N^o Année 2002

UNIVERSITÉ D'AIX-MARSEILLE I – U.F.R. M.I.M.

THÈSE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ D'AIX-MARSEILLE I

Spécialité Informatique

ECOLE DOCTORALE DE MATHEMATIQUES
ET D'INFORMATIQUE DE MARSEILLE

par

Cyril TERRIOUX

Approches structurelles et coopératives pour la résolution des problèmes de satisfaction de contraintes

Soutenue le 10 décembre 2002 devant le jury composé de :

M.	Boi Faltings	Rapporteur
M.	Philippe Jégou	Directeur de thèse
M.	Thomas Schiex	Rapporteur
M.	Pierre Siegel	Président
M.	Gérard Verfaillie	Examinateur

Laboratoire des Sciences de l'Information et des Systèmes

Remerciements

Je voudrais, tout d'abord, remercier tous les membres du jury : Monsieur Boi Faltings et Monsieur Thomas Schiex qui m'ont fait l'honneur d'être les rapporteurs de ce mémoire, Monsieur Pierre Siegel et Monsieur Gérard Verfaillie qui ont accepté de participer à mon jury.

Je ne remercierai jamais assez Philippe Jégou pour tout ce qu'il a fait pour moi durant ces trois années. Il m'a fourni des conditions de travail idéales pour mener à bien cette thèse. Il est toujours parvenu à me consacrer du temps, et cela malgré un emploi du temps surchargé. Et par dessus tout, il a su guider mon travail vers des voies que, sans lui, je n'aurais pas explorées.

Je remercie aussi les membres de mon équipe de recherche pour leur accueil chaleureux et l'intérêt qu'ils ont porté à mon travail au cours de ces trois années. Je remercie particulièrement Nicolas, Laurent et Stéphane. Les discussions que j'ai pu avoir avec chacun d'entre eux m'ont permis d'aborder certains problèmes d'un point de vue différent. Merci à tous les doctorants de l'équipe, Lamia, Gilles, Jean-Michel, ...

Je souhaite également remercier les collègues du bâtiment Henri Poincaré pour leur accueil ainsi que pour l'ambiance agréable qui règne dans ces locaux.

Je tiens à remercier mes parents, mes grands-parents et plus généralement ma famille qui m'a toujours encouragé dans mes études.

Enfin, je remercie ma chère et tendre Afina pour sa patience, son soutien et ses encouragements qui ont été pour moi une source de motivation tout au long de ces trois années.

Table des matières

Introduction

L'intelligence artificielle regroupe un ensemble de disciplines qui visent chacune à reproduire une partie de l'intelligence humaine. Parmi ces disciplines, nous pouvons mentionner, par exemple, la représentation et le traitement des connaissances, la preuve de théorème, la planification de tâches, le traitement du langage naturel, la perception de l'environnement, ...

Pour notre part, nous nous intéressons à la représentation et au traitement des connaissances. Cette discipline cherche à modéliser la connaissance et à raisonner à partir de cette modélisation. Ainsi, durant ces dernières décennies, de nombreux formalismes (accompagnés généralement de méthodes de déduction) ont été proposés pour modéliser la connaissance. Certains se veulent dédiés à la résolution d'un problème précis. D'autres, plus généraux, permettent de représenter des problèmes a priori totalement différents. C'est en particulier le cas du formalisme CSP (problème de satisfaction de contraintes[1]) introduit par Montanari [Mon74]. Ce formalisme permet d'exprimer une multitude de problèmes comme des problèmes de coloration de graphes, de logique propositionnelle, d'ordonnancement, de vision, de conception, de configuration, etc. Le point commun entre ces différents problèmes réside dans la possibilité d'exprimer sous forme de contraintes les propriétés et les relations qui existent entre les objets manipulés. Le pouvoir d'expression du formalisme CSP repose donc essentiellement sur les contraintes. Il est d'autant plus important que les contraintes peuvent être décrites de multiples façons (par une équation, une inéquation, un prédicat, une fonction booléenne, une énumération des combinaisons de valeurs autorisées, etc.). Cependant, le formalisme CSP ne permet pas d'exprimer certaines notions comme celle de préférence (qui se révèle fort utile lorsqu'on recherche la meilleure solution d'un problème). Aussi, plusieurs extensions ont été proposées, parmi lesquelles les CSPs valués (VCSPs [SFV95, SFV97]). Le formalisme VCSP augmente le pouvoir d'expression du formalisme CSP en autorisant la violation de certaines contraintes.

Cette thèse traite de la résolution de CSPs binaires à domaines finis et aborde également la résolution de CSPs binaires valués. La principale difficulté réside dans le fait que résoudre un CSP constitue un problème NP-complet. Aussi, devant la difficulté du problème, différents axes de recherche ont été explorés, durant ces trois dernières décennies, conduisant à la définition de nombreuses méthodes de résolution.

La technique la plus simple pour déterminer s'il existe ou non une solution consiste à effectuer une exploration complète de l'espace de recherche. Une méthode exploitant cette technique énumère donc toutes les possibilités. De nombreux travaux ont porté sur l'amélioration de cette approche afin de réduire le nombre de possibilités à étudier (ce nombre étant généralement exponentiel). Une amélioration peut consister à analyser les échecs rencontrés (notion de retour arrière intelligent), à limiter le nombre de possibilités (notion de filtrage), à utiliser une partie du travail déjà accomplie (notion de mémorisation), ... Ces méthodes sont généralement guidées par des heuristiques dont le rôle se révèle souvent prépondérant pour l'efficacité de ces méthodes.

Un autre axe de recherche a été étudié essentiellement d'un point de vue théorique. Il s'agit des

[1] Constraint Satisfaction Problem en anglais

méthodes de résolution par décomposition. Ces méthodes divisent le problème initial en plusieurs sous-problèmes, en exploitant généralement des propriétés structurelles du problème. Les sous-problèmes générés étant de taille inférieure à celle du problème initial, leur résolution est alors souvent plus facile.

Enfin, d'autres techniques résident en une exploration partielle de l'espace de recherche. Elles sont généralement regroupées sous le nom de "recherche locale". Elles permettent, dans le meilleur des cas, de produire une solution. Par contre, si aucune solution n'est produite, il est impossible de conclure à l'absence de solution. Le principal intérêt de ces méthodes incomplètes réside dans leur rapidité à produire une solution (par rapport aux méthodes énumératives).

Ainsi, de nombreuses méthodes ont été proposées pour résoudre le problème de satisfaction de contraintes. Actuellement, d'un point de vue pratique, la résolution de CSPs semble avoir atteint certaines limites. Au niveau des méthodes énumératives, les progrès constatés ces dernières années semblent dûs plutôt à de meilleures implémentations et aux progrès technologiques (en particulier l'augmentation de la cadence des processeurs) qu'à des avancées conceptuelles, même si différentes méthodes ou heuristiques ont été proposées. D'autre part, les méthodes structurelles, qui présentent l'avantage de garantir des bornes de complexité en temps meilleures que celles des méthodes énumératives, n'ont toujours pas démontré leur intérêt pratique (sauf dans de trop rares cas). Les travaux présentés dans cette thèse s'intéressent à la résolution de CSPs grâce à des méthodes énumératives et structurelles qui tirent profit des informations explicitées durant la recherche.

Durant la recherche d'une solution, toutes les méthodes de résolution explicitent des informations qui concernent aussi bien des échecs que des réussites. Certaines méthodes mémorisent de telles informations afin de les exploiter ultérieurement, évitant ainsi de nombreuses redondances dans la recherche. Le problème CSP étant NP-complet, la quantité d'informations explicitées est généralement exponentielle. Aussi, devant l'impossibilité pratique de mémoriser toutes ces informations, seules les plus judicieuses doivent être retenues. Hélas, définir la pertinence d'une information est loin d'être une tâche aisée. Cette tâche est d'autant plus délicate que de la pertinence et du nombre de ces informations dépend l'efficacité de la méthode qui les exploite. Cette thèse s'intéresse à deux types d'informations que peut expliciter une méthode de résolution durant la recherche. Plus précisement, nous étudions l'apport aussi bien théorique que pratique de ces informations pour la résolution des problèmes de satisfaction de contraintes (classiques ou valués).

Une première partie de cette thèse est consacrée à la définition et à l'exploitation d'informations explicitées grâce à la structure du problème. Ces informations se traduisent par la notion de good et de nogood structurels que nous proposons. Un good (respectivement un nogood) structurel est une affectation consistante qui peut être (resp. ne peut pas être) étendue en une affectation consistante sur une partie bien déterminée du problème. Nous définissons alors une méthode énumérative, nommée BTD (pour Backtracking sur Tree-Decomposition), qui produit et exploite ces goods et nogoods structurels. Elle repose sur une recherche énumérative et sur l'exploitation de la structure (grâce à la notion de décomposition arborescente [RS86]). L'objectif de cette méthode est de bénéficier des avantages à la fois des approches énumératives (pour leur efficacité pratique) et des approches par décomposition (pour leur borne de complexité en temps). L'intérêt pratique d'une telle méthode est d'ailleurs mis en avant par des expérimentations. Dans un second temps, nous généralisons naturellement ce premier travail au cadre des CSPs valués.

La seconde partie de ce manuscrit porte sur les méthodes concurrentes coopératives, la coopération étant basée sur l'échange d'informations explicitées durant la résolution. Employer une méthode concurrente permet d'éluder au moins partiellement la question du choix d'une heuristique pour guider la recherche. Cette question est généralement délicate puisque l'efficacité des méthodes énumératives est due en partie aux heuristiques employées. Aussi, sachant qu'un mauvais choix

peut fortement pénaliser une méthode, la réponse à cette question se révèle donc cruciale. Devant la difficulté à choisir entre plusieurs heuristiques, la solution prônée par la concurrence consiste à lancer plusieurs solveurs indépendants dotés chacun d'une heuristique différente. La recherche s'arrête alors dès qu'un solveur a trouvé une solution ou a prouvé qu'il n'en existe aucune. De telles approches présentent souvent des résultats satisfaisants pour les problèmes possédant des solutions. Par contre, pour les problèmes sans solution, les gains sont faibles, voire même inexistants. Une parade à ce défaut peut consister à ajouter une forme de coopération entre les solveurs. Cependant, l'ajout de la coopération peut aussi engendrer quelques problèmes. D'une part, l'usage de la coopération est susceptible de modifier le comportement de l'algorithme employé. D'autre part, pour être, efficace, la coopération nécessite une certaine proximité des solveurs et de leurs heuristiques tandis que la concurrence, au contraire, réclame la plus grande diversité possible au niveau des heuristiques. Aussi, face à ces besoins antagonistes, l'efficacité d'une méthode concurrente coopérative repose en partie sur le compromis effectué au niveau de la diversité. Malheureusement, la définition d'un tel compromis n'est pas simple.

Dans [MV96], la coopération repose sur un échange de nogoods (i.e. d'affectations ne conduisant pas à une solution). Ce travail soulève alors la question de l'efficacité d'une méthode concurrente coopérative dont la coopération est basée sur l'échange de nogoods. Nous nous efforçons de répondre à cette question en poursuivant ce travail. Nous proposons d'abord trois schémas qui diffèrent les uns des autres par la manière dont sont échangés les nogoods. En particulier, dans deux de ces schémas, nous nous efforçons de limiter les communications de nogoods au strict nécessaire. Nous intégrons ensuite à chaque solveur une phase d'interprétation qui permet de tirer le meilleur parti des nogoods reçus. Puis, nous proposons un compromis possible concernant la diversité des heuristiques. Enfin, l'intérêt pratique d'une telle approche est mis en évidence grâce à une étude expérimentale.

Une telle méthode concurrente coopérative obtient souvent en pratique de très bons résultats. Il paraît donc naturel de vouloir étendre ce travail en échangeant d'autres types d'informations. Aussi, nous présentons d'abord une méthode concurrente avec une coopération basée sur l'échange de goods et de nogoods structurels avant de généraliser ce travail au cadre des CSPs valués.

Cette thèse est organisée selon le plan suivant :

Le chapitre 1 présente le cadre CSP. Nous décrivons d'abord le formalisme CSP. Ensuite, nous présentons les principales méthodes de résolution de CSPs. D'une part, nous rappelons les travaux réalisés dans un cadre séquentiel en présentant essentiellement les approches énumératives et structurelles. D'autre part, nous évoquons les travaux portant sur une résolution parallèle du problème CSP en nous focalisant principalement sur les méthodes de résolution coopératives. Enfin, nous présentons les jeux de tests couramment employés pour évaluer expérimentalement les algorithmes.

Le chapitre 2 est consacré à la définition d'une nouvelle méthode de résolution de CSPs (la méthode BTD) qui est basée à la fois sur une recherche énumérative et sur la notion de décomposition arborescente. Après avoir décrit la méthode, nous présentons des résultats expérimentaux qui mettent en avant l'intérêt de l'approche.

Le chapitre 3 poursuit le travail commencé au chapitre 2, en étendant la méthode BTD aux CSPs valués (VCSPs). Nous dressons d'abord un bref état de l'art du problème VCSP. En particulier, nous présentons le formalisme VCSP ainsi que quelques méthodes de résolution. Puis, nous généralisons la méthode BTD et son cadre de travail aux CSPs valués.

Le chapitre 4 s'attache à déterminer si l'échange de nogoods constitue une forme efficace de coopération. Nous décrivons entre autres trois schémas pour une recherche concurrente avec échange de nogoods. Nous présentons également des résultats expérimentaux.

Le chapitre 5 propose un schéma pour une recherche concurrente avec échange de goods et de nogoods structurels. Ce premier schéma est ensuite étendu au cadre des CSPs valués.

Chapitre 1

Le problème de satisfaction de contraintes

Le principal objectif de cette thèse est de définir et/ou d'étudier des méthodes de résolution du problème de satisfaction de contraintes, ces méthodes étant basées sur des approches structurelles ou coopératives. Dans ce premier chapitre, nous proposons un bref état de l'art concernant le problème de satisfaction de contraintes (CSP) et sa résolution. En aucun cas, nous ne prétendons effectuer ici un état de l'art exhaustif, l'essentiel des rappels concernant des notions ou des méthodes exploitées dans les travaux présentés dans les chapitres suivants. Pour plus de détails sur les travaux existants, on pourra consulter [Jég91, Dec92, Tsa93, DF02].

Nous rappelons d'abord le formalisme introduit par Montanari [Mon74]. Ensuite, nous présentons les principales méthodes employées pour résoudre ce problème. Nous nous intéressons d'abord aux travaux développés dans un cadre séquentiel en limitant essentiellement nos rappels aux approches énumératives et structurelles. Puis, nous évoquons les travaux portant sur une résolution parallèle du problème CSP en nous focalisant principalement sur les méthodes de résolution coopératives. Enfin, nous décrivons les principaux types d'instances CSPs utilisées pour évaluer expérimentalement les méthodes de résolution.

1.1 Le problème CSP

Un problème de satisfaction de contraintes (CSP) se définit par la donnée d'un ensemble de variables et d'un ensemble de contraintes. Chaque variable peut prendre une valeur choisie dans le domaine qui lui est associé. Les contraintes, quant à elles, décrivent les combinaisons autorisées de valeurs pour les variables. L'objectif est d'attribuer une valeur à chaque variable de sorte que toutes les contraintes soient satisfaites. Plus formellement :

Définition 1.1 (instance CSP [Mon74])
Une ***instance CSP*** $\mathcal{P}$ *est un quadruplet* (X, D, C, R) *où :*

- X *est un ensemble* $\{x_1, x_2, \ldots, x_n\}$ *de variables.*
- D *est un ensemble de domaines finis* $\{d_{x_1}, d_{x_2}, \ldots, d_{x_n}\}$ *tel que le domaine* d_{x_i} *soit associé à la variable* x_i *de* X*. Le domaine* d_{x_i} *contient les valeurs que peut prendre la variable* x_i*.*
- C *est un ensemble* $\{c_1, c_2, \ldots, c_m\}$ *de contraintes. Chaque contrainte* c_i *de* C *est définie par l'ensemble de variables sur lesquelles elle porte.*
- R *est un ensemble* $\{r_{c_1}, r_{c_2}, \ldots, r_{c_m}\}$ *de relations tel que la relation* r_{c_i} *soit associée à la contrainte* c_i*. La relation* r_{c_i} *est définie sur* $\prod_{x \in c_i} d_x$*. Elle représente les affectations compatibles*

entre les variables contraintes par c_i.

Notation 1.2 *Dans tout le document, nous noterons n le nombre de variables, m le nombre de contraintes et d la taille du plus grand domaine.*

On appelle **arité** d'une contrainte c le nombre de variables sur lesquelles elle porte (c'est-à-dire $|c|$). Une contrainte est dite **unaire** (respectivement **binaire**) si elle est d'arité un (resp. deux). Dans les autres cas, on la qualifie de **n-aire**. Notons que les contraintes unaires imposent simplement une restriction des domaines des variables correspondantes. On désigne alors par **CSP binaire** tout CSP dont les contraintes sont soit unaires, soit binaires. On qualifie de **général** ou de **n-aire** un CSP dont l'arité des contraintes est quelconque. L'essentiel des travaux réalisés sur le problème CSP porte sur les CSPs binaires. Ce document ne fait pas exception. Par la suite, nous nous focalisons donc essentiellement sur les CSPs binaires.

La définition des relations peut être donnée soit en intention, soit en extension. En intention, les relations peuvent être décrites par une équation, une inéquation, un prédicat ou une fonction booléenne, tandis qu'en extension, elles sont représentées par des listes de tuples autorisés. Dans la suite du document, les définitions, les propriétés, ... sont données pour des relations représentées en extension. On appelle **dureté** d'une contrainte le nombre de tuples qu'elle interdit.

Afin de représenter la structure du problème, on emploie un hypergraphe ou un graphe de contraintes :

Définition 1.3 *Étant donnée une instance CSP $\mathcal{P} = (X, D, C, R)$, l'hypergraphe (X, C) est l'**hypergraphe de contraintes** associé au problème $\mathcal{P}$. Si le CSP $\mathcal{P}$ est binaire, il s'agit d'un graphe, dit **graphe de contraintes**.*

On désigne également le graphe de contraintes sous le terme de **réseau de contraintes**. L'emploi d'un graphe de contraintes permet d'utiliser la terminologie existant en théorie des graphes, comme les notions de voisinage, de degré, ... (voir [Ber87a, Ber87b] pour une description détaillée de la théorie des graphes et des hypergraphes).

Exemple 1.4 *Nous considérons le CSP binaire $\mathcal{P} = (X, D, C, R)$ avec :*

- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$,
- $D = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$ *avec* $d_1 = d_2 = d_3 = d_6 = d_7 = \{a, b, c\}$ *et* $d_4 = d_5 = \{a, b\}$,
- $C = \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}, c_{56}, c_{57}, c_{67}\}$ *avec* $c_{ij} = \{x_i, x_j\}$,
- $R = \{r_{12}, r_{13}, r_{15}, r_{23}, r_{24}, r_{35}, r_{56}, r_{57}, r_{67}\}$.

La définition des relations en intention est la suivante :

- $r_{12} : x_1 \neq x_2$,
- $r_{13} : x_1 \neq x_3$,
- $r_{15} : x_1 \leq x_5$ *(ordre alphabétique)*,
- $r_{23} : x_2 \neq x_3$,
- $r_{24} : x_2 \leq x_4$,
- $r_{35} : x_3 \leq x_5$.
- $r_{56} : x_5 \neq x_6$,
- $r_{57} : x_5 \neq x_7$,
- $r_{67} : x_6 \neq x_7$,

La définition des relations en extension est la suivante :

r_{12}	
x_1	x_2
a	b
a	c
b	a
b	c
c	a
c	b

r_{13}	
x_1	x_3
a	b
a	c
b	a
b	c
c	a
c	b

r_{15}	
x_1	x_5
a	a
a	b
b	b

r_{23}	
x_2	x_3
a	b
a	c
b	a
b	c
c	a
c	b

r_{24}	
x_2	x_4
a	a
a	b
b	b

r_{35}	
x_3	x_5
a	a
a	b
b	b

r_{56}	
x_5	x_6
a	b
a	c
b	a
b	c

r_{57}	
x_5	x_7
a	b
a	c
b	a
b	c

r_{67}	
x_6	x_7
a	b
a	c
b	a
b	c
c	a
c	b

La figure 1.1 représente le graphe de contraintes (X, C) associé à $\mathcal{P}$.

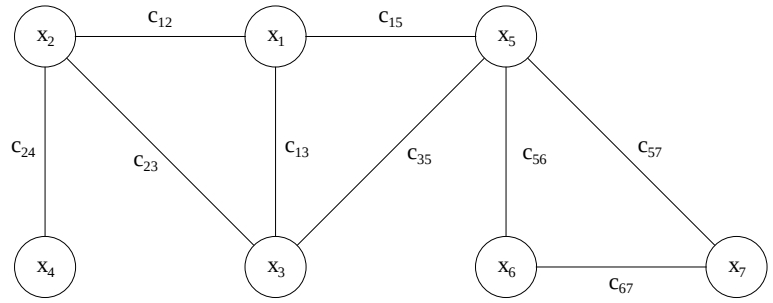

FIG. 1.1 : Le graphe de contraintes associé au CSP de l'exemple 1.4.

Définition 1.5 (instanciation) *Étant donné $Y \subseteq X$ avec $Y = \{y_1, \ldots, y_k\}$, une **instanciation** des variables de Y est un k-uplet $(v_1, \ldots, v_k)$ de $d_{y_1} \times \cdots \times d_{y_k}$. Une instanciation est dite **complète** si elle porte sur toutes les variables de X, **partielle** sinon.*

On emploie également le terme d'**affectation** en lieu et place d'instanciation.

Notation 1.6 *Par la suite, nous écrirons l'instanciation $(v_1, \ldots, v_k)$ sous la forme plus explicite $(y_1 \leftarrow v_1, \ldots, y_k \leftarrow v_k)$. Si $\mathcal{A}$ est une instanciation, on note $X_\mathcal{A}$ l'ensemble des variables affectées dans $\mathcal{A}$. Étant donné un ensemble $Y \subseteq X$, $\mathcal{A}[Y]$ correspond à l'affectation $\mathcal{A}$ restreinte aux variables qui sont à la fois dans Y et dans $X_\mathcal{A}$.*

Par abus de langage, nous désignerons souvent l'instanciation $(y_1 \leftarrow v_1, \ldots, y_k \leftarrow v_k)$ sous la forme ensembliste $\{y_1 \leftarrow v_1, \ldots, y_k \leftarrow v_k\}$, afin de pouvoir exploiter les opérateurs ensemblistes. C'est en particulier le cas pour la notion d'extension d'une affectation :

Définition 1.7 (extension) *Soient $\mathcal{A}$ et $\mathcal{A}'$ deux affectations telles que $X_\mathcal{A} \cap X_{\mathcal{A}'} = \emptyset$. On appelle **extension** de $\mathcal{A}$ par $\mathcal{A}'$ l'affectation $\mathcal{A} \cup \mathcal{A}'$.*

Définition 1.8 *Une affectation $\mathcal{A}$ **satisfait** une contrainte c de C si $\mathcal{A}[c] \in r_c$. Sinon, $\mathcal{A}$ **viole** c. Une instanciation $\mathcal{A}$ est qualifiée de **cohérente** si $\forall c \in C, c \subseteq X_{\mathcal{A}}, \mathcal{A}[c] \in r_c$. Elle est dite **incohérente** sinon.*

Dans la littérature, on emploie souvent le terme de **consistant** (resp. **inconsistant**) en lieu et place de cohérent (resp. incohérent).

Définition 1.9 (solution) *Une **solution** est une instanciation complète consistante.*

Autrement dit, une solution est une affectation de toutes les variables de X avec une valeur prise dans leur domaine respectif, qui satisfait toutes les contraintes. Une instance CSP $\mathcal{P}$ est dite **consistante** si $\mathcal{P}$ possède au moins une solution. Elle est qualifiée d'**inconsistante** sinon.

Exemple 1.10 *Reprenons le CSP de l'exemple 1.4. L'instanciation $\{x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c\}$ est consistante. Par contre, son extension par l'affectation de x_4 à la valeur a (c'est-à-dire l'affectation $\{x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c, x_4 \leftarrow a\}$) est inconsistante puisqu'elle viole la contrainte c_{24}.*
L'instance $\mathcal{P}$ est inconsistante.

Le problème CSP se définit formellement ainsi :

Définition 1.11 (problème CSP) *Le problème de satisfaction de contraintes (CSP) se définit par la donnée d'une instance $\mathcal{P} = (X, D, C, R)$ et par la question : l'instance $\mathcal{P}$ possède-t-elle au moins une solution ?*

Théorème 1.12 *Le problème CSP est NP-complet.*

Une preuve peut être obtenue en utilisant une transformation polynomiale du problème de k-coloration au problème CSP. Étant donné un graphe à colorer, il suffit de construire un CSP tel que :

- les variables soient les sommets du graphe,
- les domaines soient composés des différentes couleurs possibles,
- les contraintes correspondent aux arêtes du graphe,
- les relations soient des relations de différence.

1.2 Résolution séquentielle du problème CSP

La résolution d'une instance CSP consiste soit à trouver une solution (si le problème est consistant), soit à prouver qu'il n'existe pas de solution (si le problème est inconsistant). Les travaux portant sur la résolution séquentielle du problème CSP peuvent être divisés en trois principaux axes de recherche :

- les techniques de simplification par filtrage,
- les algorithmes énumératifs de type backtracking,
- les méthodes de décomposition reposant sur l'exploitation de classes polynomiales.

Nous allons successivement présenter les principaux travaux pour chacun de ces axes. Un quatrième axe peut être dégagé avec les méthodes incomplètes. Ces méthodes recherchent généralement une solution en utilisant différentes techniques (réparation locale, recherche tabou, algorithmes génétiques, ...). Souvent, de telles méthodes produisent des solutions plus rapidement que les algorithmes énumératifs. Cependant, si aucune solution n'est produite, il est impossible de conclure à l'inconsistance du problème. Aussi, comme notre objectif est de résoudre des instances CSPs aussi bien consistantes qu'inconsistantes, nous ne décrirons pas ces méthodes.

1.2.1 Simplification du problème par filtrage

Devant la difficulté du problème CSP, il peut se révéler intéressant de réduire l'espace de recherche à explorer en simplifiant les instances avant d'entamer leur résolution. La simplification par filtrage consiste à supprimer des combinaisons de valeurs qui ne peuvent pas participer à une solution. Les combinaisons de valeurs à supprimer dépendent de la forme de consistance locale employée. De nombreuses formes de consistances locales ont été définies pour les CSPs binaires. Cependant, seules la consistance d'arc [Wal75], la consistance de chemin [Mon74, Mac77] ainsi que des consistances intermédiaires sont vraiment exploitées en pratique.

Définition 1.13 (consistance d'arc) *Soit $\mathcal{P} = (X, D, C, R)$ un CSP.*
Un domaine d_x est ***arc-consistant*** *si $d_x \neq \emptyset$ et $\forall v \in d_x, \forall c = \{x, y\} \in C, \exists w \in d_y, (v, w) \in r_c$.*
$\mathcal{P}$ est dit ***arc-consistant*** *si tous ses domaines sont arc-consistants.*

Le filtrage par consistance d'arc consiste à rendre tous les domaines arc-consistants. Dans ce but, il supprime de chaque domaine d_x toutes les valeurs qui rendent d_x arc-inconsistant. Le filtrage par consistance d'arc est de loin le plus utilisé pour la simplicité de sa mise en œuvre et son efficacité pratique. Il peut être employé comme prétraitement avant la résolution ou pour maintenir le problème arc-consistant durant la résolution. De nombreux algorithmes de filtrage par arc-consistance ont été proposés parmi lesquels AC-3 [Mac77], AC-4 [MH86], AC-6 [Bes94], AC-8 [CJ98] et dernièrement AC-2001 [BR01]. Ils se distinguent les uns des autres par les mécanismes qu'ils mettent en jeu pour établir la consistance d'arc. Des mécanismes employés dépendent alors la facilité de mise en œuvre des algorithmes et leur efficacité pratique. Notons que les algorithmes AC-6 et AC-2001 ont une complexité optimale en temps en $O(md^2)$ pour une complexité en espace en $O(md)$.

La consistance de chemin constitue une forme de consistance plus forte que la consistance d'arc. Dans la définition suivante, s'il n'existe pas de contrainte dans C entre deux variables, on considère alors qu'elles sont liées par une contrainte universelle (i.e. une contrainte qui autorise toutes les combinaisons).

Définition 1.14 (consistance de chemin) *Soit $\mathcal{P} = (X, D, C, R)$ un CSP.*
Une paire de variables (x, y) est ***chemin-consistante*** *si $\forall (v, w) \in r_{\{x,y\}}, \exists u \in d_z, (v, u) \in r_{\{x,z\}}$ et $(w, u) \in r_{\{y,z\}}$.*
$\mathcal{P}$ est dit ***chemin-consistant*** *si toute paire (x, y) de variables est* ***chemin-consistante****.*

Le filtrage par consistance de chemin permet donc de supprimer certains couples de valeurs des relations. Dans l'éventualité où la contrainte correspondante n'existerait pas, celle-ci est alors ajoutée. La consistance de chemin est donc susceptible d'altérer la structure du problème. Notons cependant que le CSP demeure binaire. Comme pour la consistance d'arc, plusieurs algorithmes ont été proposés dont PC-2 [Mac77], PC-4 [HL88], PC-5 [Sin95, Sin96], PC-6 [Chm96] ou PC-8 [CJ98]. Les algorithmes PC-4, PC-5 et PC-6 obtiennent une complexité optimale en $O(n^3d^3)$ avec une complexité en espace en $O(n^3d^3)$ pour PC-4 et en $O(n^3d^2)$ pour PC-5 et PC-6. Ces complexités expliquent que la consistance de chemin est essentiellement utilisée comme prétraitement, le maintien de la chemin-consistance durant une recherche arborescente se révélant trop coûteux.

Ces deux niveaux de consistance ont été généralisés :

Définition 1.15 (k-consistance [Fre78]) *Un CSP est dit* **k-*consistant*** *si toute affectation consistante sur $k-1$ variables peut être étendue en une instanciation consistante sur k variables. Un CSP vérifie la* **k-*consistance forte*** *s'il vérifie la i-consistance pour tout i inférieur ou égal à k.*

La consistance d'arc et la chemin-consistance correspondent respectivement à la 2-consistance et à la 3-consistance. Dans [Coo89], un algorithme optimal pour établir la k-consistance est présenté. Sa complexité est en $O(n^k d^k)$, ce qui explique que la k-consistance pour k supérieur à 3 soit peu usitée. De plus, pour k supérieur à 3, la k-consistance est susceptible de générer des contraintes d'arité $k-1$ et donc de transformer un CSP binaire en CSP n-aire. La notion de i-consistance a elle même fait l'objet d'une généralisation :

Définition 1.16 ((i, j)-consistance [Fre85]) *Un CSP est dit* **(i, j)-*consistant*** *si toute affectation consistante sur i variables peut être étendue en une instanciation consistante sur $i+j$ variables. Un CSP vérifie la* **(i, j)-*consistance forte*** *s'il vérifie la (i', j)-consistance pour tout i' inférieur ou égal à i.*

La k-consistance correspond à la $(k-1, 1)$-consistance. Ces travaux restent cependant essentiellement théoriques. Plus récemment, des travaux ont porté sur la définition de formes de consistance qui soient supérieures à la consistance d'arc mais qui ne modifient pas la structure du problème. On consultera notamment [DB01] pour plus de détails.

1.2.2 Algorithmes énumératifs de résolution

Nous présentons, dans cette partie, les principales méthodes énumératives de résolution du problème CSP. Nous détaillons d'abord l'algorithme Backtrack, qui constitue la méthode de base. Puis, nous décrivons plusieurs techniques pour améliorer cette méthode.

1.2.2.1 Backtrack

L'algorithme Backtrack ou Backtracking Chronologique (noté BT) constitue la méthode de base pour la résolution des CSPs. Il consiste à construire progressivement une solution en affectant les variables une par une et en revenant en arrière à chaque échec rencontré. Plus précisément, étant donnée une affectation consistante $\mathcal{A}$, l'algorithme BT tente d'étendre de façon consistante $\mathcal{A}$. Dans ce but, il choisit une variable x parmi les variables non instanciées et lui attribue une valeur v issue de d_x. L'ordre d'instanciation des variables peut alors être soit prédéfini, soit calculé grâce à une heuristique. Il en est de même pour l'ordre d'instanciation des valeurs. Ainsi, BT étend l'affectation $\mathcal{A}$ en $\mathcal{A} \cup \{x \leftarrow v\}$. Il évalue alors la consistance de l'affectation en vérifiant que $\mathcal{A} \cup \{x \leftarrow v\}$ ne viole aucune contrainte liant x et une variable affectée dans $\mathcal{A}$. Si $\mathcal{A} \cup \{x \leftarrow v\}$ est inconsistante, BT essaie d'étendre $\mathcal{A}$ avec une nouvelle valeur pour x. S'il n'existe plus de valeur possible dans le domaine de x, alors il faut revenir en arrière sur la variable instanciée juste avant x. Si l'affectation $\mathcal{A} \cup \{x \leftarrow v\}$ est consistante, BT cherche à l'étendre comme précédemment. La recherche se poursuit soit jusqu'à ce que toutes les variables aient été instanciées de façon consistante (c'est-à-dire jusqu'à l'obtention d'une solution), soit jusqu'à ce que toutes les possibilités aient été envisagées (afin de prouver l'inconsistance du problème).
L'algorithme BT est décrit à la figure 1.2. $\mathcal{A}$ représente l'affectation courante et V l'ensemble des variables non instanciées. Initialement $\mathcal{A} = \emptyset$ et $V = X$. La complexité en temps de cet algorithme est en $O(md^n)$, dans le pire des cas.

1.2.2.2 Algorithmes avec retour arrière non chronologique

Quand il rencontre un échec, l'algorithme BT remet en cause le dernier choix et essaie alors une nouvelle valeur pour la variable courante x. Lorsque toutes les valeurs ont été utilisées, il revient en arrière sur la variable précédente. Cependant, rien ne garantit que cette variable soit en cause dans les différents échecs rencontrés lors de l'affectation de x. Par exemple, si cette variable n'est pas liée à x par une contrainte, elle ne peut être impliquée dans ces échecs. Par conséquent, essayer de nouvelles valeurs pour cette variable conduira aux mêmes échecs au niveau de l'affectation de

```
   BT(A, V)
1. If V = ∅ Then Return True
2. Else
3.    Choisir x ∈ V
4.    d ← d_x
5.    Consistance ← False
6.    While d ≠ ∅ and ¬Consistance Do
7.       Choisir v dans d
8.       d ← d\{v}
9.       If ∄c ∈ C telle que c viole A ∪ {x ← v}
10.      Then Consistance ← BT(A ∪ {x ← v}, V\{x})
11.      EndIf
12.   EndWhile
13.   Return Consistance
14. EndIf
```

FIG. 1.2 : L'algorithme BT.

x. Pour pallier ce défaut de BT, plusieurs méthodes dotées de retour arrière non chronologique ont vu le jour. Le retour arrière non chronologique consiste à analyser les causes des échecs et à revenir sur la variable la plus profonde qui est en cause dans l'échec. C'est la notion de **saut en arrière** ou **backjump**. La façon d'analyser les causes des échecs détermine alors l'algorithme. Parmi les algorithmes exploitant une technique de retour arrière non chronologique, on dénombre :

- l'algorithme Backjumping (noté BJ [Gas79]) : lorsque toutes les extensions de l'affectation courante avec une valeur de x sont inconsistantes, BJ revient en arrière sur la variable la plus profonde dans l'arbre dont la valeur est en conflit avec une valeur de x.
- l'algorithme Graph-based Backjumping (noté GBJ [Dec90]) : le retour arrière repose sur le graphe de contraintes. Si l'affectation courante ne peut être étendue de façon consistante à une variable x, alors GBJ revient en arrière sur la variable la plus profonde du voisinage de x. Soit y cette variable. Si toutes les valeurs du domaine de y ont été essayées, GBJ revient sur la variable la plus profonde qui appartienne soit au voisinage de y, soit au voisinage de toute variable instanciée après y qui soit une cause potentielle de l'échec d'une extension de l'affectation courante (x est en une), et ainsi de suite.
- l'algorithme Conflict-directed Backjumping (noté CBJ [Pro93]) : pour chaque variable instanciée x, CBJ maintient un ensemble dit ensemble des conflits. Cet ensemble contient toutes les variables instanciées avant x avec lesquelles x est en conflit pour au moins une de ses valeurs ainsi que les variables qui sont en cause dans l'échec d'une extension d'une affectation contenant x. Lorsqu'un échec survient ou lorsque toutes les valeurs ont été essayées, on revient en arrière jusqu'à la variable la plus profonde de l'ensemble des conflits de la variable courante.

La principale différence entre BJ et CBJ réside dans le retour en arrière réalisé quand toutes les extensions de l'affectation courante ont été essayées sans succès. BJ se contente de revenir à la variable précédente tandis que CBJ effectue si possible un saut en arrière. Comparé à CBJ, GBJ effectue une surestimation de l'ensemble des conflits de CBJ. En effet, GBJ prend en compte toutes les variables qui sont susceptibles d'être une cause des différents échecs survenus sans tenir compte des conflits réellement rencontrés. Dans tous les cas, ce type de méthode permet de réduire la taille de l'arbre de recherche à explorer en élaguant des sous-arbres redondants. Pour plus de détails sur les algorithmes dotés de retour arrière non chronologique, on pourra consulter [DF02].

1.2.2.3 Algorithmes avec filtrage avant

L'algorithme BT, comme les méthodes avec retour arrière intelligent, teste la consistance de l'affectation courante en vérifiant que $\mathcal{A} \cup \{x \leftarrow v\}$ ne viole aucune contrainte liant x et une variable affectée dans $\mathcal{A}$. Autrement dit, le test de consistance s'effectue en fonction des variables déjà instanciées. C'est le concept de consistance en arrière (ou look-back scheme). Une autre technique consiste à exploiter le concept de consistance en avant (ou look-ahead scheme). Suivant ce concept, à chaque affectation d'une variable x, les valeurs des variables non instanciées qui ne sont pas compatibles avec la valeur de x sont supprimées. Le calcul des valeurs à supprimer dépend alors du niveau de filtrage utilisé.

Un algorithme utilisant ce concept cherche à étendre une affectation consistante $\mathcal{A}$. Dans ce but, il choisit une variable non instanciée et lui affecte une valeur v du domaine d_x. Le choix des variables et des valeurs à instancier fait généralement appel à des heuristiques. Une fois l'affectation $\mathcal{A} \cup \{x \leftarrow v\}$ construite, il supprime du domaine de chaque variable non affectée les valeurs qui ne sont pas compatibles avec v selon le niveau de filtrage utilisé. Si un domaine devient vide, alors l'affectation $\mathcal{A} \cup \{x \leftarrow v\}$ ne possède pas d'extension consistante. Il faut alors essayer une nouvelle valeur pour x. Si toutes les valeurs ont été essayées, l'algorithme revient en arrière sur la variable affectée juste avant x. Si tous les domaines possèdent au moins une valeur, alors l'algorithme tente d'étendre $\mathcal{A} \cup \{x \leftarrow v\}$ en procédant comme précédemment. La recherche se termine quand toutes les variables sont instanciées (découverte d'une solution) ou si toutes les possibilités ont été étudiées (preuve de l'inconsistance du problème). Notons que ces méthodes ne construisent que des affectations consistantes. Les branches de l'arbre de recherche qui, pour BT, correspondent à des affectations inconsistantes sont en fait élaguées grâce au filtrage.

L'algorithme Forward Checking (noté FC [HE80]) emploie ce concept. Le filtrage consécutif à l'affectation de la variable x avec la valeur v consiste à supprimer du domaine de chaque variable y non affectée du voisinage de x les valeurs qui violent la contrainte liant x et y. L'algorithme FC est décrit à la figure 1.3. $\mathcal{A}$ représente l'affectation courante et V l'ensemble des variables non instanciées. Initialement $\mathcal{A} = \emptyset$ et $V = X$. La fonction Forward-Check réalise le filtrage. Elle renvoie $True$ si le filtrage n'engendre pas de domaine vide, $False$ sinon. La fonction Unforward annule le filtrage consécutif à l'affectation de la variable x La complexité en temps de FC dans le pire des cas est identique à celle de BT, c'est-à-dire en $O(md^n)$. Par contre, en pratique, FC obtient généralement de meilleurs résultats que BT.

L'algorithme FC limite la suppression des valeurs au voisinage de la dernière variable instanciée. D'autres algorithmes propagent en plus les suppressions. Par exemple, l'algorithme Maintaining Arc-Consistency (noté MAC [SF94]) applique un filtrage par arc-consistance après chaque instanciation. La suppression d'une valeur peut alors entraîner la suppression d'autres valeurs. Par rapport à BT ou à FC, de telles méthodes développent généralement des arbres de recherche de taille réduite. Cependant, selon la nature du filtrage, maintenir un certain niveau de consistance à chaque nœud de l'arbre peut se révéler plus coûteux en temps que d'utiliser un algorithme plus simple comme FC.

Récemment, dans [Bac00], une extension de l'algorithme FC a été proposée. Elle permet d'économiser certains tests de contrainte grâce à un filtrage plus judicieux.

1.2.2.4 Algorithmes avec mémorisation

Les algorithmes avec retour arrière non chronologique permettent d'éviter certaines redondances dans l'arbre de recherche. Toutefois, il subsiste tout de même des redondances. Aussi, pour ne pas visiter plusieurs fois les mêmes sous-arbres, une solution consiste à mémoriser des informations portant sur le travail déjà réalisé. Ces informations sont généralement mémorisées sous la forme de contraintes induites, qui sont souvent désignées sous le terme de **nogood**. Un nogood correspond à

1. FC($\mathcal{A}, V$)
2. **If** $V = \emptyset$ **Then** Return **True**
24. **Else**
25. Choisir $x \in V$
26. $d \leftarrow d_x$
27. $Consistance \leftarrow$ **False**
28. **While** $d \neq \emptyset$ **and** $\neg Consistance$ **Do**
29. Choisir v dans d
30. $d \leftarrow d \backslash \{v\}$
31. **If** Forward-check($\mathcal{A} \cup \{x \leftarrow v\}, x$)
32. **Then** $Consistance \leftarrow$ FC($\mathcal{A} \cup \{x \leftarrow v\}, V \backslash \{x\}$)
11. Unforward(x)
33. **EndIf**
34. **EndWhile**
35. Return $Consistance$
36. **EndIf**

FIG. 1.3 : L'algorithme FC.

une affectation qui ne peut être étendue en une solution. Leur mémorisation permet donc d'interdire certaines affectations et ainsi d'éviter de reproduire plusieurs fois les mêmes sous-arbres.

Plusieurs méthodes mémorisant des informations ont été proposées, parmi lesquelles :

- Value-based shallow learning, Graph-based shallow learning et Deep learning [Dec90]),
- Jump-back learning [FD94],
- Nogood Recording (noté NR [SV93, SV94]),
- Dynamic Backtracking (noté DBT [Gin93]),
- Learning-Tree-Solve [BM96].

L'algorithme Nogood Recording sera décrit plus précisément dans le chapitre 4, dans sa version hybridée avec FC. Notons que l'algorithme Learning-Tree-Solve mémorise des nogoods, mais aussi des goods (i.e. des affectations consistantes qui peuvent être étendues de façon consistante sur une partie bien déterminée du problème).

Le principal défaut de ces méthodes est la quantité de mémoire requise pour mémoriser toutes les contraintes induites produites, le nombre de nogoods étant potentiellement exponentiel. Une parade consiste alors soit à limiter l'arité des contraintes mémorisées, soit à ne les conserver que temporairement. Les cinq premières méthodes optent pour la première solution. Quant à l'algorithme DBT, il exploite la seconde. Enfin, Learning Tree-Solve est proposée en deux versions, l'une limitant l'arité des goods ou des nogoods, l'autre leur durée de conservation.

1.2.2.5 Algorithmes hybrides

Nous venons de décrire les trois principaux types d'améliorations de l'algorithme BT. À partir d'algorithmes provenant de deux voire de trois de ces types, il est possible de produire de nombreux algorithmes hybrides, parmi lesquels :

- FC-CBJ [Pro93] ou MAC-CBJ [Pro95] pour l'hybridation entre retour arrière non chronologique et filtrage,
- NR (qui intègre sans surcoût un retour arrière similaire à celui de CBJ) pour l'hybridation entre retour arrière non chronologique et mémorisation,
- FC-NR [SV93, SV94] ou MAC-DBT [JDB00] pour l'hybridation entre filtrage et mémorisation,

- FC-NR (qui exploite un retour arrière similaire à celui de CBJ) pour l'hybridation entre retour arrière non chronologique, filtrage et mémorisation.

Cependant, un inconvénient majeur pour l'hybridation est que les améliorations de BT ne sont pas totalement indépendantes. Par exemple, l'apport réalisé par le filtrage peut restreindre voire annihiler la contribution du retour arrière non chronologique ou de la mémorisation. En particulier, d'après [CvB01], l'emploi d'un retour arrière non chronologique comme CBJ constitue d'autant moins une amélioration qu'un niveau de consistance élevé est maintenu.

1.2.2.6 L'algorithme Forward-Checking avec Nogood Recording

L'algorithme FC-NR constitue l'algorithme de base de la méthode coopérative étudiée au chapitre 4. Aussi, nous allons décrire plus en détails cet algorithme. Nous rappelons d'abord les principales définitions et propriétés concernant les nogoods [SV93].

Les nogoods : définitions et propriétés
Un nogood correspond à une affectation qui ne peut être étendue en une solution. Plus formellement :

Définition 1.17 (nogood [SV93])
Soient $\mathcal{A}$ une affectation et J un sous-ensemble de contraintes ($J \subseteq C$).
*$(\mathcal{A}, J)$ est un **nogood** si le CSP (X, D, J, R) ne possède pas de solution contenant $\mathcal{A}$. J est appelé la **justification** du nogood (on note X_J les variables soumises aux contraintes de J). On désigne par **arité** du nogood $(\mathcal{A}, J)$ le nombre de variables affectées dans $\mathcal{A}$.*

Remarque 1.18 *Toute affectation inconsistante correspond à un nogood. La réciproque est fausse.*

On peut également écrire la définition des nogoods sous la forme : $(\mathcal{A}, J)$ est un nogood si le CSP (X, D, J, R) ne possède pas de solution S telle que $S[X_{\mathcal{A}}] = \mathcal{A}$.

Exemple 1.19 *Si nous reprenons le CSP de l'exemple 1.4, $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$ est un nogood bien que l'affectation $\{x_1 \leftarrow a, x_3 \leftarrow c\}$ soit consistante. $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$ est aussi un nogood.*

Pour des raisons d'efficacité pratique, notre objectif est d'employer l'algorithme Nogood Recording doté du filtrage de Forward-Checking. Aussi, les définitions et théorèmes suivants sont adaptés afin d'en tenir compte.
Quand on applique un filtrage durant la recherche, une inconsistance est détectée dès qu'un domaine devient vide. Par conséquent, les causes de l'inconsistance peuvent être multiples. Aussi, pour être en mesure de calculer les justifications des nogoods, Schiex et Verfaillie ont introduit la notion de "*value-killer*" ([SV93]). Nous rappelons cette notion et nous l'étendons (ajout du point (iii)) afin de pouvoir l'exploiter ensuite dans la méthode concurrente coopérative étudiée au chapitre 4. Pour définir formellement la notion de value-killer, nous avons besoin de la notion de *CSP induit* :

Définition 1.20 (CSP induit)
Soient $\mathcal{P} = (X, D, C, R)$ un CSP et $\mathcal{A}_i$ une affectation ($\mathcal{A}_i = \{x_1 \leftarrow v_1, x_2 \leftarrow v_2, \ldots, x_i \leftarrow v_i\}$).
*$\mathcal{P}(\mathcal{A}_i) = (X, D(\mathcal{A}_i), C, R(\mathcal{A}_i))$ est le **CSP induit** par $\mathcal{A}_i$ sur $\mathcal{P}$ avec un filtre de type Forward-Checking avec :*

- $\forall j, 1 \leq j \leq i,\ d_{x_j}(\mathcal{A}_i) = \{v_j\}$
- $\forall j, i < j \leq n,\ d_{x_j}(\mathcal{A}_i) = \{v_j \in d_{x_j} | \forall c_{kj} = \{x_k, x_j\} \in C, 1 \leq k \leq i, (v_k, v_j) \in r_{c_{kj}}\}$

- $\forall j, j',\ r_{c_{jj'}}(\mathcal{A}_i) = r_{c_{jj'}} \cap (d_{x_j}(\mathcal{A}_i) \times d_{x_{j'}}(\mathcal{A}_i))$.

L'affectation $\mathcal{A}_i$ est dite ***FC-consistante*** *si* $\forall j, d_{x_j}(\mathcal{A}_i) \neq \emptyset$.

D'après cette définition, $d_{x_i}(\mathcal{A})$ représente le domaine courant de x_i (après les différents filtrages inhérents à la construction de $\mathcal{A}$), alors que d_{x_i} est le domaine initial de x_i. De même, $r_{c_{jj'}}(\mathcal{A}_i)$ correspond à la relation initiale $r_{c_{jj'}}$ réduite aux domaines courants. Une value-killer correspond à une contrainte qui autorise la suppression d'une valeur par filtrage. Cette contrainte peut être une contrainte initiale du problème (point (ii)) ou une contrainte ajoutée sous la forme d'un nogood (point (iii)). Plus précisément :

Définition 1.21 (value-killer)
Soient $\mathcal{A}_i$ une affectation et $\mathcal{P}(\mathcal{A}_i)$ le CSP induit par $\mathcal{A}_i$.
Soit N l'ensemble des nogoods trouvés par les différents solveurs. Une contrainte c_{kj} $(j > i \geq k)$ est une ***value-killer*** *de la valeur v_j de d_{x_j} pour $\mathcal{A}_i$ si une des trois conditions suivantes est vérifiée :*

(i) c_{kj} *est une value-killer de* v_j *pour* $\mathcal{A}_{i-1}$

(ii) $k = i$ *et* $(v_k, v_j) \notin r_{c_{kj}}(\mathcal{A}_i)$ *et* $v_j \in d_{x_j}(\mathcal{A}_{i-1})$

(iii) $\{x_k \leftarrow v_k, x_j \leftarrow v_j\} \in N$.

L'ensemble des value-killers du domaine d_{x_j} est défini comme l'ensemble des contraintes qui sont une value-killer pour au moins une valeur v_j de d_{x_j}.

Pour une valeur v donnée, une value-killer de v est donc une contrainte qui permet de supprimer v par filtrage, et par la même, d'expliquer la suppression de v. Ainsi, si on suppose qu'une inconsistance est détectée car un domaine d_{x_i} est devenu vide, alors l'ensemble des value-killers de d_{x_i} constitue les raisons de l'inconsistance (i.e. les justifications). On peut alors énoncer le théorème suivant qui formalise la création des nogoods à partir des inconsistances détectées :

Théorème 1.22 *Soient $\mathcal{A}$ une affectation et x_i une variable non affectée. Soit K l'ensemble des value-killers de d_{x_i}. S'il n'existe plus de valeurs dans $d_{x_i}(\mathcal{A})$, alors $(\mathcal{A}[X_K], K)$ est un nogood.*

Ce théorème permet donc de créer des nogoods à partir des inconsistances détectées. Quant aux deux théorèmes suivants, ils rendent possible la création de nouveaux nogoods à partir de nogoods existants. Le premier théorème construit un nouveau nogood à partir d'un nogood existant.

Théorème 1.23 (projection de nogood [SV93]) *Si $(\mathcal{A}, J)$ est un nogood, alors $(\mathcal{A}[X_J], J)$ est un nogood.*

Autrement dit, on ne conserve de l'affectation que les variables réellement mises en cause dans l'échec. Ainsi, le nogood généré possède une arité réduite à son strict minimum. Du fait de cette arité limitée, on pourra considérer ce nouveau nogood comme meilleur par rapport à celui de départ.
Le théorème 1.24 construit un nouveau nogood à partir d'un ensemble de nogoods.

Théorème 1.24 *Soient $\mathcal{A}$ une affectation et x_i une variable non affectée. Soit K l'ensemble des value-killers de d_{x_i}. Soit $\mathcal{A}_j$ l'extension de $\mathcal{A}$ par l'affectation de x_i à la valeur v_j $(\mathcal{A}_j = \mathcal{A} \cup \{x_i \leftarrow v_j\})$.*
Si $(\mathcal{A}_1, J_1)$, ..., $(\mathcal{A}_{|d_{x_i}|}, J_{|d_{x_i}|})$ sont des nogoods, alors $(\mathcal{A}, K \cup \bigcup_{j=1}^{|d_{x_i}|} J_j)$ est un nogood.

Grâce à ce théorème, quand on a essayé toutes les extensions d'une branche, sans succès, on peut construire un nouveau nogood.

Les nogoods peuvent être utilisés de deux façons :

- pour ajouter une nouvelle contrainte ou pour renforcer une contrainte existante,
- pour backjumper.

Si on découvre le nogood $(\mathcal{A}, J)$, une contrainte liant les variables de $X_{\mathcal{A}}$ est ajoutée (ou renforcée si elle existe déjà) en interdisant le tuple $\mathcal{A}$. Cette contrainte peut alors être exploitée au même titre qu'une contrainte du problème initial, par exemple pour supprimer des valeurs lors du filtrage. Quant au backjump provoqué par un nogood, il est similaire à celui de l'algorithme CBJ [Pro93]. Le lemme suivant caractérise la phase de backjump :

Lemme 1.25 *Si $(\mathcal{A}[X_J], J)$ est un nogood, alors il est correct de revenir en arrière jusqu'à la variable la plus profonde de X_J.*

Bien entendu, lorsque nous disposons de plusieurs nogoods $(\mathcal{A}[X_J], J)$ portant sur l'affectation courante, il faut revenir en arrière jusqu'à la variable la plus profonde parmi tous les X_J.
Dans les deux cas, il résulte de l'utilisation des nogoods un élagage de l'arbre de recherche. En particulier, leur emploi permet d'éviter de visiter certaines parties redondantes de l'arbre de recherche.

L'algorithme Nogood Recording L'algorithme FC-NR est décrit dans la figure 1.4. Il explore l'arbre de recherche à la manière de Forward-Checking. Il tente d'étendre une affectation FC-consistante $\mathcal{A}$. Dans ce but, FC-NR choisit une nouvelle variable x_i dans V et essaie de lui affecter une valeur v afin de construire une extension $\mathcal{A}' = \mathcal{A} \cup \{x_i \leftarrow v\}$ de $\mathcal{A}$. L'application d'un filtre de type Forward-Checking sur les variables non affectées voisines de x_i détermine si $\mathcal{A}'$ est FC-consistante. La fonction Forward-check réalise ce travail. Elle renvoie $\emptyset$ si l'affectation est FC-consistante, l'ensemble des contraintes responsables de l'inconsistance sinon. Quant à la fonction Unforward, elle permet d'annuler le filtrage consécutif à l'affectation de x_i.
Lors du parcours, FC-NR profite des inconsistances et des échecs rencontrés pour créer et mémoriser des nogoods. Ces nogoods sont alors utilisés comme décrit ci-dessus pour élaguer l'arbre de recherche. Pour réaliser ce travail, on introduit les ensembles de contraintes J et J' qui correspondent aux causes des échecs rencontrés lors des tentatives d'extension respectivement de $\mathcal{A}$ et de $\mathcal{A}'$.
L'inconvénient d'un algorithme comme FC-NR est que le nombre de nogoods est potentiellement exponentiel (car il est minoré par le nombre d'affectations menant à un échec). Nous limiterons donc le nombre de nogoods en suivant la proposition de Schiex et Verfaillie ([SV93]). Cette proposition consiste à ne mémoriser que les nogoods d'arité au plus i. Pour notre part, nous fixerons i à 2, ce qui permet d'obtenir un nombre de nogoods raisonnable. De plus, cette valeur de i présente l'avantage que le CSP demeure binaire.

1.2.3 Heuristiques

Lorsqu'on emploie une méthode énumérative pour résoudre le problème CSP, l'ordre d'instanciation des variables utilisé a une grande influence sur la taille de l'arbre de recherche (et donc sur la qualité de la méthode). Il est en de même, dans une moindre mesure, pour l'ordre d'instanciation des valeurs. Le choix d'un bon ordre d'instanciation dépend essentiellement du problème traité. Mais, même pour un problème donné, il n'est pas toujours évident de déterminer un ordre de bonne qualité (i.e. un ordre conduisant à un arbre de recherche de taille restreinte). Aussi, le choix de la prochaine variable (ou valeur) à instancier fait généralement appel à une heuristique.

1.2.3.1 Heuristiques de choix de variables

En général, les heuristiques de choix de variables couramment utilisées ont pour objectif de rencontrer l'échec le plus tôt possible (*fail-first principle*). Le but est bien sûr de réduire la taille de l'arbre de recherche.

FC-NR($\mathcal{A}, V$)
Entrées : une affectation FC-consistante $\mathcal{A}$ et V l'ensemble des variables non affectées. Initialement, $\mathcal{A} = \emptyset$ et $V = X$.
Sorties : $\emptyset$ si le problème est consistant ($\mathcal{A}$ est la solution trouvée), l'ensemble des justifications de l'échec sinon.

1. **If** $V = \emptyset$ **Then** $\mathcal{A}$ est une solution, Stop
2. **Else**
3. Choisir $x_i \in V$
4. $d \leftarrow d_{x_i}(\mathcal{A})$
5. $J \leftarrow \emptyset$
6. $BackJump \leftarrow False$
7. **While** $d \neq \emptyset$ **and** $BackJump = False$
8. Choisir $v \in d$
9. $d \leftarrow d - \{v\}$
10. $\mathcal{A}' \leftarrow \mathcal{A} \cup \{x_i \leftarrow v\}$
11. $K \leftarrow$ Forward-check$(\mathcal{A}', x_i)$
12. **If** $K = \emptyset$
13. **Then**
14. $J_{sons} \leftarrow$ FC-NR$(\mathcal{A}', V - \{x_i\})$
15. Unforward(x_i)
16. **If** $x_i \in X_{J_{sons}}$ /* *Si x_i est une cause de l'échec de $\mathcal{A}'$* */
17. **Then** $J \leftarrow J \cup J_{sons}$
18. **Else**
19. $J \leftarrow J_{sons}$
20. $BackJump \leftarrow vrai$
21. **EndIf**
22. **Else**
23. Unforward(x_i)
24. $J \leftarrow J \cup K$
25. Mémoriser $(\mathcal{A}'[X_K], K)$ /* *théorème 1.22* */
26. **EndIf**
27. **EndWhile**
28. **If** $BackJump = False$
29. **Then**
30. $J \leftarrow J \cup value\text{-}killer(x_i)$
31. Mémoriser $(\mathcal{A}[X_J], J)$ /* *théorèmes 1.23 et 1.24* */
32. **EndIf**
33. Return J
34. **EndIf**

FIG. 1.4 : L'algorithme Forward-Checking avec Nogood Recording (FC-NR).

Il existe deux types d'heuristiques pour l'ordonnancement des variables : les heuristiques statiques et les heuristiques dynamiques. Les heuristiques statiques consistent à calculer un ordre pour les variables avant d'entamer la résolution du problème. Elles reposent essentiellement sur des propriétés structurelles du problème à résoudre. Parmi ces heuristiques statiques, on dénombre :

- l'heuristique *md* [DM89] : elle ordonne les variables suivant l'ordre décroissant des degrés des variables (i.e. du nombre de contraintes portant sur chaque variable).
- l'heuristique *mst* : on choisit les variables dans l'ordre décroissant de la somme des duretés des contraintes pesant sur chaque variable. Il s'agit d'un raffinement de l'heuristique *md*.
- l'heuristique *mc* [DM89] : on sélectionne la variable qui possède le plus grand nombre de voisines instanciées. Pour la première variable, un choix arbitraire est effectué.
- l'heuristique *mw* [Fre82] : on ordonne les variables dans l'ordre inverse d'un ordre de largeur minimum.

Les heuristiques statiques sont principalement utiles pour des algorithmes dépourvus de filtrage avant comme BT. Par contre, dès qu'on exploite du filtrage, les heuristiques dynamiques produisent de meilleurs résultats. Elles tirent parti de l'existence de tailles de domaine différentes, ces différences étant créées ou accrues par l'emploi du filtrage. L'heuristique *dom* [HE80] a été la première heuristique dynamique proposée. Elle choisit comme prochaine variable à instancier une variable qui possède un domaine de taille minimum. L'inconvénient majeur de cette heuristique est qu'elle ne tire pas profit de la structure. Aussi, elle a été dans un premier temps couplée avec les heuristiques statiques présentées ci-dessus. Ce couplage consiste à employer d'abord l'heuristique *dom*, puis, s'il existe plusieurs variables possédant un domaine de taille minimum, à les départager en exploitant une des quatre heuristiques statiques. Par exemple, l'heuristique $dom + deg$ [FD95] exploite d'abord *dom*, puis elle choisit la variable de plus grand degré parmi les éventuelles variables ex aequo. Le défaut de telles heuristiques est de n'exploiter la structure que s'il existe des variables ex aequo. Ainsi, pour éviter ce problème, des heuristiques exploitant à part égale la structure et la taille des domaines ont été définies. Par exemple, l'heuristique dom/deg [BR96] qui choisit comme prochaine variable à instancier une variable x qui minimise le rapport $\frac{|d'_x|}{|\Gamma_x|}$ (avec d'_x le domaine courant de x et Γ_x l'ensemble des variables voisines de x). Une variante $dom/futdeg$ de cette heuristique peut être définie en considérant l'ensemble des variables voisines non instanciées. Pour notre part, nous exploiterons également un raffinement de cette heuristique. Il s'agit de l'heuristique dom/st qui sélectionne comme prochaine variable à instancier une variable x qui minimise le rapport $\frac{|d'_x|}{\Sigma_x}$ (Σ_x la somme des duretés des contraintes impliquant x). Enfin, récemment, des heuristiques exploitant un voisinage étendue à plusieurs niveaux ont été proposées dans [BCS01a, BCS01b]. Elles généralisent entre autres les heuristiques $dom + deg$ et dom/deg qui exploitent un voisinage de niveau 1.

1.2.3.2 Heuristiques de choix de valeurs

Les heuristiques de choix de valeurs ont fait l'objet de peu d'études. Leur apport est restreint aux problèmes consistants à condition de ne rechercher que la première solution. Dans les autres cas, l'apport est inexistant puisqu'il est nécessaire d'essayer toutes les valeurs. Cet apport limité explique certainement le peu d'intérêt suscité par les choix d'ordonnancement des valeurs. La principale étude est proposée dans [FD95]. Elle définit quatre heuristiques dont la meilleure est incontestablement $min-conflicts$. Pour chaque valeur v de la variable courante, cette heuristique compte le nombre de valeurs des variables non instanciées avec lesquelles v est incompatible. Elle sélectionne ensuite les valeurs dans l'ordre croissant des nombres de conflits.

1.2.4 Résolution par décomposition

Les méthodes de résolution par décomposition du problème se divisent en deux grandes catégories :

- les méthodes de décomposition basées sur le graphe de contraintes,
- les méthodes de décomposition basées sur la micro-structure.

Nous allons décrire successivement ces deux types de méthodes.

1.2.4.1 Méthodes de décomposition basées sur le graphe de contraintes

Parmi tous les algorithmes de résolution de CSPs, seules les méthodes de décomposition basées sur le graphe de contraintes fournissent des garanties en terme de complexité théorique avant la résolution d'un problème. Elles procèdent en isolant des parties *a priori* intraitables en temps polynomial, pour parvenir à une seconde étape qui garantira un temps de résolution polynomial. En général, ces méthodes exploitent les propriétés topologiques du graphe de contraintes et sont basées sur la notion de *décomposition arborescente* (tree-decomposition) de graphes [RS86], dont la définition est rappelée ci-dessous :

Définition 1.26 *Soit $G = (X, E)$ un graphe. Une **décomposition arborescente** de G est une paire $(\mathcal{C}, \mathcal{T})$ avec $\mathcal{T} = (I, F)$ un arbre et $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque $\mathcal{C}_i$ correspond à un nœud de $\mathcal{T}$ et vérifie :*

(1) $\bigcup_{i \in I} \mathcal{C}_i = X$,

(2) pour toute arête $\{x, y\} \in E$, il existe $i \in I$ avec $\{x, y\} \subseteq \mathcal{C}_i$, et

(3) pour tout $i, j, k \in I$, si k est sur un chemin de i à j dans $\mathcal{T}$, alors $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$.

La largeur d'une décomposition arborescente $(\mathcal{C}, \mathcal{T})$ est égale à $\max_{i \in I} |\mathcal{C}_i| - 1$. La tree-width d'un graphe G est la largeur minimale sur toutes les décompositions arborescentes de G.

Les éléments $\mathcal{C}_i$ de $\mathcal{C}$ sont généralement appelés **regroupements** ou **clusters**. Par abus de langage, nous assimilerons les éléments de I aux clusters auxquels ils sont associés.
Notons, pour le lecteur qui n'est pas familier avec ces notions, que la définition d'un arbre $\mathcal{T} = (I, F)$ fait intervenir un ensemble F d'arêtes. Cet ensemble est nécessaire pour satisfaire la partie (3) de la définition 1.26.

La complexité du problème de recherche d'une décomposition arborescente est NP-dur [ACP87]. Toutefois, de nombreux travaux ont été développés dans cette direction [BG01]. Ceux-ci sont fréquemment basés sur l'exploitation de la notion de graphe *triangulé* (voir [Gol80] pour une introduction plus détaillée aux graphes triangulés) :

Définition 1.27 *Un graphe $G = (X, E)$ est dit **triangulé** s'il ne possède pas de cycle de longueur 4 ou plus sans corde (c'est-à-dire sans arête joignant deux sommets non consécutifs dans le cycle).*

Les liens entre graphes triangulés et décompositions arborescentes sont évidents. En effet, étant donné un graphe triangulé, l'ensemble de ses cliques maximales $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}$ de (X, E) correspond à la famille de sous-ensembles associée à cette décomposition. Comme un graphe quelconque $G = (X, E)$ n'est pas nécessairement triangulé, une décomposition arborescente peut être approximée en triangulant G. Nous appelons **triangulation** l'ajout à G d'un ensemble E' d'arêtes de telle sorte que $G' = (X, E \cup E')$ soit triangulé. La largeur d'une triangulation G' du graphe G est égale à la taille maximum des cliques moins un dans le graphe résultant G'. La tree-width de G est alors égale à la largeur minimale pour toutes les triangulations.

Exemple 1.28 *Considérons le graphe représenté à la figure 1.5. Ce graphe n'est pas triangulé, mais une triangulation possible pour ce graphe est fournie à la figure 1.6 ; la taille maximum des cliques est 4 (figure 1.7). Cette triangulation étant optimale, la tree-width de ce graphe est donc 3. Dans la figure 1.8, l'arbre dont les nœuds correspondent aux cliques maximales du graphe triangulé constitue une décomposition arborescente possible pour le graphe de la figure 1.5. Nous*

avons ainsi $\mathcal{C}_1 = \{A, B, C, D\}$, $\mathcal{C}_2 = \{C, D, E\}$, $\mathcal{C}_3 = \{E, F, G\}$, $\mathcal{C}_4 = \{C, D, H\}$, $\mathcal{C}_5 = \{D, H, I\}$, $\mathcal{C}_6 = \{H, I, J\}$, $\mathcal{C}_7 = \{H, J, K\}$, $\mathcal{C}_8 = \{B, D, L, M\}$, $\mathcal{C}_9 = \{L, M, N\}$ *et* $\mathcal{C}_{10} = \{M, N, O\}$.

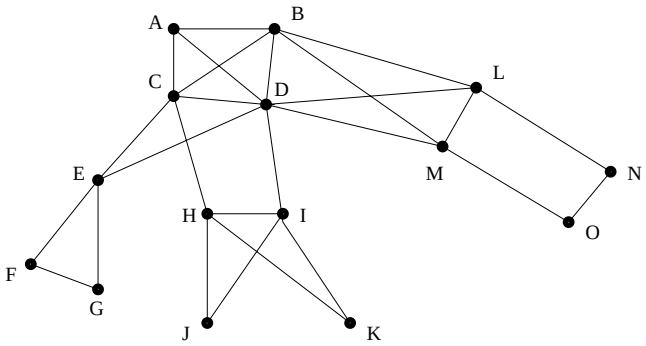

FIG. 1.5 : Un graphe de contraintes sur 15 variables.

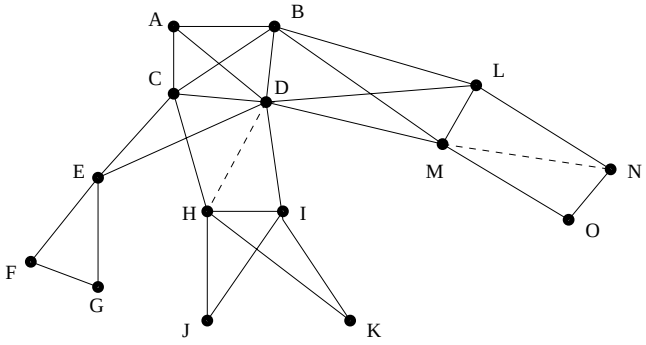

FIG. 1.6 : Le graphe de la figure 1.5 après une triangulation (lignes pointillées).

La méthode de décomposition de CSPs appelée Tree-Clustering, proposée par Dechter et Pearl [DP89] et notée TC, est basée sur ces notions (voir également [DF01] pour une description plus récente). Elle procède en 4 étapes. La première réalise la triangulation du graphe de contraintes (une discussion sur les différents algorithmes de triangulation est proposée dans la partie 2.3.2.2). La seconde calcule les cliques maximales du graphe triangulé (chaque clique correspond alors à un sous-problème). La troisième étape résout les différents sous-problèmes obtenus, et la dernière étape est constituée par la résolution du nouveau CSP général acyclique. L'idée directrice de cette méthode est de fournir un schéma systématique qui, pour tout CSP, permet de produire un CSP équivalent par un recouvrement de l'ensemble des contraintes avec pour objectif de construire un hypergraphe de contraintes acyclique. Ce dernier CSP pouvant alors être résolu en temps polynomial par rapport à sa taille.

Cette méthode est généralement présentée ([DP89]) en utilisant l'approximation d'une triangulation optimale. Les étapes 1 et 2 sont réalisables en temps polynomial, plus précisément, en $O(n + m')$ où m' est le nombre d'arêtes du graphe après la triangulation ($m \leq m' < n^2$). Par ailleurs, étant données les cliques maximales, l'arbre associé à l'hypergraphe acyclique peut être calculé en temps linéaire. L'étape 3 est réalisable en $O(m.d^{w^+ + 1})$ où w^+ est la taille moins un de la plus grande clique produite ($w^+ + 1 \leq n$). La dernière étape possède la même complexité. La complexité en espace, qui est bornée par le coût de stockage des solutions de chaque sous-problème,

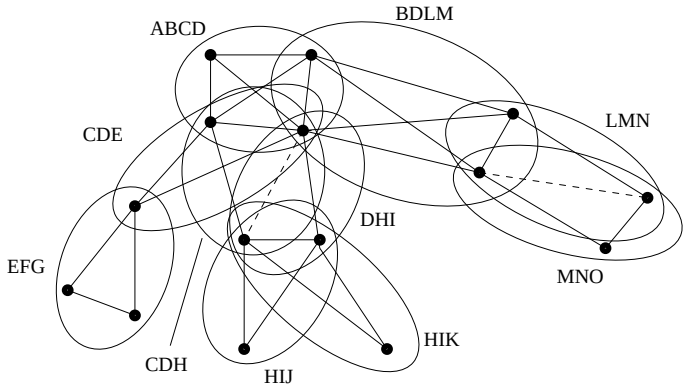

FIG. 1.7 : L'hypergraphe acyclique induit par les cliques maximales du graphe triangulé de la figure 1.6.

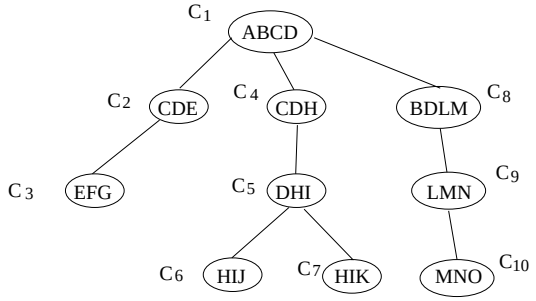

FIG. 1.8 : L'arbre associé à une décomposition arborescente du graphe de la figure 1.6.

peut être réduite à $O(n.s.d^s)$ où s est la taille maximum des séparateurs minimaux, c'est-à-dire, la taille de la plus grande intersection entre sous-problèmes ($s \leq w^+$). Finalement, notons que toute décomposition induit une valeur w^+, qui est telle que $w \leq w^+$ où w est la tree-width du graphe de contraintes initial.

Les figures 1.5 à 1.7 peuvent être considérées comme une illustration de cette méthode. Dans la figure 1.5, nous avons un graphe de contraintes. Lors de la première étape (figure 1.6), la triangulation a ajouté deux arêtes (les lignes pointillées). Un recouvrement de ce graphe par des cliques maximales définit un hypergraphe acyclique (figure 1.7). Chaque clique maximale définit alors un sous-problème.

En plus de TC, d'autres méthodes de décomposition ont été proposées, parmi lesquelles :

- la décomposition en composantes biconnexes [Fre85],
- la méthode Coupe-Cycle [Dec90],
- la hinge-décomposition [GJC94],
- la décomposition en hyperarbre [GLS99].

On pourra consulter [GLS00] pour obtenir une description détaillée ainsi qu'une comparaison théorique de ces différentes méthodes. Des méthodes hybridant plusieurs techniques de décomposition ont été également développées, comme, par exemple, la méthode Cyclic-Clustering [Jég90] qui effectue un compromis entre TC et la méthode Coupe-Cycle. Plus récemment, une méthode

exploitant successivement la décomposition en composantes biconnexes, la hinge-décomposition et la décomposition en hyperarbre a été proposée dans [GHW02].

Bien que théoriquement intéressantes, ces méthodes n'ont pas encore démontré tout leur intérêt pratique, même s'il est clair que pour certaines classes de CSPs, elles peuvent fournir une approche utile ([DF01]). Une raison du manque d'efficacité pratique des méthodes comme TC est vraisemblablement la lourdeur de l'approche, en particulier l'espace mémoire requis. Pour le cas où toutes les solutions sont recherchées, ces méthodes peuvent être utile. Par contre, pour le simple test de consistance ou pour la recherche d'une seule solution, il sera préférable d'employer des algorithmes énumératifs tels que FC [HE80] ou bien MAC [SF94].

1.2.4.2 Méthodes de décomposition basées sur la micro-structure

Nous commençons par rappeler la définition de la micro-structure d'une instance CSP. Cette définition requiert que le graphe de contraintes du problème $\mathcal{P}$ soit complet. Si ce n'est pas le cas, le graphe est complété en utilisant des relations universelles.

Définition 1.29 (micro-structure) *Soit une instance CSP $\mathcal{P} = (X, D, C, R)$.*
On appelle microstructure de $\mathcal{P}$ (notée $\mu(\mathcal{P})$) le graphe $(X_{\mu(\mathcal{P})}, E_{\mu(\mathcal{P})})$ avec :

- $X_{\mu(\mathcal{P})} = \{(x_i, v) \mid x_i \in X \text{ et } v \in d_i\}$
- $E_{\mu(\mathcal{P})} = \{((x_i, v), (x_j, w)) \mid \exists c = \{x_i, x_j\} \in C, (v, w) \in r_c\}$

Autrement dit, la micro-structure représente le graphe de compatibilité entre des paires variable-valeur.

Deux décompositions exploitant la micro-structure ont été proposées. La première ([Jég93]) recherche des cliques maximales dans la micro-structure. Ce calcul est réalisé en exploitant les notions de graphe triangulé et de triangulation (ces notions sont présentées dans le chapitre 2). La méthode construit ensuite des sous-problèmes à partir des cliques produites. Chaque sous-problème peut alors être résolu de façon indépendante en utilisant des algorithmes énumératifs classiques comme FC ou MAC. Le problème initial possède une solution si un des sous-problèmes est consistant. La seconde méthode ([CN98]) exploite en fait le graphe complémentaire de la micro-structure. Elle utilise des cliques de ce graphe complémentaire pour définir des sous-problèmes. Notons que ces cliques correspondent en fait à des nogoods.

Comparées aux méthodes de décomposition utilisant le graphe de contraintes, ces deux méthodes ne fournissent aucune borne de complexité. Elles permettent seulement de définir et de résoudre des sous-problèmes de taille réduite par rapport à la taille du problème initial.

1.3 Résolution parallèle

Dans les différentes communautés travaillant sur des problèmes à base de contraintes (CSP, SAT, problèmes d'optimisation, ...), un des principaux objectifs a toujours été de résoudre plus rapidement les problèmes. Aussi, c'est tout naturellement que le parallélisme a fait son apparition dans ces différents domaines. Diverses approches ont été développées parmi lesquelles :

- la parallélisation des algorithmes existants,
- le lancement de plusieurs solveurs sur un même problème (concurrence),
- la coopération entre différents solveurs.

Avant de décrire ces diverses approches, nous rappelons comment mesurer la performance d'un algorithme parallèle.

1.3.1 Performance d'un algorithme parallèle

D'un point de vue expérimental, on peut considérer deux mesures de la performance d'un algorithme parallèle. La première consiste à positionner l'algorithme parallèle par rapport aux algorithmes classiques de la littérature, alors que la seconde concerne la qualité de la parallélisation.

En fait, la première mesure est une simple comparaison des résultats de l'algorithme parallèle et de ceux obtenus par les principaux algorithmes de la littérature (qu'ils soient séquentiels ou parallèles). Elle permet d'évaluer l'intérêt pratique de l'utilisation de l'algorithme parallèle pour résoudre une tâche (ou un ensemble de tâches) donnée. Une telle étude comparative étant habituellement menée pour tout nouvel algorithme (parallèle ou non), nous ne nous étendrons pas sur ce sujet.

La parallélisation d'un algorithme séquentiel a pour but d'accélérer l'exécution de la tâche accomplie par cet algorithme. Aussi, quand on souhaite évaluer la qualité d'un algorithme parallèle, il semble naturel de le comparer à une version séquentielle de cet algorithme, ce qui conduit aux notions d'accélération et d'efficacité. Dans les définitions de ces deux notions, nous employons volontairement le terme plus général de "processus" en lieu et place du terme habituel de "processeur". Le but est tout simplement de pouvoir utiliser ces notions aussi bien dans un cadre multi-processeurs que mono-processeur. Cette liberté nous apparaît d'autant plus nécessaire qu'en pratique, nous ne pouvons pas toujours exécuter un seul processus par processeur, faute de disposer d'un nombre suffisant de processeurs.

Définition 1.30 (accélération) *Soit T_1 (respectivement T_p) le temps d'exécution d'un algorithme utilisant un processus (resp. p processus). On appelle* ***accélération*** *le rapport $\frac{T_1}{T_p}$. Une accélération est dite* ***linéaire*** *si elle est égale à p,* ***superlinéaire*** *si elle est supérieure à p,* ***sublinéaire*** *sinon. L'accélération est qualifiée d'****absolue*** *si T_1 est le temps du meilleur algorithme séquentiel, ou de* ***relative*** *si T_1 est le temps de l'algorithme parallèle utilisant un seul processus.*

L'accélération doit normalement correspondre à une valeur comprise entre 1 et p. Ce résultat suppose, entre autres, que T_1 est le temps de la meilleure version séquentielle de l'algorithme. Pour les problèmes qui nous intéressent, les différentes versions d'un algorithme se distinguent généralement par les heuristiques employées et par l'implémentation même de l'algorithme. Ainsi, pour un problème ou une classe de problèmes donné(e), il paraît difficile de déterminer quelle est la meilleure version. Aussi, il n'est pas rare d'observer, en pratique, des accélérations superlinéaires, quand on résout des problèmes comme le problème de satisfaction de contraintes. Pour la même raison, nous considérerons par la suite des accélérations relatives.

À partir de l'accélération, on définit l'efficacité :

Définition 1.31 (efficacité) *Soit T_1 (respectivement T_p) le temps d'exécution d'un algorithme utilisant un processus (resp. p processus). On appelle* ***efficacité*** *le rapport de l'accélération divisée par le nombre de processus, c'est-à-dire $\frac{T_1}{p.T_p}$.*

Si, bien sûr, l'idéal est d'obtenir une efficacité de 1, en pratique, une efficacité supérieure ou égal à 0,95 apparaît raisonnable. Au niveau expérimental, l'efficacité peut parfois devenir supérieure à 1, à cause de l'existence d'accélérations superlinéaires. Notons enfin que l'efficacité dépend du nombre de processus employés et qu'elle a généralement tendance à décroître avec l'augmentation du nombre de processus.

1.3.2 Parallélisation des algorithmes existants

Aux vues des limites rencontrées par les algorithmes classiques, une première possibilité pour accélérer les résolutions est offerte par la parallélisation des algorithmes existants. Cette pa-

rallélisation concerne aussi bien les algorithmes de résolution (énumératifs ou non) que les algorithmes de filtrages. Concernant ces derniers, de nombreux travaux ont été développés (parmi lesquels [Kas89, Kas90, KD94, CS92, SH87, SHZ+91, CBB91, CA95]). Ils concernent essentiellement l'arc-consistance ou la chemin-consistance. La mise en œuvre de ces algorithmes comme prétraitement apparaît intéressante, en particulier dans le cas de la chemin-consistance qui présente en pratique un coût proche de l'arc-consistance ([SHZ+91]). Par contre, leur utilisation en vue d'un maintien d'un niveau de consistance pendant la recherche semble moins prometteuse. En effet, si l'algorithme employé pour le parcours de l'arbre de recherche reste séquentiel, le gain en temps sera probablement limité. Dans le cas d'un parcours de l'arbre en parallèle, le nombre de processus (ou de processeurs) requis à chaque nœud pour établir la consistance, multiplié par le nombre de solveurs (car chaque solveur développe simultanément un nœud), constituera certainement un obstacle pour une telle approche. Notre sujet concernant l'étude des méthodes de résolution, nous n'en dirons pas plus sur ces travaux.

La parallélisation des algorithmes de résolution repose sur deux approches :

- parallélisation du parcours de l'arbre de recherche pour les algorithmes énumératifs,
- résolution en parallèle des différents sous-problèmes pour les algorithmes par décomposition.

1.3.2.1 Parcours de l'arbre de recherche en parallèle

Parcourir l'arbre de recherche en parallèle consiste à diviser l'arbre de recherche en plusieurs sous-arbres et à répartir ces sous-arbres entre différents solveurs. Si l'objectif est de déterminer l'existence d'une solution, la recherche se termine dès qu'un solveur a trouvé une solution (problèmes consistants) ou quand tous les solveurs ont achevé l'exploration de leurs sous-arbres (problèmes inconsistants). Au niveau du parallélisme, la principale difficulté est d'allouer à chaque solveur une quantité de travail suffisante pour éviter que des solveurs ne soient à court de travail et donc deviennent inactifs (phénomène de *famine*). En effet, l'inactivité d'une partie des solveurs va entraîner une chute de l'efficacité de la méthode. L'allocation des tâches peut être soit statique soit dynamique. Dans le premier cas, la répartition est effectuée avant de commencer la résolution. La question cruciale est de déterminer comment partager équitablement la charge de travail entre les solveurs afin d'éviter tout phénomène de famine. Avec une allocation dynamique, le travail est réparti pendant la résolution, avec des échanges de travail si un équilibrage des charges est nécessaire. Utiliser une allocation dynamique soulève plusieurs questions :

- quand équilibrer les charges ? doit-on attendre qu'un solveur n'ait plus de travail ou doit-on anticiper en équilibrant régulièrement la charge ?
- avec quel solveur échanger du travail ?
- quand peut-on diviser une tâche, comment la diviser et la transférer ?

Ces questions sont toutes aussi déterminantes les unes que les autres pour l'efficacité obtenue en pratique. À travers ces différentes questions, se pose également le problème de l'évaluation de la charge de travail. Ce problème apparaît aussi pour l'allocation statique. La nature même du problème à résoudre (que ce soit le problème CSP, SAT ou d'optimisation) rend impossible le calcul de la taille d'un sous-arbre de l'arbre de recherche avant son exploration. Aussi, généralement, la charge de travail est évaluée grâce à une fonction heuristique. À partir de cette fonction, on peut définir plusieurs politiques d'équilibrage. Les communications nécessaires à la mise en œuvre de l'équilibrage peuvent aussi pénaliser l'efficacité de la méthode si elles sont trop coûteuses ou trop nombreuses.

Au niveau du problème CSP, l'utilisation du parallélisme pour résoudre des instances CSPs a fait l'objet de peu de travaux (parmi lesquels [PNB96, Mér98]). Par contre, pour des problèmes voisins, le parallélisme a suscité plus d'intérêt. C'est le cas, par exemple, pour le problème SAT ([BS96, JLU01, SBK01]) ou pour le problème d'optimisation ([Rou92, Lau93, dBKT95]). Certains travaux (comme [RK87, KR87, RK93, San95]) considèrent plus généralement des recherches arborescentes

avec parcours en profondeur d'abord. Notre objectif n'étant pas d'employer ce type de méthodes, nous n'entrerons pas plus dans les détails. On pourra consulter un état de l'art plus développé dans [Mér98].

1.3.2.2 Parallélisme et décomposition

Pour la plupart des décompositions, la résolution du problème se déroule en plusieurs phases :

1. construction des sous-problèmes,
2. résolution de chaque sous-problème,
3. résolution du problème global à partir des résultats des résolutions des sous-problèmes.

En règle générale, les phases de construction des sous-problèmes et de résolution du problème global restent séquentielles (ce qui n'est pas dramatique car elles ont souvent une complexité en temps polynomiale). Par contre, dans la majorité des cas, les sous-problèmes peuvent être résolus en parallèle, car ils sont indépendants les uns des autres.

Dans [HKS00], une version parallèle de la décomposition par micro-structure ([Jég93]) est étudiée expérimentalement. Le parallélisme consiste uniquement en une résolution en parallèle des sous-problèmes. Les résultats expérimentaux sur des problèmes aléatoires montrent un gain très marqué pour les problèmes consistants (avec une accélération linéaire ou superlinéaire dans la quasi-totalité des cas). Par contre, pour les problèmes inconsistants, l'efficacité est nettement inférieure à 1. Un tel résultat pourrait s'expliquer par l'existence de sous-problèmes faciles à résoudre et d'autres plus difficiles. Dans un tel cas, un solveur ayant plusieurs sous-problèmes simples à résoudre terminerait avant un solveur qui posséderait au moins un sous-problème difficile, ce qui conduirait à une inactivité d'une partie des solveurs.

Notons enfin qu'un algorithme parallèle pour la résolution de CSPs acycliques est proposé dans [ZM93]. Cet algorithme peut s'avérer utile si on utilise une méthode de résolution par décomposition comme le Tree-Clustering.

1.3.3 Approches basées sur la concurrence

La concurrence consiste à lancer en parallèle plusieurs solveurs sur un même problème. Dans cette approche, chaque solveur travaille indépendamment des autres sur le problème dans son intégralité, à la différence de la parallélisation où les solveurs travaillent chacun sur une partie (a priori indépendante) du problème ou de l'espace de recherche. Bien entendu, les solveurs doivent être différents afin de ne pas effectuer plusieurs fois exactement la même recherche. Cette différence peut s'exprimer, par exemple, par l'utilisation d'heuristiques différentes pour ordonner les variables et/ou les valeurs. Elle peut être aussi obtenue en employant des solveurs conceptuellement différents (par exemple, en mêlant des méthodes complètes et incomplètes). En pratique, l'intérêt d'une telle approche repose essentiellement sur cette différence, avec l'espoir que l'un des solveurs soit mieux adapté au problème à résoudre et ainsi permette de trouver plus rapidement une solution. Du point de vue du parallélisme, un des avantages de ce type de méthodes (avec l'indépendance des recherches) est que la résolution s'arrête dès qu'un des solveurs a trouvé une solution (ou a prouvé qu'il n'en existe pas), ce qui permet d'éviter tout problème de famine. Par contre, un inconvénient majeur réside dans la redondance de l'espace de recherche visité par l'ensemble des solveurs. En effet, comme les recherches sont indépendantes, un solveur peut visiter un sous-arbre déjà exploré par un autre solveur. Il en découle généralement une dégradation des performances, en particulier quand les problèmes traités sont inconsistants.

L'approche concurrente a été expérimentée sur le problème SAT ([GBR96]) ou sur le problème de coloration de graphe ([HW94]). Dans les deux cas, les résultats présentés montrent une légère amélioration par rapport à un solveur unique. Cependant, les accélérations sont majoritairement

sublinéaires. Quelques accélérations linéaires ou superlinéaires sont tout de même observées. En guise de conclusion à leur travail ([HW94]), Hogg et Williams préconisent d'employer des méthodes basées sur la coopération plutôt que des méthodes simplement concurrentes.

1.3.4 Approches coopératives

Durant la résolution, les solveurs explicitent des informations contenues dans le problème, par exemple que telle ou telle affectation consistante ne peut pas conduire à une solution. Les informations découvertes par un solveur peuvent se révéler utiles pour un autre solveur. En échangeant des informations entre eux, les solveurs pourraient s'entraider et ainsi déterminer plus rapidement si le problème possède ou non une solution. Ce principe constitue la base de toute approche coopérative. Les informations échangées peuvent permettre de guider les solveurs vers une éventuelle solution, d'éviter certaines redondances dans les arbres de recherche explorés par les différents solveurs, ... L'approche coopérative peut être basée soit sur un parcours en parallèle de l'arbre de recherche ([PNB96, SBK01]), soit sur une approche concurrente ([CHH91, CHH92, HH93, HW93, MV96]). Dans les deux cas, l'efficacité d'une telle approche repose, en grande partie, sur les réponses aux questions suivantes :

- quelle est la nature de l'information à échanger ?
- comment réaliser l'échange proprement dit ?
- quand un solveur doit-il informer les autres ?
- quand un solveur peut-il recevoir des informations ?
- comment tirer parti des informations reçues ?

L'information échangée est généralement soit une affectation partielle consistante (i.e. le début d'une éventuelle solution [CHH91, CHH92, HH93, HW93]), soit une affectation partielle qui ne peut pas s'étendre en une solution (i.e. un nogood [PNB96, MV96, SBK01]). La réalisation de l'échange d'informations est une question essentiellement matérielle. L'échange est généralement réalisé soit par le biais de messages émis par les différents solveurs, soit via une mémoire partagée. Quant aux trois dernières questions, les réponses dépendent directement des algorithmes de résolution utilisés. Elles doivent également tenir compte du coût des communications, qui peut devenir, dans certains cas, prohibitif. En effet, plus le nombre d'informations échangées est grand, plus la coopération mais aussi le coût global des communications sont importants. Notons enfin que l'ajout de la coopération à un algorithme est susceptible de modifier ses propriétés comme par exemple sa complétude.

Dans le cas particulier des méthodes coopératives basées sur la concurrence, rendre efficace de telles méthodes requiert, en plus, de maximiser à la fois la diversité des solveurs (coté concurrence) et l'exploitation des informations échangées (coté coopération). La principale difficulté est que maximiser la diversité peut s'effectuer au détriment de l'utilisation des informations. En effet, si la diversité est trop importante, il se peut que les informations produites par certains solveurs ne soient pas exploitables par d'autres (qui exploreraient un sous-espace de recherche totalement différent). Un compromis apparaît donc nécessaire. D'une part, les solveurs doivent être suffisamment différents pour ne pas explorer exactement le même sous-espace de recherche. D'autre part, cette diversité doit être limitée pour que les informations produites soient exploitables par un maximum de solveurs.

Du point de vue expérimental, l'approche a été testée sur divers types de problèmes :

- des problèmes cryptarithmétiques ([CHH91, CHH92, HH93]),
- des problèmes de coloration de graphes ([HW93, HH93]),
- des problèmes de satisfaction de contraintes aléatoires ([MV96]),

- des benchmarks SAT ([SBK01]).

Les accélérations obtenues sont généralement satisfaisantes (hormis pour [MV96]). Elles sont même parfois superlinéaires. Dans la quasi-totalité des cas, l'approche coopérative apparaît meilleure que l'approche concurrente ou que le parcours en parallèle de l'arbre de recherche (suivant l'approche sur laquelle est basée la coopération).

Nous décrivons maintenant plus précisément le travail de Martinez et Verfaillie qui constitue la base du travail présenté dans le chapitre 4. Dans [MV96], Martinez et Verfaillie proposent une méthode coopérative basée sur la concurrence, dans laquelle la coopération repose sur un échange de nogoods. Tous les solveurs exécutent en concurrence l'algorithme Forward-Checking avec Nogood Recording dont seule l'heuristique employée pour ordonner les variables (ou les valeurs) varie d'un solveur à l'autre. La recherche s'arrête dès qu'un des solveurs a résolu le problème en découvrant soit une solution, soit l'inconsistance du problème. Chaque solveur produit des nogoods qu'il communique aux autres solveurs. Ainsi, les nogoods échangés peuvent permettre aux solveurs d'élaguer leur propre arbre de recherche. On peut alors s'attendre à obtenir plus rapidement une solution.

L'implémentation réalisée réunit tous les solveurs dans un seul processus, qui simule le parallélisme. Elle est donc fortement orientée vers un système monoprocesseur. L'échange de nogoods est effectué grâce à une structure de données commune à tous les solveurs. A priori, aucune exploitation particulière des nogoods échangés n'est effectuée, hormis celle accomplie par l'algorithme FC-NR, au niveau des tests de contraintes. Testée sur des problèmes aléatoires, la recherche coopérative apparaît comme meilleure que la recherche concurrente. Toutefois, le faible gain par rapport à un seul solveur laisse un doute sur l'efficacité d'une telle méthode, en particulier dans le cas d'une implémentation multiprocessus (i.e. une implémentation tournée vers un système mono ou multiprocesseurs).

1.4 Jeux de tests

Comparer théoriquement les algorithmes de résolution de CSPs est loin d'être une tâche facile. Les quelques travaux menés dans ce sens (dont [KvB97, CvB01]) nécessitent en général d'effectuer des hypothèses restrictives (comme d'utiliser un ordonnancement statique des variables ou un ordre particulier). Aussi, faute de pouvoir comparer théoriquement les algorithmes, les études comparatives sont principalement expérimentales. Elles ont alors recours à des jeux de tests. Ces jeux de tests sont divisisés en trois grandes catégories :

- les problèmes académiques,
- les problèmes aléatoires,
- les instances du monde réel.

Les problèmes académiques étaient souvent utilisés par le passé. Mais, actuellement, la tendance de la littérature serait plutôt à exploiter des instances aléatoires et/ou des instances du monde réel.

1.4.1 Problèmes académiques

Les problèmes académiques correspondent généralement à des jeux mathématiques ou logiques. De nombreux problèmes académiques peuvent être représentés comme un CSP. Nous présentons maintenant quelques-uns des plus usités.

1.4.1.1 Le problème des pigeons

Le problème des pigeons consiste à placer n pigeons dans $n-1$ pigeonniers tel qu'un pigeon niche dans au moins un pigeonnier et tel qu'un pigeonnier n'abrite pas plus d'un pigeon. Il peut être formalisé sous la forme d'un CSP utilisant n variables (qui correspondent aux pigeons) dont les domaines sont tous de taille $n-1$. Chaque valeur d'un domaine correspond alors à un pigeonnier. Le CSP possède une contrainte par paire de variables (le graphe de contraintes est donc complet). Chaque contrainte impose alors que les deux pigeons correspondants ne nichent pas dans le même pigeonnier. Il s'agit donc de contraintes de différence. Remarquons que la contrainte imposant qu'un pigeon niche dans au moins un pigeonnier est traduite simplement par le domaine, puisque le problème CSP a pour objectif d'associer exactement une valeur à chaque variable.

Ce problème est trivialement inconsistant. Cependant, l'emploi de la plupart des algorithmes énumératifs conduit tout de même à une résolution en temps exponentiel. Notons enfin que ce problème est aussi connu sous le nom de "tiroirs et chaussettes".

1.4.1.2 Le problème des n reines

Le problème des n reines a pour objectif de positionner n dames sur un échéquier $n \times n$ tel qu'aucune dame ne soit en prise (i.e. on ne doit pas placer deux dames sur la même ligne, la même colonne ou la même diagonale). Pour l'exprimer sous la forme d'un CSP, on emploie n variables qui possèdent toutes un domaine de taille n. La variable x_i est associée à la dame située sur la ligne i, ce qui garantit que chaque ligne ne contient qu'une et une seule dame. Chaque valeur d'un domaine correspond alors à une colonne. Une contrainte lie chaque paire de variables (le graphe de contraintes est donc complet). Chacune de ces contraintes impose que les deux dames correspondantes ne se trouvent pas dans la même colonne ou sur la même diagonale.

1.4.1.3 Le problème du zèbre

Le problème du zèbre a été proposé par le mathématicien Charles Dodgson (aussi connu sous le pseudonyme de Lewis Carroll). Il consiste à retrouver la nationalité (norvégienne, ukrainienne, anglaise, espagnole ou japonaise) de cinq personnes ainsi que la couleur de leur maison (jaune, bleue, rouge, blanche ou verte), leur boisson préférée (eau, thé, jus d'orange, café ou lait), leur animal de "compagnie" (zèbre, renard, cheval, chien ou escargot) et leur profession (diplomate, médecin, acrobate, sculpteur ou violoniste). En particulier, on souhaite déterminer qui sont le propriétaire du zèbre et le buveur d'eau. En fait, en pratique, on recherche la maison dont le propriétaire possède un zèbre et celle dont le propriétaire boit de l'eau. Les maisons sont numérotées de 1 à 5 (1 correspondant à la maison la plus à gauche, 5 à celle la plus à droite). Pour résoudre le problème, on dispose d'un ensemble de faits :

(1) L'Anglais habite dans la maison rouge.

(2) Le chien appartient à l'Espagnol.

(3) On boit du café dans la maison verte.

(4) L'Ukrainien boit du thé.

(5) La maison verte est située à droite de la blanche.

(6) Le sculpteur élève des escargots.

(7) Le diplomate habite dans la maison jaune.

(8) On boit du lait dans la maison du milieu.

(9) Le Norvégien habite la première maison à gauche.

(10) Le médecin habite la maison voisine de celle où demeure le propriétaire du renard.

(11) La maison du diplomate est à côté de celle où il y a un cheval.

(12) Le violoniste boit du jus d'orange.

(13) Le Japonais est acrobate.

(14) Le Norvégien demeure à côté de la maison bleue.

Notons qu'il existe plusieurs variantes de l'énoncé de ce problème.

Le problème se formalise par un CSP qui possède 25 variables, 5 pour chacune des catégories (maison, boisson, animal, profession et nationalité). À chaque variable est associé un domaine de taille 5. Chaque valeur correspond ainsi au numéro d'une maison. Toute paire de variables appartenant à la même catégorie est liée par une contrainte de différence. À ces contraintes s'ajoutent les contraintes qui traduisent les faits. Les contraintes traduisant les faits (8) et (9) sont unaires. Les autres sont des contraintes binaires de différence ou d'égalité qui portent sur des variables issues de différentes catégories (la contrainte correspondant au fait (5) fait exception). Ce problème ne possède qu'une seule solution.

1.4.1.4 Les problèmes cryptarithmétiques

Ces problèmes consistent à associer des chiffres à des lettres afin de réaliser des additions ou des multiplications. Un exemple classique de problèmes cryptarithmétiques est l'addition $SEND + MORE = MONEY$. L'addition ou multiplication obtenue en remplaçant les lettres par leur valeur doit bien sûr être valide. Évidemment, les lettres apparaissant plusieurs fois doivent avoir toujours la même valeur et les lettres les plus à gauche sont non nulles. De plus, deux lettres distinctes possèdent des valeurs distinctes.

Ces problèmes se formalisent en associant une variable à chaque lettre plus une variable par retenue éventuelle. Les domaines sont constitués des chiffres de 0 à 9 pour les variables associées à une lettre, des chiffres 0 et 1 pour les autres. Ces problèmes emploient :

- des contraintes unaires pour interdire la valeur 0 pour les lettres les plus à gauche,
- des contraintes binaires liant toutes les paires de variables afin d'assurer que deux lettres distinctes possèdent des valeurs distinctes,
- des contraintes n-aires pour garantir la validité de l'addition (ou de la multiplication).

Ces problèmes ne possèdent généralement qu'une seule solution.

1.4.2 CSPs aléatoires

Afin d'évaluer expérimentalement les algorithmes, il est d'usage courant d'utiliser des instances générées aléatoirement en complément des problèmes du monde réel et des problèmes académiques. L'intérêt d'employer de telles instances réside principalement dans la possibilité de produire de nombreuses instances.

Les instances aléatoires sont généralement générées suivant le modèle proposé par Prosser [Pro94]. Un CSP produit selon ce modèle se caractérise alors par un quadruplet (n, d, p_1, p_2) avec :

- n le nombre de variables,
- d la taille des domaines (la même pour tous les domaines),
- p_1 la probabilité qu'il existe une contrainte entre deux variables,
- p_2 la probabilité qu'un couple de valeurs soit interdit par une relation (la probabilité est la même pour toutes les relations).

Le principal inconvénient de ce modèle réside dans l'uniformité des instances générées.

Pour les expérimentations présentées dans cette thèse, la production des problèmes aléatoires classiques est réalisée grâce au générateur aléatoire écrit par D. Frost, C. Bessière, R. Dechter

et J.-C. Régin. Ce générateur[1] possède l'avantage d'être indépendant du système utilisé pour les expérimentations. Il permet donc de reproduire facilement les expérimentations. Il prend en entrée 4 paramètres n, d, m et t. Il construit un CSP de la classe (n, d, m, t) avec n variables, chacune ayant un domaine de taille d. Chaque instance possède m contraintes binaires ($0 \leq m \leq \frac{n(n-1)}{2}$). Pour chaque relation, t tuples sont interdits($0 \leq t \leq d^2$). Ce générateur respecte donc le modèle proposé dans [Pro94]. Parmi les instances produites par le générateur, nous ne conservons, pour nos expérimentations, que les problèmes dont le graphe de contraintes est connexe.

1.4.3 Problèmes issus du monde réel

La comparaison expérimentale d'algorithmes de résolution emploie souvent des problèmes académiques ou des instances aléatoires. L'utilisation de ces problèmes s'explique en partie par leur nombre important et leur diversité. En particulier, pour les instances aléatoires, la possibilité de produire un grand nombre d'instances différentes pour des jeux de paramètres variés s'avère très intéressante. De plus, les problèmes académiques ou aléatoires sont facilement accessibles, ce qui se révèle important lorsqu'on souhaite que les expériences soient reproductibles. À l'opposé, si de nombreux problèmes issus du monde réel peuvent être décrits dans le formalisme CSP, ils ne sont que trop rarement accessibles à tous. Cependant, le faible nombre de problèmes disponibles ne diminue pas pour autant leur intérêt pour l'évaluation pratique des algorithmes. En effet, les problèmes académiques et les instances aléatoires sont généralement trop uniformes pour pouvoir refléter la réalité.

La plupart des problèmes réels correspondent à des CSPs n-aires. C'est par exemple le cas pour le problème SPOT-5. Ce problème consiste à planifier des prises de vue du satellite SPOT-5. Initialement, il s'agit d'un problème d'optimisation. Toutefois, on peut se contenter de rechercher une solution.

Pour notre part, durant nos expérimentations, nous utilisons comme instances du monde réel des problèmes issus de l'archive FullRLFAP[2]. Ces instances sont de plus en plus utilisées, probablement car elles présentent l'avantage d'employer des contraintes unaires ou binaires. Elles correspondent à des problèmes d'allocation de fréquence. Cette allocation de fréquence est soumise à des contraintes géographiques ou physiques. Comme pour le problème SPOT-5, il s'agit à la base d'un problème d'optimisation. Cependant, la recherche d'une solution se révèle être aussi un problème intéressant puisque ces instances sont de taille importante (de 200 à 916 variables) avec des domaines de taille différente. Les onze instances SCEN-xx correspondent à une simplification d'un problème réel. Par contre, les quatorze instances GRAPH-xx sont produites par un générateur. Par construction, elles sont voulues aussi proches que possibles des instances SCEN-xx. Pour plus de détails sur ces problèmes, on pourra consulter [CdGL+99].

1.5 Conclusion

Dans ce premier chapitre, nous avons d'abord rappelé le formalisme CSP avant de présenter quelques unes des méthodes de résolution de CSPs. Dans le cadre séquentiel, nous nous sommes focalisés sur les méthodes énumératives et structurelles. D'un point de vue pratique, les méthodes énumératives semblent avoir atteint certaines limites, aucune avancée significative n'ayant été constatée ces dernières années. D'un autre coté, les méthodes par décomposition basées sur la structure du problème garantissent des bornes de complexité en temps meilleures que celles des méthodes énumératives. Par contre, elles n'ont toujours pas prouvé leur intérêt pratique. Dans le chapitre 2, nous montrons comment la référence à la décomposition structurelle permet de proposer

[1] téléchargeable à l'adresse http://www.lirmm.fr/∼bessiere/generator.html

[2] nous remercions le Centre d'Électronique de l'Armement (CELAR).

une procédure de recherche basée sur l'énumération tout en garantissant des bornes de complexité similaires à celles offertes par le Tree-Clustering.

Dans le cadre parallèle, nous nous sommes intéressés principalement aux méthodes concurrentes avec coopération. Bien que l'approche semble prometteuse, peu de travaux l'ont étudiée. Parmi ces travaux, ceux de Martinez et Verfaillie [MV96] proposent d'échanger des nogoods. Si l'approche semble intéressante, les travaux de Martinez et Verfaillie ne permettent pas de déterminer si l'échange de nogoods est une forme efficace de coopération. Dans le chapitre 4, nous poursuivons ces travaux avec pour but de répondre à cette question.

Enfin, nous avons présenté quelques jeux de tests couramment employés pour évaluer et comparer expérimentalement les algorithmes de résolution du problème CSP.

Chapitre 2

Décomposition arborescente et résolution énumérative

Dans ce chapitre, nous proposons une méthode pour la résolution de CSPs basée à la fois sur les techniques de recherche arborescente et sur la notion de décomposition arborescente du réseau de contraintes. Cette approche mixte nous permet de définir un cadre pour l'énumération dont nous espérons qu'il bénéficiera d'une part des avantages des techniques d'énumération pour leur efficacité pratique, et d'autre part, des garanties en terme de complexité théorique qu'offrent les méthodes de décomposition de CSPs. Les résultats expérimentaux que nous avons obtenus nous ont permis de nous assurer de l'intérêt pratique de cette approche.
Une partie du travail présenté dans ce chapitre a fait l'objet d'une publication dans [JT02]. Une version complète [JT03] est à paraître.

2.1 L'algorithme BTD

2.1.1 Présentation

La méthode BTD (pour Backtracking sur Tree-Decomposition) procède par une recherche énumérative qui est guidée par un ordre partiel statique préétabli à partir d'une décomposition arborescente du réseau de contraintes. Aussi, la première étape de BTD consiste à calculer une décomposition arborescente ou une approximation de décomposition arborescente. L'ordre partiel considéré permet d'exploiter quelques propriétés structurelles du graphe pendant la recherche afin d'élaguer certaines branches de l'arbre de recherche. En fait, ce qui distingue BTD des autres techniques de backtracking concerne les points suivants :

- l'ordre d'instanciation des variables est induit par une décomposition arborescente du graphe de contraintes,
- certaines parties de l'espace de recherche ne seront plus visitées dès que leur consistance sera connue (notion de *good structurel*),
- certaines parties de l'espace de recherche ne seront plus visitées si on sait que l'instanciation courante, bien que consistante, conduit à un échec (notion de *nogood structurel*).

Notons dès à présent que cette méthode peut être implémentée avec des variantes plus sophistiquées que le backtracking de base, comme, par exemple, FC ou MAC (ou d'autres algorithmes encore).

2.1.2 Justifications formelles

Soit $\mathcal{P} = (X, D, C, R)$ une instance telle que (X, C) est un graphe, avec $(\mathcal{C}, \mathcal{T})$ une décomposition arborescente (ou une approximation) où $\mathcal{T} = (I, F)$ est un arbre. Nous supposerons que les éléments de $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ sont indicés à partir de la notion de *numérotation compatible* :

Définition 2.1 *Une numérotation sur $\mathcal{C}$ compatible avec une numérotation préfixée de $\mathcal{T} = (I, F)$ dont $\mathcal{C}_1$ est la racine est appelée numérotation compatible $N_{\mathcal{C}}$.*

L'exemple de décomposition arborescente donné dans la figure 1.8 (page 27) présente une numérotation compatible sur $\mathcal{C}$. Nous notons $Desc(\mathcal{C}_j)$ l'ensemble de variables appartenant à l'union des descendants $\mathcal{C}_k$ de $\mathcal{C}_j$ dans l'arbre enraciné dans $\mathcal{C}_j$, $\mathcal{C}_j$ inclus. Par exemple, $Desc(\mathcal{C}_4) = \mathcal{C}_4 \cup \mathcal{C}_5 \cup \mathcal{C}_6 \cup \mathcal{C}_7 = \{C, D, H, I, J, K\}$. La numérotation $N_{\mathcal{C}}$ définit un ordre partiel sur les variables qui permet d'établir un ordre d'énumération sur les variables de $\mathcal{P}$:

Définition 2.2 *Un ordre $\preceq_X$ sur les variables de X tel que $\forall x \in \mathcal{C}_i, \forall y \in \mathcal{C}_j$, avec $i < j$, $x \preceq_X y$ est un ordre d'énumération compatible.*

Dans l'exemple, l'ordre alphabétique $A, B, \ldots, N, O$ est un ordre d'énumération compatible. La décomposition arborescente avec la numérotation N_C permet de clarifier quelques relations dans le graphe de contraintes.

Théorème 2.3 *Soit $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ (avec $i < j$). Il n'existe pas d'arête $\{x, y\}$ dans le graphe (X, C) où $x \in (\cup_{k=1}^{j-1} \mathcal{C}_k) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ et $y \in Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$.*

Preuve :
Par définition, $\mathcal{C}_i \cap \mathcal{C}_j$ est clairement un séparateur du graphe qui déconnecte $(\cup_{k=1}^{j-1} \mathcal{C}_k) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ et $Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$. □

Par exemple, soit $i = 1$, $j = 4$, et $\mathcal{C}_4$ un fils de $\mathcal{C}_1$. Il n'y a pas d'arête dans (X, C) entre $(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{A, B, C, D, E, F, G\} \backslash \{C, D\} = \{A, B, E, F, G\}$ et $Desc(\mathcal{C}_4) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{C, D, H, I, J, K\} \backslash \{C, D\} = \{H, I, J, K\}$.

En terme de CSP, cela signifie qu'il n'existe pas de contrainte joignant ces deux sous-ensembles de variables et par conséquent ces deux sous-problèmes. De fait, les relations de compatibilité entre les instanciations passent uniquement par le séparateur $\mathcal{C}_i \cap \mathcal{C}_j$.

La méthode BTD est basée sur un ordre d'énumération compatible et sur ce premier théorème. Considérons une instanciation consistante $\mathcal{A}$ des variables de $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_i \cup \mathcal{C}_{i+1} \cup \ldots \cup \mathcal{C}_{j-1}$, avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$. Du fait de la définition des ordres compatibles, l'énumération se poursuit sur les variables de la descendance $Desc(\mathcal{C}_j)$ à l'exception de celles qui appartiennent à $\mathcal{C}_i \cap \mathcal{C}_j$. Deux cas se présentent alors, selon qu'il existe ou non une extension consistante de l'instanciation sur $Desc(\mathcal{C}_j)$:

- **Il n'existe pas d'extension consistante**. Dans ce cas, la raison de l'inconsistance est essentiellement liée à l'insatisfaction de contraintes reliant deux variables de $Desc(\mathcal{C}_j)$, ou (ou non exclusif) une variable de cet ensemble et une variable qui la précède dans cet ordre, c'est-à-dire qui appartient à $\mathcal{C}_i \cap \mathcal{C}_j$ (cf. théorème 2.3). Dans ce cas, si on essaie une nouvelle instanciation consistante $\mathcal{A}'$ telle que $\mathcal{A}'$ et $\mathcal{A}$ soient identiques sur $\mathcal{C}_i \cap \mathcal{C}_j$, son extension sur $Desc(\mathcal{C}_j)$ conduira au même échec, indépendamment de ce qui précède. En fait, l'instanciation restreinte à $\mathcal{C}_i \cap \mathcal{C}_j$ peut être considérée comme un *nogood* au sens usuel du terme, bien qu'ici, il ait été trouvé sur la base de critères structurels. Ce nogood peut alors être exploité lors des développements ultérieurs de l'arbre de recherche.

– **Il existe une extension consistante**. Par un raisonnement similaire au précédent, il est possible de montrer que toute instanciation identique sur $\mathcal{C}_i \cap \mathcal{C}_j$ conduira à un succès sur $Desc(\mathcal{C}_j)$, parce qu'elle est indépendante de ce qui précède. Cette affectation peut être considérée comme un *good* au sens où, sur une partie du problème, $Desc(\mathcal{C}_j)$, cette instanciation possède une extension consistante. Comme les nogoods, les goods peuvent être enregistrés et utilisés lors des recherches ultérieures, autorisant des sauts dans l'arbre de recherche (*forward-jumping*), qui permettent de poursuivre l'énumération sur les variables situées après celles de $Desc(\mathcal{C}_j)$ dans l'ordre d'énumération compatible.

Les travaux les plus proches de notre approche sont ceux de Bayardo et Miranker dans [BM94] dont l'étude est limitée à la résolution de CSPs binaires arborescents. Toutefois, BTD peut être considérée comme une généralisation de leur travail puisque leurs goods et leurs nogoods sont des instanciations de variables tandis que les nôtres correspondent à des affectations d'ensembles de variables (les séparateurs). Dans [BM96], Bayardo et Miranker proposent une autre généralisation des goods et des nogoods qui n'est pas basée sur les séparateurs, mais sur des ensembles d'ancêtres sur la base d'un graphe de contraintes ordonné. Formellement, ce travail est différent du nôtre, bien que l'exploitation des goods et des nogoods pendant la recherche soit similaire à la nôtre (nous y revenons dans la partie 2.4).

Nous définissons maintenant formellement notre notion de goods et de nogoods fondée sur les séparateurs.

Définition 2.4 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, un **good** (resp. **nogood**) de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ est une affectation consistante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$ telle qu'il existe (resp. il n'existe pas) d'extension consistante de $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$.*

Le lemme suivant et son corollaire montrent que les interactions entre un sous-problème enraciné en $\mathcal{C}_j$ et le reste du CSP transitent via l'intersection entre $\mathcal{C}_j$ et son père $\mathcal{C}_i$. Ces propriétés sont à l'origine des coupes (pour les nogoods) et des sauts (pour les goods) qui seront réalisés dans l'arbre de recherche.

Lemme 2.5 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, toute instanciation consistante $\mathcal{B}$ de $Desc(\mathcal{C}_j)$ est compatible avec toute instanciation $\mathcal{A}$ de Y si et seulement si $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

Preuve :
D'après le théorème 2.3 et par construction, les seules contraintes joignant les variables de Y à celles de $Desc(\mathcal{C}_j)$ sont les contraintes qui impliquent les variables communes à $Desc(\mathcal{C}_j)$ et à Y, i.e. $\mathcal{C}_i \cap \mathcal{C}_j$. Il en résulte que $\mathcal{A}$ et $\mathcal{B}$ sont compatibles si et seulement si chaque variable commune a la même valeur dans $\mathcal{A}$ et dans $\mathcal{B}$ (i.e. $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$). □

Ce lemme conduit directement au corollaire suivant :

Corollaire 2.6 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, toute instanciation consistante $\mathcal{B}$ de $Desc(\mathcal{C}_j)$ est compatible avec toute instanciation $\mathcal{A}$ de $(X \backslash Desc(\mathcal{C}_j)) \cup \mathcal{C}_i$ si et seulement si $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

L'exploitation des goods peut alors être formalisée comme suit :

Lemme 2.7 (saut par les goods (forward-jumping)) *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, alors, pour tout good g de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, toute instanciation consistante $\mathcal{A}$ de Y telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g$ possède une extension consistante sur $Desc(\mathcal{C}_j)$.*

Preuve : Soit $\mathcal{A}$ une instanciation consistante telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g$. Par définition des goods, il existe une instanciation $\mathcal{B}$ sur $Desc(\mathcal{C}_j)$ telle que $\mathcal{B}$ soit consistante et $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j] = g$. Comme $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$, $\mathcal{A}$ et $\mathcal{B}$ sont compatibles (d'après le lemme 2.5). Donc, $\mathcal{B}$ est une extension consistante de $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$. □

Ainsi, si une instanciation partielle $\mathcal{A}$ est telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est un good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, alors il n'est pas nécessaire d'étendre $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$. Aussi, l'énumération se poursuit alors sur les variables du premier $\mathcal{C}_k$ localisé à l'extérieur de $Desc(\mathcal{C}_j)$, par exemple le premier frère de $\mathcal{C}_j$, s'il en existe un.

Lemme 2.8 (coupe par les nogoods) *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, alors pour tout nogood ng de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, il n'existe pas d'affectation $\mathcal{A}$ de Y telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = ng$ et telle que $\mathcal{A}$ possède une extension consistante sur $Desc(\mathcal{C}_j)$.*

Preuve : D'après la définition des nogoods, il n'existe pas d'extension consistante de ng sur $Desc(\mathcal{C}_j)$. Comme $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = ng$, $\mathcal{A}$ ne peut être étendue de façon consistante sur $Desc(\mathcal{C}_j)$. □

2.1.3 L'algorithme de base

La méthode que nous proposons sur la base des notions précédentes peut être implémentée de plusieurs façons, selon le filtrage associé (ou non) à l'énumération. Cependant, les mécanismes seront similaires. La méthode BTD explore l'espace de recherche en utilisant un ordre compatible $\preceq_X$, qui débute sur les variables de $\mathcal{C}_1$. Dans un $\mathcal{C}_i$, l'énumération procède classiquement. Par ailleurs, quand toutes les variables sont affectées en satisfaisant toutes les contraintes concernées, nous obtenons une instanciation consistante $\mathcal{A}$ des variables de $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_i$. La recherche se poursuit alors sur les variables du premier fils $\mathcal{C}_{i+1}$ de $\mathcal{C}_i$, s'il en existe un. Plus généralement, nous considérons le cas d'un fils $\mathcal{C}_j$ de $\mathcal{C}_i$. Nous testons alors si $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est un good ou un nogood et l'action appropriée est alors exécutée :

- Dans le cas d'un nogood, nous modifions l'instanciation courante de $\mathcal{C}_i$.
- Dans le cas d'un good, un "forward-jump" est réalisé, afin de poursuivre l'énumération sur la première variable localisée après celles de $Desc(\mathcal{C}_j)$. La figure 2.1 illustre le cas d'un forward-jump, en supposant que $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5] = \mathcal{A}[\{D, H\}]$ est un good. Nous montrons dans la partie (a) le saut réalisé dans l'ordre d'énumération compatible, et dans la partie (b), la poursuite de la recherche par rapport à la structure de l'instance.
- Dans les autres cas, i.e. $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ n'est ni un good ni un nogood, $\mathcal{A}$ doit être étendue de manière consistante sur les variables de $Desc(\mathcal{C}_j)$. Si tel est le cas, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est enregistré en tant que good ; dans le cas contraire, si $\mathcal{A}$ ne peut être étendue de manière consistante, le nogood $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est mémorisé.

La figure 2.2 décrit l'algorithme BTD restreint au test de consistance de CSP : il retourne **True** si l'instanciation consistante $\mathcal{A}$ peut être étendue en une instanciation consistante sur $V_{\mathcal{C}_i}$ et sur la descendance de $\mathcal{C}_i$; **False** dans les autres cas. $V_{\mathcal{C}_i}$ représente des variables non affectées de $\mathcal{C}_i$ et G et N respectivement les ensembles de goods et de nogoods enregistrés. Cet algorithme est bien sûr exécuté après le calcul d'une décomposition arborescente (ou d'une approximation) du graphe de contraintes.

Théorème 2.9 *L'algorithme BTD est correct, complet et termine.*

Preuve : La preuve s'effectue par induction, en exploitant les propriétés des goods et nogoods structurels. L'induction est réalisée sur le nombre de variables apparaissant dans la descendance de $\mathcal{C}_i$ excepté les variables déjà affectées de $\mathcal{C}_i$. Cet ensemble de variables est noté $VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) =$

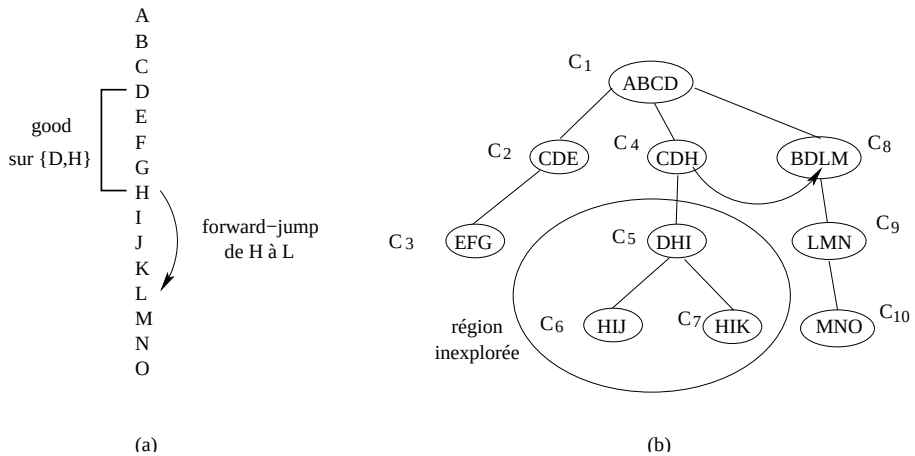

FIG. 2.1 : Exemple de forward-jump avec un good $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5]$ sur $\{D, H\}$. Dans (a), nous montrons le saut dans l'ordre d'énumération, tandis qu'en (b) nous voyons le saut réalisé dans la structure du problème.

$$V_{\mathcal{C}_i} \cup (\bigcup_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} (Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)))$$

$VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$ correspond, en fait, à l'ensemble des variables à instancier pour déterminer si $\mathcal{A}$ peut être étendue en une affectation consistante sur $V_{\mathcal{C}_i}$ et sa descendance.

Pour démontrer la correction de l'algorithme BTD, nous devons prouver la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ définie ainsi :
"BTD$(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i})$ retourne True si l'affectation consistante $\mathcal{A}$ peut être étendue en une affectation consistante sur $V_{\mathcal{C}_i}$ et sur la descendance de $\mathcal{C}_i$; sinon, BTD retourne False".

Considérons $P(\mathcal{A}, \emptyset)$:
Si $VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) = \emptyset$, alors $V_{\mathcal{C}_i} = \emptyset$ et $Fils(\mathcal{C}_i) = \emptyset$. Puisque $\mathcal{A}$ est une affectation consistante, $\mathcal{A}$ peut s'étendre en une affectation consistante sur $V_{\mathcal{C}_i}$ et la descendance de $\mathcal{C}_i$. Donc $P(\mathcal{C}_i, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ est vraie.

Pas d'induction : $P(\mathcal{A}, S)$ avec $S \neq \emptyset$. Supposons que $\forall S' \subset S, P(\mathcal{A}, S')$ soit vérifiée.

- Si $V_{\mathcal{C}_i} \neq \emptyset$:
  Durant la boucle **While** (lignes 27-33) l'assertion "il n'existe pas de valeur v de x déjà examinée telle que $\mathcal{A}$ étendue par cette valeur conduise à une affectation consistante sur $V_{\mathcal{C}_i}$ et sur la descendance de $\mathcal{C}_i$" est vérifiée.
  Si BTD est appelé (ligne 31), $\mathcal{A} \cup \{x \leftarrow v\}$ est donc consistante (puisque aucune contrainte n'est violée) et $VAR(\mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}) \subset VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$. D'après l'hypothèse d'induction, l'affectation $\mathcal{A}$ a été étendue si BTD$(\mathcal{A} \cup \{x \leftarrow v\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\})$ renvoie True. Dans ce cas, BTD$(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i})$ retourne True et la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ est satisfaite.
  Après la boucle (ligne 34), toutes les valeurs possibles ont été essayées sans succès (i.e. aucune extension consistante de $\mathcal{A}$ n'a été trouvée). Donc, BTD$(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i})$ retourne False et $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ est vérifiée.
- Si $V_{\mathcal{C}_i} = \emptyset$:
  Durant la boucle **While** (lignes 7-20) l'assertion "pour chaque fils $\mathcal{C}_f$ déjà examiné, $\mathcal{A}$ peut être étendue en une affectation consistante sur $Desc(\mathcal{C}_f)$" est vérifiée.
  Nous allons montrer que cette assertion est encore vraie à l'issue de la boucle.

```
   BTD(A, C_i, V_{C_i})
1. If V_{C_i} = ∅
2. Then
3.     If Fils(C_i) = ∅ Then Retourner True
4.     Else
5.       Consistance ← True
6.       F ← Fils(C_i)
7.       While F ≠ ∅ and Consistance Do
8.           Choisir C_j dans F
9.           F ← F\{C_j}
10.          If A[C_j ∩ C_i] est un good de C_i/C_j dans G Then Consistance ← True
11.          Else
12.            If A[C_j ∩ C_i] est un nogood de C_i/C_j dans N Then Consistance ← False
13.            Else
14.              Consistance ← BTD(A, C_j, C_j\(C_j ∩ C_i))
15.              If Consistance
16.              Then Enregistrer le good A[C_j ∩ C_i] de C_i/C_j dans G
17.              Else Enregistrer le nogood A[C_j ∩ C_i] de C_i/C_j dans N
18.            EndIf
19.          EndIf
20.      EndWhile
21.      Retourner Consistance
22.   EndIf
23. Else
24.   Choisir x ∈ V_{C_i}
25.   d ← d_x
26.   Consistance ← False
27.   While d ≠ ∅ and ¬Consistance Do
28.       Choisir v dans d
29.       d ← d\{v}
30.       If ∄c ∈ C telle que c viole A ∪ {x ← v}
31.       Then Consistance ← BTD(A ∪ {x ← v}, C_i, V_{C_i}\{x})
32.       EndIf
33.   EndWhile
34.   Retourner Consistance
35. EndIf
```

FIG. 2.2 : L'algorithme BTD.

Soit $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ à examiner.

+ Si $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ est un good de $\mathcal{C}_i/\mathcal{C}_j$, d'après le lemme 2.7, nous savons que $\mathcal{A}$ peut être étendue sur $Desc(\mathcal{C}_j)$. Donc, l'assertion est vérifiée à l'issue de la boucle.
+ Si $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ est un nogood de $\mathcal{C}_i/\mathcal{C}_j$, d'après le lemme 2.8, $\mathcal{A}$ ne peut être étendue sur $Desc(\mathcal{C}_j)$. La boucle est alors terminée.
+ Si $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ n'est ni un good, ni un nogood, alors, BTD est appelé avec $\mathcal{A}$ qui est une affectation consistante et $VAR(\mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i)) \subset VAR(\mathcal{C}_i, \emptyset)$. Par conséquent, d'après l'hypothèse d'induction, BTD$(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i))$ retourne True si $\mathcal{A}$ possède une affectation consistante sur $Desc(\mathcal{C}_j)$, et alors l'assertion est vérifiée. Sinon, la boucle est terminée.

Après la boucle (ligne 21), BTD$(\mathcal{A}, \mathcal{C}_i, \emptyset)$ retourne True si $\mathcal{A}$ a été étendue de façon consistante sur chaque fils ; retourne False sinon.

Donc, $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ est satisfaite. Notons que la mémorisation des goods et des nogoods se justifie par leur définition.

Pour résumer, comme BTD satisfait la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$, et en particulier la propriété $P(\emptyset, VAR(\mathcal{C}_1, \mathcal{C}_1))$ pour le premier appel, BTD est correct, complet et termine. □

2.1.4 Extensions de BTD

Nous discutons, à présent, des extensions de l'algorithme BTD. La première version de BTD présentée est basée sur le Backtracking Chronologique dont on connaît la relative inefficacité. Aussi, afin de rendre notre approche opérationnelle en pratique, est-il nécessaire de doter BTD de techniques de filtrage comme la consistance d'arc ou le forward-checking. Comme extension naturelle, nous proposons donc **FC-BTD** (respectivement **MAC-BTD**) qui consiste en un BTD couplé avec un filtrage de type Forward-Checking (resp. de type consistance d'arc). Plus généralement, les propriétés de l'approche, sous réserve de se limiter à des filtrages n'altérant pas la structure du réseau de contraintes, nous garantissent la validité de telles extensions. L'application durant la recherche d'un filtrage plus puissant, comme la consistance de chemin ([Mon74, Mac77]), n'est pas possible. En effet, un tel filtrage est susceptible d'ajouter de nouvelles arêtes, modifiant ainsi les propriétés structurelles exploitées par BTD.

Intuitivement, il faut considérer que les différents filtrages transiteront nécessairement par les séparateurs du graphe, et qu'en conséquence, l'exploitation des goods et des nogoods demeurera correcte. Pour FC-BTD, la correction de l'extension est triviale. Pour MAC-BTD, la correction est également évidente, mais, nous considérons qu'il est préférable de l'établir formellement par le théorème suivant :

Théorème 2.10 *Soient $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ et $\mathcal{A}$ une affectation consistante sur $\cup_{k=1}^{j-1}\mathcal{C}_k$. Supposons que la fermeture par arc-consistance du CSP $\mathcal{P}$ après l'affectation $\mathcal{A}$ (notée $AC(\mathcal{P}, \mathcal{A})$) n'ait pas de domaine vide. Si g est un good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ dans $\mathcal{P}$ tel que $g = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$, alors g est un good dans $AC(\mathcal{P}, \mathcal{A})$.*

Preuve :
Soit $\mathcal{B}$ une affectation consistante sur $Desc(\mathcal{C}_i)$ associée au good g. Autrement dit, $\mathcal{B}$ est une solution du sous-problème de $\mathcal{P}$ induit par les variables de $Desc(\mathcal{C}_i)$. Donc, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$. Par définition, $\mathcal{B}$ satisfait toutes les contraintes appartenant à $Desc(\mathcal{C}_j)$. De plus, toutes les valeurs dans $\mathcal{A}$ sont compatibles avec toutes les valeurs dans $\mathcal{B}$. En effet, les contraintes entre $\mathcal{A}$ et $\mathcal{B}$ associent des paires de variables $\{x_i, x_j\}$ telles que $x_i \in \mathcal{C}_i$ et $x_j \in \mathcal{C}_j$. Alors, il existe trois cas :

1. $x_j \in \mathcal{C}_i$. Comme $\mathcal{A}$ est une affectation consistante, $\mathcal{A}$ satisfait les contraintes apparaissant dans $\mathcal{C}_i$, en particulier $\{x_i, x_j\}$.

2. $x_i \in \mathcal{C}_j$. Comme $\mathcal{B}$ est une affectation consistante, $\mathcal{B}$ satisfait les contraintes apparaissant dans $\mathcal{C}_j$, en particulier $\{x_i, x_j\}$.
3. $x_i, x_j \in \mathcal{C}_i \cap \mathcal{C}_j$ qui est un cas particulier des deux cas précédents.

Donc, l'affectation définie par l'instanciation $\mathcal{A}$ étendue par $\mathcal{B}$, c'est-à-dire $\mathcal{A} \cup \mathcal{B}$, est une solution du sous-problème défini par les variables appartenant à $\mathcal{A}$ ou à $Desc(\mathcal{C}_j)$, puisque toutes les contraintes sont satisfaites. Ainsi, $\mathcal{A} \cup \mathcal{B}$ est une affectation consistante, et les valeurs dans $\mathcal{B}$ figurent nécessairement dans $AC(\mathcal{P}, \mathcal{A})$. □

Bien que permettant déjà un backtracking non chronologique, il est malgré tout possible d'étendre encore BTD avec le *Backjumping* au sens de [Gas79]. Ceci conduit alors à trois extensions directes selon qu'un filtrage est ou non utilisé et selon sa puissance (de type FC ou MAC) : **BTD-BJ**, **FC-BTD-BJ** et **MAC-BTD-BJ**. Cette phase de backjump est accomplie quand BTD revient en arrière au cluster $\mathcal{C}_i$ après avoir rencontré un échec durant la recherche d'une extension de l'affection courante sur la descendance de $\mathcal{C}_i$ enraciné en un fils $\mathcal{C}_j$ de $\mathcal{C}_i$. Elle consiste à revenir sur la variable la plus profonde qui appartient à $\mathcal{C}_i$ et $\mathcal{C}_j$. La correction de cette phase de backjump est évidente, puisque l'affectation courante des variables de l'intersection $\mathcal{C}_i \cap \mathcal{C}_j$ est un nogood.

Enfin, notons que dans la version présentée, l'algorithme BTD se limite à la construction d'une instanciation consistante partielle dont on a la garantie qu'elle pourra s'étendre à une solution du CSP traité. Pour le cas où une solution est recherchée, il suffit d'étendre l'affectation produite par BTD grâce à une recherche de type backtracking et en exploitant les goods et les nogoods comme de nouvelles contraintes. Cette modification de l'algorithme ne changera en rien les résultats de complexité que nous donnons ci-dessous, seule l'efficacité chronométrique pouvant alors s'en trouvée altérée.

2.2 Complexités en temps et en espace

Dans cette section, nous estimons d'abord les complexités en temps et en espace de l'algorithme BTD. Ensuite, nous comparons BTD avec les algorithmes Backtracking Chronologique et Tree-Clustering. Notons que les résultats qui suivent sont toujours valables si nous employons comme algorithme de base pour BTD un algorithme plus évolué comme FC ou MAC.

Dans ce qui suit, nous supposerons qu'une décomposition arborescente du graphe de contraintes (ou bien une approximation) est disponible. Les paramètres de la complexité seront donc relatifs notamment aux caractéristiques de cette décomposition supposée connue. Nous commençons par évaluer la complexité en espace de BTD :

Théorème 2.11 *BTD a une complexité en espace en $O(n.s.d^s)$ où s est la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$.*

Preuve : BTD ne mémorise que les goods et les nogoods. Les goods et les nogoods sont des instanciations sur les intersections $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$. Donc, si s est la taille de la plus grande de ces intersections, BTD a une complexité en espace en $O(n.s.d^s)$ parce que le nombre d'intersections $\mathcal{C}_i \cap \mathcal{C}_j$ est majoré par n tandis que le nombre de goods et de nogoods pour une intersection est majoré par d^s et la taille d'un good ou d'un nogood est au plus s. □

À présent, nous calculons la complexité en temps de BTD :

Théorème 2.12 *BTD a une complexité en temps en $O(n.s^2.m.\log(d).d^{w^+ + 1})$ avec $w^+ + 1$ la taille du plus grand $\mathcal{C}_k$ et s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$.*

Preuve :
Supposons que nous souhaitons étendre une instanciation sur $\mathcal{C}_j$. Il existe deux cas :

- Soit $\mathcal{C}_j = \mathcal{C}_1$. Trouver une instanciation consistante sur $\mathcal{C}_j$ s'avère alors, dans le pire des cas, de complexité en $O(m.d^{|\mathcal{C}_j|})$. Notons que le facteur m provient du nombre de contraintes à vérifier pour tester la consistance.
- Soit $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$. Soit $\mathcal{A}$ une affectation consistante sur Y ($Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$).
  Trouver une extension consistante de $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ sur $\mathcal{C}_j$ s'avère, dans le pire des cas, de complexité en $O(m.d^{|\mathcal{C}_j \setminus (\mathcal{C}_j \cap \mathcal{C}_i)|})$.

  BTD ne recherche une extension consistante de $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ qu'une seule fois (grâce aux goods et aux nogoods mémorisés). Comme il existe au plus $d^{|\mathcal{C}_i \cap \mathcal{C}_j|}$ affectations $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$, la complexité en temps pour trouver une extension consistante sur $\mathcal{C}_j$ est, dans le pire des cas, en $O(d^{|\mathcal{C}_j|})$.

Donc, si $w^+ + 1$ est la taille du plus grand $\mathcal{C}_i$, la recherche d'une extension consistante par BTD a une complexité en $O(n.m.d^{w^+ + 1})$, à laquelle doit être ajouté le coût de la gestion et de l'exploitation des goods et des nogoods. Comme ce coût est nul pour $\mathcal{C}_1$, nous nous focalisons sur le cas où $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$.
La comparaison entre $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ et un good (ou un nogood) mémorisé requiert $O(|\mathcal{C}_i \cap \mathcal{C}_j|)$. L'ajout ou la recherche d'un good ou d'un nogood est en $O(|\mathcal{C}_i \cap \mathcal{C}_j| \log(d^{|\mathcal{C}_i \cap \mathcal{C}_j|}))$. Par conséquent, la gestion et l'exploitation des goods et des nogoods ont une complexité en $O(d^{|\mathcal{C}_i|} |\mathcal{C}_i \cap \mathcal{C}_j| \log(d^{|\mathcal{C}_i \cap \mathcal{C}_j|}))$, étant donnés $\mathcal{C}_i$ et un des ces fils $\mathcal{C}_j$.
Donc, sur l'ensemble de la recherche, le coût est en $O(n.s.m.d^{w^+ + 1} \log(d^s))$.
Ainsi, la complexité en temps de BTD est en $O(n.m.d^{w^+ + 1} + n.s.m.d^{w^+ + 1} \log(d^s))$, i.e. une complexité en $O(n.s^2.m.d^{w^+ + 1}.\log(d))$. □

Les complexités en temps et en espace de BTD sont comparables à celles du Tree-Clustering. Nous allons, maintenant, montrer que BTD développe moins de nœuds (ou au pire autant) que le Backtracking Chronologique et que le Tree-Clustering. Afin de rendre possible ces comparaisons, nous considérons que BT et TC utilisent le même ordre variables/valeurs que BTD et que TC exploite la même décomposition arborescente (ou la même approximation) que BTD. L'emploi d'ordres compatibles permet de comparer facilement BT et BTD. Néanmoins, il est clair qu'un ordre compatible n'est pas forcément un bon ordre pour BT. Une comparaison plus générale entre BTD et BT (ou FC ou MAC) nécessite d'utiliser des ordres différents. Donc, notre analyse devra être étendue dans le futur afin de considérer des ordres différents. Nous comparons d'abord BT et BTD :

Théorème 2.13 *Étant donné un ordre compatible, BTD développe au plus autant de nœuds que BT.*

Preuve : L'emploi des goods et des nogoods permet à BTD d'éviter certaines redondances dans l'arbre de recherche. Donc, BTD développe au plus autant de nœuds que BT. □

Comme BT, BTD termine dès que la consistance du problème a été déterminée. D'un autre côté, TC construit toutes les affectations consistantes de chaque $\mathcal{C}_i$. De plus, quand BTD ne développe pas une instanciation consistante sur $\mathcal{C}_i$, il s'ensuit une économie en nombre de nœuds sur toute la descendance de $\mathcal{C}_i$. Ainsi, le théorème suivant montre le gain en nœuds de BTD par rapport à TC :

Théorème 2.14 *Étant donné un ordre compatible, BTD développe au plus autant de nœuds que TC utilisant BT pour la résolution de chaque $\mathcal{C}_i$.*

Preuve : BTD et TC développent, dans le pire des cas, le même nombre de nœuds pour $\mathcal{C}_1$. Pour tout autre $\mathcal{C}_j$ ($j \neq 1$), TC recherche systématiquement toutes les affectations consistantes sur $\mathcal{C}_j$, tandis que BTD ne construit que les instanciations consistantes sur $\mathcal{C}_j$ qui sont compatibles avec

l'affectation courante sur $\mathcal{C}_i$, le père de $\mathcal{C}_j$. Ainsi, BTD développe au plus autant de nœuds que TC. □

Enfin, pour conclure cette section, remarquons que si nous employons FC ou MAC en lieu et place de BT (et donc FC-BTD ou MAC-BTD au lieu de BTD), le théorème 2.13 est encore vérifié. De plus, au niveau de la complexité en temps, nous obtenons la complexité en temps théorique en multipliant le coût par un facteur dépendant du coût d'un filtrage, dans le même esprit que l'analyse de complexité proposée dans [Lar00].

2.3 Résultats expérimentaux

Dans cette section, nous estimons l'intérêt pratique d'une méthode comme BTD.

2.3.1 Jeux de tests

Les premières expériences concernent les réseaux dont la tree-width n'est pas forcément petite. Pour eux, nous espérons que BTD se montrera aussi efficace que n'importe quel algorithme énumératif classique. Pour cette serie d'expériences, nous employons des instances aléatoires générées suivant le modèle de Prosser [Pro94]. Nous qualifions par la suite ces instances aléatoires de "classiques". Les secondes expériences concernent des CSPs structurés : nous espérons que BTD exploitera efficacement les propriétés topologiques du réseau, quand ces propriétés sont relatives à une décomposition arborescente, c'est-à-dire un CSP avec une petite tree-width. Pour ces expérimentations, nous utilisons des instances aléatoires structurées générées suivant le modèle défini ci-dessous. Enfin, nous analysons le comportement de notre méthode sur quelques instances du monde réel.

Les problèmes aléatoires classiques possèdent un inconvénient : ils ne possèdent généralement pas de propriétés structurelles intéressantes. C'est pourquoi nous définissons un nouveau modèle de CSPs aléatoires. Les instances produites selon ce modèle possède un graphe de contraintes structuré. En particulier, ce graphe est triangulé, ce qui permet de connaître exactement sa tree-width et donc la complexité des méthodes structurelles comme l'algorithme Tree-Clustering.
Dans ce modèle, nous considérons cinq paramètres n, d, r_{max}, t et s_{max}. Il construit un CSP binaire de la classe $(n, d, r_{max}, t, s_{max})$ avec n variables qui ont chacune un domaine de taille d et dont le graphe de contraintes vérifie les propriétés suivantes :

- chaque variable v appartient à au moins une clique maximale de taille supérieure à 1,
- chaque clique a une taille au plus r_{max},
- l'intersection entre deux cliques est de taille au plus s_{max},
- les cliques forment une arborescence de cliques et le graphe de contraintes est triangulé.

Pour construire de telles instances, nous choisissons d'abord un ensemble de r_{max} variables pour former la clique racine. Ensuite, tant qu'il reste des variables, nous procédons ainsi :

1. choisir aléatoirement la clique parent $\mathcal{C}_i$,
2. choisir aléatoirement la taille de l'intersection entre $\mathcal{C}_i$ et son fils $\mathcal{C}_j$ (la taille est comprise entre 1 et s_{max}),
3. choisir aléatoirement la taille de la clique $\mathcal{C}_j$ (la taille est d'au moins 3 et est bornée par la taille de l'intersection plus un et r_{max}),
4. choisir aléatoirement les variables de $\mathcal{C}_i$ qui participent au séparateur $\mathcal{C}_i \cap \mathcal{C}_j$.

Nous associons à chaque contrainte une relation pour laquelle t tuples sont interdits ($0 \leq t \leq d^2$). Le principale défaut de ce générateur est que le nombre de contraintes varie suivant les instances produites.

2.3.2 Protocole expérimental

Avant de décrire le protocole expérimental proprement dit, nous présentons les algorithmes implémentés ainsi que la méthode employée pour calculer l'approximation d'une décomposition arborescente.

2.3.2.1 Les algorithmes de résolution implémentés

Nous implémentons différentes versions de l'algorithme BTD. Pour des raisons évidentes de temps, nous n'expérimentons pas BTD dans sa version de base (i.e. avec le Backtracking Chronologique). La première version, notée FC-BTD, correspond à une simple implémentation de l'algorithme BTD basé sur l'algorithme Forward-Checking. Dans la seconde, notée FC-BTD-BJ, FC-BTD se voit doter de la phase de backjump supplémentaire (voir le paragraphe 2.1.4 pour plus de détails). Ensuite, nous implémentons deux versions, notées respectivement FC-BTD$^-$ et FC-BTD-BJ$^-$, qui correspondent respectivement à FC-BTD et FC-BTD-BJ dépourvus de la mémorisation et de l'emploi des goods et des nogoods. Ces deux dernières versions nous sont utiles pour estimer la contribution des goods et des nogoods aux résultats obtenus par FC-BTD et FC-BTD-BJ. Autrement dit, ces versions correspondent à FC dans lequel le choix de la prochaine variable à instancier est en partie guidé par l'ordre d'énumération compatible de BTD. De même, nous définissons les versions correspondantes de BTD basées sur l'algorithme MAC.

Afin d'établir des comparaisons avec les algorithmes classiques de la littérature, nous implémentons FC [HE80], FC-CBJ [Pro93], FC-NR [SV93, SV94] et MAC [SF94]. Pour MAC, l'arc-consistance est accomplie grâce à l'algorithme AC-2001 ([BR01]). Dans FCNR, la mémorisation des nogoods est limitée aux nogoods portant sur au plus deux variables.

Dans le but de comparer le nombre de nœuds développés et l'espace requis par BTD et le Tree-Clustering, nous implémentons une version partielle de Tree-Clustering. Par version partielle, nous entendons que nous calculons seulement toutes les solutions de chaque cluster. Nous nous contentons de mesurer l'espace mémoire requis sans mémoriser réellement ces solutions (pour des raisons évidentes de capacité mémoire). De plus, nous ne résolvons pas le CSP acyclique obtenu à partir de calculs précédents, parce que cette étape ne présente aucun intérêt pour notre étude comparative. La version implémentée du Tree-Clustering utilise l'algorithme Forward-Checking pour la recherche des solutions de chaque cluster. Nous la notons TC-FC. Bien sûr, BTD et TC-FC exploitent la même décomposition arborescente (ou la même approximation).

Concernant le choix de la prochaine variable à instancier, nous employons l'heuristique dom/deg pour tous les algorithmes sauf FC-NR, qui utilisera l'heuristique dom/st (cette heuristique donnant de meilleurs résultats avec FC-NR). La sélection de la prochaine variable s'effectue :

- parmi toutes les variables non instanciées du problème pour FC, FC-NR, FC-CBJ ou MAC,
- parmi toutes les variables non instanciées du cluster courant pour les différentes versions de BTD.

Enfin, nous terminons par deux remarques concernant les implémentations. D'une part, il est important de noter que les différentes versions de FC-BTD (respectivement de MAC-BTD) emploient exactement le même ordre d'instanciation des variables. D'autre part, l'implémentation de FC (respectivement MAC) dans FC-BTD (resp. MAC-BTD) est exactement la même que celle de FC (resp. MAC) seul. Par conséquent, toute amélioration de l'implémentation de l'un entraîne la même amélioration pour l'autre.

2.3.2.2 Approximation d'une décomposition arborescente par triangulation

Comme trouver une décomposition arborescente est un problème NP-dur, nous nous contentons d'employer une approximation de décomposition arborescente en triangulant le graphe de

contraintes.

Nous avons essayé différents algorithmes de triangulation dont les algorithmes LEX-M [RTL76], LB-TRIANG [Ber99] et Fill-in Computation [TY84]. Ce dernier a été initialement proposé pour tester si un graphe est triangulé ou non. Cependant, durant ce test, il ajoute les arêtes nécessaires pour rendre le graphe triangulé. Les deux premiers algorithmes produisent une triangulation minimale (une triangulation E' d'un graphe $G = (V, E)$ est minimale s'il n'existe pas de triangulation E'' telle que $E'' \subset E'$). Ils ont une complexité en temps en $O(nm)$ avec n et m respectivement les nombres de sommets et d'arêtes du graphe, tandis que celle de la méthode basée sur l'algorithme Fill-in Computation est linéaire en $O(n + m')$ (m' étant le nombre d'arêtes du graphe triangulé). D'après des expérimentations sur les problèmes classiques, l'algorithme LEX-M fournit les meilleurs résultats pour l'algorithme BTD. Par conséquent, pour les résultats suivants, nous utilisons l'algorithme LEX-M pour calculer une triangulation.

À partir de cette triangulation, nous obtenons des cliques et des séparateurs de taille moyenne raisonnable. Autrement dit, le temps et l'espace mémoire requis par BTD sont réalisables en pratique. Par contre, la taille maximale des séparateurs peut être trop importante, c'est-à-dire que BTD requiert trop de mémoire. Aussi, afin d'éviter ce problème, nous proposons de limiter la taille de séparateurs par un paramètre s_{max} donné, à l'image de [DF01]. Ce compromis s'effectue au détriment de la taille des séparateurs (et donc du temps). D'abord, nous calculons normalement l'arbre de cliques. Puis, nous parcourons l'arbre de cliques obtenu en largeur d'abord. Si le fils $\mathcal{C}_j$ a une intersection avec son père $\mathcal{C}_i$ de taille inférieure à s_{max}, le fils et son père restent inchangés. Sinon, nous fusionnons le père $\mathcal{C}_i$ et son fils $\mathcal{C}_j$. Le cluster obtenu remplace alors $\mathcal{C}_i$ dans l'arbre (aussi nous appellerons $\mathcal{C}_i$ ce nouveau cluster). De plus, les fils de $\mathcal{C}_j$ deviennent des fils de $\mathcal{C}_i$. Enfin, il est important de remarquer que ces modifications n'affectent pas la taille de l'intersection entre $\mathcal{C}_i$ et les frères de $\mathcal{C}_j$.

Pour les résultats fournis, nous limitons la taille des séparateurs à 5. Avec une telle taille, la taille des séparateurs n'est ni trop petite, ni trop grande.

2.3.2.3 Protocole expérimental

Au niveau matériel, les expérimentations ont été réalisées :

- sur un PC sous Linux équipé d'un processeur Athlon XP 1800+ d'AMD et de 512 Mo de mémoire vive pour les problèmes aléatoires classiques et structurés,
- sur un PC sous Linux équipé d'un processeur Pentium III 550 MHz d'Intel et de 256 Mo de mémoire pour les instances du monde réel.

Pour les instances aléatoires, nous résolvons cent problèmes par classe. Les résultats fournis correspondent alors aux moyennes des résultats obtenus sur ces cent problèmes. Par contre, pour les instances réelles, nous présentons les résultats instance par instance.

Précisons que les temps présentés pour les méthodes structurelles (BTD et TC-FC) incluent les temps de calcul de l'approximation de la décomposition arborescente. Pour la résolution des instances aléatoires structurées, nous imposons à tous les algorithmes une limite de temps. Si le temps de résolution d'un problème excède une demi-heure, la recherche est stoppée et la consistance de ce problème est considérée indéterminée. Notons, enfin, que les tests de contraintes tiennent compte :

- des contraintes initiales et des contraintes ajoutées dans le cas de l'algorithme FC-NR,
- des contraintes initiales pour tous les autres algorithmes.

2.3.3 Résultats expérimentaux pour les CSPs aléatoires classiques

2.3.3.1 Comparaisons des différentes versions de BTD

Avant de comparer BTD aux algorithmes classiques comme FC ou MAC, nous étudions le comportement de notre algorithme. D'abord, nous estimons la contribution du backjumping en comptant le nombre de nœuds développés par FC-BTD qui ne sont pas visités par FC-BTD-BJ. Nous observons alors qu'il n'y a pas de gain pour la plupart des classes et un faible gain pour la classe (50,25,123,439) (les classes expérimentées sont données dans le tableau 2.1). Mais, même lorsque le gain existe, il est insignifiant. Comme un good ou un nogood est mémorisé chaque fois que BTD revient en arrière d'un cluster à son père, nous pouvons conclure, d'après le faible nombre de goods et de nogoods mémorisés, que FC-BTD et FC-BTD-BJ ne visitent pas souvent la descendance du cluster racine. Par conséquent, la phase de backjump est rarement exploitée, d'où la similitude des résultats obtenus par FC-BTD et FC-BTD-BJ.

Ensuite, nous mesurons la contribution des goods et des nogoods en comptant le nombre de nœuds développés par FC-BTD-BJ$^-$ qui ne sont pas visités par FC-BTD-BJ. Comme pour la première comparaison, le gain est faible ou nul. En effet, à cause du faible nombre de goods et de nogoods mémorisés, seulement quelques goods ou nogoods sont utilisés par FC-BTD-BJ pour élaguer l'arbre de recherche. Ainsi, FC-BTD-BJ$^-$ et FC-BTD-BJ présentent des résultats similaires. Pour information, nous obtenons des résultats de même nature avec MAC-BTD. Comme les différentes variantes de BTD basées sur FC (respectivement sur MAC) obtiennent des résultats comparables, pour la suite des comparaisons sur les problèmes aléatoires classiques, nous ne présenterons que les résultats de FC-BTD-BJ (resp. MAC-BTD-BJ).

2.3.3.2 Comparaisons entre FC-BTD-BJ et FC et entre MAC-BTD-BJ et MAC

Le tableau 2.1 présente le nombre de nœuds, de tests de contraintes, et le temps de résolution pour FC et FC-BTD-BJ. Nous observons que FC-BTD-BJ et FC obtiennent de résultats comparables. Pour certaines classes, FC-BTD-BJ améliore même les résultats de FC, en développant moins de nœuds et en réalisant moins de tests de contraintes que FC.

Classe (n, d, m, t)	FC			FC-BTD-BJ		
	Nb. nœuds	Nb. tests (milliers)	Temps (ms)	Nb. nœuds	Nb. tests (milliers)	Temps (ms)
(50,15,184,112)	223 588	7 347	795	229 901	7 522	805
(50,15,245,93)	1 742 077	64 695	7 072	1 690 389	62 741	6 825
(50,15,306,78)	6 695 576	275 447	30 483	6 516 523	268 223	29 340
(50,15,368,68)	19 899 917	865 863	99 673	20 202 681	880 491	99 842
(50,25,123,439)	148 793	5 969	576	183 304	7 107	652
(50,25,150,397)	819 793	34 743	3 462	915 561	38 713	3 693
(75,10,277,43)	464 382	12 279	1 591	486 416	12 828	1 683

TAB. 2.1 : [CSPs aléatoires classiques] Nombre de nœuds, nombre de tests de contraintes et temps de résolution (en millisecondes) pour FC et FC-BTD-BJ.

Des résultats de même nature sont observés avec MAC et MAC-BTD-BJ, comme le montre le tableau 2.2.

Classe (n, d, m, t)	MAC			MAC-BTD-BJ		
	Nb. nœuds	Nb. tests (milliers)	Temps (ms)	Nb. nœuds	Nb. tests (milliers)	Temps (ms)
(50,15,184,112)	10 570	4 750	1 472	10 589	4 767	1 485
(50,15,245,93)	115 272	53 354	18 612	111 641	51 618	17 852
(50,15,306,78)	577 928	263 295	101 093	560 541	255 317	97 821
(50,15,368,68)	2 024 325	936 054	403 698	2 053 352	948 798	404 879
(50,25,123,439)	2 912	2 601	599	2 703	2 476	563
(50,25,150,397)	27 628	21 813	5 848	26 639	21 335	5 682
(75,10,277,43)	10 655	3 533	1 211	10 631	3 535	1 210

TAB. 2.2 : `[CSPs aléatoires classiques]` Nombre de nœuds, nombre de tests de contraintes et temps de résolution (en millisecondes) pour MAC et MAC-BTD-BJ.

2.3.3.3 Comparaisons entre FC-BTD-BJ, FC-CBJ et FC-NR

Comme FC-BTD-BJ exploite des techniques de backjumping et de "forward-jumping", nous comparons notre algorithme avec un algorithme classique de backjumping, à savoir FC-CBJ. Le tableau 2.3 fournit le nombre de nœuds, celui de tests de contraintes et le temps pour FC-CBJ. Nous notons que FC-CBJ développe souvent moins de nœuds que FC-BTD-BJ. Cependant, si nous considérons le temps de calculs, nous constatons que FC-BTD-BJ est plus rapide que FC-CBJ pour toutes les classes testées sauf une. Une partie de ce résultat peut s'expliquer par le coût du calcul des conflits qui s'avère trop important comparé au nombre de nœuds économisés par FC-CBJ.

Classe (n, d, m, t)	FC-CBJ			FC-BTD-BJ		
	Nb. nœuds	Nb. tests (milliers)	Temps (ms)	Nb. nœuds	Nb. tests (milliers)	Temps (ms)
(50,15,184,112)	214 314	7 089	949	229 901	7 522	805
(50,15,245,93)	1 707 839	63 629	8 462	1 690 389	62 741	6 825
(50,15,306,78)	6 612 237	272 582	35 763	6 516 523	268 223	29 340
(50,15,368,68)	19 722 533	859 100	116 053	20 202 681	880 491	99 842
(50,25,123,439)	127 093	5 208	632	183 304	7 107	652
(50,25,150,397)	761 514	32 648	4 046	915 561	38 713	3 693
(75,10,277,43)	426 619	11 406	1 846	486 416	12 828	1 683

TAB. 2.3 : `[CSPs aléatoires classiques]` Nombre de nœuds, nombre de tests de contraintes et temps de résolution (en millisecondes) pour FC-CBJ et FC-BTD-BJ.

FC-BTD-BJ mémorisant des goods et des nogoods, il semble naturel de vouloir le comparer à un algorithme mémorisant des nogoods classiques comme FC-NR. Le tableau 2.4 présente le nombre de nœuds, celui de tests de contraintes et le temps pour FC-NR. Nous constatons qu'à l'image de FC-CBJ, FC-NR développe moins de nœuds que FC-BTD-BJ. Cependant, il se révèle plus lent, à cause principalement des tests de contraintes supplémentaires engendrés par la mémorisation des nogoods classiques.

2.3.3.4 Comparaisons entre BTD et Tree-Clustering

Nous comparons l'espace requis par FC-BTD-BJ et notre version partielle du Tree-Clustering. Afin de mesurer la quantité de mémoire requise, nous comptons une unité par valeur affectée contenue dans l'affectation partielle mémorisée. Dans le cas de BTD, cette affectation partielle

Classe (n, d, m, t)	FC-NR			FC-BTD-BJ		
	Nb. nœuds	Nb. tests (milliers)	Temps (ms)	Nb. nœuds	Nb. tests (milliers)	Temps (ms)
(50,15,184,112)	171 664	9 984	1 331	229 901	7 522	805
(50,15,245,93)	1 577 806	81 847	11 758	1 690 389	62 741	6 825
(50,15,306,78)	6 496 131	287 455	44 615	6 516 523	268 223	29 340
(50,15,368,68)	19 637 545	870 618	140 340	20 202 681	880 491	99 842
(50,25,123,439)	63 587	5 479	635	183 304	7 107	652
(50,25,150,397)	522 592	52 620	6 181	915 561	38 713	3 693
(75,10,277,43)	254 649	12 496	1 912	486 416	12 828	1 683

TAB. 2.4 : `[CSPs aléatoires classiques]` Nombre de nœuds, nombre de tests de contraintes et temps de résolution (en millisecondes) pour FC-NR et FC-BTD-BJ.

correspond à un good ou à un nogood structurel. Dans le cas de TC, il s'agit d'une affectation consistante portant sur toutes les variables d'un cluster (pour la version de base de TC), ou d'une affectation d'un séparateur qui peut être étendue de façon consistante sur le cluster correspondant (version améliorée de TC). Par exemple, mémoriser un good portant sur cinq variables aura un coût de cinq unités.

Le tableau 2.5 présente les quantités de mémoire employées par FC-BTD-BJ (pour mémoriser les goods et les nogoods), et par TC-FC (pour enregistrer les solutions de chaque sous-problème respectivement sur les séparateurs et sur les clusters), le nombre de nœuds développés et le temps de résolution (en millisecondes) de TC-FC. Nous observons que TC-FC requiert significativement plus de mémoire que FC-BTD-BJ, car FC-BTD-BJ ne mémorise qu'une partie des goods que TC-FC enregistre. Notons que pour certaines classes comme la classe (50,25,123,439), TC-FC nécessite trop de mémoire en pratique. De plus, TC-FC développe nettement plus de nœuds et se révèle plus lent que FC-BTD-BJ. Par conséquent, il semble difficile d'exploiter le Tree-Clustering en pratique.

Classe (n, d, m, t)	FC-BTD-BJ	TC-FC			
	Mémoire	Mémoire		Nb nœuds	Temps (ms)
		séparateur	cluster		
(50,15,184,112)	9,9	163 523	1 840 482	942 758	1 845
(50,15,245,93)	1,3	33 217	401 269	2 438 672	8 433
(50,15,306,78)	0,5	11 620	199 244	12 932 108	51 388
(50,15,368,68)	0,1	7 052	53 470	25 859 906	114 721
(50,25,123,439)	19,2	1 560 479	375 943 617	89 379 304	37 551
(50,25,150,397)	15,3	1 456 900	137 799 883	35 497 081	18 983
(75,10,277,43)	16,8	65 551	242 102 465	43 105 943	15 521

TAB. 2.5 : `[CSPs aléatoires classiques]` Comparaisons entre FC-BTD-BJ et Tree-Clustering basé sur FC.

2.3.3.5 Résumé

FC-BTD et MAC-BTD obtiennent des résultats qui sont comparables avec ceux de FC (ou meilleurs que ceux de FC-CBJ ou de FC-NR) et de MAC respectivement car peu de goods ou de nogoods structurels sont produits. Il paraît difficile, voire impossible, d'employer en pratique le Tree-Clustering, à cause de l'espace requis.

2.3.4 Résultats expérimentaux pour les CSPs aléatoires structurés

2.3.4.1 Comparaisons des différentes versions de BTD

Comme pour les problèmes classiques, avant d'effectuer les comparaisons entre BTD et les algorithmes classiques comme FC, FC-CBJ ou MAC, nous étudions le comportement de notre algorithme. D'abord, afin de comparer FC-BTD et FC-BTD-BJ, nous évaluons la contribution du backjumping en comptant le nombre de nœuds développés par FC-BTD qui ne sont pas visités par FC-BTD-BJ. La figure 2.3 présente le nombre de nœuds développés par FC-BTD et FC-BTD-BJ. Nous constatons, sur cette figure, que FC-BTD-BJ développe significativement moins de nœuds que FC-BTD. Cette économie en terme de nombre de nœuds varie entre 18% et 38%. Cependant, l'emploi du backjumping a un coût. En effet, d'après la figure 2.4 (qui représente le temps de résolution pour FC-BTD et FC-BTD-BJ), nous observons que le gain en temps est légèrement moins important que celui en nœuds. Il est compris entre 5% et 25%.

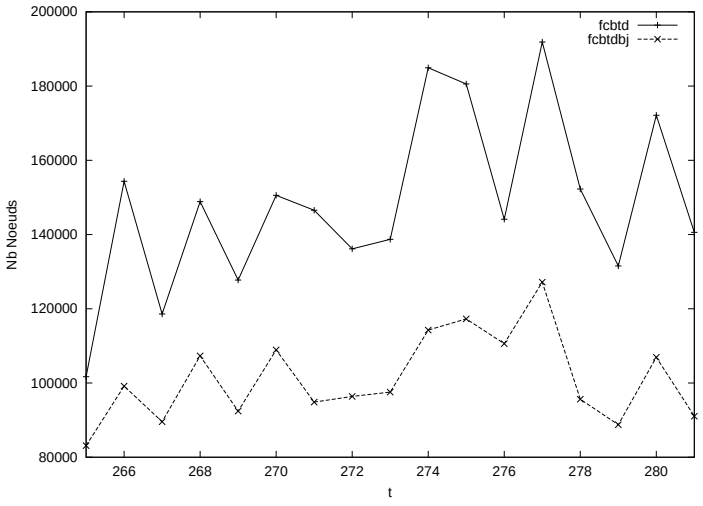

FIG. 2.3 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Nombre de nœuds développés par FC-BTD et FC-BTD-BJ.

Pour estimer la contribution des goods et des nogoods, nous comptons le nombre de nœuds développés par FC-BTD-BJ$^-$ qui ne sont pas visités par FC-BTD-BJ. D'après la figure 2.5, il apparaît que FC-BTD-BJ développe toujours moins de nœuds que FC-BTD-BJ$^-$ et que ce gain peut être très important dans certains cas, en particulier à proximité du seuil de satisfaisabilité. Les deux algorithmes ne diffèrent qu'au niveau de la mémorisation des goods et des nogoods. Il s'ensuit que le gain en nœuds est obtenu grâce à l'exploitation des goods et des nogoods. Cette économie se traduit par un gain en temps considérable, comme le montre la figure 2.6 (qui représente le temps de résolution de FC-BTD-BJ et FC-BTD-BJ$^-$).

Des expériences similaires ont été menées avec FC-BTD et FC-BTD$^-$. D'abord, il résulte de ces expérimentations que FC-BTD$^-$ est incapable de résoudre certaines instances en moins d'une demi-heure. Le tableau 2.6 précise le nombre d'instances non résolues. Donc, afin de comparer FC-BTD et FC-BTD$^-$, nous ne tenons compte que des problèmes résolus par FC-BTD$^-$. La figure 2.7 montre le nombre de nœuds développés par FC-BTD et FC-BTD$^-$. Nous pouvons alors observer que FC-BTD développe moins de nœuds que FC-BTD$^-$, grâce à l'emploi des goods et des nogoods. De plus, la différence entre FC-BTD et FC-BTD$^-$ est plus importante que celle entre FC-BTD-BJ et FC-BTD-BJ$^-$. Cet écart met en évidence l'existence d'un grand nombre de redondance dans

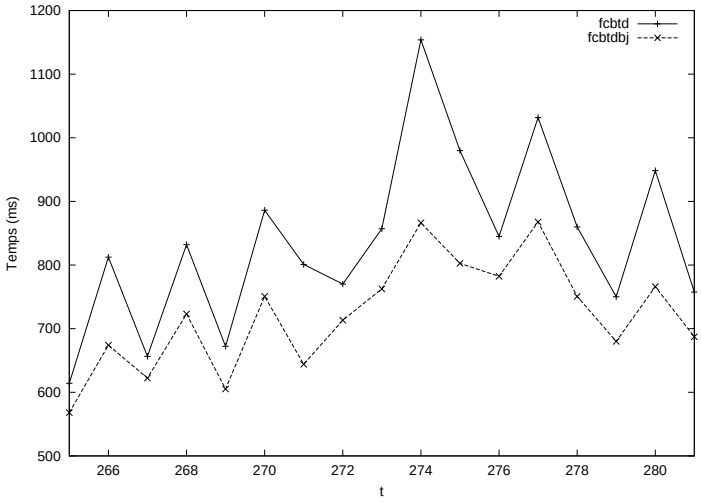

FIG. 2.4 : `[CSPs aléatoires structurés` $(50, 25, 15, t, 5)$`]` Temps de résolution (en millisecondes) pour FC-BTD et FC-BTD-BJ.

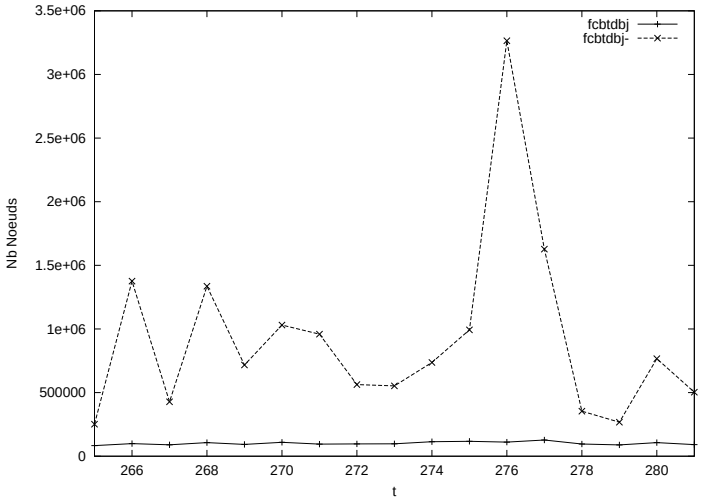

FIG. 2.5 : `[CSPs aléatoires structurés` $(50, 25, 15, t, 5)$`]` Nombre de nœuds développés par FC-BTD-BJ et FC-BTD-BJ$^-$.

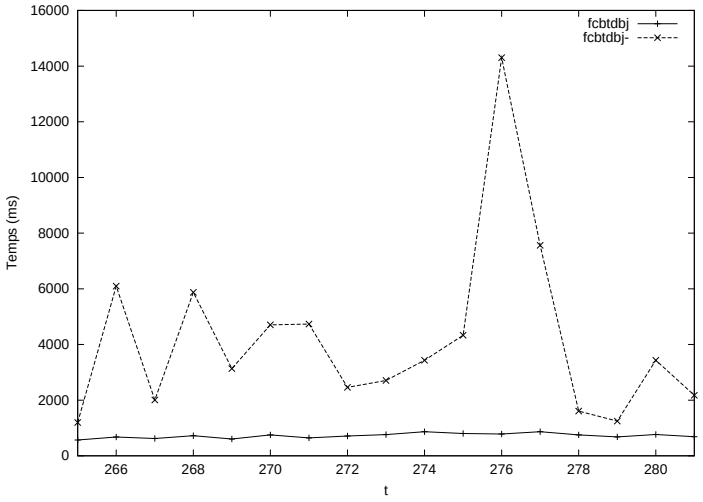

FIG. 2.6 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Temps de résolution (en millisecondes) pour FC-BTD-BJ et FC-BTD-BJ$^-$.

l'arbre de recherche développé par FC-BTD$^-$, ce qui souligne d'autant plus la contribution des goods et des nogoods et/ou de la phase de backjumping (car FC-BTD-BJ$^-$ n'est pas aussi pénalisé que FC-BTD$^-$).

À la vue des résultats précédents, nous focaliserons notre étude sur FC-BTD-BJ, pour toutes les prochaines comparaisons.

2.3.4.2 Comparaisons entre FC-BTD-BJ et FC et entre MAC-BTD-BJ et MAC

FC et MAC se révèlent incapables de résoudre certains problèmes en moins d'une demi-heure. Aussi, pour comparer FC (respectivement MAC) et FC-BTD-BJ (resp. MAC-BTD-BJ), nous ne comptabilisons que les résultats des instances que FC (resp. MAC) est capable de résoudre en moins d'une demi-heure. Le tableau 2.6 donne le nombre de problèmes non résolus par FC (resp. MAC).
La figure 2.8 présente le temps de résolution de FC, FC-BTD-BJ et FC-BTD-BJ$^-$. Nous constatons que FC-BTD-BJ est nettement plus rapide que FC. En effet, le rapport du temps de FC par celui de FC-BTD-BJ est compris entre 7 et 130. Nous économisons du temps non seulement grâce aux goods et aux nogoods, mais aussi grâce au backjumping. En effet, l'apport du backjumping est établi par le temps de résolution de FC-BTD-BJ$^-$, qui s'avère meilleur que celui de FC dans la plupart des cas.

La même série d'expériences avec MAC et MAC-BTD-BJ fournit des résultats similaires (voir la figure 2.9). MAC-BTD-BJ est de 5 à 24 fois plus rapide que MAC.

2.3.4.3 Comparaisons entre FC-BTD-BJ, FC-CBJ et FC-NR

Comme FC-BTD-BJ exploite des techniques de backjumping et de "forward-jumping", nous devons comparer FC-BTD-BJ avec un algorithme qui emploie du backjumping comme FC-CBJ. De même, FC-BTD-BJ tirant profit des goods et des nogoods structurels, il est nécessaire de le comparer à FC-NR. Les figures 2.10 et 2.11 représentent le nombre de nœuds et le temps de résolution pour FC-CBJ, FC-NR et FC-BTD-BJ. Au niveau du nombre de nœuds, ni FC-CBJ ni FC-BTD-BJ n'est

	FC-BTD		FC-BTD$^-$			FC			MAC		
t	C	I	C	I	NR	C	I	NR	C	I	NR
265	70	30	70	30	0	69	30	1	67	30	3
266	61	39	61	38	1	57	39	4	56	37	7
267	63	37	62	36	2	63	37	0	61	36	3
268	57	43	57	43	0	55	42	3	55	42	3
269	63	37	62	37	1	59	37	4	57	35	8
270	60	40	60	39	1	53	40	7	53	40	7
271	53	47	51	46	3	46	47	7	46	47	7
272	51	49	49	49	2	47	49	4	46	49	5
273	51	49	50	47	3	47	47	6	45	45	10
274	39	61	38	60	2	34	61	5	33	61	6
275	37	63	37	62	1	34	61	5	32	60	8
276	29	71	27	70	3	28	71	1	26	71	3
277	39	61	36	57	7	38	61	1	34	61	5
278	35	65	34	65	1	31	65	4	25	64	11
279	41	59	40	57	3	36	57	7	34	57	9
280	24	76	24	73	3	22	75	3	21	75	4
281	27	73	26	72	2	25	73	2	25	72	3

TAB. 2.6 : `[CSPs aléatoires structurés` $(50, 25, 15, t, 5)$`]` Nombre de problèmes consistants (C), inconsistants (I) et non résolus (NR).

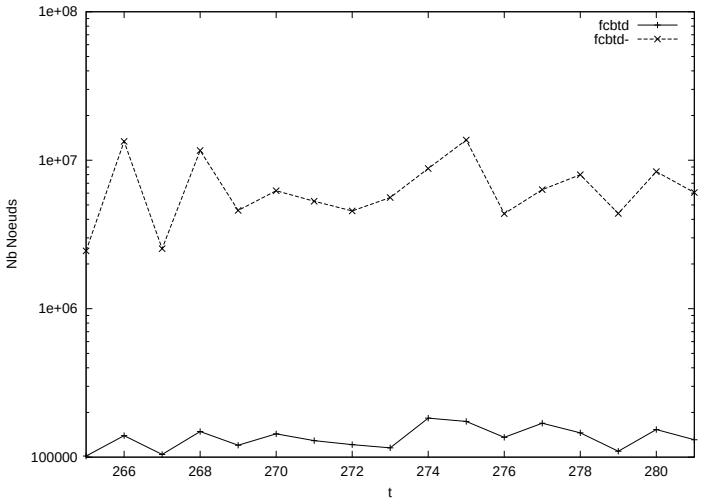

FIG. 2.7 : `[CSPs aléatoires structurés` $(50, 25, 15, t, 5)$`]` Nombre de nœuds développés par FC-BTD et FC-BTD$^-$ (avec une échelle logarithmique).

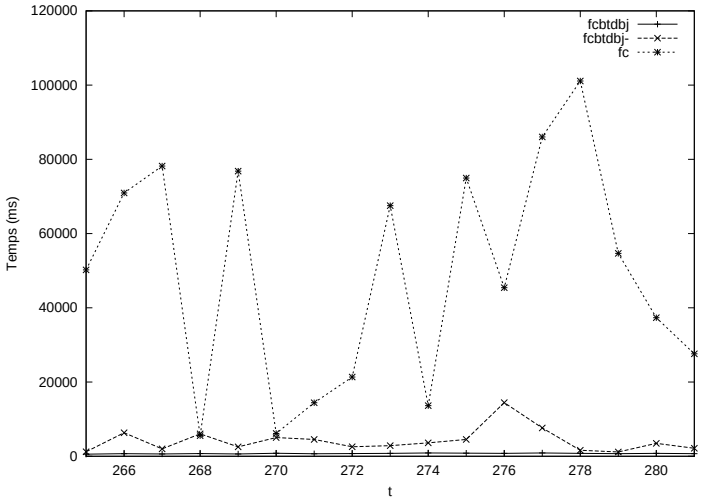

FIG. 2.8 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Temps de résolution (en millisecondes) pour FC-BTD-BJ, FC-BTD-BJ$^-$ et FC.

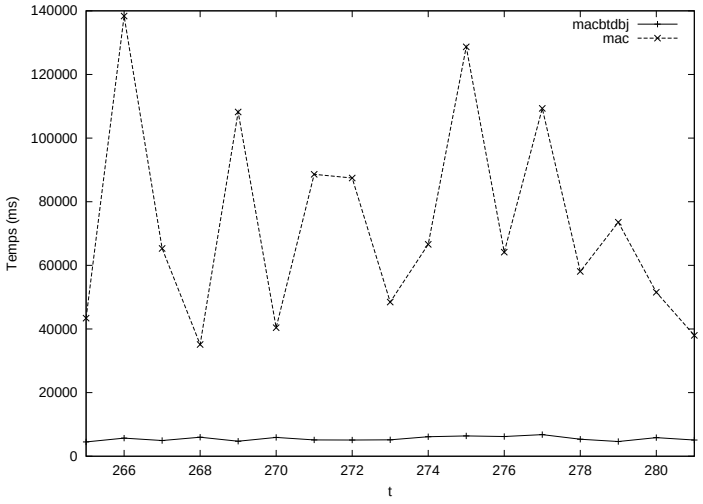

FIG. 2.9 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Temps de résolution (en millisecondes) pour MAC et MAC-BTD-BJ.

toujours meilleur que l'autre. Par contre, FC-BTD-BJ se révèle plus rapide que FC-CBJ. Cet écart de temps s'explique en grande partie par le coût du calcul des conflits dans FC-CBJ qui n'est pas compensé par l'économie en nœuds réalisée grâce au backjumping.

FC-NR développe moins de nœuds que FC-BTD-BJ. Hélas, il est pénalisé par la structure du problème, qui, apparamment, est peu propice à la production de nogoods portant seulement sur une ou deux variables. Ainsi, son temps de résolution est supérieur à celui de FC-BTD-BJ.

Notons enfin que FC-CBJ et FC-NR réussissent tout de même à résoudre toutes les instances.

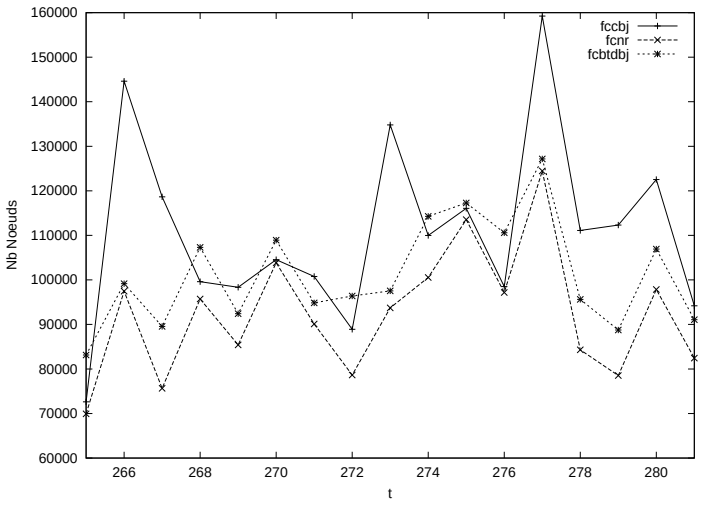

FIG. 2.10 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Nombre de nœuds développés par FC-CBJ, FC-NR et FC-BTD-BJ.

2.3.4.4 Comparaisons entre BTD et Tree-Clustering

Comme pour les problèmes aléatoires classiques, nous comparons l'espace requis par FC-BTD-BJ et par notre implémentation partielle du Tree-Clustering. Le tableau 2.5 montre la quantité de mémoire nécessaire à FC-BTD-BJ (pour mémoriser les goods et les nogoods) et par TC-FC (pour mémoriser les solutions de chaque sous-problème respectivement sur les séparateurs et les clusters) ainsi que le nombre de nœuds développés et le temps de résolution (en millisecondes) pour TC-FC. Nous observons que FC-BTD-BJ surclasse TC-FC en nécessitant significativement moins de mémoire. De plus, FC-BTD-BJ développe moins de nœuds et est plus rapide que TC-FC. Aussi, comme pour les problèmes aléatoires classiques, l'utilisation du Tree-Clustering semble difficilement envisageable dans la pratique.

2.3.4.5 Résumé

Parmi les différentes variantes de FC-BTD (respectivement MAC-BTD), la meilleure est incontestablement FC-BTD-BJ (respectivement MAC-BTD-BJ). Ainsi, FC-BTD-BJ et MAC-BTD-BJ s'avèrent plus rapides que FC et MAC respectivement. Notons, de plus, que FC et MAC se montrent incapables de résoudre certaines instances en moins d'une demi-heure. Concernant FC-CBJ, FC-BTD-BJ est plus rapide que FC-CBJ bien que les deux algorithmes développent un nombre de nœuds sensiblement équivalent. De même, il se révèle meilleur en temps que FC-NR. Enfin, FC-BTD-BJ requiert moins de mémoire et est plus rapide que TC-FC.

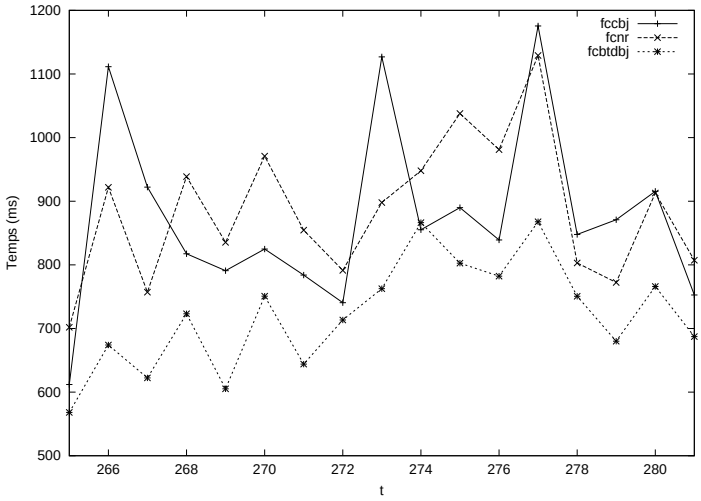

FIG. 2.11 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Temps de résolution (en millisecondes) pour FC-CBJ, FC-NR et FC-BTD-BJ.

t	FC-BTD-BJ	TC-FC			
	Mémoire	Mémoire		Nb nœuds	Temps
		séparateur	cluster		(ms)
265	3 599	563 376	16 269 923	4 329 007	5 200
266	4 054	544 683	15 741 988	4 081 041	4 643
267	3 471	391 140	13 536 010	3 630 053	4 300
268	4 174	500 454	14 331 723	3 779 950	4 347
269	3 218	426 517	12 862 473	3 477 518	4 092
270	5 567	457 120	13 071 460	3 505 728	4 043
271	5 005	413 560	13 010 768	3 451 125	3 916
272	4 273	453 395	11 745 237	3 192 403	3 748
273	5 476	401 098	11 476 495	3 083 502	3 575
274	9 008	444 808	10 237 218	2 805 207	3 351
275	5 289	393 569	9 107 353	2 575 782	3 180
276	5 134	342 977	8 301 716	2 385 563	2 987
277	8 408	379 848	9 794 940	2 712 032	3 192
278	5 910	350 243	8 589 484	2 384 354	2 836
279	6 734	416 477	8 265 270	2 307 709	2 716
280	8 304	319 735	7 267 237	2 066 851	2 502
281	5 637	247 736	6 232 299	1 790 943	2 203

TAB. 2.7 : [CSPs aléatoires structurés $(50, 25, 15, t, 5)$] Quantité de mémoire requise par FC-BTD-BJ et Tree-Clustering basé sur FC.

2.3.5 Instances du monde réel

Le tableau 2.8 présente les résultats obtenus pour certaines instances du CELAR issues de l'archive FullRLFAP. Dans plusieurs cas, MAC-BTD-BJ réalise soit moins de tests de contraintes soit autant que MAC, sauf pour les problèmes SCEN#02 et SCEN#05. Pour SCEN#02, MAC-BTD-BJ réalise quelques tests supplémentaires. Par contre, pour SCEN#05, MAC-BTD-BJ réalise une économie en nombre de tests significative. Au niveau du temps d'exécution, MAC-BTD-BJ et MAC sont comparables, sauf pour l'instance SCEN#05, où MAC-BTD-BJ se révèle nettement plus rapide. Pour ce problème, MAC-BTD-BJ apparaît significativement plus rapide que MAC grâce au nombre réduit de tests de contraintes.
Nous ne présenterons pas les résultats concernant TC-FC car TC-FC est incapable de trouver toutes les solutions du cluster racine, sauf pour les problèmes trivialement inconsistants.

Instance	MAC		MAC-BTD-BJ	
	Nb. tests	Temps	Nb. tests	Temps
SCEN#01	1 857 660	640	1 855 040	800
SCEN#02	427 104	130	427 306	160
SCEN#03	947 199	310	930 909	420
SCEN#04	246 034	90	246 013	120
SCEN#05	9 220 866	14 310	1 190 682	230
SCEN#06	691 367	100	691 367	100
SCEN#07	1 123 856	130	1 123 856	130
SCEN#08	2 346 455	270	2 346 455	260
SCEN#09	84	10	84	10
SCEN#10	84	10	84	10
SCEN#11	22 520 823	25 630	22 513 770	25 350

TAB. 2.8 : [Instances du monde réel] Nombre de tests de contraintes et temps de résolution (en millisecondes) de MAC et de MAC-BTD-BJ pour quelques instances de l'archive FullRLFAP.

2.3.6 Résumé

Dans cette section, nous avons présenté des résultats expérimentaux sur trois types de benchmarks CSPs :

- les CSPs aléatoires classiques,
- les CSPs aléatoires structurés,
- les instances du monde réel.

Pour la première classe, BTD, c'est-à-dire FC-BTD ou MAC-BTD, obtient des résultats similaires à ceux de FC ou MAC. Par conséquent, rechercher à exploiter la structure, lorsque celle-ci ne recèle pas a priori de propriétés particulières, ne pénalise pas l'efficacité de la méthode BTD. Pour les problèmes aléatoires structurés, nous avons observé une amélioration significative en utilisant FC-BTD (respectivement MAC-BTD) par rapport à FC (resp. MAC). Nous avons aussi constaté que FC-CBJ développe autant de nœuds que FC-BTD, mais FC-BTD s'avère plus rapide. De même, bien qu'il développe moins de nœuds, FC-NR est plus lent que FC-BTD. Enfin, sur les problèmes du monde réel, BTD obtient soit des résultats meilleurs que les algorithmes classiques, soit des résultats comparables.

Pour ces différents types de problèmes, nous avons noté que le Tree-Clustering ne peut être employé en pratique pour deux raisons. D'une part, sa complexité pratique en temps est trop élevée. D'autre part, l'espace mémoire requis est réellement prohibitif, ce qui rend la méthode

inexploitable, alors que ce critère ne constitue pas un problème pour BTD.

Pour conclure, BTD semble être une approche permettant d'exploiter les propriétés structurelles des CSPs, sans avoir les défauts relatifs à la complexité en espace des autres méthodes de décomposition structurelle.

2.4 Discussion

Il semblerait que BTD constitue une approche permettant d'exploiter les caractéristiques structurelles de certains CSPs, sans pour autant hériter des désagréments inhérents aux méthodes fondées sur la décomposition en terme de complexité en espace. Pour nous en assurer sur le plan théorique, mais aussi pour vérifier l'originalité de BTD, nous avons recherché parmi les travaux les plus proches les éléments qui permettent de les distinguer.

Ces travaux peuvent être classés selon trois orientations principales :

- Le Backtracking exploitant les goods et les nogoods au sens de Bayardo et Miranker [BM94, BM96].
- Le Tree-Clustering [DP89] et ses améliorations théoriques [GLS00].
- Les approches hybrides recherchant un compromis entre le Tree-Clustering (ou la consistance adaptative [DP89]) et le Backtracking [DF01, Lar00].

Comme indiqué dans la présentation de BTD, les travaux les plus proches sont ceux de Bayardo et Miranker [BM94, BM96]. Notons que notre approche peut être considérée comme une généralisation naturelle de [BM94] puisque leur étude est limitée aux CSPs binaires acycliques (forêts). Par rapport à [BM96], alors que l'exploitation des goods et des nogoods est similaire à la nôtre, nos notions de goods et de nogoods sont formellement différentes. Dans [BM96], un good (ou un nogood) est défini par rapport à une variable x_i et à un ordre sur les sommets. Un good (ou un nogood) est une affectation d'un ensemble de variables qui précèdent x_i dans l'ordre et qui sont connectées à au moins une variable appartenant à l'ensemble des descendants de x_i dans la décomposition arborescente. Cette définition est alors formellement différente de la nôtre. Par exemple, si nous considérons un graphe de contraintes triangulé, et $x_i \in \mathcal{C}_j$, la dernière variable dans $\mathcal{C}_j$, alors un good (ou un nogood) sera une affectation de $\mathcal{C}_j \backslash \{x_i\}$. Ainsi, l'espace requis pour Learning-Tree-Solve (l'algorithme de [BM96]) est alors $O(n.d^{w^+ + 1})$ ($w^+ + 1$ étant la taille du plus grand $\mathcal{C}_j$) alors que l'espace requis pour BTD est limité à $O(n.d^s)$ où s est la taille du plus grand séparateur. La complexité en temps de Learning-Tree-Solve est $O(exp(w^+ + 1))$ comme BTD.

Par ailleurs, l'intérêt pratique de Learning-Tree-Solve n'est pas présenté dans [BM96]. De plus, dans [BP00], Bayardo et Pehoushek rappellent les avantages supposés de l'exploitation des nogoods pour le test de consistance, tout en évoquant la difficulté pour fournir une implémentation efficace des goods au sens de [BM96], implémentation qui n'a d'ailleurs pas été présentée ni dans [BM96] ni dans [BP00].

Le travail de Baget et Tognetti [BT01] peut être considéré comme une approche similaire à la nôtre. En effet, dans leur méthode, les clusters sont définis par les composantes biconnexes, et les goods et nogoods - les auteurs n'utilisent pas ces expressions - sont limités à des affectations de variables, celles qui séparent les composantes biconnexes. La complexité en temps de leur méthode est alors $O(n.d^k)$ avec k la taille maximum des composantes biconnexes. Dans ce cas, on a $w^+ + 1 \leq k$. Si nous considérons le graphe de contraintes de la figure 1.5, nous avons deux composantes biconnexes, $\{E, F, G\}$ et $\{A, B, C, D, H, I, J, K, L, M, N, O\}$, et donc, $k = 12$ alors que $w^+ = 3$. Néanmoins, Baget et Tognetti indiquent différents moyens pour améliorer leur approche en exploitant une généralisation aux composantes k-connexes. Notons enfin qu'aucun résultat expérimental n'est présenté dans [BT01].

BTD est principalement basé sur la décomposition arborescente. Aussi, les travaux proches du Tree-Clustering et de ses améliorations sont de fait pertinents dans cette comparaison. Notamment,

dans [GLS00], une amélioration du Tree-Clustering est présentée de même qu'une comparaison théorique entre les principales méthodes de décomposition. Ces résultats peuvent indiquer des pistes pour des améliorations (théoriques) de BTD bien que leur incidence pratique demeure plus qu'hypothétique.

BTD peut être considéré comme une approche hybride réalisant un compromis entre l'efficacité pratique en temps et l'économie de ressources en espace. Dans [DF01], Dechter et El Fattah présentent un schéma réalisant un compromis temps-espace. Ce schéma permet de proposer une gamme d'algorithmes dont le tree-clustering et la méthode du coupe-cycle (linéaire en espace) seraient les deux extrêmes. Une autre idée séduisante dans ce travail est fournie par la possibilité de modifier la taille des séparateurs afin de minimiser l'espace. Dans les expérimentations que nous avons réalisées, nous avons d'ailleurs exploité cette idée afin de minimiser la taille des goods et des nogoods. Finalement, notons que leurs résultats expérimentaux sont limités à l'unique évaluation des paramètres structurels (w^+ et s) sur des instances structurées du monde réel (circuits combinatoires), et aucun résultat sur l'efficacité de la résolution effective des instances n'est présenté.

Enfin, dans [Lar00], Larrosa propose une méthode hybride basée sur la Consistance Adaptative (Adaptive Consistency [DP89]) et sur le Backtracking (ou FC ou MAC). La Consistance Adaptative (AdCons) s'appuie sur un schéma général d'élimination de variables qui opère en remplaçant des ensembles de variables par de nouvelles contraintes qui "résument" les effets des variables éliminées. AdCons possède les même bornes de complexité en temps et en espace que le Tree-Clustering. Aussi, l'aspect exponentiel de la complexité en espace constitue-t-il une limitation considérable pour l'intérêt pratique de l'algorithme. L'idée de Larrosa consiste à limiter la taille des nouvelles contraintes produites par AdCons à un paramètre k. Si des contraintes d'arité supérieure doivent être produites, alors cette production sera remplacée par une phase d'énumération (BT, FC, MAC, ...). Cette approche hybride permet de borner la taille de l'espace à $O(d^k)$ mais au détriment de la complexité en temps qui se trouve alors majorée par $O(exp(z(k)+k+1))$. Ici $z(k)$ est un paramètre structurel induit par k et par la largeur du graphe de contraintes ; il vérifie $z(k) + k < n$. Notons que dans le cas des graphes de contraintes ayant une faible densité et en limitant la valeur de k à 2, l'auteur obtient des résultats intéressants sur des CSPs aléatoires classiques.

2.5 Conclusion

Notre objectif dans ce chapitre était de proposer une nouvelle méthode de résolution de CSPs. Cette nouvelle méthode, nommée BTD (pour Bactracking sur Tree-Decomposition), est une méthode énumérative qui exploite la notion de décomposition arborescente du graphe de contraintes. Grâce à cette notion, elle produit et mémorise des goods et des nogoods structurels. Ces goods et nogoods permettent à BTD d'éviter certaines redondances dans la recherche. Il en résulte que BTD obtient alors des complexités en temps et en espace similaires à celle du Tree-Clustering. De plus, nous avons également montré qu'en théorie, BTD développe moins de nœuds que BT et TC lorsque BT et TC utilisent le même ordre d'instanciation des variables que BTD.

Les expériences réalisées montrent l'intérêt pratique de la méthode, à savoir que :

- sur les problèmes aléatoires classiques, BTD est aussi efficace que les algorithmes énumératifs classiques, voire même meilleur, dans certains cas,
- sur les problèmes aléatoires structurés, BTD affiche un gain significatif grâce à l'exploitation des goods et des nogoods,
- sur les instances du monde réel, BTD obtient soit des résultats meilleurs que les algorithmes énumératifs classiques, soit des résultats comparables.

- concernant l'espace-mémoire requis, BTD peut être utilisé en pratique, tandis que l'algorithme Tree-Clustering se révèle trop gourmand en mémoire.

La méthode BTD bénéficie donc des avantages des méthodes énumératives (efficacité pratique) et des méthodes structurelles (borne de complexité en temps).

Parmi les extensions possibles de cette méthode, la généralisation aux CSPs n-aires ne devrait pas poser de difficulté, car elle s'obtient immédiatement par construction. Une seconde extension plus prometteuse concerne les problèmes d'optimisation. Le chapitre suivant est d'ailleurs consacré à la généralisation de la méthode BTD au cadre des CSPs valués. Enfin, la comparaison théorique entre BTD et BT (respectivement FC-BTD et FC ou MAC-BTD et MAC) doit être étendue afin de prendre en compte des ordres d'instanciation des variables différents.

Chapitre 3

Extension de BTD aux CSPs valués

De nombreux problèmes réels peuvent être représentés grâce au formalisme CSP. Dans ce formalisme, les contraintes expriment l'autorisation ou l'interdiction d'une combinaison de valeurs (on qualifie ces contraintes de "*dures*"). Cependant, pour certains problèmes réels, certaines contraintes (dites "*molles*") ne traduisent, dans la réalité, qu'une préférence, une possibilité, ... Représenter ces contraintes par des contraintes dures rend souvent les CSPs correspondants inconsistants. Aussi, afin de pouvoir exprimer de telles contraintes, plusieurs extensions du cadre CSP ont été proposées. Pour notre part, nous nous intéressons au cadre des CSPs valués (VCSPs [SFV95, SFV97]) et nous proposons, dans ce chapitre, une extension de notre algorithme BTD au cadre VCSP.

3.1 Les CSPs valués

3.1.1 Définitions et propriétés

Le cadre VCSP permet d'exprimer différents types de problèmes d'optimisation. Nous limitons notre étude au problème de minimisation (le problème de maximisation est similaire). À la différence du cadre CSP, les contraintes dans le cadre VCSP peuvent être soit dures soit molles. Résoudre un problème VCSP revient alors à rechercher une affectation qui optimise une fonction portant sur la satisfaction des contraintes. Autrement dit, une solution du problème est une affectation qui peut éventuellement violer certaines contraintes et dont l'importance des violations est minimale suivant un critère et un ordre donnés. Pour quantifier l'importance d'une violation, une valuation est associée à chaque contrainte. Lorsque plusieurs contraintes sont violées simultanément, les valuations correspondantes doivent être agrégées pour déterminer l'importance de la violation de l'ensemble de ces contraintes. Dans ce but, on se dote d'une structure de valuation :

Définition 3.1 (structure de valuation [SFV95, SFV97]) *Une structure de valuation est un triplet $(E, \preceq, \oplus)$ avec un ensemble E de valuations totalement ordonné par $\preceq$, muni d'un élément minimum (noté $\bot$), d'un élément maximum (noté $\top$) et d'une loi de composition interne (notée $\oplus$) commutative, associative, monotone et telle que $\bot$ soit un élément neutre pour $\oplus$ et $\top$ un élément absorbant.*

Les valuations étant associées aux contraintes[1], $\bot$ caractérise une absence de violation et $\top$ une violation inacceptable. Quant à la loi $\oplus$, elle permet de calculer la valuation correspondant à

[1]Dans certains cas, il peut être souhaitable d'associer une valuation à chaque tuple. Dans le modèle employé, il suffit d'employer une contrainte par tuple interdit pour atteindre ce but.

la violation simultanée de plusieurs contraintes. Notons que, dans certains cas, elle peut posséder d'autres propriétés comme l'idempotence ou la stricte monotonie. À partir de cette structure de valuation, on peut donner la définition formelle d'un CSP valué :

Définition 3.2 (CSP valué [SFV95, SFV97]) *Un CSP valué (VCSP) est un CSP classique $\mathcal{P} = (X, D, C, R)$ doté d'une structure de valuation $S = (E, \preceq, \oplus)$ et d'une application ϕ de C dans E, qui associe une valuation à chaque contrainte du CSP. On le note comme un sextuplet (X, D, C, R, S, ϕ).*
*Le VCSP est dit **binaire** si chaque contrainte de C implique au plus deux variables.*

La valuation d'une instanciation complète $\mathcal{A}$ des variables de X correspond à la combinaison (ou agrégation) des valuations des contraintes violées par $\mathcal{A}$:

Définition 3.3 (valuation d'une affectation [SFV95, SFV97])
Soient un VCSP $\mathcal{P} = (X, D, C, R, S, \phi)$ et une instanciation $\mathcal{A}$ sur X. La valuation de $\mathcal{A}$ dans $\mathcal{P}$ se définit par :

$$\mathcal{V}_{\mathcal{P}}(\mathcal{A}) = \bigoplus_{c \in C | \mathcal{A} \text{ viole } c} \phi(c)$$

Étant donnée une instance $\mathcal{P}$, le problème VCSP consiste donc à trouver une affectation de toutes les variables de $\mathcal{P}$ qui soit de valuation minimum au sens de $\preceq$. Cette valuation optimale est appelée **valuation du VCSP**. Par la suite, nous notons $\alpha^*_{\mathcal{P}}$ cette valuation. Déterminer la valuation d'un VCSP est un problème NP-difficile.
Cette notion de valuation d'une affectation complète peut être étendue à des instanciations partielles :

Définition 3.4 (valuation locale [SFV95, SFV97]) *Soient un VCSP $\mathcal{P} = (X, D, C, R, S, \phi)$ et une instanciation $\mathcal{A}$ sur $Y \subset X$. La valuation locale de $\mathcal{A}$ dans $\mathcal{P}$ se définit par :*

$$v_{\mathcal{P}}(\mathcal{A}) = \bigoplus_{\substack{c \in C | c \subseteq Y \\ \text{et } \mathcal{A} \text{ viole } c}} \phi(c)$$

La propriété suivante établit le lien existant entre la valuation d'une affectation complète et la valuation locale :

Propriété 3.5 ([SFV95, SFV97]) *Soit un VCSP $\mathcal{P} = (X, D, C, R, S, \phi)$. Soient $\mathcal{A}$ une instanciation complète et $\mathcal{B} \subseteq \mathcal{A}$. On a : $v_{\mathcal{P}}(\mathcal{B}) \preceq v_{\mathcal{P}}(\mathcal{A}) = \mathcal{V}_{\mathcal{P}}(\mathcal{A})$.*

La notion de valuation locale permet donc d'obtenir un minorant de la valuation globale. L'intérêt majeur de la valuation locale réside dans la possibilité de la calculer de façon incrémentale.

Exemple 3.6 *Considérons le VCSP $\mathcal{P} = (X, D, C, R, S, \phi)$ avec (X,D,C,R) le CSP de l'exemple 1.4 (page 12). Nous considérons comme structure de valuation $(\overline{\mathbb{N}}, \leq, +)$. Les éléments minimum et maximum sont respectivement 0 et $+\infty$. Toutes les contraintes ont une valuation égale à 1.*
Soient $\mathcal{A}_1 = \{x_1 \leftarrow b, x_2 \leftarrow c, x_3 \leftarrow c, x_4 \leftarrow a\}$ et $\mathcal{A}_2 = \{x_1 \leftarrow a, x_2 \leftarrow c, x_3 \leftarrow b, x_4 \leftarrow a, x_5 \leftarrow b, x_6 \leftarrow a, x_7 \leftarrow c\}$ deux affectations. On a alors $v_{\mathcal{P}}(\mathcal{A}_1) = 2$ et $v_{\mathcal{P}}(\mathcal{A}_2) = \mathcal{V}_{\mathcal{P}}(\mathcal{A}_2) = 1$. La valuation optimale $\alpha^*_{\mathcal{P}}$ du problème est 1 ($\mathcal{A}_2$ étant une solution optimale).

Si nous avons choisi d'employer le formalisme VCSP pour exprimer des problèmes d'optimisation, c'est parce qu'il présente l'avantage de généraliser les CSPs classiques, mais aussi, plusieurs de leurs extensions, parmi lesquelles :

- les CSPs possibilistes [Sch92] et flous [RHZ76, DFP93, Rut94],
- les CSPs à pénalités (Max-CSP [FW92]),
- les CSPs probabilistes [FL93, FLMCS95],
- les CSPs lexicographiques [FLS93].

Le tableau 3.1 détaille la structure de valuation pour les CSPs classiques et quelques unes de leurs extensions.
Nous aurions également pu choisir le formalisme SCSP (semiring based CSP [BMR95]) qui est proche du formalisme VCSP. Leur principale différence réside dans l'objet mathématique employé pour représenter la structure de valuation (voir [BFM$^+$96] pour plus de détails sur une comparaison entre les VCSPs et SCSPs).

CSP	E	$\oplus$	$\bot$	$\top$	$\prec$	Propriétés
Classique	$\{t, f\}$	$\wedge = \max$	t	f	$t \prec f$	idempotence + monotonie stricte
Possibiliste	$[0, 1]$	max	0	1	$<$	idempotence
Max-CSP	$\overline{\mathbb{N}}$	$+$	0	$+\infty$	$<$	monotonie stricte
Probabiliste	$[0, 1]$	$x + y - xy$	0	1	$<$	monotonie stricte
Lexico.	$[0, 1]^* \cup \{\top\}$	$\cup$	$\emptyset$	$\top$	lex	monotonie stricte

Tab. 3.1 : Structure de valuation pour les CSPs classiques et quelques unes de leurs extensions.

3.1.2 Algorithmes de résolution pour les VCSPs

Les algorithmes de résolution du problème VCSP sont pour la plupart des adaptations d'algorithmes de résolution du problème CSP. L'objectif à présent est de trouver une solution optimale, et non plus la première solution comme c'est généralement le cas dans le cadre CSP. Cette recherche de l'optimalité s'accompagne donc en général d'un accroissement de la difficulté à résoudre les problèmes. Par exemple, pour des instances du monde réel comme celles du CELAR, la recherche d'une solution est souvent facile (voire même triviale pour certaines instances), par contre, rechercher la meilleure solution se révèle être une tâche nettement plus complexe.

La principale modification consiste à exploiter un algorithme de type branch and bound (séparation / évaluation) en profondeur d'abord au lieu d'un algorithme de type backtracking. Un algorithme de type branch and bound exploite généralement deux bornes : un minorant et un majorant. Le minorant est une sous-estimation de la valuation d'une affectation complète contenant l'instanciation courante. Quant au majorant, il correspond à une surestimation de la valuation optimale. On utilise souvent comme majorant la valuation de la meilleure solution connue. Les algorithmes de type branch and bound cherchent à étendre une affectation partielle :

- soit jusqu'à l'obtention d'une instanciation complète (i.e. d'une meilleure solution),
- soit jusqu'à ce que le minorant dépasse le majorant.

Dans les deux cas, on revient en arrière pour poursuivre la recherche en instanciant la variable la plus profonde avec la prochaine valeur de son domaine. Notons que le minorant et le majorant employés caractérisent l'algorithme. Aussi, la qualité de ces deux bornes s'avère prépondérante pour l'efficacité de l'algorithme. Plus grand sera le minorant et/ou plus petit sera le majorant, meilleur sera l'algorithme, puisque plus de branches seront élaguées. Parmi les algorithmes ainsi adaptés à partir du cadre CSP, on peut citer par exemple :

- Forward-Checking [FW92, SFV95, SFV97],
- Nogood Recording [DV96],

- Dynamic Backtracking [Dag97].

Les algorithmes exploitant un niveau de consistance supérieur à celui de Forward-Checking posent problème quand la loi $\oplus$ n'est pas idempotente. C'est le cas, par exemple, pour le maintien de la consistance d'arc. En effet, lors de l'application de tels algorithmes, la même contrainte peut être prise en compte plusieurs fois. Si tel est le cas et si la loi $\oplus$ n'est pas idempotente, on risque de surévaluer le minorant, ce qui fausse alors le résultat. Il apparaît donc nécessaire de partitionner l'ensemble des contraintes et de rechercher un minorant sur chacun des sous-ensembles obtenus. Ainsi, de nombreux travaux se sont attachés et s'attachent encore actuellement à définir différents niveaux et variantes de consistance d'arc. L'essentiel de ces travaux (parmi lesquels [LM96, Wal96, Sch98, LMS99, Sch00, PRB01, CS02, Lar02, Sch02]) concerne le problème Max-CSP.

Nous rappelons d'abord deux algorithmes issus du cadre CSP et reposant sur le branch and bound, puis trois approches radicalement différentes des méthodes énumératives classiques.

3.1.2.1 L'algorithme Branch and Bound

Pour l'algorithme Branch and Bound (noté BB), le minorant doit prendre en compte les contraintes violées par l'affectation courante. Il correspond donc tout simplement à la valuation locale $v_{\mathcal{P}}(\mathcal{A})$ de l'affectation courante $\mathcal{A}$. Ce minorant peut être calculé incrémentalement. Le minorant courant est, en effet, égal à l'agrégation du minorant au nœud parent et des valuations des contraintes violées par l'instanciation de la variable courante. Concernant le majorant, il s'agit simplement de la valuation de la meilleure solution connue.

Étant donnée une affectation $\mathcal{A}$, l'algorithme BB (décrit à la figure 3.1) calcule la valuation optimale du problème à résoudre sachant que :

- V est l'ensemble des variables non instanciées,
- α est la valuation de la meilleure solution connue,
- $l = v_{\mathcal{P}}(\mathcal{A})$.

Initialement, $\mathcal{A}$ est l'affectation vide, V est égal à X, α à $\top$ et l à $\bot$.

3.1.2.2 Forward-Checking valué

Le majorant de l'algorithme Forward-Checking valué est identique à celui de BB (i.e. la valuation de la meilleure solution connue). Par contre, son minorant diffère de celui de BB dans la mesure où il tient compte des variables non instanciées. En effet, étant donnée $\mathcal{A}$ l'affectation courante, il correspond à l'agrégation :

- des valuations des contraintes violées par $\mathcal{A}$ (c'est-à-dire $v_{\mathcal{P}}(\mathcal{A})$),
- pour chaque variable y non instanciée, des valuations des contraintes d'un des ensembles $C(y, v)$ avec $C(y, v)$ l'ensemble des contraintes violées par l'affectation de y avec la valeur v.

Pour chaque variable y non affectée, l'ensemble $C(y, v)$ choisi sera celui dont l'agrégation des valuations des contraintes est minimum.
Comme précédemment, le minorant peut être calculé incrémentalement. Dans ce but, une valuation $B(y, v)$ est associée à chaque valeur v d'une variable y non instanciée. $B(y, v)$ correspond à l'agrégation des valuations des contraintes de $C(y, v)$, c'est-à-dire des contraintes qui seront violées si on étend l'affectation courante avec y affectée à la valeur v. Le minorant est alors égal à l'agrégation de la valuation locale de l'affectation courante et d'une valuation $B(y, v)$ par variable y non instanciée, la valuation $B(y, v)$ choisie étant celle de valuation minimum. Initialement, les $B(y, v)$ ont pour valuation $\bot$. L'extension de l'affectation courante $\mathcal{A}$ en $\mathcal{A}' = \mathcal{A} \cup \{x \leftarrow a\}$ entraîne

$\mathsf{BB}(\mathcal{A}, V, \alpha, l)$
1. **If** $V = \emptyset$
2. **Then**
3. Mémoriser la solution $\mathcal{A}$
4. Retourner l
5. **Else**
6. Choisir $x \in V$
7. $d \leftarrow d_x$
8. **While** $d \neq \emptyset$ **and** $l \prec \alpha$ **Do**
9. Choisir a dans d
10. $d \leftarrow d \backslash \{a\}$
11. $L \leftarrow \{c = \{x, y\} \in C | y \notin V\}$
12. $l_a \leftarrow \bot$
13. **While** $L \neq \emptyset$ **and** $l \oplus l_a \prec \alpha$ **Do**
14. Choisir c dans L
15. $L \leftarrow L \backslash \{c\}$
16. **If** c viole $\mathcal{A} \cup \{x \leftarrow a\}$ **Then** $l_a \leftarrow l_a \oplus \ \phi(c)$
17. **EndWhile**
18. **If** $l \oplus l_a \prec \alpha$ **Then** $\alpha \leftarrow \min(\alpha, \mathsf{BB}(\mathcal{A} \cup \{x \leftarrow a\}, V \backslash \{x\}, \alpha, l \oplus l_a))$
19. **EndWhile**
20. Retourner α
21. **EndIf**

FIG. 3.1 : L'algorithme Branch and Bound pour les CSPs valués.

la mise à jour suivante des valuations $B(y, v)$:

$$B(y, v) \leftarrow B(y, v) \oplus \bigoplus_{\substack{c=\{x,y\} \in C | \\ \mathcal{A} \cup \{x \leftarrow a\} \ viole \ c}} \phi(c)$$

Quant à la valuation locale de l'affectation $\mathcal{A}'$, elle se calcule en combinant la valuation locale de $\mathcal{A}$ au nœud parent avec $B(x, a)$.

Étant donnée une affectation $\mathcal{A}$, l'algorithme Forward-Checking valué (noté FC-val et décrit à la figure 3.2) renvoie la valuation optimale du problème à résoudre sachant que :

- V est l'ensemble des variables non instanciées,
- α est la valuation de la meilleure solution connue,
- $l = v_{\mathcal{P}}(\mathcal{A})$,
- M est le minorant de l'affectation courante.

Initialement, $\mathcal{A}$ est l'affectation vide, V est égal à X, α à $\top$, l et M à $\bot$. La mise à jour des valuations $B(y, v)$ est effectuée par la fonction Mise-à-jour (pour laquelle x est la variable courante). Cette fonction calcule et renvoie le nouveau minorant. Le calcul des $B(y, v)$ nécessite de les sauvegarder avant la mise à jour puis de les restaurer quand on revient en arrière. Ce travail est réalisé respectivement par les fonctions Push-Domains et Pop-Domains.

3.1.2.3 L'algorithme des poupées russes

L'algorithme des poupées russes (RDS Russian Dolls Search [VLS96]) diffère fondamentalement des deux algorithmes précédents au niveau du calcul du minorant. En effet, pour les deux algorithmes précédents, les contraintes prises en compte dans le calcul du minorant sont au mieux

FC-val($\mathcal{A}, V, \alpha, l, M$)
1. **If** $V = \emptyset$
2. **Then**
3. Mémoriser la solution $\mathcal{A}$
4. Retourner l
5. **Else**
6. Choisir $x \in V$
7. $d \leftarrow d_x$
8. **While** $d \neq \emptyset$ **and** $M \prec \alpha$ **Do**
9. Choisir a dans d
10. $d \leftarrow d \backslash \{a\}$
11. Push-Domains
12. $M_a \leftarrow$ Mise-à-jour($\mathcal{A} \cup \{x \leftarrow a\}, V \backslash \{x\}, l \oplus B(x,a), x$)
13. **If** $M_a \prec \alpha$ **Then** $\alpha \leftarrow \min(\alpha$,FC-val($\mathcal{A} \cup \{x \leftarrow a\}, V \backslash \{x\}, \alpha, l \oplus B(x,a), M_a$))
14. Pop-Domains
15. **EndWhile**
16. Retourner α
17. **EndIf**

Mise-à-jour($\mathcal{A}, V, l, x$)
1. $L \leftarrow \{c = \{x, y\} \in C | y \in V\}$
2. **While** $L \neq \emptyset$ **Do**
3. Choisir $c = \{x, y\}$ dans L
4. $L \leftarrow L \backslash \{c\}$
5. **For each** $v \in d_y$ **Do**
6. **If** $\mathcal{A} \cup \{y \leftarrow v\}$ viole c **Then** $B(y,v) \leftarrow B(y,v) \oplus \phi(c)$
7. **EndWhile**
8. Retourner $l \oplus \bigoplus_{y \in V} \left(\min_{v \in d_y} B(y,v) \right)$

FIG. 3.2 : L'algorithme Forward-Checking valué pour les CSPs valués.

celles liant deux variables instanciées et celles liant une variable instanciée et une variable non instanciée. Pour l'algorithme RDS, le minorant exploite en plus les contraintes liant deux variables non affectées, grâce à une résolution du sous-problème formé par les variables non affectées.

Étant donné un ordre statique σ sur les variables, l'algorithme RDS effectue des résolutions successives de n problèmes emboîtés les uns dans les autres à la manière de poupées russes (avec n le nombre de variables). Le $i^{ème}$ problème porte sur les i dernières variables dans l'ordre σ. Les problèmes sont résolus avec une méthode de type branch and bound (comme BB ou FC-val). À l'issue de la résolution du $i^{ème}$ problème, l'affectation optimale de ce problème et sa valuation sont mémorisées. Ces informations sont ensuite exploitées pour résoudre le $i+1^{ème}$ problème. En effet, grâce à l'ordre statique, on peut combiner le minorant de BB ou de FC-val avec la valuation optimale d'un des problèmes déjà résolus. Cette combinaison définit le minorant de RDS, qui se révèle alors être un minorant meilleur que celui fourni par BB ou par FC-val.

Dans [MS01], une spécialisation de cet algorithme est proposée pour les problèmes Max-CSP et Weighted-CSP. Elle consiste à résoudre nd problèmes (si tous les domaines sont de taille d). Les problèmes sont en fait construits comme précédemment. Par contre, lorsqu'on souhaite résoudre le $i^{ème}$ problème, on va effectuer une résolution par valeur du domaine de la $n-i+1^{ème}$ variable

dans l'ordre statique. Cette version spécialisée de RDS permet d'obtenir un meilleur minorant que l'algorithme RDS classique.

Dans [MS00], une autre extension des poupées russes est proposée. Cette extension (appelée TRDS pour Tree-based RDS) vise à exploiter l'indépendance qui peut exister entre deux sous-problèmes dans le but de produire un meilleur minorant que celui de RDS. Par indépendance entre deux sous-problèmes, on entend que les sous-problèmes ne sont pas liés par une contrainte. Outre un minorant meilleur, cette approche permet également de réduire la taille des sous-problèmes. Par contre, le nombre de sous-problèmes reste le même (c'est-à-dire n). Chaque sous-problème peut alors être étiqueté par la variable de ce sous-problème qui n'appartient pas aux sous-problèmes qu'il contient. La méthode TRDS repose sur le concept d'arbre des sous-problèmes. Cet arbre a pour sommets les variables du problème. Il est construit récursivement en recherchant les points d'articulation. La racine r est un point d'articulation du graphe de contraintes s'il en existe un, une variable choisie arbitrairement sinon. On retire ensuite la variable r et toutes les contraintes impliquant r du graphe de contraintes et on recherche les composantes connexes. Chaque composante connexe correspond alors à un sous-problème. Chaque sous-problème possède une variable qui est un fils de r. Le sous-arbre enraciné en chaque fils de r est alors construit en appliquant récursivement ce procédé sur la composante connexe associée. Notons que l'existence de plusieurs composantes connexes implique que les sous-problèmes correspondants sont indépendants. Enfin, comme RDS, TRDS emploie un ordre statique sur les variables qui, dans le cas de TRDS, est fourni par un parcours en profondeur d'abord de l'arbre des sous-problèmes.

Expérimentalement, les deux premières méthodes présentent des résultats intéressants malgré l'exploitation d'un ordre statique et les multiples résolutions. Quant à la dernière, aucun résultat n'est fournie.

3.1.2.4 Pseudo-Tree Search

Dans [LMS02], une adaptation de l'algorithme Pseudo-Tree Search [FQ85], issu du cadre CSP, est proposée. L'algorithme Pseudo-Tree Search repose sur la notion d'arrangement en pseudo-arbre d'un graphe. L'arrangement en pseudo-arbre d'un graphe correspond à un arbre dont les sommets sont les sommets du graphe et tel que deux sommets voisins dans le graphe appartiennent à une même branche de l'arbre. L'algorithme Pseudo-Tree Search exploite un arrangement en pseudo-arbre du graphe de contraintes pour guider une recherche énumérative. Les variables sont en effet visitées suivant un parcours en profondeur d'abord du pseudo-arbre. Employer un tel arrangement présente l'avantage que la validité de l'affectation d'une variable ne dépend que des variables qui apparaissent sur le chemin allant de cette variable à la racine du pseudo-arbre. Autrement dit, étant donnée une variable x, dès que x est instanciée, les sous-arbres enracinés en chacun de ses fils correspondent à des sous-problèmes indépendants. Formellement, chaque sous-problème est formé du sous-arbre correspondant et de l'affectation des variables présentes sur le chemin allant de x à la racine du pseudo-arbre. Il en résulte que la complexité en temps de cette approche est en $O(d^h)$ où h est la hauteur du pseudo-arbre, pour une complexité en espace linéaire en la taille du problème. Par rapport aux méthodes de décomposition classiques, cette approche constitue un compromis entre le temps et l'espace requis dans la mesure où $h \leq (w^* + 1)(\log(n) + 1)$ [BM95] (w^* étant la tree-width du graphe de contraintes).

L'adaptation proposée dans [LMS02] est basée sur un algorithme de type branch and bound. Elle exploite l'indépendance des sous-problèmes. Étant donnée une variable x, dès que x est instanciée, la valuation optimale du problème $\mathcal{P}_x$ enraciné en x peut être calculée en combinant les valuations optimales des sous-problèmes $\mathcal{P}_y$ enracinés en un fils y de x. Toutefois, il faut prendre garde de ne pas compter plus d'une fois la valuation résultant de l'affectation des variables présentes sur

le chemin allant de x à la racine du pseudo-arbre, cette affectation appartenant à chaque sous-problème.

De plus, avant de résoudre un sous-problème $\mathcal{P}_y$ avec un algorithme de type branch and bound, on peut produire une valeur initiale pour le majorant. Cette valeur est calculée à partir :

- du majorant de l'algorithme de type branch and bound employé pour résoudre $\mathcal{P}_x$,
- des valuations optimales des sous-problèmes déjà résolus,
- d'une estimation de la valuation optimale de chaque sous-problème non résolu (hormis $\mathcal{P}_y$).

Cependant, si l'estimation de la valuation optimale de chaque sous-problème non résolu (hormis $\mathcal{P}_y$) est de mauvaise qualité, cette valeur peut se révéler insuffisante pour garantir une bonne efficacité à la méthode. C'est pourquoi la valeur initiale du majorant pour la résolution de $\mathcal{P}_y$ est le minimum entre la valeur décrite ci-dessus et une borne supérieure de la valuation optimale de $\mathcal{P}_y$. Cette dernière borne est calculée en ne tenant compte que du sous-problème $\mathcal{P}_y$.

Notons enfin que cette méthode se marie fort bien avec les poupées russes, les sous-problèmes étant constitués par les sous-arbres du pseudo-arbre. Expérimentalement, cette combinaison de SRDS avec les pseudo-arbres obtient de bons résultats sur des instances aléatoires. Ces résultats s'avèrent même, en général, meilleurs que ceux obtenus par SRDS.

3.1.2.5 Résolution par programmation dynamique

Dans [Kos99], une méthode de programmation dynamique est proposée pour résoudre les CSPs partiels ([Fre89, FW92]). Elle exploite une décomposition arborescente du graphe de contraintes. Étant donnée une décomposition, cette méthode commence par résoudre les problèmes constitués par chacun des clusters feuilles. Durant chacune de ces résolutions, elle mémorise toutes les affectations possibles sur chacun des clusters feuilles. Lorsque tous les fils d'un cluster $\mathcal{C}$ ont été traités, on résout alors le cluster $\mathcal{C}$ en combinant les affectations mémorisées sur les fils de $\mathcal{C}$ et en instanciant les variables de $\mathcal{C}$. Comme pour les clusters feuilles, toutes les affectations sur $\mathcal{C}$ sont mémorisées. On procède ainsi jusqu'au cluster racine, pour lequel seule la solution optimale est mémorisée. Cette méthode a une complexité en temps en $O(nd^{3k})$, pour une complexité en espace en $O(nd^{k+1})$ avec k la largeur de la décomposition arborescente employée.

Pour améliorer l'efficacité pratique de la méthode, plusieurs prétraitements sont proposés (réduction du graphe de contraintes avant de calculer la décomposition arborescente, réduction de la taille des domaines, ...). En pratique, cette méthode a permis de résoudre certaines instances de l'archive FullRLFAP. Cependant, pour d'autres instances de cette archive, la quantité de mémoire requise s'est avérée trop importante pour mener le calcul à son terme.

Pour pallier ce problème, un autre algorithme est proposé. Cet algorithme procède en instanciant non plus des valeurs mais des ensembles de valeurs. En fait, on définit un nouveau problème pour lequel les valeurs d'un domaine correspondent à une partition du domaine initial. Ce nouveau problème est alors résolu grâce à la méthode de programmation dynamique. À partir de la solution optimale trouvée, on partitionne à nouveau le domaine d'une des variables, ce qui permet de définir un nouveau problème à résoudre avec la méthode de programmation dynamique. On réitère le procédé jusqu'à l'obtention de la solution optimale.
En pratique, cet algorithme obtient de meilleurs résultats sur les instances non traitées par l'approche par programmation dynamique. Toutefois, comme précédemment certains problèmes ne peuvent être résolus à cause de la quantité de mémoire requise.

3.1.2.6 Résumé

Pour résoudre des instances VCSPs, nous disposons de différentes méthodes. L'approche purement énumérative, héritée généralement du problème CSP, semble limitée au niveau des perfor-

mances. Les meilleurs méthodes connues pour la résolution de VCSPs sont de loin les méthodes qui décomposent le problème en plusieurs sous-problèmes et qui mémorisent des informations sur la résolution de chaque sous-problème. L'algorithme RDS et ses extensions en sont des exemples, tout comme l'approche par programmation dynamique de Koster. On peut aussi considérer que la méthode BTD, présentée dans le chapitre 2 dans le cadre des CSPs, appartient à ce type d'approche. Aussi, il semble naturel d'étendre cette méthode au cadre des CSPs valués.

3.2 L'algorithme BTD pour les CSPs valués

L'exploitation de l'algorithme BTD pour résoudre des CSPs valués nécessite d'adapter certaines notions introduites dans le chapitre 2. Étant donnée une décomposition arborescente (ou une approximation d'une décomposition arborescente), les notions de numérotation et d'ordre compatibles restent valables. Il en est de même du théorème 2.3 (page 40). Par contre, nous devons adapter les notions de good et de nogood structurels afin de prendre en compte la notion de valuation.

3.2.1 Définitions et propriétés

Dans la suite de ce chapitre, nous noterons $\mathcal{P} = (X, D, C, R, S, \phi)$ l'instance VCSP à résoudre et nous exploiterons une décomposition arborescente $(\mathcal{C}, \mathcal{T})$ (ou une approximation d'une décomposition arborescente) du graphe de contraintes (X, C) associé à $\mathcal{P}$. Nous supposerons que les éléments de $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ sont indicés grâce à une numérotation compatible.

Lorsqu'on souhaite mémoriser et exploiter des nogoods, un point crucial dans le calcul de la valuation d'une affectation est de ne pas tenir compte plusieurs fois de la même contrainte. En effet, comptabiliser une contrainte plusieurs fois peut entraîner une surévaluation de la valuation de l'affectation si la loi $\oplus$ n'est pas idempotente. Pour éviter un tel problème, nous allons partitionner l'ensemble C des contraintes en exploitant les propriétés d'une décomposition arborescente.

Définition 3.7 *Soit $\mathcal{C}_i$ un cluster. L'ensemble $E_{\mathcal{P},\mathcal{C}_i}$ des contraintes propres au cluster $\mathcal{C}_i$ est défini par $E_{\mathcal{P},\mathcal{C}_i} = \{c \in C | c \subseteq \mathcal{C}_i$ et $c \not\subseteq \mathcal{C}_{p(i)}\}$ avec $\mathcal{C}_{p(i)}$ le cluster père de $\mathcal{C}_i$.*

L'ensemble $E_{\mathcal{P},\mathcal{C}_i}$ contient donc les contraintes de la forme $c = \{x, y\}$ avec x et y deux variables de $\mathcal{C}_i$ telle que x et/ou y n'appartiennent pas à $\mathcal{C}_{p(i)}$ le cluster père de $\mathcal{C}_i$.

Exemple 3.8 *Reprenons le VCSP $\mathcal{P}$ de l'exemple 3.6. Considérons la décomposition arborescente $(\mathcal{C}, \mathcal{T})$ présentée à la figure 3.3.*
Nous avons $E_{\mathcal{P},\mathcal{C}_1} = \{c_{12}, c_{13}, c_{23}\}$, $E_{\mathcal{P},\mathcal{C}_2} = \{c_{24}\}$, $E_{\mathcal{P},\mathcal{C}_3} = \{c_{15}, c_{35}\}$ et $E_{\mathcal{P},\mathcal{C}_4} = \{c_{56}, c_{57}, c_{67}\}$.

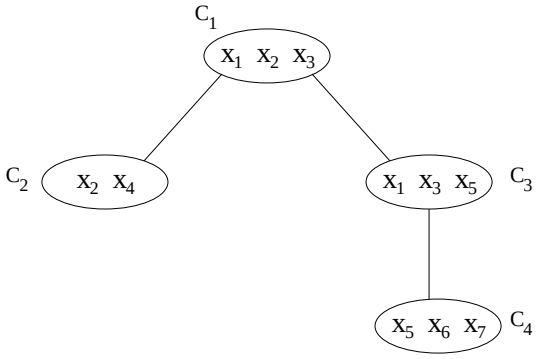

FIG. 3.3 : Une décomposition arborescente associée au VCSP $\mathcal{P}$ de l'exemple 3.6.

Propriété 3.9 *Les $(E_{\mathcal{P},\mathcal{C}_i})_i$ forment une partition de C.*

Preuve :
Montrons que $\bigcup_{\mathcal{C}_i \subseteq X} E_{\mathcal{P},\mathcal{C}_i} = C$.
Comme il est évident que $\bigcup_{\mathcal{C}_i \subseteq X} E_{\mathcal{P},\mathcal{C}_i} \subset C$, il reste à démontrer que $\bigcup_{\mathcal{C}_i \subseteq X} E_{\mathcal{P},\mathcal{C}_i} \supset C$.
Soit $c \in C$. D'après la définition d'une décomposition arborescente (définition 1.26 page 25), il existe au moins un cluster $\mathcal{C}_i$ tel que $c \subseteq \mathcal{C}_i$. En particulier, on a nécessairement $c \subseteq \mathcal{C}_k$ où $k = \min\{i | c \subseteq \mathcal{C}_i\}$ et $c \not\subseteq \mathcal{C}_{p(k)}$. Donc, $c \in E_{\mathcal{P},\mathcal{C}_k}$.
D'où $\bigcup_{\mathcal{C}_i \subseteq X} E_{\mathcal{P},\mathcal{C}_i} \supset C$
Donc $\bigcup_{\mathcal{C}_i \subseteq X} E_{\mathcal{P},\mathcal{C}_i} = C$

Montrons que $\forall \mathcal{C}_i, \mathcal{C}_j,\ E_{\mathcal{P},\mathcal{C}_i} \cap E_{\mathcal{P},\mathcal{C}_j} = \emptyset$.
Supposons qu'il existe $\mathcal{C}_i$ et $\mathcal{C}_j$ tels que $E_{\mathcal{P},\mathcal{C}_i} \cap E_{\mathcal{P},\mathcal{C}_j} \neq \emptyset$. Soit $c \in E_{\mathcal{P},\mathcal{C}_i} \cap E_{\mathcal{P},\mathcal{C}_j}$. Par définition, $c \subseteq \mathcal{C}_i \cap \mathcal{C}_j$.
D'après la définition 1.26 (page 25), il existe un chemin allant de $\mathcal{C}_i$ à $\mathcal{C}_j$ tel que si $\mathcal{C}_k$ est sur ce chemin, $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$. Sur un tel chemin, on rencontre nécessairement le père de $\mathcal{C}_i$ ou le père de $\mathcal{C}_j$. Donc $c \subseteq \mathcal{C}_{p(i)}$ ou $c \subseteq \mathcal{C}_{p(j)}$. D'où une contradiction puisque $c \in E_{\mathcal{P},\mathcal{C}_i}$ et $c \in E_{\mathcal{P},\mathcal{C}_j}$.
Donc $\forall i, j,\ E_{\mathcal{P},\mathcal{C}_i} \cap E_{\mathcal{P},\mathcal{C}_j} = \emptyset$

Les $(E_{\mathcal{P},\mathcal{C}_i})_i$ forment une partition de C. □

Nous pouvons alors définir la notion de VCSP induit :

Définition 3.10 (VCSP induit) *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$. Soit $\mathcal{A}$ une affectation sur $\mathcal{C}_i \cap \mathcal{C}_j$. $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j} = (X_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}}, D_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}}, C_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}}, R_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}}, S, \phi)$ est le VCSP induit par $\mathcal{A}$ sur la descendance de $\mathcal{C}_i$ enracinée en $\mathcal{C}_j$ avec :*

- $X_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = Desc(\mathcal{C}_j)$,
- $D_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = \{d_{x,\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = d_x | x \in Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)\} \cup \{d_{x,\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = \{\mathcal{A}[x]\} | x \in \mathcal{C}_i \cap \mathcal{C}_j\}$,
- $C_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = E_{\mathcal{P},\mathcal{C}_j} \cup \bigcup_{\mathcal{C}_d \text{ descendant de } \mathcal{C}_j} E_{\mathcal{P},\mathcal{C}_d}$,
- $R_{\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} = \{r_c \cap \prod_{x \in c} d_{x,\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}} | c \in C_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j} \text{ et } r_c \in R\}$.

Le VCSP induit $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ correspond au VCSP restreint au sous-arbre enraciné en $\mathcal{C}_j$ et dont les domaines des variables de $\mathcal{C}_i \cap \mathcal{C}_j$ sont restreints à la valeur correspondante dans $\mathcal{A}$. En effet, l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ est restreint aux variables figurant dans la descendance de $\mathcal{C}_i$ enracinée en $\mathcal{C}_j$. Quant à l'ensemble de contraintes de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$, il est réduit aux contraintes de C qui apparaissent exclusivement dans la descendance de $\mathcal{C}_i$ enracinée en $\mathcal{C}_j$.

Exemple 3.11 *Poursuivons l'exemple précédent. Soit l'affectation $\mathcal{A} = \{x_1 \leftarrow a, x_3 \leftarrow b\}$.*
Le VCSP $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3} = (X_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}}, D_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}}, C_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}}, R_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}}, S, \phi)$ est le VCSP induit par $\mathcal{A}$ sur la descendance de $\mathcal{C}_1$ enracinée en $\mathcal{C}_3$ avec :

- $X_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{x_1, x_3, x_5, x_6, x_7\}$,
- $D_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{d_{x_1,\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{a\},\ d_{x_3,\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{b\},\ d_{x_5,\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{a,b\},\ d_{x_6,\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{a,b,c\},\ d_{x_7,\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{a,b,c\}\}$,
- $C_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{c_{15}, c_{35}, c_{56}, c_{57}, c_{67}\}$,
- $R_{\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_3}} = \{r'_{15}, r'_{35}, r_{56}, r_{57}, r_{67}\}$ *où les relations r'_{15} et r'_{35} sont :*

r'_{15}	
x_1	x_5
a	b

r'_{35}	
x_3	x_5
b	b

À partir des ensembles $E_{\mathcal{P},\mathcal{C}_i}$, nous introduisons la notion de valuation locale à un cluster :

Définition 3.12 (valuation locale à un cluster) *Soient un cluster $\mathcal{C}_i$ et une instanciation $\mathcal{A}$ sur $Y \subset X$ avec $Y \cap \mathcal{C}_i \neq \emptyset$. La valuation locale $v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A})$ de $\mathcal{A}$ au cluster $\mathcal{C}_i$ dans $\mathcal{P}$ est la valuation locale de $\mathcal{A}$ restreinte aux contraintes de $E_{\mathcal{P},\mathcal{C}_i}$.*

$$v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) = \bigoplus_{\substack{c \in E_{\mathcal{P},\mathcal{C}_i} | c \subseteq Y \\ et\ \mathcal{A}\ viole\ c}} \phi(c)$$

La valuation locale à un cluster $\mathcal{C}_i$ ne tient compte que des contraintes propres au cluster $\mathcal{C}_i$, c'est-à-dire aux contraintes de $E_{\mathcal{P},\mathcal{C}_i}$. Comme pour la valuation locale, il est possible de calculer la valuation locale à un cluster de façon incrémentale.

Exemple 3.13 *Nous continuons l'exemple 3.8. Soit l'affectation $\mathcal{B} = \{x_1 \leftarrow a, x_2 \leftarrow c, x_3 \leftarrow c, x_5 \leftarrow b\}$. Nous avons $v_{\mathcal{P},\mathcal{C}_3}(\mathcal{B}) = 1$.*

Cette valuation possède plusieurs propriétés intéressantes. D'abord, son calcul ne dépend que des variables du cluster considéré.

Propriété 3.14 *Soient $\mathcal{C}_i$ un cluster et $\mathcal{A}$ une affectation sur $Y \subseteq X$ tel que $\mathcal{C}_i \subseteq Y$.*

$$v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}[\mathcal{C}_i])$$

Preuve : $v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) = \bigoplus_{\substack{c \in E_{\mathcal{P},\mathcal{C}_i} | c \subseteq Y \\ et\ \mathcal{A}\ viole\ c}} \phi(c) = \bigoplus_{\substack{c \in E_{\mathcal{P},\mathcal{C}_i} | c \subseteq Y \cap \mathcal{C}_i \\ et\ \mathcal{A}\ viole\ c}} \phi(c) = \bigoplus_{\substack{c \in E_{\mathcal{P},\mathcal{C}_i} | c \subseteq Y \cap \mathcal{C}_i \\ et\ \mathcal{A}[\mathcal{C}_i]\ viole\ c}} \phi(c) = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}[\mathcal{C}_i])$ □

Ensuite, l'agrégation des valuations locales à un cluster permet de calculer la valuation d'une affectation complète.

Propriété 3.15 *Soit une instanciation $\mathcal{A}$ sur X.*

$$\mathcal{V}_{\mathcal{P}}(\mathcal{A}) = \bigoplus_{\mathcal{C}_i \subseteq X} v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A})$$

Preuve : Puisque les $(E_{\mathcal{P},\mathcal{C}_i})_i$ forment une partition de C (d'après la propriété 3.9), chaque contrainte de C est comptabilisée une et une seule fois.
Par conséquent, $\mathcal{V}_{\mathcal{P}}(\mathcal{A}) = \bigoplus_{\mathcal{C}_i \subseteq X} v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A})$. □

Il découle de ces deux propriétés que le calcul de la valuation d'une affectation complète $\mathcal{A}$ peut s'effectuer en exploitant uniquement la valuation locale à chaque cluster $\mathcal{C}_i$ sur l'affectation restreinte $\mathcal{A}[\mathcal{C}_i]$.
Enfin, la propriété suivante assure que la valuation locale à un cluster $\mathcal{C}_j$ d'une affectation $\mathcal{B}$ reste la même que l'on considère le problème $\mathcal{P}$ ou un sous-problème induit de $\mathcal{P}$ contenant $\mathcal{C}_j$.

Propriété 3.16 *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters avec $\mathcal{C}_j$ un descendant de $\mathcal{C}_i$. Soient $\mathcal{A}$ une affectation sur $\mathcal{C}_i \cap \mathcal{C}_{p(i)}$ et $\mathcal{P}' = \mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}$. Si $\mathcal{B}$ est une affectation sur $\mathcal{C}_j$ telle que $\mathcal{B}[\mathcal{C}_j \cap \mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i \cap \mathcal{C}_{p(i)}]$, alors $v_{\mathcal{P},\mathcal{C}_j}(\mathcal{B}) = v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B})$.*

Preuve : comme $E_{\mathcal{P},\mathcal{C}_j} = E_{\mathcal{P}',\mathcal{C}_j}$ $v_{\mathcal{P},\mathcal{C}_j}(\mathcal{B}) = v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B})$. □

Nous pouvons désormais définir la notion de nogood valué structurel.

Définition 3.17 (nogood valué) *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$. Un* ***nogood valué*** *de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ est un couple $(\mathcal{A}, v)$ avec $\mathcal{A}$ une affectation sur $\mathcal{C}_i \cap \mathcal{C}_j$ et v la valuation optimale du VCSP $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$.*

Notons que cette notion de nogood valué englobe la notion de good et de nogood structurels du cadre classique. En effet, certaines contraintes pouvant être violées, la distinction entre good et nogood n'a plus de sens. Un good au sens classique correspond à un nogood valué de valuation $\bot$ et un nogood au sens classique à un nogood valué de valuation strictement supérieure à $\bot$.

Exemple 3.18 *Nous reprenons l'exemple 3.11.* $(\{x_1 \leftarrow a, x_3 \leftarrow b\}, \bot)$ *est un nogood de* $\mathcal{C}_1/\mathcal{C}_3$.

Étant donnée une affectation $\mathcal{A}$ sur $\mathcal{C}_i$, le théorème suivant exprime le fait que la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{B}[\mathcal{C}_i] = \mathcal{A}$ peut être calculée en exploitant la valuation optimale de chaque sous-problème enraciné en un fils $\mathcal{C}_f$ de $\mathcal{C}_i$ et induit par $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_f]$. Notons que la valuation optimale de chaque sous-problème est fournie soit par une résolution du sous-problème, soit par un nogood valué et qu'elle peut être calculée indépendamment de celles des autres sous-problèmes.

Théorème 3.19 *Soient $\mathcal{C}_i$ un cluster et $\mathcal{A}$ une instanciation sur $\mathcal{C}_i$. Soit $\mathcal{P}' = \mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}],\mathcal{C}_{p(i)}/\mathcal{C}_i}$.*

$$\min_{\substack{\mathcal{B}|X_{\mathcal{B}} = Desc(\mathcal{C}_i) \\ et\ \ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} v_{\mathcal{P}'}(\mathcal{B}) = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_f\ \ fils\ \ de\ \ \mathcal{C}_i} \alpha^*_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_f],\mathcal{C}_i/\mathcal{C}_f}}$$

Afin de démontrer ce théorème, nous avons besoin du lemme suivant :

Lemme 3.20 *Soient $\mathcal{C}_i$ un cluster et $\mathcal{A}$ une instanciation sur $\mathcal{C}_i$. Soit $\mathcal{P}' = \mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}],\mathcal{C}_{p(i)}/\mathcal{C}_i}$.*
Soit $\lambda = \min_{\substack{\mathcal{B}|X_{\mathcal{B}} = Desc(\mathcal{C}_i) \\ et\ \ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left(\bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right] \right)$.
Soit $\lambda' = \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \left(\min_{\substack{\mathcal{B}|X_{\mathcal{B}} = Desc(\mathcal{C}_j) \cup \mathcal{C}_i \\ et\ \ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right] \right)$.
Les quantités λ et λ' sont égales.

Preuve (lemme 3.20) : Pour tout $\mathcal{C}_j$ fils de $\mathcal{C}_i$, on pose $\lambda_{\mathcal{C}_j} = \min_{\substack{\mathcal{B}|X_{\mathcal{B}} = Desc(\mathcal{C}_j) \cup \mathcal{C}_i \\ et\ \ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right]$.
On a alors $\lambda' = \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \lambda_{\mathcal{C}_j}$.
Pour chaque $\mathcal{C}_j$ fils de $\mathcal{C}_i$, il existe une affectation $\mathcal{B}_{\mathcal{C}_j}$ sur $Desc(\mathcal{C}_j) \cup \mathcal{C}_i$ telle que $\mathcal{B}_{\mathcal{C}_j}[\mathcal{C}_i] = \mathcal{A}$ et qui vérifie $\lambda_{\mathcal{C}_j} = \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_{\mathcal{C}_j})$.

De même, il existe une affectation $\mathcal{B}_\lambda$ sur $Desc(\mathcal{C}_i)$ qui vérifie $\lambda = \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda) \right]$
et telle que $\mathcal{B}_\lambda[\mathcal{C}_i] = \mathcal{A}$.

Montrons que pour chaque fils $\mathcal{C}_j$ de $\mathcal{C}_i$, $\mathcal{B}_\lambda$ vérifie $\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_j) \cup \mathcal{C}_i]) = \lambda_{\mathcal{C}_j}$.

Supposons qu'il existe un fils $\mathcal{C}_s$ de $\mathcal{C}_i$ tel que $\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_s) \cup \mathcal{C}_i]) \neq \lambda_{\mathcal{C}_s}$.
Par définition de $\lambda_{\mathcal{C}_s}$, $\lambda_{\mathcal{C}_s} \prec \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_s) \cup \mathcal{C}_i]) = \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda)$

Soit $\mathcal{B}'$ une affectation sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{B}'[\mathcal{C}_i] = \mathcal{A}$ et $\forall \mathcal{C}_j \in Fils(\mathcal{C}_i), \mathcal{B}'[Desc(\mathcal{C}_j) \cup \mathcal{C}_i] = \mathcal{B}_{\mathcal{C}_j}$.
Une telle affectation existe car $\forall \mathcal{C}_j, \mathcal{C}_{j'} \in Fils(\mathcal{C}_i), Desc(\mathcal{C}_j) \cap Desc(\mathcal{C}_{j'}) \subseteq \mathcal{C}_i$.
De plus, elle vérifie entre autres $\lambda_{\mathcal{C}_s} = \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}'[Desc(\mathcal{C}_s) \cup \mathcal{C}_i])$.

$$\text{Donc, } \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}') \right] = \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \lambda_{\mathcal{C}_j} = \lambda'$$

$$\begin{aligned}
\lambda' &= \lambda_{\mathcal{C}_s} \oplus \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i) \setminus \{\mathcal{C}_s\}} \lambda_{\mathcal{C}_j} \\
&\prec \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_s) \cup \mathcal{C}_i]) \oplus \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i) \setminus \{\mathcal{C}_s\}} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_j) \cup \mathcal{C}_i]) \right] \\
&\prec \bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_s)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda) \oplus \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i) \setminus \{\mathcal{C}_s\}} \left[\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda) \right] = \lambda
\end{aligned}$$

D'où une contradiction avec la définition de λ.
Donc pour chaque $\mathcal{C}_j$ fils de $\mathcal{C}_i$, $\mathcal{B}_\lambda$ vérifie $\bigoplus_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}_\lambda[Desc(\mathcal{C}_j) \cup \mathcal{C}_i]) = \lambda_{\mathcal{C}_j}$.
Il s'ensuit l'égalité entre les quantités λ et λ'. □

Preuve (théorème 3.19) :
On pose $M = \min_{\substack{\mathcal{B} | X_\mathcal{B} = Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i] = \mathcal{A}}} v_{\mathcal{P}'}(\mathcal{B})$.

$$\begin{aligned}
M &\underset{propriété\ 3.5}{=} \min_{\substack{\mathcal{B} | X_\mathcal{B} = Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i] = \mathcal{A}}} \mathcal{V}_{\mathcal{P}'}(\mathcal{B}). \\
&\underset{propriété\ 3.15}{=} \min_{\substack{\mathcal{B} | X_\mathcal{B} = Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i] = \mathcal{A}}} \left(\bigoplus_{\mathcal{C}_j \subseteq Desc(\mathcal{C}_i)} v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B}) \right) \\
&= \min_{\substack{\mathcal{B} | X_\mathcal{B} = Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i] = \mathcal{A}}} \left(v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{B}) \oplus \bigoplus_{\substack{\mathcal{C}_j | j \neq i, \\ \mathcal{C}_j \subseteq Desc(\mathcal{C}_i)}} v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B}) \right) \\
&\underset{propriété\ 3.14}{=} \min_{\substack{\mathcal{B} | X_\mathcal{B} = Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i] = \mathcal{A}}} \left(v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{B}[\mathcal{C}_i]) \oplus \bigoplus_{\substack{\mathcal{C}_j | j \neq i, \\ \mathcal{C}_j \subseteq Desc(\mathcal{C}_i)}} v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B}) \right)
\end{aligned}$$

Or pour toute affectation $\mathcal{B}$ telle que $X_\mathcal{B} = Desc(\mathcal{C}_i)$ et $\mathcal{B}[\mathcal{C}_i] = \mathcal{A}$, $v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{B}[\mathcal{C}_i]) = v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A})$.
Comme $v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A})$ est une constante, on a :

$$
\begin{aligned}
M &= v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A}) \oplus \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left(\bigoplus_{\substack{\mathcal{C}_j|j\neq i, \\ \mathcal{C}_j\subseteq Desc(\mathcal{C}_i)}} v_{\mathcal{P}',\mathcal{C}_j}(\mathcal{B}) \right) \\
&= v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A}) \oplus \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left(\bigoplus_{\mathcal{C}_j\in Fils(\mathcal{C}_i)} \left[\bigoplus_{\mathcal{C}_k\subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right] \right) \\
&\underset{lemme\ 3.20}{=} v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j\in Fils(\mathcal{C}_i)} \left(\min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_j)\cup\mathcal{C}_i \\ et\ \mathcal{B}[\mathcal{C}_i]=\mathcal{A}}} \left[\bigoplus_{\mathcal{C}_k\subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right] \right) \\
&= v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j\in Fils(\mathcal{C}_i)} \left(\min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_j) \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_j]=\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j]}} \left[\bigoplus_{\mathcal{C}_k\subseteq Desc(\mathcal{C}_j)} v_{\mathcal{P}',\mathcal{C}_k}(\mathcal{B}) \right] \right) \\
&\underset{propriété\ 3.15}{=} v_{\mathcal{P}',\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j\in Fils(\mathcal{C}_i)} \left(\min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_j) \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_j]=\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j]}} \mathcal{V}_{\mathcal{P}'_{\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}}(\mathcal{B}) \right) \\
&\underset{propriété\ 3.16}{=} v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j\in Fils(\mathcal{C}_i)} \left(\min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_j) \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_j]=\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j]}} \mathcal{V}_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}}(\mathcal{B}) \right) \\
&= v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j\ fils\ de\ \mathcal{C}_i} \alpha^*_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i\cap\mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}} \ \square
\end{aligned}
$$

À partir du théorème 3.19, on déduit le corollaire suivant. Ce corollaire établit le lien existant entre la valuation optimale d'un problème enraciné en $\mathcal{C}_i$ et la valuation optimale de chaque sous-problème enraciné en un fils $\mathcal{C}_j$ de $\mathcal{C}_i$.

Corollaire 3.21 *Soient $\mathcal{C}_i$ un cluster et $\mathcal{A}$ une instanciation sur $\mathcal{C}_i \cap \mathcal{C}_{p(i)}$.*

$$
\alpha^*_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}} = \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=\mathcal{C}_i \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_{p(i)}]=\mathcal{A}}} \left(v_{\mathcal{P},\mathcal{C}_i}(\mathcal{B}) \oplus \bigoplus_{\mathcal{C}_j\ fils\ de\ \mathcal{C}_i} \alpha^*_{\mathcal{P}_{\mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}} \right)
$$

Preuve :

$$
\begin{aligned}
\alpha^*_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}} &= \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_{p(i)}]=\mathcal{A}}} \mathcal{V}_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \\
\alpha^*_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}} &\underset{propriété\ 3.5}{=} \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=Desc(\mathcal{C}_i) \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_{p(i)}]=\mathcal{A}}} v_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \\
&= \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=\mathcal{C}_i \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_{p(i)}]=\mathcal{A}}} \left(\min_{\substack{\mathcal{B}'|X_{\mathcal{B}'}=Desc(\mathcal{C}_i) \\ et\ \mathcal{B}'[\mathcal{C}_i]=\mathcal{B}[\mathcal{C}_i]}} v_{\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}') \right) \\
&\underset{théorème\ 3.19}{=} \min_{\substack{\mathcal{B}|X_{\mathcal{B}}=\mathcal{C}_i \\ et\ \mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_{p(i)}]=\mathcal{A}}} \left(v_{\mathcal{P},\mathcal{C}_i}(\mathcal{B}) \oplus \bigoplus_{\mathcal{C}_j\ fils\ de\ \mathcal{C}_i} \alpha^*_{\mathcal{P}_{\mathcal{B}[\mathcal{C}_i\cap\mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}} \right) \square
\end{aligned}
$$

3.2.2 L'algorithme BTD basé sur le Branch and Bound

3.2.2.1 L'algorithme BTD-val

L'adaptation (notée BTD-val) de la méthode BTD au cadre des CSPs valués est basée sur l'algorithme BB. À l'image de BTD, elle explore l'espace de recherche en utilisant un ordre compatible, qui débute avec les variables du cluster racine $\mathcal{C}_1$. À l'intérieur d'un cluster $\mathcal{C}_i$, elle procède classiquement comme le ferait BB. Cependant, le minorant ne tient compte que des contraintes propres au cluster $\mathcal{C}_i$ (i.e. les contraintes de $E_{\mathcal{P},\mathcal{C}_i}$) et le majorant est la valuation de la meilleure affectation $\mathcal{B}$ connue sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$ avec $\mathcal{A}$ l'affectation courante

sur $\mathcal{C}_i$. Lorsque toutes les variables de $\mathcal{C}_i$ sont affectées sans que le minorant ne dépasse le majorant, BTD-val poursuit la recherche avec le premier fils de $\mathcal{C}_i$, s'il en existe un. Plus généralement, dans le cas d'un fils $\mathcal{C}_j$ de $\mathcal{C}_i$, on teste si l'affectation $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ correspond à un nogood valué :

- Si c'est le cas, on agrège la valuation associée à ce nogood valué au minorant.
- Sinon, on doit étendre $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$ afin de déterminer la valuation de la meilleure affectation $\mathcal{B}$ telle que $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$. Une fois cette valuation v calculée, on l'agrège au minorant et on mémorise le nogood valué $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], v)$.

Si, après avoir traité le fils $\mathcal{C}_j$, le minorant ne dépasse pas le majorant, on continue la recherche avec le prochain fils de $\mathcal{C}_i$. Notons que, comme pour les goods dans BTD, l'exploitation des nogoods valués engendre un forward-jump. Lorsque tous les frères ont été examinés, si le minorant ne dépasse pas le majorant, alors, on a trouvé une solution meilleure pour le problème $\mathcal{P}_{\mathcal{A}[\mathcal{C}_{p(i)} \cap \mathcal{C}_i], \mathcal{C}_{p(i)}/\mathcal{C}_i}$. Enfin, si on rencontre un échec, alors il faut modifier l'instanciation courante sur $\mathcal{C}_i$.

L'algorithme BTD-val est décrit à la figure 3.4. Étant donnés une instanciation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, il recherche la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}], \mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \prec \alpha_{\mathcal{C}_i}$, sachant que

- $V_{\mathcal{C}_i}$ est l'ensemble des variables non instanciées de $\mathcal{C}_i$,
- $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$
- $l_{\mathcal{C}_i} = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \prec \alpha_{\mathcal{C}_i}$.

Si BTD-val trouve une telle affectation, il renvoie sa valuation, sinon il retourne une valuation supérieure ou égale à $\alpha_{\mathcal{C}_i}$. Le premier appel est effectué avec BTD-val$(\emptyset, \mathcal{C}_1, \mathcal{C}_1, \top, \bot)$.
Notons que $\mathcal{C}_i \cap \mathcal{C}_{p(i)} \subseteq \mathcal{C}_i \backslash V_{\mathcal{C}_i}$, ce qui implique que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$.

Théorème 3.22 *L'algorithme BTD-val est correct, complet et termine.*

Preuve : La preuve s'effectue par induction, en exploitant les propriétés des nogoods structurels valués. L'induction est réalisée sur le nombre de variables apparaissant dans la descendance de $\mathcal{C}_i$ excepté les variables déjà affectées de $\mathcal{C}_i$. Cet ensemble de variables est noté $VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) = V_{\mathcal{C}_i} \cup \bigcup_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} (Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j))$.
$VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$ correspond, en fait, à l'ensemble des variables à instancier pour déterminer la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$.

Pour montrer la correction de BTD-val, on doit prouver la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}), \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ définie ainsi :
"étant donnés $\alpha_{\mathcal{C}_i}$ la valuation de la meilleure affectation $\mathcal{B}'$ connue telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$ et $l_{\mathcal{C}_i} = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \prec \alpha_{\mathcal{C}_i}$, BTD-val $(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ renvoie :

- la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}], \mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \prec \alpha_{\mathcal{C}_i}$ si une telle affectation existe,
- une valuation supérieure ou égale à $\alpha_{\mathcal{C}_i}$, sinon".

Considérons $P(\mathcal{A}, \emptyset, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$:
Si $VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) = \emptyset$, alors $V_{\mathcal{C}_i} = \emptyset$ et $Fils(\mathcal{C}_i) = \emptyset$. Donc, $\mathcal{C}_i \backslash V_{\mathcal{C}_i} = \mathcal{C}_i$ et $l_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$. Comme BTD-val retourne $l_{\mathcal{C}_i}$, $P(\mathcal{C}_i, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}), \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ est vraie.

Pas d'induction : $P(\mathcal{A}, S, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ avec $S \neq \emptyset$. Supposons que $\forall S' \subset S, P(\mathcal{A}, S', \alpha_{\mathcal{C}_j}, l_{\mathcal{C}_j})$ soit vérifiée.

```
   BTD-val(A, C_i, V_{C_i}, α_{C_i}, l_{C_i})
1. If V_{C_i} = ∅
2. Then
3.     If Fils(C_i) = ∅ Then Retourner l_{C_i}
4.     Else
5.         F ← Fils(C_i)
6.         α ← l_{C_i}
7.         While F ≠ ∅ and α ≺ α_{C_i} Do
8.             Choisir C_j dans F
9.             F ← F\{C_j}
10.            If (A[C_j ∩ C_i], v) est un nogood de C_i/C_j dans N Then α ← α ⊕ v
11.            Else
12.                v ← BTD-val(A, C_j, C_j\(C_j ∩ C_i), ⊤, ⊥)
13.                α ← α ⊕ v
14.                Enregistrer le nogood (A[C_j ∩ C_i], v) de C_i/C_j dans N
15.            EndIf
16.        EndWhile
17.        Retourner α
18.    EndIf
19. Else
20.    Choisir x ∈ V_{C_i}
21.    d ← d_x
22.    While d ≠ ∅ and l_{C_i} ≺ α_{C_i} Do
23.        Choisir a dans d
24.        d ← d\{a}
25.        L ← {c = {x, y} ∈ E_{P,C_i} | y ∉ V_{C_i}}
26.        l_a ← ⊥
27.        While L ≠ ∅ and l_{C_i} ⊕ l_a ≺ α_{C_i} Do
28.            Choisir c dans L
29.            L ← L\{c}
30.            If c viole A ∪ {x ← a} Then l_a ← l_a ⊕ φ(c)
31.        EndWhile
32.        If l_{C_i} ⊕ l_a ≺ α_{C_i}
33.        Then α_{C_i} ← min(α_{C_i}, BTD-val(A ∪ {x ← a}, C_i, V_{C_i}\{x}, α_{C_i}, l_{C_i} ⊕ l_a))
34.        EndIf
35.    EndWhile
36.    Retourner α_{C_i}
37. EndIf
```

FIG. 3.4 : L'algorithme BTD-val pour les CSPs valués.

- Si $V_{\mathcal{C}_i} \neq \emptyset$:
  Durant la boucle **While** (lignes 22-36) l'assertion "pour chaque valeur v déjà examinée, $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$" est vérifiée.
  Lorsque BTD-val est appelé (ligne 34), l'affectation considérée est $\mathcal{A} \cup \{x \leftarrow v\}$ et on a $l_{\mathcal{C}_i} \oplus l_a \prec \alpha_{\mathcal{C}_i}$ et $VAR(\mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}) \subset VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$. D'après l'hypothèse d'induction, BTD-val$(\mathcal{A} \cup \{x \leftarrow v\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i} \oplus l_a)$ calcule la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash (V_{\mathcal{C}_i} \backslash \{x\})] = \mathcal{B}[\mathcal{C}_i \backslash (V_{\mathcal{C}_i} \backslash \{x\})]$. Soit α cette valuation. La mise à jour de $\alpha_{\mathcal{C}_i}$ (ligne 34) avec le minimum entre $\alpha_{\mathcal{C}_i}$ et α garantit que $\alpha_{\mathcal{C}_i}$ est toujours la valuation de la meilleure affectation $\mathcal{B}'$ connue telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$.

  À l'issue de la boucle (ligne 37), pour les valeurs de $d_x \backslash d$, $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$. Si $d = \emptyset$, comme BTD-val retourne $\alpha_{\mathcal{C}_i}$, $P(\mathcal{A}, S, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ est vérifiée. Sinon, pour les valeurs de d, la condition $l_{\mathcal{C}_i} \prec \alpha_{\mathcal{C}_i}$ est violée. Donc, toutes les affectations $\mathcal{A} \cup \{x \leftarrow v\}$ $(v \in d)$ ont une valuation qui dépasse ou égale $\alpha_{\mathcal{C}_i}$. Par conséquent, $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$. BTD-val retournant $\alpha_{\mathcal{C}_i}$, $P(\mathcal{A}, S, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ est vérifiée.
- Si $V_{\mathcal{C}_i} = \emptyset$:
  Durant la boucle **While** (lignes 7-16) l'assertion "$\alpha = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \oplus \bigoplus_{\mathcal{C}_j \in Fils(\mathcal{C}_i) \backslash F} \alpha^*_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}, \mathcal{C}_i/\mathcal{C}_j}$"
  (avec F est l'ensemble des fils de $\mathcal{C}_i$ déjà visités) est vérifiée.
  Initialement, l'assertion est vérifiée puisque $\alpha = l_{\mathcal{C}_i}$. Nous allons montrer que cette assertion est encore vraie à l'issue de la boucle.
  Soit $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ à examiner.
  + Si $(\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i], v)$ est un nogood valué de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, v est la valuation de l'instanciation optimale $\mathcal{B}$ sur $Desc(\mathcal{C}_j)$ telle que $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$. Donc, l'assertion est vérifiée.
  + Sinon, BTD-val est appelé avec $\mathcal{A}$ et $VAR(\mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i)) \subset VAR(\mathcal{C}_i, \emptyset)$. Par conséquent, d'après l'hypothèse d'induction, BTD-val$(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i), \top, \bot)$ retourne la valuation de l'instanciation optimale $\mathcal{B}$ sur $Desc(\mathcal{C}_j)$ telle que $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$. L'assertion est alors vérifiée.

  A l'issue de la boucle (ligne 17), l'assertion est vérifiée. Si $\alpha \prec \alpha_{\mathcal{C}_i}$, α, d'après le théorème 3.19, est la meilleure valuation possible pour une affectation $\mathcal{B}$ telle que $\mathcal{A}[\mathcal{C}_i] = \mathcal{B}[\mathcal{C}_i]$. Sinon, une telle affectation avec une valuation inférieure à $\alpha_{\mathcal{C}_i}$ n'existe pas. BTD-val retournant α, la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}), \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$ est satisfaite. Notons que la mémorisation des nogoods valués se justifie par leur définition.

Pour résumer, comme BTD-val satisfait la propriété $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}), \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i})$, et en particulier la propriété $P(\emptyset, VAR(\mathcal{C}_1, \mathcal{C}_1), \top, \bot)$ pour le premier appel, BTD-val est correct, complet et termine. □

3.2.2.2 Améliorations de BTD-val

Nous proposons, dans cette partie, deux améliorations de l'algorithme BTD-val concernant les minorants et majorants employés.
Généralement, le minorant et le majorant représentent deux données essentielles dans l'efficacité d'une méthode de résolution du problème VCSP. La plupart de ces méthodes emploie des minorants et des majorants portant sur l'intégralité du problème. Par exemple, le majorant global correspond souvent à la valuation de la meilleure solution connue. À l'opposé, BTD-val est doté d'un minorant et d'un majorant locaux à la descendance de chaque cluster. Avec de telles bornes locales, on peut

espérer que le majorant local soit nettement moins élevé que le majorant global, ce qui permettrait alors un élagage plus important, si le minorant local est de bonne qualité. Cependant, lors du lancement d'une recherche sur un cluster $\mathcal{C}_j$ fils de $\mathcal{C}_i$ (cf figure 3.4 ligne 12), nous initialisons grossièrement le majorant local avec $\top$. Une initialisation plus fine peut être réalisée en utilisant le majorant du cluster père $\mathcal{C}_i$, ce qui nous garantit que le majorant local sera toujours inférieur ou égal au majorant global, si ce dernier correspond à la valuation de la meilleure solution connue.

Les deux bornes locales utilisées par BTD-val tirent partie de l'indépendance de certains sous-problèmes. Néanmoins, cette indépendance ne signifie pas pour autant que les résultats obtenus pour certains sous-problèmes ne doivent pas influencer la résolution d'autres sous-problèmes. En effet, la solution optimale de chaque sous-problème apporte sa contribution à la solution optimale du problème pris dans son ensemble. Par conséquent, en tenant compte des solutions optimales des sous-problèmes déjà traités, on peut espérer élaguer plus de branches lors de la résolution des sous-problèmes restants. Par exemple, si $\mathcal{C}_j$ et $\mathcal{C}_{j'}$ sont deux fils de $\mathcal{C}_i$, alors on peut considérer qu'une fois instanciées les variables de $\mathcal{C}_i$, les sous-problèmes enracinés en $\mathcal{C}_j$ et $\mathcal{C}_{j'}$ sont indépendants. Supposons qu'on calcule d'abord la valuation optimale du sous-problème enraciné en $\mathcal{C}_j$. Alors, on peut éviter de développer les affectations sur $Desc(\mathcal{C}_{j'})$ dont la valuation locale à $\mathcal{C}_{j'}$ agrégée à la valuation optimale du sous-problème enraciné en $\mathcal{C}_j$ dépasse la valuation de la meilleure affectation connue sur $Desc(\mathcal{C}_i)$.
Plus généralement, les deux bornes locales employées par BTD-val ne tiennent pas compte du travail déjà réalisé. Aussi, nous ajoutons un majorant et un minorant globaux. Le majorant est la valuation de la meilleure solution connue. Le minorant correspond à la valuation de la meilleure extension de $\mathcal{A}$ sur tous les clusters précédant le cluster courant dans la numérotation compatible employée. Nous considérons ici une extension de $\mathcal{A}$, et non $\mathcal{A}$ directement, car $\mathcal{A}$ ne contient que les affectations des variables des clusters situés sur la branche allant du cluster racine au cluster courant. Notons enfin que ce majorant est identique à celui de BB, et que le minorant, bien que similaire, est meilleur que celui de BB.

L'algorithme BTD-val$_+$ est décrit à la figure 3.5. Il inclut ces deux améliorations à l'algorithme BTD-val. Étant donnés une instanciation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, il recherche la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}], \mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \prec \alpha_{\mathcal{C}_i}$, sachant que

- $V_{\mathcal{C}_i}$ est l'ensemble des variables non instanciées de $\mathcal{C}_i$,
- $\alpha_{\mathcal{C}_1}$ est la valuation de la meilleure solution connue,
- l_{tot} est la valuation de la meilleure extension $\mathcal{A}'$ de $\mathcal{A}$ sur tous les clusters précédant $\mathcal{C}_i$ dans la numérotation compatible employée ($l_{tot} = v_{\mathcal{P}}(\mathcal{A}') \prec \alpha_{\mathcal{C}_1}$),
- $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$
- $l_{\mathcal{C}_i} = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \prec \alpha_{\mathcal{C}_i}$.

Si BTD-val$_+$ trouve une telle affectation, il renvoie sa valuation, sinon il retourne une valuation supérieure ou égale à $\alpha_{\mathcal{C}_i}$. Le premier appel est effectué avec BTD-val$_+(\emptyset, \mathcal{C}_1, \mathcal{C}_1, \top, \bot, \top, \bot)$.

Théorème 3.23 *L'algorithme BTD-val$_+$ est correct, complet et termine.*

Preuve : La différence entre BTD-val et BTD-val$_+$ réside dans l'élagage par BTD-val$_+$ de branches qui, clairement, ne peuvent pas mener à une meilleure solution. Par conséquent, la correction, la complétude et la terminaison de BTD-val$_+$ se déduisent de celles de BTD-val. □

3.2.2.3 Résultats théoriques

Dans ce qui suit, nous supposerons qu'une décomposition arborescente du graphe de contraintes (ou bien une approximation) est disponible. Les paramètres de la complexité seront donc relatifs

$\mathsf{BTD\text{-}val}_+(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}, l_{tot}, \alpha_{\mathcal{C}_1}, l_{\mathcal{C}_i}, \alpha_{\mathcal{C}_i})$
1. **If** $V_{\mathcal{C}_i} = \emptyset$
2. **Then**
3. **If** $Fils(\mathcal{C}_i) = \emptyset$ **Then** Retourner $l_{\mathcal{C}_i}$
4. **Else**
5. $F \leftarrow Fils(\mathcal{C}_i)$
6. $\alpha \leftarrow \bot$
7. **While** $F \neq \emptyset$ **and** $\alpha \oplus l_{tot} \prec \alpha_{\mathcal{C}_1}$ **and** $\alpha \oplus l_{\mathcal{C}_i} \prec \alpha_{\mathcal{C}_i}$ **Do**
8. Choisir $\mathcal{C}_j$ dans F
9. $F \leftarrow F \backslash \{\mathcal{C}_j\}$
10. **If** $(\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i], v)$ est un nogood de $\mathcal{C}_i/\mathcal{C}_j$ dans N **Then** $\alpha \leftarrow \alpha \oplus v$
11. **Else**
12. $v \leftarrow \mathsf{BTD\text{-}val}_+(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i), l_{tot} \oplus \alpha, \alpha_{\mathcal{C}_1}, \bot, \alpha_{\mathcal{C}_i})$
13. $\alpha \leftarrow \alpha \oplus v$
14. Enregistrer le nogood $(\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i], v)$ de $\mathcal{C}_i/\mathcal{C}_j$ dans N
15. **EndIf**
16. **EndWhile**
17. Retourner $\alpha \oplus l_{\mathcal{C}_i}$
18. **EndIf**
19. **Else**
20. Choisir $x \in V_{\mathcal{C}_i}$
21. $d \leftarrow d_x$
22. **While** $d \neq \emptyset$ **and** $l_{tot} \prec \alpha_{\mathcal{C}_1}$ **and** $l_{\mathcal{C}_i} \prec \alpha_{\mathcal{C}_i}$ **Do**
23. Choisir a dans d
24. $d \leftarrow d \backslash \{a\}$
25. $L \leftarrow \{c = \{x, y\} \in E_{\mathcal{P},\mathcal{C}_i} | y \notin V_{\mathcal{C}_i}\}$
26. $l_a \leftarrow \bot$
27. **While** $L \neq \emptyset$ **and** $l_{tot} \oplus l_a \prec \alpha_{\mathcal{C}_1}$ **and** $l_{\mathcal{C}_i} \oplus l_a \prec \alpha_{\mathcal{C}_i}$ **Do**
28. Choisir c dans L
29. $L \leftarrow L \backslash \{c\}$
30. **If** c viole $\mathcal{A} \cup \{x \leftarrow a\}$ **Then** $l_a \leftarrow l_a \oplus \ \phi(c)$
31. **EndWhile**
32. **If** $l_{tot} \oplus l_a \prec \alpha_{\mathcal{C}_1}$ **and** $l_{\mathcal{C}_i} \oplus l_a \prec \alpha_{\mathcal{C}_i}$
33. **Then** $\alpha_{\mathcal{C}_i} \leftarrow \min(\alpha_{\mathcal{C}_i}, \mathsf{BTD\text{-}val}_+(\mathcal{A} \cup \{x \leftarrow a\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}, l_{tot} \oplus l_a, \alpha_{\mathcal{C}_1}, l_{\mathcal{C}_i} \oplus l_a, \alpha_{\mathcal{C}_i}))$
34. **EndIf**
35. **EndWhile**
36. Retourner $\alpha_{\mathcal{C}_i}$
37. **EndIf**

FIG. 3.5 : L'algorithme $\mathsf{BTD\text{-}val}_+$ pour les CSPs valués (les lignes dont le numéro est en gras correspondent aux modifications de l'algorithme BTD-val).

notamment aux caractéristiques de cette décomposition supposée connue.
Au niveau des complexités en temps et en espace, les résultats obtenus pour l'algorithme BTD (cf. théorèmes 2.11 et 2.12 page 46) sont toujours valables pour BTD-val et BTD-val$_+$.

Théorème 3.24 *BTD-val et BTD-val$_+$ ont des complexité en temps en $O(n.s^2.m.\log(d).d^{w^+ +1})$ et en espace en $O(n.s.d^s)$ avec $w^+ + 1$ la taille du plus grand $\mathcal{C}_k$ et s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$.*

Comme pour BTD, ces complexités sont comparables à celles obtenues par le Tree-Clustering (dans sa version classique comme dans celle dédiée au problème d'optimisation).

Nous allons, maintenant, montrer que BTD-val$_+$ développe moins de nœuds (ou au pire autant) que le BB. Afin de rendre possible cette comparaison, nous considérons que BB utilise le même ordre variables/valeurs que BTD-val$_+$. Comme pour la comparaison entre BTD et BT, l'emploi d'un ordre compatible facilite la comparaison entre BB et BTD. Cependant, comme il est évident qu'un ordre compatible n'est pas forcément un bon ordre pour BB, notre analyse devra être étendue ultérieurement afin de considérer des ordres différents.

Théorème 3.25 *Étant donné un ordre compatible, BTD-val$_+$ développe au plus autant de nœuds que BB.*

Preuve : BTD-val$_+$ exploite deux tests de comparaison entre un minorant et une valuation de la meilleure solution connue. Un de ces deux tests correspond exactement au test employé dans BB. Or, l'emploi des nogoods valués permet à BTD-val$_+$ d'éviter certaines redondances dans l'arbre de recherche. Donc, BTD-val$_+$ développe au plus autant de nœuds que BB. □

La même comparaison est impossible entre BTD-val et BB car les minorant et majorant employés par BTD-val sont locaux à chaque sous-problème enraciné en un cluster, alors que ceux utilisés par BB portent sur l'intégralité du problème.

3.2.3 L'algorithme BTD basé sur Forward-Checking valué

Nous présentons maintenant l'algorithme FC-BTD-val résultant de l'intégration de FC-val à BTD. Le fonctionnement de cet algorithme est similaire à celui de BTD-val. Aussi, nous ne décrirons que les différences existantes entre ces deux algorithmes.

Le majorant de l'algorithme FC-BTD-val est identique à celui de BTD-val. Par contre, son minorant est similaire à celui de FC-val, si ce n'est qu'il ne porte que sur les variables non instanciées et les contraintes propres à la descendance du cluster courant. En effet, étant donnés $\mathcal{A}$ l'affectation courante et $\mathcal{C}_i$ le cluster courant, il correspond à l'agrégation :

- des valuations des contraintes propres à $\mathcal{C}_i$ violées par $\mathcal{A}$ (c'est-à-dire $v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A})$),
- pour chaque variable x non instanciée de $Desc(\mathcal{C}_i)$, des valuations des contraintes d'un des ensembles $C(y,v)$ avec $C(y,v)$ l'ensemble des contraintes violées par l'affectation de y avec la valeur v.

À l'image de FC-val, pour chaque variable y non affectée de $Desc(\mathcal{C}_i)$, l'ensemble $C(y,v)$ choisi sera celui dont l'agrégation des valuations des contraintes est minimum. Les ensembles $C(y,v)$ possèdent la propriété suivante :

Propriété 3.26 *Soit $\mathcal{C}_i$ un cluster. Soit $y \in \mathcal{C}_i \backslash (\mathcal{C}_i \cap \mathcal{C}_{p(i)})$. $\forall v \in d_y, C(y,v) \subseteq E_{\mathcal{P},\mathcal{C}_i}$*

Preuve : Soit $c = \{x,y\} \in C(y,v)$. Comme $y \in \mathcal{C}_i \backslash (\mathcal{C}_i \cap \mathcal{C}_{p(i)})$, $x \in \mathcal{C}_i$ (par définition d'une décomposition arborescente). Donc $c \in E_{\mathcal{P},\mathcal{C}_i}$. D'où $\forall v \in d_y, C(y,v) \subseteq E_{\mathcal{P},\mathcal{C}_i}$ □

Cette propriété est fondamentale pour la validité de l'algorithme, en particulier, au niveau de la construction d'un nogood valué de $\mathcal{C}_{p(i)}/\mathcal{C}_i$. En effet, elle garantit que les contraintes prises en compte dans le calcul de la valuation optimale du sous-problème enraciné en $\mathcal{C}_i$ appartiennent toutes à ce sous-problème.

Comme pour FC-val, le minorant peut être calculé incrémentalement en introduisant les valuations $B(y,v)$ qui sont associées à chaque valeur v d'une variable y non instanciée. $B(y,v)$ correspond à l'agrégation des valuations des contraintes de $C(y,v)$. Le minorant est alors égal à l'agrégation de la valuation locale au cluster $\mathcal{C}_i$ de l'affectation courante et d'une valuation $B(y,v)$ par variable y non instanciée de $Desc(\mathcal{C}_i)$, la valuation $B(y,v)$ choisie étant celle de valuation minimum. Initialement, les $B(y,v)$ ont pour valuation $\bot$. L'extension de l'affectation courante $\mathcal{A}$ en $\mathcal{A}' = \mathcal{A} \cup \{x \leftarrow a\}$ entraîne la mise à jour suivante des valuations $B(y,v)$:

$$B(y,v) \leftarrow B(y,v) \oplus \bigoplus_{\substack{c=\{x,y\}\in \bigcup_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_i)} E_{\mathcal{P},\mathcal{C}_k} | \\ \mathcal{A}'\cup\{x\leftarrow a\} \ viole \ c}} \phi(c)$$

Quant à la valuation locale à $\mathcal{C}_i$ de $\mathcal{A}'$, elle se calcule en combinant la valuation locale de $\mathcal{A}$ au nœud parent avec $B(x,a)$.

L'algorithme FC-BTD-val est décrit à la figure 3.6. Étant donnés une instanciation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, il recherche la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}], \mathcal{C}_{p(i)}/\mathcal{C}_i}}(\mathcal{B}) \prec \alpha_{\mathcal{C}_i}$, sachant que

- $V_{\mathcal{C}_i}$ est l'ensemble des variables non instanciées de $\mathcal{C}_i$,
- $\alpha_{\mathcal{C}_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}] = \mathcal{B}'[\mathcal{C}_i \cap \mathcal{C}_{p(i)}]$
- $l_{\mathcal{C}_i} = v_{\mathcal{P},\mathcal{C}_i}(\mathcal{A}) \prec \alpha_{\mathcal{C}_i}$,
- M est le minorant amélioré de FC-val restreint aux variables non instanciées de $Desc(\mathcal{C}_i)$ et aux contraintes propres à la descendance de $\mathcal{C}_i$.

Si FC-BTD-val trouve une telle affectation, il renvoie sa valuation, sinon il retourne une valuation supérieure ou égale à $\alpha_{\mathcal{C}_i}$. Le premier appel est effectué avec BTD-val$(\emptyset, \mathcal{C}_1, \mathcal{C}_1, \top, \bot, \bot)$.
La mise à jour des valuations $B(y,v)$ est effectuée par la fonction Mise-à-jour2 (pour laquelle x est la variable courante). Cette fonction calcule et renvoie le nouveau minorant portant sur les variables non instanciées de la descendance de $\mathcal{C}_i$. Comme pour FC-val, les fonctions Push-Domains et Pop-Domains réalisent respectivement la sauvegarde et la restauration des $B(y,v)$ avant et après leurs modifications.

Théorème 3.27 *L'algorithme FC-BTD-val est correct, complet et termine.*

Preuve : Pour prouver ce théorème, on raisonne comme dans la preuve du théorème 3.22. □

Remarquons enfin que les résultats théoriques et les améliorations présentés pour BTD-val s'étendent sans difficulté à FC-BTD-val. En particulier, on pourrait définir FC-BTD-val$_+$ qui exploiterait, en plus des deux bornes locales de FC-BTD-val, le minorant et la majorant de FC-val.

3.3 Discussion

Les travaux les plus proches de BTD-val semblent être le Tree-Clustering [DF01] et la méthode par programmation dynamique de Koster [Kos99]. À la base, ces deux méthodes sont relativement proches l'une de l'autre. En particulier, elles souffrent du même handicap, à savoir la quantité

FC-BTD-val($\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i}, M$)
1. **If** $V_{\mathcal{C}_i} = \emptyset$
2. **Then**
3. **If** $Fils(\mathcal{C}_i) = \emptyset$ **Then** Retourner $l_{\mathcal{C}_i}$
4. **Else**
5. $F \leftarrow Fils(\mathcal{C}_i)$
6. $\alpha \leftarrow l_{\mathcal{C}_i}$
7. **While** $F \neq \emptyset$ **and** $\alpha \prec \alpha_{\mathcal{C}_i}$ **Do**
8. Choisir $\mathcal{C}_j$ dans F
9. $F \leftarrow F \backslash \{\mathcal{C}_j\}$
10. **If** $(\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i], v)$ est un nogood de $\mathcal{C}_i/\mathcal{C}_j$ dans N **Then** $\alpha \leftarrow \alpha \oplus v$
11. **Else**
12. $v \leftarrow$ FC-BTD-val($\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i), \top, \bot, \bot$)
13. $\alpha \leftarrow \alpha \oplus v$
14. Enregistrer le nogood $(\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i], v)$ de $\mathcal{C}_i/\mathcal{C}_j$ dans N
15. **EndIf**
16. **EndWhile**
17. Retourner α
18. **EndIf**
19. **Else**
20. Choisir $x \in V_{\mathcal{C}_i}$
21. $d \leftarrow d_x$
22. **While** $d \neq \emptyset$ **and** $M \prec \alpha_{\mathcal{C}_i}$ **Do**
23. Choisir a dans d
24. $d \leftarrow d \backslash \{a\}$
32. Push-Domains
33. $M_a \leftarrow$ Mise-à-jour2($V_{\mathcal{C}_i} \backslash \{x\}, \mathcal{A} \cup \{x \leftarrow a\}, l_{\mathcal{C}_i} \oplus B(x,a), x$)
34. **If** $M_a \prec \alpha_{\mathcal{C}_i}$
35. **Then**
36. $\alpha_{\mathcal{C}_i} \leftarrow \min(\alpha_{\mathcal{C}_i},$ FC-BTD-val($\mathcal{A} \cup \{x \leftarrow a\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}, \alpha_{\mathcal{C}_i}, l_{\mathcal{C}_i} \oplus B(x,a), M_a$))
37. **EndIf**
38. Pop-Domains
39. **EndWhile**
40. Retourner $\alpha_{\mathcal{C}_i}$
41. **EndIf**

Mise-à-jour2($\mathcal{C}_i, V_{\mathcal{C}_i}, \mathcal{A}, l, x$)
1. $L \leftarrow \{c = \{x, y\} \in \bigcup_{\mathcal{C}_k \subseteq Desc(\mathcal{C}_i)} E_{\mathcal{P},\mathcal{C}_k} | y \in V_{\mathcal{C}_i}\}$
2. **While** $L \neq \emptyset$ **Do**
3. Choisir $c = \{x, y\}$ dans L
4. $L \leftarrow L \backslash \{c\}$
5. **For each** $v \in d_y$ **Do**
6. **If** $\mathcal{A} \cup \{y \leftarrow v\}$ viole c **Then** $B(y,v) \leftarrow B(y,v) \oplus \phi(c)$
7. **EndWhile**
8. Retourner $l \oplus \bigoplus_{y \in Desc(\mathcal{C}_i) \backslash (\mathcal{C}_i \backslash V_{\mathcal{C}_i})} \left(\min_{v \in d_y} B(y,v) \right)$

FIG. 3.6 : L'algorithme FC-BTD-val pour les CSPs valués.

trop importante de mémoire requise. Les différences et points communs entre BTD-val et le Tree-Clustering dans sa version dédiée à la résolution de problèmes d'optimisation sont les mêmes qu'entre BTD et le Tree-Clustering (cf. section 2.4). Aussi, nous n'en dirons pas plus sur ce sujet .

Par rapport à la méthode par programmation dynamique de Koster, BTD-val présente des complexités en temps et en espace meilleures. Ensuite, ces méthodes diffèrent significativement au niveau du calcul de la décomposition arborescente (ou de son approximation) employée. Dans le cas de BTD-val, comme pour celui de BTD, ce calcul s'effectue via une triangulation du graphe de contraintes, alors que l'approche par programmation dynamique exploite une méthode heuristique basée sur une résolution d'un problème de flot dans les réseaux. De plus, la méthode par programmation dynamique se distingue en proposant plusieurs prétraitements, dont un, en particulier, qui permet de réduire la taille du graphe de contraintes. Intégrer à BTD-val certains de ces prétraitements pourrait se révéler fort utile en pratique.

L'algorithme BTD-val n'est pas très éloigné d'une approche comme celle des poupées russes [VLS96]. En effet, pour trouver la valuation optimale d'un VCSP, BTD-val résout plusieurs sous-problèmes suivant un ordre compatible préétabli, les clusters dans BTD-val jouant un rôle proche de celui des variables dans RDS. Cependant, les deux méthodes exploitent différemment les valuations optimales des sous-problèmes. Quant à la variante TRDS [MS00] de RDS, comme BTD-val, elle tire partie de la structure du graphe de contraintes pour déterminer si certains problèmes sont indépendants ou non. Si l'indépendance des sous-problèmes est exploitée de façon similaire, les sous-problèmes de TRDS restent tout de même conceptuellement différents de ceux de BTD-val. Il en est de même pour l'adaptation [LMS02] de l'algorithme Pseudo-Tree Search et de sa combinaison avec SRDS.

3.4 Conclusion

Dans ce chapitre, nous avons étendu la méthode BTD au cadre des CSPs valués. Cette extension se traduit par la définition d'un cadre formel (en particulier de la notion de nogood valué structurel) et des algorithmes BTD-val et FC-BTD-val basés respectivement sur BB et FC-val. Nous avons également présenté des améliorations possibles de BTD-val et FC-BTD-val, comme l'algorithme BTD-val$_+$. Nous avons établi qu'en théorie, ces versions améliorées de BTD-val et de FC-BTD-val produisent moins de nœuds que BB et FC-val respectivement. Cependant, une étude expérimentale devra être menée pour pouvoir juger des gains obtenus en pratique. Comme pour BTD, les complexités sont en $O(ns^2m\log(d).d^{w^+ +1})$ pour le temps et en $O(nsd^s)$ pour l'espace avec $w^+ + 1$ la taille du plus grand cluster et s la taille de la plus grande intersection entre deux clusters.

Parmi les extensions possibles de ce travail, l'utilisation comme algorithme de base de méthodes plus performantes que BB ou FC-val devra être étudiée. En particulier, vouloir intégrer l'algorithme des poupées russes ou des algorithmes exploitant de la consistance d'arc directionnelle [Wal94, Wal96, LM96] semble être une extension naturelle par rapport à la notion d'ordre d'énumération compatible employé par notre méthode.

Chapitre 4

Résolution concurrente coopérative avec échange de nogoods

Lancer plusieurs solveurs en concurrence avec chacun une heuristique différente pour ordonner les variables et/ou les valeurs semble être une alternative intéressante aux vues de la difficulté à choisir une bonne heuristique. De telles approches présentent souvent des résultats satisfaisants pour les problèmes consistants. Par contre, pour les problèmes inconsistants, les gains sont faibles, voire même inexistants. Une parade à ce défaut peut consister à ajouter une forme de coopération entre les solveurs. Dans [MV96], la coopération repose sur un échange de nogoods (i.e. d'affectations ne conduisant pas à une solution).
Dans ce chapitre, nous poursuivons ce travail afin de déterminer si l'échange de nogoods constitue ou non une forme efficace de coopération. Nous proposons d'abord trois schémas qui diffèrent les uns des autres par la manière dont sont échangés les nogoods. En particulier, dans deux de ces schémas, nous nous efforçons de limiter les communications de nogoods au strict nécessaire. Nous intégrons également à chaque solveur une phase d'interprétation qui permet de tirer le meilleur parti des nogoods reçus.
Expérimentalement, nous avons obtenu des résultats très satisfaisants avec des accélérations linéaires et superlinéaires pour des problèmes consistants comme pour des problèmes inconsistants. Dans certains cas, l'approche peut se révéler même plus efficace que certains algorithmes classiques.
Une partie de ce travail a été publiée dans [Ter01].

4.1 Méthode concurrente avec échange de nogoods

4.1.1 Présentation

Reprenant l'idée de Martinez et Verfaillie, nous définissons une nouvelle méthode concurrente avec échange de nogoods classiques (cf. définition 1.17 page 20), qui se veut orientée vers des systèmes dotés d'un ou plusieurs processeurs. Nous associons donc un processus à chaque solveur. Chacun des p solveurs, que nous exécutons en concurrence sur le même CSP, utilise le même algorithme de résolution, à savoir FC-NR. Cependant, chacun d'eux possède une heuristique d'ordonnancement des variables et/ou des valeurs différente. Ainsi, chacun parcourt un arbre de recherche distinct. Comme pour Martinez et Verfaillie, la recherche se termine dès qu'un des solveurs découvre une solution ou l'inconsistance du problème. Quant à la coopération, elle repose, comme précédemment, sur l'échange de nogoods entre les solveurs, qui peuvent alors élaguer une

partie de leur arbre de recherche au moyen d'un nogood produit par un autre solveur. Dans notre approche, nous supposons :

- que nous disposons d'une mémoire partagée accessible à tous les processus et suffisamment grande pour contenir l'instance CSP à résoudre, ainsi que l'ensemble des nogoods mémorisés et l'affectation courante de chaque solveur,
- que chaque processus peut communiquer, par échange de messages, avec n'importe quel autre processus.

Nous discutons, dans le paragraphe 4.2.2, d'un éventuel allègement de la première hypothèse. Plusieurs mises en œuvre de cette approche sont envisageables suivant la façon dont sont réalisés les échanges de nogoods. Nous proposons trois schémas.

Dans un premier schéma (noté $\mathcal{S}_0$ et présenté à la figure 4.1) que nous qualifierons de basique, chaque solveur communique les nogoods qu'il découvre à tous les autres solveurs et les ajoute à l'ensemble des nogoods trouvés. D'après un tel schéma, à chaque découverte d'un nogood, un solveur doit envoyer en tout $p-1$ messages d'informations à ses partenaires. Bien que le nombre de nogoods soit majoré par $O(n^2d^2)$ (du fait de l'arité des nogoods limitée à 2), on se rend compte que le coût des communications peut devenir très important, voire prohibitif. De plus, un nogood donné n'est pas nécessairement utile à tous les solveurs. C'est pourquoi nous définissons un second schéma (noté $\mathcal{S}_{light}$ et illustré à la figure 4.2) dans lequel les échanges de nogoods seront restreints. En fait, un nogood ne sera communiqué qu'aux solveurs susceptibles de l'utiliser immédiatement. Cette notion d'utilité d'un nogood est détaillée dans la partie suivante. Un inconvénient potentiel de ce schéma est le coût en temps de la limitation des échanges et des communications elles-mêmes. En effet, le temps passé à déterminer quels solveurs doivent recevoir un nogood et à effectuer les communications est autant de temps perdu pour la recherche d'une solution. Aussi, dans un troisième schéma (noté $\mathcal{S}_{gest}$ et représenté à la figure 4.3), nous ajoutons aux p solveurs un processus appelé "*gestionnaire de nogoods*", dont le rôle est de décharger les solveurs de la communication des nogoods et de leur ajout à l'ensemble des nogoods déjà connus. Ainsi, quand un solveur découvre un nogood, il en informe aussitôt le gestionnaire qui répandra alors la nouvelle aux solveurs intéressés. De cette manière, le solveur n'émet qu'un seul message et retourne plus rapidement à sa tâche première, qui est la résolution du CSP.

Enfin, concernant les nogoods produits, ils sont ajoutés au problème initial sous la forme de nouvelles contraintes (ou d'un renforcement des contraintes existantes). Chaque solveur, grâce à l'algorithme FC-NR, les exploitera donc de façon classique pour backjumper ou pour filtrer davantage les domaines. Toutefois, une telle utilisation des nogoods n'est pas suffisante pour en tirer pleinement parti. En effet, un solveur peut avoir commencé l'exploration d'un sous-arbre avant d'avoir reçu le nogood qui lui aurait permis d'éviter cette exploration. Dans un tel cas, l'utilisation des nogoods au niveau du filtrage s'avère insuffisante pour exploiter le nogood. Ainsi, suivant la taille et le nombre de tels sous-arbres, la perte en terme d'efficacité peut devenir importante. Aussi, quel que soit le schéma employé, nous ajoutons à FC-NR une phase d'interprétation, qui, grâce aux nogoods reçus, limite la taille de l'arbre de recherche, en stoppant le développement de branches vouées à l'échec.

Nous commençons par détailler la restriction des échanges et le gestionnaire de nogoods.

4.1.2 Restriction des échanges et gestionnaire de nogoods

4.1.2.1 Restriction des échanges et rôle du gestionnaire

Le gestionnaire de nogoods a pour mission :

- de mettre à jour l'ensemble de nogoods,

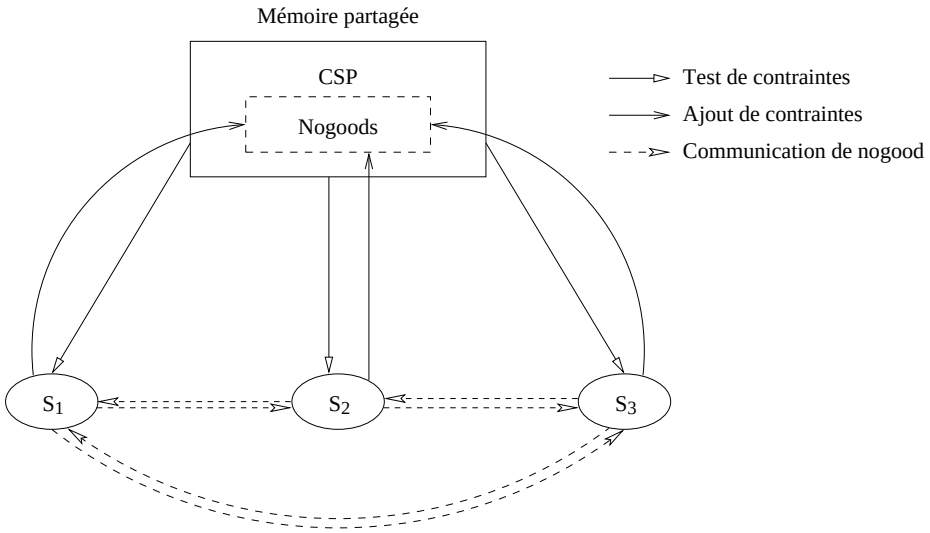

FIG. 4.1 : Schéma $\mathcal{S}_0$ de notre méthode concurrente avec échange de nogoods pour 3 solveurs. Les flèches en trait plein représentent des accès à la mémoire partagée, celles en trait discontinu des communications inter-processus.

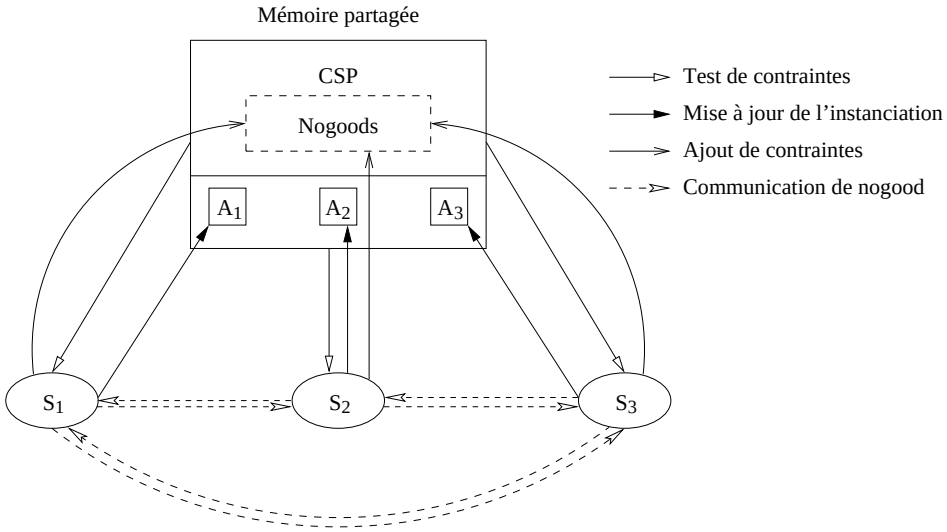

FIG. 4.2 : Schéma $\mathcal{S}_{light}$ de notre méthode concurrente avec échange de nogoods pour 3 solveurs. Les flèches en trait plein représentent des accès à la mémoire partagée, celles en trait discontinu des communications inter-processus.

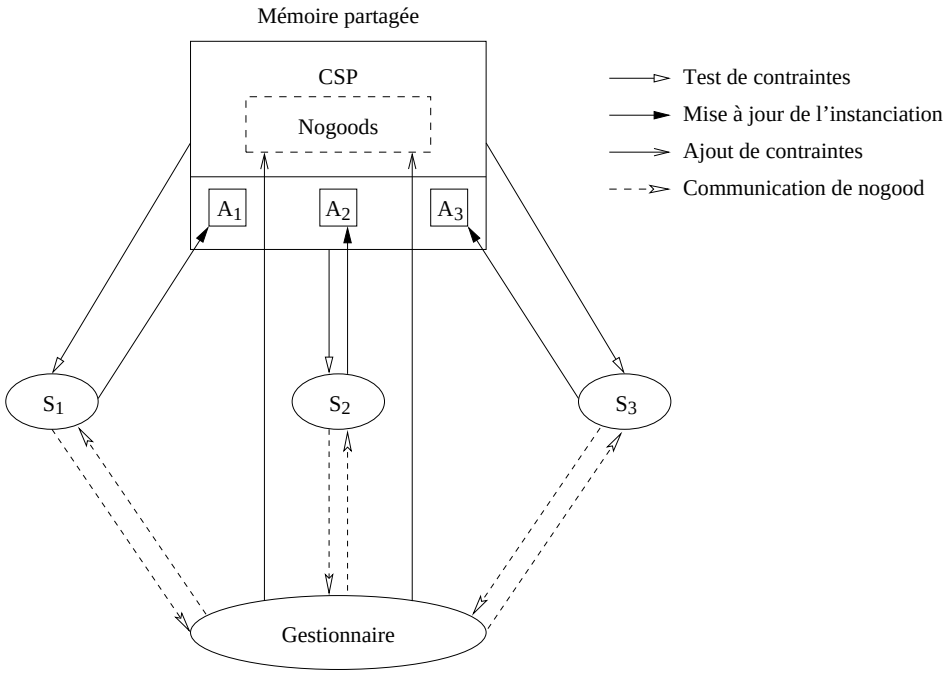

FIG. 4.3 : Schéma $\mathcal{S}_{gest}$ de notre méthode concurrente avec échange de nogoods pour 3 solveurs. Les flèches en trait plein représentent des accès à la mémoire partagée, celles en trait discontinu des communications inter-processus.

- de communiquer les nogoods découverts aux solveurs.

Mettre à jour l'ensemble de nogoods revient en fait à ajouter des contraintes au problème initial ou à renforcer des contraintes existantes. À un nogood d'arité 1 (respectivement 2) correspond une contrainte unaire (resp. binaire). Tout nogood communiqué au gestionnaire est ajouté à l'ensemble (s'il n'y figurait pas déjà). Dans les schémas S_0 et S_{light}, ce travail est directement réalisé par le solveur qui découvre un nogood.
Concernant les communications, l'échange de nogoods est restreint suivant la même méthode dans $\mathcal{S}_{light}$ et $\mathcal{S}_{gest}$. À ce niveau, la seule différence notable entre ces deux schémas réside dans la nature du processus qui met en œuvre ces restrictions. Dans $\mathcal{S}_{light}$ il s'agit d'un solveur (celui qui vient de trouver le nogood à communiquer) alors que dans $\mathcal{S}_{gest}$, cette tâche est déléguée au gestionnaire.

En imposant des restrictions sur les échanges de nogoods, dans les schémas $\mathcal{S}_{light}$ et $\mathcal{S}_{gest}$, notre but est de limiter le nombre de communications, mais aussi de ne pas interrompre inutilement les solveurs dans leur recherche. Aussi, les solveurs ne doivent être informés que des nogoods qui peuvent leur être utiles.

Définition 4.1 (utilité d'un nogood) *Un nogood est dit* ***utile*** *à un solveur s'il permet à ce solveur de limiter la taille de son arbre de recherche.*

Parmi ces nogoods utiles, on peut distinguer deux grandes catégories de nogoods :

- les nogoods utiles dès leur réception : ce sont des nogoods dont la réception par un solveur va lui permettre de limiter la taille de la recherche courante (voire même de mettre un terme immédiatement à cette recherche),
- ceux qui seront utiles ultérieurement grâce au filtrage.

L'exploitation de ces derniers est réalisée via les contraintes ajoutées ou renforcées, et donc grâce à la mémoire partagée. Par conséquent, seuls les nogoods utiles dès leur réception doivent être communiqués. Ainsi, le solveur qui découvre un nogood (dans $\mathcal{S}_{light}$) ou le gestionnaire (dans $\mathcal{S}_{gest}$) va établir si un nogood peut être utile dès réception ou non à un solveur. À cette fin, la seule connaissance dont il dispose sur le solveur destinataire est son instanciation courante, qui est stockée dans la mémoire partagée (ces accès à la mémoire partagée n'apparaissent pas dans les figures 4.2 et 4.3 pour ne pas les surcharger). Aussi, le lemme suivant caractérise l'utilité des nogoods, dès réception, en fonction de leur arité et de l'affectation courante du solveur considéré.

Lemme 4.2 (caractérisation de l'utilité des nogoods dès réception)
Soit S un solveur et $\mathcal{A}_S$ son affectation courante.

(a) un nogood unaire est toujours utile pour S,

(b) un nogood binaire $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ est utile pour S si x_i et x_j sont affectées respectivement à a et b dans $\mathcal{A}_S$,

(c) un nogood binaire $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ est utile pour S si, dans $\mathcal{A}_S$, x_i (resp. x_j) est affectée à a (resp. b), x_j (resp. x_i) n'est pas encore instanciée et $b \in d_{x_j}(\mathcal{A}_S)$ (resp. $a \in d_{x_i}(\mathcal{A}_S)$).

Preuve :

(a) Un nogood unaire correspond à une contrainte unaire (i.e. à interdire une valeur). Donc, dans tous les cas, un nogood unaire permettra de réduire définitivement la taille du domaine de la variable concernée. Par conséquent, il sera toujours utile.

(b) Soit un nogood binaire $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$.
Si l'affectation courante contient x_i instanciée à a et x_j affectée à b, alors le nogood sera utile. En effet, grâce à ce nogood, on sait que tout sous-arbre issu d'une branche dans laquelle x_i et x_j sont affectées respectivement à a et b ne contient pas de solution. On n'a donc pas besoin d'explorer un tel sous-arbre (si l'exploration a débuté, il est inutile de la poursuivre).

(c) Soit un nogood binaire $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$.
Supposons que x_i soit affectée à a et que x_j ne soit pas encore instanciée. Alors, ce nogood autorise le solveur à supprimer par filtrage la valeur b du domaine d_{x_j}. □

À partir de ce lemme, nous précisons quels solveurs doivent recevoir tels ou tels nogoods (en fonction de leur utilité) :

(a) tout nogood unaire est communiqué à tous les solveurs (hormis celui qui l'a découvert),

(b) un nogood binaire $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ est transmis à tout solveur (hormis celui qui l'a découvert) dont l'affectation courante contient x_i instanciée à a ou x_j affectée à b.

Autrement dit, les nogoods communiqués à un solveur sont ceux susceptibles d'être utilisés dès réception par ce solveur. Cependant, cela ne garantit pas que le nogood sera réellement utilisé. En particulier, dans le cas (b), le solveur peut avoir backtracké entre le moment de l'émission du message et celui de sa réception. De plus, si, par exemple, x_i est instanciée à a et que x_j n'est pas affectée, le gestionnaire ignore si la valeur b appartient encore au domaine courant de x_j. Si b a déjà été supprimée, le nogood ne sera pas utile.

Toujours dans le but de limiter le coût des communications, seule l'affectation $\mathcal{A}$ du nogood $(\mathcal{A}, J)$ est transmise. En fait, communiquer la justification J n'est pas nécessaire car ce nogood vient d'être ajouté au problème sous la forme d'une contrainte c portant sur toutes les variables de $X_{\mathcal{A}}$. Grâce à l'information reçue, les solveurs peuvent donc interdire $\mathcal{A}$ avec pour justification c.

Exemple 4.3 *Utilisons la méthode concurrente coopérative avec trois solveurs S_1, S_2 et S_3 pour résoudre le CSP de l'exemple 1.4 (page 12). S_1 exploite l'ordre sur les variables $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, S_2 l'ordre $(x_2, x_1, x_3, x_4, x_5, x_6, x_7)$ et S_3 l'ordre $(x_4, x_1, x_3, x_5, x_2, x_6, x_7)$. Pour les valeurs, les solveurs emploient l'ordre alphabétique. De plus, nous supposons que l'affectation courante de S_2*

(respectivement S_3) est $\{x_2 \leftarrow a, x_1 \leftarrow b\}$ (resp. $\{x_4 \leftarrow a, x_1 \leftarrow a\}$).
Si S_1 produit le nogood $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$, il ne le communiquera qu'à S_3. En effet, seul S_3 est susceptible de l'utiliser immédiatement (en supprimant par filtrage c du domaine de x_3). Lorsque S_2 développera l'affectation $\{x_2 \leftarrow b, x_1 \leftarrow a\}$, S_2 aura alors besoin de ce nogood. Il pourra l'exploiter via la mémoire partagée, puisque ce nogood a été ajouté en supprimant le couple (a, c) de $r_{c_{13}}$. Par contre, si S_1 produit le nogood $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$, il communiquera ce nogood à S_2 et S_3. Ainsi, dès la réception de cette information, S_2 et S_3 peuvent supprimer b du domaine de x_1. Dans les deux cas, seule l'affectation interdite est transmise, c'est-à-dire $\{x_1 \leftarrow a, x_3 \leftarrow c\}$ ou $\{x_1 \leftarrow b\}$.

4.1.2.2 Evaluation du nombre de messages

Dans cette partie, nous évaluons, pour chacun des trois schémas, le nombre de messages échangés par tous les processus sur l'ensemble de la résolution.
Soit N le nombre total de nogoods échangés par l'ensemble des solveurs. On comptabilise U nogoods unaires et B binaires. Notons que, parmi ces N nogoods, il peut exister des doublons (deux solveurs pouvant découvrir séparément un même nogood).

Dans le schéma $\mathcal{S}_{gest}$, les nogoods sont d'abord transmis par les solveurs au gestionnaire. Sur l'ensemble de la recherche, les solveurs envoient au gestionnaire U messages pour les nogoods unaires et B pour les binaires. Le gestionnaire envoie alors u nogoods unaires aux $p-1$ solveurs. Ces u nogoods correspondent aux U nogoods desquels on a retiré les doublons. De même, pour les nogoods binaires, les doublons ne sont pas communiqués. De plus, les nogoods binaires qui restent ne sont pas systématiquement communiqués à tous les autres solveurs, le gestionnaire restreignant le nombre de destinataires. Soit b le nombre total de messages envoyés par le gestionnaire pour les nogoods binaires. Globalement, dans le schéma avec gestionnaire, on émet $U + u(p-1)$ messages pour les nogoods unaires et $B + b$ messages pour les binaires.

Dans le schéma $\mathcal{S}_0$, chaque solveur communique les nogoods qu'il trouve aux $p-1$ autres solveurs. Ainsi, $U(p-1)$ messages sont émis pour les nogoods unaires et $B(p-1)$ pour les binaires. Par contre, dans le schéma $\mathcal{S}_{light}$, le solveur établit l'utilité d'un nogood, avant de le communiquer aux autres solveurs, comme le ferait le gestionnaire. Ainsi, seulement $u(p-1)$ messages seront émis pour les nogoods unaires et b pour les binaires.

Par conséquent, dans le pire des cas, le schéma $\mathcal{S}_{gest}$ produit jusqu'à N messages supplémentaires par rapport aux schémas $\mathcal{S}_0$ et $\mathcal{S}_{light}$. Mais, en général, nous avons observé que u et b étaient suffisamment petits pour que le schéma $\mathcal{S}_{gest}$ expédie moins de messages que le schéma $\mathcal{S}_0$. En particulier, le gestionnaire restreint significativement le nombre de messages émis pour les nogoods binaires. Par contre, l'avantage du schéma $\mathcal{S}_{gest}$ par rapport aux schémas $\mathcal{S}_0$ et $\mathcal{S}_{light}$ est que, pour un faible surcoût en communications, il décharge les solveurs de l'envoi de messages. Les solveurs peuvent ainsi se consacrer pleinement à la résolution du problème. Remarquons enfin que cette comparaison se place dans un cadre idéal où les solveurs développent le même arbre et produisent les mêmes nogoods indépendamment du schéma utilisé. En pratique, les recherches et donc le nombre de nogoods produits peuvent différer sensiblement suivant les schémas.

4.1.3 Phase d'interprétation

Avant de décrire la phase d'interprétation, nous introduisons la notation suivante :

Notation 4.4 *Soit $\mathcal{A}$ une affectation. On note $\mathcal{A}_{x_k}$ la restriction de $\mathcal{A}$ aux variables instanciées avant x_k, x_k comprise.*

Quel que soit le schéma employé, chaque solveur intègre une phase d'interprétation à l'algorithme FC-NR. La phase d'interprétation est appliquée à chaque fois qu'un nogood parvient au solveur. Pour information, on teste si un message a été reçu après chaque développement d'un nœud et avant de réaliser le filtrage.
Dans la phase d'interprétation, les solveurs analysent les nogoods reçus afin de limiter la taille de leur arbre de recherche en stoppant le développement d'affectations menant à des échecs ou en appliquant un filtrage supplémentaire. Pour les nogoods unaires, cette phase se traduit par une suppression définitive d'une valeur et éventuellement par un retour en arrière. La méthode 4.5 détaille cette phase pour de tels nogoods :

Méthode 4.5 (phase d'interprétation pour les nogoods d'arité 1)
Soient $\mathcal{A}$ l'affectation courante et $(\{x_i \leftarrow a\}, J)$ le nogood reçu par le solveur. On note K l'ensemble des value-killers de d_{x_i}.
On supprime a de d_{x_i}. De plus :

(a) Si x_i n'est pas affectée et que $d_{x_i}(\mathcal{A})$ est vide :
  (i) on mémorise le nogood $(\mathcal{A}[X_K], K)$.
  (ii) Si la mémorisation du nogood rend vide un domaine, alors on revient en arrière jusqu'à la variable la moins profonde entre la variable la plus profonde de X_K et la variable la plus profonde de $X_{K'}$ (avec K' l'ensemble des value-killers de ce domaine).
  Sinon on revient en arrière jusqu'à la variable la plus profonde de X_K.

(b) Si x_i est affectée à la valeur a :
on backjumpe jusqu'à la variable x_i. Soit $\mathcal{A}' \cup \{x_i \leftarrow a\}$ l'affectation ainsi obtenue.
Si $d_{x_i}(\mathcal{A}')$ est vide :
  (i) on mémorise le nogood $(\mathcal{A}'[X_K], K)$.
  (ii) Si la mémorisation du nogood rend vide un domaine, alors on revient en arrière jusqu'à la variable la moins profonde entre la variable la plus profonde de X_K et la variable la plus profonde de $X_{K'}$ (avec K' l'ensemble des value-killers de ce domaine).
  Sinon on revient en arrière jusqu'à la variable la plus profonde de X_K.

Notons que la suppression de a du domaine d_{x_i} est définitive, ce qui implique que quelle que soit l'affectation $\mathcal{A}$, la valeur a n'appartiendra pas à $d_{x_i}(\mathcal{A})$.
Les phases de backjump des cas (a)(ii) et (b)(ii) permettent de poursuivre l'exploration de l'arbre de recherche avec une affectation FC-consistante. Autrement dit, ces deux phases de backjump garantissent que tous les domaines courants soient non vides avant la construction de la prochaine affectation.
Enfin, dans le cas où x_i est affectée à une valeur différente de a, il n'y a rien d'autre à faire que supprimer la valeur a du domaine d_{x_i}. En effet, d'une part, le domaine $d_{x_i}(\mathcal{A})$ ne peut être vide, car x_i est affectée à une valeur distincte de a. D'autre part, recevoir le nogood $(\{x_i \leftarrow a\}, J)$ ne donne aucune information sur la réussite ou l'échec de l'extension de l'affectation $\mathcal{A}$.

Théorème 4.6 *La phase d'interprétation pour les nogoods d'arité 1 est correcte.*

Preuve :
Comme $(\{x_i \leftarrow a\}, J)$ est un nogood, l'affectation $\{x_i \leftarrow a\}$ ne peut être étendue en une solution. On peut donc retirer a du domaine d_{x_i} sans changer la nature du problème.
À présent, nous étudions la validité de chacun des deux cas :

(a) Cas où x_i n'est pas affectée :
La suppression de a est susceptible de rendre le domaine $d_{x_i}(\mathcal{A})$ vide. Si $d_{x_i}(\mathcal{A})$ est vide, alors l'affectation $\mathcal{A}$ est FC-inconsistante. D'après les théorèmes 1.22 et 1.23 (page 21), $(\mathcal{A}[X_K], K)$ est un nogood. De plus, d'après le lemme 1.25 (page 22), il est correct de backjumper jusqu'à

la variable la plus profonde de X_K. Si la mémorisation du nogood $(\mathcal{A}[X_K], K)$ rend le domaine de x_l vide, alors, d'après le lemme 1.25, il est aussi correct de backjumper jusqu'à la variable la plus profonde de $X_{K'}$. Donc, dans un tel cas, on peut backjumper jusqu'à la variable la moins profonde des deux.

(b) Cas où x_i est affectée à la valeur a :
Comme $(\{x_i \leftarrow a\}, J)$ est un nogood, toute affectation contenant x_i instanciée à a ne conduira pas à une solution. Par conséquent, il est inutile de développer de telles affectations et il est donc correct de backjumper jusqu'à x_i.
Soit $\mathcal{A}' \cup \{x_i \leftarrow a\}$ l'affectation obtenue suite au backjump. Avec la suppression de a de d_{x_i}, $d_{x_i}(\mathcal{A}')$ peut devenir vide. Si tel est le cas, d'après les théorèmes 1.22 et 1.23, il faut mémoriser le nogood $(\mathcal{A}'[X_K], K)$. Démontrer que la phase (ii) de backjump est correcte s'effectue comme pour (a). □

Exemple 4.7
Nous poursuivons l'exemple 4.3. Lorsque S_3 reçoit le nogood $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$, il supprime définitivement b de d_{x_1}. Pour S_2, la réception de ce nogood engendre d'une part la suppression définitive de b de d_{x_1}, d'autre par l'arrêt de la recherche courante. S_2 poursuit alors directement sa recherche avec l'instanciation $\{x_2 \leftarrow a, x_1 \leftarrow c\}$.

Pour les nogoods binaires, la phase revient à appliquer un filtrage supplémentaire et éventuellement à effectuer un retour en arrière. La méthode 4.8 décrit les actions réalisées durant cette phase pour des nogoods binaires.

Méthode 4.8 (phase d'interprétation pour les nogoods d'arité 2)
Soit $\mathcal{A}$ l'affectation courante. Soit $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ le nogood reçu par le solveur.

(a) Si $\{x_i \leftarrow a\} \subseteq \mathcal{A}$ (resp. $\{x_j \leftarrow b\} \subseteq \mathcal{A}$) et $x_j \notin X_{\mathcal{A}}$ (resp. $x_i \notin X_{\mathcal{A}}$) :
on supprime par filtrage b (resp. a) de $d_{x_j}(\mathcal{A}_{x_i})$ (resp. $d_{x_i}(\mathcal{A}_{x_j})$).
Si $d_{x_j}(\mathcal{A})$ (resp. $d_{x_i}(\mathcal{A})$) est vide :

(i) on mémorise le nogood $(\mathcal{A}[X_K], K)$ avec K l'ensemble des value-killers de d_{x_j} (resp. d_{x_i}).

(ii) Si la mémorisation du nogood rend vide un domaine, alors on revient en arrière jusqu'à la variable la moins profonde entre la variable la plus profonde de X_K et la variable la plus profonde de $X_{K'}$ (avec K' l'ensemble des value-killers de ce domaine).
Sinon on revient en arrière jusqu'à la variable la plus profonde de X_K.

(b) Si $\{x_i \leftarrow a, x_j \leftarrow b\} \subseteq \mathcal{A}$:
on revient en arrière sur la variable la plus profonde parmi x_i et x_j.
Si x_j (resp. x_i) est cette variable, on note $\mathcal{A}' \cup \{x_j \leftarrow b\}$ (resp. $\mathcal{A}' \cup \{x_i \leftarrow a\}$) l'affection obtenue.
On supprime par filtrage b (resp. a) de $d_{x_j}(\mathcal{A}_{x_i})$ (resp. $d_{x_i}(\mathcal{A}_{x_j})$).
Si $d_{x_j}(\mathcal{A}')$ (resp. $d_{x_i}(\mathcal{A}')$) est vide :

(i) on mémorise le nogood $(\mathcal{A}'[X_K], K)$ avec K l'ensemble des value-killers de d_{x_j} (resp. d_{x_i}).

(ii) Si la mémorisation du nogood rend vide un domaine, alors on revient en arrière jusqu'à la variable la moins profonde entre la variable la plus profonde de X_K et la variable la plus profonde de $X_{K'}$ (avec K' l'ensemble des value-killers de ce domaine).
Sinon on revient en arrière jusqu'à la variable la plus profonde de X_K.

Contrairement à la phase d'interprétation pour les nogoods unaires, les suppressions ici ne sont pas définitives. En fait, elles sont de même nature que les suppressions par filtrage de l'algorithme FC-NR.

Théorème 4.9 *La phase d'interprétation pour les nogoods d'arité 2 est correcte.*

Preuve :

(a) Supposons que x_j soit la variable la plus profonde parmi x_i et x_j (le raisonnement est similaire si x_i est la plus profonde). Le nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ interdit à x_j de prendre pour valeur b tant que x_i est affectée à la valeur a. Donc, on peut supprimer b de $d_{x_j}(\mathcal{A}_{x_i})$ par filtrage.
Si $d_{x_j}(\mathcal{A})$ est vide, on raisonne comme dans la preuve du théorème 4.6 pour prouver que la mémorisation du nogood et la phase de backjump sont correctes.

(b) Comme $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ est un nogood, toute affectation contenant $\{x_i \leftarrow a, x_j \leftarrow b\}$ ne peut conduire à une solution. Il est donc correct de revenir à la variable la plus profonde parmi x_i et x_j.
Supposons que x_j soit cette variable (le raisonnement est similaire si x_i est la plus profonde). $\mathcal{A}' \cup \{x_j \leftarrow b\}$ est donc l'affectation obtenue suite au backjump. Comme pour le point (a), on montre que le filtrage, ainsi que la mémorisation et la phase de backjump (si le domaine $d_{x_j}(\mathcal{A}')$ est vide) sont correctes. □

Exemple 4.10 *Nous poursuivons l'exemple 4.3. Lorsque le solveur S_3 reçoit le nogood $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$, il supprime par filtrage c du domaine de x_3. Cette suppression sera annulée dès que le solveur S_3 développera l'instanciation $\{x_4 \leftarrow a, x_1 \leftarrow b\}$.*

4.2 Extensions et adaptations à d'autres algorithmes

4.2.1 Généralisation à un filtre quelconque

Le schéma et les notions que nous venons de décrire pour l'algorithme FC-NR peuvent être étendus à n'importe quel algorithme maintenant un certain niveau de consistance et utilisant du nogood recording.

D'abord, la notion de CSP induit peut être généralisée pour un filtre quelconque ϕ. Dès lors, toutes les définitions faisant intervenir cette notion peuvent être étendues. En particulier, on peut généraliser la FC-consistance en ϕ-consistance.

Ensuite, dans la phase d'interprétation, suivant le filtre ϕ choisi, il faut procéder à la propagation des suppressions, afin de maintenir le niveau de consistance. Par exemple, en employant l'algorithme MAC avec le nogood recording, la réception d'un nogood unaire donne lieu à une propagation de la suppression, alors que FC-NR lui se contente de supprimer la valeur.

4.2.2 Adaptations à d'autres schémas parallèles ou distribués

Nous discutons, dans cette partie, de plusieurs allégements potentiels de l'hypothèse concernant l'existence d'une mémoire partagée de taille suffisamment importante pour contenir l'instance CSP à résoudre, l'ensemble des nogoods trouvés et l'affectation courante de chaque solveur. Par contre, nous n'abordons pas la question d'un relâchement de l'hypothèse relative aux communications, car un allégement de cette hypothèse semble contraire au concept même de coopération.

Une première possibilité consiste à ne mémoriser en mémoire partagée qu'une partie de l'affectation courante de chaque solveur. Un tel allégement peut avoir un sens quand, pour chaque solveur, l'accès en écriture à la mémoire partagée se révèle nettement plus coûteux qu'un accès à sa mémoire privée (i.e. la mémoire accessible seulement à ce solveur). Dans un tel cas, mettre à jour l'affectation courante en mémoire partagée après chaque modification peut s'avérer pénalisant pour l'efficacité de la méthode. Aussi, un solveur ne maintiendra en mémoire partagée que l'affectation des variables les moins profondes de son arbre de recherche. Le choix de cette profondeur limite

dépend alors de l'algorithme employé et du problème à résoudre. Intuitivement, les nogoods portant sur les variables situées au début de l'arbre de recherche sont les plus intéressants car ils ont un grand pouvoir de coupe. De plus, on peut penser qu'une fois dépassée une certaine profondeur, le solveur va rencontrer rapidement soit une solution, soit un échec. Dans ce dernier cas, recevoir un nogood annonçant cet échec n'a qu'un intérêt limité puisque le nombre de nœuds économisés est a priori faible. Par conséquent, il semble judicieux de limiter ainsi les mises à jour. Notons que cette stratégie, employée avec un schéma comme S_{light} ou S_{gest}, permet de réduire également le nombre de communications.

Dans le cas extrême où nous ne disposons pas de mémoire partagée, il faut se placer dans le cadre d'un schéma distribué. Dans un tel schéma, chaque solveur possède sa propre copie de l'instance CSP à résoudre. Il doit également détenir son propre ensemble de nogoods. Aussi, il n'est plus question de communiquer à un solveur que les nogoods qui lui seront utiles immédiatement. En effet, en l'absence de la mémoire partagée, le solveur n'aurait aucune connaissance des nogoods existants en vue d'un usage ultérieur. Par conséquent, dans un schéma distribué, il est nécessaire qu'un solveur qui découvre un nogood informe tous les autres solveurs de sa découverte. Bien sûr, les doublons ne sont communiqués qu'une seule fois si possible. Ce type de schéma trouve son utilité, par exemple, lorsqu'on souhaite exploiter un cluster de PCs pour résoudre un problème.

4.3 Étude expérimentale

Nous allons, à présent, mener une étude expérimentale afin de déterminer d'une part si l'échange de nogoods est une forme efficace de coopération et d'autre part si l'approche coopérative est une alternative intéressante à l'approche énumérative classique. Dans un premier temps, nous évaluerons la qualité de la méthode coopérative d'un point de vue parallèle en étudiant entre autres l'efficacité obtenue en pratique lors de la résolution d'instances aléatoires. Puis, nous nous comparerons l'approche coopérative aux principaux algorithmes séquentiels employés pour la résolution de CSPs. Enfin, nous nous intéresserons au comportement de la méthode coopérative sur des instances du monde réel.

4.3.1 Implémentation et protocole expérimental

4.3.1.1 Implémentation

Algorithmes implémentés
Pour les besoins des différentes études comparatives, nous avons implémenté plusieurs méthodes :

- la méthode concurrente coopérative pour les schémas S_0, S_{light} et S_{gest},
- une version concurrente sans coopération,
- les algorithmes classiques FC [HE80], FC-CBJ [Pro93], MAC [SF94], et FC-NR [SV93, SV94].

L'implémentation de la méthode concurrente coopérative est basée sur les *pthreads*. Les pthreads (et plus généralement les threads) présentent l'avantage d'utiliser une mémoire partagée. Un pthread est associé à chaque solveur. De même, un pthread est associé au gestionnaire de nogoods dans S_{gest}. Ces pthreads sont exécutés en parallèle par le système d'exploitation jusqu'à la résolution du problème (découverte d'une solution ou de l'inconsistance). Lorsqu'un des solveurs résout le problème, tous les pthreads terminent immédiatement leur exécution. Dans la mesure où nos expériences sont effectuées sur une machine monoprocesseur, il s'agit en fait de pseudo-parallélisme. L'exécution des pthreads par le système fait alors appel à la notion d'ordonnancement des processus dans les systèmes d'exploitation multitâches. En pratique, la politique d'ordonnancement correspond à une variante de Round-Robin.
Les communications entre deux solveurs sont réalisées par des messages envoyés d'un pthread à

l'autre.

La méthode concurrente sans coopération consiste à lancer les solveurs dotés exactement des mêmes heuristiques que la méthode coopérative. La seule différence réside dans l'absence de tout échange d'informations que ce soit par des messages ou par le biais de la mémoire partagée. Ainsi, chaque solveur possède son propre exemplaire du problème à résoudre et ajoute localement les nogoods qu'il trouve à cet exemplaire. La recherche se termine dès qu'un des solveurs résout le problème. L'implémentation repose également sur des pthreads. Elle est similaire à celle de la méthode coopérative, hormis le fait que chaque pthread utilise sa propre copie du problème à résoudre, et non un exemplaire commun à tous les pthreads.

Concernant les algorithmes classiques, nous employons l'algorithme AC-2001 [BR01] pour établir et maintenir la consistance d'arc dans MAC. Pour l'algorithme FC-NR, nous limitons l'arité des nogoods mémorisés à 2.

Les heuristiques employées
Afin de parcourir des arbres de recherche distincts, nous devons garantir que les heuristiques employées pour ordonner les variables et/ou les valeurs sont différentes. Si ces heuristiques doivent être différentes, elles doivent cependant rester proches les unes des autres afin de favoriser la coopération. Il apparaît également nécessaire qu'elles soient aussi efficaces les unes que les autres sous peine de voir toujours le même solveur résoudre le problème, et ce sans aucun apport de la coopération. Hélas, il existe peu d'heuristiques efficaces. C'est pourquoi nous nous limitons aux heuristiques dom/deg et dom/st pour ordonner les variables. Pour les valeurs, nous utilisons soit l'ordre d'apparition des valeurs dans le domaine, soit l'ordre inverse. En combinant les heuristiques d'ordonnancement des variables et celles d'ordonnancement des valeurs, nous ne pouvons produire que quatre solveurs distincts. Pour éviter ce problème, nous fixons le choix de la première variable de chaque solveur, puis nous utilisons pour les autres variables une des heuristiques dom/deg ou dom/st. Enfin, pour favoriser l'échange de nogoods, nous imposons que deux solveurs commencent l'énumération avec la même variable, mais en ordonnant les valeurs l'un dans leur ordre d'apparition, l'autre dans l'ordre inverse.

Comme nous l'avons dit précédemment, la méthode concurrente sans coopération emploie exactement les mêmes heuristiques que la méthode coopérative.

Pour ordonner les variables, les algorithmes FC, FC-CBJ et MAC emploient l'heuristique dom/deg tandis que l'algorithme FC-NR utilise l'heuristique dom/st. Le choix de dom/st pour FC-NR s'explique par les meilleurs résultats obtenus par cette heuristique par rapport à dom/deg. Pour les valeurs, aucune heuristique particulière n'est utilisée. Les valeurs sont simplement considérées dans leur ordre d'apparition.

4.3.1.2 Protocole expérimental

Au niveau matériel, les expérimentations ont été réalisées :

- sur un PC sous Linux équipé d'un processeur Athlon XP 1800+ d'AMD et de 512 Mo de mémoire vive, pour les problèmes aléatoires classiques,
- sur un PC sous Linux équipé d'un processeur Pentium III 550 MHz d'Intel et de 256 Mo de mémoire, pour les instances du monde réel.

Les méthodes concurrentes avec et sans coopération ne sont pas déterministes à cause de la concurrence et des échanges d'informations. Aussi, nous résolvons chaque instance quinze fois afin de réduire l'impact du non-déterminisme sur la qualité des résultats. Les résultats pour une instance

correspondent donc à la moyenne des résultats obtenus lors de ces quinze résolutions. Pour une résolution, les résultats considérés sont ceux du solveur qui résout le problème le premier. Pour les problèmes aléatoires classiques, les résultats fournis correspondent aux moyennes des résultats obtenus en résolvant cent problèmes par classe. Pour les instances réelles, nous présentons les résultats instance par instance.

On procède de même pour les algorithmes classiques si ce n'est que chaque instance n'est résolue qu'une et une seule fois.

Enfin, précisons que les tests de contraintes tiennent compte :

- des contraintes initiales et des contraintes ajoutées dans le cas de la méthode coopérative, de la méthode concurrente sans coopération et de l'algorithme FC-NR,
- des contraintes initiales pour tous les autres algorithmes.

4.3.2 Efficacité de l'approche coopérative

4.3.2.1 Accélérations et efficacités

Dans cette partie, nous présentons les accélérations et les efficacités obtenues pour la méthode concurrente avec coopération pour les schémas S_0, S_{light} et S_{gest}. Pour le calcul des accélérations et des efficacités, nous considérons que nous disposons d'un processeur par solveur, même si, en pratique, l'ordinateur utilisé pour les tests ne dispose que d'un seul processeur. L'efficacité correspond donc au rapport $\frac{T_1}{p\ T_p}$ avec p le nombre de solveurs et T_1 (resp. T_p) le temps mis par un solveur (resp. p solveurs) pour résoudre un problème (ou un ensemble de problèmes). Notons que cette pratique ne fausse nullement les résultats. En effet, d'une part, les solveurs s'arrêtent dès que l'un d'entre eux a trouvé une solution ou prouvé l'inconsistance. Par conséquent, la somme des temps mis par les p solveurs est quasiment égale à p fois le temps mis par le solveur qui résout le problème. D'autre part, pour le schéma S_{gest}, le temps d'exécution du gestionnaire de nogoods se révèle négligeable par rapport au temps de résolution des solveurs. En pratique, selon la méthode employée pour calculer l'efficacité, nous obtenons une différence d'efficacité qui est généralement de quelques millièmes et au plus de deux centièmes.

Le tableau 4.1 présente l'efficacité obtenue par la méthode concurrente coopérative pour les trois schémas S_0, S_{light} et S_{gest}, pour différentes classes de CSPs aléatoires classiques (pour des problèmes consistants et inconsistants). Le tableau 4.2 (respectivement le tableau 4.3) détaille ces résultats pour les problèmes consistants (resp. inconsistants).
Nous constatons d'abord que les trois schémas obtiennent des efficacités similaires. Ensuite, à partir du tableau 4.1, nous observons que la méthode coopérative obtient une accélération linéaire ou superlinéaire sur trois des cinq classes jusqu'à dix solveurs. Pour les deux autres classes (à savoir (50,15,245,93) et (50,25,123,439)), l'accélération est tantôt linéaire ou superlinéaire, tantôt sublinéaire suivant le nombre de solveurs employés. Si on distingue les problèmes suivant leur consistance, on observe que notre méthode est, en général, plus efficace sur les problèmes consistants. En effet, d'une part, l'efficacité pour les problèmes consistants s'avère plus importante que pour les problèmes inconsistants (l'efficacité pour la classe (50,25,150,397) avec deux solveurs dans S_{gest} étant la seule exception). D'autre part, pour les problèmes consistants, l'accélération est linéaire ou superlinéaire dans la grande majorité des cas, tandis que pour les problèmes inconsistants, le bilan est plus mitigé. Pour les classes (50,15,245,93) et (50,25,123,439), l'accélération devient sublinéaire à partir de quatre ou six solveurs. Pour les autres classes, elle reste linéaire ou superlinéaire jusqu'à dix solveurs. Enfin, nous notons une diminution de l'efficacité avec l'augmentation du nombre de solveurs, pour les problèmes consistants, comme pour les inconsistants. Cette diminution est un phénomène classique pour les méthodes parallèles.

Classe (n, d, m, t)	Schéma	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1,207	1,130	1,150	1,105	1,048
	S_{light}	1,213	1,131	1,146	1,090	1,053
	S_{gest}	1,235	1,159	1,167	1,104	1,064
(50,15,245,93)	S_0	1,101	1,082	1,004	0,931	0,846
	S_{light}	1,099	1,085	1,008	0,934	0,851
	S_{gest}	1,133	1,094	1,018	0,942	0,849
(50,25,123,439)	S_0	1,112	1,073	0,939	1,010	0,932
	S_{light}	1,121	1,093	0,966	1,041	0,973
	S_{gest}	1,219	1,160	1,015	1,046	1,022
(50,25,150,397)	S_0	1,085	1,404	1,373	1,330	1,233
	S_{light}	1,101	1,408	1,354	1,324	1,240
	S_{gest}	1,030	1,381	1,353	1,321	1,227
(75,10,277,43)	S_0	1,339	1,319	1,285	1,218	1,183
	S_{light}	1,334	1,325	1,324	1,192	1,174
	S_{gest}	1,325	1,318	1,298	1,186	1,161

TAB. 4.1 : Efficacité obtenue par S_0, S_{light} et S_{gest} pour les problèmes consistants et inconsistants.

Classe (n, d, m, t)	Schéma	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1,304	1,174	1,228	1,132	1,084
	S_{light}	1,313	1,167	1,220	1,108	1,087
	S_{gest}	1,356	1,219	1,267	1,162	1,135
(50,15,245,93)	S_0	1,485	1,401	1,053	0,984	0,842
	S_{light}	1,488	1,398	1,057	0,977	0,843
	S_{gest}	1,537	1,425	1,080	1,012	0,853
(50,25,123,439)	S_0	1,236	1,189	0,941	1,093	1,001
	S_{light}	1,240	1,211	0,969	1,124	1,044
	S_{gest}	1,403	1,314	1,076	1,230	1,175
(50,25,150,397)	S_0	1,115	1,647	1,553	1,571	1,388
	S_{light}	1,132	1,657	1,523	1,570	1,395
	S_{gest}	0,980	1,517	1,441	1,473	1,309
(75,10,277,43)	S_0	1,368	1,444	1,368	1,298	1,258
	S_{light}	1,363	1,450	1,433	1,267	1,249
	S_{gest}	1,350	1,463	1,392	1,267	1,244

TAB. 4.2 : Efficacité obtenu par S_0, S_{light} et S_{gest} pour les problèmes consistants.

4.3.2.2 Origines des gains

À présent, nous nous intéressons aux origines des gains. Il existe deux principales raisons possibles : la concurrence et la coopération. Afin de déterminer la part de la concurrence et celle de la coopération dans la qualité des résultats, nous comparons les efficacités obtenues par les méthodes concurrentes avec coopération et sans coopération. Le tableau 4.4 présente l'efficacité obtenue par la méthode concurrente dépourvue de coopération pour différentes classes de CSPs aléatoires classiques (pour des problèmes consistants et inconsistants). Le tableau 4.5 (respectivement le tableau 4.6) détaille ces résultats pour les problèmes consistants (resp. inconsistants).

On constate d'abord que la version coopérative est quasiment toujours meilleure que la version concurrente. Les seules exceptions apparaissent pour les problèmes consistants. Pour de tels problèmes, l'efficacité de la version coopérative est, en général, soit voisine soit supérieure à celle de la version concurrente sans coopération. Cela signifie que la qualité des résultats, pour les problèmes consistants, provient en grande partie de la concurrence. Cependant, à plusieurs reprises, la méthode coopérative s'avère meilleure, ce qui implique que l'échange de nogoods participe également à la qualité des résultats obtenus. Par contre, pour les problèmes inconsistants, on observe que la méthode concurrente sans coopération n'obtient jamais d'accélérations linéaires ou superlinéaires. Autrement dit, les résultats obtenus sur les problèmes inconsistants proviennent en majeure partie de la coopération. Pour les problèmes consistants, l'apport de la coopération est moindre par rapport à celui pour les problèmes inconsistants, car la recherche s'arrête dès qu'une solution est trouvée. Notons que l'apport de la coopération est dû aussi bien à l'échange de nogoods qu'à la phase d'interprétation.

Il faut aussi souligner le rôle prépondérant des heuristiques de choix de valeurs employées, en particulier pour les problèmes inconsistants. Pour chaque solveur s (sauf un si le nombre de solveurs est impair), il existe un solveur qui débute son énumération par la même variable que s et qui exploite l'heuristique inverse de celle de s pour le choix des valeurs. Sans échange de nogoods, ces deux solveurs parcourent des arbres de recherche voisins. Avec l'échange de nogoods, chacun ne visite plus qu'une partie de leur arbre, grâce en partie aux nogoods trouvés par l'autre.

Nous nous focalisons maintenant sur les causes possibles de la diminution de l'efficacité. Avec des schémas comme les nôtres, une cause courante de la perte d'efficacité est l'importance du coût des communications. Notre méthode ne fait pas exception. Mais, dans notre cas, il existe une autre cause qui explique la chute des performances. En effet, la méthode concurrente sans coopération voit également son efficacité diminuer quand le nombre de solveurs augmente. Or, les solveurs ne diffèrent les uns des autres que par les heuristiques qu'ils emploient pour ordonner les variables et/ou les valeurs. Il semble hélas que la diversité de ces heuristiques ne soit pas suffisante pour garantir de bons résultats quand le nombre de solveurs augmente. Ce phénomène est particulièrement visible pour les classes (50,15,245,93) et (50,25,123,439). Pour ces deux classes, la méthode concurrente se révèle moins performante que pour les autres classes, en particulier au niveau des problèmes consistants. Apparemment, quand on emploie la méthode coopérative, la dégradation de performance engendrée par la concurrence n'est pas suffisamment compensée par la coopération. On obtient alors des accélérations sublinéaires.

4.3.2.3 Nombre de messages échangés

Dans ce paragraphe, nous comparons les nombres de messages échangés par la méthode concurrente coopérative pour les schémas S_0, S_{light} et S_{gest}. Les tableaux 4.7 et 4.8 présentent le nombre de messages échangés respectivement pour les nogoods unaires et binaires pour les problèmes consistants et inconsistants. Nous ne détaillons pas ces résultats suivant la consistance des problèmes, car la tendance observée est la même que pour l'ensemble des problèmes. Notons simplement que dans le schéma S_{gest}, le nombre de messages émis quand on exploite un seul solveur est non nul,

Classe (n, d, m, t)	Schéma	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1,128	1,091	1,085	1,079	1,017
	S_{light}	1,132	1,099	1,084	1,073	1,022
	S_{gest}	1,134	1,106	1,083	1,053	1,003
(50,15,245,93)	S_0	1,024	1,014	0,991	0,917	0,847
	S_{light}	1,021	1,018	0,995	0,923	0,853
	S_{gest}	1,050	1,023	1,001	0,923	0,848
(50,25,123,439)	S_0	1,016	0,982	0,936	0,943	0,875
	S_{light}	1,028	1,000	0,963	0,974	0,915
	S_{gest}	1,082	1,041	0,962	0,913	0,906
(50,25,150,397)	S_0	1,048	1,174	1,190	1,104	1,075
	S_{light}	1,062	1,174	1,180	1,096	1,081
	S_{gest}	1,094	1,250	1,263	1,179	1,143
(75,10,277,43)	S_0	1,277	1,101	1,130	1,066	1,042
	S_{light}	1,271	1,107	1,128	1,051	1,031
	S_{gest}	1,270	1,075	1,124	1,034	1,006

TAB. 4.3 : Efficacité obtenue par S_0, S_{light} et S_{gest} pour les problèmes inconsistants.

Classe (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	0,851	0,538	0,424	0,349	0,292
(50,15,245,93)	0,663	0,408	0,289	0,223	0,182
(50,25,123,439)	0,764	0,495	0,385	0,331	0,271
(50,25,150,397)	0,786	0,753	0,584	0,475	0,423
(75,10,277,43)	1,116	0,779	0,627	0,522	0,441

TAB. 4.4 : Efficacité obtenue par la méthode concurrente pour les problèmes consistants et inconsistants.

Classe (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	1,395	1,140	1,113	1,043	1,080
(50,15,245,93)	1,574	1,316	0,969	0,854	0,721
(50,25,123,439)	1,262	1,105	0,797	0,931	0,802
(50,25,150,397)	1,052	1,654	1,456	1,526	1,428
(75,10,277,43)	1,362	1,248	1,238	1,181	1,107

TAB. 4.5 : Efficacité obtenue par la méthode concurrente pour les problèmes consistants.

à cause des échanges avec le gestionnaire de nogoods.

Globalement, pour les nogoods unaires, on constate que le nombre de messages échangés est similaire dans les schémas S_0 et S_{light}. Le schéma S_{gest} produit en général autant de messages que S_0 et S_{light}. Cependant, dans quelques cas, moins de messages sont émis dans S_{gest}, car le nombre de nogoods unaires produits dans S_{gest} s'avère plus faible que dans les deux autres schémas. La similitude des résultats pour les trois schémas s'explique essentiellement par la faible sélection opérée par la limitation des communications des schémas S_{light} et S_{gest}. En effet, dans ces deux schémas, un solveur envoie les nogoods unaires qu'il découvre à tous les autres solveurs. Seuls les doublons ne sont envoyés qu'une seule fois. Or, bien qu'il augmente avec le nombre de solveurs, le nombre de doublons pour les nogoods unaires reste toujours très faible. Il en résulte donc une économie négligeable en nombre de messages émis.

Pour les nogoods binaires, le nombre de messages échangés est significativement plus important que pour les nogoods unaires, tout simplement car les solveurs produisent nettement plus de nogoods binaires que de nogoods unaires. Le schéma S_{light} se révèle alors le meilleur, suivi de S_{gest}, puis de S_0. Or, le schéma S_{light} ne diffère de S_0 qu'au niveau des échanges de nogoods. Dans S_{light}, nous restreignons les communications au strict nécessaire, en n'envoyant les nogoods binaires qu'à une partie des solveurs. Il est de même dans S_{gest}. Aussi, l'écart existant entre les schémas S_0 et S_{light} (ou entre S_0 et S_{gest}) illustre la contribution de notre limitation des communications. Grâce à cette limitation, le nombre de communications est significativement plus faible dans S_{light} ou dans S_{gest} que dans S_0. On peut ainsi espérer que le nombre de solveurs à partir duquel le coût des communications pénalise l'efficacité soit plus important dans un schéma comme S_{light} ou S_{gest} que dans un schéma comme S_0.
Concernant la différence de résultats obtenus entre les schémas S_{light} et S_{gest}, elle est imputable aux communications systématiques de tous les nogoods au gestionnaire. En particulier, parmi toutes ces communications, un certain nombre correspond à des communications de doublons. En effet, le nombre de doublons est beaucoup plus important (d'un facteur 10 à 100) dans S_{gest} que dans S_{light}. L'importance de ce nombre de doublons est due au laps de temps qui s'écoule entre la découverte d'un nogood et son ajout à l'ensemble des nogoods en mémoire partagée. En fait, ce laps de temps est plus long dans S_{gest} que dans S_{light} car l'ajout des nogoods est délégué au gestionnaire par les solveurs.
Enfin, si nous étudions de plus près les résultats obtenus pour S_{light}, nous constatons que le nombre de messages binaires s'avère moins important que le nombre de nogoods binaires découverts. Il en résulte donc que dans la majorité des cas, un nogood binaire n'est utile immédiatement qu'à un petit nombre de solveurs, voire même à aucun solveur. Il en est de même dans S_{gest}. La limitation des communications employée dans S_{light} et S_{gest} permet donc aux solveurs de réduire le temps passé à gérer les communications (en particulier dans la réception de nogoods qui ne sont pas immédiatement utiles). Les solveurs peuvent alors se consacrer pleinement à la résolution du problème.

4.3.2.4 Résumé

Pour les trois schémas S_0, S_{light} et S_{gest}, la méthode concurrente coopérative obtient des accélérations linéaires ou superlinéaires jusqu'à dix solveurs pour trois des cinq classes de problèmes traitées. Pour les deux autres classes, les accélérations sont soit linéaires ou superlinéaires, soit sublinéaires selon le nombre de solveurs exploités. Globalement, nous avons constaté que l'efficacité de la méthode était meilleure pour les problèmes consistants que pour ceux inconsistants. Cependant, nous obtenons tout de même des accélérations linéaires ou superlinéaires pour les problèmes inconsistants de certaines classes. La qualité de ces résultats est due :

- essentiellement à la coopération pour les problèmes inconsistants,

Classe (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	0,621	0,358	0,267	0,214	0,173
(50,15,245,93)	0,566	0,339	0,239	0,183	0,149
(50,25,123,439)	0,556	0,325	0,259	0,206	0,167
(50,25,150,397)	0,589	0,438	0,326	0,249	0,219
(75,10,277,43)	0,789	0,419	0,295	0,229	0,186

TAB. 4.6 : Efficacité obtenue par la méthode concurrente pour les problèmes inconsistants.

Classe (n, d, m, t)	Schéma	p					
		1	2	4	6	8	10
(50,15,184,112)	S_0	0	2,84	21,09	47,99	74,67	100,99
	S_{light}	0	2,82	20,88	47,42	74,54	99,84
	S_{gest}	0,12	0,33	20,40	44,15	67,13	88,91
(50,15,245,93)	S_0	0	4,61	30,35	80,77	142,57	211,41
	S_{light}	0	4,56	29,98	79,87	140,87	207,51
	S_{gest}	0	0,05	28,22	80,12	138,84	203,52
(50,25,123,439)	S_0	0	8,66	47,81	103,35	161,37	224,54
	S_{light}	0	8,67	47,90	103,91	156,87	227,27
	S_{gest}	1,94	4,23	38,57	82,24	126,00	176,69
(50,25,150,397)	S_0	0	5,53	43,32	95,10	155,14	221,99
	S_{light}	0	5,50	43,05	94,42	154,30	222,60
	S_{gest}	0,30	0,67	43,37	89,96	146,34	208,76
(75,10,277,43)	S_0	0	1,70	13,75	31,51	52,39	72,76
	S_{light}	0	1,69	13,70	30,99	51,14	70,83
	S_{gest}	0,40	0,94	14,85	32,06	50,78	68,04

TAB. 4.7 : Nombre total de messages échangés pour les nogoods unaires dans S_0, S_{light} et S_{gest} pour les problèmes consistants et inconsistants.

- à la concurrence et en moindre partie à la coopération pour les problèmes consistants.

Nous avons également noté une diminution de l'efficacité avec l'augmentation du nombre de solveurs. Cette diminution est causée par un manque de diversité des solveurs (en particulier au niveau de leurs heuristiques de choix de variables et/ou de valeurs). Au niveau des échanges de nogoods, nous avons constaté que les nogoods binaires n'étaient immédiatement utiles qu'à un petit nombre de solveurs. Ce résultat explique qu'on échange nettement moins de messages dans S_{light} que dans S_{gest} ou dans S_0 et montre l'intérêt de limiter ainsi les communications. Cependant, il semblerait que dans nos expérimentations, les échanges de messages s'avèrent peu coûteux puisque la méthode coopérative obtient des résultats similaires dans les trois schémas (en particulier au niveau du temps de résolution et de l'efficacité). Dans le cas où les échanges se révéleraient plus coûteux, le schéma à utiliser serait sans conteste S_{light}. C'est pourquoi, par la suite, toutes les comparaisons seront réalisées en exploitant ce schéma pour la méthode coopérative.

4.3.3 Comparaisons par rapport aux algorithmes classiques

4.3.3.1 Pour un système monoprocesseur

Dans un système monoprocesseur, nous devons tenir compte du travail de chaque solveur. Aussi, les résultats considérés pour la méthode coopérative correspondent à la somme des résultats obtenus par chaque solveur, tous les solveurs étant stoppés dès que l'un d'eux a résolu le problème. Les tableaux 4.9 et 4.10 présentent respectivement la somme des temps et la somme des nombres de tests pour la méthode coopérative (dotée de 1 à 4 solveurs), ainsi que le temps et le nombre de tests pour les algorithmes classiques, à savoir FC, FC-CBJ, FC-NR et MAC.

Nous constatons d'abord que la méthode coopérative avec un solveur obtient des résultats différents de ceux de l'algorithme FC-NR. Cela s'explique simplement par la différence d'heuristiques d'ordonnancement des variables, en particulier pour le choix de la première variable. Lorsque les heuristiques sont exactement les mêmes, y compris pour le choix de la première variable, FC-NR et la méthode coopérative avec un solveur obtiennent bien évidemment les mêmes résultats. On note également que FC-NR et la méthode coopérative avec un solveur effectuent généralement plus de tests de contraintes que FC, FC-CBJ et MAC. Ce résultat s'explique par les contraintes que FC-NR ajoute au problème durant la recherche. En effet, lors des tests de consistance, FC-NR vérifie indifféremment les contraintes initiales et les contraintes ajoutées tandis que les autres algorithmes classiques ne testent que les contraintes initiales. Ces tests additionnels se traduisent généralement par un surcoût en temps.

Par contre, dès que nous utilisons deux ou quatre solveurs, nous constatons que la méthode coopérative obtient des résultats soit équivalents, soit meilleurs que ceux de FC-NR. Les gains en temps varient de quelques pourcents à 21%. Ils sont essentiellement dus à la concurrence et la coopération qui permettent de réduire le nombre de tests de contraintes. L'amélioration est telle que pour certaines classes, la méthode coopérative peut alors se révéler plus rapide que FC-CBJ ou que MAC alors que, pour ces classes, FC-NR est plus lent que ces deux algorithmes. Toutefois, dans la plupart des cas, la méthode coopérative s'avère moins rapide que FC ou que FC-CBJ.

Si nous détaillons les résultats selon la consistance des problèmes, nous observons globalement la même tendance. Néanmoins, pour les problèmes consistants, la méthode coopérative avec deux ou quatre solveurs réalise, pour plusieurs classes, moins de tests de contraintes que FC (cf. tableau 4.12). Cependant, ces gains en nombre de tests ne se signifient pas systématiquement un gain en temps par rapport à FC (voir tableau 4.11). Notons qu'il n'est pas étonnant d'observer une tendance légèrement meilleure pour les problèmes consistants puisque la méthode coopérative se montre plus efficace pour de tels problèmes.

Classe (n, d, m, t)	Schéma	p					
		1	2	4	6	8	10
(50,15,184,112)	S_0	0	357	1 322	2 437	3 655	4 944
	S_{light}	0	16	77	149	211	295
	S_{gest}	462	487	652	807	955	1 060
(50,15,245,93)	S_0	0	208	948	1 968	3 308	4 970
	S_{light}	0	11	85	183	310	463
	S_{gest}	240	247	440	625	840	1 077
(50,25,123,439)	S_0	0	1 375	4 573	8 561	12 100	16 017
	S_{light}	0	40	145	284	348	497
	S_{gest}	1 708	1 806	2 140	2 671	3 013	3 153
(50,25,150,397)	S_0	0	1 364	4 669	8 583	12 634	17 349
	S_{light}	0	37	174	326	464	653
	S_{gest}	1 668	1 761	2 132	2 502	2 788	3 111
(75,10,277,43)	S_0	0	434	1 474	2 564	3 885	5 237
	S_{light}	0	26	107	181	289	387
	S_{gest}	612	665	831	982	1 143	1 374

TAB. 4.8 : Nombre total de messages échangés dans S_0, S_{light} et S_{gest} pour les nogoods binaires pour les problèmes consistants et inconsistants.

Classe (n, d, m, t)	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1 599	1 325	1 426	795	949	1 331	1 472
$(50, 15, 245, 93)$	12 711	11 567	11 724	7 072	8 462	11 758	18 612
$(50, 25, 123, 439)$	662	595	613	576	632	635	599
$(50, 25, 150, 397)$	7 164	6 513	5 099	3 462	4 046	6 191	5 848
$(75, 10, 277, 43)$	2 772	2 084	2 114	1 591	1 846	1 912	1 211

TAB. 4.9 : Temps de résolution (en ms) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants et inconsistants.

Classe (n, d, m, t)	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	11 246	9 234	10 045	7 347	7 089	9 984	4 750
$(50, 15, 245, 93)$	85 394	76 259	77 654	64 695	63 629	81 947	53 354
$(50, 25, 123, 439)$	5 122	4 669	4 888	5 969	5 208	5 479	2 601
$(50, 25, 150, 397)$	54 249	49 680	39 436	34 743	32 648	52 620	21 813
$(75, 10, 277, 43)$	17 297	12 820	13 089	12 279	11 406	12 496	3 533

TAB. 4.10 : Nombre de tests de contraintes (en milliers) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants et inconsistants.

4.3.3.2 Pour un système multiprocesseur

Dans un système multiprocesseur, les résultats considérés pour la méthode coopérative sont ceux du solveur qui résout le premier le problème. Faute de disposer d'un système doté de plusieurs processeurs, les résultats que nous allons présenter sont le fruit d'une simulation sur un système monoprocesseur. Les tableaux 4.13 et 4.14 présentent respectivement le temps et le nombre de tests pour la méthode coopérative (dotée de 1 à 4 solveurs) et pour les quatre algorithmes classiques. Comme nous obtenons la même tendance pour les problèmes consistants et pour les problèmes inconsistants, nous ne détaillons pas les résultats selon la nature des problèmes.

Si la méthode coopérative utilise un seul solveur, les résultats sont identiques à ceux obtenus dans un système monoprocesseur. Par contre, lorsque la méthode coopérative exploite deux ou quatre solveurs, elle réalise significativement moins de tests de contraintes que les quatre algorithmes classiques utilisés. Elle se révèle alors nettement plus rapide qu'eux. Nous obtenons des résultats similaires lorsqu'on augmente le nombre de solveurs. La qualité de ces résultats provient essentiellement de la bonne efficacité pratique de la méthode coopérative.

Les résultats que nous venons de présenter sont obtenus en simulant le parallélisme. Bien sûr, des expérimentations dans un vrai environnement parallèle devront être menées dans le futur, afin de confirmer les tendances observées. De plus, il serait intéressant également de comparer la méthode coopérative avec des versions parallèles de FC ou de MAC.

4.3.4 Comportement pour des instances du monde réel

Nous allons maintenant étudier le comportement de la méthode concurrente coopérative sur des instances du monde réel. Les problèmes considérés sont issus de l'archive FullRLFAP. Nous ne conservons de cette archive que les instances dont la résolution avec l'algorithme FC-NR requiert plus de 100 millisecondes. Les résultats présentés pour la méthode coopérative correspondent aux résultats d'une seule résolution. Ces résultats sont ceux du premier solveur qui résout le problème. Autrement dit, nous nous plaçons dans le cadre d'un système multiprocesseur, même si la machine utilisée pour les expérimentations ne dispose que d'un seul processeur. Nous imposons une limite pour le temps de résolution. Au delà de 15 minutes, la recherche est stoppée. Les tableaux 4.15 et 4.16 présentent respectivement le temps de résolution et le nombre de tests pour la méthode coopérative (dotée de 1 à 4 solveurs) et pour les quatre algorithmes classiques.

Pour la plupart des instances considérées (SCEN-01, SCEN-11 et les instances GRAPH), nous constatons que le temps de résolution de la méthode coopérative reste quasiment identique que nous utilisions un, deux ou quatre solveurs. Ce résultat est principalement dû à une coopération très limitée à cause du faible nombre de nogoods produits. De plus, pour ces instances, l'apport de la concurrence n'est pas très important. Aussi, à l'arrivée, les gains en temps et en nombre de tests sont insignifiants. Notons que toutes les instances sélectionnées (hormis SCEN-08) sont consistantes, ce qui explique en partie le faible nombre de nogoods. Comparée à FC-NR, la méthode coopérative obtient des résultats similaires.

Pour les instances SCEN-04 et SCEN-08, le nombre de nogoods produits est nettement plus important. La coopération alliée à la concurrence permet alors d'améliorer le temps de résolution quand le nombre de solveurs augmente. Toutefois, le gain n'est pas suffisant pour obtenir des accélérations linéaires et pour envisager de lancer leur résolution sur un système monoprocesseur (les temps cumulés des solveurs étant trop importants). Pour l'instance SCEN-11, le temps de résolution reste similaire pour un, deux ou quatre solveurs. Par contre, pour six solveurs, la méthode résout le problème en seulement 130 millisecondes (non présenté dans le tableau 4.15), ce qui permet d'obtenir une accélération superlinéaire. Ce résultat est dû uniquement à la concurrence, puisqu'aucun nogood n'est échangé. Le temps cumulé des six solveurs est de 770 millisecondes. Il se

Classe	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1 256	964	1 088	616	738	1 016	1 073
$(50, 15, 245, 93)$	8 023	5 398	5 746	4 767	5 708	7 622	12 147
$(50, 25, 123, 439)$	826	666	683	675	741	738	545
$(50, 25, 150, 397)$	7 293	6 448	4 410	3 130	3 663	5 619	4 693
$(75, 10, 277, 43)$	2 717	1 998	1 900	1 561	1 811	1 885	1 088

TAB. 4.11 : Temps de résolution (en ms) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants.

Classe	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	8 585	6 511	7 478	5 601	5 397	7 342	3 420
$(50, 15, 245, 93)$	53 200	34 602	37 021	43 236	42 481	51 955	34 613
$(50, 25, 123, 439)$	6 060	5 036	5 188	6 708	5 781	6 051	2 240
$(50, 25, 150, 397)$	54 224	48 267	33 372	30 732	28 772	46 519	17 088
$(75, 10, 277, 43)$	16 757	12 166	11 672	11 919	11 049	12 215	3 139

TAB. 4.12 : Nombre de tests de contraintes (en milliers) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants.

Classe	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1 599	659	353	795	949	1 331	1 472
$(50, 15, 245, 93)$	12 711	5 781	2 928	7 072	8 462	11 758	18 612
$(50, 25, 123, 439)$	662	295	151	576	632	635	599
$(50, 25, 150, 397)$	7 164	3 254	1 272	3 462	4 046	6 191	5 848
$(75, 10, 277, 43)$	2 772	1 039	523	1 591	1 846	1 912	1 211

TAB. 4.13 : Temps de résolution pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants et inconsistants.

Classe	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	11 246	4 593	2 507	7 347	7 089	9 984	4 750
$(50, 15, 245, 93)$	85 394	38 125	19 390	64 695	63 629	81 947	53 354
$(50, 25, 123, 439)$	5 122	2 339	1 236	5 969	5 208	5 479	2 601
$(50, 25, 150, 397)$	54 249	24 800	9 853	34 743	32 648	52 620	21 813
$(75, 10, 277, 43)$	17 297	6 391	3 253	12 279	11 406	12 496	3 533

TAB. 4.14 : Nombre de tests de contraintes (en milliers) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour les problèmes consistants et inconsistants.

révèle meilleur que les temps obtenus par les quatre algorithmes classiques. Enfin, pour l'instance SCEN-05, la méthode coopérative dotée de deux solveurs obtient des résultats proches de ceux obtenus avec un seul solveur. Pour quatre solveurs, la concurrence et la coopération permettent d'obtenir une amélioration considérable du temps de résolution. Le temps cumulé des quatre solveurs est de 1210 millisecondes. Pour cette instance, la méthode concurrente coopérative s'avère significativement plus rapide que les quatre algorithmes classiques, que nous utilisions un système monoprocesseur ou un système multiprocesseurs. L'économie en nombre de tests est également considérable.

En résumé, parmi les instances du monde réel considérées, certains problèmes ne se prêtent pas du tout à la résolution par une méthode coopérative avec échange de nogoods, puisque leur résolution produit peu de nogoods, voire même aucun. Bien sûr, pour de telles instances, les résultats obtenus ne sont pas intéressants. Pour d'autres instances comme SCEN-04 et SCEN-08, les gains apportés par la coopération et la concurrence existent, mais ne sont pas suffisants pour envisager une résolution sur un système monoprocesseur. Enfin, pour des instances comme SCEN-05 ou SCEN-11, les temps cumulés de tous les solveurs surclassent les temps obtenus par les quatre algorithmes classiques, autorisant par la même une résolution sur un système monoprocesseur. Ces résultats sont dus soit à la concurrence seule, soit à la concurrence et à la coopération suivant les problèmes. Enfin, notons que l'emploi d'heuristiques spécifiques à ces problèmes pourraient permettre une amélioration des résultats.

4.4 Conclusion

Dans [MV96], une méthode concurrente coopérative est présentée. Dans cette méthode, la coopération repose sur un échange de nogoods (i.e. d'affectations ne conduisant pas à une solution). Dans ce chapitre, nous avons poursuivi ce travail. Nous avons proposé d'abord trois schémas qui diffèrent les uns des autres au niveau des échanges de nogoods. En particulier, dans deux de ces schémas, nous avons tenté de restreindre au strict nécessaire les échanges de nogoods. Nous avons aussi ajouté à chaque solveur une phase d'interprétation qui permet de tirer le meilleur parti des nogoods reçus.

Nous avons d'abord expérimenté la méthode sur des instances aléatoires. Nous avons alors obtenu des résultats très satisfaisants avec des accélérations linéaires et superlinéaires pour des problèmes consistants comme pour des problèmes inconsistants. L'échange de nogoods apparaît donc être une forme efficace de coopération. Nous avons néanmoins constaté une diminution de l'efficacité avec l'augmentation du nombre de solveurs. Cette diminution entraîne, dans certains cas, l'apparition d'accélérations sublinéaires. Elle est principalement due à une diversité insuffisante au niveau des heuristiques des solveurs.

Comparée aux algorithmes classiques, la méthode coopérative se révèle souvent plus efficace que l'algorithme FC-NR et parfois même plus rapide que FC-CBJ ou MAC (principalement pour les problèmes consistants), dans le cas d'un système monoprocesseur. Par contre, pour un système multiprocesseur, la méthode coopérative apparaît meilleure dans la quasi-totalité des cas.

Pour les instances du monde réel, les résultats sont plus contrastés. Pour certains problèmes, le faible apport de la concurrence et l'absence d'informations à échanger rendent l'emploi de la méthode coopérative inintéressant. Pour d'autres instances, l'apport de la coopération et de la concurrence existe. Hélas, celui-ci n'est pas toujours suffisant pour obtenir des accélérations linéaires. Par contre, lorsque l'apport est significatif, les gains en temps sont tels que la méthode concurrente coopérative surclasse les quatre algorithmes classiques, que nous utilisions un ou plusieurs processeurs.

Instance	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	100	100	90	100	110	100	640
SCEN-04	170	110	50	120	50	170	90
SCEN-05	19 170	19 090	300	-	335 300	18 430	14 310
SCEN-08	120	80	20	20	20	120	270
SCEN-11	2 620	2 610	2610	12 740	1 250	2 930	25 630
GRAPH-08	110	90	90	70	60	110	430
GRAPH-09	120	110	110	100	110	130	670
GRAPH-10	690	690	690	-	-	680	930
GRAPH-14	110	100	100	100	110	120	530

TAB. 4.15 : Temps de résolution pour la méthode coopérative avec le schéma S_{light} et pour certaines instances de l'archive FullRLFAP.

Instance	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	176,3	176,3	176,3	185,4	185,4	176,3	1 857,7
SCEN-04	208,0	140,7	61,9	255,3	51,1	203,1	246,0
SCEN-05	31 259,2	31 199,4	630,0	-	829 057,7	31 251,1	9 220,9
SCEN-08	356,3	256,4	85,7	52,7	46,8	356,3	2 346,4
SCEN-11	5 459,7	5 459,7	5 459,7	32 095,2	2 828,3	5 459,7	22 520,8
GRAPH-08	204,2	183,7	183,7	158,4	158,4	204,2	1 251,8
GRAPH-09	191,0	190,8	190,8	199,2	199,2	191,0	1 819,7
GRAPH-10	1 500,3	1 500,1	1 500,1	-	-	1 498	2 531
GRAPH-14	174,0	173,9	173,0	183,4	183,4	174,0	1 599,2

TAB. 4.16 : Nombre de tests de contraintes (en milliers) pour la méthode coopérative avec le schéma S_{light} et pour les algorithmes classiques pour certaines instances de l'archive FullRLFAP.

La poursuite de ce travail nécessite d'abord des expérimentations sur une vraie machine parallèle afin de confirmer les tendances que nous avons observées. Dans le même temps, il peut s'avérer nécessaire de trouver de nouvelles heuristiques qui soient à la fois efficaces et suffisamment diverses pour améliorer l'efficacité ainsi que pour augmenter le nombre de solveurs. Ensuite, nous pouvons étendre la méthode en exploitant tout algorithme qui maintient un certain niveau de consistance, ou en utilisant plusieurs algorithmes différents (ce qui permettrait, par exemple, de mêler des recherches complètes et incomplètes à l'image de [HW93], ainsi que d'augmenter la diversité des solveurs). Il est également envisageable de l'adapter à d'autre forme de coopération avec échange d'informations. En particulier, dans le chapitre suivant, nous proposons un schéma coopératif basé sur l'échange de goods et de nogoods structurels (respectivement de nogoods structurels valués) et sur l'algorithme BTD (resp. BTD-val). Enfin, il semble naturel d'étendre ce travail au cadre des CSPs valués. Cette extension paraît d'autant plus réalisable qu'une partie du cadre de travail est déjà définie dans [DV96] (en particulier la notion de nogood valué et l'algorithme Nogood-Recording valué).

Chapitre 5

Hybridation BTD et recherche concurrente coopérative

Nous venons de voir, dans le chapitre 4, qu'une méthode concurrente avec une coopération basée sur l'échange de nogoods classiques pouvait obtenir de très bons résultats. Il paraît donc naturel de vouloir étendre ce travail aux goods et aux nogoods structurels définis dans le chapitre 2. Aussi, dans ce chapitre, nous présentons d'abord une méthode concurrente avec une coopération basée sur l'échange de goods et de nogoods structurels. Puis, dans un second temps, nous généralisons ce travail au cadre des CSPs valués.

5.1 Origines de l'approche

Nous avons observé, dans le chapitre 4, que l'échange de nogoods classiques se révélait être une forme efficace de coopération quand il est mis en œuvre au sein d'une recherche concurrente. Ces bons résultats constituent la motivation première pour l'extension de ce travail aux goods et aux nogoods structurels. Cependant, rien ne garantit que les résultats que nous obtiendrons seront de la même qualité que ceux observés pour la méthode concurrente avec échange de nogoods classiques. En effet, la méthode BTD diffère conceptuellement de l'algorithme FC-NR. De plus, les nogoods structurels sont de taille plus importante en général que les nogoods classiques. Leur pouvoir de coupe (et de réutilisation) semble donc moindre. Toutefois, leur emploi dans BTD n'est pas strictement identique à celui réalisé dans FC-NR, bien que similaire en plusieurs points. Quant aux goods structurels, ils présentent quelques propriétés intéressantes (par exemple la possibilité de réaliser des sauts en avant). Ainsi, à première vue, il semble difficile de juger de l'efficacité que pourrait avoir une telle approche. Aussi, il paraît intéressant et légitime de s'interroger sur l'efficacité de cette approche.

Une seconde motivation pour ce travail réside dans les différences observées dans les résultats suivant les choix réalisés. D'une part, ces choix portent sur la méthode employée pour calculer une décomposition arborescente (ou une approximation d'une décomposition arborescente). En effet, il existe plusieurs méthodes possibles donnant des décompositions de plus ou moins bonne qualité pour BTD. Nous avons ainsi pu constater que l'efficacité de BTD variait sensiblement suivant la qualité de la décomposition arborescente. D'autre part, durant le calcul de la décomposition arborescente, ces choix concernent également l'arborescence que nous allons utiliser par la suite. En effet, la décomposition fournit un arbre de clusters. Or, n'importe quel nœud de cet arbre peutt être employé comme racine de l'arborescence. De même, si on modifie l'ordre des fils d'un nœud donné, on obtient encore une arborescence différente. Autrement dit, à partir d'une décomposition donnée,

nous pouvons produire de nombreuses arborescences différentes. Bien évidemment, l'efficacité de BTD n'est pas la même suivant l'arborescence que nous employons. Notons que ce choix influe sur la numérotation compatible, et donc, par la suite, sur l'ordre d'énumération compatible. Là aussi, plusieurs ordres peuvent être construits à partir d'une numérotation. Or, l'ordre d'énumération peut se révéler crucial pour l'efficacité de BTD. Hélas, tous ces choix sont loin d'être aisés à faire. En effet, outre des paramètres comme la taille des cliques ou celle des séparateurs, le nombre de contraintes par cluster (ainsi que leur dureté), les propriétés de l'arborescence (équilibrée ou non, profonde ou non), ... sont autant de paramètres à prendre en compte pour choisir une décomposition, une numérotation et un ordre d'énumération. Devant la difficulté à effectuer ces choix, la concurrence peut permettre de diminuer le risque de faire un mauvais choix. Si, de plus, les différents choix sont effectués en respectant certaines conditions (voir section 5.2), on peut adjoindre à la concurrence une coopération basée sur un échange de goods et de nogoods structurels.

5.2 Conditions nécessaires et hypothèses de travail

Dans toute la suite de ce chapitre, nous supposons que $\mathcal{P} = (X, D, C, R)$ (respectivement $\mathcal{P} = (X, D, C, R, S, \phi)$ dans le cas des CSPs valués) soit l'instance à résoudre.
Notre objectif est de résoudre le problème en exploitant de la concurrence et en échangeant des goods et des nogoods structurels (resp. des nogoods structurels valués). Si exploiter de la concurrence ne pose aucun problème, il n'en est pas de même de l'échange de goods ou de nogoods. En effet, pour rendre possible un tel échange, certaines conditions sont requises. Les notions de good et de nogood structurels sont fondées essentiellement sur la notion de séparateur. Si nous utilisons deux décompositions arborescentes qui ne possèdent pas de séparateur commun, aucun échange ne pourra être réalisé, ou du moins tout échange sera inutile, car inexploitable. Cela nous conduit à définir la notion de compatibilité de deux décompositions.

Définition 5.1 (compatibilité de deux décompositions)
Soient $(\mathcal{C}, \mathcal{T})$ et $(\mathcal{C}', \mathcal{T}')$ deux décompositions arborescentes du graphe de contraintes (X, C).
Soit $Y \subseteq X$.
*Les décompositions $(\mathcal{C}, \mathcal{T})$ et $(\mathcal{C}', \mathcal{T}')$ sont dites **compatibles** par rapport à Y si :*

(i) il existe $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters de $\mathcal{C}$ tels que $\mathcal{C}_j$ soit le fils de $\mathcal{C}_i$ dans $\mathcal{T}$ et que $\mathcal{C}_i \cap \mathcal{C}_j = Y$,

(ii) il existe $\mathcal{C}'_k$ et $\mathcal{C}'_l$ deux clusters de $\mathcal{C}'$ tels que $\mathcal{C}'_l$ soit le fils de $\mathcal{C}'_k$ dans $\mathcal{T}'$ et que $\mathcal{C}'_k \cap \mathcal{C}'_l = Y$.

En d'autres termes, deux décompositions sont compatibles par rapport à Y, si Y est un séparateur du graphe de contraintes qui correspond à l'intersection de deux clusters dans chacune des deux décompositions. Notons que la condition pour la compatibilité de deux décompositions porte uniquement sur le séparateur et l'intersection entre deux clusters. Rien n'impose que les clusters dans une décomposition soient identiques à ceux de l'autre décomposition.

Exemple 5.2 *Nous reprenons l'exemple du graphe de contraintes de la figure 1.5 (page 26)et nous notons $(\mathcal{C}, \mathcal{T}_1)$ la décomposition arborescente calculée (voir figure 1.8 page 27). Durant la triangulation dont le résultat est présenté à la figure 1.6, nous avons choisi d'ajouter une arête joignant D et H et une liant M et N. À la place de ces deux arêtes, nous aurions aussi pu choisir d'ajouter une arête entre C et I et une entre L et O. Nous aurions alors obtenu la décomposition arborescente $(\mathcal{C}', \mathcal{T}')$ (voir figure 5.1).*
Les décompositions $(\mathcal{C}, \mathcal{T}_1)$ et $(\mathcal{C}', \mathcal{T}')$ sont compatibles entre autres, pour :

- *le séparateur CD, pour les clusters $\mathcal{C}_1$ et $\mathcal{C}_2$, et $\mathcal{C}'_1$ et $\mathcal{C}'_2$,*
- *le séparateur CD, pour les clusters $\mathcal{C}_1$ et $\mathcal{C}_4$, et $\mathcal{C}'_1$ et $\mathcal{C}'_4$,*
- *le séparateur HI, pour les clusters $\mathcal{C}_5$ et $\mathcal{C}_6$, et $\mathcal{C}'_5$ et $\mathcal{C}'_6$ (bien que $\mathcal{C}_5$ soit différent de $\mathcal{C}'_5$).*

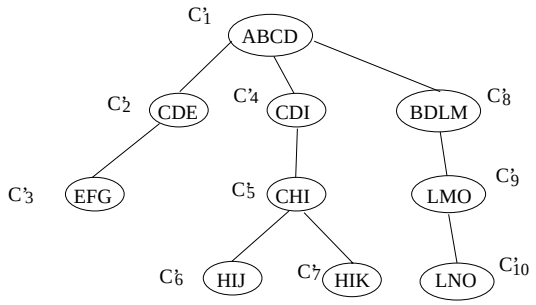

FIG. 5.1 : L'arborescence associée à la décomposition arborescente $(\mathcal{C}', \mathcal{T}')$ du graphe de la figure 1.6 (page 26).

Au niveau des numérotations compatibles, aucune condition particulière n'est requise. Du moment que les deux décompositions sont compatibles pour un séparateur Y, toute numération $N_{\mathcal{C}}$ (respectivement $N_{\mathcal{C}'}$) compatible avec $\mathcal{T}$ (resp. $\mathcal{T}$') convient. Il en est de même pour les ordres d'énumérations compatibles.

Parmi les différentes décompositions compatibles possibles, nous distinguons les décompositions parfaitement compatibles :

Définition 5.3 *Deux décompositions arborescentes $(\mathcal{C}, \mathcal{T})$ et $(\mathcal{C}', \mathcal{T}')$ sont dites parfaitement compatibles si*

(i) $\mathcal{C} = \mathcal{C}'$,

(ii) et, pour chaque cluster $\mathcal{C}_j$ fils de $\mathcal{C}_i$ dans $\mathcal{T}$, on a :

- *soit $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$ dans $\mathcal{T}'$,*
- *soit $\mathcal{C}_i$ est un fils de $\mathcal{C}_j$ dans $\mathcal{T}'$.*

En d'autres termes, deux décompositions parfaitement compatibles possèdent les mêmes clusters et la même arborescence, qui est simplement parcourue différemment en utilisant une racine différente et/ou un ordre différent sur les fils. Remarquons que ce cas particulier paraît offrir le cadre le plus propice à l'échange de goods et de nogoods, car les décompositions $(\mathcal{C}, \mathcal{T})$ et $(\mathcal{C}', \mathcal{T}')$ sont compatibles pour chacune des intersections $\mathcal{C}_i \cap \mathcal{C}_j$.

Exemple 5.4 *Si on reprend l'exemple précédent, les décompositions $(\mathcal{C}, \mathcal{T}_1)$ et $(\mathcal{C}', \mathcal{T}')$ ne sont pas parfaitement compatibles puisque $\mathcal{C}' \neq \mathcal{C}$. Considérons la décomposition arborescente $(\mathcal{C}'', \mathcal{T}_2)$ (voir figure 5.2) du graphe de contraintes de la figure 1.5 (page 26). Les décompositions $(\mathcal{C}, \mathcal{T}_1)$ et $(\mathcal{C}'', \mathcal{T}_2)$ sont parfaitement compatibles. En effet, nous avons $\mathcal{C}'' = \mathcal{C}$ et l'arbre $\mathcal{T}_2$ est identique à $\mathcal{T}_1$, si ce n'est que sa racine est $\mathcal{C}_4$ et non $\mathcal{C}_1$.*

5.3 Hybridation dans le cadre classique

5.3.1 Propriétés

Dans la suite du chapitre, afin d'améliorer la lisibilité et la clarté des propos, nous limitons notre étude au cas particulier des décompositions parfaitement compatibles. Néanmoins, les résultats présentés par la suite s'étendent sans difficulté au cas général. Nous exploiterons donc deux décompositions arborescentes parfaitement compatibles $(\mathcal{C}, \mathcal{T}_1)$ et $(\mathcal{C}, \mathcal{T}_2)$ du graphe de contraintes

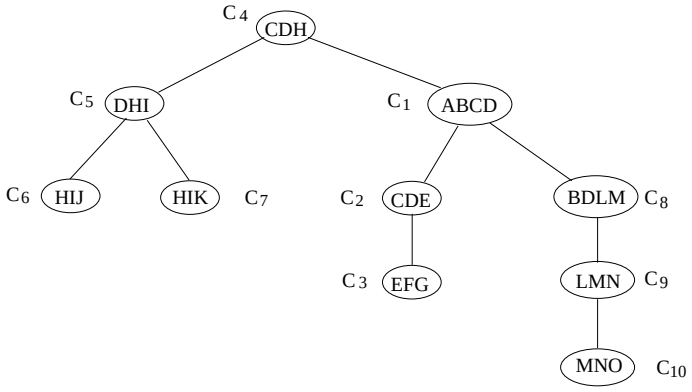

FIG. 5.2 : L'arborescence associée à la décomposition arborescente $(\mathcal{C}'', \mathcal{T}_2)$ du graphe de la figure 1.6 (page 26).

(X, C). L'emploi de deux décompositions différentes nous impose de modifier la notation des descendances. Nous notons $Desc_{\mathcal{T}}(\mathcal{C}_j)$ l'ensemble des variables appartenant à l'union des descendants $\mathcal{C}_k$ de $\mathcal{C}_j$ dans le sous-arbre de $\mathcal{T}$ enraciné en $\mathcal{C}_j$, $\mathcal{C}_j$ inclus. De même, nous désignons par $Fils_{\mathcal{T}}(\mathcal{C}_j)$ les clusters fils de $\mathcal{C}_j$ dans $\mathcal{T}$.

Exemple 5.5 *Si nous poursuivons l'exemple 5.4, nous avons* $Desc_{\mathcal{T}_1}(\mathcal{C}_4) = \{C, D, H, I, J, K\}$ *et* $Desc_{\mathcal{T}_2}(\mathcal{C}_4) = X$. *Nous obtenons aussi* $Desc_{\mathcal{T}_1}(\mathcal{C}_2) = Desc_{\mathcal{T}_2}(\mathcal{C}_2) = \{C, D, E, F, G\}$.

Comme nous le voyons dans l'exemple précédent, les descendances dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$ peuvent être identiques ou complètement différentes. Tout dépend de la relation père-fils dans l'arborescence de clusters, cette relation définissant une orientation de l'arborescence. Remarquons que la définition des goods et nogoods structurels (définition 2.4 (page 41)) tient naturellement compte de cette orientation. En effet, quand nous évoquons un good ou un nogood, nous précisons toujours "de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$" (avec $\mathcal{C}_j$ fils de $\mathcal{C}_i$).

Nous sommes maintenant en mesure de présenter les propriétés fondamentales sur lesquelles repose l'échange de goods et de nogoods structurels. La première de ces propriétés garantit la validité d'un tel échange.

Propriété 5.6 *Soient* $\mathcal{C}_i$ *et* $\mathcal{C}_j$ *deux clusters. Si* $\mathcal{C}_j$ *est un fils de* $\mathcal{C}_i$ *dans* $\mathcal{T}_1$ *et dans* $\mathcal{T}_2$, *alors les équivalences suivantes sont vérifiées :*

(i) g *good de* $\mathcal{C}_i/\mathcal{C}_j$ *dans* $T_1 \Leftrightarrow g$ *good de* $\mathcal{C}_i/\mathcal{C}_j$ *dans* T_2

(ii) n *nogood de* $\mathcal{C}_i/\mathcal{C}_j$ *dans* $T_1 \Leftrightarrow n$ *nogood de* $\mathcal{C}_i/\mathcal{C}_j$ *dans* T_2

Preuve :
Si $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$ dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$, les sous-arbres de $\mathcal{C}_i$ enracinés en $\mathcal{C}_j$ respectivement dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$ sont identiques. Par conséquent, les sous-problèmes correspondants le sont aussi. Donc, les équivalences (i) et (ii) sont vérifiées. □

En d'autres termes, on peut échanger les goods et les nogoods portant sur les sous-arbres communs de $\mathcal{T}_1$ et de $\mathcal{T}_2$. Cette propriété offre aussi la possibilité de limiter les échanges de goods et de nogoods structurels, en n'envoyant les goods et nogoods qu'à une partie des solveurs (voir paragraphe 5.3.2).

Nous ne faisons aucune distinction particulière entre les goods et les nogoods produits par le solveur lui-même et ceux reçus de la part d'autres solveurs. Leur emploi reste donc identique à

celui présenté dans le chapitre 2. Ainsi, les nogoods vont stopper immédiatement la recherche et provoquer un retour en arrière, alors que les goods permettront un forward-jump. Cependant, dans le cas des problèmes consistants, il est possible de tirer un profit supplémentaire des goods grâce à la propriété suivante :

Propriété 5.7 *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters tels que $\mathcal{C}_j$ soit un fils de $\mathcal{C}_i$ dans T_1 et $\mathcal{C}_i$ soit un fils de $\mathcal{C}_j$ dans T_2.*
Si l'affectation consistante g sur $\mathcal{C}_i \cap \mathcal{C}_j$ est un good de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1 et un good de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2, alors g peut être étendue en une affectation consistante sur X.

Preuve :
Si g est un good de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1, alors il existe une affectation consistante $\mathcal{A}_1$ sur $Desc_{T_1}(\mathcal{C}_j)$ telle que $\mathcal{A}_1[\mathcal{C}_i \cap \mathcal{C}_j] = g$. Si g est un good de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2, alors il existe une affectation consistante $\mathcal{A}_2$ sur $Desc_{T_2}(\mathcal{C}_j)$ telle que $\mathcal{A}_2[\mathcal{C}_i \cap \mathcal{C}_j] = g$. Donc $\mathcal{A}_1[\mathcal{C}_i \cap \mathcal{C}_j] = g = \mathcal{A}_2[\mathcal{C}_i \cap \mathcal{C}_j]$. D'après le lemme 2.5 (page 41), $\mathcal{A}_1$ et $\mathcal{A}_2$ sont compatibles. Or $Desc_{T_1}(\mathcal{C}_j) \cup Desc_{T_2}(\mathcal{C}_i) = X$. Donc, g peut être étendue en une affectation consistante $(\mathcal{A}_1 \cup \mathcal{A}_2)$ sur X. □

Autrement dit, nous pouvons conclure à la consistance d'un problème sans même qu'un solveur ait trouvé une solution. Il suffit simplement de trouver une affectation consistante g sur $\mathcal{C}_j \cap \mathcal{C}_i$ qui soit un good de $\mathcal{C}_i/\mathcal{C}_j$ et de $\mathcal{C}_j/\mathcal{C}_i$.

La propriété suivante rend possible "l'inversion" de l'orientation d'un nogood structurel. Dans cette propriété, il est important de constater que le nogood fruit de l'inversion de l'orientation n'est pas nécessairement un nogood au sens de la définition 2.4 (page 41). Toutefois, nous nous permettons de le nommer ainsi par abus de langage, puisque la consistance du problème n'est pas remise en cause (il s'agit tout de même d'un nogood au sens classique du terme).

Propriété 5.8 (inversion) *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters tels que $\mathcal{C}_j$ soit un fils de $\mathcal{C}_i$ dans T_1 et $\mathcal{C}_i$ soit un fils de $\mathcal{C}_j$ dans T_2.*
Si l'affectation consistante n sur $\mathcal{C}_i \cap \mathcal{C}_j$ est un nogood de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1, alors BTD peut exploiter le nogood n de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2.

Preuve : Par définition de n, il n'existe pas d'affectation consistante sur $Desc_{\mathcal{T}_1}(\mathcal{C}_j)$ qui contient n. Par conséquent, aucune affectation consistante sur X ne peut contenir n. Aussi, BTD peut exploiter le nogood n de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2. □

À première vue, cette propriété peut sembler inutile. Mais, il n'en est rien. Au contraire, dans certains cas, elle peut se révéler fort utile comme nous le voyons dans l'exemple suivant :

Exemple 5.9 *Reprenons les deux décompositions $(\mathcal{C}, \mathcal{T}_1)$ et $(\mathcal{C}, \mathcal{T}_2)$ de l'exemple 5.4 et supposons qu'un solveur exploitant $(\mathcal{C}, \mathcal{T}_1)$ produise un nogood n de $\mathcal{C}_1/\mathcal{C}_4$. Fournir le nogood n de $\mathcal{C}_4/\mathcal{C}_1$ à un solveur utilisant $(\mathcal{C}, \mathcal{T}_2)$ est inutile puisqu'une telle affectation n ne peut être atteinte. Par contre, si un solveur exploitant $(\mathcal{C}, \mathcal{T}_3)$ (voir figure 5.3) reçoit le nogood n de $\mathcal{C}_4/\mathcal{C}_1$, il évitera de visiter inutilement toute la descendance dans $\mathcal{T}_3$ de $\mathcal{C}_4$ enracinée en $\mathcal{C}_1$.*

5.3.2 Schéma coopératif

Nous allons, à présent, décrire un schéma coopératif pour l'échange de goods et de nogoods structurels.
Nous lançons p solveurs en concurrence, chaque solveur étant associé à un processus. La recherche se termine quand un des solveurs a trouvé une solution ou a établi l'inconsistance du problème. Tous ces solveurs utilisent l'algorithme BTD (ou une de ces variantes) et exploitent des décompositions

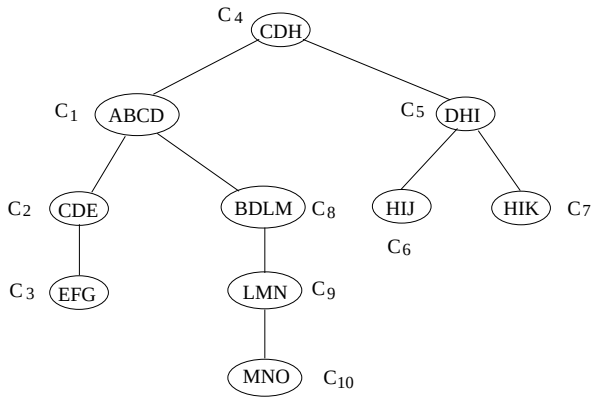

FIG. 5.3 : L'arborescence associée à la décomposition arborescente $(\mathcal{C}, \mathcal{T}_3)$ du graphe de la figure 1.6 (page 26).

arborescentes toutes parfaitement compatibles entre elles. L'exploitation de telles décompositions devrait permettre de favoriser les échanges par rapport à un schéma dans lequel les solveurs emploieraient des décompositions arborescentes quelconques. En effet, les décompositions sont alors compatibles deux à deux pour chaque intersection $\mathcal{C}_i \cap \mathcal{C}_j$, avec $\mathcal{C}_i$ fils ou père de $\mathcal{C}_j$. De plus, en procédant ainsi, nous pouvons nous contenter de ne calculer qu'une seule décomposition arborescente. Les autres décompositions sont alors déduites de celle-ci en modifiant la racine et/ou l'ordonnancement des fils d'un ou plusieurs nœuds.

Durant leur recherche, chaque solveur produit des goods et des nogoods structurels. La propriété 5.6 autorise l'échange entre deux solveurs des goods et des nogoods structurels qui portent sur un sous-arbre commun. Ensuite, la propriété 5.8 offre la possibilité d'inverser l'orientation des nogoods sans modifier la consistance du problème. Par conséquent, tout nogood produit doit être expédié à tous les solveurs. Par contre, lorsqu'un solveur produit un good de $\mathcal{C}_i/\mathcal{C}_j$, ce good ne doit être communiqué à un solveur que si $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$ dans la décomposition arborescente exploitée par ce solveur.
Réaliser ces échanges uniquement avec des communications ne semble pas être une solution envisageable, étant donné le nombre potentiellement important de goods et de nogoods. À l'image du travail décrit dans le chapitre 4, il apparaît préférable d'utiliser une structure commune à tous les solveurs, qui contiendrait l'ensemble des goods et des nogoods trouvés. Aussi, nous supposons que nous disposons d'une quantité de mémoire partagée suffisamment importante pour contenir :

- l'instance à résoudre,
- l'ensemble des goods et des nogoods produits,
- les données nécessaires pour limiter au minimum les communications.

Lorsqu'un solveur découvre un good ou un nogood, il l'ajoute à l'ensemble de goods et de nogoods déjà trouvés. Dans le cas d'un good de $\mathcal{C}_i/\mathcal{C}_j$, il teste préalablement s'il n'existe pas le good correspondant de $\mathcal{C}_j/\mathcal{C}_i$. Si ce good existe, il conclut directement à la consistance du problème grâce à la propriété 5.7. Après avoir mis à jour l'ensemble de goods et de nogoods, il prévient certains solveurs de sa découverte, en vertu des propriétés 5.6 et 5.8. Pour un nogood, la découverte est signalée à tous les solveurs tandis que pour un good de $\mathcal{C}_i/\mathcal{C}_j$, on n'annonce la nouvelle à un solveur que si, dans sa décomposition arborescente, $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$.
Parmi tous ces messages échangés, nombre d'entre eux se révèlent encore inutiles. En effet, un solveur n'a besoin d'être informé que des messages concernant la branche de l'arborescence de clusters

qu'il en est train de traiter. Lorsqu'ultérieurement il explorera une autre partie de l'arborescence, il pourra toujours exploiter les goods et nogoods déjà trouvés pour cette partie, en accédant à la mémoire partagée. Aussi, la découverte d'un good de $\mathcal{C}_i/\mathcal{C}_j$ n'est annoncée qu'à un solveur dont la branche en cours de traitement contient les clusters $\mathcal{C}_i$ et $\mathcal{C}_j$ et tel que dans la décomposition arborescente associée à ce solveur, $\mathcal{C}_i$ est le père de $\mathcal{C}_j$. Quant à la découverte d'un nogood de $\mathcal{C}_i/\mathcal{C}_j$, elle est signalée à tout solveur dont la branche en cours de traitement contient les clusters $\mathcal{C}_i$ et $\mathcal{C}_j$. Naturellement, l'information concernant la branche en cours de traitement de chaque solveur est stockée dans la mémoire partagée, pour la rendre accessible à tous. De plus, nous excluons de l'ensemble des destinataires du message le solveur qui découvre le good ou le nogood. Enfin, comme plusieurs solveurs sont susceptibles de découvrir en même temps le même good (ou nogood), les doublons ne sont communiqués qu'une et une seule fois.

Exemple 5.10 *Nous considérons trois solveurs S_1, S_2 et S_3 qui exploitent respectivement les décompositions arborescentes $(\mathcal{C}, \mathcal{T}_1)$, $(\mathcal{C}, \mathcal{T}_2)$ et $(\mathcal{C}, \mathcal{T}_3)$. Les trois solveurs utilisent BTD comme algorithme de résolution. Supposons que les branches en cours de traitement de S_1, S_2, S_3 soient respectivement $\{\mathcal{C}_1, \mathcal{C}_4, \mathcal{C}_5\}$, $\{\mathcal{C}_4, \mathcal{C}_1\}$ et $\{\mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6\}$.*
Il est alors inutile de communiquer à S_1 un good (ou un nogood) de $\mathcal{C}_2/\mathcal{C}_3$ ou de $\mathcal{C}_1/\mathcal{C}_8$, puisque ni l'un, ni l'autre ne permet à S_1 d'interrompre sa recherche en cours. Si par la suite, S_1 a besoin de ces goods il pourra les obtenir via l'ensemble de goods déjà trouvés qui se trouve en mémoire partagé.
Supposons maintenant que S_1 trouve un good de $\mathcal{C}_4/\mathcal{C}_5$. Alors, seul le solveur S_3 sera informé de cette découverte. Dans le cas où S_1 découvrirait un nogood de $\mathcal{C}_1/\mathcal{C}_4$, il communiquerait cette information à S_2.

Un raffinement de ce modèle est possible en considérant les affectations des séparateurs rencontrés sur la branche en cours de traitement. Le good ou nogood n'est transmis que s'il concorde avec l'affectation courante du séparateur correspondant. Cependant, ce raffinement est un peu plus coûteux en temps et en espace. En espace, pour chaque solveur, il est nécessaire de stocker en mémoire partagée l'affectation courante sur chaque séparateur de la branche en cours de traitement, afin que cette information soit accessible à tous les solveurs. Au niveau du temps, il faut comparer les deux affectations et mettre à jour régulièrement l'affectation courante des séparateurs. Le choix entre le modèle de base et ce raffinement dépend entre autres du coût d'un accès à la mémoire partagée, du coût des communications et du nombre de messages économisés en pratique grâce à ce raffinement.
Notons enfin que comme pour l'échange de nogoods classiques, une partie de ce travail de communication et de mise à jour peut être déléguée à un processus supplémentaire comme le gestionnaire de nogoods.

Pour terminer, nous intégrons à chaque solveur une phase d'interprétation. Durant cette phase, le solveur compare le good ou le nogood reçu à l'affectation courante du séparateur correspondant. Si l'affectation diffère ou si le séparateur n'est plus affecté (car l'information est arrivée trop tard), on ne tient pas compte de l'information. Par contre, dans le cas contraire, on peut stopper la recherche courante, car, grâce à l'information reçue, on connaît désormais la consistance du sous-problème correspondant. On se contente ensuite d'exploiter classiquement l'information. S'il s'agit d'un good, le solveur procède à un forward-jump. S'il s'agit d'un nogood, il revient en arrière et essaie une nouvelle affectation pour le séparateur concerné.

Exemple 5.11 *En reprenant l'exemple précédent, si S_3 reçoit de S_1 un good de $\mathcal{C}_4/\mathcal{C}_5$ et si ce good correspond à l'affectation courante de $\mathcal{C}_4 \cap \mathcal{C}_5$ dans S_3, alors S_3 va interrompre sa recherche. Il devrait ensuite réaliser un forward-jump en passant au prochain cluster qui n'appartient pas à la descendance de $\mathcal{C}_5$. Comme il n'existe pas de tel cluster, S_3 a donc trouvé une solution.*

Supposons à présent que S_2 reçoive de S_1 le nogood de $\mathcal{C}_1/\mathcal{C}_4$ qui correspond à l'affectation courante de $\mathcal{C}_4 \cap \mathcal{C}_1$ dans S_2. Dans un tel cas, S_2 interrompt sa recherche et essaie une nouvelle affectation pour $\mathcal{C}_4 \cap \mathcal{C}_1$.

5.4 Hybridation dans le cadre valué

5.4.1 Propriétés

Dans cette section, nous reprenons les notations de la section précédente et nous généralisons les résultats obtenus aux nogoods structurels valués du cadre VCSP. La notion de nogood structurel valué généralise celles de good et de nogood structurels du cadre classique. Aussi, nous n'avons plus que deux propriétés. La première garantit la validité des échanges de nogoods valués structurels.

Propriété 5.12 *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters. Si $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$ dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$, alors on a l'équivalence suivante :*

$$(n, v) \text{ nogood valué de } \mathcal{C}_i/\mathcal{C}_j \text{ dans } T_1 \Leftrightarrow (n, v) \text{ nogood valué de } \mathcal{C}_i/\mathcal{C}_j \text{ dans } T_2$$

Preuve :
Si $\mathcal{C}_j$ est un fils de $\mathcal{C}_i$ dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$, les sous-arbres de $\mathcal{C}_i$ enracinés en $\mathcal{C}_j$ respectivement dans $\mathcal{T}_1$ et dans $\mathcal{T}_2$ sont identiques. Par conséquent, les sous-problèmes correspondants le sont aussi. Donc, (n, v) nogood valué de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1. $\Leftrightarrow$ (n, v) nogood valué de $\mathcal{C}_i/\mathcal{C}_j$ dans T_2 □

Cette propriété autorise donc l'échange de nogoods structurels valués à condition qu'ils portent sur des sous-arbres communs de $\mathcal{T}_1$ et de $\mathcal{T}_2$. D'autre part, elle peut également permettre de restreindre le nombre de solveurs auxquels est envoyé un nogood donné (voir paragraphe 5.4.2).

Comme dans le cadre classique, aucune distinction n'est faite parmi les nogoods structurels valués. Qu'ils soient produits par le solveur ou reçus grâce à la coopération, leur utilisation est la même. Tous permettent donc de réaliser des forward-jumps. Toutefois, la propriété suivante rend possible une nouvelle exploitation de certains de ces nogoods :

Propriété 5.13 *Soient $\mathcal{C}_i$ et $\mathcal{C}_j$ deux clusters tels que $\mathcal{C}_j$ soit un fils de $\mathcal{C}_i$ dans T_1 et $\mathcal{C}_i$ soit un fils de $\mathcal{C}_j$ dans T_2. Si l'affectation n sur $\mathcal{C}_i \cap \mathcal{C}_j$ correspond à un nogood valué (n, v_1) de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1 et à un nogood valué (n, v_2) de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2, alors n peut être étendue en une affectation sur X de valuation $v_1 \oplus v_2 \oplus \bigoplus_{\substack{c \in C | c \subseteq \mathcal{C}_i \cap \mathcal{C}_j \\ et\ c\ viole\ n}} \phi(c)$.*

Preuve :
Si (n, v_1) est un nogood valué de $\mathcal{C}_i/\mathcal{C}_j$ dans T_1, alors il existe une affectation $\mathcal{A}_1$ sur $Desc_{T_1}(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ telle que $v_{\mathcal{P}_{n,\mathcal{C}_i/\mathcal{C}_j,T_1}}(\mathcal{A}_1) = v_1$. Si (n, v_2) est un nogood valué de $\mathcal{C}_j/\mathcal{C}_i$ dans T_2, alors il existe une affectation $\mathcal{A}_2$ sur $Desc_{T_2}(\mathcal{C}_i) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ telle que $v_{\mathcal{P}_{n,\mathcal{C}_j/\mathcal{C}_i,T_2}}(\mathcal{A}_2) = v_2$. Les ensembles de contraintes $E_{\mathcal{P},\mathcal{C}_j,T_1}$, $E_{\mathcal{P},\mathcal{C}_i,T_2}$ et $\{c \in C | c \subseteq \mathcal{C}_i \cap \mathcal{C}_j \text{ et } c \text{ viole } n\}$ sont disjoints. Donc, l'affectation complète $\mathcal{A}_1 \cup \mathcal{A}_2 \cup n$ a pour valuation $v_1 \oplus v_2 \oplus \bigoplus_{\substack{c \in C | c \subseteq \mathcal{C}_i \cap \mathcal{C}_j \\ et\ c\ viole\ n}} \phi(c)$. □

En d'autres termes, grâce à cette propriété, nous pouvons calculer la valuation d'une affectation complète et donc éventuellement fournir une nouvelle meilleure solution.

Concernant la propriété d'inversion de l'orientation d'un nogood (propriété 5.8), elle ne peut être généralisée au cadre des CSPs valués. En effet, il nous est impossible de déterminer la valuation optimale associée au nogood "inversé" sans résoudre le sous-problème formé par $Desc_{\mathcal{T}_2}(\mathcal{C}_i)$.

5.4.2 Schéma coopératif

Nous décrivons, dans cette partie, un schéma coopératif pour l'échange de nogoods structurels valués. Ce schéma reprend les principales idées du schéma coopératif pour l'échange de goods et de nogoods structurels auquel il rajoute une forme de coopération propre au cadre VCSP : l'échange des meilleures valeurs connues pour le majorant.
Nous lançons p solveurs en concurrence, chaque solveur étant associé à un processus. La recherche se termine dès qu'un des solveurs a déterminé la valuation optimale du problème. Tous ces solveurs utilisent l'algorithme BTD-val (ou une de ces variantes) et exploitent des décompositions arborescentes toutes parfaitement compatibles entre elles.

Durant leur recherche, chaque solveur produit des nogoods structurels valués, qu'il va échanger avec une partie de ces partenaires grâce à la propriété 5.12. Mais, pour des raisons identiques à celles du cadre classique, ces échanges ne sont pas tous réalisés via des communications. Nous estimons à nouveau préférable d'utiliser une structure commune à tous les solveurs, qui contiendrait l'ensemble des nogoods trouvés. Ainsi, nous supposons que nous disposons d'une quantité de mémoire partagée suffisamment importante pour contenir :

- l'instance à résoudre,
- l'ensemble des nogoods structurels valués produits,
- les données nécessaires pour limiter au minimum les communications et réduire l'espace de recherche à explorer.

Lorsqu'un solveur produit un nogood valué structurel de $\mathcal{C}_i/\mathcal{C}_j$, il l'ajoute à l'ensemble des nogoods déjà trouvés. Il informe ensuite une partie des solveurs de sa découverte, en exploitant la propriété 5.12. Cependant, tous les solveurs potentiellement concernés n'ont pas besoin de recevoir cette information. Seuls les solveurs dont la branche courante de l'arborescence de clusters contient $\mathcal{C}_i$ et $\mathcal{C}_j$ doivent être prévenus. Pour les autres solveurs concernés, ce nogood n'est pas nécessaire immédiatement. Il est donc inutile de les prévenir de son existence. Si, ultérieurement, ils ont besoin de ce nogood, ils pourront toujours l'exploiter via la mémoire partagée. Par conséquent, la découverte d'un nogood structurel valué de $\mathcal{C}_i/\mathcal{C}_j$ est signalée à tout solveur dont la branche en cours de traitement contient les clusters $\mathcal{C}_i$ et $\mathcal{C}_j$ et tel que, dans cette branche, $\mathcal{C}_i$ soit le père de $\mathcal{C}_j$. Comme précédemment, le solveur qui découvre le nogood est automatiquement exclu de la liste des destinataires de l'annonce et les doublons ne sont communiqués qu'une seule fois. Bien entendu, la branche en cours de traitement de chaque solveur est une information stockée dans la mémoire partagée, pour la rendre accessible à tous.
Comme dans le cadre classique, on peut raffiner le modèle en considérant les affectations des séparateurs rencontrés sur la branche en cours de traitement. Le nogood n'est alors transmis que si son affectation concorde avec l'affectation courante du séparateur correspondant. Notons également que comme pour l'échange de nogoods classiques, une partie de ce travail de communication et de mise à jour peut être déléguée à un processus supplémentaire comme le gestionnaire de nogoods.

Pour en terminer avec la description du schéma coopératif de base, nous présentons la phase d'interprétation que nous intégrons à chaque solveur. Durant cette phase, le solveur compare l'affectation du nogood reçu à l'affectation courante du séparateur correspondant. Si l'affectation diffère ou si le séparateur n'est plus affecté (i.e. l'information est arrivée trop tard), on ne tient pas compte de cette information. Par contre, dans le cas contraire, la recherche courante est arrêtée, car, grâce à l'information reçue, on connaît désormais la valuation optimale du sous-problème correspondant. On exploite ensuite classiquement le nogood reçu en procédant à un forward-jump.

Le schéma décrit ci-dessus est une simple adaptation du schéma présenté précédemment pour les goods et nogoods structurels du cadre classique. Nous nous proposons maintenant de discuter de quelques améliorations possibles propres au cadre des CSPs valués. Ces améliorations

se concrétisent par deux formes de coopération supplémentaires entre les solveurs. Leur emploi conduit naturellement vers un meilleur élagage de l'arbre de recherche de chaque solveur.

À l'instar des méthodes qui résolvent des problèmes d'optimisation en décomposant l'arbre de recherche, nous ajoutons à notre schéma un échange des meilleures valeurs connues pour le majorant. Dans notre cas, le majorant correspond à la valuation de la meilleure solution connue. Ainsi, chaque fois qu'un solveur trouve une meilleure solution, il va transmettre sa valuation à tous les autres solveurs. Notons qu'il n'est aucunement nécessaire de transmettre la solution proprement dite. Seule sa valuation nous intéresse. Il est également important de constater que cet échange s'effectue indépendamment du cluster racine. Autrement dit, deux solveurs ayant des clusters racines distincts peuvent s'échanger des valeurs pour le majorant. Cette forme de coopération paraît d'autant plus attrayante que grâce à la propriété 5.13, un solveur peut produire une nouvelle valeur pour le majorant. Dans le même temps, nous devons modifier la phase d'interprétation pour qu'elle tienne compte de ce type de messages. Quand une nouvelle valeur pour le majorant est reçue, le solveur doit mettre à jour son majorant $\alpha_{\mathcal{C}_r}$ (avec $\mathcal{C}_r$ la racine de son arborescence de clusters). Dans le cas de BTD-val (ou de FC-BTD-val), on peut souhaiter vérifier si l'affectation des variables de la branche courante a une valuation qui dépasse la nouvelle borne. Si c'est le cas, il est possible de backjumper. Pour des algorithmes comme BTD-val$_+$ (ou FC-BTD-val$_+$), ce travail est directement pris en compte par les minorant et majorant globaux, et donc aucun travail additionnel n'est nécessaire de la part de la phase d'interprétation.

Une autre amélioration peut consister à échanger des majorants locaux. Si plusieurs solveurs explorent en même temps un même sous-problème, ils peuvent s'informer mutuellement des nouvelles valeurs qu'ils découvrent pour le majorant local (i.e. la valuation de la meilleure solution connue pour le sous-problème en question). La phase d'interprétation doit être à nouveau modifiée afin de tenir compte de ces messages. Quand un solveur réceptionne un tel message, il met d'abord à jour le majorant local correspondant. Puis, si le minorant local correspondant dépasse la nouvelle valeur du majorant, il revient en arrière autant que nécessaire. Notons que pour pouvoir déterminer quels autres solveurs explorent le même sous-problème que lui, un solveur doit avoir accès à l'affectation courante sur chaque séparateur de la branche courante de chacun des autres solveurs. Cette information doit donc se trouver en mémoire partagée.

5.5 Conclusion

Dans ce chapitre, nous avons étendu le travail réalisé dans le chapitre 4 à deux autres formes de coopération : l'échange de goods et de nogoods structurels d'une part, l'échange de nogoods structurels valués d'autre part. La première de ces extensions consiste donc à lancer en concurrence plusieurs solveurs exploitant l'algorithme BTD (ou une de ses variantes). Ces solveurs échangent alors entre eux les goods et nogoods structurels qu'ils produisent. Pour favoriser cette coopération, il semble préférable que les solveurs emploient des décompositions arborescentes parfaitement compatibles entre elles. Naturellement, nous avons ensuite proposé un schéma coopératif similaire dans le cadre des CSPs valués, où les goods et nogoods structurels sont remplacés par des nogoods structurels valués. De plus, dans ce cadre là, il est possible d'échanger, dans le même temps, des valeurs de majorants locaux ou globaux afin d'augmenter la force de l'élagage. Bien sûr, la poursuite de ce travail requiert d'étudier expérimentalement ces deux schémas coopératifs afin de pouvoir juger de leur intérêt pratique.

Conclusion

La résolution des CSPs fait généralement appel à des recherches arborescentes exploitant des améliorations du backtracking (comme le filtrage ou le retour arrière intelligent). De telles méthodes obtiennent souvent des résultats satisfaisants en pratique. Cependant, leur complexité en temps est bornée par la taille de l'espace de recherche, qui est exponentielle. Par contre, les méthodes qui fournissent de meilleures bornes de complexité en temps - et qui sont, en général, basées sur une décomposition arborescente du graphe de contraintes du CSP - n'ont toujours pas prouvé leur efficacité en pratique. Dans le chapitre 2, notre première contribution a consisté à définir une nouvelle méthode (nommé BTD pour Backtracking sur Tree-Decomposition) pour résoudre les CSPs. L'algorithme BTD repose à la fois sur des techniques de backtracking et sur la notion de décomposition arborescente du graphe de contraintes.

Un des principaux intérêts de la méthode BTD est qu'elle bénéficie des avantages des deux approches, à savoir : l'efficacité pratique des algorithmes énumératifs et la garantie de complexités en temps et en espace limitées des méthodes structurelles. En effet, nous avons prouvé que les complexités théoriques en temps et en espace de BTD sont égales aux meilleurs résultats connus, c'est-à-dire en $O(n.s^2.m.d^{w^+ +1})$ pour le temps et en $O(n.s.d^s)$ pour l'espace (avec n le nombre de variables, m le nombre de contraintes, d la taille du plus grand domaine, $w^+ + 1$ la taille du plus grand cluster et s la taille de la plus grande intersection entre deux clusters).

En pratique, BTD peut être associé à tout filtrage ne modifiant pas la structure du problème. Nous avons expérimenté la méthode sur différentes types d'instances. Pour des problèmes dépourvus de structure, BTD est aussi efficace que les algorithmes classiques (voire même meilleur). Pour les problèmes structurés, BTD affiche un gain significatif grâce à l'exploitation des goods et des nogoods. Enfin, un des résultats les plus intéressants est que BTD requiert peu de mémoire, contrairement à l'algorithme Tree-Clustering [DP89] qui se révèle inexploitable en pratique, car nécessitant trop de mémoire.

Une première extension de BTD consiste en une généralisation aux CSPs n-aires (qui est en fait triviale). Ensuite, la comparaison théorique entre BTD et BT (respectivement FC-BTD et FC ou MAC-BTD et MAC) devra être étendue. En particulier, il faudra s'affranchir de l'ordre statique unique utilisé pour les variables, afin de prendre compte des ordres quelconques et différents selon les algorithmes.

Ensuite, dans le chapitre 3, nous avons étendu la méthode BTD au cadre des CSPs valués. Cette extension se traduit par la définition d'un cadre formel (en particulier de la notion de nogoods valués structurels) et des algorithmes BTD-val et FC-BTD-val basés respectivement sur l'algorithme Branch and Bound et Forward-checking valué. Nous avons également présenté des améliorations possibles de BTD-val et FC-BTD-val, comme l'algorithme BTD-val$_+$. Nous avons établi qu'en théorie, ces versions améliorées de BTD-val et de FC-BTD-val produisent moins de nœuds que BB et FC-val respectivement. Cependant, une étude expérimentale devra être menée pour pouvoir juger des gains obtenus en pratique, et plus généralement pour étudier le comportement de nos différents algorithmes. Comme pour BTD, les complexités sont en $O(ns^2md^{w^+ +1})$ pour le temps et en $O(nsd^s)$ pour l'espace avec $w^+ + 1$ la taille du plus grand cluster et s la taille de la plus grande intersection

entre deux clusters.

Parmi les extensions possibles de ce travail, nous envisageons l'emploi de méthodes plus performantes que BB ou FC-val comme algorithme de base. En particulier, l'utilisation d'algorithmes comme l'algorithme des poupées russes ou des algorithmes exploitant de la consistance d'arc directionnelle semble être une extension naturelle par rapport à la notion d'ordre d'énumération compatible employée par notre méthode.

Dans le cadre des CSPs classiques comme dans celui des CSPs valués, la méthode BTD s'appuie sur l'exploitation d'informations explicitées durant la recherche (goods et nogoods structurels ou nogoods structurels valués). Il en est de même de la seconde partie de ce travail. Cependant, le type d'informations explicitées diffère. Il s'agit de nogoods au sens classique du terme. Ces nogoods sont alors exploitées pour mettre en œuvre une coopération entre différents solveurs.

Dans [MV96], une méthode concurrente avec échange de nogoods est présentée. Dans le chapitre 4, nous avons poursuivi ce travail, en proposant d'abord trois schémas qui diffèrent les uns des autres au niveau des échanges de nogoods. En particulier, deux de ces schémas limitent au strict nécessaire les échanges de nogoods. Nous avons aussi intégré à chaque solveur une phase d'interprétation afin d'exploiter au mieux les nogoods reçus.

Nous avons ensuite expérimenté la méthode sur des instances aléatoires classiques et des instances du monde réel. Pour les instances aléatoires, nous avons alors obtenu des résultats très satisfaisants avec des accélérations linéaires et superlinéaires pour des problèmes consistants comme pour des problèmes inconsistants. L'échange de nogoods apparaît donc être une forme efficace de coopération. Nous avons néanmoins constaté une diminution de l'efficacité avec l'augmentation du nombre de solveurs. Cette diminution entraîne, dans certains cas, l'apparition d'accélérations sublinéaires. Elle est principalement due à une diversité insuffisante au niveau des heuristiques des solveurs. Comparée aux algorithmes classiques, la méthode coopérative se révèle souvent plus efficace que l'algorithme FC-NR, et plus rarement plus rapide que FC-CBJ ou MAC, dans le cas d'un système monoprocesseur. Par contre, pour un système multiprocesseur, la méthode coopérative apparaît meilleure dans la quasi-totalité des cas. Pour les instances du monde réel, les résultats sont plus contrastés. Certains problèmes se révelent peu propices à l'emploi d'une telle méthode. Pour d'autres instances, l'apport de la coopération et de la concurrence existe. Hélas, celui-ci n'est pas toujours suffisant pour obtenir des accélérations linéaires. Par contre, lorsque l'apport est significatif, les gains en temps sont tels que la méthode concurrente coopérative surclasse les quatre algorithmes classiques, que nous utilisions un ou plusieurs processeurs. Globalement, le résultat expérimental le plus intéressant est que dans certains cas, une implémentation monoprocesseur de la méthode concurrente coopérative peut se révéler plus rapide qu'un algorithme classique.

La poursuite de ce travail nécessite d'abord des expérimentations sur une vraie machine parallèle afin de confirmer les tendances que nous avons déjà observées. Dans le même temps, il peut s'avérer nécessaire de définir de nouvelles heuristiques qui soient à la fois performantes et suffisamment diverses pour améliorer l'efficacité ainsi que pour augmenter le nombre de solveurs. Ensuite, il semble naturel d'étendre ce travail au cadre des CSPs valués (la notion de nogood valué et l'algorithme Nogood-Recording valué sont déjà définies dans [DV96]). Par la suite, nous pouvons aussi étendre la méthode en exploitant tout algorithme qui maintient un certain niveau de consistance, ou en équipant les solveurs d'algorithmes différents. Il est également envisageable de l'adapter à d'autre forme de coopération avec échange d'informations.

Dans le chapitre 5, nous avons proposé une des extensions possibles de la méthode concurrente coopérative du chapitre 4. Cette extension consiste à employer deux autres formes de coopération : l'échange de goods et de nogoods structurels d'une part, l'échange de nogoods structurels valués d'autre part. La première de ces extensions consiste donc à lancer en concurrence plusieurs solveurs exploitant l'algorithme BTD (ou une de ses variantes). Ces solveurs échangent alors entre eux les goods et nogoods structurels qu'ils produisent. Pour favoriser cette coopération, il semble préférable

que les solveurs emploient des décompositions arborescentes parfaitement compatibles entre elles. Naturellement, nous avons ensuite proposé un schéma coopératif similaire dans le cadre des CSPs valués, où les goods et nogoods structurels sont remplacés par des nogoods structurels valués. De plus, dans ce cadre là, il est possible d'échanger, dans le même temps, des valeurs de majorants locaux ou globaux afin d'augmenter la force de l'élagage. Bien sûr, avant même d'envisager toute extension de ce travail, il sera nécessaire de valider expérimentalement les deux schémas coopératifs proposés.

Bibliographie

[ACP87] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embedding in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[Bac00] F. Bacchus. Extending Forward Checking. In *Proceedings of the 6th CP (CP'2000)*, pages 35–51, 2000.

[BCS01a] C. Bessière, A. Chmeiss, and L. Saïs. Heuristiques multi-niveaux pour ordonner les variables dans les CSP. In $7^{ème}$ *Journées Nationales Résolution Pratique des Problèmes NP-Complets*, pages 41–60, Toulouse, France, 2001. JNPC'2001.

[BCS01b] C. Bessière, A. Chmeiss, and L. Saïs. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'01*, pages 565–569, Paphos, Cyprus, 2001.

[Ber87a] C. Berge. *Graphes*. Gauthier-Villars, 1987.

[Ber87b] C. Berge. *Hypergraphes – Combinatoire des ensembles finis*. Gauthier-Villars, 1987.

[Ber99] A. Berry. A Wide-Range Efficient Algorithm for Minimal Triangulation. In *Proceedings of SODA'99 SIAM Conference*, january 1999.

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[BFM+96] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs : Basic properties and comparison. *LNCS*, 1106, 1996.

[BG01] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal clique trees. *Artificial Intelligence*, 125 :3–17, 2001.

[BM94] R. J. Bayardo and D. P. Miranker. An optimal backtrack algorithm for tree-structered constraint satisfaction problems. *Artificial Intelligence*, 71 :159–181, 1994.

[BM95] R. J. Bayardo and D. P. Miranker. On the Space-Time Trade-off in Solving Constraint Satisfaction Problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 558–562, 1995.

[BM96] R. J. Bayardo and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraints Satisfaction Problem. In *Proceedings of 13th National Conference on Artificial Intelligence*, pages 298–304, 1996.

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence*, Montréal, Canada, 1995.

[BP00] R. Bayardo and J. Pehoushek. Counting Models using Connected Components. In *Proceedings of AAAI 2000*, pages 157–162, 2000.

[BR96] C. Bessière and J.-C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. In *Proceedings of CP 96*, pages 61–75, 1996.

[BR01] C. Bessière and J.-C. Régin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 309–315, 2001.

[BS96] M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver – Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17 :381–400, 1996.

[BT01] J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 291–296, 2001.

[CA95] J. Conrad and D. Agrawal. Asynchronous Parallel Arc Consistency Algorithms on a Distributed Memory Machine. *Journal of Parallel and Distributed Computing*, 24(1) :27–40, 1995.

[CBB91] J. Conrad, D. Bahler, and J. Bowen. Static Parallel Arc Consistency. In Z. Ras and M. Zemankova (eds), editors, *Proceedings of the sixth International Symposium of Methodologies for Intelligent Systems*, volume 542 of *Lecture Notes in Artificial Intelligence*, pages 500–509. Springer-Verlag, 1991.

[CdGL+99] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

[CHH91] S. Clearwater, B. Huberman, and T. Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254 :1181–1183, Nov. 1991.

[CHH92] S. Clearwater, T. Hogg, and B. Huberman. Cooperative Problem Solving. In B. Huberman, editor, *Computation : The Micro and the Macro View*, World Scientific, pages 33–70. 1992.

[Chm96] A. Chmeiss. Sur la consistance de chemin et ses formes partielles. In *Actes du Congrès AFCET-RFIA 96*, pages 212–219, Rennes, France, 1996.

[CJ98] A. Chmeiss and P. Jégou. Efficient Path-Consistency Propagation. *International Journal of Artificial Intelligence Tools*, 7 :121–142, 1998.

[CN98] B. Choueiry and G. Noubir. A Disjunctive Decomposition Scheme for Discrete Contraint Satisfaction Problems Using Complete No-Good Sets. Technical Report KSL-98-23, Knowledge Systems Laboratory, Stanford University, 1998.

[Coo89] M. Cooper. An Optimal k-Consistency Algorithm. *Artificial Intelligence*, 41, 1989.

[CS92] P. Cooper and M. Swain. Arc consistency : parallelism and domain dependence. *Artificial Intelligence*, 58 :207–235, 1992.

[CS02] M. Cooper and T. Schiex. Arc consistency for soft constraints. *submitted to JACM*, 2002. See arXiv.org/abs/cs.AI/0111038.

[CvB01] X. Chen and P. van Beek. Conflict-Directed Backjumping Revisited. *Journal of Artificial Intelligence Research*, 14 :53–81, 2001.

[Dag97] P. Dago. *"Extensions d'algorithmes dans le cadre des problèmes de satisfaction de contraintes valués : application à l'ordonnancement de systèmes satellitaires"*. PhD thesis, ENSAE, Toulouse (France), Juin 1997.

[DB01] R. Debruyne and C. Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[dBKT95] A. de Bruin, G. Kindervater, and H. Trienekens. Asynchronous Parallel Branch-and-Bound and Anomalies. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Dec90] R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[Dec92] R. Dechter. Constraint Networks. In *Encyclopedia of Artificial Intelligence*, volume 1, pages 276–285. John Wiley & Sons, Inc., second edition, 1992.

[DF01] R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125 :93–118, 2001.

[DF02] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136 :147–188, 2002.

[DFP93] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 1131–1136, 1993.

[DM89] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. In *Proceedings of the tenth International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 271–277, Detroit, Michigan, August 1989.

[DP89] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[DV96] P. Dago and G. Verfaillie. Nogood Recording for Valued Constraint Satisfaction Problems. In *Proceedings of ICTAI 96*, pages 132–139, 1996.

[FD94] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.

[FD95] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 572–578, Montréal, Canada, 1995.

[FL93] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probabilistic approach. In *Proceedings of ECSQARU'93*, volume 747 of *LNCS*, pages 97–104. 1993.

[FLMCS95] H. Fargier, J. Lang, R. Martin-Clouaire, and T. Schiex. A constraint satisfaction framework for decision under uncertainty. In *Proceedings of the 11^{th} International Conference on Uncertainty in Artificial Intelligence*, 1995.

[FLS93] H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in Fuzzy Constraint Satisfaction Problems. In *Proceedings of the first European Congress on Fuzzy and Intelligent Technologies (EUFIT'93)*, 1993.

[FQ85] E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, 1985.

[Fre78] E. Freuder. Synthesizing constraint expressions. *CACM*, 21(11) :958–966, 1978.

[Fre82] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 :24–32, 1982.

[Fre85] E. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 :755–761, 1985.

[Fre89] E. Freuder. Partial constraint satisfaction. In *Proceedings of the eleventh International Joint Conference on Artificial Intelligence*, pages 278–283, Detroit, MI, 1989.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[Gas79] J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[GBR96] R. Génisson, B. Benhamou, and A. Rauzy. Une version concurrente de la procédure de Davis et Putnam. *Revue d'intelligence artificielle*, 10 :499–506, Avril 1996.

[GHW02] G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 161–165, 2002.

[Gin93] M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GJC94] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66 :57–89, 1994.

[GLS99] G. Gottlob, N. Leone, and F. Scarcello. On Tractable Queries and Constraints. In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS-99)*, pages 21–32, 1999.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[Gol80] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[HE80] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[HH93] T. Hogg and B. Huberman. *Better Than The Best : The Power of Cooperation*, pages 164–184. SFI 1992 Lectures in Complex Systems. Addison-Wesley, 1993.

[HKS00] Z. Habbas, M. Krajecki, and D. Singer. Domain Decomposition for Parallel Resolution of Constraint Satisfaction Problems with OpenMP. In *Proceedings of 2nd. European Workshop on OpenMP*. EWOMP 2000, 2000.

[HL88] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36 :125–130, 1988.

[HW93] T. Hogg and C.P. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proceedings of AAAI 93*, pages 231–236, 1993.

[HW94] T. Hogg and C.P. Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. In *Proceedings of AAAI 94*, pages 331–336, Seattle, WA, 1994.

[JDB00] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP'2000)*, pages 249–261, 2000.

[Jég90] P. Jégou. Cyclic-Clustering : a compromise between Tree-Clustering and the Cycle-Cutset method for improving search efficiency. In *Proceedings of European Conference on Artificial Intelligence (ECAI-90)*, pages 369–371, 1990.

[Jég91] P. Jégou. *Contribution à l'étude des problèmes de satisfaction de contraintes : Algorithmes de propagation et de résolution – Propagation de contraintes dans les réseau dynamiques*. PhD thesis, Université des Sciences et Techniques du Languedoc, Janvier 1991.

[Jég93] P. Jégou. Decomposition of Domains Based on the Micro-Structure of Finite Constraint Satisfaction Problems. In *Proceedings of AAAI 93*, pages 731–736, Washingtown, DC, 1993.

[JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing Satz Using Dynamic Workload Balancing. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier Science Publishers, 2001.

[JT02] P. Jégou and C. Terrioux. Recherche arborescente bornée. In $8^{ème}$ *Journées Nationales Résolution Pratique des Problèmes NP-Complets*, pages 127–141, Nice, 2002. JNPC'2002.

[JT03] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 2003. À paraître.

[Kas89] S. Kasif. Parallel Solutions to Constraint Satisfaction Problems. In R. Brachman, H. Levesque, and R. Reiter, editors, *KR'89 : Principles of Knowledge Representation and Reasoning*, pages 180–188. Morgan Kaufmann, San Mateo, California, 1989.

[Kas90] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3) :275–286, 1990.

[KD94] S. Kasif and A. Delcher. Local consistency in parallel constraint satisfaction networks. *Artificial Intelligence*, 69 :307–327, 1994.

[Kos99] A. Koster. *Frequency Assignment - Models and Algorithms*. PhD thesis, University of Maastricht, Novembre 1999.

[KR87] V. Kumar and N. Rao. Parallel depth-first search. Part II : Analysys. *International Journal of Parallel Programming*, 16 :501–519, 1987.

[KvB97] G. Kondrak and P. van Beek. A Theorical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89 :365–387, 1997.

[Lar00] J. Larrosa. Boosting Search with Variable Elimination. In *Proceedings of the 6^{th} CP*, 2000.

[Lar02] J. Larrosa. On arc and node consistency. In *Proceedings of AAAI'2002*, Edmonton (Canada), 2002.

[Lau93] P. Laursen. Simple approaches to parallel Branch and Bound. *Parallel Computing*, 19 :143–152, 1993.

[LM96] J. Larrosa and P. Meseguer. Exploiting the use of DAC in Max-CSP. In *Proceedings of the 2^{nd} CP*, pages 308–322, 1996.

[LMS99] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1) :149–163, 1999.

[LMS02] J. Larrosa, P. Meseguer, and M. Sánchez. Pseudo-Tree Search with Soft Constraints. In *Proceedings of the 15^{th} European Conference on Artificial Intelligence*, pages 131–135, Lyon (France), 2002.

[Mac77] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8 :99–118, 1977.

[Mér98] P.-P. Mérel. *Les problèmes de satisfaction de contraintes : recherche n-aire et parallélisme - Application au placement en CAO*. PhD thesis, Université de Metz, Février 1998.

[MH86] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28, 1986.

[Mon74] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Artificial Intelligence*, 7 :95–132, 1974.

[MS00] P. Meseguer and M. Sánchez. Tree-based Russian Doll Search. In *Proceedings of Workshop on soft constraint*. CP'2000, 2000.

[MS01] P. Meseguer and M. Sánchez. Specializing Russian Doll Search. In *Proceedings of the 7th CP*, volume LNCS 2239, pages 464–478, 2001.

[MV96] D. Martinez and G. Verfaillie. Echange de Nogoods pour la résolution coopérative de problèmes de satisfaction de contraintes. In $2^{ème}$ *Conférence Nationale sur la Résolution de Problèmes NP-Complets*, pages 261–274, Dijon, France, 1996. CNPC 96.

[PNB96] N. Prcovic, B. Neveu, and P. Berlandier. Distribution de l'arbre de recherche des problèmes de satisfaction de contraintes en domaines finis. In $2^{ème}$ *Conférence Nationale sur la Résolution de Problèmes NP-Complets*, pages 275–290. JNPC 96, 1996.

[PRB01] T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In *Proceedings of the* 7^{th} *CP*, volume LNCS 2239, pages 451–463, 2001.

[Pro93] P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9 :268–299, 1993.

[Pro94] P. Prosser. Binary Constraint Satisfaction Problem : Some are Harder than Others. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 95–99, 1994.

[Pro95] P. Prosser. MAC-CBJ : maintening arc consistency with conflict-directed backjumping. Research Report 95/117, May 1995.

[RHZ76] A. Rosenfeld, R. Hummel, and S. Zucker. Scene Labelling by Relaxation Operations. *IEEE Transaction on System, Man, and Cybernetics*, 6(6) :420–433, 1976.

[RK87] N. Rao and V. Kumar. Parallel depth-first search. Part I : Implementation. *International Journal of Parallel Programming*, 16 :479–499, 1987.

[RK93] N. Rao and V. Kumar. On the Efficiency of Parallel Backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4) :427–437, 1993.

[Rou92] C. Roucairol. *Exploration parallèle d'espace de recherche en Recherche Opérationnelle et Intelligence Artificielle*, chapter 14, pages 201–211. Masson, 1992. Dans *Algorithmique Parallèle* de M. Cosnard, M. Nivat et Y. Robert.

[RS86] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of tree-width. *Algorithms*, 7 :309–322, 1986.

[RTL76] D. Rose, R. Tarjan, and G. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on computing*, 5 :266–283, 1976.

[Rut94] Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the* 3^{rd} *International Conference on Fuzzy Systems*, 1994.

[San95] P. Sanders. Better algorithms for parallel backtracking. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 333–347. Springer-Verlag, 1995.

[SBK01] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT—Parallel SAT-Checking with Lemma Exchange : Implementation and Applications. In Henry Kautz and Bart Selman, editors, *SAT-2001 Workshop on Theory and Applications of Satisfiability Testing*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science.

[Sch92] Thomas Schiex. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints ?". In *Proceedings of the Eight International Conference on Uncertainty in Artificial Intelligence*, pages 268–275, Stanford, CA, 1992.

[Sch98] T. Schiex. Maximizing the reversible DAC lower bound in Max-CSP is NP-complete. Technical Report 1998/02, INRA, 1998.

[Sch00] T. Schiex. Arc Consistency for Soft Constraints. In *Proceedings of the* 6^{th} *CP*, volume LNCS 1894, pages 411–424, 2000.

[Sch02] T. Schiex. Une comparaison des cohérences d'arc dans les Max-CSP. In *Actes des* $9^{ème}$ *Journées pour la résolution pratique de problèmes NP-Complet*, pages 209–223. JNPC'2002, 2002.

[SF94] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of eleventh ECAI*, pages 125–129, 1994.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems : hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.

[SFV97] T. Schiex, H. Fargier, and G. Verfaillie. Problèmes de satisfaction de contraintes valués. *Revue d'Intelligence Artificielle*, 11(3), 1997.

[SH87] A. Samal and T. Henderson. Parallel Consistent Labeling Algorithms. *International Journal of Parallel Programming*, 16(5) :341–364, 1987.

[SHZ+91] S. Susswein, T. Henderson, J. Zachary, C. Hansen, P. Hinker, and G. Marsden. Parallel Path Consistency. *International Journal of Parallel Programming*, 20(6) :453–473, 1991.

[Sin95] M. Singh. Path Consistency Revisited. In *Proceedings of the 5th International Conference on Tools for Artificial Intelligence*, pages 318–325, 1995.

[Sin96] M. Singh. Path Consistency Revisited. *IJAIT*, 5 :127–141, 1996.

[SV93] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *Proceedings of the 5^{th} IEEE International Conference on Tools with Artificial Intelligence*, Nov. 1993.

[SV94] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[Ter01] C. Terrioux. Cooperative Search and Nogood Recording. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 260–265, 2001.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[TY84] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 (3) :566–579, 1984.

[VLS96] G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proceedings of the 14^{th} AAAI*, pages 181–187, 1996.

[Wal75] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. P. H. Winston, McGraw–Hill, New York, 1975.

[Wal94] R. Wallace. Directed arc consistency preprocessing. In *Proceedings of the ECAI-94 Workshop on Constraint Processing*, volume LNCS 923, pages 121–137, 1994.

[Wal96] R. Wallace. Enhancements of Branch and Bound Methods for the Maximal Constraint Satisfaction Problem. In *Proceedings of AAAI*, pages 188–195, 1996.

[ZM93] Y. Zhang and A. Mackworth. *Parallel and Distributed Finite Constraint Satisfaction : Complexity, Algorithms and Experiments*, chapter 1. Parallel Processing for Artificial Intelligence. Elsevier Sience Publishers B.V., 1993.

Approches structurelles et coopératives pour la résolution des problèmes de satisfaction de contraintes

Résumé : La résolution des problèmes de satisfactions de contraintes (CSPs) fait généralement appel à des recherches arborescentes exploitant des améliorations du backtracking comme le filtrage ou le retour arrière non chronologique. Dans cette thèse, nous nous intéressons à une autre amélioration possible : l'exploitation des informations explicitées durant la résolution.

Dans un premier temps, cette information correspond à la notion de good et de nogood structurels. Un good (respectivement un nogood) structurel est une affectation consistante qui peut (resp. ne peut pas) être étendue en une affectation consistante sur une partie bien déterminée du problème. L'emploi de ces goods et nogoods structurels permet d'éviter certaines redondances dans la recherche. Nous définissons alors une méthode énumérative, nommée BTD, qui produit et exploite ces informations. Cette méthode bénéficie de l'efficacité pratique des méthodes énumératives tout en garantissant des bornes de complexités identiques à celles des meilleures méthodes structurelles. Suite à ce premier travail, nous étendons la méthode BTD au cadre des CSPs valués.

Dans un second temps, nous étudions l'apport de la coopération à une méthode concurrente, la coopération étant basée sur l'échange d'informations produites durant la résolution. Une coopération judicieuse peut alors améliorer considérablement une méthode concurrente en évitant une partie des redondances inhérentes à la concurrence. Nous poursuivons et approfondissons d'abord des travaux initiés par Martinez et Verfaillie sur l'échange de nogoods classiques (i.e. des affectations consistantes qui ne participent pas à une solution). En particulier, nous proposons différents schémas de coopération. Une telle méthode concurrente coopérative obtenant de bons résultats en pratique, nous l'étendons ensuite aux goods et aux nogoods structurels utilisés par la méthode BTD. Enfin, cette extension est adaptée au cadre des CSPs valués.

Mots-clés : problème de satisfaction de contraintes (valué), méthode structurelle, décomposition arborescente, recherche concurrente coopérative.

Structural and cooperative approaches for solving constraint satisfaction problems

Abstract : For solving constraint satisfaction problems (CSPs), we generally use enumerative searches exploiting some improvements of backtracking like filtering or backjumping techniques. In this thesis, we deal with another possible improvement : the exploitation of informations explicited during the search.

In the one hand, this information corresponds to the notion of structural good and nogood. A structural good (respectively nogood) is a consistent instantiation which can (resp. can't) be extended to a consistent instantiation on well-defined part of the problem. Using these structural goods and nogoods allows us to avoid some redundancies in the search. We then define an enumerative method, called BTD, which produces and exploits these informations. This method benefits from the practical efficiency of backtracking searches while providing the same complexity bounds as the best structural methods. Then, we extend the BTD method to valued CSPs framework.

In the other hand, we study the contribution made by the cooperation to a concurrent method, when the cooperation is based on exchanging some informations produced during the search. A sensible cooperation can then improve significantly a concurrent method by avoiding some redundancies inherent in the concurrency. We first go on with and develop some works of Martinez and Verfaillie about the exchange of classical nogoods (i.e. instantiation which can't be extended to a solution). Namely, we propose different schemes of cooperation. As such a cooperative concurrent method obtains in practice goods results, we then extend it to structural goods and nogoods used by the BTD method. Finally, this extension is adapted to the valued CSPs framework.

Keywords : (valued) constraint satisfaction problem, structural method, tree-decomposition, cooperative concurrent search.

Laboratoire des Sciences de l'Information et des Systèmes (LSIS)
Domaine universitaire de Saint-Jérôme – Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20