D-029

# ダブル配列における一方向分岐の遷移あたりに要する計算量の抑制

## Proposal of Efficient Transition Method Focused on the One-Way Branch on a Double-Array Structure

芳本　貴男[†]　　重越　秀美[†]　　蔵満　琢麻[†]　　望月　久稔[†]

Takao YOSHIMOTO　Hidemi SHIGEKOSHI　Takuma KURAMITSU　Hisatoshi MOCHIZUKI

### 1．はじめに

自然言語処理システムの辞書を中心に広く用いられるトライのデータ構造としてダブル配列 [1] がある．ダブル配列は節点間の遷移を $O(1)$ で決定できる．また，遷移あたりに要する時間計算量および記憶領域をより抑制するため，キーを決定づける節点以降の一方向分岐を遷移列として 1 次元配列に格納する．

しかし，一方向分岐はキーを決定づける節点以前にも存在する．ダブル配列の探索処理を高速化する研究として，キャッシュミスの発生を抑制する手法 [2] が提案されているが，キーを決定づける節点以前の一方向分岐については触れられていない．

本論文では，ダブル配列上に連続して存在する一方向分岐を 1 次元配列に格納することで遷移時間計算量および記憶領域を抑制する手法を提案する．また，実験によりその有効性を評価する．

### 2．ダブル配列の探索処理

ダブル配列は，2 つの配列 BASE，CHECK によりトライを表現し，トライにおける節点 $s$ から遷移種 $a$ による節点 $t$ への遷移を式 (1) で定義する．ここで，BASE 値には遷移先節点の位置を求めるための基底位置を，CHECK 値には遷移種を格納する [2]．以下，ダブル配列上の要素 $x$ に関する BASE 値，CHECK 値をそれぞれ $B[x]$，$C[x]$ で表す．

$$\begin{cases} t = B[s] + a \\ C[t] = a \end{cases} \quad (1)$$

トライ上の各キーを一意に決定する最前方の節点をセパレート節点 (以下，SP 節点)[1] と呼ぶ．SP 節点以降に存在するすべての節点は，遷移先が一意な一方向分岐となるため，これらを遷移列として配列 TAIL に格納する．これにより，配列 TAIL 上に連続する遷移種との照合のみで遷移できるため，探索処理を高速化できる．配列 TAIL の格納位置を SP 節点の BASE 値に負値で表し，SP 節点と他の節点とは BASE 値の符号で区別する．以下，配列 TAIL 上の $x$ 番目の要素を $T[x]$ で表す．

ダブル配列によるキーの探索アルゴリズムを以下に示す．

**関数 Search($key[1\cdots n]$)**

**(手順 1) 状態の初期化**

状態 $s$ に根を，キーの走査位置 $k$ に 1 を，遷移先 $t$ に $B[s] + key[k]$ を設定する．

**(手順 2) 配列 BASE，CHECK による遷移**

$C[t]$ と $key[k]$ が一致しなければ遷移が成立せず探索失敗として終了する．一致すれば遷移が成立し，$s$ に $t$ を，$k$ に $k+1$ を設定する．$B[s]$ が正値であれば $t$ に $B[s] + key[k]$ を設定し，手順 2 を繰り返す．$B[s]$ が負値であれば手順 3 へ進む．

**(手順 3) 配列 TAIL による遷移**

$-B[s]$ を $p$ とし，遷移列 $T[p\cdots p+(n-k)]$ と $key[k\cdots n]$ とを照合する．一致すれば探索成功として，一致しなければ探索失敗として終了する．[関数終]

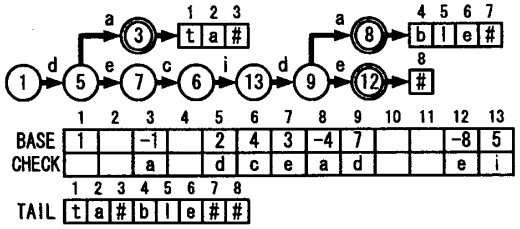


図 1　キー集合 $K$ に対するトライとダブル配列

**例 1:** キー集合 $K = \{$"data#", "decidable#", "decide#"$\}$ に対するトライとダブル配列を図 1 に示す．遷移種 '#' は終端記号を表し，'#', 'a', 'b', $\cdots$, 'z' の内部表現値をそれぞれ 0,1,2,$\cdots$,26 とする．キー "decidable#" を探索する．まず，$s$ に根である 1 を，$k$ に 1 を，式 (1) より遷移先 $t$ に 5(= $B[1]$+'d') を設定する．$C[5]$ ='d' より，節点 5 への遷移が成立し，$s$ に 5 を，$k$ に 2 を設定する．続いて，$B[5]$ は正値であるため，遷移先 $t$ に 7(= $B[5]$+'e') を設定する．$C[7]$ ='e' より，節点 7 への遷移が成立する．同様に，節点 6，節点 13，節点 9，節点 8 へ遷移し，$s$ は 8，$k$ は 7 となる．ここで，$B[8]$ が負値であるため，配列 TAIL での照合を開始する．配列 TAIL の格納位置は 4(= $-B[8]$) であるため，$T[4\cdots 7]$ と $key[7\cdots 10]$ とを照合し，"ble#" で一致するため，探索成功として終了する．(例終)

### 3．一方向分岐の遷移方法を改善したダブル配列の探索処理

本論文では，SP 節点の前後に関わらず連続して存在する一方向分岐を遷移列として 1 本の 1 次元配列に格納する手法を提案し，遷移あたりに要する時間計算量の抑制による探索処理の高速化および記憶領域の抑制を図る．以下，この配列を配列 ST と呼び，$x$ 番目の要素を $ST[x]$ で表す．

一方向分岐をもつ連続する節点を $s_1, s_2, \cdots, s_n$ とし，それぞれがもつ遷移種を $a_1, a_2, \cdots, a_n$ とする．図 1 の場合，節点 7，節点 6，節点 13 が $s_1, s_2, s_3$，遷移種 'c','i','d' が $a_1, a_2, a_3$ にあたる．このとき，遷移列 $a_1, a_2, \cdots, a_n$ を配列 ST に格納し，その格納位置を $B[s_1]$ に負値で設定する．格納位置を $p$(= $-B[s_1]$) とすると，$ST[p\cdots p+n-1]$ に遷移列 $a_1, a_2, \cdots, a_n$ が格納される．

節点 $s_1$ が SP 節点である場合は，遷移種 $a_n$ が終端記号 '#' となり，従来のダブル配列における配列 TAIL と同様である．しかし，節点 $s_1$ が SP 節点でない場合，節点 $s_n$ から遷移種 $a_n$ による遷移先節点を $t$ とすると，節点 $t$ は多方向の分岐をもつ．図 1 の場合，節点 9 が節点 $t$ にあたる．配列 ST において遷移種 $a_n$ との照合が成功した時点で節点 $t$ からの遷移を実現するためには，節点 $t$ の BASE 値が必要となる．まず，多方向分岐の存在を示すため，いかなる遷移種とも重ならない記号 '$' を定義し，$ST[p+n]$ に格納する．次に，BASE 値を表現するバイト幅を $M$ とし，$ST[p+n+1]$ から $M$byte の領域を使用して配列 ST 上に節点 $t$ の BASE 値を格納する．これにより，'$' との照合で多方向分岐の存在を確認した後，配列 ST に格納された BASE 値を取得することで，配列 ST から配列 BASE，CHECK 上の節点への遷移を実現できる．

[†] 大阪教育大学, Osaka Kyoiku University

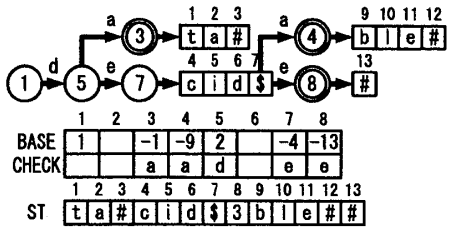


図 2　遷移方法を改善したダブル配列

ただし，遷移方法の変更には '$' を確認する処理が発生するため，連続する一方向分岐数が少ない場合も遷移方法を変更するのは効率的ではない．そこで，定数 $L$ を定義し，一方向分岐が $L$ 回以上連続しなければ遷移方法を変更しないように制限する．一方向分岐が $L$ 回以上連続せず遷移方法を変更しない場合は，従来のダブル配列と同様に式 (1) の定義にしたがって遷移する．

関数 Search の遷移方法を改善した探索アルゴリズムを以下に示す．手順 1，手順 2 は従来のダブル配列と同様である．

**(手順 3) 配列 ST による遷移**

$-B[s]$ を $p$ とし，$ST[p]$ と $key[k]$ とを照合する．終端記号 '#' で一致する場合は，探索成功として終了する．終端記号 '#' 以外で一致する場合は，$p$ に $p+1$，$k$ に $k+1$ を設定し，手順 3 を繰り返す．$ST[p]$ と $key[k]$ とが一致しない場合，$ST[p]$ が多方向分岐を表す '$' 以外であれば，探索失敗として終了する．$ST[p]$ が '$' であれば，$ST[p+1 \cdots p+M]$ を BASE 値として，$t$ に $ST[p+1 \cdots p+M] + key[k]$ を設定し，手順 2 へ進む．

**例 2:** キー集合 $K$ に対する遷移方法を改善したダブル配列を図 2 に示す．キー "decidable#" を探索する．BASE 値のバイト幅 $M$ を 1 とする．例 1 と同様に，式 (1) の定義にしたがい，根から節点 5，節点 7 へ遷移し，$s$ は 7，$k$ は 3 となる．ここで，$B[7]$ は負値であるため，遷移方法を配列 ST に変更する．配列 ST 上での照合位置は $4(= -B[7])$ であるため，$ST[4]$ ='c'，$ST[5]$ ='i'，$ST[6]$ ='d' と照合を進め，$k$ が 6 のとき $ST[7] \neq$'a' となり照合が失敗する．$ST[7]$ は多方向分岐を表す '$' であるため，$ST[8]$ から BASE 値 3 を得る．遷移先 $t$ に $4(= 3+$'a') を設定し，$C[4]$ ='a' より，節点 4 への遷移が成立する．さらに，$B[4]$ は負値であるため，再び配列 ST 上で照合を開始し，$ST[9]$ ='b'，$ST[10]$ ='l'，$ST[11]$ ='e'，$ST[12]$ ='#' と進め，終端記号 '#' で照合が成功するため，探索成功として終了する．(例終)

## 4．　実験による評価

提案手法の有効性を示すため，ダブル配列 [2](以下，対象手法) との比較実験を Intel Pentium Dual 1.6GHz，Fedora7 上で行った．実験では，キー集合として URI をランダムに 20 万語抽出したもの (平均キー長 57.5) を用いた．ここで，遷移種となる各構成文字の内部表現値を ASCII コードとし，配列 BASE，CHECK における 1 要素あたりのサイズをそれぞれ 3byte，1byte，配列 TAIL および配列 ST における 1 要素あたりのサイズを 1byte とした．また，配列 ST 上に格納する BASE 値のバイト幅 $M$ は配列 BASE のサイズに合わせ 3 とした．

20 万語のキー集合より 1 万語から 20 万語まで 1 万語毎にキー数を増やした部分キー集合を作成し，提案手法および対象手法について，各部分キー集合に対する成功探索を行った．1 キーあたりの探索時間を図 3 に示す．図 3 の提案手法は，定数 $L$ を本実

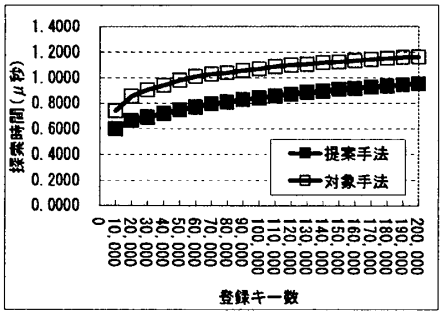


図 3　URI キー集合に対する探索時間

表 1　URI キー集合 20 万語に対する実験結果

| | 提案手法 | 対象手法 |
|---|---|---|
| 探索時間 [$\mu$ 秒] | 0.95(0.82) | 1.16(1.00) |
| 遷移回数 (BASE,CHECK) | 12.1(0.30) | 40.9(1.00) |
| 遷移回数 (ST/TAIL) | 46.3(2.65) | 17.5(1.00) |
| 節点数 (ST/TAIL 除く) | 332,029(0.35) | 955,066(1.00) |
| BASE,CHECK[MB] | 1.33(0.35) | 3.83(1.00) |
| ST/TAIL[MB] | 4.36(1.25) | 3.50(1.00) |
| 合計使用領域 [MB] | 5.69(0.78) | 7.33(1.00) |

験において最も遷移時間計算量が抑制された 3 に設定した探索時間を示す．また，登録キー数 20 万語における探索時間および遷移回数，配列 BASE，CHECK 上の節点数，使用領域を表 1 に示す．表中の括弧は対象手法を 1 としたときの対比を表す．

提案手法は，ダブル配列上に連続して存在する一方向分岐による遷移を配列 ST 上に作成するため，表 1 に示すように配列 BASE，CHECK 上の節点数は対象手法に対して 0.35 倍となった．提案手法における各配列の使用領域は，配列 BASE，CHECK が対象手法に対して 0.35 倍，配列 ST が対象手法の配列 TAIL に対して 1.25 倍となった．配列 ST における 1 要素あたりのサイズが配列 BASE，CHECK に対して 1/4 であるため，提案手法の合計使用領域は対象手法に対して 0.78 倍に抑制された．さらに，探索において，表 1 に示すように遷移時間計算量が小さい配列 ST 上での遷移回数の割合が対象手法に対して大きくなるため，図 3 に示すように探索時間が対象手法に対して短縮され，登録キー数 20 万語における探索時間は 0.82 倍となった．以上のことから，提案手法は対象手法に対して，時間的にも領域的にも効率的である結果が得られた．

## 5．　おわりに

本論文では，ダブル配列上に連続して存在する一方向分岐の遷移方法を改善した探索手法を提案し，実験により探索時間の短縮および使用領域の削減を示した．今後の課題として，構築処理を評価することが挙げられる．

### 参考文献

[1] Aoe J., An Efficient Digital Search Algorithm by Using a Double-Array Structure, IEEE Transactions on Software Engineering, Vol.15, No.9, pp.1066-1077, 1989.

[2] 矢田晋，森田和宏，泓田正雄，平石亘，青江順一，ダブル配列におけるキャッシュの効率化，FIT2006，LD-008 pp.71-72, 2006.