Amélioration des méthodes basées sur une décomposition dans le cas du problème #CSP

Sommaire

6.1 Introduction 206
6.2 Similitudes entre les problèmes CSP et #CSP 206
6.2.1 Adaptation de BTD à #BTD 207
6.2.2 Inconvénient de #BTD 210
6.3 Recherche plus adaptée au comptage : #EBTD 210
6.3.1 Comptage aveugle vs comptage conscient 211
6.3.2 Extension du type d'enregistrements 212
6.3.3 Description de l'algorithme #EBTD 214
6.3.3.1 Entrées et Sorties 214
6.3.3.2 Similitudes avec #BTD 214
6.3.3.3 Modifications réalisées pour #EBTD 215
6.3.4 Fondements théoriques 216
6.4 Étude expérimentale 221
6.4.1 Benchmark utilisé 221
6.4.2 Protocole expérimental 221
6.4.3 Observations et analyse des résultats 222
6.5 Conclusion 233

6.1 Introduction

La résolution du problème #CSP est extrêmement difficile sur le plan théorique comme sur le plan pratique. Ainsi, ce problème a été étudié depuis longtemps et reste l'objet de plusieurs travaux pendant les dernières années. D'un point de vue pratique, des méthodes de résolution ont été proposées. Cependant, en raison de la difficulté théorique et pratique de ce problème, la plupart des travaux se sont focalisés sur les méthodes qui se contentent d'approximer le nombre de solutions ou de fournir une borne inférieure sur le nombre de solutions du problème. En effet, il est souvent difficile de résoudre ces instances exactement ou, en d'autres termes, d'obtenir le nombre exact de leurs solutions. Au contraire, en exploitant certaines propriétés des instances, il est possible de proposer des méthodes exactes qui peuvent être efficaces en théorie comme en pratique. Notamment, dans ce chapitre, nous nous intéressons aux méthodes de recherche qui exploitent la structure de l'instance comme BTD. Contrairement à [Favier et al., 2009] qui utilise essentiellement $\#BTD$, l'adaptation de BTD au problème du comptage, telle une sous-routine pour une méthode d'approximation, nous visons dans ce chapitre à améliorer directement cette approche pour le comptage exact. En particulier, $\#BTD$ effectue beaucoup de calculs qui peuvent s'avérer inutiles dans le contexte du comptage. C'est ainsi que pour une affectation partielle donnée, $\#BTD$ peut compter le nombre de ses extensions cohérentes pour un sous-problème sans avoir la garantie que cette affectation possède au moins une extension cohérente sur tout le problème. Le fait que ces calculs inutiles peuvent être coûteux peut conduire à une dégradation de la performance de $\#BTD$. L'objectif du nouvel algorithme appelé $\#EBTD$ est alors de garantir, lors du comptage du nombre d'extensions d'une affectation partielle pour un sous-problème donné, que cette dernière admet au moins une extension cohérente sur tout le problème.

Le plan de ce chapitre est le suivant. Dans la section 6.2, nous évoquons comment les similitudes entre les deux problèmes CSP et #CSP ont permis d'étendre les méthodes de résolution pour le premier aux méthodes de résolution pour le second. Nous nous focalisons dans cette section sur la méthode BTD qui a été adaptée en $\#BTD$ pour le problème #CSP et nous montrons un inconvénient de cette méthode. Ensuite, dans la section 6.3, nous décrivons les modifications apportées à $\#BTD$ et nous expliquons l'algorithme $\#EBTD$. Nous illustrons par la suite son intérêt par une étude expérimentale avant de conclure.

6.2 Similitudes entre les problèmes CSP et #CSP

Trivialement, les deux problèmes CSP pour la décision et #CSP pour le comptage se rapprochent du fait de la nature de la question à laquelle ils répondent. Étant donnée une instance, le problème de décision CSP consiste à *dire si cette instance possède une solution*. Quant au problème du comptage #CSP, il vise à *compter le nombre de solutions de cette instance*. Évidemment, si nous ne sommes pas capables de décider si une instance a une solution, nous ne serons pas en mesure de compter son nombre de solutions. De même, si nous pouvons compter efficacement le nombre de solutions d'une instance, nous répondons automatiquement à la question de la cohérence de l'instance.

La relation étroite entre ces deux problèmes a permis d'étendre les méthodes de résolution du problème CSP naturellement aux méthodes de comptage. Ainsi, les méthodes énumératives classiques de résolution du problème CSP telles que MAC et RFL ont été étendues et exploitées pour le comptage du nombre de solutions d'une instance. Dans ce cas, elles explorent l'espace de recherche afin d'énumérer l'ensemble des solutions du

problème et ne se contentent pas de la détection d'une seule solution. La façon dont ces méthodes procèdent induit une redondance des sous-espaces de recherche visités. Ces redondances sont encore plus pénalisantes dans le cadre du problème du comptage puisque l'effort effectué pour chaque sous-problème est plus important vu que l'espace de recherche visité serait potentiellement plus grand que dans le cas du problème de décision. De ce fait, ces méthodes ne sont pas efficaces en pratique notamment pour les problèmes ayant un très grand nombre de solutions (sur le benchmark auquel nous allons nous intéresser dans ce chapitre certaines instances possèdent un nombre de solutions de l'ordre de 10^{250} solutions). Dans ce cas, la capacité de l'énumération est dépassée.

Quant aux méthodes structurelles, elles ont été adaptées au problème #CSP. Leur intérêt réside dans leur aptitude à fournir des méthodes exactes de comptage qui ont une complexité théorique en temps beaucoup plus intéressante que celle des méthodes énumératives, exponentielle en n. En pratique, ces méthodes ont également prouvé une amélioration importante en termes d'efficacité. Dans cet esprit, dans [Favier et al., 2009], il a été proposé d'adapter la méthode BTD au problème #CSP aboutissant ainsi à une méthode appelée $\#BTD$. $\#BTD$ a recours à l'enregistrement à la façon de BTD ce qui le rend efficace en pratique contrairement aux méthodes classiques. En effet, BTD enregistre pour chaque sous-problème induit par une affectation donnée un (no)good structurel qui indique si cette affectation admet une extension sur ce sous-problème. Quant à $\#BTD$, c'est le nombre d'extensions de cette affectation qui sera enregistré. Si ultérieurement pendant la recherche, cette même affectation est réalisée, $\#BTD$ réutilise le nombre de solutions enregistré pour le sous-problème déjà visité et par voie de conséquence évite certaines redondances.

Nous nous focalisons dans la partie suivante sur $\#BTD$ et nous expliquons comment cet algorithme compte le nombre de solutions des instances CSP grâce à la décomposition arborescente de leur (hyper)graphe de contraintes.

6.2.1 Adaptation de BTD à #BTD

La méthode $\#BTD$ se base sur les mêmes principes que la méthode BTD utilisée pour résoudre le problème CSP. Elle exploite une décomposition calculée préalablement avant le début de la recherche. Dans la suite de ce chapitre, nous nous basons sur l'exemple de la décomposition de la figure 6.1 pour illustrer le comportement de l'algorithme $\#BTD$ et plus tard celui de l'algorithme $\#EBTD$.

$\#BTD$, à l'instar de BTD, exploite la propriété essentielle de la décomposition arborescente. En effet, le fait d'assigner les variables d'un séparateur entre deux clusters de la décomposition sépare le problème initial en deux problèmes, qui peuvent être résolus indépendamment. Si le séparateur en question sépare deux clusters E_i et E_j de la décomposition (avec E_j le cluster fils de E_i), le premier sous-problème est celui enraciné en E_j. Il contient l'ensemble des variables des clusters de $Desc(E_j)$ (noté $V_{Desc(E_j)}$) et est noté $P_j|\mathcal{A}[E_i \cap E_j]$ (ou simplement P_j lorsqu'il n'y a pas d'ambiguïté). En outre, à l'image de BTD, la décomposition induit un ordre de choix de variables partiel comme cela a été décrit dans la sous-section 4.2.2. En particulier, si le cluster racine $E_r = E_g$ dans la figure 6.1, l'ordre partiel de choix de variables est : $\Lambda = [\{x_1, x_4, x_5\}, \{\{x_2, x_3\}, [\{x_8, x_9\},$ $\{\{x_6, x_7\}, [\{x_{10}, x_{11}\}, \{x_{12}, x_{13}, x_{14}\}], [\{x_{15}, x_{16}\}, \{x_{17}, x_{18}, x_{19}\}]\}]\}]$. Lorsqu'un cluster E_i est totalement assigné, pour chaque cluster E_j dans $Fils(E_i)$, le sous-problème $P_j|\mathcal{A}[E_i \cap E_j]$ est résolu indépendamment. Comme dans le cas du problème de décision où BTD enregistre le résultat de l'exploration du problème $P_j|\mathcal{A}[E_i \cap E_j]$ (enregistrement de *goods* et de *nogoods*), $\#BTD$ évite certaines redondances en enregistrant également les informations nécessaires. En particulier, $\#BTD$ enregistre le nombre exact de solutions de

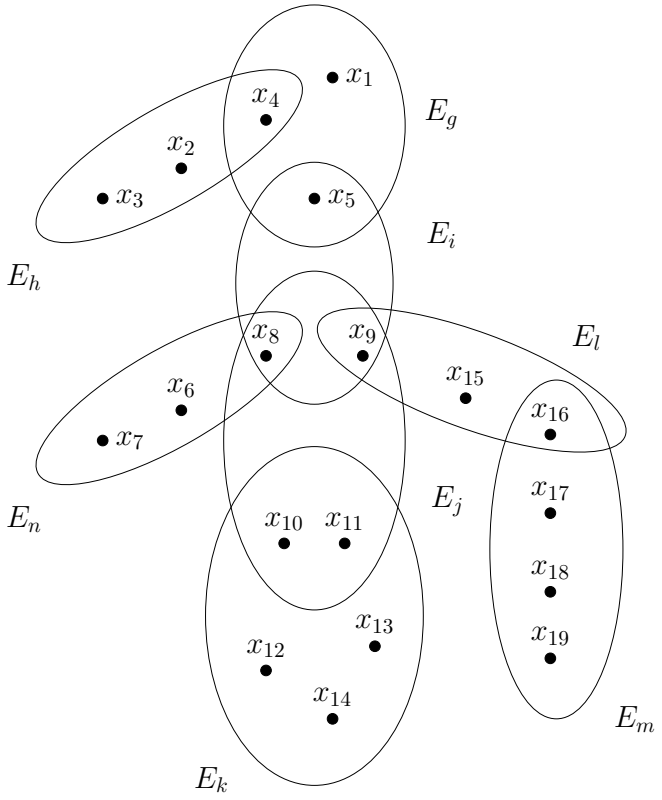

FIGURE 6.1 – L'ensemble des clusters d'une décomposition.

$P_j|\mathcal{A}[E_i \cap E_j]$, noté $\#sol_{E_j}$, comme un *#good structurel*($\mathcal{A}$,$\#sol_{E_j}$). Ce faisant, le nombre de solutions ne sera jamais recalculé pour la même affectation $E_i \cap E_j$. C'est ainsi que $\#BTD$, comme BTD, est capable de maintenir une complexité exponentielle en fonction de la taille du plus grand cluster de la décomposition.

$\#BTD$ est décrit dans l'algorithme 6.1. Étant donnés une affectation $\mathcal{A}$ et un cluster E_i, $\#BTD$ cherche le nombre d'extensions cohérentes $\mathcal{B}$ de $\mathcal{A}$ sur $Desc(E_i)$ telles que $\mathcal{A}[E_i \backslash V_{E_i}] = \mathcal{B}[E_i \backslash V_{E_i}]$. Le premier appel réalisé est $\#BTD(P, (E, T), \emptyset, E_r, E_r, \emptyset)$ et retourne le nombre de solutions du problème global. $\#BTD$ commence par assigner les variables du cluster racine. Si, dans l'exemple de la figure 6.1, $E_r = E_g$, les premières variables à assigner sont les variables x_1, x_4 et x_5. Au sein de chaque cluster à affecter, $\#BTD$ procède classiquement en assignant une valeur à une variable et en faisant un retour-arrière si la recherche rencontre une incohérence (lignes 15-22). Des algorithmes tels que $\#BT$, $\#MAC$ ou $\#RFL$ peuvent être utilisés. La présentation est basée sur $\#BT$. Soit E_i le cluster courant. Une fois toutes les variables de E_i instanciées de façon cohérente (ligne 1), $\#BTD$ calcule le nombre de solutions du sous-problème induit par chaque cluster fils de E_i, s'il en a (lignes 2-12). Dans le cas de la figure 6.1, E_i possède 3 fils : E_n, E_j et E_l. Considérons par exemple le cluster fils E_j. Étant donnée une affectation courante $\mathcal{A}$ de E_i, $\#BTD$ vérifie d'abord si l'affectation $\mathcal{A}[E_i \cap E_j]$ correspond à un #good (ligne 7). Si $\mathcal{A}[E_i \cap E_j]$ est effectivement un #good, $\#BTD$ multiplie le nombre de solutions enregistré avec le nombre de solutions de E_i avec $\mathcal{A}$ comme affectation (ligne 8). Sinon, $\#BTD$ étend $\mathcal{A}$ sur les variables restantes de $Desc(E_j)$ dans le but de calculer le nombre d'extensions cohérentes $\#sol_{E_j}$ (ligne 10). Les variables impliquées dans le cas de la figure 6.1 sont x_{10}, x_{11}, x_{12}, x_{13} et x_{14}. Ensuite, il enregistre le #good $(A[E_i \cap E_j], \#sol_{E_j})$ (ligne 11). Après, $\#BTD$ calcule le nombre de solutions de chaque sous-problème induit par le

Algorithme 6.1 : #BTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, G^d)

Entrées : Une instance CSP $P = (X, D, C)$, (E,T) la décomposition arborescente, l'affectation courante $\mathcal{A}$, le cluster courant E_i, l'ensemble V_{E_i} des variables non assignées de E_i
Entrées-Sorties : L'ensemble G^d de #goods enregistrés
Sorties : Le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$

```
1  si V_{E_i} = ∅ alors
2      Q_{E_i} ← Fils(E_i)
3      #sol ← 1
4      tant que Q_{E_i} ≠ ∅ et #sol ≠ 0 faire
5          Choisir un cluster E_j ∈ Q_{E_i}
6          Q_{E_i} ← Q_{E_i} \ {E_j}
7          si (A[E_i ∩ E_j], #sol_{E_j}) est un #good dans G^d alors
8              #sol ← #sol × #sol_{E_j}
9          sinon
10             #sol_{E_j} ← #BTD(P, (E,T), A,E_j, V_{E_j} \ (E_i ∩ E_j),G^d)
11             Enregistrer (A[E_i ∩ E_j], #sol_{E_j}) comme #good de E_i par rapport à E_j
                dans G^d
12             #sol ← #sol × #sol_{E_j}
13     retourner #sol
14 sinon
15     Choisir x ∈ V_{E_i}
16     d ← D_x
17     #sol_x ← 0
18     tant que d ≠ ∅ faire
19         Choisir v ∈ d
20         d ← d − {v}
21         si A ∪ {x ← v} satisfait toutes les contraintes de C alors
22             #sol_x ← #sol_x+#BTD(P, (E,T), A ∪ {x ← v}, E_i, V_{E_i} \ {x},G^d)
23     retourner #sol_x
```

prochain fils de E_i (les clusters E_l et E_n dans notre exemple). Finalement, lorsque chaque cluster fils de E_i est examiné, $\#BTD$ essaye de modifier l'affectation courante de E_i. Le nombre de solutions au niveau de E_i est la somme du nombre de solutions pour chaque affectation cohérente de E_i.

Les complexités en temps et en espace de $\#BTD$ sont les mêmes que pour BTD, à savoir $O(n.w^+.log(d).d^{w^+ +1})$ et $O(n.s.d^s)$ respectivement [Favier et al., 2009].

$\#BTD$ a permis d'exploiter la structure de l'instance et de bénéficier des enregistrements réalisés au niveau des séparateurs pour rendre les méthodes de comptage plus efficaces. Cependant, dans [Favier et al., 2009], les expérimentations ont montré qu'en pratique, $\#BTD$ dépasse souvent la limite du temps ou de mémoire. En effet, la manière dont $\#BTD$ procède pour calculer le nombre de solutions de P présente un défaut majeur que nous détaillons dans la partie suivante.

6.2.2 Inconvénient de #BTD

La façon dont $\#BTD$ procède pour calculer le nombre de solutions d'un problème peut effectivement engendrer des calculs coûteux et inutiles. L'idée de base est qu'étant donnée une affectation partielle $\mathcal{A}$, un appel à $\#BTD$, comme celui de $\#sol_{E_j} \leftarrow \#\text{BTD}(P, (E,T), A, E_j, V_{E_j} \backslash (E_i \cap E_j))$ (ligne 10), compte toutes les solutions du sous-problème courant. Pour autant BTD n'a pas la garantie que l'affectation partielle s'étend à une solution sur tout le problème.

Nous illustrons cet inconvénient de $\#BTD$ sur l'exemple de la figure 6.1. Soit $E_r = E_g$ le cluster racine de la décomposition par lequel $\#BTD$ débute le comptage. Supposons que E_i est le cluster courant et qu'il vient d'être instancié d'une façon cohérente (E_g et E_h sont déjà instanciés). $\#BTD$ traite alors les clusters fils de E_i l'un après l'autre en calculant le nombre exact de solutions correspondant à chaque sous-problème enraciné en chaque cluster fils. Il considère, par exemple, le cluster E_j et calcule le nombre exact de solutions du problème $P_j|\mathcal{A}[E_i \cap E_j]$, puis considère le cluster E_l et calcule celui de $P_l|\mathcal{A}[E_i \cap E_l]$ et enfin considère le cluster E_n afin de calculer celui de $P_n|\mathcal{A}[E_i \cap E_n]$. L'inconvénient d'une telle approche est que toutes les solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ sont calculées avant de s'assurer qu'il existe au moins une solution pour $P_l|\mathcal{A}[E_i \cap E_l]$ et pour $P_n|\mathcal{A}[E_i \cap E_n]$. Plus précisément, le nombre de solutions d'un sous-problème donné est calculé avant de vérifier que l'affectation courante peut s'étendre en une solution globale, c'est-à-dire une affectation cohérente de toutes les variables du problème. En effet, dans le pire des cas, $\#BTD$ pourrait calculer le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ et de $P_l|\mathcal{A}[E_i \cap E_l]$ avant d'établir que $P_n|\mathcal{A}[E_i \cap E_n]$ ne possède aucune solution. Dans ce cas, le nombre de solutions calculé et enregistré pour les sous-problèmes $P_j|\mathcal{A}[E_i \cap E_j]$ et $P_l|\mathcal{A}[E_i \cap E_l]$ s'avère inutile (sauf si le #good enregistré pour un sous-problème est utilisé ultérieurement). La situation devient de plus en plus pénalisante pour $\#BTD$ lorsque le nombre de sous-problèmes examinés avant l'exploration du sous-problème pour lequel il n'existe aucune extension cohérente augmente. Pour résumer, $\#BTD$ n'exploite pas le fait qu'il est inutile de compter le nombre d'extensions d'une affectation sur un sous-problème si cette affectation ne s'étend pas en une solution globale, c'est-à-dire une solution du point de vue du problème de décision. Ce principe s'applique aussi localement pour un sous-problème en particulier. C'est ainsi que pour compter le nombre d'extensions cohérentes d'une affectation donnée de E_i sur $Desc(E_i)$, il est inutile de compter le nombre d'extensions cohérentes de cette affectation pour un fils donné de E_i si elle n'admet pas au moins une extension sur toutes les variables de $Desc(E_i)$.

En conséquence, cet inconvénient peut détériorer l'efficacité de $\#BTD$ vu que des sous-espaces de recherche sont explorés inutilement ce qui consomme du temps mais aussi de l'espace mémoire en enregistrant des informations qui ne seront éventuellement pas réutilisées par la suite. Ainsi, une première possibilité pour améliorer $\#BTD$ consiste à éviter ces recherches inutiles. Dans la section suivante, nous exploitons cette idée dans le but de définir un algorithme plus sophistiqué que nous appellerons $\#EBTD$. Ce travail a fait l'objet de la publication [Jégou et al., 2016a].

6.3 Recherche plus adaptée au comptage : #EBTD

Dans cette section, nous détaillons l'algorithme $\#EBTD$ et nous montrons comment il est capable d'améliorer la recherche faite par $\#BTD$. Notons que si l'idée sur laquelle se base $\#BTD$ est simple et naturelle, sa mise en œuvre est complexe.

6.3.1 Comptage aveugle vs comptage conscient

Tout d'abord, nous illustrons l'objectif de $\#EBTD$.

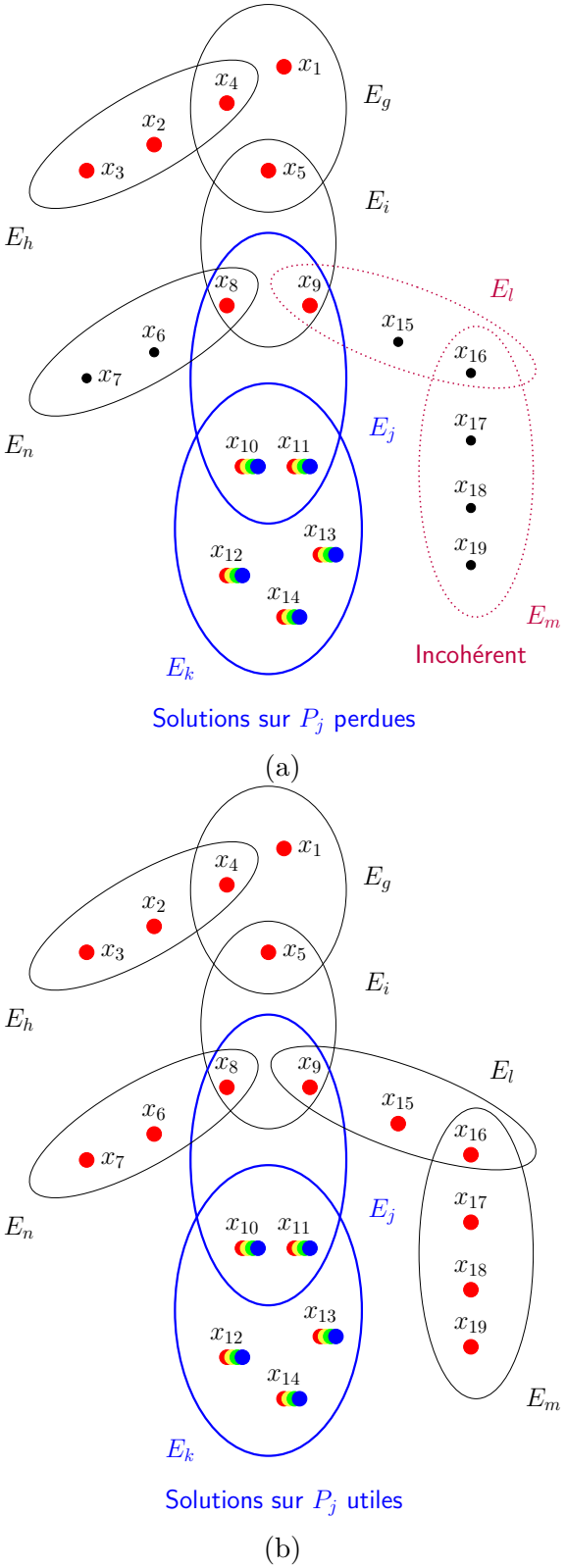

FIGURE 6.2 – Illustration du comptage aveugle (a) et du comptage conscient (b).

Dans la figure 6.2, les variables affectées sont colorées en rouge. Les clusters E_j et E_k du sous-problème P_j pour lequel nous avons compté le nombre d'extensions cohérentes de l'affectation du séparateur $E_i \cap E_j$ sont représentés en bleu. Les différentes solutions sont colorées en : rouge, jaune, vert et bleu. Nous distinguons deux cas de figure :

- Pour la figure 6.2(a), après l'affectation des variables x_1, x_2, x_3, x_4, x_5, x_8 et x_9, nous comptons le nombre d'extensions cohérentes de $\mathcal{A}[\{x_8, x_9\}]$ pour P_j. C'est ce que nous appelons le *comptage aveugle*.
- En ce qui concerne la figure 6.2(b), le comptage du nombre d'extensions cohérentes de $\mathcal{A}[\{x_8, x_9\}]$ pour P_j n'est réalisé qu'après l'affectation de toutes les variables du problème de façon cohérente. Nous parlons ici du *comptage conscient*.

La différence entre les deux cas est que dans la figure 6.2(a), le calcul de $\#sol_{E_j}$ est fait sans que la présence d'une solution globale ne soit garantie. En revanche, dans la figure 6.2(b), le calcul de $\#sol_{E_j}$ n'est réalisé que lorsque l'affectation $\mathcal{A}[\{x_8, x_9\}]$ est garantie extensible de façon cohérente sur le reste des variables du problème. Dans la figure 6.2(a), une fois le nombre de solutions calculé sur P_j, une incohérence est détectée pour P_l. En effet, aucune extension cohérente n'est trouvée sur ce sous-problème. Ainsi, le nombre de solutions trouvé pour P_j s'avère inutile. Cependant, le nombre de solutions trouvé sur P_j dans la figure 6.2(b) est utile et participe au calcul du nombre de solutions de P.

L'objectif de $\#EBTD$ est d'imiter le cas de la figure 6.2(b) et de *garantir lors du comptage du nombre d'extensions d'une affectation sur un sous-problème que cette même affectation est extensible de façon cohérente sur les variables du reste du problème.*

6.3.2 Extension du type d'enregistrements

À l'instar de $\#BTD$, $\#EBTD$ (pour *Enhanced Backtracking with Tree-Decomposition*), est basé sur la notion de la décomposition arborescente des graphes. $\#EBTD$ accomplit aussi une recherche basée sur le retour-arrière en utilisant un ordre de variables compatible avec la décomposition. La première différence avec $\#BTD$ réside dans la définition du concept de goods qui est étendu ainsi :

Définition 64 (Goods structurels exacts et partiels) *Soient (E, T) une décomposition arborescente, E_i et E_j deux clusters de E avec E_j un cluster fils de E_i et $\mathcal{A}$ une affectation cohérente de $E_i \cap E_j$. Un* good structurel exact *est un triplet ($\mathcal{A}$,=,$\#sol_{E_j}$) avec $\#sol_{E_j}$ le nombre exact de solutions de $P_j|\mathcal{A}$. Un* good structurel partiel *est un triplet ($\mathcal{A}$,$\geq$,$\#sol_{E_j}$) avec $\#sol_{E_j}$ une borne inférieure sur le nombre de solutions de $P_j|\mathcal{A}$.*

Notons que les goods exacts structurels ($\mathcal{A}$,=,$\#sol_{E_j}$) sont identiques aux #good($\mathcal{A}$,$\#sol_{E_j}$) structurels exploités dans $\#BTD$ [Favier et al., 2009]. En outre, nous exploitons aussi la notion du nogood structurel :

Définition 65 (nogood structurel [Jégou and Terrioux, 2003]) *Soient (E, T) une décomposition arborescente et E_i et E_j deux clusters de E tel que E_j est un cluster fils de E_i. Un* nogood structurel *de E_i par rapport à E_j est une affectation cohérente $\mathcal{A}$ des variables de $E_i \cap E_j$ de sorte que $P_j|\mathcal{A}$ n'admet aucune solution.*

Bien qu'un nogood structurel soit équivalent à un good exact tel que le nombre de solutions attribué est nul, l'exploitation d'un nogood s'avère plus efficace. En effet, un nogood est exploité *dès que possible* et permet de détecter une incohérence au plus tôt rendant la résolution plus efficace.

Algorithme 6.2 : #EBTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, Z, G^d, N^d)

Entrées : Une instance CSP $P = (X, D, C)$, une décomposition arborescente (E, T), l'affectation courante $\mathcal{A}$, le cluster courant E_i, l'ensemble V_{E_i} des variables non instanciées de E_i

Entrées-Sorties : Z une pile de clusters, l'ensemble G^d de goods enregistrés, l'ensemble N^d de nogoods enregistrés

Sorties : (Le nombre de solutions trouvées pour $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$, le cluster vers lequel nous faisons un retour-arrière)

```
si V_{E_i} = ∅ alors
    #sol ← 1
    Q_{E_i} ← Fils(E_i)
    Z_{good≥} ← ∅
    Z_{inconnu} ← ∅
    tant que Q_{E_i} ≠ ∅ faire
        Choisir un cluster E_j ∈ Q_{E_i}
        Q_{E_i} ← Q_{E_i}\{E_j}
        suivant A[E_i ∩ E_j] faire
            cas où good (A[E_i ∩ E_j], =, #sol_{E_j}) de E_i vis-à-vis de E_j dans G^d faire
                #sol ← #sol * #sol_{E_j}
            cas où good (A[E_i ∩ E_j], ≥, #sol_{E_j}) de E_i vis-à-vis de E_j dans G^d faire
                Z_{good≥} ← Z_{good≥} ∪ {E_j}
            autres cas faire
                Z_{inconnu} ← Z_{inconnu} ∪ {E_j}
                Z ← Z ∪ {E_j}
    si Z ≠ ∅ alors
        E_j ← Premier(Z)
        Z ← Z\{E_j}
        (#sol_{E_j}, E_bt) ← #EBTD(P,(E,T),A,E_j,E_j\(E_{p(j)} ∩ E_j), Z,G^d,N^d)
        si #sol_{E_j} > 0 alors
            si E_bt = E_j alors
                Enregistrer (A[E_{p(j)} ∩ E_j], =, #sol_{E_j}) comme good de E_{p(j)} vis-à-vis de E_j dans G^d
            sinon
                Enregistrer (A[E_{p(j)} ∩ E_j], ≥, #sol_{E_j}) comme good de E_{p(j)} vis-à-vis de E_j dans G^d
                si E_i = E_bt alors
                    Z ← Z\Fils(E_i)
                    retourner (0, E_i)
                sinon retourner (#sol, E_bt)
        sinon
            Enregistrer A[E_{p(j)} ∩ E_j] comme nogood de E_{p(j)} vis-à-vis de E_j dans N^d
            si E_i = E_{p(j)} alors
                Z ← Z\Fils(E_i)
                retourner (0, E_i)
            sinon retourner (#sol, E_{p(j)})
    pour chaque E_j ∈ Z_{inconnu} faire
        #sol ← #sol * #sol_{E_j}
    tant que Z_{good≥} ≠ ∅ faire
        Choisir un cluster E_j ∈ Z_{good≥}
        Z_{good≥} ← Z_{good≥}\{E_j}
        (#sol_{E_j}, E_bt) ← #EBTD(P,(E,T),A,E_j,E_j\(E_i ∩ E_j), Z,G^d,N^d)
        Enregistrer (A[E_i ∩ E_j], =, #sol_{E_j}) comme good de E_i vis-à-vis de E_j dans G^d
        #sol ← #sol * #sol_{E_j}
    retourner (#sol, E_i)
sinon
    Choisir x ∈ V_{E_i}
    d ← D_x
    #sol_x ← 0
    E_bt ← E_i
    répéter
        Choisir v ∈ d
        d ← d − {v}
        si A ∪ {x ← v} satisfait toutes les contraintes de C ∪ N^d alors
            (#sol_{xv}, E_bt) ← #EBTD(P,(E,T),A ∪ {x ← v},E_i, V_{E_i}\{x},Z,G^d,N^d)
            #sol_x ← #sol_x + #sol_{xv}
    jusqu'à d = ∅ ou x ∉ E_bt
    retourner (#sol_x,E_bt)
```

6.3.3 Description de l'algorithme #EBTD

6.3.3.1 Entrées et Sorties

$\#EBTD$ est décrit dans l'algorithme 6.2. Étant données une instance CSP $P = (X, D, C)$ et une décomposition arborescente (E, T), l'appel $\#EBTD$ (P, (E, T), $\mathcal{A}$, E_i, V_{E_i}, Z, G^d, N^d) vise à calculer le nombre de solutions du sous-problème $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ avec $\mathcal{A}$ l'affectation courante, E_i le cluster courant et V_{E_i} l'ensemble de variables non assignées de E_i. G^d et N^d représentent respectivement l'ensemble de goods (exacts et partiels) et de nogoods structurels qui sont enregistrés durant la recherche. L'algorithme exploite aussi une pile Z qui contient les prochains clusters à traiter. $\#EBTD$ retourne une paire $(\#sol, E_{bt})$. E_{bt} représente le cluster vers lequel $\#EBTD$ fait un retour-arrière ce qui permet de réaliser un saut au cluster pertinent lorsqu'un nogood est trouvé. Concernant le nombre de solutions $\#sol$, nous distinguons trois cas possibles :

- Si $\#sol > 0$ et $E_{bt} = E_i$, alors $\#sol$ est le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Ce cas se produit lorsque l'affectation $\mathcal{A}[E_i \backslash V_{E_i}]$ possède au moins une extension cohérente sur tout le problème.

- Si $\#sol > 0$ et $E_{bt} \neq E_i$, alors $\#sol$ est une borne inférieure du nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Ce cas se produit lorsque l'affectation $\mathcal{A}[E_i \backslash V_{E_i}]$ admet au moins une extension cohérente sur P_i alors qu'il existe un cluster E_k tel que $\mathcal{A}[E_k \cap E_{p(k)}]$ est un nogood.

- Si $\#sol = 0$ alors $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ ne possède aucune solution. Ainsi, $\mathcal{A}[E_i \backslash V_{E_i}]$ ne possède aucune extension cohérente sur P_i.

La pile Z résultante peut être définie comme étant l'ensemble de clusters E_j tels que $E_{p(j)}$ est totalement instancié de façon cohérente et tel que $\mathcal{A}[E_{p(j)} \cap E_j]$ est de nature inconnue (i.e. ne correspond ni à un good exact ni à un good partiel de $E_{p(j)}$ par rapport à E_j). Notons qu'elle est nécessairement vide dans le cas où $\#sol > 0$ et $E_{bt} = E_i$. En effet, d'après la spécification de $\#EBTD$, ce dernier ne calcule le nombre exact de solutions d'un sous-problème induit par une affectation $\mathcal{A}$ que lorsque cette affectation s'étend de façon cohérente au reste du problème comme dans le cas de la figure 6.2(b). D'après la définition de la pile Z, cette dernière contient les clusters E_k tel que $\mathcal{A}[E_k \cap E_{p(k)}]$ ne correspond pas à un good. Or, comme pour chaque cluster E_k, l'affectation $\mathcal{A}[E_k \cap E_{p(k)}]$ est garantie extensible de façon cohérente sur $Desc(E_k)$, aucun cluster ne serait inclus dans Z.

6.3.3.2 Similitudes avec #BTD

L'appel initial est $\#EBTD(P, (E, T), \mathcal{A}, E_r, E_r, Z, G^d, N^d)$ où Z, G^d et N^d sont vides. À la façon de $\#BTD$, $\#EBTD$, démarre sa recherche en assignant d'une manière cohérente les variables du cluster racine. Nous considérons par la suite la décomposition de la figure 6.1 avec $E_r = E_g$. Les premières variables à assigner sont alors x_1, x_4 et x_5. Par la suite, $\#EBTD$ explore les clusters fils du cluster courant. Lorsque $\#EBTD$ explore un nouveau cluster E_i, vu que les variables de son cluster parent $E_{p(i)}$ sont déjà instanciées (et ainsi celles de son séparateur), il doit uniquement instancier les variables qui apparaissent dans $E_i \backslash (E_i \cap E_{p(i)})$. Par exemple, si nous considérons le cluster E_i de la figure 6.1, vu que son instanciation se fait après celle de E_g, les seules variables restant à affecter sont x_8 et x_9. Dans le but de résoudre chaque cluster (lignes 46-57), $\#EBTD$ (comme $\#BTD$) peut exploiter n'importe quel algorithme qui n'altère pas la structure. Par souci de simplicité,

la version présentée dans l'algorithme 6.2 est basée sur un simple algorithme de retour-arrière chronologique (BT). Toutefois, dans les expérimentations fournies dans la section suivante, nous considérons un algorithme plus puissant qui est RFL. Ainsi, $\#EBTD$ choisit en premier la prochaine variable x à assigner parmi les variables de V_{E_i} (ligne 46). Après, il sélectionne une valeur v de D_x selon l'heuristique de choix de valeur (lignes 51-52) et l'assigne à la variable x. Si la nouvelle affectation est cohérente vis-à-vis des contraintes initiales de C et des nogoods structurels de N^d (ligne 53), la recherche continue en choisissant une nouvelle variable et en effectuant une nouvelle affectation. Sinon, $\#EBTD$ essaye d'assigner à x une autre valeur de D_x. Lorsque toutes les valeurs possibles sont essayées, $\#EBTD$ fait un retour-arrière.

Lorsque $\#EBTD$ a instancié d'une façon cohérente toutes les variables du cluster courant et du moment où il est certain que cette affectation admet une extension cohérente sur tout le problème, il vise ensuite à compter le nombre de solutions de chaque sous-problème enraciné en chaque cluster fils du cluster courant comme ferait $\#BTD$. Plus précisément, si E_i est, par exemple, le cluster courant dans la figure 6.1 (E_g et E_h sont déjà instanciés), son but est de calculer le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$, de $P_l|\mathcal{A}[E_i \cap E_l]$ et de $P_n|\mathcal{A}[E_i \cap E_n]$ vu que E_i possède dans ce cas trois fils E_j, E_l et E_n. Lorsque le nombre de solutions exact de chaque sous-problème enraciné en un cluster fils est calculé $\#EBTD$ enregistre un good exact (ligne 23). Ainsi, dans le cas de la figure 6.1, $\#EBTD$ enregistre les goods exacts $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$, $(\mathcal{A}[E_i \cap E_l], =, \#sol_{E_l})$ et $(\mathcal{A}[E_i \cap E_n], =, \#sol_{E_n})$. Similairement, $\#BTD$ enregistrerait les #goods correspondants à chaque sous-problème (ligne 11), c'est-à-dire les #good $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$, $(\mathcal{A}[E_i \cap E_l], \#sol_{E_l})$ et $(\mathcal{A}[E_i \cap E_n], \#sol_{E_n})$.

6.3.3.3 Modifications réalisées pour #EBTD

La différence principale entre $\#EBTD$ et $\#BTD$ réside dans l'exploration des clusters fils. Si dans l'exemple $\#BTD$ calcule consécutivement le nombre de solutions exact de $P_j|\mathcal{A}[E_i \cap E_j]$, $P_l|\mathcal{A}[E_i \cap E_l]$ et de $P_n|\mathcal{A}[E_i \cap E_n]$ (s'il visite les clusters fils dans l'ordre E_j, E_n puis E_l), $\#EBTD$ ne procède pas ainsi. Il vise au contraire à éviter les inconvénients de cette approche décrits dans la partie 6.2.2. $\#EBTD$ explore alors les clusters fils différemment. Plus précisément, il vise à garantir qu'il ne compte exactement le nombre de solutions du sous-problème enraciné en un cluster fils que s'il existe une solution globale du problème compatible avec l'affectation courante, c'est-à-dire une extension de l'affectation courante qui s'étend sur toutes les variables du problème. Ce faisant, le good exact ainsi enregistré est nécessairement utilisé au moins une fois. Pour illustrer ce principe, dans la figure 6.1, après l'affectation du cluster E_i, $\#EBTD$ essaye d'instancier, par ordre, les clusters E_j, E_k, E_l, E_m et E_n ($\#EBTD$ explore, dans ce cas, les clusters fils de E_i dans l'ordre E_j, E_l et E_n). Une fois tous les clusters instanciés, nous savons qu'une solution globale existe. C'est à ce moment que $\#EBTD$ calcule, $\#sol_{E_n}$, $\#sol_{E_l}$ et finalement $\#sol_{E_j}$. Au contraire, si après l'affectation des variables de $Desc(E_j)$ et de $Desc(E_l)$, celles de $Desc(E_n)$ ne peuvent pas être instanciées d'une façon cohérente, l'affectation $\mathcal{A}[E_i \cap E_n]$ est enregistrée comme un nogood de E_i par rapport à E_n tandis que $\mathcal{A}[E_i \cap E_l]$ et $\mathcal{A}[E_i \cap E_j]$ sont enregistrées comme des goods partiels (respectivement $(\mathcal{A}[E_i \cap E_l], \geq, 1)$ de E_i par rapport à E_l et $(\mathcal{A}[E_i \cap E_j], \geq, 1)$ de E_i par rapport à E_j). Le nombre de solutions associé à ces goods partiels est 1 puisqu'une seule solution a été trouvée pour chaque sous-problème. Ces (no)goods structurels peuvent être utilisés ultérieurement afin d'éviter des parties redondantes de l'arbre de recherche Ce principe est appliqué récursivement à tous les niveaux des clusters lors du calcul du nombre de solutions exact d'un sous-problème. Par exemple, si $E_r = E_g$ est le cluster courant qui vient d'être instancié, le nombre de

solutions exact de $P_i|\mathcal{A}[E_g \cap E_i]$ ne peut être calculé avant de s'être assuré qu'il existe au moins une extension cohérente pour $P_h|\mathcal{A}[E_g \cap E_h]$ et vice versa.

La condition $x \notin E_{bt}$ (ligne 56) suspend l'énumération des solutions relatives à E_i lorsqu'un nogood de $E_{p(\ell)}$ par rapport à E_ℓ est enregistré avec E_ℓ est un cluster exploré après E_i. Dans ce cas, $E_{bt} = E_{p(\ell)}$ et la recherche fait retour-arrière vers le cluster $E_{p(\ell)}$. Par exemple, dans la figure 6.1, si $\#EBTD$ assigne E_g puis les clusters de $Desc(E_i)$ mais ne réussit pas à étendre l'affectation pour les variables de $Desc(E_h)$, $\mathcal{A}[\{x_4\}]$ est enregistrée comme un nogood de E_g par rapport à E_h. Le test à la ligne 56 bloque ensuite l'énumération des extensions de $\mathcal{A}[\{x_5\}]$ sur E_i puisque dans ce cas $E_{bt} = E_g$ et ainsi le test échoue. Lorsque toutes les variables de E_i sont instanciées d'une façon cohérente (ligne 1), $\#EBTD$ considère chaque cluster fils de E_i (lignes 2-16). Si $\mathcal{A}[E_i \cap E_j]$ correspond à un exact good (ligne 10), le good est exploité et la recherche passe au prochain cluster fils. $\#EBTD$ utilise deux piles locales $Z_{inconnu}$ et $Z_{good_\geq}$. Il ajoute à Z et à $Z_{inconnu}$ chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ ne correspond pas ni à un good, ni à un nogood de E_i par rapport à E_j. Au contraire, si $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j, E_j est ajouté à $Z_{good_\geq}$. $Z_{good_\geq}$ est l'ensemble de clusters fils E_j de E_i pour lesquels $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j avec une borne inférieure sur le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$. Pour chaque cluster de $Z_{good_\geq}$, le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ est calculé ultérieurement (lignes 38-43) uniquement si cela est nécessaire, c'est-à-dire si l'affectation courante $\mathcal{A}$ a pu être étendue à une solution globale. En plus, pour chaque cluster de $Z_{inconnu}$, à la ligne 36, le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ est garanti être calculé (lignes 17-35) et peut ainsi être exploité. Si la pile Z n'est pas vide (ligne 17), $\#EBTD$ continue la recherche sur le cluster E_j, premier élément de la pile Z (ligne 20). Si $\#EBTD$ réussit à étendre $\mathcal{A}$ au sous-problème enraciné en E_j, il enregistre $\mathcal{A}[E_{p(j)} \cap E_j]$ comme un good. Si $\mathcal{A}$ peut être étendue à une solution globale ($E_{bt} = E_j$), le nombre associé à ce good est le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ et nous disposons d'un good exact (ligne 22-23). Sinon, il correspond à une borne inférieure du nombre de solutions et nous parlons d'un good partiel (lignes 25-29). Au contraire, si $\mathcal{A}$ n'a pas pu être étendue au sous-problème enraciné en E_j, $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un nogood (lignes 31-35). Il est à noter que $E_{p(j)}$ n'est pas forcément E_i. Les lignes 27 et 33 permettent finalement de supprimer de la pile Z les fils du cluster E_{bt} (s'il y en a) en raison de l'enregistrement d'un nogood de E_{bt} par rapport à un de ses fils.

6.3.4 Fondements théoriques

Concernant $\#EBTD$, nous distinguons deux phases qui s'alternent :

- La première phase vise à s'assurer que pour une solution partielle, il existe une solution globale, c'est-à-dire une affectation qui s'étend de façon cohérente sur toutes les variables du problème.
- La deuxième consiste à énumérer, une fois la présence d'une solution globale garantie, l'ensemble des extensions cohérentes d'une affectation induisant un sous-problème pour ce dernier.

Ces deux phases garantissent que, chaque fois que le nombre de solutions d'un sous-problème est calculé, celui-ci est exploité pour le calcul du nombre de solutions du problème. Nous allons donc maintenant démontrer que $\#EBTD$ est capable de calculer le nombre exact de solutions d'un problème tout en garantissant d'éviter certains calculs inutiles au sens évoqué précédemment.

Théorème 13 *#EBTD est correct, complet et termine.*

Dans ce qui suit, nous exploitons les piles Z, $Z_{good_{\geq}}$ et $Z_{inconnu}$. Nous rappelons alors le contenu de celles-ci :

- Z est une pile globale contenant des clusters E_k à traiter tels que $\mathcal{A}[E_{p(k)} \cap E_k]$ ne correspond pas à un (no)good.
- $Z_{good_{\geq}}$ est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j.
- $Z_{inconnu}$ est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ ne correspond pas à un good de E_i par rapport à E_j, ni à un nogood.

Preuve :
Soit $G_{=}(E_i)$ l'ensemble des affectations correspondant aux goods exacts de $E_p(i)$ par rapport à E_i dans G. Nous nous basons dans cette preuve sur l'ensemble de variables $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ défini par :

$$V_{E_i} \cup (\bigcup_{E_j \in Fils(E_i) | \mathcal{A}[E_i \cap E_j] \notin G_{=}(E_j)} V_{Desc(E_j)} \backslash (E_i \cap E_j)) \cup (\bigcup_{E_k \in Z} V_{Desc(E_k)} \backslash (E_{p(k)} \cap E_k)).$$

$VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ contient les variables qui doivent être explorées par $\#EBTD$ afin de déterminer si une solution globale contenant $\mathcal{A}$ existe et le cas échéant de compter le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Plus précisément, $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ contient :

- V_{E_i} : l'ensemble de variables non assignées du cluster courant E_i.
- $V_{Desc(E_j)} \backslash (E_i \cap E_j)$: l'ensemble des variables de $Desc(E_j)$ sans compter les variables de $E_i \cap E_j$ tel que E_j est un cluster fils de E_i pour lequel $\mathcal{A}[E_i \cap E_j] \notin G_{=}(E_j)$. Si $\mathcal{A}[E_i \cap E_j] \in G_{=}(E_j)$, le good $\mathcal{A}[E_i \cap E_j]$ est exploité selon la définition d'un good exact et le sous-problème correspondant n'est pas visité. Si $\mathcal{A}[E_i \cap E_j]$ est un good partiel, le sous-problème enraciné en E_j n'est pas exploré lors de la recherche d'une solution globale (vu qu'au moins une extension cohérente existe pour ce sous-problème). Cependant, une fois une solution globale trouvée, il sera visité afin de calculer $\#sol_{E_j}$. Sinon, $\mathcal{A}[E_i \cap E_j]$ est inconnu et le sous-problème correspondant doit être exploré pour déterminer si une solution globale existe. Notons que dans le cas où $\mathcal{A}[E_i \cap E_j]$ est un nogood aucune solution globale contenant $\mathcal{A}$ n'existe et le sous-problème P_j ne sera pas visité.
- $V_{Desc(E_k)} \backslash (E_{p(k)} \cap E_k)$: l'ensemble des variables de $Desc(E_k)$ sans compter les variables de $E_{p(k)} \cap E_k$ tel que $E_k \in Z$. Le cluster E_k est inséré dans Z vu que $\mathcal{A}[E_{p(k)} \cap E_k]$ n'est ni un good partiel, ni un good exact ou un nogood. Ainsi, Z contient les clusters dont les variables de leur descendance doivent être explorées pour déterminer si une solution globale existe.

Pour prouver ce théorème, nous procédons par induction sur $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ et nous considérons la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ définie par :
"$\#EBTD(P, (E, T), \mathcal{A}, E_i, V_{E_i}, Z, G^d, N^d)$ retourne une paire $(\#sol, E_{bt})$ où $\#sol$ est le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ si $E_i = E_{bt}$, et une borne inférieure sinon". Lorsqu'il n'y a pas d'ambiguïté, $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ est noté VAR.

- Cas de base : Considérons $\mathcal{P}(\emptyset)$. Si E_i est un cluster feuille, $\#EBTD$ retourne $(1, E_i)$ vu que la seule extension possible de $\mathcal{A}$ est $\mathcal{A}$. Sinon, comme $VAR = \emptyset$ alors les trois termes de VAR sont vides. Alors, pour chaque cluster E_j fils de E_i, $\mathcal{A}[E_i \cap E_j] \in G_=(E_j)$. Donc, en considérant pour chaque cluster fils E_j de E_i (lignes 10-11), $\#EBTD$ met à jour successivement le nombre de solutions de $\#sol$. Vu que pour chaque cluster fils E_j de E_i, $\mathcal{A}[E_i \cap E_j]$ est un good exact, $Z_{good_\geq}$, $Z_{inconnu}$ et Z restent vides. Ainsi $\#EBTD$ retourne $(\#sol, E_i)$ où $\#sol$ est le exact nombre de solutions de $P_i|\mathcal{A}[E_{p(i)} \cap E_i]$. En conséquence, la propriété $\mathcal{P}(\emptyset)$ est valide.

- Cas général : Considérons maintenant $\mathcal{P}(VAR)$ avec $VAR \neq \emptyset$. Nous supposons que $\forall VAR' \subset VAR$, $\mathcal{P}(VAR')$ est valide.

  - Si $V_{E_i} \neq \emptyset$: la boucle (lignes 50-56) explore le cluster E_i comme $\#BT$. La seule différence est que cette boucle est suspendue lorsque $E_{bt} \neq E_i$ ce qui est nécessaire si un nogood est trouvé ultérieurement pendant la recherche. Si $E_{bt} = E_i$, la boucle est équivalente à celle de $\#BT$ et $\#sol_x$ est le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ pour les valeurs v déjà affectées à x. Lorsque $\#EBTD$ essaye d'assigner v à x, deux sous-cas existent :
    * $\mathcal{A} \cup \{x \leftarrow v\}$ est incohérente, il n'y a pas d'extension possible et $\#sol_x$ et E_{bt} restent inchangés.
    * $\mathcal{A} \cup \{x \leftarrow v\}$ est cohérente, nous faisons un appel récursif à $\#EBTD$ (ligne 54) sur $V_{E_i} \backslash \{x\}$. Dans ce cas, d'après l'hypothèse d'induction, $\mathcal{P}(VAR \backslash \{x\})$ est valide. Si $E_{bt} = E_i$, alors $\#sol_{xv}$ est le nombre exact de solutions du problème $P_i|\mathcal{A} \cup \{x \leftarrow v\}$. $\#sol_x$ est alors mis à jour pour la valeur v. Si $E_{bt} \neq E_i$, $\#sol_{xv}$ est une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A} \cup \{x \leftarrow v\}$ et $\#sol_x$ est alors mis à jour. Comme x est affectée pour la première fois en E_i alors $x \notin E_{bt}$ et la boucle est alors suspendue retournant une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$.

    Ainsi, la propriété $\mathcal{P}(VAR)$ est valide dans les deux sous-cas.
  - Si $V_{E_i} = \emptyset$: Comme $VAR \neq \emptyset$, pour chaque cluster fils E_j de E_i, $\mathcal{A}[E_i \cap E_j]$ peut être un good ou une affectation inconnue. Aussi, Z peut être vide ou non. Les lignes 6-16 considèrent chaque cluster fils E_j et mettent à jour, si nécessaire, Z, $Z_{good_\geq}$, $Z_{inconnu}$ et $\#sol$. Ainsi, si $\mathcal{A}[E_i \cap E_j]$ correspond à un good exact $\#sol$ est alors mis à jour en fonction du nombre de solutions enregistré $\#sol_{E_j}$. Si $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel, le cluster E_j est inséré dans $Z_{good_\geq}$. Sinon, E_j est inséré dans Z et $Z_{inconnu}$. Les lignes 17-35 permettent de lancer l'exploration des clusters de Z dans le but de déterminer si une solution globale contenant $\mathcal{A}$ existe. En ce qui concerne l'appel récursif à la ligne 20, nous montrons que $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Si $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subset VAR(V_{E_i}, Z, \mathcal{A}, G^d)$, par hypothèse d'induction, $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Sinon, nous avons $VAR(V_{E_j}, Z, \mathcal{A}, G^d) = VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. Toutefois, nous savons que pour l'appel suivant, $V_{E_j} \neq \emptyset$ parce que nous explorons le cluster E_j comme dans les lignes 46-57. Cela correspond au cas $V_{E_j} \neq \emptyset$ décrit ci-dessus. Alors la propriété $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Nous distinguons à ce stade trois sous-cas possibles :
    * Si $\#sol_{E_j} = 0$, $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$ n'admet aucune solution et l'affectation $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un nogood structurel (ligne 31). En raison de l'enregistrement d'un nogood, la recherche doit faire un retour-arrière vers le cluster impliqué dans cette incohérence qui est $E_{p(j)}$. Si

$E_i = E_{p(j)}$ (ligne 32), les fils de E_i sont supprimés de la pile Z vu que leur exploration est désormais inutile (ligne 33). Ainsi, $\#EBTD$ retourne $(0, E_i)$ puisqu'il n'existe aucune extension possible de $\mathcal{A}$ sur les variables de $Desc(E_j)$ et le cluster E_i doit être modifié (ligne 34). Sinon, $E_i \neq E_{p(j)}$ et la recherche doit faire un retour-arrière plus loin. Ainsi, $\#EBTD$ retourne $(\#sol, E_{p(j)})$ avec $\#sol$ une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A}[E_i]$ et le nouveau cluster E_{bt} est $E_{p(j)}$ (ligne 35). En conséquence, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.

* Si $\#sol_{E_j} > 0$ et $E_{bt} \neq E_j$ alors le nombre de solutions retourné est nécessairement une borne inférieure sur le nombre de solutions $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$. $\mathcal{A}[E_{p(j)} \cap E_j]$ est alors enregistrée comme un good partiel qui peut être exploité ultérieurement, si nécessaire. Le raisonnement des lignes 26-29 est similaire à celui pour les lignes 32-35. Soit le cluster courant E_i est impliqué dans l'incohérence (lignes 26-28), soit la recherche fait un retour-arrière jusqu'au cluster E_{bt} et essaye de modifier l'affectation de ses variables (ligne 29). Ainsi, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.
* Si $\#sol_{E_j} > 0$ et $E_{bt} = E_j$ alors le nombre de solutions retourné est celui du nombre exact de solutions de $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$. $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un good exact. À ce niveau, Z est vide puisque sinon l'exploration des clusters de Z aurait continué jusqu'à atteindre le dernier cluster. En d'autres termes, l'enregistrement des goods exacts ne peut pas se faire avant que tous les clusters de Z ne soient explorés. En outre, sachant que la traversée de la décomposition se fait grâce à une pile, si $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un good exact alors tous les clusters E_k insérés dans Z avant E_j sont déjà exhaustivement explorés et l'affectation $\mathcal{A}[E_{p(k)} \cap E_k]$ est déjà enregistrée comme un good exact. Ainsi, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.

La propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est alors valide pour $Z \neq \emptyset$ pour les trois sous-cas possibles. À la ligne 36, $Z = \emptyset$, une solution globale a été déjà trouvée et pour chaque cluster fils $E_j \in Z_{inconnu}$, $\#sol_{E_j}$ est calculé et $\#sol$ est mis à jour (ligne 37). Si $Z_{good_\geq} \neq \emptyset$, chaque cluster $E_j \in Z_{good_\geq}$ est exploré (ligne 41). Si $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subset VAR(V_{E_i}, Z, \mathcal{A}, G^d)$, par hypothèse d'induction, $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Sinon, nous avons $VAR(V_{E_j}, Z, \mathcal{A}, G^d) = VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. Cependant, comme pour l'appel à la ligne 20, nous savons que pour l'appel suivant $V_{E_j} \neq \emptyset$ et $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Par définition d'un good partiel, $P_j|\mathcal{A}[E_i \cap E_j]$ a au moins une seule solution. Comme Z est vide (une solution globale existe), l'appel récursif à $\#EBTD$ retourne nécessairement le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$. Une fois, tous les clusters fils E_j considérés, $\#sol$ est mis-à-jour et $\#EBTD$ retourne $(\#sol, E_i)$. En conséquence, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide pour $V_{E_i} \neq \emptyset$.

En conséquence, nous pouvons déduire que la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide. Au niveau de la terminaison, nous pouvons constater que la taille de VAR décroît à chaque appel. En effet, dans le cas où $V_{E_i} \neq \emptyset$, l'appel récursif fait par $\#EBTD$ considère l'ensemble $V_{E_i} \backslash \{x\}$ qui contient strictement moins de variables que V_{E_i} (une variable en moins). Ainsi, la taille de VAR est désormais plus petite. Si $V_{E_i} = \emptyset$, comme nous l'avons déjà vu, les appels récursifs aux lignes 20 et 41 garantissent que $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subseteq VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. En cas d'égalité, nous exploitons le fait que pour l'appel suivant

$V_{E_j} \neq \emptyset$ et ainsi l'appel récursif à la ligne 54 garantit que le nombre de variables de $VAR(V_{E_j}, Z, \mathcal{A}, G^d)$ diminuera.

L'algorithme $\#EBTD$ est alors correct, complet et termine.□

Finalement, nous donnons les complexités en temps et en espace qui sont comparables avec celles de $\#BTD$.

Théorème 14 *$\#EBTD$ a une complexité en temps en $O(n.(r.m + n.s).d^{w^+ + 1})$ et en espace en $O(n.s.d^s)$.*

Preuve : Le nombre maximal de contraintes impliquées dans chaque vérification de cohérence est m et le coût de chaque vérification de contrainte est de $O(r)$. En outre, nous supposons que les nogoods d'un cluster E_i par rapport à un de ses clusters fils E_j sont stockés comme une contrainte dont la portée est l'ensemble de variables de $E_i \cap E_j$ et qu'il y a au plus n-1 séparateurs (vu que le nombre de clusters est majoré par n). De plus, la vérification de l'existence d'un nogood peut être réalisée en $O(s)$. Ainsi, la vérification de cohérence de la ligne 53 est réalisée en $O(r.m + n.s)$.

Dans le pire des cas, $\#EBTD$ explore tous les clusters de la décomposition et essaye toutes les valeurs possibles de chaque variable du cluster. Comme w^+ est la largeur de la décomposition arborescente employée, la taille maximale d'un cluster est de $w^+ + 1$. Alors, le nombre d'affectations d'un cluster est borné par $d^{w^+ + 1}$. Un good exact enregistré assure que le cluster correspondant n'est pas exploré plus d'une fois pour la même affectation des variables de son séparateur. Si un good partiel est enregistré, le cluster est, au plus, visité deux fois avec la même affectation sachant que la deuxième fois le sous-problème correspondant est entièrement exploré et ainsi le good partiel est remplacé par un good exact. En plus, supposons que les goods sont mémorisés pour chaque séparateur comme les nogoods. Mémoriser un good ou vérifier son existence est alors réalisé en $O(s)$. En conséquence, $\#EBTD$ a une complexité temporelle en $O(n.(r.m + n.s).d^{w^+ + 1})$.

En ce qui concerne la complexité spatiale, $\#EBTD$ enregistre des goods exacts, partiels et des nogoods. Ces enregistrements sont réalisés par rapport à un séparateur $E_i \cap E_j$ où E_j est un fils de E_i. Étant donné que s est la taille maximale des séparateurs, la taille d'un (no)good est bornée par s. Pour chaque séparateur, il existe au plus d^s affectations possibles. Ainsi, comme le nombre de séparateurs est borné par n, la complexité spatiale est en $O(n.s.d^s)$. □

Notons que, lorsque $\#EBTD$ explore, pour la deuxième fois, un sous-problème donné dans le but de calculer un good exact, nous pouvons éviter de redémarrer la recherche du début. En effet, si nous supposons que nous utilisons un ordre de valeurs lexicographique, il suffit d'enregistrer la solution partielle correspondante au good partiel conjointement avec ce good partiel. Ce faisant, si $\#EBTD$ a besoin de réexplorer le sous-problème correspondant il peut en toute sécurité démarrer à partir de cette solution partielle. Un tel problème ne change pas la complexité temporelle tandis que la complexité spatiale devient $O(n.w^+.d^s)$. Cependant, d'un point de vue pratique, les gains peuvent être significatifs.

Avant d'évaluer expérimentalement l'intérêt pratique de $\#EBTD$, nous devons préciser que suite à la publication de ce travail, nous avons été informés de l'existence d'un travail similaire présenté dans la thèse de Karakashian [Karakashian, 2013]. L'idée est fondamentalement la même. L'algorithme proposé évite de compter le nombre de solutions pour un sous-problème avant de s'être assuré que l'affectation partielle s'étend en une solution globale. Néanmoins, du point de vue théorique, certaines faiblesses peuvent être soulignées. L'algorithme fourni ne traduit pas exactement le déroulement décrit. Aucune distinction

n'est faite entre un good partiel et un good exact. En plus, il n'est fourni ni preuve de validité, ni preuve de complexité. Finalement, la partie expérimentale donnée est peu développée et la comparaison entre $\#BTD$ et le nouvel algorithme proposé ne met pas ce dernier suffisamment en valeur.

6.4 Étude expérimentale

Nous évaluons dans cette section l'intérêt pratique de notre approche.

6.4.1 Benchmark utilisé

Nous considérons des instances CSP représentées dans le format XCSP3 [Boussemart et al., 2016]. Tous les détails nécessaires peuvent être trouvés dans [XCS, 2017]. Nous considérons 4 069 instances cohérentes. Elles rassemblent des instances provenant notamment des applications réelles comme la famille *Renault* ou la famille $Rlfap$. Une description de telles familles peut être retrouvée dans [XCS, 2017]. Une majorité d'instances représentent des problèmes académiques dont une partie est générée aléatoirement. Le nombre de variables des instances est situé entre 4 et 28 000 variables avec des domaines d'une taille variant de 1 à 6 561. Le nombre de contraintes varie de 2 à 139 500 d'une arité allant de 2 à 1 000. Il s'agit des contraintes de divers types exprimées en intention ou en extension ou sous forme de contraintes globales (*AllEqual*, *AllDiff*, *Sum*, *Element*) [Beldiceanu et al., 2005; van Hoeve and Katriel, 2006]. Ces instances peuvent être composées d'une seule composante connexe ou d'un plus grand nombre allant jusqu'à 458 composantes connexes. Le nombre de solutions varient d'une instance à l'autre. Certaines instances ne possèdent qu'une seule solution alors que d'autres peuvent avoir un nombre de solutions de l'ordre de 10^{303} solutions. Ce benchmark sera noté I_4.

6.4.2 Protocole expérimental

En ce qui concerne les décompositions, nous retenons les décompositions suivantes (cf. chapitre 3 pour plus de détails sur H_i et $Min\text{-}Fill\text{-}MG$) :

- $Min\text{-}Fill$: considérée comme étant l'heuristique de l'état de l'art de calcul de décompositions dans la communauté CP ; elle vise à minimiser w^+.
- H_1 : elle vise également à minimiser w^+ sans recourir à la triangulation (cf. chapitre 3).
- H_2 : son objectif consiste à construire des décompositions composées uniquement de clusters connexes.
- H_3 : elle permet de calculer une décomposition telle que chaque cluster possède plusieurs clusters fils.
- H_5 : elle vise à limiter la taille des séparateurs de la décomposition (H_4 est écartée parce qu'elle détecte moins de séparateurs que H_5). Toutefois, nous l'utilisons sans limite de taille pour les séparateurs afin de trouver le plus de séparateurs possibles. Elle sera notée H_5^∞.
- $Min\text{-}Fill\text{-}MG$: elle vise à minimiser w^+ en triangulant d'une façon plus attentionnée que $Min\text{-}Fill$.

Au niveau des algorithmes de résolution, nous considérons les algorithmes suivants :

Algorithme	*Min-Fill*		H_1		*Min-Fill-MG*	
	#rés.	temps	#rés.	temps	#rés.	temps
#*EBTD*	3 023	158 998	2 942	134 147	3 029	162 559

TABLE 6.1 – Nombre d'instances résolues et temps d'exécution en secondes pour $\#EBTD$ selon *Min-Fill*, H_1 et *Min-Fill-MG* pour le benchmark I_4.

Algorithme	H_2		H_3		H_5^∞	
	#rés.	temps	#rés.	temps	#rés.	temps
#*EBTD*	2 970	141 380	2 791	169 357	2 739	148 512

TABLE 6.2 – Nombre d'instances résolues et temps d'exécution en secondes pour $\#EBTD$ selon H_2, H_3 et H_5 pour le benchmark I_4.

- $\#EBTD$: nous considérons une version de $\#EBTD$ basée sur RFL que nous implémentons dans notre propre bibliothèque. Nous choisissons RFL au lieu de MAC vu que RFL réalise un branchement d-aire et que pour le comptage toutes les valeurs d'une variable seront testées. La cohérence d'arc est renforcée en prétraitement via $AC\text{-}3^{rm}$ [Lecoutre et al., 2007c] et pendant la résolution via $AC\text{-}8^{rm}$ (AC-8 [Chmeiss and Jégou, 1998] avec des résidus multi-directionnels). La prochaine variable à affecter est choisie grâce à l'heuristique $dom/wdeg$ [Boussemart et al., 2004]. Le cluster racine est celui qui maximise la somme des poids $wdeg$ des contraintes qui intersectent le cluster.

- $\#BTD$: nous considérons l'implémentation de $\#BTD$ fournie dans *Toulbar*2 [Favier et al., 2009]. La décomposition employée par $\#BTD$ est *Min-Fill* également fournie dans Toulbar2. Le cluster racine est celui ayant la plus grande taille.

- $\#RFL$: nous implémentons une version dans notre bibliothèque.

- *cn*2*mddg* [Koriche et al., 2015] (cf. chapitre 2) : nous considérons l'implémentation du compilateur fournie à l'adresse suivante `http://www.cril.univ-artois.fr/KC/mddg.html`.

- #SAT solveurs (cf. chapitre 2) : nous considérons *cachet* [Sang et al., 2004], *c*2*d* [Darwiche, 2001a], *relsat* [Bayardo and Pehoushek, 2000] et *sharpsat* [Thurley, 2006].

Les expérimentations ont été réalisées sur des serveurs lame sous Linux Ubuntu 14.04 dotés chacun de deux processeurs Intel Xeon E5-2609 à 2,4 GHz et de 32 Go de mémoire.

Chaque instance dispose de 20 minutes (incluant le cas échéant, le temps de calcul de la décomposition) et de 16 Go de mémoire.

6.4.3 Observations et analyse des résultats

#EBTD selon la décomposition utilisée Nous comparons tout d'abord le comportement de $\#EBTD$ selon la décomposition utilisée. La table 6.1 montre le nombre d'instances résolues et le temps cumulé d'exécution pour $\#EBTD$ selon les décompositions visant à minimiser w^+ qui sont : *Min-Fill*, H_1 et *Min-Fill-MG*. Quant à la table 6.2, elle montre le nombre d'instances résolues et le temps cumulé d'exécution pour $\#EBTD$ selon les décompositions dont le but est d'augmenter l'efficacité de la résolution des instances CSP. Nous remarquons que les décompositions dont l'objectif est de minimiser w^+ sont généralement plus efficaces vis-à-vis du comptage du nombre de solutions que les

Paramètre	*Min-Fill*	H_1	*Min-Fill-MG*	H_2	H_3	H_5^∞
p_1	122	108	122	57	34	37
p_2	42	41	42	29	15	18
p_3	29	29	29	46	57	48

TABLE 6.3 – La valeur des paramètres p_1, p_2, p_3 pour chaque décomposition pour le benchmark I_4 : p_1 est la moyenne des nombres des séparateurs, p_2 est la moyenne des pourcentages du nombre des séparateurs par rapport à n, p_3 est la moyenne des pourcentages de w^+ par rapport à n.

Min-Fill	H_1	*Min-Fill-MG*	H_2	H_3	H_5^∞
111 150	114 379	107 478	97 596	116 930	112 016

TABLE 6.4 – Temps d'exécution en secondes pour $\#EBTD$ selon les différentes décompositions pour les instances résolues par $\#EBTD$ avec n'importe quelle décomposition du benchmark I_4.

autres décompositions. En effet, *Min-Fill-MG* permet de résoudre le plus grand nombre d'instances (3 029), suivie par *Min-Fill* (3 023), puis par H_2 (2 970), ensuite par H_1 (2 942) et enfin par H_3 (2 791) et H_5 (2 739). Ces résultats sont également représentés dans la figure 6.3. Il semble ainsi que les critères jugés pertinents pour permettre une résolution efficace des instances CSP ne sont pas les plus intéressants pour la résolution des instances #CSP. Si les décompositions telles que H_3 et H_5 permettent souvent de trouver rapidement une première solution (comme dans le cas des CSP), l'étape du comptage se trouve être pénalisée. En conclusion, la taille des clusters semble être un paramètre déterminant pour une résolution efficace des instances #CSP. Nous examinons maintenant les différentes décompositions de plus près. La table 6.3 montre pour chaque décomposition pour le benchmark I_4 la valeur de trois paramètres p_1, p_2 et p_3. p_1 est la moyenne des nombres des séparateurs, p_2 est la moyenne des pourcentages du nombre des séparateurs par rapport à n et p_3 est la moyenne des pourcentages de w^+ par rapport à n. Vu les résultats de la table 6.3, il semble que les décompositions $H_{2,3,5}$ sont pénalisées, en plus de la taille des clusters, par le nombre restreint de séparateurs. Ainsi, nous pouvons établir un lien entre le nombre de séparateurs et l'efficacité du comptage. En effet, plus le nombre de séparateurs augmente, plus l'efficacité du comptage est susceptible d'augmenter. Ceci est dû à l'exploitation des séparateurs et l'augmentation du taux d'enregistrements pouvant être réalisés. Notons que ce taux peut être à l'origine de la saturation de l'espace mémoire disponible. Par exemple, $\#EBTD$ explose en mémoire pour 32 instances avec *Min-Fill*, pour 53 instances avec H_2, pour une instance avec H_3 et H_5^∞, pour 165 instances avec H_1 et pour 25 instances avec *Min-Fill-MG*. Ainsi, il faut veiller à équilibrer le taux d'enregistrements réalisés. Lorsque nous nous limitons aux instances dont le ratio $\frac{n}{w^+}$ est supérieur ou égal à 10, les résultats sont représentés dans la figure 6.4. Ils montrent notamment l'importance de la connexité sur cette catégorie d'instances. Au niveau des temps d'exécution, nous comparons les temps d'exécution de $\#EBTD$ avec les différentes décompositions sur le benchmark formé des instances résolues avec toutes les décompositions. Ce benchmark contient 2 619 instances. Les résultats sont montrés par la table 6.4. Les temps d'exécutions cumulés ne sont pas significativement éloignés. Les meilleurs temps de résolution sont ceux obtenus avec H_2 et *Min-Fill-MG*. Nous retenons pour la suite la décomposition *Min-Fill-MG*. Il est à noter que $\#EBTD$ avec *Min-Fill-MG* est capable de calculer, soit le nombre exact de solutions, soit une borne inférieure non nulle sur le nombre de solutions pour environ 86% des instances considérées.

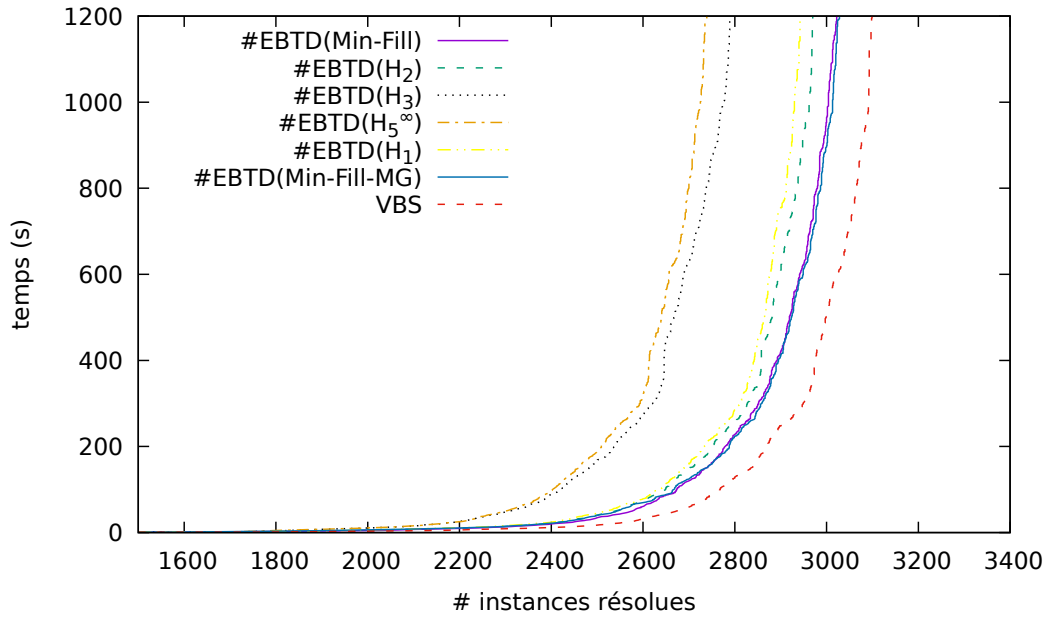

FIGURE 6.3 – Le nombre cumulé d'instances résolues pour $\#EBTD$ selon la décomposition employée et leur VBS.

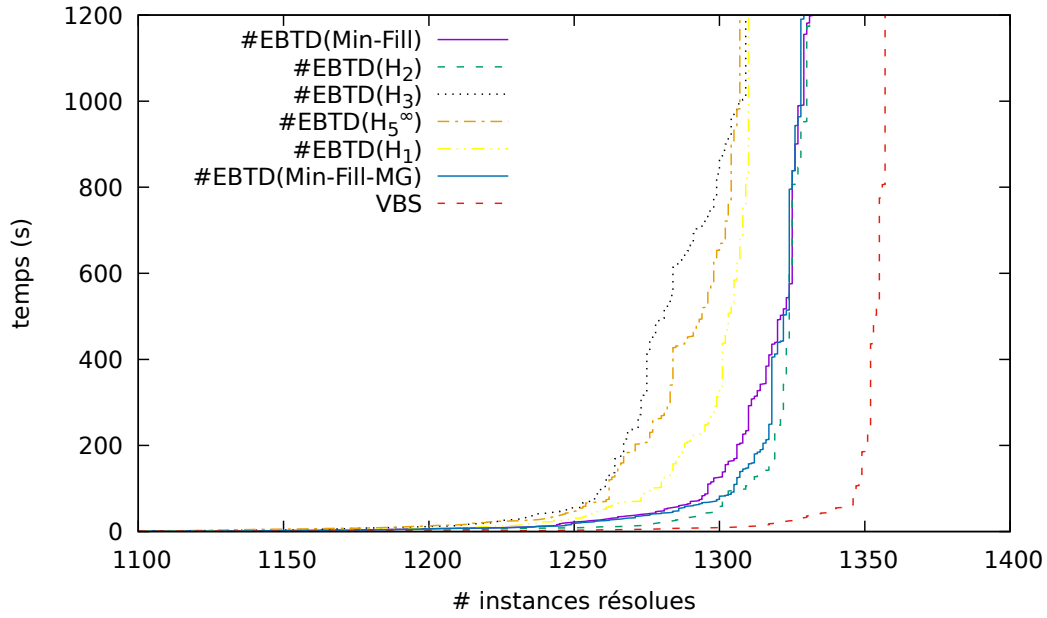

FIGURE 6.4 – Le nombre cumulé d'instances résolues pour $\#EBTD$ selon la décomposition employée et leur VBS pour les instances telles que $\frac{n}{w^+} \geq 10$ du benchmark I_4.

#EBTD vs #RFL Nous comparons à présent le comportement de $\#EBTD$ vis-à-vis de $\#RFL$. $\#RFL$ permet de compter exactement le nombre de solutions de 2 607 instances contre 3 029 pour *Min-Fill-MG*. $\#EBTD$ est alors significativement meilleur que $\#RFL$ en nombres d'instances résolues exactement. La figure 6.5 montre le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$ et leur VBS (pour *Virtual Best Solver*). La figure montre clairement l'intérêt de $\#EBTD$ par rapport à $\#RFL$. Elle montre

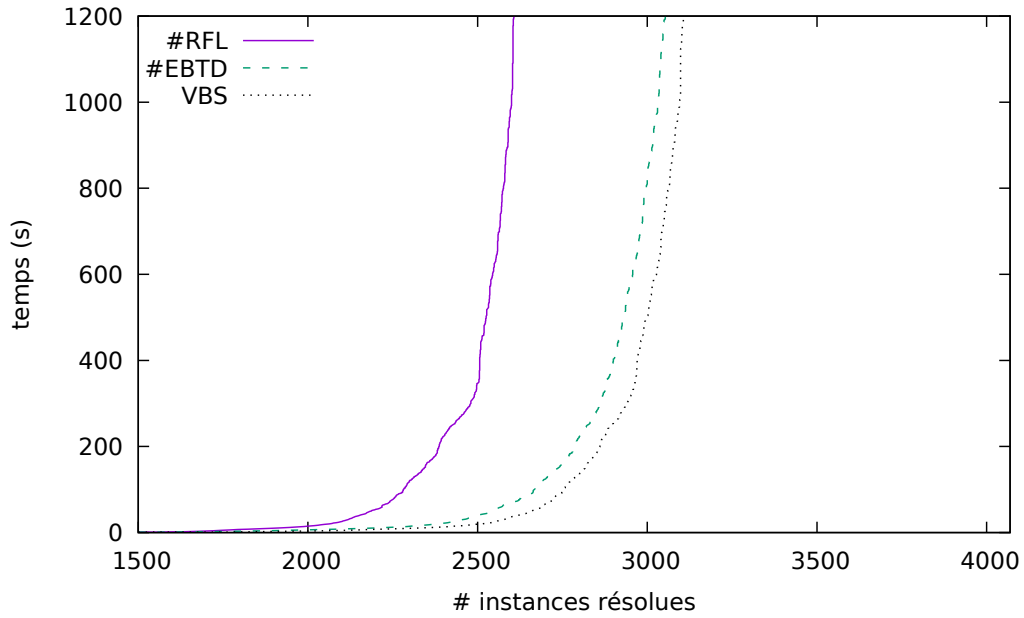

FIGURE 6.5 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$ et leur VBS pour le benchmark I_4.

également que $\#EBTD$ a un comportement proche de celui du VBS. Il y a cependant 56 instances résolues par $\#RFL$ et pas par $\#EBTD$. Parmi ces 56 instances, pour 35 instances $\#EBTD$ ne parvient à trouver aucune solution. Ce fait met en avant la difficulté que rencontre $\#EBTD$ à trouver la première solution dans certains cas. Ceci n'est pas étonnant vu les résultats obtenus dans le cadre du problème CSP. En effet, la décomposition *Min-Fill-MG* est loin d'être efficace pour résoudre une instance CSP par rapport à une méthode telle que MAC ou RFL. Nous comparons maintenant les temps d'exécution cumulés des deux algorithmes. Afin de comparer d'une façon équitable les temps, nous nous basons sur l'ensemble d'instances résolues à la fois par $\#EBTD$ et $\#RFL$. Il compte 2 551 instances résolues par $\#RFL$ en environ 120 000 s et par $\#EBTD$ en 130 000 s. Lorsque $\#RFL$ et $\#EBTD$ comptent tout les deux exactement le nombre de solutions et que l'instance est résolue plus rapidement avec $\#RFL$ qu'avec $\#EBTD$, ceci est *a fortiori* dû à l'exploitation de la liberté totale de l'heuristique de choix de variables. Il s'agit le plus souvent d'instances ayant un nombre de solutions relativement faible (une solution unique dans certains cas). En effet, lorsque le nombre de solutions d'une instance est considérablement élevé, la capacité de l'énumération est dépassée en l'absence de plusieurs composantes connexes. En plus, la redondance des espaces de recherche visités est handicapante et entraîne une forte dégradation de l'efficacité de la recherche. D'ailleurs, le plus souvent lorsque le nombre de solutions trouvées par $\#RFL$ est élevé, l'instance en question contient plusieurs composantes connexes. Ainsi, ce nombre de solutions élevé résulte de la multiplication du nombre de solutions de chaque composante connexe. Nous comparons finalement les bornes inférieures sur le nombre de solutions calculées par $\#RFL$ et $\#EBTD$ lorsque le nombre exact de solutions n'est pas trouvé en raison du dépassement du temps limite ou de l'espace mémoire disponible. 544 instances sont concernées. Pour 278 instances la borne inférieure calculée par $\#RFL$ est strictement inférieure à celle calculée par $\#EBTD$. Cependant, pour 118 instances la borne inférieure calculée par $\#EBTD$ est strictement inférieure à celle calculée par $\#RFL$. Pour le reste d'instances, la borne

inférieure calculée par les deux algorithmes est la même. Cette comparaison est illustrée

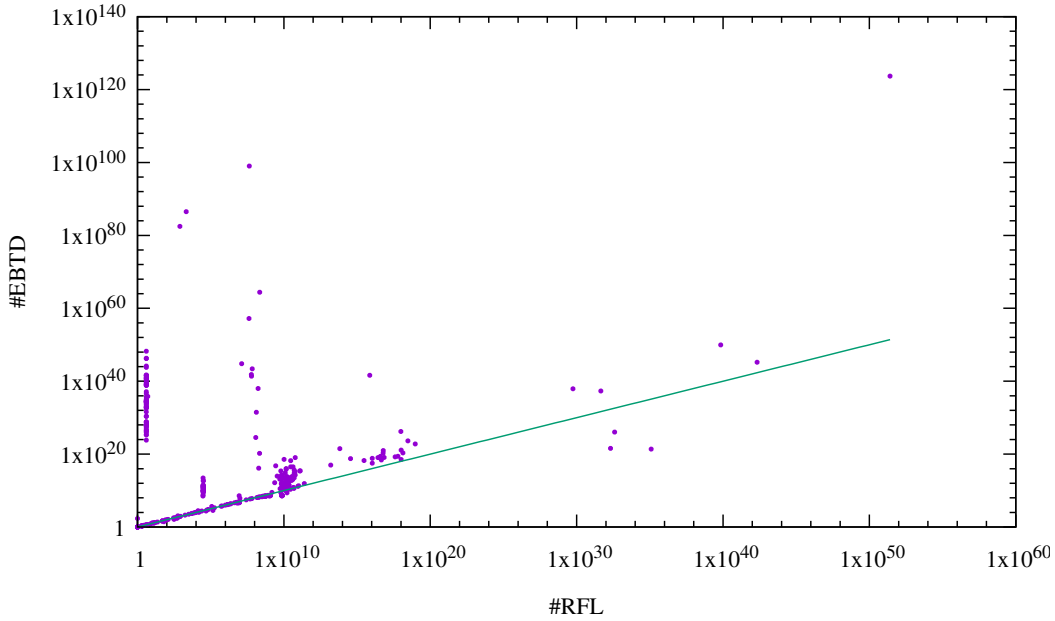

FIGURE 6.6 – Comparaison des bornes inférieures pour le nombre exact de solutions calculées par $\#RFL$ et $\#EBTD$.

par la figure 6.6.

#EBTD vs #BTD Nous comparons à présent $\#EBTD$ à $\#BTD$, la méthode structurelle de l'état de l'art pour le comptage du nombre de solutions. Afin d'y parvenir, nous encodons les instances considérées en un format supporté par *Toulbar*2. Pour $\#BTD$, nous encodons les instances CSP en format WCSP. Cet encodage nécessite la mise à plat des contraintes. Une telle opération pourrait être particulièrement coûteuse en temps ainsi qu'en espace mémoire notamment pour les instances contenant des contraintes en intention ou des contraintes globales. Ainsi, pour notre comparaison, nous considérons 3 956 instances que nous avons réussi à convertir dans moins de 25 minutes et en limitant la taille du fichier résultat à $200Mo$. Ce benchmark est noté $I_{4.1}$ avec $I_{4.1} \subset I_4$. Notons que dans ces expérimentations les temps de conversion du format XCSP3 à n'importe quel autre format ne sont pas inclus dans les temps de résolution des méthodes considérant les instances converties. Sur cet ensemble d'instances, $\#BTD$ résout 2 508 instances tandis que $\#EBTD$ en résout 2 996. $\#BTD$ résout donc considérablement moins d'instances que $\#EBTD$. En outre, le temps cumulé réalisé par $\#EBTD$ est d'environ 153 000 s tandis que celui de $\#BTD$ est d'environ 192 000 s. Ainsi, l'augmentation du nombre d'instances dont $\#EBTD$ calcule le nombre exact de solutions est accompagnée d'une baisse significative du temps de résolution. La figure 6.7 montre que $\#EBTD$ améliore clairement $\#BTD$. Finalement, la figure 6.8 montre l'apport important de $\#EBTD$ par rapport à $\#BTD$. Il est à noter que $\#BTD$ n'est pas capable de donner une borne inférieure sur le nombre de solutions en cas de dépassement de limite de temps ou d'espace mémoire. Cependant, $\#EBTD$ est capable de fournir une borne inférieure non nulle pour 370 instances additionnelles.

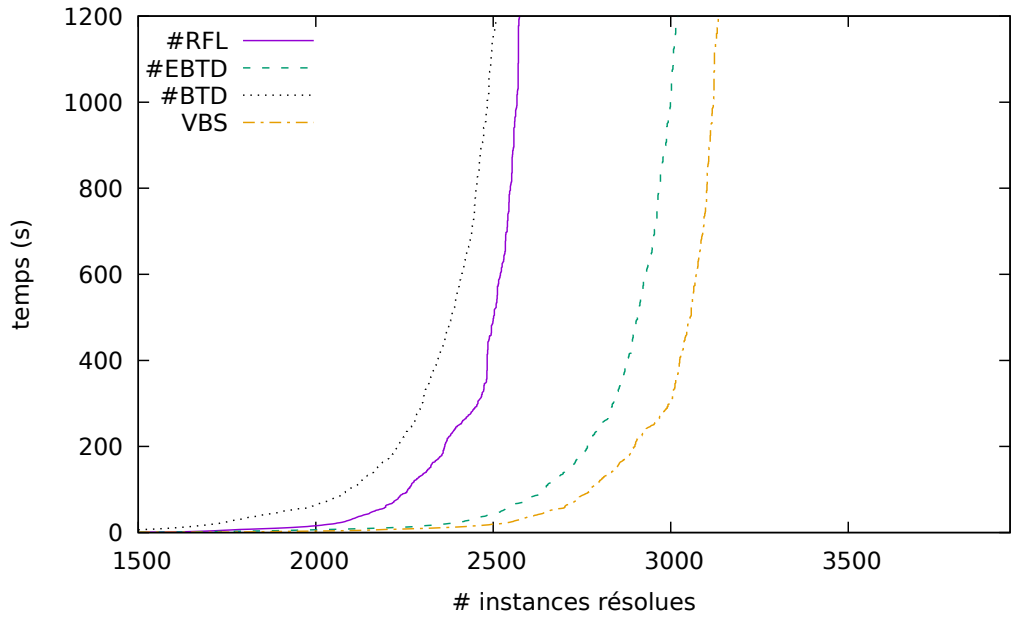

FIGURE 6.7 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$, $\#BTD$ et leur VBS pour le benchmark $I_{4.1}$.

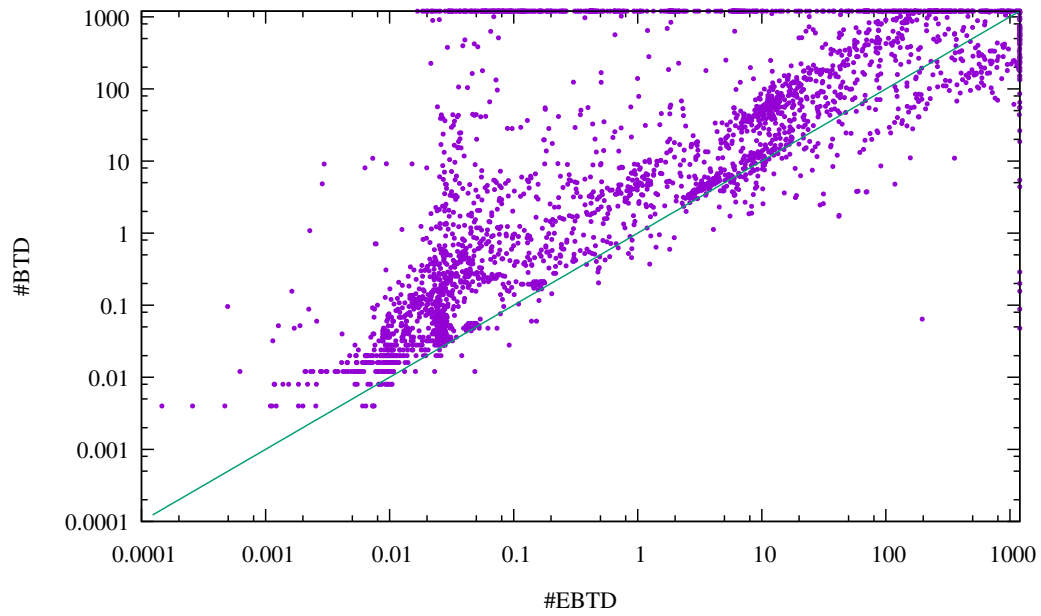

FIGURE 6.8 – Comparaison des temps d'exécution cumulés de $\#EBTD$ et de $\#BTD$.

#EBTD vs cn2mddg Nous comparons maintenant $\#EBTD$ à *cn2mddg* qui prend en entrée des instances au format XCSP2.1 [Roussel and Lecoutre, 2009]. C'est pourquoi la conversion nous oblige également à nous limiter aux 3 956 instances du benchmark $I_{4.1}$. *cn2mddg* permet de compter exactement le nombre de solutions de 3 177 instances en 146 430 s. La compilation réalisée par *cn2mddg* semble permettre de compter le nombre de solutions d'une instance plus efficacement que $\#EBTD$. La figure 6.9 montre que *cn2mddg* a une meilleure performance que les autres méthodes sur l'ensemble total des instances.

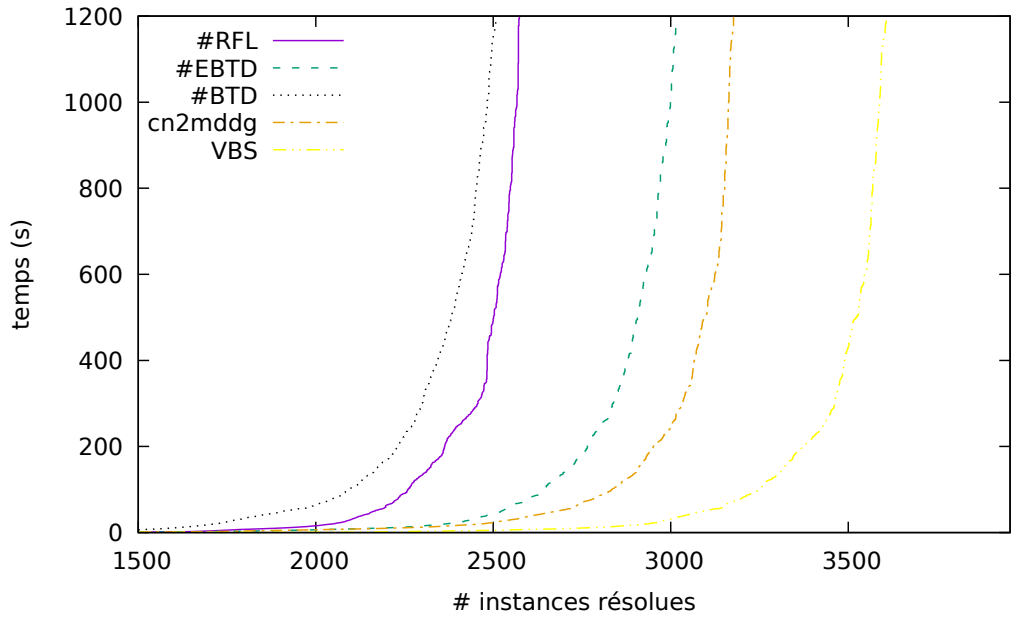

FIGURE 6.9 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$, $\#BTD$, $cn2mddg$ et leur VBS pour le benchmark $I_{4.1}$.

Le comportement du VBS est davantage éloigné de celui du reste des méthodes. Cela montre que les différentes méthodes ne résolvent pas les mêmes instances. Il est à noter que contrairement à $\#EBTD$, le nombre de solutions d'une instance est soit compté exactement, soit aucune solution n'est trouvée dans le cas de $cn2mddg$. Ensuite, nous regardons l'évolution du nombre d'instances résolues à l'exact par $\#EBTD$ et $cn2mddg$ en se restreignant à un ensemble d'instances de plus en plus difficile pour $cn2mddg$. Par exemple, la deuxième ligne du tableau 6.5 indique que nous nous limitons aux instances du benchmark $I_{4.1}$ résolues en plus de 10 s. Les résultats sont montrés dans la table 6.5 et la figure 6.10. Nous remarquons à travers ce tableau que l'augmentation de la difficulté des instances pour $cn2mddg$ met de plus en plus en valeur $\#EBTD$. En effet, pour des instances faciles pour $cn2mddg$, le temps de résolution pourrait potentiellement même être inférieur au temps de calcul de la décomposition. Dans ces cas, l'utilisation d'une décomposition n'est pas justifée. Ce fait met en évidence la complémentarité des deux approches. Ainsi, lorsque l'instance est difficile pour $cn2mddg$, l'utilisation de $\#EBTD$ semble être une alternative intéressante.

#EBTD vs #SAT solveurs Dans cette partie, nous comparons $\#EBTD$ aux #SAT solveurs. Nous devons donc considérer un format supporté par les différents #SAT solveurs. Nous exploitons alors l'encodage direct du format CSP au format SAT [Walsh, 2000], l'encodage logarithmique [Walsh, 2000] et l'encodage tuple [Hurley et al., 2016]. La conversion étant coûteuse en temps et en espace, nous sommes contraints de considérer un sous-ensemble d'instances pour les différents encodages. Nous considérons les instances que nous avons réussi à convertir dans un limite de 25 minutes et dont la taille en arrivée ne dépasse pas $200Mo$. Pour l'encodage direct, nous considérons un benchmark de 2 573 instances noté $I_{direct} \subset I_{4.1}$. En ce qui concerne l'encodage logarithmique, nous nous limitons à un benchmark de 2 424 instances noté $I_{log} \subset I_{4.1}$. Cependant, nous considérons un benchmark de 3 052 instances pour l'encodage tuple noté $I_{tuple} \subset I_{4.1}$.

Temps de résolution de *cn2mddg*	#rés par $\#EBTD$	#rés par *cn2mddg*
≥ 0	2 996	3 177
≥ 10	882	980
≥ 20	644	711
≥ 50	444	493
≥ 100	336	344
≥ 200	248	226
≥ 300	200	144
≥ 400	179	111
≥ 500	166	80
≥ 600	153	52
≥ 700	145	35
≥ 800	138	25
≥ 900	138	20
≥ 1000	134	14
≥ 1100	133	9
≥ 1200	129	0

TABLE 6.5 – Le nombre d'instances résolues à l'exact par $\#EBTD$ et *cn2mddg* en se restreignant à un ensemble d'instances de plus en plus difficile pour le compilateur *cn2mddg*.

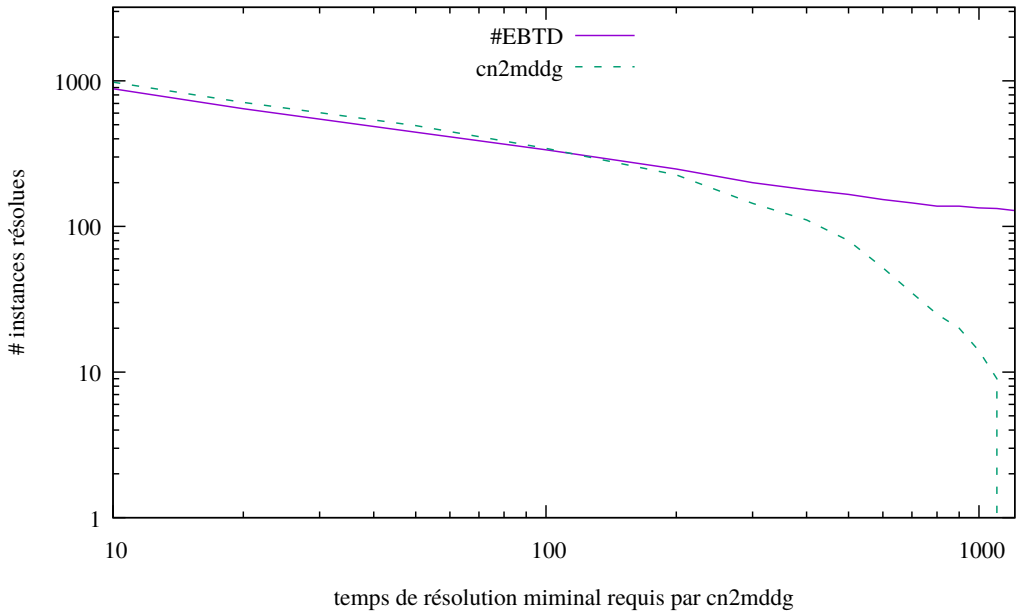

FIGURE 6.10 – Évolution du nombre d'instances résolues par $\#EBTD$ et *cn2mddg* selon les instances considérées du benchmark $I_{4.1}$. Nous nous restreignons à un benchmark de plus en plus difficile pour *cn2mddg*.

Les tables 6.6, 6.7 et 6.8 ainsi que les figures 6.11, 6.12 et 6.13 montrent que les #SAT solveurs ne sont pas compétitifs vis-à-vis des autres compteurs. Nous constatons aussi qu'à chaque fois le VBS a une performance bien meilleure que tous les algorithmes. Nous pouvons alors déduire que les algorithmes évalués résolvent des instances différentes. Lorsque l'encodage logarithmique est utilisé, nous pouvons facilement déduire que les compteurs #SAT ont une performance très médiocre. Nous pouvons alors écarter ces compteurs.

Algorithme	#rés.	temps (s)
$\#RFL$	1 788	86 043
$\#EBTD$	2 231	106 070
$\#BTD$	2 007	156 906
cn2mddg	2 254	97 938
c2d	1 008	223 265
relsat	1 411	130 884
cachet	1 088	208 630
sharpsat	1 990	232 123
VBS	2 388	65 862

TABLE 6.6 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{direct}.

Algorithme	#rés.	temps (s)
$\#RFL$	1 661	86 658
$\#EBTD$	2 102	106 235
$\#BTD$	1 880	157 842
cn2mddg	2 115	96 102
c2d	703	60 737
relsat	649	38 079
cachet	657	168 061
sharpsat	886	132 225
VBS	2 251	68 594

TABLE 6.7 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{log}.

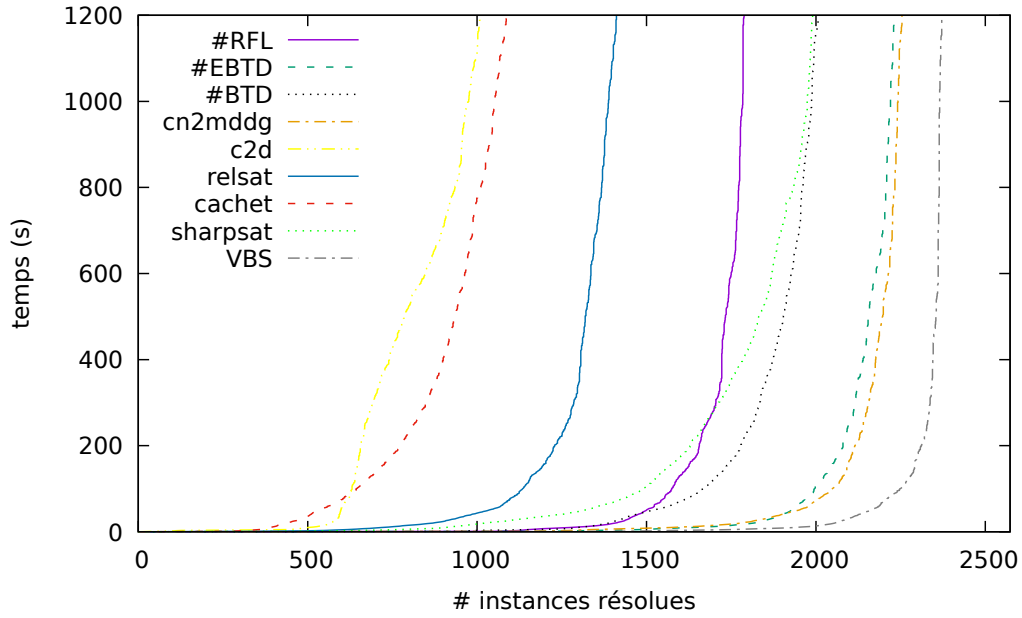

FIGURE 6.11 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{direct}.

Algorithme	#rés.	temps (s)
$\#RFL$	2 313	102 351
$\#EBTD$	2 753	123 995
$\#BTD$	2 394	180 354
cn2mddg	2 758	102 240
c2d	1 074	172 571
relsat	1 430	72 787
cachet	1 942	160 521
sharpsat	2 321	293 360
VBS	2 902	89 533

TABLE 6.8 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{tuple}.

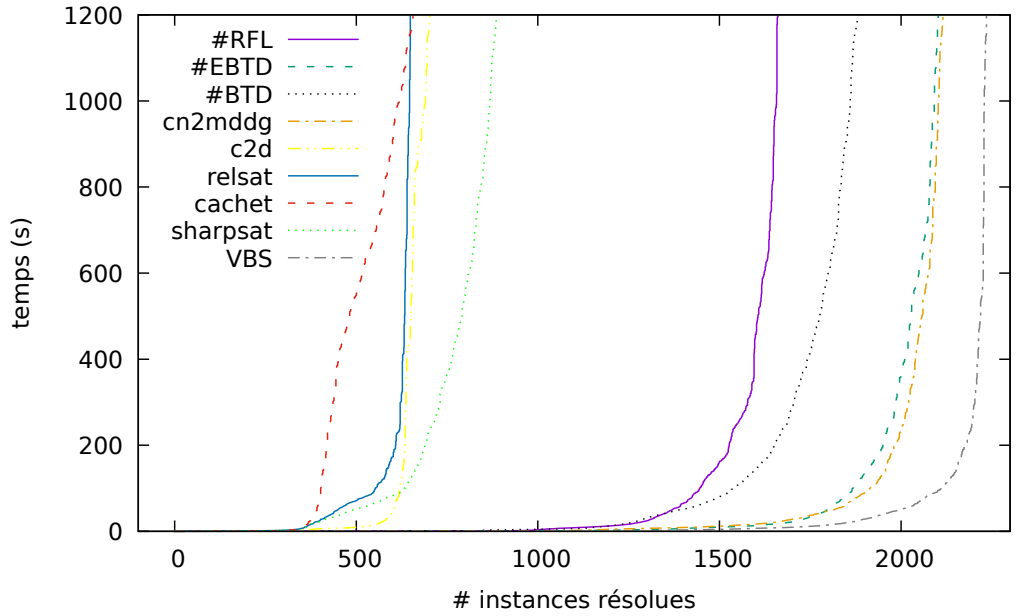

FIGURE 6.12 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{log}.

Pour les deux autres encodages, le compteur qui semble le plus efficace est *sharpsat*. Nous nous intéressons maintenant au benchmark résolu à la fois par $\#EBTD$ et *sharpsat*. Sur les deux figures 6.14 et 6.15, nous pouvons constater que malgré l'existence de certaines instances résolues plus rapidement avec *sharpsat* qu'avec $\#EBTD$, $\#EBTD$ domine fortement *sharpsat*. En effet, pour la majorité des instances $\#EBTD$ est capable de compter le nombre exact de solutions plus rapidement que *sharpsat*. D'ailleurs, les temps cumulés d'exécution de $\#EBTD$ et de *sharpsat* sont respectivement 77 611 s et 223 859 s lorsque l'encodage direct est exploité et 34 973 s et 274 256 s si l'encodage tuple est utilisé. Il est à noter finalement qu'aucun compteur $\#SAT$ ne fournit une borne inférieure lorsqu'un dépassement de limite de temps ou d'espace a lieu.

Bilan Dans cette partie, nous avons bien mis en avant l'intérêt pratique de $\#EBTD$. Nous l'avons comparé à de multiples compteurs comme $\#RFL$, $\#BTD$, *cn2mddg*, *c2d*,

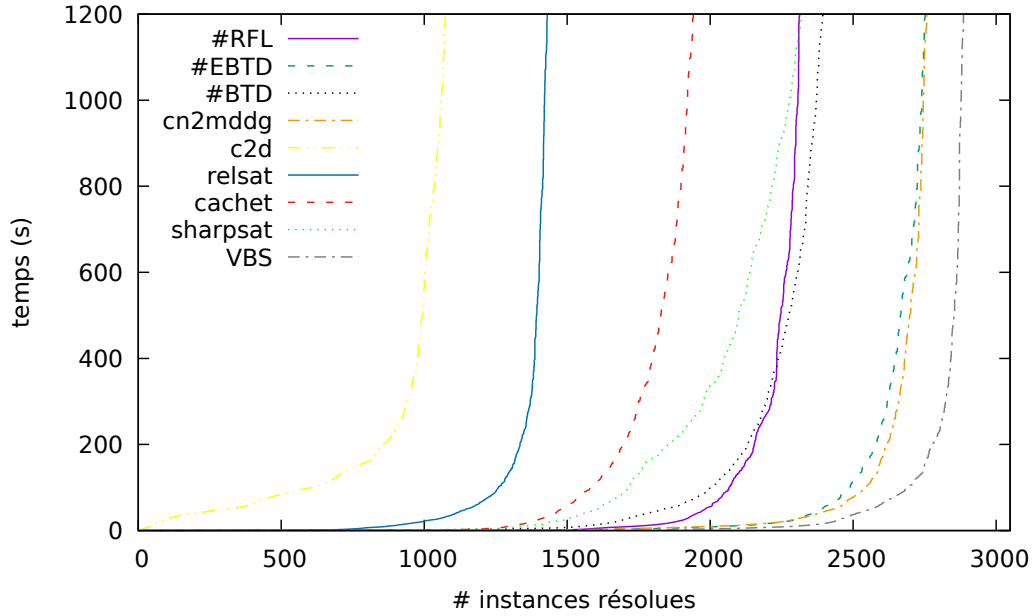

FIGURE 6.13 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{tuple}.

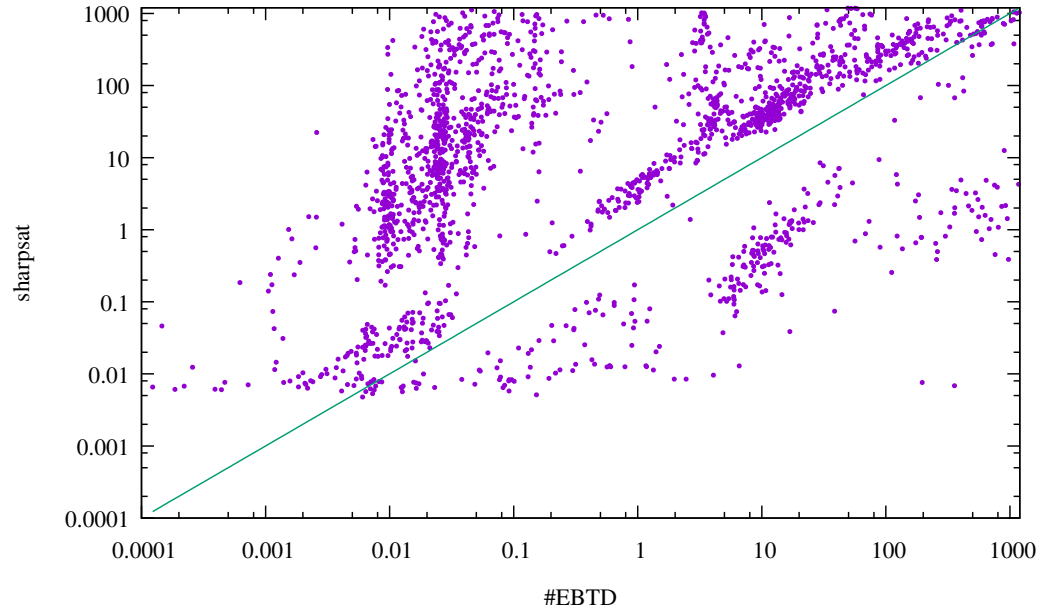

FIGURE 6.14 – Comparaison des temps de résolution de $\#EBTD$ et de sharpsat pour les 1 883 instances résolues à la fois par les deux algorithmes sur le benchmark I_{direct}.

relsat, *cachet* et *sharpsat*. La comparaison du comportement de $\#EBTD$ selon la décomposition utilisée révèle que les décompositions qui visent à minimiser w^+ ainsi que la décomposition H_2 sont les plus efficaces vis-à-vis du comptage du nombre exact de solutions d'une instance. Ainsi, la décomposition $Min\text{-}Fill\text{-}MG$ qui détériorait significativement l'efficacité de BTD pour la résolution d'instances CSP, prouve un intérêt majeur dans le cadre du comptage. Malgré la liberté de choix de variables dont profite $\#RFL$, la

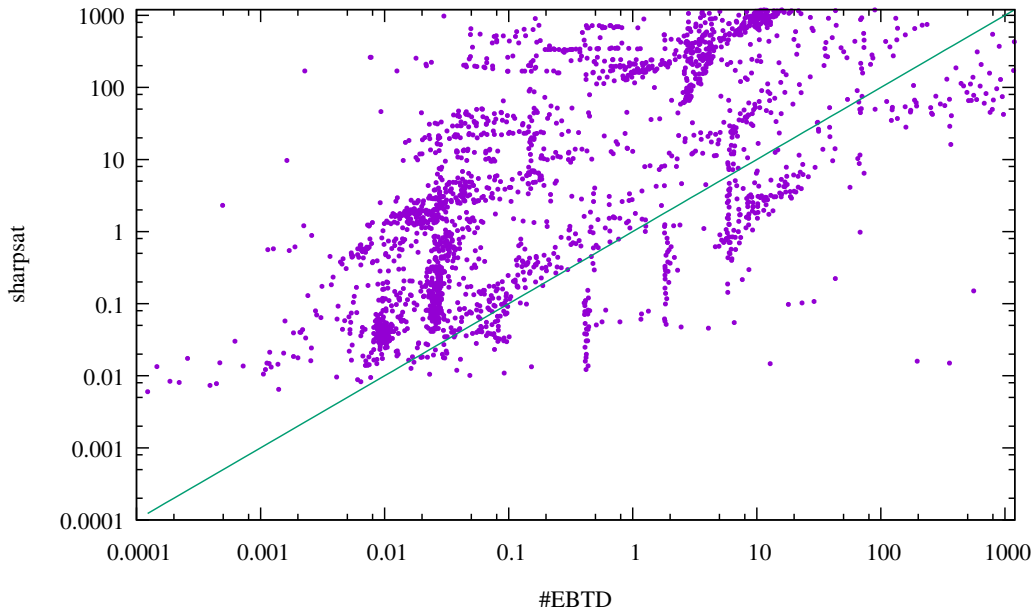

FIGURE 6.15 – Comparaison des temps de résolution de $\#EBTD$ et de sharpsat pour les 2 231 instances résolues à la fois par les deux algorithmes sur le benchmark I_{tuple}.

saturation de la capacité de l'énumération et la redondance des sous-espaces de recherche visités empêche souvent $\#RFL$ de résoudre des instances ayant un très grand nombre de solutions. Ces problèmes sont évités par une méthode à base de décomposition. Dans cette catégorie, la comparaison entre $\#BTD$ et $\#EBTD$ montre que $\#EBTD$ surpasse $\#BTD$ en nombre d'instances et en temps de résolution. Nous comparons aussi $\#EBTD$ au compilateur *cn*2*mddg* qui s'avère particulièrement efficace. L'avantage de $\#EBTD$ par rapport à *cn*2*mddg* est qu'il est capable de fournir une borne inférieure une fois le temps limite dépassé sans que le nombre exact de solutions soit compté. En plus, $\#EBTD$ est mis en valeur pour des instances difficilement résolues par *cn*2*mddg*. Ceci semble évident du fait que le calcul de la décomposition peut être coûteux ne justifiant pas son utilisation pour des instances dont le nombre de solutions peut être facilement compté par *cn*2*mddg*. Finalement, la comparaison de $\#EBTD$ aux #SAT solveurs montre que ces derniers sont clairement surclassés par $\#EBTD$. Elle met en avant aussi les problèmes de conversion entre le format CSP et le format SAT qui s'avère coûteuse en temps et en espace. Plus important, il semble que la structure des instances CSP sera moins présente lorsque cette dernière est convertie en format SAT.

6.5 Conclusion

Dans ce chapitre, nous avons abordé le problème du comptage. Ce problème est très intéressant, du fait des applications qui en découlent en intelligence artificielle et bien au-delà dans d'autres domaines plus éloignés de l'informatique comme la physique ou la chimie mais surtout parce que ce problème constitue un défi sur le plan théorique et le plan pratique. En effet, sa difficulté en théorie et en pratique nous incite à étudier ce problème de plus près. D'une part, des études théoriques ont été réalisées visant à analyser ce problème du point de vue de la complexité théorique en élaborant des classes traitables [Slivovsky and Szeider, 2013] ou en analysant leur difficulté par le biais des théorèmes de

dichotomie [Bulatov and Dalmau, 2003; Bulatov, 2008; Dyer and Richerby, 2013]. D'autre part, d'un point de vue pratique des méthodes de comptage ont été proposées. La difficulté de ce problème a orienté la plupart des travaux vers les méthodes d'approximation qui calculent soit une approximation du nombre de solutions, soit une borne inférieure sur le nombre exact de solutions [Wei and Selman, 2005; Gomes et al., 2006, 2007a; Gogate and Dechter, 2007; Gomes et al., 2007b; Kroc et al., 2008; Gogate and Dechter, 2008]. Malheureusement, même les méthodes d'approximation rencontrent des difficultés pour pouvoir fournir des approximations de qualité.

Ainsi, nous nous sommes intéressés dans ce chapitre aux méthodes exactes. Afin de proposer des méthodes efficaces en théorie et en pratique, nous exploitons les propriétés structurelles de l'instance. En particulier, nous nous concentrons sur le paramètre de la largeur arborescente car les instances ayant une tree-width bornée par une constante peuvent être résolues en temps polynomial. Le fait d'exploiter la structure du problème fournit des bornes de complexité en temps et en espace comme c'est le cas de $\#BTD$. En plus de son intérêt théorique, $\#BTD$ a prouvé son intérêt pratique en permettant de compter le nombre de solutions des instances ayant un nombre élevé - voire gigantesque - de solutions. Cet algorithme exploite une décomposition arborescente et évite certaines redondances en empêchant l'exploration répétée du même sous-espace de recherche grâce aux enregistrements auxquels il a recours.

Bien que certaines redondances soient évitées, la façon dont $\#BTD$ parcourt la décomposition induit des calculs inutiles qui peuvent être coûteux en temps et en espace. C'est pourquoi, nous avons proposé l'algorithme $\#EBTD$ qui vise à éviter ce défaut. Plus précisément, $\#BTD$ compte le nombre d'extensions d'une affectation pour un sous-problème sans la garantie que cette affectation partielle s'étend de façon cohérente sur tout le problème. Ainsi, l'objectif de $\#EBTD$ est de s'assurer qu'une affectation partielle admet au moins une extension cohérente sur toutes les variables du problème avant de calculer le nombre de solutions d'un sous-problème. Ce faisant, le nombre de solutions d'un sous-problème n'est calculé que si nécessaire.

$\#EBTD$ n'a pas vocation à améliorer la complexité théorique mais l'efficacité pratique. En effet, les expérimentations ont montré qu'il surclasse certains compteurs de l'état de l'art. En particulier, $\#EBTD$ surclasse $\#BTD$, $\#RFL$ et certains compteurs #SAT. La comparaison entre $\#EBTD$ et *cn2mddg* est plus intéressante notamment sur les instances difficilement résolues par *cn2mddg*. Finalement, à travers ces expérimentations, nous pouvons déduire que vu la difficulté du problème du comptage, les décompositions visant à minimiser la largeur w^+ sont désormais mises en valeur. En raison de la difficulté du problème, un temps de calcul de décompositions élevé peut être toléré contrairement au problème de décision. En plus, la minimisation du paramètre central de la complexité théorique semble constituer l'une des clés pour compter efficacement le nombre de solutions d'une instance.

Nous arrivons donc à un constat paradoxal. En effet, la décomposition doit minimiser w^+ et augmenter le nombre de séparateurs pour pouvoir permettre le dénombrement efficace des solutions. Cependant, la résolution doit aussi considérer des décompositions mieux adaptées à la recherche d'une première solution, celles-ci n'étant pas nécessairement celles minimisant la largeur. Concilier ces deux points de vue antagonistes peut être très prometteur et peut contribuer à améliorer considérablement l'efficacité des méthodes du comptage basées sur l'exploitation des décompositions.