Algorithme de mise en correspondance de chemins

Dans le chapitre précédent, nous avons présenté des fonctions noyaux comme fonctions de similarité entre graphes. Bien que ces fonctions possèdent des propriétés intéressantes, leur évaluation est coûteuse en temps de calcul. Nous allons présenter dans ce chapitre un cadre permettant de limiter le temps de calcul de ces fonctions.

Le temps de calcul de ces fonctions est lié au temps nécessaire pour apparier deux chemins de deux graphes et calculer leur similarité. Par exemple, la fonction noyau de Kashima utilise la somme des similarités de tous les appariements possibles. Pour résoudre ce problème, on recense différentes méthodes dans la littérature pour :

- réduire le nombre de chemins (et donc le nombre d'appariements) ,
- réduire le nombre de calculs par une approximation de la fonction de similarité,
- réduire la redondance dans les chemins par une construction récursive,
- réduire la redondance de calcul via le stockage de résultats partiels communs à divers calculs.

La première méthode réduit le nombre de chemins appariés. Par exemple, Suard a fortement réduit ce nombre en n'utilisant que les chemins représentant des plus courts chemins entre 2 sommets donnés. Outre une réduction, ce choix permet l'utilisation d'algorithmes performants pour la génération des chemins. De plus, la formule du noyau considéré reste inchangée en utilisant une méthode de réduction du nombre de chemins. Néanmoins ce genre de réduction peut poser des problèmes quant à la conservation des propriétés mathématiques de la fonction noyau sur graphe. Enfin, il impose un choix qui peut se révéler important selon la morphologie structurelle du graphe.

La méthode d'approximation est le plus souvent utilisée lorsque le nombre de chemins est infini ou très grand. Il s'agit d'approximer les valeurs des fonctions en effectuant soit des calculs de fonctions proches mais plus simples, soit en supprimant certains calculs ayant une faible contribution sur la valeur finale, soit en arrêtant, avant terminaison, le calcul en utilisant des propriétés de convergence. Kashima l'utilise pour pallier une

possibilité de temps de calcul infini (si le nombre de chemins est infini, le nombre d'appariements le sera. Par conséquent, le calcul de la somme sur toutes les similarités des appariements ne se finit jamais) . Néanmoins, celle-ci pose des problèmes de non conservation des propriétés mathématiques des fonctions noyaux utilisées dans la similarité.

La limitation de la redondance dans les chemins s'effectue le plus souvent à l'aide de création récursive des chemins. Celle-ci nécessite de considérer des ensembles de chemins ayant des propriétés de récursivité.

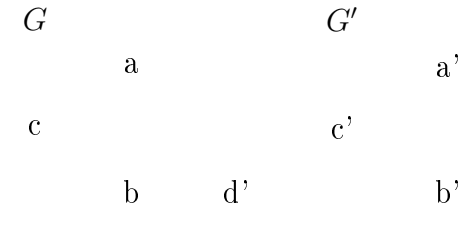
Le stockage d'éléments s'arrête le plus souvent au stockage des similarités sur les arêtes et les sommets. La similarité des chemins peut s'écrire le plus souvent comme une combinaison de la similarité de ses sous-éléments arêtes et sommets. La similarité des graphes peut elle-même combiner la similarité de nombreux chemins. Ainsi ces similarités d'arêtes et de sommets interviennent de nombreuses fois dans le calcul des similarités de chemins et de graphes.

Notre méthode exploite la redondance et la récursivité de l'expression des appariements de chemins dans une représentation sous forme d'arbre. Dans cette représentation, un chemin entre la racine et la feuille de l'arbre est une solution d'appariement de chemins. Cette représentation permet de se placer dans le cadre d'exploration d'un arbre de recherche. Par conséquent des algorithmes de type $A*$, par exemple le "branch and bound" et d'autres algorithmes d'exploration d'arbre de recherche peuvent être employés pour résoudre l'appariement de deux graphes par similarité d'ensembles de chemins.

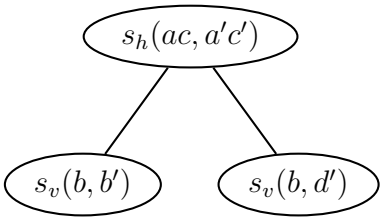
Nous présentons la mise en représentation sous forme d'arbre d'appariements pour une catégorie de fonctions de mise en correspondance de graphe. Puis nous montrons l'application de cette représentation d'arbre avec des ensembles de chemins pour construire un arbre de recherche parcourant l'ensemble des possibilités. Enfin nous présentons l'utilisation algorithmique de ces représentations pour calculer les similarités noyaux sur graphes.

4.1 Représentation arborescente du calcul de similarité de chemins

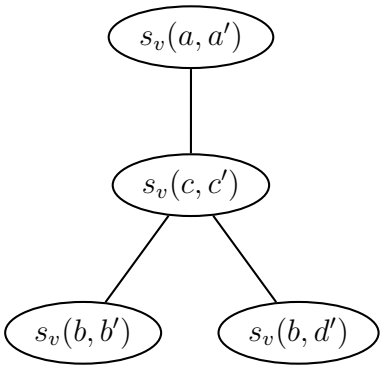
Les chemins ainsi que leur fonctions de similarité basées sur la similarité entre sommets et arêtes sont calculables ou constructibles récursivement. Les éléments possédant cette propriété récursive peuvent facilement se représenter sous forme d'arbre en regroupant les parties communes. La littérature a détaillé l'utilisation d'arbres de recherche sur des appariements de chemins entre deux graphes. Ce qui nous intéresse est d'obtenir un arbre de recherche dans le cadre d'ensemble de chemins particuliers. Et de plus, il faut que la fonction de similarité entre les chemins soit calculable lors de l'exploration de l'arbre. Pour cela il faut vérifier que le calcul peut lui aussi se représenter sous forme d'arbre. Nous allons dans un premier temps montrer au travers d'exemples comment on peut effectuer cette représentation arborescente puis dans un deuxième temps, formaliser l'écriture de ces similarités au travers de l'arbre. Enfin nous mettrons en place une structure arborescente qui permettra la mise en place dans la prochaine



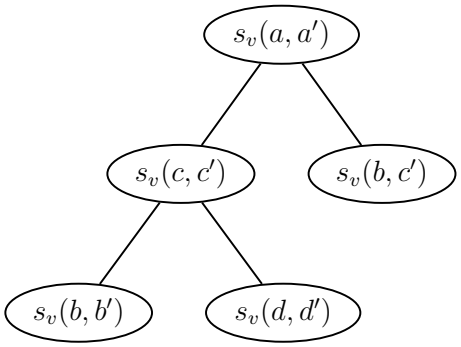
(a) Exemple de deux graphes G et G'



(b) Arbre T_1 où la similarité commune des appariements de chemins (issues de G et G') est compactifiée sur le nœud racine



(c) Arbre T_2 avec une séparation de la racine de T_1 en deux nœuds



(d) Ajout d'un nœud portant la similarité de $s_v(b, c')$

FIGURE 4.1 – On calcule des valeurs de similarité pour des appariements de chemin entre deux graphes, G et G', à l'aide d'arbre.

section, d'un arbre de recherche valide pour des ensembles de chemins.

4.1.1 Un exemple simple

Nous proposons d'illustrer la représentation du calcul de la similarité de graphes sous forme d'arbre au travers d'un exemple. Dans celui-ci, nous voulons comparer deux graphes G et G' (figure 4.1(a)) à l'aide des chemins de longueur deux (soit deux arêtes voir tableau 4.1). Pour cela nous considérons une fonction de similarité de graphes, S_G, qui renvoie la valeur du meilleur appariement parmi l'ensemble des appariements de chemins de longueur deux, issus des graphes comparés. Autrement dit une fonction qui sélectionne l'appariement avec la plus haute valeur de similarité entre chemins, noté s_h, dans l'ensemble des appariements de chemins. Ce qui revient à la formule suivante :

$$S_G(G, G') = \max_{h \in H^2(G)} \max_{h \in H^2(G')} s_h(h, h')$$

avec $H^2(G)$ l'ensemble des chemins de longueur 2 de G.

	liste des chemins
$H^2(G)$	abc,acb,bac,bca,cab,cba
$H^2(G')$	$a'c'b'$,$a'c'd'$,$b'c'a'$,$b'c'd'$,$d'c'b'$,$d'c'a'$

TABLE 4.1 – Tableau récapitulatif des chemins de longueur 2 issus des deux graphes G et G' de la figure 4.1(a)

Pour rappel un chemin est une séquence ordonnée de sommets et d'arêtes, on le note h. On peut écrire la séquence de sommets et d'arêtes d'un chemin h de la manière suivante :

$$h = v_0 e_1 v_1 ... e_n v_n$$

ou bien se contenter d'énumérer les sommets $h = v_0 v_1 ... v_n$.

Pour définir la ressemblance entre deux chemins, on peut simplement sommer les similarités s_v entre chaque paire de sommets des deux chemins appariés :

$$s_h(h = v_0 \dots v_n, h' = v'_0 \dots v'_n) = \sum_{i=0}^{n} s_v(v_i, v'_i)$$

En reprenant les graphes d'exemple (figure 4.1(a)), pour l'appariement du chemin abc de G et du chemin $a'c'b'$ de G' on a :

$$s_h(acb, a'c'b') = s_v(a, a') + s_v(c, c') + s_v(b, b')$$

Le caractère récursif de la fonction de similarité nous permet de réécrire cette similarité entre chemins de longueur deux en fonction de celle de l'appariement de chemins de longueur une :

$$s_h(acb, a'c'b') = s_h(ac, a'c') + s_v(b, b')$$

Nous allons, maintenant, comparer la similarité entre ces deux chemins avec celle de acb de G et $a'c'd'$ de G'.

$$s_h(acb, a'c'b') = s_v(a, a') + s_v(c, c') + s_v(b, b') = s_h(ac, a'c') + s_v(b, b')$$

et

$$s_h(acb, a'c'd') = s_v(a, a') + s_v(c, c') + s_v(b, d') = s_h(ac, a'c') + s_v(b, d')$$

On remarque :

- qu'ils ont des valeurs communes ($s_h(ac, a'c')$),
- qu'ils peuvent se réécrire à partir de similarité de chemins de longueur inférieure (chemin ac de G apparié avec le chemin $a'c'$ de G').

La représentation de la figure 4.1(c) se rapproche plus d'un arbre représentant les appariements. La figure 4.1(b) montre une représentation de ces similarités sous forme d'arbre : la similarité commune est encore plus visible (premier nœud). La valeur de similarité portée par le premier nœud de l'arbre de la figure 4.1(b) est égale à la somme des similarités portées par les deux premiers nœuds de l'arbre de la figure 4.1(c).

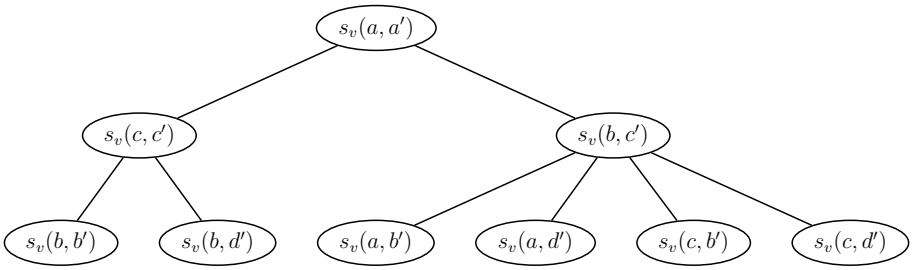


FIGURE 4.2 –

En sommant chaque similarité le long du parcours du premier nœud de l'arbre à une feuille de l'arbre on obtient la similarité de l'appariement de chemin représenté par ce parcours. Dans le cas de la figure 4.1(c) les similarités des appariements acb avec $a'c'd'$ et acb avec $a'c'b'$.

Si l'on considère de nouveaux appariements de chemins commençant par ab et $a'c'$ qui comparent non plus c avec c' mais b avec c', on développe dans un premier temps le nœud $s_v(a, a')$ pour lui rajouter le fils $s_v(b, c')$ (figure 4.1(d)). Pour obtenir toutes les feuilles de profondeur 2 qui portent les solutions d'appariements de longueur 2, on développe ce nœud. On obtient de nouvelles feuilles à l'arbre (figure 4.2) qui sont autant de nouvelles solutions d'appariements.

La fonction de similarité peut être vue comme la fonction d'évaluation(idéalement calculable pendant l'exploration de l'arbre) pour des algorithmes de type $A*$. Pour un arbre de recherche représentant les diverses solutions d'appariements entre les chemins de deux graphes. Un arbre de recherche est un arbre représentant un espace de solutions finies.

Pour utiliser ce genre de représentation et ce type d'algorithme, il est nécessaire de s'assurer des contraintes qui se posent sur les fonctions de similarités de chemins et de la complète représentation de l'espace par l'arbre de recherche considéré. Dans la prochaine section nous traitons le premier point.

4.1.2 Similarités sur chemins pour arbre de recherche

Le processus introduit dans la section précédente peut se généraliser à d'autres fonctions de similarités de chemins. Dans cette section, nous allons présenter les catégories de fonctions qui sont utilisables dans le cadre d'arbre de recherche. Lors de la création des arbres pour l'exemple (section 4.1.1), nous avons utilisé cette fonction de similarité sur chemins :

$$s_h(h = v_0 \dots v_n, h' = v'_0 \dots v'_n) = \sum_{i=0}^{n} s_v(v_i, v'_i)$$

Cette fonction peut être calculée via des méthodes récursives :

$$\begin{aligned} s_h(h = v_0 \dots v_n, h' = v'_0 \dots v'_n) = & \sum_{i=0}^{n} s_v(v_i, v'_i) \\ = & \sum_{i=0}^{n-1} s_v(v_i, v'_i) + s_v(v_n, v'_n) \\ = & s_h(h = v_0 \dots v_{n-1}, h' = v'_0 \dots v'_{n-1}) + s_v(v_n, v'_n) \end{aligned}$$

La valeur de similarité de notre appariement (h, h') peut être calculée à partir de la valeur d'un sous appariement de celui-ci $(v_0 \dots v_{n-1}, v'_0 \dots v'_{n-1})$. Cette propriété est nécessaire pour un calcul de la similarité pendant l'exploration de l'arbre de recherche d'appariement.

La fonction précédente peut se réécrire sous une forme récursive :

$$s(h_n, h'_n) = \begin{cases} s_v(v_0, v'_0) \text{ si } h = v_0 \text{ et } h' = v'_0 \\ s_h(h_{n-1}, h'_{n-1}) + s_v(v_n, v'_n) \text{ si } n > 1 \end{cases}$$

avec $h_n = h_{n-1} e_n v_n$ et $h'_n = h'_{n-1} e'_n v'_n$.

La fonction prise comme exemple possède la particularité supplémentaire d'être une fonction noyau si la fonction s_v l'est. De nombreuses fonctions noyaux pour les similarités de chemins sont comme la fonction précédente des fonctions récursives. Plus particulièrement, les similarités de chemins utilisées dans les noyaux sur graphes sont souvent récursives. Car elles sont généralement des combinaisons de fonctions noyaux sur les sous-éléments qui composent un chemin : les sommets et les arêtes.

Par la suite, nous ne considérons que les fonctions récursives (notées s_h) que l'on pourra réécrire sous cette forme à l'aide de trois opérateurs op_v,op_e et op_{noeud} :

$$s_h(h_n, h'_n) = \begin{cases} op_v(v_0, v'_0) \text{ si } h_n = v_0 \text{ et } h'_n = v'_0 \\ op_e(v_0 e_1 v_1, v'_0 e'_1, v'_1) \text{ si } h_n = v_0 e_1 v_1 \text{ et } h'_n = v'_0 e'_1 v'_1 \\ op_{noeud}(e_n, v_n, e'_n, v'_n, s_h(h_{n-1}, h'_{n-1})) \text{ sinon} \end{cases}$$

Les opérateurs op_v et op_e similarité de sommets et d'arêtes :

$$op_v : V \times V \to \Re$$

$$op_e : E \times E \to \Re$$

On utilise ces opérateurs pour les premiers nœuds de l'arbre. Dans l'arbre T_1 de notre section exemple (4.1(b)), on utilise $op_e(ac, a'c') = s_v(a, a') + s_v(c, c')$ pour la racine. Dans l'arbre T_2(4.1(c)), on utilise $op_v(a, a') = s_v(a, a')$ pour la racine. Le choix entre ces deux opérateurs dépend de la fonction de similarité. Notre fonction exemple utilise uniquement les sommets ; par conséquent, il vaut mieux travailler avec l'opérateur op_v.

L'opérateur op_{noeud} ($op_{noeud} : E \times V \times E \times V \times \Re \to \Re$) correspond aux opérations à effectuer à partir de la valeur de similarité du sous-appariement (h_{n-1}, h'_{n-1}) et des nouveaux éléments (sommets v_n et v'_n et arêtes e_n et e'_n) pour obtenir celle de l'appariement (h_n, h_n). C'est ce dernier opérateur qui est utilisé dans nos arbres de recherche pour mettre la valeur de similarité à jour à chaque création de nœud.

Pour la suite, nous supposons que toutes les fonctions de similarité sur les chemins sont récursives. Ainsi dans le cadre d'arbre de recherche sur des espaces d'appariement

de chemins, on pourra calculer ou estimer le calcul de la similarité pendant l'exploration de l'arbre. Ainsi ces fonctions pourront être utilisées dans les algorithmes d'exploration d'arbre de recherche comme fonction de référence.

4.1.3 Arbre d'appariements sur chemins de longueur fixe

Nous allons dans cette section définir un arbre qui représente l'espace des appariements de chemins de longueur fixe et de manière exhaustive : un arbre d'appariement. Cet arbre nous permettra de représenter complètement et uniquement cet espace des appariements entre $H(G)$ et $H(G')$ (H une fonction générant un ensemble de chemins à partir d'un graphe). De plus les fonctions de similarité des chemins de cet arbre sont calculables pendant d'exploration de l'arbre.

Nous allons dans un premier temps définir l'arbre d'appariement de façon à obtenir un arbre de recherche valide pour l'espace constitué de tous les appariements possibles entre les chemins de longueur fixe entre deux ensembles. La définition d'arbre d'appariement, nous assure que l'arbre contient toutes les solutions d'appariement entre les chemins de deux ensembles A et B. Puis on s'intéresse à ne considérer que l'espace de solution des appariements de chemins de longueur fixe.

Un ***arbre d'appariement*** $T(A, B)$ *pour deux ensembles de chemins A et B est un arbre de recherche dont tous les parcours entre la racine et une feuille représentent des appariements entre deux chemins de même longueur issus de A et B. De plus, tout appariement de deux chemins de même longueur issus de A et B est représenté par un parcours entre la racine et un nœud de l'arbre.*

On note $App(A, B)$ l'ensemble des appariements entre deux chemins de A et de B.

Le dernier arbre de recherche présenté (Fig. 4.2) lors de l'exemple de la section 4.1.1 est un arbre d'appariement entre les ensembles de chemins $A = \{ac, acb, aba, abc\}$ et $B = \{a'c', a'c'b', a'c'd'\}$. En effet, quelque soit l'appariement de chemins choisi, on trouve un chemin ξ dans l'arbre qui le représente. De plus tous les chemins entre la racine et les feuilles de l'arbre sont bien des appariements entre deux chemins de A et B (ex : le chemin en rouge dans la fig.4.3 correspond à un appariement entre les chemins acb de A et *a'c'b'* de B) :

$$\begin{aligned} App(A,B) = \quad & \{(ac, a'c'), \\ & (acb, a'c'b'), (acb, a'c'd'), (aba, a'c'b'), (aba, a'c'd'), \\ & (abc, a'c'b'), (abc, a'c'd')\} \end{aligned}$$

On remarque que les ensembles A et B contiennent des chemins avec une seule arête et des chemins avec deux arêtes. L'appariement des chemins d'une arête est représenté dans un arbre de ce type par des parcours de la racine au nœud de profondeur 1. Par exemple pour l'arbre de la figure 4.2, l'appariement de l'arête ac avec l'arête $a'c'$ (noté $\alpha(ac, a'c')$), correspond au parcours entre la racine ($s_v(a, a')$) et le nœud le plus à gauche parmi les fils de la racine : ($s_v(c, c')$).

N'importe quel parcours dans un arbre d'appariement ne représente pas forcément un des appariements des deux ensembles de chemins (A et B dans notre exemple). Dans

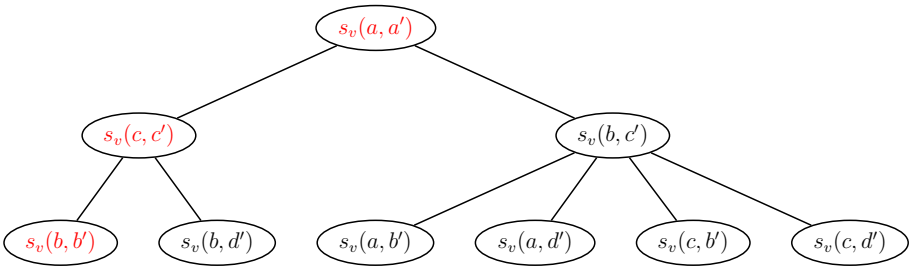


FIGURE 4.3 –

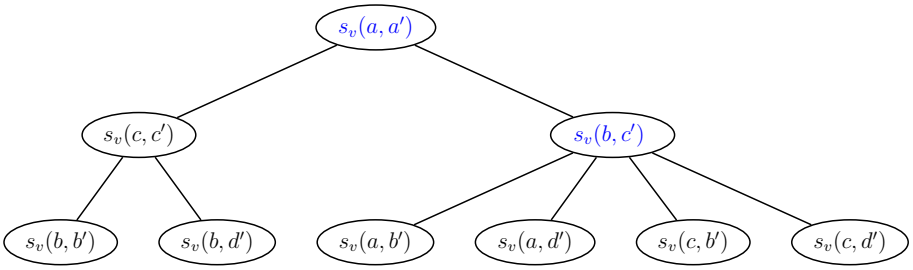


FIGURE 4.4 –

l'arbre de la figure 4.4, le chemin bleu représente l'appariement (ab,$a'c'$) qui n'est pas contenu dans l'ensemble $App(A,B)$.

Par contre, comme on peut le voir sur notre arbre d'exemple, tous les parcours de la racine jusqu'au nœud définissent un appariement appartenant à l'ensemble App (pour notre exemple $App(A,B)$). Il est plus intéressant de travailler uniquement sur les chemins entre la racine et les feuilles car nous sommes certains qu'ils définissent un appariement entre deux chemins issus des ensembles. Dans un cadre de construction de ces arbres de manière récursive, il est pertinent de travailler avec des ensembles qui nous assurent que les seuls appariements possibles entre les deux ensembles soient tous contenus dans les parcours de la racine aux feuilles de l'arbre (et uniquement dans ces chemins).

Soit A^k le sous-ensemble de A ne contenant que les chemins de longueur k dans A (la longueur k étant le nombre d'arêtes constituant le chemin). Cette notation est aussi utilisée dans le cadre des fonctions H (génèrent un ensemble de chemins pour un graphe donné) : H^k est une fonction qui ne donne que des chemins de k arêtes pour un graphe donné. Cette notation nous permet de simplifier l'écriture de propriétés par la suite.

Revenons à nos deux ensembles A et B ainsi qu'à l'arbre(fig.4.4) de l'exemple, pour les ensembles A^2 et B^2 nous avons :

$$\begin{aligned} App(A^2,B^2) = \quad & \{(acb,a'c'b'),(acb,a'c'd'),\\ & (aba,a'c'b'),(aba,a'c'd'),\\ & (abc,a'c'b'),(abc,a'c'd')\} \end{aligned}$$

On peut vérifier que tous les appariements entre A^2 et B^2 sont représentés par les

parcours de la racine aux feuilles de cet arbre et uniquement par ceux-ci. Dans les arbres de recherche, les solutions sont représentées par les feuilles de l'arbre ; le parcours de la racine jusqu'à une feuille est forcément unique et représente dans notre cas une solution d'appariement pour des chemins de longueur liée à la profondeur de la feuille dans l'arbre. Il nous reste à considérer le cas des ensembles issus de fonctions H^k pour s'assurer que l'espace de solution est bien complet et unique.

Nous allons voir que pour deux ensembles de chemins $H^k(G)$ et $H^k(G')$:

à chacun de leur appariement $\alpha(h, h')$ il existe un unique parcours $p(racine, feuille)$ le représentant dans l'ensemble $P(T(H^k(G), H'^k(G')))$.

$P(T(H^k(G), H'^k(G')))$ est l'ensemble des parcours de la racine aux feuilles de l'arbre d'appariement $T(H^k, H'^k)$.

Le parcours p peut se noter comme une séquence de couples de sommets puisqu'il représente un appariement de chemins. Les notations pouvant être utilisées sont :

$$p = (racine)(n_0)(n_k)$$

où n est un nœud l'arbre,

$$p = (racine)\binom{v_0}{v'_0} \cdots \binom{v_k}{v'_k},$$

$$p = \binom{h}{h'}.$$

Théorème 4.1.1. *Soit H une fonction générant un ensemble de chemins à partir de graphe.*
Soient $G = (V, E)$ et $G' = (V', E')$ deux graphes.
Soit $T(H^k(G), H^k(G'))$ un arbre d'appariement
On note l'ensemble des parcours de la racine à la feuille d'un arbre T : $\Xi(T)$
Alors on a : $App(H^k(G), H^k(G')) = \Xi(T(H^k(G), H^k(G'))$.
Autrement dit :
Quelque soit l'appariement entre deux chemins h de $H^k(G)$ et h' de $H^k(G')$ il existe un parcours p de la racine à la feuille de l'arbre T tel que : $p = (racine)\binom{h}{h'}$
Quelque soit le parcours $p = (racine)\binom{v_0}{v'_0} \cdots \binom{v_p}{v'_p}$ de la racine à une feuille de l'arbre T, $\alpha = (v_0 ... v_p, v'_0 ... v'_k)$ est un appariement entre deux chemins de $H^k(G)$ et $H^k(G')$.

Ce lemme se démontre simplement(voir annexe A), il permet de nous assurer de l'exploration complète et unique des solutions(voir annexe A).

Il est possible de construire récursivement les arbres de comparaison, si les ensembles associés à ces arbres sont constructibles récursivement.

Nous avons donc une définition d'un arbre qui, pour une longueur de chemins donnée, va produire tous les appariements possibles de cette longueur. Par la suite nous représenterons uniquement le couple de sommets au lieu de leur similarité dans les arbres.

Notre but est d'obtenir un arbre de recherche pour les fonctions H qui produisent des ensembles de chemins à partir de graphes. Puisque ces fonctions sont couramment

utilisées dans les cadre de fonctions noyaux sur graphes. Pour cela nous allons introduire le terme de famille d'arborescences liées à une fonction H qui nous permettra de définir un arbre de recherche appliqué à l'espace des appariements entre chemins issus des ensembles $H(G)$ et $H(G')$ pour deux graphes G et G'.

4.2 Ensembles de chemins et arbres

Dans la section précédente nous avons travaillé sur une représentation d'arbre de recherche ne comprenant que des chemins de même longueur. Pour travailler sur l 'espace des appariements engendré par le triplet : graphe G , graphe G'et la fonction générative de chemins H, nous allons généraliser notre représentation et introduire des familles d'arbres. Les ensembles de chemins sont des éléments importants qui peuvent influer sur l'apprentissage et la complexité du calcul. Nous allons tout d'abord revoir les ensembles de chemins issus des fonctions génératives introduites à la section 2.1.2. Puis nous nous appuierons sur le théorème des arbres d'appariements pour définir des familles de forêts d'arbres d'appariements liées à un type d'ensembles de chemins particulier. Nous montrerons que ces familles, pour une fonction donnée et un couple de graphe donné, nous permettent d'obtenir un arbre de recherche représentant l'espace des appariements de chemins possibles sur lequel travaillent les fonctions noyaux présentées dans le chapitre précédent. Puis nous montrerons les applications d'exploration de l'arbre de recherche définies par cette famille d'arbres.

4.2.1 Ensembles de chemins

Nous avons besoin de manipuler tous les appariements possibles entre deux ensembles. On doit s'assurer en effet que nos arbres de recherche permettent d'explorer l'espace complet des appariements entre les deux ensembles de chemins $H(G)$ et $H(G')$.

Par exemple, la fonction $H_{sansccycle}$ renvoie un ensemble contenant tous les chemins sans cycle du graphe G. On utilise les notations suivantes :

- H^k fonction générant des chemins de longueur k,
- $H^{\leq k}$ fonction générant des chemins de longueur inférieure ou égale à k.

Une condition suffisante pour obtenir l'arbre de recherche couvrant l'espace complet des appariements entre deux chemins(de même longueur) issus des ensembles $H(G)$ et $H(G')$, est d'avoir des ensembles dits récursifs(voir chapitre 1 section 2.1.2). Autrement dit, il faut un ensemble A, dont tout chemin de longueur supérieure ou égal à k peut être construit à partir d'un chemin de taille inférieure appartenant à A. Cette propriété a déjà été exploitée dans la section précédente.

Ces ensembles peuvent être par conséquent divisés en sous-ensembles, ne contenant que des chemins de même longueur, ce qui donne pour un ensemble de chemins A dit récursif : $A = \cup_{k=0}^{\infty} A^k$

Une fonction générative H est dite récursive si elle produit pour tout graphe G un ensemble $H(G)$ qui est lui même récursif. En considérant l'ensemble d'arbres (noté F^k) , pour deux graphes G et G', représentant chaque appariement de chemins issus de $H^k(G)$

et $H^k(G')$, on s'aperçoit que pour tout arbre T^k(solutions d'appariement de longueur k), $k > 0$ il existe T^{k-1} à partir duquel on peut construire T^k grâce au caractère récursif de H. De ce fait l'arbre de recherche final peut être construit totalement ou partiellement en se basant sur des opérations récursives.

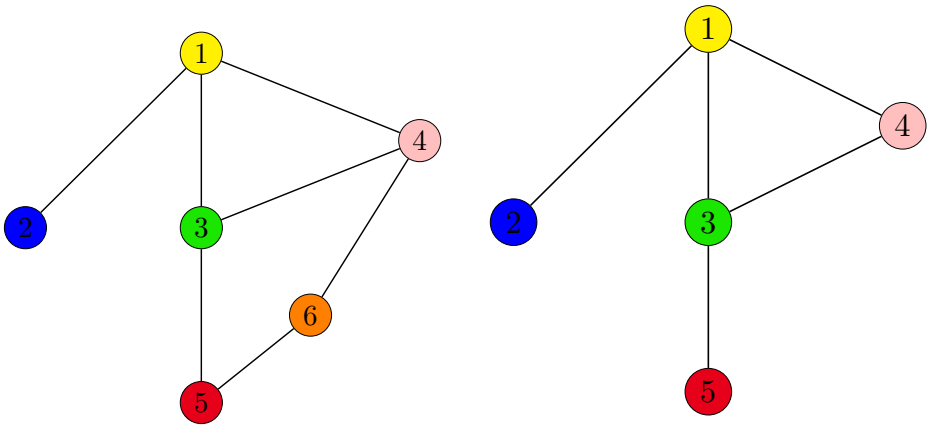


FIGURE 4.5 – Deux graphes pris pour illustrer les motifs induits par les noyaux sur graphes lors de leur calcul.

4.2.2 motifs d'appariement dans les graphes induits par les noyaux

Nous allons regarder pour quelques noyaux de graphe sur un exemple de deux graphes le poids des petits éléments dans le calcul du noyau et les motifs d'appariements produits si il y en a.

La figure 4.5 présente les deux graphes utilisés dans les exemples. Ces deux graphes possèdent un sous-graphe commun composé des sommets 1,2,3,4 et 5.

4.2.2.1 K_{max}

Ce noyau a bien pour équivalence un appariement de chemins. Par conséquent il couvre peu le graphe et ne garantit pas de passer par les sommets et les arêtes constituant l'objet.

4.2.2.2 K_{sum}

Ce noyau somme tous les appariements possibles entre les chemins de même longueur. Il recouvre avec seulement les arêtes pratiquement tout le graphe.

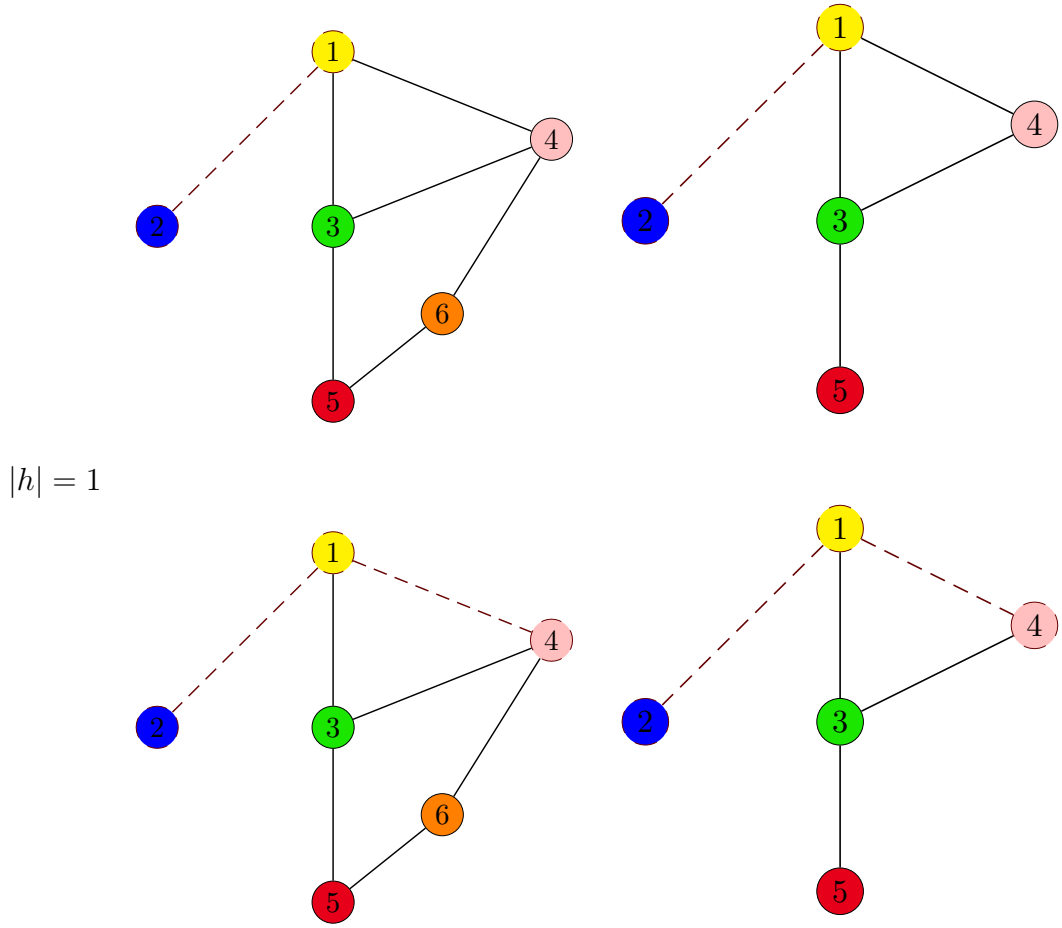
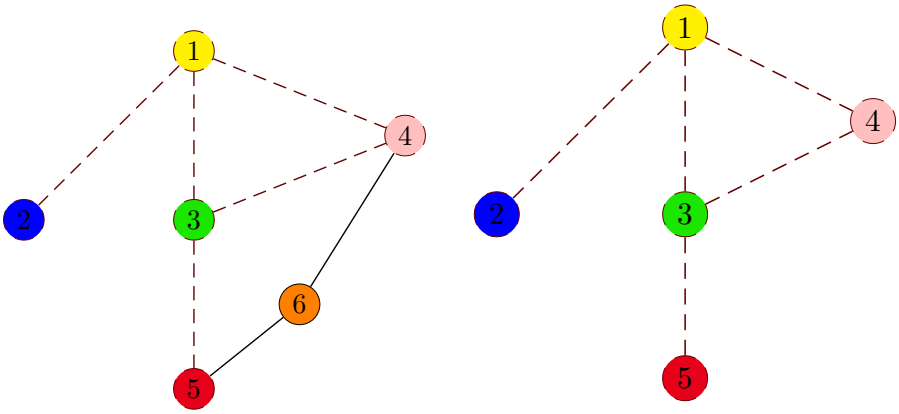


FIGURE 4.6 – K_{max}. Représentation des arêtes et sommets contribuant au calcul du noyau en trait non continu rouge foncé dans le cadre d'une hypothèse de noyau mineur se comportant de manière binaire. Plus les traits sont larges, plus les arêtes ou sommets ont contribué à ce calcul

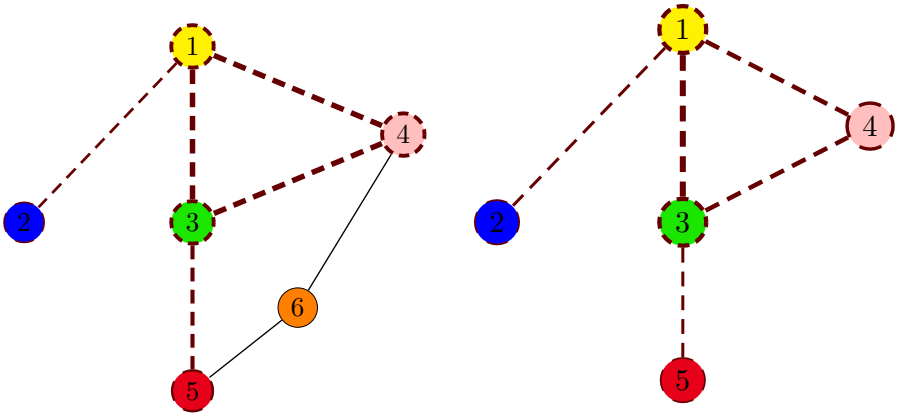
La figure 4.7 montre pour une hypothèse de comportement binaire des noyaux (0 si les éléments sont différents, 1 s'ils sont identiques). On s'aperçoit que les éléments non communs aux deux graphes ne comptent pas dans le calcul.

La figure 4.8montre pour une hypothèse de comportement non binaire des noyaux (strictement entre 0 et 1 si les éléments sont différents, 1 s'ils sont identiques). Dans ce cas tous les éléments des graphes participent au calcul.

Dans les deux figures on remarque que le nombre de voisins possibles des sommets influent sur leur participation au calcul. Plus le sommet est entouré de voisins plus il aura de poids dans le noyau.



$|h| = 1$



$|h| = 2$

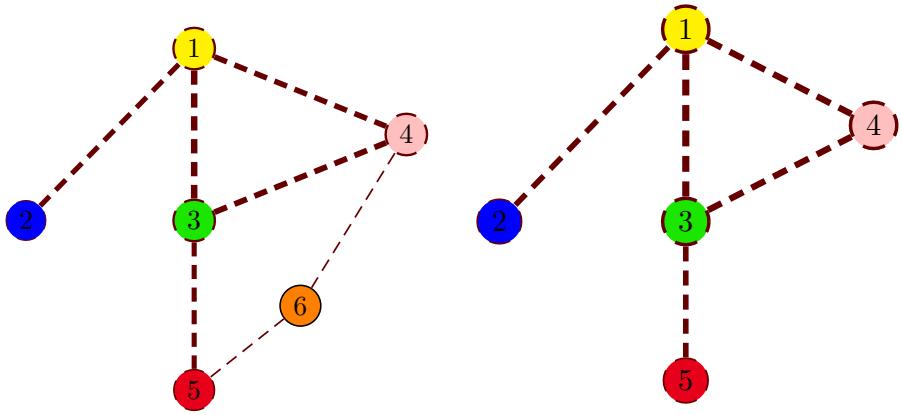
FIGURE 4.7 – K_{sum}. Représentation des arêtes et sommets contribuant au calcul du noyau en trait non continu rouge foncé dans le cadre d'une hypothèse de noyau mineur se comportement de manière binaire. Plus les traits sont larges, plus les arêtes ou sommets ont contribué à ce calcul

4.2.2.3 K_{new}

Contrairement au noyau précédent le sommet de plus forte valence n'est pas forcément celui qui compte le plus dans le noyau. Le noyau exploite bien une grande majorité du graphe dès la longueur 1.

4.2.3 Arbre de recherche pour ensembles de chemins

Les arbres d'appariements nous permettent d'explorer par des algorithmes l'espace des appariements entre deux ensembles de chemins. Par conséquent, il faut vérifier que nous pouvons explorer totalement cet espace d'appariements or on peut difficilement



$|h| = 1$

FIGURE 4.8 – K_{sum}. Représentation des arêtes et sommets contribuant au calcul du noyau en trait non continu rouge foncé dans le cadre d'une hypothèse de noyau mineur se comportement de manière non binaire. Plus les traits sont larges, plus les arêtes ou sommets ont contribué à ce calcul.

le garantir en n'utilisant qu'un seul arbre ; il est préférable de travailler sur une forêt d'arbres d'appariements cohérente pour les graphes comparés et la fonction générative d'un ensemble de chemins utilisée. Chaque arbre de cette forêt aura une profondeur donnée et on ne considère que les appariements définis par un chemin entre une feuille et la racine de l'arbre. Ainsi ils représentent tous les appariements liés à une longueur donnée de chemins. De ce fait, la forêt doit contenir tous les appariements avec ces arbres. De plus, l'hypothèse de récurrence sur les chemins nous permet la construction récursive des arbres de cette forêt. Suite à cette hypothèse, les appariements sont eux aussi récursifs et permettent cette récursivité au niveau des arbres de la forêt.

Une fois les forêts définies, nous passerons à la construction récursive de ces arbres dans ces familles.

4.2.3.1 Définition des familles d'arbres

En théorie, nous travaillons sur ces familles d'arbres pour calculer les similarités. En pratique, nous n'utilisons qu'un seul arbre auquel nous augmentons la profondeur quand c'est nécessaire grâce aux propriétés récursives. La famille d'arborescences doit donner pour deux graphes un ensemble d'arbres de différentes profondeurs deux à deux. De plus, on doit les construire par récursivité.

Définissons d'abord la famille d'arborescences de comparaison :

Définition 4.2.1. *Soit une fonction générative H .*
Soit F une forêt d'arborescences d'appariements.
On dit que F est une famille d'arbres d'appariements de H si et seulement si elle contient

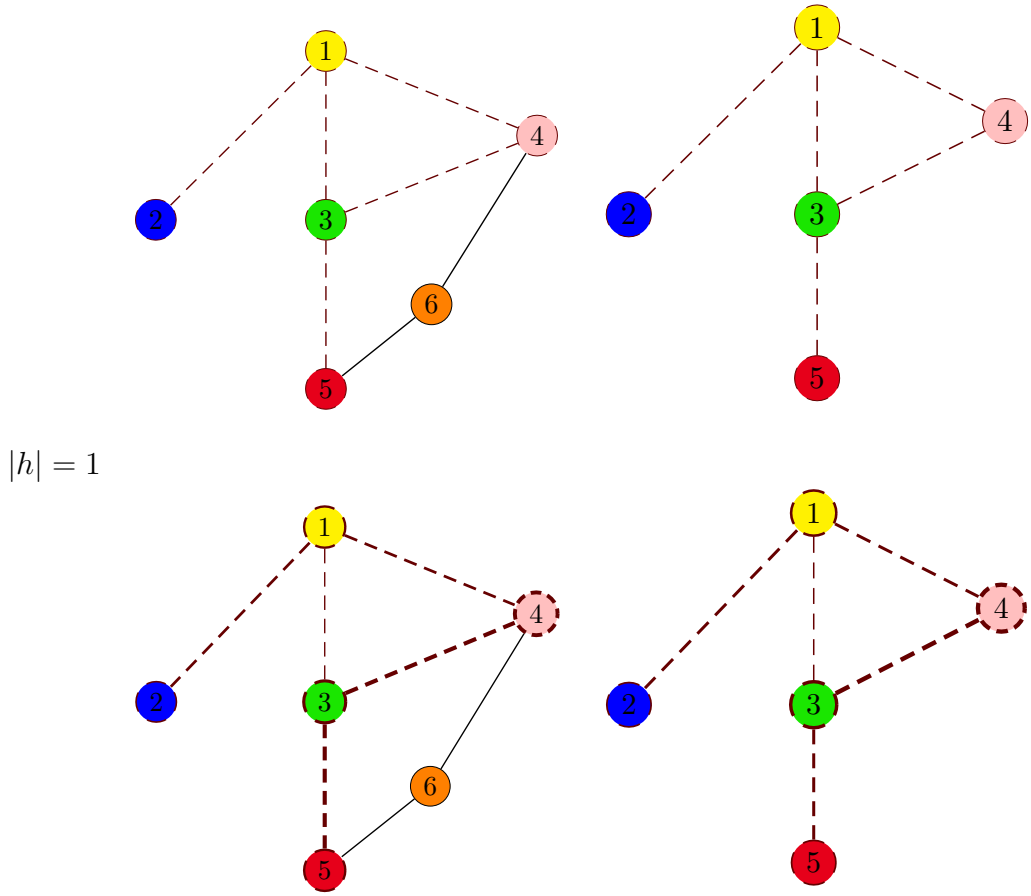


FIGURE 4.9 – K_{new} Représentation des arêtes et sommets contribuant au calcul du noyau en trait non continu rouge foncé dans le cadre d'une hypothèse de noyau mineur se comportement de manière binaire. Plus les traits sont larges, plus les arêtes ou sommets ont contribué à ce calcul

toutes les arborescences T liées à H et exclusivement celles-ci quelque soit les graphes considérés.

4.2.3.2 Construction récursive

Nous allons aborder la construction récursive d'un arbre dans une famille d'arbres donnée pour un couple de graphes donné. La validation de cette construction et l'exhaustivité de cette représentation seront détaillées dans la section suivante.

Dans un premier temps, nous reprenons les graphes de l'exemple de la section 4.1.1 pour illustrer la mise en place de cette construction récursive. Nous restreignons les chemins à ceux contenus dans l'ensemble des chemins sans cycle (fonction H_{sc}). Le

	G	G'
H^1_{sc}	ab, ba, bc, cb, ac, ca	$a'c', c'a', c'd', d'c', c'b', b'c'$
H^2_{sc}	abc, bac, bca cba, acb, cab	$a'c'b', d'c'a', b'c'a'$ $a'c'd', d'c'b', b'c'd'$

TABLE 4.2 – Tableau récapitulatif des chemins de longueur 1 et 2 contenu dans $H_{sc}(G)$ et $H_{sc}(G')$

tableau (4.2) récapitule les chemins de longueur 1,2 appartenant à cet ensemble.

Nous supposons que l'arbre d'appariements $T_1(G, G')$(figure 4.10) de la famille $F_{H_{sc}}$ est déjà construit. Nous construisons ensuite $T_2(G, G')$ de cette famille à partir de $T_1(G, G')$. Puis cette construction sera généralisée.

On prend la première feuille en partant de la gauche (b, c') qui correspond au couple de chemins : $ab, a'c'$. On cherche tous les voisins de b dans G et tous les voisins de c' dans G', a et c sont les voisins de b. a', b'etd' sont les voisins de c'. Ce qui nous donne comme chemins pour G : abc, aba ; et pour G' on a : $a'c'a', a'c'b, a'c'd'$. On élimine les sommets donnant des chemins n'appartenant pas à $H^2_{sc}(G)$ ou $H^2_{sc}(G')$: a et a'. On rajoute les nœuds à la feuille représentant tous les appariements possibles entre c et b',d' : c, b' et c, d'.

Pour toutes les autres feuilles, on applique le même fonctionnement :

1 On récupère les voisins des sommets de la feuille
2 On élimine ceux qui donnent des chemins cycliques (n'appartenant pas à H^2_{sc})
3 On rajoute les nœuds à la feuille représentant tous les appariements possibles entre les deux ensembles de sommets (vidés des sommets donnant des cycles).

Au final on obtient $T_2(G, G')$ de la famille $F_{H_{sc}}$. Cette construction est valable pour tout $T_{k+1}(G, G')$ à partir d'un $T_k(G, G')$ existant appartenant à la même forêt F.

4.2.3.3 Obtention d'un arbre de recherche pour les ensembles

La famille d'arborescence précédemment définie nous permet d'une part de la manipuler en arbre de recherche représentant tous les appariements entre deux ensembles de sommets issus d'une fonction générative et d'autre part de pouvoir le construire en même temps que son exploration si cette fonction est une fonction générative récursive. Cet arbre est constitué de cette famille d'arbres ceci étant possible du aux propriétés récursives de la famille d'arbres.

Cet arbre de recherche représente l'espace de solutions des appariements uniquement si la famille d'arbres qui le constitue, contient l'ensemble des solutions et uniquement celles-ci.

A partir de la définition des familles de forêts nous démontrerons l'exhaustivité et la validité de la construction récursive.

Nous introduisons le théorème suivant qui concerne l'exhaustivité.

Théorème 4.2.1. *Soit une fonction générative H.*

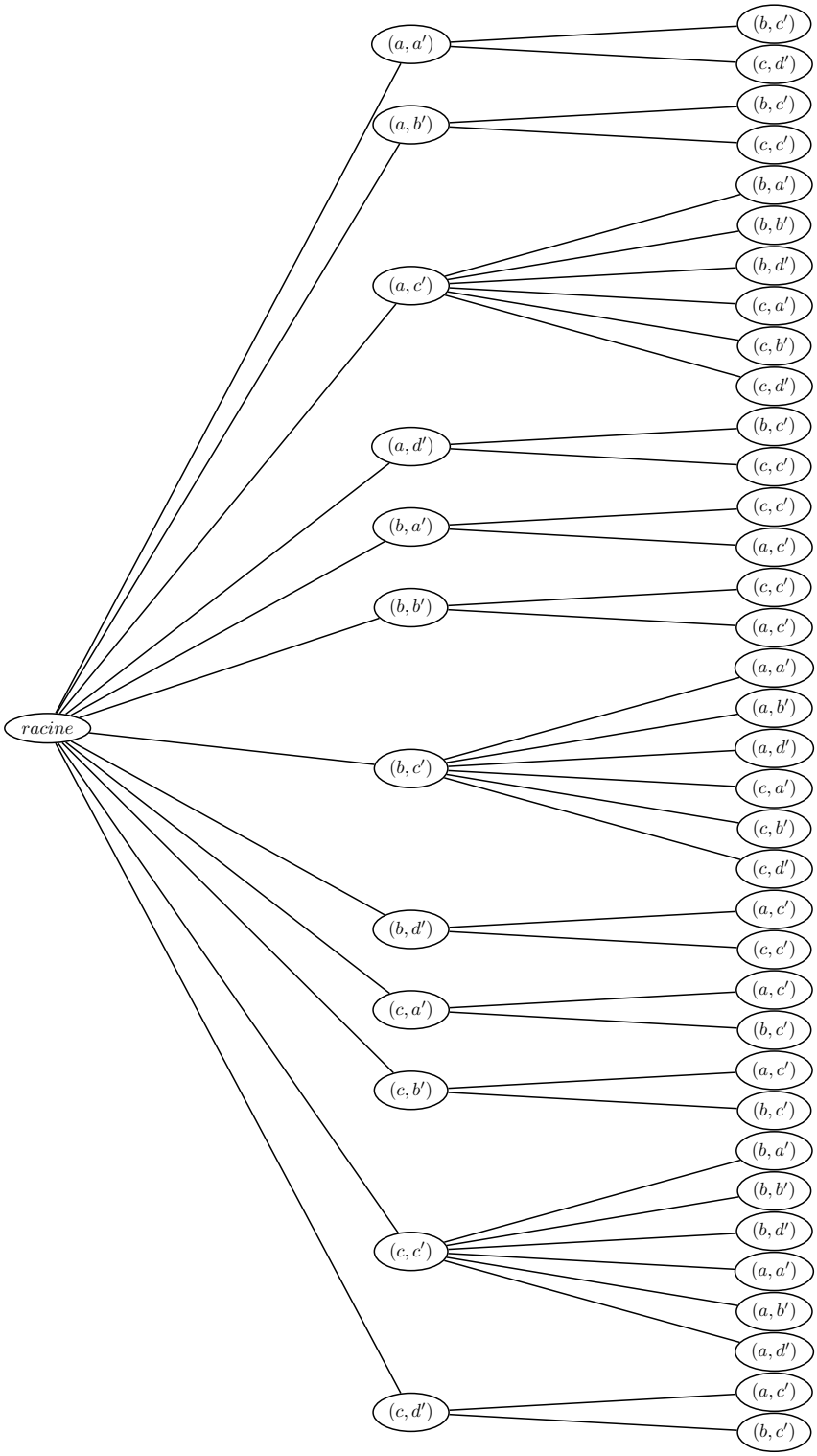


FIGURE 4.10 – $T_1(G, G')$ de la famille $F_{H_{sc}}$

Soit la famille d'arborescences F_H associée à H.
Pour tout couple de graphes G et G' on a :

$$App(H(G), H(G')) = \cup_{i=0}^{+\infty} \Xi(T_i(G, G'))$$

Avec T_i des arborescences de F_H restreintes à H^i.

Ce théorème justifie l'obtention de tous les appariements possibles.

Pour que nous puissions construire facilement un arbre $T_k(H_k, ., .)$ il faut nous assurer de la possibilité de le construire par une méthode récursive à partir de $T_{k-1}(H_{k-1}, ., .)$.

Théorème 4.2.2. *Soit une fonction générative H qui produit des ensembles de chemins récursifs, et la famille d'arborescence F_H associée à H.*
Alors, Pour tout couple de graphes G et G' on peut construire T_{k+1} à partir de T_k, G, G' et H_k

Les preuves de ces théoremes se trouvent en annexe. Nous obtenons un arbre lié à une définition de famille d'arbre, nous assurant de l'exhaustivité de l'espace de solution représenté par l'arbre ainsi que de l'unicité. Cet arbre est utilisé par la suite dans nos algorithmes présentés dans la section suivante.

4.3 Algorithmes

Nous présentons dans cette section des algorithmes qui s'appuient sur les outils que nous avons précédemment décrits pour calculer rapidement des similarités entre graphes, une fois la représentation implémentée par un type arbre. Ce type possède une méthode d'ajout d'un nœud qui calcule automatiquement la similarité en fonction d'opérateurs , et une méthode de création de l'arbre à la profondeur 1. L'intérêt de la représentation traitée dans les sections précédentes est de pouvoir mettre en œuvre simplement le calcul de différentes fonctions de similarité de graphes.

Le premier exemple traité montre la facilité de mise en place du calcul d'une fonction de similarité de type somme. Le deuxième exemple traite de la mise en place d'algorithme de recherche de maximum, appliquée au calcul d'un de nos noyaux de similarité présenté au chapitre précédent. Enfin nous verrons des implications et des notions générales pour l'utilisation d'algorithme dans cette représentation .

4.3.1 Exemple avec un noyau somme

Les noyaux à base de somme ont été utilisés dans le cadre des travaux de Kashima et de Suard. L'algorithme de calcul à l'aide de l'arbre de recherche a une formulation simple. En terme de complexité, Kashima a présenté une méthode non arborescente, de complexité moindre, mais qui suppose des contraintes sur les fonctions noyaux mineurs utilisés (k_v et k_e). L'utilisation d'arbre de recherche pour les sommes peut se discuter en terme de complexité si des convergences sont trouvées au niveau de la fonction de similarité sur graphes. Ces convergences peuvent permettre une expression du problème

en résolution de systèmes linéaires ce qui réduit la complexité. Par exemple, Kashima a utilisé des propriétés de convergence pour le résoudre comme un système linéaire. Néanmoins il est intéressant d'utiliser un algorithme utilisant l'arbre de recherche pour y voir la simplicité du calcul via cette méthode.

Pour présenter l'algorithme nous utilisons le noyau somme suivant :

$$K(G,G') = \sum_{h \in H(G)} \sum_{h' \in H(G')} K_C(h,h') \tag{4.1}$$

avec :

$$K_C(h,h') = k_v(v_0,v'_0) \prod_{i=1}^{|h|} (k_e(e_i,e'_i)k_v(v_i,v'_i))$$

H une fonction générative de chemins.

Pour rappel, on utilise la notation $K(G,G')$ alors que dans notre cas la fonction réelle est plutôt $K(H,G,G')$ puisque le choix de H influe sur la possibilité d'obtenir l'arbre et la complexité du calcul.

K_C peut s'écrire récursivement :

$$K_C(h_k,h'_k) = k_v(v_0,v'_0) \prod_{i=1}^{k-1} (k_e(e_i,e'_i)k_v(v_i,v'_i)) \times (k_e(e_k,e'_k)k_v(v_k,v'_k))$$

$$K_C(h_k,h'_k) = K_C(h_{k-1},h'_{k-1}) \times (k_e(e_k,e'_k) \times k_v(v_k,v'_k))$$

avec $h_k = h_{k-1}(e_k,e'_k)(v_k,v'_k)$. On définit, en s'appuyant sur la section 4.1 de ce chapitre, les trois opérateurs : op_v,op_e et op_{noeud}. Les opérateurs op_v et op_e se définissent par :

$$op_v = k_v$$

$$op_e = k_e$$

.

L'opérateur op_{noeud} est décomposable en deux sous opérateurs dans ce cas :

un opérateur (op_{n_1}) combinant la similarité entre h_{k-1} et h'_{k-1} et les similarités combinées des sommets et des arêtes nécessaires pour passer de h_{k-1} et h_{k-1} à h_k et h'_k soit $\times$ dans notre exemple,

un opérateur ($op_{n_2} = \times$) combinant les similarités des sommets et des arêtes nécessaires pour passer de h_{k-1} et h_{k-1} à h_k et h'_k .

$$op_{noeud}(n_{k+1}) = op_{n_1}(op_{n_2}(op_e(e,e'),op_v(v,v')),K_C(h_{k-1},h'_{k-1}))$$

Ces opérateurs ne sont utilisés qu'au moment de la création d'un nouveau nœud et ne sont donc pas présents dans le premier algorithme. Celui-ci suppose en effet qu'il existe une méthode initialisant l'arbre T_0 et une méthode créant l'arbre T_{k+1} à partir de T_k. Dans les algorithmes que nous décrivons par la suite, nous avons choisi de noter les similarités portées par un nœud n ou une feuille s comme des attributs d'objets dans certains langages : n.similarité ou f.similarité .

L'algorithme du calcul de la similarité $CalculSomme$(voir algo.1) prend en entrée les deux graphes G et G', la fonction générative d'ensembles H et un nombre réel (Borne). L'algorithme est simple, il crée un arbre de recherche T_0 pour H,G et G'. Puis il stocke la valeur de la somme des feuilles de cet arbre dans une variable $SommeArbre$. Puis tant que la différence entre cette somme et celle de toutes les similarités (la variable $Somme$ initialisée à 0) est supérieure à la borne : il additionne $Somme$ et $SommeArbre$ puis il développe l'arbre de un niveau. La somme des similarités des feuilles de cet arbre est mise dans la variable $SommeArbre$. Ainsi on obtient bien la somme de toute les similarités portées par toutes les feuilles de chaque arbre (de la famille correspondante) jusqu'à ce que l'apport soit négligeable.

Les parties non détaillées dans cet algorithme sont : la construction du premier arbre de profondeur 1. ($ConstruireArbre(H, G, G', 0)$) et l'obtention de l'arbre de profondeur augmentée de un. Les calculs de similarité des nœuds et des feuilles ne s'effectuent que lors de la construction de ceux-ci et ne sont donc pas présents dans l'algorithme $CalculSomme$.

La construction du premier arbre(algo.2) qui contient toutes les similarités entre les chemins de longueur 0, s'effectue en produisant tous les appariements possibles entre sommets qui sont des débuts de chemins dans $H(G)$ et $H(G')$. Puis pour chaque couple de sommets les rajouter comme nœud fils de la racine($root$) de l'arbre, avec leur similarité pour chacun. Enfin pour chaque nœud fils de la racine, leur rajouter tous les nœuds des couples de sommets produisant lors du parcours entre la racine et eux-même un appariement entre $H(G)$ et$H(G')$. En même temps on calcule la similarité portée par ces nouveaux nœuds.

Il ne nous reste plus qu'à définir la partie3 qui passe d'un arbre T_k à un arbre T_{k+1}, k étant supérieur ou égal à un. Comme précédemment, on va itérer sur les nœuds pour produire de nouveaux nœuds valides. Ces nœuds seront les feuilles de l'arbre. Ainsi la profondeur est augmentée de un.

Une amélioration possible de ce type d'algorithme, valable pour ce K_C, est de ne pas développer les fils d'un nœud dont la similarité est nulle. Ainsi on évite des multiplications inutiles, aboutissant à des similarités nulles. Une autre possibilité est d'effectuer le calcul en parallèle d'un certain nombre de branches de l'arbre.

4.3.2 Exemple avec le noyau de graphe proposé

Après nous être intéressés à la mise en place de calcul de similarité pour un cas simple, nous allons dans cette section aborder celle d'une fonction noyau que nous avons proposée dans le chapitre précédent. Ce noyau nous permet d'aborder l'exploration partielle de l'arbre à travers des algorithmes classiques de type recherche de maximum par profondeur d'abord ou de type "branch en bound".

4.3.2.1 Le noyau de graphe proposé

Le noyau considéré est un mélange de sommes et de recherche de maximums. Il se traduit par un choix au début de l'algorithme de $(v+v')$ nœuds. Ces nœuds représentent

Entrées : H une fonction générative, G et G' deux graphes
Sortie : un réel Somme
CalculSomme $(H, G, G', Borne)$
T_0 : $ConstruireArbre(H, G, G', 0)$
$SommeArbre = 0$
$Somme = 0$
$k = 1$
$Borne$: **un nombre réel fixant l'apport minimal autorisé**

Pour chaque feuille f de l'arbre T_0 **faire**
  | $SommeArbre = SommeArbre + f$.similarité
Fin Pour
Tant que ($|Somme - SommeArbre| > Borne$) **faire**
  | $Somme = Somme + SommeArbre$
  | Construire l'arbre T_k à partir de $T_{(k-1)}$, H, G et G'
  | *[rq : équivaut à (AugmenterArbre($T_{(k-1)}$, H, G et G'))]*
  | $SommeArbre = 0$
  | **Pour** chaque feuille f de l'arbre T_{Entier} **faire**
  | | $SommeArbre = SommeArbre + f$.similarité
  | **Fin Pour**
  | $Entier = Entier + 1$
Fait
Retourner $Somme$

Algorithme 1: Algorithme de calcul d'un noyau somme .Cet algorithme suppose l'existence de deux fonctions : le calcul de T_0 et la construction de T_{k+1} à partir de T_k. Le noyau calculé par cet algorithme est le K_{sum}. L'opérateur + sert à sommer les similarités des feuilles(f.similarité) entre elles à chaque arbre T puis à additionner ces sommes entre elles pour obtenir la valeur de cette formule : $\sum_{h \in H(G)} \sum_{h' \in H(G')} K_C(h, h')$.

Entrée :H une fonction générative, G et G' deux graphes
Sortie : un arbre T_0
ConstruireT_0
Soit T_0 un arbre de comparaison de profondeur 1
Créer T_0 avec une racine *root [le nœud root ne contient aucune information]*
Pour chaque sommet v dans $H(G)$ **faire**
  Pour chaque sommet v' dans $H(G')$ **faire**
    ajouter le nœud $n(v, v')$ au nœud *root* de T
    $n.similarit = k_v(v, v')$
  Fin Pour
Fin Pour

Algorithme 2: ConstruireT_0 construit un arbre de profondeur 1 qui ne comporte que les similarités de sommets dans notre cas.

les meilleurs appariements pour tous les sommets de G d'une part et tous les sommets de G' d'autre part. Il somme ensuite toutes les meilleures similarités issues de ces débuts d'appariements choisis. Ci-dessous l'équation de ce noyau :

$$\begin{aligned} K_{new}(G,G') &= \frac{1}{|V|} \sum_{\substack{v\in G \\ v'=s(v)}} \max_{h\in H_v(G)} \max_{\substack{h'\in H_{v'}(G') \\ |h'|=|h|}} K_C(h,h') \\ &+ \frac{1}{|V'|} \sum_{\substack{v'\in G \\ v=s'(v')}} \max_{h'\in H_{v'}(G)} \max_{\substack{h\in H_v(G) \\ |h|=|h'|}} K_C(h',h) \end{aligned} \tag{4.2}$$

avec,

$$s : V \to V'$$

$$s(v) = argmax_{x\in V'}(k_v(v,x)$$

et

$$s' : V' \to V$$

$$s'(v') = argmax_{x\in V}(k_v(x,v')$$

.

L'algorithme général 4 de ce noyau comporte deux parties. La première constitue la liste des nœuds initiaux, la deuxième à partir d'un nœud choisi et de son arbre trouve le meilleur appariement de chemins, de longueur k, représenté par un parcours dans l'arbre débutant par ce nœud. Au final toutes les similarités des feuilles représentant ces maximums sont sommées.

Entrées : Un arbre T , une fonction générative H et deux graphes G et G'
Sortie : T modifié
AugmenterProfondeur
$T_k(H, G, G')$: **un arbre de comparaison de profondeur k sur les ensembles** $(H(G), H(G'))$
Pour toute feuille $f(h, h')$ de T_k **faire**
  Pour tout e et v_2, hev_2 dans $H(G)$ **faire**
    Pour tout e' et v'_2, $he'v'_2$ dans $H(G')$ **faire**
      ajouter le nœud $fils(v_2, v'_2)$au nœud f
      fils.similarité= op_{n_1}(f.similarité, $op_{n_2}(op_e(e, e'), op_v(v_2, v'_2))$)
    Fin Pour
  Fin Pour
Fin Pour

Algorithme 3: Algorithme augmentant la profondeur d'un arbre de recherche T de un. Dans le cas du calcul de K_sum, on prend les opérateurs fixés précédemment. L'algorithme génère dans un premier temps des nœuds fils portants les appariements de sommets(v_2,v'_2) qui permettent de constituer dans $H(G)$ et $H(G')$ de nouveaux chemins hev_2 et $h'e'v'_2$. Diverses méthodes peuvent être utilisées en fonction de la fonction H concernée pour obtenir ces sommets(liste d'adjacences, liste tabou...). Une fois ce nœud créé on met à jour sa similarité (soit celle de l'appariement chemins qu'il représente).

Les recherches de maximum peuvent bien sûr être effectuées en parallèle (même si nous ne l'avons pas fait au niveau des expériences). Le choix d'algorithme performant pour la recherche de ces max dépend du comportement de la fonction K_C en fonction de la longueur des chemins considérés. On entend par là de savoir si la fonction va décroître à l'ajout d'une arête au chemin considéré. Cela revient à regarder si pour un nœud, la valeur de la fonction est toujours supérieure à celle portée par les nœuds fils.

C'est pourquoi, dans les paragraphes suivants, dans un premier temps, nous montrons une recherche par profondeur d'abord pour une fonction K_C décroissante avec la longueur du chemin. Dans un deuxième temps, nous effectuons une recherche par "branch and bound" pour des fonctions a priori non décroissantes.

4.3.2.2 Recherche du max dans le cas d'un K_C décroissant

On peut utiliser dans ce cas une recherche en profondeur d'abord d'un max et des coupures des branches de l'arbre. L'arbre est partiellement développé à la fin de l'algo-

Entrées : une fonction générative H, deux graphes G et G' et un entier K
Sortie : un réel SommeMax
CalculerNoyauSemiMax
Developper T_1 de H,G, G'
$listeNoeud$: **liste de nœud vide**

Pour tout $v \in V$ **faire**
  chercher $v' \in V'$ tel que $op_v(v, v')$ est maximisé
  stocker $noeud(v, v')$ dans $listeNoeud$
Fin Pour
Pour tout $v' \in V'$ **faire**
  chercher $v \in V$ tel que $op_v(v, v')$ est maximisé
  stocker $noeud(v, v')$ de T dans $listeNoeud$
Fin Pour
$SommeMax$: **0**
Pour tout nœud n de listeNoeud **faire**
  $SommeMax = SommeMax + ChercherMax(n, T, H, G, G', K)$.similarité
Fin Pour
retourner sommeMax

Algorithme 4: Algorithme général de calcul du noyau K_{new}. Il recherche dans un premier pour chaque sommet de G et G' son meilleur appariement selon l'opérateur op_v choisi(dans le cas du noyau k_v). Ensuite il crée les nœuds correspondant à ces appariements ($|V|+|V'|$ nœuds). A chacun de ces nœuds, on applique l'algorithme $ChercherMax$ (algo.6) pour obtenir la meilleure similarité entre chemins commençant par la paire de sommets de ce nœud. Une variante possible est de récupérer l'appariement lui même pour savoir quels chemins ont été les mieux appariés selon les opérateurs choisis.

ChercherMax(G,G',H,K : entier)
Entrées : une fonction générative H, deux graphes G et G' et un entier K
Sortie : un noeud d'arbre meilleurnoeud
T =arbre d'appariement développé à la profondeur 1 pour H, G et G'
n = meilleur nœud de T à la profondeur 1

Noeud meilleurnoeud=RechercherMeilleurNoeud(n,T,H,G,G',K)

MeilleurNoeud=RechercherAutresolution(
MeilleurNoeud,
meilleurnœud,
T,H,G,G',K,profondeur(n))
Retourner MeilleurNoeud

Algorithme 6: ChercherMax(G,G',H ,K)

rithme.

A l'initialisation on construit $T_0(H, G, G')$ et initialise n comme le noeud portant le meilleur appariement de sommet. On cherche un premier maximum à une profondeur k(algo.7). Pour cela on choisit le fils le plus prometteur à chaque fois jusqu'à la profondeur k. Une fois cette première estimation trouvée, on cherche s'il existe un autre maximum en remontant dans l'arbre (algo.8) pour tous les autres nœuds non visités dont la valeur est supérieure au maximum actuel pour la profondeur k. Si un nouveau maximum est trouvé il devient le maximum actuel.

Par exemple le K_C du précédent exemple est décroissant en fonction $|h|$:

$$K_C(h, h') = k_v(v_0, v'_0) \prod_{i=1}^{|h|} k_e(e_i, e'_i) k_v(v_i, v'_i))$$

en supposant que $k_e < 1$ et $k_v < 1$.

Plus on augmente la profondeur de l'arbre, plus les valeurs sont petites. Et la valeur

de similarité d'un nœud fils est toujours inférieure à celle du nœud père.

Entrées : un noeud N, un arbre T, une fonction générative H, deux graphes G et G', un entier K
Sortie : un noeud d'arbre MAX
RechercherMeilleurNoeud(N,T,H,G,G',K)
MAX : **une variable de type Noeud**
$MAX = NOEUDVIDE$
Pour chaque couple de $(ev, e'v')$ **faire**
  Si (hev appartient à $H(G)$ et $he'v'$ appartient à H(G')) **Alors**
    ajouter le nœud $f(v, v)$ comme fils de N
  Fin Si
  f.similarité=N.similarité$*k_e(e,e')k_v(v,v')$
  Si (MAX = VIDE) **Alors**
    MAX=f
  Sinon
    Si (f.similarité > MAX.similarité) **Alors**
      ALORS MAX=f
    Fin Si
  Fin Si
Fin Pour
Si (f.profondeur < K) **Alors**
  MAX=RechercherMeilleurNoeud(MAX, T, H, G, G', K)
Fin Si
Retourner MAX

Algorithme 7: *RechercherMeilleurNoeud*, cet algorithme renvoie soit un noeud avec une similarité plus élévée que N, soit N lui même. Il effectue sa recherche en profondeur et développe qu'un seul noeud de N.

```
Entrées : deux noeuds N et MAX, un arbre T, une fonction générative H,
deux graphes G et G', deux entiers i et K
Sortie : un noeud d'arbre MAX
RechercherAutresolution(MAX : Noeud,N : Noeud,T,H,G,G',K : entier
,i :entier)
Si ( N.profondeur > i) Alors
    Noeud Ancetre=N.parent
    Pour  tout fils f de Ancetre  faire
        Si ( f.profondeur<K
        et f non développé
        et f.similarité >MAX.similarité) Alors
            Noeud solPart=RechercherAutreConcurrent(MAX, f, T, H, G, G', K)
            Si ( solPart.similarité > MAX.similarité) Alors
                MAX=solPart
            Sinon
                Si ( f.profondeur=K et f.similarité >MAX.similarité)
                Alors
                    MAX=f.similarité
                Fin Si
            Fin Si
            MAX=RechercherAutreSolution(Max, Ancetre, T, H, G, G', K, i)

        Fin Si
    Fin Pour
Fin Si
Retourner MAX
```

Algorithme 8: *RechercherAutreSolution*, cet algorithme va chercher une autre solution possible en développant uniquement les nœuds qui peuvent avoir un de leur fils avec une valeur de similarité plus élevée que la solution courante MAX.

Entrées : un noeud MAX, un arbre T, une fonction générative H, deux graphes G et G', deux entiers i et K
Sortie : un noeud d'arbre MeilleurNoeud
RechercherAutreConcurrent(MAX, T, H, G, G', K, i)
MeilleurNoeud : **Noeud**
$MeilleurNoeud = RechercherMeilleurNoeud(n, T, H, G, G', K)$

Si (MAX<MeilleurNoeud) **Alors**
  | MAX=MeilleurNoeud
Fin Si
MeilleurNoeud=
RechercherAutreSolution(Max,meilleurnœud,T,H,G,G',K,n.profondeur,i)
Retourner MeilleurNoeud

Algorithme 9: *RechercherAutreConcurrent*.

4.3.2.3 K_C non décroissant (branch and bound)

Dans ce cas on ne peut plus appliquer telle quelle la recherche de solution précédente. Le branch and bound (séparer et borner) consiste à s'appuyer sur des fonctions d'estimation, que l'on note b_{min} et b_{max}, d'un nœud et de borne pour éliminer les branches inutiles. Ces fonctions doivent permettre de borner les résultats de similarité possibles dans les noeuds fils. Elles doivent se calculer facilement sans nécessiter de développer l'arbre. Le choix au niveau du nœud n'est plus basé uniquement sur la similarité portée par le nœud mais celle de b_{min},b_{max} et de la valeur de similarité du nœud max actuel. Si cette fonction de borne est toujours supérieure à la similarité pour un nœud donné alors la recherche du maximum est exhaustive. Sinon certaines branches de l'arbre qui peuvent contenir la solution finale sont écartées.

On cherche à construire b de façon à borner K_C. On suppose que nos noyaux mineurs sont bornés par une valeur α (i.e $k_v(v,v) <= \alpha$ et $k_e(e,e) <= \beta$). Dans ce cas on pose :

$$b(n(h,h'),k) = n.\text{similarité} + k_C(h_(k-|h|), h_(k-|h|)) - k_v(v,v)$$

"b" renvoie une valeur hypothétique supposant que l'appariement des sommets et des arêtes soient parfaits par la suite. Prenons ce K_C non monotone :

$$K_{C_{new2}}(h,h') = K_V(v_0,v'_0) \times \prod_{i=1}^{|h|} K_E(e_i,e'_i) \times (1 + K_V(v_i,v'_i)) \tag{4.3}$$

$$b(n(h,h'),k) = n.similarit + \alpha + \prod_{i=1}^{k-|h|} ((1+\alpha) * \beta) - \alpha$$

La fonction b est calculable aisément pour tout nœud. Il s'agit de modifier la fonction *RechercherAutreSolution* pour utiliser cette fonction et éliminer les développements inutiles.

```
Entrées : deux noeuds N et MAX, un arbre T, une fonction générative H,
deux graphes G et G', deux entiers i et K
Sortie : un noeud d'arbre MAX
RechercherAutreSolution
T : un arbre de comparaison
MAX :Noeud de T avec la meilleur similarité
N(h,h') : Noeud de T
H : fonction generative
G,G'deux graphes
K,i : entier

Si ( N.profondeur > i) Alors
    Noeud Ancetre=N.parent
    Pour  tout fils f de Ancetre faire
        Si    ( f.profondeur<K    et    f    non    developpé    et
        (f.similarité+b(f))>MAX.similarité) Alors
            Noeud solutionPartielle=RechercherAutreConcurrent(MAX,f,T,H,G,G',K)
            Si ( solutionPartielle.similarité > MAX.similarité) Alors
                MAX=solutionPartielle
            Sinon
                Si  ( f.profondeur=K  et  f.similarité  >MAX.similarité)
                Alors
                    MAX=f.similarité
                Fin Si
            Fin Si
        Fin Si
    Fin Pour
    MAX=RechercherAutreSolution(Max,Ancetre,T,H,G,G',K,i)

Fin Si
RETOURNER MAX
```

Algorithme 10: *RechercherAutreSolution*$(MAX, N, T, H, G, G', K, i)$ variante ; cet algorithme inclut des coupures issues du "branch and bound".

4.3.3 Complexités des algorithmes

La complexité du calcul dans notre représentation dépend d'une part de l'espace considéré, des fonctions de calculs impliquées et des algorithmes utilisés.

Si on prend le cas du calcul d'un noyau de graphe de type somme, la complexité est égale à l'espace des solutions soit l'ensemble des couples possibles entre deux chemins de G et G' de longueur k. A chaque niveau de l'arbre on calcule de nouveaux nœuds autant de fois qu'il y a de chemins de longueur égale à ce niveau. Par exemple, au niveau 0 on obtient tous les appariements de sommets et au niveau 1 tous les appariements d'arêtes possibles.

La complexité du niveau 2 peut être bornée dans un premier temps par le nombre de sommets des graphes et des arêtes : le nombre de nœuds au niveau 1 ($|E|.|E'|$) fois le nombre d'appariements possibles entre les sommets ($|V|.|V'|$). Cependant on peut affiner cette borne en utilisant le voisinage des noeuds du graphe. En effet le nombre de nœuds fils possibles pour le nœud(h, h') dépend des degrés(nombre de voisins directs) des sommets finissant les deux chemins h et h'. Au final il est possible d'approximer cette borne en utilisant le degré moyen du graphe(notons la d_{moy}). Le coût de calcul d'un noyau somme pour une longueur k, pour des graphe d'ordre n($|V| = n$) et de degré moyen d_{moy}, peut s'approximer par : $n^2.d_{moy}^{2k}$.

Si on impose des restrictions sur les ensembles de chemins considérés cela revient à remplacer le d_{moy} par un autre paramètre qui est le nombre moyen de sommets voisins utilisables et par conséquent de réduire le nombre de voisins et donc la complexité considérée.

Pour les graphes complets le degré moyen est : $n-1$. Dans ce cas la complexité devient de l'ordre de n^{2k}. Pour le noyau proposé on réduit les choix initaux de n^2 couples de sommets à $2n$ couples de sommets l'approximation devient :$2n.d_{moy}^{2k}$ et dans le cas d'un graphe complet on a : $(2n)n^{2(k-1)}$. La complexité au pire d'une recherche de maximum est semblable à la somme. Dans le cadre d'exploration d'arbre on s'intéresse au coût moyen de l'algorithme. Nous appuyerons sur les résultats de temps calculs dans nos experiences pour en produire une estimation de ce coût moyen dans le chapitre suivant pour un ensemble de graphes dont les sommets varient entre 15 à 25 sommets et un nombre moyen de voisins compris entre 4 et 5.

4.3.4 Généralités

Nous avons désormais une représentation des appariements sous la forme d'une forêt. Dans cette section nous allons voir la manipulation classique de cet arbre selon les types d'opérations utilisées dans nos fonctions de similarité. Ces fonctions présentent plusieurs niveaux d'opérations :

- les noyaux sur sommets et arêtes,
- les noyaux sur chemins,
- les sélections des similarités de chemins,
- les combinaisons des similarités,
- les restrictions de la taille des ensembles de chemins considérés.

Nous présenterons les 3 derniers types d'opérations puisque nous avons déjà présenté comment effectuer le calcul de similarité sur chemins et que le calcul des similarités d'arêtes et de sommets n'est pas le propos du chapitre.

4.3.4.1 Opérations autour des ensembles de chemins

Les fonctions de similarité de sacs de chemins sélectionnent et combinent des appariements de chemins en fonction de leur valeur de similarité. Ils sont dotés pour cela d'une fonction donnant une valeur de similarité entre chemins. Cette catégorie de fonction de similarité sur graphes se prête de manière générale bien à notre représentation. Une des conditions nécessaires est que la fonction de similarité entre chemins se plie aux règles que nous avons précédemment étudiées.

Restrictions de chemins Une restriction de chemins dans notre représentation est donnée par la fonction générative de chemins H. Par exemple pour que H_{sc} génère des chemins sans cycle dans les graphes, il est possible d'utiliser une fonction de test simple vérifiant qu'il n'existe pas deux fois le même sommet dans les chemins. Dans le cas de la génération des chemins en même temps que la construction de l'arbre, il suffit de vérifier que le nouveau sommet ajouté pour chaque graphe n'est pas déjà présent dans les chemins. Les opérations de restrictions de l'espace d'appariements hors conditions sur la similarité correspondent à utiliser des ensembles ou sacs de chemins à moindre cardinalité. Les opérations de sélection concernent, quant à elles, des choix d'appariements (par exemple le max) dont les valeurs de similarité seront par la suite combinées entre elles. Les combinaisons sont les opérations sur les valeurs entre les appariements choisis : somme, moyenne, produit...

Restrictions de sacs de chemins Les restrictions de sacs de chemins correspondent à un choix de fonction génératrice H. L'intégration de génération d'un chemin appartenant à H dans l'arbre peut se faire à partir de fonctions déterminant(cf algo.11)

si le chemin peut être généré par H.

Entrées : un arbre T, une fonction générative H, deux graphes G et G'
Sortie : T modifié
H : **fonction générative**
G et G' : **deux graphes**
n : **un nœud** $n = (h, h')$ **un arbre** $T(H, G, G')$
$estGenereeparH(h : chemins)$:
une fonction qui renvoie vraie
si h **peut être généré par** H.
listAdjG et listAdjG : **les listes d'adjacences de** G **et** G'.
v : **le dernier sommet de** h
v' : **le dernier sommet de** h'

Pour élément e de listAdj[v] **faire**
  Pour élément e' de listAdj[v'] **faire**
    soit l'arête $a = (v, e)$
    soit l'arête $a' = (v', e')$
    Si ($estGenereeparH(hae)$ et $estGenereeparH(h'a'e')$ sont vrais)
    Alors
      rajouter un nœud fils f à n tel que $f = (hae, h'a'e)$
    Fin Si
  Fin Pour
Fin Pour

Algorithme 11: Simple algorithme pour la restriction à des chemins issus de $H(G)$ et $H(G')$

Les restrictions de chemins possibles doivent pouvoir être liées à une fonction H telle que $H(G)$ ait des propriétés récursives. Le choix de $H(G)$ a une influence directe sur la largeur de l'arbre exploré.

Sélections d'appariements de chemins La sélection d'appariements peut s'effectuer soit sur les valeurs des similarités associées aux appariements, soit par d'autre contraintes. C'est celle qui aura le plus d'influence quant au choix d'exploration de l'arbre d'appariement. Une sélection de type maximum sera couplée avec des algorithmes de "branch and bound" ou recherche de min/max.

De ce fait, le type de sélection choisie fait varier la complexité. Les sélections qui identifient les différents chemins (exemple : chercher le meilleur appariement pour un chemin donné) augmentent la complexité par le mécanisme d'identification des chemins.

Notons que si l'on veut obtenir des fonctions noyaux il faudra rendre symétrique cette sélection d'appariements.

4.3.4.2 Pondérations

Les pondérations peuvent être soit directement introduites dans les fonctions de similarité et calculées au cours du calcul de la similarité, soit calculées de manière séparées afin de les utiliser comme éléments de coupure dans l'arbre.

Par conséquent, il est préférable de choisir des pondérations répondant aux mêmes critères que les fonctions de similarité. Par exemple les fonctions de probabilité sont de bonnes candidates pour les pondérations. De plus elles ont déjà été introduites pour des fonctions noyaux autour des graphes.

4.3.4.3 Cas particuliers

Effectuer des appariements de chemins de longueurs différentes Dans le cadre d'appariement de chemins il devient intéressant d'apparier des chemins de longueurs différentes pour pallier aux problèmes liés aux fusions ou aux divisions de sommets. Ces problèmes sont courants dans la segmentation en régions qui peut sur deux images très similaires découper une même zone en une région sur l'une des images et en deux sur l'autre. Dans ce cas les images ne possèdent pas le même graphe alors qu'elle sont proches.

Les appariements qui sont effectués dans le cadre de nos arbres sont exclusivement entre chemins de longueur semblable. Nous avons trouvé une solution permettant dans notre formalisme de comparer des chemins de longueur différente.

Nous avons donc introduit une arête particulière, la boucle (notée *loop*), ayant les mêmes propriétés quel que soit le sommet considéré. La boucle se traduit dans un chemin par la redondance d'un sommet : aab soit $a\ loop\ a\ e_{a,b}b$ (en affichant les arêtes).

Ainsi nous pouvons apparier les chemins ab et $a'b'c'$:

- aab avec $a'b'c'$
- abb avec $a'b'c'$

Nous avons plusieurs appariements possibles qui correspondent à des choix de fusion (par exemple $a'b'$ donnant a ou $b'c'$ donnant b). La valeur de la similarité s_h de ces appariements détermine l'appariement considéré comme le plus proche.

Nous imposons la propriété suivante aux fonctions de similarité entre arêtes :

Pour tout graphe G,

$\forall e \neq loop,\ s_e(loop, e) = s_e(e, loop) = \delta$

avec δ un paramètre.

On impose donc une similarité identique quelle que soit l'arête comparée à *loop*. Ceci revient à mettre la boucle dans une position particulière dans l'espace des arêtes. Notons

que si $\delta < min(s(e, e'))$ alors on rend pratiquement impossible l'appariement d'arêtes boucle avec non boucle. Alors que l'inverse $\delta = max(s(e, e'))$ favorise cette situation.

Dans le cadre de similarité normalisée ($s_e(e, e) = constante$), si δ est strictement égal à cette constante, alors une boucle se comporte comme un élément neutre. Le coût de la fusion ou division dans ce cas est porté uniquement sur les sommets.

Cette solution a été testée dans nos expérimentations présentées dans le chapitre suivant.

Prise en compte de l'information de la non adjacence Dans notre cadre de recherche qui est la mise en correspondance de graphes issus de segmentation sur les images à comparer, on peut voir le fait que deux régions soient non adjacentes comme une information supplémentaire. Or nos graphes utilisés sont des graphes d'adjacences. Une solution envisageable quoique possiblement coûteuse, est de considérer un graphe complet issu du graphe d'adjacences. Ce graphe est constitué de deux type d'arêtes : celle représentant une adjacence et les autres.

Afin d'éviter un coût en complexité on peut imposer l'impossibilité d'apparier une arête d'adjacence avec une arête de non-adjacence. Cette opération est une opération de sélection d'appariement et donc de coupure dans l'arbre. Nous n'avons pas effectué d'expériences autour de cette représentation d'image.

Conclusion

Nous avons présenté des algorithmes de calculs pour nos noyaux sur graphes basés sur des représentations arborescentes de l'espace d'appariement. Afin de garantir que cette représentation pouvait être utilisée comme un arbre de recherche et que les calculs effectués étaient exacts, nous avons formalisé cette représentation. Cette représentation est exhaustive et chaque appariement de chemins y est unique. De plus, elle ne contient pas d'appariement non existant.

Nous avons montré dans ce cadre les diverses utilisations d'algorithmes et notamment spécifié une catégorie de fonction de borne pour l'algorithme "branch and bound" en fonction du noyau sur chemin choisi.

Enfin nous avons présenté quelques modifications possibles pour intégrer des pondérations, des appariements de chemins de longueurs inégales et la prise en compte de l'information de non adjacences.