Suffix Arrayの効率的な構築法

An Efficient Method for Constructing Suffix Arrays

伊東　秀夫[*]
Hideo Itoh

要　　旨

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になっている．そこで任意のsuffix間の関係ではなく，高速かつコンパクトなsuffix array構築法を提案した．このアルゴリズムは隣接するsuffix間の関係のみを使用する利点があり，二段階ソート法と呼ばれる．アルゴリズムはQuicksortの4.5～6.9倍高速であり，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，今まででで最も高速なアルゴリズムとして知られているSadakaneの方法に対しても2.5～3.6倍高速である．

ABSTRACT

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. An efficient algorithm for sorting suffixes is proposed based on the specific relationships between an adjacent suffix pair, called the Two--Stage Suffix Sort. The experiments on several text data sets (including 514MB Japanese newspaper) 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 algorithm which is known as the fastest one.

* 画像システム事業本部 ソフトウエア研究所
Software Research Center, Imaging System Business Group.

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中の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．文字列の索引を表現するデータ構造として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，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.

- 索引のサイズが最もコンパクト
- 字彙のサイズに依存しない計算量
- 高速な検索が可能

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であり，対象文字列中の各suffixの開始位置(ポインタと呼ぶ)を格納した配列である．ただしポインタはそれに対するsuffixが辞書順になるように配列上に並べる．こうすることで検索キーとなる任意の文字列の全出現位置はsuffix array上のある連続区間に並ぶことになる．この連続区間は2分探索により高速に同定できる．

例としてFig.1にテキスト``BANANA''に対するsuffix arrayの例を示す．対象文字列のアルファベットにない仮想文字(``$''で表す)を末尾に加えることで辞書順を確定させる．

ポインタ	suffix
0	B A N A N A $
1	A N A N A $
2	N A N A $
3	A N A $
4	N A $
5	A $
6	$

辞書順ソート

ポインタ	suffix
6	$
5	A $
3	A N A $
1	A N A N A $
0	B A N A N A $
4	N A $
2	N A N A $

suffix array

5	3	1	0	4	2

Fig.1　An example of suffix array.

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，Fig.2に示す初期bucket sortを行う．テキスト``BANANA''に対する初期bucket sortの例をFig.3に例示する．同じ先頭文字を持つsuffixへのポインタは配列 $A$ 上の或る連続区間に並べられる．この区間をbucketと呼ぶ．Fig.3で文字``N''に対するbucketは $A[5,6]$ で，そのサイズ(つまり $B1[N]$ の値)は2である．

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

# Step-0 : initialize of bucket table
for  i := a  in     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
 do
   B2[a] := c
   c := c + B1[a]   done

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

Fig.2　Initial bucket sorting algorithm.

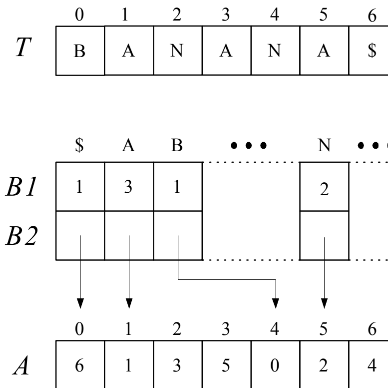

Fig.3　An example of initial bucket sorting.

初期bucket sort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ポインタ列が配列 A 上にセットされる．

次にMSD radix sortは，サイズが2以上のbucketについて，bucket内のポインタが指す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．

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．

3-2　Sadakaneの方法

Sadakaneの方法[8]はManber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．Fig.4に概要を示す．KMRアルゴリズム[18]がベースであり次の処理ステップからなる．

(step-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_1 に対し $N[\mathrm{i}] = \mathrm{n}$ とする．例えばFig.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．

(step-2)　bucket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_1 を $N_{\mathrm{i+k}}$ をキーとしてソートし，$N_{\mathrm{i+k}}$ 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Fig.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．

(step-3)　doubling法の適用

全bucketのサイズが1ならば終了，そうでなければ $\mathrm{k} = \mathrm{k} \times 2$ としてstep-2を行う．

ただし上記step-2はサイズが2以上のbucketのみについて行えばよい．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を記録し利用する．Fig.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，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4byte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．

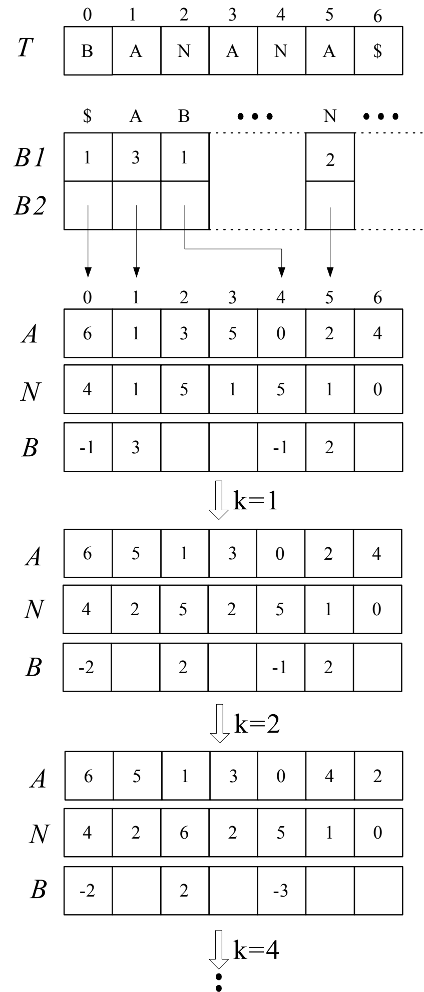

Fig.4　Sadakane's method.

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 $\mathrm{k}(>0)$ 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] $S_i >_1 S_{i+1}$ を満たす S_i のbucket

TypeB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]TypeBのsuffixに関するソート

サイズが2以上のTypeB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]TypeA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TypeAであれば， $i-1$ を文字 $T[i-1]$ に対するTypeAのbucketの領域にセットする．

以上の処理ステップを図Fig.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TypeB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TypeA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TypeBのbucketに属するsuffixへのポインタのみである．よって図中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TypeB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，TypeA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TypeA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．TypeA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．

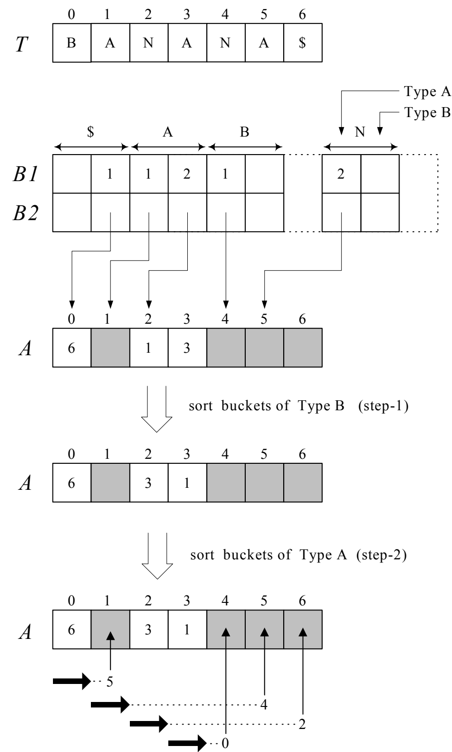

Fig.5 Two-stage suffix sorting.

4-2 二段階ソート法の高速性

第2段階のTypeAのソートは，suffix arrayをleft to rightに一回走査し，決定的にTypeA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，TypeBのbucketに属するsuffixの数がTypeAに比べて少ないほど，高速になると考えられる．

仮にTypeAとTypeB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，TypeAとTypeB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TypeBになってしまう場合でも，さらに S_{i+2} との比較で，TypeAになるチャンスが生まれるため，全体に占めるTypeBの割合を小さくすることができる．

Table 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TypeBの割合(実測値)を示す．ratio-1は前頁で述べた $>_1$ のみの比較による場合，ratio-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ず一定である．

Table 1　Ratios of Type　B　suffixes.

data	ratio-1	ratio-2
日本語	51%	同左
英　語	51%	35%
ゲノム	62%	40%

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(Fig.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．

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 byteからなるとしsuffix arrayの配列要素を4 byte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Table 2にまとめる．

Table 2　Memory requirement.

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

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2byte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．

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およびCanterburyコーパスを用いた．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．

Table 3　Sorting time on large texts(sec)

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

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

Table 3とTable 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．

(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Table 1のratio-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である．

(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，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．またゲノム列は文字種が少ないため(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．

Table 4　Sorting time on small texts (sec)

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

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う予定である．

参考文献

[1] G.A.Stephen: "String Searching Algorithms", World Scientific Publishing (1994).

[2] M.Crochemore, W.Rytter: "Text Algorithms", Oxford University Press (1994).

[3] J.Blumer, A.Blumer, D.Haussler: "The smallest automaton recognizing the subwords of a text", Theoretical Computer Science, 40巻 (1985), pp. 31-55.

[4] E.Ukkonen: "On-line construction of suffix-trees", Algorithmica, 14巻 (1995), pp. 249-260.

[5] U.Manber, G.Myers: "Suffix arrays: a new method for on-line string searches", SIAM Journal of Computing, 22巻, 5号(1993), pp. 935-948.

[6] D.Gusfield: "Algorithms on Strings, Trees, and Sequences", Cambridge University Press, (1997).

[7] A.Apostolico: "The myriad virtues of subword trees", in Combinatorial Algorithms on Words, Springer-Verlag (1985), pp. 85-96.

[8] K.Sadakane: "A fast algorithm for making suffix arrays and for burrows-wheeler transformation", Proc. IEEE Data Compression Conference (1998), pp. 129-138.

[9] G.H.Gonnet, et al.: "New indices for text: PAT trees and PAT arrays", Information Retrieval: Data Structure and Algorithms, Prentice-Hall (1992), pp. 66-82.

[10] 笠井, その他: "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",『高度データベース』福井ワークショップ講演論文集(1998), pp. 99-104.

[11] M.Nagao, 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", Proc. COLING'94 (1994), pp. 611-615.

[12] 山下, 松本: "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(1998), pp. 524-527.

[13] D.E.Knuth: "Sorting and Searching", The Art of Computer Programming [3], Addison-Wesley (1973).

[14] K.W.Church: "You shall know a word by the company it keeps", Proc. NLPRS'95 (1995), pp. 22-34.

[15] 山下, その他: "Suffix Arrayを用いた高速文字列検索システム SUFARY", http//cactus.aist-nara.ac.jp/lab/nlt/ss.html

[16] P.M.McIlroy,K.Bostic: "Engineering radix sort", Computing Systems, 6巻, 1号(1993), pp. 5-27.

[17] J.L.Bentley, R.Sedgewick: "Fast algorithms for sorting and searching strings", Proc. the 8th Annual ACM SIAM Symposium on Descrete Algorithms (1997), pp.360-369.

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

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