Vol.2018-AL-167 No.1
2018/3/8

高頻度な繰り返し構造を含むテキストに対する省領域な動的索引

西本 崇晃[1,a)]　高畠 嘉将[2,b)]　田部井 靖生[1,c)]

概要：本論文では高頻度反復テキスト集合用の新しい圧縮動的索引を提案する．シグネチャエンコーディングとは短いパターンに対する検索速度が遅いという欠点を持つ圧縮動的索引である．本論文はシグネチャエンコーディングに枝刈りされた接尾辞木 (TST) を組み合わせることで高頻度反復テキスト集合に対して動的な更新操作と高速なパターン検索の両立を可能にし，また，この TST 索引と名付けた新しい圧縮動的索引は実験において性能の目覚ましい改善を示した．

A dynamic compressed self-index for highly repetitive text collections

Nishimoto Takaaki[1,a)]　Takabatake Yoshimasa[2,b)]　Tabei Yasuo[1,c)]

1. 導入

高頻度反復テキスト集合 (highly repetitive text collection) とはどのテキストの組み合わせでもわずかな編集によって他方に変換できるような類似したテキストの集合であり，そのような集合はいくつかの分野で見かけることができる．例として，同種のゲノム配列，リポジトリ内のバージョン管理された文書やソースコードなどである．また，ヒトゲノム配列では，個々の差異は 0.1%程度であり，1000 human genomes project [1] のような巨大なヒトゲノム集合が存在する．他にも，バージョン管理された文書集合の例としてウィキペディアが該当する．このような巨大な高頻度反復テキスト集合を効率よく処理する需要が近年ますます高まっている．

自己索引 (self-index) とはテキストに対してランダムアクセスとパターン検索ができるデータ構造である．高頻度反復テキストに対する自己索引はこれまでに多く提案されてきた．RLFM 索引 (RLFM-index) [2] は，連長符号化した Burrows-Wheeler 変換 (BWT) を用いた自己索引で，省領域しか用いないオンライン構築と各種操作を高速に行うことができる，近年提案された革新的な自己索引である．高頻度反復テキスト用の既存の自己索引は，省領域で構築可能で，各種操作を高速に行うことができるが，ほとんどはテキストの編集に対する索引の更新（動的更新）をサポートしていない． したがって，そのような自己索引を開発することが重要である．

ESP 索引 (ESP-index) [3] とシグネチャエンコーディング (signature encoding) [4] はともに局所一致分解 (locally consistent parsing) を用いた高頻度反復テキスト集合用の自己索引である．これらの索引はオンライン構築や動的更新をサポートしているものの短いパターンの検索が遅いという欠点がある．より具体的には，ESP 索引はテキストとパターンを導出する構文木を局所一致分解を用いて構築し，パターンの構文木を用いてテキストの構文木中に出現しているパターンの候補をトップダウン検索で列挙してから，実際のパターンの出現を求める．一方シグネチャエンコーディングは二次元の範囲検索クエリを用いて直接パターンの出現を求める．このような検索方法はパターンが短い場合にはパターンの候補が増大するなど範囲検索に割く時間が無視できなくなるという欠点を持つ．

本論文では，高頻度反復テキスト用の枝刈りされた接

[1] 革新知能統合研究センター：RIKEN Center for Advanced Intelligence Project
[2] 九州工業大学：Kyushu Institute of Technology
[a)] takaaki.nishimoto@riken.jp
[b)] takabatake@ai.kyutech.ac.jp
[c)] yasuo.tabei@riken.jp

尾辞木 (truncated suffix tree (TST)) に基づいた新しい索引を提案する．本論文ではこれを TST 索引と名付けた．TST 索引は TST [5] のアイデアを用いることにより局所一致分解を使った索引の短いパターンの検索を高速化できる．提案索引は入力テキストから構築される TST と TST によって変換されたテキスト上の自己索引からなり，索引の更新も容易である．また，本論文の最後ではベンチマーク用の高頻度反復テキストを用いた実験を行い，既存の ESP 索引より検索が高速であることを示し，加えていくつかのデータでは RLFM 索引よりも使用領域が少なく，同等の検索速度であったことを示す．

2. 準備

Σ を整列したアルファベット，$\sigma = |\Sigma|$ とする．T, P を Σ 上の文字列とし，T を入力テキスト，P を検索クエリのパターンとする．さらに $N = |T|, m = |P|$ とする．$T = xyz$ を満たす文字列 x, y, z をそれぞれ接頭辞，部分文字列，そして接尾辞と呼ぶ．二つの文字列 x, y に対して $x \cdot y$ と書いたとき，x と y を連結させた文字列を表す，すなわち，$x \cdot y = xy$.

ε を長さ 0 の空文字列，$\Sigma^+ = \Sigma^* - \{\varepsilon\}$ とする．任意の $1 \le i \le j \le |T|$ について $T[i]$ は i 番目の文字を，$T[i..j]$ は i 番目から j 番目の文字列を表す．また便宜上 $T[i..] = T[i..|T|]$, $T[..i] = T[1..i]$ と略す．文字列 T について，挿入操作と削除操作を次のように定義する．$insert(T, i, K) = T[..i-1] \cdot K \cdot T[i..]$, $delete(T, i, k) = T[..i-1] \cdot T[i+k-1..]$. ここで K は挿入文字列．$Occ(P, T)$ は文字列 T 中の P の全ての出現位置の集合を表す．また，$occ = |Occ(P, T)|$. 同様に任意の文字 c について，$cOcc(c, T) = Occ(c, T)$ とする．長さ q の文字列を q グラムと呼ぶ．Σ_T^q は T に含まれている q グラムと長さ q 未満の接尾辞の集合を表す，すなわち，$\Sigma_T^q = \{T[i..\min\{i+q-1, |T|\}] \mid i \in 1 \le i \le |T|\}$. アルファベット Σ 上の文字列 T 中の隣り合う文字が全て異なるとき，T を $|\Sigma|$ 色シーケンスと呼ぶ．

T の分解 $f_1, \ldots, f_d$ とは $f_1 \cdots f_d = T$ が成り立つような文字列の列である．このとき，各文字列はファクターと呼ばれ，d を分解のサイズとする．T の連長符号化とは同じ文字からなる最長の各部分文字列が各ファクターを表すような T の分解である．このとき k 個の文字からなる文字 a を a^k と表す．たとえば $T = aabbbbbabb$ としたとき，$RLE(T) = a^2, b^5, a^1, b^2$ と分解される．また，$RLE(T)$ は各ファクターを文字とみなしたとき $|RLE(T)|$ 色シーケンスと扱うことができる．

T の自己参照無し LZ77 分解 [6] とは以下の性質を満たす T の分解 $\mathsf{LZ}(T) = f_1, \ldots, f_z$ である．(1) $T = f_1 \cdots f_z$ かつ $f_1 = T[1]$. (2) 各 $1 < i \le z$ について，$T[|f_1..f_{i-1}| + 1]$ が $T[|f_1..f_{i-1}|]$ に一度も出現していないならば，$f_i = T[|f_1..f_{i-1}| + 1]$. さもなければ f_i は $f_1 \cdots f_{i-1}$ に出現する $f_i \cdots f_z$ の最長の接頭辞である．

自己索引とは T に対して以下の操作が行えるデータ構造である．

- **出現回数クエリ (Count):** パターン P を受け取り T 中の P の出現回数を返す．
- **出現位置クエリ (Locate):** パターン P を受け取り T 中の P の出現位置を全て返す．
- **展開クエリ (Extract):** T の部分文字列の範囲を表す $[i..i+\ell-1]$ を受け取り $T[i..i+\ell-1]$ を返す．

このような自己索引は静的自己索引と呼ばれる．上記の三つの操作に加えてテキストに対する $insert(T, i, K)$ と $delete(T, i, k)$ 操作が可能な索引を動的自己索引と呼ぶ．本論文では挿入や削除する部分文字列の長さを k と表記し，T の取りうる最大の長さを $M \ge N$ とする．

計算モデルはワード長 $W = \Omega(\log_2 M)$ の RAM モデルとする．空間複雑性はワードの数で評価する．また，それに $\log_2 M$ を掛けることでビット単位の評価とする．

3. 既存研究と結果

高頻度反復テキスト集合の自己索引に関する研究は活発な分野であり多くの手法がこれまでに提案されている．表 1 は既存手法と本論文の主結果を並べたものである．高頻度反復テキスト集合の動的な自己索引が重要にも関わらず，高圧縮率を維持したまま高速な検索クエリと動的な更新をサポートしているのは無い．本論文が提案する動的な自己索引である TST 索引はその両方を実現し実用的である．その理論的な性能は次の章以降で解説するが，結果として以下の定理を得る．

定理 1 パラメータ q と文字列 T を与えられたとき，T の TST 索引の使用領域は $O(w') = O(z(q^2 + \log N \log^* M))$ ワード領域であり，次の 4 つの操作をサポートする．(i) 長さ q 以下のパターンの出現回数クエリを $O(m(\log\log\sigma)^2)$ 時間，(ii) 出現位置クエリを $O(m(\log\log\sigma)^2 + occ \log N)$ 時間，(iii) 展開クエリを $O(\ell + \log N)$ 時間，(iv) 更新操作を $O(f_{\mathcal{B}}(k+q+\log N \log^* M) + (k+q)q(\log\log\sigma)^2)$ 時間．ここで $f_{\mathcal{B}} = f(w', M)$ かつ $f(a, b) = O(\min\{\frac{\log\log b \log\log a}{\log\log\log b}, \sqrt{\frac{\log a}{\log\log a}}\})$ とする．

4. 枝刈りされた接尾辞木を用いた高速なクエリ

この章では q-TST 変換と名付けた新たなテキスト変換について述べる．この変換を用いることで短いパターンだったときの自己索引の検索時間を改善することができる．最初に枝刈りされた接尾辞木 (TST) を導入し，次にそれを用いた q-TST 変換のアイデアを説明する．

表 1 高頻度反復テキスト集合用の自己索引の要約．n は T を導出する文脈自由文法のサイズ，r は T の BWT を連長符号化したときの長さ，$\hat{z} \leq z$ は T を自己参照あり LZ77 分解したときのファクター数，$M \geq N$ は T がとり得る最大の長さ，$0 < \epsilon \leq 1$，$0 < \epsilon' < 1$，$0 < \epsilon''$ は任意の定数，$f_{\mathcal{A}} = f(z \log N \log^* M, M)$，$f_{\mathcal{B}} = f(z(q^2 + \log N \log^* M), M)$，$occ_c \geq occ$ はパターンの出現候補の数．

索引	ワード領域	更新時間
RLFM 索引 [2]	$O(r)$	非対応
Bille ら [7]	$O(\hat{z} \log(N/\hat{z}))$	非対応
BT 索引 [8]	$O(z \log(N/z))$	非対応
SLP 索引 [9]	$O(n)$	非対応
signature encoding [4]	$O(z \log N \log^* M)$	$O((k + \log N \log^* M) \log z \log N \log^* M)$(平均)
静的 TST 索引 (本研究)	$O(z(q + \log N \log^* N))$	非対応
動的 TST 索引 (本研究)	$O(z(q^2 + \log N \log^* M))$	$O(f_{\mathcal{B}}(k + q + \log N \log^* M) + (k + q)q(\log \log \sigma)^2)$

索引	検索時間
RLFM 索引 [2]	$O(m \log \log_W(\sigma + N/r) + occ \log \log_W(N/r))$
Bille ら [7]	$O(m(1 + \frac{\log^{\epsilon'} \hat{z}}{\log(N/\hat{z})}) + occ(\log \log N + \log^{\epsilon'} \hat{z}))$
BT 索引 [8]	$O(m^2 \log(N/z) + m \log^{\epsilon''} z + occ(\log \log N + \log^{\epsilon''} z))$
SLP 索引 [9]	$O(\frac{m^2}{\epsilon} \log(\frac{\log N}{\log n}) + (m + occ) \log n)$
signature encoding [4]	$O(m f_{\mathcal{A}} + occ \log N + \log z \log m \log^* M (\log N + \log m \log^* M)$
静的 TST 索引 (本研究)	$O(m + occ)$ $(m \leq q)$ $O(m + occ_c \log m \log^* N \log N)(m > q)$
動的 TST 索引 (本研究)	$O(m(\log \log \sigma)^2 + occ \log N)$ $(m \leq q)$ $+(k + q)q(\log \log \sigma)^2)$

4.1 トライ木，コンパクトトライ木および枝刈りされた接尾辞木

文字列集合 F のトライ木 $\mathcal{X}$ とは枝に文字ラベルがついた根付き木であり，F 中の文字列の全ての接頭辞を各ノードがそれぞれ表している．図 1 の上図はトライ木の例である．$\mathcal{X}$ 中のノードの集合と葉の集合をそれぞれ U と U_L する．$\mathcal{X}$ に対して次の操作を定義する．ここで，u, v はノード，P は文字列，c は文字とする．

- $path_{\mathcal{X}}(u)$: ノード u が表す文字列を返す，すなわち根から始まりノード u で終わるパスのラベルが連結した文字列を返す．
- $locus_{\mathcal{X}}(P)$: $path_{\mathcal{X}}(u) = P$ となるようなノード u があれば返す．
- $leave_{\mathcal{X}}(P)$: P を接頭辞として持つ葉の集合を返す．
- $child_{\mathcal{X}}(u, c)$: $path_{\mathcal{X}}(u) \cdot c = path_{\mathcal{X}}(v)$ を満たす u の子 v があればそれを返す．
- $slink_{\mathcal{X}}(u)$: $path_{\mathcal{X}}(v) = path_{\mathcal{X}}(u)[2..]$ を満たす v があればそれを返す．

$\mathcal{X}$ 中のノードには暗黙的 (implicit) ノードと明示的 (explicit) ノードの二つのタイプが存在する．暗黙的ノードとは子がただ一つの内部ノード．それ以外は全て明示的ノードである．

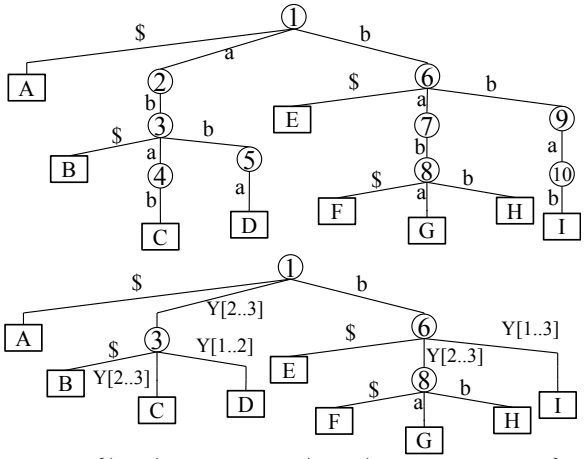

図 1 $F = \{\$, ab\$, abab, abba, b\$, bab\$, baba, babb, bbab\}$ のトライ木 (上) と参照文字列として $Y = bab\$$ を用いたコンパクトトライ木 (下)．

$expl_{\mathcal{X}}(u)$ はノード u の先祖の中で最も深い明示的なノード v を返す．次にこれらの操作の計算時間について述べる．$expl_{\mathcal{X}}(u)$, $locus_{\mathcal{X}}(P)$, $path_{\mathcal{X}}(u) = P$ および $leave_{\mathcal{X}}(P)$ はそれぞれ $O(1)$, $O(|P|g)$, $O(|P|)$ および $O(|P|g + |leave_{\mathcal{X}}(P)|)$ 時間で計算できる．ここで g は $child_{\mathcal{X}}(u, c)$ の計算時間である．$O(|U_{exp}|)$ ワード領域の完全ハッシュ [10] を用いると $g = O(1)$ となる．さらに $slink_{\mathcal{X}}(u) = v$ は u が v へのポインタを持っておけば定数時間で計算できる．

次にコンパクトトライ木について定義する．コンパクトトライ木とはトライ木 $\mathcal{X}$ の連続する暗黙的ノードをつぶして一つの枝にした，空間効率の良い表現である．よってコンパクトトライ木のノードは明示的ノードしかなく，枝のラベルは文字ではなく文字列である．ただし，枝の文字列をそのまま保持する場合はトライ木と使用領域がほとんど変わらないので，ある文字列 Y を用いた参照文字列によるコンパクトトライ木を考える．これは枝にラベルの文字列をそのまま持たせる代わりに，そのラベルと同等な Y の部分文字列を表す二つの整数を持たせたコンパクトトライ木である．このとき，コンパクトトライ木のサイズは $O(|U_{exp}| + |Y|)$ ワード領域である．図 1 の下図はコンパクトトライ木の例である．

本論文では参照文字列付きのコンパクトトライ木を静的な状況で，参照文字列を用いないコンパクトトライ木は，動的な状況で用いる．よって，コンパクトトライ木への文字列 K の挿入と削除を考える．これは $O(|K|\hat{g})$ 時間で可能である [11]．ここで $\hat{g}$ はあるノードに子を挿入，または削除した場合の $child_{\mathcal{X}}(u, c)$ のデータ構造の更新時間である ($g \leq \hat{g}$ と仮定)．ここで Beame と Fich の predecessor/successor のデータ構造 [12] を用いると，追加領域は $O(|U_{exp}|)$ ワード領域で，$g, \hat{g} = f(u', \sigma) = O((\log \log \sigma)^2)$ となる．ここ

で u' は u の子の数である．

次に q-枝刈りされた接尾辞木 (q-TST) を定義する．文字列 T の q-TST とは Σ_T^q に対するトライ木である．このとき各葉 u の $slink_{\mathcal{X}}(u)$ は必ず存在する．Vitale らはコンパクトトライ木で表現された q-TST の参照文字列 Y はたかだか長さ $O(|\Sigma_T^q|)$ で表現できることを示した．また，このような Y はコンパクトトライ木とともに $O(|T|\hat{g})$ 時間と $O(|\Sigma_T^q|)$ ワード作業領域でオンライン構築できる [13]. 同時に，各葉の $slink_{\mathcal{X}}(u)$ を求めて保持できる．

4.2 q-TST 変換

テキスト T の q-TST 変換とそれを用いた検索のアイデアについて述べる．q-TST 変換とは q-TST を用いて T の各 q グラム P を $locus_{\mathcal{X}}(P)$ によって対応する葉 (を表す ID) に置き換えた新しい文字列 T_q への変換である．厳密に，$T_q = C(T,1)\cdots C(T,|T|)$．ここで $C(T,i) = locus_{\mathcal{X}}(T[i..\min\{i+q-1,|T|\}])$．つまり，$C(T,i)$ は i 番目から始まる q グラムを表す葉である．同様に，T の q-TST を用いて P の変換後の文字列 $P_q = C(P,1)\cdots C(P,|P|-q+1)$. T_q と違い，P_q は必ずしも計算できるとは限らない．この二つの定義より，次の補題が成り立つ．

補題 1

$$Occ(P,T) = \begin{cases} \bigcup_{c\in leave_{\mathcal{X}}(P)} cOcc(c,T_q) & (|P|\le q) \\ Occ(P_q,T_q) & (|P|>q) \end{cases}$$

が成り立つ．ただし，P_q が計算できないとき，つまり P が T にはない q グラムを含んでいるときは，$Occ(P,T)=\phi$ が成り立つ．

証明 1 省略.

$leave_{\mathcal{X}}(P)$ は q-TST を用いて $O(|P|g+|leave_{\mathcal{X}}(P)|)$ 時間で計算できる．P_q は $slink_{\mathcal{X}}$ と $child_{\mathcal{X}}$ 関数を使って $O(|P|g)$ 時間で計算できる．なぜなら任意の $q\le i<m$ における P_q 上の隣り合う葉 $u=C(P,i+1)$, $v=C(P,i)$ について，$u = child_{\mathcal{X}}(slink_{\mathcal{X}}(v), P[i+q])$ が成り立つからである．

補題 1 は q より短いパターンの出現位置クエリを自己索引と q-TST を用いることで $O(|P|g+c_1\times|leave_{\mathcal{X}}(P)|)$ 時間で計算できることを示している．ここでの c_1 は自己索引を用いて $cOcc$ を計算する時間である．つまり，自己索引の検索時間が上記の計算時間より大きい場合は，q-TST を併用することで高速化できる．補題 1 を使って併用する場合，パターンの長さが q 以下の場合はまず $leave_{\mathcal{X}}(P)$ を求める．このとき，各葉の T_q 上での出現位置が P の出現位置を表している．パターンが q より長い場合は q-TST を用いて P_q に変換する．そして T_q 中の P_q の出現を自己索引を使って探す．このときの出現位置が P の出現位置に対応している．よって一般的な自己索引を q-TST と併用させた場合，以下の定理がなりたつ．

定理 2 $\mathcal{I}(T)$ を $cOcc(P,T)$ と $Occ(P,T)$ をそれぞれ $O(c_1)$ 時間と $O(c_2)$ 時間で計算できる索引とする．このとき，長さ q 以下の P に対する $Occ(P,T)$ を $O(m+c_1\times|leave_{\mathcal{X}}(P)|)$ 時間, q より長いときの $Occ(P,T)$ を $O(m+c_2)$ 時間で検索できる $O(|\Sigma_T^q|+|\mathcal{I}(T_q)|)$ ワード領域の索引が存在する．ここで $|\mathcal{I}(T)|$ は $\mathcal{I}(T)$ のサイズである．

本論文ではこの結果をシグネチャエンコーディングと呼ばれる自己索引に用いる．その結果得られる索引を TST 索引と呼ぶ.

5. TST 索引

TST 索引は定理 2 とシグネチャエンコーディングを組み合わせたものである．この章ではまずシグネチャエンコーディングとその中で用いられている局所一致分解について説明し，その後 TST 索引について述べる．

5.1 局所一致分解とシグネチャエンコーディング

局所一致分解は与えられた文字列 T を T から得られるある二値文字列 $\tau(T)$ を用いた T のある分解である．説明のために，p_i を $\tau(T)$ 中の i 個目の 1 の位置とする．そのとき，$\tau(T)$ と p_i は次の性質を満たす．

補題 2 ([14]) c 色シーケンスの文字列 T に関して，同じ長さでかつ以下の性質を満たす二値文字列を返す関数 $\tau(T)$ が存在する．

- $1\le i\le d$ について $2\le p_{i+1}-p_i\le 4$ かつ $p_1=1$．ここで d は $\tau(T)$ に含まれる 1 の数, すなわち $d=|cOcc(1,\tau(T))|$. また，$p_{d+1}=|T|+1$ とする．
- 任意の i と j について $T[i-\Delta_L..i+\Delta_R]=T'[j-\Delta_L..j+\Delta_R]$ が成り立つならば, $\tau(T)[i]=\tau(T')[j]$ が成り立つ．ここでの T と T' は c 色シーケンスであり，$\Delta_L=\log^* c+6$ かつ $\Delta_R=4$.

c 色シーケンス T の局所一致分解 $LC_c(T)$ は p_i を用いて次のように定義する．$LC_c(T)=T[p_1..p_2-1],\ldots,T[p_d..p_{d+1}-1]$.

文字列 T のシグネチャエンコーディング (signature encoding) [14] とは単一の文字列 T を導出するある文脈自由文法 $\mathcal{G}=(\Sigma,\mathcal{V},\mathcal{D},S)$ であり，この文法は局所一致分解と連長圧縮を交互に適用して得られる．ここで，$\mathcal{V}=\{e_1,\ldots,e_w\}$ は変数と呼ばれる正の整数の集合，$\mathcal{D}=\{e_i\to f_i\}_{i=1}^w$ は生成規則の集合で，各 f_i は変数の列か Σ に含まれる文字である．S は T を導出する開始記号.

シグネチャエンコーディングは T から構築される高さ $O(\log N)$ の平衡なある導出木と一致し，各内部ノードは V の変数か Σ の文字と一致する．この導出木は各高さでのノード列が積み重なったノード列の列とみなすことができる．このとき，一番下のノード列から順に $SE_0^T,\ldots,SE_h^T$ とおくと，各 $0\le t\le h$ における SE_t^T は次のように定義される．

- $SE_t^T = Assgn^+(T[1], \ldots, T[|T|])$ $(t = 0)$,
- $SE_t^T = Assgn^+(LC_{4M}(SE_{t-1}^T))$ (t は 0 でない偶数),
- $SE_t^T = Assgn^+(RLE(SE_{t-1}^T))$ (t は奇数).

このとき, h とは $|SE_h^T| = 1$ を満たす最小の正の整数であり, これは導出木の一番上の変数列であるから $S = SE_h^T[1]$ である．また, $Assgn^+$ は変数列もしくは文字の列 $f_1, f_2, \ldots, f_d$ を受け取ってそれぞれに対応する変数の列 $e_1, e_2, \ldots, e_d$ を返す関数である，すなわち，各 $1 \le j \le d$ について $(e_j \to f_j) \in \mathcal{D}$ を満たす e_j を返す．図 2 はシグネチャエンコーディングの例である．

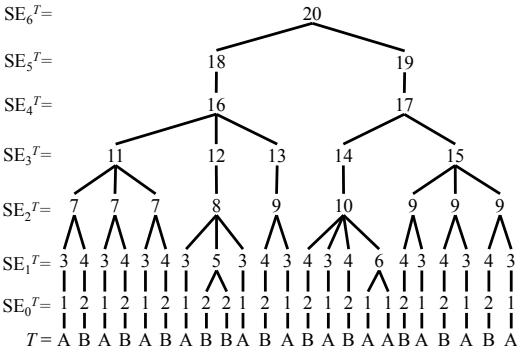

図 2 シグネチャエンコーディングの導出木.

$w = |\mathcal{V}|$ とおき，これを $\mathcal{G}$ のサイズとする．このとき $w = O(\min(z \log N \log^* M, N))$ が成り立つ [15].

$\mathcal{G}$ は定義より次の性質を満たす．(1) 各偶数 t にて $|SE_t^T| \le \frac{1}{2}|SE_{t-1}^T|$ となるので $h = O(\log N)$．(2)$4M$ 色シーケンスを満たす SE_t^T 各に対して局所一致分解が適用されるので各変数の値は高々 $4M$ でなければならない．(3) 生成規則の右側はある変数の繰り返しか，たかだか長さ 4 の変数列か，もしくは文字である．よって生成規則は $O(1)$ ワード領域で表現できるので，$\mathcal{G}$ は $O(w)$ ワード領域で表現できる．

5.2 静的 TST 索引

シグネチャエンコーディングを用いて自己索引を作ることができる．そのような自己索引は [3], [4] などがある．基本的なアイデアとして，T 中に部分文字列として存在する P は導出木上で長さ $O(\log |P| \log^* M)$ の共通のある変数列で表現されている．これは P のコアと呼ばれ，$\mathcal{G}$ を用いて P から計算可能である．したがって，P の代わりに導出木を展開しながら P のコアを探す．コアは P より短いので高速に検索することができる．この性質を用いて次の補題が得られる．

補題 3 (ESP 索引 [3]) シグネチャエンコーディングに関して，出現回数クエリと出現位置クエリを $O(m + occ_c \log m \log^* M \log N)$ 時間，$cOcc(c, T)$ を $O(occ)$ 時間，展開クエリを $O(\ell + \log N)$ 時間で計算できる $O(w)$ ワード領域の索引が存在する．ここで $occ_c \ge occ$ は索引による P の出現候補の数である．

証明 2 省略.

T_q から構築したシグネチャエンコーディングを q-シグネチャエンコーディングと呼ぶことにする．本手法の静的自己索引は T の q-TST $\mathcal{X}$ と q-シグネチャエンコーディングで構成されており，T から $\mathcal{X}$ と T_q を構築し，次に q-シグネチャエンコーディングを構築，そして最後に補題 3 の索引を構築する．検索アルゴリズムは定理 2 に基いており，P が短い場合は q-TST で出現位置を探し，長い場合は P_q を求めて補題 3 のクエリを使って出現位置を探す．よって定理 2 と補題 3 から以下の定理が得られる．

定理 3 (i) 長さ q 以下のパターンに対して出現位置クエリを $O(m + occ)$ 時間，(ii)q より長いパターンに対して出現回数クエリと出現位置クエリを $O(m + occ_c \log m \log^* M \log N)$ 時間，(iii) 展開クエリを $O(\ell + \log N)$ 時間で計算できる $O(|\Sigma_T^q| + w')$ ワード領域の索引が存在する．ここでの w' は q-シグネチャエンコーディングのサイズとする．

また，次の定理から w' の上界が得られる．

定理 4 T の q-シグネチャエンコーディングのサイズ w' について，$w' = O(z(q + \log N \log^* M))$ が成り立つ．

証明 3 省略.

さらに，$|\Sigma_T^q| = O(zq)$ である [16]．よって本手法の索引のサイズは $O(z(q + \log N \log^* M))$ ワード領域である．この自己索引の実験的評価は 6 章にて述べる．

5.3 動的 TST 索引

動的テキスト T に対して本手法の自己索引を効率よく更新する方法について述べる．本手法の自己索引は T の q-TST と T_q のシグネチャエンコーディングで構成されている．動的な場合においてもこの自己索引は定理 2 に基いているが，補題 3 で与えられる索引の代わりに以下の動的索引を利用する．

補題 4 (動的シグネチャエンコーディング [17]) $\mathcal{G}$ に $O(w)$ ワード領域のデータ構造を追加することで，T への編集操作を $O(f_{\mathcal{A}}(k + \log N \log^* M))$ 時間で行うことができる．このデータ構造はさらに $cOcc(c, T)$ と展開クエリをそれぞれ $O(occ \log N)$ 時間と $O(\ell + \log N)$ 時間で行うことができる．

したがって，本手法の動的索引は長さ q 以下のパターンに対して出現位置クエリを $O(mg + occ \log N)$ 時間で計算できる．出現回数クエリを効率よく計算するために，各 $v \in U_{exp}$ に対して $|Occ(path_{\mathcal{X}}(v), T)|$ をもたせる．これで q 以下のパターンに対して出現回数クエリを $O(mg)$ 時間で計算できる．また各 $v \in U_L$ に $slink_{\mathcal{X}}(v)$ の値を持たせる．

あとは T が変更されたときにどうやってこのデータ構造を効率よく更新させるかである．文字列 T と整数 i, j について，$C(T, i, j) = C(T, i) \cdots C(T, j)$ とする．すなわち，$C(T, i, j) = T_q[i..j]$ である．このとき，文字

列 $x, y, z \in \Sigma^*$ について，$s = x[|x| - q + 1]z[1..q]$ と $s' = x[|x| - q + 1] \cdot y \cdot z[1..q]$ とおいたとき，次の式が成り立つ．(1)$C(xz, 1, |xz|) = C(x, 1, |x| - q + 1)C(s, 1, |s| - q + 1)C(z, 1, |z|)$，(2)$C(xyz, 1, |xyz|) = C(x, 1, |x| - q + 1)C(s', 1, |s'| - q + 1)C(z, 1, |z|)$．

上記の式は T の編集に対して T_q が部分的にしか変化しないことを意味している．より具体的には，$x = T[1..i-1]$, $y = K$，$z = T[i..|T|]$ として T を $insert(T, i, K)$ で編集することを考える．このとき $T_q = C(xz, 1, |xz|)$ は上記の式より $C(xyz, 1, |xyz|)$ となる．同様に $x = T[1..i-1]$, $y = T[i..i+k-1]$, $z = T[i+k..|T|]$ として T を $delete(T, i, k)$ で編集することを考える．このときもまた，$T_q = C(xyz, 1, |xyz|)$ は上記の式より $C(xz, 1, |xz|)$ となる．したがって，$T = xz$ とおいて x と z の間に y を挿入したとき，T_q 上の文字列はたかだか長さ $|y| + 2q$ の部分文字列が変更されるのみである．これは T_q の各文字が q-TST の葉であり，T 上の同じ位置の q グラムに対応していることを考えれば当然といえる．よって T の挿入や削除操作がなされたとき，定数回の挿入削除操作によって T_q を正しく更新できる．

これを踏まえて，本手法の自己索引の更新アルゴリズムを述べる．$insert(T, i, K)$ が行われたとき，以下のようにして $\mathcal{X}$ と $\mathcal{G}$ を更新する．(i) $\mathcal{X}$ に s' に含まれている q グラムを全て追加する．(ii) $\mathcal{X}$ を用いて $C(s', 1, |s'| - q + 1)$ を計算する．(iii) T_q に $C(s', 1, |s'| - q + 1)$ を追加して T_q から $C(s, 1, |s| - q + 1)$ を取り除く．(iv) $\mathcal{X}$ から使われなくなった q グラムを取り除く．$delete(T, i, k)$ に対しても同様に更新できる．(iii) は補題 4 を用いて $O(f_{\mathcal{B}}(k+q+\log N \log^* M))$ 時間で更新できる．また，使われなくなった q グラムは V を調べることで検出できる．これは $\mathcal{G}$ は更新によって使われなくなった変数を取り除くからである．

次に，q-TST $\mathcal{X}$ をどうやって更新するかについて述べる．$\mathcal{X}$ に $|Occ(path_{\mathcal{X}}(v), T)|$ などの付加的な情報を持たせていないとき，$O((k+q)q\hat{g})$ 時間で挿入や削除が行える．したがって，更新時間を増やすことなしにこれらの付加的な情報も更新したい．編集によって T 中に新しい q グラム P が生成されたとき，q-TST に P を追加して，根から $u = locus_{\mathcal{X}}(P)$ までの P のパス上の各明示的なノードの $|Occ(path_{\mathcal{X}}(v), T)|$ を更新する．これは $O(|P|\hat{g})$ 時間でできるので更新時間を増やすことはない．もし u が挿入によって新しく出来たノードならば，u は葉なので $slink_{\mathcal{X}}(u)$ を計算して付加させる必要がある．これは $locus_{\mathcal{X}}(P[2..])$ を計算することで $O(|P|\hat{g})$ 時間でできる．また，(1) の更新アルゴリズムによって $locus_{\mathcal{X}}(P[2..])$ は必ず存在することを強調しておく．q-TST から q グラムが削除されたときも同様な手順で付加的な情報を更新時間を増やすこと無く更新できる．したがって (iii) 以外の計算時間は $O((k+q)q\hat{g})$ 時間である．

表 2 各テキストに対する索引のサイズ (MB)．TST 索引は q-TST を表記している．q は q グラムの長さ．

	DNA	english 200MB	einstein en.txt	einstein de.txt	Escherichia Coli
Text	385	200	445	88	107
ESP	438	248	2	1	42
RLFM	2,429	760	3	1	146
4-TST	501	388	8	3	48
8-TST	826	1,987	27	10	87
16-TST	23,927	10,615	48	17	1,794
32-TST	32,287	13,199	69	24	2,292

	cere	influenza	para	world leaders
Text	439	147	409	44
ESP	47	23	60	5
RLFM	124	31	164	6
4-TST	54	24	71	13
8-TST	94	37	124	47
16-TST	1,417	277	1,960	88
32-TST	1,876	762	2,835	155

最終的に $|U| = O(q|\Sigma_T^q|) = O(zq^2)$，補題 4 と定理 4 より，定理 1 が得られる．

6. 実験

この章では静的な場合での本手法の TST 索引をベンチマーク用の高頻度反復テキスト集合のデータセットを用いて評価する．ベンチマーク用のデータセットは Pizza & Chili コーパス (`http://pizzachili.dcc.uchile.cl`) から 9 つの高頻度反復テキストである DNA，english.200MB，einstein.en.txt，einstein.de.txt，Escherichia_Coli，cere，influenza，para そして world_leaders を用いた．評価方法として，ベンチマークテキストからそれぞれ長さ $m = \{4, 8, 16, \ldots, 2048\}$ の部分文字列をランダムに選び，検索クエリに用いた．評価指標として索引のメモリ使用領域と出現回数クエリと出現位置クエリの検索時間を使った．実験環境は 256GB のメインメモリと 4 コアの CPU を持つ Intel(R) Xeon(R) E5-2680 v2 (2.80 GHz) を用いた．

本手法の TST 索引を ESP 索引 [3] と RLFM 索引 [2] の二つの自己索引と比較した．TST 索引は ESP 索引の改良を目的としているので，ESP 索引の性能は良い指標となる．一方，RLFM 索引は高頻度反復テキスト集合用の自己索引の中で優秀な性能を持った，最先端の自己索引である．本手法の TST 索引は q-TST と ESP 索引を組み合わせたものであるが，これらは C++で実装した．またパラメータ q は {4,8,16,32} の値を用いて本手法の索引の性能を調べた．ESP 索引 (`https://github.com/tkbtkysms/esp-index-I`) と RLFM 索引 (`https://github.com/nicolaprezza/r-index`) は URL にある実装を用いた．

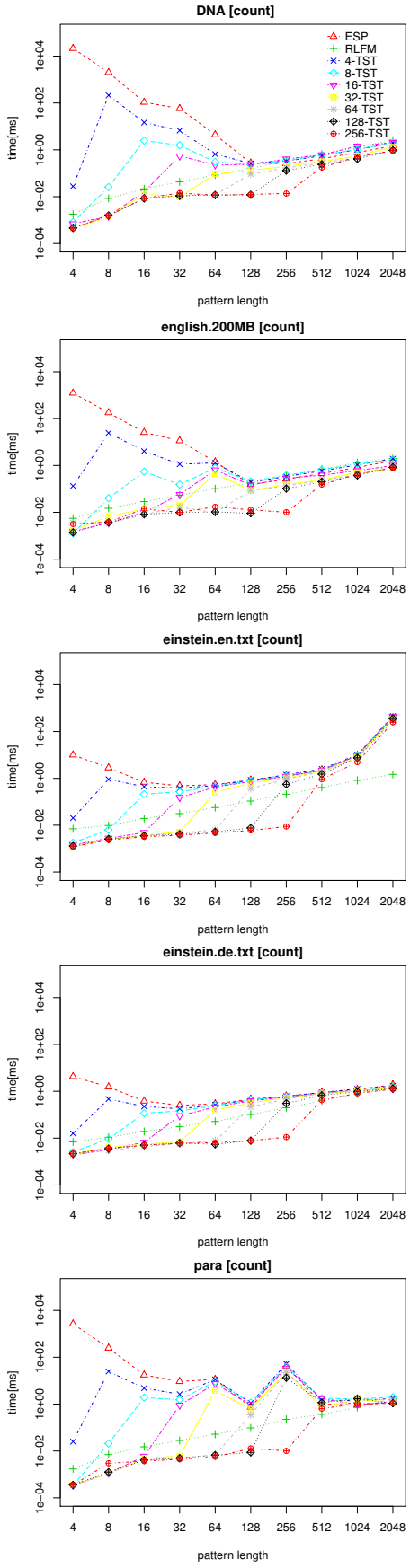

図 3 各ベンチマークテキストの出現回数クエリの計測.

図 3-5 は各手法の出現位置クエリと出現回数クエリの計算時間を示している．ただし，紙面の都合により一部を省

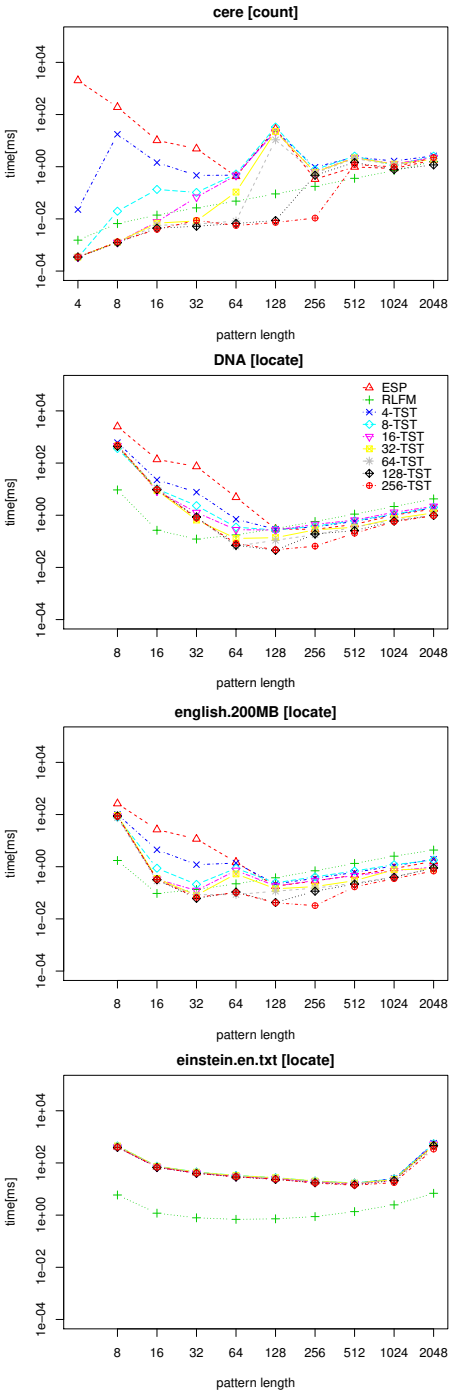

図 4 各ベンチマークテキストの出現回数クエリと出現位置クエリの計測.

略している．加えて，表 2 は各手法の索引の使用領域を示している．TST 索引は q の大きさと比べて指数的に増加している．これは q グラムの種類が指数的に増加しているからである．実用上な観点から q はたかだか 8 までにとどめる必要がある．

TST 索引は ESP 索引より高速で，パターンの長さがたかだか 64 までは効果的である．これは q-TST と ESP 索引を組み合わせることが効果的であることを示している．結果を見る限り TST 索引は出現位置クエリよりも出現回数クエリのほうが高速である．特に DNA のファイルでは

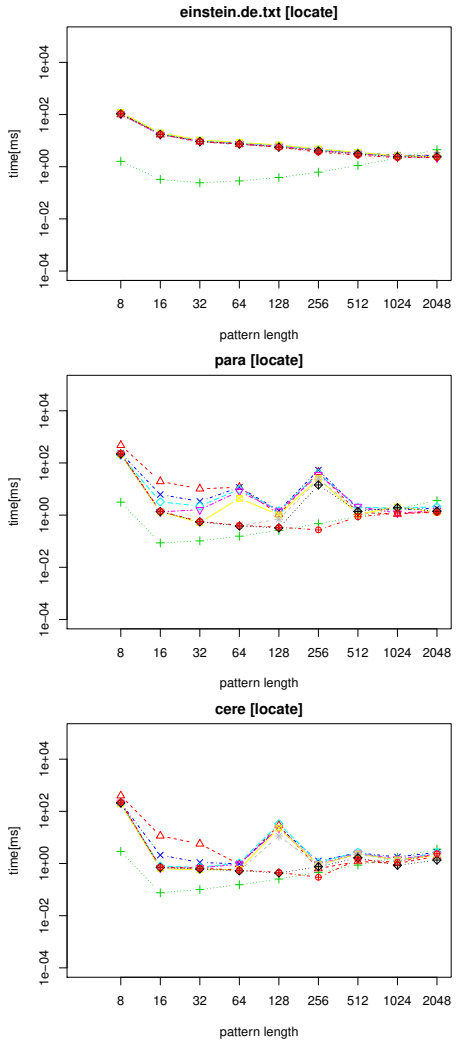

図 5 各ベンチマークテキストの出現位置クエリの計測.

パターンの長さが 8 のときでは，TST 索引は ESP 索引よりも約一万倍高速だった．しかし出現位置クエリに関してはたかだか二倍ほどしか高速化していない．出現位置クエリの計算の改善が小さい理由として，ESP 索引の $cOcc$ 関数の計算が遅いことが考えられる．これは ESP 索引の実装を修正することで効率を上げることができると思われる．索引のサイズを見てみると，TST 索引は ESP 索引に比べてたかだか三倍ほどだった．これは高頻度反復テキスト集合が非常に小さく圧縮できることを考えると実用上は十分な大きさであると思われる．

短いパターンでは，TST 索引は RLFM 索引に匹敵していた．q が小さい TST 索引のサイズはいくつかのテキストにおいては RLFM 索引よりも小さかった．加えて，TST 索引は RLFM 索引とは違い動的な更新操作を効率的に行うことができる．よって，テキストの動的な編集が必要な状況では TST 索引は有効であるといえる．

参考文献

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, Vol. 467, pp. 1061–1073, 2010.

[2] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. *CoRR*, Vol. abs/1705.10382, , 2017.

[3] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pp. 338–350, 2014.

[4] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In *Proceedings of the 20th Annual Symposium on Prague Stringology Conference*, pp. 158–170, 2016.

[5] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, Vol. 304, pp. 87–101, 2003.

[6] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23, pp. 337–343, 1977.

[7] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time-space trade-offs for lempel-ziv compressed indexing. In *Proceedings of 28th Annual Symposium on Combinatorial Pattern Matching*, Vol. 78 of *LIPIcs*, pp. 16:1–16:17, 2017.

[8] Gonzalo Navarro. A self-index on block trees. In *Proceedings of the 24th Symposium on String Processing and Information Retrieval*, pp. 278–289, 2017.

[9] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proceedings of the 19th Symposium on String Processing and Information Retrieval*, pp. 180–192, 2012.

[10] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, Vol. 31, pp. 538–544, 1984.

[11] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, Vol. 15, pp. 514–534, 1968.

[12] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, Vol. 65, No. 1, pp. 38–72, 2002.

[13] Luciana Vitale, Alvaro Martín, and Gadiel Seroussi. Space-efficient representation of truncated suffix trees, with applications to markov order estimation. *Theoretical Computer Science*, Vol. 595, pp. 34–45, 2015.

[14] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, Vol. 17, pp. 183–198, 1997.

[15] S. C Sahinalp and Uzi Vishkin. Data compression using locally consistent parsing. *Technical report, University of Maryland Department of Computer Science*, 1995.

[16] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space encoding LCE data structure with constant-time queries. *CoRR*, Vol. abs/1702.07458, , 2017.

[17] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proceedings of 41st International Symposium on Mathematical Foundations of Computer Science*, pp. 72:1–72:15, 2016.