Atelier 7

Graphes et appariement d’objets complexes

Mohand Boughanem et Hamamache Kheddouci

Actes de l'atelier GAOC :

Graphes et Appariement d'Objets Complexes

En conjonction avec la conférence EGC'2010
26-29 Janvier 2010,
Hammamet, Tunisie

Comité d'organisation :

Mohand BOUGHANEM, Hamamache KHEDDOUCI

http://www710.univ-lyon1.fr/~gaoc/

Préface

Ces actes regroupent les articles présentés lors de l'Atelier GAOC : Graphes et Appariement d'Objets Complexes qui a eu lieu à Hammamet en Tunisie le 26 janvier 2010 en conjonction avec la conférence EGC'2010 : 10ième Conférence Internationale Francophone sur l'Extraction et la Gestion des Connaissances.

Les applications telles que les bibliothèques numériques, les systèmes de médiation ou les architectures à base de services Web produisent des masses de données générant un grand nombre de modèles destinés a être stockés dans l'objectif d'être accessibles pour des utilisateurs (clients) exprimant un besoin spécifique, traduit totalement ou en partie à l'aide d'une requête. Ces modèles, décrivant des objets complexes tels que les documents, les processus métiers ou les ontologies relatives à des domaines de connaissances, sont de ce fait hétérogènes et riches du point de vue sémantique. Actuellement, la plupart des applications utilisant de tels modèles, se basent sur une représentation sous forme de diagrammes ou graphes dont l'accès se décline par un processus d'appariement qui s'appuie uniquement sur des propriétés de structures (topologiques). La représentation sous forme de graphes devient complexe dès qu'elle tente de capturer toute la sémantique du modèle qu'elle représente. Par conséquent, les problèmes d'appariement de ces graphes deviennent difficiles à résoudre.

Par ailleurs, l'appariement de graphes est connu dans la littérature des graphes sous le nom de « graph matching problem ». Ce problème est difficile. Plusieurs algorithmes et heuristiques ont été développés pour des matchings exacts (isomorphismes) ou inexacts (placements, plongements,…) sur des familles de graphes ou des graphes quelconques.

Notre objectif dans cet atelier a été d'offrir aux communautés de chercheurs académiques ou industriels travaillant sur des thématiques liées aux : documents, graphes, services web, web sémantique et autres, une tribune de présentation, de synthèse, de discussion et de démonstration de leurs résultats sur l'appariement d'objets à base de graphes.

Nous remercions les auteurs pour leurs contributions (articles et démonstrations), les membres du comité de programme, les membres du comité d'organisation et les sponsors.

Les articles soumis à GAOC 2010 ont été évalués par deux membres du comité de programme suivant : Mohand Boughanem, IRIT - Université Paul Sabatier, Toulouse; Mokrane Bouzeghoub, PRISM - Université de Versailles; Sylvie Calabretto, LIRIS - INSA de Lyon; Lyes Dekar, LIESP - Université Claude Bernard, Lyon1;. Eric Duchêne, LIESP - Université Claude Bernard Lyon1; Brice Effantin, LIESP - Université Claude Bernard, Lyon1; Daniela Grigori, PRISM - Université de Versailles; Mohand-said Hacid, LIRIS - Université Claude Bernard, Lyon1; Mohammed Haddad, LIESP - Université Claude Bernard, Lyon1;. Allel Hadjali, IRISA - Université de Rennes 1; Hamamache Kheddouci, LIESP - Université Claude Bernard Lyon1; Ludovic Liétard, IRISA - Université de Rennes 1; Guy Melançon, LABRI - Université de Bordeaux 1; Jean-Marc Petit, LIRIS - INSA de Lyon; Daniel Rocacher, IRISA - Université de Rennes1; Karen Sauvagnat, IRIT - Université Paul Sabatier Toulouse; Hamida Seba, LIESP - Université Claude Bernard, Lyon1.

Mohand BOUGHANEM, Hamamache KHEDDOUCI

Appariement analogique de séquences et d'arbres : Application au parsing automatique.

Anouar Ben Hassena*, Laurent Miclet*

*IRISA / Université de Rennes 1 - ENSSAT
6 rue de Kerampont, B.P. 80518, F-22305 Lannion Cedex
{benhasse,miclet}@enssat.fr

Résumé. Dans cet article, nous faisons d'abord une présentation de nos travaux sur l'appariement par proportion analogique entre séquences et entre arbres, puis nous en présentons une application à l'apprentissage à base de corpus de l'arbre syntaxique d'une phrase donnée.

1 Introduction

1.1 Appariement analogique de séquences et d'arbres

La proportion analogique est une relation entre quatre objets qui exprime que la manière de transformer le premier objet en le second est la même que la manière de transformer le troisième en le quatrième. En appelant les objets O_1, O_2, O_3 et O_4, leur relation de proportion analogique s'énonce : « O_1 est à O_2 comme O_3 est à O_4 » et se note $O_1 : O_2 : : O_3 : O_4$.

Par exemple, les quatre séquences de lettres suivantes forment une proportion analogique :

déridés : ridons : : démarchés : marchons

En effet, pour passer de la première séquence à la seconde, il faut enlever le préfixe « dé » et remplacer le suffixe « és » par le suffixe « ons ». Le passage de la troisième séquence à la quatrième se fait exactement par les mêmes opérations. Une définition plus précise de la proportion analogique est donnée dans le paragraphe 2.

Les proportions analogiques entre séquences ont été étudiées d'un point de vue algorithmique d'abord par Lepage (2003), puis par Stroppa et Yvon (2004) et notre équipe (Miclet et al. (2008)). Pour notre part, à partir d'une relation de proportion analogique sur les lettres, nous avons défini une quantité appelée « dissemblance analogique » (DA) qui mesure de combien (au minimum) quatre séquences « ratent » la proportion analogique. Nous avons aussi défini des algorithmes de programmation dynamique pour deux tâches : la première, de mesurer la DA entre quatre séquences ; la seconde, de construire à partir de trois séquences la (ou les) quatrième(s) à DA minimale. Ce travail est détaillé au paragraphe 3.

A partir d'algorithmes de calcul d'appariement entre deux arbres (Jiang et al. (1994)), nous avons étendu la notion de proportion analogique aux arbres étiquetés et ordonnés, défini de manière analogue la DA entre quatre arbres et proposé deux algorithmes réalisant les mêmes tâches dans les structures d'arbres (Ben Hassena et Miclet (2009)). La Figure 1 illustre la notion de proportion analogique entre arbres ; le paragraphe 4 lui est consacré.

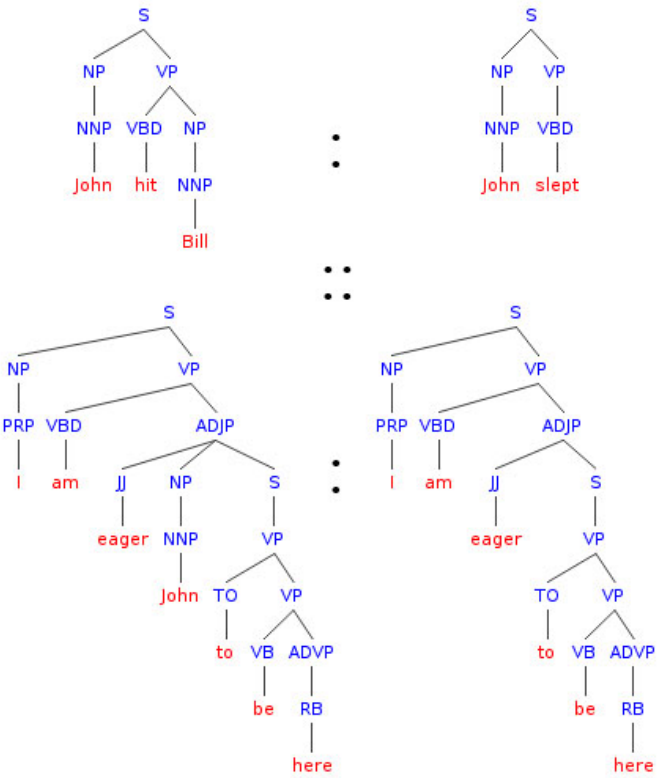

FIG. 1 – *Exemple de quatre phrases appariées en proportion analogique par leurs séquences*

NNP VBD NNP : NNP VBD : : PRP VBD JJ NNP TO VB RB : PRP VBD JJ TO VB RB

et appariées aussi en proportion analogique par leurs arbres syntaxiques.

1.2 Expériences

Nous appliquons ces algorithmes d'appariement analogique entre objets structurés au parsing automatique selon la démarche suivante. Nous disposons d'un corpus de phrases parsées, c'est à dire qu'à chacune de ces phrases est associé son arbre syntaxique. Les phrases sont composées comme des séquences de catégories grammaticales, que l'on trouve comme étiquettes des feuilles des arbres syntaxiques. Aux nœuds des arbres se trouvent des étiquettes syntaxiques comme « groupe verbal », etc. Notre hypothèse de base est simple : si quatre phrases ont leurs séquences en proportion analogique (ou à DA faible), alors leurs arbres syntaxiques sont aussi en proportion analogique (ou à DA faible). Elle a été vérifiée expérimentalement sur une partie de nos données.

Cette hypothèse mène à une méthode de génération automatique d'arbre syntaxique (de parsing automatique). Notons (S_i, T_i) une phrase du corpus, avec $i = 1, m$, constituée d'une

séquence et d'un arbre syntaxique. Soit S_0 une séquence extérieure au corpus, dont on cherche l'arbre syntaxique T_0. On trouve dans le corpus un triplet de phrases (S_i, T_i), (S_j, T_j) et (S_k, T_k) tel que les séquences S_0, S_i, S_j et S_k soient en proportion analogique. T_0 se construit alors comme l'arbre à DA minimale de T_i, T_j et T_k.

Nous présentons au paragraphe 5 des expériences sur un corpus de 316 phrases extrait de la base the Penn Wall Street Journal Treebank, divisé aléatoirement en 216 phrases pour l'apprentissage et 100 pour le test. Le protocole de base décrit ci-dessus est un peu modifié pour accélérer les calculs. Les résultats provisoires donnent une restitution exacte ou presque (DA nulle ou égale à 1) de l'arbre à partir de la séquence dans 85% des cas.

2 La proportion analogique et ses applications

2.1 La proportion analogique

Definition 1 *Comme défini dans Lepage (2003), une proportion analogique sur un ensemble $\mathbb{E}$ est une relation sur $\mathbb{E}^4$ telle que, pour tout quadruplet d'éléments A, B, C et D en relation dans cet ordre (ce qui est noté $A : B :: C : D$), on ait :*

1. $A : B :: C : D \Leftrightarrow C : D :: A : B$
2. $A : B :: C : D \Leftrightarrow A : C :: B : D$

De plus, on doit avoir pour chaque couple d'éléments : $A : B :: A : B$

La définition de $A : B :: C : D$ implique cinq autres relations :
$B : A :: D : C$, $D : B :: C : A$, $D : C :: B : A$, $B : D :: A : C$, $C : A :: D : B$.

Au total, les deux premiers axiomes de la définition 1 conduisent donc à huit proportions analogiques équivalentes.

On appelle *équation analogique* une expression du type $A : B :: C : X$, que l'on cherche à résoudre en X. Une équation analogique peut avoir zéro, une seule ou plusieurs solutions. On ajoute souvent aux axiomes qui définissent la proportion analogique celui du *déterminisme*, qui exige que l'équation suivante n'ait qu'une seule solution :

$$A : B :: A : X \quad \Rightarrow \quad X = B$$

Il y a 24 façons d'arranger quatre objets en une proportion analogique, qui se réduisent à trois classes d'équivalence dont les représentants sont par exemple : $A : B :: C : D$, $A : B :: D : C$ et $A : C :: D : B$.

2.2 Sémantique et exemples

La relation $A : B :: C : D$ s'interprète comme "A est à B comme C est à D", ce qui signifie que pour transformer A en B, il faut faire les mêmes opérations que pour transformer C en D. De même, d'après la définition 1, pour transformer A en C, il faut faire les mêmes opérations que pour transformer B en D.

La proportion analogique et sa sémantique ont été particulièrement explorées dans les cas suivants :

$\{0,1\}^d$ **:** Quatre objets définis par des vecteurs d'attributs binaires sont en proportion analogique lorsque, sur chaque coordonnée, on a l'une des 6 relations suivantes : $0:0::1:1$, $1:1::0:0$, $1:0::1:0$, $0:1::0:1$, $0:0::0:0$ ou $1:1::1:1$.

$\mathbb{R}^d$ **:** Quatre objets définis comme des vecteurs d'attributs numériques sont en proportion analogique additive quand $\overrightarrow{AC} = \overrightarrow{BD}$, c'est à dire que sur chaque coordonnée, on a :

$$A_i : B_i :: C_i : D_i \;\Leftrightarrow\; A_i + D_i = B_i + C_i\,.$$

$\mathbb{R}^d$ **:** Quatre objets définis comme des vecteurs d'attributs numériques sont en proportion analogique multiplicative quand on a sur chaque coordonnée

$$A_i : B_i :: C_i : D_i \;\Leftrightarrow\; A_i \times D_i = B_i \times C_i\,.$$

$\Sigma^\star$ **:** La théorie et les algorithmes de la proportion analogique sur les séquences ont été étudiés par Lepage (2003); Stroppa et Yvon (2005); Miclet et al. (2008); Hofstadter et the Fluid Analogies Research Group (1994).

Arbres ordonnés : Ce sujet n'a été abordé avant nous, à notre connaissance, que dans Stroppa et Yvon (2005).

De plus, l'étude de la proportion analogique comme un outil de raisonnement en logique propositionnelle (et son interprétation en théorie des ensembles) et en logique floue a été abordée dans Lepage (2003); Miclet et Prade (2009). D'autres études sur l'analogie dans les groupes non commutatifs (permutations, matrices) se trouvent dans Barbot et Miclet (2009).

3 Appariement analogique de séquences

3.1 Proportions analogiques sur les séquences

Nous utilisons dans ce paragraphe les notions usuelles de la théorie des langages : alphabet Σ, mot (ou séquence) sur $\Sigma^\star$, longueur d'un mot, mot vide ϵ, facteur, sous-séquence. Par exemple, sur l'alphabet $\Sigma = \{a, b\}$, la séquence $aabbaa$ est un élément de $\Sigma^\star$ de longueur $|aabbaa| = 6$, $bbaa$ en est un facteur et aba en est une sous-séquence.

Nous ajoutons une nouvelle lettre à Σ, que nous notons $\backsim$, pour obtenir un alphabet augmenté Σ'. Son interprétation est celle d'un symbole « vide » nécessaire pour les sections qui suivent.

Definition 2 (Equivalence sémantique.) *Soit x une séquence de $\Sigma^\star$ et y une séquence sur $\Sigma'^\star$. Les séquences x et y sont* sémantiquement équivalentes *si la sous-séquence de y composée des lettres de Σ est x. Nous notons cette relation par $\equiv$. Par exemple : $ab \backsim a \backsim a \equiv abaa$.*

Un alignement est une correspondance lettre à lettre entre quatre séquences, dans lesquelles des lettres $\backsim$ peuvent être insérées de façon à ce qu'elles prennent la même longueur. La correspondance $(\backsim, \backsim, \backsim, \backsim)$ n'est pas permise.

Definition 3 (Alignement entre quatre séquences.) *Un* alignement *entre quatre sequences $u, v, w, x \in \Sigma^*$, est un mot z sur l'alphabet $(\Sigma \cup \{\backsim\})^4 \setminus \{(\backsim, \backsim, \backsim, \backsim)\}$ dont la projection sur la première, la seconde, la troisième et la quatrième composante sont sémantiquement équivalentes à u, v, w and x.*

Nous supposons qu'il existe une relation de proportion analogique dans Σ', c'est-à-dire que pour chaque quadruplet a, b, c, d dans Σ', la relation $a : b :: c : d$ vaut soit $VRAI$ soit $FAUX$. Nous proposons maintenant de définir la proportion analogique entre quatre séquences d'objets en utilisant à la fois la proportion analogique entre les objets qui les composent et l'alignement entre les quatre séquences.

Definition 4 (Proportion analogique entre séquences.) *Soit u, v, w and x quatre séquences de Σ, sur lequel existe une proportion analogique. Nous disons que u, v, w et x sont en* proportion analogique *s'il existe quatre séquences u', v', w' and x' de même longueur dans Σ', avec les propriétés suivantes :*

1. *$u' \equiv u$, $v' \equiv v$, $w' \equiv w$ et $x' \equiv x$.*
2. *$\forall i \in [1, n]$ $u'_i : v'_i :: w'_i : x'_i$ sont des proportions analogiques dans Σ'.*

Par exemple, soit $\Sigma' = \{a, b, c, \backsim\}$. L'alignement suivant entre les quatre séquences $acab$, cb, $aaaa$ et aa est une proportion analogique sur $\Sigma^\star$:

$$\begin{array}{cccc} a & c & a & b \\ \backsim & c & \backsim & b \\ a & a & a & a \\ \backsim & a & \backsim & a \end{array}$$

3.2 Dissemblance analogique entre séquences

La dissemblance analogique entre séquences mesure de combien quatre séquences « ratent » la proportion analogique. Elle est définie de manière générale dans Miclet et al. (2008) ; nous en donnons ici une définition plus simple, mais suffisante pour la suite. Nous commençons par définir une dissemblance analogique DA sur l'alphabet, qui est le nombre minimum de lettres qu'il faut changer dans un quadruplet pour qu'il soit en proportion analogique. Par exemple, le quadruplet (a, a, b, b), qui est en proportion analogique, est à dissemblance nulle, ce que l'on note : $DA(a, b, a, b) = 0$. On a de même $DA(a, b, c, b) = 1$ puisqu'il suffit de remplacer c par a pour retrouver une proportion exacte. On a aussi $DA(a, b, c, d) = 2$ et aucun quadruplet de lettres n'a une DA supérieure à 2.

En passant aux séquences, on définit le coût d'un alignement entre quatre séquences comme la somme des DA sur les colonnes de l'alignement. La dissemblance analogique entre quatre séquences est alors le coût de l'alignement de coût minimal.

Par exemple, les quatre séquences $acab$, cb, $aaaa$ et ab ont une DA égale à un, par l'alignement optimal :

$$\begin{array}{cccc} a & c & a & b \\ \backsim & c & \backsim & b \\ a & a & a & a \\ \backsim & a & \backsim & b \end{array}$$

3.3 Algorithmes sur les séquences

Nous avons décrit dans Miclet et al. (2008) deux algorithmes de programmation dynamique qui permettent

– à partir de quatre séquences, de calculer leur DA et leur alignement optimal ;
– à partir de trois séquences, de calculer une quatrième à DA minimale avec les trois premières.

Dans le paragraphe suivant, nous traitons les mêmes problèmes sur des arbres étiquetés.

4 Appariement analogique d'arbres

Les objets sur lesquels nous voulons maintenant réaliser une tâche d'appariement sont des arbres ordonnés [1] et étiquetés. Nous nous intéressons de manière plus générale au cas des forêts, où une forêt F est un ensemble fini ordonné d'arbres. Comme illustré dans la figure 2, nous notons :

– $F[v]$ la forêt obtenue à partir de l'arbre $T[v]$ en supprimant la racine v,
– $F[v_s, v_d]$ la forêt partielle issue de $F[v]$ composée par les arbres adjacents enracinés en $v_s, ..., v_d$.

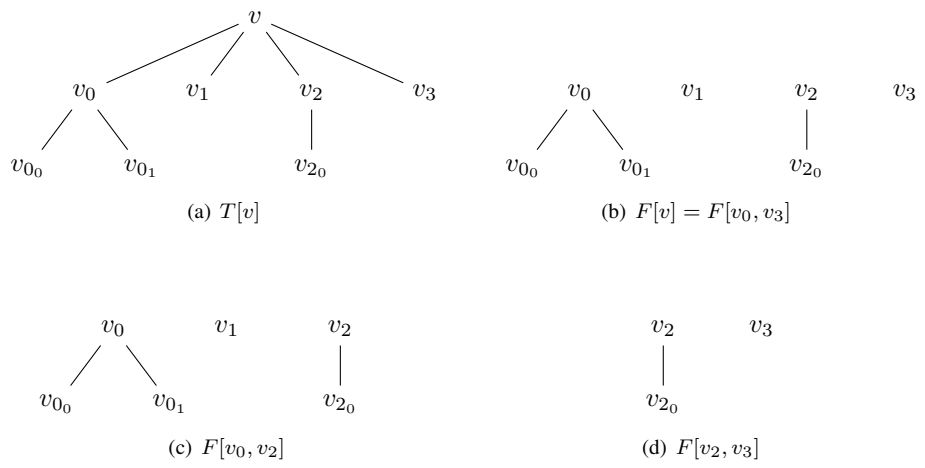

FIG. 2 – *Terminologie pour les forêts*

A partir d'algorithmes de calcul d'appariement entre deux arbres (Jiang et al. (1994)), nous avons étendu la notion de proportion analogique aux arbres étiquetés et ordonnés, défini la DA entre quatre arbres et proposé un algorithme d'appariement analogique entre quatre arbres. En ce qui concerne les détails de notre méthodologie, la référence (Ben Hassena et Miclet (2009)) en donne les principes, les définitions et la démarche suivie.

1. Un arbre est *ordonné* s'il existe un ordre parmi les fils de chaque nœud.

4.1 Calcul de la dissemblance analogique entre quatre arbres

Le coût d'appariement ou « dissemblance analogique » (DA) est une quantité qui mesure de combien (au minimum) quatre arbres « ratent » la proportion analogique. Dans ce qui suit, nous présentons deux algorithmes de programmation dynamique pour deux tâches : la première, mesurer la DA entre quatre arbres ; la seconde, construire à partir de trois arbres la (ou les) quatrième(s) à DA minimale.

L'entrée de l'algorithme est l'alphabet Σ' des étiquettes dans lequel est définie une dissemblance analogique comme au paragraphe 3.2. La sortie de cet algorithme est la DA entre quatre arbres.
Notons $\prod$ la dissemblance analogique entre arbres et DA la dissemblance analogique sur les étiquettes des nœuds des arbres. La récurrence suivante forme la base de notre algorithme *AnaTree* qui calcule la dissemblance analogique entre quatre arbres.

Algorithme 1 Algorithme AnaTree

ENTRÉES: Σ'
SORTIES: la dissemblance analogique entre quatre arbres

Récurrence :

$$\prod_{F_3[k_r,k_f]\ ,\ F_4[l_p,l_h]}^{F_1[i_s,i_d]\ ,\ F_2[j_t,j_q]} =$$

$$\min \begin{cases} DA(\backsim,\backsim,\backsim,l) + \underset{\substack{s\le u\le d+1\\ t\le v\le q+1\\ r\le w\le f+1}}{min} \{\prod_{F_3[k_r,k_{w-1}]\ ,\ F_4[l_p,l_{h-1}]}^{F_1[i_s,i_{u-1}]\ ,\ F_2[j_t,j_{v-1}]} + \prod_{F_3[k_w,k_f]\ ,\ F_4[l_h]}^{F_1[i_u,i_d]\ ,\ F_2[j_v,j_q]}\} \\ DA(\backsim,\backsim,k,l) + \underset{\substack{s\le u\le d+1\\ t\le v\le q+1}}{min} \{ \prod_{F_3[k_r,k_{f-1}]\ ,\ F_4[l_p,l_{h-1}]}^{F_1[i_s,i_{u-1}]\ ,\ F_2[j_t,j_{v-1}]} + \prod_{F_3[k_f]\ ,\ F_4[l_h]}^{F_1[i_u,i_d]\ ,\ F_2[j_v,j_q]}\} \\ DA(\backsim,j,k,l) + \underset{s\le u\le d+1}{min} \{ \prod_{F_3[k_r,k_{f-1}]\ ,\ F_4[l_p,l_{h-1}]}^{F_1[i_s,i_{u-1}]\ ,\ F_2[j_t,j_{q-1}]} + \prod_{F_3[k_f]\ ,\ F_4[l_h]}^{F_1[i_u,i_d]\ ,\ F_2[j_q]}\} \\ DA(i,j,k,l) + \prod_{F_3[k_r,k_{f-1}]\ ,\ F_4[l_p,l_{h-1}]}^{F_1[i_s,i_{d-1}]\ ,\ F_2[j_t,j_{q-1}]} + \prod_{F_3[k_f]\ ,\ F_4[l_h]}^{F_1[i_d]\ ,\ F_2[j_q]} \end{cases}$$

Nous n'avons donné que quatre des quinze cas possibles pour calculer le coût d'appariement. Les autres sont similaires. Par exemple, dans le cas de $DA(\backsim,\backsim,\backsim,l)$, il est similaire de traiter les cas des $DA(\backsim,\backsim,k,\backsim)$, $DA(\backsim,j,\backsim,\backsim)$ et $DA(i,\backsim,\backsim,\backsim)$.

Nous avons montré que la complexité de notre algorithme d'appariement de quatre arbres est de $O(|T|^4 * (\text{ degré }(T))^4)$ en temps et $O(|T|^4 * \text{ degré }(T))$ en espace. Par convention, degré(T) est le degré du nœud de T de degré maximal.

4.2 Résolution d'équation analogique et prédiction d'arbre

Quand l'un des quatre éléments est inconnu, la proportion analogique devient une équation analogique. Par exemple résoudre l'équation sur les lettres $a : b :: a : x$, consiste à calculer l'ensemble des lettres x satisfaisant la proportion analogique (ici, $x = b$). Cette équation possède d'une manière générale zéro, une ou plusieurs solutions exactes. Dans le cas où il n'y

pas de solution, la notion de dissemblance analogique va permettre d'autoriser le concept de solution approchée. Si cette résolution d'équation analogique est triviale dans l'ensemble des lettres, il n'est pas simple d'écrire un algorithme capable de résoudre une telle équation sur les arbres, surtout si on cherche une ou plusieurs solutions approchées.

Nous présentons dans ce paragraphe un algorithme appelé *SolvTree*, qui produit l'ensemble des meilleures solutions approchées et/ou exactes (DA faible ou nulle). Le principe est de réaliser un alignement des trois arbres en parcourant à chaque étape huit cas possibles, dans chacune nous calculons le coût de prédiction de l'étiquette d'un nœud de l'arbre à générer.

En notant x l'étiquette du nœud produite à chaque étape, la récurrence suivante forme la base de notre algorithme. Nous n'avons donné que trois des huit cas possibles.

Algorithme 2 Algorithme SolvTree

ENTRÉES: Σ'

SORTIES: L'arbre généré avec le coût de prédiction.

Récurrence :

$$\prod_{F_3[k_r,k_f]\ ,?}^{F_1[i_s,i_d]\ ,\ F_2[j_t,j_q]} =$$

$$\min \begin{cases} \underset{x\in\Sigma'}{Min}\ DA(\backsim,\backsim,k,x) + \underset{\substack{s\le u\le d+1\\ t\le v\le q+1}}{min}\{\prod_{F_3[k_r,k_{f-1}]}^{F_1[i_s,i_{u-1}]\ ,\ F_2[j_t,j_{v-1}]} + \prod_{F_3[k_f]}^{F_1[i_u,i_d]\ ,\ F_2[j_v,j_q]}\} \\ \underset{x\in\Sigma'}{Min}\ DA(\backsim,j,k,x) + \underset{s\le u\le d+1}{min}\{\prod_{F_3[k_r,k_{f-1}]}^{F_1[i_s,i_{u-1}]\ ,\ F_2[j_t,j_{q-1}]} + \prod_{F_3[k_f]}^{F_1[i_u,i_d]\ ,\ F_2[j_q]}\} \\ \underset{x\in\Sigma'}{Min}\ DA(i,j,k,x) + \prod_{F_3[k_r,k_{f-1}]}^{F_1[i_s,i_{d-1}]\ ,\ F_2[j_t,j_{q-1}]} + \prod_{F_3[k_f]}^{F_1[i_d]\ ,\ F_2[j_q]} \end{cases}$$

En réalité, nous sauvegardons à chaque étape non seulement le coût de prédiction, comme indiqué ci-dessus, mais aussi la ou les étiquette(s) x des nœuds trouvées par la résolution de l'équation analogique au long de la progression. Quand le calcul est terminé, un *backtracking*, grâce à cette sauvegarde des étiquettes, construit l'arbre généré avec la dissemblance analogique optimale. La complexité de cet algorithme est de $O(|T|^3 * (\text{ degré } (T))^3)$ en temps et $O(|T|^3 * \text{ degré } (T))$ en espace.

5 Application au parsing automatique

Dans cette section nous présentons une utilisation de l'appariement de séquences et d'arbres dans le traitement automatique des langues : le parsing des langages naturels. Le parsing s'applique dans divers problèmes notamment dans la traduction automatique, le problème de Questions-Réponses et même dans les système de synthèse de parole. Il consiste à effectuer l'analyse syntaxique d'une phrase, c'est à dire que nous avons en entrée une phrase composée comme une séquence de catégories grammaticales et nous cherchons en sortie à construire la structure syntaxique de cette phrase comme un arbre dans les nœudsduquel se trouvent des étiquettes syntaxique comme "Groupe Verbal", "Groupe Nominal", etc. La figure 3 montre un exemple de parsing.

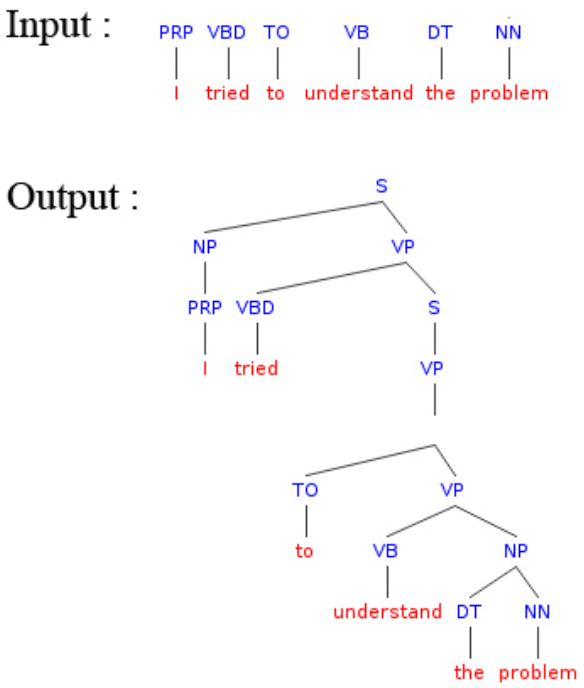

FIG. 3 – *Exemple de parsing. VP=Groupe verbal, NP=Groupe Nominal, etc.*

5.1 Schéma de prédiction par analogie

Comme expliqué au paragraphe 1.2, nous considérons une phrase P_0, dont la séquence S_0 des mots grammaticaux est connue et dont la structure syntaxique T_0 est recherchée. Soit AP un ensemble de phrases composées des paires de séquences et de structures connues. Le processus de prédiction de T_0 par analogie s'exprime alors ainsi :

1. Rechercher un triplet de phrases (P_1, P_2, P_3) de séquences (S_1, S_2, S_3) et de structures syntaxiques (T_1, T_2, T_3) tel que les structures S_0, S_1, S_2, S_3 définissent une proportion analogique exacte.
$$S_0 : S_1 :: S_2 : S_3$$
2. Notre hypothèse suppose que si les séquences sont en analogie, les structures le sont aussi. Donc nous pouvons prédire T_0 à partir de la résolution de l'équation analogique $x : T_1 : : T_2 : T_3$.

Si la première tâche, la recherche des triplets de phrases, est peu problématique sur le plan théorique, elle possède par contre une complexité pratique importante. Sans optimisation d'une sorte ou d'une autre, cette recherche dans une base de m phrases se fait au prix de l'ordre de m^3 combinaisons de trois séquences, chaque triplet (S_1, S_2, S_3) nécessitant le calcul de la dissemblance analogique entre S_0, S_1, S_2, S_3 qui est de complexité de l'ordre de $|T|^3$. Ce qui rend cette tâche en pratique non opérationnelle.

5.2 Protocole expérimental

Nous avons modifié la première étape du schéma précédent de façon à améliorer le processus de recherche du triplet de séquence en analogie avec la séquence cible S_0.

1. Nous compilons au préalable tous les calculs de prédiction faits sur tous les triplets de la base d'apprentissage, c'est à dire, pour chaque triplet S_i, S_j et S_k nous générons la quatrième séquences S_l par résolution d'équation analogique. Soit S_l la solution optimale pour ce triplet(DA la plus faible). Nous ne conservons que les triplets qui ont une DA nulle avec S_l.
2. Le problème est alors réduit à la comparaison entre S_0 et chaque S_l générée. Si S_0 est égale à S_l, nous prenons le triplet de phrases associé pour pouvoir compléter la deuxième étape de notre processus. Sinon, nous ne serons pas capables de terminer le processus de prédiction.

5.3 Evaluation des performances et résultats

Le corpus à notre disposition est composé de 316 phrases extrait de la base The Penn Wall Street Journal Treebank (Marcus et al. (1993)). C'est une taille relativement petite pour évaluer les capacités de notre méthode. Une évaluation classique nécessite la partition des données d'apprentissage en deux sous-ensembles : un ensemble d'apprentissage proprement dit et un ensemble de test.

Dans notre cas, nous avons choisi un ensemble d'apprentissage APP de 216 phrases et le reste pour l'ensemble de test TEST, soit 100 phrases, tirées au hasard.

D'abord, pour chaque séquence de l'ensemble TEST, nous identifions le triplet de séquences appropriée. Puis, nous passons à la génération de la structure syntaxique de chaque séquence de TEST. Enfin, nous comparons notre structure générée avec celle d'origine.

La Figure 4 présente nos premiers résultats provisoires obtenus lors des évaluations. A la lecture de ces résultats, 73% des structures syntaxiques sont correctement prédites ($DA = 0$). Les autres structures sont classées selon leurs degrés d'erreur de prédiction. Par exemple, 12% des structures ont eu un seul nœud mal prédit ($DA = 1$), 6% ont eu deux nœuds mal prédits et ainsi de suite.

6 Conclusion

Dans cet article, nous avons proposé un nouvel appariement entre arbres par proportion analogique pouvant s'énoncer : « l'arbre T_1 est à l'arbre T_2 comme l'arbre T_3 est à l'arbre T_4 ». Nous l'avons étendu au concept de dissemblance analogique, mesurant le coût d'appariement lorsque les arbres ne sont pas en analogie exacte. Quatre algorithmes ont été mis en place : les deux premiers pour mesurer la DA entre 4 séquences et entre 4 arbres, les deux derniers pour construire à partir de trois séquences ou de trois arbres le quatrième objet à DA minimale.

En ce qui concerne les applications utilisant ce type d'appariement et mise en œuvre de ces algorithmes, nous avons présenté un système de parsing automatique. Les résultats selon notre premier protocole d'évaluation semblent encourageants. Un autre protocole est envisagé. Il se base sur la technique de la validation croisée. Elle consiste à partitionner l'ensemble des

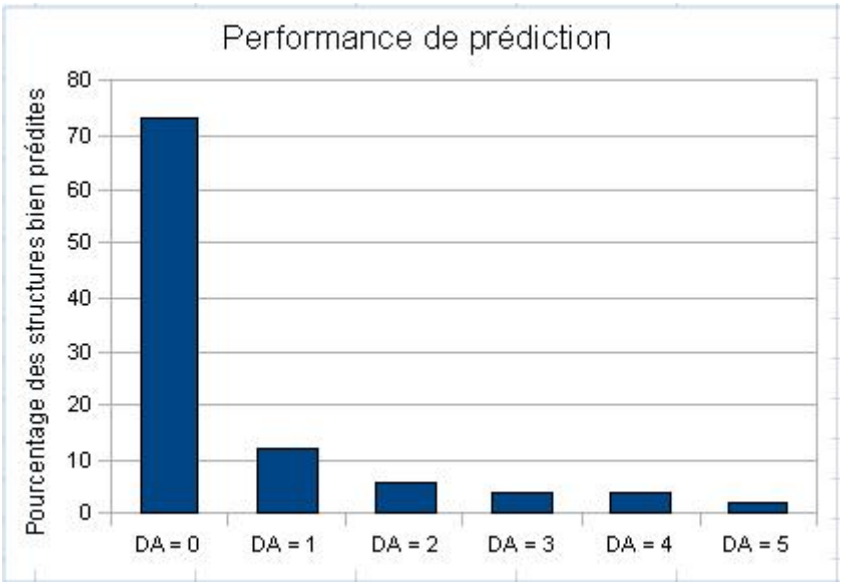

FIG. 4 – *Evaluation de notre système de prédiction.*

m données non plus en deux mais en Q sous-ensembles de taille m/Q, et à effectuer Q évaluations. Lors de chacune, l'ensemble d'apprentissage est constituée de $Q-1$ sous-ensembles et les tests sont effectués sur le dernier sous-ensemble. Les Q évaluations sont effectuées en faisant tourner le sous-ensemble de test, et le résultat final est donné par la moyenne des résultats des Q apprentissages. Cette approche est plus coûteuse en temps d'évaluation, mais elle fournit une évaluation de la prédiction avec une moins grande variance.

D'autres perspectives de recherche sont envisageables pour des travaux ultérieurs. Notre but est d'appliquer ces méthodes d'appariement par analogie à d'autres arbres, en particulier ceux qui codent les structures prosodiques des phrases, dans la synthèse de la parole. D'autres applications peuvent être envisagées dans le traitement automatique du langage naturel et en bio-informatique..

Références

Barbot, N. et L. Miclet (2009). La proportion analogique dans les groupes. applications à l'apprentissage et à la génération. In *Actes de la 11ème Conférence Francophone sur l'Apprentissage Automatique*, Hammamet, Tunisie, pp. 145–160.

Ben Hassena, A. et L. Miclet (2009). Dissimilarité analogique et apprentissage d'arbres. In *CAp'09 : 11ème Conférence d'apprentissage CAp 2009*, pp. 121–132.

Hofstadter, D. et the Fluid Analogies Research Group (1994). *Fluid Concepts and Creative Analogies*. New York : Basic Books.

Jiang, T., L. Wang, et K. Zhang (1994). Alignment of trees - an alternative to tree edit. In *CPM '94 : Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, London, UK, pp. 75–86. Springer-Verlag.

Lepage, Y. (2003). *De l'analogie rendant compte de la commutation en linguistique*. Grenoble. Habilitation à diriger les recherches.

Marcus, M. P., M. A. Marcinkiewicz, et B. Santorini (1993). Building a large annotated corpus of english : the penn treebank. *Comput. Linguist. 19*(2), 313–330.

Miclet, L., S. Bayoudh, et A. Delhay (2008). Analogical dissimilarity : Definition, algorithms and two experiments in machine learning. *Journal of Artificial Intelligence Research 32*, 793–824.

Miclet, L. et H. Prade (2009). Handling analogical proportions in classical logic and fuzzy logics settings. In *Lecture Notes in Computer Science 5590 Springer 2009. Proceedings of the 10th European Conference ECSQARU 2009, Verona, Italy, July 1-3, 2009*, pp. 638–650. Springer-Verlag.

Stroppa, N. et F. Yvon (2004). Analogie dans les séquences : un solveur à états finis. In *TALN 2004*.

Stroppa, N. et F. Yvon (2005). Analogical learning and formal proportions : Definitions and methodological issues. Technical Report ENST-2005-D004, École Nationale Supérieure des Télécommunications.

Summary

In Artificial Intelligence, analogy is used as a non exact reasoning technique to solve problems, for natural language processing, for learning classification rules, etc.

We focus in this paper on the matching by analogical proportion of sequences and trees. The analogical proportion a relation between four objects that expresses that the way to transform the first object into the second is the same as the way to transform the third in the fourth. Let call the objects O_1, O_2, O_3 and O_4 ; their relationship of analogical proportion is generally formulated by: "O_1 is to O_2 as O_3 is to O_4" and is denoted $O_1 : O_2 :: O_3 : O_4$. In this article, we firstly give algorithms to compute the analogical proportion between four sequences, or four trees. Secondly, we present an application to the learning of the syntactic parsing of a sentence. As a result, we have measured the performance of our parser on a corpus extracted from the Penn Wall Street Journal Treebank. Preliminaries results on a test set give an exact or almost exact restitution of the parsing tree from the sequence in 85 % of cases.

Logical, graph based knowledge representation with CoGui

Jean Francois Baget*, Michel Chein*, Madalina Croitoru*, Alain Gutierrez*, Michel Leclere*, Marie-Laure Mugnier*

*LIRMM (CNRS and University Montpellier II, France)
«lastname»@lirmm.fr

Abstract. This paper reports on the ongoing effort in building an RDF ontology for the de-facto standard conceptual model for library catalogs. We motivate our work by a concrete real world application and demonstrate how using the CoGui Conceptual Graphs ontology editor will highly benefit the task.

1 Introduction

RDF [1](Resource Description Framework) is a language standardized by W3C (World Wide Web Consortium) dedicated to the representation of knowledge on the Web. Its popularity made it an important backbone of knowledge base related applications. A querying language for RDF has also been proposed (SPARQL) which allows to uniformly access different RDF data sources. To facilitate cross-questioning of such repositories, many class hierarchies (ontologies) have been proposed as a standard allowing the modeling of different areas: see for example FOAF dedicated to description of persons and relationships they have among themselves, or BIBO dedicated to bibliographic information etc. The use of ontologies standards ensures that we can link "in house" knowledge bases to many other existing databases via the Web. Moreover, extracting the RDF knowledge from a data repository is an indispensable tool for ensuring the quality of the database at hand.

To conclude, by the use of RDF modelling we provide (1) a standardized syntax for interoperability purposes and (2) a consistent vocabulary.

This paper reports on the ongoing effort in building an RDF ontology for the de-facto standard conceptual model for library catalogs. We motivate our work by a concrete real world application and demonstrate how using the CoGui Conceptual Graphs ontology editor highly benefits the task.

The paper is organised as follows: Section 2 details the motivation of our work in great detail and shows how using CoGui is advancing the state of the art. Section 3 puts down the theoretical foundations behind the visual representation of CoGui: Conceptual Graphs. Finally Section 4 presents current and future work within this project.

1. http://www.w3.org/

2 Motivating example

2.1 Scenario Description

Our motivating example is a real case scenario of representing library catalogue data. This work is being carried out in the context of a collaboration with ABES within the TGE-ADONIS [2] project. ABES [3] is a French public institution managing library catalogue records created in 1994.

Let us begin by briefing on the work of ABES before our collaboration started. ABES has designed and implemented SUDOC (the university documentation system), Calames (an on-line catalogue of higher education archives and manuscripts) and Star (a tool for posting and permanently archiving theses). Since this paper is mainly concerned with SUDOC, from here onwards we will only focus on this system. SUDOC encompasses more than 1100 public or private libraries, and its knowledge base consists of more than 8 million bibliographical data referencing more than 25 million documents. The main aim of SUDOC is to design a coherent set of referentials identifying people, collectivities, works, domains etc. for libraries records.

SUDOC has been initially encoded in UNIMARC [4]: a standard for assigning labels to catalogue records. A snapshot of a UNIMARC file is shown in Figure 1. All the constructs needed for interoperability (either with other libraries, or simply ensuring coherence of the catalogue records amongst each other) are represented using well defined functional blocks identified by three-character numeric tags. These blocks organise the data according to its function in a traditional catalogue record. Unfortunatelly, the semantics of these blocks is not made explicit in the language itself. Obviously, this will impose certain limitations when either (1) checking for consistency (e.g. identifying multiple entries of the same entity) or (2) for search optimisation. To this end a translation of UNIMARC in XML [5] was performed.

However, at this point - SUDOC records expressed in XML - the problem of semantics for the above mentioned problems was far from being solved. Moreover, an opening towards data available on the Web (that could benefit the system) was not at all possible. The decision was then taken to create an ontology for SUDOC. The language of choice was RDF(S) [6] given its standard status and the expressivity of the records. This is the moment when the collaboration with our group begun.

The motivation of our paper is modelling RDF(S) ontologies in a practical setting. To this end we will present a visual tool for representing and reasoning with ontologies: CoGui. We will fully detail CoGui further on in the paper. For the remainder of this section we will put our work in context and present the representational needs of the application scenario.

The question of the ABES joint project was how to design an RDF(S) ontology for SUDOC. Obviously, the design of such ontology has to be done in close collaboration with the ABES domain experts and needs to contain all the information already present in SUDOC. Basically, from a representation view point, the envisaged SUDOC ontology had to:

– Model already existing catalogue records based on the information already available in the XML file

2. http://www.tge-adonis.fr/
3. http://www.abes.fr/abes/index.html
4. http://www.ifla.org/en/unimarc
5. http://www.w3.org/XML/
6. http://www.w3.org/RDF/

```
001 0192122622@
010##$a0-19-212262-2$d£12.95@
020##$aUS$b59-12784@
020##$aGB$bb5920618@
100##$a19590202d1959####|||y0engy0103####ba@
1011#$aeng$cfre@
102##$aGB$ben@
105##$aac######000ay@
2001#$a{NSB}The {NSE}lost domain$fAlain-Fournier$gtranslated from the French by Frank
Davison$gafterword by John Fowles$gillustrated by Ian Beck@
210##$aOxford$cOxford University Press$d1959@
215##$aix,298p,10 leaves of plates$cill, col.port$d23cm@
311##$aTranslation of: Le Grand Meaulnes. Paris : Emile-Paul, 1913@
454#1$1001db140203$150010$a{NSB}Le {NSE}Grand Meaulnes$1700#0$aAlain-Fournier$f1886- 1914$1210##$aParis$cEmile-Paul$d1913@
50010$a{NSB}Le {NSE}Grand Meaulnes$mEnglish@
606##$aFrench fiction$2lc@
676##$a843/.912$v19@
680##$aPQ2611.O85@
700#0$aAlain-Fournier,$f1886-1914@
702#1$aDavison,$bFrank@
801#0$aUK$bWE/N0A$c19590202$gAACR2@
98700$aNov.1959/209@
```

FIG. 1 – *Example of a UNIMARC representation of the book "Le Grand Meaulnes" by Alain-Fournier*

– Ensure interoperability with data (e.g. other catalogue records, existing ontologies etc.) that could potentially be of interest in SUDOC in the course of its future development

A first direction of work was to take the SUDOC document expressed in XML and, based on a XSLT, to create the "corresponding" RDF(S) file. Of course, the output file has to be further enriched with a lot of information based on the implicit semantics of the SUDOC-XML but not explicitly expressed due to the limitations in the choice of language. Naturally, making explicit all the implicit rules in the XML file boils down to understanding the basis of the interchange formats of library catalogs.

It is at this point that we decided to study in detail the FRBR model. The FRBR model [7] is the de-facto standard conceptual model developed by an IFLA (International Federation of Library Associations and Institutions) group of experts. The model standardizes the functions a library catalogue should perform and lays the foundations for innovative catalogs that will use state-of-the-art techniques. A 142 pages document available online [8] makes explicit all the constructs that should be present when modeling a catalogue library record. Of course this document will subsume the information already present in SUDOC. The decision was then taken to use FRBR as the basis for the SUDOC ontology.

2.2 Limitations of existing work

Now that we set the context of the considered application scenario, let us look at existing work and how it compares to our approach. As mentioned in the previous section the problem at hand was how to represent the FRBR document in RDF(S).

7. http://www.bnf.fr/pages/zNavigat/frame/version_anglaise.htm?ancre=normes/no-acFRBR_gb.htm
8. http://archive.ifla.org/VII/s13/frbr/frbr_current_toc.htm

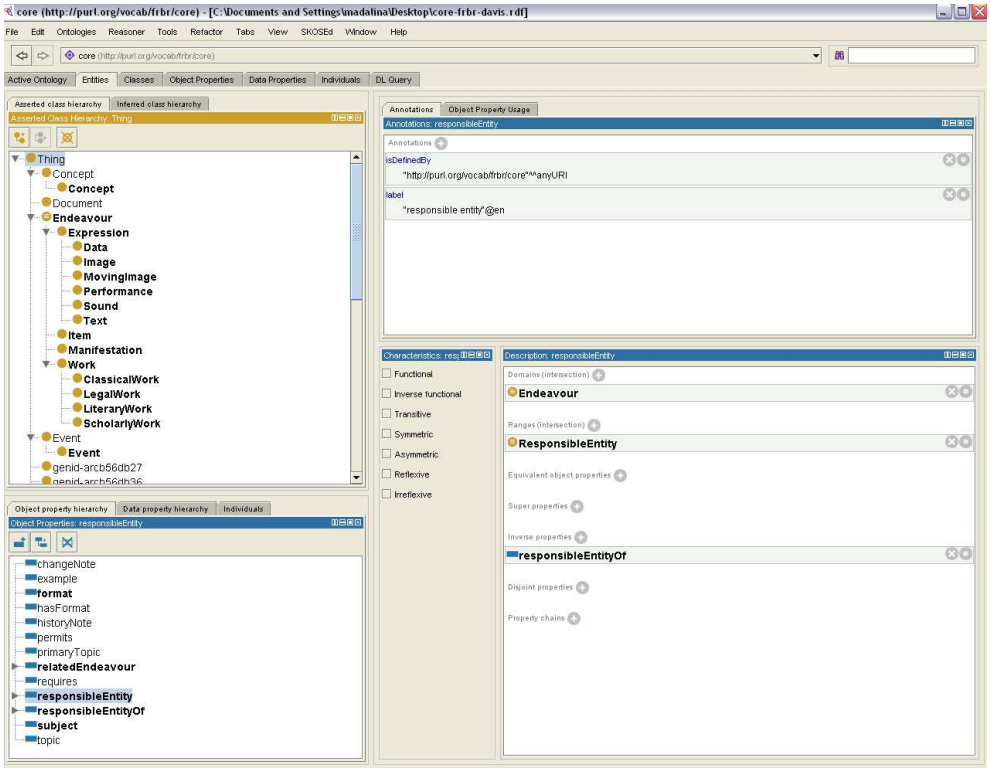

FIG. 2 – *Visualisation of* `http://vocab.org/frbr/core.rdf` *in Protégé*

The literature search revealed an existing RDF(S) file expressing FRBR. This file is publicly available on the web [9] and has been created by Ian Davis and Richard Newman. As no scientific paper on its creation exists (to the knowledge extend of the authors) the document had to be examined by simply opening it with an RDF(S) ontology editor and trying to understand the modelling choices behind it.

The de-facto standard for ontology editors is Protégé [10]. Of course, other tools for visualising RDF(S) exist [11] but their purpose is at most visualising RDF(S) (sometimes using hierarchical graphs, clustering techniques etc.) and not ontology (1) building and (2) reasoning using RDF(S).

Figure 2 shows a screen-shot of the above mentioned FRBR file in Protégé. Please note that there are no suitable visualisations for Protégé 4 which render information more intuitive (OWL Viz [12] is incompatible, Jambalaya [13] is not suitable for knowledge modelling etc.).

9. `http://vocab.org/frbr/core.rdf`
10. `http://protege.stanford.edu/`
11. `http://planetrdf.com/guide/`
12. `http://protegewiki.stanford.edu/index.php/OntoViz`
13. `http://protegewiki.stanford.edu/index.php/Jambalaya`

Unfortunately, as pointed out by the domain experts, in the context of our application the visualisation in Figure 2 has a number of drawbacks:

– Difficulty to visualise / explore concept / relation hierarchies
– Impossibility to represent any poset (and not only trees)
– No visual distinction between the model and the instances
– Impossibility to visualise relevant knowledge about instances

In this paper we present a tool for visually representing knowledge: CoGui [14]. CoGui is a Conceptual Graphs editor compatible with RDF(S). Conceptual Graphs are a logical graph-based knowledge representation language equivalent to the positive existential fragment of first order logic further detailed in Section 3. In the reminder of this section we will focus on how using CoGui will overcome all of the above mentioned drawbacks of Protégé.

2.3 Approach

Let us begin by presenting at a glance how the RDF(S) file available at `http://vocab.org/frbr/core.rdf` will be visualised and manipulated when being opened in CoGui. Figure 3 presents the concept type hierarchy, while Figure 4 presents the relation type hierarchy. Finally Figure 5 presents a fact in the ontology.

An ontology is composed by background knowledge and factual knowledge. In the context of this example we consider the background knowledge composed only by a hierarchy of concept types and a hierarchy of relation types with arity greater or equal to 1. For the general case of Conceptual Graphs (and not just Conceptual Graphs rendering of RDF(S)) please see Section 3.

In Figure 3 and 4 the concept / relation type hierarchies are displayed on the left hand side of the screen using a type list view and on the right hand side rendered graphically (types are displayed as vertices). Please note that type items may appear more than once hence allowing for the edition of posets. Types can be dragged from type views to graph views and vice-versa.

The edition of the concept hierarchy is being assisted by the control button (top menu, center) which allows to detect poset circuits or redundant edges in the concept types. As for relation types they have associated a signature (an ordered list of concept types). In this context the control button will forbid circuits and ensure that the signatures compatibility is respected.

All of the above demonstrates how CoGui syntactically and semantically overcomes the two first drawbacks of Protégé: visualising / exploring concept / relation hierarchies and representing posets in general.

In the main project window the background knowledge (consisting of concept type hierarchy, relation type hierarchy etc.) is presented alongside with the factual knowledge. Each fact is visualised in the editor graphically. Let us consider Figure 5 depicting a labeled bipartite graph where one class of nodes is the concept nodes and the other the relation nodes. A concept node is labeled by a concept type (e.g. *cc:Work*) and, possibly, by an individual (e.g. *file:///var/www/vocab.org/www/htdocs/frbr*). Incident edges nodes have to have compatible signatures with the concept nodes. Similarly, the control button will highlight any incompatibility. It is clear how CoGui allows for the representation of relevant knowledge about instances hence addressing the last two drawbacks mentioned above of Protégé.

14. `http://www.lirmm.fr/cogui/`

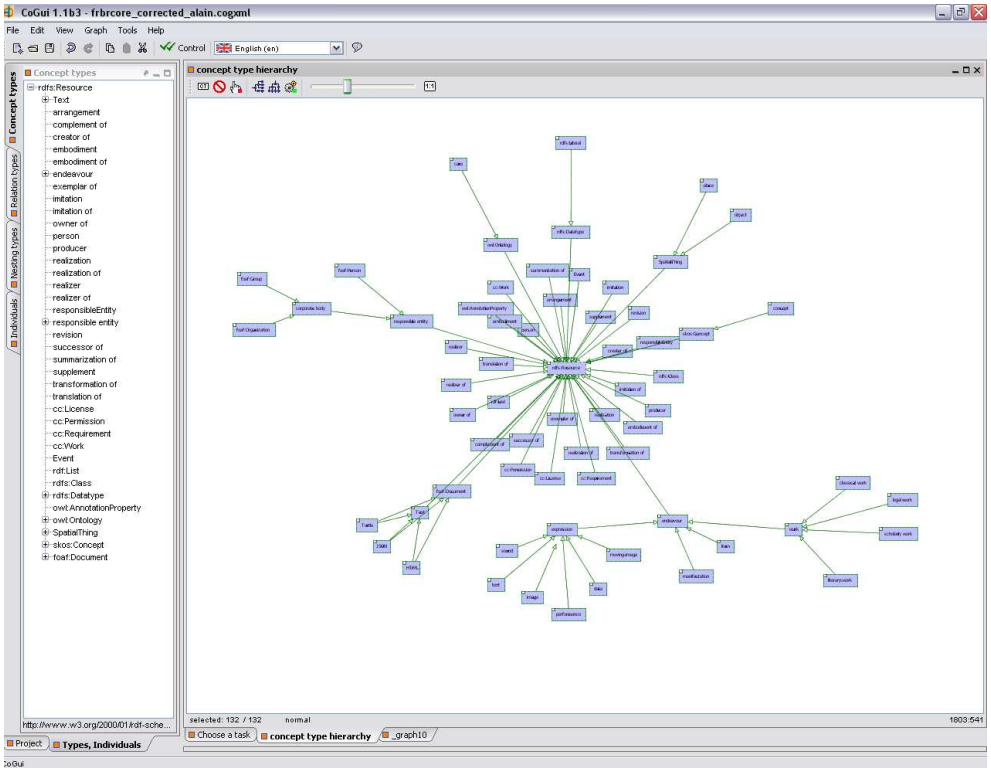

FIG. 3 – *Visualisation of concept types in* `http://vocab.org/frbr/core.rdf` *using CoGui*

2.4 Discussion

Please note that CoGui is not just a simple edition tool for ontologies but it also allows for reasoning (querying). This is done by calling the COGITANT library [15]. The approach of the querying mechanism is to search for homomorphisms between the query graph and the "knowledge base". The founded pieces of graphs (called projections) are specializations (in the logical sense) of the query graph. More details on homomorphism and Conceptual Graphs are given in Section 3.

It is also important to note here that CoGui offers additional constructs to assist with the edition of ontologies as well as with reasoning. An important number of expressive constructs is provided to the user, namely:

– **Rules**: used to represent implicit (common sense) knowledge. For instance, let us assume that "Eve is the mother of Abel" is a fact graph. If the ontology contains a rule saying that if x is the mother of y then y is a child of x then the system can automatically add the information that Abel is a child of Eve.

15. `http://cogitant.sourceforge.net/`

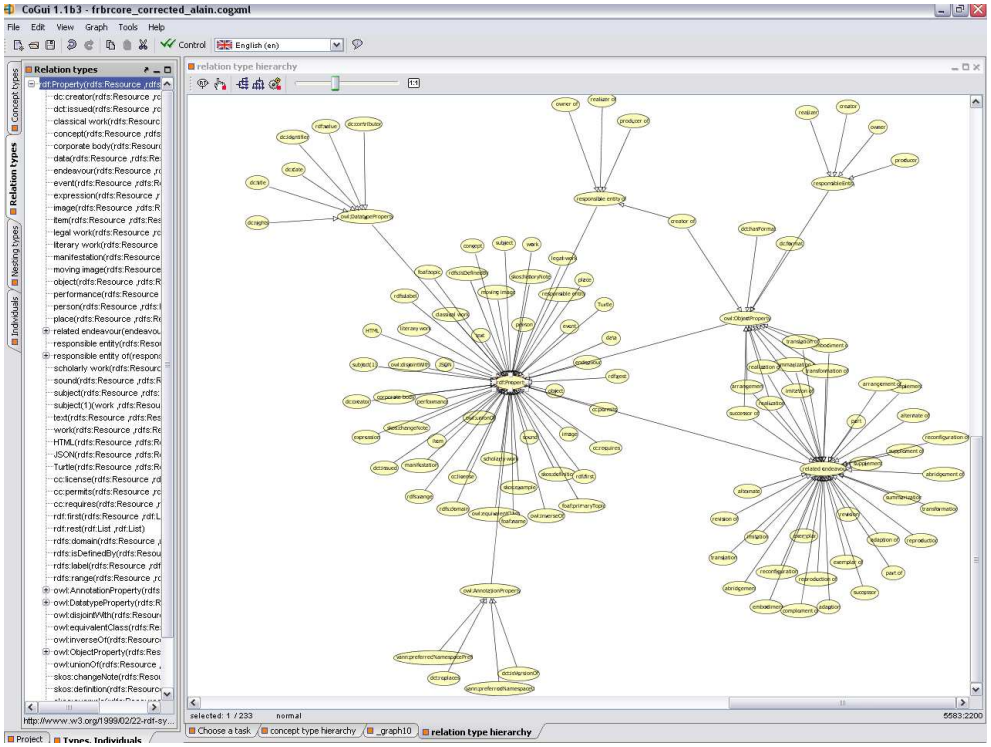

FIG. 4 – *Visualisation of relation types in* `http://vocab.org/frbr/core.rdf` *using CoGui*

- **Nesting types**: used to facilitate the construction of (well-structured) nested graphs (graphs detailing a certain concept). For instance, in RDF(S) this will correspond to easily editing *rdf:Statement*.
- **Individual graphs**: a simple conceptual graph having a special concept node (the head) which represents factual knowledge about an individual.
- **Pattern graphs**: usually used in an annotation scenario, representing a starting point when describing a document with respect to the chosen annotation type.

All of the above functionalities are fully described in the online [16] manual of CoGui.

3 Conceptual Graphs

The CoGui editor described above is a free graph-based visual tool, developed in Java, for building Conceptual Graph knowledge bases. CoGui allows the import / export of RDF(S) files.

16. `http://www.lirmm.fr/cogui/userguide.php`

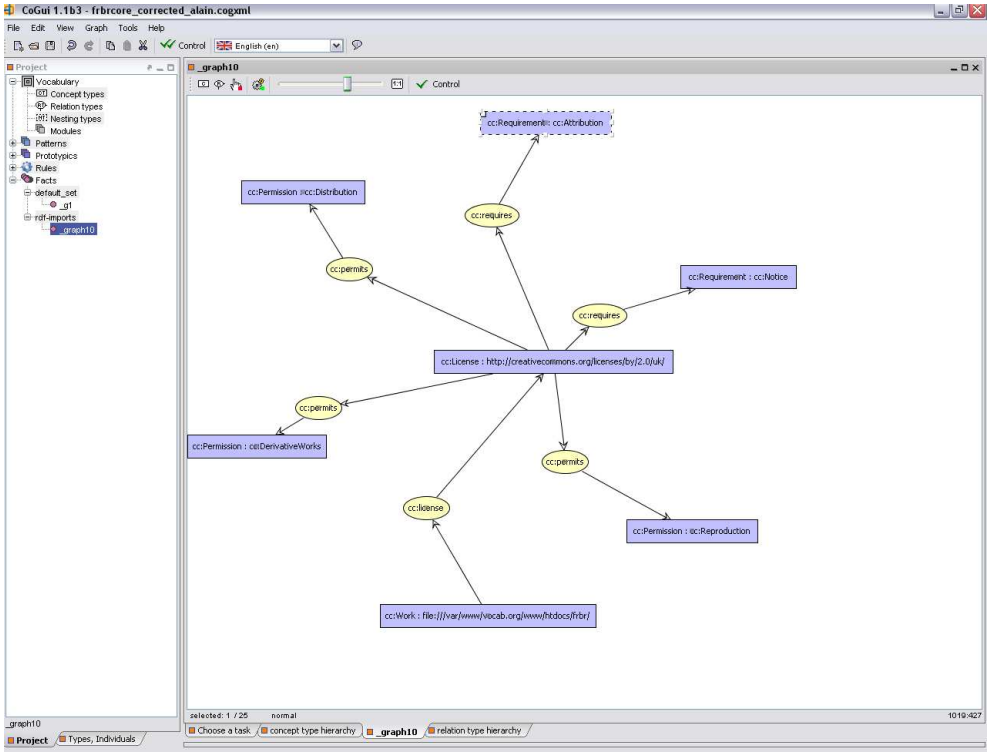

FIG. 5 – *Visualisation of individuals in* `http://vocab.org/frbr/core.rdf` *using CoGui*

In this section we will describe Conceptual Graphs, hence laying the theoretical foundations of the applicative work described in the Section 2. It is important to mention at this point that our choice of Conceptual Graphs is not only based upon visualisation and expressivity, but also on the potential for optimisation induced by using graphs for reasoning (see Chein and Mugnier (2009) for an in depth analysis). This is an important aspect to be considered in the context of this application and further work in this direction is detailed in Section 4.

Conceptual Graphs were introduced by Sowa (cf. Sowa (1976, 1984)) as a diagrammatic system of logic with the purpose "to express meaning in a form that is logically precise, humanly readable, and computationally tractable". In this paper we use the term "Conceptual Graphs" to denote the *family of formalisms* rooted in Sowa's work and then enriched and further developed with a graph-based approach in Chein and Mugnier (2009).

Conceptual Graphs encoded knowledge as graphs and thus can be visualized in a natural way:

– The vocabulary, which can be seen as a basic ontology, is composed of hierarchies of concepts and relations. These hierarchies can be visualized by their Hasse diagram, the usual way of drawing a partial order.
– All other kinds of knowledge are based on the representation of entities and their rela-

tionships. This representation is encoded by a labeled graph, with two kinds of nodes, respectively corresponding to entities and relations. Edges link an entity node to a relation node. These nodes are labeled by elements of the vocabulary.

The **vocabulary** is composed of two partially ordered sets: a set of concepts and a set of relations of any arity (the arity is the number of arguments of the relation). The partial order represents a specialization relation: $t' \leq t$ is read as "t' is a specialization of t". If t and t' are concepts, $t' \leq t$ means that "every instance of the concept t' is also an instance of the concept t". If t and t' are relations, then these relations have the same arity, say k, and $t' \leq t$ means that "if t' holds between k entities, then t also holds between these k entities").

A **basic graph** (BG) is a bipartite graph: one class of nodes, called *concept* nodes, represents entities and the other, called *relation* nodes represents relationships between these entities or properties of them. A concept node is labeled by a couple $t : m$ where t is a concept (and more generally, a list of concepts) and m is called the marker of this node: this marker is either the generic marker, denoted by $*$, if the node refers to an unspecified entity, otherwise this marker is a specific individual name. BGs are used to represent assertions called *facts*. They are also building blocks for more complex kinds of knowledge (such as rules, or nested graphs). In this paper we only detail rules as they are of direct interest to the framework we are proposing.

A **rule** expresses implicit knowledge of form "if *hypothesis* then *conclusion*", where hypothesis and conclusion are both basic graphs. Using such a rule consists of adding the conclusion graph (to some fact) when the hypothesis graph is present (in this fact). There is a one to one correspondence between some concept nodes of the hypothesis with concept nodes of the conclusion. Two nodes in correspondence refer to the same entity. These nodes are said to be *connection nodes*. The knowledge encoded in rules can be made explicit by applying the rules to specific facts.

These graphical objects are provided with a **semantics in first-order-logic**, defined by a mapping classically denoted by Φ in conceptual graphs (see Sowa (1984)). First, a FOL language corresponding to the elements of a vocabulary $\mathcal{V}$ is defined: concepts are translated into unary predicates and relations of arity k into predicates of arity k. Individual names become constants. Then, a set of formulas $\Phi(\mathcal{V})$ is assigned to the vocabulary. These formulas translate the partial orders on concepts and relations: if t and t' are concepts, with $t' < t$, one has the formula $\forall x(t'(x) \rightarrow t(x))$; similarly, if r and r' are k-ary relations, with $r' < r$, one has the formula $\forall x_1 \ldots x_k(r'(x_1 \ldots x_k) \rightarrow r(x_1 \ldots x_k))$. A fact G is naturally translated into a positive, conjunctive and existentially closed formula $\Phi(G)$, with each concept node being translated into a variable or a constant: a new variable if it is a generic node, and otherwise the constant assigned to its individual marker. The logical formula assigned to a rule R is of form $\Phi(R) = \forall x_1 \ldots x_p\ ((hyp) \rightarrow \exists y_1 \ldots y_q\ (conc))$, where: hyp et $conc$ are conjunctions of atoms respectively translating the hypothesis and the conclusion, with the same variable being assigned to corresponding connection nodes; $x_1 \ldots x_p$ are the variables assigned to the concept nodes of the hypothesis; $y_1 \ldots y_q$ are the variables assigned to the concept nodes of the conclusion except for the connection nodes.

More importantly, first order logic subsumption can be translated in a graphical operation: homomorphism. A homomorphism from G to H is a mapping between the node sets of G to the node sets of H, preserving the adjacency between nodes of G and decreasing the node labels. If there is a homomorphism (say π) from G to H, we say that G *maps* to H (by π).

4 Current and future work

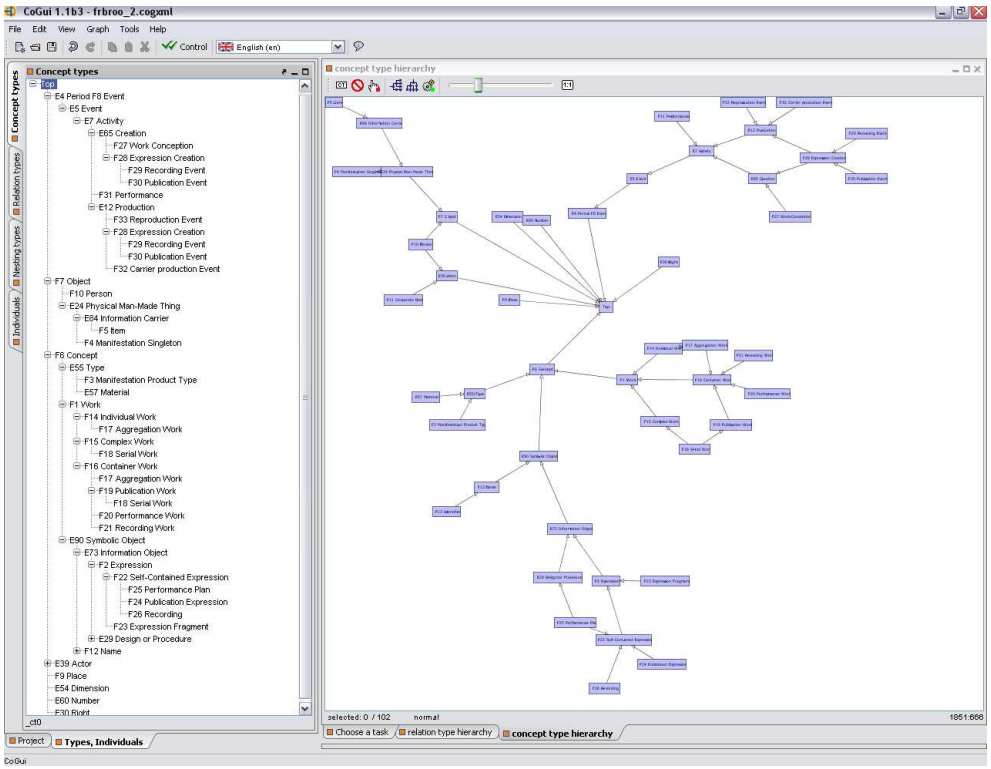

FIG. 6 – *Visualisation of the current state of the proposed RDF FRBR ontology using CoGui*

Let us now get back to the example presented in Section 2. We have shown our need for an FRBR RDF ontology and why we chose CoGui as the ontology editor for this task. Moreover, CoGui is now the de-facto tool used for our modelling collaboration with the ABES domain experts (which are not trained computing scientists).

Let us now further analyse the RDF(S) file presented in `http://vocab.org/frbr/core.rdf`. Following discussions with ABES it became obvious that this existing RDF file is not suitable for the application at hand. This is due to:

- The lack of important structures present in FRBR indispensable for incorporating innovative catalogs. Moreover, the missing structures cannot be easily integrated in the existing ontology given the different level of granularity chosen by Ian Davis.
- The lack of semantic relations between certain essential concepts
- The semantic ambiguity and overlapping of certain ontological constructs

While the first two items are self explanatory, we will detail the third. The RDF ontology at `http://vocab.org/frbr/core.rdf` aims at interoperability within the LinkedData [17]

17. `http://linkeddata.org/`

cloud by employing different but semantically overlapping constructs from different ontologies already in the cloud (for instance foaf:Author [18] and dc:Author [19]). It is not clear how these constructs will be directly used for reasoning in the ontology.

In the light of the above mentioned reasons we have thus decided to use the 142 pages FRBR reference document as a common base between the experts and our group and to build, using CoGui, the complete [20] FRBR RDF ontology. Figure 6 shows this endeavor at the current date.

We plan to extend this work in a number of ways. First, an obvious extension will be the finalisation of the ontology. To this end we are investigating how using Conceptual Graphs rules (formally defined in Section 3) can speed up the ontology building process. Second we are interested in how to achieve interoperability within the LinkedData cloud without the above mentioned drawbacks of semantic ambiguity. Third we are interested in different important theoretical problems arising from this applicative scenario. More precisely we are investigating not only how to manipulate, but also store, in an intelligent way, the large number of graphs this application will output.

References

Chein, M. and M. Mugnier (2009). *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer.

Sowa, J. F. (1976). Conceptual Graphs. *IBM Journal of Research and Development 20*(4), 336–375.

Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley.

Résumé

Cet article décrit le travail de modélisation d'une ontologie RDF pour des notices bibliographiques. Nous motivons notre approche par une application concrète du monde réel et démontrons comment l'utilisation d'un éditeur visuel à base de graphes améliorera cette tâche.

18. From the FOAF ontology: `http://www.foaf-project.org/`
19. From the DUBLIN CORE ontology: `http://dublincore.org/`
20. Regarding the needs of the SUDOC

Utilisation de la distance d'édition pour l'appariement sémantique de documents XML

Mai Dong Lee*, Karen Pinel-Sauvagnat**

* ledongbk@yahoo.com
**IRIT-SIG
118 route de Narbonne
31 062 Toulouse Cedex 4
sauvagnat@irit.fr

Résumé. A notre connaissance, peu de travaux utilisent directement la théorie des graphes pour la recherche d'information structurée ad-hoc, c'est à dire pour calculer l'appariement approximatif entre une requête structurée et un document de type XML. Dans cet article, nous proposons une adaptation de l'algorithme de Tai (1979) pour le calcul de la distance d'édition entre l'arbre des documents et l'arbre de la requête. Cet algorithme est combiné à une recherche sur le contenu des documents.

1 Introduction

L'accès aux documents structurés de type XML (*eXtensible Markup Language*) soulève de nouvelles problématiques liées à la co-existence de l'information structurelle et de l'information de contenu. Les approches de l'état de l'art en Recherche d'Information Structurée considèrent que la dimension structurelle permet de mieux répondre aux différentes attentes des utilisateurs, en se focalisant sur leur besoin. Les méthodes proposées s'appliquent à renvoyer des parties de documents répondant de manière *spécifique* et *exhaustive* au besoin en information de l'utilisateur, que celui-ci soit formulé sous forme de requête structurée ou non. On trouvera dans Pinel-Sauvagnat et Boughanem (2006), Manning et al. (2008), Fuhr et al. (2008) et Geva et al. (2009) un état de l'art de la recherche d'information structurée actuelle.

De nombreuses approches utilisent la représentation arborescente des documents XML pour la recherche d'une information pertinente. Cependant, à notre connaissance, la théorie des arbres (et plus largement la théorie des graphes) n'est pas ou peu utilisée de façon explicite pour la recherche. Cette théorie est pourtant très puissante et pourrait donner des indications précieuses sur les algorithmes de recherche les plus intéressants à étudier.

Nous proposons dans cet article d'appliquer l'algorithme de Tai (1979) pour calculer la similarité de structure entre la requête et les documents. Cet algorithme, bien qu'assez simple, sert de base à la majorité des autres algorithmes d'appariement, et pourra donc servir de socle à de futures propositions. La mesure de similarité que nous proposons est la combinaison de cette similarité de structure avec une similarité de contenu, évaluée à l'aide d'un moteur de recherche plein texte classique.

Le reste de l'article est organisé comme suit: dans un premier temps (section 2), nous présentons ce qu'est l'appariement de graphes et d'arbres, ainsi que quelques applications à la recherche d'information structurée. Dans la section 3, nous proposons notre adaptation de l'algorithme de Tai (1979) à la recherche d'information structurée. La section 4 illustre cette adaptation sur un exemple.

2 Etat de l'art

2.1 Graphes et arbres

Les graphes sont une structure de données utile pour la représentation d'objets. Typiquement, des parties d'un objet complexe sont représentées par des *nœuds* (ou des sommets), et relations entre ces parties par des *arêtes*. Des *labels* (ou étiquettes) et attributs pour les nœuds et des arêtes sont employés pour incorporer les informations supplémentaires dans une représentation de graphes.

Si des graphes sont employés pour la représentation d'objets, trouver des objets similaires à d'autres revient à déterminer la similitude entre les graphes : on appelle cela *l'appariement de graphes*. Il y a deux types d'appariement de graphes (Bunke (2000)): l'appariement exact et l'appariement approximatif. L'appariement exact de graphes n'est pas très utile dans la recherche d'information, où on ne cherche pas à répondre de façon exacte au besoin de l'utilisateur. Nous nous intéressons plus à l'appariement approximatif, pour lequel les labels et le contenu d'un nœud peuvent jouer un rôle important.

L'appariement approximatif de graphes est une méthode basée sur le calcul de mesure de similarité de deux graphes. Il y a deux types principaux d'appariement approximatif : (1) l'*appariement basé sur le sous-graphe commun maximal* (Hidovi D., Pelillo M. (2004), Wallis et al. (2001), Bunke et Shearer (1998)) et (2) l'*appariement basé sur la distance d'édition* (Bunke (1998) (2000)).

1. Dans le premier cas, le sous-graphe commun maximal G de deux graphes G_1 et G_2 est un sous-graphe de G_1 et de G_2 ayant le nombre de nœuds maximal parmi tous ces sous-graphes. Plus deux graphes sont similaires, plus leur sous-graphe maximal est grand.
2. L'appariement basé sur la distance d'édition est une extension de la distance de chaînes de caractères dans le domaine des graphes. La distance d'édition de deux graphes G_1 et G_2 est définie comme le nombre minimum d'opérations nécessaires pour transformer G_1 en G_2. Plus cette distance est petite, plus deux graphes sont similaires: cette distance convient donc pour mesurer la similarité de deux graphes.

Les graphes généraux ont cependant des structures et propriétés complexes, qui sont difficiles à utiliser dans la recherche d'information structurée. Dans notre cas, les documents XML, comme de nombreux autres objets, peuvent être représentés par des arbres, qui sont un cas particulier de graphes n'ayant aucun cycle.

Dans des applications basées sur des arbres, la similarité entre ces arbres est importante. Il y a trois types d'appariements principaux d'arbres : la *distance d'édition* (Tai (1979), Zhang (1996), Demaine et al. (2007), Sasha et Zhang (1997), Klein (1998)), la *distance d'alignement* (Jiang et al. (1995)) et l'*inclusion d'arbres* (Chen (1998)). On trouvera dans Lee (2009) le détail de ces algorithmes pour l'appariement. Nous proposons dans la section suivante quelques exemples d'applications de ces algorithmes à la recherche d'information structurée.

2.2 Applications à la recherche d'information structurée

Abdeslame Alilaouar (2007) utilise un type d'appariement basé sur la distance d'édition des arbres, pour l'interrogation flexible de données semi-structurées en général et des données XML en particulier. Les travaux redéfinissent un cadre permettant de déterminer jusqu'à quel niveau une réponse est pertinente ou non vis-à-vis d'une requête. Ce cadre est appelé arbre de recouvrement. Il est issu du rapprochement entre l'approche par relaxation et l'approche basée sur la distance d'édition. La relaxation concerne les descendants, lorsqu'un critère de descendance directe relaxe vers un critère de descendance indirect. L'arbre de recouvrement peut être également vu comme un cas particulier de la distance d'édition si on n'autorise qu'une seule opération élémentaire d'édition : l'ajout de nœud. Le cadre de la théorie des sous-ensembles flous a été utilisé dans deux cas : (i) d'abord pour représenter les préférences des utilisateurs, c'est-à-dire que les critères d'interrogation sont vus comme des préférences pondérées et non des contraintes strictes, (ii) ensuite pour évaluer le degré de correspondance des données avec un critère en utilisant la fonction d'appartenance. Pour mettre en œuvre et valider cette approche, l'auteur a développé et expérimenté un algorithme d'interrogation basé sur le principe de l'arbre de recouvrement. Cet algorithme construit les sous-arbres recouvrant de l'arbre de requête progressivement en parcourant l'arbre de cette dernière en post-ordre. Cet algorithme se compose de deux modules : le premier consiste à chercher pour un nœud tous les nœuds qui sont suffisamment similaires, le second concerne la validation de la structure.

Dans de Rougemont (2005) et de Rougemont et Vieilleribière (2008), on trouve une approche pour vérifier qu'un fichier XML est conforme à une DTD. Les fichiers XML sont représentés par des arbres T, et les DTD par un langage L ou un automate (et un automate est représenté par un arbre). Les auteurs utilisent ensuite des méthodes basées sur des changements des états d'automate pour estimer la distance d'édition.

Ismael Sanz et al. (2005) utilisent l'inclusion pour trouver des motifs pertinents dans la recherche de documents XML au sein d'une collection hétérogène. Ils proposent l'inclusion approximative entre les sous-arbres d'arbre de motif et l'arbre de la collection hétérogène des documents XML. L'arbre de la collection hétérogène est créé par l'ajout

d'une racine appelé db, et toutes les racines des arbres de documents deviennent des enfants de la racine db .L'inclusion approximative (par l'indexation et la similarité entre les labels) produit des sous-arbres de collection qui sont appelés des fragments. Les auteurs proposent certains critères pour l'union des fragments du même document pour créer des régions, et assigner la similarité pour ces régions. Ces régions sont considérées comme les résultats de requête d'utilisateur, et elles sont ordonnées par la similarité.

Theodore Dalamagas et al. (2004)(2006) ont utilisé une mesure de similarité basée sur la distance d'édition pour le clustering des documents XML. Ils considèrent le clustering des documents XML par structure comme le clusturing des arbres de labels. Ils calculent tout d'abord la distance d'édition, qu'ils évaluent avec des algorithmes basés sur la programmation dynamique (algorithmes de Tai (1979), Zhang (1996), …). Pour augmenter la vitesse de calcul de la distance d'édition, ils proposent des méthodes de réduction de structures, comme par exemple la réduction de nœuds de sous-arbre répété, la réduction de nœuds nichés,.., .Ces réductions sont exécutées par un parcours pré-ordre d'arbre. Deuxièmement, ils proposent la similarité de structure de deux documents XML D_1, D_2 qui sont représentés par deux arbres de labels T_1, T_2 (respectivement) :

$S(T_1,T_2) = d(T_1,T_2) / d'(T_1,T_2)$

Où : $d(T_1,T_2)$ est la distance d'édition de deux arbres T_1,et T_2

$d'(T_1,T_2)$ est la distance d'édition de deux arbres T_1 et T_2 quand il n'y a que des opérations d'édition de suppression et d'insertion.

3 Proposition pour la recherche d'information structurée

A notre connaissance, peu de travaux utilisent directement la théorie des arbres pour la recherche d'information structurée ad-hoc (à part les travaux d'Alilaouar (2007)), c'est à dire pour calculer l'appariement approximatif entre une requête structurée et un document XML. La distance d'alignement et l'inclusion d'arbres étant des cas particuliers de la distance d'édition, nous proposons l'utilisation de la distance d'édition pour obtenir une mesure générale et flexible dans la recherche d'information structurée. Nous proposons ici d'utiliser l'algorithme de Tai (1979) pour calculer la distance d'édition de deux arbres, et nous combinons cette distance d'édition portant sur la structure avec le score portant sur le contenu. D'autres algorithmes que celui là auraient pu être choisis, mais l'algorithme de Tai, le premier à avoir été proposé, nous semble être le plus simple à mettre en œuvre dans un premier temps.

3.1 Algorithme de Tai

Dans la suite de cette section, nous utilisons les notations suivantes :

1. Une forêt/arbre n'ayant pas de nœud (forêt/arbre vide) est dénoté(e) par le symbole Ø.
2. T(v) : arbre avec nœud racine v

3. F,G: forêt[1]
4. F-v : forêt obtenue après la suppression du nœud v de la forêt F
5. F- $T(r_F)$: forêt obtenue après la suppression entière de l'arbre extrême droite de la forêt F
6. n1n2 : forêt qui ne contient que deux nœuds n1, n2
7. cd(v) : coût de la suppression d'un nœud v
8. cs(a,b) : coût de substitution d'un nœud a par un nœud b
9. d(F,G) : distance d'édition entre les forêts F et G

L'algorithme de Tai est un algorithme récursif basé sur le lemme suivant (Tai (1979), Zhang (1996), Demaine et al. (2007)):

$$d(\emptyset, \emptyset) = 0$$
$$d(F, \emptyset) = d(F - r_F, \emptyset) + cd(r_F)$$
$$d(\emptyset, G) = d(\emptyset, G - r_G) + cd(r_G)$$

$$d(F, G) = \min \begin{cases} d(F - r_F, G) + cd(r_F), \\ d(F, G - r_G) + cd(r_G), \\ d(T(r_F) - r_F, T(r_G) - r_G) + d(F - T(r_F), G - T(r_G)) + cs(r_F, r_G) \end{cases}$$

Où F, G sont des forêts

L'algorithme est alors le suivant:

```
Function d(F,G)
{
      si (F = Ø) alors
            si (G = Ø) alors renvoyer 0
            si non renvoyer d(F- r_F, Ø)
      si (G = Ø) alors renvoyer d(Ø, G- r_G)
      a = d(F- r_F,G)+cd(r_F)
      b = d(F, G- r_G) + cd(r_G)
      c = d(T(r_F)- r_F,T(r_G)- r_G) + d(F- T(r_F),G- T(r_G)) +
cs(r_F,r_G)
      renvoyer {(a<[(b<c)?b:c])?a: [(b<c)?b:c]}
}
```

3.2 Application de l'algorithme à la recherche d'information structurée

Nous pouvons utiliser l'algorithme de Tai pour calculer la distance d'édition entre l'arbre de document (ou d'une partie de document) et l'arbre de la requête dans la recherche d'information dans des documents XML. Cette distance d'édition est combinée avec le score de contenu des nœuds feuilles descendants, obtenu avec un système de recherche

[1] Une **forêt** est un ensemble d'arbres. Un arbre peut être considéré comme une forêt d'un seul arbre. Après la suppression de la racine d'un arbre, il reste une forêt.

d'information XML classique (tel que XFIRM (Sauvagnat (2005)) par exemple) pour obtenir le score final d'un nœud ou un document[2]. Nous distinguons quatre cas :

1. **Cas 1 : Pour les requêtes simples du type $S[t_1,\ldots t_k]$[3], on a :**

$$score(n) = \lambda \,.\, \sum score_{nfi}(t_1 \ldots t_k) + (1- \lambda) \,.\, (|T(n)|-d(T(n),T(S))) \quad (1)$$

S : condition de structure de la requête

$t_1 \ldots t_k$: condition de contenu de la requête

n : nœud renvoyé (document ou partie de document (nœud interne n avec tous ses descendants))

T(n) : arbre avec nœud racine n

|T(n)| : nombre de nœuds dans l'arbre de racine n

T(S) : arbre de la requête

d(T(n),T(S)): distance d'édition entre l'arbre racine n et l'arbre T(S) de la requête

$score_{nfi}(t_1 \ldots t_k)$: score d'un nœud feuille descendant du nœud n et contenant un ou plusieurs termes de la requête.

λ: paramètre permettant d'affiner l'importance du contenu. $\lambda \in [0,1]$

Le score d'un nœud n répondant à la requête est donc fonction du score de contenu des nœuds feuilles qu'il contient et de la distance d'édition entre l'arbre dont il est racine et l'arbre de la requête.

2. **Cas 2 : Pour les requêtes du type $S1[t_1 \ldots t_p]$ // ec : $S2[t_{p+1} \ldots t_k]$[4],**

comme par exemple la requête *article[peinture] // ec: section[Dali]*[5], le score des nœuds n2 qui satisfont cette requête est le suivant :

$$score(n2) = \alpha \,.\, \sum score_{nfi}(t_{p+1} \ldots t_k) + \beta \,.\, score(n1) + (1- \alpha - \beta).(|T(n2)|-d(T(n2),T(S2))) \quad (2)$$

$score_{nfi}(t_{p+1} \ldots t_k)$: score d'un nœud feuille descendant du nœud n2 et contenant un ou plusieurs termes de la requête.

n1 : nœud qui satisfait la requête $S1[t_1 \ldots t_p]$, n1 est ancêtre de n2, et score(n1) est calculé comme le premier cas, et s'il n'existe pas de nœud n1, la partie β . score(n1) est égale à 0.

d(T(n2),T(S2)) : distance d'édition de l'arbre ayant pour racine n2 et du sous-arbre de la requête ayant pour racine S2

---

[2] Nous rappelons que dans un document XML, les informations de contenu sont situées au niveau des noeuds feuilles.

[3] On cherche une structure S qui vérifie la condition de contenu $t_1 \ldots t_k$.

[4] On cherche une structure S2 (structure cible) qui vérifie la condition de contenu $t_{p+1} \ldots t_k$, contenue dans une structure S1 (structure support) qui vérifie la condition de contenu $t_1 \ldots t_p$.

[5] Cette requête signifie que l'on cherche une section qui parle de « Dali » (structure cible) contenue dans un article qui parle de « peinture » (structure support).

α, β ∈ [0,1] sont des paramètres permettant d'affiner respectivement l'importance donnée au contenu et au nœud ancêtre.

Le score d'un nœud n2 répondant à la requête est donc fonction du score de contenu des nœuds feuilles qu'il contient, de la distance d'édition entre l'arbre dont il est racine et le sous-arbre de la requête portant sur l'élément cible, et du score d'un nœud n1 étant son ancêtre et répondant à la première partie de la requête.

3. Cas 3 : Pour les requêtes du type ec : S1[$t_1 \ldots t_p$] // S2[$t_{p+1} \ldots t_k$][6],

le score des nœuds n1 qui satisfont cette requête est le suivant :

$$score(n1) = \alpha \,.\, \sum score_{nfi}(t_1 \ldots t_p) + \beta \,.\, \sum score(n2) + (1- \alpha - \beta).\,(|T(n1)| - d(T(n1),T(S1))) \quad (3)$$

n2 : nœud qui satisfait la requête S2[$t_{p+1} \ldots t_k$], n2 est descendant de n1, et score(n2) est calculé comme le premier cas.

d(T(n1),T(S1)) : distance d'édition de l'arbre ayant pour racine n1 et l'arbre de la requête T(S1).

α, β ∈ [0,1] : paramètres permettant d'affiner respectivement l'importance donnée au contenu et aux nœuds internes descendants.

Le score d'un nœud n1 répondant à la requête est donc fonction du score de contenu des nœuds feuilles qu'il contient, de la distance d'édition entre l'arbre dont il est racine et le sous-arbre de la requête portant sur l'élément cible, et du score d'un nœud n2 étant son descendant et répondant à la seconde partie de la requête.

4. Cas 4 : Pour les requêtes du type S1[$t_1 \ldots t_p$] // ec : S2[$t_{p+1} \ldots t_q$] // S3[$t_{q+1} \ldots t_k$][7],

le score des nœuds n2 qui satisfont cette requête est le suivant :

$$score(n2) = \alpha \,.\, \sum score_{nfi}(t_{p+1} \ldots t_q) + \beta \,.\, \sum score(n3) + \gamma \,.score(n1) + (1- \alpha - \beta - \gamma).(|T(n2)| - d(T(n2),T(S2))) \quad (4)$$

n1 : nœud qui satisfait la requête S1[$t_1 \ldots t_p$], n1 est ancêtre de n2, et score(n1) est calculé comme le premier cas.

n3 : nœud qui satisfait la requête S3[$t_{q+1} \ldots t_k$], n3 est descendant de n2, et score(n3) est calculé comme le premier cas.

d(T(n2),T(S2)) : distance d'édition de l'arbre ayant pour racine n2 et du sous-arbre de requête ayant pour racine S2

---

[6] On cherche une structure S1 (structure cible) qui vérifie la condition de contenu $t_1 \ldots t_p$, et contenant une structure S2 (structure support) qui vérifie la condition de contenu $t_{p+1} \ldots t_k$.

[7] On cherche une structure S2 (structure cible) qui vérifie la condition de contenu $t_{p+1} \ldots t_q$, contenue dans une structure support S1 vérifiant $t_1 \ldots t_p$ et contenant une structure S3 vérifiant $t_{q+1} \ldots t_k$.

$\alpha, \beta, \gamma \in [0,1]$: paramètres permettant d'affiner respectivement l'importance donnée au contenu, aux nœuds internes descendants, et au nœud ancêtre.

Le score d'un nœud n2 répondant à la requête est donc fonction du score de contenu des nœuds feuilles qu'il contient, de la distance d'édition entre l'arbre dont il est racine et le sous-arbre de la requête portant sur l'élément cible, du score d'un nœud n3 étant son ancêtre et répondant à la première partie de la requête et enfin du score d'un nœud n1 étant son descendant et répondant à la troisième partie de la requête.

Les types de requêtes énoncés ici sont représentatifs de la majorité des requêtes structurées portant sur des documents structurés. Pour plus d'information sur ces requêtes, on peut se référer à Trotman et Sigurbjornsson (2005) et Sauvagnat (2005).

4 Illustration sur un exemple

Nous présentons dans cette section un exemple de calcul des scores de pertinence pour le document XML de la figure 1 (A) et la requête de la figure 1 (B) : *article[peinture] // ec: section[Dali]* (recherche de sections qui parlent de « Dali », dans un article qui parle de « peinture »), en utilisant l'algorithme combinaison de Tai et le score des nœuds feuilles descendants calculé sur le contenu avec un système de recherche d'information XML classique.

Pour calculer la distance d'édition, le coût de suppression d'un nœud v (cd(v)) et le coût de substitution d'un nœud a par un nœud b (cs(a,b)) sont proposés par exemple comme suit :

cd(v) = 1
cs(a,b) = 1-similarité(a,b)
Lorsque cd(v) = 1, d(F, Ø) = d(Ø, F) = nombre de nœuds de F.

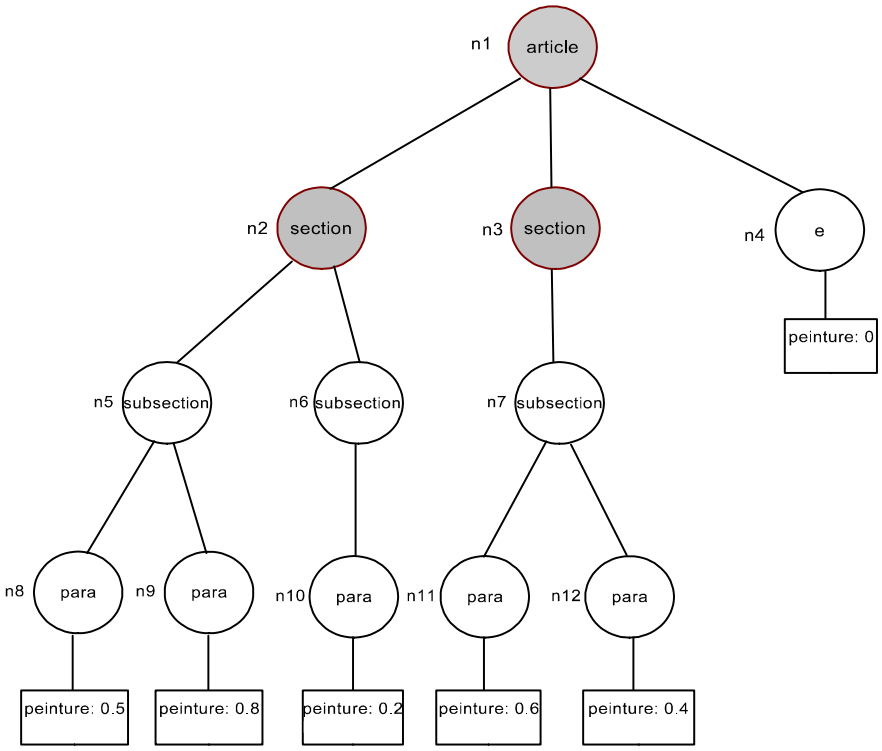

FIG. 1 (A) – *Arbre du document.*

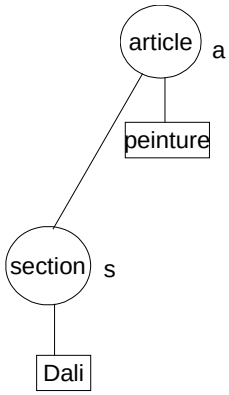

FIG. 1 (B) – *Arbre de la requête.*

Le tableau 1 donne les similarités possibles entre les nœuds du document et les nœuds de la requête. Ces similarités sont évaluées manuellement, en fonction de la sémantique des balises des nœuds concernés. Par exemple, un nœud *section* et un nœud *subsection*

ont une similarité de ½, (ils ont des sémantiques relativement proches) alors qu'un nœud *section* et un nœud *para* ont une similarité de ¼ (ils ont des sémantiques plus éloignées).

Le coût de substitution cs(a,b) est égal à 1 – similarité(a,b). Par exemple cs(a,n1) = 1- 1= 0 ; cs(s,n5) = 1- 1/2 =1/2.

	n1 (article)	n2,n3 (section)	n4 (e)	N5,n6,n7 (subsection)	n8=>n12 (para)
a(article)	1	1/2	0	0	0
s(section)	1/2	1	0	1/2	1/4

TAB. 1 – Similarité entre les nœuds

Au début, deux requêtes structurées *article[peinture]* et *section[Dali]* sont traitées par le système de recherche XML classique . Les résultats de ces requêtes sont respectivement le nœud n1 (article) et les deux nœuds section (n2,n3).

Ensuite, trois distances d'édition : la distance d(T(n1),T(a)) de l'arbre T(n1) avec nœud racine n1 et l'arbre de requête T(a) , la distance d(T(n2), T(s)) de l'arbre avec nœud racine n2 et le sous-arbre de requête T(s) avec nœud racine s, et la distance d(T(n3),T(s)) de l'arbre racine n3 et le sous-arbre T(s) de la requête avec racine s sont calculées par l'algorithme de Tai. Les étapes du calcul de ces distances sont détaillées en annexe 1.

Le score de nœud article n1 est calculé ainsi:

$$score(n1)= \lambda . \sum_{nf_i \in descendants(n1)} score_{nf_i}(peinture) +(1- \lambda) . (|D|-d(T(n1),T(a)))$$

où : descendants(n1) est l'ensemble des nœuds feuilles descendants de n1.

$score_{nf_i}(peinture)$ est le score de pertinence du nœud feuille nf_i descendant de n1 qui contient le mot clé « *peinture* »

d(T(n1),T(a)) = 10 et |T(n1)| = 12

Enfin, les résultats de la requête sont les nœuds section n2 et n3, avec le score final

$$score(n2)= \alpha \text{ x } \sum_{nf_i \in descendants(n2)} score_{nf_i}(Dali) + \beta \text{ x } score(n1) + (1- \alpha - \beta) \text{ x } (|T(n2)|-d(T(n2),T(s)))$$

$$score(n3)= \alpha \text{ x } \sum_{nf_i \in descendants(n3)} score_{nf_i}(Dali) + \beta \text{ x } score(n1) + (1- \alpha - \beta) \text{ x } (|T(n3)|-d(T(n3),T(s)))$$

où : descendants(n2) est l'ensemble des nœuds feuilles descendants de n2.

descendants(n3) est l'ensemble des nœuds feuilles descendants de n3.

$score_{nf_i}(Dali)$ est le score de pertinence du nœud feuille nf_i qui contient le mot clé « *Dali* ».

d(T(n2),T(s)) = 5, d(T(n3),T(s)) = 3, |T(n2)| = 6, et |T(n3)| = 4

5 Conclusion et Perspectives

Dans cet article, nous avons proposé une mesure de similarité de structure entre des documents XML. Cette mesure est basée sur la distance d'édition des arbres qui représentent ces documents, en utilisant l'algorithme de Tai (1979). La similarité entre des documents XML et la requête est la combinaison de la similarité de structure et la similarité de contenu (en utilisant la similarité de contenu d'un moteur de recherche XML classique).

Nos propositions sont en cours d'évaluation sur les collections d'INEX, tâche Ad-hoc. INEX (*Initiative for the Evaluation of XML Retrieval*) est la campagne d'évaluation traditionnellement utilisée pour évaluer les systèmes de recherche d'information structurée.

Notre approche de recherche ad-hoc peut cependant être améliorée selon plusieurs points :

- Concernant la vitesse de calcul de la distance d'édition : (i) des algorithmes plus rapides comme par exemple l'algorithme de Demaine et al. (2007) ou encore des algorithmes tournant sur des arbres de faible hauteur peuvent être utilisés; (ii)des contraintes issues de la DTD peuvent être utilisées pour réduire les calculs et des limitations sur des opérations d'édition peuvent également être mises en œuvre.
- Une mesure de similarité flexible peut être obtenue en utilisant des opérations d'édition floues.
- La combinaison entre la similarité de contenu et la similarité de structure doit être approfondie.

D'autre part, nous envisageons de proposer des méthodes basées sur les graphes pour résoudre d'autres problèmes de la recherche d'information structurée, comme par exemple les problèmes d'hétérogénéité de corpus et de classification de documents XML. Ces problématiques et les solutions associées pourront être évaluées sur les tâches hétérogènes et XML Mining de la campagne d'évaluation INEX.

6 Bibliographie

Alilaouar A. (2007). *Contribution à l'interrogation flexible de données semi-structurées*. Thèse de l'Université Paul Sabatier 2007.

Bunke H. (1998). *Error-Tolerant Graph Matching: A Formal Framework and Algorithms*. Lecture notes in computer science, 1998 – Springer.

Bunke H., Shearer K. (1998). *A graph distance metric based on the maximal common subgraph*. Pattern Recognition Letters 19(1998)255–259.

Bunke H. (2000). *Recent Developments in Graph Matching*, IEEE 2000.

Chen W. (1998). *More Efficient Algorithm for Ordered Tree Inclusion*. JOURNAL OF ALGORITHMS, Academic Press 1998.

Dalamagas T., Cheng T., Winkel K., and Sellis T. (2004). *Clustering XML Documents Using Structural Summaries*. W. Lindner et al. (Eds.): EDBT 2004 Workshops, LNCS 3268, pp. 547–556, 2004.

Dalamagas T., Cheng T., Winkel K., and Sellis T. (2006). A methodology for clustering XML documents by structure. Information Systems 31.

Demaine E. D., Mozes S., Rossmao B., et Weimann O. (2007). *An Optimal Decomposition Algorithm for Tree Edit Distance*. ACM Journal Name, Vol. TBD, No. TBD, TBD 20TBD (2007).

Fuhr N. Kamps J., Lalmas M., Trotman A. (2008). *Focused Access to XML Documents, 6th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2007*, Dagstuhl Castle, Germany, December 17-19, 2007. Selected Papers,Springer,Lecture Notes in Computer Science, volume 4862, 2008.

Geva S., Kamps J. , Trotman A. (2009) *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008*, Dagstuhl Castle, Germany, December 15-18, 2008. Revised and Selected Papers, Springer,Lecture Notes in Computer Science, volume 5631, 2009.

Hidovi D., Pelillo M. (2004) . *Metrics for Attributed Graphs Based on the Maximal Similarity Common Subgraph*. International Journal of Pattern Recognition and Artificial, 7/2004.

Jiang T., Wang L. , Zhang K. (1995). *Alignment of trees - an alternative to tree edit*. Theoretical Computer Science 143, 1995.

Klein P. N. (1998). *Computing the Edit-Distance Between Unrooted Ordered Trees*. Springer-Verlag Berlin Heidelberg 1998.

Lee M. D. (2009). *Graphes et appariement sémantique de documents XML*. Rapport de Master 2 Recherche, Université Paul Sabatier, 2009.

Manning C. D., Raghavan P., et Schutze H (2008).*Introduction to Information Retrieval.* Cambridge University Press 2008

Pinel-Sauvagnat K., Boughanem M (2006). *Propositions pour la pondération des termes et l'évaluation de la pertinence des éléments en recherche d'information structurée.* Revue I3 - Information Interaction Intelligence - Volume 6, n°2.

de Rougemont M. (2005). *The correction of XML data*. Lecture notes in computer science, 2005.

de Rougemont M., Vieilleribière A.. *Approximate Schemas, Source-Consistency and Query Answering*. Journal of Intelligent Information Systems, 2008 – Springer.

Sauvagnat K. (2005). *Modèle flexible pour la Recherche d'Information dans des corpus de documents semi-structurés*. Thèse de l' Université Paul Sabatier de Toulouse 2005.

Shasha D. , Zhang K. (1997). *Approximate tree pattern matching*. In Pattern Matching Algorithms, 1997.

Sanz I., Mesiti M., Guerrini G., and Berlanga Llavori R. (2005). *Approximate Subtree Identification in Heterogeneous XML Documents Collections*. Lecture notes in computer science, 2005.

Tai K (1979). *The tree-to-tree correction problem*, J. ACM 26 (3), p. 422-433.

Trotman A., Sigurbjornsson B. (2005). *NEXI, Now and Next*. INEX 2004, Springer-Verlag Berlin Heidelberg 2005.

Wallis W.D. , Shoubridge P., Kraetz M., Ray D.. *Graph distances using graph union*. Pattern Recognition Letters 22 (2001).

Zhang K. (1996). *A Constrained Edit Distance Between Unordered Labeled Trees*. Springer-Verlag, Algorithmica 1996.

Annexe 1

Étapes de calcul de la distance d'édition d(T(n1),T(a)):

d(T(n1),T(a)) = min {d(T(n1)-n1,T(a)) + cd(n1),
d(T(n1),s) + cd(a),
d(T(n1)-n1,s) + d(Ø,Ø) + cs(n1,a) }
d(T(n1)-n1,T(a)) = min {d(T(n1)-n1-n4,T(a)) + cd(n4),
d(T(n1)-n1,s) + cd(a),
d(Ø,s) + d(T(n1)-n1-n4,Ø) + cs(n4,a) }
d(T(n1),s) = min { d(T(n1)-n1,s) + cd(n1),
d(T(n1),Ø) + cd(s),
d(T(n1)-n1,Ø) + d(Ø,Ø) + cs(n1,s) }
d(T(n1)-n1,s) = min {d(T(n1)-n1-n4,s) + cd(n4),
d(T(n1)-n1,Ø) + cd(s),
d(Ø,Ø) + d(T(n1)-n1-n4,Ø) + cs(n4,s)
d(T(n1)-n1-n4,T(a)) = min { d(T(n1)-n1-n4-n3,T(a)) + cd(n3),
d(T(n1)-n1-n4,s) + cd(a),
d(T(n7), s) + d(T(n2),Ø) + cs(n3,a) }
d(T(n1)-n1-n4,s) = min {d(T(n1)-n1-n4-n3,s) + cd(n3),
d(T(n1)-n1-n4,Ø) + cd(s),
d(T(n7),Ø) + d(T(n2),Ø) + cs(n3,s) }
d(T(n1)-n1-n4-n3,T(a)) = min { d(T(n1)-n1-n4-n3-n7,T(a)) + cd(n7),
d(T(n1)-n1-n4-n3,s) + cd(a),
d(n11n12,s) + d(T(n2),Ø) + cs(n7,a) }
d(T(n7), s) = min { d(n11n12, s) + cd(n7),

d(T(n7), n) + cd(s),
d(n11n12,Ø) + d(Ø,Ø) + cs(n7,s) }

d(T(n1)-n1-n4-n3,s) = min { d(T(n1)-n1-n4-n3-n7,s) + cd(n7),
d(T(n1)-n1-n4-n3,Ø) + cd(s),
d(n11n12,Ø) + d(T(n2),Ø)) + cs(n7,s)}

d(T(n1)-n1-n4-n3-n7,T(a)) = min { d(T(n1)-n1-n4-n3-n7-n12,T(a)) + cd(n12),
d(T(n1)-n1-n4-n3-n7,s) + cd(a),
d(Ø, s) + d(T(n2) + n11,Ø) + cs(n12,a) }

d(n11n12,s) = min { d(n11,s) + cd(n12),
d(n11n12,Ø) + cd(s),
d(Ø,Ø) + d(n11,Ø) + cs(n12,s)

d(T(n1)-n1-n4-n3-n7,s) = min { d(T(n1)-n1-n4-n3-n7-n12,s) + cd(n12),
d(T(n1)-n1-n4-n3-n7,Ø) + cd(s),
d(Ø, n) + d(T(n2) + n11,Ø) + cs(n12,s) }

d(T(n1)-n1-n4-n3-n7-n12,T(a)) = min { d(T(n2),T(a)) + cd(n11),
d(T(n1)-n1-n4-n3-n7-n12,s) + cd(a),
d(Ø,s) + d(T(n2),Ø) + cs(n11,a) }

d(n11,s) = min {d(Ø,s) + cd(n11),
d(n11,Ø) + cd(s),
d(Ø,Ø) + d(Ø,Ø) + cs(n11,s)

d(T(n1)-n1-n4-n3-n7-n12,s) = min { d(T(n2),s) + cd(n11),
d(T(n1)-n1-n4-n3-n7-n12,Ø) + cd(s),
d(Ø,Ø) + d(T(n2),Ø) + cs(n11,s) }

d(T(n2),T(a)) = min {d(T(n2)-n2,T(a)) + cd(n2),
d(T(n2),s) + cd(a),
d(T(n2)-n2,s) + d(Ø,Ø) + cs(n2,a) }

d(T(n2),s) = min { d(T(n2)-n2,s) + cd(n2),
d(T(n2),Ø) + cd(s),
d(T(n2)-n2,Ø) + d(Ø,Ø) + cs(n2,s) }

d(T(n2)-n2,T(a)) = min { d(T(n2)-n2-n6,T(a)) + cd(n6),
d(T(n2)-n2,s) + cd(a),
d(n10,s) + d(T(n5),Ø) + cs(n6,a) }

d(T(n2)-n2,s) = min { d(T(n2)-n2-n6,s) + cd(n6),
d(T(n2)-n2,Ø) + cd(s),
d(n10,Ø) + d(T(n5),Ø) + cs(n6,s) }

d(T(n2)-n2-n6,T(a)) = min { d(T(n5),T(a)) + cd(n10),
d(T(n2)-n2-n6,s) + cd(a),
d(Ø,s) + d(T(n5),Ø) + cs(n10,a) }

d(n10,s) = min {d(Ø,s) + cd(n10),
d(n10,Ø) + cd(s),
d(Ø,Ø) + d(Ø,Ø) + cs(n10,s)

d(T(n2)-n2-n6,s) = min { d(T(n5),s) + cd(n10),
d(T(n2)-n2-n6,Ø) + cd(s),
d(Ø,Ø) + d(T(n5),Ø) + cs(n10,s) }

d(T(n5),T(a)) = min { d(n8n9,T(a)) + cd(n5),

d(T(n5),s) + cd(a),
d(n8n9,s) + d(Ø,Ø) + cs(n5,a) }
d(T(n5),s) = min { d(n8n9,s) + cd(n5),
d(T(n5),Ø) + cd(s),
d(n8n9,Ø) + d(Ø,Ø) + cs(n5,s) }
d(n8n9,T(a)) = min { d(n8,T(a)) + cd(n9),
d(n8n9,s) + cd(a),
d(Ø,s) + d(n8,Ø) + cs(n9,a) }
d(n8n9,s) = min { d(n8,s) + cd(n9),
d(n8n9,Ø) + cd(s),
d(Ø,Ø) + d(n8,Ø) + cs(n9,s) }
d(n8,T(a)) = min { d(Ø,T(a)) + cd(n8),
d(n8,s) + cd(a),
d(Ø,s) + d(Ø,Ø) + ds(n8,a) }
d(n8,s) = min { d(Ø,s) + cd(n8),
d(n8,Ø) + cd(s),
d(Ø,Ø) + cs(n8,s) }
(d(n8,s) = min { 1 + 1, 1 + 1, 1-1/4 }= 3/4
d(n8,T(a)) = min { 2 + 1, 3/4+ 1,1 + 1 } = 7/4
d(n8n9,s)= min { 7/4+ 1, 2 + 1, 1 + 3/4 } = 7/4)
d(n8n9,T(a)) = min { 1, 7/4 (10.2) + 1, 1 + 1 +1 } = 11/4
d(T(n5),s) = min { 7/4 + 1, 3 +1, 2 + 0 + 1/2 } = 5/2
d(T(n5),T(a)) = min { 11/4 + 1, 5/2 + 1, 7/4 + 0 + 1 }=11/4
d(T(n2)-n2-n6,s) = min { 5/2+ 1, 4+ 1, 3 + 3/4 }=7/2
d(n10,s) = min {1 + 1, 1 + 1, 3/4} =3/4
d(T(n2)-n2-n6,T(a)) = min { 11/4+ 1, 3.75, 7/2 + 1, 4.5,1 + 3 + 1 } = 3.75
d(T(n2)-n2,s) = min { 7/2+ 1, 5 + 1, 1 + 3 + 1/2 }= 4.5
d(T(n2)-n2,T(a)) = min {3.75 + 1, 4.5 + 1, 3/4+ 3 + 1 }=4.75
d(T(n2),s) = min { 4.5 + 1, 6 + 1, 5 + 0 + 0 } = 5
d(T(n2),T(a)) = min { 4.75+ 1,) 5 + 1, 4.5 + 1/2 } = 5
d(T(n1)-n1-n4-n3-n7-n12,s) = min { 5+ 1, 7 + 1, 0 + 6+ 3/4 } = 6
d(n11,s) = min {1 + 1, 1 + 1, 0 + 3/4} = 0.75
d(T(n1)-n1-n4-n3-n7-n12,T(a)) = min { 5 + 1, 6+ 1, 1 + 6 + 1 } = 6
d(T(n1)-n1-n4-n3-n7,s) = min { 6+ 1, 8 + 1, 0 + 7 + 3/4 } = 7
d(n11n12,s) = min { 0.75+ 1, 2 + 1, 0 + 1+ 0.75} = 1.75
d(T(n1)-n1-n4-n3-n7,T(a)) = min { 6+ 1, 7 + 1, 1 + 7 + 1} = 7
d(T(n1)-n1-n4-n3,s) = min { 7 + 1, 9 + 1, 2 + 6 + 1/2} = 8
d(T(n7), s) = min { 1.75+ 1, 3 + 1, 2 + 0 + 1/2 } = 2.5
d(T(n1)-n1-n4-n3,T(a)) = min { 7+1, 8 + 11.75+ 6 + 1 } = 8
d(T(n1)-n1-n4,s) = min { 8+ 1, 10 + 1, 3 + 6 + 0 } = 9
d(T(n1)-n1-n4,T(a)) = min { 8+ 1, 9+ 1, 2.5 + 6 + 1/2 } = 9
d(T(n1)-n1,s) = min { 9+ 1, 11 + 1, 0 + 10 + 1 } = 10
d(T(n1),s) = min { 10+ 1, 12 + 1, 11 + 0 + 1/2} = 11
d(T(n1)-n1,T(a)) = min { 9+ 1, 10 + 1, 1 + 10 + 1 } = 10
d(T(n1),T(a)) = min { 10+ 1, 11+ 1, 10+ 0+ 0} = 10)

d(T(n1),T(a)) = 10

Étapes de calcul de la distance d'édition d(T(n2),T(s))

d(T(n2),T(s)) = d(T(n2),s) = min { d(T(n2)-n2,s) + cd(n2),
d(T(n2),Ø) + cd(s),
d(T(n2)-n2,Ø) + d(Ø,Ø) + cs(n2,s) }
d(T(n2)-n2,s) = min { d(T(n2)-n2-n6,s) + cd(n6),
d(T(n2)-n2,Ø) + cd(s),
d(n10,Ø) + d(T(n5),Ø) + cs(n6,s) }
d(T(n2)-n2-n6,s) = min { d(T(n5),s) + cd(n10),
d(T(n2)-n2-n6,Ø) + cd(s),
d(Ø,Ø) + d(T(n5),Ø) + cs(n10,s) }
d(T(n5),s) = min { d(n8n9,s) + cd(n5),
d(T(n5),Ø) + cd(s),
d(n8n9,Ø) + d(Ø,Ø) + cs(n5,s) }
d(n8n9,s) = min { d(n8,s) + cd(n9),
d(n8n9,Ø) + cd(s),
d(Ø,Ø) + d(n8,Ø) + cs(n9,s) }
d(n8,s) = min { d(Ø,s) + cd(n8),
d(n8,Ø) + cd(s),
d(Ø,Ø) + d(Ø,Ø) + cs(n8,s) }
(d(n8,s) = min { 1 + 1, 1 + 1, 0 + 0 + 0.75 } = 0.75
d(n8n9,s) = min { 0.75 + 1, 2 + 1, 0 + 1 + 0.75 } = 1.75
d(T(n5),s) = min { 1.75 + 1, 3 + 1, 2 + 0 + 1/2} = 2.5)
d(T(n2)-n2-n6,s) = min { 2.5+ 1, 4+ 1, 0 + 3+ 075 } = 3.5
d(T(n2)-n2,s) = min { 3.5+ 1, 5 + 1, 1 + 3 + 1/2} =4.5
d(T(n2),s) = min { 4.5+ 1, 6 + 1, 5 + 0 + 0 } = 5)

d(T(n2),T(s)) = 5

Étapes de calcul de la distance d'édition d(T(n3),T(s))

d(T(n3),T(s))=d(T(n3),s) = min { d(T(n7),s) + cd(n3),
d(T(n3),Ø) + cd(s),
d(T(n7),Ø) + d(Ø,Ø) + cs(n3,s) }
d(T(n7),s) = min {d(n11n12,s) + cd(n7),
d(T(n7),Ø) + cd(s),
d(n11n12,Ø) + d(Ø,Ø) + cs(n7,s) }
d(n11n12,s) = min { d(n11,s) + cd(n12),
d(n11n12,Ø) + cd(s),
d(Ø,Ø) + d(n11,Ø) + cs(n12,s)
d(n11,s) = min {d(Ø,s) + cd(n11),

d(n11,Ø) + cd(s),
d(Ø,Ø) + d(Ø,Ø) + cs(n11,s)

(d(n11,s) = min {1+ 1, 1 + 1, 0+ 0.75 } = 0.75
d(n11n12,s) = min { 0.75+ 1, 2 + 1, 0 + 1 + 0.75 } = 1.75
d(T(n7),s) = min { 1.75 + 1, 3 + 1, 2 + 0 + 1/2} = 2.5
d(T(n3),s) = min { 2.5+ 1, 4 + 1, 3 + 0 + 0 } = 3)

d(T(n3),T(s)) = 3

Les interdépendances structurelles, un problème complexe traité par les graphes de coordination

Khalil Drira*,**

*CNRS ; LAAS ; 7 Avenue du Colonel Roche, F-31077 Toulouse, France
**Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
khalil@laas.fr
http://homepages.laas.fr/khalil/page

Résumé. Les travaux présentés traitent les problèmes relatifs à la coordination d'un ensemble de composants, laquelle inclut, en particulier, l'intégration et la distribution de ces composants sous différentes contraintes architecturales : interdépendance, dynamisme et distribution. La gestion de la dynamique de l'architecture et de ses interactions distribuées ainsi que son application pour le support des activités génériques de coopération ont été deux axes majeurs dans les problèmes traités. Cet article présente une approche basée sur les graphes pour le traitement de ces problèmes.

1 Introduction

Une activité coopérative est une activité en groupe, collaborative et coordonnée qui implique simultanément/ou à différents moments, différents “acteurs" artificiels ou humains, ayant des objectifs communs, qui produisent et échangent des données/des informations/ou des connaissances, au cours de sessions spontanées ou planifiées, et selon des règles de coordination implicites ou explicites pour la gestion des tâches et le partage de l'espace de travail.

Un environnement de support des activités en groupe (1) supporte la collaboration de ses utilisateurs, en mettant à leur disposition des outils spécialisés (dépendant du domaine d'activité) ou génériques (outils de discussion multimédia, outils d'édition partagée), et (2) intègre ces outils via des services de coordination qui introduisent la structuration, le contrôle, et gèrent la cohérence globale.

Les **interdépendances structurelles** constituent la pierre angulaire du problème de coordination qui garantit la cohérence. Nous les classons en trois catégories :

La première catégorie est celle des interdépendances relatives à l'**architecture des sites de coopération** (impliquant un ou plusieurs composants d'une ou plusieurs applications). Elles concernent la coordination des interdépendances entre composants induites par

- des contraintes du niveau communication, relatives notamment à la gestion des connexions (le serveur et le proxy doivent partager le même canal de communication et leur initialisation doit être coordonnée)
- ou des exigences du niveau coopération relatives aux règles de répartition des privilèges et d'attribution des droits (le composant central de gestion du droit de parole doit être activé d'abord sur la machine du modérateur de session).

La deuxième catégorie est celle des interdépendances relatives à la **structuration du groupe de coopération**. Elles concernent la coordination des interdépendances entre les acteurs et expriment généralement des liens de type producteur / consommateur à gérer de façon évolutive dans les structures dynamiques ou de façon figée dans les structures hiérarchiques.

La troisième catégorie est celle des interdépendances relatives à la **structure de l'espace de coopération**. Elles concernent la coordination des interdépendances entre les "objets de coopération" partagés (par ex. documents ou composantes de documents) et expriment par exemple des relations de "structuration linéaire" de type "prédécesseur (avant) / successeur (après)" ou des relations de "décomposition arborescente (ou hiérarchique)" de type "objet de coopération / composantes". On peut voir ainsi qu'un document édité en groupe (c'est ce que nous appelons objet de coopération dans ce cas) est structuré :

- de façon linéaire comme la succession de caractères, de mots, de lignes, ou de paragraphes (c'est que nous appelons composantes dans ce cas)
- ou de façon arborescente (ou hiérarchique) comme l'imbrication de parties, de chapitres, de sections (c'est que nous appelons composantes dans ce cas).

Pour traiter le problème de la **gestion des interdépendances structurelles**, nous avons conduit des études couvrant l'ensemble des catégories distinguées. Nous avons développé une technique générique de **"coordination guidée par la structure"** qui s'appuie sur des descriptions de *graphes* évolutifs étendus associés à des *règles de transformation* pouvant décrire les différentes évolutions d'une structure quelconque en général et d'une architecture logicielle plus particulièrement, leurs conditions et leurs conséquences, ceci de façon adaptée aux exigences du niveau coopération et aux contraintes et capacités du niveau communication Drira (2000). Les exigences du niveau coopération peuvent concerner, par exemple, la distribution du droit de modification des documents partagés dans un groupe d'utilisateurs avec différents critères de dynamicité et de granularité : par exemple, ne pas associer le droit obligatoirement à la totalité du contenu d'un document, et ne pas attribuer ce droit selon une répartition statique basée sur les identités ou les rôles, mais plutôt permettre de définir et d'attribuer dynamiquement le droit par partie de document et aux différents participants.

Le papier est organisé comme suit. La section 2.2 positionne nos travaux par rapport aux autres approches de coordination. La section 3 décrit la descrition des interdépendances structurelles par les graphes de coordination. La section 4 décrit, de façon informelle puis formelle, l'application de notre approche pour la gestion des architectures dynamiques appariées comme des objets de coordination complexes. La dernière section est consacrée au bilan et état de nos travaux.

2 Position du problème et positionnement

2.1 Le problème général de la coordination

De nombreux travaux sur la coordination dans les systèmes de coopération font référence aux définitions de Malone et Crowston qui considèrent que la coordination est l'acte de travailler ensemble harmonieusement. ("The act of working together harmoniously" Malone et Crowston (1990)) et qu'elle consiste à gérer les dépendances entre les activités" ("coordination is the managing of dependencies between activities" Malone et Crwoston (1994)). L'identifi-

cation des dépendances[1] (les domaines de coordination, les éléments de coordination et les relations) et la définition des procédures des gestion (mécanismes, protocoles, algorithmes) sont les deux axes autour desquels s'articulent les travaux de ce domaine. C'est aussi le cas pour leurs dérivations ou spécialisations en CSCW, en DAI, et en CSCL, ainsi que leurs applications à la gestion des espaces partagés, à la coordination pour le groupware et à la gestion du workflow.

2.2 La coordination dans les logiciels de support à la coopération

Les travaux liés aux supports logiciels pour les activités de coopération distribuée développent de nouvelles architectures et de nouveaux logiciels pouvant servir à une nouvelle organisation des procédures de travail en groupes géographiquement distribués qui coopèrent par l'échange d'informations qu'ils interprètent et transforment en connaissance pour l'achèvement d'un ou plusieurs objectifs. Ces objectifs peuvent être communs justifiant l'intérêt partagé pour leur achèvement, ou individuels, et il s'agit dans ce cas de "coopération" motivée par la complémentarité des moyens et des compétences. Les membres de ces groupes se réunissent périodiquement ou selon les besoins pour coordonner leurs contributions lors de sessions de travail selon un planning prédéfini ou improvisé.

Lorsque l'on s'intéresse aux travaux traitant des systèmes multi-utilisateurs, et plus particulièrement de leur application au support des activités de coopération (CSCW), l'on distingue de fortes contributions centrées sur la coordination qui visent l'une ou l'autres des deux fonctions que sont : "la gestion de la concurrence", et la gestion de la cohérence (ou consistence)". Les deux axes principaux concernés par cette recherche appliquée au CSCW sont, d'après la taxonomie de ce domaine Mills (2003), (1) la gestion du workflow, et (2) la gestion d'un espace d'information commun partagé (Figure 1).

Dans la catégorie du workflow, les utilisateurs agissent localement selon des procédures (ou tâches) préalablement planifiées, sur des données (objets, documents, bases de données, fichiers) privées ou communes.

Les travaux de recherche dans le cadre de cette catégorie relèvent du domaine de l'automatisation des processus d'entreprise selon des théories et des approches sociologiques ou organisationnelles. Il s'agit principalement de décomposer une activité en tâches et sous tâches, d'ordonnancer les tâches (principal thème de recherche de la communauté de recherche workflow) et de gérer leur affectation aux participants. Il s'agit essentiellement de la planification à laquelle s'intéresse aussi la recherche dans le domaine de l'intelligence artificielle distribuée pour la résolution coopérative de problèmes (Figure 1).

Les activités de la deuxième catégorie sont caractérisées par des sessions de coopération synchrones au cours desquelles les participants agissent simultanément et depuis des points d'accès distribués sur des objets partagés en suivant des règles de coordination pouvant être implicites ou explicites et en utilisant un ensemble d'outils qui leur permettent de progresser de façon coordonnée. Les objets peuvent être virtuellement ou réellement centralisés dans un espace de coopération partagé. Les objets peuvent représenter des fenêtres graphiques d'une application, des documents ou des parties dans ces documents, ou tout autre support en fonction de l'activité.

[1] appelées aussi interdépendances.

Les interdépendances structurelles traitées par les graphes de coordination

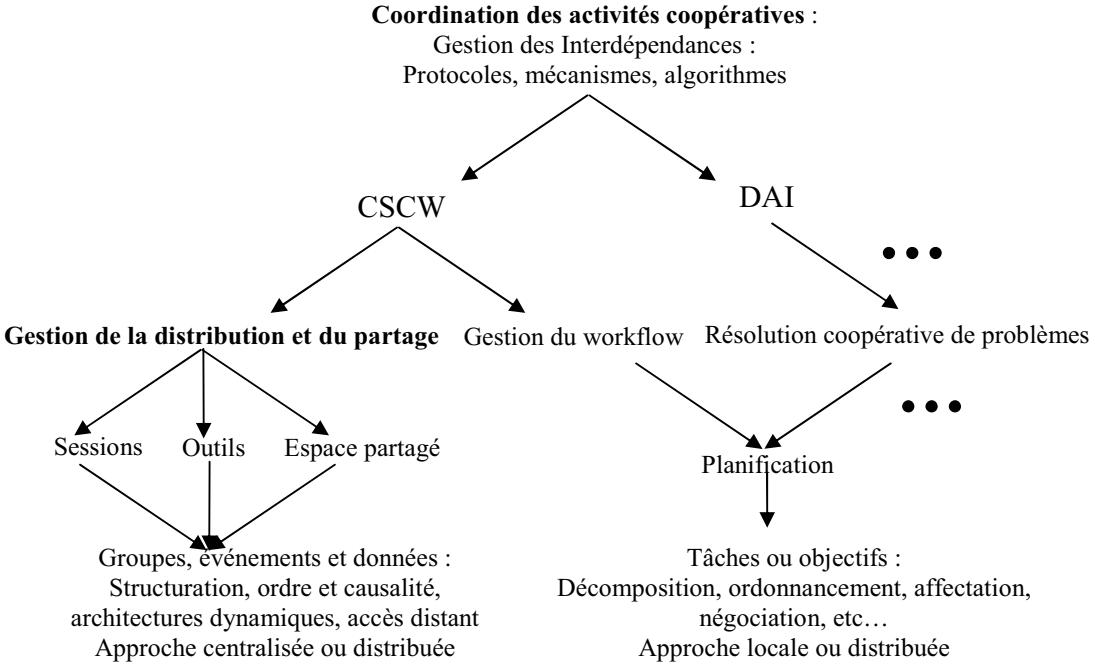

FIG. 1 – *Les différents domaines de recherche sur la coordination*

L'ensemble de nos contributions se situent dans cette deuxième catégorie. Elles portent sur la gestion de la cohérence au niveau de "l'espace d'information commun partagé" (vu comme une collection "d'objets de coopération") et au niveau de la structure logicielle qui soutient les utilisateurs dans le partage de cet espace et dans leur interaction.

Ces contributions concernent, dans une première partie, des solutions pour la gestion de la causalité entre les événements émis ou reçus par les participants et les outils de coopération qu'ils utilisent.

Dans une deuxième partie, nos contributions portent sur des solutions pour initialiser et adapter la structure afin de permettre à l'ensemble du système de fonctionner sous l'hypothèse de limitation ou d'absence de capacité de traitement de la coordination par les outils de coopération et par les composants qu'ils intègrent.

3 La description des interdépendances structurelles par les graphes de coordination

Notre approche se base sur les graphes de coordination définis pour représenter l'architecture des applications coopératives distribuées. L'étude de différentes familles d'applications coopératives a montré en effet qu'il était possible de les concevoir selon une architecture sy-

métrique d'un site à l'autre. Les particularités de chaque site sont représentées en paramètre de l'instance des composants qui constituent ce site. Le graphe de coordination est un graphe dont les nœuds sont associés à 3 types d'information : un type, une étiquette, et un ensemble de propriétés (appelées aussi paramètres) définies par le programmeur pour les besoins de son protocole. A ces informations, s'ajoutent : un identifiant géré par le système et une association $\langle$*type de nœud, classe de comportement*$\rangle$ choisie par le programmeur et utilisée par le système pour créer des instances de comportements.

L'évolution du graphe de coordination se fait selon un protocole de coordination qui utilise un ensemble de règles de transformation. De façon similaire aux règles de production des grammaires de graphes, l'application d'une règle de transformation exige la présence d'un certain motif dans le graphe de coordination. Le motif recherché fait partie d'un schéma de transformation plus général appelé règle de coordination et qui se décline en trois parties :

- le motif "Recherché" : R,
- la partie du motif à effacer ("Détruire") sur le graphe de coordination initial : $D \subseteq R$
- une partie supplémentaire à insérer ("Ajouter") au graphe nouvellement obtenu : A.

L'ensemble de ces trois parties, $\langle (R \setminus D) \cup D \cup A \rangle$, doit former un graphe, appelé graphe de la règle. Le graphe de la règle déclare des nœuds et des arcs pouvant être identiques à ceux du graphe de coordination. La recherche d'un motif dans le graphe initial applique alors des règles d'unification considérant les types des nœuds : deux nœuds sont unifiables s'ils ont le même type. Lorsque l'unification des types est insuffisante pour le problème en cours de description, le modèle offre la possibilité d'unifications des labels qui étiquettent les nœuds : deux nœuds sont unifiables s'ils sont unifiable par le type et s'ils ont les mêmes étiquettes. Et pour augmenter la puissance d'expression des règles de coordination, il est possible d'utiliser des nœuds dont le type et/ou l'étiquette sont génériques et unifiables avec des types différents ou des étiquettes différentes. Lorsque ces 2 contraintes d'unification sont insuffisantes, nous offrons la possibilité de programmer des fonctions booléennes sur des paramètres attachés aux nœuds du graphe. Le concepteur est libre de choisir les paramètres et les fonctions qui conviennent à son application. Dans le cadre de la gestion du partage de document, ces paramètres ont été représentés par une structure contenant un couple d'entiers et un attribut booléen indiquant les coordonnées de la zone de texte dans le document et son état (voir fig. 3).

Des formats visuels ou textuels peuvent être utilisés pour décrire une règle. Nous présentons, dans la figure 3, un exemple de représentation de chaque catégorie.

Nous présentons, dans la figure 2, les conventions de la notation que nous utilisons pour décrire visuellement de façon simplifiée une règle de transformation de graphe. La description complète tient compte des types des nœuds de leurs paramètres ainsi que des classes de comportement Java (ou corba dans d'autres exemples) qui leurs sont associées.

La notation visuelle (figure 2) décompose le graphe de la règle en trois parties correspondant respectivement :

- au motif recherché R représenté par tout le fragment se trouvant :
  - à gauche du symbole "parenthèse ouvrante", "**(**"
- à la partie de ce motif à effacer D représenté par tout le fragment se trouvant :
  - à gauche du symbole "parenthèse fermante", "**)**",
- et le motif à insérer A représenté par tout le fragment se trouvant :
  - à droite du symbole "parenthèse ouvrante", "**(**".

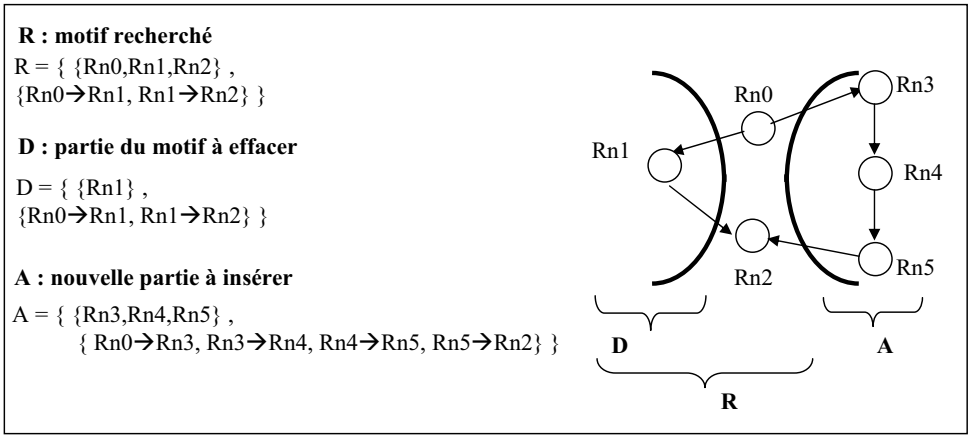

FIG. 2 – *Notation visuelle pour décrire une règle simplifiée de transformation de graphe*

D'après les lois de transformation, une règle peut être applicable plusieurs fois à plusieurs parties du graphe de coordination. En effet celui-ci peut contenir à plusieurs endroits le même motif exigé pour l'application de la règle. Lorsque l'on désire qu'une règle applicable soit appliquée à toutes ces parties, on a la possibilité, par l'ajout d'un attribut à la description de cette règle, d'exiger son application partout où il est possible de le faire dans le graphe de coordination. Ce type de règle est utilisé dans le modèle de coordination linéaire (défini pour la coordination de l'édition coopérative) pour autoriser l'évolution de l'espace non seulement en terme de nombre de zones mais aussi en terme de contenu de chaque zone. En effet, les zones étant dynamiquement définies par leurs positions les unes par rapport aux autres, la mise à jour du contenu d'une zone entraîne la modification des coordonnées des nœuds qui lui succèdent dans le graphe de coordination.

4 Application à la gestion de l'architecture dynamique des sites de coopération

Dans cette section, nous décrivons le principe de la gestion de l'architecture des sites de coopération et indirectement des applications qu'ils hébergent. Nous commençons par caractériser les applications distribuées coopératives et nous détaillons ensuite l'utilisation des graphes de coordination pour la description de l'architecture d'un site de coopération et la gestion de sa dynamique.

4.1 Le contexte général de la description des architectures logicielles

Les deux buts principaux de la description des architectures sont d'après IEEE-S2ESC Ellis et al. (1996) :

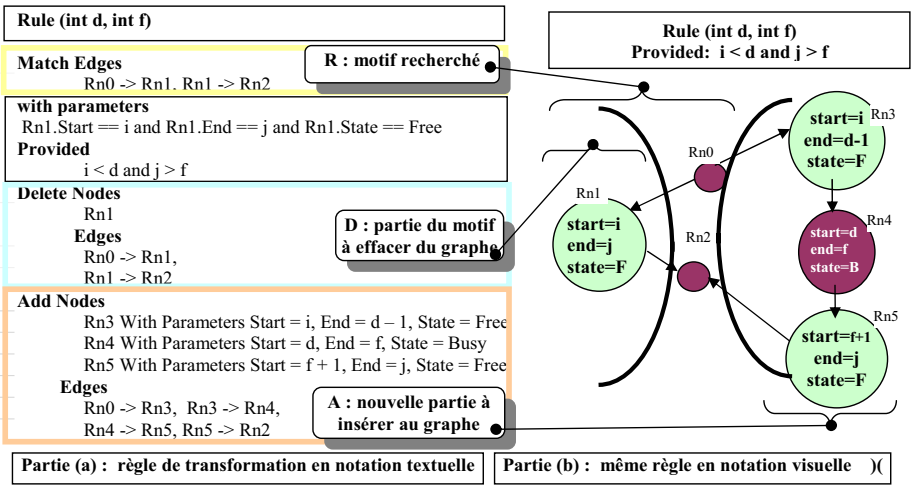

FIG. 3 – *Exemple de règle simplifiée au format textuel et visuel*

– "La description pour la conception" ("architecture as design") utilisée comme un moyen d'exprimer des caractéristiques architecturales de haut niveau du système et permettant de définir et organiser ses éléments et leurs interactions.
– "La description de type style" ("architecture as type") permettant de raisonner sur l'architecture, par exemple en terme de compatibilité, d'interopérabilité, et d'interchangeabilité de composants. Un style d'architecture est alors défini comme un ensemble de modèles ou de règles pour créer une ou plusieurs architectures de manière consistante.

Nos contributions ont couvert chacun des deux objectifs identifiés ci-dessus. Elles ont principalement concerné "la description pour la conception" que nous allons décrire dans la suite. D'autres travaux ont porté sur la "description de type style".

4.2 Le contexte des applications distribuées coopératives

Une *application distribuée coopérative* est définie comme l'intégration :

– de '*n*' *composants coopératifs* (élémentaires ou composites) symétriquement distribués sur 'n' *sites de coopération*,
– et d'un ensemble de services de communication et de coordination pouvant être :
  – regroupés sur un site (ce site peut faire partie de l'ensemble des sites de coopération ou non)
  – ou distribués (symétriquement ou non) sur l'ensemble des sites de coopération.

Considérant la symétrie de la distribution, nous associons le graphe de coordination et ses règles de transformation à la gestion de la dynamique de l'architecture logicielle interne aux sites de coopération. Un site est identifié en pratique à la machine sur laquelle travaille le participant, mais aucune corrélation n'existe entre site et machine en général : un site peut

correspondre à plusieurs machines, et une machine peut exécuter les composants associés à plusieurs sites. Des problèmes de portabilité peuvent ainsi être résolus quand c'est nécessaire.

Lorsqu'une règle de transformation est applicable et est appliquée, un ensemble d'actions est exécuté pour modifier l'architecture logicielle de chaque site. Ces actions ont lieu lors de l'application de la règle et selon la loi suivante :

- l'ajout d'un nouveau nœud au graphe de coordination entraîne la création, sur tous les sites (machine ou groupe de machines) connectés, d'une instance du composant (objet Corba ou Java dans notre implantation) qui lui a été associé lors de la déclaration de son type.
- L'effacement d'un nœud du graphe de coordination entraîne la destruction, sur tous les sites connectés, du composant qui lui a été associé. Un modèle de gestion plus affiné a été aussi défini. Il introduit des actions supplémentaires permettant l'activation et la désactivation sans détruire et recréer les composants. Aussi les conditions d'application des règles tiennent compte de l'état des composants (actif/inactif) et permettent de le modifier.
- Les composants créés sont paramétrés par une copie des paramètres (dits utilisateurs) qui ont été associés aux nœuds de coordination lors de leur déclaration. Ceci permet de concevoir une architecture symétrique tout en permettant des comportements différents de l'instance du même composant d'un site à l'autre.
- Certaines règles permettent de modifier le comportement d'un composant sur un ou plusieurs sites sans modifier le graphe de coordination. Ces règles particulières agissent sur les paramètres des composants sans effacer les nœuds correspondants. Ce type de règle est utilisé par exemple pour gérer le droit de parole dans un groupe de coopération en associant un privilège aux composants qui soutiennent les actions des participants. La circulation de ces privilèges se fait par l'application de ces règles.

4.2.1 Formalisation

Dans cette section, nous utilisons les notations qui se basent sur la structure abstraite de graphe appelée ACG (Graphe Abstrait de Composants) définie initialement dans Guennoun et al. (2003) . De cette structure sont dérivées deux autres structures de graphe par instatiation partielle ou totale des attributs des nœuds : le "graphe d'architecture" et le "graphe de règles". Ces deux structures permettent de formaliser respectivement le "graphe de coordination" et une "règle de transformation".

La structure ACG

Le graphe abstrait de composants (ACG) est une structure marquée et générique. Elle permet de définir les graphes d'architecture comme des ACG totalement instantiés, et les graphes de règles comme des ACG partiellement instantiés. La première structure décrit une architecture comme un ensemble de composants associés aux nœuds du graphe et un ensemble d'arcs dénotant les relations d'interdépendance entre ces composants.

$$\textbf{\textit{ACG}} : \mathcal{P}(Nodes) \times \mathcal{P}(Edges)$$

Dans une structure ACG, les nœuds décrivent des composants logiciels et sont marqués par les champs suivants : la classe, le comportement, l'état du composant (actif/inactif), les

facettes de son comportement, sa localisation (le site sur lequel il est exécuté), et une liste supplémentaire de paramètres concernant le niveau applicatif. Nous distinguons deux catégories de nœuds : les nœuds de règles, et les nœuds de graphes. La différence entre ces deux types de nœuds est que le premier est une abstraction d'un type de composants et peut avoir des champs variables [2], alors que le second correspond à un composant instantié de l'architecture et ne peut donc avoir que des champs totalement instantiés.

$$\textbf{\textit{Node}} : Class \times State \times Facets \times Location \times Parameters$$

Les arcs sont orientés et sont définis par le couple de nœuds qu'ils relient. Ils constituent le deuxième moyen élémentaire de la description d'architecture. Ils peuvent modéliser un large panel de relations comme les dépendances de niveau communication, ou des dépendances de niveau applicatif entre les composants.

$$\textbf{\textit{Edge}} : Node \times Node$$

Structure des règles de transformation
Nous définissons une *règle de transformation* par une partition d'un *graphe de règle* permettant de décrire les contraintes qui conditionnent l'évolution de l'architecture et les changements qui se produisent quand une règle est applicable. Au niveau supérieur de l'abstraction, une règle de transformation peut être vue comme un triplet, $RT \equiv <$Partition, constraints, Substitutions$>$, où :

- **Partition** : est une décomposition du graphe de la règle de transformation en quatre zones :
  - La zone *R :* Un fragment du graphe de la règle qui devrait être identifié (par homomorphisme) dans le graphe d'architecture. Ce fragment du graphe restera inchangé après l'application de la règle.
  - La zone *D :* Un fragment du graphe de la règle qui doit être identifié (par homomorphisme) dans le graphe de l'architecture. Le fragment du graphe qui lui a été associé par l'homomorphisme est supprimé après l'application de la règle.
  - La zone[3] *Abs :* Un fragment du graphe de la règle qui ne doit pas être identifié (par homomorphisme) dans le graphe de l'architecture pour que la règle de transformation soit applicable.
  - La zone *A :* Le fragment du graphe de la règle qui sera ajouté après l'application de la règle.
- **Constraints** : Décrit des contraintes sur les champs des nœuds. La règle n'est applicable que si toutes ses contraintes sont satisfaites par les nœuds du graphe de l'architecture qui sont unifiés avec les nœuds de la règle. Une contrainte est un couple dont le premier champ est la fonction d'évaluation de la contrainte δ (une fonction prenant en paramètre un ensemble de nœuds et renvoyant un booléen), et le deuxième est l'ensemble des nœuds qui sera évalué par δ.

$$\textbf{Const} : \mathcal{P}(\delta(\ :\mathcal{P}(\text{Nodes}) \rightarrow \text{boolean}) \times \mathcal{P}(\text{Nodes})))$$

[2] Les variables seront préfixées par le symbole "_". Par exemple, la notation <E,_x,F,Ad1> dénote un nœud de la classe E avec une facette F, situé dans le site Ad1. La variable _x indique que le nœud peut être dans un état actif ou inactif.

[3] Cette zone correspond à la "restriction", une contrainte non décrite dans les section précédentes. Elle permet d'alléger certains modèles, mais n'est pas indispensable pour la modélisation.

– **Substitutions** : Modélise les différentes substitutions que devraient subir les champs de certains nœuds du graphe après l'application de la règle. Les substitutions sont modélisées par la procédure de substitution σ qui prend en paramètres un ensemble de nœuds et permet de substituer à certains de leurs champs des nouvelles valeurs. Ceci permet de spécifier l'évolution dynamique à l'échelle du composant (migration, changement de comportement, . . .etc).

$$\textbf{\textit{Sub}} : \mathcal{P}\ (\ \sigma\ (\ :\mathcal{P}\ (Nodes) \rightarrow void) \times \mathcal{P}(Nodes))$$

Ainsi, avec les définitions précédentes, une règle possède la structure suivante :

$$\textbf{\textit{Rule}} : \underbrace{R \times D \times A \times Abs}_{Partition} \times Const \times Sub$$

Le protocole de coordination

Le protocole de coordination est en charge de gérer et de décrire l'évolution dynamique de l'architecture du système. Ce protocole manipule, le graphe courant, les règles de coordination et les événements à traiter, et associe à chaque type d'événements les règles de transformation correspondantes. Il associe, aussi, pour chaque type d'événements et pour chacune des règles lui correspondant, la procédure de transformation qui doit être appliquée à chaque règle (au niveau de son graphe, son champ *constraints*, et son champ *substitutions*) avant l'unification avec le graphe. Le protocole de coordination peut introduire aussi des événements spéciaux traduisant, par exemple, des vérifications de propriétés telles que des propriétés de *sûreté* ou de *complétude*. Ces propriétés seront décrites sous la forme d'une ou de plusieurs règles de coordination.

$$\textbf{\textit{Protocol}} : Graph \times \mathcal{P}(Rules) \times \mathcal{P}(EventType \times Trans \times \mathcal{P}(Rules))$$

L'événement décrit l'action de déclenchement menant à l'application d'un ensemble de règles de transformation. Il peut être produit par le système, ou par son environnement et est décrit comme un couple contenant le type de l'événement, et les paramètres qu'il transporte.

$$\textbf{\textit{Event}} : EventType \times EventParameters$$

Le champ *Trans* permet de répercuter les paramètres des événements sur les règles de transformation. Les règles sont ainsi instantiées en affectant des valeurs à certaines de leurs variables.

$$\textbf{\textit{Trans}} : \mathcal{P}(\alpha\ (\ :\mathcal{P}(Nodes) \times Event \rightarrow void) \times \mathcal{P}(Nodes))$$

Application des règles de transformation

Nous définissons la fonction d'unification comme une fonction récursive qui établit un homomorphisme entre la partition du graphe de la règle de transformation et le graphe de l'architecture (en tenant compte du champ *constraints* de la règle). Elle est basée sur les quatre définitions suivantes qui décrivent l'unification d'un ensemble de nœuds du graphe de règle avec un ensemble de nœuds du graphe de l'architecture.

Définition 1 (Unification de champs de nœuds)

$$\textbf{\textit{Unifiable}}(champ_i, champ_j) \equiv \begin{cases} \exists\_X \in Variables, \textbf{\textit{Tel que}}\ champ_i = \_X, \textbf{\textit{Ou}} \\ champ_i = champ_j \end{cases}$$

Définition 2 (unification de deux nœuds) *Soit,*
$N_1 = (N_1.champ_1, \ldots, N_1.champ_n)$ *un nœud d'un graphe de règle (r), et*
$N_2 = (N_2.champ_1, \ldots, N_2.champ_m)$ *un nœud d'un graphe d'architecture. Alors,*

$$\textbf{\textit{Unifiable}}(N_1, N_2) \equiv \begin{cases} (n = m), \textbf{\textit{ Et}} \\ \forall i \in [1, .., n], \textbf{\textit{Unifiable}}(N_1.champ_i, N_2.champ_i), \textbf{\textit{ Et}} \\ \textbf{\textit{Unifiable}}(N_1.successeurs, N_2.successeurs), \textbf{\textit{ Et}} \\ Const(r), \textbf{\textit{ Et}} \\ \textbf{\textit{Pas d'inconsistence dans les unifications}} \end{cases}$$

Définition 3 (Unification de deux ensembles ordonnés de nœuds) *Soit EO_1 un ensemble ordonné de nœuds d'un graphe de règle, (r), tel que $EO_1 = [N_{1,1}, \ldots, N_{1,n}]$. Soit EO_2 un ensemble ordonné de nœuds d'un graphe d'architecture tel que $EO_2 = [N_{2,1}, \ldots, N_{2,m}]$. Alors,*

$$\textbf{\textit{S\_Unifiable}}(EO_1, EO_2) \equiv \begin{cases} (n = m), \textbf{\textit{ Et}} \\ \forall i \in [1, .., n], \textbf{\textit{Unifiable}}(N_{1,i}, N_{2,i})), \textbf{\textit{ Et}} \\ Const(r), \textbf{\textit{ Et}} \\ \textbf{\textit{Pas d'inconsistence dans les unifications}} \end{cases}$$

Définition 4 (Unification de deux ensembles de nœuds) *Soit NR un ensemble de nœuds d'un graphe de règle tel que $NR = \{N_{1,1}, \ldots, N_{1,n}\}$ et NG un ensemble de nœuds de graphe d'architecture tel que $NG = \{N_{2,1}, \ldots, N_{2,m}\}$. Alors,*

$$\textbf{\textit{Unifiable}}(NR, NG) \equiv \begin{cases} n < m, \textbf{\textit{ Et}} \\ \exists \{N_{2,i1}, \ldots, N_{2,in}\} \subset \{N_{2,1}, \ldots, N_{2,m}\}, \textbf{\textit{ Tel que}} \\ \textbf{\textit{S\_Unifiable}}([N_{1,1}, \ldots, N_{1,n}], [N_{2,i1}, \ldots, N_{2,in}]) \end{cases}$$

Définition 5 (Unification d'une règle avec un graphe) *Soit r une règle et g un graphe, soit précondition(r) l'ensemble des nœuds et des arcs appartenant à ($R(r) \cup D(r)$) et soit restriction(r) l'ensemble des nœuds et des arcs appartenant à ($R(r) \cup D(r) \cup Abs(r)$), alors,*

$$\textbf{\textit{Unifiable}}(r, g) \equiv \begin{cases} \exists s_g = (\{n_0, \ldots, n_i\}, \{e_0, \ldots, e_j\}) \in g, \textbf{\textit{ Tel que}} \\ \textbf{\textit{S\_Unifiable}}(precondition(r), s_g), \textbf{\textit{ Et}} \\ Const(r), \textbf{\textit{ Et}} \\ \{(\forall s'_g = (\{n_0, \ldots, n_i, \ldots, n_{i+k}\}, \{e_0, \ldots, e_j, \ldots, e_{j+l}\}) \subset g) \\ \quad \Longrightarrow \neg (\textbf{\textit{S\_Unifiable}}(restriction(r), s'_g))\} \end{cases}$$

L'unification d'une règle r avec un graphe g suivra la démarche suivante :

- Si r n'est pas unifiable avec g alors g reste inchangé.
- Sinon : - On rajoute dans le graphe g des copies des nœuds et des arcs contenus dans le champs *A(r)*. On détruit les nœuds et les arcs du graphe qui ont été unifiés avec des nœuds et des arcs du champs *D(r)*. On modifie les champs des nœuds de g qui ont été unifiés avec les nœuds figurant dans le champs *Sub(r)*.

5 Conclusion

Nos travaux ont abouti à la conception d'un modèle formel pour la coordination, constituant une approche originale de synthèse de services de coordination. Nous avons validé et mis

en œuvre cette approche selon une architecture multi-niveaux qui étend le modèle composants et services distribués. L'architecture distingue 3 niveaux fonctionnels : (1) La communication, (2) la coordination, et (3) la coopération. Le niveau coordination implante les fonctions relatives à la coordination des sites (instantiation, activation/désactivation des objets internes aux sites) et des activités (gestion de l'espace de travail partagé) pour un accès cohérent aux objets partagés.

Un outil efficace a été développé en C++ pour la comparaison et la transformation de graphes `http://homepages.laas.fr/khalil/GTE`, et utilisé pour valider nos modèles. Son utilisation est actuellement facilité par son implantation en librairie C++ utilisable de façon interactive via une interface textuelle ou graphique, ou bien depuis un programme extérieur (Java ou C++). Une analyse de la complexité de l'algorithme et des mesures expérimentales ont démontré l'efficacité de notre implémentation.

Références

Drira, K. (2000). A coordination middleware for collaborative component-oriented distributed applications. *Special Issue on information and communication middleware. NETNOMICS (Economic Research and Electronic Networking) 2*(2000), 85–99. (Baltzer Science Publishers journal).

Ellis, W., R. Hilliard, P. Poon, D. Rayford, T. Saunders, B. Sherlund, et R. Wade (1996). Toward a recommended practice for architectural description. In *Proceedings of 2nd IEEE International Conference on Engineering of Complex Computer Systems, Montreal, Quebec, Canada, October 21-25, 1996*.

Guennoun, K., K. Drira, et M. Diaz (2003). A proved component-oriented approach for managing dynamic software architectures. In *7th IASTED International Conference on Software Engineering and Applications (SEA 2003)*, Marina del Rey, CA, USA. ACTA PRESS. ISBN 0889863946.

Malone, T. et K. Crowston (1990). What is coordination theory and how can it help design cooperative systems ? In *Proceedings of CSCW '90*, New York. ACM.

Malone, T. et K. Crwoston (1994). The interdisciplinary Study of Coordination. *ACM computing Surveys 26*(1), 87–119.

Mills, K. L. (2003). Computer-Supported Cooperative Work. In *Encyclopedia of Library and Information Sciences (2nd Edition), Marcel Dekker (eds), New York DOI : 10.1081/E-ELIS 120008706*.

Summary

The work presented addresses the problems of coordinating a set of components, which includes, in particular, the integration and distribution of these components under various architectural constraints: interdependence, dynamism and distribution. The management of the dynamics of architecture and its distributed interactions as well as its application for the support of the generic cooperative activities were two major axes within the addressed problems. This paper presents an approach based on graphs for handling these problems.

Tree Matching for Querying XML Data*

Lei NING

LIESP Lab, Department of Computer Science,
University Claude Bernard Lyon1, University of Lyon
69622 Villeurbanne Cedex
France

Abstract. Integration of multiple heterogeneous data sources continues to be a crucial problem for many application domains and a challenge for researches world-wide. With the increasing popularity of the XML model and the proliferation of XML document on-line, automated matching of XML documents and databases has become a crucial problem. We reduce the problem of answering queries against XML document to the well-known tree matching problem. In this paper we first present existing methods for querying XML data. Then we present an approach to find any matching of a query tree in the XML database without the notion of degree of relevance.
Keywords: XML, tree matching

1 Introduction

The Extensible Markup Language (XML) is rapidly emerging as the new standard for data representation and exchange on the Internet, which promulgated by W3C (Word Wide Web Consortium) on February (Bary et al., 1998). The simple, self-description nature of the XML standard promises to enable a board suite of next-generation Internet applications, ranging from intelligence Web searching and querying to electronic commerce. In many respects, XML documents are instances of semi-structured data: the underlying data model comprise an unordered, labelled tree of elements nodes (Kilpeläinen, 1992).

The proliferation of the Internet and the large acceptance of the XML standard have led to a growing number of XML documents on the Web, making the comparison of the Web to a "database" closer to reality than ever before. And like for any other database, users of this large database require fast and efficient querying to get the desired nuggets of information. Today keyword searches such as those supported by Google and Baidu are the most popular form of querying the Web. However, while these queries to indeed narrow down the contents of the Web to pertaining subset of information, they do not have the expressive power for the user to specify an exact query; and they often still require users to page through anywhere from 10 to 100 pages of results. One of the key issues in the querying of these XML documents is thus that of matching, where the query must be matched with the XML documents.

In this paper, to address this key issue, we present the existing methods for querying the XML documents and propose an approach to find any matching of a query tree.

* This work is supported by Agence Nationale de la Recherche (ANR) with the reference ANR-08-CORD-009.

The remainder of the paper is organized as follows. Section 2 presents the existing methods for querying XML. In section 3 we introduce our approach. We conclude with summary and discussion of future work in section 4.

2 XML Tree Matching Related Work

In an XML schema, both complex elements, wherein an element may nest other elements and have attributes. The match between such complex elements termed *tree matching* (Claypool et al., 2005). There are two kinds of matching: *exact tree matching*, *approximate tree matching*.

2.1 Exact tree matching

Let pattern *P* and target *T* be labelled trees of size *m* and *n* respectively, *P* matches *T* at node *v* if there exists a one-to-one mapping from the nodes of *P* into nodes of *T* (Apostolico, Galil, 1997).

In (Denis et al, 2002), the authors proposed an exact tree matching algorithm called *Path-fix* which is decomposed into two phases:

1. Build a suffix array (Manber, Mayer, 1990) database for all the trees in the database which contains strings where each string corresponds to a root-to-leaf path in a data tree.
2. Compare the root-to-leaf paths of the query tree with the paths in the suffix array database.

In the first phase, the runtime complexity is $\Theta(MN^2)$ where *M* is the number of trees in the database, *N* is the maximum number of nodes in each data tree. In the second phase, the runtime complexity is $\Theta(q^2 logS)$ where *q* is the number of nodes in the query tree, *S* is the size of a suffix array.

In (Yao, Zhang, 2004), the authors proposed an algorithm called *TreeMatch* that can directly find any matching of a tree pattern. There are two advantages of *TreeMatch*. First, *TreeMatch* does not need to decompose the query tree pattern as it matches the pattern against the data source directly. Second, the final results are compactly encoded in stacks and explicit representation of the results. The drawback of the algorithm is that the queries with self-containment (Yao, Zhang, 2004) cannot be performed.

In (Haw, Lee, 2008), the authors proposed a hybrid system, *TwigX-Guide*; an extension of the DataGuide index to support the query processing. With *TwigX-Guide*, a query can be decomposed into a set of path queries, which are evaluated individually by retrieving the path or node matches from the DataGuide index table (Goldman, Widom, 1997). Each node in the data tree is labelled with <*start*, *end*, *level*> where *start* and *end* attributes can be generated by doing a preorder traversal (Chen, 2006) of the tree and *level* is the distance of a node from a root of the tree. Using this labeling scheme, node1 is the ancestor of node2 if and only if *node1.start*<*node2.start* and *node1.end*>*node2.end*. On the other hand, DataGuide provides general path indexes that summarize all paths in the data tree starting from the root. Each label path in DataGuide is unique. DataGuide is effective to answer query with parent-child edges by matching the query against the label path directly.

2.2 Approximate Tree Matching

Approximate tree matching is the process of determining the best possible match of one tree against another where there is not much correspondence between the trees. The approximate tree matching problem is NP-hard.

In (Schlieder, 2000), the author introduced a pattern matching language *approXQL*. An *approXQL* query retrieval exact matches but find also results that are similar to the query. A sub-tree of an XML document is considered to be similar to the query if there is a sequence of basic query transformations such that the transformed query matches the sub-tree exactly. Each of basic query transformations *insertion*, *deletion*, and *renaming* has a cost; the total cost of a sequence of transformations determines the degree of similarity between the original query and the result. The complexity result of (Kilpelänen, 1992) suggests that there is no polynomial algorithm solving the approximate tree inclusion problem. An algorithm for the approximate tree inclusion problem is introduced and that is still exponential in the worst case. The algorithm is based on dynamic programming. For each query node only those data nodes are fetched that are members of potential embedding of the query tree in the data tree. The query tree is processed bottom up. For each query node *q* the embeddings of the query tree rooted at *q* are combined from the embeddings of the query trees rooted at the child nodes of *q*. Each embedding is presented by a match. The matches belonging to a certain query node are grouped by their data nodes. From each group only the minimal match is selected. The set of minimal matches belonging to the query root are the results of the algorithm. They are sorted by increasing cost and retrieved to the user.

In (Zezula et al., 2004), the authors used the *tree signatures* (Zezula et al., 2003) as the index structure and find qualifying patterns through integration of structurally consistent query path qualifications. The algorithm is decomposed into three phases:

1. Decomposition of the query tree into a set of paths;
2. Evaluation of the inclusion of the corresponding signatures in the data signature;
3. Identification of the set of answers to the unordered inclusion of the query tree in the data tree.

This approach is typically a decomposition-matching-merging process. The drawback of such process is that the size of intermediate results may be much larger than the final answers.

3 A new approximate tree matching method for XML query

In this section we present in detail our approach for querying the XML document called LAD (Level Array Database). Our approach can find any matching of a tree pattern in the XML data source directly.

3.1 The key idea

A query tree consists of a set of edges and tags. Each node in the XML data tree is labelled with <*pre, post*> where *pre* attribute can be generated by doing a pre-order traversal of the tree and sequentially assigning a number at each visit and *post* can be generated by doing a post-order traversal of the tree and sequentially assigning a number at each visit. Using this

labeling scheme, *node1* is the ancestor of *node2* if and only if *node1.pre* < *node2.pre* and *node1.post* > *node2.post*. Each node labelled is registered in LAD according to its level. We propose an algorithm to find directly any matching of the query tree in LAD.

3.2 Basic algorithm

Our approach consists of two phases: OFF-Line, ON-Line. In the first phase, we build a LAD (Level Array Database) for all trees in XML database and call this phase LAD building phase. In the second phase, we find directly all occurrences of the query nodes in LAD and the relationship between them, the results are sorted to the user.

3.2.1 OFF-Line phase

We model a collection of XML documents as a forest XML tree. Currently, we ignore ID-references and hyperlinks. To simply our model, we only use a single node type. Each node *d* of that type has a label. We use a preorder and postorder numbering of the nodes in the XML data tree. In this way, each node is labelled as (*node, preorder, postorder*). The advantage of using (*node, preorder, postorder*) to represent nodes is that we can determine the relationships between nodes in a constant time. Preorder/Postorder traversal of a tree with *n* nodes has complexity Θ (*n*) (Black, 2008). Each node labelled is registered in Level Array Database (LAD).

Level Array (LA) In order to better adapt to the most interesting properties of XML documents, we construct a new data structure called Level Array (LA). LA not only adapts to the most properties of XML document, but also to better adapts to an important property: An XML tree is always very large; its width is much greater than its height. LA is an array which contains all nodes labelled at the same level in the XML tree. Each LA is labelled with the level number.

As an example, consider the data tree D in Figure 1 each node in D is indexed with (*pre-order. postorder*). In figure 2, the integer in front of each array indicates the level position in the data tree. This integer serves as a pointer to all nodes at the same level in the data tree.

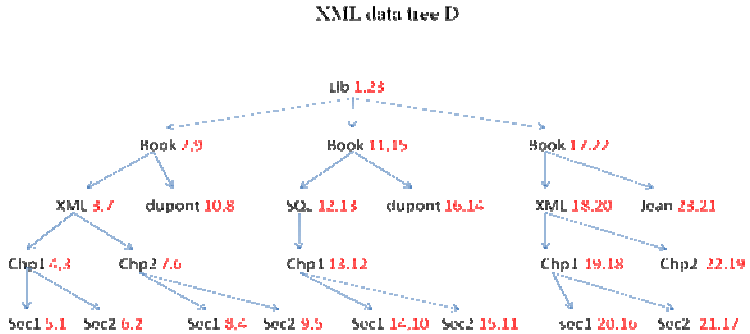

Fig. 1- *XML data tree D*

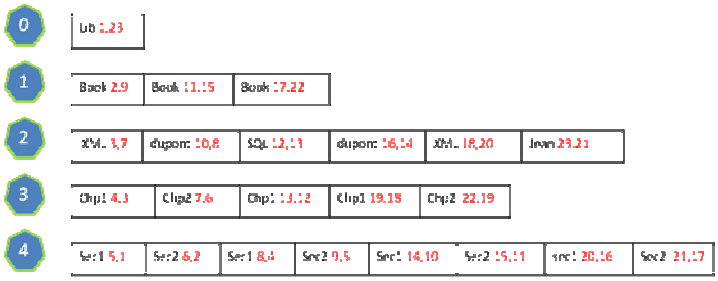

Fig. 2- *A set of level arrays*

Figure 3 shows the procedure for building the level array database LAD.

Procedure Build_Level_Array_Database
Input: the XML data tree *D* labelled by (preorder, postorder).
Output: the level array database *LAD*.

1. **for** every level *L* in the data tree *D*
2. build an empty array *LA*;
3. **for** each node *d* labelled by (preorder, postorder) in *L*;
4. add *d* to the level array corresponding *LA*;
5. **end for**;
6. **end for**;
7. return the global set of level arrays, which is the level array database *LVD*.

Fig. 3- *LAD building procedure.*

Suppose the data tree has at most *M* levels, and each level has at most *N* nodes. The complexity of the Build_Level_Array_Database is Θ(*MN*). In practice, a level has fewer nodes than *N* and M is much less than *N*, so the runtime complexity is linear.

3.2.2 ON-Line search phase

In the ON-Line search phase, the query tree *Q* is compared to each data tree in XML database. First, for each node *q* in *Q*, we find all occurrences of *q* in *D* and store them in a set of Temporary Level Array (TLA), and then we check the relationships between nodes occurrences by the properties of preorder and postorder numbering and sort the results to users according to their relevance. Temporary Level Arrays (TLAs) is simply a set of arrays containing all nodes occurrences of *Q* in *D*. The integer in front of each TLA indicates the level position. This integer serves as a pointer to all nodes occurrences at the same level in the XML data tree.

As an example, consider a user's query tree Q as shown in Figure 4 and XML data tree in Figure 2.

Fig. 4- *User's query tree Q.*

For each node *q* in *Q*, we find all occurrences and store them in TLA as shown in figure 5. For example, three occurrences of Book are found in LAD: *Book 2,9*, *Book 11,15*, *Book 17,12*, and we store the three occurrences in TLA.

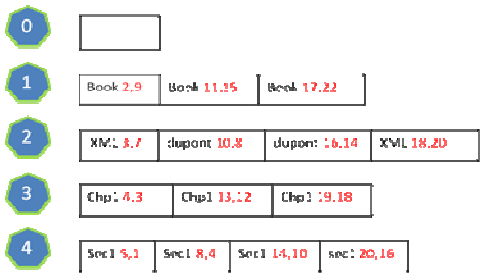

Fig.5- *TLAs construction*

In figure 5, all occurrences of the queries nodes are stored in TLAs in which we can determine immediately the vague relationships between them. Intuitively, the relationship between Book 2,9 and Book 11,15 is Left-Right; the relationship between Book 2,9 and XML 3,7 is Up-Down etc. With TLAs, a lot of calculations will be reduced. Figure 6 shows the TLAs building procedure.

Procedure Build_Temporary_Level_Arrays
Input: the user's query tree *Q*, the *LAD* of XML data tree *D*;
Output: the temporary level arrays *TLAs*.

```
1.  for each level L in LAD
2.      build an empty temporary array TLA;
3.  end for;
4.  for each node q in the query tree Q
5.      for each level L in LAD
6.          for each node d in L
7.              if (q.label == d.label)
8.                  add d to the TLA corrsponding;
9.              end if;
10.         end for;
11.     end for;
12. end for;
13. return the global set of temporary arrays, which is the TLAs.
```

Fig.6- *TLAs construction procedure*

Suppose the query tree has *P* nodes, the data tree has at most *M* levels, and each level has at most *N* nodes. The runtime complexity of the Build_Temporary_Level_Arrays is theoretically Θ(*PMN*).

In a tree, the test whether a node is an ancestor of another node can be done in constant time after a linear time preprocessing of the tree (Havalk, 1997). Thanks to the definition of ancestor (Chen, 2006), we can determine easily the relationship between any two nodes in LAD. Function Ancdestester is called to determine whether a node is the ancestor of another node. Figure 7 shows the function.

```
Function  Ancdestester
Input:  node A and node B, each node is labelled by
      (preorder,postorder);
Output:  the relationship between A and B.

1.   if  (A.preorder < B.preorder) and (A.postorder > B.postorder)
2.       A is the ancestor of B;
3.   endif;
4.   if  (A.preorder > B.preorder) and (A.postorder < B.postorder)
5.       A is the descendant of B;
6.   endif;
7.   return the relationship between A and B.
```

Fig.7 - *Relation Ancestor-Descendant test function*

In the rest of the section, we present our main algorithm for finding all approximatif matchings of a user's query tree in an XML database. Due to the TLAs, all operations can be performed direclty on this temporary arrays database. Figures 8-12 show the pseudo code of our main algorithm and procedures.

```
Algorithm LAD_treematch
Input:  the temporary level arrays T ;
Output:  all sub-trees solutions AR found in T.

1.        Current_level = 0;
2.        Index_AR = 0;      /*index of the sub-tree found in
LAD*/
3.        Pr = null;          /*current parent*/
4.        marked_selected_tester (Current_level, Index_AR, 0);
```

Fig.8- *Main algorithm*

```
Procedure marked_selected_tester
Input:   Level_current, Index_AR ;
Output:  find the first node which is neither marked nor selected.

1.  if (Level_current < Level_max_index)
2.     if (there is a node n_Level_current,col that is neither marked nor
       selected)
3.        define n_Level_current,col like the root of AR(Index_AR);
```

```
4.      Pr = n_{Level_current,col} ;
5.      find_mark_nearest_descendant(Level_current+1, Index_AR,
                                     Level_current);
6.   endif;
7.  else Level_current = Level_current+1;
8.    marked_selected_tester(Level_current, Index_AR);
9.  endif;
```

Fig.9- *Procedure marked_selected_tester*

```
Procedure find_mark_nearest_descendant
Input: Index_level, Index_AR, Level_current;
Output: visit the descendant level.

1.  if (Index_level = Level_max_index + 1)
2.    Visit_level_max(Index_AR, Level_current);
3.  else Mark_descendant_nodes (Index_level);
4.  endif;
```

Fig. 10- *Procedure find_mark_nearest_descendant*

```
Procedure Mark_desendant_nodes
Input: Index_level;
Output:  find the first descendant node and add it in AR(Index_AR).

1.    travel from Level_{Index_level} to the lastest level until
      find a node n_{i,j} that is neither marked nor selected and n_{i,j} is
the
      descendant node of Pr;
2.        select this node;
3.        add edge(Pr, n_{i,j} ) in AR(Index_AR);
4.    mark all nodes n_{i,k} in Level_i which are the descendant nodes of
      Pr and make P(n_{i,k}) = Pr;
5.    define n_{i,j} like Pr;
6.    find_mark_nearest_descendant(Index_level+1, 0, Index_AR,
                                   Level_current);
```

Fig. 11- *Procedure mark_descendant_nodes*

```
Procedure Visit_level_max
Input:  Index_AR, Level_current;
Output: add the marked nodes at the last level in AR(Index_AR).

1.        select all nodes n_{level_max,j} which are marked but not
          selected and add edge (P(n_{level_max,j} ), n_{level_max,j} ) in
          AR(Index_AR);
2.        travel from level_{max-1} to level_current until find a node
n_{i,j}
          that is marked but not selected;
3.        Add edge(P(n_{i,j} ), n_{i,j} ) in AR(Index_AR);
```

```
4.          define n_{i,j} like Pr;
5.          find_mark_nearest_descendant(i+1, Index_AR,
                                         Level_current);
6.      if there isn't a such node
7.          marked_selected_tester(0, Index_AR);
```

Fig.12- *Procedure Visit_level_max*

Given a TLA, the goal is to find all sub-trees solution in TLA. First of all, 3 variables are initialized (figure 8), the procedure marked_selected_tester (figure 9) is called to travel all levels to find the first node that is neither marked nor selected and define this node as the root of the first sub-tree solution. Then, the procedure Mark_descendant_node (figure11) is utilized to find the root's first descendant node and select this node, add this node to the sub-tree solution. At the same time, this node is defined as current parent and looking down in search of its nearest descendant node and mark all other descendant nodes at the same level of this descendant node, and so on. When the node selected is in the last level of TLA (see line 2 in figure 10), the procedure Visit_level_max (figure 12) is called to select all nodes which are marked but not selected and add these nodes the current sub-tree solution. Next, looking up a node which is marked but not selected from the penultimate level of TLA and add this node to the current sub-tree solution. Define this node as the current parent node, and so on (figure 10). Until all nodes marked are selected, the procedure marked_selected_tester (figure 9) is called to construct another sub-tree solution and find it's root, and so on.

3.2.3 Runtime Complexity

Suppose there are *D* levels and each level has at most *M* nodes. The runtime complexity of the procedure marked_selected_tester is $\Theta(D^2M^2)$.

Proof. We note that l_i is the current level in TLAs, to find a node to select, there exist two cases. The first case (there exist a node marked and no selected): if there is a descendant node on the descendant level l_r of l_1, then we need to iterate over at most $|l_r| \leq$ M nodes. If the index of current level is the max level, then we need to iterate over at most $|D\text{-}1\text{-}i| \leq$ D levels to find a node which is marked and no selected. So we need at most $\sum_{k=D-1}^{i+1} |l_k|$ loops. The second case (there isn't a node marked): we need to iterate over at most $\sum_{t=0}^{i-1} |l_t|$ nodes, so we need at most $\sum_{t=0}^{i-1} |l_t|$ loops to find a node to select.

For each node $n_{p,q}$ selected, we need iterate over at most D-l_p to find and mark its descendant nodes which are on the nearest level, so we need at most $\sum_{t=l_p+1}^{t=D-1} |l_t|$ loops. There are at most $\sum_{r=0}^{r=D-1} |l_r|$ nodes selected. So we need at most

$\left(\sum_{r=0}^{r=D-1} [[(\square)]_r] * \sum_{t=l_p+1}^{t=D-1} [[\square]_t] \right)$ loops. Therefore, the overall runtime complexity is $\Theta(D^2M^2)$.

3.2.4 Example

Consider again the user's query tree Q in the figure 4 and TLAs in the figure 5. We start with the node "Book 2,9" and find all its nearest descendant nodes in level 2,3 and 4, thanks to the preorder and postorder properties, the node XML3,7 and dupont10.8 are selected. We first consider the descendant node XML3,7 and find all nearest descendant nodes in the same way, so Chp1 4,3 is selected. The same operation is performed on the Chp1 4,3, so the node Sec1 5,1 is found. Because we have found two nearest descendants nodes of Book2,9, so we have to run the same operation on dupont10,8. There for the first solution is generated: Book 2,9 is the parent of XML 3,7 and dupont 10,8, XML3,7 is the parent of Chp1 4,3, and Sec1 5,1 is the child of Chp1 5,1. The second solution is generated by using the same way as the first solution: Book 11,15 is the ancestor of Chp1 13,12, the Sec1 14,10 is child of Chp1 13,12 and Book 11,15 is the parent of dupont16,14. The third solution is: Book 17,22 is the parent of XML 18,20, XML 18,20 is the parent of Chp1 19,18, Chp1 19,18 is the parent of Sec1 20,16. Figure 13 shows the three solutions of Q in the XML database.

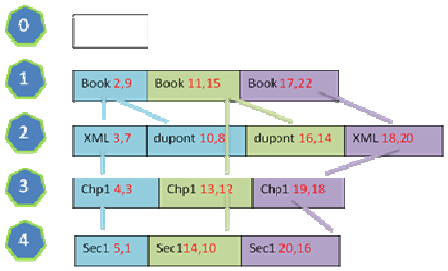

Fig.13- *A set of solutions*

We have compared the Level Array Database with the query processing algorithms of ApproXQL proposed in (Schlieder, 2000) and Treesignatures proposed in (Zezula, 2004). These three query processing algorithms can answer both query structural and content queries but have different designs leading to their performance differences. Figure 14 shows a summary of comparing the three methods. LAD performance advantage over ApproXQL and Treesignatures comes from using hash table (level array) instead of string match in a tree or in a very long string for finding all occurrences. For ApproXQL, all operations are performed in the traditional trees and a downside is that user have to learn a complex query language and understand the schema of underlying XML, in addition, the approach is more focus on the views of information retrieval without taking into account the views of database . Furthermore, the runtime complexity of the processing algorithm proposed in (Schlieder, 2000) is always exponential. The author believed that DataGuides (Haw, Lee, 2008) can be

used to accelerate the search for embeddings in the data tree. For Treesignatures , a data tree is transformed into a very long string. In this case, the number of tag and the number of tree nodes can never exceed 65536. Like ApproXQL, this approach also does not take into account the views of database. The algorithm proposed in (Zezula, 2004) for tree pattern matching in XML is typically a decomposition-matching-merging process. The drawback of the decomposition-matching-merging method is that the size of intermediate results may be much larger than the final answers. The main reason of having larger intermediate results and repeated matching of sub-patterns is due to the consideration of self-containment XML documents, i.e., an XML element that has the same tag with its sub-elements. Furthermore, the recoverability of Treesignatures is very bad. The reason is that the transformation of a tree in a string is much easier than the transformation of a string in a tree.

Criteria	**ApproXQL**	**Tree signatures**	**LAD**
match method	label node against the data tree	string match	label node against hash tables
element identification	postorder	Signature	preorder, postorder
node type	1	1	1
query type	CAS	CAS	CAS
approach type	IR+DB	IR	IR+DB
storage capacity	entire database	65,536 nodes	entire database
Recoverability	very easy	difficult	very easy
intermediate results	No	large number	No
runtime complexity	Exponential	Polynomial	Polynomial, $\Theta(D^2M^2)$.

Fig.14- *Comparison of ApproXQL, Tree signatures, LAD.*

4 Conclusion and Future work

In this paper we have studied the problem of evaluation of unordered query trees in XML tree structured data collections. We have identified two kinds of tree matching problem, where the first is exact tree matching problem and the second is approximate tree matching problem. For each problem, several approaches are presented in detail.

We have presented a new approach to directly find any matching of a query tree in XML data source. The main idea of this is store all nodes of the XML data trees in a set of level arrays called LAD and find directly all occurrences of the query nodes in LAD to build a set of temporary level arrays, finally, test the relationships among the nodes which are in TLAs by using the preorder and postorder properties. Our approach is efficient for querying the query CAS (Content and Structure) against the huge database. In addition, our approach not only supports the most properties of XML document, but also to better support an important property: An XML tree is always very large and its width is much greater than its height. To our knowledge, this is the first attempt to combine level with array in an XML context. Furthermore, the recoverability of our approach is very strong. Therefore, we can easily update our structure, i.e., add or remove nodes without influencing others.

In general, ranking of search results is a big challenge for XML searching. As future work we plan to study the similarity metric for XML documents based on sets and costs and the distance metric of clustered XML documents based on tree-edit distance. The essence of similarity for XML documents is to support the query on XML documents and find out all

candidate documents according to the similarity metric, in which we can choose the candidates that are most close to or next close to the source document and order them in turn, etc.

References

Apostolico, A., Z. Galil. (1997). *Pattern matching algorithms*. Oxford University, 377 pages.

Bary, T., J. Paoli, C.M. Sperberg-McQueen. (1998). Extensible Markup Language (XML) 1.0 Specification, w3c.org, available : http://www.w3.org/TR/1998/REC-xml-19980210.

Black, P.E. (2008). *preorder traversal*, in Dictionary of Algorithm and Data Strcutures, U.S. National Institute of Standards and Technology.

Claypool, K., T. Hegde, V. Tansalarak, N. (2005). *QMatch - A hybrid match algorithm for XML schemas*. In Data Engineering Workshops, 21st International Conference on.

Chen, Y.J., Y.B. Chen. (2006). *A new tree inclusion algorithme*. University of Winnipeg, Manitoba, Canada . Information Processing Letters 98 253-262.

Denis, S.S., J.T.L. Wang, H.Y. Shan. (2002). *AtreeGrep : Approximate Searching in Unordered Trees*. Scientific and Statistical DAtabase Management. Proceedings. 14th International Conference.

Goldman, R., J. Widom. (1997). *Data Guides: Enabling Query Formulation and Optimization in Semi-structured Databases*. Proceedings of VLDB, pp: 436-445.

Havalk, P. (1997). *Nesting of reducible and irreducible loops*. ACM Transactions on Programming Languages and Systems.

Haw, S.C., C.S. Lee. (2008). *TwigX-Guide: twig query pattern matching for XML trees*. American Journal of Applied Sciences.

Kilpeläinen, P. (1992). *Tree matching problems with applications to structured text databases*. University of Helsinki, Finnland.

Manber, U., G. Myers. (1990). *Suffix arrays: A new method for on-line string searches*. In proceedings of 1st Annual ACM-SIAM Symposium on Discrete Algorithms, pages 319-327.

Schlieder, T. (2000). *ApproXQL: Design and implementation of an approximate pattern matching language for XML*. In ACM SIGIR Workshop on XML and Information Retrieval.

Yao, J.T., M. Zhang. (2004). *A fast tree pattern matching algorithm for XML query*. Web Intelligence. Proceedings. IEEE/WIC/ACM International Conference.

Zezula, P., G. Amato, F. Debole, and F. Rabitti. (2003). *Tree signatures for XML Querying and Navigation*. In Proceedings of the XML Database Symposium, XSym 2003, Berlin.

Zezula, P., F. Mandreoli and R. Martoglia. (2004). *Tree Signatures and Unordered XML Pattern Matching*, SOFSEM 2004, 1893-1913.

Résumé. Intégration de multiples sources de données hétérogènes continue d'être un problème crucial pour les nombreux domaines d'application et un défi pour les recherches du monde entier. Avec la popularité croissante du modèle XML et la prolifération des documents XML en ligne, appariement automatisé de documents XML et bases de données est devenu un problème crucial. Nous réduisons le problème de répondre à des requêtes sur des documents XML à problème bien connu l'appariement d'arbre. Dans ce document, nous présenterons d'abord les méthodes existantes d'interrogation de données XML. Puis nous présentons une approche pour trouver tous les appariements distincts d'une requête sans la notion de degré de pertinence.

A Demonstration of an Efficient Tool for Graph Matching and Transformation

Khalil Drira *,**, Ismael Bouassida Rodriguez *,**

*CNRS ; LAAS ; 7 Avenue du Colonel Roche, F-31077 Toulouse, France
**Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
khalil@laas.fr
http://homepages.laas.fr/khalil/page

Résumé. Nous avons implanté un outil efficace de recherche de morphismes et de transformation de grands graphes et l'avons utilisé pour modéliser les architectures dynamiques, les activités coopératives et d'autres applications. L'outil est en cours d'extension avec des interfaces graphiques et un module d'interopérabilité en XML. Une version préliminaire est disponible sous `http://homepages.laas.fr/khalil/GTE/`

1 Description

GTE, the graph matching and transformation engine is an efficient tool we have been implementing in C++ since a decade now. It is an efficient implementation of an extension of Messmer's algorithm (see Messmer (1995)). Our experiments presented in Bouassida-Rodriguez et al. (2008) show that the tool is capable of searching small and medium graph patterns in huge graphs in a short time. A computational complexity analysis of our algorithm has conducted and performant experimental results are obtained. We have also shown that, when only constant labels are considered, this complexity is similar to the complexity of Ullmann's algorithm (see Ullmann (1976)). Both pattern graph (called rule graph) and host graph have labelled nodes and edges. The rule graph labels may be totally or partially instatiated. Unification is conducted for non-instantiated labels.

The tool can be used non-interactively as a C++ library providing a function that can be invoked from either a C++ or a Java main program. The tool can be used through as a C++ executable that reads rule graph and host graph description from input TXT or XML files. It has been recently associated with a graphical user interface (Figure 1) composed of the following zones and components :

- A menu bar offering to the user many items to manipulate the interface contents such as creating, deleting, saving projects, graphs and rules.
- A tools bar that the user can use to edit graphs and rules (saving, undo, redo...).
- Project Explorer giving the user a tree representing the list of opened projects, graphs and rules.
- A components panel containing a list of buttons for creating nodes and edges.
- A graph representing zone offering to the user the possibility to open and show graphs she/he is manipulating.

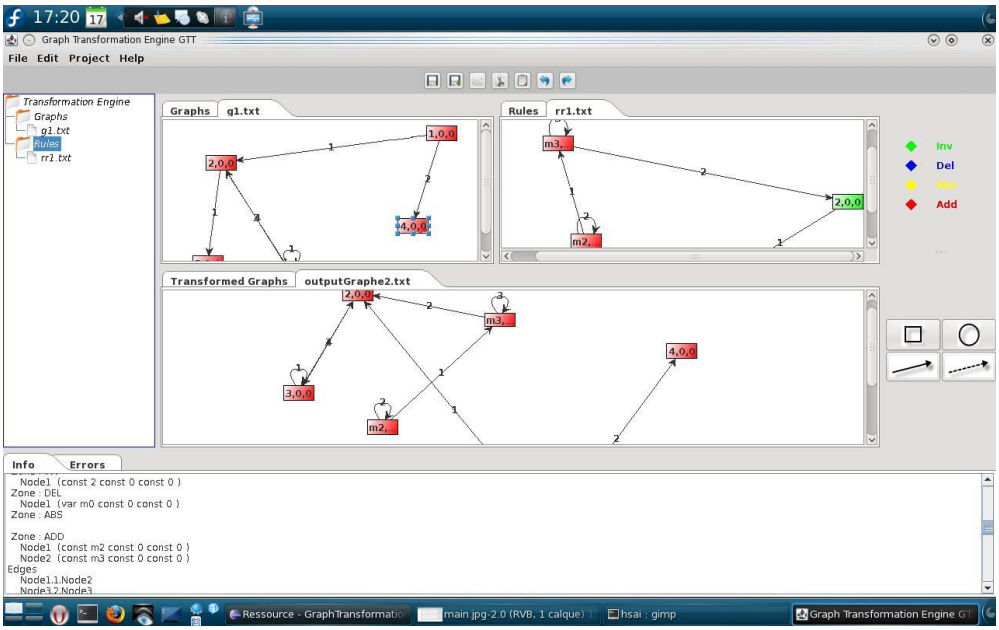

FIG. 1 – *The Graphical User Interface*

– A rule representing zone offering to the user the possibility to open and show rules she/he is manipulating.
– A transformed graph zones offering to the user the possibility to open and show graphs she/he had transformed.
– A rule legend with which the user can distinguish between rule zones *(Inv, Del, Abs, Add)*.
– Two tabs showing to the user information and errors when transforming a graph.

The user can export graphs and rules from the application to TXT or XML according to the standard RuleML format as well as an image. The interface offers an export wizard which gives the user the possibility to specify file name, directory where to export and the export format. The exported XML graph file an example of which is shown in Figure 2 is composed of a Graph element containing a list of nodes and edges Elements with different attributes describing each element. The exported XML rule file is composed of a Rule element containing a list of nodes representing the different zones of the rule *(Inv, Del, Abs, Add)*. Each zone element is composed of a graph containing a list of node and edge Elements with different attributes describing each one.

2 Acknowledgment

Have contributed to this work : Ismael Bouassida Rodriguez, Khalil Drira, Rony Ghostine, Karim Guennoun, Moncef Sadoq and Hachemi Sai.

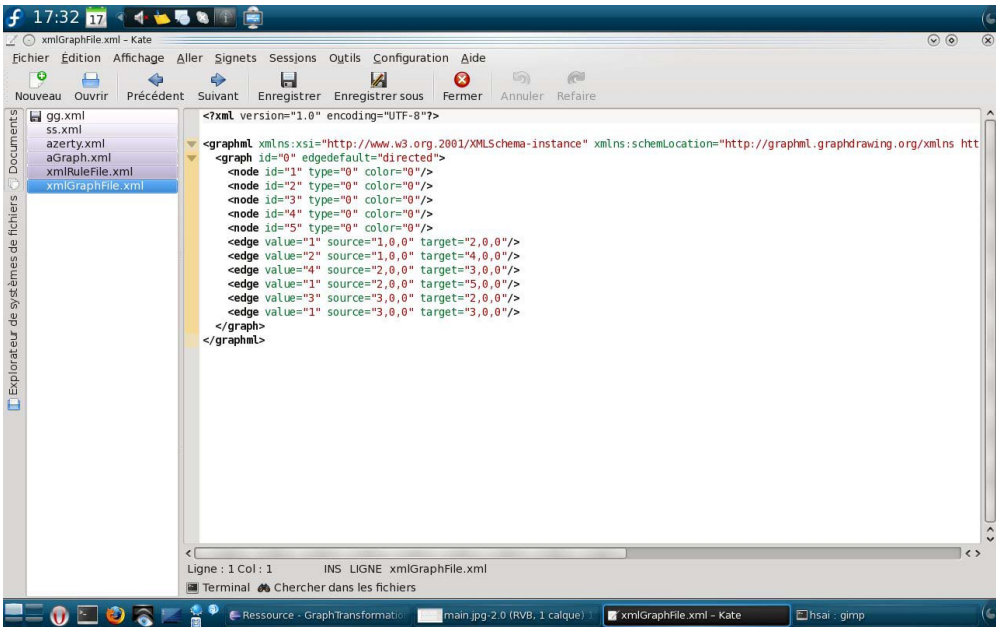

FIG. 2 – *The XML window*

Références

Bouassida-Rodriguez, I., K. Guennoun, K. Drira, C. Chassot, et M. Jmaiel (2008). Implementing a rule-driven approach for architectural self configuration in collaborative activities using a graph rewriting formalism. In *CSTST '08 : Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, New York, NY, USA, pp. 484–491. ACM.

Messmer, B. (1995). *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*. Ph. D. thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, Switzerland.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM 23*(1), 31–42.

Summary

We implemented an efficient tool for graph matching and transformation. We have used it to model dynamic architectures, co-operative activities and other applications. The tool is being extended with graphical user interfaces and XML interoperability. A preliminary version is available under `http://homepages.laas.fr/khalil/GTE/`