拡張ハッシュ法における部分文字列検索の設計と実現

望月久稔† 森田和宏†
獅々堀正幹† 青江順一†

ハッシュ関数とファイル構造を局所的に再構成し，あふれを解消する拡張ハッシュ法は，ハッシュ法の検索の高速性を維持し，キー総数が予想できない分野にも応用できるが，任意の文字列を部分文字列として含むキーの検索を効率的に行うことはできない．本論文では，拡張ハッシュ法でこの部分文字列検索を実現するために，まず，特徴ベクトルと呼ばれるビット列をハッシュ値として用いて，トライを構成する．次に，アクセスすべきバケットを決定するための疑似ベクトルを定義し，トライ上で疑似ベクトルに対応した枝のみを走査する限定深さ優先探索法を提案する．さらに，キー数の増加に対応したトライをコンパクトに構成するために，均整ベクトルを導入し，そのベクトルを増進的に生成する手法を提案する．これにより，キーの取扱いに必要なビット数をおさえつつ，限定深さ優先探索法をより効率的に行うことができる．また，探索後に参照されるバケット数を抑制するためにディスクリプタを利用する．本手法は，約10万語の日本語キー集合と約8万語の英語キー集合に対する実験結果より，ディスクリプタだけを利用した拡張ハッシュ法に比べ，読み込みバケット数が60～90%減少し，検索時間が2～10倍高速となることが分かった．

The Design and Implementation of a Substring Search in Extendible Hashing

HISATOSHI MOCHIZUKI,† KAZUHIRO MORITA,† MASAMI SHISHIBORI† and JUN-ICHI AOE†

An extendible hashing scheme resolves bucket overflows by reorganizing the hash function and file structure locally, so it is very suitable for fast key retrievals of dynamic key sets. However, it can't search keys that contain a given string as substrings efficiently. In this paper, in order to design this substring search in extendible hashing, signature vectors are introduced as hash values, and a trie structure as an extendible hash table, where each vector is composed by a bit stream. Pseudo signature vectors are defined to identify the buckets, and a constrained depth-first search is presented to traverse the arcs of the trie structure. To construct a compact trie despite of an increase in the number of keys, uniform signature vectors are introduced, and the method for an incremental expansion of the hash table is proposed. This approach can restrict the size of the bit stream for each key, making constrained depth-first search efficient. From simulation results by applying the presented schemes to Japanese and English key sets, it was shown that the number of accessed buckets decreased from 60 to 90% in comparison with the traditional extendible hashing for which only descriptors were used. In addition, the searching time cost of the presented approach is 2 to 10 times faster.

1. まえがき

キー検索法の一手法であるハッシュ（hash）法は，衝突（collision）・あふれ（overflow）が生じない場合には非常に高速な手法である．しかし，ハッシュ表の大きさが変化しない静的（static）ハッシュ法[1),2)]をキーの総数が予測できない分野に応用する場合，ハッシュ表を大きくとりすぎると記憶領域が無駄になり，逆に小さく見積もると衝突が頻発して検索効率が低下する．これに対して，拡張（extendible）ハッシュ法[1),3)]は，表の大きさを動的（dynamic）に伸縮できるので，キーの総数が予測できない分野に適し，かつ検索の高速性も合わせ持っている．

ハッシュ法は，キーをハッシュ表の番地に分散して格納するので，2進木やB木等の探索木法[4)]の特徴であるキーの順検索（order search）や，文字列照合法[5)]の特徴である部分文字列検索（substring search）には適さない．これに対して，獅々堀ら[6)]は拡張ハッシュ法に対する順検索法を近年提案しているが，部分文字

† 徳島大学工学部
Faculty of Engineering, Tokushima University

列検索の導入はいまだ実現されていない．

Burkhard[7]，Rivest[8]らは，一定長のキー x を対象とし，x の特定位置の文字を検索する手法を提案しているが，任意長のキー x を部分文字列として含むキー y の検索（本論文における部分文字列検索）を対象としていない．この部分文字列検索が実現できれば，キー検索法の応用範囲はさらに広いものとなる．

本論文は，拡張ハッシュ法における部分文字列検索を効率的に実現する手法を提案する．

以下，2章で拡張ハッシュ法の概要について述べる．3章では，部分文字列検索を実現するために，特徴（signature）ベクトル[9),10)]に基づく疑似ベクトルを定義する．4章で，これらのベクトルを用いた限定深さ優先探索によるキーの部分文字列検索アルゴリズムを提案する．また，この探索を効率的に行うために均整ベクトルを定義し，ハッシュ表の構成法を提案する．さらに，バケット検索の高速化法を導入する．5章では，提案した部分文字列検索法を理論的，具体的に評価する．6章では，本論文の総括と今後の課題について触れる．

2. 拡張ハッシュ法

拡張ハッシュ法は，値域の大きさを動的に伸縮できるハッシュ関数が必要であり，キー x を長さ m のビット列に写像する関数 $\mathrm{H}(x)$ を使用する[1)]．$\mathrm{H}(x)$ の末尾 L ビットにより次のハッシュ関数 $\mathrm{h}_L(x)$ を定義する．

$$\mathrm{H}(x) = b_{m-1}\cdots b_L\cdots b_2b_1b_0 \qquad (1)$$

（b_L $(0 \le L \le m-1)$ は，0または1のビット）

$$\mathrm{h}_L(x) = b_{L-1}\cdots b_2b_1b_0 \qquad (2)$$

（例1） 日本の15の地名をキー集合 K とし，キー x を構成する各文字の内部コード値の総和を $\mathrm{w}(x)$ で表すとき，$\mathrm{w}(x)$ の末尾8ビットより決定された関数 $\mathrm{H}(x)$ を**表1**に示す．ただし，各文字'a'，'b'，'c'，…，'z'の内部コード値をそれぞれ1，2，3，…，26とする．

$\mathrm{H}(x)$ の末尾1ビットに対応するハッシュ関数 $\mathrm{h}_1(x)$ により，キー "tokushima"，"osaka"，"tokyo" を格納したハッシュ表を**図1**(a)に示す．この例では，バケット（bucket）に格納できるキーの最大数は2とする．ハッシュ表の上部に示す値1は，ハッシュ表全体の番地計算に必要なビット数を表す全域深さ（global depth）であり，各バケットの上部に示す値1は，そのバケットを他のバケットと区別するために必要な番地計算のビット数を表す局所深さ（local depth）である．なお，バケット内のキー x には，理解を助けるために $\mathrm{H}(x)$ を付記してある．

次に，h_1("okayama") = 1 なるキー "okayama" を追加すると，バケット2があふれるので，2ビット分のハッシュ関数 $\mathrm{h}_2(x)$ を採用して，ハッシュ表の大きさを2倍に成長させる（図1(b)）．図1(b)では，番地01と11は唯一のバケットに対応するが，バケット2には複数の番地00と10が対応する．これは，バケット2が将来キーの追加で領域があふれたとき，番地00と10にバケットを分割できる余裕があることを意味する．このことは，バケット2の局所深さ1が全域深さ2より小さいことからも分かる．（例終）

ハッシュ表が大きくなるにつれて各バケットの局所深さは不均一になり，ハッシュ表の多くの番地が特定の（局所深さの浅い）バケットに集中するので，ハッシュ表が冗長になる．したがって，ハッシュ表のデータ構造として，ビット列を2進木（binary tree）構造で表現する場合が多い[11)~13)]．この構造は対象文字をビットに限定したトライ（trie）[3)]であり，本論文でもこのトライで議論を進める．

表1 ハッシュ値の例
Table 1 An example of hash values.

x	$\mathrm{w}(x)$	$\mathrm{H}(x)$	x	$\mathrm{w}(x)$	$\mathrm{H}(x)$
akita	42	00101010	tokushima	117	01110101
okayama	67	01000011	nagasaki	63	00111111
okinawa	74	01001010	nara	34	00100010
osaka	47	00101111	niigata	61	00111101
kagawa	44	00101100	hiroshima	100	01100100
kumamoto	109	01101101	fukui	68	01000100
kochi	46	00101110	mie	27	00011011
tokyo	86	01010110			

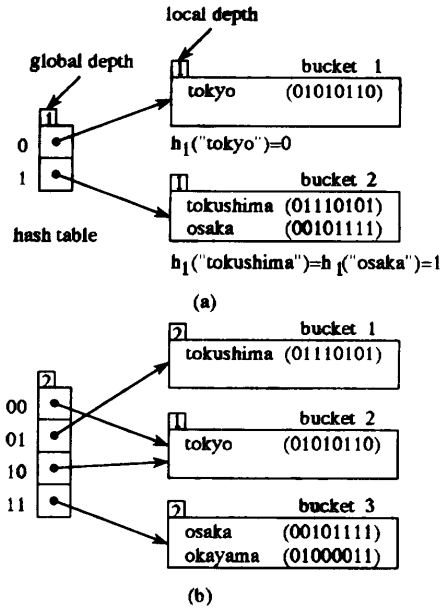

図1 拡張ハッシュ表の例
Fig. 1 An example of extendible hash tables.

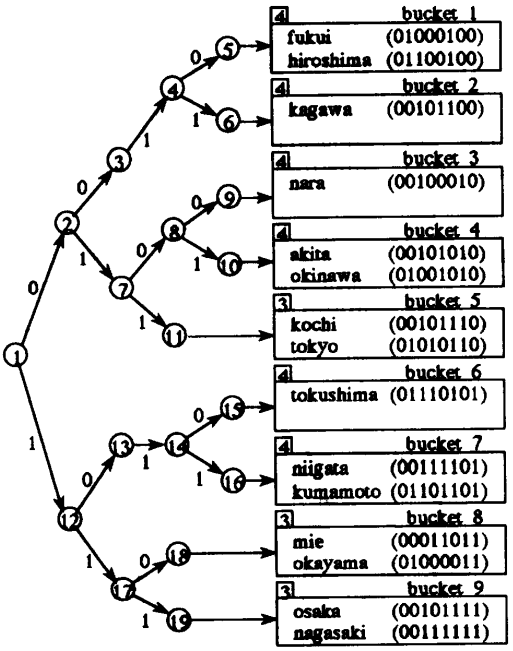

図2 トライ表現による拡張ハッシュ表の例
Fig. 2 An example of the extendible hash table constructed with a trie.

(例2) 表1のキーで構成したトライを**図2**に示す.

全域深さ4に対して,局所深さ3のバケット5には,$2^{4-3}=2^1=2$ より,図1ならば2個の番地が対応するが,トライ構造ではノード11をバケット5に連結するだけでよい.

h_4("okayama") = 0011 なるキー "okayama" の検索は,まずトライのルートノード1よりスタートし,最後尾の3ビット110によるパス(ノード1, 12, 17, 18)を経由して,トライの葉であるノード18までたどり,バケット8から検索できる.(例終)

3. 部分文字列検索への準備

3.1 特徴ベクトル

ハッシュ関数 $h_L(x)$ はキー x の部分文字列の情報をまったく持たないので,部分文字列検索を実現するためには,各バケットのキーを総あたりで調べるしかない.したがって,本手法での重要な提案は,$h_L(x)$ に対して,x に関する部分文字列の情報を反映することである.そこで,文字列照合で利用されている特徴ベクトル[9),10)]を $h_L(x)$ に導入して,x を特徴ベクトルに写像する特徴関数を次に定義する.

[特徴関数 Sign(x)]

f(ab) は2文字 ab を0から $m-1$ までの整数に写像する関数とするとき,文字列 x に対する特徴ベクトル $b_{m-1}\cdots b_i\cdots b_2b_1b_0$ ($0\le i<m$, b_i はビット値)を返す特徴関数 Sign(x) は,次の手順で定義できる.

(1) すべてのビット b_i を0にする.
(2) x のすべての隣接する2文字 ab に対し,$i=$ f(ab) なる i を計算し,$b_i=1$ とする.

また,特徴関数 Sign に対して f(ab) を明示する必要がある場合には,Sign:f(ab) と表記する.(関数終)

(例3) 剰余演算を mod で表し,特徴関数 Sign を

$$\text{Sign} : (a+2b) \bmod 8 \tag{3}$$

で定義するとき,キー "shima" の特徴ベクトル Sign("shima") は次のように決定される.

f("sh") = $(19+2\times 8) \bmod 8 = 3$
f("hi") = 2, f("im") = 3, f("ma") = 7
Sign("shima") = 10001100 (4)

また,例1のキー集合 K に対する特徴ベクトルを**表2**に,表2に対応するトライを**図3**に示す.(例終)

表2 特徴ベクトルの例
Table 2 An example of signature vectors.

x	Sign(x)	x	Sign(x)
akita	11100010	tokushima	10101100
okayama	10101000	nagasaki	10100011
okinawa	10100011	nara	00110001
osaka	10100000	niigata	11001011
kagawa	10100010	hiroshima	10101101
kumamoto	10101100	fukui	10101001
kochi	00101110	mie	10001000
tokyo	10100100		

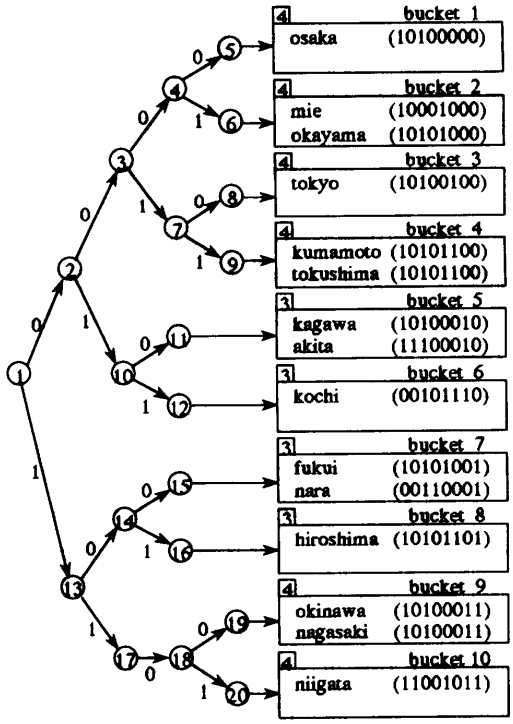

図3 特徴ベクトルによる拡張ハッシュ表の例
Fig. 3 An example of the extendible hash table constructed by the trie with signature vectors.

3.2 特徴ベクトルの比較

特徴ベクトルを使用することで，文字列 x，y 自身を比較することなしに部分文字列の判定を行うことができる[4),5),10)]．したがって，この判定関数 Comp_Vec を次に定義する．

[関数 Comp_Vec(Sign(x), Sign1(y))]

Sign(y) のビットを反転したベクトルと Sign(x) との論理積が，ゼロベクトルならば TRUE を返し，そうでなければ FALSE を返す．(関数終)

この関数は，Sign(x) と Sign(y) の論理積が Sign(x) と同じベクトルかどうかを判定することを意味している．この判定により，論理積が Sign(x) と同じベクトルならば，文字列 y の特徴ベクトルは，文字列 x の文字列情報である特徴ベクトルのビットパターンを包含☆するので，y は x を部分文字列として含む可能性がある．

したがって，この関数が FALSE を返す場合，x は y の部分文字列でないことが保証されるので，x，y の直接比較は不要である．しかし，TRUE を返す場合，この保証は得られないので，直接比較を行う必要がある．この比較で x が y の部分文字列でない場合をフォルスドロップ (false drop) と呼び[9)]，フォルスドロップが少ないほど，関数 Comp_Vec による前処理は有効となる．

(例 4) 関数 Comp_Vec を説明するために，例 3 より，"shima" の部分文字列検索をキー "kochi"，"kumamoto"，"hiroshima" に対して行う．

"shima" の特徴ベクトル vec_{sh} = Sign("shima") = 10001100 と "kochi" の特徴ベクトル vec_{ko} = Sign("kochi") = 00101110 に対して，

$$\text{Comp\_Vec}(vec_{sh},\ vec_{ko}) = \text{FALSE}$$

となるので，"kochi" は "shima" を部分文字列として含まないことが分かる．一方，vec_{ku} = Sign("kumamoto") = 10101100 と vec_{hi} = Sign("hiroshima") = 10101101 に対して，

$$\text{Comp\_Vec}(vec_{sh},\ vec_{ku}) = \text{TRUE} \qquad (5)$$
$$\text{Comp\_Vec}(vec_{sh},\ vec_{hi}) = \text{TRUE}$$

となるが，"shima" は "hiroshima" にしか含まれないので，式 (5) による Comp_Vec の判定は，フォルスドロップを生じたことになる．(例終)

3.3 疑似ベクトル

3.2 節より，文字列 x を部分文字列として含むキーを検索するためには，x の特徴ベクトル Sign(x) のビットパターンを包含するベクトルのすべてを調べればよい．そこで，本節ではこれらのベクトルを疑似 (pseudo) ベクトルとして導入する．

文字列 x の特徴ベクトル vec に対する疑似ベクトルの集合 S(vec) を，次式で定義する．

$$\text{S}(vec) = \{vec' | \text{Comp\_Vec}(vec, vec') = \text{TRUE}\}$$

疑似ベクトル vec' は，vec のビット 0 を 0 または 1 に置き換えることにより生成することができる．また，vec 自身も S(vec) に含まれる．

(例 5) 特徴ベクトル 0110 に対する疑似ベクトルの集合 S(0110) は，次のようになる．

$$S(0110) = \{0110, 0111, 1110, 1111\}$$ (例終)

キー y が文字列 x を部分文字列として含むとき，y の特徴ベクトルは S(vec) に含まれるので，x の部分文字列検索では，トライ上で vec の疑似ベクトル vec' のパスのみを探索することで，アクセスすべきバケットを絞り込むことが可能となる．しかし，疑似ベクトル vec' は一般に複数存在しているので，類似したビットパターンを持つ疑似ベクトルをそれぞれトライの根からたどることは，同じパスを何度もたどることになり効率的でない．また，バケットの局所深さがハッシュ表の全域深さよりも小さい場合には，1 つのバケットが複数の疑似ベクトルに対応することがあり，同じバケット番号に対応する複数の疑似ベクトルの走査を別々に行うことも冗長である．

(例 6) 図 3 において，例 5 で用いた疑似ベクトルのパスを探索する場合，疑似ベクトル 0110 と 0111 のパスはともに同じノード 1，2，10，12 をたどる．また，これらのパスによるバケットは，局所深さがハッシュ表の全域深さよりも小さく，同じバケット番号 6 に対応している．よって，このように疑似ベクトルの走査を別々に行うことは冗長である．(例終)

3.4 限定深さ優先探索法

3.3 節で述べた冗長な走査を避けるために，枝が 2 本出ているノードはビット 0 の枝を優先してたどる深さ優先探索法 (depth first search) を導入する．この探索法により，vec 以外の疑似ベクトルに対応する葉は，vec に対応する葉よりも必ずあとに到達する．

しかし，深さ優先探索法は vec の疑似ベクトルに相当しないビットパターンをも走査するので，さらに走査の制約条件を以下に提案する．

(i) vec に対応する葉より走査を開始し，その後の走査順序は深さ優先探索法に従う．

(ii) 特徴ベクトル vec のビット $b_i = 1$ なる i に対し，トライ上の $b_i = 0$ に相当する枝は走査しない．

☆ Sign(x) の i 番目のビットが 1 ならば，Sign(y) の i 番目のビットも 1 であることを意味する．

この走査法を限定深さ優先探索法（constrained depth first search）と呼ぶ．この限定深さ優先探索法により，疑似ベクトルに対応するバケットだけが決定される．

（例7）図3において，例5で用いた疑似ベクトルのパスを探索する場合，まず特徴ベクトル0110に対応するパスを走査する．これは，特徴ベクトル自身が，疑似ベクトルの集合 $S(0110)$ に含まれていて，かつ，ビット0によるものを優先してたどる深さ優先探索法の場合に最初に走査するパスだからである．次に，疑似ベクトル0111に対応するパスを走査するが，このベクトルは先に探索した特徴ベクトルと同じバケット6へとリンクされているので，走査する必要はない（例6）．また，限定深さ優先探索法の制約条件より，ノード13からビット0によるノード14への枝を走査する必要はなく，残りの疑似ベクトル1110，1111に対応するパスを走査するが，それぞれに対応するパスが，ノード17以降より存在しないので，探索は終了する．（例終）

4. 部分文字列検索の実現

4.1 アルゴリズム

部分文字列検索アルゴリズムを関数Search_SubStrとして与えるが，まず，この関数で使用する変数について述べ，また，この関数で使用する関数Advance，Replaceを先に準備する．

4.1.1 準　　備

限定深さ優先探索法では，走査したノードをスタックに記憶し，以下の大域変数を用いる．

- 現在処理中のノード番号を *node* で表す．
- 疑似ベクトルの現在処理中のビット位置を i で表す．i はスタックに格納されたノード数にも対応する．
- 文字列 x の特徴ベクトルを $vec = \mathrm{Sign}(x) = b_{m-1}\cdots b_i\cdots b_1 b_0$ （$0 \leq i < m$，b_i はビット値）で表す．

4.1.2 遷移アルゴリズム

トライの葉までをベクトルに従い走査するアルゴリズムを関数Advanceとして与える．この遷移アルゴリズムは，走査されたノードを保持するためにスタックを用いる．

［関数 Advance()］

node を起点として，ビット b_i からトライを走査し，葉に到達すればTRUEを返し，到達しなければFALSEを返す．

手順 (A-1)：{ ノードの判別 }

node がトライの葉であればTRUEを返し，葉でなければ手順 (A-2) へ．

手順 (A-2)：{ トライ上の特徴ベクトルによる遷移 }

node から b_i による枝が存在しなければFALSEを返し，存在すれば手順 (A-3) へ．

手順 (A-3)：{ ノードの設定と格納 }

node をスタックにプッシュし，*node* から b_i により遷移したノードを新たな *node* とする．ビット位置 i をインクリメントする．手順 (A-1) に戻る．（関数終）

手順 (A-2) で，深さ優先探索法により走査するパスから，特徴ベクトルに対応したパスに限定している．

（例8）例3の図3で $vec = \mathrm{Sign}(\text{“shima”})$ に対する関数 Advance を考える．ここで，*node* は1，i は0と仮定し，vec のビット列は $b_0 = 0$，$b_1 = 0$，$b_2 = 1$，$b_3 = 1$，$\cdots$，$b_7 = 1$ となる．

手順 (A-1) で *node* は1であり，葉ではないので，手順 (A-2) でノード1からビット $b_0 = 0$ による枝をたどる．手順 (A-3) でノード1をスタックに格納後，ノード2を新たな *node* とし，i を1にインクリメントする．以下同様にノード2，3，7，9をたどり，9が葉であるのでTRUEを返す．このとき，スタックにはノード1，2，3，7が積まれ，ビット位置 i は4となる．（例終）

4.1.3 疑似ベクトルパスの探索アルゴリズム

文字列 x の疑似ベクトル集合 S(vec) の要素を調べるアルゴリズムを関数Replaceとして与える．この関数は，関数 Advance で保持されたスタックのノードを利用して，*node* からトライの根に向かって遷移をバックしながら次の疑似ベクトルを探す．

［関数 Replace()］

node からトライの根に向かって遷移をバックしながらビット0の枝を探して，ビット1の枝への置換を試み（疑似ベクトルパスの探索），この置換が成功すればTRUEを返し，そうでなければFALSEを返す．

手順 (R-1)：{ スタック情報の確認 }

スタックが空ならばFALSEを返し，空でなければ，スタックのトップノードから *node* への枝であるビット値 c を決定後，トップノードをポップして *node* にセットする．ビット位置 i をデクリメントする．

手順 (R-2)：{ 疑似ベクトル候補の判別 }

c がビット1ならば，ビット0による枝を探すために手順 (R-1) へ戻る．c がビット0ならば手順 (R-3) へ．

手順 (R-3)：{ 疑似ベクトルの存在を判別 }

node からビット1による枝が存在しなければ，手順 (R-1) へ戻り，存在すれば手順 (R-4) へ．

手順 (R-4)：{ ビット 1 の枝に対する処理 }

node をスタックにプッシュし，*node* からビット 1 で遷移するノードを新たな *node* とする．i をインクリメントし，TRUE を返す．(関数終)

手順 (R-3) におけるビット 1 の遷移は，*vec* の疑似ベクトルによる遷移となる．つまり，3.4 節で述べた限定深さ優先探索法を実現するうえで，手順 (R-2)，(R-3) が遷移条件となり，走査するパスを制約している．

(例 9) 例 8 で関数 Advance が TRUE を返した状態からの関数 Replace の実行を考える．

手順 (R-1) で，スタックのトップノード 7 から 9 への枝がビット 1 なので，c はビット 1 となり，スタックのポップより *node* は 7，i は 3 となる．手順 (R-2) で，c がビット 1 より手順 (R-1) に戻る．同様にして，*node* は 3，2 へバックし，手順 (R-4) で，ノード 2 からのビット 1 よりノード 10 へ進み，TRUE を返す．このときのスタックにはノード 1，2 が積まれ，ビット位置 i は 2 となるので，疑似ベクトルは 10001110 となる．(例終)

4.1.4 部分文字列検索アルゴリズム

部分文字列検索アルゴリズムを関数 Search_SubStr として与える．この関数は，関数 Replace により設定される S(*vec*) の各疑似ベクトルに対応したパスを，関数 Advance により走査して，部分文字列 x を含むキーを検索する．ただし，トライの葉が指すバケット *bucket* から部分文字列 x を含むキーを探し，条件に該当するキーの集合を返す関数 Search_Bucket(x, *bucket*) を利用する．

[関数 Search_SubStr(x)]

入力キー x を部分文字列として含むキーの集合 $SUB\_SET(x)$ を出力する．

手順 (S-1)：{ 初期設定と特徴ベクトルの決定 }

node を 1，i を 0 に設定し，x に対する特徴ベクトル *vec* を関数 Sign(x) により決定する．また，$SUB\_SET(x)$ を空集合に設定する．

手順 (S-2)：{ トライの走査とバケット検索 }

関数 Advance により，*node* を起点として，ビット位置 i からビット $b_i, b_{i+1}, \cdots$ の枝を走査し，バケット番号 *bucket* に対応する葉に到達 (関数 Advance が TRUE となる) すれば，関数 Search_Bucket(x, *bucket*) が返したキーの集合と $SUB\_SET(x)$ との和集合を新たな $SUB\_SET(x)$ とする．

手順 (S-3)：{ 疑似ベクトルの設定 }

関数 Replace が TRUE となり，次に走査する疑似ベクトルが存在すれば手順 (S-2) に戻り，存在しなければ $SUB\_SET(x)$ を出力して終了する．(関数終)

関数 Search_SubStr は，最初に走査する疑似ベクトルを手順 (S-1) でキー x の特徴ベクトルに設定し，以降の疑似ベクトルを手順 (S-3) で設定する．この疑似ベクトルのパスを手順 (S-2) で走査することにより，限定深さ優先探索法を実現する．

(例 10) 図 3 と例 8，例 9 を参考にして，関数 Search_SubStr("shima") の実行を考える．

手順 (S-1) における最初の疑似ベクトル

vec = Sign("shima") = 10001100

$b_0 = 0,\ b_1 = 0,\ b_2 = 1,\ b_3 = 1,\ \cdots,\ b_7 = 1$

に対して，手順 (S-2) で関数 Advance により葉であるノード 9 に到達するので，関数 Search_Bucket ("shima", 4) より，SUB_SET("shima") = {"tokushima"} が得られる．手順 (S-3) の関数 Replace により，疑似ベクトル 10001110 が存在するので，手順 (S-2) に戻り走査を続行する．以後，処理を繰り返すことにより，SUB_SET("shima") = {"tokushima", "hiroshima"} が得られる．

この検索で，疑似ベクトル 10001110 の走査によるバケット 6 には，"shima" を部分文字列として含むキーはなく，フォルスドロップが生じている．(例終)

4.1.5 アルゴリズムの正当性

提案アルゴリズムの正当性は，関数 Search_SubStr が，3.3 節で定義した疑似ベクトルをすべて探索していることを証明することで得られる (付録参照)．

4.2 均整ベクトルを用いたトライの構成

特徴ベクトルにおけるビット 0，1 の生起確率がともに等しく 0.5 であるとき，1 ビットあたりの平均情報量を表すエントロピーは最大となる[14]．したがって，ビット 0 と 1 の割合が近いほど，トライの構成に必要なビット数が最も少なくなる．このような特徴ベクトルを均整 (uniform) ベクトルと呼ぶ．

キー数の増加とともに，バケットあふれを解消するために必要な特徴ベクトルのビット長も長くなるが，関数 f で隣接 2 文字に対して定義するビット 1 の数はキーの長さを越えることはないので，ベクトル中のビット 0 の割合が高くなる．その結果，トライのビット長が増大し，検索効率，記憶効率が低下する．本手法では，均整ベクトルの特徴を維持しながらバケットのあふれを解消するために，特徴ベクトルを状況に合わせて自由に伸縮する方法を提案する．

キー x に対して，特徴関数 Sign_1 により生成した均整ベクトル vec_1 ($= \mathrm{Sign}_1(x) = b_{m_1-1}\cdots b_1 b_0$) のビット長 m_1 が不足するとき，Sign_1 とは独立なビット長 m_2 の均整ベクトルを得る特徴関数 Sign_2 を用い

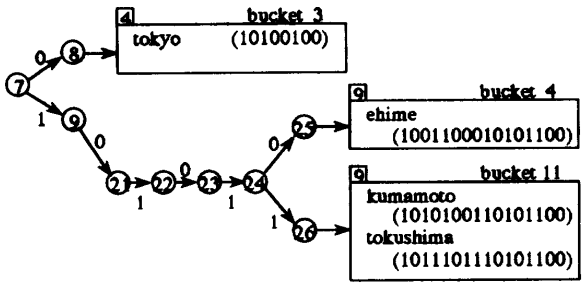

図4 均整ベクトルによるトライの例
Fig. 4 An example of the trie with uniform signature vectors.

て，均整ベクトル vec_2 $(= \mathrm{Sign}_2(x) = c_{m_2-1} \cdots c_1 c_0)$ を生成する．この vec_2 を vec_1 に連鎖した

$$vec = c_{m_2-1} \cdots c_1 c_0 b_{m_1-1} \cdots b_1 b_0$$

を，キー x の新しい特徴ベクトルに拡張する．逆に，キーの削除により，ビット長を $m_1 + m_2$ から m_1 に縮小することも可能となる．この拡張を段階的に実行する関数を Modify(x, vec_1) で定義する．

(例11) 表2と図3に新しいキー "ehime" (特徴ベクトル $vec_{eh} = 10101100$) を追加する場合，バケット4があふれ，vec_{eh} はキー ("kumamoto", "tokushima") の特徴ベクトル (vec_{ku}, vec_{to}) と同じであるので，ビット長が不足する．ここで，ビット長8の特徴ベクトルを生成する特徴関数 $\mathrm{Sign}_2 : ((3a + 4b) \bmod 8)$ により，関数 Modify を用いて特徴ベクトルを次のように拡張する．

$$vec_{eh} = \mathrm{Modify}(\text{"ehime"}, 10101100)$$
$$= 10011000 \cdot 10101100$$

同様に，$vec_{ku} = 10101001 \cdot 10101100$

$$vec_{to} = 10111011 \cdot 10101100$$

以上より，図3にキー "ehime" を追加したトライの部分図 (図4) が得られる．(例終)

均整ベクトルによるトライで部分文字列検索を行う場合，トライの最大深さを大域変数 D で保持し，手順 (S-1) で決定する特徴ベクトル vec のビット長が D となるように関数 Modify を用いればよい．このように，均整ベクトルのビット長を伸縮することで，トライの構成に必要なビット数は効率的に抑制でき，大きなキー集合に対する実用性が広がる．

4.3 バケット検索の高速化

限定深さ優先探索法により部分文字列キーの探索候補となるバケットを絞り込むが，例10のようなフォルスドロップが生じて，完全な絞り込みはできない．したがって，依然として，2次記憶上のバケットの冗長な転送が余儀なくされる．この冗長な転送をもう一段階抑制するために，各バケットに対応する長さ n のビット列であるディスクリプタ (descriptor)[15]を主記憶上に置き，バケット転送の可否を判定する．

表3 キーに対するディスクリプタの例
Table 3 An example of descriptors for keys.

x	K_Disc(x)
akita	0000001001001100
okayama	0001010100000001
okinawa	1100000100100100
osaka	0000010001000101
kagawa	0100010000000100
kumamoto	1111000100000001
kochi	0010000000000001
tokyo	1000000100000100
tokushima	1110010100000000
nagasaki	1100000001000100
nara	1000100000000010
niigata	0100111010010000
hiroshima	0010001100000011
fukui	0100100100000100
mie	0000000000000001

キー x に対するディスクリプタは特徴関数 K_Disc(x)☆で定義し，バケット *bucket* に対応するディスクリプタ B_Disc(*bucket*) は *bucket* 中の各キー y のディスクリプタ K_Disc(y) を論理和演算で重ね合わせた (superimposed) ものとする．このディスクリプタを用いた関数 Search_Bucket(x, *bucket*) を以下に示す．

[関数 Search_Bucket(x, *bucket*)]

関数 Comp_Vec(K_Disc(x), B_Disc(*bucket*)) が，FALSE ならば関数 Search_Bucket は空集合を返し，TRUE ならば *bucket* 中のすべてのキー y において x を部分文字列として含むキー y のみを要素とする集合を返す．存在しなければ空集合を返す．(関数終)

(例12) 表2のキー集合 K に対して，ビット長16の特徴ベクトルを生成する特徴関数

$$\text{K\_Disc} : (7a + 13b) \bmod 16 \qquad (6)$$

によるキーのディスクリプタを表3に，各バケットのディスクリプタを表4に示す．

関数 Search_Bucket("shima", 6) の実行を考える．

K_Disc("shima") = 0010000100000000

B_Disc(6) = 0010000000000001

に対する特徴ベクトルの比較は，

Comp_Vec(K_Disc("shima"), B_Disc(6))

☆ 前述の特徴ベクトルを生成した特徴関数と同様に x に対する特徴ベクトルを生成するが，ベクトル長，関数 f が異なるものとする．これは，トライを構成するのに用いた特徴ベクトルをディスクリプタとして使用すると，トライ走査に用いたビット列をさらに関数 Comp_Vec で比較することになり冗長となるからである．

表4 バケットに対するディスクリプタの例
Table 4 An example of descriptors for buckets.

bucket	B_Disc(*bucket*)	*bucket*	B_Disc(*bucket*)
1	0000010001000101	6	0010000000000001
2	0001010100000001	7	1100100100000110
3	1000000100000100	8	0010001100000011
4	1111010100000001	9	1100000101100100
5	0100011001001100	10	0100111010010000

より，FALSEとなるので，たとえトライ走査でバケット6が探索されても，この場合はバケット6の転送はしなくてよい.(例終)

バケットのディスクリプタは，バケット中のキーのディスクリプタをすべて重ね合わせるので，ビット1の率が高くなる．したがって，ディスクリプタのビット長は，バケット中に格納できるキーの最大数に合わせて，十分に大きくとる必要がある．

5. 実験結果の評価と考察

5.1 理論的評価

部分文字列検索法をまずバケットのアクセス回数により評価する．

トライの最大深さを D，トライの総ノード数を N $(N \leq 2^{D+1}-1)$，総バケット数を B $(B \leq 2^D)$，検索キーの特徴ベクトルにおける下位 D ビット中に含まれるビット1の数を A とすると，3.4節の探索法ではバケットのアクセス回数は $B/2^A$ 以下となる．これにディスクリプタを用いることで，この回数はより小さくなる．また，均整ベクトルはトライのノード数を少なくするので，やはりこの回数の軽減に有効である．

部分文字列検索のコストは，以上のアクセス回数だけでなく，限定深さ優先探索法による走査ノード数 α にも関係するので，この理論評価を行う．

完全2進木（$N=2^{D+1}-1$）を仮定して，限定深さ優先探索法による枝刈りで未走査となるノード数 β を議論する．ただし，最大コスト（最も枝刈りの少ない場合）となる $A=1$ の特徴ベクトルを考える．たとえば，$D=4$ の完全2進木（$N=31$）に対して，ビット1の位置を p $(0 \leq p < D)$ とすると，$p=3$ の特徴ベクトル（1000）の場合，枝刈りの対象となる枝は8本あり，これらの枝以降に存在するノードはそれぞれ1個なので，$\beta = 1 \times 8 = 8$ となる．$p=2$ の特徴ベクトル（0100）の場合，枝刈りの対象となる枝は4本あり，これらの枝以降に存在するノードはそれぞれ3（$=1+2$）個なので，$\beta = 3 \times 4 = 12$ となる．同様に，$p=1$ の場合は，$\beta = (1+2+4) \times 2 = 14$，$p=0$ の場合は，$\beta = (1+2+4+8) \times 1 = 16$ となる．すなわち，走査しないノード数は，

$$\beta = 2^p \times \sum_{i=1}^{D-p} 2^{i-1} = \frac{2^p(2^{D-p}-1)}{2-1} = 2^D - 2^p \quad (7)$$

となる．この式(7)は，他の D の値に対しても $D=4$ と同様に，$A=1$ の特徴ベクトルに対して成立する．したがって，走査ノード数 α は，

$$\alpha = (2^{D+1}-1) - (2^D - 2^p) = 2^D + 2^p - 1 \quad (8)$$

となる．

走査ノード数 α は特徴ベクトルのビット1の位置により変化する．すなわち，トライのルート側の枝にビット1があれば α は小さくなるが，第 $(D-1)$ ビットだけが1である場合，α は最大となる．4.2節で提案した長さの短い均整ベクトルを連鎖する方法は，ビット0がトライの根から極端に長く続く場合（α の爆発を招く場合）の回避に有効であるといえる．

以上をまとめると，均整ベクトルによるトライの構成は，最大深さ D および総ノード数 N の増加を効率的におさえる．また，大きなキー集合で D，N が大きくなった場合でも A/D の値はほぼ一定に保つので，部分文字列検索の効率低下を防いでいる．

表5 実験に用いたキー集合と拡張ハッシュ表の構成
Table 5 Description of the test data and the resulting extendible hash tables.

	日本語名詞	英単語
単語数	103,336	80,425
単語の平均長	3	9
特徴ベクトルのビット長	32	32
特徴ベクトルの構成	12+10+10	16+16
均整ベクトルの数	3	2
ディスクリプタのビット長	64	64
トライの最大深さ	29	19
トライの総ノード数	22,463	16,211
バケットの大きさ	16	16
バケット数	11,020	8,076

5.2 具体的評価

本手法の構成システムはC言語で記述され，Sun Microsystemsの Sparc Station2 上で稼働している．

綴りがすべて分かっているキーの検索（完全一致検索と呼ぶ），キーの追加，削除に関する性能は従来の拡張ハッシュ法と同等であるので，部分文字列検索に対する実験結果を示す．ICOTの形態素辞書[16]における日本語名詞，ロングマン辞書[17]の英単語をキー集合とする拡張ハッシュ表のデータを表5に，本手法による部分文字列検索の結果と，従来の拡張ハッシュ法に

表6 部分文字列検索の実験結果
Table 6 The simulation results of the substring search.

キー集合		日本語名詞					英単語					
文字列長		2	3	4	5	6	3	4	6	8	10	12
本手法	走査ノード数	1,431	589	289	177	112	4,180	2,456	975	430	303	180
	割合(%)	6.4	2.6	1.3	0.8	0.5	25.8	15.2	6.0	2.7	1.9	1.1
	到達バケット数	568	202	90	54	30	1,875	1,018	354	136	89	47
	割合(%)	5.2	1.8	0.8	0.5	0.3	23.2	12.6	4.4	1.7	1.1	0.6
	読込バケット数	170	50	22	14	9	824	343	104	32	23	12
	割合(%)	1.5	0.5	0.2	0.1	0.1	10.2	4.2	1.3	0.4	0.3	0.1
	時間(秒)	0.31	0.08	0.03	0.02	0.01	1.54	0.65	0.19	0.05	0.04	0.02
対象手法	読込バケット数	1,122	425	224	126	80	2,211	1,341	590	248	175	116
	割合(%)	10.2	3.9	2.0	1.1	0.7	27.4	16.6	7.3	3.1	2.2	1.4
	時間(秒)	2.00	0.85	0.39	0.21	0.13	3.34	2.19	1.07	0.46	0.32	0.20

ディスクリプタのみを採用した検索(ディスクリプタ比較によりバケットの絞り込む方法で,以後対象手法と呼ぶ)の結果を**表6**に示す.

表6の値は,部分文字列の長さ別に100件ずつ部分文字列検索を行った1件あたりの平均値を示す.走査ノード数はトライ上を疑似ベクトルにより走査したノード数であり,全ノード数に対する割合も示す.疑似ベクトルのトライ走査によって到達したバケット数(対象手法は除く),ディスクリプタ検査後実際に読み込んだバケット数に対しても,全バケット数に対する絞り込みの割合を示す.

実験結果より,本手法は対象手法と比較して2~10倍ほど高速であることが分かる.これは,本手法と対象手法の読み込みバケット数を比較すると,本手法は2/5から1/10と少なくなっており,提案した限定深さ優先探索法により到達したバケット数の軽減の効果からも分かる.また,本手法の走査ノード数は全ノード数の25.8%から0.5%におさえられ,均整ベクトルと限定深さ優先探索法によるトライ走査の効果が表れていることが分かる.検索キー x が短い場合,特徴ベクトルにおけるビット1の数が減少するので,走査ノード数は増加する.しかし,この状況では x を部分文字列として含むキー y の候補数も増加し,バケット転送数も必然的に多くなるので,全体として検索コストの増加はやむをえないといえる.ただし,本手法は,その場合でも,対象手法よりは効率的である.

また,ビット長が32である特徴ベクトルを1つだけ用いたハッシュ表の実験を行った.この結果,日本語名詞,英単語ともに数十語でトライの最大深さが32を超えてしまい,実用に耐えうるものとはならなかった.したがって,提案した均整ベクトルの有効性が分かる.

部分文字列検索で用いる特徴ベクトルの生成は,隣接した2文字により1つのビットを1とする手法[4),10)]が一般的であるので,ベクトル中におけるビット1の数は,最大でも単語の長さを超えることはない.また,4.2節の初めで述べたように,ベクトル中におけるビット0と1の数の割合が近いほど,トライの構成に必要なビット数は最も少なくなる.したがって,ベクトルの長さ m を単語の平均長 l の2倍の $2l$ とした.表5において,日本語名詞の平均長は3文字であり,日本語は2バイト文字であることから,ビット1の数は6個未満となる.よって,特徴ベクトルを構成する均整ベクトルの長さを10, 12ビットとした.同様に,英単語の平均長は9文字(バイト)であったため,均整ベクトルの長さを16ビットとした.

この均整ベクトルの長さの決定法により,使用するビット数を極力おさえて,トライをコンパクトに構成することが可能となる.しかし,フォルスドロップが多く生じるようであれば,ベクトルの長さを長くすることにより,フォルスドロップは減少されうる.

また,ベクトルを生成する際に用いる特徴関数は,ベクトルにおけるビット0と1を均等に振り分けるために,関数fを用いた.この関数fで,隣接した2文字それぞれの文字の内部コード値の和を求める際に,コード値に重みを掛けることにより,ベクトル中における1となるビットが均等に存在するように工夫した.

5.3 応用範囲への考察

トライ検索法や探索木法は,キー順検索が可能であるので,任意のキーに対して接頭辞を含むキーを検索することが可能である.また,キーを接尾辞に含む検索を行うためには,キーの文字列の並びを逆にした別個の検索表を作成する必要がある.しかしながら,接頭辞でも接尾辞でもない部分文字列検索を実現することは,現在のキー検索で実現することは不可能である.したがって,部分文字列検索は,文書全体を走査する全文検索法として実現されているが,これらの方法は文書が大きくなると検索時間も長くなる.したがって,

キー検索法で部分文字列検索を可能にした本手法の有用性は高い．

たとえば，データベースや文献検索[9]などにおけるキーワードの選定[18]で複合語をどのように分割するかは，非常に難しい問題である．しかし，本手法を利用すれば，長い単位の複合語主体としたキーワードを抽出すればよいので，それらを短い単位に分解する必要もなくなる．

6. むすび

本論文では，拡張ハッシュ法に対して部分文字列検索を可能にする方法を提案し，その理論的・具体的評価を行った．キー検索法は完全一致検索を主体に考案されているので，本手法は，キー検索法の応用範囲を広げるものとなる．

多くの実験結果により有効な均整ベクトルの長さと特徴関数を決定することが今後の課題である．

謝辞 本論文をまとめるうえで有益なご助言をいただいた株式会社リコー研究開発本部情報通信研究所の森本勝士氏に深謝いたします．

参考文献

1) Enbody, R.J. and Du, H.C.: Dynamic Hashing Schemes, *ACM Computing Surveys*, Vol.20, No.2, pp.85–113 (1988). 遠山元道（訳）：動的ハッシュ法，*bit* 別冊号，pp.43–68 (1990).
2) 青江順一：静的ハッシュ法とその応用，情報処理，Vol.33, No.11, pp.1359–1366 (1992).
3) 青江順一：動的ハッシュ法とその応用，情報処理，Vol.33, No.12, pp.1465–1472 (1992).
4) Knuth, D.E.: *The Art of Computer Programming*, Ch.3 (Sorting and Searching), pp.422–480, pp.559–563, Addison-Wesley, Reading, MA (1973).
5) Aoe, J.: *Computer Algorithms - String Pattern Matching Strategies*, IEEE Computer Society Press (1994).
6) 獅々堀正幹，清原 聡，青江順一：階層化による2進ディジタル探索（BDS）木の改善，信学論（D-I），Vol.J79-D-I, No.2, pp.79–87 (1996).
7) Burkhard, W.A.: Hashing and Trie Algorithms for Partial Match Retrieval, *ACM Trans. Database Syst.*, Vol.1, No.2, pp.175–187 (1976).
8) Rivest, R.L.: Partial-Match Retrieval Algotirthms, *SIAM J.Comput.*, Vol.5, No.1, pp.19–50 (1976).
9) 有川節夫，篠原 武，松本一教，張 裕民：重ね合わせ符号を用いた文献検索システムについて — キーワードのための重ね合わせ符号，情処学データベース・システム研報，54-2 (1986).
10) Harrison, M.C.: Implementation of the Substring Test by Hashing, *Commun. ACM*, Vol.14, No.12, pp.777–779 (1971).
11) Flajolet, P.: On The Performance Evaluation of Extendible Hashing and Trie Searching, *Acta Inf.*, Vol.20, No.4, pp.345–369 (1983).
12) Mehlhorn, K.: Dynamic Binary Search Tree, *SIAM J. Comput.*, Vol.8, No.2, pp.175–198 (1979).
13) Jonge, W.D., Tanenbaum, A.S. and Riet, R.P.: Two Access Methods Using Compact Binary Trees, *IEEE Trans. Softw. Eng.*, Vol.SE-13, No.7, 1987.
14) 島田良作，木内陽介，大松 繁：わかる情報理論，pp.7–12，日新出版 (1982).
15) Du, H.C.: On the File Design Problem for Partial Match Retrieval, *IEEE Trans. Softw. Eng.*, Vol.SE-11, No.2, pp.213–222 (1985).
16) ICOT 形態素辞書評価版，新世代コンピュータ技術開発機構（Institute for New Generation Computer Tecnology: ICOT）(1992).
17) Ilson, R.F. (Ed.): *Longman Dictionary of Contemporary English*, Longman Group (1978).
18) 小川泰嗣，望主雅子，別所礼子：複合語キーワードの自動抽出法，情報処理学会自然言語処理研究会，97-15, pp.103–110 (1993).

付録 アルゴリズムの正当性の証明

まず，最初に探索される部分文字列 x（特徴ベクトル vec）の疑似ベクトル（S(vec) の要素）について考える．手順 (S-1) より，最初に探索されるベクトルは vec に設定される．ビット 0 の枝を優先してたどる深さ優先探索法において，vec が最初に探索される疑似ベクトルであることは，疑似ベクトルの定義より明らかであり，手順 (S-2) で vec に対応したパスが走査される．

次の手順 (S-3) で設定される疑似ベクトルについて，関数 Replace で処理されるノードと枝を考える．ベクトルをビット列 α，β とビット値 c を用いて $\alpha c\beta$ と表し，ビット列 β を逆順に並べたビット列を REV(β)，ノード n からノード m へのパス上に存在するビット列 γ を PATH(n, m) $=\gamma$ で表し，スタックからポップされたトップノードを TOP とする．手順 (R-1) から (R-3) は，

PATH(1, TOP) = REV(β),
PATH(TOP, p) = 0,
PATH(TOP, q) = 1.

なるノード TOP からノード p, q の遷移まで，トライ上を根に向かって逆に走査する．そして，手順 (R-4)

で，次に探索されるベクトルを $\alpha 1\beta$ に設定する．このとき設定されたパス REV($\alpha 1\beta$) において，論理積を&で表すとき，

$vec \& \alpha 1\beta = vec$

(Comp_Vec(vec, $\alpha 1\beta$) = TRUE を意味する)，

$\alpha 1\beta \in$ S(vec).

が成立することは明らかである．すなわち，$\alpha 1\beta$ は疑似ベクトルである．この時点で関数 Replace は終了し，手順 (S-2) において，疑似ベクトル $\alpha 1\beta$ に対応したパス REV($\alpha 1\beta$) がたどられる．

したがって，以上の手順 (S-2) と (S-3) を繰り返すことにより，提案した限定深さ優先探索法は，トライ上の

PATH(n, m) = 0

(特徴ベクトルにおけるビット 0)，

PATH(n, m') = 1

なる枝を持つノード n のみを処理し，しかも，この条件を満足するすべてのノード n を探索することは明らかである．

以上より，提案したアルゴリズムは，すべての疑似ベクトルを探索する．

(平成 8 年 5 月 31 日受付)
(平成 8 年 11 月 7 日採録)

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 年同大学院博士前期課程修了．現在同大学院博士後期課程在学中．情報検索，自然言語処理の研究に従事．電子情報通信学会会員．

森田 和宏（正会員）

昭和 47 年生．平成 7 年徳島大学工学部知能情報工学科卒業．現在同大学院博士前期課程在学中．自然言語処理の研究に従事．

獅々堀正幹（正会員）

昭和 40 年生．平成 3 年徳島大学工学部情報工学科卒業．平成 5 年同大学院博士前期課程修了．平成 7 年同大学工学部知能情報工学科助手．情報検索，文書処理，自然言語処理の研究に従事．情報処理学会第 45 回全国大会奨励賞受賞．電子情報通信学会会員．

青江 順一（正会員）

昭和 26 年生．昭和 49 年徳島大学工学部電子工学科卒業．昭和 51 年同大学院修士課程修了．同年同大学工学部情報工学科助手．現在同大学工学部知能情報工学科教授．この間コンパイラ生成系，パターンマッチングアルゴリズムの能率化の研究に従事．最近，自然言語処理，特に理解システムの開発に興味を持つ．著書「Computer Algorithms — Key Search Strategies —」，「Computer Algorithms — String Matching Strategies —」IEEE CS press．平成 4 年度情報処理学会「Best Author 賞」受賞．工学博士．電子情報通信学会，人工知能学会，日本認知科学会，日本機械翻訳協会，IEEE，ACM，AAAI，ACL 各会員．