自 然 言 語 処 理 129－5
(1 9 9 9. 1. 2 1)

大規模テキストに対するSuffix Arrayの効率的な構築法

伊東秀夫
(株) リコー情報通信研究所
〒222 神奈川県横浜市港北区新横浜3-2-3
hideo@ic.rdc.ricoh.co.jp

Suffix arrayは文字列索引の一種であり、suffix treeに比べ単純でコンパクトなデータ構造で実装できる。文字列処理に対して多くの優れた性質を持つsuffix arrayだが、特に大規模なテキストに対しては索引構築に多大な記憶量と計算コストを必要とし実用上の問題なっている。我々は、高速かつコンパクトなsuffix array構築法を提案する。そのキーとなるアイデアは、任意のsuffix間の関係ではなく、隣接するsuffix間の関係のみを利用する点にある。このアルゴリズムを二段階ソート法と呼ぶ。514MBの毎日新聞記事を含む様々なデータセットを用いた評価実験により、我々のアルゴリズムはQuicksortの4.5～6.9倍高速であり、また、今までで最も高速なアルゴリズムとして知られているSadakaneの方法に対し2.5～3.6倍高速であることが示される。

An Efficient Method for Constructing Suffix Arrays of Large Texts

Hideo Itoh
Information and Communication R & D Center
3-2-3, Shin'yokohama, Kohoku-ku, Yokohama, Kanagawa, 222, Japan
hideo@ic.rdc.ricoh.co.jp

The *Suffix array* is a string indexing structure and a memory efficient alternative of the *Suffix tree*. It has myriad virtues on string processing. However, it requires large memory and computation to build suffix arrays for large texts. We propose an efficient algorithm for sorting suffixes. One of the key ideas is to use specific relationships between an adjacent suffix pair. We call this algorithm the *Two-Stage Suffix Sort*. Our experiments on several text data sets (including 514MB japanese newspapers) demonstrate that our algorithm is 4.5 to 6.9 times faster than the popular sorting algorithm Quicksort, and 2.5 to 3.6 times faster than Sadakane's algorithms which is known as the fastest one.

1 はじめに

電子化データの蓄積が進み、大規模なデータ集合に含まれている情報を効率的かつ効果的に利用したいという要求が高まってきている。電子化データの多くは文字列であるから、この要求を満たすことを目的とする多くの応用系において文字列検索 [1][2] は重要である。

大規模な文字列に対し高速な検索を可能にするには索引の利用が欠かせない。従来、文字列索引のためのデータ構造として suffix automaton[3]、suffix tree[4]、suffix array[5] など提案されてきた。いずれの構造も、suffix と呼ばれる文字列を索引単位とする。ここで suffix とは、索引対象となる文字列T中の任意の位置からTの末までの範囲の文字列である。Tの長さ (文字数) が N であれば N 個の suffix が定義され、T中の文字出現位置と一対一に対応する。検索キーとなる文字列Qが与えられた場合、Tの全 suffix の集合の中から、Qに半直線マッチする suffix を効率的に求めることに文字列索引は利用される。

文字列索引は、ゲノムデータベースの構築と利用に関連する研究が技術的発展の牽引力となったが [2][6]、これらに限らず幅広い応用の可能性がある [7]。とくに近年の計算機パワーの増大から、テキスト圧縮、テキスト検索/分析、コーパスベースの自然言語処理などの分野において、大規模な自然言語テキストを索引対象とした応用が広がりつつある [8] [9][10] [11][12]。

前述した諸データ構造の内、大規模なテキストを対象とする文字列索引としては、suffix array が最も実用的である。suffix array の特長を示す。

1. 索引のサイズが最もコンパクト
2. 字彙のサイズに依存しない計算量
3. 高速な検索
4. シンプルな枠組

しかし suffix array には以下の問題点がある。

1. 索引構築時の計算コスト (時間と記憶量)
2. 動的データ (文字列の追加や削除) の扱い

特に上記の問題1は、大規模テキストを索引対象とする場合に実用上の障害になる。また小規模テキストTに対しても、もし非常に高速に suffix array が構築できるならば、Tを質問入力とするテキスト検索など、即応が要求される様々な応用にも適用範囲が広がる。このような背景から上記問題1に対し、我々は従来法に比べて効率的な構築アルゴリズムを提案する。

2 Suffix array

Suffix array は、Manber と Myers [5] により提案された文字列索引法である[1]。以下に、本稿で用いる記法とともに、suffix array を定義する。

長さ N の文字列を $a_0, a_1, \ldots, a_{N-1}$ で表す。ここで各 a_i は、アルファベットの有限集合 Σ の要素であり文字と呼ぶ。$|\Sigma|$ によりアルファベットの総数を表す。文字には固有の文字値が定義されており、この文字値に基づいて文字列間にはいわゆる辞書順 $<, =, >$ が定義される。テキスト $T = a_0, a_1, \ldots, a_{N-1}$ に対し文字列 $S_i = a_i, a_{i+1}, \ldots, a_{N-1}$ をテキスト T の先頭から i 番目の文字位置から始まる suffix と呼ぶ。この文字位置 i をポインタと呼ぶ。Suffix array は全ての suffix を辞書順に並べて得られる長さ n のポインタ列 $A = p_0, p_1, \ldots, p_{N-1}$ である。すなわち suffix 間の辞書順は $S_{p_0} < S_{p_1} < \ldots < S_{p_{N-1}}$ となる。

なお、文字列検索は suffix array を介してテキスト中を二分探索することで行われる。

本稿では、suffix 間の辞書順を確定するためにアルファベットに含まれない仮想文字 (ここでは "$") をテキスト末に加える。"$" の文字値として最小値 0 を想定する。また、文字列およびポインタ列を表現するデータ構造として配列を用いる。配列 X 中の添字 $i = i, i+1, .., j$ $(i \leq j)$ に対応する部分を $X[i, j]$ で表す。図1にテキスト "BANANA" の配列と suffix array の例を示す。

text array

0	1	2	3	4	5	6
B	A	N	A	N	A	$

suffix array

5	3	1	0	4	2

図 1: Suffix array の例

[1] Oxford English Dictionary プロジェクトにおける研究 [9]、および、n-gram の効率的獲得に関する研究 [11] においても関連する提案がなされている。

3 従来の suffix array 構築法

Suffix array の構築は、suffix の辞書順ソートにより行なわれる。一般にソートは一次記憶のみを用いる内部ソートと二次記憶も用いる外部ソートに分けられる [13]。外部ソートによる suffix array の構築法に関する研究 [9] もあるが、近年はギガバイトの主メモリが利用可能になってきたことを鑑み、内部ソートによる suffix array 構築に注目する。

3.1 文字列ソートによる方法

もっとも簡易に suffix array を構築するには、従来の汎用的なソートアルゴリズム [13] を用いればよい。文献 [14] では Quicksort を用いた suffix array の構築例が紹介されている [15]。

また、文字列を辞書順にソートする問題に対しては、MSD radix sort [13] [16] と Multikey Quicksort [17] が、現在までに提案されている主なアルゴリズムの内、最も高速である。これら 2 つのアルゴリズムは我々が提案する構築法にも関連するので、その概要を説明する。

MSD radix sort はまず図 2 に示す初期 backet sort を行う。テキスト “BANANA” に対する初期 bucket sort の例を図 3 に例示する。この処理により、テキスト中の各 suffix を、その先頭 1 文字のみに着目して辞書順にソートした際の、それら suffix へのポインタ列が配列 A 上にセットされる。

```
var T  : array[0,..,N] for text array
var A  : array[0,..,N] for suffix array
var B1,B2 : array[0,..,|Σ|] for bucket table

# Step-0 : initialize of bucket table
for i := a in Σ do B1[a] := 0

# Step-1 : counting of characters
for i := 0,..,N do B1[T[i]] := B1[T[i]] + 1

# Step-2 : allocation of buckets
c := 0
for i := a in Σ
 do
  B2[a] := c
  c := c + B1[a]                    -- (*)
 done

# Step-3 : initial set of pointers on A
for i := 0,..,N
 do
  A[B2[T[i]] := i
  B2[T[i]] := B2[[T[i]] + 1
 done
```

図 2: 初期 bucket sort のアルゴリズム

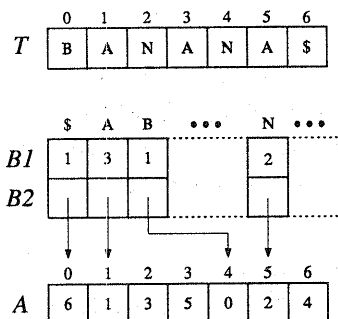

図 3: 初期 bucket sort の例

同じ先頭文字を持つ suffix へのポインタは、配列 A 上の或る連続領域に並べられる。この領域を bucket と呼ぶ。図 3 で文字 N に対する bucket は $A[5,6]$ で、そのサイズは 2 である。

次に MSD radix sort は、サイズが 2 以上の bucket について、bucket 中のポインタが指す suffix の先頭から 2 文字目に着目し、上記とほぼ同様の bucket sort により、その bucket を細分割する。このような分割を、全 bucket のサイズが 1 になるまで各 suffix 中の文字着目位置をずらしながら再帰的に繰り返すことで、配列 A 上に suffix array を得る[2]。

MSD radix sort が要する記憶量はテキストと suffix array を格納する為の配列 T, A、バケット表 $B1, B2$、および再帰に用いるスタックの総計である（バケット表は再帰の各ステップにおいて共用できるので 1 つ用意しておけばよい）。

Bentley らの Multikey Quicksort[17] は multikey（文字列等、複数要素で構成されるソートキー）を、より効率的に扱えるよう Quicksort を拡張したものである。Quicksort はソート対象となる要素配列を再帰的に 2 分割しながら処理が進むが、Multikey Quicksort では着目要素 (pivot) との比較結果が小 (<)、同 (=)、大 (>) となることに対応して 3 分割しながら処理が進む。同 (=) の部分は、着目位置を一つ進めて以降の分割を行う。このアルゴリズムは、Quicksort と MSD radix sort を効果的にブレンドしたアルゴリズムといえる。

[2] この再帰処理の為には、図 2 の処理 (*) の際に、bucket のサイズが 2 以上であれば別に用意したスタック上に、その bucket が配列 A 上に占める範囲 ($B1, B2$ の情報) と次の文字着目位置を記録しておき、以降の分割対象とすればよい。また初期 bucket sort 以外の bucket sort では、新たな作業領域を要せずに配列 A ポインタの並べ換えをするため Step-3 の代わりに permuting in place（お手玉法）[13] を用いる。

3.2 Sadakaneの方法

Sadakaneの方法 [8] は、Manberらの方法 [5] を含め現在までに提案されたsuffix arrayの構築法の中で最速のものである。図4に概要を示す。KMRアルゴリズム [18] がベースであり、以下の処理ステップからなる。

1. 初期 bucket sort
   Suffixの先頭文字に着目しsuffixへのポインタを配列 A 上でbucket sortする。各bucketの辞書順位 n (配列 A 上での開始位置) をその bucket に属する各 suffix に対応させる (numbering 法)。この対応は、配列 N (番号配列と呼ぶ) により表現する。即ち suffix S_i に対し $N[i] = n$ とする。例えば図4で文字Aに対するbucketの辞書順位は1なので $N[1] = N[3] = N[5] = 1$ となる。
2. バケットの分割
   着目位置を表す変数 k を1とする。各bucket X 毎に、X に属するsuffix S_i を N_{i+k} をキーとしてソートし、N_{i+k} の異同によりbucket X を分割する (Multikey Quicksortの三分割法を用いる)。この分割結果に沿って各suffixの番号配列の値を更新する。例えば図4で文字Aに関するbucket $A[1,3]$ はA[1,1], A[2,3]に分割され、N[5]=1, $N[1] = N[3] = 2$ となる。
3. 全てのbucketのサイズが1ならば終了、そうでなければ $k = k \times 2$ として上記ステップ2を行なう (doubling法)。

ただし上記ステップ2はサイズが2以上のbucketのみについて行なえばよい。この判断を高速化するため、配列 B (バケット配列と呼ぶ) を用意し、bucket群のサイズ (隣接するソート済みのbucketは一つに統合する) とソート済みか否かのフラグを記録して利用する。図4の配列 B の要素で負数はソート済みbucket群のサイズを表している。。

3.3 従来法の問題点

文字列ソート法はsuffixのソートに固有の性質 (suffix間の関係) を何ら利用していない。一方、Sadakaneの番号配列やManberらの転置配列 [5] は、suffix S_i から任意の距離だけ離れたsuffix S_j に関する情報 (辞書順位等) を利用可能にするため、

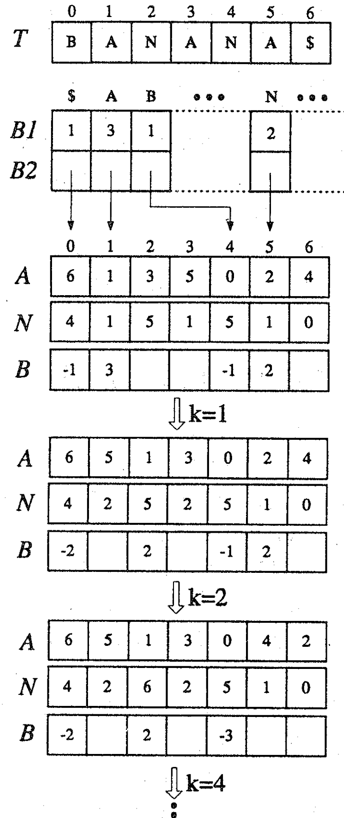

図 4: Sadakaneの方法の説明

文字列ソート法に比べ高速になると期待できる。

ソート時に必要な記憶量の点では、SadakaneやManberらの方法は文字列ソート法に比べて劣る。これはsuffix間の位置関係を陽に表現するために番号配列等を用いるからである。番号配列の要素に4 byte割り当てるならば、これだけでテキスト配列の4倍の記憶量が必要になる。

また日本語テキスト検索のように、索引対象となる日本語テキスト中に1byteと2byteの文字が混在し、かつ2byte文字の下位バイト位置については索引を施す必要がない状況が考えられるが、Sadakaneの番号配列等による方法では、このような部分索引は不可能である。

我々はsuffix間の隣接関係のみを利用することで、高速かつコンパクトであり上記の部分索引も可能なsuffix array構築アルゴリズム (二段階ソート法) を得た。このアルゴリズムを次節で説明する。

4　二段階ソート法

4.1　アルゴリズム

アルゴリズムを説明するための記法を定義する。$\leq, >$ は文字列間の辞書順を表す。suffix S_i の長さ k (>0) の prefix を $P(S_i, k)$ で表す。任意の二つの suffix S_i と S_j の間に関係 $\leq_k, >_k$ を次のように定義する。

- $S_i \leq_k S_j \Leftrightarrow P(S_i, k) \leq P(S_j, k)$
- $S_i >_k S_j \Leftrightarrow P(S_i, k) > P(S_j, k)$

我々のアルゴリズムは、次のステップからなる。

step-0 初期 bucket のセット
テキスト配列を走査し各 suffix を先頭文字に着目し bucket に分配する。ただし各 bucket をさらに次の2つのタイプに分ける。

Type A $S_i >_1 S_{i+1}$ を満たす S_i の bucket
Type B $S_i \leq_1 S_{i+1}$ を満たす S_i の bucket

Type B の bucket に属する suffix へのポインタのみを配列 A 上にセットする。

step-1 Type B の suffix に関するソート
サイズが2以上の Type B の bucket を文字列ソート法により配列 A 上でソートする。

step-2 Type A の suffix に関するソート
配列 A の要素 i を辞書的昇順に取り出す。S_{i-1} と S_i の関係が Type A であれば、$i-1$ を文字 $T[i-1]$ に対する Type A の bucket の領域にセットする。

以上の処理ステップを図5を用いて説明する。最初の step-0 では、テキスト配列を走査し文字毎に2種類の bucket (Type A と Type B) の情報をバケット表にセットする。例えば図中で文字 "A" を先頭とする suffix は S_1, S_3, S_5 である。suffix S_1, S_3 は Type B に属する。なぜならテキスト中で S_1, S_3 の直後に位置する suffix S_2, S_4 に対し $S_1 \leq_1 S_2, S_3 \leq_1 S_4$ が成り立つからである。一方、$S_5 >_1 S_6$ なので S_5 は Type A に属する。これらの判断は各 suffix 間の先頭1文字の比較でできる。次に再度テキストを走査し、バケット表に基づき suffix array 上にポインタをセットする。ただし、セットするのは Type B の bucket に属する suffix へのポインタのみである。よって、図中で斜線で示した領域は空のまま残される。

次の step-1 では、サイズが2以上の Type B の bucket について文字列ソート法によりポインタのソートを行う。この結果、suffix array 中のポインタ1と3の位置が正しく決定される。

最後に step-2 で、配列 A を辞書的昇順（図中で Left to Right）に走査しながら、Type A の bucket に相当する配列 A 中の領域（斜線部分）を埋めてゆく。例えば、suffix array の先頭要素6を取り出し $S_5 <_1 S_6$ の関係にあることがテキスト配列 T を参照することで分かる。よって S_5 はその先頭文字 "A" に対する Type A の bucket X に属する。しかもこの処理は辞書的昇順に行っていることから、配列 A 上で bucket X が占める領域 $A[1,1]$（バケット表を参照すれば求まる）の内、未だポインタが埋まっていない最左位置1にポインタ5を格納すれば辞書順に並ぶ。Type A の要素を配列 A 上の空位置に格納する毎に、バケット表 $B2$ の値を右に進めれば上の処理が効率化できる。

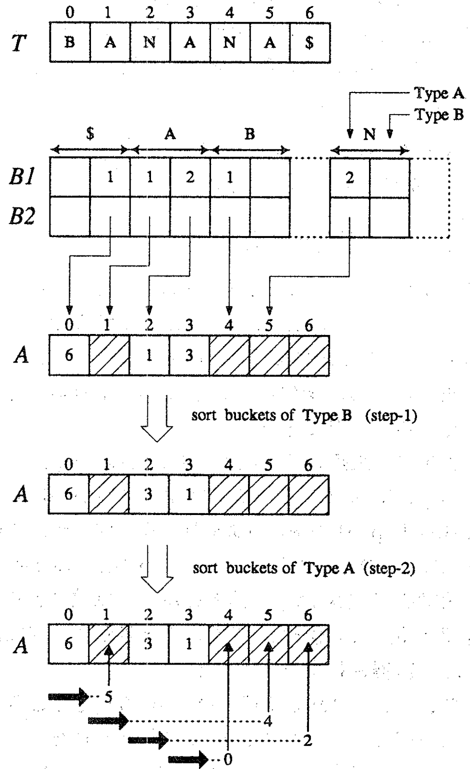

図5: 二段階ソート法の動作例

4.2 二段階ソート法の高速性

第２段階の Type A のソートは、suffix array を Left to Right に一回走査し、決定的に Type A の bucket をソートするため高速である。計算オーダーはテキスト長を N として明らかに $O(N)$ である。一方、第一段階のソートは文字列ソート法によるため線形オーダーにはならない。よって、Type B の bucket に属する suffix の数が Type A に比べて少ないほど、高速になると考えられる。

仮に Type A と Type B の suffix の数はほぼ等しく、第２段階のソート時間が第１段階のそれに比べ無視できるほど小さいならば、全体のソート時間は従来の文字列ソート法の約２倍高速になると見積もれる。

4.3 さらなる高速化

英語、音素ラベル、ゲノム (ATCG) など、1byte の文字コードからなるテキストは多い。この場合、これらの文字 2-gram を 64K のサイズのバケット表で管理したほうが効率的である。さらに、Type A と Type B の分類条件を以下のように換えてみよう。

Type A $S_i >_1 S_{i+1}$ または $S_i >_2 S_{i+2}$ を満たす S_i の bucket

Type B 上記以外の S_i の bucket

この変更により、suffix S_i が直後の suffix S_{i+1} との比較では Type B になってしまう場合でも、さらに S_{i+2} との比較で、Type A になるチャンスが生まれるため、全体に占める Type B の割合を小さくすることができる。

表１に、実際の日本語 (EUC)、英語 (ascii)、ゲノム (ascii) のテキストを文字単位に索引づけした場合の Type B の割合（実測値）を示す。rate–1 は前頁で述べた $>_1$ のみの比較による場合、rate–2 は上記で述べた $>_1, >_2$ の両方を用いた比較による場合である。この値はテキストの内容やサイズにほどんど影響されず一定である。

なおここでは詳しく述べないが、Type B の割合をなるべく小さくするように文字値の変換を行うと、英語の場合、Type B の割合はさらに 28 %に改善される。日本語とゲノムについては、この文字値変換による改善はわずかだった。

data	rate–1	rate–2
日本語	51 %	同左
英　語	51 %	35 %
ゲノム	62 %	40 %

表 1: Type B に属する suffix の割合

4.4 文字列ソート法の改良

二段階ソート法の step–1 では、文字列ソート法により各 bucket をソートする。この際に用いるソート法として MSD radix sort をベースに以下の改良を加える。

MSD radix sort は文字列ソートアルゴリズムとしては最も高速であるが問題点もある。特にバケットの配置を求めるためにバケット表全体を走査する負荷（図２の Step–2）の影響は大きい。そこでソート対象となる bucket のサイズが小さい場合は insertion sort などに切り替えることが一般的であるが [13]、この方法はバケット表のサイズが上記のように 64K の場合、あまり効果がない[3]。

この問題に対して、我々は bucket のサイズが大きい順に、MSD radix sort → Multikey Quicksort → insertion sort の３段階に文字列ソートアルゴリズムを切り替えることで対処した。

4.5 二段階ソート法の記憶量

テキストが N bytes からなるとし、suffix array の配列要素を 4 bytes で表現するものとする。

テキストを byte 単位ではなく文字単位に索引づけを施す場合について、ソートに必要な記憶量を表２にまとめる[4]。

アルゴリズム	記憶量 (英語)	記憶量 (日本語)
文字列ソート法	$5N + stack$	$3N + stack$
Manber–Myers	$8N$	$4N$
Sadakane	$9N + stack$	$5N + stack$
二段階ソート法	$5N + stack$	$3N + stack$

表 2: ソートに必要な記憶量 (byte)

[3] 文献 [19] ではバケット表のサイズを 64K から 256 に途中で切り替える方法 (adaptive radix sort) を提案しているが目立った効果は得られていない。

[4] バケット表の容量は一定であり、他に比べて無視できる大きさなので含めていない。

5 評価

5.1 目的と方法

大規模（数十 MB ～数百 MB）および小規模（～数 MB）のデータを対象に、二段階ソート法と従来法の処理時間を比較し、高速性を検証する。

従来法として Quicksort(QS)、Multikey Quicksort (MQ)、MSD radix sort (RS)、Sadakane の方法 (SS) をC言語で実装した。QS はライブラリ関数 (qsort) を利用。MQ の実装では pivot 値を 2byte とした。RS は最も高速な実装法として知られる McIlroy らの方法 [16] に従った。Sadakane 法の実装では文献 [8] に従って初期 bucket sort を 64K のバケット表を用い先頭 2bytes に着目して行う。二段階ソート法は 64K のバケット表を用い、前述の文字値変換は行っていない。いずれの実装でも仮想記憶ユーティリティ(mmap 等) は用いていない。

使用計算機は Sun Ultra30 (300MHz UltraSPARC II) で 1GB のメモリを搭載している。

大規模データとして、英語テキストは LDC から供給された DOE (報告書の抄録集)、日本語テキストは毎日新聞 91 ～ 95 年版の 5 年分（記事題名と本文）を用いる。英語テキストは byte 単位に索引づけし、日本語テキストは EUC と ascii コードが混在するため Sadakane 法では byte 単位に、それ以外は文字単位に索引づけした。

一方、小規模データとして、テキスト圧縮技術の評価によく用いられる Calgary corpus および Canterbury corpus を用いた。書籍、プログラムソース、ゲノム列 (E.coli)、ニュースが含まれる。全て ascii テキストであり byte 単位に索引づけした。

データの特徴付けとして AML（Average Matching Length）[8] を求めた。AML は辞書順位で隣り合う suffix 間の最長共通接頭長の平均であり AML が長いほどソート困難なテキストであるといえる。

data	size(byte)	AML	QS	MQ	RS	SS	ours
英語 031MB	30935873	21	411	176	163	153	59
英語 060MB	60172753	29	865	385	363	334	128
英語 180MB	179540613	29	3083	1334	1325	–	444
毎日 051MB	51129551	31	345	159	132	246	67
毎日 356MB	355858264	27	3450	1876	1673	–	763
毎日 514MB	513810521	26	11136	7881	4876	–	1818

表 3: 大規模テキストのソート時間 (sec)

計測した処理時間はソートに要する時間であり、いずれもテキストの読み込みと suffix array 構築後のディスクへのセーブ時間は含んでいない。小規模データについては 10 回、大規模データについては 3 回の計測時間の平均を取った。

5.2 結果と考察

表 3 と表 4 に各ソート時間の計測結果を示す。表中で “–” で示した箇所は、主記憶の不足により計測不能であった。

［大規模データでの比較］いずれのデータにおいても、二段階ソート法の高速性は顕著である。QS に対しては 4.5 ～ 6.9 倍、MQ に対しては 2.3 ～ 4.5 倍、RS に対しては 2.0 ～ 3.0 倍、SS に対しては英語で 2.5 倍、日本語で 3.6 倍高速である。文字列ソート法 (QS,MQ,RS) との比較では、英語の方が日本語より二段階ソートの高速性が際立っている。これは表 1 の rate–2 で示すように英語の方が日本語より Type B の割合が少ないためである。

Sadakane 法は 1byte と 2byte 文字が混在する日本語テキストでは byte 単位での索引づけになるため、QS 以外の方法に比べて遅い。また英語についてもそれほど顕著な高速性は見られない。この原因は Sadakane 法が横型アルゴリズムでありメモリ参照が局所的でないためと思われる[5]。

［小規模データでの比較］二段階ソート法は数百 KB のデータに対し 1sec 以下のソート時間であり、第 1 節で述べた即応性を要求する応用への適用も期待できる。またゲノム列は $|\Sigma|$ が小さいため (ATGC の 4 種)、ソート初期には bucket が非常に大きくなる。このような状況でも二段階ソート法は英語の場合と同じ理由で高速性が際立っている。

data	size(byte)	AML	QS	MQ	RS	SS	ours
book1	768771	7	4.91	1.55	1.13	1.76	0.74
progc	39611	8	0.17	0.05	0.04	0.04	0.04
book2	610856	10	3.65	1.11	0.86	1.21	0.53
bible	4047392	14	38.99	14.99	13.08	14.02	5.57
E.coli	4638690	17	49.91	19.25	18.39	20.97	7.94
news	377109	18	2.17	0.67	0.53	0.60	0.38
world192	2473400	23	23.24	10.11	11.04	7.97	4.02
progl	71646	25	0.37	0.13	0.12	0.08	0.10

表 4: 小規模データのソート時間 (sec)

[5] つまり主メモリの二次キャッシュ機構が効果的に働かない。

6 まとめ

大規模テキストを対象とする文字列索引法として最も実用的と思われる suffix array に着目し，従来に比べて効率的な構築法 (二段階ソート法) を提案した。また、いくつかのコーパスを用いた評価実験により、その高速性を検証した。

今後は、suffix array に関する残された問題 (索引の効率的な更新や圧縮など) に取り組み更に実用性を高めるとともに、いくつかの応用研究を行なう予定である。

参考文献

[1] G. A. Stephen. *String Searching Algorithms.* World Scientific Publishing, 1994.

[2] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford Univ. Press, New York, 1994.

[3] J. Blumer A. Blumer and D.Haussler. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, Vol. 40, pp. 31–55, 1985.

[4] E. Ukkonen. On–line construction of suffix–trees. *Algorithmica*, Vol. 14, pp. 249–260, 1995.

[5] U. Manber and G.Myers. Suffix arrays: a new method for on–line string searches. *SIAM Journal of Computing*, Vol. 22, No. 5, pp. 935–948, 1993.

[6] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge Univ. Press, 1997.

[7] A. Apostolico. The myriad virtues of subword trees. In *In Combinatorial Algorithms on Words*, pp. 85–96. Springer–Verlag, 1985.

[8] K. Sadakane. A fast algorithm for making suffix arrays and for burrows–wheeler transformation. In *Proc. IEEE Data Compression Conference*, 1998.

[9] G.H.Gonnet, et al. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structure and Algorithms*, pp. 66–82. Prentice-Hall, New Jersey, 1992.

[10] 笠井透, 有村博紀, 藤野亮一, 有川節夫. 最適パタン発見に基づくテキストデータマイニング:大規模テキスト索引における高速な実装方式. 科研費『高度データベース』福井ワークショップ講演論文集, pp. 99–104, 1998.

[11] M. Nagao and S. Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrase form large text data of japanese. In *Proc of COLING '94*, pp. 611–615, 1994.

[12] 山下達雄, 松本祐治. 品詞タグ付きコーパスを直接利用した形態素解析. 言語処理学会第四回年次大会予稿集, pp. 524–527, 1998.

[13] D. E. Knuth. *Sorting and Searching, volume 3 of The Art of Computer Programming.* Addison-Wesley, 1973.

[14] K. W. Church. You shall know a word by the company it keeps. In *Proc. of NLPRS'95*, pp. 22–34, 1995.

[15] 山下達雄, 熊谷俊高, 米沢恵司, 松本祐治. Suffix Array を用いた高速文字列検索システム SUFARY, 1997. http//cactus.aist-nara.ac.jp/lab/nlt/ss.html.

[16] P. M. McIlroy and K. Bostic. Engineering radix sort. *Computing Systems*, Vol. 6, No. 1, pp. 5–27, 1993.

[17] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *the 8th Annual ACM–SIAM Sympo. on Descrete Algorithms*, pp. 360–369, 1997.

[18] K. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. In *Proc. of 4th ACM Symposium on Theory of Computing*, pp. 125–136, 1972.

[19] S. Nilsson. *Radix Sorting and Searching.* PhD thesis, Lund University, Sweden, 1996.