ダブル配列辞書の時間効率の改善

松本 拓真

概要

文字列は計算機システムにおける最も基本的なデータ表現方法の一つであり，大量の文字列データを扱う現代において，文字列の集合を効率よく管理することは非常に重要である．文字列集合を管理するデータ構造の基礎は主に，ハッシュテーブル，もしくはトライと呼ばれるラベル付きグラフに基づいている．ダブル配列はラベル付きグラフを表現するデータ構造の一つであり，ラベル付きグラフの状態遷移表を 1 次元に圧縮した構造として表現される．ダブル配列の特徴として，入力文字列に基づく検索を高速に実行できる反面，構築速度が遅く，メモリを比較的多く消費するというボトルネックを持つ．その特徴から，数百 MB から数 GB 程度のデータに対するキーの追加や削除などの更新を必要としない静的辞書として利用されることが多い．本研究では，ダブル配列のボトルネックの改善に取り組む．具体的には，構築アルゴリズムの根幹である XCHECK と呼ばれる計算の高速化と，更新が容易でありノード数の少ないグラフ表現であるパトリシアトライをダブル配列で表現する方法を提案する．XCHECK の高速化では，ダブル配列の要素あたり 1bit のデータを追加する代わりに，構築速度の最大 5 倍の高速化を実現した．パトリシアトライを用いた動的キーワード辞書の実装では，自然言語からなるデータセットでダブル配列に基づく従来の技法と性能比較すると，メモリと検索速度を改善した上，XCHECK の高速化と合わせることで同程度の構築時間を維持した．

目次

第 1 章　はじめに　2

1.1　目的 4

1.2　章構成 4

第 2 章　準備　5

2.1　基本表記 6

2.2　トライ 6

2.3　ダブル配列 7

2.4　辞書の実装 11

第 3 章　ダブル配列の高速な構築　13

3.1　ビット列による集合の表現 14

3.2　bit-parallel XCHECK 16

3.3　理論的評価 20

3.4　実験的評価 21

第 4 章　ダブル配列によるパトリシアトライ　28

4.1　内部の遷移ラベルの表現 30

4.2　提案手法の検索 31

4.3　空間効率の良いキーの追加処理 32

4.4　理論的評価 34

4.5　実験的評価 36

第 5 章　おわりに　42

参考文献　44

第 1 章

はじめに

昨今の計算機システムの急速な発達と普及により，あらゆる計算機で大量のデータ処理が行われるようになっている．文字列は最も代表的なデータ表現の一つであり，文書，web ページ，URL，ゲノム配列，データの索引キーなどの様々な箇所で利用されている．そのことは，多くの計算機システムが効率よく動作するためには，大量の文字列からなるデータを効率よく管理するための仕組みが非常に重要であることを意味している．

本研究では特に，文字列に基づく索引を効率よく実現する技法に着目する．文字列による索引を実現する上で重要なのは，文字列からなるキーの集合をコンパクトに表現し，かつキーに基づく高速な検索を提供するデータ構造である．キー集合を管理するデータ構造の基礎は，ハッシュ法 [1] や，トライ [2,3] と呼ばれるラベル付き木に基づいているものが殆どである．ハッシュ法に基づく技法は，キーから作成した数バイトのフィンガープリントに基づいて探索する．どのようなデータでも比較的高速に検索が出来るため，文字列をキーとする連想配列の実装として多くのプログラミング言語の標準ライブラリに採用されている[*1][*2]．トライに基づく技法は，各文字列を木の根から連なる遷移ラベルの列として文字列集合を表現し，検索は木の上を入力文字列に対応するラベルによりノードを遷移していくことで検索を行う．トライは文字列集合のデータ構造として多くのクエリを実現する技法の基礎として利用されており，メモリ効率や検索効率を追求するようなデータ構造の設計や，入力キーの接頭辞に一致するキーの検索なども可能にする．

ダブル配列 [4,5] は，ラベル付きグラフを表現するデータ構造の一つであり，その基本構造は，ラベル付きグラフの状態遷移表を 1 次元に圧縮した表現として捉えることが出来る．状態遷移表とは，グラフのノード番号と遷移ラベルを索引とする 2 次元配列であり，遷移ラベルから直接遷移先の情報にアクセス出来るが，グラフが大きくなると膨大なメモリが必要になる．ダブル配列では，状態遷移表をノード番号毎の行に分割し，それぞれの行に存在する遷移要素が同じインデックスに 2 つ以上存在しないように 1 次元の配列上に並べ替えることで構築される．こうすることで，遷移表のような遷移先への高速なアクセス性能を持ちつつ，遷移表より大幅にメモリ効率が良くなる．

ダブル配列は非常に検索速度に秀でた実装を提供する一方で，遷移表を一次元に圧縮する工程をボトルネックとして構築が遅く，また他の索引データ構造よりメモリ消費量が比較的大きい．こういった特徴から，現在では数百 MB から数 GB 程度のサイズのデータに対する，キーの更新や削除を必要としない静的辞書として利用されることが多い．代表的な応用は形態素解析器の単語辞書であり，数 MB から数百 MB 程度の単語集合に対する高速な索引として利用されている．他には，トライを利用した言語モデルの実装で用いられ，クエリ速度は非常に速い反面，数 GB のデータセットに対するデータ構造の構築に数十時間を必要とする．

ダブル配列は，その計算量をヒューリスティックに抑えているデータ構造であり，ダブル配列を用いた実用的な計算量を持つ単語辞書の実現までに数多くの研究成果がある．Aoe [4,5] はダブル配列によるトライ表現の基本となるデータ構造と，検索とキー追加のアルゴリズムの基本形を提案した．Morita ら [6] は，ダブル配列の空要素が全体の数%しかないことに着目し，空要素のイ

[*1] `https://en.cppreference.com/w/cpp/container/unordered_map`

[*2] `https://doc.rust-lang.org/std/collections/struct.HashMap.html`

ンデックスを双方向連結リストで管理することで，キーの挿入を 6 倍から 320 倍まで高速化した．矢田ら [7] は，遷移先をノード番号と遷移ラベルの XOR によって決定することで，検索時にアクセスするインデックスがキャッシュラインにヒットしやすくなることを示した．矢田ら [8] は，遷移に XOR を用いる方法が，ダブル配列を一定の間隔のブロック毎に区切る性質を持つと言う特徴を利用し，辞書の更新時間を実用的なものにする手法を提案した．前田ら [9] は，ダブル配列によりトライだけでなく任意のラベル付きグラフを表現できることを示した．Kanda ら [10] は，キーの削除により増加する空要素の削減に用いる再配置法と再構築法を定量的に評価し，空要素率が数十 % の程度になるなら再構築が有用であることを示した．

1.1 目的

本研究では，ダブル配列のボトルネックの改善として 2 つの方法を提案する．

一つ目は，ダブル配列の構築において根幹となる XCHECK と呼ばれるアルゴリズムの改善である．XCHECK とは，ラベル集合を入力として，部分的に空要素を持つ配列上から，ラベル集合の全ての要素が空要素にマッチするようなオフセットを探索する演算である．提案するアルゴリズムは，ビットレベル並列化を元にしたアルゴリズムであり，複数のオフセット候補をまとめて計算することが出来る．ダブル配列の遷移の演算でよく用いられる足し算 (+) と XOR に対し，それぞれ適切なビット演算を用いたアルゴリズムを提案する．このアルゴリズムは，ダブル配列の要素あたり 1bit の記憶量を追加で必要とする．提案手法は従来の貪欲法に基づくアルゴリズムと同じ演算結果を得られるため，既存のダブル配列構築アルゴリズムをほぼ全ての状況で代替できる．

二つ目は，キー集合を表現するラベル付きグラフの一つである，キーの更新が容易でありノード数の少ないパトリシアトライ [11] と呼ばれるグラフをダブル配列で表現する方法を提案する．パトリシアトライでは，グラフ上で連続するシングルノードを文字列として一つの遷移ラベルで表現したグラフである．ここでシングルノードとは，遷移先を一つしか持たないノードである．パトリシアトライを用いることで，元のトライの形状によっては大幅にノード数を削減できる上，検索時にシーケンシャルアクセス出来る箇所が増加することで検索の高速化が期待できる．

1.2 章構成

章 2 では，提案手法の説明で用いる基本表記と知識の説明をする．章 3 では，XCHECK をビットレベル並列化により高速化する手法を提案する．章 4 では，パトリシアトライをダブル配列により表現する方法を提案する．章 5 では，本研究を総括し，結論を述べる．

第 2 章

準備

本章では，手法の説明で用いる基本表記と知識の説明を行う．

2.1 基本表記

アルゴリズムやデータ構造の解説で用いる基本表記を定義する．

整数 l, r について，l 以上 $r-1$ 以下の値からなる整数集合を $[l, r)$ と表す．配列，あるいは文字列の添え字は 0 から始まる（0-indexed）．文字集合，あるいはアルファベット集合，を Σ と表し，そのサイズを $\sigma = |\Sigma|$ と表す．文字列はアルファベットの列からなる．列 s の $i+1$ 番目の文字を $s[i]$ と表す．また, 列 s の $l+1$ 番目から r 番目の部分列を $s[l, r)$ と表す．例えば, $s = \{a, b, c, d, e\}$ なら，$s[3] = d$，$s[1, 4) = \{b, c, d\}$ である．s を文字列として"abcde"とも表す．ある文字列 s, t について，$s = t[0, i)$ $(1 \leq i \leq |t|)$ であるとき，s を t の接頭辞 (Suffix) と呼ぶ．また，文字列 p, s について，$p = t[i, |t|)$ $(0 \leq i < |t|)$ のとき，p を t の接尾辞 (Prefix) と呼ぶ．

計算モデルは Word-RAM を仮定する．Word-RAM は以下の特徴を持つ．

- 計算機は U ビットのメモリを持ち，計算機の語長は $w = \log U$ ビットである．ただし語長 w は 2 の冪乗である．
- w ビットの算術・論理演算，および連続する $w$bit のメモリアクセスが 1 単位時間で実行できる．
- メモリから読み込んだ値は $[0, U)$ の整数と見なす

値のビット列の番地 0 に近い方から k ビットのビット列を上位 k ビット，遠い方から k ビットのビット列を下位 k ビットと呼ぶ．形式的には，値 x の上位 k ビットは $x \mod 2^k$，下位 $w-k$ ビットは $\lfloor x/2^k \rfloor$ である．

計算量を概算する表記としてオーダー表記を用いる．まず定義を与える．

定義 2.1.1. ある関数 $g(n)$ に対し，$\mathrm{O}(g(n))$ を次のような関数の集合と定義する．

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c, n_0, \forall n \geq n_0, cg(n) \geq f(n) \geq 0\}$$

$f(n) \in \mathrm{O}(g(n))$ とは，n がある程度大きいときには $f(n)$ は $g(n)$ の定数倍以下であることを意味する．つまり関数の上界を表す．なおこれを $f(n) = \mathrm{O}(g(n))$ と書く．また，$\mathrm{O}(1)$ はある正定数を表す．

2.2 トライ

ラベル付き木，トライ [2,3] は，文字列の集合を管理するデータ構造である．図 2.1 にトライの例を示す．トライでは，文字列を根から葉へのパスの遷移ラベルの列により表現する．トライにおける検索は，入力文字列の先頭から一文字ずつ取り出し，トライを根から取り出した文字に対応する遷移を葉まで繰り返すことで行われる．あるノードの遷移先のノードを子ノード，あるいは単に子や子供と呼ぶ．逆に，あるノードの遷移元を親ノード，親と呼ぶ．同じ親を持つノードを兄弟

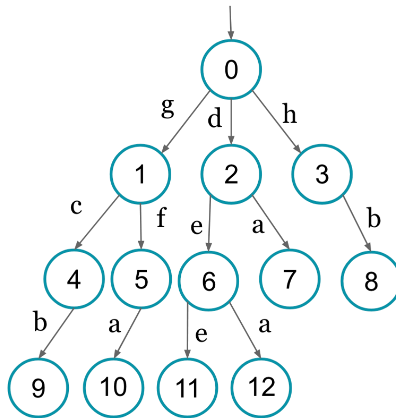


図 2.1: 文字列集合 {”da”,”dea”,”dee”,”gcb”,”gfa”,”hb”} を表現するトライ

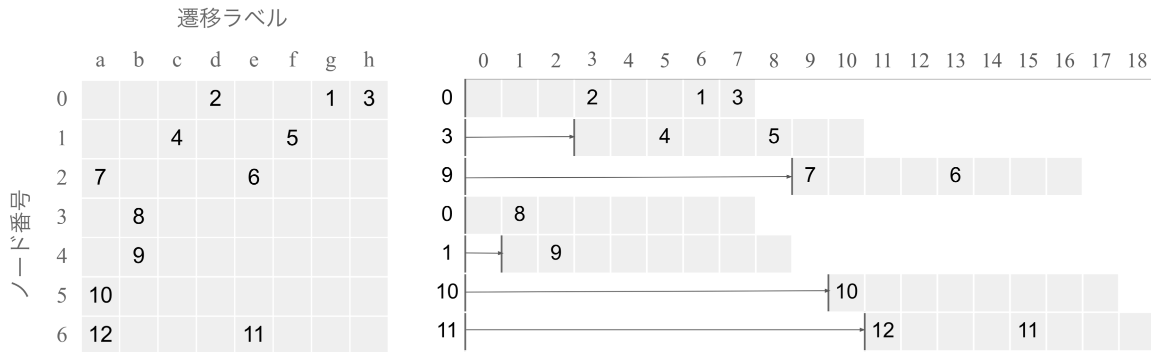


図 2.2: 状態遷移表の行にオフセットを与え，要素を一次元上のインデックスに割り当てる様子.

ノード，兄弟と呼ぶ.

2.3　ダブル配列

ダブル配列 [4,5] は，ラベル付きグラフの状態遷移表を一次元配列に圧縮した表現をコンセプトとしている．状態遷移表とは，ノード番号と遷移ラベルを索引として遷移先の情報を保存する 2 次元配列であり，ラベル付きグラフの最も基本的な表現の一つである．ダブル配列では，この状態遷移表を行毎に分割し，それぞれの行の要素が同じインデックスに 2 つ以上存在しないように行のオフセットを決定することで一次元のインデックスに要素を割り当てる．図 2.2 は図 2.1 のトライの状態遷移表を一次元に圧縮する様子を表しており，遷移表の各要素が重ならないように各行にオフセットを与えている．

ダブル配列のアルゴリズムでは二項演算 op を用いて要素の配置を決定する．op は，ノードに対応する行のオフセット x と遷移ラベル c から，子ノードの位置 $t = \mathrm{op}(x, c)$ を得るために用いる．二項演算には基本的に足し算 $(+)$ か XOR$(\oplus)$ のどちらかが用いられる．XOR とは，整数

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BASE	0			9		1	3	0	10					11		
CHECK		7	5	0		6	0	0	6	3	8	13		3		13

図 2.3: 図 2.1 のトライを表現するダブル配列．各ノードのインデックスは図 2.2 に対応する．

値同士のビット毎の排他的論理和からなるビット列を計算する演算である．足し算を用いる二項演算を plus-op，XOR を用いる二項演算を xor-op と表す．また，op の逆演算を iop と表し，$\forall x \in \mathbb{N}, \forall c \in \Sigma : \texttt{iop}(\texttt{op}(x, c), c) = x$ を満たす．iop は，要素の位置 t と t への遷移ラベル c から，親ノードの遷移行のオフセット $x = \texttt{iop}(t, c)$ を得るために用いる．iop は，plus-op なら引き算 ($\texttt{plus-iop}(t, c) = t - c$)，xor-op なら XOR ($\texttt{xor-iop}(t, c) = t \oplus c$) に対応する．図 2.2 は plus-op を用いた場合のダブル配列の要素の配置に対応している．plus-op を用いる場合は，子ノードの位置 t は遷移行のオフセット x と遷移ラベル c の和で決まるため，図 2.2 と同様の配置を行うが，xor-op を用いる場合は遷移先は XOR で決定するため，図と異なる様子になることに注意してほしい．plus-op を用いた場合の方が要素の配置が直感的に得られるため，本論文の例では plus-op を用いた場合を基本としている．Aoe [4,5] がダブル配列を提案した際は plus-op を用いていたが，xor-op を用いる方法が Yata ら [7] により提案された．Yata らによれば，遷移に XOR を用いるとオフセットと子ノードのインデックスが同じキャッシュアラインメントに乗りやすくなり，配列要素へのアクセスが効率的になるとしている．

ダブル配列は BASE と CHECK と呼ばれる二つの配列でラベル付きグラフを表現する．BASE[u] と CHECK[u] はそれぞれノード u に対応し，単にノード u の BASE，CHECK，と表すこともある．BASE[u] は，ノード u の遷移行のオフセット，CHECK[u] はノード u の親ノードを表す．トライのルートには常にインデックス 0 を割り当てる．図 2.3 は，図 2.1 に対応するトライを図 2.2 の配置で表現するダブル配列である．図のダブル配列のインデックスは，図 2.3 で構築した行のオフセットを元に配置してあり，それぞれのノードに対応する行のオフセットが BASE に，親ノードのインデックスが CHECK に保存されている．ダブル配列のノード番号は配置されたインデックスによって決定する．つまり図 2.1 と図 2.3 のノード番号は異なる．グラフにノード s からラベル c に対応する遷移先 t が存在するとき，式 (2.1) を満たす．

$$t \leftarrow \texttt{op}(\mathrm{BASE}[s], c), \mathrm{CHECK}[t] = s \tag{2.1}$$

式 (2.1) の意味を言い換えると，ノード s からラベル c に基づく子ノード t への遷移はわずか 2 ステップで行われ，子ノード t は s の BASE とラベル c から計算でき，t の親ノードは t の CHECK から得られる．このように単純で少ない演算によって遷移を計算できるため，ダブル配列は非常に高速な検索を実現できる．

状態遷移表を一次元に圧縮するようにダブル配列を構築すると，殆どの場合にいくつかの空要素を含むようになる．ここで，EMPTY[i] $\in \{0, 1\}$ によって，要素 i が空なら EMPTY[i] $= 1$，空

でないなら $\mathrm{EMPTY}[i]=0$ と表すことにする．要素 0 は常にルートを表し，常に $\mathrm{EMPTY}[0]=1$ である．ダブル配列の配列サイズを $N=|\mathrm{BASE}|=|\mathrm{CHECK}|$ と表す．トライのノード数を n, ダブル配列の空要素数を m とすると，$N=n+m$ が成り立つ．

2.3.1 ダブル配列の構築

ダブル配列の構築で重要になるのが，XCHECK と呼ばれる演算である．XCHECK とは，ダブル配列の要素が空かどうかを表す配列 EMPTY と遷移ラベルの集合 C が与えられたとき，式 (2.2) を満たす BASE 値 x を探す演算である．

$$\mathrm{XCHECK}(C)=x\in\mathbb{N}\mid\forall c\in C:(\mathrm{EMPTY}[\mathtt{op}(x,c)]=1) \tag{2.2}$$

即ち，子ノードへの遷移としてラベル集合 C を持つノードの子ノードをダブル配列のどの位置に格納するかを探す演算である．遷移の二項演算に `plus-op` を用いているなら，図 2.2 における遷移表の行のオフセットを探すことに対応する．

ダブル配列の構築は，それぞれのノードから子ノードへの遷移ラベルの集合を取得し XCHECK を繰り返し，決定したインデックスに子ノードを保存していくことで実行される．要素の保存は，まず現在のノード u の BASE に XCHECK で得た x を保存し，次に子ノードのインデックスの CHECK に現在のノードのインデックスを保存する．つまり，$\mathrm{BASE}[u]\leftarrow x$ と $\mathrm{CHECK}[\mathtt{op}(x,c)]\leftarrow u$ $(\forall c\in C)$ となる．このとき空要素に子ノードの要素を割り当てたので，EMPTY の更新も行う．例えば, ノード $u=1$ の子ノードへの遷移ラベル集合 $C=\{2,4,5\}$ が与えられ, $x=\mathrm{XCHECK}(C)=3$ が得られたとき, 保存する値は $\mathrm{BASE}[1]\leftarrow 3,\mathrm{CHECK}[3+2=5]\leftarrow 1,\mathrm{CHECK}[3+4=7]\leftarrow 1,\mathrm{CHECK}[3+5=8]\leftarrow 1$ である．加えて，EMPTY のインデックス $\{5,7,8\}$ に 0 を与える．

XCHECK を計算する最も単純な方法は，式 (2.2) の解候補となる x を貪欲に探す方法である．このアルゴリズムの計算量は，x の候補が N 個 (ダブル配列の配列長を超えた箇所は必ず空要素であるため)，遷移ラベルの個数が $|C|\leq\sigma$ であるため，$\mathrm{O}(N|C|)\leq\mathrm{O}(N\sigma)$ と大きい．

ノード集合 V と遷移集合 E からなるグラフのダブル配列を構築するとき，各ノードについてそれぞれ XCHECK を行う場合の計算量を考える．ノード u が持つ遷移のラベル集合を C_u, ノード u に対する XCHECK 呼び出し時のダブル配列の長さを N_u とすると，全体の計算量は $\sum_{u\in V}\mathrm{O}(N_u|C_u|)\leq\sum_{u\in V}\mathrm{O}(N|C_u|)\leq\mathrm{O}(N|E|)$ になる．グラフが木構造の時，$|E|=|V|\leq N$ より $\mathrm{O}(N^2)$ であり，非常に大きいと言える．

しかし，いくつかのヒューリスティックな工夫により，全てのノードに対する XCHECK を行っても実用的な計算時間で抑えられるようになることが分かっている．まず重要なのは，x を最小値から昇順に探すことを繰り返すことだ．このようにすると，配列の先頭に近い位置から順に要素が保存されていく．すると配列の先頭に近いほど空要素の割合が少なく，末尾に近いほど空要素の割合は多くなり易い．空要素が少ない領域は，解候補 x が式 2.2 の条件を満たすかどうかの確認の打ち切りが早くなり，XCHECK の探索が速くなる．空要素が多い領域は x が条件を満たす可能性が

高く，解が見つかれば XCHECK の計算を打ち止めできる．

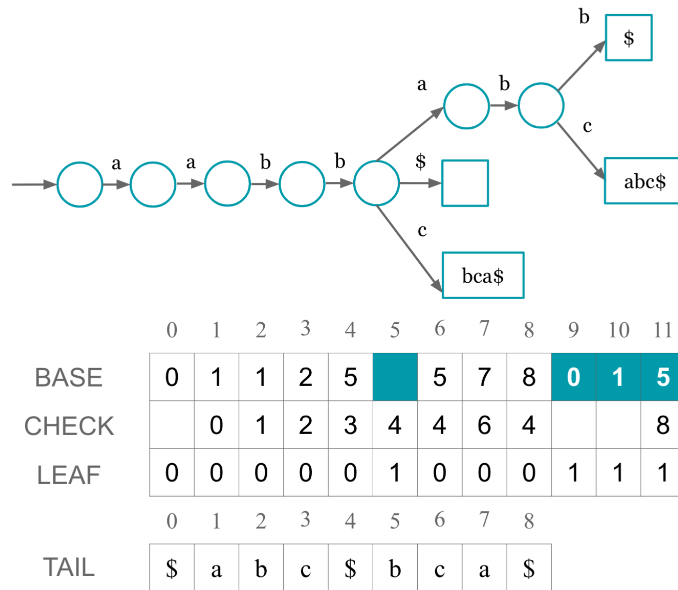
Algorithm 1 Empty-Link Method による XCHECK

Require: E: 空要素のインデックス集合の双方向連結リスト
Require: C: 遷移ラベル集合の配列

```
1: for e ∈ E do
2:     x ← iop(e, C[0])
3:     ok ←true
4:     for c ∈ C[1, |C|) do
5:         t ← op(x, c)
6:         if t < N & EMPTY[t] = 0 then
7:             ok ← false
8:             break
9:     if ok then
10:        return iop(N, C[0])
11: return N
```

次に，空要素のインデックスを双方向連結リストで管理し，ダブル配列の空要素を効率的に走査出来るようにする．双方向連結リストとは，n 個の要素の走査が $\mathrm{O}(n)$，任意の位置への要素の挿入と削除が $\mathrm{O}(1)$ で出来るデータ構造である．そして XCHECK では，空要素 e に対して，$x = \mathtt{iop}(e, \min C)$ となる x のみを解の候補とする．この方法を Empty-Link Method(ELM) [6] と呼ぶ．ELM による XCHECK のアルゴリズムを Algorithm1 に示す．Morita ら [6] によれば，一般的なキーセットから構築されるダブル配列の空要素の割合は非常に小さいことから，ELM により構築時間を 6 倍から 320 倍まで高速化できるとしている．

キー集合 S を保存したダブル配列に新たにキー挿入する場合，分岐が発生するノード r では遷移ラベル集合に新たなラベル c' を追加することになる．このとき，$\mathrm{EMPTY}[\mathtt{op}(\mathrm{BASE}[r], c')] = 1$ であればそのまま新たな遷移を空要素に保存すればよいが，0 の場合は新たな遷移を保存したい箇所に既に要素が存在してしまっている．この現象を衝突と呼ぶ．この場合，ノード r の現在の遷移ラベル集合と c' を全て保存できるよういくつかの要素を再配置する必要がある．具体的には，分岐が発生したノード r か，衝突箇所 i に元から存在するノードの親ノード $k = \mathrm{CHECK}[i]$ のいずれかの子ノードを全て再配置する．r と k の選び方は，それぞれの子ノードの数が少ない方とする．選ばれたノードを u とし，u の遷移ラベル集合を C とすると，XCHECK(C) で新たな BASE 値を得て，得られた BASE に対応するように子ノードを再配置する．ただし，r を選んだ場合は C にキー挿入によって追加される遷移ラベル c' を加えておく．子ノードの数が少ない方を選んだ理由は，XCHECK の計算量は入力の遷移ラベルの個数に依存するため，少ない方を選ぶ方が計算が速くなるためである．特にトライの構造に注目すると，根付近のノードは頻繁に遷移ラベルの追加が行われ，結果として遷移ラベルの数も多くなりやすく，無作為にノードを選ぶと遅い XCHECK



	0	1	2	3	4	5	6	7	8	9	10	11
BASE	0	1	1	2	5		5	7	8	0	1	5
CHECK		0	1	2	3	4	4	6	4			8
LEAF	0	0	0	0	0	1	0	0	0	1	1	1

	0	1	2	3	4	5	6	7	8
TAIL	$	a	b	c	$	b	c	a	$

図 2.4: MP トライを表現するダブル配列による辞書．キー集合 {"aabb","aabbabb","aabbabcabc","aabbcbca",} を表現している．BASE の白い値は TAIL のインデックスを指している．

を頻繁に実行してしまうことになる．

ダブル配列での子ノードの列挙は，BASE,CHECK のみでは効率的に行えない．なぜなら，全ての遷移文字 $c \in \Sigma$ に対し，式 (2.1) を満たすかどうかを確認しなければならず，$O(\sigma)$ 時間かかるからだ．Nakamura ら [12] は，ノードの兄弟要素に対する連結リストを追加することで子の列挙を効率化している．

矢田ら [8] は，`xor-op` を用いた場合の XCHECK の高速化を行っている．まず，ダブル配列を長さ σ ごとのブロックに分割し，全てのブロックを双方向連結リストで管理する．このとき，σ は 2 の冪乗の値とする．この方法を Block-Link Method (BLM) と呼ぶ．`xor-op` を用いると，BASE 候補 x と遷移先 $\mathtt{xor}-\mathtt{op}(x,c) = x \oplus c$ $(c < \sigma)$ は明らかに同じブロック内のインデックスになる．したがって，XCHECK をブロック毎に考えることが出来る．このとき，あるブロックにおける XCHECK で解が見つからなかった時の回数をブロック毎に記録する．そして，一定回数以上解が見つからなかったブロックを，ブロックの連結リストから削除する．こうすることで，XCHECK で解が見つかる可能性の低いブロックを探索対象から除外出来る．

2.4 辞書の実装

最小接頭辞トライ (Minimal-prefix trie, MP トライ) [4,5,13] は，辞書の実装によく用いられる．MP トライでは，保存したキーが一意になるまでの最小の接頭辞集合をトライとして表現し，残り

の接尾辞を文字列として表現する．また，保存されたキーがそれぞれ一意な葉を持つようにするため，保存するキーの終端に終端文字'$' を追加して保存する．こうすることで，あるキー s が他のキー t の接頭辞であるとき，つまり $s = t[0, i)$ であるときであっても，キー $s'\$'$ のみで到達可能な一意な葉を持つようになる．図 2.4 は MP トライとそれを表現するダブル配列である．MP トライをダブル配列で表現する場合，接頭辞集合を BASE と CHECK からなる配列に保存し，接尾辞を TAIL と呼ばれる配列に保存する．葉ノードは TAIL に保存した対応する接尾辞の先頭アドレスを持つ．ダブル配列の葉 u の BASE は未使用であるため，接尾辞の先頭アドレス f を $\mathrm{BASE}[u] = f$ に保存する．このとき，ノード i が葉かどうかをビット列 $\mathrm{LEAF}[i] = \{0, 1\}$ により表現し，i が葉なら 1，違うなら 0 とする．

第 3 章

ダブル配列の高速な構築

本章では，XCHECK の演算を高速化する方法を提案する．我々が提案する手法では，ダブル配列の要素の使用状況を表すビット列 EMPTY を用いて，ビットレベルで並列化することにより高速化を実現する．

まず，XCHECK の解が満たす条件である式 (2.2) を拡張し，式 (2.2) を満たす値の集合 X，つまり XCHECK の解の集合を定義する．

定義 3.0.1. 遷移ラベルの集合 C と空要素のインデックス集合 E が与えられたとき，XCHECK の解の集合 X を次のように定義する．

$$X = \{x \in \mathbb{N} \mid \forall c \in C : \texttt{op}(x, c) \in E\}$$

定義 3.0.1 を式変形することで，以下の補題を得る．

補題 3.0.1. 遷移ラベルの集合 C と空要素のインデックス集合 E が与えられたとき，XCHECK の解の集合 X は以下のようになる．

$$X = \bigcap_{c \in C} \{\texttt{iop}(e, c) \mid e \in E\} \tag{3.1}$$

Proof. 定義 3.0.1 を式変形し証明する．

$$\begin{aligned} X &= \{x \in \mathbb{N} \mid \forall c \in C : \texttt{op}(x, c) \in E\} \\ &= \bigcap_{c \in C} \{x \in \mathbb{N} \mid \texttt{op}(x, c) \in E\} \\ &= \bigcap_{c \in C} \{\texttt{iop}(e, c) \mid e \in E\} \end{aligned}$$

1 行目から 2 行目の変形は，遷移ラベル $c \in C$ についてそれぞれ解集合を得て，それらの積集合として表現した．2 行目から 3 行目の変形は，$\texttt{op}(x, c) = e \to \texttt{iop}(e, c) = x$ から得た． □

即ち，解集合 X は，空要素集合 E から，遷移ラベル c に対応する $\texttt{iop}(e, c)$ でそれぞれ集合を作り，それらの積集合として表現できる．得られた式 (3.1) には，任意の解候補 x が現れない．言い換えれば，解集合 X を得るために，任意の解候補 x を網羅的に探索する必要は無く，X は C と E から直接構成できるということを表している．

3.1 ビット列による集合の表現

次に，集合をビット列で表現する方法を説明する．集合 S を表すビット列 $\mathrm{B}[i] = \{0, 1\}$ を用意し，$i \in S$ なら 1，$i \notin S$ なら 0 を表す．このビット列は，語長 w の計算機で $w$bit の整数からなる長さ $\lceil N/w \rceil$ の配列 W として実装する．集合要素 $i \in S$ に対応するビットは，$W[\lfloor i/w \rfloor]$ のインデックス $i \mod w$ のビットに対応する．図 3.1 は，集合を表すビット列の例である．図

0								1								2								3							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1

$$S = \{2,3,5,7,11,13,17,19,23,29,31\}, \quad w = 8$$

図 3.1: 集合 S を表現するビット列

```
1  constexpr int w=64;
2  std::vector<uint64_t> W((N+w-1)/w);
3  bool get(size_t i) {
4    size_t q = i / w;
5    size_t r = i % w;
6    return (W[q] >> r) & 1ull;
7  }
8  void set(size_t i, bool bit) {
9    size_t q = i / w;
10   size_t r = i % w;
11   if (bit) W[q] |=  1ull << r;
12   else     W[q] &= ~(1ull << r);
13 }
14 uint64_t get_wbit(size_t from) {
15   size_t q = from / w;
16   size_t r = from % w;
17   if (r == 0) return W[q];
18   else        return (W[q+1] << (w-r)) | (W[q] >> r);
19 }
```

図 3.2: 集合をビット列で表現する方法を，$w$bit 整数の配列として実装する C++ コード.

3.2 は，$w$bit 整数の配列により集合を表現する方法の C++ 言語による実装例である．この実装では，集合の中で任意の値の範囲からなる部分集合を表す $w$bit のビット列を定数時間で得られる (図 3.2,14 行目, 関数 get_wbit)．`get_wbit`(f) で得られた $w$bit のビット列を S_f とすると，$S_f[0,w) = B[f, f+w)$ に対応する．このデータ構造を bitset として定義する．

定義 3.1.1 (データ構造 bitset)**.** bitset は，整数集合 S を $w\lceil n/w \rceil$bit で表現し，以下の操作を定数時間で実行できる．

- `get`(i): $i \in S$ なら 1，$i \notin S$ なら 0 を返す．
- `set`(i,b): $b=1$ なら $S \leftarrow S \cup \{i\}$，$b=0$ なら $S \leftarrow S \setminus \{i\}$ を行う．
- `get_wbit`(f): $\{i \in S \mid f \le i < f+w\}$ を表すビット列を返す．

3.2 bit-parallel XCHECK

空要素のインデックス集合 E の bitset を構築し，ビットレベル並列化により高速に XCHECK を実行する方法を説明する．最終的には `plus-op` と `xor-op` それぞれについてアルゴリズムを提案するが，まず共通する概念を説明する．

まず，bitset の，幅 w の値域の部分集合のビット列を定数時間で得られるという特徴を利用するため，補題 3.0.1 から以下の補題を得る．

補題 3.2.1. 遷移ラベル集合 C と空要素インデックス集合 E が与えられたとき，XCHECK の解集合 X の，幅 w の値域 $[f, f+w)$ からなる部分集合 $X_f = \{i \in S \mid f \le i < f+w\}$ は以下のように表せる．

$$X_f = \bigcap_{c \in C} \{\mathtt{iop}(e,c) \mid f \le \mathtt{iop}(e,c) < f+w, e \in E\}$$

Proof. 補題 3.0.1 の証明と同様の手順で式変形を行う．

$$\begin{aligned} X_f &= \{x \mid f \le x \le f+w, \forall c \in C, \mathtt{op}(x,c) \in E\} \\ &= \bigcap_{c \in C} \{x \mid f \le x \le f+w, \mathtt{op}(x,c) \in E\} \\ &= \bigcap_{c \in C} \{\mathtt{iop}(e,c) \mid f \le \mathtt{iop}(e,c) \le f+w, e \in E\} \end{aligned}$$

□

補題 3.2.2. 二項演算が `plus-op` のとき，X_f は以下のように表せる．

$$X_f = \bigcap_{c \in C} \{e - c \mid e \in E_{f+c}\} \quad (3.2)$$

Proof. 整数 x, y, z が $x < y < z$ のとき，$\texttt{plus-op}(x, c) = x + c$ より明らかに $\forall c \in \sigma : x + c < y + c < z + c$ である．よって以下の式変形が成り立つ．

$$
\begin{aligned}
X_f &= \bigcap_{c \in C} \{e - c \mid f \leq e - c < f + w, e \in E\} \\
&= \bigcap_{c \in C} \{e - c \mid f + c \leq e < f + c + w, e \in E\} \\
&= \bigcap_{c \in C} \{e - c \mid e \in E_{f+c}\}
\end{aligned} \tag{3.3}
$$

□

補題 3.2.3. 二項演算が xor-op のときかつ f が w の倍数の時，X_f は以下のように表せる．

$$
X_f = \bigcap_{c \in C} \{e \oplus c \mid e \in E_{\lfloor (f \oplus c)/w \rfloor w}\} \tag{3.4}
$$

Proof. 整数集合 $X = [kw, (k+1)w)$ の要素 x について，$kw \leq x < (k+1)w$ より明らかに $\lfloor x/w \rfloor = k$ が常に成り立つ．即ち，$x \in [kw, (k+1)w)$ の下位 $w - \log w$ ビットは全て同じである．次に，任意の整数 c により X の全ての要素に対する $x \oplus c$ を行う場合を考える．X の要素の下位 $w - \log w$ ビットは常に同じであるため，$X' = \{x \oplus c \mid x \in [kw, (k+1)w)\}$ の要素の下位 $w - \log w$ ビットも常に同じである．よって，任意の整数 k' により $X' = [k'w, (k'+1)w)$ と表せる．k' は $kw \oplus c$ の下位 $w - \log w$ ビットを取り出した整数なので，$k' = \lfloor (kw \oplus c)/w \rfloor$ として得られる．即ち，

$$
\{x \oplus c \mid kw \leq x < (k+1)w\} = \{y \mid \lfloor (kw \oplus c)/w \rfloor w \leq y < (\lfloor (kw \oplus c)/w \rfloor + 1)w\} \tag{3.5}
$$

が成り立つ．

よって f が w の倍数の時，以下の式変形が成り立つ．

$$
\begin{aligned}
X_f &= \bigcap_{c \in C} \{e \oplus c \mid f \leq e \oplus c \leq f + w, e \in E\} \\
&= \bigcap_{c \in C} \{e \oplus c \mid gw \leq e \leq (g+1)w, e \in E\} \\
&= \bigcap_{c \in C} \{e \oplus c \mid e \in E_{gw}\} \\
&\quad (\text{ただし}, g = \lfloor (f \oplus c)/w \rfloor)
\end{aligned} \tag{3.6}
$$

□

このとき，kw 以上 $(k+1)w$ 未満の整数 x について，$x = kw + m$ $(0 \leq m < w)$ より，x の上位 w 以外のビットは f と同じである．

補題 3.2.2,3.2.3 は，ある共通のことを表している．どういうことかというと，XCHECK の解集合 X の部分集合 $X_f = \{x \mid f \leq x < f + w\}$ の計算過程で，各遷移ラベル c に対して計算する解

集合は，E の get_wbit から得られる $w$bit のビット列が表す集合から得られると言うことである．よって，`plus-op` と `xor-op` それぞれのアルゴリズムを構築するには，式 (3.2) と式 (3.4) の集合の項をそれぞれどのように計算するかにのみ注目すれば良い．

3.2.1 `plus-op` の bit-parallel XCHECK

式 (3.2) の集合の項 $\{e - c \mid e \in E_{f+c}\}$ を表すビット列を考える．集合の要素全てに対し任意の値 c を引くと言うことは，bitset の各要素を c 個先頭にずらすことに対応する．即ち，bitset の `get_wbit`$(f + c)$ で得られる長さ w のビット列そのものである．

Algorithm 2 `plus-op` の bit-parallel XCHECK のアルゴリズム

Require: e_0: 空要素のインデックスの先頭
Require: E: 空要素のインデックス集合の bitset
Require: C: 遷移ラベル集合の配列

```
 1: f ← max{e_0 − C[0], 0}
 2: while f + C[0] < N do
 3:     b ← 2^w − 1                          ▷ w 個の 1 からなる値
 4:     for c ∈ C do
 5:         b ← b AND E.get_wbit(f + c)
 6:         if b = 0 then
 7:             break
 8:     if b ≠ 0 then
 9:         return f + ctz(b)
10:     f ← f + w
11: return N − C[0]
```

X_f を計算するには，各遷移ラベル c について，`get_wbit`$(f + c)$ で得たビット列全てに対する論理積を計算する．このとき，X_f に一つ以上要素があれば，いずれかの 1 に対応する要素を XCHECK の解として返却する．解を $\min X_f$ とすれば，従来のアルゴリズムと全く同じ解になる．ここで，以下の演算を定義する．

- ctz(b): (count trailing zeros)．値 b の上位から連続する 0 の数を返す．言い換えれば，b の 1 を表すビットの中で最上位のビットのインデックスを返す．

ctz は，定数個のビット演算により計算できる他，命令セット BMI1 が組み込まれた CPU であれば 1 単位時間で計算できる[*1]．X_f が解を持つとき，X_f を表すビット列 b を元に $\min X_f$ を $f + \mathrm{ctz}(b)$ で計算する．`plus-op` の bit-parallel XCHECK のアルゴリズムを Algorithm2 に示す．

[*1] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_tzcnt_64

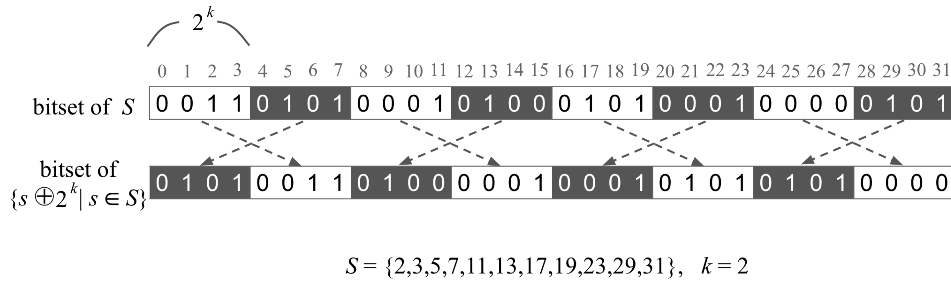


図 3.3: bitset 上で，集合 S から $\{s \oplus 2^k \mid s \in S\}$ を計算するときの blockwise-swap.

3.2.2 xor-op の bit-parallel XCHECK

式 (3.4) の集合の項 $\{e \oplus c \mid e \in E_{\lfloor (f \oplus c)/w \rfloor w}\}$ を表すビット列を考える．XOR の演算は，ビット毎の排他的論理和なので，演算をビット毎に考えることが出来る．ここで，値 b の上位から k ビット目のビットを反転させるということの意味を考える．つまり，bitset のインデックス b の k ビット目を反転させたとき，b から $b \oplus 2^k$ はどのように移動するかを考える．これは，インデックスを先頭から連続する 2^k 個ごとのビット列のブロックに分割して考えると分かり易い．先頭から $i+1$ 個目のブロックをブロック i と表すと，ブロック $2n$ とブロック $2(n+1)$ のビット列を入れ替えることに対応する (図 3.3)．この操作を，blockwise-swap と呼ぶことにする．値の全てのビットに対する XOR は，c のビットを一つづつ見て，$c_k = 1$ ならブロック長 2^k の blockwise-swap を実行し，$c_k = 0$ なら何もしないということで演算できる．

XOR を実行するビット k が $k < \log_2 w$ のとき，即ちブロック長 $2^k < w$ のとき，swap 先のビット列は同じ w ビットの値の範囲内にある．よって，遷移ラベルの上位 $\log_2 w$ ビットの XOR に対応する演算は，bitset の同じ整数値に対する演算としてまとめて処理できる．つまり，w ビットの bitset で表現できる集合 S の $\{s \oplus c \mod 2^k \mid s \in S\}$ をまとめて計算する．blockwise-swap を用いて，$\{s \oplus c \mod 2^k \mid s \in S\}$ を計算する C++ コードを図 3.4 に示す．ここでは遷移ラベル一文字を 1 バイトで表現している．14 行目の処理が blockwise-swap に対応する．blockwise-swap の演算は図のように AND，OR，ビットシフトで計算できる．

最後に，`xor-op` の bit-parallel XCHECK のアルゴリズムを Algorithm3 に示す．`plus-op` の bit-parallel XCHECK との主な違いは，1 行目の解候補の先頭インデックス f の初期値の与え方と, 5 行目の解集合の積集合を計算するときの右項である．`xor-op` の bit-parallel XCHECK では，ラベル c に対する解集合は，`get_wbit` で得た $w$bit の bitset の各要素 x への $x \oplus (c \mod 2^{\log_2 w})$ を `xor_bitset_element` で計算している．

図 3.4: blockwise-swap を用いて，w ビットの bitset で表現できる集合 S から $\{s \oplus (c \mod 2^k) \mid s \in S\}$ を計算する C++ コード

```
1  constexpr uint64_t mask[6] = {
2    0b0101 * 0x1111111111111111,
3    0b0011 * 0x1111111111111111,
4    0x0F0F0F0F0F0F0F0F,
5    0x00FF00FF00FF00FF,
6    0x0000FFFF0000FFFF,
7    0x00000000FFFFFFFF
8  };
9  uint64_t xor_bitset_element(uint64_t b, char c) {
10   for (int k = 0; k < 6; i++) { // log_2 64 = 6
11     int len = 1 << k;
12     if ((c & len) == 0) continue;
13     // blockwise-swap of length 2^k
14     b = ((b & mask[i]) << len) | ((b >> len) & mask[i]);
15   }
16   return b;
17 }
```

3.3 理論的評価

まず最悪時計算量の概算を行う．`plus-op` と `xor-op` それぞれに共通して，提案手法では XCHECK の解候補を連続する w 個をまとめて計算する．そのため，解候補の先頭 f は w 個間隔で設定すれば良い．遷移ラベルに関するループはそのままである．よって一回の bit-parallel XCHECK の計算量は $\mathrm{O}\left(\lceil \frac{N}{w} \rceil\right) \mathrm{O}(|C|G) = \mathrm{O}\left(\frac{N}{w}|C|G\right)$ である．ここで G は，ラベル $c \in C$ に関する解集合を表すビット列を得るのに必要な計算量を表しており，`plus-op` における E.`get_wbit`$(f+c)$，`xor-op` における `xor_bitset_element`$(E$.`get_wbit`$(\lfloor (f \oplus c)/w \rfloor w), c)$，の計算量に対応する．

`plus-op` の E.`get_wbit`$(f+c)$ は $\mathrm{O}(1)$ の関数であるため，XCHECK の計算量は $\mathrm{O}\left(\frac{N}{w}|C|\right)$ である．

`xor-op` の `xor_bitset_element`$(E$.`get_wbit`$(\lfloor (f \oplus c)/w \rfloor w), c)$ について，算術演算 $\lfloor (f \oplus c)/w \rfloor$ と関数 `get_wbit` は $\mathrm{O}(1)$ であり，関数 `xor_bitset_element` は $\log_2 w$ 回のループで定数

Algorithm 3 `xor-op` の bit-parallel XCHECK のアルゴリズム

Require: e_0: 空要素のインデックスの先頭
Require: E: 空要素のインデックス集合の bitset
Require: C: 遷移ラベル集合の配列

```
1:  f ← ⌊(e_0 ⊕ c)/w⌋w                          ▷ e_0 ⊕ c の上位 log_2 wbit を 0 とした値
2:  while f + C[0] < N do
3:      b ← 2^w − 1                             ▷ w 個の 1 からなる値
4:      for c ∈ C do
5:          b ← b AND xor_bitset_element(E.get_wbit(⌊(f ⊕ c)/w⌋w), c)
6:          if b = 0 then
7:              break
8:      if b ≠ 0 then
9:          return f + ctz(b)
10:     f ← f + w
11: return N − C[0]
```

時間のビット演算が必要である．よって全体の計算量は $\mathrm{O}\left(\frac{N}{w}|C|\log w\right)$ である．

`plus-op` と `xor-op` の bit-parallel XCHECK を比較すると，`plus-op` の方が $\mathrm{O}(\log w)$ 倍少ない．最初のループで直ちに解が得られる場合でも，G の計算量の比率は $\mathrm{O}(\log w)$ 倍と同じである．よって提案手法による計算量改善の効果は `plus-op` の方が高いといえる．

従来法と計算量を比較する．従来の XCHECK は ELM を利用すると $\mathrm{O}(|E||C|)$ となる．提案手法のループは先頭の空要素から開始しているものの，ぞれぞれの解候補は連続する要素の走査を行っている．よって最悪時計算量のみで従来法と性能を比較することは難しい．

そこでダブル配列のヒューリスティックな特徴に基づいて評価を試みる．ただし，ダブル配列の要素の密度は末尾になるほど疎で空要素が多くなるため，XCHECK のループが最悪時まで回ることはまれである．加えて，空要素の密度が高いということは，ELM により次の空要素へインデックスをスキップするときの間隔も小さくなり，配列の末尾に近づくほど ELM の高速化の効果は低くなる．また，解が解候補 f の初期値から w 個以内に見つかる場合，bit-parallel XCHECK は 1 回目のループで解を見つけることになるため，直ちに解を得ることが出来る．よって提案手法はダブル配列の特徴と相性が良いと考えられる．実際の性能は次節で実験的に評価する．

3.4 実験的評価

本節では bit-parallel XCHECK の性能を，ダブル配列が実際に利用される状況を想定した実験により評価する．実験設定として 3 つの状況を考える．

一つ目は，形態素解析器の単語辞書である．辞書の実装にダブル配列を用いている形態素解析器

表 3.1: 実験に用いる基本実装の詳細

	dynamic/static	ラベル付きグラフ	遷移ラベルの単位 ($\max \sigma$)	遷移関数 `op`
crawdad	static	トライ [2,3]	ユニコードのコードポイント (2^{32})	`xor-op`
DALM	static	Reverse Trie [2]	単語 ID (2^{31})	`plus-op`
Cedar	dynamic	MP トライ [4,5,13]	バイト文字 (2^8)	`xor-op`

に，Mecab[*2]や vibrato[*3]がある．vibrato の単語辞書では，単語文字列をユニコードのコードポイント毎に分割した文字列として辞書で管理している．

二つ目は，トライを用いた言語モデルである．言語モデルのデータ構造として，Backword Suffix Tree [2] や Reverse Trie [14] がある．これらは共通して単語列を保存するラベル付き木を基本としており，単語を ID として表し，ID を遷移ラベルとして保存する．Yasuhara ら,Norimatsu ら [15,16] の DALM では，Backword Suffix Tree および Reverse Trie をダブル配列で実装することで，高速なクエリ応答を可能とする言語モデルを実現した．

三つ目は，バイト文字列の動的辞書である．Yoshinaga ら [17] の Cedar[*4]はダブル配列を用いた効率的な動的辞書の実装であり，Yoshinaga らの提案する自己適応分類器の共通分類問題の管理に Cedar を用いている．

これらのダブル配列の実装はいくつか設定が異なる．具体的には，辞書が動的か静的か，表現するラベル付きグラフ，遷移ラベルの設定に基づくアルファベットサイズの見積もり，遷移関数の選択 (`plus-op` か `xor-op` か)，である．これらの対応関係を表 3.1 にまとめる．

3.4.1 実験環境

全ての実験は以下の計算機環境で行う，CPU: Intel Xeon Processor E5540(8M Cache, 2.53 GHz, 5.86 GT/s Intel QPI), RAM: 256GB, OR: CentOS Linux release 7.9.2009.

3.4.2 データセット

実験で用いる全てのデータセットを以下に示す．

- `UNIDIC`: 国立国語研究所の現代書き言葉辞書[*5]
- `JAWIKI`: 日本語 Wikipedia タイトル集合[*6]
- `ENWIKI`: 英語 Wikipedia タイトル集合[*7]

[*2] `https://taku910.github.io/mecab/`
[*3] `https://github.com/daac-tools/vibrato`
[*4] `https://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/`
[*5] `https://clrd.ninjal.ac.jp/unidic/index.html`
[*6] `https://dumps.wikimedia.org/jawiki/`
[*7] `https://dumps.wikimedia.org/enwiki/`

表 3.2: 静的ユニコード辞書の実験に用いるデータセット

	ファイルサイズ (MiB)	キー数	バイト数/キー数	文字数/キー数
UNIDIC	8.4	674,927	12.0	4.03
JAWIKI	47.1	2,159,582	21.9	8.31

- GEONAMES: GeoNames dump[*8]から抽出した地名集合
- INDOCHINA: indochina ドメイン上でクロールし得た URL 集合[*9]
- LUBM: Lehigh University Benchmark [18] の生成した RDF データセットから抽出した URI 集合[*10]
- 1BIL: 1 Billion Word Language Model Benchmark [19] の，言語モデルベンチマークのために用意されたデータセット[*11]
- PATENT: NTCIR Project[*12]が公開した日本公開特許公報全文の 1993 から 2005 年のコーパス[*13]

3.4.3 比較手法

従来法を base，提案手法を BP と表記する．それぞれ，ダブル配列の構築に，base は ELM を用いた XCHECK(Algorithm1)，BP は bit-parallel XCHECK を用いている．

3.4.4 静的ユニコード辞書での実験

比較手法の実装は，vibrato の単語辞書である crawdad[*14](v0.4.0) を基に提案手法を追加で実装した．データセットには，UNIDIC，JAWIKI を用いる．UNIDIC は元は N-gram データセットだが，N-gram の中に出現する単語のみを抽出した単語集合をデータセットとしている．JAWIKI は記事タイトル集合をそのまま用いている．データセットの詳細を表 3.2 に示す．

crawdad での構築手順は，データセットからトライの各ノードの遷移ラベル集合を得た後，各ノードに対して一度づつ XCHECK を行い要素の配置を決定していく．

実験結果を表 3.3 に示す．いずれの結果も BP の方が高速であり，特に UNIDIC で 23% まで効果的に構築時間を削減できている．

[*8] *asciiname* column, http://download.geonames.org/export/dump/
[*9] indochina-2004,https://law.di.unimi.it/index.php
[*10] DS5, https://exascale.info/projects/web-of-data-uri/
[*11] https://www.statmt.org/lm-benchmark/
[*12] https://research.nii.ac.jp/ntcir/index-ja.html
[*13] https://research.nii.ac.jp/ntcir/permission/ntcir-10/perm-ja-PatentMT.html
[*14] https://github.com/daac-tools/crawdad

表 3.3: crawdad によるダブル配列の構築時間

	base	BP	(BP/base)
UNIDIC	3.647	0.837	(0.230)
JAWIKI	2.356	1.769	(0.751)

表 3.4: 言語モデルデータセットの詳細

	単語数	N-gram エントリ数
1BIL	2,425,341	1,139,186,177
PATENT	1,170,894	687,106,803

表 3.5: DALM によるダブル配列の構築時間 (秒)

	base	BP	(BP/base)
1BIL	174,201	34,043	(0.195)
PATENT	9,871	2,535	(0.257)

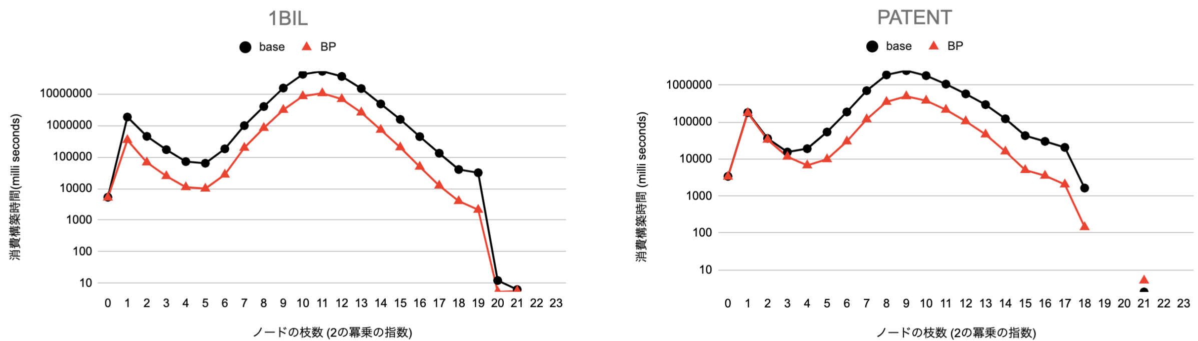


図 3.5: DALM の ReverseTrie のノード毎に要する XCHECK の計算時間を，ノードの枝数毎に合計した総消費時間．枝数 x が $2^{k-1} < x \leq 2^k$ なら横軸を k に計上している．

3.4.5　静的言語モデルでの実験

比較手法の実装は，DALM[*15](3.0.0) による Reverse Trie [14] の実装を基に提案手法を追加で実装した．データセットには 1BIL,PATENT を用いる．データセットの詳細を表 3.4 に示す．

図 3.5 は，Reverse Trie のノードごとの XCHECK に要する時間を，そのノードが持つ枝数 (遷移ラベルの数) ごとに合計した累計消費時間を表している．1BIL では枝数 2^1，PATENT では枝数 2^3 を超えたあたりから BP が高速に動作していることが分かる．それ以上の枝数では BP の構築速度は base の約 5 倍から 10 倍を維持している．これは提案手法がビットレベル並列化による一定倍率の計算量改善が出来たことを表している．枝数 $2^0 = 1$ でほぼ変化がないのは，サイズ 1 の遷移ラベル集合 $C = \{c\}$ の XCHECK は空要素連結リストの先頭位置 e_0 から $x \leftarrow \mathtt{iop}(e_0, c)$ と容易に計算できるため，提案手法による改善効果が無いからである．

DALM の構築時間の結果を表 3.5 に示す．提案手法は構築時間を 19.5 – 25.7% まで大幅に削減

*15 https://nowlab.github.io/DALM/

表 3.6: バイト文字列データセットの詳細

	ファイルサイズ (MiB)	キー数	バイト数/キー数
ENWIKI	365.7	16,871,019	21.7
GEONAMES	642.7	7,414,866	14.5
INDOCHINA	179.1	12,358,056	86.7
LUBM	3349.3	52,616,588	63.7

している．特に 1BIL の構築時間は 48.4 時間から 9.5 時間まで小さくなっており，利用者への時間的恩恵が非常に大きい．

表 3.5 から分かるように，元々 DALM の構築は数時間から数日を必要とするほど時間のかかる計算処理である．Bogoychev ら [20] の調査では DALM の構築時間は他の言語モデルより大幅に大きいとしており，DALM の実用性における最大のボトルネックといえる．そのため，今まで DALM の構築高速化のために様々な研究が行われてきた [15, 16, 21–23]．しかしそれらの手法は構築されるダブル配列の要素の配置の大幅な変更や，複数プロセスを利用した並列処理を必要とする．ダブル配列の実行速度は要素の配置に依存するヒューリスティックであるため，構築を有利にするための配置の変更は他のクエリ速度へ影響してしまう側面を持ち，性能の分析を難しくする．また，複数プロセスによる並列処理は計算リソースに十分な余裕がある状況でしか利用できない．それに対し，提案手法は XCHECK 内部の演算の変更のみで高速化を実現しており，構築される要素の配置を変更せず，またプロセス並列化を必要としない．XCHECK 演算以外への副作用を持たずに構築時間を 20% 程度まで削減できたことは，DALM のボトルネックを大幅に解消したといえる．

3.4.6 動的バイト文字列辞書での実験

比較手法の実装は，Cedar と同じ仕様の MP トライを我々が再現した実装を用いた．我々の実装の詳細は 3 章 4.5.1 節に記載している．

データセットには ENWIKI,GEONAMES,INDOCHINA,LUBM を用いる．データセットの詳細を表 3.6 に示す．

実験方法は，あらかじめランダム順に並べ替えたデータセットを 1 キーづつ挿入し，各データセットサイズにおける平均挿入時間を計測した．実験結果を図 3.6 に示す．まず全体の傾向として，挿入済みのキー数が増えるほど挿入は遅くなる．これは主にダブル配列のサイズが大きくなることに伴う，ランダムアクセス時のキャッシュヒット率の低下が原因である．データセット毎に最大キー数挿入時の BP の性能を見ると，GEONAMES,LUBM で高速化できており，ENWIKI で base と同程度，INDOCHINA で悪化している．

LUBM ではキー数 2^{21} 程度までは base の方が高速だが，それ以降は BP が高速である．INDOCHINA や LUBM などの URI 集合では，MP トライ以上に多くのシングルノードが存在し，ダ

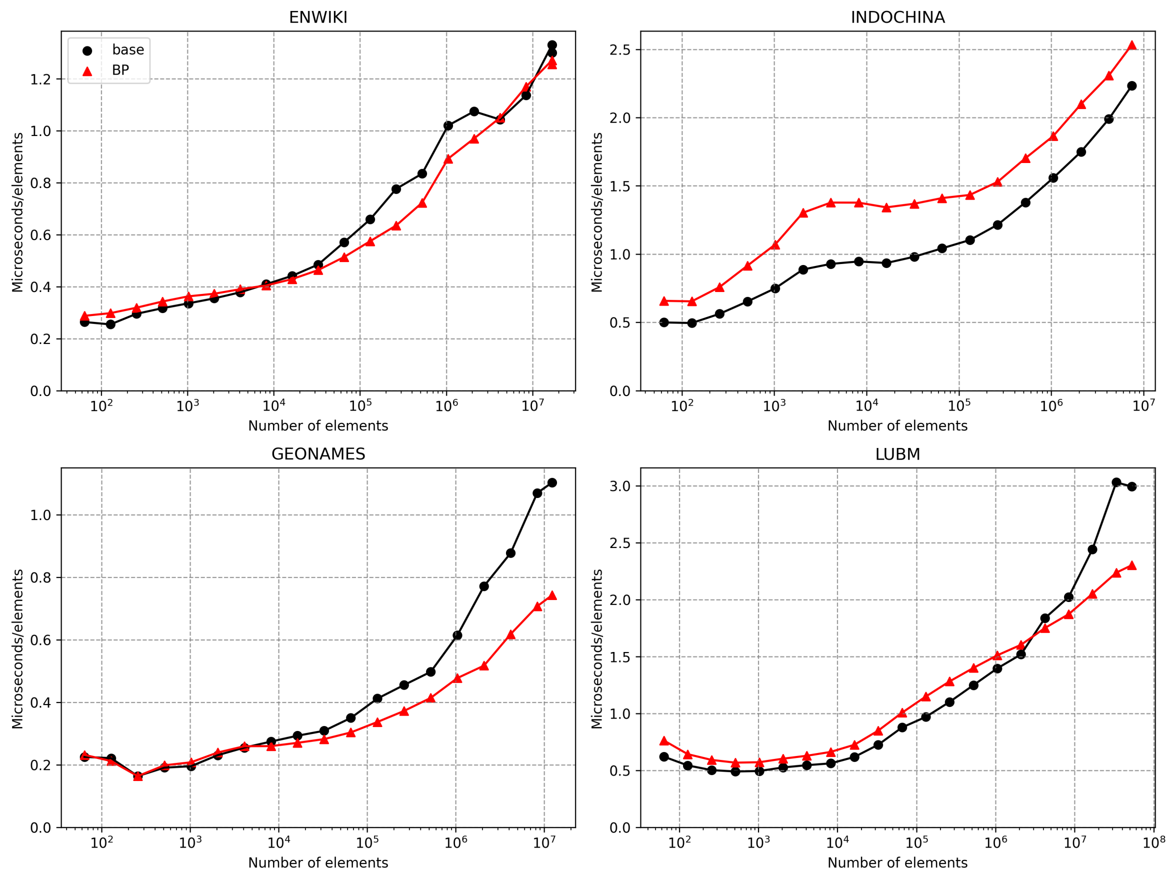


図 3.6: バイト文字列辞書へのランダム順キー挿入時のキー当たり挿入時間

ブル配列の未使用要素でできる隙間を埋めるように働く．この場合，XCHECK は直ちに解を見つける可能性が高いため，BP の効果が薄くなる．しかし，BP ではダブル配列と bitset のランダムアクセスが発生するため，キャッシュヒット率の低下の影響が実行速度に反映されていると思われる．ただし，LUBM のように巨大なデータセットを用いた場合，BP が有利になる空要素が点在するような分布が生まれていると思われる．

GEONAMES では，キー数が増加するほど BP が高速になる傾向がみれる．GEONAMES のような地名は接頭辞を共有する単語が現れにくい単語傾向があるため，MP トライにシングルノードがあまりないことで空要素の隙間が多くなり，BP が有利に働くと思われる．

ENWIKI では，キー数 2^{22} 程度まで BP が高速だが，それ以降はほぼ同程度である．

結論として，BP のバイト文字列集合に対する性能はデータセットの特徴に左右され，必ずしも BP が有効に働くとは限らない．

3.4.7 実験的評価の総括

BP はダブル配列の構築時間を最良の場合約 5 倍以上高速化し，最悪の場合でも 0.88 倍と従来より少し低い程度に留めている．特にマルチバイト文字や単語 ID 列辞書のようなアルファベットサイズの大きい辞書に対する高速化の効果が非常に高く，DALM のような大規模なトライの実装にダブル配列を用いる場合の実用性を大きく向上させている．

BP の効果が薄い状況を分析する．アルファベットサイズが小さい場合，XCHECK の解が直ちに見つかる可能性が高く，BP の効果が薄くなる．またバイト文字列辞書で BP の効果が薄かったのは，URI 集合のように接頭辞の共通化部分が大きく，トライに大量のシングルノードが存在する状況である．シングルノードはダブル配列の空要素で出来る隙間を埋めるように働くため，これも XCHECK の解が直ちに見つかり易い配置を作るため，BP の効果が薄くなると考えられる．また，遷移関数に `xor-op` を用いたことも，ビット演算の計算量の多さが不利に働いたと言える．

第 4 章

ダブル配列によるパトリシアトライ

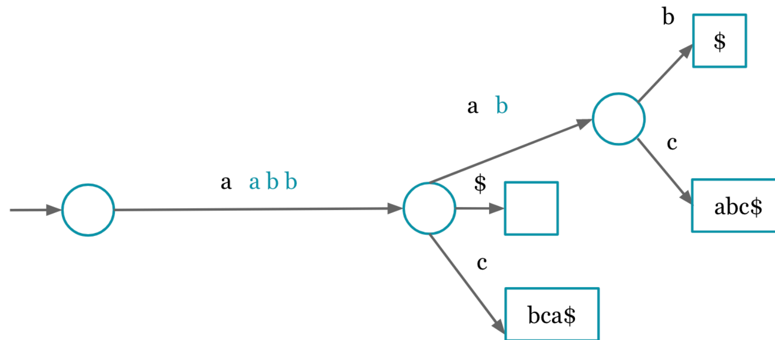


図 4.1: キー集合 {"aabb","aabbabb","aabbabcabc","aabbcbca",} を表現するパトリシアトライ

本章ではダブル配列によりパトリシアトライを表現する方法を提案する．また，提案手法を用いて実装した動的辞書の性能を評価する．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は松本ら [24] により提案されたが，本章では第 3 章で提案した bit-parallel XCHECK を用いた性能評価も行う．

ダブル配列を用いた辞書の実装は，基本的に MP トライを BASE,CHECK,TAIL で表現し実装される (図 2.4)．これを MP ダブル配列と呼ぶ．MP ダブル配列では，キーが一意になるノード以降の遷移ラベルを文字列として表現する．これにより，通常のトライと比較して大幅にノード数を削減できる．また文字列でそのまま保存することは，検索時にシーケンシャルアクセスで文字同士の比較が出来るため，検索効率も良くなる．しかし，MP トライの内部には子を一つしか持たないノードが多くある．

パトリシアトライ [11] は，トライの子を一つしか持たないノードを削除し，それらの連続する遷移ラベルを文字列として表現したものである．図 4.1 は図 2.4 と同じキー集合を表現するパトリシアトライである．キー集合 S に対するパトリシアトライのノード数は $2|S|$ である．パトリシアトライのノード数は定義より明らかに MP トライのノード数以下であり，また遷移ラベルのより多くを文字列で表現出来るため，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のシーケンシャル比較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．MP 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 TAIL 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．

ここで，伊藤 [25] のアイデアが解決策の参考になる．伊東は，静的な辞書の実装として，文字列集合を Minimal Directed Acyclic Word Graph(MDAWG) 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 [26] 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

表 4.1: LABEL と LEAF の値に対応する BASE 値の役割

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

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，検索時に文字列と BASE を連続するアドレスで得ることができ，またノード u の遷移ラベルへのアドレスを BASE[u] 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，伊藤の手法を応用した方法で遷移ラベルを表現することで，ダブル配列によりパトリシアトライを実現する方法を提案する．また，辞書へのキーの追加によって遷移ラベルの内部で分岐が発生した際の，効率の良い更新アルゴリズムを提案する．

4.1 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．例えば，BASE 配列の要素を 4 バイト，POOL の要素を 1 バイトで表現していたならば，$W = 4$ である．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 を遷移ラベルの前に保存することで b_t のアドレスをポインタから直接得られるため，辞書を更新する際の b_t の更新を容易にしている．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．頂点 i の BASE が POOL へのポインタかどうかを区別するため，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}) を導入する．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 の値の役割を表 4.1 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 4.2 に示す．図においてアルファベットは $\{\text{`a'} = 0, \text{`c'} = 1, \text{`e'} = 2, \text{`i'} = 3, \text{`l'} = 4\}$ 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[*1]．

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

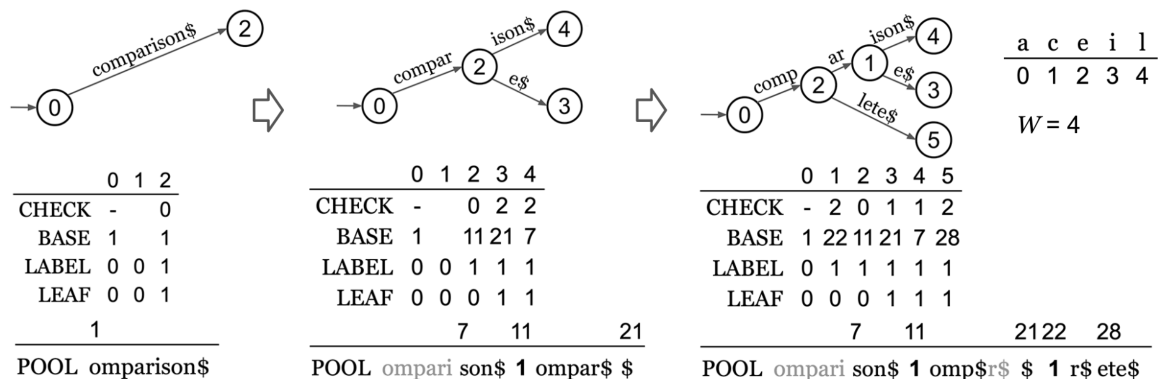


図 4.2: 文字列 “comparison”,“compare”,“complete” を順に挿入した際のパトリシア-ダブル配列の過程

4.2 提案手法の検索

提案手法の検索アルゴリズムを Algorithm 4 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．

$$\textsc{Base}(s) = \begin{cases} \text{BASE}[s] & (\text{LABEL}[s] = 0) \\ \text{POOL}[\text{BASE}[s], \text{BASE}[s] + W) & (\text{otherwise}) \end{cases} \tag{4.1}$$

関数 $\textsc{Goto}(s, c)$ はノード s から t の遷移 (式 (2.1)) 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 4 パトリシア-ダブル配列の検索アルゴリズム

```
1: function Goto(s, c)
2:     t ← Base(s) + c
3:     if CHECK[t] ≠ s then
4:         return False
5:     return t
6:
```

```
7:  function SEARCH(key)
8:      s ← 1
9:      keypos ← 1
10:     while keypos ≤ |key| and LEAF[s] = 0 do
11:         t ← GOTO(s, key[keypos])
12:         if t = False then
13:             return False
14:         if LABEL[t] = 1 then
15:             u ← BASE[t] + W
16:             while POOL[u] ≠ '$' do
17:                 if keypos > |key| or
18:                     POOL[u] ≠ key[keypos] then
19:                     return False
20:                 keypos ← keypos + 1
21:                 u ← u + 1
22:         else
23:             keypos ← keypos + 1
24:         s ← t
25:     if keypos = |key| then
26:         if GOTO(s, '$') = False then
27:             return False
28:     else
29:         u ← BASE[s]
30:         while keypos ≤ |key| do
31:             if POOL[u] ≠ key[keypos] + 1 then
32:                 return False
33:             u ← u + 1
34:         if POOL[u] ≠ '$' then
35:             return False
36:     return True
```

4.3 空間効率の良いキーの追加処理

まず従来法と共通して用いる関数を定義する．

- CHILDREN(s): 頂点 s の子へのラベル集合 $C = \{c_i : \mathrm{CHECK}[\mathtt{op}(\mathrm{BASE}[s], c_i)] = s\}$ を

返す．

- $\textsc{Grow}(s, c)$: 葉 s から文字 c により遷移される子を追加する．$\text{BASE}[s] \leftarrow \text{XCHECK}(\{c\})$ とした後，$\text{CHECK}[\mathtt{op}(\text{BASE}[s], c)] \leftarrow s$ を設定する．
- $\textsc{ResolveCollision}(s, t, c)$: 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 の内，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 XCHECK 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，$\text{BASE}[v] \leftarrow b$ とする．
- $\textsc{InsertEdge}(s, c)$: 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．$\text{EMPTY}[\mathtt{op}(\text{BASE}[s], c)] = \text{False}$ の場合, $\textsc{ResolveCollision}(s, \text{CHECK}[\mathtt{op}(\text{BASE}[s], c)], c)$ を行い，最後に，$\text{CHECK}[\mathtt{op}(\text{BASE}[s], c)] \leftarrow s$ とする．
- $\textsc{InsertInBc}(s, \mathit{suffix})$: 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 の末尾は終端文字 '\$' である．まず $\textsc{InsertEdge}(s, \mathit{suffix}[0])$ を実行する．$l \leftarrow |\text{POOL}| + 1$ を得た後，残りの文字列 $\mathit{suffix}[1, |\mathit{suffix}|)$ を TAIL の末尾に追加する．そして $\text{LEAF}[\text{BASE}[s] + \mathit{suffix}[0]] \leftarrow 1$ と $\text{BASE}[\text{BASE}[s] + \mathit{suffix}[0]] \leftarrow l$ を設定する．
- $\textsc{InsertInTail}(s, \mathit{tailpos}, \mathit{suffix})$: 葉 s への遷移の接尾辞の $\mathit{tailpos}$ 文字目から，文字 suffix による分岐を生成する．$i \leftarrow \text{BASE}[s]$ を得て，$\text{POOL}[i, i + \mathit{tailpos})$ の文字列から一文字ずつ $\textsc{Grow}$ 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，$\textsc{Grow}(s', \text{POOL}[\mathit{tailpos}])$ と $\text{CHECK}[\text{BASE}[s'] + \text{POOL}[\mathit{tailpos}]] \leftarrow s'$ 及び $\text{BASE}[\text{BASE}[s'] + \text{POOL}[\mathit{tailpos}]] \leftarrow \mathit{tailpos}{+}1$ として接尾辞の先頭インデックスを移動する．その後 $\textsc{InsertInBc}(s', \mathit{suffix}[1, |\mathit{suffix}|) +$ '\$') を実行する．

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 $\text{TAIL}[j]$ 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．

続いて，提案手法のキーの追加処理を Algorithm 5, 6 に示す．関数 $\textsc{Insert}(\mathit{key})$ 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 $\textsc{InsertInBc}$ と $\textsc{InsertInTail}$ 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 $\textsc{InsertInInternalLabel}$ 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 $\textsc{LeastPreffix}(\mathit{poolhead}, \mathit{poolpos})$ は POOL 内での L の先頭位置 ($\mathit{poolhead}$) と R の先頭位置 ($\mathit{poolpos}$) を入力とし，$|L| < |R|$ の場

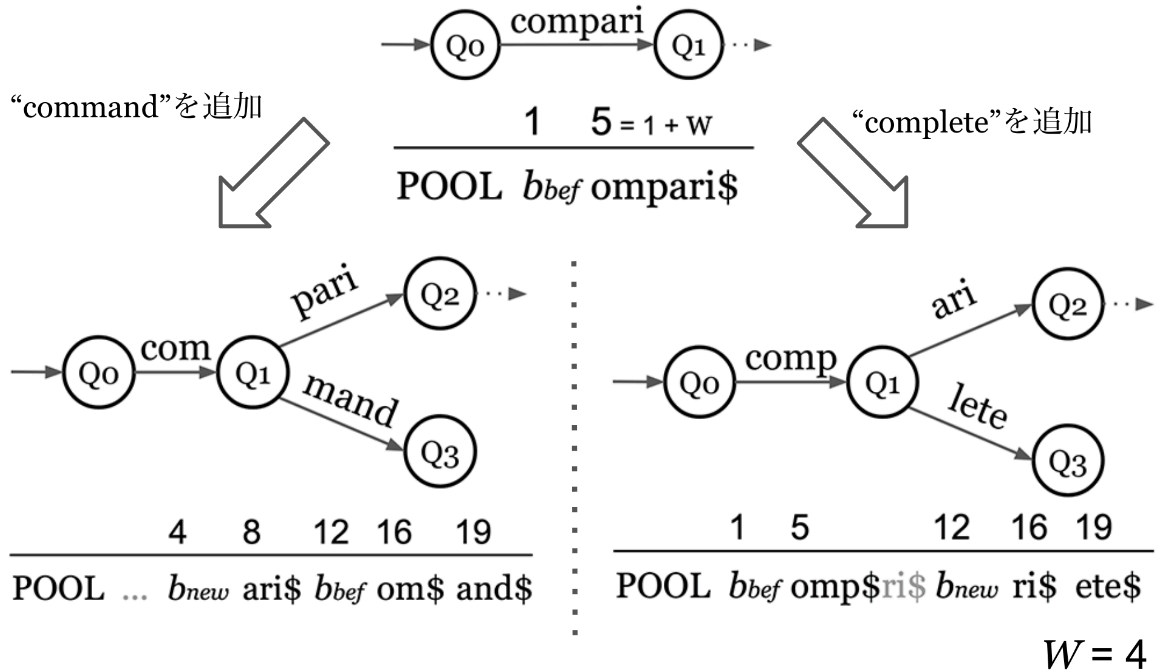


図 4.3: 内部遷移ラベル “compari” 上で，“command” か “complete” による分岐が行われる場合の部分グラフと POOL 配列の変化

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 4.3 に示す．図 4.2 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

4.4 理論的評価

BASE の要素には BASE 値か TAIL のインデックスが保存されるため，少なくとも $\max\{\lceil \log_2 N \rceil, \lceil \log_2 |TAIL| \rceil\}$bit が必要である．ダブル配列の実際の実装では，実行速度を重視

Algorithm 5 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 4
 3:         INSERTINBC(s, key[keypos])
 4:     lines (14-18) of Algorithm 4
 5:             INSERTININTERNALLABEL(s, u, key[keypos])
 6:     lines (20-26) of Algorithm 4
 7:         INSERTINBC(s, Empty)
 8:     lines (28-31) of Algorithm 4
 9:           INSERTINTAIL(s, u, key[keypos])
10:     lines (33-34) of Algorithm 4
11:         INSERTINTAIL(s, u, Empty)
12:     line (36) of Algorithm 4
```

して BASE,CHECK の要素サイズは固定長で設定されることが多い．よって簡単のため，BASE と CHECK の要素の記憶量をそれぞれ z ビットとする．LABEL 配列は各要素 1bit である．POOL 配列の各要素は少なくとも $\lceil \log_2 \sigma \rceil$ 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 $\{s_1, s_2, ..., s_{m+1}\}$ 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 z ビットを要するため，上記の場合には提案手法により $(m-1)(2z - \lceil \log_2 \sigma \rceil) + z - \lceil \log_2 \sigma \rceil$ ビット削減されることになる．これは，$z > \log \sigma \wedge m \geq 2$ 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．

検索時間について，提案手法では MP トライで遷移の連続として表現していたラベルの一部を文字列で表現するため，文字列に変換出来る遷移が多いほど検索が早くなると考えられる．

第 3 章で提案した bit-parallel XCHECK との相性についても評価する．まずパトリシアトライの特徴として，全てのノードが二つ以上の枝を持つというところにある．MP トライでは一つしか枝を持たないシングルノードが多数存在した．シングルノードは，ダブル配列において隙間を埋める役割を果たすため，シングルノードの多い MP トライでは空要素の先頭要素の後ろには十分な数の空要素が存在する状態になり，従来の XCHECK でも高速に動作する．これは節 3.4.6 で bitparallel-check の効果が非常に薄かったことからも裏付けられる．しかし，パトリシアトライではシングルノードが現れないことで空要素が点在する状況が生まれやすく，パトリシアトライは MP トライより bit-parallel XCHECK の効果が現れやすいと予想される．

Algorithm 6 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 LeastPrefix(poolhead, poolpos)
 2:     i ← 0
 3:     while POOL[poolpos + i] ≠ '$' do
 4:         if i ≥ poolpos − poolhead then
 5:             return True
 6:         i ← i + 1
 7:     return False
 8: function InsertInInternalLabel(s, poolpos, suffix)
 9:     u ← BASE[s] + W
10:     beforeB ← Base(s)
11:     targetC ← POOL[poolpos]
12:     newB ← XCHECK({targetC, suffix[0]})
13:     t ← newB + targetC
14:     C ← Children(s)
15:     if LeastPrefix(u, poolpos) then
16:         if poolpos > u then
17:             l ← |POOL|
18:             POOL の末尾に文字列 {newB, POOL[u, poolpos), '$'} を追加
19:             BASE[s] ← l
20:         else
21:             LABEL[s] ← 0
22:             Base(s) ← newB
23:         if POOL[poolpos + 1] ≠ '$' then
24:             BASE[t] ← poolpos + 1 − W
25:             LABEL[t] ← 1
26:         else
27:             LABEL[t] ← 0
28:         Base(t) ← beforeB
```

4.5 実験的評価

提案手法であるパトリシアトライを用いたダブル配列による辞書の性能を，実社会で用いられるデータセットを基に評価する．実験環境は 3.4.1 節の実験的評価時と同じである．データセットは 3.4.6 節と同じものを用いる．

29: **else**

30: **if** POOL[$poolpos+1$] $\neq$ '\$' **then**

31: $l \leftarrow |\text{POOL}|$

32: POOL の末尾に文字列 {$beforeB$, POOL[$poolpos+1$] から '\$' までの文字列 } を追加

33: BASE[t] $\leftarrow l$

34: LABEL[t] $\leftarrow 1$

35: **else**

36: LABEL[t] $\leftarrow 0$

37: $\textsc{Base}(t) \leftarrow beforeB$

38: POOL[$poolpos$] $\leftarrow$ '\$'

39: $\textsc{Base}(s) \leftarrow newB$

40: $\textsc{InsertEdge}(s, targetC)$

41: **for** $c \in C$ **do**

42: CHECK[$beforeB + c$] $\leftarrow t$

43: $\textsc{InsertInBc}(s, suffix[1, |suffix|])$

4.5.1 実装

実装する辞書の仕様は，バイト文字列で表現される文字列をキーとした辞書を想定した．細かい仕様は Yoshinaga ら [17] の Cedar の実装とほぼ同じとした．具体的には以下の通りである:

- BASE,CHECK に保存できる最大値は $2^{31}-1$
- 文字は 1 バイトで表現し，POOL 配列の要素サイズは 1 バイト
- 遷移関数は `xor-op`
- ELM [6]，BLM [8] を用いる．BLM におけるブロックをリンクから削除するための XCHECK 失敗回数の閾値は 1

`xor-op` を用いるのは，BLM を用いるには `xor-op` を前提としているからである．

4.5.2 パトリシアトライの評価

ダブル配列パトリシアトライの評価は，ダブル配列 MP トライとの比較で行う．また，キー挿入時間の計測を行う実験では，第 3 章で提案した bit-parallel XCHECK との組み合わせも評価する．比較手法の表記を以下に示す．

- MP: MP トライによるダブル配列 (Cedar と同等)

表 4.2: データセットのキーをランダム順に全て挿入した時のダブル配列の BASE,CHECK および POOL 配列の配列長と有効要素数.

	BASE,CHECK			POOL		
	配列長	有効要素数	$\frac{\text{有効要素数}}{\text{配列長}}$	配列長	有効要素数	$\frac{\text{有効要素数}}{\text{配列長}}$
ENWIKI						
MP	41,168,128	41,167,920	1.00	217,587,574	178,643,600	0.82
PAT	29,671,168	25,369,760	0.86	256,705,796	209,522,748	0.82
GEONAMES						
MP	14,386,176	14,068,744	0.98	79,673,371	66,520,471	0.83
PAT	13,433,344	11,257,314	0.84	89,489,422	73,494,369	0.82
INDOCHINA						
MP	22,572,800	22,572,605	1.00	175,390,031	153,143,419	0.87
PAT	12,680,448	10,114,327	0.80	196,647,883	170,714,565	0.87
LUBM						
MP	85,989,632	75,934,829	0.88	405,590,315	329,655,489	0.81
PAT	86,738,688	60,319,246	0.70	432,977,690	351,243,716	0.81

- MP-bp: MP と bit-parallel XCHECK の組み合わせ
- PAT: パトリシアトライによるダブル配列
- PAT-bp: PAT と bit-parallel XCHECK の組み合わせ

ダブル配列の要素の使用状況

データセットのキーをランダム順に全て挿入した時のダブル配列の BASE,CHECK および POOL 配列の配列長と有効要素数を表 4.2 に示す．BASE,CHECK における有効要素数はトライのノード数と同じ意味である．PAT のノード数は MP より小さく，`ENWIKI` で 62%，`GEONAMES` で 80%，`INDOCHINA` で 45%，`LUBM` で 79% である．しかし，配列長当たりの有効要素数，つまり配列の使用率は PAT の方が少し低く，若干メモリ効率が悪い．POOL 配列の有効要素数は PAT の方が少し大きく，`ENWIKI` で 117%，`GEONAMES` で 110%，`INDOCHINA` で 111%，`LUBM` で 107% である．こちらは配列の使用率に殆ど違いが無い.

キーの検索時間

キーの検索時間の実験結果を図 4.4 に示す．PAT は全ての状況で MP より高速である．特に `INDOCHINA,LUBM` の URI 集合での PAT の高速化の影響が大きい．URI 集合は，共通の接頭辞が現れやすい文書構造であり，かつ共通化される接頭辞のキー長も長くなり易い．そのことから，パトリシアトライによる文字列ラベルとして表現出来る箇所が多くなり，検索に有利に働いていると

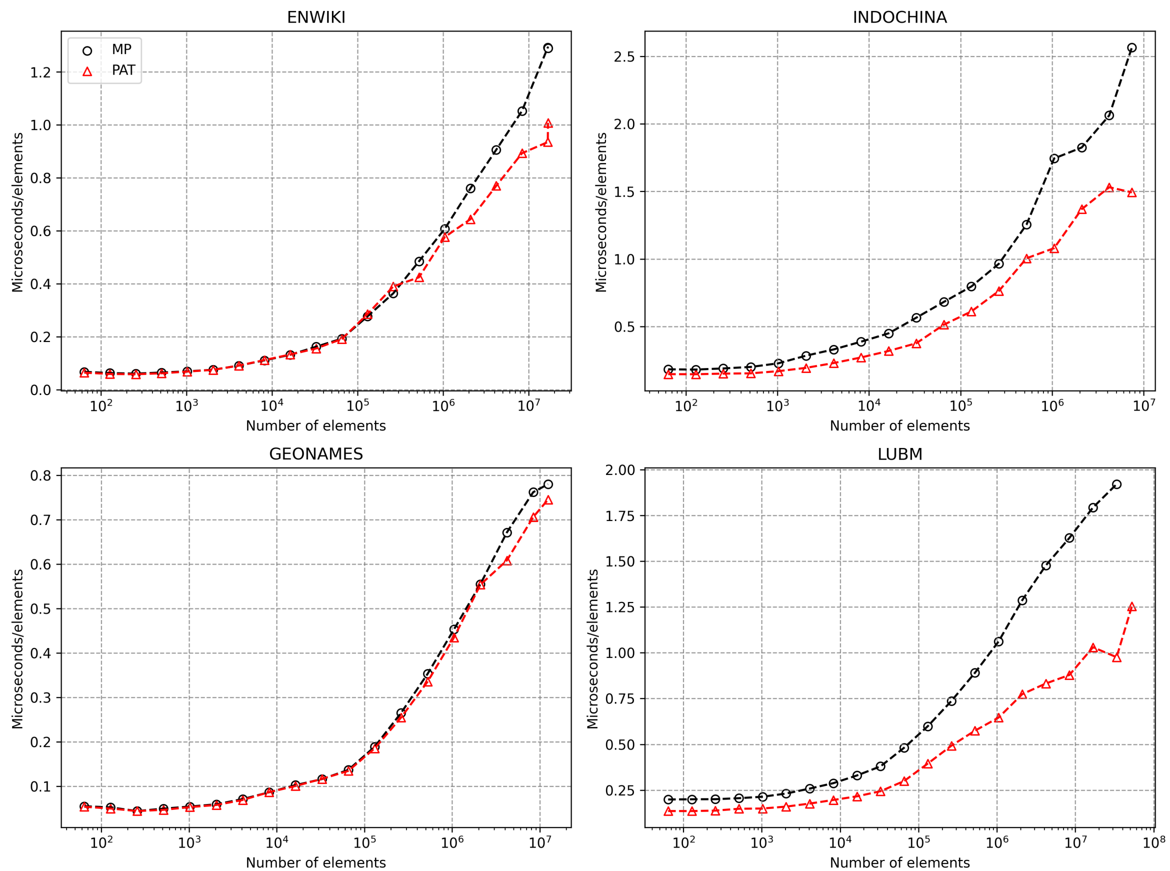


図 4.4: 動的辞書へのランダム順キー挿入時のキー当たり検索時間.

考えられる．一方 GEONAMES ではほぼ同程度である．地名は接頭辞に長い共通文字を含みにくいことから文字列によるラベルを持ちにくいからだと考えられる．

キーの挿入時間

キー挿入時間の実験結果を図 4.5 に示す．MP と PAT を比較すると，ENWIKI,GEONAMES では遅くなっており，LUBM では速くなっている．INDOCHINA ではキー数 2^{22} 程度で MP が高速になるような逆転をしている．遅くなる原因として，パトリシアトライにはシングルノードが現れないことから，空要素が点在するようになり XCHECK に時間がかかるようになる [6]．一方速くなる原因として，ノード数が減ることが，XCHECK の呼び出し回数自体の減少が考えられる．特に LUBM のようなキー長の長いデータセットでは，大量のシングルノードに対する XCHECK が呼び出されてしまう．また，ダブル配列の配列長が大きくなることによるランダムアクセスのキャッシュヒット率の低下も要因の一つとして考えられる．

bit-parallel XCHECK の影響も大きく反映されている．特に，PAT においては全ての状況で 30% 以上の大幅な高速化を実現しており，MP トライの時の bit-parallel XCHECK の評価とは異なる．これは，パトリシアトライにシングルノードが現れないことから XCHECK 単体の平均計

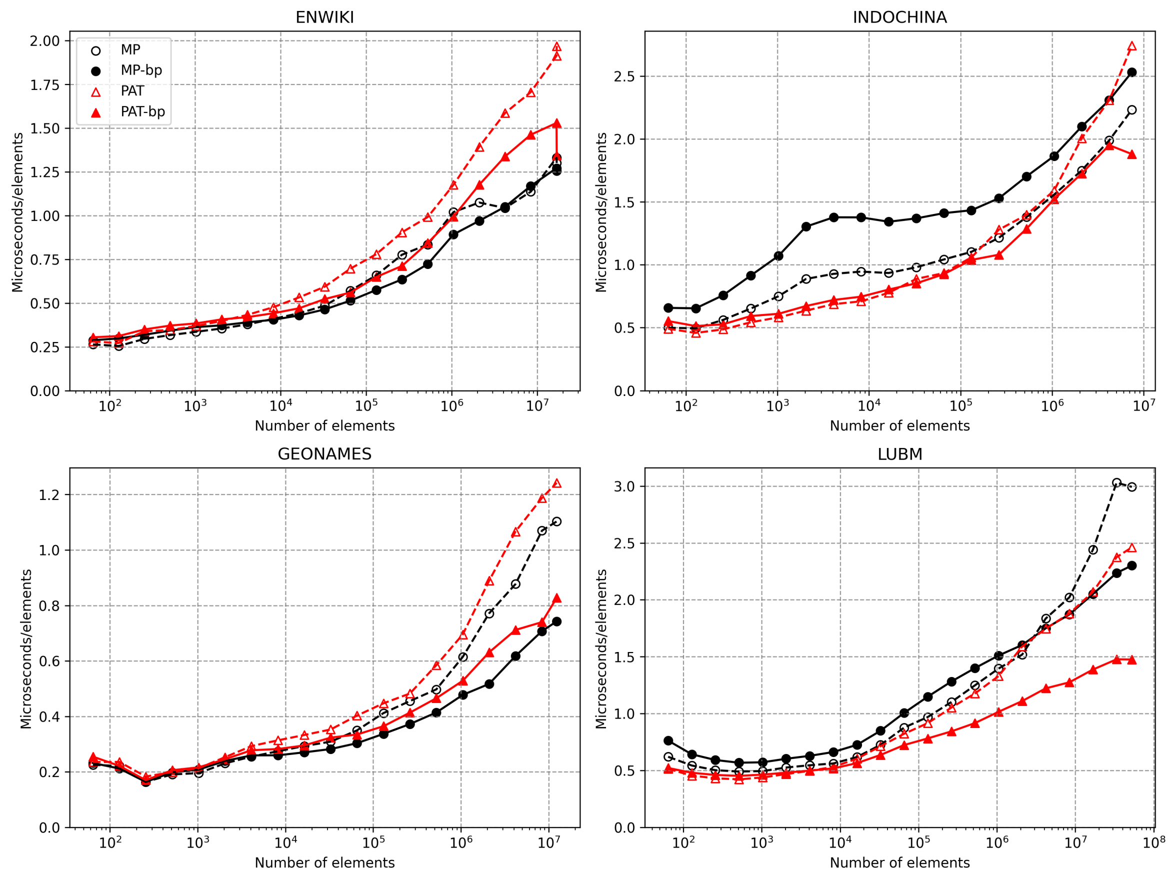


図 4.5: 動的辞書へのランダム順キー挿入時のキー当たり挿入時間.

表 4.3: 動的辞書へのランダム順キー挿入時の最大メモリ消費量.

	MP	PAT	(PAT/MP)
ENWIKI	660,279,296	571,686,912	0.87
GEONAMES	261,541,888	234,057,728	0.90
INDOCHINA	446,414,848	351,858,688	0.79
LUBM	1,379,713,024	1,344,651,264	0.97

算量が基本的に大きくなるため，bit-parallel XCHECK が効果的に働いたと思われる．

つまり，パトリシアトライは MP トライよりキー挿入時間が大きくなる可能性があるが，bit-parallel XCHECK と併用することでキー挿入時間の悪化を抑制できる．

メモリ消費量

最大メモリ消費量を表 4.3 に示す．いずれのデータセットでも PAT は MP より少ないメモリ消費量を維持している．特に INDOCHINA ではメモリ消費量を 21% 削減しており，シングルノードを

遷移ラベルに変更することのメモリ削減効果が大きく反映されている．

第 5 章

おわりに

本論文ではラベル付きグラフのデータ構造であるダブル配列 [4,5] の時間計算量を改善する二つの手法を提案した．

一つ目に，ダブル配列の構築の基盤となる演算である XCHECK をビットレベル並列化により高速化する手法，bit-parallel XCHECK，を提案した．この手法は，ダブル配列の構築時間を最大 5 倍高速化することを可能にした．特に従来法ではダブル配列の構築に多くの時間を要する，アルファベットサイズの大きい状況で顕著な高速化を実現している．また，提案手法は高速化のために構築されるダブル配列の要素配置を変更する必要は無く，単一プロセスのまま動作するため，ダブル配列を用いるあらゆる状況において副作用無く利用できる．

二つ目に，ダブル配列でパトリシアトライ [11] を表現する手法，ダブル配列パトリシアトライ [24]，を提案した．パトリシアトライを表現するためには文字列のラベルを表現する必要があるため，伊藤 [25] の技法を基にダブル配列の遷移ラベルが文字列ラベルを持つような表現を可能にするデータ格納方法を提案した．具体的には，文字列ラベルとノードの BASE 値を連続したメモリに保存し，その先頭アドレスをダブル配列内で管理するというものである．パトリシアトライによる表現は，ノード数の削減に伴うメモリ消費量の削減と，探索時のシーケンシャルアクセスの増加によるクエリ速度の向上を実現した，ただし，キーの挿入に関しては有利に働く点と不利に働く点があり，結果として若干の挿入時間増加を招く状況が多く見られた．

しかし，bit-parallel XCHECK の高速化の効果がダブル配列パトリシアトライに非常に効果的に働くことも分かった．これにより，ダブル配列パトリシアトライの構築と bit-parallel XCHECK を併用することで，パトリシアトライによるキー挿入時間の悪化を打ち消すような結果が確認できた．ダブル配列パトリシアトライと bit-parallel XCHECK を併用して実装された動的バイト文字列辞書は，従来の MP トライの辞書に対しし常に同程度以上の性能を示し，特に URI 集合に対しては，検索時間の約 30% の削減，メモリ消費量の約 21% の削減，キー挿入時間の約 55% の削減を実現した．この研究成果として実装された動的バイト文字列辞書は，MIT ライセンスの元で公開した[*1]．

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

参考文献

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Edward Fredkin. Trie memory. *Communications of the ACM*, Vol. 3, No. 9, pp. 490–499, sep 1960.

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

[4] J.-I. Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077, 1989.

[5] Jun - IchiAoe, Katsushi Morimoto, Takashi Sato. An efficient implementation of trie structures. *Software: Practice and Experience*, Vol. 22, No. 9, pp. 695–721, sep 1992.

[6] Kazuhiro Morita, Masao Fuketa, Yoshihiro Yamakawa, Jun - ichiAoe. Fast insertion methods of a double - array structure. *Software: Practice and Experience*, Vol. 31, No. 1, pp. 43–65, jan 2001.

[7] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. Cache-efficient double-array. In *Proceedings of the 5th Forum on Information Technology (FIT)*, pp. 71–72, 2006.

[8] 矢田晋, 田村雅浩, 森田和宏, 泓田正雄, 青江順一. ダブル配列による動的辞書の構成と評価. 情報処理学会, Vol. 71, No. 3B-6-1, pp. 263–264, 2009.

[9] Atsushi Maeda and Kouta Mizushima. A compressed-array representation of automata and its application to programming language. *Proceedings of the 49th IPSJ Programming Symposium*, pp. 49–54, 2008.

[10] Shunsuke Kanda, Yuma Fujita, Kazuhiro Morita, and Masao Fuketa. Practical rearrangement methods for dynamic double-array dictionaries. *Software: Practice and Experience*, Vol. 48, No. 1, pp. 65–83, jan 2018.

[11] Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, Vol. 15, No. 4, pp. 514–534, oct 1968.

[12] Yasumasa Nakamura, Yu Nomura, and Hisatoshi Mochizuki. Implementation of Upda-

tion Techniques for the Trie Strucuture Based on the Set of Terminal Node. *IPSJ SIG Technical Report DD*, Vol. 33, No. 1, pp. 1–6, 2006.

[13] John A. Dundas. Implementing dynamic minimal ‐ prefix tries. *Software: Practice and Experience*, Vol. 21, No. 10, pp. 1027–1040, 1991.

[14] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., USA, 1990.

[15] Makoto Yasuhara, Toru Tanaka, Jun Ya Norimatsu, and Mikio Yamamoto. An efficient language model using double-array structures. In *EMNLP 2013 - 2013 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pp. 222–232, Seattle, Washington, USA, 2013.

[16] Jun-Ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. A Fast and Compact Language Model Implementation Using Double-Array Structures. *ACM Transactions on Asian and Low-Resource Language Information Processing*, Vol. 15, No. 4, pp. 1–27, jun 2016.

[17] Naoki Yoshinaga and Masaru Kitsuregawa. A Self-adaptive Classifier for Efficient Text-stream Processing. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, No. Aug, pp. 1091–1102, Dublin, Ireland, 2014. Dublin City University and Association for Computational Linguistics.

[18] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, Vol. 3, No. 2-3, pp. 158–182, oct 2005.

[19] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 2635–2639, dec 2013.

[20] Nikolay Bogoychev and Adam Lopez. N-gram language models for massively parallel devices. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1944–1953, Stroudsburg, PA, USA, 2016. Association for Computational Linguistics.

[21] 芳賀駿平, 谷口正訓, 山本幹雄. 部分転置ダブルアレイを用いた ngram 言語モデルの検討. In *The 30th Annual Conference of the Japanese Society for Artifical Intelligence*, 2016.

[22] 石井瑛彦, 芳賀駿平, 竹中孝介, 大隅賢二, 山本幹雄. 細粒度並列処理によるダブル配列言語モデルの構築高速化. 言語処理学会 第 24 回年次大会 発表論文集, pp. 216–219, 2018.

[23] 石井瑛彦, 山本幹雄. 双方向配置によるコンパクトかつ高速なダブル配列言語モデル構築. 情報処理学会第 81 回全国大会, Vol. 81, No. 4U-06, pp. 463–464, 2019.

[24] 松本拓真, 森田和宏, 泓田正雄. 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. 情報処理学会論文誌 データベース, Vol. 13, No. 2, pp. 34–44, 2020.

[25] 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.

[26] D. Revuz and D. Perrin. *Dictionnaires et lexiques. Méthodes et algorithmes.* Université de Paris VII, 1991.