情報処理学会論文誌　データベース　Vol. 2　No. 2　96–109 (June 2009)

ダブル配列を用いたACマシンにおける遷移の分岐別管理による効率的な辞書構造の実現

蔵満 琢麻[†1]　望月 久稔[†2]

自然言語処理における辞書構造として，トライ法が広く用いられているが，日本語のように分かち書きされていない言語のテキストからキーワードを検出するためには，解析対象となるテキストのあらゆる位置から探索する必要がある．より高速に形態素解析を行うため，複数のキーワードをテキストから線形時間で検出するAC法を用いる手法が提案されているが，AC法はトライ法よりも使用する記憶領域が大きい．本論文では，ACマシンにおける遷移のうち，多分岐の節点における遷移をダブル配列に，1方向分岐の節点における遷移をダブル配列と異なる配列にそれぞれ定義することで，照合時に必要な記憶領域を抑制し，高速性とコンパクト性をあわせ持つACマシンを実現する手法を提案する．日本語形態素40万語を登録した実験で，提案手法はトライを用いた辞書システムDartsとほぼ同等の記憶領域で対象テキストを60〜87%の時間で照合した．

An Efficient Implementation of Aho-Corasick Machine Using Double-array by Transition Management Based on Branch

Takuma Kuramitsu[†1] and Hisatoshi Mochizuki[†2]

Trie structure is used widely, such as dictionary for natural language processing. However, it is not so effective using a trie structure for the morphological analysis of languages without explicit word boundaries like Japanese because we have to perform dictionary lookup for all possible substrings of the text. This paper proposes an efficient dictionary structure that is Aho-Corasick Machine using Double-array defining multi-way branch and different arrays defining one-way branch. Our experiments show that the matching time of the proposal machine decreased to about 60%–87% against other structures.

1. はじめに

近年，インターネットなどの情報ネットワークの発達や，コンピュータの性能向上により大量の情報が溢れている．膨大な量の情報の中から必要な情報を見つけ出すためには効率的な情報検索技術が必要不可欠であり，この情報検索には，索引の高速性と記憶領域のコンパクト性をあわせ持つ手法が求められる．

トライ法は自然言語処理の辞書を中心に広く用いられる検索手法である．トライは，登録キー集合における各キーの共通する接頭辞を併合した木構造で，キーの探索速度が登録キー数に依存せず，共通の接頭辞を持つキーの探索（以下，共通接頭辞探索）が容易な構造を持つ．しかし，日本語のように分かち書きされていない言語のテキストからキーワードを検出するためには，テキストのあらゆる位置から共通接頭辞探索を行う必要がある．

そこで，より高速に形態素解析を行うために，複数文字列マッチングに用いられるAC法[1]を辞書として用いる手法[2)–4)]が提案されている．AC法はACマシンと呼ばれる一種の有限オートマトンを登録キー集合から構築し，対象テキストを1度走査することで，テキストに含まれるすべてのキーを検出する手法である．しかし，ACマシンはトライよりも，照合時に必要な記憶領域が大きく，従来の形態素解析システム[5),6)]はトライ法を形態素辞書として採用している．ここで，ACマシンをコンパクトに実現できれば，効率的な辞書構造としてAC法を利用できると考えられる．

ACマシンを実用的な記憶領域で実現する手法として，有川らは，キーの構成要素を分割して取り扱うことで，各節点における遷移先を定義する配列のサイズを抑制する手法[7)]を提案した．この手法はキーの構成要素を細分化するほど，1節点あたりの記憶領域を抑制できるが，照合時の遷移回数が増加し，照合速度が低下する．また，信種らは，トライを高速かつコンパクトに実現するダブル配列[8)]を拡張したACマシン[9)]を提案した．しかし，拡張による節点の追加により，トライと比較して節点数が大幅に増加する．

本論文では，ダブル配列を用いたACマシンにおいて，遷移先の候補が一意に決定できる1方向分岐節点における遷移先を，ダブル配列と異なる配列に定義することでACマシンの節点数を抑制し，さらに高速性とコンパクト性を向上させる手法を提案する．

†1 大阪教育大学大学院教育学研究科
Graduate School of Education, Osaka Kyoiku University
†2 大阪教育大学教育学部教養学科情報科学講座
Information Science, Osaka Kyoiku University

以下，2 章でトライ法，AC 法を実現する既存手法について説明し，3 章で提案手法について述べる．その後，4 章で提案手法に理論的，実験的評価を与え，5 章で本論文の総括と今後の課題を述べる．

2. トライ法，AC 法

本章では，トライ法，AC 法を用いたテキスト照合についてそれぞれ説明し，AC 法を形態素辞書として扱う利点を述べる．

2.1 トライ法による照合

本節ではトライを効率的に実現するダブル配列について説明し，トライ法によるテキスト照合の問題点に触れる．その後，ダブル配列の記憶領域をさらに抑制する手法を説明する．

2.1.1 ダブル配列

ダブル配列[8] は 2 つの配列 Base，Check を用いてトライを実現する手法であり，配列における添字は節点番号を表す．以下，ダブル配列を DA と呼び，DA 上の節点 s における配列 Base，Check の要素を，それぞれ $B[s]$，$C[s]$ と表記し，遷移種 a の内部表現値を $A(a)$ と表記する．DA は，節点 s から遷移種 a による遷移先節点 t を式 (1) により一意に決定し，1 節点あたりの記憶領域が Base 値と Check 値のみであるため，高速性とコンパクト性をあわせ持つデータ構造である．

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

DA は，終端遷移 ‘#’ を登録キーの末尾に付加し，トライの葉とキーを 1 対 1 対応させ，キーのインデクスを終端遷移による遷移先の節点（以下，終端節点）の Base 値に負値で格納する．キーのインデクスはレコード情報へのリンクとして用いられる[5]．

図 1 にキー集合 K = {“A”, “ABA”, “ACB”, “BACAA”, “BACAB”} を登録した DA を示す．ここで，登録キー集合におけるキーの構成要素の集合 Σ を {‘A’, ‘B’, ‘C’} とし，遷移種 ‘A’，‘B’，‘C’，‘#’ の内部表現値をそれぞれ 0，1，2，3 とする．また，キー “A”，“ABA”，“ACB”，“BACAA”，“BACAB” のインデクスをそれぞれ，1，2，3，4，5 とする．

例 1：図 1 を用いてテキスト T = $A_1A_2B_3A_4C_5A_6B_7$ における照合例を示す．テキスト T における遷移種の添字はテキストにおける位置を表す．

まず，テキストの先頭 A_1 から共通接頭辞探索を開始する．初期節点 0 において，$B[0] + A(\text{‘A}_1\text{’}) = 1$，$C[1] = 0$ で遷移が成立し，節点 1 へ遷移する．ここで，$C[B[1] + A(\text{‘\#’})] = C[5] = 1$ より，節点 1 が終端節点 5 への遷移を持つため，節点 5 の Base 値を参照してキー 1（“A”）を検出する．次に，節点 1 から遷移種 ‘A_2’ による遷移の有無を確認するが，$C[B[1] + A(\text{‘A’})] = C[2] \neq 1$ より，遷移に失敗するため，共通接頭辞探索の開始位置を A_2 へと 1 つ進め，再び初期節点 0 から探索を始める．遷移種 ‘A_2’ により節点 1 へ遷移してキー 1 を検出し，遷移種 ‘B_3’，‘A_4’ により節点 3，6 へ遷移してキー 2（“ABA”）を検出する．以下，同様にテキストの末尾まで共通接頭辞探索を繰り返して終了する．（例終）

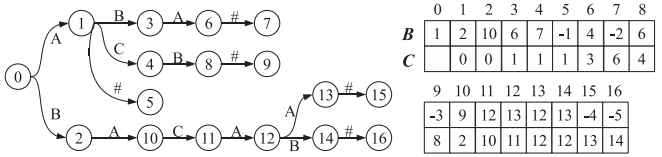

図 1　キー集合 K を登録したダブル配列
Fig. 1　Double-array for key set K.

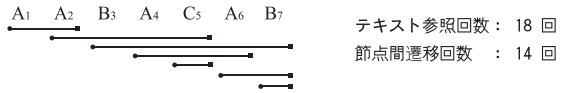

図 2　トライ法を用いた照合
Fig. 2　String matching using trie.

図 2 に，上記例において各共通接頭辞探索により参照したテキストの文字を下線で示す．テキストを参照する回数，トライ上の節点間を遷移する回数もあわせて示す．図 2 からも分かるように，トライ法を用いてテキストを照合する場合，テキスト内の同じ文字を複数回参照する必要がある．また，遷移に失敗するまで次の共通接頭辞探索を開始できないため，入力が逐次的に行われるオンライン処理には不向きであるといえる．

形態素解析エンジンの茶筅[5] や Mecab[6] が形態素辞書として利用している Darts[10] は図 1 と同様のデータ構造である．

2.1.2 ダブル配列における記憶領域の抑制手法

ダブル配列 DA は配列 Tail を用いて接尾辞を遷移列として管理することでさらに記憶領域を抑制できる[8),11]．また，近年では配列 Check に節点番号の代わりに遷移種の内部表現値を格納することで 1 節点あたりの記憶領域を抑制する手法[12] が提案されている．以下，SP 節点，シングル節点，マルチ節点について説明した後，各手法について説明する．

トライにおいて遷移先の候補が一意に決まる節点を 1 方向分岐節点と呼び，探索するキー

を一意に決定できる 1 方向分岐節点をセパレート節点（以下，SP 節点）と呼ぶ．また，遷移元の節点が等しい節点を兄弟節点と呼び，兄弟節点が存在しない節点をシングル節点，存在する節点をマルチ節点と呼ぶ．図 1 に示す DA では，節点 2〜4，6，8，10，11，13，14 が 1 方向分岐節点であり，節点 3，4，13，14 が SP 節点に該当する．また，節点 6〜12，15，16 がシングル節点で，節点 1〜5，13，14 がマルチ節点である．

SP 節点以降はすべてシングル節点であるため，遷移先の候補は一意に決まる．DA は記憶領域をさらに抑制するため，このシングル節点からなる遷移列を配列 Tail により管理し，SP 節点の Base 値に遷移列へのリンクとして配列 Tail の添字を負値で格納する．配列 Tail 内においては，キーのインデクスを終端遷移 ‘#’ の隣接要素に格納する[8]．配列 Tail は 1 要素あたりの記憶領域が DA より小さいため，照合時の記憶領域を抑制できる．また，配列 Tail 内の遷移は，単に文字どうしの比較であるため，DA 上の遷移よりも高速である．以下，配列 Tail における添字 i の要素を $T[i]$ と表記する．

次に，配列 Check に節点番号の代わりに遷移種の内部表現値を格納することで 1 節点あたりの記憶領域を抑制する手法について説明する．遷移先を決定するための基底値である Base 値が重複しないように条件を設けると，配列 Check に格納する値を節点番号ではなく，遷移種の内部表現値を格納することで遷移を定義できる[12]．Check 値に遷移種を格納する DA の遷移定義式を式 (2) に示す．遷移種を表現するためのバイト幅は，配列の全要素を表現するためのバイト幅よりも小さい場合が多いため，配列 Check の 1 要素あたりの記憶領域を抑制できる．たとえば，遷移種が 2 バイト，配列の全要素が 4 バイトで表現できる場合，節点番号による配列 Check と比較して半分の記憶領域で遷移種による配列 Check を実現できる．

$$\begin{cases} t \leftarrow B[s] + A(a) \\ C[t] = A(a) \end{cases} \tag{2}$$

以下，Check 値を遷移種の内部表現値で管理し，SP 節点以降を配列 Tail に格納した MP トライ（最小接頭辞トライ，Minimal-Prefix トライ）[11] を MPDA と呼ぶ．**図 3** にキー集合 K を登録した MPDA を示す．節点 3，4，9，10 が SP 節点に該当し，それぞれ $T[1]$，$T[4]$，$T[7]$，$T[9]$ へのリンクを Base 値で管理する．

2.2　AC 法による照合

AC 法は，トライを拡張して AC マシンと呼ばれる一種の有限オートマトンを構築し，テキストを入力として与えることで，テキストに含まれるすべてのキーの出現位置を検出する手法である．以下，AC マシンについて説明し，既存手法について述べる．AC マシンは Goto 関数，Failure 関数，Output 関数から構成される[1]．Goto 関数は節点 s から遷移種 a による遷移先節点 t を返し，Goto$(s, a) = t$ と表す．遷移が未定義である場合は失敗を表す fail を返す．Failure 関数は Goto 関数が fail を返した際に呼び出され，遷移失敗時における節点 s からの遷移先節点 f を返し，Failure$(s) = f$ と表す．以下，Goto 関数，Failure 関数による遷移をそれぞれ Goto 遷移，Failure 遷移と呼ぶ．Output 関数は，Goto 遷移により到達した節点 s において検出するキー集合 O_s を出力し，Output$(s) = O_s$ と表す．以下，Output 関数により出力するキー集合 O_s を Output 集合と呼ぶ．

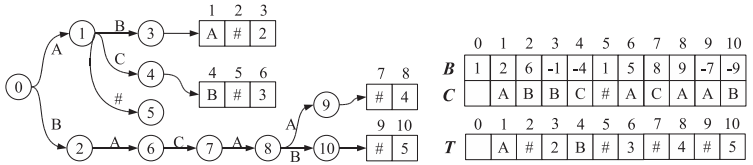

図 3　キー集合 K を登録した MPDA
Fig. 3　Minimal prefix double-array for key set K.

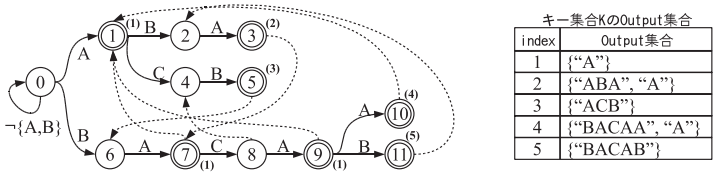

図 4　キー集合 K を登録した AC マシン
Fig. 4　Aho-Corasick machine for key set K.

図 4 にキー集合 K を登録した AC マシンを示す．図 4 において実線が Goto 遷移，破線が Failure 遷移を表す．ただし，破線が存在しない節点の Failure 遷移先は初期節点とする．二重丸の節点が Output 集合を持つ節点であり，括弧内の数値は Output 集合のインデクスを表す．図 4 内の表はインデクスに対応する Output 集合を示す．

例 2：図 4 を用いてテキスト T における照合例を示す．まず初期節点 0 において，Goto(0, ‘A_1’) = 1 より，節点 1 へ Goto 遷移する．ここで，節点 1 が Output 集合を持つため，Output(1) = {“A”} を検出する．次に，Goto(1, ‘A_2’) = fail より Failure 関数を呼び出し，Failure(1) = 0 より初期節点へ Failure 遷移する．その後，遷移種 ‘A_2’ により節点 1 へ Goto 遷移し，Output 集合 {“A”} を検出する．同様に遷移種 ‘B_3’，‘A_4’ により節点 2，

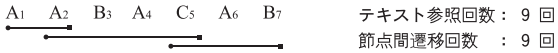

図 5　AC 法を用いた照合
Fig. 5　String matching using Aho-Corasick algorithm.

3 へ Goto 遷移し，Output 集合 {“ABA”, “A”} を検出する．以下，遷移を繰り返し，節点 9 で {“A”}，節点 11 で {“BACAB”} を検出して照合を終える．（例終）

図 5 に，上記例において Goto 遷移の有無を判定したテキストの文字を下線で示す．テキストを参照する回数，AC マシン上の節点間を遷移する回数もあわせて示す．トライを用いるよりも，テキスト参照回数，節点間遷移回数が少ないことが分かる．また，AC 法はテキストを逐次的に照合する（テキストをバックトラックしない）ため，トライ法よりもオンライン処理に適しているといえる．

このように，AC 法はトライ法よりも高速にテキストからキーを検出できる．さらに，AC マシンは登録キー集合のトライ構造を含むため，ある文字列を辞書から検索する通常の索引も可能である．これらの性質から AC マシンを形態素辞書として用いることがトライを用いるよりも機能面において有効であるといえる[4]．しかし，AC マシンは Failure 遷移や Output 集合の管理が必要であるため，トライよりも記憶領域が大きい．以下，AC マシンを実現する既存手法について説明する．

2.2.1　配列構造を用いた AC マシン

AC マシンは一種の有限オートマトンであるため，各節点にすべての遷移種による Goto 遷移先を定義し，決定性有限オートマトンに変換することで照合速度をさらに高速化できる[1),7)]．各節点における遷移先を遷移種数分の要素を持つ配列構造を用いて定義できるが，登録キー数の増加により節点数が増加することで記憶領域が膨大になる．有川らはキーの構成要素を分割して遷移種の数を削減し，遷移を定義するための配列（以下，遷移先管理配列）のサイズを抑制する手法[7)] を提案した．この手法はデータベースから任意の情報を取り出すシステム[13)] に利用されている．遷移種を細かく取り扱うほど照合時の記憶領域を抑制できるが，遷移種の分割によりテキストを走査する回数が増加して照合速度と記憶領域がトレードオフ[7)] の関係になり，形態素辞書のように膨大な数のキーを登録するには適していない．照合速度，記憶領域に関しての考察は 4 章で後述する．

2.2.2　ダブル配列を用いた AC マシン

信種らはダブル配列 DA を用いて AC マシンを構築する手法を提案した[9)]．以下，信種らにより提案された AC マシンをマシン X と呼ぶ．マシン X はトライを構成する Goto 遷移

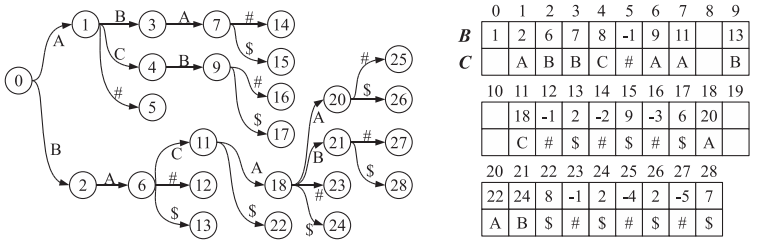

図 6　キー集合 K を登録したマシン X
Fig. 6　Machine-X for key set K.

を DA の遷移定義式 (2) を用いて定義し，Output 集合のインデクスを，Output 集合を持つ節点に終端節点への遷移を定義して管理する．ただし，初期節点から初期節点への Goto 遷移が未定義になるため，照合時に初期節点からの Goto 遷移に失敗したときは，テキストの照合ポインタを 1 つ進めることで Goto 遷移を実現する[9)]．

マシン X は，記憶領域を抑制するため，節点を用いて Failure 遷移を定義する．以下，DA の遷移式 (2) を用いて，同一節点へ複数の遷移を擬似的に定義する方法について説明し，マシン X における Failure 遷移の定義方法を説明する．

Check 値を遷移種で管理する遷移式 (2) では，同じ Base 値を持つ節点は等しい遷移を持つ[9),14)]．そこで，既存の節点 s への遷移を新たに定義する場合，節点 s と等しい Base 値を持つ節点 s' を作成し，節点 s' への遷移を定義することで節点 s への遷移を擬似的に定義できる[14)]．以下，既存節点への遷移を定義するために作成する節点を擬似節点と呼ぶ．

マシン X は初期節点以外への Failure 遷移先を持つ節点 s に，キーの構成要素に含まれない遷移種 ‘$’（$\neq$‘#’, $\notin \Sigma$）による遷移先節点 s_f を作成し，節点 s_f に Failure 遷移先節点の Base 値を格納することで Failure 遷移を定義する．以下，Failure 遷移を定義するために作成する擬似節点 s_f を Failure 節点と呼ぶ．照合時において Goto 遷移に失敗した場合，マシン X は Failure 節点への遷移が存在すれば Failure 節点へ，存在しなければ初期節点へ Failure 遷移する[9)]．Failure 節点を付加することで節点数は増加するが，Failure 遷移先を定義するための領域を各節点に設けるよりも 1 節点あたりに必要な記憶領域を抑制できるため，照合時に必要な記憶領域の抑制を図れる．

図 6 にキー集合 K を登録したマシン X を示す．節点 13，15，17，22，24，26，28 が Failure 節点であり，それぞれ節点 1，6，2，4，1，1，3 と等しい Base 値を持つ．

初期ノードからノード s までの遷移により構成される文字列を str(s) とすると，ノード s

における Failure 遷移先は，str(f)（$f \neq s$）が str(s) の接尾辞となるノード f である．マシン X における Failure 節点は Failure 遷移先が初期節点である節点には作成されないが，登録キー数が増加すると，トライ上の深さが浅い節点では大半の遷移種による Goto 遷移先が定義され，初期節点へ Failure 遷移する節点が減少する[14]．そのため，大規模なキー集合を登録すると，ほとんどすべての節点に Failure 節点を作成することになり，節点数がトライよりも大幅に増加する．

3. AC マシンの 1 方向分岐における遷移計算量の抑制

AC マシンを従来の手法よりもさらにコンパクトに実現できれば，高速性とコンパクト性をあわせ持つ辞書構造として自然言語処理に応用できると考えられる．ダブル配列 DA は SP 節点以降の 1 方向分岐を配列 Tail に格納して 1 方向分岐における記憶領域と遷移計算量を抑制する．本章では，DA を用いた AC マシンにおけるすべての 1 方向分岐を他の配列を用いて管理することで，記憶領域の抑制と照合速度の高速化を図る手法を提案する．

まず，提案マシンにおける Goto 遷移の定義方法について説明し，新たに定義する配列について述べる．次に提案マシンにおける Failure 遷移の定義方法について説明し，節点数の抑制方法を説明する．その後，Output 集合の取扱いについて説明し，提案マシンの各遷移を定義式にまとめる．最後に提案手法の照合アルゴリズム，構築アルゴリズムについて，それぞれ例を用いて解説する．

3.1 Goto 遷移の定義

本節では，まず多分岐節点における Goto 遷移の定義方法について述べ，次に 1 方向分岐節点における Goto 遷移の定義方法について述べる．

提案マシンは多分岐節点の Goto 遷移を DA の遷移定義式 (2) を用いて定義する．ただし，初期節点から初期節点への Goto 遷移を，提案マシンは擬似節点を用いて定義する．初期節点からすべての遷移種による遷移先を定義することで，Goto 遷移失敗時に参照中の節点が初期節点であるか判定することなくテキストを照合できる[14]．

節点 s が 1 方向分岐節点である場合，Goto 遷移先の候補は一意に決まるため，連続するシングル節点への遷移を遷移列として管理することが考えられる．そこで，提案手法ではシングル節点を格納するための領域として，新たに配列 SingleBase と SingleCheck を設ける．以下，配列 SingleBase，SingleCheck を合わせて SDA と呼び，SDA 上の節点 t におけるそれぞれの配列の要素を $SB[t]$，$SC[t]$ と表す．

$SC[t]$ には，DA 上の節点の Check 値と同様に，節点 t へ遷移するための遷移種を格納する．マルチ節点 s から n 個のシングル節点が連続する遷移として存在する場合，連続する SDA 上の節点 $t_1, t_2, \ldots, t_n$ の SingleCheck 値に各遷移種 $a_1, a_2, \ldots, a_n$ の内部表現値を順に格納し，$SC[t_{n+1}]$ にシングル節点の終端を表して確実に Goto 遷移に失敗する遷移種 '$' (≠'#', ∉ Σ) の内部表現値を格納する．このとき，連続するシングル節点の末となる節点 t_n は多分岐であるか，Goto 遷移が存在しない．たとえば，図 4 において，節点 7，8，9 は連続するシングル節点であり，シングル節点 9 は多分岐である．また，トライ構造上の葉に該当するシングル節点 3 には Goto 遷移が存在しない．提案マシンにおいて，シングル節点 t_n が多分岐である場合は $SB[t_{n+1}]$ に DA 上の各 Goto 遷移先への基底値を格納し，Goto 遷移先が存在しない場合は直前の節点 t_n の Failure 遷移先の遷移情報を格納する．Failure 遷移先の遷移情報に関しては次節で解説する．提案マシンは，DA 上の節点 s から SDA 上のシングル節点 t_1 への遷移を，節点 s の Base 値に節点 t_1 の節点番号を負値で格納することで定義する．

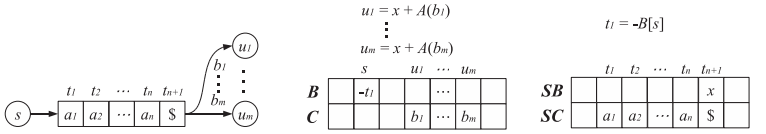

図 7　提案マシンの Goto 遷移
Fig. 7　Goto function of proposal machine.

提案マシンの Goto 遷移の定義方法を**図 7** に示す．提案マシンの状態遷移図において，丸枠を DA 上の節点，四角枠を SDA 上の節点とする．DA 上の節点 s は 1 方向分岐であり，SDA 上のシングル節点 t_1 への Goto 遷移を持つ．SDA 上のシングル節点 t_n は，Base 値 x を基点とする m 個の Goto 遷移先節点 $\{u_i;\ u_i = x + A(b_i),\ b_i \in \Sigma,\ 1 \leq i \leq m\}$ を DA 上に持つ多分岐節点であり，提案マシンは SDA 上の節点 t_{n+1} の SingleBase 値に Base 値 x を格納することで SDA 上から DA 上への Goto 遷移を定義する．

3.2 Failure 遷移の定義

本節では，まず多分岐節点における Failure 遷移の定義方法について述べ，次に，1 方向分岐節点における Faillure 遷移の定義方法について述べる．

節点 s における Failure 遷移先の節点を f とし，節点 f における Goto 遷移先節点の集合を $G(f)$ とする．提案マシンは，初期節点以外への Failure 遷移先を持つ多分岐節点 s において，Failure 節点 s_f への遷移を DA 上に作成し，式 (3) により Failure 遷移先の遷移情報を Failure 節点 s_f の Base 値に格納することで Failure 遷移を定義する．Failure 遷移

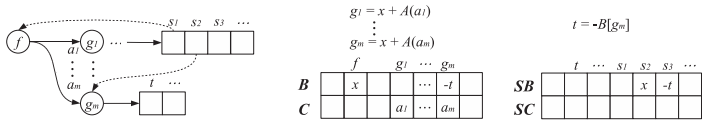

図 8　SDA 上の Failure 遷移
Fig. 8　Failure-transition on SDA.

定義式 (3) において，$base$ は節点 f の Goto 遷移先集合の基点を表し，a_i は遷移種を表す．また，t はシングル節点である．

$$B[s_f] = \begin{cases} base,\ (G(f) \ = \ \{base + A(a_i);\ i = 1,\ 2,\ \ldots,\ m\ (m > 1)\}) \\ -t, \quad (G(f) \ = \ \{t\}) \end{cases} \tag{3}$$

次に 1 方向分岐節点における Failure 遷移の定義方法について説明する．提案マシンは 1 方向分岐節点 s における Failure 遷移を，Goto 遷移先節点の SingleBase 値に式 (3) に示す値を格納することで定義する．Failure 遷移先の遷移情報を SDA 上の Goto 遷移先節点に格納することで，SDA 上の節点への遷移は Goto 遷移にも Failure 遷移にもなりうる．照合時において SDA 上の節点 s が Goto 遷移先候補であるとき，提案マシンは節点 s へ遷移し，$SC[s]$ とテキスト内の文字 a を比較する．このとき，$SC[s] = A(a)$ であれば節点 s への Goto 遷移が成立し，$SC[s] \neq A(a)$ であれば節点 s への Failure 遷移が成立する．

SDA 上の Failure 遷移の定義方法を**図 8** に示す．SDA 上の節点 s_1，s_2 はそれぞれ，DA 上の多分岐節点 f，1 方向分岐節点 g_m を Failure 遷移先として持つ．ここで，$G(f)$ は Base 値 x を基底値とする節点集合 $\{g_i;\ g_i = x + A(a_i),\ i = 1,\ 2,\ \ldots,\ m\ (m > 1)\}$ であり，$G(g_m)$ はシングル節点 t である．提案マシンは節点 s_1 の Goto 遷移先である節点 s_2 の SingleBase 値に，Failure 遷移先節点 f の遷移情報である Base 値 x を格納し，節点 s_2 の Goto 遷移先である節点 s_3 の SingleBase 値に，Failure 遷移先節点 g_m の Goto 遷移先であるシングル節点 t を負値で格納して Failure 遷移を定義する．

提案マシンでは 1 方向分岐節点において Failure 節点を作成する必要がないため，節点数の抑制を図れる．また，SDA 上の Failure 遷移は，Failure 節点の有無を確認する必要がある DA 上の Failure 遷移よりも遷移に要する計算量が少ないため，SDA 上の遷移が多いほど照合速度の高速化を図れる．

3.3　Output 集合の管理

ある節点における Output 集合は共通する接尾辞を持つキーの集合である[1] ため，キー長の長いキーから順にリンクを定義すれば，登録キー数分のリンク領域で Output 集合を管理できる．そこで，配列 Next を新たに定義し，配列の添字をキーのインデクス i（$i > 0$）と対応させる．以下，キー i における配列 Next の要素を $N[i]$ と表記する．提案マシンは，Output 集合に含まれるキーにおいて，キー長が最も長いキーのインデクスを Output 集合のインデクスとして用い，キー i の接尾辞と一致するキーのうち，キー長が最も長いキーのインデクスを $N[i]$ に格納してリンクを定義する．キー i の接尾辞と一致するキーが存在しない場合，$N[i]$ にキーのインデクスとして使用しない値 0 を格納してリンクの終端を定義する．

図 9　SDA 上の終端節点
Fig. 9　Output-node on SDA.

提案マシンもマシン X と同様に，終端節点を作成して Output 集合のインデクスを管理するが，すべての終端節点を DA 上に作成すると，多分岐節点が増加し，提案手法の利点が生かされなくなる．たとえば，図 4 において初期節点から遷移種 ‘B’，‘A’ により到達する節点 7 は 1 方向分岐であるが，図 6 において同様の遷移種により到達する節点 6 は終端節点の付加により多分岐である．そこで，提案マシンでは，Goto 遷移先が一意に決まる節点をすべて 1 方向分岐として取り扱い，1 方向分岐節点や Goto 遷移が存在しない節点における終端節点を SDA 上への遷移先として定義し，1 方向分岐節点においては Goto 遷移先を終端節点の隣接要素に定義する．

マルチ節点 x から，シングル節点 y，z が連続する遷移として存在し，各節点が Output 集合を持つ AC マシンの状態遷移図を**図 9** の左に示す．ただし，節点 y，z への遷移種をそれぞれ a_y，a_z とし，Failure 遷移と Output 集合のインデクスは省略する．提案マシンは，図 9 の右に示すように，1 方向分岐であるマルチ節点 x における終端節点 x_o を SDA 上に作成し，SDA 上のシングル節点 y，z における終端節点 y_o，z_o を，それぞれの SDA 上の隣接要素に作成する．各節点における Goto 遷移先の節点は，作成した終端節点の隣接要素に定義する．

キー集合 K を登録した提案マシンを**図 10** に示す．DA 上の節点 3 は初期節点から初期節点への遷移を定義するための擬似節点であり，初期節点と等しい Base 値を持つ．SDA 上の節点 2，5，8，11，13，15 が Output 集合のインデクスを管理する節点であり，それぞ

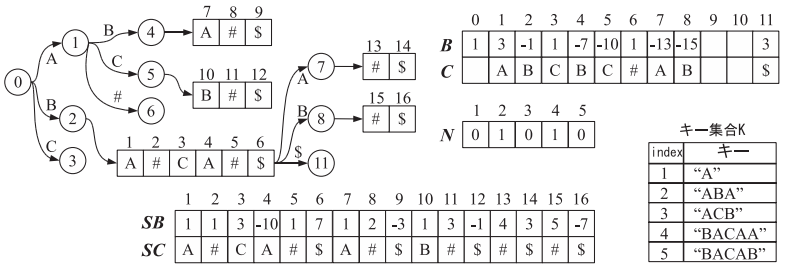

図 10　キー集合 K を登録した提案マシン
Fig. 10　Proposal machine for key set K.

表 1　提案マシンの遷移定義式
Table 1　Transition definition of proposal machine.

	マルチ節点（DA上の節点）	シングル節点（SDA上の節点）
$\|G(s)\| > 1$ 多分岐	$t = B[s] + A(a),\ C[t] = A(a)$; $s_f = B[s] + A(\text{'\$'}),\ C[s_f] = A(\text{'\$'})$;　$(f \neq \text{root})$ $s_o = B[s] + A(\text{'\#'}),\ C[s_o] = A(\text{'\#'})$;　$(O_s \neq \Phi)$	$t = \begin{cases} SB[s+1] + A(a) & (O_s = \Phi) \\ SB[s+2] + A(a) & (O_s \neq \Phi) \end{cases},\ C[t] = A(a)$; $s_f = \begin{cases} SB[s+1] + A(\text{'\$'}) & (O_s = \Phi) \\ SB[s+2] + A(\text{'\$'}) & (O_s \neq \Phi) \end{cases},\ C[s_f] = A(\text{'\$'})$;　$(f \neq \text{root})$ $s_o = s+1,\ SC[s_o] = A(\text{'\#'})$;　$(O_s \neq \Phi)$
$\|G(s)\| = 1$ 一方向分岐	$t = s_f = \begin{cases} -B[s] & (O_s = \Phi) \\ -B[s]+1 & (O_s \neq \Phi) \end{cases},\ SC[t] = A(a)$; $s_o = -B[s],\ SC[s_o] = A(\text{'\#'})$;　$(O_s \neq \Phi)$	$t = s_f = \begin{cases} s+1 & (O_s = \Phi) \\ s+2 & (O_s \neq \Phi) \end{cases},\ SC[t] = A(a)$; $s_o = s+1,\ SC[s_o] = A(\text{'\#'})$;　$(O_s \neq \Phi)$
$\|G(s)\| = 0$ Goto遷移が存在しない	$s_o = -B[s],\ SC[s_o] = A(\text{'\#'})$; $s_f = -B[s]+1,\ SC[s_f] = A(\text{'\$'})$;	$s_o = s+1,\ SC[s_o] = A(\text{'\#'})$; $s_f = s+2,\ SC[s_f] = A(\text{'\$'})$;

れ Output 集合に含まれる最長のキーのインデクスを SingleBase 値に保持する．SDA 上の終端節点 8 が管理する Output 集合に含まれるキーは，$SB[8] = 2$ よりキー 2 “ABA” と，$N[2] = 1$ よりキー 1 “A” であることが分かり，$N[1] = 0$ より他のキーは含まれないことが分かる．SDA 上の節点 1 は，SDA 上の終端節点 2 への遷移を持ち，終端節点 2 の隣接要素である SDA 上の節点 3 への遷移種 ‘C’ による Goto 遷移を持つ．

以上までの提案マシンが用いる各遷移の定義式を**表 1** にまとめる．表 1 は，初期節点を除く節点 s における，Goto 遷移先節点 t への遷移，Failure 節点 s_f への遷移，終端節点 s_o への遷移を，節点 s の分岐別に示す．各行を節点 s における Goto 遷移の数 $|G(s)|$ によって分類し，各列を節点 s がマルチ節点かシングル節点かによって分類する．表 1 において，$root$ は初期節点を表し，f は節点 s の Failure 遷移先の節点を表す．次節で表 1 に則した照合アルゴリズムについて例を用いて解説する．

3.4　照合アルゴリズム

以下に示す提案マシンの照合アルゴリズムは，テキスト $X = a_1 a_2 \ldots a_n$（$a_i \in \Sigma$）を入力すると，検出したすべてのキーのインデクスを出力する．アルゴリズム内で用いる変数 $node$ は，照合中に参照する節点を表し，変数 $base$ は遷移を切り替えるために用いる．また，変数 i は対象テキストの照合ポインタとして用いる．アルゴリズム内で用いる Output 関数は，Output 集合のインデクスを与えることで Output 集合に含まれるキーを出力する関数である．

提案手法は初期節点からすべての遷移種による Goto 遷移先を定義するため，初期節点からの遷移を一意に決定できる[14]．そこで，多分岐節点において遷移種 ‘a’ により Goto 遷移に失敗し，Failure 節点への遷移が存在しない場合，提案マシンは初期節点から遷移種 ‘a’ による遷移先の節点へ直接 Goto 遷移する．

手順 1：初期設定を行う．

$base$ に初期節点 0 の Base 値 $B[0]$ を，i にテキストの先頭位置 1 を設定する．

手順 2：DA 上で照合する．

手順 2-1：DA 上で Goto 遷移の有無を確認する．

$C[base + A(a_i)] = A(a_i)$ で遷移種 a_i による節点 $base + A(a_i)$ への Goto 遷移が存在すれば $node$ に Goto 遷移先節点 $base + A(a_i)$ を設定して手順 2-3 へ．存在しなければ手順 2-2 へ．

手順 2-2：DA 上で Failure 遷移の有無を確認する．

まず，$C[base + A(\text{'\$'})] \neq A(\text{'\$'})$ で Failure 節点への遷移が存在しなければ $node$ に初期節点からの Goto 遷移先節点 $B[0] + A(a_i)$ を設定して手順 2-3 へ．存在すれば $node$ に DA 上の Failure 節点 $base + A(\text{'\$'})$ を設定する．次に，$B[node] \geq 0$ で Failure 節点 $node$ が多分岐であれば $base$ に Goto 遷移先集合の基底値 $B[node]$ を設定して手順 2-1 へ．多分岐でなければ $node$ にシングル節点 $-B[node]$ を設定して手順 3-1 へ．

手順 2-3：DA 上で次の遷移方法を選択する．

$B[node] >= 0$ で Goto 遷移先節点 $node$ が多分岐であれば $base$ に Goto 遷移先集合の基底値 $B[node]$ を設定して手順 2-4 へ．多分岐でなければ $node$ にシングル節点 $-B[node]$ を設定して手順 3-3 へ．

手順 2-4：DA 上で Output 集合の有無を確認する．

まず，$C[base + A(\text{`\#'})] = A(\text{`\#'})$ で終端節点 $base + A(\text{`\#'})$ が存在すれば Output 集合のインデクス $B[base + A(\text{`\#'})]$ を引数にして Output 関数を呼び出す．次に，照合ポインタ i に次の照合位置 $i + 1$ を設定する．ここで，$i \leq n$ なら手順 2-1 へ．そうでなければ照合を終える．

手順 3：SDA 上で照合する．

手順 3-1：SDA 上で Goto 遷移の有無を確認する．

$SC[node] = A(a_i)$ でシングル節点 $node$ への Goto 遷移が成立すれば $node$ に終端遷移先候補 $node + 1$ を設定して手順 3-3 へ．そうでなければ手順 3-2 へ．

手順 3-2：SDA 上で次の遷移方法を選択する．

$SB[node] >= 0$ で Goto 遷移先の候補が複数存在すれば $base$ に Goto 遷移先集合の基底値 $SB[node]$ を設定して手順 2-1 へ．そうでなければ $node$ にシングル節点 $-SB[node]$ を設定して手順 3-1 へ．

手順 3-3：SDA 上で Output 集合の有無を確認する．

まず，$SC[node] = A(\text{`\#'})$ で参照中のシングル節点 $node$ が終端節点であれば Output 集合のインデクス $SB[node]$ を引数にして Output 関数を呼び出し，$node$ に次の Goto 遷移先候補 $node + 1$ を設定する．このとき，$SC[node] \neq A(\text{`\#'})$ で参照中のシングル節点 $node$ が終端節点でなければ Output 集合を出力する処理を行わない．次に，照合ポインタ i に次の照合位置 $i + 1$ を設定する．ここで，$i \leq n$ なら手順 3-1 へ．そうでなければ照合を終える．

例 3：図 10 を用いてテキスト T における照合例を示す．まず手順 1 で $base$ に初期節点の Base 値 1（$B[0]$），テキストの照合ポインタ i にテキストの先頭位置 1 を設定する．手順 2-1 において，$C[1 + A(\text{`A}_1\text{'})] = C[1] = A(\text{`A}_1\text{'})$ で DA 上に Goto 遷移先が存在するため，$node$ に Goto 遷移先節点 1 を設定する．Output 集合の有無は手順 2-3 で次の遷移方法を決定してから確認する．節点 1 の Base 値 $B[1]$ が 0 以上であるため，次回の遷移方法を DA 上の遷移に決定し，$base$ に 3（$B[1]$）を設定する．手順 2-4 で $C[3 + A(\text{`\#'})] = C[6] = A(\text{`\#'})$ より，終端節点 6 への遷移が存在するため，$\text{Output}(B[6]) = \text{Output}(1)$ を呼び出してキー 1（“A”）を対象テキストから検出する．

テキストの照合ポインタ i を 1 つ進め，手順 2-1 で遷移種 ‘A_2’ による Goto 遷移の有無を確認する．$C[3 + A(\text{`A}_2\text{'})] \neq A(\text{`A}_2\text{'})$ より，遷移種 ‘A’ による Goto 遷移が存在しないため，手順 2-2 で Failure 遷移の有無を確認する．$C[3 + A(\text{`\$'})] \neq A(\text{`\$'})$ より，Failure 遷移先が存在しないため，$node$ に初期節点からの Goto 遷移先節点 1（$B[0] + A(\text{`A}_2\text{'})$）を設定する．このように提案マシンでは，Failure 遷移が存在しない場合の Goto 遷移先を一意に決定できる．

手順 2-1 で遷移種 ‘B_3’ により節点 4 へ Goto 遷移し，手順 2-3 で次の遷移方法を選択する．ここで，$B[4] < 0$ より，次回の遷移方法を SDA 上の遷移に決定し，$node$ に終端節点候補となるシングル節点 7（$-B[4]$）を設定する．手順 3-3 において，$SC[7] \neq A(\text{`\#'})$ より，この時点ではキーを検出せずにテキストの照合ポインタ i を 1 つ進める．

次に，手順 3-1 で $SC[7] = A(\text{`A}_4\text{'})$ より，SDA 上の節点 7 への Goto 遷移が確定する．$node$ に終端遷移先候補 $node + 1$ を設定し，手順 3-3 で終端遷移の有無を確認する．$SC[8] = A(\text{`\#'})$ より，$\text{Output}(SB[8]) = \text{Output}(2)$ を呼び出してキー 2（“ABA”），1（“A”）を検出する．このように SDA 上では，$node$ をインクリメントすることで Goto 遷移や Output 集合の有無を確認できるため，DA 上の遷移よりも高速であるといえる．

以下，同様に遷移を繰り返し，SDA 上の終端節点 5 でキー 1，SDA 上の終端節点 15 でキー 5 を検出し，テキストの末尾まで照合して終了する．（例終）

3.5　構築アルゴリズム

提案マシンのデータ構造は，構築済みの AC マシンを幅優先探索し，表 1 に示す遷移定義式に従って，多分岐節点における遷移を DA 上に，1 方向分岐節点における遷移を SDA 上に写像することで実現する．以下，提案マシンの構築元となる AC マシンを AC と呼び，AC 上の節点 s を s_A，DA 上の節点 s を s_D，SDA 上の節点 s を s_S と表記し，節点 s_A における遷移の写像方法を分岐別に説明する．アルゴリズム内で用いる関数 MakeNode($base$, L) は，Base 値 $base$ を基点として遷移集合 L による遷移先節点集合 $\{base + A(a_i);\ a_i \in L\}$ を DA 上に定義する関数で，各遷移先節点の Check 値に遷移種の内部表現値を格納する．

節点 s_A が初期節点である場合

初期節点 0_D における Goto 遷移先を DA 上に定義する．まず，初期節点 0_D の Base 値に 1 を格納し，MakeNode($B[0_D]$, Σ) を呼び出してキーの構成要素の集合 Σ に含まれるすべての遷移種による Goto 遷移先節点を DA 上に定義する．ここで，初期節点から初期節点への Goto 遷移を擬似的に定義するため，AC 上の初期節点 s_A から遷移種 a による遷移先が初期節点 s_A である場合，擬似節点となる節点 $base + A(a)$ の Base 値に初期節点 0_D の Base 値 1 を格納する．

節点 s_A がマルチ節点かつ $|G(s_A)| > 1$（多分岐）である場合

節点 s_A と対応する節点 s_D における各遷移を DA 上に定義する．まず，節点 s_A の

遷移集合 L を取得し，節点 s_A が Output 集合を持つならば遷移集合 L に終端遷移 '#' を追加し，初期節点以外の節点へ Failure 遷移を持つならば遷移集合 L に失敗遷移 '$' を追加する．次に，DA 上に遷移集合 L を定義可能な Base 値 $base$ を求めて[15] 節点 s_D の Base 値に格納し，MakeNode($B[s_D]$, L) を呼び出して DA 上に遷移を定義する．

次に，節点 s_D における Output 集合と Failure 遷移を定義する．節点 s_A が Output 集合を持つならば，終端節点 $B[s_D] + A('\#')$ の Base 値に Output 集合のインデクスを格納し，節点 s_A が初期節点以外の節点へ Failure 遷移を持つならば，Failure 節点 $B[s_D] + A('\$')$ の Base 値に Failure 遷移定義式 (3) に示す値を格納する．

節点 s_A がマルチ節点かつ $|G(s_A)| \leq 1$ である場合

節点 s_A と対応する節点 s_D における遷移を SDA 上へ定義する．まず，節点 s_A から，多分岐節点か Goto 遷移が存在しない節点まで AC 上を深さ優先探索し，表 1 に従って SDA 上に定義する遷移列 T（SingleCheck 値に格納する遷移種の並び）を取得する．このとき，遷移列 T の末尾にシングル節点の終端を表す失敗遷移 '$' を追加する．次に，SDA 上の未使用である最前方の要素 t_S（> 0）から遷移列 T を格納する．このとき，SDA 上の遷移を定義するため，遷移列 T における各遷移種の内部表現値を SingleCheck 値に格納し，DA 上から SDA 上への遷移を定義するため，節点 s_D の Base 値に遷移列の開始位置 t_S を負値で格納する．

次に，節点 s_D における Output 集合と Failure 遷移を定義する．節点 s_A が Output 集合を持つならば，節点 s_D における終端節点 t_S の SingleBase 値に Output 集合のインデクスを格納し，Failure 遷移定義式 (3) に示す値を終端節点 t_S の隣接要素（$t_S + 1$）における SingleBase 値に格納する．節点 s_A が Output 集合を持たなければ，節点 s_D における Goto 遷移先のシングル節点 t_S の SingleBase 値に Failure 遷移定義式 (3) に示す値を格納して Failure 遷移を定義する．

節点 s_A がシングル節点かつ $|G(s_A)| > 1$（多分岐）である場合

節点 s_A と対応する節点 s_S における各遷移を DA 上に定義する．まず，節点 s_A の遷移集合 L を取得し，節点 s_A が初期節点以外の節点へ Failure 遷移を持つならば遷移集合 L に失敗遷移 '$' を追加する．次に，DA 上に遷移集合 L を定義可能な Base 値 $base$ を求め[15]，MakeNode($base$, L) を呼び出して DA 上に遷移先節点を定義する．

次に，節点 s_S における Output 集合と DA 上への遷移を定義するための Base 値を SDA 上に定義する．節点 s_A が Output 集合を持つならば，節点 s_S における終端節点（$s_S + 1$）の SingleBase 値に Output 集合のインデクスを格納し，終端節点の隣接要素（$s_S + 2$）の SingleBase 値に Base 値 $base$ を格納して DA 上への遷移を定義する．節点 s_A が Output 集合を持たなければ，節点 s_S における隣接要素（$s_S + 1$）の SingleBase 値に Base 値 $base$ を格納して DA 上への遷移を定義する．

最後に，節点 s_S における Failure 遷移を定義する．節点 s_A が初期節点以外の節点へ Failure 遷移を持つならば，Failure 節点 $base + A('\$')$ の Base 値に Failure 遷移定義式 (3) に示す値を格納して Failure 遷移を定義する．

節点 s_A がシングル節点かつ $|G(s_A)| \leq 1$ である場合

節点 s_A と対応する節点 s_S における Output 集合と Failure 遷移を SDA 上に定義する．節点 s_A が Output 集合を持つならば，節点 s_S における終端節点（$s_S + 1$）の SingleBase 値に Output 集合のインデクスを格納し，終端節点（$s_S + 1$）における隣接要素（s_S+2）の SingleBase 値に Failure 遷移定義式 (3) に示す値を格納する．節点 s_A が Output 集合を持たなければ，節点 s_S の隣接要素（$s_S + 1$）における SingleBase 値に Failure 遷移定義式 (3) に示す値を格納して Failure 遷移を定義する．

例 4：図 4 の AC マシンを AC とし，初期節点 0_A，多分岐のマルチ節点 1_A，1 方向分岐のマルチ節点 6_A，1 方向分岐のシングル節点 7_A における遷移を，DA 上と SDA 上へ写像する例を以下に示す．

AC 上の初期節点 0_A における遷移を写像する場合，まず，提案マシンの初期節点 0_D の Base 値に 1 を格納し，節点 1_D，2_D，3_D の Check 値にそれぞれ $A('A')$，$A('B')$，$A('C')$ を格納して節点 0_D における Goto 遷移を定義する．次に，初期節点へ遷移するための擬似節点 3_D の Base 値に初期節点 0_D の Base 値 1 を格納する．

多分岐のマルチ節点 1_A における遷移を写像する場合，まず，節点 1_A における遷移集合 $L = \{'B', 'C'\}$ を取得し，節点 1_A が Output 集合を持つため，遷移集合 L に終端遷移 '#' を加える．次に，遷移集合 L を DA 上に格納可能な Base 値 3 を求め，節点 1_A に対応する節点 1_D の Base 値に格納する．節点 4_D，5_D，6_D の Check 値にそれぞれ $A('B')$，$A('C')$，$A('\#')$ を格納して DA 上に遷移を定義し，終端節点 6_D の Base 値に Output 集合のインデクス 1 を格納して節点 1_D における Output 集合を定義する．

1 方向分岐のマルチ節点 6_A における遷移を写像する場合，まず，節点 6_A から多分岐節点 9_A まで深さ優先探索を行い，表 1 に従って SDA 上に定義する遷移列 T = 'A', '#', 'C', 'A', '#', '$'（遷移列の終端）を得る．節点 1_S〜6_S の SingleCheck 値に遷移列 T の内部表現値を順に格納して SDA 上の遷移を定義する．次に，節点 6_A に対応する節点 2_D の Base 値に遷移列 T の開始位置 1_S を負値で格納することで SDA 上への遷移を定義し，節点 2_D

の Failure 遷移先節点 0_D の Base 値 1 を節点 1_S に格納して Failure 遷移を定義する．

1 方向分岐のシングル節点 7_A においては連続する遷移列として Goto 遷移が定義済みであるため，Output 集合のインデクスと Failure 遷移のみを定義する．節点 7_A の Output 集合のインデクス 1 を，節点 7_A と対応する節点 1_S における終端節点 2_S の SingleBase 値に格納し，節点 1_S における Failure 遷移先節点 1_D の Base 値 3 を，終端節点 2_S の隣接要素 3_S における SingleBase 値に格納して Failure 遷移を定義する．（例終）

4. 評　　価

信種らにより提案された AC マシン[9]（以下，マシン X），有川らにより提案された AC マシン[7]（以下，マシン Y），Darts[10]，Check 値を遷移種として取り扱う最小接頭辞ダブル配列（以下，MPDA）を比較手法とし，照合時に必要な記憶領域，照合時間，キー集合の登録に要する構築時間について評価する．ただし，キーの構成要素を 1 バイトとして取り扱う．また，レコード情報は 4 バイトのインデクスのみとし，提案マシン，マシン X，マシン Y の Output 集合はすべて配列 Next を用いて管理する．

4.1 理論的評価

まず照合時間について評価する．照合時間を遷移回数と出力判定の和と考える．以下，テキストのサイズを n，登録キー集合の最大キー長を m とし，マシン Y が取り扱うキーの構成要素のサイズを d ビット（$d > 0$）とする．

AC マシンの Goto 遷移回数はテキストのサイズと等しい n 回であり，Failure 遷移回数は最大でも n 回である[1),7)] ため，マシン X の総遷移回数は最大で $2n$ 回である．これらの遷移に加え，提案マシンは SDA 上から DA 上へ遷移する際に，シングル節点の終端を表す Failure 節点への遷移が発生する．この遷移は連続して発生しないため，最大で n 回である．遷移が決定的であるマシン Y は Failure 遷移が発生しないが，遷移種を分割して取り扱うことで遷移回数が $8/d$ 倍に増加する．よって，提案マシン，マシン X，マシン Y における総遷移回数はそれぞれ最大で $3n$ 回，$2n$ 回，$(8/d)n$ 回である．

一方，Darts，MPDA は n 回の共通接頭辞探索を行う．各探索は遷移に失敗するまで継続するため，最大で m 回の遷移が発生する．よって，Darts，MPDA の総遷移回数は最大で mn 回である．

AC マシンにおける出力判定の回数はテキストのサイズと等しい n 回であり，トライにおける出力判定の回数は総遷移回数と等しい mn 回である．

以上より，提案マシン，マシン X，マシン Y，Darts，MPDA における照合時間はそれぞれ，$4n$，$3n$，$(8/d+1)n$，$2mn$，$2mn$ になる．大規模なキー集合を登録した場合 m は 1 よりも大きいと考えられるため，提案マシン，マシン X のほうが Darts や MPDA よりも高速であるといえる．$d = 8$ のときは，マシン Y が最も高速であるといえる．

ただし，DA は 1 遷移あたりの計算量が通常の配列構造よりも遷移式の演算分増すため，$d < 8$ のときでもマシン Y の照合速度が他手法よりも高速になる可能性がある．また，提案マシンや MPDA は遷移計算量の少ない遷移列上での遷移が 1 方向分岐節点において存在するため，DA のみを用いて遷移するマシン X や Darts よりも高速になると考えられる．以上の点は，次節の実験的評価により検証する．

次に照合時に必要な記憶領域を評価する．AC マシンの照合時に必要な記憶領域は，Goto 遷移，Failure 遷移を構成するための領域と，Output 集合を管理するための領域の和とし，トライにおいては，遷移を定義するための領域と，キーのインデクスを管理するための領域との和とする．よって，提案マシン，マシン X，マシン Y における照合時の記憶領域は，1 節点あたりの記憶領域に節点数を乗算し，配列 Next のサイズを加えたものになる．

表 2 に各手法における 1 節点あたりの構成要素とバイト数をまとめる．提案手法，マシン X，MPDA における遷移種を格納するための領域は，‘#’ や ‘$’ などのキーの構成要素として含まれない値を表すために 2 バイトとした．

$d > 0$ より，マシン Y の 1 節点あたりの記憶領域は，提案マシンやマシン X よりもつねに大きい．さらに，1 バイトの遷移種を d ビットに分割して取り扱うと，1 バイトの遷移種

表 2　各手法における 1 節点あたりの構成要素
Table 2　The elements of one node of each machine.

		構成要素	使用領域	計
提案マシン	DA 上	Base 値	4B	
		Check 値	2B	6B
	SDA 上	SingleBase 値	4B	
		SingleCheck 値	2B	6B
マシン X		Base 値	4B	
		Check 値	2B	6B
マシン Y		遷移先管理配列	$2^d \times 4$B	
		Output 集合のインデクス	4B	$2^d \times 4 + 4$B
Darts		Base 値	4B	
		Check 値	4B	8B
MPDA	DA 上	Base 値	4B	
		Check 値	2B	6B
	配列 Tail 上	Tail 値	2B	2B

を取り扱うトライと比較して節点数が約 $8/d$ 倍に増加する[7]．一方，提案マシンやマシン X の節点数は終端節点や Failure 節点により増加するが，トライを構成する節点 1 つに対して，たかだか 1 つずつの付加になるため，トライと比較すると最大でも 3 倍程度である．$d \leq 2$ であるときは，提案マシン，マシン X の 1 節点あたりの記憶領域，節点数がともにマシン Y よりも小さく，$d > 2$ であるときは提案マシン，マシン X の 1 節点あたりの記憶領域がマシン Y の 1/3 以下となるため，提案マシン，マシン X の照合時の記憶領域はマシン Y よりもつねに小さいといえる．

提案マシンは，1 方向分岐節点に Failure 節点への遷移を作成しないことで，マシン X よりも記憶領域を抑制できるため，AC マシンにおいては提案マシンが最も記憶領域の小さい手法であるといえる．しかし，Failure 節点や終端節点の追加により，提案マシンの節点数はトライである Darts や MPDA の節点数よりも多い．そのため，提案マシンの記憶領域は，DA 上の 1 節点あたりの記憶領域が等しい MPDA より大きい．

提案手法の照合時間と記憶領域における有効性は 1 方向分岐の節点数に依存する．トライにおける節点数は，キー集合のキー数とキー長に依存し，多分岐の節点数は最大でもキー数を超えず，1 方向分岐の節点数はキー長が長いほど多い．よって，同程度のキー数を持つキー集合においては，キー長の長いキー集合ほど提案マシンは照合速度の高速化と記憶領域の抑制を図れる．

最後に構築時間について評価する．トライはキー集合を 1 度走査することで構築できるため，キー集合のサイズに対して線形時間で構築可能である．AC マシンはトライ構造を構築した後，Failure 遷移や Goto 遷移を追加するために各節点を幅優先探索する[1]が，この走査回数は最大でも節点数であるため，キー集合のサイズに対して線形時間で構築できる．

提案マシンはキー集合から構築した AC マシンを幅優先探索，深さ優先探索して DA 上と SDA 上に遷移を定義する．いずれの走査回数も節点数を上回らないが，DA 上に遷移を定義する場合は，DA 上を走査して遷移定義式を満たす Base 値を求める[15]必要があるため，通常の配列構造を用いて遷移を定義する手法[11]よりも時間を要する．DA 上で最後方の使用要素までに存在する未使用要素の数を e とし，ある節点 x における出次数を $|L_x|$ とすると，節点 x からの遷移を定義するために DA 上を走査する回数は最大で $e \times |L_x|$ 回である[15]．Failure 節点や終端節点の付加に加え，すべてのシングル節点を SDA 上に定義する提案マシンは，DA 上の節点における出次数の平均がトライよりも増加し，通常の DA よりも DA 上への遷移定義に時間を要するといえる．提案マシンの構築時間と Base 値を決定する際に DA 上を走査する回数との関連性は，次節で実験的に検証する．

4.2　実験的評価

提案マシンを評価するために，他手法との比較実験を Intel Pentium Dual 1.60 GHz, Fedora 7 上で行った．ただし，マシン Y は遷移種を 4 ビットに分割して登録した．実験では，IPA の電子辞書[16]などから抽出した日本語形態素 40 万語を母集団とし，4 万語から 40 万語まで 4 万語ごとに語数を増やしたキー集合を作成し，Web 上の記事から抽出した 20 MB のテキストを照合した．また，Web 上[17]から抽出した英単語 30 万語を母集団とし，3 万語から 30 万語まで 3 万語ごとに語数を増やしたキー集合を作成し，母集団の英単語を連結して 20 MB に調節したテキストを照合した．以下，日本語形態素 4 万語の集合を $J_{(40k)}$，40 万語の集合を $J_{(400k)}$，英単語 3 万語の集合を $E_{(30k)}$，30 万語の集合を $E_{(300k)}$ とする．

$J_{(40k)}$，$J_{(400k)}$，$E_{(30k)}$，$E_{(300k)}$ の詳細と，各キー集合を登録した提案マシンにおけるマルチ節点の数とシングル節点の数を**表 3** に示す．ただし，表 3 におけるマルチ節点数，シングル節点数は，終端節点，Failure 節点の数を含まない．それぞれの実験における照合時間を**図 11**，**図 12** に，照合時に必要な記憶領域を**図 13**，**図 14** に示す．図 13，図 14 には登録したキー集合のサイズ（キーを並べたテキストファイルのサイズ）もあわせて示す．

また，**表 4** に，キー集合 $J_{(40k)}$，$J_{(400k)}$，$E_{(30k)}$，$E_{(300k)}$ を登録したときの実験結果として，照合時の遷移回数，照合時間，トライ上節点数，遷移列の配列サイズ，記憶領域を示す．トライ上節点数は，提案マシン，マシン X，Darts，MPDA において DA のサイズを表す．遷移列配列サイズは，提案マシンにおいて SDA の配列サイズを表し，MPDA において配列 Tail の配列サイズを表す．遷移列走査回数は，提案マシンにおいて SDA 上を走査した回数を表し，MPDA において配列 Tail 上を走査した回数を表す．記憶領域は，キーや Output 集合のインデクスを管理する領域を含むサイズを表す．ただし，提案手法，マシン

表 3　キー集合の詳細
Table 3　The details of the key sets.

	$J_{(40k)}$	$J_{(400k)}$	$E_{(30k)}$	$E_{(300k)}$
登録語数	40.000	400.000	30.000	300.000
平均キー長 [byte]	11.56	11.17	9.45	9.46
最大キー長 [byte]	44	54	25	31
キー集合のサイズ [byte]	502,269	4,866,340	313,424	3,138,617
マルチ節点数	55,067 (22%)	471,304 (26%)	42,265 (29%)	356,206 (41%)
シングル節点数	199,738 (78%)	1,314,725 (74%)	104,036 (71%)	521,409 (59%)

() 内はマルチ節点数とシングル節点数の合計を基準としたときの割合

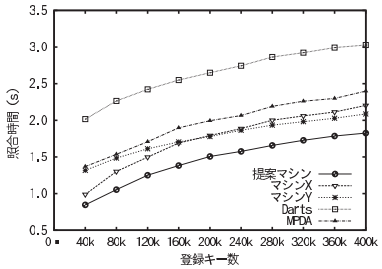

図 11　照合時間（日本語形態素）
Fig. 11　Matching time (Japanese words).

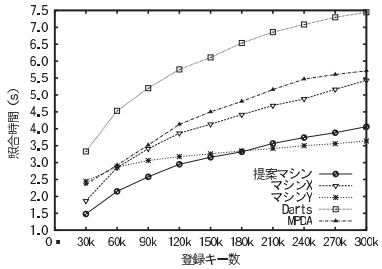

図 12　照合時間（英単語）
Fig. 12　Matching time (English words).

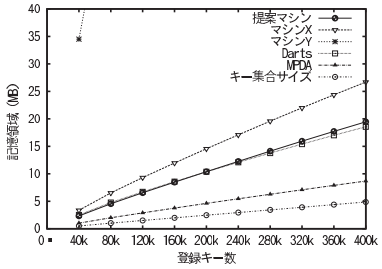

図 13　記憶領域（日本語形態素）
Fig. 13　Memory required (Japanese words).

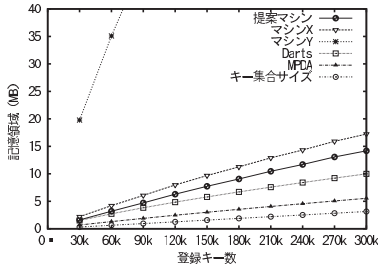

図 14　記憶領域（英単語）
Fig. 14　Memory required (English words).

X，マシン Y は配列 Next を用いて Output 集合を管理し，MPDA は，配列 Tail の 2 要素分を使ってキーのインデクスを管理する．

まず照合時間について評価する．図 11 に示すように，日本語形態素における実験では提案マシンが最も高速であり，表 4 に示すように，$J_{(400k)}$ 登録時の提案マシンの照合時間は，マシン X の 83%，マシン Y の 87%，Darts の 60%，MPDA の 76%であった．しかし，図 12 に示すように，英単語における実験では登録キー数が増加するとマシン Y のほうが高速になる．以下，登録キー数と照合時間の関係について述べ，配列構造であるマシン Y と DA を用いた各手法における照合速度の変化について考察する．その後，提案手法の遷移管理が照合速度の高速化に有効であることをマシン X との比較により示す．

図 11，図 12 に示すように各手法とも登録キー数の増加により照合時間が増加する．AC マシンである提案マシン，マシン X，マシン Y の照合速度が低下するのは，表 4 に示すように登録キー数の増加によりテキスト中から検出されるキー数が増加し，Output 関数を呼

表 4　日本語形態素，英単語における実験結果
Table 4　The results of the experiment for Japanese words and English words.

	提案マシン	マシン X	マシン Y	Darts	MPDA
キー集合 $J_{(40k)}$					
トライ上遷移回数	19,301,038	25,801,445	40,000,000	48,017,926	45,120,247
遷移列走査回数	5,721,623	-	-	-	2,897,679
総遷移回数	25,022,661	25,801,445	40,000,000	48,017,926	48,017,926
キー検出数	1,515,298	1,515,298	1,515,298	1,515,298	1,515,298
照合時間 [sec]	0.85	0.99	1.32	2.02	1.37
トライ上節点数	70,378	534,500	504,974	313,835	70,359
遷移列配列サイズ	292,055	-	-	-	300,673
記憶領域 [MB]	2.33 (4.66)	3.37 (6.74)	34.50 (69.00)	2.51 (5.02)	1.02 (2.04)
キー集合 $J_{(400k)}$					
トライ上遷移回数	18,716,587	25,447,303	40,000,000	54,837,526	51,104,889
遷移列走査回数	7,207,858	-	-	-	3,732,637
総遷移回数	25,924,445	25,447,303	40,000,000	54,837,526	54,837,526
キー検出数	8,640,915	8,640,915	8,640,915	8,640,915	8,640,915
照合時間 [sec]	1.82	2.20	2.09	3.02	2.39
トライ上節点数	632,289	3,530,195	2,319,422	4,181,274	748,134
遷移列配列サイズ	2,343,158	-	-	-	2,095,034
記憶領域 [MB]	19.45 (4.00)	26.69 (5.48)	241.65 (49.66)	18.56 (3.81)	8.68 (1.78)
キー集合 $E_{(30k)}$					
トライ上遷移回数	23,638,703	32,790,585	40,000,000	84,467,452	76,378,362
遷移列走査回数	9,080,528	-	-	-	8,089,090
総遷移回数	32,719,231	32,790,585	40,000,000	84,467,452	84,467,452
キー検出数	5,193,494	5,193,494	5,193,494	5,193,494	5,193,494
照合時間 [sec]	1.48	1.87	2.45	3.33	2.36
トライ上節点数	59,940	346,840	289,133	188,366	53,241
遷移列配列サイズ	187,462	-	-	-	179,819
記憶領域 [MB]	1.60 (5.16)	2.20 (7.10)	19.78 (63.81)	1.51 (4.87)	0.68 (2.19)
キー集合 $E_{(300k)}$					
トライ上遷移回数	18,883,830	28,722,014	40,000,000	104,766,252	95,068,856
遷移列走査回数	10,810,842	-	-	-	9,697,396
総遷移回数	29,694,672	28,722,014	40,000,000	104,766,252	104,766,252
キー検出数	50,385,538	50,385,538	50,385,538	50,385,538	50,385,538
照合時間 [sec]	4.06	5.43	3.64	7.44	5.71
トライ上節点数	586,318	2,670,916	1,729,496	1,248,905	561,792
遷移列配列サイズ	1,576,413	-	-	-	1,085,005
記憶領域 [MB]	14.18 (4.52)	17.23 (5.49)	118.80 (37.85)	9.99 (3.18)	5.54 (1.77)

() 内は登録したキー集合のサイズを基準としたときの割合

び出す回数が増加するためであり，トライである Darts，MPDA は表 4 に示すようにキー数の増加により遷移回数が増加するためである．また，辞書サイズの増加も原因の 1 つとして考えられる．キー数の増加により節点数が増加すると，遷移時にキャッシュのヒット率が低下する[9] ため，辞書サイズが小さいほど照合速度も高速になるといえる．

表 4 に示すように，提案マシンやマシン X の総遷移回数は，マシン Y よりも少ないが，$E_{(300k)}$ 登録時の実験ではマシン Y が最も高速であった．これは，DA 上の遷移よりも，配列構造であるマシン Y の遷移に要する演算量が少ないためと考えられる．しかし，$E_{(30k)}$ における実験では，提案マシン，マシン X，MPDA のほうが高速であった．これは登録するキー数が比較的少ないときに，DA を用いた手法の辞書サイズが特に小さくなり，キャッシュのヒット率が高かったためと考えられる．英単語における実験と同様に，日本語形態素においても，キー数がさらに増加するとマシン Y のほうが高速になると考えられる．

提案マシンの総遷移回数は，トライ法である Darts や MPDA よりも少ないが，$J_{(400k)}$，$E_{(300k)}$ 登録時の総遷移回数がマシン X よりも多い．これは，SDA 上から DA 上への遷移する際に '$' 節点へ Failure 遷移するためである．しかし，提案マシンはマシン X と比較して $J_{(400k)}$ 登録時で 17%，$E_{(300k)}$ 登録時で 25%照合時間を短縮した．これは SDA 上の遷移が DA 上の遷移よりも遷移計算量が少ないためであり，提案手法における分岐別の遷移管理が照合速度の高速化に有効であることを示す．

次に，照合時の記憶領域について評価する．図 13，図 14 に示すように，配列構造で遷移を定義するマシン Y の記憶領域は，DA を用いた提案マシン，マシン X，Darts，MPDA と比較すると遙かに大きく，マシン Y は大規模なキー集合を登録するには不向きであるといえる．

提案マシンはマシン X と比較すると，表 4 に示すように，$J_{(400k)}$ 登録時で 27%，$E_{(300k)}$ 登録時で 18%記憶領域を削減した．これは，キー集合により構成されるトライの 1 方向分岐を遷移列として扱い，Failure 遷移を定義するための節点を抑制したことで，表 4 に示すように，トライ上の節点数と遷移列配列サイズの和がマシン X の節点数よりも小さくなったためである．$J_{(400k)}$ 登録時と比較して，$E_{(300k)}$ 登録時の抑制効果が低いのは，表 3 に示すように，英単語は日本語形態素よりも平均キー長が短く，シングル節点の割合が少ないためである．

照合時に必要な記憶領域が最も小さかった MPDA と比較すると，提案マシンの記憶領域は表 4 に示すように，$J_{(400k)}$ 登録時で 2.24 倍，$E_{(300k)}$ 登録時で 2.56 倍となったが，図 13 に示すように，日本語形態素の実験において Darts と同程度であった．この結果から Darts を形態素辞書として採用しているシステム[5),6)] に提案マシンを応用できるといえる．

表 5　構築実験結果
Table 5　The results of the building experiment.

		$J_{(40k)}$	$J_{(400k)}$	$E_{(30k)}$	$E_{(300k)}$
提案マシン	平均出次数	4.13（+2.97）	4.38（+3.16）	4.26（+3.05）	4.66（+3.32）
	平均走査回数	192.78	1321.01	279.96	3659.47
	構築時間 [sec]	0.83	52.73	0.77	77.33
マシン Y	構築時間 [sec]	0.29	2.53	0.17	1.36
Darts	平均出次数	1.16	1.22	1.21	1.34
	平均走査回数	4.06	3.80	3.72	3.23
	構築時間 [sec]	0.63	4.85	0.36	2.62

() 内は Darts における平均出次数との差

最後にマシン Y，Darts を比較手法とし，提案マシンの構築時間について評価する．**表 5** に，キー集合 $J_{(40k)}$，$J_{(400k)}$，$E_{(30k)}$，$E_{(300k)}$ を登録したときの各手法における構築時間と，提案マシン，Darts における DA 上の平均出次数，DA 上に遷移を定義する際に Base 値を求めるために DA 上を走査した回数の平均を示す．

表 5 に示すように，提案マシン，Darts はマシン Y よりも構築に時間を要した．これは，提案マシンと Darts が DA 上に遷移を定義する際，Base 値を取得する処理に時間を要するためである．今回の実験において，DA 上の節点における出次数がほぼ 1 であった Darts に対し，Failure 節点や終端節点の付加に加え，すべてのシングル節点を SDA 上に定義する提案マシンは DA 上の出次数が多い．これにより，Base 値を求める際に DA 上を走査する回数が増加し，提案手法は Darts よりも DA 上への遷移定義に時間を要した．

以上，提案手法は多手法と比較して構築に時間を要するものの，実用的な記憶領域で照合速度を高速化できることを示した．

5. おわりに

本論文では，ダブル配列を用いた AC マシンにおいて，遷移先の候補が一意に決定できる 1 方向分岐節点における遷移先を，ダブル配列と異なる配列に定義することで照合速度の高速化と記憶領域の抑制を図る手法を提案した．日本語形態素を登録した実験の結果，提案手法は辞書構造として実用的な記憶領域で，他手法の 60%〜87%の時間で対象データを照合した．提案手法は形態素辞書として利用することが考えられる．

今後の課題として，提案手法を利用した自然言語処理システムを開発して評価することや，効率的な動的構成法を与えることで提案マシンの応用範囲を広げることがあげられる．

参 考 文 献

1) Aho, A.V. and Corasick, M.J.: Efficient String Matching an Aid to Bibliographic Search, *Comm. ACM*, Vol.18, No.6, pp.333–340 (1975).
2) Maruyama, H.: Backtracking-Free Dictionary Access Method for Japanese Morphological Analysis, *Proc. 15th Int. Conf. on Computational Linguistics*, pp.208–213 (1994).
3) 森　信介：DFA による形態素解析の高速化，電子情報通信学会技術研究報告，NLC，言語理解とコミュニケーション，Vol.96, No.158, pp.17–23 (1996).
4) Ando, K., Fuketa, M., Shishibori, M. and Aoe, J.: Dictionary Structure for Morphological Analysis of Oriental Languages, *Proc. 18th Int'l Conf. on Computer Processing of Oriental Languages*, pp.533–538 (1999).
5) ChaSen. http://chasen-legacy.sourceforge.jp/
6) MeCab. http://mecab.sourceforge.net/
7) 有川節夫，篠原　武：文字列パターン照合アルゴリズム，コンピュータソフトウェア，Vol.4, No.2, pp.98–119 (1987).
8) Aoe, J.: An Efficient Digital Search Algorithm by Using a Double-Array Structure, *IEEE Trans. Softw. Eng.*, Vol.15, No.9, pp.1066–1077 (1989).
9) 信種真人，森田和宏，泓田正雄，青江順一：ダブル配列を用いたマシン AC の効率的格納方法，言語処理学会第 13 回年次大会，pp.831–834 (2007).
10) Darts. http://chasen.org/taku/software/darts/
11) 青江順一：トライとその応用，情報処理，Vol.34, No.2, pp.244–251 (1993).
12) Yata, S., Oono, M., Morita, K., Fuketa, M., Sumitomo, T. and Aoe, J.: A compact static double-array keeping character codes, *Information Processing and Management*, Vol.43, No.1, pp.237–247 (2007).
13) 有川節夫，篠原　武，宮原哲浩，武谷峻一，宮野　悟，竹田正幸，大島一彦，白石修二，酒井　浩，山本章博：テキストデータベース管理システム SIGMA とその利用，情報処理学会研究報告，Vol.1989, No.1989-FI-014, pp.1–8 (1989).
14) 蔵満琢麻，松浦寛生，望月久稔：ダブル配列を用いた AC 法の効率的なパターン照合，情報処理学会論文誌：データベース，Vol.1, No.2, pp.1–14 (2008).
15) 中村康正，望月久稔：圧縮ディジタル探索木における辞書情報更新の高速化手法，情報処理学会論文誌：データベース，Vol.47, No.SIG13(TOD31), pp.16–27 (2006).
16) 情報処理振興事業協会技術センター：IPA 日本語辞書.
17) Atkinson, K.: Ispell English Word Lists. http://wordlist.sourceforge.net/

(平成 20 年 12 月 20 日受付)
(平成 21 年　4 月 13 日採録)

(担当編集委員　池田 哲夫)

蔵満　琢麻（学生会員）

昭和 61 年生．平成 20 年大阪教育大学教育学部教養学科情報科学専攻卒業．現在，同大学大学院修士課程在学中．情報検索，パターン照合，自然言語処理に関する研究に従事．電子情報通信学会，日本データベース学会各学生会員．

望月　久稔（正会員）

昭和 44 年生．平成 5 年徳島大学工学部知能情報工学科卒業．平成 7 年同大学大学院博士前期課程修了．平成 10 年同大学院博士後期課程修了．博士（工学）．同年大阪府立工業高等専門学校電子情報工学科講師．平成 15 年大阪教育大学教育学部教養学科情報科学講座講師．平成 19 年大阪教育大学教育学部教養学科情報科学講座准教授．現在に至る．情報検索，自然言語処理，知識表現の研究に従事．電子情報通信学会，人工知能学会，自然言語処理学会各会員．