RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieure et de la Recherche Scientifique

Université Mira Abderrahmene de Béjaia

DÉCOMPOSITIONS ARBORESCENTES POUR RÉSOUDRE LES PROBLÈMES DE SATISFACTION DE CONTRAINTES AVEC MISE EN ŒUVRE DU PARLLÉLISME

PAR

LALOU Mohammed

PRÉSENTÉ À LA FACULTÉ DES SCIENCES EXACTES

DÉPARTEMENT D'INFORMATIQUE, ÉCOLE DOCTORALE D'INFORMATIQUE

ReSyD

(Réseaux et Systèmes Distribués)

POUR l'OBTENTION DU GRADE DE MAGISTÈRE EN INFORMATIQUE

À

L'Université Mira Abderrahmene

Béjaia, 06000

Accepté sur proposition du jury

Président :	Moussa Kerkar, Professeur à Université de Béjaïa, Algérie
Rapporteur :	Zineb Habbas, MCF-HDR, IUT de Metz, France
Examinateurs :	Boukeram Abdellah, MCF à l'Université de Sétif, Algérie
	Moussaoui Abdelouhab, MCF à l'Université de Sétif, Algérie
Invité :	Amroun Kamal, Chargé de cours, Université de Béjaïa, Algérie

Table des Matières

Table des matières **iv**

Liste des Tableaux **viii**

Liste des Figures **ix**

Remerciements **x**

Abstract **xi**

Introduction générale **1**

1 Le formalisme CSP et les différentes approches de résolution **4**
1.1 Que ce qu'un CSP ? . . . 4
1.2 Définitions de base . . . 5
1.3 Notion de consistance . . . 6
1.3.1 Consistance de noeud . . . 6
1.3.2 Consistance d'arc . . . 7
1.3.3 Consistance d'une hyper-arc . . . 7
1.3.4 La k-consistance . . . 7
1.4 Extension des CSP . . . 7
1.5 Exemple de modélisation à l'aide d'un CSP (*problème de coloration de graphe*) . . . 8
1.6 Les différentes approches de résolution des problèmes CSP . . . 9
1.6.1 Les méthodes incomplètes . . . 9
1.6.2 Les méthodes complètes . . . 10
1.7 Résolution séquentielle par des méthodes complètes . . . 11
1.7.1 Résolution par des algorithmes énumératifs . . . 11
1.7.1.1 Algorithme Backtrack (BT) . . . 12
1.7.2 Algorithmes énumératifs avec heuristiques . . . 12
1.7.2.1 Heuristiques sur le retour arrière . . . 13
1.7.2.2 Heuristiques sur l'instanciation des variables . . . 14
1.7.3 Résolution par des algorithmes de Filtrage par propagation de contraintes . . . 15
1.7.4 Résolution par décomposition . . . 19
1.7.5 Résolution par des algorithmes hybrides . . . 19
1.8 Conclusion . . . 20

2 les méthodes de décomposition structurelle 21
2.1 Introduction . . . 21
2.2 Complexité des CSP . . . 22
2.2.1 NP-complétude ? . . . 22
2.3 Traitabilité . . . 22
2.3.1 Traitabilité dûe à la structure . . . 23
2.3.2 Traitabilité due aux relations . . . 23
2.4 Réseaux de contraintes . . . 23
2.5 Les méthodes de décomposition structurelles . . . 25
2.5.1 Principe de base . . . 25
2.5.2 Méthode Biconnected components . . . 26
2.5.3 Méthode Cycle Cutsest . . . 26
2.5.4 Méthode Hypercutset . . . 28
2.5.5 Méthode Hinge decomposition . . . 28
2.5.6 Méthode de décomposition arborescente (Tree decomoposition . . . 30
2.5.7 Méthode Tree clustering . . . 31
2.5.8 Méthode hypertree decomposition . . . 33
2.5.8.1 Calcul de l'hypertree decomposition . . . 34
2.5.9 Comparaison . . . 35
2.5.10 Conclusion . . . 36

3 La résolution séquentielle par méthodes de décomposition structurelle 38
3.1 Introduction . . . 38
3.2 La résolution séquentielle des problèmes CSP acycliques . . . 39
3.2.1 La résolution d'un CSP binaire acyclique . . . 39
3.2.2 La résolution d'un CSP n-aire acyclique . . . 40
3.3 Résolution du CSP après une Tree decomposition . . . 41
3.4 Résolution du CSP après une décomposition hypertree . . . 42
3.5 L'algorithme S-HBR *(Sequential Hash Based Resolution)* . . . 44
3.5.1 Introduction . . . 44
3.5.2 Algorithme S-HBR . . . 45
3.5.3 L'algorithme SP-HBR *(Sub Problem Hash Based Resolution)* . . . 45
3.5.4 L'algorithme A-HBR *(Acyclic Hash Based Resolution)* . . . 49
3.5.4.1 L'opération CONTRACT . . . 50
3.5.4.2 L'opération S-SEARCH . . . 50
3.6 Complexité de l'algorithme *S-HBR* . . . 52
3.7 Correction de notre algorithme *S-HBR* . . . 52
3.8 Expérimentations . . . 53
3.8.1 Conclusion . . . 55

4 Le parallélisme 57
4.1 Introduction . . . 57
4.2 Modèles de programmation parallèle . . . 58
4.2.1 Parallélisme de données . . . 58
4.2.2 Parallélisme de programme . . . 58

4.3 Les types du parallélisme d'un opérateur de jointure . . . 59
4.3.1 Le parallélisme intra-opérateurs . . . 59
4.3.2 Le parallélisme inter-opérateurs . . . 59
4.4 le parallélisme dans l'exécution des jointures et semi jointures . . . 60
4.4.1 Exécution séquentielle parallèle . . . 61
4.4.2 Exécution synchrone parallèle . . . 61
4.4.3 Exécution segmentée en profondeur droite . . . 61
4.4.4 Exécution parallèle complète . . . 61
4.5 Les architectures parallèles . . . 62
4.5.1 Architecture à mémoire partagée (*SM*) . . . 62
4.5.2 Architecture à disque partagé (*SD*) . . . 62
4.5.3 Architecture à disques répartis (*SN*) . . . 63
4.6 Les techniques de bases du parallélisme . . . 63
4.6.1 La contraction d'arbre [32] . . . 64
4.6.1.1 Les opérations *RAKE*, *COMPRESS* et *CONTRACT* . . . 64
4.7 Mesure de performance d'un algorithme parallèle . . . 66
4.8 Conclusion . . . 67

5 La résolution parallèle des problèmes CSP 68
5.1 Introduction . . . 68
5.2 La complexité parallèle des problèmes CSP . . . 69
5.3 Les approches de résolution parallèle . . . 69
5.3.1 L'approche concurrente . . . 69
5.3.2 L'approche coopérative . . . 69
5.4 Les algorithmes de résolution parallèles des CSP . . . 70
5.4.1 La résolution parallèle des CSP cycliques . . . 70
5.4.1.1 Parallélisation des algorithmes énumératifs . . . 70
5.4.1.2 Parallélisation des algorithmes de filtrage . . . 71
5.4.2 La résolution parallèle des CSP acycliques . . . 71
5.4.2.1 L'algorithme PTAC (Parallel Tree Arc Consistency) . . . 72
5.5 L'algorithme de résolution parallèle P-HTR (*Parallele HyperTree Resolution*) . . . 74
5.5.1 Introduction . . . 74
5.5.2 Algorithme *P-HTR* . . . 76
5.5.3 L'algorithme P-SBR . . . 76
5.5.4 L'algorithme parallèle P-Solving . . . 78
5.5.4.1 L'algorithme P-CONTRACT . . . 79
5.5.4.2 L'operation P-RAKE . . . 79
5.5.4.3 L'operation P-COMPRESS . . . 82
5.5.4.4 L'algorithme PS-SEARCH . . . 83
5.6 Complexité de P-HTR . . . 83
5.7 Expérimentations et analyses . . . 84
5.7.1 Environnement de simulation . . . 84
5.7.2 Modèle de simulation . . . 84
5.7.3 Déroulement de simulation . . . 85
5.7.4 Protocoles expérimentaux . . . 85

5.7.5 Simulation de l'algorithmes P-SBR 86
5.7.6 Etude des deux techniques $P - RAKE_1$ et $P - RAKE_2$ 88
5.7.7 Simulation de l'algorithme P-solving 90
5.8 Conclusion 91

Conclusion générale et Perspectives **93**

Bibliographie **95**

Liste des tableaux

3.1 Résultats expérimentaux de l'algorithme *S-HBR* 54

5.1 Simulation de l'algorithme *P-SBR* pour des sous problèmes des instances $\mathcal{CSP}$ 86
5.2 Simulation de l'algorithme *P-SBR* pour des instances $\mathcal{CSP}$ 87
5.3 L'efficacité des deux techniques $\mathcal{P}$-$\mathcal{RAKE}$ 89
5.4 Simulation de l'algorithme *P-solving* 91

Table des figures

1.1 Problème de coloration de graphe 8
1.2 les différentes approches de résolution des *CSP* 10

2.1 (a) hypergraphe d'un CSP (b) Son graphe primal (c) Son graphe dual 25
2.2 Un graphe (a) et sa Biconnected décomposition (b) 26
2.3 Un graphe et sa décomposition *Cycle Cutset* 27
2.4 Un graphe et sa décomposition *Cycle HyperCutset* 28
2.5 Un hypergraphe (a) et son *Hinge Tree* (b) 29
2.6 Un hypergraphe (a) et sa *Tree decomposition* (b) 31
2.7 Etapes de décomposition *Tree clustring* 32
2.8 (a) hypertree décomposition généralisée (b) hypertree décomposition 34
2.9 Comparaison entre les méthodes de décomposition 36

3.1 Résolution à partir d'une hypertree décomposition 42
3.2 Résolution à partir d'une hypertree décomposition 43
3.3 Exemple d'exécution de l'algorithme *SP-HBR* 48

4.1 Les différent types d'arbre d'opérateurs (jointure) 60
4.2 Les architectures parallèles (a)Shared Memory, (b) Shared Disks, (c) Shared Nothing . . 63
4.3 L'opération *RAKE* 65
4.4 L'opération *COMPRESS* 65

5.1 (a)Contraction avec l'opération (*RAKE*)(b) Contraction avec l'opération (*P-RAKE*) . . 80
5.2 (a)Contraction avec l'opération ($RAKE_2$) 82
5.3 Les performances spaciales de *P-SBR* selon (a) la taille des relations (b) le nombre d'etage par ligne du pipeline 88
5.4 L'accélération de P-RAKE (2) 90

Remerciements

Je loue Dieu de la grâce qu'il m'a faite en me donnant la santé, la patience et la détermination sans les quelles je n'aurais jamais pu porter mon projet à son terme.

Je voudrais tout d'abord exprimer mes remerciements et ma gratitude à Mme Habbas pour m'avoir encadré durant ce projet, son aide et sa confiance m'ont grandement aidé à mener à bien mon travail, ainsi que Mr Amroun pour ses conseils et critiques.

Je remercie les membres de jury qui ont accepté de juger ce travail. J'adresse mes très sincères remerciements à Mr Kerkar, Mr Boukeram et à Mr Moussaoui de me faire l'honneur de s'intéresser à ce travail et d'avoir accepté de faire partie de jury.

Je remercie également Mr TARI Kamel, le chef de département d'informatique de l'université de Béjaia pour son aide, ses conseils et sa disponibilité. Un merci bien distingué à mes enseignants de l'école doctorale ReSyD. J'aimerai remercier également tous les étudiants de l'école doctorale pour l'environnement de travail très agréable durant ces deux dernières années.

Enfin, et surtout, je remercie vivement ma famille qui m'a beaucoup soutenu et encouragé, en particulier mes parents et surtout ma mère qui a été toujours derrière moi pour m'encourager et me soutenir. Merci encore.

Abstract

The $\mathcal{CSP}$ formalism ($\mathcal{C}$onstraint $\mathcal{S}$atisfaction $\mathcal{P}$roblems) forms a powerful and general setting to represent and solve lot of problems. $\mathcal{CSP}$s are generally $\mathcal{NP}$-$\mathcal{C}$omplete. Many works have been led to reduce the complexity boundaries of these problems. Among these methods we are interested in the methods say structural decomposition , and precisely to the hypertree decomposition for which it is proven that it generalizes them all. This method consists to decompose the structure of a $\mathcal{CSP}$ that is a hypergraphe on a hypertree structure of limited width. A $\mathcal{CSP}$ of which the structure is a tree can be solved in a polynomial time. However, the cost of work achieved in each node of the tree is sometimes prohibtive in time and memory space. In this work we propose a new efficient algorithm for solving $\mathcal{CSP}$s problems after a structural decomposition.

In a first time, we present an algorithm $\mathcal{S}$-$\mathcal{HBR}$ for sequential resolution which benefits of the hash technique. In a second time, we propose an algorithm $\mathcal{P}$-$\mathcal{HTR}$ for parallel resolution which is based on a pipeline technique for the resolution of the sub problems in order to optimize the spatial complexity, and for the global resolution of the problem, we propose a new parallel tree contraction technique.

Keywords : Constraint satisfaction problems ($\mathcal{CSP}$), hypertree decomposition, solving $\mathcal{CSP}$, parallelism.

Introduction générale

De nombreux problèmes du monde réel sont représentés par des contraintes. Notre mode de vie est largement déterminé par nos revenus. Le temps disponible limite la quantité et la qualité de travail que nous sommes en mesure d'accomplir. Ainsi, chaque jour, nous essayons de résoudre des problèmes de manière à ce que les contraintes inherentes à ces problèmes soient satisfaites. Dans la science, nous sommes souvent amenés à trouver des solutions à des problèmes qui satisfont des contraintes, et comme leur complexité augmente nous faisons recours aux ordinateurs afin de les résoudre ce qui necéssite une bonne formulation. Àpartir des travaux de Montanari [45] les chercheurs en intelligence artificielle ont developpé un formalisme pour la modélisation et la résolution de cette classe de problèmes combinatoires, connu sous le nom de $\mathcal{CSP}$ pour $\mathcal{C}$onstraints $\mathcal{S}$atisfaction $\mathcal{P}$roblem.

Un grand nombre de problèmes en intelligence artificielle et dans d'autres domaines de l'informatique peuvent être interprétés comme des cas spéciaux de problèmes de satisfaction de contraintes. Quelques exemples d'applications sont les problèmes de planification, le problème de coloration des graphes, les problèmes $\mathcal{SAT}$ (boolean satisfiability), les problèmes de conjonction des requêtes et beaucoup d'autres problèmes intéressants.

Certains formalismes qui ont été proposés pour modéliser et résoudre les problèmes rencontrés se veulent dédiés à la résolution d'un problème précis. D'autres, plus généraux, permettent de représenter des problèmes a priori totalement différents. C'est le cas du formalisme $\mathcal{CSP}$. Cependant, ce formalisme ne permet pas d'exprimer certaines notions comme celle de préférence . Pour cela, plusieurs extensions ont été proposées, parmi lesquelles les $\mathcal{CSP}$s valués ($\mathcal{V}CSP$s), les $\mathcal{M}$ax-$\mathcal{CSP}$ et autres.

D'une façon informelle, un $\mathcal{CSP}$ est constitué de variables et de valeurs qui peuvent être affectées à ces variables. Ces variables sont liées par une ou plusieurs contraintes. Une solution d'un $\mathcal{CSP}$ est une affectation des valeurs aux variables qui ne viole aucune contrainte.

Le principal avantage du formalisme $\mathcal{CSP}$ est qu'il représente une classe générale des problèmes y compris des problèmes réels. une méthode de résolution des problèmes $\mathcal{CSP}$s est une méthode pour résoudre tous les problèmes qui possèdent une formulation

$\mathcal{CSP}$. La principale difficulté réside dans le fait que résoudre un $\mathcal{CSP}$ constitue un problème $\mathcal{NP}$-$\mathcal{C}$omplet. Devant cette difficulté, différents axes de recherche ont été explorés, durant ces trois dernières décennies, conduisant à la définition de nombreuses méthodes de résolution qui ressortent de deux approches : une repose sur l'exploration complète de l'espace de recherche, tandis que l'autre utilise des heuristiques.

Pour la première approche, certaines des méthodes utilisent différentes techniques de recherche sur le problème pour trouver la solution. D'autres utilisent des algorithmes de filtrage pour réduire l'espace de recherche par propagation de contrainte afin de simplifier le problème. D'autres encore exploitent certaines propriétés structurelles des problèmes. La combinaison de ces différentes techniques permet de trouver des algorithmes encore plus performants. Toutes ces techniques tendent vers le même but : résoudre les problèmes posés en vue de trouver des solutions les plus optimales possibles et en un minimum de temps et d'espace.

La technique la plus simple pour déterminer s'il existe ou non une solution consiste à effectuer un parcours systématique de l'espace de recherche. Une méthode exploitant cette technique énumère donc toutes les possibilités. De nombreux travaux ont porté sur l'amélioration de cette approche afin de réduire le nombre de possibilités à étudier, parmi lesquelles on peut citer, le retour arrière intelligent, le filtrage, la mémorisation...etc. Ces méthodes sont généralement guidées par des heuristiques dont le rôle se révèle souvent prépondérant pour l'efficacité de ces méthodes.

Parmi les axes de recherche étudiés pour l'amélioration de la resolution des $\mathcal{CSP}$ nous nous interessons aux méthodes de décomposition structurelle. Une méthode de décomposition transforme une instance $\mathcal{CSP}$ en une autre instance qui peut être résolue de manière efficace. En effet, ceci est assuré par la décomposition du $\mathcal{CSP}$ en un arbre de sous-problèmes. Les sous problèmes étant, en général, de taille plus petite que celle du $\mathcal{CSP}$ original, et donc leur résolution est a priori plus rapide.

Ces méthodes structurelles présentent l'avantage de garantir des bornes de complexité inférieur à celles des méthodes énumératives, ceci avant de procéder à la resolution du problème. Les travaux présentés dans ce mémoire s'intéressent à la résolution de $\mathcal{CSP}$s après une décomposition structurelle en tirant profit des techniques de parallélisme.

Une première partie de ce mémoire est consacrée a la définition et l'implementation d'un algorithme séquentiel de résolution. Nous définissons alors un algorithme $\mathcal{S}$-$\mathcal{HBR}$ (pour $\mathcal{S}$equential $\mathcal{H}$ash $\mathcal{B}$ased $\mathcal{R}$esolution) qui exploite la notion de hachage afin d'apporter ses avantages à la résolution des problèmes $\mathcal{CSP}$. L'objectif de cet algorithme est d'optimiser la complexité en terme de temps, en accélarant la recherche d'un tuple dans une relation.

La seconde partie porte sur la parallélisation de l'algorithme séquentiel $\mathcal{S}$-$\mathcal{HBR}$. Sachant que les deux critères à considérer dans le calcul de performances d'une méthode sont l'espace et le temps, leurs optimisation est souvent la question posée. Pour répondre à cette question, la parallelisation de $\mathcal{S}$-$\mathcal{HBR}$ étant basée sur deux techniques, le pipeline et la contraction d'arbre parallèle. Employer une technique de pipeline pour l'exécution d'une ligne d'opérations de jointure permet d'optimiser au moins partiellement la complexité spatiale en évitant le stackage des relations intermediaires. Généralement la complexité spatiale est générée par la résolution des sous problèmes. Ainsi, cette technique est utilisée pour la proposition d'un algorithme $\mathcal{P}$-$\mathcal{SBR}$ (pour $\mathcal{P}$ipelined $\mathcal{S}$ub $\mathcal{P}$roblem $\mathcal{R}$esolution) de résolution des sous problèmes.

Pour la résolution globale du problème, nous avons proposé deux schémas de contraction d'arbre parallèle dans un algorithme $\mathcal{P}$-$\mathcal{HTR}$ (pour $\mathcal{P}$arallel $\mathcal{H}$yper$\mathcal{T}$ree $\mathcal{R}$esolution). Une étude comparative est menée pour choisir le plus éfficace.

Ce mémoire est organisé en cinq chapitres :

Le chapitre 1 présente le formalisme $\mathcal{CSP}$. Nous présentons d'abord les different concepts de ce formalisme. Ensuite, nous évoquons les principales approches de résolution séquentielle.

Le chapitre 2 est consacré à l'introduction de differentes méthodes de décompositions structurelles des problèmes $\mathcal{CSP}$. Après avoir présenté la classe des $\mathcal{CSP}$ dite traitable, nous definissons le principe de fonctionnement de chacune des méthodes. Nous finissons par une comparaison entre ces méthodes de décomposition.

Le chapitre 3 est consacré a la présentation des principales méthodes de résolution séquentielle après une décomposition structurelle. D'une part, nous rappelons les travaux realisé dans ce cadre. D'autre part, nous définissons une nouvelle méthode $\mathcal{S}$-$\mathcal{HBR}$ qui est basée sur la notion de hachage. Après avoir décrit la méthode, nous présentons des résultats expérimentaux qui mettent en avant l'intéret de l'approche.

Dans le chapitre 4, nous introduisons les differentes techniques de parallélisation en nous focalisant principalement sur celles utilisées dans la parallélisation de l'algorithme $\mathcal{S}$-$\mathcal{HBR}$.

Le chapitre 5 propose une nouvelle méthode de résolution parallèle des problèmes $\mathcal{CSP}$. Nous décrivons entre autre une méthode pour la résolution des sous problèmes dont l'objectif principal est d'optimiser la complexité spaciale, et une autre pour la résolution du problème global. Cette dernière est proposée selon deux schémas differents parmi lesquels nous choisissons le plus adapté. Nous présentons également une simulation de différentes méthodes parallèles proposées.

Chapitre 1

Le formalisme CSP et les différentes approches de résolution

Un grand nombre de problèmes issus de différents domaines de l'informatique peuvent être modélisés comme des problèmes de satisfaction de contraintes (CSP). Ceci, car le formalisme *CSP* s'inscrit dans la catégorie des formalismes généraux qui permettent de modéliser et résoudre des problèmes totalement différents, d'une part. D'autre part, la modélisation d'un problème sous forme de CSP se révèle souvent pratique et intuitive. Pour cela, il a fait l'objet d'une recherche intense à la fois en Intelligence Artificielle (IA) et en Recherche Opérationnelle (RO). Parmi les problèmes étudiés en utilisant ce formalisme on peut citer le maintien de la cohérence [9], l'ordonnencement [29], le raisonnement temporel [38], la théorie des graphes [5] [31] ...etc.

Dans ce chapitre, Nous rappelons d'abord le formalisme *CSP* introduit par Montanari [45]. Ensuite, nous présentons les principales méthodes utilisées pour la résolution des problèmes représentés par ce formalisme. Nous nous intéressons d'abord aux travaux développés dans un cadre séquentiel en limitant essentiellement nos rappels aux approches énumératives et structurelles.

1.1 Que ce qu'un CSP ?

Le problème de satisfaction de contraintes *CSP* (pour *Constraint Satisfaction Problem*) est un formalisme très puissant pour la résolution des problèmes combinatoires. Informellement, un *CSP* est défini par un ensemble fini de variables, chaque variable prend ses valeurs dans un ensemble fini de valeurs du domaine de la variable, et un ensemble de contraintes sur ces variables. Chaque contrainte restreint les combinaisons des valeurs possibles prises par les variables.

La résolution d'un problème de satisfaction de contrainte consiste à trouver une

affectation pour toutes les variables de telle sorte qu'aucune contrainte ne soit violée.

1.2 Définitions de base

Définition 1.1 (*Formalisme CSP*)[45]

Un problème de satisfaction de contraintes (*CSP*) est un quadruplet *(X, D, C, R)*, où :

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

Définition 1.2 (*Contrainte*)

Une contrainte est une relation logique (une propriété qui doit être vérifiée) entre différentes variables, chacune prenant ses valeurs dans un ensemble donné, appelé domaine. Une contrainte restreint les valeurs que peuvent prendre simultanément les variables, elle peut être :

- *Implicite*, définie par une expression logique. Par exemple : $X + Y < 4$.
- *Explicite*, définie par une relation composée de toute les tuples autorisés. Par exemple : pour un domaine $D = \{1, 2, 3\}$ de X et Y, la contrainte précédente est $R = \{(1, 1), (1, 2), (2, 1)\}$.

Les variables sur lesquelles porte la contrainte C_i sont appelées *Portée* de la contrainte, noté $S(C_i)$.

Définition 1.3 (*Arité*)

L'arité d'une contrainte $C_i = \{X_{i_1}, ..., X_{i_{ni}}\}$ est le nombre i_{ni} de variables sur lesquelles porte C_i. Un *CSP* est dit binaire si pour toute contrainte $C_i \in C$, l'arité de C_i est au plus de 2, sinon il est n-aire.

Définition 1.4 (*Réseaux de contraintes*)

Le réseau de contraintes associé à un *CSP* est un graphe (pour les *CSP* binaire) ou un hypergraphe (pour les *CSP* n-aire) dont les noeuds sont les variables et les arêtes sont les contraintes.

Définition 1.5 (*Instanciation*)

Etant donné un *CSP*, $P = (X, D, C, R)$ et soit $Y \subset X$ un sous ensemble de variables de X. On appelle instanciation de Y l'association de chaque variable y de Y a une valeur de $D(y)$ (avec $D(y)$ est le domaine de la variable y).

Définition 1.6 (*Instanciation consistante*)

Etant donné un *CSP*, $P = (X, D, C, R)$ et soit $Y \subset X$ un sous ensemble de variables de X. Une instanciation des variables de Y sur D est dite consistante si et seulement si elle satisfait toutes les contraintes portant sur ses variables.

Définition 1.7 (*Solution d'un CSP*)

On appelle une solution d'un CSP $P = (X, D, C, R)$, l'instanciation consistante des variables de X sur D.

Une instance *CSP* est dite *consistante* si elle possède au moins une solution.

1.3 Notion de consistance

Un domaine est consistant si toutes ses valeurs sont valides.

1.3.1 Consistance de noeud

La consistance de noeud concerne les contraintes unaires (ne portant que sur une seule variable). Ce sont les contraintes les plus simples, car elles n'affectent qu'une seule variable.

Définition 1.8

Un *CSP* est noeud consistant si pour chaque variable $x \in X$, et pour toute valeur v de $D(x)$, l'affectation partielle $\{(x, v)\}$ satisfait toutes les contraintes unaires de C.

1.3.2 Consistance d'arc

De manière informelle, une contrainte binaire (x,y) est consistante d'arc si chaque valeur de x (resp. y) possède une valeur consistante dans le domaine de y (resp. x).

Définition 1.9

On dit qu'un *CSP* est arc consistant si pour chaque contrainte binaire $C_i \in C$ portant sur deux variables X_1 et X_2 de domaines respectifs D_{X_1}, D_{X_2}. pour chaque valeur a de D_{X_1} (resp. D_{X_2}) il existe un tuple d dont la première (resp. deuxième) composante est egale a a.

1.3.3 Consistance d'une hyper-arc

La notion de consistance hyper-arc généralise celle d'arc pour les contraintes n-aires.

Définition 1.10

On dit qu'un *CSP* est hyper-arc consistant si pour chaque contrainte $C_i \in C$ portant sur les variables $\{X_1, X_2, \ldots, X_n\}$ avec leur domaine respectif $D_{X_1}, D_{X_2}, \ldots, D_{X_n}$ de sorte que $C_i \subset D_{X_1} \times D_{X_2} \times \cdots \times D_{X_n}$. Pour chaque $j = [1..n]$ et $a \in D_{X_j}$, il existe un n-uplet d dans C_i de sorte que la j-ième composante de d soit égale à a.

1.3.4 La k-consistance

Définition 1.11

Soit s une instanciation partielle de longueur k, i.e. k variables sont instanciées pour un $CSP = (X, D, C, R)$, si s satisfait toutes les contraintes, on dit alors que l'affectation est $k - consistante$.

Un algorithme permettant d'établir la k-consistance a été présenté en 1989 [7].

La $k_c onsistance$ est une généralisation de la simple consistance. En particulier la noeud consistance correspond à la 1-consistance et l'arc consistance correspond à la 2-consistant.

1.4 Extension des CSP

Pour que le formalisme *CSP* apporte des réponses satisfaisantes même pour la modélisation des problèmes dans lesquels entre en jeu la notion de coût, de préférence, etc.

des extensions ont été proposées :

- ***Max-CSP*** : qui consiste à trouver l'affectation totale qui viole le moins de contraintes possibles pour un problème sur contraint (qui n'a pas de solution).

- ***VCSP*** : ou les CSP valués introduit par Schiex Fargier et Verfaillie [42] qui consiste à chercher l'affectation totale qui minimise la somme des poids des contraintes, oú le poids est une valeur associée à chaque contrainte.

- ***CSOP*** : (Constraint Satisfaction Optimisation Problem). Il consiste à chercher la solution du CSP qui maximise une fonction objectif, lorsque les différentes solutions d'un problème sous contraint (admet beaucoup de solutions différentes) ne sont pas toutes équivalentes.

1.5 Exemple de modélisation à l'aide d'un CSP (*problème de coloration de graphe*)

Il s'agit de colorier toutes les régions d'une carte, de telle sorte que deux régions ayant une frontière en commun soient coloriées avec deux couleurs différentes. La figure illustre une instance possible :

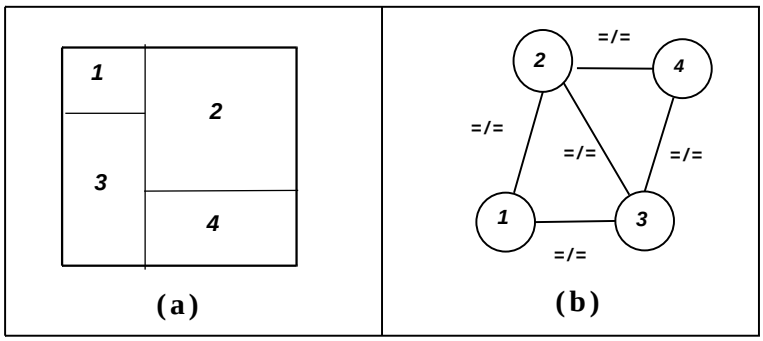

FIG. 1.1 – Problème de coloration de graphe

Ici notre carte est divisé en 4 régions et l'on dispose des trois couleurs Jaune, Rouge et Bleu pour le coloriage. Donc, on associe une variable X_i différente par région i à colorier

- $P = (X, D, C, R)$
- $X = \{X_1, X_2, \ldots, X_4\}$ où X_i correspond au numéro de la région à colorier.
- $D = \{D_1, D_2, \ldots, D_4\}$ avec $D_i = \{j, r, b\}$
- $C = \{(X_i, X_j)\}$ telles que X_i et X_j sont 2 variables de X correspondant à des régions voisines. Et les C_{ij} vérifient :

$$X_i \neq X_j.$$

1.6 Les différentes approches de résolution des problèmes CSP

Pour les problèmes *CSP*, il n'existe pas de méthode universelle pour une résolution efficace. Au cours des vingt dernières années, de nombreux algorithmes et systèmes ont été mis au point pour résoudre ce type de problèmes, notamment en utilisant les techniques de consistance et de recherche avancé, ainsi que des combinaisons de ces techniques pour obtenir des algorithmes plus performants.

Classiquement, on identifie deux grandes familles au sein de ces techniques de résolution. D'une part, les méthodes complètes (ou exactes), d'autre part, les méthodes incomplètes (ou approchées) La figure (Fig. 1.2) illustre les différentes approches de résolution des problèmes *CSP*.

1.6.1 Les méthodes incomplètes

Les méthodes incomplètes considèrent l'espace de recherche dans sa totalité mais elles ne l'explorent qu'en partie en se dotant des combinaisons d'heuristiques pour choisir les zones d'exploration, ces combinaisons sont appelées méta heuristique. Ces méthodes visent à donner un résultat acceptable en un temps raisonnable mais elles ne peuvent prouver l'inexistence de solution d'un problème sur contraint. Autrement dit, elles abordent la résolution d'un *CSP* comme un problème d'optimisation combinatoire pour lequel il s'agit de calculer une affectation satisfaisant le plus grand nombre de contraintes, l'objectif final étant de les satisfaire toutes.

Les différentes approches possibles d'une résolution incomplètes appartiennent a deux grandes familles de métaheuristques, celle de *la recherche locale* avec ses algorithmes : recuit simulé (*RS*), la recherche Tabou, colonies de fourmis (*AC,Ant Columns*), et celle dites *évolutionniste* avec les algorithmes génétiques.

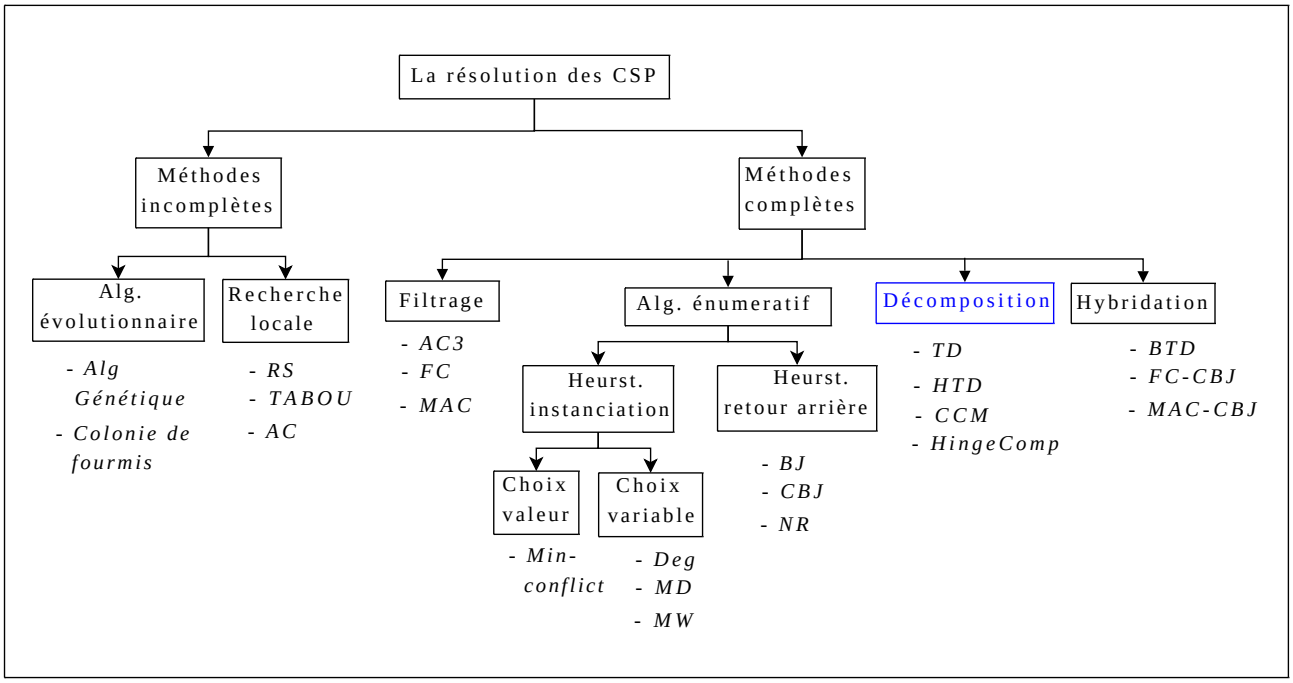

FIG. 1.2 – les différentes approches de résolution des *CSP*

1.6.2 Les méthodes complètes

Les méthodes complètes explorent totalement l'espace de recherche et elles sont toujours capables de répondre par vrai ou faux concernant l'existence d'une solution. En effet, elles peuvent prouver la satisfiabilité d'un problème, tout comme determiner l'ensemble des solutions de ce problème.

Les nombreux travaux qui ont été réalisés dans cet axe et qui tentent d'améliorer ces méthodes de résolution peuvent être classés en trois grandes classes : [27]

1. **Les algorithmes de recherche énumérative**
   Ces algorithmes consistent à visiter toutes les configurations possibles de l'espace de recherche (l'ensemble des affectations possibles des variables). L'algorithme de type *Backtracking* constitue la méthode de base. Cette algorithme, beaucoup trop coûteux, a conduit à la recherche d'algorithmes intélligents les plus efficaces.

2. **Les algorithmes de consistance (filtrage par consistance)**
   Ces algorithmes visent à réduire l'espace de recherche à explorer pour simplifier les instances avant ou pendant la recherche d'une solution. Ils sont utilisés comme des algorithmes de prétraitement pour améliorer le travail des algorithmes de recherche en utilisant des techniques de consistance d'arc, de chemin,...etc.

3. **Résolution par décomposition des *CSP***
   Ces algorithmes exploitent le fait que la traitabilité d'un *CSP* est reliée à ses propriétés structurelles. Etant donné une instance *CSP*, ces algorithmes transforment cette instance en une instance acyclique constituée d'un ensemble de sous problèmes dont la complexité est inférieure à celle de l'instance originale.

Nous nous intéressons dans la suite de ce mémoire aux méthodes complètes c'est pourquoi nous détaillons dans ce qui suit ces algorithmes.

1.7 Résolution séquentielle par des méthodes complètes

La résolution des problèmes CSP par des méthodes complètes se base, généralement, sur des algorithmes de recherche énumérative de type Backtrack [21]. Le Backtrack combiné à des heuristiques (telles que les techniques de retour arrière intelligent et de l'ordre des variables),et à des techniques de filtrage, induit, généralement, de bonnes performances en pratique. La résolution par décomposition présente des bons résultats comme nous allons voir dans la suite.

Dans ce qui suit, nous allons présenter les différents algorithmes de recherche, puis, nous présenterons les différents algorithmes issus de nombreuses recherches qui ont été menées pour améliorer l'algorithme Backtrack et qui peuvent être classées en trois principaux axes [27], en commencant par les algorithmes énumératifs avec heuristiques (sur le retour arrière, l'instanciation,...etc), les algorithmes de filtrage par consistance qui sont utilisés pour rendre le travail des algorithmes énumératifs plus efficace, et en fin, les algorithmes par décomposition qui cherchent a réduire la complexité des *CSP* par leur transformation en des *CSP* acycliques avant de commencer la résolution.

1.7.1 Résolution par des algorithmes énumératifs

Les algorithmes énumératifs effectuent un parcours systématique de l'espace de recherche. La manière la plus simple est de générer toutes les configurations possibles, c'est à dire toutes les combinaisons possibles de valeurs des variables, et de tester si elles vérifient les contraintes. Cette approche est connue sous le nom de Generate and test. Cependant, l'algorithme de base et le plus répandu pour une recherche systématique est celui du Backtrack chronologique [21].

1.7.1.1 Algorithme Backtrack (BT)

Cet algorithme consiste à instancier les variables, les unes après les autres, jusqu'à l'obtention d'une affectation complète. Contrairement à un generate-and-test, pour chaque instanciation de variables, le *Backtrack* vérifie les contraintes dont les variables sont déjà instanciées sur l'affectation partielle courante, si cette affectation partielle viole une contrainte, le *Backtrack* supprime le sous-espace de recherche en dessous du point de choix et affecte une nouvelle valeur a la dernière variable. Si le domaine d'une variable devient vide, l'algorithme effectue un retour arrière chronologique.

Algorithm 1 Algorithme Backtrack

Entrée : Un *CSP* $P = (X, D, C, R)$, et une affectation A de ce *CSP*. Initialement $A = \oslash$.
Sortie : Une solution du *CSP* s'il est consistant, sinon déterminer qu'il est inconsistant.
si A est non consistant **alors**
  Retourner Faux.
sinon
  si A est une affectation totale **alors**
    Retourner vrai /* *A est une solution* */
  sinon
    choisir une nouvelle variable X_i de X
    pour toute valeur V_i de $D(X_i)$ **faire**
      si $Backtrack(A \cup \{(X_i, V_i)\}, (X, D, C, R)) = vrai$ **alors**
        Retourner vrai
      finsi
    fin pour
  finsi
  Retourner Faux.
finsi

Comme nous avons vu, l'algorithme *Backtrack* se base sur une recherche énumérative, en effet, pour un nombre de variables n et une taille des domaines d (le nombre maximum des valeurs par domaine), la complexité en temps de la recherche est bornée par la taille de l'espace de recherche et qui est égale à d^n, par conséquent, pour un problème de m contraintes, la complexité théorique du *Backtrack* est $O(md^n)$. Pour cela, de très nombreuses recherches ont été menées pour améliorer cet algorithme.

1.7.2 Algorithmes énumératifs avec heuristiques

Le premier axe de recherche pour l'amélioration du *Backtrack* repose sur l'utilisation des heuristiques. Deux types d'heuristiques peuvent être envisagées : le premier type

concerne le retour arrière lui-même, et le deuxième type concerne le choix des variables et des valeurs.

Les définitions suivantes sont nécessaires pour assurer la compréhension de la description de ces algorithmes.

Définition 1.12 (*Conflit)*
Un *conflit* est une situation telle que l'affectation d'une valeur $v \in D_i$ à la variable X_i viole une des contraintes C du problème.

Définition 1.14 (*Good, Nogood)*
Le *good* (resp.*nogood*) est une affectation partielle consistante qui peut (resp.ne peut pas) s'étendre en une affectation totale consistante (solution). Un *good* (resp.*nogood*) *minimal* est tout *good* (resp.*nogood*) non composé lui-même d'un *good* (resp.*nogood*).

1.7.2.1 Heuristiques sur le retour arrière

1. ***Algorithmes Backtrack intelligent***
   Le mécanisme de base est le retour arrière chronologique. Selon Dechter et Frost [12], le BT a comme principal problème de retomber sans cesse dans les mêmes situations d'inconsistance. Lorsque un échec survient, l'algorithme Backtrack remet en cause la dernière instanciation et après avoir essayé toutes les valeurs il revient en arrière sur la variable précédente. Cependant, rien ne garantie que cette variable a une quelconque responsabilité dans l'inconsistance, par conséquent, essayer de nouvelles valeurs pour une variable qui n'est pas en cause de l'échec conduira aux mêmes échecs. Le saut en arrière (*Backjump*) est la technique dotée pour pallier à cet inconvénient du Backtrack, elle consiste à faire un retour arrière non chronologique (intelligent) sur la variable qui est en cause dans l'échec, plutôt que d'effectuer un retour en arrière vers la $(i-1)^{ieme}$ variable affectée. Parmi les algorithmes exploitant le retour arrière non chronologique :

   (a) *Algorithme Back Jumping (BJ)*
   L'algorithme *BJ* [19] a été développé pour accélérer le processus de retour arrière. C'est un algorithme qui fait un retour arrière sur la variable la plus profonde qui est en conflit avec la variable courante (dont la valeur est incompatible avec la valeur de la variable courante), si toutes les instanciations de cette dernière sont inconsistantes. Cet algorithme utilise certains formalismes pour déterminer vers quelle variable effectuer un saut lors du retour en arrière.

   (b) *Algorithme Conflict Directed Back Jumping (CD-BJ)*
   L'algorithme *CD-BJ* [12] se comporte d'une façon similaire a *BJ* sauf, pour

le retour arrière, si les extensions de la variable la plus profonde sont toutes essayées, l'algorithme *BJ* fait un Backtrack chronologique par contre le *CBJ* effectue si possible un Back jumping, tout en maintenant pour chaque variable un ensemble de variables instanciées avant cette dernière et avec lesquelles elle est en conflit. Quand il rencontre un échec, *CBJ* revient en arrière jusqu'à la variable la plus profonde dans l'ensemble de conflits de la variable courante.

2. ***Algorithmes Backtrak avec mémorisation***
   Le Backtrack intelligent a amélioré l'algorithme de base (BT) par une optimisation de la recherche en évitant les redondances (les memes situations d'inconsistance). D'autres méthodes pour éviter la redondance, même dans les cas ou les algorithmes de retour intelligent peuvent échoués, ont été décrites. Ces méthodes reposent sur la mémorisation des informations sur les sous arbre déjà visités, ces informations sont souvent appelées de *nogood* (*Définition1.7*). Donc, la mémorisation de celle instanciation partielle qui ne peut pas étendue à une solutionpermet permet de ne pas la régénérer une autre fois et ainsi de ne pas explorer les mêmes sous arbres.

   Cependant, l'inconvénient majeur de ces méthodes est l'espace mémoire requis pour la mémorisation des nogood.

   Parmi les algorithmes qui adoptent la technique de mémorisation, on peut citer :
   – Nogood Recording (NR) [41].
   – Learning Tree-Solve [2].
   – Dynamic Backtracking (DBT) [20].

1.7.2.2 Heuristiques sur l'instanciation des variables

1. ***Choix de variables***
   L'ordre d'instanciation des variables utilisé par une méthode énumérative a une influence sur la taille de l'arbre de recherche. Le choix d'un bon ordre dépend du problème traité. La majorité des heuristiques de choix des variables ont pour objectif de détecter l'échec le plus tôt possible (*fail first* principal). L'ordre peut être statique ou dynamique.

   Dans le cas statique, l'ordre repose sur les propriétés structurelles du problème, et il est prédéfini avant la résolution. *Deg* ou *MinDeg* [13] sont des heuristiques statiques qui utilise le degré de variable (le nombre de contraintes portant sur chaque variable) et il existe d'autres comme : *MinCon* (ordre selon le nombre de variables voisines instanciées) et *MinWid* [14] (ordre selon la largeur).

2. ***Choix de valeurs***
   L'apport des heuristiques de choix de valeur est limité par rapport à celle de variable ce qui explique la rareté des propositions dans cette direction. La meilleure heuristique de choix de valeurs est celle de *min-conflict* [15] qui consiste à instancier la variable courante par la valeur qui a le nombre minimum de valeurs incompatibles avec les valeurs des variables non encore instanciées, ainsi, le choix de valeurs est selon l'ordre croissant des nombres de conflits.

1.7.3 Résolution par des algorithmes de Filtrage par propagation de contraintes

Comme nous avons vu, la complexité du Backtrack est exponentielle en la taille du problème, ce qui rend intéressant de réduire l'espace de recherche en utilisant des techniques de filtrage. Ce filtrage consiste à supprimer les valeurs des domaines ne pouvant satisfaire certaines contraintes et ainsi d'éviter de tester toutes les combinaisons variable-valeur inhérentes à la formulation d'un *CSP*. Les valeurs à supprimer dépendent de la forme de consistance utilisée. Plusieurs formes de consistance ont été proposées. Cependant, la plus utilisé est celle de la consistance d'arc dûe à son efficacité et sa simplicité à mettre en oeuvre.

Le filtrage par consistance d'arc peut être employé comme prétraitement avant la résolution ou pour maintenir le problème consistant durant la résolution, il se base sur la propagation de contraintes, c'est-à-dire, il consiste à supprimer de chaque domaine D_i d'une variable X_i, les valeurs qui ne vérifient pas l'arc consistance, cette suppression peut être propagée vers le voisinage de X_i dans le réseau de contraintes, si la valeur supprimée est la seule valeur avec laquelle une valeur d'une autre variable vérifie la consistance d'arc, cette dernière sera a son tour supprimée. La propagation se termine s'il n y a plus de suppression.

Le premier algorithme qui implémente la consistance d'arc est le *AC1* dû à *Mackworth* [30], sa complexité est de $O(mnd^3)$, avec m le nombre de contraintes, n le nombre de variables et d la taille des domaines. Cette complexité est dûe à des vérifications de consistances inutiles répétées à chaque fois qu'il y a une élimination d'une valeur. Dans le même travail [30], *Mackworth* a élaboré deux autres algorithmes de renforcement de l'arc consistance, *AC2* et *AC3* [30]. Par la suite, un algorithme *AC4* [33] a été développé par *Mohr* et *Henderson* [33] et qui a fait une amélioration notable des algorithmes précédents. D'autres variantes ont été proposés, *AC5* [36], *AC6* [4] et *AC7* [37], pour pallier aux problèmes de complexité en terme d'espace et en terme de temps par la diminution du nombre de vérifications de consistance inutiles. Enfin, *Bessière* et *Régin*

[3] ont développé le *AC2000* et le *AC2001*. Cependant le plus utilisé est le *AC3* (voir *Algorithme2*) dont le complexité est $O(md^3)$.

Algorithm 2 Algorithme Arc Consistent 3 (AC3)

Entrée : Un *CSP* $P = (X, D, C, R)$.
Sortie : Vérifier l'arc consistance du *CSP*.
$Q \leftarrow \{(i,j)/(i,j)$ est un arc et $i \neq j\}$
tantque Q est non vide **faire**
  Extraire un couple *(k,m)* de Q
  si $Revise(k,m)$ **alors**
    $Q \leftarrow Q \cup \{(i,k)/(i,k)$ est un arc et $i \neq k$ et $i \neq m\}$
  finsi
fin tantque

Fonction Révise (i, j) : Booléen
Suppression <- Faux
pour chaque valeur x de domaine (i) **faire**
  si il n'existe pas de support dans le domaine (j) **alors**
    supprimer x de domaine (i)
  finsi
  Suppression <- Vrai
fin pour
Retourner Suppression

Etant donné un CSP, un solveur a pour objectif de trouver une affectation pour chaque variable satisfaisant toutes les contraintes, l'idée est alors de réduire l'espace de recherche. (l'espace des possibilités).

Pour ce faire, les algorithmes de résolution consistent à supprimer des valeurs dans les domaines, dites inconsistants.

Algorithme AC

Pour illustrer l'arc consistance, nous présentons l'algorithme $AC3$ qui est le plus répandu qui est en $O(md_3)$.

1. **Algorithme Forward Checking (FC) [23]**
   L'algorithme *FC* est une amélioration du Backtrak en appliquant un filtrage avant (anticipé). Il vérifie la consistance entre la variable courante et celles du futur qui ne sont pas encor instaciées, en filtrant leurs domaines par la suppression des valeurs incompatibles avec la valeur courante. Autrement dit, on évalue l'impact d'une instanciation avant même que la décision ne soit prise " *Lookahead and anticipate the future in order to succeed in the present* ". En effet, il ne construit que des affectations consistantes (succès) car les affectations inconsistantes sont, en fait, éliminées par le filtrage anticipé (Lookahead) (voir *Algorithme3*).

Algorithm 3 Algorithme Forward Checking (FC)

Entrée : Pour un *CSP* $P = (X, D, C, R)$, l'entrès est Le quadreplet (A, NA, D, C), avec A est une affectation, initialement vide ($A = \{\}$), NA est l'ensemble des variables non encore instanciers, initialement $NA = X$, D l'ensemble des domaines et C celui des contraintes.
Sortie : Une solution du *CSP* si le problème est consistant, sinon déterminer que le *CSP* est inconsistant.
si $NA = \{\}$ **alors**
  Retourner A /* *A est une solution*/
sinon
  Choisir x de NA
  répéter
    Choisir une valeur v de D_x ; $D_x = D_x - \{v\}$;
    si $A \cup \{x, v\}$ ne viole aucune contrainte **alors**
      $D' = Revise(NA - \{x\}, D, C, < x, v >)$
      si aucun domain de D' n'est vide **alors**
        $Result \leftarrow FC(NA - \{x\}, A+ < x, v >, D', C)$
        si $Result \neq NULL$ **alors**
          Retourner $(Result)$
        finsi
      finsi
    finsi
  jusqu'à $D_x = \{\}$
  Retourner (NULL)
finsi

Procedure Revise (W, D, C, a)
a : est une affectation d'une variable.
D' = D
pour chaque variable y dans W **faire**
  pour chaque valeur v dans D'_y **faire**
    si $< y, v >$ est incompatible avec a en respect avec les contraintes de C **alors**
      $D'_y = D'_y - \{v\}$
    finsi
    Retourner D'
  fin pour
fin pour

L'algorithme 3 présente l'algorithme *FC* pour des *CSP* binaires. La complexité en temps de *FC* dans le pire des cas est identique à celle de *BT*, c'est-à-dire en $O(md^n)$. Par contre, en pratique, *FC* obtient généralement de meilleurs résultats que *BT*.

Des généralisation de *FC* au cas des *CSP* n- aires *(nFC2, nFC3, nFC4, nFC5)* [6] sont mises en oeuvre.

2. **Algorithme Maintaining Arc Consistency (MAC) [40]**
   Dans cet algorithme, la consistance d'arc est maintenue tout au long de la recherche. Contrairement à *FC* qui limite la vérification de consistance au voisinage de la dernière variable instanciée, le *MAC* propage cette vérification pour toutes les variables du *CSP*, celles qui sont en relation avec la variable instanciée et celles qui ne le sont pas, tout en garantissant un maintient de la consistance. Ainsi, à chaque étape, le *MAC* applique un algorithme de filtrage par arc consistance *AC*. La maintenance de consistance est une tache coûteuse ce qui rend le *MAC* plus lourd que le *FC*.

Algorithm 4 Algorithme MAC

Entrée : Pour un *CSP* $P = (X, D, C, R)$, l'entrès est Le quadreplet (A, NA, D, C), avec A est une affectation, initialement vide ($A = \{\}$), NA est l'ensemble des variables non encore instanciers, initialement $NA = X$, D l'ensemble des domaines et C celui des contraintes.
Sortie : Vérfier l'arc consistance du *CSP*.
si $NA = \{\}$ **alors**
  Return A /*A est une solution*/
sinon
  Choisir une variable x de NA
  répéter
    Choisir une valeur v de D_x ; supprimer v de D_x
    si $A+ < x, v >$ est consistante **alors**
      $D' = Rvise(NA - \{x\}, D, C, < x, v >)$; //même fonction que celle de FC
      $AC - X(NA - \{x\}, D', C)$; D' *est filtré à l'aide d'un l'algo AC*
      si aucun domaine de D' n'est vide **alors**
        $Result = MAC(A+ < x, v >, NA - \{x\}, D', C)$
        si $Result \neq NULL$ **alors**
          Return $Result$
        finsi
      finsi
    finsi
  jusqu'à $D_x = \{\}$
  Return NULL
finsi

Il existe plusieurs versions du *MAC* selon l'algorithme *AC* utilisé, *MAC3* s'il

utilise le *AC3*, *MAC 2001* pour *AC 2001* est ainsi de suite.

1.7.4 Résolution par décomposition

Les méthodes de résolution des problèmes de satisfaction de contraintes, présentées jusqu'ici, se basent sur des techniques de recherche de type *Backtrack* dont la complexité temporelle et spatiale restent exponentielles en la taille de l'instance du *CSP* traitée. Pour l'amélioration de cette complexité, des techniques d'optimisation par filtrage (arc consistance,etc), par heuristiques (retour arrière intelligent, mémorisation,. . .etc) et par le choix de l'ordre des variables et des valeurs induisent des bonnes performances en pratique. Cependant, elles ne prennent que très peu en compte les particularités structurelles des problèmes *CSP*.

L'exploitation des caractéristiques structurelles de l'instance traitée a donné naissance à une nouvelle approche de résolution qui est la résolution par décomposition. Cette approche exploite le fait que la traitabilité d'un CSP est liée aux caractéristiques topologiques de son réseau de contraintes (graphe ou hypergraphe), ce résultat théorique et formalisé par un théorème du a freuder [14] et qui peut être résumé comme suit :

" Si un réseau de contraintes est acyclique et arc consistant, il peut être résolu en appliquant un algorithme de degré polynomial (Backtrack free) ".

La résolution par décomposition est une téchnique qui analyse les CSP et définit un schéma de décomposition avant la recherche d'une solution. Autrement dit, elle utilise la décomposition comme un prétraitement pour la résolution.

Plusieurs méthodes de décomposition ont été proposées dans la littérature (voir chapitre 2),par conséquent, des algorithmes de résolution par décomposition, qui utilisent ces méthodes, ont été mis au point (voir chapitre 3).

1.7.5 Résolution par des algorithmes hybrides

La combinaison des trois approches de résolution (énumérative, par filtrage et par décomposition) induit des algorithmes de résolution dites hybrides. Plusieurs algorithmes hybrides ont été proposés :

BTD [26] pour Backtracking et Tree Decomposition, un algorithme qui fait l'hybridation entre un algorithme énumératif (*BT*) et la résolution par décomposition (*Tree Decomposition*), proposé par Cyril Terrioux. L'hybridation entre les algorithmes énumératif et ceux de filtrage se voit dans le *FC* et le *MAC*. FC-MRV est un algorithme qui hybride l'algorithme FC et l(heuristique MRV. Il est avéré meilleur que MAC en pratique.

1.8 Conclusion

Dans ce chapitre, nous avons présenté un bref état de l'art sur le formalisme *CSP* ainsi que les différentes approches de résolution, en nous focalisant sur les méthodes dites *complètes*, énumératives et structurelles.

Dans le cadre enumératif, Nous avons commencé par détailler quelques uns des algorithmes naifs (BT), puis nous avons montré les differentes approches pour les résoudre.

Dans le deuxième chapitre, nous allons mettre l'accent sur les différentes méthodes de décomposition structurelles apparues existe dans la littérature en présentant leurs fonctionnement ainsi qu'une comparaison qui permettent de les classer.

Chapitre 2

les méthodes de décomposition structurelle

Les méthodes énumératives de résolution des problèmes de satisfaction de contraintes présentées au chapitre précédent, si elles sont efficaces pour la plupart des problèmes, présentent cependant une lacune, celle de n'exploiter que faiblement la structure de l'instance à résoudre. Pourtant, la structure du graphe de contraintes, si elle est particulière, et exploitée à bon escient, peut servir de guide efficace pour la recherche de solutions, qui est le cas pour les *CSP* acycliques dont la structure peut être représentée sous forme arborescente.

les méthodes de décomposition sont des techniques dont le but est de transformer n'importe quel *CSP* en un *CSP* acyclique et de borner ainsi sa complexité théorique.

L'objectif de ce chapitre est d'introduire les techniques de décomposition des *CSP* les plus connues. Pour ce faire, nous commencerons par présenter les instances de *CSP* traitables ainsi que quelques définitions sur les réseaux de contraintes, puis nous présenterons le principe de chacune des méthodes de même que les différents concepts correspondants, et en terminant par une classification de ces méthodes selon des critères prédéfinis.

2.1 Introduction

De nombreux problèmes de satisfaction de contraintes issus du monde réel ont une structure topologique assez particulière représentée par un graphe de contraintes. Au niveau théorique, des travaux ont porté sur la recherche d'algorithmes tenant en compte ces particularités topologiques des problèmes *CSP*.

La classe des *CSP* acycliques est l'une des classes dites traitables, pour lesquelles des algorithmes de résolution polynomiales existent. Pour profiter de cette propriété,

des méthodes de décomposition arborescente ont été introduites pour transformer un *CSP* dont le graphe de contraintes est cyclique en un autre *CSP* équivalent dont le graphe de contraintes est acyclique, ceci avant la résolution qui se fait par un algorithme polynomial.

Les heuristiques d'ordonnancement de l'affectation des variables font les premiers résultats de la décomposition présentés par *Freuder* [14]. Dans cet article [14], *Freuder* a introduit une procédure de pré-traitement dont l'objet est de construire un bon ordre sur les variables avant l'exécution d'une procédure de recherche. Il a par la suite présenté une condition suffisante pour éviter des backtrack durant la recherche. Cette condition est liée à la propriété structurelle du graphe de contraintes.

2.2 Complexité des CSP

Le problème de satisfaction de contraintes se définit par la donnée d'une instance $P = (X, D, C, R)$ et par la question :*l'instance P est-elle cohérente ?*.

Il s'agit d'un problème de décision qui est *NP-complets* [30]. Donc, le test d'existence de solution pour un *CSP* est un problème *NP-complet*, cependant, le problème de recherche pour le *CSP* est *Difficile*.

2.2.1 NP-complétude ?

La classe des problèmes *NP-complets* recouvre une très grande partie dans le domaine de recherche Informatique : *SAT* (satisfaction d'expressions booléennes), *CSP* (problèmes de satisfaction de contraintes), Programmation Linéaire, Programmation par Contraintes, Théorie des Graphes, Combinatoire,. . .etc, et elle se définit ainsi :

Etant donné un ensemble fini d'éléments et une propriété logique facile à vérifier. la *NP-complétude* est la question de savoir s'il existe ou non un élément de cet ensemble satisfaisant cette propriété, il s'agit d'un problème décisionnel.

Pour fournir des bornes de complexité meilleures, plusieurs travaux ont été développés pour définir des classes de *CSP* traitables.

2.3 Traitabilité

La méthode de base pour la résolution des *CSP* est fondée sur l'énumération de type backtracking. Cette approche, quoi que souvent efficace en pratique a une complexité

théorique de l'ordre de $O(md^n)$ où n : nombre de variables, m : nombre de contraintes et d la taille maximale des domaines des variables.

Beaucoup d'efforts ont été fournis afin de démontrer que pour certaines classes de problèmes, il est possible de développer des algorithmes polynomiaux en la taille du problème. Ces classes sont dites *traitables*.

La traitabilité des *CSP* est liée soit à la structure topologique du réseau de contraintes (le graphe ou l'hypergraphe de contraintes associé au *CSP*), soit aux relations des contraintes. Donc, les travaux portés sur la recherche des classes traitables peuvent être divisés en deux grandes approches, les techniques qui se basent sur la topologie et celle qui se basent sur les relations.

2.3.1 Traitabilité dûe à la structure

Cette approche inclue toutes les classes traitables des *CSPs* qui sont identifiés uniquement par leurs ensembles de contraintes indépendamment de leurs ensembles de relations. Une de ces classes est celle des *CSP* ayant des structures acycliques.

2.3.2 Traitabilité due aux relations

Cette approche inclue toutes les classes traitables dûe à des propriétés particulière de leurs ensemble de relations.

Dans la suite de notre travail, nous nous intéresserons, uniquement, à la traitabilité dûe à la structure du problème. Pour cela on peut définir la traitabilité ainsi :

Definition 2.1 (*Traitabilité)*
Une classe I des instances des *CSP* est dite traitable s'il existe :

1. Un algorithme, en temps polynomial, qui permet de décider si une instance du *CSP* P ($P \in I$), peut avoir une décomposition et la calculer.
2. Un algorithme, en temps polynomial, qui résout le *CSP* P.

2.4 Réseaux de contraintes

La structure d'un problème *CSP* est visualisée par son réseau de contraintes, et selon l'arité des contraintes (*Définition 1.3*), son réseau peut être un graphe ou un hypergraphe.

Definition 2.2 (*Graphe)*
Un graphe $G = (V, E)$ est une structure constituée d'un ensemble fini de noeuds $V = \{v_1, \ldots, v_n\}$, et d'un ensemble d'arêtes $E = \{e_1, \ldots, e_m\}$. Chacune des arêtes e_i relie une paire des sommets $\{u, v\}$.

Définition 2.3 (*Hypergraphe)*
Un hypergraphe est une structure $H = (V, S)$ qui est constituée d'un ensemble de sommets $V = \{v_1, \ldots, v_n\}$ et d'un ensemble de sous ensembles de ces noeuds ou clusters $S = \{s_1, \ldots, s_m\}$, $s_i \subset V$, appelés hyprarêtes. Une hyperarête est une arête définie sur plus de deux noeuds. Notons qu'un graphe peut être vu comme un hypergraphe dont les hyper-arête relient au plus deux noeuds (voir Fig. 2.1*(a)*).

Definition 2.4 (*Gaifman graphe, graphe primal)*
Soit $H = (V, S)$ un hypergraphe. Le graphe de $Gaifman$ ou le graphe *primal* de H est un graphe obtenu à partir de H comme suit :(voir Fig. 2.1*(b)*)

1. Le graphe primal possède le même ensemble de noeuds que celui du H.
2. deux noeuds v_i et v_j sont connéctés par une arête dans le graphe primal si et seulement si v_i et v_j appartiennent à la même hyperarête dans H.

Définition 2.5 (*graphe dual)*
Le graphe dual H_{dual} d'un hypergraphe $H = (V, S)$ est un graphe dont les noeuds sont les hyperarêtes de H, et deux noeuds sont connectés dans H_{dual} si leurs hyperaretes correspondantes partagent un noeud dans H. chaque noeud du graphe dual est etiquetté par les noeuds de l'hyperarête correspondante dans H (voir Fig. 2.1*(c)*).

Définition 2.6 (*Connectivité)*
Etant donné un graphe dual d'un hypergraphe, un sous graphe du graphe dual vérifie la propriété de connectivité (*connectedness*) ssi pour chaque paire de noeuds partageant des variables, il existe au moins un chemin d'arcs étiqueté par les variables partagées entre ces noeuds.

Définition 2.7 (*Join Tree)*
Le $JoinTree$ est un $Joingraph$ dont la structure est un arbre. Un $Joingraph$ est un sous graphe du graphe dual qui vérifie la propriété de connectivité.

Définition 2.8 (*CSP acyclique)*
Un *CSP* qui possède un *Join Tree* est appelé un *CSP acyclique*.

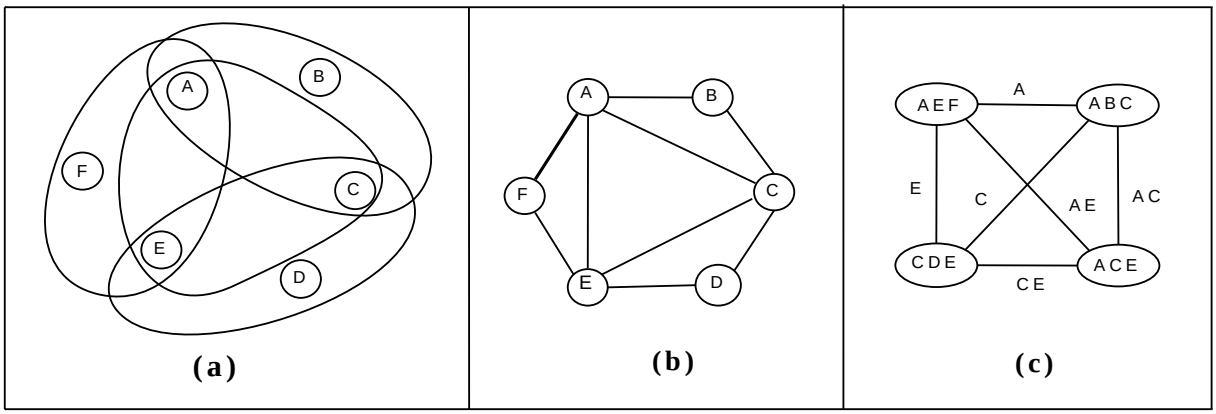

FIG. 2.1 – (a) hypergraphe d'un CSP (b) Son graphe primal (c) Son graphe dual

2.5 Les méthodes de décomposition structurelles

Les méthodes de décomposition fonctionnent selon le même principe que ce soit pour des *CSP binaires* ou *n-aires*. Notons que dans le cas ou il existe des contraintes d'arité supérieure strictement à deux, il existe des méthodes spécifiques fondées sur la décomposition de l'hypergraphe de contraintes, et qui sont plus efficaces que les méthodes fondées sur le graphe primal.

2.5.1 Principe de base

L'objectif d'une méthode de décomposition est de transformer une instance *CSP* cyclique en une autre instance équivalente (ayant les mêmes solutions) acyclique, qui peut être résolue d'une manière plus efficace, selon le théorème du *Freud*[14], et comme il a été démontré que la classe des *CSP* dont le graphe de contraintes est un arbre est traitable. Informellement, celle-ci est assurée par la décomposition du réseau de contraintes du problème *CSP* donné en un ensemble de sous problèmes, en formant des clusters de variables ou de contraintes, dont l'interaction a une structure d'arbre. Si la taille de chacun des sous problèmes est plus petite que celle du *CSP* original, la résolution des sous problèmes (clusters) peut être réalisé d'une manière plus efficace que la résolution du problème original. Une solution du *CSP* original peut être dérivée de l'arbre de sous problèmes, d'une manière aussi plus efficace.

Chaque méthode de décomposition définit un concept de *largeur* (*width*) qui est le critère de la mesure de cyclicité du graphe ou de l'hypergraphe de contraintes, telle que pour chaque constante k fixée, une méthode de décomposition peut décider, en un temps polynomial, si un hypergraphe donné peut avoir une décomposition de largeur inférieur ou égale à k. Après la décomposition, la complexité de la résolution du *CSP*

est en fonction de la largeur de la décomposition. L'objectif principal de toutes les méthodes de décomposition est donc de minimiser cette largeur.

2.5.2 Méthode Biconnected components

La méthode *Biconnected components* repose sur le concept de la *composante biconnexe* définie ainsi :

Définition 2.9 (*composante biconnexe*)

Une composante biconnexe d'un graphe est un sous graphe connecté qui ne contient pas de noeud séparateur. Un noeud séparateur d'un graphe est un noeud dont l'extraction décompose le graphe (le décompose en des composantes connexes).

Cette méthode consiste à chercher des composantes biconnexes dans le graphe de contraintes de l'instance à résoudre. En effet, elle transforme un graphe $G = (V, E)$ en un arbre T tout en associant à chaque noeud de T un sous ensembles de sommet de V formant une composante binconnexe de G. Deux noeuds de T sont connectés par un arc s'ils partagent le même séparateur (Fig. 2.2).

La méthode *Biconnected components* est une méthode développée pour les *CSP* binaires. Ainsi, la décomposition *Biconnected* d'un *CSP* n-aires est la décomposition *Biconnected* de son graphe primal.

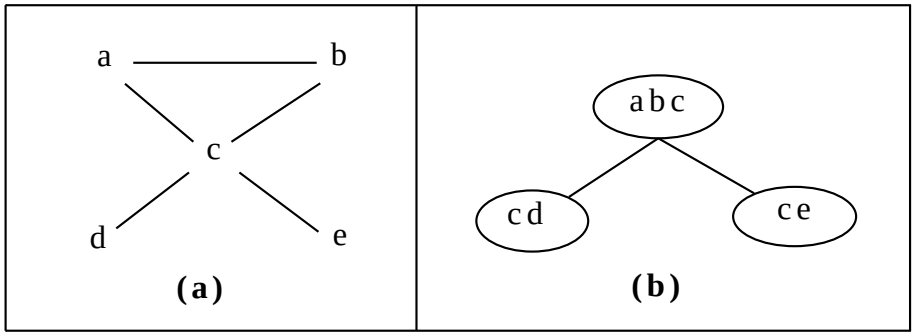

FIG. 2.2 – Un graphe (a) et sa Biconnected décomposition (b)

2.5.3 Méthode Cycle Cutsest

La définition de la méthode *cyclecutset* est différente par rapport aux autres méthodes dont le principe est de générer un arbre. Le résultat de cette méthode n'est

pas un arbre mais, plutot, un ensemble de variables (du cutset) et un arbre (formé des variables qui ne sont pas dans le cuset) en se basant sur la définition suivante :

Définition 2.10 (*Cycle Cutset*)

Etant donné un graphe de contraintes, l'élimination de certaines variables après les avoir instanciées change la connectivite du graphe. Cette ensemble de variables est appelé coupe cycle ou *CycleCutset*.

Le principe de la méthode *cyclecutset* repose sur cette notion et peut être résumé ainsi et initialement, on définit un ensemble cutset pour le réseau de contraintes. On instancie les variables de cette ensemble par une affectation consistante en utilisant un algorithme quelconque. Le résultat après élimintion des variables instanciées, est un réseau de contrainte acyclique (Fig 2.3(c)) qui sera résolu par propagation de consistance en utilisant un algorithme polynomial selon le théorème de *Freuder*[14]. Si cette instanciation de l'ensemble cutset n'aboutit pas à une solution on fait un backtrack sur les variables de cet ensemble (voir Fig 2.3).

La largeur d'une décomposition *CycleCutset* est égale à la taille de l'ensemble couple cycle, et la largeur (width) d'une instance *CSP* est égale à la largeur minimale des décompositions *CycleCutset* de cette instance.

Cette méthodes a été développée pour des *CSP* binaires. Pour les CSP n-aires le même principe s'applique au graphe primal du *CSP*.

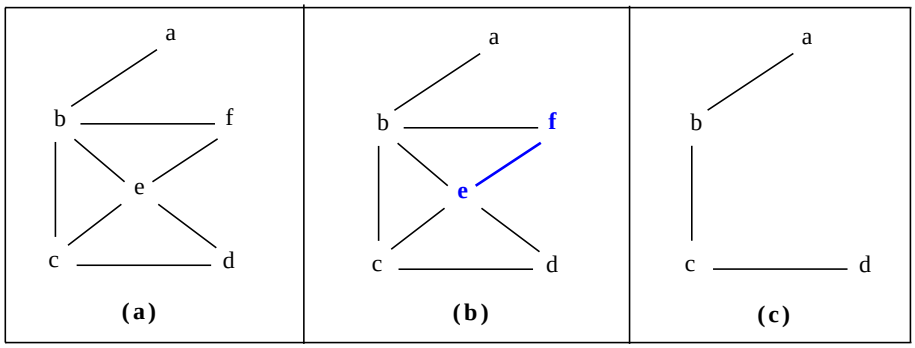

FIG. 2.3 – Un graphe et sa décomposition *Cycle Cutset*

Dans l'exemple de la figure Fig 2.3, l'instanciation de l'ensemble $\{e, f\}$ dans (b) du graphe G dans (a) rend le graphe acyclique (c).

2.5.4 Méthode Hypercutset

La méthode *Hypercutset* est une variante de la méthode *Cycle Cutset* qui s'applique aux *CSP* n-aires en utilisant la définition de Cutset pour un hypergraphe.

Définition 2.11 (*Cycle HyperCutset*)

Un ensemble *Hypercutset* d'un hypergraphe est un ensemble d'hyperarêtes (au lieu de sommets) dont l'extraction des variables impliquées par ces hyperarêtes induit un graphe acyclique.

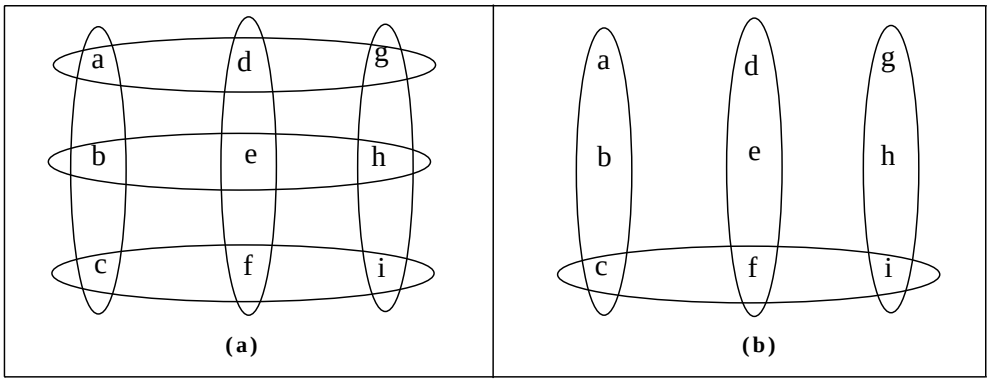

FIG. 2.4 – Un graphe et sa décomposition *Cycle HyperCutset*

La figure Fig 2.4 illustre un exemple d'une decomposition *Cycle Hypercutset* (b) d'un hypergraphe (a). Il est évident que pour rendre cet hypergraphe acyclique, il faudra supprimer deux hyperarêtes horizontales ou verticales. Après avoir choisir le plus petit ensemble hypercutset, on procède à la résolution du *CSP* de la même manière que celle de la méthode Coupe Cycle.

La difficulté de cet méthode réside dans la recherche du plus petit ensemble hypercutset qui est, en général, NP-Complet. Cependant, la recherche du plus petit ensemble hypercutset dont la taille est inférieur à k, s'il existe, est traitable [49].

La largeur *Hyper Cutset width* d'une instance *CSP* est égale à la cardinalité minimale de l'ensemble hypercutset des décompositions *Hyper Cutset* de cette instance.

2.5.5 Méthode Hinge decomposition

La méthode de *Hinge decomposition* propose une autre caractérisation de l'acyclicité en se basant sur le concept des *Hinges*. Elle definit des Hypergraphes acycliques en exploitant, directement, des hypergraphes sans passer par le graphe primal.

Définition 2.12 (*Composante connectée*)

Soit $G = (V, E)$ un hypergraphe, et soit $H \subseteq E$ et $F \subseteq (E - H)$. F est dit *connecté par rapport à* H si pour chaque paire d'arêtes $\{e, f\}$ de F, il existe une séquence d'arêtes $e_1, \ldots, e_n$ dans F, tel que $e = e_1$ et $f = e_n$ et pour $i = 1$ à $n-1 : [(e_i \cap e_{i+1}) - (\cup H)] \neq \phi$.

Définition 2.13 (*Hinge*)

Soit $G = (V, E)$ un hypergraphe réduit et connexe et soit $H \subseteq E$ contenant au moins deux arêtes. Soient $H_1, \ldots, H_m$ les composantes connexes de $(E - H)$ par rapport à H. H est dit un *Hinge* si, pour $i = 1$ à m, il existe une arête h_i telle que : $(\cup H_i) \cap (\cup H) \subseteq h_i$. Un hinge est dit *minimal*, s'il ne contient aucun autre hinge.

Une *Hinge decomposition* d'un hypergraphe $G = (V, E)$ est un arbre vérifiant les quatres conditions suivantes :

1. Les noeuds de l'arbre sont les hinges minimaux de G.
2. Chaque arête dans E est contenue dans au moins un noeud de l'arbre.
3. Deux noeuds adjacents A et B de l'arbre partagent précisément une arête L dans E. De plus, L consiste en l'ensemble des variables partagées par A et B.
4. Les sommets de V partagés par deux noeuds sont entièrement contenus dans tout noeud sur le chemin les liant.

L'*Algorithme 5* [8] illustre les différentes étapes à suivre pour calculer la *Hinge decomposition* ou *Hinge Tree* d'un hypergraphe $G = (V, E)$ donné en entrés. Sa complexité est de l'ordre de $O(|V||E|^2)$.

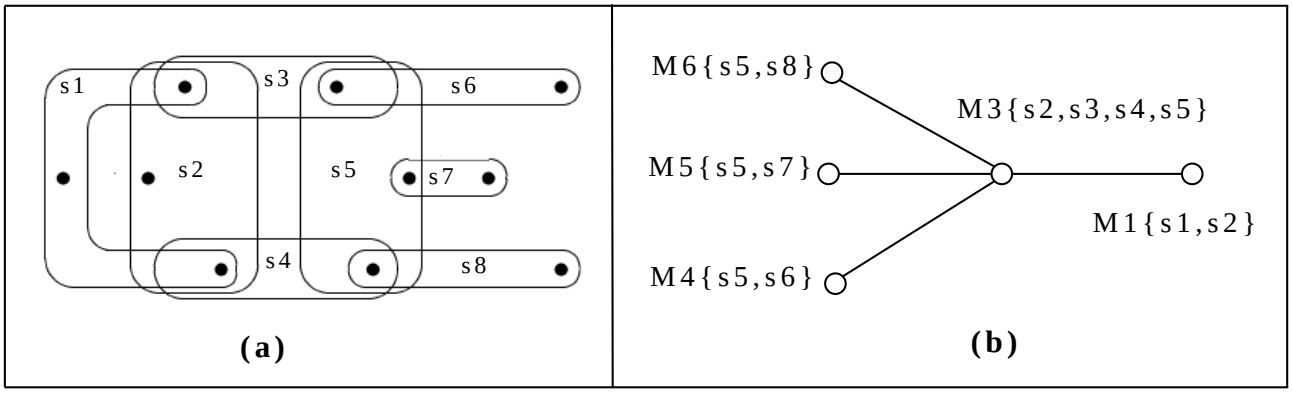

FIG. 2.5 – Un hypergraphe (a) et son *Hinge Tree* (b)

La figure Fig 2.5 illustre un exemple de cette méthode de décomposition, elle montre (a) un hypegraphe, (b) son *Hing Tree*. Cet *Hinge Tree* n'est pas le seul, on peut avoir un autre du meme hypergraphe en remplacant M_1 par $M_3 = \{s_1, s_3, s_4, s_5\}$.

Proposition 2.1
Soit $G(V, E)$ un hypergraphe, et soit T un *Hing Tree* de H, la taille du noeud le plus large de T est égale a la taille de la hinge minimale la plus large de H, celle-ci mesure le degré de cyclicité de H.

La largeur *Hinge width* de cette décomposition est déterminée par le dégré de cyclicité de l'hypergraphe à décomposer. Donc, si le degré de cyclicté est grand les sous problèmes deviennent plus larges et il faut faire recours vers d'autres méthodes pour les résoudre.

2.5.6 Méthode de décomposition arborescente (Tree decomoposition

Définition 2.14 (*Arbre (Tree)*) [18]

Soit $G = (V, E)$ un hypergraphe. Un *arbre (Tree)* de l'hypergraph G est une paire $\langle T, \chi \rangle$, ou $T = (V(T), E(T))$ est un arbre enraciné, et χ est une fonction qui associe à chaque noeud $p \in V(T)$ l'ensemble de variables $\chi(p) \subseteq V$.

Définition 2.15 (*Tree decomposition*) [10]

Une tree décomposition d'un hypergraphe $G = (V, E)$ est un cuple couple $TD =< T, \chi >$ de G qui respecte les conditions suivante :

1. $\forall h \in E$, il existe $t \in T$ tel que $h \subseteq \chi(t)$.
2. $\forall x \in V$, l'ensemble $\{t \in T / x \in \chi(t)\}$ induit un sous arbre connecté de T.

La première condition assure que chacune des contraintes du CSP original doit apparaitre dans, au minimum, un sous problème du nouveau CSP acyclique, et la deuxième condition, garantit que toute variable doit avoir la même valeur affectée dans chaque sous problème dans lequel elle apparait (propriété de connectivité).

Le concept de *Tree decomposition* a été présenté par *Robertson et Seymour* [39] et n'a été, initialement, défini que pour les graphes. Comme chaque graphe peut être vu comme un hypergraphe avec deux sommets dans chacune de ses hyperarêtes, la définition 2.14 étend le concept de *Tree décomposition* aux hypergraphes. Une autre correspondance entre la *Tree decomposition* des graphes et des hypergraphes peut être donnée par le lemme suivant [28].

Lemme 2.1 [28]

Un arbre *(tree)* $\langle T, \chi \rangle$ est une *Tree decomposition* d'un hypergraph H si et seulement si est une *Tree decomposition* de son graphe primal.

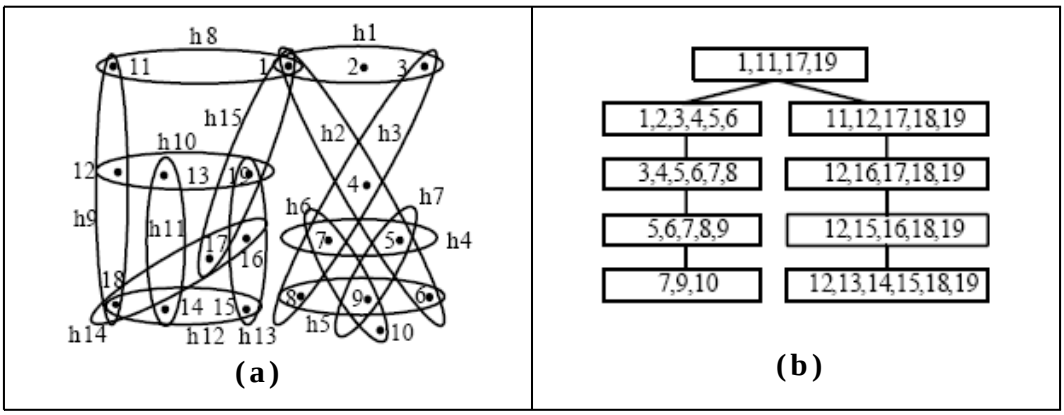

FIG. 2.6 – Un hypergraphe (a) et sa *Tree decomposition* (b)

La figure Fig. 2.6 montre, (a) un hypergraphe H constitué de 15 hperarêtes et 19 sommets, (a) sa Tree decomposition.

La largeure *width* d'une *Tree decomposition* $\langle T, \chi \rangle$ est le $max_{p \in V(T)} \mid \chi(p) - 1 \mid$. La *Treewidth* de H est la largeur *(width)* minimum de toutes les *Tree decompositions*.

Etant donné un *CSP*, son graphe de contrainte, et sa *Tree decomposition* de largeur k, une solution de ce *CSP* peut être calculée en un temps $O(nd^{k+1})$, avec n est le nombre de variables du *CSP* et d le nombre maximum de valeurs par domaine [10].

La *Tree decomposition* est calculée par différentes méthodes la plus importante est la méthode *Tree clustering*.

2.5.7 Méthode Tree clustering

Cette méthode est proposée pour des problème *CSP* naires. Elle consiste a former des *clusters* de variables du problème original pour avoir des sous problèmes structurés en arbre. Elle ne manipule pas l'hypergraphe de l'instance à traiter mais, plutot, son graphe primal (*Définition 2.4*) en se basant sut la concept suivant :

Définition 2.16 (*Graphe triangulé*)
Un CSP est *acyclique* si et seulement si son graphe primal est *Chordale* (*triangulé*) et *conforme*.

Définition 2.17 (*Graphe conforme*)
Un graphe est dit *conforme* (*triangulé*) si tout cycle de longueur supérieure à trois admet une corde. i.e une arête joignant deux sommets non consécutifs le long du cycle.

Définition 2.18 (*Graphe chordal*)
Un graphe primal est dit *conforme* si chacune de ses cliques maximales correspond à une contrainte dans le *CSP* d'origine.

Cette méthode est basée sur un algorithme efficace de triangulation [43], qui transforme n'importe quel graphe en un graphe triangulé en lui ajoutant des arêtes. Les cliques maximales du graphe triangulé obtenues correspondent aux clusters nécessaires pour former un *CSP* acyclique. Cet algorithme de triangulation consiste en deux étapes :

1. Calculer un ordre en utilisant l'algorithme *maximum cardinality search (MCS)* qui numérote les sommets de 1 à n dans l'ordre croissant, en assignant, toujours, le prochain numéro au sommet ayant le plus de voisins numérotés.
2. Relier les noeuds non adjacents connectés par des noeuds dont le numéro d'ordre est inférieur à ceux des deux sommets, et ceci d'une manière récursive.

Si aucune arête n'est ajoutée à l'étape 2, le graphe est triangulé.

Lafigure Fig.2.7 illustre un exemple d'une décomposition *Tree clustring*, elle montre un graphe primal *(a)* d'un problème *CSP*, un ordre de variables possible *(b)* par l'algorithme *MCS*, le graphe triangilé *(C)*, son graphe dual *(d)* et son Join Tree *(e)*.

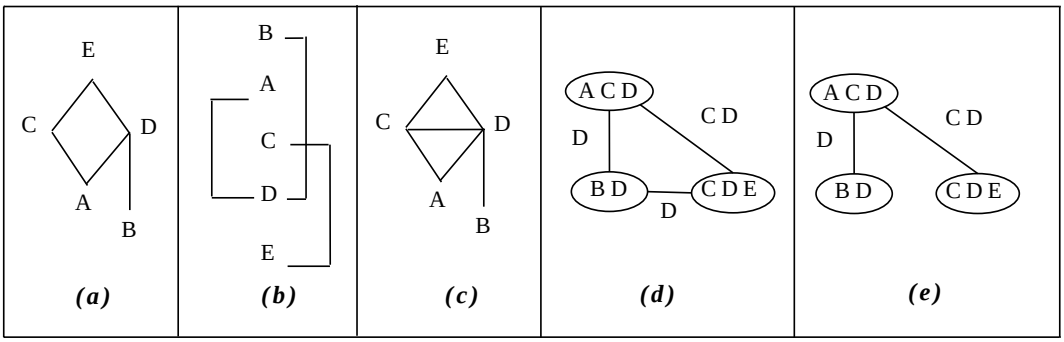

FIG. 2.7 – Etapes de décomposition *Tree clustring*

la compléxité de cette méthode est de $O(nrk^r)$ avec k la taille maximale des domaines, et r le nombre de variables dans la clique maximale la plus large du graphe primal. la complexité est, donc, exponentielle en r.

2.5.8 Méthode hypertree decomposition

Dans [18], *Gottlob et al.* ont proposé une nouvelle méthode de décomposition appelée *hypertree decomposition*. Cette méthode est, à l'origine, introduite dans la théorie des bases de données comme une méthode de décomposition des requêtes. Cependant, dans [17], *Gottlob et al.* ont montré que la *decomposition hypertree* peut etre appliquée dans le domaine des *CSP*.

Définition 2.19 (*Hyperarbre (Hypertree)*)

Soit $H = (V, E)$ un hypergraphe. Un *Hyperarbre (Hypertree)* de l'hypergraph H est un triplet $\langle T, \chi, \lambda \rangle$ ou $T = (V(T), E(T))$ est un arbre anraciné et χ et λ sont deux fonctions qui associent à chaque noeud $p \in V(T)$ deux ensembles $\chi(p) \subseteq V$ et $\lambda(p) \subseteq E$.

Définition 2.20 (*Generalised Hypertree decomposition*)

Une décomposition hypertree généralisée d'un hypergraphe $H =< V, E >$ est un hyperarbre *(hypertree)* $GHD =< T, \chi, \lambda >$ de H qui satifait les conditions suivantes :

1. $\forall t \in T : \chi(t) \subseteq (\cup\lambda(t))$.
2. $\forall h \in E$, il existe $t \in T$ tel que $h \subseteq \chi(t)$.
3. $\forall x \in V$, l'ensemble $\{t \in T / x \in \chi(t)\}$ induit un sous arbre connecté de T.

La première et la deuxième condition de la *décomposition hypertree généralisée* sont identiques à celles de tree decomposition, donc une *décomposition hypertree généralisée* d'un hypergraphe H est, en même temps, une tree decomposition de H. La troisième condition assure que pour chaque noeud de la *décomposition hypertree généralisée*, chacune des variables de l'ensemble χ doit être contenue par au moins une hyperearête de l'ensemble λ de ce noeud.

Définition 2.21 (*Hypertree decomposition*)

Une *décomposition hypertree* [16] d'un hypergraphe $H = (V, E)$ est une décomposition hypertree généralisée $HD =< T, \chi, \lambda >$ de H qui vérifie la condition suivante :

$\forall t \in T : \chi(T_t) \cap (\cup\lambda(t)) \subseteq \chi(t)$ tel que T_t est le sous arbre de T enraciné en t.

La figure Fig. 2.8 illustre, (a) la décomposition hypertree généralisée, et (b) la décomposition hypertree de l'hypergraphe dans la figure (a) (Fig. 2.6). chaque noeud est constitué de deux ensembles, un des hyperarêtes (λ) et un autre des variables (χ). Il est bien claire que la décomposition hypertree généralisée viole la condition dans (*Définition 2.21*), car la variable 13 disparait du noeud dont $\lambda = \{h_{10}, h_{14}\}$, et apparait dans le sous arbre enraciné en ce noeud.

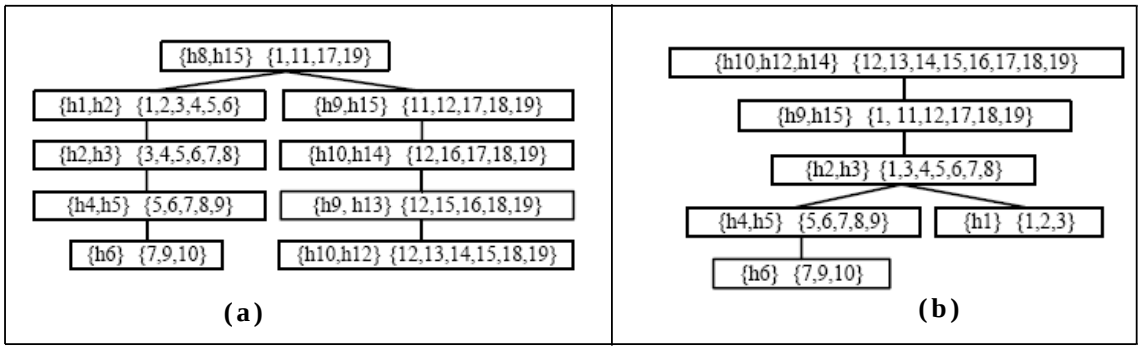

FIG. 2.8 – (a) hypertree décomposition généralisée (b) hypertree décomposition

2.5.8.1 Calcul de l'hypertree decomposition

Il existe de nombreux algorithmes de calcul de la *décomposition hypertree*, qui peuvent etre classées en deux catégories, les méthodes exactes et les méthodes heuristiques.

1. ***Méthodes exactes***
   Les algorithme de cette approche déterminent l'existence ou non d'une *décomposition hypertree* d'une largeur inferieure ou égale à un constant k donné, pour un hypergraphe. Le premier algorithme exacte proposé est celui de *k-decomp* [16], cependant, il est indeterministe. D'autres algorithmes sont proposés pour palier au non determinisme de *k-decomp* et pour améliorer le calcul de la *décomposition hypertree*, on peut citer, *opt-k-decomp*, *Red-k-decomp* et *Det-k-decomp*.

   L'*Algorithme 7* illustre l'algorithme *k-decomp* qui se base sur la définition des *[V]-component*.

2. ***Méthodes heuristiques***
   Les algorithmes exactes sont couteux en temps d'exécution et en espace mémoire, ce qui les rend inefficaces pour la décomposition des problèmes de grande taille. Pour remedier à cet inconvinient, des heuristiques ont été proposées pour le calcul de la *décomposition hypertree*.

   Il existe une variété d'heuristiques pour le calcul de cette décomposition. Les algorithmes de cette approche reposant sur ces heuristiques permettent de calculer une decomposition avec une meilleure complexité mais ne donne aucune garantie sur sa largeur de la décomposition.

   Les deux algorithmes heuristiques les plus connus et les plus éfficaces sont le BE (*Bucket Elimination*) et DBE (*Dual Bucket Elimination*).

Définition 2.22 (*couverture d'hyperarête*)

Soit $H = (V, E)$ un hypergraphe et T sa décomposition hypertree. On dit qu'une hyperarête $h \in H$ est complètement couverte par la décomposition Hypertree s'il existe un noeud $p \in T$ tel que $var(h) \subset \chi p$ et $h \in \lambda p$.

Définition 2.23 (*décomposition complète*)

Une décomposition Hypertree HD d'un hypergraphe H est dite complète si toute hyperarête h de H est complètement couverte par HD.

Définition 2.24 (*Compléter l'Hypertree decomposition*)

Pour compléter une décomposition hypertree, on procède ainsi, pour chaque hyperarête h qui n'est pas couverte par l'hypertree decomposition, on ajoute un nouveau noeud fils N a celui qui couvre h dans T, tel que $\lambda(N) = \{h\}$ et $\chi(N) = porte(h)$.

2.5.9 Comparaison

Gottlob et al. [17] ont donné une comparaison de différentes méthodes de décomposition structurelles des *CSP*, cette étude a révélé que la décomposition hypertree decomposition généralise, fortement, toutes les autres méthodes de décomposition observées (voir (a) Fig. 2.9). Cela signifie que chaque classe de *CSP* résolue en un temps polynomial par une autre méthode de décomposition, la décomposition hypertree est, aussi, capable de la résoudre polynomialement, mais il existe des classes qui peuvent être résolues en temps polynomial par la décomposition hypertree mais ne les sont pas par toute autre méthode explorée dans [17].

Cependant, *Jégou et al.* [**?**] ont invalidé une partie de ce résultat, en démontrant que le concept de la décomposition hypertree, situé au sommet de cette hiérarchie (voir (a) Fig. 2.9), n'est finalement pas meilleur que celui de la décomposition arborescente (Tree decomposition).

En accord avec le constat qu'il existe une classe de problèmes pour lesquels la largeur *hypertree width* est bornée alors que sa largeur *tree width* ne l'est pas, on a déduit que la décomposition hypertree généralise, fortement, Tree clustering. Néan- moins, ils ont montré que ce résultat est faux sur la base d'une nouvelle analyse de la complexité temporelle du Tree clustering.

Cette nouvelle expression de la complexité est exprimée à la taille de relations de compatibilité associées aux contraintes (la représentation en extension des contraintes).

De plus, en se basant sur la définition du *recouvrement minimum* d'un ensemble (*Définition 1* [**?**]), ils ont proposé une autre borne de complexité qui depend du paramètre du recouvrement.

Une décomposition hypertree d'un hypergraphe associe une Tree decomposition à un recouvrement par hyperarêtes de chaque cluster. Ainsi, une décomposition hypertree définit un recouvrement des ensembles $\chi(p)$, $\forall p \in sommets(T)$ dont la taille est au plus h, avec T est larbre résultat de la Tree decomposition.

Donc, la largeur de la décomposition hypertree est le parametre de recouvrement de la Tree decomposition induite est le meme, ce qui permet de prouver que Tree clustring est au moins aussi bon que la résolution par décomposition en hyperarbre.

Ainsi, les deux décompositions Tree et Hypertree sont équivalentes dans la hiérarchie des méthodes structurelles. La figure (b) Fig. 2.9 illustre la correction de la hiérarchie proposé dans [17].

Les crirères adoptés par les deux comparaison sont ceux introduit dans [17].

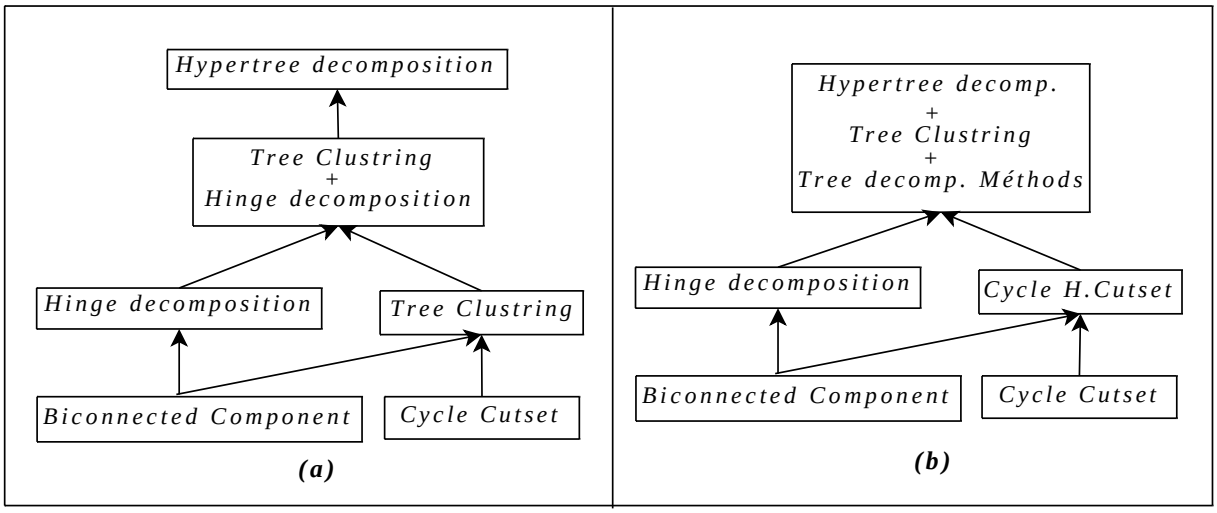

FIG. 2.9 – Comparaison entre les méthodes de décomposition

2.5.10 Conclusion

Dans ce chapitre, nous avons étudié les méthodes de décompositions structurelles qui constituent l'une de très importantes procédures de prétraitement exploitées avant ou pendant la résolution des problèmes *CSP*. Le gain espéré par la complexité théorique de ces méthodes est indéniable dans de nombreux cas, cela a ouvert un nouveau axe de recherche consistant à :

1. Trouver les meilleures décompositions.
2. Combiner ces méthodes pour obtenir des algorithmes plus éfficaces.
3. Développer des heuristiques pour le calcul des décompositions.
4. Optimiser les mises en oeuvre de ces méthodes et proposer les algorithmes qui les calculent.

Tous ceci en vu de réduire la complexité des méthodes de décomposition.

Généralement, la mise en oeuvre de la résolution exploitant ces méthodes se fait en trois étapes. Premièrement, on calcule la décomposition du problème à traiter, puis on fait la résolution en commencant par les sous problèmes (clusters de contraintes), et en fin, on fait la résolution du problème global.

Dans le chapitre suivant, nous allons étudier la résolution des problèmes *CSP* après une décomposition.

Chapitre 3

La résolution séquentielle par méthodes de décomposition structurelle

---

L'idée des algorithmes de résolution fondés sur l'exploitation de la structure du graphe de contraintes est d'exploiter certaines particularités topologiques de celui-ci, pour réduire sa complexité. En général, ils procèdent en isolant les parties a priori intraitables en temps polynomial, qui seront traitées séparément avant d'effectuer une seconde étape de résolution afin de calculer une solution au problème global.

Quand on parle de La résolution des *CSP* en utilisant des méthodes de décomposition structurelles on considère, généralement, la classe des approches qui analyse le *CSP* et définit un schéma de décomposition avant la recherche d'une solution et pas pendant.

Dans ce chapitre, nous proposons un algorithme séquentiel pour la résolution des problèmes *CSP* après une décomposition structurelle. Cet algorithme est basé sur la notion de hachage dont l'avantage est de garantir un accée direct aux données, ce qui nous permet une performance en terme de temps. Les résultats expérimentaux que nous avons obtenus nous ont permis de nous assurer de l'interêt pratique de cet algorithme.

Tout d'abord, nous présentons les différents algorithmes séquentiels de résolution des *CSP* acycliques binaires et n-aire proposés dans ce cadre, puis nous présentons de façons détaillée notre algorithme.

3.1 Introduction

Il est bien connu [14] [30] qu'un *CSP* dont le graphe de contraintes est acyclique peut être résolu plus efficacement (en un temps linéaire) qu'un autre qui ne l'est pas. Des algorithmes de résolution des *CSP* binaires ou naires ont été proposés.

3.2 La résolution séquentielle des problèmes CSP acycliques

3.2.1 La résolution d'un CSP binaire acyclique

Pour la résolution des *CSP* binaire acycliques, un algorithme polynomial est proposé par *Rina Dechter*. Cet algorithme est le *Tree solving* [11].

Algorithm 5 Algorithme Tree Solving

1: **Entrée** : Un arbre de contrainte T d'un *CSP* $P = (X, D, C, R)$.
2: **Sortie** : Un Backtrack free sur l'ordre d spécifié.
3: Générer un ordre $d = X_1,\ldots, X_n$ tel que un noeud père précède toujours ses noeuds fils.
4: Soit $X_p(i)$ le noeud père du noeud X_i.
5: **pour** i = n à 1 **faire**
6: Révise $(X_p(i), X_i)$
7: **si** le domaine de $X_p(i)$ est vide **alors**
8: Exit */*Pas de solution*/*
9: **finsi**
10: **fin pour**

Révise $(X_p(i), X_i)$
11: **pour** chaque valeur x du $dom(X_p(i))$ **faire**
12: **si** x n'a pas de support dans $dom(X_i)$ **alors**
13: Supprimer x du $dom(X_p(i))$
14: **finsi**
15: **fin pour**

Le Tree-solving accepte en entrée un réseau de contraintes acyclique (arbre) d'un *CSP* binaire. Chaque noeud de cet arbre, hormis la racine, possède un noeud père et peut avoir plusieurs noeuds fils.

La première étape de cet algorithme est la construction d'un arbre enraciné orienté, tout en spécifiant un ordre entre les noeuds de telle sorte que chaque noeud père doit apparaître avant ses noeuds fils dans l'ordre (3).

De l'étape 5 à 10, l'algorithme traite chaque arc et la contrainte associée, ceci des noeuds feuilles au noeud racine, en vue de réaliser une arc consistance orientée.

Pour chaque arc orienté de X_i vers X_j, la procédure *Révise* est appelée pour enlever les valeurs du domaine de X_j qui n'ont pas de support (valeur consistante) dans le domaine de X_i. Après avoir traité tous les noeuds jusqu'à la racine, et si aucun domaine des variables n'est vide, l'algorithme *Backtrack free* est utilisé pour trouver une solution.

La complexité de *Tree-solving* est linéaire en le nombre de variables. Elle est de l'ordre de $O(nd^2)$, où n est le nombre de variables et d la taille des domaines des variables (la procédure *Revise* est limité par d^2 et elle est exécutée n fois).

3.2.2 La résolution d'un CSP n-aire acyclique

Un *CSP* binaire dont la structure est acyclique peut être résolu en un temps polynomial en utilisant *Tree solving*. Cette propriété d'acyclicité peut être étendue pour les *CSP* n-aires en utilisant la notion de *Join Tree* (*Définition 2.7 et 2.8*).

Acyclic Solving [11] est une reformulation de *Tree solving* pour la résolution des problèmes CSP acycliques n-aire. Cet algorithme tient en compte aussi bien les variables que les contraintes (voir *Algorithme 9*).

Algorithm 6 Algorithme Acyclic Solving

1: **Entrée** : Un *CSP* acyclique $P = (X, D, C, R)$, $R = \{R_1, \ldots, R_t\}$. Un join-tree T de P.
2: **Sortie** : Vérifier la consistance et générer une solution.
3: $d = (R_1, \ldots, R_t)$ est un ordre tel que chaque relation apparaît avant ses fils dans l'arbre T enraciné en R_1.
4: **pour** j = t à 1 **faire**
5: **pour** chaque arête (j, k), k < j, dans l'arbre **faire**
6: $R_k \leftarrow SemiJoin(R_k, R_j)$
7: **si** une relation vide est créée **alors**
8: Exit /*pas de solution pour le problème*/.
9: **finsi**
10: **fin pour**
11: **fin pour**
12: Retourne les relations mises à jour et la solution est cherchée comme suit, Sélectionner un tuple dans R_1
13: **pour** chaque relation R_i, $i = 1$ à t **faire**
14: Prendre un tuple qui est consistant avec toutes les relations qui la précèdent $(R_1, \ldots, R_{i-1})$
15: **fin pour**

Etant donné un *CSP* acyclique et son Join Tree, *Acyclic solving* est capable de générer une solution quelque soit l'arité du problème. Dans une première étape, il traite l'arbre de jointure du bas vers le haut (*botom-up*) (4 à 11), et à chaque étape, il fait l'élimination des tuples de la relation père de la relation courante qui ne peuvent pas participer à une jointure avec les tuples de cette dernière, en appliquant une semi jointure (6).

Si le *CSP* est inconsistant, une relation vide est créée dans l'un des noeuds de l'arbre de jointure, et l'algorithme retourne *problème inconsistant* (7).

Après la fin de cette première phase (*botom-up*) (4 à 11), on aura une arc consistance dirigée (*DAC, Directed Arc Consistancy*) du noeud racine vers les noeuds feuilles.

Dans une deuxième étape (12 à 14), *Acyclic solving* procède à la recherche d'une solution en parcourant l'arbre de jointure de la racine vers les feuilles (*top-dowm*) (13) et à chaque noeud, il prend un tuple consistant avec ceux déjà pris.

La complexité de *Acyclic solving* est de l'ordre de $O(n.r.\log r)$, avec n le nombre de relations et r le nombre de tuples par relation.

N.B : Acyclic Solving prend les contraintes sous leurs représentation explicite où chacune d'elle est représentée par une relation contenant les tuples autorisés par cette contrainte. Cette représentation est celle adoptée dans la suite de ce mémoire.

3.3 Résolution du CSP après une Tree decomposition

Pour la résolution d'un *CSP* à partir d'une decomposition arborescente on applique l'étape (4) et (5) de l'algorithme *Join Tree Clustering (JTC)* proposé dans [11]. Ces deux étapes désignent l'algorithme *Join Tree Processing (JTP)*. Etant donné un CSP et sa decomposition arborescente, l'algorithme *Join Tree Processing* procède à la résolution de ce *CSP* (voir *Algorithme 10*) comme suit :

Algorithm 7 Join Tree Processing (JTP)

1: **Entrée** : un ensemble de sous problèmes $P = \{P_1, \ldots, P_n\}$ avec $P_i = \{R_{i1}, \ldots, R_{ij}\}$, et un arbre $T = (P, E)$.
2: **Sortie** : une solution du CSP s'il est consistant.
3: **pour** i = 1 à n **faire**
4: Résoudre P_i et soit R'_i l'ensemble de solutions.
5: Appliquer *Acyclic solving* sur le nouvel arbre $T = R'_1, \ldots, R'_n$.
6: **fin pour**

1. Chaque contrainte est placée dans le noeud de la décomposition arborescente qui contient les variables impliquées dans cette contrainte. Chaque noeud représente un sous problème dont la résolution consiste à trouver toutes les affectations des variables consistantes avec les contraintes correspondantes.
2. Chaque sous problème est résolu indépendamment (4). Par conséquent, le résultat est un arbre de jointure d'un problème acyclique équivalent au problème original.
3. On applique l'algorithme Acyclic Solving pour la résolution du problème global (5).

Exemple 3.1

Soit le *CSP* $P =< X, D, C, R >$, avec :
$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$
$D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}\}$
$C = \{C_1, C_2, C_3\}$
$C_1 = \{x_1, x_2, x_3\}$, $C_2 = \{x_1, x_5, x_6\}$, $C_3 = \{x_3, x_4, x_5\}$
$R = \{R_1, R_2, R_3\}$
$R_1 = \{(a, b, c), (a, c, b), (b, b, c)\}$, $R_2 = \{(a, b, c), (a, c, b)\}$, $R_3 = \{(c, b, c), (c, c, b)\}$

La figure Fig. 3.1 montre, *(a)* une tree decomposition de ce *CSP*, (b) sa résolution. Dans la figure (a), on peut remarquer que toutes les contraintes sont placées dans le noeud racine. Dans la figure (b), chaque noeud est représenté par une relation dont les tuples sont le résultat de la jointure des relations impliquées dans le noeud.

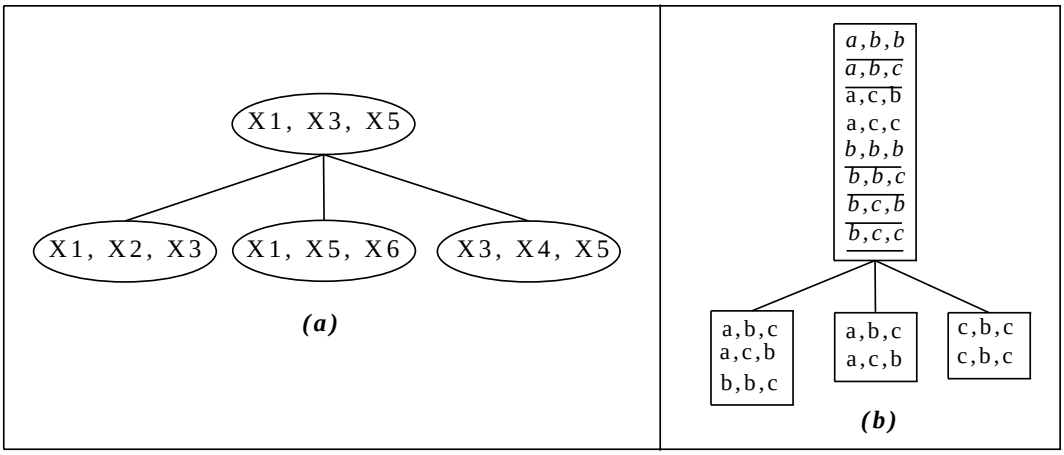

FIG. 3.1 – Résolution à partir d'une hypertree décomposition

Les tuples soulignés sont ceux qui ne peuvent pas participer à une jointure avec les tuples des relations des noeuds fils, et ils seront éliminés.

3.4 Résolution du CSP après une décomposition hypertree

Etant donné un *CSP* et sa decomposition hypertree, la résolution de ce CSP suit les étapes suivantes :

1. Compléter la décomposition hypertree si nécessaire.

2. Pour chaque noeud p, calculer une nouvelle relation R_p qui est la projection des jointures des relations des contraintes dans $\lambda(p)$ sur l'ensemble des variables de $\chi(p)$.

$$Rp = Join(\lambda(p))[\chi(p)].$$

   Le résultat est un arbre de jointure d'un problème acyclique équivalent au problème original.

3. On applique un algorithme adapté (ex. Acyclic Solving) sur l'arbre de jointure pour la résolution du problème global.

La première étape consiste à vérifier si l'hypertree est complet conformement à la définition (*Définition 2.23*) la *complétude* d'un problème CSP avant de procéder à sa résolution.

Exemple 3.2

On reconsidère l'exemple *3.1*. La figure Fig. 3.2 montre *(a)* une décomposition hypertree du problème *CSP*, (b) sa résolution. On fait la jointure entre les contraintes du premier ensemble λ et on projette sur les variables du deuxième ensemble χ. Dans la figure (b), les tuples soulignés sont ceux à éliminer, car ils n'ont pas de correspondants dans les relations des noeuds fils.

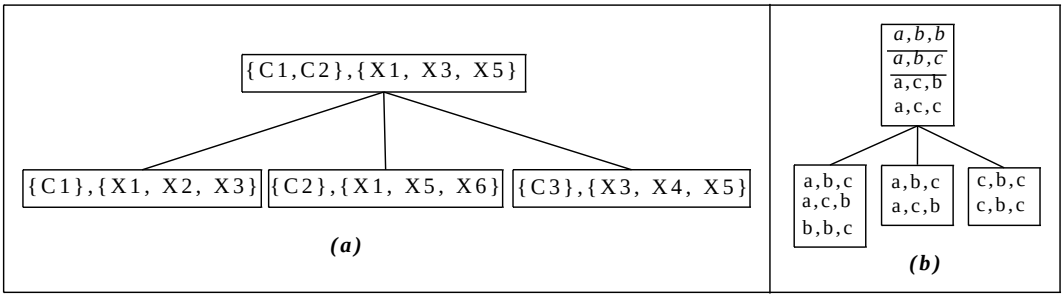

FIG. 3.2 – Résolution à partir d'une hypertree décomposition

3.5 L'algorithme S-HBR *(Sequential Hash Based Resolution)*

3.5.1 Introduction

Pour les problèmes combinatoires, les deux critères à optimiser pendant la résolution sont, le temps et l'espace. La difficulté des problèmes *CSP* est dûe au problème spatial généré par l'explosion mémoire pendant la résolution, et temporel où le temps pour la résolution d'une instance *CSP* est, généralement, combinatoire. Pour ce dernier, nous proposons un algorithme qui permet d'y remédier au moins partialement.

Pour la résolution des problèmes *CSP* après une décomposition hypertree, aucun algorithme n'est proposé dans la littérature, d'après nos connaissances, en dehors la méthode générale présentée dans (*3.4*), avec un calcul simple de jointures pour la résolution des sous problèmes, et quelques algorithmes (ex. *Acyclic Solving*) pour la résolution globale du problème. Dans cette section, nous proposons un algorithme qui permet de faire une résolution plus éfficace des sous problèmes et du problème global.

la relation entre les problèmes *CSP*, issus du domaine de l'*Intelligence Artificielle (IA)*, et ceux des *requêtes conjonctives* (*Conjunctive Queries (CS)*), issus des bases de données, est bien connue dans la littérature [28]. Dans cette contribution, nous allons utiliser la notion de *hachage* dérivée des bases de données afin d'apporter ses avantages a la résolution des problèmes *CSP*.

Avant de présenter notre approche, nous définissons la notion de *hachage*.

Définition 3.1 ***(Hachage)***
Une fonction de hachage est une fonction particulière facilement calculable qui, à partir d'une donnée fournie en entrée M, calcule une clé N servant à identifier rapidement, bien qu'incomplètement, la donnée initiale M. La donnée N est dite *clé* de hachage.

La procédure de hachage est l'application d'une fonction de hachage pour transformer certaines quantités de données en des nombres entiers relativement petits, qui peuvent servir comme des indexes des collections de données. Cette procédure est utilisée dans une base de données pour accélérer la recherche dans les tables ainsi que la comparaison des données.

Pour une contrainte C_i, soit R_i la relation associée, η_i l'ensemble des variables d'intersection de C_i avec toutes les contraintes qui la précèdent dans l'ordre, et HT_i la table de hachage de R_i. Pour un noeud n, Y_n est l'ensemble des variables d'intersection de

n avec son noeud père, et $X(n, a_i)$ est l'ensemble des variables d'intersection de n avec son noeud fils a_i.

Nous allons donner le principe de l'algorithme avant de le détaillller.

3.5.2 Algorithme S-HBR

Dans l'approche de résolution d'un *CSP* après une décomposition hypertree, les deux opérations coûteuses sont : la jointure de l'étape 2, utilisée pour la résolution des sous problèmes, et la semi jointure de l'étape 3. Puisque l'application des algorithmes plus efficaces pour le calcul de ces opérations permet d'avoir des résultats plus performants, l'algorithme *S-HBR* (pour *Sequential Hash Based Resolution algorithm*) est proposé.

Algorithm 8 Sequential hash Based Resolution (S-HBR)

1: **Entrée** :un hyper tree $< T, \lambda, \chi >$ d'un hyper- graph $H = (V(H), E(H))$.
2: **Sortie** : Solution si elle existe.
3: **pour** chaque noeud n de T **faire**
4: SP-HBR (n) // *Appliquer SP-HBR sur le noeud n.*
5: **fin pour**
6: soit T' le join tree representant le problème acyclic généré, avec pour tout $t' \in T'$, t' contient une et une seule contrainte.
A-HBR (T'), Appliquer l'algorithme A-HBR sur T'.
7: **End.**

Considérons un hypertree de n noeuds où chaques noeud constitue un sous problème. L'algorithme *S-HBR* (*cf. Alg. 11*) procède en deux étapes : la première est la résolution des sous problèmes qui est une optimisation du calcul des jointures. Elle est assurée en utilisant l'algorithme *SP-HBR* (pour *Sub Problem Hash Based Resolution*) (ligne 4). Le résultat est arbre de jointure. La deuxième étape est la résolution globale du problème qui est une optimisation de Acyclic Solving. Elle est effectuée en utilisant l'algorithme *A-HBR* (pour *Acyclic Hash Based Resolution*) (ligne 6). Le résultat de l'algorithme S-HBR est la solution du CSP initial.

Les contraintes sont représentées de façon explicite par un ensemble de tuples autorisés.

3.5.3 L'algorithme SP-HBR *(Sub Problem Hash Based Resolution)*

Dans l'hypetree, chaque noeud est un sous problème. Pour la résolution de chaque sous problème, nous appliquons l'opération de jointure, en utilisant le hachage, entre les

relations de chaque composante connexe. Ensuite nous appliquons le produit cartésien entre les résultats de ces composantes. Ainsi, si le problème est inconsistant, nous évitons de faire l'opération de produit cartésien dont le calcul est coûteux.

Le résultat d'exécution de SP-HBR à un noeud p est, seulement, une table R_p qui contient les tuples des valeurs permises par toutes les contraintes du sous problème. Les noeuds feuilles sont représentés sous forme d'une paire $< \chi(p), Hrel(p) >$ définie comme suit :

- $\chi(p)$: est l'ensemble des variables du noeud p,
- $Hrel(p)$: est la relation de la contrainte générée par l'operation de jointure, dont les tuples sont hachés sur les variables d'intersection avec le noeud père dans l'hyper tree.

Dans l'algorithme *SP-HBR* (*cf. Algo. 12*), premièrement nous décomposons chaque sous problème en des composantes connexes (ligne 3). Ensuite, nous appliquons l'opération de jointure pour chacune des composantes indépendamment des autres composantes, selon le principe de la jointure par hachage (ligne 4 à 28) : pour chaque composante C = $\{R_1, R_2, R_3, R_4\}$ nous appliquons une liste d'opérations de jointure $(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$. Pour accomplir ceci, nous commencons par effectuer un ordre entre les relations de la même composante (ligne 5), puis, nous hachons toutes les relations, hormis la première, (ligne 6 à 9). L'opération de hachage est appliquée sur les variables d'intersection avec toutes les relations précédentes dans l'ordre (ligne 7). Les variables de jointure d'une relation sont celles de l'intersection avec toutes les relation qui la précède. L'opération de jointure est exécutéee en appelant la procédure *Join* (ligne 10).

La procedure de jointure *Join* (ligne 29 à 42) procède d'une manière récursive. A chaque itération, nous appliquons l'opération de jointure entre deux relations : le résultat de la dernière opération de jointure R_{tmp} (initialement R_1, cf. ligne 29) et la relation R_j suivante dans l'ordre, de la manière suivante : nous hachons chaque tuple t de la relation R_{tmp} sur l'ensemble des variables η_j (ligne 35). Si la valeur de hachage correspond à une partition p de la table de hachage de la relation R_j, nous faisons la jointure de t avec tous les tuples de p. Ainsi, nous construisons la nouvelle relation R_{tmp} (ligne 37), et nous passons à l'opération suivante.

Si le résultat de la jointure $Res_i = \phi$ ceci implique que le problème n'a pas de solution (ligne 11). Sinon, nous calculons la dernière relation, dénoté Res en utilisant le produit cartésien des résultats des composantes (ligne 14 à 27). Res est calculée progressivement avec le calcul des opérations de jointure. Pour le calcul des jointures de chaque composante Res_i, nous faisons le produit cartésien de celle ci avec Res. Dans ce qui suit nous illustrons cet algorithme par un exemple.

Algorithm 9 Sub Problem Hash Based Resolution (SP-HBR)

Entrée : un noeud n, de l'hyper tree, representé par le sous graph de constraints $G_n = \langle \chi(t), \lambda(t) \rangle$ avec : $\chi(t) \subset V$, $\lambda(t) \subset E$ et $t \in T$. $\langle T, \chi, \lambda \rangle$ est l'hypertree de l'hypergraph $H = (V, E)$.
Sortie : Generer des solutions pour le sous problem n representées par une relation Res (initialement $Res = \phi$).
Extraire l'ensemble des composantes connexes, C, du graphe G_n (qui represente le noeud n).
pour chaque composante $C_i = \{R_1, \ldots, R_k\} \in C$ **faire**
  Construire un ordre $d = R_1, \ldots, R_k$ entre les contraintes de C_i.
  pour chaque relation $R_i, \forall i \in \{2, \ldots, k\}$ **faire**
    $\eta_i = \bigcup_{j \in m}(R_i \cap R_j)$ou $m = \{$ identities des relations qui precèdent R_i dans l'ordre $\}$
    Calculer Hash(R_i, η_i), $\forall t \in \mathcal{R}_i$.
  fin pour
  $Res_i = Join(C_i, n)$, ou Res_i est le résultat de l'operation de jointure.
  si $\mathcal{R}es_i = \phi$ **alors**
    Exit. /* le problem n'a pas de solution */
  sinon
    si $i = 1$, est la première composante **alors**
      $\mathcal{R}es \leftarrow \mathcal{R}es_1$
    sinon
      $\mathcal{R}es_t \leftarrow \mathcal{R}es$, $\mathcal{R}es \leftarrow \phi$.
      pour tout tuple t de Res_t **faire**
        $t \leftarrow t \times t'$, $\forall t' \in \mathcal{R}es_i$,
        si $i = |C|$, et n est une feuille **alors**
          Hash (t, Y_n), hacher sur les variables d'intersection avec le noeud père,
        finsi
        $\mathcal{R}es \leftarrow \mathcal{R}es \cup t$,
      fin pour
      $\mathcal{R}es \leftarrow \mathcal{R}es_t$
    finsi
  finsi
fin pour

La procédure Join $(\mathcal{C}_i, n)$
$\mathcal{R}_{tmp} \leftarrow \mathcal{R}_1$, $\mathcal{R}_{tmp'} \leftarrow \mathcal{R}_{tmp}$, $\mathcal{R}_{tmp} \leftarrow \phi$.
pour j = 2,...,k **faire**
  pour chaque tuple $t \in \mathcal{R}_{tmp'}$ **faire**
    si j = k, la dernière opération de jointure **alors**
      $\mathcal{R}es_i = \mathcal{R}es_i \cup t$.
    sinon
      $h = Hash(t, \eta_j)$
      si la partition p correspendante à h dans $\mathcal{HT}_j \neq \phi$ **alors**
        $\forall t' \in p, \mathcal{R}_{tmp} = \mathcal{R}_{tmp} \cup (t \bowtie t')$.
      finsi
    finsi
  fin pour
  $\mathcal{R}_{tmp'} \leftarrow \mathcal{R}_{tmp}$, $\mathcal{R}_{tmp} \leftarrow \phi$
fin pour

Exemple 3.3

Soit un noeud d'un hypertree constitué du sous ensemble de contraintes suivant :

$C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$, avec :

$D_1 = D_2 = D_3 = D_4 = D_5 = D_6 = \{0, \ldots, 5\}$

$S(C_1) = \{a, b, c\}$, $S(C_2) = \{i, j\}$, $S(C_3) = \{a, d\}$, $S(C_4) = \{d, b, f\}$,
$S(C_5) = \{f, g\}$, $S(C_6) = \{i, k\}$.

et $R_1 = \{(0, 0, 2), (3, 2, 2), (1, 1, 3)\}$, $R_2 = \{(2, 1), (0, 4), (1, 1)\}$,
$R_3 = \{(0, 1), (0, 5), (1, 1)\}$, $R_4 = \{(3, 1, 1), (1, 2, 1), (4, 0, 4)\}$,
$R_5 = \{(2, 0), (2, 1), (3, 4)\}$, $R_6 = \{(4, 0), (0, 1), (3, 2)\}$,

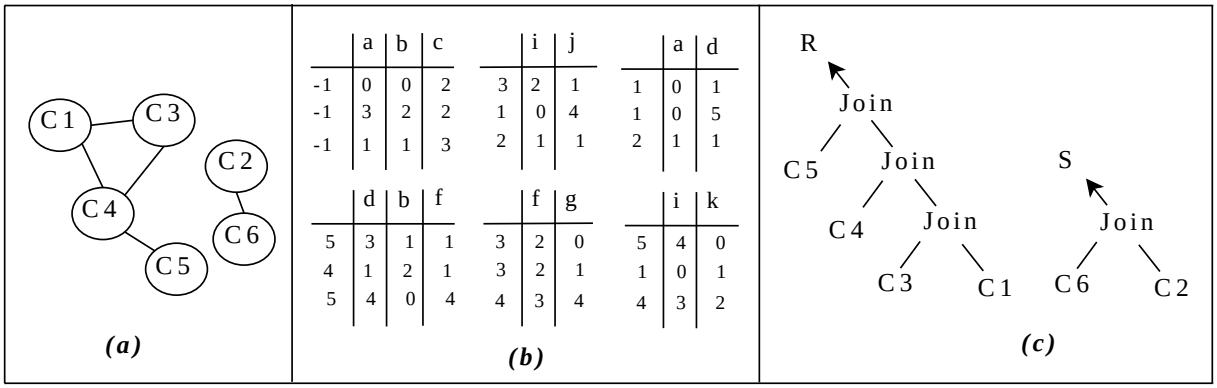

FIG. 3.3 – Exemple d'exécution de l'algorithme *SP-HBR*

Il est évident que nous avons deux composante connexes $\{C_1, C_3, C_4, C_5\}$ et $\{C_2, C_6\}$, voir (a) (Fig 3.3), et selon l'ordre des contraintes dans chacune des composantes nous avons :

$\eta_3 = S(C_3) \cap \{S(C_1)\} = \{a\}$,
$\eta_4 = S(C_4) \cap \{S(C_1) \cup S(C_3)\} = \{b, d\}$
$\eta_5 = S(C_5) \cap \{S(C_1) \cup S(C_3) \cup S(C_4)\} = \{f\}$
$\eta_6 = S(C_6) \cap \{S(C_2)\} = \{i\}$

Donc, les tuples de R_3 seront hachés sur l'ensembles des variables d'intersection $\eta_3 = \{a\}$, ceux de R_4 sur l'ensemble de variables $\eta_4 = \{b, d\}$, et ceux de R_5 sur $\eta_5 = \{f\}$. Pour la deuxième composante les tuples de R_6 seront hachés sur l'ensembles des variables d'intersection $\eta_6 = \{i\}$.

La figure (b) Fig 3.3 montre le hachage de ces relations sur les variables d'intersection en appliquant une simple fonction $((\sum x_i mod 5) + 1)$. La clé de hachage est stockée dans la première colonne de chacune des tables des relations.

Après l'exécution de tous les produits cartésiens d'un noeud n (ligne 20), nous hachons le tuple obtenu t sur les variables d'intersection avec le noeud père de n, si n est une feuille. Donc, t sera prêt pour l'opération de semi jointure dans l'algorithme *A-HBR*. Nous n'ajoutons t à sa partition correspendante p que si elle est vide ($p = \phi$). Ainsi, nous aurons au maximum un tuple par partition en éliminant les doublons. Cette élimination n'a aucune influence sur la résolution car :

1. Pour le calcul de la semi jointure (algorithme *A-HBR*), on fait le filtrage des noeuds père sur leurs noeuds fils (il suffit d'avoir un seul tuple par partition dans la relation du noeud fils).
2. On s'interesse à trouver une seule solution pour le problème traité.

Remarques

1. Le mot *Hash* est utilisé pour désigner le hachage d'une relation *Hash (R,x)* ou le hachage d'un tuple *Hash (t,x)*.
2. Une bonne fonction de hachage est cruciale pour les performances de l'algorithme. Dans notre cas, son efficacité est mesurée par le nombre de collisions générées. Une collision est engendrée lorsque deux tuples ont la même valeur de hachage, ces tuples ne peuvent être stockées à la même position, et donc une stratégie de résolution des collisions est utilisée.

   Ainsi, une mauvaise fonction de hachage, i.e. produisant beaucoup de collisions et par conséquent donnant plusieurs accès à la même position dans la table de hachage, va fortement dégrader la rapidité de la recherche dans l'algorithme.

3.5.4 L'algorithme A-HBR *(Acyclic Hash Based Resolution)*

L'algorithme *A-HBR (Acyclic Hash Based Resolution)* vérifie la consistance d'un réseau de contraintes acyclique et génère une solution si elle existe. L'entrée de cet algorithme est un arbre de jointure, chaque noeud de cet arbre contient le résultat de l'algorithme *SP-HBR*. Cet algorithme (*cf. Algo. 13*) consiste en deux étapes. La première est l'opération de contraction *CONTRACT* qui fait la contraction de l'arbre enraciné T (ligne 3). Une opération de semi jointure est associée à chaque opération de contraction. Le résultat de cet opération est un arbre T' arc consistant orienté du noeud racine vers les noeuds feuilles. La deuxième est l'opération de la recherche d'une solution *S-SEARCH* (ligne 4) qui commence du noeud racine de l'arbre T' et propage la solution vers les noeuds feuilles.

Algorithm 10 Acyclic Hash Based Resolution algorithm (A-HBR)

1: **Entrée** :une Join Tree $T =< A, E >$, A est l'ensemble de nodes, et E est l'ensemble d'arcs. *//Le résultat de l'algorithme SP-HBR.*
2: **Sortie** : Determineer une consistance arc dirigée er générer une solution.
3: T' = CONTRACT (T), T' est le Join Tree générée.
4: S - SEARCH (T'),
5: **End.**

3.5.4.1 L'opération CONTRACT

L'algorithme de la contraction (*cf. Alg. 14*) force l'arc consistance orientéée dans l'arbre de jointure d'entrée en appliquant l'opération de contraction, pour chaque noeud père, avec chacun de ses noeuds fils feuilles. Ainsi, la vérification de toutes les contraintes est assurée car dans la décomposition hyper tree, il ne peut pas arriver qu'une variable qui disparaît dans un noeud paraîtra dans un noeud successeur.

Après chaque opération de contraction, les noeuds feuilles traités sont marqués, on les considère comme s'ils sont élagués, et on aura un nouvel arbre, possèdant d'autres noeuds feuilles, avec lequel on fait la même chose, et ainsi de suite, jusqu'a ce que l'arbre de jointure ne contient aucun noeud père qui a des noeuds fils feuilles ou jusqu'à ce que tous les noeuds, hormis la racine, seront marqués.

Pour chaque opération de contraction (ligne 3 à 9), nous hachons chaque tuple t de la relation d'un noeud père p (ligne 4 à 14) sur les variables d'intersection avec ses noeuds fils feuilles a_i (ligne 6). h_i est la valeur de hachage obtenue. Les relations des noeuds fils a_i sont hachées sur les variables d'intersection avec leurs noeuds père pendant l'exécution de l'algorithme *SP-HBR*, ce qui permet un test de jointure plus rapide par un accès direct. Si les partitions des tables de hachage des noeuds a_i correspendantes à h_i ne sont pas vides (ligne 7), avant de stocker t nous le hachons sur les variables d'intersection du noeud p avec son noeud père (ligne 10). Sinon, t est éliminé (ligne 8).

Le résultat final de l'opération de contraction d'un noeud père p est un noeud $p' =< \chi(p'), Hrel(p') >$ dont la relation est constituée des tuples qui sont consistants avec ceux des relations des noeuds fils, et hachés sur les variables d'intersection du p avec son noeud père. Si $Hrel(p')$ est vide, ceci implique que le problème n'a pas de solution (ligne 17).

3.5.4.2 L'opération S-SEARCH

Pour la recherche d'une solution au problème CSP, un algorithme de *Backtrack free* est appliqué sur l'arbre de jointure arc consistant dirigé résultant de l'opération de contraction.

Algorithm 11 CONTRACT algorithm

Entrée :une Join Tree $T =< A, E >$, A est l'ensemble de noeuds, et E est l'ensemble d'arcs.
Sortie : Générer une join tree $T' =< A', E' >$ arc consistant dirigée, avec $\forall a' \in A', a' =< \chi(p), Hrel(p) >$.
pour chaque noeud n de T, qui possède des fils feuilles **faire**
  pour chaque tuple t de R_n **faire**
    pour chaque fils feuille a_i de n **faire**
      Calculer Hash $(t, X(n, a_i))$.
      si la partition correspondante dans la table de hachage de a_i est vide **alors**
        Eliminer t, $(R_n \leftarrow R_n - t)$.
        si c'est le dernier fils **alors**
          Calculer Hash (t, Y_n), hacher sur les variables d'intersection avec le noeud père.
          $HrelT'(n) \leftarrow HrelT'(n) \cup t$.
        finsi
      finsi
    fin pour
  fin pour
  Marquer tous les noeuds fils a_i de n.
  si $R_n = \emptyset$ **alors**
    Exit /* le problème n'a pas de solution */
  finsi
fin pour
End.

Algorithm 12 S - SEARCH algorithm

Entrée : Un arbre de jointure $T' = \langle A', E' \rangle$, avec $\forall a' \in A', a' = \langle \chi(a'), Hrel(a') \rangle$. *//Le résultat de l'opération CONTRACT.*
Sortie : Génère une solution au problème *CSP*.
$sol \leftarrow \emptyset$, la solution du probléme. *//ensemble des valeurs des variables.*
Marquer le noeud racine,
prendre un tuple $t \in Hrel(root)$, $sol \leftarrow t$.
pour chaque noeud a de T' **faire**
  si le noeud père de a est marqué **alors**
    Hash(sol, Y_a), prendre un tuple $t \in Hrel(a)$, de la partition correspondante de la table de hachage du noeud a.
    $sol \leftarrow sol \cup t$,
    Marquer le noeud a,
  finsi
fin pour
retourner sol.

Dans l'algorithme *S-SEARCH* (*cf. Algo. 15*), initiallement, nous marquons le noeud racine r (ligne 3) et prenons un tuple t de sa relation. Pour chacun de ses noeuds fils a de r (ligne 5 à 11), nous hachons t sur les variables d'intersection de r avec a (ligne 7) et prenons un tuple de la partition de hachage correspondante dans la relation du noeud a. nous faisons la même chose pour tout les noeuds de l'arbre pour lesquels le noeud père est marqué. A chaque itération, les noeuds fils traités sont marqués (ligne 9)). La solution est mise à jour à chaque recherche d'un tuple correspodant dans un noeud (ligne 8).

3.6 Complexité de l'algorithme *S-HBR*

Pour une implémentation simple de la résolution après une hypertree décomposition (*SIRH*), la complexité de la résolution des sous problèmes est de l'ordre de $O(\sum_{i=1}^{h} r^i)$, avec r est la taille maximale des relations du *CSP*, et h est la largeur de la décomposition hypertree (*Hypertree width*). La compléxité de la résolution du *CSP* résultant (arbre de jointure) est $O(nl^2)$, avec n le nombre de contrainte dans le CSP résultatnt (le nombre d'arcs), et l le nombre maximum de tuples dans un noeud de l'arbre de jointure.

La complexité de notre algorithme *S-HBR*, et de l'ordre de $O(\sum_{i=1}^{h-1} r * \mathcal{P}^i)$ pour la résolution des sous problème (algorithme *SP-HBR*), avec h est l'hypertree width,et $\mathcal{P}$ est le nombre maximum de tuples dans une partition de hachage sur les variables η (η est l'ensemble des variables d'intersection d'une relation avec les relations qui la précèdent dans l'ordre). Pour la résolution entière du problème (algorithme *A-HBR*), la complexité est de $O(n\mathcal{P}^2)$, avec n le nombre de contraintes dans le CSP résultant.

Si on suppose que $r = d^s$, avec d le nombre maximum de valeurs par domaine, s le nombre de variables par contrainte, donc $V(r, \eta) = d^{s-|\eta|}$.

La fonction que nous avon utilisé nous permet d'avoir un seul tuple par partition (pour un nombre limité de tuples), ceci pour l'algorithme *A-HBR* (car on fait l'élimination des doublants). Donc, nous pouvons conclure que la complexité de *A-HBR*, en utilisant cette fonction, est de $O(n)$.

3.7 Correction de notre algorithme *S-HBR*

Proposition : L'algorithme S-HBR vérifie la propriété de complétude.

Preuve : Pour prouver la proposition, il sffit de montrer que la résolution avec hachage (en utilisant S-HBR) se comporte de la même manière qu'avec celle sans hachage. Autrement dit, une résolution avec hachage recouvre toutes les solutions qu'une autre sans hachage.

La fonction de hachage que nous avons utilisé est la suivante :

pour le hachage d'un tuple t sur n variable d'intersection, $f(t) = \sum_{i=1}^{n}((x_i + 1) * 10^{\ln(f(t))+1})$.

1. Il est évident qu'une fonction de hachage ne fait que la réorganisation des tuples de la relation dans un ordre mieux adapté pour le recherche des tuples.
2. Soit le tuple t' (resp. la relation R') est le résultat de hachage d'un tuple t (resp. d'une relation R), et soit $h(t')$ désigne la partition de hachage dont t' fait partie. Il est claire que $\forall t \in R$, $t' \in R'$, car $\forall t, f(t) \neq null$.
3. Pour deux relations R_1 et R_2, $\forall t_1 \in R_1, \forall t_2 \in R_2, \forall x \in \eta_i$, $t'_1(x) = t'_2(x) \Longrightarrow h(t'_1) = h(t'_2)$ Il est bien évident que cette propriété est assurée par notre fonction.

La première et la deuxième condition garantissent qu'aucun tuple n'est sauté en utilisant la technique de hachage. La troisième condidtion assure que le teste de consistance en utilisant le hachage ne peut jamais échouer. Ainsi, nous pouvons garantir que notre fonction de hachage ne provoque aucune perte de tuple ce qui permet de dire que notre algorithme verifie la propriété de complétude.

Sur le plan pratique, ce qui permet de confirmer cette proposition est l'otention des mêmes résultats pour une meme instance CSP en utilisant l'algorithme S-HBR (avec hachage) ou l'implémentation simple sans hachage.

3.8 Expérimentations

Pour montrer l'interêt pratique de notre approche, nous avons implémenté l'algorithme *S-HBR* (notre solveur), et, pour évaluer pratiquement nos résultats par rapport aux algorithmes existent, nous avons fait une implémentation simple de la résolution après une décomposition hypertree (*SIRH*).

Dans cette section, nous allons donner les résultats expérimentaux des algorithmes implémentés. Nous avons testé les deux solveurs sur une série de benchmarks *CSP* en extension, qui existent dans la littérature, et qui regroupe aussi bien des problèmes académiques que des problèmes issus du monde réel.

Cette collection de benchmarks est représentée sous un nouveau format qui est le format *XCSP 2.1* (le format étendu pour la représentation des réseaux de contraintes

en utilisant le *XML*) proposé par *Le comité organisationnel de la 3 eme compétition internationale des solveurs CSP* [35]. Par conséquent, nous avons extrait les instances proposées par cette dernière pour servir de challenge dans le cadre d'une compétition de solveurs de *CSP* [1].

Les entrées des solveurs sont des instances *CSP* sous format *XCSP* et les hypertrees correspondants générés par un algorithme de décomposition hypertree, sous format *GML*. Pour cela, Nous avons exploité, aussi, un Parser de fichier *GML* (proposé par *M. Raitner, M. Himsol*)[2] et un autre pour le ficher *XCSP* (proposé par *O. Roussel*)[3] pour les besoins d'analyser les données d'entrée (l'instance *CSP* et sa décomposition hypertree).

Notre approche de résolution est constituée d'un ensemble de classes permettant la manipulation de la structure arbre du *CSP*. Un hypertree est un ensemble de noeuds qui est constitué d'un ensemble de variables (χ) et d'un ensemble de contraintes (λ). Les contraintes sont représentées en extension, ce qui implique. Chacune des classes possèdent des méthodes qui s'occupent des traitements appropriés (*Hachage* pour calculer les valeurs de hachage,etc.),

Le tableau (Tab. 3.1) résume les résultats des tests obtenus.

CSP						SIRH (sec.)	S-HBR (sec.)
Name	|V|	|E|	Nd	HTW	|r|		
3-insertions-3-3	56	110	86	7	3	>1000	127
domino-100-100	100	100	50	2	100	27	7
domino-100-200	100	100	50	2	200	103	28
domino-100-300	100	100	50	2	300	241	61
geom-30a-4	30	81	70	4	4	5	2
hanoi-6	62	61	59	1	2148	8	2
hanoi-7	126	125	125	1	6558	71	9
haystacks-06	36	95	88	3	8	9	4
haystacks-07	49	153	144	4	9	514	66
langford-2-4	8	32	31	4	8	11	3
pigeons-7	7	21	19	3	6	12	3
series-6	11	30	27	3	30	5	2
series-7	13	42	39	4	42	>1000	106
Renault	101	134	81	2	48721	780	20

TAB. 3.1 – Résultats expérimentaux de l'algorithme *S-HBR*

[1] http ://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html
[2] http ://www.cs.rpi.edu/ puninj/XGMML/GML-XGMML/gml-parser.html
[3] http ://www.cril.univ-artois.fr/ roussel/CSP-XML-parser/

Remarque
$|V|$, $|E|$ et $|r|$ désignent, respectivement, le nombre de variables, le nombre de contraintes du *CSP*et le nombre maximum de tuples par relation. Nd et HTW sont, repectivement, le nombre de noeuds et la largeur de l'hypertree. $S-HBR(sec)$ représente le temps d'exécution de l'algorithme *S-HBR* en seconde, et $SIRH(sec)$ celui de l'implémentation simple de la résolution après une décomposition hypertree.

Les instances pour lesquelles le nombre de tuples par relation ainsi que le nombre de relation (contraintes) est très grand, génèrent un problème d'espace mémoire pour notre machine. Toutes les expérimentations sont faites sur un *Intel Pentium IV équipé d'un CPU 2.4 GHz* avec *600 Mo* de *RAM* sous *Linux* version *6.0.52*.

Les résultats du tableau (Tab. 3.1) nous permet de conclure que :

1. Les instances dont l'hypertree width est grand prend un temps de résolution énorme même si la taille des relations est petite (*3-insertion-3-3*) car la complexité est exponentielle en la largeur, d'ou l'objectif de minimiser la largeur de la décomposition pour toutes les méthodes.
2. Pour les instances dont le nombre de tuples traité par relation est relativement petit, une implémentation simple de la résolution après une décomposition hypertree peut donner des résultats prochent de ceux de *S-HBR* (pour *domino-100-100* à cause d'une hypertreewidth de 2, et *hanoi-6* à cause du petit nombre de noeud qui est 27), à cause des opérations de calcul des valeurs de hachage, qui sont négligeables d'autant plus que l'instance est grande.
3. L'algorithme *S-HBR* tire le plus de profit par rapport à une implementation simple pour les instances dont le nombre de tuples et de plus en plus grand, car il parcourt une partie de la relation plutot que la totalité comme dans SIRH (le gain en complexité est en exponentiel). Ce profit est de plus en plus remarquable pour les instance dont La taille de la relation est grande en raison d'une hypertree width grande (*3-insertion-3-3*), d'un nombre maximum de tuples par relation est grand (*Renault* et *hanoi-7*) et pour des instances à un grand nombre de noeud (*hanoi-7* et *haystacks-07*).

3.8.1 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme *S-HBR*, un nouvel algorithme pour la résolution des *CSP* acycliques après une décomposition hypertree. Il repose sur une technique de hachage qui nous permet de gagner en terme de complexité. Nous avons prouvé son efficacité pratique en l'appliquant à des instances de *CSP* et avons montré, par des résultats expérimentaux, son interet pratique.

D'autres optimisations peuvent être apportées à cet algorithme pour le rendre plus performant en utilisant les différentes techniques connues. Ceci fait l'objet du *Chapitre 5*, dans lequel nous allons intégrer le parallélisme.

Chapitre 4

Le parallélisme

Le besoin de solutions rapides et de résolution des problèmes de taille importante fait apparaître l'idée d'utiliser plusieurs processeurs en concurrence, ce qui est connu sous le nom du calcul parallèle (*parallel processing*).

Dans ce chapitre, nous allons présenter les différentes architectures parallèles et leurs classification, ainsi que les différents modes et types du parallélisme utilisés. Nous nous intéressons à la parallélisation de l'opération de jointure et nous détaillons un technique de base de parallélisation qui est la *contraction d'arbre parallèle* [32]. Enfin, nous présenrons les métriques pour mésurer la performance d'un algorithme paralléle.

4.1 Introduction

L'utilisation des machine parallèles pour la résolution des problèmes constitue une solution aux besoins de performances des applications concernées. En effet, l'approche du parallélisme implique la part software et hardware dans le domaine de l'informatique. Le premier se présente dans les différentes architectures parallèles qui existent, et le deuxième dans les modes du parallélisme possibles pour un problème donné.

L'opération utilisée pour faire le filtrage des relations dans la résolution des *CSP* est celui de jointure ou de semi jointure. De ce fait, nous allons étudier les differents types du parallélisme de cet opération, qui peuvent être *inter-operateur* ou *intra-opérateur*. Pour le parallélisme *inter-opérateur*, la structure de l'opérateur (grpahe, arbre, liste ...etc.) à exécuter spécifie la technique du parallélisme la plus adaptée.

Différentes techniques de base du parallélisme ont été introduites, chacune d'elles est adoptée pour un cas bien spécifique. La *contraction d'arbre parallèle* est l'une de ces techniques, qui est utilisée dans le cas où on a une structure arborescente.

Nous avons vu dans le *Chapitre 2* que les méthodes de décomposition cherchent à transformer un *CSP* quelconque en un CSP acyclique (dont la structure est un arbre). Donc, une résolution parallèle d'un *CSP* après une décomposition, se fait en utilisant une technique adaptée à la structure arbre. Nous faison référence ici à une technique bien connue qui est celle de la contraction d'arbre parallèle.

4.2 Modèles de programmation parallèle

On distingue, classiquement et selon la taxonomie de Flynn-Tanenbaum, quatre types principaux de parallélisme : *SISD*, *SIMD*, *MISD* et *MIMD*. Cette classification est basée sur les notions de flot de contrôle représenté par les deux premières lettres (*SI* ou *MI*, *I* voulant dire *Instruction*), et flot de données représenté par les deux dernières lettres (*SD* ou *MD*, *D* voulant dire *Data*).

Autrement dit, selon le modèle de programmation, Il existe deux types de parallélisme : le parallélisme de données (*Data Parallelism*) et le parallélisme de programme appelé, généralement, le parallélisme de tâche ou de contrôle (*control Parallelism*).

4.2.1 Parallélisme de données

Dans ce modèle de parallélisme, les données sont partitionnées en des sous ensembles ou fragments, le nombre de fragments appelé *degré de partitionnement*. Les différents fragments sont alloués aux différents processeurs. Le partitionnement peut être statique, selon un modèle fixe adapté, ou dynamique (le mieux adapté) dans lequel les données sont partitionnées dynamiquement.

L'un des problèmes importants de partitionnement de données est le placement des données qui a un impact sur la distribution de la charge (*Load Balancing*).

4.2.2 Parallélisme de programme

Appelé, généralement, parallélisme *pipeliné*, il se base sur la décomposition du traitement en des sous traitements, qui seront exécutés en parallèle. Il est difficile a implémenter dans des architectures à mémoire distribuée (le nombre de tâches et le temps d'exécution varient dynamiquement).

Comme le placement de données dans le *data parallelism*, l'allocation des sous traitements aux processeurs a un impact sur les performances pour le *parallelisme de contrôle*.

4.3 Les types du parallélisme d'un opérateur de jointure

Il existe deux types du parallélisme d'un opérateur, le parallélisme *intra-opérateur* et le parallélisme *inter-opérateurs*.

4.3.1 Le parallélisme intra-opérateurs

Le terme *'intra'* fait référence à un traitement d'une même tache par plusieurs processeurs. Le parallélisme *intra-opérateur* est l'une des techniques les plus utilisées pour augmenter les performances d'un solveur. Il permet à plusieurs processeurs d'être utilisés pour le traitement d'une même opération. Il consiste à répartir les données d'un même opérateur sur plusieurs processeurs/mémoire, et à exécuter une copie de l'opérateur sur chaque processeur en agissant sur des sous ensembles disjoints de données.

Ce type du parallélisme est, généralement, très efficace pour réduire le temps d'exécution d'un opérateur. Cependant, il est très sensible au déséquilibre des charges des différents processeurs, puisque le temps d'exécution global de l'opérateur est le temps d'exécution du processeur le plus lent, celui-ci nécessite un équilibrage de charges qui est plus simple à mettre en oeuvre dans un environnement mémoire partagée ($PRAM$) que dans un environnement distribué (mémoire répartie).

4.3.2 Le parallélisme inter-opérateurs

Le terme *'inter'* fait référence à un traitement concurrent de plusieurs tâches indépendantes. Le parallélisme *inter-opérateur* consiste à exécuter en parallèle plusieurs opérateurs, chaque opérateur est exécuté sur un ou plusieurs processeurs. Ce type du parallélisme est classifié en deux catégories :

1. *Le parallélisme inter opérateurs indépendant*
   Dans ce type du parallélisme, deux ou plusieurs opérateurs sont exécutés sur des groupes disjoints de processeurs de manière à ce qu'un processeur ne puisse exécuter à la fois deux opérateurs différents.
2. *Le parallélisme inter opérateurs pipeliné*
   Dans ce type de parallélisme, un même processeur peut exécuter un ou plusieurs opérateurs reliés par un lien producteur/consommateur. Ainsi les résultats d'un opérateur peuvent être immédiatement utilisés dans le traitement de l'opérateur suivant dés la génération des premiers résultats. En effet, les points forts de ce parallélisme sont :

(a) L'exécution de l'opérateur suivant peut être déclenchée dès la génération des premiers résultats de l'opérateur précédent.

(b) Les résultats intermédiaires ne seront pas stockés car il seront utilisées immédiatement après leurs génération, ce qui permet un gain en complexité spaciale

4.4 le parallélisme dans l'exécution des jointures et semi jointures

Pour un problème dont la solution nécessite l'application d'une multitude d'opérations de jointures/semi jointures, le graphe d'opérateurs est un arbre qui peut être un arbre linéaire gauche (a), un arbre linéaire droit (c) ou de type bushy (b) (voir Fig. 4.1).

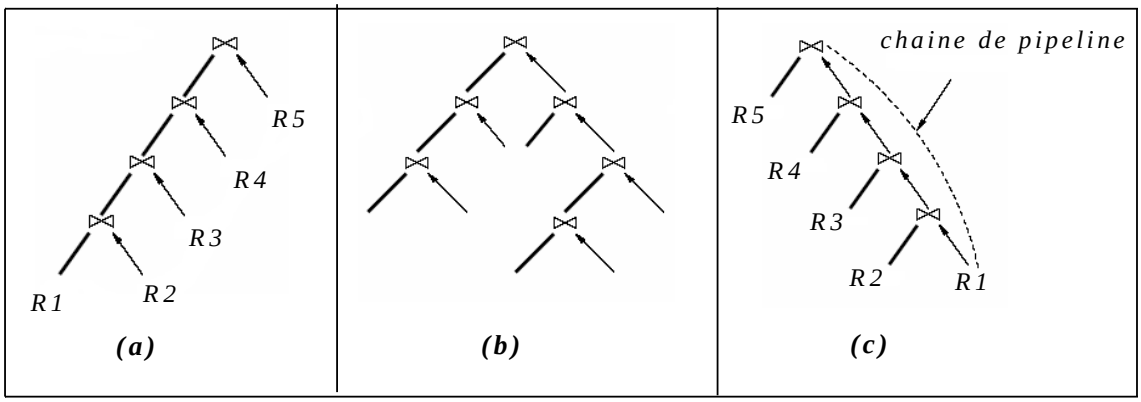

FIG. 4.1 – Les différent types d'arbre d'opérateurs (jointure)

Dans la figure (a) et (c) de Fig. 4.1, la relation R_1 est appelée relation ***Probe***, et les relation $\{R_2, R_3, R_4, R_5\}$ sont appelées des relation ***Builds***.

L'avantage de la ligne (c) par rapport a la ligne (a), est que les résultats des opérateurs intermédiaires ne seront pas stockées (ils sont utilisés tout de suite dans l'opération qui suit dans la ligne, car elle sont des relations probes), contrairement a la ligne (a), où il faut stocker les résultats intermédiaires (car, elles sont des relations builds) pour exécuter les opérations qui suivent.

Plusieurs stratégies ont été proposées pour l'évaluation parallèle d'un tel problème (multitude d'opérations de jointure). Dans [1], ces stratégies sont réparties en quatre principales catégories :

4.4.1 Exécution séquentielle parallèle

L'exécution séquentielle parallèle est la stratégie la plus simple, du fait qu'elle n'utilise que le parallélisme *intra-opérateur* et elle ne nécessite aucun parallélisme *inter-opérateur*. Les opérations de jointure sont évaluées l'une après l'autre de manière parallèle. En effet, à un instant donné une seule opération de jointure est exécutée en parallèle par tous les processeurs. Par conséquent, le temps d'exécution d'une opération est celui du processeur le plus lent.

4.4.2 Exécution synchrone parallèle

En plus du parallélisme *intra-opérateur*, l'exécution synchrone parallèle utilise le parallélisme *inter-opérateur*. Plusieurs opérations de jointures peuvent être exécutées en parallèle par des groupes disjoints de processeurs, avec une seule opération par groupe. Le temps d'exécution d'une opération dépend du nombre de processeurs utilisés et de la manière d'allouer ces processeurs, ce qui constitue le point difficile de cette stratégie.

4.4.3 Exécution segmentée en profondeur droite

Cette stratégie du parallélisme utilise un parallélisme *intra-opérateur* et un parallélisme *inter-opérateur pipeliné* qui est utilisé pour l'évaluation d'une ligne d'opérateurs (branche d'un arbre linéaire droit). La Fig. 4.1 (c) illustre le principe d'un parallélisme *inter-opérateur pipeliné*. Les processeurs s'occupent de toutes les opérations simultanément.

L'avantage majeur d'un arbre linéaire droit qui permet de faire un parallélisme inter-opérateur pipeliné, est qu'il permet d'éviter le stockage des résultats intermédiaires des opérateurs ce qui permet un gain en espace. Dans la Fig. 4.1 (c), le résultat de la relation probe R_1 avec la relation build R_2 ne sera pas stocké il est utilisé immédiatement pour l'opération de jointure avec la relation R_3 et ainsi de suite.

4.4.4 Exécution parallèle complète

L'exécution parallèle complète utilise les trois types du parallélisme : *intra-opérateur*, *inter-opérateur* et *inter-opérateur pipeliné*. Dans cette stratégie, chaque opération de jointure est allouée a un groupe de processeur (*intra-opérateur*), et le parallélisme *inter-opérateur* indépendant et pipeliné et exploité en fonction du graphe d'opérateur.

4.5 Les architectures parallèles

Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent entre eux. En se basant sur le type d'interconnexion des différents composants (les processeurs, les mémoires principales et les disques), les machines parallèles ont été classifiées en trois principales catégories :

4.5.1 Architecture à mémoire partagée (*SM*)

Une machine à mémoire partagée consiste en un nombre de processeurs connectés à un bus permettant l'accès à une mémoire globale et à tous les disques (Fig. 4.2 (a)). Parmi les caractéristiques de cette machine :

La communication se fait via la mémoire commune (l'écriture en mémoire est une communication vers tous les processeurs qui accèdent à cet emplacement), par conséquent, le coût de communication est très faible dû à l'accès direct à cette mémoire, au partage des données et aux mécanismes d'implantation des barrières de synchronisation qui sont peu coûteux. En plus, ce type de machine permet une flexibilité dans le choix du degré du parallélisme et un équilibrage de charge facile.

comme inconvénient, on peut citer l'accès concurrent à la mémoire qui constitue un goulot d'étranglement. Ca implique une limitation d'extensibilité. L'approche de résolution qui conssite à utiliser des caches induit des coûts supplémentaires de leur mise a jour et de leur cohérence au delà d'un certain nombre de processeurs.

4.5.2 Architecture à disque partagé (*SD*)

Dans une machine à disque partagé, chaque processeur dispose de sa mémoire privée (locale) alors que les disques sont partagés par tous les processeurs (Fig. 4.2 (b)).

La communication se fait via un réseau d'interconnexion, par passage de message. Comme la machine à mémoire partagée, ce type de machine permet une flexibilité dans le choix du degré du parallélisme lié au partage de données (disques) sans pour autant souffrir du problème d'extensibilité (qui est le cas dans *SM*), et un potentiel d'équilibrage de charge dû au fait que les résultats intermédiaires sont accessibles a tous les processeurs via les disques.

Les inconvénients de cette machine sont : le coût de gestion de la cohérence des données dans les mémoires des processeurs, et le goulot d'étranglement lié a l'accès concurrent aux disques.

4.5.3 Architecture à disques répartis (*SN*)

Dans une machine à disques répartis, rien n'est partagé entre les processeurs, chaque processeur dispose d'une mémoire privée (locale) et de ses propres disques (Fig. 4.2 (c)).

La communication se fait via le réseau d'interconnexion, par passage de message. Ce type de machines minimise les interférences liées à l'accès concurrent en minimisant le partage des ressources, et assure l'extensibilité. Cependant, il est sensible au déséquilibre de charge de différents processeurs, au coût de communication et de synchronisation qui sont assez élevés.

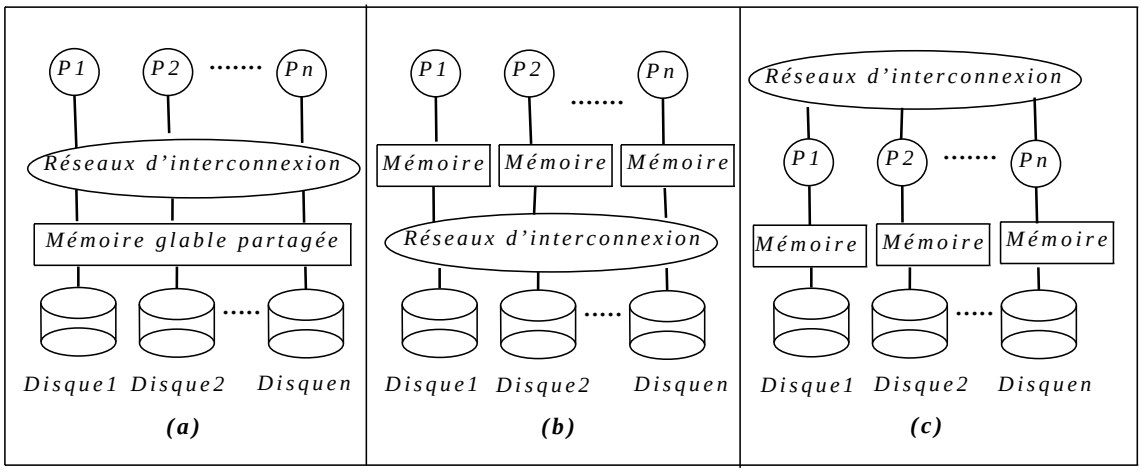

FIG. 4.2 – Les architectures parallèles (a)Shared Memory, (b) Shared Disks, (c) Shared Nothing

4.6 Les techniques de bases du parallélisme

La parallèlistion ou la conception parallèle des algorithmes est une tache très délicate à cause de l'absence d'une méthode bien définie. Ce manque est compensé par la collection d'un ensemble de techniques de base de parallélisation, parmi lesquelles on peut citer :

Les arbres équilibréss (*Balanced Trees*), le saut de pointeur (*pointer Jumping*), diviser pour régner (*Divide and Conquer*), le partitionnement (*Partitionning*), le pipeline (*Pipelining*) et la cascade accélérée (*accelerated cascading*) [25].

Selon la structure du problème à résoudre, il existe des techniques dédiées, comme par exemple la technique *list Ranking* pour des problèmes listes, la *contraction d'arbre parallèle* et *Euler tour* pour des problèmes arborescents, etc.

Dans la suite, nous présentons la *contraction d'arbre parallèle* pour la parallélisation des problèmes dont la structure est un arbre.

4.6.1 La contraction d'arbre [32]

La structure d'arbre joue un rôle fondamental dans les deux types de calculs, séquentiel et parallèle. *Diviser pour régner* (pour *Divide and conquer*) était la technique de base utilisée pour générer des algorithmes parallèles dans le cas d'une structure arbre. Cependant, cette approche a des complications. Pour cela, une approche ascendante appelée *contraction d'arbre parallèle* (pour *parallel tree contraction*) a été définie par *Miller* et *Reif* [32].

Il existe différents algorithmes optimaux de contraction d'arbre parallèle qui s'exécutent sur une mémoire *EREW* avec une complexité temporelle de $O(\log n)$ où n est le nombre de processeurs.

L'algorithme de contraction d'arbre parallèle s'applique à un arbre binaire enraciné. Chaque noeud est, soit une feuille, ou en dehors des noeuds feuilles les noeuds internes possèdent deux fils.

Définition 4.1 (*Contraction d'arbre*)

La contraction d'arbre est une méthode systématique pour contracter un arbre en un seul noeud par une application successive d'une opération de fusion des noeuds feuilles avec ses noeuds parents [25].

La contraction d'arbre parallèle est une contraction d'arbre dont les opérations sont exécutées en parallèle, elle est appelée *CONTRACT* et elle est constituée de deux opérations élémentaires, *RAKE* et *COMPRESS*.

4.6.1.1 Les opérations *RAKE*, *COMPRESS* et *CONTRACT*

Soit $T = (V, E)$ un arbre binaire enraciné composé de n noeuds et d'une racine r, tel que pour chaque noeud v, avec $v \neq r$, $p(v)$ représente le noeud père de v, et on suppose que chaque noeud interne possède, exactement, deux noeuds fils. Deux opérations parallèles, *RAKE* et *COMPRESS*, sont décrites sur T.

1. **L'opération *RAKE* (*SHUNT*)**
   L'opération *RAKE* est celle de la suppression des noeuds feuilles de T, elle nécessite un nombre linéaire d'exécutions pour réduire un arbre complètement non équilibré en un seul noeud. Cette opération primitive est définine comme suit :
   Etant donné un noeud feuille u, l'opération *RAKE* appliquée au noeud u consiste

à supprimer u de l'arbre T et mettre à jour le noeud père $p(u)$. La figure Fig. 4.3 montre l'application de l'opération *RAKE* sur l'arbe illustré.

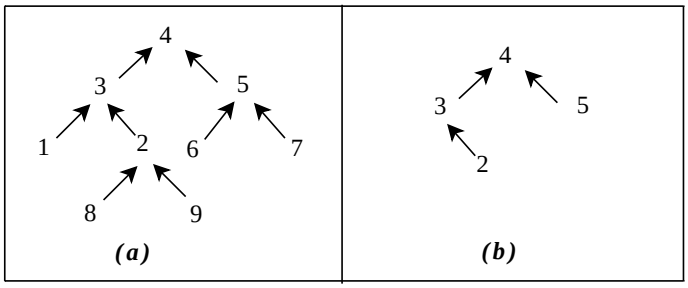

FIG. 4.3 – L'opération *RAKE*

2. **L'opération *COMPRESS***

   Une séquence de noeuds $v_1, v_2, \ldots, v_k$ est appelée chaîne si v_{i+1} est le seul fils de v_i pour $1 \leq i < k$, et v_k possède, exactement, un seul fils qui n'est pas une feuille. L'opération *COMPRESS* et l'opération qui contracte une chaîne de T, en remplaçant les noeuds dont l'identité est impaire par ceux dont l'identité est est paire (les noeuds impaires sont supprimés après avoir appliqué l'opération de calcul appropriée). L'opération *COMPRESS* ne s'applique pas à une chaîne de longueur un.

   La figure Fig. 4.4 montre l'application de l'opération *COMPRESS* sur la chaine des noeuds $\{3, 4, 5, 6, 7\}$.

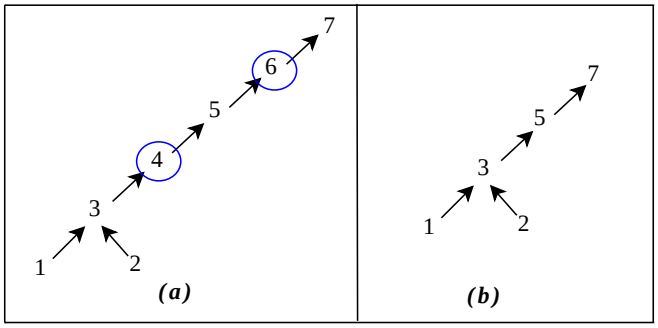

FIG. 4.4 – L'opération *COMPRESS*

3. **L'opération CONTRACT**

   CONTRACT est l'application simultanée des deux opérations *RAKE* et *COMPRESS* sur tout l'arbre T. L'opération *CONTRACT* nécessite $O(\log n)$ exécutions pour réduire l'arbre en un seul noeud.

4.7 Mesure de performance d'un algorithme parallèle

Le but de l'exécution parallèle d'un *CSP* est d'obtenir un gain en performance. Il existe plusieurs métriques permettant de quantifier le gain obtenu par une exécution parallèle. Les plus connues sont l'*accélération* (*speed-up*), l'*efficacité* et la scalabilité (*scale-up*).

Définition 4.2 (*accélération, speed-up*)
Soit un algorithme Al, si T_{seq} est le temps d'exécution séquentielle de Al en utilisant un processus, et T_{par} le temps d'exécution parallèle de Al en utilisant p processeurs, l'accélération (*speed-up*) obtenue par l'exécution parallèle est définie par :

$$S = T_{seq}/T_{par}$$

L'accélération est dite linéaire si $S = p$ (l'accélération idéale), superlinéaire si $S > p$, sublinéaire sinon. Elle est qualifiée d'absolue si T_{seq} est le temps du meilleur algorithme séquentiel, ou de relative si T_{seq} est le temps de l'algorithme parallèle utilisant un seul processus.

Informellement, une *speed-up* idéale indique qu'un algorithme peut être exécutée n fois plus rapide si l'on dispose de n fois plus de ressources.

A partir de l'accélération on peut déduire l'efficacité définie ainsi :

Définition 4.3 (*l'efficacité*)
L'efficacité Eff d'un algorithme parallèle est le rapport entre l'accélération (*speed-up*) effectif et l'accélération idéale, i.e l'accélération divisée par le nombre de processus. Si T_{seq} est le temps d'exécution séquentiel d'un algorithme et T_{par} son temps d'exécution parallèle sur p processeurs, on a :

$$Eff = T_{seq}/(T_{par} * P)$$

L'efficacité idéale est 1, mais en pratique une efficacité supérieure ou égale à $0,95$ est considérée bonne. Parfois, elle devient supérieure à 1 et ceci est à cause d'une accélération superlinéaire.

Définition 4.4(*scale-up*)
Soit T_pD (respectivement T_pD') le temps d'exécution d'un algorithme pour des données d'entrée D (resp. $D' = n * D$) en utilisant P processeurs, on appelle scalabilité, le rapport Scl,

$$Scl = T_pD/T_pnD$$

On dit que la *scale-up* est idéale si elle est égale à 1 quelque soit n.
Une *scale-up* idéale signifie qu'un algorithme qui s'exécute en un temps T, s'exécute dans le même temps pour n fois plus de données si l'on dispose de n fois plus de ressources.

Remarques

1. Un système parallèle idéal est celui qui permet d'obtenir à la fois une *speed-up* linéaire et une scalabilité constante.
2. Généralement, le calcul du *speed-up* est plus facile que celui du *scale-up* à cause de l'existence des opérateurs dont la complexité n'est pas linéairement proportionnelle à la taille de leurs opérandes, d'où la difficulté de calculer un problème n fois plus grand qu'un autre prédéfini.

4.8 Conclusion

L'utilisation de plusieurs processeurs en concurrence fait son apparition pour des besoins d'optimisation des performances de calcul. Dans ce chapitre, nous avons fait un survol sur les differentes techniques et architectures parallèles.

Nous avons détaillé l'une des techniques de parallélisation qui est la contraction d'arbre parallèle, vu que les *CSP* acyclique, dont la résolution peut se faire plus efficacement, a une structure arborescente pour laquelle la technique de contraction d'arbre parallèle est bien adaptée.

Chapitre 5

La résolution parallèle des problèmes CSP

La résolution des problèmes *CSP* est l'une des tâches les plus coûteuses dans le domaine de l'intelligence artificielle. La parallélisation de cette tâche devient une approche prometteuse pour atteindre des performances acceptables. Cependant, peu de travaux sont occupés de cette direction se sont orientés.

Dans ce chapitre, nous allons faire un survol sur les algorithmes de résolution parallèle, des *CSP*, qui existent dans la littérature, en commençant par présenter les différentes approches de résolurion parallèle, les algorithmes de résolution parallèle des *CSP* cyclique, et ceux des *CSP* acyclique en décrivant l'algorithme *PTAC* [48] qui est l'un de ces derniers algorithmes.

Nous allons, aussi, proposer un algorithme parallèle de résolution, qui est une parallélisation de l'algorithme *S-HBR* (proposé dans le *Chapitre 3*), en utilisant une technique de pipeline et de contraction d'arbre parallèle. La première est utilisée dans la parallélisation des sous problèmes eux-mêmes en vu d'optimiser en espace mémoire. Pour la deuxième technique, nous proposons deux schémas pour sa mise en oeuvre, le plus éfficace est celle utilisée pour une résolution parallèle du problème global.

5.1 Introduction

L'approche parallèle constitue le moyen le plus efficace pour l'optimisation de la complexité d'un algorithme, en teme de temps et d'espace. Ces deux points représentent la difficulté d'un problème *CSP*.

5.2 La complexité parallèle des problèmes CSP

La complexité parallèle des problèmes *CSP* est dans la classe *NC*, pour un CSP binaire dont le réseau de contraintes est un arbre, et dans la classe *P-complete*, en général [48].

Definition 4.1
Un problème est dans la classe *NC* ($NC \subseteq P - complete$) si et seulement s'il existe un algorithme parallèle pour ce problème dans le modèle *SM* (*Shared Memory*) (voir *4.5.1*), qui prend un temps polylogarithmique en utilisant un nombre polynomial de processeurs.

5.3 Les approches de résolution parallèle

Les processeurs utilisés dans une résolution parallèle peuvent collaborer pour la résolution du problème comme ils peuvent se concurrencer, ceci implique qu'une résolution parallèle se présente dans l'une des deux approches : la concurrence ou la coopération.

5.3.1 L'approche concurrente

La résolution concurrente consiste à lancer plusieurs solveurs différents, traitant d'une manière indépendante le même problème à résoudre. La différence entre les solveurs est le point sur lequel repose l'efficacité de cette approche, cette différence peut être faible ou forte. La différence est faible dans le cas d'utilisation d'un même solveur avec des heuristiques différentes, par exemple, l'utilisation d'un algorithme énumératif avec des heuristique sur le backtrack et d'autres sur l'instanciation. La différence est forte dans le cas de l'utilisation des solveurs de différentes classes, complet et incomplet, énumératif et avec décomposition, etc.

L'un des points forts de cette approche est la rapidité de résolution dans le cas où l'un des solveurs est mieux adapté au problème. Ceci est dû à la différence entre les solveurs. Cependant, l'inconvénient majeur est l'absence de coopération entre les solveurs, en effet, le même espace de recherche peut être exploré par plusieurs solveurs ce qui conduit à une mauvaise performance, surtout si l'espace ne contient pas de solutions, d'où le recours a une approche coopérative.

5.3.2 L'approche coopérative

Dans cette approche, les solveurs collaborent entre eux pour la résolution du problème. Cette collaboration se présente dans l'echange de certaines informations utiles

pour le fonctionnement de chauqu'un d'entre eux. Ces informations sont, généralement, appélées des *good* ou *nogood* (voir *Définition 1.14*).

Il a été démontré [24], l'inefficacité des méthodes concurrentes, qui est le contraire pour l'approche coopérative. Donc, on peut conclure que, pour les problèmes *CSP*, l'approche coopérative peut apporter des résultats meilleurs que celles de l'approche concurrente.

5.4 Les algorithmes de résolution parallèles des CSP

Pour mettre en oeuvre la stratégie parallèle, deux méthodes sont possibles, soit on parallélise des algorithmes séquentiels, soit on propose de nouveaux algorithmes parallèles. Pour la première, et comme nous avons vu dans le *Chapitre 1* qu'il existe trois grands axes de la résolution séquentielle, le parallélisme est mis en oeuvre pour ces trois axes.

Les deux types de parallélisme, à mémoire partagée et à mémoire distribuée (*Chapitre 4*), sont exploités pour la résolution des *CSP*.

En se basant sur le critère de la cyclicité, les algorithmes de résolution parallèle des *CSP* se divisent en deux grandes classes, ceux de la résolution des *CSP cyclique* et ceux de la résolution des *CSP acyclique*.

5.4.1 La résolution parallèle des CSP cycliques

Les algorithmes de cette classe sont la parallélisation des algorithmes des deux premières approches de résolution séquentielle, les *algorithmes énumératifs* (voir *1.7.1* et *1.7.2*) et *les algorithmes de filtrage* (voir *1.7.3*).

5.4.1.1 Parallélisation des algorithmes énumératifs

Pour la résolution distribuée, une approche suggérée consiste à distribuer l'algorithme *BT*. Cette approche est représenté par l'algorithme *Backtracking distribué (DiBT)* développé dans [46]. L'algorithme *DiBT* n'implique aucun échange de *nogoods* mais il est basée sur des échanges de sous solutions consistantes entre les agents. Cependant, il fut prouvé incomplet, c'est-à-dire qu'il ne garantit pas un parcours complet de l'espace de solutions.

Un autre algorithme, développé dans [6] nommé *Asynchrone Backtracking (ABT)* consiste à la distribution du *BT* ainsi qu'au partage des *nogoods*, entre les différents agents contribuant à la résolution du problème, pour diminuer l'espace de recherche.

L'échange de *nogoods* permet à ces agents d'éviter de nombreuses situations d'inconsistance.

L'enregistrement des *nogoods*, dont la manipulation en cours de la recherche peut représenter un aspect important de l'efficacité, a été étudié en détail par *Terrioux* dans [44]. L'étude de cette idée sert à spécifier les algorithmes de la reconnaissance des *nogoods* ainsi que de la détermination de quels *nogoods* sont utiles et doivent être enregistrés.

Pour la résolution parallèle, *Habbas, Krajecki et al* [47] développèrent une approche de décomposition de problèmes. Cette méthodes consistent en une décomposition des domaines du problème ou encore en une distribution de l'arbre de recherche sur plusieurs processeurs. Cette décomposition permet de mettre à profit la résolution de plusieurs sous problèmes du problème initial afin de résoudre ce dernier.

5.4.1.2 Parallélisation des algorithmes de filtrage

Pour la distribution des algorithmes d'arc consistance, un algorithme, qui démontre l'exactitude de *AC4 distribué* fonctionnant avec un procédé d'échanges de messages entre les ordinateurs participant à la résolution, a été proposé dans [34].

De plus, Un algorithme parallèle *DRA5* a été proposé dans [22] pour la vérification parallèle de l'arc consistance.

5.4.2 La résolution parallèle des CSP acycliques

Les algorithmes de cette classe sont la parallélisation des algorithmes de résolution par décomposition de l'approche séquentielle. Dans la littérature, il y a peu de travaux qui se charge de la parallélisation de la résolution après la décomposition, parmi lesquels,on site l'algorithme *PTAC* (pour *Parallel Tree Arc Consistency*) proposé par *Zhang et Mackworth* [48].

Dans la suite nous allons prendre en détaille cette algorithme, vu qu'une partie de notre travail repose beaucoup plus sur ce que fait ces deux personnes, mais avant, nous donnons la règle générale d'une résolution parallèle par décomposition.

La résolution parallèle d'un *CSP* après une décomposition se déroule en trois étapes :

1. Décomposition du problème global en des sous problème a l'aide d'une méthode de décomposition.
2. Résolution des sous problèmes en parallèle.
3. Résolution du problème global.

On peut remarquer qu'il y a un parallélisme inhérent dans la deuxième étape. Pour la première étape, on peut aussi avoir des versions parallèles pour la décomposition mais peu de travaux s'occupent de ça, et pour la parallélisation de la résolution globale, *PTAC* est un algorithme qui la fait en utilisant la contraction d'arbre parallèle.

5.4.2.1 L'algorithme PTAC (Parallel Tree Arc Consistency)

L'algorithme *PTAC* calcule l'arc consistance non orientée sur un *CSP* acyclique, et puis la résolution se fait par un parcours *botom-up* ou *top-dowm* de l'arbre de contraintes après la mise à jour des relations. Il consiste en deux principales phases :

- Une phase de contraction de l'arbre *ContractAC*.
- Une phase d'expansion de l'arbre *ExpandAC*.

Algorithm 13 Algorithme Parallel Arc Consistency (PTAC)

1: **Entrée** :Un Join Tree enracinée T.
2: **Sortie** :une Join Tree arc consistant T'.
3: $T'' = ContractAC(T)$,
4: $T' = ExpandAC(T'')$,

L'entrée de l'algorithme *PTAC* (voir *Algorithme 17*) est le Join Tree du *CSP* à résoudre. On peut le voir comme une parallélisation de la troisième étape de la résolution d'un *CSP* après sa décomposition (voir *5.4*) i.e. après la résolution des sous problèmes, on peut appliquer le *PTAC* pour faire la résolution parallèle du problème entier.

Le principe est d'appliquer une contraction d'arbre parallèle (*ContractAC*) suivie d'une expansion (*ExpandAC*), aussi, parallèle de l'arbre. La phase de contraction consiste à réduire l'arbre en un seul noeud, et l'expansion est l'opération inverse.

La contraction d'arbre parallèle est présentée en détail dans le *Chapitre 4*.

1. **La procédure *ContractAC***

   La procedure *ContractAC* (voir *Algorithme 18*) fait la contraction parallèle du Join Tree du *CSP*, elle peut être vue comme une parallélisation des trois premières étapes de l'algorithme *Acyclic Solving* de *Rina Dechter* (voir *3.2.2 Algorithme 9*). Chacune des opérations *RAKE* et *COMPRESS* (opérations principales d'une contraction d'arbre parallèle, voir *chapitre 4*) est associée, respectivement, une opération de semi jointure et une opération de jointure.

   Soit a un noeud de T, on appelle $pre(a)$ (resp. $fils(a)$) le noeud père (resp. le noeud fils) de a. $cp(a)$ dénote le père contracté du noeud a. $r(R)$ désigne la relation r (ensemble de tuples) sur le schéma R, et $con(a)$ désigne la contrainte du noeud a.

L'opération *RAKE* effectue un filtrage de la relation d'un noeud par les relations de ses noeuds fils (en appliquant une opération de semi jointure).

Par contre, l'opération *COMPRESS* suppose qu'il y a deux noeuds a_i et a_{i+1} consécutifs d'une chaîne (voir *4.6.1.1*), tel que $pre(a_i) = a_{i-1}$ et $fils(a_{i+1}) = a_{i+2}$, avec $con(a_k) = r_k(R_k)$ et $L_k = R_k \cap R_{k+1}$, pour $i - 1 \leq k \leq i + 1$.
Quand a_i est identifié par a_{i+1} (en appliquant l'opération *COMPRESS*), l'opération $\prod_{L_{i-1} \cup L_i \cup L_{i+1}}(r_i \bowtie r_{i+1})$ est appliquée, pour produire la contrainte pour le nouveau noeud.

Algorithm 14 Algorithme ContractAC

Entrée :Une Join Tree enracinée $T =< A, E >$.
Sortie :Joint Tree arc consistance dirigée.
pour chaque noeud $a \in A/\{racine\}$ **en parallèle faire**
  $r(R) = con(a)$, $p(P) = con(pre(a))$
  si a possède des fils feuilles **alors**
    pour chaque noeud fils de contrainte $l(L)$ **faire**
      $r = r \ltimes l$, supprimer c, $cp(c) = a$.
    fin pour
  sinon
    si $chane(a)$ **alors**
      Créer un noeud a', $c(C) = con(fils(a))$,
      $p'(P') = con(pre(pre(a)))$, $p'' = (C \cap R) \cup (R \cap P) \cup (P \cap P')$,
      $P'' = (r \bowtie p)[p'']$. $con(a') = p''(p'')$,
      $pre(fils(a)) = a'$, $fils(a') = fils(a)$,
      $fils(pre(pre(a))) = a'$, $pre(a') = pre(pre(a))$, $cp(a) = a'$,
      $cp(pre(a)) = a'$.
    finsi
  finsi
fin pour

Cette procédure est itérée jusqu'à ce que T ne contienne qu'un seul noeud, qui est le noeud racine. Le résultat est un arbre arc consistant dirigé.

2. **La procédure ExpandAC**
   La procédure *ExpandAC* (voir *Algorithme 19*) effectue l'arc consistance dans le sens inverse de *ContractAC* (de la racine vers les feuilles). Elle fait propager les solutions d'un noeud père vers ses noeuds fils, en marquant initialement le noeud racine et a chaque fois q'un noeud est marqué on calcul les solutions pour ces noeud fils qui seront marqués à leur tour.

   Le nombre d'itération de cette procédure est égal au nombre d'itération de la procédure *ContractAC*.

Algorithm 15 Algorithme ExpandAC

1: **Entrée** :Résultat de ContractAC, $T =< A, E >$.
2: **Sortie** :Un Join Tree arc consistant.
3: Marquer (Racine) = 1.
4: **pour** chaque noeud $a \in A/\{racine\}$ **en parallèle faire**
5: **si** marquer $(cp(a))$ **alors**
6: $r(R) = con(a)$, $p(P) = con(cp(a))$, $r = r \ltimes p$.
7: Marquer $(a) = 1$.
8: **finsi**
9: **fin pour**

La complexité de *PTAC* est de $O(\log n)$ en utilisant $O(n)$ processeurs dans un modèle *EREW SM* (*Exclusif Read Exclusif Write Shared Memory*). Si l'arbre T n'est pas binaire une transformation est effectué en un temps $O(\log n)$ en parallèle.

5.5 L'algorithme de résolution parallèle P-HTR (*Parallele HyperTree Resolution*)

5.5.1 Introduction

Nous avons vu que la résolution d'un problème *CSP* après une décomposition hypertree se déroule en trois étapes (voir *3.4*) :

1. Compléter la décomposition hyper Tree.
2. Pour chaque noeud p, calculer une nouvelle relation R_p qui est la projection des jointures des relations des contraintes dans λp sur l'ensemble des variables de χp.

$$Rp = Join(\lambda p)[\chi p].$$

   Le résultat est un arbre de jointure d'un problème acyclique équivalent au problème original.
3. Applique un algorithme adapté (ex. Acyclic Solving) sur l'arbre de jointure pour la résolution globale du problème.

Dans cette séction, nous allons proposer un algorithme parallèle *P-HTR* (*Parallel Hyper Tree Resolution*), qui assure à la fois une résolution parallèle des sous problèmes, ainsi que du problème globale.

En effet, *P-HTR* est une parallélisation de l'algorithme *S-HBR* proposé dans le *Chapitre 3*. Ce dernier est constitué de deux algorithmes, l'algorithme *SP-HBR* qui

fait la résolution des sous problèmes, et l'algorithme *A-HBR* qui fait la résolution du problème globale. L'algorithme *P-HTR* réalise une pârallélisation de *SP-HBR* dans un algorithme appelé *P-SBR* (*Parallel Sub Problem Resolution*), et une paralléisation de *A-HBR* dans un algorithme appelé *P-slving* (*Parallel solving*). Ceci, En utilisant deux téchniques de paralléisation, la contraction d'arbre parallèle et le pipeline.

Afin que notre algorithme assure une optimisation en complexié, à la fois spaciale et temporelle, nous avons procédé ainsi :

1. Pendant la résolution des sous problèmes (*P-SBR*), les résultats des opérations de jointure sont la cause d'une mauvaise complexité spatiale. Nous avons vu dans *4.3.2* que l'avantage de la techniques pipeline est qu'elle permet d'éviter le stockage des résultats intermédiares, car ils seront utilisés immédiatement dans les opérations qui suivent dans la ligne du pipeline. Ainsi, nous utilisons cette technique pour optimiser la complexité spaciale de l'algorithme pendant cette phase de résolution.

2. Pour une optimisation temporelle de la résolution globale du problème (*P-solving*), la contraction d'arbre parallèle est l'approche la plus adaptée vu que le problème a une structure d'hyperarbre. Mais, la contraction d'arbre connue (voir *4.6.1*) se fait, à chaque itération, pour chacun des noeuds père avec un de ses noeuds fils feuilles. Pour une augmentation du degré du parallélisme dans cette opération, nous avons developpé deux techniques de contraction d'arbre parallèle :

   (a) Une repose sur la technique du pipeline, dans laquelle nous construisons une ligne du pipeline constituée des opérateurs de semi jointure d'un noeud père avec tous ses noeuds fils (voir plus loin *5.5.4.1*), ceci pour tous les noeuds père.

   (b) Une autre repose sur le partage de la relation père (la ressource critique) entre les processeurs dont chacun s'occupe d'un operateur de semi jointure, ce qui permet la manipulation simultanée de la relation père et par conséquent est réglée.

3. Dans les deux phases de résolution, celles des sous problèmes et du problème entier, nous avons utilisé la technique de hachage pour une accélération des traitements.

Une étude comparative a été mise en oeuvre pour l'évaluation des performances des deux techniques (voir 5.7.4).

5.5.2 Algorithme *P-HTR*

L'algorithme *P-HTR* (cf. Alg. 20) transforme l'arbre dont les noeuds sont des sous ensembles de contraintes formant des sous problèmes en un problème acyclique, dans une première étape en appliquant l'algorithme *P-SBR* pour la résolution de chacun des sous problème (ligne 4), ce qui génère une seule contrainte par chaque noeud. Dans une deuxième étape, PHTR résout le *CSP* acyclique obtenu en utilisant l'algorithme *P-Solving* (ligne 6).

Algorithm 16 Parallel-Hyper Tree Resolution (P-HTR)

1: **Entrée** : Un hypertree $< T, \chi, \lambda >$ d'un hyper- graphe $H = (V, E)$, avec :T est l'arbre, χ associe à chaque t de T un ensemble de noeuds $\chi(t) \subset V$, et λ, un ensemble d'arêtes $\lambda(t) \subset E$.
2: **Sortie** : Génerer une solution au problème.
3: **pour** chaque noeud n de T **en parallèle faire**
4: P-SBR (n), Appliquer P-SBR sur le noeud n.
5: **fin pour**
6: soit T' le join tree representant le problème acyclic généré, avec pour tout $t' \in T'$, t' contient une et une seule contrainte.
P-Solving (T'), Appliquer l'algorithme P-Solving sur T'.
7: **End.**

L'algorithme suppose que les contraintes sont représentées en extension de façon a ce que chacune d'elles est définie par une relation composée de tous les tuples autorisés.

5.5.3 L'algorithme P-SBR

L'algorithme P-SBR est un algorithme de résolution parallèle des sous problèmes. Il assure une parallélisation de l'algorithme *SP-HBR* (voir *3.5.3*),

Le résultat est une seule table contenant les tuples des valeurs autorisées par toutes les contraintes du sous problème.

Après l'exécution de l'opération de jointure, les noeuds feuilles sont représentés sous forme d'une paire $< \chi(p), Hrel(p) >$ qui est définie comme suit :

- $\chi(p)$: est l'ensemble des variables du noeud p,
- $Hrel(p)$: est la relation de la contrainte générée par l'operation de jointure, dont les tuples sont hachés sur les variables d'intersection avec le noeud père dans l'hyper tree.

Initialement, L'algorithme *P-SBR* (voir *Algorithme 21*) procède de la même manière que *SP-HBR*. Après avoir effectué un ordre entre les relations , où la première relation est la relation probe et le reste des relations sont des relations build. Nous construisons

Algorithm 17 Sub Problem Hash Based Resolution ($\mathcal{S}$-$\mathcal{HBR}$)

Entrée : Un noeud n représenté par un sous graphe de contraintes $G = \langle \chi(t), \lambda(t) \rangle$ d'un hypertree $\langle T, \chi, \lambda \rangle$ d'un hypergraphe $H = (V, E)$ tel que $\chi(t) \subset V$ et $\lambda(t) \subset E$ et $t \subset T$.
Sortie : Génèrer les solutions pour le sous problème n représentée par une relation notée Res.
Effectuer un ordre $d = \mathcal{R}_1, \ldots, \mathcal{R}_k$ entre les contraintes de $\mathcal{C}_i$.
Construire une ligne de pipeline $\mathcal{L} = \mathcal{R}_1, ..., \mathcal{R}_k$, où $\mathcal{R}_1$ est la relation probe et $\forall i \in \{2, \ldots, k\}$, $\mathcal{R}_i$ est la relation build.
pour chaque relation build **en parallel faire**
  $\eta_i = \bigcup_{j \in m}(\mathcal{R}_i \cap \mathcal{R}_j)$ avec $m = \{$ les identités des relations qui précèdent $\mathcal{R}_i\}$
  Calculer Hash(t, η_i), $\forall t \in \mathcal{R}_i$
fin pour
$\mathcal{R}es_i = Pipeline(\mathcal{L})$, avec $\mathcal{R}es_i$ est le résultat de la jointure.
si $\mathcal{R}es_i = \phi$ **alors**
  Exit. /* le problème n'a pas de solution*/
finsi

La procédure Pipeline ($\mathcal{L}$)
pour chaque étage i de la ligne pipeline **en parallel faire**
  si i = 1, la première opération de jointure **alors**
    pour chaque tuple $t \in \mathcal{R}_1$ **faire**
      $h = Hash(t, \eta_j)$
      si la partition p correspendante à h dans $\mathcal{HT}_{i+1} \neq \phi$ **alors**
        $\forall t' \in p, \mathcal{R}_{tmp}[i+1] = \mathcal{R}_{tmp}[i+1] \cup (t \bowtie t')$.
      finsi
    fin pour
  sinon
    pour chaque tuple $t \in \mathcal{R}_{tmp}[i-1]$ **faire**
      $h = Hash(t, \eta_j)$
      si la partition p correspendante à h dans $\mathcal{HT}_{i+1} \neq \phi$ **alors**
        si i = k, la dernière opération de jointure **alors**
          $\forall t' \in p, \mathcal{R}_{tmp}[i+1] = \mathcal{R}_{tmp}[i+1] \cup (t \bowtie t')$.
        sinon
          $\mathcal{R}es_i = \mathcal{R}es_i \cup t$.
        finsi
      finsi
    fin pour
  finsi
fin pour

une ligne de pipeline des opérateurs de jointure (ligne 6). Le traitement de la jointure sera fait selon le principe de la jointure par hachage pipelinée . Pour cela les relations build sont hachées sur les variables d'intersection (η_i) avec les relations précédentes dans l'ordre (ligne 9). La jointure se fait par un appel à la procédure *pipeline*.

Le parallélisme dans cet algorithme est assuré par la procedure *Pipeline* (ligne 27 à 38). En effet, dans cette procédure, chaque processeur s'occupe d'un etage de la ligne du pipeline, ainsi les processeurs de différents opérateurs font un parallélisme *inter-opérateur*. A chaque étage de la ligne du pipeline correspond une opération de jointure avec ses opérandes R_i (la relation probe) et R_{i+1} (la relation build). Par conséquent, η_{i+1} (ligne 30) désigne les variables d'intersection de R_{i+1} avec les relations qui la précèdent. Dans cette procedure nous hachons chaque tuple t de la relation probe Res_i (ligne 29) (initialement $Res_i = R_1$) sur l'ensemble η_{i+1}, s'il correspond à une entrée p de la table de hachage de la relation R_{i+1}, nous faisons la concaténation de t avec tous les tuples de p (ligne 32), sinon, le tuple t sera éliminé.

Si l'un des résultats de jointure d'un noeud est vide, ça implique que le problème n'a pas de solutions (ligne 12).

Après l'exécution de toute la ligne de pipeline, dans le noeud n, on fait le hachage du tuple résultat sur les variables d'intersection du noeud n avec son noeud père si le noeud n est une feuille (ligne 21), car la relation du noeud n constitue la relation build pour l'opération de semi jointures (algorithme *P-solving*).

5.5.4 L'algorithme parallèle P-Solving

L'algorithme *P-solving* est la version paralléle de l'algorithme *A-HBR* (voir *3.5.4*). Il consiste en deux opérations, l'opération de contraction *P-CONTRAT* (pour *Parallel CONTRACT*), et l'opération *P-SEARCH* (pour *Parallel SEARCH*). La première assure une parallélisation de l'operation *CONTRACT*, tandis que la deuxième fait la parallélisation de l'operation *S-SEARCH*.

Algorithm 18 P-Solving

1: **Entrée** :Une Join Tree T.
2: **Sortie** : Déterminer une arc consistance orientée et génère une solution.
3: $T' = P - CONTRACT(T)$,
4: $PS - SEARCH(T')$
5: **End.**

L'algorithme parallèle *P-Solving* (Algorithme 22) est l'application simultanée de *P-CONTRACT* et *P-SEARCH*.

5.5.4.1 L'algorithme P-CONTRACT

L'opération *P-CONTRACT* force l'arc consistance orientée dans l'arbre d'entrée. Elle est constituée de deux opérations, *P-RAKE* (pour *Parallel RAKE*), et *P-COMPRESS* (pour *Parallel COMPRESS*), avec *RAKE* et *COMPRESS* sont les opérations de bases d'une contraction d'arbre parallèle (voir *4.6.1*). Pour l'opération P-RAKE, nous avons proposé deux alternatives.

Algorithm 19 Pipelined-CONTRACT(P-CONTRACT)

1: **Entrée** : Un Join Tree $T =< A, E >$, A est l'ensemble des noeuds, et E est l'ensemble d'arcs.
2: **Sortie** : Determiner une consistance d'arc dirigée.
   HT_i : la table de hachage de la relation a_i.
3: **pour** chaque noeud a de T **en parallèle faire**
4: **si** t possède des noeuds feuilles **alors**
5: $P - RAKE_1$ *(a)* ou $P - RAKE_2$ *(a)*
6: **finsi**
7: **si** *chaîne(a)* **alors**
8: *P-COMPRESS (a)*
9: **finsi**
10: **fin pour**

Dans cet algorithme (cf. *Alg. 23*) et pour chauqe noeud n de l'arbre de jointure, si n possède des noeuds fils feuilles on lui applique l'une des opérations *P-RAKE* ($P - RAKE_1$ ou $P - RAKE_2$). Si n fait partie d'une chaine, on lui applique l'opération *P-COMPRESS*.

5.5.4.2 L'operation P-RAKE

Dans notre proposition parallèle, nous avons essayé d'augmenter le degré du parallélisme de l'opération P-RAKE par deux facons différentes :

La première ($P - RAKE_1$) consiste à intégrer la technique du pipeline dans l'opération RAKE de telle sorte que pour chaque noeud père, au lieu d'effectuer, à chauque fois, une contraction avec l'un de ses noeuds fils feuilles, nous construisons une ligne du pipeline de ce noeud avec tous ses noeuds fils feuille.

La figure (a) Fig.5.1 illustre les étapes de contraction parallèle de l'arbre T en appliquant l'opération *RAKE*, et la figure (b) illustre cette contraction en appliquant l'opération *P-RAKE*. Dans cet arbre, les nœuds $\{n_4, n_5, n_6\}$ sont les noeuds feuilles du noeud n_3, et n_2 est le noeud père du noeud n_3. L'opération *P-RAKE* appliquée au noeud n_3 résulte dans un nouveau arbre contracté T' dans lequel les noeuds $\{n_4, n_5, n_6\}$ sont m̂arqués, le noeud n_3 est transformé comme suit :

– $\chi_{T'}(n_3) = \chi_T(n_3)$
– $Hrel_{T'}(n_3) = Hrel_T(n_4) \ltimes Hrel_T(n_5) \ltimes Hrel_T(n_6)$ (ligne du pipeline).

Avec $\ltimes$ dénote l'opérateur de semi jointure, et la relation $Hrel_{T'}(n_3)$ est haché sur les variables d'intersection du noeud n_3 avec son noeud père n_2. Ainsi, l'opération *P-RAKE* est appliquée à un sous arbre avec la relation probe est celle du noeud racine, et les relations build sont celles des noeud fils.

La deuxième ($P - RAKE_2$) façon repose sur le fait que la cause de la séquentialisation réside dans l'existence d'une ressource unique, qui est la relation R_p du noeud père. Cette ressource doit être manipulée simultanément par tous les processeurs. Donc, pour qu'ils puissent l'utiliser en meme temps, elle doit être partagée entre eux. Ainsi, cette technique consiste à partager la relation R_p entre les processeurs qui s'occupent du calcul de la semi jointure de cette dernière avec ses relations fils. La manipulation des partitions de la relation père se fait à tour de role d'une manière périodique entre les processeurs. La figure FIG.5.2 montre le déroulement de cette opération.

Initialement, chaque processeur P_i manipule une partition T_i de la relation du noeud père. Ensuite, il passe à la manipulation de la partition suivante T_{i+1} après que cette dernière sera libérée par le processeur qui l'occupe P_{i+1}. Ainsi, la partition T_i est libérée par le processeur P_i et elle sera manipulée par le processeur P_{i-1}, et ainsi de suite jusqu'à ce qu'il manipule toutes les partitions.

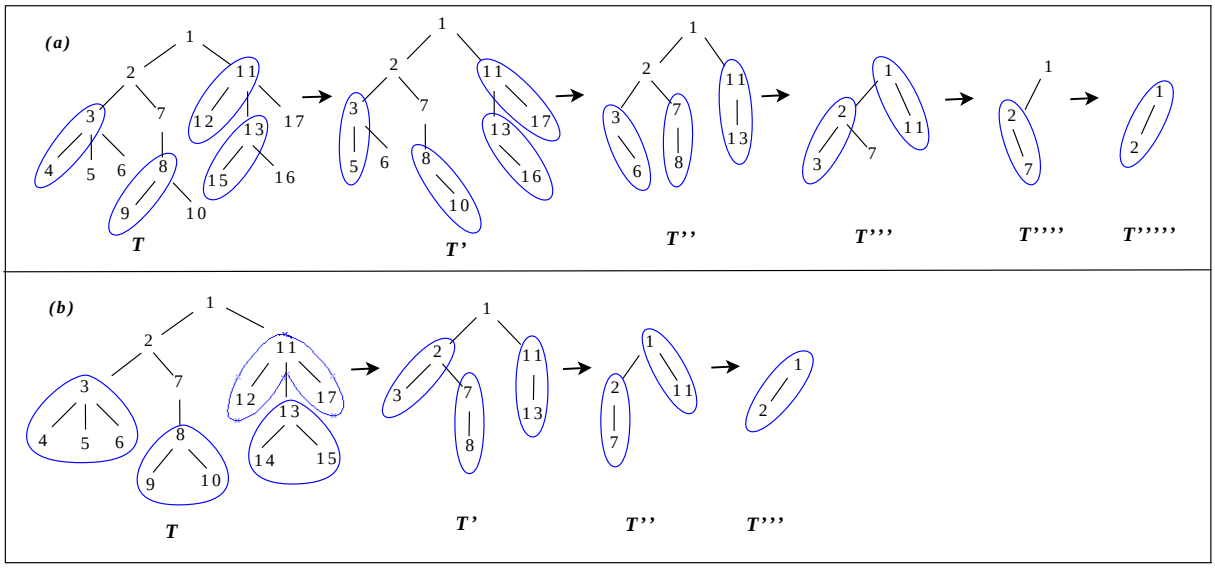

FIG. 5.1 – (a)Contraction avec l'opération (*RAKE*)(b) Contraction avec l'opération (*P-RAKE*)

Dans les deux cas $P - RAKE_1$ et $P - RAKE_2$, nous avons une fragmentation horizontale des relations par une fonction de hachage.

Algorithm 20 La procédure P-RAKE (a_1)

Soient $a_2, a_3, \ldots, a_j$ sont des noeuds fils de a_1. **P-RAKE (1)**
Construire la ligne du pipeline $L = (a_1 \propto a_2 \propto \cdots \propto a_j)$.
$r_1 \leftarrow a_1$,
pour chaque étage i de la ligne L **en parallèle faire**
  pour chaque tuple t de r_i **en parallèle faire**
    Calculer $Hash(t, \eta_{i+1})$.
    si la partition correspondante p_k dans HT_{i+1} n'est pas vide **alors**
      $r_{i+1} = r_{i+1} \cup t$,
    finsi
    si c'est le dernier opérateur de la ligne **alors**
      $res = res \cup t$
    finsi
  fin pour
fin pour
$a_1 \leftarrow res$,

P-RAKE (2)
Soit R_{a_1} la relation du noeud a_1 partitionnée en n partition (n est le nombre de relations fils).
pour chaque noeud fils n_i **en parallèle faire**
  pour $j = 1$ à n **faire**
    $k \leftarrow (i + j) mod$ n.
    si P_k est libre **alors**
      pour chaque tuple t de P_k **faire**
        Calculer $Hash(t, \eta_{k+1})$.
        si la partition correspondante dans HT_{i+1} est vide **alors**
          Marquer t
        finsi
      fin pour
    finsi
  fin pour
fin pour
pour chaque tuple t de a_1 **faire**
  si t est marqué **alors**
    $R_{a_1} \leftarrow R_{a_1} - t$.
  finsi
fin pour

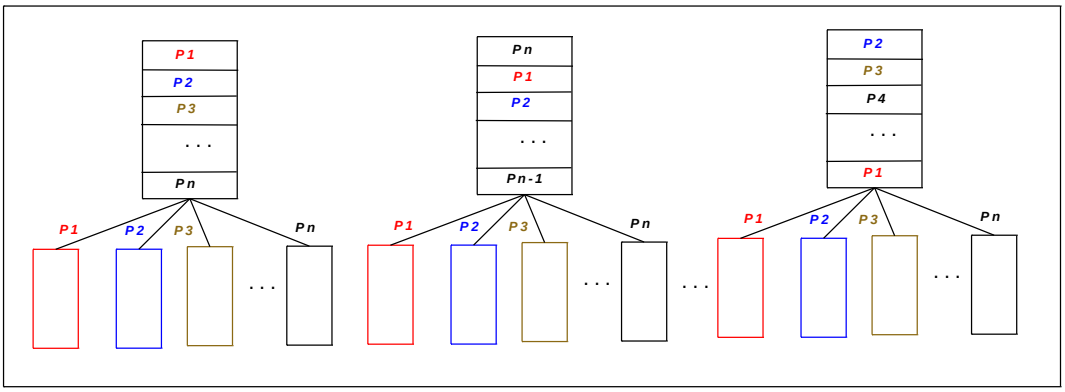

FIG. 5.2 – (a)Contraction avec l'opération ($RAKE_2$)

5.5.4.3 L'operation P-COMPRESS

L'opération P-COMPRESS est une exécution pipeline de l'opération *COMPRESS*. Elle s'applique sur une chaîne. Soit $n_1, n_2, \ldots, n_j$ une chaîne dans un arbre T avec n_{i+1} est le seul fils de n_i, l'opération *P-COMPRESS* appliquée a cette chaîne résulte dans un nouvel arbre contracté T' dans lequel le noeud n_1, est transformé ainsi :

- $\chi_{T'}(n_1) = \chi_T(n_1)$
- $Hrel_{T'}(n_1) = \Pi_{\chi(n_1)}(Hrel_T(n_2) \bowtie Hrel_T(n_3) \bowtie \ldots \bowtie Hrel_T(n_j))$.

Algorithm 21 La procédure P-COMPRESS (a)

Soit $a_1, a_2, \ldots, a_j$ est la chaîne dont a fait partie.
Construire la ligne du pipeline $L = (a_1 \bowtie a_2 \bowtie, \ldots, \bowtie a_j)$.
$r_1 \leftarrow a_1$,
pour chaque étage i de la ligne L **en parallèle faire**
  pour chaque tuple t du noeud r_i **en parallèle faire**
    Calculer $Hash(t, \eta_{i+1})$.
    si P_k est la partition correspondante dans la table de hachage **alors**
      $t = t \bowtie t_k, t_k, \forall t_k \in Pk$.
      $r_{i+1} \leftarrow r_{i+1} \cup t$
    finsi
    si c'est le dernier opérateur de la ligne **alors**
      pour chaque noeud a_i de la chaîne **faire**
        $t \leftarrow \Pi_{\chi(a_i)}(t)$.
        Calculer $Hash(t, Y_{a_i})$,
        $a_i \leftarrow a_i \cup t$
      fin pour
    finsi
  fin pour
fin pour

, qui consiste à faire une ligne de pipeline de jointure entre les noeuds de cette chaine (ligne 29 à 44), et à chaque fois qu'on prend un tuple de la relation probe (ligne 30), on le joint avec tous les tuples de la partition correspondante dans la table de hachage de la relation build (ligne 33), et on le fait passer à la relation probe de l'opérateur de l'étage suivant (ligne 34). Après l'exécution du dernier opérateur, on projette les tuples résultats sur l'ensemble de variable χ de chacun des noeuds de la chaine (ligne 38), on le hache sur les variables d'intersection de chacun des noeuds avec son noeud père (ligne 39), et puis en ajoute le tuple résultat à la relation du noeud correspondant.

5.5.4.4 L'algorithme PS-SEARCH

Algorithm 22 PS-SEARCH

1: **Entrée** :Le résultat de l'opération $P - CONTRACT$, $T' = \langle A', E' \rangle$, avec $\forall a' \in A', a' = \langle \chi(a'), Hrel(a') \rangle$.
2: **Sortie** : Génère une solution au problème *CSP*.
$sol = \emptyset$, la solution du probléme. *//ensemble des valeurs des variables.*
Y_a : est l'ensemble des variables d'intersection du node a avec son noeud père.
3: Prendre un tuple quelconque $t \in Hrel(racine)$, $sol = t$
4: Marquer le noeud racine
5: **pour** chaque noeud a de T' **en parallèle faire**
6: **si** le noeud père de a est marqué **alors**
7: Hash(sol, Y_a), prendre un tuple $t \in Hrel(a)$, de la partition correspondante de la table de hachage du noeud a.
8: $sol = sol \cup t$,
9: Marquer le noeud a,
10: **finsi**
11: **fin pour**
12: retourner sol.
13: **End.**

L'algorithme *PS-SEARCH* (*Algorithme 24*) est la version parallèle de l'algorithme *S-SEARCH*, le parallélisme se présente dans la boucle pour (ligne 5 à 11). En effet, il fonctionne de la même manière, sauf que chaque fois qu'il prend un tuple d'un noeud père, la recherche des tuples correspondants dans les noeuds fils se fait en parallèle, ainsi jusqu'à ce qu'il arrive au noeuds feuilles.

5.6 Complexité de P-HTR

Etant donné un hypertree binaire d'un réseau acyclique de contraintes avec une hypertree width h bornée. La complexité temporelle de l'algorithme *P-SBR* est de

$O(r * \mathcal{P}^{(h-1)})$ en utilisant $O(h)$ opérations, dans une *PRAM* de type *EREW*, où r est la taille d'une relation d'une contrainte et h l'hypertree width. La complexité temporelle de la résolution du *CSP* par l'algorithme *P-solving* est de l'ordre de $O(log\frac{n}{2})$ en utilisant $O(n)$ opérations, avec n et la taille de l'hypertree.

Par contre la complexité spatiale de *P-SBR* et de $O(hr + r^{h-1})$ au lieu de $O(hr + \sum_{i=1}^{h-1}(r^i))$ pour une parallélisation simple, avec r est la taille d'une relation d'une contrainte.

5.7 Expérimentations et analyses

Nous allons, à présent, mener une étude expérimentale des algorithmes parallèles présentés précédement, mais avant, nous décrivons l'environnement de simulation developpé ainsi que les protocoles expérimentals.

5.7.1 Environnement de simulation

Nous avons developpé un environnement de simulation pour l'algorithme P-SBR ainsi que pour P-solving, basé sur l'hypothèse du temps logique.

5.7.2 Modèle de simulation

Notre système est constitué de modules assez simples décrits sous la forme de programmes séquentiels classiques, qui sont assemblés et exécutés en parallèle par le simulateur. Un module de système est le traitement efféctué par un processeur qu'on appelle désormais processus.

Pour le fonctionnement interne d'un processus, nous avons découpé les tâches qui peuvent être efféctuées en trois principales opérations qui sont : la *lecture* d'un tuple, la *recherche* dans les tables de hachage y compris la jointure des tuples et l'*écriture* du tuple résultat. Ces opérations sont exécutées séquentiellement entre elles et parallèlement avec celles des autres processus.

Pour que chaque processus s'exécute comme si le parallélisme était réel, il faut que son environnement (ses variables d'entrée) ne change pas pendant l'exécution des autres processus.

Pour une exécution pipeline, si les processus ne sont pas adjacents il n'y a pas de dépendances et par conséquent la condition est vérifiée. Sinon, nous utilisons le mécanisme de sémaphore pour synchroniser les opérations.

5.7.3 Déroulement de simulation

Chaque opération (ou événement) s'exécute dans un intervalle de temps et le simulateur doit, donc, maintenir un compteur de temps logique.

Quand faire avancer ce compteur, quelle est la relation entre ce compteur et les temps d'exécution des événements ?

Nous faisons une simulation pas à pas de telle sorte que pour chaque pas (unité de temps logique) toutes les opérations qui peuvent être exécutées en même temps sont déclenchées ainsi :

1. Le simulateur exécute tous les processus et à chaque itération il incrémente par un le temps logique.
2. Pour tous les processus qui ne sans pas en conflit, il avance dans l'exécution par un pas du temps physique
3. Après un pas de temps logique, il modifie toutes les variables partagées manipulées.
4. Il recommence un nouveau pas en considérant les nouveaux états des variables.

Les opérations qui sont en conflit sont remplies dès la terminaison des actions attendues. Pour une exécution pipeline, dans la simulation chaque tuple est effectivement passé par les étages de la ligne. Ainsi, les conflits d'accés et les actions associées sont capturés.

5.7.4 Protocoles expérimentaux

Nous avons fait une simulation de l'algorithme *P-HTR* et les algorithmes qui le constituent pour determiner d'une part s'ils donnent une bonne complexité pratique temporelle et spaciale et d'autre part si l'approche de contraction utilisée est une alternative interessante a la contraction d'arbre connue dans la littérature, et afin de comparer nos résultats, nous avons simulé l'algorithme $PTAC$.

Les expérimentations dans cette section sont divisées en trois catégories :

1. Simulation de l'algorithme *P-SBR* pour estimer son interêt pratique en terme d'espace mémoire, vu que la résolution des sous problèmes est la source d'une complexité spatiale
2. Etude comparative entre $P - RAKE_1$ (l'opéraration RAKE avec pipeline) et $P - RAKE_2$ (l'opération RAKE avec partage de la relation père) afin de mesurer la contribution de l'équilibrage de charge dans l'opération RAKE.
3. Simulation de l'algorithme *P-solving* afin d'estimer son interet en terme de temps, et mesurer l'apport de la nouvelle technique de contraction d'arbre parallèle qui utilise l'opération $P-RAKE_2$ en menant une comparaison avec celle connue dans la littérature (l'algorithme $PTAC$ présentée dans 5.4).

Au niveau matériel, les simulations ont été réalisées sur un PC sous *Linux* version *6.0.52* équipé d'un CPU *Intel Pentium IV 2.4 GHz* et de *512 Mo* de mémoire vive.

les simulation sont réalisées sur une série de benchmarks *CSP* existe dans la littérature, et qui regroupent aussi bien des problèmes academiques que des problèmes réels, en utilisant des données collectées de l'exécution séquentielle. Ces données sont la moyenne des données obtenues après plusieurs exécutions.

Dans la suite, nous allons adopter la notation suivante :

$|V|$, $|E|$ et $|r|$ désignent, repectivement, le nombre de variables, le nombre de contraintes du *CSP*et le nombre maximum de tuples par relation. Nd, N_f et HTW sont, repectivement, le nombre de noeuds, le nombre de feuilles et la largeur de l'hypertree. NP désigne le nombre de processeurs.

Pour assurer une indépendance matérielle, les unités de mesure sont le *Tuple* pour l'espace et l'*Opération* (lecture, recherche ou écriture) pour le temps.

5.7.5 Simulation de l'algorithmes P-SBR

Dans cette simulation, nous considérons des sous problèmes de CSP avec différents nombre d'étage par ligne de pipeline ainsi que différentes tailles de relation de contraintes. Le tableau Tab. 5.1 présente l'espace mémoire requis pour une exécution avec pipeline (R_{P-SBR}) et sans pipeline (R_{join}) des opérations de jointure pour la résolution des sous problèmes.

Sous Problème					
n°	$\|NbE\|$	$\|R\|$	$\|V\|$	$\|R_{join}\|$	$\|R_{P-SBR}\|$
1	1	16037	7	0	0
2	2	26	6	226	1
3	3	7	8	224	2
4	3	2175	10	75	1
5	3	26	8	6526	2
6	3	49	8	23760	2
7	4	26	11	56952	3
8	5	20	8	84568	4
9	5	177	9	421362	3
10	7	12	9	68231	6
11	8	153	13	132241	6
12	9	10	10	264275	8

TAB. 5.1 – Simulation de l'algorithme *P-SBR* pour des sous problèmes des instances $\mathcal{C}SP$

Nous observons que les deux exécutions obtiennent des résultats comparables pour

un nombre d'étages inférieur à deux (une seule opération de jointure et par conséquent pas de résultats intermédiaires). Pour un nombre d'étages supérieur à deux, une exécution pipeline requiert moins d'espace mémoire qu'une autre sans pipeline à cause des résultats intermédiaires générés par cette dernière. Nous remarquons aussi que le gain en espace mémoire est proportionnel en nombre d'étages de la ligne du pipeline (cf. probleme 2, 5, 7) et en la taille des relations en entrée (cf. probleme 3, 5, 6). Nous remarquons aussi qu'il se peut que le gain est plus important pour l'instance dont le nombre d'etage est le plus petit pour une même taille de relation (cf. probleme 9 et 11, le gain dans 9 est plus important que dans le 11), la même chose pour la taille des relations (cf. probleme 4 et 5), ce qui depend de la consistance du problème.

N.B : pour une exécution sans pipeline, nous avons considéré le meilleur des cas, dans lequel une seule relation intermédiaire est générée à la fois.Aussi, pour une exécution pipeline la taille des tempons est limitée à un.

CSP								
Name	|V|	|E|	Nd	HTW	|r|	HTW>3	$\|R_{join}\|$	$\|R_{P-SBR}\|$
geom-30a-4	30	81	70	4	4	20	12168	36
haystacks-07	49	153	144	4	9	27	109368	48
Renault	101	134	81	2	48721	3	979	3
pret-60-25	60	40	28	5	4	19	3288	40
pret-150-60	150	100	50	5	4	50	6136	114
pret-150-75	150	100	54	5	4	54	6552	127
queen-5-5-3	25	160	7	10	2	7	1582	38
haystacks-06	36	95	88	3	8	18	7870	18
langford-2-4	8	32	31	4	8	5	3520	9
pigeons-7	7	21	19	3	6	4	6600	7
series-6	11	30	27	3	30	5	2920	5
queen-12	12	66	61	6	12	6	36180	24
mug-100-25-4	100	166	133	3	31	32	26762	32

TAB. 5.2 – Simulation de l'algorithme *P-SBR* pour des instances $\mathcal{CSP}$

Le tableau 5.2 présente la gain en complexité spatiale obtenu par le calcul pipeline de jointure pour différentes instances CSP (consistantes et inconsistantes). Nous notons que le gain en espace mémoire est très important dans les instances pour lesquelles le nombre d'exécutions pipeline (HTW>2) est élévé (haystacks-07 et langford-2-4). Ainsi, il depend du nombre de noeuds par hypertree (pret-60-25, pret-150-60 et pret-150-75). Il est claire aussi qu'il est proportionnel au nombre d'étage par ligne et au nombre de tuple par relation tel qu'il est montré plus haut.

Cependant, si nous considérons le pire des cas pour l'algorithme P-SBR qui est le

cas pour les instances dont l'HTW<2, l'espace mémoire requis est identique à celui d'une exécution sans pipeline.

Comme les deux facteurs qui influent sur la complexité des problèmes CSP sont la taille des relations des contraintes, et la largeur de la décomposition structurelle, et afin d'estimer la contribution de P-SBR, dans la figure Fig.5.3, nous évaluons, dans (a) et (b), le comportement de l'algorithme P-SBR selon, respectivement, la taille des contraintes et le nombre d'étage par ligne de pipeline , ceci pour deux classes de CSP, *aim-100-6* et *full-insertion*.

N.B : l'algorithme P-SBR forme une ligne du pipeline dans chaque noeud de l'hypertree, ainsi, le nombre d'etages dans la ligne est celui de contraintes dans le noeud moins un.

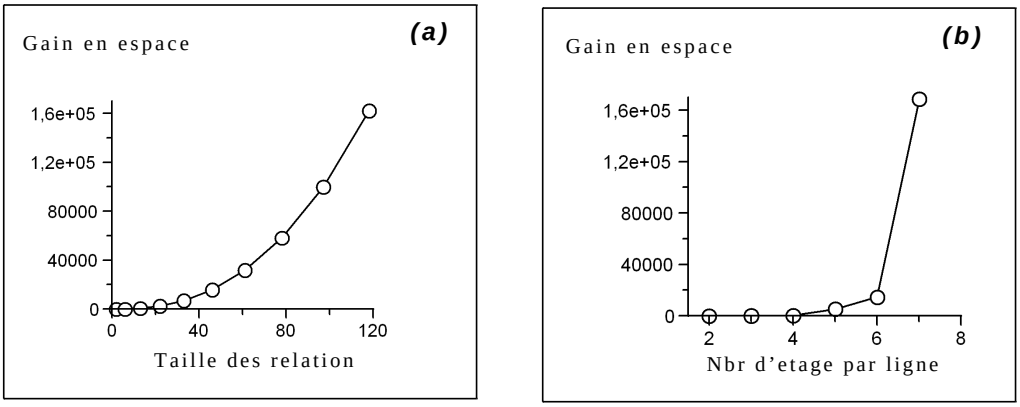

FIG. 5.3 – Les performances spaciales de *P-SBR* selon (a) la taille des relations (b) le nombre d'etage par ligne du pipeline

la figure Fig.5.3 montre que plus la taille des relations est importante plus l'espace économisé est considérable. Une interpretation similaire est illustrèe pour le nombre d'étages par ligne de pipeline.

5.7.6 Etude des deux techniques $P-RAKE_1$ et $P-RAKE_2$

$P-RAKE_1$ et $P-RAKE_2$ sont deux versions parallèles de l'opération connue RAKE, qui cherchent à optimiser le temps de résolution du problème global. Dans cette partie, nous présentons une étude comparative entre les deux algorithmes en calculant leurs efficacité et accélération. L'efficacité correspond donc au rapport $\frac{T_s}{P*T_p}$ avec P le nombre de processeurs et T_s (reps. T_p) le temps de résolution séquentielle (reps. parallèle) du sous arbre de l'hypertree.

Sous arbre								
n°	$\|NP\|$	$\|r\|$	T_{lent}	T_{seq}	$T_{pr}(1)$	$T_{pr}(2)$	Eff (1)	Eff (2)
1	2	196	557	1034	560	550	0.92	0.94
2	3	4500	75964	230720	75973	59974	0.76	0.96
3	4	27000	12661	31401	12667	10911	0.83	0.96
4	4	12000	33316	95095	33325	24875	0.71	0.95
5	5	18000	49974	156884	49979	33061	0.63	0.95
6	6	680	2039	9812	2045	1659	0.80	0.98
7	6	87808	251122	1055270	251131	176943	0.70	0.99
8	8	7776	20434	90125	20455	12252	0.55	0.92
9	9	18144	50085	297846	50107	34410	0.66	0.96
10	10	46656	1283734	820947	128405	84798	0.64	0.97

TAB. 5.3 – L'efficacité des deux techniques $\mathcal{P}$-$\mathcal{R}AKE$

T_{seq} et T_{lent} sont, respectivement le temps d'exécution séquentiel et celui du processeur le plus lent pour une exécution pipeline. $Tpr(1)$ (resp. $Tpr(2)$) est le temps d'exécution en utilisant l'opération $P-RAKE_1$ (resp. $P-RAKE_1$). Eff_1 et Eff_2 sont respectivement leurs éfficacité.

Le tableau Tab.5.3 présente les résultats obtenus pour certins sous arbres de différents instances de CSP. Nous constatons que le temps d'une exécution pipeline de la contraction d'arbre est identique à celui du processeur le plus lent ce qui donne une mauvaise efficacité. Tandis que le temps d'exécution avec partition de la relation père donne une bonne accélération jusqu'à dix processeurs, avec un temps équitable pour tous les processeurs.

Discussion

Il semblerait que la cause d'une faible efficacité pour $P-RAKE_1$ est le deséquilibre de charge entre les processeurs où le nombre de tuples à tester à chaque fois qu'on avance dans la ligne. Cet inconvenient est surmonté par $P-RAKE_2$ qui assure un équilibrage de charge (load balancing) par le partage des tuples à filtrer entre les processeurs.

Résumé

$P-RAKE_2$ peut etre considérée comme une parallélisation de l'opération RAKE qui assure à la fois une accélération linéaire et un equilibrage de charge entre les processeurs. En plus, la possibilité d'etre exécutée en mémoire partagée comme en mémoire distribuée, qui n'est pas le cas pour $P-RAKE_1$.

Dans l'algorithme P-CONTRACT, nous allons utiliser l'opération $P-RAKE_2$ au lieu de $P-RAKE_1$ a cause de son efficacité.

N.B :Le nombre de processeurs exploité pour la contraction d'un sous arbre est égale au nombre d'opération de semi jointure qui est égale au nombre d'étage de la ligne de pipeline pour P-RAKE.

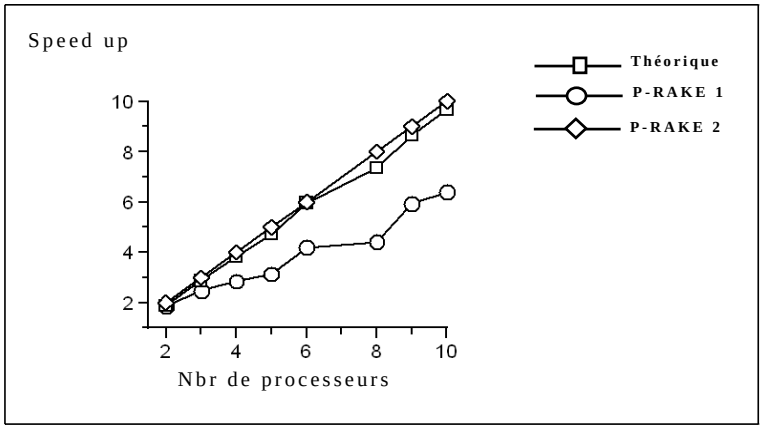

FIG. 5.4 – L'accélération de P-RAKE (2)

L'accélération (speed up) de l'algorithme $P-RAKE_1$ ainsi que celle de $P-RAKE_2$ est présentée dans la figure Fig.5.4 il est claire que l'accélération de $P-RAKE_2$ est proche de l'accélération théorique (linéaire).

5.7.7 Simulation de l'algorithme P-solving

Avant de comparer P-solving a A-HBR, il semble naturel de vouloir comparer l'algorithme P-CONTRACT a un algorithme appliquant la contraction d'arbre (comme CONTRACT présenté dans 5.4). Nous notons que P-CONTRACT augmente le degré du parallélisme. En plus la parallélisation de contraction entre les sous arbres independants (celle assurée par CONTRACT), il fait une parallélisation de contraction dans le meme sous arbre en utilisant l'operation $P-RAKE_2$.

N_c représente le nombre de contraction efféctués et T le temps d'exécution. L'unité de mesure est de 1000 opérations.

Le tableau Tab.5.4 fourni le temps d'exécution d'une part de l'algorithme A-HBR et d'autre part de l'algorithme PTAC (l'opération CONTRACT) et de celui de P-soving (l'opération P-CONTRACT) . Il résulte que P-CONTRACT génère un gain en complexité temporelle inversement proportionnel au nombre de contraction. Plus le nombre des noeuds feuilles contractés est grand plus le nombre de contraction est moins et plus le gain en temps est important (myc-3-4, Renault et hole-6), et inversement, ce qui est le cas pour hayst06 et mug25-4. Nous pouvons remarqué aussi que dans hole-6

CSP							*A-HBR*	PTAC		P-solving	
Name	|V|	|E|	Nd	*htw*	|r|	N_f	T	N_c	T	N_c	T
hayst06	36	95	88	3	8	23	16736	13	253	10	158
lang2-4	8	32	31	4	8	6	1562	11	1018	5	372
dom200	100	100	50	2	200	1	3632	48	3632	48	3632
hanoi-6	62	61	59	1	2148	2	260	37	169	37	169
mug25-4	100	166	133	3	31	26	119	12	47	9	38
myc-3-4	11	20	15	4	12	4	267	6	164	3	42
myc-4-4	23	71	48	6	6	12	124	8	315	4	299
quee-12	12	66	61	6	6	6	1330	15	740	5	85
Renault	101	134	81	2	48721	12	7758	26	38167	9	1456
series-6	11	30	27	3	30	5	622	8	256	4	155
hole-6	42	133	132	2	8	127	1970	85	1201	1	29

TAB. 5.4 – Simulation de l'algorithme *P-solving*

le gain est considérable, dans une exécution PTAC l'hypertree est contracté en 85 fois par contre en P-solving en une seule contraction.

Dans le pire des cas où il y a un seul noeud feuille par sous arbre, $P - RAKE_2$ se comporte de la même façon que RAKE et le nombre de contraction dans P-solving est identique à celui dans PTAC, par conséquent le temps d'exécution est le même (dom200 et hanoi-6).

Le degré du paralklélisme dépend de la structure de l'hypertree, il est proportionnel au nombre de noeuds feuilles.

5.8 Conclusion

Dans ce chapitre, nous avons proposé une nouvelle méthode parallèle PHTR pour la résolution des *CSP* après une décomposition structurelle. Cette méthode est basée à la fois sur le parallélisme dans le sous problème, lui meme, et entre les sous problèmes, en utilisant deux techniques de parallélisation, la contraction d'arbre et le pipeline.

Cette approche mixte nous permet de définir un algorithme dont nous espérons qu'il bénéficiera d'une part des garanties en terme de complexité théorique qu'offrent les méthodes de décomposition de *CSP*, et d'autre part des avantages des techniques de parlléisation pour leur efficacité pratique.

Conclusion générale et Perspectives

Plusieurs techniques de la résolutions des problèmes de satisfaction de contraintes ont été développé depuis 1970. Ces techniques peuvent être divisées grossièrement en deux catégories, ceux basés sur la recherche avec backtrack et ceux basés sur la propagation de contraintes. Pour les deux approches, la complexité des algorithmes est exponentielle en la taille du problème. Afin de réduire cette complexité, on a essayé d'extraire des classes de $\mathcal{CSP}$ dites tractable dont celle des $\mathcal{CSP}$ acyclique fait partie. Ainsi, La recherche est dirigée vers un autre axe qui consiste à décomposer la structure d'un $\mathcal{CSP}$ qui est un graphe ou plus généralement un hypergraphe en une structure arborescente de largeur bornée.

De nombreuse méthodes de décomposition structurelle ont été developpées, nous nous avons intéressé a la méthode de décomposition hypetree qui généralise toutes les autres méthodes.

Il a été prouvé qu'un $\mathcal{CSP}$ dont la structure est acyclique peut être résolu en un temps polynomial ce qui met en accent l'importance de l'étape de décomposition. Cependant ce résultat théorique est confronté à de nombreux problèmes lors de sa mise en oeuvre parmi lesquelles le coût du travail réalisé au niveau de chaque noeud de l'arbre qui est parfois prohibitif en terme de temps et d'espace.

Dans ce mémoire nous avons proposé une solution à ces problèmes. Nous avons présenté une nouvelle méthode de resolution des problèmes $\mathcal{CSP}$ après une décomposition hypertree aussi bien dans le cadre parallèle que dans le cadre séquentiel.

Dans le chapitre 3, nous avons proposé un algorithme $\mathcal{S}$-$\mathcal{HBR}$ pour une résolution séquentielle. Cet algorithme est constitué de deux parties, il fait la résolution des sous problèmes en utilsant l'algorithme $\mathcal{SP}$-$\mathcal{HBR}$, et puis la résolution globale du problème en utilisant l'algorithme $\mathcal{A}$-$\mathcal{HBR}$. Ces algorithmes sont basés sur la notion de hachage pour assurer un accée directe aux donnée et ainsi optimiser la complexité temporelle.

Dans le chapitre 5, nous avons proposé un algorithme parallèle $\mathcal{P}$-$\mathcal{HTR}$ qui fait la résolution des sous problème en utilisant la technique du pipeline, pour objectif d'optimiser la complexité spatiale vu que cette partie du problème est la source de cette complexité. Pour la résolution globale du problème, nous avons definir une nouvelle

alternative pour la contraction d'arbre parallèle et nous l'avons mise en oeuvre pour la résolution globale dans un algorithme $\mathcal{P}$-$\mathcal{S}$olving.

Un point intéressant pour les futurs travaux de recherche est l'expérimentation de ces algorithmes sur des machines parallèles, afin de confirmer leur efficacité dans un environnement du parallélisme réel.

Un autre point est l'optimisation de ces algorithmes par combinaison d'autres heuristiques de résolution et par augmentation du degré du parallélisme, ce qui nous amène aux perspectives suivantes :

- Effectuer un load balancing pour l'algorithme $\mathcal{P}$-$\mathcal{S}BR$ afin d'avoir une complexité en terme de temps aussi.
- Utiliser des heuristiques sur l'ordre des opération de jointures et de semi jointures ou même entre les noeuds de l'arbre.
- Réaliser une parallélisation entre les sous arbres par declenchement des opérations de filtrage dès le produit des premiers tuples des noeuds fils.
- Faire un parallélisme intra opérateur pour les opération de jointure et de semi jointure par l'attributoin de plusieurs processeurs à la même opération (avec un load balancing).

Une autre alternative autre que celle que nous avons exploré, et qui peut être efficace, est l'utilisation des algorithmes énumératifs pour la résolution des sous problèmes et même de la résolution globale, afin de bénificier d'une part des avantages des technique d'énumération, et d'autre part, des garanties en terme de complexité qu'offrent les méthodes de décomposition structurelle.

Bibliographie

[1] J. Flokstra A. N. Wilschut and P. Apers, *Parallel evaluation of multijoin queries*, Proceedings of the ACM-SIGMOD, 24(2) :115-126 (1995).

[2] R. J. Bayardo and D. P. Miranker, *A complexity analysis of space-bounded learning algorithms for the constraints satisfaction problem*, Proceedings of 13th National Conference on Artificial Intelligence, pages 298-304, (1996).

[3] C. Bessier and J. Regin, *Refining the basic constraint propagation algorithm*, Proceeding of IJCAI'01 (2001).

[4] C. Bessière, *Arc-consistency and arc-consistency again*, Artificial Intelligence 65 :179-190 (1994).

[5] M. Bruynooghe, *Graph coloring and constraint satisfaction*, Technical Report CW 44, Katholieke Universiteit Leuven (1985).

[6] E.C. Freuder C. Bessiere, P. Meseguer and J. Larrosa, *On forward checking for non binary constraints satisfaction problems*, Artificial Intelligence (2002).

[7] M. C. Cooper, *An optimal k-consistency algorithm*, Artif. Intell., 41(1) :89-95 (1989).

[8] M. Gyssens D. Cohen and P. Jeavson, *Decomposing constraint satisfaction problems using database techniques*, Artificial Intelligence volume 66 (1994).

[9] R. Dechter, *A constraint-network approach to truth-maintenance*, Technical Report 870009 (R-80), UCLA Cognitive Systems Laboratory (1987).

[10] ———, *Constraint processing*, Morgan Kaufmann Publisher (2003).

[11] ———, *Tractable structures for constraint satisfaction problems*, Chapitre dans "Handbook of constraint programming", Elsevier (2006).

[12] R. Dechter and D. Frost, *Backjump-based backtracking for constraint satisfaction problems*, Artificial Intelligence 136 :147-188 (2002).

[13] R. Dechter and I. Meiri, *Experimental evaluation of preprocessing algorithms for constraint satisfaction problems*, Proceedings of the tenth International Joint Conference on Artificial Intelligence (IJCAI-89), pages 271-277, Detroit, Michigan, (1989).

[14] E.C Freuder, *A sufficient condition for backtrack free search*, Journal of the ACM Vol29 N°1 (1982).

[15] D. Frost and R. Dechter, *Look-ahead value ordering for constraint satisfaction problems*, Proceedings of IJCAI 95 (1995).

[16] M. Grohe G. Gottlob and N. Musliu, *Hypertree decomposition : structure, algorithme and applications*, 31st International Workshop, WG 2005, Metz, France (2005).

[17] N. Leone G. Gottlob and F. Scarcello, *A comparison of structural csp decomposition methods*, Artif. Intell., 124(2), 243-282 (2000).

[18] ———, *Hypertree decompositions and tractable queries*, J. Comput. Syst. Sci., 64(3), 579-627 (2002).

[19] J. Gaschnig, *Performance measurement and analysis of certain search algorithms*, Technical Report CMU-CS-79-124, Carnegie-Mellon University, (1979).

[20] M. Ginsberg, *Dynamic backtracking*, Journal of Artificial Intelligence Research,1 :25-46 (1993).

[21] S. Golumb and L. Baumert, *Backtrack programming*, Journal of the ACM, pages 516-524 (1965).

[22] J. Gu and R. Sosic, *A parallel architecture for constraint satisfaction*, International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, June 1991, Kauai, Hawaii, pp. 229-237 (1991).

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

[24] T. Hogg and C.P. Williams, *Expected gains from parallelizing constraint solving for hard problems*, Proceedings of AAAI 94, pages 331-336, Seattle, WA, (1994).

[25] J. Jaja, *An introduction to parallele algorithms*, Addison-Wesley Publishing Company, 1992.

[26] P. Jégou and C. Terrioux, *Hybrid backtracking bounded by tree-decomposition of constraint ntworks*, Artificial Intelligence, 146 :43(75) (2003).

[27] Ph. Jégou, *Csp decomposition methods. parallel implementation, a future prospect ?*, JIM'99 - Journées de l'Informatique Messine, NP-Complétude et Parallélisme, Metz, France. (1999).

[28] P. G. Kolaitis and M. Y. Vardi, *Conjunctive query containment and constraintsatisfaction*, PODS, pages 2054-213 (1998).

[29] N. Sadeh M. S. Fox and C. Baykan, *Constrained heuristic search*, In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, pages 309-315, Detroit, MI, (1989).

[30] K. Mackworth, *Consistency in networks of relations*, Artificial intelligence 8 :99-118 (1977).

[31] J. J. McGregore, *Relational consistency algorithms and their application in finding subgraph and graph isomorphisms*, Information Sciences, 19 :229-250, (1979).

[32] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, 26 th Symposium on Foundations of Computer Science, pp.478-489, Portland, Oregon (1985).

[33] R. Mohr and T. C. Henderson, *Arc and path consistency revisited*, Artificial Intelligence 28 :225-233 (1986).

[34] T. Nguyen and Y. Deville, *A distributed arc-consistency algorithm*, Science Computer Program 30 : 227-250 (1998).

[35] O. C. of the Third International Competition of CSP Solvers, *Xml representation of constraint networks format xcsp 2.1*, Disponible sur http ://www.cril.univ-artois.fr/CPAI08/XCSP2-1.pdf, dernière mise à jour 15 janvier 2008.

[36] Y. Deville P. V. Hentenryck and C. M. Teng, *A generic arc-consistency algorithm and its specializations*, Artificial Intelligence 57 :291-321 (1992).

[37] ———, *Neighborhood-based variable ordering heuristics for the constraint satisfaction problem*, Proceedings CP, 565-569, Paphos, Cyprus (1995).

[38] I. Meiri R. Dechter and J. Pearl., *Temporal constraint networks*, Artif. Intell., 49(1-3) :61-95 (1991).

[39] N. Robertson and P. D. Seymour, *Graph minors, algorithmic aspects of tree width*, J. Algorithmis, 7(3), 309-322 (1986).

[40] D. Sabin and E. Freuder, *Contradicting conventional wisdom in constraint satisfaction*, Proceedings of eleventh ECAI, pages 125-129, (1994).

[41] T. Schiex and G. Verfaillie, *Nogood recording for static and dynamic constraint satisfaction problems*, Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence (1993).

[42] et G. Verfaillie T. Schiex, H. Fargier, *Valued constraint satisfaction problems : hard and easy problems*, Proceedings of IJCAI-95, pages 631-637 (1995).

[43] R. Tarjan and M. Yannakakis, *Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs*, SIAM Journal Vol 13 N° 3 (1984).

[44] C. Terrioux, *Approche structurelles et coopératives pour la résolution des problèmes de satisfaction de contraintes.*, Ecole Doctorale de mathématiques et d'informatique de Marseille. Université d'Aix-Marseille I :134, 2002.

[45] U.Montanari, *Networks of constraintes : Fondamental properties and application to picture*, Proc. of Information Sciences (1974).

[46] C. Bessière Y. Hamadi and J. Quinqueton, *Distributed intelligent backtracking*, Proceedings ECAI, 219-223, Brighton, UK (1998).

[47] M. Krajecki Z. Habbas and D. Singer, *Domain decomposition for parallel resolution of constraint satisfaction problem with openmp,*, Proceedings European Workshop on OpenMP, 8, Edinburgh, UK. (2002).

[48] Y. Zhang and A. K. Mackworh, *Parallel and distributed finite constraint satisfaction : Complexity, algorithms and experiments*, Parallel Processing for Artificial Intelligence, Elsevier Pub (1993).

[49] Y. Zheng and B. Choueiry, *New structural decomposition techniques for constraint satisfaction problems*, ASpringer Verlag. Berlin Heidelberg (2005).

Decomposition arborescente pour resoudre les problèmes de satisfaction de contraintes avec mise en oeuvre du parllèlisme

Résumé : Le formalisme $\mathcal{CSP}$ ($\mathcal{C}$onstraint $\mathcal{S}$atisfaction $\mathcal{P}$roblems) constitue un cadre puissant et général pour représenter et résoudre un grand nombre de problèmes. Les $\mathcal{CSP}$ sont généralement $\mathcal{NP}$-$\mathcal{C}$omplet. De nombreux travaux ont été menés pour réduire les bornes de complexité de ces problèmes. Parmi ces méthodes nous nous intéressons aux méthodes dites de décomposition structurelle, et exactement à la décomposition hypertree dont il est prouvé qu'elle les généralise toutes. Cette méthode consiste à décomposer la structure d'un $\mathcal{CSP}$ qui est un hypergraphe en une structure arborescente de largeur bornée. Un $\mathcal{CSP}$ dont la structure est un arbre peut etre résolu en un temps polynomial. Cependant, le cout du travail réalisé au niveau de chaque noeud de l'arbre est parfois prohibtif en terme de temps et d'espace mémoire. Dans ce travail nous proposons un nouveau algorithme efficace de résolution des $\mathcal{CSP}$ après une décomposition structurelle.

Dans un premier temps, nous présentons un algorithme de résolution séquentiel $\mathcal{S}$-$\mathcal{HBR}$ qui tire profit de la technique de hachage. Dans un deuxième temps, nous proposons un algorithme de résolution paralléle $\mathcal{P}$-$\mathcal{HTR}$ qui repose sur une technique du pipeline pour la résolution des sous problèmes en vu d'optimiser la complexité spatiale, et pour la résolution globale du problème, nous proposons une nouvelle technique de contraction d'arbre parallèle.

Mots clés : Problèmes de satisfaction de contraintes ($\mathcal{CSP}$), décomposition arborescente, résolution $\mathcal{CSP}$, parallélisme.

Hypertree decomposition for solving constraint satisfaction problems with parallelism implementation

Abstract : The $\mathcal{CSP}$ formalism ($\mathcal{C}$onstraint $\mathcal{S}$atisfaction $\mathcal{P}$roblems) forms a powerful and general setting to represent and solve lot of problems. $\mathcal{CSP}$s are generally $\mathcal{NP}$-$\mathcal{C}$omplete. Many works have been led to reduce the complexity boundaries of these problems. Among these methods we are interested in the methods say structural decomposition , and precisely to the hypertree decomposition for which it is proven that it generalizes them all. This method consists to decompose the structure of a $\mathcal{CSP}$ that is a hypergraphe on a hypertree structure of limited width. A $\mathcal{CSP}$ of which the structure is a tree can be solved in a polynomial time. However, the cost of work achieved in each node of the tree is sometimes prohibtive in time and memory space. In this work we propose a new efficient algorithm for solving $\mathcal{CSP}$s problems after a structural decomposition.

In a first time, we present an algorithm $\mathcal{S}$-$\mathcal{HBR}$ for sequential resolution which benefits of the hash technique. In a second time, we propose an algorithm $\mathcal{P}$-$\mathcal{HTR}$ for parallel resolution which is based on a pipeline technique for the resolution of the sub problems in order to optimize the spatial complexity, and for the global resolution of the problem, we propose a new parallel tree contraction technique.

Keywords : Constraint satisfaction problems ($\mathcal{CSP}$), hypertree decomposition, solving $\mathcal{CSP}$, parallelism.