ダブル配列を用いたパトリシアトライによる動的キーワード辞書の実装

松本 拓真[1,a)]　森田 和宏[2,b)]　泓田 正雄[2,c)]

受付日 2019年9月10日, 採録日 2020年1月6日

概要： ダブル配列トライは文字列をキーとした辞書の実現に広く用いられている．辞書の動的な更新を必要とする用途において，従来手法では，分岐のない頂点以降の接尾辞を文字列で表現した接頭辞トライとしてキー集合を表現し，空間効率の良い辞書を実現している．しかし辞書を実現するためのトライとしては，分岐のある頂点のみで構成されるパトリシアトライとして表現することでより少ない頂点数で辞書を実現できる．そこで我々は，ダブル配列トライをパトリシアトライで構成する方法と，空間効率の無駄のない枝の分岐アルゴリズムを提案した．実社会で用いられるデータセットを用いて辞書を構築する実験では，提案手法は従来手法に対してメモリ消費量と検索時間を改善し，特に検索時間を大きく改善した．

キーワード： キーワード辞書，ダブル配列，パトリシアトライ

Implementation of Dynamic Keyword Dictionary by Patricia Trie Using Double-Array

Takuma Matsumoto[1,a)]　Kazuhiro Morita[2,b)]　Masao Fuketa[2,c)]

Received: September 10, 2019, Accepted: January 6, 2020

Abstract: Double-Array Trie is widely used for implementing the keyword dictionaries. For applications that require dictionary updates, conventional implementations are space-efficient by representing key sets as Minimal-prefix Trie. However, it is fact that Patricia Trie which has only vertexes more than 2 degrees can represent key sets using less number of vertexes than Minimal-prefix Trie. Therefore, we propose implementation of dictionary as Patricia Trie using Double-Array and update methods that lean for memory consumption. Experiments using datasets in real world shows our proposed method is more efficient in memory consumption and time in search, that is large contribution for the time in search particularly.

Keywords: keyword dictionary, Double-Array, Patricia Trie

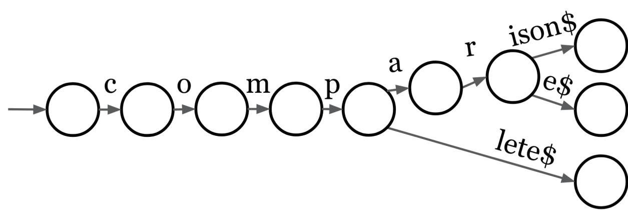
1. はじめに

現代の計算機において，大規模な文字列情報をいかに効率良く扱うかというのは古くから基本的な課題の１つである．文字列を扱うデータ構造は様々に存在するが，本研究では動的なキーワード辞書の設計と実装について論じる．

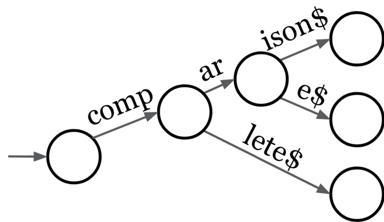
キーワード辞書とは，文字列からなるキー集合を主記憶で効率的に保存し検索を提供するデータ構造であり，多くの用途においてキーに対応した連想値の対を記憶する．キーワード辞書の設計と実装については古くから研究されており，ハッシュ表やトライ，FrontCoding など，様々な技法を基礎とした実装が存在する．辞書の性能は一般にメモリ消費量と検索速度で評価され，用途に応じて効率的な辞書が必要とされている．特に近年，ウェブエンジンやセマンティックウェブグラフ，バイオインフォマティクスなどの応用において大規模な辞書を扱う事例が報告されており [1]，コンパクトで検索速度の良いキーワード辞書が提案されている [1], [2], [3], [4], [5]．しかしこれらで用いられる辞書はあらかじめ用意されたデータセットから構築され，キーの追加や削除を必要としない用途である．一方で，辞書に対するキーの追加や削除を必要とした動的

[1] 徳島大学大学院先端技術科学教育部
Faculty of Engineering, Tokushima University, Tokushima 770–8506, Japan
[2] 徳島大学大学院ソシオテクノサイエンス研究部
Institute of Technology and Science, Tokushima University, Tokushima 770–8506, Japan
[a)] tkm.matsumoto@gmail.com
[b)] kam@is.tokushima-u.ac.jp
[c)] fuketa@is.tokushima-u.ac.jp

© 2020 Information Processing Society of Japan



(a) 最小接頭辞トライ



(b) パトリシアトライ

図 1　キー集合 {“comparison”, “compare”, “complete”} を表現するトライ

Fig. 1　Tries representing key set {“comparison”, “compare”, “complete”}.

な用途でキーワード辞書を用いる事例が存在し，更新速度や検索速度に優れた動的キーワード辞書が提案されている [6], [7], [8], [9], [10], [11]．動的キーワード辞書を用いた事例の１つとして，Yoshinaga ら [12] は多くの自然言語処理タスクに向けた効率的なテキストストリーム処理のための自己適応的分類器において，一般分類問題の管理に動的キーワード辞書を用いている．

Yoshinaga らの研究で用いられている動的キーワード辞書に採用されている実装は，最小接頭辞ダブル配列 [7] という検索速度に優れたデータ構造である．ダブル配列 [13] は，2 つの 1 次元配列によりトライを表現するデータ構造であり，トライ上の枝の遷移を定数時間で実行できる特徴から高速な検索を提供する．また最小接頭辞トライは，文字列集合をトライで表現した際の分岐がなくなった頂点より後に続く経路のラベルを文字列で表現したものである．最小接頭辞トライの例を図 **1** (a) に示す．文字列集合の表現に最小接頭辞トライを用いることで，頂点数の減少による空間効率の改善と，単純な文字列比較で検索できることによる時間効率の向上が期待できる．Yata ら [7] は最小接頭辞トライ表現をダブル配列により動的に構成する方法を提案し，空間効率の改善と検索速度に優れた辞書を実現している．

一方で，文字列集合の表現に関してより簡潔なトライ構造であるパトリシアトライが存在する．パトリシアトライはトライを分岐のある頂点のみで構成するため，最小接頭辞トライで存在した分岐のない頂点を削除して文字列集合を表現できる．パトリシアトライの例を図 1 (b) に示す．頂点数を削減した文字列による遷移ラベル表現で辞書を構成することでさらに空間効率と時間効率の改善が期待できるが，現時点でダブル配列において接尾辞以外の遷移ラベルを文字列として表現する方法は提案されていない．この問題と類似した課題に関して，伊藤 [14] はダブル配列と同じく単配置法 [15] を基礎としたグラフ表現技法を基にした静的辞書の構成において，遷移ラベルに文字列を利用するための表現方法を提案している．

キーワード辞書の設計において，キー集合に含まれる文字列を空間効率に無駄のない程度に可能な限り文字列のまま保管する工夫は，検索時に単純な文字列比較が行える箇所を増やし，時間効率に優れた辞書を設計するうえで重要な要素になる．Dynamic Path-Decomposed Trie [9], [10] はトライの根から葉までの経路を文字列で表現するよう再帰的にトライを分解する Path Decomposition [16] を利用した動的辞書であり，高い空間効率と時間効率を実現している．文字列集合表現にパトリシアトライを用いることも，キー集合を可能な限り文字列のまま保管する工夫の１つであり，辞書の空間効率と時間効率の向上が期待できる．

本稿では，パトリシアトライをダブル配列で実現した検索効率の良い動的キーワード辞書の構成方法を提案する．方法は Yata らの動的接頭辞ダブル配列を基礎とし，伊藤の遷移ラベル表現方法を基にトライのすべての遷移ラベルを文字列で表現する方法と効率的な動的更新方法を提案する．この方法は，トライの頂点数を削減することで頂点の表現に必要な記憶量を削減し，キー集合に含まれる多くの文字列を単純に保存できるため高速な検索を実現できる．最後に，提案手法を用いて実装した動的キーワード辞書の性能を実験により評価し，提案手法が従来手法よりメモリ消費量と検索速度に優れた手法であることを示す．また，研究の成果として提案手法による動的キーワード辞書をライブラリ[*1]として公開している．

2. 最小接頭辞ダブル配列の動的な構成

トライ（Trie）[17], [18] は，枝に文字を付随した木構造により文字列集合を表現する技法であり，各文字列に対応した葉に連想値を保存するという方針で容易に辞書を実現できる．また最小接頭辞トライ（Minimal-prefix Trie, MP Trie）は，トライにおける分岐のなくなった頂点から葉までの経路に対応する接尾辞を文字列で表現することで頂点数を削減した技法である．最小接頭辞トライを用いて動的キーワード辞書を設計した手法として，Yata らの最小接頭辞ダブル配列（MP Double-Array）[7] がある．本章では，トライを構成するダブル配列という技法と，最小接頭辞ダブル配列の動的な構成方法を説明する．

2.1 最小接頭辞ダブル配列

ダブル配列は，BASE と CHECK と呼ばれる 2 つの 1 次元配列によりトライを表現する．ダブル配列の各インデッ

[*1] https://gitlab.com/MatsuTaku/patricia-double-array-tries

© 2020 Information Processing Society of Japan

クスに対応する要素はトライの各頂点に対応しており，ダブル配列における有効要素数はトライの頂点数と等しい．ダブル配列における頂点 s から t への文字 c による遷移は以下の式により実現される．

$$\begin{cases} t \leftarrow \mathrm{BASE}[s] + c \\ \mathrm{CHECK}[t] = s \end{cases} \tag{1}$$

ダブル配列におけるトライの遷移は式 (1) により定数時間で行われるため，高速な検索が実現できる．また，ダブル配列でトライを表現する場合，各文字列の末尾に終端文字 ‘\$’ を付随して保存することで，保存された文字列に対応する葉を生成する．

最小接頭辞トライを表現する場合には，接尾辞を保存するための配列 TAIL を導入する．最小接頭辞トライにおいて頂点 s が葉である場合，対応する接尾辞の 2 文字目以降の接尾辞を TAIL に保存し，TAIL 上の接尾辞の先頭インデックスを BASE[s] に保存する．また，頂点 s が葉か否かを表現するビット列 LEAF（LEAF[i] = {1：頂点 i が葉，0：葉でない }）を導入することで，BASE の値を区別する．以上により最小接頭辞ダブル配列での検索は，頂点 s について LEAF[s] = 0 である限り検索文字から 1 文字ずつ得た文字によりダブル配列上で遷移を行い，最後に終端文字 ‘\$’ による遷移が成功するか，LEAF[s] = 1 となった場合に TAIL 上で残りの検索文字との文字列比較を行うことで実行される．TAIL 上の接尾辞上での検索は，単純な文字列の比較で行えるため高速である．

例として，キー集合 {“compare”, “comparison”, “complete”} で構成される最小接頭辞ダブル配列を図 **2** に示す．図においては，アルファベットは {‘a’=0, ‘c’=1, ‘e’=2, ‘i’=3, ‘l’=4, ‘m’=5, ‘o’=6, ‘p’=7, ‘r’=8} のように対応するとしている．たとえば，キー “complete” で検索する場合，まず頂点 0 から文字 ‘c’ による遷移が実行され，

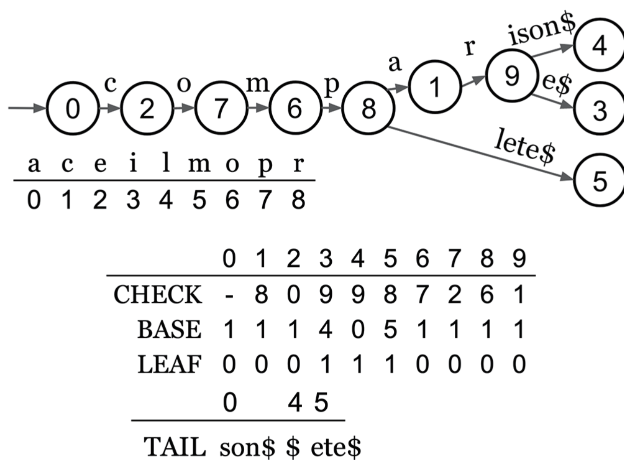


図 2 キー集合 {“compare”, “comparison”, “complete”} で構成される最小接頭辞ダブル配列

Fig. 2 A minimal-prefix double-array constructed from key set {“comparison”, “compare”, “complete”}.

BASE[0] + ‘c’(= 1) = 2, CHECK[2] = 0 より頂点 0 から 2 への遷移に成功する．以後 ‘o’，‘m’，‘p’，‘l’ による遷移の実行により，頂点 5 に到達する．ここで LEAF[5] = 1 により，TAIL 配列の添字 BASE[5] = 5 が得られ，TAIL[5] から残りの文字列 “ete” による比較が行われる．“ete” による比較が成功し，終端文字 ‘\$’ を確認することで，“complete” による検索が成功したことになる．

2.2 最小接頭辞ダブル配列の動的更新

まず，ダブル配列の動的更新で用いる諸関数を以下に示す．

- Empty(i)：ダブル配列の i 番目の要素が空要素かどうかの真偽値を返す．
- FindBase(C)：アルファベット集合 $C = \{c_1, c_2, \ldots, c_m\}$ に対し，$\prod_{\cap}$ Empty($x + c_i$) = True を満たすいずれかの x を返す．
- Children(s)：頂点 s の子へのラベル集合 $C = \{c_i :$ CHECK[BASE[s] + c_i] = $s\}$ を返す．
- Grow(s, c)：葉 s から文字 c により遷移される子を追加する．BASE[s] ← FindBase({c}) とした後，CHECK[BASE[s]] ← s を設定する．
- ResolveCollision(s, t, c)：頂点 s から文字 c による子を追加する際に，頂点 t の子のインデックスと衝突している場合に，いずれかの頂点の子要素の移動により衝突を回避する．頂点 s と t のうち，子が少ない方の頂点を v とすると，頂点 v の新たな BASE 値 b を FindBase で得た後，頂点 v の子に対応するダブル配列の要素を b に基づいた位置に移動し，BASE[v] ← b とする．
- InsertEdge(s, c)：葉ではない頂点 s に文字 c により遷移される子を追加する．Empty(BASE[s] + c) = False の場合，ResolveCollision(s, CHECK[BASE[s] + c], c) を行い，最後に，CHECK[BASE[s] + c] ← s とする．
- InsertInBc(s, *suffix*)：ダブル配列の頂点 s から接尾辞 *suffix* による分岐を生成する．*suffix* の末尾に終端文字 ‘\$’ を追加し，まず InsertEdge(s, *suffix*[1]) を実行する．l ← |TAIL| + 1 を得た後，残りの文字列 *suffix*[2, . . .] を TAIL の末尾に追加する．そして LEAF[BASE[s] + *suffix*[1]] ← 1 と BASE[BASE[s] + *suffix*[1]] ← l を設定する．
- InsertInTail(s, *tailpos*, *suffix*)：葉 s への遷移の TAIL に保存されている接尾辞の *tailpos* 文字目から，文字 *suffix* による分岐を生成する．i ← BASE[s] を得て，TAIL[$i, \ldots, i$ + *tailpos* − 1] の文字列から一文字ずつ Grow 関数の実行と遷移を繰り返し，分岐位置まで 1 文字ずつの枝に成長させる．ここで到達した頂点番号 s' により，Grow(s', TAIL[*tailpos*]) と CHECK[BASE[s'] + TAIL[*tailpos*]] ← s' および

© 2020 Information Processing Society of Japan

$\mathrm{BASE}[\mathrm{BASE}[s'] + \mathrm{TAIL}[tailpos]] \leftarrow tailpos + 1$ として接尾辞の先頭インデックスを移動する．その後 $\textsc{InsertInBc}(s', \mathit{suffix}[2, \ldots] + \text{'\$'})$ を実行する．

関数 $\textsc{FindBase}(C)$ に関して，Morita ら [19] や Yata ら [7] はダブル配列の空要素を双方向連結リストで管理する手法（*empty-link method*）により，空要素数に依存した計算量で効率的に x を見つける方法を提案している．関数 $\textsc{Children}(s)$ に関して，Morita らはダブル配列の各要素に 2 文字分の記憶量を追加することで兄弟を単方向連結リストで管理することで効率的に処理している．

最小接頭辞ダブル配列に新たなキー k を追加する場合，まず k による検索が実行される．検索に成功した場合はキーの追加処理は行わないが，検索が失敗した場合に失敗した箇所に応じた更新処理が行われる．k の i 番目の文字でダブル配列上の遷移に失敗した場合，遷移に失敗した頂点を s とすると，$\textsc{InsetInBc}(s, k[i, \ldots])$ により k が挿入される．また，TAIL 上の接尾辞で TAIL[j] で検索に失敗した場合，対応する葉を s として $\textsc{InsertInTail}(s, j, k[i, \ldots])$ により k が挿入される．

2.3　連想値の保存

キーワード辞書をトライで実現する場合は，キーに対応する葉に連想値を保存するという方針でキーと連想値を対応させるのが簡潔な方法である．Yoshinaga ら [12] の実装では，TAIL 上に追加した接尾辞の後ろに続く領域に連想値の保存に必要な記憶量を確保することで，検索時に確認した接尾辞のアドレスから連続して連想値にアクセスできるようにしている．

3.　パトリシアトライのダブル配列表現と動的構成

文字列集合を表現するトライの変形としてパトリシアトライがある．パトリシアトライではすべての頂点は分岐のある頂点のみで構成され，遷移ラベルは文字列で表現される．図 1 (b) は図 2 の最小接頭辞トライと同様の文字列集合を表現したパトリシアトライである．最小限の頂点数を持つパトリシアトライによって文字列集合を表現することで，頂点数に起因するデータ構造の肥大化を防ぎ，検索時の遷移回数の削減と文字列表現の単純化による検索の高速化が期待できる．

ダブル配列を用いてパトリシアトライを表現する場合の問題点は，接尾辞ではない遷移ラベルの文字列を表現する方法が確立されていないことである．接尾辞ダブル配列での接尾辞の表現は，遷移を持たない葉に対応する BASE の要素に遷移ラベルのアドレスを保存することで接尾辞を文字列で表現している．しかしパトリシアトライで内部の遷移ラベルへのアドレスを同様に BASE に保存してしまうと，次の遷移に利用する BASE 値を保存する領域を失ってしまう．伊東 [14] は，静的な辞書を実装するうえで，文字列集合を MDAWG（Minimal Directed Acyclic Word Graph）により表現する方法と，その実装にダブル配列と同じく単配置法を基にした決定性有限オートマトンの実装法である Revuz の手法 [20] を基礎とした手法を提案している．伊東の手法では，遷移ラベルを配列 POOL に保存し，ラベルの末尾に現在の頂点の BASE 値を保存することで POOL の文字列で比較を行った後も以降の遷移を実行できるようにしている．この方法は遷移ラベルと BASE 値が連続して保存されていることで，トライの探索時に効率的に値を得ることができ，また内部の遷移ラベルへのアドレスを BASE へ保存した場合でも頂点の BASE 値を保存する領域を与えている．

本章では，伊藤らの手法を基にしたアイデアで内部の遷移ラベルを表現することでパトリシアトライをダブル配列において実現する方法と，遷移ラベル上で分岐が発生した際の空間効率の良い更新アルゴリズムを提案する．

3.1　準備

以降の説明で用いるいくつかの定義を以下に記す．本稿で議論する計算機の語長を w ビットとし，整数値は w ビットで表現される*2．また，辞書に出現するアルファベット集合を Σ とし，アルファベットの種類数を $\sigma = |\Sigma|$ とする．よって各文字は $\lceil \log_2 \sigma \rceil$ ビットで表現される*3．

3.2　内部の遷移ラベルの表現

まず，最小接頭辞ダブル配列で用いていた TAIL を POOL と改める．そして頂点 s から頂点 t への遷移に対応するラベルを $l_{s \to t}$，頂点 t の BASE 値を b_t として，b_t と $l_{s \to t}$ の 2 文字目以降の文字列は POOL 上に $\{b_t, l_{s \to t}[2, \ldots], \text{'\$'}\}$ の順で連続して保存される．b_t は POOL の要素の $W = \left\lceil \frac{w}{\lceil \log_2 \sigma \rceil} \right\rceil$ 個分の記憶量を要する．そしてこれらが保存された位置の先頭のインデックスを BASE[t] に保存する．伊藤の手法との違いとして，b_t を先頭に保存することで，辞書を更新する際の b_t の更新時に値にアクセスしやすいようにしている．ただし，頂点 t が葉である場合には後続する遷移が存在せず b_t を保存する必要がないため，従来手法と同様に接尾辞と '\$' のみを POOL に保存する．これを区別するため，ビット列 LABEL（LABEL[i]=$\{$1：頂点 i の遷移ラベル長が 2 以上，0：遷移ラベル長が 1$\}$）導入する．これにより，BASE[i] に保存される値は，頂点 i の BASE 値，葉 i への遷移接尾辞が保存される POOL の先頭アドレス，頂点 i の BASE 値と頂点 i への内部遷移ラベルが保存される POOL の先頭アドレス，の 3 つの役割を区別して用いることになる．LABEL，LEAF の値に基づく BASE の値の役

*2　一般に w は 32 や 64 である．

*3　多くの場合で $\lceil \log_2 \sigma \rceil = 8$ として文字を表現することで，計算機で扱いやすくしている．

© 2020 Information Processing Society of Japan

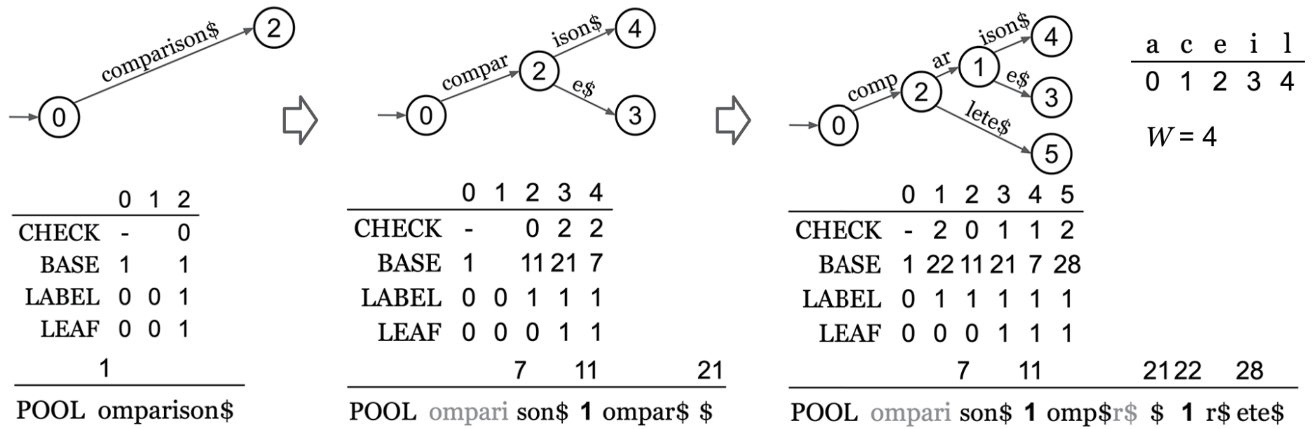


図 3　文字列 “comparison”，“compare”，“complete” を順に挿入した際のパトリシア–ダブル配列の過程

Fig. 3　Process of inserting strings “comparison”, “compare”, “complete” sequentially.

表 1　LABEL と LEAF の値に対応する BASE 値の役割

Table 1　The roles of BASE value determined by LABEL and LEAF.

LABEL	LEAF	BASE の役割
0	0	BASE 値
1	0	{BASE 値, 内部遷移ラベル, ‘$’} の先頭アドレス
1	1	{ 接尾辞, ‘$’} の先頭アドレス

割を**表 1** に整理する．

例として，文字列 “comparison”，“compare”，“complete” を順に挿入した際のパトリシア–ダブル配列の過程を**図 3** に示す．図においてアルファベットは {‘a’ = 0, ‘c’ = 1, ‘e’=2, ‘i’=3, ‘l’ = 4} に対応しているとする．また，BASE 値の表現に必要な POOL の要素数 W を 4 としてインデックスを与えている[*4]．

3.3　提案手法の検索

提案手法の検索アルゴリズムを Algorithm 1 に示す．関数 $\textsc{Search}(key)$ は，文字列 key が辞書に登録されているか否かの真偽値を返す．提案手法では頂点 i の BASE 値 b_i が BASE 配列と POOL のいずれかに保存されることになるため，関数 $\textsc{Base}(s)$ により BASE 値を得ることになる．b_i は整数値であり，POOL 配列の各要素より基本的に大きいため，POOL に保存された BASE 値は POOL の複数の要素を利用して保存される．

$$\textsc{Base}(s) = \begin{cases} \text{BASE}[s] & (\text{LABEL}[s] = 0) \\ \text{POOL}[\text{BASE}[s], \text{BASE}[s] + W - 1] & (\text{otherwise}) \end{cases} \quad (2)$$

関数 $\textsc{Goto}(s, c)$ は式 (1) と同義の処理を関数 $\textsc{Base}(s)$ を用いて記述したものであり，遷移に成功した場合は遷移先 t を返し，失敗した場合は False を返す．

Algorithm 1 パトリシア–ダブル配列の検索アルゴリズム

1: **function** $\textsc{Goto}(s, c)$
2: 　$t \leftarrow \textsc{Base}(s) + c$
3: 　**if** CHECK$[t] \neq s$ **then**
4: 　　**return** False
5: 　**return** t
6:
7: **function** $\textsc{Search}(key)$
8: 　$s \leftarrow 1$
9: 　$keypos \leftarrow 1$
10: 　**while** $keypos \leq |key|$ and LEAF$[s] = 0$ **do**
11: 　　$t \leftarrow \textsc{Goto}(s, key[keypos])$
12: 　　**if** t = False **then**
13: 　　　**return** False
14: 　　**if** LABEL$[t] = 1$ **then**
15: 　　　$u \leftarrow$ BASE$[t] + W$
16: 　　　**while** POOL$[u] \neq$ ‘$’ **do**
17: 　　　　**if** $keypos > |key|$ or
18: 　　　　　POOL$[u] \neq key[keypos]$ **then**
19: 　　　　　**return** False
20: 　　　　$keypos \leftarrow keypos + 1$
21: 　　　　$u \leftarrow u + 1$
22: 　　**else**
23: 　　　$keypos \leftarrow keypos + 1$
24: 　　$s \leftarrow t$
25: 　**if** $keypos = |key|$ **then**
26: 　　**if** $\textsc{Goto}(s,$ ‘$’$)$ = False **then**
27: 　　　**return** False
28: 　**else**
29: 　　$u \leftarrow$ BASE$[s]$
30: 　　**while** $keypos \leq |key|$ **do**
31: 　　　**if** POOL$[u] \neq key[keypos] + 1$ **then**
32: 　　　　**return** False
33: 　　　$u \leftarrow u + 1$
34: 　　**if** POOL$[u] \neq$ ‘$’ **then**
35: 　　　**return** False
36: 　**return** True

[*4]　$w = 32$，$\lceil \log_2 \sigma \rceil = 8$ を想定している．

3.4　空間効率の良いキーの追加処理

提案手法のキーの追加処理を Algorithm 2，3 に示す．関数 Insert(key) では，キー key の追加に際してまず key による検索を行い，検索失敗箇所がダブル配列上の遷移や POOL 内の接尾辞であった場合には，従来手法で用いた InsertInBc と InsertInTail と同じ処理でキーを追加する．ただし POOL 内の内部遷移文字列上で検索に失敗した場合，関数 InsertInInternalLabel によりキーが追加される．ここで，分岐が発生する POOL 上の内部遷移ラベル l について，l 上の分岐位置より左側の文字列を L，分岐位置を含めた右側の文字列を R とすると，L か R のいずれかと BASE 値の対を POOL の末尾に追加することになる．この際に，L と R の短い方のラベルを選択して再配置することで，POOL の成長を最小限に留める．関数 LeastPreffix($poolhead, poolpos$) は POOL 内での L の先頭位置（$poolhead$）と R の先頭位置（$poolpos$）を入力とし，$|L| < |R|$ の場合に True を返す関数である．関数 LeastPreffix を用いて関数 InsertInInternalLabel で L か R の短い方のラベルを選択して再配置した後，key の残りの接尾辞の挿入が行われる．関数 LeastPreffix における比較回数は $\min(|L|, |R|)$ 回である．

分岐位置に基づいたラベルの再配置を例を用いて説明する．内部遷移ラベル “compari” 上で，“command” か “complete” による分岐が行われる場合の部分グラフと POOL 配列の変化を図 4 に示す．図 3 と同様に BASE 値の表現に必要な POOL の要素数 W を 4 としている．また，分岐前の頂点 Q_1 の BASE 値を b_{bef}，分岐後の頂点 Q_1 の BASE 値を b_{new} としている．それぞれの遷移ラベルの先頭文字はダブル配列上の遷移として保存されるため，POOL 配列上に表現する必要はない．“command” を追加する場合，“compari” 上の 4 番目の文字で分岐することになる．この場合，左側の文字列 “com” が右側の文字列 “pari” より短いため，左側を選択して $\{b_{bef}, \text{“om”}, \text{‘\$’}\}$ を POOL の末尾に追加した後，接尾辞 “and$” を追加する．“complete” を追加する場合，“compari” 上の 5 番目の文字で分岐することになる．この場合は右側の文字列 “ari” を選択して $\{b_{new}, \text{“ri”}, \text{‘\$’}\}$ を POOL の末尾に追加した後，接尾辞 “ete$” を追加する．

Algorithm 2 パトリシア–ダブル配列のキー追加アルゴリズム

```
 1: function Insert(key)
 2:     lines (8–12) of Algorithm 1
 3:             InsertInBc(s, key[keypos])
 4:     lines (14–18) of Algorithm 1
 5:                     InsertInInternalLabel(s, u, key[keypos])
 6:     lines (20–26) of Algorithm 1
 7:             InsertInBc(s, Empty)
 8:     lines (28–31) of Algorithm 1
 9:                 InsertInTail(s, u, key[keypos])
10:     lines (33–34) of Algorithm 1
11:             InsertInTail(s, u, Empty)
12:     line (36) of Algorithm 1
```

3.5　提案手法の記憶量

BASE と CHECK の整数値の表現にそれぞれ w ビットを要しているため，各要素 $2w$ ビットを有する．LABEL と LEAF の表現には，BASE に保存されている値の上位 2 ビットの領域を間借りすることで，記憶量を追加するこ

Algorithm 3 パトリシア–ダブル配列の内部遷移ラベルにおける枝分岐アルゴリズム

```
 1: function LeastPreffix(poolhead, poolpos)
 2:     i ← 0
 3:     while POOL[poolpos + i] ≠ ‘$’ do
 4:         if i ≥ poolpos − poolhead then
 5:             return True
 6:         i ← i + 1
 7:     return False
 8:
 9: function InsertInInternalLabel(s, poolpos, suffix)
10:     u ← BASE[s] + W
11:     beforeB ← Base(s)
12:     targetC ← POOL[poolpos]
13:     newB ← FindBase({targetC, suffix[1]})
14:     t ← newB + targetC
15:     C ← Children(s)
16:     if LeastPrefix(u, poolpos) then
17:         if poolpos > u then
18:             l ← |POOL| + 1
19:             POOL の末尾に {newB, POOL[u, poolpos − 1], ‘$’}
    を追加
20:             BASE[s] ← l
21:         else
22:             LABEL[s] ← 0
23:             Base(s) ← newB
24:         if POOL[poolpos + 1] ≠ ‘$’ then
25:             BASE[t] ← poolpos + 1 − W
26:             LABEL[t] ← 1
27:         else
28:             LABEL[t] ← 0
29:         Base(t) ← beforeB
30:     else
31:         if POOL[poolpos + 1] ≠ ‘$’ then
32:             l ← |POOL| + 1
33:             POOL の末尾に {beforeB, POOL[poolpos + 1] から
    ‘$’ までの文字列 } を追加
34:             BASE[t] ← l
35:             LABEL[t] ← 1
36:         else
37:             LABEL[t] ← 0
38:             Base(t) ← beforeB
39:         POOL[poolpos] ← ‘$’
40:         Base(s) ← newB
41:     InsertEdge(s, targetC)
42:     for c ∈ C do
43:         CHECK[beforeB + c] ← t
44:     InsertInBc(s, suffix[2, . . .])
```

© 2020 Information Processing Society of Japan

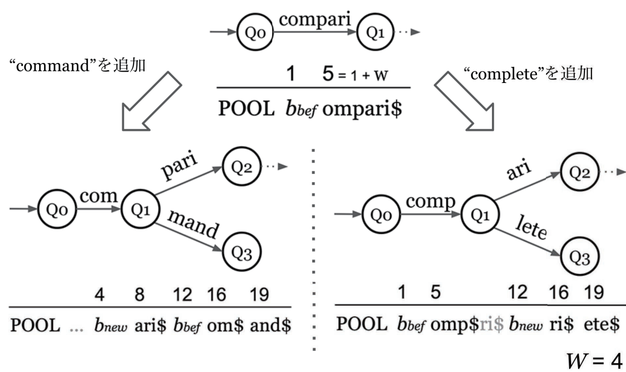


図 4　内部遷移ラベル “compari” 上で，“command” か “complete” による分岐が行われる場合の部分グラフと POOL 配列の変化

Fig. 4　Changes of internal tree and POOL array when forking at the internal transition label “compari” by “command” or “complete”.

となく表現できる．ただし，BASE 値の表現に必要なビット数が $w-2$ ビット以下の場合でのみ利用できる技法であることには注意しなければならない．POOL 配列の各要素は $\lceil \log_2 \sigma \rceil$ ビットを有する．提案手法による記憶量の変化について，従来の最小接頭辞トライにおいて頂点集合 $\{s_1, s_2, \ldots, s_{m+1}\}$ を文字列 $\{c_1, c_2, \ldots, c_m\}$ から順に 1 文字ずつ得られる文字の枝によって連結していた頂点が，パトリシアトライ表現によって頂点 s_1 から s_{m+1} に文字列 $\{c_1 c_2 \ldots c_m\}$ による遷移で表現できるようになった場合，削減される頂点数は $m-1$ 個である．POOL 配列へのアドレスの表現に w ビットを要するため，上記の場合には提案手法により $(m-1)(2w-\lceil \log_2 \sigma \rceil) + w - \lceil \log_2 \sigma \rceil$ ビット削減されることになる．これは，$w > \log \sigma$ の仮定の上では，$m \geq 2$ の場合においてダブル配列の有効な要素の記憶量が削減できることを意味する．

4.　実験による評価

本章では，実験により提案したパトリシア–ダブル配列のキーの追加時間，消費メモリ，検索時間を評価する．

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，32GB RAM であり，OS は CentOS7.3 である．実装言語は C++であり，最適化オプションO3 を指定して GCC 8.3.1 20190311（Red Hat 8.3.1-3）を用いてコンパイルした．実験に用いたデータセットは，Titles-enwiki：英語 Wikipedia タイトル集合（ファイルサイズ：227.20 MiB，キー数：11,414,967，平均長：20.8）*5，URIs-LUBM-DS5：LUBM ベンチマークにより生成されたデータセットから抽出した URI 集合（ファイルサイズ：3,194.1 MiB，キー数：52,616,588，平均長：63.7）*6，の 2 つである．

実験ではデータセットのキーをランダム順に追加し辞書をオンラインで構築し，10 万キーの追加ごとにメモリ消費量と構築時間を計測した．検索時間はそれぞれの時点での辞書に対して挿入済みのキーからランダム順に 10 万キーを検索した際の総検索時間を計測した．実行時間の測定には `std::chrono::duration_cast` を用いた．メモリ消費量の計測には `/proc/self/statm` に記録される実メモリ上のメモリ使用量を用いた．

*5　https://dumps.wikimedia.org/enwiki/

*6　https://exascale.info/projects/web-of-data-uri/

4.2　キーワード辞書の実装

提案手法を C++を用いて実装した*1．比較手法について，従来手法である最小接頭辞ダブル配列の実装として Yoshinaga ら [12] の Cedar *7を用いた．ただし Cedar では配列の成長にともなうリアロケーションの実装がカスタマイズされているため，従来手法との比較で議論し辛い点がある．そのため，我々も最小接頭辞トライによる動的辞書を実装し，Cedar と合わせて比較する．Cedar では BASE/CHECK の要素サイズはそれぞれ 32 ビットとなっており，基準を揃えるため我々も $w=32$ で実装した．ビット列 LEAF と LABEL の表現には配列 BASE の各要素の上位 2 ビットを割り当てた．この場合，BASE/CHECK 配列長か POOL 配列長の最大長 n が $n \leq 2^{30} \approx 10^9$ の範囲で表現できる辞書の大きさであれば問題なく動作する．また，文字の表現に用いるビット数は 8 bit とした．マルチバイト文字は 8 bit ＝ 1 Byte ごとの文字列として保存される．BASE 値の表現に要する POOL 配列の要素数は $W = \frac{32}{8} = 4$ である．

4.3　実験結果

比較手法を以下のように表す．

- `MPDA(Cedar)`：Yoshinaga らの最小接頭辞ダブル配列の実装
- `MPDA(our code)`：我々の最小接頭辞ダブル配列の実装
- `PDA(proposed)`：提案手法のパトリシアダブル配列

すべてのキーを辞書へ挿入した時点での，BASE/CHECK 配列と POOL 配列の配列長およびそれぞれの有効要素数を**表 2** に示す．また，キー挿入の過程の結果を**図 5**，**図 6**，**図 7** に示す．それぞれの図の横軸は追加されたキー数を表している．図 5 は縦軸にメモリ消費量を表している．図 6 は縦軸に 10 万キーの追加に要した時間を表している．図 7 は縦軸に 10 万キーを追加するたびに構成された辞書の 10 万キーあたりの検索時間を表している．

4.3.1　構築された辞書の詳細

表 2 の有効要素とは実際に値を格納している要素であり，値がない要素は空要素と表現する．ダブル配列の有効要素数

*7　http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/

ⓒ 2020 Information Processing Society of Japan

表 2　実験により構築された辞書の詳細

Table 2　The details of constructed dictionaries.

	BASE/CHECK			POOL		
	配列長	有効要素数	有効要素数/配列長	配列長	有効要素数	有効要素数/配列長
Titles-enwiki						
`MPDA(our code)`	35,559,680	35,559,543	0.999	144,449,714	110,841,822	0.767
`PDA(proposed)`	28,035,584	22,151,270	0.790	178,103,328	137,323,059	0.771
URIs-LUBM-DS5						
`MPDA(our code)`	96,252,416	75,934,829	0.789	247,740,551	171,805,725	0.693
`PDA (proposed)`	89,681,664	60,319,246	0.673	272,461,852	193,393,952	0.710

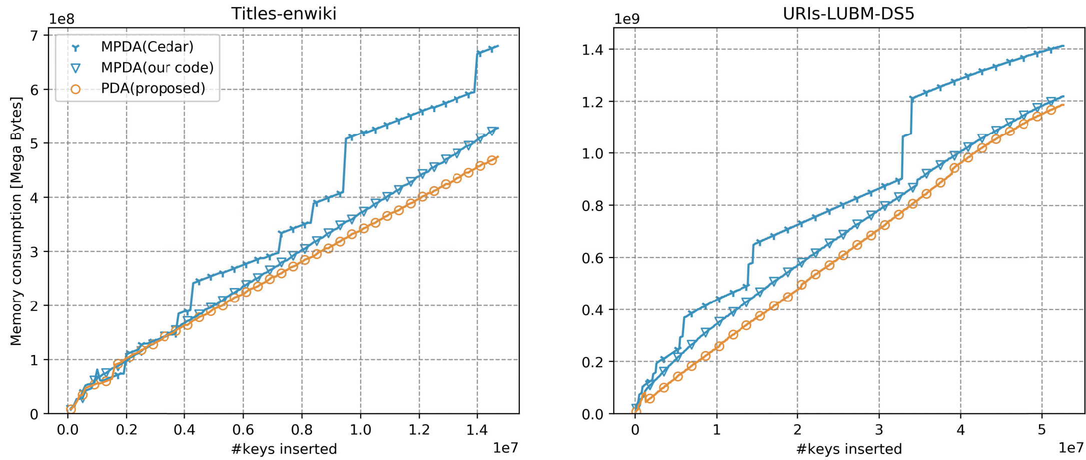


図 5　メモリ消費量の実験結果

Fig. 5　Experimental results of memory consumption.

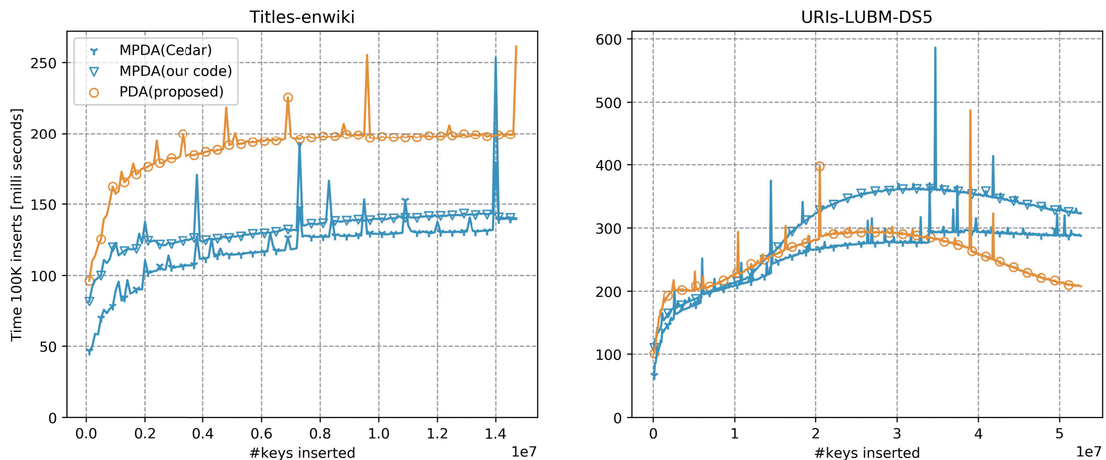


図 6　キーの追加時間の実験結果

Fig. 6　Experimental results of key insertion.

はトライの頂点数に対応しているため，BASE/CHECK 配列の有効要素数を比較することで最小接頭辞トライとパトリシアトライの頂点数の変化を確認できる．BASE/CHECK の有効要素数は Titles-enwiki においては約 0.62 倍に削減され，URIs-LUBM-DS5 では約 0.79 倍に削減されており，パトリシアトライとして表現することによる頂点数の削減が確認できる．一方で BASE/CHECK 全体の配列長は Titles-enwiki では約 0.79 倍，URIs-LUBM-DS5 では約 0.93 倍になっている．また，有効要素の割合が Titles-enwiki では約 0.79 倍，URIs-LUBM-DS5 では約 0.85 倍と減少しており，提案手法は確保したメモリ領域を十分に利用しきれていないことが分かる．提案手法の空要素が多い原因として，最小接頭辞ダブル配列では出次数が 1 の頂点が頻出するため，空要素を比較的効率的に利用できるという特徴があるが，パトリシアトライはすべての頂点が複数の分岐を持つため，点在する空要素を効率的に利用しにくいという点があげられる．POOL 配列については，提案手法では POOL 内で表現する文字列が増加するため配列長の増加が

© 2020 Information Processing Society of Japan

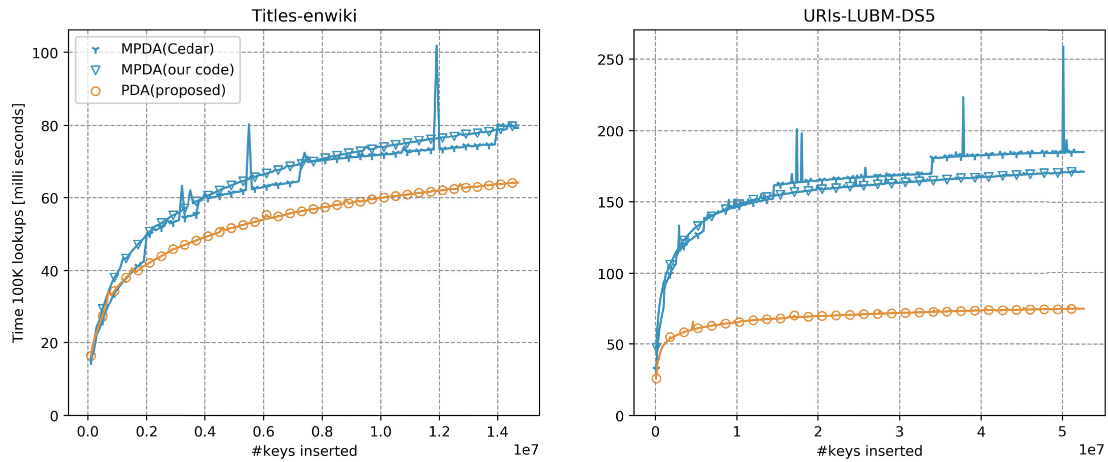


図 7　キーの検索時間の実験結果

Fig. 7　Experimental results of key lookup.

表 3　既存の動的キーワード辞書との比較実験結果

Table 3　The comparisons of experimental results by proposed and existing dynamic keyword dictionaries.

	Titles-enwiki			URIs-LUBM-DS5		
	メモリ消費量 [MByte]	構築時間 [sec]	平均検索時間 [micro sec/key]	メモリ消費量 [MByte]	構築時間 [sec]	平均検索時間 [micro sec/key]
HT	428.52	**12.3**	0.103	2,216.89	104.1	0.131
PDT-PFK	734.64	15.7	**0.094**	2,654.36	**73.2**	**0.060**
PDT-CFK	**259.39**	25.5	0.159	**690.52**	110.2	0.095
MPDA(Cedar)	679.91	17.5	0.273	1,413.39	134.7	0.319
MPDA(our code)	531.02	19.6	0.267	1,218.00	159.0	0.289
PDA(proposed)	474.53	28.0	0.217	1,187.59	132.4	0.131

確認できるが，有効要素の割合についてはいずれのデータセットでも大きな変化は確認できない．

4.3.2　メモリ消費量

まず，MPDA(Cedar) は MPDA(our code) に比べて比較的余裕を持ったメモリ領域を確保していることが確認できる．MPDA(our code) と PDA(proposed) の比較では，Titles-enwiki では 400 万キーの追加時以降で，URIs-LUBM-DS5 ではほぼ全体を通して提案手法が省メモリで実行されていることが確認できる．すべてのキーを追加した時点でのメモリ消費量は，PDA は MPDA に対し，Titles-enwiki では約 0.89 倍，URIs-LUBM-DS5 では約 0.98 倍である．

4.3.3　キーの追加時間

図 6 においてたびたび現れる大きな値は，BASE/CHECK や POOL の成長にともなう配列のリアロケーションが行われている箇所である．しかしこれは全体の追加時間に大きく影響する値ではないため，本項では無視して議論する．MPDA(Cedar) はメモリ領域を余裕を持って確保していることから，いずれのデータセットでも我々の実装よりも高速になっている．Titles-enwiki においては，PDA は MPDA に対し約 1.43 倍遅くなっている．URIs-LUBM-DS5 では PDA は 1,400 万キー挿入以降は MPDA(our code) より高速で，3,400 万キー挿入以降は MPDA(Cedar) より高速である．URIs-LUBM-DS5 で PDA が高速な理由は，挿入される頂点が減ることで FindBase の呼び出し回数が抑えられることが大きい．一方で Titles-enwiki で遅速な理由は，BASE/CHECK の有効要素率が低いことにより FindBase の計算量が増加していることが原因 [19] であり，頂点数の減少以上に構築時間に不利な影響を与えていると考えられる．

4.3.4　キーの検索時間

いずれのデータセットでも，提案手法が従来手法の実装より高速な検索が実現できている．これには，提案手法がキーの大多数を単純な文字列で表現し，ダブル配列の遷移回数が削減されることで検索速度が改善したと考えられる．すべてのキーを追加した状態で PDA(proposed) は MPDA(our code) に対し，Titles-enwiki では約 0.81 倍，URIs-LUBM-DS5 では約 0.45 倍の時間で検索が実行できており，特に URI 集合における検索高速化に大きく貢献している．

4.4　既存の動的キーワード辞書との比較

前節での実験で用いた辞書に加え以下の動的キーワード辞書の実装を比較対象とし，メモリ消費量，総構築時間，1 キーあたりの平均検索時間を実験し，結果を表 3 に示す．

© 2020 Information Processing Society of Japan

- `HT`：HAT-trie [8] の Tessil による実装*8．HAT-trie では分岐数が一定以上の頂点からの遷移集合を単純なテーブルに書き出す Burst と呼ばれる処理を動的に行い，特に根付近の検索を効率的に行える．
- `PDT-PFK`：Dynamic Path-Decomposed Trie（DynPDT）[9], [10] の Kanda による実装*9．Path Decomposition [16] を用いた動的な辞書であり，トライ構造を単純なハッシュテーブルで表現している．
- `PDT-CFK`：DynPDT のトライ構造をコンパクトハッシュテーブル [17] で実装し，加えてラベルのポインタをコンパクトに表現したバージョン．

4.4.1 提案手法の既存の動的キーワード辞書に対する評価

Titles-enwiki において，メモリ消費量は `PDT-PFK` より少ないものの，構築時間と検索時間で `HT,PDT-(PFK,CFK)` に劣っている．URIs-LUBM-DS5 においては，メモリ消費量は `PDT-CFK` に次いで 2 番目に少なく，検索時間は `PDT-PFK` に次いで `HT` と同程度に少ない．構築時間は `HT,PDT` より大きい．提案手法は URI 集合の表現において，`PDT-CFK` ほどではないものの，空間使用量に対して検索効率に優れた手法であるといえる．

5. おわりに

本稿では，動的キーワード辞書の設計として，ダブル配列によるパトリシアトライ表現を用いた実装方法を提案した．現在のダブル配列にすべての遷移ラベルに文字列を用いる表現方法がないという課題に対して，伊藤のアイデアを基に遷移ラベルとダブル配列の BASE 値を連続して保存する解決方法を提案し，加えて記憶量の無駄の少ない枝の分岐アルゴリズムを提案した．また，提案手法を動的キーワード辞書ライブラリとして公開した*1．実社会で用いられるコーパスを用いた実験では，メモリ消費量と検索時間を改善し，特に検索時間に大きな改善が見られた．構築時間に関しては，最小接頭辞トライからパトリシアトライへ変形することの課題として，提案手法の BASE/CHECK 配列に空要素が増加しやすいことにより，空間効率の悪化および構築時の計算量が増加することが分かった．

提案手法の空要素が多いという問題に関する改善案の 1 つとして，配列の後方の要素の前方への再配置を再帰的に行うことで空要素を削減する手法 [7], [21], [22] が存在するが，この手法は空要素率が高い場合には実行時間が大きいことが分かっている．提案手法のようにキーの追加を繰り返すだけで空要素が増加する場合では，再配置を実行しすぎるとキーの追加時間が増加し，再配置の実行が遅すぎると空要素が増えすぎるというジレンマが起こる．そのため再配置を行う適切なタイミングについて結論付けるのは難しく，基本的に空間効率と更新時間がトレードオフの関係となる．一方で，ダブル配列の全体を再構築することで空要素をほとんど削減できることが分かっている [23]．また，再構築時にトライの節を深さ優先順に選んで構築することで，検索時に確認する要素のキャッシュヒット率が高まり，検索時間の改善も期待できる．Kanda らは，空要素の割合が 1–2 割を超える場合には再構築が再配置より高速に行えることを実験により示している．このことから，提案手法に対して，辞書サイズの成長が一定スケールを超えるたびに再構築を行うなどの工夫が考えられる．

今後は，再構築の適応などの工夫により提案手法の記憶量，追加時間および検索時間の改善を行い，他の動的辞書に対する優位性を示すことにより，動的辞書への高速な検索が求められる様々なシステムへのパトリシア–ダブル配列の応用を検討したい．

*8 https://github.com/Tessil/hat-trie
*9 https://github.com/kampersanda/poplar-trie

参考文献

[1] Grossi, R. and Ottaviano, G.: Fast Compressed Tries through Path Decompositions, *Journal of Experimental Algorithmics*, Vol.19, No.1, pp.1.1–1.20 (online), DOI: 10.1145/2656332 (2015).

[2] Martínez-Prieto, M.A., Brisaboa, N., Cánovas, R., Claude, F. and Navarro, G.: Practical compressed string dictionaries, *Information Systems*, Vol.56, pp.73–108 (online), DOI: 10.1016/j.is.2015.08.008 (2016).

[3] 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 (online), DOI: 10.1007/s10115-016-0999-8 (2017).

[4] Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K. and Pavlo, A.: SuRF: Practical Range Query Filtering with Fast Succinct Tries, *Proc. 2018 International Conference on Management of Data, SIGMOD '18*, pp.323–336, ACM Press (online), DOI: 10.1145/3183713.3196931 (2018).

[5] 松本拓真，神田峻介，森田和宏，泓田正雄：ダブル配列オートマトンによる圧縮文字列辞書の実装，情報処理学会技術報告 (2018).

[6] Baskins, D.: A 10-MINUTE DESCRIPTION OF HOW JUDY ARRAYS WORK AND WHY THEY ARE SO FAST (2002).

[7] Yata, S., Oono, M., Morita, K., Fuketa, M. and Aoe, J.-I.: An efficient deletion method for a minimal prefix double array, *Software - Practice and Experience*, Vol.37, No.5, pp.523–534 (online), DOI: 10.1002/spe.778 (2009).

[8] Askitis, N. and Sinha, R.: Engineering scalable, cache and space efficient tries for strings, *VLDB Journal*, Vol.19, No.5, pp.633–660 (online), DOI: 10.1007/s00778-010-0183-9 (2010).

[9] Kanda, S., Morita, K. and Fuketa, M.: Practical Implementation of Space-Efficient Dynamic Keyword Dictionaries, Springer, Cham, pp.221–233 (online), DOI: 10.1007/978-3-319-67428-5_19 (2017).

[10] Kanda, S., Köppl, D., Tabei, Y., Morita, K. and Fuketa, M.: Dynamic Path-Decomposed Tries (2019) (online), available from ⟨http://arxiv.org/abs/1906.06015⟩.

[11] Tsuruta, K., Köppl, D., Kanda, S., Nakashima, Y., Inenaga, S., Bannai, H. and Takeda, M.: Dynamic

Packed Compact Tries Revisited (2019) (online), available from ⟨http://arxiv.org/abs/1904.07467⟩.

[12] Yoshinaga, N. and Kitsuregawa, M.: A Self-adaptive Classifier for Efficient Text-stream Processing, *Proc. COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, Dublin, Ireland, Dublin City University and Association for Computational Linguistics, pp.1091–1102 (2014) (online), available from ⟨https://www.aclweb.org/anthology/C14-1103⟩.

[13] Aoe, J.I.: An Efficient Digital Search Algorithm by Using a Double-Array Structure, *IEEE Trans. Software Engineering*, Vol.15, No.9, pp.1066–1077 (online), DOI: 10.1109/32.31365 (1989).

[14] 伊東秀夫：辞書検索に用いる有限オートマトンの構成と実装，言語処理学会，Vol.4, pp.47–50 (1998).

[15] Tarjan, R.E. and Yao, A.C.-C.: Storing a Sparse Table, *Comm. ACM*, Vol.22, chapter 11, pp.606–611 (1979).

[16] Ferragina, P., Grossi, R., Gupta, A., Shah, R. and Vitter, J.S.: On searching compressed string collections cache-obliviously, *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.181–190 (online), DOI: 10.1145/1376916.1376943 (2008).

[17] Knuth, D.E.: *The Art of Computer Programming, Volume 3:* (*2nd Ed.*) *Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998).

[18] Fredkin, E.: Trie Memory, *Comm. ACM*, Vol.3, No.9, pp.490–499 (online), DOI: 10.1145/367390.367400 (1960).

[19] Morita, K., Fuketa, M., Yamakawa, Y. and Aoe, J.I.: Fast insertion methods of a double-array structure, *Software: Practice and Experience*, Vol.31, No.1, pp.43–65 (online), DOI: 10.1002/1097-024X(200101)31:1 <43::AID-SPE356>3.0.CO;2-R (2001).

[20] Revuz, D.: Dictionnaires et lexiques. Méthodes et algorithmes, PhD Thesis, Paris 7 (1991).

[21] Oono, M., Fuketa, M., Morita, K., Kashiji, S. and Aoe, J.I.: An improvement key deletion method for double-array structure using single-nodes, *Information Processing and Management*, Vol.40, No.1, pp.47–63 (online), DOI: 10.1016/S0306-4573(02)00090-0 (2004).

[22] Morita, K., Tanaka, A., Fuketa, M. and Aoe, J.: Implementation of update algorithms for a double-array structure, *2001 IEEE International Conference on Systems, Man and Cybernetics, e-Systems and e-Man for Cybernetics in Cyberspace* (*Cat.No.01CH37236*), Vol.1, pp.494–499, IEEE (online), DOI: 10.1109/ICSMC.2001.969862 (2001).

[23] Kanda, S., Fujita, Y., Morita, K. and Fuketa, M.: Practical rearrangement methods for dynamic double-array dictionaries, *Software: Practice and Experience*, Vol.48, No.1, pp.65–83 (online), DOI: 10.1002/spe.2516 (2018).

松本 拓真　（学生会員）

徳島大学大学院先端技術科学教育部博士前期課程在学．アルゴリズムとデータ構造の研究に従事．2018 年 WebDB Forum 学生奨励賞受賞．

森田 和宏　（正会員）

徳島大学大学院ソシオテクノサイエンス研究部准教授．2000 年徳島大学大学院工学研究科博士後期課程修了．博士（工学）．2000～2006 年徳島大学工学部助手，2006～2014 年同大学大学院ソシオテクノサイエンス研究部講師を経て，現職に至る．主に情報検索，自然言語処理の研究に従事．

泓田 正雄　（正会員）

徳島大学大学院ソシオテクノサイエンス研究部教授．1998 年徳島大学大学院工学研究科博士後期課程修了．博士（工学）．1998～2000 年徳島大学工学部助手，2000～2006 年同大学工学部講師，2006–2015 年同大学工学部准教授を経て，現職に至る．主に情報検索，自然言語処理，高齢者支援の研究に従事．

（担当編集委員　田中 剛）

© 2020 Information Processing Society of Japan