République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A/Mira de Béjaia
Faculté des Sciences Exactes
Département d'Informatique

MÉMOIRE DE MASTER PROFESSIONEL

En
Informatique

Option
Administration et Sécurité Réseaux

Thème

Résolution des CSPs par décomposition

Présenté par : Mlle. Benouaret Amel & Mlle. Chelalou Souad

Soutenu le 04 Juillet 2016 devant le jury composé de :

Président	**Dr A. Touazi**	**Maître de conf. B**	**U. A/Mira Béjaia.**
Rapporteur	**Dr K. Amroun**	**Maître de conf. A**	**U. A/Mira Béjaia.**
Examinateur	**Dr Z. Farah**	**Maître de conf. B**	**U. A/Mira Béjaia.**
Examinatrice	**Mlle S. Benmarbi**	**Doctorante "LMD"**	**U. A/Mira Béjaia.**

Béjaia, Juillet 2016.

⁕ *Remerciements* ⁕

Nous remercions DIEU tout puissant de nous avoir donné la force, la santé, le courage et la patience de pouvoir accomplir ce travail.

Un grand merci à toutes nos familles surtout nos parents pour leur encouragement et leur suivi avec patience du déroulement de notre projet.

Nos plus vifs remerciements vont d'abord à l'encadreur Mr K.Amroun, de nous avoir fait l'honneur et le plaisir de diriger ce travail. Il a su nous guider avec patience, compréhension et rigueur. On lui en sera toujours reconnaissante.

Nos sincères remerciements s'adressent aussi à tous ceux qui ont contribué de prés ou de loin à la réalisation de ce travail.

Enfin, nous tenons aussi à remercier également tous les membres de jury pour avoir accepter d'évaluer notre travail : Dr A. Touazi qui nous ont fait honneur par sa présence en qualité de président de jury, Dr Z. Farah et Mlle S. Benmarbi d'avoir accepter d'examiner ce travail et de consacrer leurs temps pour l'évaluer..

*M. **Merci à tous***

⁕ *Dédicaces* ⁕

Je dédie ce modeste travail aux personnes les plus chères à mes yeux
mes parents.
A ces deux grands cœurs qui m'entourent toujours par leur tendresse.
A ceux qui m'ont toujours encouragée et soutenue dans mes études et
m'ont éclairée et ouvert la vie de l'avenir.
Je vous dédie le fruit de mes efforts, comme un symbole de gratitude.
Que Dieu vous me garde, et que vous soyez toujours prés de moi.
A tous ceux qui me sont chers, qui m'ont aidée par leur soutien moral,
en particulier :
A ma très chère sœur Yasmin et mes frères Yacine, Okba, Mouhand,
qui occupent une place particulière dans mon cœur.
Tous les membres de ma famille,
A tous mes amis, pour leur amitié, leur soutien moral, et leurs conseils
Toute personne qui nous a aidées et encouragées à faire notre projet.

Amel

⁕ *Dédicaces* ⁕

Je dédie ce travail à :

Mes chers parents, que nulle dédicace ne peut exprimer Mes sincères sentiments, pour leur patience illimitée, leur encouragement contenu, leur aide, en témoignage de nous profond amour et respect pour ses grands sacrifices. Que dieu leur réserve la bonne santé et une langue vie.

Ma chère sœur Lynda et mon petit frère Massi à qui je leur souhaite une vie pleine de bonheur, de santé et de réussite.

nos chers amis, qui sans leur encouragement ce travail n'aura jamais vu le jour.

Que dieu nous garde toujours unis.

Toute personne qui nous a aidées et encouragées à faire notre projet.

Tout nos enseignants

Je n'oublierai pas mes amis de la promotion ASR.
Ceux qui cherchent leurs noms ici.

Souad

Table des matières

Table des figures

Liste des algorithmes

Liste des abréviations

AC Arc Consistency.
BE Bucket Elimination.
BJ Backjumping.
BICOMP Biconnected Component.
BM Backmarking.
BT Backtracking.
BTD Backtracking on Tree-Decomposition.
CBJ Backmarking and Backjumping.
SCDnew Spread CutNEW.
CCM CycleCluster Methode.
CHD Component Hypertree.
CSP Constraint Satisfaction Problems.
DAC Directional arc consistency.
DBE Dual Bucket Elimination.
ECHD Extend Component Hypertree.
FC Forward Checking.
FHD Fractional Hypertree Décomposition.
GHD Generalized Hypertree Decomposition.
HD Hypertree Décomposition.
Hinge Hinge Decomposition.
MAC Maintaining Arc-Consistency.
MIMD Multiple Instruction, Multiple Data.
MISD Multiple Instruction, Single Data.

MPI Message Passing Interface.
MRV Minimum Remaining Value.
NUMA Non Uniform Memory Access.
PVM Parallel Virtual Machine.
RMA Remote Memory Access.
TD Tree Decomposition.
SCD Spread Cuts.
SIMD Single Instruction, Multiple Data.
SISD Single Instruction, Single Data.
TCLUSTER Tree Clustering.
UMA Uniform Memory Access.

Introduction générale

L'intelligence Artificielle est un domaine où l'on propose un ensemble de cadres de représentation de la connaissance et de méthodes de résolution. Parmi ses formalismes, nous pouvons mentionner, par exemple, la représentation et le traitement des connaissances, la preuve de théorème, la planification de tâches, la perception de l'environnement,...
Pour notre part, Parmi les formalismes existants, nous nous intéressons aux Problèmes de Satisfaction de Contraintes (CSP) introduit par Montanari, qui offrent un cadre simple et formel pour représenter des problèmes de décision sur des domaines discrets. Ce formalisme couvre un grand nombre de problèmes pratiques (planification, ordonnancement, coloration de graphes, conception ...)
Le point commun entre ces différents problèmes réside dans la possibilité d'exprimer sous forme de contraintes les propriétés et les relations qui existent entre les objets manipulés.
Le pouvoir d'expression du formalisme CSP repose donc essentiellement sur les contraintes.
L'expression des contraintes est très variée elles peuvent être une équation, une inéquation, un prédicat, une fonction booléenne, une énumération des combinaisons de valeurs autorisées, etc.)
La principale difficulté réside dans le fait que résoudre un CSP constitue un problème NP-complet.

La technique la plus simple pour déterminer s'il existe ou non une solution

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

Dans notre projet, nous proposons d'éxploiter les techniques de décomposition structurelles pour la résolution des CSPs.
Le reste du mémoire est structuré comme suit :
Le premier chapitre présente le formalisme CSP. Nous présentons d'abord les différents concepts de ce formalisme. Ensuite, nous évoquons les principales approches de résolution.
Le second chapitre décrira les principales méthodes de décomposition structurelles des CSP .
Le troisième chapitre est consacré à la résolution en exploitant les décompositions structurelles .
Le quatrième chapitre nous présentons notre contribution relative à la résolution de CSPs basée sur la GHD.
Le cinquiéme chapitre nous présentant notre résolution paralléle des probléme de satisfaction de contrainte via une GHD.

Chapitre 1

les problèmes de satisfaction de contraintes

1.1 Introduction

Les problèmes de Satisfaction de contrainte (CSP pour Constraint Satisfaction Problems) ont été introduits dans le milieu des années 70. Ils rassemblent un ensemble de problèmes de décision venant de l'intelligence Artificielle (comme la satisfiabilité d'un ensemble de clauses de la logique propositionnelle) ainsi que des problèmes venant de la Recherche Opérationnelle (comme les problèmes d'ordonnancement).
Dans ce chapitre Nous rappelons d'abord le formalisme introduit par Montanari. Ensuite, nous présentons les principales méthodes employées pour résoudre ce problème.

1.2 formalisme CSP

Dans ce chapitre, nous présentons les problèmes de satisfaction de contraintes :

1.2.1 Le problème de satisfaction de contraintes

Un problème de satisfaction de contraintes (CSP) est défini par un triplet (X, D, C) où $X = \{x_1, \ldots, x_n\}$ est l'ensemble fini des n variables du problème, $D = \{D_{x_1}, \ldots, D_{x_n}\}$ est l'ensemble des n domaines finis pour les variables. D_{x_i} est l'ensemble des valeurs possibles pour la variable x_i, $C = \{c_1, \ldots, c_m\}$ est l'ensemble des m contraintes. Les contraintes peuvent être exprimées sous différentes formes de table de valeurs compatibles, formules mathématiques, etc. Ce sont des relations entre des variables qui définissent la structure du problème à résoudre.
Une solution du CSP est une affectation de chaque variable dans X à une valeur dans son domaine associé de telle sorte qu'aucune contrainte $c \in C$ soit violée [1].

1.2.2 Définition de base

Définition 1.2.1. (Contrainte)
Soit un $CSP(X, D, C)$, une contrainte $c \in C$ sur les variables $x_{i_1}, ..., x_{i_k}$ pour $i_1, \ldots, i_k \in \{1, \ldots, n\}$ est une relation dans $D_{x_{i_1}} \times \ldots \times D_{x_{i_k}}$, domaines des variables $x_{i_1}, ..., x_{i_k}$. On a donc :

$$C \subseteq \cap i_k j = i_1 D_{x_j}$$

On note $var(c)$, l'ensemble des variables intervenant dans la contrainte c [1].

Définition 1.2.2. (Arité)
L'arité d'une contrainte $c \in C$ est le nombre de variables sur lesquelles elle porte (i.e. le cardinal de $var(c)$). On dira que la contrainte est :

- unaire si son arité est égale à 1.

- binaire si son arité est égale à 2.

- n-aire si son arité est égale à n.

La contrainte d'arité maximale d'un réseau de contraintes permet de définir sa nature. Si toutes les contraintes d'un réseau de contraintes P sont binaires, P est lui-même dit binaire. Réciproquement, lorsqu'au moins une contrainte est d'arité supérieure à deux, le réseau de contraintes est dit non-binaire (n-aire). Un réseau de contraintes non-binaire peut toujours être transformé en un réseau de contraintes binaires en appliquant une technique de binarisation [2].

Définition 1.2.3. (affectation)
On appelle affectation un ensemble de couples (x_i, v_i), tels que x_i est une variable du CSP et v_i est une valeur appartenant au domaine $D(x_i)$ de cette variable.

- Une affectation est dite totale si elle instancie toutes les variables du problème ; elle est dite partielle si elle n'en instancie qu'une partie.

- Une affectation A viole une contrainte c_j si toutes les variables de c_j sont instanciées dans A, et si la relation définie par c_j n'est pas vérifiée pour les valeurs des variables de c_j définies dans A.

- Une affectation (totale ou partielle) est consistante si elle ne viole aucune contrainte, et inconsistante si elle viole une ou plusieurs contraintes.

- Une solution est une affectation totale consistante, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte [3].

Définition 1.2.4. (Solution)
Une solution d'un CSP $P = (X, D, C)$ est une affectation $s \in S$ qui satisfait toutes les contraintes. On note $Sol(P)$ l'ensemble des solutions. $Sol(P) = s \in S | \forall c \in C, s \in c$

L'ensemble des solutions correspond à tout élément de l'espace de recherche (affectations) appartenant aussi aux valeurs permises pour chaque contrainte [1].

Définition 1.2.5. (CSP consistant)
Un CSP P est dit consistant s'il admet au moins une solution. Il est qualifié d'inconsistant sinon. La résolution d'un CSP dépend du résultat attendu. Résoudre un CSP peut consister à :

- rechercher toutes les solutions : il s'agit de déterminer toutes les solutions (si le CSP est consistant).
- rechercher une solution : l'algorithme de résolution s'achève dès qu'une solution est trouvée.
- vérifier la consistance d'une instanciation complète : il s'agit de la vérification de la consistance de l'instanciation donnée [4].

Définition 1.2.6. (instanciation)
Soit $x \in X$ une variable et $v \in d(x)$.on dit que x est instanciée avec la valeur v lorsque v a été affectée à x (on dit également que $x = v$). Soit un ensemble de variables $Y \subseteq X$ tel que $\forall y_i \in Y$, y_i est instanciation partielle de X. I est dite complète si elle porte sur toutes les variable de X [4].

Définition 1.2.7. (instanciation consistante)
Etant donné un problème $p = (X, D, C)$ une instanciation I des variable de Xest dite consistante si et seulement si elle ne viole aucune contrainte de C. Elle est dite inconsistante sinon [4].

Définition 1.2.8. (CSP binaire)
Un CSP $P = (X, D, C)$ dont toutes les contraintes sont d'arité deux est appelé un

CSP binaire.
On représente le CSP par un graphe de contraintes $G = (X, C)$ [7].

1.3 Représentation du graphe associé à un CSP

En générale une instance CSP est représentée par un graphe non orienté ou un hypergraphe. Les nœuds du graphe correspondent aux variables du problème et les contraintes correspondent aux arêtes reliant ces nœuds. La représentation des contraintes binaires est une arête reliant les 2 variable de la contrainte .pour les contraintes d'arité supérieure à 2, la représentation la plus souvent utilisé est les une sorte d'ellipse (appelée hypernoeud) entourant toutes les variables, mais aussi une hyperarête. Voir la figure 1.
Dans le cas des CSP binaire de nouveaux paramètres prennent en compte le graphe de contrainte. Notons d'abord que si toutes les variables du problème sont reliées par une contrainte à toutes les autres variables, le graphe de contraintes est dit complet. Si n est le nombre de variable, alors le nombre de contrainte est égal à n*(n-1)/2 pour un graphe complet [6].

Exemple 1.3.1. Afin de bien mettre en évidence toutes les notions abordées jusqu'à présent, nous allons utiliser un exemple simple de coloration de carte .le problème est de colorier tous les pays d'une carte de telle sorte que deux pays frontaliers ne soit pas peints avec la même couleur. A l'origine le but de ce problème était de minimiser le nombre de couleurs utilisées.
Cet exemple comme tous les problèmes de coloriage de cartes ou de graphes, se formalise naturellement sous forme de CSP, et contient :

L'objectif consiste à colorier une carte, de sorte que deux régions ayant des frontieres en commun soient coloriées avec des couleurs différentes.
Ce problémée peut-étre modélisé par le CSP suivant :

- L'ensemble des variables $X = \{x_1, x_2, \ldots, x_6\}$ est l'ensemble des régions à colorer ;

- $D = \{d_1, d_2, \ldots, d_6\}$ est l'ensemble des domaines de chaque variables. Quel que soit i, d_i est l'ensemble des couleurs pouvant étre attribuées à la variable x_i. $d_i = \{rouge, jaune, bleu\}$;

- $C = \{c_1, c_2, \ldots, c_{10}\}$ est l'ensemble fini des contraintes.Chaque contrainte c_i est définie par un couple (x_{i1}, x_{i2}) de variables de X reprsentant des régions voisines sur la carte.$c_1 = (x_1, x_2), c_2 = (x_1, x_3), \ldots, c_{10} = (x_5, x_6)$;

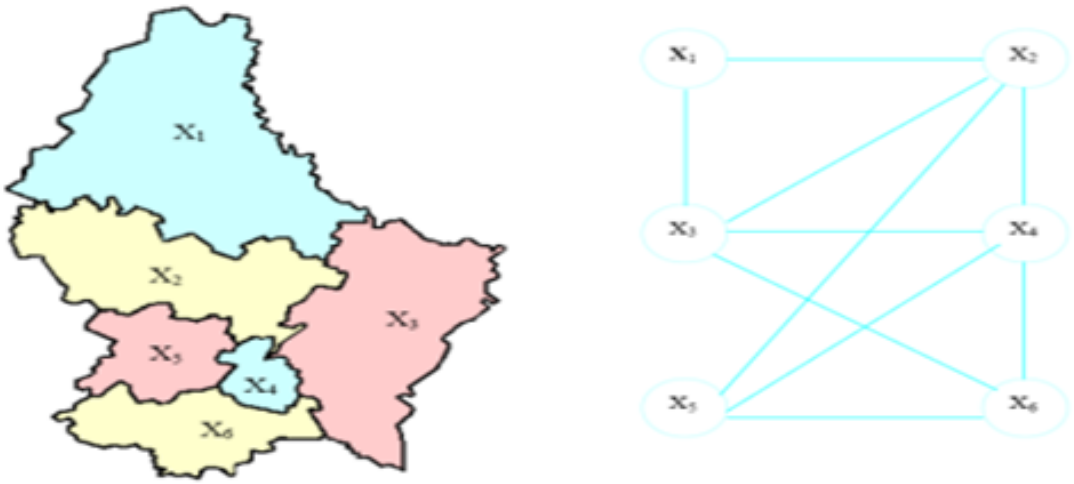

FIGURE 1.1 – Probléme de coloriage de carte et son graphe des cntraintes associé

Cette représentation ne fait pas apparaitre le détail des domaines et des valeurs respectant chaque contrainte. Mais elle donne une idée gnérale de la structure du probléme.

1.4 Les méthodes de résolution sequentielles

Nous présenterons les différentes méthodes pour la résolution des CSP par des algorithmes complets :

1.4.1 Generate-and-test

Une méthode simple pour la résolution des CSP 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. Le nombre de possibilités testées est alors le cardinal du produit cartésien des domaines des variables, ce qui pour les problèmes de grandes tailles devient impossible à envisager [1].

1.4.2 Le backtrack

L'algorithme backtrack correspond à une recherche aveugle de la solution en essayant successivement les ensembles des affectations de variables jusqu'à trouver une solution. Cet algorithme est fondé sur les points de retour. Chaque fois qu'un choix est effectué, un point de retour est mentionné dans l'algorithme. L'ordre de variable à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations. Le chemin en cours de test est éliminé de l'ensemble de solutions si un domaine donné devient vide. La procédure choisir revient à l'itération précédente et essaie une nouvelle valeur pour la variable en question, si aucune valeur n'est possible on revient au point précédent. Si par contre, nous sommes au premier point alors on sort de l'algorithme en échec (autrement dit avec impossibilité de trouver une solution) [5].

Amélioration de backtrack

- Réduire de l'espace de recherche (filtrage ou consistance d'arc (AC)).
- Amélioration de l'algorithme par des technique rétrospectives backtrack intelligent (Backjumping et backmarking) (BJ ,CBJ,BM)
- Amélioration de l'algorithme par des technique prospectives : (forward checking et maintaining Arc Consistency) (FC, MAC).
- Heuristique d'ordre d'instanciations dynamique des variables Minimum Remainig Value(MRV).
- Décomposition : remplace si possible un CSP par plusieurs sous CSP plus simple.

1.4.3 Filtrage au cours de la résolution

Dans les techniques de résolution présentées précédemment, les contraintes sont utilisées de manière passive. Elles sont exploitées uniquement pour tester la consistance des affectations partielles et complètes. Les techniques de filtrage utilisent les contraintes de manière active pour effectuer des déductions sur le problème. L'objectif principal des techniques de filtrage est la détection précoce d'instanciation partielle localement ou totalement incohérente. Une des techniques les plus utilisées est la technique de renforcement de la consistance locale ou consistance d'arc.
Le premier algorithme qui utilise ce concept de filtrage est nommé Forward checking (FC). Lors de chaque instanciation d'une variable courante, FC supprime les valeurs des variables non encore instanciées inconsistantes avec l'instanciation liées à la variable courante. D'autres algorithmes propagent en plus les suppressions. Par exemple, l'algorithme Maintaining Arc-Consistency (noté MAC) applique un filtrage par arc-consistance après chaque instanciation [4].

1.4.3.1 Notion de consistance

Une consistance est une propriété qui doit être assurée à chaque étape de la recherche de la solution dans l'objectif de couper l'arbre de recherche. Cette contrainte ne doit pas être coûteuse à calculer. Elle retire des domaines des variables, les valeurs qui ne participent pas à aucune solution [5].

Nous définissons ici deux types de consistance :

1.4.3.2 Consistance de nœud

Consiste à éliminer des domaines de variables toutes les valeurs qui n'appartiennent pas aux solutions des contrainte unaires.

Définition 1.4.1. (Consistance de noeud)
On dit qu'un CSP est consistant de nœud si pour chaque variable $x \in X$, toute contrainte unaire partant sur x, coïncide avec le domaine de x. Soit un $CSP(X, D, C)$ et $c \in C : var(c) = x$ et $\forall d \in Dx$, $d \in c$ [1].

1.4.3.3 Consistance d'arc

De manière informelle, une contrainte binaire est consistante d'arc si chaque valeur de chaque domaine appartient à au moins une paire de valeurs consistantes définie par la contrainte. On parle alors de CSP arc consistant si toutes ses contraintes binaires sont arc consistantes [1].

Définition 1.4.2. (Consistance d'arc) Soit un $CSP(X, D, C)$, soit une contrainte binaire $c \in C$ portant sur les variables x et y avec leur domaine respectif D_x et D_y, telle que $c \subseteq D_x \times D_y$. Nous dirons que c est arc consistante si :
$\forall a \in D_x$, $\exists b \in D_y, (a, b) \in c$;

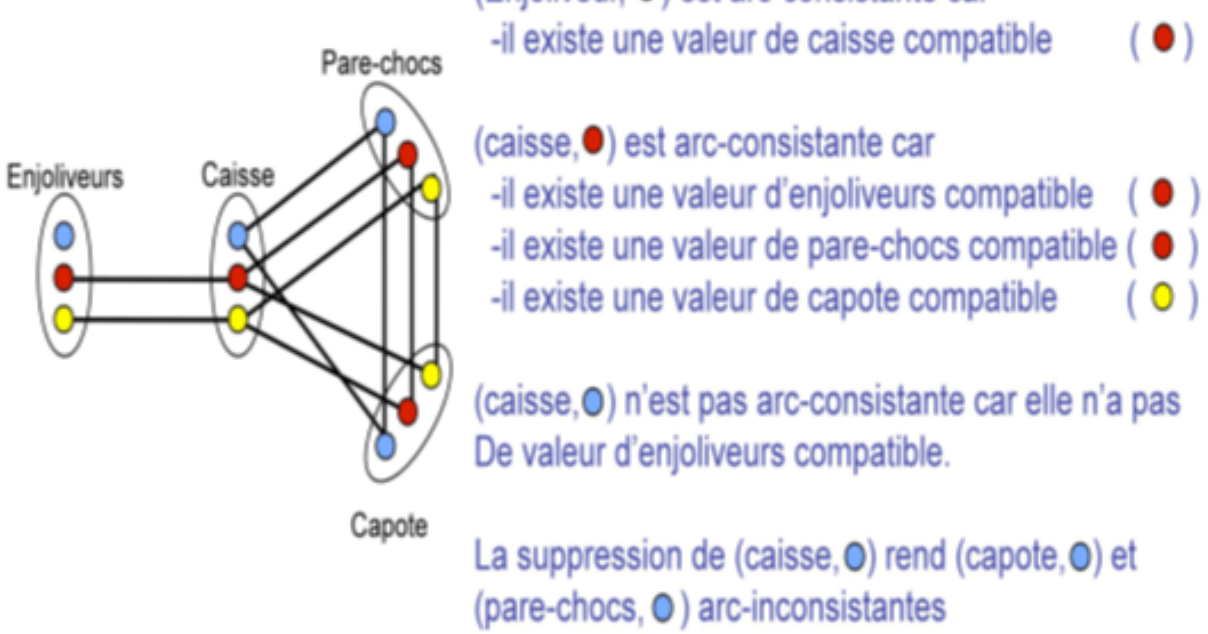

FIGURE 1.2 – Exemple d'une consistance d'arc

1.4.3.4 k-consistance

La k-consistance correspond à une généralisation des différentes notions vues jusqu'à présent et se définit ainsi :

Définition 1.4.3. (k-consistance) Soit s une instanciation partielle de longueur$k$, i.e. k variables sont instanciées pour un $CSP(X, D, C)$, si s satisfait toutes les contraintes, on dit alors que l'affectation est k-consistante [1].

1.4.3.5 Consistance hyper-arc

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

Définition 1.4.4. (Consistance hyper-arc) Soit un $CSP(X, D, C)$, et une contrainte $c \in C$ 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 $c \subseteq D_{x_1} \times D_{x_2} \times, \ldots, \times D_{x_n}$. On dit alors que c est hyper-arc consistante si pour chaque $i \in [1..n]$ et $a \in D_{x_i}$, il existe un n-uplet d

dans c de sorte que la i-ième composante de d soit égale à a. Un CSP est hyper-arc consistant si toutes ses contraintes sont hyper-arc consistantes. Une des faiblesses des consistances présentées est qu'elles considèrent les contraintes de manière isolée les unes des autres, ce qui leur confère l'aspect local, alors que dans la plupart des cas, elles partagent des variables [1].

1.4.3.6 Intérêt de l'arc-consistance

- réduit la combinatoire.
- ne touche pas aux solutions valides.
- il existe des algorithmes polynomiaux pour propager l'arc-consistance dans un CSP [8].

1.4.3.7 Propriétés des algorithmes de consistance d'arc

- Soit S_x le domaine courant de la variable X, soit c une contrainte sur X et Y ;
- la valeur $v_x \in s_x$ est arc-consistante pour c si : $\exists v_y \in s_y$ tel que $(v_x, v_y) \in sol(c)$;
- De même on dit que la contrainte c est arc-consistante si :
  $\forall v_x \in s_x$, v_x est arc-consistante et vice-versa en échangeant le rôle de$X$ et de Y.
-
- Finalement, un système de résolution de contraintes est :
  Arc-consistant ssi chaque contrainte est arc-consistante [9].

1.4.3.8 Les algorithmes qui rendent un système AC

de nombreuses procédures existent, nous ne considérerons que les CSP binaires on se limite au calcul de contraintes unaires, induites par des "sous-CSP" définis par des couples de variables, les algorithmes les plus anciens s'appuient sur la fonction Revise. Tel que AC, AC3.

Procédure Revise

L'ingrédient principal pour maintenir l'arc consistance ou la vérification en avant est la procédure revise, qui s'applique à une paire de variables $(x_i; x_j)$ reliées par une contrainte ck, cette procedure supprime du domaine de xi toute valeur qui rend impossible la satisfaction de la contrainte entre xi et xj, et qui retourne, en plus des effets de bord sur les domaines, un booléen qui indique si le domaine de xi a été modifié [10].

Algorithm 1 Réviser

$Reviser(c : contrainte; X, dx : variable) : Bool$
Var supprime :Bool , v,w : Valeurs ;
$Supprime \leftarrow faux$;

For $(x \leftarrow 0\ N\ )$

If il n'y a pas de $w \in d_y$ tq $(v, w) \in sol(c)$
supprimer v de d_x ;
$supprime \leftarrow vrai$;

RETOURNER supprime

Algorithme AC1

l'algorithme AC-1 utilise revise pour rendre un CSP arc-consistant, Il appèle $revise(x, y)$ pour toutes les paires de variables (x, y) liées par une contrainte binaire, et répète cette boucle jusqu'à ce que Revise retourne faux pour tous les couples de

variables[11].

Algorithm 2 AC1

Input CSP(X, D, C)
la partie initialisation $Q \leftarrow \emptyset$
foreach $V_i \in V$
foreach$V_j \in V$
If $R_{ij} \in R$
$Q \leftarrow Q \cup (V_i, V_j)$;
la partie propagationrepeatchange $\leftarrow False$;
foreach $(V_i, V_j) \in Q$
if$D_i = \emptyset$
return False ;
change $\leftarrow True$
until not (change)
Return true

Algorithme AC3 Dans cette section, nous présentons l'algorithme de base pour la consistance d'arc, à savoir AC3, Même si cet algorithme a subi la concurrence sévère d'algorithmes tels que AC4, AC6 et AC7, sa simplicité, ainsi que son efficacité relative lui ont assuré une place de choix dans de nombreuses implémentations.
Une amélioration possible pour la maintenance de l'arc consistance est d'appeler revise(x, y) seulement quand le domaine de y a diminué, en particulier lors d'une affectation de y, quand le domaine a été réduit à un singleton. Cet algorithme travaille avec un ensemble de variables dont le domaine a diminué.
AC3(y) maintient l'arc consistance après une affectation à y. Initialement $Q = y$. Puis tant que Q n'est pas vide : $y = Q$.dépile(). Pour toute variable $x$liée à y, appeler $revise(x, y)$. Si l'appel retourne vrai, alors ajouter x à Q s'il n'y est déjà [11].

Algorithm 3 AC3

la partie initialisation
$Q \leftarrow \emptyset$
foreach $V_i \in V$
foreach$\mathrm{V}_j \in V$
If $R_{ij} \in R$
$Q \leftarrow Q \cup (V_i, V_j)$;
la partie propagation
while$\mathrm{Q} \neq \emptyset$
selectionnée et suprimé $(V_i, V_j) \in Q$;
if revise(V_i, V_j)
if $D_i = \emptyset$
return False ;
else
foreach variable $V_k \in Vetk \neq j$
if $R_{ki} \in R$
$Q \leftarrow Q \cup (V_k, V_i)$;

return True

Algorithme AC4 AC3 étant non optimale, Mohr et Henderson a proposé AC4 pour améliorer le temps de la complexité . L'idée de AC4, contrairement à AC3, est de stocker plusieurs 'informations. AC3 effectue le montant minimum de travail à l'intérieur d'un appel revise, tout en veillant à ce que toutes les valeurs restantes de xi sont compatibles avec c et la mémorisation de rien. Le prix à payer est de refaire une grande partie du travail si le même revise est rappelé. AC4 stocke la quantité maximum d'informations dans une étape de prétraitement afin d'éviter de refaire plusieurs fois la même vérification de contrainte lors de la propagation des suppressions. AC4 est présenté dans l'algorithme 4. sa complexité temporelle est O(ed2) [12].

Dans AC-4 l'ensemble des tuples est pré calculé et sauvegardé dans une structure de données que nous appellerons table. Cette table contient pour chaque valeur $(x; a)$ un pointeur vers le prochain tuple contenant $(x; a)$.
Plutôt que de vérifier régulièrement qu'une valeur $v \in D_y$ a encore un support dans

D_x à chaque fois que D_x a diminué, on peut travailler avec une structure de données plus subtile, qui stocke pour chaque triplet (x, u, y), le support de $x := u$ dans le domaine de y, c'est-à-dire $v : v \in D_y$, $(u, v) \in R_{x,y}$. Notons support$[x, u, y]$ cette donnée.
Cet algorithme travaille alors avec un ensemble Q de couples variable-valeur. Initialement pour établir l'arc consistance Q contient tous les couples (x, u) tel qu'il existe une variable y avec support $[x, u, y] = \varnothing$. Alors que pour maintenir l'arc consistance après une affectation $x := u$, la valeur initiale pour Q sera $f(x, w) : w \in D_x, w \neq u$. Où Dx est le domaine de x avant l'affectation. Puis tant que Q n'est pas vide, on extrait un couple (x, u) de Q : Puis on supprime u du domaine de x, et pour chaque variable y et valeur $v \in \mathit{support}[x, u, y]$, on enlève u de support $[y, v, x]$. Si jamais cet ensemble est devenu vide, alors on ajoute (y, v) à Q.

Algorithm 4 AC4

function AC4(in X : **set) : Boolean ;**
la partie initialisation
$Q \leftarrow \emptyset$
foreach $V_i \in V$
foreach$V_j \in V$
If $R_{ij} \in R$
$Q \leftarrow Q \cup (V_i, V_j)$;
la partie propagation
whileQ $\neq \emptyset$
selectionnée et suprimé $(V_i, V_j) \in Q$;
if revise(V_i, V_j)
if $D_i = \emptyset$
return False ;
else
foreach variable $V_k \in Vetk \neq j$
if $R_{ki} \in R$
$Q \leftarrow Q \cup (V_k, V_i)$;

return True

1.5 Les methodes de résolution parallèles

1.5.1 Définition du parallèlisme

La mise en œuvre du parallélisme correspond à l'exécution simultanée de plusieurs instructions. Il est nécessaire que ces instructions soient logiquement indépendantes, faute de quoi l'exécution séquentielle sera obligatoire [36].

1.5.2 Performance d'un algorithme parallèle

On peut considérer plusieurs mesures de la performance d'un algorithme parallèle, telles que Les plus connues sont l'accélération et l'efficacité ; La première consiste à positionner l'algorithme parallèle par rapport aux algorithmes classiques de la littérature, alors que la seconde concerne la qualité de la parallélisation.
En fait, la première mesure est une simple comparaison des résultats de l'algorithme parallèle et de ceux obtenus par les principaux algorithmes de la littérature (qu'ils soient séquentiels ou Parallèles). Elle permet d'évaluer l'intérêt pratique de l'utilisation de l'algorithme parallèle pour résoudre une tâche (ou un ensemble de tâches) donnée. Une telle étude comparative étant habituellement menée pour tout nouvel algorithme (parallèle ou non), nous ne nous étendrons pas sur ce sujet.
La parallélisation d'un algorithme séquentiel à pour but d'accélérer l'exécution de la tâche accomplie par cet algorithme. Aussi, quand on souhaite évaluer la qualité d'un algorithme parallèle, il semble naturel de le comparer à une version séquentielle de cet algorithme, ce qui conduit aux notions d'accélération et d'efficacité. Dans les définitions de ces deux notions, nous employons volontairement le terme plus général de "processus" en lieu et place du terme habituel de "processeur".
Le but est tout simplement de pouvoir utiliser ces notions aussi bien dans un cadre multi-processeurs que mono-processeur. Cette liberté nous apparaît d'autant plus nécessaire qu'en pratique, nous ne pouvons pas toujours exécuter un seul processus par processeur, faute de disposer d'un nombre suffisant de processeurs [37].

1.5.3 Classification des systèmes parallèles

En 1966, Flynn a proposé une classification en quatre groupes des systèmes parallèles [38] :

1. **Single Instruction, Single Data (SISD)**
   Cette catégorie correspond aux stations capables de ne traiter par cycle d'horloge qu'une instruction sur une donnée par cycle d'horloge.

2. **Single Instruction, Multiple Data (SIMD)**
   Chaque processeur de cette architecture exécute la même instruction à chaque cycle d'horloge, mais les données traitées sont différentes. L'exécution est synchrone et déterministe sur chaque processeur. Ces machines sont adaptées aux traitements réguliers, comme le calcul matriciel sur matrices pleines ou le traitement d'images.

3. **Multiple Instruction, Single Data (MISD)** Il existe peu d'exemples de cette classe de systèmes parallèles. Elle correspond aux systèmes capables d'exécuter plusieurs instructions sur la même donnée durant le même cycle d'horloge.

4. **Multiple Instruction, Multiple Data (MIMD)**
   C'est dans cette catégorie que l'on trouve le plus de systèmes parallèles. A chaque cycle d'horloge, chaque processeur exécute une instruction différente sur une donnée différente. Ces exécutions peuvent être synchrones ou non, déterministes ou non.

1.5.4 Modèle de programmation parallèle

Il existe plusieurs modèles pour la programmation d'applications parallèles. Ces modèles ne tiennent pas compte de l'architecture matérielle (processeurs et mé-

moires). Chacun d'entre eux peut être réalisé quel que soit le choix de l'architecture matérielle [38].

1.5.4.1 Programmation par mémoire partagée

Dans ce modèle les différentes tâches partagent le même adressage mémoire, elles peuvent lire et écrire dedans de manière indépendante et asynchrone. Cela permet de s'affranchir du problème de la communication des données entre les tâches. Mais le principal désavantage de ce modèle est que la cohérence des données et les accès concurrents doivent être gérés par le programmeur à l'aide de sémaphores ou de verrous, au risque de diminuer les performances du système.

1.5.4.2 Programmation par envoi de message

Dans ce modèle, chaque processeur utilise sa propre mémoire locale. La communication des données et la synchronisation se font à l'aide de message dont le format est laissé à la discrétion du programmeur. Les différentes instances de l'application répartie doivent être synchronisées, en effet, l'envoi d'un message doit faire l'objet d'une réception explicite par le destinataire.
Il existe des bibliothèques pour implémenter le passage de messages :

- Message Passing Interface (MPI).
- Le standard de facto.
- Parallel Virtual Machine (PVM).

1.5.4.3 Programmation sur des données en parallèle

Dans ce modèle, les données sont découpées et distribuées vers les différentes unités de calcul. Ces dernières appliquent les mêmes traitements aux données qui leur sont envoyées. Si on a une architecture à mémoire partagée, les différentes tâches accèdent aux données grâce à la mémoire globale, sinon, dans le cas de mémoires distribuées, les données sont transmissent à chaque unité qui les copie dans sa propre mémoire locale.

1.5.4.4 Hybride

Dans ce modèle, plusieurs modèles de programmation présentés précédemment sont combinés. Par exemple, on peut citer l'utilisation simultanée de threads et de communications par passages de messages ou par la mémoire partagée. Ce qui est souvent le cas pour les machines SMP mises en réseau.

1.5.5 Les architectures parallèles

1.5.5.1 Mémoire partagée

Le premier type d'architecture mémoire pour une application distribuée est la mémoire partagée. Dans ce cas, plusieurs processeurs ont accès à la même mémoire physique. Ils peuvent opérer avec elle de manière indépendante et les changements faits par l'un des processeurs sont immédiatement visibles par les autres [38].

Il existe deux types de mémoires partagées :

- Les mémoires à accès uniformes (UMA : Uniform Memory Access) où l'accès à la mémoire est le même pour chaque processeur, on parle alors d'accès

équitable. Pour cela, il faut que les processeurs soient identiques.

– Les mémoires à accès non uniformes (NUMA : Non Uniform Memory Access) où chaque processeur peut accéder à la mémoire indépendamment des autres. Ils peuvent donc être de types et de vitesses différentes. Comme pour l'architecture UMA, on peut y rajouter un système de cohérence de cache, mais l'implémentation de ce dernier est compliquée par ces accès non uniformes.

Ce type de mémoire présente l'avantage de permettre un partage immédiat des données, facilitant la programmation. Mais cette solution coûte chère, ce qui limite le nombre de processeurs que l'on peut ajouter sur une même mémoire. De plus, les mécanismes de cohérence de cache sont coûteux en performance et plus on ajoute de processeurs, plus ce type de mécanisme devient indispensable.

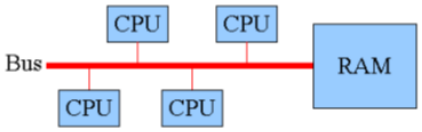

FIGURE 1.3 – Architecture simplifiée d'une machine à mémoire partagé

1.5.5.2 Mémoire distribuée

Dans ce cas, chaque processeur possède sa propre mémoire. La modification par l'un des processeurs de sa propre mémoire n'a pas d'influence directe sur celle des autres processeurs. Cela suppose donc de mettre en place une communication explicite entre les processeurs (souvent par l'intermédiaire d'un réseau).
Ce type d'architecture présente l'avantage de permettre une hausse des performances processeurs / mémoires plus intéressante que dans le cas de la mémoire partagée, mais c'est au programmeur de gérer la plupart des détails de la communication entre les unités de calcul. Elle rend également difficiles les échanges complets de structures de données, pose des problèmes d'accès non uniformes dans le temps et

elle rend la cohérence de données plus dure à maintenir[38].

1.5.5.3 Mémoire partagée et distribuée

Ce dernier type de mémoire est un mélange des deux premiers. Dans cette architecture, il y a plusieurs groupes de processeurs partageant de la mémoire qui communiquent grâce à un réseau. Cela permet, dans une certaine mesure, de tirer les avantages de deux précédentes architectures et d'en réduire les inconvénients [38].

1.5.6 Les techniques de résolution parallèle des CSP

On peut distinguer deux grandes catégories d'approches pour la résolution de contraintes parallèles [8] :

1.5.6.1 Recherche concurrente

Chaque processeur exécute le même algorithme de recherche avec un ordre différent sur les variables ou carrément des algorithmes différents. L'intérêt est que les exécutions sont totalement indépendantes, quand un processus termine sa recherche c'est la fin globale de l'algorithme. La méthode n'est pas trop efficace (quantité de travail est augmentée pour le même calcul).

1.5.6.2 Résolution parallèle au niveau du nœud

La recherche se déroule de façon séquentielle, au niveau d'un nœud les contraintes sont vérifiées en parallèle mais les contraintes concernées ne sont pas les mêmes, ce qui pose problème (allocation statique ou dynamique de la charge).
Certains auteurs ont préféré l'allocation statique, les processeurs libres peuvent

exécuter par exemple un algorithme de filtrage.

1.5.6.3 Distribution de l'arbre de recherche

La distribution de l'arbre de recherche est connue sous le nom d'OÙ-parallèle en programmation logique. L'exploitation en parallèle des branches conduit à une solution si une branche au moins aboutit à une solution.
Les différentes parties séquentielles s'exécutent indépendamment, la difficulté réside dans la technique de distribution et de l'équilibrage de charge.

1.6 Conclusion

Dans ce premier chapitre, nous avons d'abord rappelé la notion de CSP avant de présenter quelques unes des méthodes de résolution de CSPs. Nous avons présenté surtout les méthodes énumératives qui pour être efficaces doivent s'appuyer sur des heuristiques et sur des techniques de filtrages.
Dans le prochain chapitre, nous allons présenter les techniques de décomposition structurelles permettant de borner cette complexité théorique.

Chapitre 2

Les techniques de décomposition structurelles

2.1 Introduction

Les problèmes de satisfaction de contraintes forment une grande classe de problèmes combinatoires qui contient de nombreux problèmes importants "du monde réel". L'objectif de CSP est de trouver une instanciation possible pour chaque variable qui satisfait toutes les contraintes. De nombreuses méthodes ont été proposées pour résoudre les CSPs, parmi ces méthodes on trouve la décomposition structurelle.

2.2 Quelques Définitions de base

2.2.1 Graphe

Un graphe G est un couple (V, E) ou V est l'ensemble des sommets de G, et E est l'ensemble de ses arêtes, une arête étant un ensemble de deux éléments de V. Une arête caractérise donc un lien entre deux sommets du graphe [13].

2.2.2 Connexité

Un graphe est connexe s'il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres en suivant les arêtes. Un graphe non connexe se décompose en composantes connexes [14].

2.2.3 Composant connexe

Une composante connexe C d'un graphe G = (S,A) est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne :
si $x \in C$, alors $\forall y \in C$, il existe une chaîne reliant x à y.
$\forall z \in S\ C$, il n'existe pas de chaîne reliant x à z[15].

2.2.4 Un cycle

Un cycle est une suite $(x_1, e_1, x_2, ..., x_k, e_k, x_1)$ de sommets et d'arêtes de H tous distincts, et tels que x_i, $x_{i+1} \in e_i$ pour $i = 1, ..., k$ (en considérant $x_{k+1} = x_1$). La longueur d'un cycle est son nombre d'arêtes, soit k. On dit qu'un cycle est pair (resp. impair) s'il est de longueur paire (resp. impaire) [13].

2.2.5 Hypergraphes

Un hypergraphe H est un couple (V, E) ou V est l'ensemble des sommets de H, et E est l'ensemble des hyperarêtes de H (également appelées plus simplement arêtes), chaque arête étant un sous-ensemble de V. S'ils ne sont pas nommés explicitement, on désignera par V(H) l'ensemble des sommets de H, et par E(H) l'ensemble de ses arêtes. Les hypergraphes sont donc une généralisation des graphes, dans le sens ou une arête contient un nombre quelconque de sommets, et pas nécessairement deux.

La plupart des définitions sont alors similaires, mais demandent toutefois d'être adaptées[13].

2.2.6 Un séparateur

Un séparateur est un sous-ensemble W de sommets dans un graphe connexe $G = (V, E)$ tel que le graphe $G[V - W]$ est non connexe [14].

2.2.7 Graphe primal

Graphe primal d'un hypergraphe H =<V,E> est un graphe G =(V,E') et il existe une arête entre deux sommets dans G si ces derniers participent à une même hyperarêtes dans H [16].

2.2.8 Graphe dual

Le dual d'un hypergraphe $H = (V, E)$ est un graphe $G = (E, E')$ dont les sommets $e_1, \ldots, e_m$ correspondent aux hyperarêtes de H et il existe une arête $\{e_i, e_j\} \in E'$ si les hyperarêtes $\{e_i, e_j\} \in E$ partagent au moins une variable $(ie : e_i \subseteq e_j\sigma = \varnothing)$ [16].

2.2.9 Chaine

Une chaîne dans G, est une suite ayant pour éléments alternativement des sommets et des arêtes, commençant et se terminant par un sommet, et telle que chaque arête est encadrée par ses extrémités.
On dira que la chaîne relie le premier sommet de la suite au dernier sommet. En

plus, on dira que la chaîne a pour longueur le nombre d'arêtes de la chaîne [14].

2.3 traitabilité

2.3.1 définition d'un probléme

un probléme est dit traitable s'il existe un algorithme polynomial permettant de répondre a la question de P [16].

2.3.2 Acyclicité d'un hypergraphe

Un hypergraphe H est acyclique si, et seulement si, il possède un join tree [17].

2.3.3 Join Tree

Un join tree est un arbre T dont les nœuds sont les hyperarêtes de H tel que si un sommet x de H appartient à deux hyperarêtes différentes h1et h2 dans H, alors x se produit dans chaque nœud sur le chemin unique reliant h1 et h2 dans T.

2.3.4 Graphe triangulé

Un graphe est triangulé si tous ses cycles de plus de 3 sommets contiennent au moins une corde (arête reliant deux sommets non adjacents d'un cycle) [14].

2.4 Les principales méthodes structurelles

2.4.1 Principe de ces méthodes

L'objectif d'une méthode de décomposition est de transformer une instance CSP cyclique en une autre instance équivalente acyclique.
Cette méthode utilise la notion de largeur (width) qui est le critère de mesure de la cyclicité de l'hypergraphe de contrainte, telle que pour une constante k fixe, toute instance CSP peut avoir une décomposition de largeur inferieure ou égale à k. Après la décomposition, le CSP peut être résolu en un temps polynomiale (en fonction de la largeur de la décomposition), c'est pour cela l'objectif principal de toutes les méthodes de décomposition est de minimiser cette largeur.

2.4.2 La méthode CCM Cycle Cutset

Cette méthode est proposée par R.Dechter dans [18]. Un ensemble de cycle d'un hypergraphe H est un ensemble $S \subseteq var(H)$ de telle sorte que le sous-graphe du graphe primal de H (Vertex) induite par var $(H) - S$ est acyclique. Autrement dit, après la suppression des sommets de S, le graphe primal de H devient acyclique. La largeur de d l'ensemble CUSTEL de H est 1 si H est acyclique ; sinon, elle est la cardinalité minimale possible sur toute sa cycle de coupe ensembles.
La complexité temporelle de cette méthode est en $O(dk + 2)$ où k est la taille de l'ensemble coupe cycle et d est la taille du plus large domaine. En effet, le coût nécessaire à la recherche d'une affectation consistante sur l'ensemble coupe cycle est $O(d2)$. Pour chaque affectation consistante, il faut résoudre le CSP acyclique induit. La complexité en temps de cette tâche est en $O(d2)$ et donc la complexité totale est $O(d2)$. La complexité totale en espace est linéaire .

2.4.3 La méthode cycle Hypercutset

Cette méthode est proposée par R.Dechter dans [18]. C'est une simple modification de la méthode de coupe-cycle où l'ensemble CUSTEL se compose d'hyperarêtes plutôt que les sommets de l'hypergraphe donné. Un ensemble hypercustet d'un hypergraphe H est un ensemble d'hyperarêtes $H' \subseteq arrt(H)$ de telle sorte que le sous-hypergraphe induit par $var(H) - var(H')$ est acyclique. La largeur HYPERCUTSET de h est 1 Si est acyclique ; sinon, elle est la cardinalité minimale sur l'ensemble de ses hypercutsets possibles .

2.4.4 La méthode Biconnected Components (BICOMP)

Cette méthode a été introduite dans [19].
Soit $G = (V, E)$ un graphe. Un sommet $p \in V$ est un sommet de séparation pour G si, par la suppression de p à partir de G, le nombre de composantes connexes de G augmente.
A biconnexes composante de G est un ensemble de sommets maximal $C \subseteq V$ tel que le sous-graphe de G induite par C est connecté et reste connecté après tout retrait d'un sommet, à savoir, n'a pas de sommets de séparation. Il est bien connu que, de tout graphe G, on peut calculer en temps linéaire un sommet marqué Arbre (T, X), où la fonction d'étiquetage X est une fonction bijective qui associe à chaque sommet de l'arborescence T un ensemble de sommets S de G, tel que S est soit un composant biconnexes de G, ou un singleton contenant un sommet de séparation pour G. Il existe un bord ; $\{p, q\}$ dans l'arbre T, si $X(p)$ est un composant biconnexes de G et $X(q)$ contient un sommet de séparation G appartenant à la composante $X(p)$, i.e , $X(q) \subseteq X(p)$, est titulaire. Nous disons que (T, X) Est la décomposition BICOMP de G.
Pour un hypergraphe H, la décomposition de BICOMP de Sa décomposition BICOMP de son graphe primal, et la largeur de biconnexes, notée $BICOMP - largeur(H)$, est le nombre maximum de sommets sur les com-

posants biconnexes du primal graphique de H .

2.4.5 La méthode TCLUSTER (Tree Clustering)

Cette méthode a été introduite dans [20].
Le procédé de mise en grappe d'arbre est basé sur un algorithme de triangulation, qui transforme le graphe primal $G = (V, E)$ de toute instance CSP Dans un graphe G' chordal. Le hypergraphe acyclique $H(G')$ ayant le même ensemble de sommets que G' et les cliques maximales de G' comme ses hyperarêtes est une décomposition de CLUSTER de HI. Intuitivement, les quantifications de H (G') sont utilisés pour construire les contraintes d'un I' CSP acyclique équivalent à I. La largeur de la CLUSTER décomposition H (G') est le maximum cardinalité de ses quantifications. L'arbre-regroupement avec (court : TCLUSTERwidth) de H est 1 si HE est un hypergraphe acyclique ; sinon, elle est égale à la largeur minimale sur la décomposition de CLUSTER de HI .

La description de TCLUSTER est donnée par l'algorithme 1.

Algorithm 5 TCLUSTER

Input : Un CSP P et son graphe primal G.
Output : Une solution si elle existe.

1 : Trianguler le graphe primal G .
2 : Identifier les cliques maximales dans le graphe primal triangulé. Soient $C_1, \ldots, C_t$ de telles cliques indexées par les rangs des nœuds les plus élevés.
3 : Former le graphe dual correspondant aux clusters et identifier un de ses join tree en connectant chaque cluster C_i avec un autre cluster C_j (avec $j < i$) avec lequel il partage le maximum de variable.
4 : Résoudre les sous-problèmes définis par $C_1, \ldots, C_t$.
5 : Résoudre le CSP obtenu en considérant les clusters comme des variables singleton comme suit :
1. Réaliser une arc consistance directionnelle (DAC) sur le join tree.
2. Résoudre le join tree par un algorithme sans backtrack.

2.4.6 La méthode TD (Tree decomposition)

Cette méthode a été introduite dans [21].
Une décomposition arborescente d'un graphe $G = (V, E)$ est une paire (T, X), où $T = (N, F)$ est un arbre, et X est une fonction d'étiquetage associant à chaque sommet $P \in N$ ensemble de sommets $X(p) \subseteq V$, de telle sorte que les conditions suivantes sont satisfaites :

- pour chaque sommet a de G, il existe un sommet $P \in N$ tel que $a \in X(p)$.
- pour chaque arête $a, b \in E$, il existe un sommet $P \in N$tel que $a, b \subseteq X(p)$.
- pour chaque sommet a de G, l'ensemble $P \in N/a \in X(p)$; induit (connecter) un Sous-arbre T.

La largeur d'une décomposition arborescente (T, X), Est $max_{p \in N}|X(p) - 1|$, la Treewidth de G est la largeur minimale sur toutes ses décompositions d'arbres. La

Treewidth d'un hypergraphe H est 1 si H est un hypergraphe acyclique ; sinon, elle est égale à la largeur arborescente de son graphe primal .

2.4.7 La méthode BTD (Backtracking on Tree-Decomposition)

Cette méthode a été introduite dans [22].
La méthode BTS (pour Backtracking avec Tree-Decomposition) procède par une recherche énumérative guidée par un ordre partiel préétablie statique induite par une décomposition arborescente de la contrainte-réseau. Ainsi, la première étape de BTD consiste à calculer une décomposition arborescente ou une approximation d'une décomposition arborescente. L'ordre partiel obtenu permet d'exploiter certaines propriétés structurelles du graphe, au cours de la recherche, afin d'élaguer certaines branches de l'arbre de recherche. Par conséquent, ce qui le distingue des autres techniques BTD concerne les points suivants :

1. l'ordre d'instanciation des variables est induit par une décomposition arborescente de la contrainte graphique,

2. certaines parties de l'espace de recherche ne seront pas visités à nouveau une fois que leur cohérence est enfin déterminée (notion de good structurel) certaines parties de l'espace de recherche ne seront pas rendus à nouveau si l'on sait que la courante instanciation conduit à un échec (notion de nogood structurel) .

Définition 2.4.1. (Good et Nogood) Soient Ci un cluster et Cj un de ses fis, un goode(resp. Nogood) de C_i par rapport à C_j, notée $g(C_i, C_j)$ (resp.$ng(C_i, C_j)$), est une affectation consistante sur $C_i \cap C_j$ de sorte qu'il n'y existe (resp. n'existe pas) une extension consistante de cette affectation sur Desc (C_j). Où $Desc(C_j)$ est l'ensemble des variables du sous-arbre enraciné au niveau de C_j.

BTD combine l'efficacité pratique de l'énumération et les bornes de complexités issues de la décomposition arborescente du graphe de contraintes. Elle exploite particulièrement les notions de nogood et de good structurels a n d'éviter certaines redondances. Les variables sont affectées selon l'ordre induit par la décomposition arborescente. Mais au sein d'un cluster, elle exploite un ordre d'affectation dynamique.

L'algorithme BTD (algorithme 6) procède comme suit : au cours de la recherche, il choisit une variable x non instanciée du cluster courant E_i (ligne 24), si elle existe, et lui affecte une valeur v (ligne 28). Si $A \cup \{x \leftarrow v\}$est inconsistante, BTD choisit une nouvelle valeur pour x (boucle while : lignes 27-33). Sinon, l'instruction $BTD(A \cup x \leftarrow v,\ E_i\ ,\ V_{Ei}\ x)$ (ligne 31) va essayer d'étendre A $\cup\{x \leftarrow v\}$ sur $VE_i\ \{x\}$ (VE_i désigne l'ensemble des variables du cluster E_i). En cas d'échec, une nouvelle valeur pour x est considérée. Si toutes les valeurs ont été essayées, un retour-arrière est effectué sur la variable précédente. Si toutes les variables de Ei ont été instanciées de manière consistante, alors BTD choisit un lsE_j de E_i (ligne 5). Trois cas se présentent alors : 1. Si $A[E_i \cap E_j]$ est un good (ligne 7), alors A peut être étendue de manière consistante sur le sous-problème enraciné en Ej . Dans ce cas, un saut en avant "forward-jump" est effectué pour continuer l'énumération avec la première variable après celles de $Desc(E_j)$ dans l'ordre induit par la décomposition. 2. $A[E_i \cap E_j]$ est un nogood (ligne 10), alors A ne peut être étendue en une solution, alors l'affectation actuelle des variables de $[E_i \cap E_j]$ doit être modiée. 3. Si $A[E_i \cap E_j]$ n'est ni un good ni un nogood, alors BTD va tenter d'étendre A sur les variables du sous-arbre enraciné en Ej (ligne 13). En cas de succès, BTD enregistre $A[E_i \cap E_j]$ comme un good (ligne 15), sinon $A[E_i \cap E_j]$ est enregistré comme un nogood (ligne 17). Si A a été étendue de façon consistante sur toutes les variables du sous-arbre enraciné en Ei alors BTD retourne True, sinon il retourne False (lignes 4-21). Finalement, $BTD(\emptyset, E1, VE1)$ retourne donc True si l'instance P est consistante, False sinon.

La complexité temporelle de BTD est en O(exp(w + 1)) tandis que sa com-

Algorithm 6 BTD

Input : l'affectation A courante, initialement $A = \{\}$
Input : le cluster Ei courant, initialement Ei = E1
Input : VEi : ensemble des variables du cluster Ei qui ne sont pas encore instanciées.
Output : True si A a été étendue avec succès à toutes les variables de la descendance de Ei , False sinon.
1 : **if** $VE_i = \{\}$ then
2 : Consistency ← True
3 : $F = Sons(Ei)/*\, Sons(Ei)$ désigne les ls de $E_i*/$
4 : **while** $F6 = \{\}$ and Consistency do
5 : choisir E_j in$F$
6 : $F = F\ E_j$
7 : **if** $A[E_i \cap E_j]$ est un good then
8 : Consistency =True
9 : **else**
10 : **if** $A[E_i \cap E_j]$est un nogood then
11 : Consistency ← False
12 : **else**
13 : Consistency ← $BTD(A, Ej, Ej\ (Ei?Ej))$
14 : **if** Consistency then
15 : Enregistrer le good $A[E_i \cap E_j]$
16 : **else**
17 : Enregistrer the nogood $A[E_i \cap E_j]$
18 : **end if**
19 : **end if**
20 : **end if**
21 : **while**
22 : Return Consistency
23 : **else**
24 : Choisir$x \in VE_i$
25 : $d_x = D_x$
26 : Consistency ← $False$
27 : **while** $d_x6 = \{\}et\neg Consistency$ do
28 : Choisir v dans d_x
29 : $d_x \leftarrow d_x\ v$
30 : **if** 6 $\exists C_i \in C$ tel que C_i n'est pas satisfaite par $A \cup x \leftarrow v$ then
31 : Consistency ← BTD(A $\cup x \leftarrow v, E_i, VE_i\{x)$
32 : **end if**
33 : **while**
34 : Return Consistency
35 : **end if**

plexité spatiale est en O(n • s • exp(s)) avec s la taille de la plus grande intersection entre clusters et w est la largeur de la décomposition arborescente utilisée.

2.4.8 La méthode Hinge

La méthode de décomposition Hinge (HINGE) est proposée par Gyssens et al. [23] pour la décomposition des hypergraphes.

2.4.8.1 Définition Hinge

Soit $H = (V, E)$ un hypergraphe connexe et soit H une hyperarête E ou bien un sous-ensemble de E contenant au moins deux arêtes. Soient $H_1, \ldots, H_m$ les composantes connexes de $E - H$ relativement à H. H est dit un Hinge si, pour $i = 1, \ldots, m$, il existe une arête $h_i \in H$ telle que : $(\bigcup H_i) \bigcap (\bigcup H) \subseteq h_i$. Un hinge est dit minimal, s'il ne contient aucun autre hinge.

2.4.8.2 Définition : décomposition Hinge

Soit H= (V, E) un hypergraphe. Une décomposition hinge de H est un arbre (N, A) de sorte que :
Les nœuds de l'arbre sont des hinges minimaux de H
Chaque bord dans E est contenu dans au moins un nœud de l'arbre.
Deux nœuds adjacents partagent exactement une arête de E.
Les sommets de partagé par deux nœuds d'arbre sont entièrement contenus dans chaque nœud de l'arbre sur leur chemin de connexion.

2.4.9 La méthode GHD (Décomposition en hypertree généralisée)

La méthode de décomposition en hyperarbre généralisée (Generalized Hypertree Decomposition) GHD est proposée dans [25].

2.4.9.1 Hypertree

Un hypertree pour un hypergraphe H est un triplet $< T, \chi, \lambda >$ où $< T = (N, E) >$ est un arbre enraciné et χ et λ sont deux fonctions d'étiquetage associant à chaque sommet (nœud) p de N un ensemble de variables $\chi(p)$ et un ensemble de contraintes $\lambda(p)$. On note aussi sommets(T) l'ensemble des sommets de T et on note la racine de T par root (T). Tp indique l'ensemble des variables du sous arbre qui a pour racine le nœud p.

2.4.9.2 Décomposition en hypertree généralisée

Une décomposition hypertree généralisée d'un hypergraphe H $=<$ V, E $>$ est un hypertree HD $=< T, \chi, \lambda >$ qui satisfait les conditions suivantes :
1− Pour chaque hyperarêtes $h \in E$, il existe $p \in sommet(T)$ telle que $var(h) \subseteq \chi(p)$. On dit que p couvre h.
2− Pour chaque variable $v \in V$, l'ensemble $p \in sommets(T)|v \in (p)$ induit un sous arbre connexe de T.
3− Pour chaque sommet $p \in sommets(T)$, $\chi(p) \subseteq var(\lambda(p))$.

La figure montre l'exemple d'un hypergraphe et son hypertree décomposition généralisée.

2.4.9.3 La décomposition en hyperarbre (hypertree décomposition)

L'hypertree décomposition d'un hypergraphe H $=<$ V , E $>$, est une hypertree décomposition généralisée HD $=< T, \chi, \lambda >$ qui satisfait la condition supplémentaire suivante :
Pour chaque sommet $p \in sommets(T)$, $var((p))cap\chi(T(T_p)\chi(p)$.
La largeur d'une hypertree décomposition généralisée $< T, \chi, \lambda >$ est $MAX_{p \in vertices(T)}|\lambda(p)|$. La largeur d'une décomposition en hypertree (généralisée) $(g)htw(H)$ d'un hypergraphe H est la largeur minimum de toutes ses décompositions en hypertree (généralisée).

Définition 2.4.2. Une hyperarêtes h est fortement couverte dans un hyperarbre s'il existe un nœud p tel que $var(h) \subseteq \chi(p) eth \in \lambda(p)$.

2.4.9.4 Définition

Un hyperarbre décomposition $< T, \chi, \lambda >$ d'un hypergraphe H $=< V, E >$ est complète si chaque hyperarêtes h de H est fortement couverte dans HD $=< T, \chi, \lambda >$.
Calcule d'une (G) HD Calculer une hypertree décomposition généralisée optimale est un problème NP difficile. Pour cette dernière, il existe deux approches de calcul : les méthodes exactes et les heuristiques.

Les méthodes exactes

Parmi les méthodes exactes, nous citons opt-k-decomp qui permet de calculer une décomposition de largeur optimale bornée par une constante k fixée si une telle décomposition existe. Cependant, ces algorithmes dits exacts ne sont efficaces que pour les problèmes de petites tailles. Pour contourner ce problème, des méthodes heuristiques sont proposées.

Les méthodes heuristiques

Beaucoup de méthodes heuristiques ont été proposées pour calculer une décomposition en hyperarbre (généralisée). Korimort a proposé une heuristique basée sur la connectivité des sommets de l'hypergraphe. Dermaku et al ont proposé les heuristiques suivantes : BE (Bucket Elimination), DBE (Dual Bucket Elimination) et les techniques de partitionnement de l'hypergraphe. Musliu et Schahauser ont utilisé les algorithmes génétiques, etc.

2.4.9.5 Résolution des instances CSP via une (G) HD

Cette approche nécessite d'abord le calcul d'une décomposition par un algorithme exact ou une heuristique. (En pratique, les algorithmes exacts ne sont pas efficaces). Une fois que la décomposition est obtenue, on la complète de telle sorte que toutes les contraintes figurent dans au moins un nœud de l'hypertree.
Ensuite l'approche proposée par Gottlob et al [39] pour résoudre le CSP obtenu est donnée par l'algorithme 7.

Algorithm 7 Méthode de résolution proposée par Gottlob

Input : Une hypertree décomposition HD =< T ; X ; λ > $associeun CSP donn$.
Output :$Unesolution A$.
1 :$\delta = \{n_1, n_2, \ldots, n_m\}$un ordre sur les noeuds de l'hypertree decomposition ou n_1 est la racine et chaque noeud précéde ses fils dans l'ordre.
2 : foreach p noeud de l'hypertree do
3 : $Rp = join(\lambda(p)[X(p)]$.
4 : end.
5 : for $i = mto2$ do
6 : Soit v_j le pére de v_i dans l'ordre ;
7 : $R_j = semijoin(R_j, R_i)$.
8 : end.
9 : end.
10 : for $i = 2$ to m do
11 : Construire une solution A en sélectionnant un tuple dans R_i compatible avec tous ceux qui le précédent.
12 : end.
13 : return A .
14 : end.

2.4.10 Les décompositions gardées et Spread Cuts

Cette méthode a été proposée dans [28].

Définition 2.4.3. (bloc gardé) Un bloc gardé d'un hypergraphe H est une paire (λ, X), où λ est un sous-ensemble des hyperarêtes de H, et X est un sous-ensemble des sommets de la garde [23].

Définition 2.4.4. (recouvrement gardé complet) Un bloc gardé (λ, X) d'un hypergraphe H couvre une hyperarête e de H si $e \subseteq X$.Un ensemble de blocs gardés Ξ d'un hypergraphe H est appelé un recouvrement gardé de H si chaque hyperarêtes de H est couverte par un bloc gardé de Ξ.Un ensemble de blocs gardés Ξ d'un hypergraphe H est appelé un recouvrement gardé complet de H si pour chaque

hyperarêtes e de H se produit dans la garde d'un certain bloc gardé de Ξ recouvrant e.

Théorème 2.4.1. *Un ensemble de blocs gardés Ξ d'un hypergraphe H est une décomposition gardée de H si et seulement si Ξ est un recouvrement gardé complet de H.*

2.4.10.1 La méthode SCD (Spread cuts)

Définition 2.4.5. (X-composante) Soit $H = (V, E)$ un hypergraphe et $X \subseteq V$ un ensemble de sommets. Nous disons que deux hyperarêtes e, $f \in E$sont X-adjacent Si $e \cap fX$.
un X-chemin reliant ef est une séquence $e = e_0, e_1, \ldots, e_{r-1}$, $e_r = f$ telle que ei est un X-adjacent de e_{i+1}, pour tout $i = 0, \ldots, r-1$.Un ensemble d'hyperarêtes $C \subseteq E$est un X-connexe Si, pour toute paire des hyperarêtes de C, il y a un X-chemin qui les relie.
Un ensemble d'hyperarêtes est une X-composante d'hyperarêtes De H si il un sous ensemble non-vide, maximal et X-connexe de E.
Un ensemble non vide de sommets C est une X-composante de sommet de H s'il y a une hyperarête X-composante C_X Pour laquelle $C = \bigcup CX - X$ [29].

Définition 2.4.6. (unbroken components)
Un bloc gardé (λ, X) d'un hypergraphe H a des unbroken component si pour chaque x-composante de H se réunit au plus une Uλ-composante de H et $e_1 \cap e_2 | e_1, e_2 \in \lambda \subseteq X$.
un spread cut de H est un recouvrement gardé acyclique dans lequel tous les bloc gardé ont des unbroken component.

Définition 2.4.7. (label d'un sommet)
Soit λ est un sous ensemble d'hyperarêtes de H, on définit $L\lambda(v)$ un label de n'importe quel sommet de H est un ensemble d'hyperarêtes $\bigcup \lambda - composantes$ incluant une hyperarête contenant x.

On dit qu'un bloc gardé (λ, X) est canonique si pour chaque hyperarête $e \in \lambda$ les sommets de e en dehors de X sont exactement ceux qui ont un label particulier, $\exists v \in e - X$, quelque soit $w \in e$, $w \notin X \leftrightarrow L_\lambda(w) = L_\lambda(v)$.

Définition 2.4.8. (decomposition spread cut)
Une décomposition spread cut (SCD) est un ensemble de recouvrements gardés acycliques dans lequel tous les blocs gardés ont des unbroken components et ils sont tous canoniques.

2.4.10.2 La méthode Subedge-based decompositions

cette méthode a été proposé par Zoltan Miklos [31].

Proposition 2.4.1. *Soit H un hypergraphe et soit $D =< T, X, \lambda >$ une GHD de H. $D' =< T, X, \lambda >$, où $\lambda'(p) = e \cap X(p) | e \in X(P)$, pour chaque nœud p de T est une HD de $HUe \cap X(p) | p \in T, e \in \lambda(p)$. En outre, la largeur de D'est supérieur ou égale a la largeur D .*

Définition 2.4.9. (Décomposition spread cutNEW)
Nous disons qu'un algorithme A met en œuvre une méthode de décomposition M si pour chaque valeur fixe k et pour chaque hypergraphe d'entrée HA retourne une GHD de M (H) de largeur au plus k, s'il existe une telle décomposition, sinon il délivre "échouer".
Nous appelons un sous-ensemble d'une hyperarête e d'un hypergraphe H un Subedge de H. chaque fonction f liant un hypergraphe H à un ensemble de sous hyperarêtes de H induit à une méthode de décomposition qui peut être calculée de la manière suivante :
calculer f(H)
calculer une HD minimal de H UF(H).
Comme l'étape (2) est seulement possible pour une constante k, il est logique de

permettre à f de dépendre de k. Une fonction Subedge associe un ensemble de sous hyperarêtes pour une paire (H, k). Pour éviter les complications techniques, nous exigeons que les fonctions de Subedge doit être monotone [31].

Définition 2.4.10. (fonction subedge)
Une fonction de Subedge est une fonction f qui associe un ensemble de sous hyperarêtes de H à chaque paire (H, K) et pour chaque $i < j$, $f(H, i) \subseteq f(H, j)$.

Définition 2.4.11. Soit$D =< T, X, \lambda >$ une HD d'un hypergraphe $H \cup f(H, k)$ et $D' =< T, X, \lambda >$ un GHD de H. Nous disons que D' couvre D si, pour chaque nœud p de T, et pour chaque $e \in X(p)$, il existe une arête $e' \in \lambda'(p)$, de telle sorte que e est un Subedge de e', à savoir que vertices (e) inclus $vertices(e')$.

Si l'on ajoute sous-hyperarêtes à un hypergraphe H, alors la largeur de hyperarbre du hypergraphe résultant H' est au plus la largeur de hypertree de H, à savoir $htw(H') \leq htw(H)$. Ainsi, pour un hypergraphe H donné, et fonction Subedge f, la largeur hyperarbre de $H \cup f(H, i)$ est décroit de façon monotone (eni). La largeur de l'hyperarbre de $H(H, i)$ dépend de la structure de H et de la fonction f-Subedge. Nous tenons à définir une méthode de décomposition en utilisant Subedge-fonctions. Dans notre définition nous avons liée une GHDs d'un hypergraphe H à HDS d'un autre hypergraphe, à savoir $Hf(H, i)$, pour un certain i.

Définition 2.4.12. (méthode de décomposition)
Soit H un hypergraphe et $f(H, k)$ une fonction Subedge. On définit une méthode de décomposition $Mf(H)$ comme suit. Mf (H) est l'ensemble de tous les GHDs D' de H pour laquelle il existe un k tel que $k \leq |D'|$ et il existe une HD D de l'hypergraphe $HUf(H, K)$, de telle sorte que D' couvre $D3$. Cette méthode de décomposition de la forme Mf est appelé subeged-based.
Dans ce qui suit, nous allons présenter trois méthodes de subeged-based.

2.4.11 La méthode CHD (Component Hypertree)

cette méthode est poposer par Miklos [43].
Dans cette section nous donnons une Subedge spécifique définie une décomposition appelé "component hypertree decomposition".La définition de Subedge fonction fc du component hypertree decomposition est basée uniquement sur certaines caractéristiques structurales de l'hypergraphe. Nous avons besoin de quelques définitions techniques supplémentaires, avant que nous définissons la fonction Subedge de la méthode CHD.

Définition 2.4.13. Soit M un ensemble d'hyperarêtes d'un hypergraphe H, Nous définissons $prop(e, M)$, la partie appropriée d'une hyperarêtes e lié à M comme $prop(e, M) = e/ \cup e'\ M, e \neq e'e$.
par exemple, On a M contient trois hyperarêtes, $a(A, B, C), b(C, D, E, F), c(F, G, H)$. Alors,$prop(b, M) = (D, E)$.

Définition 2.4.14. Soit M un ensemble d'hyperarêtes de l'hypergraphe H et soit e une hyperarête dans M. Nous définissons internal $(e, M) = \{v|v \in vertices(e)$ et il n'existe pas $[vertices(M)] - composantC$ de H, tel que $v \in vertices(edges(C))\}$. Soit $M = e(A, B, C), f(C, D, E)\}$ donc $internal(e, M) = \{B\}$.

Définition 2.4.15. Soit H un hypergraphe, M un ensemble d'hyperarêtes de H, soit e une hyperarête de M et C une [vertices (M)] - composante. La fonction $elim(M, C, e)$ associe un ensemble contenant trois sous hyperarêtes à un triplet (M, C, e) :

1. (1) $e \cap vertices(edges(C))$,

2. (2) $prop(e, M) \cap vertices(edges(C))$,

3. (3) $internal(e, M)$.

On définit maintenant la fonction subedge de la méthode CHD :

Définition 2.4.16. Soit H un hypergraphe et $K(0 \leq K)$ un entier. La fonction Subedge est définit fc comme suit :
$fc(H, K) = e\, e' | M$ est un ensemble de $\leq K$ hyperarêtes de H, $e \in M$, D est une$[vertices(M)] - component$, et $e' \in elim(M, D, e)\}$.
La méthode Mfc sera désigner en tant que décomposition component hypertree (CHD).

2.4.11.1 La méthode ECHD (Extend Component Hypertree)

Cette méthode , proposée dans [43] , est une généralisation de la CHD. Avant de définir la fonction Subedge f E pour ECHD, on a besoin de quelques définitions techniques [31].

Définition 2.4.17. Soit M un ensemble d'hyperarêtes d'un hypergraphe H. Soit N un sous-ensemble d'hyperarêtes de M, $N \subseteq M$. On définit slice (N, M) comme suit : slice $(N, M) = v | \forall e \in N, v \in vertices(e)$ et $\forall e' \in (M\ n), v \notin vertices(e')$.
Ainsi, slice (N, M) est un ensemble de sommets en vertices(M), qui sont adjacentes aux hyperarêtes de N, mais à aucun autre hyperarêtes de M. De toute évidence, $vertices(M) = \cup NuMslice(N, M)$ [31].

Définition 2.4.18. Soit M un ensemble d'hyperarêtes de l'hypergraphe H et soit N un sous-ensemble de M. On définit internal $(N, M) = \{v | v \in slice(N, M)$ et il n'existe pas $[sommets(M)] - lecomposantC, telquev \in vertices(edges(C))\}$[31] .

Définition 2.4.19. Soit H un hypergraphe, soit M un ensemble d'hyperarêtes de H et soit C un [Vertices(M)] - composante. Soient $N_1, .., N_r (1 \leq r \leq 2|M| - 1)$ Des

sous-ensembles de M.
La fonction $lim(M, C, N_1, \ldots, N_r)$ associe un ensemble contenant les sous hyper-arêtes suivantes au tuple $(M, C, N_1, \ldots, N_r)$.
$\cup 1 \leq i \leq r(slice(N_i, M) \cap vertices(edges))$
$\cup 1 \leq i \leq rinternal(N_i, M)$[31].

Définition 2.4.20. Soit H un hypergraphe et soit k (0 . k) un entier. La fonction subedge fE est définie comme suit :
f E(H, k) = e e' | M est un ensemble de ? k hyperarêtes de l'hypergraphe H,
$e \in M$;
C est une [vertices (M)]-composante,
$1 \leq r \leq 2k - 1$
$N_1 \ldots .., N_r$, sont des ensembles d'hyperarêtes,
Pour chaque $i(1 \leq i \leq r); N_i \subseteq M$,
$e' \in lim(M, C, N_1, \ldots, N_r)$
Le procédé de décomposition MfE est appelée composante étendue hyper arborescente décomposition (ECHD) [31].

2.4.12 La méthode FHD (Fractional Hypertree Décomposition)

Cette méthode a été proposée dans [33].

Définition 2.4.21. (Le fractional edge cover)
Est une affectation de poids sur les bords de telle sorte que chaque sommet est couvert par poids total d'au moins 1[32].

Définition 2.4.22. (Décomposition FHD (Fractional Hypertree Décomposition))
La décomposition d'un hyperarbre fractionnelle d'un hypergraphe $H = (V, E)$ est un triple (T, B, g), où (T, B) est une décomposition arborescente de H et

$g = (gt)t \in T$est une famille des fonctions de E vers $[0, \ldots, \infty]$ tels que pour tout $t \in T$, il soutient que $\sum(ewithv \in e[gt(e) \geq 1]$ Pour $v \in Bt$ [33].

2.5 Classification des méthodes de décomposition

Dans cette partie, nous allons présenter la classification des méthodes de décomposition structurelle selon les critères élaborer par Gottlob et al dans [34] ; il ont comparer ces méthodes et étudier lesquelles des concepts de décomposition capturer les plus grandes classes de hypergraphes. Cohen et al. Montrent que toutes les méthodes de décomposition suivent le même schéma de définition, ils ne diffèrent que par une seule condition, caractéristique de la méthode particulière.
La hiérarchie des méthodes de décomposition structurelles est résumée dans la figure 2.3 où une flèche de A à B signifie qu'il existe des structures avec une largeur limitée définie par B, mais la largeur illimitée définie par A. Les flèches avec la ligne pointillée ont une sémantique déférente : s'il y a une forme de flèche de A à B avec une ligne pointillée alors la largeur des hypergraphes définies par la méthode B est inférieure ou égale à la largeur définie par la méthode A.
Dans cette hiérarchie, la méthode la plus générale est la décomposition FHD, puis la décomposition GHD qui est représenté avec un trait gras sur la figure, parce que si un hypergraphe H admet une décomposition de largeur inférieure ou égale à une constante k est un problème NP-complet. Comme il a été montré récemment dans [34].

2.6 Conclusion

Dans ce chapitre, nous avons présenté les principales techniques de décomposition des CSPs les plus connues qui se basent sur les propriétés topologiques du réseau de contrainte.

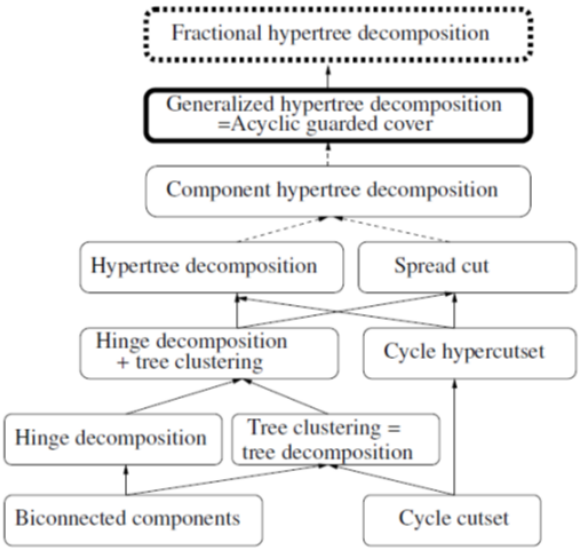

FIGURE 2.1 – Hiérarchie des méthodes de décomposition structurelles

Chapitre 3

résolution des CSPs en exploitant les décompositions structurelles

3.1 Introduction

Dans ce chapitre, nous présentans quelques méthodes pemettant d'exploiter les décompositions structurelles pour résoudre des CSP.

3.2 La résolution d'un CSP binaire acyclique

Pour la résolution des CSP binaire acyclique, un algorithme polynomial est proposé par rina decheter qui est le tree solving [39].
Le tree-solving accepte en entrée un réseau de contraintes acyclique (arbre) d'un CSP binaire. Chaque nœud de cet arbre (hormis la racine) possède un nœud père dirigé vers lui et peut avoir plusieurs nœuds fils vers lesquels il est, à son tour, dirigé. La première étape de cet algorithme est la construction d'un arbre enraciné dirigé, tout en spécifiant un ordre pour les nœuds de telle sorte que chaque nœud père doit apparaitre avant ses nœuds fils dans l'ordre (3). De l'étape 5 à 10, l'algorithme traite chaque arc et la contrainte lui associée, ceci des nœuds feuille au nœud racine, en vue de réalisé une arc consistance dirigée. Pour chaque arc dirigé de Xi vers Xj la

Algorithm 8 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 $d = X_1, \ldots .X_2$ tel qu'un nœud père précède toujours ses nœuds fils.
4 : Soit $Xp(i)$le nœud père du nœud Xi .
5 : pour $i = n$ à 1 faire
6 : Revise$(Xp(i), Xi)$
7 : **si** le domaine de $Xp(i)$ est vide alors
8 : Exit /*Pas de solution*/
9 : **finsi**
10 : **fin pour**.
Revise $(Xp(i), Xi)$
11 : pour chaque valeur x du $dom(Xp(i))$ faire
12 : si x n'a pas de support dans $dom(Xi)$
13 : suprimmer x du $dom(Xp(i))$
14 : **finsi**
15 : **fin pour**.

procédure Revise est appelé pour enlever les valeurs du domaine de Xj qui n'ont pas de support (valeur consistante) dans le domaine de Xi .Apres avoir traité tous les nœuds jusqu'à la racine, et si aucun domaine des variable n'est vide, un Backtrack free est utilisé pour trouver une solution.

3.3 Acyclic Solving

Nous avant vu qu'un CSP binaire acyclique peut être résolu en un temps polynomial en utilisant la notion de Join Tree. La propriété d'acyclicité peut être étendue pour les CSP n-aires en utilisant la notion de Join Tree.
Acyclic Solving est une reformulation de Tree Solving pour la résolution des problèmes CSP acycliques en générale, en tenant compte, pas seulement des variables, mais aussi des contraintes (voir l'algorithme 9)[39].

Algorithm 9 Acyclic Solving

1 : **Entrée :** un CSP acyclique $p = (X, D, C, R)$,$R = \{R_1, \ldots, R_2\}$. Un Join Tree T de P.
2 : **Sortie :** vérifier la consistance et générer une solution.
3 : $d = (R_1, \ldots, R_2)$ est un ordre tel que chaque relation apparait avant ses fils dans l'arbre T enraciné en R_1.
4 : **pour** $j = t$ à 1 faire 5 : **pour** chaque arête $(j, k), k \leq j$, dans l'arbre faire
6 : $RK \leftarrow SemiJoin(R_K, R_j)$
7 : si une relation vide est créé 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 R1
13 : pour chaque relation R_i, $i = 1$ à t faire
14 : prendre un tuple qui est consistant avec touts les relation qui la précèdent $(\mathrm{R}_1, \ldots, R_{i-1})$
15 : **fin pour**

Etant donné un CSP acyclique et son Join Tree, Acyclic solving est capable de générer une solution. Il traite le Join Tree du bas vers le haut (botom-up) (4 à11), et à chaque étape, il fait l'élimination des tuples de la relation courante qui ne peuvent pas participer à une jointure avec de cette dernière, en applique une semi jointure (6).
Si le CSP est inconsistant, une relation vide sera créée dans l'un des noued de Join Tree, et l'algorithme retourne problème inconsistant.(7)
Apres la fin de cette première phase (botom-up) (4 à 11), on aura un arc conssistance dirigée (DAC, Directed Arc Consistancy) du nœud racine vers les nœuds feuilles.
Dans la deuxième étape (12 à 14), Acyclic solving procède à la recherche d'une solution en parcourant le Join Tree de la racine vers les feuilles (top-down) (13) et à chaque nœud, il prend un tuple consistant avec ceux déjà pris.

3.4 Join Tree Processing (JTP)

Pour la résolution d'un CSP à partir d'une decomposition arboriscente(Tree decomposition) on applique l'étape (4) et (5) de l'algorithme Join Tree Clustering(JTC) proposé dans [16], ces deux étapes désgigentt l'algorithme Join Tree Processing(JTP).
Donc, étant donné un CSP et sa décomposition arborescente, l'algorithme Join Tree Processing procède la résolution de ce CSP (voir l'algorithme 3) comme suit :

Algorithm 10 Join Tree Processing (JTP)

1 : **Entré :** un ensemble de sous problem $p = P_1, \ldots, P_n$ avec $P_1 = \{R_{i1}, \ldots, R_{ij}\}$, Et un arbre $T = (P, E)$.
2 : **Sortie :** une solution du CSP s'il consistant
3 : **pour** $i = 1$à n faire
4 : Résoudre P_i et soit P_i' l'ensemble de solutions.
5 : Appliquer Acyclic solving sur le nouvel arbre T= R1,. . .,Rn'.
6 : **fin pour**

3. Chaque contrainte est placé dans le nœud de la décomposition arborescente qui contient les variables impliquer dans cette contrainte. chaque nœud représente un sous problème dont la résolution consiste à trouver toutes les affectations des variables consistantes avec les contraintes correspondantes
4. Chaque sous problème est résolu indépendamment (4). Par conséquent, le résultat est un Join Tree du problème acyclique équivalent au problème original.
5. On applique l'algorithme Acyclic Solving pour la résolution du problème entier(5).

3.5 Parallélisme et décomposition

Pour la plupart des décompositions, la résolution du problème se déroule en plusieurs phases :
1. construction des sous-problèmes,
2. résolution de chaque sous-problème,
3. résolution du problème global à partir des résultats des résolutions des sous-problèmes.
En règle générale, les phases de construction des sous-problèmes et de résolution du problème global restent séquentielles (ce qui n'est pas dramatique car elles ont souvent une complexité en temps polynomiale). Par contre, dans la majorité des cas, les sous-problèmes peuvent être résolus en parallèle, car ils sont indépendants les uns des autres.

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

difficile, ce qui conduirait à une inactivité d'une partie des solveurs.
Notons enfin qu'un algorithme parallèle pour la résolution de CSPs acycliques est proposé dans[ZM93]. Cet algorithme peut savéer utile si on utilise une méthode de résolution par décomposition comme le Tree-Clustering.

3.6 Conclusion

Dans ce chapitre nous avons présenté quelques unes des méthodes de résolution des CSPs par les méthode de décomposition, dans le cadre séquentiel et parallél.

Chapitre 4

Hybridation MAC-GHD

4.1 Introduction

Dans ce chapitre, nous présentons notre contribution relative à la résolution de CSPs basée sur la GHD.notre approche consiste à tirer profit à la fois des avantages des algorithmes énumératifs et de ceux de la décomposition GHD.

4.2 Maintaining Arc Consistency (MAC) basé sur une GHD (MAC-GHD)

Pour présenter cette contribution, nous avons besoin des définitions suivantes.

Définition 4.2.1. Soit ni un noeud de la décomposition GHD. Le sous-problème associé avec n_i est le CSP$< X_{n_i}, D_{n_i}, C_{n_i} >$où $X_{n_i} = X(ni)$, D_{n_i} est l'ensemble des domaines définis dans le CSP d'origine pour les variables qui sont dans X_{n_i} et $C_{n_i} = \lambda(ni)$. P_{n_i} dénote le sous-problème associé avec le noeud n_i et $sol(P_{n_i})$ désigne la solution de P_{n_i} .

Définition 4.2.2. Soient C_i et C_j deux contraintes , C_i et C_j sont dits compatibles

si $\forall X_i C_i[X_i] = C_j[X_i]$, sinon C_i et C_j sont dits incompatibles.

Remarque 4.2.1. Soit δ la liste des noeuds de la $GHD =< T, X, >$ et soit ni un noeud dans δ. On note $prec(n_i)$ le noeud précédant ni dans δet $succ(n_i)$ le noeud suivant n_i dans δ. Notons que généralement $prec(n_i)$ est différent de $Parent(n_i)$ qui est défini comme le parent du noeud n_i dans T.

Pour la résolution d'un CSP, nous avons fait une implémentation séquentielle de l'algorithme MAC$_G HD$.

4.2.1 L'implémentation séquentielle

B. **Le module MAC**
L'algorithme MAC-GHD calcule une seule solution pour le sous problème associé avec la racine de l'hypertree et tente d'étendre cette solution aux autres sous-problèmes induits par la GHD en respectant l'ordre en profondeur d'abord .
Si un sous-problème P_{n_i} n'a pas de solution, alors MAC-GHD effectue un retour-arrière vers P_{n_j} avec n_j est le noeud parent de n_i dans T, il calcule une autre solution pour P_{n_j} et recommence à partir de là.

MAC-GHD prend en entrée une $GHD =< T, X, \lambda >$ complète de l'hypergraphe de contraintes d'une instance CSP $P =< X, D, C >$. Les noeuds de T sont organisés dans une liste δ en respectant l'ordre en profondeur d'abord (pré-ordre).

Les sous-problèmes sont résolus séquentiellement par la procedure resoudre dans l'ordre δ. Si P_{n_i} a une (autre) solution alors la procédure Filter teste la satisfabilité des variables pour toute les contrainte au niveau des noeuds descendants de ni. Si toutes ces contraintes sont satisfaites, alors toutes les contraintes au niveau de chaque noeud fils de n_i sont filtrées et le sous-problème associé au noeud suivant dans δ est traité. Dans le cas contraire, une autre solution est calculée pour P_{n_i} si elle existe.

Si P_{n_i} n'a pas de (autre) solution, alors MAC-GHD appelle restaurer pour restaurer toute les valeur des variables éffacés par le processus de filtrage et retourner en arrière vers $Parent(n_i)$. MAC-GHD s'arrête dans deux cas :

1. Tous les sous-problèmes sont résolus, donc une solution globale pour le CSP est calculée (ligne 16) .

2. Le sous-problème associé avec la racine n'a pas de (d'autre) solution, donc le CSP est insatisfable.

Algorithm 11 MAC-GHD

Input : une GHD $=< T, X, \lambda >$ complète associée à un CSP P
Output : une solution A de P si elle existe
1 : $\delta \leftarrow (n_1, n_2, \ldots, n_e)$ /*δ pré-ordre des noeuds de T avec n_1 sa racine */
2 : $n_i =\leftarrow n_1$
3 : while $n_i \neq \emptyset$ do
4 : $sol(P_{n_i)\leftarrow resoudre(P_{n_i})}$
5 : if $sol(P_{n_i)=\emptyset}$ then
6 : if $n_i = n_1$ then
7 : $A \leftarrow \emptyset$;
8 : exit /* P est insatisfable */
9 : else
10 : $n_i = Parent(n_i)$
11 : restaurer n_i
12 : end if
13 : else
14 : Filter(n_i)
15 : end if
16 : end while
17 :$A \leftarrow \bowtie_{i=0,i=e} sol(P_{n_i})$

Les principales procédures et fonctions de MAC-GHD sont expliquées ci-après.

procedure resoudre la fonction resoudre permet de calculer une solution pour un sous problème Pni associé au nœud ni en utilisant l'algorithme MAC, a chaque instanciation d'une nouvelle variable on vérifier sa satisfiabilité pour toute les contrainte existant (c'est a dire si la variable se trouve dans une autre contrainte elle doivent avoir la même valeur si ce n'est pas le cas un retour en arrière est effectuer), a chaque fois que cette fonction est appelée, elle retourne une autre solution.

Procedure Filter Quand une solution de sous problème Pni associée avec le nœud ni est calculée , la procédure Filter est appelée pour tester la satisfiabilité des variables au niveau des nœuds du sous – arbre enraciné au niveau de ni. Si toutes ces variables sont satisfaites elle consiste a suprimer toute les valeurs incompatible avec sol (Pni). Cependant , les valeurs effacé peuvent étre restaurés en cas de retour-arriére . A cet effet, toute les valeur des variables sont étiquetés. L'étiquette d'une valeur peut étre soit '0' sinifiant que t n'est pas effacé , soit '1' signifiant que t est effacé a cause de son incompatibilité avec sol (P nx) . Initialement l'étiquette de chaque valeur est '0'.
Quand le sous- probléme Pni n'as pas de solution un retour – arrière est effectuer du nœud ni Vers son pere (ni) et pour cela on a utiliser la fonction ordonner qui permet d'ordonner l'arbre binaire en une liste doublement chinée en respectant l'ordre de profondeur d'abord , quand un retour arrière est effectuer on apelle la fonction restaurer quand va expliquer juste après.

Algorithm 12 Filtrer

Procedure Filtrer(in ni : node)
1 : pour chaque $variable_i \in sol(p)$ faire
2 : pour chaque $valeur_i \in variable_i$ faire
3 : si $valeur_i$ incompatible avec $sol(p)$ alors
4 : $valeur_i \leftarrow etat = 1$ 5 : fin si.
6 : fin pour.
7 : fin pour.

procédure restaurer la procédure retaurer est appelée pour restaurer toute

les valeurs supprimer à partir de chaque variable du nœud ni , cette procedure consiste simplement à assigner l'étiquette '0' à chaque variable du nœud ni ayant l'étiquette '1'.

Algorithm 13 Restaurer

Procedure Restaurer(in ni : node)
1 : pour chaque $variable_i \in n_i$ faire
2 : pour chaque $valeur_i \in variable_i$ faire
3 : $valeur_i \leftarrow etat = 0$ 4 : fin pour.
5 : fin pour.

4.3 test et résultat

Nous allons présenter quelques tests et résultat éfféctués sur quelques problémes .

exemple 1 :

soit le probleme suivant :

$C = \{C3, C5, C6, C7, C8, C9\}$.
$X = \{V27, V56, V76, V62, V57, V92, V69, V49, V70, V34, V93, V88, V57, V70, V32\}$.
$D = \{D27, D56, D76, D62, D57, D92, D69, D49, D70, D34, D93, D88, D57, D70, D32\}$.
$R = \{R0, R1\}$.
$R0 = \{001|011|100|101|110\}$.
$R1 = \{000|001|010|011|100|101|110\}$.

la figure 4.1 suivante décrit notre probléme aprés une decomposition .

l'ensemble de solutions pour se CSP est {00100000000000}. qui est présenter dans la fiqure 4.2.

FIGURE 4.1 – Présentation de l'ensemble de solution

4.4 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthode appelée MAC-GHD pour résoudre les instances de CSP non booléennes en utilisant la Generalized Hypertree Décomposition(GHD). MAC-GHD combine un algorithme énumératif de type MAC qui est efficace en mémoire et une GHD qui a de bonnes bornes de complexité théorique.

Chapitre 5

La résolution parallèle des problème de satisfaction de contrainte via une GHD

5.1 Introduction

Dans ce chapitre nous présentons notre proposition de parallélisation de la technique de la résolution basée décomposition.

5.2 L'environnement de développement

5.2.1 Langage de programmation

Nous avons développé notre application en utilisant le langage C++ sous linux après avoir configurer la bibliothèque MPI.

5.2.2 MPI

5.2.2.1 Qu'est-ce que c'est MPI

MPI (Message Passing Interface) est une spécification pour les développeurs et les utilisateurs des bibliothèques de passage de messages. En soi, ce n'est pas une bibliothèque - mais plutôt la spécification de ce qu'une telle bibliothèque devrait être.
MPI porte principalement sur le modèle de programmation parallèle passage de messages : les données sont déplacées de l'espace d'adressage d'un processus visant à celle d'un autre processus à travers des opérations de coopération sur chaque processus.
Simplement dit, l'objectif de la Passing Interface Message est de fournir un standard largement utilisé pour écrire des programmes de passage de messages [40]. L'interface se veut :

- ⋆ Pratique.
- ⋆ Portable.
- ⋆ Efficace.
- ⋆ Flexible.

MPI fournit également les fonctionnalités suivantes (non exhaustif) :

- Environnement d'exécution.
- Types de données dérivés .
- Communicateurs et topologies.
- Gestion dynamique de processus.

– Entrées-sorties parallèles.

– Interface de profilage.

5.2.2.2 Raisons de l'utilisation MPI

- **Normalisation :** MPI est la seule bibliothèque de passage de message qui peut être considéré comme une norme. Il est soutenu sur pratiquement toutes les plateformes HPC. Pratiquement, il a remplacé toutes les bibliothèques de passage de messages précédents.

- **Portabilité :** Il y a peu ou pas besoin de modifier le code source lorsque vous portez votre application à une autre plate-forme que les supports (et est compatible avec) le standard MPI.

- **Occasions de rendement :** implémentations de fournisseurs devraient être en mesure d'exploiter les fonctionnalités matérielles indigènes pour optimiser les performances.

- **Fonctionnalité :** Il y a plus de 440 routines définies dans MPI - 3, qui comprend la majorité des personnes dans MPI - 2 et MPI - 1.

5.2.2.3 Type de communication de MPI :

a) **Communication point à point :**
Une communication point à point a lieu entre deux processus, l'un est appelé émetteur, l'autre récepteur, identifiés par leur rang. Il s'agit de la communication de base de MPI et les opérations effectuées sont un VHQG et un UHFHLYH. Il existe différentes manières d'effectuer la communication. De

manière générale, les paramètres nécessaires à sa construction sont [42] :

- les rangs des processus émetteur et récepteur,

- l'étiquette du message,

- le nom du communicateur qui définit le contexte de la communication,

Le type des données échangées, les données.

1. **Communications collectives :**
   Les communications collectives permettent de réaliser des communications impliquant plusieurs processus. Elles peuvent toujours être simulées par un ensemble de communications point à point (mais peuvent être fortement optimisées) avec éventuellement desopérations de réduction. Une communication collective implique l'ensemble des processus du communicateur utilisé.
   Jusqu'à MPI 2.2, il s'agissait d'appels bloquants (c-à-d qu'un processus ne sort del'appel que lorsque sa participation à la communication est terminée). La normeMPI 3.0 introduit les appels non bloquants pour les communications collectives.
   Les communications collectives n'impliquent pas de synchronisation globale ($sauf MPI\_Barrier$)et n'en nécessitent pas.
   Elles n'interfèrent jamais avec les communications point à point[43].

2. **communications mémoire à mémoire :**
   Les communications mémoire à mémoire (ou RMA pour Remote Memory Access ou one sided communications) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement. Le processus cible n'intervient donc pas lors du transfert [43].

5.2.2.4 les avantages et les inconvénients de MPI

1. **Avantages**

   - Permet de mettre en place plus efficacement certains algorithmes.
   - Plus performant que les communications point à point sur certaines machines (utilisation de matériels spécialisés tels que coprocesseur, mémoire spécialisée...).
   - Possibilité pour l'implémentation de regrouper plusieurs opérations.

2. **Inconvénients**

   - La gestion des synchronisations est délicate.
   - Complexité et risques d'erreurs élevés.
   - Pour les synchronisations cible passive, obligation d'allouer la mémoire avec MPI_Alloc_mem qui ne respecte pas la norme Fortran (utilisation de pointeurs Cray non supportés par certains compilateurs).
   - Moins performant que les communications point à point sur certaines machines.

5.2.2.5 Compilation et exécution d'un programme MPI

1. **Implémentations**
   Les implémentations classiques sont en C/C++ et FORTRAN, mais il existe aussi des implémentations en Python, OCaml, Perl et Java. L'implémentation

initiale du standard MPI 1.x était MPICH, aujourd'hui devenue MPICH 2 en supportant le standard MPI-2 ; c'est un logiciel libre. On trouve aussi sous licence libre OpenMPI. À côté de ces implémentations généralistes, des implémentations optimisées ont été faites par certains constructeurs [42].

5.3 L'implémentation parallèle

Nous avons essayé de développer cette partie , mais on n'a pas réussit à l'implimenter mais elle consiste à l'utilisation de l'algorithme MAC et l'utilisation des fonctions de la bibliotheque MPI.
La résolution parallèle du problème se déroule en plusieurs phases :

1. résolution de chaque sous-problème,

2. résolution du problème global à partir des résultats des résolutions des sous-problèmes.

Tout au début, il faudrait que chaque processus exécute les instructions initialisant la bibliothèque MPI, ensuite si le rang du processus est a 0, il reçoit le problème et le diffuse vers tout les processus. après avoir reçu le csp, chaque processus calcule les branches selon son identité (chaque processus utilise le MAC pour trouver une solution) et s'il trouve de résultat il les fait envoyer au processus p0 qui est en attente et en fin il affiche le résultat qu'il a reçu.

A. les fonction ajoutées au séquentiel

- Initialisation du MPI
  chaque proccesus doit initialiser la bibiothèque MPI en executant les instruction

suivante :

- $MPI\_Init(argc, argv)$;
- $MPI\_Comm\_rank(MPI\_COMM\_WORLD, my\_rank)$;
- $MPI\_Comm\_size(MPI\_COMM\_WORLD, p)$;

- $MPI\_Reseive()$: permet de recevoire un message envoyé par un processus ;
- $MPI\_Send()$: permet d'envoyer un message d'un processus à un autre ;
- $MPI\_Bcast()$: permet de diffuser un msg, nous l'avant utulisé pour diffuser le CSP par le processus () vers les autre processus.
- **la répartition initiale :** pour la realisation de cette partie on a utiliser les fonctions suivantes :
  - $calcul\_niveau()$
    nous avons utulisé la technique modulo pour repartir initialement les branche de l'arbre, et pour qu'aucun processus ne reste inactif il faut que le nombre de branche soit supérieur ou égale au nombre de processus , la fonction calcul niveau permet d'assurer ces condition.
  - **distribuer_branche ()**
    procedure permet à chaque processus de calculer les branche à traiter .
- **l'équilibrage de la charge :**
  Son principe est : le processus qui termine son travail envoie un message au processus p0, qui se charge de trouver un processeur capable de satisfaire la demande d'équilibrage, il va déffuser un message à tous les processus, puis il

choisit l'un d'entre eux, il lui envois l'identité du processeur, a qui le processeur surchargé va partager sn travail, en fin le processeur demandeur retourne le résultat au processeur donneur.

5.4 Conclusion

Nous avons étudier, dans ce chapitre, l'implimentation paralléle de l'un des algorithmes utilisé pour la resolution des probléme de satisfaction de contraintes qui est l'algorithme Maintaining Arc Consistency (MAC).

Conclusion et perspectives

Notre contribution se situe dans le domaine des Problèmes de Satisfaction de Contrainte (CSP), qui est une partie de l'Intelligence Artificielle. Le domaine CSP offre une multitude de méthodes qui permettent de résoudre ou réduire des problèmes et qui ont été développé depuis 1970. Plusieurs algorithme ont été proposé, mais nous nous sommes intéressé à tirer profit à la fois des avantages des algorithmes énumératifs et de ceux de la décomposition GHD.

Dans ce travail nous avons proposé une solution à ce problème ausi bien dans le cadre parallèle que dans dans le cadre sequentiel.

Dans le chapitre 1, nous avons définit certains algorithmes pour la résolution séquentiel et parallèle des CSP, en se basant sur les algorithmes consistants qui permets de réduire l'espace de recherche à exploré pour simplifier les instances avant ou pendant la recherche d'une solution.
Dans le chapitre 4 nous avons proposer une solution au CSP on utilisant la combinaison des deux méthodes (la decomposition et maintaining arc consistant(MAC)).
Dans le chapitre 5, nous avons proposé une parallelisation de cet algorithme permettant ainsi d'optimiser cette complexité.

Un point intéressant pour les futurs travaux de recherche est l'expérimentation de cet algorithme sur des machines parallèle, afin de confirmer leur efficacité dans un environnement du parallèlisme réel.
un autre point est l'optimisation de cet algorithme par combinaison d'autres

heuristiques de résolution et par augmentation du degré du parallèlisme.

Bibliographie

[1] Tony LAMBERT, Hybridation de méthodes complètes et incomplètes pour la résolution de CSP, université Nantes 27 Octobre 2006.

[2] Jean-Marie Lagniez, Satisfiabilité propositionnelle et raisonnement par contraintes : modèles et algorithmes, l'Université d'Artois, 6 Décembre 2011.

[3] Christine Solnon ,Résolution de problèmes combinatoires et optimisation par colonies de fourmis.

[4] Blaid SAAD, Intégration des problèmes de satisfaction de contrainte distribués et sécurisés dans les systèmes d'aide à la décision à base de connaissances, Université de Paul Verlaine 10 décembre.

[5]. Maria Malek, APPLICATIONS EN PROLOG, EISTI, Deuxième année.

[6] David MARTINEZ, Résolution interactive des problèmes de satisfaction de contraintes, 1998 ,école nationale supérieure de l'aéronautique et de l'espace de Toulouse.

[7] Patrice Perny, Introduction aux problèmes de satisfaction de contraintes

(CSP), Université Paris 6.

[8] David Savourey, Programmation Par Contraintes Cours 2 - Arc-Consistance et autres amusettes, CNRS, Ecole Polytechnique.

[9] Maria Malek, Programmation par Contraintes Les algorithmes Arc-Consistances, Options GL, ISICO IdSI EISTI.

[10] Yacine Zemali, programation par contrainte :consistances local et filtrage, École Nationale Supérieure d'Ingénieurs de Bourges, 2012 – 2013.

[11] Christoph Durr ; Programmation par contraintes et programmation mathématique, Ecole polytechnique, 2015.

[12] Christian Bessiere, Constraint Propagation, Technical Report LIRMM 06020. CNRS/University of Montpellier, March 2006.

[13] David Défossez, coloration d'hypergraphe et clique-coloration, Hal 2 nov 2006.

[14] Didier Muller, Introduction à la théorie des graphes, CRM 2012.

[15] G.Montcouquiol, théorie des graphes ,2006 – 2007.

[16] Kamel Amroun, Décomposions d'hypergraphes pour la résolution des problèmes de satisfaction de contraintes(CSP) ,$23nov2014$.

[17] Gianluigi Greco, Franco Scarcello, Tree projections and structural decomposition methods Minimality and game-theoretic characterization ,university of

Calabria ,87036,Rende,Italy,11Dec2012.

[18] R.Dechter, Constraint Networks. In Encyclopedia of Artificial Intelligence, second edition, Wiley and Sons, pp. $276 - 285, 1992$.

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

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

[21] N. Robertson and P.D. Seymour. Graph Minors II. Algorithmic aspects of tree width.

Journal of Algorithms, $7 : 309 - 322, 1986$.

[22] P.Jégou and C.Terrioux, Hybrid backtracking bounded by tree-decomposition of constraint networks, Arti$_c$$ialIntelligence$,146 (2003).

[23] Georg Gottlob, Marti Hutle, Franz Wotawa, Combining hypertree, bicomp, and hinge decomposition, conference paper, janvier, 2002.

[24] Sathiamoorthy Subbarayan,Integrating CSP decomposition techniques and BDDs for compiling configuration problems, IT University of Copenhagen , Denmark.

[25] Kamal Amroun, Zineb Habbas, HD DBT : hypertree décomposition pour la résolution des problèmes de satisfaction de contraintes basée sur un dual backtracking, Hal, juin 2010.

[26] Georg Gottlob, Marko Samer, Heuristic Methods for Hypertree decompositions, DBAI TECHNICAL REPORT, 2005.

[27] G.Gottlob, N.Leone, F.Scarcello, Hypertree decompositions : A survey. In proceedings of MFCS ,2001.

[28] David Cohen, Peter Jeavons and Marc Gyssens, A unified theory of structural tractability for constraint satisfaction and spread cut decomposition.

[29] Zoltan Miklos. Understanding Tractable Decompositions for Constraint Satisfaction. university of oxford.

[30] Martin Grohe and Daniel Marx. Constraint Solving via Fractional Edge Covers. Humboldt-Universitat zu Berlin Institut fur Informatik Symposium on Discrete Algorithms (SODA) 2006 January 22, 2006.

[31] Martin Grohe. The structure of tractable constraint satisfaction problems. Institut fur Informatik, Humboldt Universitat Unter den Linden 6, 10099 Berlin, Germany.

[32] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. Artifical Intelligence.

[33] David Cohen, Marc Gyssens, and Peter Jeavons. A unifying theory of structural decompostions for the constraint satisfaction problems. In Complexity of Constraints, number 06401 in Dagstuhl Seminar Proceedings, 2006.

[34] Georg Gottlob, Zoltan Miklos, and Thomas Schwentick. Generalized hypertree decompositions : NP-hardness and tractable variants. Copyright °c 2007

by the authors.

[35] Loïc Gouarin, Violaine Louvet, Laurent Series, Formation en calcul scientifique-LIEM2I, Introduction en calcul parallèle, avril 2012.

[36] Erven Rohou, Etude du parallélisme d'instructions, Irisa, 1993 – 1994.

[37] Cyril TERRIOUX, thèse pour obtenir le titre de docteur, Approches structurelles et coopératives pour la résolution des problèmes de satisfaction de contraintes, école doctorale de mathématique er d'informatique de Marseille, 10*dcembre*2002.

[38] Conception d'un système à haute performance - Le calcul parallèle, CETMEF 2004.

[39] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions : A survey. In Proceedings of MFCS 01, pages 37˘57, 2001.

[40] Message Passing Interface (MPI). Isabelle Dupays Marie Flé Jérémie GAIDAMOUR Dimitri LECAS

[41] MPI Message Passing Interface. Programmation hybride MPI-OpenMPI. CNRS — IDRIS Version 2.1¯30mars 2015.

[42]Frédéric Desprez INRIA LIP ENS Lyon Equipe Avalon INRIA F. Desprez - UE Parallélisme (2013 – 2014).

[43] Z. Miklos, Understanding tracable decompositions for constrain satisfaction, Ph.D. thesis, University of Oxford, 2008.

RÉSUMÉ

La résolution des CSPs fait généralement appel à des recherches arborescentes exploitant des améliorations du backtracking , De telles méthodes obtiennent souvent des résultats satisfaisants en pratique. Cependant, leur complexité en temps est bornée par la taille de l'espace de recherche, qui est exponentielle. A cette effet, plusieurs algorithme on été proposées pour la résolution efficace de ces problème parmi les quelles les algorithmes énumératifs. L'algorithme Maintaining Arc Consistency (MAC) est l'un des algorithmes qui donne toujours de meilleurs résultats en pratique. Notre approche consiste à tirer profit à la fois des avantages des algorithmes énumératifs et de ceux de la décomposition GHD.

Mots clés : Probléme de satisfaction de contraintes (CSP), Maintaining Arc Consistency (MAC), Résolution CSP , Parallélisme

ABSTRACT

Solving CSPs generally uses of tree research operating improvements backtracking . Such methods often achieve satisfactory results in practice. However, the time complexity is bounded by the size of the search space, which is exponential. To this effect, several algorithms have been proposed for the effective resolution of these problems among which the enumerative algorithms. The algorithm Maintaining Arc Consistency (MAC) is a algorithm that always works better in practice. Our approach is to take advantage of both benefits enumerative algorithms and those of GHD decomposition.

Key words : Constrainte Satisfaction Problems (CSP), Maintaining Arc Consistency (MAC) , Solving CSP , Parallelism.