制限付き最右拡張を用いた効率的な飽和順序木の発見

尾崎知伸[†] 大川剛直[††]

近年，構造データマイニングの分野において，大量のパターンが発見されるという頻出パターン発見の問題点を解決するための 1 つの手法として，飽和パターン発見が注目を集めている．本論文では，順序木データベースからのより効率的な飽和順序木の発見に焦点をあて，制限された最右拡張を用いた順序木の列挙とブランケットに基づく枝刈りを用いた，(1) 幅優先探索および (2) 深さ優先＋幅優先探索に基づく 2 つの新たなアルゴリズムを提案する．合成データと実データを用いた既存研究との比較実験より，提案手法において，列挙される候補木の数が削減され，その結果として多くの場合において実行時間を短縮できることが確認された．

Efficiently Mining Closed Induced Ordered Subtrees Based on the Restricted Rightmost Expansion

Tomonobu Ozaki[†] and Takenao Ohkawa[††]

In this paper, we focus on the problems of mining closed induced ordered subtrees. By combining the restricted rightmost expansion and the pruning based on the blanket, we propose two new closed ordered subtree miners based on the breadth-first and depth-first/breadth-first enumeration strategy, respectively. Through the experiments with synthesized and real world datasets, we discuss the effects of the difference of the search strategies in mining closed induced ordered subtrees.

1. はじめに

近年，頻出パターン発見を中心に，構造データを対象としたデータマイニング手法が注目され，数多くの研究が行われている[4),5)]．また，大量のパターンが発見されるという頻出パターン発見の問題点を解決するための手法として，利用者により与えられる制約を満たすパターンのみを発見するアプローチ[9),12)]や，頻出パターンの代表元である飽和パターンのみを発見するアプローチ[3),8),11),13)]などが提案されている．

本論文では，順序木データベースからの効率的な**飽和順序木発見手法**について議論する．これまでに，飽和順序木発見手法として，**最右拡張**[2),14)]と**ブランケットに基づく枝刈り**[3),13)]を用いた深さ優先探索に基づくアルゴリズム *CMTreeMiner*[3)]が提案されている．しかしこの方法は，飽和順序木になりえない候補パターン（候補木）を大量に生成するという問題を含んでいる．この問題を解決し，より効率的な飽和順序木発見を実現するために，本論文では，制限付きの最右拡張とその上でのブランケットに基づく枝刈りを用いた，(1) 幅優先探索に基づくアルゴリズム *ClootBF*，および (2) 深さ優先+幅優先探索に基づくアルゴリズム *ClootDB* を提案する．提案手法の特徴や既存手法との関係を**表 1** にまとめる．詳しくは後述するが，既存手法とは異なる探索戦略や枝刈り手法を採用することで，*ClootBF* では，メモリ使用量の増大は予想されるものの，生成される候補木数の大幅な削減が期待される．一方 *ClootDB* では，*ClootBF* ほどではないが，メモリ使用量の増大を回避したうえでの候補木数の削減が期待される．

以下に本論文の構成を示す．2 章で用語の導入を行い，3 章で議論の出発点となる既存手法について概観する．4 章で幅優先探索に基づく飽和順序木発見手法を，5 章で深さ優先＋幅優先探索に基づく飽和順序木発見手法をそれぞれ提案する．6 章で実験結果を示し，最後に 7 章でまとめを行う．

2. 準　　備

本章では，文献 2)，4)，6) などに従い，飽和順序

† 神戸大学自然科学系先端融合研究環
Organization of Advanced Science and Technology, Kobe University
†† 神戸大学大学院工学研究科
Graduate School of Engineering, Kobe University

表 1 既存手法と提案手法の比較
Table 1 Comparison between existing and proposed algorithms.

手法名	探索戦略	木の列挙方法	枝刈り手法	候補木数	メモリ使用量
CMTreeMiner (*ClootDF*)	深さ優先	最右拡張	左ブランケット刈り 右ブランケット刈り	–	–
ClootBF	幅優先	左右木結合 直列木拡張	中央ブランケット刈り 右ブランケット刈り 逆右ブランケット刈り	大幅に削減	増大
ClootDF	深さ優先＋幅優先	兄弟木結合 最右葉拡張	左ブランケット刈り 制限付き左ブランケット刈り	削減	同程度

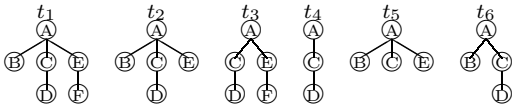

図 1 順序木の例
Fig. 1 Example of ordered trees.

木発見に関する用語を導入する．

$\mathcal{L} = \{l_0, l_1, \cdots\}$ をラベルの集合とする．$\mathcal{L}$ 上の**ラベル付き順序木**とは，五項組 $t = (V, E, S, r, L)$ で表される r を根とする木である．ここで，V は節点の集合を，辺集合 $E \subseteq V^2$ は節点間の親子関係を，$S \subseteq V^2$ は節点間の兄弟関係をそれぞれ表す．また $L : V \to \mathcal{L}$ はラベル関数である．簡略化のため，以後，ラベル付き順序木を単に木と呼ぶ．木 t の大きさ $|t|$ を $|t| = |V|$ と定義する．ちょうど 1 つの葉を持つ木を**直列木**と呼ぶ．t の最右葉を $\mathrm{rml}(t)$，最左葉を $\mathrm{lml}(t)$ と表記する．また，t から $\mathrm{lml}(t)$ を取り除いた木を $\mathrm{right}(t)$，同じく $\mathrm{rml}(t)$ を取り除いた木を $\mathrm{left}(t)$ とそれぞれ表記する．なお本論文では，$\mathrm{left}(t)$ を $p(t)$ とも表記する．また $p(t)$ を t の**親**と呼ぶ．木 t に，最左葉として節点 v を追加することで得られる木を $v \bullet t$ と表記する．同様に，木 t に最右葉として節点 v を追加することで得られる木を $t \cdot v$ と表記する．節点 $v \in V$ のラベルを $\mathrm{l}(v)$ と表記する．また節点 $v \in V$ の深さ $\mathrm{d}(v)$ を根 r から v への経路上の辺の数と定義する．木 t 中の節点を前順に並べた系列を $(v_1, \cdots, v_{|t|})$ としたとき，$(\mathrm{d}(v_1), \mathrm{l}(v_1)) \cdot (\mathrm{d}(v_2), \mathrm{l}(v_2)) \cdot \cdots \cdot (\mathrm{d}(v_{|t|}), \mathrm{l}(v_{|t|}))$ を t の**深さラベル列**[2] と呼ぶ．順序木とその深さラベル列は 1 対 1 に対応するので，以後，順序木とその深さラベル列を同一視する．

図 1 にラベル付き順序木の例を示す．t_1 の深さラベル列は $(0, A) \cdot (1, B) \cdot (1, C) \cdot (2, D) \cdot (1, E) \cdot (2, F)$ である．$\mathrm{rml}(t_1)$ は F，$\mathrm{lml}(t_1)$ は B であり，$\mathrm{left}(t_1) = t_2$，$\mathrm{right}(t_1) = t_3$ である．また t_4 は直列木であり，$t_6 = (1, B) \bullet t_4$，$t_1 = t_2 \cdot (2, F)$ である．

2 つの木 $t = (V_t, E_t, S_t, r_t, L_t)$，$s = (V_s, E_s, S_s, r_s, L_s)$ に対し，(1) $(v_1, v_2) \in E_t$ iff $(\phi(v_1), \phi(v_2)) \in E_s$，(2) $(v_1, v_2) \in S_t$ iff $(\phi(v_1), \phi(v_2)) \in S_s$，(3) $L_t(v) = L_s(\phi(v))$ を満たす 1 対 1 関数 $\phi : V_t \to V_s$ が存在する場合，t は s の部分木であると定義し，$t \prec s$ と表記する．D を順序木データベースとしたとき，t の頻度を，$sup_D(t) = |\{s \in D \mid t \prec s\}|/|D|$ と定義する．

データベース D および最小頻度 σ（$0 < \sigma \leq 1$）が与えられたとき，$sup_D(t) \geq \sigma$ を満たす順序木 t を**頻出順序木**と呼ぶ．また，大きさ k のすべての頻出順序木の集合を $\mathcal{F}_k$ と表記する．一方，$\forall s \succ t (sup_D(s) = sup_D(t) \to s = t)$ を満たす頻出順序木 t を**頻出飽和順序木**と呼ぶ．本論文では，D と σ，$\mathcal{F}_2$ が与えられたとき，大きさ 2 以上のすべての頻出飽和順序木を発見する問題について議論する．ここで，既存手法と提案手法の間で，大きさ 2 以下の候補木の列挙に関して本質的な差異はなく，議論を明確にするために，$\mathcal{F}_2$ を前提としている．

3. 深さ優先探索による飽和順序木の発見

本論文での議論の出発点として，本章では，最右拡張[2),14)] と出現マッチによる枝刈りを用いた深さ優先探索による飽和順序木発見アルゴリズム[3)] を導入する．

順序木 t から，新たな順序木の集合

$\mathrm{refine}(t) = \{t \cdot (d, l) \mid 1 \leq d \leq \mathrm{d}(\mathrm{rml}(t)) + 1, l \in \mathcal{L}\}$

を得る操作を**最右拡張**と呼ぶ．また $t_0 = (0, l)$，$l \in \mathcal{L}$ に対し，最右拡張を繰り返し適用することで，すべての順序木を重複なく列挙することが可能である[2)]．

順序木 t のブランケットを $B(t) = \{t' \mid t \prec t', |t'| = |t| + 1\}$ と定義する．データベース D と木 t に対して，

$$TM_D(t) = \{t' \in B(t) \mid sup_D(t') = sup_D(t)\ \}$$

と定義する．σ を最小頻度としたとき，定義により，$TM_D(t) = \emptyset \wedge sup_D(t) \geq \sigma$ であれば t は頻出飽和順序木である．一方，データベース D において，t の全出現を覆う（含む）形で $t' \in B(t)$ の出現が存在する[3)] とき，$OM_D(t, t')$ と表記する．**図 2** に例を示す．$D = \{t_A, t_B\}$ に対し，木 $t_x = (0, B) \cdot (1, C)$ は，t_A に 2 回，t_B に 1 回それぞれ出現する．また木

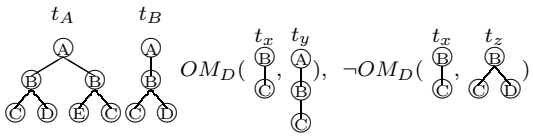

図 2 $OM_D(t, t')$ の例
Fig. 2 An example of $OM_D(t, t')$.

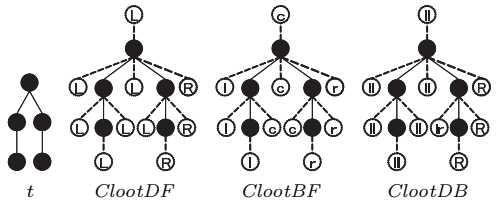

図 3 各アルゴリズムにおける部分木のブランケット：部分木 t にラベル X である節点を加えた木が $OM_D^X(t)$ に含まれる
Fig. 3 Blanket of a subtree: A tree obtained by adding a vertex labeled as X to t is in $OM_D^X(t)$.

$t_y = (0, A) \cdot (1, B) \cdot (2, C) \in B(t_x)$ は，t_x のすべての出現を覆う形で出現するので，$OM_D(t_x, t_y)$ が成り立つ．一方，木 $t_z = (0, B) \cdot (1, C) \cdot (1, D) \in B(t_x)$ は，t_A，t_B の双方に出現するが，t_A 中の t_x の出現を 1 つ含まないので，$OM_D(t_x, t_z)$ とはならない．

2 つの順序木 t と t' に対して $OM_D(t, t')$ であれば，$\forall s \succ t\ \exists s' \succ t'$ s.t. $OM_D(s, s')$ が成り立つ[3]．したがって，$s \succ t$，$s \not\succ t'$ である木 s は飽和順序木にはなりえないので枝刈りの対象となる．最右拡張を基本とした探索に枝刈りを導入するために，データベース D と木 t に対して，以下の 3 つの集合を定義する（**図 3** の *ClootDF* 参照）．

$$OM_D(t) = \{t' \in B(t) \mid OM_D(t, t')\}$$
$$OM_D^R(t) = OM_D(t) \cap \mathrm{refine}(t)$$
$$OM_D^L(t) = OM_D(t) \setminus OM_D^R(t)$$

最右拡張に基づく順序木の列挙では，t と $t' \in OM_D^L(t)$ に対し，$s \succ t$ は $s' \succ t'$ の生成に影響を与えないので，t を枝刈りすることが可能である．このように，$OM_D^L(t) \neq \emptyset$ なる木 t 自身を候補から排除する枝刈りを，**左ブランケット刈り**[3] と呼ぶ．一方，t と $t' \in OM_D^R(t)$ および $\mathrm{d}(v) < \mathrm{d}(\mathrm{rml}(t'))$ である節点 v に対し，$t' \cdot v \in OM_D^L(t \cdot v)$ が成り立つ．したがって，$t \cdot v$ には左ブランケット刈りが適用されるので，その生成は無意味である．このように，$OM_D^R(t) \neq \emptyset$ である t を利用し，$OM_D^L(t \cdot v) \neq \emptyset$ なる木 $t \cdot v$ の生成を回避する枝刈りを，**右ブランケット刈り**[3] と呼ぶ．右ブランケット刈りは，条件を満たす木の生成を回避するので，最右拡張に制限を加えるものであるが，その目的は飽和順序木になりえない木の排除である．本論文では，頻出順序木になりえない木の排除をともなう最右拡張を制限付き最右拡張，飽和順序木になりえない木の生成を回避する操作を枝刈りと呼び，両者を区別する．

Algorithm *ClootDF*($\mathcal{F}_2$, D, σ)
1: **for each** $t \in \mathcal{F}_2$
2: *ClootDF*-Enum(t, D, σ)
3: **end for**

Subroutine *ClootDF*-Enum(t, D, σ)
1: **if** $OM_D^L(t) \neq \emptyset$ **then** return
2: **if** $TM_D(t) = \emptyset$ **then** output t
3: **if** $OM_D^R(t) = \emptyset$ **then** $m := 1$
4: **else** $m := max\{\mathrm{d}(\mathrm{rml}(t')) \mid t' \in OM_D^R(t)\}$
5: **for each** $d (m \leq d \leq \mathrm{d}(\mathrm{rml}(t)) + 1)$
6: **for each** $l \in \mathcal{L}$
7: $s := t \cdot (d, l)$
8: **if** $sup_D(s) \geq \sigma$ **then**
9: *ClootDF*-Enum(t, D, σ)
10: **end for**
11: **end for**

図 4 飽和順序木発見アルゴリズム *ClootDF*
Fig. 4 Pseudo code of *ClootDF*.

以上の準備のもと，最右拡張を用いた深さ優先探索による頻出飽和順序木発見アルゴリズム *ClootDF* を**図 4** に示す．なおこのアルゴリズムは，文献 3) で示されたアルゴリズムを，大きさ 2 以上の飽和順序木発見に特化させたものである．*ClootDF*-Enum 中の 1 行目が左ブランケット刈りに，3〜5 行目が右ブランケット刈りに，それぞれ相当する．また，木 t を出力する前に $TM_D(t)$ を確認することで，飽和木以外の木の出力を回避している．

4. 幅優先探索による飽和順序木の発見

4.1 制限付き最右拡張に基づく頻出順序木の列挙

これまでに，アプリオリアルゴリズム[1] の自然な拡張として，幅優先探索に基づく頻出順序木発見アルゴリズム AMIOT[6] が提案されている．

AMIOT では，$\mathrm{right}(t) = \mathrm{left}(s)$ なる 2 つの頻出順序木 t, s から，新たな候補 $t \cdot \mathrm{rml}(s) = \mathrm{lml}(t) \bullet s$ を生成する．この拡張を**左右木結合**と呼ぶ．また t が直列木の場合，新たな候補の集合 $C = \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$ を生成する．この拡張を**直列木拡張**と呼ぶ．左右木結合および直列木拡張により t から生成される候補木は，いずれも t の最右枝上に新たな節点を加えたものであるので，これらは制限付きの最右拡張であると

Algorithm AMIOT($\mathcal{F}_2$, D, σ)
1: BF-Enum($\mathcal{F}_2$, D, σ)
Subroutine BF-Enum(F, D, σ)
1: **for each** $t \in F$
2: output t
3: **end for**
4: $C := \emptyset$
5: **for each** $t, s \in F$ s.t. right(t) = left(s)
6: $C := C \cup \{t \cdot \mathrm{rml}(s)\}$
7: **end for**
8: **for each** $t \in F$ s.t. t is serial
9: $C := C \cup \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$
10: **end for**
11: $F' := \{c \in C \mid sup_D(c) \geq \sigma\}$
12: **if** $F' \neq \emptyset$ **then** BF-Enum(F', D, σ)

図 5 頻出順序木発見アルゴリズム AMIOT
Fig. 5 Pseudo code of AMIOT.

いえる．また s から生成される候補木は，いずれも s の最左枝上に節点を加えたものであるので，逆方向への制限付き最右拡張，すなわち制限付きの最左拡張であるといえる．**図 5** に，$\mathcal{F}_2$ を入力とする AMIOT のアルゴリズムを示す．図中において，5〜7 行目が左右木結合に，8〜10 行目が直列木拡張に対応する．

4.2 *ClootBF*：幅優先探索による飽和順序木の発見

AMIOT による制限付きの最右拡張（左右木結合と直列木拡張）を利用することで，深さ優先探索を利用した場合と比べ，頻出順序木の候補数を減らすことが可能となる．しかしその一方で，飽和木の発見を考えた場合，ブランケットに基づく枝刈りについても別の条件を与える必要がある．例として，図 1 中の 3 つの木 t_1，t_2，t_3 を考える．データベースを $D = \{t_1\}$ としたとき，$t_1 \in OM^L_D(t_3)$ である．しかし，飽和木である t_1 の生成には t_2，t_3 が必要とされるので，t_3 に対し左ブランケット刈りを適用することは適切ではない．

$OM_D(t)$（$|t| \geq 2$）を 3 つに分け（図 3 の *ClootBF* 参照），ブランケットに基づく枝刈りの適用可能性について議論する．

$$OM^l_D(t) = \{t' \in OM^L_D(t) \,|\, \mathrm{right}(t') = t\}$$
$$OM^c_D(t) = OM^L_D(t) \setminus OM^l_D(t)$$
$$OM^r_D(t) = \{t' \in OM^R_D(t) \,|\, t'\text{は直列木でない}\}$$

Case1 $OM^c_D(t) \neq \emptyset$**：**以下に示す補題 1 より，t に対し左ブランケット刈りを適用することが可能である．

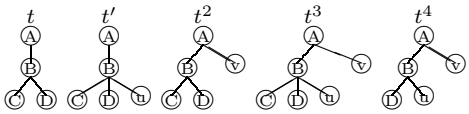

図 6 部分木と右ブランケット刈り
Fig. 6 Subtrees for right-blanket pruning.

すなわち，左ブランケット刈りが適用できる条件は，$OM^L_D(t) \neq \emptyset$ ではなく $OM^c_D(t) \neq \emptyset$ となる．この枝刈りを**中央ブランケット刈り**と呼ぶ．

補題 1 $OM^c_D(t) \neq \emptyset$ なる木 t が与えられたとき，t を枝刈りすることは，$t' \in OM^c_D(t)$ および t' に最右拡張を適用して得られる任意の木 $t' \cdot v_1 \cdots v_n$ の生成を妨げない．

証明 $t' \in OM^c_D(t)$ は，$\mathrm{right}(t') \not> t$ と $\mathrm{left}(t') \not> t$ から生成される．一方 $t' \cdot v_1$ は，t' と $\mathrm{right}(t') \cdot v_1$ とから生成されるが，この 2 つの木の生成に必要とされる 3 つの木 $\mathrm{left}(t')$，$\mathrm{right}(t')$，$\mathrm{right}(\mathrm{right}(t')) \cdot v_1$ は，いずれも t と等しくない．この関係を帰納的に適用することで，t の枝刈りが $t' \cdot v_1 \cdots v_n$ の生成に影響を与えないことが導出される． □

Case2 $OM^r_D(t) \neq \emptyset$**：**以下に示す補題 2 より，t に対し右ブランケット刈りを適用することが可能である．

補題 2 2 つの木 t と $t' = t \cdot u \in OM^r_D(t)$ に対し，$t^2 = t \cdot v$ s.t. $\mathrm{d}(u) > \mathrm{d}(v)$ を枝刈りすることは，$t^3 = t \cdot u \cdot v$ の生成を妨げない（**図 6** 参照）．

証明 $t^3 = t \cdot u \cdot v$ の生成に必要とされる 2 つの木 t'，$t^4 = \mathrm{right}(t) \cdot u \cdot v$ はいずれも t^2 と等しくない（図 6 参照）． □

Case3 $OM^l_D(t) \neq \emptyset$**：**$t' \in OM^l_D(t)$ を生成するためには，$\mathrm{left}(t')$ と $\mathrm{right}(t')$ が必要となる．しかし $\mathrm{right}(t') = t$ であるので，t に対して左ブランケット刈りを適用することはできない．

その一方で，以下に示す補題 3 により，t に対し，異なる形式での右ブランケット刈りを適用することが可能である．この枝刈りを**逆右ブランケット刈り**と呼ぶ．

補題 3 2 つの木 t と $t' = u \bullet t \in OM^l_D(t)$ に対し，$t^2 = v \bullet t$ s.t. $\mathrm{d}(u) > \mathrm{d}(v)$ を枝刈りすることは，$t^3 = v \bullet u \bullet t$ の生成を妨げない．

証明 $t^3 = v \bullet u \bullet t$ の生成に必要とされる 2 つの木 $u \bullet t$ と $v \bullet u \bullet p(t)$ は，いずれも t^2 と等しくない． □

上記の議論に基づき，AMIOT における制限付きの最右拡張とブランケットに基づく枝刈りを用いた，幅優先探索による頻出飽和順序木発見アルゴリズム *ClootBF* を提案する（**図 7**）．ここで，*ClootBF*-Enum 中の 1 行目が中央ブランケット刈りに，7〜12 行目が右ブランケット刈りと逆右ブランケット刈りに相当する．

Algorithm *ClootBF*($\mathcal{F}_2$, D, σ)
1: *ClootBF*-Enum($\mathcal{F}_2$, D, σ)
Subroutine *ClootBF*-Enum(F, D, σ)
1: $F' := F \setminus \{t \in F \mid OM^c_D(t) \neq \emptyset\}$
2: **for each** $t \in F'$
3: **if** $TM_D(t) = \emptyset$ **then** output t
4: **end for**
5: $C := \emptyset$
6: **for each** $t, s \in F'$ s.t. right(t) = left(s)
7: **if** $OM^r_D(t) = \emptyset$ **then** $m := 1$
8: **else** $m := max\{\mathrm{d}(\mathrm{rml}(t')) \mid t' \in OM^r_D(t)\}$
9: **if** $OM^l_D(t) = \emptyset$ **then** $n := 1$
10: **else** $n := max\{\mathrm{d}(\mathrm{lml}(t')) \mid t' \in OM^l_D(t)\}$
11: **if** $m \leq \mathrm{d}(\mathrm{rml}(s)) \wedge n \leq \mathrm{d}(\mathrm{lml}(t))$ **then**
12: $C := C \cup \{t \cdot \mathrm{rml}(s)\}$
13: **end for**
14: **for each** $t \in F'$ s.t. t is serial
15: $C := C \cup \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$
16: **end for**
17: $F'' := \{c \in C \mid sup_D(c) \geq \sigma\}$
18: **if** $F'' \neq \emptyset$ **then** *ClootBF*-Enum(F'', D, σ)

図 7 頻出飽和木発見アルゴリズム *ClootBF*
Fig. 7 Pseudo code of *ClootBF*.

ClootBF は制限付きの最右拡張を採用しているので，頻出木 t が与えられたとき，そこから生成される候補木の数は *ClootDF* 以下となることは明らかである．加えて，$OM^L_D(t) = \emptyset$，$OM^c_D(s) \neq \emptyset$，right(t) = left(s) なる 2 つの頻出木 t，s に対し，*ClootDF* では $t \cdot \mathrm{rml}(s)$ を生成した後，$OM^L_D(t \cdot \mathrm{rml}(s)) \neq \emptyset$ により左ブランケット刈りが適用されることとなるが，*ClootBF* では，s に対し枝刈りが適用されるので $t \cdot \mathrm{rml}(s)$ の生成自体が行われない．一方，$OM^l_D(t) \neq \emptyset$，$OM^c_D(t) = \emptyset$ の場合，*ClootDF* では枝刈りされる木 t が，*ClootBF* では枝刈りされない．このように，*ClootDF* と *ClootBF* の枝刈り手法は，互いに枝刈りが可能である場合とそうでない場合が存在する．

ClootBF に対し，以下の定理が成立する．

定理 1 $\mathcal{F}_2$ を入力としたとき，*ClootBF* は，大きさ 2 以上のすべての頻出飽和順序木を漏れなくかつ重複なく列挙する．

証明 AMIOT の列挙方法と，補題 1，補題 2，補題 3 から明らか． □

定理 1 は，*ClootBF* において，AMIOT の制限付き最右拡張に影響を与えることなく，ブランケットに基づく枝刈りが導入されていることを意味する．すなわち *ClootBF* では，制限付き最右拡張により頻出順序木になりえない候補木の生成が抑制され，さらに，ブランケットに基づく枝刈りにより飽和順序木になりえない候補木の排除が達成される．これらの結果として，単純な最右拡張に基づく既存手法と比較し，生成される候補木数の削減と実行時間の改善が期待できる．

Algorithm FreqTDB($\mathcal{F}_2$, D, σ)
1: **while** $\mathcal{F}_2 \neq \emptyset$
2: select one tree t from $\mathcal{F}_2$
3: $F := \{s \in \mathcal{F}_2 \mid p(t) = p(s)\}$
4: DFBF-Enum(F, D, σ)
5: $\mathcal{F}_2 := \mathcal{F}_2 \setminus F$
6: **end while**
Subroutine DFBF-Enum(F, D, σ)
1: **for each** $t \in F$
2: output t
3: $C := \emptyset$
4: **for each** $s \in F$ s.t. d(rml(s)) $\leq$ d(rml(t))
5: $C := C \cup \{t \cdot \mathrm{rml}(s)\}$
6: **end for**
7: $C := C \cup \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$
8: $F' := \{c \in C \mid sup_D(c) \geq \sigma\}$
9: **if** $F' \neq \emptyset$ **then** DFBF-Enum(F', D, σ)
10: **end for**

図 8 頻出順序木発見アルゴリズム FreqTDB
Fig. 8 Pseudo code of FreqTDB.

5. 深さ優先＋幅優先探索による飽和順序木の発見

5.1 制限付き最右拡張に基づく頻出順序木の列挙

これまでに，深さ優先探索と幅優先探索の組合せに基づく，頻出埋め込み木発見アルゴリズム *TreeMiner* [14] が提案されている．本節では，*TreeMiner* における列挙方法を基に，制限付き最右拡張を用いた，頻出順序木発見アルゴリズムを提案する．

図 8 に，$\mathcal{F}_2$ を入力とする提案アルゴリズム FreqTDB を示す．FreqTDB では，頻出木 t と木の集合 $T = \{s \mid p(s) = p(t)\}$ から，新たな候補木の集合

$C = C_1 \cup C_2$
$C_1 = \{t \cdot \mathrm{rml}(s) \mid s \in T, \mathrm{d}(\mathrm{rml}(s)) \leq \mathrm{d}(\mathrm{rml}(t))\}$
$C_2 = \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$

を生成する（DFBF-Enum の 4〜7 行目）．ここで，C_1 を得る操作を**兄弟木結合**，C_2 を得る操作を**最右葉拡張**と呼ぶ．また，C 中のすべての候補木 t' の親 $p(t')$ は t であるので，その生成方法は制限付きの最右拡

張と見なすことができる．したがって生成される候補木の数は，単純な最右拡張を用いた場合以下となる．また FreqTDB では，ある頻出木を親に持つすべての頻出木が 1 度に導出されることになる（DFBF-Enum の 8 行目）ので，その列挙の順序は，深さ優先＋幅優先探索に基づくととらえられる．

FreqTDB に対し，以下の定理が成り立つ．

定理 2 $\mathcal{F}_2$ を入力としたとき，FreqTDB は，大きさ 2 以上のすべての頻出順序木を漏れなくかつ重複なく列挙する．

証明 FreqTDB における候補木の列挙方法は，制限付きの最右拡張であるので，部分木の重複列挙は起こらない．一方，候補木 $t \cdot v$ を生成するためには，2 つの頻出木 t および $p(t) \cdot v$ が同じ集合 F に含まれる必要がある．ここで，t および $p(t) \cdot v$ の親はいずれも $p(t)$ である．また，最右拡張においては，$p(t)$ を親に持つ木は，$p(t)$ を拡張することでしか得られず，また $p(t)$ はちょうど 1 度しか生成されないので，$p(t)$ とそれを親に持つ頻出木の集合は 1 対 1 の関係にある．したがって，$p(t)$ を親に持つすべての頻出木は同じ集合に含まれる．結果として，列挙の完全性が保たれる． □

5.2 *ClootDB*：深さ優先＋幅優先探索による飽和順序木の発見

幅優先探索の場合と同様，飽和順序木の発見に FreqTDB による制限付き最右拡張（兄弟木結合と最右葉拡張）を用いる場合，ブランケットに基づく枝刈りについても別の条件を与える必要がある．例として，図 1 中の 3 つの木 t_2，t_5，t_6 を考える．データベースを $D = \{t_2\}$ としたとき，$t_2 \in OM^L_D(t_5)$ である．しかし，飽和木である t_2 の生成には t_5，t_6 が必要とされるので，t_5 に対し左ブランケット刈りを適用することは適切ではない．

$OM^L_D(t)$（$|t| \geq 2$）を 2 つに分け（図 3 の *ClootDB* 参照），ブランケットに基づく枝刈りに対し，どのような制限が必要かを議論する．

$$OM^{lr}_D(t) = \{t' \in OM^L_D(t) \mid p(p(t')) = p(t)\}$$
$$OM^{ll}_D(t) = OM^L_D(t) \setminus OM^{lr}_D(t)$$

Case1 $OM^{ll}_D(t) \neq \emptyset$**：**以下に示す補題 4 より，$t$ に対し左ブランケット刈りを適用することが可能である．

補題 4 $OM^{ll}_D(t) \neq \emptyset$ なる木 t に対し，t の枝刈りは，$t' \in OM^{ll}_D(t)$ および t' に最右拡張を適用して得られる任意の木 $t' \cdot v_1 \cdots v_n$ の生成に影響を与えない．

証明 $t' \in OM^{ll}_D(t)$ は，2 つの部分木 $p(t') \not\succ t$，$p(p(t')) \cdot \mathrm{rml}(t') \not\succ t$ から生成される．一方 $t' \cdot v_1$ は，

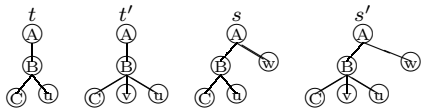

図 9 部分木と制限付き左ブランケット刈り
Fig. 9 Subtrees for restricted left-blanket pruning.

t' と $p(t') \cdot v_1$ とから生成されるが，この 2 つの木の生成に必要とされる 3 つの木 $p(t')$，$p(p(t')) \cdot rml(t')$，$p(p(t')) \cdot v_1$ は，いずれも t と等しくない．この関係を帰納的に適用することで，t の枝刈りが $t' \cdot v_1 \cdot v_n$ の生成に影響を与えないことが導出される． □

Case2 $OM^{lr}_D(t) \neq \emptyset$**：**$t' \in OM^{lr}_D(t) = p(t) \cdot u \cdot v$ とする．このとき，t' の生成には，$p(t) \cdot u$ と $p(t) \cdot v = t$ が必要とされるので，t に対して左ブランケット刈りを適用することはできない．一方，t そのものを枝刈りすることはできないが，以下に示す補題 5 より，t を親に持つ木を探索空間から除外することが可能である．すなわち，$OM^{ll}_D(t) = \emptyset$，$OM^{lr}_D(t) \neq \emptyset$ なる木 t に対して，特殊な形で枝刈りを適用する．この枝刈りを，**制限付き左ブランケット刈り**と呼ぶ．

補題 5 $OM^{lr}_D(t) \neq \emptyset$，$s = t \cdot w$ なるすべての木 s に対して，左ブランケット刈りを適用することが可能である．

証明 t の最右拡張を考える（**図 9** 参照）．$t' \in OM^{lr}_D(t)$ より，木 $s = t \cdot w = p(t) \cdot u \cdot w$ に対し，$s' = t' \cdot w = p(t) \cdot v \cdot u \cdot w \in OM^{ll}_D(s)$ が存在する．また，s' の生成には，s は利用されない． □

Case3 $OM^R_D(t) \neq \emptyset$**：**$t' \in OM^R_D(t) = t \cdot u$ を考える（図 6 参照）．右ブランケット刈りを適用するためには，$t^2 = t \cdot v$（$\mathrm{d}(u) > \mathrm{d}(v)$）の枝刈りが，$t^3 = t \cdot u \cdot v$ の生成に影響を与えないことが必要とされる．しかし，t^3 は t' と t^2 から生成されるので，右ブランケット刈りを適用することはできない．

以上の議論に基づき，FreqTDB における制限付きの最右拡張とブランケットに基づく枝刈りを用いた，深さ優先＋幅優先探索による頻出飽和順序木発見アルゴリズム *ClootDB* を提案する（**図 10**）．*ClootDB* では，$OM^{ll}_D(t) \neq \emptyset$ なる木（*ClootDB*-Enum の 1 行目）と，$OM^{lr}_D(t) \neq \emptyset$ なる木（3 行目）が，それぞれ枝刈りされる．その一方で，右ブランケット刈りは適用されない．*ClootDB* は，制限付きの最右拡張を採用しているので，頻出木 t が与えられたとき，そこから生成される候補木の数は *ClootDF* のそれ以下となる．加えて，$OM^L_D(t) = \emptyset$，$OM^{ll}_D(s) \neq \emptyset$，$p(t) = p(s)$ なる 2 つの頻出木 t，s に対し，*ClootDF* では $t \cdot \mathrm{rml}(s)$ を生成した後 $OM^L_D(t \cdot \mathrm{rml}(s)) \neq \emptyset$ により左ブラン

Algorithm *ClootDB*($\mathcal{F}_2$, D, σ)

1: **while** $\mathcal{F}_2 \neq \emptyset$
2: select one tree t from $\mathcal{F}_2$
3: $F := \{s \in \mathcal{F}_2 \mid p(t) = p(s)\}$
4: *ClootDB*-Enum(F, D, σ)
5: $\mathcal{F}_2 := \mathcal{F}_2 \setminus F$
6: **end while**

Subroutine *ClootDB*-Enum(F, D, σ)

1: $F' := F \setminus \{t \in F \mid OM_D^{ll}(t) \neq \emptyset\}$
2: **for each** $t \in F'$
3: **if** $OM_D^{lr}(t) \neq \emptyset$ **then** continue
4: **if** $TM_D(t) = \emptyset$ **then** output t
5: $C := \emptyset$
6: **for each** $s \in F'$ s.t. $\mathrm{d}(\mathrm{rml}(s)) \leq \mathrm{d}(\mathrm{rml}(t))$
7: $C := C \cup \{t \cdot \mathrm{rml}(s)\}$
8: **end for**
9: $C := C \cup \{t \cdot (\mathrm{d}(\mathrm{rml}(t)) + 1, l) \mid l \in \mathcal{L}\}$
10: $F'' := \{c \in C \mid sup_D(c) \geq \sigma\}$
11: **if** $F'' \neq \emptyset$ **then** *ClootDB*-Enum(F'', D, σ)
12: **end for**

図 10 頻出飽和順序木発見アルゴリズム *ClootDB*
Fig. 10 Pseudo code of *ClootDB*.

ケット刈りが適用されることとなるが，*ClootDB* では，s に対し枝刈りが適用されるので $t \cdot \mathrm{rml}(s)$ の生成自体が行われない．その一方，$OM_D^R(t) \neq \emptyset$, $OM_D^L(t) = \emptyset$ の場合，*ClootDF* では生成されない木 $t' = t \cdot v$ $(\mathrm{d}(v) < max\{\mathrm{d}(\mathrm{rml}(t'')) \mid t'' \in OM_D^R(t)\})$ が，*ClootDB* では生成されることとなる．しかし実際には，t' に対し $OM_D^{lr}(t') \neq \emptyset$ が成り立つので，制限付き左ブランケット刈りが適用される．

ClootDB に対し，以下の定理が成り立つ．

定理 3 $\mathcal{F}_2$ を入力としたとき，*ClootDB* は，大きさ 2 以上のすべての頻出飽和順序木を漏れなくかつ重複なく列挙する．

証明 定理 2 と補題 4，補題 5 より明らか． □

上記の定理は，*ClootDB* において，制限付き最右拡張を用いた深さ優先+幅優先探索に，ブランケットに基づく枝刈りが自然な形で導入されていることを意味する．これにより，制限付き最右拡張による非頻出な候補木生成の抑制と，ブランケットに基づく枝刈りによる非飽和順序木の排除が達成され，既存手法と比較し，候補木数の削減と実行時間の改善が期待できる．

6. 評価実験

提案手法の有効性を評価するため，Java 言語を用いて既存手法 *ClootDF* と提案手法 *ClootBF*，*ClootDB* を実装し，評価実験を行った．なお実装による差異を軽減するため，候補木の頻度計算や $TM_D(t)$，$OM_D(t)$ の計算[3] など，各実装間で可能な限り共通のモジュールを利用している．また各実装において，最右拡張の最適化手法の 1 つである辺スキップ（edge-skip）[2] を導入している．

表 2 実験で利用したデータセットの概要
Table 2 Overview of datasets used in experiments.

	$\lvert D\rvert$	Size	Depth	$\lvert\mathcal{L}\rvert$
D_0	50,000	3.22 / 70	1.00 / 8	10
D_1	50,000	11.13 / 106	5.50 / 10	10
Glycan	10,951	6.52 / 54	3.71 / 25	875
CSLOGS	59,691	12.93 / 428	3.43 / 85	13,355
Treebank	52,851	43.73 / 398	9.52 / 34	218

$\lvert D\rvert$ Number of trees in the database
Size Average / Maximal number of nodes per tree
Depth Average / Maximal height per tree
$\lvert\mathcal{L}\rvert$ Number of labels

評価には，Tree Generator [14] を利用して生成した 2 つの合成データ D_0，D_1 と，糖鎖データ（Glycan）[7]，Web アクセス履歴データ（CSLOGS）[14]，構文木データ（Treebank）[10] の 3 つの実データを用いた．データセットの概要を**表 2** に示す．詳細は各文献を参照されたい．

実験では，最小頻度 σ を徐々に下げながら，$\mathcal{F}_2$ が与えられた後の実行時間，生成された候補木の数，メモリ使用量を計測した．またすべての実験は，Linux マシン（CPU: Intel(R) Xeon(TM) 2.8 GHz，メインメモリ 2 GB）上で行った．実験結果を**表 3** に示す．表中の実行時間（execution time）において，太字は各設定における最速値を，下線は既存手法である *ClootDF* より遅い場合を表す．なおメモリ不足により，Treebank データの頻度 3%以下では，*ClootBF* の実行は不可能であった．

ClootBF は，合成データと Glycan データ，高頻度設定における Treebank データにおいて，既存手法より高速に動作している．特に Glycan データを対象とした実験では，生成された候補木数の大幅な減少に成功し，結果として実行時間の顕著な改善が認められた．しかし CSLOGS データに対しては，候補木数の減少の割合が小さいため，既存手法に対する実行時間の改善が見られなかった．また幅優先探索の採用に起因し，他のアルゴリズムと比較し，多くのメモリを必要としている．一方，生成された候補木数に関しては，実行可能であったすべての場合において，*ClootBF* が最小であった．このことから，*ClootBF* では，基となった

表 3 実験結果：実行時間，使用メモリ量，生成された候補木の数の比較
Table 3 Experimental results: execution time, memory usage and number of candidate subtrees.

Results of D_0

	execution time (sec.)			memory usage (MB.)			# of candidates		
σ (%)	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*
5	0.63	0.61	**0.56**	35.36	37.68	34.56	101	65	65
3	1.03	**0.86**	0.87	44.27	44.62	45.01	307	138	154
1	2.22	2.14	**1.64**	54.89	57.39	53.75	2,700	785	1,105
0.5	4.39	3.28	**3.10**	60.32	69.58	59.93	11,301	2,602	4,452
0.1	7.82	6.33	**5.00**	59.70	106.91	59.33	86,090	11,921	29,032

Results of D_1

	execution time (sec.)			memory usage (MB.)			# of candidates		
σ (%)	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*
5	32.39	**20.34**	23.73	285.03	280.68	286.36	4,710	985	1,883
3	44.97	**26.12**	28.24	303.98	281.91	298.20	10,937	1,668	3,459
1	70.15	**40.72**	42.74	290.78	357.83	311.45	47,817	5,642	15,843
0.5	86.37	**51.71**	52.28	298.26	342.60	308.71	106,422	10,531	35,435
0.1	118.87	78.75	**73.21**	294.92	457.89	318.16	595,050	47,586	202,741

Results of 'Glycan'

	execution time (sec.)			memory usage (MB.)			# of candidates		
σ (%)	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*
1	4.09	**2.48**	2.73	26.50	49.83	27.63	18,640	2,144	5,319
0.5	5.86	**3.06**	3.60	28.96	56.47	26.87	52,574	4,525	13,133
0.4	6.36	**3.20**	3.68	29.21	57.86	28.34	69,295	5,818	17,070
0.3	7.24	**3.46**	4.14	29.32	61.61	29.72	101,199	8,126	25,000
0.2	7.93	**4.24**	4.31	27.99	63.97	28.45	148,029	12,752	37,443
0.1	11.01	**5.08**	5.45	29.08	70.20	29.89	377,625	29,669	94,449

Results of 'CSLOGS'

	execution time (sec.)			memory usage (MB.)			# of candidates		
σ (%)	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*
1	0.54	0.55	**0.47**	132.19	133.84	132.48	826	586	612
0.5	1.99	2.11	**1.63**	141.34	145.68	144.44	6,878	4,984	5,043
0.4	2.87	3.06	**2.24**	148.46	154.36	143.77	11,810	8,440	8,561
0.3	4.29	4.76	**3.25**	148.05	169.75	146.84	23,571	15,339	15,537
0.2	20.97	19.91	**16.59**	669.31	715.77	666.86	73,217	30,066	31,981
0.1	201.20	295.29	**166.02**	756.79	1,017.44	712.67	803,258	144,295	167,720

Results of 'Treebank'

	execution time (sec.)			memory usage (MB.)			# of candidates		
σ (%)	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*	*ClootDF*	*ClootBF*	*ClootDB*
40	28.06	**20.57**	26.20	565.46	617.26	573.91	725	305	429
30	49.48	**29.09**	32.99	668.46	742.55	642.37	1,619	546	826
20	96.31	**54.35**	61.34	694.12	842.19	726.94	5,275	1,328	2,211
10	299.89	1,548.09	**192.15**	751.90	1,325.91	887.81	34,135	4,534	11,009
5	779.23	10,342.70	**297.76**	948.51	1,492.74	910.22	154,376	13,435	39,875
3	1,347.76	–	**644.75**	919.47	–	939.89	446,759	–	106,282
1	4,116.23	–	**1,938.80**	931.70	–	977.88	4,531,325	–	901,349

AMIOT における効率的な制限付き最右拡張を阻害しない形で枝刈り手法が導入されていると考えることができる．またそれと同時に，多くの場合において，幅優先探索を導入したことによる枝刈りに対する負の影響（$OM^l_D(t) \neq \emptyset$，$OM^c_D(t) = \emptyset$ では枝刈りが適用できない）が，それほど問題とならないことを示唆している．しかしその一方で，候補木数の減少に対して実行時間の改善が顕著でないのは，後述する左右木結合におけるオーバヘッドによるものであると考えられる．

ClootDB は，多くの場合で *ClootBF* より高速であり，またすべての場合で既存手法に対する実行効率の改善が確認された．特に Treebank データの頻度 5%以下では，既存手法より 2 倍以上の高速化を達成している．メモリ使用量に関しても，既存手法と大きな差はなく，比較的大規模な問題に対しても，安定して良い性能を発揮している．一方，*ClootBF* ほど顕著ではないが，生成された候補木の数に関しても，その削減に成功している．このことは *ClootBF* 同様，制限付きの最右拡張の利点を阻害しない形で枝刈り手法が導入されていることを示している．また *ClootDB* は，*ClootBF*

より多くの候補木を生成しながらもなお，*ClootBF* よりも高速である場合が多い．これは，最右拡張に対する制限の違いによるものであると考えられる．すなわち *ClootBF* では，同じ大きさを持つ頻出木を対象に左右木結合の相手を検索する必要があり，頻出木の数が多くなると，比較的大きなオーバヘッドが生じてしまう．これに対し *ClootDB* では，結合相手となる同じ親を持つ頻出木の集合を，アルゴリズム中で特別な操作なしに得ることができ，そのオーバヘッドは小さい．この違いにより，候補木数の削減の効果がより直接的に実行時間に反映されることとなり，実行効率の改善という観点で，*ClootBF* より良い結果が得られていると考えられる．

7. ま と め

本論文では，頻出飽和順序木の発見問題に対し，パターン列挙とその上での枝刈り戦略について考察し，幅優先探索に基づく *ClootBF*，深さ優先＋幅優先探索に基づく *ClootDB* の 2 つのアルゴリズムを提案した．また，合成データおよび実データを用いた実験を通じ，両者の違いを考察するとともに，その有効性を示した．

今後の課題としては，無順序木やグラフなど，より複雑な構造データを対象とした飽和パターン発見への提案手法の適用などがあげられる．

参 考 文 献

1) Agrawal, R. and Srikant, R.: Fast algorithms for mining association rules, *Proc. 20th International Conference on Very Large Data Bases*, pp.487–499 (1994).
2) Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H. and Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data, *Proc. 2nd Annual SIAM Symposium on Data Mining*, pp.158–174 (2002).
3) Chi, Y., Xia, Y., Yang, Y. and Muntz, R.R.: Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees, *IEEE Trans. Knowledge and Data Engineering*, Vol.17, No.2, pp.190–202 (2005).
4) Chi, Y., Nijssen, S., Muntz, R.R. and Kok, J.N.: Frequent Subtree Mining—An Overview, *Fundamenta Informaticae*, *Special Issue on Graph and Tree Mining*, Vol.66, No.1-2, pp.161–198 (2005).
5) De Raedt, L., Washio, T. and Kok, J.N. (Eds.): *Advances in Mining Graphs, Trees and Sequences*, Vol.124, Frontiers in Artificial Intelligence and Applications, IOS Press (2005).
6) 比戸将平，河野浩之：頻出順序木の高速なマイニングアルゴリズム，電子情報通信学会論文誌，Vol.J89-D, No.2, pp.163–171 (2006).
7) Kanehisa, M., Goto, S., Kawashima, S., Okuno, Y. and Hattori, M.: The KEGG Resource for Deciphering the Genome, *Nucleic Acids Research*, Vol.32, pp.D277–D280 (2004).
8) Shiozaki, H., Ozaki, T. and Ohkawa, T.: Mining Closed and Maximal Frequent Induced Free Subtrees, *Workshops Proc. 6th IEEE International Conference on Data Mining* (*ICDM 2006*), *Workshop on Ontology Mining and Knowledge Discovery from Semistructured Documents*, pp.14–18 (2006).
9) Pei, J., Han, J. and Wang, W.: Mining Sequential Patterns with Constraints in Large Databases, *Proc. 11th International Conference on Information and Knowledge Management*, pp.18–25 (2002).
10) Tatikonda, S., Parthasarathy, S. and Kurc, T.: TRIPS and TIDES: New Algorithms for Tree Mining, *Proc. ACM International Conference on Information and Knowedge Management* (2006).
11) Wang, J. and Han, J.: BIDE: Efficient Mining of Frequent Closed Sequences, *Proc. International Conference on Data Engineering* (*ICDE'04*), pp.79–90 (2004).
12) Wang, C., Zhu, Y., Wu, T., Wang, W. and Shi, B.: Constraint-Based Graph Mining in Large Database, *Proc. 7th Asia Pacific Web Conference*, pp.133–144 (2005).
13) Yan, X. and Han, J.: CloseGraph: Mining Closed Frequent Graph Patterns, *Proc. 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD'03*), pp.286–295 (2003).
14) Zaki, M.J.: Efficiently Mining Frequent Trees in a Forest, *Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.71–80 (2002).

(平成 18 年 12 月 8 日受付)
(平成 19 年 4 月 7 日採録)

(担当編集委員　高須 淳宏)

尾崎 知伸

1973 年生．1996 年慶應義塾大学総合政策学部卒業．1998 年同大学大学院政策・メディア研究科前期修士課程修了．2002 年同研究科講師．2005 年神戸大学大学院自然科学研究科助手．現在，神戸大学自然科学系先端融合研究環助教授．博士（政策・メディア）．帰納論理プログラミング，構造データマイニング等の研究に従事．人工知能学会会員．

大川 剛直（正会員）

1963 年生．1986 年大阪大学工学部通信工学科卒業．1988 年同大学大学院工学研究科通信工学専攻博士前期課程修了．大阪大学助手，講師，助教授を経て，2005 年神戸大学大学院自然科学研究科教授．現在，神戸大学大学院工学研究科教授．博士（工学）．知的ソフトウエア，バイオインフォマティクス等の研究に従事．IEEE，人工知能学会，電子情報通信学会，電気学会等の各会員．