木直列化に基づくXMLデータの類似結合における木構造の統合

寺島慎太郎† 天笠 俊之†† 北川 博之††

† 筑波大学第三学群情報学類
†† 筑波大学大学院システム情報工学研究科
E-mail: †{shintaro,amagasa,kitagawa}@kde.cs.tsukuba.ac.jp

あらまし 近年 XML フォーマットが爆発的に普及し XML データに対する情報統合が重要な課題となっている。内容は類似しているが構造が異なる二つの XML データに対する類似結合手法の一つとして、XML データを直列化し、配列として扱う手法が研究されている。この手法は、XML データを木構造のまま扱うよりも計算時間が短くて済むという利点がある。しかしながら、既存の研究では木構造の統合までは考慮していなかった。そこで本稿では、木直列化した XML データ群に対し、その構造を考慮して統合する手法を提案する。 二つの直列化した XML データに対し、双方で一致しているノード・部分木の情報を付加し、その情報を元に類似結合を行う。さらに実験によって、実用的な計算時間で構造を考慮した統合が行われている事を確認する。
キーワード XML, 情報統合, 木直列化

Integrating XML trees in similarity join based on tree serialization

Shintaro TERAJIMA†, Toshiyuki AMAGASA††, and Hiroyuki KITAGAWA††

† College of Information Science, University of Tsukuba, Tennodai 1–1–1 Tsukuba-shi, Ibaragi 305–8573, Japan
†† Graduate School of Systems and Information Engineering, University of Tsukuba, Tennodai 1–1–1 Tsukuba-shi, Ibaragi 305–8573, Japan
E-mail: †{shintaro,amagasa,kitagawa}@kde.cs.tsukuba.ac.jp

Abstract XML has become a de facto standard format for data representation and exchange, and information integration techniques for XML are therefore becoming important to utilize XML data resource. Similarity join methods are used to integrate a pair of XML trees. However, computing the similarity between two trees is computationally intensive. So, a method based on tree serialization has been proposed; it can reduce execution time for matching XML trees compared to methods using tree structure information. However, the paper does not discuss how to integrate tree structures. This paper proposes a method for integrating XML trees. We add information of nodes and sub-trees corresponding with each other to the serialized XML data, and try to integrate the hierarchical structure using those information. We experimentally show that the proposed method is feasible enough to perform the integration process.
Key words XML, information integration, tree serialization

1. はじめに

XML が W3C で勧告されて以来、XML 形式のデータが様々な分野で使用されるようになった。その普及の早さは留まるところを知らず、今や多種多様な XML 形式のデータを Web 上から入手できるまでになっている。

しかしその急速な普及に伴い、表現する内容は同じであるにも関わらず、異なる構造をもつ XML データ群が多く存在する様になった。例えば Web 上に公開されている XML データ「DBLP Bibliography」と「ACM SIGMOD」は、類似した内容を表している。それに加え双方とも、もう一方が持っていない特別な情報を所有している。

図 1 は「ACM SIGMOD」(左)「DBLP Bibliography」(右)からそれぞれ引用した XML データである。タイトルと著者を見ると分かるように、この二つの XML は同じ論文を表している。しかし、前者と後者で著者とタイトルの順番が逆だったり、前者には無い URL の情報が後者にはあったりと、その構造は異なっている。

<article>
<title>
Call for Book Reviews
</title>
<authors>
<author>
Karl Aberer
</author>
</authors>
</article>

ACM SIGMOD

<article>
<author>
Karl Aberer
</author>
<title>
Call for Book Reviews.
</title>
<pages>81</pages>
<year>2003</year>
<volume>32</volume>
<journal>
SIGMOD Record
</journal>
<number>1</number>
<ee>
http://www.acm.org...
</ee>
<url>
db/journals/sigmod...
</url>
</article>

DBLP Bibliography

図 1 「ACM SIGMOD」「DBLP Bibliography」内の XML

異なる情報元からより多くの情報を引き出すには、このような XML を統合することが重要である。このようなデータ群を統合するために、XML の分野で類似結合の技術が注目されている。XML はマークアップにより記述された構造を有するので、通常の文書とは異なり、その文書の構造を考慮しなくてはならない。構造を考慮した類似結合の一般的な手法として、比較する部分木同士の編集距離を類似度として使用する方法が既に研究されている [1]（編集距離とは、一方の文字列や木構造をもう一方の文字列や木構造と同じ形にするのに要する、最少の編集回数である）。しかし木構造の編集距離の計算は大変時間がかかることが分かっている。具体的には、木のノード数を n とすると、$O(n^4)$ もの計算時間がかかってしまう。そのため、巨大な XML を扱うのには向いていない。

この問題を解決するため、比較する部分木同士を直列化し、文

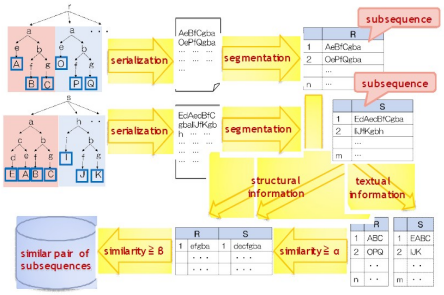

図 2 木直列化を用いた類似結合手法の概要

字列同士の編集距離を類似度として使用する方法が研究されている [2]。図 2 はその概要を表したものである。まず統合する部分木を NoK パターン（2 節で説明する）に直列化する。その際、テキストノードのリストも作成する。次にテキストノードに対する類似度（テキスト類似度）を計算する。テキスト類似度は、双方の全テキストノードの数と、双方で共通するテキストノードの数との比となる。その後、テキスト類似度が閾値を超えた部分木ペアに対し、直列化した文字列に対する類似度（構造類似度）を計算する。構造類似度は、直列化した文字列に対する編集距離となる。最後に、構造類似度が閾値を超えた部分木ペアを抽出して保存する。この類似度はあくまで概算であり、正確さでは直接木構造を扱う場合より劣ってしまう。しかし木構造を文字列で表す事で、木構造をそのまま扱うよりも、類似度の計算時間が大幅に短縮される。そのため、巨大な XML でも現実的な計算時間で類似結合が可能となる。

Wen らの手法で、類似する部分木同士を見つけ出すことが可能である。しかし類似する部分木を実際に統合する所まではまだ未検討である。そこで本研究では、この木直列化を行って出来た文字列を利用し、木構造を考慮して統合する手法を提案する。類似度が閾値を超えた部分木のペアに対し、その構造を考慮して統合を行う事で、XML データに対する情報統合を実現する。

本稿の構成は以下の通りである。2 節で NoK パターンについて述べ、3 節で XML の差分の表現について説明する。4 節で本手法について述べる。5 節で評価実験とその考察についてを説明し、最後に 6 節で本稿をまとめる。

2. XML 木の直列化

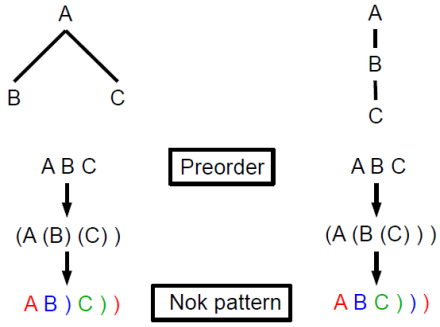

図 3 NoK パターンの例

Wen らの手法では、XML の木構造を構造を維持したまま直列化する手法として、NoK パターン [3] を利用している。図 3 は、NoK パターンの例である。

NoK パターンを利用した直列化手法は次の通りである。まず XML 木を前置順に走査していき、ラベルを取得していく。この際、部分木を「()」で囲う。「(」は部分木の始まりを表し、「)」は部分木の終わりを意味する。木全体を走査した後、出来た文字列から「(」を取り除く。各々のノード（『)』以外の文字）が部分木の始まりの意味を含むため、「(」を除いても木構造の表現には影響が無く、各々のノードと「)」は一対一対応するようになる。あるノードに対応する「)」は、左にあって高さが等しい一番近い「)」となる。例えば図 3 の左側の NoK パターンの場合、ノード”A”と最後の「)」、ノード”B”と最初の「)」、ノード”C”と 2 番目の「)」が対応する。

また NoK パターンを利用すると、文字列上の各文字と木の高さを容易に対応付けることができる。「)」が連続すると木の高さが上がり、ノードが連続すると木の高さが下がる。「)」とノードが交互に来る場合は、木の高さは変化しない。

3. XML の差分の表現

XML 木の統合を行う際、双方で一致する情報はそのまま残し、双方の一致しない情報だけを統合すると、無駄のない統合木を作ることができる。そのためには、まず比較する XML 木の一致部分、不一致部分を判定する必要がある。

比較する XML 木の差分を表す手法が、Lindholm らによって提案されている [4]。比較する二つの XML 木を直列化し、そこに一致する部分木・ノードの情報を付加する。直列化した XML 木をシーケンスと呼ぶ。シーケンス s は n 個の要素（トークンと呼ぶ）からなるリスト、$s = \langle s^0, ..., s^{n-1} \rangle$ として表される。また、シーケンスの一部 $\langle s^i, ..., s^j \rangle$ $(i < n \leqq j)$ を $s^{i...j}$ と表し、サブシーケンスと呼ぶ。Lindholm らの手法では、直列化

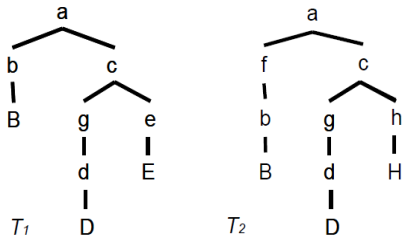

図 4　直列化する XML 木

する際に XAS [5] という、SAX に似た JAVA API を用いている。トークンは XML の開始タグや終了タグ等の XML の要素に相当し、シーケンスはその集合となる。例えば、図 4 の XML 木 T_1, T_2 を XAS を用いて直列化した場合、それぞれの シーケンス s_1, s_2 は以下のようになる。

$$s_1 = \langle SE(a), SE(b), SE(B), EE(B), EE(b), SE(c), SE(g), SE(d), SE(D), EE(D), EE(d), EE(g), SE(e), SE(E), EE(E), EE(e), EE(c), EE(a) \rangle$$

$$s_2 = \langle SE(a), SE(f), SE(b), SE(B), EE(B), EE(b), EE(f), SE(c), SE(g), SE(d), SE(D), EE(D), EE(d), EE(g), SE(h), SE(H), EE(H), EE(h), EE(c), EE(a) \rangle$$

SE は開始タグ、EE は終了タグを指す。

ここで、二つのサブシーケンス $s_1^{i...j}, s_2^{k+...k+(j-i)}$ が一致している場合、$cpy(i, j)$ と表すものとする。双方のシーケンス内で一致しているサブシーケンスを、全て $cpy()$ で置き換えたものをマッチリストと呼ぶ。例えば、図 4 の XML 木 T_1, T_2 のマッチリスト u, v は以下のようになる。

$$u = \langle cpy(0,0), cpy(2,5), cpy(7,7), cpy(8,13), SE(e), SE(E), EE(E), EE(e), cpy(18,18), cpy(19,19) \rangle$$

$$v = \langle cpy(0,0), SE(f), cpy(1,4), EE(f), cpy(5,5), cpy(6,11), SE(h), SE(H), EE(H), EE(h), cpy(16,16), cpy(17,17) \rangle$$

ノード *a, c*、及び部分木 *b*〜*B, g*〜*d*〜*D* が一致しているので、s_1, s_2 内で該当するサブシーケンスを全て置き換えている。

4. 提 案 手 法

本稿で提案する木構造の統合手法は、大きく二つの手順に分けられる。

（1） 直列化した XML 木に対しマッチリストを作成する

（2） そのマッチリストを元に、木構造を考慮して文字列の統合を行う

手順 (1) の説明を 4.1 節、手順 (2) の説明を 4.2 節で行う。

4.1 マッチリストの作成

本稿では、NoK パターンで直列化した文字列に対するマッチリストの作成手法を三つ提案する。それぞれ 4.1.1 節、4.1.2 節、4.1.3 節で説明する。なお、統合するシーケンスを $s_1 = \langle s_1^0, ..., s_1^{n-1} \rangle$,$s_2 = \langle s_2^0, ..., s_2^{m-1} \rangle$、そのマッチリストを $u = \langle u^0, u^1, ..., u^{n-1} \rangle$, $v = \langle v^0, v^1, ..., u^{m-1} \rangle$ と表す。さらに「)」のトークンを「括弧トークン」（あるいは単に『括弧』）、それ以外の文字のトークンを「文字トークン」（あるいは単に『文字』）と呼ぶ。

4.1.1 提案手法 1：XAS

NoK パターンで直列化した文字列を、2 節で説明した XAS で直列化した文字列に置き換えた後、[4] で提案されている手法を用いてマッチリストを作成する。また一致しているサブシーケンスを発見する際、シーケンス全体を大きなサブシーケンスに分け、そのサブシーケンス単位で探索していく方法をとる。この方法をとる事で、一つ一つトークンを探索するよりも効率的にマッチリストを作ることができる。手法の詳細を表 1 に示す。

表 1　手法 1：XAS

1. **For** $i = 0, 1, ...n(m) - 1$ **, if** $s^i_{1(2)}$ = 文字 **:**
   $s^i_{1(2)} = SE(s^i_{1(2)})$
   $s^i_{1(2)}$に対応する括弧 $= EE(s^j_{1(2)})$
2. $s = 32, u = s_1, v = s_2$
   while $s > 0$ **do:**
   For each $u^{ps...(p+1)s}$ $(p = 0, 1...)$ **do:**
   For $i \to 0, 1..., m-1$**, if** $u^{ps...(p+1)s} = v^{i...i+s}$ **:**
   while $u^{(p+1)s+j} = v^{i+s+j}$ **,** $j = j + 1$
   $u^{ps...(p+1)s+j}(v^{i...i+s+j})$ を
   $cpy(i, i+s+j)(cpy(ps, (p+1)s+j))$ に置き換え
   $s = s/2$
3. **For** $i \to 0, 1, ..., n(m)-1$ **, if** $u(v)^i = SE(EE)$ かつ $SE(EE)$ と対応する $EE(SE)$ が同じ $cpy()$ 内に無い**:**
   if $EE(SS)$ が $cpy()$ 内に無い**:** $u(v)^i$ を $cpy()$ からはずす
   else : $u(v)^i$ を $cpy(i, i)$ に置き換え

1. で NoK パターンを XAS に置き換える。2. では一致しているサブシーケンスを探す。その時 u は s 個のトークンで構成されたサブシーケンス毎に比較する（s は段々小さくしていく）。一致しているサブシーケンスが見つかったら、双方のサブシーケンスの隣のトークンを比較する。一致していたらそのトークンをサブシーケンスに入れて拡大し、これを繰り返す。3. では 2. で作られたマッチリストを木構造に対応した形にす

る。*cpy*() 内には一つのノードまたは部分木だけが入るようにし、半端なものは分断する。

図 5 は図 4 の XML 木に表 1 の手法を適用した例である。

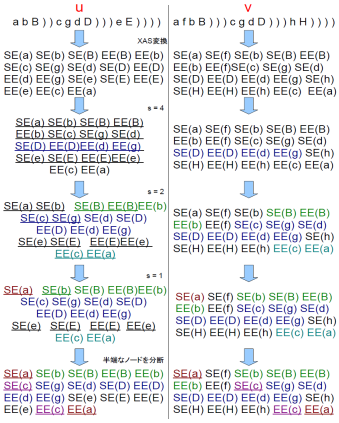

図 5　手法 1：XAS の例

この図はマッチリスト u, v の動きを示しており、*cpy*() に入ったサブシーケンスに色をつけ、（最初に）探索する s 個のサブシーケンスに下線を引いている。

一致しているサブシーケンスの長さは最長でも 6 なので、$s = 8$ までは変化はない。$s = 4$ の時、$u^{8\ldots11}$ と $v^{10\ldots13}$ が一致する。$s = 2$ の時に $u^{2\ldots3}$ と $v^{3\ldots4}$ が一致し u^4 と v^5 も一致するので、$u^{2\ldots4}$ と $v^{3\ldots5}$ を *cpy*() に置換する。また $u^{5\ldots6}$ と $v^{7\ldots8}$ と u^7 と v^9 が一致し、次の u^8 と v^{10} が *cpy*() 内にあって既に一致することが分かっているので、$u^{5\ldots11}$ と $v^{7\ldots13}$ を *cpy*() に置換する。同様に $u^{16\ldots17}$ と $v^{18\ldots19}$、$s = 1$ の時に u^0 と v^0、$u^{1\ldots4}$ と $v^{2\ldots5}$ を *cpy*() に置換する。最後に u^0 と u^{17}、u^5 と u^{16}、v^0 と v^{19}、v^7 と v^{18} が別々の *cpy*() 内にあるので分断する。

4.1.2　提案手法 2：ボトムアップ

NoK パターンで直列化したものを再び XAS に変換するのは非効率的である。手法 2 と手法 3 は文字列を XAS に変換せず、NoK パターンのままマッチリストを作成する。また手法 2 では一致している部分木を探索する際、まず一致するリーフノードを探し、そこから徐々に探索範囲を広げる方法をとる。この手法を用いることで、リーフノードが一致している部分木のみを効率的に発見できることが期待される。手法の詳細を表 2 に示す。

1. では一致する部分木（テキストノードを含む）を探索する。u と v で一致する「連続する文字トークン、同数の連続する括弧トークン」の並びのサブシーケンスを探索する（但し *cpy*() 内のトークンは無視する）。2. では一致する個別ノード（一致しない子孫ノードが存在する）を探索する。まだ *cpy*() に入っておらず、かつ一致する文字トークンを、対応する括弧トークンと共に *cpy*() に入れる。

図 6 は図 4 の XML 木に表 2 の手法を適用した例である。まず u^2 と v^3 が文字トークン、u^3 と v^4 が括弧トークンで一致する。さらに u^1 と v^2 が文字トークン、u^4 と v^5 が括弧トークンで一致するので、$u^{1\ldots4}$ と $v^{2\ldots5}$ を *cpy*() に入れる。同様

表 2　手法 2：ボトムアップ

1. $u = s_1, v = s_2$
   For $i \to 0, 1..., n-2$ **,** $j \to 0, 1..., m-2$ **, if** $u^i = v^j =$ 文字 かつ $u^{i+1} = v^{j+1} =$ 括弧**:**
   while $v^{j-k} = u^{i-k}$ かつ $v^{j+1+k} = u^{i+1+k}$ **,** $k = k+1$
   （但し *cpy*() 内のトークンは無視）
   $u^{i-k\ldots i+k+1}(v^{j-k\ldots j+k+1})$ を
   $cpy(i-k, i+k+1)(cpy(j-k, j+k+1))$ に置き換え
2. **For** $i \to 0, 1..., n-1$ **,** $j \to 0, 1..., m-1$**, if** $u^i = v^j = cpy()$ 内に無い文字**:**
   $u(v)^{a(b)} = u(v)^{i(j)}$に対応する括弧
   $u^{i(a)}(v^{j(b)})$ を $cpy(j(b), j(b))(cpy(i(a), i(a)))$ に置き換え

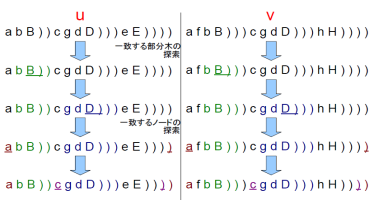

図 6　手法 2：ボトムアップの例

に u^8 と v^{10}、u^7 と v^9、u^6 と v^8 が文字トークン、u^9 と v^{11}、u^{10} と v^{12}、u^{11} と v^{13} が括弧トークンで一致するので、$u^{6\ldots11}$ と $v^{8\ldots13}$ を *cpy*() に入れる。最後に u^0 と v^0、u^5 と v^7 が文字トークン一致するので、対応する括弧トークン u^{17},v^{19},u^{16},v^{18} ごと *cpy*() に入れる。

4.1.3　提案手法 3：トップダウン

手法 2 ではまず一致するリーフノードを探し、そこから徐々に探索範囲を広げたが、手法 3 ではまず最大の部分木を探索し、そこから徐々に探索範囲を狭める。この手法を用いることで、大きな部分木を早く発見できることが期待される。手法の詳細を表 3 に示す。

表 3　手法 3：トップダウン

1. $u = s_1, v = s_2$
   For $i \to 0, 1, ..., n-1$ **,** $j = 0, 1, ..., m-1$ **, if** $u^i = v^j =$ *cpy*() 内に無い文字 **:**
   $u(v)^{a(b)} = u(v)^{i(j)}$に対応する括弧
   if $u^{i\ldots a} = v^{j\ldots b}$ **:** $u^{i\ldots a}(v^{j\ldots b})$ を $cpy(i, a)(cpy(j, b))$ に置き換え
2. 表 2 の 2. と同じ

1. では一致する部分木を探索する。u の文字トークンから対応する括弧トークンまでのサブシーケンスが v のサブシーケンスと一致していたら *cpy*() に入れる。2. では一致する個別ノードを探索する。手法 2 と同じ過程なので説明は省略する。

図 7 は図 4 の XML 木に表 3 の手法を適用した例である。まず、文字トークン u^0 から対応する括弧トークン u^{17} までのサブシーケンス $u^{0\ldots17}$ を探索する。しかし v に一致するサブシーケンスが存在しないので、次の文字トークン u^1 を含むサブシーケンス $u^{1\ldots4}$ を探索する。これは $v^{2\ldots5}$ に一致するので、双方とも *cpy*() に入れる。同様に、u^5 を含む $u^{5\ldots16}$ はどこにも一致

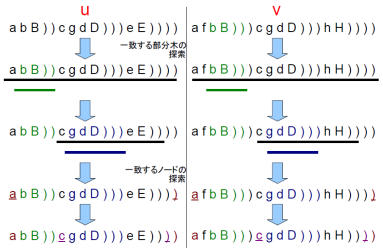

図 7　手法 3：トップダウンの例

せず、u^6 を含む $u^{6\ldots11}$ が $v^{8\ldots13}$ と一致する。$u^{12\ldots15}, u^{13\ldots14}$ はどこにも一致しない。最後に u^0 と v^0、u^5 と v^7 が文字トークン一致するので、対応する括弧トークン u^{17},v^{19},u^{16},v^{18} ごと $cpy()$ に入れる。

4.2　マッチリストを使った木構造の統合

4.1 節で作成したマッチリストを使って、木構造の統合を直列化した文字列の統合として行う。

4.2.1　統合木の構造

木構造の統合を行った木（統合木）の構造は、統合に用いる手法に応じて特徴が出てくる。本研究の手法で作成する統合木は、以下の条件を満足する。なお、統合する前の木を t_1, t_2（挿入される方の木が t_1）、統合木を t_{uni} と表す。

（1）　t_1, t_2 のどちらかに存在する任意のノードは、t_{uni} のいずれかのノードと一致する。

（2）　t_1 の 2 つの任意の親子 (兄弟) ノードを t_1^a, t_1^b、それらと一致する t_{uni} のノードを t_{uni}^a, t_{uni}^b とする。t_1^a が t_1^b の親 (t_1^a が t_1^b より左にある) ならば、t_{uni}^a は t_{uni}^b の親となる (t_{uni}^a は t_{uni}^b より左にある)。

（3）　t_2 の 2 つの任意の親子 (兄弟) ノードを t_2^a, t_2^b、それらと一致する t_{uni} のノードを t_{uni}^a, t_{uni}^b とする。t_{uni}^b と一致するノードが t_1 に存在せず、かつ t_2^a が t_2^b の親 (t_2^a が t_2^b より左にある) ならば、t_{uni}^a は t_{uni}^b の親となる (t_{uni}^a は t_{uni}^b より左にある)。

（4）　t_{uni} の 2 つの任意の兄弟ノードを t_{uni}^a, t_{uni}^b とする。t_{uni}^a と一致するノードが t_2 にのみ存在し、t_{uni}^b と一致するノードが t_1 にのみ存在するならば、t_{uni}^a は t_{uni}^b より左にある。

（5）　t_{uni} の 2 つの任意の部分木を t_{uni}^A, t_{uni}^B とする。t_{uni}^A の先祖ノードの集合を ANC^a、t_{uni}^B の先祖ノードの集合を ANC^b とすると、($t_{uni}^A \neq t_{uni}^B$)　または　($t_{uni}^A = t_{uni}^B$ かつ $ANC^a \nsubseteq ANC^b$ かつ $ANC^a \nsupseteq ANC^b$)

(2)、(3) は統合前の木の親子関係・兄弟関係が、統合木でも維持される事を示す。但し双方の木で親子・兄弟関係が異なる場合、挿入される方の木の構造が優先される。(5) は統合木内で同じ構造を持つ部分木は、その先祖が包含関係に無い事を示す。

4.2.2　提 案 手 法

4.2.1 節の条件を満たした統合木を作成する手法を提案する。手法の詳細を表 4、表 5 で示す。なお、統合に用いるマッチリストを $u = \langle u^0, u^1, ..., u^{n-1} \rangle$,　$v = \langle v^0, v^1, ..., v^{m-1} \rangle$、統合木のシーケンスを $T = \langle T^0, T^1, ..., T^o \rangle$ と表す。また $u(v)(T)$ の各トークンの高さを $h^i_{u(v)(T)}$　$(i = 0, 1...n(m)(o) - 1)$ と表す。

表 4　木構造の統合手法

1. $T = u$
   For $j \to 0, 1, ..., m-1$ **, if** $v^j =$ 文字 **:**
   $v^b = v^j$に対応する括弧
   For $i \to 0, 1, ..., n-1$ **, if** $u^i = v^j$ かつ $h^i_u = h^j_v$ **:**
   $u^a = u^i$に対応する括弧
   if $u^i = v^j =$ 一文字分のみの $cpy()$ **:** 入力を $u^{i\ldots a}, v^{j\ldots b}$ にして 2 行目に戻る
   else if $u^i = v^j =$ 二文字分以上の $cpy()$ の一部 かつ $u^{i\ldots a} = v^{j\ldots b}$ **:** 何もせずに 2 行目に戻る
   $T =$ サブシーケンスの挿入（j, b, u, v, T）
2. **For** $i, j \to 0, 1, ..., o-1$ **, if** $T^i = T^j =$ 文字 **:**
   $T^{a(b)} = T^{i(j)}$に対応する括弧
   if $T^{i\ldots a} = T^{j\ldots b}$ **:**
   $H = h_T^{i(j)}$, $ANC_{i(j)} = T^{i\ldots a}(T^{j\ldots b})$ の先祖ノードの集合
   For $k \to i(j), i(j)-1, ..., 0$ **, if** $T_k =$ 文字 かつ $h_T^k < H$ **:**
   $ANC_{i(j)}$に T_kを追加
   $H = h_T^k$
   if $ANC_{i(j)} \subseteqq ANC_{j(i)}$ **:** $T^{i\ldots a}(T^{j\ldots b})$ を削除

表 5　関数：サブシーケンスの挿入

入力:	挿入するサブシーケンスの始点と終点 S, E　マッチリスト u, v 挿入前のの統合木の NoK パターンシーケンス T
出力:	挿入後の統合木の NoK パターンシーケンス T

1. **For** $i \to 0, 1, ..., n-1$ **,** $j = S, S-1, ..., 0$ **, if** $u^i = v^j =$ 文字 かつ $h_v^S = h_v^j = h_u^i$**:**
   $u(v)^{a(b)} = u(v)^{i(j)}$に対応する括弧
   if $u^i = v^j =$ 一文字分のみの $cpy()$ または ($u^i = v^j =$ 二文字分以上の $cpy()$ の一部 かつ $u^{i\ldots a} = v^{j\ldots b}$)
   $v^{S\ldots E}$ を T の u^a に相当する位置に挿入して T を返す
   $v^{S\ldots E}$ を T の u^1 に相当する位置に挿入して T を返す

表 4 の 1. で木構造の統合を行う。「一文字分のみの $cpy()$（一致する個別ノードに相当）」「二文字分以上の $cpy()$（一致する部分木に相当）」「$cpy()$ 内に無い トークン（一致しないノードに相当）」で処理を分ける。一文字分のみの $cpy()$ の場合、それが u 側と v 側で高さが等しい時は探索範囲を文字トークンから対応する括弧トークンまでのサブシーケンスに狭め、等しくない時はそのサブシーケンスを挿入する。二文字分以上の $cpy()$ の場合、それが u 側と v 側で高さが等しい時は何もせず、等しくない時はそのサブシーケンスを挿入する。$cpy()$ 内に無い トークン の場合、文字トークンから対応する括弧トークンまでのサブシーケンスを挿入する。2. では 4.2.1 節の条件 (5) を満たすように、統合木の枝切りを行う。構造が全く同じサブシーケンスが複数存在した場合、各サブシーケンスの全先祖ノードを探索する。もし一方の全先祖が、他方の全先祖を包含していた場合、包含されている方のサブシーケンスを削除する。

「関数：サブシーケンスの挿入」では、サブシーケンスの挿入位置を決める。挿入するサブシーケンスの同じ高さの左側に、u 側と v 側で高さが等しい $cpy()$ が存在する場合、挿入位置はその $cpy()$ の右隣となる（4.2.1 節の条件 (3)）。存在しない場

合、現在の探索範囲の先頭から 2 番目が挿入位置となる（4.2.1 節の条件 (4)）。

図 8 は図 4 の XML 木に表 4・5 の手法を適用した例である。

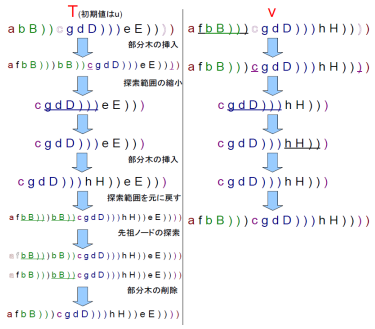

図 8　木構造の統合手法の例

v^0 が一文字のみ $cpy()$ 内にあるので、最初の探索範囲は $v^{0\cdots19}$ である。$v^{1\cdots6}$ が $cpy()$ に入っていないので挿入する。v^1 より左に同じ高さの文字トークンが無いので、挿入位置は T^1 となる。v^7 は一文字分のみの $cpy()$ であり、u（の現在の探索範囲内）にも同じ高さに一致するトークンが存在するため、捜索範囲を $u^{5\cdots16}, v^{7\cdots18}$ に狭める。$v^{7\cdots14}$ 内の二文字分以上の $cpy()$ $v^{8\cdots13}$ は、$u^{5\cdots16}$ 内にも同じサブシーケンスが存在するので挿入しない。同じく $v^{7\cdots14}$ 内の $v^{14\cdots17}$ は $cpy()$ に入ってないので挿入する。v^{14} より左に同じ高さの $cpy()v^{8\cdots13}$ が存在し、$u^{5\cdots16}$ 内にも同じ高さで一致する $cpy()u^{6\cdots11}$ が存在するので、挿入位置はその右隣の T^{18} となる。探索終了後の統合木を見ると、$T^{2\cdots5}$ と $T^{7\cdots10}$ が一致しているので、双方の先祖ノードをみる。T^2 よりも高さが高い左側の文字トークンは ”f”,”a”、T^7 よりも高さが高い左側の文字トークンは ”a” のみなので、$T^{7\cdots10}$ を削除する。最終的に出来上がる統合木は、図 9 の通り

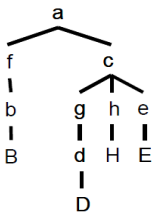

図 9　最終的に出来上がる統合木

である。

また、表 4 の 11〜18 行目の過程を省略すると、4.2.1 節で説明した統合木の条件 (5) が、以下のように変化する。

(5)　t_{uni} の 2 つの任意の部分木を t^A_{uni}, t^B_{uni} とする。t^A_{uni} の先祖ノードの集合を ANC^a、t^B_{uni} の先祖ノードの集合を ANC^b とすると、$(t^A_{uni} \neq t^B_{uni})$　または　$(t^A_{uni} = t^B_{uni}$ かつ $ANC^a \neq ANC^b)$

同様に、表 5 の 1〜6 行目の過程を省略すると、4.2.1 節で説明した統合木の条件 (4) が、以下のように変化する。

(4)　t_{uni} の 2 つの任意の兄弟ノードを t^a_{uni}, t^b_{uni} とする。t^a_{uni} と一致するノードが t_1 に存在せず、t^b_{uni} と一致するノードが t_1 に存在するならば、t^a_{uni} は t^b_{uni} より左にある。

5. 評価実験

本実験では提案手法の有用性を実データを用いて検証する。実験で使用したのは AMD Opteron(TM) 2-way Dual Core(2.4GHz) の CPU と 16GB のメインメモリを持つ Sun OS 5.10. マシンで、Java SE 1.6.0 07 で実装した。

本節で行う実験は以下の二つである。

（1）　4 節で説明した各手法に対する比較

（2）　比較手法を用いた本手法の有用性の検証

実験で用いるデータの概要を 5.1 節、実験 (1) の結果と考察を 5.2 節、実験 (2) の結果と考察を 5.3 節で説明する。

5.1 データセット

本実験では、1 節で紹介した XML 形式の論文目録データ「ACM SIGMOD」と「DBLP Bibliography」を用いる。「ACM SIGMOD」はそのまま用い、「DBLP Binliography」は一部データを抽出した、4 種類のデータを作成する。抽出方法は次の通りである。まず「ACM SIGMOD」と「DBLP Binliography」の各論文単位の部分木ペアに対し、テキスト類似度 T と構造類似度 S を計算する。それに以下の四つの条件を満たす部分木ペアの、「DBLP Binliography」側の部分木をそれぞれ抽出する。

（1）　$T \geqq 0.3$ かつ $S \geqq 0.4$　（全 112 個）

（2）　$T \geqq 0.3$ かつ $0.3 \leqq S < 0.4$　（全 282 個）

（3）　$T \geqq 0.3$ かつ $0.2 \leqq S < 0.3$　（全 408 個）

（4）　$T \geqq 0.3$ かつ $S < 0.2$　（全 212 個）

その (1)〜(4) に、ランダムに選んだ条件を満たさない部分木を加え、各データの部分木の合計が 100,000 個になるようにする。

5.2 各手法の比較

4 節で説明した、マッチリストを作成する手法三つ、及びマッチリストを元に木構造の統合を行う手法四つに対し、実行結果及び実行時間に関して比較を行う。実験手順は次の通りである。まず 5.1 節で説明したデータに対し、Wen らの手法で直列化し、テキスト類似度と構造類似度が一定以上の部分 XML データのペアを列挙する。その部分木に対し、4.1.1〜4.1.3 節で説明した「XAS」「ボトムアップ」「トップダウン」の 3 手法を用いてマッチリストを作成し、4.2.2 節で説明した「表 4 の 11〜18 行目の過程を省略する手法 (Max)、及び省略しない手法 (Min)」「表 5 の 1〜6 行目の過程を省略する手法 (Rough)、及び省略しない手法 (Precise)」の 4 手法を用いて木構造の統合を行う。その後その実行結果、及び実行時間を比較する。

5.2.1 実行結果

例として図 1 の二つの XML を統合した場合を見ていく。各統合手法毎の実行結果を図 10 に表す（同一タグ内にテキストノードが複数並ぶ時、それぞれを [] で括るようにしている）。図 1 の双方の XML で一致する title，authors タグを保持し、さらに DBLP 側にしかない pages，years 等のタグが追加されている。また、title タグは双方に存在するが、タグ内のテキストノードが異なるため（ピリオドの有無）、title タグ内に双方

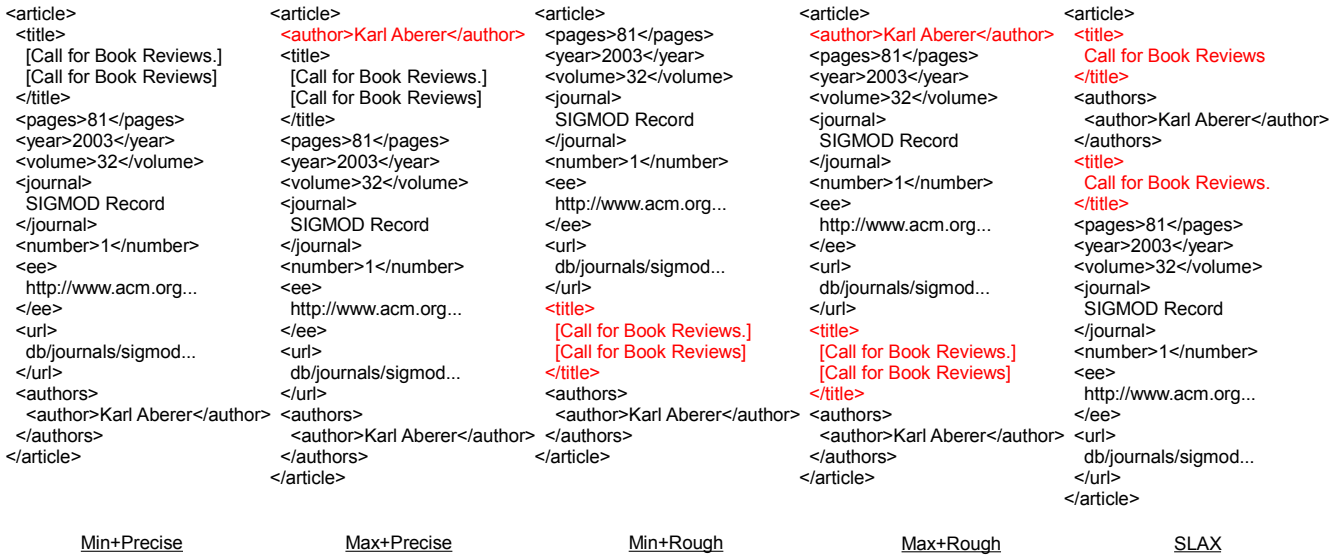

図 10　各手法における統合木の構造

のテキストノードが入る。

手法 Min と手法 Max との実行結果の違いは 2 行目の author タグの存在である。同一のタグとテキストノードが authors タグ内に存在するため、手法 Min では削除していたが、手法 Max ではその過程を省略したため、そのまま残る。手法 Precise と手法 Rough との実行結果の違いは title タグの位置である。手法 Precise ではタグの挿入の際、挿入される方でない部分木（今回の場合 DBLP 側）の兄弟の順番を考慮する。そのため title タグは pages タグの上になる。一方手法 Rough ではそれを優先しないため、title タグは authors タグの上になる。

5.2.2　実行時間

手法/構造類似度	0.4〜	0.3〜0.4	0.2〜0.3	〜0.2
〜部分木抽出	2580936.8	2721545.6	2647963.4	2776329.6
XAS	492.6	1182.8	1684.2	1249.6
Bottom-up	340.8	1122.2	1834.2	1101.8
Top-down	359.2	1107.2	1809.6	1090.4

表 6　マッチリスト作成における実行時間（ミリ秒）

手法/構造類似度	0.4〜	0.3〜0.4	0.2〜0.3	〜0.2
〜マッチリスト作成	2581277.6	2722652.8	2649647.6	2777420
Min + Precise	638	1929.5	3832.3	2120.2
Max + Precise	564.5	1907.3	3755.2	1911.7
Min + Rough	604.5	1897.3	3801.7	2038.8
Max + Rough	557.2	1891.3	3747.7	1896.8

表 7　木構造の統合における実行時間（ミリ秒）

表 6 は、直列化から部分木抽出までの実行時間とマッチリスト作成の各手法の実行時間を表したものである。同じく表 7 は、直列化からマッチリストの作成までの実行時間と、マッチリストを用いた木構造の統合の各手法の実行時間を表したものである（マッチリスト作成の手法は、各データ毎最も早い手法を用いている）。類似結合全体に対するマッチリスト作成、及び木構造の統合の実行時間は非常に短い事が分かる。

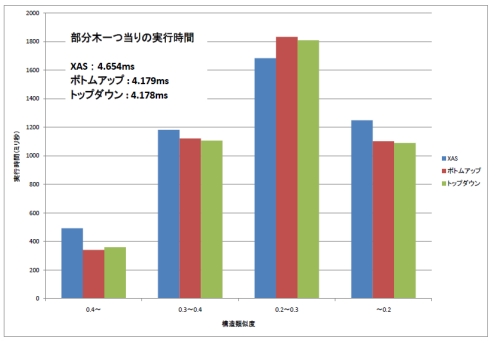

図 11　マッチリスト作成における各手法の実行時間の比較

図 11 は、マッチリスト作成の各手法の実行時間を、構造類似度毎にグラフ化したものである。また、部分木一つ辺りの実行時間も記した。

結果から、手法 XAS が平均的に実行時間が遅いことが分かった。手法ボトムアップ、手法トップダウンが NoK パターンをそのまま利用するのに対し、手法 XAS は NoK パターンを XAS に変換するため、その分手間がかかるためだと推測できる。しかし構造類似度別に見ると、0.2〜0.3 のみ実行時間が早くなった。手法 XAS ではデータを決められた長さのサブシーケンスに分断するため、そのシーケンスの構成（一致する位置や長さなど）によって実行時間が左右されやすいからだと考えられる。また、手法ボトムアップと手法トップダウンでは大きな差は見られなかった。

図 12 は、木構造の統合の各手法の実行時間を、構造類似度毎にグラフ化したものである。また、部分木一つ辺りの実行時間も記した。

結果から、どのデータでも、手法 Max は手法 Min よりも早

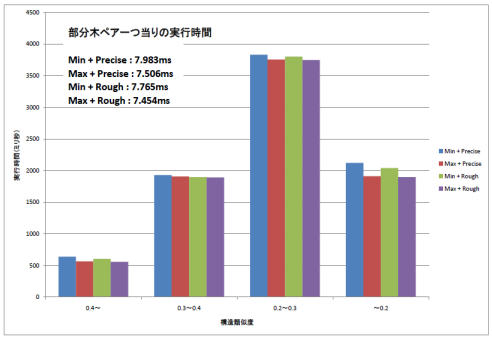

図 12　木構造の統合における各手法の実行時間の比較

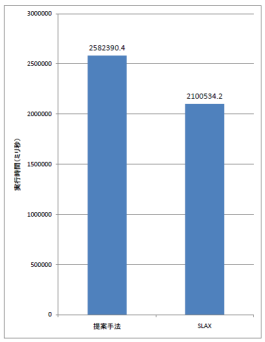

図 13　類似結合全体の実行時間

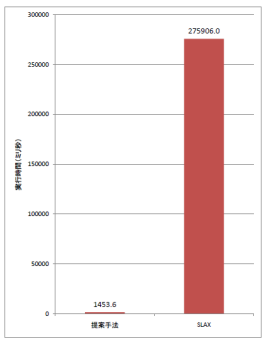

図 14　統合部分のみの実行時間

く、手法 Rough は手法 Precise よりも早いことが分かった。手法 Max、手法 Rough に比べて、手法 Min、手法 Precise は処理が多いため、それに比例した実行時間がかかるためだと考えられる。

5.3　比較手法との比較

提案手法の有用性を検証するため、従来手法である SLAX [6] との比較を行う。

SLAX は Liang らが提案した、XML のテキストノードに対して類似度計算を行う手法である。双方の XML 木のテキストノードを比較して類似度を計算し、類似度が閾値を超えた部分木ペアを抽出する。そして一致しないテキストノードをその親ごと、ルートノードの下に挿入する。比較的容易に部分木の統合を行う事が可能だが、統合する際にその構造までは考慮していない。

実験手順は次の通りである。まず 5.1 節で説明したデータに対し SLAX を適用し、類似度 0.11 以上の部分木ペアを抽出する。「DBLP DBLP Binliography」側はデータ (1)（構造類似度 0.4 以上）を使用する。次に部分木ペアの数を提案手法側と同じにするため、部分木ペアをランダムに 112 個選択し、それに対して統合を行う。その実行結果及び実行時間を、5.2 節の結果と比較する。

5.3.1　実 行 結 果

図 1 の二つの XML を SLAX を用いて統合した結果を図 10 で表す（一番右）。大きな違いは二つ目の title タグの存在である。title タグの一致を考慮していないため、提案手法の結果と異なり双方の title タグがそのまま残る。また兄弟ノードの順番も考慮していないため、authors タグの位置も異なっている。

5.3.2　実 行 時 間

図 13、図 14 は SLAX と提案手法の実行時間をグラフ化したものである。図 13 が類似結合全体、図 14 が部分木の統合部分のみの実行時間を表す。なお提案手法側は手法ボトムアップ、手法 Min+Precise を用いている。

実行時間全体では、提案手法の方がおよそ 1.25 倍遅かった。木構造の直列化など、提案手法が SLAX に比べて処理が多いからだと推測される。しかし部分木の統合部分に関しては、提案手法の方が圧倒的に早かった。統合を行う際、SLAX は木構造を直接扱うのに対し、提案手法は文字列に対し統合を行っているからだと推測される。

6.　お わ り に

本稿では NoK パターンを用いて木直列化した XML データ群に対し、その構造を考慮して統合を行う手法を提案した。これは比較する直列化した XML 木に対し、一致する部分木やノードの情報を表すマッチリストを作成する手法と、それを元に文字列の統合を行う手法で構成されている。この二つの手法を組み合わせる事で、木構造の統合を行う事が出来る。さらに実データを用いた実験によって、本手法によって木構造を考慮した部分木の統合が行えることを確認した。それに加え、従来の手法に比べて部分木の統合にかかる実行時間が短いことが分かった。

今後の課題としては、DTD を利用した、XML 文書の整合性を考慮した統合手法の開発などが挙げられる。

謝辞　本研究の一部は、科学研究費補助金（♯19024006, ♯19700083）による。

文　　献

[1] K.Zhang and D.Shasha. *Tree Pattern Matching*. Pattern Matching Algorithm, Aposolico and Galil Editors Oxford University Press, 1997.

[2] Lianzi Wen, Toshiyuki Amagasa, and Hiroyuki Kitagawa. An approach for xml similarity join using tree serialization. In *13th Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, 2008.

[3] Zhang N., Varun Kacholia, and Ozsu M.T. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *20th International Conference on Data Engineering*, 2004.

[4] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple xml tree differencing by sequence alignment. In *ACM symposium on Document engineering*, 2006.

[5] T. Lindholm J. Kangasharju. A sequence-based type-aware interface for xml processing. In *European Internet and Multimedia Systems and Applications*, 2005.

[6] W. Liang and H. Yokota. *SLAX*: An improved leaf-clustering based approximate xml join based on clustered leaf nodes for xml data integration at subtree classes. データベースと Web 情報システムに関するシンポジウム, 2005.