Vol.2010-AL-128 No.1
2010/1/26

高頻度なフレーズの検索が高速な索引

田 中 洋 輔[†1]　小 野 廣 隆[†1]
定 兼 邦 彦[†2]　山 下 雅 史[†1]

大規模データに対する高速な文字列検索は接尾辞配列 (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, 提案手法は, SA と転置インデックスを組み合わせることで, 出現頻度の高いフレーズの検索が既存の索引に比べ高速な索引を実現する. 最終的には実験により, 以前提案した別手法[5] と似た性能で, ある程度検索時間を調整可能であることを示す.

Index with fast searching in case phrase is frequent

Yosuke Tanaka ,[†1] Hirotaka Ono ,[†1]
Kunihiko Sadakane [†2] and Masafumi Yamashita [†1]

String pattern matching for large-scale data is efficiently achieved by suffix array (SA), but SA requires a large space. Therefore, various methods to compress SA have been proposed. In this paper, we combine SA and Inverted index. As a result, performance of the proposed method is better than existing ones when searching frequent phrases. Experiments show that the proposed method is similar to one which is proposed by us in other times, and able to control search time at some level.

1. はじめに

1.1 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ている. 本研究では大規模文字列データを扱う. 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 の文字列 T 中から, フレーズ P の全出現位置を得る.

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[4] (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.

1.2 関連研究

Pizza&Chili Corpus (`http://pizzachili.dcc.uchile.cl`) では様々な接尾辞配列の圧縮法が公開されている[1]. 具体的には FM-index [Ferragina, Manzini 05], Compressed Suffix Array [Grossi, Vitter 06], Repair Suffix Array (RPSA)[2] 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, RPSA は圧縮率は低いが高頻度な場合でも高速な検索が可能である.

1.3 本研究

提案手法では, 転置インデックスと SA を組み合わせる. 具体的には, 頻度の高いフレーズは転置インデックスで頻度の低いフレーズは SA で保存する. 結果として, RPSA と同じく, FM-index などと比べると圧縮率が低いが高頻度なフレーズの検索が高速であるという性質をもつ.

2. 準備

2.1 接尾辞配列

まず接尾辞とは, $T[j...n]$ と表され, テキスト T に対して n 個の接尾辞が存在する. 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

†1 九州大学 Kyushu University
†2 国立情報学研究所 National Institute of Informatics

Vol.2010-AL-128 No.1
2010/1/26

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 $32n$ ビット ($4n$ バイト) であり, n が大きい場合かなり大きくなる.

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 $O(m \log n + occ)$ の時間で検索を実現できる[⋆1] [⋆2].

j		i	SA[i]
0 gcgacacgac	ac	0	8
1 cgacacgac	acacgac	1	3
2 gacacgac	acgac	2	5
3 acacgac	c	3	9
4 cacgac	cacgac	4	4
5 acgac	cgac	5	6
6 cgac	cgacacgac	6	1
7 gac	gac	7	7
8 ac	gacacgac	8	2
9 c	gcgacacgac	9	0

図 1 T:gcgacacgac に対する SA の例
Fig. 1 SA for T:gcgacacgac.

2.2 RPSA

RPSA では SA そのものではなく, 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 [Lasson, Moffat 00] 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 中の頻出するペアの多くは, 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, 比較的省メモリかつ $O(n)$ で作成できる方法が提案されている. 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[1)] を用いて, $H_k \log(1/H_k)n \log n + n$ となり ($0 < \alpha < 1$, $0 \leq k \leq \alpha \log_\sigma n$), P を検索する時間は $O(m \log n + occ)$ となる.

2.3 SADIV

自身の別研究であり, 提案手法と似たような容量, 検索時間のトレードオフとなる索引圧縮法である SA 分割 (以下 SADIV) [5)] について説明する. SADIV での索引の作成方法は, まず 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 dif_sSA を符号化し保存する. またソート前の区間の先頭の SA の値 $samp\_SA$ も別途保存しておく.

samp_SA		SA	sorted	dif_sSA
14	ac	14	2	(2)
	acgactgcgac	5	5	3
	actacgactgcgac	2	8	3
	actgcgac	8	12	4
	c	15	14	2
	cgac	12	15	1
0	cgactacgactgcgac	0	0	(0)
	cgactgcgac	6	1	1
	ctacgactgcgac	3	3	2
	ctgcgac	9	6	3
	gac	13	9	3
	gactacgactgcgac	1	13	4
7	gactgcgac	7	4	(4)
	gcgac	11	7	3
	tacgactgcgac	4	10	3
	tgcgac	10	11	1

図 2 SADIV での索引作成
Fig. 2 Index construction in SADIV.

索引容量は, $samp\_SA$ の容量は S おきの値で良いので十分小さくなり, これは無視するとすると dif_sSA に依存するが, この中の値は差分をとっているので小さい数字が出やすくなっており, 可変長の符号化を用いることで圧縮することができる. ここで, この符号

⋆1 本論文で log と表記したとき, 底は 2 であるとする.
⋆2 m はパターン P の長さ ($m = |P|$), occ は P の T 中での出現数.

化にはゴロム符号化[3] という符号化を用いる．結果として, dif_sSA の理論的最悪容量は $(n+S)\left(\log\frac{n}{S}+3\right)$ となる．

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いくことで全ての P の出現位置を得ることができる．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 $O(m\log\lceil n/S\rceil)$, 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 $|P|$ まで探索するため $O(mS)$, よって全体で $O(mS+|P|\log(\frac{n}{S}+1)+occ)$ となる．

3. 提 案 手 法

3.1 索引作成法

提案手法では転置インデックスと SA を組み合わせ, 頻度の高い (以下 FREQ) フレーズに対応する位置情報を転置インデックスで登録し, 頻度の低い (以下 RARE) フレーズに対応する位置情報を SA で保存する．そのために, 以下のようなことを考える

まず, テキスト T の Q-gram を考える． つまりそれらを $t_1, t_2 \ldots t_{n-Q}$ とするとき, $t_j = T[j \ldots j+(Q-1)]$ である．そして閾値 TH を定め, 長さ n の配列 A を, t_j の T 中での頻度が TH 以上ならば $A[j]=F$, t_j の T 中での頻度が TH より小さければ $A[j]=R$ と定義し, $A[j]=F$ である位置情報 i は転置インデックスで, $A[j]=R$ である位置情報 i は SA で保存する．以下では, このときの転置インデックスを FREQ_INV, SA を RARE_SA と呼ぶ．また $A[j]=F$ となる j の数を n_F, $A[j]=R$ となる j の数を n_R と定義する．

3.1.1 FREQ_INV

前述の通り, FREQ_INV には $A[j]=F$, つまり FREQ なフレーズに対応する位置情報を登録する．今回この転置インデックスは trie 木を用いて実装する．この trie 木に全ての接尾辞が登録された場合はそれは接尾辞 trie となるが, 今回はそれらの接尾辞の接頭辞の中で FREQ である最長のもの (ただし長さ Q 以上) を登録し, その接尾辞の出現位置をそのノードに保存する．このとき, 例えば “the”, “them”, “they” という 3 つのフレーズが trie に登録されていたとすると, “them”, “they” に関しては, それぞれに対応するノードにその位置情報が格納されるが, ‘the” に対応するノードには “the” の位置であり, かつ “them”, “they” でない位置情報が格納させることに注意する．また, trie の大きさは (TH が小さすぎなければ) 大きくならず, 1 つのノードが多数の位置情報をもつことになることがわかる．この trie 木のノードに登録されている位置情報の列を posting と呼ぶことにする．最後に posting 中の出現位置 (ソートされているとする) の差分をとり, ゴロム符号化を行うことで posting の圧縮を行う．

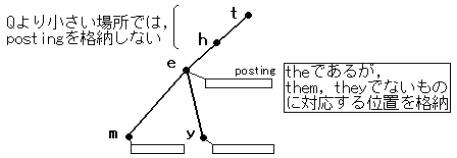

図 3 FREQ_INV (Q=3)
Fig. 3 FREQ_INV (Q=3).

3.1.2 RARE_SA

前述の通り, RARE_SA には $A[j]=R$, つまり RARE なフレーズに対応する位置情報を登録する．RARE_SA は長さ n_R の配列で, $A[SA[i]]=R$ となる $SA[i]$ の値を詰めて保存したものである．よって, その容量は $O(n_R\log n)$ となる．検索に関しても通常の SA と同様に行うことで, $A[j]=R$ である位置から始まる全てのフレーズを検索でき, その時間計算量は $O(m\log n_R+occ)$ となる．また RARE_SA は SADIV などを用いて圧縮することも可能である．

3.2 検 索 法

3.2.1 $m \geq Q$ の場合

まず初めに, 検索を行うフレーズ P の Q-gram を考え, それらを $p_1, p_2 \ldots p_{m-Q}$ とする．そしてこれらの 1 つ 1 つを INV_FREQ で検索すると[⋆1], もし登録されていれば FREQ, 登録されていなければ RARE であることがわかる．

もしその中に RARE なものが存在すれば, まず RARE_SA によりその p_i の全ての存在位置を得た後に[⋆2], P の p_i 以外の位置でも一致しているかを T を用いて逐次的にチェックする[⋆3]．このとき, このチェック (以下 LAST_CHECK と呼ぶ) は $O(m\cdot TH)$ で行える．結

⋆1 trie 木が大きくなければ, この時間は十分小さい
⋆2 実際は RARE_SA を用いて, p_i でなく, $P[i...m]$ の存在位置を得ることができる．つまり P の i 以降の部分について全てチェック可能．
⋆3 RARE_SA で調べた位置は調べなくてよい．

果としてこの場合の時間計算量は $O(m \log n_R + m \cdot TH)$ となる[⋆1].

しかし, RARE なものが存在せず全てが FREQ である場合が考えられる. その場合には INV_FREQ を用いて, P の全ての部分文字列の中で最も頻度の低いもの (P' とする) を探索し, P' の全出現位置を得る. この出現位置は trie 木の P' に対応するノードと, その全ての子孫のノードの posting を合わせたものであるので, trie 木の P' に対応するノード以下のノードを巡回することで得ることができる[⋆2]. そして先程と同様に LAST_CHECK を行うことで検索が完了する. 時間計算量は, P' の頻度を occ', また trie が十分小さく探索時間を無視できるとすると, posting の復元に $O(occ')$, LAST_CHECK は $O(m \cdot occ')$ となるので, 全体として $O(m \cdot occ')$ となる.

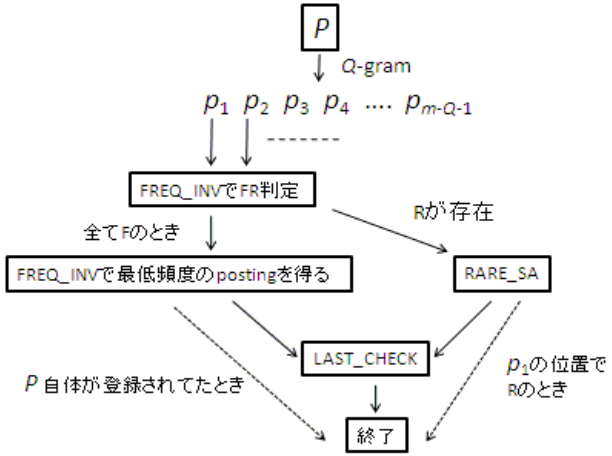

図 4 提案手法全体の流れ ($m \geq Q$ の場合)
Fig. 4 Flow of the proposed method (in case $m \geq Q$).

⋆1 RARE_SA の検索で見つかる “一致する可能性のある位置” の数 (occ' とする) は TH よりも少ないので, これを SA から得るための時間 $O(occ') = O(TH)$ は第二項に吸収される.
⋆2 P' が P の接尾辞でない場合は, P' の次の 1 文字の不一致の情報を用いることができ, 結果として巡回をする必要はなく, 対応するノードの posting のみで十分となる.

3.2.2 $m < Q$ の場合

$m < Q$ の場合は, 先程と検索法が異なる. P が FREQ だとしても, trie 木の P に対応するノードには何の位置情報も登録されておらず (posting が存在しない), そのノードの子孫を巡回することになるが, それ以外にも RARE_SA 内に P に対応する位置情報が存在する可能性がある. よって, この場合は INV_FREQ と RARE_SA の両方で検索を行うことで, 全ての出現位置を得ることができる (LAST_CHECK は必要ない).

3.3 各パラメータと容量の関係

提案手法の容量は RARE_SA のサイズ, INV_FREQ の posting のサイズ, INV_FREQ の trie 木のサイズの 3 つに分類できる. パラメータ TH を小さくすると INV_FREQ が大きく, RARE_SA は小さくなり, TH を大きくすると INV_FREQ が小さく, RARE_SA は大きくなる. パラメータ Q を小さくすると INV_FREQ が大きく, RARE_SA は小さくなり, Q を大きくすると INV_FREQ が小さく, RARE_SA は大きくなる. そこで Q を大きくすると同時に, パラメータ TH を小さくして容量を元の大きさに合わせるとすると, trie 木のサイズが大きくなる. つまり, RARE_SA のサイズ, INV_FREQ の posting のサイズ, INV_FREQ の trie 木のサイズの間にトレードオフが存在する. また RARE_SA を SADIV で圧縮する場合は, その圧縮にもパラメータ (S) が存在する.

4. 実験と考察

4.1 提案手法の容量と検索時間

他の手法との比較を行う前に, 各パラメータを変化させたときの, 提案手法の容量, 検索時間などについて調査する.

表 1 は 50M バイトの英文に対する提案手法の索引容量とその内訳で, パラメータは Q は 3 で固定で, TH を変動させている. FI:posting は FREQ_INV の posting の総容量, FI:trie は FREQ_INV の trie 木のみ容量, RSA は RARE_SA の容量を表している (RARE_SA は固定長の符号化を行っている). TH が小さいと n_R が非常に小さく, 結果的に RSA の容量が非常に小さくなり, 合計容量はほぼ FREQ_INV の posting の容量が占めている. 逆に, TH が大きいと n_R が非常に大きく, 結果的に RSA の容量が非常に大きくなり, 合計容量はほぼ RARE_SA の容量が占めている. また, trie 木のサイズは (今回調べた TH の範囲内では) 十分小さくなっている.

表 2 も 50M バイトの英文に対する提案手法の索引容量とその内訳で, パラメータは TH は 2048 で固定で, Q を変動させている. (TH 固定で) Q が 1 増えたとき, trie 木中の Q の高

表 1 50M bytes の英文での提案手法の容量とその内訳 ($Q = 3$, TH 変動)
Table 1 Index sizes of proposed method and its details in English text ($Q = 3$, change TH).

TH	n_R	FI:posting	FI:trie	RSA	合計
256	1,199,593	116.114	4.262	3.148	124.274
512	1,965,958	108.018	2.040	5.160	116.448
1024	3,216,439	99.078	0.976	8.845	110.508
2048	5,098,616	89.279	0.450	14.658	106.300
4096	8,122,730	77.871	0.196	23.352	104.466
8192	12,895,746	64.606	0.091	38.687	106.608
16384	19,867,249	49.218	0.042	62.085	113.829
32768	27,590,757	34.689	0.020	86.221	124.380
65536	35,826,960	21.184	0.009	116.437	137.632

表 2 50M bytes の英文での提案手法の容量とその内訳 ($TH = 2048$, Q 変動)
Table 2 Index sizes of proposed method and its details in English text ($TH = 2048$, change Q).

Q	n_R	FI:posting	FI:trie	RSA	合計
3	5,098,616	89.279	0.450	14.658	106.300
4	16,623,750	67.973	0.450	49.871	122.451
5	30,121,207	42.516	0.450	94.128	140.861
6	39,518,644	24.651	0.450	128.435	153.537
7	45,835,803	12.662	0.450	148.966	162.079
8	49,456,828	5.750	0.450	160.734	166.935

さにある posting はすべて RARE_SA に移動し, 結果的に, n_R が大きくなる. また posting は移動するが木の形自体は変化しないので, trie 木のサイズは変化しない.

表 3 は 50M バイトの英文での提案手法の検索時間 (*locate*) を調べたものである. ここで検索時間は, 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合計時間である[⋆1]. パラメータは $Q = 3$, TH は 512, 32768 である (表 1 より, TH512, 32768 のときの容量は同等であることがわかる). 表 3 中の FI:RSA は, 1000 回の検索中で, LAST_CHECK 前の位置情報 (posting) を FREQ_INV から得た回数と, RARE_SA から得た回数を計測したものである. *length* による FI:RSA の変化は, *length* が大きくなると, RARE な P 内に Q-gram が存在する可能性が高くなり, RARE_SA での検索回数が増加することによる.

length が大きいとき (検索フレーズ低頻度), $TH = 32768$ のほうが高速な理由は, ほとんど RARE_SA での検索割合が高く, また RARE_SA は圧縮しておらず位置情報を高速に得ることができるからと考えられる. 逆に, *length* が小さいとき (検索フレーズ高頻度), $TH = 512$ のほうが高速な理由は, FREQ_INV の検索割合が高く, P の Q-gram が全て FREQ であり, その中で一番頻度の低いものを探す必要があることが多くなるが, TH が小さい方がその 1 番小さいものの頻度 occ' が小さい (真の occ に近い) からと考えられる.

⋆1 *tot_occ* は 1000 回の検索で得られたフレーズの総出現数であり, *length* が小さいほど大きい.

表 3 50M bytes の英文での提案手法の検索時間 (locate は検索 1000 回での合計時間)
Table 3 Retreive times of proposed method in English text (locate : total of 1000 times search).

length	tot_occ	TH=512		TH=32768	
		locate (sec)	FI:RSA	locate (sec)	FI:RSA
3	101,857,226	5.85	963:37	5.80	505:495
4	36,812,591	2.16	937:63	2.94	332:668
5	16,687,917	1.02	924:76	2.22	236:764
6	4,648,718	0.36	903:97	1.64	197:803
7	1,975,328	0.20	893:107	1.34	143:857
8	1,034,485	0.16	881:119	0.75	88:912
9	645,195	0.17	859:141	0.52	62:938
10	124,734	0.14	849:151	0.34	43:957
20	48,492	0.14	753:247	0.03	3:997
50	6,073	0.36	518:482	0.00	0:1000
100	2,471	0.05	41:959	0.00	0:1000

4.2 性能比較

計算機実験により, 提案手法の性能と RPSA, SADIV の性能を比較する. また, 高頻度のフレーズの検索には時間がかかるが索引容量が小さい圧縮法との比較を行う. 具体的には FM-index を改良した af-index[1)] という実装を用いる[⋆2]. 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. また, SADIV と RPSA との比較は[5)] で行っている.

表 4 は 50M バイトの英文での結果である. 提案手法のパラメータは $Q = 3$, $TH = 2048$ である. 容量 *size* はパラメータを調整し, 提案手法, SADIV で同等となるよう調整している. 検索時間 *locate* に関しては, *length* が 9 以下では提案手法より SADIV の方が少し高速となり, 一方 *length* が 20 以上では提案手法のほうが高速となった. これは, SAVID はど

⋆2 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, 提案手法の容量と af-index の容量が近い値になる場合の $S = 4, 8$, また検索は遅いが容量が非常に小さくなる場合の $S = 64$ の 3 つの S で実験を行っている. ただし, af-index は他の索引とは異なりそれ自体が T の情報を持つ, つまり T をメモリ内にもたずに検索を行えるので, その T の 50M バイト分容量に差をつけて比較する.

表 4 50M bytes の英文での索引容量と検索時間 (locate は検索 1000 回での合計時間)
Table 4 Index sizes and retreive times in English text (locate : total of 1000 times search).

	length	tot_occ	RPSA	af-index	SADIV	提案手法
S, (Q, TH)			-	4	2048	3, 2048
size(M bytes)			126.134	142.326	106.063	106.300
locate(sec)	3	101,857,226	12.34	89.42	5.06	6.13
locate(sec)	4	36,812,591	4.63	34.33	2.08	2.42
locate(sec)	5	16,687,917	2.17	16.03	1.17	1.25
locate(sec)	6	4,648,718	0.59	4.34	0.58	0.59
locate(sec)	7	1,975,328	0.26	1.75	0.44	0.47
locate(sec)	8	1,034,485	0.16	0.89	0.39	0.44
locate(sec)	9	645,195	0.12	0.48	0.34	0.41
locate(sec)	10	124,734	0.05	0.16	0.31	0.31
locate(sec)	20	48,492	0.02	0.05	0.30	0.23
locate(sec)	50	6,073	0.05	0.12	0.28	0.13
locate(sec)	100	2,471	0.05	0.14	0.25	0.03

表 5 50M bytes の英文での索引容量と検索時間 (locate は検索 1000 回での合計時間)
Table 5 Index sizes and retreive times in English text (locate : total of 1000 times search).

	length	tot_occ	SADIV	提案手法	
S, (Q, TH, S)			16384	3, 8192, 16384	4, 8192, 16384
size(M bytes)			86.125	85.906	86.399
locate(sec)	3	101,857,226	7.99	6.53	8.47
locate(sec)	4	36,812,591	4.56	3.44	3.73
locate(sec)	5	16,687,917	3.41	2.81	2.91
locate(sec)	6	4,648,718	2.80	2.34	2.52
locate(sec)	7	1,975,328	2.69	2.33	2.55
locate(sec)	8	1,034,485	2.48	2.34	2.51
locate(sec)	9	645,195	2.50	2.27	2.50
locate(sec)	10	124,734	2.45	2.28	2.52
locate(sec)	20	48,492	2.48	2.34	2.50
locate(sec)	50	6,073	2.47	2.34	2.48
locate(sec)	100	2,471	2.23	2.24	2.25

れだけ検索フレーズの頻度が小さくても (*tot_occ* に対応) S 回の逐次検索が必要になるのに対し, 提案手法では今回 RARE_SA の圧縮を行っていないので *length* が大きいとき高速となるからと考えられる. このように, RARE_SA を圧縮しないことで (もしくは SADIV で圧縮する場合は, 比較的小さい S で圧縮), 高頻度に関しては SADIV に似た検索速度で, 低頻度なフレーズでの検索を高速化できる.

ここで, 低頻度の検索は遅くてもよいので, 高頻度の検索を高速化できないかということを考える. そのためには, RARE_SA を SADIV で圧縮するならば, TH に比べ S を大きくすればよいということが予測される. 表 4 はそれに関する実験である. 提案手法のパラメータは Q が 3 と 4, $TH = 8192$, $S = 16384$ であり, TH よりも S を大きくとっている. また SADIV のパラメータも $S = 16384$ なので, *length* が十分大きいとき, 両者の検索時間は同じになる. *length* が小さいところでは予測通り SADIV よりも高速にすることができていることがわかる[⋆1]. しかし, その差はあまり大きくはなく, 似たような検索時間といえる. また, 高頻度での検索を更に速くしようと TH, S を調節しても SADIV に対する検索時間は高速とはならなかった. これは S に対して TH を小さくすると n_R が小さくなり, RARE_SA を圧縮が全体の容量に効かなくなること, また S を非常に大きくすると RARE_SA での検索が非常に遅くなり, 全体的に検索時間が遅くなってしまうというような理由が考えられる.

以上の結果より, 提案手法は SADIV と似たような性能を持ち, SADIV よりも多くのパラメータを持つことで, より詳細にフレーズ頻度 (もしくは *length*) に対する検索時間をある程度変化させることが可能となることがわかる.

5. おわりに

本論文では, 高頻度のパタンの検索を高速に行える索引を提案し, 実験により, 以前提案した別手法[5] と似た性能で, ある程度検索時間を調整可能であることを示した.

今後の課題は, 索引作成についての改良が必要であることである.

参考文献

1) G. Navarro, V. Mäkinen, “Compressed Full-Text Indexes,” ACM Computing Surveys 39(1), article 2, pp.61, 2007
2) R. González, G. Navarro, “Compressed Text Indexes with Fast Locate,” Proc.CPM’07, pp.216-227, LNCS 4580.
3) S. W. Golomb, “Run-length encodings,” IEEE Trans. Information Theory, IT–12(3), pp.399-401
4) U. Manber, G. Myers, “Suffix arrays: a new method for on-line string searches,” SIAM J. Comput., Vol.22, Issue 5, October 1993, pp.935-948
5) 田中洋輔 小野廣隆 定兼邦彦 山下雅史 : “高速復元可能な接尾辞配列圧縮法” FIT2009 第 8 回情報科学技術フォーラム

⋆1 $Q = 4$, $length = 3$ は RARE_SA での検索も多くおこるため, SADIV より低速となっている.