現実的な圧縮付全文索引

岡野原　大輔
東京大学大学院情報理工学系研究科コンピュータ科学専攻
hillbig@is.s.u-tokyo.ac.jp

本稿では，大規模なデータの全文索引を可能とする圧縮付全文索引技術の工学的な面からの解説を行う．全文索引は，長さ N のテキスト中の任意の部分列の出現回数，出現位置を $O(1)$，もしくは $O(logN)$ 時間で報告可能なデータ構造である．しかし，全文索引は作業領域量が大きいため，Web データやゲノム配列といった大規模なデータへの適用は難しかった．本稿で紹介する，圧縮索引技術はほぼ同じ同じ計算量で，全文索引の作業領域量を約 1/10 に減らす技術である．本稿では，特に工学的な側面から，圧縮索引の実現を難しくしている問題を解決する方法を紹介，提案する．また転置ファイルと同等の操作を $O(AH_0)$ (A は登録パタン総数，H_0 はパタンの 0 次エントロピー) の作業領域で可能で索引技術 IIWT を提案する．

Practical Compressed Full-text Indices

Daisuke Okanohara
Department of Computer Science, University of Tokyo
hillbig@is.s.u-tokyo.ac.jp

This paper describes practical compressed full-text indices. Full-text indices report the number of occurrence and all positions of an arbitrary pattern in $O(1)$ or $O(logN)$ time. However, since full-text indices need large work space, it cannot be applied to web data or genomic sequence. Compressed full-text indices, on the other hand, can operate same operations in about 1/10 of original work space. In this paper, we propose several methods which solve the problems which have not been solved in original papers. We also propose a new indexing method, IIWT, which can operate same functions as Inverted file indexing in $O(AH_0)$ work space (A is the number of all patterns, and H_0 is the 0-th entropy of patterns).

1　はじめに

近年，膨大なテキスト情報を利用した情報抽出，機械学習，自然言語処理が盛んに研究されている．その情報の例として，Web 情報，ゲノム配列，特許文書，論文集，そして対訳コーパスが挙げられる．しかしここで問題となるのは，どのように処理を行えば抽出，学習できるのかについては多くの技術が提案されている一方で，それらの技術が大規模な情報に対応していない場合が多いため，現実的には適用できないという点である．本稿ではその問題を解決するための圧縮索引技術を紹介する．多くの情報抽出，機械学習，自然言語処理は単純な操作からなっていることが多い．例えば，ある部分列が何回テキスト中に出現したかを答える部分列頻度計数や，何回文書に出現したかの文書頻度計数などである．これらは圧縮索引技術を用いることにより，現実的なリソースで大規模なデータを高速に処理することが可能となり（メモリー 4GB の PC で約 10GB のテキストが処理可能），数 GB から数 TB といった大

規模なデータに対し高度な情報抽出，機械学習が現実的な計算量，領域量で可能となる．

本稿は，近年盛んに研究されている接尾辞配列と Succinct なデータ構造を組み合わせた圧縮索引手法を紹介する．一つ目は Suffix Arrays（接尾辞配列）を圧縮表現した Compressed Suffix Arrays (CSA)，二つ目は Suffix Trees（接尾辞木）を圧縮表現した Compressed Suffix Tree (CST), 三つ目は転置ファイルのインデクスを Wavelet Tree を用いて圧縮表現したした Inverted file Index with Wavelet Tree (IIWT) である．

CSA は Grossi, 定兼らによって提案された圧縮索引技術であり，既にいくつか実装も存在している [5, 6, 7]．本稿では，はじめに従来の CSA を説明した後に，具体的な実装方法，及び約 10 倍高速な処理を実現する新符号法 Vertical Code，また実装の面の注意などを述べる．

CST は木構造の Succinct な表現 [2] や，節点の深さ情報の圧縮表現を実現したことで，はじめて Suffix Tree の全ての操作を処理可能なものが提案された [16]．CST は索引対象データ $T[1...N]$ に対し $6N + |CSA|$ bit の作業領域で Suffix Tree の各操作が実現可能である．但し，$|CSA|$ は T に対する CSA のサイズである．本稿では，特に計算量，領域量の両面で重要となる木構造の圧縮表現について述べる．

IIWT は本稿で初めて提案するデータ構造である．転置ファイル (Inverted File Index) は現在最も使われている索引手法の一つであり，各単語やパタン毎にその出現位置や出現文書番号を記録したものである．転置ファイルを保存するには，登録パタン数が A，テキスト長が N の時，$AlogN$ bit 必要である．また，Rice 符号や Golumb 符号等を使うことによりこの作業領域は $\Sigma_P|P|log(N/|P|)$ まで抑えられる．但し，P は各出現パタンであり，$|P|$ はその出現パタンの出現回数である．それに対し IIWT は Wavelet Tree[5, 7] を用いて同一の索引情報を AH_0 bit で保存可能である．ただし，H_0 はパタン集合から求められる 0 次エントロピーであり $H_0 = \Sigma_P|P|log(A/|P|)$ である．Wavelet Tree は各パタンの前後に出現したパタンを定数時間で求められるなど転置ファイルには無い様々なクエリをサポート可能である．また，作業領域をほとんど必要とせずに索引を逐次的に構築できる他，二次記憶領域との相性が良いなどの特徴がある．

本稿では，2 章で，転置ファイル，接尾辞配列 (Suffix Arrays)，接尾辞木 (Suffix Trees) を説明した後に，それらを圧縮表現した圧縮索引を 3, 4，5 章で説明する．

2 背景

本章では，転置ファイル，接尾辞配列，接尾辞木，Static Dictionary の概要を説明する．

2.1 転置ファイル

転置ファイル (Inverted File Index) は，各パタンの出現位置,，または出現文書番号を各パタン毎に保存したものである．一般的に転置ファイルは，パタンが出現した文書番号を記録するものが多いが，本稿中での転置ファイルは，テキスト中の出現位置を保持したものとする．(本稿で述べた技術はそのまま文書番号を記録した転置ファイルにも応用可能である)．具体的に，転置ファイルは各パタン t 毎に次のリストを保存している．$< f_t; p_1, p_2, p_3, \ldots, p_{f_t} >$ 但し，f_t はパタン t の出現回数であり，p_i はパタン t の i 番目の出現位置 $(0 \leq p_i < n)$ を表す整数である．これをそのまま保存した場合，パタン 1 回の出現に付き $logN$bit 必要なため，全パタンの出現回数が A の時，$Alogn$ bit 必要である．

転置ファイルを圧縮して保存するには，$< f_t; p_1, p_2 - p_1, p_3 - p_2, \ldots, p_{f_t} - p_{f_{t-1}} >$ のように，リスト要素の差分を求めた後，これらの整数を整数符号で保存する方法が一般的である．この差分リストから元の出現リストの復元は一意に可能である．

代表的な整数符号としては，Byte aligned code が挙げられる．これは，整数 x が与えられた時，初めに x を，その下位 7bit，x_{low} とそれ以外の上位 bit,

x_{high} に分ける．次に，$x_{high} = 0$ ならば $x_{low} + 128$ をバイト出力して終了し，そうでないならば x_{low} をそのままバイト出力した後に x_{high} に対し同じ操作を再帰的に繰り返す．Byte aligned code の符号，復号ともに非常に高速であり，そして通常の文字列探索手法を用いて符号化されたデータ中から任意の整数を探索できる特徴がある．

その他の整数符号化法としては Golomb 符号 [3] が利用される．Golomb 符号は整数パラメータ b を用いる．この符号は，x を b で割った商と余りが，それぞれ q と r であった時 q を unary 符号で出力し，r を binary 符号で出力する．特に b に 2 のべき乗を選んだ場合は Rice Coder [15] と呼ばれており，シフト演算とマスク操作だけで符号が決定できるので高速な処理が可能である．Golomb 符号は整数が超幾何分布に従って出現した場合に適切なパラメータ b を選ぶことで最小冗長の符号となることが知られてい [11].

この他の整数符号法としては出現分布に偏りがある場合に効率良く符号化が可能な Interpolative 符号 [12] や，高速な復元が可能な Recursive Integer 符号 [11] 等が知られている．

2.2 接尾辞配列

検索対象テキストを $T[0...n-1] = T[0]T[1]...T[n-1]$ とする．また，$T[n]$ は T 中に現れるどの文字より小さい文字 \$ と仮定する．この時 T の接尾辞 $S_i, i \in 0, ..., n$ は $S_i = T[i...n]$ と定義される．また接尾辞間の辞書式順序を $S_i < S_j \Leftrightarrow T[i] < T[j]$ or $S_{i+1} < S_{j+1}$ と定義する．この時，$T[i] = \$$ から全ての接尾辞間で順序関係が定義される．Suffix Arrays (接尾辞配列)$SA[0...n]$ は全ての接尾辞を辞書式順序で整列させた際の添え字番号を保持したものであり，任意の $i < j$ に対し，$S_{SA[i]} < S_{SA[j]}$ が成り立つ．SA を保持するには $nlogn$bit 必要である．一般に整数は 4 バイトで保存されるので SA の保存には $4n$ バイト必要である．

Suffix Arrays を用いて，任意の部分列 $Q[0...m-1]$ の出現位置は次の通りに求められる．Suffix Arrays 上では Q を接頭辞（Prefix) として持つ接尾辞は連続した領域に存在するため，この領域を Suffix Arrays 上で二分探索を行って求める．この二分探索は，Q と接尾辞の一回の比較に $O(m)$ 時間必要であり，二分探索なので $O(logn)$ 回の比較を行う．よって全体で $O(mlogn)$ 時間必要である．Q の出現回数は $O(mlogn)$ 時間で求まり，全ての出現位置は Q の出現回数が $occ(Q)$ の時，$O(mlogn + occ(Q))$ で求められる．

2.3 接尾辞木

Suffix Tree(接尾辞木) は T の全ての接尾辞から構成される圧縮表現された Trie 木である．但し，検索対象テキスト，接尾辞は接尾辞配列と同様に定義する．Suffix Tree は各接尾辞に対応する N 個の葉を保持する．また，各枝には Edge Label が付与されており，根から葉までの全ての Edge Label をつなげたものは葉が表現する接尾辞に一致する．各内部節点は，必ず 2 個以上の子を保持し，それぞれの子との間の枝に付与された Edge Label の先頭文字は全て異なっている．節点 t の Path Label を根からその節点までの Edge Label をつなげたものと定義する．根を除く各内部節点は必ず一つの Suffix Link を保持する．節点 t,u が与えられた時，t の Path Label の先頭文字を除いたものと U の Path Label が一致する時，t から u への Suffix Link が存在すると定義する．接尾辞木は一般に次の操作をサポートする [16].

1. root(): 根を返す
2. isleaf (v): v が葉の時 true, そうでない場合は false を返す
3. child (v, c): Edge Label が c で始まる節点 w を返す．もしそのような節点がない場合は NULL を返す
4. sibling (v): v の次の兄弟を返す.
5. parent (v): v の親を返す．もし無い場合 (v が根の場合) は v を返す.

6. edge(v, d): v を指している枝の d 番目の文字を返す.

7. depth(v): v の Path Label の文字数を返す

8. lca(v,w): v と w の最小共通祖先を返す

9. sl(v): v の suffix link を返す.

Suffix Tree を用いて様々な文字列アプリケーションが実現可能であることが知られている [8]. 一般的な実装では, Suffix Tree の各節点は最初の子及び次の兄弟, そして親へのポインタを保持する. これより Suffix Tree を保持するには $O(nlogn)$ の作業領域が必要である.

2.4 Static Dictionary

0 と 1 のみから構成される bit 配列 $B[0...n-1]$ が与えられた時, $rank_1(B,i)$ は, $B[0..i]$ 中の 1 の出現回数を返し, $select_1(B,i)$ は, (i+1) 番目の 1 の位置を返す. $rank_0(B,i)$, $select_0(B,i)$ も同様に定義される.

$o(n)$ の補助領域を用いて定数時間の rank, select 操作を実現するデータ構造はいくつか提案されている [13, 14]. しかし, これらはいずれも計算量, 領域量的に実際のデータでは非現実的であることが指摘されている [4, 2, 9]. 現実的には $o(n)$ の補助領域で $O(logn)$ 時間で答える手法 [4], もしくは $O(n)$ (約 $2n$bit) の補助領域を用いて $O(1)$ 時間で答える手法 [9] を用いる. また, rank のみが必要, もしくは select のみが必要, 1 が非常に疎な bit 配列の場合等, 状況に応じて最適なデータ構造は変わってくる.

本稿では, 1 と 0 の数が同程度の場合に有効な例として [9] を, 1 の数が非常に少ない場合の Static Dictionary として Vertical Code を紹介する. Vertical Code については 3.2 章で詳しく述べる.

[9] では二つの方法が提案されているが, その中で実際のデータで最も高速であった Byte Aligned をさらに簡略化したものを解説する. はじめに rank, 次に select の実装方法を述べる.

rank の実装では, 初めに $B[0...(n-1)]$ を 256 個ずつのブロック $B_0, B_1, ..., B_{(n-1)/256}$ に分割しそれぞれの先頭の rank の結果, $rank_1(B,0), rank_1(B,256), ...,$ を前計算しておき, それらを $T_1[0, ..., (n-1)/256]$ に保存する. 次に分割された各ブロックにおいて 32 個ずつの rank の結果 $rank_1(B_0,0), rank_1(B_0,32), ..., rank_1(B_{(n-1)/256})$ を前計算し, $T_2[0, ..., (n-1)/32]$ に保存する. T_1 は各エントリに 4 バイト必要であり, T_2 は各エントリに 1 バイト必要である. これらのデータ構造を用いて $rank_1(B,i)$ は, $T_1[i/256] + T_2[i/32]$ と残り $B[(i/32*32)...i]$ 中の rank の和で求められる. 残りの rank は実際の bit 配列とテーブル参照を用いて求められる. $rank_0(B,i)$ は $i - rank_1(B,i)$ で求められる.

select の実装では, はじめに $B0...(n-1)]$ を 8 個ずつのブロック $C_0, C_1, ..., C_{(n-1)/8}$ に分割する. bit 配列 $P[0...(n-1)/8]$ を , C_i が全て 0 の時 $P[i] = 0$, それ以外の時, $P[i] = 1$ と定義する. Bit 配列 $Q[0....m]$ を B 中の i 番目 $(i-1)$ 番目の 1 が属するブロックが違う時に $Q[i] = 1$, それ以外で $Q[i] = 0$ と定義する. この時 $select_1(P, rank_1(Q,i))$ は (i+1) 番目の 1 が所属するブロック番号を返す. Q の select の実装には Q のほとんどが 1 であることを用いる. Q 中で 0 が連続する部分を clump と呼ぶ. i 番目の clump の連続する長さを $ClumpArray[i]$ に保存しておき, また各 clump の直後が 1, それ以外は 0 である bit 配列 $R[0...(n-1)/8]$ を用意する. この時 $select_1(Q,i)$ は $ClumpArray[rank_1(R,i)] + i$ として求められる. これらをまとめると, $select_1(B,i)$ を求める場合は, $j = select_1(P, rank_1(Q,i))$ で (i+1) 番目の 1 が属するブロック番号 j を求めた上で, C_j までの 1 の数を求め $(rank_1(B, 8*j-1))$, 残りの 8bit 中の select を実際の bit 配列とテーブル参照を使って求める. $select_0(B,i)$ は B を bit 反転させた bit 配列に対して同じようにしてデータ構造を前計算しておき, 求める.

3 圧縮接尾辞配列

本章では接尾辞配列を圧縮表現した圧縮接尾辞配列の概要，そして改良手法を述べる．

3.1 圧縮接尾辞配列の概要

圧縮接尾辞配列 (Compressed Suffix Arrays, CSA) にはいくつかの変種が存在する [17, 5, 7]．ここでは [17] に基づいて説明するが，いくつかの部分で簡略化している．

SA はそのままでは圧縮できない．なぜなら SA は $0...n$ までの整数の並び替えであり，冗長性が利用できないからである．CSA は，SA を直接保存する代わりに，次のように定義される配列 Ψ を保持することで，圧縮表現を可能とする．

$$\Psi[i] \equiv \begin{cases} SA^{-1}[SA[i]+1] & (SA[i] < n) \\ SA^{-1}[0] & (SA[i] = n) \end{cases}$$

但し，SA^{-1} は $SA[j] = i$ の時 $SA^{-1}[i] = j$ と定義される配列である．Ψ は直感的には SA 上を移動するための関数であり，

$$SA[i] = SA[\Psi[i]] - 1$$

の関係式が成り立つ．なぜなら右辺を変形すると，$SA[SA^{-1}[SA[i]+1]] - 1 = SA[i] + 1 - 1 = SA[i]$ が成り立つからである．CSA は，Ψ の他に $SA[i]$ が k で割り切れる時 $B[i] = 1$，そうでない時は $B[i] = 0$ と定義される bit 配列 $B[0...(n/k)]$，および SA をサンプリングした配列 $SA_s[1...(n/k)]$，$SA_s[i] = SA[select_1(B, i)]$ を保存する．

これら Ψ，B，SA_s を用いて，任意の i に対する $SA[i]$ は次のようにして求められる，はじめに $B[i] = 1$ の時は $SA_s[rank(B, i)]$ を返す．そうでない場合は $SA[i] = SA[\Psi[i]] - 1$ の関係式を用いて SA 上を移動する．そして t 回の移動で $B[\Psi^t[i]] = 1$ が成立した場合，$SA_s[rank(B, \Psi^t[i]] - t$ を返す．この t は最大でも $k-1$ であることが，SA のサンプリングの仕方から保障される．それは，Ψ を一回適用する毎に，1 大きい値を持つ SA の index が返ってくるが，SA はその値が k で割り切れる値でサンプリングされているので，Ψ を多くとも $k-1$ 回適用すれば必ずサンプリングされている SA が見つかるからである．

次に，Ψ の重要な性質を述べる．$i < j$ かつ $T[SA[i]] = T[SA[j]]$ の時，$\Psi[i] < \Psi[j]$ が成り立つ．なぜなら，Suffix Arrays 上では，Suffix は辞書式順序で整列されているため，1 文字目が同じ場合は 2 文字目以降で順序が決定されている．これより 2 文字目以降からなる Suffix 間でも順序は保たれている．これを式で表すと $SA^{-1}[SA[i]+1] < SA^{-1}[SA[j]+1]$ であるが，これは，それぞれ定義より，$\Psi[i]$，$\Psi[j]$ であるから $\Psi[i] < \Psi[j]$ が成り立つ．

よって，Ψ はその Suffix の先頭文字が同じ部分では単調増加列となる．この単調増加列から差分列 $d[i] = \Psi[i] - \Psi[i-1]$ を構成した後に σ 符号で保存することにより $O(H_0)$ で保存できることが示されている [17]．

さらに CSA は元テキストを保存せずに検索が行える上に，任意の位置の部分文字列復元が可能である [17]．

3.2 Vertical Code

Ψ の差分列 $d[0...n]$ を σ 符号等の符号法で圧縮表現する場合は，復元時に多くの bit 演算が必要である．さらに，差分列から元の列を復元するためには差分列の合計を求める操作が必要となる．本章では高速な復元，加算操作が可能な符号法である Vertical Code を提案する．

以降は次で定義される差分列 $d[0...n]$ を符号化する場合を考える．

$$d[i] \equiv \begin{cases} \Psi[i] - \Psi[i-1] - 1 (i \neq 0) \\ \Psi[0] (i = 0) \end{cases}$$

しかし，Ψ は先頭文字が同じ時のみ部分単調増加列であるので，違う文字の間では $d[i] < 0$ となりうる．そのような $d[i]$ には n を加えておくことで常

に単調増加であるようにしておく．この場合 $\Psi[i] = \Sigma_{k=0}^{i} d[k] + i \bmod n$ として求められる．以降ではこの前処理を d に適用しておき，d が 0 以上の整数のみからなる場合を扱う．

Vertical Code は任意の 0 以上の整数列に対し適用可能な符号法である．Vertical Code はバイト単位で処理可能でかつ，圧縮率は σ 符号と同程度である．

Vertical Code は，はじめに，$d[i]$ を一定数 M(M=8,16) ずつのブロック $B_0, B_1, ..., B_{n/M}$ に分割する．i 番目のブロックは $d[iM, iM+1, ..., (i+1)M-1]$ を保持する．また，i 番目のブロック中の最大数を $Max[i]$，$Max[i]$ を binary 表現した時に必要な bit 数を $MSB[i]$ とする．この時，i 番目のブロック中の数は全て $MSB[i]$bit を用いて表現できることに注意する．

次に，各 $d[i]$ を binary 表現した時の j 桁目の bit を $bit[i][j]$ と定義する．例えば $d[i] = 6$ の時，$d[i][0] = 0$, $d[i][1] = 1$，$d[i][2] = 1$ である．Vertical Code は次のように定義される $V_i[0, ..., MSB[i]]$ を保持する．

$$V_i[j] = d[iM][j], d[iM+1][j], ..., d[(i+1)M-1][j] \quad (1)$$

$V_i[j]$ はブロック B_i を binary 表現した時の j 桁目の bit をつなげた bit 列である．M が 8 の倍数の時，$V_i[j]$ は全てバイト単位で表現できることに注意する．Vertical Code はこの $V_i[j]$ に加え，各 Block の先頭 $d[iM]$，及び $MSB[i]$ を保持する．

Vertical Code がどのように $\Psi[i]$ を復元するかを以下に述べる．$\Psi[i]$ は前述したように $\Psi[i] = \Sigma_{k=0}^{i} d[k] + i$ として求められる．この時，$p = \frac{i}{M}$, $q = i \bmod M$ とすると，この右辺は $\Sigma_{k=0}^{i} d[k] = d[pM] + \Sigma_{k=0}^{MSB[p]} (V_p[k] \& ((1 << q) - 1)) \ll k$ として求められる．但し & は bit マスク操作，$\ll$ は左シフト演算である．この時，M が 8 の倍数の時，$V_i[j]$ は全てバイト単位であり，上の操作は全てバイト単位のシフト，マスク演算で処理可能である．

Ψ を求める最悪計算量は，$O(logn)$ であり，最悪領域量は $O(nlog(n/M))$bit である．しかし，実際のデータでは，同一ブロック中の $d[i]$ は同程度の大きさの値を持つ場合が多いため，実際の領域量，計算量はこれらより小さい．例えば，$\Psi[i]$ を求める計算量は，$O(MSB[i/M])$ であるが，多くの $MSB[i]$ は 0 や 1 と非常に小さく高速な計算が可能となる．

また，$d[iM]$ は $MSB[i]$ は [11] と同様に，再帰的に同じデータ構造を用いて保存することで領域量を削減することが可能である．

3.3 現実的な rank, select 操作の実現

CSA は多くの場面で rank, select 操作を利用する．その実現には 2.4 章で述べた Static Dictionary を利用することも考えられるが，CSA で利用する Bit 配列は 1 が非常に疎な Bit 配列であり，かつ $rank_1$,$select_1$ しか使わない．そのため，そのまま Bit 配列を Static Dictionary を使って保存するより効率良く符号化することが可能である．

本章では Vertical Code を使った実装を紹介する．Vertical Code を使って $select_1$ を求めるには，3.2 章で述べたように i 番目の 1 が出現している位置を $\Psi[i]$ だと考えて同様にして差分列を構成し符号化する．$select_1(B, i)$ を求めるのに必要な計算量は $O(MSB[i/M])$ である．それに対し $rank_1$ を求める場合は，Vertical Code で $select$ を用いて二分探索を行う必要がありこの場合の計算量は O(logN) となる．実際には $B[0...N]$ を M ずつブロック毎に区切った後の rank 結果を持っておき，それより細かい単位は表引きを行う．この場合，select を求める際の最悪計算量は $O(logN)$ のままだが，現実的には，多くの操作が $O(1)$ で可能である．

3.4 実装時の注意

他の CSA の多くの実装では添え字側が k で割り切れる場合にサンプリングを行っている場合がある．この場合，rank 操作は必要ないが，T 中に非常に長い一致列がある場合は $SA[i]$ を求める計算量が非常に大きくなることがある．この場合，SA を求める計算量は $O(n)$ となる．

4　圧縮接尾辞木

本章では，接尾辞木を圧縮表現した圧縮接尾辞木の概要，および実装手法を説明する．

4.1　圧縮接尾辞木の概要

圧縮接尾辞木 (Compressed Suffix Trees, CST) は Compressed Suffix Arrays, Hgt 配列，Parenthesis Tree から構成される．

それらのうち特に実現が難しかった低領域での rank, select 操作は Vertical Code を用いて実現できる他，Hgt Array も Vertical Code を用いて圧縮表現が可能である．Parenthesis Tree の保持は [2] の方法が現実的である．以下では，これらのうち特に重要な Parenthesis Tree に焦点を絞って解説する．

Parenthesis Tree (PT) は木構造を括弧列 PA（実装では開き括弧を'1'，閉じ括弧を'0' で表現した Bit 配列）で表現することで，M 個の節点を持つ木を $2Mbit$ で表現する．木構造から PA への変換は木の節点，葉を深さ優先探索でたどり，最初に節点を辿った時 '1' を，親へ戻る時 '0' を出力することで行われる．例えば葉は '10' で表現され，二つの葉を持つ節点からなる木構造は '110100' となる．本稿では各節点は開き括弧 '1' で表現する．

Parenthesis Tree は，$o(M)$ サイズの補助データを用いて，次の操作を定数時間で実現する．

1. findclose(i): i が開き括弧の場合は対応する閉じ括弧の位置を返す．そうでない場合は NULL を返す．
2. findopen(i): i が閉じ括弧の場合は対応する開き括弧の位置を返す．そうでない場合は NULL を返す．
3. enclose(i): i をもっともきつく囲む括弧対の開き括弧の位置を返す．

これらの操作を用いた，木上の各操作の例を次に示す．

- root(): 0 を返す
- isleaf (v): v が開き括弧，かつ，v+1 が閉じ括弧の時 true, そうでない場合は false を返す
- sibling (v): findclose(v)+1
- parent (v): enclose(v)

他の木の操作についても，findclose, findopen enclose，そして PA 上での rank, sleect を用いることで定数時間で操作可能である [16].

4.2　現実的な実装方法

findclose, findopen, enclose を定数時間で行うためのデータ構造を [2] に沿って解説する．

このデータ構造では，括弧列 $PA[0...n]$ を一定数 B ごとのブロックに分割する．$PA[i]$ が属するブロック番号を $b(i)$ で表すことにする ($b(i) = i/B$)．また，$PA[i]$ に対応する括弧の位置を $m(i)$ とする．各括弧を 'near' と 'far'，'far' をさらに 'pioneer' と 'non pioneer' に以下のように分類する．$b(i) = b(m(i))$ であるような i の括弧を 'near'，そうでない括弧を 'far' と定義する．さらに，$\forall j, j < i, b(j) = b(i)$ について $b(m(j)) \neq b(m(i))$ となる i を 'pioneer'，そうでない 'far' を 'non pioneer' と定義する．この pioneer の定義は開き括弧についての定義だが，閉じ括弧についても同様に定義する．また，対応する括弧が pioneer である括弧も pioneer であると定義する．この時，pioneer である括弧対はブロック数が B の時，括弧対に交差がないことから最大で $4B - 6$ 個であることが示せる [2].

pioneer を利用するのは，'far' である括弧の数は最大で n となるが (ブロック数が偶数であり，括弧列の前半全てが開き括弧，後半全てが閉じ括弧の場合），'pioneer' である括弧の数は $4B - 6$ と，B で抑えられるからである．

次に，Parenthesis Tree の具体的な操作のうち $findclose$ についてを説明する (その他の操作については [2] を参照)．以下では $findclose(i)$ を求める

場合を考える．もし $PA[i]$ が near ならば前もって構築しておいたテーブル参照で対応する括弧の位置を求める．$PA[i]$ が pioneer ならば，前もって構築しておいた全 pioneer の答えからなるテーブル参照で求める (もしくは pioneer のみから構成される括弧列について再帰的に木を構築する)．far の場合は直前の pioneer，$PA[j]$ を求め，対応する括弧が存在するブロック番号を $b(m(i)) = b(m(j))$ と求めた後に，$PA[j+1, ..., i-1]$ 中の開き括弧と閉じ括弧の差を求め，$b(m(j))$ においてテーブル参照を用いて $m(j)$ を求める．これらの操作はいずれも定数時間で行える．

5 IIWT

本書では転置ファイルを圧縮表現した IIWT の概要，そして実装手法を説明する．

5.1 Wavelet Tree

Wavelet Tree は [5] で初めて提案されたデータ構造である．

Wavelet Tree は，アルファベット集合 Σ から構成される文字列 $T[0...N]$ が与えられた時，任意のアルファベットに対する rank, select を NH_0bit を用いて平均 $O(H_0)$ 時間で求めることができるデータ構造である．ただし H_0 は T の 0 次エントロピーである (2.4 章で述べたデータ構造は $|\Sigma| = 2$ の場合である)．

Wavelet Tree は次のように構成される．はじめに全てのアルファベットを T 中の出現確率に基づいて Huffman 符号で符号化する．次に, 構成された Huffman 木の各節点に，以下のように定義される Bit 配列を保持する．文字 a を Huffman 符号で符号化した後の bit 列を B_a，B_a の k 番目の bit を $B_a[k]$ と定義する．この時，根の節点に対応する bit 配列は，T 中の各文字の $B_{T[0]}[0], B_{T[1]}[0], ...,$ を順に並べたものとする．次に根の左側の子には $B_a[0] = 0$ だった文字 $T[i_1]T[i_2]....$ を，右側の子には $B_a[0] = 1$ であった文字 $T[j_1]T[j_2]...$ を移動させる．次に左側の子の bit 配列は $T[i_1]T[i_2]...$ の 2 番目の bit 配列を並べた $B_{T[i_1]}[1], B_{T[i_2]}[1], ..,$ とする．以下同様にして，深さ k の節点では k 番目の bit を並べたものを配置し，その時の bit に応じて文字を左右の子に分配していく．これを全ての節点に対して行い，各節点の bit 配列を決定する．

このようにして構成された木のサイズは文字 a の 1 回の出現に付き $|B_a|$bit 必要であるので，全節点の bit 配列をつなげた長さは NH_0bit となる．

任意の文字 c の $rank_c(T, i)$ は次のように再帰的に求められる．c の bit 表現が 110 であったとする．最初に根に対応する bit 配列 B_{root} で c の 1bit 目の'1' の $p_1 = rank_1(B_{root}, i)$ を求める．次に右の子に移動し，2bit 目の 1 に対応する位置 $p_2 = rank_1(B_1, p_1)$ を求める．最後に右の子で 3bit 目の 0 に対応する $p_3 = rank_0(B_{11}, p_2)$ を求める．$rank_c(T, i) = p_3$ である．同様にして $select_c(T, i)$ は葉から再帰的に select 操作を適用することにより求められる．これより，各 bit 配列において rank, select が $O(1)$ で求められるならば，Wavelet tree を用いて任意の文字 c に対する rank, select は平均 $O(H_0)$ 時間で求めることができる．

5.2 IIWT の概要

IIWT は転置ファイルを Wavlet Tree を用いて圧縮表現したものである．

登録するパタン $P_0, ..., P_m$ とそれらの出現位置 $p(P_0), ..., p(P_m)$ が与えられた時 $(0 \leq p(P_i) < n)$，パタンテキスト $T_p[0...n-1]$ を，$p(P_j) = i$ の時 $T_p[i] = P_j$ と定義し，それ以外は $T[i] = 0$ とする．ただし 0 は，パタンの先頭位置ではないことを表す文字であり，$0 \neq P_i$ である．この T_p から Wavelet Tree を構築したものが IIWT である．必要な領域量は，登録パタン総数を A，パタンから計算される 0 次エントロピーを H_0 とした時，$O(AH_0)$ である．

パタン q の全出現位置は $select_q(T_p, 0), select_q(T_p, 1), ...,$ として求めることが出来る．このとき q の全ての出現位置を求める計算量は，q の出現回数を C_q とした時 $O(C_q log H_0)$

となる．さらに fractional cascading を用いることで，select を順に求めた場合の計算量を減らすことができる（係数で）．Wavelet Tree は，転置ファイルとは違って異なるパタン間の位置関係が保存されているので，パタンの前後に出現したパタンを $O(H_0)$ で求めることが可能である．またテキスト中の連続する位置中（例えば 1 文書中に）にパタンが何回出現したかなどを $O(H_0)$ で求めることができる．

6 謝辞

この研究成果の一部は 2004 年度第 2 回未踏ソフトウェア創造事業の成果である．

7 質疑応答

- 競合研究グループは多いのか．それはどういったグループなのか．

  競合グループは非常に多い．日本国内，欧米，アジアの世界各地で研究グループが存在し，また緊密に連携をとりあって研究を進めている．しかし，現時点では，理論的，工学的な面に対する研究が多いのに対して，その応用面についての研究はまだまだ発展途上である．今後さらに，自然言語処理，ゲノム解析，機械学習などの研究分野との交流が進むにしたがって，双方単独では発見し得なかった圧縮索引の応用例が出てくる可能性がある．また，テキストではなくツリーやグラフ構造の情報の索引といった技術も近年登場しつつある [1]．例えば構文解析や情報抽出といったタスクにおいては，木構造の情報を用いることにより精度が上がる報告がいくつかなされている [10] [18]．今後，応用面の需要からも，圧縮索引技術はより複雑なデータ構造に適用されると考えられる．

- IIWT では同一箇所に複数のパターンを登録できるのか

  そのままでは，IIWT は同一箇所に複数のパターンを登録できないが，IIWT のデータ構造の一部変更することで可能である．その変更は，重複を許した T_p に加えて，元の各ポジションの最後の位置には 1，それ以外の位置には 0 である bit 配列である delimiter bit 配列 D を扱うことで実現できる．詳細は述べないが，これを用いることにより，$rank_c(T, i)$ を求める場合は，最初に $p = select_1(D, i)$ を用いて，複数パタンを含めた T_p 中の対応する位置を求めてから $rank_c(T_p, p)$ で求められる．$select_c(T, i)$ を求める場合は，$p = select_c(W, i)$ を求めてから $rank_1(D, p)$ を行う．

- 大規模なデータへの分散はどうなのか

  CSA や CST は元データでは隣接していた情報が，分散される性質があるので，分散処理をうまく行うのは難しいと思われる．また，構築方法についても分散処理で効率良く処理する方法は，現実的な方法としてはまだ無いと思われる．しかし，一番単純な方法であり、かつ現実的な方法として，最初にデータを分割しておき，それぞれに対し索引を構築し，クエリーが与えられた時は，それぞれの索引で独立に処理結果を求め，その結果マージする方法が考えられる．この方法では，いくつかの操作は求められないが，それでも出現回数や全出現位置などは正しい結果が保障される．この方法では，分散数が D の場合でも，時間計算量は元の $\frac{1}{D}$ とはならずに元と同じ時間計算量のままだが，これにより処理可能なデータサイズを容易にスケールアップすることができる．一般に転置ファイル等の索引では，ハードディスクなど 2 次記憶領域の利用を前提としており，今回紹介した手法とは応用範囲が重ならないと思われる．しかし，大規模なデータを索引するためにはハードディスクを利用しても処理できることが望まれる．今回紹介した IIWT はそのような 2 次記憶領域の利用が可能であり（構築時がメモリーを消費せずに逐次的に処理可能），それを利用して，TB 以上の大規模なデータに対する索引付けが可能

だと考えられる..

参考文献

[1] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *FOCS*, 2005.

[2] R. Geary., N. Rahman., R. Raman., and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. of CPM*, pages 159–172, 2004.

[3] W. Golomb. Run-length encodings. *IIEEE Trans. on Information Theory*, 1966.

[4] R. Gonzalez., S. Grabowski., V. Makinen., and G. Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, 2005.

[5] R. Grossi., A. Gupta., and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of SODA*, pages 841–850, 2003.

[6] R. Grossi., A. Gupta., and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. of SODA*, pages 636–645, 2004.

[7] R. Grossi. and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 2005.

[8] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[9] D. K. Kim., J.C. Na., J.E. Kim., and K. Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. of WEA*, 2005.

[10] T. Kudo., J. Suzuki., and H. Isozaki. Boosting-based parse reranking with subtree features. In *Proc. of ACL*, 2005.

[11] A. Moffat and V. Anh. Binary codes for non-uniform sources. In *Proc. of DCC*, pages 133–142, 2005.

[12] A. Moffat. and L. Stuiver. Binary interpolative coding for effective index compressoin. *Information Retrieval*, 3(1):24–47, 2000.

[13] J. I. Munro. Tables. In *Proc. of FSTTCS*, pages 37–42, 1996.

[14] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Computation*, 31(2):353–363, 2001.

[15] R. F. Rice. Some practical universal noiseless coding techniques. Technical report, Technical Report 79-22 Jet Propulsion Laboratory, Pasadena, California, 1979.

[16] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*.

[17] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Algorithms*, 48(2):294–313, 2003.

[18] M. Zhang., J. Su., D. Wang., and G. Zhou. Discovering relations from a large raw corpus using tree similarity-based clustering. In *Proc. of IJCNLP*, 2005.