

ACADÉMIE DE MONTPELLIER  
UNIVERSITÉ MONTPELLIER II  
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

# Mémoire de Stage de Master

SPÉCIALITÉ : Recherche en Informatique  
*Mention* : Informatique, Mathématiques, Statistiques

effectué au laboratoire LIRMM/INFO  
—  
sous la direction de FRÉDÉRIC KORICHE

## Comptage de solutions dans un CSP

par

**Sébastien Andary**

Soutenu le le 20 Juin 2006

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Le problème #CSP</b>	<b>3</b>
2.1	Définitions préliminaires . . . . .	3
2.2	Énoncé du problème #CSP . . . . .	4
2.3	État de l'art . . . . .	5
<b>3</b>	<b>L'algorithme</b>	<b>7</b>
3.1	Structure arborescente . . . . .	7
3.1.1	Comptage des solutions dans un arbre . . . . .	7
3.1.2	Généralisation aux $k$ -arbres . . . . .	9
3.2	L'algorithme kTreeCount . . . . .	12
3.2.1	Structure de données . . . . .	13
3.2.2	Description de l'algorithme . . . . .	14
3.2.3	Correction . . . . .	17
3.2.4	Complexité . . . . .	19
3.3	Améliorations . . . . .	20
3.3.1	Finalisation des tables . . . . .	20
3.3.2	Détection de contradictions . . . . .	20
3.3.3	Occupation de la mémoire . . . . .	21
<b>4</b>	<b>Expérimentations</b>	<b>22</b>
4.1	Protocole d'expérimentation . . . . .	22
4.1.1	Graphes aléatoires . . . . .	22
4.1.2	$k$ -arbres . . . . .	22
4.2	Résultats . . . . .	23
4.2.1	Instances aléatoires . . . . .	23
4.2.2	Instances artificiellement structurées . . . . .	25
4.3	Instances Structurées . . . . .	26
4.4	Discussion . . . . .	27
<b>5</b>	<b>Conclusion et perspectives</b>	<b>29</b>

## Résumé

Ce mémoire présente un algorithme de comptage du nombre de solutions d'un réseau de contraintes qui tire parti de la structure du réseau. Bien que le problème soit  $\#P$ -complet dans le cas général, l'algorithme présenté a une complexité polynomiale en la taille du réseau si la *treewidth* du graphe de contraintes est bornée par une constante. Notre algorithme peut s'étendre à d'autres problèmes de graphes de  $\#P$  vérifiant certaines propriétés. Deux cadres d'utilisations sont possibles : calcul non incrémental en une passe, , avec quantité de mémoire limitée, ou calcul incrémental avec interaction avec l'utilisateur, nécessitant cependant plus de ressources.

# 1 Introduction

La programmation par satisfaction de contraintes est un outil puissant de modélisation et de résolution de problèmes. Cette approche de modélisation a fait ses preuves dans les domaines de planification, configuration, allocation de ressources et d'autres. Un réseau de contraintes consiste en un ensemble de variables, leurs domaines de valeurs et un ensemble de contraintes entre ces variables. Ce formalisme conceptuellement simple permet de modéliser de nombreux problèmes de manière intuitive. Une des tâches fondamentales des réseaux de contraintes est de trouver une solution au problème modélisé, c'est-à-dire une affectation de valeurs aux variables du réseau qui satisfasse l'ensemble des contraintes. Ce problème est appelé le problème de satisfaction de contraintes (CSP) et fait partie de la classe NP (réduction directe du problème SAT). Cependant, de nombreux algorithmes de recherche ont été développés, permettant aux *solveurs*, programmes dédiés à la résolution de CSP, de résoudre des problèmes de plus en plus complexes.

La problématique de ce stage s'inscrit dans le cadre, plus général, du comptage des solutions d'un CSP ( $\#CSP$ ). Le problème  $\#CSP$  trouve des applications dans des domaines divers comme l'apprentissage, le raisonnement probabiliste ou la modélisation interactive. Un utilisateur peut, par exemple, ajouter des contraintes progressivement à un réseau tout en contrôlant le nombre de solutions à chaque étape. Ou encore, dans le raisonnement probabiliste, on peut calculer le pourcentage de similitude entre deux réseaux (aussi appelé degrés de croyance). Aussi, il est souvent utile, dans un cadre d'apprentissage, de pouvoir compter le nombre de modèles d'une théorie qui peut être modélisée comme un réseau de contraintes. A ce jour, très peu de solveurs sont capables de compter le nombre de solutions d'un réseau de contraintes en un temps raisonnable.

Ce mémoire présente un algorithme de comptage du nombre de solutions d'un réseau de contraintes qui tire parti de la structure du réseau. La complexité est polynomiale en la taille du réseau si le graphe de contraintes a une treewidth bornée par une constante. La treewidth est une mesure qui capture la *ressemblance* d'un graphe à un arbre (une treewidth de 1 correspond à un arbre).

Beaucoup de problèmes  $NP$ -durs deviennent polynomiaux ou linéaires lorsque la *treewidth* est bornée par une constante. L'algorithme proposé peut s'étendre à d'autres problèmes de la classe  $\#P$  des problèmes de comptage associés aux problèmes de  $NP$ , si toutefois le problème peut être exprimé sous la forme d'un problème de graphes et vérifiant certaines propriétés. L'élaboration d'un nouvel algorithme se réduit alors à la formalisation de la notion de solution partielle et de la définition de quelques méthodes élémentaires.

Après avoir défini la problématique et établi un état de l'art dans le Chapitre 2, le Chapitre 3 présente l'algorithme *kTreeCount* en détails. Nos résultats d'expérimentation sont interprétés dans le Chapitre 4. Enfin le Chapitre 5 présente des perspectives de recherches et une conclusion.

## 2 Le problème #CSP

### 2.1 Définitions préliminaires

Un réseau de contraintes  $N$  est défini par un triplet  $(var(N), dom(N), cont(N))$  où  $var(N)$  est l'ensemble des variables du problème,  $dom(N)$  est l'ensemble des domaines de ces variables, c'est-à-dire les ensembles respectifs des valeurs que peut prendre chaque variable et  $cont(N)$  est un ensemble des contraintes portant sur ces variables.

**Définition 1** *Un réseau de contraintes  $N = (var(N), dom(N), cont(N))$  est défini par :*

- $var(N)$  est un ensemble fini de variables  $\{X_1, \dots, X_n\}$ .
- $dom(N)$  est un ensemble fini  $\{D_1, \dots, D_n\}$  de domaines.
- $\forall i \leq n, D_i$  est un ensemble fini  $\{v_{i_1}, \dots, v_{i_{|D_i|}}\}$  de valeurs possibles pour la variable  $X_i \in var(N)$ .
- $cont(N)$  est un ensemble fini de contraintes  $\{c_1, c_2, \dots, c_e\}$ .

On notera par la suite  $dom(X_i)$  le domaine  $D_i$  de  $X_i$ .

Une contrainte  $c \in C$  est une fonction booléenne qui implique une séquence de variables  $var(c) \subset var(X)$  appelée schéma de  $c$  (scope) :

$$c : \prod_{X_i \in var(c)} dom(X_i) \longrightarrow \{0, 1\}$$

On dit d'un tuple de valeurs  $(v_1, \dots, v_{|var(c)|})$  qu'il *satisfait*  $c$  si  $c(v_1, \dots, v_{|var(c)|}) = 1$ . Pour simplifier la présentation, on supposera les contraintes binaires et telles que  $\forall c, c' \in cont(N), var(c) \neq var(c')$ . On nommera  $c_{ij}$  l'unique contrainte entre  $X_i$  et  $X_j$ .

Voici un exemple de réseau de contraintes binaires  $N$  :

- $var(N) = \{X_1, \dots, X_4\}$
- $dom(X_i) = \{1, \dots, 3, \} \forall i \leq 3$
- $cont(N) = \{X_1 \neq X_2, X_2 \neq X_3, X_2 \neq X_4, X_3 \neq X_4\}$

A un réseau de contraintes  $N$ , on peut associer un (hyper)graphe  $G_N$  appelé graphe de contraintes. Chaque sommet du graphe est identifié à une variable et chaque (hyper)arête à une contrainte ; deux sommets sont reliés par une arête si

les variables associées participent à la même contrainte. On confondra dans la suite l'ensemble  $var(N)$  et l'ensemble des sommets de  $G_N$ .

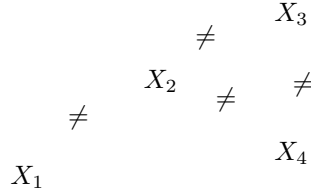


FIG. 2.1 – le graphe de contraintes associé à  $N$

**Définition 2** Une affectation  $\alpha$  de valeurs aux variables de  $Y \subset var(N)$  est un ensemble de couples  $\{(X_1, v_1), \dots, (X_{|Y|}, v_{|Y|})\}$  tel que  $\forall (X_i, v_i) \in \alpha, v_i \in dom(X_i)$

Étant donné une affectation  $\alpha$  de valeurs à un ensemble de variables  $Y$  et un sous-ensemble  $Z \subset Y$ , on note  $\alpha|_Z$  la restriction de  $\alpha$  à  $Z$ .

Pour un ensemble de variables  $Y \subset var(N)$ , on note  $cont(Y)$  la restriction de  $cont(N)$  aux variables de  $Y$  :

$$cont(Y) = \{c \in cont(N) \mid var(c) \subset var(Y)\}$$

Une affectation pour  $Y \subset var(N)$  est *consistante* si  $\forall c \in cont(Y), c$  est satisfaite.

Une *solution* du réseau de contraintes  $N$  est une affectation consistante qui porte sur toutes les variables de  $var(N)$ , on notera  $sol(N)$  l'ensemble des solutions de  $N$ .

Dans l'exemple précédent,  $\{(X_1, 1), (X_2, 2), (X_3, 3), (X_4, 1)\}$  est une solution car toutes les variables sont affectées et chaque contrainte est satisfaite.

Le problème de satisfaction de contraintes s'exprime alors ainsi :

### Définition 3 CSP

**Entrée** : Un réseau de contraintes  $N$

**Résultat** : Existe-t-il une solution de  $N$  ? ( $sol(N) \neq \emptyset$ )

CSP fait partie de la classe NP des problèmes décisionnels (CSP est NP-complet)

## 2.2 Énoncé du problème #CSP

Le problème de comptage des solutions d'un réseau de contraintes  $N$  se définit ainsi :

**Définition 4 #CSP**

**Entrée** : Un réseau de contraintes  $N$

**Résultat** :  $|sol(N)|$  le nombre de solutions distinctes de  $N$ .

A un problème de la classe NP, on peut associer un problème de comptage. Cette classe de problème est nommée #P.

On montre facilement que #CSP appartient à la classe #P. Tout d'abord on introduit le problème SAT :

**Définition 5 SAT**

**Entrée** : Une formule  $f$  de la logique propositionnelle exprimée sous forme normale conjonctive (FNC)

**Résultat** : Existe-t-il une affectation des variables de  $f$  qui rend  $f$  valide ?

Le problème #SAT est alors le problème de comptage associé au problème décisionnel SAT. #SAT est #P-difficile ([Valiant, 1979]) et il se réduit à #CSP ([Roth, 1996]) donc #CSP est #P-difficile.

Étant donnée une instance  $f$  de #SAT, on construit un réseau  $N$  ainsi : les variables de  $N$  sont les variables de  $f$  (ces variables ont pour domaine  $\{0, 1\}$ ). Pour chaque clause de  $f$  on ajoute une contrainte  $c$  dans  $N$ , qui lie les variables présentes dans la clause, telle que  $c$  est satisfaite par une affectation lorsqu'elle correspond à une valuation qui rend vrai cette clause (0 est interprété comme  $\perp$  et 1 comme  $\top$ ). Le nombre de modèles de la FNC est alors égal au nombre de solutions dans le réseau de contraintes produit.

## 2.3 État de l'art

Comparativement à #SAT, le problème #CSP a été assez peu étudié. La majorité des algorithmes sont des adaptations d'algorithmes de #SAT ([Lozinskii, 1992; Birnbaum et Lozinskii, 1999; Bayardo et Pehoushek, 2000]). En dehors de méthodes approchées [Kask *et al.*, ; Pesant, 2005; Wei et Selman, 2005] que nous ne développerons pas, on trouve peu d'algorithmes exacts.

L'algorithme CountingDavisPutnam (CDP) décrit dans [Birnbaum et Lozinskii, 1999] est basé sur l'algorithme DavisPutnam (DP), de résolution d'instance de SAT. La procédure DP est modifiée pour compter le nombre de solutions de la FNC. L'algorithme CDP tire parti du fait que lorsque toutes les clauses de la FNC sont satisfaites, les variables non affectées peuvent prendre n'importe quelle valeur pour former une solution.

L'algorithme DecomposingDavisPutnam (DDP) introduit dans [Bayardo et Pehoushek, 2000] peu après CDP, est lui aussi basé sur l'algorithme DP. A chaque nœud de l'arbre de recherche, les composantes connexes du sous-graphe induit par les variables non affectées sont identifiées et la procédure est rappelée récursivement sur chaque composante. Les résultats sont multipliés entre eux pour obtenir le nombre total de solutions. DDP se comporte mieux que CDP sur des graphes aléatoires.

L'algorithme de comptage présenté dans [Angelsmark *et al.*, 2002] effectue une décomposition d'un réseau de contraintes binaires en instances de #2-SAT dont la somme des nombres de solutions est égale au nombre de solutions dans le réseau de contraintes de départ. La décomposition en instance de #2-SAT consiste en une partition des domaines en paires de valeurs, multipliant ainsi les instances pour chaque combinaison de paires. Le nombre d'instance ne dépend donc que du nombre de variables et de la taille des domaines.

Dans une approche différente, [Weigel et Faltings, 1999] propose un cadre de *compilation* de réseaux de contraintes. Le concept d'interchangeabilité de variables et de valeurs  $y$  est introduit et est utilisé en conjonction de méthodes de décomposition structurelle pour construire une structure qui décrit le CSP à différents niveaux. Des "métavariabes" (respectivement des "métavaleurs") regroupent des tuples de variables (respectivement des classes d'équivalence de valeur) des niveaux inférieurs. Une fois la phase de compilation terminée, il est possible de répondre rapidement (linéaire en la taille de la structure) aux requêtes d'un utilisateur, en spécifiant des valeurs pour "filtrer" la réponse. La technique permet, entre autre de compter les solutions du réseau.

L'algorithme présenté ici est une adaptation d'une technique issue des travaux de Robertson et Seymour sur les mineurs d'un graphe [Robertson et Seymour, 1986]. Ces travaux ont débouché sur une série d'articles qui introduisent, entre autres, la décomposition arborescente d'un graphe et la *treewidth*. On peut se référer à [Bodlaender, 1993] pour une présentation générale des résultats. La *treewidth* est une mesure qui capture la *ressemblance* d'un graphe à un arbre. La programmation dynamique dans les graphes de *treewidth* bornée, présentée de manière générique dans [Bodlaender, 1997], est un patron d'algorithme pour les problèmes NP décisionnels et d'optimisation sur les graphes. Beaucoup de problèmes NP-difficiles deviennent polynomiaux ou linéaires lorsque la *treewidth* est bornée. Nous proposons une adaptation de cette technique pour le problème #CSP, laquelle peut aussi s'étendre aux problèmes de comptage de solutions associés aux problèmes décisionnels de graphes dans NP. [Dechter et Pearl, 1989] utilise la décomposition arborescente pour le problème CSP dans la méthode de TreeClustering dont nous réutilisons la phase de décomposition.

## 3 L'algorithme

### 3.1 Structure arborescente

L'idée générale de l'algorithme est de tirer parti de la structure du réseau, plus précisément de la structure du graphe de contraintes associé. En effet, si le graphe de contraintes est un arbre, le calcul du nombre de solutions peut s'effectuer en un temps polynomial. Nous proposons donc d'effectuer une décomposition arborescente du graphe de contraintes. Ceci nous permet de modéliser un nouveau problème dont le nombre de solutions est identique au problème d'origine. Les variables du nouveau problème représentent des sous-réseaux du réseau original. Le comptage est effectué sur le nouveau problème qui a une structure arborescente.

#### 3.1.1 Comptage des solutions dans un arbre

Nous allons montrer l'intérêt calculatoire des réseaux à structure arborescente pour le problème #CSP. L'algorithme suivant est un algorithme de comptage de solutions d'un réseau dont le graphe de contraintes est un arbre.

---

**Algorithme 1** : TreeCount

---

**Data** : Un réseau de contraintes  $N$  tel que  $G_N$  est un arbre.

**Result** : Le nombre de solutions de  $N$ .

```
1 begin
2   |   calculer  $G_N$ 
3   |   enracer  $G_N$  en une racine  $X_r$ 
4   |   return  $\sum_{v \in \text{dom}(X_r)} \text{nbsol}(X_r, v)$ 
5 end
```

---

L'algorithme TreeCount utilise une procédure récursive  $\text{nbsol}()$  définie ci-dessous ( $\Gamma^+$  désigne la fonction successeurs dans  $G_N$  enraciné) :

Il est clair que si la fonction  $\text{nbsol}()$  est correcte, l'algorithme TreeCount calcule bien le nombre de solutions de  $N$ , c'est à dire la somme du nombre de solutions affectant chaque valeur de la variable  $X_r$ .

---

**Fonction**  $nbsol(X_i, v)$ 

---

**Data** : Un noeud  $X_i$  de l'arbre  $G_N$ , une valeur  $v \in dom(X_i)$ .

**Result** : Le nombre de solutions affectant  $v$  à  $X_i$  du problème associé au sous-arbre de  $G_N$  enraciné en  $X_i$ .

```
1 begin
2    $res \leftarrow 1$ 
3   if  $\Gamma^+(X_i) \neq \emptyset$  then
4     foreach  $X_j \in \Gamma^+(X_i)$  do
5        $sum \leftarrow 0$ 
6       foreach  $v' \in dom(X_j)$  do
7         if  $c_{ij}(v, v') = 1$  then
8            $sum \leftarrow sum + nbsol(X_j, v')$ 
9        $res \leftarrow res \times sum$ 
10  return  $res$ 
11 end
```

---

**Théorème 1** Pour tout  $X_i \in G_N$  et tout  $v \in dom(X_i)$ ,  $nbsol(X_i, v)$  calcule le nombre de solutions du problème associé au sous-arbre de  $G_N$  enraciné en  $X_i$ , affectant  $v$  à  $X_i$ .

**Preuve**

Pour tout  $X_i \in G_N$  notons  $P_i$  le problème associé au sous-arbre de  $G_N$  enraciné en  $X_i$ .

Pour tout  $X_j \in \Gamma^+(X_i)$ , notons aussi  $P_i(X_j)$  le problème associé au sous-arbre de  $G_N$  enraciné en  $X_i$ , dont l'ensemble des successeurs directs est restreint à  $X_j$ .

Pour toute valeur  $v \in dom(X_i)$ , notons enfin  $\eta(P_i, v)$  (respectivement  $\eta(P_i(X_j), v)$ ) le nombre de solutions de  $P_i$  (respectivement  $P_i(X_j)$ ) affectant  $v$  à  $X_i$ .

Montrons que, pour tout  $X_i \in G_N$  et tout  $v \in dom(X_i)$ , on a :

$$nbsol(X_i, v) = \eta(P_i, v)$$

Nous allons procéder par induction structurelle sur  $G_N$ .

Si  $X_i$  est une feuille, alors le sous-arbre enraciné en  $X_i$  est un sommet isolé et  $\eta(P_i, v)$  est évidemment 1, valeur renvoyée par  $nbsol(X_i, v)$ .

Supposons maintenant que  $X_i$  ait des successeurs. Soient  $X_j \in \Gamma^+(X_i)$  et  $v' \in dom(X_j)$ .

Toute affectation contenant les couples  $(X_i, v)$  et  $(X_j, v')$  s'étend à une solution de  $P_i(X_j)$  si  $c_{ij}(v, v') = 1$  et si  $(X_j, v')$  s'étend à une solution de  $P_j$ . On a donc :

$$\eta(P_i(X_j), v) = \sum_{v' \in dom(X_j) | c_{ij}(v, v')=1} \eta(P_j, v')$$

Comme le sous-arbre de  $G_N$  enraciné en  $X_j$  contient nécessairement moins de sommets que celui enraciné en  $X_i$ , par hypothèse d'induction on obtient :

$$\eta(P_i(X_j), v) = \sum_{v' \in \text{dom}(X_j) | c_{ij}(v, v')=1} \text{nbsol}(X_j, v')$$

Maintenant, comme les problèmes  $P_i(X_j)$  associés à chaque successeur  $X_j \in \Gamma^+(X_i)$  sont deux à deux indépendants, on a pour tout  $X_i \in G_N$  et tout  $v \in \text{dom}(X_i)$  :

$$\eta(P_i, v) = \prod_{X_j \in \Gamma^+(X_i)} \eta(P_i(X_j), v)$$

Ce qui correspond exactement à la valeur calculée par l'appel à  $\text{nbsol}(X_i, v)$ .  $\square$

**Théorème 2** *TreeCount a une complexité polynomiale en la taille du réseau. (considérée comme le nombre de variables)*

### Preuve

Si on utilise une mémorisation des valeurs de  $\text{nbsol}$  pour chaque couple  $(X, v)$  où  $X \in \text{var}(N)$  et  $v \in \text{dom}(X)$ , par une table de hachage par exemple, une grossière analyse de TreeCount donne une complexité en  $\mathcal{O}(n^2 d^2)$  où  $n = |\text{var}(N)|$  et  $d = \max_{D_i \in \text{dom}(N)} |D_i|$ . Le nombre d'appels à  $\text{nbsol}()$  est inférieur à  $nd$  grâce à la mémorisation car il y a au plus  $nd$  combinaisons de variable/valeur. L'appel à  $\text{nbsol}()$  effectue au plus  $\mathcal{O}(nd)$  opérations :  $d$  additions répétées au plus  $n - 1$  fois et au plus  $n - 1$  multiplications ( $|\Gamma^+| < n$ ).  $\square$

Un bref résumé des propriétés des réseaux de contraintes à structure arborescente peut être trouvé dans [Russell et Norvig, 2003].

### 3.1.2 Généralisation aux $k$ -arbres

Pour pouvoir procéder de manière similaire dans un réseau quelconque, nous utilisons une technique de décomposition du graphe de contraintes. Le procédé, appelé programmation dynamique dans les  $k$ -arbres, consiste en une phase de décomposition, suivie d'une exploration de cette décomposition.

**Définition 6** *Une décomposition arborescente d'un graphe  $G$  est une paire  $(T, \lambda)$  où  $T$  est un arbre et  $\lambda$  une fonction d'étiquetage des sommets de  $T$ , à valeurs dans  $2^{\text{sommets}(G)}$ , qui vérifie :*

- $\forall s \in \text{sommets}(G), \exists t \in \text{sommets}(T)$  tel que  $s \in \lambda(t)$  (couverture des sommets)
- $\forall (s_1, s_2) \in \text{aretes}(G), \exists t \in \text{sommets}(T)$  tel que  $s_1 \in \lambda(t) \wedge s_2 \in \lambda(t)$  (couverture des arêtes)

- $\forall s \in \text{sommets}(G)$ , l'ensemble  $\{t \in \text{sommets}(T) \text{ tels que } s \in \lambda(t)\}$  induit un sous-arbre connexe de  $T$  (connexité)

**Définition 7** La treewidth d'une décomposition est la taille maximum des étiquettes de ses sommets moins un :

$$\text{treewidth}(T) = \max_{t \in \text{sommets}(T)} (|\lambda(t)|) - 1$$

**Définition 8** La treewidth d'un graphe est le minimum des treewidth sur l'ensemble de ses décompositions.

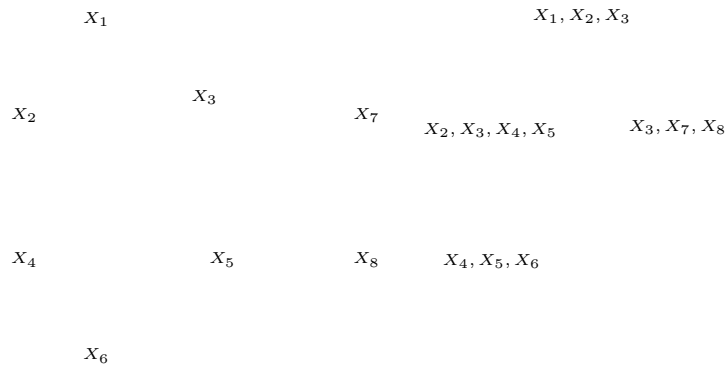


FIG. 3.1 – Un graphe de treewidth 3 et une de ses décompositions arborescentes

Les graphes de treewidth bornée par une constante  $k$  sont aussi appelés  $k$ -arbres partiels (une généralisation des arbres en théorie des graphes).

Par construction de la décomposition, deux sommets adjacents dans  $T$  partagent des variables<sup>1</sup>. L'ensemble de ces variables communes est appelé *séparateur* de ces deux sommets.

**Définition 9** Pour un ensemble de variables  $V \subset \text{var}(N)$ , on définit  $N(V)$  la restriction de  $N$  aux variables de  $V$  :

- $\forall X_i \in \text{var}(N), X_i \in V \Rightarrow X_i \in \text{var}(N(V))$
- $\text{dom}(N(V))$  est la restriction de  $\text{dom}(N)$  aux variables de  $V$
- $\forall c_{ij}(X_i, X_j) \in \text{cont}(N), X_i \in V \wedge X_j \in V \Rightarrow c_{ij}(X_i, X_j) \in \text{cont}(N(V))$

Pour un sommet  $t \in \text{sommets}(T)$  on notera  $N_t = N(\lambda(t))$ , la restriction de  $N$  aux variables associées à  $t$ .

La décomposition peut être vue comme un nouveau réseau de contraintes  $N'$  qui a pour graphe de contraintes l'arbre  $T$ .

<sup>1</sup>On considère que  $T$  ne contient pas d'arête entre deux sommets ne partageant pas de variables, bien que la définition ne l'interdise pas.

**Définition 10** Pour un réseau de contraintes  $N$  et une de ses décomposition  $T$ , on définit le réseau résultant  $N'$  ainsi :

- $\forall t \in \text{sommets}(T), \exists X_t \in \text{var}(N')$
- $\forall X_t \in \text{var}(N'), \text{dom}(X_t) = \text{sol}(N_t)$
- $\forall \{t_i, t_j\} \in \text{aretes}(T), \exists c_{ij} \in \text{cont}(N') \mid c_{ij}(v_i, v_j) = 1 \Leftrightarrow v_i|_S = v_j|_S$  (où  $S = \lambda(t_i) \cap \lambda(t_j)$  est le séparateur de  $t_i$  et  $t_j$ )

**Théorème 3** Tout réseau de contraintes a le même nombre de solutions que ses réseaux résultants.

### Preuve

Montrons qu'il existe une bijection de  $\text{sol}(N)$  dans  $\text{sol}(N')$ .

Soit  $\varphi : \text{sol}(N) \longrightarrow \text{sol}(N')$  une application définie par :

$$\forall \alpha \in \text{sol}(N), \varphi(\alpha) = ((X_{t_1}, \alpha|_{\lambda(t_1)}), \dots, (X_{t_q}, \alpha|_{\lambda(t_q)}))$$

Où  $q = |\text{sommets}(T)|$ .

**Proposition 1**  $\varphi$  est bien définie.

$\varphi$  est bien définie si pour toute solution  $\alpha$  de  $N$ ,  $\varphi(\alpha)$  est une affectation des variables de  $N'$  à une valeur dans leur domaine respectif, et si toutes les contraintes de  $N'$  sont satisfaites.

Soit  $(X_t, \alpha|_{\lambda(t)}) \in \varphi(\alpha)$ , comme  $\alpha$  est une solution de  $N$ , toutes les contraintes de  $\text{cont}(N)$  sont satisfaites, en particulier celles de  $\text{cont}(N_t)$  donc  $\alpha|_{\lambda(t)}$  est une solution de  $\text{sol}(N_t)$ , donc  $\alpha|_{\lambda(t)} \in \text{dom}(X_t)$ .

D'autre part les contraintes de  $N'$  sont satisfaites puisque les valeurs associées aux variables de  $\text{var}(N')$  par  $\varphi(\alpha)$  sont des projections de la même affectation  $\alpha$ . D'où  $\varphi(\alpha) \in \text{sol}(N')$ .

**Lemme 4**  $\varphi$  est injective :

$$\forall \alpha, \alpha' \in \text{sol}(N), \alpha \neq \alpha' \Rightarrow \varphi(\alpha) \neq \varphi(\alpha')$$

Soit  $\alpha, \alpha' \in \text{sol}(N)$  telles que  $\alpha \neq \alpha'$ . Il existe au moins une variable  $X \in \text{var}(N)$  qui est affectée à une valeur différente dans  $\alpha$  et  $\alpha'$ . D'autre part il existe au moins un sommet  $t \in T$  tel que  $X \in \lambda(t)$ . Donc  $\alpha|_{\lambda(t)} \neq \alpha'|_{\lambda(t)}$  d'où  $\varphi(\alpha) \neq \varphi(\alpha')$  (elles diffèrent par l'affectation d'au moins une variable).

**Lemme 5**  $\varphi$  est surjective :

$$\forall \beta \in \text{sol}(N'), \exists \alpha \in \text{sol}(N) \mid \varphi(\alpha) = \beta$$

Pour deux affectations  $\alpha(Y)$  et  $\alpha'(Y')$  telles que  $\alpha|_{Y \cap Y'} = \alpha'|_{Y \cap Y'}$ , on note  $\alpha \oplus \alpha'$  la jonction de  $\alpha$  avec  $\alpha'$ .  $\alpha \oplus \alpha'$  est une affectation de valeur aux variables de  $Y \cup Y'$

Soit  $\beta \in \text{sol}(N')$ . On va construire une affectation  $\alpha$  des variables de  $\text{var}(N)$  telle que  $\alpha \in \text{sol}(N)$  et  $\varphi(\alpha) = \beta$ .

$$\text{Soit } \alpha = \bigoplus_{(X_t, v_t) \in \beta} v_t.$$

Par définition des contraintes du réseau résultant  $N'$ , quelque soit  $t_i, t_j \in T$ , les variables communes à  $X_{t_i}$  et  $X_{t_j}$  ont même valeur dans  $v_i$  et  $v_j$ , donc  $\alpha$  est bien définie.

Par définition de la décomposition arborescente, ces deux égalités sont vérifiées :

$$\text{var}(N) = \bigcup_{t \in T} \lambda(t) \quad (3.1)$$

$$\text{cont}(N) = \bigcup_{t \in T} \text{cont}(N_t) \quad (3.2)$$

D'après 3.1,  $\alpha$  est une affectation de toutes les variables de  $N$ . On montre ensuite que  $\alpha$  satisfait toutes les contraintes de  $N$ .

Pour tout sommet  $t \in T$ ,  $v_t$  appartient au domaine de  $X_t$ , donc les contraintes de  $\text{cont}(N_t)$  sont satisfaites (par définition de  $\text{dom}(N')$ ). Donc, avec 3.2, toutes les contraintes de  $\text{cont}(N)$  sont satisfaites. D'où :

$$\alpha \in \text{sol}(N)$$

Il nous faut vérifier que  $\varphi(\alpha) = \beta$ .

En remarquant que pour tout  $t \in \text{sommets}(T)$ , on a  $\alpha|_{\lambda(t)} = v_t$ , on obtient :

$$\varphi(\alpha) = ((X_{t_1}, v_1), \dots, (X_{t_q}, v_q)) = \beta$$

$\varphi$  est injective et surjective, donc elle est bijective. □

## 3.2 L'algorithme kTreeCount

La décomposition arborescente a des propriétés intéressantes. Les variables d'un sommet de la décomposition se divisent en deux parties : les variables dites *internes* au sous-problème ne sont présentes que dans ce sommet de la décomposition et les variables dites *externes* au sous-problème sont présentes dans au moins un autre sommet.

Comme les variables internes sont locales à un sous-problème, c'est-à-dire qu'aucune contrainte ne lie une de ces variables avec une variable présente dans un autre sommet de la décomposition, seules les affectations des variables

externes sont nécessaires pour étendre la solution partielle à une solution plus globale.

A chaque noeud de la décomposition, les solutions partielles sont calculées puis restreintes aux variables externes. On peut combiner ces solutions partielles pour former des solutions globales, si les variables communes sont affectées aux mêmes valeurs.

### 3.2.1 Structure de données

A chaque sommet  $t$  de la décomposition arborescente du graphe de contraintes est associée une table  $F_t$  qui mémorise le nombre de solutions partielles du sous-réseau induit par les sommets de  $T$  déjà parcourus.

Les indices de la table d'un sommet  $t$  sont des affectations des variables contenues dans les séparateurs entre  $t$  et ses voisins dans  $T$ . Chaque table est donc bornée en nombre d'entrées par  $d^{tw}$  où  $d$  est la taille maximale des domaines et  $tw$  la treewidth de la décomposition.

Pour un sommet  $t \in T$ , soient  $\{t_1, \dots, t_p\}$  les  $p$  sommets successeurs de  $t$  et  $t_j$  son prédécesseur.

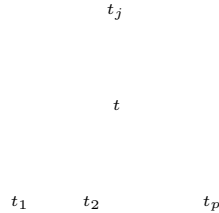


FIG. 3.2 – Indices des voisins d'un sommet  $t$  de  $T$ .

Soit  $I = \{1, \dots, p\} \cup \{j\}$ . On définit  $sep(t) = \bigcup_{i \in I} (S_i)$  l'union des séparateurs de  $t$  où  $S_i = \lambda(t) \cap \lambda(t_i)$  est le séparateur de  $t$  et  $t_i$ ,  $\forall i \in I$ .

$$F_t : \prod_{X_i \in sep(t)} dom(X_i) \longrightarrow \mathbb{N}$$

Si  $t \in T$  on définit  $N_t^\downarrow$ , la restriction de  $N$  aux variables associées aux sommets du sous-arbre de  $T$  enraciné en  $t$  :

$$N_t^\downarrow = N(\lambda(t)) \cup \bigcup_{t_i \in desc(t)} \lambda(t_i)$$

Où  $desc(t)$  désigne l'ensemble des descendants de  $t$ .

$F_t$  associe à chaque affectation possible  $\alpha$  des variables de  $sep(t)$ , le nombre de solutions de  $N_t^\downarrow$  dont la restriction sur  $sep(t)$  est  $\alpha$ .

### 3.2.2 Description de l’algorithme

Dans une phase de prétraitement, une décomposition arborescente du graphe de contraintes est effectuée.

#### Phase de décomposition

Différentes approches de recherche d’une décomposition arborescente, facilitant la résolution du réseau de contraintes associé, sont comparées dans [Gogate et Dechter, 2004; Jégou *et al.*, 2005b].

La recherche d’une décomposition arborescente de treewidth optimale est un problème NP-difficile dans le cas général ([Arnborg *et al.*, 1987]). L’heuristique MaximumCardinalitySearch (MCS) est basée sur un algorithme de reconnaissance de graphe triangulaire (chordal) introduit par [Tarjan et Yannakakis, 1984] et utilisée dans la méthode de TreeClustering [Dechter et Pearl, 1989]. Cette heuristique semble être un bon compromis entre rapidité et qualité de la décomposition, tout en étant simple à implémenter.

Par ailleurs l’algorithme de [Reed, 1992] a une complexité en  $\mathcal{O}(n \log(n))$  et fournit une décomposition  $T$  de treewidth  $tw_T \leq 4tw_G$ .

Pour des raisons pratiques, c’est l’heuristique MCS qui a été choisie pour l’implémentation de la phase de décomposition du graphe de contraintes.

#### Phase principale

Deux approches sont possibles : incrémentale et non incrémentale. L’approche incrémentale permet une interaction avec l’utilisateur, les tables sont maintenues en mémoire et mises à jour lorsqu’il y a modification par l’utilisateur. L’approche, non incrémentale, minimise la mémoire utilisée en effaçant les tables au fur et à mesure du calcul. C’est cette dernière que nous développons par la suite.

Pour minimiser le nombre de tables présentes en mémoire simultanément, la décomposition est parcourue des feuilles vers la racine et en largeur. La table d’un successeur de  $t$  peut être effacée de la mémoire dès qu’elle a servi à mettre à jour la table de  $t$ . [Aspvall *et al.*, 1998] présente un algorithme générique de minimisation d’espace mémoire utilisé lors du parcours de la décomposition. L’algorithme se base sur la taille de la table générée à chaque sommet. Le résultat est un ordre total de traversée de la décomposition, déterminant ainsi la racine de la décomposition et l’ordre dans lequel les descendants d’un sommet doivent être explorés.

Il introduit une procédure d'exploration de l'arbre appelée DFS qui est la boucle principale de l'algorithme (légèrement modifiée<sup>2</sup>).

---

**Algorithme 3** : DFS

---

**Data** : Un noeud  $t \in T$  ayant pour successeurs ordonnés  $(t_1, \dots, t_p)$  (ou  $\emptyset$ ).  
**Result** :  $F_t$  est construite et remplie.

```

1 begin
2   if  $\Gamma^+(t) = \emptyset$  then
3     create-table( $t$ )
4     propagate( $t$ )
5   else
6     DFS( $t_1$ )
7     create-table( $t$ )
8     update-table( $t_1, t$ )
9     discard-table( $t_1$ )
10    for  $i$  in  $[2 \text{ to } p]$  do
11      DFS( $t_i$ )
12      update-table( $t_i, t$ )
13      discard-table( $t_i$ )
14    finalize-table( $t$ )
15    propagate( $t$ )
16 end

```

---

La généralité de cet algorithme se traduit par les trois procédures de création, de mise à jour et d'effacement des tables. L'astuce de la procédure DFS est de retarder la création de la table  $F_t$  jusqu'à la fin du calcul de la table du premier successeur de  $t$ . La table  $F_t$  est ensuite mise à jour pour chacun des successeurs  $\{t_2, \dots, t_p\}$  de  $t$ . Une fois les mises à jour avec tous les successeurs effectuées, la table est prête pour la mise à jour de la table de son unique prédécesseur  $F_{t_j}$ .

L'ordre, calculé par l'algorithme de minimisation de l'espace mémoire requis, assure que le premier successeur  $t_1$  d'un sommet  $t$  est celui qui nécessite le plus de mémoire. [Aspvall *et al.*, 1998] établit une borne sur le nombre de tables en mémoire simultanément, grâce à cet ordre de traversée, à  $2 \times \text{pathwidth}(T)$ <sup>3</sup>.

---

<sup>2</sup>Les procédures *finalize-table()* et *propagate()* ont été ajoutées pour des raisons techniques détaillées plus loin.

<sup>3</sup>La *pathwidth* est définie de manière similaire à la *treewidth*, la décomposition doit être une chaîne au lieu d'un arbre.

### Création de la table $F_t$

L'appel de la procédure *create-table*( $t$ ) initialise la table  $F_t$  et la remplit avec les solutions de  $N_t$ .

La valeur dans  $F_t$  associée à une affectation  $\alpha$  des variables de  $sep(t)$  est égale au nombre de solutions de  $N_t$  dont la restriction sur  $sep(t)$  est  $\alpha$ .

$$\forall(\alpha, F_t(\alpha)) \in F_t, F_t(\alpha) = |\{s \in sol(N_t) \text{ telles que } s|_{sep(t)} = \alpha\}|$$

Pour un sous-problème  $N_t$  tout l'arbre de recherche doit être exploré par le solveur. Donc *create-table*() a un temps d'exécution en  $\mathcal{O}(d^{|var(N_t)|})$  (avec  $d$  la taille maximum des domaines). Or  $|var(N_t)| \leq treewidth(T) + 1$  donc la procédure *create-table*() est exponentielle en la taille de  $N$  dans le pire des cas, mais polynomiale si la treewidth est bornée par une constante.

Les solutions sont énumérées à l'aide du solveur CHOCO dont on peut trouver les détails d'implémentation dans [Laburthe, 2000].

### Mise à jour de la table $F_t$

L'appel de la procédure *update-table*( $t_i, t$ ) effectue une sorte de "jointure" entre les tables  $F_t$  et  $F_{t_i}$ . Une solution de  $N_t$  est *compatible* avec une solution de  $N_{t_i}$  si les variables de  $S_i$  (le séparateur entre  $t$  et  $t_i$ ) sont affectées aux mêmes valeurs dans  $N_t$  et dans  $N_{t_i}$ . Les entrées de  $F_t$  sont donc multipliées par les entrées de  $F_{t_i}$  compatibles.

$F_{t_i}$  est indexée par des affectations des variables de  $S_i$ <sup>4</sup>. Comme  $S_i \in sep(t)$ , les entrées de  $F_t$  peuvent être restreintes à  $S_i$ .

La mise à jour consiste en une énumération des éléments de la table  $F_t$ . Pour chaque entrée  $(\alpha, F_t(\alpha))$ , on met à jour  $F_t(\alpha)$  avec  $F_{t_i}(\alpha|_{S_i})$  c'est-à-dire :

$$F_t(\alpha) \leftarrow F_t(\alpha) \cdot F_{t_i}(\alpha|_{S_i})$$

Après la fin de l'énumération pour le sommet  $t_i$ , la table  $F_t$  associe à chaque affectation  $\alpha$  des variables de  $sep(t)$  le nombre de solutions du réseau induit par l'union de  $t$  avec les sous-arbres enracinés en  $t_\ell$ ,  $\ell \in \{1, \dots, i\}$ .

A la fin de la mise à jour avec tous les successeurs de  $t$ , la valeur dans  $F_t$  associée à une affectation  $\alpha$  est égale au nombre de solutions de  $N_t^\downarrow$  dont la restriction sur  $sep(t)$  est  $\alpha$ . Pour chaque entrée  $(\alpha, F_t(\alpha))$  de  $F_t$ , on a :

$$F_t(\alpha) = |\{\beta \in sol(N_t^\downarrow) \mid \beta|_{sep(t)} = \alpha\}|$$

Le nombre total de solutions de  $N_t^\downarrow$  est alors :

$$|sol(N_t^\downarrow)| = \sum_{(\alpha, F_t(\alpha)) \in F_t} F_t(\alpha)$$

---

<sup>4</sup>La procédure *finalize-table*() effectue une réduction de l'espace des index de  $F_{t_i}$  de  $sep(t_i)$  vers  $sep(t_i) \cap sep(t) = S_i$  après la mise à jour de  $F_{t_i}$  avec les tables des successeurs de  $t_i$ . (voir 3.3.1)

A la fin du parcours, si on appelle  $r$  la racine de  $T$ , on a :  $N_r^\downarrow = N$  donc :

$$|sol(N)| = \sum_{(\alpha, F_r(\alpha)) \in F_r} F_r(\alpha)$$

### 3.2.3 Correction

La procédure DFS est un parcours des feuilles vers la racine. En particulier l'ordre de traversée produit par DFS assure qu'à chaque mise à jour d'une table  $F_t$  avec une table  $F_{t_i}$ ,  $t_i \in \Gamma^+(t)$ ,  $F_{t_i}$  a déjà été créée et mise à jour avec les tables de tous les successeurs de  $t_i$  ([Aspvall *et al.*, 1998]).

On va montrer que, sous cette hypothèse, la table  $F_t$  contient les solutions du problème associé au sous-arbre de  $T$  enraciné en  $t$ , après un appel à la procédure DFS( $t$ ).

**Théorème 6** *Pour un sommet  $t$  de la décomposition d'un réseau  $N$ , ayant pour prédécesseur  $t_j$ , après l'appel à la procédure DFS( $t$ ), on a :*

$$\forall (\alpha, F_t(\alpha)) \in F_t, F_t(\alpha) = |\{\beta \in sol(N_t^\downarrow) \mid \beta|_{S_j} = \alpha\}|$$

#### Preuve

Si  $t$  est une feuille ( $\Gamma^+(t) = \emptyset$ ), l'appel à *create-table*( $t$ ) remplit  $F_t$  avec les affectations de *sep*( $t$ ) qui s'étendent à une solution de  $N_t$  (associées au nombre de ces solutions). Or  $N_t = N_t^\downarrow$  puisque  $t$  est une feuille. L'appel à *finalize-table*( $t$ ) restreint l'espace d'indexage de  $F_t$  aux affectations de variables de  $S_j$ . Le théorème est donc vrai<sup>5</sup>.

Si  $t$  n'est pas une feuille, soit  $\{t_1, \dots, t_p\} = \Gamma^+(t)$ . DFS effectue alors les mise à jour successives de  $F_t$  avec les successeurs de  $t$ .

Pour un indice  $i$  d'un successeur  $t_i$  de  $t$ , on note  $N_t^i$  le sous-réseau de  $N$  associé aux variables des sommets du sous-arbre de  $T$  enraciné en  $t$  et restreint aux successeurs de  $t$  indexés par un indice  $\ell < i$ .

$$N_t^i = N(var(N_t) \cup \bigcup_{\ell=1}^i var(N_{t_\ell}^\downarrow))$$

On procède alors par induction sur  $i$  en exprimant l'hypothèse d'induction suivante :

---

<sup>5</sup>Les explications de la procédure *finalize-table*() sont repoussées en fin de preuve pour des raisons de clarification

**Hypothèse d'induction 1** Après la mise à jour de  $F_t$  avec  $F_{t_i}$ ,  $i \leq p$ , on a :

$$\forall (\alpha, F_t(\alpha)) \in F_t, F_t(\alpha) = |\{\beta \in \text{sol}(N_t^i) \mid \beta_{|_{\text{sep}(t)}} = \alpha\}|$$

Si  $i = 1$ . Soit  $(\alpha, F_t(\alpha)) \in F_t$  avant la mise à jour avec  $F_{t_i}$  et soit  $S_i \subset \text{sep}(t)$  le séparateur de  $t$  et  $t_i$ .

On va montrer le lemme suivant qui justifie la mise à jour de  $F_t(\alpha)$  :

**Lemme 7**  $|\{\beta \in \text{sol}(N_t) \mid \beta_{|_{\text{sep}(t)}} = \alpha\}| \cdot |\{\beta \in \text{sol}(N_{t_i}^\downarrow) \mid \beta_{|_{S_i}} = \alpha_{|_{S_i}}\}| = |\{\beta \in \text{sol}(N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))) \mid \beta_{|_{\text{sep}(t)}} = \alpha\}|$

- ( $\leq$ ) Soit  $s$  une solution de  $N_t$  telle que  $s_{|_{\text{sep}(t)}} = \alpha$  et  $s'$  une solution de  $N_{t_i}^\downarrow$  telle que  $s'_{|_{S_i}} = \alpha_{|_{S_i}}$ . Soit  $\beta = s \oplus s'$ .  $\beta$  est une affectation de valeur aux variables de  $\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow)$ . Comme  $\text{var}(N_t) \cap \text{var}(N_{t_i}^\downarrow) = S_i$  par définition de la décomposition (propriété de connexité) et comme  $s_{|_{S_i}} = s'_{|_{S_i}} = \alpha_{|_{S_i}}$  par construction de  $s$  et  $s'$ ,  $\beta$  est bien définie.

Montrons que  $\beta$  satisfait toutes les contraintes de  $N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))$ . La restriction de  $\beta$  aux variables de  $N_t$  est  $s$  par construction de  $\beta$ . Or  $s$  est solution de  $N_t$ , donc  $\beta$  satisfait les contraintes de  $\text{cont}(N_t)$ . De la même manière,  $\beta_{|_{\text{var}(N_{t_i}^\downarrow)}} = s'$ , par construction de  $\beta$ , avec  $s'$  solution de  $N_{t_i}^\downarrow$ .  $\beta$  satisfait donc aussi les contraintes de  $\text{cont}(N_{t_i}^\downarrow)$ . En remarquant que  $\text{cont}(N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))) = \text{cont}(N_t) \cup \text{cont}(N_{t_i}^\downarrow)$ , par définition de la décomposition, on en conclue que  $\beta$  est une solution de  $N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))$ . Enfin  $\beta_{|_{\text{sep}(t)}} = s_{|_{\text{sep}(t)}} = \alpha$  par construction de  $\beta$ .

- ( $\geq$ ) Soit  $\delta$  une solution de  $N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))$  telle que  $\delta_{|_{\text{sep}(t)}} = \alpha$ .  $\delta_{|_{\text{var}(N_t)}}$  est une solution de  $N_t$  puisque les contraintes de  $\text{cont}(N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow)))$  sont satisfaites par  $\delta$ , donc en particulier celles de  $\text{cont}(N_t)$ . Par définition de  $\text{create-table}()$ ,  $\delta_{|_{\text{var}(N_t)}}$  est comptabilisée par  $F_t(\delta_{|_{\text{sep}(t)}}) = F_t(\alpha)$ . De la même manière,  $\delta_{|_{\text{var}(N_{t_i}^\downarrow)}}$  est une solutions de  $N_{t_i}^\downarrow$ , donc elle est comptabilisée dans  $F_{t_i}(\delta_{|_{S_i}}) = F_{t_i}(\alpha_{|_{S_i}})$ , par hypothèse sur  $F_{t_i}$ . Finalement  $\delta$  est la jonction de deux solutions, respectivement de  $N_t$  et de  $N_{t_i}^\downarrow$ .

$F_t$  vient d'être construite, donc  $\alpha$  est une affectation des variables de  $\text{sep}(t)$  qui peut s'étendre à exactement  $F_t(\alpha)$  solutions distinctes de  $N_t$ . Par hypothèse sur  $F_{t_i}$ ,  $F_{t_i}(\alpha_{|_{S_i}})$  est le nombre de solutions distinctes de  $N_{t_i}^\downarrow$  dont la restriction aux variables de  $S_i$  est  $\alpha_{|_{S_i}}$ . La proposition 7 étant vérifiée, on a donc :

$$F_t(\alpha) \cdot F_{t_i}(\alpha_{|_{S_i}}) = |\{\beta \in \text{sol}(N(\text{var}(N_t) \cup \text{var}(N_{t_i}^\downarrow))) \mid \beta_{|_{\text{sep}(t)}} = \alpha\}|$$

En remarquant que  $N(\text{var}(N_t) \cup \text{var}(N_{t_1}^\downarrow)) = N_t^1$  on obtient :

$$F_t(\alpha) \cdot F_{t_1}(\alpha|_{S_1}) = |\{\beta \in \text{sol}(N_t^1) \mid \beta|_{\text{sep}(t)} = \alpha\}|$$

L'hypothèse d'induction est donc vraie pour  $i = 1$ . Supposons la vraie pour  $i - 1$  ( $i > 1$ ). La démonstration est alors semblable au cas  $i = 1$ , il suffit de remplacer  $N_t$  par  $N_t^{i-1}$ . On démontre alors que l'égalité est vraie grâce à l'hypothèse d'induction et en remarquant que  $N(\text{var}(N_t^{i-1}) \cup \text{var}(N_{t_i}^\downarrow)) = N_t^i$ .

Finalement on utilise l'hypothèse d'induction pour  $i = p$ , en remarquant que  $N_t^p = N_t^\downarrow$ . On obtient alors, après les mises à jour effectuées :

$$\forall(\alpha, F_t(\alpha)) \in F_t, F_t(\alpha) = |\{\beta \in \text{sol}(N_t^\downarrow) \mid \beta|_{\text{sep}(t)} = \alpha\}|$$

L'appel à *finalize-table*( $t$ ) restreint l'espace d'indexage de  $F_t$  aux affectations de variables de  $S_j$ , le séparateur de  $t$  et  $t_j$ . Comme deux affectations  $\alpha$  et  $\alpha'$  telles que  $\alpha \neq \alpha'$  et  $\alpha|_{S_j} = \alpha'|_{S_j}$  ne seront envoyés que sur un seul index, les valeurs associées sont sommées. En effet, les solutions représentées par  $F_t(\alpha)$  sont toutes différentes de celles représentées par le nombre associé à  $F_t(\alpha')$  dans puisque  $\alpha \neq \alpha'$ . Après l'appel à *finalize-table*( $t$ ), le nombre associé à  $\alpha|_{S_j}$  dans  $F_t$  doit donc représenter l'ensemble de ces solutions (soit la somme de leur nombre). On a alors, si  $t$  n'est pas la racine :

$$\forall(\alpha, F_t(\alpha)) \in F_t, F_t(\alpha) = |\{\beta \in \text{sol}(N_t^\downarrow) \mid \beta|_{S_j} = \alpha\}|$$

□

On peut remarquer que la finalisation de la table de la racine, réduit l'espace d'indexage à  $\emptyset$ , ce qui revient à sommer toutes les entrées. Cette opération est effectuée par l'appelant de DFS, la table n'étant pas effacée.

### 3.2.4 Complexité

On considère la taille de la donnée comme le nombre de variables du réseau et la taille des domaines. On suppose la treewidth du graphe de contraintes bornée par une constante  $k$ .

**Théorème 8** *L'algorithme  $k\text{TreeCount}(N)$  a une complexité polynomiale en la taille de  $N$  et exponentielle en la treewidth du graphe de contraintes de  $N$ .*

#### Preuve

La procédure DFS est un parcour de  $T$ , donc DFS est appelée récursivement au plus  $|T|$  fois. Comme  $|T| \in \mathcal{O}(n)$  ([Bodlaender, 1993]) avec  $n = |\text{var}(N)|$ ,

le nombre d'appel à DFS est borné par  $f(n)$ , avec  $f(n)$  une fonction linéaire en  $n$ .

A chaque appel de DFS, *create-table()* est appelée une fois et *update-table()* au plus  $n$  fois ( $|\Gamma^+(t)| \leq n$ ). L'appel à *create-table()* énumère les solutions de  $N_t$ , chaque solution est projetée sur  $sep(t)$ . Donc l'appel à *create-table()* est en  $\mathcal{O}(kd^k)$  avec  $d = \max_{D \in dom(N)} |D|$ . L'appel à *update-table()* parcourt la table  $F_t$  qui est bornée en taille par  $d^k$ . Pour chaque entrée, de  $F_t$  une projection est calculée ( $\mathcal{O}(k)$ ) et un accès à la table  $F_{t_i}$  est effectué ( $\mathcal{O}(1)$ ). Donc l'appel à *update-table()* est en  $\mathcal{O}(kd^k)$ . La procédure *discard-table()* peut être implémentée en  $\mathcal{O}(1)$ . Enfin la procédure *propagate()* est en  $\mathcal{O}(n^2d^2)$  ([Russell et Norvig, 2003]) et est appelée au plus  $n$  fois.

Au total, l'algorithme a une complexité en  $\mathcal{O}(f(n)(nkd^k + n^3d^2))$ , l'algorithme est donc polynomial en  $n$  et exponentiel en la treewidth.  $\square$

### 3.3 Améliorations

#### 3.3.1 Finalisation des tables

Lors de la mise à jour de la table associée au prédécesseur  $t_j$  d'un sommet  $t$  avec la table  $F_t$ , les seules variables communes aux deux problèmes  $N_t$  et  $N_{t_j}$  sont celles du séparateur  $S_j$  de  $t$  et  $t_j$ . Or les entrées de  $F_t$  sont indexées par des affectations des variables de  $sep(t) \supset S_j$ . Pour ne pas avoir à effectuer une recherche dans les index de  $F_t$  à chaque accès à la table, il faut *formatter*  $F_t$ .

L'appel à la procédure *finalize-table(t)* réduit l'espace d'indexage de  $F_t$  aux affectations des variables de  $S_j$ . Une nouvelle table est créée et remplie avec les entrées de  $F_t$ , dont les index ont été restreints aux variables de  $S_j$ . Les entrées ayant la même valeur d'index dans la nouvelle table sont cumulées (somme). On peut alors accéder efficacement au nombre de solutions associé à une affectation des variables de  $S_j$ . La procédure est appelée après les mises à jour effectuées pour tous les successeurs de  $t$ .

#### 3.3.2 Détection de contradictions

Dans le cas d'une instance insatisfiable et si le problème n'est pas très sur-contraint, l'algorithme peut perdre du temps à énumérer des solutions partielles qui ne s'étendent pas à une solution lorsque l'on arrive à la racine. Dès qu'une incompatibilité est trouvée (une table se retrouvant vide après sa création ou après une mise à jour) la procédure DFS est interrompue, une exception est alors levée, qu'il faut interpréter comme un nombre nul de solution.

Une autre amélioration permet de diminuer la taille des tables lorsque les sous-problèmes associés à des sommets voisins n'ont que peu de solutions en commun. Le problème global est modélisé avec la bibliothèque de manière standard. Aucune résolution n'est lancée, mais les domaines des variables  $y$  sont réduits après chaque création ou mise à jour de table, lorsque des valeurs ne font partie d'aucune solution partielle. Grâce au maintien de l'arc-consistance par le solveur (appel de la procédure *propagate()*), les modifications sont répercutées sur les variables du problème global. A chaque création de table, avant l'énumération des solutions du sous-problème associé, le domaine de chaque variable du sous-problème instancié est réduit au domaine de la variable correspondant dans le problème global.

### 3.3.3 Occupation de la mémoire

Lors des calculs des tables des successeurs autres que le premier, la table  $F_t$  est déjà créée mais n'est pas nécessaire : elle le devient à la fin du calcul de chaque successeur pour la mise à jour. Les tables peuvent donc être stockées sur mémoire auxiliaire le temps du calcul des tables des successeurs, diminuant ainsi à 2 le nombre total de tables chargées simultanément en mémoire. Le temps de calcul étant dominé par l'étape de création de  $F_t$  (exponentielle en la treewidth dans le pire des cas), cette option ne dégrade sensiblement la vitesse d'exécution que sur les "petits" problèmes. Cependant la mise en cache des fichiers par le système d'exploitation permet d'économiser la mise en mémoire auxiliaire si la mémoire principale le permet.

D'autre part, nous avons implémenté l'algorithme décrit dans [Aspvall *et al.*, 1998] en proposant une estimation simple de la taille des tables. Ces tailles dépendant du nombre de solutions dans les sous-problèmes, leur estimation peut être améliorée grâce à une estimation plus réaliste du nombre de solutions du réseau de contraintes, par exemple selon la méthode décrite dans [Pesant, 2005].

# 4 Expérimentations

## 4.1 Protocole d'expérimentation

Les expérimentations se sont déroulées sur des plateformes Linux avec des instances produites de deux manières différentes : par un générateur standard de réseaux binaires uniformément aléatoires ou par un générateur de réseaux à structure  $k$ -arborescente. Les modèles de génération des instances sont basés sur un générateur pseudo-aléatoire reconnu (`ran2`), de manière à obtenir des résultats comparables avec d'autres algorithmes.

### 4.1.1 Graphes aléatoires

Les réseaux binaires uniformément aléatoires sont générés avec le générateur `urbcsp`.

Les paramètres du générateur sont :

- le nombre  $n$  de variables
- la taille  $d$  des domaines (toutes les variables ont le même domaine)
- le nombre  $e$  de contraintes ( $0 \leq e \leq \frac{n(n-1)}{2}$ )
- la dûtreté  $t$  des contraintes (*tightness*,  $0 \leq t \leq d^2$ )

Le générateur produit alors un réseau de contraintes avec  $n$  variables dont le domaine est  $\{0, \dots, d-1\}$  avec  $e$  contraintes binaires tirées aléatoirement et sans répétition. Chacune des  $e$  contraintes refuse  $t$  couples de  $\{0, \dots, d-1\}^2$  tirés aléatoirement et sans répétition.

Le nombre  $n$  de variables est fixé à 20, la taille des domaines à 6. Trois catégories d'instances sont différenciées, définissant le nombre de contraintes  $e$ . Pour chaque catégorie, 50 instances ont été générées pour chaque valeur  $t$  de la dûtreté des contraintes (exceptés les extremums 0 et  $d^2 = 36$ ).

### 4.1.2 $k$ -arbres

Afin d'observer la corrélation polynomiale entre le temps d'exécution de `kTreeCount` et le nombre de variables du réseau, nous avons développé un générateur

de réseaux à structure  $k$ -arborescente. En effets, les réseaux générés avec le générateur aléatoire `urbcsp` ont une treewidth variable, dont la distribution sur l'ensemble des instances dépend du nombre de contraintes et du nombre de variables.

Les paramètres du générateur sont :

- le nombre  $n$  de variables
- la taille  $d$  des domaines (toutes les variables ont le même domaine)
- la treewidth  $k$  désirée ( $k < n$ )
- la dûreté  $t$  des contraintes (*tightness*,  $0 \leq t \leq d^2$ )

Le générateur produit alors un réseau de contraintes avec  $n$  variables dont le domaine est  $\{0, \dots, d - 1\}$  et tel que le graphe de contraintes associé soit un  $k$ -arbre. Le graphe est initialisé à une clique de taille  $k$ . Des sommets sont ensuite successivement ajoutés et reliés à une clique de taille  $k$  choisie au hasard, jusqu'à obtenir  $n$  sommets au total. Le nombre d'arêtes est alors égal à  $nk - \frac{k(k+1)}{2}$ . Le graphe est enfin traduit en un CSP. Pour chaque arête, une contrainte refusant  $t$  couples est générée. Les  $t$  couples sont pris au hasard, sans répétition dans  $\{0, \dots, d - 1\}^2$ .

La taille des domaines est fixée à 6 et la treewidth à 5. Le nombre de contraintes est alors dépendant du nombre  $n$  de variables. Ce dernier varie de 20 à 180 par paliers de 20. Enfin la dûreté  $t$  des contraintes est fixée à 10. Pour chaque configuration, 20 instances ont été générées.

## 4.2 Résultats

### 4.2.1 Instances aléatoires

La première catégorie fixe  $e$  à 20, les instances ont un graphe de structure creux. C'est-à-dire que le graphe de contraintes associé possède peu de cycles. On peut voir un exemple d'instance dans la figure 4.1.

La transition de phase de CSP est l'intervalle de dûreté  $23 < t < 29$ . Comme on peut le voir sur la figure 4.2, lorsque toutes les instances générées ont au moins une solution ( $t \leq 23$ ), le temps de réponse augmente légèrement, alors que le nombre de solutions croit de manière substantielle quand la dûreté diminue.

Les résultats obtenus s'expliquent par le fait que la treewidth des graphes de contraintes associés aux instances de cette catégorie est bornée par un petit nombre. Si l'on regarde la distribution de la treewidth calculée par MCS sur l'ensemble des instances pour  $n = 20$  et  $e = 20$  (170 instances au total), la treewidth est inférieure à 4 (4 étant rarement atteint).

Cette seconde catégorie fixe  $e$  à 45. Les graphes de contraintes associés aux

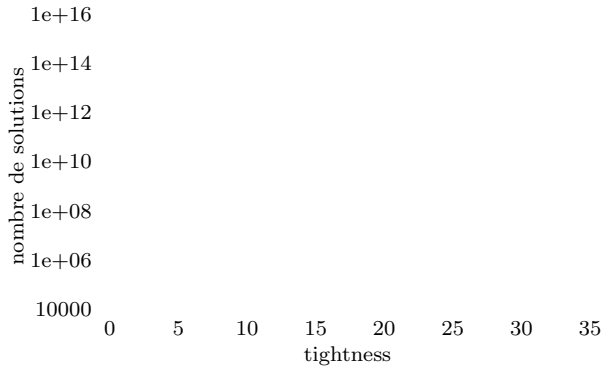


FIG. 4.1 – moyenne du nombre de solutions et exemple d’instance pour  $n = 20$ ,  $d = 6$ ,  $e = 20$

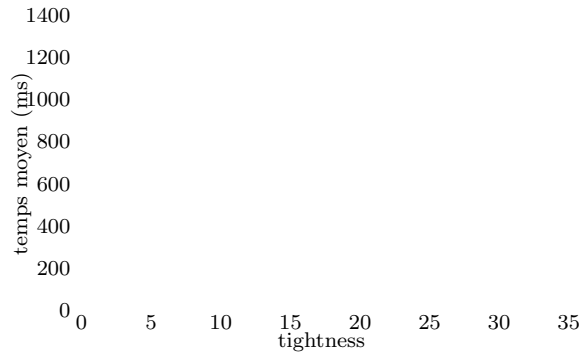


FIG. 4.2 – moyenne des temps de réponse pour  $n = 20$ ,  $d = 6$ ,  $e = 20$

instances sont alors plus complexes, comme on peut le voir dans la figure 4.3.

La transition de phase de CSP se situe dans l’intervalle de d  t    $16 < t < 21$ . L’algorithme r  pond rapidement pour un  $t > 21$ . A la diff  rence de la premi  re cat  gorie, pour un  $t < 16$  et jusqu’    $t = 1$ , on peut voir dans la figure 4.4 que le temps de r  ponse cro  t de mani  re substantielle quand la d  t   diminue. On peut aussi remarquer que le temps de r  ponse commence    augmenter dans la transition de phase CSP, et ce, m  me si l’on n’observe que les r  sultats pour des instances sans solutions.

La forte corr  lation entre la d  t   des contraintes et le temps de r  ponse s’explique par la valeur non born  e de la treewidth des graphes g  n  r  s. En effet pour  $e = 45$ , les graphes de contraintes des instances g  n  r  es ont une treewidth (calcul  e par MCS) allant de 5    11 avec une grande majorit   entre 7 et 9. L’algorithme est exponentiel en la treewidth dans le pire des cas qui correspond    une d  t   nulle. La variation de la d  t   influe sur le nombre de solutions trouv  es pour chaque sous-probl  me. Lorsque la valeur de la d  t   se rapproche

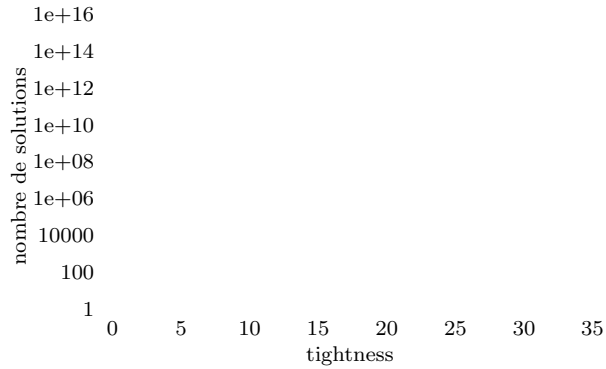


FIG. 4.3 – moyenne du nombre de solutions et exemple d’instance pour  $n = 20$ ,  $d = 6$ ,  $e = 45$

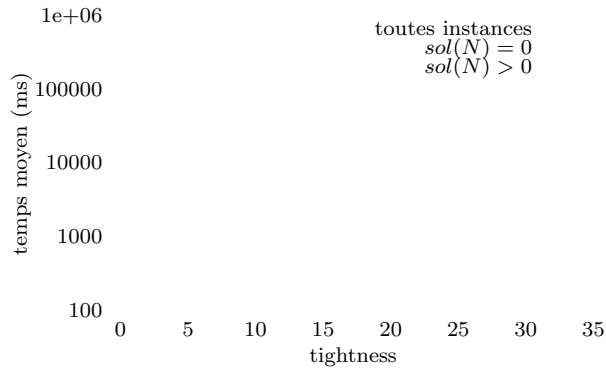


FIG. 4.4 – moyenne des temps de réponse pour  $n = 20$ ,  $d = 6$ ,  $e = 45$

de 0, le temps de réponse augmente donc de manière exponentielle.

Lorsque le nombre de contraintes augmente encore, la structure du graphe de contraintes devient de plus en plus complexe, aussi le problème n’est plus décomposable et le comportement de l’algorithme tend vers l’énumération naïve de toutes les solutions du problème global. Les expérimentations relatives à cette catégorie n’ont abouti que pour des valeurs triviales de  $t$ , essentiellement à cause du manque de mémoire. Aussi nous nous sommes restreint aux problèmes suffisamment décomposables pour pouvoir faire une observation significative du comportement de l’algorithme.

#### 4.2.2 Instances artificiellement structurées

Afin de montrer la corrélation entre le temps de réponse et le nombre de variables, nous avons généré des instances en fixant la taille  $d$  des domaines, la

treewidth  $tw$  du graphe de contraintes associé à l'instance et la dureté  $t$  des contraintes.

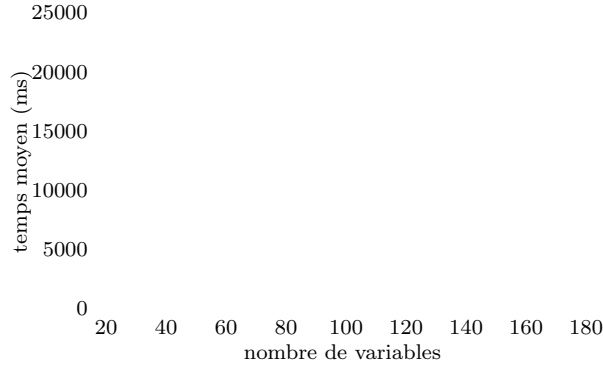


FIG. 4.5 – moyenne des temps de réponse pour  $d = 6$ ,  $t = 10$ ,  $tw = 5$

Les résultats observable sur la figure 4.5 confirment la corrélation polynomiale du temps de réponse au nombre de variables. La dépendance semble même être linéaire.

### 4.3 Instances Structurées

L'algorithme a aussi été testé sur des instances de problèmes structurés à plus de 100 variables. Les résultats sont regroupés dans la table 4.1. La colonne intitulée  $n$  indique le nombre de variables, la colonne  $tw$  la treewidth calculée par MCS et la colonne  $d$  la taille des domaines. Les temps d'exécution de l'algorithme kTreeCount sont présentés dans la colonne intitulée kTC. La dernière colonne indique le temps d'exécution du solveur CHOCO pour déterminer si l'instance est satisfiable. Ces résultats sont présentés à titre indicatif de la difficulté de l'instance pour le problème CSP.

nom	$n$	$tw$	$d$	kTC	CHOCO
renault	101	10	$[2-42]^1$	319s	0.2s
domino-100-100	100	2	100	15.7s	7.6s
domino-100-200	100	2	200	66s	56s
domino-100-300	100	2	300	216s	182s
domino-300-100	300	2	100	55s	23s
domino-300-200	300	2	200	205s	172s
domino-300-300	300	2	300	587s	553s

TAB. 4.1 – temps de réponse sur instances structurées

Les temps de réponse sont proches pour la série d'instance "domino" car ces instances ne possèdent qu'une seule solution, on remarque cependant que cet écart se réduit quand la taille  $d$  des domaines augmente.

## 4.4 Discussion

Les expérimentations montrent que l'algorithme est efficace pour les réseaux dont le graphe de contraintes est un  $k$ -arbre partiel, c'est-à-dire de treewidth bornée par une constante  $k$ , à condition toutefois que cette constante  $k$  ne soit pas trop grande. Dans le cas général, lorsque le nombre de contraintes augmente, le comportement de l'algorithme se rapproche de l'énumération naïve des solutions par un solveur quelconque. Il montre alors une forte dépendance à la dureté des contraintes comme le montre la figure 4.4.

Ce comportement est explicable par le fait que lorsque le nombre de contraintes ajoutées au hasard augmente, la treewidth augmente jusqu'à atteindre  $n - 1$  dans le cas d'un graphe de contraintes complet (tous les arcs sont présents). La décomposition se réduit dans le pire des cas à un sommet singleton de taille  $n$ . Le comportement de l'algorithme est alors exactement l'exploration complète de l'arbre de recherche. Les techniques de résolution du solveur permettent de limiter l'exploration des branches inconsistantes (qui ne mènent pas à une solution). L'arbre de recherche est alors réduit lorsque la dureté des contraintes augmente (le nombre de solutions diminue). La dureté influe donc sur le temps de réponse. L'algorithme est donc plus efficace sur les réseaux possédant des contraintes assez restrictives.

Typiquement, un réseau possédant des contraintes très molles (peu restrictives) est considéré comme une instance facile pour CSP. En effets, la recherche d'une solution abouti vite, puisque peu d'affectations ne sont pas solution. Du point de vue  $\#CSP$ , le grand nombre de solutions n'implique pas forcément un grand espace de recherche. Le nombre de réponses différentes possibles d'un algorithme pour  $\#CSP$  diminue quand la dureté diminue, jusqu'à un dans le cas d'une dureté nulle (la réponse est alors le produit des tailles des domaines). On peut donc penser qu'il est possible d'améliorer l'algorithme `kTreeCount` dans le cas d'instances très sous-contraintes, peut-être en comptant les affectations non-valides des variables. La technique n'a pas été implémentée car il aurait alors été nécessaire de modifier le solveur assez profondément et le temps nécessaire manquait.

Lorsque le nombre de variables  $n$  augmente alors que la treewidth reste fixe (figure 4.5), l'algorithme se montre de plus en plus efficace. Ceci s'explique en par-

---

<sup>1</sup>La taille des domaines varie de 2 à 42.

tie par le fait que si la treewidth est bornée, la densité du graphe diminue quand  $n$  augmente. Les sous-réseaux induits par les sommets de la décomposition sont alors de plus en plus nombreux tout en restant de taille bornée. Les observations laissent à penser qu'une analyse plus fine de la complexité est possible. En effet, la borne calculée dans la Section 3.2.4 n'est pas optimale.

## 5 Conclusion et perspectives

Ce mémoire présente un algorithme de comptage du nombre de solutions d'un réseau de contraintes qui tire parti de la structure du réseau. En effet, le nombre de solutions d'un réseau à structure arborescente peut se calculer en un temps polynomial en la taille du réseau. L'utilisation de la décomposition arborescente nous permet de décomposer le réseau en sous-réseaux connectés, formant une structure d'arbre. L'algorithme présenté a une complexité polynomiale en la taille du réseau et exponentielle en la treewidth du graphe de contraintes. Nous avons validé cette analyse par des expérimentations sur des graphes aléatoires, dans un premier temps, puis sur des graphes, générés aléatoirement, mais dont la structure est contrôlée. Plus précisément, un générateur de réseaux dont la structure est un  $k$ -arbre, développé spécialement pour cette étude, a permis de valider la complexité de l'algorithme.

Des expérimentations plus approfondies, notamment des comparaisons avec les méthodes existantes sont cependant nécessaires pour valider la performance de l'approche présentée.

Une nouvelle notion de décomposition d'hypergraphes est introduite dans [Gottlob *et al.*, 1999b] dans le cadre du traitement de requêtes conjonctives. Cette décomposition généralise la notion de décomposition arborescente. [Gottlob *et al.*, 1999a] présente de façon précise une comparaison des différentes méthodes de décomposition de graphes dans le cadre du problème CSP. Bien que la décomposition en hyperarbre soit équivalente à la décomposition arborescente pour des réseaux de contraintes binaires, dans le cas de contraintes  $n$ -aires, cette dernière peut décomposer le réseau de contraintes plus finement. La résolution du problème est alors plus efficace. On peut adapter l'algorithme `kTreeCount` à cette décomposition, et ainsi améliorer les performances sur les réseaux à contraintes d'arité plus élevée.

Aussi, une étude plus approfondie des propriétés des sous-réseaux pourrait améliorer la méthode, pour l'instant naïve, d'énumération des solutions partielles. En effet les sous-réseaux peuvent être très sous-contraints (prendre par exemple un cycle). On peut envisager de compiler l'information nécessaire à l'énumération des solutions partielles en une structure plus complexe qu'une table de hachage,

par exemple un automate.

L'algorithme nécessite une quantité de mémoire limitée, mais n'est pas incrémental. Dans une approche incrémentale, plusieurs types d'interaction sont possibles. Les plus simples sont celles qui ne modifient pas la décomposition, comme l'affectation de valeur à une variables, ou plus généralement la modification d'un domaine, ou encore la création et/ou modification de contraintes liant des variables présentes dans un même sommet de la décomposition. Ces interactions n'entraînent que des mises à jour locales. Les autres interactions sont plus difficile à mettre en oeuvre, comme l'ajout d'une contrainte entre deux variables de sommets différents de la décomposition. La décomposition doit alors être modifiée et les tables changées en conséquence. Les techniques employées dans le domaine des bases de données peuvent alors être appliquées pour créer les nouvelles tables à partir des anciennes. L'étude de l'approche incrémentale n'a malheureusement pas été approfondie, à cause de la durée limitée du stage.

Enfin, l'adaptation de cette technique pour un autre problème de comptage de la classe  $\#P$  peut être utilisée comme une méthode générique de comptage des solutions d'un problèmes de NP (par exemple  $\#Homomorphisme$  [Díaz *et al.*, ]). Il suffit d'exprimer celui-ci comme un problème de graphes et vérifier que certaines conditions soient remplies (voir [Bodlaender, 1997]). La création d'un algorithme de comptage se résume alors en la définition des méthodes de création des tables et de jonction de solutions partielles.

# Bibliographie

- [Angelsmark *et al.*, 2002] Ola Angelsmark, Peter Jonsson, Svante Linusson, et Johan Thapper. Determining the number of solutions to binary csp instances. In *Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, éditeur Pascal Van Hentenryck, numéro 2470 dans Lecture Notes in Computer Science, pages 327–340. Springer-Verlag, Sep 2002.
- [Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, et Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2) :277–284, 1987.
- [Aspvall *et al.*, 1998] Bengt Aspvall, Andrzej Proskurowski, et Jan Arne Telle. Memory requirements for table computations in partial k-tree algorithms. In *Scandinavian Workshop on Algorithm Theory*, pages 222–233, 1998.
- [Bacchus *et al.*, 1996] Fahiem Bacchus, Adam J. Grove, Joseph Y. Halpern, et Daphne Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87(1-2) :75–143, 1996.
- [Bayardo et Pehoushek, 2000] Roberto J. Bayardo et Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the National Conference in Artificial Intelligence (AAAI/IAAI)*, pages 157–162, 2000.
- [Birnbbaum et Lozinskii, 1999] Elazar Birnbbaum et Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10 :457–477, 1999.
- [Bodlaender, 1993] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11 :1–21, 1993.
- [Bodlaender, 1997] Hans L. Bodlaender. Treewidth : Algorithmic techniques and results. In *Proceedings 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97, Lecture Notes in Computer Science, volume 1295*, éditeurs Igor Privara et Peter Ruzicka, pages 19–36, Berlin, 1997. Springer-Verlag.
- [Dechter et Pearl, 1989] Rina Dechter et Judea Pearl. Tree clustering for constraint networks (research note). *Artificial Intelligence*, 38(3) :353–366, 1989.

- [Díaz *et al.*, ] Josep Díaz, Maria Serna, et Dimitrios M. Thilikos. Counting h-colorings of partial k-trees.
- [Gogate et Dechter, 2004] V. Gogate et R. Dechter. A complete anytime algorithm for treewidth, 2004.
- [Gottlob *et al.*, 1999a] G. Gottlob, N. Leone, et F. Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394–399. Morgan Kaufmann, 1999.
- [Gottlob *et al.*, 1999b] Georg Gottlob, Nicola Leone, et Francesco Scarcello. Hypertree decompositions and tractable queries. In *PODS '99 : Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32, New York, NY, USA, 1999. ACM Press.
- [Gyssens *et al.*, 1994] Marc Gyssens, Peter Jeavons, et David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1) :57–89, 1994.
- [Jégou *et al.*, 2005a] P. Jégou, S. N. Ndiaye, et C. Terrioux. Computing and exploiting tree-decompositions for (max-)csp. Rapport Technique LSIS.RR.2005.005, LSIS, 2005.
- [Jégou *et al.*, 2005b] P. Jégou, S. N. Ndiaye, et C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP-2005)*, volume 3709 de *LNCS*, pages 777–781. Springer, octobre 2005.
- [Kask *et al.*, ] Kalev Kask, Rina Dechter, et Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction.
- [Laburthe, 2000] François Laburthe. Choco : Implémentation du noyau d'un système de contraintes. In *actes des 6<sup>e</sup> Journées Nationales sur la résolution de Problèmes NP-Complets*, juin 2000.
- [Lozinskii, 1992] Eliezer L. Lozinskii. Counting propositional models. *Information Processing Letters*, 41(6) :327–332, 1992.
- [Pesant, 2005] Gilles Pesant. Counting solutions of csps : A structural approach. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 260–265, 2005.
- [Reed, 1992] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *STOC '92 : Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 221–228, New York, NY, USA, 1992. ACM Press.
- [Robertson et Seymour, 1986] Neil Robertson et Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3) :309–322, 1986.

- [Roth, 1996] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2) :273–302, 1996.
- [Russell et Norvig, 2003] Stuart Russell et Peter Norvig. *Artificial Intelligence - A modern approach*, chapitre 5, pages 151–155. Englewood Cliffs, NJ : Prentice-Hall (2d Edition), 2003.
- [Tarjan et Yannakakis, 1984] Robert E. Tarjan et Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3) :566–579, 1984.
- [Telle, ] J. Telle. Tree-decompositions of small pathwidth.
- [Valiant, 1979] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3) :410–421, 1979.
- [Wei et Selman, 2005] Wei Wei et Bart Selman. A new approach to model counting. In *SAT*, pages 324–339, 2005.
- [Weigel et Faltings, 1999] Rainer Weigel et Boi Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2) :257–287, 1999.