3M1-1

頻出パターン発見アルゴリズム入門 ー アイテム集合からグラフまで ー

An Introduction to Frequent Pattern Mining – Itemsets to Graphs -

宇野 毅明[*1]
Takeaki Uno

有村 博紀[*2]
Hiroki Arimura

[*1] 国立情報学研究所・総合研究大学院大学
National Institute of Informatics,
The Graduate University for Advanced Studies

[*2] 北海道大学大学院情報科学研究科
Hokkaido University, Graduate School for
Information Science and Technology

This paper provides a brief survey of efficient algorithms for the frequent itemset mining (FIM) problem, which is one of the most popular and basic problems in data mining. Firstly, we give the descriptions of two well-known algorithms for the FIM problem, called *Apriori* and *Backtrack*, which are based on breadth-first search and depth-first search over the space of frequent itemsets, respectively. Then, we discuss advanced techniques to speed-up the algorithms for the FIM problem, namely, database reduction, frequent itemset trees, down-project, occurrence-deliver, recursive database reduction, which improve the key operations of FIM computation such as database access and frequency computation. We also explain how to extend *Backtrack* algorithm for semi-structured data including trees and graphs.

1. はじめに

データマイニングは，大量のデータから有用な規則性やパターンを見つけるための効率よい手法に関する研究である．データマイニング研究は，1994 年のAgrawalらの結合規則発見に関する研究を契機として，急速に発展した．この結合規則発見の鍵となる基本的な技術が頻出アイテム集合発見である． この 10 年間で，結合規則発見にとどまらず，深さ優先探索法の提案や，飽和集合発見，構造データへの拡張など，さまざまな形で盛んな研究が続いており，データマイニングの主要なトピックの一つとなっている．

本レクチャーでは，**頻出集合発見アルゴリズム**の最近の進展について解説する．特に，アルゴリズム研究の立場から，その基本的な構造と，実際のデータのための高速な実装について解説する．本解説は，筆者らによる実装コンテストFIMI'04 での優勝プログラムLCMの開発経験に基づいたものである．現在，LCMを含む，各種の頻出集合発見アルゴリズムが公開され，利用されている．本解説がこれらのアルゴリズムを実際の問題に適用する場合に参考になれば幸いである．

本稿の構成は次のとおりである．2 節で問題の定義を与え，3 節でアプリオリ法と深さ優先探索法の二つの基本的手法を与える．4 節では高速化技術について述べ，5 節では実データ上での性能比較，6 節では応用について，7 節では木構造データからの頻出パターン発見への拡張について述べ，8 節でまとめる．なお，本稿は，人工知能学会誌のレクチャーシリーズ「知能コンピューティングとその周辺」[西田 07]の第 2 回レクチャー[宇野 07]の一部分を基に加筆修正したものである．他の回のレクチャーとも合わせてご覧いただければ幸いである．

2. 頻出パターン

2.1 トランザクションデータベース

今，手元に**トランザクションデータベース** (transaction database)があるとする．これは，各**レコード**(record)がトランザクション(transaction)であるデータベースのことであり，**トランザクション**とはアイテム(item)の集合である．言い換えれば，アイテムの集合 $\Sigma=\{1,\cdots,n\}$に対して，トランザクション はΣの部分集合であり，トランザクションデータベースはトランザクションの集合である．トランザクションデータベースには，同一のトランザクションが複数含まれても良い．Σの部分集合をここでは**アイテム集合**(itemset)とよぶ．

A: 1,2,5,6,7,9	**3つ以上に含まれるアイテム集合**
B: 2,3,4,5	{1} {2} {7} {9}
C: 1,2,7,8,9	{1,7} {1,9}
D: 1,7,9	{2,7} {2,9} {7,9}
E: 2,3,7,9	

図 1 トランザクションデータベースと頻出集合

図 1 の左にトランザクションデータベースの例がある．ここに，アイテムの集合は{1,2,3,4,5,6,7,8,9}であり，トランザクションの集合は{A, B, C, D, E, F}である．例えば，スーパー等の売り上げデータは，各レコードが「1 人のお客さんがどの商品を買ったか」という記録になっているので，スーパーの全商品の集合をアイテム集合と思えば，トランザクションデータベースになっている．

このほか，アンケート結果のデータも，1つのレコードが，ある人がチェックを入れた項目の集合になっていると考えればよいし，それぞれの項目について，1から5の点数を与えるようなものであれば，「質問と回答の組」(Q1 に 5 がついている，等)がアイテムであると考えることで，トランザクションデータベースとみなせる．このほか，実験データ等もこの範疇に入り，非常に多くのデータベースがトランザクションデータベースであるとみなせる．

2.2 頻出アイテム集合を使ったデータベース分析

さて，このデータを解析しようとすると，どのような視点があるだろうか．視点として，最も多く含まれるアイテムは何か，トランザクションの数はいくつか，トランザクションの平均的な大きさはどの程度か，あるいは大きさの分散はどの程度か，といったものが

連絡先: 有村博紀，北海道大学 大学院情報科学研究科
〒060-0814 札幌市北区北 14 条西 9 丁目
TEL/FAX: 011-706-7680, E-mail: arim@ist.hokudai.ac.jp

思いつくだろう. これらは, いわば, データを全体的に捉えて特徴を見出す手段だと言える.

では逆に, データの局所的な特徴を捉えることはできないだろうか. 局所的というと, 1つ1つのレコードを見る方法が考えられるが, これでは各レコードの個性に解析が引きずられてしまう. また, トランザクションの大きさ, どのアイテムを含むか, といった特徴では, 局所的な個性を語るには弱すぎる. その意味では, 「どのようなアイテム集合が多くのトランザクションに含まれるか」という点に注目することは, 両方の要求を満たしており, 都合が良い. また, 多くのレコードに含まれる組合せに注目することで, 局所的だが全体にある程度共通する構造を見つけることができる. データの中から, 特殊だが顕著な例を見つけるのには適しているだろう.

このように, データベースの中に良く現れるアイテム集合を**頻出アイテム集合** (frequent itemset), あるいは単に**頻出集合**とよぶ. 正確には, **最小サポート**とよばれる閾値 θ を用いて, θ 個以上のトランザクションに含まれるようなアイテム集合として定義される. 図 1 では, 右側に示した集合はどれも少なくとも3つ以上のトランザクションに含まれるため, 最小サポート3に対する頻出集合となっている. アイテム集合Xに対して, それを含むトランザクションを**出現**(occurrence)とよぶ. また, 出現の集合を**出現集合**(occurrence set), 出現の数を**頻出度**(frequency), あるいは**サポート**(support)といい, それぞれOcc(X) とfrq(X) と表記する. 図 1 の例では, アイテム集合{2,5}の出現は 1 行目と 2 行目のトランザクションである.

頻出集合の発見は, 1993 年に Agrawal, Imielinski ,Srikant らによって, 結合規則の発見に関連して提案されたのが最初である [AIS 93].

2.3 頻出アイテム集合発見問題

頻出集合は, 上述のようにデータベースの局所的・特異的だが全体に共通する構造が見られるので, 知識発見の分野で重宝されている.

例えば, 頻出集合に含まれるアイテム同士は, 共起する確率が高いと考えられるため, アイテム間の関係を見ることができるし, 「トランザクションが○と△を含むなら, □を含むことが多い」といったルールを発見する場面で用いられる.

また, 正例と負例からなるデータがあるときに, 正例に多く含まれるアイテム集合を見つけることで, 正例の多くが共通して持ち, 負例のデータには見られない性質を捕らえよう, というアプローチもある. これは 6.5 節の重みつきトランザクションで扱う.

さらに, 頻出集合は多くのレコードに共通して含まれるため, エラーや異常値等に振り回されることなく, 安定している. そのため, 全ての頻出集合を求め, それをデータベースの統計量とみなすことで, データベース間の比較を行うこともできる[ACP 06].

通常, 頻出集合はある種の知識やルールの候補であるため, 1つだけ見つけてもあまり意味がない. また, 統計量として用いる場合は全てを見つける必要がある. そのため列挙的なアプローチが必須である. 与えられたトランザクションデータベースと閾値 θ に対して頻出集合を全て見つける問題は**頻出集合発見問題** (frequent itemset mining), あるいは頻出集合の列挙(enumeration)とよばれる. 頻出度はサポート(support)ともよばれるため, 閾値 θ は**最小サポート**(minimum support)とよばれる.

頻出集合は θ の大きさによりその数が変化する. 計算の立場からは, θ が大きければ解となる頻出集合の数は小さく, 有利であるが, モデルの立場からは, 有用な知識をもらさず見つけ出すため, θ を小さめに設定することが望ましい. ここにはトレードオフがある. θ を上手に設定する, あるいは頻出度の高いものから k 個頻出集合を見つけることで, 解の数を爆発させずに有用な知識を取り出すことが重要である.

2.4 構造データ

XMLデータ, 時系列データ, 文字列データ等, データベースが構造を持つ場合, アイテムの集合だけでなく, 構造を付加したようなパターンを考えることができる. 多くのレコードに含まれるパターンは**頻出パターン**(frequent pattern)とよばれ, 頻出集合と同じようにモデル化することができる. データベースによっては, 構造が持つ意味がとても大きいものもあり, このような場合には, 構造を持つパターンを見つけることに大きな意義がある. 頻出パターンを見つける問題は頻出パターン発見(frequent pattern mining)とよばれ, 多くの研究がある. 7 節で, 順序木パターンの族に対する頻出パターン発見について述べる.

3. 頻出集合列挙アルゴリズム

3.1 計算時間の評価

実用上, 頻出集合発見を用いるような場合, 入力するデータベースは巨大であることが多く, また出力する解の数も大きい. そのため, 計算時間が爆発的に増加する解法は現実的には使用できない. 現実的な時間内に求解を終了させるためには, アルゴリズム的な技術が欠かせない.

一般に, 頻出集合列挙問題のような, 多くの解を列挙する**列挙アルゴリズム**(enumeration algorithm)の計算時間は, 問題を入力して初期化する部分と, 実際に求解を行って解を出力する部分とに分かれる. 入力・初期化に関してはどの実装・アルゴリズムでもほぼ線形時間である. ボトルネックはファイルの読み込み部分であり, アルゴリズムの技術はあまり関係がない.

計算時間に変化が出るのは, 解の数が変化したときである. 一般的にアルゴリズムの求解時間は**入力の大きさ**に対してのみ評価されるが, このように解を列挙するアルゴリズムの場合, **出力の大きさ**, すなわち, この場合は解の数に対する評価も重要となる. そのため, 計算時間が[入力の大きさ＋出力の大きさ]の多項式時間となっているアルゴリズムは**出力多項式時間**(output polynomial)とよばれ, 計算効率の1つの指標となっている. また, より強い条件として, 1つの解を出力してから次の解を出力するまでの時間が, 入力とそれまでの出力の大きさの多項式になっているとき, **逐次多項式**(incremental polynomial)とよばれ, 特にこれが出力に依存せず, 入力のみの多項式時間となっているとき**多項式遅延**(polynomial delay)とよばれる.

アルゴリズムの多項式性は, 問題が構造を持つかどうかの重要な指標であることから, 列挙型のアルゴリズムの多項式性もさかんに議論されている. グラフのパス, 全張木, マッチング, クリークといった構造, シークエンス, 木などのパターンは多項式時間で列挙できることが知られている. 逆に, 多面体の頂点や, 機械学習の分野で著名な極小ヒッティングセット列挙, あるいはハイパーグラフ双対化といった問題, 頻出するグラフパターンを発見する問題等は, 出力多項式時間で解けるかどうかが知られていない.

実用的には, 列挙アルゴリズムの多項式性は効率良さの必要十分条件だが, それだけではあまり意味がない. 列挙問題は非常に多くの解を持つため, 出力の2乗や3乗の時間がかかるアルゴリズムは, 到底動かないのである. 多項式遅延であれば, 解の数に対して線形にしか計算時間が増えず, 実用的である

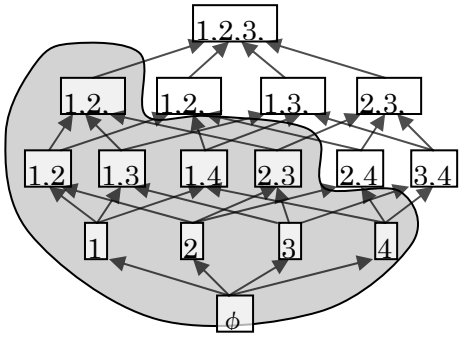

図２　アイテム集合が作る束と頻出集合の領域

が，入力に対する計算時間がより大きいと，巨大なデータベースに対しては動かなくなる．また，実用的には，全体の計算時間が短ければ解の間の計算時間は長くても良く，そういった意味では，解1つあたりの計算時間が入力の低い次数のオーダーになっていることが，実用的に効率良いアルゴリズムの条件となる．

興味深いことに，今までに知られている多項式時間の列挙アルゴリズムは，ほぼ全てこの条件を満たす．加えて，後に述べる末広がり性を考慮した実装を行うことで，解1つあたりの計算時間が定数に近くなる．つまり，出力多項式時間の解法が存在すれば，一工夫加えることで効率良い実装が得られると考えてよい．

メモリの使用量も，列挙アルゴリズムでは大きな問題である．解が多いため，発見した解を全てメモリに蓄えるアルゴリズムは，多大なメモリを消費することとなる．発見した解をメモリに保存する必要があるかないか，という点が，メモリの点での計算効率に関する重要なポイントである．

3.2 アプリオリ法.

あるトランザクションtが頻出集合Xを含むならば，tはXの任意の部分集合をも含む．このことは，Xの部分集合の出現集合はXの出現集合を含み，頻出度はXより同じか大きくなるということを示す．そのため，頻出集合の族は**単調性**を満たす．これは，図 2 で示すようなアイテム集合の束(部分集合間の包含関係を表したグラフ)の上で，頻出集合は下部に連結に存在しているということである．

そのため，空集合から出発し，アイテムを1つずつ追加していくことで任意の頻出集合を得ることができる．この操作を網羅的に行うことでアイテム束上の探索を行えば，全ての頻出集合が見つかる．この探索を幅優先的に行うものが**アプリオリ**(apriori)，深さ優先的に行うものが**バックトラック法**(backtracking)である．

アプリオリはとても有名な手法であり，耳にされたことのある読者も多いことと思う．空集合から出発し，大きさが1の頻出集合を全て見つけ，それらにアイテムを 1 つ追加することで大きさ 2 の頻出集合を見つける，というように，頻出集合を大きさ順で，幅優先的に見つけ出すアルゴリズムである．**アプリオリ**のアルゴリズムを以下に示す．

アプリオリ
1. D_1=大きさ 1 の頻出集合の集合, k:=1
2. while $D_k \neq \phi$
3. 大きさがk+1 であり，D_k の 2 つのアイテム集合の和集合となっているものを全てD_{k+1} に挿入する
4. frq(S)< θ となるSを全てD_{k+1}から取り除く
5. k := k+1
6. end while

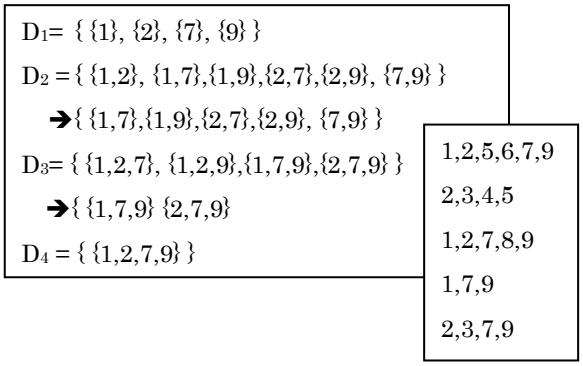

図３　θ =3 でのアプリオリの実行例

このアルゴリズムでは，k回目のループが始まるとき，D_kは大きさが k である頻出集合の集合になっている．そして，ステップ 3 でD_kの集合2つを組合わせてできる大きさk+1 のアイテム集合を全てD_{k+1}に挿入する．頻出集合の単調性から，大きさがk+1 の任意の頻出集合は必ずD_{k+1}に含まれる．そのため，ステップ 4 でD_{k+1}から頻出でないアイテム集合を取り除くことで，大きさ k+1 の頻出集合の集合が得られる．　アプリオリは 1994 年の AgrawalとSrikant らの論文 [AS 94]で最初に与えられた．これと独立に，Mannila と Toivonen らも同じアルゴリズムを与えている [AMST 96].

アプリオリのステップ 4 で Dk+1 の各アイテム集合の頻出度を計算する際に，次のアルゴリズムを使うと効率的である．

1. for each S∈D_{k+1} do frq(S) := 0
2. for each transaction t∈T do
3. for each S∈D_{k+1} do
   if S⊆t then frq(S) := frq(S)+1
4. end for

まずステップ 1 で全てのアイテム集合 S について frq(S)を 0 に設定し，ステップ 2 で1つずつトランザクションを選び，それが含むアイテム集合 S に対して frq(S)を1増加させる．全てのトランザクションに対してこの処理をすると，frq(S)は S の頻出度となる．

アプリオリの計算時間は，(候補の総数)×(データベースの大きさ)に比例し，候補の総数は頻出アイテム集合の数の n 倍(n はアイテムの数)を超えないことを考えると，頻出集合1つあたり，最悪でも(データベースの大きさ)×n に比例する時間となることがわかる(本稿では，データベースの大きさとはデータベースの各トランザクションの大きさを合計したものとする)．頻出集合を全てメモリに保持する必要があるため，メモリ使用量は見つける頻出集合の数に比例する．アプリオリは，候補を生成する時間が入力の多項式時間とならないため，多項式遅延とはならない．

アプリオリの特徴は，データベースへのアクセス数の少なさである．上記の頻出度計算方法は，トランザクションを一回スキャンすることで全ての候補の頻出度を計算することができる．そのため，もしデータベースが巨大でメモリに入らないような場合でも，データベースを(頻出集合の大きさの最大)回だけしかスキャンしない．

図 3 でアプリオリの実行例を見てみよう．この図は，アプリオリの実行に伴い，各D_kがどう計算されるかを示している．アプリオリはまず大きさ 1 の頻出集合を全て求め，その集合をD_1とする．これが 1 行目の集合である．次にD_1の2つのアイテム集合の和集合で大きさが 2 のものを全てD_2に挿入する．それが 2 行目

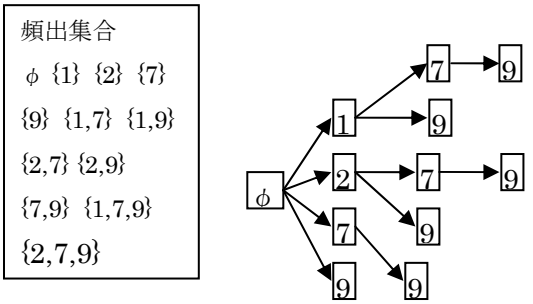

図 4 頻出パターン木

の集合である. ここから頻出度が 3 未満のものを全て取り除くと, 3 行目の集合が得られる. 以下同様にして, D_3, D_4と求める. D_4は頻出集合を1つも含まないため, 空集合となり, ここでループが終了して, アルゴリズムは終了する.

3.3 プルーニング

アプリオリの高速化する方法として, **プルーニング**(pruning)がある. XをD_{k+1}のアイテム集合とする. もしXの大きさkの部分集合でD_kに含まれないものがあれば, Xは頻出集合でないアイテム集合を含むということがわかる. よって, Xは頻出集合となることはない. この検査をD_{k+1} の全てのアイテム集合に対して行えば, 頻出度の検査対象を少なくすることができる. 例えば, 図2でD_3に最初に含まれている{1,2,7}は, D_2に含まれないアイテム集合{1,2}を含んでいる. このことから, {1,2,7}は頻出集合ではないと, 頻出度の計算をする前に結論付けられるので, ただちにD_3から取り除くことができる.

3.4 頻出パターン木

アプリオリは, 各Dkをメモリに保持する必要がある. これを, トライ (trieあるいはprefix tree)のような木構造で保持する. 本来トライは文字列を格納するものであるので, 各頻出集合を添え字順にソートした列を文字列とみなし, 格納する. 任意の頻出集合から1つアイテムを取り除いたものは頻出集合である, という単調性から, メモリの使用量は頻出集合の数に比例する量になる. このように, 頻出集合を保持するためのトライ型の木構造を**頻出パターン木**(frequent pattern tree, あるいはFP-tree)とよぶ.

図 4 に例を示した. 根から始まり, 任意の頂点で終了するパスに対して, そのパスを構成する頂点の集合が頻出集合になっている. 後述するように, トライは入力したデータベースを効率良く格納するのにも使われている. FP-growth [HPY 00] は, 深さ優先型のアルゴリズムにおいて, トランザクションデータベースを, 組を表す要素列のトライとして表現したものである. ZBDD-growth [MA 07] は, このアイディアを拡張して, データベースをZBDD（論理関数を表す縮約された DAG）で表している.

3.5 バックトラック法

アプリオリが幅優先的な探索を行うのに対し, **バックトラック法**は深さ優先的な探索を行う. アイテムを1つずつ追加する, という方法で網羅的に頻出集合を生成すると, 大きさkのアイテム集合を 2^k回生成してしまう. 通常, グラフ探索においては, 重複を避けるために一度訪れた頂点にマークをつける. 頻出集合列挙の場合には, この作業は見つけた頻出集合をメモリに保存しておくことに相当する. しかし, こうせずとも簡単な方法で重複は回避できる.

アイテム集合Xに対して, その中の最大のアイテムを**末尾**(tail)とよび, tail(X)と表記する. ただしtail(φ)は－∞とする. 例

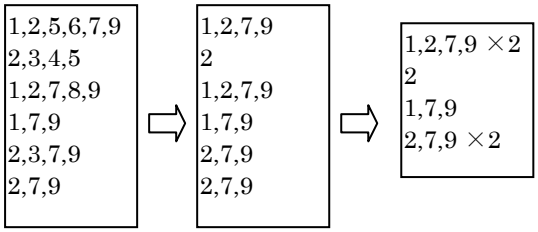

図 5 データベース縮約

えば, アイテム集合{1,3,5}の末尾は 5 である. そして, 各アイテム集合Xを生成する際には, tail(X)を最後に追加する, という生成ルールを定めよう. つまりXは, 末尾を含まない集合X – tail(X)からのみ生成する, ということである. このようにすれば, X が複数のアイテム集合から生成されることがなくなり, X – tail(X)も 1 度しか生成されないため, Xも 1 度しか生成されない. このように, XをX - tail(X)から生成する方法を**末尾拡張**(tail extension)とよぶ.

例として, {1,3,5,6}は{1,3,5}の末尾拡張だが, {1,2,3,5}はそうでない. アイテム集合 X から末尾拡張をしてアイテム集合 Y が得られたとき, tail(X)<tail(Y)が成り立つ. つまり, Y は必ず tail(X)より大きなアイテムを追加して得られる. また, tail(X)より大きなアイテムを追加したものは必ず X の末尾拡張になっている.

以上から, 次のアルゴリズム**バックトラック**が得られる. バックトラック(φ)を呼び出すことで, 全ての頻出集合が列挙される.

```
バックトラック(X)
1. output X
2. for each i>tail(X) do
3.    if frq(X∪{i})≧θ then call バックトラック(X∪{i})
```

バックトラック法の長所は, 既に発見した解をメモリに保存しておく必要がないため, 基本的に入力したデータベースを保持するだけのメモリがあれば動く点である. 計算量はアプリオリと同じで, 1つあたり(データベースの大きさ)×n に比例する時間がかかる. また, 各反復で解を必ず1つ出力するため, 多項式遅延となる. しかし, もしデータベースがメモリに入りきらないような場合, 各反復でデータベースをスキャンすることになるため, 少なくとも頻出集合の数だけデータベースのスキャンを行うこととなり, 効率が落ちる.

図 5 は, 図 1 の頻出集合をバックトラック法で列挙した図である. バックトラック法は, 空集合から出発し, アイテムを 1 から順に追加していく. まず 1 を追加した際に, frq({1})≧3 が成り立つので再帰呼び出しを行う. {1}を引数として呼び出された反復では, tail({1})=1 より大きなアイテムを追加する. 2,…,6 までは, 追加しても頻出集合とはならないため, 再帰呼び出しは行われず, 7 を追加した際に frq({1,7})≧3 が成り立つため, 再帰呼び出しが行われる. 以後同様に処理が進められる. {1,7}に関する再帰呼び出しが終了した後, {1}に関する反復では, 次に 8 を追加する. これは頻出ではないので, 再帰呼び出しを行わず, 次に 9 を追加する. これは頻出となるため, 再帰呼び出しが行われる.

バックトラック法に基づくアルゴリズムは, 1998 年から 2000 年頃にかけて複数の著者によって提案された. [BJ 98] は, 極大集合発見のために末尾拡張を用いるアイディアを提案している. [HPY 00, MN 99, MS 00, ZPOL 97]は末尾拡張を用いた頻出集合発見について述べている.

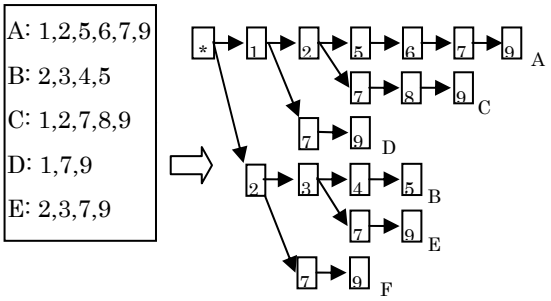

図 6 トライを用いたデータベースの圧縮

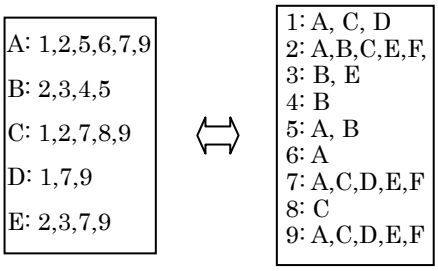

図 7 データの水平配置と垂直配置

4. 高速化の技術

前述のアプリオリ, バックトラックともに, 計算量の上では出力数依存のアルゴリズムであり, 出力数多項式時間を達成している. しかし, 実際にはデータベース全体をスキャンするという作業は非常に時間がかかり, 素直な実装ではとても現実的な時間内に求解できない. そのため, いくつかの高速化技法が提案されている.

4.1 データベース縮約

最初の技法は, データベース縮約とよばれるものである. これは, 入力するデータベースから, あらかじめ不要な部分を除去することでデータベースを小さくし, 空間コストを下げるとともに, スキャンにかかる時間を短縮するものである.

アイテム i を含むトランザクションの数が θ 未満であるとき, i を含む任意のアイテム集合の頻出度は θ 未満になり, 頻出集合とならない. よって i は追加するアイテムの候補からはずしてよく, さらには, データベースからアイテム i を全て除去しても計算結果に変化はない. また, 全てのトランザクションに含まれるアイテムは, 任意のアイテム集合に追加しても頻出度を変化させないため, 同様に不要であることがわかる. これらを取り除くことで, データベースを縮小できる. さらに, もし同一のトランザクションが k 個あれば, それらは1つに併合し, 併合された数 k を重みのような形で保存することができる. この2つの操作により, 現実的なデータは, 特に最小サポートが大きい場合, 大幅に縮小することができる.

図5に最小サポートを 3 としてデータを縮約した例を示した. 左端のデータベースから 3 つ未満のトランザクションにしか含まれないものを除いたものが真ん中のデータベースであり, さらに同一のトランザクションを併合したものが右である.

4.2 トライを用いたデータベースの圧縮

入力したデータベースを小さくするには, データベース縮約のほかにも, 頻出パターン木と同様にして, 図 6 のようにデータベースをトライを用いて木構造として保持する方法がある.

頻出度の小さいアイテムをデータベースから取り除く点は同じだが, 取り除いて得られるデータベースを, トライで格納するところが異なる. トライで格納することで, 自動的に同一のトランザクションは1つに縮約される. 頻出パターン木と同様, 各トランザクションは添え字順にソートして格納する. トライによる圧縮効率はアイテムに対する添え字のつけ方によるが, 一般に頻出度の大きいアイテムに小さな添え字をつけることで, トランザクションの多くの接頭辞が共有され, 圧縮効率が高くなる.

4.3 ダウンプロジェクト

バックトラック法と組み合わせて効果を発揮するのが, 次に述べる**ダウンプロジェクト**(down project)である.

今, あるアイテム集合 X にアイテム i を追加して Y を作り, その頻出度を計算したいとする. このとき, Y を含むトランザクションは必ず X を含むので, Y の出現集合は X の出現集合に含まれることがわかる. よって, Y の頻出度は, X の出現集合のみから求まる. さらに, Y の出現集合 Occ(Y)は, X の出現集合 Occ(X)とアイテム集合{i}の出現集合 Occ({i})の共通部分である. つまり, 任意の $Y = X \cup \{i\}$ に対して, 次の等式が成立する:

$Occ(Y) = Occ(X) \cap Occ(\{i\})$.

この, 共通部分をとる作業は, データベース全体をスキャンするよりはるかに短時間でできる. この手法をダウンプロジェクトとよぶ.

例えば図 5 で, {1}に 5 を追加する際には$Occ(\{1\})=\{A, C, D\}$, $Occ(\{5\})=\{A, B\}$であるので, $Occ(\{1\}) \cap Occ(\{5\})=\{A\}$となる. ダウンプロジェクトを行う際には, トランザクションデータベースを, 各iに対する$Occ(\{i\})$, つまり「各アイテムが含むトランザクションの集合」として保持すると都合が良い. このデータの保持の仕方を**垂直配置**(vertical layout)とよぶ. それに対して, 各トランザクションに対して, それに含まれるアイテムを保持する方法をデータの**水平配置** (horizontal layout)とよぶ. 図 7 に例を示した.

ダウンプロジェクトとデータの水平配置は, 1990 年代終わり頃に, バックトラック型のアルゴリズム[MN 99, MS 00, ZPOL 97]と共に導入された. また, FP-growth [HPY 00]もダウンプロジェクトの変種といえる. このバックトラック型アルゴリズムとデータの水平配置を合わせて, **パターン成長法**(pattern growth method) [PHPCDH01] と呼ぶこともある.

4.4. ビットマップ表現

データベースが密であり, トランザクションが平均的にアイテム集合の何割かのアイテムを含む場合, ビット演算を使うことでダウンプロジェクトを効率化できる. 各$Occ(\{i\})$, および$Occ(X)$をビット列で保持しよう. すると, $Occ(\{i\}) \cap Occ(X)$は単なるビット演算のANDでトランザクション 32 個(あるいは 64 個)ずつ一度に計算できる. そのため, 非常に高速化される. この方法は**ビットマップ**(bitmap)とよばれる.

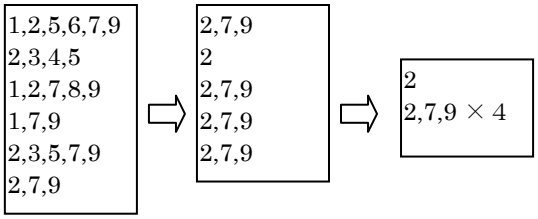

図 8 再帰的データベース縮約： アイテム集合{2}に対する条件付データベース

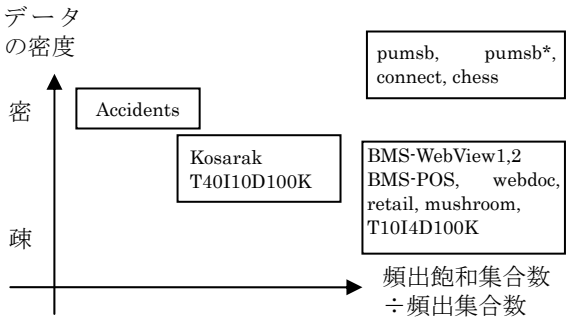

図 9 FIMI データセットの分類

4.5. 振り分け

振り分け(occurrence-deliver)はダウンプロジェクトをまとめて行うことでさらに高速化する技法である [UAUA04, UKA 04]. これは, 異なるアイテムの種類が非常に多い場合など, 疎なデータベースにおけるバックトラック法の高速化において有用な技法である.

今, アイテム集合 X にアイテム 1, …, n 各々を加えて X より1つ大きなアイテム集合を作成し, その頻出度を全て計算する, という操作を考える. この作業は, ダウンプロジェクトを使うと, データベース全体を 1 回スキャンするのと同じ時間がかかる.

t を X の出現とする. t がアイテム i を含むなら, またそのときに限り, X∪{i}は t に含まれる. そこで, t＼X に含まれるアイテム全てについて, t に含まれる, というマークをつけよう. この作業を, X の出現全てに対して行うと, アイテム i についたマークの集合は, X∪{i}の出現集合になる.

これが**振り分け**であり, 次のアルゴリズムとなる.

振り分け(X)
1. for each t∈Occ(X) do
2. for each i∈t, i>tail(X) do
3. i に t のマークをつける
4. end for

振り分けは, 水平配置で記述されたデータベースから, その垂直配置を求めることに相当する. そのため, 振り分けの計算時間は X の出現集合が作るデータベースの大きさに比例する. これは X の頻出度が小さくなれば小さくなるほど, ダウンプロジェクトに比べて高速化されるということを意味する. バックトラック法の場合, i>tail(X)を満たす i のみについて頻出度の計算をするため, 各出現の, tail(X)より大きなアイテムのみについて, マークを付加する. 各トランザクションのアイテムを添え字順でソートしておくことで, 効率良く実行できる.

データベースが疎であると, 多くのアイテム i に対して X∪{i}の頻出度が 0 となる. このため, ダウンプロジェクトは多くの頻出度計算を行っても少ない頻出集合しか得られなくなり, 解 1 つあたりの計算時間は大きくなる. 振り分けの場合は, マークがついたアイテムのみに関してのみ頻出度を見ればよく,「頻出度が 0 となるアイテムはそもそも調べない」ということができる.

4.6. 再帰的データベース縮約と末広がり性

バックトラック法のためのもう一つの有効な高速化技法が, 再帰的データベース縮約である.

バックトラック法の各反復では, 必ずtail(X)より大きなアイテムのみが追加される. そのため, 再帰的に呼び出された反復の中でも, 追加されるアイテムはtail(X)より大きい. つまり, Xを受け取る反復, およびそこで呼び出される反復では, tail(X)より小さいアイテムは不要である. Xの各出現からtail(X)より小さなアイテムを全て除去すると, スキャンすべき部分のみを持つデータベースができる. これを **条件付データベース**(conditional database)とよぶ.

条件付データベースの中では, 各アイテム集合の頻出度はもともとのデータベースでの頻出度と同じかまたは小さくなる. そのため, 条件付データベースをさらにデータベース縮約すると, より小さいデータベースが得られる. 条件付データベースを作成し, データベース縮約を行う, という作業を各反復で行うと, 再帰的にスキャンすべきデータベースは小さくなる. これを**再帰的データベース縮約**(recursive database reduction)とよぶ. 図 8 に最小サポートが 3 であるとき, アイテム集合{2}に対する条件付データベースと, それを再帰的に縮約したものを示した.

バックトラックのような列挙形のアルゴリズムは, 一般に各反復で再帰呼び出しを複数回行う. そのため, 再帰呼び出しが作る計算木の構造は, 根の部分が小さく, 葉の方へ進むほど頂点数が多くなる末広がり的な構造を持つ. つまり, 深いレベルの反復での計算時間が全体の計算時間のほとんどの部分を占めるようになる. そのため, 上記の手法のように, 反復が深くなれば深くなるほど計算時間が縮小されるような工夫を加えることで, 大きな計算時間の改善が行えるのである. この性質は多くのパターンマイニングアルゴリズムを始め, 列挙アルゴリズム全般に成り立つ性質である.

実験的には, このような改良を行わない場合, 解の増加とともに計算時間は増加していくが, 再帰的に計算時間を減少させると, ある程度解数が増えるまでは初期化の部分がボトルネックとなり, 計算時間の増加はみられず, 解が十分に大きくなったとしても, 解の数に比例はするものの, ゆるやかに計算時間が増大していく. そのため, 十分解の数が大きくなると, 解の数のみに計算時間が依存するようになり, 現実的に非常に優れた解法となる.

再帰的データベース縮約は, 2000 年代に入って PrefixSpan [PHPCDH01]や LCM [UAUA 04, UKA 04]等のアルゴリズムで用いられた.

5. 実装の比較

さて, これらのアルゴリズムと技術の組合せで, 実際のデータに対して有用なものはどれであろうか. この疑問を解決すべく, FIMI(frequent itemset mining implementations)ワークショップ

[FIMI HP]にていくつもの実装の比較が行われているので，その結果を紹介しよう．

5.1 データセットの特性

FIMIワークショップで用いられたデータセット(**FIMIデータセット**と呼ぶ)は，現実問題から得られたものが中心である．POSデータ，webのクリックデータ等があり，多くがトランザクションの平均サイズが 10 未満と疎であるが，パワー則を満たし，大きなトランザクションもいくつか含まれている．また，密なデータセットもあり，特に機械学習のベンチマークから来た問題は密である．

図 9 にデータの密度と構造に基づいたデータセットの分類を示した．縦軸が密度，つまり全アイテム数に対する各トランザクションの平均的な大きさである．疎なものはアイテム数 10000 に対してトランザクションの平均的な大きさが 5-6 程度である．密なものは，多くのアイテムが半分以上のトランザクションに含まれる．横軸は，解数が十分に大きいような（データベースの大きさと同程度）最小サポートでの，頻出飽和集合数÷頻出集合数である．この値が小さいと，データはランダム性を持ち，大きいと，データは構造を持つと考えてよい．

まず，全体的な傾向として，頻出集合数がデータベースの大きさに対して十分小さい場合，計算のボトルネックは問題の入力と初期化になる．そのため，トライ等を用いない，シンプルなアルゴリズムが高速となる．しかし解数の増加とともに両者はすぐに逆転し，入力と出力がほぼ等しい大きさになるころには両者の差は開く．

5.2 アプリオリ法 vs. バックトラッキング法

アプリオリは，スキャンの回数が少ないためデータベースがメモリに入らない場合でも高速であること，グラフマイニング等の，包含関係の判定に時間がかかる場合でも，既発見解を用いて包含関係の判定を少なくできることが利点である．しかし，このワークショップで用いられたデータは十分メモリに収まる大きさであり，かつ包含関係の判定は短時間でできる．そのため，アプリオリ型のアルゴリズムはどれも悪い成績を収めており，特に最小サポートが大きく，かつ解数が大きい場合 100 倍を超える差が出ることもある．

5.3 頻度計算とデータベースの格納方法

ダウンプロジェクトと振り分けは，振り分けのほうが高速であるようだ．特に疎なデータではその差が大きい．データベース縮約は，最小サポートが大きくかつ解数が小さい場合に有効であるが，疎なデータ等，最小サポートが 10-20 と非常に小さい場合には効果がない．図 9 で上のほうに位置するデータに有効である．

再帰的なデータベース縮約も同じであるが，特に密なデータで解数が大きくなったときの計算時間の改善率は劇的であり，使用していない実装では計算が終了しないような問題でも解くことができる．

ビットマップは，データベースが密である場合に多少有効であるが，データベース縮約と組合せるとビットマップの再構築に手間がかかるため，解数が多い場合の計算効率が悪い．そのため，効果は限定的である．

入力をトライで格納する方法は，疎なデータでは効果が小さく，かえって何もしないほうが速いことが多い．しかし密なデータでは効果があり，非常に密なデータ，図 9 で上部に位置するデータでは 5-6 倍の高速化ができている．

高速な実装では，計算のボトルネックは，ほぼどの問題でも解の出力部分であり，その意味では，十分解の数が大きければ，頻出集合列挙は，現実的にはほぼ最適な時間で行えると考えて良いだろう．

6. 他の問題への適用

頻出集合列挙はそれ自体有用なツールであるが，他の問題を帰着して，幅広い問題を解くためにも使える．最近では，多くの頻出集合列挙問題のプログラム実装が公開されているので FIMI HP, 宇野 HP], この種のアプローチも有用と思われる．ここでは，そのような頻出集合列挙の使い方を解説しよう．

6.1 マルチセットの発見

頻出集合列挙では，トランザクションに同一のアイテムが複数個含まれることはないとしていた．しかし，実際のデータでは複数個のアイテムが含まれる場合もある．つまりトランザクションを集合でなく**マルチセット**（多重集合）として扱いたい．このようなデータから，同一のアイテムを複数個含むことをゆるした頻出集合を見つける問題を**マルチセット発見問題**とよぶ．

マルチセット発見問題を頻出集合列挙に帰着する際には，アイテムiがトランザクションtにk回現れているとき，全てのiを取り除き，かわりに$i_1,\cdots,i_k$を挿入する，という変換を行う．i_jは「j個目のアイテムi」という意味である．すると，iがj個入ったマルチセットはアイテム集合$\{i_1,\cdots,i_j\}$に対応し，両者の頻出度は等しい．そのため，変換したデータベースで頻出集合を見つけることで，元のデータベースの頻出マルチセットを見つけることができる．

ただし，$\{i_3,i_4\}$のように，マルチセットに対応しない頻出集合も見つかるため，効率は落ちる．しかし，このような不要な頻出集合は必ず飽和集合にならないため，頻出飽和集合を列挙することで，これらの不要な頻出集合を自動的に取り除くことができる．

6.2 頻出部分構造パターン問題の帰着

一般に，任意の頻出パターン発見問題は，頻出集合発見問題に帰着して解くことができる．レコード t に含まれるパターン全てからなる集合を作り，レコード t をこの集合で置き換える．例えば，各レコードが木であるデータベースから頻出する木を見つけたい場合は，各レコード t に対して，t に含まれる部分木を全て列挙し，その集合を新たなレコードとして t を置き換える．各パターンがアイテムであるとみなすと，置き換えたあとの t はトランザクションであるとみなせる．よって，全てのレコードを置き換えて得られるデータベースの頻出集合を列挙することで，元のデータベースの全ての頻出パターンを見つけることができる．

この方法でも，マルチセットのマイニングのように不要な頻出集合が多く見つかる．飽和集合を求めることで多くは自動的に回避できるが，例えば，ある飽和集合 X の出現集合に対応するレコードが，包含関係にないパターンをいくつも共通して含む場合，それらが全て飽和集合に含まれてしまう．そのため，各飽和集合は，冗長な情報を持つこととなる．

また，レコードが大きいと，問題の変換時のパターン列挙で膨大な時間がかかる．そのため，全てのレコードがあまり大きくない，という状況で使用することが望ましい．

6.3 大きな共通部分を持つトランザクション集合

よく知られた関係として，トランザクションデータベースは 2 部グラフと等価である．つまり，アイテムの集合と，トランザクションの集合を頂点集合とし，あるアイテム X がトランザクション t に含まれるとき，X と t の間に辺が存在するような 2 部グラフである．

このとき，アイテムとトランザクションは対称な関係にあり，両者の立場を入れ替えたデータはやはりトランザクションデータベースとなる．このデータベースに対して頻出集合を発見すると，それはある一定数以上の大きさの共通部分を持つトランザクションの集合になる．例えば，トランザクションデータベースから，共通部分の大きいトランザクションの組を見つけたい場合は，この変換を行って大きさが 2 の頻出集合を見つけることで，全てのそのような組を見つけられる．

6.4 top-K 頻出集合の発見

頻出集合発見のモデルの面での弱点に，適切な量の解を得ようとすると，最小サポートを調整して何度も解きなおさなければならない，という点がある．そのため，逆に解の数Kを指定し，頻出集合の数がKになるような最小サポートを求めよう，という**top-K頻出集合発見**が研究されている．

アルゴリズムのアイディアは簡単で，最初最小サポートを 1 にして頻出集合発見アルゴリズムを走らせ，K 個の解を見つけた時点で，解集合として保持し，最小サポートをその中で最も小さな頻出度に設定する．以後は，新たに最小サポートより頻出度の大きな解が見つかるたびに，解集合から頻出度が最も小さな解を取り除き，最小サポートを更新する，という作業を繰り返す．アルゴリズムが終了したときに，解集合は K 番目までに頻出度が大きなアイテム集合の集合になっている．実データでのパフォーマンスはそれほど悪くなく，直接，解が K 個となる最小サポートを与えて求解するのに比べ，2-3 倍しか時間がかからない．

6.5 重み付きトランザクションと最適集合発見

頻出集合発見では，通常トランザクションには重みがなく，どのトランザクションに含まれていても重要性は等価であるとみなされる．しかし現実の問題では，重みが付いたデータを扱う必要が生じる．また，これをさらに拡張して**正負の重みが付いたデータ**を考えることも，データの分類では有用である．例えば，正例と負例を分類する場合など，「含まれてほしくないトランザクション」が存在するときは，負の重みを考慮するほうが，都合が良い [MS 00]．このように，頻出集合をデータの分類に用いて分類精度やその他の目標関数が最大の頻出集合を求める最適アイテム集合発見は，**経験リスク最小化**や，**ブースティング**など，各種の機械学習問題でしばしば現れる [KMM 04, TK 06]．

正負の重みをゆるした場合には，もはや頻出集合は単調でないため，出力数に依存したアルゴリズムの構築は難しくなる．また，理論的には広い範囲の評価関数に対して，経験誤差や経験リスクの最小化問題は，アイテム集合や多くのパターンクラスに対して，近似的にすら計算困難であることが知られている [Mo 98, SAA 00]．

しかし，頻出集合列挙のアルゴリズムをベースにして，正重みに対する最小サポート[KMM 04]や，目標関数の凸性を用いた枝刈り[MS 00, TK 06]等を導入することにより，ある程度効率の良いアルゴリズムを構成することができる．この場合，出力数に対する計算時間は非常に長くなるが，使用に耐えないほどではなく，データ解析の手法として，1つの良い道具となるだろう．

7. 頻出順序木パターン発見

7.1 ラベル付き順序木と頻出パターン発見

構造データからの頻出パターン発見に対して，これまでに述べてきた頻出集合列挙の技術を適用できる [浅井 04, WM03]．ここでは，例として，順序木からの頻出部分構造の発見アルゴリズムを説明する．

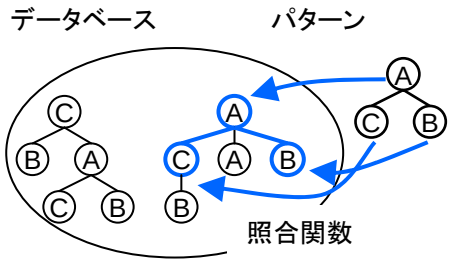

図 10 順序木パターン

頻出部分集合発見問題では，図 10 に示したように，データとパターンはともに順序木で表される．頂点ラベルの集合 Σ に対して，Σ 上の**ラベル付き順序木**(labeled ordered tree)とは，図 10 に示したように，各頂点が Σ の要素でラベル付けられた根付き木 T である．Tにおいて各頂点の子の順序は意味をもつ．すなわち，子の順序を入れ替えて得られる木は別の順序木であると考える．

グラフデータベースは，各レコードがラベル付き順序木であるようなデータベース $D = \{D_1, \ldots, D_m\}$ であり，**パターン**は任意のラベル付き順序木である．パターンである順序木 S が，レコードTに**出現する**とは，図 10 のように，順序木Sの頂点ラベルと辺をうまく保存したまま，Sを順序木Tの一部分にうまく埋め込むような照合関数 f が存在することと定義する．このとき，S≦Tと書き，SはTの部分木であるともいう．データベースにおけるパターンSの**出現**は，照合関数の全体TOcc(S) = { f : 照合関数 f によってSはある D_i に出現する } や，文書出現DOcc(S) = { i : S はある D_i に出現する }，根出現 ROcc(S) = { x : S の根はあるレコードD_iの頂点 x に出現する }などさまざまな定義が考えられる．ここでは，さまざまな良い性質から，根出現にもとづく出現の定義 Occ(S) = ROcc(S)を採用する．

このとき，パターンSの頻度はfrq(S) = |ROcc(S)|である．閾値 $\theta \geqq 1$ に対して，もしfrq(ならば θ ならば，SをD上の頻出パターン(frequent pattern)という．与えられたグラフデータベースDと閾値 θ に対して，D上の頻出順序木を全て見つける問題を**頻出順序木発見問題**(frequent tree mining) という．

7.2 素朴な方法の問題点

根出現または文書出現にもとづく頻度を採用したとき，順序木パターンの族は，頻度に関する**単調性**を満たす．すなわち，部分木関係≦に関して，もしS≦Tならばfrq(S)≧frq(T)が成立する．これは，部分木関係≦に関して順序木全体がつくる束構造で，小さな木は下方に，大きな木は上方にあるとすると，頻出パターンは束の下半分に連結して存在することを意味する．アイテム集合場合と同様に，サイズがゼロの空木から出発して，頂点を一つずつ追加していくことで，任意の頻出順序木を得ることができる．

しかし，一般に上記のやり方で単純に網羅的な探索を行うと，サイズkの順序木を 2^k回以上重複して生成してしまう．アイテム集合の場合と同様に，すでに見つけた頻出順序木を メモリに保存して，幅優先探索を行うことでこの問題は解決できる．しかし，次のように注意深く探索を行うことで，メモリへの保存を回避して，バックトラック法を用いた深さ優先探索を行うことが可能である．

7.3 順序木の系列表現

鍵となるアイディアは，順序木を頂点とその深さの対からなる系列として表現することである．まず，順序木の各頂点vを，その

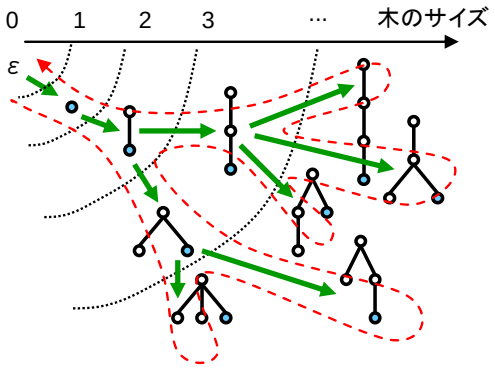

図 11　最右拡張による順序木の列挙

深さ(根からのパスの長さ)とラベルの対$\mathbf{v}$ =〈dep(v),lab(v)〉で表す. サイズkの順序木Sの**深さラベル列**(depth-label sequence)は, Sの全頂点の深さとラベルの対を, S上の前置順巡回(pre-order traversal)の順に並べた系列code(S)= $\mathbf{s} = (\mathbf{v}_1, \ldots, \mathbf{v}_k) = (\langle \mathrm{dep}(v_1), \mathrm{lab}(v_1) \rangle, \ldots, \langle \mathrm{dep}(v_k), \mathrm{lab}(v_k) \rangle)$である. アイテム集合の場合と同様に, 順序木の深さラベル列$\mathbf{s}$に対して, 最後の対$\mathbf{v}_k$を**末尾**とよび, tail($\mathbf{s}$)と表記する. このとき, 先頭の対$\mathbf{v}_1$はSの根に対応し, 末尾$\mathbf{v}_k$はSの最も右側にある葉に対応する. 例えば, 図 10 のパターンS_1は系列$\mathbf{s_1}$= (〈0, A〉, 〈1, C〉, 〈1, B〉)で表わされ, その末尾はtail($\mathbf{s_1}$)= 〈1, B〉である. 簡単のため, 以後は順序木と深さラベル列を区別しないことにする.

7.4 最右拡張を用いたバックトラック法

頻出順序木の列挙では, 親となるサイズk－1 の順序木Sに新しい頂点vを一つ追加して, 子となるサイズkの順序木 T を生成する. この際に, 新しく追加した頂点vが, Tの深さラベル列上でv = tail(T)を満たすようにするという生成ルールを用いる. これは頂点の追加に際して, (1)親の木の**最右枝**(rightmost branch)の直下で, (2)兄弟中で末弟(最も右の頂点)となる場所にのみ追加するという制限を加えることになる. これを**最右拡張**(rightmost expansion)という [AAKASA 02, Nakano 02, Zaki 02]. 深さラベル列の言葉でいえば, Sの最右葉すなわち末尾 tail(S)の深さをdep(tail(S))とすると, 任意の深さ 1≦dep≦dep(tail(S))+1 と任意のラベルlabからなる対〈dep, lab〉を, 列Sの末端に追加することに対応する.

以上をまとめると次のアルゴリズム **最右拡張バックトラック**が得られる. サイズ 0 の空木 ε (すなわち空の深さラベル列)に対して, 最右拡張バックトラック(ε)を呼び出すことで, 全ての頻出集合が列挙される.

最右拡張バックトラック(S)
1. output S
2. for each 1≦*dep*≦dep(tail(S))+1 do
3. for each label *lab* do
4. if frq(S・〈dep, lab〉) ≧ θ then
5. call 最右拡張バックトラック(S・〈dep, lab〉)

ただし, 空木 ε には無条件に一つの頂点が追加可能とする. 図11に最右拡張による順序木の列挙の例を示す. バックトラックアルゴリズムが, サイズ 0 の空木から出発して, 最右の葉(影で示されている)を追加しながら, サイズの小さな順に全ての頻出順序木をもれなく, かつ重複なく列挙することがわかる.

7.5 頻度計算の高速化

実用的な実装のためには, 生成された木Sの頻度frq(S)を高速に計算することが重要である. そのために, 各順序木パターンSに対して, 最右出現とよぶ中間的なデータを保持し, これを最右拡張と共に漸増的に更新することで, 効率よい頻度計算を行える. **最右出現**(rigihtmost occurrence)とは, このデータベース中でSが照合関数によって埋め込まれている全ての埋め込みにおけるSの最右葉(すなわち末尾頂点)の出現位置をリストに保持したものである.

根出現の代わりに最右出現を用いることで, 4 節で議論したダウンプロジェクトや振り分け等のアイテム集合発見の高速化技法がそのまま拡張できる. また, 最右出現から根出現にもとづく頻度が低いコストで計算できる. 詳細は[AAKASA 02, 浅井 04]を参照されたい.

7.6 グラフマイニング

最右拡張は, 無順序木や一般のグラフなど, 順序木以外の構造データに対して正規形表現(canonical representation)等の技法と組み合わせて, バックトラック型の頻出パターン発見アルゴリズムを実現するために用いられている[浅井 04, YH 02]. PrefixSpan も同様の技法である. AGM 等の幅優先型のグラフマイニングアルゴリズムにおける隣接行列の正規形表現の利用も, より一般的なスキーマとみなすことができる.

注意点として, 最右拡張によるバックトラック型のアルゴリズムは, 順序木パターンにおいてはきわめて効率が良いが, より一般のグラフ族に対するマイニングにおいては, 単独では性能が出ないことがある. 例えば, 一般のグラフに対する最右拡張の構成は難しく, グラフの同型性判定による検査と組み合わせる必要がある[YH 02]. また木より一般的なグラフ族に対しては, 最右拡張とダウンプロジェクトの組み合わせは, 探索の枝刈りが十分に利かない場合があるので, Apriori のような幅優先探索[IWM 03]を採用するか, グラフの族の性質を考慮した枝刈り手法を, バックトラック型に組み込む等の工夫が必要になる.

8. まとめ

本稿では, 頻出アイテム集合発見問題を, アルゴリズムを中心にして解説した. 飽和集合マイニングや一般のグラフマイニング等, 本稿で触れられなかった話題に関しては, [宇野 07,浅井 04, WM 03,鷲尾 07]等の解説を参照されたい.

頻出集合はデータマイニングの基礎であり, 実際にツールとして使う機会も多い. 本稿で紹介したアルゴリズムの多くは FIMI のサイトに実装と解説論文があり, またベンチマーク問題も手に入る. また, 著者のホームページ[宇野 HP]では, LCM の他, いくつかの頻出パターン発見アルゴリズムの実装と問題変換等各種のツールが入手できる. 例えば, 3.5 節のバックトラック法による頻出集合発見アルゴリズムに, 4 節の述べた各種の高速化手法を加えた高速版実装に, 飽和集合発見を追加したものは LCM ver.4 として, 6.6 節の重み付きトランザクション対応版の実装は LCM ver.5 として, また, 7 節の頻出木マイニングは Freqt ver.4 と Unot として, 同ホームページで利用可能である.

本稿と実装がデータマイニング研究推進の助力となれば幸いである.

参考文献

[AIS 93] R. Agrawal, T. Imielinski, A. N. Swami: Mining Association Rules between Sets of Items in Large Databases. Proc. SIGMOD Conference 1993, pp. 207-216 (1993)

[AMST 96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo: Fast Discovery of Association Rules. Advances in Knowledge Discovery and Data Mining, pp. 307-328 (1996)

[AS 94] R. Agrawal, R. Srikant: Fast Algorithms for Mining Association Rules in Large Databases. Proc. VLDB 1994, pp. 487-499 (1994)

[AAKASA 02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, S. Arikawa, Efficient Substructure Discovery from Large Semi-structured Data, Proc. 2nd SIAM Data Mining (SDM'02), pp. 158-174 (2002)

[浅井 04] 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, データ工学論文特集, 電子情報通信学会論文誌, Vol.J87-D-I, No.2, pp. 79-96 (2004)

[BJ 98] R. J. Bayardo Jr.: Efficiently Mining Long Patterns from Databases, Proc. SIGMOD Conference 1998: pp. 85-93 (1998)

[EM 02] T. Eiter, K. Makino: On Computing all Abductive Explanations, Proc. AAAI-2002, pp.62 (2002)

[FIMI HP] B. Goethals, the FIMI repository, http://fimi.cs.helsinki.fi/ (2003)

[HPY 00] J. Han, J. Pei, Y. Yin: Mining Frequent Patterns without Candidate Generation. Proc. SIGMOD Conference 2000, pp. 1-12 (2000)

[IWM 03] A. Inokuchi, T. Washio, H. Motoda, Complete Mining of Frequent Patterns from Graphs: Mining Graph Data, Machine Learning, 50(3), pp. 321-354 (2003)

[KMM 04] T. Kudo, E. Maeda, Y. Matsumoto: An Application of Boosting to Graph Classification, Proc. NIPS 2004 (2004)

[MA 07] S. Minato, H. Arimura: Frequent Pattern Mining and Knowledge Indexing Based on Zero-suppressed BDDs, Post-Proc. of the 5th International Workshop on Knowledge Discovery in Inductive Databases, LNAI, (2007)

[Mo 98] S. Morishita, On Classification and Regression, Proc. Discovery Science (DS'98), pp. 40-57 (1998)

[MN 99] S. Morishita, A. Nakaya: Parallel Branch-and-Bound Graph Search for Correlated Association Rules, Large-Scale Parallel Data Mining: pp. 127-144 (1999); S. Morishita: On Classification and Regression, Proc. Discovery Science 1998, LNAI 1532, pp. 40-57 (1998)

[MS 00] S. Morishita, J. Sese: Traversing Itemset Lattice with Statistical Metric Pruning, Proc. PODS 2000, pp. 226-236 (2000)

[MTV 94] H. Mannila, H. Toivonen, A. I. Verkamo: Efficient Algorithms for Discovering Association Rules, Proc. KDD Workshop 1994: pp. 181-192 (1994)

[Nakano 02] S. Nakano, Efficient generation of plane trees, Inf. Process. Lett., 84(3), pp. 167-172 (2002)

[PHPCDH 01] J. Pei, J. Han, B. Pinto, Q. Chen, U. Dayal, M. Hsu: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth, Proc. IEEE ICDE 2001, pp. 215-224 (2001)

[MS 00] S. Morishita, J. Sese, Traversing Itemset Lattice with Statistical Metric Pruning, Proc. PODS 2000, pp. 226-236 (2000)

[西田 07] 西田豊明(編), レクチャーシリーズ「知能コンピューティングとその周辺」, 人工知能学会誌, 2007 年 3 月号～ (2007)

[SAA 00] S. Shimozono, H. Arimura, S. Arikawa, Efficient Discovery of Optimal Word-Association Patterns in Large Text Databases, New Generation Computing, 18(1), pp. 49-60 (2000)

[TK 06] K. Tsuda, T. Kudo: Clustering graphs by weighted substructure mining, Proc. ICML 2006, pp. 953-960 (2006)

[宇野 HP] 宇野 毅明のホームページ, http://research.nii.ac.jp/~uno/index-j.html

[UKA 04] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets, Proc. FIMI'04 (2004)

[UAUA 04] T. Uno, T. Asai, Y. Uchida, H. Arimura: An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases, Proc. Discovery Science 2004, LNAI 3245, pp. 16-31 (2004)

[宇野 07] 宇野毅明, 有村博紀: データインテンシブコンピューティング その 2 頻出アイテム集合発見アルゴリズム, レクチャーシリーズ「知能コンピューティングとその周辺」, (編)西田豊明, 第 2 回, 人工知能学会誌, 2007 年 5 月号 (2007)

[WM 03] T. Washio, H. Motoda: State of the art of graph-based data mining. SIGKDD Explorations, 5(1): pp. 59-68 (2003)

[鷲尾 07] 鷲尾隆: データインテンシブコンピューティング その1 離散構造マイニング, レクチャーシリーズ「知能コンピューティングとその周辺」, (編)西田豊明, 第 1 回, 人工知能学会誌, 2007 年 3 月号 (2007)

[YH 02] X. Yan, J. Han, gSpan: Graph-Based Substructure Pattern Mining, Proc. IEEE ICDM 2002, pp. 721-724 (2002)

[Zaki 02] M. J. Zaki, Efficiently mining frequent trees in a forest, Proc. ACM KDD'00, pp. 71-80 (2002)

[ZPOL 97] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li: New Algorithms for Fast Discovery of Association Rules, Proc. KDD 1997, pp. 283-286 (1997)