社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

頻出部分文字列のマイニング

坪井　祐太†

† 日本アイ・ビー・エム (株) 東京基礎研究所
〒 242-8502 神奈川県大和市下鶴間 1623 番 14 号
E-mail: †yutat@jp.ibm.com

あらまし　可変長 N グラムのカウントを頻出パターンマイニング問題の文脈で捉え直し、ある閾値以上出現するすべての部分文字列を列挙する問題として定義する。また、この問題を高速かつ少ないメモリで解くアルゴリズムを提案する。計算機実験により、接尾辞木を使用する方法に比べて高速であることが確かめられた。
キーワード　頻出部分文字列, 可変長 N グラムモデル, 頻出パターンマイニング, 部分構造マイニング, 3 分割法

Mining Frequent Substrings

Yuta TSUBOI†

† Tokyo Research Laboratory, IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan
E-mail: †yutat@jp.ibm.com

Abstract　This work defines the fast counting of variable-length N-grams as the problem of enumerating all substrings appearing more frequently than some threshold in a given string. This paper introduces a novel mining algorithm that is faster and requires less working memory than existing algorithms. A trial performance study shows the proposed algorithm runs faster than the pattern enumeration using a suffix tree.
Key words　Frequent Substrings, variable-length N-gram model, Frequent Pattern Mining, Substructure Mining, Ternary Partitioning

1. 導　　入

言語現象の生起をマルコフ過程と考え N 個の文字や単語の列の分布によって言語を特徴づける N グラムモデルは、統計的自然言語処理の分野で広く使われている言語モデルである。

代表的な N グラムモデルの例として、Shannon game と呼ばれる途中までの自然言語文から次の単語 w_t を予測する問題がある。一般的な N グラムモデルでは、w_t が直前の $N-1$ 単語に依存すると仮定し、条件付き確率 $P(w_t|w_{t-1},\cdots,w_{t-N-1})$ が大きい単語を予測することになる。観測されたデータから最尤推定により条件付き確率を次のように求めることができる。

$$P(w_t|w_{t-1},\cdots,w_{t-N-1}) = \frac{C(w_t,w_{t-1},\cdots,w_{t-N-1})}{C(w_{t-1},\cdots,w_{t-N-1})}$$

ただし、$C(w_i,\cdots,w_j)$ は単語列 $w_i,\cdots,w_j$ の観測データ中の出現頻度である。

N グラムモデルにおいて、N は 2 や 3 などの比較的小さい値に固定されることが多いが、可変長 N グラムモデルと呼ばれる履歴によって N の値を動的に変えるモデルも研究されている。Ron [21] らは、N グラムの長さを伸ばすことで確率分布が変わる場合のみ、より長い N グラムをモデルに加える手法を提案している。有効な長さの N グラムを見付けるためにはすべての長さ N グラムを評価する必要があるが、組合せの問題から現実的ではないため、実際には出現頻度の低い N グラムを評価対象から取り除く [12], [21]。また N を固定した場合でも、出現頻度の低い N グラムを使って統計的に信頼性のある値を推定することが難しいので、実際の応用では低頻度の N グラムは無視されることが多い。

そこで、現実には低頻度の N グラムは考慮しないことが多いことをふまえ、本研究では N の長さは固定せずある閾値より多い N グラムのみを高速に列挙しその頻度をカウントする方法を提案する。最初に 2. 章で問題を一般化し、頻出部分文字列マイニング問題として定義する。また、頻出パターンマイニング問題との関連についても触れる。続く 3. 章および 4. 章でこの問題を分割統治法によって解くアルゴリズムを提案する。提案アルゴリズムは、高速かつ少ないメモリ使用量で頻出する部分文字列を列挙することができる。5. 章では、先行研究である N-gram PrefixSpan を紹介し、提案アルゴリズムとの類似点を示す。6. 章では、接尾辞木と呼ばれるデータ構造を使って問題

を解く方法を説明する。7. 章では計算機実験により、理論的には計算オーダが低い接尾辞木を使った方法より提案アルゴリズムのほうが性能が良い事を示す。

2. 頻出部分文字列マイニング問題

2.1 問題定義

ここでは、論文中の用語と頻出部分文字列マイニング問題を定義する。

用語定義: 文字 (character) を頻度を数える最小単位とし、文字列 (string) とは文字を要素とする順序付きリストとする。長さ n の文字列 s の部分文字列とは、s の i 文字目から j 文字目 ($0 \leqq i \leqq j < n$) までの連続する要素列である。ただし、s の 0 文字目は s の先頭の文字を示すものとする。また、部分文字列 $s[i..j]$ が部分文字列 $s[k..l]$ の上位文字列であるとは、$i \leqq k \leqq l < j$ または $i < k \leqq l \leqq j$ の時(注1)を示す。

ここでの文字及び文字列は一般化された概念であり、応用として単語 N グラムを考える時は、文字は実際には単語であり文字列は単語列である。

問題定義: 文字列 s に部分文字列 ($s[i..j]$) が閾値以上出現するとき、つまり $C(s[i..j]) \geqq \xi$ の時、$s[i..j]$ を頻出部分文字列と呼ぶ。頻出部分文字列マイニング問題とは、文字列 s 内の全ての頻出部分文字列を列挙する問題である。

例をあげる。文字列 $s =$ *sakurasaku\$*で、*kuras*（$s[2..6]$）は部分文字列であり、部分文字列 *kurasaku*（$s[2..9]$）は文字列 s の接尾辞 (suffix_2) である。また、接尾辞 *kurasaku* は部分文字列 *kuras* の上位文字列である。閾値 $\xi = 2$ の時、s の全ての頻出部分文字列は *a*, *ak*, *aku*, *k*, *ku*, *s*, *sa*, *sak*, *saku*, *u* である。

論文内で使用する各記号を以下にまとめる。

- Σ: 文字の集合
- $|\Sigma|$: 文字の集合のサイズ
- \$: 文字列の終端記号（$\$ \notin \Sigma$）
- n: 文字列の長さ（ただし終端記号を含む）
- $s[i..j]$: s の i 文字目から j 文字目までの部分文字列
- suffix_i: s の i 文字目からの接尾辞（$= s[i..n-1]$）
- $C(s[i..j])$: 部分文字列 $s[i..j]$ が s に出現する回数
- ξ: 閾値（正値）

2.2 頻出パターンマイニング問題

2.1 節で定義した頻出部分文字列マイニング問題は、データマイニング分野の中で中心的に扱われている問題の一つである頻出パターンマイニング問題の一種であると考えられる。頻出パターンマイニング問題は、集合や構造を各データとして持った巨大なデータベースから、ある閾値以上頻出する部分集合や部分構造をパターンとして高速に列挙する問題である。元々は相関ルール発見などに応用がある頻出部分集合の列挙問題 [2] が扱われていたが、近年では頻出部分構造マイニングの研究も盛んで、部分系列 [3], [20]、部分木 [1], [27]、部分グラフのマイニングアルゴリズム [10], [26] などが提案されている。これら多くの頻出パターンマイニングアルゴリズムは、あるパターンが頻出でないとき、そのパターンを含む上位のパターンもまた頻出ではないという事実を使うことで効率的に列挙をおこなう。次章で提案するアルゴリズムもこの手法を基礎としている。

本研究で定義した頻出する部分文字列を列挙する問題に類似した問題として系列パターン問題 (sequential pattern mining problem) がある [3]。系列パターン問題はある閾値以上出現する非連続な場合も含む順序が保たれた要素列を見付ける問題である。ただし、系列パターン問題では要素が連続している場合としていない場合を区別しない。例えば、*saku*rasaku\$と*shi*b*a*r*aku*\$という文字列どちらも *saku* というパターンを含んでいると考える。そのため、系列パターン問題では連続した要素列 (部分文字列) だけをマイニングしたい場合には適さない。

3. アルゴリズムの全体像

ここでは、2.1 節 で定義した頻出部分文字列マイニング問題を効率的に解くアルゴリズムを提案する。本章ではアルゴリズムの全体像を示し、次章ではアルゴリズムの効率的な実装について述べる。

本アルゴリズムは分割統治法に基づき、3 分割法 [7] を使いマイニング問題を部分問題に分割しながら再帰的に頻出部分文字列を探す。さらに、部分文字列が頻出でないときその上位文字列も頻出ではないという事実に基づき探索空間の枝刈りを行う。

$\boldsymbol{S}$ を文字列 s の全ての接尾辞の配列とする（$\boldsymbol{S} = [\text{suffix}_0, \cdots, \text{suffix}_{n-1}] = [s[0..n-1], s[1..n-1], \cdots, s[n-2..n-1], s[n-1]]$）。アルゴリズムは、分割の基準となる値 v と接尾辞の d 番目の文字とを比較することで、$\boldsymbol{S}$ を 3 つの配列 $\boldsymbol{S}_=, \boldsymbol{S}_<, \boldsymbol{S}_>$ に分割する。$\boldsymbol{S}_=$ は d 番目の文字が v と等しい接尾辞だけを含む配列、$\boldsymbol{S}_<$ は d 番目の文字が v より辞書順で小さい接尾辞だけの配列、そして $\boldsymbol{S}_>$ は d 番目の文字が v より辞書順で大きい接尾辞だけの配列を示す。$\boldsymbol{S}_=$ のサイズを $n_=$ と書く ($n_<$、$n_>$ も同様)。この分割手順を $\boldsymbol{S}_=$ では接尾辞の $d+1$ 番目の文字を基に、$\boldsymbol{S}_<$ と $\boldsymbol{S}_>$ では d 番目の文字を基に繰り返す。

以下に再帰アルゴリズムの疑似プログラムを示す。ただし、再帰関数は最初に $\textbf{mine}(\boldsymbol{S}, 0)$ として呼ばれる。

```
mine(S', d)
  if n < ξ then
    return
  end
  分割するキー v を選択
  各 suffix_i[d](suffix_i ∈ S') と v を比較し、
    配列 S'_<, S'_=, S'_> に分割
    (各配列のサイズ は n_<, n_=, n_>)
  mine(S'_<, d)
  if n_= ≧ ξ and v ≠ $ then
    部分文字列 suffix_i[0..d](suffix_i ∈ S'_=) を出力
    mine(S'_=, d+1)
  end
  mine(S'_>, d)
end
```

上記の関数の引数は以下に説明する通りである。

(注1)：$s[i..j] \neq s[k..l]$

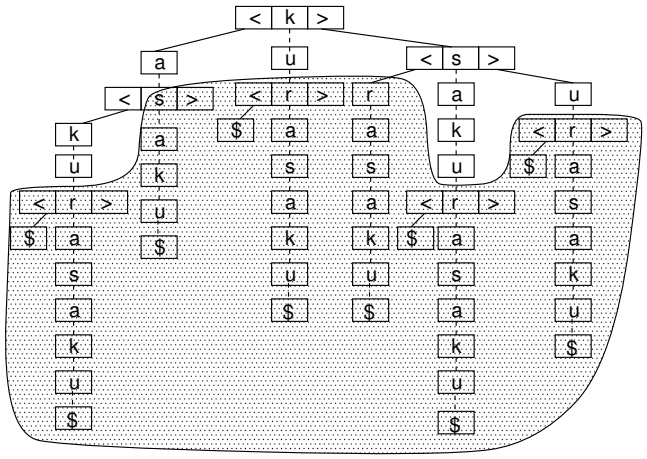


図 1　枝刈りされた 3 分探索木
Fig. 1　A pruned ternary search tree.

$\boldsymbol{S'}$　$0..d-1$ 文字目までは同一である接尾辞の配列
d　比較対象とする文字位置

接尾辞の配列の分割において重要な事実は、$\boldsymbol{S'}_{=}$ のサイズ $n_{=}$ が $\boldsymbol{S'}_{=}$ に含まれる接尾辞 $\mathrm{suffix}_i \in \boldsymbol{S'}_{=}$ の先頭から d 番目までの部分文字列 ($\mathrm{suffix}_i[0..d]$、つまり $s[i..i+d]$) の出現回数を示すことである。よって、$n_{=}$ が閾値 ξ と等しいかより大きいとき ($n_{=} \geqq \xi$)、$\mathrm{suffix}_i[0..d](\mathrm{suffix}_i \in \boldsymbol{S'}_{=})$ は頻出部分文字列である。逆に分割された配列のサイズ $n_x \in \{n_{<}, n_{=}, n_{>}\}$ が閾値より小さい場合 ($n_x \leqq \xi$)、すべての接尾辞 $\mathrm{suffix}_i \in \boldsymbol{S'}_x$ の先頭から d 番目以降の部分文字列 $(\mathrm{suffix}_i[0..d], \mathrm{suffix}_i[0..d+1], \cdots)$ はすべて頻出部分文字列ではない。この事実に従い、分割された接尾辞の配列のサイズが閾値より小さいとき、その配列に対する再帰手順を止める。ただし、$v = \$$ の時にも、文字列の末端であるため再帰を止める。

上記のアルゴリズムは終端記号 \$ で終わる複数の文字列を連結した文字列 $s_0, \cdots, s_m$ を入力とすると、各文字列の最後の終端記号を越えない頻出部分文字列のみを出力する。例えば、文字列を文とし文字を単語と考えると文を越えない頻出単語 N グラムの頻度がカウントできる。

このアルゴリズムの挙動は、全ての接尾辞を 3 分探索木 (ternary search tree) で表現し探索木を頻度によって枝刈りしながら頻出部分文字列を列挙する事と同等である。つまり、分割する文字 v が探索木の各ノードに相当し、子孫の葉の数が閾値 ξ 以上のノードのみを残した探索木を描く。例として、文字列 s =*sakurasaku\$*、閾値 $\xi = 2$ でこのアルゴリズムが描く探索木を図 1 に示す。図の斜線部が閾値によって枝刈りされた探索空間である。

4. アルゴリズムの実装

3. 章では、アルゴリズムの概要を説明するために文字列 s 中のすべての接尾辞の配列 $\boldsymbol{S}$ を入力とする素朴な再帰手続きを考えた。しかし各接尾辞は重複した情報をもっており領域の使用効率が良くない。そこで、接尾辞を示すインデックスの配列を考えその並び換えをすることで、$\boldsymbol{S}$ 及び、分割された配列

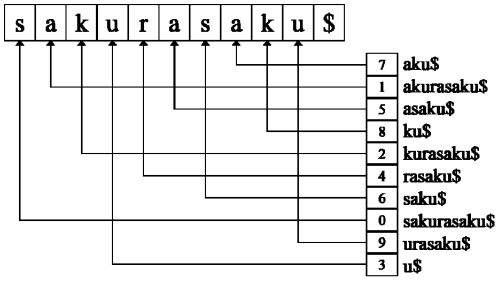


図 2　接尾辞配列の例
Fig. 2　Example of suffix arrays.

$(\boldsymbol{S'}_{<}, \boldsymbol{S'}_{=}, \boldsymbol{S'}_{>})$ を生成することなく実装する方法を示す。

インデックス配列 $\boldsymbol{I}$ の長さは文字列 s の長さ n と同じであり、その初期値を、$\boldsymbol{I} = [0, \cdots, n-1]$、つまり s の各文字の位置を先頭から順に入れたものとする。配列の i 番目 $(0 \leqq i < n)$ の値が $j = \boldsymbol{I}[i]$ のとき、$\boldsymbol{I}[i]$ は疑似的に s の j 文字目から始まる接尾辞 suffix_j を示し、その接尾辞の文字列は $s[\boldsymbol{I}[i]..n-1]$ としてアクセスされる。再帰手順では、分割された配列 $(\boldsymbol{S'}_{<}, \boldsymbol{S'}_{=}, \boldsymbol{S'}_{>})$ を新たに生成する代わりに、インデックス配列を $\boldsymbol{I} = \boldsymbol{I}_{<}, \boldsymbol{I}_{=}, \boldsymbol{I}_{>}$ となるように並び替える。ただし、$\boldsymbol{I}_{=}$ は $\boldsymbol{S'}_{=}$(d 文字目までが等しい接尾辞の配列) を示すインデックスのみを含む配列である。同様に $\boldsymbol{I}_{<}$ は $\boldsymbol{S'}_{<}$ を示すインデックス配列であり、$\boldsymbol{I}_{<}$ は $\boldsymbol{S'}_{>}$ を示すインデックス配列である。次の再帰呼び出しでは、それぞれ分割された配列を渡す代わりに、並び替えられたインデックス配列内の分割領域の始点と終点を示すポインタを渡す。付録 1. に、インデックス配列を使用した C++ [23] 言語による実装を示す。

接尾辞の配列 $\boldsymbol{S}$ には $(n(n-1)/2)$ 文字が必要であり、かつ再帰手続きの 3 分割でさらに接尾辞の配列 $\boldsymbol{S'}$ の領域を使うが、インデックス配列を使う実装では入力文字列の長さ n に対して、n ワード (ワードサイズは少なくとも $\log n$ ビット) の領域のみを使い効率的である。

上記のインデックス配列を使った実装は、接尾辞配列 (suffix array) [15] を整列によって作成する過程に非常に類似している。接尾辞配列は文字列の接尾辞を表現するインデックスを辞書順に並べた配列であり、2 分探索により部分文字列の出現位置の検索を行うことができる。例として s =*sakurasaku\$*の接尾辞配列を図 2 に示す。インデックスの入った縦の配列が接尾辞配列で、配列の横には各インデックスが表す接尾辞を示してある。本手法は、閾値による探索空間の枝刈りを除けば、Bentley と Sedgewick による Multikey Quicksort [8] によってインデックス配列を整列し接尾辞配列を生成する過程と等しい。Multikey Quicksort は、文字列のように順序付きの要素列を値として持つデータを整列する場合に有効な方法である。提案する実装は各再帰において、ある閾値 ξ 以上出現しない要素以降は無視して整列している事と同等である。Multikey Quicksort と同様に、配列 $\boldsymbol{S'}$ の全ての接尾辞の d 番目の文字の中から分割する文字 v として中央値を選ぶ事ができれば、このアルゴリズムの計算量の期待値は $O(n \log n)$ になる [8]。この部分的な整列の結果として、頻出部分文字列の列挙が完了した時のインデックスの

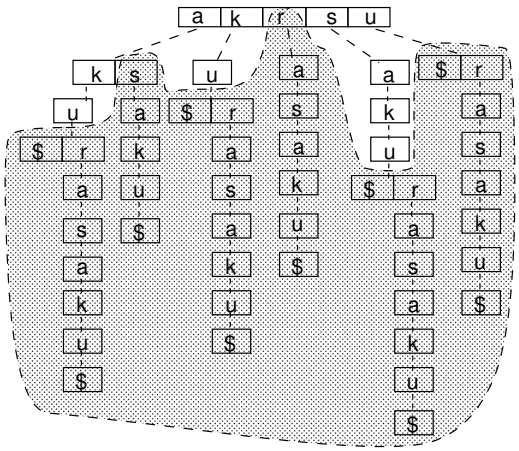


図 3　枝刈りされた接尾辞トライ
Fig. 3　A pruned suffix trie.

並びは、頻出する部分文字列のみを検索することができる接尾辞配列となる。この性質を使うことで、頻出部分文字列の元文字列中でのコンテクストをブラウジングすることが可能になる。さらに、最終的なインデックス配列を使うと、頻出部分文字列が次の 3 つの数字によりコンパクトに表現できる。具体的には各頻出部分文字列の (1) 長さ d、(2)$\boldsymbol{I}$ 内の分割領域 $\boldsymbol{I}_{=}$ の始点 offset、(3) 出現頻度 count で表現する。部分文字列の文字列表現は $s[\boldsymbol{I}[\text{offset}]..\boldsymbol{I}[\text{offset}]+d]$ で得られる。さらに、頻出部分文字列の s 内の全ての出現位置は $\boldsymbol{I}[\text{offset}], \cdots, \boldsymbol{I}[\text{offset}+\text{count}]$ である。

5. N-gram PrefixSpan

工藤ら [14] は、系列パターンマイニングアルゴリズムである PrefixSpan [20] を拡張し、4.3 章で頻出部分文字列のマイニング手法を提案した (N-gram PrefixSpan)。N-gram PrefixSpan と提案手法は非常に類似しているが、提案手法が接尾辞の配列を 3 分割するのに対し、N-gram PrefixSpan では各再帰手順において d 文字目の文字の異なりの数分割をおこなう (つまり $|\Sigma|$ 分割)。工藤らのアルゴリズムの手順を提案手法の疑似コードに合わせて書き直すと以下の通りになる。ただし、実際には提案手法と同様に N-gram PrefixSpan でも接尾辞の配列を作る代わりにインデックス配列を使用する。

mine$(\boldsymbol{S}', d)$
  for all $\text{suffix}_i \in \boldsymbol{S}'$
    $\boldsymbol{S}'_\sigma \leftarrow \text{suffix}_i$
    where $\sigma = \text{suffix}_i[d] (\sigma \in \Sigma)$
  end
  for all $\boldsymbol{S}'_\sigma$ where $\sigma \in \Sigma$ (n_σ は $\boldsymbol{S}_\sigma$ のサイズ)
    if $n_\sigma \geqq \xi$ **and** $\sigma \neq \$$ **then**
      部分文字列 $\text{suffix}_i[0..d] (\text{suffix}_i \in \boldsymbol{S}'_\sigma)$ を出力
      mine$(\boldsymbol{S}_\sigma, d+1)$
    end
  end
end

$|\Sigma|$ 分割の結果、探索木が提案アルゴリズムでは 3 分木になるのに対し、N-gram PrefixSpan ではトライになる。例として、文字列 $s = sakurasaku$ を閾値 $\xi = 2$ でマイニングしたときの探索木を 図 3 に示す。また、4. 章で提案アルゴリズムと Multikey Quicksort との類似性を示したが、工藤らの手法は基数を $|\Sigma|$ とした時の基数ソートと類似している。 表 1 に、提案手法と N-gram PrefixSpan と整列法、探索木との対応関係を示す。

マイニングアルゴリズム	整列法	探索木
提案手法	Multikey Quicksort	3 分木
N-gram PrefixSpan	基数ソート	トライ

表 1　整列法、探索木との対応関係

N-gram PrefixSpan は $|\Sigma|$ 分割するために提案手法に比べて探索木の深さが浅くなるという利点がある。一方で各再帰手順で $\boldsymbol{S}'$ のサイズに対して線形時間で $|\Sigma|$ 分割するには $\boldsymbol{S}'$ と同じ領域が必要になり、提案手法に比べて N-gram PrefixSpan は 2 倍以上の作業領域が必要になる。

6. 接尾辞木

ここでは、接尾辞木 [9], [16], [19] を使った頻出部分文字列の列挙方法を簡単に示す。

接尾辞木 (Suffix Tree) は文字列の全ての接尾辞を含んだトライのような構造をしている。図 4 に $s =$*sakurasaku*\$の接尾辞木の例を示す。接尾辞トライとの違いは、枝分かれのないノードを削除するパス圧縮 (path compression) である [5], [19]。パス圧縮により最悪の場合でノード数は最大 $2n$ ですむため、接尾辞トライよりも接尾辞木は領域使用効率が良い。

接尾辞木の根から各ノードまでのパスは入力文字列内の部分文字列を表現している。各部分文字列の出現頻度は、部分文字列の終端ノードの子孫の葉の数と等しい。ノード v の子孫の葉の数 $\text{count}(v)$ は以下のように再帰的に計算することができる。

$$\text{count}(v) = \begin{cases} 1 & \text{if } \mathit{children}(v) = \emptyset \\ \sum_{c \,\in\, \mathit{children}(v)} \text{count}(c) & \text{otherwise} \end{cases}$$

ただし、$\mathit{children}(v)$ はノード v の直下の子供の集合を示す。よって、全てのノードを巡回することで全てのノードの $\text{count}(v)$

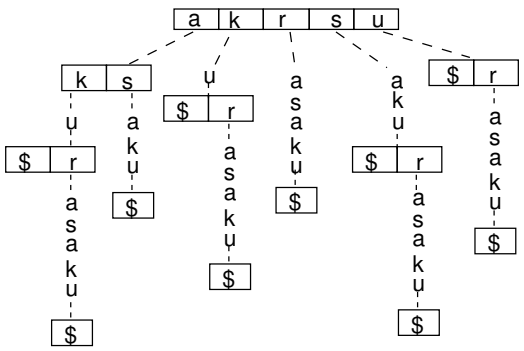


図 4　接尾辞木の例
Fig. 4　Example of suffix trees.

を計算することができ、接尾辞木のノードの数が $O(n)$ なのでその計算時間は $O(n)$ である。部分文字列の終端ノードの子孫の葉の数 count(v) が閾値より大きいとき (count(v) $> \xi$) その部分文字列は頻出部分文字列であるから、全てのノードを巡回することで全ての頻出部分文字列を列挙することができる。ただしパス圧縮によってノード v から親ノードへの枝は枝分かれのない暗黙ノードを表現しているため、count(v) $> \xi$ のとき v の暗黙ノードが表す部分文字列も頻出である。

接尾辞木は、Ukkonen [24] により接尾辞木を線形時間で作成するオンラインアルゴリズムが提案されている。よって、接尾辞木の構築と上記の接尾辞木による列挙法をあわせても、n に比例する時間で頻出部分文字列の列挙が行える。しかし、接尾辞木の生成は線形とは言っても内側のループに非常に多くの命令があり入力サイズにかかる定数部分が大きい。より深刻な問題は、各ノードから子ノードへのリンクを表すデータ構造に必要な領域である。リンクを表すデータ構造に文字のサイズ $|\Sigma|$ の配列を使用した場合、接尾辞木全体で必要となる領域計算量は $\Theta(n|\Sigma|)$ となる。そのため、大きいファイルではかなり速くなるかも知れないが、$|\Sigma|$ が大きいデータでは作業領域の大きさが負担になると考えられる。この問題を 7. 章で計算機実験により示す。

7. 計算機実験

7.1 実験手法

本提案手法の有効性を検証するために、N-gram PrefixSpan および接尾辞木を使う方法と比較する実験を 2 つの視点から行った。

実験 1 では、閾値を変化させた場合の計算時間 (秒) を提案手法 (ternary-patition) と接尾辞木を使用した方法 (suffixtree) とで比較した。ただし、頻度閾値はデータのバイトサイズに対する比率 (%) によって表す。閾値は、 0.001% から 0.0001% まで変化させた。これは 5MB の文字列の場合、出現回数にすると 52 回から 5 回まで閾値を変化させた事に相当する。

実験 2 は入力文字列のサイズを変化させた場合の計算時間を比較した。閾値はデータサイズの 0.001%に固定した。入力文字列にはデータサイズを 1MB から 31MB に変化させて使用した。ただし、接尾辞木の実装では 13MB 以上 (suffixtree(balacedtree))、または 22MB 以上 (suffixtree(sortedvector)) のデータではメモリーを使い果したため、1MB から 12MB および 21MB までの文字列においてのみ評価した。

実験で使用した入力文字列データは、自然言語データと DNA シークエンスである。自然言語データは、UCI KDD Archive [(注2)]が公開しているニュースグループの記事を、ヘッダ情報の除去や空白の正規化をして一つの長い文字列として使用した。文字集合のサイズ ($|\Sigma|$) は記号や制御記号を含め 141 であり、全ての記事をつなげたもので 31MB である。DNA シークエンスは、 Wisconsin 大学ゲノムセンターが公開している大腸菌 (strain K-12, substrain MG1655, version M52) [(注3)]のアノテーションなしのデータを使用した。文字集合のサイズ ($|\Sigma|$) は 4(AGTC) であり、データサイズは 4.4MB である。実験 1 では、ニュースグループ記事 5MB と DNA シークエンス 4.4MB の両方のデータで評価した。実験 2 では、ニュースグループ記事データのみを使用した。

	パラメータ	固定値	データ
1	閾値 (0.0001% – 0.001%)	入力サイズ (5, 4.4MB)	News・DNA 列
2	入力サイズ (1 – 31MB)	閾値 (0.001%)	News

表 2 実験設定
Table 2 Experimental Parameters.

実験の詳細を表 2 にまとめる。

接尾辞木と提案手法は、それぞれ 2 通りの実装を用意した。接尾辞木の実装は子ノードへのリンクのデータ構造に選択肢があり、提案手法の実装は分割の基準となる値 v の選択方法に選択肢がある。

接尾辞木を生成するアルゴリズムは Ukkonen [24] の方法を用いた。接尾辞木の実装では各ノードから子ノードへのリンクを表現するデータ構造によって領域使用量が大きく変わる [9]。この比較実験では、接尾辞木のノードとして平衡木[(注4)]を採用した実装 (balancedtree) と整列した配列[(注5)]を採用した実装 (sortedvector) を使った。k をノードの子供の数とすると、平衡木を使用した場合は子の追加および探索は $O(\log k)$ であり、整列した配列の場合は追加に $O(k)$ で探索は $O(\log k)$ 時間である。

提案アルゴリズムは、分割の基準となる値 v の選択方法として 2 つ選択肢を設けた。一つはランダムにピボット v を選ぶ実装 (rand) と、もう一つはサンプリングによって疑似中央値を選択する実装 (pseudomedian) を採用した。

実験環境は、CPU は Intel Pentium 4 2.67GHz、メインメモリーは 2.5GB、OS は Linux を使用した。実装言語は C++ で、gcc(version 3.2) の-O3 最適化オプションを使用してコンパイルした。

7.2 結果と考察

実験 1、2 の結果を図 5、 6 に示す。計算時間は、3 回の試行の平均時間を採用した。ただし、入力文字列の読み込み時間は計算時間に含めない。

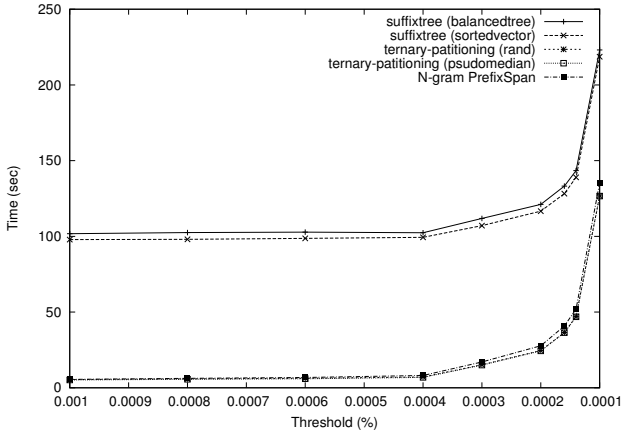
実験 1 では、図 5 のグラフに示されるように、自然言語データ (図 5(a)) と DNA シークエンス (図 5(b)) の両方において閾値に依らず提案アルゴリズムは接尾辞木を使用した方法に対して良い性能を示した。両手法の速い実装同士を比較すると平均してそれぞれ 4 倍 (自然言語)2 倍 (DNA) 提案手法の方が速い。N-gram PrefixSpan との比較ではそれほど性能に差はなかったが、平均的には提案手法の方が速かった。全てのアルゴリズムにおいて閾値を小さくした場合に急激に性能が悪化する理由は、閾値を小さくする事によって発見される頻出部分文字列数

(注2)：<http://kdd.ics.uci.edu/>

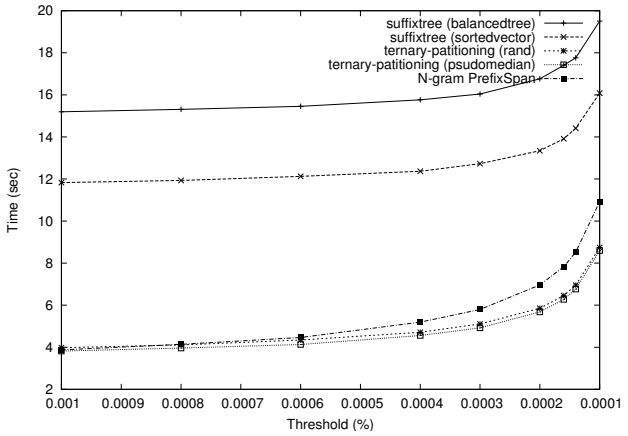
(注3)：<http://www.genome.wisc.edu/sequencing/k12.htm>

(注4)：C++ Standard Template Library の map

(注5)：Alexandrescu [4] による AssocVector



(a) news group articles (5MB)



(b) E. coli K-12 genome sequence (4.4MB)

図 5　閾値を変化させた時の計算時間
Fig. 5　Execution time with threshold parameter.

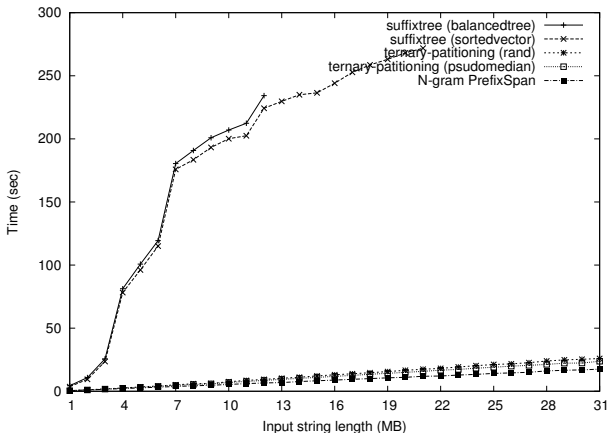


図 6　データサイズを変化させた時の計算時間
Fig. 6　Execution time with input length parameter.

が指数的に増えるためである。

実験 2 でも、図 6 のグラフで示されるように提案アルゴリズムはデータサイズに依らず、接尾辞木を使用した方法と比べて

アルゴリズム	メモリ使用量 (MB)
suffixtree(map)	693
suffixtree(sortedvector)	391
ternary-partitioning	26
N-gram PrefixSpan	50

表 3　5MB の入力に対する各実装のメモリー使用量
Table 3　Memory Usage for 5MB of text

とても良い性能を示した。N-gram PrefixSpan と比較すると、データ数が増えるにつれてわずかながら N-gram PrefixSpan が良い性能を示した。

次に各アルゴリズムの領域使用率を示す。表 3 は、5MB のニュースグループ記事データの処理における各実装の最大使用メモリーをしめす。提案アルゴリズムは使用メモリーが入力文字列サイズの 5 倍程度に収まっているため大きなデータに対しても適応可能である一方、接尾辞木を使った場合は非常に大きな作業領域が必要であることがわかる。N-gram PrefixSpan は提案手法のおよそ倍の作業量を使用した。

また、各実装の違いは各手法の違い程、性能に影響しないことがわかった。なお、接尾辞木の比較実験において sortedvector に比べて理論的には挿入・検索が速いはずの balancedtree が遅いのは、STL の map がバランスを保つために複雑な仕組みを持っているためと考えられる [6]。提案手法では、v の選択にサンプリングを使った方法 (pseudomedian) がランダムに選択する方法に比べて常に良い結果であった。

この結果から、理論的な計算量は線形時間のアルゴリズムである接尾辞木による方法が勝っているが、現実の性能では提案手法が勝っていることが示された。この接尾辞木と提案手法の関係は、線形時間の整列法とクイックソートなどのよりシンプルな整列法との関係に非常に類似している。基数整列法などの線形時間アルゴリズムは非常に特殊な目的には向いている場合があるが、汎用性においてはクイックソートに及ばない [22]。

8. 関 連 研 究

森と長尾 [18] は全ての可変長部分文字列を列挙し、その頻度をカウントするデータ構造を提案した。それらの部分文字列を使うことでデータに含まれる未知語、複合語、コロケーションなどが抽出できると報告している。提案されたデータ構造は、実際には接尾辞配列と高さ配列 (height array) [15] と同等のものである。Kasai ら [11] も、全ての部分文字列を接尾辞配列と高さ配列で列挙する方法を提案している。

Krishnan ら [13] はデータベースの部分文字列クエリの選択推定を、頻度により枝刈りした頻度接尾辞木 (count-suffix tree) によって行うことを提案している。頻度接尾辞木は接尾辞木のノードに元文字列で部分文字列の頻度を記録したデータ構造である。この枝刈り頻度接尾辞木を生成する手順は 6. 章で説明した接尾辞木を使った頻出部分文字列の列挙方法とほぼ同じであると考えられる。

9. 結　　論

本論文では、シンプルで計算時間および領域使用効率が良く、部分文字列のブラウジングも容易な頻出部分文字列マイニング

手法を提案した。このアルゴリズムで列挙された頻出部分文字列は可変長 N グラムモデルを構築する候補集合になる。また、自然言語だけでなく Web サイトのアクセス記録や商品購買記録などの離散的な時系列データも文字列と考える事ができるため、自然言語処理分野以外での応用も期待される。

最後に、今後の展開として以下の二つを紹介する。

頻度以外の閾値による探索空間の枝刈り： 森下ら [17] はカイ二乗値やクロスエントロピーなど二次統計量の上限を指標にして探索空間の枝刈りを行なう手法を提案している。この方法を使うことで、可変長 N グラムモデルで N グラムの選択に使われる統計値を使い直接有効な N グラムを発見することができると考えられる。

閉じたパターン： 閉じた頻出パターンマイニング (frequent closed pattern mining) は、データマイニング分野で近年研究がおこなわれている問題である。閉じたパターンとは同じ頻度をもつパターンであればもっとも長い上位のパターンのみを列挙する問題である [25]。例えば、文字列 $s = sakurasaku\$$、閾値 $\xi = 2$ の時、頻出部分文字列は $a(3)$, $ak(2)$, $aku(2)$, $k(2)$, $ku(2)$, $s(2)$, $sa(2)$, $sak(2)$, $saku(2)$, $u(2)$ である一方、閉じた頻出部分文字列は、$a(3)$, $saku(2)$ だけである (括弧内は部分文字列の頻度)。この例からもわかるように、閉じた頻出パターン数は頻出パターン数に比べて非常に少なくなる。また、最長の頻出パターン (例では $saku$) だけを出力する場合と比べるとより短いパターン (a) の頻度の情報 (3 回) が落ちない点で優れている。この様に、閉じた頻出部分文字列は頻出部分文字列の頻度情報を落とさない最適な圧縮形式となっており、より有効な出力形式であると考えられる。

10. 謝　　辞

渋谷哲朗氏には、接尾辞木の実装および DNA シークエンスデータに付いて有益なアドバイスを頂いた。ここに感謝を示したい。

文　　献

[1] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Optimized Substructure Discovery for Semi-structured Data. In *Proc. of Conference on Principles and Practice of Knowledge Discovery in Databases*, 2002.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pp. 487–499. Morgan Kaufmann, 12–15 1994.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. of International Conference of Data Engineering*, pp. 3–14. IEEE Press, 6–10 1995.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[5] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[6] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, second edition, 2000.

[7] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. Vol. 23, No. 11, pp. 1249–1265, 1993.

[8] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1997.

[9] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[10] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. In *Proc. of Conference on Principles and Practice of Knowledge Discovery and Data Mining*, 2000.

[11] Toru Kasai, Hiroki Arimura, and Setsuo Arikawa. Efficient substring traversal with suffix arrays. Technical report, Department of Informatics, Kyushu University, 2001.

[12] Kenji Kita. *Stochastic Language Model*. University of Tokyo Publisher, 1999.

[13] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pp. 282–293, 1996.

[14] Taku Kudo, Kaoru Yamamoto, Yuta Tsuboi, and Yuji Matsumoto. Text mining using linguistic information. In *IPSJ SIGNL-148 (in Japanese)*, 2002.

[15] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1990.

[16] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, Vol. 23, No. 2, pp. 262–272, 1976.

[17] Shinichi Morishita and Jun Sese. Traversing itemset lattice with statistical metric pruning. In *Proc. of Symposium on Principles of Database Systems*, pp. 226–236, 2000.

[18] Makoto Nagao and Shinsuke Mori. A New Method of N-gram Statistics for Large Number of n and Automatic Extraction of Words and Phrases from Large Text Data of Japanese. In *Proc. of International Conference on Computational Linguistics*, 1994.

[19] Mark Nelson. Fast string searching with suffix trees. *Dr. Dobb's Journal*, 1996.

[20] Jian Pei, Jiawei Han, B. Mortazavi-Asl, Q. Chen H. Pinto, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of International Conference on Data Engineering*, pp. 215–224, Heidelberg, Germany, April 2001.

[21] Dana Ron, Yoram Singer, and Naftali Tishby. *The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length*. Kluwer Academic Publishers, 1996.

[22] Robert Sedgewick. *Algorithms in C++ (second edition)*. Addison-Wesley, 1992.

[23] B. Stroustrup. *The C++ Programming Language, 2d edition*. Addison-Wesley, 1991.

[24] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, Vol. 14, No. 3, pp. 249–260, September 1995.

[25] Jianyong Wang, Jiawei Han, and Jian Pei. CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. In *Proc. of International Conference on Knowledge Discovery and Data Mining*, 2003.

[26] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proc. of International Conference on Data Mining*, 2002.

[27] Mohammed J. Zaki. Efficiently Mining Frequent Trees in a Forest. In *Proc. of International Conference on Knowledge Discovery and Data Mining*, 2002.

付　　録

1. 提案アルゴリズムの C++による実装

ここでは、4. 章で説明した提案アルゴリズムの実装の C++言語 [23] による具体例を示す。以下に示すサンプルコードは、Multikey Quicksort の論文 [8] のサンプルコード *Program 1. A C program to sort strings* を基にしている。

3 分割法は Bentley ら [8] の効率的な実装に従い、d 文字目が等しい接尾辞を示すインデックスを一時的にインデックス配列の両端に配置し、最後に中央に移動する方法を採用している。この方法を図 A·1 に示す。

最初に以下のサンプルコードの変数の説明をする。入力文字列 *str* は **string** 型、インデックス配列 *idx* は *str* の **size** の値を要素として持つ **vector** である。*str*、*idx* ともグローバル変数またはメンバ変数として、以下に示す関数から自由にアクセスできるものとする。入力文字列の終端文字は、*str* の **value** 型の定数 *EOS* とする。頻度閾値 *minsup* は *idx* の **size** 型である。これらの値も自由にアクセスできるものとする。

```
typedef string::size_type s_t;
typedef string::value_type v_t;
typedef vector<s_t>::size_type i_t;
string str;
vector<s_t> idx;
i_t minsup;
```

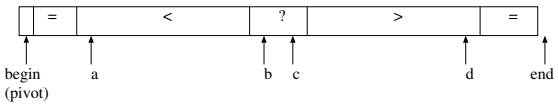
インデックス配列 *idx* の初期値は以下のように初期化される。

```
for(s_t i = 0; i < str.size(); i++) {
  idx.push_back(i);
}
```

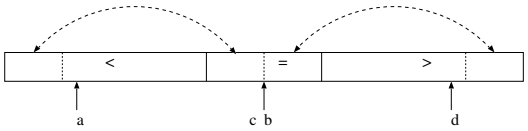
以下が、中心のマイニング関数である。

```
void mine(i_t begin, i_t end, s_t depth)
{
    if(end - begin < minsup) {
        return;
    }
    i_t pivot = selectPivot(begin, end);
    swap(idx[begin], idx[pivot]);
    v_t t = getValue(begin, depth);
    i_t a = begin+1, c = end-1;
    i_t b = a      , d = c;
    v_t r;
    while(true) {
        while(b <= c && ((r=getValue(b, depth)-t) <= 0)) {
            if (r == 0) { swap(idx[a], idx[b]); a++; }
            b++;
        }
        while(b <= c && ((r=getValue(c, depth)-t) >= 0)) {
            if (r == 0) { swap(idx[c], idx[d]); d--; }
            c--;
        }
        if(b > c) {
            break;
        }
        swap(idx[b], idx[c]);
        b++;
        c--;
    }
    i_t range = min(a - begin, b - a);
    vectorSwap(begin, b - range, range);
    range = min(d - c, end - d - 1);
    vectorSwap(b, end - range, range);

    range = b - a;
    mine(begin, begin + range, depth);

    i_t eq_begin = begin+range;
    i_t eq_end   = range+a+end-d-1;
    if((eq_end - eq_begin) >= minsup && t != EOS) {
        print(eq_begin, eq_end-eq_begin, depth);
        mine(eq_begin, eq_end, depth+1);
    }
    range = d - c;
    mine(end - range, end, depth);
}
```

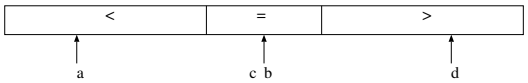
上記の関数は *mine(0, str.size(), 0);* として最初に呼ばれる。



(a) starts partitioning from both sides



(b) exchanges the equal regions



(c) the end of ternary partitioning

図 A·1　3 分割法の実装
Fig. A·1　Ternary Partitioning Implementation

以下で、上記のマイニング関数の補助関数を示す。

getValue 関数は接尾辞 suffix_i の *depth* 版目の文字を返す。

```
v_t getValue(i_t i, s_t depth)
{
    return str[idx[i] + depth];
}
```

vectorSwap 関数は一時的に *idx* の両端に位置している *pivot* と等しい文字を示すインデックスを中心に動かすのに使われている。

```
void vectorSwap(i_t i, i_t j, i_t len)
{
    while(len-- > 0) {
        swap(idx[i], idx[j]);
        i++;
        j++;
    }
}
```

分割の基準値 *pivot* の選び方は様々な方法が考えられるがここでは現在のインデックスの範囲からランダムな値を返す関数を示す。

```
i_t selectPivot(i_t begin, i_t end)
{
    return begin + rand() % (end - begin);
}
```

print 関数は発見された頻出部分文字列をその頻度とともに出力する。

```
void print(i_t offset,i_t count,v_t depth)
{
    cout << count << "\t"
         << str.substr(idx[offset], depth+1)
         << endl;
}
```