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

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

Ecole Doctorale d'informatique

Mémoire de Magister en Informatique

Option

Réseau et systèmes distribués

Thème

Exploitation de l'Hypertree decomposition pour la résolution des problèmes de satisfaction de contraintes

Présenté par :

AIT AMOKHTAR AbdelMalek

Dirigé par :

MCF-HDR Zineb HABBAS

Devant le jury composé de :

Président :	Moussa Kerkar, Professeur, Université A.Mira de Béjaïa, Algérie
Rapporteur :	Zineb Habbas, MCF-HDR, IUT de Metz, France
Examinateurs:	Mohamed Ahmed-Nacer, Professeur, USTHB, Alger, Algérie
	Mouloud Koudil, Professeur, INI, Alger, Algérie
Invité :	Kamal Amroun, Chargé de cours, Université A.Mira de Béjaïa, Algérie

Avril, 2008

Tables des matières

Introduction……..……- 1 -

Chapitre I : Les problèmes de satisfaction de contrainte…….. - 4 –

1. Définitions de base : _____ - 5 -

2. Résolution des CSP : différentes méthodes _____ - 6 -

2.1. Réduction de l'espace de recherche:_____- 7 -
2.1.1. L'Algorithme AC3 _____- 8 -

2.2. Résolution d'un CSP_____- 9 -
2.2.1. Algorithme Backtrack (BT): _____- 9 -
2.2.2. Algorithme Back Jumping (BJ)_____- 10 -
2.2.3. Algorithme Conflict Directed Back jumping (CBJ) _____- 10 -
2.2.4. Algorithme Forward Checking (FC): _____- 10 -
2.2.5. Algorithme Maintenance de consistance d'arc (MAC) :_____- 11 -

2.3. Heuristiques pour l'amélioration de la résolution des CSP: _____- 12 -
2.3.1. Heuristique sur le choix des variables :_____- 12 -
2.3.2. Heuristiques sur le choix des valeurs _____- 12 -

3. Résolution des CSP n-aires : _____ - 13 -

3.1. Transformation de CSP n-aires en CSP binaires : _____- 13 -

3.2. Résolution direct d'un CSP n-aires : _____- 13 -
3.2.1. La consistance d'arc dans les CSP n-aires :_____- 13 -
3.2.2. Algorithme GAC2001 (GAC3.1) : _____- 14 -
3.2.3 Algorithme nFC _____- 15 -

4. Vers une plateforme multi approches de résolution des CSP :_____ - 15 -

4.1. Notre solveur CSP par des méthodes énumératives :_____- 16 -

4.2. Résultats Expérimentaux : _____- 19 -

Conclusion _____ - 21 -

Chapitre II: Les méthodes de décomposition structurelle……. - 22 -

1. Tractabilité _____ - 23 -

1.1. Tractabilité due à la structure_____- 23 -

1.2. Tractabilité due aux relations_____- 23 -

1.3. Structures d'un CSP_____- 23 -

2. Méthodes de décomposition structurelle _____ - 24 -

2.1. Méthode Cycle cutset_____- 24 -

2.2. Méthode Biconnected-component _____- 26 -

2.3. Méthode Tree-Clustering _____- 27 -

2.4. Méthode Hinge decomposition _____- 28 -

2.5. La méthode Hinge decomposition + Tree clustering _____- 30 -

2.6. Hypertree decomposition: _____- 31 -

2.7. Spread cut :_____- 31 -

3. Comparaison _____ - 34 -
Conclusion: _____ - 35 -

Chapitre III: Hypertree decomposition……..- 37 -

1. L'hypertree _____ - 38 -
2. L'hypertree decomposition _____ - 38 -
3. Forme normale de l'hypertree decomposition : _____ - 40 -
4. Calcul de l'hypertree decomposition _____ - 41 -
   4.1. Méthodes exactes : _____ - 41 -
      4.1.1. L'algorithme k-decomp _____ - 41 -
      4.1.2. L'algorithme Opt-k-decomp _____ - 42 -
      4.1.3. L'algorithme Red-k-decomp _____ - 44 -
      4.1.4. L'algorithme Det-k-decomp _____ - 45 -
   4.2. Méthodes heuristiques _____ - 47 -
      4.2.1. L'algorithme Bucket Elimination (BE) _____ - 48 -
      4.2.2. L'algorithme Dual Bucket Elimination (DBE): _____ - 49 -
      4.2.3. Les Algorithmes génétiques : _____ - 49 -

Conclusion _____ - 50 -

Chapitre IV: Construct & Reduce……..…….- 51 -

1. Notre heuristique: Construct & Reduce: _____ - 52 -
2. Extraction des Hypertrees : _____ - 53 -
3. La fusion des hypertrees : _____ - 55 -
4. La réduction de la largeur de l'hypertree : _____ - 57 -
5. Complexité de l'approche : _____ - 59 -
6. Résultats expérimentaux _____ - 60 -

Conclusion _____ - 63 -

Chapitre V: Résolution des CSP acycliques…….. - 64 -

1. Résolution d'un CSP binaire acyclique : _____ - 65 -
2. Résolution d'un CSP n-aire acyclique : _____ - 66 -
3. Résolution d'une Tree décomposition : _____ - 67 -
   3.1. Construction de CSP acyclique équivalent : _____ - 68 -
   3.2. Résolution de CSP acyclique équivalent : _____ - 68 -
4. Résolution d'une Hypertree decomposition : _____ - 68 -
   4.1. Compléter l'hypertree decomposition : _____ - 68 -
   4.2. Construction de CSP acyclique équivalent : _____ - 69 -
   4.3. Résolution de CSP acyclique : _____ - 69 -
5. Etudes de cas : _____ - 69 -
   5.1. Le problème Renault : _____ - 69 -
      5.1.1. Décomposition : _____ - 70 -

5.1.2. Résolution : _____- 70 -
5.1. La famille de problèmes Dubois : _____- 71 -
5.1.1. Décomposition :_____- 71 -
5.1.2. Résolution : _____- 72 -
Conclusion : _____ - 73 -

Conclusion et perspectives……..……..- 74 -

Bibliographie …….. - 77 -

Liste des figures

Fig.I.1. Différentes approches pour résoudre les CSPs …….. -14-
Fig.I.2. Solveur de CSP…… -23-
Fig.I.3. Diagramme de classe complet de solveur de CSP définit en extension…….. -23-
Fig.I.4. Diagramme de classe complet de solveur de CSP définit en intension …….. -25-

Fig.II.1. (a) hypergraphe **(b)** graphe primal **(c)** graphe dual ……... -31-
Fig.II.2. Graphe d'un CSP avant et après l'instanciation de X5 …… -32-
Fig.II.3. Algorithme général de résolution avec la méthode coupe cycle ……... -32-
Fig.II.4. Un graphe est sa biconnected decomposition … …….. -33-
Fig.II.5. Etapes de l'algorithme tree clustering …….. -35-
Fig.II.6. Etape de décomposition de P …… -35-
Fig.II.7. Un hypergraphe et son hinge tree …… -37-
Fig.II.8. Hypergraphe de P et sa Spread Cut ……... -41-
Fig.II.9. Comparaison entre les méthodes de décomposition …… -42-

Fig.III.1. L'hypergraphe H …… -46-
Fig.III.2. (a) l'hypertree decomposition généralisée **(b)** l'hypertree decomposition de H … -47-
Fig.III.3. Résumer de l'algorithme Det-k-decomp …… -54-

Fig.IV.1. Les hypertrees extraites de problème de Schure …… -61-
Fig.IV.2. L'Hypertree obtenue de la contraction …… -63-
Fig.IV.3. L'Hypertree obtenue après la réduction …….. -66-
Fig.IV.4. Diagramme de classe complet …… -68-

Fig.V.1. Un arbre dirigé enraciné …… -73-
Fig.V.2. Hypergraphe, graphe primal, graphe dual et join graphe …… -74-
Fig.V.3. Temps de résolution avec l'approche par décomposition …… -79-
Fig.V.4. Temps de résolution avec l'approche énumérative …….. -79-

Liste des tableaux

Tab.I.1. Résultats de la résolution de problème des n-reines ……. -27-
Tab.I.2. Résultats de la résolution de problème de Schur …….. -28-
Tab.IV.1. Comparaison BE, DBE et Construct & Reduce …… -69-

Remerciements

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

Je remercie Messieurs les professeurs Kerkar, Ahmed-Nacer et Koudil de m'avoir honoré en acceptant d'examiner mon travail.

Je ne remercierai jamais assez mon encadreur Mme Habbas de m'avoir orienté et d'avoir toujours été là quand j'en avais le plus besoin, ainsi que Mr Amroun pour ses conseils et critiques.

Mes premières pensées vont à ma famille, qui m'a toujours soutenu et encouragé.

En fin, je remercie tous mes collègues de l'école doctorale et notamment Foudil et Balzak pour avoir lu et corrigé mon mémoire.

Introduction

La représentation des problèmes que nous sommes amenés à résoudre a depuis toujours été une des disciplines les plus importantes en l'intelligence artificielle. Plusieurs formalismes accompagnés de leurs méthodes de résolution ont été proposés. Certains d'entre eux sont spécifiques à certaines classes de problèmes alors que d'autres se veulent être généraux pour à la fois représenter et résoudre des problèmes à priori totalement différents. C'est dans cette deuxième catégorie que s'inscrivent les problèmes de satisfaction de contraintes (CSP pour Constraints Satisfaction Problems).

Les Problèmes de satisfaction de contraintes sont un cadre générique introduits par Montanari **[1]** dans les années 70 et qui permettent de formaliser un très grand nombre de problèmes tels que les problèmes de coloration de graphe, les problèmes d'ordonnancement, les problèmes de placement, …etc. Un CSP est défini par un ensemble de variables et de contraintes reliant les variables entre elles. Les variables prennent leurs valeurs dans un ensemble fini de valeurs ou domaine. Résoudre un CSP consiste à associer une valeur à chaque variable de telle façon que toutes les contraintes soient satisfaites.

Plusieurs variantes du formalisme CSP ont été proposées pour répondre à des besoins spécifiques de certains problèmes. On recense ainsi les formalismes VCSP et Max-CSP qui modélisent des problèmes dont les contraintes n'ont pas toutes la même importance, et le formalisme CSOP qui englobe les problèmes dont on a des préférences sur la solution à trouver. Nous nous intéressons ici au formalisme de base des CSP.

L'algorithme de base utilisé pour résoudre les CSP est l'algorithme Backtrack. C'est un algorithme très naïf qui parcourt dans le pire des cas l'espace complet des solutions

possibles ou espace de recherche. Cette technique a été le noyau central autour duquel plusieurs travaux de recherche ont été menés pour améliorer la résolution des CSP en pratique. Ces travaux basés sur diverses notions de consistance, les notions de filtrage et de propagation de contraintes, d'heuristiques d'ordre d'instanciation de variables et de valeurs, ont donné naissance à une large famille d'algorithmes (BJ, CBJ, FC, MAC, FC-MRV , …etc).

Mais les CSP étant NP-complets il n'existe pas d'algorithme exact capable de résoudre de façon efficace ces problèmes dès lors qu'on s'intéresse à des problèmes de grande taille qu'ils soient académiques ou réels.

Il existe par contre des sous classes de CSP dites « tractables » ou traitables. La recherche de certaines de ces classes a motivé le développement de nouvelles approches de résolution. Une classe de sous CSP intéressante est celle des CSP acycliques pour laquelle on connaît des algorithmes de résolution polynomiaux : un CSP dont la structure est un arbre peut être résolu par un algorithme sans backtack. Plusieurs travaux ont depuis porté sur les décompositions de CSP. Nous nous sommes intéressés ici aux techniques de décomposition structurelles dont le but est de transformer un CSP dont la structure est en général un hypergraphe en un CSP, dont la structure est un arbre, ayant le même ensemble de solutions. Plusieurs méthodes de décompositions structurelles se sont vues succéder ces dernières décennies. Chaque une d'elle propose une mesure de cyclicité qui lui est propre appelée largeur de la décomposition.

Parmi toutes ces méthodes de décomposition, l'hypertree decomposition **[24**] est la plus prometteuse et ce grâce à sa définition à la fois concise et suffisamment générale pour être applicable à une très large palette de CSP. De plus, il existe plusieurs algorithmes permettant de la construire. Les premières méthodes proposées pour le calcul de l'hypertree decomposition, appelées méthodes exactes, avaient comme objectif de rechercher la décomposition de largeur optimale. Ce qui induisait un coût en calcul et en occupation mémoire si importants qu'elle devenait inexploitable en pratique. C'est pour cette raison que la recherche s'est tournée vers des algorithmes basés sur des heuristiques.

Dans ce mémoire nous nous sommes intéressés à la résolution des problèmes de satisfaction de contraintes par décomposition structurelle. Plus concrètement nous proposons une nouvelle heuristique pour un calcul efficace de l'hypertree decomposition nommée Construct & Reduce.

Ce mémoire s'articule autour de cinq chapitres définis succinctement comme suit :

Le chapitre 1 présente le formalisme CSP, les différentes méthodes utilisées pour leur résolution, ainsi qu'une étude empirique de certains algorithmes de résolution faisant partie du solveur que nous avons développé.

Le chapitre 2 est consacré aux méthodes de décomposition structurelle.

Le chapitre 3 poursuit le travail commencé au chapitre 2 en présentant d'une manière détaillée la méthode hypertree decomposition.

Le chapitre 4 est consacré à la méthode Construct & Reduce une nouvelle heuristique que nous proposons pour le calcul de l'hypertree decomposition.

Le chapitre 5 de ce mémoire montre la faisabilité de notre approche, en montrant comment notre méthode peut être intégrée dans une démarche de résolution d'un CSP, et d'une manière plus générale comment le formalisme CSP peut être exploité pour la résolution des problèmes du monde réel.

CHAPITRE I

Les Problèmes de Satisfaction de Contraintes

Les problèmes de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) sont au coeur de nombreuses applications en Intelligence Artificielle et en Recherche Opérationnelle. Un CSP est un formalisme à la fois simple et puissant permettant de représenter et de résoudre un grand nombre de problèmes. Un CSP se définit comme un ensemble de contraintes impliquant un ensemble de variables. L'objectif est de trouver une valeur pour chaque variable afin de satisfaire l'ensemble des contraintes ou bien de satisfaire le maximum de contraintes. La recherche dans le domaine des CSP est motivée par la découverte de méthodes toujours plus efficaces en pratique pour les résoudre. Mais, comme les CSP font partie de la classe des problèmes combinatoires NP-Complets, les algorithmes connus pour les résoudre nécessitent un temps exponentiel en fonction du nombre de variables. La méthode la plus employée pour les résoudre est la recherche énumérative qui consiste à explorer systématiquement un arbre de recherche.

1. Définitions de base :

Définition 1.1 : (Due à Montanari **[1]**)

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

- $X = (x_1, x_2,\ldots, x_n)$: est un ensemble de n variables ;
- $D = (D_1, D_2,\ldots, D_n)$: est un ensemble de n domaines.
- $C = (C_1, C_2, \ldots, C_m)$: est un ensemble de m contraintes
- $R = (R_1, R_2, \ldots, R_m)$: est un ensemble de m relations. Chaque relation R_i définit l'ensemble des n_i-uplets sur $D_{i1} \times D_{i2} \ldots \times D_{ini}$ autorisés par la contrainte C_i.

Définition 1.2 : Le ***domaine*** d'une variable est l'ensemble des valeurs, discret ou continu que peut prendre celle-ci. Ainsi, une variable est dite numérique si les valeurs qu'elle peut prendre sont entières. Elle est dite booléenne si ses valeurs sont booléennes et symbolique dans le cas où les valeurs qu'elle prend sont un ensemble énuméré d'objets.

Définition 1.3 : une ***affectation*** ou une instanciation est un ensemble de couples (variable, valeur) exprimant l'association des valeurs aux variables. Cette affectation est dite **totale** si toutes les variables ont été instanciées et on parle de ***k-affectation*** si k variables ont été instanciées.

Définition 1.4 : Une ***contrainte*** est une relation logique ou une propriété devant être vérifiée par un sous ensemble de variables. Une contrainte permet de restreindre les valeurs pouvant être prises simultanément par un ensemble de variables. Une contrainte peut être implicite, définie par une expression logique. Elle peut être explicite et donc définie par une relation composée de tous les tuples autorisés. Ainsi, une contrainte est définie par un ensemble de variable et la relation entre ces variables.

Définition 1.5 : une k-affectation est dite ***consistante*** si elle ne viole aucune contrainte. Une ***solution*** d'un CSP est une affectation totale consistante, c'est à dire une assignation d'une valeur à chaque variable de X tel que toutes les contraintes soient satisfaites. Un CSP est dit ***consistant*** s'il possède au moins une solution, dans le cas contraire le CSP est dit ***inconsistant***.

Définition 1.6 : le nombre de variables sur lequel porte la contrainte est appelé ***arité*** de la contrainte. En particulier, un CSP est dit ***binaire*** si toutes les contraintes sont d'arité 2, sinon il est appelé un CSP ***n-aire***.

Exemple : Problème des 4-reines

Le problème des 4 reines est un problème académique qui consiste à placer 4 reines sur un échiquier 4*4 de telle sorte que 2 reines ne soient pas en prise mutuelle. Deux reines ne peuvent être ni sur la même ligne, ni sur la même colonne, ni sur la même diagonale. Ce problème peut être modélisé comme un CSP de plusieurs façons. Ici on illustre l'exemple par une des façons possibles.

P = (X, D, C, R) tel que

- $X = (x_1, x_2, x_3, x_4)$ où x_i représente le numéro de colonne de la reine placée sur la ligne i.
- $D = (D_1, D_2, D_3, D_4)$ tel que Di = {1, 2, 3, 4} (la colonne de la reine sur la ligne i)
- $C = (C_{12}, C_{13}, C_{14}, C_{23}, C_{24}, C_{34})$ tel que C_{ij} = {xi, xj}
- $R = (R_{12}, R_{13}, R_{14}, R_{23}, R_{24}, R_{34})$ tel que pour chaque R_{ij} :

  $x_i \neq x_j$ (les deux reines ne se trouvent pas sur la même colonne)

  $x_i \neq x_j + (j - i)$

  $x_i \neq x_j - (j - i)$ (les deux reines ne se trouvent pas sur la même diagonale)

2. Résolution des CSP : différentes méthodes

La résolution d'un CSP consiste à parcourir toutes les combinaisons possibles ou alors d'explorer un arbre de recherche (arbre de résolution) afin de trouver une ou toutes les solutions. Pour résoudre les CSP plusieurs approches ont été explorées :

- **La recherche énumérative:** consiste à parcourir l'arbre de recherche selon une certaine stratégie de parcours.
- **Le filtrage ou la relaxation des contraintes :** technique mise en œuvre pour réduire la taille des domaines des variables.
- **Les heuristiques :** tentent d'accélérer l'exécution des algorithmes.
- **La décomposition :** méthodes permettant de transformer un CSP dont la structure est généralement un graphe en un CSP acyclique ou arborescent.
- **La combinaison :** c'est l'utilisation des techniques précédentes simultanément.

Dans ce qui suit, nous allons présenter de façon générale les différents algorithmes proposés dans la littérature pour résoudre les CSP binaires, avant de montrer dans la section 3, comment ceux-ci ont été adaptés pour la résolution des CSP n-aires.

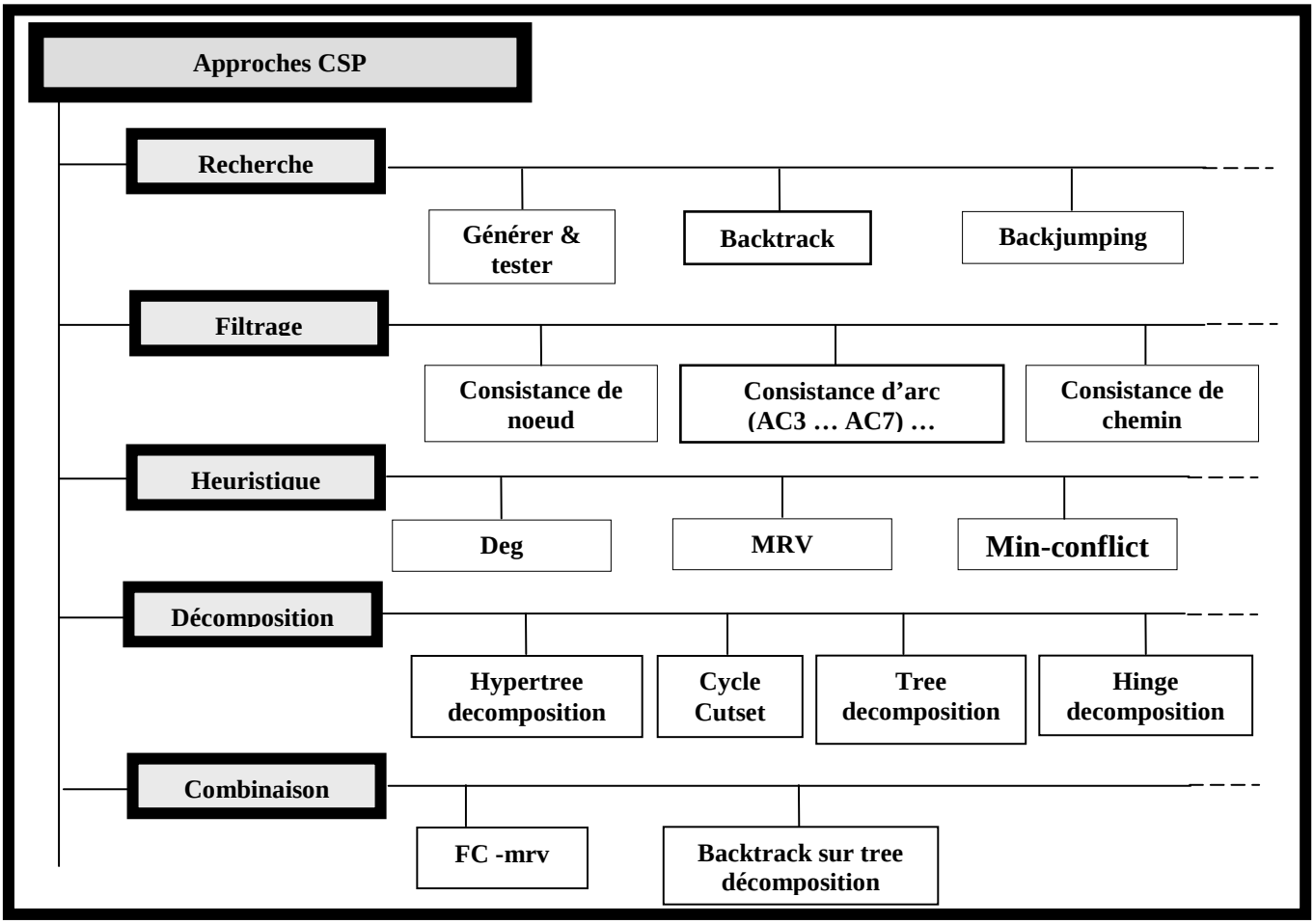

Fig.I.1. Différentes approches pour résoudre les CSPs

2.1. Réduction de l'espace de recherche:

Comme nous allons le voir plus tard, la résolution d'un CSP se base généralement sur la recherche énumérative c'est-à-dire sur l'exploration d'arbre dont la complexité est exponentielle en taille de problème. Pour améliorer les performances des algorithmes de résolution de CSP, il est crucial de réduire l'espace de recherche en utilisant des techniques de filtrage. Plusieurs techniques de filtrage ont été proposées dans la littérature (consistance d'arc, consistance de chemin). La consistance de chemin est très coûteuse et a été très vite abandonnée. La technique la plus utilisée est la consistance d'arc car en plus de son efficacité, certains algorithmes sont faciles à mettre en oeuvre.

La consistance d'arc consiste à supprimer des domaines des variables, des valeurs qui ne peuvent pas participer à une solution du problème. Elle ne garde donc que les valeurs qui sont viables.

Une valeur v_i d'un domaine dom(x_i) d'une variable x_i a un support v_j dans le domaine dom(x_j) d'une variable x_j si le couple (v_i, v_j) appartient à la relation associée à la contrainte $\{x_i,x_j\}$. Une valeur v_i d'un domaine dom(x_i) d'une variable x_i est dite viable si elle a un support v_j dans tous les domaines dom(x_j) des variable x_j telle que $\{x_i,x_j\}$est une contrainte du problème.

L'algorithme le plus utilisé pour implémenter la consistance d'arc est l'algorithme AC3 **[2].** Cet algorithme, très simple à implémenter, a donné naissance à d'autres algorithmes plus récents, héritant du même principe tels que les algorithmes AC2000 **[3]**, AC2001(AC3.1) **[3, 4]**, AC3d**[5]**, AC3.2 **[6]** et AC3.3 **[6]**.

2.1.1. L'Algorithme AC3

Cet algorithme est certainement le plus simple des algorithmes qui implantent la consistance d'arc et a pu être intégré dans de nombreux solveurs de CSP malgré la concurrence de quelques algorithmes plus efficaces théoriquement (complexité théorique en $O(md^2)$ alors que la complexité théorique de AC3 est en $O(md^3)$) mais plus complexes a mettre en œuvre en pratique , voir AC4 **[7]**, AC6 **[8]** et AC7 **[9].**

```
Algorithme AC3 (X, D, C, R)
        Q ← {(i, j) / {i, j} est une contrainte de C}
        Tant que Q est non vide faire
                Extraire un couple (k, m) de Q
                Si Révise(k, m) alors
                        Q ← Q U {(i, k) / {i, k} est une contrainte et i <> m}
                Fsi
        Fait
Fin

Procédure Révise (i,j) : Booléen
        SUPPRESSION ← Faux
        Pour chaque valeur v de domaine(i) faire
                Si v n'a pas de support dans domaine(j) alors supprimer v de domaine(i)
                SUPPRESSION ← Vrai
        FinPour
        Retourner SUPPRESSION
Fin
```

L'algorithme AC3 consiste à appliquer une révision de l'ensemble des contraintes (x_i,x_j). Si après la révision d'une contrainte (x_i,x_j), le domaine de la variable x_i est réduit, alors la révision est faite d'une manière récursive sur toute autre contrainte où x_i apparaît.

2.2. Résolution d'un CSP

Le but majeur de la résolution d'un CSP est de trouver une ou toutes les solutions, c'est à dire des affectations totales consistantes, ou d'affirmer l'inexistence d'une solution. Néanmoins, d'autres objectifs peuvent être attendus.

- Dans le cas où le CSP est **sur-contraint** (inexistence de solution) l'objectif peut être de trouver l'affectation totale qui viole le moins de contraintes possibles; dans ce cas en parle de **max-CSP.**

- Une généralisation de max-CSP est le **VCSP** ou CSP valué qui consiste à donner un poids à chaque contrainte. Le but étant de trouver une affectation totale qui minimise la somme des poids des contraintes violées.

- Un troisième type représente le cas où le CSP est **sous-contraint** (admet plusieurs solutions). Dans ce cas le but peut être de trouver la solution qui maximise une fonction objective. ce type de CSP est appelé **CSOP** (Constraint Satisfaction Optimisation Problem).

2.2.1. Algorithme Backtrack (BT):

Cet algorithme est l'algorithme de base pour la résolution des CSP. Il consiste à instancier successivement les variables et à vérifier le viol d'une contrainte dès que toutes les variables d'une contrainte ont été instanciées. Si une contrainte est violée l'algorithme affecte une nouvelle valeur à la dernière variable jusqu'à trouver une valeur valide ou que son domaine devienne vide. Dans ce dernier cas l'algorithme effectue un retour vers l'avant dernière variable (backtrack) et lui affecte une nouvelle valeur.

L'inconvénient majeur de cet algorithme est le fait de faire des retours en arrière simples ou chronologiques, lors de la découverte d'une inconsistance. En effet, la variable précédente peut ne pas être la cause de l'inconsistance.

```
Algorithme Backtrack (A, (X, D, C, R)) : booléen  // A est une affectation, au premier appelle A = ∅
    Si A est non consistant Alors Retourner faux Fsi
    Sinon si A est une affectation totale alors retourner vrai   /* A est une solution */
    Sinon choisir une nouvelle variable Xi de X
        Pour toute valeur Vi de D(Xi) faire
            Si Backtrack(AU{(Xi,Vi)},(X,D,C,R)) = vrai alors retourner vrai
        Fin pour
    Retourner faux
Fin
Programme principal
    Backtrack(∅, (X, D, C, R))
Fin
```

2.2.2. Algorithme Back Jumping (BJ)

Dans cet algorithme, si toute instanciation de la variable courante génère une inconsistance alors l'algorithme revient vers la variable la plus profonde qui soit reliée avec la variable courante par une contrainte. Si le domaine de cette variable est vide alors l'algorithme fait un simple retour arrière.

2.2.3. Algorithme Conflict Directed Back jumping (CBJ)

Le CBJ est une amélioration de BJ. La principale différence entre ces deux algorithmes réside dans le retour arrière quand on fait un backjumping vers une variable dont le domaine est vide. Pendant que BJ fait un simple retour chronologique, CBJ par contre, tente d'en faire plusieurs. Ceci est réalisé en maintenant pour chaque variable instanciée un ensemble des variables instanciées avant elles et qui sont en conflit avec elles pour au moins une de ses valeurs. Face à un échec, CBJ revient vers la variable la plus profonde dans cet ensemble.

2.2.4. Algorithme Forward Checking (FC):

Cet algorithme tente d'anticiper les inconsistances en maintenant des listes de valeurs légales (qui ne vont probablement pas générer une inconsistance) des variables non encore instanciées. L'algorithme FC se base sur le backtrack pour lequel il applique un filtrage des domaines de variables non encore instanciées de sorte que après l'instanciation d'une variable x, toute les valeurs inconsistantes d'une variable *y* non encore instanciées et reliées à *x* par une contrainte soient supprimées.

```
Algorithme FC (A, NA, D, C)      // A est une affectation , NA ensemble des variables non encore instanciers
        Si NA = {} alors Retourner A        // A est une solution
        Sinon
                Choisir x de NA ;
                Répéter
                        Choisir une valeur v de Dx ; Dx = Dx – {v} ;
                        Si A ∪ {x,v} ne viole aucune contrainte alors
                                D' = Revise(NA – {x}, D, C, <x,v>) ;
                                Si (aucun domain de D' n'est vide) alors
                                        Result ← FC(NA - {x}, A + <x,v>, D', C);
                                        Si Result <> NULL alors Retourner(Result) ;
                                Fsi
                        FSI
                Jusqu'a (Dx = {})
                Retourner (NULL) ;
        FSI
Fin

Procedure Revise (W, D, C, a)    // a est une affectation  d'une variable
        D' = D;
        Pour chaque variable y dans W faire
                Pour chaque valeur v dans D'y faire
                   Si <y,v> est incompatible avec a en respect avec les contraintes de C alors D'y = D'y  - {v}
                Fait
        Fait
        Retourner D'
Fin

Programme principale (X,D,C,R)
        Solution = FC({}, X, D,C) ;
Fin
```

2.2.5. Algorithme Maintenance de consistance d'arc (MAC) :

Dans cet algorithme, la consistance des valeurs est vérifiée pour toutes les variables non encore instanciées et non seulement celles qui sont en relation avec la variable courante comme c'est le cas avec le Forward checking.

```
Algorithme MAC (A, NA, D, C)
Si NA = {} alors return (A)                    // A est une solution
Sinon
        Choisir une variable x de NA
        Répéter
                Choisir une valeur v de Dx ; supprimer v de Dx
                Si A + <x,v> est consistante alors
                        D' = Réviser(NA – {x}, D, C, <x,v>) ; //même fonction que celle de FC
                        AC-X (NA – {x}, D', C) ;        // D' est filtré à l'aide d'un l'algo AC
                        Si (aucun domaine de D' n'est vide) alors
                                Result = MAC(A + <x,v>, NA – {x}, D', C);
                                Si (Result <> NULL) alors return Result;
                        FSI
                FSI
        Jusqu'à Dx = {};
        Return NULL;
FSI
```

Ainsi, après chaque instanciation de variable l'algorithme MAC applique la consistance d'arc en s'appuyant sur n'importe qu'elle algorithme d'arc consistance ; ainsi on parle de MAC3 quand c'est l'AC3 qui est utilisé, et de MAC2001 quand on se base sur l'AC2001.

2.3. Heuristiques pour l'amélioration de la résolution des CSP:

Les expérimentations des algorithmes de résolution de CSP ont montré que l'ordre d'instanciation des variables ainsi que l'ordre des valeurs qu'elles peuvent prendre peuvent influer sur le temps de recherche d'une solution. Ainsi plusieurs heuristiques ont été proposées.

2.3.1. Heuristique sur le choix des variables :

La majorité des heuristiques de choix des variables ont pour but d'arriver à une situation d'échec le plus tôt possible (*First fail principle*). Ce qui a pour effet de réduire la taille des branches et ainsi d'accélérer la recherche. L'ordre d'instanciation des variables peut être statique ou dynamique.

Dans le cas statique, l'ordre est prédéfini avant la résolution de CSP et se base généralement sur les propriétés structurelles du CSP notamment l'heuristique *Deg* qui utilise le degré des variables (nombre de contraintes qui porte sur la variable), ainsi une variable de degré élevé est considérée comme étant critique (responsable de beaucoup d'échec) et sera instanciée plus tôt.

L'ordre dynamique par contre, est défini au fur est à mesure de la résolution de CSP. L'heuristique la plus utilisée est la *Mean Remain Value* (MRV). Cette heuristique est utilisée notamment dans le cas de résolution avec filtrage (FC-MRV) et consiste a instancier en priorité la variable dont le domaine est le plus petit.

2.3.2. Heuristiques sur le choix des valeurs

Min-conflict **[10]** est une des rares propositions d'heuristiques de choix des valeurs. Pour chaque valeur *v* de domaine de la variable courante, cette heuristique calcule le nombre de valeurs des variables non encore instanciées avec lesquelles *v* est incompatible, ainsi la

valeur qui a le plus petit nombre de conflit est affectée en premier. Les heuristiques sur le choix des valeurs sont très lourdes à mettre en œuvre ce qui les rend rarement utilisables.

3. Résolution des CSP n-aires :

Les CSP n-aires sont des CSP où au moins une contrainte est d'arité supérieur à 2. Pour les résoudre deux approches existent : les transformer en CSP-binaires puis les résoudre avec les méthodes précédemment citées, ou les résoudre directement à l'aide d'algorithmes spécifiques.

3.1. Transformation de CSP n-aires en CSP binaires :

Il existe plusieurs façons de transformer un CSP n-aire en CSP binaire. Une des façons possibles consiste à considérer l'ensemble des contraintes du CSP original comme l'ensemble des variables du CSP binaire. Une variable du CSP binaire prend ses valeurs dans l'ensemble des tuples autorisés par la contrainte. Deux variables seront reliées dans le CSP binaire si elles représentent deux contraintes partageant au moins une variable dans le CSP d'origine. Le graphe associé à ce CSP est appelé graphe dual.

3.2. Résolution direct d'un CSP n-aires :

Dans ce cas, il s'agit d'adapter les méthodes conçues pour les CSP binaires afin qu'elle soit applicables aux CSP n-aires. Il s'agit notamment de la généralisation de la consistance d'arc et de l'algorithme forward checking aux contraintes n-aires.

3.2.1. La consistance d'arc dans les CSP n-aires :

Pour pouvoir filtrer directement un CSP n-aire, la consistance d'arc a été généralisée pour tenir compte des contraintes d'arité supérieur à 2.

Définition.3.3 : la consistance d'arc généralisée (consistance d'hyper arc) est définie comme suit :

Soit var(C_j) = {x_{j1}, x_{j2},…, x_{jq} } l'ensemble des variables sur lequel porte la contrainte C_j, rel(C_j) est l'ensemble des tuples qui vérifient la contrainte C_j et $D_{x_i=a}^{var(c_j)}$ l'ensemble des tuples de D_{j1}* D_{j2}*…* D_i' *…* D_{jq} tel que D_i' = {a}.

On appelle support de (xi, a) dans C_j les tuples τ de l'ensemble $D_{x_i=a}^{var(c_j)} \cap$ rel(C_j).

On dit qu'une contrainte C_j est arc consistante généralisée si pour toute variable x_i de var(C_j), chaque valeur v de Di a un support dans C_j.

3.2.2. Algorithme GAC2001 (GAC3.1) :

GAC2001 **[11]** est une généralisation de AC2001 pour le filtrage des CSP n-aires en se basant sur la consistance d'arc généralisée définie précédemment.

Cet algorithme opère sur un CSP dont les tuples de rel(C_j) (les tuples qui ne violent pas la contrainte Cj) sont supposés être totalement ordonnés et le dernier support de chaque couple (xi, a) pour une contrainte Cj est sauvegardé dans une variable Last((xi, a), Cj) initialisé à NIL. Ainsi, lors de la recherche de support d'une affectation on vérifie premièrement si la valeur sauvegardée dans la variable **last** correspondante existe toujours. Si c'est le cas l'affectation possède un support sinon la recherche commence à partir de la valeur suivante (fonction **succ**), les précédentes valeurs ayant déjà fait l'objet d'une vérification.

```
procedure REVISE2001(xi, Cj)
Début
supprime = faux
pour chaque  a ∈ Di  faire
        τ =  Last((xi, a), Cj )
        Si ∃ k= τ [xjk ] ∉ Djk  Alors
                τ =  succ(τ , D_{xi=a}^{var(cj)} )
                Tant que  (τ ≠ NIL)  ET  (τ  ∉ Rel(Cj)) Faire τ = succ(τ, D_{xi=a}^{var(cj)} )
                Si τ ≠ NIL Alors  Last((xi , a), Cj ) = τ
                Sinon
                        Supprimer a de Di
                        supprime = Vrai
                Fsi
        Fsi
Fait
Return supprime
Fin


Algorithme GAC2001
Début
        Pour chaque variable xi faire
                Pour chaque valeur a de Di faire
                        Pour chaque contrainte Cj faire Last((xi , a), Cj ) =Nil
        Q  = {(xi , Cj ) / Cj  ∈  C et xi ∈ var(Cj)}
        Tant que  Q non vide  Faire
            Choisir  (xi , Cj ) de Q
            Q = Q – {xi, Cj}
            Si REVISE2001(xi; Cj ) Alors Q =  Q ∪ {(xk , Cm) | Cm ∈ C;  xi,xk ∈ var(Cj)  et k ≠ i et j ≠ m}
        Fait
Fin
```

3.2.3 Algorithme nFC

L'algorithme nFC **[12]** est une généralisation de l'algorithme forward checking au cas n-aire, Ainsi nFC définit 2 ensembles : $C^n_{c,f}$ composé des contraintes qui contiennent la variable courante et au moins une variable de future, et $C^n_{p,f}$ qui est composé des contraintes qui contiennent au moins une variable de passé et au moins une variable de futur. La grande différence avec le cas binaire est que nous avons à traiter des cas où plusieurs variables d'une contrainte ne sont pas encore instanciées. Plusieurs variantes de nFC existent suivant qu'on utilise $C^n_{c,f}$ ou $C^n_{p,f}$ et suivant qu'on applique la consistance d'arc à tout l'ensemble ou aux contraintes une à une en une seule passe.

- **nFC5 :** après l'instanciation de la variable courante, rendre l'ensemble $C^n_{p,f}$ arc consistant puis continuer l'instanciation en cas de succès sinon en fait un retour arrière.
- **nFC4 :** après l'instanciation de la variable courante, appliquer l'arc consistance à chaque contrainte de $C^n_{p,f}$ en une seule passe puis continuer l'instanciation en cas de succès sinon en fait un retour arrière.
- **nFC3 :** après l'instanciation de la variable courante, rendre l'ensemble $C^n_{c,f}$ arc consistant puis continuer l'instanciation en cas de succès sinon en fait un retour arrière.
- **nFC2 :** après l'instanciation de la variable courante, appliquer l'arc consistance à chaque contrainte de $C^n_{c,f}$ en une seule passe puis continuer l'instanciation en cas de succès sinon en fait un retour arrière.

4. Vers une plateforme multi approches de résolution des CSP :

Dans le cadre de ce mémoire, nous avons commencé par étudier et implémenter les différents algorithmes de résolution de CSP proposés dans la littérature. Même si l'objectif que nous nous sommes fixés n'est pas de résoudre directement les CSP par des méthodes énumératives, il n'en demeure pas moins que ces algorithmes nous seront utiles pour la suite de notre travail et seront intégrés comme nous le verrons plus loin dans notre solveur.

Nous avons donc assez rapidement proposé un solveur permettant la résolution des CSPs par des techniques énumératives.

4.1. Notre solveur CSP par des méthodes énumératives :

Apres la modélisation de CSP, Nous l'avons enrichi afin d'obtenir notre premier outil qui consiste en un solveur implantant quelques algorithmes de résolution de CSP les plus utilisés.

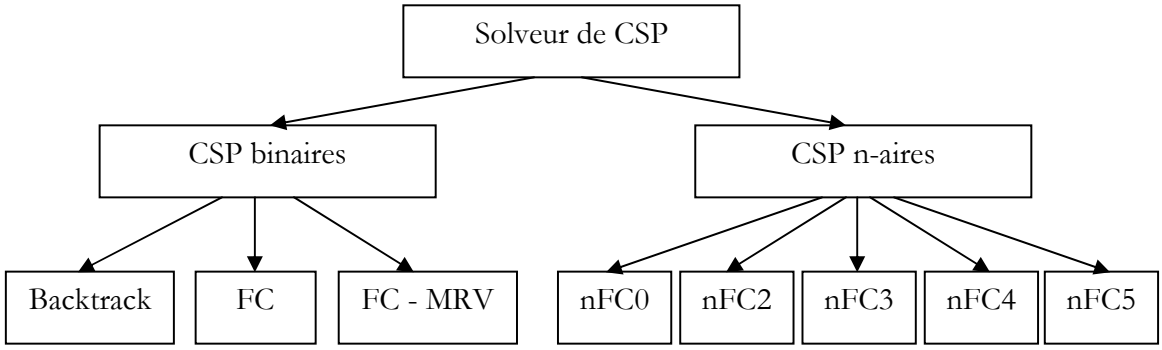

Fig.I.2. Solveur de CSP

Le diagramme de classe complet de notre solveur pour les CSP définit en extension est illustré dans la figure suivante (Fig.1.3).

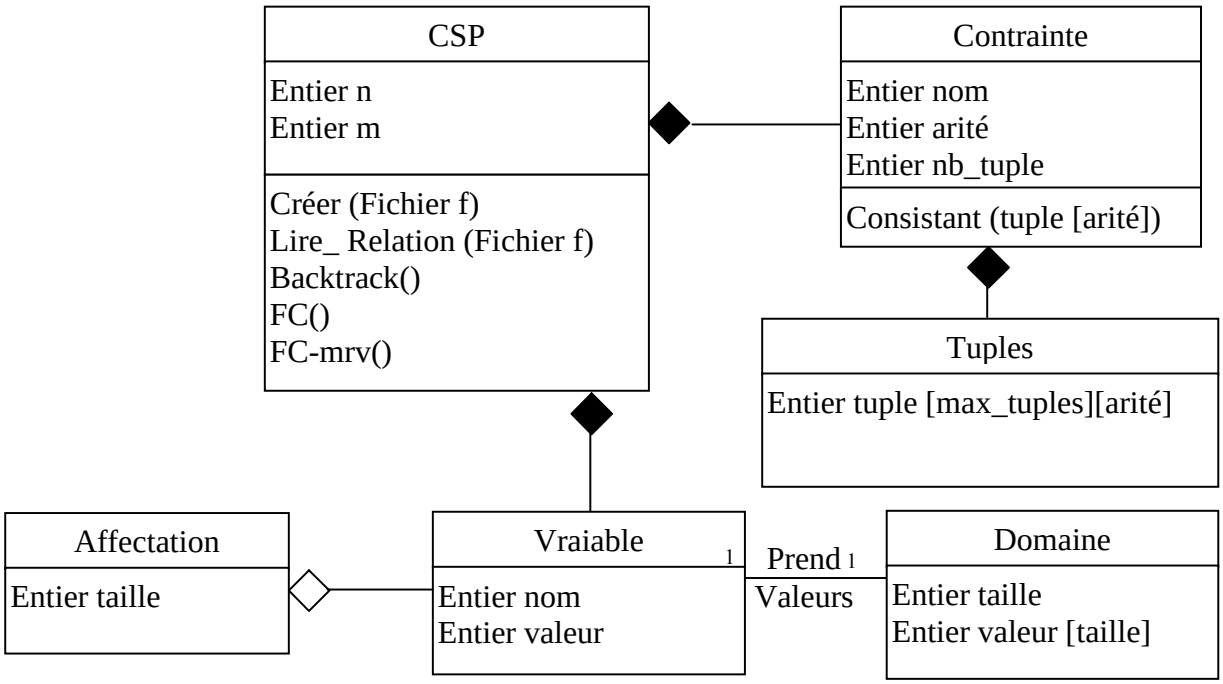

Fig.I.3. Diagramme de classe complet de solveur de CSP définit en extension

Ce diagramme de classe est le résultat de plusieurs itérations dans le développement du solveur, qui a été réalisé en C++ pour la performance de code généré et de sa parfaite portabilité entre Windows et Linux. Des modifications ont été apportées successivement en réponse à des problèmes rencontrés lors des phases de tests et notamment le problème d'explosion mémoire induit par la gestion des tuples des contraintes. Ainsi, l'approche purement statique qui se base sur un tableau a 2 dimensions (tuples[nb_tuple][arité]) a été écarté d'emblée; quant a l'approche purement dynamique, elle a été écartée car, même si l'approche est idéale pour la gestion de la mémoire, elle est très gourmande en temps de calcul. Donc nous avons opté pour une combinaison des approches statique et dynamique. La première méthode testée consiste à utiliser un tableau statique avec une case pour chaque tuple de la contrainte, chacune d'elle pointant vers un tableau créé dynamiquement suivant le nombre de tuples effectif. Cette approche bien qu'efficace, causait un problème d'explosion de la pile pour les problèmes de grande taille, car sous Windows les tableaux statiques sont placés dans la pile d'exécution de programme. Pour remédier à ce dernier problème, nous avons opté pour le regroupement des tuples dans des tableaux d'une certaine taille qui sont créés dynamiquement. Chacun d'eux étant accessible par une des cases d'un tableau statique utilisé à la manière d'une table de hachage. Ainsi, comme le tableau statique est de taille modeste, celui-ci peut être placé facilement dans la pile d'exécution de programme.

Comme une multitude de CSP sont disponibles sous forme de fichier texte, nous avons équipé la classe CSP de deux méthodes. La première permet de lire la structure de CSP à partir d'un fichier contenant la description des contraintes alors que la deuxième permet de définir la relation de chaque contrainte en extrayant d'un fichier texte les tuples associés à chaque contrainte.

La structure du fichier de contraintes est définie par la grammaire g1=(N1,T1,A1, R1) suivante :

R1 = {S -> Contrainte, S / Contrainte.

Contrainte -> nom_contrainte (variables)

Variables -> nom_variable, Variables / nom_variables

}

N1 = {S, Contrainte, Variable, nom_contrainte, nom_variables};

T1 = { ‘,’, ‘.’, ‘(‘, ‘)’} ; A1 = {S}

Celle de fichier des relations est g2 = (N2, T2, A2, R2):

R2 = { S -> Relation, S / Relation.

Relation -> nom_contrainte [Tuples]

Tuples -> (Tuple), Tuples / (Tuple).

Tuple ->valeur, Tuple / valeur

}

N2 = {S, Relation, Tuples, Tuple, nom_contrainte, valeur}; T2 = {‘,’, ‘.’, ‘(‘, ‘)’}; A2 = {S}

Pour les CSP définit en intension nous avons optez pour la modélisation suivante

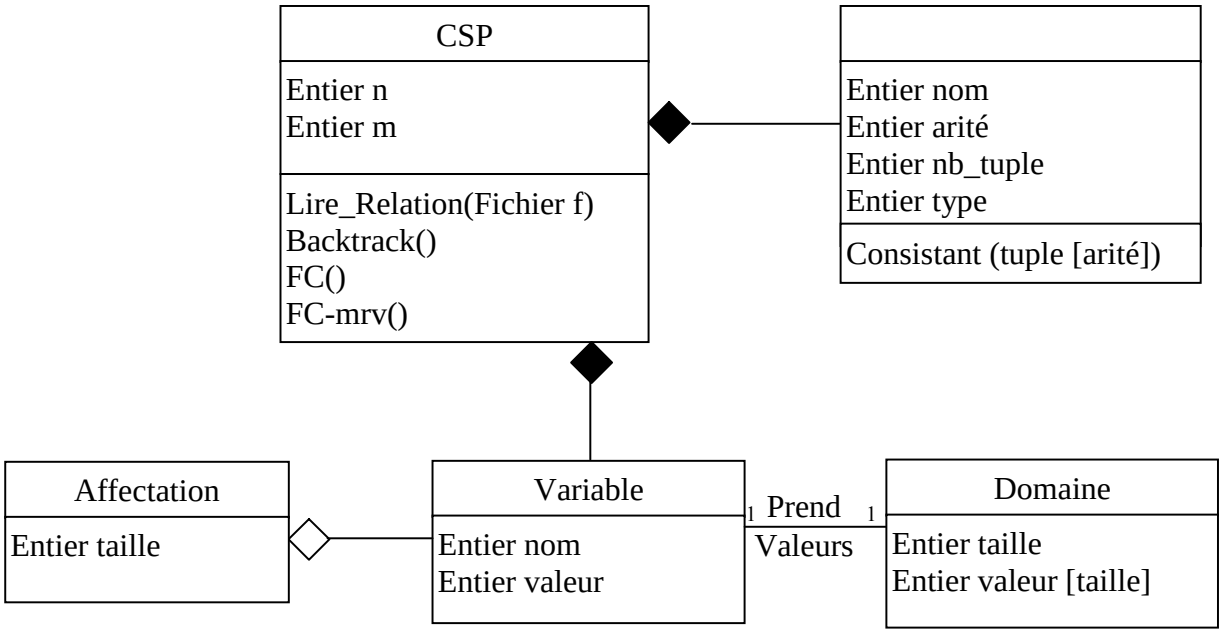

Fig.I.4. Diagramme de classe complet de solveur de CSP définit en intension

Ainsi, chaque déclaration d’une contrainte sera considérée comme un type de contrainte en utilisant le champ « type » rajouté dans la classe «Contrainte» ; de cette manière, la méthode « Consistant() » réagira différemment en fonction de ce champ. Dans ce cas, la structure et les relations sont lues à partir de même fichier. Sa structure est définie par la grammaire g3 = (N3, T3, A3, R3) suivante

R3 = { S -> Var valeur; Dom valeur; Relations

Relations -> Relation Relations/Relation

Relation ->Contrainte (Variables) ; Precondition (Expression_booléenne) ;

Postcondition (Expression_booléene) ;

Variables -> nom_variable, Variables/ nom_variable

}

A3 = {S} ; N3 = {S, Relations, Relation, Variables, Expression_booléenne, valeur, nom_variable, valeur}

T3 = {Var, Dom, Contrainte, postcondition, precondition, ' ;', ',', '(', ')'}

Remarque :

Les non terminaux nom_contrainte, nom_variable, valeur sont définit de la même manière que les entiers en C++.

Le non terminal Expression_booléenne est défini de la même manière que les expressions booléenne en C++

Exemple : pour le problème des 4 reines (voire sa définition dans la première section) les fichier utiliser pour le définir en extension sont comme suit :

Fichier de la structure de CSP

0 (0,1), 1 (0,2), 2 (0,3), 3 (1,2), 4 (1,3), 5 (2,3).

Fichier des relations (en extension)

0[(0,2), (0,3), (1,3), (2,0), (3,0), (3,1)], 1[…

Pour définir le problème en intension

 # 4 variables et taille de domaine = 4

Var 4; Dom 4 ;

 # Les contraintes sont toute binaire

Contrainte (i,j)

 # Tout couple de variables différentes sont régie par cette contrainte

Precondition (i < j)

 # La relation que doivent vérifier les variables

Postcondition(((Xi!=Xj)&&(Xi!=Xj-(j-i))&&(Xi!=(Xj+(j-i))));

 # Pour une variable i, Xi est la valeur ce celle ci.

4.2. Résultats Expérimentaux :

Après la réalisation du solveur, nous avons réalisé une petite étude comparative entre quelque uns des algorithmes de résolution de CSP que nous avons implémenté. Les résultats sont résumés dans les tableaux I.1 et I.2.

Instances	Métrique	Backtrack	FC	FC-mrv
8	**Nombre de solutions**	92	92	92
	Nombre de nœuds	15720	1632	1248
	Temps (ms)	75	5	5
10	**Nombre de solutions**	724	724	724
	Nombre de nœuds	348150	27108	18862
	Temps (ms)	2886	55	41
12	**Nombre de solutions**	14200	14200	14200
	Nombre de nœuds	10103868	627774	399286
	Temps (ms)	161013	1535	1049
14	**Nombre de solutions**	-	365596	365596
	Nombre de nœuds	-	19422066	11348418
	Temps (ms)	-	62275	37183
15	**Nombre de solutions**	-	2279184	2279184
	Nombre de nœuds	-	119219329	66919801
	Temps (ms)	-	427285	245498
16	**Nombre de solutions**	-	-	14772512
	Nombre de nœuds	-	-	422188406
	Temps (ms)	-	-	1752276

Tab.I.1. résultats de la résolution de problème des n-reines

Le tableau précédent regroupe les résultats de la résolution de problème des n-reines définies en extension. On peut aisément déduire que FC-MRV, qui combine l'algorithme FC avec l'heuristique MRV pour ordonner les variables, est l'algorithme le plus efficace car il permet de visiter un plus petit nombre de nœuds dans l'arbre de recherche et permet ainsi de minimiser le temps de calcul.

Pour les CSP n-aires, nous avons pris comme exemple le lemme de schur. Ce problème consiste a placer n balles numérotées de 1 à n dans k boites de telle façon que 3 balles numérotées respectivement i, j, k tel que k = i+j ne soient jamais dans la même boite. En plus de la contrainte d'arité 3 mentionnée précédemment nous avons ajouté une contrainte binaire qui interdit à chaque couple de balles i, j d'être dans la même boite si i = 2*j ; ceci pour voir la réaction des algorithmes quand ils se trouvent en face d'un problème dont l'arité des contraintes n'est pas uniforme. Ne avons modélisé les problèmes de 20, 21 et 22 balles avec 4 boites en intension et nous avons cherché toutes les solutions existantes. Les résultats sont résumés dans le tableau I.2.

instance	Métrique	nFC0	nFC2	nFC3	nFC4	nFC5
20	Nombre de solutions	91982616	91982616	91982616	91982616	91982616
	Nombre de nœuds	47718232	47718208	47718225	47706818	47703929
	Temps (ms)	335653	340540	375106	634433	824029
21	Nombre de solutions	190188720	190188720	190188720	190188720	190188720
	Nombre de nœuds	93329560	93329560	93329560	93309102	93303939
	Temps (ms)	701182	733214	772420	1305306	1563173
22	Nombre de solutions	2114937	2114937	2114937	2114937	2114937
	Nombre de nœuds	111883864	111883882	111883921	111842477	111832468
	Temps (ms)	925623	1002816	1035772	1784216	2155614

Tab.I.2. Résultats de la résolution de problème de Schur

Du tableau précédent nous pouvons remarquer que c'est l'algorithme nFC5 qui minimise le nombre de nœuds visités grâce a son filtrage plus avancé, mais la lourdeur de filtrage dans les CSP n-aires induit un coût en calcul qui surpasse le gain obtenu par la réduction du nombre de nœuds de l'arbre de recherche ; ce qui explique la supériorité de nFC0, qui se base sur un filtrage plus léger, en terme de temps de résolution.

Conclusion

Ce chapitre a servi de prélude à notre travail en nous permettant d'aborder les aspects formels liés au concept de problème de satisfaction de contrainte (CSP) et sur les approches utilisées pour les résoudre. Nous avons détaillé quelques uns des algorithmes de résolution qui se basent sur la recherche énumérative en mettant l'accent sur les différences qui existent entre les CSP binaires et les CSP n-aires. Enfin, nous avons présenté notre outil de résolution que nous avons développé dans le but de disposer d'une plateforme multi-approches de résolution de CSP et nous permettant ainsi d'expérimenter aisément nos propres algorithmes et les comparer avec les algorithmes de la littérature. Avec cet outil nous avons pu confirmer l'intérêt pratique des algorithmes de résolution qui se base sur la recherche. Néanmoins, leurs complexités théoriques, étant exponentielles en la taille de problème, laissent présager l'impossibilité de leur utilisation pour des classes de CSP de grande taille. A cet effet nous allons explorer, dans les chapitres suivants, une autre approche de résolution de CSP qui se base sur les propriétés structurelles des CSP et permettant de réduire la complexité théorique de leur résolution.

Chapitre II

Les méthodes de décomposition structurelle

La résolution des CSP se base généralement sur des techniques de recherche de type Backtrack. Le Backtrack combiné à des techniques de retour arrière intelligent et des techniques de filtrage en plus de l'utilisation d'heuristiques sur l'ordre des variables induit généralement de bonnes performances en pratique comme nous l'avons vu dans le chapitre précédent. Néanmoins, la complexité théorique temporelle et spatiale de ce type de méthodes reste exponentielle en la taille du problème. Les retours arrière chronologique et intelligent, de même que les techniques de propagation par consistance induisent pour certaines classes de CSP, un coût très important en temps CPU et en espace mémoire.

Au niveau théorique, des travaux ont porté sur la recherche d'algorithmes capable de résoudre certaines classes de CSP en un temps polynomial. Ces classes de CSP sont dites « tractables » ou traitables. Une de ces classes est celle des CSP acycliques, dont la structure peut être représentée sous forme arborescente.

Plusieurs méthodes exploitant cette caractéristique et recherchant des moyens pour transformer n'importe quel CSP en un CSP acyclique ont été introduites. Ces dernières portent le nom de méthodes de décomposition structurelle.

1. Tractabilité

La résolution de CSP en général est un problème NP-Complet et il n'existe aucun algorithme général qui permet de résoudre n'importe quel CSP en un temps polynomial. Ainsi, beaucoup d'efforts ont été faits, notamment dans les communautés de l'Intelligence Artificielle et des bases de données, pour identifier des classes de CSP pour lesquels des algorithmes polynomiaux spécifiques peuvent leur être associés.

Définition 1.1 : Une classe de CSPs est dite tractable si il existe un algorithme polynomial pour les résoudre.

1.1. Tractabilité due à la structure

Celle-ci inclue toutes les classes de CSP tractables identifiés uniquement par la structure de leur ensemble de contraintes C indépendamment de leur ensemble de relations R.

1.2. Tractabilité due aux relations

Celle-ci inclue toutes les classes de CSP tractables identifiées par certaines propriétés particulières de leurs relations.

Dans le reste de chapitre nous allons nous intéresser uniquement à la tractabilité structurelle.

1.3. Structures d'un CSP

A partir des variables d'un CSP et des contraintes qui le composent, plusieurs structures graphiques peuvent être définies.

Définition 1.2 : L'hypergraphe d'un CSP P = <X, D, C, R> est l'hypergraphe H = (X, C), celui-ci est appelé graphe si toute les contrainte dans C sont binaires.

Définition 1.3 : Le graphe primal d'un hypergraphe H = (V, E) est le graphe G = (V, E') tel que (v1, v2) $\in$ E' si il existe une hyper-arête h $\in$ E tel que {v1, v2} $\subseteq$ h.

Définition 1.4 : Le graphe dual d'un hypergraphe H = (V, E) est le graphe G = (E, E') tel que chaque sommet de G représente une hyper-arête de H et une arête (E1, E2) $\in$ E' si E1 et E2 partagent des variables (E1 $\cap$ E2 $\neq \emptyset$).

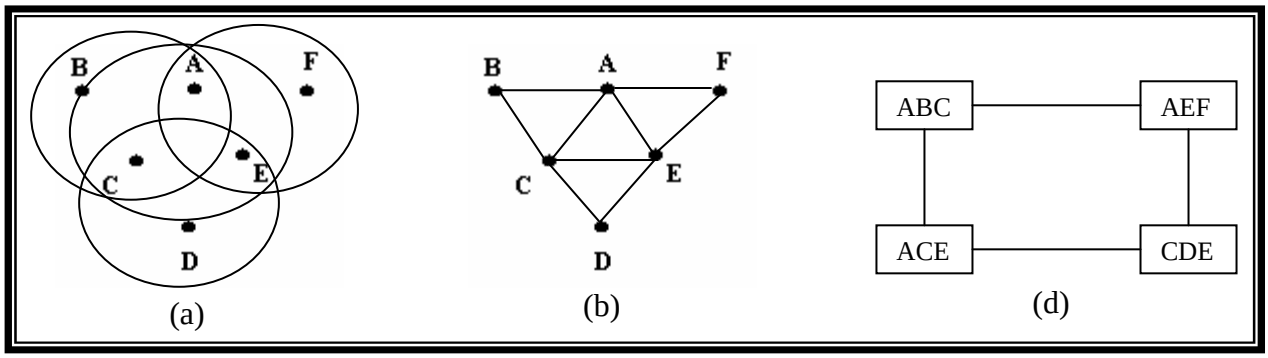

Fig.II.1. (a) hypergraphe (b) graphe primal (c) graphe dual

2. Méthodes de décomposition structurelle

L'identification de classes de CSP dont la structure particulière permet une résolution polynomiale, a été à l'origine des méthodes de décomposition structurelle. Celles-ci ont pour but de transformer un CSP quelconque en un CSP tractable équivalent : un CSP ayant les mêmes solutions que les CSP de départ.

Les méthodes de décomposition consistent à regrouper les sommets et les arêtes de l'hypergraphe d'un CSP en clusters de variables ou d'arêtes et d'organiser les clusters entre eux pour former un CSP dont la structure est un arbre.

De plus, chaque méthode de décomposition propose une mesure de la cyclicité de CSP appelée largeur de la décomposition. Cette mesure garantit que la résolution n'est plus exponentielle en la taille d'un problème mais en fonction de cette largeur. C'est pour cette raison que il est crucial de minimiser la largeur de la décomposition.

Dans ce qui suit, nous allons explorer quelque unes des méthodes de décomposition les plus intéressante de la littérature.

2.1. Méthode Cycle cutset

La méthode cycle cutset **[18]** ou méthode coupe cycle développée pour les CSP binaires, se base sur le principe que l'instanciation d'une variable revient à la supprimer du réseau de contrainte associé à un CSP. Ceci permet donc de changer la connectivité du graphe.

On appelle ensemble coupe cycle (cycle Cutset) un ensemble de variables dont l'instanciation permet la création d'un graphe acyclique. De cette manière, l'identification de

l'ensemble coupe cycle permet donc l'application de l'algorithme de résolution d'arbre au reste de problème pour trouver la solution en un temps polynomial.

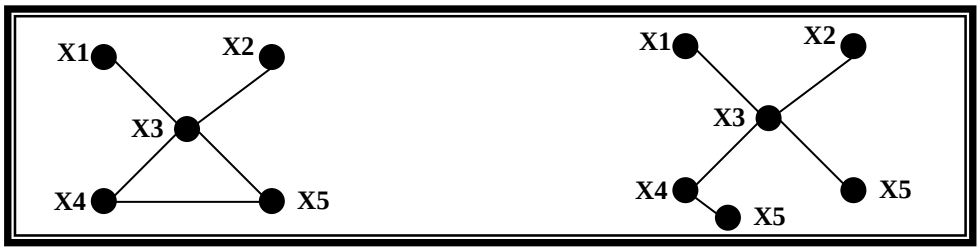

Fig.II.2. graphe d'un CSP avant et après l'instanciation de X5

L'algorithme général de résolution avec cette méthode consiste a utiliser une technique de résolution de type backtrack et de tester à chaque instanciation de variable la cyclicité du graphe résultant jusqu'à ce que celui-ci soit reconnu comme étant acyclique. Dans ce cas, l'ensemble des variables instanciées forme l'ensemble coupe cycle. Un algorithme de résolution d'arbre est utilisé pour résoudre le problème restant. Si une solution est trouvée alors l'algorithme s'arrête sinon il effectue un backtrack sur les variables de l'ensemble coupe cycle. Un résumer de cet algorithme est illustrer dans la figure II.3.

La mesure de cyclicité de la méthode coupe cycle est appelée Cutset width et elle est égale à la taille (nombre de sommets) de l'ensemble coupe cycle le plus petit. La méthode Cycle Cutset permet donc de résoudre un problème dont l'ensemble coupe cycle a été identifié en un temps polynomial car la complexité temporelle de la procédure backtrack utilisée n'est pas exponentielle en la taille du CSP mais exponentielle en la taille de l'ensemble coupe cycle alors que la complexité de l'algorithme de résolution d'arbre est polynomial.

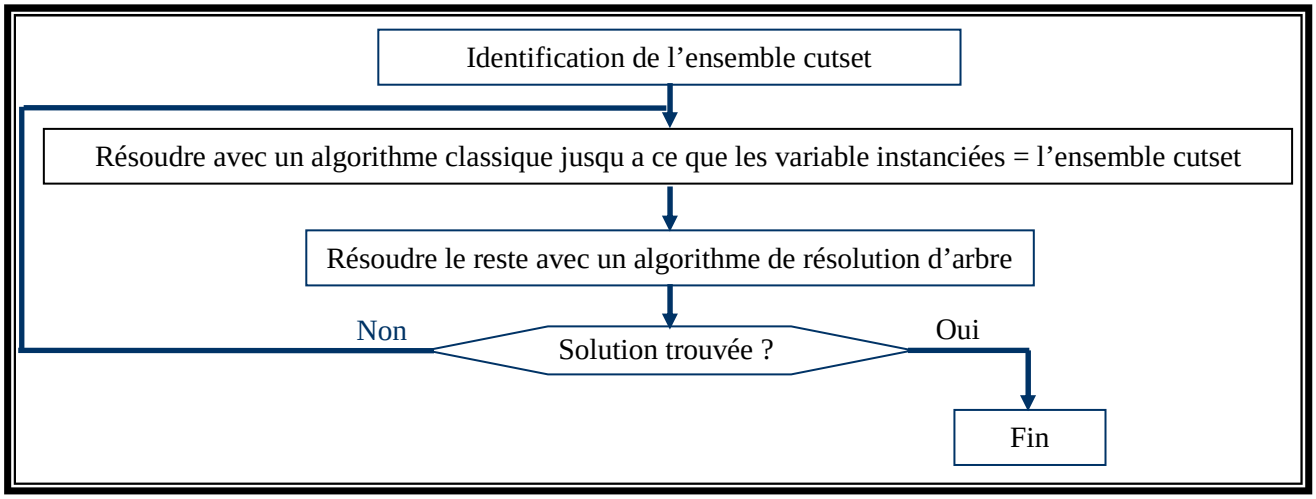

Fig.II.3. Algorithme général de résolution avec la méthode coupe cycle

2.2. Méthode Biconnected-component

La méthode biconnected component **[19]** est une méthode qui se base sur la notion de composante biconnexe pour la décomposition des CSP binaires.

Définition 2.1 : Un séparateur est un sommet dont la suppression induit l'augmentation du nombre des composantes connexes.

Définition 2.2 : Un graphe est dit biconnexe si il existe deux chemins sommets disjoints (qui ne passe pas par les même sommets) qui relient chaque couple de variables (u, v) tel que $u \neq v$ (c'est-à-dire qu'il ne contient pas de séparateurs).

Définition 2.3 : Une composante biconnexe est un sous graphe biconnexe minimal c'est-à-dire qui ne contient pas un autre sous graphe biconnexe.

Ainsi, Une bicomponent decomposition de H = (G, E): est un couple (T, χ) tel que T est un arbre et χ une fonction qui associe à chaque sommet de T un ensemble S de sommets de G tel que S est une composante biconnexe. Deux noeuds p et q de T sont connectés si ils partagent un séparateur.

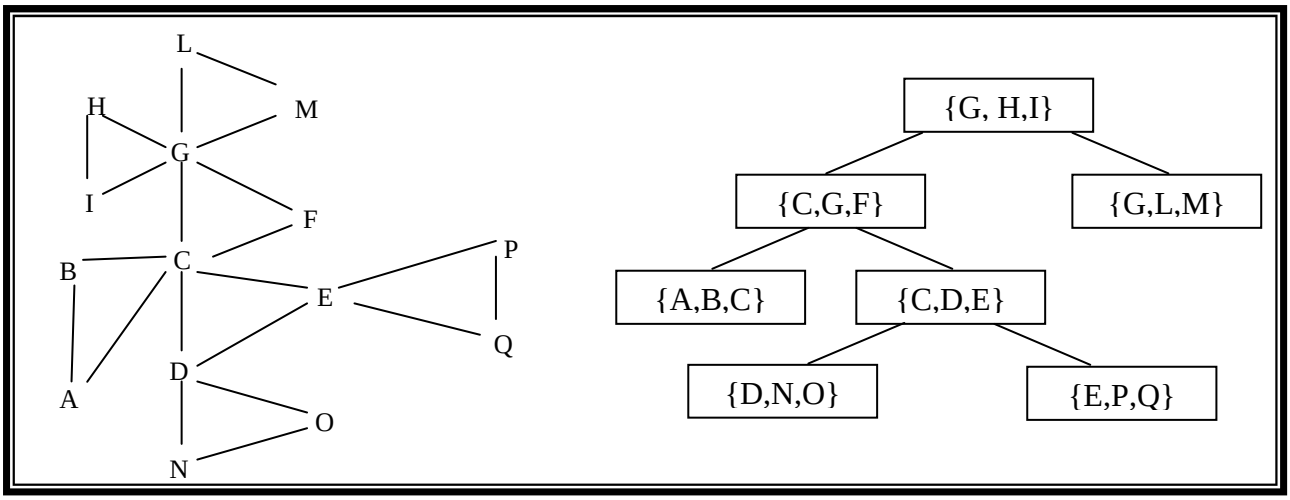

Fig.II.4. Un graphe est sa biconnected decomposition

La largeur de décomposition considérée dans cette méthode est la taille maximale des composantes biconnexes. Cette décomposition peut être intéressante car son calcul est linéaire. Malheureusement, seul un nombre réduit de CSPs peut être décomposé avec cette méthode comparativement aux autres méthodes.

2.3. Méthode Tree-Clustering

La méthode tree clustering introduite par Rina Dechter **[20]** est l'une des méthodes de décomposition les plus intéressantes avant l'apparition de l'hypertree decomposition.

Elle se base sur le fait qu'un CSP est acyclique si et seulement si son graphe primal est à la fois chordal et conforme **[21]**.

Définition 2.4 : Le graphe primal d'un CSP est dit chordal si chaque cycle de longueur supérieur à trois possède une corde i.e. une arête joignant deux nœuds non adjacents dans le cycle.

Définition 2.5 : Le graphe primal d'un CSP est dit conforme si chaque clique maximale (composante complètement connectée maximale) correspond à une contrainte du CSP.

La tree clustering s'appuie sur l'algorithme de triangulation mis au point par Tarjan et Yannakakis dans **[22]** qui transforme un graphe quelconque en un graphe triangulé. Les cliques maximales résultant de cet algorithme formeront les contraintes du CSP acyclique équivalent. L'algorithme de triangulation consiste en deux étapes :

1) Ordonner les sommets du graphe primal en utilisant l'heuristique *Maximum Cardinality Search* (MCS) **[22]**.

2) ajouter récursivement des arêtes entre chaque couple de nœuds non adjacents qui sont connectés par des nœuds supérieurs dans l'ordre calculé précédemment.

L'heuristique MCS numérote les sommets du graphe primal de 1 à n (n : nombre de variable de CSP) de telle sorte que le prochain numéro à attribuer est assigné au sommet ayant le plus de voisins déjà numérotés. Cette méthode est résumée dans la figure II.5.

Le join tree généré lors de la troisième étape est un arbre dont les sommets sont les mêmes que ceux du graphe dual et son ensemble d'arêtes est un sous ensemble d'arêtes du graphe dual. Les arêtes sont choisies de telle façon que pour chaque variable du CSP, l'ensemble des sommets constitués de contraintes contenant cette variable soit connectés

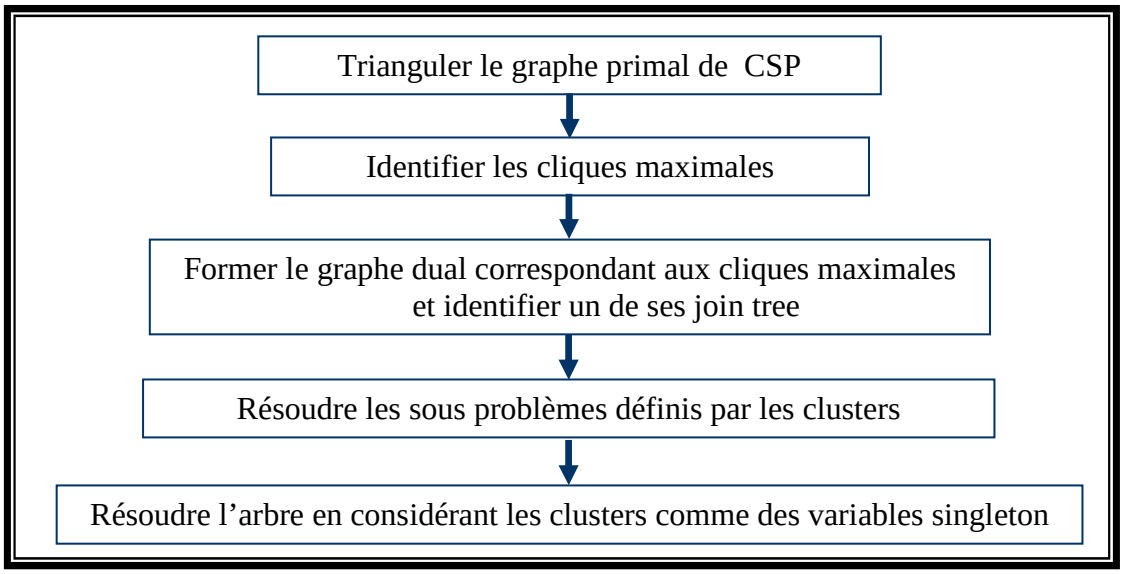

Fig.II.5. Etapes de l'algorithme tree clustering

Exemple : Soit le CSP P = (X, D, C, R) tel que X = {A, B, C, D, E} et C = {{A,D},{A,C},{C,E},{D,E},{B,D}}.

Le graphe de contraintes de P ainsi que son graphe triangulé, le graphe dual du CSP acyclique généré et son join tree sont illustrés dans la figure II.6 suivante

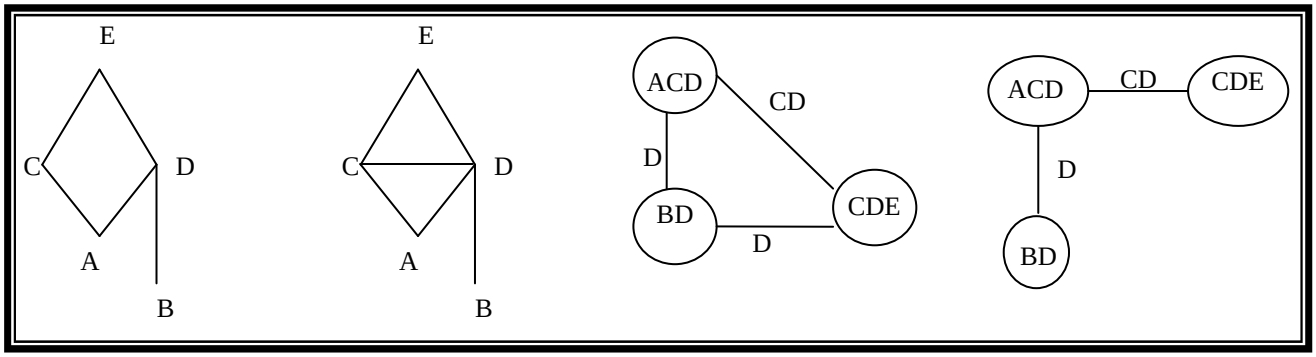

Fig.II.6. Etape de décomposition de P

La largeur de décomposition de la tree clustering est égale au nombre maximal de variables des cliques maximales.

2.4. Méthode Hinge decomposition

La méthode Hinge decomposition **[23]** est une des premières méthodes à exploiter directement l'hypergraphe d'un CSP en se basant sur le concept des hinges, ce qui la rend applicable aussi bien pour les CSP binaires que pour les CSP n-aires.

Définition 2.6 (Composante connectée) : soit (V, E) un hypergraphe, et soit H $\subseteq$ E et F $\subseteq$ (E –H). F est dit *connecté par rapport à H* si pour chaque couple d'arêtes e et f de F, il

existe une séquence d'arêtes $e_1,\ldots,e_n$ dans F tel que $e = e_1$ et $f = e_n$ et pour i = 1 à n-1 : $[(e_i \cap e_{i+1}) - (\cup H)] \neq \emptyset$. L'ensemble maximal de (E – H) qui soit connecté par rapport à H est appelé une *Composante connectée de (E – H) par rapport à H*.

Définition 2.7 (Hinge) : soit (V, E) un hypergraphe et $H \subseteq E$ un ensemble contenant au moins deux arêtes. Soit $H_1, \ldots, H_m$ les composantes connectées de (E – H) par rapport à H. alors H est appelé ***Hinge*** si pour i = 1 à m, il existe une arête h_i dans H tel que $(\cup Hi) \cap (\cup H) \subseteq h_i$. Et dans ce cas hi est appelé séparateur de H_i. Un hinge est dit minimal s'il ne contient aucun hinge.

Définition 2.8 : soit (V, E) un hypergraphe ; une hinge tree est un arbre T = (N, A) tel que :

1) les nœuds de T sont les hinge minimaux de (V, E)
2) chaque arête de E est contenue dans au moins un nœud de T
3) deux nœuds adjacents dans l'arbre partagent exactement une arête de E. De plus, les variables qu'ils partagent sont exactement les membres de cette arête.
4) chaque sommet de V partagé par deux nœuds de l'hinge tree doit être contenu dans chaque nœud appartenant au chemin qui les relie.

Pour calculer un hinge tree d'un hypergraphe, un algorithme a été proposé , fondé sur les concepts des hinge et des composantes connectées vus plus haut.

Algorithme Calcule de hinge tree
Entrée : un hypergraphe (V, E)
Sortie : un hinge tree T

1. Marquer chaque arête de E comme non utilisé. Initialiser i = 0, $N_0 = \{E\}$ et $A_0 = \emptyset$ et marquer le noeud E de N_0 comme non minimal.
2. Si tout les nœud de Ni sont marquer comme étant minimal alors T = (N_i, A_i) et Fin. Sinon, choisir un noeud non minimal F de N_i
3. Si toute els arêtes de F sont marquer comme étant utilisé alors marquer F comme minimal et aller à 2. Sinon choisir une arête e de F et la marquer comme utilisé.
4. Soit $\Gamma = \{G \cup \{e\} |$ G est une composante connectée de (F - e) par rapport à e$\}$; et soit γ n'import quelle fonction de F vers Γ tel que quelle que soit $f \in F$, $f \in \gamma(f)$. Si $|\Gamma| = 1$ aller à 3.
5. Initialiser :
   $N_{i+1} = (N_i - \{F\}) \cup \Gamma$
   $A_{i+1} = (A_i - \{(\{F, F'\}, f) / (\{F, F'\},f) \in A_i\}) \cup \{(\{\gamma(f), F'\}, f) / (\{F, F'\},f) \in A_i\}$
   $\cup \{(\{\gamma(f), \gamma(e)\}, e) / f \in F, \gamma(f) \neq \gamma(e)\}$
   Et marquer tous les nouveaux nœuds ajoutés à N_{i+1} comme non minimal.
6. Incrémenter i et aller à 2.

Fin

Ainsi, pour chaque hinge (nœud de T), la jointure des arêtes qui le composent représentera une contrainte de CSP acyclique équivalent.

Exemple : soit le CSP P dont l'hypergraphe est représenté dans la figure II.7 suivante. Son hinge tree est représenté dans cette même figure à droite.

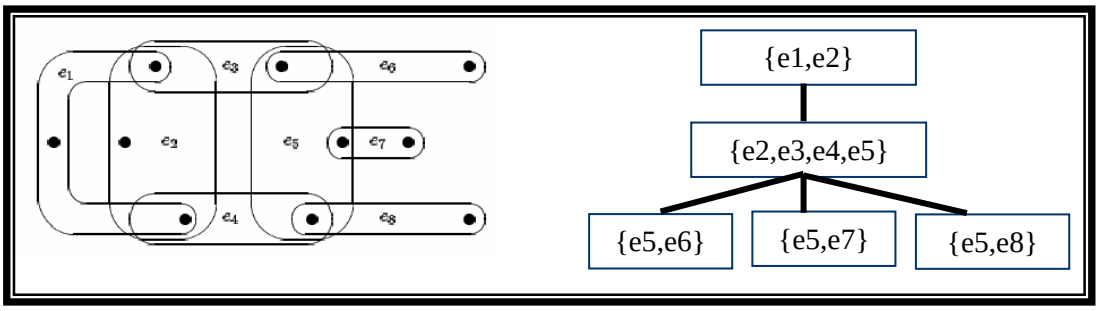

Fig.II.7. un hypergraphe et son hinge tree

La largeur induite par une hinge decomposition est égale à la taille (nombre d'arêtes) du plus grand noeud de l'hinge tree généré.

L'avantage principal de la hinge decomposition est l'efficacité de son algorithme de calcule de hinge tree qui réalise cette tache en $O(|V|\ |E|^2)$ tel que $|V|$ et $|E|$ représentent respectivement le nombre de variables et le nombre de contraintes de CSP original. Cette méthode a été souvent utilisée en combinaison avec d'autres méthodes.

2.5. La méthode Hinge decomposition + Tree clustering

Comme nous l'avons vu dans la section précédente, la hinge decomposition dispose d'un algorithme très efficace pour le calcul des hinge tree. Par contre l'inconvénient de cette méthode réside dans le faite que les hinges générés sont généralement trop grands et peuvent être décomposés à leur tour.

Cette constatation a motivé les auteurs de l'hinge decomposition à combiner celle-ci avec de nombreuses autres méthodes et notamment avec la tree clustering **[23]**.

Le principe de cette combinaison est d'utiliser la hinge decomposition en premier pour calculer des hinges puis d'appliquer la méthode tree clustering sur chaque hinge individuellement.

2.6. Hypertree decomposition:

L'hypertree decomposition **[24]** exploite, comme l'hinge decomposition, directement l'hypergraphe des CSP ce qu'il lui vaut d'être utilisé aussi bien pour les CSP binaires que pour les CSP n-aires. Elle se base sur le concept d'hypertree pour laquelle des algorithmes capables de la résoudre en un temps polynomial existes. Elle tente d'offrir un moyen pour pouvoir transformer un CSP ayant un hypergraphe quelconque en un CSP équivalent dont la structure est une hypertree. Ainsi, la largeur considérée est le nombre maximal d'hyper-arêtes contenu dans un nœud de l'hypertree decomposition.

Deux versions de l'hypertree decomposition existent, la simple et la généralisée. La première, grâce à une définition assez dure, a permit de développer des algorithmes capables d'affirmer ou d'infirmer d'une manière exacte l'existence d'une hypertree decomposition de largeur inférieur ou égale à un paramètre donnée. Alors que la deuxième comme son nom l'indique généralise la première en offrant une définition plus lâche et permet donc de construire des hypertree de largeur inférieur à celle de la première. Par contre, l'allégement de la définition rend l'hypertree decomposition généralisée calculable uniquement avec des algorithmes heuristiques.

Vu la richesse de cette méthode, nous avons préféré lui consacrer un chapitre entier. Le lecteur est donc invité à lire le chapitre III suivant pour de ample détailles.

2.7. Spread cut :

La spread cut **[25]** est issue d'une tentative d'uniformisation des méthodes existantes sous un même formalisme nommée la guarded decomposition qui est très proche de celui de l'hypertree decomposition. Ainsi, toutes les méthodes que nous avons vues précédemment seront modélisées sous forme d'une guarded decomposition qui doit vérifier certaines conditions supplémentaires propres à chaque méthode de décomposition. Et c'est pour mieux exploiter ce nouveau formalisme, qu'a été proposé la méthode Spread Cut.

Définition 2.9 : Un *guarded block* d'un hypergraphe H = (V, E) est un couple $<\lambda, \chi>$ tel que la *garde* $\lambda \subseteq E$ et le *block* $\chi \subseteq V$.

Définition 2.10 : Pour chaque CSP p et un *guarded block* b = <λ, χ> de son hypergraphe, la *relation générée* par P dans b est la projection sur χ de la jointure relationnelle des relation de P dont la contrainte est incluse dans λ.

Définition 2.11 : un guarded block <λ, χ> *couvre* une hyper-arête h si h ⊆ χ. Ainsi, un ensemble de guarded block de H est appelé *guarded cover* de H si chaque arête de H est couverte par un guarded block ; Celui-ci est appeler *complete guarded cover* si chaque arête de H été contenue dans la garde d'un guarded block qui la couvre.

Définition 2.12 : une *guarded decomposition* est une complete guarded cover. Ainsi il a été démontrer que pour un CSP P = <X,D,C,R> tel que l'ensemble Ξ est une *guarded decomposition* de l'hypergraphe de P, le CSP P' = <X,D,C',R'>, tel que C' est définit par les block des guarded block et R' est l'ensemble des relations généré par p dans les guraded bloc, sont équivalent (ils ont les même solutions).

Définition 2.13 : un *join tree* d'un ensemble Ξ de guarded block d'un hypergraphe H est un arbre T dont les nœuds sont les guarded block de Ξ tel que quelque soit une variable v qui apparaît dans deux block différent de Ξ, v doit apparaître dans chaque block de chemin unique reliant ces deux blocks dans T. Ainsi, un ensemble de guarded block est dit *acyclique* s'il possède un join tree.

Définition 2.14 : Soit H = (V, E) un hypergraphe et un ensemble X ⊆ V. une paire de sommets x, y est *X-connected* s'il existe une séquence d'hyper-arêtes $e_0,\ldots, e_m$ tel que $x \in (e_0 - X)$, $y \in (e_m - X)$ et $\forall i\ ((e_i \cap e_{i+1}) - X) \neq \emptyset$. Une *X-component* est un ensemble maximal C de sommets tel que chaque paire de sommets dans C est X-connected.

Définition 2.15: Soit H un hypergraphe et soit T un arbre dont les noeuds N sont des guarded blocks de H. pour chaque paire de noeuds adjacents n et n' de T, on définie la *n'-branche de T par rapport à n*, noté br_n (n') l'ensemble des nœuds de T dont le chemin (unique) vers n inclue n'. On définie aussi les sommets de br_n(n'), noté $\chi(br_n(n'))$, l'union des sommets inclus dans les blocks des nœuds de br_n(n') qui ne sont pas dans χ (n).

Définition 2.16: Un *decomposition tree*, T, d'un hypergraphe H est un join tree à racine de l'ensemble des guarded blocks de H qui satisfait les conditions suivantes:

1) Pour chaque arc (n, n') de T, et pour chaque arête e de H, si ($e \cap \chi(br_n(n')) \neq \emptyset$ alors e est couverte par un noeud de $br_n(n')$.
2) Pour chaque arc (n, n') de T, il existe une seule χ(n)-component $C_{(n,n')}$ de H tel que $\chi(br_n(n')) = C_{(n,n')}$

Cette dernière définition assure que pour chaque nœud n de l'arbre, chaque sous arbre enraciné dans un de ces fils est une χ(n)-component, ce qui limite donc l'ensemble des guarded block.

Définition 2.17 : Un guarded block b d'un hypergraphe H a une unbroken components si chaque χ(b)-component de H partage des sommets avec au plus une ($\cup\lambda$(b))-component de H.

Definition 2.18 : Soit λ un ensemble d'hyper-arêtes d'un hypergraph H = (V, E). On définie le label $L_\lambda(v)$, pour chaque sommet $v \in (\cup\lambda)$ comme une paire, dont la première composante est l'ensemble des ($\cup\lambda$)-components qui partage des variables avec une hyperarête contenant v, et la seconde composante l'ensemble des hyperarêtes de λ qui inclues v. Ainsi :

$L_\lambda(v)[1] = \{C / C$ est une ($\cup\lambda$)-component, $\exists e \in E, e \cap C \neq \emptyset, v \in e\}$

$L_\lambda(v)[2] = \{e \in \lambda / v \in e\}$

Définition 2.19 : on dit qu'un guarded block b = <λ, χ> respecte les labels si

$\forall v,w \in (\cup\lambda), (v \in \chi$ et $L_\lambda(w) = L_\lambda(v)) \Rightarrow w \in \chi$.

Cette définition permet aussi de réduire l'ensemble des guarded block en se limitant uniquement a ceux qui respectent cette condition.

Définition 2.20 : La Spread cut decomposition d'un hypergraphe H est une guarded cover acyclique Ξ qui possède un decomposition tree, et ou chaque guarded block dans Ξ a une unbroken components qui respecte les labels.

Exemple : Soit le CSP P = <X,D,C,R> tel que X={1,2,3,4,5,6,7,8,9,10} et C={c1, c2, c3, c4, c5, c6, c7, c8}tel que :

c1 = {1,2}	c2 = {2,3,9}	c3 = {3,4,10}	c4 = {4,5}
c5 = {5,6,9}	c6 = {6,7,10}	c7 = {7,8,9}	c8 = {1,8,10}

L'hypergraphe de P ainsi que son Spread Cut sont illustrés dans la figure II.8

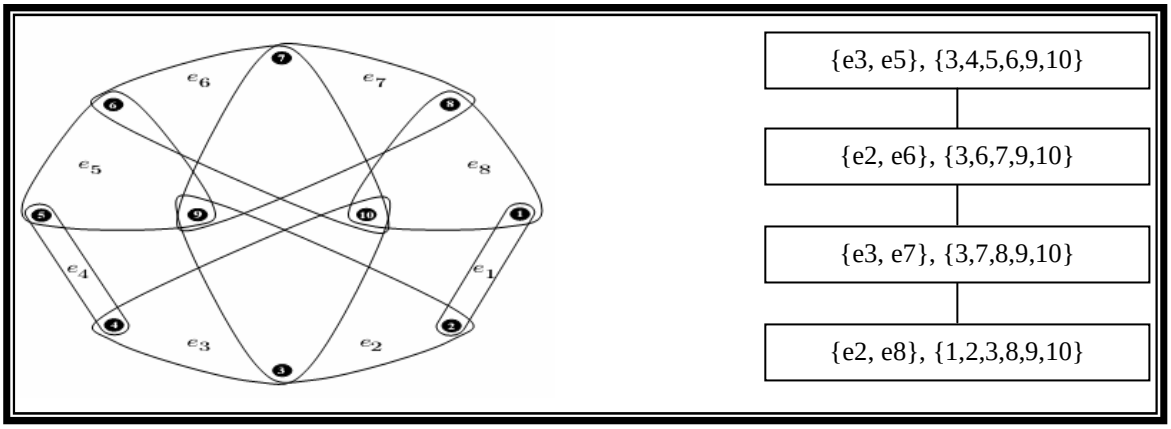

Fig.II.8. Hypergraphe de P et sa Spread Cut

La largeur induite par la Spread Cut decompositon est égale à la taille maximale des gardes (nombre d'arêtes). Cette nouvelle méthode, bien que elle est équivalent en termes de généralisation a l'hypertree decomposition et bien quelle offre pour certaines classes très particulières de CSP des largeurs de décomposition meilleurs, n'a pas encore montré son efficacité pratique ce qui est due notamment à un manque d'algorithmes capable de la calculer efficacement.

3. Comparaison

Une comparaison des différentes méthodes de décomposition structurelle a été réalisée par Gottlob et al dans **[26]**, une comparaison qu'ils ont mise à jour pour prendre en considération de nouvelles méthodes tel que la Spread cut.

Pour les besoins de la comparaison, des métriques ont été introduites

Définition 3.1 : pour n'import quel entier $k > 0$, la classe des hypergraphe k-tractable par rapport à une méthode de décomposition D noté C(D, k) est l'ensemble des hypergraphes dont la largeur obtenue après leur décomposition avec la méthode D est inférieur ou égale à k.

Définition 3.2 : une méthode de décomposition D2 généralise une autre méthode D1 noté $D1 \leq D2$, si $\exists\ \delta \geq 0$ tel que $\forall\ k \geq 1$, $C(D1, k) \subseteq C(D2, k + \delta)$. C'est-à-dire que chaque classe d'hypergraphe tractable en utilisant D1 reste tractable en utilisant D2.

Définition 3.3 : une méthode de décomposition D1 bat une autre méthode D2 noté D1 ► D2, si $\exists$ k ≥ 1 tel que $\forall$ m ≥ 0, C(D1, k) $\not\subset$ C(D2, m). C'est-à-dire qu'il existe une classe d'hypergraphe tractable en utilisant D1 mais pas tractable en utilisant D2.

Définition 3.4 : une méthode de décomposition D2 généralise fortement une autre méthode D1 noté D1 << D2, si D1 ≤ D2 et que D2 ► D1.

La figure suivante montre les résultats de cette comparaison ainsi une flèche relie une méthode D1 avec une autre méthode D2 si D1 <<D2.

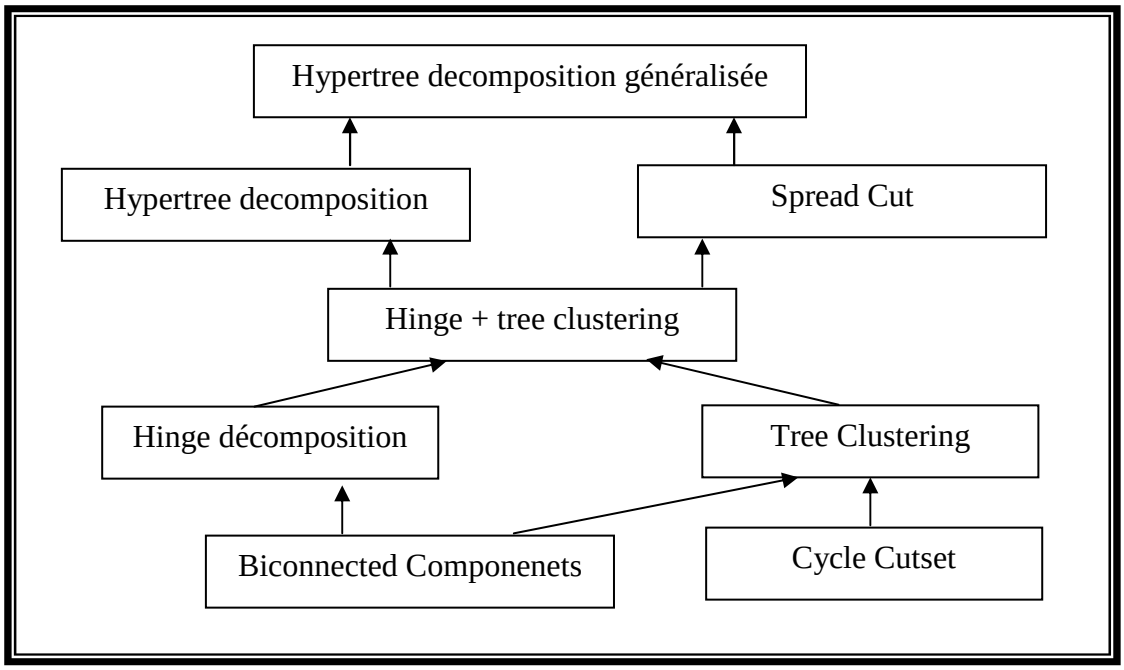

Fig.II.9. Comparaison entre les méthodes de décomposition

Conclusion:

Ces deux dernières décennies ont vues l'apparition de plusieurs méthodes de décomposition structurelle afin de pouvoir transformer n'import quel CSP en un CSP acyclique équivalent.

Unes de ces méthodes les plus intéressantes sont les deux versions de l'hypertree decomposition : la simple et la généralisée. Ces deux dernières méthodes sont les plus utilisées même si elle sont concurrencé par des méthodes équivalent tel que la *Spread Cut* et même plus générale tel que la *Fractional decomposition*. La large utilisation de l'hypertree decomposition dans ces deux versions au détriment des autres méthodes est due en partie à sa

définition a la fois simple et générale qui permet, comme nous allons le voire dans le chapitre suivant, la mise en œuvre de nombreux algorithmes pour la calculer.

Chapitre III

Hypertree decomposition

Ces deux dernières décennies ont vu l'apparition d'un bon nombre de méthodes de décomposition dont le but est de transformer n'importe quel CSP en un CSP acyclique et de borner ainsi la complexité théorique de leur résolution. Dans le chapitre précédent, nous avons vu quelque unes des méthodes de décomposition les plus intéressantes de la littérature et nous avons vu que certaines d'entre elles sont plus générales que d'autres relativement à des mesures de cyclicité plus optimales.

L'hypertree decomposition et sa variante l'hypertree decomposition généralisée sont les plus générales. Malgré l'apparition de méthodes plus récentes telles que la Spread Cut et la Fractional decomposition, l'Hypertree decomposition conserve ses avantages grâce à l'existence de nombreux algorithmes exacts et heuristiques qui la calculent.

1. L'hypertree :

Un ***sous graphe d'arc*** d'un graphe G = (V, E) est un graphe G' = (V', E') qui a le même ensemble de sommets que G (V' = V) et dont l'ensemble des arêtes est un sous ensemble de G (E' $\subseteq$ E).

Un sous graphe d'arcs de graphe dual d'un hypergraphe respecte la propriété de ***connexion*** si pour tout couple de nœuds (contraintes) qui partage une variable, il existe un chemin qui les relie la contenant. Si de plus ce sous graphe est un arbre alors celui-ci est appelé un **join-tree**.

Un hypergraphe dont le graphe dual possède un join-tree est appelé **Hypertree [34]**.

2. L'hypertree decomposition

L'hypertree decomposition est la première méthode de décomposition qui se base sur le concept d'hypertree. Celle-ci, contrairement aux méthodes proposées avant elle, manipule directement l'hypergraphe de CSP sans passer par le graphe primal ou le graphe dual ce qui explique qu'aujourd'hui, l'hypertree decomposition est considérée comme une avancée majeure dans le domaine de la décomposition structurelle des CSP.

Définition 1.1 : Une tree décomposition **[34]** d'un hypergraphe H = (V(H),E(H)) est un couple

<T, χ> tel que T = (V(T), E(T)) est un arbre et χ est une fonction d'étiquetage qui associe à chaque nœud t de T un ensemble de variables $\chi(t) \subseteq V(H)$ et qui respecte les conditions suivante :

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

Définition 1.2 : Une hypertree decomposition généralisée **[24]** d'un hypergraphe H = (V(H),E(H)) est un triplet <T, χ, λ> tel que T = (V(T), E(T)) est un arbre, et χ et λ sont des fonctions d'étiquetage définies comme suit :

$\chi : V(T) \rightarrow 2^{V(H)}$ qui associe à chaque nœud t de T un ensemble de sommets $\chi(t) \subseteq V(H)$

$\lambda : V(T) \rightarrow 2^{E(H)}$ qui associe à chaque nœud t de T un ensemble d' arêtes $\lambda(t) \subseteq E(H)$

Et qui vérifient les contraintes suivantes :

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

La largeur d'une hypertree decomposition généralisée est la cardinalité maximale des $\lambda(t)$

Définition 1.3 : Une hypertree decomposition **[24]** d'un hypergraphe H = (V(H),E(H)) est une hypertree decomposition généralisée HD = $<T, \chi, \lambda>$ de H qui vérifie la condition suivante : $\forall t \in T : \chi(T_t) \cap (\cup\lambda(t)) \subseteq \chi(t)$ tel que T_t est le sous arbre de T enraciné en t.

Une hypertree decomposition est dite complète si $\forall h \in E(H)$, il existe $t \in T$ tel que $h \subseteq \chi(t)$ et $h \in \lambda(t)$.

Exemple : la figure III.1 illustre un hypergraphe H composé de 19 sommets et de 15 hyper-arêtes. Son hypertree decomposition généralisée et Son hypertree decomposition son illustrées par la figure III.2. Chaque nœud est composé d'un couple (λ,χ) où λ est un ensemble d'hyper-arêtes et χ est un ensemble de variables. Dans cette figure, on voit bien que l'hypertree decomposition généralisée viole la condition spéciale, car la variable 13 a été enlevée du nœud dont λ = {h10, h14} et qui apparaît dans le sous arbre enraciné en celui-ci. On voit aussi que la condition spéciale a entraîné une augmentation de la largeur (3 contre 2 sans la condition).

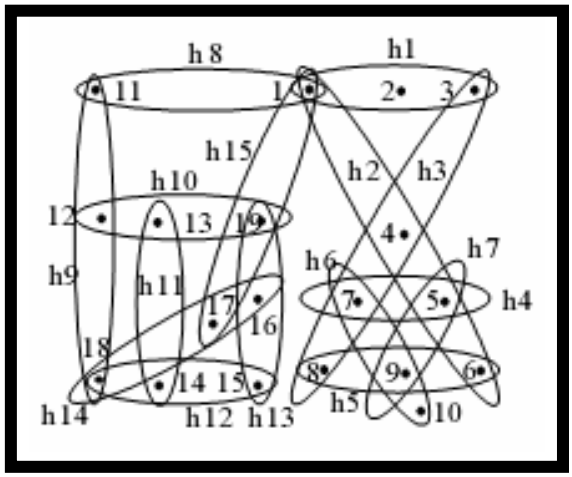

Fig.III.1. l'hypergraphe H

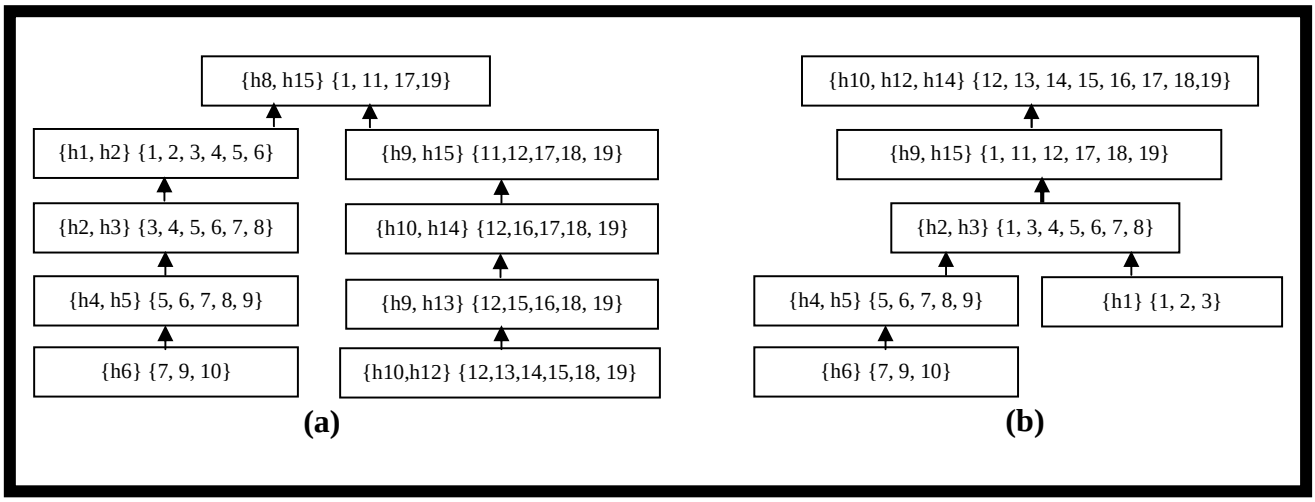

Fig.III.2. (a) l'hypertree decomposition généralisée **(b)** l'hypertree decomposition de H

3. Forme normale de l'hypertree decomposition :

Afin de permettre le calcul de l'hypertree decomposition, une nouvelle définition de celle-ci a été proposée sous le nom de l'hypertree decomposition normalisée.

Soit H = (V, E) un hypergraphe et soit V' $\subseteq$ V un ensemble de sommets et *a*, *b* deux sommets de V. *a* est dit [V']-adjacent à *b* si il existe une arête h $\in$ V tel que {a, b}$\in$ (h – V'). Un [V']-chemin de a à b est une séquence $a=a_0, a_1, \ldots, a_t=b$ tel que chaque sommet a_i soit [V']-adjacent à a_{i+1} pour i= 0 à t-1. Un ensemble W $\subseteq$ V est dit [V']-connected si pour chaque couple de sommets a, b $\in$ W il existe un [V']-chemin entre a et b. Une [V']-component est un ensemble maximal non vide W $\subseteq$ (V –V') de sommets qui soit [V']-connected.
Une k-vertex est n'importe quel ensemble de k hyper-arêtes au maximum.

Enfin, quel que soit un ensemble de sommet C, soit arêtes(C) = {h $\in$ E / (h $\cap$ C) $\neq \emptyset$}

Une hypertree decomposition généralisée HD = <T, χ, λ> est dite en forme normale si pour chaque nœud r de T et pour chaque fils s de r les conditions suivantes sont vérifiées :

1) Il y'a exactement une [χ(r)]-component Cr tel que : $\chi(Ts) = Cr \cup (\chi(r) \cap \chi(s))$
2) $\chi(s) \cap Cr \neq \emptyset$ où Cr est la [χ(r)]-component qui vérifie la condition 1
3) $(\cup\lambda(s)) \cap \chi(r) \subseteq \chi(s)$

Les auteurs de l'hypertree decomposition ont montré que pour chaque hypertree decomposition de largeur *hw* il existe une hypertree decomposition en forme normale de même largeur *hw*. De plus, à partir de la définition de la forme normale, il est possible de dériver à partir des seuls ensembles λ les ensembles χ. C'est sur ce dernier résultat en particulier que les auteurs de l'hypertree decomposition se sont basés pour proposer des algorithmes exacts de calcul de l'hypertree decomposition.

4. Calcul de l'hypertree decomposition

Un des avantages de la méthode de l'hypertree decomposition par rapport aux autres méthodes de décomposition structurelle est l'existence d'un nombre assez conséquent de méthodes permettant de la calculer. Ces méthodes peuvent être classées en deux catégories : les méthodes exactes et les méthodes heuristiques.

4.1. Méthodes exactes :

La première caractéristique que doit vérifier une méthode de décomposition est de pouvoir, pour chaque paramètre k fixé, déterminer l'existence ou non d'une décomposition de largeur inférieure ou égale à k. A cet effet, plusieurs méthodes ont été proposées pour le calcul de l'hypertree decomposition de largeur inférieure ou égale à un paramètre k donné si elle existe ou d'affirmer l'inexistence d'une telle décomposition. Celles-ci sont basées sur la définition de la forme normale de l'hypertree decomposition.

4.1.1. L'algorithme k-decomp

C'est le premier algorithme de calcul de l'hypertree decomposition **[24],** proposé dans le but de démontrer qu'il été possible d'affirmer pour n'importe quelle constante k donnée, s'il existe une hypertree decomposition de largeur ≤ k ou non. Il se base sur la définition de la forme normale pour pouvoir choisir les nouveaux noeuds de l'hypertree un à un.

Algorithme k-decomp
Entrée un Hypergraphe H = (V, E)
Sortie accepté si H a une hypertree decomposition de largeur ≤ k, **rejeté** sinon
Début
Si k-decomposable(V, ∅) **=** accepté **alors** accepté

Sinon rejeté
Fsi
Fin
Procédure k-decomposable(ensemble de sommets Cr, ensemble d'hyper-arêtes R)
Début
Choisir aléatoirement une k-vertex $S \subseteq E$
Si $(\forall P \in$ arêtes(Cr)$)\ (P \cap \cup R) \subseteq \cup S)$ et $(\cup S \cap Cr \neq \emptyset)$ **alors**
Soit CE = {C ⊆ Cr / C est une [∪S]-component}
Pour chaque C ∈ CE **si** k-decomposable(C, S) = Rejeté **alors retourner** Rejeté
Fait
Retourner Accepté
Sinon Retourner Rejeté
Fsi
Fin

L'algorithme k-decomp tente de trouver une k-vertex S qui décompose récursivement le problème. S doit vérifier une condition, dite de ***connectivité*** :
$(\forall P \in$ arêtes(Cr)$)\ (P \cap \cup R) \subseteq \cup S)$.

De plus, pour que le problème à décomposer se réduit à chaque itération une autre condition dite de ***monotonicité*** doit être vérifiée : $(\cup S \cap Cr \neq \emptyset)$.

L'inconvénient majeur de cet algorithme est son non déterminisme. Son intérêt est purement théorique. D'autres algorithmes déterministes ont été proposés en reprenant le principe de k-decomp.

4.1.2. L'algorithme Opt-k-decomp

L'algorithme opt-k-decomp **[27]** est le premier algorithme déterministe apparu dans la littérature. Il a pu pallier au non déterminisme de l'algorithme k-decomp en traitant toutes les combinaisons possibles de k-vertex. Outre son caractère déterministe, il permet aussi, dans le cas où une hypertree decomposition de largeur inférieure à k existe, de trouver celle dont la largeur est optimale.

Algorithme opt-k-decomp
Entrée: un hypergraphe H = (V, H)
Sortie: une hypertree decomposition HD de H de largeur optimale si elle existe
Début
CG = (Ne U Na, A, poids): graph valué dirigé
HD = (T, χ, λ): hypertree de H;
compute-CG();
weight-CG();

```
        Si poids((racine, V) = ∞ Alors  Retourner Echec; // pas de décomposition de largeur ≤ k
        Sinon                  // la décomposition existe, calculer l'hypertree
                Créer racine' dans T;
                χ (racine') = ∅; λ( racine') = ∅;
                Compute-hypertree((racine, V), racine')
    Retourner HD;
Fin

Procedure Compute-CG
        Ne = {(root, V)} ∪ {(R, C) / R est une  k-vertex et C une [R]-component};
        Na = ∅; A = ∅;
        Pour chaque (R, C)∈ Ne Faire
                Soit rc = (∪h ⊂ arêtes (C) var(h)) ∩ var(R);

                Pour chaque k-vertx S Faire
                        Si variables(S) ∩ C ≠ ∅ et rc ⊆ var(S) Alors
                                Na = Na U{(S,C)};
                                Ajouter un arc de (S, C) vers (R, C) dans A;
                                Pour chaque (S, C')∈ Ne tel que C' ⊆ C Faire
                                        Ajouter un arc de (S, C') vers (S, C) dans A;
                                Fait
                        Fsi
                Fait
        Fait
Fin

Procedure Weight-CG
        Poids ((R, C)) = ∞, pour chaque (R,C)∈ Ne  qui n'a pas d'arc entrant dans CG;
        Poids ((S, C)) = |S|, pour chaque (S,C)∈ Ne  qui n'a pas d'arc entrant dans CG;
        Pour chaque p∈ NeU Na, soit |p| le nombre d'arcs entrant vers p d'un nœud sans poids
        Fait
        Tant que il y'a des noeuds sans poids dans Ne U Na Faire
                Soit p = (S, C) un noeud sans poids tel que |p| = 0;
                Si p∈ Ne Alors
                        Poids(p) = min({Poids (q) | (q,p)∈ A});
                Sinon
                        Poids(p) = max({|S| }U { Poids (q)| (q,p)∈ A});
                Fsi
        Fait
Fin

Procédure Compute-hypertree (p : un noeud de GC, r : un nœud de HD)
        Choisir un prédécésseur (S, C) de p de poids minimum dans GC
        Créer un noeud s fils de r dans T
        λ(s) = S; χ(s) = var(S) ∩ (C ∪ χ(r))
        Pour chaque prédécésseur q de (S, C) faire Compute-hypertree (q, s);
Fin
```

L'algorithme opt-k-decomp se compose de quatre procédures essentielles. Le programme principal vérifie l'existence d'une décomposition de largeur inférieure ou égale à *k* en appelant les procédures *compute-CG* et *weight-CG* et dans le cas où cette décomposition

existe, il fait appel à la méthode *Compute-hypertree* pour calculer l'hypertree optimale. La procédure *Compute-CG* crée deux ensemble *Ne* et *Na*, le premier représente les sous problèmes qui peuvent se présenter lors de la décomposition. Chacun d'eux peut être relié à un ou plusieurs séparateurs possible dans *Na* capables de le décomposer. La procédure *weight-CG*, quant à elle, affecte des poids à chaque élément de Ne qui représentent la largeur maximale des nœuds de l'hypertree jusqu'à son niveau. A la fin, si le poids de la racine est égal à ∞ alors aucune décomposition de largeur inférieure ou égale à *k* n'existe. Dans le cas où le poids de la racine est un entier, celui-ci représente la largeur optimale de la décomposition et la procédure *Compute-hypertree* est appelée pour calculer les nœuds de l'hypertree decomposition.

4.1.3. L'algorithme Red-k-decomp

Le principal reproche qu'on peut faire à l'algorithme *opt-k-decomp* est la prise en considération de tous les k-vertex alors que certaines d'entre elles n'apparaîtront jamais dans une décomposition de largeur optimale. Evidemment le traitement de ces k-vertex inutiles va induire un surcoût en terme de temps CPU et en terme d'occupation mémoire.

C'est à partir de ce constat que la méthode *red-k-decomp* **[28]** a été introduite et propose un ensemble de lemmes caractérisant le fait qu'une k-vertex soit présente dans une décomposition de largeur optimale. Ces lemmes sont regroupés pour définir la forme normale réduite.

Définition 4.1 : Une hypertree decomposition $<T, \chi, \lambda>$ d'un hypergraphe H = (V, E) est en forme normale réduite (RNF) si elle est en forme normale et si pour chaque nœud p de T :

1) Les contraintes dans $\lambda(p)$ ne couvrent pas complètement les contraintes du nœud parent de p :

   $\forall p \in T \; var(\lambda(parent(p))) \not\subset var(\lambda(p))$

2) Chaque contrainte c de $\lambda(p)$ contient une variable qui n'est couverte par aucune autre contrainte de $\lambda(p)$ et qui est adjacente à une autre variable non couverte par c :

   $\forall c \in \lambda(p), \exists v \in var(c) : (\forall c' \in \lambda(p)\text{-}\{c\} \; v \notin var(c')$) et ($\exists v' \in V, \exists h \in E :$ $v' \notin var(c)$ et $\{v, v'\} \subseteq var(h)$)

3) Si il n'y a qu'une seule [$var(\lambda(p))$]-component alors p est un nœud feuille.

La définition de la RNF limite l'ensemble des k-vertex qui peuvent participer à la décomposition ainsi que la place qu'ils vont tenir dans l'Hypertree. L'idée de base est que seuls les nœuds qui décomposent le CSP en plusieurs sous composantes peuvent apparaître dans le corps de l'Hypertree. Si ce n'est pas le cas alors ce nœud peut apparaître uniquement comme nœud feuille.

4.1.4. L'algorithme Det-k-decomp

L'algorithme *det-k-decomp* **[29]** est une autre approche qui tente de palier au non déterminisme de *k-decomp* qui consiste à remplacer la phase de choix aléatoire de *k-decomp* par une recherche basée sur l'algorithme backtrack. De plus, l'algorithme *Det-k-decomp* garde un historique de ses manipulations afin d'optimiser le temps de décomposition en évitant le traitement des situations qui ont déjà été traitées.

Algorithme Det-k-decomp
Entrée : Un hypergraphe H = (V, E)
Sortie : une hypertree decomposition HTree
Début
    FailSeps = ∅
    SuccSeps = ∅
    HTree = DecompCov(E, ∅) ;
    Si HTree ≠ NULL **Alors** Expand(HTree)
Fin
Procédure DecompCov(aretes**,** conn**)**
Début
    Si |arêtes| ≤ k **alors**
        HTree = getHTNode(aretes, ∪ aretes, ∅)
        Retourner HD
    Fsi
    BoundEdges = {e ∈ E | e ∩ Conn ≠ ∅};
    Pour chaque CovSep ∈ cover (Conn, BoundEdges) **Faire**
        HTree = decompAdd(aretes, Conn, CovSep);
        Si HTree ≠ NULL **Alors Retourner** HTree;
    Fait
    Retourner NULL
Fin
Procédure DecompAdd(aretes**,** Conn, CovSep**)**
Début
    InCovSep = CovSep ∩ aretes;
    Si InCovSep ≠ ∅ **Ou** k − |CovSep| > 0 **Alors**
        Si InCovSep = ∅ **Alors** AddSize = 1 **Sinon** AddSize = 0 **Fsi**;
        Pour chaque AddSep ∈ aretes **tel que** |AddSep| = AddSize **Faire**
            Separator = CovSep ∪ AddSep;

```
                    Components = separate(aretes, Separator );
                    Si ∀ Comp ∈ Components.(Separator, Comp) ∉ FailSeps Alors
                        Subtrees = decompSub(Components, Separator );
                        Si Subtrees ≠ ∅ Alors
                            Chi = Conn ∪ [∪(InCovSep ∪ AddSep)];
                            HTree = getHTNode(Separator ,Chi , Subtrees );
                            Retourner HTree;
                        Fsi
                    Fsi
                Fait
        Fsi
Retourner NULL;
Fin
Algorithme DecompSub(Components, Separator )
Début
        Subtrees = ∅;
        Pour chaque Comp ∈ Components Faire
            ChildConn = (∪Comp) ∩ (∪Separator) ;
            Si (Separator, Comp) ∈ SuccSeps Alors
                HTree = getHTNode(Comp, ChildConn, ∅);
            Sinon
                HTree = decompCov(Comp, ChildConn);
                Si HTree = NULL Alors
                    FailSeps = FailSeps ∪ {(Separator, Comp)};
                    Retourner ∅;
                Sinon
                    SuccSeps = SuccSeps ∪ {(Separator, Comp)};
                Fsi
            Fsi
            Subtrees = Subtrees ∪ {HTree};
        Fait
        Retourner Subtrees;
Fin
```

L'algorithme *det-k-decomp* se compose essentiellement de quatre procédures. La procédure principale lance la décomposition en appelant la procédure *DecompCov***.** Dans le cas où une décomposition existe, il la complète. Cette procédure déclare les deux ensembles *SuccSeps* et *FailSeps* qui contiendront des couples de séparateurs (k-vertex) et leur composante. Le premier ensemble définit les couples dont la décomposition est valide et le deuxième ceux qui ne le sont pas afin de mémoriser d'un coté un historique des échecs et d'un autre coté pouvoir réduire la taille de la recherche en omettant de traiter les cas valides qui se répètent et les compléter à la fin de l'algorithme par simple copiage à l'aide de la méthode *Expand*. La procédure *DecompCov* applique la condition de connectivité, définie dans l'algorithme *k-decomp*, à l'aide de la procédure *Cover*. Puis celle-ci appelle la procédure

DecompAdd qui, elle, vérifie la condition de monotonicité vue dans l'algorithme *k-decomp* puis calcule les sous problèmes issus de l'application du séparateur choisi et ceci à l'aide de la fonction *Separate*. Enfin, la procédure *DecompSub* est appelée pour vérifier si les sous problèmes ont déjà été traités. Si c'est le cas et si ces sous problèmes ont débouché sur un succès alors ceux ci seront marqués pour être complétés à la fin de l'algorithme avec la fonction *Expand*. Dans le cas contraire, un retour arrière est effectué directement alors que dans le cas où un sous problème n'a pas été traité précédemment la procédure *DecompCov* est rappelé récursivement pour le décomposer.

Cet algorithme est résumé par la figure III.3 suivante.

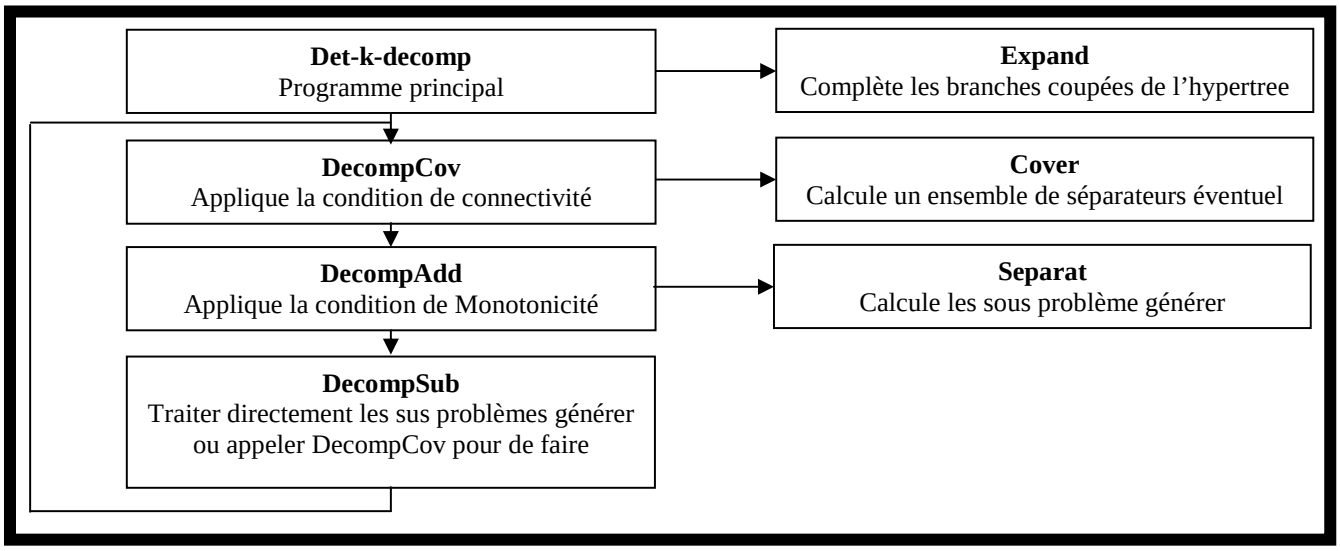

Fig.III.3. Résumer de l'algorithme Det-k-decomp

4.2. Méthodes heuristiques

Les méthodes exactes sont très intéressantes car elles permettent de calculer, directement ou indirectement des décompositions de largeur optimale. Cependant, elles exigent un coût important en termes de temps CPU et place mémoire. Ce qui les rend rapidement obsolètes dés qu'il s'agit de décomposer des problèmes de grande taille.

Pour remédier à ce problème de nouvelles méthodes ont été proposées afin de pouvoir calculer des hypertree decomposition en un temps raisonnable. Comme ces méthodes n'offrent aucune garantie sur la largeur de la décomposition, celles-ci sont appelées méthodes heuristiques.

De plus, comme celle-ci ne se basent pas sur le principe de k-decomp, elles ne se soucient en général pas de la 4éme condition de l'hypertree decomposition et sont donc un moyen pour calculer l'Hypertree decomposition généralisée.

4.2.1. L'algorithme Bucket Elimination (BE)

La méthode *bucket Elimination* **[30]** est la méthode la plus utilisée pour le calcul de l'hypertree decomposition généralisée. C'est une adaptation d'une méthode utilisée pour le calcul de la tree decomposition à laquelle on a rajouté une procédure de couverture des sommets pour permettre la déduction des ensembles λ à partir des ensembles χ.

BE utilise un ordre *d* sur les variables de CSP. Plusieurs ordres sur les variables existent. On peut citer les ordres MCS (pour *Minimum Carinality Search*), MIW (pour *Minimum Induced Width)* et Min Fill (pour *Minimum Fill).*

La deuxième phase consiste à créer un sceau (bucket) pour chaque variable et ensuite de placer les variables de chaque contrainte dans le sceau (ensemble χ) de sa variable la plus haute dans *d*. A partir de là, les sceaux sont traités dans l'ordre *d*. Ainsi après avoir traité le sceau B_{vi} de la variables v_i, on calcule $A = \chi(B_{vi}) - \{ v_i \}$. A est alors mis dans le sceau B_{vj} de sa variable la plus haute v_j. De plus, B_{vi} et B_{vj} sont reliés par une arête. A la fin, on obtient une tree decomposition où les sceaux et les arêtes ajoutées forment un arbre et les variables contenues dans chaque sceaux représentent les ensembles χ.

Enfin, pour transformer la tree décomposition en hypertree decomposition généralisée il suffit juste de calculer les ensembles λ pour respecter la condition $\forall t \in T : \chi(t) \subseteq (\cup\lambda(t))$ et en tentant de minimiser la largeur par exemple en affectant une fonction coût a chaque contrainte et de couvrir les ensembles χ avec des contraintes de coût minimum.

Algorithme Bucket Elimination
Entrée : Un hypergraphe H = (V, E) et un ordre d = (v1,…, vn) des variables dans V
Sortie : Une hypertree decomposition généralisée GHD = <T = (N, ET), χ, λ>
Début
  Pour chaque variable vi $\in$ V, lui affecter un sceau Bvi, χ (Bvi) = $\varnothing$; N = N $\cup$ {Bvi}
  Pour chaque arête h $\in$ E **faire**
    Soit v la variable de h la plus haute dans d
    χ (Bvi) = χ (Bvi) $\cup$ h
  Fait
  Pour i = n à 2 faire
    Soit A = χ (Bvi) – { v_i }
    Soit v_j la plus haute variable de A dans d
    χ (Bvj) = χ (Bvj) $\cup$ A
    ET = ET $\cup$ (Bvi, Bvj)
  Fait
  Pour chaque Bvi $\in$ N : λ(Bvi) = couvrir(χ(Bvi))
Fin
Couvrir(C : ensemble de variable) // *retourne un ensemble minimale d'arêtes de E qui contient toutes les variables de C.*

4.2.2. L'algorithme Dual Bucket Elimination (DBE):

L'algorithme *Dual Bucket Elimination* **[30]** est très similaire à l'algorithme BE, mais au lieu de travailler avec les variables pour construire d'abord l'ensemble χ, la DBE se base sur les contraintes. La BE de par son mode de fonctionnement tente de minimiser la cardinalité des ensembles χ, mais la largeur de décomposition dépend de la taille des ensembles λ. C'est à partir de ce constat que la méthode DBE a été proposée en appliquant les même étapes que ceux de la BE mais en utilisant le graphe dual de CSP au lieu de son hypergraphe. Ainsi, les contenus des sceaux représenteront les ensembles λ et on finira par calculer les ensembles χ.

4.2.3. Les Algorithmes génétiques :

Le problème de l'algorithme BE est le faite qu'il soit sensible à l'ordre des variables. Ce dernier étant aléatoire, l'algorithme BE doit être utilisé plusieurs fois pour trouver une décomposition de bonne largeur. Pour palier à ce problème, les algorithmes génétiques **[31]** ont été utilisés pour tenter de trouver le meilleur ordre des variables qui va générer une largeur minimale.

Les algorithmes génétiques font partie des méta heuristiques utilisées pour résoudre des problèmes d'optimisations en imitant le principe de l'évolution chez les espèces vivantes. Nous commençons par rappeler un ensemble de concepts utilisés dans cette technique :

- *Population* : ensemble de solutions candidates
- *Individu* : une solution candidate
- *Chromosome* : ensemble de paramètres déterminant les propriétés d'une solution
- *Géne* : un paramètre
- *Mutation :* un changement de valeur d'un ou de plusieurs gène d'un individu.
- *Croisement* : remplacement de la valeur d'un ou de plusieurs gène d'un individu par celle d'un autre individu.

Pour commencer, l'algorithme génétique crée une population qui contient des individus créés aléatoirement ou d'une manière heuristique. Puis chaque individu sera évalué et se verra affecter une valeur (fonction fitness). La population va évoluer en un certain nombre de générations jusqu'à ce que un critère d'arrêt soit satisfait. A chaque étape, un

ensemble d'individus est sélectionné puis ceux-ci sont croisés entre eux puis mutés pour créer la génération suivante.

Cette méthode, tente de trouver un ordre optimal (qui va donner une largeur de décomposition minimum) des variables pour l'algorithme BE. Ainsi, l'algorithme génétique va générer une population correspondant à un nombre défini d'ordres de variables possibles. La fonction coût de chaque ordre sera la largeur générée par l'algorithme BE. Puis, à chaque étape un ensemble d'individus seront sélectionnés et subiront une mutation et un croisement pour générer la population suivante. Une fois le bon ordre trouvé, celui-ci sera utilisé par l'algorithme pour calculer une hypertree decomposition généralisée comme nous l'avons vu plus haut.

Conclusion

L'hypertree decomposition reste une des méthodes de décomposition les plus intéressantes car les largeurs induites restent inférieures à presque toutes les autres méthodes de la littérature.

De plus, un grand nombre d'algorithmes exacts ont été proposés pour le calcule des hypertree decomposition de largeur optimale. Mais comme nous avons pu le voire dans leur description, les algorithmes exacts induisent généralement un coût en temps CPU et en consommation mémoire très important ce qui les rend inutilisables pour décomposer des problèmes de grande taille. C'est pourquoi la majeure partie des recherches actuelles est orientée vers la proposition d'algorithmes heuristiques qui offrent un compris entre la largeur de décomposition et le temps de calcul. Ceux ci peuvent, par la suite, être utilisés directement ou exploités pour définir rapidement une première borne de la largeur et utiliser cette dernière comme paramètre pour une méthode exacte.

Les méthodes heuristiques qui ont été proposées jusqu'à aujourd'hui sont généralement basées sur la tree decomposition. C'est la raison pour laquelle nous avons orienté notre travail vers la recherche d'une nouvelle heuristique pour le calcul de l'hypertree decomposition.

Chapitre IV

Construct & Reduce : Une nouvelle heuristique pour le calcul de l'hypertree decomposition

Le calcul d'une hypertree decomposition de taille optimale se fait de deux manières :

- soit en utilisant un algorithme qui à partir d'un paramètre k, calcule toutes les décompositions possibles de largeur inférieure ou égale à k et sélectionne celle dont la largeur et la plus petite. On peut citer pour ce cas les algorithmes Opt-k-decomp **[27]** et Red-k-decomp **[28]** qui calculent tous les deux une hypertree decomposition optimale.
- soit en utilisant un algorithme qui à partir du même paramètre k, calcule une décomposition de largeur $\leq$ k. On peut citer ici les algorithmes k-decomp **[24]** et det-k-decomp **[29]** qui sont utilisés de façon itérative pour calculer une hypertree decomposition de largeur optimale.

Ces méthodes dites méthodes de décomposition exactes, sont très intéressantes dans le sens où elles permettent d'optimiser la largeur qui est le paramètre crucial de l'hypertree decomposition. Cependant ces méthodes ont un coût trop important en terme de temps de calcul et d'espace mémoire et de ce fait elles deviennent rapidement obsolètes dès qu'il s'agit de décomposer des problèmes de taille importante.

Pour palier aux inconvénients des méthodes exactes, des heuristiques ont été développées pour calculer une hypertree decomposition de CSPs même de grande taille en un temps acceptable mais sans garantir la condition d'optimalité. La décomposition ainsi obtenue peut être, soit utilisée directement pour la résolution, soit exploitée pour calculer une décomposition optimale en considérant la largeur obtenue comme paramètre d'une méthode de décomposition exacte. Nous présentons dans ce chapitre une nouvelle heuristique que nous avons proposé pour le calcul de l'hypertree decomposition.

1. Notre heuristique: Construct & Reduce

Pour tenter d'apporter une solution au calcul de l'hypertree decomposition de problèmes de grandes tailles, nous proposons une nouvelle heuristique, originale à notre connaissance, qui se base sur le principe de la réduction de la largeur après construction d'une première hypertree decomposition, alors qu'en général, les méthodes existantes tentent plutôt de réduire la largeur pendant l'étape de construction. A cet effet, nous avons restreint la définition originale de l'hypertree decomposition en lui rajoutant une nouvelle condition permettant de garantir que le résultat de la réduction reste une hypertree decomposition. Ainsi la condition 3 de l'hypertree decomposition **[**Voir définition 1.3, chapitre III**]** qui stipule que $\chi(t) \subseteq \cup\lambda(t)$ est remplacée par $\chi(t) = \cup\lambda(t)$. De cette manière, la 4ieme condition ($\cup\lambda\ (t) \cup \chi(Tt) \subseteq \chi(Tt)$) sera forcément vérifiée si la 3éme l'est. Donc les algorithmes que nous allons proposer doivent juste maintenir la nouvelle conditions 3 et la 2eme de l'hypertree decomposition vérifiées.

Notre méthode pour la construction de l'hypertree decomposition initiale consiste à extraire des hypertrees decomposition de largeur et de profondeur 1 à partir du CSP initial puis de les fusionner pour obtenir une hypertree decomposition de CSP. Les algorithmes que nous proposons pour réaliser cette tâche sont polynomiaux. Les temps de calcul que nous obtenons en pratique sont négligeables même pour des problèmes de grande taille.

Pour résumer, notre démarche se décompose en trois étapes:

- L'extraction des hypertrees decomposition incluses dans le CSP initial.
- La fusion de ces hypertrees decomposition pour former un seul hypertree.
- La réduction de la taille des nœuds de l'hypertree decomposition pour minimiser la largeur.

Dans les paragraphes qui vont suivre, nous allons détailler chacune de ces étapes en vérifiant leur correction, avant de finir par une étude expérimentale de notre approche.

2. Extraction des Hypertrees

La première phase de notre démarche consiste à extraire les sous ensembles d'arêtes qui forment des hypertrees decomposition de largeur et de profondeur 1. Chacune d'elle sera composée uniquement de la racine et de ses fils (arbre de profondeur 1). L'ensemble λ de chaque nœud contiendra exactement une arête et l'ensemble χ se composera des variables de l'arête appartenant à λ, afin de vérifier les 3éme et 4éme conditions de l'hypertree decomposition. Après le choix des racines, les fils seront choisis de manière à ne pas violer la 2éme condition de l'hypertree decomposition ceci en s'assurant que deux fils d'un même père ne partagent jamais de variables (de l'ensemble χ) qui ne soient pas contenues dans l'ensemble χ de leur père comme nous allons le voir plus tard (condition 3 de théorème 1).

L'étape d'extraction d'hypertree est décrite formellement par l'algorithme1.

Algorithme1 : Extraction des hypertrees decomposition
Entrée : H = (V, E) : hypergraphe de CSP originale
Sortie : {HTi = (Ti, χi, λi) / i = 1 à k} ensemble de k hypertrees
Début
    i = 0 ;
    Tant que Non vide(E)
        i = i+1
        Racine = **choisir**(E)
        HTi = **Nouvelle_Hypertree()**
        N = **Nouveau_Noeud(**HTi**)**
        λi(N) = Racine
        χi(N) = var(Racine)
        Nogoods = $\varnothing$
        Décompose (Racine, HTi**)**
    Fait
Fin

Void décompose (**Arête** Racine, **Hypertree** HT)
Début
    E = E – Racine
    Tant que ($\exists$e $\in$ E / var(e) $\cap$ var(Racine) $\neq \varnothing$ **et** var(e) $\cap$ Nogoods = $\varnothing$) **faire**
        Courrant = **Choisir** (E, Racine)
        N = **Nouveau_Noeud(**HT**)**
        λ (N) = courant
        χ (N) = var(Courrant)
        Nogoods = Nogoods $\cup$ var(Courrant) - var(racine)
        E = E – Courrant
    Fait

Hypertree Nouvelle_Hypertree() // crée une hypertree vide

Noeud Nouveau_Noeud(**Hypertree** HT) //crée un nouveau noud pour l'hypertree HT

Arête choisir (**Ensemble** E, **Arête** e)
// Retourne une arête e' ∈ E tel que var(e) ∩ var(e') ≠ ∅ et var(e) ∩ Nogoods = ∅)

Arête choisir (**Ensemble** E) // Retourne une arête e' ∈ E

Remarque : la première fonction « choisir » sert à sélectionner les fils de la racine d'une hypertree decomposition. L'heuristique que nous préconisons consiste à choisir l'arête qui partage le plus de variables avec la racine pour avoir des hypertrees fortement liés. La deuxième fonction « choisir » sert à choisir la racine d'une hypertree. La nouvelle racine choisie sera celle qui aura le moins de variables en commun avec les racines des hypertree précédents. Ceci pour avoir des hypertrees plus indépendants et faciliter la 2éme étape de notre démarche comme nous le verrons ultérieurement.

Exemple : l'extraction des hypertrees à partir de problème de Schure (9 variables, 16 contraintes)

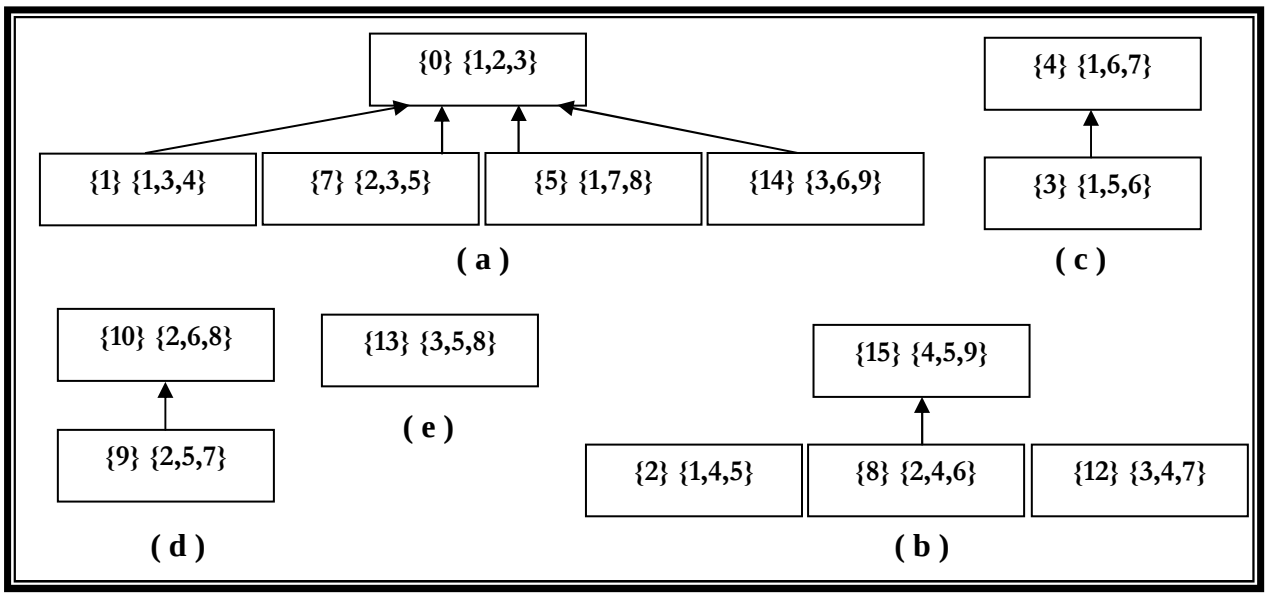

Fig.IV.1. Les hypertrees (a) HT1 (b) HT2 (c) HT3 (d) HT4 (e) HT5 extraites de problème de Schure

3. La fusion des hypertrees

Apres avoir extrait les hypertrees {HTi}, celles-ci sont contractées afin d'obtenir l'hypertree decomposition HT de CSP. La fusion est composée de deux étapes principales suivantes qui se basent sur le théorème 1.

1) Création de la racine : la racine est l'union des racines des hypertrees extraites par l'étape d'extraction.

2) Rajout des autres nœuds : les autres nœuds sont rajoutés comme nœuds fils de la racine un à un de façon à respecter les conditions de l'hypertree decomposition. Pour ce faire, à l'ajout d'un nouveau nœud N, vérifier s'il y a un ensemble de fils {Fi} précédemment rajouté. tel que $\forall i\ [\chi(N) \cap \chi(Fi)] \not\subset \chi(racine)$

a. si {Fi} est vide alors rajouter directement N comme fils de la racine

b. sinon créer un nouveau Fils F tel que $\chi(F) = [\cup_i\ \chi(Fi)] \cup \chi(N)$ et $\lambda(F) = \cup_i\ \lambda\ (Fi) \cup \lambda\ (N)$, supprimer N et tout les nœuds contenus dans {Fi}

Théorème 1 : Soit un hypergraphe H=< V, E >. Un triplet HT = < T, χ, λ > tel que T = <V(T),E(T)> est de profondeur 1 et qui vérifie les conditions suivantes :

1) $\forall\ h \in E, \exists\ p \in V(T)$ tel que $var(h) \subseteq \chi(p)$.

2) $\forall\ p \in V(T), \chi(p) = var(\lambda(p))$.

3) $\forall\ p \in V(T) : [\chi(p) - \chi(père(p))] \cap [\cup\chi(f) / f \in frère(p)\ ] = \varnothing$;

Alors HT est une hypertree decomposition

Preuve : - La condition 1 et la même que la première condition de l'HTD

- La condition 2 implique trivialement la condition 3 de l'HTD

- La condition 2 du théorème 1 implique que $\forall p \in V(T), \chi\ (p) = var(\lambda(p))$

De plus, $\forall p \in V(T) : (var(\lambda(p)) \cap \chi(Tp)) \subseteq var(\lambda(p))$ est trivialement vraie. Donc $\forall p \in V(T) : (var(\lambda(p)) \cap \chi(Tp)) \subseteq \chi(p)$. Ce qui implique que la condition 4 de l'HTD est vérifiée.

- La condition 2 de l'HTD est aussi vérifiée car le seul cas invalide quand T est de profondeur 1 consiste à avoir des noeuds feuilles partageant des sommets non couvert par la racine ce qui est contraire à la condition 3 de théorème 1.

L'étape de fusion s'appui sur le théorème 1 et est décrite formellement par l'algorithme2.

Algorithme2 : Fusion des hypertrees decomposition

Entrée : {HTi} = (Ti, χi, λi / i = 1 à k) : ensemble d'hypertrees decomposition

Sortie: HT = (T, χ, λ) hypertrees de CSP

Début

HT = **Nouvelle_Hypertree()**

racine = **Nouveau_Noeud(**HT**)**

Pour i =1 **à** k **faire**

λ(racine) = λ(racine) ∪ λi(racine(HTi))

χ (racine) = χ (racine) ∪ χi(racine(HTi))

Fait

Pour i =1 **à** k **faire**

Pour tout N ∈ Ti **faire**

N' = **Nouveau_Noeud(**HT**)**

λ(N') = λ(N)

χ (N') = χ (N)

Pour tout N'' ∈ (V(T) **–** {N'}) **alors**

Si χ (N'') ∩ (χ (N') - χ (racine)) ≠ ∅ **alors**

χ (N') = (χ (N') ∪ (χ (N'')

λ (N') = (λ (N') ∪ (λ (N'')

Supprimer(N'' **)**

Fsi

Fsi

Fait

Fait

Fait

Fin

Exemple : La fusion des hypertrees de l'exemple précédent donne le résultat illustré dans la figure suivante (Figure.2).

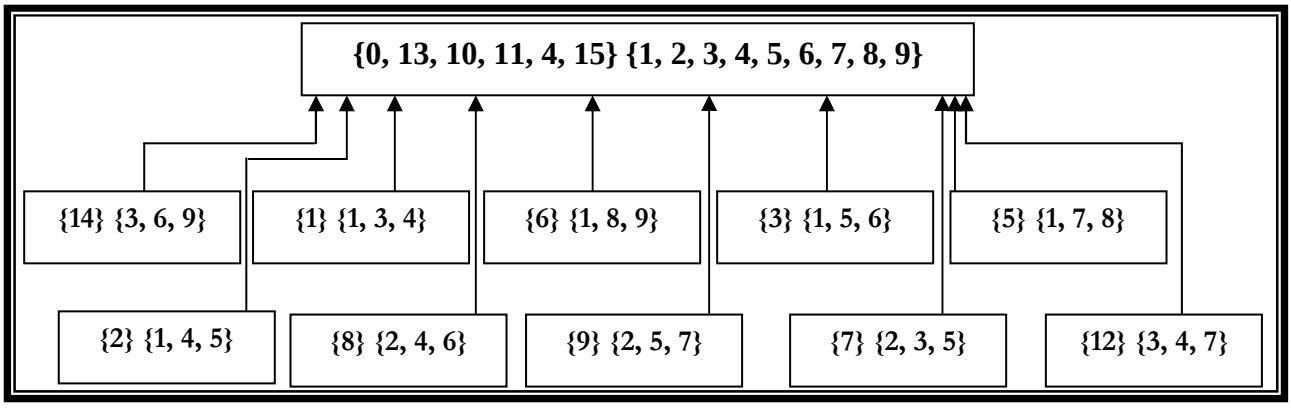

Fig.IV.2. L'Hypertree obtenue de la contraction

Remarque : Comme on peut le voir dans la figure 2, le bon choix des racines lors de la première phase a permis de générer une hypertree dont tous les nœuds feuilles sont de largeur 1. En général, un bon choix des racines permet de minimiser la taille des nœuds feuille.

4. La réduction de la largeur de l'hypertree decomposition

C'est l'étape la plus coûteuse en temps CPU, de notre démarche. Elle s'appuie sur le théorème 2 qui permet de réduire la largeur des nœuds en respectant les conditions de l'hypertree decomposition. Le principe de ce théorème est de retirer une arête de l'ensemble λ d'un nœud si celle-ci permet de créer un nœud dont la largeur est inférieure à celle du nœud réduit multiplié par un certain facteur (≥ 1) qui permet de descendre les nœuds de plus grande largeur plus en profondeur dans la décomposition et réduit ainsi d'avantage la largeur globale. Nous privilégions les arêtes qui induisent la création d'un nœud de plus petite largeur. Ce processus est répété plusieurs fois jusqu'à ce que plus aucun nœud ne soit réduit.

Théorème 2 Soit une hypertree decomposition, HT = <T=(V(T),E(T)), χ, λ > d'un hypergraphe H=(V,E) tel que :

1) $\forall p \in V(T) : \chi(p) = var(\lambda(p))$

2) Et soit $N \in V(T)$ et $h \in \lambda(N)$ tel que : $[var(h) - var(\lambda(N) - \{h\})] \cap \chi(père(N)) = \emptyset$

3) soit $F \subseteq Fils(N)$ tel que : $\forall p \in F, [\chi(p) - var(\lambda(N) - \{h\})] \cap var(h) \neq \emptyset$

Soit N' le nouveau noeud qui sera créé à partir de h et le triplet HT' = < T'=(V(T'),E(T')), χ', λ'> tel que :

4) $V(T') = V(T) \cup \{N'\} - F$

5) $E(T') = E(T) \cup \{(N,N')\} \cup \{(N',p') / (p,p') \in E(T) \text{ et } p \in F\} - \{(p,p') \in E(T) \text{ et } p \in F\} - \{(N,p) \in E(T) \text{ et } p \in F\}$

6) $\forall p \in \{V(T') - \{N'\}\} : \lambda'(p) = \lambda(p)$

7) $\lambda'(N') = \{h\} \cup \{\cup\lambda(p) / p \in F\}$

8) $\forall p \in V(T') : \chi'(p) = var(\lambda(p))$

Alors HT' est une HTD de H

Preuve :

La condition 8 implique trivialement la condition 3 de l'HTD

La condition 8 implique la condition 4 de l'HTD (voir preuve théorème1)

Les conditions 4, 6, 7 et 8 impliquent que chaque arête de H est couverte => condition 1 de l'HTD.

Pour la condition 2 de l'HTD : HT est une HTD donc par définition, aucun noeud de T ne partage un sommet avec h qui ne soit couvert par le père ou un fils de N (condition 2 de

l'HTD) donc pour que l'ajout de nouveau fils N' de N dans HT', qui est créé à partir de h, garde la condition 2 vrai il suffit que :

1. N' ne partage pas de variable avec le père de N qui ne soit couverte par p ce qui est impliqué par la condition 2 de théorème

2. N' ne partage pas de variable avec les autres fils de N ce qui est évité en fusionnant les fils de N qui ne vérifie pas cette condition (Ensemble F) ceci est induit par les conditions 3, 4, 5, et 7 de théorème.

La réduction de la largeur se base sur le théorème 2 et est décrite formellement dans l'algorithme suivant :

Algorithme Réduction de la largeur de l'hypertree

Entrée : HT = (T, χ, λ) : hypertree
Sortie: HT = (T, χ, λ) hypertree
largeur = 0 ; largeur_finale = m
Continuer = vrai

Début
Tant que (Continuer = vrai) **faire**
Continuer = faux
∀ N ∈ T **faire**
Pour i = 0 **à** |λ (N))| ***** facteur **faire**
min_nb = 0;
∀ e ∈ λ (N)) **faire Si [**var(e) ∩ [χ(Pere(N)) – var(λ (N) -e)]**]** =∅ **Alors**
E' = {F∈ Fils(N)/ χ(F) ∩ [var(e) – var(λ (N) -e)] ≠∅}
Si ∑**(**|λ (N')|) tel que N' ∈ E' ≤ i **alors**
Continuer = vrai
Si min_nb > ∑**(**|λ (N')|) / N' ∈ E' **alors**
min_nb = ∑|λ (N'))| / N' ∈ E'
Fsi
N'' = **Nouveau_Noeud(**HT**)**
χ (N'') = var(e) ∪ χ (E')
λ (N'') = {e} ∪ λ (E')
Pere(N'') = N
∀ N' ∈ E' **Supprimer(**N'**)**
Fsi
Fsi Fait
i = min_nb – 1;
Fait
Fait
Fait
Fin

Remarque : Le théorème 2 est une généralisation de lemmes existant et notamment celui de P. Harvey **[28]** qui affirme qu'une arête qui ne couvre aucune variable non couverte par les autres arêtes peut être réduite.

Exemple : La réduction de la largeur de l'hypertree obtenu dans l'exemple 2 est illustrée dans la figure suivante (Figure.IV.3), la largeur a été réduite de 6 à 4.

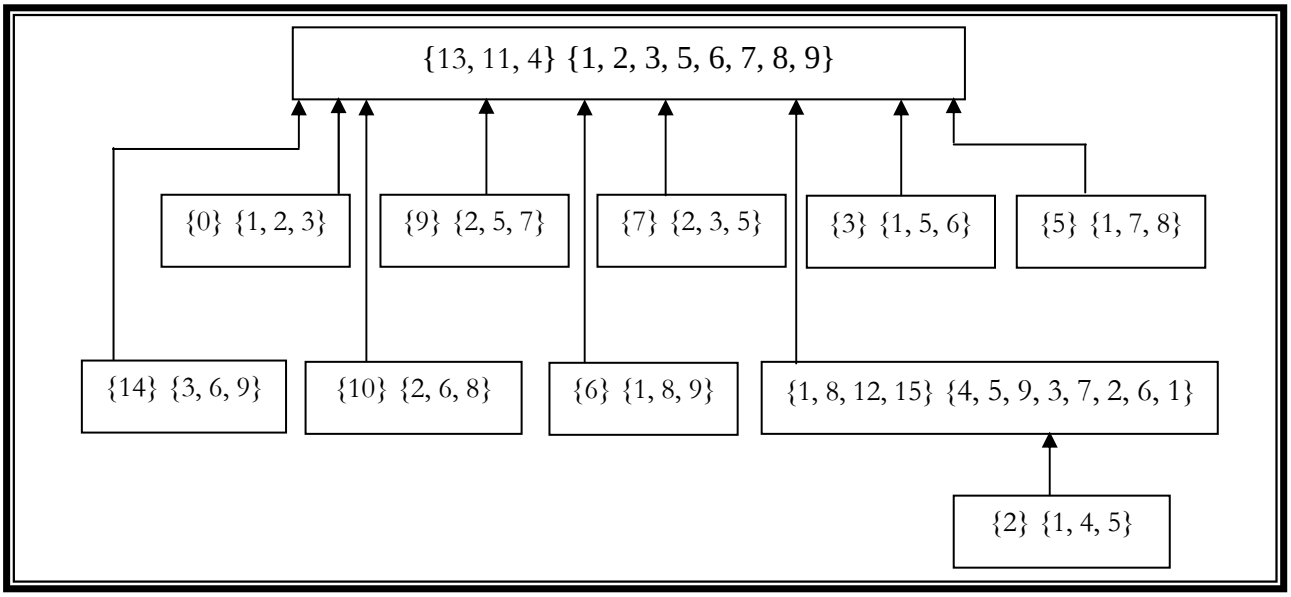

Fig.IV.3. L'Hypertree obtenue après la réduction

5. Complexité de l'approche

L'un des atouts des méthodes heuristiques dans le calcul de l'hypertree decomposition est d'être capable de trouver un résultat en un temps acceptable même pour des problèmes de grande taille. Notre proposition respecte cette condition comme le montre la proposition suivante.

Proposition 5.1

La complexité spatiale des algorithmes est en $O(m^2+mn)$ ce qui correspond à la taille maximale de l'hypertree decomposition maintenu.

Proposition 5.2

La complexité temporelle de l'extraction est en $O(m^4 + m^3n^2)$ en prenant compte les heuristiques que nous avons mentionné pour le choix des arêtes;

La complexité de la fusion est en $O(m^4 + m^2n^2)$.

Et enfin, La complexité de la réduction est en $O(m^6 + m^5n^2)$.

6. Résultats expérimentaux

Pour démontrer l'intérêt pratique de cette démarche, nous avons implémenté un outil qui complète ainsi notre plateforme de résolution multi approches. Pour cela, nous avons étendu le modèle de CSP que nous avons utilisé pour l'approche de résolution qui se base sur la recherche en ajoutant une nouvelle méthode à la classe CSP qui va s'occuper de sa décomposition et de nouvelles classes permettant la manipulation de la structure d'arbre c'est-à-dire la classe nœud, la classe étage dont chaque instance va englober des nœuds de l'arbre qui se trouvent à la même profondeur, ceci permettra de faciliter la manipulation de l'arbre comme nous le verrons notamment dans la résolution qui sera le sujet du prochain chapitre. Le diagramme de classes obtenu est illustré par la figure IV.4.

Nous avons implémenté et expérimenté notre approche sur une série de benchmarks cités dans la littérature. Nous avons effectué une étude comparative entre notre méthode et les méthodes Bucket Elimination (BE) **[30]** et Dual Bucket Elimination (DBE) **[30]** qui font partie des méthodes les plus performantes de la littérature. Ces méthodes se basent sur des algorithmes d'ordonnancement de variables. Nous avons choisi les algorithmes *minimum cardinality search* (MCS) et *minimum induced width* (MIW) pour leur large utilisation et leur équivalence avec notre méthode en terme de temps d'exécution.

Les tests ont été réalisés sur une collection de CSP qui regroupe aussi bien des problèmes académiques que des problèmes issus de monde réel. Nous avons extrait ces problèmes de ceux proposés par l'équipe DBAI **[36]** de l'université de Vienne constituant une base commune de tests des différentes méthodes de décomposition et de ceux proposés par Lecoutre qui eux, ont été proposés pour servir de challenge dans le cadre d'une compétition de solveurs de CSP[1].

[1] *Third International Workshop on Constraint Propagation and Implementation 2006* http://cpai.ucc.ie/06/

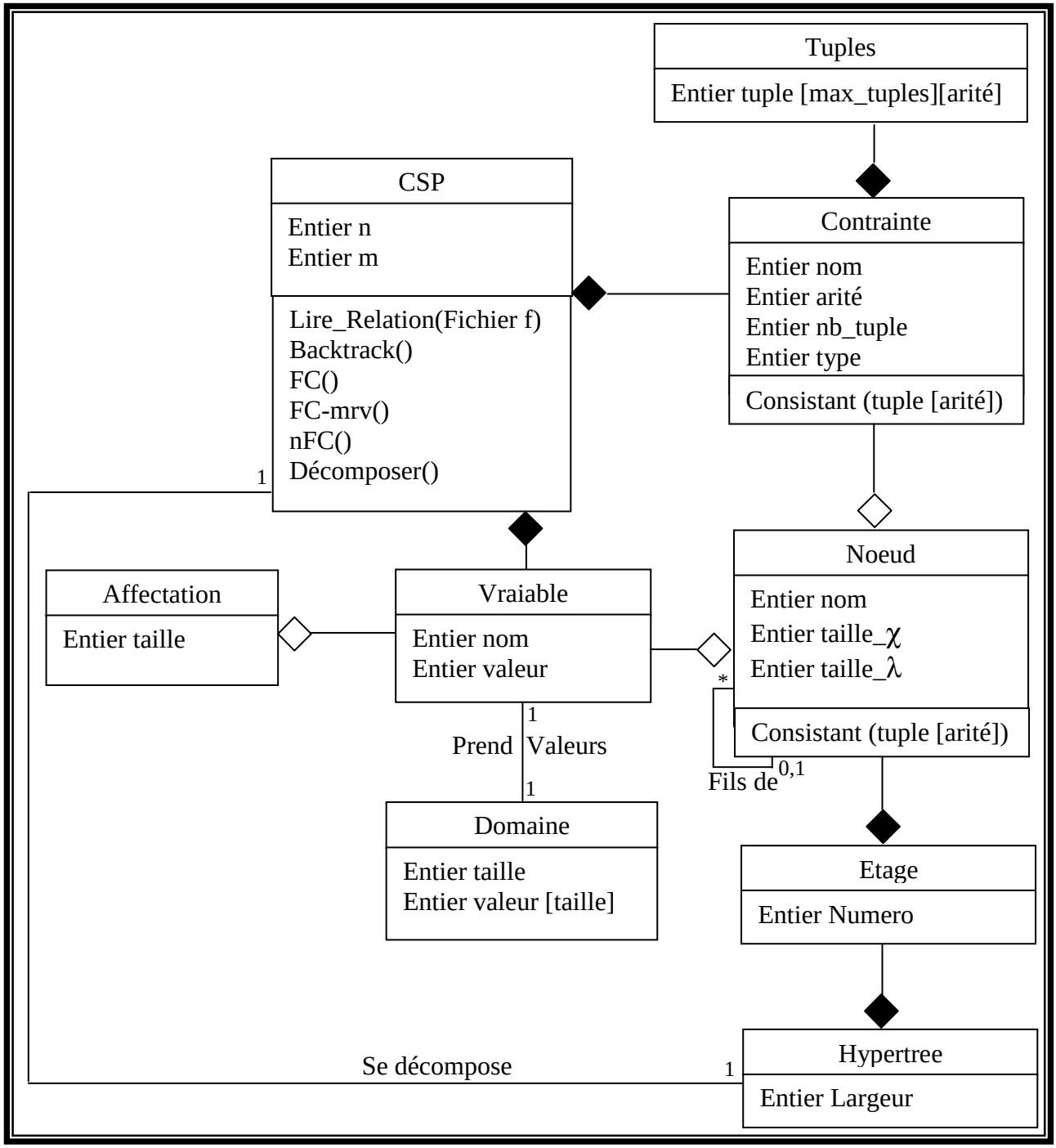

Fig.IV.4. Diagramme de classe complet

Nous avons aussi exploité l'outil de décomposition[1] de l'équipe DBAI (auteurs de la BE et de la DBE) pour les besoins d'expérimentation des méthodes BE et DBE. Comme les méthodes BE et DBE sont aléatoires, nous avons effectué cinq tests de suite pour chaque exemple et c'est la moyenne des résultats qui est prise en compte pour chaque CSP. Les résultats des tests sont résumés dans le tableau IV.1 suivant.

[1] http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.htm

CSP			BE(MCS)		DBE(MCS)		BE(MIW)		DBE(MIW)		Construct&Reduce	
Nom	**|V|**	**|E|**	**W**	**T**	**W**	**T**	**W**	**T**	**W**	**T**	**W**	**T**
Nasa	579	680	79	18	69.6	138	28.2	10	57.2	160	44	13
Reines	20	190	10	0	16.8	11	10	0	10.8	25	10	0
Renault	101	113	2.4	1	8	3	3	0	7.6	6	5	0
Schure	19	81	7	0	9.2	2	7	0	8.2	4	7	0
Fischer1	308	284	43.6	3	36.4	17	19.6	1	30.4	8	28	2
Fischer6	1523	1419	188.8	177	210.6	947	75.2	33	118	154	115	224
aim50	50	80	10.8	0	12.2	1	10.2	0	13.6	1	11	0
Aim100	100	160	20.8	1	22.4	6	21.8	1	28.8	6	24	0
grid5	25	40	4.4	0	5.4	0	5	0	6.2	0	6	0
Ii8a1	66	186	9	0	9	5	9.4	0	9	5	12	0
Par8-1-c	64	254	12.2	0	12	1	7	0	14.4	0	9	0
Par8-2-c	68	270	14.2	0	12.6	1	7	0	14.8	0	10	0
complex	414	849	22.8	2	15.4	11	10	2	12	10	17	12
uf20-50	20	91	6.8	0	9.4	2	6.4	0	8.4	4	8	0
uf75-99	75	325	22.4	1	32.6	42	20	2	31	139	24	0

Tab.IV.1. Comparaison BE, DBE et Construct & Reduce

Remarque : |V| et |E| désignent respectivement le nombre de variables et le nombre de contraintes. W désigne la largeur de la décomposition et T le temps pris par la décomposition en secondes. Pour les méthodes BE et DBE, la largeur et le temps sont la moyenne des largeurs et des temps obtenus lors de 5 exécutions successives.

A partir du tableau précédent, nous pouvons constater un certain nombre de points.

- Concernant les méthodes BE et DBE, le tableau permet de constater que la méthode BE est celle qui tire le plus de profit de l'ordre MIW et qui lui permet de réduire la largeur par rapport à l'ordre MCS dans presque tous les cas.

- On constate aussi, quand l'ordre MCS est utilisé, le fait que la BE est plus performante que la DBE quand il s'agit de décomposer des problèmes avec peu de variables et beaucoup de contraintes (Reines, Schure, uf20-050). L'inverse se produit lorsque on se trouve face à des problèmes qui ont beaucoup de variables et peu de contraintes (nasa, fischer1, complex) où c'est la DBE qui donne de meilleurs résultats. Ceci peut être expliqué par le fait que la BE est une méthode orientée variables alors que la DBE, elle, est orientée contraintes.

- On peut voir aussi que notre méthode surpasse la DBE que celle-ci se base sur l'ordre MCS ou sur l'ordre MIW que ce soit en terme de temps de calcul ou en terme de largeur de la décomposition et cela dans presque tous les problèmes de l'échantillon.

- Pour la BE, l'ordre des variables va devenir un facteur prépondérant dans la comparaison. Ainsi, quand on compare notre méthode avec la méthode BE qui se base sur l'ordre MCS, les résultats en terme de largeur sont équivalents alors qu'en terme de temps de décomposition les résultats restent légèrement en notre faveur.

- Par contre, la méthode BE avec ordre MIW reste la plus performante même si notre méthode obtient en général des résultats comparables.

Conclusion

Dans ce chapitre nous avons présenté notre principale contribution dans ce travail. Nous avons présenté de façon formelle Construct & Reduce une nouvelle heuristique pour construire une hypertree decomposition. Nous avons prouvé sa correction et montré sa faculté d'être appliquée à des problèmes de grande taille car elle se base sur des algorithmes polynomiaux. Nous avons ensuite intégré cette nouvelle approche à notre prototype de solveur multi approches et l'avons comparé à quelque unes des méthodes les plus performantes et les plus citées de la littérature. Les résultats de la comparaison ont montré l'intérêt pratique de notre méthode notamment face à *la Dual Bucket Elimination* (DBE) et à la version MCS de la B*ucket Elimination* (BE).

La décomposition n'étant qu'un prétraitement permettant de transformer un CSP quelconque en un CSP acyclique, celle-ci doit être accompagnée d'algorithmes qui peuvent tirer profit de la structure particulière qu'offrent les CSP acycliques pour pouvoir les résoudre d'une manière efficace. Ce dernier point sera l'objet de notre chapitre final où nous allons détailler l'exploitation de l'acyclicité des CSP dans leur résolution.

Chapitre V

Résolution des CSPs acycliques

Les méthodes de décomposition structurelle peuvent être décomposées en deux grandes classes, celles qui tentent de regrouper les sommets d'un graphe de manière à construire un arbre (tree decomposition) et celles qui se focalisent plutôt sur le regroupement des arêtes pour former un hyper-arbre ou hypertree (hypertree decomposition).

Toutefois ces deux classes de méthodes se rejoignent sur le but de la décomposition qui est de transformer un CSP quelconque en un CSP acyclique équivalent en terme de solutions et qui soit solvable en un temps polynomial.

La réalisation de cet objectif dépend de l'existence d'algorithmes polynomiaux qui à partir d'une décomposition arborescente d'un CSP, soit en mesure de générer un CSP acyclique équivalent et, par la suite, de le résoudre par un algorithme de type backtrack-free.

1. Résolution d'un CSP binaire acyclique

La plupart des méthodes de décomposition structurelle se basent sur le principe qu'un CSP binaire dont le graphe de contrainte est un arbre peut être résolu en un temps polynomial **[37]**. Ceci est réalisé à l'aide de l'algorithme *tree-solving* que nous rappelons dans ce qui suit :

Algorithme Tree-solving
Entrée: Un CSP acyclique = (X, D, C, R) et son arbre de contrainte T
Sortie: Une solution de CSP s'il est consistant
Début
Générer un arbre orienté et enraciné T'
Générer un ordre d = X_1, …, X_n tel que si X_i est le père de X_j dans T' **alors** i < j
Pour i = n à 2 **faire**
Révise (Père(X_i), X_i);
Si le domaine de Père(X_i) est vide **alors Exit** (Problème inconsistant).
Fait
// Le problème est consistant, rechercher une solution

Instancier X_1
Pour i = 2 à n **faire**
Affecter à X_i une valeur consistante avec les variables précédemment instancier
Fait
Fin

Etant donné un CSP et son arbre de contrainte, la première étape de *tree-solving* est de construire un arbre orienté et enraciné. Chaque nœud de cet arbre (hormis la racine) aura un seul père dirigé vers lui, et peut avoir des nœuds fils vers qui il pointe. Les nœuds qui n'ont pas de fils sont appelés nœuds feuilles. Un ordre sur les variables du CSP d = X_1, …, X_n est calculé de telle façon qu'un nœud précède toujours ses fils dans cet ordre. La deuxième étape de l'algorithme est appelée la consistance d'arc dirigée (DAC pour Directed Arc Consistency). Elle consiste à traiter chaque arc de l'arbre orienté en partant des nœuds feuilles et en remontant niveau par niveau jusqu'à la racine. Ainsi la méthode Révise() est appelée pour chaque variable afin de supprimer les valeur du domaine de son père qui n'ont pas de support dans son propre domaine. Enfin, si aucun domaine de variable n'est vide l'algorithme Backtrack-free est utilisé pour trouver une solution au CSP.

La complexité de *tree-solving* est en $O(nd^2)$ où n est le nombre de variables du CSP et d la taille maximale des domaines des variable.

Exemple : soit le CSP P = <X,C,D,R> tel que X= {a, b, c, d, e, f, g} et C = {{a,b},{a,c},{b,d},{b,e},{c,f},{c,g}}. La figure V.1 montre l'arbre orienté enraciné de P.

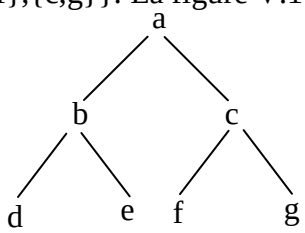

Fig.V.1. Un arbre orienté enraciné

L'algorithme *tree-solving* permettant de résoudre les CSPs binaires acycliques en un temps polynomial a été à l'origine du développement des méthodes de décomposition structurelle.

2. Résolution d'un CSP n-aire acyclique

La résolution des CSP binaires acycliques est assez simple comme nous avons pu le voir dans le paragraphe précédent car le graphe de contraintes et son graphe primal sont identiques, et chaque arête du graphe primal représente une contrainte de CSP. Donc si le graphe primal est acyclique alors le CSP l'est aussi et vice versa. Par contre, dans le cas des CSP n-aires cette équivalence n'est plus vraie car l'hypergraphe de contraintes est différent de son graphe primal. Pour cela, une nouvelle définition plus générale de la cyclicité d'un CSP a été introduite. Un CSP est acyclique s'il possède un join tree.

Pour prendre en considération la nouvelle définition de la cyclicité, l'algorithme tree-solving a été adapté sous le nom de *Acyclic-Solving* en manipulant cette fois ci n'ont pas les variables mais les contraintes de CSP.

Algorithme ACYCLIC-SOLVING
Entrée: Un CSP acyclique P = (X, D, C, R), R = {R1, …, Rt}. Un join-tree *T* de P.
Sortie: Vérifier la consistance et chercher une solution dans le cas d'un CSP consistant
Début
d = (R_1,…, R_t) un ordre tel que chaque relation apparaît avant ses fils dans l'arbre de T enraciné en R_1.
Pour *j* = *t* à 1, pour chaque arête (j, k) , *k* < *j*, dans l'arbre **Faire**
$R_k \leftarrow$ ***Join***(R_k, R_j)[C_k]
Si une relation vide est créée **alors Exit**(le problème est inconsistant).
Fait
// Le CSP est consistant, chercher une solution
Sélectionner un tuple dans R_1.
Pour i = 2 à t **faire**

Sélectionner un tuple dans R_i qui soit consistant avec les tuples choisis précédemment
Fait
Fin
Join(ensemble de contraintes c**)** est la jointure relationnelle entre les relations de l'ensemble des contraintes c.
R[ensemble de variables V**]** dénote la projection de la relation R sur les variables V.

Exemple : Soit le CSP n-aire P, son hypergraphe ainsi que son graphe primal, son graphe dual et son join tree sont illustrés dans la figure V.2 suivante :

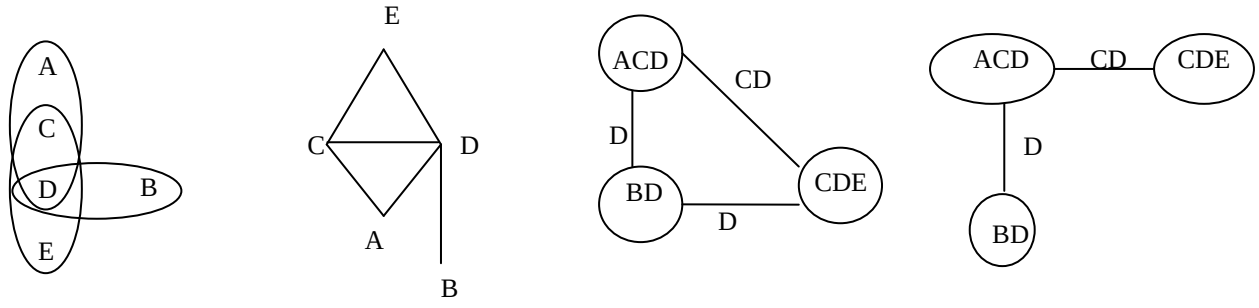

Fig.V.2. hypergraphe, graphe primal, graphe dual et join graphe

L'exemple précédent montre un CSP acyclique (qui possède un join tree) dont le graphe primal n'est pas acyclique.

3. Résolution d'une Tree décomposition

Les premières méthodes de décomposition été généralement basées sur le concept de la tree decomposition qui consiste à regrouper les variables de façon à obtenir un arbre. Nous commençons par rappeler la définition d'une tree decomposition.

Définition 2.1 : Une tree décomposition ou décomposition arborescente d'un hypergraphe H = (V(H),E(H)) est un couple <T, χ> tel que T = (V(T), E(T)) est un arbre et χ est une fonction d'étiquetage qui associe a chaque nœud t de T un ensemble de variables $\chi(t) \subseteq V(H)$ et qui respecte les conditions suivante :

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

A partir de cette définition, l'algorithme Acyclic-solving a été adapté pour pouvoir résoudre les problèmes ayant une décomposition arborescente

3.1. Construction de CSP acyclique équivalent

Une fois la décomposition arborescente <T, χ> d'un problème P = <X, D, C, R> obtenue, le CSP acyclique équivalent P' = <X', D', C', R'> doit être créé tel que

X' = X

D' = D

C' = {χ(n) / n $\in$ T}

R' = **{** ***Join*** **(**{c $\in$ C/ c $\subseteq$ χ(n)})[χ(n)**)]** / n $\in$ T**}**

3.2. Résolution de CSP acyclique équivalent

Une fois le CSP acyclique équivalent P' construit, comme celui-ci est généralement un CSP n-aire par construction, l'algorithme Acyclic-Solving est appelé pour le résoudre. Comme P' est équivalent au CSP initial, la solution trouvée pour P' est aussi une solution du CSP initial.

4. Résolution d'une Hypertree decomposition

Comme l'hypertree decomposition généralisée et l'hypertree decomposition sont traitées de la même manière lors de la résolution, nous allons utiliser le terme hypertree decomposition pour désigner ces deux méthodes.

4.1. Compléter l'hypertree decomposition

Une fois une hypertree decomposition HD = <T, λ, χ> d'un hypergraphe H est calculée celle-ci doit être d'abord complétée pour créer une hypertree decomposition complète. Ceci est fait simplement en rajoutant, pour chaque hyperarête h de H qui n'est pas complètement couverte par HD, un nouveau nœud N tel que λ(N) = {h} et χ(N) = ($\cup$h) comme fils d'un nœud de T qui couvre h. Cette tâche est réalisée en un temps linéaire.

4.2. Construction de CSP acyclique équivalent

Une fois l'hypertree decomposition complétée, la deuxième étape consiste à créer le CSP acyclique équivalent à partir de cette décomposition. Ainsi, pour un CSP quelconque P = <X, D, C, R> dont l'hypertree decomposition est HD = <T, λ, χ> le CSP acyclique équivalent P' = <X', D', C', R'> est défini comme suit :

X' = X

D' = D

C' = {χ(n) / n ∈ T}

R' = **{** ***Join*****(**λ(n)**)[** χ(n)**]** / n ∈ T**}**

4.3. Résolution de CSP acyclique

Pour finir, la dernière étape est la résolution proprement dite c'est-à-dire le filtrage et la recherche d'une solution. Ceci est fait comme dans le cas de la tree decomposition par l'algorithme *acyclic-solving*.

5. Etudes de cas

Nous allons illustrer cette approche de résolution par une étude de cas pour résumer la manière dont l'approche de résolution par décomposition peut être utilisée pour résoudre des problèmes concrets en général et, en particulier, comment la méthode que nous avons proposé pour le calcul de l'hypertree decomposition peut être intégrée dans un projet réel. L'étude de cas choisis concerne le problème Renault et la famille de problèmes Dubois.

5.1. Le problème Renault

Le problème Renault est un CSP issu du monde de l'industrie qui est souvent utilisé dans les compétitions de solveurs de CSP.

C'est un problème d'usinage de véhicule Renault Mégane, où chaque variable représente une option ou une caractéristique d'un composant de véhicule et les contraintes sont les configurations possibles de la Mégane.

Ainsi, parmi les variables, on trouve par exemple :

Une variable pour le type de consommation (essence ou diesel)

Une variable pour le type de kilométrage (KM ou Milles)

Une variable pour la présence ou non de l'ABS

…

Le problème Renault a été modélisé sous la forme d'un CSP avec 101 variables et 113 contraintes. Ce problème est un problème facile à résoudre avec les méthodes de type Backtrack car le nombre de configuration possibles de la Mégane et donc les solutions de CSP qui le formalise sont très nombreuses.

5.1.1. Décomposition

La première étape de notre travail a consisté à décomposer le problème pour obtenir son hypertree decomposition Pour cela nous avons utilisé notre propre méthode Construct & Reduce. La décomposition du problème avec notre méthode s'est faite en 0.3 secondes pour obtenir une décomposition de largeur 5, alors que la BE et la DBE ont obtenu respectivement des décompositions de largeur 2 et 7.

A partir de cette décomposition, comme la largeur deviendra un paramètre crucial lorsqu'il s'agira de faire les jointures entre les relations, nous avons utilisé la méthode exacte Det-k-decomp[1] d'une manière itérative jusqu'à obtenir la largeur optimale c'est à dire une décomposition de largeur 2.

5.1.2. Résolution

Pour les besoins de cette phase, nous avons d'abord enrichi notre plateforme de résolution des CSP par un nouvel outil de résolution des CSP ayant une hypertree decomposition. Cet outil se compose de deux fonctionnalités principales, la première s'occupe de la génération d'un CSP acyclique équivalent alors que la deuxième implémente l'algorithme Acyclic-Solving pour la résolution de CSP acyclique.

[1] www.dbai.tuwien.ac.ac/proj/hypertree/downloads

Nous avons donc utilisé cet outil pour résoudre l'hypertree decomposition de largeur optimale que nous avons calculé précédemment. Les coûts en temps obtenus sont les suivants :

- Création de CSP équivalent : 234 secondes
- Filtrage des relations : 3791 secondes
- Recherche d'une solution : 0.03 secondes

De ces résultats on peut voir que même si le gain en temps lors de la recherche de la solution est évident, le coût nécessaire en prétraitement (jointure et filtrage), dans le cas de problème Renault, reste encore très important même si l'hypertree decomposition utilisée est bien de largeur optimale. Ces résultats sont d'autant plus décevants que l'algorithme de résolution énumératif nFC2-mrv résout le problème en 0.3 secondes.

Néanmoins ces résultats restent très logiques. Le problème Renault est caractérisé par des relations de taille très importantes (> 30000 tuples pour certaines d'entre elles) ce qui alourdit fortement les opérations de jointures et de semi-joinures de l'approche de résolution par décomposition. De l'autre coté, le nombre important de solutions de Renault facilite la recherche d'une solution ce qui explique la rapidité de résolution de la méthode énumérative nFC2-mrv.

5.1. La famille de problèmes Dubois

Nous avons aussi considéré des problèmes de différentes tailles de la famille Dubois qui, contrairement à Renault, sont caractérisés par l'absence de solutions (insatisfiable) et une taille de relation très faible (4 tuples par relation).

5.1.1. Décomposition

La décomposition s'est faite de la même manière que pour le problème Renault, notre méthode donne une largeur de 3 puis le Det-k-decomp calcule la décomposition optimale (largeur 2)

5.1.2. Résolution

Les résultats de la résolution sont illustrés par les figures V.3 et V.4

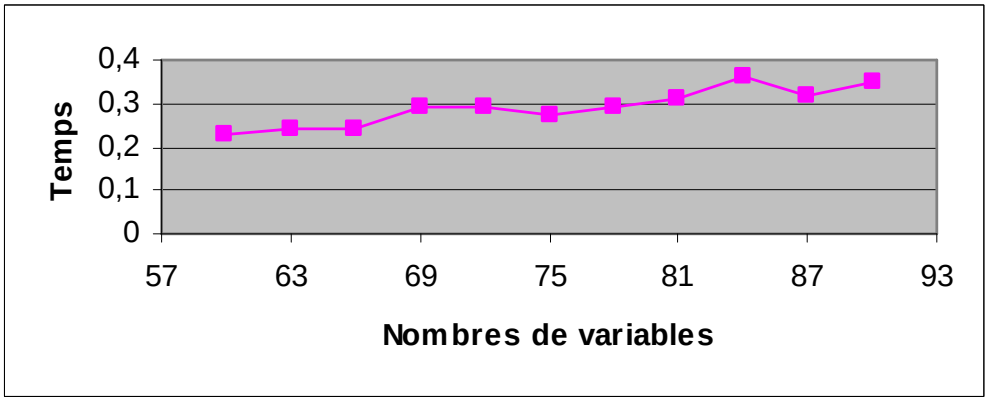

Fig.V.3. Temps de résolution avec l'approche par décomposition

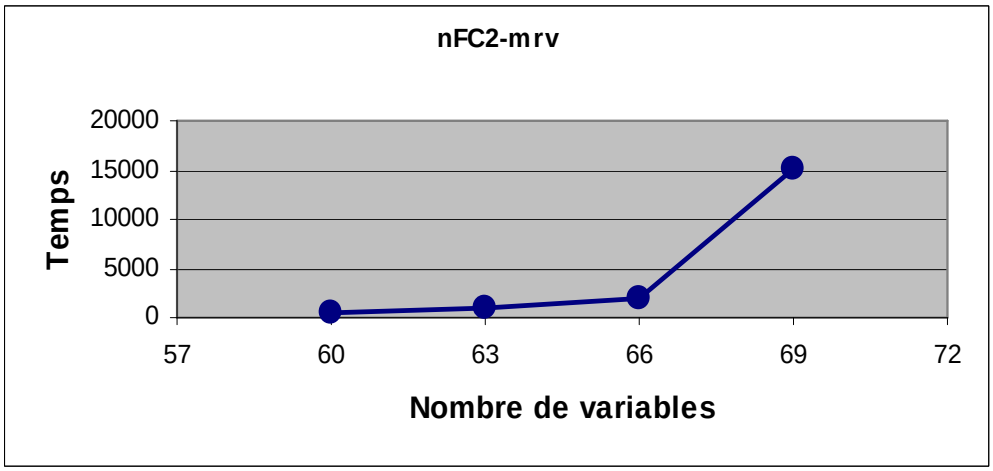

Fig.V.4. Temps de résolution avec l'approche énumérative

Ici, on constate que l'approche de résolution par décomposition est bien plus rapide que l'approche énumérative. En effet, alors que la première a besoin de moins de 0.5 secondes pour affirmer l'insatisfiabilité des problèmes, la deuxième a besoin parfois de plus de 4 heures pour arriver au même résultat. De plus, pour l'approche de résolution par décomposition, une évolution de la taille de problème n'induit que augmentation minime de temps de résolution alors que pour l'approche énumérative, cette augmentation est exponentielle.

Ces résultats s'expliquent par la taille limitée des relations de ces problèmes et la largeur très faible de leur décomposition qui a accéléré sensiblement les opérations de jointures et de semi-jointures. De plus, l'insatisfiabilité de problème a induit le ralentissement de nFC2-mrv qui est obligé de parcourir l'arbre de recherche en entier.

Conclusion

Apres avoir détaillé dans les chapitres précédents quelques unes des méthodes de décomposition, nous avons expliqué dans ce dernier chapitre comment les décompositions obtenues peuvent être exploitées pour résoudre un CSP en un temps polynomial. Pour cela nous avons d'abord parlé des algorithmes de résolution spécifiques aux CSP acycliques qui soient binaires ou n-aires. Ensuite, nous avons montré comment on peut passer d'un CSP quelconque dont on connaît une décomposition à un CSP acyclique équivalent en terme de solution.

Finalement, nous avons choisi d'illustrer notre travail par des étude de cas, d'une part pour montrer l'utilité de l'approche CSP en général et l'approche de résolution par décomposition en particulier pour résoudre des problèmes quels que soit leur nature ou leur origine. D'autre part cette étude nous permet de résumer tout ce que nous avons réalisé jusque là et de montrer comment la méthode de calcul de l'hypertree decomposition que nous avons proposé peut être intégrée dans une démarche globale de résolution par décomposition.

Pour réaliser cette étude, nous avons d'abord développé un outil de résolution qui vient ainsi s'ajouter à la plateforme que nous sommes entrain de réaliser. Les résultats obtenus nous laisse supposer que l'approche de résolution par décomposition peut être très intéressante quand il s'agit de résoudre des problèmes difficiles (peu de solutions) caractérisés par des tailles de relation et des largeur de décomposition très faible alors que le contraire handicape fortement cette approche.

Ces résultats mitigés nous poussent à plus de recherches dans le domaine de la définition même des méthodes de décomposition structurelle notamment dans la façon dont les décomposition obtenues sont utilisées pour la résolution et le faite que celle-ci tente juste de minimiser la largeur alors que le coût en calcul dépend de beaucoup d'autres facteurs tel que la taille des relations, la sélectivité d'une variable dans les relations …

Conclusion et perspectives

L'utilisation de la recherche arborescente pour la résolution des CSP, combinée à des techniques rétrospectives tel que le retour arrière intelligent, des techniques prospectives tel que le filtrage par consistance d'arc ainsi que l'exploitation d'heuristiques sur le choix des variables, obtient en général des résultats pratiques satisfaisants sur une large palette de problèmes. Néanmoins, la complexité théorique d'une telle approche est exponentielle en taille de problème et ceci qu'il s'agisse de la complexité temporelle ou de la complexité spatiale. Ce constat a motivé le développement des méthodes de décomposition structurelle, qui permettent de borner la complexité théorique en la reliant à la largeur de la décomposition au lieu de la taille de problème.

L'hypertree decomposition est l'une de ces méthodes de décomposition les plus intéressantes, notamment parce qu'elle généralise une bonne partie des méthodes existantes en plus de l'existence de méthodes exactes qui peuvent calculer une hypertree decomposition de largeur optimale. Malheureusement, ces méthodes exactes induisent un coût exorbitant en terme de calcul et en occupation mémoire, ce qui a orienté les recherches ces dernières années vers le développement de méthodes heuristiques offrant un bon compromis entre la largeur et le coût de la décomposition.

Pour contribuer à la résolution de problème de calcul de l'hypertree decomposition, nous avons proposé une méthode heuristique originale dite Construct & Reduce. Notre méthode est composée de 3 phases :

- L'extraction des hypertrees contenues dans le CSP.
- La contraction des hypertrees pour former l'hypertree decomposition de CSP.
- Et en fin la réduction de la largeur.

De plus, pour garantir la correction de nos algorithmes, nous avons introduit deux théorèmes sur lesquels se basent tous nous algorithmes. Ce qui nous a permis de disposer d'un moyen pour prouver que notre approche calcule bien une hypertree decomposition.

Les CSP étant un cadre ou les résultats pratiques sont primordiaux, nous avons entamé la réalisation d'une plateforme de résolution multi approches de CSP afin d'être en mesure de tester et de vérifier l'intérêt de la méthode que nous avons proposé. Cette plateforme est constituée d'un outil de résolution par la recherche qui implémente les algorithmes Backtrack, FC, FC-mrv et nFC(2 à 5), un outil pour le calcul de l'hypertree decomposition qui implémente notre méthode Construct & Reduce et enfin un outil de résolution des CSP qui dispose d'une hypertree decomposition. De plus, comme notre plateforme est le fruit d'un développement orienté objet, les modules existants sont fortement réutilisables et permettent ainsi de faciliter l'implantation de nouveaux algorithmes.

L'expérimentation de Construct & Reduce et la comparaison des résultats avec ceux de deux des méthodes les plus intéressantes de la littérature, à savoir la Bucket Elimination (BE) et la Dual Bucket Elimination (DBE), ont montré l'intérêt pratique de notre méthode qui donne des résultats meilleurs que ceux de la DBE et généralement équivalents à ceux de la BE.

En fin, nous avons clos notre mémoire par une étude de cas pour montrer comment notre méthode peut être utilisée dans le cadre de la résolution de CSP. Nous avons ainsi préconisé la combinaison de notre méthode avec une méthode exacte, la Det-k-decomp, pour tirer avantage de la vitesse de la première et de l'optimalité de la largeur obtenue par la deuxième.

Les résultats de l'utilisation de l'approche de résolution par décomposition sont à la fois encourageants et non satisfaisants. Encourageants car ils montrent bien qu'après la phase de prétraitement, constituée des jointures et de filtrage, la recherche d'une solution se fait extrêmement rapidement. Mais non satisfaisants car le coût de prétraitement et encore très élevé quand ils s'agit de problèmes dont la taille des relations est importante même avec l'utilisation d'une décomposition de largeur optimale, ce qui nous amène aux perspectives à court terme suivantes :

- L’adaptation de notre algorithme de réduction de la largeur aux autres méthodes notamment la BE
- La parallèlisation de la résolution.
- La prise en compte d’autres paramètres que la largeur dans la décomposition tel que la taille des tuples et la sélectivité des variables dans les relations.
- La recherche de nouvelles méthodes de décomposition plus générale et plus efficace que l’hypertree decomposition.

Bibliographie:

[1] U. Montanari, « Networks of constraints: Fundamental properties and applications to picture », processing. Information Sciences, 1974.

[2]A. Mackworth. « Consistency in Networks of Relations ». Artificial Intelligence, 1977.

[3] C. Bessier et J. Regin, « Refining the basic constraint propagation algorithm », Proceeding of IJCAI'01, 2001.

[4] Y.Zhang et R.H.C. Yap, « Making AC3 an optimal algorithm », Proceeding of IJCAI'01, 2001.

[5] V. Dongen, « AC3d an efficient arc consistency algorithm with a low space complexity », Proceeding of CP'02, 2002.

[6] C. Lecoutre, F. Boussemart, F. Hemery, «Au coeur de la consistance d'arc », JNPC, 2003.

[7] R. Mohr et T.C. Henderson, « Arc and Path consistency revisited », Artificial Intelligence, 1986.

[8] C. Bessiere, « Arc consistency and Arc consistency again », Artificial Inteligence, 1994.

[9] C. Bessier, E.C. Freuder et J.C. Regin, « Using constraint metaknowledge to reduce arc consistency computation », Artificial Intelligence, 1999.

[10] D. Frost et R. Dechter. « Look-ahead value ordering for constraint satisfaction problems ». Proceedings of IJCAI 95, 1995.

[11] C. Bessiere, R.H.C. Yap, JC. Régin, Y. Zhang. « An Optimal Coarse-grained Arc consistency Algorithm ». Artificial Intelligence, 2005.

[12] C. Bessiere, P. Meseguer, E.C. Freuder, J. Larrosa « On forward checking for non binary constraints satisfaction problems », Artificial Intelligence, 2002.

[13] C. Bessiere, « Constraint Propagation», Technical Report LIRMM 06020, CNRS/University of Montpellier, 2006.

[14] V. Castro, « Solving binary CSP using computational systems », Elsevier science, 1996.

[15] V. Kumar, « Algorithms for Constraint Satisfaction Problems: A Survey », AI Magazine, 1992.

[16] C. Lecoute, F. Boussemart, F. Hemery, « De AC à AC7 », Actes de JFPLC'03, 2003

[17] E. Tsang, « FOUNDATIONS OF CONSTRAINT SATISFACTION», Academic Press Limited,1995.

[18] R. Dechter and J. Pearl, "The Cycle Cutset Method for improving search Performance in AI applications". Proccedings 3 rd IEEE Conference on AI applications Orlando 1987.

[19] E. Freuder, "A Sufficient Condition for Backtrack-Bounded Search", JACM 32(4), 1985.

[20] R. Dechter and J. Pearl, "Tree clustering for constraint networks", Artificial Intelligence 38, 1989.

[21] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. "On the desirability of acyclic database schemes". Journal of the ACM,30(3), 1983.

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

[23] D. Cohen, M. Gyssens, P. Jeavson, « Decomposing Constraint Satisfaction Problems Using Database Techniques », Artificial Intelligence volume 66, 1994.

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

[25] D. Cohen, P. Jeavons, M. Gyssens, "A Unified Theory of Structural Tractability for Constraint Satisfaction Problems", Proceedings of 19th IJCAI, 2005.

[26] G. Gottlob, N. Leone, F. Scarcello. "A comparison of structural CSP decomposition methods". Proceedings of the 16th IJCAI, 1999.

[27] G. Gottlob, N. Leone, F. Scarcello, "On tractable queries and Constraints", In *Proc. DEXA'99*, 1999.

[28] P. Harvey and A. Ghose, "Reducing Redundancy in the Hypertree Decomposition Scheme", ICTAI03, 2003.

[29] G. Gottlob, M. Samer, "A Backtracking-Based Algorithm for Computing Hypertree decompositions", Technical Report arXiv:cs.DS/0701083, 2007.

[30] A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, M. Samer, "Heuristic Methods for Hypertree Decompositions", DBAI-TR-2005.

[31] N. Musliu, W. Schafhauser. "Genetic Algorithms for Generalized Hypertree Decompositions". *European Journal of Industrial Engineering Volume1 No.3* 2007.

[32] M. Samer, " Hypertree-decomposition via Branch-decomposition", IJCAI, 2005.

[33] N. Musliu, "Tabu Search for Generalized Hypertree Decompositions", MIC 2007.

[34] R. Dechter, "Tractable Structures for Constraint satisfaction problems", Chapitre dans "Handbook of constraint programming", Elsevier, 2006.

[35] S. Subbarayan and H. Andersen, **«** Backtracking Procedures for Hypertree, HyperSpread and Connected Hypertree Decomposition of CSPs », IJCAI, 2007.

[36] T. Ganzow, G. Gottlob, N. Musliu, M. Samer, "A CSP Hypergraph Library », *DBAI-TR*-2005.

[37] E. Freuder, « A sufficient condition for backtrack free search », JACM volume 29, 1982.

Résumé :

Les problèmes de satisfaction de contraintes (CSP) sont un cadre générique permettant la représentation et la résolution d'une large palette de problèmes. Plusieurs approches existent pour la résolution des CSP, l'une d'elles est l'approche de résolution par décomposition qui regroupe un ensemble de méthodes dites de décomposition structurelle.

Dans ce mémoire, nous nous intéressons particulièrement à une méthode récente de décomposition structurelle appelée hypertree decomposition et plus particulièrement au calcul de cette décomposition.

Nous proposons alors une nouvelle méthode heuristique appelée Construct&Reduce, pour le calcul de l'hypertree decomposition puis nous décrivons la manière dont cette nouvelle méthode peut être utilisée dans le cadre de la résolution des CSP.

Mots clés : problème de satisfaction de contrainte (CSP), méthode de décomposition structurelle, hypertree decomposition.

Abstract

Constraints Satisfaction Problems (CSP) are a generic framework allowing the representation and the resolution of a large set of problems. Many approaches exist to solve CSPs, one of them is the resolution by decomposition which includes many structural decomposition methods.

In this report, we have studied a recent structural decomposition method called hypertree decomposition and more particularly the problem of computing this decomposition.

Then, we defined a new heuristic method we called Construct & Reduce, for computing hypertree decomposition and illustrated how our method can be used in the resolution.

Keywords: Constraints satisfaction problems (CSP), structural decomposition methods, hypertree decomposition.