Introduction aux problèmes de satisfaction de contraintes

Par Dr AMROUN Kamal

Sommaire

1 Les problèmes de satisfaction de contraintes **1**

1.1 Introduction 1

1.1.1 Représentation graphique 2

1.1.2 Exemples d'instances de CSP 2

1.2 Les méthodes de résolution 5

1.2.1 Filtrage 6

1.2.1.1 Consistance de nœud NC 6

1.2.1.2 Consistance d'arc AC 6

1.2.1.3 Consistance d'arc directionnelle (DAC) 8

1.2.1.4 Consistance de chemin PC 8

1.2.1.5 Consistance de chemin directionnelle DPC 9

1.2.1.6 Consistance de chemin restreinte RPC 10

1.2.1.7 k-consistance 11

1.2.1.8 (i, j)-Consistance 11

1.2.1.9 Consistance inverse 11

1.2.2 Filtrage des CSPs n-aires 12

1.2.2.1 Arc Consistance généralisée GAC 12

1.2.2.2 Algorithme GAC 2001 12

1.2.2.3 Interconsistance 12

1.2.2.4 Consistance relationnelle 14

1.2.3 Algorithmes énumératifs 16

1.2.3.1 Le retour-arrière chronologique (Backtrack) 17

1.2.3.2 Algorithmes avec retour arrière non chronologique 17

1.2.3.3 Heuristiques d'ordre 18

1.2.3.4 Algorithmes avec filtrage avant 19
1.2.3.5 Algorithmes Backtrack avec mémorisation 22
1.3 CSPs n-aires 24
1.3.1 Transformation de CSPs n-aires en CSPs binaires 24
1.3.2 Résolution directe des CSPs n-aires 25
1.4 Conclusion 25

2 Les techniques de décomposition structurelles 26
2.1 Graphes et hypergraphes 26
2.2 Traitabilité 27
2.2.1 Acyclicité d'un hypergraphe 28
2.3 Les principales méthodes structurelles 29
2.3.1 Principe de ces méthodes 29
2.3.2 La méthode CCM (Cycle cutset) 29
2.3.3 La méthode Cycle Hypercutset 30
2.3.4 La méthode BICOMP (Biconnected Components) 31
2.3.5 La méthode *TCLUSTER* (Tree clustering) 33
2.3.6 La méthode TD (Tree decomposition) 34
2.3.7 La méthode BTD (Backtracking on Tree-Decomposition) 35
2.3.8 La méthode Hinge (HINGE decomposition) 38
2.3.8.1 Définitions 38
2.3.8.2 Algorithme de calcul de la décomposition Hinge 39
2.3.9 La méthode GHD (Generalized Hypertree Decomposition) 40
2.3.9.1 Définitions 40
2.3.9.2 Calcul d'une (G)HD 42
2.3.9.3 Résolution des instances CSP via une (G)HD 44
2.3.10 Les décompositions gardées et Spread Cuts 46
2.3.10.1 Les décompositions gardées 46
2.3.10.2 La méthode SCD (Spread cuts) 47
2.3.10.3 La méthode SCD_{NEW} (Spread Cut_{NEW}) 48
2.3.11 Subedge-based decompositions 49
2.3.11.1 La méthode CHD (Component Hypertree) 51
2.3.11.2 La méthode ECHD (Extend Component Hypertree) 53

2.3.11.3 La méthode ECHD[d] (Parameterized Extend Component Hypertree) 54
2.3.12 La méthode FHD (Fractional Hypertree Decomposition) 55
2.4 Classification des méthodes de décomposition 56
2.5 Conclusion 57

Bibliographie **58**

Chapitre 1

Les problèmes de satisfaction de contraintes

Dans ce chapitre, nous présentons le formalisme des problèmes de satisfaction de contraintes (CSP) ainsi que les principales méthodes de résolution.

1.1 Introduction

La notion de problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) a été introduite par Montanari [67]. Un CSP est modélisé sous la forme d'un ensemble de contraintes portant sur des variables, chacune prend ses valeurs dans un domaine particulier.

Définition 1.1.1. (**Instance CSP** [67])

Une instance CSP est un triplet $P = (X, D, C)$ où

- $X = \{X_1, \cdots, X_n\}$ est un ensemble de n variables,
- $D = \{D_1, \cdots, D_n\}$ est un ensemble de n domaines finis. Chaque domaine D_i est associé à une variable X_i (noté Dom(X_i)=D_i).
- C est un ensemble de m contraintes. Chaque contrainte C_i est définie par une paire $\langle Scope(C_i), Rel(C_i)\rangle$. Scope($C_i$) = $(X_{i_1}, \cdots, X_{i_r})$ [1] est une liste de variables sur lesquelles est définie C_i. $Rel(C_i)$ (notée aussi R_i) définit l'ensemble des tuples sur $D_{i_1} \times \cdots \times D_{i_r}$ autorisés par la contrainte C_i. L'*arité* d'une contrainte est le nombre de variables qu'elle

[1] Pour simplifier, C_i désigne aussi les variables $\{X_{i_1}, \cdots, X_{i_r}\}$ du Scope(C_i). De plus, nous considérons le scope d'une contrainte comme une liste mais nous le manipulons comme un sous-ensemble de variables.

met en relation. Si l'arité d'une contrainte C_i est égale à 2 alors C_i est dite binaire sinon elle est dite n-aire. Si l'arité de chaque contrainte est égale à 2 alors l'instance de CSP est dite *binaire*, sinon elle est dite *n-aire*. En général, il y a deux possibilités pour définir une contrainte. La première est dite *explicite* où les tuples de la relation sont donnés en extension. La seconde est une représentation *implicite* (équations, inequations, etc.).

Définition 1.1.2. (**Instanciation**)
Soient l'instance CSP $P = (X, D, C)$ et $Y \subseteq X$, on appelle *instanciation* de Y l'affectation à chaque variable $X_i \in Y$ d'une valeur de $\mathrm{Dom}(X_i)$.

Définition 1.1.3. (**Instanciation consistante**)
Soient l'instance CSP $P = (X, D, C)$ et $Y \subseteq X$, une instanciation $\mathcal{A}$ sur Y est dite *consistante* si elle satisfait toute contrainte $C_i \in C$ telle que $Scope(C_i) \subset Y$.

Définition 1.1.4. (**Solution**)
Une *solution* d'une instance CSP $P = (X, D, C)$ est une instanciation consistante de toutes les variables de X.

Le problème de l'existence d'une solution pour un CSP est un problème NP-complet [34].

1.1.1 Représentation graphique

Une instance CSP dont toutes les contraintes sont binaires peut être représentée par un graphe dont les sommets sont les variables et il existe une arête entre deux sommets X_i et X_j s'il existe une certaine contrainte dont la portée (scope) contient les variables X_i et X_j. Ce graphe est appelé *graphe de contraintes*. Plus généralement, une instance CSP quelconque (*n-aire*) est représentée par un *hypergraphe de contraintes* dont les sommets sont les variables et il existe une hyperarête $\{X_1, \cdots, X_k\}$ si ce dernier ensemble est la portée d'une certaine contrainte.

1.1.2 Exemples d'instances de CSP

Exemple 1.1.1. *(Problème des 4 reines)*

Le problème consiste à placer 4 reines sur un échiquier 4X4 de telle sorte qu'aucune reine

ne puisse atteindre une autre (deux reines ne sont jamais sur la même ligne ou sur la même colonne ou sur la même diagonale).

Modélisation du problème

- *Associer à chaque ligne i de l'échiquier une variable X_i. Les domaines des variables X_1, X_2, X_3 et X_4 sont respectivement D_1, D_2, D_3 et D_4 tel que $\forall i \in \{1,2,3,4\}, D_i = \{1,2,3,4\}$. Chaque valeur d'un domaine correspond à une colonne.*

- *Les contraintes du problème sont : $C_{12}, C_{13}, C_{14}, C_{23}, C_{24}, C_{34}$ telles que :*

  $C_{12} = \langle (X_1, X_2), R_{12} \rangle$

  $C_{13} = \langle (X_1, X_3), R_{13} \rangle$

  $C_{14} = \langle (X_1, X_4), R_{14} \rangle$

  $C_{23} = \langle (X_2, X_3), R_{23} \rangle$

  $C_{24} = \langle (X_2, X_4), R_{24} \rangle$

  $C_{34} = \langle (X_3, X_4), R_{34} \rangle$

- *Les relations des contraintes sont :*

  $R_{12} = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$

  $R_{13} = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\}$

  $R_{14} = \{(1,2), (1,3), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,2), (4,3)\}$

  $R_{23} = \{(1,3), (1,4), (2,4), (3,1)(4,1), (4,2)\}$

  $R_{24} = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4)(4,1), (4,3)\}$

  $R_{34} = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$

- *Une solution à ce problème est X_1=2, X_2=4, X_3=1, X_4=3.*

Exemple 1.1.2. *(Coloration de carte)*

Le coloriage de carte est un cas spécial du coloriage de graphe. Il s'agit de colorier toutes les régions d'une carte avec un ensemble de couleurs différentes de telle sorte que deux régions ayant une frontière en commun soient coloriées avec deux couleurs différentes. La figure1.1 illustre une instance possible de ce problème pour 3 couleurs.

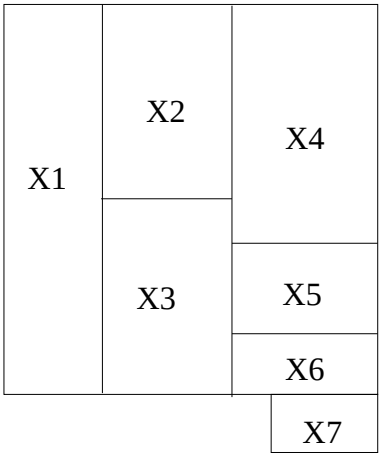


FIG. *1.1 – Exemple de 3-coloration*

Modélisation du problème

- *Les variables du problème sont les régions à colorier :* X_1, X_2, X_3, X_4, X_5, X_6, X_7.
- *Les domaines des variables sont* D_1, D_2, D_3, D_4, D_5, D_6 *et* D_7 *tel que* $\forall i \in \{1,2,3,4,5,6,7\}, D_i = \{rouge, vert, bleu\}$
- *Les contraintes sont* C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9 *et* C_{10} *telles que :*

  $C_1 = \langle (X_2, X_1), R_1 \rangle$

  $C_2 = \langle (X_3, X_1), R_2 \rangle$

  $C_3 = \langle (X_2, X_4), R_3 \rangle$

  $C_4 = \langle (X_2, X_3), R_4 \rangle$

  $C_5 = \langle (X_4, X_3), R_5 \rangle$

  $C_6 = \langle (X_5, X_4), R_6 \rangle$

  $C_7 = \langle (X_5, X_6), R_7 \rangle$

  $C_8 = \langle (X_5, X_3), R_8 \rangle$

  $C_9 = \langle (X_3, X_6), R_9 \rangle$

  $C_{10} = \langle (X_7, X_6), R_{10} \rangle$

  $\forall i \in 1, \cdots, 10, R_i = \{(rouge, bleu), (rouge, vert), (bleu, rouge), (bleu, vert), (vert, rouge), (vert, bleu)\}$.

  Une solution possible à ce problème est : X_1=*rouge,* X_2=*vert,* X_3=*bleu,*X_4=*rouge,* X_5=*vert,*

X_6=*rouge*, X_7=*bleu*.

1.2 Les méthodes de résolution

Les dernières années ont vu le développement de nombreuses méthodes et techniques pour résoudre les instances de CSP. Ces méthodes et techniques peuvent être classées comme suit :

- **Les techniques de simplification par filtrage**
  L'objectif des techniques de filtrage n'est pas de trouver la solution d'un CSP mais d'éviter l'exploration des régions de l'espace de recherche qui ne contiennent pas de solution.

- **Les méthodes incomplètes**
  Elles considèrent l'espace de recherche dans sa totalité mais elles ne l'explorent qu'en partie et ceci en exploitant des heuristiques pour choisir les zones d'exploration. Ces méthodes visent à donner un résultat acceptable en un temps raisonnable mais elles ne peuvent prouver l'inexistence de solution. Autrement dit, elles abordent la résolution d'un CSP comme un problème d'optimisation combinatoire pour lequel il s'agit de calculer une affectation satisfaisant le plus grand nombre de contraintes, l'objectif final étant de les satisfaire toutes. Les différentes approches possibles d'une résolution incomplètes sont : Recuit Simulé (RS), Recherche Taboue, Algorithmes Génétiques, Colonies de Fourmis, etc.

- **Les méthodes énumératives complètes**
  La résolution des instances CSP par des méthodes énumératives complètes se base, généralement, sur des algorithmes de recherche énumérative de type Backtrack.

- **Les méthodes de décomposition structurelles**
  L'exploitation des caractéristiques structurelles de l'instance traitée a donné naissance à une nouvelle approche de résolution qui est la résolution par décomposition. Cette approche exploite le fait que la traitabilité d'un CSP est liée aux caractéristiques topologiques de son graphe ou hypergraphe de contraintes. Ce résultat théorique est formalisé par un théorème dû à Freuder [29] et qui peut être résumé comme suit : "si un CSP est acyclique et arc consistant, il peut être résolu en appliquant un algorithme de degré polynomial". La résolution par décomposition est une technique qui analyse les CSPs et définit un schéma de décomposition avant la recherche d'une solution. Plusieurs méthodes

de décomposition ont été proposées dans la littérature (cette approche sera détaillée dans le prochain chapitre).

- **Les méthodes hybrides**
  Plusieurs algorithmes hybrides ont été proposés : hybridation d'une méthode énumérative et une Tree Décomposition (BTD [56]), hybridation d'une méthode énumérative et une méta-heuristique, etc.

Dans le reste de ce chapitre, nous présentons les principales techniques de filtrage puis nous présentons les méthodes énumératives car elles sont en relation avec l'objet de cette thèse. Le prochain chapitre sera consacré aux techniques de décomposition structurelles.

1.2.1 Filtrage

L'objectif du filtrage est de simplifier l'instance CSP à résoudre afin de réduire la taille de l'espace de recherche à explorer. A cet effet, plusieurs techniques de vérification de consistance locale ou globale et de propagation de contraintes ont été développées. Ces techniques permettent de réduire la taille des domaines des variables en enlevant certaines valeurs qui ne peuvent pas figurer dans une solution. Ceci permet d'accélérer la recherche de solutions du fait que les sous-arbres enracinés au niveau des valeurs enlevées ne sont pas explorés. Il est aussi possible de réduire la taille des relations par la suppression des tuples qui ne peuvent participer à aucune solution. La propriété de consistance est atteinte lorsque aucune valeur d'un domaine ou tuple d'une relation ne pourra être supprimé. Les valeurs ou les tuples à supprimer dépendent de la forme de la consistance utilisée.

1.2.1.1 Consistance de nœud NC

Une instance CSP est consistante de nœud si pour toute variable X_i et pour toute valeur d de D_i, l'affectation partielle $X_i = d$ satisfait toutes les contraintes unaires du problème. Son principe consiste à supprimer de chaque domaine D_i associé à la variable X_i, toute valeur qui viole une contrainte unaire.

1.2.1.2 Consistance d'arc AC

Un domaine D_i de la variable X_i est arc consistant si et seulement si pour chaque valeur $a \in D_i$, et pour toute variable X_j telle qu'il existe une contrainte C_{ij} entre X_i et X_j alors il existe une valeur b de D_j telle que (a, b) satisfait la contrainte C_{ij}. Un CSP binaire est arc

consistant si tous les domaines des variables sont arc consistants. Le filtrage par consistance d'arc consiste à supprimer des valeurs des domaines des variables qui ne satisfont pas la propriété de consistance d'arc. Le premier algorithme implémentant la consistance d'arc est AC1 (algorithme 2) proposé par Mackworth [63].

Algorithm 1 Réviser (C_k)

1: $C_k = (X_i, X_j)$
2: **for** tout $d_i \in D_i$ **do**
3: **if** $\nexists d_j \in D_j \mid (d_i, d_j) \in R_k$ **then**
4: supprimer d_i de D_i
5: $modification \longleftarrow True$
6: **end if**
7: **end for**
8: retourner *modification*

Algorithm 2 AC1

1: **repeat**
2: $modification \longleftarrow false$
3: **for** chaque contrainte C_k **do**
4: $modification \longleftarrow$ Réviser$(C_k) \vee modification$
5: **end for**
6: **until** *not modification*

L'inconvénient majeur de AC1 est que si une valeur d'un domaine est supprimée alors toutes les contraintes sont révisées même si le domaine modifié n'a aucune influence sur la plus part de ces contraintes. Une amélioration a été proposée dans AC3 par Mackworth [63]. Elle consiste à ne pas réappliquer la procédure *Réviser* à toutes les contraintes, mais uniquement à celles susceptibles d'être affectées par la suppression d'une valeur d'un domaine. Pour cela, AC3 (algorithme 3) utilise une file pour mémoriser ces contraintes (les contraintes à re-réviser).

Algorithm 3 AC3

1: $L \longleftarrow (X_i, X_j), i \neq j$
2: **while** $L \neq \emptyset$ **do**
3: choisir et supprimer de L un couple (X_i, X_j)
4: **if** Réviser$((X_i, X_j))$ **then**
5: $L \longleftarrow L \cup \{(X_k, X_i)\}$ / $\exists$ une contrainte liant X_k et X_i
6: **end if**
7: **end while**

Parmi les successeurs de AC3, nous pouvons citer : AC4 [66], AC5 [83], AC6[7], AC7[84] et les deux algorithmes AC2000 et AC2001 développés par Bessière et Régin [10].

Signalons que AC-6 et AC-2001 ont une complexité temporelle en $O(m \cdot d^2)$ et une complexité spatiale en $O(m \cdot d)$.

Remarque 1.2.1. La consistance d'arc peut être exploitée en prétraitement, mais aussi durant la recherche pour maintenir une certaine cohérence locale.

Remarque 1.2.2. Un CSP arc consistant n'est pas forcément cohérent.

1.2.1.3 Consistance d'arc directionnelle (DAC)

L'objectif de l'arc consistance directionnelle (Directional Arc Consistency DAC) est d'affaiblir l'arc consistance de telle sorte à réviser un arc dans une seule direction. Cela suppose la donnée d'un ordre $(X_1, \cdots, X_n)$ sur les variables du CSP. La définition suivante est valable pour les CSPs binaires.

Définition 1.2.1. (**Consistance d'arc directionnelle (DAC)**)
Un CSP est **arc consistant directionnel** pour un ordre donné sur les variables $(X_1, \cdots, X_n)$ ssi toute variable X_i est arc consistante avec toute autre variable X_j (avec $i < j$).

L'algorithme 4 de l'arc consistance directionnelle suppose que les variables du problème sont ordonnées.

Algorithm 4 DAC(G)

```
1: for  j=|nœuds(G)| downto 1 do
2:    for chaque arc (i,j) dans G | i < j  do
3:       Réviser(i,j)
4:    end for
5: end for
```

Il existe des instances CSP pour lesquelles DAC suffit. Par exemple quand l'instance CSP est acyclique alors cette forme de consistance (DAC) suffit pour résoudre le CSP sans backtrack. En général, on applique cet algorithme directionnel de la racine aux feuilles.

1.2.1.4 Consistance de chemin PC

La consistance de chemin (Path Consistency PC) est une forme de consistance plus forte que l'arc consistance. Un CSP est chemin-consistant PC si toute affectation consistante sur

deux variables peut être étendue de manière consistante à une troisième variable. La définition suivante est valable uniquement pour les CSPs binaires.

Définition 1.2.2. (**Consistance de chemin**)

Soit un CSP $P = (X, D, C)$, une paire de variable (X_i, X_j) est PC ssi $\forall X_k \in X$,$\forall (a, b) \in R_{ij}$, $\exists c \in D_k \mid (a, c) \in R_{ik}$ et $(b, c) \in R_{jk}$. Un CSP est PC ssi $\forall X_i, X_j \in X$, (X_i, X_j) est PC.

Remarque 1.2.3. Dans la définition 1.2.2, s'il n'existe pas de contrainte entre deux variables, on suppose qu'elles sont liées par une contrainte universelle (la relation de la contrainte est un produit cartésien entre leurs domaines respectifs).

Le filtrage par consistance de chemin permet de supprimer certains couples de valeurs des relations des contraintes. Si la contrainte correspondante n'existe pas, celle-ci est alors ajoutée **modifiant ainsi la structure du problème de départ**.

Plusieurs algorithmes ont été proposés pour la consistance de chemin, nous citons : PC-1 et PC-2 [63], PC-3 [66], PC-4 [50], etc.

L'algorithme PC-4 obtient une complexité optimale en $O(n^3d^3)$ et une complexité en espace en $O(n^3d^3)$. La consistance de chemin est surtout utilisée comme prétraitement car le maintien de cette propriété durant la recherche est trop coûteux.

Inconvénients de PC

Les principaux inconvénients de la consistance de chemin sont :
- sa complexité en espace,
- le maintien du CSP en extension,
- la modification du graphe de contraintes.

Remarque 1.2.4. Comme AC, PC n'est pas suffisante pour résoudre un CSP. Il existe des CSPs qui sont chemin consistants mais pas cohérents.

1.2.1.5 Consistance de chemin directionnelle DPC

Cette forme de consistance nécessite un ordre sur les variables. La définition 1.2.3 est valable pour les CSPs binaires.

Définition 1.2.3. (**Consistance de chemin directionnelle**)

Un CSP est chemin consistant directionnel DPC relativement à un ordre $(X_1, \cdots, X_n)$ si $\forall (X_i, X_j)$ et $\forall (a, b) \in R_{ij}$, $\forall k, k > i, j, \exists c \in D_k$ tq $(a, c) \in R_{ik}$ et $(b, c) \in R_{jk}$

1.2.1.6 Consistance de chemin restreinte RPC

Cette forme de consistance a été introduite par Berlandier [6]. Son objectif est d'avoir une consistance plus forte que la consistance d'arc mais sans les inconvénients de la consistance de chemin. La définition suivante s'applique aux CSPs binaires.

Définition 1.2.4. (**Consistance de chemin restreinte RPC**)

Soit un CSP $P = (X, D, C)$. $\forall X_i \in X, \forall a \in D_i \mid \forall X_j \in X$ tel que a a un support unique b dans D_j, $\forall X_k \in X$ telle que $C_{ik}, C_{jk} \in C$, $\exists c \in D_k$ telle que $(a, c) \in Rel(C_{ik}) \wedge (b, c) \in Rel(C_{jk})$.

Avec cette forme de consistance, la structure du problème n'est pas modifiée. En effet, RPC n'efface pas de paires de valeurs et donc aucune contrainte n'est ajoutée ou modifiée dans le CSP. RPC a été étendue à Max-RPC [19].

1.2.1.7 k-consistance

Les trois niveaux de consistance (NC, AC et PC) ont été généralisés.

Définition 1.2.5. (**k-consistance** [28])
Un CSP P est **k-consistant** si toute instanciation consistante de $k-1$ variables différentes peut être étendue en une instanciation consistante avec une autre variable supplémentaire. P est **fortement k-consistant** ssi P est j-consistant $\forall j \leq k$.

Remarque 1.2.5. La consistance de nœud est équivalente à la 1-consistance forte, la consistance d'arc est équivalente à la 2-consistance forte et la consistance de chemin est équivalente à la 3-consistance forte.

Pour établir la k-consistance, un algorithme en $O(n^k d^k)$ est présenté par Cooper [17]. En pratique, vu ces coûts prohibitifs, les consistances utilisées ne dépassent pas en général $k = 3$.

1.2.1.8 (i, j)-Consistance

La (i, j)-consistance [30] généralise la k-Consistance.

Définition 1.2.6. (**(i, j)-Consistance**)
un CSP P est **(i, j)-consistant** ssi toute instanciation consistante de i variables peut être étendue à j nouvelles variables. P est **(i, j)-fortement consistant**, ssi il est (k, j)-consistant pour tout $k \leq i$.

Remarque 1.2.6. La k-consistance est égale à la (k-1, 1)-consistance, l'AC est égale à la (1, 1)-consistance forte et la PC =(2, 1)-consistance forte.

1.2.1.9 Consistance inverse

La (1, k-1)-consistance est appelée ***k-consistance inverse*** [31]. On supprime les valeurs qui ne peuvent pas être étendues de manière consistante à k-1 nouvelles variables.
Quand i=1 et k=3, nous avons la ***consistance de chemin inverse (PIC)*** [31].
Une forme particulière de consistance inverse est ***la consistance de voisinage inverse (NIC)*** [31] : on supprime les valeurs de la variable X_i ne pouvant pas être étendues de manière consistante à toutes les variables directement reliées à X_i.

1.2.2 Filtrage des CSPs n-aires

Beaucoup de travaux ont été menés pour généraliser les techniques de filtrage aux CSPs n-aires. Dans cette sous-section, nous présentons l'arc consistance généralisée, l'interconsistance et la consistance relationnelle.

1.2.2.1 Arc Consistance généralisée GAC

Avant de décrire cette forme de consistance, nous présentons la notion de support.

Définition 1.2.7. (**Support pour une contrainte**)

Un **support** pour une contrainte C_i est un ensemble d'instanciations pour, exactement, l'ensemble des variables de C_i (Scope(C_i)) tel que C_i est satisfaite. Un support d'une contrainte C_i qui inclut $X_j = v$ est appelé **support de** $X_j = v$ dans C_i.

Définition 1.2.8. (**Arc consistance généralisée**)

Une contrainte C_i est **arc consistante généralisée (Generalized Arc consistent GAC)** s'il existe des supports pour toutes les valeurs des domaines de toutes les variables de Scope(C_i). Un problème P est **GAC** si toutes ses contraintes sont **GAC**.

1.2.2.2 Algorithme GAC 2001

GAC 2001 (algorithme 5) [12] est une généralisation de AC 2001 pour le filtrage des CSPs n-aires en se basant sur la consistance d'arc généralisée définie précédemment. GAC 2001 opère sur un CSP dont les tuples des relations sont supposés être totalement ordonnés et le dernier support de chaque couple (X_i, a) pour une contrainte C_j est sauvegardé dans une variable ***Last***$((X_i, a), C_j)$ initialisée à NIL. Ainsi, lors de la recherche du 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 ont déjà fait l'objet d'une vérification.

1.2.2.3 Interconsistance

Définition 1.2.9. (**Interconsistance**)

Un CSP $P = (X, D, C)$ est dit **interconsistant** ssi toutes les relations sont compatibles deux

Algorithm 5 GAC 2001

1: **for** chaque variable X_i **do**
2: **for** chaque valeur a de D_i **do**
3: **for** chaque contrainte C_j **do**
4: $Last((X_i, a), C_j) = NIL$
5: **end for**
6: **end for**
7: **end for**
Q = $\{(X_i, C_j) \mid C_j \in C$ et $X_i \in Scope(C_j)\}$
8: **while** Q non vide **do**
9: choisir (X_i,C_j) de Q
10: $Q = Q - \{X_i, C_j\}$
11: **if** REVISE2001(X_i, C_j) **then**
12: $Q = Q \cup \{(X_k, C_m) | C_m \in C, X_i, X_k \in Scope(C_m)$ et $k \neq i$ et $j \neq m\}$
13: **end if**
14: **end while**
Procédure REVISE 2001(X_i,C_j)
15: supprime=False
16: **for** chaque $a_i \in D_i$ **do**
17: $\tau = Last((X_i, a), C_j)$
18: **if** $\exists k \mid \tau[X_{j_k}] \notin [D_{j_k}]$ **then**
19: $\tau = succ(\tau, D_{X_i=a}^{Scope(C_j)})$
20: **while** $\tau \neq NIL$ and $\tau \notin rel(C_j)$ **do**
21: $\tau = succ(\tau, D_{X_i=a}^{Scope(C_j)})$
22: **end while**
23: **if** $\tau \neq NILL$ **then**
24: $Last((X_i, a), Cj) = \tau$
25: **else**
26: supprimer a de D_i
27: supprime=True
28: **end if**
29: **end if**
30: **end for**
31: return supprime

à deux. $\forall Ci, Cj \in C \mid Scope(Ci) \cap Scope(Cj) \neq \emptyset$ et $Ri, Rj \neq \emptyset$,
$Ri[Scope(Ci) \cap Scope(Cj)] = Rj[Scope(Ci) \cap Scope(Cj)] \neq \emptyset$

1.2.2.4 Consistance relationnelle

Dans la consistance relationnelle [25], l'entité primitive est la contrainte (relation) au lieu de la variable. Donc elle s'applique aux CSPs binaires et n-aires.
Soient $C_i = \langle S_i, R_i \rangle$ et $C_j = \langle S_j, R_j \rangle$ deux contraintes.

Définition 1.2.10. (**Contrainte relationnellement arc consistante**)
C_i est dite *relationnellement arc consistante* relativement à un sous-ensemble de variables $A \subseteq S_i$ ssi toute instanciation des variables de A peut être étendue à un tuple de R_i. Une contrainte $C_r = \langle S_r, R_r \rangle$ est *relationnellement arc consistante* si elle est relationnellement arc consistante relativement à tout sous-ensemble de variables $A \subseteq S_r$. Un CSP est *relationnellement arc consistant* si toutes ses contraintes sont relationnellement arc consistantes.

Définition 1.2.11. (**Contrainte relationnellement chemin consistante**)
C_i et C_j sont dites *relationnellement chemin consistantes* relativement à un sous-ensemble de variables $A \subseteq S_i \cup S_j$ ssi toute instanciation consistante des variables de A peut être étendue à toutes les variables de $S_i \cup S_j$ satisfaisant simultanément C_i et C_j. C_i et C_j sont *relationnellement chemin consistantes* si elles sont relationnellement chemin consistantes relativement à n'importe quel sous-ensemble A de $S_i \cup S_j$. Un CSP est *relationnellement chemin consistant* si chaque paire de ses contraintes sont relationnellement chemin consistantes.

Définition 1.2.12. (**Contraintes relationnellement m-consistantes**)
Soient $C_1 = \langle S_1, R_1 \rangle, \cdots, C_m = \langle S_m, R_m \rangle$ m contraintes. $C_1, \cdots, C_m$ sont *relationnellement m-consistantes* relativement à un sous-ensemble de variables $A \subseteq S_1 \cup \cdots \cup S_m$, ssi toute instanciation des variables de A peut être étendue à $S_1 \cup \cdots \cup S_m$ de telle sorte que les contraintes $C_1, \cdots, C_m$ soient simultanément satisfaites. Un sous-ensemble de contraintes $\{C_1, \cdots, C_m\}$ sont dites relationnellement m-consistantes ssi elles sont relationnellement m-consistantes pour n'importe quel sous-ensemble de variables $A \subseteq S_1 \cup \cdots \cup S_m$. Un CSP est relationnellement m-consistant ssi tout ensemble de m contraintes est relationnellement m-consistant.

Définition 1.2.13. (**CSP fortement (i,m)-consistant relationnel**)

Soient $C_1 = \langle S_1, R_1 \rangle$, $\cdots$, $C_m = \langle S_m, R_m \rangle$ m contraintes. C_1, $\cdots$, C_m sont *relationnellement (i,m)-consistantes* ssi elles sont relationnellement m-consistantes relativement à tout ensemble de variables A de taille i. Un CSP est relationnellement (i,m)-consistant si tout ensemble de m contraintes est relationnellement (i,m)-consistant. Un CSP est fortement (i,m)-consistant relationnel s'il est (j, m)-consistant relationnel pour tout $j \leq i$.

Définition 1.2.14. (**m-consistance directionnelle relationnelle**)

Etant donné un ensemble de variables ordonné $\sigma = X_1, \cdots, X_n$, un CSP est *relationnellement m-consistant directionnel* ssi pour tout l, tout sous-ensemble de contraintes $\{C_1, \cdots, C_m\}$ dont la variable ayant le plus large index est X_l et pour tout sous-ensemble $A \subseteq \{X_1 \cup \cdots \cup X_{l-1}\}$, toute instanciation de A peut être étendue à X_l satisfaisant simultanément toutes les contraintes.

L'algorithme 6 est un algorithme de consistance relationnelle ou $RC_{(i,m)}$ [25] proposé pour assurer la (i,m)-consistance relationnelle sur un CSP.

Remarque 1.2.7. Dans l'algorithme 6, les symboles π et $\bowtie$ désignent respectivement les opérations de projection et de jointure issues des bases de données. R_{S_i} désigne la contrainte C_i dont le scope est S_i et la relation est R_{S_i}. $\Re$ désigne l'ensemble des contraintes (relations).

Algorithm 6 Consistance relationnelle ($\Re$,i,m)($RC_{(i,m)}$)

1: **repeat**
2: $Q \leftarrow \Re$
3: **for** chaque m relations $R_{S_1}, \cdots, R_{S_m} \in Q$ et chaque sous-ensemble de variables A_i de taille i, $A_i \subseteq \cup_{j=1}^{m} S_j$ **do**
4: $R_{A_i} \leftarrow R_{A_i} \cap \pi_{A_i}(\bowtie_{j=1}^{m} R_{S_j})$
5: **if** R_{A_i} est une relation vide **then**
6: exit et retourner un réseau vide
7: **end if**
8: **end for**
9: **until** $Q = \Re$

Une amélioration de l'algorithme 6 est d'assurer la consistance relationnelle dans une seule direction. Dans ce cadre, un algorithme appelé $DRC_{(i,m)}$ (algorithme 7) a été proposé dans [25]. Etant donné un ordre $o = X_1, \cdots, X_n$ sur les variables, l'algorithme $DRC_{(i,m)}$ partitionne les

contraintes en buckets. Dans le bucket associé à la variable X_j, on place toutes les contraintes dont l'index le plus haut est X_j. Les buckets sont traités dans un ordre décroissant et les relations résultantes sont placées dans les buckets d'un ordre inférieur.

Algorithm 7 Consistance directionnelle relationnelle ($\Re$,i,m,o)($DRC_{(i,m)}$)

1: Initialize : générer une partition ordonnée des contraintes, $bucket_1, \cdots, bucket_n$ avec $bucket_i$ contient toutes les contraintes ayant X_i comme la variable la plus élevée.
2: **for** $p \leftarrow n$ downto 1 **do**
3: **for** chaque $S_i, S_j \in bucket_p$ tel que $S_i \supseteq S_j$ **do**
4: $R_{S_i} \leftarrow \pi_{S_i}(R_{S_i} \bowtie R_{S_j})$
5: **end for**
6: **if** $bucket_p$ contient la contrainte $X_p = u$ **then**
7: **for** chaque $S_i \in bucket_p$ **do**
8: $A \leftarrow S_i - \{X_p\}$
9: $R_A = \pi_A(\sigma_{X_p=u} R_{S_i})$
10: **if** R_A n'est pas vide **then**
11: ajouter R_A au bucket approprié
12: **else**
13: exit et retourner un réseau vide
14: **end if**
15: **end for**
16: **else**
17: $j \leftarrow min\{$cardinalité de $bucket_p, m\}$
18: **for** chaque j relations $R_{S_1}, \cdots, R_{S_j} \in bucket_p$ **do**
19: $F_j \leftarrow \bowtie_{i=1}^{j} R_{S_i}$
20: **for** chaque sous-ensemble de variables A de taille i, $A \subseteq \cup_{t=1}^{j} S_t - \{X_p\}$ **do**
21: $R_A \leftarrow \pi_A F_j$
22: **if** R_A n'est pas vide **then**
23: ajouter R_A à son bucket approprié
24: **else**
25: exit et retourner un réseau vide
26: **end if**
27: **end for**
28: **end for**
29: **end if**
30: **end for**
31: return $E_{(i,m)}(R) = \cup_{j=1}^{n} bucket_j$

1.2.3 Algorithmes énumératifs

L'algorithme de base le plus répandu pour une recherche systématique est celui du **Backtrack** chronologique (BT).

1.2.3.1 Le retour-arrière chronologique (Backtrack)

A chaque étape, l'instanciation partielle courante $\{X_1 = d_1, \cdots, X_{k-1} = d_{k-1}\}$ est étendue à $\{X_1 = d_1, \cdots, X_k = d_k\}$ en instanciant une nouvelle variable X_k avec une valeur $d_k \in D_k$ compatible avec l'instanciation partielle courante. Si aucune valeur du domaine de X_k n'est compatible, alors il y a un retour-arrière vers la variable X_{k-1} pour essayer une autre valeur $d'_{k-1} \in D_{k-1}$. On obtient alors l'instanciation partielle suivante : $\{X_1 = d_1, \cdots, X_{k-1} = d'_{k-1}\}$. Le principe du BackTrack [38] (BT) est décrit par l'algorithme 8. Sa complexité est bornée par la taille de l'espace de recherche et qui est égale à d^n avec n nombre de variables et d est la taille du plus grand domaine (le nombre maximum des valeurs par domaine). Par conséquent, pour un problème de m contraintes, la complexité théorique du Backtrack est $O(m \cdot d^n)$.

Algorithm 8 Backtrack

Input : un CSP $P = (X, D, C)$, et une affectation A. Initialement $A = \emptyset$.
Output : une solution du CSP si le CSP est consistant.

```
1:  if A est non consistante then
2:     retourner False
3:  else
4:     if A est une instanciation totale then
5:        retourner True
6:     else
7:        choisir une nouvelle variable X_i de X
8:        for Chaque valeur d_i ∈ D_i do
9:           if Backtrack(P = (X, D, C), A ∪ {X_i = d_i}) then
10:             retourner True
11:          end if
12:       end for
13:    end if
14: end if
```

1.2.3.2 Algorithmes avec retour arrière non chronologique

Lorsque un échec survient, l'algorithme BT remet en cause la dernière instanciation. En effet, après avoir essayé toutes les valeurs possibles pour la variable courante X_i, il revient en arrière vers la variable précédente X_{i-1}. Cependant, rien ne garantit que X_{i-1} a une quelconque responsabilité dans l'inconsistance, par conséquent essayer de nouvelles valeurs pour cette variable conduira aux mêmes échecs. Pour remédier à ce problème, le saut en arrière (Backjump) a été proposé et consiste à faire un retour-arrière non chronologique (intelligent) sur

la variable responsable de l'échec, plutôt que d'effectuer un retour en arrière vers la (i-1)ème variable. Parmi les algorithmes exploitant le retour-arrière non chronologique, nous citons les algorithmes suivants.

- **Algorithme de BackJumping (BJ)** [35]
  C'est un algorithme qui fait un retour-arrière sur la variable la plus profonde qui est en conflit avec la variable courante X_i (dont la valeur est incompatible avec une valeur de la variable courante), si toutes les instanciations de cette dernière sont inconsistantes.

- **Graph-based Backjumping** [20]
  Cet algorithme exploite le graphe des contraintes. En cas d'échec d'extension cohérente sur une variable X_i, un retour-arrière est effectué vers la dernière variable X_j (la dernière instanciée dans le voisinage de X_i). Si la valeur actuelle de la variable X_j était la dernière, un retour-arrière se produit vers la variable la plus profonde appartenant au voisinage de X_j ou au voisinage de toute variable instanciée après X_j qui soit une cause possible de l'échec.

- **Algorithme Conflict Directed BackJumping (CD-BJ) [73]**
  L'algorithme CD-BJ maintient pour chaque variable instanciée un ensemble de conflits. Cet ensemble inclut toutes les variables (instanciées avant la variable courante X_i) qui sont en conflit avec X_i pour une valeur de cette dernière, ou qui sont la cause de l'échec de l'extension d'une affectation contenant X_i. Dans cet algorithme, le retour-arrière se produit vers la dernière variable instanciée dans l'ensemble des conflits de X_i.

1.2.3.3 Heuristiques d'ordre

Pour améliorer les techniques énumératives, deux types d'heuristiques ont été explorées.

1. **Heuristiques sur l'ordre d'instanciation des variables**
   La taille de l'espace de recherche dépend de l'ordre d'affectation des variables. Même si trouver un bon ordre n'est pas facile, plusieurs heuristiques permettant de définir des ordres de qualité sont proposées. Elles sont basées sur le principe du "first-fail" consistant à faire les choix les plus contraints d'abord pour détecter rapidement les échecs. L'ordre peut être statique ou dynamique. Dans le cas statique, il est prédéfini avant la résolution du problème et il repose généralement sur les propriétés structurelles du problème : degré, tailles des domaines et satisfiabilité des contraintes. Parmi ces heuristiques, nous pouvons citer :

- *Maxdeg [21] :* les variables sont ordonnées selon un ordre décroissant sur les degrés (le nombre de contraintes portant sur chaque variable).
- *MinCon [21] :* la première variable est choisie arbitrairement. La prochaine variable à affecter est celle qui a le plus grand nombre de variables voisines déjà ordonnées.

Dans le cas dynamique, les ordres sont calculés durant la recherche. Parmi ces heuristiques, nous pouvons citer :

- *dom+deg [32] :* elle consiste à choisir la variable ayant le domaine courant de taille (nombre de valeurs) minimum et en cas d'égalité choisir celle qui a le degré maximum.
- *dom/deg [11] :* la prochaine variable à instancier est celle qui minimise le ratio entre la taille du domaine courant et le degré.

Citons aussi les heuristiques multi-niveaux [8]. Ces heuristiques tiennent compte des voisinages des variables afin d'orienter la recherche vers les parties difficiles du CSP.

2. **Heuristiques sur l'ordre des valeurs à affecter en priorité aux variables**
   Parmi ces heuristiques, nous citons *min-conflict* [32] qui consiste à instancier la variable courante par la valeur qui a le nombre minimum de valeurs incompatibles avec les valeurs des variables non encore instanciées. Ainsi, le choix de valeurs est selon l'ordre croissant des nombres de conflits.

1.2.3.4 Algorithmes avec filtrage avant

Une amélioration possible pour l'algorithme du BackTrack BT est de le combiner avec les techniques de filtrage. Dans BT, la vérification de consistance s'effectue en fonction des variables déjà instanciées. Une autre technique consiste à exploiter le concept de consistance en avant. Suivant ce concept, à chaque fois qu'une nouvelle variable X_i est instanciée, les valeurs dans les domaines des variables qui ne sont pas encore instanciées et qui ne sont pas compatibles avec la valeur affectée à X_i sont supprimées. Notons que le choix de la propriété de consistance à maintenir n'est pas évident car plus la propriété de consistance maintenue est forte, plus la complexité de l'algorithme de filtrage est grande.

Dans ce qui suit, nous présentons deux algorithmes de ce type : Forward Checking (FC) et Maintaining Arc Consistency (MAC).

Algorithme Forward Checking FC

L'algorithme FC [51] (algorithme 9) améliore l'algorithme du BackTrack (BT) en appliquant un filtrage avant. Il vérifie la consistance entre la variable courante et celles qui ne sont pas encore instanciées dans son voisinage, en filtrant leurs domaines par la suppression des valeurs incompatibles avec la valeur courante. Il ne construit que des affectations consistantes (succès) car les affectations inconsistantes sont en fait éliminées par le filtrage anticipé. La complexité en temps de FC, dans le pire des cas, est identique à celle de BT, c'est-à-dire en $O(m \cdot d^n)$. En pratique par contre, FC obtient généralement de meilleurs résultats que BT.

Algorithm 9 FC

Input : le quadruplet (A,NA,D,C), avec A est une affectation, initialement A = {}, NA est l'ensemble des variables non encore instanciées, initialement $NA = X$, D est l'ensemble des domaines des variables et C est l'ensemble des contraintes.
Output : une solution du CSP si le problème est consistant, sinon le CSP est inconsistant.

1: **if** $NA = \{\}$ **then**
2: retourner A /* A est une solution*/
3: **else**
4: choisir X_i de NA
5: **repeat**
6: choisir une valeur v de D_i
7: $D_i = D_i - \{v\}$
8: **if** $A \cup \{X_i, v\}$ ne viole aucune contrainte **then**
9: $D' = Revise(NA - \{X_i\}, D, C, \langle X_i, v\rangle)$
10: **if** aucun domaine de D' n'est vide **then**
11: $result = FC(NA - \{X_i\}, A \cup \langle X_i, v\rangle, D', C)$
12: **if** $result \neq NULL$ **then**
13: retourner (result)
14: **end if**
15: **end if**
16: **end if**
17: **until** $D_i = \{\}$
18: retourner(NULL)
19: **end if**
Procedure Revise (W, D, C, a)
a est une affectation d'une variable.
20: $D' = D$
21: **for** chaque variable X_j dans W **do**
22: **for** chaque valeur v dans D'_j **do**
23: **if** $\langle X_j, v\rangle$ est incompatible avec les contraintes de C **then**
24: $D'_j = D'_j - \{v\}$
25: **end if**
26: retourner D'
27: **end for**
28: **end for**

Algorithme nFC

nFC [9] généralise *FC* au cas n-aire. Il est basé sur la définition de deux ensembles :

1. $C^n_{c,f}$: composé des contraintes qui contiennent la variable courante et au moins une variable du futur.
2. $C^n_{p,f}$: composé des contraintes qui contiennent au moins une variable du passé et au moins une variable du futur.

Plusieurs approches existent pour nFC 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 de contraintes ou à chaque contrainte séparément.

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 effectuer 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 effectuer 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 effectuer 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 effectuer un retour-arrière.

Algorithme Maintaining Arc Consistency MAC [76] :

Dans MAC (algorithme 10), la consistance d'arc est maintenue tout au long de la recherche. Contrairement à FC qui limite la vérification de la consistance au voisinage de la dernière variable instanciée, MAC propage cette vérification sur toutes les variables du CSP qu'elles soient en relation avec la variable instanciée ou pas, tout en garantissant un maintien de la consistance. Ainsi, à chaque étape, MAC applique un algorithme de filtrage AC-x.

Il existe plusieurs versions du MAC selon l'algorithme AC utilisé, MAC 3 s'il utilise le AC3, MAC 2001 pour AC 2001 et ainsi de suite. Notons qu'il existe des algorithmes qui effectuent plus de propagations que FC et moins que MAC : le partial looking ahead [51] ou le directional arc consistency lookahead [81].

1.2.3.5 Algorithmes Backtrack avec mémorisation

Pour la compréhension de ces heuristiques, nous avons besoin des définitions suivantes.

Algorithm 10 MAC

Input : le quadruplet (A,NA,D,C), avec A est une affectation, initialement vide $A = \{\}$, NA est l'ensemble des variables non encore instanciées, initialement NA = X, D est l'ensemble des domaines et C celui des contraintes.
Output : *true* si le CSP est arc consistent, *false* sinon.

1: **if** $NA = \{\}$ **then**
2: retourner A /* A est une solution*/
3: **else**
4: choisir une variable X_i de NA
5: **repeat**
6: choisir une valeur v de D_i
7: $D_i = D_i - \{v\}$
8: **if** $A \cup \{X_i, v\}$ est consistante **then**
9: $D' = Revise(NA - \{X_i\}, D, C, \langle X_i, v\rangle)$ /*Revise est similaire à celle de FC*/
10: AC-x$(NA - \{X_i\}, D', C)$ /*D' est filtré à l'aide d'un algorithme AC*/
11: **if** aucun domaine de D' n'est vide **then**
12: *result*=MAC$(A \cup \{\langle X_i, v\rangle\}, NA - \{X_i\}, D', C)$
13: **if** $result \neq NULL$ **then**
14: retourner (result)
15: **end if**
16: **end if**
17: **end if**
18: **until** $D_i = \{\}$
19: retourner(NULL)
20: **end if**

Définition 1.2.15. (**Conflit**)

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

Définition 1.2.16. (**Good et Nogood**)

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

Les algorithmes avec mémorisation mémorisent des instanciations qui ne peuvent pas être étendues en une solution. La mémorisation de ces nogoods permet de ne pas les régénérer une autre fois et ainsi de ne pas explorer les mêmes sous-arbres. Cependant, l'inconvénient majeur de ces méthodes est l'espace mémoire requis pour l'enregistrement des Goods et Nogoods. Parmi les algorithmes qui adoptent la technique de mémorisation, on peut citer Nogood Recording (NR) [77], Learning Tree-Solve [4] et Dynamic Backtracking (DBT) [37].

1.3 CSPs n-aires

Deux approches sont proposées dans la littérature pour la résolution des CSPs n-aires.

1.3.1 Transformation de CSPs n-aires en CSPs binaires

Trois principales approches sont proposées pour cette binarisation des CSPs n-aires.

- **Représentation duale :** cette transformation consiste à utiliser chaque contrainte du CSP original comme une variable dans le nouveau CSP binaire. Son domaine est l'ensemble des tuples permis par sa relation correspondante. Ainsi, chaque couple de variables du nouveau CSP qui représentent deux contraintes partageant des variables dans le CSP original seront reliées par une contrainte dans le nouveau CSP binaire. Soient X_{C_i} et X_{C_j} deux variables duales associées respectivement avec les deux contraintes C_i et C_j, alors un tuple $t \in Rel(C_i)$ est supporté dans la relation de la nouvelle contrainte liant X_{C_i} et X_{C_j} s'il existe un tuple $t' \in Rel(C_j)$ tel que $t[Scope(C_i) \cap Scope(C_j)] = t'[Scope(C_i) \cap Scope(C_j)]$.
- **Représentation en variables cachées [75] :** elle consiste à introduire des variables supplémentaires, appelées h-variables, en plus des variables initiales du CSP. Il y a une

variable duale v_i pour chaque contrainte C_i du CSP. Le domaine de chaque variable duale est l'ensemble des tuples de la relation de la contrainte correspondante. Pour chaque variable duale v_i, une contrainte binaire est introduite pour faire le lien entre v_i et chaque variable initiale $X_j \in Scope(C_i)$. Un tuple $\tau \in Dom(v_i)$ est supporté dans la relation de la contrainte entre v_i et X_j ssi il existe une valeur $a \in Dom(X_j)$ telle que $\tau[X_j] = a$.

- **Représentation double [78] :** cette représentation combine la représentation duale et la représentation en variables cachées. Les variables du nouveau CSP sont les variables initiales du CSP non binaire, les variables duales et les h-variables. En plus des contraintes introduites dans la représentation en variables cachées, d'autres contraintes binaires sont introduites entre chaque paire de variables duales dont les contraintes correspondantes dans le CSP non binaire initial partagent au moins une variable.

1.3.2 Résolution directe des CSPs n-aires

Cette approche consiste à résoudre le CSP directement dans sa formulation d'origine. Ceci nécessite bien sûr l'extension et l'adaptation des techniques et méthodes développées pour les CSPs binaires. Un algorithme nFC et certaines techniques de filtrage développées pour les CSPs n-aires ont été présentés dans la section précédente. Un exemple de cette approche consiste à appliquer l'algorithme *nFC2* directement.

1.4 Conclusion

Dans ce chapitre, nous avons présenté le cadre CSP ainsi que les différentes approches de résolution proposées dans la littérature. Nous avons présenté surtout les méthodes énumératives qui pour être efficaces doivent s'appuyer sur des heuristiques et sur des techniques de filtrage. Cependant même si ces méthodes énumératives sont relativement efficaces en pratique, leurs bornes de complexité théorique sont très mauvaises (exponentielles en n : nombre de variables du problème). Dans le prochain chapitre, nous allons présenter les techniques de décomposition structurelles permettant de borner cette complexité théorique.

Chapitre 2

Les techniques de décomposition structurelles

Dans ce chapitre, nous présentons les notions de base relatives aux graphes et hypergraphes, puis nous présentons les principales méthodes de décomposition structurelles proposées dans la littérature pour résoudre des instances CSP.

2.1 Graphes et hypergraphes

Définition 2.1.1. (**Graphe**)

Un *graphe* G est une paire (V, E) où V est l'ensemble des sommets du graphe et E l'ensemble de ses arêtes.

Définition 2.1.2. (**Connexité**)

Un graphe est *connexe* si et seulement s'il existe un chemin entre chaque paire de ses sommets.

Définition 2.1.3. (**Composante connexe**)

Une *composante connexe* CC d'un graphe G =(V,E) est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par un chemin : si $x \in CC$, alors

- $\forall y \in CC$, il existe un chemin reliant x à y,
- $\forall z \in V \setminus CC$, il n'existe pas de chemin reliant x à z.

Définition 2.1.4. (**Séparateur**)

Soit $G = (V, E)$ un graphe. Un *séparateur* de G est un sommet ou un ensemble de sommets dont la suppression induit l'augmentation du nombre de composantes connexes.

Définition 2.1.5. (**k-connexité**)

Un graphe G est dit *k-connexe* si la suppression de $k-1$ sommets n'augmente pas le nombre de composantes connexes. Une composante k-connexe de G est un sous-graphe maximal de G k-connexe. Une composante bi-connexe est une 2-composante connexe de G.

Définition 2.1.6. (**Hypergraphe**)

Un *hypergraphe* est un couple $\langle V, E\rangle$ où V est l'ensemble des sommets et E est l'ensemble des hyperarêtes. Chaque hyperarête est un sous-ensemble de V. Si toutes les hyperarêtes sont de dimension deux alors l'hypergraphe est un graphe.

Définition 2.1.7. (**Graphe primal**)

Le *graphe primal* d'un hypergraphe $\mathcal{H} = \langle V, E\rangle$ est un graphe $G = (V, E')$ dont les sommets sont ceux de V et il existe une arête entre deux sommets dans G si ces derniers participent à une même hyperarête dans $\mathcal{H}$.

Définition 2.1.8. (**Graphe dual**)

Le *dual d'un hypergraphe* $\mathcal{H} = \langle V, E\rangle$ est un graphe $G = (E, E')$ dont les sommets $e_1, \cdots, e_m$ correspondent aux hyperarêtes de $\mathcal{H}$ et il existe une arête $\{e_i, e_j\} \in E'$ si les hyperarêtes $e_i, e_j \in E$ partagent au moins une variable (ie : $e_i \cap e_j \neq \emptyset$).

Définition 2.1.9. (**Chaîne**)

Une *chaîne* de longueur q dans un hypergraphe $\langle V, E\rangle$ est définie comme une séquence $(x_1, e_1, \cdots, x_q, e_q)$ où $x_1, \cdots, x_q$ sont des sommets distincts de V et $e_1, \cdots, e_q$ sont des hyperarêtes distinctes de E et x_k , $x_{k+1} \in e_k$.

Une *chaîne* est dite cyclique si $x_1 = x_{q+1}$. Un hypergraphe est un *hyperarbre* s'il ne contient aucune chaîne cyclique.

2.2 Traitabilité

Définition 2.2.1. Un problème P est dit *traitable* s'il existe un algorithme polynomial permettant de répondre à la question de P.

La *traitabilité d'une instance* CSP peut être liée aux *propriétés des relations de contraintes* ou *à la structure topologique de l'instance CSP*. Dans le cadre de cette thèse, nous nous intéressons uniquement à la traitabilité liée à la structure du CSP : il a été démontré [29] que la classe de CSPs dont l'hypergraphe de contraintes est acyclique est traitable.

Dans ce paragraphe, nous présentons les moyens de reconnaître l'acyclicité d'une structure d'un CSP.

2.2.1 Acyclicité d'un hypergraphe

L'algorithme 11 dû à Graham [44], appelé ***GYO reduction***, permet de vérifier si un hypergraphe est acyclique. Si l'algorithme de Graham retourne un ensemble vide d'hyperarêtes

Algorithm 11 GYO reduction

1: **Input :** un hypergraphe $\mathcal{H}$.
2: **Output :** $\mathcal{H}$ cyclique ou non
3: **while** $\mathcal{H}$ contient des (hyper)arêtes **do**
4: Supprimer tous les sommets appartenant uniquement à une (hyper)arête
5: Supprimer les (hyper)arêtes vides où celles complètement contenues dans une autre (hyper)arête et ajouter ces (hyper)arêtes à l'ordre (n'importe quel ordre)
6: **end while**

alors l'hypergraphe est *acyclique*. Sinon il est cyclique.

Remarque 2.2.1. Il existe d'autres caractérisations de l'acyclicité.

Définition 2.2.2. (**Join tree**)

Un ***Join tree*** (arbre de jointure) pour un hypergraphe $\mathcal{H} = \langle V, E \rangle$ est un arbre T dont les nœuds sont les hyperarêtes de $\mathcal{H}$, tels que si un sommet x de $\mathcal{H}$ appartient à deux hyperarêtes différentes e_i et e_j dans $\mathcal{H}$, alors x apparaît dans chaque nœud sur le chemin unique reliant e_i et e_j dans T.

Notons qu'un hypergraphe est **acyclique** si et seulement s'il a un join tree [5].

Définition 2.2.3. (**Graphe triangulé**)

Un graphe est dit ***triangulé*** si tout cycle de longueur supérieure strictement à 3 a une corde (une arête qui joint deux sommets non consécutifs).

Définition 2.2.4. (**Hypergraphe conforme**)

Un hypergraphe de contraintes d'un CSP est dit ***conforme*** si les cliques maximales de son graphe primal correspondent à des contraintes du CSP.

Un hypergraphe est **acyclique** s'il est *conforme* et que son graphe primal est *triangulé* [5].

Pour tirer profit des avantages de l'acyclicité, il est intéressant de trouver des classes plus larges que celles dont l'hypergraphe est acyclique. Pour cela, plusieurs travaux ont été menés et beaucoup de méthodes ont été développées.

2.3 Les principales méthodes structurelles

2.3.1 Principe de ces méthodes

Une méthode de décomposition structurelle transforme une instance CSP en une autre instance équivalente (ayant les mêmes solutions) en formant des clusters de variables ou de contraintes dont l'interaction a une structure d'arbre. Chaque méthode de décomposition D associe à chaque hypergraphe $\mathcal{H}$ un paramètre D-largeur (D-width) tel que pour une constante k, toute instance CSP dont l'hypergraphe de contraintes a une D-largeur inférieure à k peut être résolue en un temps polynomial (la résolution n'est plus exponentielle en la taille du problème mais en fonction de cette largeur).

Dans ce qui suit, nous présentons les principales méthodes proposées dans la littérature.

2.3.2 La méthode CCM (Cycle cutset)

Cette méthode est proposée dans [22]. Un ensemble cutset (coupe cycle) d'un graphe est un ensemble de sommets dont la suppression induit un graphe acyclique. La mesure de cyclicité de cette méthode est appelée *cutset width* et elle est égale à la taille (nombre de sommets) de l'ensemble coupe cycle le plus petit. CCM est basée sur le fait que l'affectation des variables de l'ensemble coupe cycle rend le graphe de contraintes acyclique.

La taille de l'ensemble coupe cycle constitue la hauteur du backtracking nécessaire à la résolution du problème.

La complexité temporelle de cette méthode est en $O(d^{k+2})$ où k est la taille de l'ensemble coupe cycle et d est la taille du plus large domaine. En effet, le coût nécessaire à la recherche

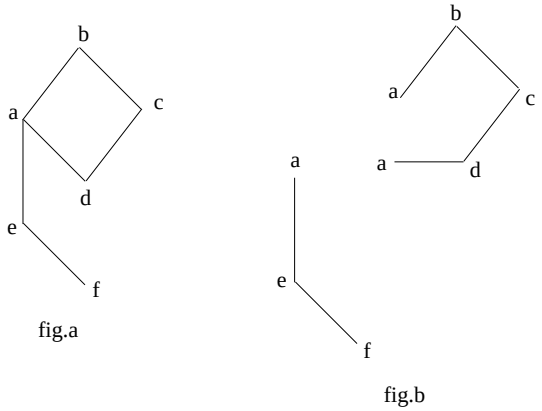


FIG. *2.1 – Un ensemble coupe cycle*

d'une affectation consistante sur l'ensemble coupe cycle est $O(d^k)$. Pour chaque affectation consistante, il faut résoudre le CSP acyclique induit. La complexité en temps de cette tâche est en $O(d^2)$ et donc la complexité totale est $O(d^{k+2})$. La complexité totale en espace est linéaire.

Exemple 2.3.1. *Considérons le graphe (fig.a) de la figure 2.1. L'instanciation de la variable a supprime le cycle a-b-c-d-a, donnant lieu alors à un graphe acyclique de la figure 2.1 (fig.b).*

2.3.3 La méthode Cycle Hypercutset

Cette méthode a été introduite dans [40]. C'est une simple modification de la méthode CCM où l'ensemble CUTSET est composé d'hyperarêtes au lieu de sommets de l'hypergraphe. Un ensemble hypercutset d'un hypergraphe $\mathcal{H}$ est un ensemble d'hyperarêtes $\mathcal{H}'$ de $\mathcal{H}$ telles que le sous-hypergraphe induit par var($\mathcal{H}$) - var($\mathcal{H}'$) est acyclique.

La largeur d'un hypergraphe est la cardinalité minimale de tous les hypercutsets possibles.

Exemple 2.3.2. *Considérons l'hypergraphe de la figure 2.2.*

Il est évident que pour rendre cet hypergraphe acyclique, il faudra supprimer une des hyperarêtes C1 ou C2 ou C3 ou C4.

Pour résoudre un CSP par cette méthode, on procède comme suit : on détermine le plus petit ensemble hypercutset puis on réalise la jointure des contraintes contenues dans cet ensemble. Cela donnera des solutions au sous-problème généré par les contraintes de l'ensemble

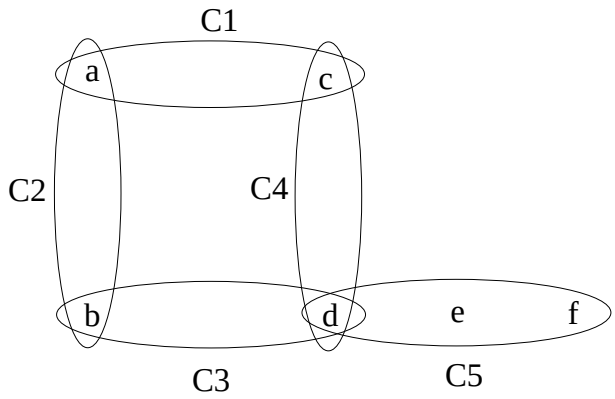


FIG. *2.2 – Un hypergraphe* $\mathcal{H}$

hypercutset. Par la suite, on propage les valeurs assignées aux variables partagées avec l'autre composante (non hypercutset). On résout alors la composante acyclique. Si aucune solution n'est trouvée, on répète l'opération jusqu'à ce qu'il n'y ait plus de solution au sous-problème représenté par les contraintes de l'ensemble hypercutset.

2.3.4 La méthode BICOMP (Biconnected Components)

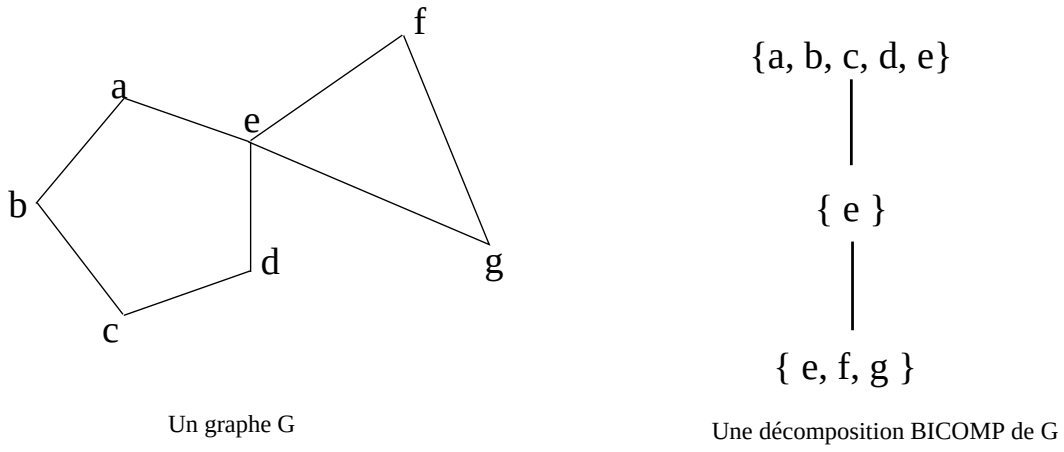
Soit G=(V,E) un graphe. Une décomposition BICOMP [30] de G est un couple $\langle T, \chi \rangle$ où T est un arbre et χ est une fonction d'étiquetage qui associe à chaque sommet de T un ensemble de sommets S de G tels que S soit une composante bi-connexe de G ou un sommet singleton contenant un séparateur de G. Il y a une arête {p,q} dans l'arbre T si $\chi(p)$ est une composante bi-connexe de G et $\chi(q)$ contient un séparateur de G appartenant à $\chi(p)$ ($\chi(q) \subseteq \chi(p)$). Pour un hypergraphe, sa décomposition BICOMP est celle de son graphe primal. La figure 2.3 donne un exemple de décomposition par cette méthode.

Il a été démontré que la décomposition BICOMP peut être calculée en un temps linaire [30]. La *largeur* bi-connexe de l'hypergraphe $\mathcal{H}$ est le nombre maximum de sommets à travers les composantes bi-connexes du graphe primal de $\mathcal{H}$.

Pour résoudre des CSPs avec cette technique, un algorithme BCC a été proposé dans [3]. Pour présenter cet algorithme, nous avons besoin des définitions suivantes.

Définition 2.3.1. (**Ordre naturel des nœuds**)

Un ordre naturel des nœuds de l'arbre est un ordre où chaque nœud précède tous ses nœuds



Un graphe G

Une décomposition BICOMP de G

FIG. 2.3 – Exemple de décomposition par la méthode BICOMP

fils.

Définition 2.3.2. (**Ordre BCC-compatible des variables**)

Un ordre BCC-compatible des variables est un ordre compatible avec un ordre naturel des nœuds de l'arbre.

Définition 2.3.3. (**Accesseur d'une bi-composante**)

Pour un ordre BCC-compatible, l'*accesseur* d'une bi-composante est sa plus petite variable dans l'ordre. Cette variable est un séparateur du graphe de contraintes. L'accesseur d'une variable est l'accesseur de la plus petite bi-composante à laquelle elle appartient.

Avec BCC, les variables sont instanciées selon un ordre statique BCC-compatible. Si E_i et E_j sont des bi-composantes telles que $i < j$, alors les variables de E_i sont instanciées avant celles de E_j. BCC évite certaines redondances en marquant certaines valeurs des accesseurs si on a déjà prouvé qu'elles pouvaient être étendues de façon consistante sur les prochaines variables dans l'ordre. Ainsi, la prochaine fois que ces valeurs seront affectées à ces variables, un saut en avant (forward-jump) sera effectué vers une partie non explorée de l'espace de recherche. Par contre, si une valeur d'un accesseur ne peut être étendue de façon consistante sur une partie des prochaines variables alors elle est supprimée du domaine de cette variable.

De plus, lorsqu'un échec se produit, BCC effectue un retour-arrière non chronologique vers la dernière variable instanciée pouvant être responsable de l'échec.

La complexité temporelle de l'algorithme BCC est en $O(n \cdot d^k)$ avec k la taille de la plus grande bi-composante, d est la taille du plus grand domaine et n est le nombre de composantes bi-connexes.

2.3.5 La méthode *TCLUSTER* (Tree clustering)

La méthode de tree clustering [23] procède en construisant un CSP défini à partir des clusters (regroupements) de variables de telle sorte que l'interaction entre clusters soit structurée en arbre. Ce nouveau CSP est équivalent à celui d'origine (ils ont les mêmes solutions), sauf que l'hypergraphe de contraintes associé maintenant est acyclique. Donc la propriété des CSPs acycliques est applicable.

Rappelons qu'un CSP est acyclique si et seulement si son hypergraphe de contraintes est conforme et son graphe primal est triangulé.

TCLUSTER est basée sur un algorithme de triangulation [80] qui transforme n'importe quel graphe (primal) en un graphe triangulé en lui ajoutant des arêtes. Les cliques maximales du graphe triangulé obtenu correspondent aux clusters nécessaires pour former un CSP acyclique.

La description de *TCLUSTER* est donnée par l'algorithme 12.

Exemple 2.3.3. *[23] Soit le CSP défini sur les variables A, B, C, D, E. Les portées (scopes) des contraintes sont : (A,C), (A,D), (B,D), (C,E), (D,E). Le graphe primal est donné par la figure 2.4.*

Un ordre d = E, D, C, A, B est possible par l'algorithme Maximum Cardinality Search précédent (voir la figure 2.5, fig.a). Par cet ordre, une arête (C, D) est ajoutée au graphe pour obtenir finalement un graphe triangulé (figure 2.5, fig.b). Les cliques maximales de ce graphe sont : {E,C,D}, {A,C,D}, {B,D}. Le graphe dual et le join tree associés sont donnés par la figure 2.6.

Pour résoudre le problème initial, on commence par résoudre les sous-problèmes associés aux ensembles de variables {A, D, C}, {D, C, E} et { D,B}. Puis l'ensemble de ces solutions

Algorithm 12 *TCLUSTER*

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

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

constitue les domaines pour ces c- variables. L'arbre est alors résolu d'une manière gloutonne (sans retour-arrière). Par exemple, la résolution du sous-problème A, D, C consiste à trouver les instanciations aux variables A, D, et C qui soient consistantes avec les contraintes (A, C) et (A, D).

La complexité totale de cette méthode est bornée par $O(n \cdot w \cdot d^w)$ où d est la taille du plus grand domaine des variables et w est la taille de la plus grande clique. La complexité en espace est $O(n \cdot s \cdot d^s)$ où s est la taille de la plus grande intersection entre sous-problèmes.

2.3.6 La méthode TD (Tree decomposition)

La décomposition arborescente TD (Tree Decomposition) [74] d'un graphe $G = (V, E)$ est une paire $\langle T, \chi \rangle$ où $T(N, F)$ est un arbre et χ est une fonction d'étiquetage qui associe à chaque sommet p de N un ensemble de sommets $\chi(p) \subseteq V$ satisfaisant les conditions suivantes :

1. Pour chaque sommet a de G, il existe un sommet p de N tel que $a \in \chi(p)$
2. Pour chaque arête $\{a, b\} \in E$, il existe un sommet p de N tel que $\{a, b\} \subseteq \chi(p)$
3. Pour chaque sommet a de G, l'ensemble $\{p \in N / a \in \chi(p)\}$ induit un sous-arbre connecté.

La largeur d'une décomposition arborescente est $max_{p \in N} |\chi(p) - 1|$, la Treewidth de G est la largeur minimale de toutes ses décompositions arborescentes. La treewidth d'un hypergraphe est la treewidth de son graphe primal. Si la treewidth d'un hypergraphe est 1 alors l'hypergraphe est acyclique.

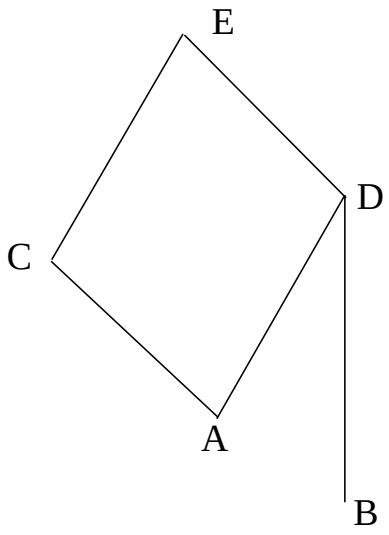


FIG. *2.4 – Graphe primal de l'hypergraphe de contraintes*

Pour exploiter cette technique pour la résolution des CSPs, plusieurs algorithmes sont proposés dans la littérature.

2.3.7 La méthode BTD (Backtracking on Tree-Decomposition)

La méthode BTD (algorithme 13) est proposée par Jégou et Terrioux [56]. Elle exploite une décomposition arborescente TD (tree decomposition) pour résoudre des CSPs. De ce fait, la première étape de BTD consiste à calculer une décomposition TD.

Définition 2.3.4. (**Good et Nogood**)
Soient E_i un cluster et E_j l'un de ses fils. Un **Good** (resp. **Nogood**) structurel de E_i par rapport à E_j est une affectation consistante sur $E_i \cap E_j$ telle qu'il existe (resp. n'existe pas) d'extension consistante de cette affectation sur Desc(E_j) où Desc(E_j) est l'ensemble des variables du sous-arbre enraciné au niveau de E_j.

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

Algorithm 13 BTD

Input : l'affectation $\mathcal{A}$ courante, initialement $\mathcal{A} = \emptyset$
Input : le cluster E_i courant, initialement $E_i = E_1$
Input : V_{E_i} : ensemble des variables du cluster E_i qui ne sont pas encore instanciées.
Output : *True* si $\mathcal{A}$ a été étendue avec succès à toutes les variables de la descendance de E_i, *False* sinon.

1: **if** $V_{E_i} = \emptyset$ **then**
2: $Consistency \leftarrow True$
3: $F = \text{Sons}(E_i)$ /*Sons(E_i) désigne les fils de E_i*/
4: **while** $F \neq \emptyset$ and $Consistency$ **do**
5: choisir E_j in F
6: $F = F \setminus \{E_j\}$
7: **if** $\mathcal{A}[E_i \cap E_j]$ est un good **then**
8: $Consistency = True$
9: **else**
10: **if** $\mathcal{A}[E_I \cap E_j]$ est un nogood **then**
11: $Consistency \leftarrow False$
12: **else**
13: $Consistency \leftarrow BTD(\mathcal{A}, E_j, E_j \setminus (E_i \cap E_j))$
14: **if** $Consistency$ **then**
15: Enregistrer le good $\mathcal{A}[E_i \cap E_j]$
16: **else**
17: Enregistrer the nogood $\mathcal{A}[E_i \cap E_j]$
18: **end if**
19: **end if**
20: **end if**
21: **end while**
22: Return $Consistency$
23: **else**
24: Choisir $x \in V_{E_i}$
25: $d_x = D_x$
26: $Consistency \leftarrow False$
27: **while** $d_x \neq \emptyset$ et $\neg Consistency$ **do**
28: Choisir v dans d_x
29: $d_x \leftarrow d_x \setminus \{v\}$
30: **if** $\nexists C_i \in C$ tel que C_i n'est pas satisfaite par $\mathcal{A} \cup \{x \leftarrow v\}$ **then**
31: $Consistency \leftarrow BTD(\mathcal{A} \cup \{x \leftarrow v\}, E_i, V_{E_i \setminus \{x\}})$
32: **end if**
33: **end while**
34: Return $Consistency$
35: **end if**

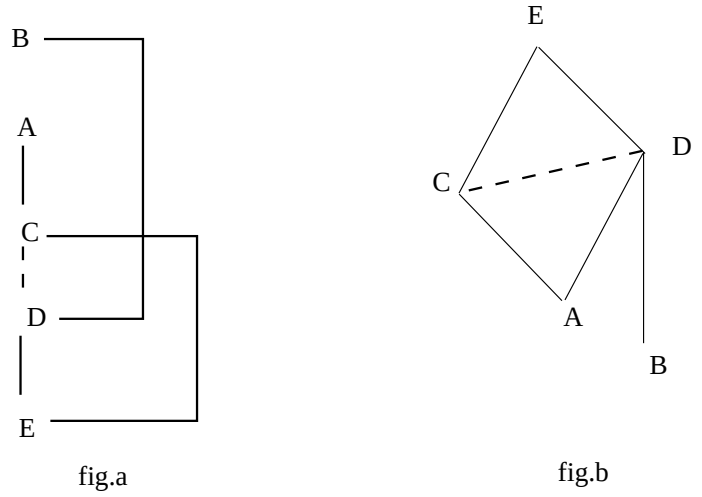


FIG. 2.5 – Triangulation du graphe. L'arête en pointillés dans fig.b est ajoutée pour trianguler le graphe de la figure 2.4.

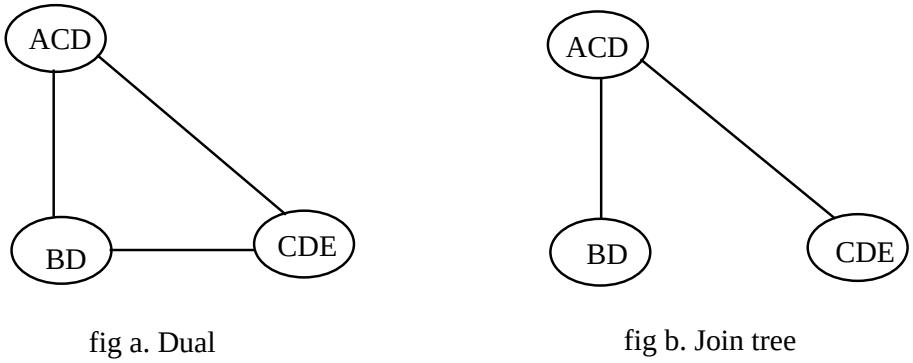


FIG. 2.6 – Graphe dual et join tree du graphe de la figure 2.4

L'algorithme BTD (algorithme 13) procède comme suit : au cours de la recherche, il choisit une variable x non instanciée du cluster courant E_i (ligne 24), si elle existe, et lui affecte une valeur v (ligne 28). Si $\mathcal{A} \cup \{x \leftarrow v\}$ est inconsistante, BTD choisit une nouvelle valeur pour x (boucle while : lignes 27-33). Sinon, l'instruction BTD($\mathcal{A} \cup \{x \leftarrow v\}$, $E_i, V_{E_i} \setminus \{x\}$) (ligne 31) va essayer d'étendre $\mathcal{A} \cup \{x \leftarrow v\}$ sur $V_{E_i} \setminus \{x\}$ (V_{E_i} désigne l'ensemble des variables du cluster E_i). En cas d'échec, une nouvelle valeur pour x est considérée. Si toutes les valeurs ont été essayées, un retour-arrière est effectué sur la variable précédente. Si toutes les variables de E_i ont été instanciées de manière consistante, alors BTD choisit un fils E_j de E_i (ligne 5). Trois cas se présentent alors :

1. Si $\mathcal{A}[E_i \cap E_j]$ est un good (ligne 7), alors $\mathcal{A}$ peut être étendue de manière consistante

sur le sous-problème enraciné en E_j. Dans ce cas, un saut en avant "forward-jump" est effectué pour continuer l'énumération avec la première variable après celles de $Desc(E_j)$ dans l'ordre induit par la décomposition.

2. Si $\mathcal{A}[E_i \cap E_j]$ est un nogood (ligne 10), alors $\mathcal{A}$ ne peut être étendue en une solution, alors l'affectation actuelle des variables de $E_i \cap E_j$ doit être modifiée.
3. Si $\mathcal{A}[E_i \cap E_j]$ n'est ni un *good* ni un *nogood*, alors BTD va tenter d'étendre $\mathcal{A}$ sur les variables du sous-arbre enraciné en E_j (ligne 13). En cas de succès, BTD enregistre $\mathcal{A}[E_i \cap E_j]$ comme un good (ligne 15), sinon $\mathcal{A}[E_i \cap E_j]$ est enregistré comme un nogood (ligne 17).

Si $\mathcal{A}$ a été étendue de façon consistante sur toutes les variables du sous-arbre enraciné en E_i alors BTD retourne *True*, sinon il retourne *False* (lignes 4-21). Finalement, BTD$(\emptyset, E_1, V_{E_1})$ retourne donc *True* si l'instance P est consistante, *False* sinon.

La complexité temporelle de BTD est en $O(exp(w+1))$ tandis que sa complexité spatiale est en O$(n \cdot s \cdot exp(s))$ avec s la taille de la plus grande intersection entre clusters et w est la largeur de la décomposition arborescente utilisée.

2.3.8 La méthode Hinge (HINGE decomposition)

La méthode de décomposition Hinge (HINGE) est proposée par Gyssens et al. [46] pour la décomposition des hypergraphes.

2.3.8.1 Définitions

Définition 2.3.5. (**Ensemble d'arêtes connexes**)
Soit $\mathcal{H}$ un hypergraphe, $H \subseteq edges(\mathcal{H})$ ($edges(\mathcal{H})$ désigne les hyperarêtes de $\mathcal{H}$) et $F \subseteq edges(\mathcal{H}) - H$. F est dit connexe relativement à H si pour chaque paire d'arêtes $e, f \in F$, il existe une séquence $e_1, \cdots, e_n$ d'arêtes dans F telles que :

- $e_1 = e$;
- pour $i = 1, \cdots, n-1, e_i \cap e_{i+1}$ n'est pas incluse dans $\cup_{h \in H} h$
- $e_n = f$.

Définition 2.3.6. (**Composantes connexes**)

Les sous-ensembles connexes maximaux de $E - H$ sont dits composantes connexes de $E - H$ relativement à H.

Définition 2.3.7. (**Hinge**)

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

Définition 2.3.8. (**Décomposition Hinge**)

Soit $\mathcal{H} = \langle V, E \rangle$ un hypergraphe, une ***décomposition Hinge*** de $\mathcal{H} = \langle V, E \rangle$ est un arbre vérifiant toutes les conditions suivantes :

1. Les nœuds de l'arbre sont les hinges minimaux de $\mathcal{H} = \langle V, E \rangle$.
2. Chaque arête dans E est contenue dans au moins un nœud de l'arbre.
3. Deux nœuds adjacents A et B de l'arbre partagent précisément une arête L dans E. De plus, L consiste en l'ensemble des variables partagées par A et B.
4. Les sommets de V partagés par deux nœuds x et y sont entièrement contenus dans tout nœud sur le chemin x-y.

2.3.8.2 Algorithme de calcul de la décomposition Hinge

L'algorithme 14 est proposé dans [46] pour le calcul de la décomposition Hinge. La complexité en temps de l'algorithme 14 qui calcule une décomposition HINGE est $O(|V| \cdot |E|^2)$. D'autre part, il a été démontré que la cardinalité du sommet le plus large constitue un invariant de l'hypergraphe et est appelée la ***HINGE width***. Ce nombre est appelé aussi degré de cyclicité [46].

La procédure de résolution d'un CSP via une décomposition HINGE se compose des étapes suivantes :

1. Trouver un hinge tree T pour le CSP.
2. Résoudre les sous-problèmes correspondants aux nœuds de T.

Algorithm 14 calcul de la décomposition Hinge

Input : un hypergraphe $\langle V, E\rangle$
Output : un hinge tree T

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

3. Résoudre le CSP global sans retour-arrière comme dans la méthode de Dechter pour le Tree Clustering.

La complexité de cette procédure est en $O(|V| \cdot |E|^2) + O(|E| \cdot r^w \cdot w \cdot logr)$, où r est la taille maximale des relations des contraintes dans C et w est le degré de cyclicité de l'hypergraphe $\langle V, E\rangle$. L'algorithme ci-dessus est indiqué pour les sous-problèmes ayant un degré de cyclicité faible. Quand ce degré est important, les sous-problèmes engendrés deviennent larges. Alors il serait plus efficace de combiner cet algorithme avec d'autres techniques pour traiter ces sous-problèmes. Un exemple de telle combinaison est la décomposition Hinge combinée avec Tree Clustering [46].

2.3.9 La méthode GHD (Generalized Hypertree Decomposition)

La méthode de décomposition en hyperarbre généralisée (Generalized Hypertree Decomposition) GHD est proposée dans [1, 41].

2.3.9.1 Définitions

Définition 2.3.9. (**Hyperarbre**)

Soit $\mathcal{H} = \langle V, E\rangle$ un hypergraphe. Un hyperarbre (hypertree) de $\mathcal{H}$ est un triplet $\langle T, \chi, \lambda\rangle$, avec

$T = (N, B)$ un arbre enraciné, χ et λ sont des fonctions d'étiquetage qui à chaque sommet $p \in N$ de T associent deux ensembles $\chi(p) \subseteq V$ et $\lambda(p) \subseteq E$.

Pour chaque sous-ensemble d'hyperarêtes $K \subseteq E$, on désignera par $var(K) = \cup_{e \in K} e$ les sommets appartenant aux hyperarêtes de K, $var(\mathcal{H}) = V$ et edges$(\mathcal{H}) = E$.
Si $T' = (N', E')$ est un sous-arbre de T, on définit $\chi(T') = \cup_{v \in N'} \chi(v)$ et $\lambda(T') = \bigcup_{v \in N'} \lambda(v)$. On désigne l'ensemble des nœuds N de T par $sommets(T)$ ou $vertices(T)$ et la racine de T par $root(T)$. $\forall p \in N$, T_p désigne le sous-arbre de T enraciné au niveau de p. $Parent(p)$ est le nœud parent de p dans T.

Définition 2.3.10. (Décomposition GHD)

Une **décomposition en hyperarbre généralisée** GHD d'un hypergraphe $\mathcal{H} = \langle V, E \rangle$ est un hyperarbre $\langle T, \chi, \lambda \rangle$ qui satisfait les conditions suivantes :

1. Pour chaque hyperarête $h \in E$, il existe $p \in sommet(T)$ telle que $var(h) \subseteq \chi(p)$. On dit que p couvre h.
2. Pour chaque variable $v \in V$, l'ensemble $\{p \in sommets(T) | v \in \chi(p)\}$ induit un sous-arbre connexe de T.
3. Pour chaque sommet $p \in sommets(T)$, $\chi(p) \subseteq var(\lambda(p))$.

La figure 2.7 montre un exemple d'hypergraphe et une décomposition en hyperarbre généralisée.

La ***décomposition en hyperarbre (Hypertree Decomposition)*** [40] HD d'un hypergraphe $\mathcal{H} = \langle V, E \rangle$, est une GHD $\langle T, \chi, \lambda \rangle$ qui satisfait la condition supplémentaire suivante : *Pour chaque sommet* $p \in sommets(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

La largeur d'une décomposition en hyperarbre (généralisée) $\langle T, \chi, \lambda \rangle$ est $max_{p \in vertices(T)} |\lambda(p)|$. La largeur en hyperarbre (généralisée) $(g)htw(\mathcal{H})$ d'un hypergraphe $\mathcal{H}$ est la largeur minimale de toutes ses décompositions en hyperarbre (généralisée).

Définition 2.3.11. Une hyperarête h est **fortement couverte** dans un hyperarbre $\langle T, \chi, \lambda \rangle$ s'il existe un nœud p tel que $var(h) \subseteq \chi(p)$ et $h \in \lambda(p)$.

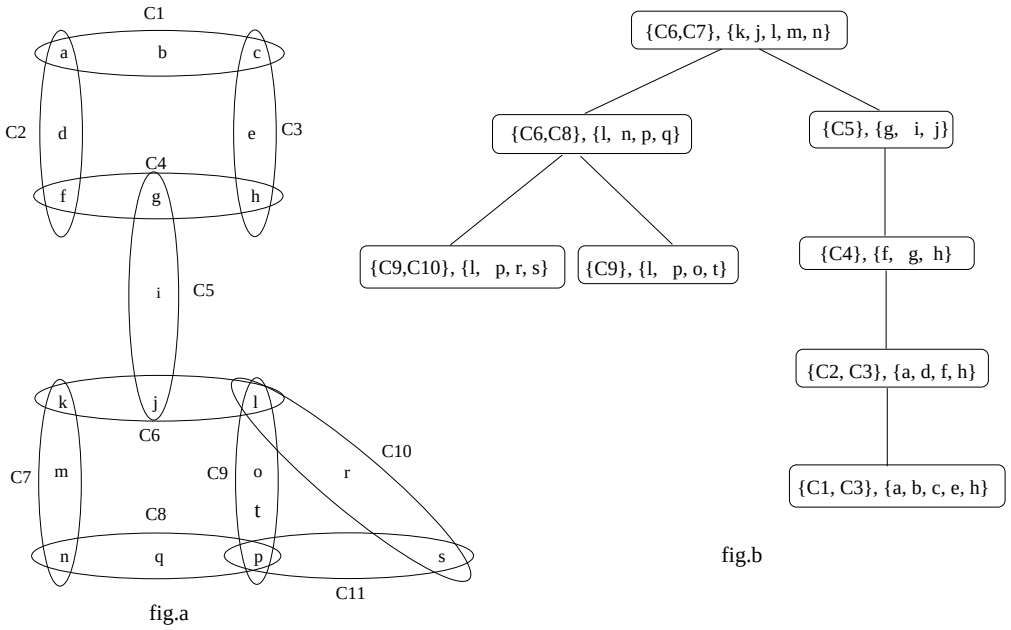


FIG. 2.7 – Un hypergraphe $\mathcal{H}$ (fig.a) et une GHD de $\mathcal{H}$. (fig.b)

Définition 2.3.12. (**GHD complète**)

Une décomposition en hyperarbre (généralisée) $\langle T, \chi, \lambda \rangle$ d'un hypergraphe $\mathcal{H} = \langle V, E \rangle$ est **complète** si chaque hyperarête h de $\mathcal{H} = \langle V, E \rangle$ est fortement couverte dans $\langle T, \chi, \lambda \rangle$.

Calculer une GHD optimale est un problème NP-complet [42].

Remarque 2.3.1. Les hypergraphes acycliques ont une largeur en hyperarbre (généralisée) égale à 1.

Remarque 2.3.2. Si on pose la troisième condition de la définition de la GHD comme égalité ($p \in sommets(T)$, $\chi(p) = var(\lambda(p))$), nous obtenons une **query decomposition** [13]. Cette dernière vérifie même les conditions de la décomposition en hyperarbre.

2.3.9.2 Calcul d'une (G)HD

Deux approches sont proposées dans la littérature pour calculer une décomposition en hyperarbre (généralisée) : les méthodes exactes et les méthodes heuristiques.

- **Méthodes exactes** : le premier algorithme proposé pour trouver une décomposition en hyperarbre optimale est *opt-k-decomp* [39]. Etant donné un hypergraphe $\mathcal{H} = \langle V, E \rangle$ et une constante k, *opt-k-decomp* calcule une décomposition en hyperarbre de $\mathcal{H}$ de largeur

$\leq k$ (si $htw(\mathcal{H}) \leq k$). Si $htw(\mathcal{H}) > k$, cet algorithme retourne échec. *opt-k-decomp* a une complexité temporelle de $O(m^{2k}n^2)$ où m est le nombre d'hyperarêtes, n est le nombre de sommets et k est une constante. Pour améliorer *opt-k-decomp*, beaucoup d'algorithmes sont proposés. Nous pouvons citer *Red-k-decomp* [52], *det-k-decomp* [43] et l'algorithme proposé par Subbarayan et Reif Anderson [79] qui est une version backtrack de *opt-k-decomp*.

Malheureusement ces méthodes exactes ont un inconvénient majeur. Leur exploitation nécessite beaucoup de mémoire et beaucoup de temps d'exécution. Cela fait que ces méthodes ne sont pas efficaces en pratique pour des instances de tailles réalistes. Pour contourner ce problème, des méthodes heuristiques sont proposées.

- **Méthodes heuristiques et méta-heuristiques :** beaucoup de méthodes heuristiques et méta-heuristiques ont été proposées pour calculer une décomposition en hyperarbre (généralisée). Korimort [61] a proposé une heuristique basée sur la connectivité des sommets de l'hypergraphe. Dermaku et al. [26] ont proposé les heuristiques suivantes : *BE* (Bucket Elimination), *DBE* (Dual Bucket Elimination) et les techniques de partitionnement de l'hypergraphe. Musliu et Schahauser [68] ont utilisé les algorithmes génétiques, etc.

Ci-après, nous présentons brièvement l'algorithme BE car nous l'utiliserons aussi dans nos expérimentations dans les prochains chapitres.

Algorithme BE : l'algorithme *BE* a été utilisé avec succès pour le calcul d'une décomposition arborescente d'un graphe (ou du graphe primal d'un hypergraphe). Cet algorithme a été étendu dans [26] pour le calcul des décompositions en hyperarbre généralisées. L'idée derrière cette extension est qu'une GHD satisfait les propriétés d'une décomposition arborescente TD. Donc pour calculer une GHD, *BE* procède comme suit : il construit d'abord une décomposition arborescente puis il crée le λ label pour chaque nœud de l'hyperarbre de telle sorte à satisfaire la troisième condition de la définition de la décomposition en hyperarbre généralisée (définition 2.3.10) en couvrant les variables dans le χ label par le minimum possible d'hyperarêtes. Notons que pour être efficace, *BE* nécessite un ordre sur les variables.

2.3.9.3 Résolution des instances CSP via une (G)HD

Pour résoudre des instances CSP via une (G)HD, Gottlob et al. [40] ont proposé l'algorithme 15 appelé dans cette thèse *GLS*. (GLS acronyme de **G**ottlob, **L**eone et **S**carcello).

Algorithm 15 *GLS*

Input : une instance CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ et son hypergraphe de contraintes $\mathcal{H}$
Output : une solution de P si elle existe

1: Calculer une GHD de $\mathcal{H}$
2: Compléter la décomposition GHD obtenue
3: Pour chaque nœud p, calculer une nouvelle contrainte (relation) R_p dont les tuples sont la projection sur les variables de $\chi(p)$ de la jointure des relations des contraintes dans $\lambda(p)$
4: Résoudre l'instance acyclique obtenue par n'importe quel algorithme proposé pour les instances acycliques.

Les principales étapes de *GLS* sont :

1. Premièrement, une décomposition GHD de l'hypergraphe de contraintes $\mathcal{H}$ est calculée par n'importe quelle méthode.

2. Deuxièmement, la décomposition GHD obtenue à l'étape (1) est transformée en une décomposition GHD complète de $\mathcal{H}$ comme suit. Pour chaque contrainte C_i, non fortement couverte, choisir un nœud p de l'hyperarbre tel que $Scope(C_i) \subseteq \chi(p)$ (ce nœud existe grâce à la condition (1) de la définition 2.3.10) puis créer un nœud q comme fils de p avec $\chi(q) = Scope(C_i)$ et $\lambda(q) = C_i$. Cette opération est nécessaire sinon les contraintes manquantes ne sont pas prises en considération et donc l'algorithme n'est pas "sound".

3. La troisième étape associe à chaque nœud p une nouvelle contrainte C_p dont la relation est la projection sur $\chi(p)$ de la jointure des relations des contraintes dans $\lambda(p)$, et $Scope(C_p) = \chi(p)$. Après cette opération, on obtient un join tree d'un hypergraphe acyclique d'une instance P' qui est solution-équivalente à P.

4. Résoudre P' (donc P) par n'importe quel algorithme proposé pour les instances acycliques. Dans ce travail, nous considérons l'algorithme de Yannakakis [85] à cet effet.

L'algorithme 16 implémentant la troisième et quatrième étapes de *GLS* est appelé *Join Acyclic Solving algorithm* (*JAS*) dans la suite de cette thèse. *JAS* utilise les opérations des bases de données suivantes. Soient C_i et C_j deux contraintes telles que : Scope(C_i) = S_i, Rel(C_i)=R_i, Scope(C_j) = S_j et Rel(C_j)=R_j. Soient t et t' deux tuples tels que $t \in R_i$ et

$t' \in R_j$.

- L'opération de jointure $\bowtie$
  $t \bowtie^t t' = t$". t" est un tuple défini sur $S_i \cup S_j$ *avec* $t"[S_i] = t$ *et* $t"[S_j] = t'$.[1, 2]
  $R_i \bowtie R_j = \{t" \mid t"$ *est un tuple* défini sur $S_i \cup S_j$ *avec* $t"[S_i] \in R_i$ *et* $t"[S_j] \in R_j\}$.
- L'opération de semi-jointure $\ltimes$
  $R_i \ltimes R_j = \Pi_{S_i}(R_i \bowtie R_j) = \{t \in R_i \mid \exists t' \in R_j$ *avec* $t'[S_i \cap S_j] = t[S_i \cap S_j]\}$.

L'algorithme *JAS* a deux étapes.

1. **Etape 1** : après avoir fixé un ordre de résolution des sous-problèmes (ligne 1), tous les sous-problèmes associés aux nœuds de la GHD sont résolus séparément par les opérations de jointure et de projection (lignes 2-4).
2. **Etape 2** : le join tree est transformé en un join tree arc consistant directionnel relationnel. Chaque tuple t de la relation de la contrainte associée à chaque nœud n_i qui n'a pas de tuples compatibles dans la relation de la contrainte associée à chaque nœud fils de n_i est supprimé. Ceci grâce à l'opération de semi-jointure effectuée de bas en haut (lignes 5-7), et finalement le CSP est résolu sans retour-arrière (lignes 8-10). Notons que cette étape correspond à une variante de l'algorithme de Yannakakis.

Algorithm 16 *Join Acyclic Solving*

Input : Une GHD complète $\langle T, \chi, \lambda \rangle$ associée à une instance CSP P
Output : Une solution $\mathcal{A}$ de cette instance CSP si elle existe

1: $\sigma \leftarrow (n_1, n_2, \ldots, n_e)$ /* σ : pre-order sur les nœuds de T avec n_1 la racine et e est le nombre de nœuds de T */
2: **for** chaque nœud n_i dans σ **do**
3: $\quad R_{n_i} \leftarrow (\bowtie_{C_j \in \lambda(n_i)} Rel(C_j))[\chi(n_i)]$
4: **end for**
5: **for** $i = e$ **downto** 2 **do**
6: $\quad R_{n_j} \leftarrow R_{n_j} \ltimes R_{n_i}$ /*n_j : parent de n_i dans T*/
7: **end for**
8: **for** $i = 1$ **to** e **do**
9: $\quad$ Construire une solution $\mathcal{A}$ en choisissant un tuple dans R_{n_i} compatible avec tous les tuples choisis précédemment
10: **end for**
11: Retourner $\mathcal{A}$

[1] Les crochets [ ] désignent l'opérateur de projection : $t"[S_i]$ est la projection du tuple $t"$ sur les variables de S_i.
[2] $t"$ *exists* ssi $t[S_i \cap S_j] = t'[S_i \cap S_j]$.

2.3.10 Les décompositions gardées et Spread Cuts

Le formalisme Décompositions gardées (Guarded Decompositions) et la méthode Spread Cuts sont proposés dans [15]. Nous présentons d'abord ce formalisme, puis nous présentons la méthode de décomposition Spread Cut SCD et la méthode Spread Cut_{New} SCD_{New} qui est légèrement différente de SCD.

2.3.10.1 Les décompositions gardées

Définition 2.3.13. (**Bloc gardé**)

Un ***bloc gardé*** (guarded block) d'un hypergraphe $\mathcal{H} = \langle V, E \rangle$ est une paire (λ, χ) où le *garde* λ est un sous-ensemble d'hyperarêtes de E et le *bloc* χ est un sous-ensemble de sommets formé de l'union des sommets des hyperarêtes de λ.

Définition 2.3.14. (**Recouvrement gardé complet**)

Un bloc gardé (λ, χ) d'un hypergraphe $\mathcal{H}$ ***recouvre*** une hyperarête e de $\mathcal{H}$ si tous les sommets de e sont contenus dans χ.

Un ensemble de blocs gardés Ξ est appelé ***recouvrement gardé*** (guarded cover) de $\mathcal{H}$ si chaque hyperarête de $\mathcal{H}$ est recouverte par un bloc gardé de Ξ.

Un ensemble de blocs gardés Ξ d'un hypergraphe $\mathcal{H}$ est appelé recouvrement gardé **complet** de $\mathcal{H}$ si toute hyperarête e de $\mathcal{H}$ apparaît dans le λ d'un bloc gardé recouvrant e.

Théorème 2.3.1. *[15] Un ensemble de blocs gardés Ξ d'un hypergraphe $\mathcal{H}$ est une décomposition gardée de $\mathcal{H}$ si et seulement si Ξ définit un recouvrement gardé complet de $\mathcal{H}$.*

Le join tree de l'ensemble des blocs gardés d'un hypergraphe $\mathcal{H}$ est un arbre T dont les nœuds (sommets) sont les blocs gardés de Ξ et l'ensemble des nœuds de T dont le bloc contient un sommet X_i induit un sous-arbre (connexe) de T.

Théorème 2.3.2. *[15] Si l'ensemble Ξ des blocs gardés est un recouvrement gardé acyclique de $\mathcal{H}$ alors l'ensemble $\Xi \cup \{(\{e\}, e) \mid e \in E\}$ est une décomposition gardée acyclique de $\mathcal{H}$.*

Définition 2.3.15. (**Recouvrement gardé étendu**)

Soit $\mathcal{H} = \langle V, E \rangle$ un hypergraphe. Un recouvrement gardé de $\langle V, E \cup \{X_i\} \mid X_i \in V \rangle$ est appelé

recouvrement gardé étendu de $\mathcal{H}$.

Définition 2.3.16. (**Recouvrement gardé défini par les hyperarêtes**)
Soit (λ, χ) un bloc gardé. Un recouvrement gardé Ξ d'un hypergraphe $\mathcal{H}$ est défini par les hyperarêtes si le χ de chaque bloc gardé de Ξ contient exactement l'ensemble des sommets couverts par les hyperarêtes de son λ.

La *largeur* d'un ensemble de blocs gardés est le nombre maximum d'hyperarêtes dans ses λ.
Ce formalisme de décompositions gardées permet de représenter beaucoup de méthodes structurelles [15].

2.3.10.2 La méthode SCD (Spread cuts)

Définition 2.3.17. (χ-**composante**)
Soient $\mathcal{H} = \langle V, E \rangle$ un hypergraphe et $\chi \subseteq V$ un ensemble de sommets. Deux hyperarêtes e et f de $\mathcal{H}$ sont dites χ-*adjacentes* si $e \cap f \not\subset \chi$.
Un χ-*chemin* reliant deux hyperarêtes e et f est une séquence d'hyperarêtes $e = e_0, \cdots, e_r = f$ telle que e_i est χ-adjacente à e_{i+1} pour tout $i = 0, \cdots, r-1$. Un ensemble d'hyperarêtes $E' \subset E$ est χ-*connexe* s'il existe un χ-chemin reliant toute paire d'hyperarêtes de E'. Un ensemble d'hyperarêtes $E' \subset E$ est une χ-*composante* d'hyperarêtes de $\mathcal{H}$ s'il est un sous-ensemble non vide, maximal et χ-connexe de E (il existe un χ-chemin reliant toute paire d'hyperarêtes de E'). Un ensemble non vide S' de sommets de $\mathcal{H}$ est une χ-*composante* de sommets de $\mathcal{H}$ s'il existe une hyperarête χ-composante e'_χ pour laquelle $S' = \bigcup e'_\chi - \chi$.

Définition 2.3.18. (**"unbroken components"**)
Un bloc gardé $\langle \lambda, \chi \rangle$ d'un hypergraphe $\mathcal{H}$ a des "unbroken components" si chaque χ- composante de $\mathcal{H}$ a une intersection non vide avec au plus une $\cup\lambda$-composante de $\mathcal{H}$ et $\{e_1 \cap e_2 / e_1, e_2 \in \lambda\} \subset \chi$.

Un **Spread Cut** de $\mathcal{H}$ est un recouvrement gardé acyclique dans lequel tous les blocs gardés ont des "unbroken components".

Forme canonique des blocs gardés avec des "unbroken components"

Définition 2.3.19. (**Label d'un sommet**)

Soit λ un sous-ensemble d'hyperarêtes de $\mathcal{H}$. Le **label** $L_\lambda(x)$ de n'importe quel sommet de $\mathcal{H}$ est un ensemble d'hyperarêtes $\cup\lambda$-composantes incluant une hyperarête contenant x.

Définition 2.3.20. (**Bloc gardé canonique**)

Un bloc gardé est dit **canonique** si pour chaque hyperarête $e \in \lambda$, les sommets de e qui n'appartiennent pas à χ sont exactement ceux qui ont un label particulier, à savoir $\forall x \in e - \chi, \forall y \in e, y \notin \chi, \longleftrightarrow L_\lambda(x) = L_\lambda(y)$.

Définition 2.3.21. (**Décomposition spread cut**)

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

Théorème 2.3.3. *[15] Si $\mathcal{H}$ a une décomposition spread cut de largeur k, alors $\mathcal{H}$ a une décomposition spread cut de largeur au plus k dans laquelle chaque bloc gardé est canonique.*

Il a été prouvé dans [15] qu'il existe une famille d'hypergraphes qui ont une largeur spread cut de 2n mais une largeur hypertree (en hyperarbre) de 3n. Cependant, nous ne savons pas est ce que la décomposition SCD est meilleure que la décomposition en hyperarbre.

Pour résoudre des CSPs avec SCD, on peut utiliser tous les algorithmes proposés pour la GHD.

2.3.10.3 La méthode SCD_{NEW} (Spread Cut_{NEW})

Cette méthode a été proposée dans [16] et qui est légèrement différente de la méthode proposée par les mêmes auteurs avec le même nom dans [15].

Définition 2.3.22. (**Label d'un sommet**)

Soit l'hypergraphe $\mathcal{H} = \langle V, E\rangle$. On définit le **label** $L_\lambda(x)$ d'un sommet $x \in \cup\lambda$ comme une paire de composantes où la première est un ensemble de $\cup\lambda$-composantes qui intersectent une hyperarête contenant x. La deuxième est un ensemble d'hyperarêtes de λ contenant x. En d'autres termes :

- $L_\lambda(x)[1] = \{C \mid \text{C est une } (\cup\lambda)\text{- composante}, \exists e \in E, e \cap C \neq \emptyset, x \in e\}$,
- $L_\lambda(x)[2] = \{e \in \lambda \mid x \in e\}$

Définition 2.3.23. On dit qu'un bloc gardé (λ, χ) respecte les labels si $\forall x, y \in (\cup\lambda), (x \in \chi$ et $L_\lambda(x) = L_\lambda(y)) \Rightarrow y \in \chi$.

Définition 2.3.24. (**Décomposition spread** cut_{NEW})

Une décomposition spread cut_{NEW} (SCD_{NEW}) est un recouvrement gardé acyclique où chaque nœud est un bloc gardé qui n'a pas de "unbroken components" et il respecte les labels.

Dans [16], une classe d'hypergraphes dont la largeur *spread* cut_{New} est inférieure à la largeur *hypertree (en hyperarbre)* est présentée, mais en général, nous ne savons pas est ce que la décomposition SCD_{New} est meilleure que la décomposition en hyperarbre.

Pour résoudre des CSPs avec SCD_{New}, on utilise les algorithmes proposés pour la GHD.

2.3.11 Subedge-based decompositions

Les méthodes définies sous-hyperarêtes (subedge defined) sont proposées par Miklos [65]. Elles sont basées sur le fait qu'ajouter des sous-hyperarêtes ne change pas le résultat du problème de départ.

Définition 2.3.25. (**sous-hyperarête**)

Un sous-ensemble de sommets d'une hyperarête e d'un hypergraphe $\mathcal{H}$ est appelé sous-hyperarête de $\mathcal{H}$.

Définition 2.3.26. (**Décomposition structurelle** [65])

Une méthode de décomposition structurelle M est une application qui associe un ensemble M($\mathcal{H}$) de GHDs à l'hypergraphe $\mathcal{H}$.

Proposition 2.3.4. *[65] Soit $\mathcal{H}$ un hypergraphe et soit $D = \langle T, \chi, \lambda\rangle$ une GHD de $\mathcal{H}$. Alors $D' = \langle T, \chi, \lambda'\rangle$ où $\lambda'(p) = \{e \cap \chi(p) \mid e \in \lambda(p)\}$ pour chaque nœud p de T est une HD de $\mathcal{H} \cup \{e \cap \chi(p) \mid p \in T, e \in \lambda(p)\}$. De plus, la largeur de D' est inférieure ou égale à la largeur de D.*

La proposition 2.3.4 explique la relation entre les décompositions HD et GHD.

Définition 2.3.27. On dit qu'un algorithme A implémente une méthode M si pour toute constante k et pour chaque hypergraphe $\mathcal{H}$, A retourne une GHD dans $M(\mathcal{H})$ de largeur au plus égale à k si elle existe, sinon il retourne échec.

Chaque fonction f liant un hypergraphe $\mathcal{H}$ à un ensemble de sous-hyperarêtes de $\mathcal{H}$ définit une méthode de décomposition qui peut être calculée comme suit :

1. Calculer $f(\mathcal{H})$
2. Calculer une HD minimale de $\mathcal{H} \cup f(\mathcal{H})$.

Comme l'étape 2 est faisable uniquement pour une constante k, alors f doit aussi dépendre de k. Donc une fonction subedge (sous-hyperarête) associe un ensemble de sous-hyperarêtes à la paire ($\mathcal{H}$, k). On suppose aussi que les fonctions subedge sont monotones.

Définition 2.3.28. (**Fonction subedge**)

Une fonction subedge f est une fonction qui associe un ensemble de sous-hyperarêtes à chaque paire ($\mathcal{H}$, k) et pour chaque $i < j$, $f(\mathcal{H}, i) \subseteq f(\mathcal{H}, j)$.

Définition 2.3.29. Soient $D = \langle T, \chi, \lambda \rangle$ une HD de $\mathcal{H} \cup f(\mathcal{H}, k)$ et $D' = \langle T, \chi, \lambda' \rangle$ une GHD de $\mathcal{H}$. On dit que D' *recouvre* D si, pour chaque nœud p de T et pour chaque $e \in \lambda(p)$, il existe $e' \in \lambda'(p)$ telle que e est une sous-hyperarête de e', i.e. $vertices(e) \subseteq vertices(e')$.

Si on ajoute des sous-hyperarêtes à un hypergraphe $\mathcal{H}$, alors la largeur hypertree (en hyperarbre) de l'hypergraphe résultant $\mathcal{H}'$ est au plus celle de $\mathcal{H}$, i.e. $htw(\mathcal{H}') \leq htw(H)$. Donc pour un hypergraphe $\mathcal{H}$ et une fonction subedge f, la largeur hypertree de $\mathcal{H} \cup f(H, i)$ décroît de façon monotone (en i). L'hypertree-width de $\mathcal{H} \cup f(H, i)$ dépend de la structure de $\mathcal{H}$ et de la fonction subedge f.

L'idée est donc de définir des méthodes de décomposition en utilisant les fonctions subedge en liant la GHD de $\mathcal{H}$ à l'HD de $\mathcal{H} \cup f(H, i)$ pour une certaine valeur de i.

Définition 2.3.30. (**Méthode de décomposition**)

Soit $\mathcal{H}$ un hypergraphe et soit $f(\mathcal{H}$, k) une fonction subedge. On définit une méthode de décomposition $M_f(\mathcal{H})$ comme suit. $M_f(\mathcal{H})$ est l'ensemble des GHDs D' de $\mathcal{H}$ pour lesquelles il existe k tel que $k \leq |D'|$ ($|D'|$ désigne la largeur de D') et il existe une décomposition en hyperarbre D de l'hypergraphe $\mathcal{H} \cup f(H, k)$ telle que D' recouvre D. La méthode de décomposition de la forme M_f est dite basée sous-hyperarête (subedge-based).

Dans ce qui suit, nous allons présenter trois méthodes basées subedge.

2.3.11.1 La méthode CHD (Component Hypertree)

Cette méthode proposée par Miklos [65] est une "subedge defined method". On note par vertices(edges(C)) l'ensemble des variables des hyperarêtes ayant une intersection non vide avec la composante C.

Définition 2.3.31. Soit M un ensemble d'hyperarêtes d'un hypergraphe $\mathcal{H} = \langle V, E \rangle$. On définit "$\mathbf{prop}(e, M) = e \setminus \cup_{e' \in M, e \neq e'} e'$. C'est la part de l'hyperarête e à M toute seule.

Définition 2.3.32. Soit M un ensemble d'hyperarêtes de $\mathcal{H}$ et soit e une hyperarête dans M. ***internal(e,M)***=$\{x \mid x \in vertices(e)$ et il n'existe pas de $[vertices(M)] - composante$ C de $\mathcal{H}$ tel que $x \in vertices(edges(C))\}$. C'est l'ensemble des variables x de e qui n'appartiennent à aucune [var (M)]-composante ayant une hyperarête contenant la variable x.

Définition 2.3.33. Soient $\mathcal{H}$ un hypergraphe, M un ensemble d'hyperarêtes de $\mathcal{H}$, e une hyperarête de M et C une [vertices(M)]-composante. La fonction "***elim*** (M, C, e) associe 3 sous-hyperarêtes au triplet (M,C,e) :

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

On définit maintenant la fonction subedge comme suit :

Définition 2.3.34. Soit $\mathcal{H}$ un hypergraphe et soit k $(0 \leq k)$ un entier. La *fonction subedge* est définie comme suit : $f^C(H, k)$=$\{e \setminus e' \mid M$ est un ensemble de $\leq k$ hyperarêtes de H, $e \in M$, D est une [vertices(M)]-composante et $e' \in elim(M, D, e)$ $\}$

La méthode M_{f^C} est appelée Décomposition Component Hypertree (CHD).

Pour calculer une CHD, on calcule une décomposition en hyperarbre de l'hypergraphe $\mathcal{H}' = H \bigcup f^C(\mathcal{H}, k)$. La largeur de $\mathcal{H}'$ sera nécessairement inférieure à l'hypertree width de $\mathcal{H}$ [65].

De cette façon, on trouve une largeur inférieure à la largeur hypertree optimale. Mais qui n'est

toujours pas une largeur en hyperarbre généralisée optimale. Il a été démontré dans [65] que la largeur de l'hypergraphe retournée par la méthode CHD est toujours inférieure ou égale à celles retournées par les décompositions en hyperarbre HD et spread cut SCD. De plus décider si pour une constante k, un hypergraphe $\mathcal{H}$ a une décomposition component hypertree de largeur au plus égale à k est faisable en un temps polynomial.

Exemple 2.3.4. ***décomposition par CHD***

Soit l'exemple [65] suivant : $C1 = \langle (X1, X2), R1 \rangle$, $C2 = \langle (X2, X3, X9), R2 \rangle$,
$C3 = \langle (X3, X4, X10), R3 \rangle$ $C4 = \langle (X4, X5), R4 \rangle$, $C5 = \langle (X5, X6, X9), R5 \rangle$,
$C6 = \langle (X6, X7, X10), R6 \rangle$, $C7 = \langle (X7, X8, X9), R7 \rangle$, $C8 = \langle (X1, X8, X10), R8 \rangle$.

Sa décomposition CHD est donnée par la figure 2.8 où C'2 et C'3 sont les hyperarêtes obtenues par la fonction ***subedge*** *: C'2(X3,X9) et C'3(X3,X10).*

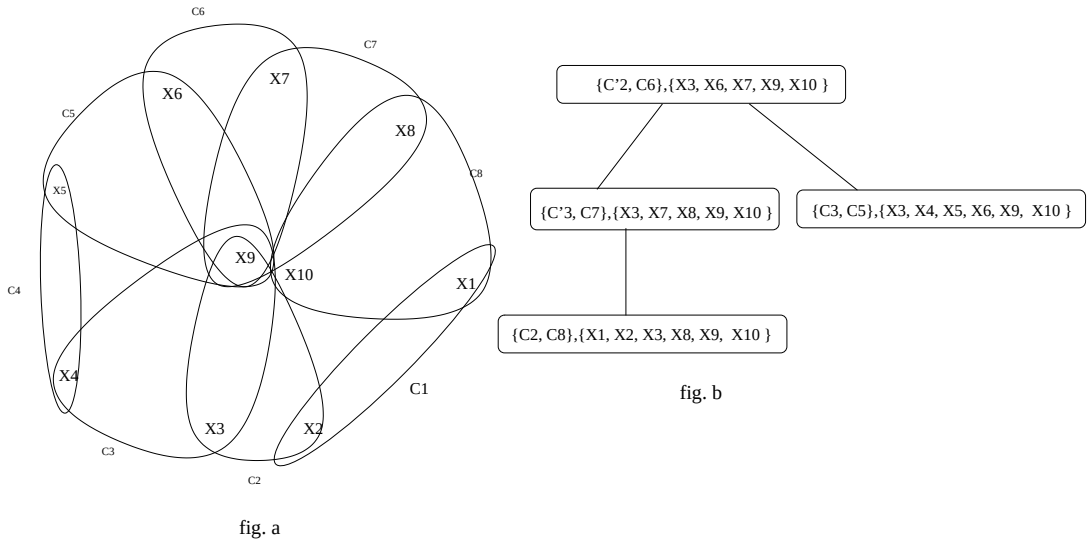


FIG. *2.8 – Exemple de décomposition CHD. fig.a est un hypergraphe et fig.b est une décomposition CHD.*

Cette méthode est meilleure que les décompositions en hyperarbre et Spread cut SCD [65].

Pour la résolution des CSPs via *CHD*, toutes les méthodes proposées pour les GHD sont applicables.

2.3.11.2 La méthode ECHD (Extend Component Hypertree)

Cette méthode, proposée dans [65], est une généralisation de la CHD.

Définition 2.3.35. Soit M un ensemble d'hyperarêtes d'un hypergraphe $\mathcal{H}$. Soit N un sous-ensemble d'hyperarêtes de M, $N \subseteq M$. On définit $Slice(N, M)$ comme suit :

$Slice(N, M) = \{x \mid \forall e \in N, x \in vertices(e)$ et $\forall e' \in (M \setminus N), x \notin vertices(e')\}$.

$Slice(N, M)$ est un ensemble de sommets de vertices(M) qui sont adjacents aux hyperarêtes de N mais pas aux autres hyperarêtes dans M. Il est clair que vertices(M) $= \bigcup_{N \subseteq M} Slice(N, M)$.

Définition 2.3.36. Soit M un ensemble d'hyperarêtes d'un hypergraphe $\mathcal{H}$ et N un sous-ensemble d'hyperarêtes de M. On définit $Internal(N, M)$ comme suit :

$Internal(N, M) = \{x \mid x \in slice(N, M)$ et il n'existe pas de $[vertices(M)] - composante$ C tel que $x \in vertices(edges(C))\}$

Définition 2.3.37. Soit $\mathcal{H}$ un hypergraphe. Soit M un ensemble d'hyperarêtes de $\mathcal{H}$ et soit C une [vertices(M)]-composante. Soient $N_1, \cdots, N_r$ $(1 \leq r \leq 2^{|M|-1})$ des sous-ensembles de M.

La fonction $\boldsymbol{elim}(M, C, N_1, \cdots, N_r)$ associe un ensemble contenant les sous-hyperarêtes suivantes au tuple $(M, C, N_1, \cdots, N_r)$.

1. $\bigcup_{1 \leq i \leq r} (slice(N_i, M) \bigcap vertices(edges(C)))$
2. $\bigcup_{1 \leq i \leq r}$ internal(N_i, M)

Définition 2.3.38. Soit $\mathcal{H}$ un hypergraphe et soit k $(0 \leq k)$ un entier. La fonction subedge f^E est définie comme suit :

$f^E(H, k) = \{e \setminus e' \mid$ M est un ensemble de $\leq k$ hyperarêtes de l'hypergraphe $\mathcal{H}$,

$e \in M$,

C est une [vertices (M)]-composante,

$1 \leq r \leq 2^{k-1}$,

$N_1, \cdots, N_r$ sont des ensembles d'hyperarêtes,

Pour chaque i $(1 \leq i \leq r), Ni \subseteq M$,

$e' \in elim(M, C, N_1, \cdots, N_r)\}$

}

M_{f^E} est la décomposition Extended Component Hypertree (ECHD). Il a été démontré dans [65] que les largeurs retournées par ECHD sont toujours inférieures ou égales à celles retournées par CHD et SCD_{New} ($ECHD \leq CHD$ et $ECHD \leq SCD_{New}$).

Pour la résolution des CSPs via ECHD, toutes les méthodes proposées pour les GHD sont applicables.

2.3.11.3 La méthode ECHD[d] (Parameterized Extend Component Hypertree)

Une version paramétrée de la méthode ECHD a été présentée dans [65]. Dans cette version, la fonction elim définie précédemment est redéfinie comme suit :

Définition 2.3.39. Soit $\mathcal{H}$ un hypergraphe, soit M un ensemble d'hyperarêtes de $\mathcal{H}$. Soient $N_1, \cdots, N_r$ ($1 \leq r \leq 2^{M-1}$) des sous-ensembles de M. Aussi considérons une constante d ($1 \leq d \leq |E(\mathcal{H})|$) et soient $C_1, \cdots, C_d$ des [vertices(M)]-composantes.
La fonction $elim(M, C_1, \cdots, C_d, N_1, \cdots, N_r)$ associe un ensemble de sous-hyperarêtes au tuple $(M, C_1, \cdots, C_d, N_1, \cdots, N_r)$:

1. $\bigcup_{1 \leq i \leq r} Slice(Ni, M) \cap \bigcup_{1 \leq j \leq d} vertices(edges(Cj))$
2. $\bigcup_{1 \leq i \leq r} internal(N_i, M)$.

Définition 2.3.40. Soit $\mathcal{H}$ un hypergraphe et soit k ($0 \leq k$) un entier. La fonction subedge paramétrée $f^E[d]$ est définie comme suit :

$f^E(H, k) = \{e \setminus (f0 \cup f1) \mid$ M est un ensemble de $\leq k$ hyperarêtes de l'hypergraphe $\mathcal{H}$,
$e \in M$,
$C_1, \cdots, C_d$ sont des [vertices (M)]-composantes
$N_1, \cdots, N_r$, $1 \leq r \leq 2^{k-1}$, sont des ensembles d'hyperarêtes.
Pour chaque i ($1 \leq i \leq r$), $Ni \subseteq M$,
$f0 \in elim(M, C_1, \cdots, C_d, N_1, \cdots, N_r)$
$f1 \in elim(M, C_1, \cdots, C_d, N_1, \cdots, N_r)$
$\}$

$M_{f^E}[d]$ est la décomposition Extended Component Hypertree paramétrée (ECHD[d]).

Remarque 2.3.3. ECHD est équivalente à ECHD(1)

Cette méthode est la plus générale et la plus pertinente parmi toutes les méthodes ***traitables***. Aussi, un point important à signaler est le fait que la découverte d'une décomposition par cette méthode est traitable en temps polynomial pour une constante k donnée. Seulement, elle n'est pas encore au même niveau que la méthode en hyperabre généralisée.

Pour la résolution des CSPs via *ECHD[d]*, toutes les méthodes proposées pour les GHD sont applicables.

2.3.12 La méthode FHD (Fractional Hypertree Decomposition)

Cette méthode a été proposée dans [45].

Définition 2.3.41. (**fractional edge cover**)

Le ***fractional edge cover*** d'un hypergraphe $\mathcal{H} = \langle V, E\rangle$ est une application $\psi : E \to [0, \infty]$ telle que $\Sigma_{e\in E(\mathcal{H}),x\in e}\psi(e) \geq 1$ pour tout $x \in V$.

Le nombre $\sum_{e\in E} \psi(e)$ est appelé le ***poids*** de ψ.

Le nombre ***fractional edge cover*** ρ^* de $\mathcal{H}$ est le poids minimum de tous les fractional edge cover de l'hypergraphe $\mathcal{H}$.

Définition 2.3.42. Pour un hypergraphe $\mathcal{H}$ et une application $\gamma : E \to [0, \cdots, \infty]$, on pose : $B(\gamma) = \{x \in V \mid \sum_{e\in edges(\mathcal{H}),x\in e} \gamma(e) \geq 1\}$

Définition 2.3.43. (Décomposition FHD (fractional hypertree decomposition))

La décomposition en hyperarbre fractionnelle (fractional hypertree decomposition) FHD d'un hypergraphe $\mathcal{H}$ est un triplet $\langle T, (B_t), (\gamma_t)_{t\in vertices(T)}\rangle$ où $\langle T, (B_t)\rangle$ est une décomposition arborescente de $\mathcal{H}$, et $(\gamma_t)_{t\in vertices(T)}$ est une famille d'applications de E vers $[0, \cdots, \infty]$ telle que pour chaque $t \in vertices(T)$, on a $B_t \subseteq B(\gamma_t)$.

La largeur de $\langle T, (Bt), (\gamma_t)_{t\in vertices(T)}\rangle$ est max $\{poids(\gamma_t) \mid t \in vertices(T)\}$ et la largeur de la FHD est la largeur minimale des largeurs de toutes ses décompositions FHD.

Il a été prouvé dans [45] que cette méthode généralise la méthode GHD et par conséquent, elle généralise toutes les autres méthodes.

Il existe d'autres méthodes de décomposition structurelles, sans être exhaustif, on peut citer Pseud-arbre [27], Cyclic clustering [57], Recursive Conditioning [18], etc.

2.4 Classification des méthodes de décomposition

Dans cette section, nous présentons une synthèse des hiérarchies proposées dans [40] et dans [54]. Nous exploitons aussi les résultats donnés dans [45] et [65]. Ceci pour les critères élaborés par Gottlob et al. [40].
La hiérarchie des méthodes de décomposition structurelles est résumée dans la figure 2.9 où une flèche d'un nœud A vers un nœud B veut dire que la méthode B est plus générale que

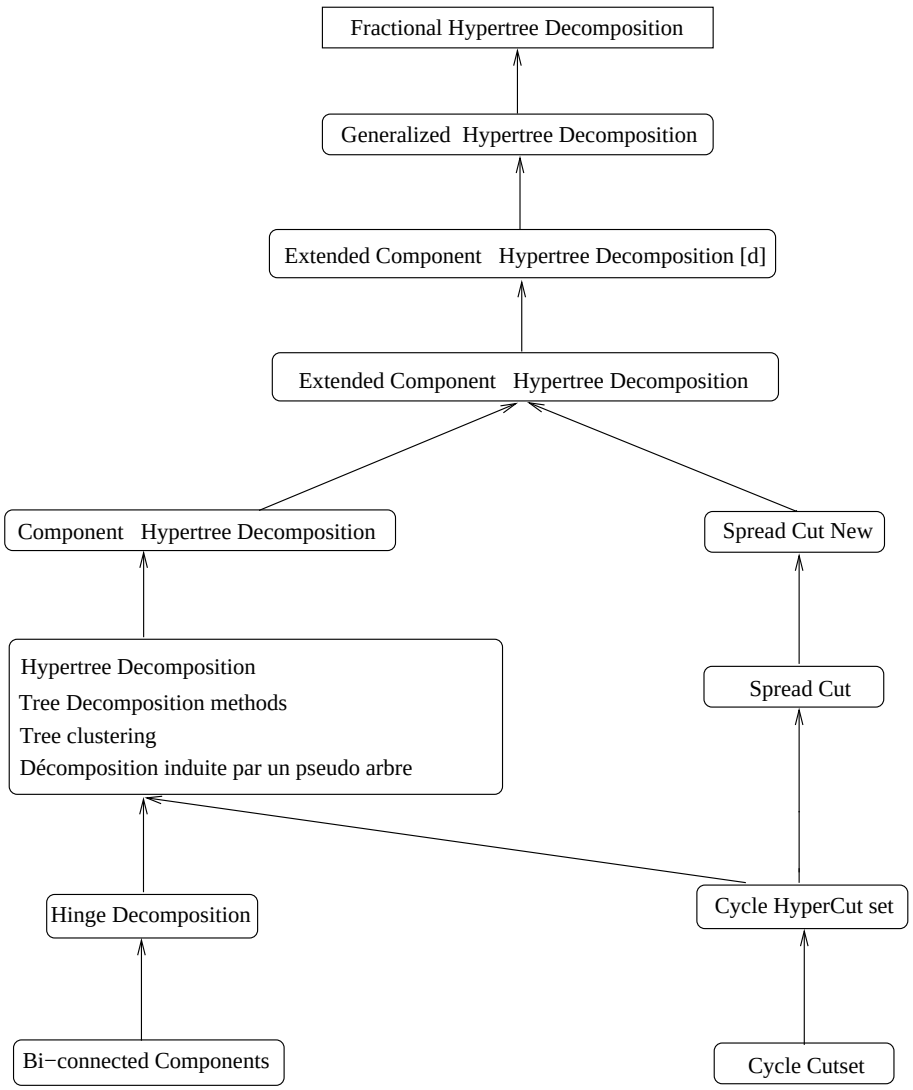


FIG. 2.9 – Hiérarchie des méthodes de décomposition structurelles

A. Dans cette hiérarchie, la méthode la plus générale est la décomposition FHD. La deuxième qui vient juste en dessous est la décomposition GHD. Pour celle-ci, il a été démontré dans [42] que le problème de reconnaissance si un hypergraphe $\mathcal{H}$ admet une décomposition de largeur inférieure ou égale à une constante k est un problème NP-complet. Par contre, pour les autres méthodes dans la hiérarchie, la méthode de décomposition traitable la plus générale est ***ECHD paramétrée***.

On remarque aussi une autre différence importante par rapport à la hiérarchie donnée dans [40]. En effet, dans cette classification, la méthode de tree clustering [24] est au même niveau que la décomposition en hyperabre. Ce résultat a été démontré dans [54].

2.5 Conclusion

Dans ce chapitre, nous avons présenté les principales méthodes de décomposition structurelles. Chaque méthode définit son propre concept de largeur w qui peut être interprétée comme une mesure de cyclicité du graphe ou de l'hypergraphe de contraintes associé à l'instance CSP. Le gain théorique de ces méthodes est indéniable du fait que la complexité de la résolution est bornée par la largeur de la décomposition. Les méthodes les plus générales sont la décomposition FHD et la décomposition GHD. La première étape de toute méthode de résolution de CSP basée décomposition consiste d'abord à calculer une bonne décomposition (car en général, le calcul d'une décomposition optimale est très coûteux aussi bien en temps qu'en espace) avant de résoudre le problème global. Dans le chapitre suivant, nous allons présenter notre première contribution pour le calcul d'une GHD d'un hypergraphe.

Bibliographie

[1] I. Adler, G. Gottlob, and M. Grohe, *Hypertree-width and related hypergraph invariants*, the 3rd European Conference on Combinatorics, Graph Theory, and Applications (EUROCOMB'05), DMTCS Proceedings Series, vol AE, 2005, pp. 5–10.

[2] M. Aït-Amokhtar, K. Amroun, and Z. Habbas, *Hypertree decomposition for solving constraint satisfaction problems*, Proceedings of International conference on Agents and Artificial Intelligence, ICAART 2009 (Portugal), 2009, pp. 398–404.

[3] J.F. Baget and Y. Tognetti, *Backtracking throught biconnected components of a constraint graph*, Proceedings of IJCAI 2001, 2001, pp. 291–296.

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

[5] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, *On the desirability of acyclic database schemes*, Journal of the Association for Computing Machinery **30** (1983), 479–513.

[6] P. Berlandier, *Improving domain filtering using restricted path consistency*, Proceedings of IEEE CAIA 95, 1995.

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

[8] C. Bessière, A. Chmeiss, and L. Sais, *Neighborhood-based variable ordering heuristics for the constraint satisfaction problem*, Proceedings of CP'01, 2001, pp. 565–569.

[9] C. Bessière, P. Meseguer, E.C Freuder, and J. Larrosa, *On forward checking for non-binary constraint satisfaction*, Artificial Intelligence **141** (2002), 205–224.

[10] C. Bessière and J. Régin, *Refining the basic constraint propagation algorithm*, Proceeding of IJCAI, 2001.

[11] C. Bessière and J. C. Régin, *MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems*, Proceedings of of CP'96, 1996, pp. 61–75.

[12] C. Bessière, J. C. Régin, R. H.C. Yap, and Y. Zhang, *An optimal coarse-grained arc consistency algorithm*, Artificial Intelligence **165** (2005), no. 2, 165–185.

[13] C. Chekuri and A. Rajaraman, *Conjunctive query containment revisited*, Journal of Theoretical Computer Science **239** (2000), no. 2, 211–229.

[14] V. Chvatal, *A greedy heuristic for the set covering problem*, Math. of operations Research (1979), 233–235.

[15] D. Cohen, P. Jeavons, and M. Gyssens, *A unified theory of structural tractability for constraint satisfaction and spread cut decomposition*, Proceedings of IJCAI'05, 2005.

[16] D. Cohen, P. Jeavons, and M. Gyssens, *A unified theory of structural tractability for constraint satisfaction problems*, Journal of Computer and System Sciences 74 pp 721-743 (2008).

[17] M. Cooper, *An optimal k-consistency algorithm*, Artificial Intelligence **41** (1989).

[18] A. Darwiche, *Recursive conditioning*, Artificial Intelligence **126** (2001), 5–41.

[19] R. Debruyne and C. Bessière, *From restricted path consistency to max-restricted path consistency*, Proceedings of CP (CP-97), 1997.

[20] R. Dechter, *Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition*, Artificial Intelligence **41** (1990), 273–312.

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

[22] R. Dechter and J. Pearl, *The cycle-cutset method for improving search performance in AI applications*, Proceedings of the third IEEE on Artificial Intelligence Applications (Orlando), 1987, pp. 224–230.

[23] ———, *Tree-clustering schemes for constraint-processing*, Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-88) (Saint Paul, MN), 1988, pp. 150–154.

[24] ———, *Tree clustering for constraint networks*, Artificial Intelligence **38** (1989), 353–366.

[25] R. Dechter and P. van Beek, *Local and global relational consistency*, Theoretical Computer Sciences **173** (1997), 283–308.

[26] A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, and M. Samer, *Heuristic methods for hypertree decompositions*, Proceedings of the 7th. Mexican Int. Conf. on Artificial Intelligence : Advances in Artificial Intelligence (Saint Paul, MN), 2008, pp. 1–11.

[27] E. Freuder and M. Quinn, *Taking advantage of stable sets of variables in constraint satisfaction problems*, Proceedings of IJCAI, 1985, pp. 1076–1078.

[28] E. C. Freuder, *Synthesizing constraint expressions*, Communication of the Association for Computing Machinery **21** (1978), 958–966.

[29] ———, *A sufficient condition for backtrack-free search*, Journal of the Association for Computing Machinery **29** (1982), 24–32.

[30] ———, *A sufficient condition for backtrack-bounded search*, Journal of the Association for Computing Machinery **32** (1985), 755–761.

[31] E.C Freuder and C.D. Elfe, *Neighborhood inverse consistency preprocessing*, Proceedings of AAAI-96 (Portland, Oregon), 1996, pp. 202–208.

[32] D. Frost and R. Dechter, *Look-ahead value ordering for constraint satisfaction problems*, Proceedings of the fourteenth International Joint Conference on Artificial Intelligence (Montreal), 1995, pp. 572–578.

[33] T. Ganzow, G. Gottlob, N. Musliu, and M. Samer, *A CSP hypergraph library*, Tech. report, DBAI-TR-2005-50, Technische Universität Wien, 2005.

[34] M.R. Garey and D.S. Johnson, *Computer and intractability*, Freeman, 1979.

[35] J. Gaschnig, *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis, Carnegie Mellon University, 1979.

[36] I.P. Gent, C. Jefferson, and P. Nightingale, *Data structures for generalized arc consistency for extensional constraints*, Proceedings of AAAI'07, 2007, pp. 191–197.

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

[38] S. Golumb and L. Baumert, *Backtrack programming*, ACM (1965), 233–235.

[39] G. Gottlob, N. Leone, and F. Scarcello, *On tractable queries and constraints.*, Proceedings of DEXA'99, 1999.

[40] ———, *A comparison of structural CSP decomposition methods*, Artificial Intelligence **124** (2000), 243–282.

[41] ———, *Robbers, marshals and guards : Game theoretic and logical characterizations of hypertree width*, Journal of Computer and System Sciences **66** (2003), 775–808.

[42] G. Gottlob, Z. Miklos, and T. Schwentick, *Generalized hypertree decomposition : NP-hardness and tractable variants.*, Journal of the ACM **56** (2009).

[43] G. Gottlob and M. Samer, *A backtraching based algorithm for computing hypertree decompositions*, ACM Journal of Experimental Algorithmics (JEA) **13** (2009).

[44] M.H. Graham, *On the universal relation*, Tech. report, Univ. of Toronto, Canada, 1979.

[45] M. Grohe and D. Marx, *Constraint solving via fractional edge covers*, Proceedings of SODA 2006, 2006, pp. 289–298.

[46] M. Gyssens, P. G. Jeavons, and D. A. Cohen, *Decomposing constraint satisfaction problems using database techniques*, Artificial Intelligence **66** (1994), 57–89.

[47] Z. Habbas, K. Amroun, and D. Singer, *A forward checking algorithm based on a generalized hypertree decomposition for solving non-binary constraint satisfaction problems*, Journal of Experimental and Theoritical Artificial Intelligence JETAI, to appear.

[48] ———, *A cluster forward checking algorithm based on generalized hypertree decomposition*, RCRA 2013 International Workshop (Roma, Italy), 2013.

[49] ———, *A compression algorithm for solving efficiently non binary CSP using generalized hypertree decomposition*, Colloque COSI 2013 (CDTA, Alger, Algérie), 2013.

[50] C.-C. Han and C.-H. Lee, *Comment on Mohr and Henderson's path consistency algorithm*, Artificial Intelligence **36** (1988), 125–130.

[51] M. Haralick and G.L. Elliot, *Increasing tree-search efficiency for constraint satisfaction problems*, Artificial Intelligence **14** (1980), 263–313.

[52] P. Harvey and A. Ghose, *Reducing redundancy in the hypertree decomposition scheme*, Proceeding of ICTAI'03 (Montreal), 2003, pp. 474–481.

[53] L. Hyafi and R.L. Rivest, *Constructing optimal decision binary decision trees is NP-complete*, Information processing letters **5** (1976), 15–17.

[54] P. Jégou, S. N. Ndiaye, and C. Terrioux, *Complexité du forward checking et hiérarchie des décompositions revisitées*, Actes JFPC 2008), 2008.

[55] ———, *Combined strategies for decomposition-based methods for solving CSPs*, Proceedings of the 21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2009), 2009, pp. 184–192.

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

[57] P. Jégou and C. Terrioux, *A time-space trade-off for constraint networks decomposition*, Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 2004, pp. 234–239.

[58] G. Katsirelos and T. Walsh, *A compression algorithm for large arity extentional constraints*, Proceedings of CP'07, 2007, pp. 379–393.

[59] C. K. Kenil and R.H.C. Yap, *An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints*, Constraints **15** (2010), 265–304.

[60] T. Korimort, *Heuristic hypertree decomposition*, AURORA TR 2003-18 (2003).

[61] ———, *Heuristic hypertree decomposition*, Ph.D. thesis, Vienna University of Technology, April 2003.

[62] C. Lecoutre, *STR2 : Optimized simple table reduction for table constraints*, Constraints **16** (2011), 341–371.

[63] A. K. Mackworth, *Consistency in networks of relations*, Artificial Intelligence **8** (1977), 99–118.

[64] F. Mazoit, *Décomposition algorithmique des graphes*, Ph.D. thesis, École normale Supérieure de Lyon, April 2004.

[65] Z. Miklos, *Understanding tractable decompositions for constraint satisfaction*, Ph.D. thesis, University of Oxford, 2008.

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

[67] U. Montanari, *Networks of constraints : Fundamental properties and applications to pictures processing*, Information Sciences **7** (1974), 95–132.

[68] N. Musliu and W. Schafhauser, *Genetic algorithms for generalized hypertree decompositions*, European Journal of Industrial Engineering **1** (2005), no. 3, 317–340.

[69] W. Pang, *Constraint structure in constraint satisfaction problems*, Ph.D. thesis, University of Regina, Canada, 1998.

[70] W. Pang and S. D. Goodwin, *A revised sufficient condition for backtrack-free search*, In Proc of 10th Florida AI Research Symposium.

[71] ———, *A graph based backtracking algorithm for solving general CSPs*, Lecture Notes in Computer Sciences of AI 2003 (Halifax), 2003, pp. 114–128.

[72] G. Pesant, *A regular language membership constraint for finite sequences of variables*, Proceedings of CP'04, 2004, pp. 482–495.

[73] P. Prosser, *Hybrid algorithms for the constraint satisfaction problem*, Computational Intelligence **9** (1993), 268–299.

[74] N. Robertson and P.D Seymour, *Graph minors .II. algorithmic aspects of treewidth*, Journal of algorithms 7(3) pp 309-322 (1986).

[75] F. Rossi, C. Petrie, and V. Dhar, *On the equivalence of constraint satisfaction problems*, Proceedings of ECAI'90, 1990, pp. 550–556.

[76] D. Sabin and E. C. Freuder, *Contradicting conventional wisdom in constraint satisfaction*, Proceedings of the eleventh European Conference on Artificial Intelligence (Amsterdam), 1994, pp. 125–129.

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

[78] K. Stergiou and T. Walsh, *Encodings of non-binary constraint satisfaction problems*, Proceedings of AAAI'99, 1999, pp. 163–168.

[79] S. Subbarayan and H. Reif Anderson, *Backtracking procedures for hypertree, hyperspread and connected hypertree decomposition of CSPs*, IJCAI07 (2007), 180–185.

[80] R. Tarjan and M. Yannakakis, *Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs*, SIAM Journal on Computing **13** (1984), no. 3, 566–579.

[81] E. Tsang, *Foundations of constraints satisfaction*, Academic Press, London, 1993.

[82] J.R. Ullmann, *Partition search for non binary constraint satisfaction*, Information Science Journal **177** (2007), 3639–3678.

[83] P. van Hentenryck, Y. Deville, and C.-M. Teng, *A generic arc-consistency algorithm and its specializations*, Artificial intelligence **57** (1992), 291–321.

[84] ———, *Neighborhood-based variable ordering heuristics for the constraint satisfaction problem*, Proceedings CP, 1995, pp. 565–569.

[85] M. Yannakakis, *Algorithms for acyclic database schemes*, Proceedings of VLDB'81, 1981, pp. 82–92.