THÈSE

présentée à

l'UNIVERSITÉ DE METZ

pour l'obtention du grade de
DOCTEUR DE L'UNIVERSITÉ DE METZ

Spécialité : Informatique

Pierre-Paul MÉREL

Les problèmes de satisfaction de contraintes : recherche n-aire et parallélisme – Application au placement en CAO

Soutenue à Metz, le 23 février 1998

Composition du jury :

Directeur de thèse:	Yvon GARDAN	Professeur à l'Université de Reims
Rapporteurs:	Michel COSNARD	Professeur à l'ENS de Lyon
	Catherine ROUCAIROL	Professeur à l'Université de Versailles
Examinateurs:	Jean-Pierre JUNG	Professeur à l'Université de Metz
	Zineb HABBAS	Maître de conférences à l'Université de Metz
	Daniel SINGER	Maître de conférences à l'Université de Metz

. INFORMATIQUE DE METZ

THÈSE

présentée à

l'UNIVERSITÉ DE METZ

pour l'obtention du grade de
DOCTEUR DE L'UNIVERSITÉ DE METZ

Spécialité : Informatique

Pierre-Paul MÉREL

Les problèmes de satisfaction de contraintes : recherche n-aire et parallélisme – Application au placement en CAO

Soutenue à Metz, le 23 février 1998

Composition du jury :

Directeur de thèse:	Yvon GARDAN	Professeur à l'Université de Reims
Rapporteurs:	Michel COSNARD	Professeur à l'ENS de Lyon
	Catherine ROUCAIROL	Professeur à l'Université de Versailles
Examinateurs:	Jean-Pierre JUNG	Professeur à l'Université de Metz
	Zineb HABBAS	Maître de conférences à l'Université de Metz
	Daniel SINGER	Maître de conférences à l'Université de Metz

LABORATOIRE DE RECHERCHE EN INFORMATIQUE DE METZ

À la mémoire de
mon grand-père.

Remerciements[‡]

Je tiens à exprimer ma sincère gratitude à :

Yvon GARDAN, Professeur à l'Université de Reims, pour m'avoir accueilli au sein du Laboratoire de Recherche en Informatique de Metz, et pour avoir accepter de diriger cette thèse.

Michel COSNARD, Professeur à l'ENS de Lyon, et Catherine ROUCAIROL, Professeur à l'Université de Versailles, pour avoir accepté de consacrer un peu de leur précieux temps pour juger ce travail, et pour leurs remarques qui ont permis de l'améliorer.

Zineb HABBAS, Maître de Conférence à l'IUT de Metz, Francine HERRMANN et Daniel SINGER, Maîtres de Conférence à l'Université de Metz, pour m'avoir suivi tout au long de ma thèse, ainsi que pour leur amitié.

Michaël KRAJECKI, pour ses conseils techniques et son soutien amical.

Tous les membres du Laboratoire de Recherche en Informatique de Metz, pour l'accueil chaleureux qu'ils m'ont réservé, et pour l'ambiance de travail.

Ma famille et Odile SOUDANT, pour tant de choses.

[‡] Les moyens de calcul utilisés ont été mis à disposition par le Centre National de Calcul Parallèle en Sciences de la Terre (C.N.C.P.S.T.).

Table des matières

Glossaire

AC-n : algorithmes de filtrage par arc-consistance ($1 \leq n \leq 7$). 15
AC-chip : *arc-consistency chip* – processeur d'établissement de l'arc-consistance. 76

BC : algorithme *backchecking* – marquage arrière négatif. 27
bFC : algorithme de *b-forward-checking* – vérification de consistance en avant bornée. 46
BJ : algorithme de *backjumping* – retour-arrière non chronologique. 26
BM : algorithme *backmarking* – marquage en arrière. 27
BT : algorithme de *backtrack* – recherche par consistance arrière. 24
BTDH : méthode *bottom-left decreasing height* – haut-gauche hauteur décroissante. 130
BTDW : méthode *bottom-left decreasing width* – haut-gauche largeur décroissante. 129
BTIH : méthode *bottom-left increasing height* – haut-gauche hauteur croissante. 129
BTIW : méthode *bottom-left increasing width* – haut-gauche largeur croissante. 129

CAD : *computer aided design.* 124
CAO : conception assistée par ordinateur. 124
CBJ : algorithme de *conflict-based backjumping* – BJ basé sur les conflits. 26
CRCW : *concurrent read/concurrent write* – lecture/écriture concurrentes. 68
CREW : *concurrent read/exclusive write* – lecture concurrente, écriture exclusive. 68
CSP : *constraint satisfaction problem* – problème de satisfaction de contraintes. 6

DkC : algorithme *directionnal-k-consistency* – k-consistance directionnelle. 27
DAG : *directed acyclic graph* – graphes orientés sans cycles. 75
DEF : heuristique *ordre aléatoire* (de définition). 59
DRAM : *distributed random access machine* – machine à accès mémoire distribué. 68
DSPAC-n : algorithmes *distributed static parallel arc-consistency* ($1 \leq n \leq 3$). 78

EREW : *exclusive read/exclusive write* – lecture/écriture exclusives. 68

FC : algorithme de *forward-checking* – recherche par vérification de consistance en avant. 25

GBJ : algorithme de *graph-based backjumping* – BJ basé sur la structure du graphe. 26
GT : algorithme *generate-and-test* – génération et tests. 23

ILB : *initial load balancing* – équilibre de charge initial. 84

LLB : *lazy load balancing* – équilibre de charge paresseux. 87

MAC : algorithme *maintaining arc-consistency* – recherche par maintien de l'arc-consistance. 25
MbFC : *minimal b-forward-checking* – bFC minimisé. 51

MC : heuristique *min-conflict* – conflits minimums. 29
MCO : heuristique *maximal cardinality ordering* – cardinalités maximales. 29
MCV : heuristique *maximal constrained variable* – variable la plus contrainte. 54
MFC : algorithme *minimal forward-checking* – FC minimisé. 28
MIMD : *multiple instruction, multiple data* – plusieurs instruction pour plusieurs données. 70
MRV : heuristique *minimum remaining values* – nombre minimal de valeurs restantes. 29
MWO : heuristique *minimal width ordering* – largeurs minimales. 29

NC : *Nick's Class* – classe des problèmes intrinsèquement parallèles. 75

PAC-n : algorithmes *parallel arc-consistency* ($1 \leq n \leq 3$). 77
PAR de SÉQ : mise en parallèle de blocs séquentiels. 82
PPC-n : algorithmes *parallel path-consistency* ($1 \leq n \leq 3$). 78
PRAM : *parallel random access machine* – machine parallèle à accès mémoire aléatoire. 68

RLB : *regenerating load balancing* – équilibre de charge avec recalcul des chemins. 88

SÉQ de PAR : suite séquentielle de blocs parallèles. 82
SIMD : *single instruction, multiple data* – une instruction pour plusieurs données. 70
SPMD : *single program, multiple data* – un programme pour plusieurs données. 71

VLB : *virtual load balancing* – équilibre de charge virtuel. 84

Introduction

DEPUIS QUELQUES ANNÉES , on assiste à une forte émergence des applications basées sur l'Intelligence Artificielle. L'appellation « Intelligence Artificielle » regroupe en fait plusieurs domaines (programmation logique, systèmes experts, systèmes de preuve, spécification algébrique, ...). Dans le cadre de cette thèse, nous nous intéressons tout particulièrement à l'approche basée sur les contraintes. Celle-ci offre un cadre de formalisation puissant, et induit des méthodes de résolution variées et performantes.

Le cadre formel que nous étudions ici est celui des Problèmes de Satisfaction de Contraintes (CSP[1]) défini par Montanari [Mon 74]. Un CSP est la donnée d'un ensemble de variables définies sur certains domaines, et un ensemble de contraintes reliant ces variables.

L'approche CSP vise à résoudre des problèmes souvent difficiles qu'une approche classique – programmation algorithmique – ne permet pas, ou permet difficilement de résoudre. Les performances seront au cœur de la reflexion car les problèmes à résoudre seront en général NP-complets, ce qui signifie qu'il n'existera pas de méthodes de résolution polynômiales. La complexité des algorithmes sera donc le plus souvent exponentielle et les travaux de recherche porteront sur la mise en œuvre de méthodes et heuristiques qui permettent de réduire en moyenne le temps nécessaire à la résolution d'un problème.

Les méthodes de résolution définies pour un formalisme d'Intelligence Artificielle donné tentent généralement d'exploiter au mieux ses spécificités, mais on trouvera toutefois un grand nombre de similitudes méthodologiques entre les différents formalismes. Ceci nous conduira donc parfois à utiliser des techniques définies en dehors du cadre des CSP pour les appliquer à ce dernier.

Avant d'entrer dans les détails des méthodes de résolution des CSP, il est important de souligner la différence entre deux grandes classes d'algorithmes de recherche de solution : les méthodes complètes, et les méthodes incomplètes. Les algorithmes qui se classent parmi les premières permettront toujours de trouver une solution, mais rencontreront alors le problème de la NP-complétude et seront en général de complexité exponentielle. À l'opposé, les algorithmes de la seconde classe ne permettront pas de trouver de solution dans tous les cas de figure, mais seront plus rapides que les algorithmes complets pour les problèmes de grande taille. Nous nous concentrerons dans cette étude sur les algorithmes complets de recherche de solution pour les CSP.

La littérature scientifique abonde d'algorithmes complets de recherche de solution(s) pour les CSP. On trouvera de plus des méthodes qui transforment le CSP initial en un CSP équivalent plus facile à résoudre (filtrage des domaines par l'établissement d'une consistance locale) ou plusieurs CSP par décomposition.

[1] CSP : de l'anglais *Constraint Satisfaction Problems.*

Toutefois, les comparaisons expérimentales ont montré que les méthodes ne sont pas toutes aussi performantes [Pro 94, Smi 94, BR 96] et celles qui exhibent les meilleurs comportements implémentent toutes des mécanismes de vérification de consistance en avant, *i.e.* à chaque nouvelle instanciation, les objets qui ne sont pas encore instanciés sont parcourus afin de vérifier s'ils pourront être instanciés de façon compatible avec l'instanciation considérée. On notera que les expérimentations ont engendré une classification particulière des problèmes résolus. En effet, la densité des conflits dans les contraintes influe sensiblement sur la satisfiabilité et le coût de résolution des problèmes. Elle définit le phénomène de *transition de phase* [GW 94] qui correspond à la zone où se situent les problèmes les plus difficiles.

Les problèmes qui se définissent naturellement à l'aide de contraintes n-aires[2] peuvent se réécrire avec uniquement des contraintes binaires, dont l'utilisation est plus simple. Cela a conduit la plupart des auteurs à ne se concentrer que sur les CSP binaires. Ainsi, les méthodes de recherche par consistance en avant sont basées sur une structure binaire.

Un de nos objectifs pour cette étude est de montrer que la transformation des CSP n-aires en CSP binaires pour une simplification de la recherche conduit à introduire une lourdeur dans la résolution. La première partie de ce document traitera donc de la recherche n-aire. En effet, il sera nécessaire de modifier les algorithmes de résolution par consistance en avant, de telle sorte qu'ils puissent résoudre des CSP n-aires. Une expérimentation pratique conclura cette partie. Elle validera notamment l'intérêt de la résolution n-aire directe des CSP.

Comme nous l'avons vu plus haut, la résolution complète des problèmes d'Intelligence Artificielle NP-complets, et donc des CSP est particulièrement coûteuse. Ceci rend le recours au parallélisme d'autant plus important. Or, la littérature est très pauvre en terme de parallélisation des CSP. Nous étudierons dans la deuxième partie de ce document, différentes méthodes parallèles de résolution de CSP. L'objectif global de notre étude ne sera pas de définir un nouvel algorithme monolithique, mais plutôt de proposer un cadre générique de résolution qui permettra de combiner aussi bien les différentes techniques de recherche que les méthodes de parallélisation. On parlera ainsi d'*algorithmes hybrides* puisque ces techniques pourront être associées pour obtenir des performances obtimales. Tant les CSP binaires que les CSP n-aires pourront être résolus par cet algorithme.

Les problèmes spécifiques au parallélisme seront étudiés et intégrés dans notre algorithme qui présentera aussi une généricité sur ce plan. On s'intéressera en particulier à tout ce qui vise à équilibrer globalement la charge des processeurs, que se soit en début de l'algorithme (répartition initiale de la charge), ou au cours de la recherche (division judicieuse du travail). Nous expérimenterons les techniques ainsi définies sur des problèmes difficiles, *i.e.* en suivant le phénomène de transition de phase.

De même que la plupart des études présentées dans la littérature, les algorithmes que nous proposons dans les deux premières parties sont définis pour des problèmes quelconques et ne tiennent donc pas compte des spécificités d'un problème particulier. Nous étudions dans la troisième partie de ce document, les spécialisations des algorithmes de ré-

[2] une contrainte n-aire porte sur n contraintes. On utilise généralement cette appellation en opposition avec les contraintes *binaires* qui ne portent que sur deux contraintes.

solution de CSP pour un problème concret de CAO[3] : le problème du bordereau de coupe. Celui-ci consiste à placer les différentes pièces d'un patron de coupe pour un vêtement sur une surface restreinte. Les algorithmes étudiés seront conçus et expérimentés aussi bien dans un environnement séquentiel que parallèle.

Ce document se compose de trois parties, découpées chacune en trois chapitres. La logique générale de ce plan vise à séparer les trois cadres de travail que nous avons étudié : aspects séquentiels des CSP, aspects paralléle, et application pratique sur un problème de CAO. Chaque partie comprendra un chapitre sur l'état de l'art dans le domaine, un chapitre regroupant les apports proposés, et un chapitre présentant un certain nombre d'expérimentations. Le détail des chapitres est le suivant :

Le chapitre 1 présentera le cadre général des CSP. Il donnera un aperçu des différentes méthodes de résolution séquentielles, en s'attardant particulièrement sur les techniques énumératives qui constituent le cœur de cette thèse.

Le chapitre 2 proposera une définition de consistance n-aire étendue par rapport à celles existantes. Ceci nous permettra de définir un algorithme de recherche par consistance en avant capable de prendre en compte les contraintes n-aires.

Le chapitre 3 validera les algorithmes proposés dans le chapitre 2 par des expérimentations statistiques. Il mettra notamment en évidence l'intérêt de la résolution n-aire directe par rapport à la résolution d'un CSP binaire équivalent.

Le chapitre 4 présentera les méthodes proposées dans la littérature pour résoudre les CSP en parallèle. Il sortira d'ailleurs un peu du cadre des CSP pour explorer les méthodes proposées dans le cadre de la programmation logique.

Le chapitre 5 proposera un algorithme générique de résolution parallèle des CSP. Celui-ci permettra d'hybrider aussi bien les méthodes de recherche n-aires ou binaires, que les méthodes de parallélisation. Nous proposerons d'ailleurs de nouvelles méthodes parallèles qui utilisent les spécificités du formalisme CSP.

Le chapitre 6 expérimentera l'algorithme et les méthodes proposées sur des jeux d'essai aléatoires. Ceci permettra de comparer notamment les méthodes de parallélisation classiques avec celles définies spécialement pour les CSP.

Le chapitre 7 définira le problème pratique de CAO que nous étudierons : le problème du bordereau de coupe. Nous passerons en revue les principales méthodes de résolution proposées, et nous mettrons en évidence les particularités du problème vis-à-vis des CSP.

Le chapitre 8 parlera de l'utilisation des CSP pour la résolution du problème du bordereau de coupe. Nous proposerons des méthodes de recherche séquentielles et parallèles spécifiques afin d'utiliser au mieux les particularités du problème.

Le chapitre 9 testera les algorithmes proposés dans le chapitre 8 sur un exemple concret. Nous ferons ici la part entre les différents environnements de recherche (une solution ou optimisation), et les cadres d'exécution séquentiels et parallèles.

[3]CAO : Conception Assistée par Ordinateur.

Première partie

Algorithmique séquentielle pour les CSP

Chapitre 1

État de l'art : CSP binaires et n-aires

DANS CE CHAPITRE, nous présentons le cadre général de cette thèse, à savoir les *problèmes de satisfaction de contraintes.* Ceux-ci sont définis comme un ensemble de variables sur lesquelles portent des contraintes. Une solution est un ensemble d'instanciations de chaque variable qui satisfait toutes les contraintes.

L'objectif global n'est pas ici de présenter une revue exhaustive et détaillée de toutes les techniques qui peuvent s'appliquer aux CSP, mais de donner une classification précise de ces méthodes, afin de pouvoir aisément les situer les unes par rapport aux autres. Pour une présentation plus détaillée, on pourra se référer avantageusement à [Tsa 93] qui aborde la plupart des concepts et algorithmes relatifs aux CSP, ainsi que [Jég 91], [Kum 92], ou [AS 93, Chap. 14]. Toutefois, ces différents états de l'art devraient être complétés par les récents travaux sur l'hybridation [Pro 93b] et la transition de phase [GW 94, Pro 94].

Ce chapitre ne suivra pas la structure arborescente de la classification des méthodes de résolution des CSP car elle repousserait la majeure partie de cette présentation dans une sous-section volumineuse. Après avoir présenté les notions de base formalisant un CSP (section 1), nous présentons succinctement les deux grandes classes d'algorithmes de résolution (complets et incomplets) dans le paragraphe 2, puis nous détaillons plus précisément la classe des algorithmes complets. Le paragraphe 3 présente l'algorithme de synthèse de contraintes qui reste peu utilisé. Nous abordons ensuite les notions de filtrage (paragraphe 4) qui permettent de déduire d'un CSP des informations réduisant le coût pratique d'une recherche. Nous énumérons ensuite dans le paragraphe 5, les techniques de décomposition d'un CSP complexe en plusieurs sous-CSP, plus faciles à résoudre. Puis nous détaillons les méthodes qui permettent de diminuer le coût effectif d'une recherche complète (paragraphe 6). Nous terminons enfin avec la section 7 qui présente les cadres théoriques et expérimentaux de classification des CSP.

1 Notions de base

Nous présentons dans ce paragraphe, un ensemble de définitions qui seront utilisées tout au long de ce document. Nous les illustrerons en outre par un exemple de CSP : le problème classique des n-reines.

1.1 Quelques définitions

Le formalisme CSP a été introduit par Montanari [Mon 74]. Il s'appliquait alors aux problèmes de vision et de reconnaissance d'images. Toutefois, plusieurs auteurs ont par la suite utilisé ce formalisme sous des noms divers. L'appellation « CSP » est d'ailleurs due à Mackworth [Mac 77], car pour sa part, Montanari parlait de « *Constraint Networks*[1] » (CN). On trouve aussi dans des publications anciennes, l'appellation « *Consistent Labelling Problem*[2] » [HS 79].

Définition 1.1 (Montanari) *Un* **Problème de Satisfaction de Contraintes** $\mathcal{P}$ *est un quadruplet* $\mathcal{P} = (X, D, C, R)$, *où :*

- $X = \{X_1, ..., X_n\}$ *est un ensemble de* n variables.
- $D = \{D_1, ..., D_n\}$ *est un ensemble de* n domaines *finis. Chaque domaine* D_i *est associé à une variable* X_i.
- $C = \{C_1, ..., C_m\}$ *est un ensemble de* m contraintes. *Chaque contrainte* C_i *est définie par un ensemble de* n_i *variables* $\{X_{i_1}, ..., X_{i_{n_i}}\} \subseteq X$.
- $R = \{R_1, ..., R_m\}$ *est un ensemble de* m relations. *Chaque relation* R_i *définit l'ensemble des* n_i*-uplets sur* $D_{i_1} \times ... \times D_{i_{n_i}}$ *autorisés par la contrainte* C_i.

Nous ne précisons pas – volontairement – si les relations sont définies en extension (*i.e.* listes de n-uplets autorisés) ou en intention (par des relations algébriques, logiques...). Si on se réfère à la définition originelle de Montanari, elles doivent être définies en extension. Toutefois, certains auteurs considèrent que les relations sont définies en intention. Dans un souci de généralité, nous laissons une imprécision dans la définition pour n'exclure aucune particularité.

Remarque : Des CSP plus généraux portant sur des domaines infinis (à valeurs réelles) ont été définis [Dav 87]. Toutefois, les méthodes de résolution sont sensiblement différentes de celles conçues pour les CSP finis, et ne seront pas abordées dans ce document. ◇

Définition 1.2 *L'***arité** *d'une contrainte* $C_i = \{X_{i_1}, ..., X_{i_{n_i}}\}$ *est le nombre* i_{n_i} *des variables sur lesquelles porte* C_i.

Définition 1.3 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est un* CSP **binaire** *si* $\forall C_i \in C$, *l'arité de* C_i *est au plus de 2.*

[1] *Constraint Networks* = réseaux de contraintes.
[2] *Consistent Labelling Problem* = problème de l'étiquetage consistant.

Définition 1.4 *Étant donné un* CSP $\mathcal{P} = (X, D, C, R)$ *et* $Y \subseteq X$, $(d_{y_1}, ..., d_{y_{|Y|}}) \in D_{y_1} \times ... \times D_{y_{|Y|}}$ *est une* **instanciation consistante** *des variables de* Y *sur* D *si* $\forall C_i \in C / C_i \subseteq Y, (d_{y_1}, ..., d_{y_{|Y|}})[C_i] \in R_i$.[3]

Définition 1.5 *Une* **solution** *de* $\mathcal{P} = (X, D, C, R)$ *est une instanciation consistante des variables de* X *sur* D. *L'ensemble des solutions de* $\mathcal{P}$ *est noté* $Sol_{\mathcal{P}}$.

Une **solution** *de* $\mathcal{P} = (X, D, C, R)$ *est une instanciation* $(d_1, ..., d_n) \in \bowtie_{R_i \in R} R_i$.[4]

Définition 1.6 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **consistant** *si et seulement si* $Sol_{\mathcal{P}} \neq \emptyset$.

Définition 1.7 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **globalement consistant** *si et seulement si* $\forall C_i \in C$, $Sol_{\mathcal{P}}[C_i] = R_i$.

D'une manière générale, un CSP $\mathcal{P} = (X, D, C, R)$ définit la structure d'un problème sur lequel on peut être amené à poser plusieurs questions :

1. *Consistance de* $\mathcal{P}$ ($\mathrm{Sol}_{\mathcal{P}} \neq \emptyset$?) : ce problème, que l'on appelle aussi *problème de satisfiabilité*, a été démontré comme étant NP-complet [Mac 77].
2. *Recherche d'une solution* (trouver $d \in \mathrm{Sol}_{\mathcal{P}}$) : c'est un problème NP-difficile (*i.e.* au moins aussi difficile que NP-complet).
3. *Nombre de solutions* ($|\mathrm{Sol}_{\mathcal{P}}|$) : ce problème est #P-complet (notion de dénombrement [GJ 79]).
4. Recherche de l'*ensemble des solutions* ($\mathrm{Sol}_{\mathcal{P}}$) qui sera aussi un problème NP-difficile.

Remarque : Les définitions de classe des différents problèmes ont été démontrées avec des CSP binaires. La taille des problèmes considérés est alors proportionnelle au nombre de variables. ◇

On peut aussi poser d'autres questions comme trouver une valeur qui figure dans toutes les solutions, trouver les solutions où figure une valeur donnée, trouver la solution qui maximise une fonction d'évaluation, etc. Nous nous concentrerons dans cette étude, sur les questions 1 à 4. On notera par ailleurs que la « communauté CSP » s'intéresse essentiellement à la recherche d'une seule solution (question 2).

La structure d'un CSP $\mathcal{P} = (X, D, C, R)$ est donnée par le couple (X, C). Dans le cas d'un CSP binaire (toutes les contraintes $C_i \in C$ sont d'arité au plus 2), (X, C) est assimilé à un **graphe**. Dans le cas général, (X, C) sera un **hypergraphe** [Ber 70].

Définition 1.8 *Soient deux* CSP $\mathcal{P} = (X, D, C, R)$ *et* $\mathcal{P}' = (X, D', C', R')$ *; on dit que* $\mathcal{P}$ *et* $\mathcal{P}'$ *sont* **équivalents** *– noté* $\mathcal{P} \equiv \mathcal{P}'$ *– si et seulement si* $Sol_{\mathcal{P}} = Sol_{\mathcal{P}'}$.

[3] $(d_{y_1}, ..., d_{y_{|Y|}})[C_i]$ note la projection de l'instanciation $(d_{y_1}, ..., d_{y_{|Y|}})$ sur les variables de C_i.

[4] $\bowtie$ exprime la jointure naturelle de deux relations : un k-uplet t sur $C_1 \cup C_2$ est dans la jointure $R_1 \bowtie R_2$ *ssi* $t[C_1] \in R_1$ et $t[C_2] \in R_2$.

La notion d'*équivalence de* CSP est fondamentale pour les techniques de résolution car beaucoup d'entre elles sont basées sur le principe de transformation d'un CSP en un CSP équivalent, plus facile à résoudre.

Définition 1.9 *Une contrainte* $C_i = \{X_{i_1}, ..., X_{i_{n_i}}\}$ *est appelée* **contrainte vide** *si sa relation vérifie* $R_i = \emptyset$.

Une contrainte $C_i = \{X_{i_1}, ..., X_{i_{n_i}}\}$ *est appelée* **contrainte universelle** *si sa relation vérifie* $R_i = D_{i_1} \times ... \times D_{i_{n_i}}$.

Définition 1.10 *Soit un* CSP $\mathcal{P} = (X, D, C, R)$ *:*

- *Soient* C_i *et* C_j *; si* $C_i = C_j$ *et si* $R_i \subseteq R_j$*, alors on dit que la contrainte* C_i *(respectivement* C_j*) est* **plus forte** *(respectivement* **plus faible***) que la contrainte* C_j *(respectivement* C_i*).*
- *Soient* C_i *et* C_j *; on appelle* **composition** *de* C_i *et* C_j *– ou* **contrainte induite** *par* C_i *et* C_j *– la contrainte* $C_k = (C_i \cup C_j) - (C_i \cap C_j)$ *dont la relation est définie par* $R_k = (R_i \bowtie R_j)[C_k]$.
- *La contrainte* C_k *est dite* **redondante** *si* $\mathcal{P} \equiv (X, D, C \setminus \{C_k\}, R \setminus \{R_k\})$.

Du fait que l'ajout ou le retrait d'une contrainte redondante ne modifie pas l'ensemble des solutions d'un problème, on peut définir pour un même problème, une classe regroupant tous les CSP équivalents qui représentent ce problème. Comme dans tous les cas de classes d'équivalences, on cherche à connaître son représentant le plus complet (la forme canonique). Montanari définit à ce titre la notion de *réseau de contraintes minimal* – pour les CSP binaires – où les contraintes sont toutes explicites, *i.e.* on ne peut ajouter de contraintes induites non-existantes [Mon 74]. Il montre de plus, que ce réseau est unique.

Définition 1.11 *Soit* $\mathcal{P} = (X, D, C, R)$*, un* CSP *binaire. Le* **réseau de contraintes minimal** *associé à* $\mathcal{P}$ *est noté* $\mathcal{P}_m = (X, D, C_m, R_m)$ *et vérifie :*

- $\mathcal{P} \equiv \mathcal{P}_m$ *;*
- $\forall C_{m_i} \in C_m, Sol_{\mathcal{P}_m}[C_i] = R_{m_i}$ *($\mathcal{P}_m$ est globalement consistant) ;*
- $\forall C_{m_i} \in C_m, |C_{m_i}| = 2$ *(toutes les contraintes de C_m sont binaires) ;*
- (X, C_m) *est un graphe complet.*

Le calcul d'un tel réseau correspond à la recherche de la satisfiabilité d'un problème puisqu'il exhibe un CSP globalement consistant. Ce sera donc un problème difficile – NP-complet – dans le cas d'un CSP binaire quelconque.

1.2 Exemple de CSP : le problème des n-reines

Ce problème s'énonce comme suit : « *étant donné un échiquier de taille $n \times n$, placer n reines sur cet échiquier de telle sorte que deux reines ne soient pas en prise mutuelle,* i.e. *deux reines ne peuvent être placées sur une même ligne, une même colonne ou une même diagonale* ».

Une formalisation classique consiste à définir une variable par colonne. Comme il faut choisir une dimension, nous prenons ici le problème pour un échiquier de taille 4×4 puisque c'est le plus petit possible admettant une solution (on peut facilement se rendre compte qu'on ne peut pas placer 3 reines sur un échiquier de taille 3×3).

Exemple 1.1 : Le problème des 4-reines.
On définit donc le problème $\mathcal{P}_{\text{4-reines}} = (X, D, C, R)$, avec :

- $X = \{X_1, X_2, X_3, X_4\}$;
- $D = \{D_1, D_2, D_3, D_4\}$, avec $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$;
- $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$, avec $C_1 = \{X_1, X_2\}, C_2 = \{X_1, X_3\}, C_3 = \{X_1, X_4\}$, $C_4 = \{X_2, X_3\}, C_5 = \{X_2, X_4\}, C_6 = \{X_3, X_4\}$;
- $R = \{R_1, R_2, R_3, R_4, R_5, R_6\}$, avec
  $R_1 = R_4 = R_6 = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$,
  $R_2 = R_5 = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\}$,
  $R_3 = \{(1,2), (1,3), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,2), (4,3)\}$.

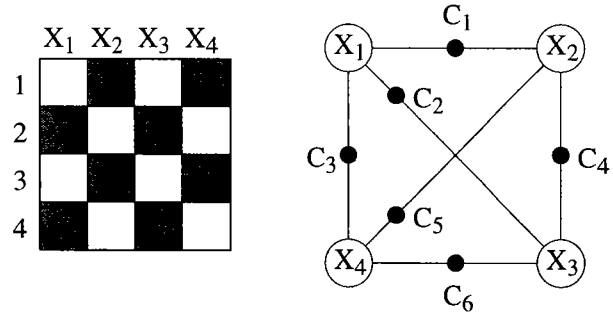


Figure 1.1 : Échiquier du problème des 4-reines et son graphe de représentation.

La figure 1.1 présente le graphe de contraintes du problème des 4-reines. Les variables sont représentées par un cercle et les contraintes par un point. L'inclusion d'une variable dans une contrainte est symbolisée par une arête la reliant à la contrainte. Dans la suite de ce document, cette convention de représentation sera toujours adoptée.

La question posée est ici de trouver **une solution**. On pourra facilement voir que l'ensemble des solutions est $\text{Sol}_{\mathcal{P}_{\text{4-reines}}} = \{(X_1 = 2, X_2 = 4, X_3 = 1, X_4 = 3), (X_1 = 3, X_2 = 1, X_3 = 4, X_4 = 2)\}$.

2 Résolution des CSP : classification des différentes méthodes

Un très grand nombre de méthodes ont été définies pour résoudre des CSP. Certaines sont issues directement du formalisme CSP, comme les méthodes de filtrage ou de décomposition, d'autres sont des transpositions de méthodes déjà existantes dans des domaines proches (logique, recherche opérationnelle, bases de données, ...). D'une façon générale, la recherche de solutions nécessite de trouver une instanciation de chaque variable qui soit compatible avec un certain nombre de contraintes. Or, l'énumération de toutes les combinaisons possibles conduit à une complexité exponentielle, donc inutilisable sur de grands problèmes. Nous délimitons dans ce paragraphe les grandes classes d'algorithmes de résolution des CSP, en précisant pour chacune d'elles, les concepts mis en œuvre pour réduire l'exploration exhaustive des combinaisons de valeurs.

2.1 Classification selon le type de résultat attendu

Comme on l'a vu dans le paragraphe 1, plusieurs questions peuvent être associées à une même structure graphique de problème, ce qui induira des caractéristiques particulières sur les algorithmes utilisés pour les résoudre.

2.1.1 Résultat complet

Si l'objectif est de trouver toutes les solutions à un problème donné, les algorithmes y répondant devront mettre en œuvre des mécanismes qui permettent le parcours de toutes les combinaisons. Il sera donc difficile dans ce cas d'éviter une énumération des combinaisons.

2.1.2 Résultat partiel

Lorsque l'objectif pour un problème est de décider s'il admet une solution (satisfiabilité), ou de trouver une solution, il n'est pas nécessaire de parcourir exhaustivement toutes les combinaisons. Dans ce cas, l'algorithme s'arrête lorsqu'il a trouvé une solution. Toutefois, ce type de question peut s'avérer aussi difficile que pour un résultat complet si le problème n'est pas consistant (on dit aussi *insatisfiable*).

2.1.3 Résultat optimal

L'objectif est alors de trouver une solution qui minimise une fonction d'évaluation donnée. Il n'est pas nécessaire de parcourir l'ensemble de l'espace des solutions si la fonction peut s'appliquer sur une instanciation partielle. La plupart des algorithmes qui cherchent un résultat optimal pour un CSP sont issus de la recherche opérationnelle[5].

On notera que la fonction d'évaluation peut porter sur le nombre de contraintes satisfaites. Dans ce cas, on ne cherche pas une solution exacte, mais une solution qui viole le moins de contraintes possibles. Cette notion est formellement définie comme des *problèmes de satisfaction partielle de contraintes* (PCSP) dans [Fre 89].

[5]le lecteur intéressé par une présentation des principaux algorithmes de ce type pourra consulter [Rou 93].

Les algorithmes qui visent cet objectif sont peu explorés dans le monde des CSP, et leurs spécificités les distinguent des algorithmes visant un résultat complet ou partiel. Nous les exclurons pour le reste de cette étude.

2.2 Classification selon la qualité du résultat attendu

Selon les applications mettant en œuvre des problèmes de satisfaction de contraintes, il n'est pas toujours nécessaire – voire pas toujours possible – d'obtenir une solution exacte. Nous distinguerons donc dans cette optique, deux grandes classes d'algorithmes : les algorithmes complets, et les algorithmes incomplets.

2.2.1 Algorithmes complets

Ces algorithmes donneront toujours une réponse exacte à la question posée, que ce soit pour la vérification de la satisfiabilité, ou pour la recherche de la solution optimale. En contre partie de la qualité du résultat, la recherche pourra être très coûteuse car elle nécessite un parcours exhaustif de l'espace des solutions. Malgré les améliorations que l'on pourra apporter à ce type de techniques, elle resteront cantonnées à des problèmes de taille réduite[6].

En fait, deux grandes approches peuvent être dégagées dans les algorithmes complets de résolution des CSP : une approche *orientée contraintes*, appelée aussi « *bottom-up* » car la recherche s'effectue des contraintes les plus petites vers celles d'arité élevée, et une approche *orientée variables*, ou *top-down* car le parcours de l'arbre de recherche s'effectue du haut vers le bas.

Les méthodes faisant l'objet de cette étude se classent dans cette catégorie. Les paragraphes qui suivent traitent des algorithmes complets. Hormis la section 3, le reste de ce chapitre est consacré aux méthodes orientées par les variables.

2.2.2 Algorithmes incomplets

L'objectif de ces algorithmes n'est pas de donner systématiquement une bonne réponse à la question posée, mais de donner une réponse acceptable en un temps raisonnable. Ces méthodes sont incapables de donner un résultat complet car l'espace des solutions ne sera pas totalement exploré. Pour un résultat partiel, elles seront très efficaces si la densité de solutions est élevée. Par contre, elles pourront ne pas donner de résultat si le problème a peu de solutions, et ne concluront jamais si le problème est insatisfiable.

Nous avons exclu les méthodes incomplètes du cadre de cette étude. Le lecteur intéressé par une étude exhaustive de ces méthodes pourra se référer au nombreux articles traitant du sujet, comme [AHU 83, Tsa 93].

3 Algorithmes de synthèse

Cette classe d'algorithmes constitue un peu le parent pauvre de la satisfaction de contraintes : elle est issue d'un « héritage » des Bases de Données et reste surtout utile pour le calcul de toutes les solutions d'un problème, alors que la plupart des travaux

[6] la limite de 80 variables est souvent citée.

de recherche dans le domaine des CSP visent à répondre au problème de la recherche d'une seule solution. Ils seront donc plus coûteux quand le nombre de solutions pour un problème est élevé.

Le principe général de ces méthodes consiste à calculer la jointure naturelle de toutes les relations qui composent le problème. Nous citerons Freuder [Fre 78] comme « père » de cette approche puisque c'est lui qui l'a introduite dans le monde des CSP[7]. Selon le schéma général de l'algorithme, on pourra distinguer deux grandes techniques de synthèse de contraintes : les algorithmes de *synthèse incrémentale* qui construisent l'espace des solutions en y introduisant successivement chaque contrainte (paragraphe 3.1), et les algorithmes de *synthèse en treillis* (paragraphe 3.2).

3.1 Synthèse incrémentale

Le principe général de la synthèse incrémentale de contraintes consiste à construire l'espace des solutions en prenant en compte successivement chaque contrainte du CSP. Nous présentons ici la version naïve de la technique de synthèse qui concatène les contraintes dans un ordre quelconque.

Algorithme 1.1 *Synthèse incrémentale naïve de contraintes.*

```
1 fonct Synthèse (R) ↦ relation ≡
2   soit Sol_P ∈ R
3   R ⟵ R \ Sol_P
4   tant que R ≠ ∅ faire
5             soit r ∈ R
6             R ⟵ R \ r
7             Sol_P ⟵ Sol_P ⋈ r fintant que
8   retourner Sol_P.
```

La fonction Synthèse est appelée avec le paramètre R pour un CSP $\mathcal{P} = (X, D, C, R)$.

Exemple 1.2 : Jointures du CSP de l'exemple 1.1.
Nous présentons les valeurs successives de $\text{Sol}_{\mathcal{P}_i} = \{C_{\text{Sol}_i}, R_{\text{Sol}_i}\}$:

Étape 1 : $C_{\text{Sol}_1} = C_1 = \{X_1, X_2\}$,
$R_{\text{Sol}_1} = R_1 = \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\}$.

Étape 2 : $C_{\text{Sol}_2} = C_{\text{Sol}_1} \cup C_2 = \{X_1, X_2, X_3\}$,
$R_{\text{Sol}_2} = R_{\text{Sol}_1} \bowtie R_2 = \{(1,3,2),(1,3,4),(1,4,2),(1,4,4),(2,4,1),(2,4,3),(3,1,2),$
$(3,1,4),(4,1,1),(4,1,3),(4,2,1),(4,2,3)\}$.

Étape 3 : $C_{\text{Sol}_3} = C_{\text{Sol}_2} \cup C_3 = \{X_1, X_2, X_3, X_4\}$,
$R_{\text{Sol}_3} = R_{\text{Sol}_2} \bowtie R_3 = \{(1,3,2,2),(1,3,4,2),(1,4,2,2),(1,4,4,2),(1,3,2,3),(1,3,4,3),$
$(1,4,2,3),(1,4,4,3),(2,4,1,1),(2,4,3,1),(2,4,1,3),(2,4,3,3),(2,4,1,4),(2,4,3,4),$
$(3,1,2,1),(3,1,4,1),(3,1,2,2),(3,1,4,2),(3,1,2,4),(3,1,4,4),(4,1,1,2),(4,1,3,2),$
$(4,2,1,2),(4,2,3,2),(4,1,1,4),(4,1,3,4),(4,2,1,4),(4,2,3,4)\}$.

[7] Freuder utilise en fait la notion de propagation de contraintes qui correspond à un *join* dans ce contexte.

Étape 4 : $C_{\text{Sol}_4} = C_{\text{Sol}_3} \cup C_4 = \{X_1, X_2, X_3, X_4\}$,
$R_{\text{Sol}_4} = R_{\text{Sol}_3} \bowtie R_4 = \{(1,4,2,2),(1,4,2,3),(2,4,1,1),(2,4,1,3),(2,4,1,4),(3,1,4,1),(3,1,4,2),(3,1,4,4),(4,1,3,2),(4,1,3,4)\}$.

Étape 5 : $C_{\text{Sol}_5} = C_{\text{Sol}_4} \cup C_5 = \{X_1, X_2, X_3, X_4\}$,
$R_{\text{Sol}_5} = R_{\text{Sol}_4} \bowtie R_5 = \{(1,4,2,3),(2,4,1,1),(2,4,1,3),(3,1,4,2),(3,1,4,4),(4,1,3,2),(4,1,3,4)\}$.

Étape 6 : $C_{\text{Sol}_6} = C_{\text{Sol}_5} \cup C_6 = \{X_1, X_2, X_3, X_4\}$,
$R_{\text{Sol}_6} = R_{\text{Sol}_5} \bowtie R_6 = \{(2,4,1,3),(3,1,4,2)\}$.

Le CSP admet donc deux solutions : $(2,4,1,3)$ et $(3,1,4,2)$.

Le principal problème de cette approche est qu'elle nécessite un grand espace pour son exécution quand le nombre de solutions est important. De plus, un ordre arbitraire engendre la construction de relations intermédiaires volumineuses qui ne seront réduites qu'en fin de traitement.

Les algorithmes de la littérature sont naturellement plus évolués. Ils évitent notamment le problème de la construction de grosses relations qui ne sont réduites qu'en fin d'algorithme. Dans [Sei 81], Seidel définit le concept d' « invasion » du graphe de contraintes : l'algorithme avance dans sa construction de l'ensemble des solutions en choisissant à chaque étape comme prochaine contrainte à intégrer, celle qui introduit le moins de variables nouvelles[8]. Ainsi, l'ensemble des solutions partielles croît de façon moins chaotique.

Pour leur part, Guesgen, Ho et Hilfinger dans [GHH 92] ne construisent pas explicitement les contraintes successives, mais les projections de celles-ci sur les domaines par un système de « marquage[9] ».

3.2 Synthèse en treillis

La synthèse des contraintes en treillis, introduite par Freuder dans [Fre 78], consiste à construire les relations successives selon un treillis des contraintes d'ordre i ($1 \leq i \leq n$), *i.e.* les contraintes portant sur i variables. La construction commence par les nœuds d'ordre 1 (les variables) puis 2 (contraintes binaires), jusqu'à n (solutions globales). la figure 1.2 présente le treillis des contraintes des différent ordres pour un CSP sur 4 variables. Les chiffres i entre accolades correspondent aux X_i inclus dans chaque nœud.

À chaque niveau de construction, les inconsistances sont propagées dans les niveaux inférieurs pour épurer les structures de données. Cette méthode demeure cependant assez théorique, puisque aucun auteur ne s'y est intéressé pour son utilisation dans une application pratique.

4 Filtrage des CSP

Comme le calcul d'un réseau de contraintes minimal (définition 1.11) équivalent à un CSP est un problème NP-complet, des propriétés locales plus faibles, mais de complexité

[8] le nom d'*invasion* vient ici du fait que l'algorithme progresse dans le graphe en suivant sa structure. Seidel définit en particulier la notion de *front* qui correspond à l'ensemble des variables desquelles partent des contraintes non vérifiées.

[9] dans l'article original en anglais, les auteurs nomment cette technique « *tagging method* ».

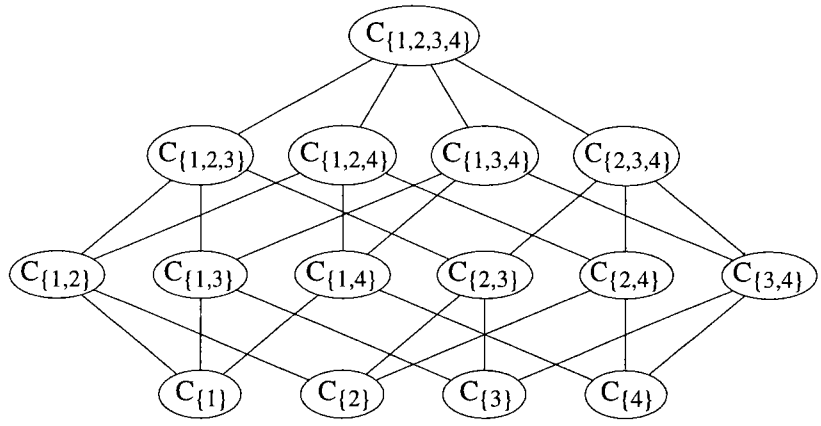


Figure 1.2 : treillis des contraintes pour un CSP sur quatre variables.

polynômiales ont été introduites. Les notions plus anciennes [Mon 74, Mac 77] portent sur des CSP binaires. Les définitions de consistance s'appliquant sur des CSP n-aires sont nettement plus récentes [Jég 91, DvB 95].

4.1 Consistances binaires

La principale notion de consistance locale portant sur des CSP binaires est la k-consistance [Fre 78] :

Définition 1.12 *Un* CSP *est dit* **k-consistant** *si pour tout k-uplet $(X_1, \ldots, X_k)$, pour toute instanciation A consistante des $k-1$ variables $(X_1, \ldots, X_{k-1})$, il existe une valeur $v \in D_k$ telle que l'instanciation $A \cup \{X_k = v\}$ soit consistante.*

Cette définition est généralement utilisée dans une extension plus utilisable sur le plan pratique :

Définition 1.13 *Un* CSP $\mathcal{P}$ *est dit* **fortement k-consistant** *ssi* $\forall i, 1 \leq i \leq k$, $\mathcal{P}$ *est i-consistant.*

La k-consistance forte assure que l'on peut choisir k valeurs quelconques sur n variables sans qu'une incompatibilité n'apparaisse entre elles. Historiquement, les notions de nœud-consistance ($k = 1$), arc-consistance ($k = 2$) et chemin-consistance ($k = 3$) ont été définies en premier [Mon 74] et restent encore très utilisées. Cooper dans [Coo 89], a développé un algorithme optimal pour l'établissement de la k-consistance en $\Theta(n^k d^k)$[10]. Ceci explique que les algorithmes de consistance de niveau élevé sont très peu utilisés.

De façon générale, l'établissement d'une k-consistance conduit à la recherche de toutes les *contraintes $(k-1)$-aires induites* par le CSP. L'arc-consistance induit donc des contraintes unaires (suppressions de valeurs dans certains domaines), la chemin-consistance induit des contraintes binaires, etc.

L'arc-consistance est la consistance locale la plus utilisée car elle est relativement simple à mettre en œuvre et reste peu coûteuse par rapport à la réduction de problème

[10] n est le nombre de variables dans le graphe, est d est la taille du plus grand domaine.

qu'elle induit. Elle repose sur l'idée suivante : si une valeur $v \in D_i$ n'apparaît dans aucun des n-uplets d'une contrainte portant sur X_i, alors v ne peut participer à une solution globale du CSP et on peut donc la supprimer de D_i.

Définition 1.14 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **arc-consistant** *ssi* $\forall X_i \in X$, *on a* $D_i \neq \emptyset$ *et* $\forall v \in D_i, \forall C_k = \{X_i, X_j\} \in C, \exists r \in R_k / r[X_i] = v$.

Diverses versions successives de l'algorithme d'établissement de l'arc-consistance ont été développées – dernièrement, Freuder à défini l'algorithme *AC-7* dans [Fre 95] – , mais elles utilisent toutes la même procédure *Filtre* qui modifie un domaine pour que la propriété d'arc-consistance soit vérifiée entre deux variables du CSP.

Algorithme 1.2 *Établissement de la consistance d'un arc.*

```
1 fonct Filtre (i,j) ↦ booléen ≡                    /* filtre D_i par rapport à D_j */
2   suppression ⟵ faux
3   soit C_k = {X_i, X_j} ∈ C      /* C_k correspond à la contrainte entre X_i et X_j */
4   pour tout x ∈ D_i faire
5       si ∄y ∈ D_j/(x,y) ∈ R_k alors
6         D_i ⟵ D_i \ {x}
7         suppression ⟵ vrai finsi finpour
8   retourner suppression.                  /* pour indiquer si D_i a été modifié */
```

La version qui montre le mieux le fonctionnement de l'arc-consistance est AC-3. Elle utilise une liste des arcs qui ont nécessité une réduction de domaine, pour ne reconsidérer que les arcs concernés par la propagation d'une suppression[11].

Algorithme 1.3 *Procédure AC-3.*

```
1 proc AC-3 ≡                                       /* filtre tous les domaines du CSP */
2   Q ⟵ {(i,j)/i ≠ j ∧ {X_i, X_j} ∈ C}
3   tant que Q ≠ ∅ faire
4           soit (i,j) ∈ Q
5           Q ⟵ Q \ {(i,j)}
6           si Filtre (i,j) alors
7             Q ⟵ Q ∪ {(k,i)/k ≠ i ∧ {X_i, X_k} ∈ C} finsi fintant que.
```

Théorème 1.15 ([MF 85]) *La complexité temporelle d'AC-3 est en* $O(md^3)$, *où* m *est le nombre de contraintes dans le graphe et* d *est le nombre maximal de valeurs par variable. Sa complexité spatiale est en* $O(m)$.

Exemple 1.3 : Filtrage par arc-consistance.

Soit le problème $\mathcal{P} = (X, D, C, R)$, avec :

- $X = \{X_1, X_2, X_3\}$;
- $D = \{D_1, D_2, D_3\}$, avec $D_1 = D_2 = D_3 = \{a, b\}$;

[11] une valeur supprimée pouvait être le support de valeurs dans d'autres domaine, et donc entraîner des suppressions en cascade.

- $C = \{C_1, C_2, C_3\}$, avec $C_1 = \{X_1, X_2\}, C_2 = \{X_1, X_3\}, C_3 = \{X_2, X_3\}$;
- $R = \{R_1, R_2, R_3\}$, avec $R_1 = \{(a,a),(a,b)\}$, $R_2 = R_3 = \{(a,a),(b,b)\}$.

La première étape de l'établissement de l'arc-consistance fait apparaître que la valeur b dans X_1 n'est supportée par aucune valeur de X_2 selon la contrainte C_1. Ceci implique la suppression de b dans X_1, qui entraîne ensuite la suppression de b dans X_2, puis de b dans X_3 par propagation.

Par la suite, Mohr et Henderson [MH 86] ont défini l'algorithme AC-4 et ont montré son optimalité en $\Theta(md^2)$. Son principe consiste à maintenir pour chaque valeur dans une variable, la liste des valeurs des autres variables qu'elle valide (notion de *support*), ce qui permet de ne reconsidérer par la procédure *Filtre*, que les contraintes nécessaires. Malheureusement, AC-4 n'est pas facilement utilisable dans la pratique car il nécessite de grosses structures de données, et son coût moyen s'avère en pratique très légèrement supérieur à AC-3 [Wal 93].

AC-5 a été défini par Deville et Van Hentenryck [DvH 91, vHDT 92]. Il s'agit en fait d'un algorithme générique conçu pour tirer partie de propriétés spécifiques de certaines classes restreintes de contraintes. Ils obtiennent ainsi des complexités inférieures à $\Theta(md^2)$ pour quelques classes de contraintes.

AC-4 a été amélioré par Béssière [Bes 94] en ne calculant et mémorisant qu'un seul support par valeur (AC-6). Sa complexité temporelle reste identique, mais son encombrement spatial est plus réduit. Les résultats expérimentaux semblent montrer que AC-6 est plus performant que AC-3.

AC-7 – la dernière version proposée – par Freuder [Fre 95] est sensiblement identique à AC-6 dans le cas général, mais permet de tirer parti de certaines propriétés des contraintes, comme la réflexivité, pour réduire le coût pratique de l'établissement de l'arc-consistance. Elle peut s'avérer deux fois plus rapide que AC-6 dans ce dernier cas.

Le concept de k-consistance peut s'appliquer de différentes manières :

- choisir un niveau de consistance arbitraire (en général 2 ou 3) et filtrer le CSP de telle sorte que la propriété de k-consistance soit vérifiée sur le CSP restant ;
- choisir un niveau de consistance arbitraire, mais n'appliquer le filtrage que de haut en bas, selon un ordre prédéfini (consistance directionnelle) ;
- choisir un ordre d'instanciation des variables, et établir une consistance à chaque niveau dont le degré puisse garantir une recherche de solution efficace : la consistance adaptative.

4.1.1 Consistance directionnelle

La définition de la *consistance directionnelle* a été proposée par Dechter et Pearl dans [DP 87]. L'idée générale est de tenir compte du fait que le filtrage est un prétraitement pour l'algorithme *backtrack* (*cf.* paragraphe 6). Or dans ce dernier, l'ordre d'instanciation est prédéfini (on suppose que l'on n'utilise pas d'heuristiques d'ordre dynamique sur les variables). Dans ce cas, l'application de la procédure *Filtre* depuis un sommet qui doit être instancié vers un sommet qui l'est déjà n'est pas utile. La consistance directionnelle (*DAC*) prend donc en compte l'ordre d'instanciation de la procédure de recherche. On évite ainsi la moitié du travail de filtrage qui est inutile pour la phase de recherche qui suit.

4.1.2 Consistance adaptative

La consistance adaptative est la conséquence du théorème de Freuder [Fre 82], lui-même basé sur le concept de largeur d'un CSP.

Définition 1.16 *Soient $\mathcal{P} = (X, D, C, R)$, un* CSP *binaire et un ordre φ sur ses variables. La* **largeur** *d'une variable X_i est égale à $|\{X_j / \{X_i, X_j\} \in C \wedge \varphi(X_i) < \varphi(X_j)\}|$. La largeur d'un ordre est le maximum de la largeur des variables dans cet ordre. La largeur d'un* CSP *est la largeur minimale sur tous ses ordres.*

Théorème 1.17 (Freuder) *Soit un* CSP *$\mathcal{P}$ de largeur w. Si le niveau de forte consistance de $\mathcal{P}$ est supérieur à w, alors il existe un ordre d'instanciation glouton*[12].

L'idée de base de l'algorithme d'établissement de consistance adaptative est donc de forcer la forte $(w+1)$-consistance pour chaque variable de largeur w. On obtient alors un niveau de consistance suffisant pour garantir une recherche sans retour-arrière. Deux points pénalisent toutefois cette méthode :

- La complexité de l'algorithme est exponentielle car elle est systématique quelle que soit la largeur du graphe. Elle réalise un filtrage optimal mais qui est souvent trop fort : les résultats expérimentaux montrent que cette méthode est moins efficace en moyenne que les algorithmes complets les plus simples [DM 94a].
- L'établissement de la k-consistance induit des contraintes d'arité $k-1$ qui modifient la largeur initiale du graphe. Les calculs que l'on peut faire avant l'algorithme pour savoir si la méthode est intéressante sont alors faussés. On ne peut donc pas évaluer la complexité de la méthode *à priori*. La complexité finale de la méthode est en $O(nd^{l*})$, où $l*$ est la largeur du graphe obtenue après l'établissement de la consistance adaptative.

4.1.3 (i, j)-consistance

La notion de (i, j)-*consistance* a été définie par Freuder dans [Fre 85] comme une généralisation de la k-consistance. Un CSP sera dit (i, j)-consistant si toute instanciation consistante sur i variables peut être étendue de façon consistante à une instanciation sur $(i+j)$ variables[13]. La forte (i, j)-consistance est vérifiée si la (i', j)-consistance est vérifiée pour tout $i' \leq i$.

Si cette notion permet d'étendre l'aspect théorique de la traditionnelle k-consistance, elle n'est utilisée dans la pratique – à notre connaissance – par aucun algorithme, ni aucun outil spécifique.

[12] qui permet de résoudre le CSP sans revenir sur l'instanciation d'une variable. Dans ce cas, la recherche est polynômiale en le nombre de variables.

[13] dans ce contexte, la k-consistance se définit donc comme la $(k-1, 1)$-consistance.

4.2 Consistances n-aires

Si les CSP binaires sont restés longtemps le principal centre d'intérêt des recherches dans le domaine, quelques auteurs se sont penchés sur les hypergraphes de contraintes, avec notamment des définitions de consistances qui puissent s'appliquer sur ce type de problèmes. Nous citerons dans ce paragraphe, deux notions importantes qui divergent quelque peu quant à leur inspiration et leur mise en pratique.

Le premier à s'être intéressé aux CSP n-aires est Jégou [Jég 91]. Il définit une consistance n-aire qui est une application de la k-consistance sur les contraintes – et non plus sur les variables :

Définition 1.18 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **hyper-k-consistant** *si et seulement si :*

- $\forall C_i \in C, R_i \neq \emptyset$.
- $\forall C_1, \ldots, C_k \in C, (\bowtie_{1 \leq i \leq k-1} R_i)[(\bigcup_{1 \leq i \leq k-1} C_i) \cap C_k] = R_k[(\bigcup_{1 \leq i \leq k-1} C_i) \cap C_k]$.

De même que pour la k-consistance, on définit la forte hyper-k-consistance comme suit :

Définition 1.19 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **fortement hyper-k-consistant** *si et seulement si* $\forall i, 1 \leq i \leq k, \mathcal{P}$ *est hyper-i-consistant.*

Plus simplement, on dira qu'un CSP n-aire est hyper-k-consistant si une instanciation consistante des variables reliées par $(k-1)$ contraintes quelconques peut être étendue à une instanciation sur une k^{e} contrainte. L'établissement de l'hyper-k-consistance s'obtient par des algorithmes reprenant la structure de la famille AC-x, mais les instanciations de valeurs sur des variables sont remplacées par des instanciations de n_i-uplets sur des contraintes, c.-à-d. des éléments des relations. On dit que l'hyper-k-consistance est une notion duale[14] de la k-consistance. Mais, les résultats théoriques obtenus sur les CSP binaires, comme le théorème de Freuder (*cf.* théorème 1.17) ne peuvent directement s'appliquer sur les CSP n-aires par cette définition.

On notera que le filtrage obtenu par l'hyper-k-consistance qui filtre les relations est sans effets dans le cadre d'un pré-filtrage pour un algorithme énumératif classique (sur les variables) car il n'induit aucune modification des domaines des variables, et donc aucune réduction de la complexité de la recherche. Pour contourner ce problème, Jégou a proposé de projeter les relations sur chaque domaine après avoir établi l'hyper-k-consistance.

De leur coté, Dechter et Van Beek ont étendu directement la k-consistance aux CSP n-aires [DvB 95] afin de pouvoir profiter des résultats théoriques qui faisaient défaut à l'hyper-k-consistance.

Définition 1.20 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **relationnel-k-consistant** *si et seulement si* $\forall C_1, \ldots, C_{k-1} \in C, \forall X_j \in \bigcap_{1 \leq i \leq k-1} C_i$,

[14] la notion de graphe dual est abordée dans le paragraphe 5. De plus, une comparaison complète des méthodes de filtrage n-aire est donnée dans le paragraphe 1 du chapitre 2.

$$\rho((\bigcup_{1\leq i\leq k-1} C_i)\setminus\{X_j\})\subseteq(\bowtie_{1\leq i\leq k-1} R_i)[(\bigcup_{1\leq i\leq k-1} C_i)\setminus\{X_j\}].$$

Ici, $\rho(A)$ dénote l'ensemble des instanciations consistantes des variables de A.

Définition 1.21 *Un* CSP $\mathcal{P}=(X,D,C,R)$ *est* **fortement relationnel-k-consistant** *si et seulement si* $\forall i, 1\leq i\leq k, \mathcal{P}$ *est relationnel-i-consistant.*

Trivialement, on dira qu'un CSP est relationnel-k-consistant si chaque valeur de chaque variable fait partie d'une instanciation consistante des variables reliées par $(k-1)$ contraintes. La principale vertu de cette définition de consistance est que l'élément de base est la variable, et non un groupe de variables. Ainsi, elle trouve toute son utilité dans un contexte de pré-filtrage pour un algorithme de recherche énumératif basé sur l'instanciation des variables.

Le problème RC_k d'établissement de la relationnelle-k-consistance est de complexité exponentielle à partir de $k=3$ si les contraintes ne sont pas binaires, car lorsqu'on force la consistance entre deux contraintes d'arité >2, on génère de nouvelles contraintes d'arité supérieure qui devront d'être vérifiées avec les contraintes déjà existantes. Ainsi, l'itération de cette procédure peut engendrer des contraintes d'arité $n-1$ (si $n=|X|$). Ces limitations sur la relationnelle-k-consistance de degré supérieur à 2 font que seule la relationnelle-2-consistance est applicable en pratique pour des CSP quelconques.

Dans la mesure où l'établissement de la relationnelle-k-consistance est une procédure de pré-traitement d'une méthode de recherche énumérative, Dechter et Van Beek ont défini une propriété plus faible : la directionnelle relationnelle-k-consistance qui tient compte de l'ordre d'instanciation de la recherche pour éviter une partie du travail de filtrage.

Algorithme 1.4 *Établissement de la relationnelle-k-consistance.*

```
1  fonct RC_k ↦ booléen ≡
2    consistant ⟵ vrai
3    répéter Q ⟵ C
4            S ⟵ R
5            pour tout {C_{i_1},...,C_{i_{k-1}}}/C_{i_j} ∈ Q faire
6                 pour tout X_α ∈ ∩_{j=1}^{k-1} C_{i_j} faire
7                      C_β ⟵ ∪_{j=1}^{k-1} C_{i_j} \ {X_α}
8                      C ⟵ C ∪ {C_β}
9                      si C_β ∈ Q alors R_β ⟵ R_β ∩ (⋈_{j=1}^{k-1} R_{i_j}[C_β])
10                     sinon R_β ⟵ ⋈_{j=1}^{k-1} R_{i_j}[C_β] finsi
11                     si R_β = ∅ alors consistant ⟵ faux finsi finpour finpour
12           jusqu'à S = R ∧ consistant finrépéter
13   retourner consistant.
```

5 Techniques de décomposition

Nous regroupons dans ce paragraphe, les techniques qui transforment le CSP initial à résoudre, en un ou plusieurs CSP dont le traitement sera plus facile. Deux objectifs

différents (voire opposés) peuvent motiver de telles méthodes : transformer un CSP n-aire en un CSP binaire équivalent (paragraphe 5.1), ou diviser un CSP complexe en un ensemble de CSP indépendants plus faciles à résoudre (paragraphe 5.1).

5.1 Transformations n-aire vers binaire

La plupart des travaux sur les CSP se contentent de traiter les CSP binaires car ils sont plus simples à appréhender, et parce que l'on peut toujours transformer un CSP n-aire en CSP binaire [Mon 74, DP 88a]. C'est cette transformation qui nous intéresse tout particulièrement ici. Nous passons en revue les différentes méthodes de transformation utilisées dans la littérature, en mettant en évidence leurs avantages et inconvénients respectifs.

5.1.1 Graphe de contraintes primal

Le passage d'un hypergraphe de contraintes – n-aire – à un graphe de contraintes primal – binaire – se fait en conservant toutes les variables et en remplaçant les contraintes n-aires par des *cliques*[15]. On substitue donc à chaque contrainte n-aire, $\frac{n(n-1)}{2}$ contraintes binaires. La relation de chaque nouvelle contrainte est définie par la projection de la relation n-aire sur le couple de ses variables.

Cette méthode a été proposée par Montanari [Mon 74]. Elle est basée sur la Théorie des Graphes [Ber 70], et notamment sur la notion de *2-section* d'un hypergraphe H notée $(H)_2$.

Définition 1.22 *Soit $\mathcal{P}_H = (X, D, C, R)$ un* CSP *quelconque. Le* **graphe de contraintes primal** *de $\mathcal{P}_H$ est le* CSP *noté $\mathcal{P}_{(H)_2} = (X, D, C_A, R_A)$ tel que :*

- $(X, C_A) = (X, C)_2$;
- $R_A = \{R_{A_1}, ..., R_{A_{m'}}\}$ *où* $\forall C_{A_q} = \{X_i, X_j\}, R_{A_q} = \bigcap_{k/C_k = \{X_i, X_j\} \subseteq C_{A_q}} R_k[\{X_i, X_j\}]$.

L'utilisation d'une transformation d'un hypergraphe en graphe de contraintes primal présente quelques inconvénients qui peuvent être rédhibitoires dans certains cas :

- L'équivalence entre le CSP n-aire et son graphe primal ne peut être garantie dans le cas général. La seule garantie que préserve cette transformation est que $\text{Sol}_{\mathcal{P}_H} \subseteq \text{Sol}_{\mathcal{P}_{(H)_2}}$.
- L'acyclicité du problème n'est pas conservée car un CSP n-aire acyclique peut engendrer un graphe primal cyclique. Cette particularité devient parfois catastrophique dans l'utilisation des méthodes de décomposition (*cf.* paragraphe 5.2).

Ces deux inconvénients majeurs font que le graphe de contraintes primal est une approche inadaptée pour traiter les hypergraphes de contraintes.

[15] une clique est un sous-graphe complet.

5.1.2 Graphe de contraintes biparti

Cette transformation est basée sur la notion de *graphe biparti* issue de la Théorie des Graphes : l'ensemble des sommets du graphe biparti est constitué des sommets X_i de l'hypergraphe et d'un sommet X_{C_k} pour chaque contrainte C_k. Les contraintes connectent chaque X_{C_k} aux sommets de X que contient C_k. Chaque contrainte n-aire se trouve donc transformée en 1 variable et n contraintes binaires.

Définition 1.23 *Soit $\mathcal{P}_H = (X, D, C, R)$ un* CSP *quelconque. Le* **graphe de contraintes biparti** *de $\mathcal{P}_H$, noté $\mathcal{P}_{B(H)} = (X_B, D_B, C_B, R_B)$ tel que :*

- $X_B = X \cup C$;
- $D_B = D \cup R$ *avec* $D_{B_i} = D_i$ *si* $X_{B_i} = X_i$ *et* $D_{B_k} = R_k$ *si* $X_{B_k} = C_k$;
- $C_B = \{\{X_i, C_j\} / X_i \in C_j\}$;
- $R_B = \{R_{B_1}, ..., R_{B_{m'}}\}$ *où* $\forall C_{B_k} = \{X_i, C_j\}, R_{B_k} = \{(d_i, r_j) \in D_i \times R_j / r_j[X_i] = d_i\}$.

Propriété 1.24 *Soit $\mathcal{P}_H = (X, D, C, R)$ un* CSP *et $\mathcal{P}_{B(H)} = (X_B, D_B, C_B, R_B)$ son graphe de contraintes biparti. On a $\mathcal{P}_H \equiv \mathcal{P}_{B(H)}$.*

Par rapport au graphe de contraintes primal, le principal inconvénient est donc résolu. Toutefois, la représentation d'un CSP n-aire sous forme de graphe de contraintes biparti ne préserve pas l'acyclicité de l'hypergraphe dans le cas général.

5.1.3 Graphe de contraintes dual

Un graphe de contraintes dual[16] [DP 89] est construit en créant une variable par contrainte de l'hypergraphe de départ et en reliant – par une nouvelle contrainte binaire – toutes les variables dont les contraintes correspondantes partagent des variables.

Définition 1.25 *Soit $\mathcal{P}_H = (X, D, C, R)$ un* CSP *quelconque. Le* **graphe de contraintes dual** *de $\mathcal{P}_H$ est le* CSP *binaire $\mathcal{P}_{D(H)} = (C, R, E, Q)$ tel que :*

- $E = \{\{C_i, C_j\} \subset C / i \neq j$ *et* $C_i \cap C_j \neq \emptyset\}$;
- $\forall E_k = \{C_i, C_j\} \in E, Q_k = \{(r_i, r_j) \in R_i \times R_j / r_i[C_i \cap C_j] = r_j[C_i \cap C_j]\}$.

Propriété 1.26 *Soit $\mathcal{P}_H = (X, D, C, R)$ un* CSP *et $\mathcal{P}_{D(H)} = (C, R, E, Q)$ son graphe de contraintes dual. On a $\mathcal{P}_H \equiv \mathcal{P}_{D(H)}$.*

La transformation en graphe dual garantit les mêmes propriétés que la transformation en graphe biparti, mais elle est plus souvent utilisée car plus facile à mettre en œuvre : il suffit de raisonner sur les contraintes au lieu des variables (instanciation de n-uplets), ce qui ne nécessite pas la construction explicite du graphe. Dans les chapitres qui suivent, ce sera d'ailleurs cette méthode qui sera toujours employée pour les comparaisons entre les méthodes par transformation binaire et les méthodes directes pour la résolution des CSP n-aires.

[16] on retrouve aussi l'appellation « graphe représentatif » [Ber 70, Jég 91]. Nous préférons l'appellation « graphe dual » pour la mettre en rapport avec la notion de graphe primal.

5.2 Décomposition d'un CSP binaire

Le principe global des méthodes de décomposition consiste à construire, à partir d'un CSP $\mathcal{P}$, un ensemble de CSP $\{\mathcal{P}_i\}$, tel qu'il soit possible de construire les solutions de $\mathcal{P}$ à partir des solutions des $\mathcal{P}_i$. Le choix des $\mathcal{P}_i$ est fait de telle sorte que l'ensemble des solutions des $\mathcal{P}_i$ soit plus facile à calculer que $\mathcal{P}$.

5.2.1 Méthode coupe-cycle

Cette méthode a été présentée dans [DP 87]. L'idée maîtresse est fondée sur le théorème de Freuder (*cf.* théorème 1.17) : si on a un graphe de largeur 1, l'arc-consistance garantit une recherche sans retour-arrière. On transformera le graphe n-aire en une série de graphes acycliques[17]. Chaque sous-CSP sera construit en instanciant une variable dans chaque cycle du graphe initial. Sa résolution sera alors polynômiale en lui appliquant un filtrage par 2-consistance.

Exemple 1.4 : Décomposition par la méthode coupe-cycle.

On se donne le CSP $\mathcal{P} = (X, D, C, R)$ de la figure 1.3, avec $D_3 = D_6 = \{a, b\}$. La décomposition donne les 4 sous-CSP arborescents de la figure. Les variables coupées apparaissent dédoublées, avec leur valeur d'instanciation.

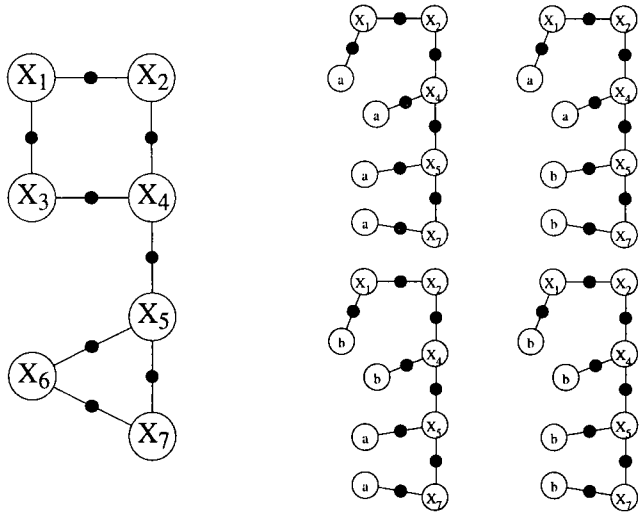


Figure 1.3 : Un CSP cyclique et sa décomposition par la méthode coupe-cycle.

Le point central de cette méthode est la détermination de l'ensemble des variables à instancier, appelé *ensemble coupe-cycle*. En effet, si la résolution d'un seul sous-CSP est polynômiale, leur nombre est exponentiel en la taille de l'ensemble coupe-cycle, puisqu'il sera nécessaire dans le pire des cas, d'instancier le produit cartésien des domaines de cet ensemble. On notera de plus, que la détermination d'un ensemble coupe-cycle minimal est un problème NP-difficile.

[17] c.-à-d. un arbre, ou plus généralement une forêt.

5.2.2 Méthode de regroupement en hyper-arbre

La méthode de regroupement en hyper-arbre s'inscrit en opposition face à la binarisation des CSP, puisqu'elle part d'un CSP binaire pour le transformer en un CSP n-aire. Toutefois, elle garantit la propriété d'acyclicité pour l'hypergraphe de contraintes résultant [DP 88b].

Le principe de l'algorithme de regroupement consiste à trianguler le graphe de contraintes initial en ajoutant des contraintes universelles pour mailler les cycles de plus de trois sommets. Il faut ensuite calculer l'ensemble des cliques maximales sur le graphe triangulé, chaque clique définissant une contrainte dans l'hyper-arbre résultat. La résolution se fera alors par l'établissement de l'hyper-2-consistance qui garantira alors une recherche sans retour-arrière.

Exemple 1.5 : Regroupement en hyper-arbre.

La figure 1.4, présente un regroupement en hyper-arbre du CSP de l'exemple 1.4 . La triangulation apparaît en tirets. Les plus grandes cliques comportent ici 3 variables, et sont ainsi transformées en contraintes 3-aires ou binaires (points noirs).

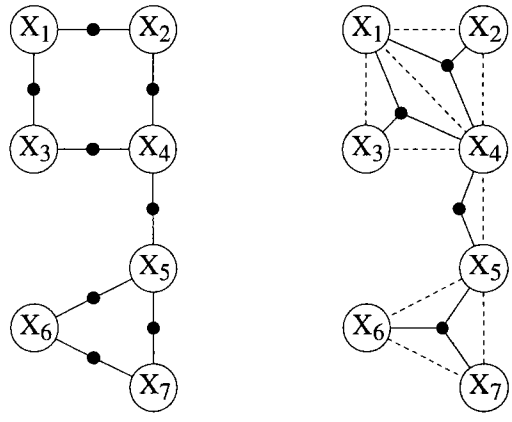


Figure 1.4 : Un CSP cyclique et son hyper-arbre de contraintes.

La méthode de regroupement en hyper-arbre reste exponentielle en la taille de la plus grande hyper-arête.

6 Recherche énumérative

L'algorithme le plus élémentaire dans le contexte de la recherche énumérative de solutions pour un CSP est l'algorithme *generate-and-test* (*GT*). Son principe consiste à générer toutes les combinaisons possibles entre les domaines, et de tester la consistance de chaque combinaison. Il induit un arbre de recherche complet comme dans la figure 1.5.

Les cercles correspondent aux phases « *generate* » et les lettres « T » sur chaque feuille de l'arbre correspondent aux tests de consistance. Dans cet algorithme, le parcours de l'arbre est rapide (génération des instanciations), mais sa taille est exponentielle.

Tous les algorithmes de recherche énumérative sont des améliorations de *GT*. De très nombreuses méthodes ont été définies pour diminuer son coût pratique, chacune mettant en avant certaines caractéristiques particulières. Pendant longtemps, les différentes

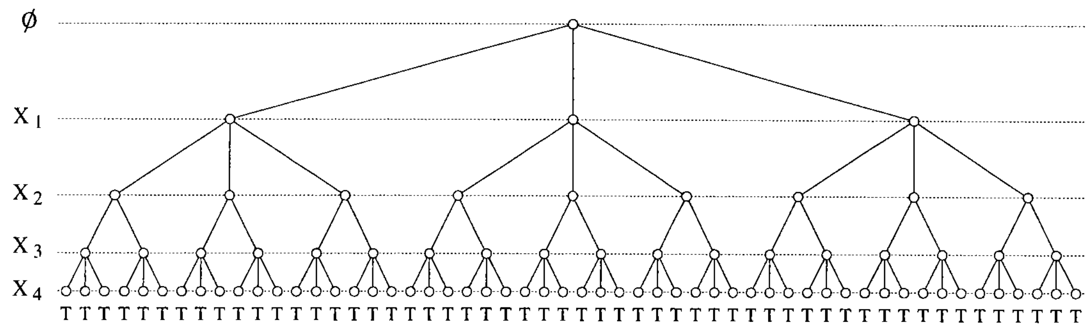


Figure 1.5 : Arbre de recherche *generate-and-test.*

améliorations se sont faites de façon assez anarchique, et les algorithmes proposés étaient peu comparables car portant sur des points algorithmiques différents. La classification que nous mettons en avant dans cette section repose sur une idée de Prosser dans [Pro 93b]. Celui-ci a mis en évidence que deux caractères fondamentaux de ces algorithmes, la descente et la remontée dans l'arbre de recherche, pouvaient être extraits des différentes méthodes, puis hybridées afin de construire des algorithmes disposant des avantages des deux méthodes « mères ». Il définit par exemple l'algorithme *FC-CBJ* qui implémente une descente de type *FC* et une remonté de type *CBJ*. Ces deux techniques sont présentées plus loin.

Dans un souci d'une plus grande précision dans la taxinomie et dans l'hybridation potentielle des différentes techniques, nous définissons en fait, non pas deux, mais huit caractéristiques algorithmiques fondamentales qui seront détaillées dans les paragraphes qui suivent :

1. la *consistance* établie à chaque instanciation ;
2. le traitement des *retours-arrière* en cas de blocage lors de la descente dans l'arbre de recherche ;
3. la *mémorisation* des informations découvertes au cours de la recherche ;
4. le *marquage* des différents éléments algorithmiques au cours de la recherche ;
5. le *retardement* de certains traitements ;
6. l'*ordonnancement* des variables lors de la descente dans l'arbre de recherche;
7. l'*ordonnancement* des valeurs au moment de chaque instanciation ;
8. l'*ordonnancement* des contraintes au moment de la vérification de consistance (locale ou globale).

Nous présenterons pour chaque point, les méthodes s'y rapportant proposées dans la littérature. Les éléments de comparaison établis seront aussi discutés.

6.1 Consistance au cours de la recherche

Les principales améliorations de *GT* consistent à vérifier une consistance à chaque instanciation. On distingue deux grands types de consistance :

- En arrière, ou *look-back schemes* dans le cas d'algorithmes comme *backtrack* (*BT*) : elle consiste à vérifier la consistance de l'instanciation en cours avec chaque instanciation déjà effectuée [Wal 60]. Elle se traduit par une suppression de branches directement sous le nœud en cours d'instanciation – dans le cas où une inconsistance est détectée.

- En avant, ou *look-ahead schemes*. Ces méthodes consistent à vérifier la consistance de l'instanciation en cours avec toutes les variables non encore instanciées. Elles se traduisent par une suppression de branches dans toute la profondeur du sous-arbre enraciné sous le nœud en cours d'instanciation.

Si *BT* constitue une amélioration significative de *GT*, il n'est plus cité aujourd'hui que dans un souci de rappel historique car son coût pratique est prohibitif face aux méthodes de consistance en avant. En effet, ces dernières méthodes de recherche sont apparues comme nettement supérieures à l'algorithme *BT* [HE 80, DM 94a].

Le concept de consistance en avant au cours de la recherche a été proposé par Haralick et Elliot dans [HE 80], avec la définition de l'algorithme de *forward-checking* (*FC*). Ce dernier est une version dégradée de l'arc-consistance, qui se contente de filtrer les domaines des variables non encore instanciées en fonction d'une nouvelle instanciation, sans propagation des suppressions.

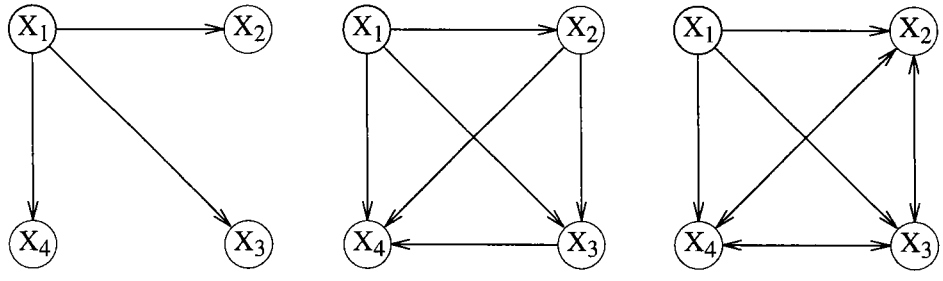


Figure 1.6 : Arc-consistances dégradées pour les méthodes *look-ahead schemes*. L'instanciation courante est X_1. Les flèches symbolisent les applications de la fonction *Filtre*.

L'étape de propagation des inconsistances peut être réduite de différentes manières. Dans le cas de *FC*, aucune propagation n'est faite (*cf.* figure 1.6(**a**)). Dans le *partial look future*, les propagations se font vers le bas, en respectant un ordre statique d'instanciation des variables (figure 1.6(**b**)). La propagation totale des inconsistances, comme dans la figure 1.6(**c**) prend le nom de *look future* dans [HE 80, HS 79, HS 80], mais de nombreux auteurs l'on renommé *maintaining arc-consistency*, ou *MAC*[18] [BR 96, FD 96, Pro 95b, SF 94].

La figure 1.7 schématise sur l'arbre de *GT* les élagages résultant des différentes techniques d'amélioration. Pour *BT*, l'algorithme supprime tout le sous-arbre dont l'instanciation inconsistante est la racine. Dans *FC*, l'algorithme supprime des branches plus profondes dans l'arbre. Les méthodes *BJ* et *DkC* schématisées dans la figure seront présentés plus loin.

[18]en français : Maintien de l'Arc-Consistance.

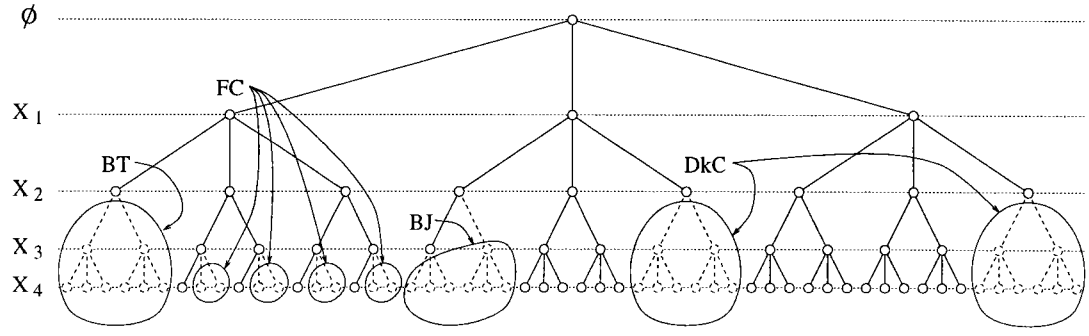


Figure 1.7 : Élagages effectués par *BT*, *FC*, *BJ* et *DkC*. Les effets sont entourés et leur origine marquée par le nom de l'algorithme.

Il est à noter que l'établissement de la consistance en avant est plus coûteuse sur chaque nœud, mais que les algorithmes qui l'appliquent explorent significativement moins de nœuds. De plus, dans un schéma général de résolution des CSP, l'utilisation d'un algorithme de pré-filtrage avec AC – ou un niveau supérieur – rend l'intérêt des schémas en avant caduque. Toutefois, il est admis dans la communauté CSP[19] que la recherche avec consistance en avant est globalement moins coûteuse que le tandem ⟨filtrage, recherche chronologique⟩.

6.2 Retour-arrière

Quand *GT* rencontre une inconsistance dans sa phase de test, il cherche l'instanciation suivante par un retour-arrière chronologique. Les améliorations classiques du retour-arrière sont :

- Le *backjumping* (*BJ*) [Gas 79] : il se fait en mémorisant le conflit le plus profond rencontré lors de l'instanciation d'une variable. Cet algorithme élémentaire en ce qui concerne le retour-arrière, nécessite peu de données : un seul conflit est stocké.
- Le *graph-based backjumping* (*GBJ*) [Dec 90] : il se fait en utilisant la structure du graphe pour les retours-arrière, sans tenir compte des conflits rencontrés, ce qui évite tout stockage de données à la descente.
- Le *conflict-based backjumping* (*CBJ*) [Gin 93, Pro 93b] : il se fait en utilisant la liste des conflits rencontrés lors de l'instanciation des variables. Son implémentation nécessite de stocker la liste des conflits rencontrés lors de la descente.

Dans l'axe « retour-arrière », le degré 0 correspond au *backtrack* chronologique puisque ce dernier n'effectue aucun élagage de l'arbre de recherche au cours de la remontée. L'élagage effectué par ces méthodes concerne des branches sœurs, à droite du nœud en cours d'instanciation (voir figure 1.7). Dans leur forme originelle, ces techniques ont été présentées avec un mécanisme de consistance en arrière, alors que les utilisations actuelles sont hybridées avec des algorithmes à consistance en avant.

Prosser a montré expérimentalement que *CBJ* était la meilleure des trois techniques de retour-arrière [Pro 93b]. Mais en hybridation avec une consistance en avant telle que

[19] ce résultat a été démontré par une expérimentation partielle dans [DM 94a].

FC, les écarts entre *CBJ* et les deux autres techniques sont moins nets. Il apparaît même selon [BR 96] et [GS 96] que *CBJ* dégrade les performances de *MAC*. D'une façon générale, l'hybridation d'une technique de retour-arrière avec un mécanisme de consistance avant réduit considérablement son effet d'élagage puisque la plupart des inconsistances auront été supprimées lors de la descente dans l'arbre [Pro 93a]. En plus de cela, *CBJ* requiert le maintient de structures de données volumineuses, ce qui explique en grande partie son effet négatif sur *MAC*. On notera que l'hybridation de *MAC* avec une méthode de retour-arrière moins coûteuse en temps comme *GBJ*, voire *BJ* n'a pas été testée.

De même que pour la vérification de consistance avant, l'utilisation d'un algorithme de filtrage en pré-traitement de la recherche rend les méthodes de retour-arrière informé inintéressantes.

6.3 Mémorisation

Lors d'une vérification de consistance, les algorithmes rencontrent fréquemment des inconsistances locales entre des variables – généralement entre deux variables puisque les consistances utilisées sont du type 2-consistance. Si *GT* ne tient pas compte de ces informations, il est possible de les utiliser afin de ne pas vérifier deux fois la même inconsistance. Cette mémorisation de nouvelles contraintes induites se fait au fil de la recherche. Selon les auteurs, ces méthodes se regroupent sous le terme de « directionnelle-k-consistance » (*DkC*) [Dec 90], ou de *nogood recording* [SV 93]. Il a été montré que la méthode est intéressante pour un niveau de consistance d'au plus 2 (arc-consistance) car au-delà, la mémorisation devient trop coûteuse par rapport au gain apporté.

La mémorisation permet de diminuer le coût de la recherche en supprimant des branches à droite du nœud en cours d'instanciation, sur toute la largeur de l'arbre (voir figure 1.7). Elle a été expérimentée avec un mécanisme de retour-arrière chronologique (*BT*) et de type *CBJ* dans [Dec 90, SV 93], en exhibant des améliorations significatives par rapport à *BT*. Mais les hybridations avec une recherche par consistance avant n'ont pas été explorées.

À l'instar des méthodes de consistance en avant, la mémorisation est incompatible avec le pré-traitement par filtrage puisque dans ce cas, les inconsistances sont détectées – et mémorisées – avant la recherche.

6.4 Marquage

À l'intérieur d'une même branche d'instanciation, un certain nombre de travaux sont répétés, notamment des vérifications de consistance partielle entre valeurs. Diverses techniques ont été définies pour limiter cette redondance. Nous classons dans les méthodes de marquage, les algorithmes qui mémorisent localement[20] de l'information : le *backchecking* (*BC*) [HE 80] et le *backmarking* (*BM*) [Gas 77].

La technique du *backchecking* consiste à mémoriser les inconsistances locales entre valeurs. Si par exemple, lors de l'instanciation de $\langle X_j, b\rangle$ il apparaît que cette valeur est incompatible avec $\langle X_i, a\rangle$, *BC* marquera l'incompatibilité pour ne pas réessayer $\langle X_j, b\rangle$ dans une autre sous-branche de l'instanciation $\langle X_i, a\rangle$. En fait, on peut considérer que

[20] c.-à-d. sur une même branche d'instanciation, par opposition aux techniques de mémorisation qui stockent de l'information pour toutes les branches à venir.

BC effectue le même filtrage que *FC* en supprimant des valeurs dans les domaines futurs d'une instanciation; la différence vient du fait que *FC* détecte l'inconsistance au moment de l'instanciation la plus haute, alors que *BC* ne la détecte qu'au moment de l'instanciation la plus basse.

L'algorithme *BM* est une évolution de *BC* puisqu'il mémorise, non seulement les inconsistances entre valeurs sur une même branche d'instanciation, mais aussi les consistances, afin de ne pas les vérifier à nouveau. On pourra ainsi considérer que *BM* inclut deux comportements : un marquage négatif pour les inconsistances – qui correspond à *BC* – et un marquage positif pour les consistances.

Le marquage négatif élague l'arbre de recherche sur les branches à droite du nœud sur lequel a été rencontrée l'inconsistance, de la même manière que *DkC*, quoique sur une portée plus faible. Le marquage positif n'a pas d'influence sur l'élagage de l'arbre de recherche, mais limite le coût interne de chaque nœud d'instanciation. De plus, si chacune des deux techniques a un effet positif avec une vérification de consistance en arrière, le marquage négatif est totalement redondant avec un schéma de consistance avant. Des expérimentations ont été menées sur l'hybridation du marquage positif avec *FC* [Pro 93b, Pro 95a], et ont montré un bénéfice très faible.

6.5 Retardement

La technique de retardement consiste à différer certaines vérifications de consistance dans l'espoir de diminuer globalement le coût de la recherche en les regroupant judicieusement. Cette notion apparaît dans le *minimal forward-checking* (*MFC*) [DM 94b] : l'idée de base est que, si le problème est satisfiable, il ne sera pas nécessaire de filtrer la totalité des domaines en avant pour aboutir à une solution. L'algorithme met donc en suspens les vérifications de consistance non prioritaires requises par *FC* pour ne les exécuter que lorsqu'elles deviennent nécessaires. Naturellement, on ne peut espérer qu'un bénéfice réduit de la méthode si on cherche toutes les solutions au problème, ou si le problème n'admet pas de solutions – ce qui nécessite le parcours de la totalité de l'arbre de recherche.

On notera que l'algorithme *MFC* présenté dans [DM 94b] inclut un marquage positif et négatif (*BM*). Comme la technique peut s'appliquer à d'autres algorithmes que *FC*, nous l'appellerons *Mxx*.

Le retardement revêt un caractère particulier dans les techniques d'amélioration de la recherche car il peut être très positif dans le cas où il est appliqué à une technique de consistance en avant (comme *FC*), mais il devient négatif dans le cas de *BT* puisqu'il tend à le rapprocher de la méthode de génération totale de *GT*. L'effet de *Mxx* ne porte pas directement sur l'élagage de l'arbre, mais sur la minimisation du travail réalisé à l'intérieur d'un nœud d'instanciation.

6.6 Ordonnancement des variables

Les algorithmes énumératifs instancient une variable à la fois. Ceci induit un ordre dans l'instanciation des variables. Cette notion est totalement neutre pour *GT* puisque toutes les variables sont instanciées avant que n'intervienne la phase de test. Par contre, elle devient prépondérante dans le cas des algorithmes qui vérifient une consistance à

chaque instanciation. En effet, l'ordre des variables détermine la « forme » de l'arbre de recherche, et donc sont coût. Les heuristiques d'ordonnancement des variables les plus courantes sont :

- *Degré maximum*: la variable de plus grand degré (plus grand nombre de voisins) est instanciée d'abord. Cette technique prend le nom de « *maximal cardinality ordering* » (*MCO*) [DM 94a] dans la littérature anglophone. Sa motivation repose sur l'idée que les variables liées par un grand nombre de contraintes sont susceptibles de générer plus de retours-arrière, et donc doivent être instanciées en premier.

- *Largeur minimum*, ou *minimal width ordering* (*MWO*) : les variables sont instanciées dans l'ordre croissant de la largeur (*cf.* définition 1.16) [Fre 82]. Cette méthode est un raffinement de *MCO* car elle reprend l'idée du regroupement des retours-arrière en haut de l'arbre de recherche, mais s'appuie en plus sur les résultats théoriques du théorème de Freuder [Fre 82].

- *Domaine minimal* ou *minimum remaining values* (*MRV*) : la variable qui a le plus petit domaine est instanciée d'abord. On notera que *MRV* a un comportement différent selon le type de consistance vérifiée : avec une consistance arrière, l'ordre sera statique puisque les domaines ne sont modifiés qu'au moment de l'instanciation, alors qu'avec une consistance en avant, l'ordre sera dynamique. Des études expérimentales ont montré que *MRV* était une heuristique d'ordre efficace pour les algorithmes de type *FC* [BvR 95].

L'ordonnancement des variables est souvent appelé *ordre vertical* car il détermine l'aspect de l'arbre de recherche de bas en haut. D'une manière générale, les heuristiques d'ordre vertical développent le principe *first-fail*: on privilégie les variables qui mettront potentiellement en évidence une inconsistance au plus tôt.

6.7 Ordonnancement des valeurs

Lors de l'instanciation d'une variable, les valeurs du domaine sont successivement testées. Il peut être utile d'utiliser un ordre particulier afin de diminuer le coût de la recherche. L'ordre de test des valeurs est souvent appelé *ordre horizontal.*

D'un point de vue théorique, l'ordre horizontal est très important car il caractérise le non-déterminisme des algorithmes NP-complets. En effet, la résolution des problèmes NP-complets est de complexité exponentielle en grande partie car on ne connaît pas de technique de choix d'affectation des variables qui permettrait de trouver une solution « du premier coup », ce qui impose de faire des tests successifs aléatoires, jusqu'à obtention de la solution.

En pratique, les heuristiques d'ordre horizontal sont très peu explorées du fait de la difficulté de concevoir une méthode efficace. On pourra citer une méthode consistant à choisir les valeurs en fonction du nombre de leurs supports le long des contraintes. Dans cette direction, l'heuristique *MC* (pour *min-conflict*) choisit les valeurs dans l'ordre croissant du nombre de conflits avec des valeurs non-instanciées. D'autres heuristiques portent sur la taille des domaines futurs, mais il a été démontré que *MC* est la plus efficace avec une recherche par consistance avant [FD 95]. Par contre, ce type d'heuristique n'apporte que peu d'amélioration pour les algorithmes de recherche en arrière.

6.8 Ordonnancement des contraintes

Pour vérifier la consistance d'une valeur (que ce soit en consistance avant ou arrière), Il est nécessaire de parcourir et vérifier individuellement toutes les contraintes concernées. Ceci induit naturellement un ordre particulier (que nous appellerons *ordre transversal*) qui n'influe pas directement sur la forme de l'arbre de recherche, mais sur le coût de vérification de chaque nœud.

Dans le cas où l'instanciation est consistante, l'ordre transversal ne joue aucun rôle puisque toutes les contraintes devront être testées. Mais si l'instanciation est inconsistante, il devient préférable de s'en rendre compte au plus tôt. On obtient généralement des améliorations notables si on utilise un ordre transversal de type *first-fail* sur la satisfiabilité des contraintes : on choisit alors de vérifier en premier les contraintes les moins satisfiables.

7 Classification des CSP

Devant la grande diversité des problèmes de satisfaction de contraintes, il est apparu rapidement nécessaire de les classifier par grandes catégories. Si les plus anciens travaux portent essentiellement sur l'isolation de classes polynômiales, les travaux plus récents se concentrent principalement sur l'observation expérimentale du coût de résolution pour un algorithme donné. Nous distinguerons naturellement les deux approches dans ce paragraphe.

7.1 Classes polynômiales

Comme on l'a vu, l'établissement de la plupart des consistances locales se fait en temps polynômial – sauf la relationnelle-k-consistance. De plus, pour certains problèmes, le filtrage par consistance locale permet une résolution en temps polynômial. Il en résulte alors que la résolution du CSP initial est elle-même polynômiale. Nous distinguons dans ce cadre deux catégories de CSP : les classes caractérisées par la structure du graphe, et les classes caractérisées par le contenu des relations.

7.1.1 Classes polynômiales structurelles

L'application du théorème de Freuder (*cf.* paragraphe 4.1) permet de définir une classe polynômiale : pour un CSP de largeur 1 (un arbre) l'arc-consistance garantit une recherche de solution sans retour-arrière, et donc polynômiale. On serait tenté de l'appliquer pour une largeur k, qui impose l'établissement d'une k-consistance en $O(n^k d^k)$. Or, cette k-consistance peut faire augmenter la largeur du graphe, et donc ne pas garantir une recherche sans retour-arrière.

Le théorème de Freuder a été étendu aux CSP n-aires [Jég 91], ce qui permet de définir une classe polynômiale n-aire :

Théorème 1.27 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *dont l'hypergraphe est acyclique peut être résolu en temps polynômial.*

7.1.2 Classes polynômiales sémantiques

Le premier résultat est dû à Dechter, dans [Dec 92]. Il porte sur la cardinalité des domaines des variables.

Théorème 1.28 *Un* CSP *dont les domaines sont tous de cardinal inférieur à d, dont les contraintes sont toutes d'arité inférieure à a et qui est fortement $(d\times(a-1)+1)$-consistant est aussi globalement consistant.*

L'application la plus simple de ce théorème concerne les CSP binaires dont les variables ont des domaines de taille 2 (booléen par exemple). Dans ce cas, la forte 3-consistance suffit pour garantir la consistance globale, et donc une recherche de solution sans retour-arrière. Ce résultat est déjà connu dans le monde de la logique propositionnelle puisqu'il correspond dans ce cadre, à la satisfaction de 2-clauses.

Une seconde caractérisation porte sur la structure des contraintes [CCJ 94].

Définition 1.29 *Une contrainte $C_k = \{X_i, X_j\}$ est dite* **0/1/tous directionnelle** *si*

$$\forall x(\exists y \exists z((x,y) \in R_k \wedge (x,z) \in R_k \wedge y \neq z) \Rightarrow \forall w \in D_j((x,w) \in R_k)).$$

Plus simplement, une contrainte est *0/1/tous directionnelle* si une valeur dans un domaine est liée à aucune, une seule, ou toutes les valeurs du domaine opposé.

Théorème 1.30 *Un* CSP *binaire $\mathcal{P} = (X, D, C, R)$ dont toutes les contraintes sont de type* 0/1/tous directionnel *peut être résolu en temps polynômial.*

7.2 Caractérisation expérimentale : la transition de phase

Comme il est impossible de déterminer de façon théorique quelles sont les techniques de parcours d'arbre les moins coûteuses, en particulier dans le cas d'algorithmes hybrides, de nombreux auteurs ont expérimenté leurs algorithmes sur des jeux d'essai. Si dans les années 80 les problèmes d'école comme le placement des n-reines (exemple 1.1) étaient en vogue pour tester les algorithmes, certains auteurs se sont rendus compte qu'ils n'étaient pas représentatifs de tous les problèmes réels, et que de plus, les expérimentations donnaient des résultats parfois divergents selon le problème choisi [DM 94a, GW 94, Pro 94]. D'un autre coté, les schémas traditionnels de génération aléatoire [Bes 94, MH 86, Wal 93] laissaient apparaître de grosses différences entre les problèmes, sans pour autant permettre une caractérisation de ces difficultés locales.

Un schéma de génération aléatoire cohérent a été présenté simultanément par plusieurs auteurs dans le cadre des CSP. Celui-ci est en fait un « héritage » de l'étude du problème SAT dans le cadre de la logique propositionnelle [GW 94]. Il a subi quelques aménagements, mais la version qui est maintenant utilisée par la plupart des auteurs est celle de Prosser [Pro 94], que nous présenterons ici.

Le schéma de génération aléatoire concerne les CSP binaires. Il définit 4 paramètres : $\langle n, m, p_1, p_2\rangle$, où :

- n est le nombre de variables dans le CSP.

- m est le nombre de valeurs par variable. Dans ce schéma, toutes les variables ont un domaine identique.

- p_1 est la densité du graphe de contraintes. Ainsi, le CSP sera constitué de $p_1 \times \frac{n(n-1)}{2}$ contraintes binaires (= probabilité que deux variables appartiennent à une contrainte).

- p_2 est le taux de conflits dans chaque contrainte. Ce taux est uniforme pour toutes les contraintes et implique que chacune rejettera $p_2 \times m^2$ combinaisons.

Définition 1.31 *La* **satisfiabilité** *d'une contrainte* $C_k = \{X_{k_1}, \ldots, X_{k_{n_k}}\}$ *(notée* $\mathcal{S}_{C_k}$*) correspond au rapport entre le nombre de n-uplets autorisés et le nombre de n-uplets possibles :*

$$\mathcal{S}_{C_k} = \frac{|R_k|}{X_{k_1} \times \ldots \times X_{k_{n_k}}}$$

Il apparaît naturellement que ce schéma ne prétend pas représenter tous les problèmes possibles, surtout à cause de sa trop grande régularité. Toutefois, il semble faire l'unanimité car il permet de cerner deux caractéristiques fondamentales des CSP :

1. la structure du graphe, qui se traduit par la densité des contraintes ;

2. la structure interne des contraintes, qui se traduit par le taux de conflits.

Le comportement le plus intéressant de ce schéma concerne la variation du taux de conflits des contraintes (p_2), les trois autres paramètres étant fixés. La figure 1.8 présente le comportement caractéristique du schéma de génération vis-à-vis du taux de satisfiabilité de l'échantillon de problèmes[21], et du coût de résolution des algorithmes.

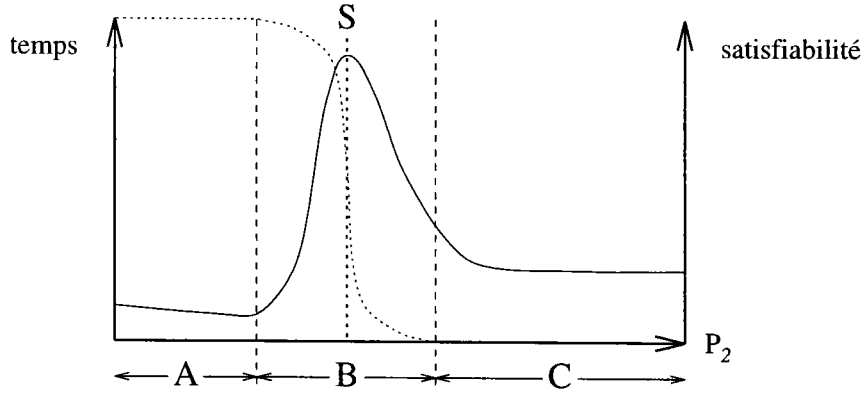


Figure 1.8 : Taux de atisfiabilité des échantillons et coût de résolution en fonction du taux de conflits p_2 des contraintes. La courbe de satisfiabilité est en pointillés, celle du coût de résolution en trait plein.

L'étude statistique des expérimentations fait apparaître 3 zones importantes – on se place ici dans le cas de la recherche d'une solution :

[21]Un problème est dit *satisfiable* s'il admet au moins une solution. Le taux de satisfiabilité de l'échantillon correspond au pourcentage de problèmes satisfiables.

A. Le taux de conflits est trop faible pour qu'il y ait des problèmes insatisfiables. De plus, la densité de solutions est importante, si bien que les algorithmes de recherche mettent peu de temps en moyenne pour en trouver une. Cette zone est appelée *facile-satisfiable.*

B. L'échantillon comporte à la fois des problèmes satisfiables et non satisfiables. Les problèmes qui le sont, comportent peu de solutions et les algorithmes mettent beaucoup de temps pour en trouver une, ou pour montrer qu'il n'y en a pas. Cette zone est donc *difficile.*

C. Le taux de conflits est trop important pour que les problèmes soient satisfiables. De plus, les blocages apparaissent tôt dans l'arbre de recherche qui n'est alors parcouru que sur une faible profondeur, d'où un faible coût moyen de résolution. Cette zone est appelée *facile-insatisfiable.*

La zone difficile est naturellement la plus étudiée car c'est elle qui demande le plus d'efforts de résolution. Les plus gros efforts de recherche ont notamment été concentrés sur le *seuil de transition* S – aussi noté $\widehat{p_2}$ – qui correspond à la valeur de p_2 pour laquelle 50% des problèmes sont satisfiables, car elle coïncide avec le point le plus coûteux de la recherche. La zone facile-insatisfiable est le plus souvent délaissée car peu de problèmes réels sont supposés appartenir à cette catégorie. Dans le domaine des algorithmes complets, seuls quelques auteurs ont signalé des cas particuliers dans la zone facile-satisfiable[22] [Pro 96, Smi 94].

On notera que les méthodes incomplètes de recherche (*cf.* paragraphe 2.2) ne peuvent montrer leur efficacité que sur la zone facile-satisfiable, ce qui justifie les nombreuses recherches pour améliorer les algorithmes complets dans la région difficile.

Conclusion

Dans ce chapitre, nous avons défini le cadre d'étude des problèmes de satisfaction de contraintes, ainsi que les méthodes de résolution généralement utilisées. Notre présentation s'est en fait concentrée sur la classe des algorithmes complets de recherche, *i.e.* qui répondent toujours et correctement – parfois lentement – à la question posée.

Historiquement, les CSP ont été définis avec des méthodes de résolution par filtrage, principalement la nœud-consistance et l'arc-consistance [Mon 74, Mac 77]. L'accent était alors mis sur les capacités de ces algorithmes à résoudre rapidement des problèmes de reconnaissance de formes (sur des graphes de contraintes binaires planaires). Différents algorithmes se sont succédés pour améliorer la complexité théorique et les performances pratiques du filtrage par arc-consistance.

Parallèlement à l'évolution des méthodes de filtrage, se sont développés un certain nombre d'algorithmes qui mettaient en œuvre des techniques de résolution directes paraissant difficilement comparables (*backjumping*, mémorisation, *forward-checking*, ...). Ce n'est que récemment [Pro 93b] qu'elles ont été exprimées dans un cadre de programmation commun afin de les comparer efficacement, et même d'exprimer leur hybridation

[22]des problèmes très difficiles apparaissent dans cette zone. Toutefois, cela semble être dû, plus à l'algorithme de recherche utilisé qu'à la caractérisation du problème.

possible en construisant des algorithmes qui tentent de combiner les avantages de plusieurs méthodes de base. Dans [Pro 93b], Prosser définit deux caractéristiques fondamentales des algorithmes de recherche : la descente et la remontée dans l'arbre de recherche. Pour cet état de l'art, nous avons affiné cette classification en y intégrant d'autres caractéristiques qui ressortent des algorithmes proposés dans la littérature (mémorisation, marquage, etc.). Ceci permet une classification plus précise des méthodes, et définit un « potentiel d'hybridation » plus important. Les avancées dans la caractérisation des CSP, et notamment la mise en évidence du phénomène de transition de phase, permettent de définir un cadre de comparaison cohérent de toutes les méthodes de résolution que les pourrait définir dans cette optique.

On peut résumer le cheminement algorithmique, depuis le CSP initial (binaire ou n-aire), jusqu'à la solution, par la figure 1.9.

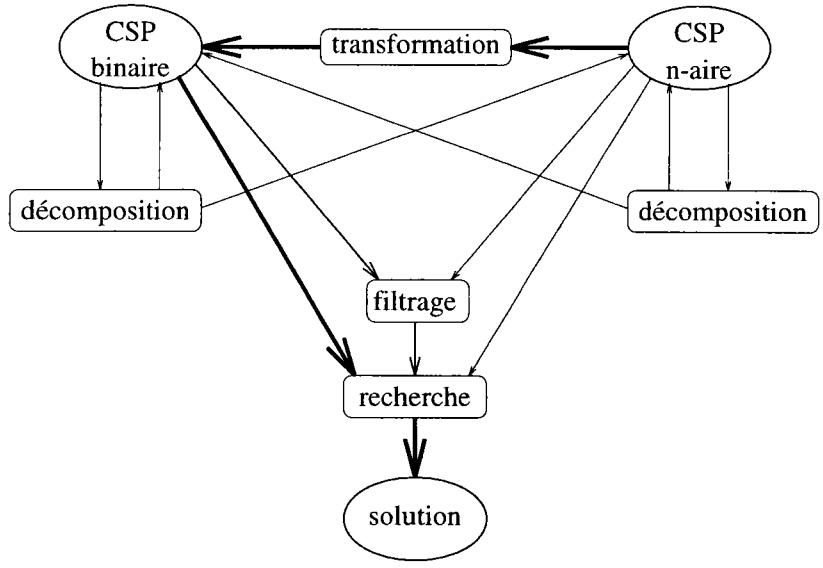


Figure 1.9 : Cheminement algorithmique pour la résolution complète des CSP binaires et n-aires.

Les principales techniques (transformation, filtrage et recherche) se trouvent ainsi agencées les unes par rapport aux autres selon leur mise en œuvre pratique. Toutefois, elles ne revêtent pas toutes le même intérêt pour les différents auteurs. En effet, la plupart d'entre eux privilégient la recherche – avec toutes ses améliorations – depuis un CSP binaire, en considérant que si le problème à résoudre est n-aire, une transformation aura été effectuée (flèches épaisses). Les techniques de décomposition (qu'elles soient n-aires ou binaires) sont étudiées, mais ne semblent pas être utilisées dans la pratique. Nous nous intéresserons par la suite à la résolution directe des CSP n-aires, sans passer par une phase de transformation binaire.

Chapitre 2

Algorithmes n-aires de recherche en avant

DANS LE CONTEXTE des CSP, la recherche sur les CSP n-aires reste peu utilisée pour la résolution complète : la voie de la transformation n-aire vers binaire prise dès la définition du formalisme CSP, a occulté l'algorithmique n-aire. Il est vrai que les algorithmes de recherche des années 70 pouvaient indifféremment traiter les contraintes binaires et les contraintes n-aires. Toutefois, l'émergence des algorithmes de recherche en avant (*FC* en tête), s'est faite sans tenir compte de la spécificité des hypergraphes de contraintes.

Très récemment, des auteurs se sont intéressés aux consistances n-aires [DvB 95, Jég 93], mais dans le seul cadre des algorithmes de pré-filtrage. Nous nous intéressons dans ce chapitre, à la définition d'un algorithme de recherche complet par consistance en avant, permettant de résoudre un CSP n-aire directement, sans transformation préalable.

La première partie de ce chapitre (paragraphe 1) dressera une comparaison systématique des principales définitions de consistance n-aires proposées dans la littérature, principalement l'hyper-k-consistance [Jég 91], et la relationnelle-k-consistance [DvB 95]. Elle mettra notamment en avant les limites et les conditions d'application de la première notion pour l'intégration dans un algorithme de recherche en avant. Par la suite (paragraphe 2), nous généraliserons la notion de (directionnelle) relationnelle-k-consistance pour sa mise en œuvre dans un algorithme de recherche en avant. Le paragraphe 3 donnera une implémentation de l'algorithme de *forward-checking* n-aire, puis nous présenterons diverses spécialisations d'algorithmes de recherche par consistance avant (paragraphe 4). Nous terminerons enfin ce chapitre (paragraphe 5) par une discussion sur les heuristiques d'ordre des variables pour les CSP n-aires.

1 Comparaison des consistances n-aires

Le but de ce paragraphe est de comparer dans un cadre de travail théorique les 3 définitions de consistance de CSP n-aires [MHHS 96]. Les deux principales définitions sont données dans le paragraphe 4.2 du chapitre 1 (l'hyper-k-consistance et la relationnelle-k-consistance), la troisième a été donnée par Jégou dans [Jég 91]:

Définition 2.1 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* k-**interconsistant**[1] *si et seulement si:*

- $\forall C_1, \ldots, C_k \in C$, $\forall j, 1 \leq j \leq k$, $(\bowtie_{1 \leq i \leq k} R_i)[C_j] = R_j$;
- $\forall C_i \in C, R_i \neq \emptyset$.

Nous donnons tout d'abord un résultat négatif quant à la comparaison des définitions originelles. Puis nous modifions quelque peu ces définitions afin qu'elles s'appliquent aux mêmes objets, et nous montrons qu'elles sont, là encore, incomparables. Enfin, nous dégageons une relation de *dualité* entre ces définitions, qui nous amène à les utiliser dans des cadres de recherche différents.

1.1 Étude comparative des définitions originelles

Les premières notions à comparer sont les deux définitions de [Jég 91]: l'hyper-k-consistance et la k-interconsistance. Puisqu'elles sont toutes deux des généralisations de l'interconsistance (consistance de niveau $k = 2$), elles sont bien évidement équivalentes pour $k = 2$. Jégou suggère qu'elles sont différentes, et donne un contre-exemple montrant la proposition suivante:

Proposition 2.2 ([Jég 91]) $\forall\ k > 2$, *k-interconsistance $\not\Rightarrow$ hyper-k-consistance.*

En fait, nous établissons le théorème suivant:

Théorème 2.3 *Hyper-k-consistance $\Rightarrow$ k-interconsistance.*

Preuve : Nous allons montrer la contraposée:
soit $\mathcal{P} = (X, D, C, R)$, un CSP non k-interconsistant, alors (1)

1. $\exists C_i \in C / R_i = \emptyset$ et donc $\mathcal{P}$ n'est pas hyper-k-consistant (d'après la définition 1.18), ou

2. $\exists C_1, ..., C_k \in C$ tels que $\exists j, 1 \leq j \leq k / (\bowtie_{1 \leq i \leq k} R_i)[C_j] \neq R_j$

   donc, ou bien $\exists r = (r_1, ..., r_{|C_j|})$ tel que $r \in (\bowtie_{1 \leq i \leq k} R_i)[C_j]$ et $r \notin R_j$ (impossible);

[1] la définition de la forte k-interconsistance est inutile puisque, si un CSP est k-interconsistant, il est aussi $(k-1)$-interconsistant (la preuve est donnée par Jégou dans [Jég 91]).

ou bien, $\exists r = (r_1, ..., r_{|C_j|})$ tel que $r \in R_j$ (2)

et $r \notin (\underset{1 \leq i \leq k}{\bowtie} R_i)[C_j]$. (3)

Supposons maintenant que $\mathcal{P}$ soit hyper-k-consistant, (4)
alors $\forall C_1, \ldots, C_k \in C, \forall j, 1 \leq j \leq k$,
$R_j[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j] \subseteq (\underset{1 \leq i \leq k}{\overset{i \neq j}{\bowtie}} R_i)[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j]$,
donc $r[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j] \in (\underset{1 \leq i \leq k}{\overset{i \neq j}{\bowtie}} R_i)[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j]$ et donc
$r[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j] \in (\underset{1 \leq i \leq k}{\overset{i \neq j}{\bowtie}} R_i)[(\underset{1 \leq i \leq k}{\overset{i \neq j}{\bigcup}} C_i) \cap C_j]$ (d'après (2)).
Et puisque $r \in R_j$ (d'après (2)), on déduit trivialement

$r \in (\underset{1 \leq i \leq k}{\bowtie} R_i)[C_j]$ (en contradiction avec (3))
et donc, $\mathcal{P}$ n'est pas hyper-k-consistant. □

Proposition 2.4 *Les relations suivantes sont vérifiées :*

- *(forte) hyper-k-consistance $\not\Rightarrow$ (forte) relationnelle-k-consistance ;*
- *k-interconsistance $\not\Rightarrow$ (forte) relationnelle-k-consistance.*

Preuve : La Figure 2.1 présente un contre-exemple :

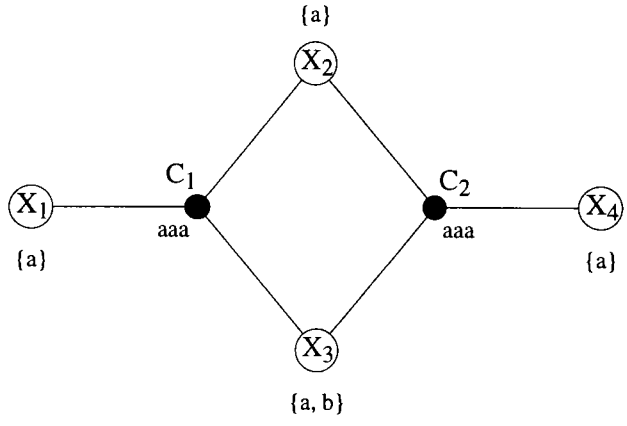


Figure 2.1 : Un CSP fortement hyper-2-consistant, non relationnel-2-consistant.

- $R_1[C_1 \cap C_2] = (aa) = R_2[C_1 \cap C_2]$, donc le CSP est fortement hyper-2-consistant, ainsi que 2-interconsistant.

– L'instanciation consistante $(X_1 = a, X_3 = b)$ ne peut être étendue à une instanciation consistante sur C_1, donc le CSP n'est pas relationnel-2-consistant.

Le contre-exemple est construit de telle sorte que toutes les relations aient été filtrées sans modifier les domaines des variables, et en laissant une valeur dans un domaine (b dans X_3) interdite par au moins une contrainte. □

Proposition 2.5 *Les relations suivantes sont vérifiées :*

– *(forte) relationnelle-k-consistance* $\not\Rightarrow$ *(forte) hyper-k-consistance ;*

– *(forte) relationnelle-k-consistance* $\not\Rightarrow$ *k-interconsistance.*

Preuve : La Figure 2.2 présente un contre-exemple :

– Le lecteur pourra facilement vérifier la relationnelle-2-consistance du CSP de la figure 2.2.

– $R_1[C_1 \cap C_2] = (aa) \neq (aa, ab) = R_2[C_1 \cap C_2]$, donc le CSP n'est pas hyper-2-consistant (faible et fort) ni 2-interconsistant.

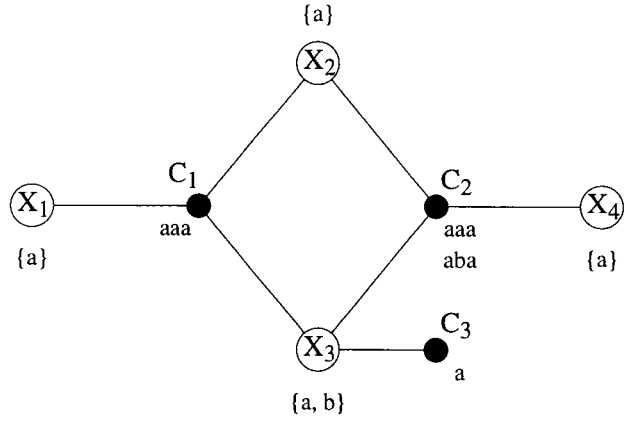


Figure 2.2 : Un CSP fortement relationnel-2-consistant, non hyper-2-consistant.

Le contre-exemple est construit de telle sorte que tous les domaines aient été filtrés (par des contraintes unaires) sans modifier les relations non unaires. □

Trois remarques peuvent être émises pour expliquer la différence entre hyper-k-consistance et relationnelle-k-consistance :

1. Tout d'abord, elles ne travaillent pas sur le même nombre de contraintes puisque la relationnelle-k-consistance filtre les sous-ensembles de $k-1$ contraintes, alors que l'hyper-k-consistance et la k-interconsistance filtrent les sous-ensembles de k contraintes. Ainsi, elles devraient être comparées avec le niveau k pour l'hyper-consistance, et le niveau $k+1$ pour la relationnelle-consistance. En fait, les mêmes contre-exemples pourraient être invoqués pour montrer la non-implication.

2. Dans les Figures 2.1 et 2.2, la relationnelle-2-consistance filtre les domaines des variables sans modifier les relations. De son coté, l'hyper-2-consistance filtre les relations sans modifier les domaines. Dans le cas général, la relationnelle-k-consistance filtre les domaines – en rajoutant des contraintes unaires – et un certain nombre de contraintes, mais pas toutes. À l'inverse, l'hyper-k-consistance filtre toutes les relations, mais laisse les domaines inchangés.

3. Dans la relationnelle-k-consistance, un réseau vide (*i.e.* au moins un domaine ou une relation vide) est considéré comme trivialement consistant (car $\emptyset \subseteq \emptyset$). Mais un réseau vide ne peut être hyper-k-consistant. Donc, les réseaux vides devraient être exclus de la définition de la relationnelle-k-consistance. On pourra noter au passage que le fait de considérer un réseau vide comme non-consistant est plus cohérent puisque celui-ci n'admet pas de solutions.

Dans le paragraphe suivante, nous prenons en compte ces remarques et présentons des définitions modifiées pour travailler dans le même champ d'application.

1.2 Consistances n-aires modifiées

Nous présentons de nouvelles définitions dans les deux axes de consistance (hyper-consistance et relationnelle-consistance) pour unifier les cadres de travail [MHHS 96].

Définition 2.6 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **hyper*-k-consistant** *si et seulement si*

1. $\forall C_1, ..., C_k \in C$, $(\bowtie_{1 \leq i \leq k-1} R_i)[(\bigcup_{1 \leq i \leq k-1} C_i) \cap C_k] = R_k[(\bigcup_{1 \leq i \leq k-1} C_i) \cap C_k]$;
2. $\forall C_i \in C, R_i \neq \emptyset$;
3. $\forall C_i \in C, \forall X_j \in C_i, R_i[X_j] = D_j$.

Cette définition correspond à l'hyper-k-consistance originelle, à laquelle on ajoute le troisième point qui projette l'effet du filtrage des relations sur les domaines des variables. Elle est en fait donnée informellement par Jégou dans [Jég 91].

Nous ne traitons pas ici la k-interconsistance puisqu'elle est induite par l'hyper-k-consistance (théorème 2.3).

Définition 2.7 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **relationnel*-k-consistant** *si et seulement si*

1. $\forall C_1, ..., C_{k-1} \in C$, $\forall X_j \in \bigcap_{1 \leq i \leq k-1} C_i$,

   $\rho((\bigcup_{1 \leq i \leq k-1} C_i) \setminus \{X_j\}) \subseteq (\bowtie_{1 \leq i \leq k-1} R_i)[(\bigcup_{1 \leq i \leq k-1} C_i) \setminus \{X_j\}]$;
2. $\forall C_i \in C, R_i \neq \emptyset$;
3. $\forall (C_i, C_j) \in C^2, C_j \subset C_i, R_i[C_j] = R_j$.

Le premier point est identique à la définition de la relationnelle-k-consistance. Le deuxième exclut les réseaux vides de l'ensemble des réseaux consistants. Le troisième point vérifie que les relations puissent être projetées sur les domaines, *i.e.* les effets du filtrage des relations subsummées sont reportés dans les relations plus générales.

Ces deux définitions qui appliquent leur filtrage à la fois sur les domaines et les relations peuvent être maintenant comparées dans une base commune.

Proposition 2.8 *(forte) hyper*-k-consistance $\not\Rightarrow$ (forte) relationnelle*-$(k+1)$-consistance.*

Preuve : La figure 2.3 présente un contre-exemple :

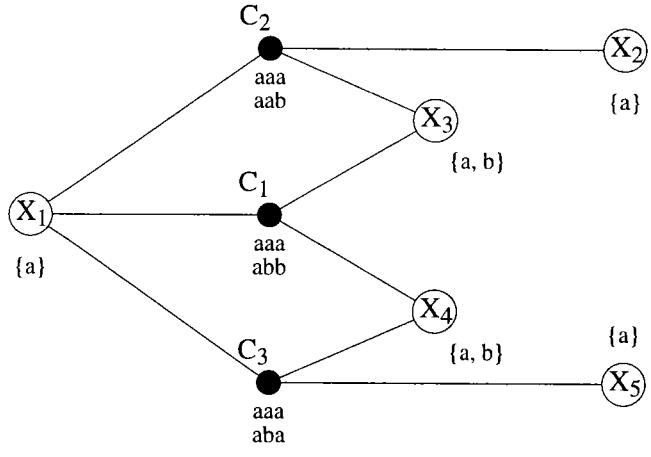


Figure 2.3 : Un CSP fortement hyper*-2-consistant, non relationnel*-3-consistant.

- Le lecteur pourra facilement vérifier la forte hyper*-2-consistance du CSP de la figure 2.3.
- $X_1 \in (C_1 \cap C_2 \cap C_3), A = \{X_2, X_3, X_4, X_5\} = (C_1 \cup C_2 \cup C_3) \setminus \{X_1\}$. L'instanciation consistante $i_A = (a, a, b, a)$ sur A ne peut être étendue à X_1 car (aab) $\notin C_1$, donc le CSP n'est pas relationnel*-3-consistant. □

Proposition 2.9 *(forte) relationnelle*-$(k+1)$-consistance $\not\Rightarrow$ (forte) hyper*-k-consistance.*

Preuve : La figure 2.4 présente un contre-exemple :

- Le CSP de la figure 2.4 est fortement relationnel*-4-consistant (la preuve est laissée au lecteur).
- $(R_1 \bowtie R_2)[(C_1 \cup C_2) \cap C_4] = (aa, ab, ba, bb) \neq (aa, ab, ba) = R_4[(C_1 \cup C_2) \cap C_4]$, donc le CSP n'est pas hyper*-3-consistant. □

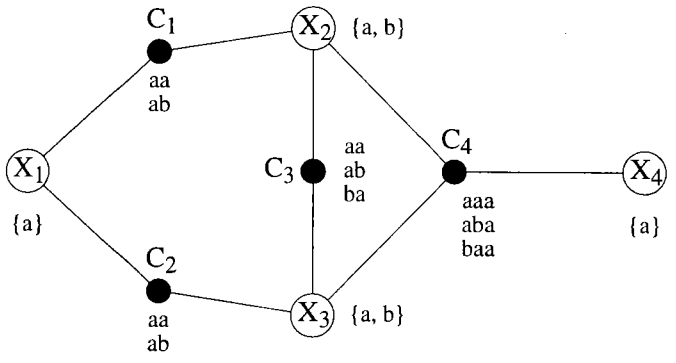


Figure 2.4 : Un CSP relationnel*-4-consistant, non hyper*-3-consistant.

1.3 Impact des consistances n-aires sur la recherche

L'intérêt principal des notions de consistance locale est de diminuer le coût de l'algorithme général de résolution. Nous comparons donc ici l'impact des différentes consistances n-aires sur les algorithmes de recherche. Rappelons tout d'abord que la complexité de ces algorithmes est en $O(d^n)$ où n est le nombre de variables et d est la taille du plus grand domaine (*cf.* chap. 1, paragraphe 1.1). Ceci implique que le principal objectif des méthodes de filtrage (induites par les notions de consistance locale), est de réduire l'incidence de n et d, par ordre d'importance décroissante. Comme il n'est pas possible de réduire le nombre de variables dans une solution, les définitions de consistance devront être jugées sur leur capacité à réduire la taille des domaines des variables (d).

Comme il est montré dans [DvB 95], la relationnelle-k-consistance est une technique de filtrage efficace pour *BT* puisqu'elle réduit les compatibilités combinatoires entre les valeurs des domaines des variables. Les habituelles considérations théoriques peuvent être appliquées sur cette définition, et en particulier le théorème de Freuder relatif à la largeur des réseaux de contraintes (théorème 1.17). Cette technique peut être considérée comme pertinente pour *BT* puisqu'elle réduit suffisamment les compatibilités entre les valeurs pour avoir un effet notable et prévisible sur le coût de la recherche. De plus, la relationnelle-k-consistance n'accomplit pas de travail inutile si on exclut la redondance induite par l'ordre d'instanciation. On notera d'ailleurs que cette redondance est supprimée dans le cas de la *relationnelle-k-consistance directionnelle* [DvB 95].

De ce même point de vue, l'hyper-k-consistance peut être considérée comme plutôt inutile puisqu'elle ne réduit pas les compatibilités entre les valeurs des domaines, mais le nombre de n_i-uplets dans chaque relation. L'hyper*-k-consistance est, à ce titre, plus efficace pour réduire la combinatoire de la recherche. Malheureusement, nous avons montré qu'elle est insuffisante pour garantir la relationnelle-k-consistance et ses propriétés inhérentes (*cf.* proposition 2.8).

En fait, les deux consistances n-aires ne portent pas sur le même chemin de résolution de la figure 1.9. La relationnelle-k-consistance constitue un filtrage direct des CSP n-aires, alors que l'hyper-k-consistance suppose, pour être efficace, une transformation binaire, comme dans la figure 2.5.

On peut donc considérer que le cadre de travail de l'hyper-k-consistance n'est pas directement un cadre n-aire, mais un cadre binaire sur le graphe dual du CSP d'origine

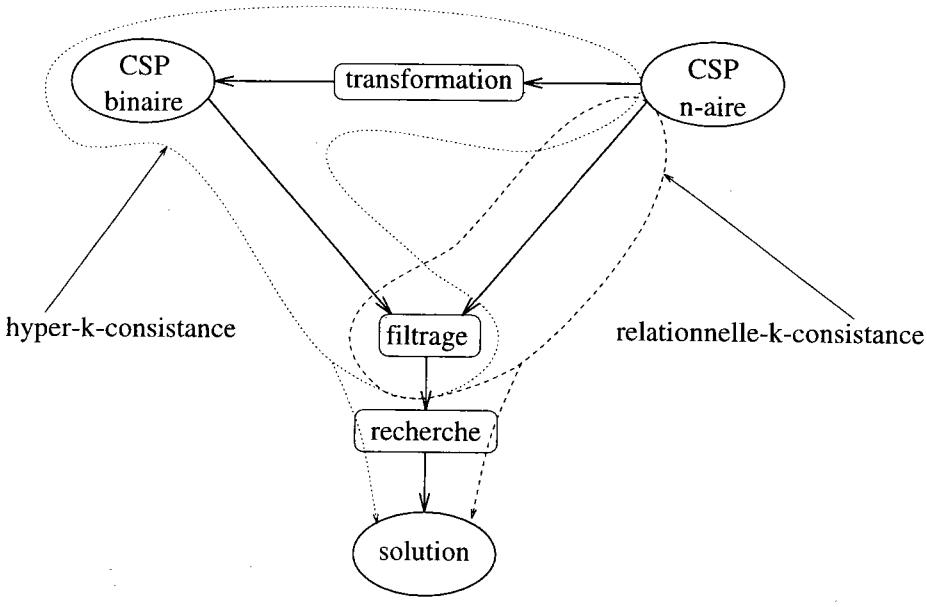


Figure 2.5 : Cheminement algorithmique des consistance n-aires.

(*cf.* définition 1.25). Les algorithmes de recherche n-aire que nous développerons seront donc plutôt basés sur une notion de consistance qui rejoint la relationnelle-k-consistance.

2 Consistance n-aire pour la recherche en avant

L'objet de ce chapitre est d'étendre aux CSP n-aires les algorithmes binaires de recherche par établissement d'une consistance en avant. Nous définissons dans ce paragraphe une notion de consistance restreinte qui permettra de concevoir ces algorithmes. L'approche choisie sera d'analyser finement le comportement et les propriétés de l'algorithme *FC* vis-à-vis des contraintes binaires pour en déduire une consistance restreinte, applicable à *FC* dans un réseau de contraintes n-aires (paragraphe 2.1). Nous en extrapolerons ensuite une définition de consistance généralisée dans la paragraphe 2.2, qui sera rattachée aux notions déjà présentées dans la littérature.

2.1 Consistance n-aire restreinte

Le principe de l'algorithme *FC*, dans sa version binaire, consiste à filtrer lors de chaque nouvelle instanciation, tous les domaines des variables non encore instanciées, de telle sorte que les valeurs restantes soient compatibles avec les instanciations déjà effectuées. Formellement, l'objectif de *FC* est de vérifier la propriété suivante :

Propriété 2.10 *Soit un* CSP $\mathcal{P} = (X, D, C, R)$ *et* $\{X^+, X^-\}$, *une bipartition de* X ($X^- = X \setminus X^+$). *Si les variables de* X^+ *sont instanciées de façon consistante par* A (i.e. $\forall C_i \subseteq X^+, A[C_i] \in R_i$) , *alors* $\forall C_i \in C, \forall X_j \in X^-, \forall v \in D_j, (A \cup \{X_j = v\})[C_i \cap (X^+ \cup X_j)] \in R_i[C_i \cap (X^+ \cup X_j)]$.

Dans le cas de contraintes binaires, la vérification d'une contrainte intervient lorsque la première des deux variables est instanciée. Il suffit alors de parcourir le domaine de

la variable non-instanciée et de supprimer les valeurs qui ne sont pas autorisées par la relation. La vérification de consistance se fera ici en O(1) pour chaque valeur.

Pour la définition d'un algorithme *FC* sur des contraintes n-aires, il sera nécessaire de vérifier la propriété 2.10 à chaque instanciation. Toutefois, la vérification d'une contrainte peut intervenir à différents moments lors du processus d'instanciation. La figure 2.6 présente un exemple des applications possibles de la vérification de la propriété 2.10 sur une contrainte 4-aire. Il est en effet possible de :

1. Filtrer les domaines des variables futures lors de l'instanciation de X_i. Dans ce cas, les trois variables X_j, X_k et X_l doivent être filtrées.
2. Filtrer les domaines des variables futures lors de l'instanciation de X_j. Là, les variables X_k et X_l sont concernées.
3. Filtrer les domaines des variables futures lors de l'instanciation de X_k. Dans ce cas, seul X_l sera filtrée.

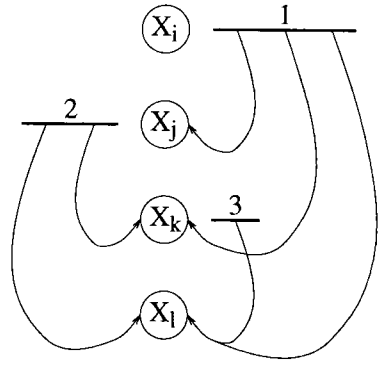


Figure 2.6 : Applications possibles de la propriété 2.10 sur une contrainte 4-aire.

Dans les cas (1) et (2), la consistance établie sur les variables futures ne sera pas totale, puisqu'elle ne dépendra que d'une partie de l'instanciation. Seule la consistance établie en (3) sera totale. On définit ainsi une notion de consistance relative :

Définition 2.11 *Soit un* CSP *n-aire $\mathcal{P} = (X, D, C, R)$ et $\{X^+, X^-\}$, une bipartition de X. Soit A, une instanciation consistante sur X^+. Une valeur $v \in D_j/X_j \in X^-$ est dite* **consistante relativement à** X^+ si et seulement si $\forall C_k/X_j \in C_k, v \in (A \bowtie R_k)[X_j]$.

Comme le nombre de variables non-instanciées au moment de la vérification de consistance est variable, nous introduisons de plus le nouveau concept suivant (qui n'avait pas de sens dans le cas binaire) :

Définition 2.12 *Soit une contrainte k-aire $C_j = \{X_{j_1}, \ldots, X_{j_k}\}$. Nous appelons* **profondeur de vérification**, *le nombre b ($b < k$) de variables non-instanciées au moment d'une vérification partielle.*

Lors d'une instanciation dans un algorithme de recherche par consistance en avant sur des contraintes n-aires, la vérification de la propriété 2.10 se fera par l'intermédiaire de l'établissement de la consistance relative. Dans la suite de ce paragraphe, nous allons étudier ce mécanisme, ainsi que son efficacité.

2.1.1 Établissement de la consistance relative

Dans le cas où $X^+ \cap C_k = \emptyset$, la formule $(A \bowtie R_k)[X_j]$ de la définition 2.11 est équivalente à $R_k[X_j]$, et peut donc être vérifiée statiquement avant la première instanciation. Ceci correspond à la nœud-consistance [Mac 77, Mon 74]. Par la suite, nous considérerons que le CSP originel est nœud-consistant.

Pour des raisons de simplicité de présentation, nous considérerons que les relations sont stockées sous forme de matrices booléennes[2]. De plus, nous partitionnons l'ensemble C_k en trois parties : 1) X_j qui est la variable instanciée au moment de la vérification, 2) C_k^+ qui correspond aux variables de C_k instanciées avant X_j, et 3) X_k^-, l'ensemble des variables de C_k non encore instanciées.

Ainsi, si $X^+ \cap C_k \neq \emptyset$, $v \in D_j$ sera consistant relativement à C_k^+ si $(A[C_k^+] \bowtie \{X_j = v\}) \in R_k[C_k^+ \cup X_j]$, et donc si $A \bowtie \{X_j = v\} \bowtie R_k \neq \emptyset$. De plus, comme $(A \bowtie \{X_j = v\} \bowtie R_k)[C_k^+ \cup X_j] \neq \emptyset$ – déjà vérifié par les instanciations précédentes – , seule la sous-matrice $(A \bowtie \{X_j = v\} \bowtie R_k)[C_k^-]$ doit être vérifiée. Le théorème suivant donne la complexité théorique d'une étape de filtrage pour une contrainte dans ce contexte.

Théorème 2.13 *Soit une contrainte k-aire $C_j = \{X_{j_1}, \ldots, X_{j_k}\}$ dans un* CSP $\mathcal{P} = (X, D, C, R)$. *L'établissement de la propriété 2.10 au moment de l'instanciation de la l^e variable de C_j* (i.e. *la profondeur de vérification est $b = k - l$*) *est en $O(bd^b)$ vérifications de bits.*

Preuve : Supposons que $l-1$ variables soient instanciées de manière consistante dans C_j. Soit A cette instanciation. Considérons de plus que $k = |C_j|$ et que X_s est la l^e variable à instancier dans C_j. La profondeur de vérification est alors $b = k - l$.

Pour être consistant relativement à A, une valeur $v \in D_s$ doit apparaître dans au moins un b-uplet de la sous-relation $(A \bowtie \{X_s = v\} \bowtie R_j)[C_j^-]$. Cette opération consiste en un « OU » logique sur une matrice de dimension b, et coûte $O(d^b)$ vérifications de bit.

De plus, l'établissement de la propriété 2.10 impose de supprimer toutes les valeurs inconsistantes dans les domaines futurs $D_t/X_t \in C_j^-$, relativement à $A \bowtie \{X_s = v\}$. Il y a $O(db)$ valeurs à tester, et chaque test consiste en un « OU » logique sur une matrice de dimension $(b-1)$ puisque un **vrai** doit se trouver dans la sous-matrice $(A \bowtie \{X_s = v\} \bowtie \{X_t = v_t\} \bowtie R_j)[C_j^- \setminus \{X_t\}]$. Ainsi, l'étape de filtrage de la contrainte C_j est en $O(bd) \times O(d^{(b-1)}) = O(bd^b)$ vérifications de bit. □

2.1.2 Efficacité du filtrage n-aire

Le but du filtrage lors d'un algorithme de recherche par consistance avant, est de réduire la taille des domaines futurs. Il est intéressant de quantifier l'effet d'un filtrage par rapport aux valeurs qu'il supprime. Nous établissons le théorème suivant :

Théorème 2.14 *Soit une contrainte k-aire $C_j = \{X_{j_1}, \ldots, X_{j_k}\}$ dans un* CSP $\mathcal{P} = (X, D, C, R)$. *Soit $\mathcal{S}_{C_j}$ la satisfiabilité de C_j*[3]. *Quand on filtre les domaines futurs de C_j pour une profondeur de vérification de b ($b < k$), chaque valeur dans les domaines des variables non-instanciées est supprimée avec une probabilité $(1 - \mathcal{S}_{C_j})^{d^{(b-1)}}$.*

[2] cette représentation est d'ailleurs optimale en terme de complexité temporelle.

[3] la satisfiabilité d'une contrainte k-aire se définit comme le rapport entre le nombre de k-uplets qu'elle autorise, et le nombre de combinaisons possibles entre les valeurs des variables qu'elle lie.

Preuve : En premier lieu, la probabilité qu'un k-uplet de la relation soit invalide est de $(1 - \mathcal{S}_{C_j})$. Lorsque $(k - b)$ variables sont instanciées dans C_j – appelons A l'instanciation correspondante – une valeur v est supprimée d'un domaine futur D_m si aucun k-uplet n'apparaît dans la relation $R_{jm} = A \bowtie \{X_m = v\} \bowtie R_j$. Comme la taille de R_{jm} est en $O(d^{(b-1)})$, la probabilité que $R_{jm} = \emptyset$ est de $(1 - \mathcal{S}_{C_j})^{d^{(b-1)}}$. □

Remarque : Le théorème 2.14 suppose que la contrainte C_j est homogène, *i.e.* que les projections de la relation R_j préservent la satisfiabilité $\mathcal{S}_{C_j}$ sur chaque sous-relation de R_j. Cette hypothèse est émise pour des raisons de simplification puisque dans une contrainte hétérogène, la probabilité pour une valeur d'être supprimée sera égale au produit des satisfiabilités des sous-contraintes concernées. ◇

Un ratio important influencera le coût pratique des algorithmes de recherche par consistance en avant :

$$\frac{\text{travail du filtrage}}{\text{coût du filtrage}}$$

L'objectif global de la recherche sera de maximiser ce rapport pour une efficacité optimale du filtrage. Or, le travail effectué par le filtrage décroît exponentiellement avec la profondeur de vérification (théorème 2.14), alors que son coût croît exponentiellement (théorème 2.13).

2.2 Consistance n-aire généralisée

Lorsque l'on combine les comportements opposés du coût de vérification et de l'efficacité du filtrage relativement à la profondeur de vérification, il devient évident que les algorithmes de recherche par consistance avant n-aires auront intérêt à limiter la profondeur de vérification des contraintes, en fonction de leur satisfiabilité. Nous proposons une définition qui limite l'application de la consistance relative.

Définition 2.15 *Soient un* CSP $\mathcal{P} = (X, D, C, R)$ *et* $A = (a_1, \ldots, a_i)$ *une instanciation consistante sur* $I = \{X_1, \ldots, X_i\}$. $\mathcal{P}$ *est dit* b-**consistant** *si et seulement si* $\forall C_k \in C / |\gamma(C_k)| \leq b \wedge C_k \cap I \neq \emptyset, \forall X_j \in \gamma(C_k), \forall v \in D_j, v \in A \bowtie R_k[X_j]$.[4]

Plus simplement, la consistance relative ne s'applique que pour une profondeur bornée. Puisque la définition de la b-consistance est une extension de l'arc-consistance binaire établie dans *FC*, la proposition suivante peut être énoncée :

Proposition 2.16 *Soit un* CSP *binaire* $\mathcal{P} = (X, D, C, R)$. *Appliquer la b-consistance avec* $b \geq 1$ *dans un algorithme de recherche en avant sur* $\mathcal{P}$ *est équivalent à l'algorithme FC binaire sur* $\mathcal{P}$.

Preuve : Si $\mathcal{P}$ est un CSP binaire, une contrainte C_k est vérifiée si $\gamma(C_k) = 1$ et donc un seul D_j est filtré tel que $v \in D_j, v \in A \bowtie R_k[X_j]$, ce qui correspond à la vérification de consistance dans *FC*.

Quand $b > 1$, la cardinalité de $\gamma(C_k)$ n'est plus bornée par 1, mais les contraintes ne peuvent pas avoir 1 variable instanciée et plus d'une qui ne le soit pas. Ainsi, l'utilisation

[4] $\gamma(C_k)$ est l'ensemble des variables non instanciées de C_k : $\gamma(C_k) = C_k \setminus (C_k \cap I)$.

de la b-consistance avec une profondeur supérieure à 1 ne change pas la vérification des contraintes binaires. □

Le choix d'une profondeur de vérification optimale est difficile sur le plan théorique car il suppose le calcul de l'efficacité du filtrage qui est fortement dépendante des contraintes. Nous aborderons dans le chapitre 3 une étude expérimentale qui permettra d'observer cet optimal sur quelques exemples.

La proposition 2.16 suggère que la consistance relative pour les contraintes n-aires est analogue à l'arc-consistance pour les contraintes binaires. Il est donc possible de généraliser cette définition pour un nombre de contraintes impliquées plus grand. En fait, il existe déjà dans la littérature, deux extensions pour la k-consistance : la relationnelle-k-consistance [DvB 95] porte sur l'extension de l'arité, alors que la (i,j)-consistance [Fre 85] porte sur le nombre de variables anticipées. Or, la consistance relative concerne à la fois des contraintes n-aires, et l'anticipation de l'instanciation sur plus d'une variable. On peut donc définir :

Définition 2.17 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **relationnel-(i,j)-consistant** *si et seulement si* $\forall C_1, \ldots, C_{i-1} \in C, \forall Y = \{X_{i_1}, \ldots, X_{i_j}\} \subset \bigcap_{1 \leq k \leq i-1} C_k$,

$$\rho((\bigcup_{1 \leq k \leq i-1} C_k) \setminus Y) \subseteq (\bowtie_{1 \leq k \leq i-1} R_k)[(\bigcup_{1 \leq k \leq i-1} C_k) \setminus Y].$$

Définition 2.18 *Un* CSP $\mathcal{P} = (X, D, C, R)$ *est* **fortement relationnel-(i,j)-consistant** *si et seulement si* $\forall i', 1 \leq i, \leq k, \mathcal{P}$ *est relationnel-(i',j)-consistant.*

Ceci amène donc à une classification plus poussée des notions de consistance n-aires (*cf.* table 2.1). On distinguera deux axes principaux de variation : 1) l'arité des contraintes sur lesquelles portent les définitions, et 2) la profondeur d'anticipation. Dans ce contexte, notre définition de relationnelle-(i,j)-consistance se place comme la notion la plus générale.

Table 2.1 : Classification des notions de consistance. On place en colonne le niveau d'anticipation, et en ligne l'arité des contraintes.

arité	profondeur = 1	profondeur = $j > 1$
binaire	k-consistance [Mac 77]	(i,j)-consistance [Fre 85]
n-aire	relationnelle-k-consistance [DvB 95]	relationnelle-(i,j)-consistance

3 Forward-checking n-aire

Nous définissons dans ce paragraphe, l'algorithme de *forward-checking* n-aire qui met en œuvre la b-consistance (*bFC*). Nous donnons d'abord le principe général de *bFC* (paragraphe 3.1), puis les structures de données et fonctions utilisées (paragraphes 3.2 et 3.3), avant de détailler l'algorithme lui-même dans le paragraphe 3.4.

3.1 principe général de l'algorithme

L'algorithme *bFC* est implémenté selon le schéma d'hybridation de Prosser [Pro 93b]. La fonction principale (résolution) est générique quant à la méthode de recherche utilisée dont l'implémentation se retrouve divisée en deux fonctions de base :

- *Descente* qui réalise une instanciation ;
- *Remontée* qui annule une instanciation.

On notera toutefois, que le réseau de contraintes est représenté par une structure un peu différente de celle de Prosser car les contraintes ne sont pas définies par une paire de variables (possible pour les contraintes binaires), mais par un numéro. Ceci impose d'orienter l'ordre de vérification des contraintes, non plus par les variables futures, mais par les contraintes elles-mêmes.

Algorithme 2.1 *Résolution d'un* CSP.

```
1  fonct Résolution (n) ↦ état ≡
2    consistant ⟵ vrai
3    état ⟵ inconnu
4    i ⟵ 1
5    tant que état = inconnu faire
6              si consistant alors ⟨i, consistant⟩ ⟵ Descente (i, consistant)
7              sinon ⟨i, consistant⟩ ⟵ Remontée (i, consistant) finsi
8              si i > n alors état ⟵ solution
9              sinon si i = 0 alors état ⟵ impossible finsi finsi fintant que
10   retourner état.
```

La fonction *Résolution* (algorithme 2.1) appelle successivement les fonctions *Descente* ou *Remontée*, jusqu'à ce que le CSP soit résolu, ou prouvé insatisfiable. Si *Résolution* s'apprête à instancier la variable X_{n+1}, alors toutes les variables sont instanciées de manière consistante, et le CSP est satisfiable (ligne 8). À l'inverse, si la fonction s'apprête à instancier la variable X_0, le CSP est insatisfiable (ligne 9).

Nous ne traitons pas dans cet algorithme de l'implémentation des heuristiques d'ordre sur les variables afin de ne pas alourdir l'écriture. Toutefois, leur inclusion dans ce schéma peut se faire aisément par renommage des variables.

3.2 Structures de données

La plupart des structures de données sont globales pour ne pas alourdir les passages de paramètres. Pour chaque tableau, nous donnons la complexité spatiale. Dans cette analyse, n est le nombre de variables, d est la taille du plus grand domaine, et m est le nombre de contraintes. Les tableaux sont :

- cnt[i] ($1 \leq i \leq n$) est la liste de toutes les contraintes portant sur la variable X_i. La complexité spatiale est en $O(nm)$.

- var[i] ($1 \leq i \leq m$) est la liste de toutes les variables sur lesquelles porte C_i. La complexité spatiale est en $O(mn)$.

- dom[i] ($1 \leq i \leq n$) est la liste des valeurs de D_i compatibles avec les instanciations déjà effectuées. Avant de résoudre le CSP, dom[i] est initialisé à D_i. La complexité spatiale est en $O(nd)$.

- v[i] ($1 \leq i \leq n$) contient l'instanciation courante de la variable X_i. La complexité spatiale est en $O(n)$.

- red[i] ($1 \leq i \leq n$) est une liste de paires de la forme $\langle X_j, v\rangle$ où X_j est une variable, et v est une valeur. Par exemple, red[3] = $(\langle 4,1\rangle, \langle 7,3\rangle)$ signifie que la valeur 1 de la variable X_4 et la valeur 3 de la variable X_7 ne sont pas compatibles avec l'instanciation courante de X_3 et ont été supprimées de dom[4] et de dom[7]. Chaque variable de red[i] appartient au futur de X_i (*i.e.* vient après X_i dans l'ordre d'instanciation). La complexité spatiale est en $O(n^2d)$.

3.3 Fonctions globales

Certaines fonctions sont utilisées dans les algorithmes, mais leur implémentation n'y est pas détaillée. Nous présentons informellement leur comportement, ainsi que leur complexité.

- *Prof_ Cour* (c, i) retourne la profondeur de vérification de la contrainte C_c au moment où la variable X_i est instanciée. Si les variables sont ordonnées dynamiquement, la complexité de la fonction *Prof_ Cour* est en $O(k)$, si k est l'arité de la contrainte C_c. Si un ordre statique est utilisé sur les variables, la fonction sera implémentée en temps constant.

- *Prof_ Max* (c) retourne la profondeur de vérification maximale autorisée pour la contrainte C_c. Cette valeur est supposée constante pour chaque contrainte.

- *Vérifie* (c, i) retourne la consistance de la contrainte C_c relativement aux variables déjà instanciées ($\forall X_j / j \leq i$). La complexité temporelle de la fonction *Vérifie* est en $O((\textit{Prof\_ Cour}\ (c,i) - 1)d^{\textit{Prof\_ Cour}\ (c,i)-1})$ selon le théorème 2.13.

3.4 Algorithme *bFC*

L'implémentation des fonctions de descente et de remontée dans l'arbre de recherche appelées dans la fonction *Résolution* de l'algorithme 2.1 dépend du type de recherche désiré. Nous nous intéressons dans ce paragraphe à *bFC*. L'algorithme 2.2 présente la fonction *Descente_ bFC* qui instancie une variable en respectant le principe de *forward-checking* n-aire développé plus haut.

La fonction *Descente_ bFC* cherche dans le domaine valide de X_i, une valeur qui soit compatible avec les variables non-instanciées – les valeurs de dom[i] sont déjà compatibles avec les variables instanciées. Pour cela, elle teste la consistance en avant de toutes les contraintes C_c qui portent sur X_i (lignes 5–6). Si l'une des contraintes invalide l'instanciation courante, la valeur est supprimée du domaine courant de X_i (ligne 8) et les réductions dans les domaines futurs provoquées par v[i] sont annulées (ligne 9). Dans le cas où la

variable a pu être instanciée de manière consistante, *Descente_bFC* renvoie le numéro de la prochaine variable à instancier.

Algorithme 2.2 *Instanciation d'une variable.*

```
1  fonct Descente_bFC (i, cons) ↦ ⟨entier, booléen⟩ ≡
2    cons ⟵ faux
3    pour v[i] chaque élément de dom[i] tant que ¬cons faire
4         cons ⟵ vrai
5         pour c chaque élément de cnt[i] tant que cons faire
6              cons ⟵ Consistance_Avant (c, i) finpour
7         si ¬cons alors
8           dom[i] ⟵ dom[i] \ {v[i]}
9           Annuler_Réductions (i) finsi finpour
10   si cons alors i ⟵ i + 1 finsi
11   retourner ⟨i, cons⟩.
```

La fonction *Consistance_Avant* (algorithme 2.3) filtre les domaines des variables concernées par une contrainte C_c. Elle ne s'applique que si la profondeur restante n'est ni nulle – cas où la variable X_i est la dernière instanciée pour C_c – ni supérieure à la profondeur maximale autorisée (ligne 2). Dans le cas contraire, la fonction renvoie **vrai** sans aucunes vérifications.

Algorithme 2.3 *Établissement de la consistance avant.*

```
1  fonct Consistance_Avant (c, i) ↦ booléen ≡
2    si Prof_Cour (c, i) = 0 ∨ Prof_Cour (c, i) > Prof_Max (c) alors
3      retourner vrai finsi
4    résultat ⟵ vrai
5    pour j chaque élément de var[c] tel que j > i tant que résultat faire
6         résultat ⟵ faux
7         pour v[j] chaque élément de dom[j] faire
8              si ¬Vérifie (c, j) alors
9                dom[j] ⟵ dom[j] \ {v[j]}
10               red[i] ⟵push ⟨j, v[j]⟩
11             sinon résultat ⟵ faux finsi finpour finpour
12   retourner résultat.
```

S'il est nécessaire de vérifier la contrainte C_c, *Consistance_Avant* parcourt toutes les valeurs encore consistantes (ligne 7) des variables X_j non-instanciées (ligne 5). Si la valeur est inconsistante, elle est supprimée du domaine (ligne 9) et l'inconsistance est mémorisée dans les réductions de X_i (ligne 10). La valeur v[i] sera consistante si au moins une valeur support a été trouvée dans chaque variable future (lignes 11–12).

Remarque : Comme elle est présentée dans l'algorithme 2.3, la fonction *Consistance_Avant* ne permet pas de traiter les contraintes unaires puisqu'elles n'ont pas de variables futures au moment de l'instanciation de la variable sur laquelle elles portent. On pourra remplacer le ligne 2 de l'algorithme par :

si (*Prof_Cour* $(c, i) = 0 \wedge |\text{var}[c]| \neq 1) \vee$ *Prof_Cour* $(c, i) >$ *Prof_Max* (c) **alors**

pour tenir compte des contraintes unaires. Nous supposons dans notre implémentation, que ces contraintes ont déjà été filtrées avant la résolution. ◇

Dans le cas où une instanciation doit être annulée, il est fait appel à la fonction *Annuler_Réductions* (algorithme 2.4). Celle-ci parcourt toutes les réductions futures rencontrées sur la variable X_i (ligne 2) afin de rajouter les valeurs futures supprimées dans leur domaine d'origine (ligne 4).

Algorithme 2.4 *Annulation des réductions sur une variable.*

```
1 proc Annuler_Réductions (i) ≡
2   tant que red[i] ≠ ∅ faire
3             ⟨j, v[j]⟩ ⟵pop red[i]
4             dom[j] ⟵ dom[j] ∪ {v[j]} fintant que.
```

Si la fonction *Descente_bFC* ne parvient pas à trouver une valeur consistante pour la variable X_i, la procédure globale de recherche branche l'exécution sur la fonction *Remonté_bFC* – présentée dans l'algorithme 2.5 – , qui se charge d'annuler l'instanciation de la variable X_{i-1} qui est alors inconsistante (ligne 2).

Algorithme 2.5 *Annulation de l'instanciation d'une variable.*

```
1 fonct Remontée_bFC (i, cons) ↦ ⟨entier, booléen⟩ ≡
2   Annuler_Réductions (i − 1)
3   dom[i − 1] ⟵ dom[i − 1] \ {v[i − 1]}
4   retourner ⟨(i − 1), dom[i − 1] ≠ ∅⟩.
```

S'il reste une valeur non explorée dans le domaine de X_{i-1}, la fonction *Remonté_bFC* met la consistance globale à **vrai**, sinon elle la met à **faux** (ligne 4) pour enclencher une nouvelle remontée.

4 Spécialisations de *bFC*

Nous discutons dans ce paragraphe, des spécialisations et adaptations à appliquer aux différentes techniques de recherche pour tenir compte des spécificités de l'algorithme *bFC*. Nous introduisons tout d'abord les techniques de retardement dans *bFC* au paragraphe 4.1, et nous présentons ensuite la mise en œuvre de l'hybridation de *bFC* avec une technique de retour-arrière non chronologique (paragraphe 4.2).

4.1 Retardement dans *bFC*

La technique du retardement a pour objectif de minimiser le temps de résolution en suspendant une partie du filtrage effectué lors d'une instanciation [DM 94b]. Pour ce faire, la consistance établie par *FC* est décomposée en deux étapes distinctes :

1. *Consistance en avant*: l'algorithme vérifie que la valeur en cours d'instanciation est compatible avec au moins une valeur dans chaque domaine futur.

2. *Filtrage en avant*: l'algorithme supprime des domaines futurs, toutes les valeurs qui ne sont pas compatibles avec la valeur en cours d'instanciation.

L'étape (1) est indispensable car elle détermine la validité de la valeur en cours d'instanciation. La seconde étape peut être différée, pour n'être réalisée qu'au moment où la valeur concernée devrait être instanciée.

Dans *bFC*, les contraintes peuvent être vérifiées avec un niveau d'anticipation différent de 1. Dans le cas où la profondeur est de 1, l'implémentation de *MbFC* (pour *minimal b-forward-checking*) est très similaire à *MFC* (*cf.* chap. 1, paragraphe 6.5). Pour une profondeur de vérification strictement supérieure à 1, on peut tirer parti de la remarque suivante :

Remarque : Supposons que C_c – la contrainte à vérifier – a une partie de ses variables instanciées, et que X_j et X_k ne le sont pas. Si *bFC* trouve une valeur consistante avec les instanciations déjà effectuées selon C_c pour X_j cela signifie qu'il existe aussi une valeur compatible avec C_c pour X_k. ◇

Ainsi, il n'est pas nécessaire de rechercher un support dans chaque domaine non instancié, mais un support dans un seul des domaines futurs.

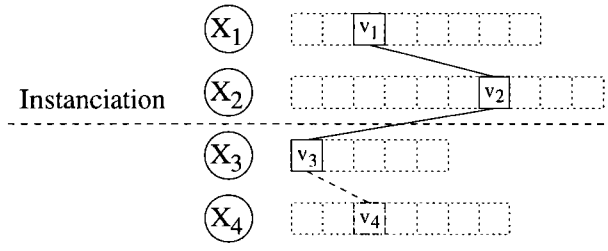


Figure 2.7 : Filtrage minimal sur une contrainte 4-aire.

Le marquage négatif (*cf.* chap. 1, paragraphe 6.4) ne se fera que pour les valeurs du premier domaine, alors que le marquage positif s'étend facilement à tous les domaines futurs. La figure 2.7 présente un exemple de filtrage en avant pour une contrainte 4-aire : au moment de l'instanciation de la valeur v_2 dans D_2, on recherche dans D_3 un élément qui valide v_2. Puisque v_3 est trouvé dans D_3, cela signifie qu'il existe une valeur v_4 dans D_4 qui fait partie du 4-uplet validant v_2. L'algorithme *MbFC* peut donc marquer positivement v_4.

4.2 Retour-arrière non chronologique

Dans les algorithmes de recherche implémentant un retour-arrière non chronologique, la difficulté induite par les contraintes n-aires porte sur la détermination des points de retour.

Dans le cas du *graph-based backjumping* (*cf.* chap. 1, paragraphe 6.2), le calcul des points de retour ne dépend que de la structure du réseau de contraintes. Si un blocage apparaît sur une variable X_i, le retour se fera sur la variable X_j la plus profonde dans l'arbre de recherche qui partage une contrainte avec X_i. Pour une même contrainte, les variables instanciées avant X_j ne doivent pas être reconsidérées sans avoir testé toutes les combinaisons avec les valeurs de X_j.

Pour les algorithmes implémentant une forme de *backjumping*, les mêmes considérations peuvent être faites quant au point de retour. Toutefois, les conflits sont mémorisés

à la descente en respectant le schéma de calcul précédent.

5 Heuristiques d'ordre n-aires sur les variables

Nous discutons dans cette partie du choix statique ou dynamique de l'ordre d'instanciation des variables. Il apparaît en effet que l'algorithme *bFC* nécessite de nouvelles heuristiques d'ordre.

On sait qu'une des heuristiques d'ordre sur les variables les plus efficaces est celle dite « *minimum remaining value* (*MRV*) » qui consiste à choisir la prochaine variable à instancier, comme celle qui possède le moins de valeurs à tester (*cf.* chap. 1, paragraphe 6.6). Bacchus et van Run ont montré que le *forward-checking* était particulièrement efficace s'il était combiné avec *MRV* [BvR 95].

Toutefois, cette heuristique n'est efficace que si les contraintes sont vérifiées dès que possible, c'est-à-dire dès qu'une de leurs variables a été instanciée. Or comme nous l'avons vu au paragraphe 2, la vérification totale des contraintes sur leurs variables coûte très cher pour un rendement plutôt faible, si bien que dans la plupart des cas, on limite la profondeur de filtrage sur les contraintes de plus grande arité.

Dans le cas de *bFC*, il devient possible d'instancier plusieurs variables avant de pouvoir vérifier une contrainte. On a donc une génération combinatoire de tous les n-uplets possibles sur ces variables lors de la recherche de solutions. La figure 2.8 présente ce type d'inconvénient : si on considère que la profondeur de vérification $b = 1$, les variables sont instanciées dans l'ordre du schéma et la recherche est totalement combinatoire sur les quatre premières variables.

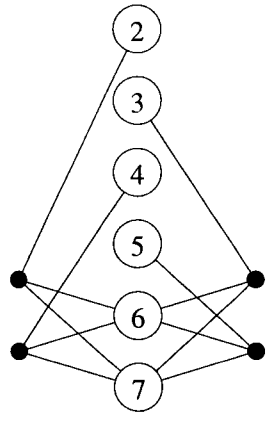


Figure 2.8 : *Cas où MRV est très inefficace* – les variables sont représentées par des cercles ; les nombres à l'intérieur correspondent au nombre de valeurs. Les contraintes sont symbolisées par des points noirs avec des traits vers les variables qu'elles lient.

Pour remédier à ce problème, il est nécessaire de tenir compte de la structure du graphe afin de vérifier les contraintes au plus tôt. Dans le cas de la figure 2.8, il serait préférable d'instancier la cinquième puis la sixième variable afin que toutes les contraintes puissent s'appliquer.

Quelle que soit l'heuristique d'ordre à choisir, son objectif principal sera de réduire au plus tôt les branches de l'arbre de recherche. Dans le cas de *FC* binaire, l'heuristique *MRV*

est très efficace puisqu'elle réduit les combinaisons entre les valeurs. Son application est aisée puisque les contraintes ont toutes la même largeur (2) et s'appliquent au plus tôt.

Dans le cas de *MRV* - sur un CSP binaire – on tient compte de la satisfiabilité des contraintes par effet de bord : une fois que l'une des variables de la contrainte la moins satisfiable a été instanciée, la contrainte est appliquée et cela réduit de façon significative le domaine de la seconde variable. Il apparaît comme nécessaire de tenir compte de la satisfiabilité des contraintes dans une heuristique d'ordre efficace sur les variables pour *bFC*.

La structure du graphe, et notamment le recouvrement entre les contraintes est aussi un paramètre important puisque, plus une variable est liée par des contraintes, plus elle est susceptible de réduire le domaine des variables futures lors du filtrage de ces contraintes. De plus, il est utile de privilégier l'application des contraintes avec une profondeur minimale car leur « travail » au sens du filtrage est plus significatif.

Le compromis entre toutes ces idées afin d'obtenir une heuristique d'ordre efficace sur les variables n'est pas du tout trivial. Il faut naturellement appliquer en priorité les critères les plus significatifs, mais leur calcul peut être perturbé au cours de la recherche :

- Le nombre de valeurs restantes n'est plus pertinent dans le cas de *MFC* qui ne filtre pas tout le domaine, mais se contente de chercher la première valeur satisfiable. Si la valeur trouvée était la première testée et que toutes les autres sont inconsistantes, la méthode ne considérera pas la variable comme prioritaire alors qu'elle aurait dû l'être. La notion de *variable prioritaire* se définie par rapport à un ordre d'instanciation optimal en terme de temps de calcul (qui n'est pas connu à priori). Dans le cas présent, le problème vient d'une trop grande différence entre l'ordre défini par l'heuristique, et l'ordre optimal.

- Une fois qu'une variable a été instanciée, il serait préférable de tenir compte de la satisfiabilité de la sous-contrainte restante qui peut différer de la satisfiabilité totale de la contrainte, mais n'est pas toujours facile à calculer en temps raisonnable.

Nous proposons ici une méthode simple pour calculer un ordre d'instanciation des variables. Il ne sera pas optimal dans tous les cas de figure, mais est susceptible d'exhiber des performances satisfaisantes en privilégiant l'échec au plus tôt car ce type général d'heuristique a déjà montré tout son potentiel.

La méthode consiste à attribuer à chaque variable, une note en fonction de différents paramètres et à choisir à chaque fois la variable dont la note est la plus faible. La fonction que nous proposons pondère le nombre de valeurs restantes dans chaque domaine par le produit des satisfiabilités des contraintes qui portent sur la variable :

$$f(i) = d_i \times \prod_{k}^{C_k \in \bar{C}_i} \mathcal{S}_{C_k} \tag{2.1}$$

Dans cette formule, $\bar{C}_i$ indique l'ensemble des contraintes qui portent sur la variable X_i, et $\mathcal{S}_{C_k}$, la satisfiabilité de la contrainte C_k. On ne tient pas compte ici de la profondeur de vérification car on suppose que la répartition des n-uplets autorisés dans chaque contrainte est homogène, *i.e.* l'instanciation d'une variable ne change pas la satisfiabilité de la sous-contrainte restante.

Sur le plan opérationnel, la fonction 2.1 se calcule en deux étapes: le produit des satisfiabilités se calcule à l'initialisation du réseau de contraintes par accumulation des produits. Il aura donc un coût négligeable – en O (1) – et ne changera pas au cours de la recherche. L'intégration de d_i dans la formule se recalcule quand une ou plusieurs valeurs sont supprimées dans le domaine et est aussi effectuée en O (1).

Il n'est pas possible de juger la pertinence de cette heuristique dans l'absolu car les CSP à résoudre sont très diversifiés. On pourra toutefois confirmer (ou éventuellement infirmer) son intérêt par une expérimentation statistique comparative de plusieurs heuristiques d'ordonnancement.

Dans la suite de ce document, et notamment dans le chapitre 3 qui établira la comparaison avec *MRV*, nous appellerons cette heuristique: *maximal constrained variable* (*MCV*).

Conclusion

Nous nous sommes intéressés dans ce chapitre, aux problèmes de satisfaction de contraintes incluant des contraintes n-aires, et en particulier aux méthodes de recherche par consistance en avant. En effet, les algorithmes complets de résolution des CSP les plus performants mettent en œuvre des mécanismes de vérification de consistance en avant. De plus, de nombreux problèmes d'Intelligence Artificielle se modélisent aisément à l'aide de contraintes n-aires. Or les algorithmes de recherche par vérification de consistance en avant sont basés sur des notions de filtrage binaire qui ne sont pas adaptées aux contraintes n-aires.

Partant d'une étude exhaustive des techniques de consistance n-aires, nous avons mis en évidence la dualité entre l'hyper-k-consistance et la relationnelle-k-consistance, et souligné les apports algorithmiques de ces deux méthodes. La première est en fait adaptée à une recherche par instanciation des n-uplets des contraintes – aussi appelée *recherche duale* – puisqu'elle conduit au filtrage des relations associées aux contraintes. Pour sa part, la seconde notion de consistance conduit à une réduction des domaines des variables et semble donc mieux adaptée comme pré-traitement à une recherche par instanciation des variables. Toutefois, elle induit aussi une génération de nouvelles contraintes n-aires qui rendrait son utilisation difficile dans un algorithme de recherche par consistance en avant.

La propriété de relationnelle-k-consistance étant la plus apte à dériver un algorithme de recherche en avant sur des contraintes n-aires, nous l'avons étendue par la définition de la *relationnelle-(i,j)-consistance* qui met en avant la notion de *profondeur de vérification* (j). Nous passons en fait par la définition d'une notion de consistance « dégénérée »: la b-consistance qui constitue une extension aux contraintes n-aires de l'arc-consistance. Sur le plan théorique, nous avons montré que la complexité de l'établissement de la b-consistance est exponentielle en la profondeur de vérification (b), pour une espérance d'efficacité inversement proportionnelle à b.

Nous avons de plus défini dans ce chapitre, un nouvel algorithme de recherche par consistance en avant dans les CSP n-aires (*bFC*), qui reprend les concepts généraux du *forward-checking*, en établissant la *b-consistance* à chaque instanciation. Les implications

de la consistance n-aire sur d'autres aspects de l'algorithme de recherche sont aussi discutées, notamment le retardement de la consistance, et les heuristiques d'ordre sur les variables efficaces pour l'algorithme *bFC*. Nous définissons ainsi l'heuristique *MCV* qui tente de minimiser le coût de recherche de l'algorithme *bFC*.

L'algorithme *bFC* et l'heuristique d'ordre sur les variables *MCV* que nous avons définis dans ce chapitre seront évalués expérimentalement dans le chapitre 3.

Chapitre 3

Expérimentations binaires et n-aires

LES ALGORITHMES DE RÉSOLUTION des problèmes de satisfaction de contraintes par recherche complète ont tous la même complexité théorique exponentielle mais ils n'ont pas tous les mêmes performances pratiques. Ainsi, seule une expérimentation statistique permet de déterminer lesquels sont plus performants.

L'objectif de ce chapitre est de comparer les nouveaux algorithmes et méthodes définis précédemment, avec les techniques de résolution classiques. Nous tenterons notamment de mettre en évidence l'intérêt que présente la recherche n-aire directe par rapport aux méthodes classiques de travail sur des graphes binaires équivalents obtenus par transformation duale.

Le paragraphe 1 présente l'environnement algorithmique des expérimentions. Elle développe en particulier un modèle de génération aléatoire de CSP n-aire qui constitue une extension du modèle de Prosser [Pro 94]. Les résultats sont ensuite exposés et analysés dans le paragraphe 2 : nous comparons tout d'abord l'heuristique d'ordonnancement des variables *MCV* (développée pour la recherche en avant n-aire) avec l'heuristique *MRV* (*cf.* chap. 1, paragraphe 6.6), conçue à l'origine pour l'algorithme *FC* binaire. Nous étudions ensuite les effets de la variation de la densité de contraintes dans les CSP, la variation de la taille des domaines, puis celle du nombre de variables. Dans ces derniers cas, la comparaison est établie avec la recherche sur le graphe binaire dual.

1 Environnement expérimental

Nous présentons l'environnement algorithmique des expérimentations en trois parties : nous étendons tout d'abord le modèle de génération binaire de Prosser pour pouvoir travailler sur des CSP n-aires. Puis nous discutons rapidement de l'implémentation des algorithmes, avant de présenter les séries d'expériences réalisées.

1.1 Modèle de génération aléatoire n-aire

Les algorithmes sont expérimentés sur des problèmes générés aléatoirement. Dans le cadre binaire, le modèle de génération de Prosser semble être le plus utilisé (*cf.* chap. 1, paragraphe 7.2), ce qui nous amène à l'étendre pour prendre en compte la notion de CSP n-aires. Un ensemble de CSP n-aires générés aléatoirement dans notre modèle, sera défini par un 5-uplet $\langle a, n, m, p_1, p_2 \rangle$, où :

- a est l'arité uniforme de chaque contrainte ;
- n est le nombre de variables dans un CSP généré ;
- m est la taille uniforme des domaines des variables ;
- p_1 est la probabilité pour une possible contrainte a-aire d'apparaître dans l'hypergraphe du CSP ;
- p_2 est la probabilité pour un a-uplet possible d'une relation d'être inconsistant, *i.e.* que le bit correspondant dans la matrice de la relation soit mis à **faux**.

Le paramètre a est spécifique à la génération des CSP n-aires, alors que les quatre autres sont communs avec le schéma général de Prosser. On notera que ce modèle génère des CSP homogènes puisque les variables ont toutes le même domaine, et que les contraintes ont des arités et satisfiabilités identiques. Il ne peut donc pas être considéré comme une modélisation de CSP « réels », mais permet d'analyser le comportement d'algorithmes donnés relativement à une caractérisation précise des problèmes.

Le choix des valeurs aléatoires est réalisé par un générateur de nombres dont la distribution est uniforme. De plus, les CSP dont l'hypergraphe n'est pas connexe sont rejetés. Il importe donc de choisir une densité de contraintes (p_1) suffisante pour garantir la possibilité de génération de CSP connexes.

1.2 Implémentation des algorithmes

Tous les algorithmes que nous avons testés sont implémentés en langage C. Les CSP sont stockés par des listes (liste des variables et liste des contraintes). Chaque relation est stockée par une matrice booléenne afin de garantir un temps de vérification de consistance équivalent pour chaque contrainte. L'arité des contraintes est bornée dans le programme par une valeur arbitraire, pour préserver l'espace mémoire. Ainsi, les CSP pourront avoir des contraintes 4-aires au plus. Les expérimentations ont été exécutées sur des Sparc 10 et Sparc Ultra 1.

L'algorithme que nous avons choisi pour représenter les méthodes de recherche par consistance en avant n-aire est *MbFC-CBJ*[1]. Ce choix représente une méthode déjà éprouvée quant au filtrage avant (pour les CSP binaires), le retardement n'étant ajouté que pour limiter le coût de recherche sur les CSP satisfiables.

La transformation des CSP n-aires en graphes de contraintes duaux n'est pas explicitement implémentée : l'algorithme *MFC-CBJ*[2] instancie directement les a-uplets des relations. Les vérifications de consistance sont calculées entre les relations, mais les méta-contraintes et méta-relations ne sont pas construites explicitement. Ainsi, la phase de transformation n'est pas comptée dans les résultats expérimentaux.

La principale unité de mesure des performances que nous utilisons est le nombre de vérifications de bit dans une matrice de relation. Ceci conduit naturellement à négliger certains traitements auxiliaires lors de la résolution – principalement la mise à jour des listes – mais donne une approximation fiable du coût de résolution, sans tenir compte des particularités d'implémentation. Nous donnons toutefois des statistiques sur les temps de résolution (en nombre de secondes par problèmes) pour les différents algorithmes dans le début du paragraphe 2.

1.3 Expériences réalisées

Les expérimentations se sont déroulées en trois étapes :

1. Des expériences préliminaires ont été réalisées sur des CSP binaires pour vérifier l'équivalence de la recherche de type *MbFC-CBJ* et de *MFC-CBJ* sur le graphe dual. Différents paramètres ont été testés, principalement le nombre de variables (n) et la taille des domaines (m). De plus, les CSP étaient résolus avec différentes profondeurs de vérification.

   Le principal intérêt de cette série d'expériences est de vérifier les algorithmes – ainsi que la proposition 2.16 – si bien que les résultats ne seront pas présentés dans ce chapitre.

2. La comparaison de l'heuristique d'ordonnancement des variables *MCV* (*cf.* chap. 2, paragraphe 5) a été comparée avec l'heuristique *MRV* (chap. 1, paragraphe 6.6) pour le même algorithme de recherche (*MbFC-CBJ*).

3. L'évaluation complète de *MbFC-CBJ* en fonction de différentes profondeurs de vérification, et par rapport à *MFC-CBJ* sur le graphe dual correspondant. Pour ce groupe d'expérimentations, les variables étaient ordonnées par *MCV*.

Toutes les contraintes dans ces expérimentations (à l'exception de la première phase) sont d'arité égale à 4 – maximal possible dans les implémentations. Nous ne testons pas la résolution sur des contraintes d'arité inférieure pour avoir une variation intéressante de la profondeur de vérification.

[1] descente dans l'arbre de recherche de type *forward-checking* borné, avec retardement ; remonté par retour-arrière orienté par les conflits.

[2] nous avons choisit *MFC-CBJ* pour la recherche duale car il est équivalent en terme de filtrage et de retour-arrière à l'algorithme utilisé pour la résolution n-aire.

1.3.1 Évaluation de *MCV*

Le groupe d'évaluation des heuristiques est caractérisé par le 5-uplet $\langle 4, 10, 5, 0.1, p_2 \rangle$. Le paramètre p_2 varie de 0 à 1 avec 58 valeurs différentes pour obtenir des courbes précises. À chaque valeur de p_2, correspond une moyenne de résolution sur 100 problèmes. La série a été calculée en 2 heures CPU.

Les problèmes sont successivement résolus avec trois heuristiques d'ordre sur les variables : 1) un ordre quelconque (*DEF*) comme référence ; 2) l'ordre *MCV* et 3) l'heuristique *MRV*.

1.3.2 Évaluation de *MbFC-CBJ*

L'algorithme *MbFC-CBJ* est évalué et comparé avec *MFC-CBJ* sur le graphe dual par quatre séries de tests qui permettent d'observer l'influence de la variation de différents paramètres de génération sur la résolution. Les CSP sont traités successivement par quatre méthodes : *MbFC-CBJ* avec des profondeurs de vérification variant de 1 à 3, et *MFC-CBJ* sur le graphe dual. Les séries des tests sont :

- La première série est caractérisée par $\langle 4, 10, 5, 0.1, p_2 \rangle$, c.-à-d. le même schéma que pour l'expérimentation des heuristiques d'ordre. Le paramètre p_2 varie d'ailleurs ici avec le même comportement (58 points de 0 à 1, et 100 problèmes par point)[3]. Le temps de calcul a été de 3 heures CPU. Cette série est utilisée comme référence pour les variations des séries suivantes.

- La deuxième série est paramétrée par $\langle 4, 10, 5, 0.5, p_2 \rangle$. Nous étudions ici l'influence de la variation de la densité de contrainte (p_1) sur les différents algorithmes de recherche. On notera que dans la série précédente, les CSP comportaient 21 contraintes, alors qu'ils en comportent ici 105. 33 valeurs pour p_2 ont été testées, avec 100 problèmes pour chaque points. Cette série a été calculée en 30 heures de temps machine environ.

- La troisième série est $\langle 4, 10, 10, 0.1, p_2 \rangle$. Elle étudie l'influence de la variation de la taille des domaines sur les algorithmes de résolution. Nous avons calculé 48 points pour chaque courbe, avec des statistiques sur 100 problèmes à chaque fois. Cette série a nécessité environ 34 heures de calcul.

- La dernière série évalue le comportement des algorithmes de résolution par rapport à la variation du nombre de variables. Elle est caractérisée par $\langle 4, 15, 5, 0.1, p_2 \rangle$, où p_2 prend 38 valeurs successives, avec 50 problèmes par point. L'exécution de cette série à mobilisé 123 heures de calcul.

Toutes les expérimentations n-aires ont été réalisées avec l'heuristique d'ordre *MCV* qui est apparue comme plus efficace sur ces problèmes. Les expériences relatives à *MFC-CBJ* sur le graphe dual ont utilisé l'heuristique *MRV*.

[3] pour des raisons pratiques, les tests de la série sur les heuristiques et celle-ci n'ont pas été exécutés sur les mêmes problèmes.

2 Analyse des résultats

Les résultats des expérimentations sont présentés dans les figures 3.1–3.5. Une courbe correspond au nombre moyen de vérifications de bit requis pour la résolution d'un problème par un algorithme donné (axe y). La densité de conflits p_2 varie le long de l'axe des x dans les courbes. On notera que le nombre de vérifications de bit est exprimé selon une échelle logarithmique.

la première observation importante est que le phénomène de *transition de phase* apparaît relativement à la variation de p_2 pour les CSP n-aires (voir chap. 1, paragraphe 7.2). Dans les courbes qui suivent, le seuil de transition $\widehat{p_2}$ est matérialisé par une ligne verticale en trait plein, alors que la zone de transition – *i.e.* où apparaissent à la fois des problèmes satisfiables et non satisfiables – est délimitée par des lignes verticales en tirets.

Avant de détailler les résultats selon le nombre de vérifications de bit, nous résumons les temps moyens de résolution pour les problèmes difficiles – dans la zone de transition – pour les quatre dernières séries de tests. La table 3.1 présente le coût moyen de résolution d'un problème, en secondes. Il apparaît que *MbFC-CBJ* avec une profondeur de vérification de 1 est moins coûteux qu'avec une profondeur plus importante. Cette différence s'explique par le fait qu'une profondeur élevée accroît notablement le coût de vérification d'un nœud, sans réduire significativement l'arbre de recherche.

Table 3.1 : Coût moyen de résolution d'un problème, en secondes. Seuls les problèmes dans la zone de transition sont comptés dans ce tableau.

Algorithme	$\langle 4, 10, 5, 0.1, p_2 \rangle$	$\langle 4, 10, 5, 0.5, p_2 \rangle$	$\langle 4, 10, 10, 0.1, p_2 \rangle$	$\langle 4, 15, 5, 0.1, p_2 \rangle$
MbFC-CBJ, $b = 1$	0.33	1.73	1.22	11.74
MbFC-CBJ, $b = 2$	0.56	4.62	1.65	46.27
MbFC-CBJ, $b = 3$	0.81	13.81	3.10	115.33
MFC-CBJ dual	0.29	12.07	20.28	60.42

La comparaison entre les recherches n-aires et duales est un peu plus délicate à la vue de ce tableau : *MFC-CBJ* dual est plus rapide pour la série $\langle 4, 10, 5, 0.1, p_2 \rangle$, mais plus lent pour les autres. La différence sera analysée en terme de vérifications de bit dans les paragraphes 2.2 à 2.5

2.1 Comparaisons des heuristiques d'ordre

Les résultats de la série $\langle 4, 10, 5, 0.1, p_2 \rangle$ sont donnés dans la figure 3.1. La zone de transition correspond à une valeur de p_2 comprise entre 0.48 et 0.62. Le seuil de transition se situe à $\widehat{p_2} \simeq 0.54$.

Les trois zones caractéristiques – facile-satisfiable, difficile et facile-insatisfiable – apparaissent assez nettement sur le graphique (rappelons que le coût de résolution est reporté sur une échelle logarithmique). Dans la zone facile-satisfiable, les heuristiques *MCV* et *MRV* restent comparables, et n'apportent d'ailleurs pas de nettes améliorations par rapport à l'ordre quelconque qui sert de référence. Dans les deux autres zones, l'heuristique *MCV* devient la plus efficace.

L'observation du comportement relatif des trois courbes, en dehors du phénomène de transition de phase montre que l'écart entre les heuristiques varie proportionnellement avec p_2 : pour les valeurs faibles de p_2, les courbes sont presque confondues, et la courbe *DEF* s'éloigne de plus en plus de *MCV* avec l'accroissement de p_2. La courbe de l'heuristique *MRV* varie entre ces deux bornes.

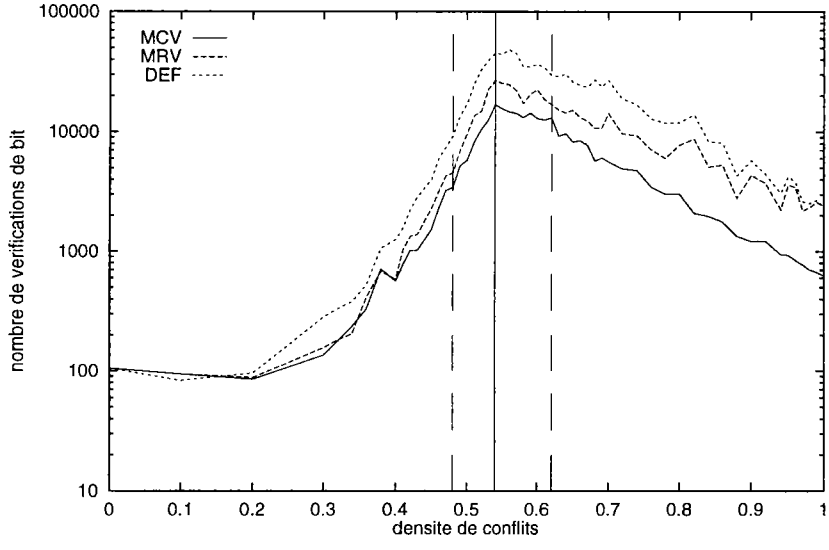


Figure 3.1 : Comparaison des heuristiques d'ordonnancement des variables. Les graphes ont 10 variables, 5 valeurs par variable et 21 contraintes 4-aires ($p_1 = 0.1$).

En fait, *MCV* prend l'avantage sur *MRV* lorsque la satisfiabilité des contraintes devient plus faible : cette dernière est orientée par l'effet du filtrage sur les domaines, qui constitue un effet de bord par rapport à la satisfiabilité des contraintes. Dans le cas de contraintes binaires, les deux heuristiques se comportent de la même manière, alors que pour les contraintes n-aires, l'effet du filtrage est réduit par le retard entre l'instanciation des variables, et l'application des contraintes.

2.2 Analyse de la série $\langle 4, 10, 5, 0.1, p_2 \rangle$

La figure 3.2 présente le résultat des résolutions de la série $\langle 4, 10, 5, 0.1, p_2 \rangle$. Les bornes des zones de transition sont identiques à celles de la figure 3.1 puisque les paramètres de génération sont les mêmes.

Globalement, les quatre courbes ont un comportement similaire : le coût de résolution augmente pour p_2 variant de 0 à $\widehat{p}_2$ et diminue pour p_2 variant de $\widehat{p}_2$ à 1. L'algorithme *MbFC-CBJ* avec une profondeur de vérification de 1 est le plus rapide sur presque tout le spectre de génération, et approximativement égal à la recherche sur le graphe dual quand $b = 2$.

Toutefois, une quatrième zone peut être observée quand $p_2 > 0.9$. Là, la recherche sur le graphe dual devient nettement plus rapide que la méthode directe n-aire. Ceci est principalement dû au fonctionnement de l'algorithme dual : quand une contrainte est hautement insatisfiable, le domaine de la méta-variable correspondante est très réduit, et l'heuristique *MRV* concentre la recherche sur les points les plus stratégiques du graphe, pour mettre rapidement en évidence une incompatibilité globale entre les valeurs. Dans

cette même zone, l'algorithme *MbFC-CBJ* avec une profondeur de vérification de 2 s'avère équivalent, voire meilleur que pour une profondeur de 1. Ceci s'explique par la forte insatisfiabilité des contraintes qui conduit à un élagage significatif de l'arbre de recherche plus tôt.

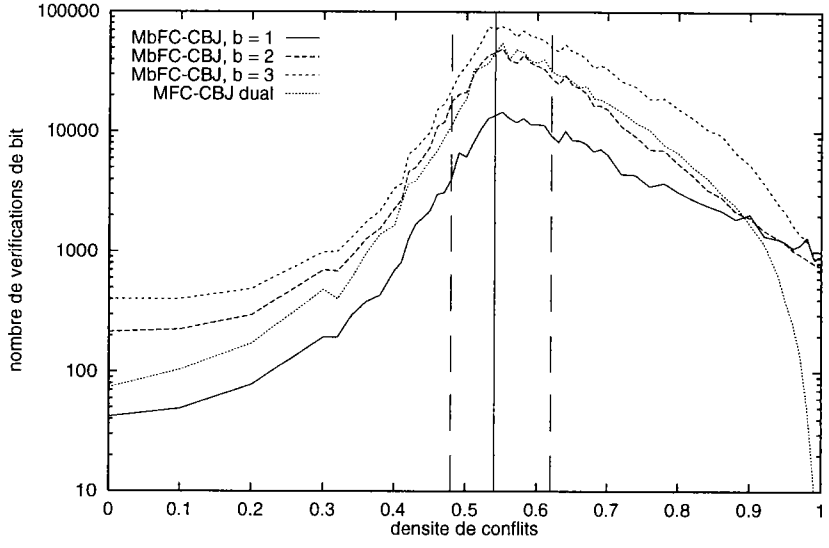


Figure 3.2: Coût de résolution des problèmes comportant 10 variables, 5 valeurs par variables et 21 contraintes 4-aires ($p_1 = 0.1$).

Dans la zone de transition, la recherche *MFC-CBJ* sur le graphe dual est moins coûteuse en terme de temps CPU bien qu'elle nécessite plus de vérifications de bits. En fait, *MbFC-CBJ* réalise sensiblement moins de vérifications par secondes à cause de mises à jour auxiliaires plus importantes dans l'implémentation. Cette particularité, combinée à une différence en nombre de vérifications relativement faible, conduit à cette situation.

2.3 Variation de la densité de contraintes

Les résultats de la série $\langle 4, 10, 5, 0.5, p_2 \rangle$ portant sur la variation de la densité de l'hypergraphe de contraintes sont reportés dans la figure 3.3. La transition de phase correspond à la zone où $0.12 \leq p_2 \leq 0.18$ (lignes verticales en tirets), alors que le seuil de transition se trouve à la valeur de $\widehat{p_2} \simeq 0.145$.

Le même phénomène de transition de phase apparaît ici, avec quatre zones distinctes. Pour les valeurs élevées de p_2, la recherche sur le graphe dual devient plus rapide, et la recherche n-aire à profondeur 2 est comparable avec celle à profondeur 1. On pourra toutefois signaler que le seuil de transition s'est nettement déplacé vers les valeurs faibles de p_2, *i.e.* les problèmes deviennent plus rapidement insatisfiables lorsque p_1 augmente.

La comparaison entre la série $\langle 4, 10, 5, 0.1, p_2 \rangle$ et la série $\langle 4, 10, 5, 0.5, p_2 \rangle$ montre que la variation de p_1 ne modifie pas le comportement de *MbFC-CBJ* par rapport à *MFC-CBJ* sur le graphe dual : le premier est meilleur pour presque toutes les valeurs de p_2, et n'est surclassé que pour les CSP hautement insatisfiables.

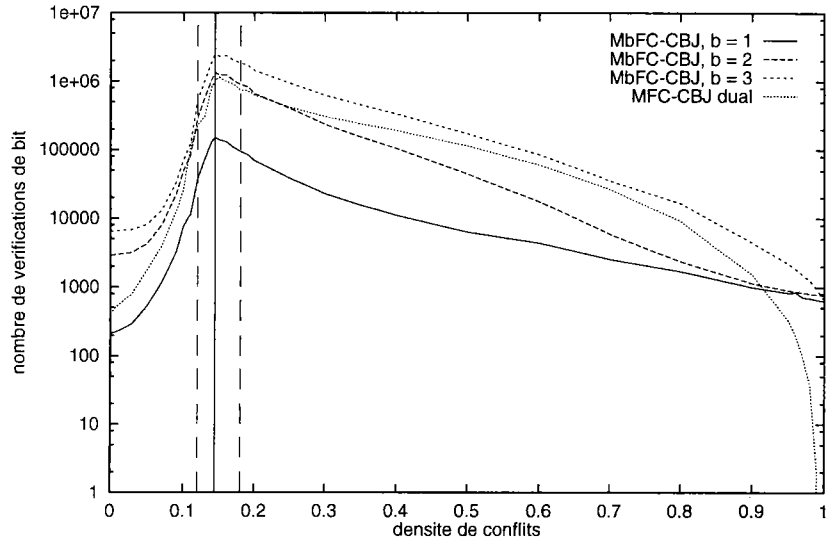


Figure 3.3 : Coût de résolution des problèmes comportant 10 variables, 5 valeurs par variables et 105 contraintes 4-aires ($p_1 = 0.5$).

2.4 Variation de la taille des domaines

La série $\langle 4, 10, 10, 0.1, p_2 \rangle$ conduit aux résultats qui sont présentés dans la figure 3.4. La zone de transition correspond aux valeurs de p_2 comprises entre 0.62 et 0.73. Le seuil de transition apparaît à $\widehat{p_2} \simeq 0.665$.

Le quatre zones se retrouvent aussi sur cette courbe, et la « hiérarchie » établie sur la série de référence est respectée dans chacune d'elles. Les différences sont en effet minimes par rapport à la figure 3.2: le seuil de transition est décalé vers les valeurs plus élevées de p_2, et les problèmes sont beaucoup plus difficiles à résoudre au seuil car l'arbre de recherche est plus large.

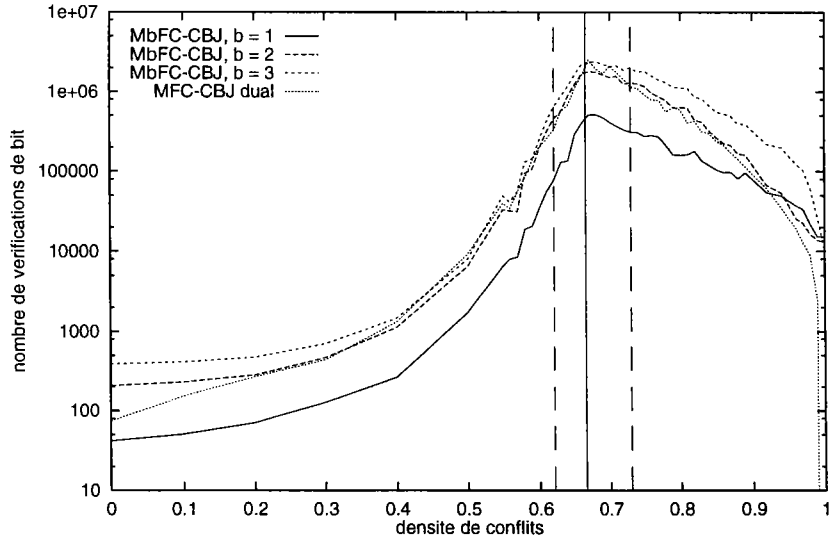


Figure 3.4 : Coût de résolution des problèmes comportant 10 variables, 10 valeurs par variables et 21 contraintes 4-aires ($p_1 = 0.1$).

Les fortes similitudes entre la série de référence et celle-ci montre que la variation de la taille des domaines ne bouleverse pas les performances relatives des algorithmes testés, et que *MbFC-CBJ* à profondeur 1 reste globalement le meilleur (sauf pour les fortes densités de conflits).

2.5 Variation du nombre de variables

La figure 3.5 rapporte les résultats des expérimentations de la série $\langle 4, 15, 5, 0.1, p_2 \rangle$. La zone de transition est étroite puisqu'elle correspond à des valeurs de p_2 comprises entre 0.14 et 0.19, alors que $\widehat{p_2} \simeq 0.165$.

Le phénomène classique de pic du coût de résolution au seuil de transition peut là encore être observé. Mais dans cette série, la plupart des problèmes sont insatisfiables car ils sont constitués de plus de contraintes : la série de référence comportait deux fois plus de contraintes que de variables, alors que celle-ci en comporte neuf fois plus.

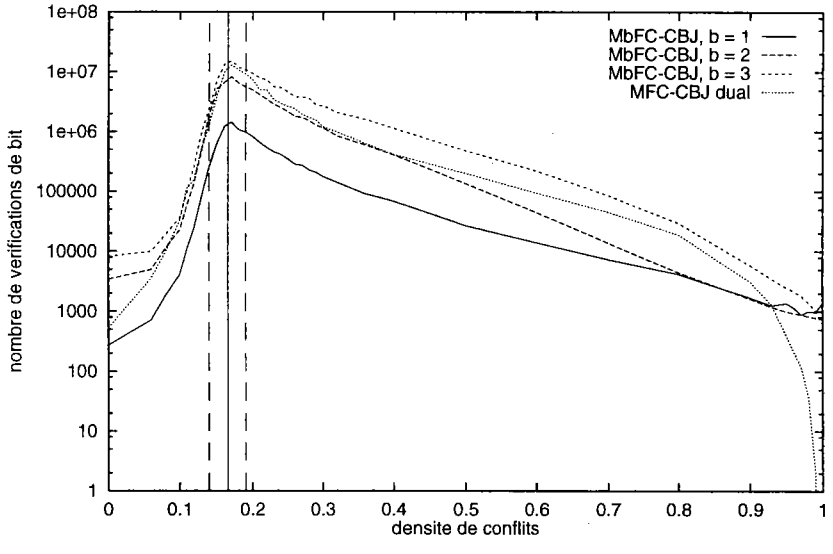


Figure 3.5 : Coût de résolution des problèmes comportant 15 variables, 5 valeurs par variables et 136 contraintes 4-aires ($p_1 = 0.1$).

Bien que la résolution moyenne d'un problème devienne nettement plus coûteuse lorsque le nombre de variables augmente, la recherche *MbFC-CBJ* reste plus rapide que *MFC-CBJ* sur le graphe dual pour les problèmes difficiles.

Conclusion

L'objectif de ce chapitre était de valider expérimentalement les méthodes et algorithmes définis au chapitre 2. On pourra donc tirer des conclusions sur différents plans.

Pour ce qui concerne l'heuristique d'ordonnancement des variables, il est apparu que l'heuristique *MCV* que nous avions définie, surclasse la traditionnelle *MRV* pour la plupart des problèmes. Elle n'est devancée que pour les problèmes très satisfiables, qui ne sont habituellement pas considérés comme prépondérants dans le cadre des résolutions

complètes. De plus, le comportement des deux heuristiques vis-à-vis du phénomène de transition de phase a montré que *MCV* devient proportionnellement plus performante que *MRV* lorsque la densité des conflits dans les contraintes augmente. On pourra donc conclure que *MCV* est une bonne heuristique d'ordonnancement des variables dans un algorithme de recherche par vérification de consistance en avant sur des contraintes n-aires.

Dans la comparaison entre les techniques de résolution directe des CSP n-aires par une méthode de recherche par vérification de consistance en avant, et la méthode classique de résolution du graphe dual, il est apparu que l'algorithme *bFC* que nous avions proposé dans le chapitre 2 est globalement plus rapide. Il est toutefois surclassé par la recherche duale lorsque les contraintes sont très fortes (densité de conflits élevée). Les variations des paramètres de génération, tant sur la densité de l'hypergraphe que sur le nombre de valeurs ou de variables ont de plus montré que cette différence était bien un phénomène général, et non une observation sur un cas favorable.

Le choix de la recherche sur le graphe dual comme représentante des recherches sur un graphe binaire par transformation du CSP n-aire pourrait être critiqué du fait qu'elle conduit à la création de variables définies sur des domaines très grands. En fait, nous avons éliminé la transformation en graphe primal (*cf.* chapitre 1) car celle-ci ne garantit pas l'équivalence des CSP, et conduirait donc à autoriser des solutions qui auraient dû être interdites. La transformation en graphe biparti garantit l'équivalence des CSP, mais conduit aussi à la construction de très grands domaines. De plus, elle serait plus compliquée à mettre en œuvre car elle nécessiterait la construction explicite du graphe binaire.

L'expérimentation de *bFC* avec des profondeurs de vérification différentes, n'a pas montré un grand intérêt pratique pour la prise en compte de profondeurs supérieures à 1. En effet, *bFC* à profondeur 2 ne s'avère meilleur que son équivalent à profondeur 1 que pour les valeurs très élevées de la densité de conflit, c.-à-d. loin du seuil de transition où se focalise généralement les recherches sur les algorithmes de résolution. On pourra toutefois remarquer que dans les problèmes réels construits avec des contraintes de satisfiabilité différente, il peut devenir intéressant de vérifier celles induisant le plus de conflicts avec une profondeur supérieure à 1.

Deuxième partie

Algorithmique parallèle pour les CSP

Chapitre 4

État de l'art : parallélisme et résolution des CSP

LA PARALLÉLISATION des algorithmes de résolution des CSP est peu étudiée dans la littérature. Par contre, des nombreuses recherches ont été menées dans des cadres de résolution proches. En fait, les algorithmes énumératifs que nous étudions pour les CSP constituent des parcours d'arbres de recherche et la parallélisation portera donc sur celle d'un arbre. Dans ce contexte, on sera souvent amené au cours de ce chapitre, à étudier des techniques issues de la communauté *programmation logique* (PROLOG) ou la communauté *recherche opérationnelle* qui utilisent des algorithmes de parcours d'arbre.

Avant de faire le point sur la résolution des CSP en parallèle, nous aborderons quelques notions indispensables lorsque l'on parle de parallélisme (paragraphe 1). Ainsi, nous présenterons les différents modèles de parallélisme utilisés pour définir les algorithmes parallèles, puis quelques notions de complexité spécifiques au parallélisme.

Une fois ces généralités introduites, nous présenterons les résultats récents sur la parallélisation des techniques de filtrage – principalement l'arc-consistance – tant sur les aspects théoriques, que sur les algorithmes SIMD ou MIMD (paragraphe 2). Nous évoquerons ensuite rapidement les algorithmes parallèles de synthèse de contraintes dans le paragraphe 3, avant de faire une revue de détail des algorithmes parallèles de recherche énumératifs (paragraphe 4).

1 Introduction au parallélisme

Avant de présenter les techniques et algorithmes parallèles conçus pour la résolution de problèmes de satisfaction de contraintes, nous introduisons quelques notions de base sur le parallélisme. Nous évoquons en particulier les modèles de programmation parallèle dans un premier paragraphe, puis nous abordons les notions de complexité dans le cadre des algorithmes parallèles (paragraphe 1.2).

Les concepts plus concrets du parallélisme, comme la répartition ou l'équilibrage de charge ne seront pas abordés dans ce paragraphe. Ils seront explicités tout au long des sections suivantes de ce chapitre (paragraphe 4.3), dans le cadre des applications à la recherche de solution(s) dans les CSP.

1.1 Modèles de programmation parallèle

De nombreux modèles de calcul parallèle ont été proposés. Cependant l'un d'entre eux prédomine [Coo 85] : le modèle PRAM (*parallel random access machine*[1]).

Différentes terminologies sont utilisées pour caractériser un algorithme parallèle. Deux critères importants permettent de classifier les différents modèles du parallélisme :

- le type d'*accès à la mémoire* ;
- le type de *synchronisation* entre les programmes des processeurs.

Dans les paragraphes suivants, nous développons la classification du parallélisme par le type d'accès à la mémoire et par le type de synchronisation, puis nous présentons les principales architectures de machine parallèle existantes.

1.1.1 Classification par l'accès mémoire

Le modèle PRAM entre dans cette classification. Schématiquement, il considère l'ordinateur comme une machine constituée de n processeurs *synchrones* qui accèdent à une mémoire partagée (*cf.* figure 4.1). Toutes les communications entre les processeurs sont assimilées à un accès à une portion spécifique de la mémoire partagée.

Le modèle DRAM, moins utilisé, considère que chaque processeur synchrone accède à une mémoire locale en temps constant. Les communications entre les processeurs se font par envoi de messages explicites selon un réseau de communication.

Une sous-classification du modèle PRAM apparaît quand on s'intéresse à la manière avec laquelle les processeurs accèdent à un mot en mémoire partagée. Lorsque deux processeurs tentent de lire ou d'écrire dans un même mot de la mémoire, il se produit un conflit : au moment de la lecture, il est nécessaire de savoir si on considère qu'ils lisent tous les deux en même temps, ou si l'un d'eux lit avant l'autre. Le problème est encore plus crucial pour une écriture : si les deux processeurs tentent d'écrire le même mot, l'ordonnancement des deux écritures n'est pas très important, mais si les deux mots à écrire sont différents, il est impossible que les deux écritures soient simultanées. On distingue ainsi plusieurs sous-modèles pour les PRAM :

[1] en français : machine parallèle à accès mémoire aléatoire.

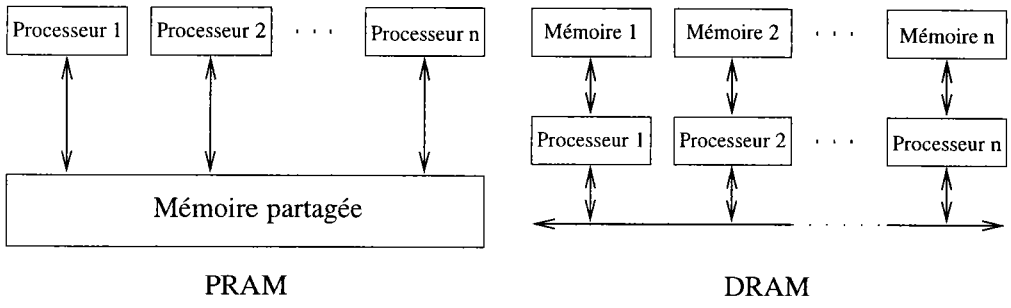


Figure 4.1 : Schéma des architectures du modèle PRAM et du modèle DRAM. Les communications sont symbolisées par des flèches.

1. **CRCW–PRAM** : (pour *concurrent read / concurrent write*). Dans ce modèle, on suppose que les processeurs peuvent lire et écrire simultanément dans un même mot mémoire. Le problème de l'écriture concurrente peut être résolu de différentes façons :
   - *Commune - CRCW* : l'écriture concurrente n'est possible que si tous les processeurs accédant à un mot mémoire tentent d'écrire la même donnée.
   - *Arbitraire - CRCW* : un choix arbitraire est fait sur le processeur – et donc la donnée – qui écrira dans un mot mémoire en cas d'écriture concurrente.
   - *Prioritaire - CRCW* : le choix sur le processeur n'est plus arbitraire, mais défini par une relation d'ordre total entre les différents processeurs.
   - *Calculée - CRCW* : un calcul spécifique est réalisé entre les différentes valeurs à écrire, et c'est le résultat qui est stocké dans le mot mémoire concerné. La plupart des auteurs considèrent des fonctions de résolution de conflits du type somme, minimum, maximum ou opérations booléennes.
2. **CREW–PRAM** : (pour *concurrent read / exclusive write*). Dans ce modèle, tous les processeurs peuvent lire simultanément depuis un même mot mémoire, mais un seul peut y écrire à la fois. Le problème de la simultanéité de l'écriture est ainsi contourné. En fait, la simultanéité de la lecture n'a pas d'influence sur l'écriture de l'algorithme, mais posera des problèmes pour la réalisation pratique de la machine.
3. **EREW–PRAM** : (pour *exclusive read / exclusive write*). L'accès à un mot mémoire ne peut se faire simultanément par plusieurs processeurs. Ce modèle est important d'un point de vue pratique, car c'est le seul qui soit réalisable physiquement.

Ces trois[2] sous-modèles ne sont pas indépendants. On définit en effet, la relation :

$$\text{EREW} \subset \text{CREW} \subset \text{CRCW}$$

Le modèle théorique CRCW est donc plus général que le modèle pratique EREW, mais il est possible de simuler un programme CRCW sur une machine EREW. La différence

[2]on pourrait définir un modèle PRAM - ERCW par la même nomenclature, mais il n'aurait pas de véritable signification d'un point de vue fonctionnel.

porte en fait sur la complexité puisque la simulation de chaque accès concurrent induira un surcoût de synchronisation. On admet que le rapport entre les deux complexités est logarithmique.

La classification par le type d'accès mémoire reste théorique car elle ne précise pas comment la mémoire partagée est gérée. Dans la pratique, il est impossible de construire des machines avec une mémoire centralisée si le nombre de processeurs est trop important. La mémoire des machines parallèles est donc répartie entre les processeurs, et c'est un mécanisme de communication par envoi de messages qui simule la mémoire partagée.

1.1.2 Classification par la synchronisation des processus

Une seconde classification souvent utilisée dans la littérature sur les modèles de programmation parallèle porte sur la synchronisation des processus. Dans les modèles de classification par l'accès mémoire, les processeurs sont synchrones. c.-à-d. ils exécutent tous la même instruction en même temps. Ici, on distingue les modèles en fonction du type de programme qui est exécuté par la machine parallèle :

1. **SIMD** pour *single instruction, multiple data*, ou synchrone : chaque processeur possède sa propre mémoire locale à laquelle il peut accéder en temps O(1). Il peut aussi accéder à la mémoire locale des autres processeurs mais en un temps non nul qui est déterminé par l'architecture de la machine. De plus, chaque processeur peut accéder à une mémoire globale en un temps comparable à celui de l'accès à la mémoire locale des autres processeurs.

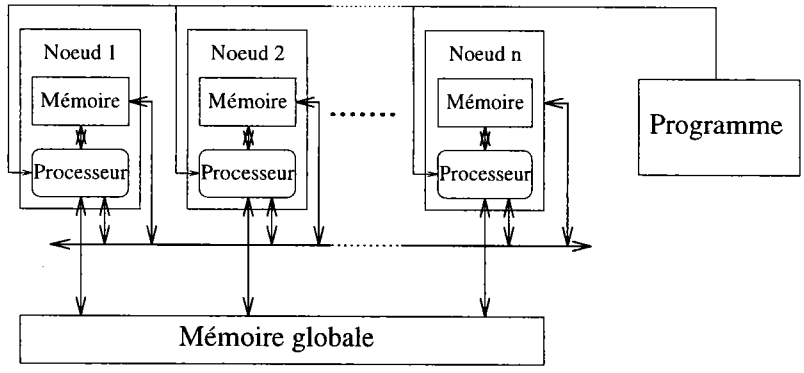


Figure 4.2 : Schéma de l'architecture SIMD. Les flux de données suivent les flèches.

L'appellation SIMD (*cf.* figure 4.2) signifie que les processus exécutent tous la même instruction en même temps. Tous les processus sont indexés et toutes les données locales sont référencées par l'index du processus qui les possède. Ce modèle peut être assimilé au modèle EREW-PRAM puisqu'il n'autorise pas l'accès concurrent aux mots de la mémoire partagée.

On parlera parfois de *machines vectorielles* pour désigner les architectures SIMD car le programme travail sur des vecteurs, et non sur des données scalaires comme dans une machine séquentielle.

2. **MIMD** pour *multiple instructions, multiple data*, ou asynchrone : chaque processeur possède sa propre mémoire locale à laquelle il peut accéder en temps O(1). Tous

les processeurs sont indépendants et n'accèdent pas à une mémoire globale (*cf.* figure 4.3). Les seules coopérations possibles entre les processeurs se font par l'envoi de messages. On a donc des communications explicites, avec un ordre *Envoyer* qui doit être appareillé avec un ordre *Recevoir* pour provoquer une communication, et donc une synchronisation entre deux processeurs. De plus, chaque processeur exécute son propre programme, indépendamment des autres en dehors des points de synchronisation. On notera que le modèle MIMD est proche dans sa conception, du modèle DRAM.

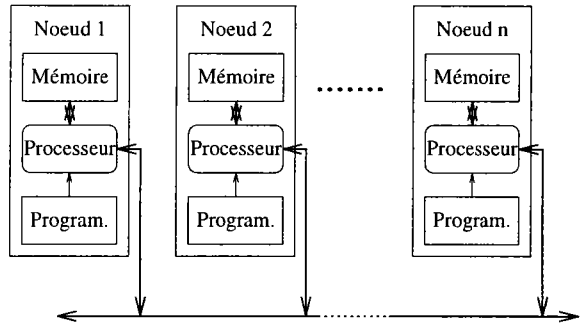


Figure 4.3 : Schéma de l'architecture MIMD. Les communications sont explicites et suivent un réseau spécial (symbolisé par des flèches).

Dans les années 70-80, le modèle SIMD était très utilisé car les programmes sont plus faciles à écrire puisque ne nécessitant pas de synchronisations explicites avec gestion des émissions/réceptions. De plus, la plupart des machines disponibles à cette époque utilisaient ce modèle de programmation[3]. Cependant, les progrès réalisés ces dernières années en algorithmique parallèle, et l'apparition d'outils de programmation parallèle sur réseau de stations de travail conduisent à un vif regain d'intérêt vis-à-vis du modèle MIMD.

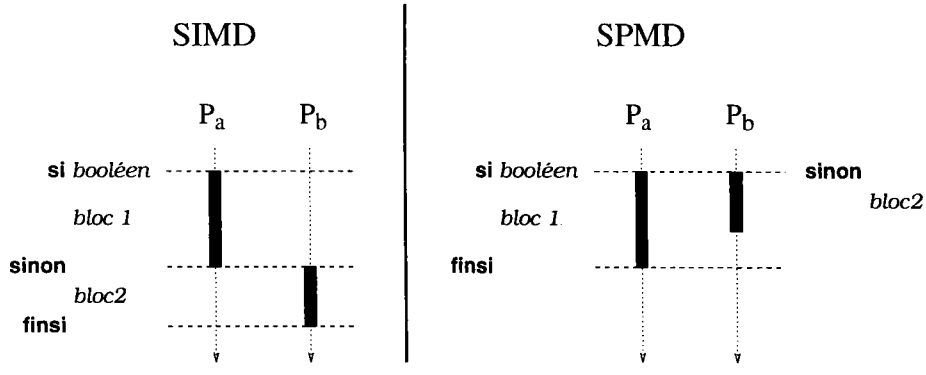


Figure 4.4 : Diagramme temporel de l'exécution d'une expression conditionnelle dans le modèle SIMD et dans le modèle SPMD.

Certains auteurs parlent d'un modèle intermédiaire entre SIMD et MIMD. Il s'agit du modèle SPMD (pour *single program, multiple data*) qui ne diffère réellement de SIMD que

[3]on peut citer dans cette catégorie, les machines de la famille du Cray I, ou la Connection Machine CM-1.

dans la réalisation des structures conditionnelles. La figure 4.4 présente les diagrammes temporels de la structure

si *booléen* **alors** *bloc* 1 **sinon** *bloc* 2 **finsi**

pour les modèles SIMD et SPMD. La synchronisation de l'exécution ne se fait plus sur chaque instruction en SPMD, mais à l'entrée et à la sortie de la structure de contrôle. Ainsi, les blocs d'instructions en exclusion mutuelle de la conditionnelle sont exécutés simultanément.

La programmation SPMD n'est donc pas un modèle de programmation en elle-même, mais une méthode définie pour programmer efficacement les machines MIMD avec des programmes SIMD, sans devoir concevoir toutes les communications explicites.

1.1.3 Architecture des machines parallèles

Si les modèles de programmation influent sensiblement sur l'écriture des programmes parallèles, et donc sur leur complexité, l'architecture joue un rôle non négligeable dans la complexité d'un programme.

La notion fondamentale qui entre en ligne de compte dans la complexité est le diamètre du graphe de communication. Intuitivement, c'est la longueur en nombre d'arêtes du plus court chemin que devra parcourir un message entre deux processeurs. Ainsi, chaque communication, qu'elle soit implicite ou explicite, se fera en temps O(*Diamètre*).

Nous présentons ici, un bref aperçu de diverses architectures, ainsi que leur diamètre respectif, exprimé en fonction du nombre n de processeurs (figure 4.5) :

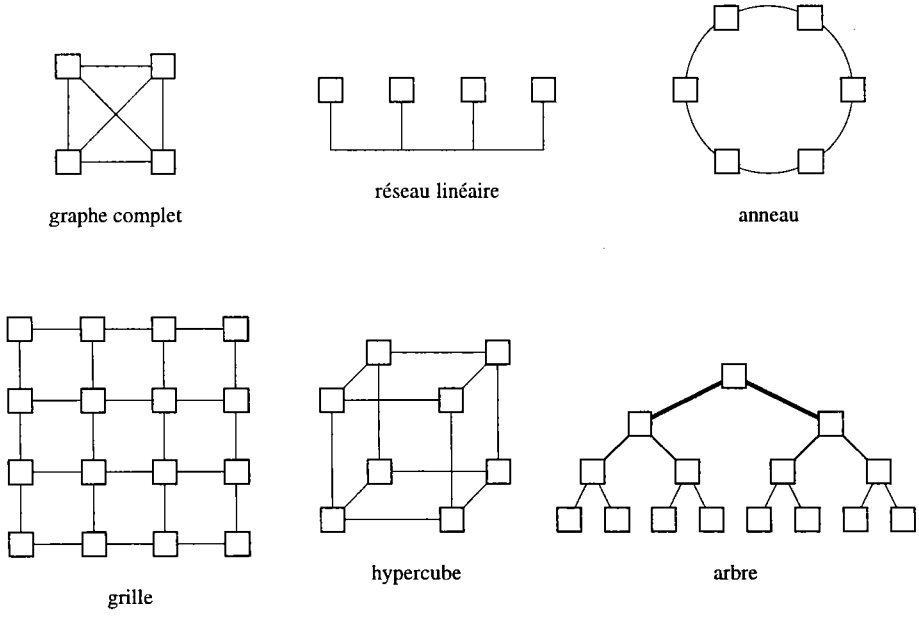


Figure 4.5 : Schéma de diverses architectures parallèles.

- **Graphe complet** : le diamètre du réseau est de 1. C'est le cas idéal où les communications prennent un temps O(1) entre chaque processeur – ce qui rameme aux modèles théoriques. Malheureusement, la gestion de toutes les communications interdit un grand nombre de processeurs, si bien qu'il n'existe pas de machine reprenant cette architecture avec un nombre important de processeurs.

- **Réseau linéaire** : le diamètre est de 1. Toutefois, le bus de communication ne permet qu'un seul échange simultané entre deux processeurs, ce qui augmente le coût pratique moyen des communications. On trouve cette architecture dans des machines avec un nombre restreint de processeurs (< 8) ou sur les réseaux de stations de travail avec programmation MIMD. Dans ces architectures, la faiblesse du réseau de communication est généralement compensée – en partie – par une puissance élevée des nœuds de calcul.

- **Anneau** : le diamètre est de $\frac{n}{2}$. On retrouve aussi des structures en hiérarchies d'anneaux car elles sont faciles à mettre en œuvre et restent d'une complexité raisonnable.

- **Grille** : le diamètre est de $2(\sqrt{n}-1)$. C'est une architecture assez répandue, de par sa simplicité de construction. Des variantes de la grille existent en trois dimensions – diamètre de $3(\sqrt[3]{n}-1)$. De plus, les « bords » de la grille peuvent être reliés entre eux pour former un tore.

- **Hypercube** : le diamètre est de $\log n$. Cette complexité est considérée comme optimale car elle constitue un bon compromis entre le graphe complet qui est le cas idéal et la faisabilité technique. Cette structure de communication revêt un caractère particulier puisque c'est elle qui est considérée pour le calcul de complexité d'un algorithme EREW. Elle est de plus utilisée en architecture de base pour de nombreux algorithme SIMD [NS 81].

- **Arbre** : le diamètre est de $2\log n$. On retrouve souvent cette architecture avec des liens de communication de bande passante différente selon le niveau des processeurs (comme dans la figure 4.5) afin d'éviter les goulots d'étranglement. Par ailleurs, certaines machines ne disposent de nœuds de calcul que sur les feuilles de l'arbre, les nœuds internes ne se chargeant que des communications (cas de la Connection Machine CM-5).

1.2 Complexité parallèle

Avant d'entrer dans des considérations purement théoriques, nous introduirons quelques définitions de base et conventions spécifiques à l'algorithmique parallèle. Ensuite nous définirons les deux grandes classes de complexité des algorithmes parallèles.

1.2.1 Définitions

Les deux facteurs qui déterminent la complexité d'un algorithme parallèle A pour la résolution d'un problème P sont :

1. **la surface** notée $H(A)$, qui définit le nombre de processeurs nécessaires pour exécuter le pas de programme qui exige le plus grand nombre de processeurs ;

2. **le temps**, noté $\mathrm{T}_{\shortparallel}(A)$ et qui correspond au nombre de pas nécessaires pour exécuter l'algorithme A avec $H(A)$ processeurs.

À partir des ces deux facteurs, on définit les notions suivantes :

Définition 4.1 *On appelle* **travail** *d'un algorithme parallèle A, la quantité $W(A)$ définie par :*

$$W(A) = H(A) \times T_{\shortparallel}(A).$$

On notera que pour un algorithme séquentiel ($H(A) = 1$), le travail correspond au temps séquentiel de l'algorithme.

Définition 4.2 *Un algorithme parallèle est dit* **efficace** *si son temps d'exécution est poly-logarithmique et si son travail correspond au meilleur algorithme séquentiel connu multiplié par un facteur poly-logarithmique.*

Définition 4.3 *Un algorithme parallèle est dit* **optimal** *s'il est efficace et si son travail est du même ordre que le travail du meilleur algorithme séquentiel connu.*

Par convention, la complexité d'un algorithme parallèle sera notée :

$$\mathrm{O}_{\shortparallel}(\mathrm{T}_{\shortparallel}(A), H(A)).$$

Un algorithme parallèle apparaîtra comme plus général qu'un algorithme séquentiel car il est toujours possible de simuler le comportement de plusieurs processeurs sur un seul, alors qu'il n'est généralement pas possible d'exécuter efficacement un algorithme séquentiel sur plusieurs processeurs. Ce principe de simulation a été explicité par Brent dans [Bre 74].

Théorème 4.4 (Brent) *Soit A un algorithme parallèle de complexité $O_{\shortparallel}(T_{\shortparallel}(A), H(A))$. On peut simuler l'exécution de A sur $\frac{H(A)}{\omega}$ processeurs ($\omega > 1$) avec une complexité en $O_{\shortparallel}(\omega \times T_{\shortparallel}(A), \frac{H(A)}{\omega})$ sur une machine à mémoire partagée.*

Sur le plan pratique, les algorithmes parallèles sont aussi comparés avec leur équivalent séquentiel. Ainsi, la performance d'un programme parallèle se mesurera comme :

Définition 4.5 *L'***accélération** *d'un algorithme est donnée par le rapport entre son temps d'exécution en séquentiel sur son temps d'exécution en parallèle ($Acc = \frac{T_{seq}}{T_{par}}$).*

La notion d'accélération est toutefois relative puisque dépendante du nombre de processeurs nécessaires pour le calcul parallèle. On définit donc une notion absolue :

Définition 4.6 *L'***efficacité** *d'un algorithme est donnée par le rapport entre son accélération et le nombre de processeurs requis ($Eff = \frac{Acc}{N_{proc}}$).*

L'efficacité d'un algorithme – en pourcentage – exprime la part du temps d'exécution nécessaire pour résoudre le problème. Le reste du temps – le complément à 100 % – exprime le surcoût dû à la parallélisation de l'algorithme. Un algorithme robuste[4], verra son efficacité décroître faiblement avec l'augmentation du nombre de processeurs. À l'inverse, un algorithme sera dit *fragile* si son efficacité diminue rapidement quand le nombre de processeurs augmente.

[4]on retrouve généralement dans la littérature de langue anglaise, l'appellation « *scalable* » qui est d'ailleurs parfois utilisée comme néologisme dans la littérature en langue française.

1.2.2 Complexité théorique

La question qui vient le plus rapidement à l'esprit quand on parle de complexité parallèle est de savoir quels sont les problèmes qui sont *intrinsèquement parallèles*, c'est à dire qui sont efficacement résolus en parallèle. Cette classe a été formalisée par Nicholas Pippenger dans [Pip 79] et baptisée par la suite NC pour « *Nick's Class* » [Coo 85].

Les problèmes de la classe NC seront résolus par un algorithme dont la complexité parallèle sera en $O_{\parallel}(\log^{O(1)} n, n^{O(1)})$. On a ainsi $NC \subseteq P$, mais l'inclusion stricte n'a pas été démontrée car on ne sait pas si tous les problèmes P sont intrinsèquement parallèles. Il y a d'ailleurs de fortes raisons de penser le contraire, c.-à-d. $NC \neq P$.

De même que la machine de Turing représente le modèle de référence pour la complexité séquentielle, il existe un modèle rudimentaire utilisé pour définir les notions de complexité parallèle théorique : *le modèle booléen.*

Définition 4.7 *Une* **machine booléenne** *est une famille* uniforme (B_n) *de graphes booléens orientés et acycliques (DAG*[5]*) telle que* B_n *a* $n^{O(1)}$ *entrées. Un nœud du circuit est une porte logique.*

La surface $H(n)$ d'une machine booléenne est le nombre de nœuds du circuit B_n, et le temps $T_{\parallel}(n)$ sa profondeur.

Définition 4.8 *Un* **oracle** *pour une fonction* f *est un nœud ayant* r *entrées* $e_1, \ldots, e_r$ *et* t *sorties* $s_1, \ldots, s_t$ *et qui calcule* $(s_1, \ldots, s_t) = f(e_1, \ldots, e_r)$.

On assimile la profondeur (c.-à-d. la complexité temporelle) d'un oracle à $\log(rt)$.

Définition 4.9 *On dit qu'une fonction (ou problème)* f *est* **NC-réductible** *à une fonction* g *s'il existe une famille uniforme de circuits qui calcule* f *en temps logarithmique et dont les nœuds sont, soit des portes booléennes, soit de oracles permettant de calculer* g.
f *NC-réductible à* g *se note* $f \leq_{NC} g$.

Cette NC-réductibilité permet de définir :

Définition 4.10 *Soit une classe de problèmes* Λ. *On dit qu'une fonction* f *est :*

- *NC-dure pour l'ensemble* Λ *(*Λ**-dure**) *ssi* $\forall g \in \Lambda : g \leq_{NC} f$;
- *complète pour* Λ *(*Λ**-complète**) *si* f *est* Λ*-dure et si* $f \in \Lambda$.

On définit ainsi la classe des problèmes *P-complets* comme la classe des problèmes contenant le moins de parallélisme intrinsèque, c'est à dire qui sont inhéremment séquentiels. On pourra faire l'analogie avec les problèmes NP-complets et P dans le cadre séquentiel : si un seul problème P-complet peut être résolu par un algorithme de classe NC, alors on a $P \subset NC$.

[5] pour *directed acyclic graph* : graphes orientés sans cycles.

2 Filtrage parallèle des CSP

Nous présentons dans ce paragraphe quelques résultats importants sur la parallélisation des méthodes de filtrage des CSP. En fait, la plupart des méthodes abordées dans le paragraphe 4 du chapitre 1 ont été « oubliées » par les différents auteurs qui se sont essentiellement concentrés sur l'établissement de l'arc-consistance, et accessoirement sur la chemin-consistance.

Les principaux résultats théoriques sur la parallélisation de l'algorithme *AC* sont présentés au paragraphe 2.1. Nous abordons ensuite les algorithmes parallèles pour l'établissement de l'arc et de la chemin consistance dans un modèle SIMD, puis dans un modèle MIMD (paragraphes 2.2 et 2.3).

2.1 Considérations théoriques sur *AC*

Le premier résultat théorique concernant la parallélisation du problème de l'établissement de l'arc-consistance a été donné par Kasif dans [Kas 89] :

Théorème 4.11 (Kasif) *AC est en $\Omega(nd)$, où n est le nombre de variables dans le* CSP, *et d est le nombre maximal de valeurs par variable.*

Ce théorème signifie qu'on ne peut espérer un algorithme parallèle efficace pour le résoudre. Cette borne minimale de complexité a été atteinte par Cooper et Swain [CS 92], avec la conception de l'« algorithme » *AC-chip*. Il s'agit en fait de la construction simulée d'un composant logique qui établit l'arc-consistance sur un CSP. Elle permet d'exhiber le parallélisme maximal du problème de l'arc-consistance.

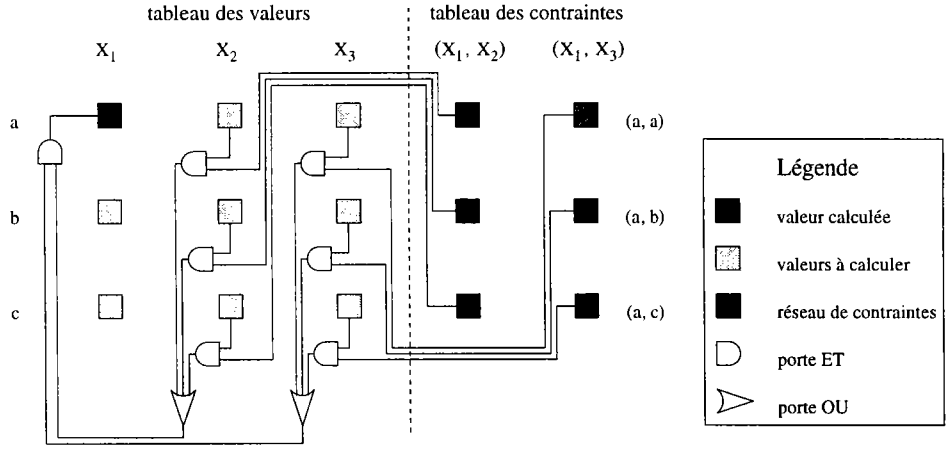


Figure 4.6 : Schéma partiel de *AC-chip*.

La figure 4.6 présente un schéma logique partiel d'*AC-chip*. Les valeurs sont entrées dans les bascules (carrés sur la figure). Au bout d'un nombre fini de tops horloge, toutes les bascules sont stabilisées à une valeur et représentent le CSP arc-consistant (ou inconsistant selon le cas). Si le nombre maximal de tops horloge (complexité temporelle) est de dn, il sera toujours minimal pour un CSP donné.

Le principal inconvénient d'*AC-chip* est sa complexité spatiale en $O(d^2n^2) = O(d^2m)$ qui le rend inutilisable sur des machines classiques.

2.2 Algorithmes SIMD pour *AC* et *PC*

Les premiers algorithmes SIMD développés pour l'établissement de l'arc-consistance en parallèle sont dûs à Samal et Henderson [SH 87]. Il s'agit d'une série d'algorithmes basés sur les versions séquentielles AC-1 à AC-4. Tous font appel à une fonction de révision de l'arc-consistance entre deux variables (*PFiltre*) qui est présentée dans l'algorithme 4.1.

Le parcours des valeurs du domaine à filtrer est réalisé en parallèle (ligne 4), soit avec une complexité en $O_{\shortparallel}(1, d)$. Pour chaque valeur, la recherche d'un support est réalisée en parallèle (ligne 6). La complexité globale de la fonction *PFiltre* est donc en $O_{\shortparallel}(1, d^2)$. On notera que l'affectation de la ligne 8 conduit à une écriture concurrente (de la même valeur) puisque une valeur peut avoir plusieurs supports. La complexité ci-dessus s'applique donc pour une architecture de type CRCW.

Algorithme 4.1 *Filtrage d'un domaine par consistance d'arc en parallèle.*

```
1  fonct PFiltre (i, j) ↦ booléen ≡                    /* Filtre D_i par rapport à D_j */
2    suppression ⟵ faux
3    soit C_ij = {X_i, X_j} ∈ C
4    pour tout x ∈ D_i faire en parallèle
5          support[x] ⟵ faux
6          pour tout y ∈ D_j faire en parallèle
7                si (x, y) ∈ R_ij alors
8                  support[x] ⟵ vrai finsi finpour
9          si ¬support[x] alors
10           D_i ⟵ D_i \ {x}
11           suppression ⟵ vrai finsi finpour
12   retourner suppression.
```

Des trois algorithmes parallèles *PAC-1*, *PAC-3* et *PAC-4* définis dans [SH 87], nous ne présenterons que le premier car il est le plus simple (*cf.* algorithme 4.2). Son principe consiste à appliquer la procédure *PFiltre* à tous les arcs du graphe tant que des changements apparaissent dans les domaines des variables. Il a une complexité optimale dans le pire des cas en $O_{\shortparallel}(nd, n^2d^2)$.[6] Outre sa simplicité, *PAC-1* présente l'intérêt d'exhiber le maximum de parallélisme.

Algorithme 4.2 *Établissement de l'arc-consistance parallèle (algorithme SIMD).*

```
1  proc PAC-1 ≡
2    change ⟵ vrai
3    tant que change faire
4            change ⟵ faux
5            pour i de 1 à n faire en parallèle
6                  pour j de 1 à n faire en parallèle
7                        change ⟵ change ∨ PFiltre (i, j)
8                  finpour finpour fintant que.
```

[6] on se place ici dans le modèle CRCW puisque la ligne 7 de l'algorithme impose une écriture de type OU-parallèle.

Contrairement au contexte séquentiel, le fait de recalculer *tous* les arcs à chaque fois qu'un changement intervient ne constitue pas un handicap car ce calcul est réalisé en O(1). De plus, comme un minimum de traitements est réalisé entre chaque étape, les résultats expérimentaux montrent que *PAC-1* est meilleur que les autres versions[7] qui constituent des versions parallèles d'algorithmes plus efficaces en séquentiel.

Henderson associé à d'autres auteurs a étendu les idées de programmation de *PAC-1* pour la chemin-consistance [SHZ+ 91]. Ils définissent l'algorithme SIMD *PPC-1* qui exhibe un maximum de parallélisme. Sa complexité est en $O_{\parallel}(n^2d^2, n^2d^2)$, mais les résultats expérimentaux ont montré un coût pratique de *PPC-1* sensiblement identique à celui de *PAC-1* sur les mêmes problèmes. Il apparaîtrait donc plus intéressant d'utiliser un filtrage par chemin-consistance dans un contexte de calcul parallèle. Cependant, les tests effectués portent sur des problèmes spécifiques dont la pertinence est maintenant contestée pour l'évaluation pratique des algorithmes de résolution (*cf.* chap. 1, paragraphe 7.2).

2.3 Algorithmes MIMD pour *AC*

L'élaboration d'algorithmes MIMD est souvent plus complexe que celle d'algorithmes SIMD car il faut tenir compte des communications explicites entre les processeurs (ainsi que des synchronisations associées). De plus, il est plus difficile et souvent plus coûteux en pratique d'exprimer un parallelisme maximum car les communications représentent souvent le goulot d'étranglement de la machine.

Algorithme 4.3 *Établissement de l'arc-consistance parallèle (algorithme MIMD).*

```
1  proc DSPAC-1-hôte ≡
2    change ⟵ vrai
3    tant que change faire
4              change ⟵ faux
5              pour i de 1 à n faire en parallèle
6                    change ⟵ change ∨ DSPAC-1-nœud (i) finpour fintant que.
7  fonct DSPAC-1-nœud (i) ↦ booléen ≡
8    pour tout j/{X_i, X_j} ∈ C faire
9          change ⟵ Filtre (i, j)
10         si change alors
11           pour tout k/{X_i, X_k} ∈ C faire
12                 envoyer D_i à k finpour finsi finpour
13   retourner change.
```

L'algorithme MIMD que nous présentons ici est issu d'une série d'algorithmes dûs à Conrad associé à divers auteurs [CA 95, CBB 91]. Comme pour le modèle séquentiel, les différentes versions sont le résultat d'une série de transformations pour tenter de réduire la complexité théorique des algorithmes. Ils ont ainsi défini trois versions : *DSPAC-1*[8], *DSPAC-2* et *DSPAC-3*. Nous ne détaillerons que la première version (algorithme 4.3) pour sa simplicité de fonctionnement.

La révision d'une consistance d'arc ne se fait plus en parallèle comme dans l'algorithme *PAC-1*, mais avec la fonction séquentielle de l'algorithme 1.2. Quand un processeur modifie

[7] *PAC-3* et *PAC-4* on la même complexité théorique que *PAC-1*.

[8] *DSPAC*: *distributed static parallel arc-consistency.*

le domaine dont il a la charge (ligne 9–10), il avertit les processeurs qui sont concernés par ce changement (ligne 11–12). La complexité théorique des algorithmes *DSPAC-1* et *DSPAC-2* est en $O_{\shortparallel}(d^3n^3 + dn^2\chi, n)$, où χ est le coût d'une communication entre deux processeurs (et dépend donc de l'architecture). Celle de *DSPAC-3* n'est que de $O_{\shortparallel}(d^3n^2 + dn^2\chi, n)$. Toutefois, les résultats expérimentaux ont montré que *DSPAC-1* est le plus performant.

3 Synthèse de contraintes

De même que dans un cadre purement séquentiel, la synthèse de contraintes a peu inspiré la recherche en termes de parallélisation. Ce désintérêt est probablement dû au fait que cette méthode n'est performante que si on cherche toutes les solutions à un problème, alors que la plupart des travaux de recherche dans le domaine des CSP sont orientés vers la mise en évidence d'une seule solution. Tant les méthodes de synthèse incrémentales que les méthodes en treillis ont été expérimentées en parallèle.

3.1 Synthèse incrémentale

À notre connaissance, la méthode de synthèse par invasion de Seidel (*cf.* chap. 1, paragraphe 3.1) n'a pas été développée dans une version parallèle. Par contre, l'algorithme de Guesgen, Ho et Hilfinger [GHH 92, HHG 93] de synthèse par marquage[9] a été directement défini et expérimenté dans un contexte parallèle. Son implémentation est basée sur une propagation distribuée des marques dans les variables. Le programme est écrit pour une machine à mémoire partagée avec synchronisations implicites entre les processeurs (PRAM).

L'accélération obtenue par les auteurs est de 6 pour 8 processeurs. Toutefois, ils restent peu diserts sur les CSP utilisés pour expérimenter leur algorithme, si bien qu'il est difficile de comparer leur algorithme avec d'autres résultats.

3.2 Synthèse en treillis

L'algorithme de synthèse en treillis de Freuder (*cf.* chap. 1, paragraphe 3.2) a été développé dans une version parallèle par Hower dans [How 90]. Tous les nœuds d'un ordre donné sont calculés sur différents processeurs virtuels, et les propagations des inconsistances se font par échange de valeurs entre les processeurs qui maintiennent les informations relatives à chaque nœud.

Le parallélisme potentiel d'un tel algorithme est limité par le nombre de nœuds dans le treillis. De plus, l'équilibre de charge est difficile à obtenir dans la phase de propagation, et l'algorithme lui-même requiert de nombreuses communications entre les processeurs. L'auteur s'est surtout intéressé à des considérations théoriques quant à l'algorithme de synthèse en treillis dans [How 90], et l'a testé sur des exemples pratiques dans [HJ 94], sur un réseau de transputers en architecture MIMD. Il reporte des accélérations quasi-linéaires si les problèmes sont suffisamment grands[10].

[9] en anglais : *tagging method.*

[10] le type de problèmes utilisés pour les expérimentations n'est cependant pas précisé.

4 Recherche énumérative

Dans le monde des CSP, la littérature en parallélisme est particulièrement peu abondante. En fait, il faut élargir un peu le sujet et étudier les méthodes de parallélisation des programmes logiques (PROLOG, optimisation combinatoire, ...) pour trouver des concepts qui se rapprochent de la recherche parallèle de solution(s) pour les CSP.

Trois voies de parallélisation de la recherche se dégagent de l'étude des algorithmes proposés dans la littérature. Les deux premières que nous présentons – plusieurs recherches en parallèle au paragraphe 4.1 et parallélisation de la vérification de consistance au paragraphe 4.2 – sont plutôt anecdotiques, et constituent essentiellement des explorations menant à des « voies sans issues ». La principale voie explorée concerne la parallélisation du parcours de l'arbre de recherche (paragraphe 4.3), et a été montrée expérimentalement comme sensiblement plus prometteuse [HW 94, LHB 92].

4.1 Méthodes des recherches concurrentes

La méthode expérimentée par Hogg et Williams dans [HW 94] consiste à chercher une solution pour un problème donné indépendamment sur plusieurs processeurs. Dans ce contexte, chaque processeur exécute le même algorithme de recherche (par exemple *FC*) avec une heuristique d'ordre aléatoire sur les variables. Le premier processeur qui trouve une solution arrête alors tous les autres.

Avantages le principal – et peut-être le seul – point positif de cette méthode, est que les exécutions sont totalement indépendantes sur chaque processeur, excepté la synchronisation pour la terminaison de l'algorithme. De plus, il n'y a pas de phénomène de famine des processeurs puisque le fait que l'un des processeurs termine sa recherche signifie la terminaison globale de l'algorithme.

Inconvénients la méthode n'est en général pas efficace car elle pose des problèmes à la fois sur le plan du parallélisme et sur le plan de la recherche :

- chaque processeur calcule la totalité d'un arbre de recherche, ce qui signifie qu'en multipliant le nombre de processeurs, on multiplie d'autant la quantité de travail à accomplir par l'algorithme parallèle ;
- l'heuristique aléatoire permet une variance dans les exécutions parallèles, et donc un gain potentiel de l'une par rapport aux autres, mais ne permet pas de tirer parti des heuristiques performantes comme *MRV*.

L'efficacité de la méthode repose sur l'espérance qu'une recherche – sur un processeur séparé – termine significativement plus rapidement que les autres. Elle nécessite donc une variance importante entre les différents arbres de recherche. Or, les auteurs ont montré que la variance est particulièrement faible dans la zone des problèmes difficiles[11], c.-à-d. précisément où l'apport du parallélisme devrait être le plus important.

Il est à noter que, si l'accélération est à peu près constante en fonction du nombre de processeurs, elle peut être sub-linéaire comparée à une heuristique performante comme *MRV*.

[11] voir le paragraphe 7.2 du chapitre 1.

4.2 Parallélisation de la vérification de consistance

Globalement, la recherche énumérative se schématise par un arbre de recherche dans lequel chaque nœud est une procédure complexe. Les deux approches de parallélisation naturelles vis-à-vis de cette structure consistent à :

1. paralléliser les traitements à l'intérieur de chaque nœud ;
2. distribuer l'arbre de recherche sur plusieurs processeurs.

L'objet de ce paragraphe concerne le premier point – le second sera traité dans le paragraphe suivant. La parallélisation des traitements à l'intérieur d'un nœud a été étudiée dans le cadre des CSP par Luo, Hendry et Buchanan [LHB 92]. La recherche d'une solution se déroule séquentiellement, mais à chaque instanciation, les contraintes sont vérifiées en parallèle. La figure 4.7 schématise une recherche de solution(s) avec 4 processeurs.

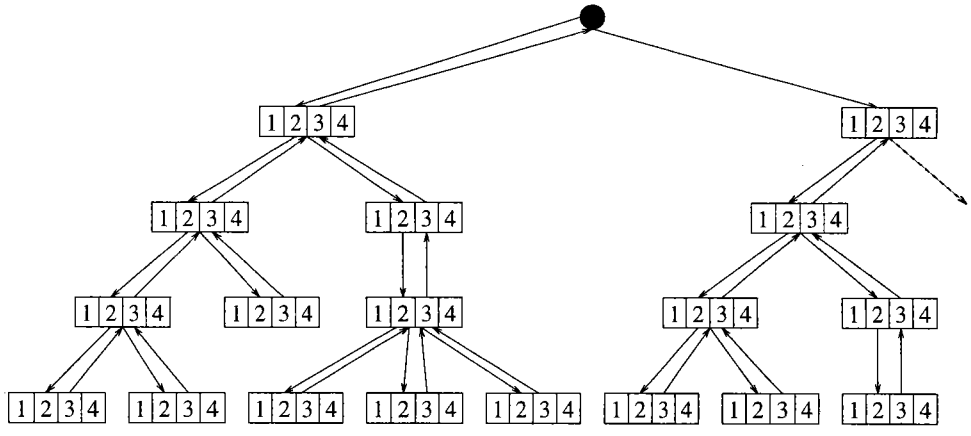


Figure 4.7 : Schéma partiel de la vérification parallèle de consistance. La recherche suit les flèches. Le travail de chaque processeur est cantonné dans les rectangles numérotés.

Un tel système impose naturellement de fréquentes synchronisations entre les processeurs qui doivent regrouper leur travail après chaque instanciation. Ceci peut constituer un handicap dans le cas de machines parallèles dont les communications sont coûteuses, comme les machines MIMD en général. On réservera donc cette technique aux architectures SIMD.

L'organisation de l'algorithme se fait par une succession de blocs parallèles qui portent sur un nombre limité de traitements. Ceci induit donc un parallélisme potentiel limité. De plus, les contraintes concernées lors de chaque instanciation sont différentes, ce qui peut conduire à répartir le travail de recherche de différentes manières :

- Allouer le travail sur les processeurs à chaque instanciation. On a ainsi une bonne répartition de la charge, mais qui se fait au détriment d'un système d'allocation lourd à gérer, et donc d'une surcharge due à la gestion du parallélisme.
- Allouer statiquement le travail de vérification sur les processeurs. Cette option induit un mauvais équilibre de charge puisque toutes les contraintes ne seront pas vérifiées à chaque instanciation.

Les auteurs de [LHB 92] ont choisi la seconde option qui laisse certains processeurs inactifs. Ils proposent toutefois d'utiliser le temps perdu pour exécuter un algorithme de filtrage de façon plus ou moins prioritaire. La recherche s'exécute donc normalement, mais le temps qui n'est pas consommé par un processeur est mis à profit pour réaliser du travail supplémentaire qui pourra réduire le coût de la recherche par effet de bord.

Cette technique de parallélisation apparaît comme satisfaisante du point de vue conceptuel, notamment avec l'utilisation d'un algorithme de filtrage en tâche de fond, qui permet d'améliorer la recherche en utilisant les périodes de calcul qu'elle laisse libre. Toutefois, les expérimentations menées par les auteurs ont montré une faible accélération pour un petit nombre de processeurs. De plus, non seulement l'efficacité parallèle de l'algorithme diminue fortement avec l'augmentation du nombre de processeurs, mais elle s'accompagne, à partir d'un certain seuil[12] d'une augmentation du temps de calcul. Les auteurs expliquent ce phénomène par le nombre croissant de communications qui pénalisent fortement les performances de l'algorithme.

On pourra noter que dans cette même série d'expérimentations, les auteurs ont montré l'intérêt de coupler la recherche énumérative avec un filtrage par arc-consistance. Les tests avec différentes priorités pour l'algorithme « parasite » ont d'ailleurs montré que l'algorithme est globalement plus performant si le filtrage est aussi prioritaire que la recherche.

Dans le monde de la programmation logique, la parallélisation de la vérification de consistance correspond aux algorithmes de type « ET-parallèle » car plusieurs assertions sont vérifiées simultanément, et elles doivent toutes être vraies pour que la vérification continue dans la même branche. Dans la littérature plus générale sur le parallélisme, on retrouve parfois l'appellation « SÉQ de PAR » pour faire ressortir le fait que l'organisation est une suite SÉQuentielle de blocs de contrôle PARallèles [Bou 93].

4.3 Parallélisation du parcours de l'arbre de recherche

La parallélisation de l'arbre de recherche a été aussi peu étudiée dans le cadre des CSP que la parallélisation des vérifications de consistances. Elle fait par contre l'objet de nombreuses recherches dans le cadre de la programmation logique ou des techniques *branch and bound.*

Les algorithmes évoqués dans ce paragraphe ont pour principe général de distribuer l'arbre de recherche sur les différents processeurs. La figure 4.8 schématise la recherche de solution(s) avec 4 processeurs. On pourra comparer la répartition du travail des processeurs avec la figure 4.7 pour la parallélisation de la vérification de consistance.

Les techniques de parallélisation de l'arbre de recherche sont souvent regroupées sous le terme de « OU-parallèle » dans le cadre de la programmation logique car l'exploration des branches de l'arbre conduit à une solution au problème si au moins une branche aboutit à une solution. Dans la littérature plus générale sur le parallélisme, on pourra trouver aussi l'appellation « PAR de SÉQ » pour faire ressortir le fait que le contrôle global de l'algorithme est réalisé en processus PARallèles qui exécutent chacun un algorithme SÉQquentiel.

[12] dans [LHB 92], les auteurs reportent un temps de résolution moyen des problèmes testés de 6,6 secondes pour 1 processeur, de 2,21 pour 5 processeurs, et de 4,21 pour 15 processeurs.

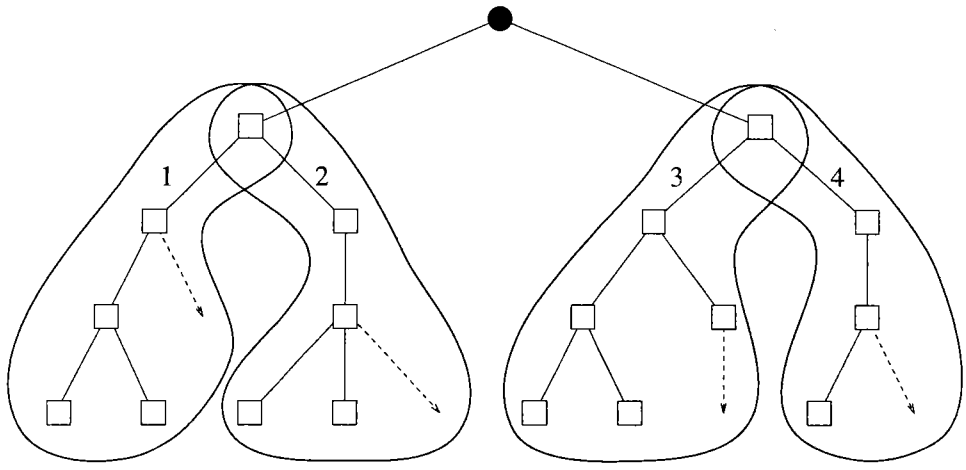


Figure 4.8 : Schéma de la parallélisation de la recherche. L'exploration affectée à chacun des processeurs est encerclée.

D'une manière générale, on pourra considérer que les parties séquentielles des algorithmes de recherche s'exécutent de manière indépendante d'un processeur à l'autre puisque les branches de l'arbre sont elles-mêmes indépendantes[13]. La principale difficulté de la parallélisation ne portera donc pas sur les synchronisations pour accéder à des parties communes de mémoire, mais sur le maintient d'une charge de travail permanente pour les processeurs, pendant toute la durée de la recherche. On distinguera deux types d'allocation de travail sur les processeurs :

1. *Allocation statique* ou « répartition de charge » qui consiste à donner du travail aux processeurs en début d'algorithme. Dans la littérature de langue anglaise, cette allocation est appelée *load sharing*. Les différents algorithmes de répartition pourront varier sur deux points importants : 1) quel(s) processeur(s) a(ont) la responsabilité de la répartition du travail ? 2) comment est divisée la charge globale de recherche ?

2. *Allocation dynamique* ou « équilibre de charge » qui consiste à répartir le travail de recherche au cours de l'exécution de l'algorithme (*load balancing*). Elle pose plusieurs questions auxquelles il faut répondre pour caractériser l'équilibrage de charge d'un algorithme :
   1) quand décider d'un équilibrage de charge ?
   2) avec quel processeur échanger du travail ?
   3) comment déterminer si une tâche doit être considérée comme divisible ?
   4) comment diviser une tâche ?
   5) comment transférer une tâche ?

Toutes ces questions seront détaillées dans les paragraphes suivants – en ce qui concerne la recherche de type « OU-parallèle ». Le lecteur pourra se reporter à [GS 93] (en français) ou [WLR 93] (en anglais) pour des survols des techniques de répartition et équilibrage de charge dans le cas général, ou [Ben 95, Cun 94] pour des problèmes plus spécifiques aux arbres de recherche.

[13] l'indépendance des branches de recherche n'est toutefois pas vérifiée dans les cas de techniques de recherche comme la mémorisation (*cf.* paragraphe 6.3 du chapitre 1).

4.3.1 Répartiteur de charge

Le rôle du répartiteur de charge est de définir le travail affecté au départ à chaque processeur. Les choix possibles sont en général limités aux techniques suivantes :

- *Répartiteur centralisé* : un seul processeur dispose de la charge globale du système et distribue séquentiellement le travail aux autres processeurs par des communications point à point [CM 94, dBKT 95, FM 87, Lau 93, NO 94, RK 87, San 95]. Cette technique est assez coûteuse car elle comporte une procédure inhéremment séquentielle, contraire à la philosophie du parallélisme. Elle est malgré tout parfois intéressante, lorsque le problème est difficilement séparable sans communications explicites.
- *Répartition distribuée* : chaque processeur calcule localement sa charge sans communication avec les autres processeurs [HS 95, NO 94]. Sur le plan du parallélisme, cette technique reste plus efficace car elle n'impose ni communications, ni séquentialisation du travail en début d'algorithme.

Bien qu'il soit tout à fait possible pour un algorithme de ne pas définir de répartition de charge initiale, mais de laisser le mécanisme dynamique allouer du travail sur chaque processeur, toutes les techniques présentées implémentent un mécanisme de répartition initiale.

4.3.2 Définition des tâches à répartir

En règle générale, les algorithmes proposés dans la littérature définissent autant de tâches à répartir qu'il y a de processeurs. La génération de ces charges se fait par un parcours *en largeur* de l'arbre de recherche, comme dans l'exemple de la figure 4.9.

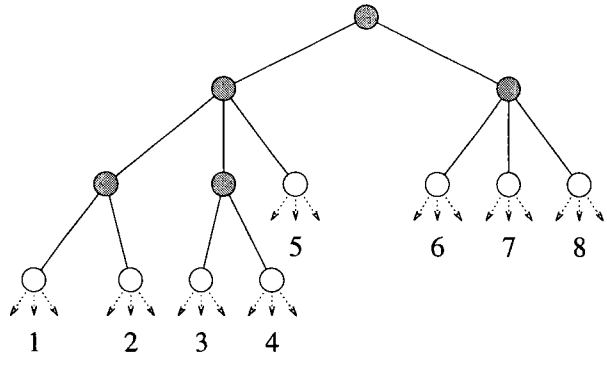


Figure 4.9 : Exemple de définition de tâches à répartir dans un arbre de recherche pour 8 processeurs.

Cette méthode est appelée *ILB* pour *initial load balancing* dans [NO 94]. Dans ce même article, Naganuma et Ogura proposent une méthode de génération de tâches plus efficace : au lieu de prévoir une seule tâche par processeur, ils en prévoient directement plusieurs, pour anticiper le manque de travail des processeurs. La répartition des différentes tâches se fait modulo le nombre de processeurs afin que ceux-ci aient des branches de recherche dérivées de « pères » différents.

Exemple 4.1 : Répartition modulo le nombre de processeurs.

Dans l'exemple de répartition de la figure 4.9, on peut utiliser les huit tâches définies pour 4 processeurs comme suit :

processeur P_1 : tâche 1 et 5	processeur P_3 : tâche 3 et 7
processeur P_2 : tâche 2 et 6	processeur P_4 : tâche 4 et 8

Cette répartition de charge prend le nom de *VLB* pour *virtual load balancing.*

4.3.3 Initiation d'un équilibrage

Les phases d'équilibrage de charge viennent interrompre le processus classique de recherche séquentielle dans les processeurs. Plusieurs méthodes permettent de déterminer le moment où un équilibrage doit être opéré. De façon théorique, il est nécessaire de rééquilibrer la charge lorsqu'un déséquilibre global apparaît sur l'ensemble des processeurs. Toutefois, il est très difficile (voire coûteux) de déterminer cette caractéristique pour l'ensemble des processeurs, si bien quen pratique les algorithmes se contentent de mesures locales.

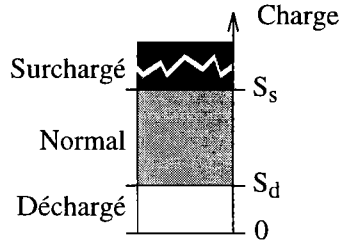


Figure 4.10 : Diagramme des différents états d'un processeur.

On considère généralement qu'un processeur peut, au cours d'un algorithme parallèle, osciller entre un des trois états suivants (*cf.* figure 4.10) :

- *déchargé* s'il a sensiblement moins de travail à réaliser que les autres processeurs (charge inférieure à un seuil de décharge S_d) ;
- *normalement chargé* ;
- *surchargé* s'il a sensiblement plus de travail à réaliser que les autres processeurs (charge supérieure à un seuil de surcharge S_s).

Le comportement classique d'un processeur qui sort de l'état de charge normale, est de demander un équilibrage. Si un processeur surchargé initie un équilibrage, on parlera alors d'équilibrage « source-initiative » ou *sender-initiative* dans la littérature de langue anglaise. Dans le cas où c'est un processeur déchargé qui est l'initiateur, on parlera d'équilibrage « serveur-initiative » ou *server-initiative.*

Toutefois, l'évaluation de la charge d'un processeur pour la recherche énumérative est hasardeuse car il est impossible de connaître à l'avance la forme d'un arbre qui n'a pas été exploré. La connaissance du nombre de nœuds non explorés n'est pas suffisante car un seul nœud peut engendrer une forte charge de calcul, alors qu'à l'opposé, un grand nombre de nœuds en attente peuvent déboucher sur un rapide manque de travail.

Cette difficulté conduit à définir le seuil de décharge des processeurs à $S_d = 0$, c.-à-d. qu'un processeur sera considéré comme déchargé seulement lorsqu'il n'aura plus de travail. À l'inverse, on ne définira pas de seuil de surcharge ($S_s = \infty$). Ainsi, seules les équilibrages de type *serveur-initiative* seront utilisés dans les algorithmes parallèles de recherche énumérative.

La mise en œuvre de l'équilibrage de charge dépend du type de machine utilisée. Dans un algorithme MIMD, l'initiation est souvent faite « à la demande » et ne concerne que le processeur inactif [CM 94, FM 87, HS 95, NO 94, RK 87]. Pour les algorithmes SIMD, l'activation d'une phase d'équilibrage porte automatiquement sur tous les processeurs, si bien que l'activation de la procédure suite à la demande d'un seul processeur peut ralentir globalement le mécanisme de recherche. Certains auteurs proposent un système d'activation périodique : les processeurs interrompent tous leur phase de recherche à intervalles réguliers pour réaliser un équilibre global de la charge [Lau 93, PFK 93].

4.3.4 Appariement

Quand un processeur a décidé d'initier un équilibrage de charge, il doit choisir un partenaire pour l'échange de travail. L'appariement peut se faire selon différentes politiques, les plus souvent utilisées étant :

- *Centralisée* : un processeur jouant le rôle de serveur gère un tableau de l'état des processeurs [HS 95, RK 87]. Un processeur désirant faire un échange de charge l'interroge pour trouver un partenaire. Cette technique est optimale en nombre de messages transitant par le réseau de communication car elle limite le nombre de tentatives infructueuses. Elle présente par contre l'inconvénient du goulot d'étranglement pour accéder au serveur.

- *Aléatoire* : le choix du partenaire se fait au hasard [FM 87, HS 95, San 95]. Cette technique est efficace en moyenne car l'interrogation se disperse rapidement sur l'ensemble des processeurs et évite ainsi les groupes de processeurs de même état.

- *En anneau* : le processeur parcours tous les processeurs selon un ordre déterministe jusqu'à trouver un partenaire [FM 87]. Elle est plus coûteuse en moyenne car linéaire en le nombre de processeurs. Finkel et Manber proposent deux variantes importantes selon le comportement des processeurs qui reçoivent une demande de travail alors qu'ils sont eux aussi en demande :

  - le processeur envoie un message négatif au demandeur qui se chargera alors de faire une demande à un autre processeur ;
  - le processeur transmet la demande à son suivant.

  Toutefois, la seconde solution est préférable en terme de messages transitant par le réseau de communication.

- *Topologique* : nous classons dans cette catégorie, tous les algorithmes qui réalisent les appariements selon la structure topologique de la machine cible. Le processeur désirant recevoir une charge, envoie une demande à ses voisins immédiats (communications en O(1)) qui lui renvoient alors une charge, ou un message négatif.

Sanders, dans [San 95] implémente un mécanisme d'appariement qui suit la structure d'un hypercube ou d'un *fat-tree*[14]. Pour leur part, Naganuma et Ogura suivent un voisinage en grille bidimensionnelle [NO 94].

Ce système d'appariement a l'avantage de n'utiliser que des communications rapides, mais a tendance à limiter la diffusion des charges vers les proches voisins, en particulier pour des architectures où les processeurs n'ont que peu de voisins.

4.3.5 Évaluation de la divisibilité d'une tâche

S'il est toujours possible, sauf cas extrême, de diviser une tâche de recherche pour un algorithme énumératif – envoi d'une partie des nœuds en attente – la division systématique n'est pas toujours souhaitable, en particulier si le processeur risque d'être lui-même à cours de travail peu de temps après. Naganuma et Ogura [NO 94] ont proposé de définir une fonction qui détermine si une tâche doit être considérée comme divisible. Ils utilisent pour cela la notion de différence de profondeur (dP dans la figure 4.11) entre le niveau divisible le plus haut et le niveau d'instanciation courant.

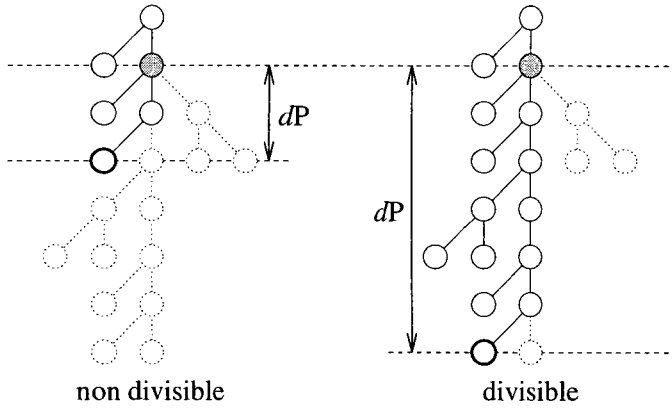


Figure 4.11 : Arbres de recherche non divisibles/divisibles. Le plus haut point divisible est grisé, alors que le point courant est en gras.

Si la différence dP est trop faible, le processeur refuse de diviser sa tâche courante. Dans [NO 94], la borne de divisibilité de dP est déterminée expérimentalement (par simulation de l'algorithme). Les auteurs ont ainsi montré qu'elle dépendait fortement du problème à résoudre. Ils ont nommé cette technique *LLB* pour *lazy load balancing*: équilibrage de charge paresseux.

La plupart des algorithmes de la littérature, plus anciens, considèrent qu'une tâche est divisible dès que plus de deux nœuds de recherche sont en attente.

4.3.6 Division de charge

Une fois qu'un processeur a décidé d'envoyer une partie de sa charge à un autre processeur, il lui faut la diviser. L'optimal pour la proportion est de $\frac{1}{2}$ pour obtenir un équilibre local entre les deux processeurs, dans le cas d'un appariement point-à-point. Rao et Kumar définissent trois types de division d'un arbre de recherche [RK 87]:

[14] le *fat-tree* est une structure arborescente. Elle est utilisée notamment dans le Connection Machine CM-5 [HT 93].

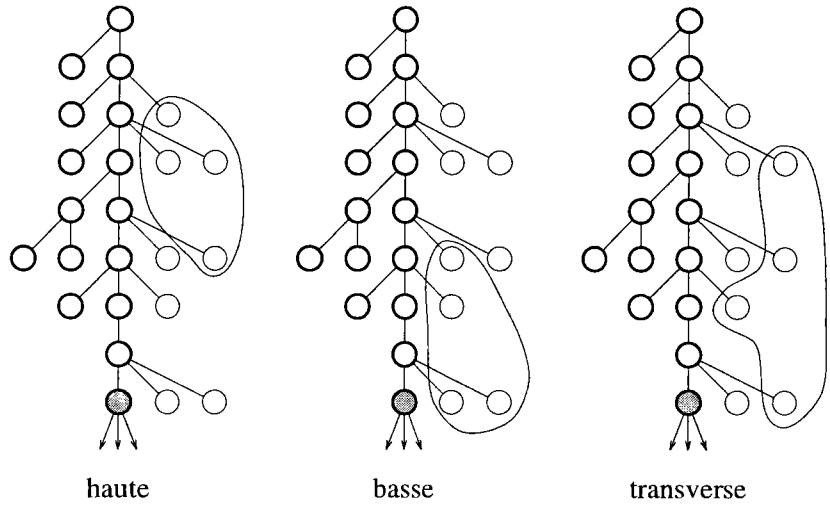


Figure 4.12 : Arbres de recherche et ensembles des nœuds envoyés pour les trois techniques de division. Les nœuds déjà explorés apparaissent en gras. Le nœud en cours d'instanciation est grisé.

- *Moitié haute* : le processeur fait le compte des nœuds qu'il n'a pas encore explorés (domaines restant dans les variables qu'il a instanciées) et donne les nœuds les plus hauts dans l'arbre. On obtient ainsi une politique dite « à gros grain » car les tâches transférées sont potentiellement de forte taille. Les auteurs ont montré que ce système est intéressant quand la machine dispose de peu de processeurs très puissants.

- *Moitié basse* : le processeur fait le même compte que précédemment mais envoie les nœuds les plus bas dans l'arbre. Cette technique appelée « à grain fin » transfère des tâches potentiellement de petite taille, qui conviennent mieux à une machine disposant d'un grand nombre de processeurs de faible puissance.

- *Coupe transversale* : le processeur donne la moitié des nœuds sur chaque niveau d'instanciation. Cette découpe se rapproche le plus de la moitié effective de la charge restante car elle est moins sensible au déséquilibre de l'arbre.

Dans [NO 94], seules les divisions en moitié haute et basse sont implémentées. Les autres auteurs s'intéressent essentiellement aux techniques d'appariement qu'ils détaillent dans leurs algorithmes, mais restent peu diserts quant aux politiques de division implémentées.

4.3.7 Transfert de charge

Le transfert effectif de charge entre deux processeurs obéit à une loi de compromis entre la puissance de calcul des processeurs et la vitesse de transmission des données entre les processeurs. Si ce rapport est élevé (processeurs puissants/communications lentes), il est généralement préférable de compresser les messages pour reporter le coût de la communication sur les éléments les plus rapides. À l'inverse, si ce rapport est faible (processeurs lents/communications rapides), il vaut mieux limiter la charge de calcul des processeurs en communicant les tâches de recherche par copie explicite.

Rao et Kumar citent le transfert par compression des charges dans leurs algorithmes, sans toutefois préciser les mécanismes effectivement mis en œuvre [RK 87]. Pour leur

part, Naganuma et Ogura décrivent les concepts de base de la technique de compression. Il appellent leur algorithme *RLB* pour *regenerating load balancing* [NO 94].

Conclusion

Les notions de base du parallélisme ont été abordées dans ce chapitre, afin de familiariser le lecteur – si besoin était – aux termes qui seront utilisés dans la suite de ce document. Les différents modèles de parallélisme ont été présentés pour souligner les différences entre eux. Toutefois, seul le modèle MIMD nous intéressera par la suite, puisque c'est celui sur lequel fonctionnent les algorithmes que nous étudierons.

Nous avons donné une vue synthétique de l'ensemble des problèmes rencontrés lors de la parallélisation d'algorithmes de résolution de CSP. Il est en fait apparu que la littérature dans ce domaine est particulièrement peu abondante, et qu'elle se concentre essentiellement sur les problèmes de filtrage par arc-consistance. Le principal résultat théorique sur l'arc-consistance porte sur sa complexité: AC est P-complet, c.-à-d. inhéremment séquentiel. Des algorithmes ont été développés pour de nombreux modèles de programmation: SIMD, MIMD et booléen. De plus il est apparu que les versions les plus élémentaires d'établissement de l'arc-consistance présentent les meilleurs résultats pratiques.

Les travaux de parallélisation des algorithmes de recherche sur les CSP étant peu abondants, nous avons fait « incursion » dans le domaine de la programmation afin d'avoir une approche constructive des problèmes liés à la parallélisation des algorithmes de recherche énumérative. Toutefois, seules les méthodes de recherche par consistance arrière telles que *backtrack* sont utilisées dans ce domaine, si bien que les problèmes plus spécifiques aux CSP – recherche par consistance avant, retour-arrière non chronologique, ... – ne sont pas abordés.

La voie de parallélisation naïve consistant à exécuter plusieurs recherches indépendantes en parallèle sur plusieurs processeurs à été proposée [HW 94] mais elle s'est avérée peu efficace. Le principal point qui ressort de cette étude est une dualité entre la recherche de type « ET-parallèle » qui induit un parallélisme fin, et la recherche de type « OU-parallèle » qui induit un parallélisme à gros grain. Ces deux écoles divisent profondément la communauté programmation logique, et en particulier les auteurs qui s'intéressent à la parallélisation de programmes PROLOG. Dans le cas du parallélisme ET, le parcours de l'arbre de recherche se fait en descendant simultanément dans les branches qui constituent une même clause. Il est alors nécessaire de synchroniser les résultats de chaque exploration. Le parallélisme OU définit pour sa part des exploration de branches totalement indépendantes.

La forte opposition entre ces deux principes de parallélisation de programmes logiques ne parait pas dominer le monde des CSP. En effet, si la recherche énumérative dans les deux formalismes induit un parcours d'arbre, sa construction est différente. Dans le cas de PROLOG, les branches partant d'un nœud sont construites par un mécanisme d'unification qui peut être récursif, alors que dans le cas de CSP, le travail sur chaque nœud est borné. Il apparaît donc que la recherche OU-parallèle sera toujours préférable pour les CSP. Les expérimentations dans ce domaine ont d'ailleurs montré que le parallélisme ET sur les CSP ne permet pas d'accélérations intéressantes avec un grand nombre de processeurs.

Chapitre 5

Algorithme générique pour la résolution parallèle des CSP

PLUSIEURS VOIES peuvent être suivies pour paralléliser la recherche de solution(s) pour les CSP. Comme on l'a vu au chapitre 4, il est possible de vérifier la consistance des instanciations successives en parallèle (orientation *ET-parallèle*), ou de diviser l'arbre de recherche pour parcourir chaque partie sur un processeur différent (orientation *OU-parallèle*). On pourra d'emblée éliminer de cette réflexion la réalisation de plusieurs recherches simultanées pour un même problème qui ne permet pas d'accélérations intéressantes.

Notre choix pour la parallélisation de la recherche énumérative de solutions pour les CSP se portera sur l'orientation « OU-parallèle ». En effet, elle est apparue comme plus performante dans le domaine de la logique, et le peu d'expérimentations réalisées en technique « ET-parallèle » sur des CSP ont confirmé la pertinence de notre choix [LHB 92].

En ce qui concerne le modèle de programmation, nous nous orienterons vers des algorithmes de type asynchrone. Les raisons qui motivent ce choix sont directement inspirées de l'orientation prise pour la parallélisation puisque les techniques OU-parallèles requièrent souvent des équilibrages par couples de processeurs qui sont plus adaptés au modèle de programmation asynchrone.

Nous présenterons, pour commencer ce chapitre, un cadre général de parallélisation de la recherche de solution(s) pour les CSP (paragraphe 1). Nous y détaillerons notamment l'organisation des données manipulées par les algorithmes de recherche, ainsi que la fonction générique de recherche. Nous aborderons ensuite les différentes techniques de recherche (classiques dans le cadre séquentiel) afin de déterminer leurs implications sur le parallélisme (paragraphe 2).

Le paragraphe 3 présentera, pour finir, différentes techniques parallèles applicables pour la résolution des CSP. Nous y détaillerons dans ce contexte les points importants qui surgissent lors de la parallélisation d'algorithmes de parcours d'arbre : la répartition initiale de la charge, et la division d'une tâche de recherche.

1 Contexte parallèle

Nous définissons dans ce paragraphe, une structure générale de recherche parallèle de solution(s) pour un CSP. L'idée directrice de l'algorithme est la généricité introduite dans [Pro 93b] pour la recherche séquentielle. Nous étendons en effet ce concept à la présentation d'un algorithme parallèle qui permettra d'hybrider, non seulement toutes les techniques de recherche séquentielles, mais aussi les techniques strictement parallèles.

Le paragraphe 1.1 présentera la fonction de résolution parallèle générique, et évoquera les comportements attendus des fonctions spécifiques. Le paragraphe suivant (1.2) énumérera de façon exhaustive toutes les données globales manipulées par les différents algorithmes.

1.1 Algorithme générique de recherche

La résolution d'un problème en parallèle se fait par l'intermédiaire d'une fonction générique exécutée par tous les processeurs (algorithme 5.1). Conceptuellement, cette fonction est constituée de deux boucles principales imbriquées : la boucle externe (lignes 5–24) regroupe l'activité du processeur. Elle alterne une phase de travail (boucle interne) et une phase de recherche de travail (ligne 23). Nous présentons la fonction de résolution de façon détaillée pour permettre au lecteur de positionner avec précision les différentes fonctionnalités que nous allons aborder par la suite.

Algorithme 5.1 *Procédure principale de recherche de solution(s) en parallèle.*

```
1  proc Résolution_Parallèle (i) ≡
2    ⟨libre, nœud⟩ ⟵ Répartir_Charge (i)
3    terminé ⟵ faux
4    consistant ⟵ vrai
5    tant que ¬terminé faire
6            tant que ¬libre faire
7                    si consistant alors ⟨nœud, consistant⟩ ⟵ Descendre (nœud)
8                    sinon ⟨nœud, consistant⟩ ⟵ Remonter (nœud) finsi
9                    si nœud > n alors
10                     terminé ⟵ vrai
11                     État ⟵ solution
12                     Message_Général (terminaison)
13                   sinon si nœud = 0 alors
14                     libre ⟵ vrai
15                     si Terminaison_Globale (i) alors
16                       terminé ⟵ vrai
17                       État ⟵ impossible
18                       Message_Général (terminaison)
19                     finsi
20                   finsi
21                   ⟨libre, terminé⟩ ⟵ Traiter_Un_Message (i, libre, terminé)
22           fintant que
23           ⟨libre, terminé⟩ ⟵ Équilibrage (i, libre, terminé)
24   fintant que.
```

La notation « ⟨var1, var2⟩ ⟵ *Fct* » signifie que l'appel de la fonction *Fct* modifie les variables var1 et var2 de la procédure. Le paramètre i est le numéro du processeur sur lequel est exécuté le programme. On considère que $0 \leq i \leq N-1$, si N est le nombre de processeurs utilisés. La procédure *Message_ Général* envoie un message donné à tous les processeurs. Elle ne change pas en fonction du choix de la méthode de résolution, mais en fonction de la topologie de la machine parallèle. Les autres fonctions implémentent les différentes méthodes de parallélisation et sont toutes spécifiques à un algorithme donné.

La procédure est constituée d'une boucle principale (lignes 5–24) qui ne se termine que lorsque le processeur a détecté la terminaison globale de la recherche. À l'intérieur de cette boucle, l'activité du processeur (oisif ou occupé) détermine l'entrée dans la boucle de travail (lignes 6–22). Au sortir de la boucle, le processeur est oisif et une phase d'équilibrage est enclenchée (ligne 23). La recherche séquentielle suit le mécanisme de descente et remontée dans l'arbre, en fonction de l'état de consistance courant (lignes 7–8). Si une solution est trouvée (ligne 9), un message est envoyé à tous les processeurs (ligne 12). Quand le processeur termine sa recherche locale sans avoir trouvé de solution (ligne 13), une synchronisation globale est requise pour déterminer si la recherche est globalement terminée (ligne 15).

Dans les paragraphes qui suivent, nous présentons les fonctionnalités des différentes fonctions génériques d'implémentation du contrôle du parallélisme.

Fonction *Répartir_ Charge*: réalise la répartition initiale de la charge. Si l'algorithme de recherche n'implémente pas de répartition de charge, seul le processeur P_0 prendra toute la charge de résolution du problème, les autres restant inactifs.

Dans le cas d'une répartition de charge centralisée, le processeur P_0 distribue la charge initiale de résolution à tous les processeurs P_k, $\forall 1 \leq k \leq N-1$. On peut décider si le processeur P_0 garde une partie de la charge de travail pour lui afin de participer à la résolution en tant que chercheur, ou s'il distribue tout le travail et joue le rôle de serveur d'équilibrage.

Dans le cas d'une répartition de charge distribuée, chaque processeur établit sa propre charge de travail en fonction du problème à résoudre et de son numéro. La fonction doit préserver :

- la complétude de la recherche afin que tout le travail de recherche soit distribué au départ ;
- l'unicité d'affectation de la charge de travail sur les processeurs afin de ne pas entraîner de surcharge de travail.

Si une charge de travail a été affectée au processeur P_i, la fonction met **libre** à **faux** et initialise la variable **nœud** à un nœud de recherche. Dans le cas où aucune charge de travail n'a été affectée au processeur P_i, les variables **libre** et **nœud** sont initialisées à (respectivement) **vrai** et 0.

Fonction *Descendre*: cherche une instanciation consistante pour la variable **nœud**. Elle est propre à chaque méthode séquentielle de recherche. Elle met **consistant** à **vrai** si elle a pu instancier la variable **nœud**, et à **faux** sinon. De plus, elle met à jour la valeur de **nœud** pour pointer sur la prochaine valeur à instancier. L'heuristique d'ordonnancement des

instanciations est définie à l'intérieur de cette fonction. On notera que son implémentation parallèle est identique à la version séquentielle.

Fonction *Remonter*: annule l'instanciation d'une ou plusieurs variables dans le sous-arbre de recherche, en fonction de la méthode séquentielle de retour-arrière choisie. Elle positionne **nœud** sur la prochaine variable à instancier, et **consistant** pour indiquer si son domaine est non vide. Dans le cas où la remontée est impossible sur le sous-arbre en cours d'exploration (celui-ci est alors totalement exploré), la fonction modifie aussi les autres variables de recherche de solution (**haut**, **domaine_courant**[k] et **v**[k]) pour enclencher éventuellement la recherche sur un autre sous-arbre.

Dans le cas où le processeur a terminé l'exploration de tous ses sous-arbres locaux, la fonction met **nœud** à 0.

Fonction *Terminaison_Globale*: indique si tous les processeurs ont fini leur recherche, auquel cas, le programme parallèle s'arrête sur la non-satisfiabilité du problème. Elle constitue le point de synchronisation le plus délicat entre les processeurs puisqu'elle nécessite une mise en commun d'une information globale – alors que les autres synchronisations se font par couples de processeurs.

Fonction *Traiter_Un_Message*: se charge de traiter tous les messages reçus par un processeur. Ces messages peuvent être de différents types :

- *Demande de travail*: le récepteur est amené à diviser sa charge de travail, si elle est divisible, selon une procédure à définir. Dans tous les cas, les variables **libre** et **terminé** ne sont pas modifiées par ce message, puisque le récepteur ne donne pas toute sa charge de travail (induisant une modification de **libre**), ni ne décide que la recherche globale est terminée (modification de **terminé**).

- *Charge*: le message contient alors une tâche de recherche que le récepteur ajoute à sa liste de sous-arbres de recherche locaux. Ce message peut induire une modification de la variable **libre** dans le cas où le récepteur reçoit du travail alors qu'il n'en avait pas. Par contre, il n'induit pas de modification de **terminé**.

- *Indication d'impossibilité*: ce message indique que son expéditeur ne peut envoyer de travail au récepteur. Il peut donc entraîner la réactivation de la procédure d'équilibrage, mais ne modifie pas les variables **libre** et **terminé**.

- *Terminaison*: ce message indique que le travail peut être arrêté car un autre processeur a terminé la recherche. Le récepteur met alors les variables **libre** et **terminé** à **vrai**, ce qui induira la fin de la procédure *Résolution_Parallèle*.

D'autres messages peuvent être utiles pour des algorithmes précis, mais ne sont pas présentés ici.

Fonction *Équilibrage*: envoie une *demande de travail* à un processeur selon la technique d'appariement définie par l'algorithme et attend les messages en réponse qu'elle traitera par l'appel de la fonction *Traiter_Un_Message*. Hormis la fonction qui détermine le type d'appariement utilisé, la fonction *Équilibrage* sera toujours implémentée de la même manière.

1.2 Structures de données

On classera les structures de données utilisées lors de la résolution d'un CSP en parallèle selon deux catégories : les données génériques qui seront nécessaires quel que soit l'algorithme de recherche ou la méthode parallèle implémentée, et les données spécifiques à l'un ou l'autre des hybrides. D'un point de vue algorithmique, les structures de données communes correspondent à celles qui sont utilisées pour la mise en œuvre de l'algorithme *generate-and-test*[1]. Les données spécifiques à un algorithme de recherche ne seront pas évoquées ici, mais seront introduites dans les différentes parties du paragraphe 2.

Dans ce paragraphe, nous assimilons une tâche dans la recherche de solution pour un CSP, à un processus dans un système distribué. Nous faisons notamment la correspondance entre les parties logiques et physiques d'une machine parallèle (processeur, code, etc.), et les structures de données manipulées par les algorithmes de recherche. L'idée qui sera sous-jacente dans les différentes parties de ce paragraphe concerne la définition d'une *tâche de recherche.*

Définition 5.1 *Une* **tâche de recherche** *dans un algorithme de recherche de solution d'un* CSP *correspond au parcours d'un sous-arbre d'exploration.*

Du fait qu'une tâche de recherche est un processus dynamique, elle est représentée par un ensemble de données définissant une vue spatiale (ensemble des données manipulées par la tâche de recherche), et une vue temporelle (ensemble des données exprimant l'état du processus de recherche). À titre d'exemple, la figure 4.8 définit quatre tâches de recherche, numérotées de 1 à 4.

1.2.1 Processeur de recherche

Le processeur de recherche est implémenté par un algorithme particulier de parcours de l'arbre de recherche (*FC-CBJ* par exemple). Sa représentation se fait par l'ensemble des algorithmes (fonctions de *descente*, *remontée*, *équilibrage*, ...) et par les données permettant de simuler son fonctionnement :

- **libre** : un booléen qui indique si le processeur est en phase de recherche (valeur **faux**) ou s'il est oisif (valeur **vrai**).
- **terminé** : un booléen qui indique si l'algorithme de recherche est terminé.

D'autres structures de données peuvent être utilisées pour un algorithme de recherche particulier. Ces deux variables sont purement locales à un processeur puisqu'elles le caractérisent lui-seul, et ne seront modifiées que par lui.

[1] L'algorithme *generate-and-test* – présenté au paragraphe 6 du chapitre 1– est à la base des algorithmes de recherche énumératifs. Tous les autres algorithmes ne font qu'y apporter des modifications pour améliorer son efficacité, en y ajoutant de nouvelles structures de données.

1.2.2 Code d'exécution

Le code d'exécution des processeurs de recherche est représenté par l'ensemble des variables et des contraintes du problème à résoudre. Les variables sont numérotées de 1 à n. On dispose de plus d'un tableau de listes de valeurs domaine_initial[k] ($1 \leq k \leq n$) qui représente les domaines de chaque variable, selon la définition globale du problème.

Les m contraintes du problème sont stockées de manière « transparente » pour les algorithmes de recherche, et donc non définie ici. Seul un tableau cnt[k], ($1 \leq k \leq n$) de listes d'entiers donne pour chaque variable, la liste des contraintes qui portent sur elle. L'utilisation explicite – pour vérification – des contraintes se fera par l'appel de la fonction *Vérifier* qui prendra simplement comme paramètre le numéro de la contrainte.

Nous avons choisi l'analogie entre ces données et le code d'exécution d'un processeur parallèle car elles ne sont pas modifiées au cours de la recherche. L'implémentation des algorithmes de recherche suppose que l'ensemble du code – c.-à-d. la représentation du problème – soit accessible par tous les processeurs. Dans le cas où une duplication des ces structures de données n'est pas possible sur tous les processeurs, un mécanisme de mémoire virtuelle partagée (en lecture seule) doit être mis en œuvre. Son utilisation sera alors transparente pour les algorithmes dont nous discutons de l'implémentation.

1.2.3 Données d'une tâche de recherche

On retrouve ici les parties de la définition d'un problème qui sont distribuées sur les différents processeurs de recherche. Elle sont séparées du « code » car elles sont susceptibles d'être modifiées au cours de la recherche. Les structures de données sont :

- domaine[k], ($1 \leq k \leq n$) : un tableau de listes de valeurs. Chaque liste correspond au domaine initial de chaque variable. Il est important de noter que par *domaine initial*, on entend ici le domaine de chaque variable pour la tâche concernée (sous-arbre), et n'est donc pas identique sur chaque processeur. Le tableau domaine[k] sera une projection du tableau domaine_initial[k].

- domaine_courant[k], ($1 \leq k \leq n$) : un tableau de listes de valeurs. Chaque liste correspond au domaine courant de chaque variable, c'est-à-dire à toutes les valeurs qu'il reste à essayer dans l'exploration du sous-arbre de la tâche de recherche pour chaque variable.

Le tableau domaine_courant[k] n'est jamais transmis car il représente les modifications successives des domaines lors de l'exploration de l'arbre de recherche. Il est obtenu par une copie intégrale du tableau domaine[k] au début de l'exécution de la tâche de recherche.

1.2.4 Pile d'exécution d'une tâche de recherche

Le parcours d'un sous-arbre de recherche se fait en profondeur d'abord, et utilise donc une pile. Chaque élément empilé correspond à l'instanciation d'une variable. Les données représentant cette pile sont :

- haut : un entier qui pointe sur la plus basse variable qui n'est pas localement instanciée dans la pile.

– v[k], ($1 \leq k \leq n$) : pile d'instanciation. La valeur v[k] correspond à l'instanciation courante de la variable X_k.

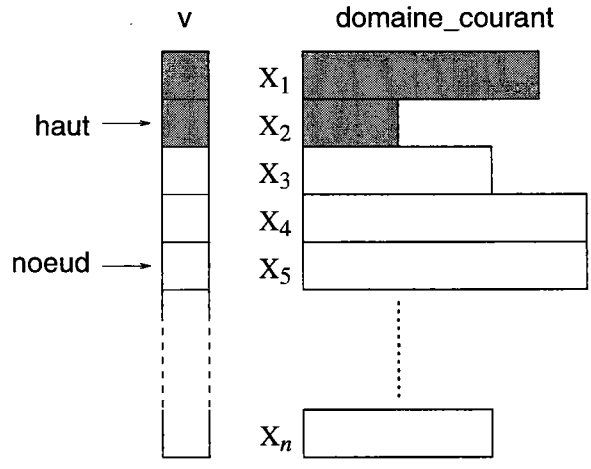


Figure 5.1 : Schéma des principales données représentant une tâche de reherche. Les parties grisées correspondent aux données qui ne sont pas modifiables par la tâche en cours.

Un sous-arbre d'exploration est toujours enraciné sur la racine de l'arbre de recherche global. Il est ainsi constitué d'un « tronc » retraçant le chemin d'instanciation qui mène à la partie de l'arbre global à explorer localement. Les valeurs d'instanciation sont empilées dans v[k], mais seules celles dont l'indice est supérieur ou égal à **haut** peuvent être modifiées par la tâche de recherche.

1.2.5 Registres du processeur de recherche

Les « registres » utilisés par chaque processeur de recherche permettent de caractériser l'état courant de l'exploration du sous-arbre. Ils sont formés des variables suivantes :

– **nœud** : un entier qui correspond au niveau courant de la recherche. Cette variable cumule les fonctions de compteur ordinal et de sommet de pile, puisque chaque « instruction » conduit à l'empilement d'une valeur.

– **consistant** : un booléen qui correspond à l'état courant de la recherche. Il indique notamment la consistance de la pile de recherche.

1.2.6 Tâches de recherche en attente

Nous considérons que le système que nous implémentons est multi-tâches en traitement par lot : chaque processeur de recherche dispose d'une liste de tâches de recherche indépendantes à exécuter. Ces tâches sont regroupées dans une variable **charge** qui correspond à une liste de triplets de la forme ⟨**haut**, v[1..n], **domaine**[1..n]⟩. Chaque triplet permet d'initialiser toutes les données relatives au sous-arbre de recherche qu'il représente.

2 Parcours de l'arbre de recherche

La parallélisation d'un algorithme séquentiel pose toujours un certain nombre de problèmes. Si l'algorithme de parcours d'arbre de recherche semble naturellement parallèle dans la mesure où il définit des tâches indépendantes, il pose néanmoins quelques difficultés. Nous passons en revue dans ce paragraphe, les caractéristiques essentielles de la recherche séquentielle, afin d'en souligner les implications sur le plan du parallélisme.

2.1 Consistance au cours de la recherche

Du point de vue du parallélisme strict, l'établissement d'un certain type de consistance au cours de la recherche n'a aucune implication puisqu'il ne remet nullement en cause l'indépendance des sous-arbres de recherche. Toutefois, les algorithmes de recherche par consistance avant, tels que *FC*, accumulent un nombre non négligeable de traitements et de données lors de la descente dans l'arbre.

En effet, avec *BT* – qui constitue la version « CSP » des algorithmes de recherche utilisés en programmation logique – les seules données accumulées lors de la descente dans l'arbre, sont les valeurs d'instanciation déjà effectuées. Les environnements de recherche[2] des différentes sous-tâches sont tous identiques puisque l'algorithme n'a qu'un regard en arrière du problème.

De leur coté, les algorithmes tels que *FC* modifient les domaines des variables non instanciées en fonction du chemin d'instanciation[3] qu'ils empruntent. Ce comportement conduit à créer un environnement spécifique pour chaque sous-tâche de recherche.

Note 5.2 *L'environnement de recherche d'une tâche dans un algorithme de recherche par consistance en avant est dépendant de son chemin d'instanciation.*

Dans ce type d'algorithme, la transmission d'une tâche de recherche lors d'une phase d'équilibrage de charge nécessitera la communication, non seulement du chemin d'instanciation, mais aussi de l'environnement de recherche de la tâche considérée. Ceci aura donc des conséquences significatives sur les politiques d'équilibrage et de division de charge, car le changement de contexte de recherche (passage d'une sous-tâche à une autre) sera coûteux.

2.2 Retour-arrière

Les phénomènes de retour-arrière sont susceptibles de poser des problèmes encore plus épineux quant à la parallélisation de la recherche. Si les retours-arrière engendrés par un algorithme de type *backtrack* chronologique, comme *BT* ou *FC*, n'ont aucune conséquence sur le parallélisme, ceux engendrés par un algorithme tel que *BJ*, peuvent se classer en deux catégories (figure 5.2) :

– les sauts *internes* à une tâche qui ne concernent qu'elle ;

[2]on appelle ici *environnement de recherche*, l'ensemble des données qui détermineront la « forme » du sous-arbre de recherche qui sera exploré. Dans notre contexte, cet environnement est constitué du tableau domaine (*cf.* paragraphe 1.2).

[3]le *chemin d'instanciation* correspond aux instanciations successives qui mènent à un sous-arbre de recherche.

– les sauts *hors contexte* qui supposent un transfert de contrôle de la tâche qui rencontre le blocage, vers une tâche différente[4].

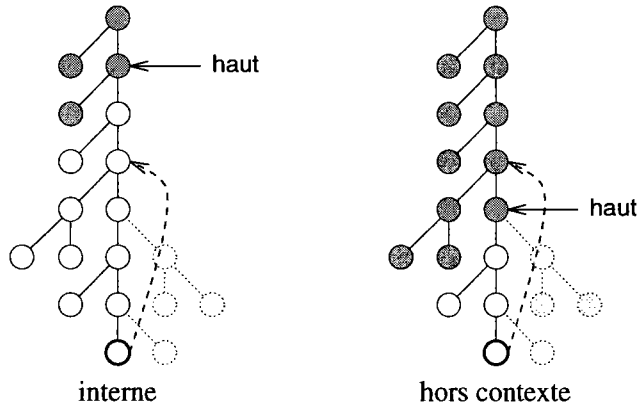


Figure 5.2 : Exemples de sauts interne et hors contexte. Les instanciations qui ne dépendent pas de la tâches sont grisées et les sauts apparaissent en tirets gras.

Dans le cas d'un saut hors contexte, la tâche en cours est naturellement terminée puisqu'aucune instanciation ne peut plus être faite dans son sous-arbre de recherche. Cependant, sa prise en compte globale suppose une communication explicite entre la tâche courante, et celle(s) qui pourrai(en)t bénéficier de cette information. La quantité d'information à transmettre est minime puisqu'il suffit d'échanger le chemin d'instanciation qui est apparu comme erroné. Par contre, la détermination du (ou des) processeur(s) concerné(s) nécessite un mécanisme spécifique.

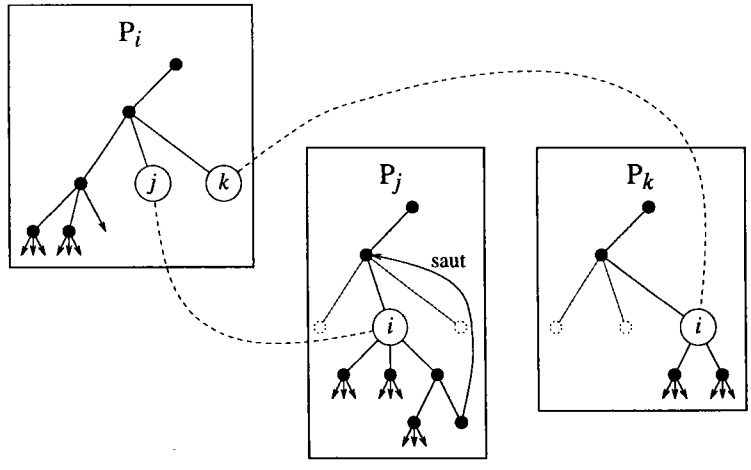


Figure 5.3 : Exemples de scénario de propagation d'un saut hors contexte.

Nous proposons un mécanisme simple qui permet de suivre les chemins d'instanciation sur plusieurs processeurs afin de propager efficacement les sauts hors contexte (illustré par la figure 5.3) : lorsqu'un processeur P_i divise son sous-arbre de recherche, il mémorise le numéro du processeur destinataire à la racine du sous-arbre envoyé. Le processeur qui reçoit une nouvelle sous-tâche note le numéro de son expéditeur. Quand un saut hors

[4]la détermination de la tâche destinataire d'un saut hors contexte est discutée plus loin.

contexte est détecté par un processeur P_j, celui-ci transmet l'information à l'expéditeur P_i de la tâche concernée qui sera à même d'en tenir compte et de propager le message aux autres processeurs.

On notera que la réception par un processeur d'un saut hors contexte peut lui même provoquer un nouveau saut hors contexte si le point de retour n'est pas dans l'espace de travail du récepteur. En effet, le mécanisme proposé ne permet pas de connaître directement le plus haut « père » d'un sous-arbre. Un seul saut peut ainsi provoquer une cascade de messages pour informer tous les processeurs concernés.

Si la propagation des sauts hors contexte présente un intérêt théorique évident pour l'optimisation du parcours parallèle de l'arbre de recherche, elle peut toutefois s'avérer plus coûteuse que bénéfique. En effet, une inconsistance entre deux nœuds distants sur un chemin d'instanciation aura de nombreuses chances d'être détectée par plusieurs processeurs partageant la portion de chemin concernée (par exemple l'expéditeur et le récepteur d'une tâche). Dans ce cas, la notification du saut conduira à une ou plusieurs communications inutiles.

Certes, de nombreux éléments plaident en défaveur des sauts hors contexte, mais il convient toutefois de rappeler certains faits : tout d'abord, la communication d'un saut n'est pas très coûteuse puisqu'elle ne contient que le chemin d'instanciation incorrect. De plus, si une tâche provoque un saut hors contexte, cela signifie qu'elle se termine et qu'il se peut alors qu'une phase d'équilibrage de charge soit nécessaire. Il peut être intéressant de combiner dans un même message le signalement de l'inconsistance et la demande de travail. Ainsi, le message n'aura peut-être pas d'intérêt pour le retour-arrière mais ne viendra pas surcharger les mécanismes de communication de la machine. Une expérimentation séquentielle (chapitre 6) viendra évaluer l'intérêt de la prise en compte des sauts hors contexte.

2.3 Mémorisation

D'un point de vue général, la mémorisation consiste à jumeler la recherche de solution avec un algorithme d'établissement de consistance locale. Quand le mécanisme de recherche rencontre une inconsistance, celle-ci est mémorisée afin de ne pas commettre plusieurs fois la même erreur d'instanciation. La difficulté pour le parallélisme vient du fait que la portée des informations découvertes au fil de la recherche est globale à l'ensemble de l'arbre de recherche. Une transmission systématique de toutes les découvertes à l'ensemble des processeurs ne paraît pas réaliste, d'autant plus que de nombreuses inconsistances seront découvertes simultanément par plusieurs processeurs, engendrant une redondance dans la diffusion d'informations.

Note 5.3 *Une inconsistance est découverte en divers points précis de l'arbre de recherche, mais sa portée est globale.*

La solution qui semble la plus raisonnable pour l'implémentation des techniques de mémorisation en parallèle consiste à appliquer en local toutes les informations apprises au cours de la recherche. Ainsi, chaque processeur se construit sa propre base de connaissances sans rien partager avec ses voisins. On considère ici que les autres processeurs auront de fortes chances de faire les mêmes constatations.

Il est clair que ce cloisonnement perd une partie de l'avantage que procure la mémorisation sur la recherche globale de solution(s), mais il permet d'éviter une explosion anarchique du nombre de communications. On pourra quelque peu transgresser sur cette loi en autorisant les communications qui semblent stratégiques, mais la difficulté réside alors dans l'évaluation *a priori* du compromis entre surcoût dû à la communication, et bénéfice dû à la diffusion de l'information.

2.4 Marquage

Qu'il soit négatif (*BC*) ou positif (*BM*), le marquage nécessite des structures de données qui accumulent des informations supplémentaires le long de la descente dans l'arbre de recherche. Ces informations sont toujours locales à une branche d'instanciation et ne génèrent donc pas de communications entre des branches différentes.

Mais de même que pour la recherche par consistance en avant, les structures de données accumulées par les algorithmes de marquage constituent un *environnement de recherche* qui varie d'une branche à l'autre. Elles induiront donc une surcharge de communication au moment de la transmission d'une tâche. Toutefois, les informations sont moins stratégiques que pour *FC* puisqu'elles ne servent qu'à essayer de gagner du temps lors des vérifications de consistance[5]. Si une partie des marques est perdue au cours de la transmission, elle sera simplement reconstruites, moyennant un léger surcoût de calcul. Un juste milieu devra alors être trouvé entre la limitation des coûts de communication qui réduit d'autant l'apport global de *BM*, et l'efficacité espérée qu'apporte ces techniques.

2.5 Retardement

Le retardement de la vérification est naturellement parallélisable puisqu'il ne nécessite aucune communication supplémentaire. De plus, il n'exige pas de structures de données spécifiques car il constitue seulement une modification de l'algorithme de vérification. En fait, son utilisation parallèle avec *FC* est souhaitable, non seulement pour la limitation séquentielle du coût de vérification des contraintes (avantage temporel), mais aussi pour la réduction de la taille des messages lors de l'équilibrage de charge puisqu'il induit moins de filtrage sur les domaines futurs (avantage spatial).

Il faut noter que le retardement de *FC* dans le cadre séquentiel, est généralement combiné avec des mécanismes de *backmarking* dans un soucis d'efficacité. Il est nécessaire de dissocier les deux techniques dans le cadre parallèle (qui gardent une certaine indépendance), car on peut être amené à garder *Mxx* pour son habileté à limiter les coûts globaux sans surcoût parallèle, sans toutefois l'associer directement à *BM* pour éviter son coût en communications.

2.6 Ordres dans la recherche

L'ordre d'instanciation des variables n'implique pas de surcoût en nombre de communications dans un cadre parallèle. Le principal problème réside dans la transmission de tâches lors des phases d'équilibrage: il est alors indispensable de transmettre le chemin

[5]dans *FC*, la transmission de l'environnement est indispensable car l'algorithme en est totalement dépendant.

d'instanciation qui mène au sous-arbre à explorer. Dans le cas où tous les processeurs disposent du moyen de connaître localement l'ordre vertical, la communication d'une tâche se fait en envoyant la liste des valeurs[6], dans l'ordre d'instanciation. La connaissance de l'ordre peut se faire par un ordre explicite (liste triée), ou par une fonction de calcul indépendante du processeur ou de la branche considérée (ordre statique). Si l'ordre d'instanciation des variables est dynamique, il faut alors, en plus de la liste des valeurs, transmettre la liste des variables auxquelles correspondent les valeurs. On a ainsi un doublement de la taille des messages qui peut être gênant dans certaines architectures.

Remarque : Les heuristiques d'ordre sur les valeurs (ordre horizontal) ou sur les contraintes (ordre vertical) n'ont pas d'influence sur la conception d'algorithmes de recherche parallèles. ◇

Dans le cadre de la recherche séquentielle, les heuristiques d'ordonnancement des variables (en particulier *MRV*) visent à minimiser le facteur de branchement des nœuds afin de limiter la largeur de l'arbre de recherche. L'objectif d'un tel choix est de minimiser la longueur séquentielle du parcours de l'arbre. Dans un contexte parallèle, les objectifs sont quelque peu différents, puisqu'on a plutôt intérêt à établir un arbre de recherche large, pour maximiser le parallélisme potentiel. Ainsi, deux techniques opposées d'ordre vertical doivent cohabiter dans un même algorithme parallèle, avec un basculement de l'une à l'autre selon que la méthode répartit des tâches sur les processeurs (maximisation de la largeur de l'arbre), ou qu'elle cherche séquentiellement une solution dans un sous-arbre (minimisation de la largeur de l'arbre).

En se basant sur le fait que la recherche est non-déterministe, il peut être intéressant d'utiliser des heuristiques de choix d'ordre des variables localement différentes sur chaque processeur. En combinant dans une même recherche des progressions différentes, notamment *first-fail* et *last-fail*, on augmente les chances de trouver plus rapidement une solution lorsque la densité de solution est forte (par les descentes *last-fail*) tout en gardant la possibilité de finir rapidement quand le CSP n'a pas de solution (techniques *first-fail*). La répartition des heuristiques, et surtout leurs proportions, peut être établie au départ en fonction de l'espérance de satisfiabilité du problème en appliquant des calculs hérités de la recherche sur les phénomènes de transition de phase.

3 Techniques parallèles

Les algorithmes parallèles de recherche énumérative sont inhéremment déséquilibrés, et la majeure partie du travail de parallélisation porte sur l'équilibrage du travail des processeurs. Dans ce paragraphe, nous détaillons toutes les techniques que nous mettons en œuvre pour la résolution des CSP. Certaines d'entre elles sont très générales et n'apportent pas de nouveauté dans ce domaine, comme par exemple les techniques d'appariement. Pour celles-ci, nous nous contenterons de présenter leur mise en œuvre dans le cadre général de résolution défini au paragraphe 1. Pour d'autres concepts comme la répartition ou la division de charge, des idées originales seront mises en œuvre. En effet, la spécificité de la structure d'un CSP permet de définir de nouvelles techniques pour optimiser le calcul parallèle.

[6] la liste peut être transmise de manière explicite, ou par des données compactes qui permettent de recalculer la séquence des valeurs.

3.1 Initiation de l'équilibrage

Il paraît difficile de définir une nouvelle technique d'initiation de l'équilibrage de charge qui sorte du contexte des seuils de surcharge et de décharge. La difficulté pour l'application à un problème spécifique – les CSP pour ce qui nous concerne – consiste à établir les fonctions de calcul de ces seuils. Or, comme on l'a vu au chapitre 4 (paragraphe 4.3), il est impossible de définir un seuil de surcharge fiable dans des problèmes aussi irréguliers que les CSP. Le seuil de décharge, quant à lui, est généralement fixé à 0 (plus aucun travail à effectuer) afin que seuls les processeurs inactifs entrent en phase d'équilibrage de charge.

On pourra définir un seuil de décharge non nul, afin d'anticiper le manque de travail d'un processeur, mais la difficulté sera la même que pour le seuil de surcharge, et on risque des situations aberrantes dans lesquelles un processeur demande du travail supplémentaire alors qu'il a déjà la plus grosse partie de la charge. Dans tous les algorithmes que nous avons implémentés, le seuil de décharge est fixé à 0 (inactivité des processeurs).

Algorithme 5.2 *Fonction d'équilibrage de charge.*

```
1 proc Équilibrage (i, libre, terminé) ↦ ⟨booléen, booléen⟩ ≡
2   tant que libre ∧ ¬terminé faire
3           envoyer demande à Appariement (i)
4           ⟨libre, terminé⟩ ⟵ Traiter_Un_Message (i, libre, terminé) fintant que
5   retourner ⟨libre, terminé⟩.
```

L'algorithme 5.2 présente la fonction d'équilibrage de charge appelée par la fonction *Résolution_Parallèle* (*cf.* algorithme 5.1). Elle garde le contrôle de l'exécution tant que le processeur est libre, ou que la recherche globale n'est pas terminée (ligne 2). Une demande de travail est envoyée à un processeur distant selon une technique d'appariement générique, puis la réponse vient modifier les valeurs d'état du processeur.

3.2 Appariement

De même que pour l'initiation de l'équilibrage de charge, les techniques d'appariement n'apporteront pas de nouveauté. Nous avons toutefois implémenté et expérimenté les principales méthodes rencontrées dans la littérature :

Centralisé : Un serveur se charge de transmettre les demandes aux processeurs qui sont référencés comme ayant du travail. Le protocole d'équilibrage suit le schéma de la figure 5.4 : le processeur oisif envoie une demande au serveur qui la transmet à un autre processeur. Celui-ci envoie alors une sous-tâche au demandeur (si possible) ou une notification d'impossibilité.

Si le demandeur reçoit une nouvelle charge, il signale au serveur qu'il a de nouveau du travail pour que celui-ci puisse le remettre dans la liste des processeurs travailleurs.

Le serveur peut être un processeur dédié qui ne se chargera que de l'appariement. Mais on peut aussi confier cette tâche à un processeur chercheur quelconque. Ce choix ne sera pas totalement anodin, puisqu'un serveur spécialisé diminuera l'efficacité du parallélisme. En effet, la charge totale de recherche sera répartie, non pas sur N, mais sur $N - 1$ processeurs.

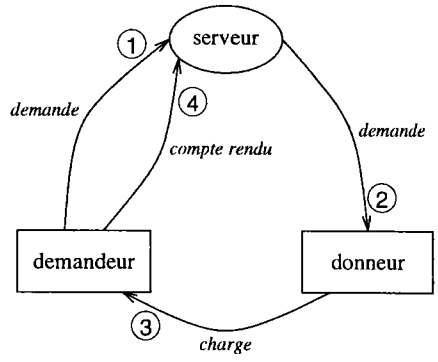


Figure 5.4 : Échanges de messages pour l'appariement centralisé.

Si le nombre de processeurs est important, il deviendra préférable d'utiliser un serveur spécialisé afin que les demandes soient traitées rapidement. À l'opposé, si le nombre de processeurs disponibles est faible, il sera certainement préférable de confier la fonction d'appariement à un chercheur pour ne pas grever les performances globales.

Aléatoire : C'est la technique la plus simple à implémenter : le demandeur envoie une requête à un processeur choisi au hasard qui lui enverra une charge s'il peut diviser la sienne. Dans le cas contraire, le processeur interrogé notifie l'impossibilité de division, et le processus d'interrogation recommence.

En début d'algorithme, cette technique sera efficace, puisque l'espérance qu'un processeur soit actif est alors importante. Elle le sera moins en fin de recherche (si le problème n'est pas satisfiable) car la charge restante est répartie sur un petit nombre de processeurs.

Circulaire : Le demandeur envoie sa requête au processeur suivant dans l'ordre de leur numérotation. Si celui-ci peut diviser sa charge, il en renvoie une partie au demandeur. Dans le cas contraire, il fait suivre la demande à son suivant. Si le demandeur reçoit sa propre demande de son prédécesseur, cela signifie qu'aucun processeur ne peut plus diviser sa charge, mais cela ne signifie pas forcément que la recherche globale est terminée (*cf.* paragraphe 3.7).

En étoile : C'est une méthode hybride entre l'appariement aléatoire et circulaire : le processeur oisif demande une charge à son suivant immédiat. Si celui-ci ne peut diviser sa charge, il répond par un acquittement négatif, et le demandeur réitère sa requête aux autres processeurs en suivant l'ordre circulaire.

3.3 Répartition de charge

Le but de la répartition de charge pour la recherche de solutions dans un CSP est double :

- affecter du travail à tous les processeurs en même temps afin que ceux-ci ne se trouvent pas en état de carence dès le début effectif de la recherche parallèle de solution ;

– éviter dans la mesure du possible, ou pour le moins retarder au maximum, le moment où la charge de travail sera globalement déséquilibrée.

La principale motivation de ces objectifs est que le programme parallèle doit s'exécuter en maximisant l'occupation des processeurs, pour obtenir une bonne accélération, et minimiser les communications entre processeurs qui sont toujours coûteuses, limitant le surcoût du parallélisme en début d'algorithme.

Pour notre étude expérimentale, nous choisissons une répartition de charge distribuée, sans aucune communication ; ainsi chaque processeur peut s'attribuer une charge de travail sans concertation explicite avec les autres. De plus, la charge initiale ne portera pas sur une seule branche de recherche par processeur, mais sur plusieurs branches indépendantes. Cette caractéristique est motivée par le fait qu'une branche peut rapidement mener à un échec lors du mécanisme de recherche puisque l'arbre n'est, en général, pas équilibré. Ainsi, chaque processeur disposera d'une série de branches de recherche, réduisant d'autant les risques de se retrouver à court de travail prématurément. La répartition des branches attribuées à un même processeur dans l'arbre de recherche est uniforme sur toute la largeur de l'arbre afin de « répartir » le déséquilibre potentiel de l'arbre sur tous les processeurs pour que le temps moyen d'exploration d'une branche d'un processeur donné soit approximativement le même pour tous les processeurs.

3.3.1 Principe de la méthode

Le principe de la méthode consiste à diviser l'arbre de recherche en tâches indépendantes de telle sorte que leur nombre soit supérieur au nombre de processeurs disponibles. Ainsi, on peut affecter plusieurs tâches sur chaque processeur. L'algorithme est paramètré par un entier **degré** qui correspond au nombre minimum de tâches de recherche par processeur.

La méthode repose sur une phase de type « *generate-and-test* » sur une profondeur de génération suffisante pour que le nombre de branches soit supérieur à **degré** fois le nombre de processeurs. La figure 5.5 présente un arbre de génération pour un problème donné, avec 7 processeurs et un degré de 3. Il a ainsi fallut développer 3 niveaux dans l'arbre pour obtenir $24 = 3 \times 4 \times 2$ tâches (les tâches sont numérotées de 0 à 23 : chiffres romans sur la figure 5.5).

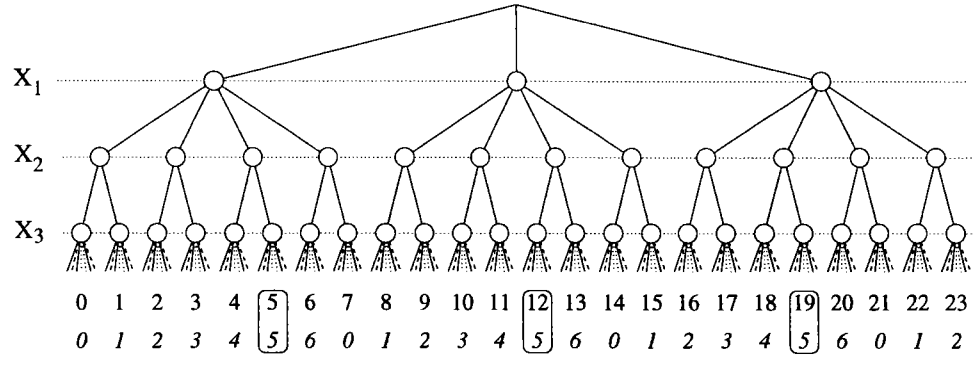


Figure 5.5 : Affectation des branches de recherche sur le processeur P_5.

L'affectation d'une tâche à un processeur se fait modulo le nombre de processeurs. Ainsi les tâches affectées à un processeur couvrent toute la largeur de l'arbre. Les numéros en italique sur la figure 5.5 correspondent aux numéros des processeurs qui reçoivent la tâche considérée. Dans cet exemple, les tâches 5, 12 et 19 sont affectées au processeur P_5 (encadrées sur la figure).

Si N est le nombre de processeurs disponibles pour la recherche parallèle, les tâches affectées à un processeur P_i ont un numéro de la forme $i + \alpha N$, si $0 \leq \alpha \leq$ **degré**. Une tâche est stockée comme un environnement de résolution autonome (*i.e.* un CSP sous-ensemble du CSP d'origine). Le partitionnement se fait par division des domaines des variables concernées par la phase de génération : un chemin d'instanciation est construit dans une tâche en réduisant le domaine de chaque variable à une seule valeur. Ainsi lors du traitement de la tâche, la recherche commence par la première variable et descend sur toute la profondeur de la génération de façon linéaire.

Le calcul du chemin d'instanciation d'une tâche dans la partie haute de l'arbre de recherche peut se faire de manière totalement locale, sans connaître les autres tâches. Ceci est permis par l'utilisation du caractère intrinsèquement équilibré d'un arbre de génération : on peut connaître par un simple calcul algébrique, le nombre de tâches qui précèdent une autre, et leur répartition au niveau des branchements à chaque nœud.

3.3.2 Discussion

Divers points sont à souligner concernant cette méthode de répartition de charge. Nous discutons dans les paragraphes suivants, de ses avantages et inconvénients.

Problème de la génération totale : le principal inconvénient de la méthode est qu'elle génère les branches de recherche selon le parcours d'un arbre total équilibré. Ceci peut poser des problèmes dus au fait que cette génération systématique construit des nœuds qui ne seraient pas explorés par les algorithmes de recherche réels, ce qui induit la création de tâches qui seraient trivialement supprimées par un algorithme simple. On utilise en fait ce système pour pouvoir calculer une affectation de manière totalement locale : en effet, si on met en œuvre un système de détection des branches trivialement infructueuses, il est nécessaire que chaque processeur parcourt tout le début de l'arbre pour calculer l'affectation de chaque tâche avant de s'attribuer ses propres branches.

Cela induit tout d'abord des difficultés dans l'évaluation de la profondeur de génération initiale puisque l'arbre est alors irrégulier. Cela transforme surtout l'initialisation de la charge en une procédure exponentielle dans la profondeur de génération initiale, alors que la méthode que nous présentons est polynômiale.

Répartition des branches de recherche : le paramètre **degré** que nous introduisons vise justement à réduire les effets négatifs de la génération totale sur un bon équilibre : en attribuant plusieurs branches indépendantes à un même processeur, on répartit les branches trivialement vides sur l'ensemble des processeurs. La répartition que nous utilisons se fait avec un modulo, et non en attribuant des tâches consécutives à un même processeur, pour que les ancêtres des branches soit toujours diffusés sur toute la largeur de l'arbre. Ainsi, si une branche conduit à un échec sur une valeur de la première variable, elle induira un déséquilibre de charge qui sera réparti sur l'ensemble des processeurs.

Conceptuellement, on peut considérer que la méthode réalise plus une répartition du déséquilibre de l'arbre de recherche, qu'un équilibre de la charge de chaque processeur puisqu'elle est conçue pour que les processeurs aient une charge à peu près équivalente, quel que soit le déséquilibre de l'arbre.

Choix du degré de multiplication : le choix du paramètre principal de la méthode (le nombre de tâches affectées à chaque processeur) est une affaire de compromis. En effet, une valeur élevée provoquera une augmentation de la profondeur de génération totale, et donc une augmentation artificielle du travail de recherche.

À l'opposé, une valeur faible pour **degré** rend l'équilibre de la charge plus sensible au déséquilibre de l'arbre, et induira donc un coût plus important pour l'algorithme en terme de communications pour l'équilibre dynamique de la charge des processeurs.

Il est difficile de discuter de cette valeur sans expérimentation préalable car elle sera très dépendante de nombreux facteurs, concernant tant la machine cible (ratio coût d'une communication/coût d'un calcul, nombre de processeurs, topologie, ...) que le problème à résoudre (nombre de variables, taille des domaines, satisfiabilité des contraintes, ...).

3.4 Divisibilité d'une tâche

La division d'une tâche constitue un mécanisme coûteux par rapport à la recherche de solution, car elle nécessite la mise à jour d'importantes structures de données. Il apparaît donc pertinent d'éviter au maximum les divisions de tâche qui entraîneraient un surplus de calcul supérieur à un traitement local de ces tâches. Nous définissons donc dans ce paragraphe, une heuristique d'évaluation de la pertinence d'une division de tâche.

À l'instar de Naganuma et Ogura pour leur *lazy load balancing* [NO 94], la méthode que nous proposons est basée sur des différences de hauteurs dans l'arbre de recherche. Toutefois, elle exploite une connaissance sur la structure de l'arbre de recherche qui fait défaut au contexte de travail utilisé par ces derniers: la profondeur maximale de l'arbre est connue, puisqu'au plus n variables seront instanciées.

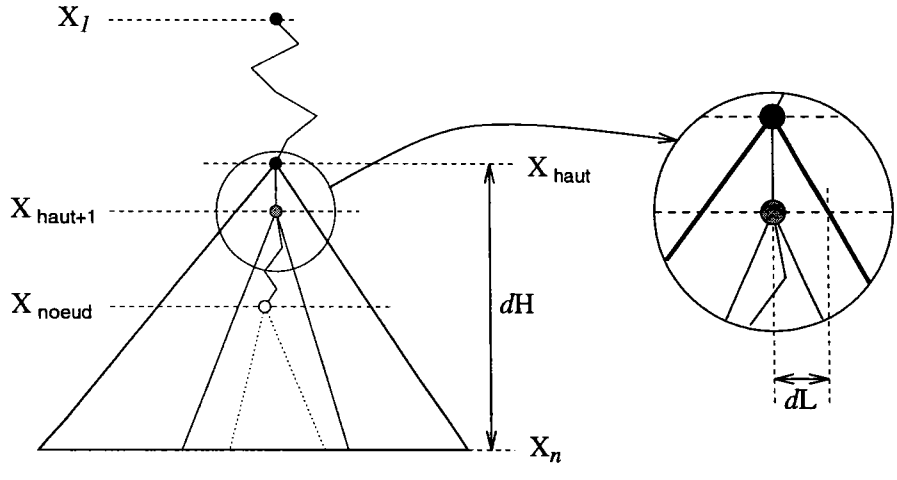


Figure 5.6 : Schéma d'évaluation de la divisibilité d'une tâche de recherche.

Au moment de décider si sa tâche courante peut être divisée, un processeur évaluera deux caractéristiques fondamentales (*cf.* figure 5.6) :

1. La *différence de hauteur* (dH) qui correspond à la profondeur potentielle d'instanciation de la tâche. On a $d\mathrm{H} = n$ - **haut**. Ce paramètre permet d'estimer le taille de l'espace de recherche qui était dévolu à la tâche – qui est alors en $\mathrm{O}(d^{d\mathrm{H}})$.

2. La *différence de largeur* (dL) qui tente d'évaluer le temps déjà passé par le processeur pour résoudre la tâche. Il correspond au nombre de valeurs non encore instanciées dans le plus haut domaine de la tâche ($D_{\mathsf{haut+1}}$).

On définit de plus, un seuil pour la hauteur en deçà duquel la tâche sera considérée comme non-divisible. Dans le cas où le seuil est atteint ou dépassé, un second seuil intervient sur la différence de largeur. L'heuristique de divisibilité obéira donc à l'équation

$$(s\mathrm{H} \leq \frac{d\mathrm{H}}{n}) \wedge (s\mathrm{L} \leq \frac{d\mathrm{L}}{\sigma})$$

pour déterminer si une tâche est divisible[7]. Les paramètres de l'heuristique (sH et sL) seront à optimiser expérimentalement.

3.5 Division d'une tâche

Dans le contexte des CSP, les heuristiques de division définies par Rao et Kumar dans [RK 87] (moitié haute, moitié basse et division transverse) restent naturellement valides. Toutefois, leur efficacité risque d'être remise en cause par le problème du contexte de recherche (*cf.* note 5.2). En effet, il est nécessaire de transmettre et de mettre à jour d'importants contextes de recherche pour un algorithme qui met en œuvre une vérification de consistance en avant, telle que *FC*, ce qui sera relativement coûteux. Le principe de la division d'une tâche aura donc intérêt à prendre en compte ce problème afin d'en limiter les effets.

Le premier point à aborder est la possibilité de regrouper les tâches qui partagent le même contexte, afin de réduire globalement les coûts de transmission/changement de contexte. La figure 5.7 présente un schéma de regroupement. Les nœuds du niveau X_h qui sont susceptibles d'être envoyés à un autre processeur conduiront tous à instancier les variables X_{h+1} à X_n. De plus, ils appartiennent tous au contexte d'un même nœud père sur X_{h-1} (en noir dans la figure).

Plutôt que d'envoyer séparément des nœuds de X_h (par exemple b et c) en calculant et transmettant séparément leur contexte, nous proposons d'envoyer le nœud père X_{h-1}, dont l'environnement à déjà été calculé. Dans ce cas, on transmet les nœuds qui doivent réellement être calculés par le processeur distant en appliquant un *sur-filtrage* sur D_h dans le contexte transmis pour X_{h-1}.

Il apparaît que les heuristiques de division de tâche pour la résolution de CSP devront privilégier les regroupements de nœuds sur un même niveau d'instanciation, afin de limiter le coût des changements de contexte. Dans cette optique, l'heuristique de division transverse qui envoie la moitié des nœuds sur chaque niveau d'instanciation sera probablement à proscrire. Nous proposons, à l'opposé de celle-ci :

[7] la valeur σ correspond au nombre de valeurs dans le domaine d'origine de la plus haute variable instanciée dans la tâche (= |domaine[haut + 1]|).

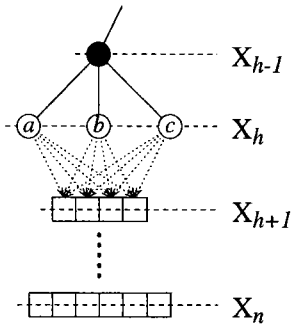


Figure 5.7 : Principe de regroupement de contextes pour plusieurs sous-tâches.

Définition 5.4 *L'heuristique de* **division minimale** *consiste à envoyer au processeur demandeur, la moitié des nœuds sur le plus haut domaine divisible.*

Le principal avantage de cette heuristique est qu'elle n'impliquera la transmission que d'un seul contexte à chaque division de tâche. Toutefois, ce contexte sera assez volumineux puisque le nombre de variables non instanciées sera alors important.

On pourrait citer dans la liste des inconvénients de l'heuristique de division minimale le fait qu'elle n'ait pas pour objectif de diviser la charge en deux parties sensiblement égales afin de réaliser un équilibre local entre l'émetteur et le récepteur de la division. En effet, elle ne tient pas compte du nombre de nœuds laissés en suspend dans la tâche courante, mais uniquement du haut du sous-arbre. Or, le sous-arbre de recherche qui se trouve sous un nœud de profondeur h induit un parcours de complexité en $O(d^{n-h})$. Ainsi, plus un nœud est haut dans l'arbre, plus son coût d'exploration sera potentiellement élevé. En se contentant de diviser le plus haut domaine en deux parties égale, on obtient donc une division proche de l'équité théorique, si l'arbre de recherche n'est pas trop déséquilibré.

3.6 Transfert de charge

Si l'intérêt de compresser les données transmises lors d'un échange de tâches entre deux processeurs est apparu dans le cadre de la programmation logique, il sera encore plus évident dans le cadre de la résolution des CSP. En effet, l'utilisation de techniques de recherche en avant induit la création de volumineux contextes de recherche qui sont dépendants des sous-arbres associés aux tâches explorées.

La principale difficulté dans le transfert d'une tâche de recherche pour un CSP concernera les modifications provoquées par les filtrages dans les domaines futurs. Le processeur récepteur devra être en mesure de les reconstituer pour pouvoir continuer la recherche dans le sous-arbre qu'il a reçu.

Il est possible de recalculer la totalité du contexte d'une tâche par son chemin d'instanciation. En effet, le filtrage des domaines non instanciés est totalement dépendant des instanciations déjà effectuées. Toutefois, ce calcul peut s'avérer particulièrement coûteux car il impose de réitérer toutes les vérifications de consistances induites par le chemin d'instanciation. Cette technique devra donc être réservée à des structures de machines dont les processeurs sont rapides par rapport au réseau de communication qui les lie – stations de travail sur réseau local, par exemple.

La Connection Machine CM-5 que nous avons utilisée pour expérimenter nos algorithmes (*cf.* chapitre 6) est constituée de processeurs relativement lents[8], mais son réseau de communication permet des vitesses de transfert de données importantes. Nous avons donc choisi une option intermédiaire réalisant un certain compromis entre la transmission intégrale des domaines, et le recalcul complet : les processeurs échangent la liste des modifications à apporter aux domaines futurs. Ainsi, le récepteur d'une tâche construira le contexte en appliquant les modifications sans toutefois avoir à les calculer explicitement.

3.7 Terminaison de la recherche

L'objectif de la recherche de solution pour un CSP est de donner le plus rapidement possible une réponse, et donc de terminer. On distinguera deux types de terminaison :

1. *la terminaison positive* lorsqu'un processeur arrête les autres après avoir trouvé une solution ;

2. *le terminaison négative* lorsque tous les processeurs ont terminé leur recherche locale et le CSP n'est donc pas satisfiable.

La terminaison positive ne pose pas de problème particulier : quand un processeur trouve une solution, il avertit les autres processeurs par un mécanisme de diffusion. Dans le cas d'une terminaison négative, nous proposons plusieurs techniques de mise en œuvre :

- Par *synchronisation explicite* : combinaison d'un booléen qui indique l'état d'activité de chaque processeur (action de type ET/OU parallèle). Certaines architectures de machines parallèles disposent de mécanismes câblés pour réaliser efficacement ce genre de calcul, car la technique est utilisée dans de nombreux contextes. Si la recherche s'exécute sur une telle machine, il sera alors intéressant de l'utiliser.

- Par *effet de bord sur l'appariement* : des mécanismes de synchronisation sont déjà mis en œuvre pour équilibrer la charge des processeurs. Certaines techniques (centralisées, circulaires, topologiques, ...) supposent un parcours systématique des processeurs et peuvent aussi être utilisées par effet de bord pour détecter la terminaison de la recherche.

On notera toutefois que la détection de terminaison par effet de bord sur l'appariement circulaire – quand une demande de travail fait le tour des processeurs et revient à son expéditeur d'origine – ne peut être utilisée avec une fonction de division paresseuse car des processeurs peuvent avoir choisi de ne pas diviser leur charge sans pour autant avoir totalement terminer leur recherche.

Conclusion

Plus qu'un simple algorithme de recherche, la méthode que nous avons développée dans ce chapitre constitue un cadre de résolution parallèle pour les CSP. La fonction

[8] à l'époque de la conception de la machine, les processeurs étaient considérés comme plutôt rapides, mais les stations de travail actuelles dépassent largement leurs performances.

Résolution_Parallèle est générique et appelle les procédures qui remplissent les différentes fonctionnalités de la recherche et du parallélisme.

Les problèmes spécifiques présentés par la parallélisation de la résolution des CSP ont été abordés. Pour ce qui concerne les techniques de recherche, nous avons mis en évidence la difficulté que fait apparaître les algorithmes de recherche par consistance en avant. En effet, ceux-ci construisent des contextes de résolution sensiblement différents d'une branche de résolution à l'autre, et les techniques parallèles devront prendre en compte ces considérations. Les techniques d'amélioration du *backtracking* font apparaître, pour leur part, des difficultés pour la prise en compte de sauts hors du contexte local d'une tâche de recherche.

Certaines techniques parallèles développées pour la recherche de solution dans le cadre de la programmation logique restent applicables pour les CSP. Nous avons toutefois défini de nouvelles méthodes qui s'adaptent mieux à la spécificité de ce type de problème. En premier lieu, nous avons présenté une méthode de répartition initiale de la charge basée sur l'algorithme *generate-and-test*, qui vise en fait à répartir le déséquilibre de l'arbre de recherche sur tous les processeurs, sans nécessiter de communications entre ceux-ci.

Le second point important relatif aux techniques parallèles spécifiques aux CSP est la division dynamique d'une tâche de recherche. Il porte tout d'abord sur la divisibilité de la tâche, en définissant deux dimensions caractéristiques de celle-ci, afin d'estimer sa taille moyenne et son « âge ». En effet, comme le nombre de variables et la taille de leurs domaines sont connus et finis, il est possible de borner la taille potentielle du travail de recherche requis pour l'exploration d'un sous-arbre de recherche. Le parcours de celui-ci se faisant de manière totalement séquentielle à l'intérieur d'un processeur, on peut de plus estimer le temps restant à parcourir le sous-arbre par rapport au parcours déjà effectué.

L'utilisation d'algorithmes de recherche par vérification de consistance en avant induit un travail important dans le filtrage des domaines des variables non encore instanciées. De plus, ce filtrage est fortement dépendant des instanciations déjà réalisées, et donc de chaque sous-arbre de recherche. Lorsqu'un processeur envoie une tâche de recherche, il est alors nécessaire de tenir compte de l'importance de la taille de ces contextes, qui rendra les communications et les mises à jour coûteuses. Pour cela, nous avons enfin défini une technique de division d'un sous-arbre de recherche qui tient compte de ces problèmes : la division minimale. Celle-ci envoie la moitié des nœuds en attente sur la variable la plus haute dans l'arbre. Elle permet ainsi de regrouper les contextes de chaque nœud en un seul, réduisant d'autant le coût des mises à jour.

Chapitre 6

Expérimentations parallèles

DE NOMBREUSES propositions de parallélisation des algorithmes de recherche de solution(s) ont été définies, tant pour le cadre de formalisation des CSP que pour celui de la programmation logique. Nous expérimentons certaines d'entre elles dans ce chapitre, pour mettre en avant le choix de certains paramètres de parallélisation au regard de l'efficacité.

Du fait de la grande diversité des architectures parallèles existantes (*cf.* chapitre 4), il ne nous sera pas possible de réaliser une étude expérimentale exhaustive, qui prendrait en compte toutes les particularités architecturales. Nous nous contenterons en fait d'une seule architecture de machine parallèle dont nous ferons varier le nombre de processeurs. Ainsi, les résultats obtenus ne constituerons pas une règle monolithique sur le choix des différentes méthodes de parallélisation de la recherche, mais une orientation permettant de valider, ou invalider selon le cas, la pertinence de l'emploi de telle ou telle technique. Toutefois, la machine utilisée (une Connection Machine CM-5) est conçue autour d'un réseau à interconnection totale et permet donc de simuler toutes les architectures. Les différences qui pourraient survenir ne concerneraient que de faibles écarts en terme de temps de communication.

Le paragraphe 1 présente l'environnement algorithmique des expérimentions. Les résultats sont ensuite exposés et analysés dans les paragraphes suivants. Nous comparons tout d'abord les algorithmes de recherche avec retour-arrière non chronologique, avec et sans prise en compte des sauts hors contexte (paragraphe 2). Nous étudions ensuite dans le paragraphe 3, les différentes combinaisons entre les techniques d'appariement et de division de tâches. Nous terminons enfin par une expérimentation de diverses heuristiques de divisibilité de tâches pour implémenter le principe de division paresseuse (paragraphe 4).

1 Environnement expérimental

Nous présentons dans ce paragraphe, l'environnement algorithmique des expérimentations parallèles. Le premier paragraphe présentera le cadre de génération aléatoire utilisé pour les expérimentations. Le second discutera de l'implémentation des algorithmes, et détaillera les caractéristiques de la machine parallèle utilisée.

1.1 Cadre de génération aléatoire

Dans la première partie de ce document, nous avons principalement concentré notre effort de travail sur les CSP n-aires. Les motivations de ce choix portaient alors sur les différences entre la recherche binaire et n-aire. Dans le cadre du parallélisme, le point crucial n'est plus la structure interne de l'algorithme de recherche, mais sur la répartition de l'arbre de recherche entre les différents processeurs. Ainsi, la distinction entre recherche n-aire et binaire n'est plus aussi importante de ce point de vue puisque l'arité des contraintes n'influe pas sur la « forme » de l'arbre de recherche.

Notre choix quant à l'arité des contraintes s'est donc porté sur une génération binaire. S'il est vrai que l'on s'écarte un peu des orientations prises dans la première partie, ce choix présente l'avantage de permettre la comparaison avec les expérimentations binaires réalisées dans le cadre séquentiel [Pro 94, Pro 96, Smi 94]. On notera que ce choix n'est pas restrictif puisque l'algorithme générique permet aussi de traiter les contraintes n-aires par des recherches en avant.

Nous rappelons le modèle de génération binaire de Prosser [Pro 94] : un ensemble de CSP binaires générés aléatoirement, sera défini par un quadruplet $\langle n, m, p_1, p_2 \rangle$, où :

- n est le nombre de variables dans un CSP généré ;
- m est la taille uniforme des domaines des variables ;
- p_1 est la probabilité pour une contrainte binaire possible d'apparaître dans le graphe du CSP ;
- p_2 est la probabilité qu'un couple de valeurs soit inconsistant, *i.e.* que le bit correspondant dans la matrice de la relation soit mis à **faux**.

Comme il a été démontré dans les études statistiques sur la recherche séquentielle (*cf.* chap. 1, paragraphe 7.2), le coût de résolution n'est pas uniforme sur tout le spectre de génération. En effet, celui-ci est sensiblement plus important pour les valeurs de p_2 qui correspondent au seuil de transition entre la *phase satisfiable* et la *phase insatisfiable.* L'intérêt du parallélisme sera maximal pour des problèmes dont la résolution séquentielle est rédhibitoire. Nous concentrerons donc les expérimentations uniquement sur des problèmes au seuil de transition.

1.2 Implémentation et parallélisme

De même que pour le cadre séquentiel, les algorithmes que nous avons testés sont implémentés en langage C. Les CSP sont stockés par des listes (liste des variables et liste des contraintes). Chaque relation est stockée par une matrice booléenne afin de

garantir un temps de vérification de consistance équivalent pour chaque contrainte. Les expérimentations ont été exécutées sur une Connection Machine CM-5.

La structure de la machine est un *fat-tree* (*cf.* figure 6.1) : les processeurs sont liés par un arbre de communication (de diamètre logarithmique) et la bande passante des niveaux de l'arbre augmente avec la hauteur [HT 93]. Nous avons utilisé une machine avec des partitions de 32 processeurs.

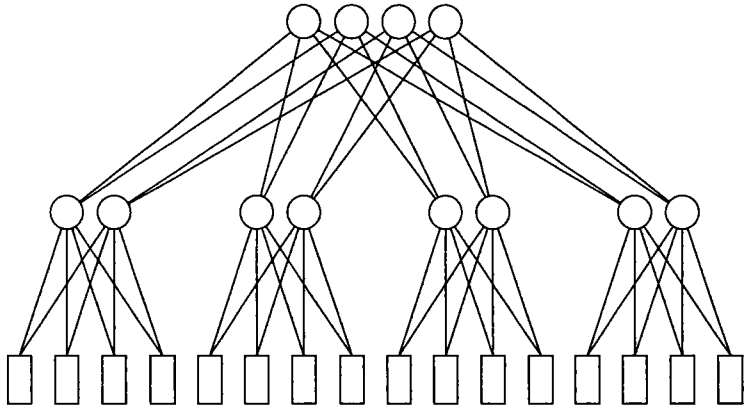


Figure 6.1 : Schéma de l'architecture de communication de la Connection Machine CM-5 avec 16 processeurs. Les processeurs sont symbolisés par des rectangles, et les routeurs de communication par des cercles.

Contrairement à l'expérimentation séquentielle, la principale unité de mesure des performances que nous utilisons est le temps d'exécution d'un algorithme. En effet, le nombre de vérifications de bit dans une matrice de relation permet bien d'évaluer un algorithme de recherche par rapport à un autre, mais ne tient pas compte des traitements annexes. Or, l'efficacité des programmes parallèles sera justement dépendante de ces traitements, si bien que seul le temps de calcul permet une bonne évaluation. On utilisera aussi le nombre de tâches échangées comme mesure de performance.

2 Prise en compte des sauts hors contexte

Les algorithmes de recherche mettant en œuvre des sauts non-chronologiques sont susceptibles de provoquer des sauts hors du contexte des tâches de recherche. Nous expérimentons dans ce paragraphe, l'opportunité de la prise en compte de tels sauts. Nous testons tout d'abord cette technique dans un algorithme de type *backjumping* (*BJ*) au paragraphe 2.1, puis dans une hybridation avec un algorithme de recherche par consistance en avant (paragraphe 2.2).

2.1 Algorithme *BJ* seul

La prise en compte des sauts hors contexte est expérimentée avec l'algorithme *BJ*. Nous donnons la courbe des temps de calcul de *BT* à titre de comparaison – sans sauts hors contexte. La répartition initiale de la charge suit le schéma présenté au paragraphe 3.3 du chapitre 5, l'appariement des processeurs est réalisé à l'aide d'un serveur, et les tâches sont divisées par leur domaine le plus haut (*cf.* chap. 5, paragraphe 3.5).

Les paramètres de génération sont $\langle 20, 10, 0.5, 0.37 \rangle$, ce qui correspond à des problèmes proches du seuil de transition. La méthode de calcul utilisée pour p_2 est $\widehat{p_2} \simeq 1 - m^{-2/p_1(n-1)}$, conformément à l'approximation théorique donnée dans [Pro 96]. Les résultats reportent des moyennes sur un total de 50 problèmes générés.

Les résultats des expérimentations sont présentés dans la figure 6.2. Une courbe correspond au temps moyen de calcul requis pour la résolution d'un problème par un algorithme donné (axe y). Le nombre de processeurs varie selon l'axe des x dans les courbes.

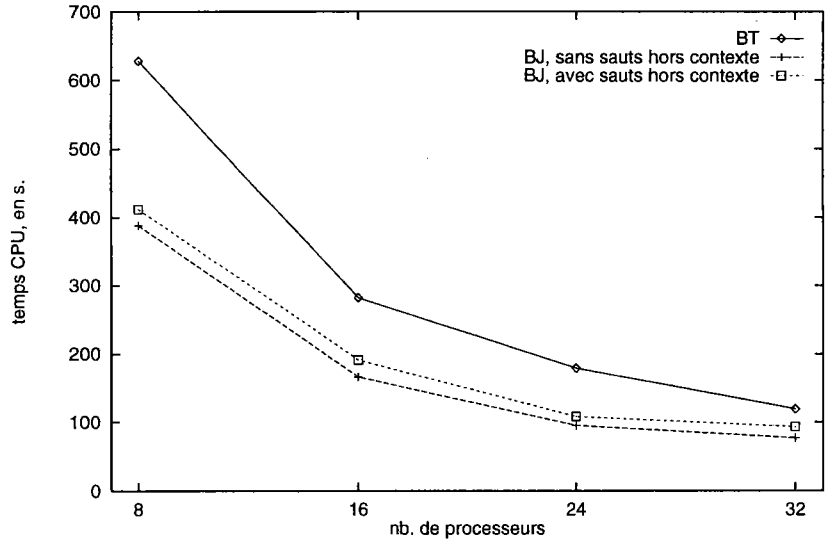


Figure 6.2 : Évaluation de la prise en compte des sauts hors contexte. Les graphes ont 20 variables, 10 valeurs par variable et 95 contraintes binaires ($p_1 = 0.5$).

Si *BJ* est toujours plus rapide que *BT* quelle que soit la prise en compte des sauts hors contexte – ce qui correspond aux résultats des comparaisons séquentielles – il apparaît que l'algorithme *BJ* est plus coûteux lorsqu'il tient compte des sauts hors contexte. Ce surcroît de temps de calcul s'explique par le nombre de communications des sauts, alors qu'une quantité significative de ceux-ci est inutile: les résultats (non présentés ici) reportent qu'environ $\frac{2}{3}$ des sauts transmis ont déjà été pris en compte par la tâche réceptrice au moment où le message est reçu.

2.2 Avec une consistance en avant

La recherche sur les CSP est généralement faite avec des algorithmes de vérification de consistance en avant. Nous expérimentons donc les sauts hors contexte avec ce type d'algorithme (*MFC-CBJ*). Les tests réalisés portent sur 50 problèmes générés aléatoirement par les paramètres $\langle 50, 10, 0.1, 0.55 \rangle$ (au seuil de transition). Les techniques de parallélisation (appariement, division, ...) sont les mêmes que dans le paragraphe 2.1.

La table 6.1 présente le nombre total de sauts hors contexte rencontrés/utilisés[1] dans ces expérimentations, en fonction du nombre de processeurs. Il apparaît nettement que le nombre de sauts est très faible pour ce type d'algorithme. Ceci s'explique par le fait que

[1] les sauts utilisés sont ceux qui n'avaient pas été rencontrés par la tâches réceptrice.

la plupart des inconsistances qui provoquaient des sauts dans *BJ* sont prises en compte au moment de la descente.

Table 6.1 : Nombre total (sur 50 problèmes) de sauts hors contexte (générés et utiles) avec l'algorithme *MFC-CBJ*.

processeurs	8	16	24	32
sauts générés	2	3	6	9
sauts utiles	0	0	1	2

Ainsi, l'algorithme *MFC-CBJ* rencontre peu de sauts en dehors du contexte d'une tâche. De plus, une importante proportion de ceux-ci sont repérés rapidement, ce qui rend leur transmission inutile. Nous ne présentons pas de courbe de temps pour cet algorithme car les différences entre *MFC-CBJ* avec et sans prise en compte des sauts hors contexte ne sont pas assez significatives. On notera de plus, que les tests avec *FC-CBJ* (c.-à-d. sans retardement ni marquage) donnent des résultats similaires.

Cette expérimentation laisse supposer que la prise en compte des sauts hors contexte dans les algorithmes parallèles de recherche sur les CSP est très peu utile sur le plan pratique. Ils constituent en fait un cas marginal dont le traitement spécifique risque le plus souvent d'alourdir la résolution plutôt que de l'améliorer.

3 Analyse des appariements et divisions

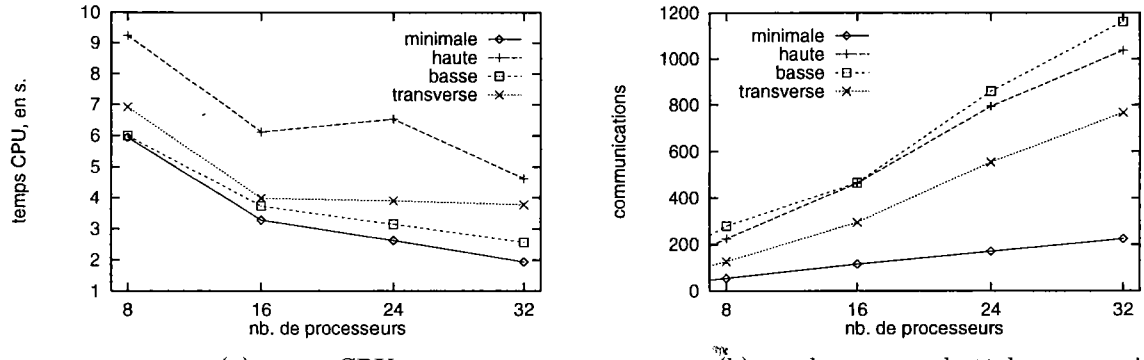
Dans ce paragraphe. nous expérimentons les différentes techniques d'appariement et de division de tâche de recherche. L'ensemble de problèmes test est composé de 50 problèmes générés aléatoirement selon le schéma de Prosser, avec les paramètres $\langle 50, 10, 0.1, 0.55\rangle$, c'est-à-dire au seuil de transition. Nous avons effectué des tests avec l'algorithme de recherche *MFC-CBJ* et *FC-CBJ*, *i.e.* avec et sans l'hybridation des techniques de retardement et de marquage. Toutefois, nous ne présentons que les résultats obtenus avec le second algorithme car il est apparu comme plus performant pour l'échantillon de test considéré. On notera que le rapport de performance entre ces deux algorithmes est constant en parallèle, quel que soit le nombre de processeurs utilisés. La résolution séquentielle (sur un seul processeur) coûtait 38 s. en moyenne pour un problème.

Nous présentons tout d'abord les résultats concernant l'appariement centralisé (paragraphe 3.1), en comparant les différentes techniques de division. Nous évaluons ensuite au paragraphe 3.2 ces dernières techniques, hybridées avec un appariement aléatoire, puis avec un appariement circulaire, avec ou sans propagation de la demande (= appariement circulaire et en étoile) au paragraphe 3.3. Nous faisons enfin la synthèse au paragraphe 3.4 de ces comparaisons avec un graphique regroupant la meilleure technique de division pour chacun des types d'appariements.

3.1 Appariement centralisé

Les figures 6.3(a) et 6.3(b) illustrent les résultats de la stratégie d'appariement centralisée. Nous pouvons observer que la technique de division minimale conduit à des

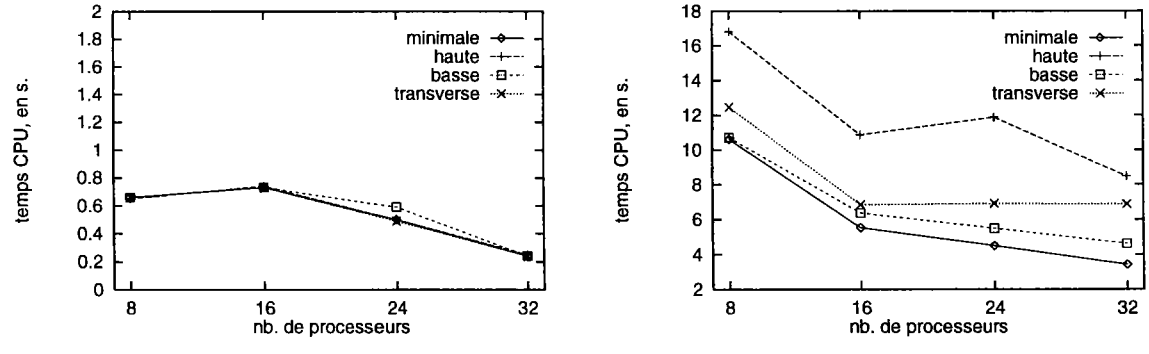
recherches plus rapides en terme de temps machine. À l'opposé, la division par l'envoi des nœuds les plus hauts est comparativement inefficace. Ces résultats doivent être mis en rapport avec le nombre de tâches transmises pour une analyse plus poussée. Là, nous observons que le nombre de tâches transmises pour la stratégie de division minimale est plus petit que pour les autres stratégies. Un tel résultat était d'ailleurs prévisible puisque cette technique a été conçue pour minimiser justement cette caractéristique.



(a) temps CPU moyen. (b) nombre moyen de tâches transmises.

Figure 6.3 : Performances pour tous les problèmes avec une stratégie d'appariement centralisée.

Toutefois, un résultat contraire à l'intuition apparaît pour les stratégies de division horizontales (division haute et basse) : les nombres de tâches transmises sont équivalents, mais les temps de résolution sont sensiblement différents. En fait, la taille des environnements transmis pour la division haute nécessite des mises à jour coûteuses lors du passage d'une tâche à une autre. À l'inverse, la division basse conduit à des transmissions et des mises à jour très réduites.



(a) problèmes satisfiables. (b) problèmes insatisfiables

Figure 6.4 : Temps CPU moyen pour les problèmes satisfiables et non satisfiables.

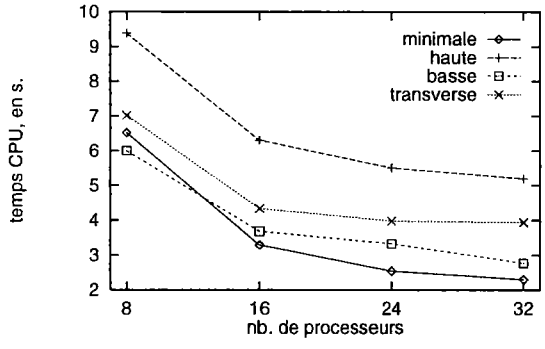
Comme les terminaisons des problèmes satisfiables et des problèmes non satisfiables sont différentes, nous séparons ces deux groupes pour une analyse plus fine[2]. Les figures 6.4(a) et 6.4(b) présentent les temps CPU moyens pour, respectivement, les problèmes satisfiables et les problèmes non satisfiables dans la même série d'expérimentation. Tout d'abord en terme de temps de calcul, les problèmes qui n'admettent pas de solution nécessitent environ huit fois plus de temps de résolution. De plus, les courbes relatives aux

[2]l'échantillon de 50 problèmes comportait 26 problèmes insatisfiables.

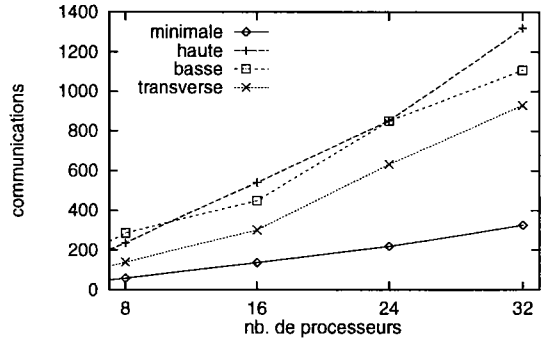
problèmes insatisfiables sont comparables à la figure 6.3(a) car les problèmes satisfiables influencent peu les performances globales. Pour les problèmes satisfiables, la stratégie de division n'a pas d'importance car une solution est généralement trouvée avant qu'un équilibre de charge ne soit nécessaire.

3.2 Appariement aléatoire

Les figures 6.5(a) et 6.5(b) présentent les performances obtenues pour la stratégie d'appariement aléatoire pour les quatre techniques de division de tâche.



(a) temps CPU moyen.

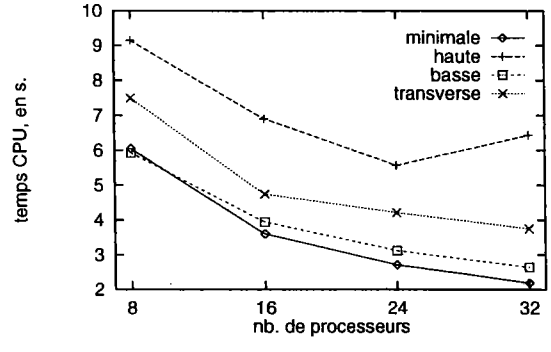


(b) nombre moyen de tâches transmises.

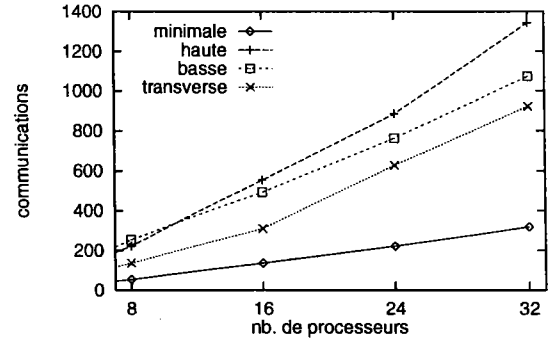
Figure 6.5 : Performances pour tous les problèmes avec une stratégie d'appariement aléatoire.

Comme pour l'appariement centralisé, on remarque que la stratégie de division minimale conduit à de meilleures performances en terme de temps de calcul. De plus, les techniques de division exhibent le même comportement relativement au nombre de tâches transmises. Toutefois, l'heuristique de division basse apparaît meilleure pour huit processeurs. Cette petite différence peut s'expliquer par un schéma d'appariement qui nécessite moins de communications pour trouver un partenaire qui puisse diviser sa tâche de recherche – rappelons que l'appariement centralisé demande au minimum quatre échanges de messages, alors que l'appariement aléatoire n'en requiert que 2.

3.3 Appariements circulaire et en étoile



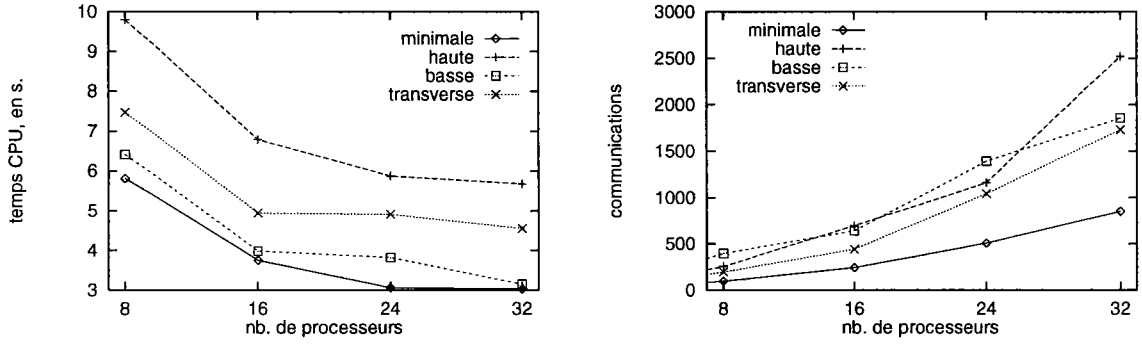
(a) temps CPU moyen.



(b) nombre moyen de tâches transmises.

Figure 6.6 : Performances pour tous les problèmes avec une stratégie d'appariement en étoile.

Les figures 6.6(a) et 6.6(b) présentent les performances obtenues pour la stratégie d'appariement en étoile. Celles de l'appariement circulaire sont reportées par les figures 6.7(a) et 6.7(b).



(a) temps CPU moyen. (b) nombre moyen de tâches transmises.

Figure 6.7 : Performances pour tous les problèmes avec une stratégie d'appariement circulaire.

La même hiérarchie entre les heuristiques de division se met globalement en place : la division minimale et la division basse sont comparables pour un petit nombre de processeurs, puis la division minimale tire parti de son faible nombre de tâches transmises pour surclasser la division basse. La stratégie de division haute apparaît comme inadaptée pour ce type d'architecture, car elle coûte sensiblement plus chère que les autres en terme de temps de calcul.

3.4 Synthèse

Les observations précédentes ont montré que l'heuristique de division minimale conduit globalement à de meilleurs résultats, quelle que soit la stratégie d'appariement. Pour sa part, la figure 6.8 compare les différentes stratégies d'appariement pour l'heuristique de division minimale. Il apparaît que l'appariement centralisé conduit à des résultats légèrement meilleurs.

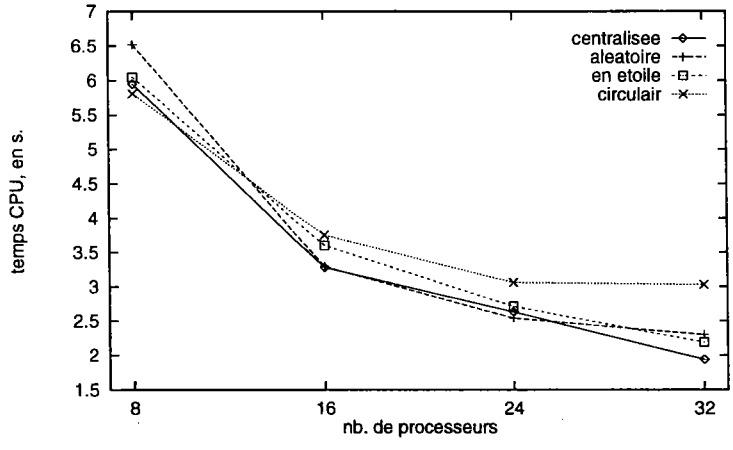


Figure 6.8 : Comparaison des 4 stratégies d'appariement avec la division minimale.

Pour conclure sur l'étude expérimentale de la combinaison entre appariements et divisions, nous considérons la meilleure technique de division, et nous explorons son efficacité

en terme de temps de calcul pour les quatre stratégies d'appariement considérées. La table 6.2 présente ces efficacités ($\frac{\text{temps séquentiel}}{\text{temps parallèle} \times \text{nombre de processeurs}}$). On peut ainsi observer des efficacités sub-linéaires pour les quatre algorithmes, mais raisonnables si l'on considère que les problèmes résolus sont dans la zone difficile.

D'un autre coté, on peut noter des efficacités super-linéaires pour les problèmes satisfiables avec un nombre réduit de processeurs (non illustré ici).

Table 6.2 : Efficacité parallèle avec l'heuristique de division minimale.

Nombre de processeurs	8	16	24	32
centralisé	80%	72%	60%	61%
aléatoire	73%	72%	62%	52%
en étoile	79%	66%	58%	54%
circulaire	82%	63%	52%	39%

4 Expérimentation des heuristiques de divisibilité

Nous expérimentons dans ce paragraphe, les heuristiques de divisibilité paresseuse. Les deux paramètres fondamentaux de ces méthodes sont :

- le seuil de hauteur maximale sH ;
- et le seuil de largeur maximale sL (*cf.* chap. 5, paragraphe 3.4).

Les échantillons d'expérimentation sont les mêmes que dans le paragraphe précédent, c.-à-d. 50 problèmes générés aléatoirement selon les paramètres $\langle 50, 10, 0.1, 0.55 \rangle$. L'initiation de l'équilibrage de charge se fait selon le schéma défini au paragraphe 3.3 du chapitre 5. La stratégie d'appariement est centralisée et la division d'une tâche reprend l'heuristique minimale.

La figure 6.9 présente les temps de calcul moyens utilisés pour la résolution d'un problème de l'échantillon. Nous ne présentons pas toutes les combinaisons entre les seuils de hauteur et et les seuils de largeur car toutes ne sont pas significatives.

La première courbe ($sH = 0, sL = 0$) correspond à une heuristique qui n'interdit jamais de division – cas des algorithmes utilisés dans le paragraphe 3.

Selon la variation du seuil de hauteur sH, on observe que le coût de résolution marque un point d'inflexion : il baisse entre 0 et $1/10^e$, puis augmente à partir de $1/10^e$. Ce phénomène s'explique par le fait que des tâches de petite taille sont échangées si $sH = 0$, alors que leur résolution coûte moins cher que l'échange. À l'inverse, si le seuil est important, on limite le parallélisme puisqu'on interdit la division à partir d'une certaine profondeur. La remonté du coût de résolution vient alors du fait qu'un grand nombre de processeurs ne travaillent plus en fin de recherche.

La variation du seuil de largeur est plus subtile. Si ce seuil est nul, cela signifie que l'heuristique ne tient pas compte de « l'age » de la tâche pour déterminer sa divisibilité. Avec un seuil de hauteur nulle, l'augmentation du seuil de largeur à un effet bénéfique,

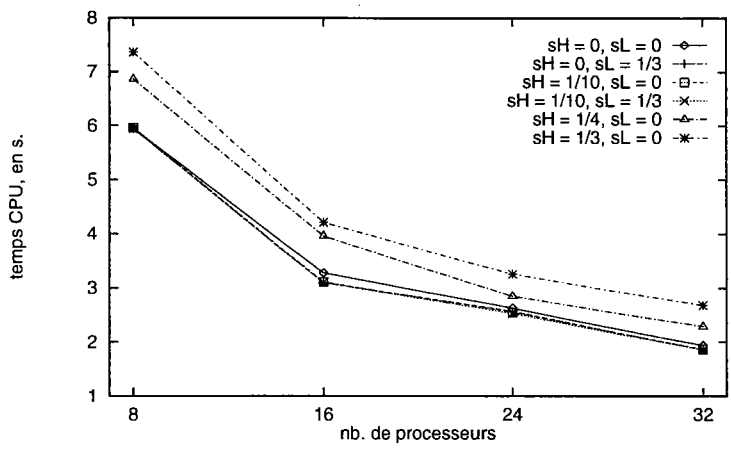


Figure 6.9 : Comparaison de différentes heuristiques de divisibilité avec la division minimale.

puisqu'il permet de diminuer le coût de résolution. Il nous est par contre difficile d'évaluer plus précisément la valeur optimale pour sL car les résultats des heuristiques avec différentes valeurs pour sL sont très proches – rappelons que les domaines ont une largeur maximale de 10 dans les échantillons, ce qui limite le nombre de seuils possibles.

Pour les autres valeurs testées du seuil de hauteur, la variation du seuil de largeur n'est pas apparu comme significative.

Conclusion

Nous avons expérimenté dans ce chapitre, un certain nombre de techniques de parallélisation développées dans le chapitre 5 afin de les valider sur le plan pratique. Toutefois, il ne faut pas perdre de vue dans les résultats exposés, le fait qu'un seul type de machine parallèle a été utilisé.

Le premier résultat concernant la prise en compte des sauts hors contexte dans les schémas de retour-arrière non-chronologique est plutôt négatif. En effet, il est apparu que leur prise en compte était généralement pénalisante puisque l'interruption que cela entraîne est coûteuse. En effet, les saut par retour-arrière non chronologique, qu'ils soient hors contexte ou non, sont repérés dans de nombreuses branches parallèles lors d'un mécanisme de descente dans l'arbre par consistance arrière (*BJ* par exemple). Ainsi, la plupart des sauts en dehors du contexte d'une tâche sont repérés et traités presque simultanément par le processeur destinataire, si bien que la notification du saut arrive généralement en retard. De plus, le phénomène de retour-arrière intra ou hors contexte est rare dans un schéma de vérification de consistance en avant, ce qui réduit d'autant son intérêt potentiel.

La combinaison des appariements avec les divisions classiques et la division minimale développée spécialement pour les CSP a permis d'exhiber deux résultats fondamentaux. Tout d'abord, la machine utilisée (une Connection Machine CM-5) est peu sensible au goulot d'étranglement des communications vers le serveur d'appariement centralisé, si bien que ce dernier fait sensiblement jeu égal avec l'appariement aléatoire. Il s'est même montré plus performant pour un nombre élevé de processeurs.

De plus, l'heuristique de division minimale que nous avons présentée dans le chapitre 5 s'est montrée plus efficace que les techniques classiques issues de la programmation logique. Elle conduit à la construction de contextes de taille importante, puisque de hauteur maximale, mais réduit sensiblement le nombre de tâches échangées. On pourra supposer que la différence sera encore plus significative dans le cas d'une architecture en réseau de stations de travail où les communications sont plus coûteuses.

Les expérimentations de différentes valeurs pour les seuils de hauteur et de largeur dans les heuristiques de détermination de la divisibilité d'une tâche ont montré l'intérêt de telles méthodes. Nous avons pu mettre en évidence un point d'inflexion dans la variation du seuil de hauteur. Toutefois, sa valeur exacte semble être très dépendante de la nature des problèmes résolus, et ne devra donc être retenu qu'à titre d'exemple.

Troisième partie

Application des CSP pour un problème de CAO

Chapitre 7

Étude de cas : le problème du bordereau de coupe

L'ÉTUDE DU FORMALISME et des méthodes de résolution pour les CSP reste très souvent générique, en faisant abstraction des spécificités de chaque problème concret. La généralisation des méthodes de génération aléatoire pour l'expérimentation des algorithmes tend d'ailleurs à privilégier certaines méthodes bien adaptées à ce type de problèmes, et ainsi à négliger des algorithmes qui permetraient un gain de temps notable pour un problème réel donné.

L'objectif de ce chapitre, et plus globalement de toute la troisième partie de ce document, est d'appliquer le formalisme des CSP à un problème spécifique. Nous pourrons ainsi y appliquer nos algorithmes, et nous tenterons de dégager les méthodes développées pour les CSP généraux qui s'appliquent le mieux au problème étudié.

Notre choix du problème spécifique à étudier s'est porté sur le *problème du bordereau de coupe*, que nous présenterons dans le paragraphe 1. Il s'agit d'un cas particulier de placement qui constitue un thème récurrent dans les applications aux problèmes de satisfaction de contraintes.

Le problème étudié présente un certain nombre de difficultés spécifiques que nous détaillerons dans le paragraphe 2. Nous présenterons pour conclure ce chapitre, un certain nombre de méthodes particulières utilisées dans la littérature ainsi que dans l'industrie pour résoudre le problème du bordereau de coupe (paragraphe 3). Celles-ci ne font toutefois pas appel au formalisme CSP, mais permettent néanmoins d'avoir un aperçu de diverses méthodes définies pour contourner les difficultés spécifiques du problème.

1 Description du problème

Le problème du bordereau de coupe fait partie d'un processus global de conception d'un vêtement. Dans le cadre qui nous intéresse, il fera donc partie du domaine de la *conception assistée par ordinateur*, ou CAO[1] puisque l'ordinateur constituera un acteur important du processus de conception. Nous présentons en premier lieu le processus global de conception (paragraphe 1.1), en détaillant notamment les objets manipulés dans chaque étape, puis nous nous concentrons sur le problème du bordereau de coupe dans le paragraphe 1.2.

1.1 Processus de conception

Qu'il soit informatisé ou non, le processus de conception/fabrication d'un vêtement suit toujours les mêmes étapes que nous divisons en quatre grandes phases – schématisées pas la figure 7.1 :

- La phase de *design* qui consiste à dessiner le modèle. Elle est généralement exécutée par un styliste et déterminera la forme et les motifs qui constitueront le vêtement final.
- La phase de réalisation du patron qui consiste à faire une liste exhaustive des pièces constituant le modèle. Cette étape est souvent réalisée en même temps que la phase de *design*, mais nous les séparons pour distinguer le coté artistique de la conception – qui ne peut être fait sans intervention humaine – du coté technique qui pourra être aisément automatisé. La *gradation*, *i.e.* la génération de pièces en foncion de différentes tailles, intervient dans cette phase.

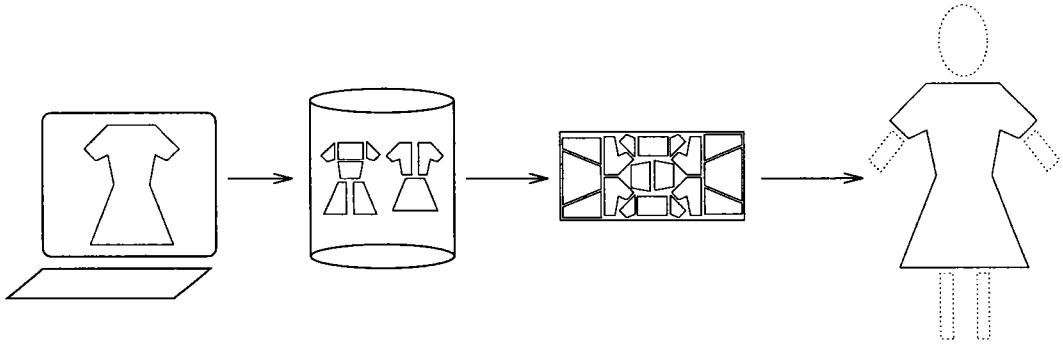


Figure 7.1 : Processus de conception/fabrication d'un vêtement.

- la phase de création du *bordereau de coupe* qui consiste à placer les éléments du patron sur les coupons de tissus.
- la phase d'*assemblage* pour coudre les pièces de tissus ensemble afin d'obtenir le vêtement voulu.

[1] le sigle CAD est utilisé dans la littérature anglaise, pour *computer aided design*.

1.2 Problème du bordereau de coupe

Informellement, le problème du bordereau de coupe peut s'énoncer comme suit : « placer les éléments d'un patron de découpe sur une pièce de tissu ». La procédure de placement aura donc en entrée une liste des pièces à découper, ainsi qu'un ensemble de contraintes particulières :

- Des *contraintes d'orientation* s'appliquant à une pièce donnée. En effet, les qualités du tissu varient selon l'orientation de la pièce par rapport à la *trame*[2], notamment en ce qui concerne la résistance à l'étirement. Concrètement, les éléments du patron qui nécessitent un alignement sur la trame sont orientés par une droite ou un vecteur.
- Des *contraintes de motif* qui permettent d'aligner les pièces à découper sur les motifs du tissu, pour des raisons esthétiques. Elles s'appliqueront en général selon un quadrillage de l'espace de placement pour correspondre aux *périodes* du coupon car les motifs sont souvent cycliques.
- Des *contraintes de fabrication* dues généralement aux méthodes d'industrialisation. On retrouvera dans cette catégorie, les espaces minimums entre les pièces pour permettre une découpe aisée, ou l'ordre des pièces sur le coupon pour optimiser les flux de matière autour des machines de découpage.

En plus de ces contraintes explicites qui s'appliqueront à une pièce, ou à un ensemble de pièces, le système de placement devra tenir compte d'un ensemble de contraintes implicites appliquées à toutes les pièces : le non recouvrement des éléments du patron sur le coupon. En sortie de la procédure de placement, on obtient donc un bordereau qui constituera le plan de coupe destiné à l'atelier de fabrication.

On associe en général au problème de placement, un certain nombre de fonctions de coût à optimiser. On pourra par exemple vouloir minimiser la taille des chutes de tissu.

2 Difficultés spécifiques au problème

En plus des difficultés liées au caractère combinatoire de ce type de problème, le bordereau de coupe en confection pose un certain nombre de difficultés spécifiques qui orienteront fortement les choix algorithmiques pour les méthodes de résolution. Nous discutons dans ce paragraphe des difficultés spécifiques relatives à ce problème.

2.1 Domaines de définition

La modélisation classique du problème du bordereau de coupe consiste à assigner une position à chaque pièce du patron sur le coupon de tissu. Chaque position est donnée par un vecteur de translation t et un angle de rotation α, comme dans la figure 7.2.

Le domaine de définition d'une position est donc de $\mathbb{R} \times \mathbb{R} \times 2\pi$ (vecteur de translation dans $\mathbb{R}^2$ et angle de rotation sur une révolution). Si on considère que les pièces sont forcément orientées selon la chaîne ou la trame, il reste tout de même $\mathbb{R}^2 \times 4$ positions possibles pour chaque pièce.

[2] les fils de *trame* croisent en largeur les fils de *chaîne* sur le métier à tisser.

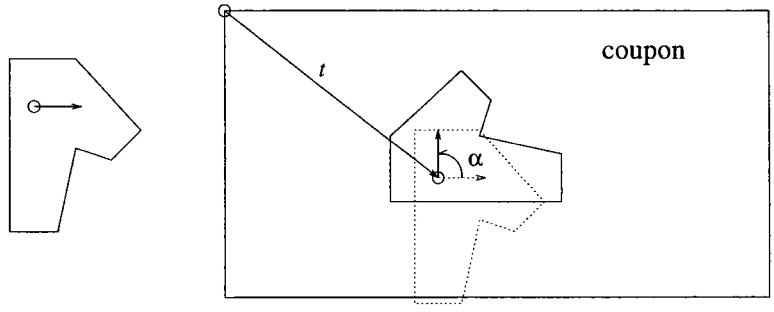


Figure 7.2 : Positionnement d'une pièce sur un coupon.

Outre les problèmes inhérents à l'arithmétique réelle sur les ordinateurs, les algorithmes de résolution de tels problèmes devront tenir compte de la taille importante des domaines de définition qui rendent les techniques énumératives difficiles d'emploi car elles ont été définies spécifiquement pour des domaines finis et discrets.

2.2 Diversité des formes

La plupart des problèmes de placement (caisses dans un camion, tâches dans un emploi du temps, ...) travaillent avec des objets convexes dans les espaces de placement. Les algorithmes sont alors simplifiés puisqu'ils n'ont pas à tenir compte des encastrements possibles entre les objets.

Dans le cas de pièces de patron pour la confection, les objets à placer peuvent être constitués de toute forme connexe dans $\mathbb{R}^2$, la seule limite dans les circonvolutions des contours semblant être l'imagination des *designers*. Il est naturellement possible de se ramener à un algorithme travaillant sur les enveloppes convexes des contours, mais cette simplification induit alors une importante perte de matière qui n'entre pas toujours dans les quotas industriels de perte. La figure 7.3 illustre la nécéssité de gestion des encastrements dans les algorithmes de résolution du problème.

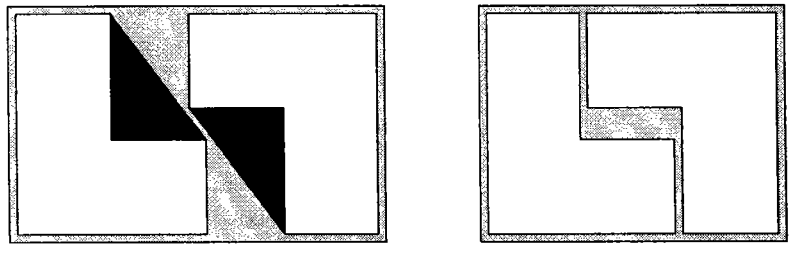


Figure 7.3 : Exemple de positionnement de deux pièces concaves en enveloppe convexe et avec encastrement.

Avec un algorithme de placement à enveloppe convexe, une partie de la surface de chute est générée en complétant les parties concaves des contours (en gris foncé sur la figure). Cette surface ne pourra être utilisée en recouvrement d'une autre pièce à cause des contraintes implicites de non-recouvrement. Il est généralement admis dans l'industrie, que les pièces de confection non convexes sont suffisament fréquentes pour qu'il en soit tenu compte dans les algorithmes de placement.

2.3 Mode d'utilisation du coupon de tissu

Dans la confection industrielle – qui constitue le principal « client » d'un algorithme de calcul d'un bordereau de coupe – la surface de placement n'est pas finie, mais est constituée en rouleau. Ainsi, les méthodes de résolution du problème se doivent de remplir *séquentiellement* le rouleau qui sera utilisé à nouveau par les placements suivants. La figure 7.4 présente un exemple d'utilisation d'un rouleau de tissu.

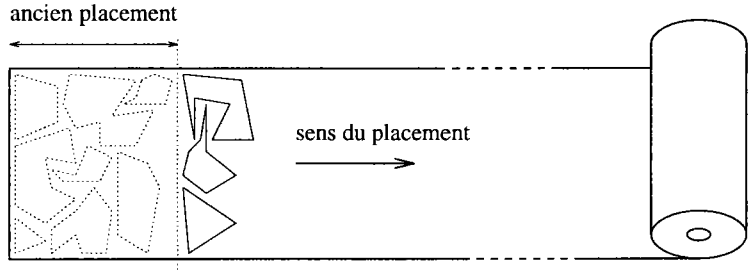


Figure 7.4 : Exemple d'utilisation d'un rouleau de tissu par rapport aux différents bordereau de coupe.

Après la découpe d'un bordereau de coupe, le rouleau est recoupé dans sa largeur (ligne pointillée) pour servir de point de départ pour un nouveau bordereau. La principale fonction à optimiser est donc la longueur du morceau utilisé pour un placement. Ce problème constituera d'ailleurs la principale difficulté rencontrée par les algorithmes incomplets de type *tabou* ou *recuit simulé*[3] car les modifications locales pourront engendrer un décalage de toutes les pièces à droite du placement modifié.

3 Méthodes spécialisées de résolution

Le problème du bordereau de coupe peut être résolu par un grand nombre de méthodes. Les algorithmes classiques restent valables pour ce type de problème, mais la conjugaison des deux premières présentées au paragraphe précédent les rendent difficiles à utiliser en pratique. En effet, la combinatoire élevée peut engendrer un espace de recherche très important, qui handicape fortement les algorithmes complets du type *branch and bound.* De plus, la gestion des encastrements rend le traitement des modifications locales périlleux dans les algorithmes incomplets de type *recuit simulé.*

Ainsi, la plupart des méthodes utilisées pour résoudre le problèmes du bordereau de coupe mettent en œuvre des techniques hybrides entre les méthodes d'optimisation et les méthodes de recherche de solution : le schéma global suit celui de la recherche exacte de solution, et l'optimisation est réalisée par des heuristiques d'ordre sur les objets et sur les positions possibles.

Les méthodes présentées dans ce paragraphe sortent quelque peu de la traditionnelle classification théorique entre algorithmes complets et algorithmes incomplets. En effet, elles peuvent être considérées comme complètes car elles trouvent toujours une solution. Elles suivent d'ailleurs un mécanisme énumératif pour respecter la nature inhéremment

[3]ces algorithmes fonctionnent en calculant une solution au hasard, puis en tentant de l'améliorer par des modifications locales.

séquentielle du mode d'utilisation du coupon de tissu. D'un autre coté, elles sont incapables de donner la solution optimale globale, ou du moins de montrer que la solution donnée est optimale.

Nous présentons dans ce paragraphe, deux méthodes spécialisées qui contournent les difficultés évoquées dans le paragraphe 2 : la méthode de translation de colonnes au paragraphe 3.1 et la méthode des modules rectangulaires (paragraphe 3.2). Les techniques spécialisées qui font appel à des notions de programmation linéaire comme [Elo 92] ne seront pas présentées ici[4].

3.1 Translation de colonnes

L'objectif de la méthode de translation de colonnes proposée par Moreau dans [Mor 73] est principalement de respecter le sens du fil et les motifs. Elle est inspirée des méthodes manuelles de placement. Son principe consiste à construire des colonnes de placement par translation des pièces – les pièces sont orientées selon le fil du tissu, et les translations permettent de respecter l'orientation au cours du placement. La figure 7.5 illustre cette méthode par un exemple.

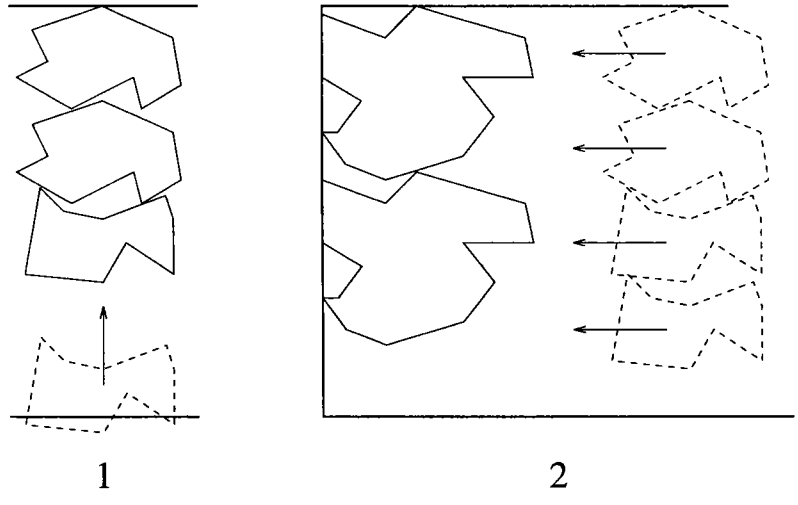


Figure 7.5 : Exemple de placement par la méthode de translations de colonnes.

Les pièces à placer sont classées par surface décroissante. Les contours sont approchés par des englobants constitués de segments de droites afin de simplifier les calculs d'intersections. Les translations sont effectuées en deux étapes :

1. Les colonnes sont construites en emboitant les pièces à placer par des translations verticales.

2. Les colonnes sont ensuites agglomérées entre elles par des translations horizontales.

L'auteur de la méthode n'a pas étudié les combinaisons de translations verticales et horizontales pour la construction des colonnes et le positionnement des colonnes les unes

[4] une présentation générale des méthodes de résolution des problèmes de placement pourra être trouvée dans [Ant 97].

par rapport aux autres. De plus, l'ordre d'insertion des pièces dans l'algorithme (par surfaces décroissantes) peut conduire à des placements largement sous-optimaux. La translation de colonnes n'est donc intéressante que comme base de développement, en y insérant des heuristiques d'ordre plus efficaces.

3.2 Modules rectangulaires

La méthode des modules rectangulaires proposée par Delaporte dans [Del 90] vise essentiellement à contourner les problèmes d'encastrement dus aux formes concaves. Son principe consiste à inscrire un petit nombre de pièces quelconques dans un englobant rectangulaire. Le placement de ces pièces les unes par rapport aux autres est réalisé à la main, ou par un algorithme complet. Chaque groupe de pièces sera appelé un *module de placement.* La construction automatique des modules optimums peut être réalisée par un algorithme énumératif complet puisque le nombre de pièces à placer est réduit.

À partir de la liste des modules de placement rectangulaires, il devient aisé d'utiliser un algorithme plus simple pour l'agencement des différents modules sur le coupon. La méthode préconisée par l'auteur consiste à suivre le schéma *bottom-left* [CGJ 84]. La figure 7.6 présente un exemple de placement de ce type.

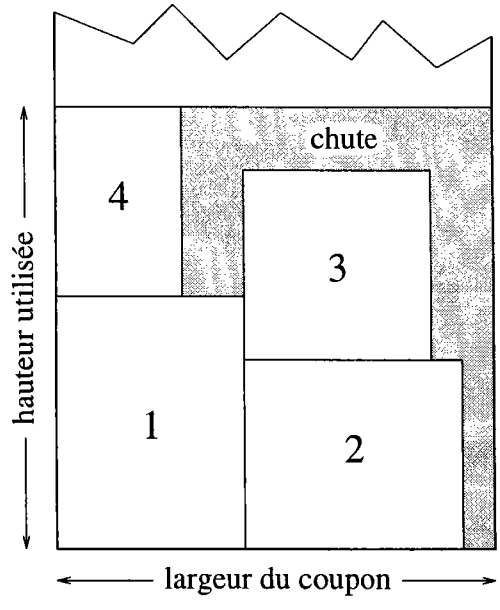


Figure 7.6 : Exemple de placement par la méthode *bottom-left.*

Chaque rectangle est placé le plus bas possible sur la surface de placement, et le plus à gauche. Dans ce type de méthodes, c'est l'ordre d'ajout des modules de placement qui déterminera la qualité du résultat (en terme de longueur de tissu utilisée). Coffman, Garey et Johnson ont proposé plusieurs types d'ordre pour le placement *bottom-left*:

- *increasing width* (*BTIW*): les rectangles sont classés par ordre croissant de leur largeur ;
- *increasing height* (*BTIH*): les rectangles sont classés par ordre croissant de leur hauteur ;

- *decreasing width* (*BTDW*) : les rectangles sont classés par ordre décroissant de leur largeur ;
- *decreasing height* (*BTDH*) : les rectangles sont classés par ordre décroissant de leur hauteur ;

Les ordres proposés ont des performances inégales en fonction du problème à résoudre, si bien que les auteurs n'on pu dégager une heuristique plus performante en général. De plus, d'un point de vue théorique Baker, Coffman et Rivest ont montré qu'il n'existait pas toujours d'ordre optimal pour la méthode *bottom-left* [BCR 80].

Conclusion

Le problème du bordereau de coupe en confection constituera le sujet d'étude principal de la partie applicative de ce document. Nous l'avons décrit dans ses grandes lignes, en le situant tout d'abord dans le processus de conception et de fabrication de vêtements dont les répercussions économiques sont non négligeables. La définition d'un problème de bordereau de coupe fait intervenir un certain nombre de contraintes issues de l'industrialisation que nous avons regroupées dans trois catégories : les contraintes d'orientation, de motif et de fabrication.

Sur le plan algorithmique, le calcul d'un bordereau de coupe pose un certain nombre de difficultés qui influenceront la conception des méthodes de résolution. Ainsi, nous avons mis en évidence la taille importante des domaines de définition qui interdit l'utilisation de certains algorithmes énumératifs, et la diversité des formes à placer qui nécessite une gestion fine des encastrements entre les pièces concaves.

Un certain nombre de méthodes de résolution ont été conçues pour contourner ou résoudre les principales difficultés que nous avons évoquées. Nous avons présenté deux méthodes spécifiques qui restent proches du cadre des CSP. La méthode des translations de colonnes vise à respecter en priorité le sens du fil des tissus en opérant une série de translations sur des pièces ou des groupes de pièces afin d'obtenir un placement présentant un taux de chute acceptable. La méthode des modules rectangulaires divise le problème global en un ensemble de sous problèmes plus simples et de tailles réduites, puis combine les placements obtenus en un plan général.

Le problème du bordereau de coupe à été abordé dans différents domaines comme la recherche opérationnelle, voire des méthodes *ad hoc*. Toutefois, nous ne présentons pas toutes ces méthodes car elles s'éloignent trop du contexte CSP qui nous intéresse dans ce document.

Le problème que nous étudions ici n'a pas été abordé à notre connaissance dans le cadre de formalisation des CSP. S'il sort un peu du cadre habituel de la recherche d'une solution difficile à trouver – l'objectif est d'exhiber une solution qui minimise en général le taux de chute – le bordereau de coupe mérite toutefois d'être formalisé et résolu à l'aide de CSP. Certains algorithmes énumératifs développés dans ce cadre permettent des recherches efficaces pour des problèmes relevant du cadre générique du placement.

Chapitre 8

Utilisation des CSP pour le problème du bordereau de coupe

LE PROBLÈME du bordereau de coupe a été étudié dans différents cadres théoriques de résolution, mais n'a pas été exploré (du moins à notre connaissance) dans le cadre d'étude des CSP. Nous nous concentrons dans ce chapitre sur les problèmes spécifiques que pose le calcul d'un bordereau de coupe quant à l'utilisation de CSP.

Le premier paragraphe de ce chapitre proposera une formalisation CSP du problème du bordereau de coupe. Elle s'attardera notamment sur les difficultés liées à la discrétisation de l'espace de placement.

Le paragraphe 2 présentera les implications de la structure générale d'un problème de bordereau de coupe sur l'algorithmique séquentielle de résolution. Nous ne considérerons en fait que les algorithmes de recherche complets qui constituent le cœur de cette étude. Nous aborderons enfin au paragraphe 3, les problèmes liés à la parallélisation des algorithmes de recherche de solution pour le calcul d'un bordereau de coupe.

1 Formalisation CSP du problème

La première difficulté rencontrée dans la résolution d'un problème est sa formalisation. Nous proposons dans ce paragraphe de transcrire dans le formalisme CSP, le problème du bordereau de coupe. En fait, deux questions importantes seront traités dans cette optique, et feront l'objet de paragraphes distincts : les domaines de définition réels, et les objectifs d'optimisation. Nous donnerons de plus dans le paragraphe 1.3, un exemple de formalisation.

1.1 Discrétisation des domaines

Habituellement, le problème du bordereau de coupe consiste à placer des pièces d'un patron de vêtement dans $\mathbb{R}^2 \times 2\pi$, soit dans un espace infini, ou tout au moins continu, qui correspond à la surface du coupon de tissu (*cf.* paragraphe 2.1 du chapitre 7). Si certains cadres de travail en CSP considèrent des domaines infinis ou continus [Dav 87], la plupart des auteurs travaillent sur des CSP dont les domaines sont finis et discrets.

La formalisation « naturelle » du problème du bordereau de coupe consiste à utiliser une variable par forme à placer, une solution étant l'affectation d'une position pour chaque variable de telle sorte que chaque affectation respecte toutes les contraintes définies dans le problème. Les domaines des variables doivent donc exprimer la totalité des positions possibles sur la surface du coupon de tissu. Il sera ainsi difficile pour un algorithme complet d'énumérer un espace infini, tel qu'il est défini dans le chapitre 7.

Il est donc nécessaire de discrétiser l'espace de placement, de telle sorte que les domaines correspondants soient finis et discrets. La solution la plus simple consiste à quadriller le coupon de tissu, en considérant chaque carreau ainsi défini comme une position élémentaire d'un domaine. La figure 8.1 présente un exemple de discrétisation de la surface de placement par quadrillage.

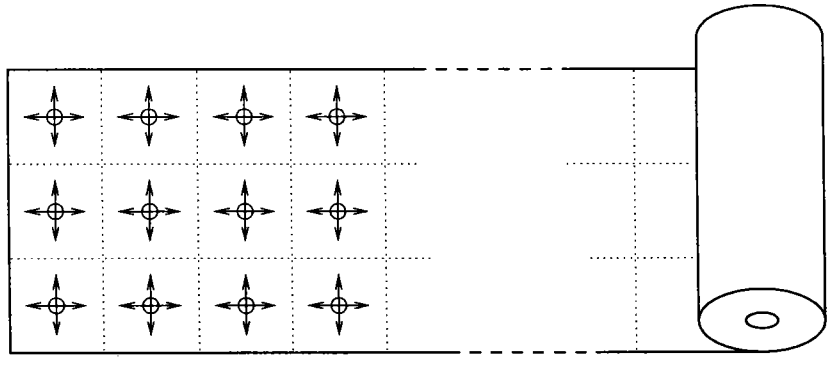


Figure 8.1 : Discrétisation de la surface de placement. les positions possibles sont matérialisées par des cercles. Les flèches correspondent aux différentes orientations possibles d'une pièce.

En plus de la surface de placement, il est indispensable de prendre en compte les différentes orientations possibles pour chaque position. Nous choisissons de ne prendre en compte que quatre orientations possibles pour une seule position. Ce choix est dicté par deux observations importantes :

1. les rotations d'un angle différent de 90° (ou d'un multiple) sur un quadrillage posent des problèmes pour les calculs de recouvrement ;

2. les pièces du patron sont généralement alignées sur la chaîne ou la trame du tissu, qui forment un angle droit entre elles.

La discrétisation de l'espace de placement induira des pertes de matière supplémentaires du fait de la discontinuïté du positionnement. Son effet peut être facilement mis en évidence en reportant le quadrillage de discrétisation sur les formes à placer. La figure 8.2 présente un exemple de discrétisation d'une pièce de vêtement.

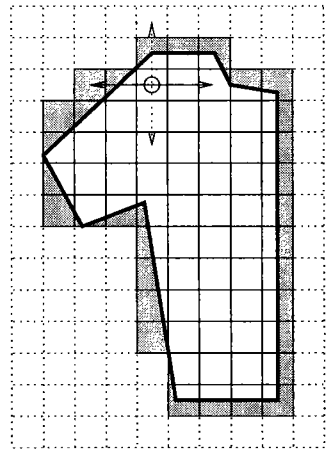


Figure 8.2 : Discrétisation d'une pièce de vêtement (en gras). Le placement se fera sur les carreaux englogant la pièce, en respectant le quadrillage de la surface de placement. Les pertes de matière dues à la discrétisation sont grisées.

Le choix de la taille du quadrillage de discrétisation de l'espace de placement a des effets directs importants sur la recherche. En effet, un quadrillage fin conduit à la définition de domaines de taille importante qui rendra la recherche plus coûteuse. À l'opposé, un quadrillage grossier conduira à des solutions de moins bonne qualité puisque les pertes inhérentes à la discrétisation seront plus grandes. Le compromis sera choisi par l'utilisateur qui formalisera un problème donné, en fonction des formes des pièces à placer, et du temps disponible pour calculer une solution. De plus, il sera utile de tenir compte des contraintes d'alignement des motifs pour que ceux-ci soient des multiples de la taille de quadrillage. On évitera ainsi les pertes de matières dues à la discrétisation des alignements.

1.2 Définition des objectifs

La complexité opérationnelle des algorithmes de résolution du problème du bordereau de coupe conduit généralement à réduire les exigences vis-à-vis de la solution obtenue. En effet, on ne pourra pas toujours requérir l'optimal parfait en un temps raisonnable. Ces limitations seront encore plus évidentes dans le cadre restreint des algorithmes complets que nous étudions dans ce document.

Nous proposons deux hypothèses de travail différentes, qui visent à chercher un compromis entre le temps de résolution et la qualité de la solution obtenue : la résolution en espace limité, et la résolution en temps limité.

1.2.1 Résolution en espace limité

Dans ce contexte de recherche, l'espace de placement est limité (en longueur de tissu) et l'objectif général de l'algorithme sera alors de trouver un placement correct dans cet espace. Dans le cas où un placement est trouvé, il est possible de faire une nouvelle recherche avec un espace encore plus réduit, et ce, jusqu'à ce que l'algorithme conclut à une impossibilité de placement. On procédera en sens inverse si la première recherche ne permet pas de trouver de solution.

La technique incrémentale (ou décrémentale selon le cas) reste toutefois coûteuse car elle suppose d'approcher l'optimal, et donc de parcourir l'ensemble de l'arbre de recherche. Pour des problèmes de très grande taille, on se contentera de trouver une solution dans un espace limité qui correspond aux critères de pertes acceptables pour l'industrialisation de la fabrication.

1.2.2 Résolution en temps limité

Dans la mesure où l'on désire obtenir une solution rapidement, on peut choisir de limiter le temps de recherche. L'objectif de l'algorithme est alors de rechercher le placement optimal selon la longueur de tissu utilisée. Toutefois, l'optimum ne sera atteint que si le temps imparti est suffisant pour que l'algorithme explore l'arbre de recherche qui y conduit. Dans le cas contraire, l'algorithme sera arrêté sur une solution sous-optimale.

1.3 Exemple de formalisation

Les principales contraintes dans un problème de bordereau de coupe concernent le non-recouvrement des pièces, c.-à-d. des contours. Comme le positionnement est discret, selon un cadrillage, les seuls recouvrements qui peuvent être rencontrés lors de la vérification de contraintes concernent des carreaux d'englobement de pièces. Il n'est donc pas utile de stocker les contours exacts des pièces, mais uniquement les carreaux qu'elles occupent. Les formes à placer seront donc stockées par :

- Une matrice de bits qui matérialise l'englobant de la pièce. Les matrices pourront avoir des tailles différentes.
- Le centre de placement de la pièce qui correspond au carreau de la matrice qui servira d'origine pour le placement.

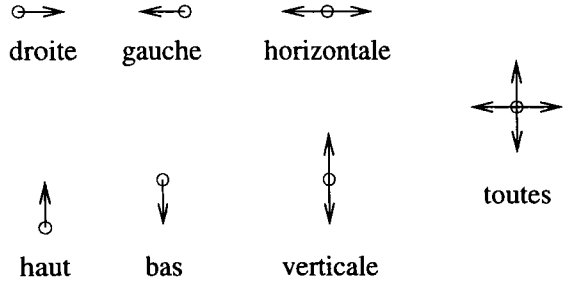


Figure 8.3 : Orientations possibles pour une pièce.

- Les orientations autorisées pour la pièce. Celles-ci suivront les cas de combinaison présentées à la figure 8.3.

Les contraintes de non-recouvrement sont implicites et ne nécessitent donc pas de structures de données particulieres. Les contraintes d'alignement sur les motifs sont propres à chaque pièce et pourraient donc faire partie de ses caractéristiques. Nous n'en tenons toutefois pas compte dans le système de résolution que nous utilisons.

```
placer
    forme 8, 12 : @ 4.2, orientation verticale,
          " . # # # # # # # # # # # #",
          " . # # # # # # # # # # # #",
          " # # # # # # # # # # # # #",
          " # # # # # # # # # # # # #",
          " # # # # # # # # # # . . .",
          " . # # # # # . . . . . .",
          " . # # # # # . . . . . .",
          " . . # # # . . . . . . .";
    forme 8, 12 : @ 4.2, orientation verticale,
          " . # # # # # # # # # # # #",
          " . # # # # # # # # # # # #",
          " # # # # # # # # # # # # #",
          " # # # # # # # # # # # # #",
          " # # # # # # # # # # . . .",
          " . # # # # # . . . . . .",
          " . # # # # # . . . . . .",
          " . . # # # . . . . . . .";
sur 13, 13.
```

Figure 8.4 : Exemple de description d'un problème de bordereau de coupe selon le formalisme CSP que nous utilisons. Le patron est constitué de deux pièces dont les contours et l'orientation correspondent à la figure 8.2. La surface de placement (13×13) permet un placement optimal.

2 Algorithmes séquentiels

La résolution d'un problème de bordereau de coupe peut être menée à bien de multiples façons. Nous nous concentrons dans ce paragraphe, sur les algorithmes complets séquentiels. De plus, si tous les algorithmes que nous avons présentés dans la première partie de ce document sont théoriquement applicables ici, tous n'exhiberont pas les mêmes facilités de résolution. Nous étudierons donc tout particulièrement les techniques qui s'adaptent le mieux à la spécificité du problème.

Nous montrons tout d'abord l'intérêt du filtrage de certaines contraintes *avant* la recherche énumérative (paragraphe 2.1), puis nous discuterons le choix de la méthode de recherche elle-même au paragraphe 2.2. Nous aborderons enfin l'influence des ordres d'instanciation dans le paragraphe 2.3.

2.1 Pré-filtrage des problèmes

Si le filtrage des domaines avant la recherche est souvent omis dans le cadre des algorithmes généraux de résolution des CSP, il sera important de ne pas négliger cette

phase pour les problèmes de bordereau de coupe. En effet, la définition de tels problèmes comporte un nombre important de contraintes unaires qui peuvent être traitées (par filtrage des domaines correspondants) avant la résolution. Nous passons ici en revue les différents types de contrainte rencontrées dans de tels problèmes, afin de déterminer l'opportunité de leur filtrage.

Contraintes de non-recouvrement : ces contraintes sont définies implicitement dans la déclaration d'un problème de bordereau de coupe. On peut en fait considérer le non-recouvrement entre les différentes pièces et l'extérieur du coupon (pour que les pièces ne « sortent » pas de l'espace de placement), et le non-recouvrement direct entre les pièces, afin que celles-ci ne se chevauchent pas.

Dans le premier cas, les contraintes sont unaires puisqu'elles concernent indépendamment chaque pièce, et un filtrage par suppression des positions qui entraînent le débordement d'une pièce est vivement conseillé. En effet, ces contraintes seront appliquées avant la recherche, et l'algorithme de résolution n'aura plus à en tenir compte.

Pour ce qui est des contraintes de non-recouvrement entre deux pièces, un filtrage des domaines les prenant en compte[1] ne pourra entraîner de suppression – et donc de réduction du temps de résolution – que si la position d'une pièce interdit toutes les positions pour une autre. Ce cas de figure ne sera rencontré que si l'on cherche à placer des pièces très grosses dans un espace restreint. Or, l'hypothèse de départ suppose que le problème comporte un nombre suffisament important de pièces, et nous permet d'exclure ces cas. Ainsi, on peut prévoir à priori que le filtrage par arc-consistance des contraintes de non-recouvrement ne permettra en général pas de réduction de domaine, alors qu'il sera très coûteux à cause du nombre de positions à tester mutuellement. Nos algorithmes de recherche n'incluront donc pas ce type de technique.

Contraintes d'orientation : elles signifient que la pièce devra respecter le sens d'un fil. Elles ne s'appliquent qu'à une seule pièce et pourront donc être traitées par pré-filtrage. Les contraintes d'alignement avec les motifs du tissu entrent dans cette catégorie et seront donc traitées de la même façon. On notera toutefois, que les algorithmes que nous avons implémentés n'incluent pas ce type de contraintes.

2.2 Consistance en avant et recherche

Le choix de la consistance à établir au cours de la recherche est une question fondamentale dans la définition de l'algorithme à mettre en œuvre. Nous proposons dans ce paragraphe, un algorithme qui permet de tirer parti au mieux des spécificités du problème du bordereau de coupe.

En considérant les résultats expérimentaux obtenus sur les CSP généraux [Pro 94, Smi 94], les algorithmes de recherche par établissement de consistance en avant semblent s'imposer. Si nous ne remettons pas en cause l'opportunité d'un tel algorithme, les particularités du problème qui nous concerne ici – voir les paragraphes suivants – induisent des choix spécifiques.

[1] ce filtrage correspond alors à de l'*arc-consistance*.

2.2.1 Notion de *non-support*

Si l'algorithme *FC* qui filtre les domaines futurs en fonction de chaque nouvelle instanciation, a longtemps fait l'unanimité dans le monde des CSP, il est maintenant sérieusement concurrencé par *MAC*[2] pour la résolution des CSP généraux. L'idée qui ressort de *MAC* est que la suppression d'une valeur inconsistante par *FC* peut entraîner la suppression de plusieurs autres valeurs si celle-ci était leur support[3] unique.

Dans le cas du problème du bordereau de coupe, le positionnement (*i.e.* l'instanciation) d'une pièce ne permet pas de supporter d'autres positions, mais conduit au contraire, à l'invalidation d'un certain nombre de positions. La suppression d'une position dans un domaine, consécutive au filtrage de *FC* conduit donc à un relachement des contraintes sur le CSP restant. On pourra ainsi parler de *non-support* puisque les contraintes se comportent en opposition avec la notion classique de support.

Ainsi, l'application d'une consistance en avant telle que *MAC* conduira à un surcroît de calcul, pour un bénéfice nul puisqu'aucune suppression ne sera propagée. On abandonnera donc l'algorithme *MAC* au profit de *FC* pour le problème du bordereau de coupe.

2.2.2 Localité des inconsistances

Lors de l'instanciation d'une variable, c.-à-d. le positionnement d'une pièce sur le coupon, les inconsistances qui doivent être filtrées par l'algorithme de recherche présentent un caractère local. La figure 8.5 permet d'illustrer cette propriété

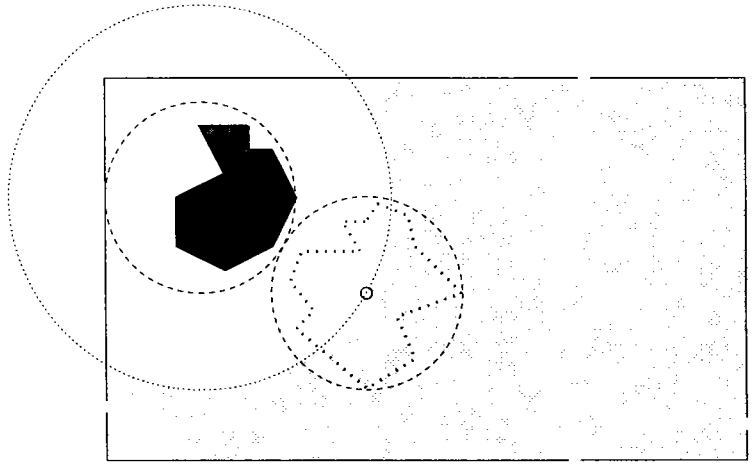


Figure 8.5 : Illustration de la localité des inconsistances.

Le placement d'une pièce (en gris foncé) conduit à filtrer les domaines des pièces en attente de placement (en pointillés gras), afin de supprimer les positions qui sont interdites par cette instanciation. Or, la pièce placée est inscrite dans un cercle de rayon α (en tirets), et le point de placement de la pièce à tester, elle même inscrite dans un cercle de rayon β, pourra être placé dans toutes les positions en dehors du cercle de rayon $\alpha + \beta$ (en pointillés) sans nécessiter de vérification non-recouvrement des contours. Ce cercle sera

[2]en plus du filtrage de *FC*, *MAC* établi l'arc-consistance entre toutes les variables non instanciées, et constitue donc une version « plus forte » de consistance en avant.

[3]la notion de support est explicitée au paragraphe 4 du chapitre 1.

appelé *cercle d'influence* de la position de la pièce. Si la surface du coupon est grande par rapport aux pièces à placer, l'algorithme de filtrage aura alors intérêt à ne filtrer que les positions des domaines inscrites dans les cercles d'influence relative d'une position.

2.2.3 Filtrages hors contraintes

Il est possible de simplifier le filtrage des domaines en avant en appliquant directement la projection de la nouvelle pièce instanciée sur les domaines futurs. En effet, son contour recouvre un certain nombre de positions, indépendamment des contours et orientations des autres pièces. Dans ce cas, le filtrage d'un domaine se décomposera en deux parties :

1. La suppression des positions qui sont directement recouvertes par la pièce instanciée. De plus, toutes les orientations de la position sont supprimées. Ce traitement sera peu coûteux puisqu'il ne prend en compte qu'un seul contour.

2. Le filtrage complet de toutes les positions restantes dans le cercle d'influence relative de la pièce instanciée.

On notera, que ce système suppose que le centre de rotation des pièces à placer (petits cercles sur la figure 8.5) est inclu dans le contour de la pièce. Si cette hypothèse n'est pas vérifiée, une translation devra être effectuée avant la résolution.

2.3 Ordres dans la recherhe

Les ordres d'instanciation revètent une importance particulière dans les algorithmes de résolution de CSP car ils seront un facteur déterminant pour le temps de résolution. Nous discuterons ici des heuristiques d'ordre pour la résolution du problème du bordereau de coupe.

2.3.1 Ordre sur les pièces à placer

Les méthodes classiques de résolution du problème du bordereau de coupe [Ant 97] ordonnent généralement les pièces à placer selon leur taille décroissante. Le but de cette heuristique est de placer en premier les pièces les plus encombrantes. De leur coté, les algorithmes généraux sur les CSP utilisent principalement l'heuristique *MRV* [BvR 95] qui instancie d'abord les variables dont les domaines sont les plus petits. L'objectif est de réduire la largeur de l'arbre de recherche.

En fait, ces deux méthodes conduiront dans le cadre qui nous intéresse ici, à des comportements sensiblement équivalents avec un algorithme de recherche par consistance avant : les plus grandes pièces pourront occuper moins de positions différentes à cause des limites du coupon de tissu, et leur domaine sera ainsi plus petit.

On pourra de même considérer que les pièces qui disposent d'un degré de liberté plus faible en rotation devraient être placées en premier. Là encore, l'heuristique *MRV* couvre ce cas de figure puisque les pièces en question auront un domaine réduit.

2.3.2 Ordre sur les positions testées

Si l'ordre d'instanciation des valeurs est souvent considéré comme secondaire dans les algorithmes généraux de résolution des CSP, la spécificité du problème du bordereau de coupe rend l'ordre horizontal prépondérant. En effet, l'objectif du placement est de minimiser la longueur de tissu utilisée, et un ordre quelconque conduirait à placer les pièces sans cette optimisation. D'une manière générale, les algorithmes de résolution devront parcourir les positions de manière à « remplir » au mieux le coupon, comme dans la figure 8.6.

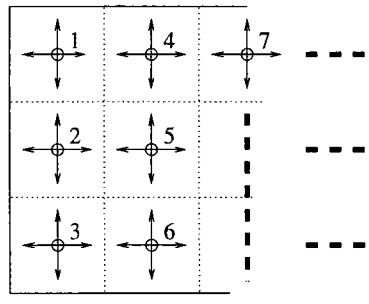


Figure 8.6 : Ordre de parcours des positions. Pour chaque point de placement, les différentes orientations sont parcourues dans un ordre quelconque.

3 Algorithmes parallèles

Du fait du coût élevé de la résolution d'un tel problème, on sera naturellement intéressé par une recherche parallèle. La plupart des méthodes que nous avons proposées dans la deuxième partie de ce document seront utilisables dans ce cadre. Nous ne nous concentrerons que sur deux points importants pour la parallélisation : la taille des contextes de recherche au paragraphe 3.1, et la recherche de la solution optimale (paragraphe 3.2).

3.1 Taille des contextes de recherche

Une caractéristique fondamentale du problème du bordereau de coupe est que les domaines des variables à instancier ont des tailles sensiblement plus importantes que les CSP généralement expérimentés. Cette situation conduit à la construction de grands contextes de recherche qui devront être échangés et mis à jour par les processeurs lors d'une phase d'équilibrage de charge. Ainsi, la limitation du nombre de changement de tâche deviendra encore plus nécessaire. Différents points seront donc à prendre en compte :

- *La fonction de divisibilité* d'une tâche qui permettra d'éviter des divisions en bas de l'arbre.
- *La stratégie de division* qui devra limiter la multiplication des contextes. Dans ce cadre, la division minimale (*cf.* chapitre 5, paragraphe 3.5) semble promettre de performances intéressantes.
- *La compression* des contextes transmis qui deviendra d'autant plus indispensable que les communications seront coûteuses.

3.2 Recherche de l'optimum

Comme on l'a vu au paragraphe 1.2, l'objectif secondaire du placement peut être d'optimiser la longueur de tissu utilisée. On sort légèrement du cadre classique des CSP, mais la différence est toutefois minime pour un problème d'optimisation. En effet, la longueur effective du coupon constitue une contrainte supplémentaire unaire qui pourra être vérifiée à chaque modification. Au début de la résolution, la longueur optimale correspond à la longueur du coupon, et le filtrage avant la résolution permet d'exclure toutes les positions qui conduisent à un dépassement (paragraphe 2.1).

Quand un processeur trouve une nouvelle solution qui améliore l'optimum, il transmet la nouvelle borne à tous les processeurs (par un mécanisme de *broadcast* par exemple). Les processeurs qui reçoivent une nouvelle borne devront tout d'abord vérifier que leur tâche en cours ne dépasse pas le nouvel optimum, puis recommenceront le filtrage pour supprimer les positions qui conduisent à un dépassement de la borne reçue.

Ce mécanisme implémente une forme de *branch and bound* élémentaire : les étapes de branchement se succèdent sans calcul de la fonction d'évaluation puisqu'elle est toujours inférieure à l'optimum global. En cas de changement d'optimum, les contextes de recherches sont mis à jour afin qu'une nouvelle étape de branchement maintienne toujours la fonction d'évaluation en deçà de l'optimum global.

Dans le cadre de la recherche d'un optimum, on pourrait contester la validité de la répartition de charge algébrique (chapitre 5, paragraphe 3.3). En effet, celle-ci construit les tâches à explorer en suivant la procédure *generate and test*. Ainsi, toutes les positions de la première pièce constitueront une base de recherche, induisant une couverture totale de l'espace disponible, et donc des débuts de recherche en dehors de la surface optimale. Toutefois, les premières positions testées permettront de trouver une solution constituant un optimum local qui éliminera les tâches inutiles.

Conclusion

Les difficultés liées à la résolution des problèmes de bordereau de coupe par des CSP ont été abordés dans ce chapitre. La premier obstacle a été naturellement la formalisation du problème. Il est notamment apparu que les algorithmes classiques énumératifs imposaient une discrétisation de l'espace de placement selon un maillage carré. Les objectifs visés par l'utilisateur – en général l'optimisation de la longueur de tissu nécessaire – induisant des coûts de recherche importants, nous a conduit à définir deux systèmes de résolution : en espace ou en temps restreint.

La recherche en espace restreint consiste à imposer une surface de placement réduite, proche d'un optimal qui correspond aux taux de chute habituellement observés. La résolution se concentre alors sur la recherche d'une solution. Dans la recherche en temps limité, l'espace de placement est suffisant pour permettre un grand nombre de solutions sous-optimales. À chaque découverte d'une solution, on réduit l'espace de placement pour ne chercher que des solutions meilleures.

Nous avons proposé pour la mise en œuvre de la recherche énumérative, différentes techniques spécifiques. Le préfiltrage des contraintes unaires (orientation des pièces par rapport à la trame du tissu, inscription dans l'espace de placement) est apparu comme

indispensable. À l'opposé, l'établissement et le maintien de l'arc-consistance ne pourra pas apporter les bénéfices classiquement observés. En effet, la suppression d'une position possible pour une pièce n'entrainera jamais la suppression d'autres positions pour d'autres pièces. Ainsi, la vérification de l'arc-consistance coûterait en temps de calcul, pour un bénéfice nul.

L'algorithme *FC* (avec ou sans hybridation avec un retour arrière non-chronologique) ne devra pas être utilisé tel quel. Nous avons mis en évidence des phénomènes de localité dans les filtrages en avant qui permettent de restreindre la vérification aux seules positions dans les domaines non instanciés qui sont proches de la position en cours de test.

Chapitre 9

Expérimentations séquentielles et parallèles

NOUS AVONS PRÉSENTÉ dans le chapitre 8 des concepts algorithmiques pour la résolution de problèmes de bordereau de coupe à l'aide du formalisme CSP. Nous nous proposons dans ce chapitre, d'expérimenter ces techniques, tant dans un cadre d'exécution séquentiel que parallèle.

Dans le paragraphe 1, nous présentons l'environnement expérimental de cette étude. Nous détaillerons notamment le choix des problèmes résolus, ainsi que les algorithmes et configurations machines utilisés. Le paragraphe 2 donnera un certain nombre de résultats obtenus sur une machine séquentielle – en fait, il s'agit d'une machine parallèle, mais dont seul un processeur est utilisé.

Nous terminons ce chapitre par des rapports d'expérimentations sur les recherches en parallèle (paragraphe 3). Aussi bien dans ce dernier cadre que dans l'environnement séquentiel, nous testerons des algorithmes en espace limité, avec incrémentation de l'espace de placement, et une expérimentation en recherche d'optimum.

1 Environnement expérimental

Avant de présenter les résultats obtenus lors de nos expérimentations, nous donnons quelques précisions sur l'environnement expérimental : les problèmes résolus dans le paragraphe 1.1, puis les algorithmes utilisés au paragraphe 1.2.

1.1 Problèmes résolus

Considérant le temps nécessaire pour résoudre un seul problème, il ne nous a pas été possible de réaliser des expérimentations sur un grand nombre de problèmes, afin d'obtenir des résultats statistiques. Nous nous sommes contentés de construire « à la main » un problème réaliste, en nous inspirant d'un patron de découpe déjà construit pour les formes des pièces.

Le problème ainsi construit contient 16 pièces de tailles et formes différentes. En fait, la forme joue un rôle peu important vis-à-vis de la taille, et surtout de la discrétisation de l'espace de placement – et donc des pièces du patron. De plus, les pièces demandent toutes la même orientation. À titre indicatif, la table 9.1 résume les tailles des matrices des différentes pièces, ainsi que la densité de remplissage de chaque matrice.

Table 9.1 : Liste des pièces à placer, avec la taille de leurs matrices, et le taux de remplissage de la pièce par rapport à sa matrice rectangulaire.

pièce	lignes	colonnes	densité	pièce	lignes	colonnes	densité
1	8	10	62%	9	7	9	56%
2	7	8	57%	10	6	5	60%
3	8	9	57%	11	11	8	55%
4	4	4	75%	12	8	7	64%
5	3	6	67%	13	6	9	70%
6	8	7	57%	14	5	9	51%
7	7	6	67%	15	8	9	57%
8	6	8	75%	16	8	6	67%

La largeur du coupon de tissu mesure 21 unités de discrétisation. Un arrangement manuel permet de placer toutes les pièces sur une longueur de tissu de 36 unités. Cette optimisation ne tient toutefois pas compte des pertes dues à la discrétisation des pièces.

On notera que les points de positionnement des pièces sont tous contenus dans les contours, mais ils ne sont pas forcément centrés par rapport à celui-ci. Ainsi, le rayon d'influence de la pièce n'est pas optimisé pour limiter la surface considérée lors d'une vérification de consistance en avant. On considère en fait que les décalages par rapport au centre de la pièce permettent d'inclure le surcoût de calcul induit par les contraintes d'alignement sur les motifs.

1.2 Algorithmes et tests effectués

Tous les algorithmes que nous avons utilisés sont implémentés en langage C. Un problème est stocké par sa liste de pièces, et chaque pièce nécessite une matrice de bits pour

le stockage des contours. De plus, les domaines sont stockés sous forme de matrices de pointeurs, comme dans la figure 9.1 afin de permettre un accès rapide à chaque position, tant sous forme matricielle pour les projections sur les surfaces, que sous forme de liste ordonnée pour les parcours d'instanciation.

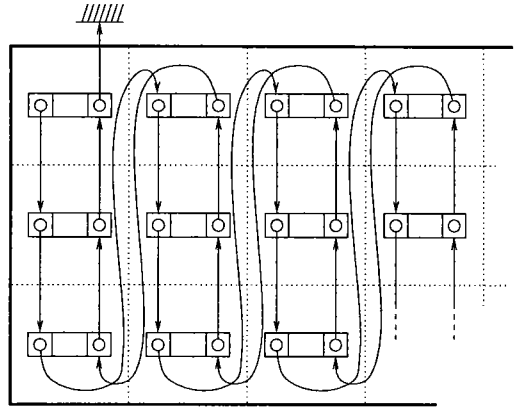


Figure 9.1 : Schéma de stockage des domaines des pièces à placer.

Pour les expériences en espace limité, nous avons testé successivement 8 problèmes, avec à chaque fois les mêmes pièces, mais avec des longueurs croissantes de tissu : de 28 à 42 unités, avec un incrément de deux unités entre chaque exécution. Ce système ne correspond pas tout à fait à la procédure pratique de résolution, mais permet de cerner les influences de la variation de la taille du coupon sur le temps de calcul.

Les expériences d'optimisation ont été menées avec les seizes pièces définies dans le paragraphe 1.1, et pour un espace limite de 21×40 unités.

La résolution des problèmes de bordereau de coupe se fait à l'aide de la plupart des optimisations présentées au chapitre 8 :

- préfiltrage des contraintes de non-recouvrement avec le contour du coupon de tissu, ainsi que des contraintes d'orientation ;
- vérification de consistance en avant optimisée avec projection de la nouvelle surface instanciée sur les domaines futurs et limitation du parcours des positions futures aux surfaces d'influence relative ;
- ordonnancement de l'instanciation des pièces par l'heuristique *MRV*.

Aussi bien les expérimentations séquentielles que parallèles ont été menées sur la Connection Machine CM-5 du CNCPST (Centre National de Calcul Parallèle en Sciences de la Terre). Toutefois, les algorithmes séquentiels ne sont pas des algorithmes parallèles sur un seul processeur avec des équilibrages triviaux, mais des versions séquentielles dépourvues de toutes les surcharges dues au parallélisme. Le choix de la CM-5 pour les expérimentations séquentielles permet essentiellement de calculer les accélérations parallèles.

2 Expérimentations séquentielles

La première partie des tests effectués concerne la recherche séquentielle. Si elle sort un peu du cadre d'étude de ce document, elle reste toutefois importante car elle per-

mettra par la suite d'évaluer les performances des résolutions paralléles. Nous étudions deux méthodes différentes de résolution : la recherche d'une solution en espace limité (paragraphe 2.1), et la recherche de la solution optimale dans la paragraphe 2.2.

2.1 Recherche en espace limité

Les temps de calcul pour la résolution du problème test en séquentiel dans un espace limité sont résumés par la figure 9.2. L'algorithme a découvert une solution pour les 3 dernières longueurs de tissus, c.-à-d. à partir de 38 unités. Par rapport à l'optimal calculé à la main sur un espace continu (36 unités), une perte de matière est observée, due à la discrétisation de l'espace de placement.

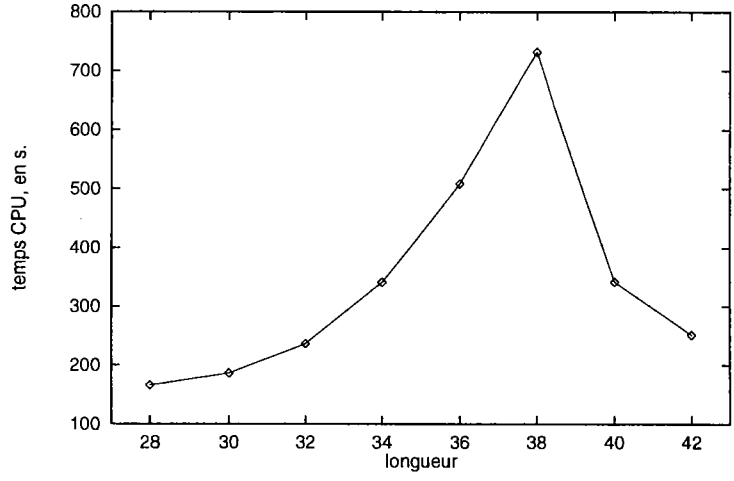


Figure 9.2 : Temps de résolution du problème du paragraphe 1.1 en séquentiel, pour des espaces limités de taille croissante.

La courbe des temps en fonction de l'espace disponible fait apparaître un pic de calcul qui correspond à la longueur minimale pour laquelle une solution a été trouvée. En deçà de cette valeur, le coût de la recherche va croissant avec l'augmentation de la taille disponible. La principale explication de ce phénomène est que la taille des domaines augmente, induisant un arbre de recherche de largeur croissante. De plus, comme les problèmes ne sont pas satisfiables, l'algorithme doit parcourir la totalité de l'arbre.

Au delà du seuil de satisfiabilité de la longueur de tissu, le temps de calcul va décroissant avec l'augmentation de la longueur disponible. Là, l'accroissement de la taille des domaines induit aussi des arbres de plus en plus larges, mais s'accompagne de plus d'une nette augmentation du nombre de solutions (sous-optimales pour la plupart). La chute du coût de résolution vient du fait que l'algorithme rencontre de moins en moins de difficultés pour trouver une solution. Dans ce cas, il n'a pas à parcourir la totalité de l'arbre.

2.2 Recherche de l'optimum

L'observation des résultats de la résolution du problème test en séquentiel pour la recherche de la solution optimale ne permettra pas beaucoup de commentaires. En effet, un seul chiffre sera disponible, puisqu'une seule résolution aura été exécutée. Les données brutes sont de 742.55 secondes pour trouver l'optimal. La solution trouvée est identique à celle trouvée par l'algorithme en espace limité à 38 unités dans le paragraphe précédent.

Ceci nous permet donc essentiellement un commentaire sur cette dernière méthode. En effet, le temps nécessaire pour trouver cette solution en espace constant, c.-à-d. en stoppant la recherche après la découverte, à mis un temps équivalent (731 secondes), alors que le calcul de l'optimum impose de parcourir totalement l'arbre sur la profondeur de la solution optimale. Ainsi, on déduit que la solution trouvée en espace limité à 38 unités a imposé le parcours d'une grande partie de l'arbre de recherche.

3 Expérimentations parallèles

La dernière partie de ce chapitre concerne l'expérimentation des algorithmes parallèles. Elle reprend la même structure que le paragraphe présentant les résultats séquentiels : recherche en espace limité dans le paragraphe 3.1 et recherche de l'optimum dans le paragraphe 3.2.

3.1 Recherche en espace limité

La série de tests avec espace limité croissant a tout d'abord été exécutée sur des nombres de processeurs différents. La figure 9.3 présente ces résultats. On notera que la division des tâches se fait selon l'heuristique minimale.

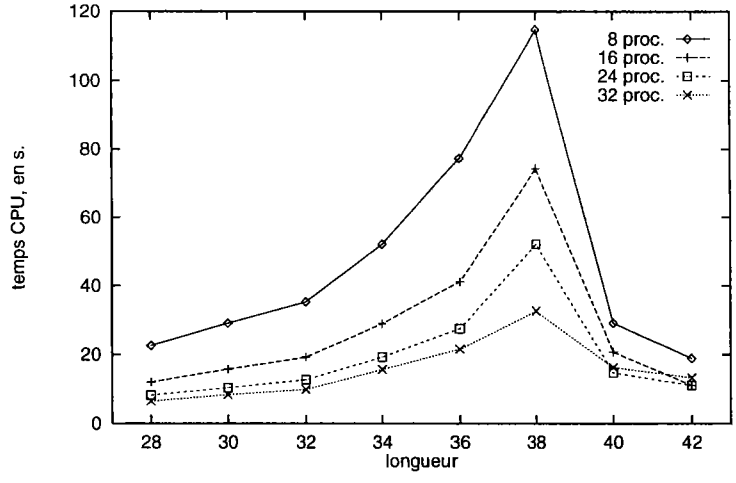


Figure 9.3 : Temps de résolution du problème du paragraphe 1.1 avec 8, 16, 24 et 32 processeurs. Les espaces sont limités, de taille croissante.

Globalement, les courbes des temps expriment le même comportement que la résolution séquentielle : le coût de résolution augmente vers le seuil de satisfiabilité de la longueur de tissu, puis diminue.

En fait, la zone croissante (non satisfiable) et la zone décroissante (satisfiable) font apparaître des comportements différents, en dehors des variations : dans la zone croissante, les rapports de temps de calcul entre les différents nombres de processeurs sont quasi constants, et les courbes sont régulières. À l'opposé, la zone décroissante fait apparaître plus de bruit entre les courbes, à tel point que la hiérarchie qui devrait ressortir entre les différents nombres de processeurs s'en trouve chamboulée. On expliquera ce phénomène par la répartition initiale des tâches de recherche qui conduit à placer la branche qui

contient une solution dans des positions différentes dans les listes d'attente selon le nombre de processeurs[1].

Afin de vérifier la validité de la stratégie de division minimale dans le cas du problème du bordereau de coupe, nous avons recommencé les tests avec deux autres heuristiques : la division haute et la division basse. La division transverse à été rejetée, car conduisant à de trop nombreux changements de contextes. Les résultats obtenus sont résumés dans la figure 9.4. Nous nous sommes concentrés ici sur la recherche limité avec 38 unités de longueur car c'est la résolution la plus coûteuse.

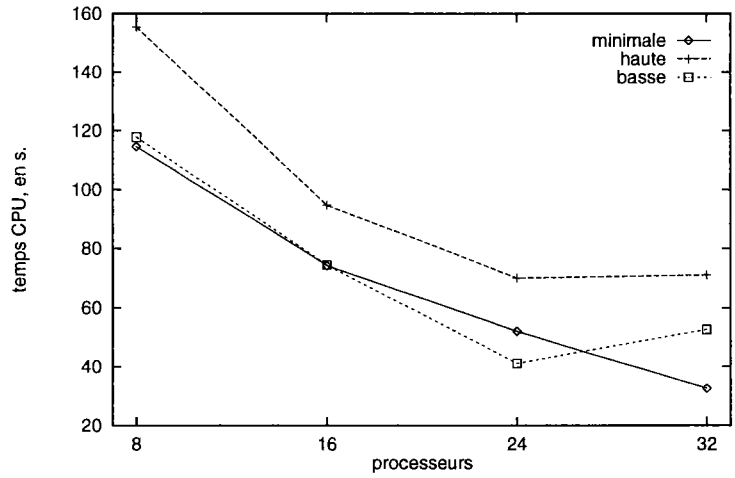


Figure 9.4 : Temps de résolution du problème du paragraphe 1.1 avec des techniques de division minimale, haute et basse. Les espaces sont limités à 21 × 38 unités.

En fait, les résultats sont moins significatifs que l'on pouvait le supposer puisque la différence entre la division minimale et la division basse n'est pas nette. La seconde se montre même sensiblement plus rapide pour 24 processeurs. Cependant, la division haute reste peu efficace, conformément aux observations faites dans le chapitre 6. La table 9.2 apporte quelques éléments d'explications.

Table 9.2 : Nombre de tâches échangées/divisées selon les différentes techniques de division, pour un nombre donné de processeur.

Nombre de processeurs	8	16	24	32
Division minimale	6/0	17/9	10/6	13/0
Division haute	20/15	49/23	12/4	32/18
Division basse	7/0	15/6	5/0	19/12

Le nombre de tâches échangées par les processeurs est réduit pour toutes les techniques de division. De plus, une proportion importante des ces tâches n'est pas issue d'une division, mais d'une simple transmission des tâches définies lors de la phase de répartition initiale. Or, comme les différences d'efficacité entre les heuristiques seront essentiellement

[1] rappelons que les problèmes satisfiables n'imposent pas le parcours de la totalité de l'arbre de recherche.

dues aux tâches divisées, il est difficile de distinguer une heuristique comme étant la meilleure.

En terme d'efficacité, la table 9.3 résume les performances de la recherche selon le nombre de processeurs et la taille de l'espace de placement. La division utilisée reprend l'heuristique minimale.

Table 9.3 : Efficacité de la résolution parallèle du problème du paragraphe 1.1 avec 8, 16, 24 et 32 processeurs pour la recherche en espace limité.

Nombre de processeurs	8	16	24	32
28 unités	92%	86%	83%	79%
30 unités	80%	74%	74%	69%
32 unités	84%	77%	77%	74%
34 unités	82%	74%	74%	68%
36 unités	82%	77%	77%	74%
38 unités	**80%**	**61%**	**58%**	**70%**
40 unités	147%	103%	97%	66%
42 unités	166%	141%	94%	59%

Les résultats sont globalement intéressants : les efficacités obtenues montrent que le parallélisme est correctement exploité lors du parcours de tout l'arbre de recherche (28 à 36 unités). On observe même des accélérations super-linéaires pour les problèmes très satisfiables. Toutefois, ces performances étaient prévisibles car le premier processeur qui trouve une solution arrête les autres, et la densité importante de solutions augmente les chances qu'un processeur aboutisse plus vite à une solution.

3.2 Recherche de l'optimum

Les temps de recherche de la solution optimale en fonction du nombre de processeurs sont donnés à la table 9.4. Nous les reportons essentiellement à titre indicatif car on ne peut en déduire de nombreux enseignements.

Table 9.4 : Performances de la résolution parallèle du problème du paragraphe 1.1 avec 8, 16, 24 et 32 processeurs pour la recherche de la solution optimale.

Nombre de processeurs	8	16	24	32
Temps, en s.	114.59	63.57	50.72	36.83
Accélération	6.48	11.68	14.64	20.16
Efficacité	81%	73%	61%	63%

On remarquera une efficacité correcte quelle que soit le nombre de processeurs utilisés. En fait, la diminution observée de l'efficacité est similaire à celle apparue dans le chapitre 6, ainsi que dans le paragraphe précédent. On l'attribuera donc pour sa plus grande part, à l'augmentation du coût des communications.

Conclusion

Nous avons expérimenté dans ce chapitre, quelques techniques de résolution du problème du bordereau de coupe dans une formalisation CSP. Les algorithmes utilisés reprennent les spécificités présentées dans le chapitre 8. Nous nous sommes de plus intéressés, autant au cadre d'exécution séquentiel qu'au cadre parallèle.

Les résolutions ont porté sur un seul problème construit à partir d'un exemple concret de patron de coupe, et avec des tailles de coupon de placement différentes. Les tests ont été exécutés sur une machine parallèle de type Connection Machine CM-5.

Les algorithmes en espace limité (avec une taille de tissu croissante) ont mis en évidence la difficulté de trouver une solution proche de l'optimum, tant pour une recherche séquentielle que parallèle. Les tailles de coupon qui permettent de nombreuses solutions exhibent des accélérations super-linéaires, alors que les tailles inférieures à l'optimum présentent des efficacités satisfaisantes, bien que sub-linéaires.

La comparaison de différents types de division n'a pu mettre en évidence la pertinence de la division minimale que nous avions supposée. La raison est essentiellement que le problème a fait apparaître un équilibre initial suffisant entre les processeurs pour que les phases de division soient marginales par rapport au reste de la résolution. On pourra supposer que ce cas est assez général pour tous les problèmes de bordereau de coupe puisque la grande taille des domaines permet une bonne répartition au départ du déséquilibre de l'arbre de recherche.

Les algorithmes en temps limité (avec optimisation) ont eux aussi montré des performances correctes sur le plan du parallélisme. Nous n'avons toutefois pas observé d'efficacités super-linéaires. On notera que les expérimentations se sont déroulées avec un temps illimité pour observer le temps maximal nécessaire à la recherche de l'optimum global.

Conclusion et perspectives

POUR CONCLURE un document de ce type, il est bon d'établir un bilan des avancées qui y sont proposées. De même que pour le plan général, on pourra les séparer en trois parties fondamentales : l'aspect séquentiel des CSP, l'aspect parallèle et l'application à un problème de CAO.

Sur le plan séquentiel, nous nous sommes concentrés sur la résolution des CSP n-aires. Dans ce contexte, il nous est apparu nécessaire de définir une méthode de recherche par consistance en avant car les algorithmes de la littérature (*FC* et *MAC* principalement) ont été conçus pour des contraintes binaires. Partant des notions de consistances n-aires proposées par différents auteurs – l'*hyper-k-consistance* [Jég 93] et la *relationnelle-k-consistance* [DvB 95] – nous avons montré l'inadéquation de la première pour la conception d'un algorithme de recherche en avant. De plus, nous avons étendu la seconde notion vers une définition plus générale de consistance : la relationnelle-(i,j)-consistance qui introduit la notion de profondeur de vérification[2]. Celle-ci nous a permis de concevoir un algorithme de recherche en avant de type *forward-checking* qui puisse prendre en compte les contraintes n-aires.

Suivant le même objectif, nous nous sommes penchés sur les adaptations de diverses techniques de recherche induites par la prise en compte de contraintes n-aires. Il est ainsi apparu que la technique du retardement de la vérification pouvait non seulement s'appliquer au différentes valeurs à l'intérieur d'un domaine, mais aussi sur plusieurs domaines car la mise en évidence d'un support suffit à valider une instanciation. De plus, nous avons défini une heuristique d'ordre sur les variables qui tente de mieux prendre en compte les contraintes n-aires, afin de minimiser la largeur de l'arbre de recherche.

Des expérimentations pratiques sont venues valider les algorithmes et heuristiques proposés. Nous avons pû mettre en évidence l'intérêt de la recherche de solution sur les contraintes n-aires par rapport à la résolution sur un système modifié pour ne contenir que des contraintes binaires. L'heuristique d'ordre sur les variables que nous avions définie s'est montrée tout aussi performante même vis-à-vis de *MRV* qui fait l'unanimité dans la communauté CSP pour la combinaison avec *FC*.

Tout le travail que nous avons fourni dans le cadre d'exécution parallèle s'articule autour d'un algorithme générique de résolution énumérative des CSP. Celui-ci est capable d'intégrer le schéma de Prosser [Pro 93b] qui permet d'hybrider différentes techniques de résolution séquentielles dans un système de résolution parallèle où les appels aux fonctionnalités parallèles sont eux aussi génériques. La recherche séquentielle étant basée sur le parcours d'un arbre, sa parallélisation consiste donc à paralléliser le parcours d'un arbre

[2] la profondeur de vérification correspond au nombre de variables non-instanciées dans une contrainte au moment de l'instanciation d'une variable.

d'autres systèmes.

Si cette étude se termine ici par ces quelques mots, elle laisse toutefois entrevoir un certain nombre de perspectives pour des recherches futures. Pour se contenter de l'axe principal que constitue la parallélisation des algorithmes énumératifs de résolution des CSP, on notera la flexibilité qu'apporte l'algorithme générique que nous avons proposé. Dans ce cadre, il sera possible d'intégrer dans la recheche parallèle, des méthodes de mémorisation binaire ou n-aire par apprentissage des inconsistances rencontrées [SV 93]. Les problèmes que cela soulève concernent essentiellement la portée globale des déductions qui pourrait nécessiter des transmissions au cours de la recherche. Dans cette optique, une reflexion sur les différentes options de transmission possibles, suivie d'une étude expérimentale approfondie serait intéressante. La parallélisation du marquage par *BM* ou *BC* serait aussi probablement à expérimenter. Cependant, ces techniques conduisent à des améliorations assez faibles dans un cadre séquentiel, si bien qu'ils semblent moins stratégiques dans un contexte d'exécution parallèle.

De même, il serait intéressant de définir des stratégies de transfert de charge différentes. Nous avons utilisé un mode de compression du contexte transmis par échange des suppressions dans les domaines. On peut toutefois imaginer des mécanismes qui prennent en compte des ratios plus précis entre le coût d'une vérification et celui de la transmission par exemple. De tels systèmes seraient cependant plus dépendants du domaine d'application. Un certain nombre d'autres expérimentations pourraient être menées afin de tester différents comportements des algorithmes proposés. L'évaluation de la fonction de détermination de la divisibilité pourrait notamment être plus poussée afin de tester les seuils de hauteur et de largeur avec de plus grandes variations.

Sur un plan plus théorique, il serait intéressant de valider ces algorithmes par des modèles mathématiques. Il existe des modèles pour celà: théorie des files d'attente, modèles matriciels ou probabilistes [Cyb 89, FG 96].

Les tests que nous avons menés dans cette étude ont été exécutés sur une Connection Machine CM-5. Cette architecture permet des communications point à point entre chaque paire de processeurs, et autorise plusieurs communications simultanées. Cependant, les architectures fortement couplées comme la CM-5 restent assez peu répandues par rapport aux réseaux de stations de travail. Or, ces architectures ne permettent pas de communications simultanées mais garantissent des communications en temps constant par rapport au nombre de processeurs dans le cas où une seule communication se produit. Il serait intéressant de mener les mêmes expériences dans ce cas de figure afin de vérifier si les comparaisons entre les techniques établies sur la CM-5 restent valides.

Comme nous l'avons abordé dans la troisième partie concernant le problème du bordereau de coupe, le formalisme CSP peut être associé à la recherche de la solution optimale. Les algorithmes énumératifs sous-jacents s'orientent alors plus vers le *branch-and-bound.* La fonction d'évaluation d'une solution constitue dans le cadre des CSP une contrainte supplémentaire – ou plus exactement un ensenble de n contraintes qui porte chacune sur une variable – dont la définition varie en fonction de l'optimal local. Sur la plan algorithmique, cette voie semble être la plus prométeuse car elle élargirait le cadre habituel de résolution des CSP.

Bibliographie

[AHU 83] A. V. AHO, J. E. HOPCROFT, et J. D. ULLMAN. Data Structures and Algorithms. Addison-Wesley, 1983.

[Ant 97] J. ANTONIO. « Les problèmes de placement : étude et résolution de quelques problèmes réels ». Thèse de doctorat, Université de Metz, 1997.

[AS 93] J.-M. ALLIOT et Th. SCHIEX. Intelligence artificielle et informatique théorique. Cépaduès-éditions, Toulouse, 1993.

[BCR 80] B. BAKER, E. G. Jr. COFFMAN, et R. L. RIVEST. « Ortogonal Packings in Two Dimensions ». SIAM Journal of Algorithms, 9(4) :846–855, 1980.

[Ben 95] M. BENAICHOUCHE. « Régulation dynamique de la charge pour le B and B distribué : théorie et application ». Rapport technique PRiSM 95/21, Université de Versailles, 1995.

[Ber 70] C. BERGE. Graphes et hypergraphes. Dunod, France, 1970.

[Ber 95] P. BERLANDIER. « Filtrage de problèmes par consistance de chemin restreinte ». Revue d'Intelligence Artificielle, 9(3) :225–238, 1995.

[Bes 94] Ch. BESSIÈRE. « Arc-consistency and Arc-consistency again ». Artificial Intelligence, 65 :179–190, 1994.

[BFR 95] C. BESSIÈRE, E. C. FREUDER, et J.-C. RÉGIN. « Using Inference to Reduce Arc Consistency Computation ». Dans Proceedings of the fourteenth International Joint Conference on Artificial Intelligence, Montreal, 1995. pages 592–598.

[Bib 88] W. BIBEL. « Constraint Satisfaction from a Deductive Viewpoint ». Artificial Intelligence, 35 :401–413, 1988.

[BM 94] R. J. BAYARDO, Jr et D. P. MIRANKER. « An Optimal Backtrack Algorithm for Tree-structured Constraint Satisfaction Problems ». Artificial Intelligence, 71 :159–181, 1994.

[Bou 93] L. BOUGÉ. « Le modèle de programmation à parallélisme de données : une perpective sémantique ». Technique et Science Informatiques, 12(5) :541–562, 1993.

[BR 96] C. BESSIÈRE et J.-C. RÉGIN. « MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems ». Dans Proceedings of the second International Conference on Principles and Practice of Constraint Programming, Cambridge Ma., 1996. pages 61–75.

[Bre 74] R. P. BRENT. « The Parallel Evolution of General Arithmetic Expressions ». Journal of the ACM, 21(2) :201–206, 1974.

[BvR 95] F. BACCHUS et P. van RUN. « Dynamic Variable Ordering in CSPs ». Dans Proceedings of the first International Conference on Principles and Practice of Constraint Programming, Cassis, 1995. pages 258–274.

[CA 95] J. M. CONRAD et D. P. AGRAWAL. « Asynchronous Parallel Arc-consistency Algorithms on Distributed Memory Machine ». Journal of Parallel Distributed Computing, 24 :27–40, 1995.

[CBB 91] J. M. CONRAD, D. BAHLER, et J. BOWEN. « Static Parallel Arc Consistency in Constraint Satisfaction ». Dans Proceedings of the sixth International Symposium on Methodologies for Intelligent Systems, Charlotte, 1991. pages 500–509.

[CCJ 94] M. C. COOPER, D. A. COHEN, et P. G. JEAVONS. « Characterising Tractable Constraints ». Artificial Intelligence, 65 :347–361, 1994.

[CCP+ 92] J.-J. CHABRIER, J. CHABRIER, O. PALMADE, M. CAYROL, A. RAUZY, P. EZEQUEL, J. KAO HAO, G. PLATEAU, H. BENNACEUR, Ch. BESSIÈRE, Ph. JANSSEN, M.-C. VILAREM, B. BENHAMOU, Ph. JÉGOU, L. OXUSSOF, L. SAIS, et P. SIEGEL. « Étude comparative des trois formalismes en calcul propositionnel ». Dans Actes des 4es journées nationales PRC-GDR IA, Marseille, 1992. pages 239–317.

[CD 94] M. COSNARD et F. DESPREZ. « Quelques architectures de nouvelles machines ». Calculateurs parallèles, pages 29–58, mars 1994.

[CGJ 84] E. G. Jr. COFFMAN, M. R. GAREY, et D. S. JOHNSON. Approximation Algorithms for Bin-packing - An Updated Survey. Bell Laboratories, Murray Hill, New Jersey, 1984.

[CM 94] J. M. CONRAD et J. MATHEW. « A Backjumping Search Algorithm for a Distributed Memory Multicomputer ». Dans Proceedings of the 1994 International Conference on Parallel Processing, 1994, volume 3, pages 243–246.

[Coo 85] S. A. COOK. « A Taxonomy of Problems with Fast Algorithms ». Information and Control, 64 :2–22, 1985.

[Coo 89] M. C. COOPER. « An Optimal k-consistency Algorithm ». Artificial Intelligence, 41 :89–95, 1989.

[CS 92] P. R. COOPER et M. J. SWAIN. « Arc Consistency : Parallelism and Domain Dependance ». Artificial Intelligence, 58 :207–235, 1992.

[Cun 94] V.-D. CUNG. « Contribution à l'algorithmique non numérique parallèle: Exploration d'arbres de recherche ». Thèse de doctorat, Université P. et M. Curie, Paris 6, 1994.

[Cyb 89] G. CYBENKO. « Dynamic Load Balancing for Distributed Memory Multiprocessors ». Journal of Parallel and Distributed Computing, 7:279–301, 1989.

[Dav 87] E. DAVIS. « Constraint Propagation with Interval Labels ». Artificial Intelligence, 32:281–331, 1987.

[Dav 93] Ph. DAVID. « When Functional and Bijective Constraints Makes a CSP Polynomial ». Dans Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chambéry, 1993. pages 224–229.

[Dav 94] Ph. DAVID. « Consistance de pivot et CSP fonctionnels: une consistance partielle pour des solutions globales ». Revue d'Intelligence Artificielle, 8(2):145–185, 1994.

[dBKT 95] A. de BRUIN, G. A.P. KINDERVATER, et H. W.J.M. TRIENENEKENS. « Asynchronous Parallel Branch and Bound and Anomalies ». Dans Proceedings of the second International Workshop on Parallel Algorithms for Irregularly Structured Problems, LNCS 980, Lyon, 1995. pages 363–377.

[Dec 90] R. DECHTER. « Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition ». Artificial Intelligence, 41:273–312, 1990.

[Dec 92] R. DECHTER. « From Local to Global Consistency ». Artificial Intelligence, 55:87–107, 1992.

[Del 90] J.-L. DELAPORTE. « Intégration des fonctions de conception et de préparation de la fabrication pour les entreprises de découpe ». Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambresis, 1990.

[DM 94a] R. DECHTER et I. MEIRI. « Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems ». Artificial Intelligence, 68:211–241, 1994.

[DM 94b] M. J. DENT et R. E. MERCER. « Minimal Forward-Checking ». Rapport technique UWO-CSD-374, University of Western Ontario, 1994.

[DP 87] R. DECHTER et J. PEARL. « The Cycle-cutset Method for Improving Search Performance in AI Applications ». Dans Proceedings of the third IEEE on Artificial Intelligence Applications, Orlando, 1987. pages 224–230.

[DP 88a] R. DECHTER et J. PEARL. « Network-based Heuristics for Constraint Satisfaction Problems ». Artificial Intelligence, 34:1–38, 1988.

[DP 88b] R. DECHTER et J. PEARL. « Tree-clustering Schemes for Constraint-processing ». Dans Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-88), Saint Paul, MN, 1988. pages 150–154.

[DP 89] R. DECHTER et J. PEARL. « Tree Clustering for Constraint Networks ». Artificial Intelligence, 38 :353–366, 1989.

[DvB 95] R. DECHTER et P. van BEEK. « Local and Global Relational Consistency ». Dans Proceedings of the first International Conference on Principles and Practice of Constraint Programming, Cassis, 1995. pages 240–257.

[DvH 91] Y. DEVILLE et P. van HENTENRYCK. « An Efficient Arc-consistency Algorithm for a Class of CSP Problems ». Dans Proceedings of the twelfth International Joint Conference on Artificial Intelligence, Sydney, 1991. pages 325–330.

[Elo 92] A. ELOMRI. « Méthodes d'optimisation dans un contexte productique ». Thèse de doctorat, Université de Bordeaux 1, 1992.

[FD 94] D. FROST et R. DECHTER. « In Search of the Best Constraint Satisfaction Search ». Dans Proceedings of the twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, 1994. pages 301–306.

[FD 95] D. FROST et R. DECHTER. « Look-ahead Value Ordering for Constraint Satisfaction Problems ». Dans Proceedings of the fourteenth International Joint Conference on Artificial Intelligence, Montreal, 1995. pages 572–578.

[FD 96] D. FROST et R. DECHTER. « Looking at Full Looking Ahead ». Dans Proceedings of the second International Conference on Principles and Practice of Constraint Programming, Cambridge, Ma., 1996. pages 539–540.

[FG 96] M. A. FRANKLIN et V. GOVINDAN. « A general matrix iterative model for dynamic load balancing ». Parallel computing, 22(7) :969–989, 1996.

[FH 93] E. C. FREUDER et P. D. HUBBE. « Using Inferred Disjunctive Constraints to Decompose Constraint Satisfaction Problems ». Dans Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chambéry, 1993. pages 254–260.

[FM 87] R. FINKEL et U. MANBER. « DIB – A Distributed Implementation of Backtracking ». ACM Transactions on Programming Languages and Systems, 9(2) :235–256, 1987.

[Fre 78] E. C. FREUDER. « Synthesizing Constraint Expressions ». Communication of the Association for Computing Machinery, 21 :956–966, 1978.

[Fre 82] E. C. FREUDER. « A Sufficient Condition for Backtrack-free Search ». Journal of the Association for Computing Machinery, 29 :24–32, 1982.

[Fre 85] E. C. FREUDER. « A Sufficient Condition for Backtrack-bounded Search ». Journal of the Association for Computing Machinery, 32 :755–761, 1985.

[Fre 89] E. C. FREUDER. « Partial Constraint Satisfaction ». Dans Proceedings of the eleventh International Joint Conference on Artificial Intelligence, Detroit, 1989. pages 278–283.

[Fre 95] E. C. FREUDER. « Using Metalevel Constraint Knowledge to Reduce Constraint Checking ». Dans M. MEYER, éd., Constraint Processing, LNCS 923, 1995, pages 171–184.

[Gas 77] J. GASCHNIG. « A General Backtrack Algorithm that Eliminates Most Redundant Tests ». Dans Proceedings of the fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

[Gas 79] J. GASCHNIG. « Performance Measurement and Analysis of Certain Search Algorithms ». Thèse de doctorat, Carnegie Mellon University, 1979.

[GHH 92] H.-W. GÜSGEN, K. HO, et P. N. HILFINGER. « A Tagging Method for Parallel Constraint Satisfaction ». Journal of Parallel and Distributed Computing, 16 :72–75, 1992.

[Gin 93] M. L. GINSBERG. « Dynamic Backtracking ». Journal of Artificial Intelligence Research, 1 :25–46, 1993.

[GJ 79] M. R. GAREY et D. S. JOHNSON. Computer and Intractability. Freeman, San-Francisco, 1979.

[GJC 94] M. GYSSENS, P. G. JEAVONS, et D. A. COHEN. « Decomposing Constraint Satisfaction Problems using Database Techniques ». Artificial Intelligence, 66 :57–89, 1994.

[GS 93] H. GUYENNET et F. SPIES. « Étude comparative de différents algorithmes de répartition de charge dans les systèmes distribués ». La lettre du transputer et des calculateurs parallèles, pages 31–55, juin 1993.

[GS 96] S. A. GRANT et B. M. SMITH. « The Phase Transition Behaviour of Maintaining Arc Consistency ». Dans Proceedings of the twelveth European Conference on Artificial intelligence, Budapest, 1996. pages 175–179.

[GW 94] I. P. GENT et T. WALSH. « The SAT Phase Transition ». Dans Proceedings of the eleventh European Conference on Artificial intelligence, Amsterdam, 1994. pages 105–109.

[HE 80] R. M. HARALICK et G. L. ELLIOT. « Increasing the Search Efficiency for Constraint Satisfaction Problems ». Artificial Intelligence, 14 :263–313, 1980.

[HHG 93] K. HO, P. N. HILFINGER, et H. W. GUESGEN. « Optimistic Parallel Discrete Relaxation ». Dans Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chambéry, 1993. pages 268–273.

[HJ 94] W. HOWER et S. JACOBI. « A Dictributed Realization for Constraint Satisfaction ». Parallel Processing for Artificial Intelligence, 15(2) :107–116, 1994.

[How 90] W. HOWER. « Constraint Satisfaction via Partially Parallel Propagation Steps ». Dans Proceedings of Second International Workshop on Parallelization in Inference Systems, LNCS 590, Dagstuhl Castle, Germany, 1990. pages 234–242.

[HR 95] M. HERMENEGILDO et F. ROSSI. « Strict and Nonstrict Independent AND-parallelism in Logic Programs : Correctness, Efficiency, and Compile-time Conditions ». The Journal of Logic Programming, pages 1–45, 1995.

[HS 79] R. M. HARALICK et L. G. SHAPIRO. « The Consistent Labeling Problem : Part I ». IEEE Transactions on Pattern Analysis & Machine Intelligence, 1 :173–184, 1979.

[HS 80] R. M. HARALICK et L. G. SHAPIRO. « The Consistent Labeling Problem : Part II ». IEEE Transactions on Pattern Analysis & Machine Intelligence, 3 :193–203, 1980.

[HS 95] H. HOPP et P. SANDERS. « Parallel Game Tree Search on SIMD Machines ». Dans Proceedings of the second International Workshop on Parallel Algorithms for Irregularly Structured Problems, LNCS 980, Lyon, 1995. pages 349–361.

[HT 93] W. D. HILLIS et L. W. TUCKER. « the CM-5 Connection Machine : a Scalable Supercomputer ». Communication of the Association for Computing Machinery, 36(11) :31–40, 1993.

[HW 94] T. HOGG et C. P. WILLIAMS. « Expected Gains from Parallelizing Constraint Solving for Hard Problems ». Dans Proceedings of the twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, 1994. pages 331–337.

[JCG 95] P. JEAVONS, D. COHEN, et M. GYSSENS. « A Unifying Framework for Tractable Constraints ». Dans Proceedings of the first International Conference on Principles and Practice of Constraint Programming, Cassis, 1995. pages 276–291.

[Jég 91] Ph. JÉGOU. « Contribution à l'étude des problèmes de satisfaction de contraintes : algorithmes de propagation et de résolution – propagation de contraintes dans les réseaux dynamiques ». Thèse de doctorat, Université des Sciences et Techniques du Languedoc, Montpellier, 1991.

[Jég 93] Ph. JÉGOU. « On the Consistency of General Constraint Satisfaction Problems ». Dans Proceedings of the eleventh National Conference on Artificial Intelligence (AAAI-93), 1993, pages 114–119.

[Kas 89] S. KASIF. « Parallel Solutions to Constraint Satisfaction Problems ». Dans Proceedings of the first International Conference on Principles of Knowledge Representation and Reasoning, Toronto, 1989. pages 180–188.

[Kas 90] S. KASIF. « On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks ». Artificial Intelligence, 45 :275–286, 1990.

[KD 94] S. KASIF et A. L. DELCHER. « Local Consistency in Parallel Constraint Satisfaction Networks ». Artificial Intelligence, 69 :307–327, 1994.

[KLG 91] S. KERETHO, R. LOGANANTHARAJ, et V. N. GUDIVADA. « Parallel Path-consistency Algorithms for Constraint Satisfaction ». Dans Proceedings of International Conference on Tools for Artificial Intelligence, San Jose, 1991. pages 516–517.

[KR 87] V. KUMAR et V. N. RAO. « Parallel Depth First Search. Part II. Analysis ». International Journal of Parallel Programming, 16(6) :501–519, 1987.

[Kum 92] V. KUMAR. « Algorithms for Constraint Satisfaction Problems : a Survey ». Artificial Intelligence Magazine, pages 33–44, spring 1992.

[Lau 93] P. S. LAURSEN. « Simple Approaches to Parallel Branch-and-Bound ». Parallel Computing, 19 :143–152, 1993.

[LHB 92] Q. Y. LUO, P. G. HENDRY, et J. T. BUCHANAN. « A New Algorithm for Dynamic Distributed Constraint Satisfaction Problems ». Dans Proceedings of the fifth Florida Artificial Intelligence Research Symposium, 1992, pages 52–56.

[LM 94] P. B. LADKIN et R. MADDUX. « On Binary Constraint Problems ». Journal of the Association for Computing Machinery, 41(3) :435–469, 1994.

[Mac 77] A. K. MACKWORTH. « Consistency in Networks of Relations ». Artificial Intelligence, 8 :99–118, 1977.

[Mac 92] A. K. MACKWORTH. « The Logic of Constraint Satisfaction ». Artificial Intelligence, 58 :3–30, 1992.

[MF 85] A. K. MACKWORTH et E. C. FREUDER. « The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems ». Artificial Intelligence, 25 :65–74, 1985.

[MH 86] R. MOHR et T. C. HENDERSON. « Arc and Path Consistency Revisited ». Artificial Intelligence, 28 :225–233, 1986.

[MHHS 96] P.-P. MÉREL, Z. HABBAS, F. HERRMANN, et D. SINGER. « N-ary Consistencies and Constraint-based Backtracking ». Dans Proceedings of the second International Conference on Principles and Practice of Constraint Programming, Cambridge, MA, 1996. extended abstract.

[MHHS 97] P.-P. MÉREL, Z. HABBAS, F. HERRMANN, et D. SINGER. « Parallel Search Algorithms for Constraint Satisfaction Problems ». Dans Proceedings of the sixteenth International Symposium on Mathematical Programming, Lausanne, 1997.

[MMH 85] A. K. MACKWORTH, J. A. MULDER, et W. S. HAVENS. « Hierarchical Arc-consistency : Exploiting Structured Domains in Constraint Satisfaction Problems ». Computer Intelligence, 1 :118–126, 1985.

[Mon 74] U. MONTANARI. « Networks of Constraints : Fundamental Properties and Applications to Pictures Processing ». Information Sciences, 7 :95–132, 1974.

[Mor 73] G. MOREAU. « Méthodes pour la résolution des problèmes d'optimisation de découpe ». Thèse de doctorat, Université Claude Bernard, Lyon, 1973.

[NO 94] J. NAGANUMA et T. OGURA. « A Highly OR-parallel Inference Machine (multi-ASCA) and its Performance Evaluation : an Architecture and its Load Balancing Algorithms ». IEEE Transaction on Computers, 43(9) :1062–1075, 1994.

[NS 81] D. NASSIMI et S. SAHNI. « Data Broadcasting in SIMD Computers ». IEEE Transactions on Computers, C-30(2) :101–106, 1981.

[PFK 93] C. POWLEY, C. FERGUSON, et R. E. KORF. « Depth-first Heuristic Search on a SIMD Machine ». Artificial Intelligence, 53 :329–342, 1993.

[PG 95] E. PONTELLI et G. GUPTA. « On the Duality between OR-parallelism and AND-parallelism in Logic Programming ». Dans Proceedings of the International Conference on Parallel Processing, Stockholm, 1995.

[Pip 79] N. PIPPENGER. « On Simultaneous Ressource Bounds ». Dans Proceedings of the twentieth Annual IEEE Foundations of Computer Science, 1979, pages 307–311.

[Pro 93a] P. PROSSER. « Domain Filtering can Degrade Intelligent Backtracking Search ». Dans Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chambéry, 1993. pages 262–267.

[Pro 93b] P. PROSSER. « Hybrid Algorithms for the Constraint Satisfaction Problem ». Computational Intelligence, 9 :268–299, 1993.

[Pro 94] P. PROSSER. « Binary Constraint Satisfaction Problems : Some are Harder than Others ». Dans Proceedings of the eleventh European Conference on Artificial Intelligence, Amsterdam, 1994. pages 95–99.

[Pro 95a] P. PROSSER. « Forward Checking with Backmarking ». Dans M. MEYER, éd., Constraint Processing, LNCS 923, 1995, pages 185–204.

[Pro 95b] P. PROSSER. « MAC-CBJ : Maintaining Arc Consistency with Conflict-directed Backjumping ». Rapport technique 95/117, University of Strathclyde, 1995.

[Pro 96] P. PROSSER. « An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems ». Artificial Intelligence, 81 :81–109, 1996.

[RK 87] V. N. RAO et V. KUMAR. « Parallel Depth First Search. Part I. Implementation ». International Journal of Parallel Programming, 16(6) :479–499, 1987.

[RK 93] V. N. RAO et V. KUMAR. « On the Efficiency of Parallel Backtracking ». IEEE Transaction on Parallel and Distributed Systems, 4(4) :427–437, 1993.

[Rou 92] C. ROUCAIROL. « Exploration Parallèle d'espace de recherche en recherche opérationnelle et intelligence artificielle ». Dans Y. Robert M. COSNARD, M. Nivat, éd., Algorithmique Parallèle, 1992, pages 201–212.

[Rou 93] C. ROUCAIROL. « La Recherche Opérationnelle ». Courrier de CNRS, Dossiers scientifiques, La Recherche Informatique, (80), 1993.

[Rou 95] C. ROUCAIROL. « On Irregular Data Structures and Asynchronous Parallel Branch and Bound Algorithms ». DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 323–336, 1995.

[San 95] P. SANDERS. « Better Algorithms for Parallel Backtracking ». Dans Proceedings of the second International Workshop on Parallel Algorithms for Irregularly Structured Problems, LNCS 980, Lyon, 1995. pages 333–347.

[Sei 81] R. SEIDEL. « A New Method for Solving Constraint Satisfaction Problems ». Dans Proceedings of the seventh International Joint Conference on Artificial Intelligence, Vancouver, 1981. pages 338–342.

[SF 94] D. SABIN et E. C. FREUDER. « Contradicting Conventional Wisdom in Constraint Satisfaction ». Dans Proceedings of the eleventh European Conference on Artificial Intelligence, Amsterdam, 1994. pages 125–129.

[SH 87] A. SAMAL et T. C. HENDERSON. « Parallel Consistent Labeling Algorithms ». International Journal of Parallel Programming, 16 :341–364, 1987.

[SHZ+ 91] S. Y. SUSSWEIN, T. C. HENDERSON, J. L. ZACHARY, C. HANSEND, P. HINKER, et G. C. MARSDEN. « Parallel Path-consistent ». International Journal of Parallel Programming, 20(6) :453–473, 1991.

[Smi 94] B. M. SMITH. « Phase Transition and the Mushy Region in Constraint Satisfaction Problems ». Dans Proceedings of the eleventh European Conference on Artificial Intelligence, 1994, pages 100–104.

[SV 93] Th. SCHIEX et G. VERFAILLIE. « Nogood Recording for Static and Dynamic Constraint Satisfaction Problems ». Dans Proceedings of the IEEE Internationnal Conference on Tools with Artificial Intelligence, Boston, 1993. pages 48–55.

[Swa 88] M. J. SWAIN. « Comment on SAMAL and HENDERSON : *Parallel Consistency Labeling Algorithms* ». International Journal of Parallel Programming, 17 :523–528, 1988.

[Tsa 93] E. TSANG. Foundations of Constraints Satisfaction. Academic Press, London, 1993.

[vHDT 92] P. van HENTENRYCK, Y. DEVILLE, et C.-M. TENG. « A Generic Arc-consistency Algorithm and its Specializations ». Artificial intelligence, 57 :113–159, 1992.

[Wal 60] R. J. WALKER. « An Enumerative Technique for a Class of Combinatorial Problems ». Proceedings of the Symposium of Applied Mathematics, 10 :91–94, 1960.

[Wal 93] R. J. WALLACE. « Why AC-3 is Almost Always Aetter than AC-4 for Establishing Arc-consistency in CSPs ». Dans Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chambéry, 1993. pages 239–245.

[WH 94] C. P. WILLIAMS et T. HOGG. « Exploiting the Deep Structure of Constraint Satisfaction ». Artificial intelligence, 70 :73–117, 1994.

[WLR 93] M. H. WILLEBEEK-LEMAIR et A. P. REEVES. « Strategies for Dynamic Load Balancing on Highly Parallel Computers ». IEEE Transactions on Parallel and Distributed Systems, 4(9) :979–993, 1993.

Index

– S –

– T –

– V –

Les problèmes de satisfaction de contraintes : recherche n-aire et parallélisme – Application au placement en CAO

Pierre-Paul MÉREL

Les problèmes de satisfaction de contraintes (CSP) constituent un cadre de formalisation puissant des problèmes d'Intelligence Artificielle. Un CSP est la donnée d'un ensemble de variables définies sur certains domaines, et d'un ensemble de contraintes reliant ces variables. De nombreuses méthodes ont été définies pour les résoudre. Nous nous intéressons dans ce document, aux méthodes complètes, *i.e.* qui permettent toujours de trouver une solution.

Les méthodes complètes de résolution les plus performantes pour les CSP utilisent un mécanisme de recherche par vérification de consistance « en avant », c.-à-d. en filtrant les valeurs incompatibles dans les domaines non instanciés lorsqu'une nouvelle instanciation est testée (*forward-checking*). Cependant, ces méthodes ne sont définies que pour des contraintes binaires. Nous étendons ces méthodes en définissant une notion de consistance n-aire généralisée : la *relationnelle-(i, j)-consistance*. Nos expérimentations montrent que la résolution n-aire directe est plus rapide que la résolution sur un CSP binaire obtenu après transformation.
Comme la recherche séquentielle est particulièrement coûteuse (problème NP-complet). le parallélisme permet de réduire le temps de calcul. Pour cela, nous définissons un algorithme générique qui permet d'hybrider aussi bien les différentes techniques de recherche séquentielle que les méthodes de parallélisation. Le principe de la recherche reprend le schéma « OU-parallèle ». Si la plupart des techniques parallèles classiques restent utilisables pour les CSP, nous définissons différentes techniques destinées à mieux s'adapter aux spécificités des CSP. Ainsi, nous proposons une méthode de répartition initiale de charge. une méthode de division adaptée à la taille des contextes de recherche et une fonction de détermination de la divisibilité d'une tâche.
Nous appliquons et spécialisons les algorithmes définis pour la résolution de problèmes du bordereau de coupe en confection, *i.e.* le placement de formes quelconques sur une surface restreinte.

Mots clés : problèmes de satisfaction de contraintes, recherche complète, consistance en avant, contraintes n-aires, parallélisme, problèmes de placement.