Suffix array の効率的な構築法

伊東秀夫†

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を用いて実装できる．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 の約 6 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 ～ 3 倍高速であることを示す．

An Efficient Method for Construction of Suffix Arrays

HIDEO ITOH†

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 about 6 times faster than the popular sorting algorithm Quicksort, and 2 to 3 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る情報を効率的に検索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として表現されるため，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．ここで文字列検索とは，既知の文字列 T に対し，検索キーとして任意の文字列 Q が与えられた場合，Q の T 中での全ての出現位置を求める問題を指す．

文字列検索において T は既知であるから，T に対して予め索引を構築し利用することで検索を高速化できる．本論文では，文字列検索に用いる索引を文字列索引と呼ぶ．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)3)]，それ以外にも幅広い応用の可能性がある [4)]．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 [5)6)7)8)9)]．

従来，文字列索引のためのデータ構造として suffix array[10)], suffix tree[11)12)], String B-tree[13),14)] などが提案されてきた．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，対象となる文字列 T 中の任意の位置から T の末尾までの範囲の文字列を指す．すなわち T の長さ（文字数）が N であれば各文字を先頭とする N 個の suffix が定義でき，その先頭文字の T 中での出現位置と 1 対 1 対応する．各 suffix とその出現位置を基に文字列索引は構成される．文字列検索時には，この索引を用いることで検索キーとなる文字列 Q に対し，Q を接頭とする全ての suffix の出現位置を高速に求めることができる．そしてこの出現位置の集合は，検索キー Q の T 中での出現位置の集合に等しい．

さて，前述した諸データ構造の内，大容量の日本語テキスト集合を対象とする文字列検索等，大規模な応用に用いる文字列索引としては suffix array が最も実用的と考えられる．すなわち suffix array は以下の特長を有する．

† リコー ソフトウェア研究所 (hideo@src.ricoh.co.jp)
Ricoh Software Research Center

- 索引のコンパクト性
  各文字が1バイトで表現され文字列TがN文字からなる場合，Tに対するsuffix arrayのサイズは4Nバイトである．これに対しsuffix treeのサイズはsuffix arrayの2.5～5倍になることをManberらは種々の文字列サンプルおよび実装法を用いた実験により示している[10]．またString B-treeは外部記憶上の索引として設計されているものの，その容量は10Nバイト以上になることから[14]，対象文字列の規模はやはり限られる．
- 字彙のサイズに依存しない計算効率
  suffix arrayには，索引の構築および検索時の記憶および時間効率が字彙（アルファベット）サイズに依存しないアルゴリズムがある．字彙サイズが大きい場合suffix arrayの方がsuffix treeよりも高速に検索できる．この性質は，日本語テキストのように字彙の多い文字列を対象とする場合，特に重要である．

一方，suffix arrayには以下の問題点がある．

- 索引の構築時間
  上述のManberらの実験ではsuffix treeに比べsuffix arrayは3～10倍の構築時間を要すると総括されている．
- 索引の更新時間
  String B-treeが動的なデータ構造であり索引更新が高速にできるのと対照的に，suffix arrayは静的データ構造であり索引更新のオーバーヘッドが大きい．

上記2つの問題点は実用面において互いに関連している．すなわち，もしsuffix arrayを非常に高速に構築できるならば，索引更新の必要が生じた際，索引を始めから作り直しても実用上の大きな障害にはならないであろう．また文献5)に示されているテキスト圧縮等，高速な構築法によりsuffix arrayの応用分野を広げることもできる．

このような背景から，我々は従来法に比べて効率的なsuffix arrayの構築法を提案する．近年の計算機のコストパフォーマンスを踏まえて，構築法の設計に当たっては，WSの1GBの主記憶上で300MBの日本語テキスト集合に対し10分程度でsuffix arrayが構築できることを目標とした．

本論文の構成を以下に述べる．2章でsuffix arrayを概説した後，3章で従来のsuffix array構築法を紹介する．4章で我々のsuffix array構築法の基本的アイデアと実装法を提示した後，その特長と改良点について述べる．5章で従来法と我々の方法を，各種コーパスに対するsuffix arrayの構築時間に関し比較評価する．最後の6章はまとめである．

2. Suffix array

suffix arrayは，ManberとMyers[10]により提案された文字列索引法である☆．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$\Sigma$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．各文字には非負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，検索キーとして入力される文字列Qに対しsuffix arrayを二分探索し，Qと接頭が一致するsuffixの集合を求めることで実現する[10]．

本稿では任意のsuffix間の辞書順を確定するためにΣに属さない文字（"$"で表す）をテキスト$T$の末尾に加える．"$"には文字値の最小値0を与える．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の添字iで指定する配列要素を$X[i]$で表し，添字iからj $(i \leq j)$の範囲に対応するXの部分を$X[i, j]$で表す．図1にテキスト"BANANA"の配列とsuffix arrayの例を示す．

テキストの配列

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

suffix array

6	5	3	1	0	4	2

図1 Suffix arrayの例
Fig. 1 An example of suffix array

☆ Oxford English Dictionaryプロジェクトにおける研究[6]，および，n-gramの効率的獲得に関する研究[8]においても関連する提案がなされている．

```
var T  : array[0..N] of char   # テキスト
var A  : array[0..N] of int    # ポインタの配列
var B1 : array[0..|Σ|] of int # バケットサイズ
var B2 : array[0..|Σ|] of int # バケット位置

# T上にテキストをセットしB1の各値を0に初期化
Initialize(T,B1)

# Step-1 配列B1をセット
for i := 0..N
  do
    B1[T[i]] := B1[T[i]] + 1
  done

# Step-2 配列B2をセット
i := 0
for a in Σ
  do
    B2[a] := i
    i := i + B1[a]
  done

# Step-3 配列Aをセット
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
Fig. 2 Initial bucket sorting algorithm

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 [15]．外部ソートによる suffix array の構築法に関する研究 [6][16] もあるが，前述した研究動機から内部ソートによる 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，従来の一般的なソートアルゴリズム [15] を用いればよい．文献 17) では Quicksort を用いた suffix array の構築例が紹介されている．しかし文字列の集合を辞書順にソートする問題に関しては，従来の諸アルゴリズムの内，MSD radix sort[15][18] と Multikey Quicksort[19] が平均的な場合について最も高速である．これら 2 つのアルゴリズムは，後の議論にも関連するので以下に概要を説明する．

MSD radix sort ではまず初期 bucket sort を行う．suffix array 構築時の初期 bucket sort のアルゴリズムを図 2 に示す．またテキスト "BANANA" に対する初期 bucket sort の例を図 3 に示す．

この処理により，テキスト中の各 suffix をその先頭 1 文字のみに関し辞書順にソートした際のポインタ列が配列 A 上に得られる．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．各 bucket はバケット表 $B1, B2$ によって管理される．即ち $B1$ には bucket のサイズが格納され，$B2$ には bucket の配列 A 上の位置が格納される．

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．

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）．

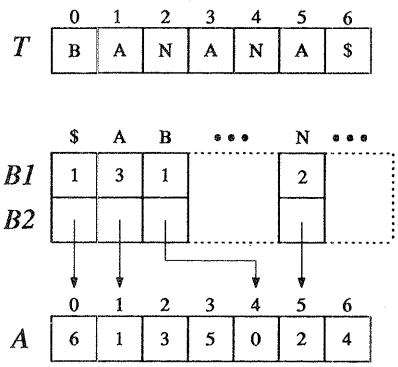

図 3 初期 bucket sort の例
Fig. 3 An example of initial bucket sorting

一方，Bentley らの Multikey Quicksort[19] は multikey（文字列のように複数要素で構成されるソートキー）を効率的に扱えるように Quicksort を拡張したものである．Quicksort ではソート対象となる要素が格納された配列を 2 分割しながら再起処理が進むのに対し，Multikey Quicksort では代表要素（pivot）との比較結果が小（<），同（=），大（>）となること

☆ 再帰処理の為には図 2 中の Step-2 の際に，以降の処理対象とすべき bucket の情報（$B1, B2$ の値，および，次の文字着目位置）を別途用意したスタック上に積んでおけばよい．また初期 bucket sort 以外では，Step-3 の代わりに permuting in place（お手玉法）[15] を用いることで，新たな作業領域を用いずに配列 A 上でのポインタの再配置ができる．

に対応して3分割しながら再起処理が進む．multikey 間の比較は，最初の3分割処理ではキーの先頭要素に着目して行われ，同（=）の部分については，この着目位置（深さ）を一つ進めて以降の処理を行う．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．

3.2 Sadakane の方法

Sadakane の方法[5]は，Manber らの方法[10]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[20)2)]をベースとし，以下のステップからなる．

(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（順位配列と仮に呼ぶ）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3分割法を用いる）．この分割結果に沿って各 suffix の順位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（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（スキップ配列と仮に呼ぶ）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

文字列ソート法が suffix のソートに固有の性質を何ら利用しないのに対し，Sadakane の順位配列や Manber らの転置配列[10]は，suffix 間の辞書順や位置関係をソートに利用するため，文字列ソート法に比べ高速になる可能性がある．特に最悪ケースの計算時間オーダーは，文字列ソート法が $O(N^2)$ に対して $O(N \lg N)$ に改善される．

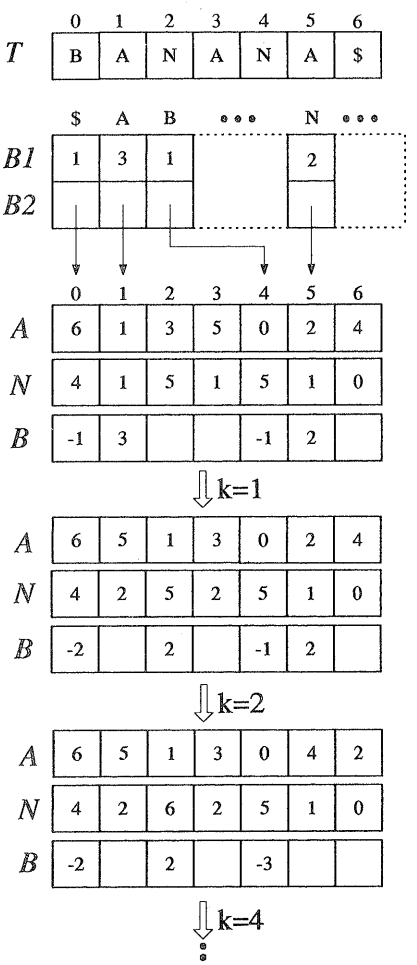

図4 Sadakane の方法の説明
Fig. 4 Sadakane's method

逆にソート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．順位配列を整数配列とし各要素に4バイトを割り当てるならば，これだけで1バイト文字からなる対象テキストの4倍の記憶量が必要になる．つまり，suffix array の最大の特長であるコンパクト性と，大規模な応用への可能性が，構築時に要請される記憶量により損なわれる場合がある．

また，日本語テキストのように，対象文字列中に様々なサイズの文字コードが混在する場合，テキスト配列の添字が，テキスト中の文字位置に単純には対応しない．よって Sadakane 法や Manber 法では，suffix array の構築後に，その要素値を補正するための後処理が必要になる．

4. 二段階ソート法

我々は隣接する suffix 間の関係のみを利用することで，文字列ソート法と同じ必要記憶量で，より高速であり，不等長文字の問題に柔軟に対応できる 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以下に説明する．

4.1 アルゴリズム

まず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 に対し，その prefix の内，長さ k 文字のものを $P(S_i,k)$ 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．接尾 S_{i-1} と S_i の関係が Type A（つまり $S_{i-1} >_1 S_i$）であれば，先頭文字 $T[i-1]$ により S_{i-1} が属する Type A の bucket を決定し，その bucket 内にポインタ値 $i-1$ をセットする．

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．

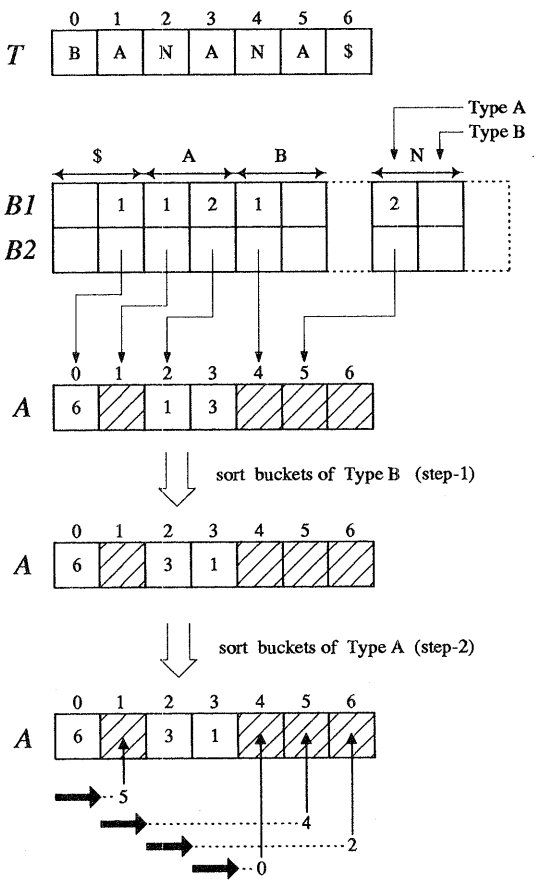

図 5 二段階ソート法の動作例
Fig. 5 Two-stage suffix sorting

4.2 アルゴリズムの妥当性

二段階ソート法の妥当性（定理1）を4つの場合（補題1～4）に分けて証明する．

定理 二段階ソート法により得たポインタ列 $A = p_0, p_1, \ldots, p_{N-1}$ について，任意の i,j $(0 \leq i,j, \leq N-1)$ に対し $i < j \iff S_{p_i} < S_{p_j}$ が成立する．

補題 1 異なるバケットに属するポインタ組 p_i, p_j $(i \neq j)$ に対して定理が成り立つ．

証明 1 *bucket* の定義より明らか．

補題 2 同一 *bucket* の異なる *Type* に属するポインタ組 p_i, p_j $(i \neq j)$ に対して定理が成り立つ．

証明 2 $i < j$ および各部分バケット間の位置関係の定義から，p_i が *Type A* に，p_j が *Type B* に属する．$S_{p_i} = a_{p_i}, a_{p_i+1}, \ldots$, $S_{p_j} = a_{p_j}, a_{p_j+1}, \ldots$ とすると，同一 *bucket* に属するから $a_{p_i} = a_{p_j}$．かつ *Type A*, *B* の定義から $a_{p_i+1} < a_{p_i} = a_{p_j} \leq a_{p_j+1}$．よって $S_{p_i} < S_{p_j}$．以上より逆も明らか．

補題 3 同じ *Type B* の部分 *bucket* 内のポインタ組 p_i, p_j $(i \neq j)$ に対して定理が成り立つ．

証明 3 文字列ソートの定義より明らか．

補題 4 同じ *Type A* の部分 *bucket* 内のポインタ組 p_i, p_j $(i \neq j)$ に対して定理が成り立つ．

証明 4 （定理の $\Longrightarrow$）$i < j$ により *step-2* において $S_{p_i+1} < S_{p_j+1}$ なる各 *suffix* へのポインタに対し p_i と p_j がセットされる．かつ p_i と p_j は同一 *bucket* に属するから $a_{p_i} = a_{p_j}$，よって $S_{p_i} < S_{p_j}$．
（定理の $\Longleftarrow$）前提の $S_{p_i} < S_{p_j}$ およびそれらの先頭文字は等しいから $S_{p_i+1} < S_{p_j+1}$．*Type A* の定義と補題 *1* から S_{p_i+1} と S_{p_j+1} への各ポインタ p_k, p_l に関し $k, l < i, j$．また，補題 *1*～*3* より $k < l$ ☆．よって *step-2* における辞書的昇順の走査時に p_i, p_j の順でセットされ $i < j$ となる．

なお補題4証明の後半から，step-2 において Type A の部分 bucket を走査する際，各要素 $A[i]$ にはポインタ p_i が既にセットされており空ではないことがわかる．

4.3 二段階ソート法の計算時間

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．計算時間のオーダーは文字列長を N として明らかに $\Theta(N)$ である．一方，第一段階のソートは文字列ソート法と同じ計算効率になる．よってソート全体の最悪時の計算オーダーは $O(N^2)$ のままであるが，Type B の bucket に属する suffix の数が Type A に比べて少ないほど，文字列ソート法に比べ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．

4.4 さらなる高速化

英語，音素ラベル，ゲノム（ATCG）など，1バイト文字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．

そしてこの時，第 4.1 節での記号定義 $>_k$ における k の単位を文字からバイトに変えて，Type A と Type B の分類条件を以下のように拡張する．

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 の先頭2文字が共に1バイト文字 a_i, a_j である場合，$a_i \leq_1 a_{i+1}$ により Type B になってしまう場合でも，さらに S_{i+2} との比較（つまり $a_i, a_{i+1} >_2 a_{i+2}a_{i+3}$）により，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 文字），ゲノム（ATCG の 4 文字）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とんど影響されない．

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
Table 1 Ratios of Type B suffixes

なお，Type B の割合をなるべく小さくするように文字値の交換を行うと，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，文字値の交換による改善はわずかだった．

☆ p_k, p_l が同じ Type A の部分 bucket に属する場合，同様の議論を $S_{p_i+m} < S_{p_j+m} (m = 2, 3, \ldots)$ について適用すれば，補題 1 ～ 3 の場合に帰着する．

前述のように Type A と Type B の分類を拡張した場合の step-2 をC言語風に示す．実際には，以下に2バイト文字に関する場合分けを加える必要がある．

```
// suffix array を昇順に走査
for(n=0; n<N; n++) {
   p0 = text + suff[n]; // 注目位置
   p1 = p0 - 1;         // 注目位置の1つ前
   p2 = p0 - 2;         // 注目位置の2つ前
   if( *p1 > *p0 ) {
     *(bucket[*p1][*p0].A++) = suff[n]-1
   }
   if( *p2 <= *p1 && memcmp(p2,p0,2) > 0 ) {
     *(bucket[*p2][*p1].A++) = suff[n]-2;
   }
}
```

4.5 文字列ソート法の改良

二段階ソート法の step-1 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 [15]，この方法はバケット表のサイズが前述のように 64K の場合，あまり効果がない．

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．

4.6 二段階ソート法の記憶量

テキストが N バイトからなるとし，suffix array の配列要素を 4 バイトで表現するものとする．英語および日本語テキストをバイト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☆．

アルゴリズム	記憶量（英）	記憶量（日）
文字列ソート法	$5N$	$3N$
Manber–Myers	$8N$	$4N$
Sadakane	$9N$	$5N$
Larsson & Sadakane[21]	$8N$	$4N$
二段階ソート法	$5N$	$3N$

表2 ソートに必要な記憶量（byte）
Table 2 Memory requirement

☆ バケット表と再起のためのスタックに要する記憶量は，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）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2バイトとした．RS は最も高速な実装法として知られる McIlroy らの方法 [18] に従った．

Sadakane の方法（SS）は最近 WWW 上で公開されたソースコード [21] を用いた．前節 3.2 で紹介した内容に以下の2点の改良が加えられている．

- 記憶効率の向上
  テキスト長を N として必要記憶量が 9N から 8N になった．
- 文字コード変換
  文字コードを連続値に変換（コンパクト化）することで初期バケットソートの効果を高めた．

ただし日本語テキストに対しては前節 3.3 で述べた構築結果を補正するための後処理を加え，他のアルゴリズムと同一内容が得られるようにした．

二段階ソート法は 64K のバケット表を用い文字値交換は行っていない．各実装において mmap 等の特別な仮想記憶管理は行っていない．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．英語テキストはバイト単位に，日本語テキスト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 テキストでありバイト単位に索引づけした．

データの特徴付けとして AML（Average Matching Length）[5] 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．

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

data	size(byte)	AML	QS	MQ	RS	SS	ours
英語 031MB	30935873	21	411	176	163	150	59
英語 060MB	60172753	29	865	385	363	343	128
英語 100MB	104857600	31	16031	713	631	765	228
英語 180MB	179540613	29	3083	1334	1325	–	444
毎日 051MB	51129551	31	345	159	132	130	67
毎日 100MB	104857600	22	760	349	305	280	143
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)
Table 3 Sorting time on large texts (sec)

data	size(byte)	AML	QS	MQ	RS	SS	ours
book1	768771	7	4.91	1.55	1.13	2.03	0.74
progc	39611	8	0.17	0.05	0.04	0.04	0.04
book2	610856	10	3.65	1.11	0.86	1.36	0.53
bible	4047392	14	38.99	14.99	13.08	14.56	5.57
E.coli	4638690	17	49.91	19.25	18.39	13.99	7.94
news	377109	18	2.17	0.67	0.53	0.70	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)
Table 4 Sorting time on small texts (sec)

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．

5.2.1 大規模データでの比較

いずれのデータにおいても，二段階ソート法が最も高速である．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 に対しては英語で約 3 倍，日本語で約 2 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 法は日本語, 英語テキストについて他の文字列ソート法とほぼ同等の処理速度となった．

5.2.2 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，テキスト圧縮等，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)，ソート初期には各 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．旧版に比べて文献 [21] の Sadakane 法は文字値変換と初期バケットソートの効率化によりゲノム列に対する性能が向上している．

5.2.3 AML が大きいデータに関する実験

前述したように Sadakane 法は最悪時の計算オーダーにおいて二段階ソート法より優れている．そして対象文字列中の反復が多いほど最悪のケースに近づく．反復により ML は 2 乗のオーダで増加するため AML も非常に大きくなる．文献 [21] のデータでは html ファイルの集合の AML が数百に達している．このようなデータに対して doubling 法を用いず，文字列ソート法のみをベースとする二段階ソート法は計算量が 2 乗のオーダになりソート時間が非常に悪化する．

しかし，自然言語テキストの検索を応用とする場合，入力される検索キーの長さに上限 (M 文字) を設け，suffix のソートを先頭から最大 M 文字に制限してもよい状況もある．この場合，MSD Radix sort を step–1 に用いる二段階ソート法の最悪時のオーダーは $\Theta(N)$ になる．そこで 128KB の英文を 10 回連接して AML が非常に大きいテキストデータを作成し，ソートの制限長とソート時間の関係を調べてみた．AML は 58983 で最長の ML は 131072 バイトである．図 6 に結果を示す．この結果によると，制限長が約 200

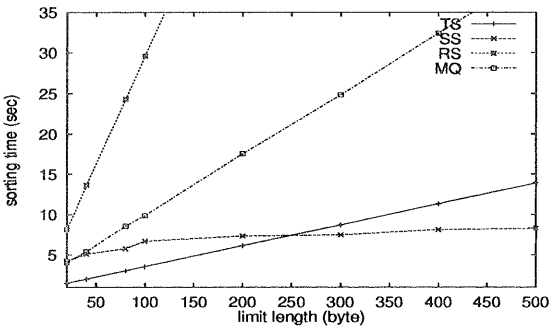

図 6 高 AML テキストに対するソート制限長とソート時間 (sec)
Fig. 6 Sorting time (sec) and limit length (byte)

バイトまでならば，二段階ソート法 (TS) は Sadakane 法よりも高速である．この制限長は検索での質問長を考えると実用上十分な長さである．Sadakane 法は制限長を長くするに従って対数オーダでソート時間が増加する．他の文字列ソート法 (RS,MQ) は非常に急激にソート時間が悪化している．

6. おわりに

大規模テキストを対象とする文字列索引として最も実用的と考えられる suffix array に着目し，高速な構築法 (二段階ソート法) を提案した．二段階ソート法の必要記憶量は文字列ソート法と同等であり Sadakane

や Manber らの方法に比べ suffix array のコンパクト性を生かせる．

また Farach の suffix tree 構築法[12]は出現位置の奇偶で suffix を分類し，奇数 suffix tree と偶数 suffix tree を構築後マージする．この手法と二段階ソート法の関連性を査読者の方から御指摘して頂いた．今後の参考としたい．

謝辞 御指導頂いている東京工業大学田中穂積教授，並びに社内研究所の皆様に深謝致します．また査読者の方々から的確なご指摘を頂きました．

参 考 文 献

1) Stephen, G. A.: *String Searching Algorithms*, World Scientific Publishing (1994).
2) Crochemore, M. and Rytter, W.: *Text Algorithms*, Oxford Univ. Press, New York (1994).
3) Gusfield, D.: *Algorithms on Strings, Trees, and Sequences*, Cambridge Univ. Press (1997).
4) Apostolico, A.: The myriad virtues of subword trees, *In Combinatorial Algorithms on Words*, Springer–Verlag, pp. 85–96 (1985).
5) Sadakane, K.: A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation, *Proc. IEEE Data Compression Conference*, pp. 129–138 (1998).
6) Gonnet, G.H. et al.: New indices for text: PAT trees and PAT arrays, *Information Retrieval: Data Structure and Algorithms* (Frakes, W. B. and Baeza-Yates, R. A.(eds.)), Prentice-Hall, New Jersey, pp. 66–82 (1992).
7) 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, 夏のＤＢワークショップ'98 in 福井, SIGDBS-116-20, pp. 151–156 (1998).
8) Nagao, M. and Mori, S.: 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, *Proc of COLING '94*, pp. 611–615 (1994).
9) 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).
10) Manber, U. and Myers, G.: Suffix arrays: a new method for on–line string searches, *SIAM Journal of Computing*, Vol. 22, No. 5, pp. 935–948 (1993).
11) Ukkonen, E.: On–line construction of suffix–trees, *Algorithmica*, Vol.14, pp.249–260 (1995).
12) Farach, M.: Optimal Suffix Tree Construction with Large Alphabets, *Proc. of 38th Symp. on Foundations of Computer Science*, pp. 137–143 (1997).
13) Ferragina, P. and Grossi, R.: The String B-Tree: a new data structure for string search in external memory and its applications, *Journal of the ACM*, Vol. 46, No. 2, pp. 236–280 (1999).
14) Ferragina, P. and Grossi, R.: A fully–dynamic data structure for external substring search, *ACM Symp. Theory of Computing*, pp.693–702 (1995).
15) Knuth, D. E.: *Sorting and Searching, volume 3 of The Art of Computer Programming*, Addison-Wesley (1973).
16) Arge, L., Ferragina, P., Grossi, R. and Vitter, J. S.: On sorting strings in external memory, *ACM Symp. Theory of Computing*, pp.540–548 (1997).
17) Church, K. W.: You shall know a word by the company it keeps, *Proc. of NLPRS'95*, pp. 22–34 (1995).
18) McIlroy, P. M. and Bostic, K.: Engineering Radix Sort, *Computing Systems*, Vol. 6, No. 1, pp. 5–27 (1993).
19) Bentley, J. L. and Sedgewick, R.: Fast algorithms for sorting and searching strings, *the 8th Annual ACM–SIAM Sympo. on Descrete Algorithms*, pp. 360–369 (1997).
20) Karp, K. M., Miller, R. E. and Rosenberg, A. L.: Rapid identification of repeated patterns in strings, arrays and trees, *Proc. of 4th ACM Symposium on Theory of Computing*, pp. 125–136 (1972).
21) Larsson, N. J. and Sadakane, K.: Faster Suffix Sorting, Technical Report LU-CS-TR:99-214, Lund University, Sweden (1999). http//www.cs.lth.se/home/Jesper_Larsson/.

(平成 11 年 9 月 20 日受付)
(平成 11 年 12 月 27 日採録)

(担当編集委員 加藤 和彦)

伊東 秀夫（正会員）

1985 年慶応義塾大学数理科学科卒業．同年（株）リコー入社．自然言語処理に関する研究開発に従事．現在，同社ソフトウェア研究所に勤務．1996 年より東京工業大学情報理工学研究科の博士後期課程に在学（計算工学専攻・田中穂積研究室）．情報処理学会，言語処理学会，各会員．