Filtrage basé sur des contraintes tous différents pour l'isomorphisme de sous-graphe

Christine Solnon

To cite this version:

Christine Solnon. Filtrage basé sur des contraintes tous différents pour l'isomorphisme de sous-graphe. JFPC 2010 - Sixièmes Journées Francophones de Programmation par Contraintes, Jun 2010, Caen, France. pp.247-256. inria-00519102

HAL Id: inria-00519102
https://inria.hal.science/inria-00519102

Submitted on 18 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Filtrage basé sur des contraintes *tous différents* pour l'isomorphisme de sous-graphe

Christine Solnon

LIRIS, CNRS UMR 5205, Université de Lyon, Université Lyon 1

christine.solnon@liris.cnrs.fr

Résumé

Le problème de l'isomorphisme de sous-graphe consiste à rechercher une copie d'un graphe motif dans un graphe cible. Ce problème peut être résolu par une exploration exhaustive combinée avec des techniques de filtrage visant à élaguer l'espace de recherche. Cet article introduit un nouvel algorithme de filtrage basé sur des contraintes *tous différents* conditionnelles. Nous montrons que ce filtrage est plus fort que les autres filtrages, dans le sens où il coupe plus de branches, et qu'il est également plus efficace, dans le sens où il permet de résoudre de nombreuses instances plus rapidement.

1 Introduction

Les graphes sont utilisés dans de nombreuses applications pour représenter des objets structurés tels que, par exemple, des molécules, des images ou des réseaux biologiques. Dans beaucoup de ces applications, on cherche une copie d'un graphe motif dans un graphe cible. Ce problème, connu sous le nom d'isomorphisme de sous-graphe, est NP-complet dans le cas général [5].

Le problème d'isomorphisme de sous-graphe peut être résolu par une exploration systématique combinée avec des techniques de filtrage visant à réduire l'espace de recherche. Différents niveaux de filtrage peuvent être envisagés ; certains sont plus forts que d'autres (ils réduisent plus fortement l'espace de recherche), mais ont également une complexité en temps supérieure.

Dans cet article, nous décrivons et comparons différents algorithmes de filtrage pour le problème de l'isomorphisme sous-graphe, et nous introduisons un nouvel algorithme de filtrage dont nous montrons qu'il est plus fort. Nous évaluons expérimentalement ce nouvel algorithme de filtrage sur un benchmark de près de 2000 instances, et nous montrons qu'il est beaucoup plus efficace pour une large majorité d'instances.

2 Définitions et notations

Un graphe $G = (N, E)$ se compose d'un ensemble de nœuds N et un ensemble d'arêtes $E \subseteq N \times N$. L'ensemble des voisins d'un nœud u est noté $adj(u)$ et est défini par $adj(u) = \{u' \mid (u, u') \in E\}$. Dans cet article, on considère implicitement des graphes non orientés, de telle sorte que $(u, u') \in E \Leftrightarrow (u', u) \in E$. L'extension de notre travail à des graphes orientés est directe et peut être trouvée dans [15].

Le problème de l'isomorphisme de sous-graphe entre un graphe motif $G_p = (N_p, E_p)$ et un graphe cible $G_t = (N_t, E_t)$ consiste à décider si G_p est isomorphe à un sous-graphe de G_t. Plus précisément, il s'agit de trouver une fonction injective $f : N_p \rightarrow N_t$, qui associe un nœud cible différent à chaque nœud motif, et qui préserve les arêtes du motif, à savoir, $\forall (u, u') \in E_p, (f(u), f(u')) \in E_t$. La fonction f est appelée *fonction de sous-isomorphisme*.

Notons que le sous-graphe n'est pas nécessairement induit de sorte que deux nœuds motifs qui ne sont pas reliés par une arête peuvent être appariés à deux nœuds cibles qui sont reliés par une arête.

Dans la suite, nous supposons sans perte de généralité que $N_t \cap N_p = \emptyset$. Nous notons u ou u' (resp. v ou v') les nœuds de G_p (resp. G_t), et $\#S$ la cardinalité d'un ensemble S. Nous définissons $n_p = \#N_p$, $n_t = \#N_t$, $e_p = \#E_p$, $e_t = \#E_t$ et d_p et d_t les degrés maximaux des graphes G_p et G_t.

3 Filtrages pour l'isomorphisme de sous-graphes

Dans cette section, nous montrons d'abord comment modéliser un problème d'isomorphisme de sous-graphe en un problème de satisfaction de contraintes (CSP). Ensuite, nous décrivons différents algorithmes de fil-

trage pour l'isomorphisme de sous-graphe dans les sections 3.2 à 3.5, et nous les comparons à la section 3.6.

3.1 Modélisation CSP

Un problème d'isomorphisme de sous-graphe peut être modélisé comme un CSP en associant une variable (notée x_u) à chaque nœud motif $u \in N_p$. Le domaine d'une variable x_u (noté D_u) contient l'ensemble des nœuds cibles qui peuvent être appariés à u. Intuitivement, l'affectation d'une variable x_u à une valeur v correspond à l'appariement du nœud motif u au nœud cible v. Le domaine D_u est généralement réduit à l'ensemble des nœuds cibles dont le degré est supérieur ou égal au degré de u car un nœud u ne peut être apparié à un nœud v que si $\#adj(u) \leq \#adj(v)$.

Les contraintes garantissent que l'affectation des variables à des valeurs correspond à une fonction de sous-isomorphisme. Il y a deux types de contraintes : (1) les *contraintes d'arête* garantissent que les arêtes motifs sont préservées, *i.e.*, $\forall (u, u') \in E_p, (x_u, x_{u'}) \in E_t$; et (2) les *contraintes de différence* garantissent que l'affectation correspond à une fonction injective, *i.e.*, $\forall (u, u') \in N_p^2, u \neq u' \Rightarrow x_u \neq x_{u'}$.

Un tel CSP peut être résolu en construisant un arbre de recherche. La taille de cet arbre peut être réduite en utilisant des techniques de filtrage qui propagent les contraintes pour supprimer des valeurs des domaines. Nous décrivons dans les sections 3.2 à 3.5 différentes techniques de filtrage qui peuvent être utilisées pour résoudre les problèmes d'isomorphisme de sous-graphe. Certains de ces filtrages (*FC(Diff)*, *GAC(AllDiff)*, *FC(Edges)* et *AC(Edges)*)) sont génériques et peuvent être utilisés pour résoudre n'importe quel CSP tandis que d'autres (*LV2002* et *ILF (k)*) sont dédiés au problème de l'isomorphisme de sous-graphe.

3.2 Propagation des contraintes de différence

Une propagation simple par vérification en avant des contraintes de différence (notée *FC(Diff)*) peut être faite à chaque fois qu'un nœud motif u est apparié à un nœud cible v en supprimant v des domaines de tous les nœuds non appariés. Cela peut être fait en $\mathcal{O}(n_p)$.

FC(Diff) propage chaque contrainte binaire de différence séparément. Un filtrage plus fort peut être obtenu en propageant l'ensemble des contraintes de différence de façon globale, assurant ainsi que chaque nœud motif peut être apparié à un nœud cible différent. Il s'agit de la cohérence d'arc généralisée (notée *GAC(AllDiff)*). Régin a montré dans [13] comment utiliser l'algorithme de couplage maximal dans un graphe bipartie proposé par Hopcroft et Karp pour assurer *GAC(AllDiff)* en $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Exemple 1 *Considérons 4 variables x_1, x_2, x_3 et x_4 telles que $D_1 = \{a\}$, $D_2 = D_3 = \{a, b, c\}$ et $D_4 = \{a, b, c, d\}$.* FC(Diff) *enlève a des domaines de x_2, x_3 et x_4.* GAC(AllDiff) *enlève en plus b et c de D_4 car si x_4 est affectée à b ou c, alors x_2 ne peut plus être affectée à une valeur différente de celles de x_3 et x_4.*

3.3 Propagation des contraintes d'arête

Une propagation simple par vérification en avant des contraintes d'arête (notée *FC(Edges)*) peut être effectuée à chaque fois qu'un nœud motif u est apparié à un nœud cible v en enlevant du domaine de chaque nœud adjacent à u tout nœud cible n'étant pas adjacent à v. Cela peut être fait en $\mathcal{O}(d_p \cdot n_t)$.

Il est possible d'aller plus loin et de vérifier, pour chaque arête motif (u, u') et chaque nœud $v \in D_u$, qu'il existe au moins un nœud $v' \in D_{u'}$ qui soit adjacent à v. Le nœud cible v' est dit *support* de l'appariement (u, v) pour l'arête (u, u'). Si un appariement (u, v) n'a pas de support pour une arête, alors v peut être enlevé de D_u. Une telle propagation des contraintes d'arête est itérée jusqu'à ce que plus aucune valeur ne puisse être enlevée, assurant ainsi la *cohérence d'arc* des contraintes d'arête (notée *AC(Edges)*), *i.e.*,
$\forall (u, u') \in E_p, \forall v \in D_u, \exists v' \in D_{u'}, (v, v') \in E_t$.

La cohérence d'arc a été très largement étudiée et différents algorithmes de filtrage ont été proposés, ayant différentes complexité en temps et en espace. Par exemple, AC4 [12] a une complexité en temps et en espace de $\mathcal{O}(c \cdot k^2)$, où c est le nombre de contraintes et k la taille du plus grand domaine. Ainsi, la complexité de *AC(Edges)* est $\mathcal{O}(e_p \cdot n_t^2)$ quand on considère AC4.

Exemple 2 *Considérons l'instance de la figure 1 (qui n'a pas de solution). Supposons que le nœud 3 ait été apparié au nœud E et que E ait été enlevé des domaines des autres nœuds motifs par* FC(Diff) *ou*

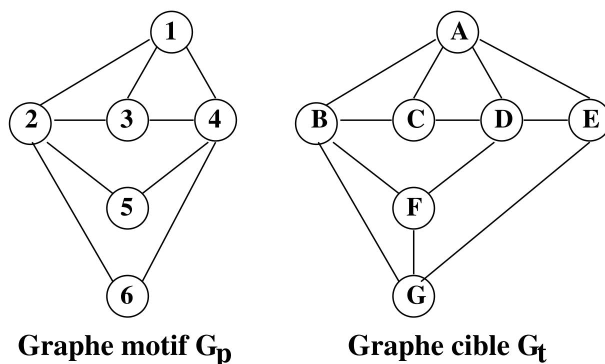


FIGURE 1 – Exemple d'instance.

GAC(AllDiff). FC(Edges) *enlève B, C et F des domaines des nœuds* 1, 2 *et* 4 *car B, C et F ne sont pas adjacents à E tandis que* 1, 2 *et* 4 *sont adjacents à* 3. AC(Edges) *enlève en plus G de D_1 car l'appariement* $(1, G)$ *n'a pas de support pour l'arête* $(1, 4)$ *(aucun des nœuds adjacents à G n'appartient à D_4).* AC(Edges) *enlève également G de D_2 et D_4.*

3.4 Propagation d'un ensemble de contraintes d'arête

FC(Edges) et *AC(Edges)* propagent chaque contrainte d'arête séparément. Un filtrage plus fort est obtenu en propageant les contraintes d'arête de façon plus globale, i.e., en propageant le fait que plusieurs nœuds doivent être adjacents à un nœud donné. En effet, un nœud motif u ne peut être apparié à un nœud cible v que si le nombre de nœuds adjacents à u est inférieur ou égal au nombre de nœuds cibles qui sont à la fois adjacents à v et appartiennent au domaine d'un nœud adjacent à u. Ainsi, Larrosa et Valiente ont proposé dans [8] un algorithme de filtrage (noté *LV2002*) qui propage cette contrainte. Plus précisément, ils définissent l'ensemble $\mathcal{F}(u, v) = \cup_{u' \in adj(u)}(D_{u'} \cap adj(v))$, qui est un sur-ensemble de l'ensemble des nœuds pouvant être appariés à des nœuds adjacents à u dans l'hypothèse où u serait apparié à v. Par conséquent, v peut être enlevé de D_u dès lors que $\#\mathcal{F}(u, v) < \#adj(u)$. Il est également possible d'enlever v de D_u dès lors qu'il existe un nœud motif $u' \in adj(u)$ tel que $D_{u'} \cap adj(v) = \emptyset$, assurant ainsi la cohérence d'arc des contraintes d'arête. L'algorithme *LV2002* a une complexité en temps de $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Exemple 3 *Considérons de nouveau l'instance de la figure 1 et supposons que le nœud* 3 *ait été apparié au nœud E et que E ait été enlevé des domaines des autres nœuds motifs. Comme* AC(Edges), LV2002 *enlève les nœuds B, C, F et G de D_1, D_2 et D_4. Il enlève également les valeurs A et D de D_1. En effet,*

$$\begin{aligned} \mathcal{F}(1, A) &= (D_2 \cup D_3 \cup D_4) \cap adj(A) = \{D, E\} \\ \mathcal{F}(1, D) &= (D_2 \cup D_3 \cup D_4) \cap adj(D) = \{A, E\} \end{aligned}$$

Comme $\#\mathcal{F}(1, A) < \#adj(1)$ et $\#\mathcal{F}(1, D) < \#adj(1)$, à la fois A et D sont enlevés de D_1 qui devient vide de sorte qu'une incohérence est détectée.

3.5 Iterated Labelling Filtering (ILF(k))

Zampelli *et al* ont proposé dans [17] un algorithme de filtrage (appelé *ILF(k)*) qui exploite la structure des graphes de façon globale afin de calculer des étiquettes qui sont associées aux nœuds et qui sont utilisées pour filtrer les domaines. Plus précisément, une relation de compatibilité est définie entre les étiquettes et est utilisée pour enlever du domaine d'un nœud motif tout nœud cible dont l'étiquette n'est pas compatible.

ILF(k) est une procédure itérative qui part d'un étiquetage initial. Cet étiquetage initial peut être défini par les degrés des nœuds. Dans ce cas, la relation de compatibilité est l'ordre $\leq$ classique. Cet étiquetage initial peut être étendu pour filtrer plus de valeurs. Etant donné un étiquetage l et une relation de compatibilité $\preceq$ entre étiquettes de l, l'étiquetage étendu l' étiquette un nœud u par le multi-ensemble des étiquettes des nœuds adjacents à u. La relation de compatibilité $\preceq'$ entre ces nouvelles étiquettes est telle que $l'(u) \preceq' l'(v)$ si pour chaque occurrence x d'une étiquette de $l'(u)$ il existe une occurrence différente y d'une étiquette de $l'(v)$ telle que $x \preceq y$. Un point clé réside dans le calcul de cette nouvelle relation de compatibilité $\preceq'$, qui est effectué en $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$ grâce à l'algorithme de couplage maximal de Hopcroft et Karp (voir [17] pour plus de détails).

Une telle extension d'étiquetage peut être itérée. Un paramètre k est introduit afin de borner le nombre d'extensions d'étiquetage. Notons que ce processus itératif d'extension peut être arrêté avant d'atteindre cette borne k si un domaine est devenu vide ou si un point fixe est atteint (tel que plus aucune valeur ne pourra être enlevée). La complexité en temps de *ILF(k)* est $\mathcal{O}(min(k, n_p \cdot n_t) \cdot n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

Exemple 4 *Considérons de nouveau l'instance de la figure 1. L'étiquetage initial basé sur les degrés est*

- $l(5) = l(6) = 2$
- $l(1) = l(3) = l(C) = l(E) = l(F) = l(G) = 3$
- $l(2) = l(4) = l(A) = l(B) = l(D) = 4$

et l'ordre sur ces étiquettes est tel que 2 *est compatible avec* 2, 3 *et* 4 *;* 3 *est compatible avec* 3 *et* 4 *; et* 4 *est compatible avec* 4. *Ainsi, les nœuds C, E, F et G peuvent être enlevés de D_2 et D_4.*

L'extension de cet étiquetage initial est telle que

- $l'(1) = l'(3) = l'(E) = l'(F) = \{\{3, 4, 4\}\}$
- $l'(2) = l'(4) = \{\{2, 2, 3, 3\}\}$
- $l'(5) = l'(6) = \{\{4, 4\}\}$
- $l'(A) = \{\{3, 3, 4, 4\}\}$
- $l'(B) = l'(D) = \{\{3, 3, 3, 4\}\}$
- $l'(C) = \{\{4, 4, 4\}\}$
- $l'(G) = \{\{3, 3, 4\}\}$

et l'ordre sur ces nouvelles étiquettes est tel que $\{\{3, 4, 4\}\}$ *est compatible avec* $\{\{3, 3, 4, 4\}\}$ *et* $\{\{3, 4, 4\}\}$ *;* $\{\{2, 2, 3, 3\}\}$ *est compatible avec* $\{\{3, 3, 4, 4\}\}$ *et* $\{\{3, 3, 3, 4\}\}$ *; et* $\{\{4, 4\}\}$ *est compatible avec* $\{\{3, 3, 4, 4\}\}$, $\{\{4, 4, 4\}\}$ *et* $\{\{3, 4, 4\}\}$. *Comme $l'(1)$ n'est pas compatible avec $l'(B)$, B est enlevé de D_1. De même, B, D et G sont enlevés de D_1, D_3, D_5 et D_6. Ce nouvel étiquetage l' peut de nouveau être étendu,*

enlevant ainsi de nouvelles valeurs et prouvant finalement l'incohérence de cette instance.

3.6 Discussion

La plupart des algorithmes qui ont été proposés pour résoudre le problème d'isomorphisme de sous-graphe peuvent être décrits par rapport aux algorithmes de filtrage décrits dans les sections 3.2 à 3.5. En particulier, [11] combine *FC(Diff)* et *FC(Edges)* ; [16] combine *FC(Diff)* et *AC(Edges)* ; [14] combine *GAC(AllDiff)* et *AC(Edges)* ; [8] combine *GAC(AllDiff)* et *LV2002* ; et [17] combine *GAC(AllDiff)*, *AC(Edges)* et *ILF(k)*.

Ces différents filtrages assurent différentes cohérences. Certaines sont plus fortes que d'autres. En particulier, *GAC(AllDiff)* est plus fort que *FC(Diff)* tandis que *LV2002* est plus fort que *AC(Edges)* qui est plus fort que *FC(Edges)*. Cependant, *GAC(AllDiff)* et *FC(Diff)* ne sont pas comparables avec *FC(Edges)*, *AC(Edges)*, *LV2002* et *ILF(k)* car ils ne propagent pas les mêmes contraintes.

Les relations entre *ILF(k)* et les autres filtrages propageant des contraintes d'arête (*i.e.*, *LV2002*, *AC(Edges)* et *FC(Edges)*) dépendent des domaines initiaux : si le domaine initial de chaque variable contient tous les nœuds cibles, alors *ILF(k)* est plus fort que *LV2002*, dès lors que le nombre d'extensions k est supérieur ou égal à 2. Cependant, si des domaines ont été réduits (ce qui est souvent le cas quand le filtrage est effectué à un nœud ne se trouvant pas à la racine de l'arbre de recherche), alors *ILF(k)* n'est pas comparable avec *LV2002* et *AC(Edges)*.

En effet, *ILF(k)* n'exploite pas les domaines pour filtrer les valeurs : les étiquettes calculées ne dépendent pas des domaines des variables mais uniquement de la structure des graphes. Afin de propager plus de réductions de domaines, une possibilité consiste à démarrer le processus itératif d'extension à partir d'un étiquetage initial intégrant complètement les réductions de domaines dans la relation de compatibilité de sorte que si un nœud cible v n'appartient pas au domaine d'un nœud motif u alors l'étiquette de v n'est pas compatible avec l'étiquette de u. Avec un tel étiquetage initial (appelé l_{dom} dans [17]), *ILF(k)* est plus fort que *LV2002* dès lors que $k \geq 1$. Cependant, si ce filtrage est plus fort, il est également beaucoup plus coûteux car la complexité en pratique de *ILF(k)* dépend du nombre d'étiquettes différentes. En effet, la complexité théorique d'une itération de *ILF(k)* correspond au pire des cas où tous les nœuds ont des étiquettes différentes. Si les nombres d'étiquettes différentes de nœuds motifs et cibles sont respectivement l_p et l_t, alors la complexité d'une itération de *ILF(k)* est $\mathcal{O}(e_t + l_p \cdot l_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

4 Filtrage LAD

Le nouveau filtrage proposé dans cet article exploite le fait que, pour chaque fonction d'isomorphisme f et chaque nœud motif $u \in N_p$, nous avons :

1. $\forall u' \in adj(u), f(u') \in adj(f(u))$
2. $\forall (u', u'') \in adj(u)^2, u' \neq u'' \Rightarrow f(u') \neq f(u'')$

Ces deux propriétés sont des conséquences directes du fait que toute fonction de sous-isomorphisme (1) préserve les arêtes et (2) est une injection. Par rapport au CSP associé à un problème d'isomorphisme de sous-graphe, ces deux propriétés peuvent être exprimées par la contrainte conditionnelle suivante, appelée *contrainte de voisinage* :

$$x_u = v \Rightarrow \quad \forall u' \in adj(u), x_{u'} \in adj(v) \\ \wedge allDiff(\{x_{u'} | u' \in adj(u)\})$$

Cette contrainte de voisinage peut être propagée en recherchant un couplage maximal dans un graphe bipartie, comme proposé par Régin pour la contrainte globale *tous différents* [13]. Rappelons qu'un couplage d'un graphe $G = (N, E)$ est un sous-ensemble d'arêtes $m \subseteq E$ tel qu'il n'y ait pas deux arêtes de m partageant une même extrémité. Un couplage $m \subseteq E$ couvre un ensemble de nœuds N_i si chaque nœud de N_i est une extrémité d'une arête de m.

Pour chaque couple de nœuds (u, v) tel que $v \in D_u$, nous définissons le graphe bipartie qui associe un nœud à chaque nœud adjacent à u ou v et une arête à chaque couple (u', v') tel que $v' \in D_{u'}$.

Définition 1 *Etant donné $(u, v) \in N_p \times N_t$ tel que $v \in D_u$, le graphe bipartie $G_{(u,v)} = (N_{(u,v)}, E_{(u,v)})$ est tel que $N_{(u,v)} = adj(u) \cup adj(v)$ et $E_{(u,v)} = \{(u', v') \in adj(u) \times adj(v) \mid v' \in D_{u'}\}$*

S'il n'existe pas de couplage de $G_{(u,v)}$ couvrant $adj(u)$, alors les nœuds adjacents à u ne peuvent pas être appariés à des nœuds différents qui soient adjacents à v et, par conséquent, v peut être enlevé de D_u. Ce filtrage doit être itéré : quand v est enlevé de D_u, l'arête (u, v) est enlevée des autres graphes biparties de sorte que certains graphes biparties peuvent ne plus avoir de couplage couvrant. Un point clé pour une implémentation efficace de ce filtrage réside dans le fait que l'arête (u, v) n'appartient qu'aux graphes biparties $G_{(u',v')}$ tels que $u' \in adj(u)$ et $v' \in adj(v) \cap D(u')$.

Exemple 5 *Considérons l'instance de la figure 2 et définissons les domaines initiaux par rapport aux degrés des nœuds, i.e.,*

$$D_1 = D_3 = D_5 = D_6 = \{A, B, C, D, E, F, G\} \\ D_2 = D_4 = \{A, B, D\}$$

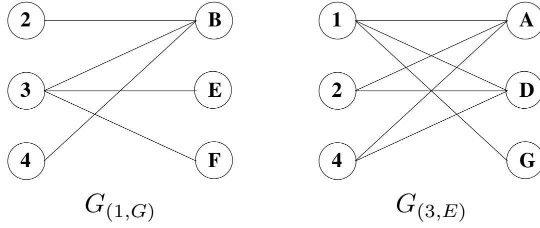


FIGURE 2 – Graphes biparties.

Le graphe bipartie $G_{(1,G)}$ est dessiné dans la partie gauche de la figure 2. Il n'existe pas de couplage de ce graphe couvrant $adj(1)$ car à la fois 2 et 4 ne peuvent être couplé qu'à B. Par conséquent, G peut être enlevé de D_1. Notons que, sur cet exemple, le filtrage LV2002 *ne peut enlever G de D_1 étant donné que $\mathcal{F}(1, G) = (D_2 \cup D_3 \cup D_4) \cap adj(G) = \{B, E, F\}$ de sorte que $\#\mathcal{F}(1, G) \geq \#adj(1)$. Notons également qu'une simple contrainte* allDiff *sur l'ensemble de variables $\{x_2, x_3, x_4\}$ ne permet pas d'enlever G à D_1 : il faut combiner cette contrainte* allDiff *avec le fait que, si 1 est apparié à G, alors 2, 3 et 4 doivent être appariés à des nœuds adjacents à G.*

*Le graphe bipartie $G_{(3,E)}$ est dessiné dans la partie droite de la figure 2. Il existe un couplage de ce graphe qui couvre $adj(3)$ (*e.g., *$m = \{(1, G), (2, A), (4, D)\}$) de sorte que E n'est pas enlevé de D_3. Cependant, dès lors que G a été enlevé de D_1, l'arête $(1, G)$ est enlevée de $G_{(3,E)}$ et il n'y a plus de couplage couvrant $adj(3)$ (car à la fois 1, 2 et 3 ne peuvent être couplés qu'à A et D). Ainsi, E est aussi enlevé de D_3.*

L'algorithme 1 décrit la procédure de filtrage correspondante, appelée LAD (Local All Different). Cette procédure prend en entrée un ensemble S de couples de nœuds motif/cible à filtrer. A la racine de l'arbre de recherche, cet ensemble devrait contenir tous les couples possibles, *i.e.*, $S = \{(u, v) \mid u \in N_p, v \in D_u\}$. Ensuite, à chaque nouveau point de choix, S devrait être initialisé avec l'ensemble des couples (u, v) tels que $v \in D_u$ et un nœud adjacent à v a été enlevé du domaine d'un nœud adjacent à u depuis le dernier appel à LAD.

Pour chaque couple (u, v) appartenant à S, LAD vérifie qu'il existe un couplage de $G_{(u,v)}$ couvrant $adj(u)$. Si ce n'est pas le cas, v est enlevé de D_u et tous les couples (u', v') tels que u' est adjacent à u et v' est adjacent à v et appartient à $D_{u'}$ sont ajoutés à S.

Un point clé réside dans l'implémentation de la procédure vérifiant qu'il existe un couplage couvrant de $G_{(u,v)}$. Nous utilisons pour cela l'algorithme de Hopcroft et Karp [7] dont la complexité en temps est $\mathcal{O}(d_p \cdot d_t \cdot \sqrt{d_t})$. Cette complexité peut être amélio-

Algorithm 1: Filtrage LAD

Entrées: Un ensemble S de couples de nœuds motif/cible à filtrer

Sorties: Echec (si une incohérence est détectée) ou Succès. En cas de succès, les domaines sont filtrés de sorte que $\forall u \in N_p, \forall v \in D_u$, il existe un couplage de $G_{(u,v)}$ qui couvre $adj(u)$.

1 **tant que** $S \neq \emptyset$ **faire**
2   Enlever un couple (u, v) de S
3   **si** *il n'existe pas de couplage de $G_{(u,v)}$ couvrant $adj(u)$* **alors**
4   Enlever v de D_u
5   **si** $D_u = \emptyset$ **alors retourner** Echec
6       $S \leftarrow S \cup \{(u', v') \mid u' \in adj(u), v' \in adj(v) \cap D_{u'}\}$
7 **retourner** Succès

rée en exploitant le fait que l'algorithme de Hopcroft et Karp est incrémental : partant d'un couplage vide, il calcule itérativement de nouveaux couplages contenant de plus en plus d'arêtes jusqu'à ce que le couplage soit maximal. Chaque itération consiste en un parcours en largeur et se fait en $\mathcal{O}(d_p \cdot d_t)$ tandis que le nombre d'itérations est en $\mathcal{O}(\sqrt{d_t})$. Cependant, si l'algorithme débute d'un couplage contenant déja k arêtes et si le couplage maximal a l arêtes alors le nombre d'itérations est également borné par $l - k$.

Nous utilisons cette propriété pour améliorer la complexité de LAD. Plus précisément, pour chaque nœud motif $u \in N_p$ et chaque nœud cible $v \in D_u$, nous mémorisons le dernier couplage de $G_{(u,v)}$ calculé. La complexité en espace pour mémoriser tous les couplages couvrants est $\mathcal{O}(n_p \cdot n_t \cdot d_p)$ car il y a $n_p \cdot n_t$ graphes biparties et le couplage couvrant de $G_{(u,v)}$ comporte $\#adj(u)$ arêtes. Il serait trop coûteux, à la fois en temps et en mémoire, de recopier tous ces couplages à chaque point de choix et de restituer ces couplages après chaque retour en arrière. Ainsi, nous ne recopions pas les couplages à chaque point de choix, mais nous les mettons simplement à jour à chaque fois que nous devons vérifier qu'il existe un couplage couvrant de $G_{(u,v)}$: nous mettons tout d'abord à jour le dernier couplage calculé en enlevant les couples (u', v') tels que v' n'appartient plus à $D_{u'}$; si aucun arête n'a été enlevée, alors le couplage reste valide ; sinon Hopcroft Karp est utilisé pour compléter le couplage ; si un nouveau couplage couvrant est trouvé, alors il est mémorisé.

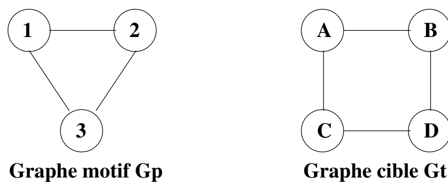
Théorème 1 *La complexité en temps de* LAD *est $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$.*

Preuve.

- La complexité du calcul d'un premier couplage couvrant pour chaque graphe bipartie est $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$; cette étape est effectuée une seule fois, à la racine de l'arbre de recherche.
- Chaque fois qu'une valeur v est enlevée d'un domaine D_u, les couplages de tous les graphes $G_{(u',v')}$ tels que $u' \in adj(u)$ et $v' \in D_{u'} \cap adj(v)$ sont mis-à-jour. Il y a $d_p \cdot d_t$ graphes biparties dans le pire des cas et chaque mise-à-jour est faite incrémentalement en $\mathcal{O}(d_p \cdot d_t)$.
- Dans le pire des cas, seulement une valeur est enlevée lors de la mise à jour des couplages couvrants de tous les voisins et $n_p \cdot n_t$ valeurs peuvent être supprimées.

Nous montrons dans [15] que le filtrage *LAD* établit la cohérence d'arc généralisée des contraintes de voisinage, notée *GAC(voisinages)*. Nous comparons également dans [15] cette cohérence partielle avec les autres cohérences partielles introduites en 3. En particulier, nous montrons que *GAC(voisinages)* est plus forte que *LV2002* et qu'elle est équivalente à *ILF(k)* quand l'étiquetage initial est l_{dom} et quand $k = \infty$. Cependant, le filtrage *LAD*, qui établit *GAC(voisinages)*, a une complexité en temps inférieure. En effet, *ILF(k)* recalcule tous les couplages, pour tous les couples de nœuds motif/cible, à chaque itération, tandis que *LAD* ne met à jour que les couplages qui ont été effectivement impactés par une réduction de domaine.

Enfin, nous montrons dans [15] que *GAC(voisinage)* est moins forte que la cohérence d'arc singleton (SAC) [1] des contraintes d'arêtes combinées à une contrainte globale *tous différents* (notée *SAC(Edges+AllDiff)*). En effet, *SAC(Edges+AllDiff)* assure que, $\forall u \in N_p, \forall v \in D_u$, si D_u est réduit au singleton $\{v\}$, alors *AC(Edges)* combinée avec *GAC(AllDiff)* ne détecte pas d'incohérence. Dans ce cas, il existe un couplage de $G_{(u,v)}$ qui couvre $adj(u)$ de sorte que *GAC(voisinage)* est également assurée. *SAC(Edges+AllDiff)* est en fait strictement plus forte que *GAC(Neighborhood)*. Considérons par exemple l'instance suivante :



Supposons que les domaines initiaux soient $D_1 = D_2 = D_3 = \{A, B, C, D\}$. *GAC(Neighborhood)* ne réduit aucun domaine car chaque graphe bipartie $G_{(u,v)}$ admet un couplage couvrant $adj(u)$. Cependant, *SAC(Edges+AllDiff)* détecte l'incohérence : si D_1 est réduit à $\{A\}$, alors *AC(Edges)* réduit D_2 et D_3 à l'ensemble des nœuds adjacents à A (i.e., à $\{C, B\}$) et l'arête $(3, 2)$ n'a plus de support (car G_t n'a pas d'arête entre C et B) de sorte que *AC(Edges)* détecte une incohérence.

Cependant, la complexité en temps optimale pour assurer la cohérence d'arc singleton d'un CSP binaire est $\mathcal{O}(n \cdot d^3 \cdot e)$ où e est le nombre de contraintes, n le nombre de variables et d la taille des domaines [1]. Si l'on considère le CSP binaire comportant uniquement les contraintes d'arêtes, nous avons $n = n_p$, $d = n_t$ et $e = e_p$ de sorte que *SAC(Edges)* est établie en $\mathcal{O}(n_p \cdot n_t^3 \cdot e_p)$. Dans le pire des cas (si les graphes sont complets), *SAC(Edges)* a la même complexité que *LAD*, à savoir $\mathcal{O}(n_p^3 \cdot n_t^3)$. Cependant, pour des graphes moins denses, *LAD* a une complexité en temps inférieure à *SAC(Edges)*.

5 Evaluation expérimentale

5.1 Jeu d'essai

Nous considérons 1993 instances provenant de 3 bases de données différentes.

Base de données "scale-free" (classes sf-d-D-n et si-d-D-n) Cette base de données a été utilisée dans [17] pour évaluer *ILF(k)*. Les graphes de ces instances sont des réseaux scale-free qui ont été générés aléatoirement en suivant une loi de distribution exponentielle pour les degrés $P(d = k) = k^{-2,5}$ (voir [17] pour plus de détails). Il y a 5 classes. Chacune des 4 premières, notées *sf-d-D-n*, contient 20 instances satisfiables telles que le graphe cible a n nœuds dont les degrés sont bornés entre d et D, et le graphe motif est généré en sélectionnant 90% des nœuds et arêtes du graphe cible. La cinquième classe, notée *si-d-D-n*, contient 20 instances non satisfiables qui ont été générées comme les instances des autres classes, sauf que 10% de nouvelles arêtes ont été ajoutées au graphe motif afin d'obtenir des instances non satisfiables.

Base de données "GraphBase" (classe LV) Cette base de données a été utilisée dans [8] pour évaluer *LV2002*. Elle contient 113 graphes non orientés de natures variées. Nous avons considéré les 50 premiers graphes, ayant de 10 à 128 nœuds. A partir de ces 50 graphes, nous avons généré 793 instances en considérant tous les couples de graphes (G_p, G_t) qui ne sont pas trivialement résolus, *i.e.*, tels que $e_p > 0$, $n_p \leq n_t$ et $d_p \leq d_t$.

Base de données "Vflib" (classes bvg-n, bvgm-n, m4D-n, m4Dr-n et r-d-n) Cette base de données a

été utilisée dans [2] pour évaluer Vflib. Elle contient 63 classes d'instances et chaque classe contient des instances telles que le graphe cible a de 20 à 1000 nœuds. Pour chaque classe, nous n'avons considéré que 4 tailles et, pour chaque taille, nous n'avons considéré que les 10 premières instances. Nous avons regroupé les classes comme suit (voir [4] pour plus de détails sur les classes d'origine).

Les classes *bvg-n* contiennent des graphes à degrés fixés et correspondent aux classes si_x-b_y-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $y \in \{3, 6, 9\}$ le degré et $n \in \{100, 200, 400, 800\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *bvg-n* contient 90 instances.

Les classes *bvgm-n* contiennent des graphes obtenus en perturbant légèrement des graphes à degrés fixés et correspondent aux classes si_x-$b_y m$-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $y \in \{3, 6, 9\}$ le degré et $n \in \{100, 200, 400, 800\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *bvgm-n* contient 90 instances.

Les classes *m4D-n* contiennent des graphes correspondant à des maillages réguliers 4D et correspondent aux classes si_x-*m4D-n* où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible et $n \in \{81, 256, 526, 1296\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *m4D-n* contient 30 instances.

Les classes *m4Dr-n* contiennent des graphes correspondant à des maillages irréguliers 4D et correspondent aux classes si_x-*m4D*r_r-*n* où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $r \in \{2, 4, 6\}$ le degré d'irrégularité et $n \in \{81, 256, 526, 1296\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *m4Dr-n* contient 90 instances.

Les classes *r-p-n* contiennent des graphes générés aléatoirement et correspondent aux classes si_x-*rand*-r_p-*n* où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $p \in \{.01, .05, .1\}$ la probabilité d'ajouter une arête entre 2 nœuds et $n \in \{100, 200, 400, 600\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *r-p-n* contient 90 instances.

5.2 Approches comparées

LAD Le filtrage *LAD* a été implémenté en C et a été intégré dans une procédure de recherche par construction d'un arbre de recherche. A chaque nœud de l'arbre, le prochain nœud motif à affecter est choisi selon l'heuristique *MinDom*, *i.e.*, on choisit le nœud motif qui a le plus petit nombre de nœuds cibles dans son domaine. Un point de choix est créé pour chaque nœud cible appartenant au domaine de la variable à affecter, et ces différents points de choix sont explorés par ordre croissant de valeur. A chaque nœud de l'arbre de recherche, le filtrage *LAD* est combiné avec *GAC (AllDiff)*.

ILF(k) L'implémentation originale de *ILF(k)* était en Gecode. Nous considérons ici une implémentation en C qui utilise les mêmes structures de données et les mêmes heuristiques d'ordre que *LAD* et qui est aussi combinée avec *GAC(AllDiff)*. Cette nouvelle implémentation est beaucoup plus efficace que l'originale. Par exemple, les instances de la classe *sf5-8-1000* sont résolues en 0.19 secondes avec la nouvelle implémentation de *ILF(1)* tandis qu'elles étaient résolues en 11.2 secondes avec l'ancienne implémentation. Nous donnons les résultats obtenus pour $k \in \{1, 2, 4\}$. Nous ne donnons pas de résultats pour $k > 4$ car cela n'améliore jamais les résultats.

Abscon *Abscon* est un solveur de CSP générique écrit en Java par Lecoutre et Tabary (voir [10] pour plus de détails). Nous considérons deux variantes de ce solveur : *Abscon(FC)*, qui correspond à *FC(Edges)* et *FC(Diff)*, et *Abscon(AC)*, qui correspond à *AC(Edges)* combinée à un filtrage des contraintes de différence intermédiaire entre *AC(Diff)* et *GAC(AllDiff)*. Les deux variantes utilisent l'heuristique d'ordre *MinDom* pour choisir la prochaine variable à affecter.

Vflib *Vflib* [2, 3] est une bibliothèque C++ dédiée à la résolution des problèmes d'isomorphisme de graphes et de sous-graphe. Elle effectue une propagation simple par vérification en avant des contraintes d'arête et de différence, mais cette propagation est limitée aux nœuds qui sont adjacents à des nœuds déjà appariés. Elle utilise des heuristiques d'ordre de variable et de valeur spécifiques : les variables et les valeurs sont choisies de sorte que le sous-graphe induit par les nœuds correspondants soit connexe (sauf si le motif ou la cible sont composés de différentes composantes connexes).

5.3 Résultats expérimentaux

Toutes les expérimentations ont été réalisées sur la même machine, à savoir un Intel Xeon E5520 cadencé à 2.26 GHz, et avec la même limite de temps, à savoir une heure pour chaque instance.

Considérons tout d'abord le problème consistant à chercher toutes les solutions à une instance, ce qui permet une comparaison qui dépend moins des heuristiques d'ordre de valeurs. Pour cela, nous avons écarté les instances qui ont trop de solutions : nous n'avons considéré que les classes de la base "scalefree" ainsi que les plus petites classes de la base "vflib". La table 1

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
sf5-8-200	16	72.45	**20**	2.04	**20**	1.75	**20**	**0.00**	**20**	0.02	**20**	0.03	**20**	0.02
sf5-8-600	0	-	**20**	138.10	**20**	135.01	**20**	**0.07**	**20**	0.15	**20**	0.15	**20**	0.29
sf5-8-1000	0	-	**20**	1651.11	**20**	1631.88	**20**	**0.19**	**20**	0.55	**20**	0.59	**20**	0.83
sf20-300-300	0	-	16	386.87	16	474.11	**20**	**0.35**	**20**	5.95	**20**	8.24	**20**	2.56
si20-300-300	0	-	6	823.20	5	1046.91	**20**	132.33	19	30.42	19	48.75	**20**	**27.77**
bvg-100	**90**	**0.02**	**90**	1.99	**90**	2.78	**90**	0.04	**90**	0.07	**90**	0.13	**90**	0.75
bvgm-100	89	6.55	89	11.06	**90**	16.57	**90**	**0.48**	**90**	0.49	**90**	**0.48**	**90**	0.53
m4D-81	**30**	0.09	**30**	1.08	**30**	1.04	**30**	0.03	**30**	0.05	**30**	0.05	**30**	**0.02**
m4Dr-81	**90**	1.65	**90**	3.70	**90**	2.40	**90**	**0.18**	**90**	0.19	**90**	0.20	**90**	**0.18**
r0.01-100	21	83.60	23	121.98	28	322.67	**29**	**158.35**	**29**	170.63	**29**	170.53	**29**	180.24
r0.05-100	2	513.01	22	28.60	**23**	56.78	**23**	135.81	22	107.18	22	108.99	**23**	**19.73**
r0.1-100	0	-	25	64.91	28	218.38	28	217.17	28	242	28	243.12	**29**	**148.38**
Tous	338	13.83	451	118.72	460	144.86	480	34.41	478	31.09	478	32.07	**481**	**22.34**

Table 1 – Comparaison pour le problème consistant à chercher toutes les solutions : nombre d'instances résolues (nb) et temps moyen correspondant.

class	(#sol)	*Abscon* *FC*	*Abscon* *AC*	*ILF* *k=1*	*ILF* *k=2*	*ILF* *k=4*	*LAD*
sf5-8-200	(1.10)	3076	108	5	0	0	0
sf5-8-600	(1.00)	418	418	4	0	0	0
sf5-8-1000	(1.05)	557	557	7	0	0	0
sf20-300-300	(4.45)	397844	29338	38	13	7	0
si20-300-300	(0.00)	913730	61191	15342	62	22	27
bvg-100	(218)	10037	2862	461	391	391	0
bvgm-100	(145855)	8977	95618	641	379	222	1
m4D-81	(1253)	1904	327	701	669	652	23
m4Dr-81	(30642)	22920	23592	1356	1304	1300	12
r0.01-100	(57291325)	38853	6749220	10621	6717	6175	60
r0.05-100	(6062230)	233044	615882	2857279	539522	539167	5243
r0.1-100	(30501838)	819714	1985225	2227792	2224579	2224408	320067

Table 2 – Comparaison du nombre de nœuds échecs (moyenne sur les instances résolues) ; la colonne #sol donne le nombre moyen de solutions des instances.

montre que pour ces classes, *LAD* a résolu 1 (resp. 3, 3, 23, 29 et 143) instance de plus que *ILF(1)* (resp. *ILF(2), ILF(4), Abscon(AC), Abscon(FC)* et *Vflib*. *LAD* est plus lent que *ILF* pour les classes *sf-5-8-** et *bvg*, mais ces instances sont en fait très faciles et *LAD* les résout en moins d'une seconde. En revanche, pour les classes plus difficiles comme *si20-300-300*, *r0.05-100* et *r0.1-100*, *LAD* est significativement plus rapide que *ILF*. Pour toutes les classes, *LAD* et *ILF* sont plus rapides que *Abscon* (notons cependant qu'*Abscon* est implémenté en Java alors que les autres approches sont implémentées en C, qui est généralement sensiblement plus rapide que Java). *Vflib* est compétitif pour les classes *bvg-100*, *m4D-81* et *m4Dr-81*, mais il n'est pas compétitif du tout pour les autres classes.

La table 2 compare les nombres de nœuds échecs de chaque approche sauf *Vflib* (pour lequel nous ne disposons pas de cette information). Pour certaines classes, comme *sf5-8-**, *LAD* et *ILF* ont un nombre comparable de nœuds échec, et cela correspond aux classes qui sont plus rapidement résolues par *ILF* que par *LAD*. Cependant, pour d'autres classes, comme *r*-100*, *LAD* explore beaucoup moins de nœuds que *ILF*. Le nombre de nœuds échecs pour *ILF* comme pour *LAD* est sensiblement plus petit que pour *Abscon*.

Vflib	*Abscon* *FC*	*Abscon* *AC*	*ILF* *k=1*	*ILF* *k=2*	*ILF* *k=4*	*LAD*
468	647	662	698	699	699	**728**
73.72	72.51	54.25	30.85	31.12	30.77	**14.57**
-	1202372	324075	297107	182588	159493	13560

Table 3 – Comparaison pour le problème consistant à chercher une solution, sur les instances de la classe LV : la première ligne donne le nombre d'instances résolues, la deuxième le temps et la troisième le nombre de nœuds échecs (moyennes sur les instances résolues).

Afin de comparer le passage à l'échelle des différentes approches et les évaluer sur un plus grand nombre d'instances, nous considérons maintenant le problème consistant à chercher uniquement la première solution. La table 3 compare les différentes approches sur la classe *LV*, contenant 793 instances de natures très variées, dont certaines s'avèrent vraiment difficiles. Pour cette classe, *LAD* a résolu 29 (resp. 29, 30, 66, 81 et 260) instances de plus que *ILF(4)* (resp. *ILF(2)*, *ILF(1)*, *Abscon(AC)*, *Abscon(FC))* et *Vflib*). *Abscon(AC)* et *ILF(1)* ont un nombre de nœuds échecs comparable, et en ont environ 3 fois moins que *Abscon(FC)*. *ILF(2)* et *ILF(4)* ont moins de nœuds échecs mais cette réduction de l'espace de recherche n'est pas suffisante pour leur permettre d'être significativement meilleurs qu'*ILF(1)*. Le nombre de nœuds échecs de *LAD* est bien plus petit (plus de 20 fois plus petit que *Abscon(AC)* et *ILF(1)*).

La table 4 compare les différentes approches pour les classes les plus faciles de la base "vflib" et montre que *LAD* est la seule approche capable de résoudre

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
bvg-200	**90**	**0.00**	**90**	0.68	**90**	0.78	**90**	**0.00**	**90**	**0.00**	**90**	**0.00**	**90**	0.14
bvg-400	**90**	**0.00**	**90**	2.85	**90**	2.99	**90**	0.01	**90**	0.01	**90**	0.01	**90**	1.06
bvg-800	**90**	**0.02**	**90**	54.13	**90**	54.86	**90**	0.03	**90**	0.04	**90**	0.05	**90**	8.41
bvgm-200	**90**	**0.00**	**90**	0.95	**90**	0.73	**90**	**0.00**	**90**	**0.00**	**90**	**0.00**	**90**	0.01
bvgm-400	**90**	**0.01**	89	3.20	89	1.66	**90**	1.55	**90**	**0.01**	**90**	**0.01**	**90**	0.04
bvgm-800	**90**	**0.03**	**90**	12.02	**90**	12.07	**90**	0.06	**90**	0.04	**90**	**0.03**	**90**	0.19
m4D-256	29	0.00	**30**	2.88	**30**	1.73	**30**	**0.01**	**30**	**0.01**	**30**	**0.01**	**30**	0.04
m4D-526	23	4.11	**30**	159.76	**30**	164.90	29	9.61	**30**	32.93	29	30.72	**30**	**1.71**
m4D-1296	20	0.05	23	252.73	23	242.47	29	52.93	29	61.21	29	73.33	**30**	**5.67**
m4Dr-256	**90**	**0.00**	**90**	7.91	**90**	1.44	**90**	0.23	**90**	1.05	**90**	2.24	**90**	0.06
m4Dr-526	**90**	**0.01**	89	23.47	89	23.35	89	14.02	89	18.31	89	19.08	**90**	0.33
m4Dr-1296	**90**	**0.06**	89	193.27	89	188.44	**90**	6.41	**90**	5.46	**90**	5.43	**90**	1.63
Tous	882	0.12	890	41.95	890	40.60	897	4.25	898	5.55	897	6.04	**900**	**1.43**

Table 4 – Comparaison pour le problème consistant à chercher une solution sur les instances faciles de "vflib" (nb = nombre d'instances résolues en moins d'une heure et temps = temps moyen pour les instances résolues).

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
r0.01-200	3	1735.93	28	192.14	**30**	0.99	28	27.48	28	44.09	28	46.49	**30**	**0.04**
r0.01-400	0	-	20	33.14	29	69.68	14	175.83	14	228.78	14	214.85	**30**	**45.58**
r0.01-600	0	-	23	226.38	23	236.14	12	428.14	9	1069.96	7	806.96	**29**	**113.51**
r0.05-200	0	-	25	266.77	28	142.80	28	125.57	28	198.68	28	198.66	**30**	**38.28**
r0.05-400	0	-	22	632.84	24	647.48	**25**	519.04	**25**	500.54	**25**	**494.12**	17	1190.88
r0.05-600	0	-	**14**	**1915.65**	**14**	1936.98	13	1505.51	5	2319.68	5	2304.85	1	2100.61
r0.1-200	0	-	27	143.36	**29**	**309.54**	26	320.52	26	357.70	26	351.10	21	646.31
r0.1-400	0	-	**6**	**1764.63**	**6**	1972.18	5	1950.67	5	1917.68	5	2070.81	1	961.35
r0.1-600	0	-	0	-	0	-	0	-	0	-	0	-	0	-
Tous	3	1735.93	165	443.14	**183**	**409.55**	151	414.03	140	447.36	138	426.67	159	268.48

Table 5 – Comparaison pour le problème consistant à chercher une solution sur les instances *r-p-n* de "vflib" (nb = nombre d'instances résolues en moins d'une heure et temps = temps moyen pour les instances résolues).

toutes les instances de ces classes. *Vflib* est très efficace et montre de bonnes propriétés de passage à l'échelle pour certaines de ces classes. De fait, *Vflib* utilise des heuristiques d'ordre de variables et de valeurs qui ne sont pas utilisées par les autres approches : à chaque itération, il choisit d'apparier le couple de nœuds (u, v) tel qu'à la fois u et v soient adjacents à des nœuds déja appariés (dans la mesure du possible). Ces heuristiques peuvent expliquer les très bonnes performances de *Vflib* sur certaines instances quand le but est de trouver juste une solution et que l'instance est satisfiable.

Enfin, la table 5 compare les différentes approches sur les classes aléatoires *r-p-n* de la base "vflib", et montre que les approches comparées exhibent des propriétés de passage à l'échelle différentes : quand la probabilité p d'ajouter une arête est 0.01, *LAD* est meilleur qu'*Abscon* qui est meilleur qu'*ILF*, mais quand cette probabilité augmente, *Abscon* devient meilleur qu'*ILF* qui est meilleur que *LAD*. En fait, plus les graphes ont de nœuds et sont denses, et plus les performances de *LAD* sont dégradées. Cela provient notamment du fait que la complexité du filtrage *LAD* est $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$: le degré est 10 fois plus grand (en moyenne) quand $p = 0.1$ que quand $p = 0.01$. En conclusion, quand les graphes sont relativement peu denses, il est intéressant de filtrer avec *LAD* tandis que quand les graphes sont plus denses, il est plus intéressant d'utiliser un filtrage tel que *AC(Edges)*.

6 Conclusion

Nous avons introduit un nouvel algorithme de filtrage pour le problème d'isomorphisme de sous-graphes. Ce filtrage propage le fait que les différents nœuds adjacents à un même nœud motif doivent être appariés à des nœuds cibles qui sont tous différents et qui sont tous adjacents à un même nœud cible. Ce filtrage est plus fort que *LV2002* et assure la même cohérence que la variante la plus forte de *ILF(k)*, *i.e.*,

quand l'étiquetage initial intègre complètement les réductions de domaines et quand les extensions d'étiquetages sont itérées jusqu'à atteindre un point fixe. Cependant, cette cohérence est assurée à un moindre coût en mettant à jour les couplages incrémentalement, au lieu de les recalculer à partir de rien à chaque fois, et en ne mettant à jour que les couplages qui ont été impactés par une réduction de domaine au lieu de recalculer tous les couplages.

Nous avons montré expérimentalement sur un jeu d'essai représentatif de près de 2000 instances que ce nouveau filtrage est capable de résoudre plus d'instances et qu'il réduit drastiquement l'espace de recherche de sorte que de nombreuses instances sont résolues sans énumération. Cependant, ce filtrage est moins bon que la cohérence d'arc pour les graphes les plus denses. Cette procédure de filtrage pourrait être facilement intégrée dans un langage de programmation par contraintes. En particulier, nous avons en projet de l'intégrer dans notre système pour apparier des graphes basé sur les contraintes [9] qui est implémenté au dessus de *Comet* [6].

Nous avons également en projet d'améliorer LAD en étudiant différentes stratégies pour choisir, à chaque itération, le prochain couple de nœuds (u, v) enlevé de S. Dans les expérimentations rapportées ici, nous avons considéré une stratégie simple de *dernier entré premier servi (LIFO)* étant donné que S est implémentée par une pile. Cependant, nous pourrions utiliser une file de priorité ordonnant les couples en fonction du nombre d'arêtes ayant été enlevées dans le graphe bipartie $G_{(u,v)}$.

Remerciements à Yves Deville, pour les nombreuses et enrichissantes discussions sur ce travail, à Jean-Christophe Luquet, qui a implémenté une première version de *ILF* pendant son stage de DUT, et à Christophe Lecoutre, qui m'a guidée avec beaucoup de patience dans l'utilisation d'Abscon. Ce travail a été effectué dans le contexte du projet Sattic (Anr grant Blanc07-1_184534).

Références

[1] Christian Bessiere and Romuald Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artif. Intell.*, 172(1) :29–41, 2008.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *ICIAP '99 : Proceedings of the 10th International Conference on Image Analysis and Processing*, page 1172, Washington, DC, USA, 1999. IEEE Computer Society.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001.

[4] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[5] M. Garey and D. Johnson. *Computers and Intractability.* Freeman and Co., New York, 1979.

[6] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* The MIT Press, 2005.

[7] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4) :225–231, 1973.

[8] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4) :403–422, 2002.

[9] V. le Clément, Y. Deville, and C. Solnon. Constraint-based graph matching. In *15th Conference on Principles and Practice of Constraint Programming (CP)*, volume 5732 of *LNCS*, pages 274–288. Springer, 2009.

[10] C. Lecoutre and S. Tabary. Abscon 112 : towards more robustness. In *3rd International Constraint Solver Competition (CSC'08)*, pages 41–48, 2008.

[11] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3) :229–250, 1979.

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

[13] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[14] J.-C. Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique.* PhD thesis, 1995.

[15] C. Solnon. *AllDifferent*-based filtering for subgraph isomorphism. *Artificial Intelligence (to appear)*, 2010.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1) :31–42, January 1976.

[17] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints (to appear)*, 2010.