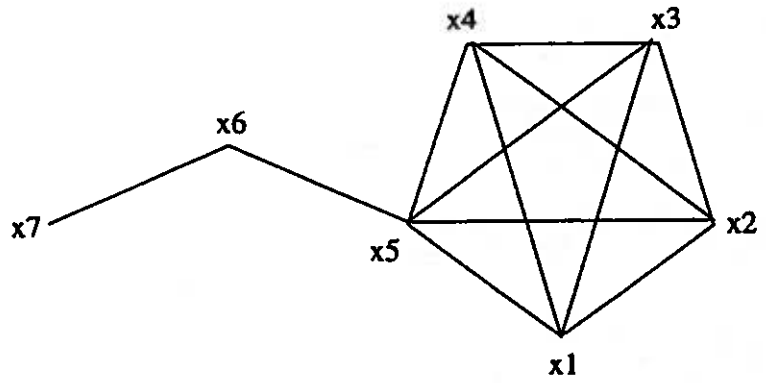


FIG. C.1 - *Un réseau de contraintes.*

mettre cela en évidence nous allons considérer le réseau donné en figure C.1.

La rapidité avec laquelle on va trouver une solution dépend de la valeur affectée à x_5, car seule la valeur e conduit à une solution. L'ordre "domaine-min" conduira à choisir d'abord la variable x_7 puis la variable x_6. Ces choix ne présente pas d'intérêt quant à la résolution du problème, car ils sont en dehors du coeur de difficulté du problème, qui est constitué par les variables x_1, x_2, x_3, x_4 et x_5. Il vaut beaucoup mieux choisir d'abord l'une de ces variables.

L'ordre que nous présentons ne résout pas tous les problèmes précédemment mentionnés, il essaie simplement de prendre en compte une information sur le graphe des contraintes : le degré des variables. Si l'on utilisait uniquement cette information on obtiendra probablement pour notre exemple un meilleur résultat que celui donné par l'ordre précédent. Mais on retrouverait le même type de défaut, à savoir ne plus considérer une information importante : les domaines et notamment leur évolution au cours de la résolution.

Notons $deg(x_i)$ le degré de la variable x_i dans le graphe des contraintes à la création du réseau. Nous proposons de prendre comme nouvelle variable à instancier celle dont le rapport de la taille de son domaine au carré par son degré est le plus faible. Le nouveau critère de choix est donc :

$$\frac{|D_i|^2}{deg(x_i)}$$

Ce critère combine donc deux paramètres important du réseau de contraintes. Notons que C. Bessière a proposé, dans sa thèse, d'utiliser un ordre statique basé sur le critère: $|D_i|/deg(x_i)$. Ce sont des essais expérimentaux qui nous ont amené à privilégié la taille des domaines par rapport au degré des variables, d'où l'apparition d'un carré. Il est important de remarquer que la taille des domaines peut varier pendant la recherche alors que le degré est toujours fixe. Aussi surprenant que cela puisse paraître au premier abord, le degré des variables ne doit pas refléter le nombre de voisins non instanciés. Supposons que l'on fasse un tel choix et reprenons l'exemple précédent. On va commencer par choisir x_5 puis on prendra certaines variables de la clique. Mais après en avoir choisi plusieurs on va décrémenter le nombre de variables voisines non instanciées des variables restantes et on risquera de quitter la clique pour choisir la variable x_7. En pratique ce phénomène se produit effectivement. De manière informelle on peut dire que cet ordre va tenter de résoudre des parties difficiles du CSP mais au moment où il sera presque parvenu à les résoudre il va instancier une variable n'ayant rien à voir avec la partie difficile en cours de traitement. Pour éviter cela il ne faut donc pas considérer les variables voisines non instanciées.

C.2 Un nouvel algorithme calculant la largeur du graphe des contraintes

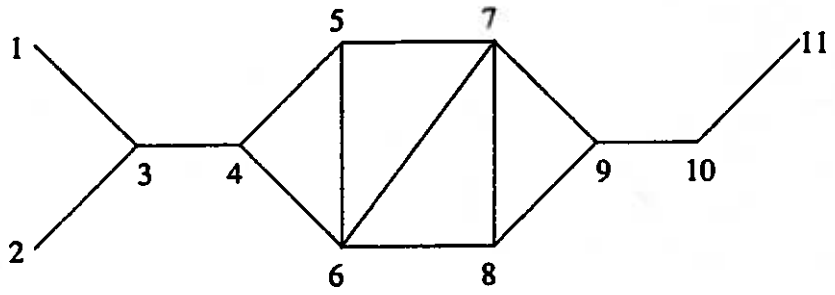
Freuder [Freuder, 1982] a introduit la largeur des graphes afin de caractériser certaines classes polynomiales.

Définition 51

• *La **largeur d'un sommet** dans un graphe ordonné est égale au nombre de voisin dans le graphe de ce sommet le précédent dans l'ordre.*

• *La **largeur d'un ordre** dans un graphe est égale à la largeur maximales de tous les sommets de cet ordre.*

• *La **largeur d'un graphe** est égale à la largeur minimale pour tous les ordres.*



Trace de l'algorithme :

k=0 : sommets éliminés : 1,2,3,11,10

k=1 : aucun sommet n'est éliminé

k=2 : sommets éliminés : 4,5,6,7,8,9

FIG. C.2 - *Calcul de la largeur d'un graphe.*

Freuder a donné un algorithme linéaire en la taille du graphe pour trouver sa largeur (cf algorithme 36).

Algorithme 36 Algorithme de calcul de la largeur d'un graphe.

```
LARGEUR(G) : entier
    k ← 0
    Enlever du graphe tous les sommets isolés
    tant que le graphe est non vide faire
    |   k ← k + 1
    |   tant que il existe des sommets connectés à au plus k autres faire
    |   |   Enlever ces sommets du graphe
    retourner k
```

L'ordre obtenu en prenant les sommets dans l'ordre inverse de l'ordre dans lequel ils ont été enlevés par l'algorithme est un ordre de largeur minimum.

Dans l'exemple C.2 la largeur du graphe est 2 et l'ordre 9 8 7 6 5 4 10 11 3 2 1 est un ordre de largeur 2. Le résultat de l'algorithme dépend de l'ordre suivant lequel les sommets sont choisis. Ainsi l'ordre 4 5 6 7 8 9 1 2 3 11 10 est également de largeur 2. Cela est relativement gênant, c'est pourquoi nous proposons un autre

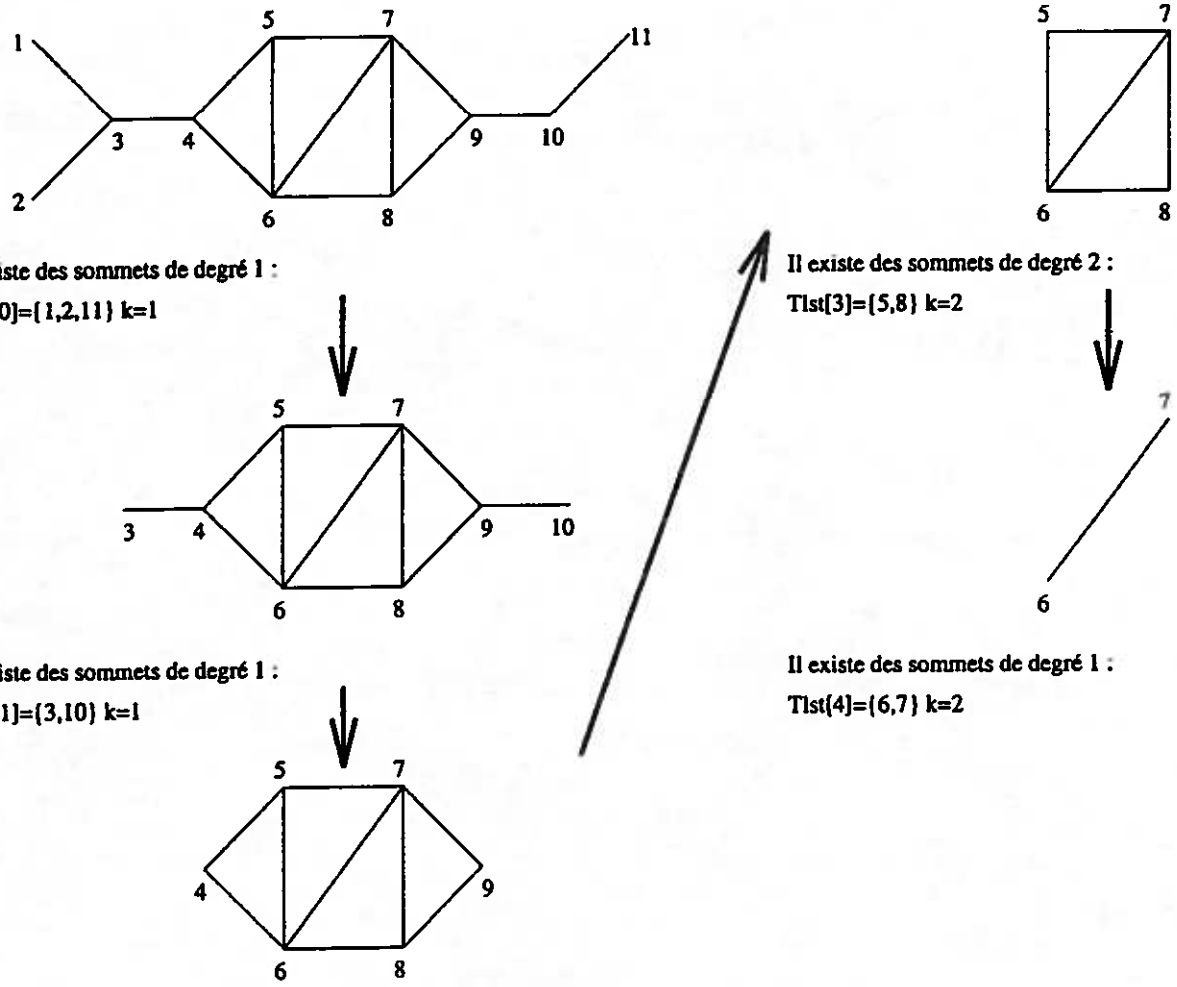


FIG. C.3 - *Trace du nouvel algorithme de calcul de la largeur d'un graphe.*

algorithme de calcul de la largeur d'un graphe. Il utilise un tableau de listes ($Tlst$) comme structures de données.

Algorithme 37 Nouvel algorithme de calcul de la largeur d'un graphe.

```
LARGEUR(G, Tlst) : entier
    ∀i = 0, .., n : Tlst[i] ← ∅
    i ← 0
    tant que le graphe est non vide faire
        j ← 0
        tant que il n'existe pas de sommets de degré j faire j ← j + 1
        si j > k alors k ← j
        Mettre dans Tlst[i] tous les sommets de degré j
        Supprimer tous les sommets de Tlst[i]
        i ← i + 1
    retourner k
```

La figure C.3 donne la trace de cet algorithme pour le graphe précédent. L'ordre donné par les listes du tableau en suivant l'ordre décroissant des indices est un ordre de largeur minimum. Ainsi l'ordre 6 7 5 8 4 9 3 10 1 2 11 est un ordre de largeur minimum. Quelque soit l'ordre dans lequel les sommets sont supprimés le tableau de listes sera toujours le même. Comme les listes ne sont pas nécessairement réduites à un seul élément, nous pourrons obtenir quand même plusieurs ordres de largeur minimum.

À partir de la remarque précédente, on peut imaginer un nouvel ordre dynamique en choisissant les variables selon l'ordre précédemment calculé et en différenciant les sommets appartenant à une même liste par la taille de leur domaine. Ce dernier critère étant choisi de façon dynamique.

C.3 Résultats expérimentaux

Dans les chapitres précédents, tous résultats que nous avons donné pour les algorithmes de type MAC ont été calculés en utilisant le nouvel ordre présenté dans cette annexe.

Pour les 10 000 instances de lso-ssgrpartiel, le degré est calculé uniquement par rapport aux contraintes d'adjacence. On observe une perte de 5% en temps avec l'ordre domaine-min et de 25% avec l'ordre de largeur minimum tel que nous proposons de le calculer.

Pour trouver une solution pour l'instance 11 des problèmes d'assignement de fréquences[2] , il faut à MAC-Ipsid :

- plus de 15h avec domaine-min;
- 314s avec un ordre statique qui considère les sommets suivant l'ordre décroissant de leur degré;
- 10s avec le nouvel ordre proposé;
- 2,9s avec l'ordre de largeur minimum à condition de le calculer selon notre méthode.

Avec le dernier ordre mentionné le problème, est résolu sans aucun retour-arrière. C'est le meilleur résultat obtenu à ce jour.

Pour finir nous proposons pour les réseaux aléatoires ayant 20 variable et 10 valeurs qui ont été définis dans le chapitre 6, une comparaison entre notre nouvel ordre et domaine-min (cf figure C.4). Chaque point représente la moyenne obtenue pour 30 instances se trouvant sur la phase de transition.

Il apparaît très clairement que l'ordre que nous proposons a de bonnes performances par rapport à domaine-min.

Pour conclure cette annexe nous dirons simplement que la combinaison de la taille des domaines et du degré des variables dans le graphe des contraintes est une idée qui mérite d'être approfondie.

2. toutes les autres instances sont très rapidement résolues quelque soit l'ordre choisi

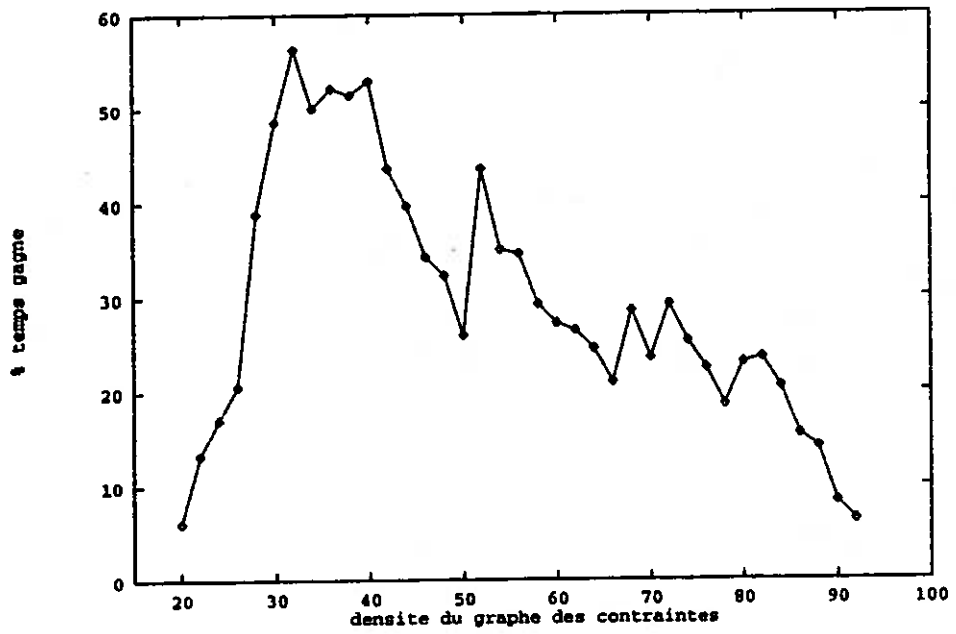


FIG. C.4 - *Pourcentage de temps gagné par le nouvel ordre d'instanciation par rapport à domaine-min pour la recherche d'une solution dans des réseaux de contraintes ayant 20 variables et 10 valeurs.*

Deuxième partie

Apprentissage des liaisons stratégiques en synthèse organique

Chapitre 1

Introduction

Le problème que je traite dans cette partie est celui de la détermination des liaisons stratégiques dans des molécules en synthèse organique.

Sans rentrer dans le détail, car c'est le propos du chapitre 2, une synthèse en chimie organique est une séquence de réactions permettant de créer une molécule complexe, appelée molécule cible, à partir de produits disponibles dans le commerce ou faciles à obtenir. Les liaisons stratégiques d'une molécule cible sont les liaisons créées au cours de la dernière étape d'une synthèse, autrement dit créées par la dernière réaction utilisée. Ce sont donc les plus importantes.

Le projet, peut-être un peu trop ambitieux, que j'ai commencé en DEA, est de résoudre ce problème en créant un système utilisant le même langage que celui des chimistes. Pour mener à bien ce projet, j'ai été assisté d'O. Gascuel, mon directeur de thèse et surtout de C. Laurenço, un expert de la synthèse organique. Les chimistes raisonnent à partir de formules moléculaires, qui sont, pour un informaticien, des graphes ayant des sommets et des arêtes étiquetés. Le premier inconvénient de cette approche est que la manipulation de graphes s'accompagne d'une forte complexité des calculs. Par exemple, comme nous l'avons vu dans la première partie de cette thèse, le problème de l'isomorphisme de sous-graphe partiel est NP-Complet.

Par ailleurs, comme le chimiste n'hésite pas à faire appel à des connaissances passées pour résoudre une nouvelle synthèse, et puisque l'on dispose d'énormes bases de réactions (que l'on peut voir comme des synthèses à une étape) nous avons pensé qu'un système d'apprentissage basé sur la recherche de ressemblances entre les molé-

cules serait un bon choix pour résoudre notre problème. Aussi nous avons commencé par utiliser la méthode ANNA. Mais elle s'est vite avérée insuffisante car les explications qu'elle donnait étaient insuffisantes [Régin *et al.*, 1994]. Il fallait donc développer un nouveau système d'apprentissage. Mais la difficulté majeure d'une telle approche est qu'il faut en parallèle construire un système d'apprentissage performant pour le problème considéré et définir les connaissances du domaines qu'il faut prendre en compte pour résoudre le problème. Les interactions entre la construction et la modélisation sont nombreuses et difficiles à gérer. Ces deux aspects sont présentés, dans cette partie, de façon bien séparée, mais dans la réalité il n'en a pas été de même. En effet, pour arriver à dissocier ces deux aspects nous avons fait appel à un étudiant de DEA (C. Fagot) afin d'étudier et d'améliorer notre système pour des problèmes jouets ayant les mêmes caractéristiques que notre problème initial. Je pense que grâce à C. Fagot, nous avons partiellement atteint l'objectif que nous nous étions fixés.

Le plan de cette partie est le suivant :

Dans le chapitre 2, je détaille le problème de la détermination des liaisons stratégiques en synthèse organique. J'explique, pour un non initié, ce qu'est une synthèse en chimie organique et une liaison stratégique. Je montre aussi pourquoi nous avons choisi d'utiliser une représentation symbolique des données plutôt qu'une représentation numérique.

Dans le chapitre 3, je présente les principes de l'apprentissage conceptuel. Puis, je donne brièvement les mécanismes de base des méthodes de voisinage. Enfin, je détaille le fonctionnement de la méthode ANNA, qui a été la source d'inspiration de la nouvelle méthode que nous avons développée.

Dans le chapitre 4, je propose une nouvelle méthode d'apprentissage appelée CNN. C'est une méthode conceptuelle de voisinage. Je détaille ses différents mécanismes et je donne les résultats qu'elle a obtenus pour des problèmes tests.

Dans le chapitre 5, je montre comment nous avons modélisé le problème. Je mets en évidence certains problèmes qui ont surgi lors de cette modélisation et j'explique pourquoi nous avons pris en compte les effets électroniques que subissent les atomes. Je présente une adaptation de l'algorithme de recherche de cliques maximales de Bron et Kerbosh au problème de la recherche de sous-graphes partiels communs maximaux

et connexes.

Dans le chapitre 6, les résultats obtenus pour notre problème sont présentés et commentés. Ils sont plutôt satisfaisants. Le reclassement des exemples de l'ensemble d'apprentissage est excellent et les règles apprises ont une réelle pertinence chimique. Je ne manque pas de mettre en évidence un problème encore non résolu.

Enfin le chapitre 7 conclut cette partie.

Chapitre 2

Présentation du problème

2.1 La synthèse en chimie organique

Cette présentation s'inspire de celles faites par C. Laurenço lors des différents conseils scientifiques du GDR TICCO.

2.1.1 Introduction

La synthèse chimique se distingue tout d'abord par son importance économique. La plupart des secteurs industriels - pharmacie, agrochimie, parfumerie, alimentation, photographie, électronique, matériaux, etc - emploient, transforment ou produisent des molécules de synthèse. A titre d'exemple, pour lancer un nouveau médicament, il faut synthétiser et tester l'activité biologique de quelques 10 000 composés, ce qui au total prend une douzaine d'années et coûte plus d'un milliard de francs.

La synthèse est également l'un des moteurs de la découverte en chimie. La préparation de molécules naturelles complexes (chlorophylles, stéroïdes, alcaloïdes, érythronolides, ...) permet non seulement de trouver de nouveaux réactifs et de nouvelles voies de synthèse ayant un intérêt général, mais aussi de prouver l'exactitude des structures moléculaires et de développer des théories qui contribuent à la compréhension de la chimie. Par ailleurs, la préparation de molécules jusque-là hypothétiques (prismane, propellanes, ...) permet de vérifier ou de réfuter des théories existantes.

La synthèse d'une molécule organique complexe est un défi lancé à l'habileté et à l'intelligence du chimiste. Pour résoudre un problème de synthèse, celui-ci s'appuiera

évidemment sur la théorie structurale et électronique de la chimie organique. Mais cette théorie, encore fort incomplète, est plus apte à expliquer qu'à prédire un résultat, à indiquer ce qui ne peut pas être fait qu'à montrer la meilleure façon d'obtenir ce qui est faisable. Aussi, le chimiste fera très largement appel à la masse considérable de données expérimentales, accumulées depuis le siècle dernier au cours de la synthèse de plus de 10 millions de molécules, qu'il trouvera dans la littérature chimique, afin d'élaborer des solutions selon un raisonnement par analogie.

Un tel domaine, où les problèmes rencontrés sont très complexes, où la théorie est incomplète, où la connaissance est vaste et hiérarchisée dans de multiples classifications, où le raisonnement est à la fois logique et analogique, et où le langage d'expression (basé sur les formules structurales) emprunte son formalisme à la théorie des graphes, est un champ privilégié de recherche et d'application pour divers aspects de l'informatique: algorithmique, bases de données, acquisition et représentation de connaissances, raisonnement, etc. Depuis une trentaine d'années, de nombreux travaux, dus principalement à des chimistes, ont été consacrés au développement de systèmes informatiques de gestion de données chimiques ou d'aide à la résolution de différents problèmes, notamment ceux posés par la synthèse organique. Certains de ces travaux ont permis de réaliser des outils dont l'efficacité est incontestable (DARC, CAS ONLINE, REACCS, DENDRAL, LHASA, etc). Cependant, beaucoup reste encore à faire pour obtenir des systèmes plus «intelligents», mieux adaptés aux besoins de la chimie organique. Certains aspects de la connaissance chimique devraient être mieux formalisés, c'est le cas des stratégies de synthèse. Nous allons montrer dans la suite comment des techniques informatiques avancées, particulièrement celles qui sont développées en intelligence artificielle, devraient venir remplacer les techniques classiques qui ont montré leurs limites.

2.1.2 Le problème de la synthèse

La résolution d'un problème dans un domaine quelconque se déroule généralement selon la séquence suivante:

1. Compréhension du problème
2. Conception d'un plan

3. Exécution du plan.

4. Évaluation de la solution.

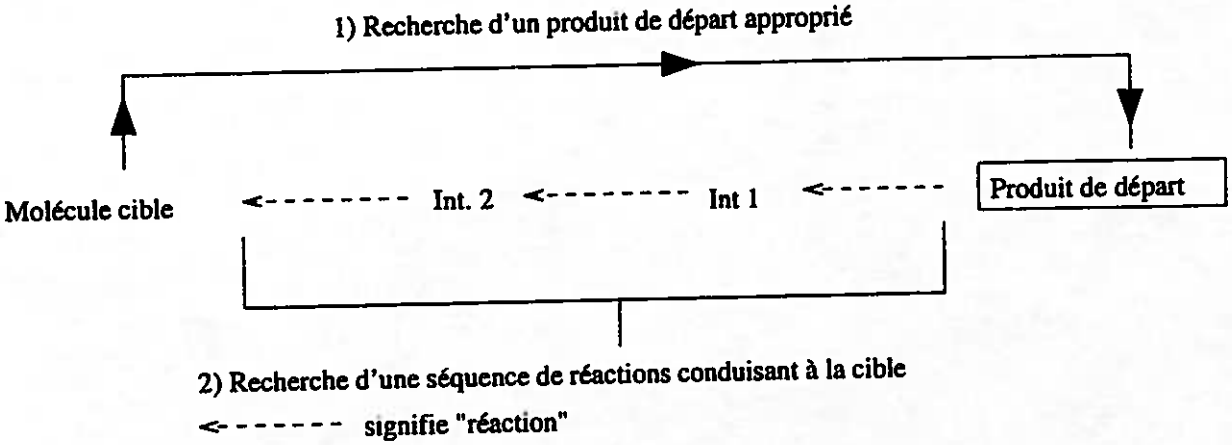
5. Acquisition d'expérience.

Dans le cas d'un problème complexe, il est rarement possible d'exécuter ces actions dans l'ordre, sans retour en arrière pour clarifier, affiner ou remettre en cause certaines interprétations ou hypothèses. Néanmoins, le succès de chaque étape dépend du succès de celle qui la précède dans cette séquence. Ainsi, quelle que soit l'habileté du chimiste, l'exécution d'un plan mal conçu n'aboutira jamais à une bonne synthèse.

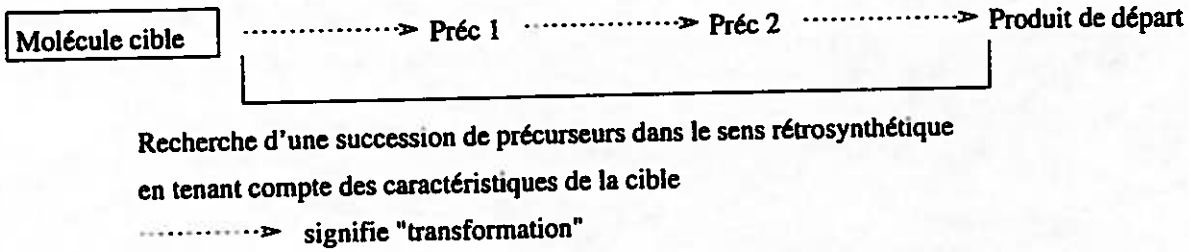
Un système d'aide à la synthèse doit pouvoir intervenir à chaque étape, à l'exception de celle d'exécution du plan qui est la phase «expérimentale» du processus. Il doit pouvoir aider à comprendre le problème en percevant les caractéristiques de la molécule à préparer. Il doit aussi permettre d'élaborer des plans de synthèse en procédant de manière systématique et en proposant des hypothèses auxquelles le chimiste n'aurait pas forcément pensé. Enfin, grâce à un tel système, on doit avoir la possibilité d'évaluer des solutions et d'acquérir, de conserver et de transmettre une connaissance d'experts.

La synthèse d'une molécule est un problème qui appartient à la classe des problèmes de transformation. Trouver une solution à un tel problème consiste à découvrir une séquence d'opérations reliant un état initial à un état final. Le triplet {état initial, état final, opérateurs} définit l'espace d'états (ensemble des configurations du problème : solutions et impasses). La recherche de solutions peut se faire en parcourant l'espace d'états du problème initial vers l'état final ou de l'état final vers l'état initial ou encore dans les deux directions à la fois. Les principales techniques de recherche de solutions sont :

- le parcours aveugle (force brute) qui n'utilise pas d'autre connaissance que celle contenue dans l'espace d'états et un test d'arrêt;
- l'utilisation d'heuristiques qui introduit dans l'espace d'états une connaissance permettant d'estimer le mérite des solutions en cours de développement;
- la réduction de l'espace de recherche par laquelle on se concentre sur ses caractéristiques essentielles et on diffère le traitement des détails.



Démarche classique



Analyse rétrosynthétique

FIG. 2.1 - *Synthèse et Rétrosynthèse.*

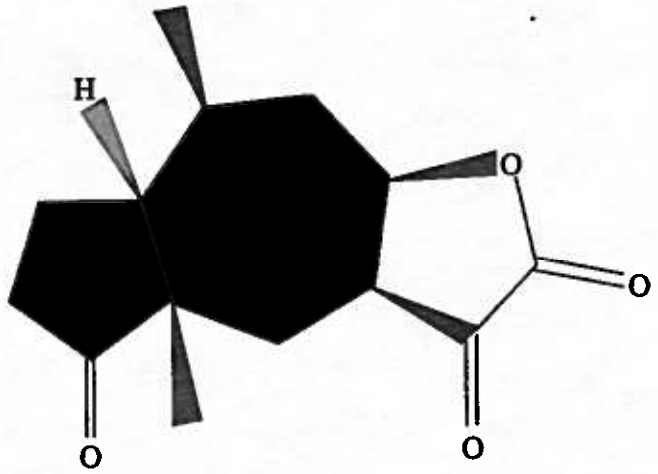


FIG. 2.2 - *La Confertine.*

En synthèse, il faut découvrir un chemin de synthèse, c'est-à-dire une séquence de réactions chimiques conduisant d'un ensemble de produits de départ (composés commerciaux ou faciles à obtenir) à la molécule qu'on veut préparer. Un chemin de synthèse peut comporter aussi bien 2 que 20 réactions logiquement enchaînées. Sa taille est fonction de la complexité de la molécule cible, de la nature des produits de départ et de l'ingéniosité du chimiste. Le plus souvent, seul l'état final d'un problème de synthèse (la molécule cible) est bien défini. Pour trouver des chemins de synthèse, on raisonne alors de manière analytique (rétrosynthétique selon la terminologie de Corey [Corey and Cheng, 1989]), autrement dit de la molécule cible vers les produits de départs. La figure 2.1 met en évidence les différences de ces deux modes de raisonnement.

Dans le cas d'une approche rétrosynthétique, de taille de l'espace d'états du problème pourra être considérable. Prenons comme exemple la Confertine (cf figure 2.2). Le squelette de cette molécule (abstraction faite des hydrogènes et des hétéroatomes terminaux – les trois oxygènes doublement liés) possède 18 liaisons. Dans les synthèses connues, 6 ou 7 liaisons, en moyenne ont été créées pour construire la molécule. Le nombre de séquences de 7 liaisons à créer parmi 18 est égal à $A_{18}^{7} = 18!/(18-7)!$, soit plus de 160 millions. Si la création de chaque liaison est de plus réalisable par 5 réactions différentes il faudra multiplier ce nombre par 5^7 pour obtenir celui des solutions possibles, soit plus de 10^{13}. La taille d'un tel espace rend évidemment inapplicable les techniques de recherche par force brute. Les experts de la synthèse emploient tout

un arsenal de connaissances stratégiques et tactiques pour sélectionner les voies à explorer parmi toutes celles qui sont possibles mais qui sont inégalement pertinentes.

2.2 Le problème et ses hypothèses

Concevoir un plan de synthèse pour une molécule complexe par une analyse rétrosynthétique conduit inévitablement à une explosion combinatoire. Aussi, pour résoudre un tel problème dans un temps raisonnable, il est nécessaire d'utiliser des stratégies pouvant guider cette analyse. Les avancées les plus remarquables ont été réalisées par Corey [Corey and Cheng, 1989], mais ce domaine très complexe reste encore partiellement formalisé. Les connaissances stratégiques dans l'élaboration de synthèses organiques prennent divers aspects et s'échelonnent sur plusieurs niveaux. Avoir une vue globale d'un problème de synthèse n'est pas aisé et la découverte de nouveaux plans de synthèse résulte le plus souvent de l'utilisation concurrente de plusieurs types de stratégies (topologique, stéréochimique, etc...). Les stratégies spécifient et ordonnent logiquement un ensemble de buts à atteindre. Un but peut être formulé en termes d'actions, par exemple casser un système cyclique, supprimer un stéréocentre etc... ***Dans le sens rétrosynthétique, casser une liaison est un but élémentaire.*** Il peut être inclus dans un but plus complexe, comme par exemple, casser un cycle. L'identification des liaisons stratégiques, c'est-à-dire des liaisons dont la déconnexion réduit la complexité de la molécule, est basée sur des critères variés. Des approches heuristiques [Corey *et al.*, 1975; Corey and Cheng, 1989] et algorithmiques [Baumer *et al.*, 1988] ont été proposées afin de percevoir de telles liaisons à partir des caractéristiques topologiques et stéréochimiques de la molécule cible. Nous proposons une autre approche basée sur la reconnaissance de similitudes entre un nouveau problème de synthèse et des problèmes déjà résolus.

Une analyse rétrosynthétique est un processus récursif. Chaque précurseur engendré à partir de la structure cible peut être considéré comme un nouveau but pour les futures analyses. Ainsi, un chemin de synthèse peut être défini comme un arbre de réactions dont la racine est la dernière étape de la synthèse, chacun des autres noeuds étant la racine d'un sous-arbre, autrement dit chaque autre réaction est la dernière étape d'une sous-synthèse. Deux classes de liaisons existent dans le produit

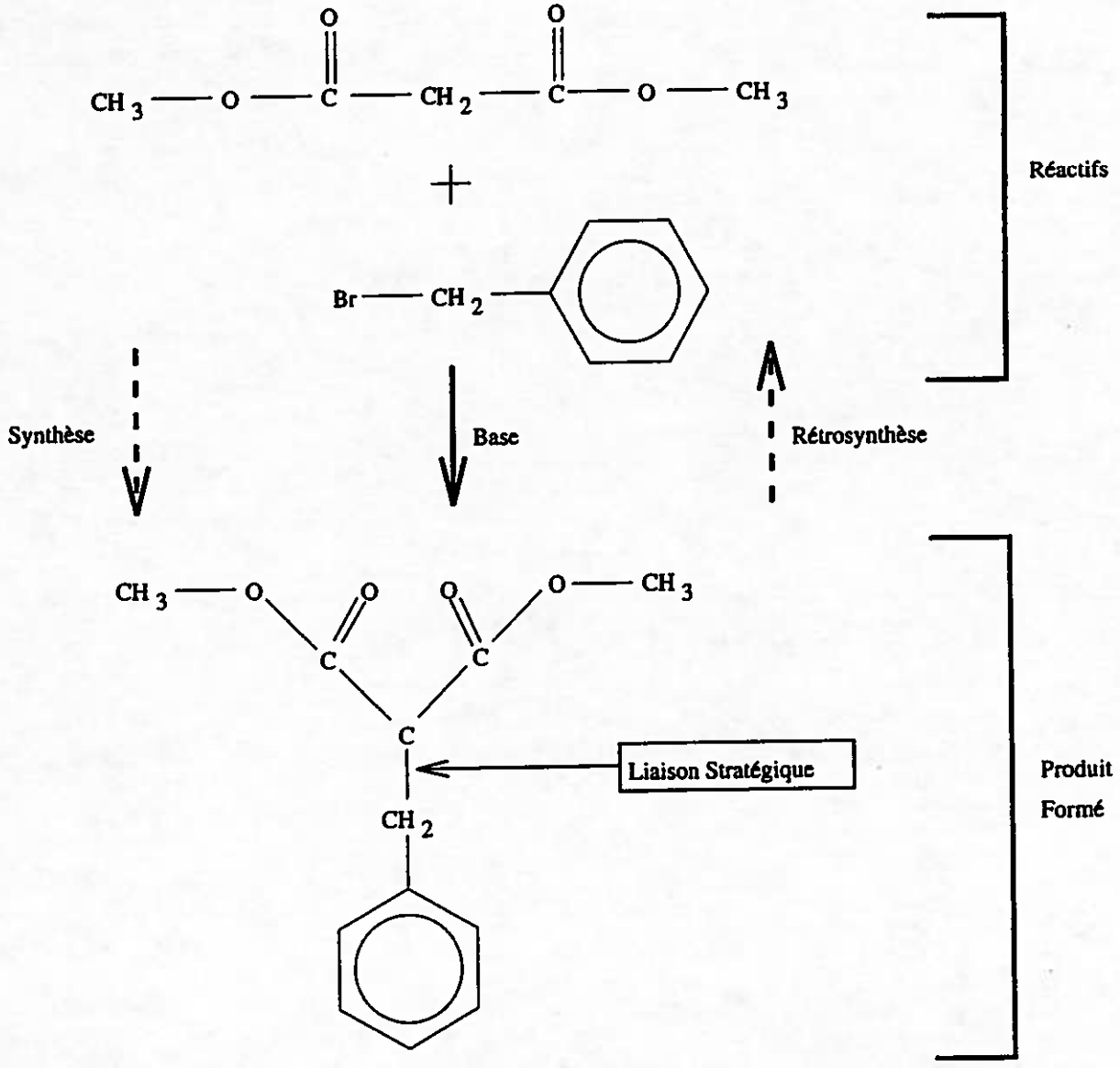


FIG. 2.3 - *Une réaction.*

d'une réaction : les liaisons créées par la réaction et les autres (cf figure 2.3). Les premières ont une grande importance stratégiques car ce sont les buts à atteindre par les méthodes de synthèse, tandis que les secondes ont peu d'intérêt ou un intérêt inconnu. ***Le caractère stratégique d'une liaison est fortement lié au contexte dans lequel elle apparaît***, c'est-à-dire à son environnement structural. Aussi, pour être capable de prédire le caractère stratégique d'une liaison d'une molécule cible donnée, nous devons apprendre à partir d'exemples les caractéristiques des contextes favorables et défavorables. Les très grandes bases de données de réactions disponibles vont nous fournir les exemples nécessaires au processus d'apprentissage. Nous partons avec l'hypothèse que les bases de données de réactions contiennent des instances de méthodes de synthèse et chacune d'entre elles est la dernière étape de la synthèse.

Nous pouvons résumer le problème de la façon suivante :

> Données : On dispose d'une base de données de molécules pour lesquelles on connaît la ou les liaisons qui sont créées au cours d'une synthèse, et d'une molécule m pour laquelle on ne dispose pas d'une telle information.
>
> Hypothèse : Le caractère stratégique d'une liaison dépend de son environnement structural.
>
> Questions : Quelle est la liaison de m qui est stratégiquement la plus intéressante (autrement dit qu'il faudrait déconnecter durant une analyse rétrosynthétique de m) ? Peut-on classer les liaisons de m en fonction de leur importance stratégique ? Comment explique-t-on les réponses données aux deux questions précédentes ?

L'objet de la seconde partie de cette thèse est de proposer un système capable de répondre à toutes ces questions.

Nous allons dans le paragraphe suivant montrer pourquoi nous avons besoin d'utiliser les données sous une forme symbolique.

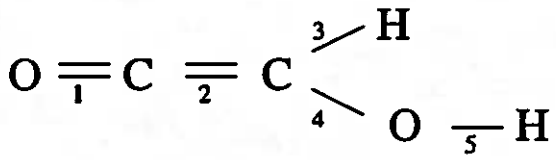


FIG. 2.4 - *Une molécule.*

2.3 Intérêt d'une représentation symbolique par rapport à une représentation numérique en chimie organique

Les méthodes numériques (càd celles qui manipulent uniquement des nombres) sont limitées à des descriptions de type attributs-valeurs (proche de la logique des propositions), tandis que les méthodes symboliques (càd celles qui manipulent les données représentées sous une forme symbolique) permettent d'exploiter des descriptions structurelles (proches de la logique des prédicats). Une description structurelle considère un objet non plus seulement en fonction des ses caractéristiques mais aussi en fonction de sa structure c'est à dire des liens existants entre ses composants. Tout ce qui est descriptible par une description attributs-valeurs peut l'être par une description structurelle, mais l'inverse n'est pas vrai.

Une formule réduite $C_2O_2H_2$ est exactement une représentation par attributs-valeurs, alors que la formule développée est une représentation structurelle. Considérons par exemple la molécule donné en figure 2.4.

Cette molécule sera représentée par un graphe où les sommets et les arêtes sont étiquetés. Il est très difficile de représenter cette molécule sous forme de vecteurs d'attributs, car dans ce type de représentation le nombre et le type des attributs est déterminé à l'avance. On pourrait, par exemple, disposer des attributs suivants :

- **numériques** : nombreC, nombreH, ... (on énumère tous les éléments), nombre-liaisonO=C, nombre-liaisonC-C, ... (on énumère tous les types de liaisons).

- **booléens** : groupement-alcool, groupement-cétone, ...

La combinatoire de ce type de représentation est énorme. De plus il est impossible de rajouter une connaissance, par exemple un groupement oublié. L'emploi de

descriptions structurelles semble donc plus approprié.

Par ailleurs, en synthèse organique le caractère stratégique d'une liaison peut difficilement se déduire de paramètres physicochimiques. Il semble plus simple de l'estimer à l'aide de critères structurels, en s'appuyant sur l'analogie existant entre des cas connus et le problème à résoudre. On cherche à apprendre des connaissances de nature symbolique (des règles) à partir de données elles-mêmes de nature symbolique (des molécules). De plus, les chimistes utilisent naturellement des représentations symboliques et structurées d'objets complexes. Or il est beaucoup plus facile d'expliquer quand on parle le même langage que les experts, donc l'utilisation d'une méthode symbolique s'impose.

2.4 Conclusion

L'un des points essentiels de la synthèse en chimie organique est la reconnaissance des liaisons stratégiques. L'état actuel des connaissances nous permet de penser que le caractère stratégique d'une liaison dépend de son environnement structurel. Aussi, si l'on arrive à déterminer les environnements stratégiquement favorables ou défavorables, on aura résolu ce problème. Malheureusement on ne sait pas définir cette connaissance. Par contre on dispose d'une grande masse d'information, à l'état brut, fournie sous la forme de base de millions de réactions. Si l'on arrive à expliciter sous forme de règles les connaissances implicitement contenues dans ces bases nous aurons atteint notre but. Or ce type d'extraction est justement le propos des méthodes d'apprentissage conceptuel, d'autant plus que les données manipulées sont de type symbolique. C'est pourquoi nous avons choisi cette approche. Les principes généraux de ces méthodes sont exposés dans le chapitre suivant.

Chapitre 3

Apprentissage conceptuel et méthodes de voisinage

L'apprentissage est une branche importante de l'Intelligence Artificielle. De manière générale, on peut le définir comme le processus qui permet de tirer bénéfice d'un ensemble de situations déjà résolues, pour traiter des situations nouvelles. Apprendre une règle c'est, par exemple, utiliser les descriptions et diagnostics d'un ensemble de patients, pour construire une règle de décision qui permettra de prédire le diagnostic de nouveaux patients. Des systèmes d'apprentissage remarquables tels que les sytèmes AQ de l'équipe Michalski, ID3 et FOIL de Quinlan et RISE de Domingos[1] ont d'ailleurs été réalisés pour des domaines aussi variés que la reconnaissance des caractères, le traitement de la parole ou l'analyse des séquences biologiques.

Nous nous intéresserons ici a une forme d'apprentissage particulière, qui est «l'apprentiss de concept à partir d'exemples». On suppose que les exemples (ou situations déjà résolues suivant notre appellation antérieure) sont des couples (description, classe). La description permettra, par exemple, de représenter les symptômes d'un patient, tandis que la classe correspondra à son diagnostic. Nous supposerons qu'il n'y a que deux classes, notées C_+ et C_-. Les exemples de C_+ seront appelés les «exemples positifs du concept à apprendre», tandis que les exemples de C_- seront «les exemples négatifs». À titre d'illustration dans la figure 3.1, l'objectif est d'apprendre le concept d'équilibre dans un monde de blocs simplifié. Les exemples positifs correspondent à

1. Ces méthodes, ainsi que d'autres, sont expliquées dans [Thrun *et al.*, 1991].

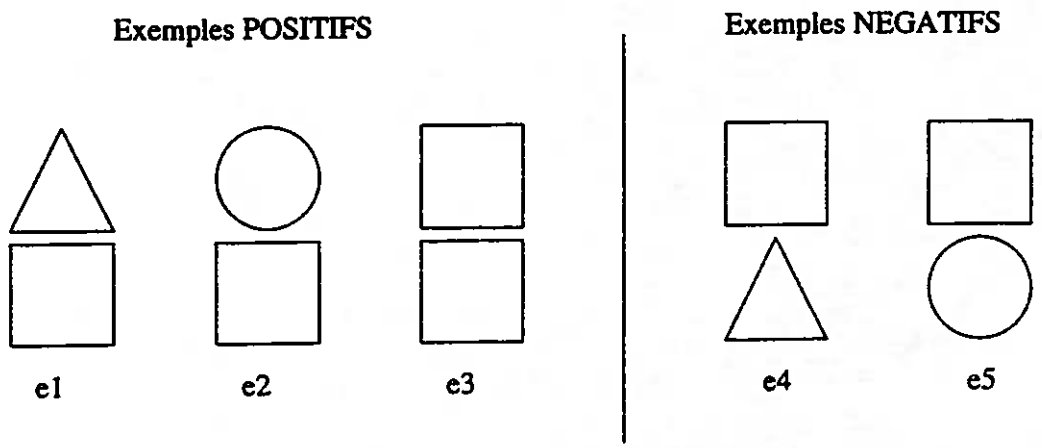


FIG. 3.1 - *Ensembles d'exemples positifs et négatifs du concept d'équilibre.*

des empilements stables d'objets, et inversement les exemples négatifs sont instables. L'objectif poursuivi par l'apprentissage est double :

- On recherche une explication intéressante et non-triviale de la partition observée entre exemples positifs et négatifs ;
- On souhaite que cette explication, ou règle de classement, permette de reclasser sans erreur, ou avec une faible probabilité d'erreur, une nouvelle description dont la classe est inconnue.

Ces deux aspects sont bien sûr liés. Une explication triviale, comme par exemple celle constituée par la disjonction des exemples positifs, ne permettra pas de reclasser correctement de nouveaux exemples. En outre, une règle de classement efficace contient certainement, même si c'est de manière implicite, une certaine connaissance sur le concept à apprendre.

L'apprentissage de concept tel que nous l'avons défini jusque là, s'apparente donc à ce qu'en statistiques on appelle un problème de discrimination à deux classes, et pour lequel il existe de très nombreuses méthodes de résolution (certaines seront décrites en section 3). Les spécificités de l'approche IA de l'apprentissage de concept tiennent avant tout :

- Au mode de description des exemples. Comme nous le verrons, ces méthodes permettent de traiter des données symboliques complexes, comme par exemple des molécules données en formule développée, donc des graphes étiquetés.

- Au langage d'expression des règles de classement apprises. La encore, ces méthodes sont capables d'utiliser des langages symboliques évolués de représentation, du type graphe ou proche de la logique du premier ordre.

- À la volonté d'obtenir des sorties explicatives, ce qui impose, entre autres, de conserver autant que faire se peut le langage de l'utilisateur, d'exprimer les règles de classement dans ce langage, de limiter le nombre de ces règles, d'adopter des procédures simples de décision à partir d'un ensemble de règles, etc...

Comme nous le verrons (en section 1) il existe plusieurs grandes approches d'apprentissage conceptuel. Celle que nous avons retenue ici est relativement peu employée, et s'apparente au «raisonnement par analogie». Pour classer une nouvelle description, on s'appuie sur les exemples qui lui sont proches, dans un certain sens que nous préciserons. Le pendant de ce type d'approche dans le monde statistique est constitué des «méthodes de voisinages», ou encore «d'estimation locale de la densité», dont les représentants les plus connus sont la méthode «des k plus proches voisins» et «les noyaux de Parzen». Ces méthodes s'appuient sur une distance pour définir la notion de proximité. Dans le cas où les descriptions sont des points de $\mathbb{R}^n$, on utilise généralement la distance euclidienne. La difficulté quand on se place dans le cadre de descriptions complexes, du type graphe ou autres, est qu'il n'existe pas de distance naturelle. Par exemple en chimie, dans certains contextes on pourra considérer que deux molécules sont très proches parce qu'elles contiennent toutes deux un groupement alcool, alors qu'en fait elles sont très différentes du point de vue de la structure et de la nature des atomes qu'elles contiennent. Il est par conséquent très délicat de définir une distance pertinente pour le problème posé, et donc pratiquement impossible d'utiliser une méthode comme les k plus proches voisins. Le traitement de cette difficulté dans le cadre d'une approche de type conceptuelle a été abordé dans la méthode ANNA [Gascuel, 1985b; Gascuel, 1985a], sur laquelle nous nous sommes appuyés.

On trouvera dans la suite de ce chapitre introductif 4 parties principales. La première décrit un problème d'apprentissage simple qui nous a servi pour mettre au point notre méthode, les problèmes liés à la chimie étant trop complexes à mettre en oeuvre dans une phase préliminaire (section 1). La deuxième donne les concepts de base de l'apprentissage conceptuel à partir d'exemples (section 2). La troisième

FIG. 3.2 - *Représentation des chiffres par des digits.*

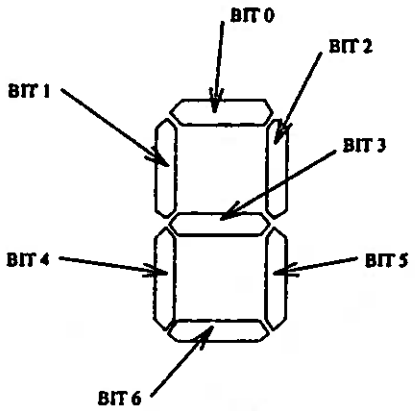


FIG. 3.3 - *Représentation d'un digit par un vecteur de 7 bits.*

décrit dans leurs grandes lignes les méthodes de voisinage (section 3). La quatrième détaille la méthode ANNA, que l'on peut qualifier de «méthode conceptuelle de plus proches voisins», et qui a été la première méthode que nous avons testée (section 4).

3.1 Un exemple simple

Nous avons choisi d'illustrer notre exposé à l'aide du problème des digits (LED displays). Ce problème a été introduit par [Breiman *et al.*, 1984], et se rapporte aux chiffres que l'on voit sur les cadrans digitaux (figure 3.2). Nous avons choisi, ici, d'apprendre le concept «chiffres pairs / impairs» en considérant chaque digit comme un vecteur de bits. Chaque bit correspond alors à une LED et si un bit est à 1, cela signifie que la LED est allumée (figure 3.3). Dans un premier temps, nous avons pris des vecteurs de 7 bits (un par LED), puis nous avons ajouté des attributs inutiles sous la forme de 4 bits supplémentaires affectés aléatoirement. De plus, lors de la génération des exemples, du bruit peut être apporté sur les valeurs des digits de référence. Pour cela, on introduit un taux d'erreur que l'on applique à chaque bit lors de la génération d'un exemple. Il représente la probabilité d'avoir ce bit mal positionné

dans sa description. Si, par exemple, on choisit 10% de bruit, chaque bit aura une chance sur dix de voir sa valeur faussée. Cela signifie que le 0 peut se retrouver avec une barre au milieu, ou perdre sa barre du haut. Un 8 peut donc représenter un 9 bruité, et ainsi faire passer un élément de la classe des impairs pour un de la classe des pairs.

Les notations suivantes seront employées dans la suite de ce chapitre :

Notations 2

- *E représente l'ensemble d'apprentissage;*
- *C_+ représente la classe des exemples positifs et C_- celle des négatifs;*
- *ν est un exemple que l'on doit classer.*

3.2 Principes de l'apprentissage conceptuel

Cette section présente, de manière générale, les notions au centre de l'apprentissage conceptuel. Le lecteur intéressé par une présentation plus détaillée pourra consulter le cours de Nicolas [Nicolas, 1993], ou celui de Quinqueton [Quinqueton, 1990]. Nous commençons par introduire l'apprentissage conceptuel à partir d'exemples en détaillant ses mécanismes de base. Puis nous caractériserons les méthodes d'apprentissage selon la stratégie de construction des généralisations qu'elles utilisent et selon la manière dont les exemples sont considérés.

3.2.1 Mécanismes de bases

Les systèmes d'apprentissage s'articulent autour de la notion de généralisation. Mitchell [Mitchell, 1982] a défini le problème de la généralisation comme une recherche dans un espace d'hypothèses possibles. Il le formalise de la façon suivante :

Étant donné :

- un langage LI pour décrire les exemples;
- un langage LG pour décrire les généralisations (descriptions de concepts);
- une relation d'appariement faisant correspondre exemples et généralisations, c'est-à-dire un mécanisme permettant de reconnaître si un

exemple est couvert par un concept et si un concept est plus général qu'un autre;

- un ensemble d'exemples positifs et éventuellement négatifs d'un concept à apprendre;
- un critère de consistance

Déterminer : les généralisations faisant partie du langage considéré, consistantes avec les exemples présentés.

Nous allons par la suite détailler les différents types de raisonnement utilisés par les systèmes d'apprentissage, c'est-à-dire la déduction et l'induction. Comme l'induction peut produire des incohérences nous verrons qu'il est indispensable de définir un processus de vérification des résultats basé sur une modélisation probabiliste. Ensuite nous présenterons le langage de généralisation et la relation d'appariement et nous montrerons que c'est par leur intermédiaire qu'une partie importante des connaissances du domaine est prise en compte. Enfin, nous exposerons les deux grandes approches possibles pour déterminer le critère de consistance.

Déduction et induction

Reprenons la célèbre phrase d'Aristote «Socrate est un homme, les hommes sont mortels, donc Socrate est mortel». Elle est une illustration ***du raisonnement déductif*** classique, fondé sur la logique dite aristotélicienne. Elle exprime la transitivité de la relation d'implication.

On peut paraphraser Aristote afin d'illustrer le raisonnement inductif[2], «Socrate est mortel, Socrate est un philosophe, donc les philosophes sont mortels». Remarquons qu'une conclusion tout aussi possible est «tous les mortels sont des philosophes». Ce qui montre le côté arbitraire de l'induction. ***L'induction*** est ***le procédé qui consiste à passer d'un ensemble de faits à une loi reliant ces faits***.

Ces deux types de raisonnement sont utilisés en apprentissage symbolique. La déduction permet de classer de nouveaux exemples à partir de règles obtenues par un raisonnement inductif.

2. Joël Quinqueton m'a présenté le premier cette paraphrase, je le lui attribue donc.

Il est important de constater que, si la déduction produit des énoncés valides ce n'est pas le cas de l'induction. Or, un aspect fondamental de l'apprentissage est le contrôle de la validité. Cette dernière est en particulier violée par la présence de faits ou de règles contradictoires :

- faits contradictoires : $P(a)$, $\neg P(a)$ simultanément présents;
- règles contradictoires : $P(x) \Rightarrow Q(x)$, $P(x) \Rightarrow \neg Q(x)$ simultanément présents.

Par rapport à ce problème les deux mécanismes de base précédemment présentés n'ont pas le même comportement. Si la déduction ne crée pas de contradiction, l'induction peut en créer.

Par déduction, en effet la production de faits contradictoires n'est possible que s'il y a, au départ, des faits contradictoires ou des règles contradictoires.

Par induction, on peut créer des règles contradictoires très facilement, sans que le problème le soit au départ :

- faits initiaux : $P(a)$, $Q(a)$, $P(b)$, $\neg Q(b)$
- règles contradictoires inutiles : $P(x) \Rightarrow Q(x)$, $P(x) \Rightarrow \neg Q(x)$

Aussi, pour contrôler cette validité, nous proposons d'introduire des critères statistiques.

Validation des résultats

Un système d'apprentissage doit être capable de bien reclasser les exemples de l'ensemble d'apprentissage. Mais ce simple principe ne suffit pas, car l'objectif est avant tout d'avoir de bonnes performances sur de nouveaux exemples. Pour estimer celles-ci, on peut utiliser deux méthodes :

1. la première consiste à employer un ensemble test indépendant de l'ensemble d'apprentissage. Puis à observer les performances du système sur cet ensemble. Des résultats très bons sur l'ensemble d'apprentissage n'entraînent pas nécessairement des résultats très bons sur le test.

2. la seconde est la méthode dite du «*jack-knife*». En pratique elle consiste à retirer un exemple de l'ensemble d'apprentissage, puis à tester le système avec cet exemple et à répéter, ensuite, ce processus pour tous les exemples.

L'une et l'autre de ces procédures trouvent leur justification dans un cadre probabiliste. On peut montrer qu'elles permettent d'obtenir des estimateurs sans biais des performances réelles de la procédure d'apprentissage. À l'inverse, les performances obtenues sur l'ensemble d'apprentissage lui-même sont souvent très optimistes et fortement biaisées. Aussi on admet généralement que le système ne reclasse pas correctement tous les exemples de l'ensemble d'apprentissage, des résultats parfaits n'étant pas nécessairement le signe de bonnes performances réelles.

Le langage de représentation et la prise en compte d'une connaissance initiale; la relation d'appariement

Le rôle de cette relation booléenne $\mathcal{A}$ est de mettre en correspondance exemples et généralisations, de spécifier quelles descriptions sont les généralisations de tels ou tels exemples. Soient e un exemple et g une généralisation, $\mathcal{A}(e,g)$ est vrai si g est une généralisation de e, et faux sinon. Si $\mathcal{A}(e,g)$ est vrai on dit que e ***est reconnu par*** g.

Cette relation induit un ordre partiel sur le langage de généralisations, lié à la notion de généralité. Cette relation d'ordre induite (notée $\geq$, plus spécifique), est définie sur l'ensemble des couples de généralisations (s,g). Si g est plus générale que s, on dit aussi que s est plus spécifique que g, alors g couvre plus d'exemples que s. Soit de manière formelle, avec E représentant l'ensemble des exemples :

$$s \geq g \Leftrightarrow \{e \in E | \mathcal{A}(e,s)\} \subseteq \{e \in E | \mathcal{A}(e,g)\}$$

La figure 3.4 donne un exemple d'espace de généralisation pour le problème des digits. Dans cette figure une étoile représente aussi bien un 1 qu'un 0.

T. Evans [Evans, 1968] a mis en évidence que la représentation des connaissances est un problème fondamental dans le domaine de l'apprentissage automatique. On ne peut apprendre que ce que l'on sait représenter. Dans le même ordre d'idée, la relation d'appariement dépend de la théorie du domaine d'application. Afin de généraliser, le système doit utiliser des connaissances générales sur le domaine. Reprenons

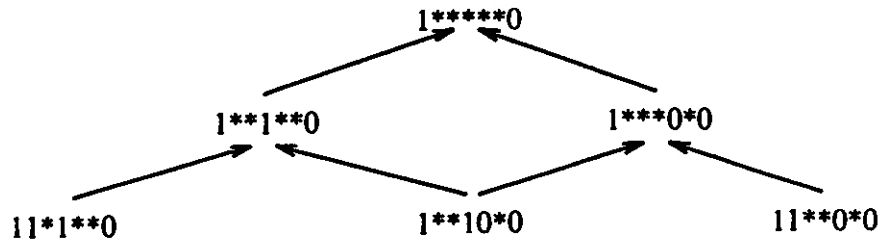


FIG. 3.4 - *Extrait d'un espace de généralisation-spécialisation*

1	0	0	1	1	0	1
1	1	0	0	1	1	0
1	*	0	*	1	*	*

FIG. 3.5 - *Généralisation de deux exemples*

l'exemple de l'équilibre: si le système sait qu'une sphère ou une pyramide sont des bases instables pour monter un équilibre, alors il pourra plus facilement apprendre qu'un empilement doit être constitué d'objets à base stable (sauf pour le plus haut), pour pouvoir être en équilibre. Très souvent, ces connaissances sont représentées sous la forme de hiérarchies de généralisation. Ces hiérarchies font partie de ce que l'on nomme la théorie initiale, et elles constituent un outil très largement utilisé en apprentissage conceptuel.

Pour illustrer cette idée, prenons deux exemples de digits: 1001101 et 1100110 et recherchons une généralisation possible de ces exemples. Si l'on sait qu'une étoile peut remplacer un 1 ou un 0 alors nous pourrons obtenir la généralisation proposée en figure 3.5. Cette connaissance est extrêmement simple. Pour le problème de la détermination des liaisons stratégiques en synthèse organique, nous introduirons certaines connaissances sous la forme de hiérarchie. La figure 3.6 donne un exemple de hiérarchie et met en évidence son influence sur le mécanisme généralisation.

Règles et sélection des règles

À partir de la notion de généralisation et des classes de l'ensemble d'apprentissage nous allons définir la notion de règle.

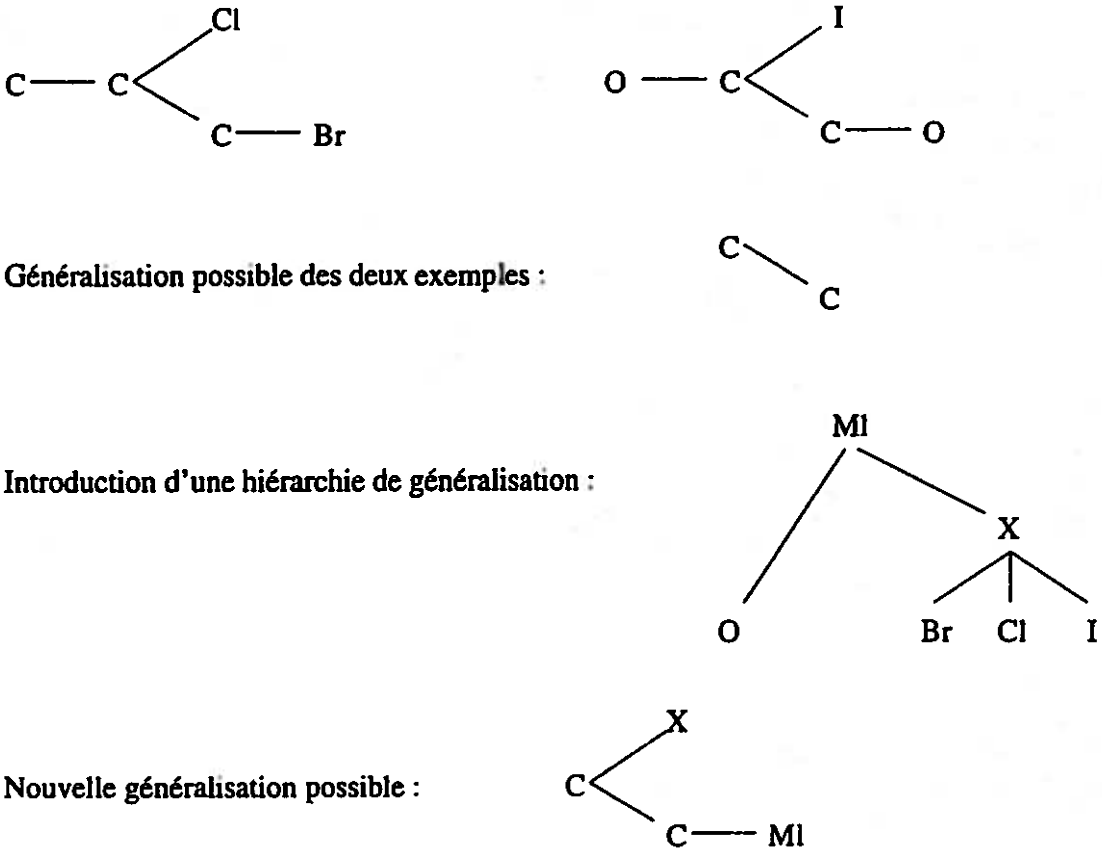


FIG. 3.6 - *Intérêt de l'introduction de connaissances du domaine pour le mécanisme de généralisation.*

Une règle correspond à l'association d'une généralisation g avec une classe C. On la note $(g \rightarrow C)$. Le nombre de fois où une règle $(g \rightarrow C)$ est vérifiée est égale au nombre d'exemples e de C tels que g reconnaisse e ($\mathcal{A}(e, g)$ est vrai). Le nombre de fois où une règle $(g \rightarrow C)$ est contredite est égale au nombre d'exemples e n'appartenant pas à C tels que g reconnaisse e ($\mathcal{A}(e, g)$ est vrai).

Pour sélectionner les règles il existe deux types de critères :

- les critères logiques
- les critères statistiques

Nous allons très brièvement rappeler les principes de chacun des ces critères (cf [Liquière, 1990]).

Critères logiques

Au «début» de l'apprentissage conceptuel la plupart des systèmes d'apprentissage utilisaient ce type de critère [Michalski, 1983; Kodratoff and Ganascia, 1986; Mitchell, 1982].

Une règle est considérée comme vraie si elle vérifie les critères de :

- **cohérence** : elle ne doit être vérifiée sur aucun exemple négatif.
- **complétude** : elle doit être vérifiée par tous les exemples positifs.

L'ensemble d'apprentissage ne peut donc pas contenir d'erreurs.

Critères statistiques

Ces critères ont été introduit par Quinlan [Quinlan, 1983] et par Quinqueton et Sallantin [Quinqueton and Sallantin, 1983].

La validité d'une règle est déterminée statistiquement. On peut introduire les notions de :

- **complétude non stricte** : une formule est statistiquement valide si elle est valide pour la plupart des exemples positifs.
- **cohérence non stricte** : une formule est statistiquement cohérente si elle est non vérifiée pour la plupart des exemples négatifs.

Ces critères ont été introduits pour prendre en compte le bruit et/ou l'utilisation de mode de description incomplet.

3.2.2 Caractérisation des méthodes d'apprentissage

Les méthodes d'apprentissage peuvent être caractérisée selon deux points de vue :

1. la stratégie de construction des généralisations;
2. la manière dont les exemples sont considérés.

Dans la pratique l'ensemble de toutes les généralisations n'est jamais donné in extenso, pour des raisons pratiques d'encombrement. Il est construit au fur et à mesure des besoins.

Il peut être engendré à partir des énoncés les plus généraux en les particularisant : cette stratégie est dite ***descendante***.

Mais il peut aussi être construit à partir des énoncés les plus particuliers, qui sont en général les exemples, en les généralisant : cette stratégie est dite ***ascendante***.

Une stratégie descendante permet de trouver des règles très générales et favorise l'emploi de critère statistiques. À l'inverse une stratégie ascendante favorise les règles spécifiques et rend plus difficile l'emploi de critères numériques. Il n'est pas possible pour un problème donné de choisir a priori entre une règle courte et une longue.

On distingue aussi les méthodes ***incrémentales*** des méthodes ***globales***. Les premières considèrent les exemples les uns après les autres et fonctionnent sans arrêt en mise à jour de la connaissance à partir du nouvel exemple présenté. À l'opposé, les secondes traitent les exemples dans leur ensemble.

3.3 Méthodes de voisinage

Comme pour l'apprentissage conceptuel, les méthodes de voisinage sont basées sur les exemples. Elles utilisent, comme leur nom l'indique, la notion de voisinage, qui elle-même s'appuie sur la notion de distance.

Le voisinage $V(\nu)$ de ν correspond à un sous-ensemble de E contenant des éléments «proches» de ν.

Nous appellerons classe majoritaire d'un ensemble d'exemple la classe qui a le plus de représentant dans cet ensemble. Le principe de ces méthodes est très simple puisqu'il peut se résumer comme suit :

La classe d'un nouvel exemple ν est la classe majoritaire de son voisinage.

Algorithme 38 Algorithme générique des méthodes de voisinage

```
METHODEDEVOISINAGE(E, ν, k) : classe
    noteC+ ← 0
    noteC− ← 0
    V(ν) ← CALCULERVOISINAGE(E, ν, k)
    pour chaque e ∈ V(ν) faire
        si e ∈ C+ alors noteC+ ← noteC+ + 1
        sinon noteC− ← noteC− + 1
    si noteC+ > noteC− alors retourner C+
    si noteC− > noteC+ alors retourner C−
    retourner indécision
```

L'algorithme générique pour ce type de méthode est l'algorithme 38. La fonction de calcul du voisinage d'un exemple est l'objet de la section suivante.

3.3.1 Voisinage

Supposons que l'on dispose d'une distance d, permettant d'exprimer la proximité entre ν et n'importe quel élément de E. Le voisinage $V(\nu)$ peut se définir à l'aide d'un paramètre k et de d selon deux approches différentes :

- Soit il est constitué des éléments e de E tels que $d(e, \nu) \leq k$. Cette idée est à la base de la méthode des ***«noyaux de Parzen»*** [Parzen, 1962].

- Soit il est constitué par les k éléments de E les plus proches de ν. C'est à dire qu'un élément e de E appartient à $V(\nu)$ s'il n'existe pas au moins k éléments de E ayant une distance avec ν inférieure à $d(e, \nu)$. Il s'agit de la ***méthode des k-plus proches voisins*** [Cover and Hart, 1967].

3.3.2 Distance

Le calcul du voisinage fait explicitement appel à une distance. Si les éléments sont représentés numériquement, il est aisé de calculer la distance séparant deux éléments. En effet l'espace de description est $\mathbb{R}^n$, où n le nombre de champs décrivant un objet, et l'on peut, par exemple, appliquer la distance euclidienne. D'autres approches utilisent la distance de Mahanalobis et peuvent aussi pondérer l'importance relative des différentes composantes de la description. Dans le cas où les descriptions sont du type $\{0,1\}^n$, la distance la plus couramment appliquée est la distance de Hamming, c'est à dire le nombre de valeurs différentes dans la description des deux objets. Ainsi, la distance séparant les deux digits 1100100 et 1001100 sera 2. On peut cependant montrer que cette distance «naturelle» n'a pas toujours de bonnes propriétés. Il se peut, par exemple, que certains éléments de descriptions soient non-discriminants (par exemple les 4 bits aléatoires dans notre problème des digits) et influent inutilement sur le calcul de la distance. Certains auteurs [Cost and Salzberg, 1993] ont proposé diverses manières de remédier à ce problème. Enfin, lorsque les objets sont de nature symbolique et ont une structure complexe, il n'existe effectivement pas de distance naturelle. Par exemple rien ne ressemble plus à une phrase que sa négation. De même, en chimie, un éther ressemble fortement à un ester, alors que ce sont deux composés très différents (cf figure 5.10).

3.3.3 Amélioration par pondération

L'inconvénient majeur des méthodes précédentes (noyaux de Parzen et k plus proches voisins) est qu'elles ne tiennent compte ni de l'importance relative des attributs composants un élément, ni du fait qu'un exemple peut être extrêmement représentatif de la classe à laquelle il appartient. Afin de combler cette lacune, et ainsi différencier les exemples en fonction de leur importance au sein de l'ensemble d'apprentissage, on applique une pondération qui peut porter soit sur les objets, soit sur leurs attributs.

Pondération des exemples

Cette technique a été proposée par Cost et Salzberg [Cost and Salzberg, 1993]. Elle consiste à ***associer un poids p_i à chaque élément*** e_i en fonction de son importance dans l'ensemble d'apprentissage. Plus la valeur du poids sera faible, et plus l'exemple sera important. Donc plus la valeur du poids sera élevée et plus l'exemple s'apparentera à une exception. La distance pondérée *dpe* entre ν, l'élément à classer et un élément e_i devient : $dpe(\nu, e_i) = p_i \times d(\nu, e_i)$.

Pour déterminer le degré d'exception d'un exemple e, Cost et Salzberg proposent de considérer le voisinage de e. Si e est une exception, son voisinage contient beaucoup d'exemples qui ne sont pas dans sa classe, et inversement.

Pondération des attributs

Cette méthode consiste à ***associer un poids p_i à chaque attribut*** a_i. Afin de déterminer le poids p_i de chaque attribut, on étudie le pouvoir discriminant de a_i sur l'ensemble d'apprentissage. Plus la valeur d'un attribut aura d'importance dans le choix de la classe d'un élément de cet ensemble, et plus le poids de cet attribut sera élevé. Par exemple, on peut utiliser la mesure du Chi-2 pour déterminer ce pouvoir de discrimination.

La distance pondérée *dpa* entre deux exemples e_1 et e_2 sera alors

$$dpa(e_1, e_2) = \sum_{a_i} p_i \times mismatch(a_i, e_1, e_2)$$

avec $mismatch(a_i, e_1, e_2) = 1$ si la valeur de l'attribut a_i est différente pour e_1 et e_2, et 0 sinon.

3.3.4 Conclusion

Les méthodes de voisinage classent très bien, et souvent plus rapidement que les méthodes d'apprentissage conceptuel, un élément à partir d'un ensemble d'apprentissage. Elles vont donc nous servir de référence. De plus, la pondération permet de mettre en évidence des exceptions et d'affiner encore la classification. Cependant, il n'est pas possible d'obtenir une explication compréhensible du classement d'un élément, ce qui est à notre avis, un inconvénient très important. En outre, l'utilisation

d'une distance fait qu'il est très difficile d'appliquer les méthodes de voisinage aux descriptions structurelles. Mais cette distance est-elle vraiment nécessaire, puisque ce qui nous importe c'est la détermination d'un bon voisinage? La méthode ANNA que nous allons maintenant présenter propose une autre approche de détermination du voisinage, qui n'est pas basée sur une distance.

3.4 ANNA : une méthode symbolique-numérique

ANNA [Gascuel, 1985b; Gascuel, 1985a] est une méthode d'extraction de concepts à partir d'exemples, autorisant un certain niveau de contradiction dans l'induction, et faisant intervenir des considérations fréquencielles dans la déduction. En ce qui concerne l'induction, une règle est considérée comme étant acceptable si elle est suffisamment vérifiée. Pour la déduction, une décision peut être prise même si des règles contradictoires s'appliquent. S'il existe une «majorité suffisante» de règles conduisant à la même décision, alors cette décision est prise.

Selon un point de vue similaire à celui des méthodes de voisinage, on peut également dire que ANNA essaie de déterminer le voisinage d'un élément sans utiliser explicitement une distance entre éléments.

Cette méthode met à la disposition de l'utilisateur deux mécanismes :

1. un mécanisme de décision permettant de classer un nouvel exemple;

2. un mécanisme d'apprentissage proprement dit.

Ce sont les buts principaux de ces mécanismes qui les distinguent. Celui du mécanisme de décision est de classer un nouvel exemple en le comparant avec les exemples de l'ensemble d'apprentissage et d'expliquer ce classement à l'aide de règles obtenues lors des comparaisons. Ce mécanisme s'apparente par certains aspects aux méthodes de voisinage. Le mécanisme d'apprentissage fonctionne différemment, puisque son but est d'extraire un ensemble de règles de l'ensemble d'apprentissage et par la suite de classer grâce à elles les nouveaux exemples.

Ces deux mécanismes font appel au même type de raisonnement basé sur l'analogie ou la similitude entre deux exemples. C'est pourquoi nous allons présenter en premier la recherche d'analogies entre éléments. Ensuite, nous montrerons les liens entre

ANNA et les méthodes de voisinage. Puis, nous détaillerons les processus de décision et d'apprentissage. Après, nous étudierons la fonction utilisée dans ANNA (fonction MATCH) qui permet l'introduction de connaissances du domaine. Finalement, nous donnerons certains résultats et nous conclurons.

3.4.1 Raisonnement «par analogie»

L'idée est de calculer toutes les ressemblances entre un exemple ν et tous les exemples de E afin de construire une liste R de règles de décision, puis de déterminer si ν doit être considéré comme un élément positif ou négatif.

Les ressemblances entre deux éléments sont déterminées par la fonction MATCH.

La fonction MATCH

Une des idées dominantes en apprentissage est que pour apprendre il faut déjà connaître beaucoup. Dans ANNA connaître c'est, comme dans tout programme, avoir un bon langage de description et c'est, de plus, disposer d'une fonction MATCH qui permette de comparer deux exemples. Cette fonction reçoit en entrée deux exemples et donne en sortie un certain nombre de ressemblances. ***Une ressemblance entre*** e_1 ***et*** e_2 ***est une généralisation commune à*** e_1 ***et*** e_2. Le plus souvent, on définit MATCH comme l'ensemble des généralisations maximalement spécifiques de e_1 et e_2. Une généralisation g_1 de e_1 et e_2 est maximalement spécifique s'il n'existe pas de généralisation g_2 de e_1 et e_2 telle que $g_2 > g_1$.

Cette fonction inclut une connaissance très importante du domaine d'application. Déterminer si cette connaissance est ou non efficace pour le problème posé est le propos d'ANNA.

Dans les problèmes réels, il y a beaucoup de connaissances à donner et les exemples sont nombreux et complexes. L'aspect implémentation devient alors très important si l'on souhaite que les temps d'exécution restent raisonnables. Aussi, pour de nombreuses applications, on utilise une fonction spécifique du domaine traité, et optimisée en fonction de ce domaine. Comme nous le verrons dans le chapitre 5, c'est ce type d'approche que nous retenue pour la chimie.

Toutefois, pour être compatible avec ANNA, la fonction MATCH et la relation

d'appariement $\mathcal{A}$ doivent vérifier deux propriétés :

- $\text{MATCH}(e_1, e_2) = \{r_i\} \Rightarrow \mathcal{A}(e_1, r_i)$ est vrai et $\mathcal{A}(e_2, r_i)$ est vrai.

- $\mathcal{A}(e, r)$ est vrai $\Leftrightarrow r \in \text{MATCH}(e, r)$[3].
  En particulier on a $e \in \text{MATCH}(e, e)$.

La fonction MATCH et la relation d'appariement $\mathcal{A}$ sont les seules procédures dépendantes du domaine dans ANNA.

Construction de la Liste R

La liste R des règles de décision s'obtient par l'application de 4 étapes successives :

1. ***Production des arguments élémentaires*** : on applique itérativement la fonction MATCH à ν et à tous les exemples : on obtient un ensemble de ressemblances. À chaque ressemblance r on associe deux règles $(r \rightarrow C_+)$ et $(r \rightarrow C_-)$, une pour chaque classe, que l'on place dans R en prenant soin de ne conserver qu'une seule occurrence de chaque règle.

2. ***Calcul des règles*** : on compare toutes les ressemblances des règles avec tous les exemples afin de déterminer le nombre de fois où les règles sont vérifiées (le nombre d'exemples recouverts par la ressemblance de la règle $(r \rightarrow C)$ et appartenant à C) et contredits (le nombre d'exemples recouverts par la ressemblance de la règle $(r \rightarrow C)$ et n'appartenant pas à C).

3. ***Suppression des contradictions*** : on élimine de R les règles inacceptables au sens de critères définis ci-dessous.

4. ***Contraction des arguments*** : on supprime dans R les redondances, c'est-à-dire toutes les règles $(r_1 \rightarrow C)$ de R telles qu'il existe $(r_2 \rightarrow C) \in R$ avec $r_1 \geq r_2$. On ne conserve donc que les règles dont les ressemblances sont les plus spécifiques.

3. Si MATCH est l'ensemble des généralisations maximalement spécifiques, alors on a dans ce cas $\text{MATCH}(e, r) = \{r\}$, et $\text{MATCH}(e, e) = \{e\}$.

Acceptabilité des règles

D'après le point 3, il est nécessaire de définir un critère d'acceptabilité. Il s'agit d'un problème extrêmement important et difficile à résoudre.

Comme nous l'avons dit précédemment ANNA autorise des contradictions dans le mécanisme inductif. Par rapport à ce principe, nous pouvons donc considérer qu'une règle est acceptable si elle remplit deux conditions. Soient n_+ le nombre d'exemples de C recouvert par la ressemblance r de la règle $(r \rightarrow C)$, et n_- le nombre d'exemple n'appartenant pas à C recouvert par r. On impose:

$n_+ >$ LMV (Level of Minimal Verification),

$n_+/n_- >$ LACI (Level of Acceptable Contradiction on Induction),

où LMV et LACI sont deux seuils définis par l'utilisateur. Le premier critère impose que la règle se déclenche un nombre suffisant de fois. Le second impose qu'elle soit «en partie» valide sur l'ensemble d'apprentissage. Typiquement, on prendra LMV=10 exemples et LACI=5 ou 10.

3.4.2 Décision

C'est le mécanisme déductif de la méthode. L'introduction de considérations sur la fréquence dans la déduction se fait par l'intermédiaire du paramètre LACD (Level of Acceptable Contradiction on Deduction).

La décision est prise par ***un vote des règles*** de R. Le vote de chaque règle est pondéré par le nombre de fois où la règle est vérifiée. Le résultat est la classe C_i si une majorité supérieure à LACD se dégage en faveur de C_i. Dans le cas contraire, le programme s'abstient.

3.4.3 Apprentissage

Pour apprendre, ANNA applique le processus de décision décrit ci-dessus à tous les exemples e de E, en considérant que l'ensemble d'apprentissage est $E-\{e\}$. ANNA apprend dans deux directions:

1. L'usager définit un critère qui permet d'évaluer les performances, par exemple le nombre d'exemples mal classés. Par une méthode d'amélioration pas à pas,

ANNA trouve un triplet de valeurs optimales pour (LACI,LMV,LACD) pour ce critère. Les temps d'exécution interdisent, le plus souvent, cette approche.

2. Ces trois paramètres étant déterminés, ANNA mémorise les règles produites par le processus de décision. Lorsqu'un nouvel exemple ν se présente, au lieu d'utiliser le processus de décision, ANNA utilise directement les règles apprises. Le résultat est similaire, mais les temps d'exécution sont considérablement réduits.

3.4.4 Ressemblances et voisinages

Cette section essaie d'établir un lien entre le raisonnement analogique d'ANNA et le raisonnement employé par les méthodes de voisinage.

Intéressons nous tout d'abord aux ressemblances et plus particulièrement à celles que l'on obtient pour le problème des Digits. Nous savons que $r = 1*0*1**$ est une ressemblance entre $e_1 = 1001101$ et $e_2 = 1100110$. Une telle généralisation permet de calculer la distance de Hamming entre les deux exemples. Considérons la méthode des noyaux de Parzen. On a $V(\nu) = \{e \in E$ tel que $d(\nu, e) \leq k\}$. Soient h la distance de Hamming entre e_1 et e_2 (ici, elle vaut 4) et E_r l'ensemble des éléments de E plus spécifiques que r. Posons $k = h$, alors on a immédiatement $E_r \subseteq V(e_1)$ et $E_r \subseteq V(e_2)$. Une ressemblance entre ν et un exemple e permet donc d'obtenir ***un sous-ensemble du voisinage de ν sans avoir besoin d'utiliser explicitement une distance entre e et ν***. De plus, si une règle associée à cette ressemblance est peu contredite ce sous-ensemble est pur ou presque pur, il ne contient presque que des éléments appartenant à la même classe.

La figure 3.7 va nous permettre de donner une idée intuitive du sous-ensemble du voisinage que calcule la méthode ANNA. Suivant la direction dans laquelle on «regarde» on sera plus ou moins exigeant sur le voisinage, l'objectif étant de voir si dans cette direction on peut trouver une zone pure. Cette méthode s'apparente donc à la pondération des attributs, mais cette pondération est réalisée, par ANNA, de manière locale (autour du point à classer) et en tenant compte des interactions entre attributs.

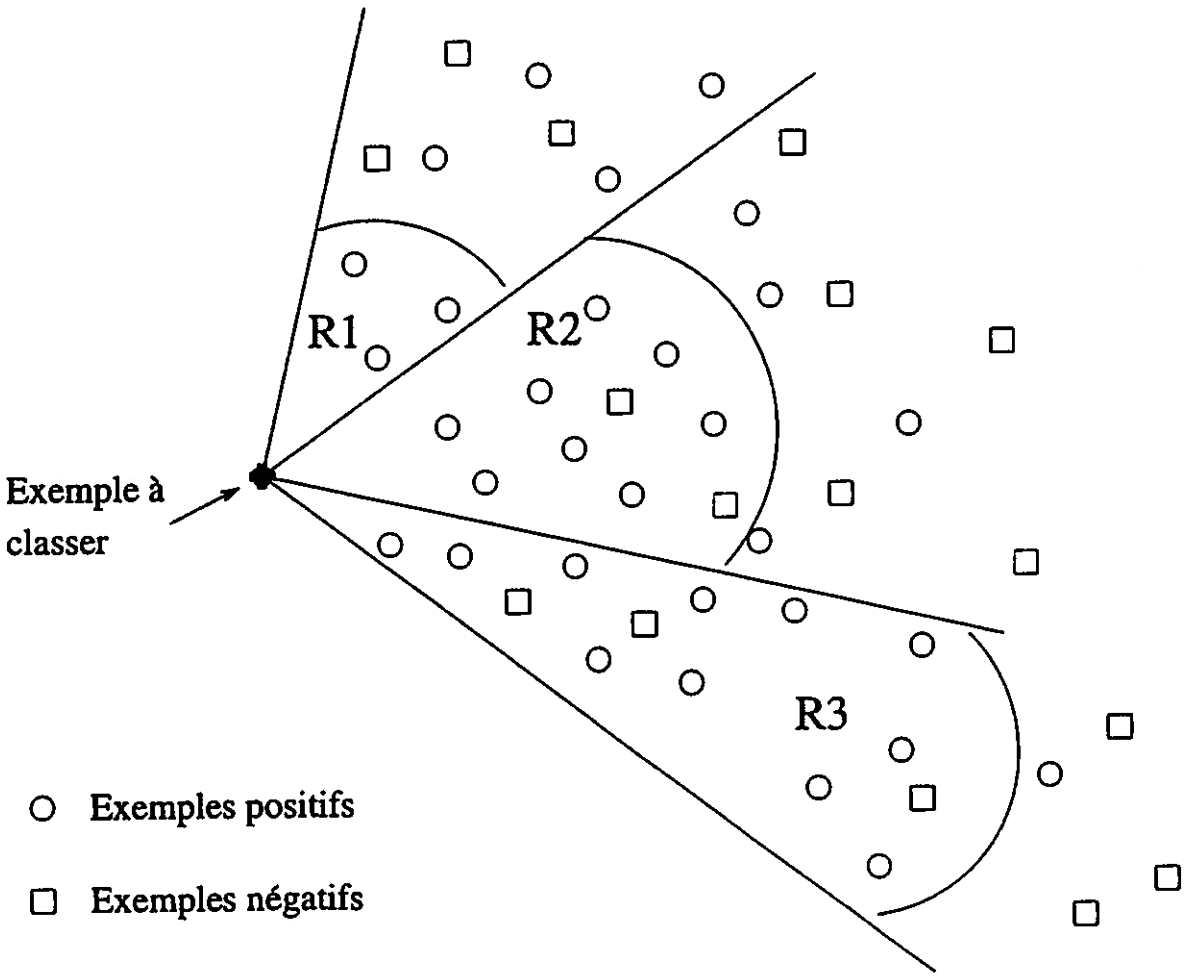

R1
R2
R3
Exemple à classer
Exemples positifs
Exemples négatifs

FIG. 3.7 -

3.4.5 Résultats

ANNA a été appliqué à plusieurs domaines :

1. Prédiction de comportement d'une souris étant donné ses comportements antérieurs ([Gascuel, 1985b]);
2. Reconnaissance de la classe sémantique de mots médicaux français en fonction de leur morphologie ([Gascuel, 1985b]);
3. Reconnaissance de la classe fonctionnelle d'ARNt (acide ribonucléique de transfert) ([Gascuel, 1985a]);
4. Apprentissage des liaisons stratégiques en synthèse organique [Régin, 1995].

Dans chacun des trois premiers domaines, ANNA a obtenu de bonnes performances. De plus les explications données par le programme, ainsi que les connaissances apprises, ont un sens et une pertinence réels. En ce qui concerne le dernier point, le système classait correctement mais n'a pas fourni d'explications intéressantes.

3.4.6 Conclusion

Nous pensons que les hypothèses de base de ANNA sont à la fois générales et naturelles. Quelque soit le mode de description structurelle, lorsqu'on se situe dans un environnement qui comporte une part d'incertitude, il est important d'admettre la contradiction et de mettre en avant des considérations fréquencielles.

L'efficacité de ANNA provient de la fonction MATCH. Pour que le programme fonctionne, il faut qu'elle donne des ressemblances pertinentes en comparant une nouvelle description avec un seul exemple, et non un ensemble d'exemples. Si dans un domaine une telle fonction existe, ANNA est parfaitement adapté, et il n'y a pas de raison d'utiliser des méthodes combinatoires.

ANNA a donné de bons résultats, cependant elle rencontre des problèmes dans certaines conditions:

- Lorsque les classes sont déséquilibrées, les seuils LACI,LMV et LACD peuvent apporter de mauvaises explications.

EXEMPLE:

$|C_+| = 20\ |C_-| = 100\ LACI = 3\ LMV = 1$

Règle	$\|e \in C_+ \ / \ e \geq r_i\|$	$\|e \in C_- \ / \ e \geq r_i\|$	Acceptée/Refusée
r_1	20	7	Refusée par LACI
r_2	15	6	Refusée par LACI
r_3	16	6	Refusée par LACI
r_4	3	1	Acceptée
r_5	1	4	Acceptée
r_6	0	1	Acceptée

On constate que les règles r_4, r_5 et r_6 sont acceptées alors qu'elles expliquent très peu d'exemples. Par contre la règle r_1 qui explique 100% des exemples de la classe 1 et réfute 93% des exemples de la classe 2 est supprimée.

- Les règles apprises sont systématiquement les plus générales, ce qui peut apporter un manque de précision dans les explications.
  EXEMPLE:

Règle	$\|e \in C_+ \ / \ e \geq r_i\|$	$\|e \in C_- \ / \ e \geq r_i\|$
r_1	50	15
r_2	45	10
r_3	45	1
r_4	12	0

  En posant que r_i est plus générale que r_j si $i < j$, ANNA va choisir r_1 alors que la règle la plus pertinente est r_3 (avec LACI=3 et LMV=1);

- ANNA fonctionne mal dans le cas où chaque classe n'est pas caractérisable.

- Ce sont les règles qui votent, pas les exemples. Un exemple reconnu par plusieurs ressemblances associées aux règles peut ainsi s'exprimer plusieurs fois et fausser la décision. Cela est possible car la relation d'appariement n'est pas un ordre total, aussi deux ressemblances peuvent être plus générales qu'un exemple sans qu'elles soient nécessairement liées par la relation $\geq$. De plus, faire voter les règles est en contradiction avec les principes des méthodes de voisinage.

3.5 Conclusion

Dans ce chapitre nous avons donné les principes de l'apprentissage conceptuel et présenté les méthodes de voisinage et une méthode d'apprentissage symbolique par ressemblance. Nous avons montré que le principe des méthodes de voisinage et de ANNA était commun : déterminer le bon voisinage d'un élément à classer. ANNA dont le propos était de traiter des données symboliques tout en étant simple, rapide et efficace comme les méthodes de voisinage et capable d'expliquer les raisons du classement d'un nouvel élément comme les méthodes d'apprentissage conceptuel, semble avoir atteint son but puisque ANNA a donné de bons résultats en pratique. Pour des problèmes simples comme le problème des Digits, il est préférable d'utiliser ANNA plutôt qu'une méthode de voisinage. Toutefois cette méthode présente de nombreux défauts quand les problèmes deviennent plus complexes (autrement dit ne sont plus des exemples «jouets») comme celui auquel nous sommes confrontés. C'est pourquoi nous avons développé une nouvelle méthode. Elle est présentée dans le chapitre suivant.

Chapitre 4

CNN : une méthode conceptuelle de plus proches voisins

Dans ce chapitre nous présentons la méthode CNN (Conceptual Nearest Neighbour). L'une des différences les plus importantes avec ANNA est que cette nouvelle méthode a pour objectif de faire voter les exemples comme dans les méthodes de voisinage. De plus, la phase d'acceptabilité des ressemblances est plus sophistiquée que celle de ANNA, pour qu'elle fournisse des explications plus pertinentes. En outre, CNN n'impose pas de connaissances préalables sur les règles à apprendre, à l'inverse de ANNA qui privilégie les règles les plus spécifiques.

Certains autres principes ne sont pas remis en cause. Deux mécanismes sont toujours mis à la disposition de l'utilisateur, à savoir une mécanisme de décision et un mécanisme d'apprentissage. Ce dernier est, cependant un peu plus évolué, et il est possible de le rendre incrémental.

CNN a la capacité de s'adapter à des problèmes utilisant des systèmes de représentation complexes. Elle accepte assez bien le bruit, autrement dit des exemples de l'ensemble d'apprentissage mal classés ou mal décrits, les classes déséquilibrées, c'est-à-dire qu'une classe peut contenir beaucoup plus d'exemples qu'une autre.

La présentation qui va suivre est plus précise que celle que nous avons précédemment faite. En effet, nous essaierons de donner systématiquement des algorithmes. Toutefois ces derniers ne sont proposés qu'à titre d'exemple et pour permettre une implémentation facile de notre méthode. Si l'on veut que celle-ci soit plus perfor-

mante en temps, il faut modifier certains algorithmes afin qu'ils tiennent compte des spécificités du domaine d'application comme la complexité de la fonction MATCH, de la relation d'appariement ou encore de la relation $\geq$.

Nous commencerons par présenter la recherche des ressemblances symboliques, nous introduirons alors deux nouvelles méthodes de suppression des règles non pertinentes. Puis nous présenterons le processus de décision et une amélioration du mécanisme d'apprentissage précédemment exposé. Ensuite nous étudierons sur des problèmes tests les performances de notre système en les comparant, entre autres, à celles des méthodes de voisinage. Enfin nous donnerons plusieurs perspectives et nous conclurons.

Avant tout, nous rappelons quelques convention d'écriture:

- E est l'ensemble d'aprentissage;
- C_+ et C_- sont respectivement les classes des exemples positifs et négatifs;
- ν est un nouvel exemple à classer;
- $(r \rightarrow C)$ est une règle associant la ressemblance r avec la classe C;
- $s \geq g$ signifie que la description s est plus spécifique que g;
- $\mathcal{A}(e, r)$ est vrai si l'exemple e est plus spécifique que la ressemblance r et faux sinon.

4.1 Recherche de ressemblances symboliques

La recherche de ressemblances s'apparente au raisonnement par analogie de ANNA. Nous distinguerons, cette fois, 6 phases:

1. création de la base de ressemblances;
2. pondération des ressemblances;
3. création des règles;
4. évaluation des règles;
5. organisation des règles;
6. détermination des règles prépondérantes.

Dans la suite nous allons détailler chacun des points précédents. Cependant nous attacherons une importance particulière aux points 4 et 6 car ce sont eux qui conditionnent le plus la qualité des résultats fournis par la méthode.

4.1.1 Création de la base des ressemblances

Comme dans ANNA, il faut générer toutes les ressemblances que l'on peut trouver entre ν et les éléments de l'ensemble d'apprentissage. L'algorithme 39 réalise cela.

Algorithme 39 Création de la base de ressemblances.

CRÉERBASERESSEMBLANCES(i E, ν io S)
  pour *chaque* $e \in E$ **faire** $S \leftarrow S \cup$ MATCH(e, ν)

Lors de la génération des diverses listes de ressemblances, il apparaît de manière systématique des ressemblances qui sont dupliquées. Un seul exemplaire par ressemblance étant suffisant pour travailler, il est nécessaire de supprimer cette redondance afin de gagner du temps et de l'espace mémoire.

Au cours de la suppression de cette redondance, il apparaît forcément un test d'égalité entre deux éléments qui peut être extrêmement coûteux en chimie puisqu'on traite des graphes non nécessairement planaires et que le test d'égalité devient alors un test d'isomorphisme, qui est un problème non classé à l'heure actuelle. C'est pourquoi nous avons choisi d'introduire une fonction de hachage de l'ensemble des ressemblances dans $\mathbb{N}$, basée sur deux critères qui sont la taille de la ressemblance, et un code donné à chaque ressemblance. Grâce à cette fonction de hachage, une table de hachage est alors établie dans laquelle les seuls éléments pouvant être égaux sont ceux appartenant à la même cellule (une cellule contenant tous les éléments ayant la même valeur pour la fonction de hachage). Nous faisons l'hypothèse que chaque ressemblance a la même chance d'être «hachée» dans l'une quelconque des cellules. Cette hypothèse est dite de hachage uniforme simple.

Ainsi, grâce à la fonction de hachage, le test d'égalité de deux ressemblances ne se fera que si la table de hachage indique que les deux ressemblances peuvent être identiques (même taille et même code), d'où un gain de temps appréciable notamment en chimie car on évitera ainsi des tests très coûteux et inutiles.

4.1.2 Pondération des ressemblances

A chaque ressemblance s est associée une note par classe de l'ensemble d'apprentissage qui représente le nombre d'exemples de la classe qui sont reconnus par s. L'algorithme 40 réalise cela. Pour ne pas recommencer plusieurs fois par la suite les mêmes tests, un tableau, noté $reconnait[s, e]$, mémorise les calculs effectués. $\mathcal{A}$ est la relation d'appariement définie dans le chapitre précédent.

Algorithme 40 Pondération de la base de ressemblances.

```
PONDÉRERBASERESSEMBLANCES(i E, S)
  pour chaque s ∈ S faire
    noteC+[s] ← 0
    noteC−[s] ← 0
    pour chaque e ∈ C+ faire
      si A(e, s) alors
        noteC+[s] ← noteC+[s] + 1
        reconnait[s, e] ← vrai
      sinon reconnait[s, e] ← faux
    pour chaque e ∈ C− faire
      si A(e, s) alors
        noteC−[s] ← noteC−[s] + 1
        reconnait[s, e] ← vrai
      sinon reconnait[s, e] ← faux
```

4.1.3 Création des règles

On peut associer à chaque ressemblance s deux règles: $(s \rightarrow C_+)$ et $(s \rightarrow C_-)$. Or s est muni de deux notes, une pour chaque classe. Si l'une de ces deux notes est plus élevée que l'autre il ne sert à rien d'engendrer une règle en faveur de la classe pour laquelle la note est la plus faible. Une ressemblance est donc toujours associée à une règle au plus. Aussi la relation $\geq$ entre les ressemblances induit une relation semblable entre les règles. Nous parlerons donc d'une règle plus générale ou

plus spécifique qu'une autre. Nous utiliserons également le symbole $\geq$ pour les règles. L'algorithme 41 crée la base de règles.

Algorithme 41 Création de la base de règles.

```
CRÉERBASERÈGLES(i S) : ensemble de règles
  R ← ∅
  pour chaque s ∈ S faire
  |  si noteC+[s] > noteC−[s] alors
  |  |  r ← CRÉERRÈGLE(s, C+)
  |  |  AJOUTER(r, R)
  |  si noteC−[s] > noteC+[s] alors
  |  |  r ← CRÉERRÈGLE(s, C−)
  |  |  AJOUTER(r, R)
  retourner R
```

4.1.4 Évaluation de la base de règles

C'est ici que l'on va juger l'importance de chaque règle et ceci en fonction des notes qui lui ont été attribuée. L'évaluation est très importante, car elle va entraîner la suppression de nombreuses règles. Elle réalise ce que nous appellerons le ***premier élagage des règles***.

L'évaluation des règles doit être réalisée afin, d'une part, d'éliminer celles qui ne sont pas acceptables, car pas assez vérifiées (ou trop contredites), et d'autre part, de permettre de juger de la qualité des règles, et surtout de pouvoir les comparer.

Nous supposerons que l'on veut évaluer une règle $(r \rightarrow C_+)$ associant une ressemblance avec la classe des exemples positifs. Un raisonnement analogue peut être fait pour les règles en faveur de la classe des exemples négatifs. Notons n_+ le nombre d'exemples cd C_+ plus spécifiques que r et n_- ceux de C_- plus spécifiques que r. Posons $n = n_+ + n_-$.

La proportion théorique d'exemples positifs peut être estimée par $p = |C_+|/(|C_+| + |C_-|)$.

La proportion d'exemples positifs recouverts par la règle est n_+/n.

«Cette proportion diffère-t-elle significativement de la proportion théorique p?»

L'approche usuelle pour répondre à cette question consiste à utiliser le test de Chi-2. Elle n'est pas praticable, ici, en raison de la faiblesse de l'effectif observé. On peut revenir à un calcul exact basé sur la loi binômiale. Celui ci conduit à rejeter l'hypothèse nulle, c'est-à-dire à considérer que la règle est significative si:

$$\sum_{i=n_+}^{n} C_n^i p^i (1-p)^{n-i} < \alpha$$

où α est le seuil de confiance (généralement $\alpha = 5$ %). Pour le cas particulier de notre problème en chimie il est très difficile d'imaginer que l'estimation de p est satisfaisante. En effet, nous savons quels sont les exemples positifs (les liaisons créées), mais les exemples négatifs sont mal définis, et a fortiori la proportion des liaisons qui pourront être créées paraît bien être une notion difficile à définir. Cette méthode n'est donc guère utilisable pour notre application en chimie. Il faut donc envisager une autre voie.

Considérons la fréquence d'erreur de la règle: $f = n_-/n$. Notre objectif est de déterminer la probabilité d'erreur p_e de cette règle et de choisir les règles ayant une faible proportion d'erreur. L'effectif n étant très réduit et, de plus, la règle étant issue d'un processus de sélection, estimer la proportion d'erreur par la fréquence conduit à la fois à une estimation imprécise et biaisée. Aussi nous allons minimiser une surestimation de la borne supérieure de la probabilité d'erreur, $\overline{p_e}$. Soit α le seuil de confiance (généralement 5 %), alors la borne supérieure $\overline{p_e}$ de p_e est donnée par l'équation (voir [Gascuel and Caraux, 1992] pour une présentation plus complète):

$$\sum_{j=0}^{n_-} C_n^j \overline{p_e}^j (1-\overline{p_e})^{n-j} = \alpha$$

De manière informelle, cela signifie que $\overline{p_e}$ est la valeur du paramètre de la loi binômiale qui est telle que f a la probabilité α d'apparaître. $\overline{p_e}$ est la plus petite surestimation possible de la borne supérieure pour p_e quand le niveau de confiance est supérieur ou égal à $1-\alpha$.

La valeur $\overline{p_e}$ ne peut pas être calculée directement et doit être approximée par une procédure pas-à-pas. Le calcul est relativement simple puisque le membre gauche

de l'égalité est une fonction monotone strictement décroissante de $\overline{p_e}$. $\overline{p_e}$ constitue la valeur associée à une règle. On ne conservera la règle que si cette valeur est inférieure à un seuil donné. De cette façon, on ne conservera que les règles apportant le minimum d'erreur pour une explication la plus significative possible du problème.

On remarquera que cette approche se distingue de la première en ce sens qu'elle ne nécessite pas d'hypothèse sur la proportion théorique des classes. Elle s'avère en pratique très bien adaptée lorsque cette proportion est voisine de 1/2. Dans le cas où elle s'en écarte significativement, on est conduit à introduire des nouveaux paramètres de réglage comme nous allons le voir ci-dessous.

Dans la suite nous noterons $val(r)$ la valeur associée à une règle r.

Introduction d'un seuil par classe

Avec un seuil commun à toutes les classes, nous avons constaté, pour les ensembles d'apprentissage dont le nombre d'exemples présents dans une classe était nettement plus important que dans l'autre, qu'une erreur de classement était commise systématiquement lorsqu'on voulait classer un élément de la classe la moins nombreuse. En fait, le système choisissait toujours la classe ayant le plus de représentants car c'est ainsi qu'il avait le moins de risque de commettre une erreur.

Exemple: $n = 100$, 20 éléments dans la classe C_+ et 80 dans la classe C_-. Supposons que l'on doive évaluer les deux règles suivantes:

$r_1 = (s_1 \rightarrow C_+)$ $noteC_+ : 20$ $noteC_- : 0$
$r_2 = (s_2 \rightarrow C_-)$ $noteC_+ : 0$ $noteC_- : 20$

Les valeurs des deux règles seront identiques (elles reconnaissent le même nombre d'exemples, seule la classe change), et par conséquent on devra soit les garder toutes les deux, soit les supprimer toutes les deux. Il est cependant clair qu'il faut garder r_1 qui explique la totalité des exemples de la classe C_+ et aucun exemple de la classe C_-. Par contre r_2 n'explique que 25% des exemples de la classe C_- et son sort dépendra de la qualité des autres règles trouvées. Bien que les bornes supérieures du taux d'erreur, au sens du critère ci-dessus, soient identiques, leur sort, lui, ne doit pas l'être.

Lors de l'évaluation de la base de règles, on n'associe donc plus un seuil unique a toutes les classes, mais ***un seuil à chaque classe***. Le fait de conserver ou non une règle dépendra ainsi de la classe qu'elle prédit et du seuil associé à cette classe. ***Cette idée permet au système d'apprendre même si une classe est plus facile à prédire qu'une autre***. Par exemple, dans le cas des digits, on se trouve ainsi avec 2 seuils, un pour les règles qui ont tendance à prédire la classe des éléments pairs, l'autre pour celles qui prédisent la classe des impairs, cette dernière étant beaucoup plus facile à apprendre.

L'évaluation de la base de ressemblance se fait donc simplement en associant à chaque règle la borne supérieure de son taux d'erreur, puis en supprimant celles dont cette valeur est supérieur au seuil de la classe qu'elle prédit (cf. algorithme 42). Deux nouveaux paramètres sont introduits : $seuilC_+$ et $seuilC_-$. Ils correspondent respectivement à la valeur maximale de la borne supérieure du taux d'erreur des règles en faveur de la classe C_+ et la classe C_-.

Algorithme 42 Évaluation de la base de règles.

```
ÉVALUERBASERÈGLES(io R,i seuilC+, seuilC−)
  pour chaque r ∈ R faire
    valeur[r] ← CALCULERTAUXERREUR(r)
    si valeur[r] > seuilC+ et C = C+ alors SUPPRIMER(r, R)
    si valeur[r] > seuilC− et C = C− alors SUPPRIMER(r, R)
```

4.1.5 Organisation de la base de règles

La base de règles, après le premier élagage, est organisée selon la relation $\geq$. Nous rappelons que $r_1 = (s_1 \rightarrow C) \geq r_2 = (s_2 \rightarrow C)$ ssi $s_1 \geq s_2$. Pour ne pas répéter certains calculs on détermine la fermeture transitive de cette relation (cf. algorithme 43).

Une nouvelle structure de données est introduite. Il s'agit d'une matrice carrée ayant autant de lignes que la base de règles contient de ressemblances différentes, elle est notée *plusGénéral*. Si la ressemblance de la règle r_i est plus générale que

celle de la règle r_j alors $plusGénéral[i,j] = 1$. Cette matrice n'est évidemment pas symétrique.

Algorithme 43 Organisation de la base de règles.

```
ORGANISERBASERÈGLES(i R)
  pour chaque r_i ∈ R faire
    pour chaque r_j ∈ R tel que i < j faire
      si r_j ≥ r_i alors
        plusGénéral[j,i] = 1
        plusGénéral[i,j] = 0
      si r_i ≥ r_j alors
        plusGénéral[j,i] = 0
        plusGénéral[i,j] = 1
```

4.1.6 Détermination des règles prépondérantes

Nous rappelons, car c'est très important dans cette étape, que notre but est de faire voter les exemples et non plus les règles. Cette phase va procéder à ***un second élagage des règles***. Nous rappelons qu'une règle $r = (s \rightarrow C)$ reconnaît un exemple e si l'on a $\mathcal{A}(e, s)$, où $\mathcal{A}$ est la relation d'appariement définie dans le chapitre précédent. Nous écrirons E_r l'ensemble des exemples reconnus par une règle r.

Avant cette étape, R est constitué des règles qui ont été considérées comme acceptables. R peut être partitionné en deux sous-ensembles. Le premier, R_+, contient les règles en faveur de la classe C_+, tandis que le second, R_-, contient celles en faveur de C_-. Comme R est organisé selon une hiérarchie définie par la relation $\geq$, nous pouvons considérer que nous sommes en présence de deux hiérarchies, l'une pour R_+ et l'autre pour R_-. Nous allons travailler sur chacune de ces hiérarchies. Les principes exprimés pour l'une reste valable pour l'autre.

Considérons une de ces hiérarchies et appelons r^* la (ou les) règle dont la valeur associée, au sens du critère défini dans la section précédente, est la plus faible. Autrement dit telles qu'il n'existe pas de règles r avec $val(r) < val(r^*)$. Il est légitime qu'elle participe au classement de ν.

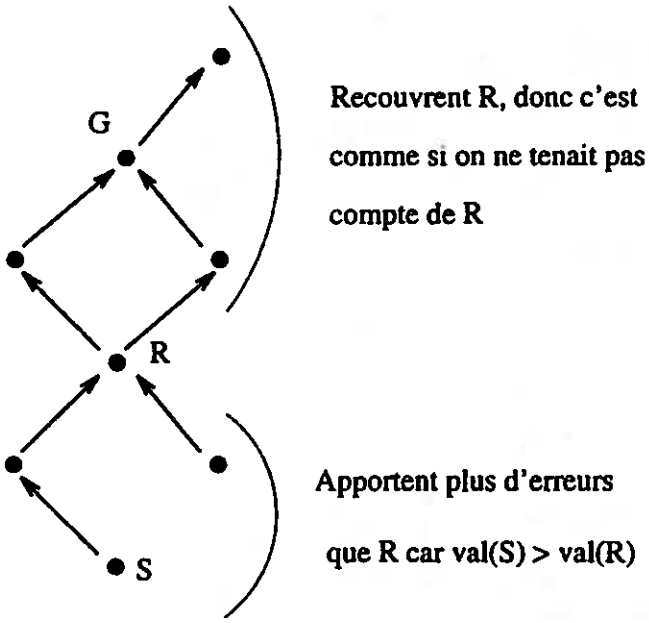


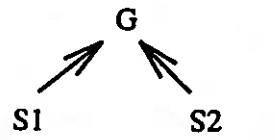
FIG. 4.1 - *Suppressions dans la hiérarchie des règles. Un arc de s vers g indique que la ressembalnce associée à s est plus spécifique que celle associée à g.*

Nous pouvons affirmer que les règles plus spécifiques que r^* sont redondantes. En effet, pour toute règle s plus spécifique que r^* on a: $E_s \subseteq E_{r^*}$. Aussi, ces règles n'apporterons aucun votant supplémentaire. De plus, elles ont des valeurs inférieures à celle de r^*.

Intéressons nous maintenant aux règles plus générales que r^*. Si l'une de ces règles (notée g) participe au classement de ν alors elle amènera plus de votants que r^* puisque $E_{r^*} \subseteq E_g$. r^* devient redondant par rapport à g, ce qui n'est pas logique puisque $val(r^*)$ est inférieure à $val(g)$. Les règles plus générales que r^* ne doivent donc pas participer au classement de ν.

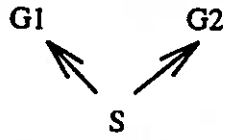
À partir de considérations logiques, nous pouvons proposer une première méthode pour déterminer les règles prépondérantes. Elle consiste à choisir tout d'abord la règle r^* ayant le plus petit taux d'erreur puis ***à supprimer les règles plus générales ou plus spécifiques*** que celle choisie. Ensuite on répète ce processus avec celles qui restent jusqu'à épuisement des règles. La figure 4.1 résume ce mécanisme.

Cependant, ce raisonnement n'est pas suffisant, comme nous allons maintenant le voir. L'inconvénient de ne supprimer que les règles plus spécifiques ou plus générales que r^* est mis en évidence dans l'exemple donné en figure 4.2.



val(S1) < val(G) < val(S2)

FIG. 4.2 - *Problème lié à la suppression des règles plus générales.*



val(G1) < val(S) < val(G2)

FIG. 4.3 - *Problème lié à la suppression des règles plus spécifiques.*

Par rapport à cette hiérarchie, r^* est égal à $S1$, car la valeur associée à $S1$ est la plus petite. D'après le mécanisme que nous venons d'exposer, G qui est une généralisation de $S1$ doit être supprimé. Il ne reste ensuite que $S2$. La suppression de règles après le choix de $S1$ est terminée. La prochaine règle que l'on choisira est $S2$. Après ce choix le processus s'arrête puisqu'il ne reste plus de règles. On aura ainsi comme ressemblances explicatives les règles $S1$ et $S2$ alors que l'on ne devrait avoir que $S1$. En effet, si on devait avoir $S2$, on choisirait plutôt G qui est plus générale que $S2$ et dont la valeur est meilleure, mais $S1$ étant meilleure que G, on ne devrait choisir que $S1$.

Une erreur similaire survient également lorsque deux règles sont plus générales qu'une troisième (cf figure 4.3).

Selon la configuration de la hiérarchie des règles, la suppression uniquement des règles plus spécifiques ou plus générales que r fait donc apparaître des règles qui auraient dû disparaître.

Dans un cas plus général, la hiérarchie des règles peut avoir une allure semblable à celle donnée en figure 4.4.

Les règles retenues sont: $S1$, $S2$ et $S3$. Mais $S2$ étant moins bonne que $G1$ qui elle n'a pas été retenue, $S2$ ne doit pas l'être non plus. Les règles qu'il faut retenir sont donc $S1$ et $S3$.

En fait, on ne doit plus éliminer à chaque étape les règles plus spécifiques ou

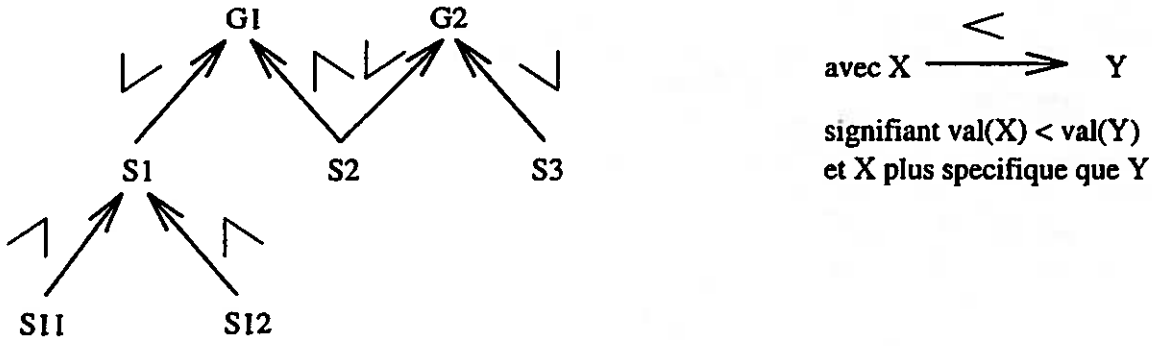


FIG. 4.4 - *Exemple de hiérarchie de règles.*

plus générales qu'une règle choisie. Désormais, ***nous allons éliminer de manière globale toutes les règles qui possèdent un ascendant ou un descendant qui possède une meilleure valeur qu'elle.*** Les règles restantes constitueront notre nouvel ensemble R.

Ce mécanisme doit être implémenté avec précaution. La présence d'un ascendant ou d'un descendant doit être réalisé par rapport à l'ensemble de toutes les règles au moment de l'appel de cette fonction. L'algorithme DÉTERMINERRÈGLESPRÉPONDÉRANTES propose une implémentation possible de ce nouveau mécanisme d'élimination.

4.2 Processus de décision

Il s'agit simplement de faire voter les exemples reconnus par l'ensemble des règles prépondérantes. Cette décision se fera de la façon suivante : on comptabilise le nombre n_c d'exemples de chaque classe c, reconnu par au moins une règle r de R. La classe choisie sera celle dont la valeur n_c est la plus élevée. S'il n'existe pas de telle classe, le système répondra qu'il ne peut pas prendre de décision. À l'inverse de ANNA, ce ne sont donc pas les règles qui votent, mais les exemples reconnus par ces règles. De plus un exemple ne vote qu'une seule fois.

L'algorithme 45 propose une implémentation possible pour le classement d'un nouvel exemple.

Algorithme 44 Détermination des règles prépondérantes.

```
DÉTERMINERRÈGLESPRÉPONDÉRANTES(io R)
  ∀r_i ∈ R : supprime[i] ← faux
  pour chaque r_i = (s_i → C_+) ∈ R faire
  |  pour chaque r_j = (s_j → C_+) ∈ R tant que ¬supprime[i] faire
  |  |  si plusGénéral[i, j] ou plusGénéral[j, i] alors
  |  |  |  si valeur[i] > valeur[j] alors supprime[i] ← vrai
  pour chaque r_i = (s_i → C_−) ∈ R faire
  |  pour chaque r_j = (s_j → C_−) ∈ R tant que ¬supprime[i] faire
  |  |  si plusGénéral[i, j] ou plusGénéral[j, i] alors
  |  |  |  si valeur[i] > valeur[j] alors supprime[i] ← vrai
  pour chaque r_i ∈ R faire
  |  si supprime[i] alors SUPPRIMER(r_i, R)
```

4.3 Apprentissage

Pour apprendre, le système va tenter de reclasser correctement tous les exemples de l'ensemble d'apprentissage. Pour ce faire, il va raisonner non plus sur l'ensemble de règles R_e constitué des règles obtenues à partir des ressemblances entre un exemple e et chaque élément de E, comme l'aurait fait le processus de décision, mais plutôt sur l'union de ces ensembles pour tous les exemples de E. Les éléments de cette base sont ensuite pondérés puis évalués. Après, le premier élagage consistant à supprimer les règles apportant trop d'erreurs est réalisé. On ne peut pas immédiatement appliquer le second élagage, qui ne conserve que les règles prépondérantes parmi celles restant après le premier élagage, car toutes les règles ainsi obtenues ne reconnaissent pas nécessairement tous les exemples de E. En effet, comme le second élagage est essentiellement basé sur les valeurs des règles, nous pourrions perdre une règle expliquant très bien un exemple pour une autre plus spécifique ou plus générale ayant une valeur meilleure mais ne reconnaissant plus notre exemple.

Exemple : prenons le problème des digits. Supposons que l'exemple à reclasser soit 1011011 et que le système possède deux règles $r_1 = (1 *$

Algorithme 45 Classement d'un nouvel exemple.

```
CLASSEREXEMPLE(i E, R) : classe
    voteC+ ← 0
    pour chaque e ∈ C+ faire
    |   vote ← faux
    |   pour chaque r = (s → C+) ∈ R tant que ¬vote faire
    |   |   si reconnait[r, e] alors
    |   |   |   voteC+ ← voteC+ + 1
    |   |   |   vote ← vrai
    voteC− ← 0
    pour chaque e ∈ C− faire
    |   vote ← faux
    |   pour chaque r = (s → C−) ∈ R tant que ¬vote faire
    |   |   si reconnait[s, e] alors
    |   |   |   voteC− ← voteC− + 1
    |   |   |   vote ← vrai
    si voteC+ > voteC− alors retourner C+
    si voteC− > voteC+ alors retourner C−
    retourner indécision
```

$*1011 \rightarrow C$) et $r_2 = (11 * 1011 \rightarrow C)$ ayant respectivement pour valeur 0,30 et 0,29. r_2 sera conservée car elle a la meilleure valeur, et r_1 sera supprimée puisqu'elle est plus générale. Or r_2 ne reconnaît pas e. Par conséquent r_2 ne peut pas expliquer e.

On ne peut donc pas dissocier le deuxième type d'élagage des exemples à reclasser. On considérera chacun des exemples de E au travers de la procédure suivante. Pour chaque exemple e, la fonction qui détermine les règles prépondérantes est appelée sur l'ensemble des règles dont la ressemblance reconnaît e. Ensuite, on va appliquer la procédure de classement à e. Si e est bien reclassé par le système, alors l'ensemble des règles expliquant son classement est ajouté à l'ensemble des règles explicatives du système. Par contre si l'exemple est mal classé, on ne procède à aucun ajout. Cela ne veut pas dire que ces règles reconnaissant e ne doivent pas figurer dans la base des règles explicatives, cela signifie seulement que si elles doivent être présentes, elles seront apportées par un autre exemple qui lui sera bien classé.

En répétant ce processus pour tous les exemples de E, le système peut fournir à l'utilisateur l'ensemble des règles expliquant le concept à apprendre.

L'algorithme 46 détaille ce mécanisme d'apprentissage.

Le classement d'un nouvel exemple ν après le processus d'apprentissage se fait à partir des règles explicatives qui reconnaissent ν. On détermine celles qui sont prépondérantes puis on fait voter les exemples. L'algorithme 47 propose une implémentation de ce classement.

Cette approche globale présente de nombreux avantages :

- Beaucoup plus de règles sont engendrées, on a donc beaucoup plus de chance d'introduire des règles intermédiaires pertinentes qui autrement n'auraient pas été présentes. On peut espérer trouver ainsi les meilleures règles.

- On évite de répéter plusieurs fois les mêmes calculs. Il est moins coûteux de faire le premier élagage sur l'ensemble des règles issu de la comparaison de tous les exemples entre eux que de réaliser cet élagage sur l'ensemble des règles obtenu pour chaque exemple.

- Il est possible d'interrompre la recherche de nouvelles ressemblances si les dernières tentatives n'en ont pas amenées.

Algorithme 46 CNN : une nouvelle méthode d'apprentissage.

APPRENTISSAGE(i $E, seuilC_+, seuilC_-$) : **ensemble de règles**
  $S \leftarrow \varnothing$
  pour *chaque* $e \in E$ **faire** CRÉERBASERESSEMBLANCES($E - \{e\}, e, S$)
  PONDÉRERBASERESSEMBLANCES(E, S)
  $R \leftarrow$ CRÉERBASERÈGLES(S)
  ÉVALUERBASERÈGLES($R, seuilC_+, seuilC_-$)
  ORGANISERBASERÈGLES(R)
  $R_{explic} \leftarrow \varnothing$
  pour *chaque* $e \in E$ **faire**
    $R_e \leftarrow \varnothing$
    pour *chaque* $r = (s \rightarrow C) \in R$ **faire**
      si *reconnait*$[s, e]$ **alors** AJOUTER(r, R_e)
    DÉTERMINERRÈGLESPRÉPONDÉRANTES(R_e)
    $cl \leftarrow$ CLASSEREXEMPLE($E - \{e\}, R_e$)
    si *la classe de e est cl* **alors** $R_{explic} \leftarrow R_{explic} \cup R_e$
  retourner R_{explic}

Algorithme 47 Classement d'un nouvel élément à partir des règles apprises.

CLASSEREXEMPLEAPARTIRRESSEMBLANCES(i $E, seuilC_+, seuilC_-$) : **classe**
  pour *chaque* $r = (s \rightarrow C) \in R_{explic}$ **faire**
    si $\mathcal{A}(\nu, s)$ **alors** AJOUTER(r, R_ν)
  DÉTERMINERRÈGLESPRÉPONDÉRANTES(R_ν)
  retourner CLASSEREXEMPLE(E, R_ν)

- L'utilisation du mécanisme d'apprentissage plutôt que celui de la décision permet d'affiner le traitement de l'indécision et d'apprendre la négation d'un concept.

Le dernier point mérite d'être approfondi, et nous le ferons ci-dessous. Néanmoins cette procédure d'apprentissage a un inconvénient qui doit être signalé : dans le cas de problème de très grande dimension, il n'est pas évident que cette procédure d'apprentissage puisse simplement être mise en oeuvre, pour de simples raisons de temps de calcul. La procédure de décision peut néanmoins rester applicable dans de tels cas.

Traitement de l'indécision

Lors du choix de la classe d'un nouvel élément, deux cas peuvent se présenter :

- Soit une classe a une note supérieure à celle de l'autre classe pour l'exemple considéré et on peut prendre une décision.

- Soit les notes sont nulles et le système ne sait pas où classer l'exemple et par conséquent ne prend pas de décision.

Dans le premier cas, la décision qui s'impose, bien entendu, est de choisir la classe ayant la meilleure note.

Dans le second cas, plutôt que de s'intéresser uniquement à la description de l'exemple, on s'intéresse également à la description que l'on a des classes, par l'intermédiaire des règles explicatives du phénomène étudié. On cherche quelle est la classe pour laquelle on possède le moins d'explications : c'est celle possédant le plus d'éléments n'étant pas reconnus par au moins une des règles explicatives. En effet, si la quasi-totalité des exemples d'une classe C est reconnue par au moins une des règles, il est fort peu probable qu'un exemple non reconnu par ces mêmes règles appartienne à C. La classe ainsi choisie sera alors la classe par défaut.

Exemple: Soient deux classes C_+ et C_-, R_{explic} l'ensemble des règles explicatives. Supposons que 80% des éléments de C_+ soient plus spécifiques qu'au moins une règle de R_{explic} contre 40% pour C_-.

On peut donc dire que 80% des exemples de C_+ et 40% des exemples de C_- sont expliqués par R_{explic}. Soit ν un nouvel élément qui n'a pas dans R_{explic} de règles le reconnaissant. On sait que dans C_+, 20% des éléments

ne sont pas expliqués par R_{explic} contre 60% à C_-. Il est donc légitime de considérer que la description de ν corresponde à celle d'un élément non expliqué de C_-, plutôt qu'à celle d'un de C_+. ν sera donc assimilé à un élément de la classe C_-. Le système aura levé l'indécision.

Ceci fonctionne d'autant mieux que le contenu des classes est extrêmement différent. Si une classe C_+ représente les exemples possèdant une propriété définie, et l'autre C_- les exemples qui ne possèdent pas cette propriété, on mettra le seuil de C_- à 0 afin de ne rien apprendre sur C_-. On obtiendra ainsi des explications ne concernant que C_+ et la règle explicative de C_- sera la négation de la conjonction des règles expliquant C_+. On peut ainsi ***apprendre la négation d'un concept.***

4.4 Complexité

Appelons r_m le nombre moyen de ressemblances entre deux exemples. n est le nombre d'exemples de l'ensemble d'apprentissage. C_M désigne la complexité de la fonction MATCH pour trouver une ressemblance entre deux exemples. C_A représente la complexité de la fonction permettant de savoir si une ressemblance est plus générale qu'un exemple ou bien de comparer deux ressemblances. Nous négligerons le coût de la fonction CALCULERTAUXERREUR.

4.4.1 Processus de décision

- La complexité de la fonction CRÉERBASERESSEMBLANCES est en $O(nr_mC_M)$. On peut considérer que l'élimination des doubles se fait en $O(nr_m \log nr_mC_A)$.

- La complexité de la fonction PONDÉRERBASERESSEMBLANCES est au plus en $O(n^2r_mC_A)$.

- La fonction CRÉERBASERÈGLES a une complexité de l'ordre de $O(nr_m)$.

- La complexité de la fonction ORGANISERBASERÈGLES est en $O(n^2r_m^2C_A)$.

- La complexité de la fonction DÉTERMINERRÈGLESPRÉPONDÉRANTES est en $O(n^2r_m^2)$.

- La fonction CLASSEREXEMPLE a une complexité de l'ordre de $O(n^2 r_m)$.

La complexité du processus de décision est en $O(nr_m C_M + n^2 r_m^2 C_A)$.

4.4.2 Apprentissage

Dans le pire des cas, la phase d'apprentissage répète le processus de décision pour tous les exemples de l'ensemble d'apprentissage. Sa complexité est donc en $O(n \times (nr_m C_M + n^2 r_m^2 C_A))$.

Ces complexités sont assez élevées. Cependant, le nombre d'application de la fonction MATCH et des fonctions permettant de comparer deux ressemblances et une ressemblance avec un exemple, reste polynômial sur le nombre d'exemples.

4.5 Résultats pour des problèmes tests

Nous avons utilisé deux problèmes tests : le problème des digits et celui des Monks (voir plus loin). Le problème des digits admet de nombreuses variations qui permettent de simuler des problèmes réels ayant des caractéristiques très différentes. Le problèmes des Monks est un benchmark classique de l'apprentissage et on connaît les résultats de nombreuses méthodes.

Pour tous les tests effectués avec CNN, nous avons utilisé un seuil par classe, et une classe par défaut. Après de nombreuses expérimentations, il est apparu que chaque problème possédait une zone de seuils, représentant les minima globaux pour le taux d'erreur de classement d'un élément à partir des règles apprises. Cette zone étant assez «grande» et s'apparentant à une sorte de «cuvette», il existe de grandes chances de trouver ces minima. On peut cependant différencier les paires de seuils appartenant à la zone des minima grâce aux règles qu'ils produisent. En effet, pour un taux d'erreur identique, il vaut mieux choisir la paire conduisant à un nombre de règles explicatives minimum car ces règles correspondent à des explications concises et efficaces du phénomène étudié.

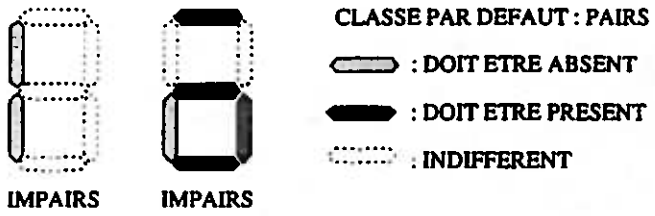


FIG. 4.5 - *Règles des Digits à 7 bits avec classes équilibrées.*

4.5.1 Problème des digits

Nous avons comparé les résultats fournis par CNN avec ceux obtenus avec les méthodes de voisinages.

Digits à 7bits et classes équilibrées

L'ensemble d'apprentissage possède 300 éléments alors que celui de test en possède 1000. Dans ces deux ensembles, il y a autant d'éléments pairs que d'impairs.

Les exemples sont bruités à 10%. Il s'agit du cas le plus simple à étudier pour le problème des digits. Quelle que soit la méthode employée, on obtient de bons résultats.

Les noyaux de Parzen font environ 6% d'erreur (avec une distance maximale de 1). Ces erreurs sont dûes au bruit qui transforme souvent un 9 en un 8, par exemple.

Les noyaux de Parzen pondérés obtiennent des résultats proches (moins de 9% d'erreurs). On peut utiliser ainsi un ensemble d'apprentissage possédant beaucoup moins d'éléments (66) tout en conservant d'excellents résultats. Les erreurs sont du même type que celles de la version non pondérée.

CNN fournit également de très bons résultats (environ 10% d'erreur) mais surtout donne des règles explicatives pertinentes. La figure 4.5 indique les règles trouvées.

Digits à 7bits et classes déséquilibrées

Les ensembles d'apprentissage et de tests possèdent autant d'éléments que pour les classes équilibrées, mais l'ensemble d'apprentissage possède plus d'éléments impairs que pairs (80% d'impairs et 20 % de pairs).

Le problème principal des classes déséquilibrées est qu'en classant systématiquement chaque élément dans la classe la plus importante, le système fera peu d'erreur

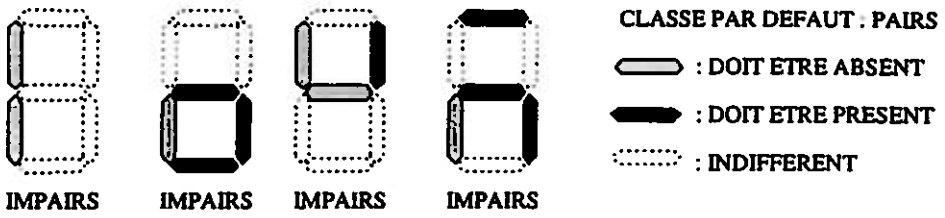


FIG. 4.6 - *Règles des Digits à 7 bits avec classes déséquilibrées*

en général (entre 20 et 25% d'erreurs), mais se trompera systématiquement sur la classe la plus faible . C'est ce qui ce passe pour les méthodes de voisinage. De plus ces méthodes apportent beaucoup d'indécision (environ 30%) que l'on peut lever en partie avec la pondération, mais en accentuant encore plus les erreurs (plus 25%).

Par contre, CNN a des performances quasi-similaires à celles réalisée avec des classes équilibrées (entre 8 et 12% d'erreur), avec des erreurs toujours dûes au bruit. Le déséquilibre effectué est en faveur de la classe des chiffres impairs, et on constate que les règles fournies ne concernent que cette classe. La figure 4.6 indique les règles obtenues.

Digits à 7bits et classe sûre

La classe dont on connaît avec précision les représentants est celle des pairs. La classe des impairs contenant tous les éléments que l'on peut obtenir avec 7 bits. Il y a 20% d'exemples pairs et 80% d'exemples impairs.

En ce qui concerne les noyaux de Parzen, leur comportement est le même que pour les classes déséquilibrées. Les erreurs observées sont également du même ordre.

CNN donne aussi des résultats similaires aux précédents, mais les règles produites sont extrêmement différentes (figure 4.7). On constate notamment qu'elles ont tendance à expliquer la classe des pairs, contrairement aux cas précédents. Il apparaît également que les chiffres 4, 5, et 9 sont reconnus par les mêmes règles, donc leur classement sera le même et il dépendra surtout de la qualité des exemples votant (la plupart du temps, ils sont considérés comme impairs, et il y a donc une erreur systématique sur le 4).

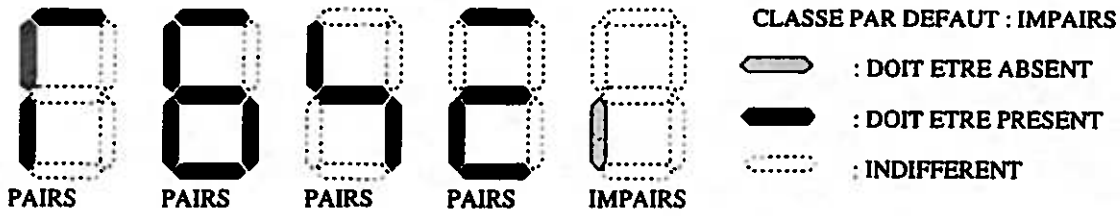


FIG. 4.7 - *Règles des Digits à 7 bits avec classes sûre.*

4.5.2 Les problèmes des Monks

Ce sont des benchmarks très répandus et très utilisés. Les instances des ensembles d'apprentissage et de test sont disponibles chez Sebastian Thrun (School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 14213, USA e-mail:thrun@cs.cmu.edu). De nombreuses méthodes ont été testées sur ces problèmes. Une présentation de ces méthodes et les résultats qu'elles obtiennent sont disponibles dans [Thrun *et al.*, 1991]. Ces problèmes sont liés à des «robots» décrits par six attributs :

Attribut	Valeurs possibles
x_1: forme de la tête	rond, carré, octogone
x_2: forme du corps	rond, carré, octogone
x_3: souriant	oui, non
x_4: objet porté	épée, ballon, drapeau
x_5: couleur de la veste	rouge, jaune, vert, bleu
x_6: porte une cravate	oui, non

Il s'agit de problèmes de classification binaire. Chaque problème est donné par la description logique d'une des classes. Chaque robot appartient ou non à cette classe, mais plutôt que de fournir une description complète de la classe comme ensemble d'apprentissage, on ne donne qu'un sous-ensemble des 432 robots possibles (avec leur classe d'appartenance). Le but est de retrouver la description initiale et de classer correctement les robots présents dans les ensembles de test. Nous ne nous intéresserons qu'aux problèmes Monk-1 et Monk-3.

- **Problème MONK-1:**
  concept: $(x_1 = x_2)$ ou $(x_5 = \text{rouge})$

La forme de la tête doit être la même que celle du corps ou la veste doit être rouge. Sur les 432 possibles, 124 sont utilisés pour l'apprentissage. Il n'y a pas d'erreur de classement dans les exemples, soit aucun bruit.

- **Problème MONK-3:**
  concept: (x_5 = vert et x_4 = épée) ou ($x_5 \neq$ bleu et $x_2 \neq$ octogone)
  La veste doit être rouge et le robot doit porter une épée, ou, la veste n'est pas bleue et le corps n'est pas octogonal. On dispose de 122 exemples dont 5% sont mal classés.

Le tableau suivant résume l'ensemble des résultats de diverses méthodes pour le problème des Monks. La méthode AQ-15 est une méthode descendante. ID-3 est une méthode basée sur les arbres de décision. Backpropagation et Cascade sont basées sur des réseaux neuronaux. Dans ce tableau : «Négation» ou «Neg» signifie que le système a appris la négation du concept, «Bonnes» veut dire que le système a fourni des règles concises et intéressante, «Mauvaises» représente un ensemble de règles apprises beaucoup trop important et inexploitable, «except» exprime que le système a considéré que certains exemples étaient des exceptions, «Matrice» indique que les résultats sont fournis sous la forme d'une matrice et n'ont donc aucun pouvoir explicatif.

Méthode	Monk-1	Monk-3	Règles1	Règles3
CNN	**100%**	**97.2%**	**Bonnes**	**Négation**
AQ17-DCI	100%	94.2%	Bonnes	
AQ17-HCI	100%	100%	Mauvaises	Neg + except
AQ15-GA	100%	100%		Neg + except
ID3	98.6%	94.4%		
Backprop.	100%	93.1%	Matrice	Matrice
Cascade	100%	97.2%		

AQ: K. De Jong, R.S. Michalski et al.
ID3: J. Kreuziger, R. Hamann et W. Wensel
Backpropagation: S. Thrun
Cascade corrélation: S. Fahlman

MONK-1

CNN reclasse correctement 100% des éléments dans ce problème. De plus les règles trouvées correspondent à celles demandées, à savoir:
(tête ronde ET corps rond) OU (tête carrée ET corps carré) OU (tête octogonale ET corps octogonal) OU (veste rouge).

Si un élément est reconnu par cette ressemblance, alors il appartient à la classe de robots désirée.

MONK-3

CNN reclasse correctement 97,2% des éléments dans ce problème. Cependant les règles qu'elle obtient ne concerne pas le concept que l'on veut apprendre, mais sa négation. Les règles obtenues sont les suivantes: (corps octogonal), (veste bleue), (corps octogonal et non souriant). Si un robot vérifie une de ces trois règles, il vérifie la négation du concept à apprendre.

Les résultats de notre système sont donc comparables aux meilleurs obtenus jusqu'à présent.

4.6 Perspectives

La méthode actuelle fonctionne bien pour des problèmes tests[1], mais elle pourrait être développée dans plusieurs directions:

- Lorsque deux règles en faveur d'une même classe ont des valeurs très proches et qu'elles sont liées par la relation $\geq$, le système conserve actuellement celle dont la valeur est minimale. Cependant ces valeurs ne sont que des estimations de la borne supérieure de leur taux d'erreur respectif, ce qui signifie que le classement de leur taux d'erreur réel pourrait inverser le choix du système dans le cas de telles règles. Il serait donc intéressant d'étudier le comportement de la méthode si on choisissait de conserver les deux règles ou de ne conserver que la plus générale des deux.

1. Nous verrons dans le chapitre 6 qu'elle donne de très bons résultats en chimie.

- Lorsqu'on détermine toutes les ressemblances existantes entre les éléments de l'ensemble d'apprentissage, il est fort possible de ne pas découvrir une ressemblance primordiale. En effet, les exemples utilisés peuvent très bien en générer qui soient plus spécifiques ou plus générales que cette ressemblance sans jamais la générer. On pourrait donc faire subir des perturbations aux ressemblances découvertes ou les recombiner afin d'obtenir de meilleures explications.

- Lors du choix de la classe par défaut, on s'intéresse au nombre d'exemples de chaque classe dont la description est reconnue par au moins une des règles de l'ensemble des règles explicatives. Cependant si 20% des exemples d'une classe sont reconnus par au moins une règle, contre 21% pour l'autre, on ne peut pas décemment choisir une classe par défaut. D'une manière générale, il faudrait cerner les cas où l'on peut déterminer une classe par défaut.

- La conservation ou la suppression d'une règle dépend de l'estimation de la borne supérieure de son taux d'erreur, et de la valeur maximale que l'expert accepte pour cette estimation (le seuil). Or il existe une combinaison des seuils des différentes classes permettant d'obtenir une classification optimale des éléments tout en conservant de bonnes explications. On pourrait donc essayer de faire découvrir les valeurs de ces seuils par le système en utilisant davantage le contenu de l'ensemble d'apprentissage.

- Enfin, il doit être relativement aisé d'étendre cette méthode, qui pour l'instant ne travaille que sur deux classes, à un apprentissage multi-classes.

4.7 Conclusion

Dans les domaines qui ont servi d'appui au développement de notre méthode CNN, celle-ci est apparue efficace. Elle classe bien des éléments n'étant pas présents dans l'ensemble d'apprentissage, et surtout elle fournit des règles pertinentes pour décrire les concepts appris. Elle reste cependant perfectible, notamment en déterminant automatiquement les seuils, et extensible en ajoutant d'autres types de ressemblances détectables, et en l'appliquant à l'apprentissage multi-classes.

Contrairement à ANNA, les explications ne sont pas nécessairement les plus générales, mais celles dont les performances sont les plus élevées et ne présentant pas de redondance. Cela signifie que parmi les règles découvertes, on ne conserve que celles qui sont suffisamment générales pour expliquer un maximum d'exemples positifs, et qui sont suffisamment spécifiques pour réfuter un maximum d'exemples négatifs. De plus il est possible d'apprendre la négation d'un concept si elle est «plus simple» que le concept lui même. En effet, il suffit de fixer à 0 le seuil de la classe C qui représente la négation du concept, de façon à ne pas conserver de règles prédisant C. La qualité des règles concernant l'autre classe dépendra ensuite de la valeur de son seuil.

Chapitre 5

Modélisation du problème

Dans le chapitre précédent, nous avons présenté une méthode d'apprentissage très générale. Pour fonctionner celle-ci nécessite :

- un ensemble d'apprentissage constitué d'exemples vérifiant le concept et d'exemples ne le vérifiant pas;
- un langage de description des exemples et des généralisations;
- un mécanisme de généralisation (la fonction MATCH);
- une relation d'appariement (les fonctions $\mathcal{A}$ et $\geq$).

Ce chapitre présente comment nous avons introduit ces 4 notions pour le problème de l'apprentissage des liaisons stratégiques en synthèse organique.

Nous présenterons donc tout d'abord la construction de l'ensemble d'apprentissage, nous discuterons alors des problèmes que nous avons rencontrés pendant la modélisation. Puis nous détaillerons le langage de description des exemples. Cela nous permettra ensuite d'introduire le mécanisme de généralisation et la relation d'appariement. Enfin nous conclurons.

5.1 Construction de l'ensemble d'apprentissage

L'ensemble d'apprentissage est construit à partir de réactions fournies par ORAC et choisies par un expert en synthèse organique (Claude Laurenço). Les produits for-

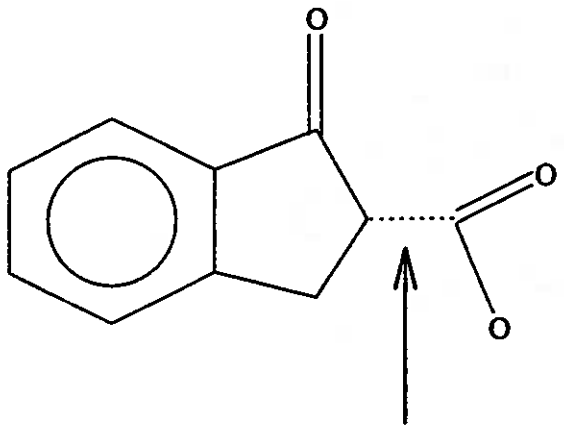


FIG. 5.1 - *Un exemple de molécule créée par une synthèse.*

més (c'est-à-dire des molécules) par chaque réaction ainsi que les liaisons créées ont été isolés. Ils sont donnés en annexe. ***Chaque liaison non aromatique entre deux carbones de chaque produit formé constitue, avec son environnement, un exemple.*** Ainsi, pour la molécule présentée en figure 5.1, nous obtiendrons 5 exemples puisqu'elle contient 5 liaisons non aromatiques entre deux carbones. Un exemple est donc totalement défini par une molécule et une liaison de cette molécule.

Un exemple est positif si la liaison qu'il référence a été créée par la réaction. Dans le cas contraire il est négatif. Dans l'exemple précédent, la liaison en pointillée avec son environnement est un exemple positif. Toutes les autres liaisons avec leur environnement sont des exemples négatifs.

Par conséquent, si l'on dispose de n molécules et si m_i représente le nombre de liaisons C-C d'une molécule i alors on aura $\sum_{i=1}^{n} m_i$ exemples. Et si pour chaque molécule i on a l_i liaisons créées par la réaction ayant formé la molécule, alors on aura $\sum_{i=1}^{n} l_i$ exemples positifs et $\sum_{i=1}^{n} m_i - l_i$ exemples négatifs.

Il est très important de noter que l'ensemble d'apprentissage n'est pas construit à partir de toutes les synthèses permettant de créer une molécule donnée. On ne dispose, en effet, dans ce domaine, que de connaissances partielles. Aussi, il est tout à fait possible qu'une liaison avec son environnement soit considérée comme un exemple négatif alors qu'il existe une synthèse réalisable la créant. L'ensemble d'apprentissage tel que nous l'avons construit est donc très particulier. Il présente deux caractéristiques

assez inhabituelles :

1. Les classes sont très déséquilibrées puisqu'une synthèse crée, en général, une seule liaison dans une molécule. Il y a dans la base environ 7 exemples négatifs pour 1 exemple positif.

2. La connaissance à l'intérieur de la classe des exemples négatifs est très incertaine. On peut assurer qu'un exemple positif représente toujours une liaison stratégique avec son environnement. Par contre, il est impossible de garantir qu'un exemple négatif ne vérifie pas le concept à apprendre. En fait la classe des exemples négatifs regroupe tous les éléments dont on ne connaît pas le rapport avec le concept à apprendre. Cette classe est donc fortement bruitée. Nous retrouvons là le troisième cas de figure envisagé pour les digits, où la classe positive contenait les chiffres pairs, et la classe négative n'importe quelle description.

5.2 Description des exemples

Une molécule se représente «naturellement» par une description structurelle. En effet une molécule peut être vue comme un graphe, non orienté et connexe, dont les sommets et les arêtes sont étiquetés[1]. Aussi un exemple est entièrement déterminé par la donnée de ce graphe et par la mise en évidence d'une arête de ce dernier. Dans la suite nous noterons (G, l) un exemple.

Les étiquettes des sommets et des arêtes permettent l'introduction de connaissances de la synthèse en chimie organique. Un sommet avec son étiquette représente un atome de la molécule. L'étiquette contient certaines propriétés chimiques d'un atome :

- son type;
- sa stéréochimie;
- sa charge;

1. Nous renvoyons le lecteur au chapitre 8 de la partie I de cette thèse pour une présentation des graphes étiquetés

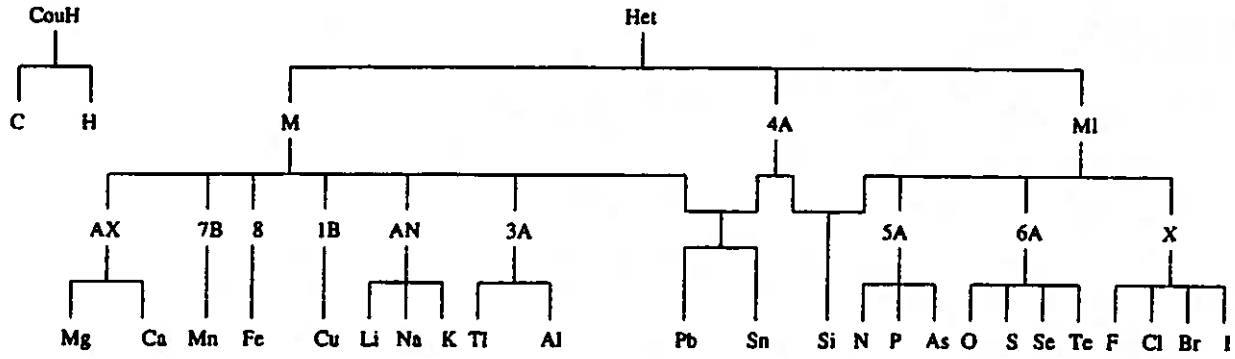


FIG. 5.2 - *Hiérarchie des types atomiques.*

- sa fonctionnalité;
- l'effet inducteur qu'il subit;
- l'effet mésomère auquel il est soumis.

De même une arête avec son étiquette représente une liaison de la molécule. L'étiquette comprend certaines caractéristiques chimiques d'une liaison :

- l'aromaticité de la liaison;
- l'ordre de la liaison;
- la stéréochimie de la liaison.

Nous allons détailler chacune de ces propriétés pour les atomes et les liaisons.

5.2.1 Étiquette des sommets

Type d'un atome

Le type d'un atome est un élément de la hiérarchie présentée en figure 5.2. Cette classification reprend en partie celle de Mendeleïev (pour les halogènes, les alcalins, les atomes du groupe IIIa ...) mais permet aussi de faire la distinction entre les métaux et les métalloïdes ou encore entre les carbones/hydrogènes et les hétéroatomes.

Stéréochimie d'un atome

La stéréochimie d'un atome est un élément de la hiérarchie donnée en figure 5.3.

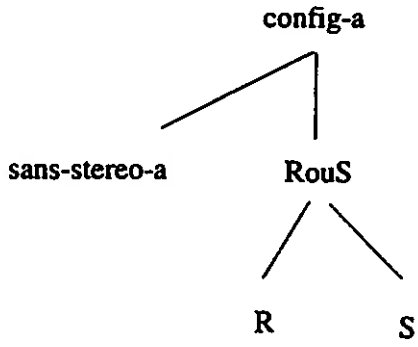


FIG. 5.3 - *Hiérarchie des stéréochimies d'un atome.*

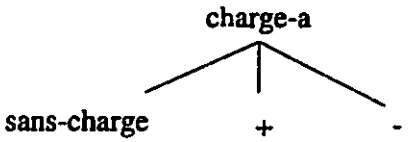


FIG. 5.4 - *Hiérarchie des charges atomiques.*

Charge d'un atome

La charge d'un atome est un élément de la hiérarchie donnée en figure 5.4.

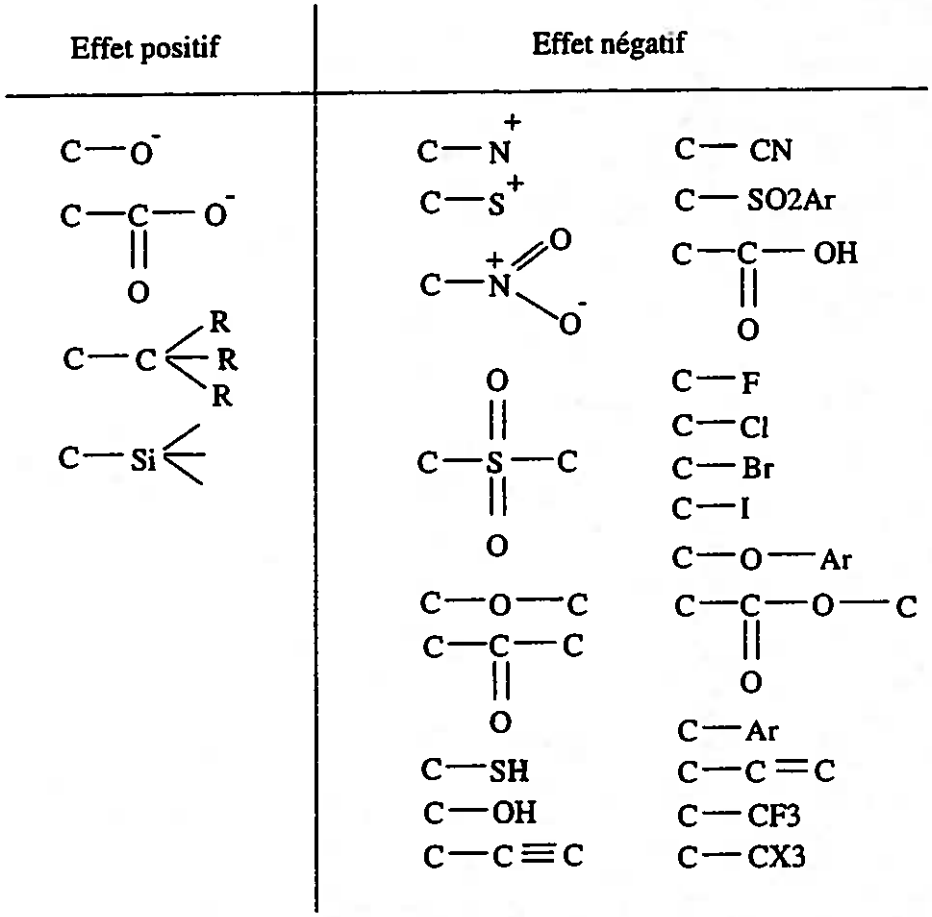
Fonctionnalité d'un atome

La fonctionnalité d'un atome est un booléen qui est vrai si l'atome est un carbone et s'il est fonctionnel. Un carbone est fonctionnel si l'une au moins des deux conditions suivantes est remplie :

- il est relié à un hétéroatome.
- il est relié à un autre atome par une liaison d'ordre strictement supérieur à 1.

Effets électroniques

Un carbone est soumis à un ***effet inducteur*** si l'un des environnements donné en figure 5.5 est inclus dans son environnement structurel. Par exemple en utilisant le premier élément positif on voit que si l'atome de carbone est relié à un oxygène chargé négativement alors il sera soumis à un effet inducteur positif. L'effet inducteur est un élément de la hiérarchie donnée en figure 5.6.



R est un carbone ou un hydrogène

Ar désigne un cycle aromatique

FIG. 5.5 - *Effets inducteurs pour les carbones les plus à gauche.*

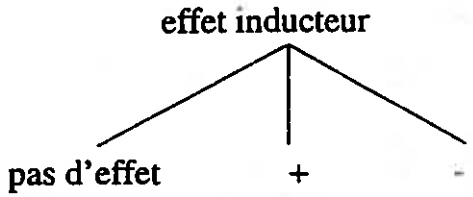


FIG. 5.6 - *Hiérarchie des effets inducteurs.*

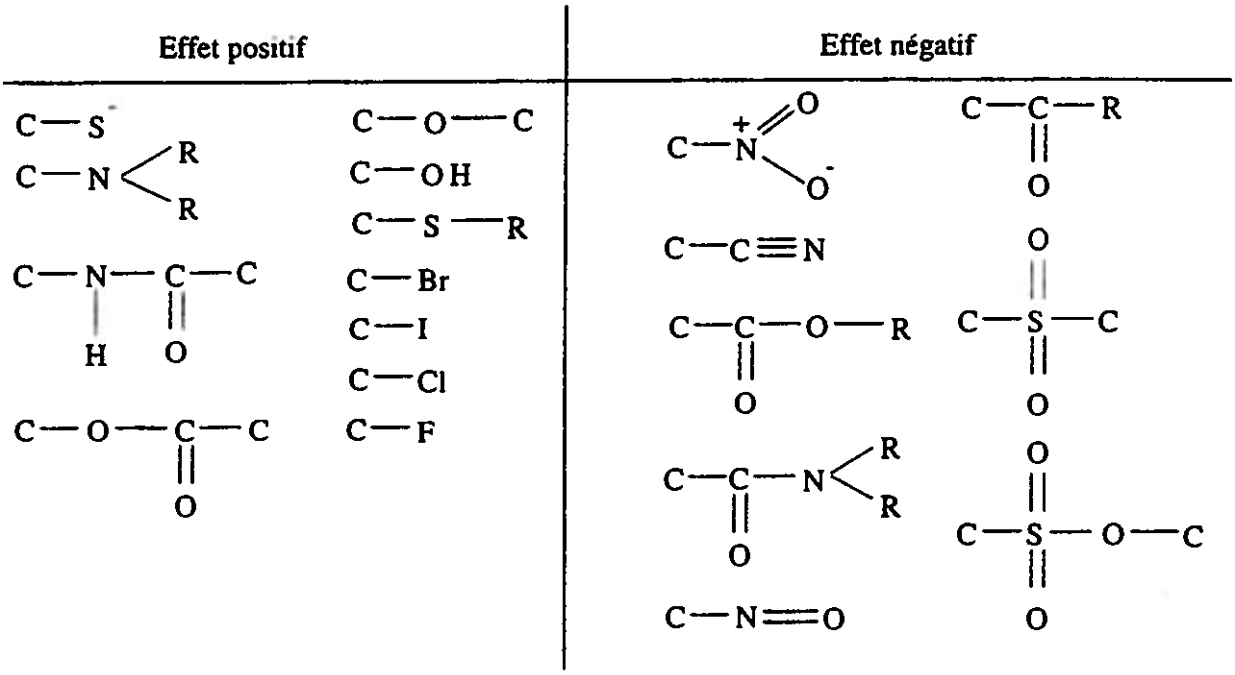


FIG. 5.7 - *Effets mésomères pour les carbones les plus à gauche*

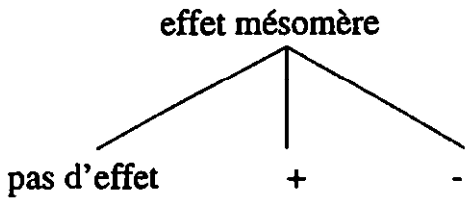


FIG. 5.8 - *Hiérarchie des effets mésomères.*

De même nous dirons qu'un carbone subit un ***effet mésomère*** si l'un des environnements donné en figure 5.7 est inclus dans son environnement structurel. L'effet mésomère est un élément de la hiérarchie donnée en figure 5.8.

5.2.2 Étiquette des arêtes

Aromaticité d'une liaison

L'aromaticité d'une liaison est un booléen qui est vrai si la liaison est aromatique, et faux sinon.

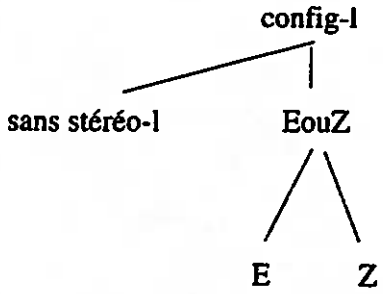


FIG. 5.9 - *Hiérarchie des stéréochimies d'une liaison.*

Stéréochimie d'une liaison

La stéréochimie d'une liaison est un élément de la hiérarchie donnée en figure 5.9.

Ordre d'une liaison

L'ordre d'une liaison est 1 si la liaison est simple, 2 si elle est double, et 3 si elle est triple.

5.3 Problèmes liés à la modélisation

Dans cette section nous allons expliquer pourquoi nous avons introduit les effets électroniques dans la modélisation des molécules. C'est, à notre connaissance, la première fois que ces effets sont pris en compte lors de la comparaison de molécules.

Les systèmes d'apprentissage imposent, le plus souvent, deux conditions sur les généralisations :

1. qu'elles représentent une information sensée du domaine d'application;
2. qu'elles s'organisent selon la relation de généralisation-spécialisation (relation $\geq$ des chapitres précédents) en un réseau comportant peu de contradictions. C'est-à-dire qu'il faut éviter, autant que possible, qu'une règle en faveur d'un concept soit plus générale qu'un règle de tendance inverse et vice et versa.

Si l'on n'y prend pas garde, il est très facile de ne pas vérifier ces deux conditions en chimie organique. Prenons l'exemple de deux composés très répandus : les éthers et les esters (cf. figure 5.10). Si l'on ne tient pas compte des effets électroniques nous

FIG. 5.10 - *Un éther et un ester.*

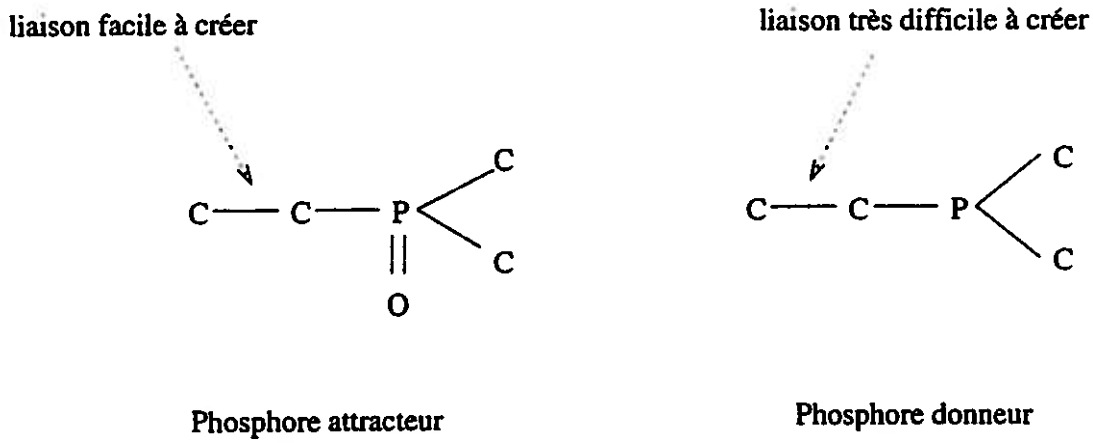


FIG. 5.11 - *Atome attracteur et atome donneur.*

trouverons qu'un éther est plus général qu'un ester. Cela signifie pour le système d'apprentissage, d'une certaine manière, qu'un ester possède les propriétés chimiques d'un éther. Or c'est totalement faux en chimie organique car ces deux composés ont des comportements extrêmement différents. En aucune façon un ester n'est un éther particulier. Donc, il ne doit pas être possible d'établir une quelconque relation de généralisation-spécialisation entre ces composés.

L'introduction des effets électroniques permet de résoudre en grande partie ce problème. D'après la figure 5.5, les carbones de l'éther subissent un effet inducteur négatif alors que le carbone de gauche de l'ester ne subit aucun effet. Si l'on tient compte des effets, c'est-à-dire si l'on interdit l'appariement de deux atomes n'ayant pas les mêmes effets, alors on ne pourra plus établir de lien entre un éther et un ester.

Par ailleurs, les effets électroniques ont une influence sur le caractère stratégique des liaisons. Considérons les deux molécules de la figure 5.11. Il est très facile de créer la liaison carbone-carbone de la molécule de gauche, alors qu'il est très difficile de créer la même liaison de la molécule de droite. Cela s'explique par la présence d'un phosphore attracteur dans la molécule de gauche, alors qu'il est donneur pour celle

de droite. L'environnement du phosphore modifie l'environnement électronique des carbones. Ceux reliés à un phosphore attracteur vont avoir tendance à perdre leur électrons, tandis que ceux relié à un phosphore donneur auront plutôt tendance à les conserver.

Par conséquent en utilisant les effets électroniques lors de la modélisation des molécules, nous nous sommes interdit de trouver une quelconque relation de généralisation entre certaines molécules et nous avons aussi introduit une connaissance supplémentaire du domaine pouvant nous aider à caractériser une liaison stratégique.

5.4 Mécanisme de généralisation

La généralisation est le mécanisme qui permet d'extraire des caractéristiques communes à deux exemples. Elle correspond à la fonction MATCH que nous avons définie dans le chapitre précédent. C'est une tâche délicate qui requiert à la fois des heuristiques et une grande quantité de connaissances. Notons que pour notre problème la recherche de généralisations entre deux liaisons et leur environnement s'apparente à la recherche de ressemblances entre les environnements, autrement dit à la recherche de similitudes entre molécules. Nous avons ici décidé que ces généralisations seraient maximalement spécifiques, c'est-à-dire telles que le fait de leur rajouter un atome ou une liaison fasse qu'elles ne seront pas plus générales qu'au moins une des deux molécules.

Ce mécanisme peut être réalisé par réécriture des descriptions des différents exemples jusqu'à ce qu'elles soient semblables. Afin d'effectuer ces réécritures, on a besoin d'une grande quantité d'informations sémantiques traduites sous forme de règles ou de hiérarchies. Cette méthode ne peut malheureusement pas être utilisée avec des molécules représentées à l'aide de graphes étiquetés, car elle entraînerait beaucoup trop de calculs. Deux solutions sont alors possibles : on tente de résoudre ce problème en utilisant momentanément une représentation attribut-valeur des données, ou on utilise un autre algorithme manipulant uniquement des données symboliques et structurées.

De nombreuses tentatives utilisant la première approche ont été faites. Certains systèmes comme ORAC ou REACCS, pour ne citer que les plus célèbres[2], proposent

2. Le lecteur particulièrement intéressé par ce type de méthodes pourra consulter le livre de

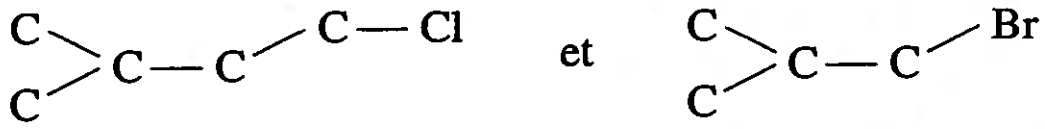
des calculs de ressemblance entre deux molécules basés sur des critères numériques. La méthode généralement employée utilise environ 1400 attributs pour décrire une molécule. Ces attributs sont par exemple : le nombre d'atomes de carbone, d'oxygène etc..., ou encore le nombre de cycles de taille 6. Elle attribue à chacun de ces attributs un poids. Puis elle calcule la similarité en utilisant un coefficient particulier [Grethe and Moock, 1990] [Moock *et al.*, 1988]. Donc à partir d'une représentation attribut-valeur des molécules elle définit une similarité numérique, qui est fournie sous la forme d'un pourcentage. Elle ne précise à aucun moment pourquoi il y a similarité et il est impossible de reconstituer une représentation structurelle à partir de cette méthode. De plus, à partir des résultats numériques le chimiste n'est pas capable de dire pourquoi les deux molécules se ressemblent. Nous ne pouvions donc pas utiliser de telles méthodes.

Nous nous sommes donc tournés vers la seconde approche. Ainsi, nous avons choisi une méthode structurelle basée sur la recherche de sous-graphes partiels communs maximaux[3] (au sens de l'inclusion). Certains travaux ont déjà été entrepris dans cette direction, mais leur propos est plutôt la recherche des plus grands sous-graphes communs[3] [Varkony *et al.*, 1979; Bayada *et al.*, 1992]. Notre choix a été motivé après de nombreuses discussions avec les experts. Prendre les sous-graphes communs maximaux semble être un bon choix puisqu'il recouvre les autres (on peut retrouver les plus grandes sous-graphes communs) et aussi parce que c'est l'approche habituelle en apprentissage. Mais cette approche est beaucoup plus complexe et difficile que la précédente, car le problème à résoudre est de nature exponentielle. Nous allons détailler dans la suite les notions que nous venons d'employer et les algorithmes que nous avons utilisés.

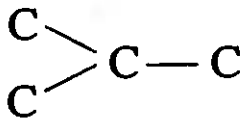
Par ailleurs, l'une des originalités de notre approche, du point de vue de la chimie, est l'introduction d'un mécanisme de généralisation des étiquettes. Habituellement ces dernières sont utilisées afin de ne pas obtenir certaines généralisations non-pertinentes. En chimie organique, on peut, par exemple, refuser une généralisation qui associe un brome avec un carbone. Pour ce faire les méthodes classiques imposent que les types des atomes soient respectés (c'est-à-dire égaux). Pour les deux

Johnson et Maggiora [Johnson and Maggiora, 1990] qui propose de nombreux algorithmes de calculs numériques de similarité entre molécules.

3. Cette notion est détaillée un peu plus loin.



Sous-graphes communs maximaux sans utiliser la hiérarchie des types atomiques :



Sous-graphes communs maximaux en utilisant la hiérarchie des types atomiques :

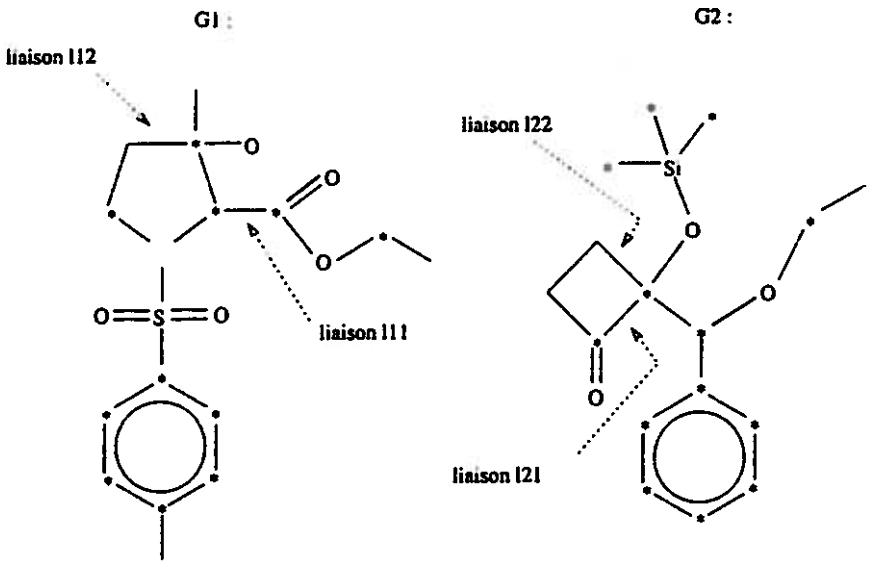
FIG. 5.12 - *Sous-graphes communs maximaux avec et sans utilisation de la hiérarchie des types atomiques.*

premières molécules de la figure 5.12, ces méthodes ne fournirons qu'une seule ressemblance. Du point de vue chimique ce renseignement n'est pas suffisant. On peut en effet considérer que les molécules ont d'autres caractéristiques communes. Pour un chimiste la ressemblance est plutôt exprimée par les deux ressemblances du bas de la figure 5.12.

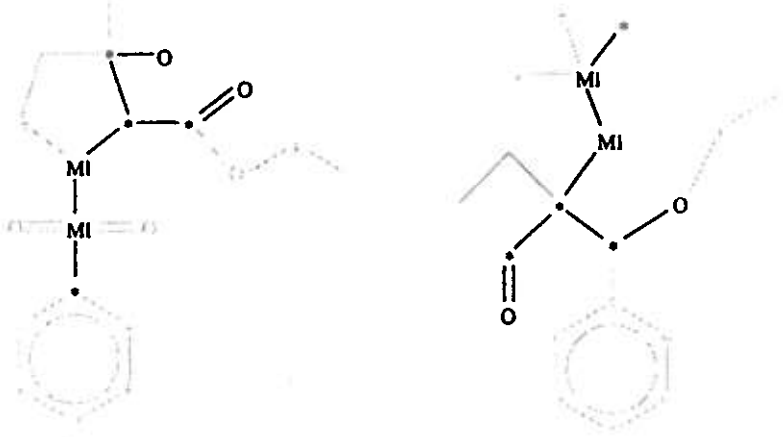
On obtient ce type de résultat en utilisant la taxonomie du type des atomes, qui exprime que le chlore et le brome sont tous deux des halogènes (représentés par un X dans la figure), autrement dit nous généralisons le chlore et le brome en halogène. La détermination de telles taxonomies, a été faite afin de prendre en compte de nombreuses propriétés chimiques des molécules comme nous l'avons montré dans la deuxième section de ce chapitre, mais aussi pour fournir des ressemblances plus pertinentes que celles habituellement trouvées.

La figure 5.13 montre deux exemples réels de généralisation. Dans cette figure $*$ représente un carbone fonctionnel, Ml un métalloïde; les effets électroniques ne sont pas mentionnés mais ils sont pris en compte.

La mise en oeuvre d'un tel mécanisme est relativement complexe, car pour pouvoir généraliser les étiquettes il faut connaître toutes les correspondances entre les sommets



Une généralisation de (G1,l11) et (G2,l21) :



Une généralisation de (G1,l12) et (G2,l22) :

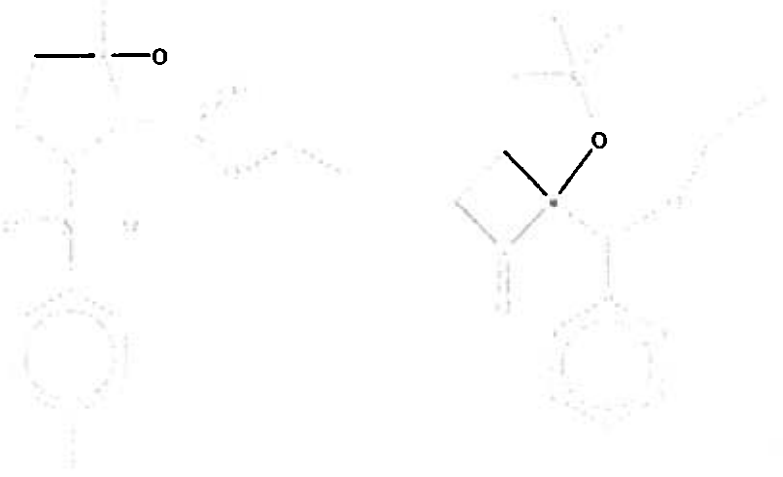


FIG. 5.13 - *Généralisations de deux exemples.*

G1 G2

2 sgpc appariés maximaux appariant ensemble les arêtes en gras:

a est apparié avec a'
b est apparié avec b'

S1

a est apparié avec b'
b est apparié avec a'

S2

FIG. 5.14 - *Deux exemples de sgpc appariés maximaux.*

du sous-graphe partiel commun maximal et les deux graphes des exemples.

Nous allons détailler dans la suite les notions que nous avons employées et la algorithmes que nous avons utilisés.

5.4.1 Sous-graphes partiels communs maximaux connexes

Définition 52 *Soient G_1 et G_2 deux graphes, le triplet (S, m_1, m_2) où :*

- *S est un graphe;*
- *m_1 un morphisme injectif de S dans G_1;*
- *m_2 un morphisme injectif de S dans G_2.*

est appelé ***sous-graphe partiel commun apparié*** *de G_1 et G_2. S est un sous-*

***graphe partiel commun, sgpc** en abrégé, de G_1 et G_2.*

Si S est connexe on parle de sous-graphe partiel connexe commun.

Nous donnons, seulement à titre indicatif, la définition de plus grand sous-graphe partiel car nous avons précédemment employé ce terme.

Définition 53 *Soient G_1 et G_2 deux graphes, (S, m_1, m_2) un sgpc apparié et f une fonction de «taille» (cette fonction peut par exemple retourner le nombre de sommets du graphes), s'il n'existe pas d'autres sgpc S' vérifiant $f(S') > f(S)$, alors S est un **plus grand sous-graphe partiel commun.***

Définition 54 *Soient G_1 et G_2 deux graphes, et (S, m_1, m_2) un sgpc apparié, on dit que :*

• *(S, m_1, m_2) **apparie un sommet** x_1 de G_1 avec un sommet x_2 de G_2 si et seulement s'il existe un sommet x_S de S vérifiant $m_1(x_S) = x_1$ et $m_2(x_S) = x_2$.*

• *(S, m_1, m_2) **apparie une arête** e_1 de G_1 avec une arête e_2 de G_2 si et seulement s'il existe deux sommets x_S et y_S tels que $\{m_1(x_S), m_1(y_S)\} = e_1$ et $\{m_2(x_S), m_2(y_S)\} = e_2$.*

Ces notions vont nous permettre de définir un sgpc apparié maximal. De manière intuitive, un sgpc apparié (S, m_1, m_2) est maximal s'il n'existe pas un autre sgpc apparié (S', m'_1, m'_2) tel que S soit isomorphe à un sous-graphe partiel de S' et tel que l'appariement de n'importe quel sommet de G_1 par (S, m_1, m_2) soit identique à l'appariement de ce sommet par (S', m'_1, m'_2). Nous proposons la définition suivante :

Définition 55 *Soient G_1 et G_2 deux graphes, et (S, m_1, m_2) un sgpc apparié avec $S = (X_S, E_S)$. (S, m_1, m_2) est **maximal** si et seulement si quelque soit le sgpc apparié (S', m'_1, m'_2) différent de (S, m_1, m_2) ,il n'existe pas de morphisme injectif f de S dans S' vérifiant :*

$$\forall x \in X_S : m_1(x) = m'_1(f(x)) \text{ et } m_2(x) = m'_2(f(x))$$

Cette définition de la maximalité est très importante pour la suite de l'exposé. Notons que S peut être isomorphe à un sous-graphe de S' et être maximal. Nous illustrons cela à partir de la figure 5.14. Dans cette figure, $S1$ est isomorphe à un

sous-graphe de $S2$. Mais $S1$ est maximal car $S1$ apparie a avec a' alors que $S2$ apparie a avec b'.

Proposition 11 *Soient G_1 et G_2 deux graphes, un sgpc apparié (S, m_1, m_2) est **connexe** si et seulement si le graphe S est connexe.*

Cette proposition est très importante car il existe toujours un nombre exponentiel de sgpc maximaux appariés de deux graphes $G_1 = (X_1, E_1)$ et $G_2 = (X_2, E_2)$, si l'on ne tient pas compte de la connexité. Pour montrer cela, considérons un sous-ensemble Y_1 quelconque de sommets de G_1, un sous-ensemble Y_2 de sommets de G_2 de même cardinal et une bijection f de Y_1 dans Y_2. Construisons un graphe S ayant comme ensemble de sommets Y_1 et un ensemble d'arête E tel que $(a, b) \in E$ si $(a, b) \in E_1$ et $(f(a), f(b)) \in E_2$. S est un sous-graphe partiel de G_1 et de G_2, c'est donc un sgpc de G_1 et G_2. Soit $k = min(|X_1|, |X_2|)$, si S a k sommets alors S est nécessairement maximal. De cela nous pouvons déduire qu'il existe au moins autant de sgpc maximaux appariés que de bijections dans un ensemble de k sommets, c'est-à-dire $k!$. Par ailleurs, la plupart de ces sgpc maximaux appariés n'ont aucun intérêt car ils contiennent de nombreux sommets isolés. L'introduction de la connexité va permettre de diminuer considérablement le nombre de sgpc maximaux appariés possibles et éviter d'en obtenir systématiquement un nombre exponentiel.

À partir des sgpc maximaux connexes appariés de deux graphes, nous pouvons proposer un premier mécanisme de généralisation pour deux exemples. Pour tenir compte des étiquettes, nous allons utiliser les deux fonctions booléennes présentées dans le chapitre 8 de la partie I, c'est-à-dire $\sigma_G(x, y)$ qui détermine la compatibilité des étiquettes des sommets x et y, et $\alpha_G(e, f)$ qui détermine celle des étiquettes des arêtes e et f. Comme nous les utilisons pour mettre en oeuvre le mécanisme de généralisation, nous avons ajouté à leur nom l'indice $_G$.

Définition 56 *Soient deux exemples $e_1 = (G_1, l_1)$ et $e_2 = (G_2, l_2)$, un sgpc maximal connexe apparié $(S = (X_S, E_S), m_1, m_2)$ de G_1 et G_2 appariant l_1 avec l_2 **prend en compte les étiquettes des sommets et des arêtes** si les les propriétés suivantes sont vérifiées :*

- $\forall x \in X_S : \sigma_G(m_1(x), m_2(x))$ *est vrai*
- $\forall \{x, y\} \in E_S : \alpha_G(\{m_1(x), m_1(y)\}, \{m_2(x), m_2(y)\})$ *est vrai.*

Pour trouver l'ensemble des généralisations de deux exemples, il faut tout d'abord résoudre le problème d'énumération ci-dessous et ensuite introduire la généralisation des étiquettes.

Sgpccm-apparié$(G_1, l_1, G_2, l_2, \sigma_G, \alpha_G)$:

> Données : G_1 et G_2 sont deux graphes étiquetés, l_1 est une arête désignée de G_1, l_2 est une arête désignée de G_2.
> Question : Quel est l'ensemble des sous-graphes partiels connexes, maximaux, communs et appariés à G_1 et G_2 qui apparient l_1 avec l_2 en tenant compte des étiquettes ?

5.4.2 Algorithme de résolution de Sgpccm-apparié

Dans un premier temps, nous ne considérerons ni l'appariement des arêtes désignées ni la connexité. Nous verrons par la suite qu'il est très simple d'introduire ces deux notions.

À partir des deux graphes étiquetés, nous allons construire un graphe semblable au graphe produit de ces graphes, puis nous montrerons que notre problème s'apparente à la recherche de cliques maximales particulières de ce graphe.

Définition 57 *Soient σ_G une fonction déterminant la compatibilité entre deux étiquettes de sommets, α_G une fonction déterminant la compatibilité entre deux étiquettes d'arêtes, $G_1 = (X_1, E_1, etq_S, etq_A)$ et $G_2 = (X_2, E_2, etq_S, etq_A)$ deux graphes étiquetés. On appelle* ***graphe de compatibilité*** *de G_1 et G_2 le quadruplet $(X_C, E_C, \sigma_G, \alpha_G)$ où :*

- *X_C est le sous-ensemble du produit cartésien de X_1 par X_2 tel que : $(x_1, x_2) \in X_C$ ssi $x_1 \in X_1, x_2 \in X_2$ et $\sigma_G(x_1, x_2)$ est vrai.*
- *$\{(x_1, x_2), (y_1, y_2)\} \in E_C$ ssi $x_1 \neq y_1$ et $x_2 \neq y_2$.*

Une arête $\{(x_1, x_2), (y_1, y_2)\}$ telle que $\{x_1, y_1\} \in E_1$ et $\{x_2, y_2\} \in E_2$ et $\alpha_G(\{x_1, y_1\}, \{x_2, y_2\})$ soit vrai, est appelée arête ***forte****. Les autres arêtes sont dites* ***faibles****.*

Proposition 12 *Soit $C = (X_C, E_C, \sigma, \alpha)$ le graphe de compatibilité de deux graphes G_1 et G_2, alors à toute clique maximale[4] de C correspond un sous-graphe partiel*

4. Une clique est maximale si elle n'est pas incluse dans une autre clique.

commun maximal apparié de G_1 et G_2 et réciproquement.

La preuve de cette proposition ne pose pas de problème particulier. Nous allons seulement expliquer comment à partir d'une clique on peut retrouver le sgpc de G_1 et G_2 et les morphismes injectifs. Considérons :

- $K_k = \{(x_{11}, x_{21}), ..., (x_{1k}, x_{2k})\}$ une clique maximale de taille k;
- X_S l'ensemble des sommets de K_k;
- Y_1 le sous-ensemble de sommets de G_1 tel que $y \in Y_1 \Leftrightarrow \exists (y, x_{2i}) \in X_S$;
- Y_2 le sous-ensemble de sommets de G_2 tel que $y \in Y_2 \Leftrightarrow \exists (x_{1i}, y) \in X_S$;
- m_1 une fonction de X_S dans Y_1 telle que $m_1((x_{1i}, x_{2i})) = x_{1i}$;
- m_2 une fonction de X_S dans Y_2 telle que $m_2((x_{1i}, x_{2i})) = x_{2i}$;
- l'ensemble E_S des arêtes fortes de K_k;
- le graphe $S = (X_S, E_S)$

On montre facilement que les fonctions m_1 et m_2 sont des morphismes injectifs respectivement de S dans G_1 et de S dans G_2.

La proposition précédente n'impose pas la connexité du sgpc maximal apparié. On pourrait énumérer tous les sgpc maximaux appariés, puis déterminer tous ceux qui apparient ensemble les deux arêtes désignées et qui sont connexes. Cette méthode n'est pas envisageable car, comme nous l'avons vu précédemment, il existe un nombre exponentiel de sgpc maximaux appariés si l'on n'impose pas la connexité. Pour résoudre Sgpccm-apparié, il faut énumérer uniquement les sgpc appariés qui présentent de l'intérêt (càd ceux qui sont connexes). Pour parvenir à ce résultat nous présentons, dans la section suivante, l'algorithme d'énumération de cliques maximales considéré comme le meilleur à ce jour. Ensuite nous expliquerons les modifications qu'il faut apporter à cet algorithme pour qu'il n'énumère que les cliques qui nous intéressent. Nous insistons sur le fait que nous ne proposons en aucun cas d'appliquer directement l'algorithme qui va suivre sur le graphe de compatibilité.

Algorithme de recherche de cliques maximales

Le problème de l'énumération de toutes les cliques maximales d'un graphe se rencontre fréquemment en pratique dans des domaines très variés et de nombreux algorithmes pour le résoudre ont été proposés [Bron and Kerbosh, 1973; Gerhards and Lindenberg, 1979; McGregor, 1982; Balas and Yu, 1986]. Nous avons choisi

d'implémenter l'algorithme de Bron et Kerbosh [Bron and Kerbosh, 1973] parce qu'il est considéré dans [Gerhards and Lindenberg, 1979] comme l'algorithme le plus efficace. De plus, il est simple à implémenter et assez facile à modifier pour qu'il ne produise que les cliques vérifiant certaines conditions [Balas and Yu, 1986; Kuhl, 1984].

L'algorithme proposé par Bron et Kerbosh est essentiellement un algorithme de type «branch and bound» basé sur une recherche ordonnée en profondeur d'abord. À chaque étape de l'algorithme, trois ensembles de sommets sont définis :

1. $COMPSUB$: contient les sommets qui font partie de la clique courante.

2. $CANDIDATES$: contient les sommets qui peuvent être sélectionnés et ajoutés dans $COMPSUB$ afin d'augmenter la taille de la clique courante. Bien entendu, un sommet x_i ne peut appartenir à $CANDIDATES$ que s'il est relié à tous les sommets de $COMPSUB$.

3. NOT : contient les sommets qui sont connectés à tous les éléments de $COMPSUB$ et qui sont apparus dans des cliques précédemment découvertes et qui (pour éviter les répétitions) ne doivent pas apparaître dans la clique courante. Cet ensemble permet d'éviter de trouver plusieurs fois les mêmes cliques ou d'énumérer toutes celles qui ne sont pas maximales.

Un élément de $CANDIDATES$ est supprimé de cet ensemble s'il est introduit dans $COMPSUB$, ou s'il n'est pas voisin de la dernière valeur introduite dans $COMPSUB$. Un élément disparaît de NOT si on introduit dans $COMPSUB$ un élément qui n'est pas un de ses voisins.

L'algorithme procède par extension successive de la clique courante jusqu'à ce qu'elle soit maximale. Pour augmenter la taille de la clique courante, il choisit un sommet de $CANDIDATES$ et l'ajoute à la clique courante. Si, à un moment donné, $CANDIDATES$ et NOT sont vides alors $COMPSUB$ est une clique maximale. Par contre, si un élément de NOT est connecté à tous les éléments de $CANDIDATES$ alors il est impossible de trouver une nouvelle clique maximale et la voie courante est abandonnée. En effet si un élément de NOT est relié à tous les candidats possibles, alors toute clique obtenue à l'aide des ces candidats et de la clique courante n'est pas maximale, puisque si on lui rajoute l'élément de NOT, on aura une nouvelle clique.

Cette remarque est très importante car elle permet de réaliser de nombreuses «coupes» dans l'espace de recherche. C'est l'aspect «branch-and-bound» de la méthode.

Algorithme 48 Algorithme de recherche des cliques maximales de Bron et Kerbosh.

```
RECHERCHERCLIQUESMAXIMALES(i C, K, NOT)
  /* C représente l'ensemble CANDIDATES */
  /* K correspond à COMPSUB */
  /* Γ(x) est l'ensemble des voisins de x */
  si C = ∅ alors
  |  si NOT = ∅ alors AFFICHERCLIQUEMAXIMALE(K)
  sinon
  |  x ← CHOISIRCANDIDAT(C, NOT) /* x est le candidat distingué */
  |  C ← C − {x}
  |  RECHERCHERCLIQUESMAXIMALES(C ∩ Γ(x), K ∪ {x}, NOT ∩ Γ(x))
  |  NOT ← NOT ∪ {x}
  |  tant que x n'est pas relié à tous les éléments de C faire
  |  |  y ← CANDIDATNONRELIÉ(C, x)
  |  |  C ← C − {y}
  |  |  RECHERCHERCLIQUESMAXIMALES(C ∩ Γ(y), K ∪ {y}, NOT ∩ Γ(y))
  |  |  NOT ← NOT ∪ {y}
```

L'efficacité de l'algorithme dépend de l'ordre selon lequel les sommets sont sélectionnés. Si l'on arrive le plus tôt possible dans la recherche à avoir dans NOT un élément relié à tous les sommets de $CANDIDATES$, alors on évitera une énumération coûteuse et inutile de nombreuses cliques non maximales. Supposons que chaque élément de NOT possède un compteur indiquant le nombre de candidats avec lesquels il n'est pas connecté. Si un compteur vaut 0 alors la situation que nous recherchons est atteinte. L'introduction dans NOT d'un sommet provoque la décrémentation du compteur des éléments de NOT qui ne sont pas reliés avec lui. Remarquons qu'un compteur n'est jamais décrémenté de plus de 1 à chaque fois qu'un élément est ajouté dans NOT. Distinguons un sommet x de NOT. Si nous sélectionnons successivement comme candidats tous les sommets de $CANDIDATES$ non reliés à x nous décré-

menterons à chaque fois le compteur de x et aucun autre sommet de NOT ne verra son compteur atteindre 0 plus rapidement que celui de x. Par conséquent si l'on distingue dans NOT l'élément ayant le plus petit compteur, alors nous serons assuré d'atteindre le plus rapidement possible la situation recherchée.

L'algorithme 48 résume tous les principes que nous venons d'énoncer. Au premier appel $CANDIDATES$ doit contenir tous les sommets du graphe, alors que $COMPSUB$ et NOT doivent être vides. Remarquons qu'il n'est pas nécessaire de représenter explicitement le graphe de compatibilité en mémoire. On peut se contenter de mémoriser uniquement son ensemble de sommets et de recalculer à chaque fois que l'on en a besoin l'ensemble des voisins d'un sommet.

Nous allons maintenant adapter cette algorithme afin qu'il tienne compte de l'appariement forcé de deux liaisons et pour qu'il ne cherche que des cliques maximales correspondant à des sspc connexes maximales et appariés.

Adaptation de l'algorithme de Bron et Kerbosh à Sgpccm-apparié

La prise en compte de l'appariement forcé des deux arêtes désignées se fait facilement en modifiant l'appel initial de l'algorithme de Bron et Kerbosh. Considérons que $l_1 = \{u_1, v_1\}$ et $l_2 = \{u_2, v_2\}$ sont les deux arêtes que l'on doit nécessairement apparier. Il existe seulement deux possibilités : soit u_1 est apparié avec u_2 et v_1 avec v_2, soit u_1 est apparié avec v_2 et v_1 avec u_2. Bien entendu, il faut aussi tenir compte des étiquettes des sommets et des arêtes. Pour trouver les cliques maximales impliquant ces deux possibilités, il suffit d'appeler deux fois la fonction de recherche de cliques maximales en modifiant l'initialisation des ensembles $CANDIDATES$ et $COMPSUB$. Choisissons une manière d'apparier les deux arêtes, par exemple u_1 avec u_2 et v_1 avec v_2, on définit alors les ensembles $CANDIDATES$ et $COMPSUB$ de la façon suivante (X_C désigne l'ensemble des sommets du graphe de compatibilité) :

- $CANDIDATES \leftarrow X_C \cap \Gamma((u_1, u_2)) \cap \Gamma((v_1, v_2))$.
- $COMPSUB \leftarrow \{((u_1, u_2), (v_1, v_2))\}$.

Par contre, si l'on impose que les sous-graphes partiels recherchés soient connexes, cela entraîne une modification directe de l'algorithme de Bron et Kerbosh (cf algorithme 49). Rappelons que les arêtes du sgpc maximal apparié correspondant à la

Algorithme 49 Algorithme de recherche des sgpc maximaux connexes et appariés.

```
RECHERCHERCLIQUESMAXIMALES(i F, C, K, NOT)
  /* C représente l'ensemble CANDIDATES */
  /* F représente l'ensemble STRONGCANDIDATES */
  /* K correspond à COMPSUB */
  /* Γ(x) est l'ensemble des voisins de x */
  /* Γ_fort(x) est l'ensemble des voisins forts de x */
  si F = ∅ alors
    si NOT = ∅ alors AFFICHERCLIQUEMAXIMALE(K)
  sinon
    x ← CHOISIRCANDIDATFORT(F, NOT)
    nvF ← (F ∩ Γ(x)) ∪ (C ∩ Γ_fort(x))
    C ← C − {x}
    F ← F − {x}
    RECHERCHERCLIQUESMAXIMALES(nvF, C ∩ Γ(x), K ∪ {x}, NOT ∩ Γ(x))
    NOT ← NOT ∪ {x}
    tant que x n'est pas relié à tous les éléments de C faire
      y ← CANDIDATNONRELIÉ(F, x)
      nvF ← (F ∩ Γ(y)) ∪ (C ∩ Γ_fort(y))
      F ← F − {y}
      C ← C − {y}
      RECHERCHERCLIQUESMAXIMALES(nvF, C ∩ Γ(y), K ∪ {y}, NOT ∩ Γ(y))
      NOT ← NOT ∪ {y}
```

clique sont déterminées par les arêtes fortes de la clique. Aussi pour garantir la connexité du sgpc apparié il suffit simplement de choisir, à chaque étape de l'algorithme, un sommet relié par une arête forte à au moins un sommet de la clique courante. Pour réaliser cela, nous avons introduit un nouvel ensemble de candidats : $STRONGCANDIDATES$. Un sommet est inséré dans $STRONGCANDIDATES$ s'il appartient à $CANDIDATES$ et s'il est relié par une arête forte au dernier candidat sélectionné. Un sommet est supprimé de $STRONGCANDIDATES$ s'il disparaît de $CANDIDATES$.

Fonctions de compatibilité

Nous avons précédemment mentionné les fonctions σ_G et α_G, nous allons les détailler pour notre application. Ces fonctions permettent de modéliser de nombreux problèmes en chimie. Leur détermination est très importante et doit être faite avec le plus grand soin car elles agissent directement sur le mécanisme de généralisation.

Après de nombreux essais nous avons fait les choix suivants :

$\sigma_G(x, y)$ est vrai si toutes les conditions suivantes sont vérifiées, sinon elle est fausse ($etq_S(x)$ représente l'étiquette d'un sommet x) :

- le type atomique de $etq_S(x)$ et le type atomique de $etq_S(y)$ ont un ancêtre commun;
- $etq_S(x)$ et $etq_S(y)$ ont la même fonctionnalité;
- $etq_S(x)$ et $etq_S(y)$ subissent le même effet inducteur ou l'un des deux subit un effet et l'autre est soumis à un effet inducteur indéterminé (effet-inducteur dans la hiérarchie);
- $etq_S(x)$ et $etq_S(y)$ subissent le même effet mésomère ou l'un des deux subit un effet et l'autre est soumis à un effet mésomère indéterminé (effet-mésomère dans la hiérarchie).

$\alpha_G(e, f)$ est vrai si les deux étiquettes ont le même ordre et la même aromaticité, sinon elle est fausse.

5.4.3 Introduction d'un mécanisme de généralisation des étiquettes

Généralisation des étiquettes des sommets

La généralisation des étiquettes s_1 et s_2 de deux sommets est une étiquette s_g dont les caractéristiques sont définies comme suit :

- le type de s_g est l'ancêtre commun le plus spécifique[5] du type de s_1 et de celui de s_2 dans la hiérarchie des types atomiques;
- la stéréochimie de s_g est l'ancêtre commun le plus spécifique [5] de la stéréochimie de s_1 et de celle de s_2 dans la hiérarchie des stéréochimies atomiques;
- la charge de s_g est l'ancêtre commun le plus spécifique [5] de la charge de s_1 et de celle de s_2 dans la hiérarchie des charges atomiques;
- la fonctionnalité de s_g est celle de s_1;
- l'effet inducteur de s_g est l'ancêtre commun le plus spécifique [5] de l'effet inducteur de s_1 et de celui de s_2 dans la hiérarchie des effets inducteurs;
- l'effet mésomère de s_g est l'ancêtre commun le plus spécifique [5] de l'effet mésomère de s_1 et de celui de s_2 dans la hiérarchie des effets mésomères.

Cette définition est cohérente avec celle de la fonction σ_G.

Généralisation des étiquettes des arêtes

La généralisation des étiquettes a_1 et a_2 de deux arêtes est une étiquette a_g définie par :

- l'aromaticité de a_g est celle de a_1;
- l'ordre de a_g est celui de a_1;
- la stéréochimie de a_g est l'ancêtre commun le plus spécifique [5] de la stéréochimie de a_1 et de celle de a_2 dans la hiérarchie des stéréochimies des liaisons;

Cette fonction est cohérente avec celle de la fonction α_G.

5. La structure de la hiérarchie nous assure l'existence d'au plus un ancêtre le plus spécifique.

5.4.4 Algorithme de recherche des généralisations de deux exemples

Soient deux exemples $e_1 = (G_1, l_1)$ et $e_2 = (G_2, l_2)$, les généralisations entre e_1 et e_2 sont obtenues à partir des sgpc connexes maximales appariés de G_1 et G_2 appariant l_1 avec l_2 et tenant compte des étiquettes, et des généralisations des étiquettes des sommets et des arêtes de G_1 et G_2. Plus précisément, on calcule les généralisations entre e_1 et e_2 à l'aide de l'algorithme suivant :

1. On détermine la liste LGC des sgpc connexes maximales appariés de G_1 et G_2 appariant l_1 avec l_2 et tenant compte des étiquettes.

2. Pour chaque élément (S, m_1, m_2) de la liste LGC on étiquette les sommets et les arêtes de S de la façon suivante :

   • un sommet x de S est étiqueté par la généralisation des étiquettes des sommets $m_1(x)$ de G_1 et $m_2(x)$ de G_2;

   • une arête $\{x, y\}$ de S est étiquetée par la généralisation des étiquettes des arêtes $\{m_1(x), m_1(y)\}$ de G_1 et $\{m_2(x), m_2(y)\}$ de G_2.

   On mémorise ensuite le couple (S, l_S) dans une liste LG. l_S est l'arête $\{x, y\}$ de S telle que $\{m_1(x), m_1(y)\} = l_1$ et $\{m_2(x), m_2(y)\} = l_2$, autrement dit l_S est l'arête de S ayant «apparié» l_1 avec l_2.

3. On ne conserve dans LG que les généralisations les plus spécifiques.

5.5 Relation d'appariement

La relation d'appariement ($\mathcal{A}$) permet de spécifier quelles descriptions sont les généralisations de tels ou tels exemples. Nous avons vu, dans le chapitre 3, que $\mathcal{A}$ induit un ordre partiel sur le langage de généralisation (relation $\succeq$). Il est possible de mettre en oeuvre ces deux relations en se dotant d'un seul mécanisme permettant de comparer deux descriptions. Ces deux descriptions peuvent représenter soit deux généralisations (cf. section précédente), soit une généralisation et un exemple. Pour une description d et une généralisation g, on aura $d \succeq g$ si et seulement si l'une des deux conditions suivantes est vérifiée :

- d est un exemple et $\mathcal{A}(d, g)$ est vrai;
- d est une généralisation et $d \geq g$.

Pour notre problème, nous avons la définition suivante:

Définition 58 *Soient deux descriptions (S, l_S) et (G, l_G) avec $G = (X_G, E_G, etq_S, etq_A)$ et $l_G = \{l_{G_1}, l_{G_2}\}$, σ une fonction de compatibilité entre étiquettes de sommets et α une fonction de compatibilité entre étiquettes d'arêtes, alors*
$(S, l_S) \succeq (G, l_G)$ si et seulement s'il existe un morphisme injectif m de G dans S vérifiant:

- *$\{m(l_{G_1}), m(l_{G_2})\} = l_S$ (appariement de l_G avec l_S);*
- *$\forall g \in X_G : \sigma(g, m(g))$ est vrai (prise en compte des étiquettes des sommets);*
- *$\forall \{g_1, g_2\} \in E_G : \alpha(\{g_1, g_2\}, \{m(g_1), m(g_2)\})$ est vrai (prise en compte des étiquettes des arêtes);*

Étant donné deux éléments e_1 et e_2 appartenant à une hiérarchie quelconque d'une étiquette, pour simplifier notre exposé, nous dirons que e_1 est reconnu par e_2 si e_2 est égal à e_1 ou bien si e_2 est un ancêtre de e_1.

5.5.1 Compatibilité entre étiquettes de sommets

Après des nombreux essais nous avons décidé qu'une étiquette s_1 d'un sommet x_1 serait plus générale qu'une étiquette s_2 d'un sommet x_2, autrement dit que $\sigma(x_1, x_2)$ est vrai, si et seulement si les conditions suivantes sont simultanément vérifiées:

- le type atomique de s_1 est reconnu par celui de s_2.
- la stéréochimie de s_1 est reconnue par celle de s_2.
- la charge de s_1 est reconnue par celle de s_2.
- s_1 et s_2 ont la même fonctionnalité.
- l'effet inducteur de s_1 est reconnu par l'effet inducteur de s_2;
- l'effet mésomère de s_1 est reconnu par l'effet mésomère de s_2.

Toutes les conditions énoncées ci-dessus sont cohérentes avec la généralisation des étiquettes des sommets.

5.5.2 Compatibilité entre étiquette des arêtes

L'étiquette a_1 d'une arête e_1 est plus générale que l'étiquette a_2 d'une arête e_1 ($\alpha(e_1, e_2)$ est vrai) si et seulement si les conditions suivantes sont simultanément vérifiées :

- a_1 et a_2 ont la même aromaticité.
- a_1 et a_2 ont le même ordre.
- la stéréochimie de a_1 est reconnue par la stéréochimie de a_2.

Toutes les conditions énoncées ci-dessus sont cohérentes avec la généralisation des étiquettes des sommets.

5.5.3 Modélisation de la relation d'appariement par un réseau de contraintes

Nous avons consacré la partie I de cette thèse au problème de l'isomorphisme de sous-graphe partiel. Pour résoudre ce problème, nous avons proposé d'utiliser des réseaux de contraintes. Ici, nous devons introduire l'appariement entre les deux arêtes des descriptions. Supposons que l'on recherche si la description (G, l_G), avec $G = (X_G, E_G)$ et $l_G = \{l_{G_1}, l_{G_2}\}$, est plus générale que la description (S, l_S), avec $S = (X_S, E_S)$ et $l_S = \{l_{S_1}, l_{S_2}\}$,en tenant compte des étiquettes grâce aux fonctions $\sigma_>$ et $\alpha_>$ données par l'utilisateur (ce peut être les fonctions α et σ précédemment exposées). Répondre à cette question est équivalent à rechercher une solution dans le réseau de contraintes suivant :

$\mathcal{R} = (X_G, \mathcal{D}, \mathcal{C})$ où :

- X_G est l'ensemble des sommets de G.
- Le domaine de chaque variable $g \in X_G - \{l_{G_1}, l_{G_2}\}$ est constitué par l'ensemble des sommets de $S - \{l_{S_1}, l_{S_2}\}$ dont le degré est supérieur ou égal à celui de g.
- Le domaine de l_{G_1} (respectivement l_{G_2}) est constitué par les variables de $\{l_{S_1}, l_{S_2}\}$ dont le degré est supérieur ou égal à celui de l_{G_1} (respectivement l_{G_2})

- A chaque variable correspond une contrainte unaire définie par la fonction σ : $C_g(s) = \sigma(g, s)$.
- A chaque arête $\{i, j\} \neq l_G$ de G correspond une contrainte binaire C_{ij} telle que $C_{ij}(a, b)$ est vrai si et seulement si $\{a, b\} \in E_S$ et $\{a, b\} \neq l_S$ et $\alpha(\{i, j\}, \{a, b\})$ est vrai.
- Il existe une contrainte $C_{l_{G_1} l_{G_2}}$ définie par :
$$C_{l_{G_1} l_{G_2}}(l_{S_1}, l_{S_2}) = C_{l_{G_1} l_{G_2}}(l_{S_2}, l_{S_1}) = \alpha(\{l_{G_1}, l_{G_2}\}, \{l_{S_1}, l_{S_2}\})$$
- Il existe une contrainte de différence impliquant tous les sommets de X_G : TOUSDIFFÉRENTS(X_G).

Nous aurions aussi pu modéliser les deux graphes par des graphes bipartis n'ayant plus d'arêtes étiquetées comme ceci est expliqué dans le chapitre 8 de la partie I.

5.6 Conclusion

Dans ce chapitre, nous avons présenté une modélisation possible du problème de l'apprentissage des liaisons stratégiques en chimie organique. Nous avons expliqué comment nous avons construit l'ensemble d'apprentissage en montrant les particularités, peu courantes, de notre problème : des classes déséquilibrées, une classe sûre et une autre incertaine. Puis, nous avons montré comment les connaissances du domaine d'application ont été prises en compte. Notamment, nous avons introduit les effets électroniques dans la modélisation. Ensuite, nous avons détaillé la mise en oeuvre du mécanisme de généralisation. Nous avons utilisé un algorithme de recherche de cliques maximales pour l'implémenter. Enfin, nous avons présenté la relation d'appariement qui nous a ramené, momentanément, aux problèmes de satisfaction de contraintes. La modélisation que nous avons proposée nous semble extrêmement souple puisqu'elle est essentiellement basée sur 4 fonctions de compatibilités reposant sur des hiérarchies données de manière déclarative.

Maintenant, nous sommes en mesure d'étudier les résultats fournis par le système. C'est l'objet du chapitre suivant.

Chapitre 6

Résultats

Ce chapitre présente et commente les résultats que nous avons obtenus pour le problème de la détermination des liaisons stratégiques en synthèse organique.

Nous avons employé la méthode CNN décrite dans le chapitre 4 et suivi la modélisation du problème présentée dans le chapitre 5.

L'ensemble d'apprentissage que nous avons considéré contient 694 liaisons avec leur environnement structurel. 91 exemples sont positifs et 603 négatifs. Les exemples ont été obtenus à partir des 75 produits formés par des réactions et donnés en annexe.

Nous rappelons que pour régler le système d'apprentissage, l'utilisateur doit seulement introduire deux valeurs: le seuil de chaque classe, ainsi que le seuil α, fixé ici à 5%. Toute règle ayant un taux d'erreur supérieur à ce seuil sera rejetée. Nous avons choisi un seuil égal à 0,76 pour la classe des exemples positifs et un seuil égal à 0 pour la classe des exemples négatifs. Ce dernier signifie que l'on ne veut pas apprendre de règles en faveur de la classe des exemples négatifs, ce qui est justifié par le fait que les exemples classés négatifs, pourraient être positifs si l'on disposait de plus de données (cf chapitre 5). Notons que pour faire apparaître une règle en faveur de la classe des négatifs, il faut porter cette valeur à 0,25. Cela fait apparaître de très nombreuses règles et le système se trompe systématiquement pour les exemples positifs. La valeur très élevée du seuil de la classe des exemples positifs mérite une explication. Ce seuil a été optimisé en fonction des résultats obtenus. La valeur retenue montre que pour un problème comme le nôtre, c'est-à-dire avec des classes très déséquilibrées, on est prêt à retenir des règles pouvant avoir une borne supérieure du taux d'erreur très

élevée. La valeur 0,76 permet d'autoriser une règle vérifiée 4 fois et contredite 1 fois (sa borne supérieure du taux d'erreur vaut 0,752 avec $\alpha = 0,05$). Une telle valeur permet donc d'accepter des règles qui sont peu vérifiées.

D'après les seuils que nous avons choisis, tout exemple qui n'est pas classé comme étant positif est considéré comme négatif. Le système ne s'abstient donc jamais.

Pour valider nos résultats nous avons utilisé la méthode du «*jack-knife*» (cf. paragraphe 3.1).

Avant de détailler les résultats nous pouvons donner quelques informations. Le nombre de ressemblances différentes engendrées par la comparaison deux à deux des ressemblances est d'environ 12 000. Le nombre moyen de ressemblances entre deux exemples est donc 0,05.

72 heures de temps cpu (avec une SPARC 2) ont été nécessaires pour mener à bien l'apprentissage[1].

6.1 Reclassement des exemples

Le système reclasse correctement les exemples positifs dans 80% des cas (73 sur 91). Pour un produit formé, il retrouve au moins une des liaisons créées par une synthèse dans 80 % des cas (60 sur 75). Le système ne classe positif que 9 exemples négatifs. Pour la classe des exemples négatifs, il ne fait donc que 1,5 % d'erreur.

Au total, le système reclasse donc correctement 96 % des exemples (667 sur 694). Ce résultat est bien meilleur que celui que nous avions obtenus avec une méthode plus proche de ANNA [Régin, 1995].

Les résultats obtenus, en terme de classement, sont donc excellents.

6.2 Règles produites

Les figures 6.1, 6.2, 6.3 et 6.4 montrent les règles apprises par le système. Dans toutes ces figures, la liaison stratégique est représentée en gras. Il y en a 24. Ce nombre

1. Si le programme était entièrement réécrit, on pourrait obtenir un temps de l'ordre d'une dizaine d'heures.

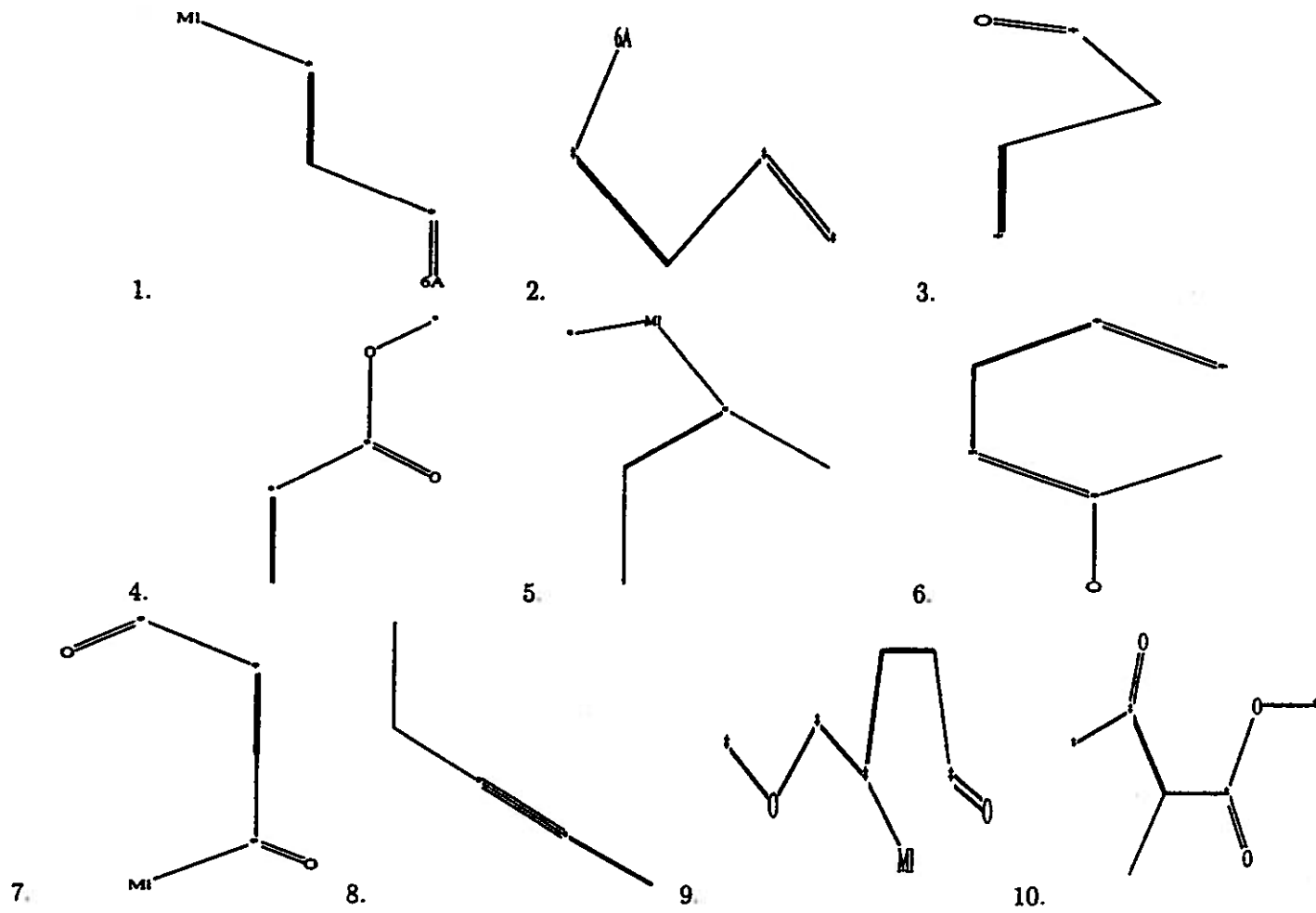


FIG. 6.1 - *Environnements structuraux en faveur du caractère stratégique d'une liaison entre un carbone non fonctionnel et un carbone fonctionnel.*

est un peu élevé, mais bien souvent ces règles ont une pertinence chimique :

- les règles 1 et 10 correspondent à une aldolisation;
- la règle 2 à une réaction sur les cétones;
- les règles 3, 9 et 15 à une addition de Michael;
- les règles 4 et 7 à l'alkylation d'un ester;
- les règles 11, 12, 16, 17, 18 et 19 à l'alkylation d'une cétone.

Les autres règles, c'est-à-dire les règles 5, 6, 8, 13, 14, 20, 21, 22, 23 et 24 sont trop abstraites. Donc 14 règles représentent une certaine connaissance pour un expert de la synthèse organique. Et le système a retrouvé 5 grandes réactions chimiques. Ce résultat est donc plus que satisfaisant.

On remarque cependant qu'il pourrait être intéressant de combiner les règles pour essayer de réduire leur nombre. Ainsi la réaction d'alkylation d'une cétone est représentée par 6 règles, ce qui est trop, d'autant plus que ces règles sont proches.

6.3 Un inconvénient

Après une étude approfondie de ces résultats nous avons mis en évidence un problème dû au domaine d'application. La figure 6.5 en est une illustration. La règle proposée dans cette figure a une réelle pertinence chimique pour caractériser une liaison stratégique. Les deux liaisons présentées sont deux exemples positifs de l'ensemble d'apprentissage qui sont classés comme étant négatif par le système. Une telle règle ne peut pas, en l'état actuel du système être apprise. Pour s'en convaincre, nous allons montrer qu'elle caractérise, pour le système, une liaison non stratégique, à cause de sa symétrie.

Par rapport à la molécule de gauche, si l'on apparie a sur a', la règle de la figure 6.5 reconnaît un exemple positif. Or, comme cette règle est symétrique, elle reconnaît aussi automatiquement un exemple négatif (appariement de a sur b'). Par conséquent le nombre d'exemples positifs qu'elle reconnaît ne peut pas être supérieur au nombre d'exemples négatifs qu'elle reconnaît. De plus, avec la molécule de droite, on se rend

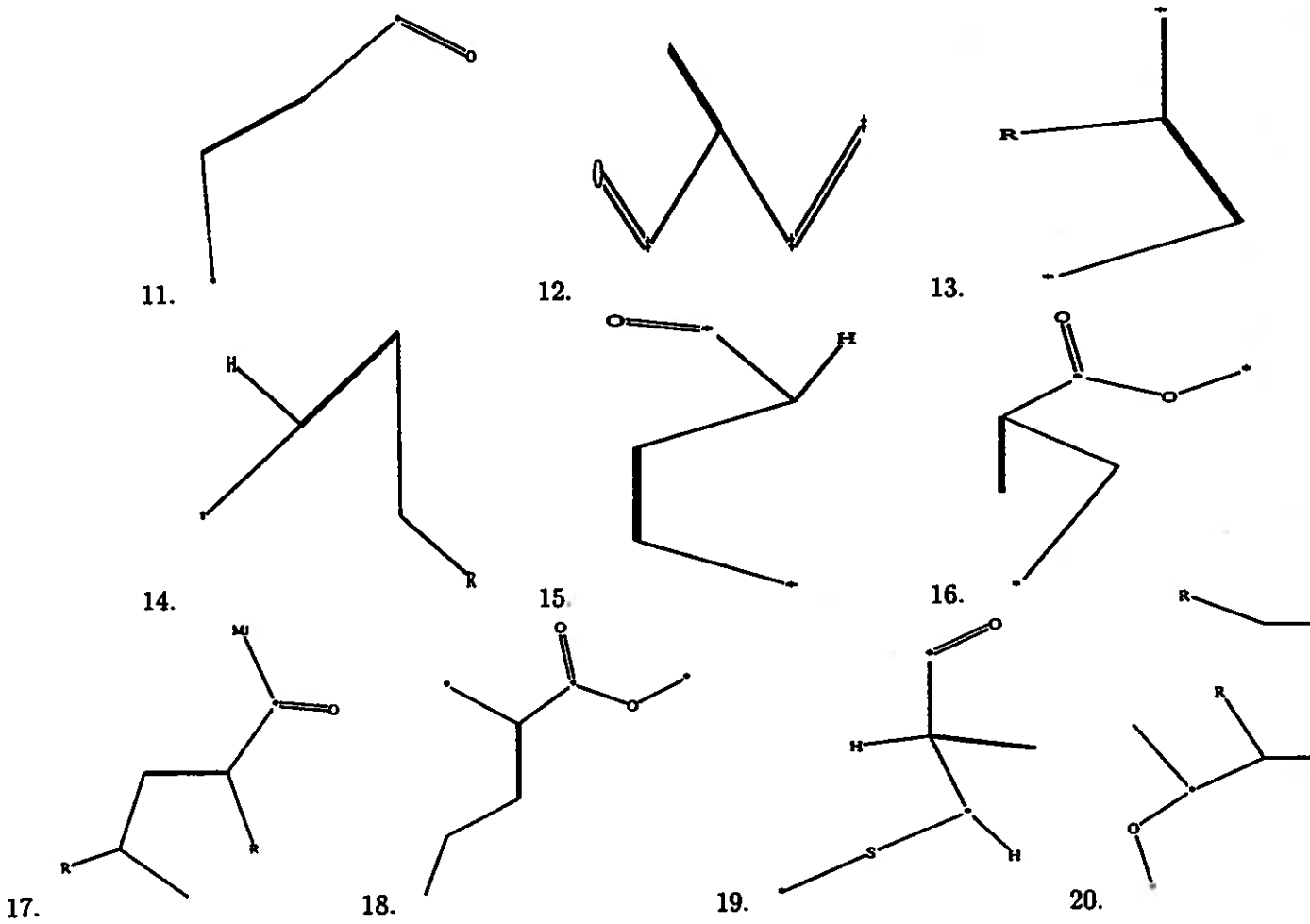


FIG. 6.2 - *Environnements structuraux en faveur du caractère stratégique d'une liaison entre deux carbones non fonctionnels.*

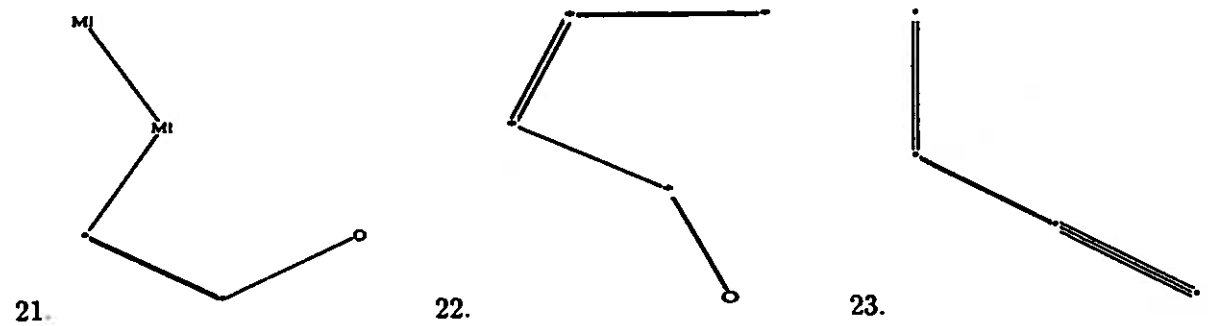
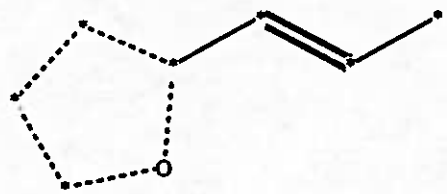
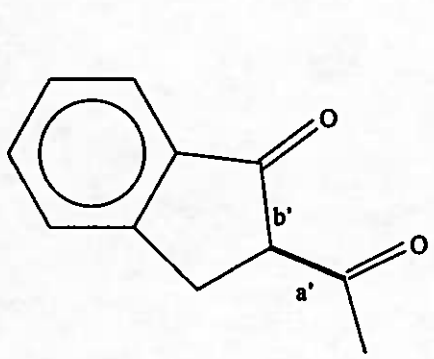


FIG. 6.3 - *Environnements structuraux en faveur du caractère stratégique d'une liaison entre deux carbones fonctionnels.*

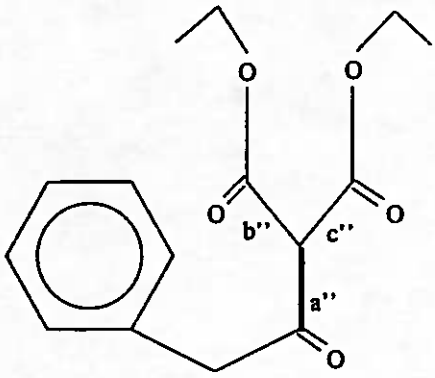


24.

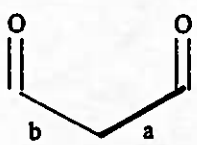
FIG. 6.4 - *Environnement structural en faveur du caractère stratégique d'une double liaison entre deux carbones.*



Liaison 1



Liaison 2



Règle

FIG. 6.5 - *Problèmes dus à la symétrie des règles.*

compte que la règle va reconnaître plus d'exemples négatifs que d'exemples positifs (appariement de a sur b'' et c'' pour les négatifs et de a sur a'' pour les positifs). Cette règle ne peut donc pas être en faveur du caractère stratégique de la liaison.

Nous n'avons pas trouvé de réponse satisfaisante à ce problème pour l'instant.

6.4 Conclusion

Le système obtient d'excellents résultats pour reclasser les exemples de l'ensemble d'apprentissage. Les règles qu'il fournit ont une pertinence chimique certaine.

Il faudrait maintenant étudier son comportement avec un ensemble d'apprentissage beaucoup plus important (à condition que les temps de calcul ne soit pas trop importants), et trouver une réponse au problème de la symétrie des règles.

Chapitre 7

Conclusion

Dans cette partie nous avons présenté une méthode d'apprentissage pour déterminer les liaisons stratégiques des molécules en synthèse organique. Cette méthode, nommée CNN, a de très bonnes performances non seulement pour des problèmes tests mais aussi pour le problème mentionné ci-dessus. En effet, le système que nous avons développé reclasse correctement 96% des liaisons de l'ensemble d'apprentissage et fournit des règles ayant une réelle pertinence chimique.

Par ailleurs nous avons montré l'intérêt de l'introduction des effets électroniques auxquels les atomes sont soumis dans la modélisation des molécules.

Nous avons également proposé une approche intéressante pour rechercher les sous-graphes partiels communs maximaux et connexes de deux graphes. Cette méthode s'appuie sur l'algorithme performant de recherche de cliques maximales de Bron et Kerbosh.

Plusieurs perspectives se dégagent de ce travail.

Il faudrait essayer d'utiliser ce système sur un plus grand nombre d'exemples, voir même sur une base de données toute entière à condition, bien sûr, que les temps de calcul ne soient pas trop longs.

Il pourrait être également intéressant de tester les capacités de notre système pour d'autres problèmes réels.

Enfin, on doit pouvoir utiliser l'algorithme de Bron et Kerbosh pour résoudre

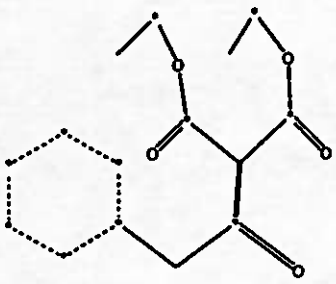
certains problèmes de satisfaction partielle de contraintes.

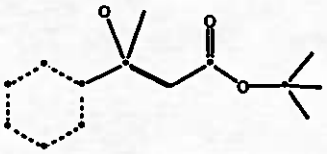
Annexe A

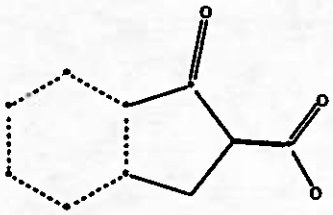
Ensemble d'apprentissage

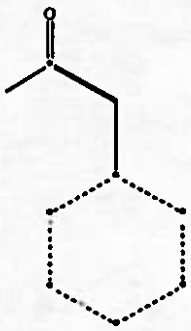
L'ensemble d'apprentissage est obtenu à partir des 75 molécules des pages suivantes. Toutes les liaisons entre deux carbones qui ne sont ni aromatiques, ni triples sont des exemples. Leur environnement est constitué par le reste de la molécule. Les exemples positifs sont la (ou les) liaison(s) créée(s) par une synthèse et sont dessinées en gras.

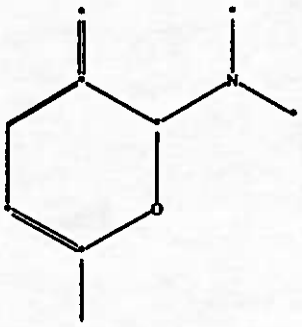
Quelques conventions ont été utilisées: les liaisons en pointillés représentent des liaisons aromatiques; les carbones ne sont pas mentionnés sauf s'ils sont fonctionnels et dans ce cas une étoile les représente.

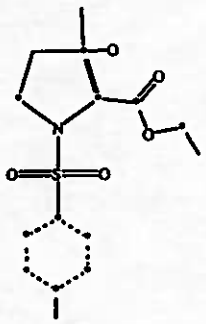


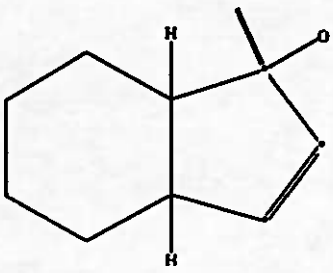


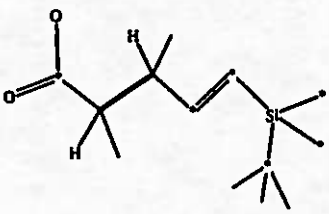


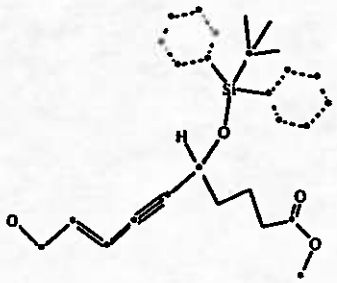


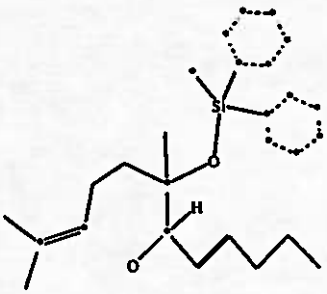


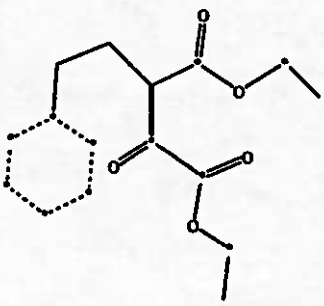


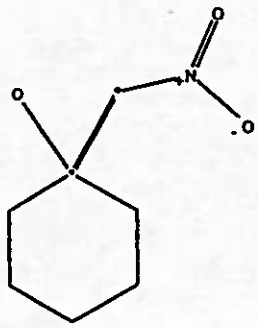


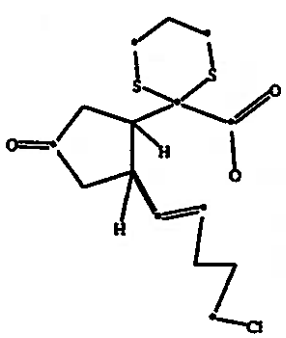


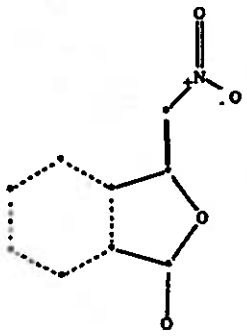


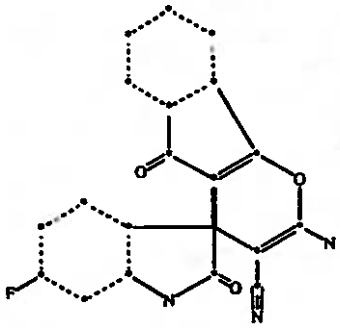


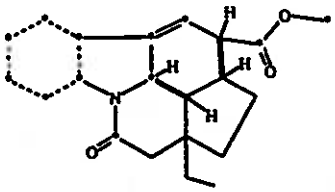


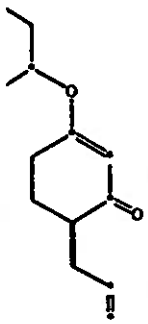


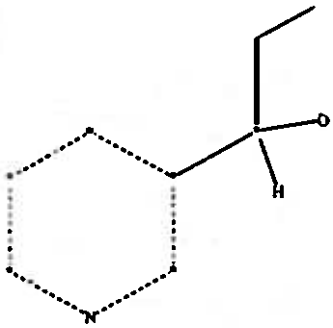


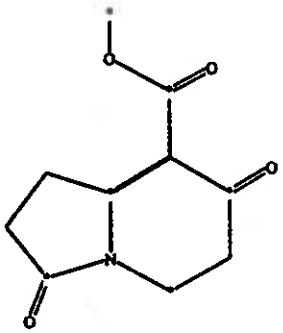


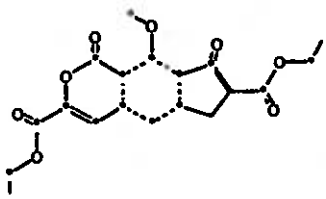


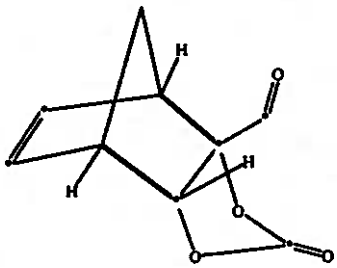


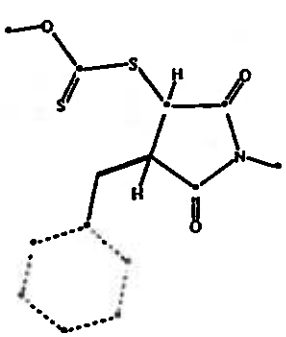


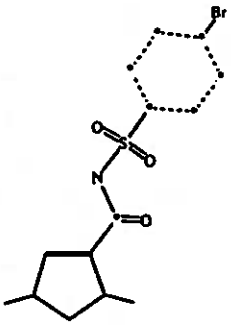


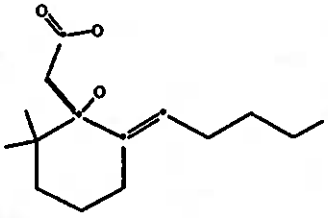


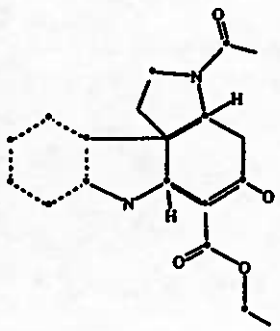
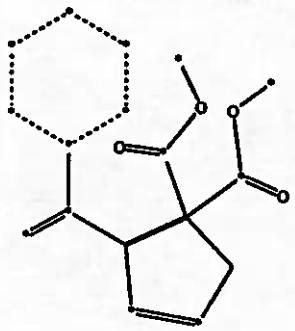
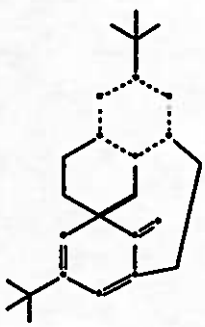


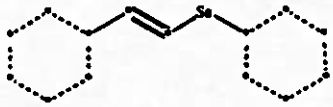
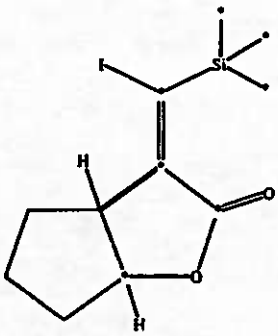
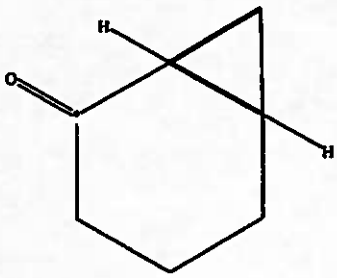
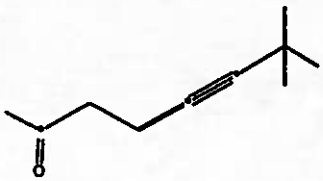
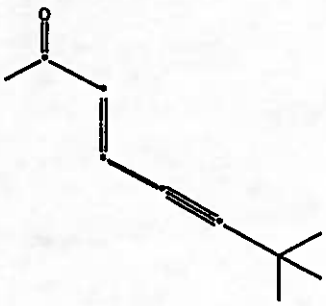
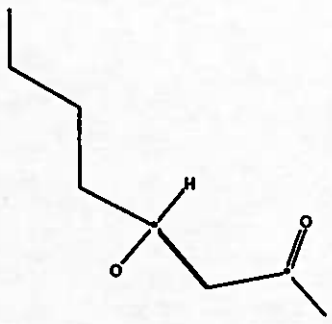
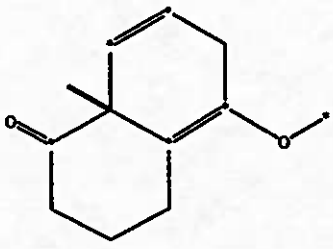
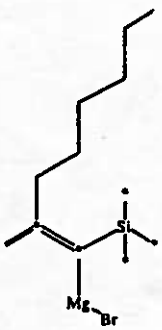


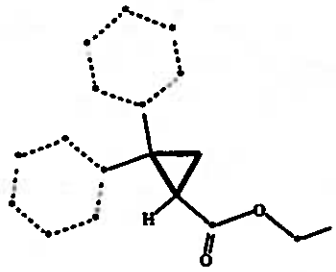


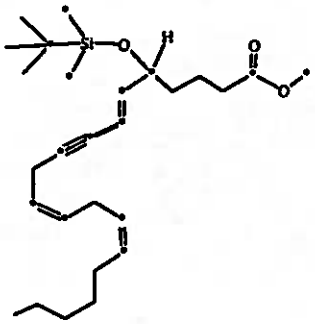


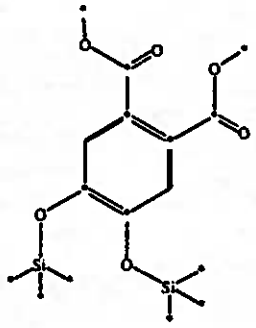


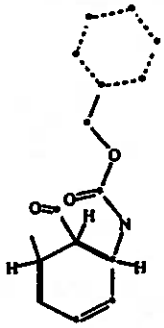




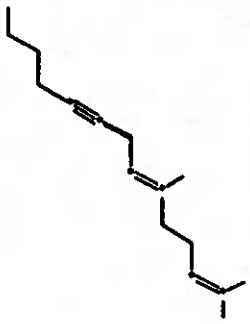









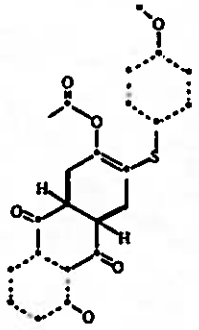


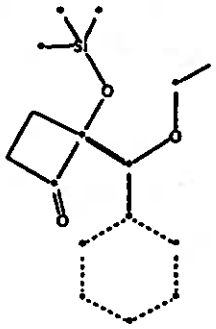


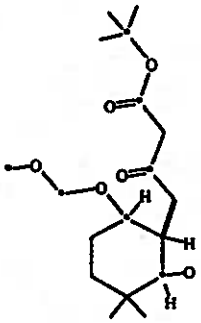


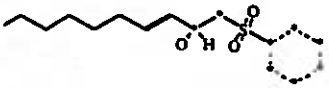


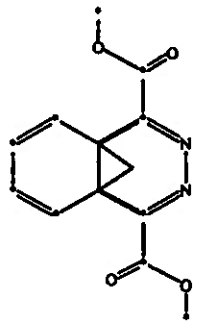


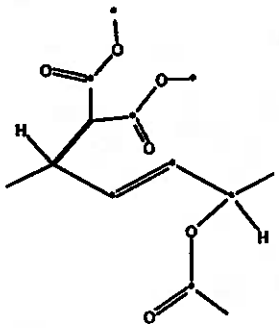


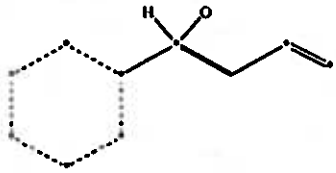


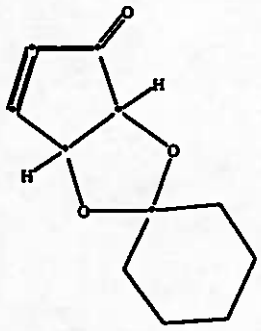
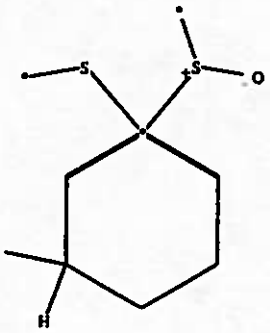
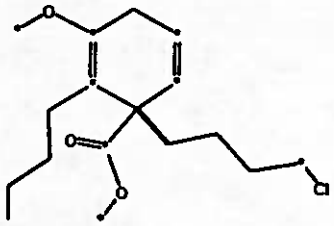
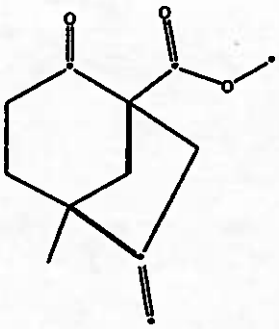
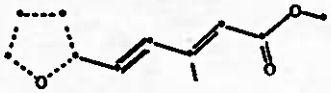
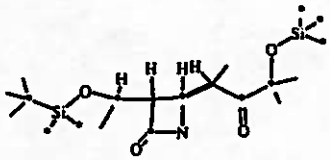
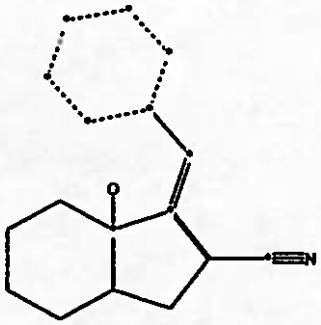
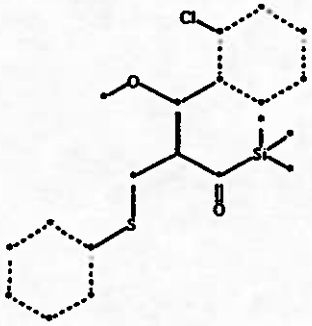
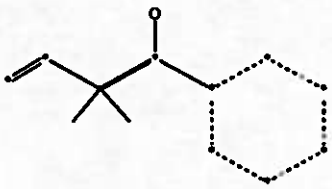
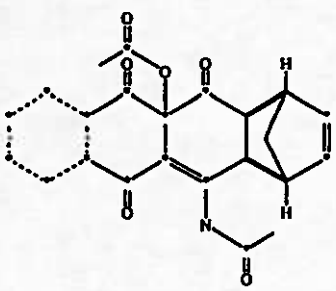
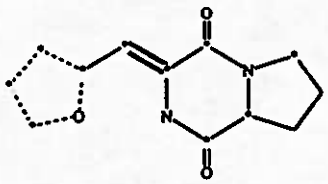
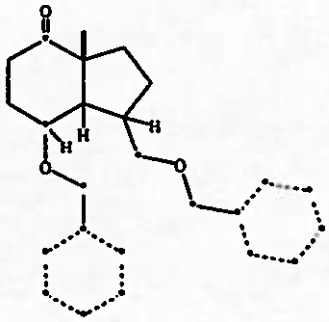


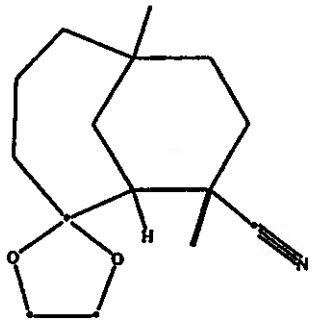
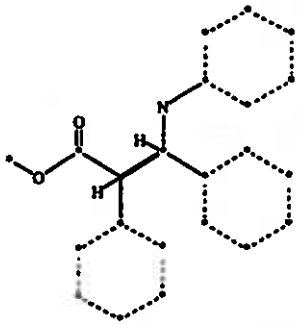
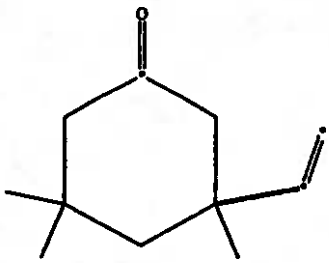
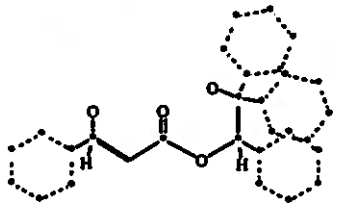
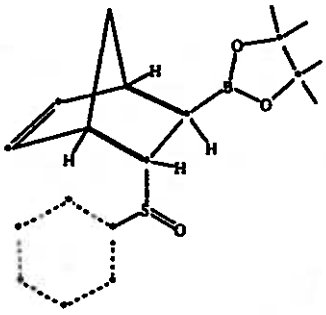
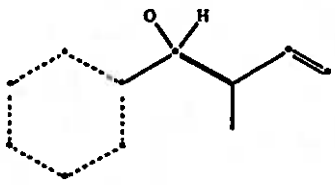
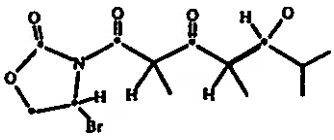
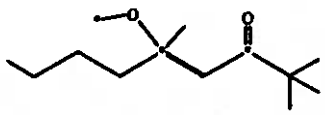
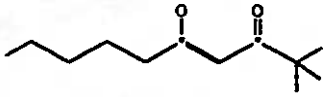
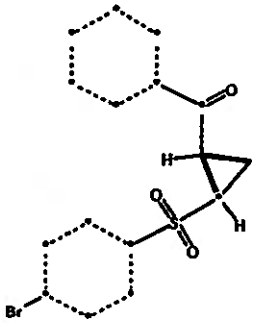
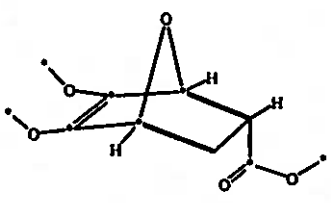
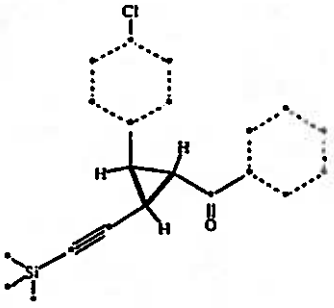


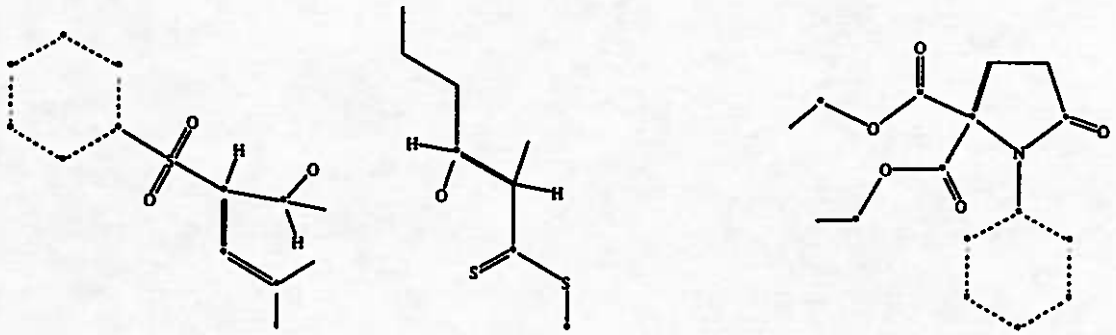








Annexe B

Le système RESYN

APPLICATION

RÉSYN: Objets et classification en chimie organique

Philippe Vismara, Jean-Charles Régin, Joël Quinqueton*, Claude Laurenço, Michel Py*****

LIRMM,
161 rue Ada,
F-34392 Montpellier Cedex 5 France

** and INRIA, BP 105, 78153 Le Chesnay Cedex*
*** and CCIPE, Montpellier.*
**** now at SEXTANT, Paris.*

résumé : Cet article présente le système RÉSYN, d'aide à la synthèse en chimie organique industrielle. C'est le résultat de plusieurs années de collaboration entre chimistes et informaticiens, depuis la première maquette du système jusqu'à la plateforme d'expérimentation actuelle. Dans la structuration de la connaissance, nous avons suivi les grandes lignes de la méthodologie KADS, plus précisément 3 couches du modèle d'expertise de KADS. Après avoir présenté ce qu'est la synthèse en chimie organique, nous présenterons les bases du système. Puis nous décrirons les différents mécanismes d'inférence, pour terminer par une spécification des tâches réalisées.

abstract : This paper presents the RÉSYN system, whose goal is computer aided synthesis in organic chemistry. It is the result of a several years collaboration between chemists and computer scientists, from the first prototype of the system to the recent experimental testbed. The knowledge is structured according the KADS model, more precisely according the 3 layers of KADS expertise model. After a short presentation of what is organic chemistry, we present the basis of the system. Then we present the various inference mechanisms, and a specification of the tasks performed by the system. An example is used to illustrate the description.

mots-clés : KADS, Classification, appariement, perception, cycles, graphes, chimie organique

key words :KADS, Classification, pattern matching, perception, cycles, graphs, organic chemistry

1. Introduction

La synthèse chimique est le processus selon lequel une molécule donnée est obtenue à partir de molécules plus simples, par l'application de réactions chimiques. Si l'on considère que cette activité consiste pour un chimiste à manipuler des objets (les molécules) avec des méthodes (les réactions), alors elle apparaît assez naturellement comme une forme de programmation par objets. On y retrouve les notions habituelles de ce type de programmation (classe, instance, héritage, exception), mais sous une forme spécifique.

L'intérêt de réaliser un système d'aide à la résolution de problèmes dans ce domaine est donc double : d'une part, modéliser la connaissance de la chimie organique, sous toutes ses facettes, et, d'autre part, enrichir les concepts de la programmation par objets de ceux issus du domaine de la chimie. L'histoire des sciences nous enseigne en effet que ce type de confrontation est toujours intéressante, comme en témoigne l'exemple de la théorie des graphes, au développement de laquelle les chimistes ont apporté une contribution significative.

Cet article présente le résultat de plusieurs années de travail, depuis la première maquette du système [QUI 92] jusqu'à la plateforme d'expérimentation actuelle. Dans la structuration de la connaissance, nous avons suivi les grandes lignes de la méthodologie KADS, plus précisément 3 couches du modèle d'expertise de KADS [SCH 93].

Après avoir présenté ce qu'est la synthèse en chimie organique, nous présenterons les modèles de base du système. Puis nous décrirons les différents mécanismes d'inférence, pour terminer par une spécification des tâches réalisées.

2. La synthèse en Chimie Organique

Pour bien comprendre l'intérêt de ce travail, une présentation des principes et mécanismes de la synthèse organique est nécessaire.

2.1. Introduction

La synthèse chimique consiste à préparer une molécule donnée à partir de ses éléments constitutifs ou de molécules disponibles plus simples, au moyen de réactions chimiques. C'est l'une des activités fondamentales du chimiste. Elle lui permet d'obtenir des substances que la nature ne produit pas ou qu'elle ne produit qu'en quantités insuffisantes pour satisfaire les besoins. Elle lui sert aussi dans le développement et la validation de théories ou encore pour prouver l'exactitude de structures proposées par l'analyse.

En chimie organique, la synthèse d'un composé complexe, naturel ou conçu pour présenter des propriétés particulières, est un problème très difficile qui ne sera résolu qu'en enchaînant logiquement de nombreuses réactions [COR 89]. La réalisation d'une telle synthèse peut faire l'objet du travail d'une équipe de chimistes pendant plusieurs années, voire de plusieurs équipes concurrentes lorsque l'intérêt, la rareté et la complexité de la molécule le justifient. Ainsi, depuis une vingtaine d'années, plus de 30 équipes ont travaillé à la synthèse du Taxol (figure 1), substance initialement extraite de l'If du Pacifique et qui possède des propriétés anticancéreuses

remarquables. Sa première synthèse totale, laquelle compte plus de 30 étapes, n'a été achevée qu'en 1993 [HOL 94a, HOL 94b].

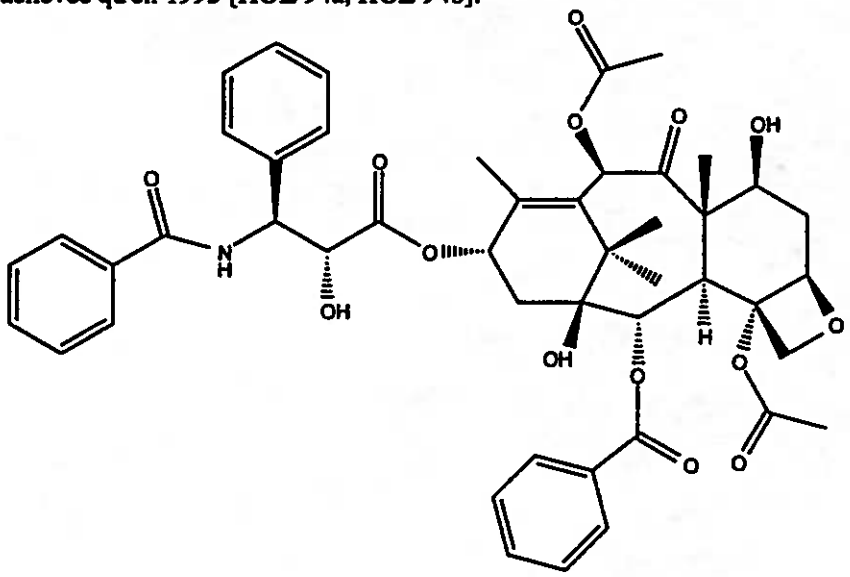


figure 1. *formule structurale*[1] *du Taxol*

2.2. Résolution d'un problème de synthèse

Une molécule organique est un objet chimique structuré tridimensionnel qui est composé d'atomes joints par des liaisons. D'un point de vue très réducteur, cet objet comporte un squelette et une fonctionnalité. Le squelette, constitué principalement d'enchaînements - linéaires ou cycliques - de liaisons reliant des atomes de carbone, définit l'architecture moléculaire. La fonctionnalité, localisée sur des liaisons multiples et des liaisons incidentes à des atomes de type différent du carbone, détermine les propriétés chimiques de la molécule. De ce point de vue, on dira que la synthèse vise à construire le squelette de la molécule en s'appuyant sur la fonctionnalité.

Pour résoudre un problème de synthèse, il faut découvrir au moins un chemin de synthèse menant d'un état initial, constitué d'un ensemble de produits de départ, à un état final, contenant la molécule souhaitée dite molécule cible. Ce chemin passe par une suite d'états intermédiaires au cours desquels la structure moléculaire se construit

[1] Une telle formule peut être vue comme un graphe représentant une structure moléculaire. Les sommets et les arêtes de ce graphe figurent respectivement les atomes et les liaisons et sont étiquetés par des symboles indiquant leur type:

O = oxygène, N = azote, ..., / = liaison simple, // = liaison double, ◄ = liaison simple située au-dessus du plan moyen, ... Pour être plus lisible, la formule ne laisse généralement pas apparaître les labels symbolisant les atomes de carbone (C) ni les atomes d'hydrogène (H) liés à des atomes de carbone.

progressivement. La transition entre les états est assurée par des méthodes de synthèse, c'est-à-dire des réactions ou des suites de réactions susceptibles d'atteindre un certain objectif (par exemple : création d'une liaison d'un type donné ou d'un motif structural cyclique, établissement ou modification d'un groupement fonctionnel, etc.). Ces méthodes ont un certain caractère de généralité puisqu'elles s'appliquent à la classe des molécules qui contiennent l'objectif correspondant.

La complexité d'un tel problème résulte tant de la faiblesse de la théorie de la chimie organique, laquelle est plus apte à expliquer les résultats d'une réaction qu'à les prédire, que de la taille considérable de son espace d'états. Par exemple, pour faire la synthèse totale *stricto sensu* du Taxol, à partir de ses éléments constitutifs : carbone, hydrogène, oxygène et azote, il faudrait créer successivement les 119 liaisons de cette molécule, ce qui peut être envisagé de plus de 10^{196} façons différentes. Ceci explique que la recherche d'une solution se fasse sous le contrôle de règles heuristiques, appelées stratégies de synthèse, et que le raisonnement du chimiste soit principalement de nature analogique. La masse des connaissances chimiques sur lesquelles on peut raisonner est très grande : plus de 13 millions de molécules, des millions de réactions particulières et des dizaines de milliers de types de réactions sont actuellement connus. Un autre facteur contribuant à la complexité d'un problème de synthèse est la variabilité des critères de jugement d'une solution en fonction des motivations. Par exemple, une solution peut être bonne du point de vue académique, car innovante, mais sans intérêt industriel, parce que trop coûteuse.

En fait, la résolution d'un problème de synthèse se déroule en 2 phases. La première est une étape de planification au cours de laquelle on élabore des hypothèses pour établir un plan de synthèse. La seconde est une étape d'expérimentation dans laquelle on essaie de réaliser ce plan. L'étape déterminante de ce processus est évidemment la première car, quelles que soient ses qualités d'expérimentateur, un chimiste ne réalisera jamais une bonne synthèse en suivant un mauvais plan.

L'élaboration d'un plan de synthèse repose sur une analyse détaillée des caractéristiques de la molécule cible, effectuée principalement à partir de sa formule structurale, et comporte 4 niveaux.

Le premier est politique, on y définit les objectifs globaux de la synthèse tels que la recherche ou non de l'innovation, du moindre coût, l'emploi de certaines réactions, etc.

Le deuxième est stratégique, on y définit et ordonne les objectifs structuraux à atteindre tels que la construction d'un système polycyclique, d'une chaîne, d'un stéréogène, etc.

Le troisième est tactique, on y choisit les méthodes de synthèse convenables pour atteindre les objectifs fixés. Ces méthodes doivent être non seulement valides, c'est-à-dire susceptibles d'atteindre ces objectifs, mais également pertinentes, c'est-à-dire ne pas être en compétition avec d'autres réactions[2], pour conduire essentiellement au produit attendu.

Le dernier est opérationnel, on y précise les conditions dans lesquelles seront effectuées les réactions (solvants, catalyseurs, température, etc.).

---

[2] Fréquemment, une molécule possède plusieurs sites réactionnels susceptibles de réagir avec un même réactif. Dans ce cas, différentes réactions se produisent concurremment pour conduire à des mélanges de produits dont les proportions dépendent de multiples facteurs. Une partie de l'art de la synthèse réside dans la capacité du chimiste à favoriser, par divers moyens, la réaction menant au produit recherché.

La cohérence du plan doit être contrôlée et maintenue à chacun de ces niveaux. Ceux-ci sont hiérarchisés, le politique étant le plus général et l'opérationnel le plus spécifique, et les contraintes définies à un niveau donné se propagent aux niveaux plus spécifiques. La démarche n'est pas linéaire, on peut être obligé d'effectuer des retours en arrière pour remettre en cause des choix faits à des niveaux supérieurs et les impondérables sont si nombreux que l'ordre final des étapes n'est souvent fixé que lors de l'expérimentation.

Selon la manière dont a été posé le problème, les stratégies utilisées ou le niveau auquel on se situe, l'élaboration du plan de synthèse s'effectue suivant différents modes de raisonnement.

Le plus souvent, c'est un raisonnement analytique qui est suivi, l'espace du problème étant alors parcouru rétrosynthétiquement de la cible vers les produits de départ. Ceci est naturel lorsqu'on cherche à synthétiser une molécule complexe et que le choix de produits de départ n'est pas évident *a priori*. Dans ce cas, la structure de la cible est la donnée la mieux définie du problème et l'objectif à atteindre en priorité est sa simplification.

Le raisonnement peut également être synthétique. Celui-ci est notamment indispensable pour vérifier la pertinence des méthodes de synthèse sélectionnées.

Il est possible aussi d'employer un mode de raisonnement bidirectionnel quand on connaît à la fois la cible et des produits de départ possibles .

La qualité d'un plan de synthèse est difficile à évaluer. Elle est liée à la brièveté du chemin, au choix judicieux des objectifs à atteindre et leur ordonnancement, choix qui est rarement trivial, et au degré de contrôle exercé sur le franchissement des étapes par la sélection de réactions spécifiques.

2.3. Synthèse assistée par ordinateur

L'idée d'employer l'ordinateur en synthèse organique est apparue au début des années 60. Dès cette époque, on a commencé à concevoir des systèmes de gestion de bases de données pour stocker et retrouver l'information sur les réactions connues. Certains d'entre eux, aujourd'hui commercialisés et gérant des bases de données de plusieurs centaines de milliers de réactions particulières, sont devenus des outils indispensables dans les laboratoires de synthèse [ZAS 90].

En 1963, Vléduts avait prédit l'émergence d'une autre classe de systèmes qui, pourvus de connaissances chimiques et de capacités de raisonnement, devaient aider plus directement le chimiste à établir des plans de synthèse [VLE 63]. Le premier de ces systèmes, dits de synthèse assistée par ordinateur, réalisé par Corey et Wipke, a été décrit en 1969 [COR 69], après que Corey[3] ait entrepris de codifier l'analyse rétrosynthétique [COR 67]. Depuis, de nombreux autres systèmes ont été développés mais peu d'entre eux ont atteint le stade opérationnel. Ces derniers ont été passés en revue récemment [OTT 92, BAD 92]. Tous possèdent une interface graphique afin de communiquer avec l'utilisateur dans le langage des formules structurales. Certains fonctionnent dans le sens rétrosynthétique pour rechercher des séquences de réactions tandis que d'autres opèrent dans le sens synthétique pour prédire le résultat de telles réactions. Quelques uns travaillent automatiquement mais la plupart sont interactifs. Ces différents systèmes ont été conçus selon 2 approches distinctes. La première est

[3] Prix Nobel de chimie en 1990

empirique. Faisant largement appel à la connaissance d'experts de la synthèse, elle nécessite la constitution et la mise à jour de bases de connaissances contenant la description de nombreuses méthodes de synthèse et de règles heuristiques. La seconde est basée sur des modèles tendant à regrouper l'ensemble des réactions sous un petit nombre de schémas, très généraux, décrits à l'aide d'un formalisme mathématique. N'employant pas - ou très peu - de connaissance d'experts, elle repose sur le respect des principes fondamentaux de la théorie structurale (valence, ...), sur des principes heuristiques généraux (convergence des chemins, distance chimique minimale, ...) et sur l'application de fonctions d'évaluation basées sur des propriétés physico-chimiques (enthalpie, électronégativité, ...). Chaque approche possède son intérêt et ses limites mais, en pratique, celle qui a fourni les meilleurs résultats jusqu'à présent est l'approche empirique. Ainsi, LHASA, le programme développé selon cette approche par l'équipe de Corey [COR 85], est-il capable de suggérer au chimiste des solutions convenables à des problèmes aussi complexes que celui de la synthèse du Taxol [ROZ 94].

2.4. Origine du projet RÉSYN

Contrairement aux systèmes de gestion de bases de données sur les réactions, les systèmes de synthèse assistée par ordinateur ne se sont pas encore imposés au chimiste comme des outils indispensables. Plusieurs raisons peuvent être avancées. En premier lieu, malgré les recherches effectuées sur le sujet, la démarche suivie par le chimiste au cours de la résolution des problèmes de synthèse reste mal comprise. Les stratégies de synthèse et les raisonnements par analogie utilisés sont des thèmes à approfondir particulièrement. Par ailleurs, la plupart des systèmes connus ne sont que des prototypes aux performances relativement modestes. Ils sont difficiles d'accès, peu conviviaux, fournissant de trop nombreuses solutions sans expliquer leurs résultats. Enfin, concernant les systèmes les plus performants, le point critique est la constitution et la mise à jour de leur base de connaissances.

Dans le cas de LHASA, le meilleur système actuel, l'élément de connaissance relatif aux méthodes de synthèse présent dans sa base est la transformation. C'est la description d'une méthode dans le sens rétrosynthétique, c'est-à-dire des produits vers les réactifs (figure 2).

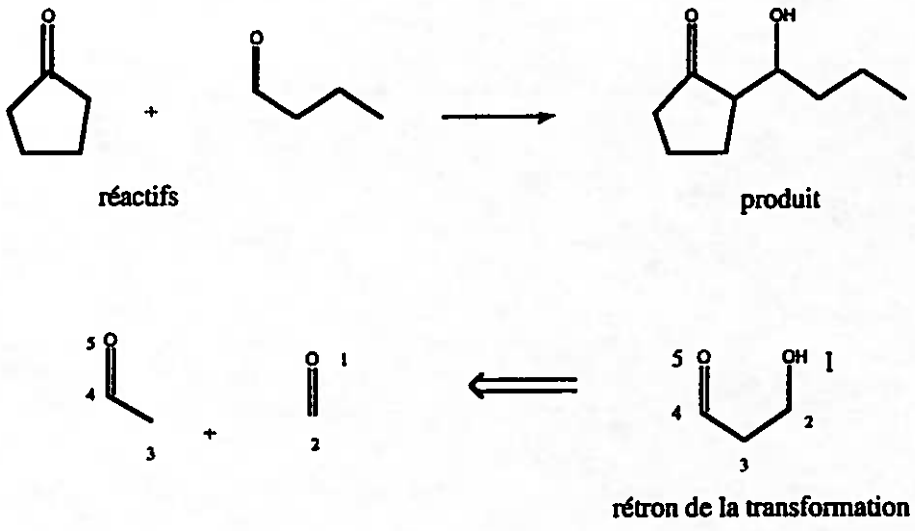


figure 2. *un exemple de réaction d'aldolisation (→) et la méthode correspondante décrite rétrosynthétiquement (⇐)*

Le système dispose d'une base de connaissances contenant plus de 2000 transformations et environ 500 combinaisons tactiques de transformations. Pour une molécule cible donnée, LHASA construit pas à pas - ou dans certains cas en enchaînant automatiquement plusieurs étapes - un arbre de solutions appelé arbre de rétrosynthèse, sous le contrôle de l'utilisateur, lequel est responsable des choix stratégiques et intervient dans l'évaluation des étapes. Partant d'une représentation du graphe moléculaire de la cible, le programme génère celui de précurseurs possibles de cette cible en appliquant les modifications spécifiées dans les transformations. Ces précurseurs peuvent eux-mêmes être pris comme de nouvelles cibles et le processus est réitéré jusqu'à rencontrer des structures connues ou suffisamment simples.

Représentée à l'aide du langage spécialisé CHMTRN, une transformation est un programme ayant la structure générale indiquée à la figure 3. Un tel programme peut comporter plusieurs dizaines d'instructions. Son écriture et, plus encore, sa modification exigent une connaissance de la programmation que ne possèdent généralement pas les experts de la synthèse organique. Les enquêtes structurales menées dans l'environnement du rétron sont imprécises et difficiles à exprimer. De plus, le manque de souplesse inhérent à ce type de langage ne facilite pas la prise en compte de nouveaux concepts. Enfin, ne transformation ainsi décrite est difficile à traduire automatiquement dans un autre mode de représentation, ce qui ne favorise pas la réutilisation de la connaissance par d'autres systèmes.

Ce modèle de la transformation est probablement trop complexe car il mêle des informations de types très différents. Ainsi, certaines d'entre elles concernent des aspects stratégiques tandis que d'autres servent au contrôle de la pertinence de la transformation. De plus, une partie des informations contenues dans la base de connaissances peut être redondante lorsqu'une même situation se trouve décrite par plusieurs transformations, lesquelles sont des éléments indépendants les uns des

autres. Finalement, ce modèle tient peu compte du caractère très structuré de l'univers de la chimie organique alors qu'on peut mettre en évidence de nombreuses relations d'ordre ou de composition entre les concepts de cet univers [LAU 90].

nom mnémonique	Exemple : ALDOLISATION (figure 2)
commentaires bibliographiques	*Références à des articles portant sur l'intérêt synthétique de la réaction.*
score initial	*Nombre mesurant l'intérêt synthétique de la transformation. Il varie au cours de l'application de celle-ci en fonction de conditions structurales ou réactionnelles*
rétron	*Sous-structure clé dont la présence dans la structure cible est requise pour que la transformation soit sélectionnée.*
indications stratégiques et/ou tactiques	Exemple : Casse une liaison C-C
enquêtes	*De la forme si ... alors ..., elles testent l'environnement de la sous-structure de la cible correspondant au descripteur et fixent les limites d'application de la transformation ; elles peuvent provoquer son rejet ou la modification de son score.* Exemple : Si un halogène est lié au carbone4 du rétron alors la transformation n'est pas applicable.
conditions réactionnelles	*Peuvent abaisser le score lorsque la cible possède des groupements sensibles* Exemple : Conditions nucléophiles
manipulations	*Actions que le système doit effectuer pour engendrer les précurseurs à partir du graphe de la cible* Exemple : Casser la liaison 2-3.

figure 3. *structure générale d'une transformation*

La société ROUSSEL UCLAF, ayant constaté les insuffisances des différents systèmes de synthèse assistée par ordinateur disponibles, a lancé un projet pour étudier la faisabilité d'un nouveau système répondant aux spécifications suivantes:

- pouvoir effectuer l'analyse rétrosynthétique d'une molécule cible, selon un mode interactif ou automatique, en tenant compte de contraintes industrielles (coût, sécurité, respect de l'environnement, etc.).
- être capable d'expliquer les solutions proposées.
- être conçu à partir de la connaissance d'un expert reconnu du domaine, E. TOROMANOFF.
- faciliter la révision et la réutilisation de la connaissance
- permettre l'expérimentation de différents modèles chimiques
- permettre d'explorer les possibilités de techniques d'intelligence artificielle plus élaborées que les techniques déjà employées dans le domaine.

Compte tenu de ces spécifications, il était logique de penser à la représentation par objets et au raisonnement par classification. C'est sur la base de ces principes qu'a été conçu et réalisé le prototype RÉSYN [QUI 91], dans le cadre d'une

collaboration entre le LIRMM, FRAMENTEC COGNITECH et ROUSSEL UCLAF.

Cet article présente les principaux résultats et les prolongements de ce travail.

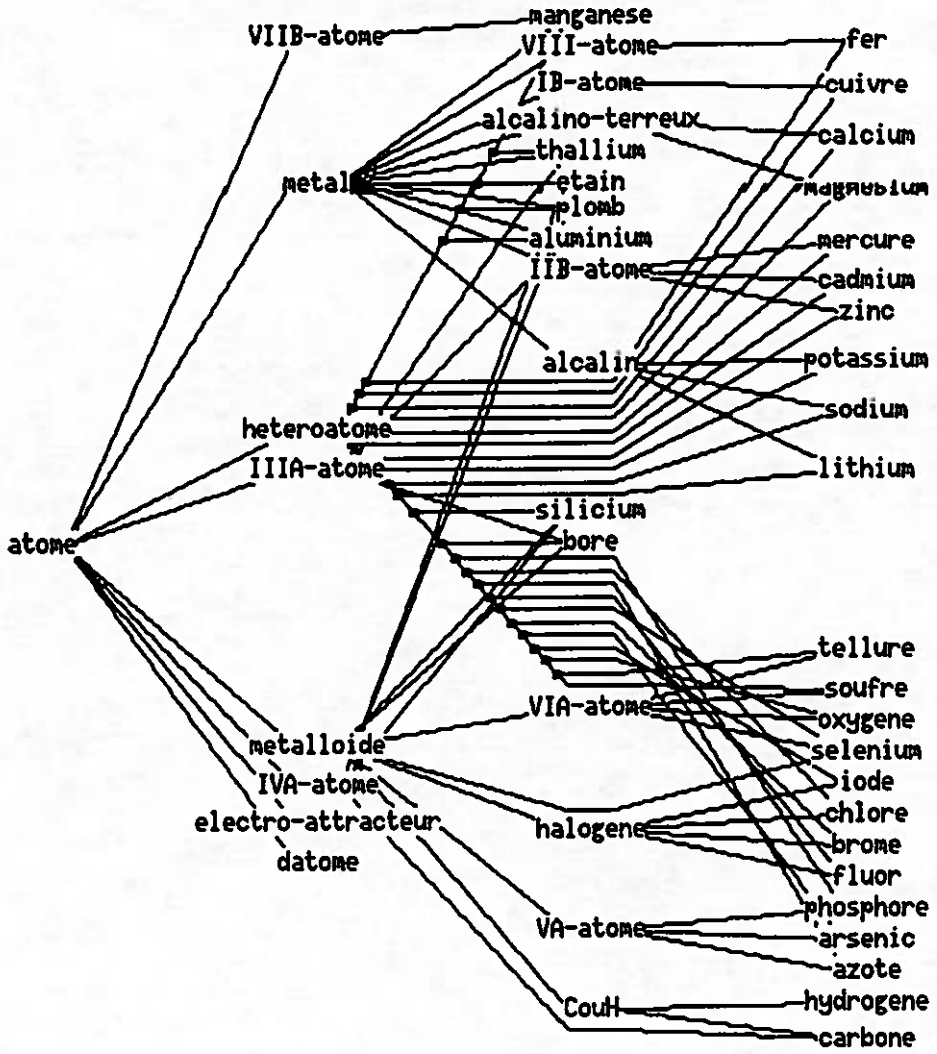


figure 4. *les classes d'atomes dans RÉSYN. Cette figure reproduit le graphe d'héritage tel qu'il est visualisé dans le système Y3.*

3. La connaissance du système RÉSYN

La connaissance de RÉSYN se répartit en 4 sous-domaines : les concepts élémentaires, les structures, les méthodes de synthèse et les heuristiques.

3.1. *Les concepts élémentaires*

Un système tel que RÉSYN doit avoir, en premier lieu, une certaine connaissance des objets chimiques élémentaires que sont les atomes et les liaisons.

Les objets appartenant à la classe des atomes sont les plus simples du domaine. Ils possèdent différents attributs tels qu'un symbole atomique, une masse atomique, une valence, etc.... La classe des atomes se subdivise en plusieurs sous-classes selon les différents types atomiques.

Un *atome réel* sera instance d'une classe représentant une case du tableau de Mendeléev. Un *atome générique* sera instance d'une classe plus abstraite, représentant un sous ensemble de la classification périodique. Ces classes d'atomes génériques ne sont pas disjointes, comme cela apparaît sur la figure 4.

On définit de même les notions de *liaison réelle* et de *liaison générique*. Les classes de liaisons sont données sur la figure 5. La propriété principale de la classe des liaisons est de relier deux atomes.

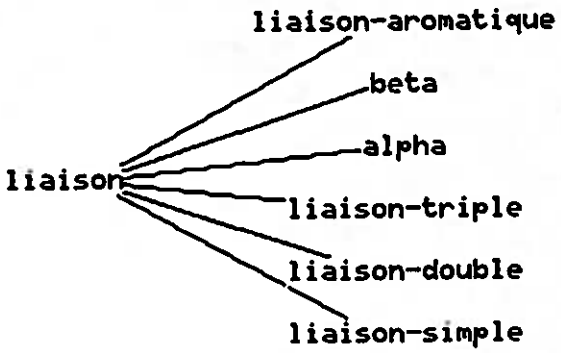


Figure 5. *les classes de liaisons dans RÉSYN*

Ces objets permettent de définir des structures moléculaires génériques, composées d'atomes génériques et de liaisons génériques.

3.2. *Les structures*

En chimie organique, la connaissance se définit essentiellement en termes de structures moléculaires et de relations entre elles. Une structure moléculaire définit en intension la classe de toutes les molécules réelles contenant cette structure.

Une structure moléculaire est composée d'atomes et de liaisons. Elle permet de représenter une molécule, mais aussi des fragments de celle-ci, comme les chaînes, les cycles, les stéréogènes, etc....

La structure la plus élaborée est la molécule, comme par exemple la molécule de taxol représentée sur la figure 1, ou bien l'acide chrysanthémique ou l'alcool de patchouli, représentés sur la figure 6, qui nous serviront par la suite d'exemples pour illustrer le fonctionnement de RÉSYN. Ses différentes caractéristiques (chaînes, liens, stéréochimie, cycles) sont autant de points de vue selon lesquels RÉSYN peut la considérer. Ces caractéristiques doivent être perçues par le système.

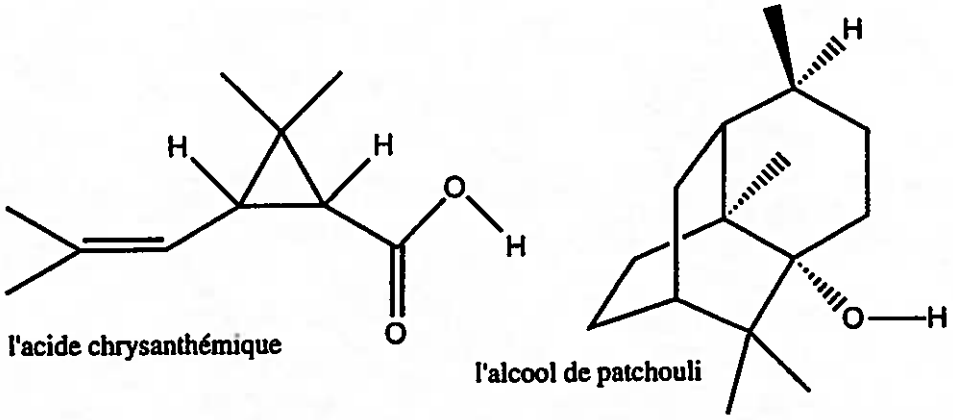


Figure 6. *Exemples de molécules représentées selon les conventions habituelles des chimistes, telles qu'elles ont été énoncées sur le figure 1.*

3.3. Les méthodes de synthèse

Une méthode de synthèse, du point de vue du chimiste, est une réaction ou une suite de réactions qui permet d'atteindre un objectif. Les connaissances que possède un chimiste sur les réactions ne sont pas toutes de même niveau. Elles sont hiérarchisées selon différents ordres (généralité, intérêt, etc....). Actuellement, RÉSYN ne prend en compte qu'un niveau de représentation des réactions, celui qui est le plus utile pour l'analyse rétrosynthétique. Lorsqu'une méthode est constituée par une suite de réactions — c'est par la connaissance de telles suites, notamment, que se distinguent les experts de la synthèse — elle peut être d'une part généralisée et d'autre part contractée par élimination des étapes intermédiaires.

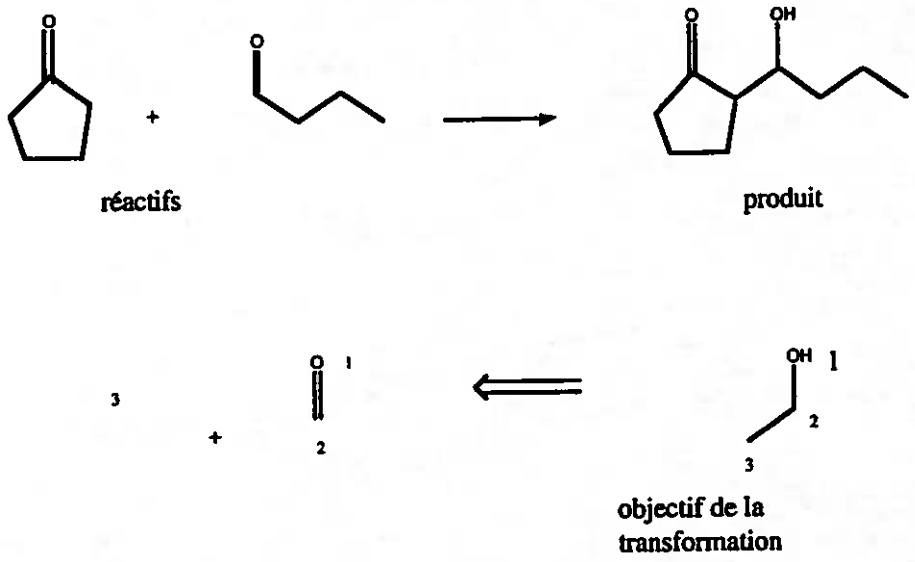


figure 7. *Notion d'objectif d'une transformation, illustré sur l'exemple de la réaction d'aldolisation.*

En fait, dans le domaine de RÉSYN, le concept de méthode de synthèse est exprimé à l'aide de 3 concepts : transformation, objectif et contexte.

Le mode de raisonnement de RÉSYN étant analytique, il faut considérer les méthodes selon le sens rétrosynthétique. En comparant le produit de réaction avec l'ensemble des réactifs, on isole les parties structurales qui ont subi une modification (comme cela est montré sur la figure 7). La sous structure modifiée qui appartient au produit est l'objectif atteint par la transformation. On nomme transformation l'opérateur qui décrit l'ensemble des actions à effectuer sur l'objectif pour passer du produit aux réactifs (casser une liaison, ajouter un atome, etc....). Un objectif peut être la référence de plusieurs transformations mais une transformation n'a qu'un seul objectif de référence.

C'est en appariant l'objectif d'une méthode avec une sous structure identique présente dans la molécule cible et en appliquant à cette sous structure la transformation correspondante, que le système engendre des précurseurs de cette cible.

Une population d'objets chimiques peut s'organiser selon diverses relations conceptuelles établies entre ces objets [DUB 75]. C'est une relation de généralisation-spécialisation qui organise l'ensemble des objectifs dans la base de connaissances de RÉSYN. Au niveau le plus élevé se situent les objectifs élémentaires (ou primitifs). Ce sont ceux qui ne comportent qu'une seule liaison et dont la présence dans la molécule cible est recherchée en premier lieu par RÉSYN. Les objectifs des autres niveaux sont des objectifs spécialisés, lesquels sont référence de transformations modifiant plusieurs liaisons. Un objectif donné est inclus, éventuellement plusieurs fois, dans les objectifs de niveaux inférieurs avec lesquels il est en relation.

La notion de restriction sur les opérateurs est introduite à l'aide du concept de contexte. Un contexte est, comme un objectif, une structure moléculaire. Si l'objectif définit la sous-structure minimale nécessairement présente dans la molécule cible pour que l'application d'une transformation donnée soit valide, le contexte définit l'environnement structural dans lequel se situe l'objectif et indique si cette application est pertinente[4]. Un contexte défavorable à l'application d'une transformation correspond à un cas connu d'exception à l'emploi de la méthode de synthèse.

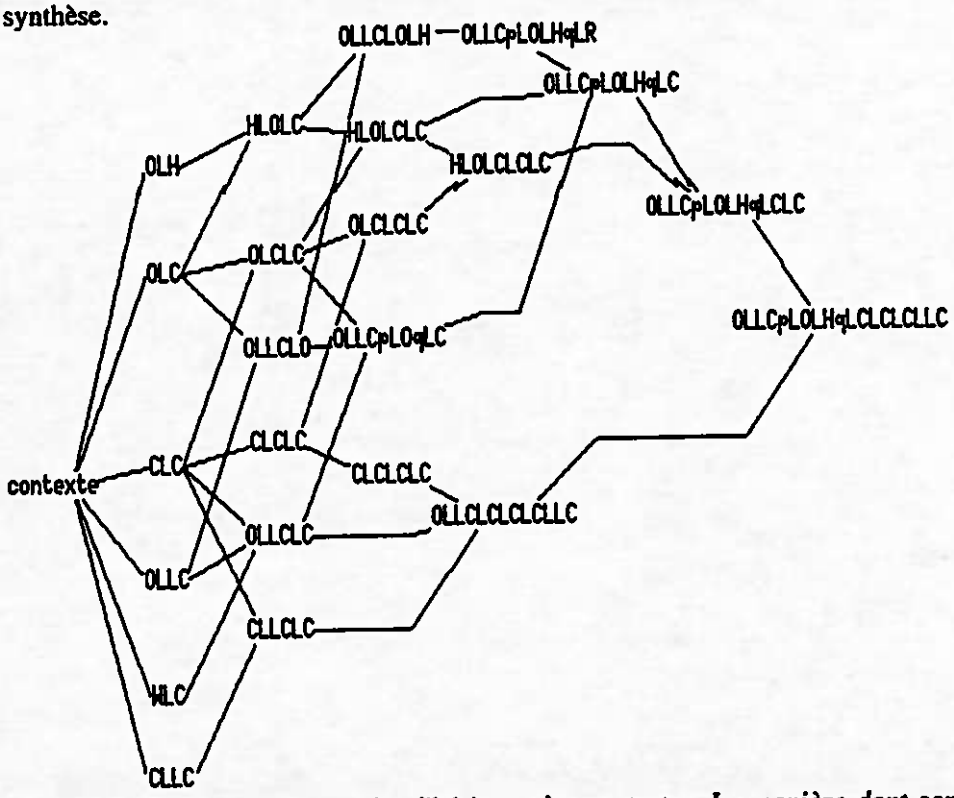


Figure 8. *Une partie du graphe d'héritage des contextes. La manière dont sont nommés les contextes a été décrite dans [LAU 90], et s'inspire des méthodes d'impression de listes partagées en LeLisp [CHA 91].*

Un contexte est donc une structure moléculaire connue pour être nécessaire, favorable, défavorable ou interdite pour l'application d'une transformation. Tout contexte d'une transformation donnée contient (au sens de l'isomorphisme de sous graphe) au moins l'objectif de cette transformation.

L'ensemble des contextes est également organisé selon une relation de généralisation-spécialisation, comme le montre la figure 8. Certaines structures peuvent être à la fois un contexte et un objectif. En effet, un objectif est toujours un contexte minimal, mais un contexte n'est pas toujours un objectif. De plus, un contexte peut s'appliquer à plusieurs transformations en étant favorable aux unes et

[4] bien que la notion de pertinence chimique ne se réduise pas à un simple contexte.

défavorable aux autres. C'est ainsi que cette organisation permet de factoriser la connaissance.

3.4. *Les heuristiques*

RÉSYN emploie des heuristiques de différents types. Elles ont toutes un but commun, qui est de chercher à réduire la combinatoire inhérente au problème de la construction d'un arbre de rétrosynthèse.

Elles peuvent être classées en trois groupes, selon le moment et la nature de leur intervention dans la construction de l'arbre :

- heuristiques de perception
- heuristiques de choix
- heuristiques d'évaluation

Un premier groupe est constitué des règles utilisées au stade de la perception, par exemple pour éliminer les redondances dues aux symétries, etc.... C'est donc un ensemble de règles auquel le système fait appel avant le début de la construction de l'arbre de rétrosynthèse.

Les autres groupes de règles servent à contrôler le fonctionnement du système, notamment pour ordonner les actions à entreprendre et les objectifs reconnus, ou encore à évaluer les chemins engendrés, en prenant en compte une certaine esthétique de la synthèse : arbre bien balancé, concision des solutions, ...

3.5. *Les différentes couches de connaissance*

Partant du principe que la mise à plat et la représentation de l'expertise d'un domaine de connaissance est un travail long et difficile, le projet européen KADS [SCH 93, WIE 94] a proposé un schéma général d'organisation d'un système à base de connaissances destiné à permettre à une même base de connaissances d'un domaine d'expertise d'être utilisée pour plusieurs types de résolutions de problèmes.

Selon ce schéma, les éléments d'un système à base de connaissances peuvent être classés en 3 couches (figure 9), que l'on s'efforce de rendre mutuellement indépendantes, afin d'en permettre une réutilisation maximale.

tâche	*mise à jour, perception, construction du plan de synthèse*
inférence	*appariement, classification, déduction heuristique, calcul des bases de cycles*
domaine	*transformations, contextes, objectifs, règles empiriques*

figure 9. *Le modèle d'expertise en couches de KADS.*

Nous nous efforcerons dans cette présentation de montrer comment ce modèle est mis en oeuvre dans le système RÉSYN où un même principe d'inférence, la classification, permet, à partir d'une même base de connaissances, de réaliser plusieurs tâches.

3.6. *Description du domaine d'expertise*

La structuration précédente a été conçue pour faciliter tant le raisonnement de RÉSYN que les opérations de mise à jour de la connaissance.

Le langage utilisé pour représenter cette connaissance doit permettre l'organisation hiérarchique de concepts, une certaine facilité de mise en oeuvre et de révision de la connaissance, la mise en perspective de celle-ci selon différents points de vue, l'accès à une base de données, un environnement de développement et d'interfaçage graphique...etc. Notre choix s'est porte sur le langage Y3 [DUC 90, DUC 91a], à la fois parce qu'il répondait assez bien à nos spécifications que parce que nous le connaissions bien [DUC 91b].

Les principaux composants de la base de connaissances de RÉSYN sont :

- Plusieurs vues d'un graphe moléculaire : les molécules, les topologies (l'une étant une forme éditable de l'autre, qui est plus efficace pour les appariements).
- Des réseaux composés de sous-graphes : les objectifs, les contextes
- Des opérateurs : les transformations.
- Des heuristiques de perception, de choix et d'évaluation.
- Une base de produits de départ.

Le modèle de connaissance décrit ici a pour origine une étude de la représentation de la connaissance en synthèse organique dont les résultats ont été décrits précédemment [LAU 90].

La définition des structures a été abordée dans la description de la représentation de connaissance du système RÉSYN, dans les paragraphes précédents.

Les transformations sont décrites cette fois par un modèle plus simple que celui de la figure 3. Il est présenté sur la figure 10.

nom mnémonique	Exemple : ALDOLISATION
intérêt	*un nombre, représentant le même type d'évaluation que le* score intitial *du modèle de la figure 3.*
commentaires	*texte libre*
référence	*objectif de la transformation*
atomes à ajouter	*des atomes*
atomes à modifier	*des atomes de l'objectif*
liaisons à casser	*des liaisons de l'objectif*
liaisons à modifier	*des liaisons de l'objectif*
liaisons à ajouter	*des liaisons*
calcul des précurseurs	*méthode*

figure 10. *modèle d'une transformation dans RÉSYN.*

C'est essentiellement d'une description fonctionnelle de l'opérateur qui va réaliser la transformation dans le système RÉSYN.

4. Les mécanismes d'inférence

Il s'agit ici de décrire les mécanismes génériques qui sont à la base des tâches réalisées par RÉSYN. Le raisonnement dans RÉSYN se fait par classification. Cette

classification s'effectue en parcourant un réseau de concepts organisés du plus général au plus spécifique. Le but est de trouver les contextes, qui correspondent à des situations chimiques, présents dans la molécule.

Les mécanismes d'inférence de RÉSYN sont donc :

- L'appariement de structures
- La classification
- La perception

Ces mécanismes reposent sur les mécanismes standards d'inférence que l'on trouve dans les langages à base de frames, dont fait partie Y3 : les réflexes. Ils sont de 3 types en Y3 :

- Les réflexes de calcul d'une valeur, déclenchés en cas d'absence de valeur d'un attribut, en remontant la hiérarchie d'héritage du frame jusqu'à ce qu'une valeur soit renvoyée. Ils sont repérés par la facette "si-besoin".
- Les réflexes de contrôle de la correction de la valeur d'un attribut, qui sont déclenchés lors de chaque tentative d'écriture d'une valeur de l'attribut, tous les réflexes présents du haut en bas de la hiérarchie étant déclenchés. Ils sont repérés par la facette "si-possible".
- Les réflexes de propagation de contraintes de consistence, qui sont déclenchés avant l'enlèvement d'une valeur (en remontant la hiérarchie) et/ou après l'ajout d'une valeur (de haut en bas de la hiérarchie). Ils sont repérés par les facettes "si-enleve" et "si-ajout".

4.1. Mécanisme d'appariement

La recherche d'appariement peut être définie comme le calcul d'une relation de généralisation-spécialisation entre objets qui tienne compte, non seulement, de la représentation structurale des objets mais aussi des relations hiérarchiques existant entre les objets comme, par exemple, la hiérarchie atomique.

D'un point de vue structurel cette relation est définie par la notion d'isomorphisme de sous-graphe partiel. Un objet o_1 est plus général qu'un objet o_2 si o_2 contient un sous-graphe partiel qui est isomorphe à o_1. L'introduction de la hiérarchie des types atomiques permet par ailleurs de prendre en considération certaines connaissances du domaine.

Ainsi nous pouvons obtenir les résultats tels que celui représenté sur la figure 11.

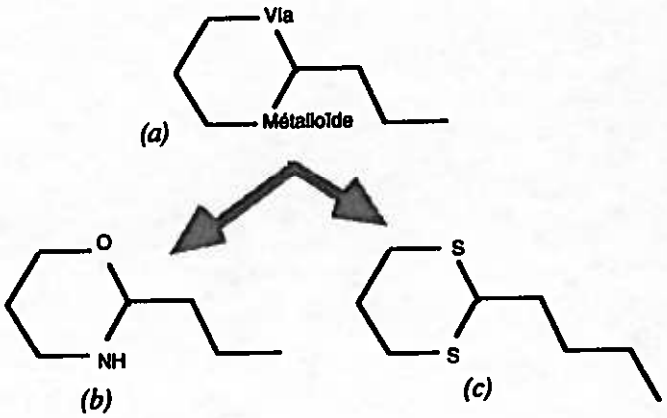


figure 11. *Exemple d'appariement de structures : (b) et (c) sont des cas particuliers de la structure (a)*

Malheureusement ce problème est NP-Complet même pour les graphes moléculaires. Nous avons décidé de représenter ce problème à l'aide d'un réseau de contraintes (CN), car la théorie des CN constitue un cadre formel simple pour modéliser des problèmes complexes et les résoudre efficacement grâce aux nombreuses méthodes qui ont été développées depuis une vingtaine d'années.

Un réseau de contraintes est défini par la donnée d'un ensemble de variables, chacune associée à un domaine et par la donnée d'un ensemble de contraintes qui mettent en relation ces variables et définissent l'ensemble des combinaisons de valeurs satisfaisant la contrainte.

Rechercher l'existence d'un isomorphisme entre un graphe H et un sous-graphe partiel d'un graphe G, correspond alors à la recherche d'une solution dans le CN suivant :

- l'ensemble des variables est l'ensemble des sommets de H;
- le domaine de chaque variable x est constitué par l'ensemble des sommets de G dont le degré est supérieur ou égal à celui de x et dont l'étiquette est compatible avec celle de x;
- le réseau possède autant de contraintes binaires que H a d'arêtes et contient en plus une contrainte n-aire. Pour chaque arête (h_1,h_2) de H on définit une contrainte binaire entre les variables h_1 et h_2 du réseau, où le couple de valeurs (g_1,g_2) est valide s'il existe une arête (g_1,g_2) dans G et si cette arête est compatible avec (h_1,h_2). La contrainte n-aire exprime que les variables doivent prendre des valeurs deux à deux différentes.

Cette modélisation est originale. Notamment, elle diffère de celle proposée par McGregor [MCG 79] par l'introduction d'une unique contrainte n-aire exprimant la notion de différence entre sommets.

Pour rechercher une solution (ou toutes), dans le réseau précédemment construit, nous avons combiné un algorithme de relaxation avec une procédure de backtrack. Nous avons utilisé comme algorithme de relaxation la consistance d'arc pour les contraintes binaires [BES 94, BES 95]et la consistance d'arc généralisée pour la contrainte de différence [REG 94].

4.2. Mécanisme de classification

Le processus de classification que nous avons employé est inspiré de celui de KL-ONE[LIP 82]. Il se décompose en deux grandes étapes :

- Recherche des subsumants les plus spécifiques (SPS).
- Recherche des subsumés les plus généraux (SPG).

La première étape correspond à un parcours en profondeur dans le graphe des structures ordonnées par la relation de généralisation-spécialisation définie au paragraphe précédent. On débute avec le sommet le plus général de ce graphe. L'algorithme s'écrit :

Algorithme *COMPARER(O,X)*
si *O ne subsume pas X*
alors *(aucun descendant de O ne peut subsumer X)*
    retourner $\emptyset$
sinon *(O est temporairement le subsumant le spécifique)*
    si *O est une feuille*
    alors *retourner (O)*
    sinon
        SPS1 := $\emptyset$
        pour *chaque descendant D de O* **faire**
            ajouter COMPARER(D,X) à SPS1
        fait
        si *SPS1* = $\emptyset$
        alors *(O est l'élément le plus spécifique)*
            retourner (O)
        sinon *retourner SPS1*
        fin si
    fin si
fin si

On remarquera que seule une partie du graphe est explorée. Si à un instant donné on atteint une structure O qui ne subsume pas X alors le sous-graphe des structures subsumées par (plus spécifiques que) O n'est pas parcouru.

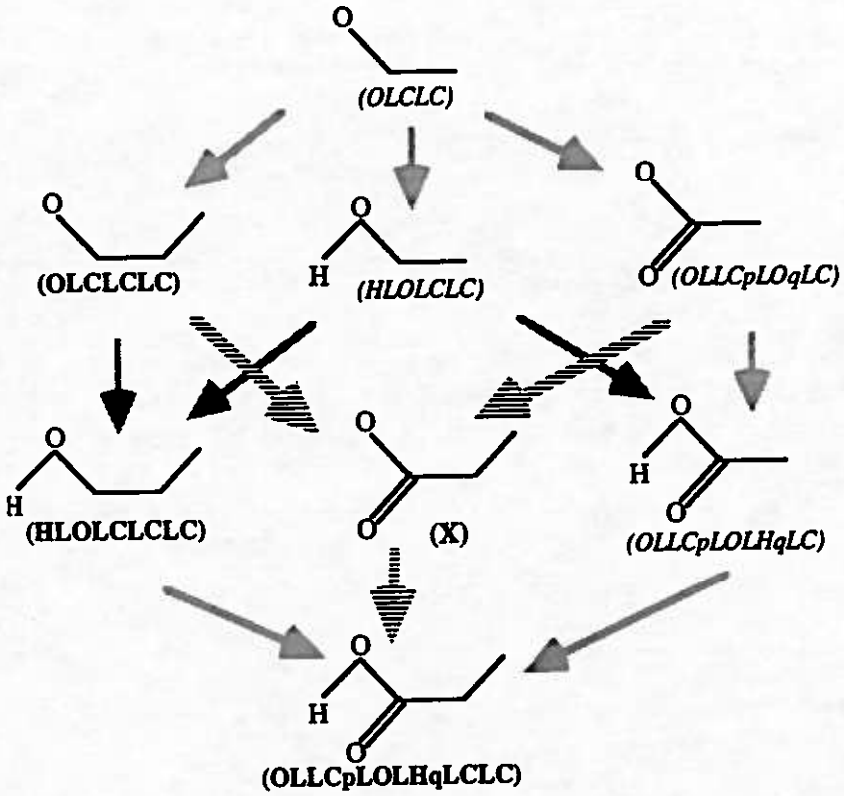


figure 12. *Classification d'une nouvelle structure X dans une partie du graphe des contextes de la figure 8 : OLCLCLC subsume HLOLCLCLC et X, X ne subsume pas HLOLCLCLC, mais un de ses descendants, OLLCpLOLHqLCLC.*

La seconde étape du processus concerne la recherche des subsumés les plus généraux. La connaissance des subsumants les plus spécifiques grâce à la première étape permet de focaliser notre recherche. En effet, comme la relation de subsomption est transitive, il suffit de n'explorer que les descendants des SPS. L'algorithme employé est donc similaire au précédent. Il est aussi important de noter que les SPG ne sont pas nécessairement les descendants immédiats des SPS. L'exemple de la figure 12 met en évidence ce problème.

4.3. Calcul des cycles pertinents

Une molécule est un objet composite regroupant des atomes et des liaisons. Pour décrire une stratégie de synthèse, il est souvent nécessaire de définir d'autres objets composites à partir de sous-ensembles (pas nécessairement disjoints) des ensembles d'atomes et d'arêtes. De ces différents objets, les plus difficiles à percevoir sont ceux qui décrivent un cycle du graphe moléculaire. Parmi tous les cycles présents dans une molécule, nous ne retenons qu'un ensemble de cycles jugés pertinents du point de

vue de la chimie. Rappelons que l'ensemble des cycles élémentaires d'un graphe non orienté, possédant m arêtes, permet d'engendrer un sous-espace vectoriel de $(0,1)^m$. L'ensemble des cycles pertinents est alors défini par l'union des bases de cycles de taille minimale (minimale au sens de la somme du nombre de sommets de chacun des cycles composant la base).

Le nombre de cycles pertinents ainsi définis peut s'avérer exponentiel pour certains graphes qui sont cependant très exceptionnels en chimie organique.

Nous avons donc élaboré un algorithme polynomial permettant de déterminer — sans l'énumérer — l'ensemble des cycles pertinents. Cet algorithme repose sur la détermination d'un ensemble de cycles pertinents prototypes, et a été décrit dans [VIS 95].

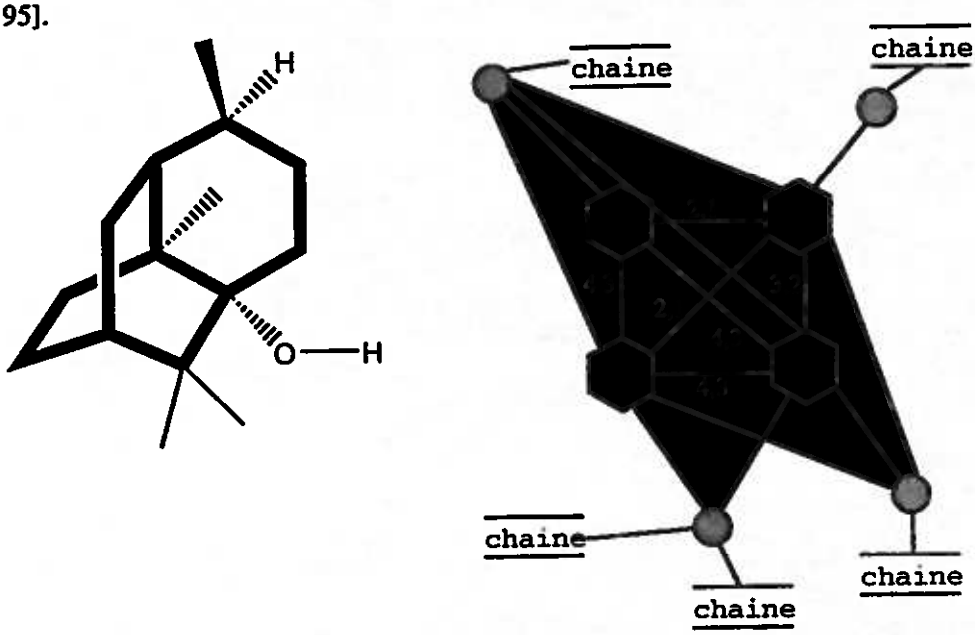


Figure 13. *Perception de la molécule d'alcool de patchouli et représentation abstraite obtenue.*

À partir de chaque prototype on peut directement énumérer un ensemble de cycles pertinents ou bien évaluer, en temps polynomial, le nombre de cycles que l'on peut dériver de chaque prototype. Les cycles prototypes sont déterminés de façon à ce que les ensembles de cycles associés aux divers prototypes constituent une partition de l'ensemble des cycles pertinents.

Le résultat obtenu sur la molécule d'alcool de patchouli (figure 6) est montré sur la figure 13.

4.4. Mécanismes de déclenchement des heuristiques

Il s'agit dans ce cas du mécanisme classique de déduction après unification de l'objectif spécifique qui constitue la prémisse de la règle avec la molécule cible. Les conclusions des règles indiquent l'application de transformations. Le rôle de ces règles sera illustré plus loin sur un exemple (figure 18).

5. Les tâches dans RÉSYN

Deux types de tâches échoient au système RÉSYN, qui sont l'analyse de la molécule cible, afin de produire le plan de synthèse, et la mise à jour de la base de connaissances. Nous allons montrer que ces deux familles de tâches font appel aux mécanismes génériques d'inférence montrés au paragraphe précédent.

5.1. L'analyse de la molécule cible

Le schéma général de fonctionnement de RÉSYN est donné sur la figure 14. Le rôle de l'utilisateur dans cette boucle générale de contrôle sera détaillé plus loin.

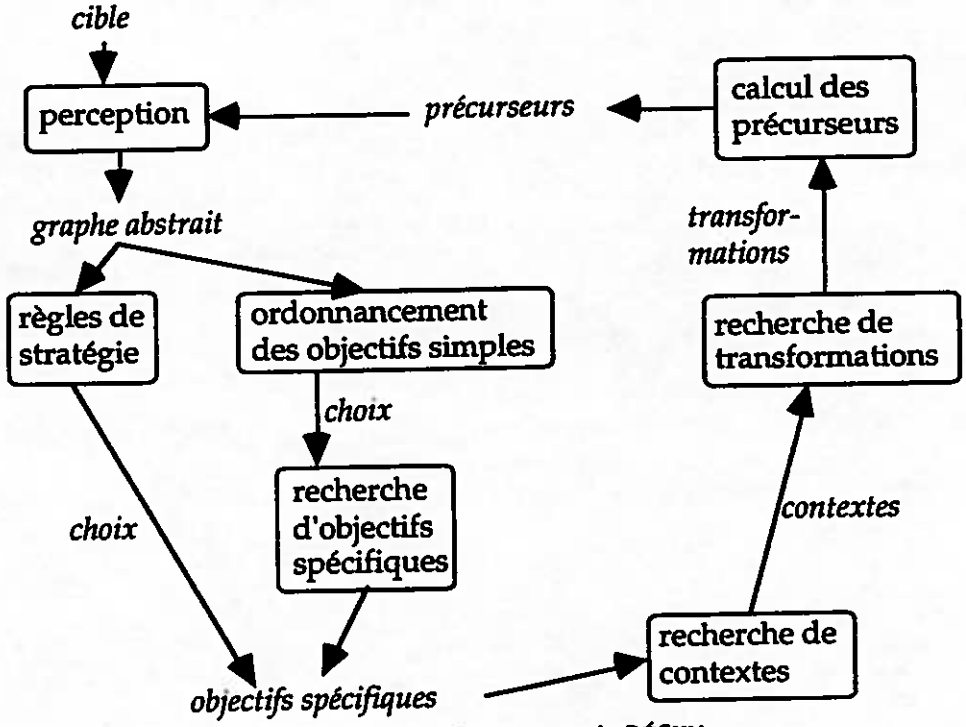


Figure 14. *Le schéma général de fonctionnement de RÉSYN*

Par rapport au modèle en couches de KADS, nous sommes ici dans la couche tâches. Ce schéma de fonctionnement représente bien en effet la décomposition de la tâche "construire des plans de synthèse" en sous tâches. Chacune de ces sous tâches est basée sur un mécanisme d'inférence.

5.1.1. Perception de la molécule cible

La tâche de perception est basée sur un mécanisme générique de recherche des cycles pertinents, c'est à dire de l'union des bases de cycles de taille minimale dans un graphe, qui est ici la molécule cible.

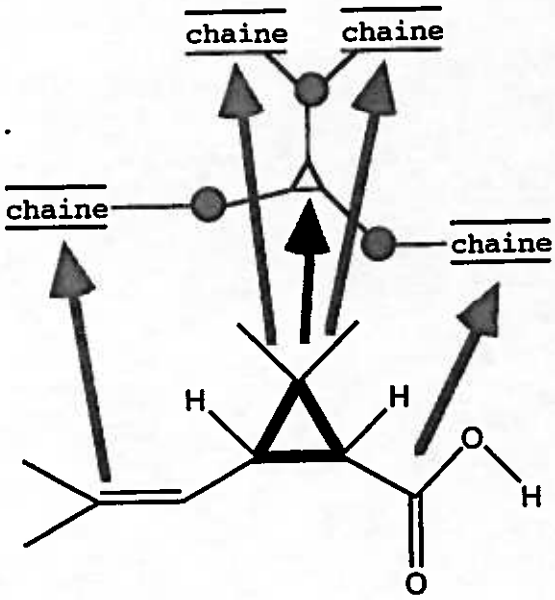


Figure 15. *Perception de la molécule d'acide chrysanthémique et représentation abstraite obtenue.*

On obtient après cette étape une représentation abstraite de la molécule, en termes de cycles et de chaînes, c'est à dire de composants de plus haut niveau que les atomes ou les liaisons. La figure 15 visualise cette représentation abstraite sur la molécule d'acide chrysanthémique présentée sur la figure 6, comme la figure 13 la montrait sur l'alcool de patchouli.

Cette représentation va être à la base de la mise en oeuvre de la stratégie de synthèse.

5.1.2. Choix stratégique et tactique pour la synthèse

La tâche de sélection d'une règle de stratégie est basée sur un mécanisme d'appariement de graphes, ici l'objectif spécifique constituant les prémisses de la règle et la molécule cible après perception. La notion de stratégie a été définie au paragraphe 2.2. Considérons ici deux exemples de stratégies.

Pour construire une molécule constituée de cycles et de chaînes, on peut chercher à construire chaque cycle ou chaîne, puis à les assembler, ou bien à construire une ébauche acyclique de la molécule et à fermer les cycles en fin de synthèse. Dans le cadre du raisonnement rétrosynthétique, la première stratégie consistera à casser les liens chaîne-cycle en premier, et la seconde à casser les cycles.

La tâche d'ordonnancement des objectifs simples (qui sont, par défaut, les liaisons de la molécule cible) consiste à évaluer chacun par un ensemble de critères numériques résultant du choix de stratégie, et calculés sur la représentation abstraite de la molécule produite par la perception.

Dans notre exemple, la première stratégie consistera à classer en premier les liaisons qui sont des liens chaîne-cycle, et la seconde à mettre en premier les liaisons appartenant à des cycles.

L'intérêt de l'étape de perception présentée au paragraphe précédent est de permettre une formalisation de cette tâche de choix stratégique sous la forme d'un raisonnement par classification sur le graphe qui représente le résultat de la perception. En effet, une directive stratégique peut s'exprimer comme un opérateur (conserver/transformer) portant sur un morceau perçu de la molécule-cible.

Dans notre exemple, sur un cycle perçu, la première stratégie dira "préserver" et la seconde dira "transformer".

A ce choix stratégique s'ajoute des considérations tactiques qui, par défaut, ordonnent les objectifs simples par ordre de centralité dans la molécule-cible.

5.1.3. *Recherche d'un objectif*

La tâche de recherche d'objectifs spécifiques à partir d'un objectif simple sélectionné consiste à appliquer les mécanismes d'appariement et de classification pour trouver, parmi les objectifs de transformation contenant l'objectif simple, tous ceux qui sont contenus dans la molécule cible.

Mentionnons ici que c'est l'appariement de l'objectif dans la cible qui doit vérifier cette propriété.

Cette tâche conditionne le choix de la transformation, en fournissant une liste de transformations possibles. Parmi ces transformations, les contextes permettront de déterminer les transformations permises.

Sur notre exemple de l'acide chrysanthémique, le premier objectif trouvé, qui correspond au cycle triangulaire, a plusieurs manières de s'apparier avec le cycle contenu dans la molécule.

5.1.4. *Recherche d'un contexte*

Nous avons vu, sur la figure 8, que les contextes étaient partiellement ordonnés par la relation d'isomorphisme de sous-graphe. Un contexte peut donc avoir plusieurs antécédents immédiats, et même, pour chaque antécédent, plusieurs endomorphismes.

La tâche de recherche de contextes à partir des objectifs spécifiques sélectionnés, consiste à rechercher les contextes les plus spécifiques contenus dans la molécule cible.

Chaque contexte donne une liste de transformations permises ou interdites. La recherche de transformations consiste alors à établir la liste des transformations permises.

Remarque: *ces deux dernières tâches ne sont pas séparables.*

5.2. *Constitution et mise à jour de la base de connaissances*

La Connaissance est constituée, à la base, par l'ensemble des transformations. Classifier les structures mises en jeu dans ces transformations a pour but, outre l'analyse d'une molécule, de faciliter la mise à jour de la base de connaissances.

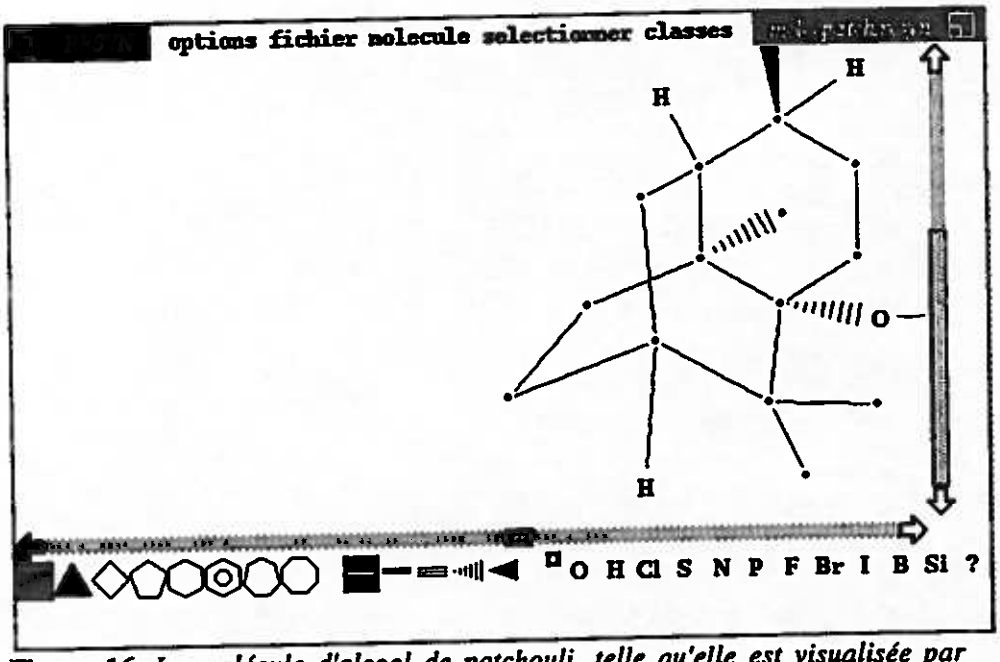


Figure 16. *La molécule d'alcool de patchouli, telle qu'elle est visualisée par l'éditeur de molécules.*

5.2.1. Ajout d'une transformation

Lorsque l'on veut ajouter une transformation à la base, elle doit être repérée par rapport aux structures présentes dans la base.

5.2.2. Ajout d'un objectif

L'objectif d'une transformation est, comme nous l'avons défini plus haut, la sous-structure mise en jeu dans la transformation.

La tâche d'ajout d'un objectif se fait, une fois décrite la transformation, par appariement et classification pour trouver l'ensemble des objectifs de transformations contenus dans l'objectif ajouté.

5.2.3. Ajout d'un contexte

Les contextes d'une transformation sont, comme cela a été défini plus haut, les structures contenant l'objectif de la transformation et dont la présence modifie le comportement de la transformation.

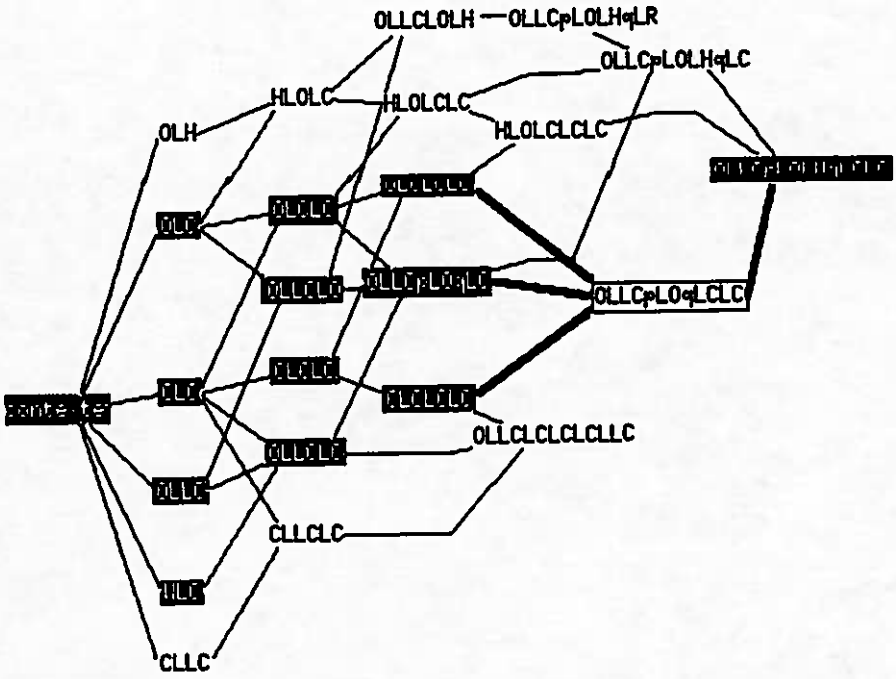


figure 17. *Classification d'un nouveau contexte OLLCpLOqLCLC dans une partie du graphe des contextes de la figure 8. Les contextes qui subsument ou sont subsumés par le nouveau contexte sont en inverse vidéo.*

La tâche d'ajout d'un contexte se fait, une fois décrite la transformation, par appariement et classification pour trouver l'ensemble des contextes de transformations contenus dans le contexte ajouté.

5.3. Étude de la conception d'un plan de synthèse.

Examinons le plan de synthèse proposé par le système RÉSYN

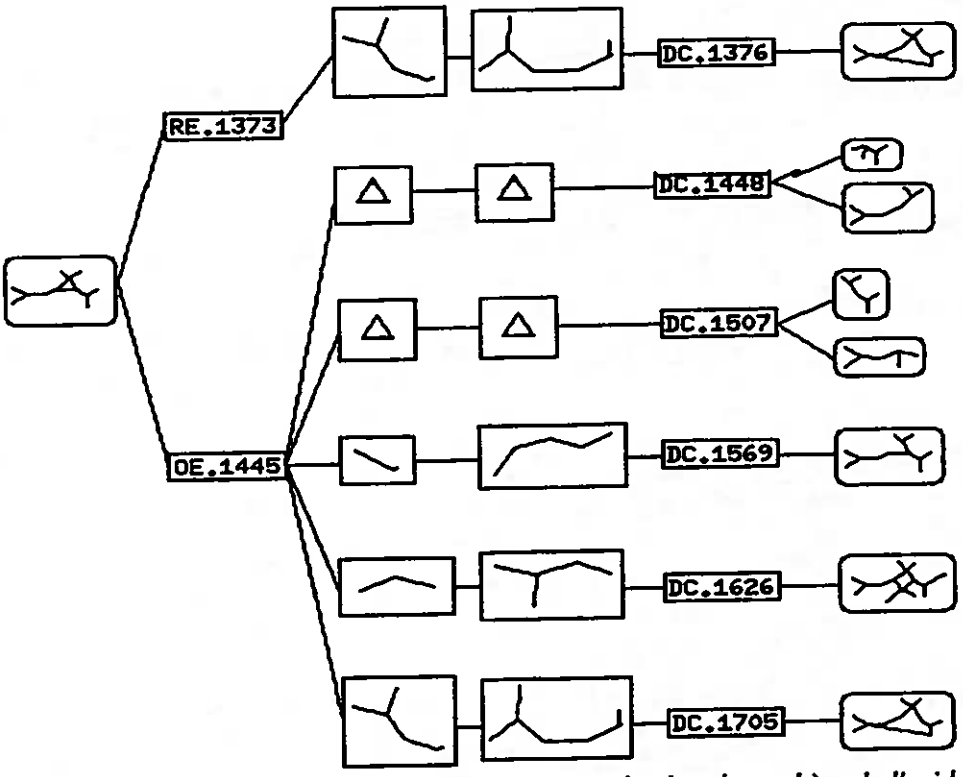
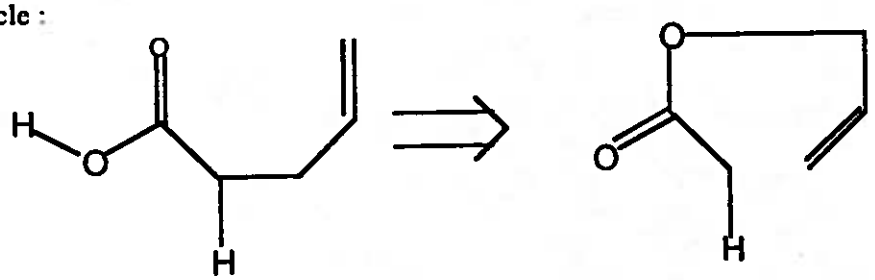


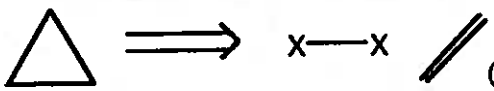
Figure 18. *Choix proposés pour la première étape du plan de synthèse de l'acide chrysanthémique.*

La première étape du plan de synthèse est présentée sur la figure 18. Elle illustre bien les rôles respectifs des objectifs et des règles heuristiques.

La transformation proposée par la règle (DC.1376 sur la figure) ne casse pas le cycle :



Par contre, les transformations proposées par les objectifs cherche d'abord à casser le cycle, comme par exemple la première d'entre elles :



(DC.1448 sur la figure)

C'est d'un précurseur de cette dernière transformation que nous partons pour proposer une deuxième étape possible du plan de synthèse, présentée sur la figure 19. On remarque cette fois que l'un des précurseurs trouvés est un produit de départ.

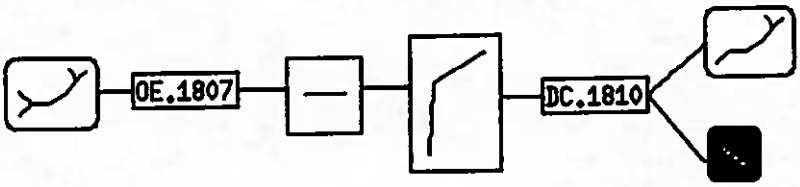
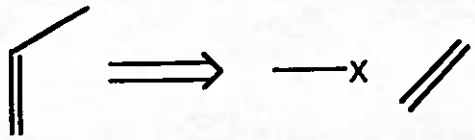


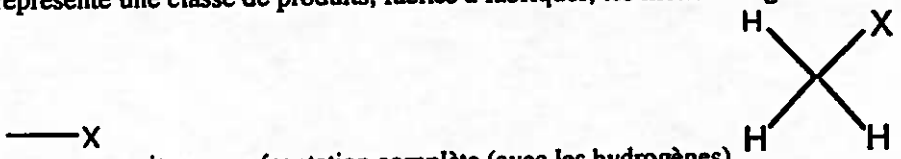
Figure 19. *Deuxième étape du plan proposé pour la synthèse de l'acide chrysanthémique, à partir de l'un des précurseurs trouvés. Le produit de départ figure en inverse vidéo.*

La transformation proposée est, comme l'une des précédentes, une transformation générique, où X désigne un halogène quelconque (c'est à dire, d'après la figure 4, un atome d'iode, de chlore, de brome ou de fluor) :



(DC.1810 sur la figure)

Le produit de départ trouvé, qui est mis en évidence sur la figure sur la figure 19, représente une classe de produits, faciles à fabriquer, les mono-halogéno-méthanes :



, soit, en représentation complète (avec les hydrogènes),

6. Conclusion

Le travail sur le système RÉSYN s'est prolongé dans différentes directions, notamment l'étude du raisonnement par cas [NAP 93], ou l'approfondissement du rôle du raisonnement par classification, où nous avons bénéficié d'apports venant d'autres travaux sur la synthèse organique [NAP 94].

7. Bibliographie

[BAD 92] BADOR, P., SURREL, M.N., ET LARDY, J.P., "Les systèmes informatiques de recherche d'information sur les réactions chimiques et les systèmes de synthèse assistée par ordinateur". *New Journal of Chemistry 16*(1992), 413-423.

[BES 94] BESSIÈRE, C. ET RÉGIN, J.C., An arc-consistency algorithm optimal in the number of constraint checks. In *proceedings IEEE Conference on Tools with Artificial Intelligence*, IEEE TAI94, New Orleans, 1994, pp. 397-403.

[BES 95] BESSIÈRE, C. ET RÉGIN, J.C., "Using bidirectionnality to speed-up arc-consistency processing". *Springer-Verlag Lecture Notes in Artificial Intelligence series* (1995).

[CHA 91] CHAILLOUX, J., *LE-LISP, Manuel de référence, version 15.24*, INRIA, Rocquencourt, Janvier, 1991.

[COR 67] COREY, E.J., "General methods for the construction of complex molecules". *Pure and Applied Chemistry 14*(1967), 19-37.

[COR 69] COREY, E.J. ET WIPKE, W.T., "Computer-assisted analysis of complex organic syntheses". *Science 166*(1969), 178-192.

[COR 85] COREY, E.J., LONG, A.K., ET RUBENSTEIN, S.D., "Computer-assisted Analysis in Organic Synthesis". *Science 228*(1985), 408-418.

[COR 89] COREY, E.J. ET CHENG, X.M., *The Logic of Chemical Synthesis*, J. Wiley and Sons, New-York (1989).

[DUB 75] DUBOIS, J.E., LAURENT, D., PANAYE, A., ET SOBEL, Y., "Système DARC: concept d'hyperstructure formelle". *C.R. Académie des Sciences, Paris, série C* , 280 (1975), 851-854.

[DUC 90] DUCOURNAU, R., *Langage à objets, Manuel de référence, version 3.5*, SEMA GROUP, Montrouge, Novembre, 1990.

[DUC 91a] DUCOURNAU, R., *Interfaces Graphiques*, SEMA GROUP, Février, 1991.

[DUC 91b] DUCOURNAU, R. ET QUINQUETON, J., "Y3/Yafool: les exemples", Sema Group, Asterix et les autres..., Avril 1991.

[HOL 94a] HOLTON, R.A., SOMOZA, C., KIM, H.B., LIANG, F., BIEDIGER, R.J., BOATMAN, P.D., SHINDO, M., SMITH, C.C., KIM, S., NADIZADEH, H., SUZUKI, Y., TAO, C., VU, P., TANG, S., ZHANG, P., MURTHI, K.K., GENTILE, L.N., ET LIU, J.H., "First Total Synthesis of Taxol. 1. Functionalization of the B ring". *Journal of the American Chemical Society 116*(1994), 1597-1598.

[HOL 94b] HOLTON, R.A., SOMOZA, C., KIM, H.B., LIANG, F., BIEDIGER, R.J., BOATMAN, P.D., SHINDO, M., SMITH, C.C., KIM, S., NADIZADEH, H., SUZUKI, Y., TAO, C., VU, P., TANG, S., ZHANG, P., MURTHI, K.K., GENTILE, L.N., ET LIU, J.H., "First Total Synthesis of Taxol. 2. Completion of the C and D rings". *Journal of the American Chemical Society 116*(1994), 1599-1600.

[LAU 90] LAURENÇO, C., PY, M., NAPOLI, A., QUINQUETON, J., ET CASTRO, B., "Représentation de connaissance en synthèse organique à l'aide d'un langage à objets". *New journal of Chemistry 14*, 12 (1990), 921–931.

[LIP 82] LIPKIS, T., A KL-ONE Classifier. In *Proceedings KL-ONE Workshop*, Jackson, Mississippi, 1982, pp. 126-143.

[MCG 79] MCGREGOR, J.J., "Relational Consistency Algorithms and their application in Finding Subgraph and Graph Isomorphism". *Information Science 19*(1979), 229-250.

[NAP 93] NAPOLI, A. ET LIEBER, J., Finding strategies in organic synthesis planning with case based reasoning. In *Proceedings of the first European Workshop on Case Based Reasoning EWCBR'93*, Richter, M., Weiss, S., Althoff, K.D., et Maurer, F., 1993, pp. 264-269.

[NAP 94] NAPOLI, A., LAURENÇO, C., ET DUCOURNAU, R., "An Object-based representation system for organic synthesis planning". *International Journal of Human-Computer Studies* , 41 (1994), 5-32.

[OTT 92] OTT, M.A. ET NOORDIK, J.H., "Computer tools for reaction retrieval and synthesis planning in organic chemistry. A brief review of their history, methods, and programs". *Recl. Trav. Chim. Pays-Bas 111*(1992), 239-246.

[QUI 91] QUINQUETON, J., VISMARA, P., RÉGIN, J.C., PY, M., LAPIED, L., ET LAURENÇO, C., "RESYN", rapport de fin de contrat, Framentec, Décembre 1991.

[QUI 92] QUINQUETON, J., VISMARA, P., RÉGIN, J.C., PY, M., LAPIED, L., ET LAURENÇO, C., Resyn: un Système d'aide à la synthèse en Chimie Organique. In *Systèmes Experts et Applications 12 Avignon'92*, Juin 1992, pp. 305-318.

[REG 94] RÉGIN, J.C., A filtering algorithm for constraints of difference in CSPs. In *proceedings Twelth National Conference on Artificial Intelligence*, AAAI, AAAI Press/MIT Press, Seattle, 1994, pp. 362-367.

[ROZ 94] VAN ROZENDAAL, E.L.M., OTT, A.M., ET SCHEEREN, H.W., "A LHASA analysis of Taxol". *Recl. Trav. Chim. Pays-Bas 113*(1994), 297-303.

[SCH 93] SCHREIBER, G., WIELINGA, B., ET BREUKER, J., *KADS. A Principle Approach to Knowledge-Based System Development*, Academic Press, London , Vol. 11, Knowledge Based Systems(1993).

[VIS 95] VISMARA, P. ET QUINQUETON, J., Design of Planification Strategies in Organic Chemistry. In *Proceedings of IASTED'95*, Innsbruck, 1995.

[VLE 63] VLÉDUTS, G.E., "Concerning one System of Classification and Codification of Organic Reactions". *Information Storage and Retrieval 1*(1963), 117-146.

[WIE 94] WIELINGA, B., "Expertise Model Definition Document : Rationale principles underlying Expertise Modelling ", no. KADSII/M2/Uva/26/5.0, UVA, Juin 1994.

[ZAS 90] ZASS, E., "A User's View of Chemical Reaction Information Sources". *Journal of Chemical Information and Computer Science 30*(1990), 360-372.

Références bibliographiques

[Alt *et al.*, 1991] H. Alt, N. Blum, K. Melhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $o(n^{1,5}\sqrt{m/\log n})$. *Information Processing Letters*, 37:237–240, 1991.

[Armstrong and Hibbert, 1989] J.L. Armstrong and D.B. Hibbert. Representation and matching of chemical structures by a prolog program. *J. Chem. Inf. Comput. Sci.*, 29:51–60, 1989.

[Baader *et al.*, 1992] F. Baader, B. Hollunder, B. Nebel, H.J. Profitlich, and E. Franconi. An empirical analysis of optimization techniques for terminological representation systems or making KRIS get a move on. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning*, Cambridge, MA, 1992. Proceedings of the 3rd International Conference.

[Baker, 1994] A. B. Baker. The hazards of fancy backtracking. In *AAAI-94, Proceedings of the Twelth National Conference on Artificial Intelligence*, Seattle, Washington, 1994.

[Balas and Yu, 1986] E. Balas and C.S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal of Computing*, 15(4):1054–1068, 1986.

[Barnard, 1988] J.M. Barnard. Problems of substructure search and their solution. In W.A. Warr, editor, *Chemical Structures*. Springer-Verlag, 1988.

[Barnard, 1991] J.M. Barnard. Structure representation and searching. In Ash J.E., Warr W.A, and Willet P., editors, *Chemical Structure Systems*. Ellis Horwood, Chichester, 1991.

[Barnard, 1993] J.M. Barnard. Substructure searching methods: Old and new. *J. Chem. Inf. Comput. Sci.*, 33:532–538, 1993.

[Barrow and Burstall, 1976] H.G. Barrow and R.M. Burstall. Subgraph isomorphism relational structures and maximal cliques. *Information Processing Letters*, 4:83–84, 1976.

[Baumer *et al.*, 1988] L. Baumer, G. Sala, and G. Sello. *Tetrahedron*, 44:1195, 1988.

[Bayada *et al.*, 1992] D.M. Bayada, R.W. Simpson, A.P. Johnson, and C. Laurenço. An algorithm for the multiple common subgraph problem. *J. Chem. Inf. Comput. Sci.*, 32:680–685, 1992.

[Berge, 1957] C. Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U.S.A.*, 43:842–844, 1957.

[Berge, 1970] C. Berge. *Graphes et Hypergraphes.* Dunod, Paris, 1970.

[Bessière and Cordier, 1993] C. Bessière and M.O. Cordier. Arc-consistency and arc-consistency again. In *AAAI-93, Proceedings Eleventh National Conference on Artificial Intelligence*, pages 108–113, Washington, DC, 1993.

[Bessière and Régin, 1994a] C. Bessière and J.C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *ECAI'94, Proceedings of the Workshop on Constraint Processing*, pages 9–16, Amsterdam, The Netherlands, 1994.

[Bessière and Régin, 1994b] C. Bessière and J.C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *TAI'94, Proceedings IEEE Conference on Tools with Artificial Intelligence*, pages 397–403, New Orleans, 1994.

[Bessière and Régin, 1995] C. Bessière and J.C. Régin. Using bidirectionnality to speed-up arc-consistency processing. In M. Meyer, editor, *Constraint Processing*, volume 923 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Heidelberg, 1995.

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *IJCAI'95, Proceedings 14th International Joint Conference on Artificial Intelligence*, pages 592–598, Montréal, 1995.

[Bessière, 1992] C. Bessière. *Systèmes à contraintes évolutifs en Intelligence Artificielle.* PhD thesis, Université de Montpellier II, Sept. 1992.

[Bessière, 1994] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[Breiman *et al.*, 1984] L. Breiman, J.F. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth Inc., Belmont, California, 1984.

[Bron and Kerbosh, 1973] C. Bron and J. Kerbosh. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[Cheng and Huang, 1981] J.K. Cheng and T.S. Huang. A subgraph isomorphism algorithm using resolution. *Pattern Recognition*, 13(5):371–379, 1981.

[Corey and Cheng, 1989] E.J. Corey and X.-M. Cheng. *The Logic of Chemical Structures.* Wiley Interscience, New York, 1989.

[Corey *et al.*, 1975] E.J. Corey, W.J. Howe, H.W. Orf, D.A. Pensak, and G. Petersson. General methods of synthetic analysis. strategic bond disconnections for bridged polycuclic structures. *J. Am. Chem. Soc.*, 97:6116–6123, 1975.

[Cost and Salzberg, 1993] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[Cover and Hart, 1967] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Info. Theory.*, 13:21–27, 1967.

[Dechter and Meiri, 1989] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *IJCAI'89, Proceedings of The Eleventh International Conference on Artificial Intelligence*, pages 271–277, 1989.

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

[Dechter, 1988] R. Dechter. Constraint processing incorporating backjumping, learning and cutset-decomposition. In *Proceedings 4th IEEE conference on AI for applications*, pages 312–319, San Diego CA, 1988.

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

[Dechter, 1992] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[Dengler and Ugi, 1991] A. Dengler and I. Ugi. A central atom based algorithm and computer program for substructure search. *Comput. Chem.*, 2:75–83, 1991.

[Dinic, 1970] E.A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[Downs *et al.*, 1988] G.M. Downs, V.J. Gillet, J. Holliday, and M.F. Lynch. The sheffield university generic chemical structures project - a review of progress of outstanding problems. In W.A. Warr, editor, *Chemical Structures*. Springer-Verlag, 1988.

[Evans, 1968] T.G. Evans. A program for the solution of a class of geometric analogy intelligence test question. In M. Minsky, editor, *Semantic Information Processing*, pages 271–353. MIT Press, Cambridge, 1968.

[Even and Tarjan, 1975] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal of Computing*, 4:507–518, 1975.

[Figueras, 1972] J. Figueras. Substructure search by set reduction. *J. Chem. Doc.*, 13(4):237–244, 1972.

[Ford and Fulkerson, 1962] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.

[Freuder, 1976] E.C. Freuder. Structural isomorphism of picture graphs. In C.H. Chen, editor, *Pattern Recognition and Artificial Intelligence*. Academic, New York, 1976.

[Freuder, 1982] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery*, 29(1):24–32, 1982.

[Freuder, 1994] E.C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In *ECAI'94, Proceedings of the Workshop on Constraint Processing*, pages 27–33, Amsterdam, The Netherlands, 1994.

[Freuder, 1995] E.C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In M. Meyer, editor, *Constraint Processing*, volume 923 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, Heidelberg, 1995.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completness*. W. H. Freeman, San Francisco, 1979.

[Garey *et al.*, 1976] M.R. Garey, D.S. Johnson, and R.E. Tarjan. The planar hamiltonian circuit problem is NP-Complete. *SIAM Journal of Computing*, 5:704–714, 1976.

[Gaschnig, 1977] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings International Joint Conference on Artificial Intelligence*, page 457, 1977.

[Gaschnig, 1978] J. Gaschnig. Experimental case studies of backtrack vs waltz-type vs new algorithm for satisficing assignment problems. In *Proceedings of the 2nd Biennal Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277, 1978.

[Gascuel and Caraux, 1992] O. Gascuel and G. Caraux. Distribution-free performance bounds with the resubstitution error estimate. *Pattern Recognition Letters*, 13:757–764, 1992.

[Gascuel, 1985a] O. Gascuel. Descriptions structurelles, discrimination et apprentissage sur ces descriptions. *Biochimie*, 67:499–507, 1985.

[Gascuel, 1985b] O. Gascuel. Discriminer sur des descriptions structurelles dans un environnement qui comporte une part d'incertitude. In *Reconnaissance des Formes et Intelligence artificielle*, pages 1263–1271, Grenoble, 1985. Congrès AFCET-RFIA.

[Gerhards and Lindenberg, 1979] L. Gerhards and W. Lindenberg. Clique detection for nondirected graphs: two new algorithms. *Computing*, 21:295–322, 1979.

[Ginsberg, 1993] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[Golomg and Baumert, 1965] S.W. Golomg and L.D. Baumert. Backtrack programming. *Journal of the Association for Computing Machinery*, 12:516–524, 1965.

[Gondran and Minoux, 1985] M. Gondran and M. Minoux. *Graphes et Algorithmes*. Eyrolles, Paris, 1985.

[Grethe and Moock, 1990] G. Grethe and T.E. Moock. Similarity searching in REACCS. a new tool for the synthetic chemist. *J. Chem. Inf. Comput. Sci.*, 30:511–520, 1990.

[Haralick and Elliot, 1980] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Harary, 1969] F. Harary. *Graph Theory*. Addison Wesley, 1969.

[Hopcroft and Karp, 1973] J.E. Hopcroft and R.M. Karp. $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2:225–231, 1973.

[Hopcroft and Tarjan, 1980] J.E. Hopcroft and R.E. Tarjan. Isomorphism of planar graphs. In *Proceedings fourth annual Symposium on Theory of Computing*, 1980.

[Janssen *et al.*, 1990] P. Janssen, P. Jegou, B. Nouguier, M.C. Vilarem, and B. Castro. SYNTHIA : Assisted design of peptide synthesis plans. *New Journal of Chemistry*, 14(12):969–976, 1990.

[Jégou, 1991] P. Jégou. *Contribution à l'Etude des Problèmes de Satisfaction de Contraintes: Algorithmes de Propagation et de Résolution, Propagation de Contraintes dans les Réseaux dynamiques.* PhD thesis, Université de Montpellier II, Jan. 1991.

[Johnson and Maggiora, 1990] M.A. Johnson and G.M. Maggiora. *concepts and applications of molecular similarity.* Wiley N.Y, 1990.

[Kodratoff and Ganascia, 1986] Y. Kodratoff and J.G. Ganascia. Improving the generalisation step in learning. In T.M. Mitchell R.S. Michalski, J.G. Carbonell, editor, *Machine Learning: An Artificial Approach*, volume 2. Morgan Kaufmann, Los Altos (CA), 1986.

[Kuhl, 1984] F.S. Kuhl. A combinatorial algorithm for calculating ligand binding. *Journal of Computational Chemistry*, 5(1):24–34, 1984.

[Kumar, 1992] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[Laurière, 1976] J.-L. Laurière. *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires.* PhD thesis, Paris VI, 1976.

[Leconte, 1995] M. Leconte. A bounds-based reduction scheme for constraints of difference. Technical report, ILOG S.A., 1995.

[Levinson, 1984] R. Levinson. A self-organizing retrieval system for graph. In *Proceedings AAAI*, pages 203–206, 1984.

[Levinson, 1992] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers Math. Apllic.*, 23(6–9):573–600, 1992.

[Lingas and Syslo, 1988] A. Lingas and M. Syslo. A polynomial algorithm for subgraph isomorphism of two-connected series-parallel graphs. In *Proc. 15th ICALP*, Tampere, Finland, 1988. Springer.

[Lingas, 1986] A. Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. In *Proc. 3rd STACS's*, Orsay, France, 1986. Springer. LNCS 210.

[Liquière, 1990] M. Liquière. *Apprentissage à partir d'objets structurés. Conception et réalisation.* PhD thesis, Université de Montpellier II, Fév. 1990.

[Lovász and Plummer, 1986] L. Lovász and M.D. Plummer. *Matching Theory.* North Holland mathematics studies 121, 1986.

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

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

[Matula, 1978] D.W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Annals of Discrete Mathematics*, 2:391–406, 1978.

[McGregor, 1979] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Science*, 19:229–250, 1979.

[McGregor, 1982] J.J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software-Practice and Experience*, 12:23–34, 1982.

[Michalski, 1983] R.S. Michalski. A theory and methodology of inductive learning. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning 1 : an Artificial Intelligence Approach*, volume 1, chapter 4. Tioga publishing company, Palo Alto California, 1983.

[Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2), 1982.

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

[Mohr and Masini, 1988a] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings ECAI*, pages 651–656, 1988.

[Mohr and Masini, 1988b] R. Mohr and G. Masini. Running efficiently arc consistency. *Syntactic and Structural Pattern Recognition*, F45:217–231, 1988.

[Moock *et al.*, 1988] T.E. Moock, D.L. Grier, W.D. Hounshell, G. Grethe, K. Cronin, J.G. Nourse, and J. Theodosiou. Similarity searching in the organic reaction domain. *Tetrahedron Computer Methodology*, 1(2):117–128, 1988.

[Moon Jung Chung, 1987] Moon Jung Chung. $o(n^{5/2})$ time algorithms for the subgraph homeomorphism problem on trees. *Journal of Algorithms*, 8:106–112, 1987.

[Nadel, 1988] B.A. Nadel. Tree search and arc-consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[Nicholson *et al.*, 1987] V. Nicholson, C.C. Tsai, M. Johnson, and M. Naim. A subgraph isomorphism theorem for molecular graphs. In R.B. King and D.H. Rouvray, editors, *Graph Theory and Topology in Chemistry*, Studies in Physical and Theoretical Chemistry (51), pages 226–230. Elsevier Science Publishers B.V., Amsterdam, Netherlands, 1987.

[Nicolas, 1993] J. Nicolas. Premisers concepts pour délimiter l'apprentissage automatique. In *Ecole sur l'apprentissage automatique, Support de cours*, PRC-GDR Intelligence Artificielle. IRISA/INRIA, Mars 1993.

[O'Korn, 1977] L.J. O'Korn. Algorithms for chemical computations. In R.E. Christofferson, editor, *Am. Chem. Soc. Symposium Series*, pages 122–148, 1977. Volume 46.

[Parzen, 1962] E. Parzen. On estimation of a probability density function and mode. *Ann. Math. Stat.*, 33:1065–1076, 1962.

[Prosser, 1993a] P. Prosser. Domain filtering can degrade intelligent backtracking. In *IJCAI-93*, pages 262–267, 1993.

[Prosser, 1993b] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Prosser, 1994] P. Prosser. Binary constraint satisfaction problems: Some are harder thant others. In *ECAI'94, Proceedings 11th European Conference on Artificiel Intelligence*, pages 95–99, Amsterdam, The Netherlands, 1994.

[Prosser, 1995] P. Prosser. Mac-cbj: maintaining arc consistency with conflict-directed backjumping. Technical report, Department of Computer Science, University of Strathclyde, May 1995. RR 95/197.

[Puget, 1994] J.-F. Puget. A C++ Implementation of CLP. Technical report, ILOG S.A., 1994.

[Quinlan, 1983] J.R. Quinlan. Learning efficient classification procedures and their application to chess and games. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning*, volume 1. Morgan Kaufmann, Los Altos, CA, 1983.

[Quinqueton and Sallantin, 1983] J. Quinqueton and J. Sallantin. Algorithms for learning logical formulas. In *Proceedings IJCAI*, 1983.

[Quinqueton, 1990] J. Quinqueton. Mécanisme et méthodes de l'apprentissage symbolique, 1990.

[Régin *et al.*, 1994] J.C. Régin, O. Gascuel, and C. Laurenço. Machine learning of strategic knowledge in organic synthesis from reaction databases. In F. Bernardi and J.L. Rivail, editors, *E.C.C.C. 1, Computational Chemistry*, number 330 in AIP Conference Proceedings, pages 618–623, Nancy, France, 1994. F.E.C.S. Conference, American Institute of Physics.

[Régin, 1994] J.C. Régin. a filtering algorithm for constraints of difference in CSPs. In *AAAI-94, Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.

[Régin, 1995] J.C. Régin. Maintaining arc consistency: it is time to jump back! Technical Report 95-0050, LIRMM, Université de Montpellier II, Juillet 1995.

[Sabin and Freuder, 1994a] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Principles and Practice of Constraint Programming Workshop*, 1994.

[Sabin and Freuder, 1994b] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI'94, Proceedings 11th European Conference on Artificiel Intelligence*, pages 125–129, 1994.

[Shapiro and Haralick, 1981] L.G. Shapiro and R.M. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(5):504–519, 1981.

[Smith, 1992] B.M. Smith. How to solve the zebra problem, or path consistency the easy way. In *Proceedings ECAI*, pages 36–37, Vienna, Austria, 1992.

[Sussenguth, Jr., 1965] E.H. Sussenguth, Jr. A graph-theoretic algorithm for matching chemical structures. *J. Chem. Doc.*, 5:36–45, 1965.

[Syslo, 1982] M.M. Syslo. The subgraph isomorphism problem for outerplanar graphs. *Theoretical Computer Science*, 17:91–97, 1982.

[Tarjan, 1972] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[Tarjan, 1975] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22:215–225, 1975.

[Tarjan, 1977] R.E. Tarjan. Graph algorithms in chemical computation. In R.E. Christofferson, editor, *Algorithms for Chemical Computations*, pages 1–20. Am. Chem. Soc. Symposium Series, 1977. Volume 46.

[Thrun *et al.*, 1991] S. B. Thrun, T. Mitchell, and J. Cheng. The MONK's problems, a performance comparison of different learning algorithms. Technical Report 197, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, 1991.

[Tonnelier, 1990] C. Tonnelier. *Outils informatiques pour l'acquisition des connaissances en synthèse assistée par ordinateur.* PhD thesis, Université de Strasbourg, Jan. 1990.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Ullmann, 1976] J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[Van Hentenryck *et al.*, 1992] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. M.I.T. Press, 1989.

[Varkony *et al.*, 1979] T.H. Varkony, Y. Shiloach, and D.H. Smith. Computer-assisted examination of chemical compounds for structural similarities. *J. Chem. Inf. Comput. Sci.*, 19(2):104–111, 1979.

[Vismara *et al.*, 1992] P. Vismara, J-C. Régin, J. Quinqueton, M. Py, C. Laurenço, and L. Lapied. RESYN : Un système d'aide à la conception de plans de synthèse en chimie organique. In *Actes de la 12 e conférence sur les systèmes experts et leurs applications*, volume 1, pages 305–318, Avignon, 1992. EC2.

[Vismara, 1995] P. Vismara. *Reconnaissance et représentation d'éléments strucraux pour la description d'objets complexes. Application à l'élaboration de stratégies de synthèse en chimie organique*. PhD thesis, Université de Montpellier II, 1995.

[von Scholley, 1984] A. von Scholley. A relaxation algorithm for generic structure screening. *J. Chem. Inf. Comput. Sci.*, 24:235–241, 1984.

[Vosselman, 1992] G. Vosselman. *Relational Matching*, volume 628 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Wallace and Freuder, 1992] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.

[Wallace, 1993] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *IJCAI'93, Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, pages 239–245, Chambéry, France, 1993.

[Waltz, 1975] D. L. Waltz. Understanding line drawings of scences with shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, 1975. d'abord paru dans Tech. Rep AI271, MIT MA, 1972.

[Wilcox and Levinson, 1986] C.S. Wilcox and R.A. Levinson. A self-organized knowledge base for recall, design, and discovery in organic chemistry. In *Artificial Intelligence Applications in Chemistry*, volume 306, pages 209–230. ACS Symposium Series, 1986.

[Willet *et al.*, 1986] P. Willet, V. Winterman, and D. Bawden. Implementation of nearest-neighbor searching in an online chemical structure search system. *J. Chem. Inf. Comput. Sci.*, 26:36–41, 1986.

[Willet, 1985] P. Willet. An algorithm for chemical superstructure search. *J. Chem. Inf. Comput. Sci.*, 25:114–116, 1985.

[Willet, 1987] P. Willet. A review of chemical structure retrieval systems. *J. Chemom.*, 1:137–155, 1987.

[Wipke and Rogers, 1984a] W.T. Wipke and D. Rogers. Artificial intelligence in organic synthesis. sst:starting material selection strategies. an application of superstructure search. *J. Chem. Inf. Comput. Sci.*, 24:71–81, 1984.

[Wipke and Rogers, 1984b] W.T. Wipke and D. Rogers. Rapid subgraph search using parallelism. *J. Chem. Inf. Comput. Sci.*, 24:255–262, 1984.

[Wong, 1992] E.K. Wong. Model matching in robot vision by subgraph isomorphism. *Pattern Recognition*, 25(3):287–303, 1992.