圧縮データ構造とその最新動向
The Latest Trend of Compressed Data Structures

定兼　邦彦
Kunihiko Sadakane

九州大学大学院システム情報科学研究院
Department of Computer Science and Communication Engineering
Kyushu University

Fukuoka 812-8581, Japan
sada@csce.kyushu-u.ac.jp

概要

大量のデータを活用するには，それを高速に検索できるようにする必要がある．しかしそのためのデータ構造の大きさが問題となっている．従来のデータ構造ではデータ自身のサイズよりも検索のためのデータ構造の方が大きくなってしまうことが多いが，近年開発された簡潔データ構造 (succinct data structures) ではデータ構造は極限まで小さくなっている．本稿ではまず基本的な簡潔データ構造である，集合，順序木，文字列を表現するものを解説する．次に最新の結果である，簡潔データ構造をさらに圧縮する手法を解説する．これはデータの冗長性を利用し，データをそのエントロピーまで圧縮するものである．通常の圧縮法では復元は低速だが，新しい圧縮法では任意の部分を定数時間で復元することができる．これにより，データ構造が圧縮してあっても従来と同じ計算量で問い合わせを行うことができる．

Keywords: データ構造, 圧縮, 情報検索

1　はじめに

大量のデータを活用するには，データをコンパクトに格納し，かつ高速に検索できるようにする必要がある．元データよりもサイズを小さくすることはデータ圧縮と呼ばれる．また，高速に検索を行うには索引と呼ばれるデータ構造を付け加える必要がある．つまり，圧縮と高速な検索を両立することはできないようにみえる．しかし，アルゴリズムの進化により，これらを両立させることが可能になった．本稿ではそのための圧縮アルゴリズムと検索アルゴリズムの基本を解説し，また最新の結果も述べる．

従来のデータ圧縮は，格納スペースの節約や，通信コストの削減のために用いられており，頻繁に用いられるデータに対しては圧縮は適用されないことが多い．これは圧縮データの復元に時間がかかるためである．データ圧縮の基本は，頻繁に現れる要素に短い符号を割り当てることである．よって，非圧縮データではランダムアクセスは容易だが，圧縮データではそれは不可能であり，先頭から１つずつ復号する必要がある．復号時間を短くするには，データを複数のブロックに分割し，それぞれを独立に圧縮すればいいが，圧縮率は低下する．また，圧縮率を上げるためにはデータ間の相関を利用する必要があるが，そのために

は圧縮・復元時にデータを逐次的に走査する必要がある．文字列などの1次元データに対しては，データは k 次マルコフ情報源から発生したと仮定することが多い．これは，ある文字の出現確率が，その直前の k 文字のみから決まる情報源である．よって，圧縮する場合には文字列の先頭から順に符号化し，復元する場合も先頭から順に行う必要がある．つまり一部分だけ復元することができない．

また，検索のための索引のサイズについては，従来の評価基準では索引を格納するために必要なメモリセルの数を用いていたが，これは現実的ではない．なぜなら，1つのメモリセルは1つのポインタを格納できると仮定するが，ポインタの占めるビット数はデータ量に依存するからである．例えば，n 要素を格納する二分木は $\mathrm{O}(n)$ 個のポインタで表現されるが，1つのポインタは $\mathrm{O}(\log n)$ ビット必要であるため，全体で $\mathrm{O}(n \log n)$ ビットの領域が必要となる．従来は扱うデータ量が少なかったため問題にはならなかったが，データ量が増えるに従って，索引の占める領域の大きさが問題となってきている．索引が大きくなると，それを格納するためには低速のメモリ，ディスクを用いる必要があり，検索速度が低下してしまう．

これらの問題を解決するために最近盛んに研究されているのが，簡潔データ構造 (succinct data structures) である．これは，データ構造のサイズを極限まで小さくし，かつ，従来のデータ構造と同じ (またはほぼ同じ) 計算量で処理を行えるものである．サイズの下界は，情報理論的下界 (information-theoretic lower bound) と呼ばれる．基数 L のある集合に含まれる1つの要素を表現するデータ構造のサイズの情報理論的下界を $\log L$ ビット[1]と定義する．簡潔データ構造は，$(1+\mathrm{o}(1))\log L$ ビット，または $\mathrm{O}(\log L)$ ビットの領域を用いて従来のデータ構造と同じ操作を行う．つまり，従来のデータ構造では領域計算量はメモリセルの数で評価していたが，それをビット数で評価することでより厳密な評価をしている．

簡潔データ構造では，計算モデルとして word RAM を用いる．このモデルでは，計算機は U ビットのメモリを持ち，計算機の語長は $\log U$ ビットとする．つまり，$\log U$ ビットの数の算術・論理演算，および $\log U$ ビットのメモリの読み書きが定数時間で行える．このモデルは RAM モデルよりも自然なモデルである．なぜなら通常の RAM モデルでは1つのメモリセルに格納できるビット数に制限がないため，情報を制限なく格納できることになるからである．また，word RAM モデルで $\log U$ ビットのメモリの読み書きができるという条件は，ポインタの操作を定数時間で行うことを意味するが，この仮定も自然である．なお，多くの簡潔データ構造では計算機の語長を $\log U$ ではなく $\mathrm{O}(\log n)$ ビットとしている (n は扱うデータの量)．これは，データを格納するには $\mathrm{O}(n)$ ビットのメモリが必要だからである．

本稿では，基本的な簡潔データ構造として，集合，順序木を表現するもの，文字列を表現するものを解説する．さらに，最新の結果にもふれる．

2 基本的な簡潔データ構造

2.1 集合の表現

最も基本的なデータ構造は順序集合を表すものである．集合 S が $U = \{0, 1, \ldots, m-1\}$ の中の n 個の要素を含んでいるとする．S に対する問合せとして最も基本的なものは *membership* である．これは，ある x が S に含まれているかどうかを答えるものである．も

[1] log の底は 2.

し S の要素がソートされているとすると，この問合せは $\mathrm{O}(\lg n)$ 時間[2]でできる [36]．これを定数時間にする方法としてはハッシュ表を用いるものがある [11]．しかしハッシュ表を用いると要素を並び替えてしまうため，S の中で i 番目に小さいものを求めることが難しくなる．一方，要素がソートしてあるとこれは定数時間で求まる．また，ある要素が存在していることが分かった場合に，そのランク (S 中でそれより小さい要素の数) も分かる．ここで，ソートされた配列とハッシュ表という 2 つの表現の利点を持つようなデータ構造を考える．このデータ構造は以下のような演算ができるとする．

- *rank*(x, S): $x \in U$ に対し $x \in S$ ならば $|\{y \in S | y < x\}|$ を返し，そうでなければ -1 を返す．
- *select*(i, S): $i \in \{1, 2, \ldots, n\}$ に対し，S の中で i 番目に小さい要素を返す．

これらをサポートするデータ構造は索引付辞書 (*indexable dictionary*) と呼ばれる．もちろん上記の 2 つの表現を両方用いればこれらの演算は定数時間で実現できるが，ここでは圧縮データ構造を考える．データ構造のサイズは情報理論的下限になるべく近くすることが目標である．集合 $U = \{0, 1, \ldots, m-1\}$ のサイズ n の部分集合の個数は $\binom{m}{n}$ であるから，この問題での下限は $\lceil \lg \binom{m}{n} \rceil$ ビットとなる．この値を $\mathcal{B}(n, m)$ で表す．$\mathcal{B}(n, m) = n \lg \frac{em}{n} - \Theta(\frac{n^2}{m}) + \mathrm{O}(\lg n)$ である (e は自然対数の底)．S をソートされた配列で表すとサイズは $n \lceil \lg m \rceil$ ビットであるため，これも下限よりは大きい．この問題は非常に基本的なものであり，多くの研究がされている [28, 20, 24, 3, 31, 30]．これらは *rank*, *select* を定数時間で行なえるが，その中で最もサイズの小さいものは Raman, Raman, Rao [30] のものであり，サイズは $\mathcal{B}(n, m) + \mathrm{o}(n) + \mathrm{O}(\lg \lg m)$ ビットである．

索引付辞書は m ビットの 0,1 ベクトル B でも表現できる．$i \in S$ のとき $B[i] = 1$，そうでないときは $B[i] = 0$ とする．このとき *rank*(x, S) は $B[0 \ldots x]$ の 1 の数，*select*(x, S) は B 中の左から x 番目の 1 の位置となる．$\mathcal{B}(n, m) \leq m$ より，索引付辞書はこのベクトルを圧縮した形式で保存しているとみなすことができる．

完全索引付辞書 (fully indexable dictionary, FID) は *rank* と *select* を S と $\bar{S} = U \backslash S$ の両方について定数時間で答えるものである．これは上のベクトル B において 0 の *rank* と *select* を求めることに相当する．FID のサイズは $\mathcal{B}(n, m) + \mathrm{O}(m \lg \lg m / \lg m)$ である．0 と 1 に対する *rank* が求まることは，次の演算が行なえることを意味する．

- *fullrank*(x, S): 任意の $x \in U$ に対し $|\{y \in S | y < x\}|$ を返す．

FID のデータ構造のサイズの下限もいくつか知られている．RAM モデルでは $n^{\mathrm{O}(1)}$ 語のデータ構造では *fullrank* を定数時間で求めることはできないことが示されている [1]．また，S を長さ m のビットベクトル B で表したとき，1 の *rank* を定数時間で求めるための補助データ構造のサイズは $\Omega(m \lg \lg m / \lg m)$ ビット [23]，1 の *select* を定数時間で求めるための補助データ構造のサイズも $\Omega(m \lg \lg m / \lg m)$ ビットであることが示されている [14]．FID の補助データ構造のサイズは *rank*，*select* ともに $\mathrm{O}(m \lg \lg m / \lg m)$ ビットであるため，これらはタイトである．

rank と *select* は 0,1 ではない一般のベクトルに対しても定義できる．つまり，$B[i] \in \mathcal{A} = \{1, 2, \ldots, \sigma\}$ $(1 \leq i \leq n)$ であるとしたときに，$\mathit{rank}_c(i, B)$ を $B[0 \ldots i]$ の中の c の数

[2] $\lg n = \log_2 n$

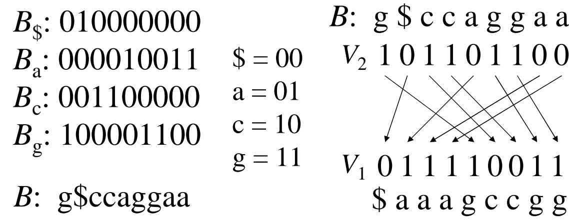

図 1: $B = g\$ccaggaa$ のウェーブレット木．

$(c \in \mathcal{A})$，$select_c(i, B)$ を i 番目の c の位置と定義する．これらの演算は後述する文字列検索のデータ構造でも用いられる．0,1 ベクトルを用いて一般のベクトルの $rank$ と $select$ を実現する簡単な方法としては，σ 個の長さ n の 0,1 を用いるものがある．つまり，0,1 ベクトル $B_c[0 \ldots n-1]$ $(c \in \mathcal{A})$ は，$B[i] = c$ のとき $B_c[i] = 1$，それ以外のとき $B_c[i] = 0$ とする．すると $rank_c(i, B) = rank(i, B_c)$, $select_c(i, B) = select(i, B_c)$ となる．ただし $B[i]$ を求めるには $B_c[i] = 1$ となる c を見つける必要がある．

このデータ構造の欠点は，サイズが冗長であるという点である．各 B_c を圧縮せずに格納するとサイズは $n\sigma + \mathrm{O}(n\sigma \lg\lg n/\lg n)$ ビットである．B のサイズは $n \lg \sigma$ であるためその差は大きい．各 B_c を FID で表現するとサイズは $\sum_{c \in \mathcal{A}} \left(n_c \lg \frac{en}{n_c} + \mathrm{O}(n \lg\lg n/\lg n) \right) = nH_0(B) + n \lg e + \mathrm{O}(n\sigma \lg\lg n/\lg n)$ ビットとなる．なお，n_c は B 中の c の頻度であり，$H_0(B)$ は B の 0 次の経験的エントロピー [21] である．$nH_0(B)$ は 各 n_c を固定した場合の情報理論的下界となる．つまり，このデータ構造には $n \lg e \approx 1.44n$ ビットの無駄が存在する．B を英語などの文章だとするとそれを圧縮すると 1 文字当たり 2 ビット程度になるため，1.44 ビットは無視できない大きさである．冗長である理由は，各 i に対し $B_c[i] = 1$ となる c は唯一であるが，B_c を独立に圧縮しているためその情報を用いていないからである．これを改善するためにウェーブレット木 (wavelet trees) が提案されている [15].

ウェーブレット木は $\lg \sigma$ 個のレベルからなる層状のデータ構造であり，各レベルは長さ n の 0,1 ベクトル V_i を格納する．各文字 $B[i]$ $(1 \le i \le n)$ は $\lg \sigma$ ビットの 0,1 列で表現できるが，最上位レベルのベクトルは $V_{\lg \sigma}[i] = 1 \iff (B[i]$ の最上位ビットが 1) と定める．1 つ下のレベルは，文字の最上位ビットが 0 であるものを左に，1 であるものを右にそろえた文字列を仮想的に考え，各文字の上から 2 ビット目を $V_{\lg \sigma - 1}$ として格納する．これを再帰的に繰り返す．図 1 は $B = g\$ccaggaa$ に対するウェーブレット木を表す．図の矢印をたどると文字が復元されるが，矢印は V_i での $rank$ で表現できるため明示的に格納する必要はない．文字 $B[i]$ の復元は $\mathrm{O}(\lg \sigma)$ 時間でできる．また，$rank_c(i, B)$ や $select_c(i, B)$ も $\lg \sigma$ 回の $rank$ と $select$ で計算できるため，$\mathrm{O}(\lg \sigma)$ 時間で求まる．また，データ構造のサイズは $nH_0(B) + \mathrm{O}(n \lg \sigma \lg\lg n/\lg n)$ ビットとなる．つまり冗長な $1.44n$ ビットを削除できる．この方法では $rank_c$ や $select_c$ が定数時間では求まらないが，$\sigma = \mathrm{polylog}(n)$ のときにこれらが定数時間で求まるデータ構造でサイズが $nH_0(B) + \mathrm{o}(n)$ ビットのものが提案されている [10].

2.2 順序木

二分木はデータ検索のための基本的なデータ構造である．データは木の内部ノードおよび葉に蓄えられる．データを葉のみに格納する場合には，内部ノードはデータの一部を格納する．

木構造を表現するにはポインタなどを使う必要がある．二分木ならば各内部ノードに対し左の子，右の子へのポインタが必要である．ポインタのサイズは $\lg n$ ビットであるため，木のサイズは $O(n \lg n)$ ビットとなる．本節では n ノードの木の構造を $O(n)$ ビット，正確には $2n+o(n)$ ビットで表現する方法を解説する．なお，n ノードの二分木の数は $\binom{2n}{n}/(n+1) \approx 2^{2n}/n^{3/2}$ 個あるため，この表現は低次の項を無視すれば最適である．

二分探索木を用いてデータを検索するには，次のような木の上での演算が必要となる．

- *lefttree*(x): ノード x の左の部分木を求める．
- *righttree*(x): ノード x の右の部分木を求める．
- *parent*(x): ノード x の親を求める．
- *preorder*(x): ノード x の行きがけ順序を求める．
- *inorder*(x): ノード x の通りがけ順序を求める．

行きがけ順，通りがけ順はノードにデータを格納するために必要である．これらの操作を括弧列の上で行う．

2.2.1 BP 表現

順序木の簡潔データ構造は何種類か存在するが，最も有名なものは括弧列表現 (balanced parenthesis representation, 以下 BP) である [25]．木 T の根ノードの部分木をそれぞれ $T_1, T_2, \ldots, T_d$ と表すと，T に対する括弧列表現 $BP(T)$ は次のように定義される．

定義 1

$$BP(T) = \begin{cases} () & (T \text{ が 1 ノードのみ}) \\ (\ BP(T_1)BP(T_2)\cdots BP(T_d)\ ) & (\text{それ以外}) \end{cases}$$

つまり，木の各ノードは開き括弧 “(” と閉じ括弧 “)” で表され，そのノードを根とする部分木はその括弧の中に符号化されている．この定義から，括弧列では括弧の対応が取れていることがわかる．

検索木を用いたデータ検索を木の括弧列表現 P を用いて行うには，木の巡回などの操作に対応する操作を P 上で行う必要がある．そのためにまず P 上での基本的な演算を定義する [25].

- *findclose*(i): $P[i]$ にある開き括弧に対応する閉じ括弧の位置を返す．
- *findopen*(i): $P[i]$ にある閉じ括弧に対応する開き括弧の位置を返す．

- $enclose(i)$: $P[i]$ にある開き括弧とそれに対応する閉じ括弧を内部に含むような最小の括弧対の位置を返す.

これらの関数の引数，返り値は共にあるノードを表す開き括弧の位置である．これらの操作は定数時間で行える．そのために必要なデータ構造のサイズは $o(n)$ ビットであり，木の上での操作はこれらの演算を用いて次のように表せる [25].

- $lefttree(x) = x + 1$
- $righttree(x) = findclose(x + 1) + 1$
- $parent(x) = enclose(x)$

図 2 は木の括弧列表現の例である.

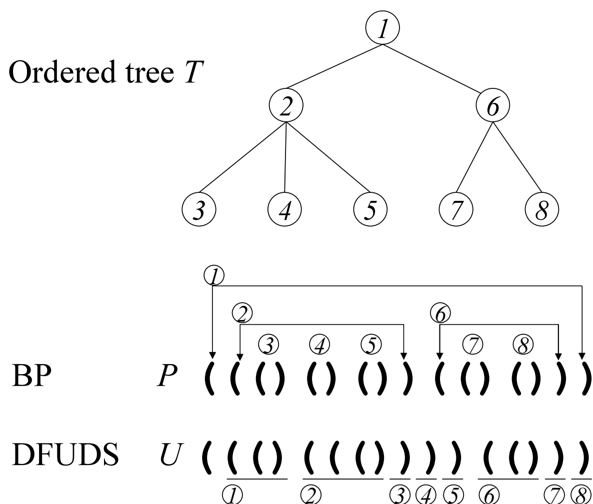

図 2: 順序木の括弧列表現.

木の内部ノード，葉ノードはどちらもそれに対応する開き括弧の位置で表現できる．さらに，その位置とノードの行きがけ順 p は互いに定数時間で変換できる:

補題 1 接尾辞木のノードの行きがけ順 p と，そのノードに対応する開き括弧の位置 i は次のように定数時間で変換できる.

$$
\begin{aligned}
p &= preorder(i) \equiv rank_{(}(P, i) \\
i &= select_{(}(P, p).
\end{aligned}
$$

なお，$rank$, $select$ 演算は $0, 1$ ではなく (,) のベクトルに対し定義する.

BP 表現では以下の演算が定数時間で行える [25, 26, 33, 27, 12, 13]: 親，最初の子，次の弟への移動，部分木のサイズ，2 つのノードの lca (lowest common ancestor)，最左/最右子孫，子の数，祖先 (level-ancestor)．ただし，あるノードの i 番目の子を求めるには，子の数に比例した時間がかかる．なぜなら，子を求めるには $findclose$ を用いるが，左から順に 1 つずつ求めていくからである.

2.2.2　DFUDS 表現

DFUDS 表現 (depth-first unary degree sequence representation) は BP とは別の順序木の表現法である [2]．サイズは BP と同じで $2n + \mathrm{o}(n)$ ビットである．DFUDS でもノードは開き括弧と閉じ括弧で表現されるが，ノードごとにそれを表現する括弧の数は異なる．ノードは根から深さ優先探索の順 (行きがけ順) に括弧列に格納される．子供の数が d であるノードは，d 個の開き括弧（と 1 つの閉じ括弧）の列で表現される．葉ノードは）のみで表現される．このようにして作られた括弧列の先頭にダミーの開き括弧を 1 つ追加すると，全体は括弧の対応の取れた列になる (図 2 参照)．この表現では，前節で挙げた演算のうち，lca 以外はすべて定数時間で行える．特に，i 番目の子を求める演算は，BP では子の数に比例した時間がかかっていたのに対し，DFUDS では定数時間で行える点が大きく異なる．一方，lca を求めることはできない．

2.3　文字列・接尾辞配列

文字列検索における基本的な問題は，長さ n の文字列 T の中から長さ m のパタン P を探し，その出現位置を答えるものである．従来のデータ構造としては接尾辞木や接尾辞配列が広く用いられている [19]．これは T に対して検索のためのデータ構造 $I(T)$ を作成しておき，P に対する問合せがあった時に T と $I(T)$ を用いて検索を行なうものである．検索時間は接尾辞木の場合に $\mathrm{O}(m)$，接尾辞配列の場合に $\mathrm{O}(m \lg n)$ である．

接尾辞配列は以下のようなデータ構造である．文字列 $T = T[1 \ldots n]$ の接尾辞とは，T の部分文字列 $T_j = T[j \ldots n] = T[j]T[j+1] \cdots T[n]$ $(j = 1, 2, \ldots, n)$ であり，T の接尾辞配列 SA はサイズ n の配列で，$SA[i] = j$ であることは 接尾辞 T_j が辞書順で i 番目であることを意味する．つまり接尾辞配列を用いればパタンを 2 分探索で探すことができる．

接尾辞木や接尾辞配列を用いれば高速な文字列検索が実現されるが，文字列のサイズが $|T| = n \lg \sigma$ ビット，データ構造のサイズが $|I(T)| = \mathrm{O}(n \lg n)$ ビットであるため，サイズの下限との間には大きな差がある．なお，サイズの下限の定義は難しいが，文字列中の各文字 c の頻度 n_c を固定した場合は下界は 0 次の経験的エントロピーを用いて $nH_0(T)$ と表現できる．文字列でよく用いられるものは k 次の経験的エントロピー $H_k(T)$ である．これは文字 $T[i]$ の出現確率が $\Pr[T[i] = c | T[1 \ldots i-1]] = \Pr[T[i] = c | T[i-k \ldots i-1]]$ と表せると仮定した場合のエントロピーである．文字列 $T[i-k \ldots i-1]$ は $T[i]$ の次数 k の文脈と呼ばれる．T の部分文字列で文脈が α である文字のみからなるものを w_α とするとき，$H_k(T) = \sum_{\alpha \in \mathcal{A}^k} |w_\alpha| H_0(w_\alpha)$ と定義する．なお $H_k(T) \leq H_{k-1}(T) \leq \cdots \leq H_0(T) \leq \lg \sigma$ である．

文字列検索のための圧縮データ構造としては圧縮接尾辞配列 [18, 16, 34, 33, 32, 22, 17] や FM-index [6, 8, 9] が存在する．これらのデータ構造ではまず T を別の文字列 W に変換し，それに対してパタンを検索のためのデータ構造 $I(W)$ を作成する．また，W での検索結果から T での検索結果 (パタンの出現位置) を求めるためのデータ構造 $J(T, W)$ も作成する．さらに W は圧縮して格納する．このとき W の圧縮後のサイズが $nH_k(T)$，$I(W)$ と $J(T, W)$ のサイズを $\mathrm{o}(n)$ ビットにすることができる．なお，このデータ構造では検索には T は用いない．また，W と $J(T, W)$ から T は復元できるため T を保存する必要がない．以下ではこのデータ構造を解説する．

まず，T から W を作成する．W は T と T の接尾辞配列から定義され，$W[i] = T[SA[i]-1]$ とする．ただし $SA[i] = 1$ のときは $W[i] = T[n]$ とする．この変換は Burrows-Wheeler 変換 [4] と呼ばれ，圧縮プログラム bzip2 で用いられているものである．この変換は単なる文字の並び替えであるため W のサイズは T と同じであるが，W は容易に圧縮できる (後述).

与えられたパタン P を T から検索する場合，まず W を用いて P と一致する T の接尾辞 T_j の辞書順 i を求める [6]．データ構造 $I(W)$ は W 上での $rank_c$ を求めるものである．計算時間は $\mathrm{O}(mt(\sigma))$ となる．ここで $t(\sigma)$ はアルファベットサイズ σ の文字列での $rank_c$ や $select_c$ を求める時間である．

次に，P の辞書順 i からその出現位置 $j = SA[i]$ を求める必要がある．これはサイズ $\mathrm{O}(n/\lg^{\epsilon} n)$ ビットのデータ構造 $J(T, W)$ を用いて $\mathrm{O}(\lg^{1+\epsilon} n)$ 時間で求まる [8].

最後に W の圧縮であるが，ウェーブレット木を用いると $nH_0(W)$ ビットに圧縮できる．W の文字は T の文字を並び替えたものなので 0 次のエントロピーは等しい．つまり $nH_0(W) = nH_0(T)$ となる．これを $nH_k(T)$ まで圧縮するには W をいくつかの文字列に区切ってそれぞれを独立にウェーブレット木で圧縮する．分割は一般には文字の文脈で定義される．接尾辞配列の定義から，W では同じ文脈を持つ文字が連続して現れるため，同じ文脈を持つ文字を 0 次のエントロピーまで圧縮できれば全体で k 次のエントロピーまで圧縮したことになる．圧縮にはロスがあるため文脈の数が多すぎると圧縮率が低下してしまうが，圧縮率を最大にする分割は $\mathrm{O}(n)$ 時間で求まる [7].

3 最近のテーマ

簡潔データ構造に関する最近のテーマは，データの冗長性を考慮したデータ構造である．従来の簡潔データ構造の多くは，どんな入力に対してもそれを同じビット数で表現している．例えば順序木の BP や DFUDS 表現では，ノード数が n の場合にはデータ構造は常に $2n$ ビットとなる．しかし，これらの表現は木構造の冗長性を考慮していない．例えば，木が 2 分木で，かつ全内部ノードがちょうど 2 つの子を持つ場合，そのような木は n ビットで表現できる．しかしそのような表現では木の上での演算を定数時間で行うことができない．一方，BP や DFUDS を用いると $2n$ ビット必要であるため，そのサイズは最適とはいえない．そこで，木の冗長度を定義し，冗長度に従った大きさまでデータ構造を圧縮したい．

第 2.1 節で述べたように，集合を表現する簡潔データ構造ではすでにデータの冗長性を考慮したものが提案されている [28, 30]．これらは集合 $U = \{0, 1, \ldots, m-1\}$ のサイズ n の部分集合を $\mathcal{B}(n, m) = \lceil \lg \binom{m}{n} \rceil$ ビット程度で表現するものだが，これは U を表現する 0,1 ベクトルの 0 次のエントロピーである．しかしこの方法では BP や DFUDS の括弧列を圧縮することができない．なぜならこれらの表現では開き括弧が n 個，閉じ括弧が n 個であり，0 次のエントロピーは $2n$ ビットであるからである．そこで括弧列を圧縮するには，高次のエントロピーを考慮する必要がある．

文字列の高次エントロピーを達成する圧縮法には前述の圧縮接尾辞配列や FM-index があるが，これらでは文字列の指定された場所を定数時間で復元することができない．よって，FID を用いていたものをこれらのデータ構造で置き換えた場合，サイズは小さくなるが問い合わせの時間計算量が増えてしまう．

最近提案された EID [35] は，FID の機能をそのままにサイズを高次エントロピーまで圧縮

したものである．これは U を表現する 0,1 ベクトルを圧縮したものであるが，もし圧縮された文字列の部分文字列を高速に復元できるならば，文字列は仮想的に圧縮されていないと見なすことができ，従来と同じアルゴリズムを用いることができる．このような考えに基づくデータ構造がいくつか提案されている．

3.1 部分復元可能な圧縮法

文字列の任意の部分を高速に部分復元可能なデータ構造が提案された [35]．データ構造のサイズは $nH_k + \mathrm{O}\left(\frac{n(\lg\sigma+\lg\lg_\sigma n+k)}{\lg_\sigma n}\right)$ であり，$decode(S,i,l)$ の時間は $\mathrm{O}(l\lg\sigma/\lg n)$ である．つまり文字列中の連続する $\lg n$ ビットを定数時間で復元できる．これは $\lg n$ ビットのメモリを定数時間でアクセスできるという計算モデルにおいては最適である．

この圧縮法は，LZ78 圧縮法 [37] を基にしている．LZ78 の圧縮，伸張は，文字列 S を分解した部分文字列 s_i (フレーズと呼ぶ) とそれに割り当てた番号 i をメモリに格納し，それを用いて行なう．これらの情報は圧縮後のデータに含める必要はなく，伸張時に再構成できるが，再構成するには文字列の先頭から伸張していく必要がある．よって部分復号に全体の復号と同じ時間がかかってしまう．新しいデータ構造では，これらの情報も格納しておくことで部分復号を高速化している．

LZ78 によって生成されたフレーズは，LZ トライと呼ばれる図 3 のような木で表現できる．各フレーズは木のノードで表され，根からそのノードまでのパス上の枝ラベルを連結したものがそのフレーズとなる．ノード内の数字はフレーズの番号を表す．図の配列 R は各フレーズの番号を格納し，配列 P は各フレーズの開始位置を格納する．従来の LZ78 では R の値は単調増加だが，新しいアルゴリズムでは各 $R[i]$ の値はそのフレーズを表す LZ トライのノードの先行順 (preorder) と一致するようにする．こうすることで余分な変換表が必要なくなり，LZ トライを表すデータ構造のサイズが小さくなる．LZ トライの木構造は括弧列 E で表現し，枝ラベルは配列 C に格納する．木は括弧列で表されていても木の巡回は通常通り行なえる [25]．

$decode(S,i,l)$ を実行する場合，まず $S[i]$ を含むフレーズを求める．これは P を用いて定数時間で求まる．P のサイズは $\mathrm{O}(n\lg\lg n/\lg n)$ ビットである [29]．次に配列 R からそのフレーズの番号を求め，括弧列 E を用いて先行順がフレーズ番号と等しいノードを求める．そして根からそのノードへのパス上の文字列を求めればフレーズを復元できる．

LZ トライのパス上の長さ $w = 1/2\lg_\sigma n$ の文字列の復元は，1 文字ずつ復元するのは簡単だが長さに比例した時間がかかってしまう．これを定数時間で行なうには別のデータ構造が必要である．あるノード v から根へのパスを求めるとする．このノードがサイズが $w/2$ 以下の部分木に含まれる場合，そのような木は $3/4\lg n$ ビットで表現でき，その中のパスの復元は $\mathrm{O}(n^{3/4}\lg^2 n)$ ビットの表を用いて定数時間で行なえる．よって LZ トライからそのような部分木を全て削除した木を別に作成する．LZ トライのノード数はフレーズ数と等しい c だが，サイズの小さい部分木を削除した木の葉の数は $4c/\lg_\sigma n$ 以下となる．すると枝分かれをしているノードの数はそれ未満となる．各枝分かれノードに対してはそこから根に向かう長さ w のパスをそのまま格納する．枝分かれをしていない部分については配列 C からパスが求まる．

このアルゴリズムにより 1 つのフレーズ中の $1/2\lg n$ ビットを定数時間で復元することができるが，復元したい w 文字が複数のフレーズで表されている場合には復元が定数時間で

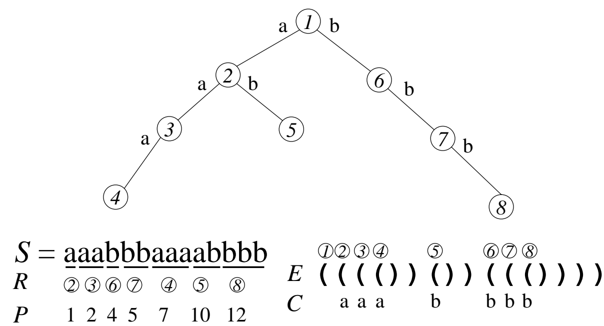

図 3: LZ78 で圧縮された $S = \mathsf{aaabbbaaaabbbb}$ を表すデータ構造.

は終わらない場合がある．そこで，長さが w 未満のフレーズについては，それが S 中で連続して現れている場合はそれらを含む S の $1/2 \lg n$ ビットをそのまま格納する．そしてこれらのフレーズについてはそれらを表すフレーズの番号を配列 R に格納しない．LZ78 の性質により，フレーズの数 c は $\sqrt{n}$ 以上であり，1 つのフレーズの添字を符号化するには $1/2 \lg n$ ビット以上必要である．よってフレーズを R とトライを用いて符号化するよりも，フレーズをそのまま符号化する方がビット数が少なくなる．つまり，このようにデータ構造を変更しても LZ78 による圧縮サイズ以下になることがわかる．

3.2 ラベル付き木

ノードにラベルがついた順序木を考える．ラベルはアルファベット $\mathcal{A}$ 中の文字だとすると，n ノードのラベル付き木は木構造は $2n$ ビット，ラベルは $n \lg \sigma$ ビットで表現できる．この木の上での演算としては部分パス問合せ $subpath(P)$ を考える．これはノードのうちでそこから根に向かうパスが P で始まるようなものを求める問合せである．これを効率よく行なうためにラベルをある規則で並び替える．この変換を xbw と呼ぶ [5]．その規則は，各ノードのラベルをそこから根に向かうパス上のラベルの辞書順でソートするものである．これは文字列に対する Burrows-Wheeler 変換の一般化になっている．変換と逆変換はともに $\mathrm{O}(n)$ 時間で行なえる．この変換後のラベル列と，木構造を表す $2n$ ビットを用いて，ノードの親，子供の数，指定されたラベルを持つ子供を定数時間で見つけることができる ($\sigma = \mathrm{polylog}(n)$ のとき)．また，部分パス問合せも $\mathrm{O}(|P|)$ 時間で行なえる．ただし，ノードの深さを定数時間では求める方法は知られていない．データ構造のサイズは更に圧縮できる．ラベル付き木のエントロピー $H_k(T)$ を，長さ k のパスを用いて定義するとき，このデータ構造のサイズは $nH_k(T) + (2 + \epsilon)n$ ビットにできる ($\epsilon > 0$ は符号化の誤差)．

参考文献

[1] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. In *Proc. ACM STOC*, pages 295–304, 1999.

[2] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing Trees of Higher Degree. *Algorithmica*, 43(4):275–292, 2005.

[3] A. Brodnik and J. I. Munro. Membership in Constant Time and Almost-minmum Space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[4] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report, 1994.

[5] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. IEEE FOCS*, pages 184–196, 2005.

[6] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. Technical Report TR00-03, Dipartimento di Informatica, Università di Pisa, March 2000.

[7] P. Ferragina and G. Manzini. Optimal compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[8] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[9] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An Alphabet-Friendly FM-index. In *Proc. SPIRE*, 2004. To appear.

[10] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Succinct Representation of Sequences. Technical Report TR/DCC-2004-5, Dept. of Computer Science, Univ. of Chile, August 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.

[11] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *JACM*, 31(3):538–544, 1984.

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

[13] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–10. Society for Industrial and Applied Mathematics, 2004.

[14] A. Golynski. Optimal Lower Bounds for Rank and Select Indexes. In *Proc. ICALP*, LNCS 4051, pages 370–381, 2006.

[15] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. ACM-SIAM SODA*, pages 841–850, 2003.

[16] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. ACM-SIAM SODA*, pages 841–850, 2003.

[17] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. ACM-SIAM SODA 2004*, pages 636–645, 2004.

[18] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[19] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[20] G. Jacobson. Space-efficient Static Trees and Graphs. In *Proc. IEEE FOCS*, pages 549–554, 1989.

[21] R. Kosaraju and G. Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[22] V. Mäkinen and G. Navarro. Compressed Compact Suffix Arrays. In *Proc. CPM*, LNCS 3109, pages 420–433, 2004.

[23] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. ACM-SIAM SODA*, pages 11–12, 2005.

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

[25] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[26] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39(2):205–222, May 2001.

[27] J. I. Munro and S. S. Rao. Succinct Representations of Functions. In *Proceedings of ICALP*, LNCS 3142, pages 1006–1015, 2004.

[28] R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[29] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-aray Trees and Multisets. In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.

[30] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.

[31] V. Raman and S. S. Rao. Static dictionaries supporting rank. In *Proc. ISAAC*, pages 18–26, 1999.

[32] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

[33] K. Sadakane. Succinct Representations of *lcp* Information and Improvements in the Compressed Suffix Arrays. In *Proc. ACM-SIAM SODA*, pages 225–232, 2002.

[34] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[35] K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. ACM-SIAM SODA*, pages 1230–1239, 2006.

[36] A. C. Yao. Should Tables Be Sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[37] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, IT-24(5):530–536, September 1978.