Vol.2022-AL-188 No.2
2022/5/19

双方向 r-index

荒川 侑馬[2]　Gonzalo Navarro[1,3,a)]　定兼 邦彦[2,b)]

概要：反復の多い文字列を効率的に圧縮する索引は，バイオインフォマティクスやバージョン管理されたリポジトリなどの分野において有用である．Burrows-Wheeler 変換 (BWT) の連長圧縮を用いて実現される r-index はそのような索引の一種であり，BWT における run の数を r としたとき，$O(r)$ 語の空間計算量でパターンの出現位置を高速に検索することができる．この検索は，パターンの後方探索の過程で接尾辞配列のサンプルを保持し，そのサンプルからパターンの出現位置を全て計算するアルゴリズムにより実現される．本研究では，このアルゴリズムを発展させ，パターンを双方向に拡張可能かつ，探索の任意の時点でパターンの出現位置列挙が可能である双方向 r-index(br-index) を提案する．また，数値実験により，索引サイズ，および DNA のアラインメントを模した検索クエリの速度を評価する．

キーワード：圧縮文字列索引，Burrows-Wheeler 変換，簡潔データ構造

Bi-directional r-indexes

Abstract: Indexing highly repetitive texts is important in fields such as bioinformatics and versioned repositories. The run-length compression of the Burrows-Wheeler transform (BWT) provides a compressed representation particularly well-suited to text indexing. The r-index is one such index. It enables fast locating of occurrences of a pattern within $O(r)$ words of space, where r is the number of equal-letter runs in the BWT. Its mechanism of locating is to maintain one suffix array sample along the backward-search of the pattern, and to compute all the pattern positions from that sample once the backward-search is complete. In this paper we develop this algorithm further, and propose a new bi-directional text index called the br-index, which supports extending the matched pattern both in forward and backward directions, and locating the occurrences of the pattern at any step of the search, within $O(r+r_R)$ words of space, where r_R is the number of equal-letter runs in the BWT of the reversed text. Our experiments show that the br-index captures the long repetitions of the text, and outperforms the existing indexes in text searching allowing some mismatches except in an internal part.

1. はじめに

文字列索引は，対象となる文字列への検索などの操作をサポートするデータ構造である．その代表例である接尾辞配列 [11] は，接尾辞木 [20] から構造を単純化し，主にサポートする機能をパターンの出現回数を数える *count*，出現位置を列挙する *locate* に限定した索引である．接尾辞配列の機能をさらにコンパクトに実現するため，圧縮接尾辞配列が現在に至るまで研究されてきた．FM-index [3] は，文字列の Burrows-Wheeler 変換（BWT）[2] に基づく圧縮接尾辞配列の一種である．FM-index におけるパターンの検索は，パターンの末尾の文字から開始し，先頭へ向けて 1 文字ずつマッチングするアルゴリズムにより実現される．双方向 BWT [8] は，FM-index を文字列と反転文字列の両方に対して構築することで，パターンの左拡張に加えて右拡張も実現した．

当初考案された圧縮接尾辞配列は，統計的な圧縮の手法に基づいていた．しかし，この手法では，非常に長いパターンが反復したケースを感知し圧縮することができない．したがって，索引のサイズは入力のサイズに比例して大きくなっていく．バイオインフォマティクスを始めとする，巨大かつ冗長な文字列を取り扱う分野にはこのようなケースが多く，その解決策として Lempel-Ziv 圧縮や文法

1 Center for Biotechnology and Bioengineering, Beauchef 851, Santiago, Chile
2 東京大学
The University of Tokyo
3 University of Chile
a) gnavarro@dcc.uchile.cl
b) sada@mist.i.u-tokyo.ac.jp

ⓒ 2022 Information Processing Society of Japan

圧縮といった手法による索引が提案されてきた [14]．それらの索引は，*locate* や *count* をサポートするものの，接尾辞配列と異なる構造に基づいているため，それ以外のより複雑な操作の実現が難しかった．高度に反復的な文字列を取り扱うのに適した圧縮接尾辞配列として初めて提案されたのが r-index [5,6] である．この索引は，BWT の連長圧縮に基づき，$O(r)$ 語という索引サイズを実現している．ここで r は，BWT における同じ文字の繰り返しの総数である．r-index は効率的な *count* および *locate* を実現したが，従来の接尾辞配列でサポートされていた，より複雑な操作の実現はさらなる研究を待つところである．同じ BWT の連長圧縮に基づき，パターンの双方向拡張を可能にした索引も提案されているが [1]，応用において重要な *locate* はサポートされていない．

本研究の主結果

本研究では，r-index を発展させ，パターンの双方向への拡張を可能にした文字列索引（br-index）を提案する．空間計算量は $O(r + r_R)$ 語となる．ここで r_R は，元の文字列を反転させた文字列の BWT における同じ文字の繰り返しの総数である．定理 1 で提案するバージョンは簡易さを重視しており，元の文字列および反転文字列にそれぞれ構築した r-index を用いて容易に実現できる．定理 2 では，*left-extension* および *right-extension* のアルゴリズムを改良することで，時間計算量がアルファベットサイズ σ に依存しないバージョンを提案する．bi-directional BWT [8] と比較すると，br-index は文字列内の長い反復により敏感であり，高度に反復的な文字列を効率よく圧縮する．Belazzougui and Cunial [1] が提案した索引と比較すると，br-index はパターンの縮小（拡張の逆操作）はサポートしないが，効率的な *locate* をサポートしており，加えて実装が比較的容易である．詳細な比較を表 1 で示す．本研究ではこれらに加えて，定理 1 に基づく索引を実装し，性能を bi-directional BWT および r-index と比較した．

本論文は以下のように構成される．2 章では，主結果の説明に必要な概念について記述する．3 章では，br-index が用いるアルゴリズムを説明する．4 章では数値実験の結果を検討する．5 章で結論を述べる．

2. 準備

本論文においては，長さ n の文字列を n 個の文字の配列 $T = T[1]T[2]\cdots T[n]$ と定義する．それぞれの文字 $T[i]$ $(i = 1,\ldots,n)$ は，全順序の定められたアルファベット $\Sigma = \{1,2,\ldots,\sigma\}$ の要素である．簡単のため，$T[n] = 1, T[i] \neq 1$ $(i = 1,\ldots,n-1)$ と定義する．すなわち，末尾の文字を辞書順最小の文字に限定する．加えて，文字列 T に対し，$T^R = T[n-1]T[n-2]\cdots T[1].1$ を反転文字列と定義する．

文字列 T に対する 2 つの操作を以下のように定義する．ここで P は m 個の文字から成るパターンとする．

$\boldsymbol{count(P)}$　パターン P の出現回数を数える．

$\boldsymbol{locate(P)}$　パターン P が開始する位置を全て列挙する．

本論文では整数の集合 $\{l, l+1, \ldots, r\}$（$l > r$ のとき空集合）を $[l, r]$ と表記する．この表記を部分文字列や配列の部分列にも適用する．例えば，部分文字列 $T[l]T[l+1]\cdots T[r]$（$l > r$ のとき空文字列）を $T[l, r]$ と表記する．

predecessor データ構造 S を，操作 *pred* をサポートする全順序集合と定義する．$pred(S, i)$ は S における i 以下の最大の要素，すなわち $\max\{s \in S \mid s \leq i\}$ を返す．

2.1 接尾辞配列，Burrows-Wheeler 変換，LCP 配列

T の接尾辞配列 [11] $SA[1, n]$ は，$SA[i]$ が，辞書順 i 番目の接尾辞が開始する位置であるような整数配列である．すなわち，接尾辞 $T[SA[i], n]$ の辞書順が i となる．接尾辞配列の逆である ISA も定義する．ISA は $SA[ISA[i]] = i$ $(i = 1,\ldots,n)$ を満たす．

T の Burrows-Wheeler 変換（BWT）[2] は，文字の配列 $L[1, n]$ であり，

$$L[i] = \begin{cases} T[SA[i]-1] & (SA[i] \neq 1) \\ 1 & (SA[i] = 1) \end{cases}$$

を満たすものである．定義から，$L[i]$ は，辞書順 i 番目の接尾辞の直前に出現する文字を意味する．ただし，辞書順 i 番目の接尾辞が T そのものである場合，$L[i]$ は末尾の文字 1 となる．L に対する操作 *rank* も定義し，$rank_c(L, i)$ は $L[1, i]$ における文字 c の出現回数を数える操作とする．i が 0 のときは 0 と定義する．

T の最長共通接尾辞配列 $LCP[1, n]$ とは，

$$LCP[i] = \begin{cases} lcp(T[SA[i-1], n], T[SA[i], n]) & (i \neq 1) \\ 0 & (i = 1) \end{cases}$$

を満たす整数配列である．ここで $lcp(P, P')$ は，P と P' に共通する接頭辞のうち最長のものの長さである．

2.2 後方探索

接尾辞配列 SA と BWT L は，パターン $P[1, m]$ の *count* および *locate* を計算するのに役立つ．あるパターン P が与えられたとき，それに対応する SA 上の区間 $[s, e]$ が一意に存在し（パターン P が T で出現しない場合は空となる），$SA[s, e]$ が T における P の出現位置のリストとなる．このことからパターン P を区間 $[s, e]$ で表現することができる．L への *rank* を用いて，このパターンの左方向への拡張が可能である．具体的には，P に対応する区間が $[s, e]$ のとき，文字 c を加えたパターン cP に対応する区間 $[s', e']$ は以下の式で計算することができる．この，パターンの左への拡張手続きを *left-extension* と呼ぶ．

索引	サイズ	*left-extension*	*left-contraction*	*locate*
bi-directional BWT [8]	$O(nH_k(T)) + o(n\log\sigma)$ ビット	$O(\frac{\log\sigma}{\log\log n})$	不可能	$O(occ \cdot \log^{1+\epsilon} n)$
Belazzougui and Cunial [1]	$O(r + r_R)$ 語	$O(H^2 \log\log n)$	$O(H^2 \log\log n)$	不可能
br-index （定理 1）	$O(r + r_R)$ 語	$O(\sigma + \log\log_w(n/r))$	不可能	$O(occ)$
br-index （定理 2）	$O(r + r_R)$ 語	$O(\frac{1}{\epsilon}\log^{2+\epsilon} r)$	不可能	$O(occ)$

表 1 既存の双方向索引との性能比較．H は文字列の longest maximal repeat の長さ．*right-extension*(*contraction*) は *left-extension*(*contraction*) と対称的．w は計算機の語長．

Table 1 Comparison of space and time with the existing compressed bi-directional indexes. H is the length of the longest maximal repeat in the text. *right-extension*(*contraction*) is symmetric to *left-extension*(*contraction*). Here w is the number of bits in the computer word.

$$\begin{cases} s' = C[c] + rank_c(L, s-1) + 1 \\ e' = C[c] + rank_c(L, e) \end{cases}$$

ここで，$C[1,\sigma]$ は整数配列であり，$C[c]$ は c よりも小さい文字が T 内に出現する回数である．cP が T 内に出現しない場合は，上式の計算結果は $e' > s'$ を満たす．

FM-index [3] は，統計的な圧縮に基づく圧縮接尾辞配列である．FM-index により *count*(P) と *locate*(P) を計算するとき，まず空文字列 ϵ に対応する区間 $[1,n]$ から手続きを開始し，P の末尾から 1 文字ずつ，左へ拡張しながら区間を更新していく．$P[i+1,m] (1 \le i \le m)$ に対応する区間を $[s_{i+1}, e_{i+1}]$ としたとき，$P[i,m]$ に対応する $[s_i, e_i]$ は

$$\begin{cases} s_i = C[P[i]] + rank_{P[i]}(L, s_{i+1} - 1) + 1 \\ e_i = C[P[i]] + rank_{P[i]}(L, e_{i+1}) \end{cases}$$

により計算し，$s_i > e_i$ あるいは $i = 1$ となった時点で手続きを終了する．$s_i > e_i$ となって終了した場合には，*count*$(P) = 0$ であり，$i = 1$ で終了した場合は *count*$(P) = e_1 - s_1 + 1$，*locate*$(P) = SA[s_1, e_1]$ である．このパターン検索アルゴリズムは，パターンを左方（後方）へ向けて探索することから後方探索と呼ばれる．類似の計算である LF-mapping $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$ も定義し，以降では *left-extension* の時間計算量を t_{LF} と表記する．同様にして SA の要素にアクセスする時間を t_{SA} と表記する．

後方探索アルゴリズムに基づくと，*count* の時間計算量は $O(m \cdot t_{LF})$，*locate* の時間計算量は $O(m \cdot t_{LF} + occ \cdot t_{SA})$ となる．ただし occ は T 内の P の出現回数である．FM-index は，$nH_k(T) + o(n\log\sigma)$ ビットの空間計算量で $t_{LF} = O(\frac{\log\sigma}{\log\log n})$，$t_{SA} = O(\log^{1+\epsilon} n)$ を実現した [4]．ここで ϵ は $0 < \epsilon < 1$ を満たす任意の定数，$H_k(T)$ は T の k 次経験エントロピーである．

2.3 BWT の連長圧縮，r-index

FM-index のように統計的な圧縮手法を用いた場合，L に相当するデータ構造のサイズは入力サイズ n にほぼ比例して大きくなる．したがって，非常に反復の多い巨大な文字列を取り扱うには，反復をより敏感に検出し，それを活かして圧縮する必要がある．

Mäkinen and Navarro [9] は，文字列の反復を検出し圧縮する手段として，L における run に注目した．BWT L の run とは，L 内の同じ文字で構成された極大部分列である．SA において，接尾辞は辞書順にソートされているため，それら接尾辞の直前に出現する文字の配列 L は，T に反復が多いほど長い run を持つと直感的に期待される．実際に彼らは，任意の定数 $0 < \alpha < 1$ に対し，$k \le \alpha \log_\sigma n$ を満たす k について $r \le nH_k(T) + o(n)$ が成り立つことを示した．ここで r は L における run の総数である．BWT の連長圧縮を適用した FM-index である RLFM-index [10] は，$O(r)$ 語の空間計算量で $t_{LF} = O(\frac{\log\sigma}{\log\log r} + (\log\log n)^2)$ を実現し，効率的な *count* を可能とした．その一方，*locate* を計算するためには追加で $O(n/s)$ ビットが必要になるという問題があった．これは，SA の値を一定間隔 s ごとに保存したサンプルを必要とするためであった．

r-index [5, 6] が，SA の一定間隔サンプルを用いず，$O(r)$ 語，$O(m \cdot (t_{LF} + \log\log_w(n/r)) + occ \cdot t_\phi)$ 時間での *locate* の計算を可能にした．L に対する *rank* を行うデータ構造には RLFM-index を改良したものを用い，$t_{LF} = O(\log\log_w(\sigma + n/r))$ を実現した．また，後方探索の際に SA のサンプルを 1 つ保持し，関数 ϕ, ϕ^{-1} をサンプルに対し繰り返し適用することで，SA の一定間隔サンプルを不要とした．これらの関数は互いが互いの逆関数で

$$\phi(i) = \begin{cases} SA[ISA[i] - 1] & (ISA[i] \ne 1) \\ SA[n] & (ISA[i] = 1) \end{cases}$$

$$\phi^{-1}(i) = \begin{cases} SA[ISA[i] + 1] & (ISA[i] \ne n) \\ SA[1] & (ISA[i] = n) \end{cases}$$

を満たす．

これらの関数は，SA の値から隣接する SA の値を計算するものである．$SA[i]$ の値が分かっているとき，$SA[i-1], SA[i+1]$ はそれぞれ $\phi(SA[i]), \phi^{-1}(SA[i])$ で計算することができる．ϕ, ϕ^{-1} を計算する時間を t_ϕ とすると，r-index は $t_\phi = O(\log\log_w(n/r))$ を実現した．後方探

ⓒ 2022 Information Processing Society of Japan

索の過程で SA のサンプルを保持する具体的なアルゴリズムについては，本研究の主結果とも密接に関連するため，以下で詳細に説明する．

$T[i]$ が BWT における run の最初か最後の文字に対応する，もしくは $i = 1$ であるとき，$T[i]$ がサンプル文字であると定義する．サンプル文字の数は $O(r)$ となる．predecessor データ構造 R_c を各文字 c に対して作成する．R_c の要素は，c に等しいサンプル文字の BWT 上の位置とする．また，それら BWT 上の位置 $q \in R_c$ に関連付けて，値の組 $\langle q, SA[q]-1\rangle$ を保存する．後方探索では，SA 上の区間 $[s, e]$ に加え，SA のサンプル $(p, SA[p])$ を R_c を用いて更新していく．$P[i+1, m]$ から $P[i, m]$ へ拡張する過程を考える．$P[i+1, m]$ に対応する SA 上の区間 $[s_{i+1}, e_{i+1}]$ とサンプル $(p, SA[p])$ $(s_{i+1} \le p \le e_{i+1})$ から，$P[i, m]$ に対応する $[s_i, e_i], (p', SA[p'])$ $(s_i \le p' \le e_i)$ を計算する必要がある．このとき，$[s_i, e_i]$ は RLFM-index により求められる．サンプルは，$L[p] = P[i]$ のとき，$LF(p) \in [s_i, e_i]$ が成り立つことから $(p' = LF(p), SA[p'] = SA[p]-1)$ と更新する．$L[p] \ne P[i]$ だが，$P[i]$ が $[s_{i+1}, e_{i+1}]$ 内には出現するとき，$pred(R_{P[i]}, e_{i+1})$ を計算して predecessor $\langle q, SA[q]-1\rangle$ を得る．q は $L[q] = P[i]$ を満たすため，サンプルは $(p' = LF(p), SA[p'] = SA[q]-1)$ と更新することができる．

また，Nishimoto and Tabei [16] が，predecessor データ構造の使用を避けることで時間計算量を改善し，$O(r)$ 語の空間計算量を維持したまま $t_{LF} = O(1)$, $t_\phi = O(1)$ を実現している．

3. r-index の双方向化

r-index によってパターンを拡張できるのは左方向のみであり，*right-extension* はサポートされていない．本論文で提案する br-index は，以下の定理で示すように，双方向の拡張をサポートし，また探索の任意の時点で *locate* を計算できる文字列索引である．

定理 1. $O(r) + O(r_R)$ 語の空間計算量で，探索の任意の時点において，*left-extension* を $O(\sigma t_{LF} + \log\log_w(n/r))$ 時間，*right-extension* を $O(\sigma t_{LF^R} + \log\log_w(n/r_R))$，*count* を $O(1)$ 時間，*locate* を $O(occ)$ 時間で計算できる．ここで occ は現在のパターンの T における出現回数，w は計算機の語長，r_R は反転文字列 T^R の BWT L^R における run の総数である．

3.1 節および 3.2 節では定理 1 を証明する．3.3 節では，ウェーブレット木 [7] を用いることで *left-extension* と *right-extension* の時間計算量を改善した定理 2 を証明する．

定理 2. 任意の実数 $\epsilon > 0$ に対し，$O(r) + O(r_R)$ 語の空間計算量で，探索の任意の時点において，*left-extension* を $O(\frac{1}{\epsilon}\log^{2+\epsilon} r)$ 時間，*right-extension* を $O(\frac{1}{\epsilon}\log^{2+\epsilon} r_R)$，*count* を $O(1)$ 時間，*locate* を $O(occ)$ 時間で計算できる．ここで occ は現在のパターンの T における出現回数である．

br-index の基礎となるアイディアは，SA のサンプルと SA^R のサンプルをそれぞれ 1 つずつ保持することにより，*locate* を効率良く計算することである．これらのサンプルは必ずしもパターンの開始位置，終了位置を示すものではない．その代わりに，サンプルの指す位置とパターンの両端との距離，およびパターンの長さも探索の過程で保持する．

3.1 *left-extension* と *right-extension* の実現

SA および SA^R 上の範囲の更新

現在のパターン P に対応する SA 上の区間を $[s, e]$ とする．同様に，P^R に対応する SA^R 上の区間を $[s_R, e_R]$ とする．*left-extension* $P \to cP$ の計算にあたっては，$[s, e]$ を FM-index と同様の方法で更新する．$[s_R, e_R]$ の更新には Lam et al. [8] の提案したアルゴリズムを用いる．まず，c より小さい各文字 $a < c$ について LF-mapping を適用し，パターン aP の出現回数を数える．それらの総和を acc としたとき，$s_R \leftarrow s_R + acc$, $e_R \leftarrow s_R + acc + e - s$ によって $[s_R, e_R]$ を更新できる．*right-extension* は対称な手続きで計算できる．この場合は LF の代わりに，LF^R，すなわち L^R に対する LF-mapping を用いる．

これらの計算を行うのに必要な構造は，T および T^R にそれぞれ構築した RLFM-index である．索引のサイズは $O(r + r_R)$ 語であり，左拡張に $O(\sigma t_{LF})$ 時間，右拡張に $O(\sigma t_{LF^R})$ 時間を要する．

サンプルの更新

探索の過程で，$[s, e]$ と $[s_R, e_R]$ に加えて，7 つの値 $p, j, d, p_R, j_R, d_R, len$ を保持し，以後はこれらの値をまとめてサンプルと呼ぶ．p はサンプルの SA における位置，j は $SA[p]$ の値，d は現在のパターン P の開始位置と j の差である．すなわち，$j = SA[p]$ かつ $T[j-d, j-d+|P|-1] = P$ が成り立つ．p_R, j_R, d_R は反転文字列における同様の値であり，$j_R = SA^R[p_R]$ かつ $T^R[j_R-d_R, j_R-d_R+|P|-1] = P^R$ を満たす．len はパターンの長さ $|P|$ である．

ただし，実際のアルゴリズムでは p と p_R の値は維持しないことに注意されたい．本論文では正当性の証明のためこれらの値も定義するが，*left-extension*, *right-extension*, *locate* の実行にあたっては不要である．

left-extension $P \to cP$ を考える．SA 上の区間 $[s, e]$ の長さが拡張に際して変化しなかった場合，T における cP と P の出現回数は同じ，すなわち P の直前の文字は全て c である．この場合，d の値を 1 増加させるのみでよい．そうでない場合は，まず $pred(R_c, e)$ を計算して predecessor $\langle q, SA[q]-1\rangle$ を得る．その後，j, j_R を $j \leftarrow SA[q]-1$, $j_R \leftarrow n - j$ と更

アルゴリズム 1 *left-extension* $P \to cP$

```
Input: c, [s,e], [s_R,e_R], j, d, len
Output: [s',e'], [s'_R,e'_R], j', j'_R, d', d'_R, len'
 1: s' ← C[c] + rank_c(L, s−1) + 1
 2: e' ← C[c] + rank_c(L, e)
 3: if s' > e' then
 4:     cP does not occur
 5: else
 6:     acc ← 0
 7:     for a = 1 to c−1 do
 8:         acc ← acc + rank_a(L, e) − rank_a(L, s−1)
 9:     end for
10:     [s'_R, e'_R] ← [s_R + acc, s_R + acc + e' − s']
11:     if e' − s' ≠ e − s then
12:         (q, j') ← pred(R_c, e), d' ← 0
13:     else
14:         j' ← j, d' ← d + 1
15:     end if
16:     j'_R ← n − j', d'_R ← len − d'
17:     len' ← len + 1
18: end if
```

新する．d, d_R はそれぞれ $d \leftarrow 0,\ d_R \leftarrow len$ と更新し，そののちに $len \leftarrow len+1$ とする．*right-extension* は対称的な操作で実行できる．*left-extension* の詳細をアルゴリズム 1 に示す．

以下の補題では，拡張の前後で常に成立する不変条件について証明する．これらの条件は，次節で *locate* の正当性を証明するために用いる．

補題 3. *left-extension* と *right-extension* を計算しており，現在のパターンが P である状況を考える．P が空文字列 ϵ である場合を除き，以下の条件が常に成り立つ．

(1) $len = |P|$

(2) $d + d_R + 1 = len$

(3) $j = SA[p],\ j_R = SA^R[p_R]$ としたとき, $s \le LF^d(p) \le e$ かつ $s_R \le (LF^R)^{d_R}(p_R) \le e_R$

証明．$P = \epsilon$ から探索を開始するとき，各区間とサンプルは $s = s_R = 1,\ e = e_R = n,\ len = d = d_R = 0$ と初期化する．$j,\ j_R$ については，任意の predecessor $\langle q, SA[q]-1\rangle$ を取得して $j = y,\ j_R = n - y$ とする．以下で，(1) (2) (3) の条件が *left-extension* を通して不変であることを証明する．*right-extension* は対称的である．

はじめに, アルゴリズム 1 において $e'-s' \ne e-s$ となった場合を考える．(1) len' の値は len に 1 加えた値であることから $len' = |cP|$ が成り立つ．(2) $d'+d'_R+1 = 0+len+1 = len'$ が成り立つ． (3) R_c の定義から $j' = SA[q]-1$ であり，p の新たな値は $p' = LF(q)$ である．また，$j'_R = n-j' = n-(SA[q]-1) = SA^R[ISA^R[n-SA[q]+1]]$ であることから, p_R の新たな値は $p'_R = ISA^R[n-SA[q]+1]$ となる．このケースでは cP と $c'P (c' \ne c)$ が双方出現していることから，$[s,e]$ 内に文字 c で構成された run の境界が存在する．ここから $s \le q \le e$ かつ $L[q] = c$ が成立し，したがって $s' \le LF(q) = p' \le e'$ となる．逆方向については，まず $SA^R[(LF^R)^{d'_R}(p'_R)] = SA^R[p'_R] - d'_R = j'_R - d'_R = (n-j') - d'_R = n - (j'+d'_R)$ が成り立つ．T^R における位置 $n-(j'+d'_R)$ は，T における位置 $j'+d'_R = j'+len'-d'-1 = SA[LF^{d'}(p')] + len' - 1$ と対応する．これは T におけるパターン cP の終了位置となっており，したがって T^R における P^Rc の開始位置に対応する．以上から $s'_R \le (LF^R)^{d'_R}(p'_R) \le e'_R$ が成り立つ．

次に，$e'-s' = e-s$ となった場合を考える．このケースは P が空文字列のときには起こりえない．これは T が少なくとも 2 つの異なる文字を含むためである．したがってこの場合には帰納法仮定を用いることができる．(1) $e'-s' \ne e-s$ の場合と同様．(2) 帰納法仮定より $d'+d'_R+1 = d+1+d_R+1 = len+1 = len'$ が成り立つ．(3) j と j_R の値が変化しないため, $p' = p,\ p'_R = p_R$ である．このケースでは P の直前に出現する文字は c のみであるため, $s'_R = s_R,\ e'_R = e_R$ が成り立つ．加えて $d'_R = d_R$ であることから, 帰納法仮定と合わせて $s_R = s'_R \le (LF^R)^{d'_R}(p'_R) = (LF^R)^{d_R}(p_R) \le e'_R = e_R$ が成り立つ．逆方向については, $L[s] = L[LF^d(p)] = c$ から $s' = C[c]+rank_c(L,s-1)+1 = C[c]+rank_c(L,s),\ e' = C[c]+rank_c(L,e),\ LF^{d'}(p') = LF(LF^d(p)) = C[c]+rank_c(L, LF^d(p))$ であることと帰納法仮定を合わせ，$s' \le LF^{d'}(p') \le e'$ が成り立つ． □

3.2 *PLCP* を用いた *locate* の実現

次に *locate* のアルゴリズムについて述べる．出現位置の列挙は，r-index がサポートする関数 ϕ, ϕ^{-1} を用いて順次計算することができる．しかしながら，このままでは $p' = LF^d(p)$ の値が分からないため，P に対応する SA 上の区間 $[s,e]$ のうち，現在計算している $SA[p']$ に対応する p' が相対的にどの位置にあるか判定することができない．

ϕ, ϕ^{-1} の反復適用を終了する判定条件として，配列 $PLCP[1,n]$ を用いる．これは LCP を文字列位置の順に並び替えたもので，$PLCP[i] = LCP[ISA[i]]\ (i = 1,\dots,n)$ を満たす．

終了条件の判定を含めた *locate* の詳細をアルゴリズム 2 に示す．以下の補題 4 では, 補題 3 の条件が成立するとき, アルゴリズム 2 が正常に動作することを証明する．補題 3 と補題 4 より，*locate* の正当性が示される．

補題 4. $[s,e]$ を現在のパターン P に対応する SA 上の区間とする．アルゴリズム *2* の入力が $j = SA[p], s \le LF^d(p) \le e, len = |P|$ を満たすと仮定する．このとき，アルゴリズム *2* は P の出現位置を正しく全て出力する．

証明．ϕ, ϕ^{-1} の正当性は [6, Lem. 3.5.] で示されている．以下で終了条件の正当性について示す．$j = SA[p]$ であることから，$j' = j - d$ は $SA[p']\ (p' = LF^d(p))$ に等しい．仮定から $s \le p' \le e$ であり，示す必要があるのは

アルゴリズム 2 $locate(P)$

Input: $p,\ j(=SA[p]),\ d,\ len(=|P|)$
Output: T における P の全開始位置
1: $j' \leftarrow j - d\ (= SA[LF^d(p)])$
2: $pos \leftarrow j'$
3: **output** pos
4: **while** $PLCP[pos] \geq len$ **do**
5: $\quad pos \leftarrow \phi(pos)$
6: $\quad$**output** pos
7: **end while**
8: $pos \leftarrow j'$
9: **while** true **do**
10: $\quad$**if** $pos = SA[n]$ **then return**
11: $\quad pos \leftarrow \phi^{-1}(pos)$
12: $\quad$**if** $PLCP[pos] < len$ **then return**
13: $\quad$**output** pos
14: **end while**

- $PLCP[SA[p']] \geq |P| \Rightarrow p' > s$
- $PLCP[SA[p']] < |P| \Rightarrow p' = s$

である．$PLCP[SA[p']] \geq |P|$ であるとき，$PLCP[SA[p']] = LCP[ISA[SA[p']]] = LCP[p'] = lcp(T[SA[p'], n], T[SA[p'-1], n]) \geq |P|$ が成り立つ．仮定より，$T[SA[p'], n]$ の先頭 $|P|$ 文字は P に等しいため，$T[SA[p'-1], n]$ の先頭 $|P|$ 文字もまた P に等しい．したがって $p'-1$ も区間 $[s, e]$ に含まれ，$p' > s$ が成り立つ．一方，$PLCP[SA[p']] < |P|$ であるとき $lcp(T[SA[p'], n], T[SA[p'-1], n]) < |P|$ が成り立つ．$T[SA[p'], n]$ の先頭 $|P|$ 文字が P に等しいことから $T[SA[p'-1], n]$ の先頭 $|P|$ 文字は P と一致せず，$p'-1$ は区間 $[s, e]$ の外にある．したがって $p' = s$ となる．p' と e の関係についても，同様にして

- $PLCP[SA[p'+1]] \geq |P| \Rightarrow p' < e$
- $PLCP[SA[p'+1]] < |P| \Rightarrow p' = e$

が成り立つ．以上より，$s \leq p' \leq e$ が成立するか正しく判定できる．□

$PLCP$ の値は，ϕ, ϕ^{-1} を $O(r)$ 語の空間計算量，$O(1)$ 時間で計算する [16] のと同時に求めることができる．以上の議論を総合して定理 1 を得る．

3.3 拡張アルゴリズムの時間計算量改善

アルゴリズム 1 の 7 行目から 9 行目にかけて，$aP(a < c)$ の出現回数を数えるために，L への $rank$ を合計 $O(\sigma)$ 回計算する必要がある．この計算は，直交領域探索問題と捉えることができるため，BWT をウェーブレット木で表現すれば $O(\log \sigma)$ のオーダーまで削減することができる [12]．本節では，この手法に変更を加え，連長圧縮した BWT をウェーブレット木で表現することで，拡張アルゴリズムの時間計算量を改善する．

BWT の run の先頭文字を並べた文字の配列を考え，$L'[1, r]$ とする．L' の各文字の位置を x 座標，文字の値を y 座標とすれば，L' はサイズ $r \times \sigma$，r 個の点を含む 2 次元グリッド G とみなすことができる．すなわち，$L'[i] = c$ のとき，グリッド G の座標 (i, c) に点があると考える．そして，それぞれの点を，対応する run の長さで重みづけする．このグリッド G に対し，以下の定理を適用する．

定理 5 ([13]). r 個の点を持つサイズ $r \times r$ の *2* 次元グリッドを考え，各点が n 以下の非負整数を持つとする．任意の定数 $\epsilon > 0$ に対して，$O(\frac{1}{\epsilon} r \log n)$ ビットの空間計算量で，任意の長方形領域内の点が持つ整数の総和を $O(\frac{1}{\epsilon} \log^{2+\epsilon} r)$ 時間で計算することができる．

サイズ $r \times \sigma$ のグリッド G を，点のない領域で拡張してサイズ $r \times r$ とし，定理 5 を適用する．また，G に加えて，$L[i]$ が属する run の番号，およびその run の開始/終了位置を特定する機能が必要である．これは，r-index の構造を用いて $O(\log \log_w (n/r))$ 時間で実行できる．

以下の手順で $L[l, r]$ に含まれる c より小さい文字の総数を数える．(1) l, r が属する run の番号 x_1, x_2 を求め，それらの run が開始する位置 l', r' を求める．(2) 定理 5 を用いて，長方形 $[x_1 + 1, x_2 - 1] \times [1, c-1]$ 内の点が持つ重みの総和を求める．(3) $L[l] < c$ であれば $l' - l + 1$ を加え，$L[r] < c$ であれば $r - r' + 1$ を加える．

定理 5 のデータ構造を L，L^R の双方に構築することで定理 2 を得る．predecessor に由来する計算コスト $O(\log \log_w (n/r))$ は，代わりに二分探索を用いることで $O(\log r)$ 時間にできることに注意されたい．

4. 数値実験

4.1 実験準備

提案した文字列索引の性能を評価するため，Pizza & Chili Repetitive Corpus [*1] より取得した文字列データセットを対象として実験を行った．データセットの特性を表 2 に示す．本実験では，br-index，r-index，bi-directional FM-index (2BWT) の 3 種の索引を同じデータセットに対して構築した．br-index の構成要素である $PLCP$ の実装には，疎なビットベクトル [17, 19] を用いた．2BWT については，SA サンプルの間隔 s を $s = 16, 32, 64, 128$ の 4 通り試した．また，2BWT における BWT の表現には，RRR ビットベクトル [18] で実装されたウェーブレット木を用いた．

索引サイズの比較に加え，双方向のパターン拡張による検索の実用性を評価するため，バイオインフォマティクスで用いられるソフトウェア BLAST のアルゴリズムを模した *seed-and-extend* クエリの実行時間を測定した．このクエリでは，3 つの部分に分割したパターン $P = P_1 P_2 P_3$ を考え，P_2 が完全に一致し，P_1, P_3 に一致しない文字が定数 k 個以下であるような部分文字列の位置を全て列挙する．

*1 http://pizzachili.dcc.uchile.cl/repcorpus.html

2BWT と br-index においては，まず P_2 を不一致を許さずに探索する．その後，P_1 については *left-extension* を，P_3 については *right-extension* を用い，合計で k 個の不一致を許容しながら探索する．

r-index では *right-extension* を実行できないので，同様の方法を用いることはできない．代わりに 2 種類の異なるアルゴリズムを試行した．1 つ目のアルゴリズム（match-first）では，パターンの末尾から先頭へ向けて *left-extension* で探索し，P_1 と P_3 にのみ文字の不一致を k 個まで許容する．2 つ目のアルゴリズム（locate-first）では，最初に T 内の P_2 の出現位置を全て特定する．その後，各出現位置について，P'_1, P'_3 の不一致文字の数が合計 k 以下であるか判定する．

文字列は influenza を対象とし，パターン P は長さ $16, 32, 64$ の部分文字列を 100 個ずつランダムに抽出した．P_2 は P の中央部に設定し，$|P_2| = \lceil |P|/3 \rceil$ とした．許容する不一致の数は 0 から 10 の間で試行した．

4.2 実験結果

索引サイズを表 3 に示す．br-index のサイズは，多くの場合 2BWT のものよりも小さかった．例外的に，r/n の値が比較的大きい escherichia においては br-index の方が大きくなった．また，r-index と比較すると，すべての場合で 3 倍程度のサイズとなった．

図 1 に *seed-and-extend* の実行時間を示す．r-index における match-first アルゴリズムは，パターン長が短い $|P| = 16$ の場合を除き，許容する不一致の増加に応じて急激に実行時間が増加した．locate-first アルゴリズムは，逆にパターン長が短く，P_2 の出現回数が多いケースで実行時間が増加した．2BWT における実行時間は，不一致の増加に応じ，パターンが長いときは br-index と同様に，短いときは locate-first と同様に推移した．br-index は，許容する不一致の数が増加しても実行時間の増加は緩やかであり，概ねすべての場合において他の索引よりも高速に動作した．

5. 結論

本研究では，$O(r + r_R)$ 語の空間計算量で双方向のパターン拡張と出現位置の列挙をサポートする br-index を提案した．実験においては，br-index の索引サイズは r-index の 3 倍程度であり，2BWT と比較した場合は反復の多い文字列において有利であることが観察された．また，柔軟な検索の有効性を測る例として，*seed-and-extend* クエリの実行時間を測定した．このクエリにおいて，br-index は r-index よりも高速に動作し，許容する不一致文字の数が増えるほどその傾向は大きくなった．

本研究は，$O(r + r_R)$ 語に可能な限り近いサイズで完全な接尾辞木の機能を実現する，という目標への一助と捉え

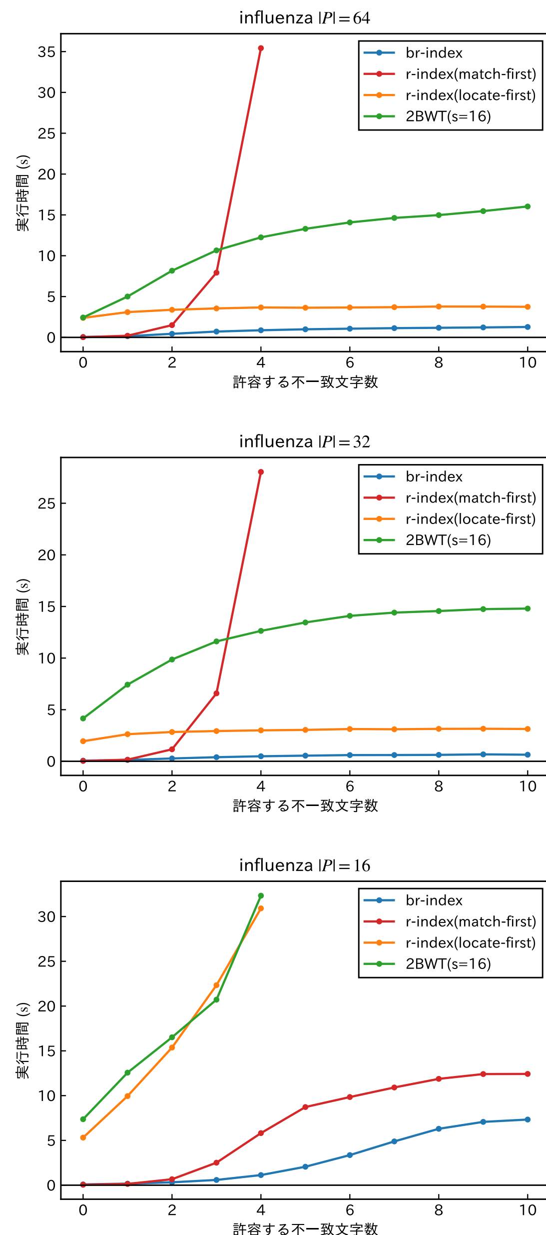

図 1 influenza に対する *seed-and-extend* の実行時間．2BWT は不明な理由で誤った位置を特定することがあるものの，発見総数は他の索引と近い値をとった．

Fig. 1 The computation times of *seed-and-extend* query on influenza. The 2BWT sometimes mistakenly locates positions for unknown reasons, but the number of reported patterns is very close to that of other indexes.

られる．これは，*left-extension* および *right-extension* がそれぞれ，接尾辞木における *weiner-link* および *child* に相当するためである．しかし，その一方で，br-index は *left-contraction* および *right-contraction* をサポートしないため，それらに対応する *suffix-link* および *parent* は実現されていない．これらの操作は，接尾辞木の木構造か，*LCP* に対する複数のクエリのいずれかで実現できることが分かっている．後者は $O(r \log \frac{n}{r})$ 語で実現可能である

ⓒ 2022 Information Processing Society of Japan

文字列	n	σ	r	r_R	r/n
escherichia	112,689,515	16	15,044,487	15,045,278	0.1335
influenza	154,808,555	16	3,022,822	3,018,825	0.0195
world-leaders	46,968,181	90	573,487	583,397	0.0122
einstein.en	467,626,544	140	290,239	286,698	0.0006

表 2 文字列データセットの特性．末尾に加えた辞書順最小の文字を含む．

Table 2 The statistics for the datasets. The lexicographically minimum character attached to the end is included.

	2BWT				r-index	br-index
	$s = 16$	$s = 32$	$s = 64$	$s = 128$		
escherichia	10.18	8.07	7.00	6.46	9.20	26.89
influenza	8.80	6.69	5.62	5.09	1.49	4.32
world-leaders	11.38	9.26	8.20	7.66	0.96	2.74
einstein.en	11.97	9.86	8.79	8.24	0.057	0.162

表 3 各文字列に構築した索引のサイズ（bits/symbol）．s は SA のサンプル間隔．

Table 3 The sizes (bits/symbol) of the indexes on the repetitive datasets. s is the sampling parameter for SA.

ことが示されているものの [6]，実用的な観点からは前者の手法が適していると考えられる [15]．これらの機能を $O(r + r_R)$ 語で実現できるか，あるいは，r 以外の実用的な指標による圧縮を行えるかが今後の課題である．

参考文献

[1] Belazzougui, D. and Cunial, F.: Smaller fully-functional bidirectional BWT indexes, *International Symposium on String Processing and Information Retrieval*, pp. 42–59 (2020).

[2] Burrows, M. and Wheeler, D. J.: A block sorting lossless data compression algorithm, *Digital SRC Research Report* (1994).

[3] Ferragina, P. and Manzini, G.: Indexing compressed text, *Journal of the ACM*, Vol. 52, No. 4, pp. 552–581 (2005).

[4] Ferragina, P., Manzini, G., Mäkinen, V. and Navarro, G.: Compressed representations of sequences and full-text indexes, *ACM Transactions on Algorithms*, Vol. 3, No. 2, p. article 20 (2007).

[5] Gagie, T., Navarro, G. and Prezza, N.: Optimal-time text indexing in BWT-runs bounded space, *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1459–1477 (2018).

[6] Gagie, T., Navarro, G. and Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space, *Journal of the ACM*, Vol. 67, No. 1, pp. 1–54 (2020).

[7] Grossi, R., Gupta, A. and Vitter, J. S.: High-order entropy-compressed text indexes, *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 841–850 (2003).

[8] Lam, T. W., Li, R., Tam, A., Wong, S., Wu, E. and Yiu, S. M.: High throughput short read alignment via bi-directional BWT, *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pp. 31–36 (2009).

[9] Mäkinen, V. and Navarro, G.: Succinct suffix arrays based on run-length encoding, *Annual Symposium on Combinatorial Pattern Matching*, pp. 45–56 (2005).

[10] Mäkinen, V., Navarro, G., Sirén, J. and Välimäki, N.: Storage and retrieval of highly repetitive sequence collections, *Journal of Computational Biology*, Vol. 17, No. 3, pp. 281–308 (2010).

[11] Manber, U. and Myers, G.: Suffix arrays: a new method for on-line string searches, *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948 (1993).

[12] Navarro, G.: Wavelet trees for all, *Journal of Discrete Algorithms*, Vol. 25, pp. 2–20 (2014).

[13] Navarro, G.: Document listing on repetitive collections with guaranteed performance, *Theoretical Computer Science*, Vol. 772, pp. 58–72 (2019).

[14] Navarro, G.: Indexing highly repetitive string collections, part i, *ACM Computing Surveys*, Vol. 54, No. 2 (2021).

[15] Navarro, G. and Ordóñez, A.: Faster compressed suffix trees for repetitive collections, *ACM Journal of Experimental Algorithmics*, Vol. 21, No. 1, p. article 1.8 (2016).

[16] Nishimoto, T. and Tabei, Y.: Optimal-time queries on BWT-runs compressed indexes, *Leibniz International Proceedings in Informatics*, pp. 101:1–101:15 (2021).

[17] Okanohara, D. and Sadakane, K.: Practical entropy-compressed rank/select dictionary, *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pp. 60–70 (2007).

[18] Raman, R., Raman, V. and Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 233–242 (2002).

[19] Sadakane, K.: Compressed suffix trees with full functionality, *Theory of Computing Systems*, Vol. 41, No. 4, pp. 589–607 (2007).

[20] Weiner, P.: Linear pattern matching algorithms, *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pp. 1–11 (1973).

ⓒ 2022 Information Processing Society of Japan