探索位置索引による系列パターンマイニング

Sequential Pattern Mining by Search Position Indexing

北原洋一[1]　櫻井茂明[1]　植野研[1]　折原良平[1]

Youichi Kitahara[1], Shigeaki Sakurai[1], Ken Ueno[1], Ryohei Orihara[1]

[1]株式会社東芝 研究開発センター

[1] Corporate Research & Development Center, Toshiba Corporation

Abstract: This paper proposes a sequential pattern mining algorithm by search position indexing. This algorithm uses two indexes to skip a redundant sequence data scan in mining processes. In an experimental evaluation, it slightly outperforms popular sequential pattern mining algorithm SPAM and PrefixSpan.

1. はじめに

頻出系列パターンマイニングは，データマイニング分野における主要な基礎技術の一つである．データが十分蓄積されており，データ要素間に順序が定義されているとき，特徴的なパターンを発見するのに広く用いられる．例えば，テキストマイニングやDNA 配列解析，HDD キャッシュの効率化[4]等の幅広いアプリケーションで利用される．

しかしながら, 頻出系列パターンマイニングでは, 大量の系列パターンを調べなければならないため, 非常に時間がかかるという問題がある．そのため, 頻出系列パターンを効率的に発見するアルゴリズムがいくつか提案されている．

代表的な頻出系列パターンマイニングとしては, SPAM[2]と PrefixSpan[5]がある. SPAM はビットマップを用いた効率的なアルゴリズムとして知られており，LAPIN-SPAM[9]や PRISM[3]といった SPAM を改良した頻出系列パターンマイニングアルゴリズムも提案されている．

一方，PrefixSpan は，射影を用いたアルゴリズムである．SPAM とは異なり，系列データをそのままスキャンすることで探索を行うため，各種拡張が行いやすいアルゴリズムである．例えば，CloSpan[8]や BIDE[7]のような飽和系列パターンマイニングに利用されている他，特定の条件を満たすパターンを抽出するのにも利用されており，実用性が高い．しかしながら，データが疎でなくなると探索効率が低下するという問題があるため，拡張性を損なわずに効率的な処理が可能なアルゴリズムが望まれる．そこで，本稿では，PrefixSpan の問題点を検討するとともに，その問題点を解決するために，索引を用いて探索を行うアルゴリズムを提案する．

本稿は，六章で構成されている．二章では，関連研究を紹介する．三章では，問題定義および従来手法の問題点を示す．四章では，探索位置索引とそれを用いた系列パターンアルゴリズムについて説明する．五章では，実験を紹介し，六章ではまとめを行う．

2. 関連研究

初期の頻出系列パターンマイニングアルゴリズムは，頻出アイテムセットマイニングにおけるアルゴリズムを系列データに拡張することによって作られた．AprioriAll [1]は頻出アイテムセットマイニングで用いられる Apriori を系列データに拡張したものである．さらに，AprioriAll に時間制約やタクソノミー等を導入して改良したのが GSP [6]である．

AprioriAll や GSP の特徴は，幅優先探索を用いているところにある．発見済みの頻出系列パターンを全て記憶しておき，それらを組み合わせることで次に探索する候補系列パターンを生成する. そのため, データ量が多くなると，主記憶を大量に消費するという問題があった．また，候補系列パターンが長くなればなるほど，パターンマッチング時間が長くなるため，サイズの大きい頻出系列パターンを扱うと非常に時間がかかるという問題もあった．

SPADE[11]は，この問題を解決すべく，深さ優先探索と頻度計数に適したデータフォーマットを導入して効率化を図ったアルゴリズムである．SPADE では, 系列データ集合をパターンごとにまとめた ID リスト同士を結合させて，深さ優先探索で，よりサイ

ズの大きい系列パターンの ID リストを生成する処理を繰り返す. 頻度は ID リストに含まれる系列データの数と等しい.

SPADE は, GSP で生じていた主記憶大量消費と長大な頻出系列パターン探索時間の問題を双方とも解決している．深さ優先探索を用いているため，主記憶に保持しておく必要のある系列パターンは，系列パターンの探索木における，親に対応する系列パターンのみとなる．そのため，GSP ほど大量の主記憶を消費することはない．また，ID リストに含まれるアイテム出現位置の前後関係を用いて結合不可能と判断された系列データは ID リストから削除される．そのため, サイズの大きい系列パターンの ID リストほどサイズが小さくなるから，頻度計数も高速になる．しかしながら，SPADE には改善の余地が残されており，ID リストの結合処理に時間がかかるという問題を有していた.

SPAM は，SPADE における結合処理時間の短縮のため，ID リストをビットマップ表現にした手法である. 基本的なアルゴリズムは SPADE と同一であるが, ID リストの結合処理を，ビット操作のみで実行できるため，高速な処理が可能である．SPAM は，もっとも高速な系列パターンマイニングアルゴリズムの一つとして知られている.

ID リストを使わないアプローチとしては，射影を用いるアルゴリズムがある．ここでの射影とは，系列データ集合から，射影対象アイテムもしくはそれを含む系列データを抽出する処理を指す．FreeSpan は，系列パターンマイニングに初めて射影の概念を導入したアルゴリズムであるが，あまり効率的なアルゴリズムではなかった．PrefixSpan は，末端アイテムを除いた頻出系列パターンである prefix への射影を繰り返すことで，深さ優先探索を実現したアルゴリズムである．射影によって，探索対象となる系列データ集合が徐々に縮小されるため，効率的な探索が可能になっている．PrefixSpan も，もっとも高速な系列パターンマイニングアルゴリズムの一つとして知られており，特に疎なデータを扱う場合に効率的である．なお，PrefixSpan には，実際に射影を行い，データ集合を逐次抽出する方法と，射影を行ったかのように探索範囲を縮小していく擬似射影と呼ばれる方法があるが，後者の方がパフォーマンスに優れているため，以下の議論では全て擬似射影を前提として話を進める.

最近提案されたアルゴリズムとしては，LAPIN や PRISM が挙げられる．双方とも SPADE をベースとしているが，LAPIN[10]には PrefixSpan や SPADE を拡張したタイプも存在する.

LAPIN は，各アイテムが各系列データにおいて最後に出現した位置をあらかじめ記憶したテーブルを利用するアルゴリズムである．このテーブルを参照することで系列データを探索する前にアイテムがその系列データに存在しないことを察知する．そのため，無駄な探索を削減することができる.

PRISM は，ビットを用いてアイテムの出現位置を示した位置エンコーディングデータと系列データにおけるアイテムの出現有無を示した系列エンコーディングデータを，さらに変換して素数の積で表現したデータを用い，素数の積の最大公約数や最小公倍数を用いて結合処理を行うアルゴリズムである．一見，特殊なアルゴリズムに思われるが，最終的にはビットマップで処理を行っており，SPAM とほぼ同じアルゴリズムである．なお，LAPIN のテーブルからは，探索途中においても，探索対象アイテムの存在有無を知ることができるが，系列エンコーディングデータからは知ることができないから，LAPIN の方が無駄な探索を削減することが可能であると思われる.

3. 問題

3.1 問題定義

まず，頻出系列パターンマイニングの問題定義を行う.

アイテム i_i の集合を $I = \{i_1, i_2 \cdots i_k\}$ と表し，系列長 l の系列データを $s = \langle e_1, e_2 \cdots e_l \rangle$ と表す．ここで, $e \in I$ である. 任意の系列データ $s_a = \langle a_1, a_2 \cdots a_n \rangle$ と $s_b = \langle b_1, b_2 \cdots b_m \rangle$ について，$a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2} \cdots a_n \subseteq b_{j_n}$ となるような整数 $1 \le j_1 < j_2 \cdots < j_n \le m$ が存在するならば，s_a は s_b に含まれていると表現し，この関係を $s_a \subseteq s_b$ で表す.

系列 s と系列データ集合 $S = \{s_1, s_2 \cdots s_k\}$ が与えられたとき，$s \subseteq s_i$ となる系列 s_i の数を s の支持度と呼び，$\sup(s)$ で表す．また，閾値 $minsup$ を与えたとき，$\sup(s) \ge minsup$ となる系列パターン p を頻出であるといい，このときの閾値 $minsup$ を最小支持度と呼ぶ．頻出系列パターンマイニングでは，系列データ集合に含まれる全ての頻出系列パターン

を発見するのが目的である．

例として，図 1 の系列データ集合の例を用いて説明する．系列データには各々ID が割り当てられており，アイテムをアルファベットで示している．また，”(”と”)”で囲まれた複数のアイテムはアイテムセットであり，同一順位であることを示している．例えば，ID が 0001 の系列データ一番目のアイテムセットに含まれる b と d と e と f は同一順位である．一方，ID が 0003 の一番目のアイテムセットに含まれる a と b は，二番目のアイテムセットに含まれる a とは別の順位であることを示している．

最小支持度として 2 を指定したときについて考える．例えば，系列パターン<(bf)(a)>は，ID が 0001 と 0002 の二つの系列データに存在しているので，支持度は 2 となり頻出であると判定される．一方，系列パターン<(a)(a)(c)>は ID が 0003 の系列データのみにしか含まれていないので，支持度は 1 となり頻出ではない．

ID	系列データ
0001	(bdef)(acdf)(acdfgh)
0002	(abcf)(abd)(abe)
0003	(ab)(a)(c)(bd)(b)(bc)

図 1　系列データ集合の例

3.2 射影型アルゴリズムの問題点

本節では PrefixSpan アルゴリズムの特性および問題点について考察する．

PrefixSpan では，射影系列データ集合をスキャンし頻出アイテムを発見する，頻出アイテムについての射影系列データ集合を生成する，という二つのステップを再帰的に繰り返すアルゴリズムである．擬似射影においては，射影系列データ集合の生成は，各系列データをスキャンし，射影系列データの開始位置を特定することと等しい．

系列パターンマイニングにおいて，処理時間のほとんどは系列データスキャンで占められているので，系列データスキャンに着目して，上記二つのステップについて考える．系列データの平均長を l とすると，頻出アイテムを発見するためには系列データの末端までスキャンを行う必要があるので，マイニング開始直後には系列データ全体をスキャンすることになり，一系列データあたり $O(l)$ の時間がかかる．また，射影系列データ生成時には，頻出アイテムが出現している位置までスキャンする必要があるので，平均 $l/2$ の時間がかかる．したがって，一系列データあたりのサイズが増加するにつれて，一系列データあたりのスキャン時間は l に比例して増加する．

PrefixSpan が効率的に働くためには，一系列データスキャンで，多数の頻出アイテムが発見されることが望ましい．つまり，系列データサイズと比較して，アイテムの種類数が多いデータにおいて効率的である．

例えば，発見されるアイテムの多くが頻出であるとき，<(ab)(cdef)(ghi)>のような系列データからは効率的に頻出アイテムを発見できる．一方，<(ab)(ab)(ab)(ab)(ab)>のような，出現アイテムに重複の多い系列データに対しては効率的ではない．前者では，発見される頻出アイテムが l に比例するので，スキャン時間が l に比例して増加しても，一頻出系列パターンあたりの処理時間はほぼ一定であるが，後者では一定とはならず l に比例して非効率になる．もし，後者のようなデータでは，不要な部分のスキャンを省略するような仕組みがあれば，効率性を保つことができる．このような仕組みを，索引を用いて実現しようと試みたのが，本稿で提案するアルゴリズムである．

4. 探索位置索引

本章では，探索位置索引について説明する．

探索位置索引とは，擬似射影を行う際に，スキャンする必要のある位置をあらかじめ記憶したデータである．索引は系列データごとに作成される．

探索位置索引には，射影対象となるアイテムの種別に対応し，二種類の索引を用いる．射影対象となるアイテムが，射影元となるアイテムと同一アイテムセットに存在しない場合に用いる索引を系列索引と呼び，同一アイテムセットに存在する場合に用いる索引をアイテムセット索引と呼ぶ．これは，それぞれ，SPAM アルゴリズムの S-Step と I-Step に対応している．例えば，系列パターン<a>に射影した後，<ab>に射影する場合は系列索引を使い，<(ab)>に射影する場合はアイテムセット索引を用いる．以下では，双方の索引および索引を利用したアルゴリズムについて説明する．なお，索引を利用して系列データスキャンを行っている点を除けば，基本的なアルゴリズムは PrefixSpan と同様であるため，マイニングアルゴリズムの説明は省略する．

4.1 系列索引

系列索引は，各アイテムから系列データの末端までに存在する異なるアイテムの位置を，アイテムセットごとに記憶したデータである．頻出アイテムを探索する際には，射影系列データの先頭アイテムのアイテムセットに付与されている系列索引を参照することで，探索すべきアイテムの位置を取得することができる．

例として，系列データ<(abce)(bcd)(abd)(abcd)(cde)>の系列索引を図 2 に示す．アイテムの上に付与されている数字はアイテムの位置を表している．また，系列データの下に付与されている縦に並べられた数字の列は系列索引である．数字はアイテムの位置を示している．なお，最後のアイテムセットには索引は不要である．また，最後から二番目のアイテムセットに属しているアイテムからスキャンする場合，最後のアイテムセット全てをスキャンする必要があるため，索引は不要である．

位置 0 のアイテム a からスキャンをはじめる場合について考える．まずスキャン開始位置に対応する系列索引を参照する．ここでは，一番目の系列索引を参照し，位置 7，4，5，6，16 を取得する．次に，その位置に対応したアイテムを取得する．ここでは，それぞれ，a，b，c，d，e を得ることができる．

PrefixSpan の一系列データあたりのスキャン時間は射影系列データの長さ l_s に依存するが，索引を用いた場合は射影系列データに含まれるアイテムの種類数 N_s に依存する．索引を用いた場合，一アイテムを取得するのに索引参照とアイテム取得の二ステップを要するから，単純にステップ数のみを考えるならば，$l_s > 2N_s$ であれば索引を用いた方が効率的になる．しかし，一般に，$l_s \geq N_s$ であるから，索引を用いると最悪で PrefixSpan の二倍処理量が必要になる．

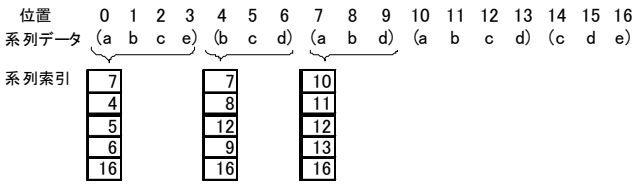

図 2　系列索引の例

系列索引は，図 3 のシンプルなアルゴリズムを用いて生成することができる．D は系列データ集合，s は系列データ，I_s は系列索引，M は位置行列である．

位置行列は，全てのアイテムを行，系列データの二番目以降のアイテムセットを列に対応させたデータである．なお，系列索引では，主にアイテムセット単位で処理が行われており，itemsetPos は現在処理されているアイテムセットを示すために使われる．

次に，図 2 の系列データについての系列索引生成処理を例に用いて，処理の流れについて説明する．図 4 は，図 2 の系列データの系列索引生成時に作成される位置行列である．

まず，initilizePositinMatrix()関数を用いて，位置行列を初期化し，skipItemset()関数を用いて，一つ分のアイテムセットを読み飛ばす．一番目のアイテムセットは索引生成には利用されない．

次に，二番目以降のアイテムセットをスキャンして，位置行列に位置を記憶させていく．

getItemPosition() 関数は，系列データ s から itemsetPos で指定されたアイテムセットに含まれる全アイテムの位置を取得して，位置行列のアイテムに対応する行の itemsetPos で指定される列に記憶させる．例えば，二番目のアイテムセットには，アイテム b, c, d が存在するので，それぞれの位置 4, 5, 6 を位置行列に記憶させると，図 4 の(a)が生成される．同様に，三番目のアイテムセットの位置も記憶すると，図 4 の(b)が生成される．

fillRow() 関数は，位置行列において，直前の getItemPosition()で新たに記憶されたアイテムの行を左方向に調べたとき，位置が記憶されていないところに新たに記憶された位置を記憶させる．例えば，三番目のアイテムセットの場合，アイテム a に対応する行の左の列には位置が記憶されていないので，位置を記憶させると図 4 の(c)が生成される．

最終アイテムセットまで getItemPosition() と fillRow()を繰り返すと，最終的に図 4 の(g)が生成される．最後に，writeToSIndex()関数を用いて，位置行列の列を左から順に，系列データのアイテムセットに対応付けることで，系列索引が生成される．

一系列データあたりの系列索引の生成に要する処理量は，アイテムの種類数 N と平均アイテムセット数 I に依存し，$O(NI)$ となる．処理量は，最小支持度に依存しないため，系列索引生成処理時間が問題となることはほとんどない．

```
function createSIndex(D, I_s) {
  foreach s (D) {
    initializePositionMatirx(M);
    skipItemset(s);
    itemsetPos = 1;
    do {
      getItemPosition(s, itemsetPos, M);
      fillRow(M);
      itemsetPos++;
    } while (itemsetPos != last itmset position);
  }
  writeToSIndex(I_s, M);
}
```

図 3　系列索引生成アルゴリズム

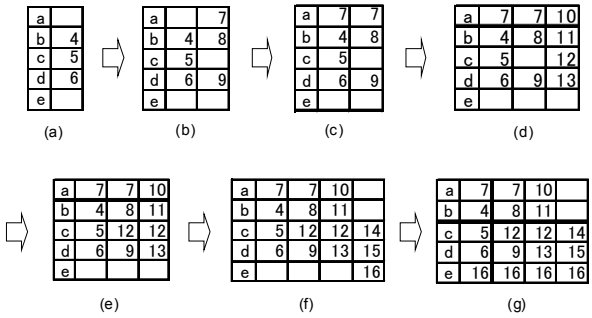

図 4　系列索引の位置行列の例

4.2 アイテムセット索引

アイテムセット索引は，各アイテムから系列データの末端までに存在する同一アイテムが次に存在する位置を，アイテムごとに記憶したデータである．ただし，同一アイテムが次に存在する位置のアイテムセット内に，重複したアイテムしか存在しない場合は，索引には記憶されない．頻出アイテムを探索する際には，アイテムに付与されている索引の参照を繰り返すことで，探索開始位置を取得することができる．

例として，系列索引と同様に，系列データ<(abce)(bcd)(abd)(abcd)(cde)>のアイテムセット索引を図 5 に示す．アイテムの上に付与されている数字はアイテムの位置を表している．また，系列データの下に付与されている数字はアイテムセット索引であり，アイテムの位置を示している．なお，最後のアイテムセットには索引は不要である．

位置 1 のアイテム b からスキャンをはじめる場合について考える．まず，スキャン開始位置に対応するアイテムセット索引を参照して次に探索を開始するアイテムの位置を取得する処理を，索引が存在しなくなるまで続ける．次に，取得した探索開始位置からアイテムセット内のスキャンを行って頻出アイテムを発見する．位置 1 のアイテムセット索引を参照すると，位置 4 が記憶されている．さらに，位置 4 の索引を参照すると，空になっているので索引を辿る処理は終了する．次に，探索開始位置 1，4 を含むアイテムセットをスキャンすると，アイテム c, d, e を得ることができる．位置 8，11 にもアイテム b は存在するが，これらのアイテムセットには重複したアイテムしか存在しないので，索引には位置が記憶されていない．このように，アイテムセット内に存在するアイテムの種類に偏りがある場合，索引を用いることで探索開始アイテムを含んでいない，もしくは，重複アイテムしか存在しないアイテムセットのスキャンを省略することができる．

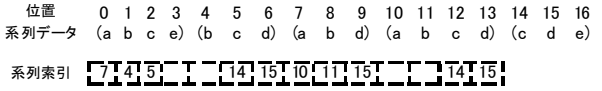

図 5　アイテムセット索引の例

アイテムセット索引は，図 6 のアルゴリズムを用いて生成することができる．I_I はアイテムセット索引である．アイテムセット索引でも，主にアイテムセット単位で系列データのスキャンが行われる．

次に，図 5 の系列データについてのアイテムセット索引生成処理を例に用いて，処理の流れについて説明する．図 7 は，図 5 の系列データのアイテムセット索引生成時に作成される位置行列である．

まず，initilizePositinMatrix()関数を用いて，位置行列を初期化し，getItemPosition()関数を用いて，一つ分のアイテムセットをスキャン，位置行列にアイテムの位置を記憶させる．一番目のアイテムセットを読み込むと，位置行列は図 7 の(a)のようになり，二番目のアイテムセットを読み込むと，図 7 の(b)のようになる．

次に，二番目以降のアイテムセットをスキャンして位置行列に位置を記憶させていくとともに，アイテムセット索引を生成していく．

writePositionToIIndex()関数は，位置行列をチェックして，アイテムセット索引に位置を記憶させる関数である．ここでは，位置行列において，直前の getItemPosition()で新たに記憶された列をカレント列と呼ぶことにする．まず，最終行のアイテムからカレント列を探索し，直前の getItemPosition()で新たに記憶されたアイテムの行を左方向に調べたとき，位

置が記憶されていないところを探す．ここでは，位置が記憶されていない箇所が存在する列を出力列と呼ぶ. 次に, 発見箇所から上方向に出力列を探索し，出力列にも，カレント列にも位置が記憶されている行を見つける．そして，その行について，アイテムセット索引の，出力列に記憶されている位置に，カレント列に記憶されている値(位置)を記憶させる．なお，索引に記憶された位置には，記憶済みのマークを付けておく．

例えば，二番目のアイテムセットを読み込んだ直後では，アイテム d の左（一列目）に位置が記憶されていない箇所が見つかる．見つかった列を上方向に探索すると，アイテム b と c では出力列にもカレント列にもアイテムが存在するので，アイテムセット索引の位置 1 に 4 を記憶させ，位置 2 に 5 を記憶させる．図 7 の(b)では，記憶された値の箇所を丸印で示し，記憶済みマークが付けられた箇所を斜線で示している．なお，最終行については，無条件に記憶済みマークが付けられる．

最終アイテムセットまで上記の処理を繰り返すと，位置行列は最終的に図 7 の(e)のようになっているとともに，アイテムセット索引が生成されている．

一系列データあたりのアイテムセット索引の生成に要する処理量は, アイテムの種類数 N と平均アイテムセット数 I に依存し，$O(NI)$ となる．系列索引と同様に, 処理量は, 最小支持度に依存しないため，索引生成処理時間が問題となることはほとんどない．しかし，writePositionToIIndex()は比較的処理量の多い関数であるため, アイテムセット索引の生成には，系列索引の生成よりも時間がかかる．

```
function createIIndex(D, I1) {
  foreach s (D) {
    initializePositionMatirx(M);
    itemsetPos = 0;
    getItemPosition(s, itemsetPos, M);
    itemsetPos++;
    do {
      getItemPosition(s, itemsetPos, M);
      writePositionToIIndex(I1, M);
      itemsetPos++;
    } while (itemsetPos != last itmset position);
  }
}
```

図 6　アイテムセット索引生成アルゴリズム

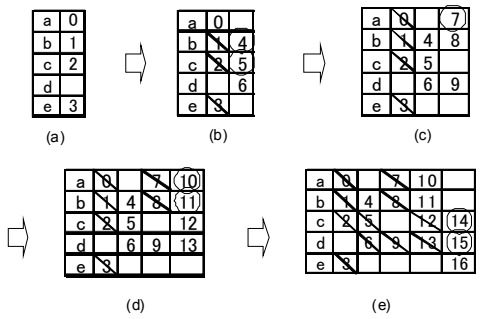

図 7　アイテムセット索引の位置行列の例

5. 実験

5.1 実験環境

本節では，提案アルゴリズムの特性を調べるために行った検証実験の実験環境について説明する．

実験では，IBM が公開しているデータジェネレータ(http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html)によって生成された人工データを用いた．データ生成におけるパラメータは**表 1** に示したものを用いた．D と N は単位が 1000 である．実験は，Intel Xeon 3.60GHz の CPU，2.75GB RAM のマシンにて実行された．

表 1　人工データ生成パラメータ

シンボル	意味
D	系列データ数
C	一系列データあたりの平均アイテムセット数
T	一アイテムセットあたりの平均アイテム数
S	最大系列データの平均長
I	最大系列データの平均アイテムセット長
N	アイテムの種類数

比較対象のアルゴリズムには，代表的な系列パターンマイニングアルゴリズムである SPAM と PrefixSpan を用いた．双方とも，作者のホームページにて公開されており，SPAM は「http://himalaya-tools.sourceforge.net/Spam」から，PrefixSpan は Illimine(http://illimine.cs.uiuc.edu)として配布されているものから入手した．

計測されたのは，最小支持度を変化させたときの実行時間と，データ生成パラメータを変化させて生成されたデータに関して同一最小支持度を指定したときの実行時間である．前者については，D1C20T20S20I20 を固定し，アイテムの種類数 N を調整して, PrefixSpan が有効なケース(N=0.8)と SPAM

が有効なケース(N=0.5)について調べた．後者については，D1C20T20S20I20N0.8 を基準として最小支持度を 0.2 に固定し，C と T と N を変化させたケースについて調べた．

5.2 実験結果

本節では，実験結果について報告し，その実験結果から推測される提案アルゴリズムの特性について考察する．

図 8 と図 9 は，それぞれ PrefixSpan が有効なケースと SPAM が有効なケースにおけるマイニング実行時間のグラフである．図 8 では，最小支持度を小さくするにしたがって SPAM の実行時間が大きく増加しているにもかかわらず，PrefixSpan と提案手法では増加が比較的抑制されていることがわかる．提案手法は，基本的に PrefixSpan と同じマイニング特性を有するため，PrefixSpan が有効な状況では提案手法も有効に働くと思われる．図 9 では，逆に PrefixSpan の実行時間が大きく増加しているものの，SPAM と提案手法では抑制されている．このことから，提案手法は，PrefixSpan の問題点を解決しているため，PrefixSpan が有効に機能しないケースでも大きな性能劣化は生じないことが期待される．

図 10 と図 11，図 12 は，データ生成パラメータを変化させたときのマイニング実行時間のグラフである．図 10 および図 11 からわかるように，平均アイテムセット数の増加や平均アイテム数の増加によって一系列データあたりのデータサイズが増加すると，PrefixSpan では大きく性能劣化が生じるが，SPAM と提案手法では大きな性能劣化は生じない．また，図 12 より，アイテム種類数が減少した場合でも，PrefixSpan の実行時間は大きく増加するが，SPAM と提案手法では抑制されている．PrefixSpan の性能劣化の原因のひとつには，3.2 節での指摘内容が考えられ，提案手法ではその問題を解決しているため PrefixSpan のような性能劣化を抑制できていると思われる．

全体的に，提案手法のパフォーマンスは PrefixSpan や SPAM よりわずかではあるものの高い傾向にあることがわかる．提案手法では，マイニング処理の前に索引生成処理を行っているが，データサイズが小さいこともあり，系列索引，アイテムセット索引とも，生成処理時間が 0.1 秒を超えることはなく，実行時間の増加にはほとんど寄与していなかった．また，アイテムセット索引生成処理時間は，系列索引生成処理時間の 1.5 倍程度の時間がかかることもわかった．

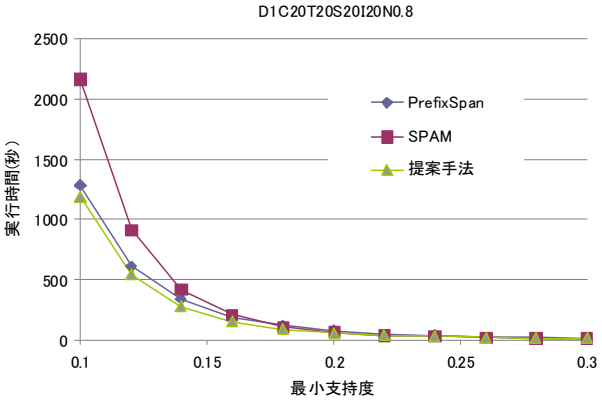

図 8　PrefixSpan が有効なデータでの実行時間

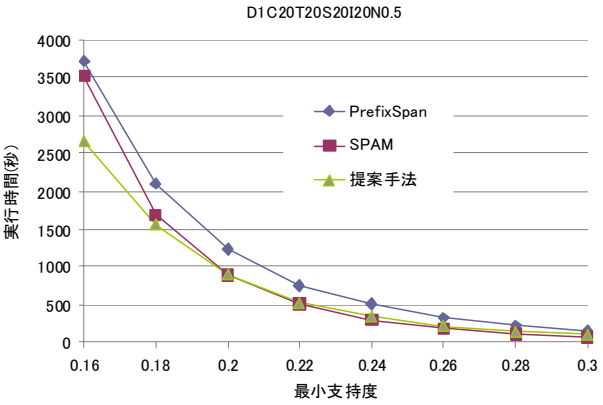

図 9 SPAM が有効なデータでの実行時間

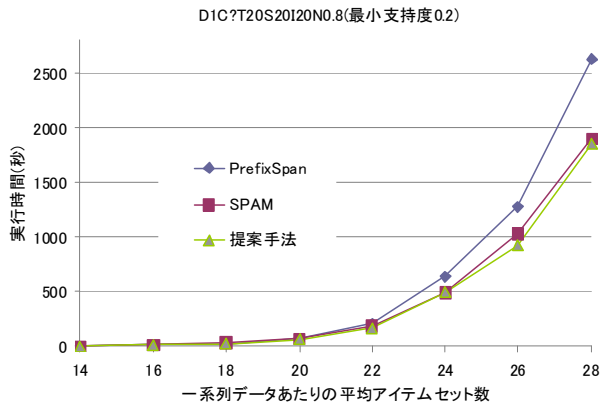

図 10　一系列データあたりの平均アイテムセット数を変化させたときの実行時間

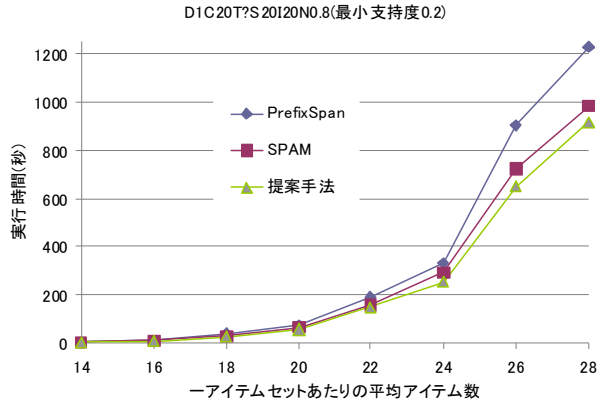

図 11 一アイテムセットあたりの平均アイテム数を変化させたときの実行時間

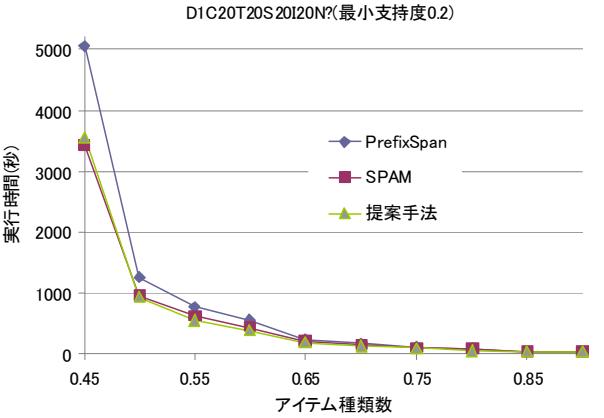

図 12 アイテム種類数を変化させたときの実行時間

6. まとめ

本稿では，PrefixSpan アルゴリズムの特性を考察し，その問題点を解決するため，探索位置索引を用いた系列パターンマイニングアルゴリズムを提案した．検証実験の結果，PrefixSpan では性能劣化が生じるケースにおいても提案手法が有効であることが確認され，代表的な系列パターンマイニングアルゴリズムである SPAM よりわずかに優れたパフォーマンスが得られた．

現在の実装には非効率な部分があり，また，索引サイズ最適化等のさらなる高速化が期待される機能も未実装である．今後は，これらの改善および実装を行う予定である．

参考文献

[1] R. Agrawal and R. Srikant: Mining Sequential Patterns, ICDE1995, pp.3-14 ,(1995)

[2] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick: Sequential PAttern Mining using a Bitmap Representation, KDD2002, pp.429-435 , (2002)

[3] K. Gouda, M. Hassaan and M. Zaki: PRISM: A Prime-Encoding Approach for Frequent Sequence Mining, ICDE2007, (2007)

[4] Z. Li, Z. Chen, S. Srinivasan and Y. Zhou: C-Miner: Mining Block Correlations in Storage Systems, FAST2004, pp.173-186, (2004)

[5] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, IEEE Trans. Knowl. Data Eng. Vol.16, No.11, pp1424-1440, (2004)

[6] R. Srikant, and R. Agrawal: Mining sequential patterns: Generalizations and performance improvements, EDBT1996, pp.3-17, (1996)

[7] J. Wang, J. Han, and C. Li: Frequent Closed Sequence Mining without Candidate Maintenance, IEEE Trans. Knowl. Data Eng., Vol. 19, No. 8, pp. 1042-1056, (2007)

[8] X. Yan, J. Han, and R. Afshar: CloSpan: Mining Closed Sequential Patterns in Large Databases, SDM2003, (2003)

[9] Z. Yang and M. Kitsuregawa: LAPIN-SPAM: An Improved Algorithm for Mining Sequential Pattern, ICDEW2005, pp. 1222, (2005)

[10] Z. Yang, Z. Wang and M. Kitsuregawa: Effective Algorithms for Sequential Pattern Mining, 日本データベース学会 Letters, Vol. 5, No. 1, pp.53-56, (2006)

[11] M. Zaki: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning, Vol.42, pp.31-60 , (2001)