左右木結合を用いた頻出順序木パターン発見アルゴリズム

比戸　将平†　　河野　浩之††

† 京都大学大学院情報学研究科システム科学専攻　〒 606–8501 京都市左京区吉田本町
†† 南山大学数理情報学部情報通信学科　〒 489–0863 愛知県瀬戸市せいれい町 27
E-mail: †hido@sys.i.kyoto-u.ac.jp, ††kawano@it.nanzan-u.ac.jp

あらまし　近年，HTML や XML 等の半構造データに対するデータマイニング処理として頻出部分構造マイニングに関する研究が盛んに行われている．本稿では，与えられたデータ木から頻出する順序木パターンを高速に発見するために，サイズ k の頻出順序木パターン集合から左木・右木を用いて結合候補クラスを生成し，左右木結合を行うことにより，サイズ $k+1$ の頻出順序木の候補を効率良く列挙するアルゴリズムを提案する．人工の木構造データと XML データを用いた実験によって，提案アルゴリズムは全ての頻出順序木パターンをデータサイズに対して概ね線形時間で重複無く発見できることを示した．同じ頻出順序木パターン発見アルゴリズムである FREQT と比較したところ，我々の手法は数十パーセントから数倍高速であるという結果が得られた．

キーワード　データマイニング，半構造データ，頻出構造マイニング，頻出順序木パターン発見

Frequent Ordered Subtree Discovery Algorithm Using Left-Right Tree Join

Shohei HIDO† and Hiroyuki KAWANO††

† Department of Systems Science, Graduate School of Informatics, Kyoto University
Yoshida Hon-machi, Sakyo-ku, Kyoto, 606–8501 Japan
†† Faculty of Mathematical Sciences and Information Engineering,
Department of Information and Telecommunication Engineering,
Nanzan University, Seirei-cho 27, Seto, Aichi, 489–0863 Japan
E-mail: †hido@sys.i.kyoto-u.ac.jp, ††kawano@it.nanzan-u.ac.jp

Abstract　Recently many studies have been conducted to discover the frequent subtrees in the tree data, as data mining on semi-structured data such as HTML or XML. In this paper, we propose a fast algorithm to discover all frequent trees in the tree data, using Left-Right Tree Join for effecient generation of the set of candidate tree with size $k+1$ from the set of frequent tree with size k. In the experiments with artificial tree data and XML data, proposed algorithm can extract all frequent subtrees without any duplicates for the linear time compexity with the size of the tree data. Our method outperforms FREQT, typical frequent subtree discovery algorithm, by up to several times in the execution time.

Key words　Data Mining, Semi-structured Data, Frequent Structure Mining, Frequent Ordered Induced Subtree Discovery

1. まえがき

近年，発展著しい Web やバイオインフォマティクスなどの分野において木やグラフで表現可能な半構造データの普及が進んでいる．例えば Web において現在文書コンテンツの主要な記述形式として用いられている HTML は木構造を持つ構造化テキストデータの一種と見なせる．また様々なデータの保存・交換フォーマットとしてビジネス分野でも広く用いられている XML も木構造データである．一方，バイオインフォマティクスにおける半構造データの例としては，RNA やタンパク質の 2 次構造がある [1]．実際，木構造のマッチングを RNA の 2 次構造の比較に利用した研究が行われている [2]．これらの半構造データはストレージの大容量化とネットワークの高速化によりデータベースに大量に蓄積されており，その中から有用な知識を発見するためのデータマイニングに対する社会的需要は大きい．

しかし半構造データはリレーショナル形式で表されるデータよりも緩い構造を持つため，柔軟な記述が可能である反面，平坦なリレーショナル形式に比べて極めて不均質で複雑な構造を持つ．そのためリレーショナルデータに対する従来のデータマイニング手法をそのまま半構造データに適用することは難しい．よって半構造データに対する効率の良いデータマイニング技術が必要とされている．

データマイニングにおける代表的な処理は，データベース中に多く出現するパターンを発見することである．リレーショナル形式のデータ集合から頻出するアイテム集合を発見するアルゴリズムは古くから研究されており，Agrawal らの Apriori [3] などが良く知られている．一方半構造データ中に頻出する部分構造を探す頻出構造マイニングはここ数年盛んに研究されており，様々なアルゴリズムが提案されてきた [4], [5].

本稿では半構造データの中でも XML や HTML などの木構造データを研究対象とし，データ木中に頻出する順序木パターンを発見する問題を扱う．同様の問題を解く代表的なアルゴリズムとしては浅井らの FREQT [6] や Zaki の TreeMiner [7] などが挙げられる．頻出順序木パターン発見問題は，リレーショナルデータにおける頻出アイテム集合発見問題の木構造データへの拡張として定式化されている．頻出順序木を発見するために用いられる基本的な方法は，サイズ k の頻出木パターンからサイズ $k+1$ の頻出候補木集合を列挙し，各候補木のデータ木中の出現回数を求め，頻出であるかどうか調べるというものである．しかし，節点数 1 の木からスタートして次々に節点を増やしていく単純な方法は，同じ候補木の生成に至る経路が複数存在するため，効率が悪い．さらに候補木の数が木のサイズに対して指数的に存在するため，サイズが増大すると性能が著しく低下する．そこで，候補木をいかに効率的に列挙するかという問題が，頻出木発見では鍵となる．FREQT 及び TreeMiner ではそれぞれ独立に，最も右に位置する枝に葉節点を加える最右拡張を提案し，頻出候補木を重複無く列挙している．本稿では新たに，左右木結合を用いた効率的な頻出候補木の列挙方法により，頻出順序木パターンを高速に求めるアルゴリズムを提案する．

本稿は以下のように構成されている．まず 2 章では頻出順序木パターン発見問題について説明する．次に 3 章では既存手法として FREQT と TreeMiner を紹介する．そして 4 章で左右木結合を用いた頻出順序木パターン発見アルゴリズムを提案する．5 章で提案手法と既存手法を比較し，人工の木構造データと XML を用いた実験によって提案アルゴリズムの性能を評価する．最後に 6 章で結論と今後の課題についてまとめる．

2. 頻出順序木パターン発見問題

本章ではラベル付き順序木のデータ構造について述べ，頻出木の定義を明らかにした後，頻出順序木パターン発見問題とその性質について説明する．

2.1 順序木のデータモデル

本稿で扱うラベル付き順序木のデータモデルについて説明する．

$\mathcal{L} = \{l_0, l_1, \cdots\}$ をラベルの有限集合とする．$\mathcal{L}$ 上のラベル付き順序木は $T = \{V, E, B, root, label\}$ として表現される．V は節点の集合であり，$V = \{v_0, v_1, \cdots\}$ である．節点数 $|V|$ を用いて，木のサイズ $|T|$ を $|T| = |V|$ と定義する．$E \subseteq V^2$ は親節点と子節点の接続を表す二項関係である．また $B \subseteq V^2$ は隣接兄弟間の順序を保持している．$root$ は木 T が唯一持つ根節点であり，また $label : V \to L$ は節点 $v \in V$ に対応するラベル $l \in \mathcal{L}$ を求めるラベル関数である．ある 2 節点 $u, v \in V$ に対して $(u, v) \in E$ が成り立つとき，u を v の親，v を u の子とする．子を持たない節点を葉節点と呼ぶ．また，節点列 $P_v = (v_0, v_1, \cdots v_n) \subseteq V$ において，任意の $0 \leqq i < n$ に対して $(v_i, v_{i+1}) \in E$ が成立する場合，P_v を v_0 から v_n への経路と呼ぶ．節点 v の深さ $depth(v)$ を，根節点から v までの経路の長さとする．そして，$0 \leqq i < n$ と $1 \leqq k \leqq n - i$ に対し，v_i を v_{i+k} の先祖と呼び，v_{i+k} を v_i の子孫と呼ぶ．

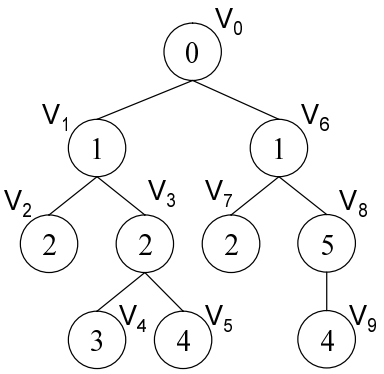

図 1　ラベル付き順序木の例

図 1 はラベルの種類数 $|\mathcal{L}| = 6$，サイズ $|T| = 10$ のラベル付き順序木の例である．ラベル 0 を持つ v_0 の節点が根節点である．そして v_1 と v_6 がその子節点であり，兄弟関係にある．以上のようにして，ラベル付き順序木を表現する．以後ラベル付き順序木を順序木または単に木と呼ぶ．

2.2 順序木の出現

データ木 D における順序木 T の出現を，以下のように定義する．

定義 2.1. 順序木 $T = \{V_T, E_T, B_T, root_T, label_T\}$ と順序木 $D = \{V_D, E_D, B_D, root_D, label_D\}$ において以下の 4 条件を満たす写像 $\varphi : V_T \to V_D$ が存在する時，T は D に出現するという．

（1） $x, y \in V_T$ が $x \neq y$ であるならば，$\varphi(x), \varphi(y) \in V_D$ も $\varphi(x) \neq \varphi(y)$ を満たす．すなわち，φ は単射である．

（2） φ はラベルを保存する．つまり，$label_T(x) = label_D(\varphi(x))$ である．

（3） φ は親子関係を保存する．すなわち，$(x, y) \in E_T$ ならば $(\varphi(x), \varphi(y)) \in E_D$ である．

（4） φ は兄弟関係を保存する．つまり，$(x, y) \in B_T$ である時，$(\varphi(x), \varphi(y)) \in B_D$ が成り立つか，もしくはある節点列 $Q_D = (v_0, v_1, \cdots, v_n) \subseteq V_D$ が存在し，$0 \leqq i < n$ について $(v_i, v_{i+1}) \in B_D$ であり，かつ $(\varphi(x), v_0) \in B_D$，$(v_n, \varphi(y)) \in B_D$ が成り立つ．

データ木中の順序木出現数の定義はいくつか存在するが，本稿では根出現数を用いる．根出現数は，データ木 D における順序木 T の出現の中で，重複を除いた根節点 root の出現数 $|Occ(root)|$ である．これにより D 中の T の出現頻度 $F_D(T)$ は以下のように定義する．

定義 2.2. 順序木 $T = \{V, E, B, root, label\}$ のデータ木 D における出現頻度 $F_D(T)$ を，

$$F_D(T) = \frac{|Occ(root)|}{|D|}$$

とする．

以後，サイズ k である頻出順序木パターンを単に k-パターンと表す．また，k-パターン全体の集合を $\mathcal{F}_k$，k-パターンを求めるためのサイズ k の頻出候補木の集合を $\mathcal{C}_k$ とする．

2.3 頻出順序木パターン発見問題

頻出木発見問題を解く際，ある木が頻出か否かを決める出現頻度の閾値として，最小サポート (minimum support) として $\sigma(0 \leqq \sigma \leqq 1)$ を与える．これを用いて，頻出順序木を以下のように定義する．

定義 2.3. 最小サポート $\sigma(0 \leqq \sigma \leqq 1)$ が与えられた時，D 中の順序木 T の出現頻度 $F_D(T)$ が $F_D(T) \geqq \sigma$ を満たす場合，順序木 T は頻出とする．

以上より，頻出順序木パターン発見問題を以下のように定式化する．

問題 2.4. データ木 $D = \{V, E, B, root, label\}$，最小サポート $\sigma(0 \leqq \sigma \leqq 1)$ が与えられた時，D 中の出現頻度 $F_D(T)$ が $F_D(T) \geqq \sigma$ を満たすような頻出順序木 T を全て求めよ．

例として，図 2 のデータ木 D において最小サポート $\sigma = 20\%$ とした頻出順序木パターン発見問題を考える．その下の頻出候補木 T_1 は，根出現数が 3 であり，$F_D(T_1) = \frac{3}{15} = 0.2 = \sigma$ より，頻出である．一方，頻出候補木 T_2 は一見出現数が 3 のように見えるが，根出現数としては 2 であるため，$F_D(T_2) = \frac{2}{15} = 0.1333\cdots < \sigma$ より，非頻出である．

3. 頻出順序木発見における関連研究

本章では頻出順序木パターン発見問題の持つ性質について述べたあと，代表的な既存アルゴリズムを紹介する．

3.1 問題の性質

頻出順序木パターンの性質として，まず以下の公理が挙げられる．

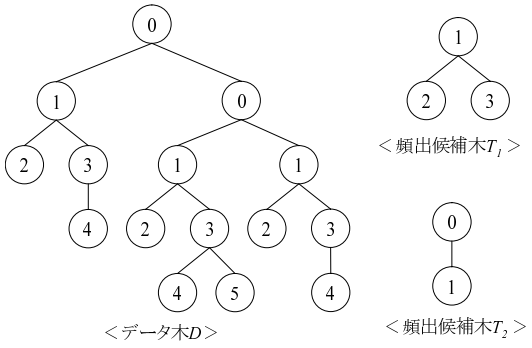

図 2　頻出順序木パターンの出現

公理 3.1. k-パターンから，いずれかの葉接点 *1* つを除いたサイズ $k-1$ の部分木は全て $(k-1)$-パターンである．

これは，頻出アイテム集合発見問題における代表的なアルゴリズムである Apriori [3] で用いられている，頻出アイテム集合の部分アイテム集合に関する公理と類似するものである．

この公理に基づき，多くの頻出木パターン発見アルゴリズムは以下のように構成されている．

STEP 1.　データ木 D を走査し，$\mathcal{F}_1$ を求める．
STEP 2.　$k > 1$ に対し，$\mathcal{F}_k$ からサイズ $k+1$ の候補木 C を列挙し，候補木集合 $\mathcal{C}_{k+1}$ に加える．
STEP 3.　データ木 D における各候補木 $C \in \mathcal{C}_k$ の出現頻度を求め，$F_D(C) \geqq \sigma$ であれば C を $\mathcal{F}_{k+1}$ に加える．
STEP 4.　$|\mathcal{F}_{k+1}| = 0$ であるならば終了．そうでなければ $k = k+1$ として STEP 2 に戻る．

ここで重要となるのは，STEP 2 においてどのように候補木を列挙し，$\mathcal{C}_{k+1}$ を生成するかということである．最も単純な方法としては，各 k-パターン $F \in \mathcal{F}_k$ において，1 つの葉節点にある $l \in \mathcal{L}$ を持つ新しい節点を付加して構成されるサイズ $k+1$ の木を $\mathcal{C}_{k+1}$ に加えるという操作を，全 k-パターンの全ての葉節点に対し，$|\mathcal{L}|$ 回繰り返す事が考えられる．しかし，この方法では同じ候補木が複数回列挙される場合が多く，STEP 3 においてデータ木の無駄な走査が行われてしまう可能性が高い．また，k が大きくなるにつれて候補木集合の大きさ $|\mathcal{C}_{k+1}|$ が指数的に増大し，現実的な時間では動作しなくなるため，極めて効率が悪く，実用的ではない．そこで，頻出候補木の列挙に最右拡張という方法を用いるアルゴリズムとして，浅井らによる FREQT [6] と Zaki による TreeMiner [7] がそれぞれ独立に提案された．

3.2 FREQT

まず最右拡張を，FREQT において用いられた形式で説明する．順序木 T において，最も右に位置する葉節点，つまり根節点から深さ優先で木を走査した時に最後に到達する葉節点を最右葉 (Right Most Leaf : $rml(T)$) と呼ぶ．また，根節点から最右葉に至るまでの経路を最右枝 (Right Most Branch) とする．

定義 3.2. 任意の順序木 S において，最右枝上の節点に対し新たな節点 v を右端の子として追加して得られる順序木 T を S の最右拡張という．

任意の木 T から最右葉を除いた親木 $P(T)$ は一意に定まる．よって，全ての頻出候補木 $C \in \mathcal{C}_{k+1}$ は $P(C) \in \mathcal{F}_k$ の最右拡張として重複無く列挙される．また，最右拡張は定数時間 $O(1)$ で行うことができる．

ただし，全てのラベルについて節点の拡張を行うことは，無駄な候補木を列挙する可能性があり，非効率的である．そこで FREQT では，$\mathcal{F}_1$ を用いて，あらかじめ各ラベル $l \in \mathcal{L}$ について $label(v) = l$ で唯 1 つの節点 $v \in V$ からなるサイズ 1 の木が $\mathcal{F}_1$ に含まれているか否かを調べ，$\mathcal{F}_1$ に含まれていないラベルを持つ節点の拡張を行わない工夫がされている．これを節点ス

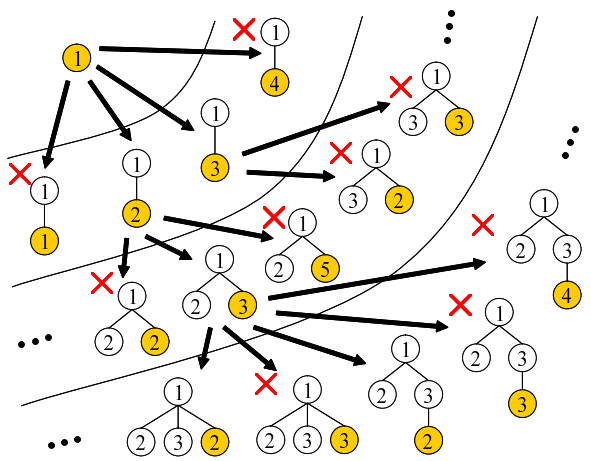

図 3　最右拡張を用いた FREQT による順序木列挙

キップという．同様に，各ラベルの組 $(l_1, l_2) \in \mathcal{L}^2$ について，2 つの節点 $v_0, v_1 \in V$ から成り，$(v_0, v_1) \in E$，$label(v_0) = l_0$，$label(v_1) = l_1$ であるサイズ 2 の木を考え，$\mathcal{F}_2$ に含まれているか否か調べておく．そして最右拡張の際，対象となっている節点のラベルが l_0 の場合，ラベル l_1 を持つ節点を用いた拡張は行われない．これを枝スキップと呼ぶ．

FREQT では，順序木は深さラベル列を用いて表現されている．深さラベル列は，根節点から深さ優先で走査して現れた順に，節点の深さとラベルの組を並べることで順序木に一意的な表現を与えることができる．

$$T = (depth(root), label(root)) \cdot (depth(v_1), label(v_1)) \cdots$$

FREQT においては，各 k-パターンの $T \in \mathcal{F}_k$ の最右葉が出現する位置として $RMO_k(T) \in RMO_k$ が保存されている．T に最右拡張を行って得られた $C \in \mathcal{C}_{k+1}$ の出現数を調べる際には，$RMO_k(T)$ を利用し，RMO_{k+1} を更新する．ただしその際，同じ節点が重複して計算される可能性がある．そこで重複検出によって重複を完全に回避している．

FREQT による頻出順序木発見の模式図を図 3 に示した．最右拡張により候補木が生成され，頻出でない順序木パターンにおいて枝刈りが行われ，その非頻出木を部分木として含む冗長な候補木の生成が抑制されている様子が見て取れる．

FREQT では，全ての頻出順序木パターンとその最右葉出現を重複無しに求められることが保証されている．また，その計算時間は $O(k^2bLN)$ である．ここで，k は極大頻出順序木パターンのサイズ，b はデータ木 D の最大枝分かれ数，N は最右葉出現リストの長さの総和，$L = |\mathcal{L}|$ である．

3.3　TreeMiner

TreeMiner は Zaki によって提案された頻出順序木パターン発見アルゴリズムである [7]．ただし，FREQT 及び本稿とは木の出現の定義が異なり，定義 2.1 の (3) を次の (3') に変更した定義に相当する．

(3')　φ は先祖親子関係を保存する．すなわち，$x, y \in V_T$ に対し x が y の先祖ならば，$\varphi(x)$ は $\varphi(y)$ の先祖である，つまり $\varphi(x)$ から $\varphi(y)$ への経路 $P \subseteq V_D$ が存在する．

サイズ $k-1$ の順序木 S の最右拡張から得られた k-パターン集合は，深さラベル列の接頭辞として S を共通に持つ，接頭辞同値類 (Prefix Equivalence Class) と考えることができる，サイズ $k+1$ の順序木 T の長さ $k-1$ の接頭辞が S である時，$0 \leqq d_2 \leqq d_1 + 1$ として $T = S \cdot (d_1, l_1) \cdot (d_2, l_2)$ と表現できる．ここで，公理 3.1 に基づき，サイズ k の $T_1 = S \cdot (d_1, l_1)$ が頻出で，かつ $d_2 \leqq d_1$ である場合に $T_2 = S \cdot (d_2, l_2)$ も頻出である時，T を頻出候補木集合 $\mathcal{C}_{k+1}$ に加え，その出現頻度を計算する．

TreeMiner では，FREQT と異なりパターンの全節点の出現位置を保持している．実装上の工夫として，全ての節点の組 (u, v) についてに対し，ある順序木の根節点が u である時に，データ木上で v が最右葉として出現しうる範囲 (Scope) をあらかじめ計算しておく．この出現範囲リスト (Scope-List) を利用することにより，出現位置の効率的な更新が可能となる．

4.　左右木結合を用いた頻出順序木パターン発見

本章ではまず左右木結合を説明したあと，それを用いた頻出順序木パターン発見アルゴリズムについて述べる．

4.1　左右木結合

順序木 T において，3.2 節で与えた最右葉と同様に，根節点から深さ優先でを辿った時に最初に現れる葉節点を最左葉 (Left Most Leaf : $lml(T)$) と呼ぶ．ある木 T から最左葉を除いた部分木を右木として $Right(T)$，最右葉を除いた部分木を左木 $Left(T)$ と表す．また，最左葉と最右葉を両方除いた木を中央木 $Center(T)$ とする．

定理 4.1. 任意の順序木 T に対し，

$$Center(T) = Right(Left(T)) = Left(Right(T))$$

が成立する．

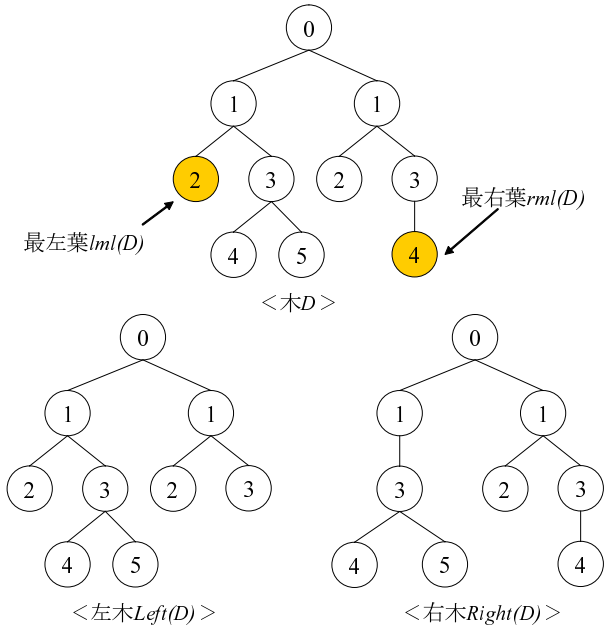

図 4　左右木表現の例

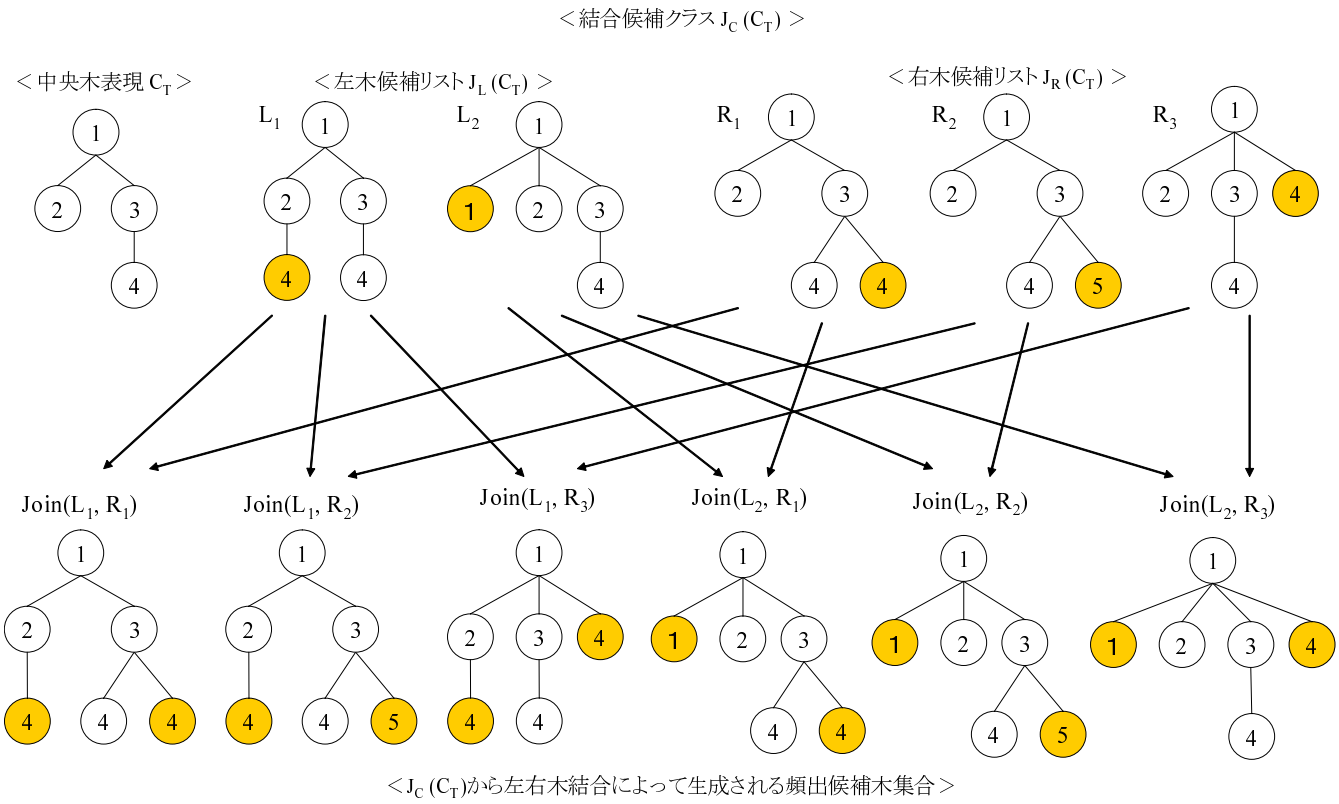

図 5　結合候補クラスにおいて左右木結合を用いた頻出候補木列挙

証明.　最左葉を取り除く操作は最右葉の表現に影響を与えず，最右葉を除く操作も最左葉に影響を及ぼさない．よって，$Right(Left(T)) = Left(Right(T))$ が成立する．□

$Right(T_1) = Left(T_2)$ である 2 つの順序木の左右木結合 (Left-Right Tree Join) を以下のように定義する．

定義 4.2. *2 つの順序木 T_1 と T_2 が $Right(T_1) = Left(T_2)$ を満たす時，その左右木結合を*

$$join(T_1, T_2) = T_1 \cup rml(T_2) = lml(T_1) \cup T_2$$

と定義する．

任意の順序木 T において次の 2 つの定理が成り立つ．

定理 4.3. 任意の順序木 T はその左木と右木の左右木結合として表現される．

証明.　定理 4.1 より，任意の順序木 T について

$$Right(Left(T)) = Left(Right(T))$$

が成立する．よって，定義 4.2 に従い，T は

$$T = join(Left(T), Right(T))$$

と表現することができる．□

定理 4.4. 左木と右木の組と，その左右木結合によって得られる順序木は一対一に対応する．つまり，

$$join(T_1, T_2) = join(S_1, S_2) \Leftrightarrow T_1 = S_1 \text{かつ } T_2 = S_2$$

が成り立つ．

証明.　(十分性) 右木と左木の表現は一意に定まるので，

$$Left(join(Left(T), Right(T))) = Left(T)$$
$$Right(join(Left(T), Right(T))) = Right(T)$$

である．よって $join(T_1, T_2) = join(S_1, S_2)$ の両辺の左木及び右木をとれば $T_1 = S_1$かつ $T_2 = S_2$
(必要性) 左右木結合の定義より明らか．□

4.2　アルゴリズム

左右木結合を頻出候補木列挙に適用する方法について説明する．

ここで $(k+1)$-パターン $T \in \mathcal{F}_{k+1}$ について考える．T の左木と右木は，公理 3.1 より，必ず k-パターン $Left(T), Right(T) \in \mathcal{F}_k$ である．よって，全ての $(k+1)$-パターン T は，$Right(T_1) = Left(T_2)$ を満たす $T_1, T_2 \in \mathcal{F}_k$ の左右木結合により得られる．

左右木結合によって頻出候補木を列挙するために，結合候補クラスを作成する．結合候補クラス $J_C(C_T)$ は中央木表現 C_T ($|C_T| = k-1$) と左木候補リスト $J_L(C_T)$，右木候補リスト $J_R(C_T)$ から成る構造 $J_C(C_T) = \{C_T, J_L(C_T), J_R(C_T)\}$ である．ある結合候補クラス $J_C(C_T)$ 内の左右木候補リスト $J_L(C_T), J_R(C_T) \subseteqq \mathcal{F}_k$ において，$(L, R) \in J_L(C_T) \times J_R(C_T)$ を用いて $C = join(L, R)$ を求め，C を $\mathcal{C}_{k+1}$ に加えることで，頻出候補木を生成する．結合候補クラスと左右木結合による頻出候補木列挙の例を図 5 に示した．サイズ 4 の中央木表現 C_T を右木に持つ左木候補 L_1,L_2 及び C_T を左木として持つ右木候

補 R_1,R_2,R_3 を用いて左右木結合による頻出候補木が 6 つ生成されている．

以下に左右木結合を用いて頻出候補木を列挙し，頻出順序木パターンを発見するアルゴリズムの流れを示す．

STEP 1. データ木 D を走査し，$\mathcal{F}_1$,$\mathcal{F}_2$ を求める．

STEP 2. $k > 1$ について，各 $T \in \mathcal{F}_k$ に対し (a)-(d) の操作を行う．

(a) $Left(T)$ を中央木表現として持つを結合候補クラス $J_C(Left(T))$ を選ぶ．

(b) その左木候補リスト $J_L(Left(T))$ 内の全ての木 T_L に対し，左右木結合 $join(T_L, T)$ を頻出候補木とする．

(c) 右木候補リスト $J_R(Left(T))$ に T を加える．

(d) 同様に，結合候補クラス $J_C(Right(T))$ を選び，$J_R(Right(T))$ 内の木 T_R と T の左右木結合 $join(T, T_R)$ を頻出候補木とし，$J_L(Right(T))$ に T を加える．

STEP 3. データ木 D における各候補木 $C \in \mathcal{C}_k$ の出現頻度を求め，$F_D(C) \geqq \sigma$ であれば C を $\mathcal{F}_{k+1}$ に加える．

STEP 4. $|\mathcal{F}_{k+1}| = 0$ であるならば終了．そうでなければ $k = k + 1$ として STEP 2 に戻る．

葉節点数が 1 の木，つまり根節点から唯一つの葉節点までの経路として表せる木を直列木と呼ぶ．直列木は葉節点数が 2 以上である木と異なり，上記の左右木結合による方法では列挙を行うことができない．そこでサイズ k-1 の頻出木パターン集合に属する直列木において，葉節点の下にさらに 1 つ節点を加えることでサイズ k の頻出候補直列木へと拡張を行う．以上の手法により、全ての頻出候補木を列挙することができる．

出現数の計算方法であるが，ここでは FREQT と同様に，順序木の最右葉がデータ木に出現する位置を保持することにより，重複を検出して避けながらその出現数を数え上げる．また木のデータ表現にも FREQT と同じ深さラベル列を用いた．

5. 提案アルゴリズムの性能評価

提案アルゴリズムの性能を検証し，その有効性を確かめるための実験を行う．

5.1 アルゴリズムの考察と最右拡張との比較

左右木結合による頻出候補木の列挙が線形時間で行えることを示す．定義 4.2 より，左右木結合によって得られた木 T は，左木 $Left(T)$ におけるラベル $label(rml(Right(T)))$ を用いた最右拡張に等しい．つまり左右木結合による木の生成は，最右拡張と同じく定数時間で実行可能である．また定理 4.4 より，全ての頻出候補木は重複無く 1 回ずつ左右木結合によって生成されることがわかる．よって，左右木結合による頻出候補木の列挙は線形時間で可能である．さらに頻出直列木の拡張による頻出候補直列木の列挙も同様に線形時間で行われる．以上より，提案アルゴリズムにおける頻出候補木の列挙は，その個数に対して線形時間で実行可能である．

次に生成される頻出候補木の個数を比較する．定義 4.2 より，左右木結合によって得られた木 T は，左木 $Left(T)$ におけるラベル $label(rml(Right(T)))$ を用いた最右拡張に等しい．つまり，左右木結合で生成される頻出候補木は必ず最右拡張によっても生成される．一方，最右拡張によって生成される頻出候補木が左右木結合によって生成される保証はない．従って最右拡張によるサイズ k の頻出候補木集合 $\mathcal{C}_k^F$ と提案アルゴリズムによるサイズ k の頻出候補木集合 $\mathcal{C}_k^{LR}$ には $\mathcal{C}_k^{LR} \subseteqq \mathcal{C}_k^F$ という関係が常に成り立つ．頻出候補木の個数が少なければ，無駄な候補木のデータ木における出現数の計算を省略することができるので，実行時間が短縮される．以上より，左右木結合を用いた列挙は最右拡張よりも無駄な候補木が少ない効率の良い列挙と考えられる．

次に，メモリ使用量を比較する．最右拡張を用いた FREQT や TreeMiner は深さ優先方式で実行できることがわかっており，少ないメモリ使用量で動作することが可能である．一方，提案アルゴリズムは結合候補クラスなどを用いているため幅優先方式となり，同時に保持するべき木の数が大きく，メモリ使用量は必然的に大きくなる．しかし，木の性質と左右木結合の特徴を利用することで探索空間を分割し，ある程度はメモリ使用量を削減することも可能である．例えば 2 つの木において最大深さや葉節点数が 2 以上異なる場合，それら同士の左右木結合は行うことができないなどの性質がある．そのため葉節点数 1 の木の集合から 1 ずつ節点数を増やしながら左右木結合を行った場合，葉節点数 3 の木集合に至る前に葉節点数 1 の木は全て不要となるため，メモリ上から削除することが可能となり，同時に使用するメモリ量を低減することができる．

5.2 性能の比較実験

本稿で提案したアルゴリズム及び FREQT を Java で実装し，性能評価実験を行った．提案手法の実装においては前節の最後に挙げたメモリ使用量削減方法を用いた．以下に実験に使用した PC 環境を示す．

表 1 実験環境

CPU	Pentium4 2.40GHz
メモリ	512MB
開発言語	Java (Sun JavaSDK 1.5.0_01)
JavaVM のメモリ制限	256MB
OS	WindowsXP Professional Edition

5.3 人工データ木を用いた実験

人工データ木は，パラメータとしてラベル集合の要素数 $|\mathcal{L}|$，節点数 $|V|$，最大深さ d_{max} を与え，ランダム関数を用いて作成したデータ木 D_A のことである．

まず，我々のアルゴリズムの規模耐性を検証するために，$|\mathcal{L}| = 6$，$d_{max} = 100$，最小サポート $\sigma = 0.04\%$ において節点数 $|V|$ を徐々に増加させながら，D_A 中の頻出順序木パターンを求めた．図 6 は実行時間のグラフである．

これによると，データ木サイズ $|D_A|$ の増加に対してほぼ線形オーダーで計算時間が増大している．頻出木パターン発見においては，出現頻度の計算において常にデータ木のサイズに比例した時間が必要となるため，計算量が線形未満になることは

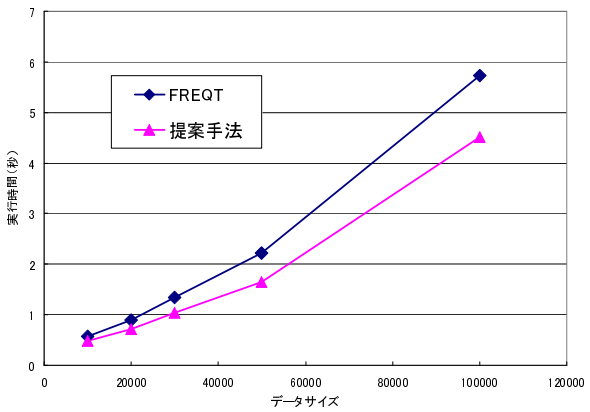

図 6　人工データ木用いた実験のアルゴリズム実行時間

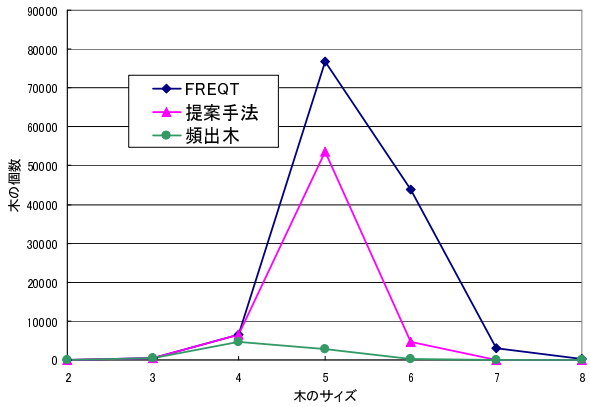

図 7　人工データ木における頻出候補木集合と頻出木集合の大きさの比較

ない．よって，このアルゴリズムは十分な規模耐性を持つと言うことができる．

また FREQT と比較すると，より短い時間で全ての頻出順序木を発見できている．その理由を検証するために，$|V| = 10000$ において FREQT と提案アルゴリズムで得られる頻出候補木集合，及び実際の頻出順序木集合の大きさを図 7 に示した．FREQT や提案手法においては，アルゴリズムの実行時間のほとんどが順序木の出現数の数え上げ操作によって占められているため，無駄な頻出候補木の生成が抑えられて出現数の計算に必要なデータ木の走査回数が少ない我々の提案手法の方が，速度的に有利となっていることがわかる．

5.4　XML を用いた実験

次に，実際の木構造データの例として XML を用いた実験を行う．実験で用いたのは，容量約 500 キロバイトから最大約 10 メガバイトの，Google の検索結果を格納した XML ファイルである．そのラベル集合の要素数は 408 であり，節点数はそれぞれ 8677 から 172932 の間である．各 XML データを読み込んでデータ木 D に変換した後，最小サポート $\sigma = 0.05\%$ として我々の頻出順序木パターン発見アルゴリズムを適用し，その実行時間を計測し，結果を図 8 に示した．

これによれば，人工データ木に対する実験と同様，データ木のサイズ n に対してほぼ線形時間で，さらに FREQT よりも高速に頻出順序木パターンを求めることが可能であることがわかる．よって，XML 等の構造化テキストの頻出パターン発見に応用することができる．

実際に我々のアルゴリズムが頻出候補木を効率的に列挙していることを確かめるために，上記の $|V| = 87853$ における実験中に，各サイズ k において生成される頻出候補木の個数 $|\mathcal{C}_k|$ と，k-パターン集合の要素数 $|\mathcal{F}_k|$ を求め，図 9 に表示した．木のサイズ別に見ると，FREQT では頻出木が多くなるほど頻出候補木の個数が著しく増加している．それに比べ提案アルゴリズムでは頻出木と頻出候補木の個数がほぼ一致し，無駄な頻出候補木の生成が抑制された効率の良い列挙が行われていることがわかる．

実験ではデータ木のサイズが数十万ノード程度だったためデータ木をメモリ上に読み込んでその上で走査を行っていた．しかし，実用上数億ノード規模のデータ木を扱わなければならない場合，データ木をメモリ上に展開することは容量的に不可能となり木構造を格納したデータベース上で出現数を計算することが必要となる．その場合，データ木の走査にかかる時間的負荷が非常に重くなるため，提案手法と FREQT の差はさらに顕著になるものと思われる．これは図 8 の 2 つのグラフの傾きの違いからも予測することができる．

人工データ木においては，FREQT に対して提案アルゴリズムが十パーセント程度高速なのに対し，XML においては 3 倍

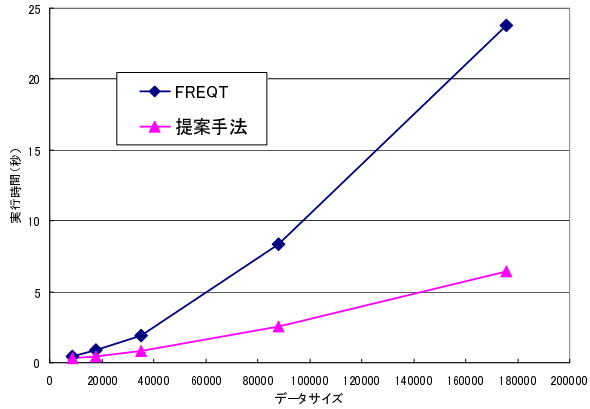

図 8　XML を用いた実験のアルゴリズム実行時間

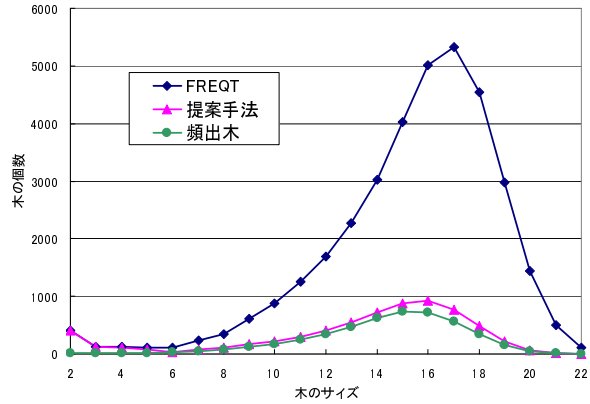

図 9　XML における頻出候補木集合と頻出木集合の大きさの比較

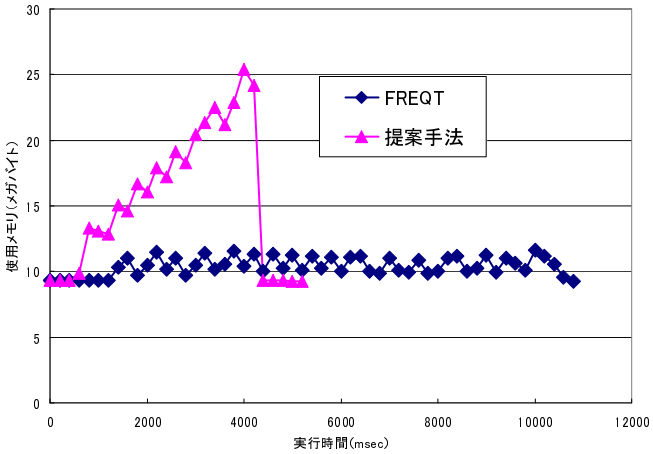

図 10　XML を用いた実験時のメモリ使用量の変化

以上高速であるという結果が得られた．これはデータの偏りに起因するものと考えられる．人工データ木はランダム関数を用いて生成されており，サイズの小さな頻出順序木が大量に出現するため，左右木結合による頻出候補木集合の絞込みがそれほど働いていない．一方で XML 中にはいくつかのサイズの大きな頻出順序木があり，その部分木が多く出現していると考えられ，このような場合には提案手法が極めて有効に機能している．一般的にデータマイニングの対象となる実データは偏りを持っていると考えられるので，提案アルゴリズムは FREQT に比べおよそ数倍程度高速に動作すると思われる．これについては他のデータ構造を用いた実験によってさらに確かめる必要がある．

次に，同じく節点数が 87853 の XML データを用いた実験におけるメモリ使用量の変化を計測し，結果を図 10 に示した．

FREQT においては実行時間を通じてメモリ使用量が低く抑えられていることがわかる．一方，提案手法においては FREQT の数倍程度のメモリを占有している．これは 5.1 節で述べたように，FREQT が効率の良い深さ優先方式であるのに対し，提案手法が原理的には幅優先方式であることが原因であると考えられる．よって，同時に探索する範囲を絞込むなどのさらなる最適化が必要である．

6. ま と め

本稿では，頻出順序木パターン発見問題に対し，左右木結合による頻出候補木列挙を用いたアルゴリズムを提案した．人工の木構造データと XML データを用いた実験によって，提案アルゴリズムは全ての頻出順序木パターンをデータサイズに対して概ね線形時間で重複無く発見できることを示した．同じ頻出順序木パターン発見アルゴリズムである FREQT との比較により，提案手法は数十パーセントから数倍高速であるという結果が得られた．

今後の課題としては，アルゴリズムの性能がデータの性質に大きく依存しているため，様々なスキーマを持つ XML データや，XML 以外の RNA の 2 次構造データなどを用いた性能評価実験を行い，どのようなデータに対しても高い性能を示すことを確かめる必要がある．また幅優先方式であるためにアルゴリズム実行時のメモリ使用量が FREQT よりも大きくなる点については，左右木結合における頻出候補木の生成に関してより詳細な解析を行い，改善を進めていきたい．さらに本稿で扱った頻出順序木パターン発見アルゴリズムは，頻出無順序木パターン発見問題に応用することが考えられる．実際，浅井らの頻出無順序木パターン発見アルゴリズム UNOT [8], [9] は FREQT が元となっている．また，木構造では無い一般のグラフ構造において，頻出する根無し木 (free tree) を発見する問題も近年研究が進んでいる [10]．よって，本稿で提案したアルゴリズムを改良し，これらの問題に適用することも考えられる．

謝　　辞

本研究の一部は，文部科学省の科学研究費補助金 (課題番号:16016248,13680482) の支援を受けた．

文　　献

[1] D.B. Searls, “The language of genes”, nature insight - computational biology, vol.420, no.6912, pp.211–217, 2002.

[2] B.A. Shapiro, and K. Zhang, “Comparing multiple rna secondary structures using tree comparisons”, Computer Applications in Biosciences, vol.6, no.4, pp.309–318, 1990.

[3] R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules”, Proceedings of the 20th International Conferece on Very Large Data Bases (VLDB’94), pp.487–499, 1994.

[4] 浅井達哉，有村博紀，“半構造データマイニングにおけるパターン発見技法”,電子情報通信学会論文誌，vol.J87-D1，no.2，pp.79–96，2004.

[5] S. Nijssen, “Frequent structure mining: Efficiency issues”, Proceedings of the 2nd International Workshop on Mining Graphs, Trees and Sequences (MGTS2004), 2004.

[6] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, “Efficient substructure discovery from large semi-structured data”, Proceedings of the 2nd International Conference on Data Mining (SIAM’02), pp.158–174, 2002.

[7] M.J. Zaki, “Efficiently mining frequent trees in a forest”, Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD’02), pp.71–80, 2002.

[8] T. Asai, H. Arimura, T. Uno, and S. Nakano, “Efficient substructure discovery from large semi-structured data”, Proceedings of the 6th International Conference on Discovery Science, pp.47–61, 2003.

[9] 房延慎二，浅井達哉，有村博紀，宇野毅明，中野眞一，“半構造データマイニングのための高速な無順序木パターン発見手法”，電子情報通信学会 DE 研究会第１５回データ工学ワークショップ (DEWS2004)，6-A-03，2004.

[10] U. Rückert, and S. Kramer, “Frequent free tree discovery in graph data”, Proceedings of the 2004 ACM Symposium on Applied Computing (SAC’04), pp.564–570, 2004.