6N-08

階層構造を用いたダブル配列の圧縮法

神田 峻介　　泓田 正雄　　森田 和宏　　青江 順一

徳島大学大学院 先端技術科学教育部

1 はじめに

キー集合を扱う代表的なデータ構造としてトライ [1] があり，自然言語処理や情報検索などで幅広く用いられている．トライとは各枝に文字を付随した木構造であり，コンパクト性に優れ，高速な完全一致検索や前方一致検索などを提供する．トライを効率的に実現する手法として，ダブル配列法 [2, 3] が存在する．ダブル配列は，高速性に秀でたデータ構造であり，BASE と CHECK の 2 つの配列を用いてトライのノード間の遷移を $O(1)$ で実現する．しかし，BASE は 4 bytes の記憶領域を要し，BASE の記憶領域がダブル配列における記憶効率の低下を招く．

そこで，本稿では，BASE 配列を階層構造により実現することで，BASE の記憶領域を削減し，ダブル配列の記憶量を圧縮する手法を提案する．

2 ダブル配列

ダブル配列は，2 つの 1 次元配列 BASE，CHECK を用いてトライの遷移を実現する．基本的に，配列の要素とトライのノードは一意に対応し，配列のインデックスはノード番号を表す．ダブル配列では，ノード s から遷移文字 c によるノード t への遷移を次式により定義する．

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

但し，複数のノードから同じノードへの遷移が成立する状況を回避するため，葉ノードを除く BASE 値の重複は禁止される．s, t はノード番号を表し，整数値として扱われる．code(c) は遷移文字 c の内部表現値を表す．一般的にダブル配列では文字列をバイト列として扱うため，code(c) には 0 ~ 255 の値が用いられる，ダブル配列は，ノード間の遷移を $O(1)$ で実現する高速なデータ構造である．しかし，BASE には子ノード番号に対応した整数値を格納する必要があるため，BASE 値は 4 bytes を要する．CHECK 値はバイト文字を表現するため，1 byte を要する．ダブル配列の要素数を n としたとき，ダブル配列の記憶量は $5 \cdot n$ bytes となる．

キー集合 K = {“*at*”,“*ate*”,“*eat*”,“*ee*”,“*tea*”,“*tee*”} に対するトライを図 1 に示し，ダブル配列を図 2 に示す．ここで，‘#’ は文字列の終端を表しており，遷移文字の内部表現値は (‘#’,‘*a*’,‘*e*’,‘*t*’) = (0, 1, 2, 3) としている．また，図 2 における要素数 n は 19 である．

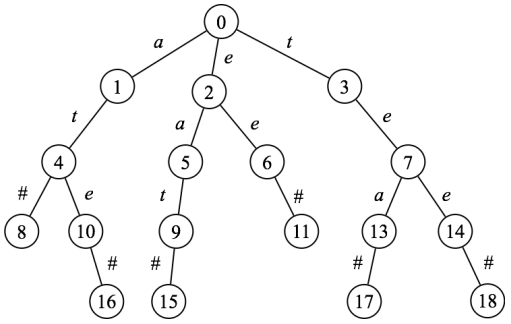

図 1: トライ

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
BASE	0	1	4	5	8	6	11	12		15	16			17	18				
CHECK		*a*	*e*	*t*	*t*	*a*	*e*	*e*	#	*t*	*e*	#		*a*	*e*	#	#	#	#

図 2: ダブル配列

3 提案手法

3.1 概要

本手法では，1 ノード当たりに必要な BASE 値の記憶量を圧縮することを目的とし，BASE 配列を 2 層の階層構造により実現する．本稿では，BASE 配列の第 1 階層を LBASE，第 2 階層を SBASE と表記する．LBASE は BASE 値をブロック単位で管理する配列であり，1 ブロック当たり複数個の要素が割り当てられる．本稿では，1 ブロックに割り当てる要素の数を *bsize* と表し，ノード s に対応する LBASE のインデックスを求める関数 block(s) = $\lfloor s/bsize \rfloor$ を定義する，SBASE は従来の BASE 配列と同様，要素とノードが一意に対応する配列である．本手法では，式 (1) における BASE[s] を次式により実現する．

$$\mathrm{BASE}[s] = \mathrm{LBASE}[\mathrm{block}(s)] + \mathrm{SBASE}[s] \tag{2}$$

BASE[s] を階層構造により実現する場合，LBASE[block(s)] にはブロック block(s) に属する BASE 値の中で最小の値が格納され，SBASE[s] には BASE[s] − LBASE[block(s)] が格納される．このとき，SBASE[s] ≥ 0 である．本手法によるキー集合 K に対するダブル配列を図 3 に示す．

本手法では，LBASE 値に 4 bytes を確保し，SBASE 値をよりコンパクトに実現することで，1 ノード当たりに必要な BASE 値の記憶量の圧縮を図る．LBASE はブロック単位で要素を管理する配列であるため，1 ノード当たりの LBASE の記憶量は $4/bsize$ bytes となる．ここで，SBASE 値に割り当てる記憶量を a bytes とすると，本手法における 1 ノード当たりの BASE 値の記憶量は $4/bsize + a$ bytes となる．そのため，この *bsize* と a の値により BASE 値の圧縮率は変動する．

3.2 圧縮率に関する考察

ダブル配列では，式 (1) に準拠しノードを配置することでトライの遷移条件を満足する．しかし，本手法では，LBASE 値をブロック単位で共有し，且つ SBASE 値に割り当てる記憶量を制限するため，*bsize* と a の値によっては式 (1) を満た

	0	1	2	3	4
LBASE	0	6	15	17	

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
SBASE	0	1	4	5	2	0	5	6		0	1			0	1				
CHECK		*a*	*e*	*t*	*t*	*a*	*e*	*e*	#	*t*	*e*	#		*a*	*e*	#	#	#	#

図 3: BASE 配列の階層化によるダブル配列

A New Compression Method by Hierarchical Double-array Structures
Shunsuke KANDA, Masao FUKETA, Kazuhiro MORITA and Jun-ichi AOE
Department of Information Science and Intelligent Systems, University of Tokushima

し得ない場合が存在する．そのため，どのようなキー集合に対しても式 (1) を満たし得る，*bsize* と *a* の値について考察する必要がある．本節では，遷移対象となる子ノードの数が最高であり，且つ，自由にノードの配置がおこなえるケースを想定することにより，*bsize* と *a* の値，及び圧縮率について考察する．

ダブル配列では，文字列をバイト列として扱うため，各ノードが保持する子ノードの数は高々256(2^8) である．1 ブロックに属するノードの数は高々*bsize* であるため，各ブロックの遷移対象となる子ノードの合計は高々256·*bsize* となる．ここで，あるブロックの遷移対象となる子ノードの合計が $256 \cdot bsize$ であり，且つ，これら子ノードが連続して配置可能なケースを想定する．このとき，これら子ノード番号の最小値を x とすると，最大値は $x + 256 \cdot bsize - 1$ となる．また，遷移文字の内部表現値は $0 \le \mathrm{code}(c) \le 255$ のため，このブロックにおける BASE 値の表現領域は，$x \sim x + 256 \cdot bsize - 1 - 255$ となる．ここで，LBASE にはブロックにおける BASE 値の最小値である x が格納される．そのため，SBASE の表現領域は $0 \sim 256 \cdot bsize - 256$ となり，$256 \cdot bsize - 256 < 2^{8a}$ を満たすとき，本手法における遷移条件は必ず満足する．本手法では，1 ノード当たりに必要な BASE 値の記憶量を圧縮することを目的としているため，$a = 1, 2, 3$ のケースが想定される．これらのケースに対する *bsize* の値と，そのときの圧縮率についての考察を以下に示す．

- $a = 1$ の場合
  $256 \cdot bsize - 256 < 2^8$，すなわち $bsize = 1 < 2$ であれば，本手法における遷移条件は満足する．このとき，1 ノード当たりの BASE 値の記憶量は，$4/bsize + a = 4/1 + 1 = 5$ bytes となる．すなわち，BASE 値の記憶量は 125%に拡張される．
- $a = 2$ の場合
  $256 \cdot bsize - 256 < 2^{16}$，すなわち $bsize < 257$ であれば，本手法における遷移条件は満足する．ここで，$bsize = 256$ のときに BASE 値の圧縮率は最も高くなり，1 ノード当たりの BASE 値の記憶量は $4/256 + 2 \simeq 2$ bytes となる．すなわち，BASE 値の記憶量は約 50%に圧縮される．
- $a = 3$ の場合
  $256 \cdot bsize - 256 < 2^{24}$，すなわち $bsize < 65537$ であれば，本手法における遷移条件は満足する．ここで，$bsize = 65536$ のときに BASE 値の圧縮率は最も高くなり，1 ノード当たりの BASE 値の記憶量は $4/65536 + 3 \simeq 3$ bytes となる．すなわち，BASE 値の記憶量は約 75%に圧縮される．

上記の考察により，$a = 2$ の場合に BASE 値の圧縮率は最も高くなる．

4 評価

本手法の有効性を実証するために *bsize* に対するに評価実験，及び従来手法との比較による評価実験をおこなった．*bsize* に対するに評価実験では，BASE 値の圧縮率が最も高い $a = 2$ において，最も記憶効率が高くなる $bsize < 257$ の値を模索した．比較実験では，比較手法を従来のダブル配列と簡潔データ構造である LOUDS[4] とし，本手法とダブル配列を C++ により実装した．LOUDS には Tx-library[1] を用いた．実験に用いた PC のプロセッサは Quad-Core Intel Xeon 2 x 2.4 GHz であり，L2 cache のサイズは 256 KB である．キー集合として，英語版 Wikipedia タイトル[2]（UTF-8）から無作為に選んだ各 50 万語〜200 万語を使用した．

bsize に対するに評価実験の結果を表 1 に示す．表 1 より，キー総数が 50 万語の場合は $bsize = 224$ において最も記憶効率は高く，100 万語〜200 万語の場合は $bsize = 256$ において最も記憶効率が高くなった．

表 1: $a = 2$ における *bsize* と記憶量に対する実験結果

	記憶量 [MB]			
bsize	50 万	100 万	150 万	200 万
32	17.7489	33.5517	48.7363	63.3791
64	17.4162	32.8935	47.7834	62.1293
96	17.3190	32.6974	47.4634	61.7157
128	17.2753	32.6021	47.3336	61.5337
160	17.2517	32.5480	47.2465	61.4151
192	17.2364	32.5144	47.1962	61.3441
224	17.2352	32.4994	47.1743	61.2916
256	17.2385	32.4948	47.1496	61.2588

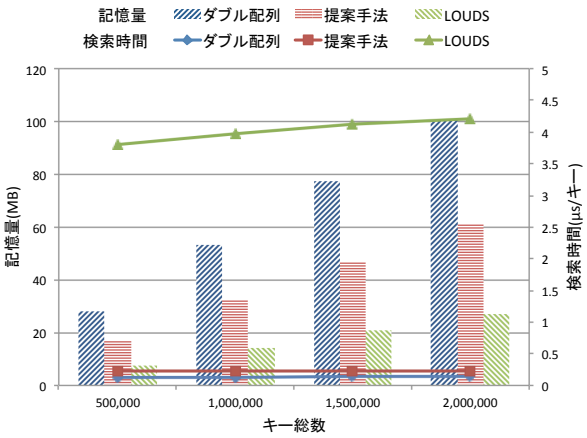

図 4: 比較実験による記憶量と検索速度に対する実験結果

$a = 2$，$bsize = 256$ における比較実験の結果を図 4 に示す．図 4 より，本手法はダブル配列を約 61%に圧縮することが示された．また，検索速度についてはダブル配列とほぼ同等であり，LOUDS よりも約 19 倍高速であることが示された．

5 おわりに

本稿では，BASE 配列を階層構造により実現する圧縮法を提案し，その有効性を示した．今後の予定としては，本手法の簡潔な構築アルゴリズムを提案することが挙げられる．

参考文献

[1] 青江順一. トライとその応用 (キー検索技法 4). 情報処理, Vol. 34, No. 2, pp. 244–251, 1993.

[2] Jun-ichi 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.

[3] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun-ichi Aoe. A compact static double-array keeping character codes. *Information Processing & Management*, Vol. 43, No. 1, pp. 237 – 247, 2007.

[4] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the louds succinct tree representation. In *Proceedings of WEA 2006*, pp. 134–145, 2006.

[1] https://code.google.com/p/tx-trie/
[2] http://dumps.wikimedia.org/enwiki/