DEWS2006 6A-i8

# 行列操作としての頻出アイテム集合列挙

上原子正利† 小柳 滋†

† 立命館大学情報理工学部
〒 525-8577 滋賀県草津市野路東 1 丁目 1-1
E-mail: †m7i@mail.goo.ne.jp, ††oyanagi@cs.ritsumei.ac.jp

**あらまし** 頻出アイテム集合列挙はデータマイニングの分野で頻繁に研究されてきた問題である．一般にこの問題はアイテムとトランザクションに対する操作として考えられるが，本論文ではこの問題を行列操作として考える．行列操作としてのこの問題は 2 値行列から一定の条件を満たす密度 1 部分行列の列集合を発見することに相当し，この問題専用のデータ構造は汎用の行列データ構造の変形と見なせる．さらに我々は，汎用の行列データ構造を用いて行列操作としての頻出アイテム集合列挙アルゴリズムが定義できることを示す．

**キーワード** 頻出アイテム集合列挙，行列，データ構造，アルゴリズム

# Frequent Itemsets Enumeration as Matrix Operation

Masatoshi KAMIHARAKO† and Shigeru OYANAGI†

† Ritsumeikan University
Department of Computer Science, College of Information Science and Engineering
Nojihigasi 1-1-1, Kusatsu, Shiga, 525-8577
E-mail: †m7i@mail.goo.ne.jp, ††oyanagi@cs.ritsumei.ac.jp

**Abstract** Frequent itemsets enumeration is a problem which has been studied frequently in datamining field. This problem is generally seen as operation on items and transactions. On the other hand, we review this problem as matrix operation. We can see this problem as finding column sets of a certain type of density 1 submatrices from a binary matrix, and the data structures for this problem as deformed general matrix data structures. We also show that we can define algorithms which use general matrix data structures and represent this problem as matrix operation.

**Key words** Frequent Itemsets Enumeration, Matrix, Data Structure, Algorithm

## 1. はじめに

データマイニングの分野で頻繁に研究されてきた問題に頻出アイテム集合列挙（Frequent Itemsets Enumeration，以下 FISE）がある [1](注1)．この問題ではアトミックなデータをアイテムと呼び，アイテムの集合をトランザクションと呼ぶ．この問題の目的は，与えられたトランザクション集合中の一定数以上のトランザクションに含まれるアイテム集合の列挙である．

FISE に関する初期の議論は，対象となるデータ集合の実装をディスク上のデータベースと想定し，データ集合へのアクセスの単位をトランザクションとしていた．それに対して 2000 年前後からのアルゴリズムは，対象データ集合を主記憶上に保持し，トランザクションとアイテムのアクセスを併用できるデータ構造を用いている．これらのアクセスの併用は FISE の性質を反映するため，このようなデータ構造は重要な進歩である．

その一方で，この問題に対する視点は 90 年代から変わっていない．その視点は，対象データを当初の表現であるアイテムとトランザクションのまま捉えるものである．FISE アルゴリズムの処理対象はデータベースやトランザクションといったデータに限らないため，この表現は必然的ではない．にもかかわらず，この問題の議論はこの視点に制約され，より一般的な表現との対応が重視されない．その結果，関連する概念が無関係な言葉で表現され，不適切な用語が利用され，議論が必要以上に複雑になる．

本論文の目的は FISE に対する単純な表現方法を示すことである．そのような表現方法として本論文では行列を用いる．行列の利点は 3 つある．まず，行列は一般的な概念であるため，

(注1)：一般には「列挙」ではなく「マイニング」と呼ばれる．本論文では後述の理由により「列挙」と呼ぶ．

「トランザクション」のような特殊な用語を排除できる．また，行列は図で示せるため，人間にとって直感的である．さらに，行列は全ての要素にアドレスを付けているため，概念的にデータ構造と近く，計算機にとって自然な表現である．

本論文の議論は 3 段階からなる．まず，FISE を行列から密度 1 部分行列の列集合を発見する操作として記述する．次に，既存の FISE アルゴリズムとそのデータ構造を行列の観点から再考する．最後に，汎用行列データ構造を用いて行列操作としての FISE アルゴリズムが定義できることを示す．このアルゴリズムの我々の実装は他のアルゴリズムの公開された実装と比べ，ある種の行列では優れ，ある種の行列では劣る．なお，本論文の議論では文献 [2] で用いられる集合列挙木を前提とする．

本論文の構成は次の通りである．2 章では FISE の問題定義と関連する概念をトランザクション集合と行列の 2 通りの方法で表現する．3 章では汎用の行列データ構造を確認し，それらと既存の FISE 専用データ構造の関係を議論し，既存の FISE アルゴリズムの操作を行列操作として記述する．4 章では汎用行列データ構造を用いた FISE アルゴリズム Belone を定義する．5 章では残された課題を議論する．

## 2. FISE と行列

本章では FISE の問題定義と関連概念について，まずトランザクションとアイテムに基づく一般的な表現で述べ，次に行列に基づく表現で記述し直す．

### 2.1 FISE の問題定義と性質

FISE は一般に次のように定義される．入力は自然数 $minsup$ と集合 $T$ であり，$T$ の各要素は集合 $I$ の部分集合である．出力は次の条件を満たす集合 $Freq$ である．$Freq$ の各要素は $I$ の空でない部分集合であり，かつ $T$ の要素のうち $minsup$ 個以上に含まれる．この条件を満たす $I$ の部分集合は全て $Freq$ に含まれる．FISE の用語は次の通りである．$I$ の要素はアイテム，$I$ の空でない部分集合はアイテム集合，$T$ の要素はトランザクション，$Freq$ の要素は頻出アイテム集合，アイテム集合 $I_s$ を部分集合として含む $T$ の要素の数は $I_s$ のサポート，$minsup$ は最小サポートとそれぞれ呼ばれる(注2)．

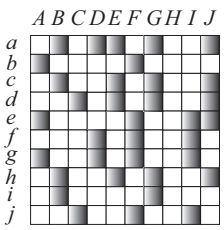
アイテム集合の重要な性質に逆単調性がある．これはアイテム集合が拡大または縮小するにつれてそのサポートが減少または増大することを意味する．この性質は次のように説明できる．$I_p \subseteqq I_q \subseteqq I$ なる $I_p$ と $I_q$ について，$I_q$ を含む $T$ の要素は必ず $I_p$ も含み，$I_p$ を含む $T$ の要素の全てか一部だけが $I_q$ を含むため，$I_p$ のサポートは $I_q$ のサポート以上になる．すなわち，$I_p$ にアイテムを付け加えるとサポートはそのままか小さくなり，$I_q$ からアイテムを取り除くとサポートはそのままか大きくなる．

逆単調性から導出できるアイテム集合の性質に，非頻出アイテム集合を含むアイテム集合は非頻出になるというものがある．すなわち，$I_p$ のサポートが $minsup$ 未満なら，$I_p$ にアイテムを付け加えた $I_q$ のサポートはさらに小さくなり，$minsup$ 未満のままである．この性質より，ひとたび非頻出アイテム集合を見つければ，それを含むどのアイテム集合も非頻出だとわかる．本論文で議論する全てのアルゴリズムがこの性質を利用する．

(注2)：$minsup$ は $T$ の要素数に対する百分率として表現されることが多い．

| TID | アイテム |
|---|---|
| a | B, D, E, G, J |
| b | A, F |
| c | B, E, G, J |
| d | C, E, G, J |
| e | A, F, I, J |
| f | D, F, I |
| g | A, D, F, I |
| h | B, E, G, J |
| i | B, G |
| j | C, F, I |

(a)




(b)

図 1　データ集合の例
Fig. 1　An Example of Dataset

図 1(a) は FISE で用いられるデータ集合の例である．ここでは $a$ から $j$ のトランザクションがあり，アイテムは $A$ から $J$ のうち $H$ を除く 9 個である．$minsup$ を 3 とすると，$Freq$ は $C$ を除くアイテムとそれらの組み合わせの $\{A, F\}$ などからなる．

なお，FISE はしばしば frequent itemsets/pattern mining と呼ばれるが，ここでの「pattern」は集合を指し,「mining」は列挙を指す．本論文ではより単純な表現を選び,「集合」と「列挙」を用いる．

### 2.2 行列操作としての FISE

FISE の各トランザクションはアイテムの非順序集合である．そのため，何らかの方法でアイテムの順序を決めれば，それに従って各トランザクションを並べ替え，データ集合を 1 つの 2 値行列として表現できる．以下ではトランザクションを行，アイテムを列とし，あるトランザクション中にあるアイテムが出現すれば，それに対応する行列要素を 1 とする.「行集合」で行番号の集合を,「列集合」で列番号の集合を指す．個々の行ベクトルに 1 要素を持つ列を「行要素」「1 要素列」と呼ぶ.「列要素」「1 要素行」も同様である．図 1(b) は図 1(a) の行列表現である．ここでのアイテムの順序はアイテム番号の昇順である．

行列で考えると，FISE は次の条件を満たす列集合の列挙操作となる．その条件は，その列集合の全ての列に 1 要素を持つ行の数が $minsup$ 以上というものである．例えば，図 1 の列集合 $\{E, G\}$ は全ての列が行集合 $\{a, c, d, h\}$ に 1 要素を持つため，$minsup$ が 4 以下のとき頻出アイテム集合となる．この条件を満たす部分行列は，部分行列中の 1 要素数 $s$，行数 $t$，列数 $u$ で $s/(tu)$ を密度とすると，密度 1 である．以下では列集合 $C$ の全ての列に 1 要素を持つ全ての行からなる集合を「$C$ の密度 1 行集合」と呼ぶ．ある列集合 $C$ の密度 1 行集合 $R$ について $|R|$ が $minsup$ 以上なら，$R$ と $C$ が作る部分行列を頻出アイテム集合部分行列と呼ぶ [3]．この部分行列の重要な性質は，列集合が決まれば行集合も一意に決まることである．そのため，この部分行列を指定するには元の行列と列集合で十分である．

FISE を行列として表現できることは，FISE に関する概念も行列で表現できることを意味する．例えば，最小サポートは部分行列に要求される最小行数に相当する．また，逆単調性は次

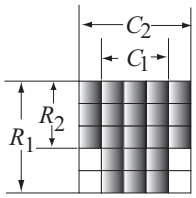



図 2　横に伸ばせば縦に縮む
Fig. 2　Stretched in Width and Shrinking in Height

のように表現できる．図 1 の $\{E\}$ の密度 1 行集合は $\{a, c, d, h\}$ だが，拡大した列集合 $\{B, E\}$ の密度 1 行集合は $\{a, c, d, h\}$ から縮退した $\{a, c, h\}$ である．すなわち，逆単調性は列集合が拡大するにつれてそれらの密度 1 行集合が縮退すること意味する．あるいは，列集合の間に包含関係が成立すれば，それらの密度 1 行集合の間には逆方向の包含関係が成立すると言える．図 2 はこの関係を示している．ここでは $R_1 \times C_1$ と $R_2 \times C_2$ がそれぞれ頻出アイテム集合部分行列であり，列集合が拡大すれば行集合が縮退している．これを直感的に表現すると,「横に伸ばせば縦に縮む」となる．

FISE を行列として表現できることはまた，FISE 専用のデータ構造を行列データ構造の一種と見なせ，FISE アルゴリズムを行列操作アルゴリズムと見なせることを意味する．次章でこれらを具体的に見る．

## 3. 行列の観点から見た FISE のデータ構造とアルゴリズム

本章ではまず汎用の行列データ構造を確認し，それらと既存の FISE 専用データ構造の関係を見る．また，既存の FISE アルゴリズムの操作を行列操作として記述し直す．

### 3.1 汎用の行列データ構造

汎用の行列データ構造として最も単純なものは全ての行列要素を記録した配列である．このデータ構造では任意の行列要素に定数時間でアクセスできるが，行列が疎になると不要な 0 要素が増え，記憶量と非ゼロ要素への選択的アクセスの両方で効率が落ちる．疎行列では一般に配列とリストを組み合わせたデータ構造が用いられる [4] [5]．これは一般に隣接リストと呼ばれるが，本論文では後述の他のデータ構造との関係上，配列・リスト行列（Array-List Matrix，以下 ALM）と呼ぶ．図 3 は図 1(b) の ALM である．なお，FISE は 2 値行列を扱うので，要素の値は記録せず列番号だけ記録する．ALM の難点は列アクセスの効率が悪いことである．すなわち，ある列に 1 要素を持つ行番号を知るには全ての行リストを調べる必要がある．以下ではこの操作を全行スキャンと呼ぶ．

ALM の問題を解決する行列データ構造に文献 [6] のものがある．これは要素アクセスの観点から見ると図 4 のように行の ALM と列の ALM を組み合わせたものに等しい．以下ではこれを配列・リスト双対行列（Array-List Dual Matrix，以下 ALDM）と呼ぶ．ALDM では行集合を表す配列を行エントリ，列集合を表す配列を列エントリと呼ぶ．

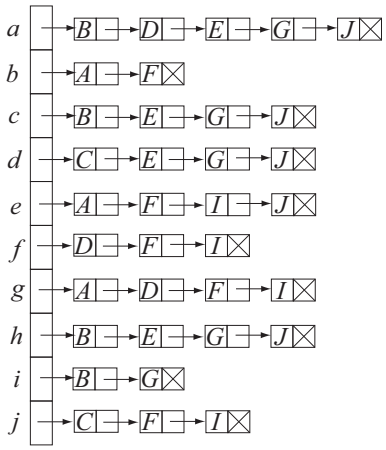



図 3　配列・リスト行列（ALM）
Fig. 3　Array-List Matrix (ALM)

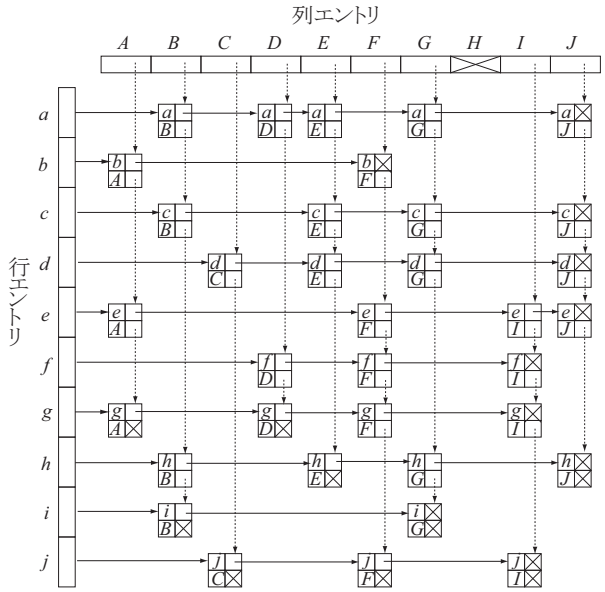



図 4　配列・リスト双対行列（ALDM）
Fig. 4　Array-List Dual Matrix (ALDM)

ALDM を単純化したものが文献 [3] のデータ構造である．これは ALDM のように行エントリと列エントリのペアからなるが，リストではなくソート済み配列を用いる，リストを配列に替えることで記憶量が減り，ランダムアクセスも可能になる．以下ではこれを配列・配列双対行列の略で AADM と呼ぶ．図 5 は図 1(b) の AADM である(注3)．なお，本論文では文献 [3] のデータ構造に加え，行・列エントリで各行・列要素配列の要素数も記録する．

### 3.2 FISE 専用データ構造と行列の関係

次に，既存の FISE アルゴリズムがデータ集合の表現に用いるデータ構造を前節のものと関連付ける．

Apriori [9] は古くから最もよく言及されてきた FISE アルゴ

(注3)：多値行列で AADM と同様のデータ構造が文書処理プログラム GETA [7] で用いられている．また，文献 [8]p.43 では，文書データ間の類似度の決定の際に文書ごとの単語集合と各単語を含む文書集合を合わせて用いる方法が述べられているが，この組み合わせも AADM と同じ機能を持つ．

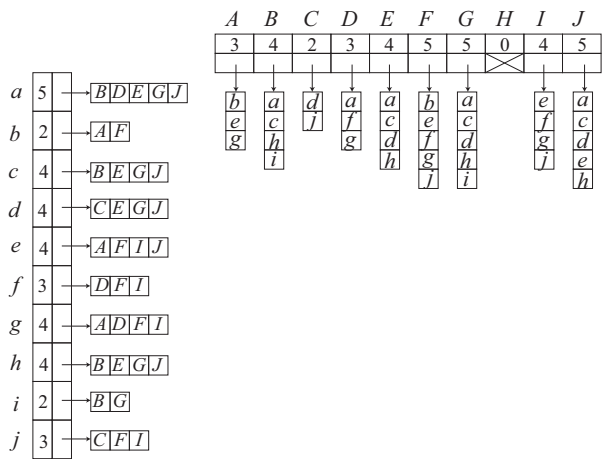



図 5　配列・配列双対行列（AADM）
Fig. 5　Array-Array Dual Matrix (AADM)

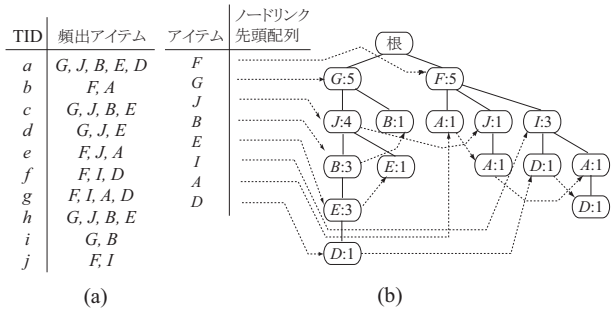



図 6　頻出アイテムだけのデータ集合とその FP 木
Fig. 6　A Dataset of Frequent Items and Its FP-tree

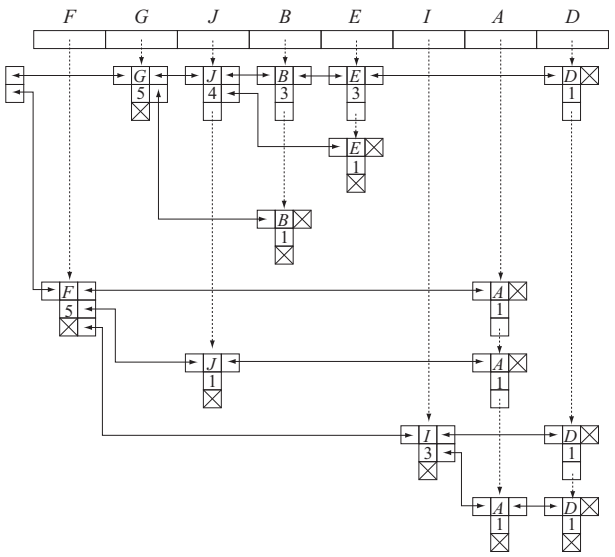



図 7　ALDM の変形としての FP 木
Fig. 7　FP-Tree as Deformed ALDM

リズムである．このアルゴリズムはデータベース操作を念頭に置いているため，データ集合のデータ構造を明示しない．しかし，データ集合に対する操作が個々のトランザクションを順に取り出すものであるため，そのデータ構造は ALM と見なせる．

Apriori は ALM から来る 2 つの難点を抱えている．その 1 つは図 2 のように列集合を拡大するとき，行列に列からアクセスできないため現在の列集合に追加すべき列がわからず，とりあえず追加してから追加後の集合のサポートを全行スキャンで調べなければならないことである．この方法は「候補生成と検査（candidate generation-and-test）」と呼ばれる．もう 1 つの難点は，処理時間を消費する全行スキャンの回数を減らすため，FISE の状態空間である集合列挙木を記憶量の効率が良い深さ優先ではなく，幅優先で探索することである．

これらの難点を解消したアルゴリズムに FP-growth [10] や H-mine(Mem) [11] がある．これらは行列に対して行と列の両方のアクセスを可能にするデータ構造を用いることで，全行スキャンをせず追加すべき列を決定でき，その結果，集合列挙木を深さ優先で探索できる．この方法は「パターン拡大（pattern growth）」と呼ばれる(注4)．FP-growth は FP 木，H-mine(Mem) は H-struct というデータ構造をそれぞれ用いる．

FP 木は名称と異なり木ではない．これは図 6(b) のように木構造とリストを組み合わせたものである．FP 木を構成するには，まず全アイテムから頻出アイテムだけを選び，それらのアイテムをサポートの降順でソートした配列 *flist* を作る．次に，各トランザクションから頻出アイテムを取り出し，*flist* の順序に従ってソートした部分トランザクションを作る．そして，複数の部分トランザクションを共通するプレフィックスでまとめた木を作り，複数のトランザクションの間の同一アイテムをポインタでつなぎ，各アイテムが最初に出現する位置を格納した配列であるノードリンク先頭配列を構成する．図 6(a) は図 1(a) から頻出アイテムだけを選んだものであり，図 6(b) は図 6(a) から作られた FP 木である．各節点の数値はその節点にまとめられたトランザクションの数を示す．

FP 木は ALDM の変形と見なせる．この変形は列の順序変更と行の圧縮からなる．図 7 は図 6(b) を図 4 と同じ形に描き直したものである．図 4 と比べると，ノードリンク先頭配列は列エントリに，木構造の根は行エントリにそれぞれ相当する(注5)．ここでの行エントリの要素は 2 つだけだが，これは行が圧縮されたためである．

一方，H-struct は FP 木と同様に非頻出アイテムを排除したデータ集合から作られるが，FP 木と異なりアイテムの順序を任意とし，行を圧縮しない．図 8 は図 6(a) から構成した H-struct である．図中のヘッダテーブルは FP 木のノードリンク先頭配列に相当し，行列としては列エントリに相当する．各行は列番号とポインタのペアを要素とする配列である．ポインタは実行時に動的に変化する．

H-struct は ALDM と AADM を複雑に組み合わせた形をしている．ここでの行ベクトルは配列で表現され，列エントリも存在する．しかし，明示的な行エントリと列ベクトルは存在しない．これらの役割はヘッダテーブル最下段のポインタと行要

(注4)：「候補生成と検査」の Apriori も「パターン」を拡大する．両者の相違点は不要な拡大「パターン」を生成するかどうかである．

(注5)：図 4 では行のリストが単方向だが，図 7 では双方向である．この相違点はそれぞれを用いるアルゴリズムから来る．ALDM のリストを双方向にしても問題は無いが，FP 木の行リストを単方向にすることはできない．

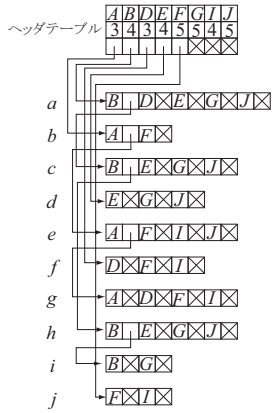



図 8　H-struct
Fig. 8　H-struct

素配列内のポインタが分担する．

### 3.3　FISE アルゴリズムの行列操作

以上のアルゴリズムはいずれも，各時点で発見した部分行列の列集合を元に新しい部分行列を構成する．以下では各アルゴリズムの操作を行列操作の点から見る．

Apriori は部分行列を列集合だけで保持する．要素数 $k$ の列集合を構成するには，まず要素数 $k-1$ の列集合から可能な全ての組み合わせを作り，次にそれらの密度 1 行集合の要素数が $minsup$ 以上あるかを全行スキャンで確認する．Apriori の「候補生成と検査」を行列として考えると，列集合から列集合を構成し，その後に行列要素を確認する「列集合-列集合-行列要素」の方法と言える．

それに対して，FP-growth と H-mine(Mem) の「パターン拡大」は，列集合から行列要素を確認して新しい列集合を作る「列集合-行列要素-列集合」の方法である．これが可能であるのは，FP 木や H-struct によって行と列の両方から行列要素へアクセスできるためである．ただし，FP-growth と H-mine(Mem) は保持する部分行列の表現方法が異なる．FP-growth は条件付き FP 木という中間データを保持するが，これを部分行列として考えると，列集合と行列要素を合わせたものに相当する．H-mine(Mem) は複雑な形で部分行列を表現し，列エントリのポインタを辿って到達できる行で部分行列の行集合を表現する．

## 4.　Belone：行と列の両方からアクセス可能な汎用行列データ構造を用いる FISE アルゴリズム

以上で見たように，FISE は行列操作として記述でき，FISE 専用データ構造は行と列の両方からアクセスできる特殊な行列データ構造と見なせる．それならば，議論を単純化して，行と列の両方からアクセスできる汎用の行列データ構造を用いて FISE アルゴリズムを定義できるはずである．本章ではこのようなアルゴリズムの例として Belone（ビローン）を定義する．

Belone は行列データ構造として概念も実装も単純な AADM を用いる(注6)．行・列アクセスが共に容易な AADM を用いる

(注6)：C++ での AADM は

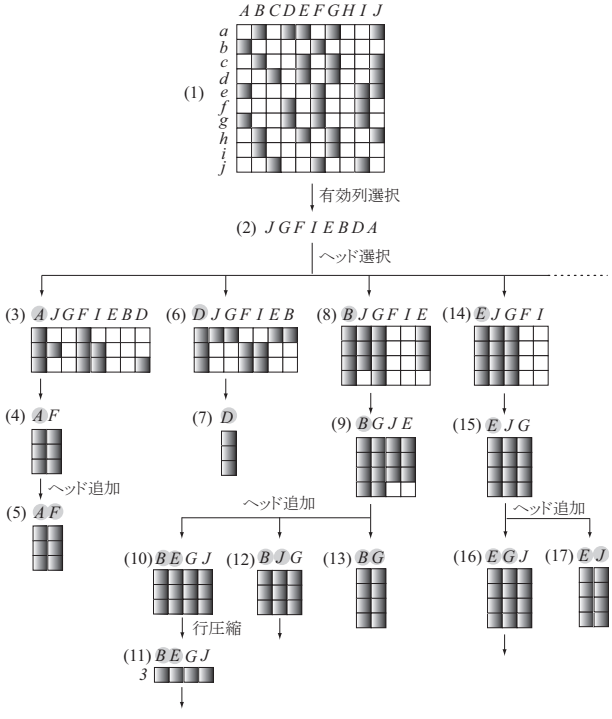



図 9　Belone の行列操作木
Fig. 9　Matrix Operation Tree of Belone

ことで，Belone は「パターン拡大」を実行でき，集合列挙木を深さ優先で探索できる．ただし，AADM は要素を行と列の二重に記憶するため，密行列に対しては全要素の配列からなる行列と比べて記憶量で劣る．そのため，Belone の主な対象は疎行列となる．また，Belone は FP 木とは異なる方法で行圧縮を行う．この圧縮は FP 木より単純であるが，圧縮効率は FP 木より劣る．なお，Belone は FP-growth や H-mine(Mem) と異なり，頻出アイテムだけを選んで主記憶上の行列データ構造を構成することはない．これは，Belone の目的が与えられた行列の操作であり，行列の構成は Belone の外部で行うためである．

### 4.1　Belone の行列操作

Belone の基本的な行列操作は，集合列挙木の節点のヘッドとその密度 1 行集合からなる部分行列を図 2 の $R_1 \times C_1$ から $R_2 \times C_2$ のように横に引き伸ばすものである(注7)．処理過程でヘッドは拡大し，その密度 1 行集合とテイルは縮退してゆく．

Belone の概念的な挙動を図 9 の木で説明する．この木は図 1(b) から最小行数 $minr$ を 3 として実行した場合のものである．図中の数字は構成された順序を示し，(11)(12)(16) より下と (14) より右は省略している．

Belone はまず行列 (1) の列ベクトルを調べ，1 要素数が $minr$ 以上の列集合を 1 要素数で降順ソートした順序集合 (2) を作る．この集合の要素を「有効列」と呼ぶ．ここでのソートは文献 [2] などでも用いられ，処理過程の部分行列を小さくして処理速度を上げる効果がある．次の (3) は，(2) の終端要素 $A$ を集合列

`vector<vector<unsigned int> > ent[2];`
である `ent[0]` が行エントリを，`ent[1]` が列エントリを表す．
(注7)：Belone という名称は何らかの物を引き伸ばす際の日本語の擬態語に由来する．

挙木の節点のヘッドとし，それ以外をテイルとしたものである．図中ではヘッドを網掛けで示す．行集合はヘッドの密度 1 行集合である．次にテイルの各列について，その列ベクトルと現在の行集合の共通要素数を調べる．対象が 2 値ベクトルなので，共通要素数は内積に等しい．内積が $minr$ 未満の列は逆単調性より不要であるため排除し，(4) を作る．この時点でヘッドは頻出アイテム集合と判明しているため出力する．

次にテイルの終端要素をヘッドに移動して (5) を作る．このような要素の移動は一般に行集合の縮退を伴うが，ここでは発生していない．図中で行縮退が発生するのは (9) から (10) と (12) への遷移である．(5) では頻出であることが判明している $\{A, F\}$ を出力し，テイルが無くなったため処理は後戻りする．この時点で $A$ を含む頻出アイテム集合が全て列挙されている．

(10) から (11) では行圧縮を行っている．(10) では部分行列中の 3 つの行が同じ列に 1 要素を持つため，これらをまとめ，まとめた行に圧縮数 3 を割り当てる．この数を用いると，(11) の継続節点での列の内積計算時に 1 つの行の確認で 3 つの行の確認と同じ結果が得られる．この圧縮は 1 要素列が完全に一致する行だけを対象とし，プレフィックスの一致で圧縮する FP 木より圧縮効率は低い．

### 4.2 Belone のデータ構造

Belone は他の FISE アルゴリズムと同様，行列の他に部分行列データ構造を必要とする．このデータ構造は次の通りである．

最初に構成されるものは (2) の有効列配列 `eff` である．次に (4) のような (2) から見て深さ 2 の部分行列 `submat` が構成される(注8)．(9) を例にするとこれらは次のようになる．

```
eff=[J,G,F,I,E]
submat={
  rsize=4, csize=3 # 行数と列数
  ctable=[G,J,E] # submat と AADM の列番号対応表
  coffset=1 # ctable の先頭の極大内積列の個数
  array=[1,1,1,0,1,1,1,0] # csize-coffset 個の列
                          # ベクトルの全要素配列
  compress=[1,1,1,1] # 各行の圧縮数
  key=[0,0,0,0] # 各行の圧縮キー
}
```

`csize` はこの時点のヘッドを含まない．`ctable` は `submat` の列 `c` が AADM で列 `ctable[c]` であること示す．`array` の要素格納は列を単位にする．つまり，同じ列の要素は `array` 内で連続し，同じ行の要素は連続しない(注9)．`coffset` の「極大内積列」とは，この部分行列の全ての行に 1 要素を持つ列を意味する．Belone では行集合が縮退するため，ある列が極大内積列になればそれ以降常に極大内積を持ち，このような列は `array` に記録する必要がない．よって，`array` には `csize-coffset` 個の列だけが格納され，`submat` での `r` 行 `c` 列の要素は `array[(c-coffset)*rsize+r]` となる．Belone は `c<coffset` なる列 `c` にアクセスしない．`compress` と `key` は行圧縮時に使われる配列で，初期値はそれぞれ 1 と 0 である．`eff` と `submat` は木を深さ方向に進む過程で保持され，生成時の節点の隣の枝に移った時点で破棄，再構築される．

`eff` と `submat` 以外にも部分行列のデータ構造が必要である．各節点ではその時点のヘッド `head`，テイル `tail`，ヘッドの密度 1 行集合 `rows` の 3 つの配列と，テイルの先頭のうち極大内積列の個数 `endmax` を保持する．(9) ではそれぞれ

`rows=[0,1,2,3]`, `head=[B]`, `tail=[0,1,2]`, `endmax=1`

となる．`head` の要素は AADM の列番号，`tail` の要素は `submat` の列番号，`rows` の要素は `submat` の行番号である．この時点での `endmax` の値は `submat.coffset` に等しいが，行縮退に伴って極大内積列が増えるとこの値も増え，極大内積列が `tail` から `head` に移るに連れてこの値は減る．

### 4.3 Belone のアルゴリズム

Belone のアルゴリズムを図 10 に示す．図中で省略された AADM からの `submat` の構成方法と圧縮キーの決定方法は次の通りである．

`submat` は AADM が行エントリと列エントリを併せ持つことを利用し，場合に応じて効率の良い方から構成される．効率の基準はメモリアクセスの回数であり，この回数は事前にわからないため以下の方法で推定する．行エントリを利用する場合は，列 `x` の全ての 1 要素行の全ての行要素が確認され，1 つでも `x` と共通 1 要素行を持つ全ての列の `x` に対する内積が決定され，その後内積が `minr` 以上の `eff` の列の個々について `x` との共通 1 要素行を決定する．列エントリを利用する場合は，`eff` の全ての列について `x` との共通 1 要素行を決定し，その要素数が `minr` 以上の列だけを残す．アクセス回数の推定のために，まず `x` の 1 要素行数 `rvsize` と `x` に 1 要素を持つ全ての行の行要素配列の要素数の和 `relem` を求める．これらはそれぞれ図 5 の列・行エントリに記録された 1 要素数からわかる．そして，`eff` のうち内積が `minr` 以上のものの割合の推定値 `effrate`，`relem`，`rvsize` および `eff` の要素数 `effsize` からアクセス回数を推定する．以下の実験では `n*relem < rvsize*effsize*(1-effrate)` が真なら行エントリを，偽なら列エントリを用いている(注10)．`n` の値は実験で 3 と決定した．`effrate` は毎回，内積が `minr` 以上の列の数とその時点の `effsize` の比で更新し，次回の `submat` 構成時に利用する(注11)．

(注8)：(2) から見て深さ 1 の部分行列が明示的に生成されることはない．

(注9)：これは処理速度に影響する．その原因として考えられるものは，Belone が列内積の計算を多用するため，列要素が連続することで参照の局所性が成立しやすくなることである．

(注10)：この式の理由は次の通りである．まず，`x` と他の 1 つの列の共通要素の決定のために AADM の列エントリから行列要素にアクセスする回数は，共通要素数が `minr` に到達しないと判明した時点で処理を打ち切ると，`rvsize*2` に近い．行エントリから全ての列の内積を決定する際には，行エントリの該当する全ての行要素 `relem` 個にアクセスする．このとき，内積を記録するためには列番号をキー，その列の `x` との内積を値としたハッシュテーブルを用いるが，このテーブルへのアクセスを考慮し，メモリアクセス回数を `p*relem` とする．これらを用いると，行エントリを利用する場合のメモリアクセス回数は `p*relem+rvsize*2*effsize*effrate` となる．同様に列エントリを用いる場合のメモリアクセス回数は `rvsize*2*effsize` となる．行エントリを用いる方がアクセス回数が少なくなるのは，`(p*relem+rvsize*2*effsize*effrate)<(rvsize*2*effsize)` が成立する場合である．移項すると `(p/2)*relem<(rvsize*effsize*(1-effrate)` になる．ここで `p/2` を `n` とした．

(注11)：ここでは有効列のうち内積が `minr` 以上のものの割合が次の回でも大き

```
大域データ:
AADM mat, 非負整数 minr, 出力先のオブジェクト out,
0以上1以下の浮動小数点数 effrate

belone():
1. matの列エントリをスキャンして有効列配列effを構成する. effは各列の
   1要素数で降順ソートする.
2. effrateを0以上1以下の適当な値で初期化する(例えば0.75).
3. effが空になるまで以下を繰り返す.
   3a. effの終端xをポップし, その列番号を空のheadに格納する.
   3b. headとeffからsubmat, rows, tail, endmaxを作る(本文参照).
   3c. belone_inner(submat,rows,head,tail,endmax)を実行する.

belone_inner(submat,rows,head,tail,endmax):
1. headをoutに出力する.
2. tailが空になるまで以下を繰り返す.
   2a. tailの終端xをポップし, headにsubmat.ctable[x]を追加する.
   2b. tailの要素数がendmaxより大きければ, submatの列xとrowsで1要素を
       共通して持つ行番号の配列xrowsを作る. xrowsの各要素rについて,
       submat.key[r]を0にする.
   2c. endmaxとtailの要素数の小さい方をactualendmaxとする.
   2d. 配列xtailを作り, tail先頭のactualendmax個を格納する.
   2e. 非負整数ペアの配列dsuを空で定義する.
   2f. tailのactualendmax+1番目以降の要素をtとして以下を繰り返す.
      2f1. submatの列tとxrowsで1要素を共通して持つ行番号の配列xxrows
           を作る.
      2f2. xxrowsとxrowsが等しいなら, xtailにtを追加し(2f)へ.
      2f3. xxrowsの各要素rについて, submat.compress[r]の和ipを求める.
           ipがminr未満なら(2f)へ.
      2f4. dsuにipとtのペアを追加する.
      2f5. xxrowsの各要素rについてsubmat.key[r]を更新する(本文参照).
   2g. dsuが空なら, xtailの要素数をxtsizeとして,
       belone_inner(submat,xrows,head,xtail,xtsize)で再帰する.
   2h. dsuが空でなければ次を行う.
      2h1. dsuを1つめで降順ソートし, 現時点のxtailの要素数をnextmaxとする.
      2h2. dsuの先頭から順に2つめをxtailに追加する.
      2h3. 非負整数ペアの集合rowtabを空で定義する. ペアの1つめをキー,
           2つめを値とする.
      2h4. 非負整数ペアの配列compを作る.
      2h5. xrowsの各要素rについて以下を繰り返す.
         2h5a. submat.key[r]をkとし, これをキーとしてrowtabを検索する.
         2h5b. キーが一致するものが存在するなら, その値をvとして, submat
               内で行rとvが同じ列に1要素を持つか調べる. ただし, 調べる列
               はdsuの2つめに存在するものだけである.
         2h5c. 行rとvが同じ列に1要素を持つなら, submat.compress[v]に
               submat.compress[r]を加え, 加えた記録をcompに追加する.
         2h5d. キーの一致するものが存在しないか, 行rとvで1要素を持つ列が
               一致しないなら, rowtabにkとrのペアを追加する.
      2h6. rowtabの全ての値からなる配列xxrowsを構成する.
      2h7. belone_inner(submat,xxrows,head,xtail,nextmax)で再帰する.
      2h8. compを見てsubmat.compressを加算前の値に戻す.
   2i. headの終端をポップする.
```

図 10　Belone のアルゴリズム
Fig. 10　The Algorithm of Belone

圧縮キーの決定方法は次の通りである．図 10(2f) で `tail` の各要素の列に数を割り当て，(2f5) でその列に 1 要素を持つ行の圧縮キーにこの数を加算する．その結果，`tail` 中の同じ列に 1 要素を持つ行は必ず圧縮キーが等しくなり，(2h3) の `rowtab` に同じキーで登録される．ただし，1 要素列の異なる行が同じキーを持つ可能性がある．例えば，`tail=[1,2,3]` とし，行 `a` と `b` のこれらの列の中での 1 要素列をそれぞれ `[1,2]`，`[3]` とする．もし各列に割り当てる数を単純に列番号とすれば，`a` と `b` は同じキーを持つことになる(注12)．この場合は (2h5b) で不一致が確認されるが，(2h5) では `rowtab` 内にキーも 1 要素列も同じ他の行が存在しても再検索をしないため，このような行の存在は圧縮効率を下げる．以下の実験では同じキーを持つ行を減らすため，列番号にある程度大きな素数をかけて，異なる 1 要素列を持つ行が同じキーを持つ可能性を下げている．

以上で Belone の定義は終了した．

### 4.4　Belone の処理速度

本論文の目的は FISE に対する単純な表現方法を示すことであり，Belone を定義した目的は行列操作として FISE を記述するアルゴリズムの存在を示すことである．そのため，本論文の主な内容はここまでで終わっている．以下は本論文の主要な目的ではないが，いくつかのデータ集合を用いて他のアルゴリズムと処理速度を比較することで Belone の性質を調べる．

データ集合には FIMI のサイト(注13)で公開されている T10I4D100K（以下 T10），T40I10D100K（以下 T40），chess，我々が文献 [12] で構成した文書行列 NSF，および今回構成したランダムな行列 RAND を用いる．NSF は元来多値行列だが，要素の存在だけを用いて 2 値行列に変換した．これらの特徴値を表 1 に示す．表中の「人工」と「自然」はそれぞれ疑似乱数あるいは現実の意味のあるデータから構成したことを示す(注14)．

表 1　行列の特徴値
Table 1　Specs of Matrices

| | T10 | T40 | RAND | NSF | chess |
|---|---|---|---|---|---|
| 密度 | 0.01 | 0.04 | 0.5 | 0.0014 | 0.49 |
| 行数 | 100000 | 100000 | 1000 | 49078 | 3196 |
| 列数 | 1000 | 1000 | 1000 | 71969 | 76 |
| 全 1 要素数 | 1010228 | 3960507 | 500000 | 4876169 | 118252 |
| 行 1 要素数平均 | 10 | 39 | 500 | 99 | 37 |
| 列 1 要素数平均 | 1010 | 3960 | 500 | 67 | 1555 |
| 人工か自然か | 人工 | 人工 | 人工 | 自然 | 自然 |
| 疎か密か | 疎 | 疎 | 密 | 疎 | 密 |

比較対象のアルゴリズムは FP-growth であり，実装は Christian Borgelt による(注15)．この実装は一般の FP-growth に加えて枝刈り [13] を利用できるため，ここでは枝刈りを行う場合と行わない場合の両方で速度を計測した．我々の実装は C++ を用いた(注16)．計算機は CPU が PentiumIV 3.2GHz，RAM が 1GB の PC で，OS は Linux，コンパイラは GCC である．

処理時間の計測結果を図 11 に示す．これらの時間は `time(1)` で各プログラムの開始から終了までの実時間を計測したものである．上段の人工データのうち，T10 では Belone が常に速いが，差が小さいため，アルゴリズムではなく実装の細部が影響している可能性がある．しかし，T40 では明白な差が出ている．枝刈りなしの FP-growth は非常に遅いが，枝刈りありの FP-growth でも Belone より明らかに遅い（minsup=300 で Belone は 59 秒，FP-growth は 210 秒）．RAND ではさらに差が開く．この 3 つを見ると，人工データでは Belone が優れていると言える．

---

く変わらないと仮定している．
(注12)：すなわち，1+2=3 である

(注13)：http://fimi.cs.helsinki.fi/data/
(注14)：chess の構成方法は文献 [2] での「ゲームの状態情報からコンパイルされた」という記述による．
(注15)：http://fuzzy.cs.uni-magdeburg.de/~borgelt/fpgrowth.html , version 1.8 (2005.12.06)
(注16)：著者までご連絡くだされば Belone のソースコードをお送りします．

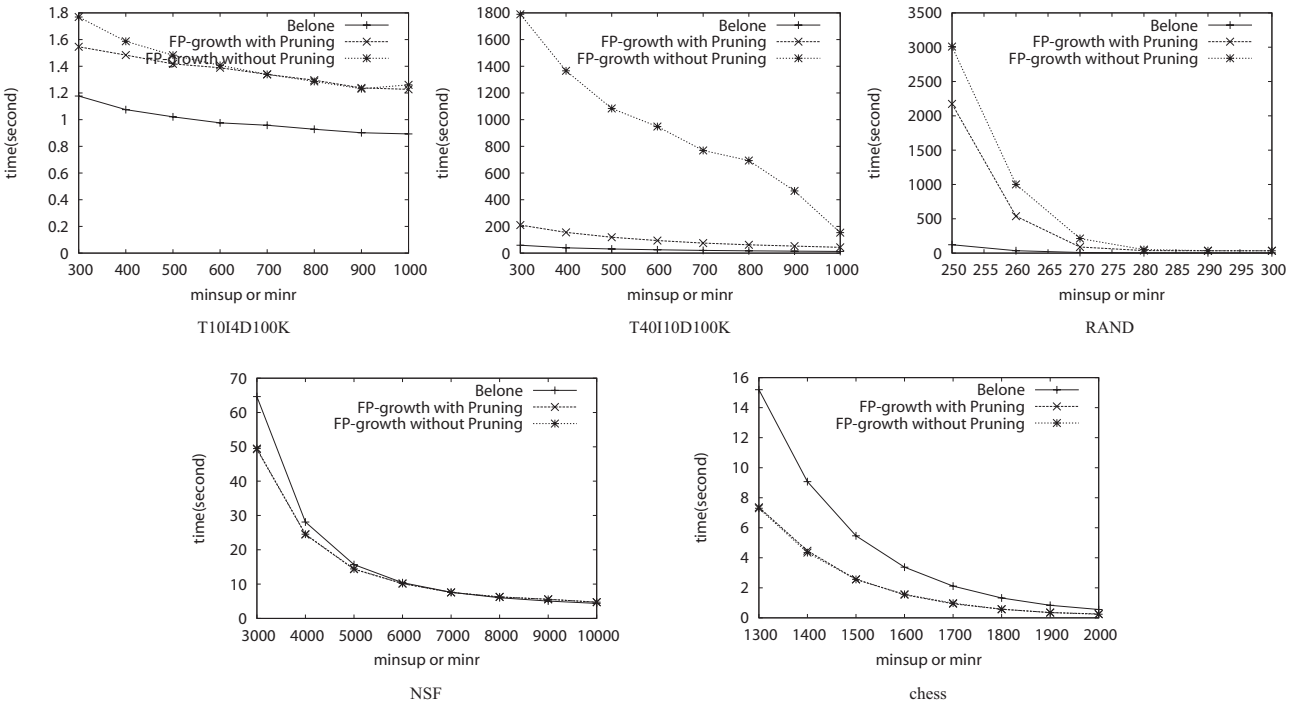



図 11　Belone と枝刈りあり/なしの FP-growth の処理時間
Fig. 11　Processing Times of Belone and FP-growth with/without Pruning

一方，下段の自然データでは優劣が逆転する．NSF では 7000 まで FP-growth が Belone を上回る．8000 以降で Belone が逆転するが，差はほとんど見えない．chess では差が明らかで，FP-growth は Belone より常に 2 倍程度速い．この 2 つを見ると，自然データでは FP-growth が優れていると言える．

データによる優劣の違いの原因として考えられるものは，FP 木の性質である．FP 木はランダムな行列をうまく圧縮できない一方，chess のように密で自然な行列は効率良く圧縮でき，NSF のように疎な行列は自然であっても chess ほどは圧縮できないと考えられる．文書行列である NSF で最小行数が増えると Belone が逆転した理由としては，最小行数が増えるに連れて考慮される列が頻出語だけになり，単語間の意味的な関連が失われてランダム行列に近づくことが考えられる．

## 5. おわりに

本論文では FISE を行列操作として捉え，逆単調性などの FISE の概念を行列の観点から記述した．また，FISE のデータ構造を汎用の行列データ構造と関連付け，FISE アルゴリズムを行列操作の観点から記述した．さらに，汎用データ構造を用いる行列操作としての FISE アルゴリズム Belone を定義した．

本論文で議論できなかった問題として，FISE の関連問題である極大頻出アイテム集合列挙 [2] などがある．このような関連問題も基本的には行列操作として記述できると考えられるため，今後このような拡張を Belone に導入できる可能性がある．

また，行列の性質によって Belone と FP-growth の優劣が変わる理由についてさらに考える必要がある．実験結果を見ると行列の圧縮可能性のようなデータに依存する値が処理速度に大きく影響していると考えられ，また，この変化がアルゴリズムではなく実装に起因する可能性もある．これらの点を明確にすることは，データの種類に合わせたアルゴリズムの選択を可能にし，効率の良い問題解決を実現する鍵になると考えられる．

## 文　献


[1] 福田, 森本, 徳山：“データマイニング”, 共立出版 (2001).
[2] J. R. J. Bayardo: “Efficiently Mining Long Patterns from Databases”, Proceedings of the ACM SIGMOD, Seattle, WA, pp. 85–93 (1998).
[3] 上原子, 小柳：“内積縮退 MC:類似行の検出と類似列の検出を組み合わせたマトリクスクラスタリングアルゴリズム”, 情報処理学会論文誌：データベース, **45**, SIG 7 (TOD22), pp. 151–162 (2004).
[4] A. V. Aho and J. D. Ullman: “Foundations of Computer Science C Edition”, Computer Science Press (1995).
[5] ジョン・ベントリー（小林健一郎訳）：“珠玉のプログラミング本質を見抜いたアルゴリズムとデータ構造”, ピアソン・エデュケーション (2000).
[6] 小柳, 久保田, 仲瀬：“Matrix Clustering : CRM 向けの新しいデータマイニング手法”, 情報処理学会論文誌, **42**, 8, pp. 2156–2166 (2001).
[7] 汎用連想検索エンジン GETA. http://geta.ex.nii.ac.jp/.
[8] 徳永：“情報検索と言語処理”, 東京大学出版会 (1999).
[9] R. Agrawal and R. Srikant: “Fast algorithms for mining association rules”, Proceedings of the 20th VLDB Conference, pp. 487–499 (1994).
[10] J. Han, J. Pei, Y. Yin and R. Mao: “Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Aproach”, Data Mining and Knowledge Discovery, **8**, pp. 53–87 (2004).
[11] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang: “H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases”, ICDM, pp. 441–448 (2001).
[12] 上原子, 小柳：“行列上の相互順序決定”, 情報処理学会論文誌：データベース, **SIG 13 (TOD27)**, pp. 16–25 (2005).
[13] C. Borgelt: “An Implementation of the FP-growth Algorithm”, Workshop Open Software for Data Mining (OSDM'05, Chicago, IL) (2005).