ダブル配列構造による自然言語辞書の記憶量圧縮法

矢田 晋 † 大野 将樹 † 森田 和宏 † 泓田 正雄 † 青江 順一 †

† 徳島大学 工学部 知能情報工学科

概要 トライ法は自然言語辞書を中心として広く用いられているキー検索法であり，トライを実現するデータ構造に高速性とコンパクト性を併せもつダブル配列がある．ダブル配列の欠点は，キーの削除によって未使用要素が発生し，空間効率が低下する点である．これに対し大野らは，兄弟のいない要素（シングル要素）を利用して，ダブル配列を詰め直すことにより未使用要素を除去する圧縮法を提案した．しかし，この手法はシングル要素が少ないとすべての未使用要素を除去できない．本稿では，最終要素の兄弟が多いほど前方への移動が難しくなることに着目し，シングル要素が少ない状況でも高い空間効率を維持できる圧縮法を提案する．実験により，提案法は極めて高い空間効率を維持することが実証された．

1. はじめに

トライ法 [1] は，キーを構成する文字単位での比較を用いて検索する手法であり，キーの総数に依存しない高速な検索が可能である．また，検索失敗位置の特定や共通接頭辞の検出が容易であるという理由から，自然言語辞書を中心として幅広い分野で利用されている．

青江の提案したダブル配列 [2] は，配列構造の高速性とリスト構造のコンパクト性を兼ね備えたデータ構造であり，二つの配列を用いてトライを表現する．ダブル配列の欠点は，削除キー数に比例して未使用要素が増加し，空間効率が低下することである．この欠点に対し，大野らは，シングル要素を利用してダブル配列を詰め直すことにより，未使用要素を除去する圧縮法を提案した [3, 4]．しかし，シングル要素の割合が少ない状況では，十分に未使用要素を除去できない．

本稿では，最終要素の兄弟が多いほど詰め直しが難しくなることに着目し，シングル要素が少ない状況でも高い空間効率を維持できる圧縮法を提案する．EDR 電子化辞書の日英単語各 10 万件に対する実験により，従来法が空間効率を維持できない状況においても，提案法は極めて高い空間効率を維持することが実証された．

2. トライ法

トライ法は，キーを構成する文字を分岐条件とする木構造によりキー集合を表現する．キー集合 $K = \{$“*babe*”, “*bad*”, “*badge*”, “*be*”$\}$ に対するトライを**図 1** に示す．図中の ‘#’ はキーの終端を表す文字であり，あるキーが別のキーの接頭辞になっている場合の判別に必要となる．

トライ法における検索は，*root* である節 0 から，検索キーを構成する文字による遷移を繰り返すことにより実現される．そのため，節の遷移を $O(1)$ で確認できれば，キーの長さのみに依存する高速な検索が可能となる．

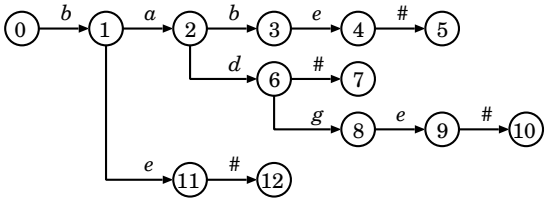


図 1 キー集合 K に対するトライ

トライ法には，検索の高速性以外にも二つの特徴がある．一つ目は，検索キーが登録されていなかった場合，検索失敗位置を容易に特定できることである．二つ目は，共通接頭辞が統合されるため，前方一致検索などを容易に実装できることである．これらの特徴から，トライ法はスペルチェック，構文解析，形態素解析などの幅広い分野で利用されている．

トライを実現する一般的なデータ構造としては，配列構造とリスト構造がある．検索速度に関しては，トライの総ノード数に依存するリスト構造に対し，キーの長さに依存する配列構造の方が優れている．一方，空間効率の面から比較すると，定義されていない遷移に空間を割り当てる配列構造に対し，定義された遷移のみに空間を割り当てるリスト構造の方が優れている．

3. ダブル配列

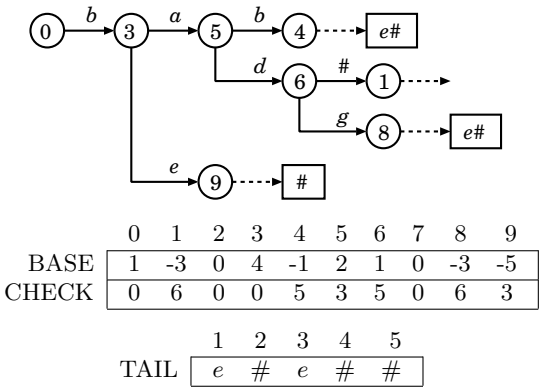
3.1 ダブル配列の概要

ダブル配列はトライの節の遷移を二つの配列 BASE, CHECK によって表現する．

BASE オフセット

CHECK 遷移元の確認

トライにおいて，節 s から節 t へ文字 c による遷移が



	0	1	2	3	4	5	6	7	8	9
BASE	1	-3	0	4	-1	2	1	0	-3	-5
CHECK	0	6	0	0	5	3	5	0	6	3

	1	2	3	4	5
TAIL	e	#	e	#	#

図 2 キー集合 K に対するトライとダブル配列

定義されている場合，ダブル配列は次式を満足する．

$$t = \mathrm{BASE}[s] + c, \quad \mathrm{CHECK}[t] = s \qquad (1)$$

すなわち，遷移先 t を $\mathrm{BASE}[s]$ と文字 c の内部コードの和で計算し，$\mathrm{CHECK}[t]$ を遷移元 s と比較することで遷移を確認できる．式 (1) の計算量は $O(1)$ であり，配列構造と同等の高速な検索が可能となる．

一般的にダブル配列では，トライの節を少なく抑えるため，キーの判別に必要な部分のみを節にする．キー集合からキーを判別できる最前方の節を分岐節といい，分岐節以降の遷移文字は連結して TAIL 配列に格納する．分岐節の BASE 値を TAIL 配列におけるオフセットとして用い，負の値にすることで他の節と区別する．

BASE 値が 0 の要素は未使用であり，トライにおいて対応する節が存在せず，検索には不要である．そのため，ダブル配列の空間効率は未使用要素数に依存する．未使用要素が少なければ，ダブル配列の空間使用量はリスト構造と同等になる．

キー集合 K に対するダブル配列を**図 2** に示す．遷移文字の内部コードは，終端文字 '#' を 0，文字 'a'〜'z' を 1〜26 とする．

[例 1] 図 2 のダブル配列からキー "*badge*" を検索する例を示す．まず，*root* である節 0 からの文字 'b' による遷移を確認するため，BASE[0]+'b' によって遷移先 3 を求める．CHECK[3] が遷移元 0 と等しいので，遷移の確認に成功し，節 0 から節 3 へと遷移する．同様にして，文字 'a', 'd', 'g' によって遷移すると，節 8 に到達する．ここで，BASE[8]<0 から，分岐節に到達したことがわかる．TAIL 配列のオフセット $-$BASE[8] 以降に格納されている文字列がキーの残りの文字列 "e" と等しいので，検索に成功する． （例終）

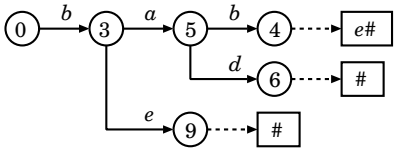
3.2 ダブル配列における削除

ダブル配列における削除は，検索によって削除キーの存在を確認し，到達した分岐節を除去することで実現される．ただし，除去する節の兄弟が分岐節一つの場合，この分岐節は対応するキーの判別に不要となるため，不要な節を除去し，TAIL 配列を更新しなければならない．ダブル配列における削除の手順を以下に示す．

step.1 削除キーを検索し，なければ終了する．
step.2 検索において到達した分岐節を除去し，その親を変数 s に設定する．
step.3 節 s の遷移先が分岐節一つであれば，その分岐節を変数 t に設定し，そうでなければ終了する．
step.4 節 s がシングル節でなければ step.6 に進む．このとき，節 s が新たな分岐節となる．
step.5 s =CHECK[s] として，step.4 に戻る．
step.6 節 s から節 t までの遷移文字と，TAIL 配列のオフセット $-$BASE[t] 以降に格納されている文字列を連結し，TAIL 配列の後方に格納する．
step.7 節 s 以降の節を除去し，新たに TAIL 配列に格納した文字列へのオフセットを BASE[s] に設定する．

ダブル配列において，不要な節の除去は，対応する要素を未使用にすることによって実現する．そのため，削除キー数に比例して未使用要素が増加する．一方，ダブル配列の要素数は，最後方にある使用要素（以降，最終要素と呼ぶ）によって決定されるので，最終要素に対応する節が除去されない限り減少することはない．削除キー数に比例して増加する未使用要素に対し，要素数はほとんど減少しないため，削除するほど空間効率は低下することになる．

[例 2] 図 2 のダブル配列において，キー "*badge*" を削除する例を示す．まず，検索によってキー "*badge*" が登録されていることを確認し，検索において到達した分岐節 8 を除去する．除去した分岐節 8 の親である節 6 の遷移先が分岐節 1 だけであるので，節 1 がキーの判別に不要となることがわかる．また，節 6 はシングル節でないため，節 6 が新たな分岐節となることがわかる．節 6 から節 1 までの遷移文字が '#' だけであるので，TAIL[6] に '#' を格納する．節 1 を除去し，BASE[6]=-6 とすれば削除終了となる．**図 3** に示す削除後のダブル配列から，要素数は変化することなく，未使用要素が二つ増加していることがわかる． （例終）



	0	**1**	2	3	4	5	6	7	**8**	9
BASE	1	**0**	0	4	-1	2	-6	0	**0**	-5
CHECK	0	**0**	0	0	5	3	5	0	**0**	3

	1	2	3	4	5	6
TAIL	*e*	#	*e*	#	#	#

図 3 キー “*badge*” を削除したダブル配列

3.3 ダブル配列の圧縮法とその問題点

先に述べたように，ダブル配列の要素数は最終要素によって決定される．そのため，最終要素を前方の未使用要素に移動すれば，ダブル配列の要素数を削減することができる．本稿では，最終要素とその兄弟要素をまとめて圧縮要素と定義する．

大野らは，削除において不要な節を除去した後，圧縮要素を前方の未使用要素とシングル要素に移動することにより，ダブル配列中の未使用要素を除去する手法を提案した．移動先として採用したシングル要素については，最終要素の後方に退避しておき，圧縮要素を移動してから未使用要素に移動する．圧縮の手順を以下に示す．

step.1 未使用要素がなければ終了する．
step.2 圧縮要素を取得する．
step.3 圧縮要素の移動先を探索し，なければ終了する．
step.4 移動先に未使用でない要素があれば，最終要素の後方に移動する．
step.5 圧縮要素を前方に移動する．
step.6 step.1 に戻る．

ダブル配列の圧縮は，未使用要素がなくなるか，圧縮要素を移動できなくなるまで継続する．そのため，圧縮要素の移動先候補が不足すると，途中で圧縮要素を移動できなくなり，すべての未使用要素を除去する前に圧縮は終了する．

大野らの圧縮法の問題点は，シングル要素が少ないと移動先候補が不足し，すべての未使用要素を除去できないことである．また，圧縮要素の移動先がない状況では，ダブル配列全体を走査することになり，圧縮速度が著しく低下する．

[例 3] 図 2 のダブル配列を圧縮する例を**図 4** に示す．まず，圧縮要素 {5,9} を取得する (a)．移動先を探索すると {3,7} が見つかるが，要素 3 は未使用でないため，

	0	1	2	3	4	**5**	6	7	8	**9**
BASE	1	-3	0	4	-1	**2**	1	0	-3	**-5**
CHECK	0	6	0	0	5	**3**	5	0	6	**3**

(a) 初期状態

	0	1	2	3	4	5	6	7	8	9	**10**
BASE	8	-3	0	2	-1	0	1	-7	-3	0	**2**
CHECK	0	6	0	10	5	0	5	10	6	0	**0**

(b) 圧縮要素 {5,9} 移動後

	0	**1**	2	3	4	5	6	7	**8**
BASE	3	**-3**	0	2	-1	2	1	-7	**-3**
CHECK	0	**6**	0	5	5	0	5	5	**6**

(c) 圧縮要素 {10} 移動後（圧縮終了）

図 4 図 2 のダブル配列を圧縮する例

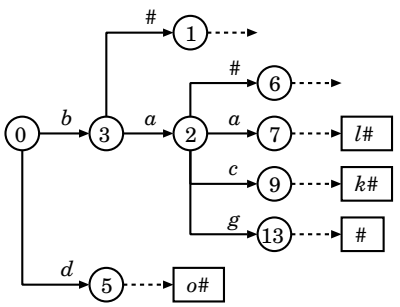
最終要素のさらに後方となる要素 10 に移動する．{3,7} が両方とも未使用になるので，{5,9} を {3,7} に移動する (b)．後方に移動した要素 10 を前方の未使用要素 5 に移動した時点で，圧縮要素 {1,8} を前方に移動できなくなり，圧縮は終了する (c)．圧縮によって，未使用要素が一つ除去され，要素数が 10 から 9 に減少したことがわかる．（例終）

4. 圧縮要素の数を考慮した効率的圧縮法

ダブル配列の圧縮において，圧縮要素が多いほど移動先候補の配置条件は厳しくなる．そのため，圧縮要素が多い状況で圧縮を継続するには，移動先候補が大量に必要となる．従来法がすべての未使用要素を除去できないのは，移動先候補を未使用要素とシングル要素に限定するため，圧縮要素が多い状況で移動先候補が不足するからである．

提案法は，この問題を解決するために移動先候補を拡張する．事前調査により，兄弟数が圧縮要素の半分以下の要素を移動先候補とするのが妥当であると判断した．従来法では圧縮要素の数に関わらず移動先候補が決定されるのに対し，提案法では圧縮要素が多いほど移動先候補も多くなる．これにより，提案法は，シングル要素が少なく圧縮要素が多い状況でも圧縮を継続できるので，高い空間効率を維持することができる．なお，提案法の圧縮手順は従来法と同じである．

[例 4] キー集合 K'={“*b*”, “*ba*”, “*baal*”, “*back*”, “*bag*”, “*do*”} に対するダブル配列を**図 5** に示す．このダブル配列の圧縮要素は {6,7,9,13} であり，従来法では移動先が存在しない．一方，提案法では兄弟数が 2 の要素 {1,2,3,5} も移動先候補となるので，圧縮要素



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

	1	2	3	4	5	6	7
TAIL	*l*	#	*k*	#	#	*o*	#

図 5 キー集合 K' に対するダブル配列

を {1,2,4,8} に移動することができる．移動先の未使用でない要素 {1,2} を後方に退避し，圧縮要素を移動してから {6,7} に戻すと未使用要素がなくなり，圧縮が終了する．（例終）

5. 評価

提案法は約 2,000 行の C 言語で実装されており，CPU:Pentium4 2.8GHz，OS:WindowsXP 上で稼動している．実験に用いたキー集合は，EDR 日英単語辞書 [5] から作成したものであり，キー数は日英各 10 万件である．実験では，各キー集合に対するダブル配列からキーを削除した後，従来法と提案法を用いて圧縮をおこなった．実験結果として，圧縮後の未使用要素数と空間使用量，単位時間に除去できた未使用要素数を**表 1** に示す．

表 1 より，従来法は，削除によって発生した未使用要素を十分に除去できず，空間効率が低下していることがわかる．また，シングル要素が少なくなる EDR 日辞書をキー集合とした場合，EDR 英辞書を用いた場合より多くの未使用要素が残留している．このことから，従来法による圧縮後の空間効率は，シングル要素の割合に依存することがわかる．

一方，提案法では，キー集合や削除キー数によらず，すべての未使用要素を除去できている．従来法が空間効率を維持できない状況でも，提案法は高い空間効率を維持することがわかる．また，単位時間に除去できる未使用要素数の比較により，提案法の圧縮速度は従来法と同等かそれ以上であることがわかる．

以上のことから，提案法はダブル配列の圧縮において有効であるといえる．

表 1 実験結果

削除キー数	10,000	20,000	30,000	40,000	50,000
EDR 英辞書					
シングル要素数	34,840	31,107	27,371	23,088	18,807
使用要素数	173,267	154,065	134,755	114,840	95,009
未使用要素数					
従来法	11,765	28,568	43,903	30,104	16,983
提案法	0	0	0	0	0
空間使用量 [KB]					
従来法	2,356	2,285	2,196	1,809	1,428
提案法	2,238	1,999	1,756	1,508	1,258
圧縮速度 [1/ms]					
従来法	120.6	321.0	412.4	606.5	966.8
提案法	345.6	429.2	558.4	650.5	817.0
EDR 日辞書					
シングル要素数	17,980	15,656	13,371	11,188	8,887
使用要素数	143,579	127,095	110,768	93,442	78,096
未使用要素数					
従来法	16,425	32,547	48,762	58,830	45,080
提案法	0	0	0	0	0
空間使用量 [KB]					
従来法	2,183	2,120	2,061	1,938	1,575
提案法	2,018	1,795	1,573	1,349	1,124
圧縮速度 [1/ms]					
従来法	7.3	28.4	63.8	310.6	710.0
提案法	139.0	197.8	277.2	381.8	485.2

6. おわりに

本稿では，圧縮要素によって移動先候補の配置条件が変化することに着目し，シングル要素が少なくても高い空間効率を維持できる圧縮法を提案した．また，実験によって提案法の有効性を実証した．提案法により，ダブル配列の応用分野がさらに広がるものと思われる．

今後の課題は，提案法を実用システムに適用し，有効性を確認することである．

参考文献

[1] Fredkin E. Trie memory. *Commun.ACM.*, Vol. 3, No. 9, pp. 490–500, 1960.

[2] 青江順一. ダブル配列による高速ディジタル検索アルゴリズム. 電子情報通信学会論文誌, Vol. J71–D, No. 9, pp. 1592–1600, 1988.

[3] 森田和宏, 泓田正雄, 大野将樹, 青江順一. ダブル配列における動的更新の効率化アルゴリズム. 情報処理学会論文誌, Vol. 42, No. 9, pp. 2229–2238, 2001.

[4] 大野将樹, 森田和宏, 泓田正雄, 青江順一. ダブル配列におけるキー削除の効率化手法. 情報処理学会論文誌, Vol. 44, No. 5, pp. 1311–1320, 2003.

[5] 日本電子化辞書研究所. EDR 電子化辞書, 1996.