メモリ効率の良い動的Trie辞書の設計と実装

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　Trie と呼ばれる木構造を用いて，キーとなる文字列の集合を保存し，各キーとその連想値への写像を実現するデータ構造を Trie 辞書と呼ぶ．本稿では，Dynamic Path-Decomposed Trie（DynPDT）を用いたメモリ効率の良い動的 Trie 辞書の設計と実装について述べる．DynPDT はキャッシュフレンドリな Trie 辞書を構築する技法 Path Decomposition を動的辞書の構築に応用したデータ構造であり，開番地法を用いた動的 Trie 表現により効率的に辞書を実現する．一般に開番地法で問題となるのが，ハッシュ表の成長である．動的 Trie 表現においては，一度 Trie を走査する必要があり，愚直にこれを実行した場合にハッシュ表の成長が大きなボトルネックとなる．そのため，DynPDT の既存の評価実験は，予め適切なハッシュ表のサイズが見積もられた状態でしか行われておらず，実用的ではない．そこで，節点数に対し線形期待時間でハッシュ表の成長が行える効率的なアルゴリズムを示し，実験により評価を与える．結果として，キーの追加に伴いハッシュ表の成長を実行したとしても，DynPDT 辞書は高い性能を示した．

キーワード　Trie 辞書，動的データ構造

1. はじめに

現代の計算機科学において，大規模な文字列データをいかに効率良く管理するかというのは基本的な課題の一つである．文字列処理を支えるデータ構造は数多く存在するが，本稿では自然言語処理や情報検索などで古くから用いられるキーワード辞書の効率的な設計と実装について論じる．

キーワード辞書とは文字列の集合を主記憶で保存し検索するためのデータ構造であり，多くの用途ではキーとなる文字列とその連想値のペアを記憶する．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 Trie，Front Coding，全文索引など，さまざまな技法をベースとしたデータ構造が提案されている．とりわけ近年，Web グラフや RDF ストアをはじめとする大規模な辞書を扱う用途が報告されており，簡潔データ構造や文字列圧縮を応用したコンパクトかつ高速なキーワード辞書の実装が数多く提案されている [1–3]．しかし，これらは静的なキーワード辞書の実装であり，予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許容しないデータ構造である．

現状として，大規模なデータを扱う多くのシステムが静的辞書で運用される．一方で，動的辞書により大規模なデータを扱うシステムが存在することも事実である．例えば，効率的な RDF ストアである dipLODocus$_{[\mathrm{RDF}]}$ [4] では，動的辞書を用いて大量の URI をオンラインで符号化しており，辞書の性能がクエリの応答速度や全体のメモリ使用量などに大きく影響する．Mavlyutov ら [5] は既存の動的キーワード辞書を比較し，どのデータ構造が URI を保存するのに適しているかを実験により評価している．

効率の良い動的キーワード辞書の実装の多くが Trie [6, 7] とよばれる木構造をベースとしており，Judy [8] や HAT-trie [9]，Cedar [10] などが存在するが，とりわけメモリ効率の良い実装として Dynamic Path-Decomposed Trie（DynPDT）[11] が提案されている．DynPDT はキャッシュフレンドリな Trie 辞書を構築する技法 Path Decomposition [12] を動的 Trie 辞書の構築に応用したデータ構造である．開番地法を用いた Hash Trie を適用することで，その Trie 構造は効率的に表現される．とりわけ，Poyias ら [13] による Hash Trie の簡潔な実装を用いれば，DynPDT を省メモリで実現できる．

一方，DynPDT が抱える問題として，キーの追加に伴う効率的な辞書の成長方法が提示されていない点があげられる．開番地法を用いた Hash Trie では，ハッシュ表の占有率が 100%に近づいた場合に，木を走査することでより大きなハッシュ表へ節点を再配置し成長する必要があるが，愚直にこれを実行すれば膨大な時間を要する．そのため，DynPDT の既存の評価実験 [11] では，予め適切なハッシュ表のサイズが見積もられた状態でしか行われておらず，成長を考慮していない．しかし，これではストリームデータなど未知のデータに対して DynPDT を応用することは難しく汎用的ではない．

そこで本稿では，Arroyuelo ら [14] の方法に基づいた効率的な Hash Trie の成長アルゴリズムを示す．このアルゴリズムは，節点数に対して線形期待時間で動作し，高速に Hash Trie を成長させることができる．このアルゴリズムを用いて Hash Trie を成長させた場合の性能を実験により評価し，予め適切なハッシュ表のサイズが見積もれない用途においても，DynPDT 辞書が高い性能を維持することを示す．また，研究成果として，DynPDT を用いた汎用的な動的 Trie 辞書ライブラリ Poplar-trie(注1) を公開している．

2. Dynamic Path-Decomposed Trie

Path Decomposition [12] とは Trie [6, 7] をキャッシュフレ

(注1)：https://github.com/kampersanda/poplar-trie

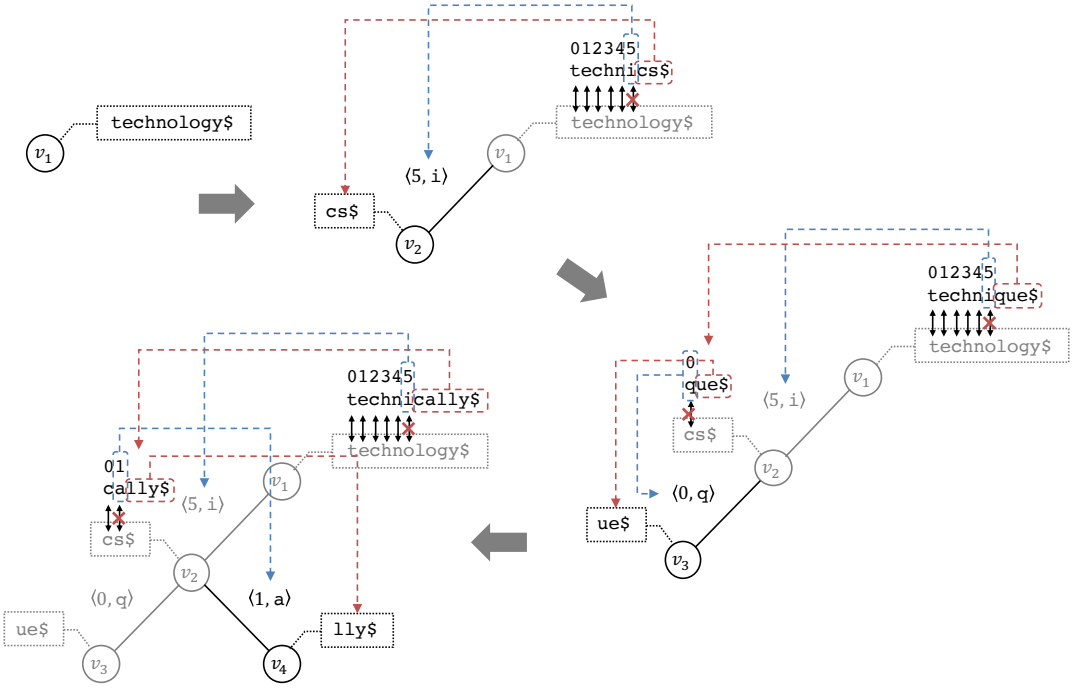

図 1: キー `technology$`，`technics$`，`technique$`，`technically$` をこの順で追加し DynPDT を構築する過程．

ンドリな辞書構造 Path-Decomposed Trie（PDT）に変形するための技法である．Trie をある節点から葉までの経路ごとに分解し，各節点がオリジナルの Trie の各経路に対応するような PDT を構築する．PDT の利点は木の高さが抑制されることであり，検索において節点を遷移する回数を減らしランダムアクセスの回数を抑えることができる．この Path Decomposition を動的辞書の構築に応用したのが DynPDT [11] である．本節では，DynPDT を用いた辞書の設計について解説する．

2.1 基本的なアイデア

DynPDT では終端に特殊文字 `$` が付随したキーを登録する．このようなキーから構築された Trie は葉がそれぞれのキーと一意に対応し，このような Trie に基づく PDT の節点は各キーと一意に対応する．この特徴を利用して，DynPDT は挿入されるキーに対応する節点を逐次的に追加することで構築される動的辞書構造である．節点に文字列のラベルを付随し，枝に整数値と文字のペアを分岐条件として付随する．DynPDT は以下の手順でキーとなる文字列 K を辞書に追加することで構築される．ただし，節点 v に付随するラベルを L_v としている．

- 木が空の場合，新たに根 v を定義し，$L_v \leftarrow K$ とする．
- 木が空でない場合，変数 v に根の ID を設定し，変数 suf に K を設定する．そして，以下の二つの手順によりキー K を探索する．手順 1 では suf と節点ラベル L_v を比較する．もし一致すれば，このキーは既に登録されているので手続きを終了する．一致しなければ次の手順へ進む．手順 2 では，$suf[0..i-1] = L_v[0..i-1]$ かつ $suf[i] \neq L_v[i]$ を満たすオフセット i を用いて，枝ラベル $\langle i, suf[i]\rangle$ により示される子を探索する．もし子がいれば，v にその子の ID を，suf に残りの接尾辞 $suf[i+1..|suf|-1]$ を設定し手順 1 へ戻る．もし子がいなければ，$\langle i, suf[i]\rangle$ をラベルとして付随した枝と残りの接尾辞 $suf[i+1..|suf|-1]$ をラベルとして付随した子を追加し，手続きを終了する．

図 1 に例を示す．この図は，キー `technology$`，`technics$`，`technique$`，`technically$` をこの順で追加し DynPDT を構築する過程を示している．例えば，最後のキー `technically$` を追加する場合，まずキーを根 v_1 のラベル L_{v_1} = `technology$` と比較する．このとき，5 番目の `i` で不一致なので，$\langle 5, \mathtt{i}\rangle$ で子 v_2 へ移動する．続いて，L_{v_2} = `cs$` と残りの接尾辞 `cally$` を比較する．このとき，1 番目の `a` で不一致なので，$\langle 1, \mathtt{a}\rangle$ で子を探索するが存在しないので，それで示される子 v_4 を新たに追加する．残りの接尾辞 `lly$` を節点ラベル L_{v_4} に設定することで，キーを登録できる．

このようにして出来上がる木構造は，各節点が各キーに対応し，節点ラベルが対応するキーの接尾辞を表す．枝ラベルは節点ラベル上でミスマッチが発生したオフセットと遷移文字のペアにより構成される．キーワード辞書を実装する上で，各キーの連想値は節点ラベルに続く領域に埋め込むことで容易に保持できる．

2.2 アルファベットの固定

仮に，キーが $\{0, \ldots, \sigma-1\}$ 上の文字により構成され，最長の節点ラベルの長さが Λ とする．DynPDT の枝ラベルを $\Sigma' = \{0, \ldots, \sigma'-1\}$ 上の文字とすると，$\sigma' = \sigma \cdot \Lambda$ である．DynPDT の問題は，σ' が未知のキー集合に対して不定なことである．大半の Trie 表現技法は定数アルファベットを想定しているため，これらを利用するには σ' を固定する必要がある．そこでパラメータ λ を導入し，アルファベットサイズを $\sigma' = \sigma \cdot \lambda$ と強制的に固定する．もし節点 v から出る子を追加するときに，ミスマッチが発生した 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 で示される子 v' を加える．

（2） $v \leftarrow v'$，$i \leftarrow i - \lambda$ と更新する．

この手順により発生する節点をステップ節点と呼ぶ．ステップ節点の導入により σ' は固定されるが，λ の設定には注意が必要である．仮に λ が小さすぎる場合，大量のステップ節点が追加されることで木のサイズが大きくなるに加え，検索において辿る節点数も増える．逆に λ が大きすぎる場合，σ' が大きくなり，Trie の表現法によっては大きなボトルネックとなる．

2.3 実装方策

DynPDT と普通の Trie との大きな違いは，節点に文字列のラベルを付随していることである．本方策では，節点ラベルを除く Trie 構造の部分は既存の動的 Trie 表現を用いて実装し，節点ラベル文字列へのポインタをその Trie 表現と関連付けて保持することで DynPDT を実装する．このとき，注意すべき点は以下の二点である．

（1） 適した動的 Trie 表現は何か

（2） どのように節点ラベルへのポインタを保持するか

前者については，節 3. で Hash Trie が適していることを示し，Hash Trie のデータ構造，及びそのコンパクトな実装を解説する．また，Hash Trie の既存の問題点としては，その効率的な成長アルゴリズムが確立されていない点である．そこで，節 4. では節点数に対して線形期待時間で動作する成長アルゴリズムを示す．後者については，Google Sparse Hash [15] の技法を応用することでポインタを省メモリで実装できることを，節 5. において示す．

3. Hash Trie

DynPDT の実装に関して，Trie 構造を T とした場合，以下の操作を提供する表現について考える．

- addchild(T, v, c)：文字 c により示される節点 v の子を新たに定義し，その子の節点 ID を返す．
- getchild(T, v, c)：文字 c により示される節点 v の子の節点 ID を返す．なければ -1 を返す．
- getparent(T, v)：節点 v の親の ID を返す．
- getedge(T, v)：節点 v と指し示す枝上の文字を返す．
- getroot(T)：根の ID を返す．

DynPDT では，枝文字のアルファベットサイズは σ' であるが，本節と次節では簡単に σ で表す．また，節点数は t で表す．

この Trie 表現には，いくつかの候補が存在する．単純な方法としては，Array Trie と List Trie の二つがある．Array Trie では，節点 v を長さ σ の配列により表し，文字 c により示される子へのポインタを c 番目の要素に格納する．getchild は定数時間だが，配列は疎であり，各節点に親のポインタも保持することを考慮すると，メモリ使用量は $t(\sigma+1)\log t$ ビットであり，あまりに大きい．List Trie は兄弟節点を単方向リストで表し，親は先頭の子へのポインタを保持する．Array Trie と違って無駄な領域は存在せず，メモリ使用量は $3t\log t + t\log\sigma$ ビットであるが，getchild は $O(\sigma)$ 時間も要する．

より優れた候補としては，Ternary Search Trie（TST）[16] や Double Array Trie（DAT）[17] がある．TST は List Trie にポインタを一つ加えた表現技法であり，getchild を $O(\log\sigma)$ 時間に削減する．しかし，ポインタが一つ加わって，そのメモリ使用量は $4t\log t + t\log\sigma$ ビットであり，少し大きい．DAT は二つの配列により Array Trie と List Trie の利点を両立した技法であり，getchild を定数時間で実行しながら，最良で $2n\log n$ ビットで表現できる．DAT はバイト文字などの σ が高々256 の用途では効率的に実装できるが，σ が大きいと配列が疎になり，メモリ効率が大幅に低下する傾向にある．そのため，DynPDT の実装には用いるのが難しい．

以上の考察の結果，DynPDT の実装にはハッシュ表上で節点を表現する Hash Trie を適用する．Hash Trie は，開番地法により節点を配置することで getchild と addchild を $O(1)$ 期待時間で実行する．ハッシュ表の長さを $m = O(t)$ とすると，$O(t\log(t\sigma))$ ビットで Trie を表現できる．また，Poyias ら [13] の技法を用いれば，$O(t\log\sigma)$ ビットにまでメモリ使用量を削減できる．本節では，Hash Trie の単純な実装とコンパクトな実装を解説する．また，Hash Trie の効率的な成長アルゴリズムは次の節で提案する．

3.1 単純な実装

ここで解説する単純な Hash Trie を PHT と表す．PHT では，長さ $m \geqq t$ のハッシュ表 H とハッシュ関数 $h : \mathbb{N} \to \mathbb{N}$ を用いる．このとき，ハッシュ表の占有率は t/m である．節点はハッシュ表のいずれかの要素に配置され，その番地はハッシュ表が再構成でもされない限り変更されることはない．そのため，この表現ではその番地を節点の ID として用いる．すなわち $H[v]$ に配置された節点の ID は v である．根の番地は任意である．

この H と h を用いて，addchild(v, c) は以下の手順で実現される．まず，子を示すためのハッシュキーとして $k = \langle v, c\rangle$ を生成する．次に，初期番地 $i = h(k) \bmod m$ を算出し，i から線形に走査し最初に出会う空要素の番地を i' としたとき，$H[i'] = k$ として番地 i' 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 i' となる．getchild も addchild と同様の手順で実行され，良いハッシュ関数 h が設計できれば，これらの操作は $O(1)$ 期待時間で実行される．getparent$(i') = v$，getedge$(i') = c$ は，$H[i'] = \langle v, i\rangle$ より簡単に実行できる．一方で H は $m\lceil\log(m\sigma)\rceil$ ビットを要し，漸近的にはポインタベースの表現と変わらない．

3.2 コンパクトな表現

PHT のメモリ使用量は，コンパクトハッシュ法 [6] を用いることで $m\log\sigma + O(m)$ ビットにまで削減できる．本稿では，この実装を CHT と表す．基本的なアイデアは同じだが，CHT では全単射ハッシュ関数 $h : \{0, \ldots, m\sigma - 1\} \to \{0, \ldots, m\sigma - 1\}$ を用いる．すなわち，$h^{-1}(h(k)) = k$ を満たす逆関数 h^{-1} が実現できるようなハッシュ関数である．このような全単射ハッシュ関数を用いて，$H[i']$ には商 $\lfloor h(k)/m\rfloor$ のみを格納する．元のハッシュ値 $h(k)$ は，初期番地 $i = h(k) \bmod m$ と商 $H[i'] = \lfloor h(k)/m\rfloor$ から復元され，h^{-1} によりハッシュキー $k = \langle v, c\rangle$ は復元できる．

この方法で問題となるのが，節点 ID の i' から，それに対応する初期番地 i を知る必要があるということである．この問題

を解決するために，Poyias ら [13] は $D[i'] = (i' - i) \bmod m$ であるような転移配列 D を導入した．D を用いることで，節点 ID i' が与えられたとき，それに対応する初期番地 i は $(i' - D[i']) \bmod m$ により算出できる．

メモリ使用量に関して，H は $m\lceil \log \sigma \rceil$ ビットで表現できる．転移配列 D が格納し得る最大値は $m-1$ であるが，良いハッシュ関数 h が設計できた場合，その平均値は非常に小さく [13, 補題 5]，CDRW 配列 [18] を用いることで $O(m)$ ビットで表されることが示されている．故に，CHT は動的 Trie を $m \log \sigma + O(m)$ ビットで表現できる．

3.2.1 実用的な実装

CDRW 配列を用いた転移配列の表現は $O(m)$ ビットであるが，実装が複雑で実用的ではない．そこで，Poyias ら [13] は代わりの実用的な転移配列の表現を提案している．この表現では，三つの補助データ構造 D_1，D_2，D_3 を用いて，以下のように転移配列 D を表現する．D_1 は各要素に Δ_1 ビットが割り当てられた長さ m の整数値配列であり，まずは D の値を D_1 に格納することから試みる．$D[i] < 2^{\Delta_1} - 1$ であれば，$D_1[i]$ に $D[i]$ をそのまま格納する．もしそうでなければ，$D[i] = 2^{\Delta_1} - 1$ とし，Compact Hash Table で実現される二番目の補助データ構造 D_2 に $D[i]$ の格納を試みる．ここで，Compact Hash Table は $\{0, \ldots, m-1\}$ 上のキーから $\{0, \ldots, \Delta_2 - 1\}$ 上の値への写像を与える．仮に $D[i] - (2^{\Delta_1} - 1) < 2^{\Delta_2}$ であれば，キーと値の組 $\langle i, D[i] - (2^{\Delta_1} - 1) \rangle$ を D_2 に格納する．もし D_1 にも D_2 にも格納できなかった場合には，平衡二分木などにより実現された単純な辞書構造 D_3 によって $D[i]$ を格納する．

前述の通り，全体として D の値は小さく，Δ_1 が小さすぎなければ大半の値を D_1 で表現できる．仮に D_1 に格納できなくても，Compact Hash Table により実現される D_2 はメモリ効率良く値を格納できる．重要なのは，Δ_1 と Δ_2 の設定であるが，Poyias ら [13] は占有率が 80%において，$\Delta_1 = 3$，$\Delta_2 = 7$ の付近を選択するのが良いということを実験により示している．

3.2.2 全単射ハッシュ関数の設計

$h(k) \bmod m$ と $\lfloor h(k)/m \rfloor$ をビットマスクとビットシフトにより高速に実行するために，実用では m には 2 の累乗を設定する．また，σ も 2 の累乗である．そのため，あるビット長 w に対し，全単射ハッシュ関数 $h : \{0, \ldots, 2^w - 1\} \to \{0, \ldots, 2^w - 1\}$ を，SplitMix [19] の技法を用いて実現する．具体的には，二つのハッシュ関数 $h_1(x) = x \oplus \lfloor x/2^a \rfloor$，$h_2(x) = xp \bmod 2^w$ を合成し，$h = h_1 \circ h_2$ として実現する．ここで，a は $\lfloor w/2 \rfloor$ より大きい整数値であり，p は 2^w より小さい素数である．h_1 は Xorshift 乱数生成器 [20] を用いた実装であり，$h_1^{-1}(x) = h_1(x)$ である．h_2 は，$pp^{-1} \bmod = 1$ であるような p^{-1} を用いて，$h_2^{-1} = xp^{-1} \bmod 2^w$ により実現できる．故に，$h^{-1} = h_2^{-1} \circ h_1^{-1}$ が実現でき，h は全単射である．

4. Hash Trie の成長

Hash Trie では開番地法を用いているため，占有率が 1 に近づいたときに性能が大幅に減少する．そのため，ある閾値を設定しておいて，占有率が閾値に到達したときにハッシュ表を成長させるのが一般的な方策である．具体的には，長さ m のハッシュ表 H の占有率が閾値に到達したとき，長さ $2m$ のハッシュ表 H' を作成し，H 内のエントリを H' に再配置する．

この再配置に関して，Hash Trie は以下の問題を有する．Hash Trie では，根を除く全ての節点が自身の親の ID を用いて構築されたハッシュキーを用いることで配置される．すなわち，自身の親の ID が決まらない限り，自分の ID を決定することができない．そのため，根から全ての節点を走査する必要があるが，単純に木を走査すると，Hash Trie は子の列挙に $O(\sigma)$ 期待時間を要するため，木の走査には $O(t\sigma)$ 期待時間を要する．

Poyias ら [13] は Hash Trie を $O(t \log \sigma)$ ビットで実装する手法を提案したが，予め節点数が与えられ，適切なハッシュ表のサイズが見積もられた状態で評価を与えている．Kanda ら [11] も予め適切なハッシュ表のサイズが見積もられた状態で DynPDT を評価している．一方で，Arroyuelo ら [14] は省メモリな LZ78 圧縮のために CHT を用いた LZ-trie の実装を提案しており，その中で $O(t)$ 期待時間で走るハッシュ表の成長アルゴリズムを示している．本節では，それを応用した Hash Trie の成長アルゴリズムを提案する．

Algorithm 1: Hash Trie の成長手続き

```
Input : A trie T with a hash table H of length m
Create an empty trie T' with a hash table of length 2m
Define the root of T'
Create an integer array map and bit array done, of length m
Set map[getroot(T)] to getroot(T')
Set done[getroot(T)] to 1 and the other elements to 0
foreach i ∈ [0, m) do
    if H[i] is empty then
        continue
    v ← i
    path ← an empty string
    while done[v] ≠ 1 do
        Push front getedge(T, v) to path
        v ← getparent(T, v)
    v' ← map[v]
    foreach c ∈ path do
        v ← getchild(T, v, c)
        v' ← addchild(T', v', c)
        done[v] ← 1
        map[v] ← v'
T ← T'
```

4.1 成長アルゴリズム

Hash Trie における再配置の手続きを Algorithm 1 に示す．このアルゴリズムは長さ m のハッシュ表 H を持つ Hash Trie T を受け取り，長さ $2m$ のハッシュ表 H' を持つ T' へと再配置することで Hash Trie を成長させる．

Algorithm 1 では補助データ構造として長さ m の整数値配列 map とビット配列 $done$ を用いる．map は T から T' への節点 ID の写像を示す配列であり，$map[v] = v'$ は $H[v]$ が $H'[v']$

に再配置されたことを示す．$done[v] = 1$ は $H[v]$ の再配置が完了していることを表す．T' の初期状態は根だけから成り，$map[\mathsf{getroot}(T)] = \mathsf{getroot}(T')$，$done[\mathsf{getroot}(T)] = 1$ が設定される．

最初のループではハッシュ表 H を左から右に走査している．この走査の中で T のある節点に出逢えば，その節点から根に向かって遷移し，その経路上の文字を記憶する．そして，それらの文字を用いて経路を下りながら T' の節点を定義する．また，T' の節点を定義する中で map と $done$ の値を更新する．根に向かって遷移するとき，$done[v] = 1$ である節点 v に出逢えば，それよりも祖先の節点は既に再配置されていることがわかるため，T' の節点 $map[v] = v'$ からその経路を下ればよい．このように，map と $done$ を用いることで既に再配置された節点を辿り直すのを回避している．

時間計算量について，行 6 のループにおける H の走査には $O(m) = O(t)$ 時間を要する．行 11 における T の走査は，$done$ の利用により同じ節点を辿り直すことがないので $O(t)$ 時間である．行 15 における節点の再配置も同じく $O(t)$ 期待時間である．すなわち，このアルゴリズムを用いれば Hash Trie を $O(t)$ 期待時間で成長させることができる．

4.2 補助領域

Algorithm 1 では，$done$，map に加えて $path$ を補助データ構造として用いる．$done$ が m ビット，map が $m\lceil\log(2m)\rceil$ ビットを要する．$path$ は T の高さを $height$ とすれば $height \cdot \lceil\log\sigma\rceil$ ビットである．DynPDT は木の高さを抑えたデータ構造であり，実際に $height$ は節点数に対して無視できるほど小さい値であり，$path$ の領域は非常に小さい．簡単のため，以下では $path$ の領域は無視して最大メモリ使用量を見積もる．

一見，map の領域が大きそうだが，map の領域は以下のようにハッシュ表 H の領域と共有できる．T の節点 v の再配置が完了した以降，すなわち，$done[v] = 1$ が設定された以降，$H[v]$ の値は用いられることはない．且つ，$done[v] = 0$ であるときに $map[v]$ には値が設定されていない．そのため，H と map の領域は共有可能である．

PHT の最大メモリ使用量に関して，H は $m\lceil\log(m\sigma)\rceil$ ビット，H' は $2m\lceil\log(2m\sigma)\rceil$ ビットを要する．H の領域を用いて map を実現する場合，$\sigma \geqq 2$ のとき map のメモリ使用量は H より小さく，map のための追加領域は必要としない．仮に Trie の条件分岐が $\{0, 1\}$ であっても $\sigma = 2$ であるため，現実的に map は追加領域なしで実装される．故に，PHT における成長アルゴリズムの補助領域は $done$ の m ビットである．最大メモリ使用量は $3m(\lceil\log(m\sigma)\rceil + 1)$ ビットである．

CHT の最大メモリ使用量に関して，H は $m\lceil\log\sigma\rceil$ ビット，H' は $2m\lceil\log\sigma\rceil$ ビットを要する．新しく定義される長さ $2m$ の転移配列を D' とすると，D_1 は $m\Delta_1$ ビット，D'_1 は $2m\Delta_1$ ビットである．もし $\lceil\log m\rceil < \lceil\log\sigma\rceil + \Delta_1$ であれば，map は完全に H と D_1 で共有できる．そうでなければ，追加で $m(\lceil\log m\rceil - (\lceil\log\sigma\rceil + \Delta_1))$ ビットが必要となる．故に，CHT の最大メモリ使用量は $3m(\lceil\log\sigma\rceil + \Delta_1) + m + m \cdot \max(0, \lceil\log m\rceil - (\lceil\log\sigma\rceil + \Delta_1))$ ビットと，D_2，D_3，D'_2，D'_3 のメモリ使用量の合計である．

5. 節点ラベルの管理

節点ラベル L_v は可変長な文字列であるため，Hash Trie の節点 ID により取得できる配列によりポインタを格納する．Kanda ら [11] はこのポインタの管理に関して，単純な管理とコンパクトな管理の二種類を提案している．

5.1 単純なラベル管理

本稿では，単純な節点ラベルの管理を PLM と表す．PLM では長さ m の配列 P を用いて，$P[v]$ に L_v を示すポインタを格納する．この実装では L_v を定数時間で取得できるが，計算機の語長を w とすると，P に mw ビットを要する．

5.2 コンパクトなラベル管理

コンパクトなラベル管理を CLM と表す．CLM では，PLM におけるポインタのオーバヘッドを削減するため，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．すなわち，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 が定義されていれば $B[i] = 1$ 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 順を維持しながら連結し，グループ g の連結ラベル文字列へのポインタを $P[g]$ に保持することにより，P の長さを $\lceil m/\ell\rceil$ に削減できる．P と B のメモリ使用量は $w\lceil m/\ell\rceil + m$ ビットである．

この P と B を用いて，CLM では L_i を以下のように得る．まず $B[i] = 0$ の場合，L_i は定義されていないことがわかる．$B[i] = 1$ の場合，目的のグループ ID $g = \lfloor i/\ell\rfloor$ から，L_i が含まれている連結ラベル文字列へのポインタ $P[g]$ を取得する．同時に，グループ g に対応するビットチャンク $B_g = B[g\ell..(g+1)\ell - 1]$ を取得する．$j = \sum_{k=0}^{i \bmod \ell} B_g[k]$ とすると，L_i は連結ラベル文字列中の j 番目のラベルであり，先頭から j 番目のラベルまで走査することで L_i が取得できる．

時間計算量に関して，popcount 命令を用いることで語の中で点灯する 1 の数を定数時間で求めることができ，j の算出は $O(\ell/w)$ 時間で実行できる．j 番目のラベルまでの走査は，節点ラベルの長さをそれぞれの先頭に記述することで各ラベルをスキップすることができるため，$O(\ell)$ 時間で L_i に到達できる．$\ell = \Theta(w)$ とすれば，CLM は節点ラベルを $O(w)$ 時間で取得でき，P と B のメモリ使用量は $O(m)$ ビットになる．

6. 実験による評価

本節では，実験により Hash Trie の成長を考慮した場合の DynPDT のメモリ効率，及びキーの追加時間，検索時間を評価する．

6.1 実験設定

実験に用いた計算機の構成は，Intel Xeon E5 @3.5 GHz CPU，32 GB RAM（L2 cache : 256 KB，L3 cache : 12 MB）であり，OS は OS X 10.12 である．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 9.0.0 (clang-900) を用いてコンパイルした．実験用データセットには 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）(注2)を用いた．

実験ではキーをランダム順に追加し辞書をオンラインで構築し，その追加時間と最大メモリ使用量を計測した．検索時間は，構築された辞書に対しランダム順にキーを検索することで計測した．実行時間の計測には `std::chrono::duration_cast` を用いた．作業領域の計測には`/usr/bin/time` コマンドにより得られる maximum resident set size を参照した．

6.2 辞書の実装

DynPDT の実装には Poplar-trie を用いた．Poplar-trie には技法の組み合わせによって何種類かの実装が含まれており，前述した Hash Trie の実装 PHT と CHT，節点ラベルの管理 PLM と CLM をそれぞれ組み合わせた，以下の四種類を評価した．

- poplar::MapPP：PHT と PLM の組み合わせ
- poplar::MapPG：PHT と CLM の組み合わせ
- poplar::MapCP：CHT と PLM の組み合わせ
- poplar::MapCG：CHT と CLM の組み合わせ

Poplar-trie における初期のハッシュ表の長さのデフォルト設定は $m = 2^{16}$ であり，本データセットは 10 回の成長を要する．すなわち，ハッシュ表の長さを $m = 2^{26}$ を設定すれば，成長なしで辞書を構築できる．本実験では，初期のハッシュ表の長さの設定として，これら二つの値を設定し，成長なしで構築される場合と 10 回の成長を要して構築された場合を比較する．[11] を参考にして，λ は 16 を設定した．CLM におけるグループサイズ ℓ は 16 を設定した．占有率の上限の閾値は 0.8 とした．

その他のメモリ効率の良い辞書の実装として以下の三つを比較した．

- Judy：HP 研究所で開発された動的 Trie 辞書 [8]
- HAT-trie：Trie と Array Hash のハイブリッド [9]
- Cedar：Double Array を用いた最小接頭辞 Trie [10]

6.3 実験結果

図 2 に実験結果を示す．それぞれのグラフの横軸は最大メモリ使用量（Resident Set Size，RSS）を表している．左のグラフの縦軸は 1 キー当たりの追加時間，右のグラフの縦軸は 1 キー当たりの検索時間を表している．どの結果も左下ほど良い値を示す．

実験結果より，Hash Trie の成長を実行しても poplar::MapPG と poplar::MapCG は非常に少ないメモリ使用量で構築されることが示された．一度も成長しない場合と比べて，poplar::MapPG は 1.14 倍，poplar::MapCG は 1.27 倍の増加に留まっている．成長を実行した場合でも，Judy と比べて 59%のメモリ使用量で poplar::MapCG は構築でき，非常にメモリ効率が良い．

追加時間は，10 回の成長により poplar::MapPG で 1.46 倍，poplar::MapCG で 1.48 倍に増加しているが，HAT-trie と Cedar との比較でも分かる通り，現実的な時間で成長が行えていることがわかる．時間効率については，poplar::MapPP と poplar::MapCP が非常に高く，poplar::MapPP は成長を実行した場合でも，どのデータ構造より高速に構築されている．

検索時間は成長アルゴリズムとは関係がないため，多少の誤差はあるものの成長には影響されない．結果として，poplar::MapPP は HAT-trie に匹敵するほどの検索性能を示している．

Poplar-trie の総合的な評価として，メモリ効率に優れた実装や時間効率に優れた実装があり，求める性能に応じて辞書の実装を選択することができる．また，Hash Trie の成長が重大な問題となることはないが，成長が少ない方が高い性能を示すことは自明であり，適したハッシュ表の初期サイズを設定することも重要である．

7. おわりに

本稿では，メモリ効率の良い動的キーワード辞書の実装である DynPDT を解説し，また原著論文 [11] で考慮されていなかった Hash Trie の成長アルゴリズムを新たに提案した．現実のデータセットを用いて実験により評価を与えることで，成長を実行した場合でも DynPDT は高いメモリ効率を維持することを示した．また，その DynPDT を用いた効率的な動的辞書ライブラリとして，Poplar-trie を作成し公開した．今後の予定として，大規模なデータを扱うさまざまなシステムへの DynPDT の応用を検討したい．

(注2)：DS5 at `https://exascale.info/projects/web-of-data-uri/`

文　献

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

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

[3] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems*, 51(3):1023–1042, 2017.

[4] Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudré-Mauroux. dipLODocus[RDF] — short and long-tail RDF analytics for massive Webs of data. In *Proceedings of the 10th International Semantic Web Conference (ISWC)*, pages 778–793, 2011.

[5] Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux. A comparison of data structures to manage URIs on the Web of data. In *Proceedings of the 12th European Semantic Web Conference (ESWC)*, pages 137–151, 2015.

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

[7] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[8] Doug Baskins. *A 10-minute description of how Judy arrays work and why they are so fast*, 2002.

[9] Nikolas Askitis and Ranjan Sinha. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660, 2010.

[10] Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 1091–1102, 2014.

[11] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa.

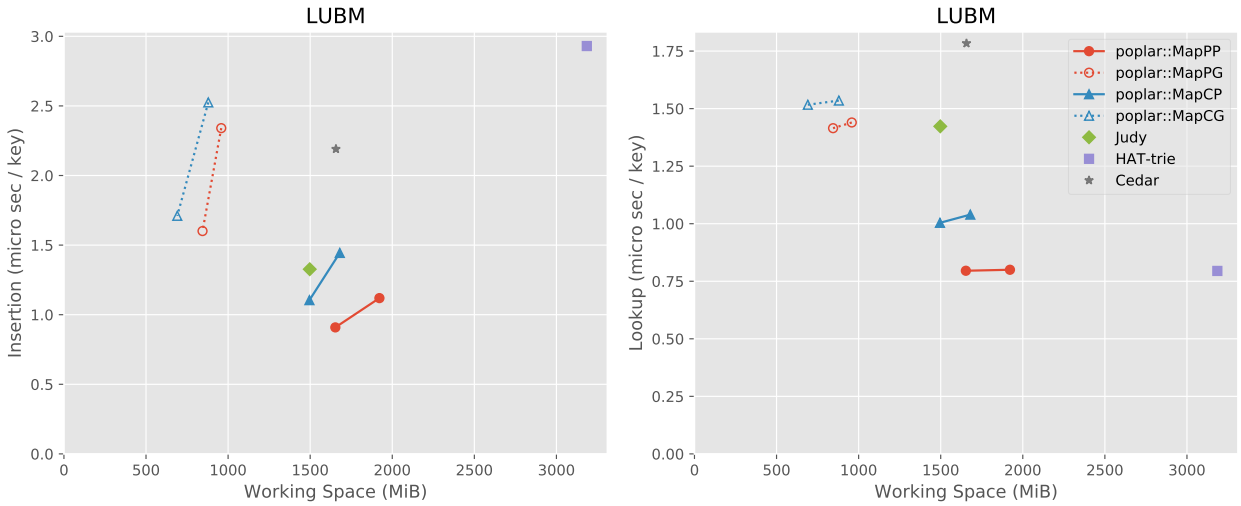

図 2: 実験結果

Practical implementation of space-efficient dynamic keyword dictionaries. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 221–233, 2017.

[12] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008.

[13] Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. m-bonsai: a practical compact dynamic trie. *CoRR*, abs/1704.05682, 2017.

[14] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Rajeev Raman. LZ78 compression in low main memory space. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 38–50, 2017.

[15] Google Inc. Sparsehash, 2015.

[16] Jon L Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 97, pages 360–369, 1997.

[17] Jun'ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, 15(9):1066–1077, 1989.

[18] Andreas Poyias, Simon J Puglisi, and Rajeev Raman. Compact dynamic rewritable (CDRW) arrays. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 109–119, 2017.

[19] Guy L Steele Jr, Doug Lea, and Christine H Flood. Fast splittable pseudorandom number generators. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 453–472. ACM, 2014.

[20] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.