D-028

遷移先節点集合の管理によるダブル配列の更新手法

Insertion Method for a Double-Array Structure Based on the Set of Reachable Nodes

重越　秀美[†]　芳本　貴男[†]　中川　知之[†]　蔵満　琢麻[†]　望月　久稔[†]
Hidemi SHIGEKOSHI　Takao YOSHIMOTO　Tomoyuki NAKAGAWA　Takuma KURAMITSU　Hisatoshi MOCHIZUKI

1．　はじめに

自然言語処理などに用いられるトライのデータ構造として，コンパクト性と探索処理の高速性を併せ持つダブル配列 [1] がある．ダブル配列の更新処理を高速化するために，矢田らにより未使用要素を管理する手法 [2] や，中村らにより遷移先節点集合を管理する手法 [3] が提案されている．

本論文では，遷移先節点集合をより小さい領域で管理することで更新処理の高速化と領域の抑制をはかる．さらに，未使用要素の管理を２つに分けて，更新時に削除される節点を管理することで，更新処理をより高速化する手法を提案する．

2．　ダブル配列における更新処理の問題点

ダブル配列はトライの遷移を一次元配列 Base と Check を用いて実現し，トライにおける最終分岐以降の遷移を一次元配列 Tail に格納することで，トライ上の節点数を抑制する [1]．節点 s から遷移種 a による節点 t への遷移を式 (1) により定義する．以下，節点 s からの K 個の遷移集合 $\{a_1, a_2, \cdots, a_K\}$ を $\mathrm{L}(s)$，$\mathrm{L}(s)$ による遷移先節点集合 $\{t_1, t_2, \cdots, t_K\}$ を $\mathrm{T}(s)$ と表記する．

$$\begin{cases} t = \mathrm{Base}[s] + a \\ \mathrm{Check}[t] = s \end{cases} \tag{1}$$

式 (1) より，節点 s の遷移先の節点番号 t は，節点 s の Base 値に依存するので，キーの追加処理において，節点 s に新たな遷移先節点 t を作成するとき，ダブル配列の要素 t が既に他の節点 u として使用されている場合がある．これを衝突といい，節点 s の Base 値，または既存節点 u の遷移元の Base 値を変更する必要がある [1]．節点 s における Base 値は，$\mathrm{T}(s)$ に含まれるすべての節点に関係するため，節点 s の Base 値を変更するときは，$\mathrm{T}(s)$ に含まれる節点を削除し，新規の $\mathrm{T}(s)$(以下，新規節点集合) を作成する．新規節点集合を効率的に取得するために，ダブル配列の未使用要素を双方向リスト (以下，未使用要素リスト) として連結する手法が提案されている [2][3]．

矢田らは，削除した節点 (以下，削除節点) を未使用要素リストの先頭に追加する手法 [2](以下，LIFO 方式) を提案した．この手法は，新規節点集合の取得の際に，未使用要素リスト上の削除節点を必ず走査し，空間効率の良いダブル配列を構築する．しかし，削除節点の周辺には既存節点が多く存在する可能性が高いため，削除節点は新規節点集合の要素とならない可能性が高い．これにより，未使用要素リストの前方に削除節点が蓄積すると，新規節点集合の取得に要する未使用要素リストの走査回数が増加する．

中村らは，未使用要素リスト上の削除節点を走査せず新規節点集合を取得するために，削除節点を未使用要素リストの最後尾に追加する手法 [3](以下，FIFO 方式) を提案した．この手法では，前方に削除節点が存在せず，LIFO 方式よりも未使用要素リストの走査回数が抑制され，より高速に新規節点集合を取得できる．しかし，削除節点が再利用されない場合が多く，未使用要素リストの後方に削除節点が蓄積し，ダブル配列の空間効率が低下する．

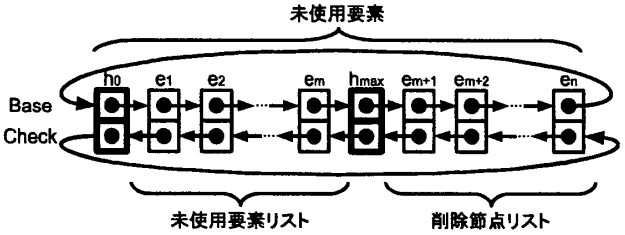

図１　未使用要素リスト及び削除節点リスト

いずれの方式で未使用要素を管理しても，遷移先節点集合を求めるためには，ダブル配列上を遷移種の数だけ線形に探索する必要がある．中村らは構築速度をさらに高速化するため，配列 Sibling を設けて遷移先節点集合をリスト構造で管理する手法 [3] を提案した．この手法では，遷移先節点を１つ見つけるまで探索し，遷移先節点からなるリストである遷移先節点集合を取得する．しかし配列 Base や Check と同じサイズの配列 Sibling を追加するため，領域が 1.5 倍に増加する．

3．　更新処理の高速化手法

3.1　削除節点を利用したシングル節点の作成

一方向分岐による遷移先節点 (以下，シングル節点) を作成する場合，未使用要素１つで新規節点になり得るため，削除節点を再利用できる．これに着目し，削除節点を別の未使用要素リスト (以下，削除節点リスト) として管理し，シングル節点を作成する場合は，削除節点リストから新規節点を作成し，それ以外の場合は未使用要素リストから新規節点を作成する手法を提案する．

未使用要素を式 (2)，図 1 に示すように双方向リストとして連結し，Check 値の符号によって，使用中の節点と区別する．ダブル配列上の未使用要素を $e_1, e_2, \cdots, e_m, e_{m+1}, \cdots, e_n$ とし，特に $e_{m+1}, \cdots, e_n$ を削除節点とする．ダブル配列の先頭を h_0，最後尾を h_{max} と定義し，h_0 と h_{max} をそれぞれ未使用要素リスト及び削除節点リストの先頭と終端として使用する．すなわち，$h_0, e_1, \cdots, e_m, h_{max}$ を未使用要素リスト，$h_0, e_n, \cdots, e_{m+1}, h_{max}$ を削除節点リストとする．

$$\begin{cases} \mathrm{Base}[e_i] = e_{i+1} & (1 \le i < m,\ m+1 \le i < n) \\ \mathrm{Base}[e_m] = h_{max}, & \mathrm{Base}[e_n] = h_0 \\ \mathrm{Check}[e_i] = -e_{i-1} & (1 < i \le m,\ m+1 < i \le n) \\ \mathrm{Check}[e_1] = -h_0, & \mathrm{Check}[e_{m+1}] = -h_{max} \\ \mathrm{Base}[h_0] = e_1, & \mathrm{Check}[h_0] = -e_n \\ \mathrm{Base}[h_{max}] = e_{m+1}, & \mathrm{Check}[h_{max}] = -e_m \end{cases} \tag{2}$$

以下に示す関数 Put により，削除節点 *Enode* を未使用要素リストではなく，削除節点リストに追加する．式 (2) より，関数 Put は，削除節点を未使用要素リストの最後尾に連結する FIFO 方式と同じ動作となる．

関数 Put(*Enode*)

まず，*next* に削除節点リストの先頭 $-\mathrm{Check}[h_0]$ をセットする．次に，削除節点リストにおける前方へのリンクを連結するた

[†] 大阪教育大学, Osaka Kyoiku University

め，Base[$Enode$] に Base[$next$] を，Base[$next$] に $Enode$ を格納する．後方へのリンクを連結するため，Check[$Enode$] に $-next$ を，Check[Base[$next$]] に $-Enode$ を格納する． [関数終]

以下に示す関数 Get は，削除節点リストからシングル節点となる新規節点を取得し戻り値とする．削除節点リストから新規節点を取得するのは，シングル節点を作成するときのみである．削除節点リストが空の場合は，未使用要素リストの先頭からシングル節点用の新規節点を取得する．シングル節点を多く必要とするのは，追加のための探索が配列 Tail 内で失敗したときである．このとき，配列 Tail 内の文字列と探索キーの共通部分の文字による遷移をトライに追加するため，シングル節点を共通文字の数だけ新規に作成する．関数 Get 内で使用される関数 Empty は，削除節点リストが空 ($-\mathrm{Check}[h_0] \neq h_{max}$) ならば真を返す．

関数 Get

まず，関数 Empty の戻り値が真ならば，シングル節点 s に削除節点リストの先頭 $-\mathrm{Check}[h_0]$ をセットする．そうでなければ，s に未使用要素リストの先頭である $\mathrm{Base}[h_0]$ をセットする．次に，Base[$-$Check[s]] に Base[s] を，Check[Base[s]] に $-$Check[s] を格納し，リンクを連結する． [関数終]

提案手法では，未使用要素リストの前方に削除節点が集中する LIFO 方式の問題点が解決されるため，新規節点集合を取得する際の時間計算量が抑制できる．また，削除節点リストをシングル節点の作成時に再利用するため，追加する未使用要素が再利用されない場合が多い FIFO 方式の問題点が解決され，高い空間効率を保つことができる．

3.2 遷移先節点集合の管理法

遷移先節点集合を高速に取得するために，遷移先集合をリスト構造で管理する一次元配列 Sibling を設ける．中村ら [3] は配列 Sibling に節点番号を格納したが，式 (1) より，節点 s の遷移先節点 t は，節点 s の Base 値に遷移種を加えたものであるため，配列 Sibling には遷移種を格納すればよい．これにより，配列 Sibling の 1 要素あたりの記憶領域を抑制できる．節点 s の遷移集合 $\mathrm{L}(s)$ を，配列 Sibling を用いて，式 (3) に示す循環リストで管理する．

$$\begin{cases} \mathrm{Sibling}[\mathrm{Base}[s]+a_i] = a_{i+1} & 1 \leq i \leq K-1 \\ \mathrm{Sibling}[\mathrm{Base}[s]+a_K] = a_1 \end{cases} \quad (3)$$

節点 s からの遷移先節点を 1 つ取得するために，一次元配列 Child を設け，遷移先節点への遷移情報を管理する．Child[s] には，$\mathrm{L}(s)$ のうち，最初に登録した遷移種を格納する．

配列 Sibling と Child を用いて，節点 s の遷移先節点集合 T を得る関数 GetTnodes を以下に示す．

関数 GetTnodes(s, T)

手順 1 :遷移先節点の取得

遷移先節点 t に Base[s]+Child[s] をセットする．

手順 2 :遷移先節点集合の取得

集合 T に t を追加し，t に次の遷移先節点 Base[s]+Sibling[t] をセットする．このとき，Sibling[t] と Child[s] が同じであれば終了し，そうでなければ手順 2 を繰り返す． [関数終]

4． 実験による評価

提案手法の有効性を示すため，矢田らの手法 [2](以下，手法 Y) と中村らの手法 [3](以下，手法 N) との比較実験を Intel Pentium Dual 1.60GHz, Fedora7 上で行った．ただし，提案手法にあわせて，手法 N の配列 Sibling には遷移種を格納した．また，提案手法の未使用要素リストを評価するため，提案手法の未使用要素リストを LIFO 方式と FIFO 方式にしたものをあわせて実験した．実験では，ランダムに抽出した 30 万語の英単語を母集団とし，1 万語から 30 万語まで 1 万語おきにキー数を増やした部分集合を用いた．

表 1 英単語 30 万語における実験結果

	提案	手法 Y	手法 N	LIFO	FIFO
構築時間（秒）	0.45	1.91	0.81	0.55	0.36
衝突回数（M 回）	0.28	0.29	0.20	0.29	0.20
走査 T(M 回)	3.19	229.36	78.24	3.37	2.27
走査 F(M 回)	2.52	7.73	0.93	8.79	0.93
使用領域（MB）	5.97	4.50	6.64	5.62	7.37
空間効率（%）	94.12	99.89	76.16	99.92	76.16

ダブル配列における空間効率を，最大節点までの使用率として定義する．表 1 に 30 万語における構築時間，衝突回数，遷移先節点集合を取得する際の走査回数 (表中，走査 T)，未使用要素リストから新規節点集合を取得する際の走査回数 (表中，走査 F)，空間効率を示す．

表 1 に示すように，配列 Child と Sibling の使用により，提案手法は走査 T の回数が手法 N，Y よりも少ない．また，提案手法は削除節点を未使用要素リストに追加せず，削除節点リストに追加するため，LIFO 方式よりも走査 F の回数が少くなり，構築時間を短縮した．さらに，提案手法は削除節点リストをシングル節点の作成に使用するため，表 1 より，FIFO 方式に対して高い空間効率を保つことができる．

30 万語の構築実験において提案手法は，配列 Child と Sibling を用いない手法 Y に対し，使用領域は 1.33 倍になるが，構築時間は 0.23 倍となった．また，FIFO 方式で未使用要素リストを管理し，配列 Sibling を用いる手法 N に対しては，使用領域が 0.90 倍，構築時間が 0.55 倍となり，コンパクト性を維持しつつ更新処理を高速化した．

5． おわりに

本論文では，未使用要素リストを 2 つに分けた管理と，2 本の一次元配列による遷移先節点集合の管理によって，更新処理をより高速化する手法を提案し，実験によりその有効性を示した．今後の課題として，提案手法を用いた削除処理の評価が挙げられる．

参考文献

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

[2] 矢田普，大野将樹，森田和宏，泓田正雄，吉成友子，青江順一，接頭辞ダブル配列における空間効率を低下させないキー削除法，情報処理学会論文誌，Vol.47，No.6，pp.1894-1902，2006.

[3] 中村康正，野村優，望月久稔，遷移先節点集合を導入したトライ構造における更新手法の実現，情報処理学会研究報告 (FI-82/DD-54)，No.33，pp.1-6，2006.