RA-007

高速復元可能な接尾辞配列圧縮法

Compressing method of Suffix Array whose decoding is fast

田中 洋輔 *　小野 廣隆 †　定兼 邦彦 ‡　山下 雅史 †
Yosuke Tanaka　Hirotaka Ono　Kunihiko Sadakane　Masafumi Yamashita

1 はじめに

1.1 概要

大規模データに対する高速な文字列検索は接尾辞配列 [5] (SA) を用いて実現できるが, SA には多くの容量が必要になってしまう. SA を圧縮する様々な方法が提案されているが, 本論文では出現頻度の高いフレーズの検索が既存の圧縮法に比べて性能がよいような圧縮方法を提案する. 提案手法では, SA を大きさ S のブロックに分割し, そのブロック内でソートを行い, 差分を取ったものを保存し, 検索時は差分からソート後の SA を取り戻し, 区間 S 内を全て逐次的に検索する. これで検索フレーズの全ての出現位置を得ることができる. 最終的には実験により特に検索フレーズの頻度が高い場合, 多くの入力データで提案手法の性能が既存の方法より優れていることを示す.

1.2 問題提起

計算機の性能の飛躍的な進歩に伴い, 様々なデータが大規模化し, また大量のデータが存在するが, この中からある文字列の存在する位置を得る機能, つまり文字列検索の機能はとても重要な機能である. 文字列検索という問題は以下のように表せる.

問題 1. 長さ n, アルファベット数 σ である文字列 *(データ)* T 中から, あるフレーズ P の全ての出現位置を得る *(*探索する*)*.

これを実現する方法を考えるが, フレーズ P をデータの先頭から見てゆくことで探索する方法では, 検索を何度も行う場合, データ長 n が非常に大きいと非常に多くの時間がかかってしまうが, 一度索引を作っておけば高速に検索を完了することができる. 索引には, データ中の各単語 (フレーズ) の出現位置を保存しておく転置インデックスと, データ中の全接尾辞の出現位置を辞書順に保存しておく接尾辞配列 [5] (以下 SA) などがある. 転置インデックスは索引容量が比較的小さいが, 文字列を単語に区切る必要があり, 英文のようなデータではホワイトスペースの間にあるものを単語とすればよいが, 日本語や DNA 配列などのデータではどのように単語を区切るのかという問題が生じる. 一方, SA では単語の概念のないようなデータを問題なく索引付けすることができる. ただし, SA は比較的多くの容量が必要で, そのまま保存するとデータサイズの 4 倍の容量が必要になるため, これを圧縮するための様々な研究がなされている.

1.3 関連研究

Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/) では接尾辞配列を圧縮する様々なアルゴリズム (実装) が公開されている [1]. また, アルゴリズムの比較法やそのためのデータも公

*九州大学大学院システム情報科学府
†九州大学大学院システム情報科学研究院
‡国立情報学研究所情報学プリンシプル研究系

表 1: 通常の SA と提案手法の索引容量と検索時間

	索引容量 (bit)	検索時間
通常の SA	$n \log n$	$O(\|P\| \log n + occ)$
提案手法	$n(\log n - \log S + 2)$	$O(\|P\| \log(n/S) + S\|P\| + occ)$
提案手法 (別表現)	$(1-\delta)n \log n + 2n$	$O(\|P\|n^{\delta} + occ)$

開されている. 具体的には FM-index [1], Compressed Suffix Array [1], Repair Suffix Array [3] といったものがある. FM-index, Compressed Suffix Array は圧縮率は高いが検索するフレーズの頻度が高い場合の検索の速度が遅いといった特徴がある. 一方, Repair Suffix Array (以下 RPSA) はそのような場合でも高速な検索が可能である (ただし圧縮率は低くなる).

1.4 本研究

提案手法は RPSA と同じく, 高頻度なフレーズの検索に適しているが FM-index などと比べると圧縮率が低いという性質をもつ. 提案手法では, SA を大きさ S のブロックに分割し, そのブロック内でソートを行い, 差分を取ったものを保存しておく. このときこの差分の列の符号化にはゴロム符号化 [4] という符号化を用いる. 検索時は差分からソート後の SA を取り戻し, 区間 S 内を全て逐次的に検索する. これで検索フレーズの全ての出現位置を得ることができる. 後の実験により, 提案手法は RPSA と比較して, (多くのデータで) 索引の容量が小さく, 頻度が特に大きいフレーズの検索では高速で, また索引作成時間が短いことを示す.

表 1 は通常の SA と提案手法の索引容量および検索時間を示す[1]. ただし occ は検索フレーズの T 中での出現数である. $S = \log n$ のとき提案手法の検索時間は $O(|P| \log n + occ)$ であり通常の SA と同じで, 一方容量は $n \log n - n(\log \log n - 2)$ となり通常の SA より小さくなる. また $S = n^{\delta}$ (ただし $0 < \delta < 1$) とすると容量は $(1-\delta)n \log n + 2n$, 検索時間は $O(|P|n^{\delta} + occ)$ と表現できる.

[1]本論文で log と表記したとき, 底は 2 であるとする.

2 準備

2.1 接尾辞配列

接尾辞とは文字列の後ろ部分, 最後の 1 語 ($T[n]$) を含む部分文字列で, T 中の j の位置にある接尾辞は $T[j...n]$ となる. 長さ n の文字列データ T に対して, 接尾辞配列 (SA) とは長さ n の配列であり, 辞書順が i 番目の接尾辞の出現位置が j のとき $SA[i] = j$ となるようなものである. 図 1 は T:gcgacacgac に対する SA の例である. SA を int 型で保存した場合, 記憶容量は $32n$ bit ($4n$ $Byte$) であり, n が大きい場合かなり大きくなる.

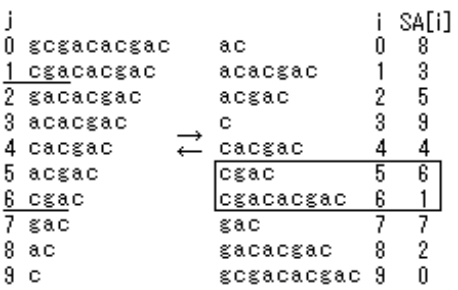

図 1: T:gcgacacgac に対する SA の例

SA を用いて任意の文字列 P の全出現位置を得ることができる. 接尾辞を辞書順に並べたとき, 同じ接頭辞を持つ接尾辞は必ず隣接し, 一定の範囲内に含まれる. その接頭辞が検索するフレーズ P であると考えると, P の全出現位置を得ることは, SA 中の対応する範囲 (l, r) を求

めることで実現できる．辞書順に並んだ接尾辞の任意の位置 (i 行, d 列) は T と SA を用いて, $T[SA[i]+d]$ で参照できるので, 並んだ接尾辞に対して 2 分探索を行うことで $O(|P|\log n + occ)$ の時間で検索を実現できる．

2.2 RPSA

RPSA では SA ではなく SA の差分 (以下 dif_SA) と一定区間 S おきの SA の値を保存しておき, 検索時にこれらから SA を取り戻し, それを用いて検索を行う．RPSA では Re-Pair [2] という圧縮アルゴリズムが用いられている．RPSA の概要は dif_SA 中の出現頻度の高い隣接するペアを 1 つの文字に変換することで dif_SA を圧縮するというものである．ただしこのとき変換法則を示す辞書の容量も必要になる．dif_SA 中の頻出するペアは偶然に生じたものもあるが, 多くはデータ T の冗長性により生じるものである．SA から RPSA を作成するとき, 頻度の大きい順にペアを作っていく場合 $O(n\log n)$ の時間がかかるか, もしくは時間が $O(n)$ だが作成時のメモリ使用量が多くなりすぎるかということなってしまう．データ T の冗長性により生じるペアのみ変換する場合に, 比較的少メモリかつ $O(n)$ で作成できる方法が提案されている [3]．後者の容量は前者と比べ 1%から 14%程の違いしかないので, 今回の実験ではこちらの方法での RPSA と比較を行った．RPSA の容量は k 次の経験的エントロピー H_k (参考文献 [1] 参照) を用いて, $H_k\log(1/H_k)n\log n + n$ となり (ただし $0 < \alpha < 1$, このとき任意の k で $k \le \alpha\log_\sigma n$), 検索時間は $O(occ + \log n)$ となる．

3 提案手法

3.1 圧縮法・検索法の概要

提案手法での索引の作成方法を示す．SA を大きさ S の一定区間 (ブロック) で区切りその範囲内で SA の値を昇順にソートし (ソート後の SA を $sorted\_SA$ と表記) , その差分の値を保存する (この差分の列を dif_sSA と表記) ．またソート前の区間の先頭の SA の値も別途保存しておく ($samp\_SA$ と表記) .

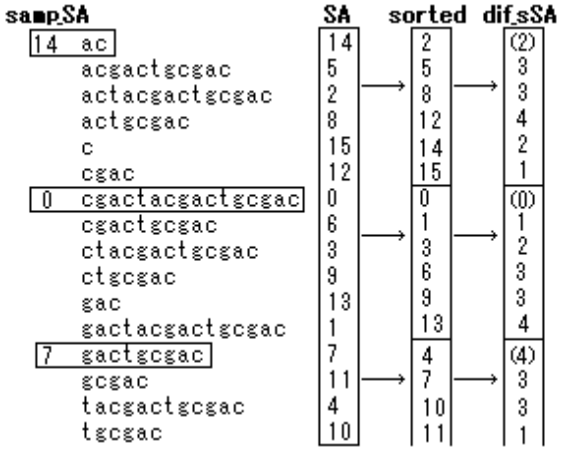

図 2: 提案手法の索引作成

次にそれらを用いた検索法を示す．P が与えられたとき, まず $samp\_SA$ (に対応する接尾辞) のみに関し 2 分探索を行う．この結果, P にマッチするものがどのブロック内に存在する可能性があるのかということがわかるので, その範囲内の $sorted\_SA$ の値を dif_sSA から全て取り戻し, その全ての値に対応する接尾辞を P と逐次的に照合してゆくことで全ての P の出現位置を得ることができる．

3.2 検索法の詳細と検索時間

$samp\_SA$ に関する 2 分探索の部分を詳細に説明する．まず, 全ての i での $samp\_SA[i]$ と P がマッチしなかった場合 (例えば図 2 で P=act のとき) は, P はある i で $samp\_SA[i]$ の接尾辞より大きく, $samp\_SA[i+1]$ の接尾辞より小さくなる．よって, P はある 1 つの区間の中にしか存在する可能性がないことがわかるので, この区間のみ逐次的に探索すればよいことになる．

次に, ある 1 つの i で $samp\_SA[i]$ と P がマッチした場合 (例えば図で P=cga のとき) は, P

はある i で $samp\_SA[i-1]$ の接尾辞より大きく, $samp\_SA[i]$ の接尾辞とマッチし, $samp\_SA[i+1]$ の接尾辞より小さくなる. よって, $i-1$ 番目と i 番目のブロックに存在する可能性があり, その 2 つのブロックを調べる必要がある.

T 中の P の頻度が高いとき, 連続する複数の i の値で $samp\_SA[i]$ と P がマッチすることが考えられる. このとき, 両端のブロックは調べる必要があるが, 中間のブロック中の接尾辞は全て P とマッチするので調べる必要はない. よってこの場合は両端 2 つのブロックのみ調べればよいことになる (SA を復元する必要はある).

提案手法の検索時間は, まず $samp\_SA$ のみに関する 2 分探索に $O(|P|\log(n/S))$, 一方, 区間 S 内の逐次検索には, 各 S 個の接尾辞で最大で長さ $|P|$ まで探索しないといけないかもしれないので, 最悪で $O(S|P|)$ の時間がかかる. よって $O(|P|\log(n/S)+S|P|+occ)$ と表記できる.

3.3 差分列の符号化

差分列 dif_sSA を文字列とみなし符号化することを考えるが, これにはゴロム符号化 [4] という符号化を用いる. 以下, 符号化の手順を示す.

パラメータ M を決定.

符号化する整数が x ($x \geq 0$) のとき, 商 $q = \lfloor \frac{x}{M} \rfloor$ を 1 進 (1 のラン) で符号化.

商と余りの符号の間の区切りとして 0 を出力.

余り $r = x \mod M$ は, M が 2 のべき乗のとき, $\log M$ ビットの 2 進符号化. そうでないとき, $b = \lceil \log M \rceil$, $2^b - M - 1$ までの数は b-1 ビットで, 残りを $2^b - M$ を加えた上で b ビットの 2 進符号列で符号化.

例えば $M = 16$ での $x = 37$ の符号は, 商 $q = \lfloor \frac{37}{16} \rfloor = 2$ の符号は"11", 余り $r = 37 \bmod 16 = 5$ の符号は"0101", よって"11 0 0101"となる. ある整数 x を符号化するのに必要になるビット数は $\lfloor x/M \rfloor + 1 + \lceil \log M \rceil$ となる.

以下では, ゴロム符号化された dif_sSA の最悪時の容量 $GL(n,S)$ を求めてゆく. $sorted\_SA$ 中のある 1 つのブロックに着目したとき, その値を $(1 \leq) x_1 < x_2 < ... < x_s (\leq n)$, 差分を $d_i = x_i - x_{i-1}$ とし, パラメータが M のゴロム符号化で符号化したときの容量は,

$$\sum_{i=1}^{S} \left(\left\lfloor \frac{d_i}{M} \right\rfloor + 1 + \log M \right) \leq \sum_{i=1}^{S} \left(\frac{d_i}{M} + 1 + \log M \right)$$

ここで, $\sum_{i=1}^{S} d_i = x_s \leq n$ が成り立つので,

$$\sum_{i=1}^{S} \left(\frac{d_i}{M} + 1 + \log M \right) = S + \frac{x_s}{M} + S\log M \leq S\log M + S + \frac{n}{M}$$

ブロック数は n/S 個存在するので,

$$GL(n,S,M) = \frac{n}{S}\left(S\log M + S + \frac{n}{M} \right) = n\log M + n + \frac{n^2}{SM}$$

$GL(n,S)$ を M で微分し, 最小となるときの M を求める.

$$\frac{n}{M\ln 2} - \frac{n^2}{SM^2} = 0$$

$$M = \frac{n\ln 2}{S} \quad (1)$$

よって, dif_sSA をゴロム符号化するとき M を上記の式のように決定するときが最良に近いことが予測できる. さらにこの M を $GL(n,S,M)$ に代入すると,

$$GL(n,S) = n(\log n - \log S) + \beta n$$

ただし $\beta = 1 + \log\ln 2 + \frac{1}{\ln 2}$ であるが, この値は約 1.913928 になるので, これを 2 とすると,

$$GL(n,S) \fallingdotseq n(\log n - \log S + 2)$$

これより, S が 2 倍 ($\log S$ が+1) になるとき容量は n bit 減少し, S が $\frac{1}{2}$ 倍 ($\log S$ が-1) になるとき容量は n bit 増加するということがわかる. また n が 2 倍になったとき同時に S も 2 倍すると容量 GL は 2 倍になる.

以上の結論として以下を得る.

定理 1. 長さ n の文字列 T に対し, 理論的最悪容量が $n(\log n - \log S + 2)$ となる索引が存在し, それは任意のフレーズ P を $O(|P|\log(n/S) + S|P| + occ)$ の時間で検索できる.

4 実験と考察

計算機実験により, 提案手法の性能と RPSA の性能を比較する. 比較内容は索引容量 ($size$), 索引作成時間 ($build$), 検索時間 ($locate$) である. 検索時間は具体的に, データ中に存在する長さ $length$ のランダムに選ばれたフレーズ 1000 個を全て検索したときの時間を測定する. 提案手法では差分列の符号化にゴロム符号化を用いているが, このときそのパラメータ M は前節で説明したような理由により $M = \frac{n \ln 2}{S}$ とする. また, 同時に高頻度のフレーズの検索に適した方法でないものとの比較を行う. 具体的には FM-index を改良した af-index [1] という実装を用いる. af-index には samplerate が存在し (S とする) , この値おきの SA を保存しておくものであるが, 実験では提案手法の容量と af-index の容量が近い値になるように S の値を選ぶことにする. 検索に用いるデータ (および実装の一部) は Pizza&Chili Corpus のものを使用している.

表 2 は 50MByte の英文のデータの場合の結果である. 索引容量 $size$ に関しては, $S = 2048$ のときでも RPSA より小さい値となっている. 検索時間 $locate$ に関しては, $length$ が 3,4 のときは, 提案手法は RPSA よりも性能がいいようだが, $length = 5$ では S の値により異なり, $length$ が 6 以上では RPSA のほうが性能がよい. また $length$ が 10 程では, 提案手法と af-index で $locate$ の値が同じくらいとなった. また tot_occ は 1000 回の検索で得られたフレーズの総出現数を表していて, $length = 3$ では 1000 回で 102,874,505 個, 1 回平均 102,874 個のフレーズが得られている. 表 3 はデータが 50MByte の C 言語のソースコードのときの実験結果である. $size$ は, $S = 16384$ では RPSA よりも小さい値になっている. また, 英文と比べると, $length$ による tot_occ の減少率が小さく, それにより $locate$ が, $length$ が 10 でも af-index とかなり差があるように思われる. RPSA と比較すると, $length$ が 5 までは提案手法のほうが良く, $length$ が 6 以上では RPSA のほうが良いように思われる. 表 4 は 50MByte の xml データの場合の実験結果である. このデータでは RPSA の圧縮率がとてもよいが, C 言語ソースよりさらに tot_occ の減少率が小さくなっていることにより, $length = 10$ でも RPSA より高速となった. 表 5 は 50MByte の DNA 配列の場合の実験結果である. $size$ は RPSA の半分程になっている. $locate$ は $length$ が 3,4 では RPSA より小さく, 7 以上では大きくなっている.

以上の結果より, RPSA と比較して, 索引容量は (多くのデータで) 小さくなり, 検索時間は, $length$ が小さい, つまり検索するフレーズの頻度が高いと提案手法のほうがよいが, $length$ が増える, つまり検索するフレーズの頻度が低くなると RPSA のほうが性能がよくなる. また索引作成時間 ($build$) はどのデータでも RPSA に比べておよそ 3 倍程高速となっている.

5 ゴロム符号化を用いる理由

文字列の各アルファベットの出現確率が幾何分布であるとき, パラメータ $M = \frac{1}{-\log p}$ のゴロム符号化で情報理論的下限にほぼ近いところまで圧縮することができる (ただし $p \gg 1-p$ であることが必要). ここで幾何分布とは確率密度関数が $P\{X = k\} = (1-p)p^{k-1}$ で与えられるような離散確率分布である. また, 以下の命題を考える.

命題 1. $1, 2, \ldots, n$ の順列の全体の集合からある 1 つの順列が選ばれる確率が一様に $\frac{1}{n!}$ であると仮定する. このとき, 選ばれた順列を区間 S 内でソートして差分をとったとき, その差分列の各差分の出現確率は幾何分布となる.

Proof. 1 つのブロックに対し, 長さ n の 0,1 文字列 B を考える. $B[i] = 1 \Longleftrightarrow i$ がブロックに

表 2: ENGLISH (english text) : 50MByte　(※ locate は検索 1000 回での合計時間)

	length	tot_occ	RPSA	提案手法		af-index	
S			-	2048	16384	8	16
size(MByte)			123.267	106.063	86.125	91.904	65.690
build(sec)			144.35	52.53	51.44	140.83	139.57
locate(sec)	3	102,874,505	8.27	3.57	6.00	228.20	648.89
locate(sec)	4	32,019,956	2.76	1.38	3.50	69.77	231.25
locate(sec)	5	11,093,265	0.95	0.75	2.44	31.71	71.09
locate(sec)	6	3,688,971	0.31	0.41	2.07	8.45	22.74
locate(sec)	7	1,129,823	0.14	0.32	1.88	2.51	6.38
locate(sec)	8	659,531	0.09	0.29	3.28	1.46	3.85
locate(sec)	9	429,527	0.08	0.27	3.22	0.68	1.80
locate(sec)	10	307,423	0.08	0.25	1.75	0.48	1.27

表 3: SOURCES (source program code) : 50MByte

	length	tot_occ	RPSA	提案手法		af-index	
S			-	2048	16384	8	16
size(MByte)			92.940	106.915	86.947	97.064	70.849
build(sec)			106.13	35.35	34.29	130.26	130.16
locate(sec)	3	62,906,189	4.95	2.28	3.92	114.66	307.45
locate(sec)	4	45,081,631	3.73	1.64	3.13	71.42	202.11
locate(sec)	5	27,900,228	2.37	1.09	2.51	40.70	115.02
locate(sec)	6	17,187,285	1.50	0.77	2.08	23.63	69.10
locate(sec)	10	6,594,561	0.69	0.39	1.71	8.31	23.15

表 4: XML (structured code) : 50MByte

	length	tot_occ	RPSA	提案手法		af-index	
S			-	2048	16384	8	16
size(MByte)			51.445	106.147	86.311	85.498	59.284
build(sec)			147.07	43.07	41.76	98.26	98.02
locate(sec)	3	188,004,221	12.35	6.16	8.37	219.16	641.16
locate(sec)	4	122,186,328	8.59	4.13	6.33	141.25	420.41
locate(sec)	5	96,771,585	7.84	3.35	5.45	118.71	338.88
locate(sec)	6	81,362,669	7.42	2.87	5.00	102.20	304.72
locate(sec)	10	38,779,962	6.12	1.53	3.54	50.48	143.88

表 5: DNA (gene DNA sequences) : 50MByte

	length	tot_occ	RPSA	提案手法		af-index
S			-	2048	16384	8
size(MByte)			172.436	105.852	86.040	80.562
build(sec)			131.42	39.85	38.50	85.21
locate(sec)	3	961,528,102	62.13	32.75	35.07	1647.79
locate(sec)	4	266,041,360	17.36	9.35	11.88	457.25
locate(sec)	5	73,052,553	4.83	2.86	5.45	124.74
locate(sec)	6	20,618,788	1.34	1.07	3.48	34.70
locate(sec)	7	6,293,532	0.44	0.61	2.53	10.42

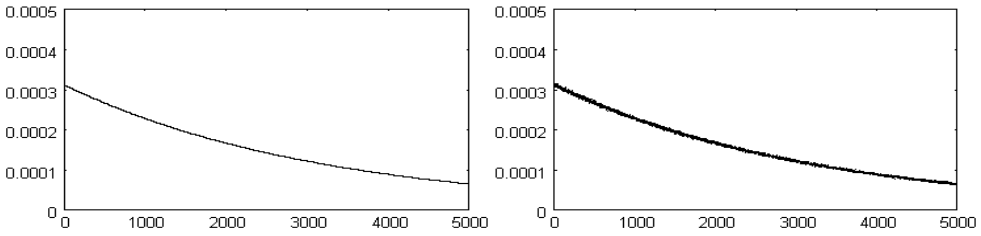

図 3: $p = 1 - \frac{S}{n}$ の幾何分布 (左) と RSA での差分の出現確率 (右)

含まれる と定義すると, $B[i] = 1$ となる確率は $\frac{S}{n}$. $B[i] = 1$ になる事象は他の i とは独立で, 確率は全て等しい. よって, 差分列の各差分の出現確率は幾何分布となる. □

この命題を確認する意味で, 以下のような実験を行う. まず, ランダムな置換を生成できるような (全ての置換の出現確率が同じで, その中から 1 つを取ってくるような) 関数を実装し, それを用いて長さ n の置換 RSA を生成し, これから dif_sSA を作成する. $n = 50M$, $S = 16384$ の値で実際に実験を行うと, ($M = \frac{n \ln 2}{S}$ で) ゴロム符号化したときの容量は 85934142Byte, dif_sSA の情報理論的下限は 85758304Byte でほぼ同じ値となった. これより RSA から作られるような dif_sSA は幾何分布と対応しそうであると予測する. 今回, $n = 50M$, $S = 16384$ としているので, パラメータ $p = 1 - \frac{S}{n} = 0.9996875$ の幾何分布との比較を行う. その結果が図 3 で, 差分がほぼ幾何分布近辺に分布していることがわかる.

以下では, 実際に文字列 T から作られた dif_sSA の差分の出現確率と幾何分布の比較, および差分列をゴロム符号化したときの容量と差分列の情報理論的下限 (エントロピー) の比較を行う. 図 4(左) は離散無記憶情報源 (DMS) から作られた dif_sSA の差分の出現確率で, 横軸は差分 x, 縦軸はその差分の出現確率 $P(x)$ を表す. 今回 DMS で作成された文字列は $n = 50M$, $\sigma = 8$ で, 各アルファベットの出現頻度が 3108735, 5770323, 826536, 29206729, 1659297, 354822, 11401858, 100500, エントロピーは約 $H(P) = 1.859085$ となっていて, これを $S = 16384$ でソートしている. 図中の点が dif_sSA の各差分の出現確率を表しているが, 各点が幾何分布付近に分布していることがわかり, dif_sSA が幾何分布と対応しそうであると予測できる. またゴロム符号化すると 85960082Byte, 差分列の情報理論的下限は 85756076Byte で, ゴロム符号

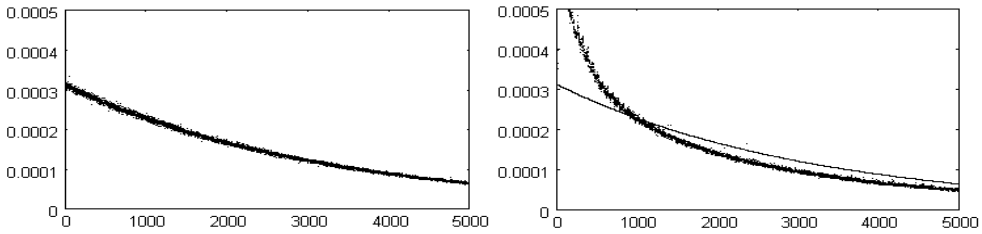

図 4: DMS での差分の出現確率 (左) と英文での差分の出現確率 (右)

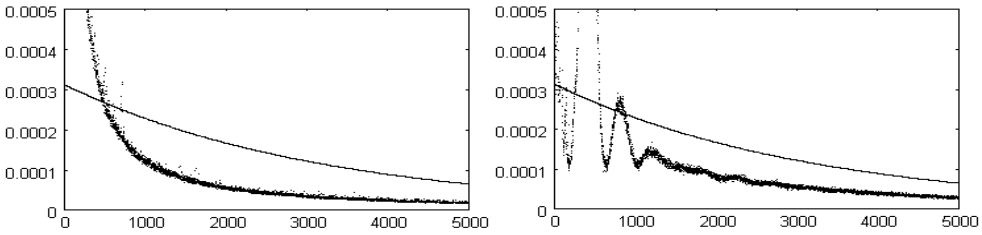

図 5: C ソースでの差分の出現確率 (左) と xml での差分の出現確率 (右)

化で下限近くまで圧縮できていることがわかる.

一方, DMS から生成されたのではない一般的な文字列に関してであるが, 例えば図 4(右) は英文データに対する出力である. DMS のときと比較して幾何分布からずれが生じる形になっている. ただ, これをゴロム符号化すると 86.125MByte, 英文からの差分列の情報理論的下限は 84.992MByte で同じくらいの値になる. また, DNA 配列で同様の実験を行うと, ゴロム符号化で 86.040MByte, 下限が 85.562MByte で同じくらいの値になる. しかし xml のデータでは, ゴロム符号化が 86.311MByte, 下限が 77.391MByte で 10%ほどの差があり, さらに C 言語のデータではゴロム符号化で 86.947MByte, 下限が 74.259MBte で 15%ほどの差が生じた. 図 5 を見ても, 英文のときに比べ幾何分布からのずれが大きいことがわかる.

6 おわりに

実験により, 提案手法は RPSA と比較して, 多くのデータで索引容量が小さく, 検索フレーズの頻度が非常に大きい場合は高速で, また索引作成時間は 3 倍程高速であることが示された. 今後の課題としては, $sorted\_SA$ を SA からではなく, T から直接作ることで索引作成時間を高速化できないかということを考えていて, それについて実験を行ってゆきたい.

参考文献

[1] G. Navarro, V. Mäkinen : “Compressed Full-Text Indexes” ACM Computing Surveys 39(1), article 2, 61 pages, 2007

[2] J. Lasson, A. Moffat: “Off-line dictionary-based compression” Proc.IEEE,88(11):1722-1732, 2000

[3] R. González, G. Navarro : “Compressed Text Indexes with Fast Locate” Proc. CPM’07, pages 216-227. LNCS 4580.

[4] S. W. Golomb : “Run-length encodings” IEEE Transactions on Information Theory, IT–12(3):399–401

[5] U. Manber, G. Myers : “Suffix arrays: a new method for on-line string searches” SIAM Journal on Computing, Volume 22, Issue 5 (October 1993), pp. 935-948