文字列辞書を用いた効率的な文字列辞書圧縮の検討と評価

神田　峻介† 　森田　和宏† 　泓田　正雄†

† 徳島大学大学院先端技術科学教育部　〒 770–8506 徳島県徳島市南常三島町 2–1
E-mail: †shnsk.knd@gmail.com, ††{kam,fuketa}@is.tokushima-u.ac.jp

あらまし　文字列集合を保管するためのデータ構造である文字列辞書に関して，近年，多くの用途でコンパクト性が求められるという実例が報告されている．また，その背景に応じて，Trie や Front-Coding などの辞書を実現するための優れた技法に，文法圧縮などの強力なデータ圧縮を組み合わせた圧縮文字列辞書が提案されている．本稿では，既存の圧縮文字列辞書の改良を目的とし，文字列辞書の圧縮に文字列辞書を用いるという方策に基づいた辞書構造を提案する．実データを用いた実験より，提案による文字列辞書は Re-Pair により圧縮した辞書と比べ，メモリ効率や検索・復元速度のトレードオフに関して同等の性能を示しつつ，短い時間で構築できることを示した．

キーワード　圧縮文字列辞書，Trie，Front-Coding，圧縮データ構造

1. はじめに

文字列辞書とは，文字列の集合を保管するためのデータ構造であり，各文字列に対し固有の ID を割り当てる．すなわち，文字列の入力に対し，その ID を報告する Lookup と，ID の入力に対し，その文字列を報告する Access の 2 つの操作を提供するデータ構造である．自然言語処理や情報検索，セマンティック・ウェブ，バイオインフォマティクス，地理情報システムなどの数多くの用途において用いられている．一方で，近年では大規模データに対し辞書本体のサイズが問題となる事例が複数報告されており [1]，メモリ効率の良い辞書構造，いわゆる圧縮文字列辞書の提案が多くなされている [1–5]．

文字列辞書の圧縮に関して，テキストデータに対する強力な圧縮手法を適用する方策が，近年の研究では多く見られる．例えば，Martínez-Prieto ら [1] は，文字列辞書を実現するための様々な技法に既存のデータ圧縮を組み合わせることにより，メモリ効率に優れた辞書を複数個提案している．中でも，Front-Coding [6] をベースとした辞書の性能が高く，部分的に現れる文字列に対しデータ圧縮を適用することで高いメモリ効率を達成している．Grossi と Ottaviano [2] は，Trie [7] をベースに Path Decomposion [8] と呼ばれる技法を用いることで，キャッシュ効率に優れた辞書を実現している．この辞書構造においても，既存のデータ圧縮と組み合わせることで高いメモリ効率を実現している．

このように，テキストデータに対する既存の圧縮手法を用いた文字列辞書が複数提案されている．中でも，Trie や Front-Coding などをベースとして構築した辞書内に部分的に現れる文字列に，Re-Pair [9] と呼ばれる文法圧縮を適用することにより得られる圧縮文字列辞書は，汎用的に高いメモリ効率を達成し，Lookup/Access の実行時間に関しても優れている．一方，Re-Pair による圧縮は入力文字列長に対し線形時間で動作するものの，実際には大幅な時間と作業領域を要し，Re-Pair を用いた辞書の構築コストは非常に大きいという問題も同時に存在する．改善のために，圧縮コストの面での Re-Pair の改良，もしくは圧縮コストの低い異なるデータ圧縮の利用などが考えられるが，それではオリジナルの Re-Pair ほどの圧縮率が得られないというジレンマに陥る．この問題は，圧縮文字列辞書の更なる実用化に向けた改善すべき点の 1 つといえる．

本研究では，文字列辞書の圧縮に文字列辞書を用いるという方策に基づき，こうした問題の改善，及び既存の圧縮文字列辞書の性能向上を目指す．この方策に関して動機となったのが，矢田 [5] により提案された辞書構造である．この辞書では Trie をベースとし，部分的に現れる文字列を再帰的に Trie 辞書により圧縮するという方法で高いメモリ効率を達成している．このように辞書の圧縮に辞書を用いるという方法は，既存の Re-Pair などにより圧縮される辞書に対しても適用でき，新たな圧縮の手段となり得る．そこで本稿では，この方策に基づいた辞書構造をいくつか提案し，実験により評価を与える．

2. 準　　備

本節では，辞書を実装する上で必要となるデータ構造を簡単に紹介する．はじめに，基本的な定義を以下に与える．文字が取り得る値の集合 Σ をアルファベットといい，そのサイズを $\sigma = |\Sigma|$ と表す．対数の底は 2 で統一する．

2.1 Rank/Select 辞書

ビット列 $B[1...n]$ に対し，Rank/Select と呼ばれる基本的な 2 つの操作を導入する．

- $\mathrm{Rank}_b(B, i) : B[1...i]$ 中の $b \in \{0, 1\}$ の数を返す．
- $\mathrm{Select}_b(B, i) : i$ 番目の $b \in \{0, 1\}$ が出現する位置を返す．

これらの操作は，$o(n)$ ビットの補助データ構造を加えることによって，定数時間で実行できる [10]．

2.2 LOUDS 表現

LOUDS（Level-Ordered Unary Degree Sequence）[10] は順序木を表現するための簡潔データ構造，いわゆる簡潔木の 1 種である．LOUDS では，子を d 個持つ節点を d 個の ‘1’ と 1 個の ‘0’ の連結により表し，これらを幅優先に連結することにより得られるビット列により木を表現する．このとき，根の親のように振る舞うスーパールートとして，ビット列の先頭に “10”

を付与する．節点の移動は，ビット列上の Rank/Select を介して実現されるため，仮に木の節点数を n としたとき，そのメモリ使用量は $2n + o(n)$ ビットとなる．

2.3 DFUDS 表現

DFUDS（Depth-First Unary Degree Sequence）[11] は，LOUDS と同じく簡潔木の 1 種であり，括弧列を用いて順序木を表現する．DFUDS では，子を d 個持つ節点を d 個の '(' と 1 個の ')' の連結により表す．そして，これらを深さ優先順に連結し，先頭に '(' を追加した括弧列により木を表現する．DFUDS における節点の移動に必要な括弧列上の操作は，$o(n)$ ビットの補助データ構造により定数時間で与えられる [10, 12]．節点数 n に対し，そのメモリ使用量は $2n + o(n)$ ビットである．

2.4 Elias-Fano 表現

Elias-Fano 表現 [13, 14] とは，最大値が n である m 個の整数値から構成される広義単調増加列に対し，定数時間でのアクセスを保証しつつ，$2m + m \log \frac{n}{m} + o(m)$ ビットで表現するための符号である．Elias-Fano 表現の効率的な実装はいくつかあるが，公開されている実装としては Succinct [15] が有名である．

3. 文字列辞書

文字列辞書とは，文字列の集合 $S \subset \Sigma^*$ を保管し，各文字列に対し固有の ID として $[1, |S|]$ 内の整数値を与えるためのデータ構造である．このとき，以下の 2 つの操作を提供する．

- Lookup(s) : $s \in S$ であれば，その ID を返す．
- Access(i) : ID i に対応する文字列を復元する．

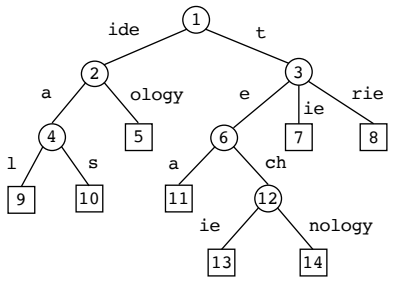
また，ある文字列の系列 $\langle s_1, s_2, ..., s_n \rangle$ を，文字列辞書を用いて整数値の系列 $\langle i_1, i_2, ..., i_n \rangle$ に変換する操作を辞書符号化と呼ぶ．基本的に，文字列よりも整数値の方が格納に必要な領域は小さく，整数値列と文字列辞書のメモリ使用量を足し合わせた結果が，元の文字列系列のメモリ使用量よりも小さい場合，辞書符号化による圧縮が達成される．

本研究では，Trie や Front-Coding 内に部分的に現れる文字列に対して辞書符号化を適用することで，既存の辞書構造の改良を図る．そこで本節では，Compact Trie（C-Trie），Path-Decomposed Trie（PD-Trie），Front-Coding の 3 つの辞書構造，および既存の実装方法について説明する．加えて，辞書符号化を適用するためのデータ構造を提案する．

3.1 Compact Trie（C-Trie）

Trie とは，文字列の共通接頭辞を併合し，枝に文字を付随することで構築されるラベル付き順序木である．登録文字列は，根から葉への経路上の枝ラベルを連結することにより復元される．そして，C-Trie とは，Trie における分岐を持たない節点を削除し，枝に文字列のラベルを与えることにより構築される木構造である．図 1a に，文字列集合 $S^{ex} = \{$"`ideal`", "`ideas`", "`ideology`", "`tea`", "`techie`", "`technology`", "`tie`", "`trie`"$\}$ に対する C-Trie の例を示す．

C-Trie を用いた圧縮文字列辞書として知られるのが，本研究の動機でもある，矢田によって提案された MARISA（Matching Algorithm with Recursively Implemented StorAge）[5] である．MARISA とは，C-Trie のラベル圧縮に C-Trie を用いるという処理を再帰的に繰り返すことで構築されるデータ構造である．ここでは MARISA を参考に，C-Trie の圧縮に辞書符号化を用いる方法を説明する．



(a) C-Trie．各文字列の終端に位置する節点が四角で描かれいる

	12	3 4 5	6 7	8	9 0 1 2	3	4
L	|ide	t|a	ology|e	ie|rie	l|s|a	ch|ie	nology

	12345678901234
L'	BtaEeCFlsaACD
F	01001011100111

Dictionary encoding

A	ch
B	ide
C	ie
D	nology
E	ology
F	rie

	12345678901234567890123456789
B	10110110111011001100000011000
T	00001011111111

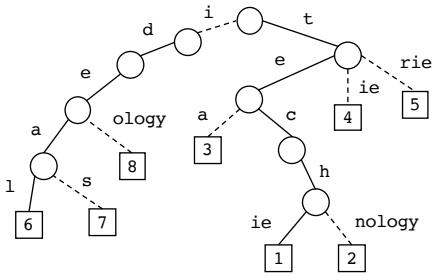
(b) C-Trie の表現

図 1: S^{ex} に対する C-Trie，及び表現の例

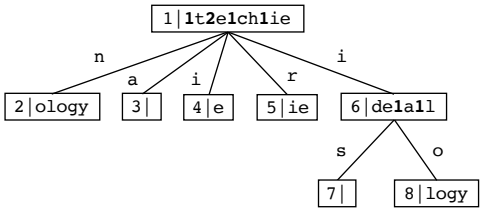
C-Trie の圧縮に辞書符号化を用いる場合，その対象となるのは枝ラベルである．MARISA では枝ラベルに関して，長さ 1 のラベルと長さ 2 以上のラベルを区別し，後者を辞書符号化により整数値に置き換える．仮に節点 v に向かう枝のラベルを格納した配列を $L[v]$ とすると，L は配列 L' とビット列 F を用いて以下のように置き換えられる．

- $|L[v]| = 1$ の場合，$L'[v] \leftarrow L[v]$ かつ $F[v] \leftarrow 0$.
- $|L[v]| \geqq 2$ の場合，文字列 $L[v]$ を辞書符号化し得られる ID を i_v とすると，$L'[v] \leftarrow i_v$ かつ $F[v] \leftarrow 1$.

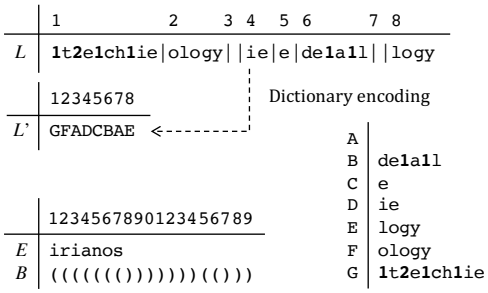
図 1a の C-Trie を，MARISA と同様の方法で辞書符号化により表現した例を図 1b に示す．MARISA では，木の表現に LOUDS を用いているため，L は枝ラベルを幅優先順に並べた結果であり，辞書符号化を用いて L の代わりに L' と F が得られる．B は木に対する LOUDS 表現であり，T は節点 v が文字列の終端に位置すれば $T[v] = 1$ となるようなビット列である．T は文字列の終端を表すだけでなく，Rank/Select により節点 ID から $[1, |S|]$ 内の文字列 ID への写像を与える役割も果たす．このとき，Lookup/Access は以下のように実行される．Lookup(s) は，s を構成する文字により根から葉 v へ遷移し，$\mathrm{Rank}_1(T, v)$ でその ID を返す．Access(i) は，$\mathrm{Select}_1(T, i)$ により復元したい文字列の終端に位置する節点 ID を取得し，そこから根までの枝ラベルを連結することで登録文字列を復元する．これら操作において枝ラベルを参照するとき，$F[v] = 1$ であれば $L'[v]$ に格納された ID を用いて辞書から枝ラベルを随時復元する．



(a) Trie．ただし，各葉に向う経路上の内部節点は省略している．



(b) PD-Trie．各節点の左部は節点 ID，右部は節点ラベルを表している



(c) PD-Trie の表現

図 2: S^{ex} に対する Trie，PD-Trie，及びその表現の例

3.2 Path-Decomposed Trie（PD-Trie）

PD-Trie とは，Trie を節点から葉までの経路に分解する操作（Path Decomposion）を再帰的に繰り返すことにより構築される木構造であり，各節点が各経路に対応する．ここでは，まず初めに，図 2 に示す例を用いて Grossi と Ottaviano [2] による文字列辞書に基づく PD-Trie のデータ構造，及び従来の実装方法を説明する．次に，辞書符号化を圧縮に適用するためのデータ構造について提案する．

図 2b に示す木構造は，図 2a の Trie を実線で繋がれた経路ごとに分解し構築された PD-Trie であり，各節点には Trie の各経路を表す節点ラベル，各枝には分岐文字が付随している．節点ラベルに含まれる $\mathbf{1}, \mathbf{2}, ...$ という文字は Σ に含まれない特殊文字であり，分岐がある箇所と分岐数を示している．こうした PD-Trie は以下の手順で構築される．まず，Trie の根から任意の葉までの経路 π を選択する．次に，経路 π 上の枝の文字と，各節点において部分 Trie が垂れ下がっている場合には，その部分 Trie の数を示す特殊文字 $\in \{\mathbf{1}, \mathbf{2}, ...\}$ とを連結することで得られる文字列を，PD-Trie の根 v_π の節点ラベルとする．根 v_π の子は，経路 π から垂れ下がる部分 Trie に対する PDT の根として再帰的に定義する．このとき，子への分岐文字には各部分 Trie へ向かう枝のラベルを与える．

Trie の分解に関して，どの子を辿り経路を選択するかは任意であるが，この例では Heavy Path と呼ばれる経路を選択している．Heavy Path とは，子の選択において，その子を根とする部分木で最も多くの葉を持つ子を選択し続けることにより得られる経路である．Heavy Path を選択し Trie を分解する操作は，一般に Centroid Path Decomposition（CPD）と呼ばれ，CPD により得られる PD-Trie の高さは高々$O(\log |S|)$ となることが知られている．すなわち，CPD は節点間のランダムアクセスの回数を抑制し，キャッシュ効率のよい Trie 辞書を構築できる．本稿では CPD による PD-Trie の構成を前提とする．

PD-Trie の表現に関して，各節点 v は節点ラベルを格納した L_v，節点 v から出る分岐文字を逆順に格納した E_v，節点 v の DFUDS 表現 B_v により表現される．節点番号は深さ優先順に与えられ，L_v, E_v, B_v を節点番号の順に連結することにより得られる配列 L, E, B により PD-Trie は表現される（図 2c）．PD-Trie については表現方法の紹介に留めるため，Lookup/Access の実行方法については原著論文 [2] を参考されたい．

3.2.1 従来の実装

L は，$\Sigma' = \Sigma \cup \{\mathbf{1}, \mathbf{2}, ..., \sigma - \mathbf{1}\}$ からなる文字を要素とし，実際 $\Sigma = [0, 256)$ なことから $\Sigma' = [0, 511)$ である．この L の実装に関して，文献 [2] では 2 つの手法を提案している．

1 つは，L を Vbyte [16] によりバイト列に置き換える単純な手法である．このとき，Σ' の各値を L において出現頻度が大きい順に 0 から割り振ることで，生成されるバイト列を小さくすることができる(注1)．もう 1 つは，L を Re-Pair [9] により圧縮する手法である．ただし，オリジナルの Re-Pair の実装ではなく，Approximate Re-Pair [17] に修正を加えた実装を適用している．この実装では，ある程度の圧縮率を犠牲に，圧縮コストと復元コストを一定に抑えることができる．また，どちらの手法に対しても，L における節点ラベルの境界は Elias-Fano 表現を用いて記憶しておく．

3.2.2 辞書符号化を用いた圧縮

辞書符号化を用いた PD-Trie の圧縮では，L に含まれる各節点ラベル（空文字列も含む）に対し辞書符号化を適用する．図 2c の L' は，L の各節点ラベルを辞書符号化し，ID に置き換えた結果である．このとき，節点ラベルは Vbyte によりバイト列に変換し辞書符号化を適用する．L' は単なる整数値列になるため，単純な固定長配列として管理でき，各節点ラベルの境界を記憶する必要はなくなる．

3.3 Front-Coding

Front-Coding とは，辞書式順に整列された文字列集合に対し，各文字列をその直前の文字列との最長共通接頭辞長と，残った接尾辞のペアに符号化する技法である．自然言語や URL など，実際のコーパスには似たような接頭辞が多く現れるという特徴を利用し圧縮する．辞書として各文字列にランダムアクセスしたい場合には，文字列集合を一定個数の文字列からなるバ

(注1)：文献 [2] の実装では，英文に対し大半の文字が $[0, 128)$ 内の値を取ることを前提としているため，このような工夫は施していない．

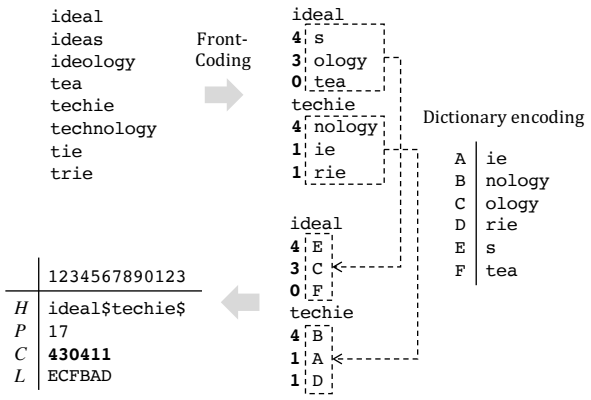


図 3: S^{ex} に対する Front-Coding の例

ケットに分割し，バケットの最初の文字列（ヘッダー）を符号化せずにそのまま格納しておく．このとき，文字列の ID は辞書式順に割り振られる．図 3 の上部は，バケットのサイズを 4 として S^{ex} に Front-Coding を適用した場合の例を示している．Lookup は，ヘッダーに対する二分探索により，その文字列が含まれているバケットを特定し，その中を逐次的に調べることで実行される．Access は，受け取った ID からバケットと，そのバケット内でのオフセット算出し，ヘッダーから順に復元していくことで実行される．

3.3.1 従来の実装

Front-Coding を用いた辞書の単純な実装としては，[1] で提案された Plain Front-Coding（PFC）があげられる．PFC は，終端文字が付与された文字列と，Vbyte によりバイト列に置き換えられた共通接頭辞長を，メモリ上に連続して配置し，各ヘッダーの先頭位置を記憶しておくデータ構造である．また [1] では，二分探索の対象となるヘッダー以外のバイト列に対し Re-Pair を適用することで，PFC を圧縮している (注2)．提案者の実装では，Navarro により公開されているオリジナルに近い Re-Pair 実装（`https://www.dcc.uchile.cl/~gnavarro/software/`）を採用している．

3.3.2 辞書符号化を用いた圧縮

辞書符号化を用いた Front-Coding の圧縮では，PFC の圧縮と同じくヘッダー以外の文字列を辞書符号化により ID に置き換える．図 3 の下部にその例を示す．本提案における Front-Coding 辞書は，ヘッダー文字列を格納するための配列 H と，ヘッダーの各先頭位置を示すポインタの配列 P，共通接頭辞長を格納する配列 C，辞書符号化により得られた ID を格納する配列 L により構成される．

4. 文字列辞書圧縮のための文字列辞書

表記の混同を避けるため，文字列辞書の圧縮を目的として用いられる文字列辞書を，本稿では補助文字列辞書と呼ぶ．補助文字列辞書は，節 3. で紹介したように，Trie や Front-Coding 内に残る文字列を整数値に置き換え，それら文字列を別領域で保管するために用いられる．ただし，本用途では補助文字列辞書が文字列に割り当てる ID について，その値域を制限しない．Lookup/Access において文字列を復元する用途で用いられる補助文字列辞書について，提供すべき操作を以下に定義する．

- Restore(i)：ID i に対応する文字列を復元する．
- Compare(i, q)：Restore(i) とクエリ文字列 q を比較する．

Restore は Access と同様の操作だが，表記の混同を避けるために改めて定義した．Compare は Lookup において呼び出される操作である．Restore が実行できれば Compare も実行できるが，Restore は復元する文字列を必ず最後まで読み込むのに対し，Compare はクエリとミスマッチしてる場合，最後まで読み込む必要はない．

本節では，以下のことを考慮し補助文字列辞書を提案する．

- 一回の Lookup または Access において，Compare または Restore は複数回呼び出される．そのため，文字列辞書以上に実行時間が重要視される．
- Trie や Front-Coding によって，接頭辞が共有され構築された辞書内に残る部分文字列を保管の対象とする．これらの文字列には，似たような接尾辞が多く現れることが予想される．

4.1 単純な接尾辞併合

補助文字列辞書を実現する最も単純なデータ構造は，終端文字を付与した文字列をメモリ上に連続して配置し，各文字列の先頭位置を ID として得る手法である．このとき，ある文字列が他の文字列の接尾辞に含まれる場合，それらを併合することができる．こうして構築される配列は一般に TAIL と呼ばれる [4, 5]．後に示す Reverse Trie や Back-Coding と比べると，併合できる文字列の割合は少ないが，単純な文字列として高速に参照できる．

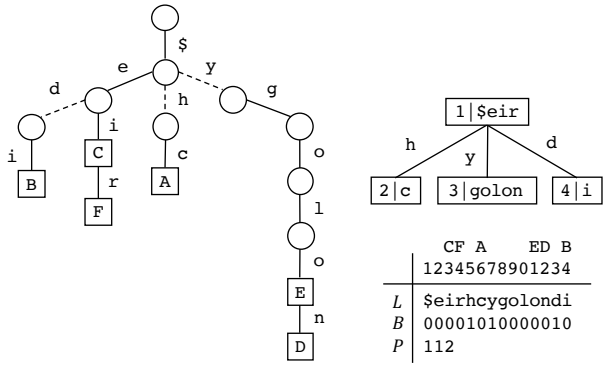
4.2 Reverse Trie

本稿では，文字列を後方から併合することによって構築される Trie を Reverse Trie と呼ぶ．Reverse Trie では，文字列の終端が根に対応し，葉から根の方向に辿ることで登録文字列が復元できる．すなわち，文字列の始点となる節点 ID を保持しておくことで，Restore や Compare が実行できる．ここでは，Compact Trie を用いた Reverse Trie の実装である Reverse Compact Trie（RC-Trie）と，PD-Trie を用いた実装である Reverse Path-Decomposed Trie（RPD-Trie）の 2 種類のデータ構造ついて説明する．RC-Trie は，MARISA の補助文字列辞書として用いられている既存の実装手法である．RPD-Trie は，本稿で新たに提案するデータ構造であり，RC-Trie よりも更に高速な復元を提供する補助文字列辞書の実装である．

4.2.1 Reverse Compact Trie（RC-Trie）

RC-Trie は，節 3.1 で説明した方法と同様の実装で，Reverse Trie による辞書を実現する．ただし，各節点 ID を文字列の ID として与え，そこから根に向かって辿ることで文字列を復元するため，各文字列の終端を表すビット列 T は必要としない．RC-Trie もまた，辞書符号化により圧縮されるデータ構造である．MARISA では予め決めた数の分だけ RC-Trie を再帰的に構築し，最後に残る文字列は TAIL により保管する．

(注2)：文献 [1] では，他にも Hu-Tucker 符号 [7] を用いたヘッダーの圧縮も提案しているが，本稿ではヘッダー以外の文字列に対し，Re-Pair を適用した場合と辞書符号化を適用した場合との比較を目的としているため，ヘッダーの圧縮については評価していない．



(a) Reverse Trie (b) RPD-Trie

図 4: Reverse Trie と RPD-Trie の例

4.2.2 Reverse Path-Decomposed Trie（RPD-Trie）

RC-Trie よりも高速な Reverse Trie の実装を考える場合，キャッシュ効率を高める Path Decomposion の利用が有力な選択肢となるが，文献 [2] の実装では，各節点ラベルを先頭から読み込んでいく必要があるため，Compare におけるミスマッチを即座に検出できない．そこで，補助文字列辞書のための PD-Trie の実装，RPD-Trie を以下に提案する．

補助文字列辞書のための Reverse Trie では，子を特定する操作を必要としないため，RPD-Trie の実装は文献 [2] の実装と比べ単純である．図 1b で用いられている補助文字列辞書を Reverse Trie により実現した例を図 4a に，それに対する RPD-Trie の例を図 4b に示す．説明の都合上，Reverse Trie には終端文字 '$' によりスーパールートを付与している．図 4b の RPD-Trie は，Reverse Trie に CPD を適用し，幅優先順に節点 ID を割り当てることにより構築されている．子を特定する必要がないため，分岐情報を示す特殊文字 $\mathbf{1}, \mathbf{2}, ...$ は含まれていない．

RPD-Trie の表現には，3 つの配列 L, B, P を用いる．L は，RPD-Trie の各節点に対して，その節点へ向かう分岐文字（スーパールート以外）と節点ラベルを連結することにより得られる文字列を，節点 ID 順に連結することにより得られる文字列である．B は，$L[i]$ が分岐文字なら $B[i] = 1$，節点ラベルなら $B[i] = 0$ であるようなビット列である．P は，各枝が節点ラベルのどの位置から分岐してきたかを示す整数値列であり，L 上の添字を節点 ID 順に格納する．RPD-Trie は，各文字列の始点に位置する L 上の添字を文字列の ID として発行する．このとき，RPD-Trie の Restore(i) は以下の手順で実行される．

（1） str を "" で初期化する．

（2） $L[i] =$ '$' なら str を返し終了する．

（3） str の末尾に $L[i]$ を加える．

（4） $B[i] = 1$ なら $i \leftarrow P[\mathrm{Rank}_1(B, i)]$，$B[i] = 0$ なら i をデクリメントすることで参照位置を更新し，2 へ戻る．

メモリ使用量に関して，Reverse Trie の節点数を n とした場合，L は $n \log \sigma$ ビット，B は $n + o(n)$ ビット要する．P は整数値列だが，幅優先に節点番号を割り当てることで広義単調増加列となるので，Elias-Fano 表現が適用できる．仮に $|P| = m$ としたとき，P は $2m + m \log \frac{n}{m} + o(m)$ ビット要する．簡潔木により Reverse Trie を表現する場合，そのメモリ使用量は $2n + n \log \sigma + o(n)$ ビットであり，おおよそ $2m + m \log \frac{n}{m} < n$ のとき，RPD-Trie は簡潔木表現より小さくなる．ここで，m は Reverse Trie の葉の数から 1 を引いた値であり，図 4 の例にも見られるように，n と比べると実際にはかなり小さい値となる．そのため，RPD-Trie は簡潔木と同等，もしくはそれ以上のメモリ効率が期待でき，かつキャッシュ効率の良い復元を提供できる．

4.3 Back-Coding

Front-Coding の技法を，接尾辞に対し適用することで補助文字列辞書を実現する．本稿では，この技法を Back-Coding と呼ぶ．Back-Coding でも Front-Coding と同様に，文字列集合をバケットに分割しヘッダーから順に復元する．補助文字列辞書では高速性が強く求められるため，本研究では Back-Coding を単純に PFC により実装する．

また，更に高速性を追求するため，直前との文字列の差異ではなく，各バケットごとにヘッダーとの差異を用いて置き換える方法 [18] も補助文字列辞書に適用する．本稿では，この技法を Fast Back-Coding（FBC）と呼ぶ．FBC による辞書では，途中の文字列を復元する必要がなく，メモリをコピーする回数が高々2 回で済む．ただし，整数値に置き換えられる文字列が少なくなる分，メモリ効率とのトレードオフとなる．

5. 実験による評価

本節では，節 3. で紹介した文字列辞書に対し，節 4. で紹介した補助文字列辞書による圧縮を適用した場合の性能に関して評価を与える．

5.1 実験設定

実験に用いた計算機の構成は，Intel Core i7 4.0 GHz CPU, 16 GB RAM (L2 cache 256 KB, L3 cache 8 MB) であり，OS は OS X 10.12 である．辞書の実装言語は C++で，最適化オプションとして-O3 を指定し，Apple LLVM version 8.0.0 (clang-800.0.42.1) を用いてコンパイルした．実行時間の計測には，`std::chrono::duration_cast` を用いた．

5.1.1 データ構造

C-Trie，PD-Trie，Front-Coding をベースとする文字列辞書に対し，TAIL，RC-Trie，RPD-Trie，Back-Coding，FBC の 5 種類の補助文字列辞書による圧縮をそれぞれ適用した．RC-Trie に関して，1 つの RC-Trie と TAIL により構成される辞書と，2 つの RC-Trie と TAIL により構成される辞書を評価した（それぞれ RCT1，RCT2 と表記）．Back-Coding と FBC のバケットサイズに関しては，4，8，16 の 3 種類を評価した（それぞれ BC4，BC8，BC16，FBC4，FBC8，FBC16 と表記）．位置付けとしては，TAIL と RC-Trie が既存の補助文字列辞書，RPD-Trie，Back-Coding，FBC が提案された補助文字列辞書となる．

PD-Trie，Front-Coding に関しては，補助文字列辞書を用いない従来の実装手法も比較に用いた．PD-Trie では，節点ラベルの表現に Vbyte と Re-Pair を用いる 2 つの実装を評価した

表 1: コーパスに関する情報

	サイズ	文字列数	平均長	σ
S_W	227.2	11,519,354	20.7	199
S_I	612.9	7,414,866	86.7	98
S_U	2,723.3	39,459,925	72.4	103

（それぞれ Plain，Re-Pair と表記）．Front-Coding では，PFC とそれに Re-Pair を適用した 2 つの実装を評価した（同じく Plain，Re-Pair と表記）．Re-Pair の実装は亜種含めいくつか存在するが，本実験では節 3. で紹介したように，それぞれの提案者が辞書の圧縮に適用している Re-Pair の実装を用いて，ベースラインとした．また，Front-Coding 辞書のバケットサイズは，過去の実験を参考に 8 で統一した．

5.1.2 コーパス

辞書の構築には，以下の 3 つのコーパスを利用した．

- S_W：英語版 Wikipedia の見出し集合（`https://dumps.wikimedia.org/enwiki/`）
- S_I：インドシナ諸国のドメイン上でクロールし得られた URL 集合（`http://data.law.di.unimi.it/webdata/indochina-2004/indochina-2004.urls.gz`）
- S_U：`.uk` ドメイン上でクロールし得られた URL 集合（`http://data.law.di.unimi.it/webdata/uk-2005/uk-2005.urls.gz`）

表 1 にコーパスの基本的なデータを示す.「サイズ」は生データのサイズ，すなわち文字列の合計長を示しており，単位は MiB である.「文字列数」は，コーパス内の重複を除いた文字列の数,「平均長」はそれらの平均文字数を示している.「σ」は，コーパスにおいて用いられている文字の種類の数を示している．

また，これらコーパスから C-Trie，PD-Trie，Front-Coding によって辞書を構築した場合に，辞書符号化の対象として現れる文字列に関する情報を表 2 に示す.「サイズ」は現れた文字列の合計長を MiB で示しており,「文字列数-前」はその文字列数を示している.「文字列数-後」はそこから重複を除いた場合の文字列数を示しており,「平均長」はその平均文字数を示している．表 2 から分かるように，重複を除くだけでも文字列数を 18–33%に削減することができる．

5.2 結果と評価

表 3 に実験結果を示す．各列の項目について,「構築時間」は辞書の構築に要した時間で単位は秒,「圧縮率」はコーパスの生データのサイズに対する圧縮率で単位は%,「Lookup」と「Access」はそれぞれ 1 回当たりの実行時間で単位はマイクロ秒である．Lookup/Access の実行時間を計測するために，コーパスからランダムに抽出した 100 万個の文字列，及びその文字列に対応する 100 万個の ID を用いた．それぞれ，10 回の試行から得られた結果の平均である．

5.2.1 C-Trie の結果（表 3a）に対する評価

構築時間に関しては，いずれも TAIL が高速であり，PRDT が低速であった．しかし，その差は高々1.4 倍であり大差ではない．圧縮率に関しては，RPT2 が総じて高く，Lookup/Access の実行時間に関しては，TAIL が総じて優れていた．新提案の補助文字列辞書に関して，S_W のように平均長が比較的短い場合には RPDT が，S_I や S_U のように長い場合には FBC4 が，全体のバランスとして優れていた．しかし，どちらの性能も TAIL と RPT1 の間に位置し，C-Trie に適用する上で特別どの補助文字列辞書が優れているといった結果は得られなかった．

表 2: 辞書内に現れる文字列に関する情報

(a) S_W

	サイズ	文字列数-前	文字列数-後	平均長
C-Trie	90.9	11,580,413	3,027,555	12.8
PD-Trie	90.3	11,519,354	3,762,732	14.4
Front-Coding	83.0	10,079,434	2,921,168	13.5

(b) S_I

	サイズ	文字列数-前	文字列数-後	平均長
C-Trie	135.8	6,472,460	1,251,149	39.1
PD-Trie	134.6	7,414,866	1,300,192	44.7
Front-Coding	121.6	6,488,007	1,204,600	42.5

(c) S_U

	サイズ	文字列数-前	文字列数-後	平均長
C-Trie	663.3	41,001,910	11,396,164	32.5
PD-Trie	655.4	39,459,925	11,958,074	35.2
Front-Coding	591.7	34,527,434	10,756,301	34.2

5.2.2 PD-Trie の結果（表 3b）に対する評価

構築時間に関しては，いずれも Plain が高速であった．補助文字列辞書を用いた場合の構築時間は，いずれも近い値を取り，Plain と比べて最大で 3.4 倍低速であった．一方，Re-Pair を用いた実装は，構築コストを一定に抑えるような実装をしているものの，Plain と比べて最大で 15 倍も低速であった．このことから，補助文字列辞書を用いた場合の構築コストは，常に低く安定している．

補助文字列辞書において，圧縮率に関しては RPT2，BC16 が優れた結果であるものの，Lookup/Access の実行時間が非常に低速である．圧縮率と実行時間のバランスで優れているのは RPDT と FBC4，FBC8 であり，Re-Pair と比べて，S_W における FBC を除いて圧縮率と実行時間で近い値を示している．これに加えて構築時間が短いことが，従来の圧縮に対する RPDT と FBC4，FBC8 の利点といえる．

また，注目すべき点として，Plain よりも TAIL の方が Lookup/Access の実行時間で上回ったという点があげられる．この要因としては，辞書符号化により Elias-Fano 表現による節点ラベルの境界の復元が必要なくなったことがあげられ，辞書符号化による圧縮が効率的に働いた 1 つの例といえる．また，TAIL による圧縮率も，S_I や S_U では Re-Pair と比べ 2–3%しか違わない．復元速度を重視した他のデータ圧縮を用いたとしても，単純な文字列参照である TAIL より高速になるとは考えづらく，Lookup/Access の実行時間を重視した実装において，TAIL は優れた手段となる．

表 3: 実験結果

(a) C-Trie

	S_W				S_I				S_U			
	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access
TAIL	**8.5**	34.0	**1.11**	**1.13**	**8.3**	11.7	**1.47**	**1.87**	**48.1**	19.0	**2.46**	**2.65**
RCT1	10.1	26.1	1.61	1.59	9.1	8.1	2.87	3.20	57.0	13.4	4.47	4.51
RCT2	10.4	**24.9**	1.68	1.67	9.3	**7.2**	3.44	3.82	59.5	**12.1**	5.22	5.22
RPDT	11.8	**27.6**	**1.39**	1.42	9.6	9.5	2.25	2.73	65.2	15.0	3.53	3.70
BC4	11.2	30.8	2.05	1.37	**9.2**	9.6	4.26	2.55	61.4	15.5	5.24	3.59
BC8	11.1	29.2	2.23	1.45	**9.2**	9.1	5.67	3.56	61.3	14.6	5.87	3.95
BC16	11.1	28.3	2.59	1.63	**9.2**	**8.8**	7.45	4.66	**61.1**	**14.1**	7.07	4.64
FBC4	11.1	31.6	1.46	**1.32**	**9.2**	9.9	**2.07**	**2.34**	61.7	15.9	**3.28**	**3.19**
FBC8	11.1	30.9	1.53	1.36	9.3	9.8	2.37	2.61	61.2	15.5	3.47	3.29
FBC16	**11.0**	30.9	1.65	1.43	**9.2**	9.9	2.86	2.84	61.4	15.5	3.83	3.50

(b) PD-Trie

	S_W				S_I				S_U			
	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access
Plain	**2.1**	49.6	**1.25**	**1.42**	**2.5**	24.5	**1.39**	**1.78**	**12.8**	27.1	**1.89**	**2.26**
Re-Pair	29.0	**31.6**	1.31	1.48	20.3	**11.8**	1.63	2.00	185.6	**17.5**	2.06	2.42
TAIL	**4.7**	41.7	**1.13**	**1.26**	**4.0**	13.5	**1.23**	**1.56**	**24.1**	20.7	**1.67**	**1.96**
RCT1	6.7	31.3	2.51	2.57	4.8	9.1	2.91	3.16	32.3	14.8	4.14	4.28
RCT2	7.2	**29.3**	3.66	3.63	5.0	**8.2**	4.55	4.75	35.5	**13.4**	6.55	6.57
RPDT	7.2	**32.4**	**1.51**	**1.60**	4.8	10.7	**1.68**	**1.92**	33.8	16.4	2.48	**2.67**
BC4	**6.5**	36.0	3.38	3.66	**4.5**	10.9	5.24	5.75	31.0	16.9	5.71	6.24
BC8	**6.5**	33.8	3.71	3.96	**4.5**	10.3	6.09	6.64	31.1	15.9	6.58	7.12
BC16	**6.5**	32.7	4.48	4.77	**4.5**	**10.0**	7.73	8.27	**30.9**	**15.4**	8.28	8.79
FBC4	**6.5**	37.0	1.61	1.74	**4.5**	11.3	1.78	2.08	31.1	17.3	**2.42**	2.69
FBC8	**6.5**	35.9	1.73	1.85	**4.5**	11.2	2.00	2.30	31.2	16.9	2.64	2.92
FBC16	**6.5**	35.9	1.97	2.12	**4.5**	11.3	2.40	2.73	31.2	16.9	3.03	3.33

(c) Front-Coding

	S_W				S_I				S_U			
	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access	構築時間	圧縮率	Lookup	Access
Plain	**0.8**	59.6	**1.11**	**0.37**	**0.8**	34.9	**1.31**	**0.33**	**3.8**	37.3	**1.60**	**0.37**
Re-Pair	470.9	**36.5**	2.09	1.31	1363.8	**18.2**	2.67	1.66	5861.6	**22.3**	3.29	1.91
TAIL	**3.7**	45.3	**1.39**	**0.69**	**2.7**	25.0	**1.72**	**0.78**	**17.7**	31.4	**2.26**	**0.88**
RCT1	5.2	37.4	2.42	1.50	3.6	21.1	2.30	1.22	26.7	25.7	3.84	2.13
RCT2	5.6	**36.0**	2.63	1.69	3.8	**20.1**	2.76	1.62	29.2	**24.4**	4.66	2.85
RPDT	6.2	**38.7**	1.95	1.10	3.9	22.3	2.13	1.08	30.6	27.1	3.31	1.73
BC4	5.7	41.9	1.82	1.03	**3.6**	22.5	2.27	1.22	27.3	27.7	3.00	1.54
BC8	**5.6**	40.2	2.01	1.19	**3.6**	22.0	2.60	1.54	**27.2**	26.8	3.40	1.87
BC16	**5.6**	39.4	2.36	1.56	**3.6**	**21.7**	3.37	2.30	27.3	**26.4**	4.31	2.72
FBC4	5.7	42.7	**1.73**	**0.95**	**3.6**	22.9	**1.98**	**0.98**	27.4	28.1	**2.64**	**1.22**
FBC8	5.7	42.0	1.78	1.01	**3.6**	22.7	2.04	1.04	27.4	27.7	2.72	1.30
FBC16	**5.6**	42.0	1.90	1.14	**3.6**	22.9	2.20	1.21	27.3	27.7	2.92	1.49

5. 2. 3 Front-Coding の結果（表 3c）に対する評価

構築時間，Lookup/Access の実行速度に関して，Plain が総じて最も優れているが，圧縮率に関しては大きく劣る結果となった．補助文字列辞書を用いた実装の構築時間に関して，Plain には及ばないものの，Re-Pair と比べ，S_W で最大 127 倍，S_I で最大 505 倍，S_U で最大 331 倍と圧倒的に高速であることがわかった．Front-Coding の Re-Pair では，オリジナルに近い実装をしているということに起因し，これほどにも差が広がった．一方で，圧縮率については Re-Pair が優れており，S_I と S_U では最もコンパクトに辞書を構築している．しかし，圧縮率に反して Lookup/Access の実行時間は比較的遅く，その差は TAIL と比べて 1.5–2.2 倍，FBC4 と比べて 1.2–1.7 倍と低

速であった．

最後に，RPDT と FBC4 を Re-Pair に対し比較すると，RPDT と FBC4 は圧縮率で 2–6%劣るものの，Lookup/Access の実行時間は 1.1–1.7 倍高速であり，構築時間は 76–379 倍と圧倒的に高速である．これより，補助文字列辞書は Re-Pair よりも現実的な圧縮手段といえる．

6. おわりに

本稿では，文字列辞書の圧縮に辞書符号化を用いる方策に基づき，既存の辞書構造に辞書符号化を適用する方法，及び辞書圧縮のための辞書構造を提案した．本実験では，PD-Trie，Front-Coding ともに，それぞれの提案者が適用した Re-Pair の実装をベースラインとして用いたが，構築コスト，圧縮率，復元速度など，どこに重点を置くかによって，Re-Pair の実装や適用するデータ圧縮法などには選択の余地がある．しかし，RPDT や FBC により圧縮した辞書は Re-Pair 圧縮による辞書と比べ，圧縮率，Lookup/Access の実行時間のトレードオフにおいて同等の性能を示しつつ，構築コストに関して大きく上回ったという結果から，辞書符号化による圧縮は有効な手段の 1 つとなり得る．ただし，本実験では構築における作業領域についての評価を与えられていないため，それは今後の課題となる．

また本稿では，文字列辞書に対する辞書符号化の適用を考察したが，部分的に多数の文字列が現れるデータ構造であれば，同様の圧縮が適用できる．とりわけ，PD-Trie への適用で Elias-Fano 表現が取り除けたように，文字列を整数値に置き換えることにより利点が見いだせるデータ構造であれば，辞書符号化による圧縮は効果的である．今後の予定としては，そうしたデータ構造の調査と適用実験があげられる．

文　献

[1] Miguel A Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016.

[2] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM Journal of Experimental Algorithmics*, 19(1):Article 1.8, 2014.

[3] Julian Arz and Johannes Fischer. LZ-compressed string dictionaries. In *DCC*, pages 322–331, 2014.

[4] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems*, Online First, 2016.

[5] 矢田 晋. Prefix/patricia trie の入れ子による辞書圧縮. In **言語処理学会**, 2011.

[6] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[7] Donald E Knuth. *The art of computer programming, 3: sorting and searching*. Addison Wesley, Redwood City, CA, USA, 2nd edition, 1998.

[8] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *PODS*, pages 181–190. ACM, 2008.

[9] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. the IEEE*, 88(11):1722–1732, 2000.

[10] Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE, 1989.

[11] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[12] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[13] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[14] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Memorandum 61, Computer Structures Group, MIT, Cambridge, MA, 1971.

[15] Roberto Grossi and Giuseppe Ottaviano. Design of practical succinct data structures for large data collections. In *SEA*, pages 5–17, 2013.

[16] Hugh E Williams and Justin Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

[17] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):16, 2010.

[18] Ingo Müller, Cornelius Ratsch, and Franz Faerber. Adaptive string dictionary compression in in-memory column-store database systems. In *EDBT*, pages 283–294, 2014.