データベースシステム 122－7
(2000. 7. 26)

圧縮接尾辞配列に関する考察

定兼 邦彦

東北大学大学院情報科学研究科システム情報科学専攻
〒980-8577 仙台市青葉区片平 2-1-1
sada@dais.is.tohoku.ac.jp

あらまし　全文検索のための索引である接尾辞配列は，他の全文検索索引と比較すると省スペースであるが，転置ファイルのような単語索引と比較するとサイズが大きい．この問題を解決するために圧縮接尾辞配列が提案されたが，検索にはテキスト自身も必要であるため，索引サイズはテキストよりも小さくならない．

本稿では圧縮接尾辞配列を用いた検索アルゴリズムを，テキスト自身が不要になるように変更する．また，テキスト全体やその一部を圧縮接尾辞配列から復元するアルゴリズムを提案する．これにより，テキストの圧縮と高速な検索の両立が可能となる．大規模な英文，和文，DNA 配列に対する実験により，検索文字列の出現数が少ない場合に転置ファイルや逐次検索よりも有効であることがわかった．

キーワード　全文検索，接尾辞配列，圧縮

A Note on the Compressed Suffix Arrays

Kunihiko Sadakane

Department of System Information Sciences
Graduate School of Information Sciences, Tohoku University
Katahira 2-1-1, Aoba-ku, Sendai 980-8577, Japan
sada@dais.is.tohoku.ac.jp

Abstract　The suffix array, a kind of full-text indices, is more space-efficient than other full-text indices; however it is still larger than word indices such as the inverted files. Though the recently proposed compressed suffix array solved the problem in part, its index size is always larger than the text because it uses the text itself for keyword queries.

In this paper we extend the text search algorithm using the compressed suffix array in order to work without the text itself. We also propose an algorithm to recover the whole or in part of the text. By using the algorithms both text comression and fast queries can be achieved. Experimental results on a large collection of English and Japanese texts and DNA sequences show that the compressed suffix array is more effective than using the inverted files or sequential searches if the number of occurrences of keywords is not too large.

key words　full-text index, suffix array, compression

1 はじめに

現在多くのテキストが蓄積され, それらからの高速な検索や高度な検索が必要となっている. 古典的な方法ではテキスト全体を走査して文字列を検索するが, 現在では高速な検索を実現するためにあらかじめ索引を作っておくことが一般的である.

検索のための索引には単語索引 (word index) と全文索引 (full-text index) の 2 種類がある. 前者の代表は転置ファイルであり, 後者の代表は接尾辞木 [9] や接尾辞配列 [8] である. 一般に, 単語索引の方がサイズが小さい, 検索が高速, 索引の構成が簡単などの利点があるが, テキストを単語に区切ってから索引を作るため, 任意の文字列を検索することが難しいという欠点がある. これは日本語のように区切り方が難しい場合や DNA 配列のように単語に区切ることができないテキストの場合に問題になる. 一方, 全文索引は任意の文字列を検索することができるが, 索引のサイズが大きくなるためあまり普及していない.

全文索引のサイズを小さくする研究はいくつも行われている [7, 3, 2, 5]. 本稿ではこの中の圧縮接尾辞配列 (compressed suffix array) (Grossi, Vitter [3]) を扱う. これは長さ n のテキストに対する接尾辞配列を $n \log n$ ビットから $O(n)$ ビットに圧縮するものである. 検索速度は圧縮前より少し遅くなる. しかしこの論文では実際のサイズや検索時間までは考察されていない.

索引を作り検索を高速にするのではなく, テキストを圧縮することでサイズを小さくし, 検索を高速にする方法も提案されている [1, 6]. しかしこの方法では検索時に圧縮されたテキスト全体を見る必要があるため, 大量のテキストに対しては検索が遅くなる.

テキストの圧縮と高速な検索を両立させる方法 (Ferragina, Manzini [2]) も提案されているが, サイズがアルファベットサイズに大きく依存するため, そのままでは現実的ではない. そこで本稿では圧縮接尾辞配列を拡張してこの両立をめざす. Ferragina, Manzini の方法では圧縮されたテキストに, 検索のための索引を加えているが, 本稿の方法は検索のための索引である圧縮接尾辞配列からテキストを復元するというものである. つまり, Grossi と Vitter の圧縮接尾辞配列は文字列の検索にテキスト自身を必要とするが, 本稿で拡張した索引ではテキストを必要としない. よってテキストのサイズよりも小さい索引で高速な検索とテキストの復元が可能になる可能性がある.

検索で必要なのは指定した単語の出現した位置を求めることと, その単語を含むテキストを得ることである. よって本稿では圧縮接尾辞配列に対する 3 つの新しい操作 *inverse, search, decompress* を定義する. *inverse* は接尾辞配列の逆関数であり, さまざまな応用がある. *search* はパタン P を含む接尾辞配列の区間を返す. *decompress* はテキストの一部分を復号する.

本稿ではこの索引を実装し, サイズと検索速度を転置ファイルと比較する. 転置ファイルは通常はテキストから切り出した単語しか検索できないが, 任意の文字列を検索するアルゴリズムも実装しその速度を実験する. その結果, DNA のように単語に区切ることができないテキストに関しては単語の出現数が少ない場合に圧縮接尾辞配列の方が高速になることがわかった. 索引サイズに関してはパラメタを選ぶことで元のテキストより小さくなることがわかった.

2 接尾辞配列とその圧縮法

2.1 接尾辞配列

長さ n のテキスト $T[1..n]$ に対する接尾辞配列 (suffix array) [8] とは, T の各接尾辞 $T[j..n]$ $(1 \leq j \leq n)$ の添字 j を, 接尾辞の辞書順でソートした配列 $SA[1..n]$ のことである. $SA[i] = j$ は 接尾辞 $T[j..n]$ が辞書順で i 番目であることを表す. テキストとその接尾辞配列を用いれば, 任意のパタン $P[1..m]$ の T 中での出現回数が二分探索で $O(m \log n)$ 時間で求まり, それらの出現位置は出現回数 occ に比例する時間で求めることができる. 図 1 は長さ 16 のテキスト T に対する接尾辞配列 SA の例である.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	e	b	d	e	b	d	d	a	d	d	e	b	e	b	d	c

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11

図 1: 接尾辞配列

文字を 1 バイト, 接尾辞の添字を 4 バイトで表すとすると, 長さ n のテキストとその接尾辞配列のサイズの合計は $5n$ バイトとなり, そのサイズが問題となる. 理論的には, サイズは $O(n \log n)$ ビットであり, $O(n)$ ビットである Lempel-Ziv index [5] よりも大きい. ただし Lempel-Ziv index は検索できる文字列の長さに上限があるため, 任意の文字列を高速に検索でき, 漸近的なサイズがテキストと等しい $O(n)$ ビットの索引が望まれている.

2.2 圧縮接尾辞配列

圧縮接尾辞配列 (compressed suffix array) [3] は, 接尾辞配列のサイズを $O(n)$ ビットにするものである. ただし配列の要素にアクセスする時間は $O(1)$ から $O(\log^{\epsilon} n)$ (ϵ は任意の正定数) となる. 基本的な考えは, 接尾辞をその添字の偶奇で分割し, 偶数の接尾辞は長さが $n/2$ の文字列に対する接尾辞配列で表し, 奇数の接尾辞はそれより 1 つ短い偶数の接尾辞へのポインタで表すというものである. 偶数接尾辞に関してはこれを再帰的に行う. 図 2 は上のテキスト T に対応する圧縮接尾辞配列である. SA_0 は圧縮前の SA である. B_k は 0,1 のビットベ

```
     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
  T  e  b  d  e  b  d  d  a  d  d  e  b  e  b  d  c

     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
SA0  8 14  5  2 12 16  7 15  6  9  3 10 13  4  1 11
B0   1  1  0  1  1  1  0  0  1  0  0  1  0  1  0  0
psi0       9              1  6    12 14     2     4  5

     1  2  3  4  5  6  7  8            1  2  3  4  5  6  7  8
SA1  4  7  1  6  8  3  5  2         T1 bd eb dd ad de be bd c
B1   1  0  0  0  1  0  0  0
psi1    5  8  2     1  4  6

                        1  2  3  4      1    2    3    4
                   SA2  2  3  4  1  T2 ebdd adde bebd c

     1  2                1        2
SA3  1  2           T3 addebebd c
```

図 2: 圧縮接尾辞配列

クトルで, $B_k[i]=1$ は $SA_k[i]$ が偶数接尾辞であることを表している. $\Psi_k[i]$ は奇数接尾辞に対して定義され,

$$SA_k[\Psi_k[i]] = SA_k[i]+1$$

となっている. SA_{k+1} は SA_k の偶数接尾辞の添字を 2 で割ったものを格納している. 以上より, $SA_k[i]$ は次のように表される.

$$SA_k[i] = \begin{cases} 2SA_{k+1}[rank(B_k,i)] & \text{if } B_k[i]=1 \\ SA_k[\Psi_k[i]]-1 & \text{if } B_k[i]=0 \end{cases}$$

Ψ は Ψ_0 を表すとする.

$rank(B_k,i)$ は $B_k[1..i]$ の中の 1 の数を返す関数であり, $O(n\log\log n/\log n)$ ビットのディレクトリと呼ばれる表を用いることで定数時間となる [4]. k を圧縮接尾辞配列のレベルと呼ぶ. レベル k には長さ $n_k = n/2^k$ のテキスト T_k に対応する接尾辞配列を格納する. ただし SA_k ではなくよりコンパクトな B_k と Ψ_k のみを格納する. k の範囲は $0 \le k < l = \lceil \log\log n \rceil$ である. レベル l では SA_l のみを格納する. そのサイズは $\log n \cdot n/2^l = n$ ビットである.

実際には全てのレベルは使われない. 図 2 ではレベル 2 は使われていない. この場合 $B_1[i]$ は 接尾辞 $SA_1[i]$ の添字が 4 の倍数かどうかを表している. また Ψ_1 は 4 の倍数以外の接尾辞に対して定義される. レベル $k+1$ $(0 \le k < l)$ で D の倍数の接尾辞のみを格納している場合, $SA_k[i]$ を求めるアルゴリズムは次のようになる.

Algorithm $SA_k[i]$

1. $v \leftarrow 0$;
2. **while** $B_k[i]=0$
3. **do** $i \leftarrow \Psi_k[i]$;
4. $v \leftarrow v+1$;
5. **return** $D \cdot SA_{k+1}[rank(B_k,i)] - v$.

接尾辞配列の要素 $SA[i]$ を得る時間は, 用いるレベルの数とレベル内での Ψ を使う回数の積に比例する. レベル 0, $l' = \lceil \frac{1}{2}\log\log n \rceil$, $l = \lceil \log\log n \rceil$ の 3 つを用いる場合, 1 つのレベル内では Ψ が $D=\sqrt{\log n}$ 回使われる. よってアクセス時間は $O(\sqrt{\log n})$ 時間となる. $1/\epsilon$ 個のレベルを用いれば, アクセス時間は $O(\log^\epsilon n)$ 時間となる. 圧縮接尾辞配列のサイズは, 各レベルが $O(n)$ ビットであるため, 全体でも $O(n)$ ビットとなる.

圧縮接尾辞配列を用いると, パタン P の出現回数 occ を求める時間は $O(|P|\log^{1+\epsilon} n)$ となる. その後, 出現位置の列挙は $O(occ \log^\epsilon n)$ 時間でできる.

3 圧縮接尾辞配列の拡張

本節では圧縮接尾辞配列を拡張する. まず, パタン P の出現回数の計算にはテキスト T が必要なく, $O(|P|\log n)$ 時間でできることを示す. 次に, 接尾辞配列の逆関数 $SA^{-1}[j]$ が $O(\log^\epsilon n)$ 時間で求まることを示す. さらに, テキスト中の長さ l の任意の部分が圧縮接尾辞配列から $O(l+\log^\epsilon n)$ 時間で復元できることを示す.

以上の手続きを実現するためのデータ構造の変更点は以下の通りである. まず, $\Psi_k[i]$ を全ての i $(1 \le i \le n_k)$ に対して定義する. このようにしてもサイズは高々 $\frac{\log^\epsilon n}{\log^\epsilon n-1}$ 倍にしかならない. また, レベル l では SA_l に加えてその逆関数 SA_l^{-1} も格納する. 逆関数を表す配列のサイズは $\log n \cdot n/2^l = n$ ビットである. また, *rank* の逆関数である *select* のためのディレクトリも必要となる. そのサイズは $O(n\log\log n/\log n)$ ビットである. さらに, レベル 0 ではテキスト T 中の文字の出現頻度を格納する必要がある.

3.1 テキストを用いない二分探索

配列 $C[1..|\Sigma|]$ を, テキスト T 中の文字の累積頻度表とする. つまり, $C[c]$ は文字 $1,\ldots,c$ の出現頻度の合計となる. その逆関数 $C^{-1}[i]$ は T 中の文字をアルファベット順に並べたときの i 番目の文字を表す. この関数を用いると次の命題が成り立つ.

命題 1

$$T[SA[m]+i] = C^{-1}[\Psi^i[m]]$$

Proof: 接尾辞 $T[SA[m]..n]$ は辞書順にソートされているため, 先頭の 1 文字 $T[SA[m]]$ を取り出すとそれらはアルファベット順に並んでいる. よって $T[SA[m]] = C^{-1}[m]$ となる. $T[SA[m]+i]$ を求めるには接尾辞 $T[SA[m]+i..n]$ の辞書順の順位 j $(SA[j]=SA[m]+i)$ が必要となるが, $SA[\Psi[m]] = SA[m]+1$ を繰り返し適用することで $j=\Psi^i[m]$ と計算される.

この命題は, 辞書順で m 番目の接尾辞を得る際に $SA[m]$ の実際の値を求める必要がないことを表している. さらに, テキスト T も必要がないことを表している. 圧縮接尾辞配列では $SA[m]$ の値を求めるのに $O(\log^\epsilon n)$ 時間かかるが, この命題により, 圧縮接尾辞配列上での二分探索を用いたパタン P の出現回数の計算が通常の接尾辞配列と同じ計算量で行えることがわかる.

定理 1 長さ n のテキストの圧縮接尾辞配列を用いた，パタン P の出現回数は $O(|P|\log n)$ 時間で計算できる．

3.2 接尾辞配列の逆関数

接尾辞配列の逆関数 $SA^{-1}[j]$ とは，$SA[i]=j$ となる i を返す関数である．これは $SA[i]$ を求めるときとは逆にレベル l から 0 の順に求めていく．レベル l では接尾辞配列 SA_l の逆関数を表す配列 SA_l^{-1} を用いる．D の倍数ではない接尾辞に対してはその辞書順の順位は Φ_k を使うことで求まる．$SA_k[i]$ から SA_{k-1} の対応する要素への移動は *rank* の逆関数である *select* を用いる．なお，各レベル k では D の倍数の接尾辞に対する辞書順の順位しか格納していないため，接尾辞 $T_k[1..n_k]$ から $T_k[D-1..n_k]$ に対してはそれより左の接尾辞から Ψ_k をたどることができない．よって各レベル k で $SA_k[i]=1$ となる i を pos_k^1 として覚えておき，そこから Ψ_k をたどるようにする．アルゴリズムは次のようになる．

Algorithm *inverse*(j)
1. **for** $k \leftarrow 0$ **to** $l-1$
2. **do** $r[k] \leftarrow j \bmod D$;
3. $q[k] \leftarrow j/D$;
4. $j \leftarrow j/D$;
5. $i \leftarrow SA_l^{-1}[j]$;
6. **for** $k \leftarrow l-1$ **downto** 0
7. **do if** $q[k]=0$
8. **then** $i \leftarrow pos_k^1$;
9. **for** $d \leftarrow 1$ **to** $r[k]-1$
10. **do** $i \leftarrow \Psi_k[i]$;
11. **else** $i \leftarrow select(B_k, i)$;
12. **for** $d \leftarrow 0$ **to** $r[k]-1$
13. **do** $i \leftarrow \Psi_k[i]$;
14. **return** i.

計算量に関しては，同一レベル内での Φ_k を用いる回数が高々 $D=\log^\epsilon n$ 回，レベル数が $1/\epsilon$ 個であるため，次の定理が成り立つ．

定理 2 圧縮接尾辞配列の逆関数 $SA^{-1}[j]$ は任意の正定数 ϵ に対して $O(\log^\epsilon n)$ 時間で求まる．

3.3 テキストと接尾辞配列の部分的な復元

本稿で拡張した圧縮接尾辞配列では検索にはテキストを必要としないが，テキストデータベースとして使用する場合には検索したパタンの前後のテキストを表示する必要がある．これは逆関数 SA^{-1} と命題 1 を用いるだけである．

定理 3 長さ n のテキストの圧縮接尾辞配列からの長さ l の部分文字列の復号は $O(l+\log^\epsilon n)$ 時間で行える．

図 3 は $T[9..13]=$ ddebe を復号する例である．まず接尾辞 $T[9..16]$ の辞書順の順位を逆関数 $SA^{-1}[9]=10$ から求める．次に接尾辞配列の要素を Ψ 関数に従いたどりながら，テキスト中の文字を C^{-1} を用いて復号する．

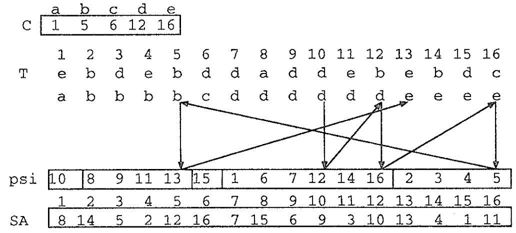

図 3: テキストの部分的な復号

また，テキストの一部分 $T[s..e]$ の中の接尾辞に対する接尾辞配列も同様に復元できる．アルゴリズムは次のようになる．

Algorithm *extractSA*(s,e)
1. $i \leftarrow inverse(s)$;
2. **for** $j \leftarrow s$ **to** e
3. **do** $J[j] \leftarrow i$;
4. $i \leftarrow \Psi[i]$;
5. **for** $j \leftarrow s$ **to** e
6. **do** $I[j] \leftarrow j$;
7. Sort $I[j]$ $(s \le j \le e)$ in order of $J[j]$ by using radix sort.
8. **return** $I[s..e]$.

$SA^{-1}[s]$ は $O(\log^\epsilon n)$ 時間，$\Psi[i]$ は定数時間で求まる．また，I のソートは基数ソートにより要素数に比例する時間で行える．以上より次の定理を得る．

定理 4 長さ n のテキスト中の長さ l の部分文字列に対する接尾辞配列は圧縮接尾辞配列から $O(l+\log^\epsilon n)$ 時間で構成できる．

3.4 実装

圧縮接尾辞配列の各レベル k には長さ $n_k=n/2^k$ の文字列に対する接尾辞配列を復元する情報 B_k, Ψ_k を格納する．これらも圧縮された形で格納されているためそのアクセス方法を記述する．

B_k B_k は長さ n_k のビットベクトルだが，$rank(B_k,i)=\sum_{j=1}^{i} B_k[j]$ を定数時間で求めるためにディレクトリ R_k, R'_k も必要である．これらは 2 つのパラメタ W, M を用いて次のように定義される．

$$R'_k[xW+y] = \sum_{j=xWM+1}^{xWM+yW} B_k[j]$$

$$R_k[x] = \sum_{j=1}^{xWM} B_k[j]$$

ここで $0 \le x < \frac{n_k}{WM}$, $0 \le y < M$ であり, W と M は $O(\log n)$ となるようにする. 本稿では $W = M = 256$ とする. ディレクトリのサイズは R'_k は長さが $\frac{n_k}{M}$ の配列で, 各要素は $\log_2(WM)$ ビットである. $WM = 2^{16}$ であるため各要素は 2 バイトで表せる. R_k は長さが $\frac{n_k}{WM}$ で各要素が 4 バイトの配列である. B_k と合わせるとバイト数は

$$n_k \left(\frac{1}{8} + \frac{2}{M} + \frac{4}{WM} \right) \approx 0.133 n_k$$

となる. R_k と R'_k の 2 つの表を引いた後は, 長さ W 以下のビット列の中の 1 の数を求める. これは 16 ビットのビット列に対する表を $W/16$ 回引くことで求められる.

rank の逆関数である *select* は $inverse(j)$ を求める時に使われるが, これは R_k と R'_k での二分探索で実現する. よって *rank* よりも遅くなる.

Ψ_k　Ψ_k は, 図 3 のように部分的には単調増加になっている. 正確には, 先頭の文字 $T_k[SA_k[i]]$ が等しい接尾辞に関しては, それらの $\Psi_k[i]$ は単調増加になっている. なぜなら先頭の文字が等しい接尾辞は, その先頭の文字を除いた接尾辞 $T_k[SA_k[\Psi_k[i]]..n_k]$ の辞書順にソートされているからである. よって転置ファイルと同様に $\Psi_k[i]$ を 1 つ左の値との差分 $\Psi_k[i] - \Psi_k[i-1]$ で表現することでサイズを小さくできる.

復号時には差分を順に足していくと $O(n_k)$ 時間かかるため, $L = O(\log n)$ 個おきに実際の値と差分が符号化されているビットの位置の表を持つことで高速化する. これで 1 つの Ψ_k の値を得る時間は $O(\log n)$ 時間になるが, これを定数時間にするには長さ $O(\log n)$ の全てのビットパタンに対する表を使う必要がある. 本稿では 16 ビットのパタンに対してその中に符号化されている差分の数, ビット数, 差分の合計を格納した表を用いる.

検索した単語 P の出現位置を列挙する際には $\Psi[i]$ を繰り返し求めるが, これは次のようにして高速化できる. P を接頭辞として持つ接尾辞は接尾辞配列中では連続した範囲に存在し, それらの Ψ の値も単調増加している. さらに, $\Psi[i]$ をたどる回数が $|P|$ 回までは Ψ の値は単調増加である. よって複数の出現位置を求める際にはそれらを一度に求める方が高速になる. ただし本稿ではまだ実装していない.

Ψ_k の計算は以下のように行う. まず, 接尾辞配列 SA_k を構成する. 次に, 接尾辞 $T_k[SA_k[i]..n_k]$ の辞書順の順位 i を $(T_k[SA_k[i]-1], i)$ をキーとして並び替える. これは基数ソートで線形時間で行える. この結果が Ψ_k となる. ただし $SA_k[i] = 1$ の場合は $T_k[SA_k[i]-1]$ が存在しないため除き, $pos_k^1 = i$ として格納する. また, $SA_k[i] = n_k$ に関しても $pos_k^2 = i$ として格納しておく.

C^{-1}　$T_k[SA_k[i]]$ を求めることはパタンの検索, テキストの復号で使われ, また $\Psi_k[i]$ を求める際にその値が符号化されている差分のリストを特定するためにも使われるが, これを定数時間で行うには *select* の定数時間アルゴリズムが必要である. しかしこれは複雑であるため, テキスト中の文字の累積頻度表 C の上での二分探索で求める.

3.5 索引の更新

テキスト T を圧縮接尾辞配列で保存している際に, 新しいテキスト T' (長さ m) を追加することを考える. T' は T の後ろにつなげ, その接尾辞の添字は T の接尾辞配列の中に挿入する. なお, T' を追加しても T の接尾辞の辞書順が変わらないようにするために, 各テキストの終りには一意な終端記号をつけておく.

まず, T' の接尾辞が挿入される場所を二分探索で求める. その際に, 挿入される場所の左右の接尾辞との一致長を求めておく. この値は各レベルでの Ψ_k の符号化での, 差分のリストの更新の際に用いる. B_k は接尾辞の添字が D の倍数かどうかで定義されるため, T' を追加しても挿入された接尾辞に対応して値が右にずれるだけである. Ψ_k に関しては挿入される接尾辞の分だけ値が大きくなるため計算し直すが, T' の接尾辞が挿入される場所はわかっているため $O(\log m)$ 時間で計算できる.

4 転置ファイルでの任意文字列検索

転置ファイルとはテキストを単語に区切り, 各単語に対してその出現位置のリストを格納した索引である. 単語の出現位置の列挙はこのリストを先頭から見るだけであるので高速である. また索引のサイズも小さい. 転置ファイルではテキストから切り出された単語に対しては高速に検索できるが, それ以外の単語に関しては見つからない. よって DNA 配列のように単語に区切ることのできないテキストからの検索には適さない. このような場合に q-gram 索引 [5] が用いられることがあるが, 単語の種類が多くなりサイズが大きくなる. そこで本節では転置ファイルを使った任意文字列検索アルゴリズムを示す. なお, テキストが単語で区切られていない場合, 単語の長さの上限 q を与えそこでテキストを区切るようにする.

まず, 探したいパタン $P[1..m]$ を含んでいる全ての単語 w に対して, その単語の出現位置に単語中の P のオフセットを加えたものを列挙する (図 4 上). 次に, P が 2 つの単語 w_1, w_2 にまたがっている場所を求める (図 4 中). これは P の接頭辞 $P[1..i-1]$ と接尾辞 $P[i..m]$ $(1 < i \le m)$ に対して, $P[1..i-1]$ を接尾辞として持つ単語 w_1 と $P[i..m]$ を接頭辞として持つ単語 w_2 の出現位置を列挙し, それらが連続している場所を求めればいい.

次に, P が 3 つ以上の単語にまたがっている場所を求める (図 4 下). これは, P の部分文字列 $P[i..j]$ $(1 < i \le j < m)$ を接尾辞として持つ単語 w_2 の出現位置と, $P[1..i-1]$ を接尾辞として持つ単語 w_1 の出現位置が連

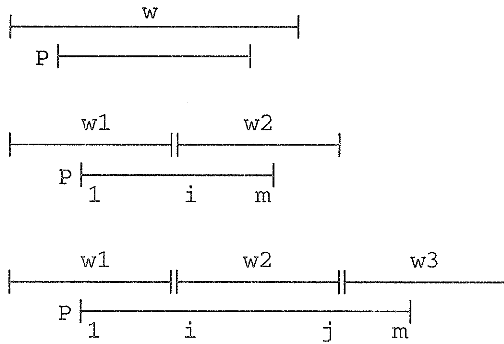

図 4: 転置ファイルでの任意文字列検索

続している場所を求め, それが $P[1..j]$ を接尾辞として持つ単語の位置であると見なして 2 つの単語用のアルゴリズムを適用することを繰り返せばいい.

DNA 配列に対して転置ファイルを索引として用いる場合, 単語の長さの上限 q を定める必要がある. 次節で用いるヒトの DNA 配列に対して, q の値を変化させたときの索引のサイズと検索時間を調べた. 索引のサイズ

表 1: DNA 配列に対する転置ファイルのサイズ

q	単語の種類	索引サイズ (バイト)
1	4	15401387
2	16	16355932
3	64	15378402
4	256	14636991
5	1024	13708916
6	4096	13174020
7	16383	13190895
8	64894	13697796

に関しては, 表 1 が示すように, $q = 6$ の場合に最も小さくなる.

図 5 はいくつかの q の値に対する, 検索結果の答えの数と検索時間を示している. q が大きいほど検索時間が短くなるが, $q = 6$ と $q = 8$ ではあまり差がない. よって次節の実験には $q = 6$ を用いる. なお, 英文の場合には単語の長さに上限は設けない.

5 実験結果

圧縮接尾辞配列と他の索引について, 索引のサイズと検索速度を比較した. 実験は Sun Ultra60 (メモリ 512MB) で行った. 実験に用いたテキストはヒトの DNA 配列の

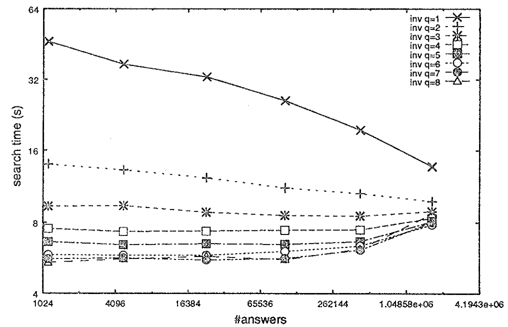

図 5: 転置ファイルによる検索時間と出現頻度

一部 30MB [1] と英文 約 27MB (Reuters corpus[2]) と和文 (1995 年の毎日新聞の記事の一部 30MB) である. 用いた索引は, 圧縮接尾辞配列, 転置ファイルである. また, 逐次検索とも比較した. 逐次検索は `grep` コマンドによる検索と, `zgrep` コマンドによる `gzip` で圧縮されたテキストからの検索である. 実験は全て主記憶上で行った. また, 時間は CPU 時間のみである.

5.1 DNA 配列に対する結果

DNA 配列に対する圧縮接尾辞配列のサイズは表 2 のようになる. ここでは $L = 32$ としている. D の値が大きくなると索引のサイズが小さくなり, $D = 64$ では無圧縮のテキストよりも小さくなる. ただしテキストを `gzip` で圧縮すると 1 文字 2 ビット程度になる. 圧縮接尾辞配列では B_0 が 1 文字辺り 1 ビットであるため, 索引はあまり小さくならない. よって, B_0 のようなビットベクトルで表すのではなく, $B_0[i] = 1$ となる i をそのまま符号化する方が D が大きい場合にサイズが小さくなる可能性がある.

表 2: DNA 配列に対する圧縮接尾辞配列のサイズ

D	索引サイズ (バイト)
8	42228544
16	36877130
32	32336054
64	30185594
無圧縮	31457280
転置ファイル ($q = 6$)	13174020
gzip	8492115

図 6 は各索引を用いてさまざまな文字列を検索した場合の検索時間と出現数 occ の関係を表している. 圧縮接

[1] ftp://ncbi.nlm.nih.gov/genbank/genomes/H_sapiens/hs_phase3.fna
[2] http://www.research.att.com/~lewis/

尾辞配列は $L = 32$, $D = 8$, 16, 32, 64 を用いている．転置ファイルでは $q = 6$ である．grep は無圧縮のテキストからの逐次検索，zgrep は gzip で圧縮したテキストからの逐次検索を表している．これらはテキスト全体を走査するため時間は出現数にはあまり依存しない．圧縮接尾辞配列 (csa) では時間は出現数にほぼ比例し，また D が小さいほど高速である．時間は $occ < 15000$ では逐次検索よりも高速であるが，出現数が多いと遅くなる．テキストがさらに長くなれば，出現数の増加よりもテキスト長の増加の方が大きいため，圧縮接尾辞配列の効果が大きくなる．

圧縮接尾辞配列では，出現数が多い場合にそれらの出現位置を列挙することは時間がかかるが，出現数を求めることは $O(|P| \log n)$ 時間で行えるため，複数単語を指定する検索ではまず出現数が少ない単語の位置を求め，その近くに他の単語が現れているかを調べる方が高速になると思われる．この際に Ψ や *inverse* を利用してテキスト中で近くにある文字列を列挙すれば良い．

転置ファイル (inv) は grep よりも遅くなっている．これは DNA 配列ではテキスト中の文字の出現頻度にあまり偏りがないため，どんな文字列を検索する場合でも文字列の長さの 1/4 (4 はアルファベットサイズ) 以上の数の単語を調べる必要があるためである．このような場合には q-gram 索引の方が効率が良くなると思われるが，q-gram 索引では扱う単語の種類が多くなり，q が大きくなると単語の種類が文字列の長さ n に近づく．その結果必要なメモリ量は無圧縮の接尾辞配列よりも多くなってしまう．この図より，DNA 配列に対しては転置ファイルは効果がなく，圧縮接尾辞配列は答えの数が多くない場合には高速であることがわかる．

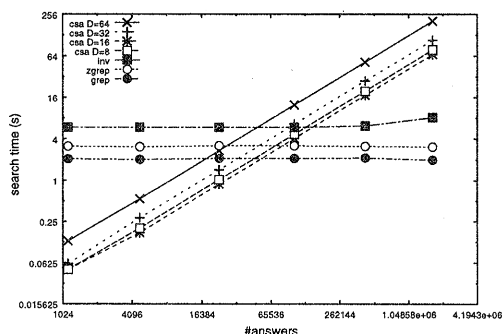

図 6: DNA 配列に対する検索時間と出現頻度

5.2 英文に対する結果

英文に関しては，頻出の単語 “the” と，それよりは少ない “Reuter” に関して検索速度を測った (図 7, 図 8)．横軸が索引のサイズで，縦軸が検索時間である．圧縮接尾辞配列に対しては $L = 32$ と $L = 128$ を試した．L が小さいほど高速になるが索引は大きくなる．D の値は 4 から 64 であり，D が小さいほど高速になり，サイズが大きくなる．$L = 32$, 128 の両方で $D \geq 16$ で元のテキストよりも小さくなる．

圧縮接尾辞配列と逐次検索では，“the” の検索は答えの数が多いため逐次検索の方が速いが，“Reuter” の検索では圧縮接尾辞配列の方が速い．ただし，転置ファイルはどちらの単語の場合でも索引サイズ，検索時間ともに圧縮接尾辞配列よりも優れている．つまり，英文では複数の単語にまたがる文字列の検索は行われないため，転置ファイルが効果的である．

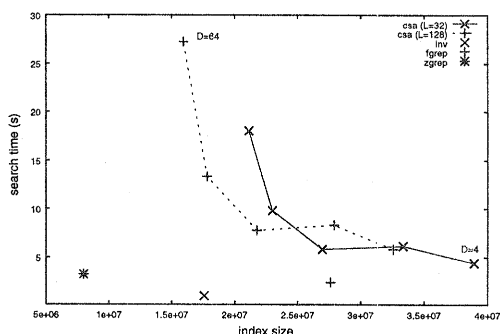

図 7: “the” の検索時間と索引サイズ

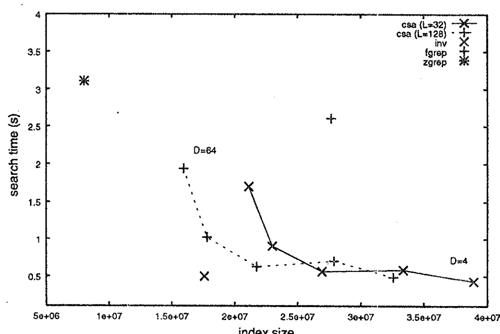

図 8: “Reuter” の検索時間と索引サイズ

5.3 和文に対する結果

次に，日本語の新聞記事に対して実験を行った．圧縮接尾辞配列のパラメタは英文に対する実験から $L = 128, D = 16$ の場合の結果のみを示す．なお，パラメタと圧縮サイズ，検索時間の関係は英文の時と同様である．表 3 は日本語の新聞記事に対する各索引のサイズを表している．また，図 9 は日本語 (毎日新聞の記事) に対する各索引を用いた検索時間と単語の出現頻度の関係を表

表 3: 和文に対する索引のサイズ

	索引サイズ (バイト)
無圧縮	31391581
圧縮接尾辞配列	29837522
転置ファイル	21491769
gzip	14911734

している. 検索の実験に用いた単語は, 出現頻度の小さい順に「東京都」「経済」「首相」「京都」「東京」「阪神」「日本」「1」「が」「を」「に」「,」「の」である. 圧縮接尾辞配列と grep, zgrep についてはDNA配列の場合と同じ傾向を示すが, 転置ファイルに関しては異なっている. 転置ファイルを用いた場合の単語の検索時間は grep よりも高速である. これは単語に含まれる文字の頻度が文字列の長さよりも相対的に少ないからである. よって転置ファイルによる和文の検索は多くの場合に高速であるが, 助詞を含む言葉の検索では遅くなる場合がある.

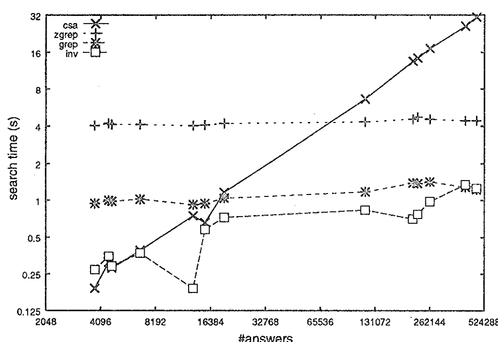

図 9: 和文に対する検索時間と出現頻度

6 結論

全文検索索引である圧縮接尾辞配列を拡張し, テキストを使わない検索を実現した. 実験により索引のサイズが元のテキストより小さくなることを確認した. これにより, テキストの圧縮と, そこからの高速な検索の両立が可能となった. 英文, 和文に対しては検索速度, 索引サイズともに転置ファイルより劣るが, DNA配列のようにテキストが単語に区切れない場合には有効である. また, 転置ファイルによる任意文字列の検索は英文, 和文に関しては効果的であることもわかった.

圧縮接尾辞配列を用いた検索の速度は, 検索単語の出現数に大きく依存する. 出現数が少ない場合には高速だが多いと逐次検索よりも遅くなる場合がある. よって, 複数の出現位置を一度に計算したり表引きの工夫による高速化や, 出現数が多い場合に他の検索法に切り替えるなどの工夫が必要である. また, 本稿での実験は主記憶上で行っており, 索引をディスク上に置く場合の実験は行っていない. これらは今後の課題とする.

参考文献

[1] M. Farach and T. Thorup. String-matching in Lempel-Ziv Compressed Strings. In *27th ACM Symposium on Theory of Computing*, pages 703–713, 1995.

[2] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. Technical Report TR00-03, Dipartimento di Informatica, Università di Pisa, March 2000.

[3] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000. `http://www.cs.duke.edu/~jsv/Papers/catalog/node68.html`.

[4] G. Jacobson. Space-efficient Static Trees and Graphs. In *30th IEEE Symp. on Foundations of Computer Science*, pages 549–554, 1989.

[5] J. Kärkkäinen and E. Sutinen. Lempel-Ziv Index for q-Grams. *Algorithmica*, 21(1):137–154, 1998.

[6] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A Unifying Framework for Compressed Pattern Matching. In *Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99)*, pages 89–96, September 1999.

[7] S. Kurtz. Reducing the space requirement of suffix trees. Technical Report 98-03, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 1998.

[8] U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[9] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(12):262–272, 1976.