Vol.2017-DBS-165 No.15
Vol.2017-IFAT-128 No.15
2017/9/18

Path Decompositionを用いたメモリ効率の良い動的キーワード辞書の実装法

神田 峻介[1,2,a)]　森田 和宏[1]　泓田 正雄[1]

概要：キーワード辞書とは文字列をキーとする連想配列であり，文字列集合を保管するためのデータ構造として古くから用いられている．一方で近年，このキーワード辞書を用いて大規模な文字列データを主記憶で管理するといった実例が数多く報告されており，メモリ効率の良い実装が求められている．現在のところ，静的用途に限定すれば数多くの実装が高いメモリ効率を達成している．しかし，それらに比べて既存の動的な実装は遥かに多くのメモリを消費する．そこで本稿では，Path Decomposition を用いたメモリ効率の良い動的キーワード辞書の実装法を提案する．Path Decomposition とは本来，キャッシュフレンドリーな Trie 辞書を実装するために用いられる技法であるが，本提案では省メモリな動的辞書を実現するためにこの技法を用いる．大規模な現実のデータセットを用いた実験により，提案手法は既存の最もメモリ効率の良い実装と比べ，最大で 2.8 倍もコンパクトに動的辞書を実現できることを示す．

1. はじめに

大規模な文字列データをいかに効率よく主記憶で管理するかというのは，現代の計算機科学において基本的な課題のひとつであり，今まで多くの研究者がメモリ効率の良いデータ構造の提案をおこなってきた．本稿では，文字列をキーとする連想配列の現実的なデータ構造について考える．このデータ構造は一般的に**キーワード辞書**とよばれ，自然言語処理や情報検索において用いられる古典的なデータ構造の一つである．一方で近年では文献 [1] で報告されるように，巨大なデータを扱うためにメモリ効率の良いキーワード辞書の実装が数多くの用途で求められている．例えば，Mavlyutov ら [2] は RDF データストアシステムにおいて 14GB から成る URI 集合の保管を考慮している．

静的な辞書に限定すれば，メモリ効率に優れた実装法がいくつか提案されている．例えば Martínez-Prieto ら [1] は，さまざまな技法を組み合わせた数種類の実装法を提案し評価を与えている．Grossi と Ottaviano は，**Trie** [3] として知られる木構造を用いたキャッシュフレンドリーな実装法を提案している．Kanda ら [4] は，これらの実装に対して辞書符号化を用いた圧縮方策を適用し評価を与えた．また Kanda ら [5] は，改良型 Double Array を用いた高速な辞書の実装法も提案している．これらによる実装は大規模なデータセットをコンパクトに保管することができるが，キーワードの追加や削除を提供していないためにその用途は限定される．

動的な辞書に関しては，Judy [6] や HAT-trie [7]，Cedar [8] などの効率的な実装がいくつか存在している．これらの実装はポインタのオーバヘッドを削減することでメモリ効率の改善を試みているが，静的な辞書と比べると依然としてはるかに多くのメモリを消費する．一方で，Trie をコンパクトかつ動的に表現するためのデータ構造もいくつか提案されている．Darragh ら [9] はコンパクトなハッシュ表を用いたデータ構造を提案しており，近年 Poyias と Raman [10] によってその改良である **m-Bonsai** が提案された．m-Bonsai は Trie を情報理論的下限に漸近的に一致するサイズで表現し，かつ基本的な木の操作を $O(1)$ 期待時間で実行できる．Takagi ら [11] もまた，オンライン文字列処理のための効率的なデータ構造を提案している．しかし，これらの動的 Trie 表現に関してキーワード辞書の実装に対する議論や評価は与えられていない．それ故に，よりメモリ効率の良いキーワード辞書の設計に取り組む必要がある．

そこで本稿では，Trie を変形するための技法である **Path Decomposition** [12] を用いた，メモリ効率の良い動的キーワード辞書の新しい実装法を提案する．Path Decomposition とは，本来キャッシュフレンドリーな Trie 構造を形成するために提案された技法であり，現在は静的な用途においてのみ用いられている [13, 14]．しかし提案手法では動的辞書を構築するためにこの技法を応用する．加えて，高

[1] 徳島大学大学院先端技術科学教育部
[2] 学術振興会特別研究員 DC
[a)] shnsk.knd@gmail.com

いメモリ効率を得るために m-Bonsai を適用する方法を提案する．提案手法に対する評価は，大規模な現実のデータセットを用いた実験により与える．

2. 準備

2.1 基本的な記法と定義

n 個の要素を持つ配列 A，すなわち $A[0]A[1]\dots A[n-1]$ を $A[0,n)$ と表す．この配列 $A[0,n)$ に対して，$|A|$ はその長さ n を表す．特殊な終端文字「$」を末尾に持つバイト文字列を**キーワード**とよぶ．すなわち，キーワード $w[0,n)$ に対して $\$ \notin w[0,n-1)$ と $w[n-1] = \$$ が成り立つ．対数の底は 2 で統一する．

2.2 Path Decomposition

Trie [3] とは，文字列集合を保存し検索するためのラベル付き順序木であり，文字列の接頭辞を併合することにより構築される．Path Decomposition [12] とは，この Trie を以下のような手順で Path-Decomposed Trie（PDT）へと変形する技法である．まず，Trie の根から任意の葉までの経路を選択し，その経路を PDT の根と関連付けて定義する．同様の処理をその経路から垂れ下がる各部分 Trie に対して再帰的に実行し，そうして得られた各部分 PDT の根を前処理で定義した PDT の根の子として，部分 Trie への分岐文字を付随した枝で繋ぐ．

Path Decomposition の本来の目的は，Trie の分解により木の高さを抑えることで，検索時に節点間のランダムアクセス回数を削減することである．Trie を分解するとき，どの子を辿り経路を選択するのかは任意であるが，どのような選び方を用いても PDT の高さはパトリシア木 [15] のそれを超えないことが保証されているため，ある程度のキャッシュ効率の改善は必ず見込める．本実装において重要な事実を以下に示す．

事実 1. PDT の各節点は元の Trie のある節点から葉までの経路に対応する．終端文字により，元の Trie の葉の数は登録キーワード数と一致するため，PDT の節点数は登録キーワード数と一致する．

2.3 m-Bonsai

m-Bonsai [10] とはメモリ効率の良い動的な Trie の表現法であり，開番地法を用いて各節点をハッシュ表 $Q[0,m)$ 上に定義する．仮に節点数を n としたとき，$\alpha = n/m$（$0 \le \alpha \le 1$）をハッシュ表 Q の**占有率**と呼ぶ．ここで n と m は予め与えられているものと想定する．各節点は Q のいずれかの要素に配置されるため，その番地 $0 \le i < m$ を節点の ID として用いる．節点 v から文字 c により示される子の追加は，以下の三つの手順により実現される．ただし，σ は枝に付随する文字のアルファベットサイズを表す．まず，子に対するハッシュキーとしてペア $\langle v, c\rangle$ を生成する．次に，ハッシュ関数 $h : \{0,\dots,m\cdot\sigma-1\} \to \{0,\dots,m-1\}$ を用いて，子の初期番地 $u = h(\langle v, c\rangle)$ を算出する．最後に，Q 上を初期番地 u から線形走査し最初に出会う空要素の番地を u' としたとき，$Q[u']$ にその子を配置する．すなわち，新たに追加された子の ID は u' となる．

メモリ使用量に関して，単純にハッシュキー $\langle v, c\rangle$ を $Q[u']$ に記述した場合，各要素は $\lceil \log(m\cdot\sigma)\rceil$ ビットを消費する．m-Bonsai では**短縮キー** [3, 演習問題 6.13] とよばれる技法を用いて，このメモリ使用量を $\lceil \log\sigma\rceil$ ビットにまで削減している．このとき u' から u までの距離，すなわち衝突回数を $D[u']$ に保持した転移配列 D を新しく導入する必要があるが，[10, 補題 1] よりその平均値は小さいことから，符号化などを用いることで D はコンパクトに表現できる．なお，節 4 の実験でも用いた第一著者による m-Bonsai の実装は `https://github.com/kamp78/bonsais` に公開している．

3. DynPDT

本節では，Path Decomposition を用いた新しい動的キーワード辞書の実装法，**DynPDT** を提案する．

3.1 基本的なアイデア

まず，DynPDT の基本的なアイデアである**逐次的 Path Decomposition** を説明する．逐次的 Path Decomposition では，各キーワードに対応する節点をその登録順に逐次的に定義することで PDT を構築する．以下にキーワード w を追加するときの手続きを示す．同時に DynPDT の構成も示す．

- 辞書が空であれば，文字列 w がラベル付けされた根を定義する．本稿では，節点 v に付随するラベルを L_v と表す．
- 辞書が空でなければ，変数 v に根の節点 ID を設定し，以下の二つの手順によりキーワードを探索する．手順 1 では w と L_v を比較する．もし $w = L_v$ であれば，このキーワードはすでに登録されているので手続きを終了する．そうでなければ，手順 2 において $w[i] \ne L_v[i]$ かつ $w[0,i) = L_v[0,i)$ であるようなラベル $\langle i, w[i]\rangle$ により示される子を探索する．もし子がなければ，$\langle i, w[i]\rangle$ をラベルとして付随した枝と接尾辞 $w[i+1,|w|)$ をラベルとして付随した子を追加する．もし子があれば，v にその子の節点 ID を，w に接尾辞 $w[i+1,|w|)$ を設定し手順 1 へ戻る．

すなわち，各節点ラベルは登録キーワードのいずれかの接尾辞を表し，枝ラベルは節点ラベル上のオフセットとバイト文字のペアにより構成される．検索も同様の手続きで実現される．逐次的 Path Decomposition の特徴としては，早くに登録されたキーワードに対応する節点ほど根の付近に配置され，そのようなキーワードに対する探索コストは

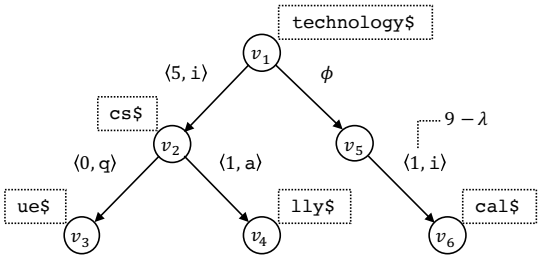


図 1: DynPDT の例（$\lambda = 8$）

低く抑えられることがあげられる．しかし，節 4 ではそのような特徴は考慮せず，無作為順にキーワードが追加された DynPDT に対して評価を与える．

3.2 m-Bonsai の適用

高いメモリ効率を得るために，DynPDT では m-Bonsai を用いてこの PDT を表現する．ただし，以下の問題が生じる．

問題 1. 枝ラベルは節点ラベル上のオフセット i とバイト文字 c から成るペア $\langle i, c\rangle$ である．節点ラベルの最大長を Λ としたとき，そのアルファベットサイズは $\sigma = 256 \cdot \Lambda$ となる．m-Bonsai はハッシュ表 Q に割り当てる領域とハッシュ関数 h を予め定義するためにパラメータ σ を固定する必要があるが，未知のキーワードを登録する動的な用途において Λ は固定し得ないパラメータである．

この問題を解決するために，新たなパラメータ λ を導入し，アルファベットサイズを $\sigma = 256 \cdot \lambda$ として強制的に固定する．もし L_v 上のオフセット i が λ 以上であれば，$i < \lambda$ となるまで以下の手順を繰り返す．

(1) 節点 v から特殊文字 ϕ によって示される子 u を加える．
(2) $v \leftarrow u$，$i \leftarrow i - \lambda$ と更新する．

この手順により定義される節点を**ステップ節点**とよぶことにする．この解決案では λ に応じてステップ節点を追加するため，λ が小さすぎれば大量のステップ節点を定義することになる．逆に λ が大きすぎれば Q の各要素は $\lceil \log(256 \cdot \lambda) \rceil$ ビットを確保するため，そのメモリ使用量が問題となる．故に適切な λ を定義する必要がある．

例として，キーワード `technology$`，`technics$`，`technique$`，`technically$`，`technological$`を，この順に追加することにより構築された DynPDT を図 1 に示す．例の節点は $v_1, v_2, \ldots, v_6$ の順に定義されている．例えば，この辞書を用いて `technically$`を検索する場合，まず w にこのキーワードを設定し節点ラベル L_{v_1} と比較する．結果として $w[0,5) = L_{v_1}[0,5) =$ `techn` なので，枝ラベル $\langle 5, w[5]\rangle = \langle 5, \mathtt{i}\rangle$ を用いて節点 v_2 に移動する．このとき，w には残りの接尾辞 `cally$`を設定する．次に w と L_{v_2} を比較し，$w[0,1) = L_{v_2}[0,1) = \mathtt{c}$ なので $\langle 1, w[1]\rangle = \langle 1, \mathtt{a}\rangle$ を用いて節点 v_4 へ移動し，w に残りの接尾辞 `lly$`を設定する．最終的に，$w = L_{v_4}$ よりこの検索キーワードが登録されていることがわかる．

加えて，`technological$`を検索するときの例も示す．前例と同じように，w に検索キーワードを設定し節点ラベル L_{v_1} と比較する．その結果は $w[0,9) = L_{v_1}[0,9) =$ `technolog` であるが，$\lambda \leq 9$ より定められたアルファベットサイズを超過するため，枝ラベル $\langle 9, \mathtt{i}\rangle$ は無効である．故に，特殊文字 ϕ を用いて節点 v_5 に移動する．$9 - \lambda < \lambda$，すなわち $1 < \lambda$ より，$\langle 1, \mathtt{i}\rangle$ を用いて節点 v_6 に移動することができる．最終的に，残りの接尾辞 `cal$`と L_{v_6} が一致するため，この検索キーワードが登録されていることがわかる．

3.3 具体的な実装

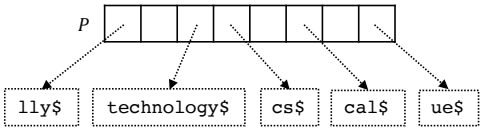
連想配列として，各キーワードに任意の値を連想付けられることは必須機能である．DynPDT では節点とキーワードが一意に対応するため，節点ラベルに続く領域を用いることで連想値を保持することができる．キーワードの削除に関しては，既存の開番地法と同様に各節点に対してそれを示すフラグを導入することで単純に実現できる．ただし，削除により発生する無駄な節点領域の解放に関しては今後の課題として残る．また m-Bonsai を利用するためには，節点数と占有率に依存した Q の要素数を予め定義する必要がある．言い換えれば，データセットに対して予期される節点数を見積もる必要がある．素朴な Trie を用いる場合この見積もりは難しいが，幸いにも事実 1 により，DynPDT の節点数は登録キーワード数と適切な λ に応じた少量のステップ節点数の合計となる．すなわち，おおよその節点数は容易に見積もることができる．

3.4 節点ラベルの保管

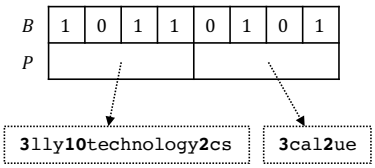
節点ラベルは可変長な文字列であるため，m-Bonsai とは，すなわちハッシュ表 Q と転移配列 D とは別の領域で保管される．このとき素朴な実装としては，配列 $P[0,m)$ を用いて $P[i]$ に L_i へのポインタを格納する方法が考えられる．本稿では，この実装法を **Plain** とよぶ．Plain は節点ラベルの参照と追加が定数時間でおこなえるが，m 個のポインタ領域を要するためメモリ使用量が大きいという欠点がある．図 2a にその例を示す．

そこで代替手段として，Google Sparse Hash[*1] と同様の技法で Plain におけるポインタのオーバヘッドを削減する実装法を提案する．この実装法では節点ラベルをその ID に対して，ℓ 個ずつの**グループ**に分割する．例えば，一番目のグループは $L_0 \ldots L_{\ell-1}$ から成り，二番目のグループは $L_\ell \ldots L_{2\ell-1}$ から成る．加えて，L_i が存在すれば $B[i] = 1$ となるようなビット列 B を導入する．各グループにおいて，$B[i] = 1$ である節点ラベル L_i を ID 順を維持しながら

[*1] https://github.com/sparsehash/sparsehash



(a) Plain



(b) Bitmap（$\ell = 4$）

図 2: 節点ラベルの保管方法の例

連結し，各グループの連結ラベル文字列へのポインタを保持することにより，P の長さを $\lceil m/\ell \rceil$ に削減できる．この実装法を **Bitmap** とよぶ．図 2b にその例を示す．ただし，節点ラベル中に含まれる太字の数字に関しては後述する．

Bitmap では，配列 P と B を用いて L_i を以下のように参照する．もし $B[i] = 0$ であれば L_i は存在しないことがわかる．$B[i] = 1$ であれば $g = \lfloor i/\ell \rfloor$ とし，$P[g]$ から L_i が含まれる連結ラベル文字列を取得する．加えて，部分ビット列 $B_g = B[g \cdot \ell, (g+1) \cdot \ell)$ も取得する．仮に，$B_g[0, i \bmod \ell + 1)$ に含まれる 1 の数を j としたとき，先ほど得た連結ラベル文字列の j 番目のラベルが L_i である．ℓ は定数なので，popcount 関数 [16] を用いることで B_g に含まれる 1 の数は定数時間で求めることができる．それ故に，参照時間は連結ラベル文字列の j 番目までを走査する時間と一致する．

単純に節点ラベルを連結した場合，この走査は終端文字を数え上げることにより実行され，参照は $O(\ell \cdot \Lambda)$ 時間でおこなわれる．この参照時間を短くするため，Array Hash [17] で用いられている **Skipping** 技法を応用する．Skipping 技法では，VByte 符号化 [18] を用いて各節点ラベルの前にその長さを記述する．このとき各節点ラベルの終端文字は排除することができる．Skipping 技法では，その長さを利用し次の節点ラベルの始点まで移動することができるので，参照を $O(\ell)$ 時間で実行できる．Skipping 技法を適用した結果が図 2b である．

4. 実験による評価

本節では実験により DynPDT を評価する．また，実験に用いた DynPDT の実装は `https://github.com/kamp78/dynpdt` に公開している．

4.1 実験設定

実験に用いた計算機の構成は，Intel Xeon E5540 @2.53 GHz CPU, 32 GB RAM（L2 cache : 1 MB，L3 cache : 8 MB）であり，OS は Ubuntu Server 16.04 LTS である．実装言語は C++で，最適化オプションとして-O9 を指定し，g++ version 5.4.0 を用いてコンパイルした．メモリ使用量の計測には `/proc/<PID>/statm` を用いた．実行時間の計測には `std::chrono::duration_cast` を用いた．実験用データセットは以下である．

表 1: データセットに関する情報

	Wiki	UK	WebBase	LUBM
Size（MiB）	227.2	2,723.3	6,782.1	3,194.1
Keys（M）	11.5	39.5	118.1	52.6
Nodes（M）	111.0	748.6	1,426.3	247.7
NPK	9.6	19.0	12.1	4.7
BPK	20.7	72.4	60.2	63.7

- Wiki：英語版 Wikipedia のタイトル記事集合[*2]
- UK：`.uk` ドメイン上で UbiCrawler [19] により得られた URL 集合[*3]
- WebBase：WebBase クローラ [20] により得られた URL 集合[*4]
- LUBM：LUBM ベンチマーク [21] により生成されたデータセットから抽出した URI 集合[*5]

表 1 にその詳細を示す．「Size」は生データの容量，「Keys」は登録キーワード数，「Nodes」は素朴な Trie を構築した場合の節点数，「NPK」はキーワード当たりの素朴な Trie の節点数の平均，「BPK」はキーワード当たりの平均バイト数，すなわちキーワードの平均長を示している．

4.1.1 実装

本実験では，DynPDT を m-Bonsai 及び既存の動的辞書と比較することで評価を与えた．DynPDT に関しては，Plain と Bitmap（$\ell = 8, 16, 32, 64$）の両方を実装した．DynPDT 辞書の連想値には `int` を設定したが，m-Bonsai は辞書として実装法が提案されていないので，連想値を持たない素朴な Trie として実装した．既存の動的辞書にはメモリ効率の良い以下の 3 つの実装を選んだ．

- Judy：Hewlett-Packard 研究所により開発された動的 Trie 辞書 [6]
- HAT-trie：Trie と Array Hash のハイブリッドによる動的辞書 [7]
- Cedar：Double Array を用いた最小接頭辞 Trie による動的辞書 [8]

DynPDT と同様に，これらにも連想値として `int` を設定した．HAT-trie には，`https://github.com/dcjones/hat-trie` で公開されている実装を用いた．Cedar のみアドレス表現に 32 ビット整数を用いているため，巨大なデータセットの WebBase に対して辞書を構築することができなかった．

[*2] `https://dumps.wikimedia.org/enwiki/`
[*3] `http://law.di.unimi.it/webdata/uk-2005/`
[*4] `http://law.di.unimi.it/webdata/webbase-2001/`
[*5] `https://exascale.info/projects/web-of-data-uri/`

表 2: メモリ使用量に関する実験結果

Data Structure	Wiki	UK	WebBase	LUBM
Plain	46.6	54.4	47.5	45.0
Bitmap-8	21.2	31.3	24.0	15.5
Bitmap-16	18.8	28.2	21.0	13.8
Bitmap-32	17.4	27.1	19.8	12.1
Bitmap-64	16.9	26.4	19.2	11.5
m-Bonsai	23.6	46.1	29.3	11.4
Judy	50.5	60.3	53.5	33.9
HAT-trie	40.2	82.3	68.9	64.7
Cedar	41.1	58.4	–	29.7

表 3: 追加時間に関する実験結果

Data Structure	Wiki	UK	WebBase	LUBM
Plain	1.14	1.65	2.37	1.65
Bitmap-8	1.38	1.99	2.64	1.91
Bitmap-16	1.57	2.29	2.93	1.99
Bitmap-32	1.93	2.91	3.47	2.29
Bitmap-64	2.65	4.12	4.60	2.87
m-Bonsai	2.22	7.13	7.69	4.80
Judy	1.06	2.15	2.94	1.53
HAT-trie	1.13	1.63	1.75	2.58
Cedar	1.07	2.56	–	2.50

表 4: 検索時間に関する実験結果

Data Structure	Wiki	UK	WebBase	LUBM
Plain	1.13	1.53	2.20	1.12
Bitmap-8	1.38	2.15	2.40	1.26
Bitmap-16	1.61	2.47	2.74	1.43
Bitmap-32	2.06	3.25	3.72	1.61
Bitmap-64	3.01	4.88	5.29	2.16
m-Bonsai	2.06	6.69	8.30	3.08
Judy	0.88	2.02	2.42	0.79
HAT-trie	0.35	0.61	0.80	0.51
Cedar	0.69	2.51	–	0.69

4.1.2 パラメータ

DynPDT と m-Bonsai は占有率 α をパラメータとして持つ．本実験では，既存の実験 [9,10] と同じように $\alpha = 0.8$ を設定した．故に，DynPDT では登録キーワード数を 0.8 で割った値をハッシュ表 Q の長さとして定義した．ここで，結果として得られる DynPDT の占有率を α' とすると，ステップ節点の数に依存し $\alpha \leq \alpha'$ であるということに留意されたい．m-Bonsai では，予め見積もった素朴な Trie の節点数を 0.8 で割った値をハッシュ表 Q の長さとして定義した．すなわち，NPK が小さくなるにしたがって DynPDT と m-Bonsai における Q の長さの差は小さくなる．

DynPDT については，ステップ節点数や Q のメモリ使用量を左右するパラメータ λ も事前に設定する必要がある．λ が大きすぎると，Q に割り当てられる領域が大きくなる．逆に λ が小さすぎると，ステップ節点が大量に作られ α' が大きくなる．α' が大きくなると衝突回数が増え D の平均値が大きくなるため，結果としてメモリ使用量は増加し探索速度も低下する．そこで適した λ を見つけるために，各データセットに対して $\lambda \in [2, 4, 8, 16, 32, 64, 128]$ の範囲で予備実験をおこなった．その結果より，$\alpha' \leq 0.81$ となるような最小の λ として，Wiki で 16，UK で 64，WebBase で 32，LUBM で 16 が得られた．これらの値を本実験では λ に設定した．

4.2 実験結果

キーワードを無作為の順に追加することで辞書を構築し，そのときの Resident Set Size をメモリ使用量として計測した．追加時間は 3 回の試行平均である．検索は各データセットから無作為に抽出した 100 万のキーワードに対し実行した．示す結果は 10 回の試行平均である．

4.2.1 メモリ使用量

表 2 に実験結果を示す．Plain におけるポインタのオーバヘッドは Bitmap により大きく削減されているのがわかる．LUBM において，Bitmap-64 は Plain よりも 3.9 倍小さい．また，m-Bonsai と Bitmap-64 を比べると，LUBM を除いて Bitmap-64 は 1.4–1.7 倍小さい．しかし LUBM では，NPK が小さく m-Bonsai と DynPDT の Q の差が小さいため，m-Bonsai の方がわずかに小さくなった．ただし，m-Bonsai は連想値を保持しておらず，仮に m-Bonsai がメモリに関してのオーバヘッドが一切なく理想的に連想値を保持したとすれば，キーワードごとに `sizeof(int)` = 4 バイトずつが加算される計算になる．すなわち，辞書としては Bitmap-64 は全てのデータセットに対して m-Bonsai よりも小さくなる．既存の辞書と比べても Bitmap-64 は最も小さく，各データセットにおける最もコンパクトな結果と比較して，その差は 2.2–2.8 倍であった．

4.2.2 追加時間

表 3 に結果を示す．DynPDT において，必然的に Plain が最も高速で Bitmap-64 が最も低速となった．しかし，Bitmap-8 と Plain にメモリ使用量ほどの大きな差は見られない．以下では，Bitmap-8 とその他の実装とを比較する．まず m-Bonsai と比較した場合，すべてのデータセットにおいて Bitmap-8 が高速であった．BPK の大きい UK や WebBase においては Path Decomposition の効果が顕著に現れ，その差は UK で 3.6 倍，WebBase で 2.9 倍であった．既存の辞書と比べた場合，Bitmap-8 はもっとも高速というわけではなかったが，Wiki を除いて 2 番目に良好な結果を示した．

4.2.3 検索時間

表 4 に結果を示す．追加時間と同様に，DynPDT において Plain がもっとも高速となり，Bitmap-8 は m-Bonsai よりも高速であった．一方で既存の辞書と比べると，Dyn-

PDT は基本的に低速であり，Bitmap-8 と HAT-trie との差は最大で 3.9 倍であった．ただし，UK と WebBase においては，Bitmap-8 は Judy と Cedar に近い値を示している．

5. おわりに

本稿では，Path Decomposition を用いたメモリ効率のよい動的キーワード辞書の新しい実装法である DynPDT を提案した．また，実験により既存の実装とくらべて DynPDT は大幅にコンパクトであることを示した．一方で，その時間効率はあまり高くなかった．m-Bonsai における節点間の遷移速度がその他のポインタベースの実装とくらべて低速であったことが，その主な原因として考えられる．また，開番地法を用いているため，ハッシュ表の拡張，縮小が簡単におこなえないことも別の問題点として挙げられる．今後の課題としては，これらの欠点を改善する別の Trie 表現を考案することがあげられる．

また本稿では，与えられたキーワードに対しその連想値を返すといった連想配列としての基本的な機能のみを取り扱ったが，キーワードに固有の ID を与え可逆な操作を提供する辞書構造も多くの用途において重要とされている [1]．m-Bonsai は葉から根に向かっての遷移を提供しているため，原理的に DynPDT はこの可逆な操作を提供できる．今後の課題としては，このような辞書に適合した DynPDT の提案および評価をおこなう．なお，これらの成果は論文 [22] で発表予定である．

謝辞 本研究は，日本学術振興会科研費 17J07555 の助成を受けておこなわれた．

参考文献

[1] Martínez-Prieto, M. A., Brisaboa, N., Cánovas, R., Claude, F. and Navarro, G.: Practical compressed string dictionaries, *Information Systems*, Vol. 56, pp. 73–108 (2016).
[2] Mavlyutov, R., Wylot, M. and Cudre-Mauroux, P.: A Comparison of Data Structures to Manage URIs on the Web of Data, *Proc. 12th ESWC*, pp. 137–151 (2015).
[3] Knuth, D. E.: *The art of computer programming, 3: sorting and searching*, Addison Wesley, Redwood City, CA, USA, 2nd edition (1998).
[4] Kanda, S., Morita, K. and Fuketa, M.: Practical string dictionary compression using string dictionary encoding, *Proc. 3rd Innovate-Data* (to appear).
[5] Kanda, S., Morita, K. and Fuketa, M.: Compressed double-array tries for string dictionaries supporting fast lookup, *Knowledge and Information Systems*, Vol. 51, No. 3, pp. 1023–1042 (2017).
[6] Baskins, D.: *Judy IV Shop Manual* (2002).
[7] Askitis, N. and Sinha, R.: Engineering scalable, cache and space efficient tries for strings, *The VLDB Journal*, Vol. 19, No. 5, pp. 633–660 (2010).
[8] Yoshinaga, N. and Kitsuregawa, M.: A self-adaptive classifier for efficient text-stream processing, *Proc. 24th COLING*, pp. 1091–1102 (2014).
[9] Darragh, J. J., Cleary, J. G. and Witten, I. H.: Bonsai: a compact representation of trees, *Software: Practice and Experience*, Vol. 23, No. 3, pp. 277–291 (1993).
[10] Poyias, A. and Raman, R.: Improved practical compact dynamic tries, *Proc. 22nd SPIRE*, pp. 324–336 (2015).
[11] Takagi, T., Inenaga, S., Sadakane, K. and Arimura, H.: Packed compact tries: A fast and efficient data structure for online string processing, *Proc. 27th IWOCA*, pp. 213–225 (2016).
[12] Ferragina, P., Grossi, R., Gupta, A., Shah, R. and Vitter, J. S.: On searching compressed string collections cache-obliviously, *Proc. 27th PODS*, pp. 181–190 (2008).
[13] Grossi, R. and Ottaviano, G.: Fast compressed tries through path decompositions, *ACM Journal of Experimental Algorithmics*, Vol. 19, No. 1, p. Article 1.8 (2014).
[14] Hsu, B.-J. P. and Ottaviano, G.: Space-efficient data structures for top-k completion, *Proc. 22nd WWW*, pp. 583–594 (2013).
[15] Morrison, D. R.: PATRICIA: practical algorithm to retrieve information coded in alphanumeric, *Journal of the ACM*, Vol. 15, No. 4, pp. 514–534 (1968).
[16] González, R., Grabowski, S., Mäkinen, V. and Navarro, G.: Practical implementation of rank and select queries, *Poster Proc. 4th WEA*, pp. 27–38 (2005).
[17] Askitis, N. and Zobel, J.: Cache-conscious collision resolution in string hash tables, *Proc. 12th SPIRE*, pp. 91–102 (2005).
[18] Williams, H. E. and Zobel, J.: Compressing integers for fast file access, *Computer Journal*, Vol. 42, No. 3, pp. 193–201 (1999).
[19] Boldi, P., Codenotti, B., Santini, M. and Vigna, S.: Ubicrawler: A scalable fully distributed web crawler, *Software: Practice and Experience*, Vol. 34, No. 8, pp. 711–726 (2004).
[20] Hirai, J., Raghavan, S., Garcia-Molina, H. and Paepcke, A.: WebBase: A repository of web pages, *Computer Networks*, Vol. 33, No. 1, pp. 277–293 (2000).
[21] Guo, Y., Pan, Z. and Heflin, J.: LUBM: A benchmark for OWL knowledge base systems, *Web Semantics*, Vol. 3, No. 2, pp. 158–182 (2005).
[22] Kanda, S., Morita, K. and Fuketa, M.: Practical implementation of space-efficient dynamic keyword dictionaries, *Proc. 24th SPIRE* (to appear).