圧縮テキストに対するパターン照合機械の高速化

宮崎正路† 深町修一†† 竹田正幸† 篠原武††

本稿では，テキストの圧縮による文字列パターン照合の高速化を扱う．通常，テキストは二次記憶装置に格納されるため，パターン照合の処理時間の大半はテキストのデータ転送に費やされる．深町ら (1992) は Huffman 符号を用いてテキストを圧縮し，データ転送量を減少させることによって処理時間を短縮する方式を提案した．その際，圧縮テキストを一度復号したうえで照合したのでは高速化されない．そこで，深町らは Aho-Corasick 法をもとに，圧縮テキストをそのまま走査するパターン照合機械の構成法を開発した．しかし，この方法では1ビットごとに状態遷移を行う必要がある．深町らは符号単位を4ビットとした符号化によって状態遷移回数を減らすことにより高速化を図っているが，圧縮率が低下するという問題がある．そこで本稿では，Huffman 符号用パターン照合機械の複数ビット分の状態遷移を1回の状態遷移で置き換えることにより，圧縮されたテキストを高速に走査可能なパターン照合機械の実現法を提案する．これにより，Huffman 符号による圧縮の効果を十分に引き出すことができる．英文テキストを用いた実験では，まとめ読みの単位を4ビット（8ビット）とした場合の走査時間は，非圧縮テキストを4ビット（8ビット）ずつ走査した場合の63%（69%）に短縮できた．

Speeding up the Pattern Matching Machine for Compressed Texts

Masamichi Miyazaki,† Shuichi Fukamachi,†† Masayuki Takeda†
and Takeshi Shinohara††

In this paper, we consider the problem of speeding up the pattern matching by text compression. In practice the text is large and is stored in secondary storage, hence most of the time required for pattern matching is devoted to data transmission. Fukamachi, *et al.* (1992) proposed a pattern matching algorithm for Huffman coded strings, which is based on the Aho-Corasick algorithm and saves the data transmission time. If the algorithm makes bit by bit state transitions, the state transition time is increased. Fukamachi, *et al.* used a special code in which the lengths of code words are multiple of four bits and showed an algorithm for constructing the pattern matching machine for texts compressed by this code. However when the compression rate is not so good, we cannot accelerate the pattern matching very much. In this paper we propose an efficient realization of the pattern matching machine for Huffman coded text. The idea is quite simple: Substitute one state transition for consecutive t state transitions of the original machine. This technique enables us to fully exploit the effect of the Huffman compression. The experiments using the Brown corpus showed that when $t = 4$ ($t = 8$) the running time is reduced to 63 (69) percent of the uncompressed case.

1. はじめに

テキスト文字列の中から目的とするパターン文字列の出現を検出する文字列パターン照合問題は，情報検索における最も基本的な問題として古くから盛んに研究され，Knuth-Morris-Pratt 法[1)]や Boyer-Moore 法[2)]をはじめとして多くの効率的パターン照合アルゴリズムが提案されている．しかし，一般にテキストはディスクなどの二次記憶装置に格納されるため，処理時間の大半がテキストのデータ転送に費やされることになる．したがって，パターン照合アルゴリズム自体の改良によって処理時間を短縮することには限界がある．

そこで，テキストをあらかじめ圧縮しておくことによってデータの転送量を減らして高速化を図る方式が提案されている[3)~5)]．その際，圧縮テキストを復号したうえでパターン照合を行ったのでは復号する手間

† 九州大学大学院システム情報科学研究科情報理学専攻
Department of Informatics, Kyushu University
†† 九州工業大学情報工学部知能情報工学科
Department of Artificial Intelligence, Kyushu Institute of Technology

がかかり高速化されないため，圧縮テキストを復号せずにそのままパターン照合する技法を開発する必要がある．

圧縮テキスト上のパターン照合に関しては，主として理論的興味から，LZ77 符号[6]や LZW 符号[7]などの適応型圧縮法（adaptive compression method）について研究が行われている[8)~12)]．しかし，これらの研究は，圧縮テキストを高速に照合することを目的としたものであって，本稿のようにテキスト圧縮によってパターン照合の高速化を目指すものとは異なる．適応型圧縮法では，文字列の符号化がそれ以前のテキストに依存するためパターン照合と同時に圧縮のための情報を何らかの形で保持しておく必要があり，そのためのコストは小さくない．したがって，非圧縮テキストをそのままパターン照合する場合と比べると，多くの処理時間を要する．そこで，本稿では非適応型圧縮法のみを考えることにする．

さて，圧縮テキスト上でパターン照合を行う際に生じる問題は，次の3つである．

(1) 圧縮テキスト上において符号化されたパターンを探索する際，符号語の先頭ビットを決定するための処理が必要である．すなわち，ビットのずれ読みによるパターンの誤検出を避けるために何らかの機構が必要である．

(2) 1ビット単位の符号を用いると，ビットごとの処理が必要となり，高速な処理が望めない．

(3) パターンの符号化が一意でないことがある．特に文字列単位の圧縮では，探索すべきパターンの符号化の数が組合せ的に増大しうる．

Manber[4]は，頻度の高い連続 2 文字の組に対して ASCII コードの未使用コードを割り当てる符号を用いてテキストを圧縮し，符号化されたパターンを任意のパターン照合アルゴリズムによって探索する方式を提案した．8ビット固定長符号であるため，(1)，(2) の問題を避けることができる．また，(3) については，任意の文字列に対してその符号化の数を小さく抑える方式を示している．しかし，この符号では圧縮率は英文テキストに対して 70%程度であり，処理時間の短縮も同程度にとどまっている．

また，松本ら[5]は，$2^n(k)$ 符号と呼ぶ符号を用いた方法を提案している．この符号は，n ビットを符号単位とし，2^n 個の符号単位のうちの k 個は符号語の最終 n ビットとしてのみ生起する．すなわち，(1) の問題が生じないように符号語に制限を加えている．しかし，(2) を避けるために n を大きくとると圧縮率が低下する．圧縮率を改善するためには，文字単位でなく文字列単位の符号化を行えばよいが，辞書が膨大になりその読み込みに時間がかかる．また (3) の問題が生じる．松本らによれば，$n = 8$ とし，2グラム統計に基づいて英文テキストを符号化した場合の走査時間の短縮は 70%程度である[5]．

一方，深町ら[3]は，Huffman 符号によって圧縮したテキストをそのまま照合するパターン照合機械の構成アルゴリズムを，Aho-Corasick 法[13]の拡張として提案している．Huffman 符号は可変長符号であるため (1) の誤検出の問題が生じるが，このアルゴリズムでは，符号語の集合を認識する有限オートマトンをパターン照合機械に埋め込むことによって，処理速度を低下させずに誤検出を防ぐことに成功しており，この点が他の方法に比べ特徴的である．この方法では (3) の問題は生じない．一方，(2) の問題は重要である．実際，圧縮テキストの 1 ビットごとにパターン照合機械の状態遷移と出力の有無の判定が必要となる．この問題を解決するためには，次の 2 つの方法が考えられる．

方法 1 圧縮率は犠牲になるが，4 ビット単位で符号化する．

方法 2 パターン照合機械の遷移表を変換して，t ビット分の状態遷移を 1 回の遷移に置き換える．

方法 1 では，テキストの種類によっては圧縮効率がかなり悪化するため，圧縮による高速化の効果が低下する．そこで，本稿では方法 2 を採用する．これにより，圧縮率を犠牲にすることなく (2) の問題を解消できるため，走査時間をより大きく短縮することが期待できる．ただし，t ビットで遷移する経路によって出力が異なるため，各状態からの遷移に対してそれぞれ出力を付ける必要がある．

パターン照合機械を高速に走査させるためには，failure 関数を除去して決定性オートマトンに変換し[13]，その状態遷移関数を表一瞥（table-look-at）法[14]により実現する方法が有効である[15]．すなわち，状態を表す整数と文字コード自身とを引数とする 2 次元配列によって実現することで，次状態をただ一度の表一瞥で決定できる．しかし，テキストを 8 ビットずつ走査する通常のパターン照合機械では，表のサイズが状態数 × 256 となり，パターン長の総和が大きいときには現実的ではない．そこで，Arikawa ら[15]は，テキストを 4 ビットずつ走査する文字符号分割法を提案している．これにより，表のサイズを 1/8 以下に抑えることができ，走査時間はたかだか 1.5 倍である．本稿で提案する方法 2 においても表一瞥法を用いるが，まとめ読みの単位 t に関して同様の問題が生じる．すなわ

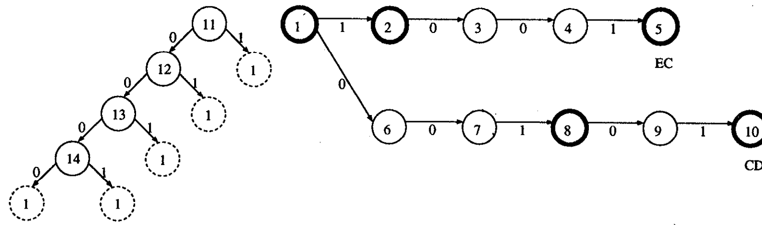


図1 Huffman 木を付け加えた goto グラフ
Fig. 1 Goto graph with Huffman tree.

ち，$t=8$ とすると走査時間は短縮されるが領域量および構成時間が増大する．したがって，パターン長の総和が比較的大きい場合には $t=4$ とするのが妥当である．

本稿では，方法2の実現法の有効性を検証するため，パターン照合機械の構成時間，領域量，および走査時間について，方法1および非圧縮テキスト用のパターン照合機械との比較実験を行った．構成時間と領域量に関しては，5.2節で示すようにまとめ読みの単位を4ビットとした場合には十分小さく，実用的であるといえる．また，走査時間に関しては，代表的な英文コーパスであるBrownコーパスを対象とした実験において，Huffman符号による圧縮率は61%であり，まとめ読みの単位を $t=8$ とした場合の走査時間は，非圧縮テキストを8ビットずつ走査した場合の69%に短縮できた．また，$t=4$ とした場合は，非圧縮テキストを4ビットずつ走査した場合の63%となった．一方，方法1を用いた場合には，圧縮率は64%であり，走査時間は，非圧縮テキストを4ビットずつ走査した場合の71%であった．このように，本稿で提案する方法2の実現法は，構成時間，領域量，および走査時間の面から実用的である．

2. 圧縮テキスト用のパターン照合機械

この節では，深町ら[3]によって提案されたHuffman符号で圧縮したテキストをそのまま走査するパターン照合機械（以下，pmmと略記する）の構成アルゴリズムとその実現について述べる．

2.1 Huffman 符号用の pmm の構成

たとえば，**表1** に示すHuffman符号によってテキストが圧縮されているとする．また，検索するパターンを EC (1 001) および CD (001 01) とする．このとき，次のようにしてpmmを構成する．

表1 Huffman 符号
Table 1 Huffman code.

文字	符号語
A	0 0 0 0
B	0 0 0 1
C	0 0 1
D	0 1
E	1

まず，初期状態1のみから成るgotoグラフを作る．次にパターン EC に対するgotoグラフを作る．EC に対するHuffman符号は1 001であるので先頭ビット符号1に対して状態2を用意して状態1からラベル1付きの辺を状態2へ伸ばす．すなわち，$goto(1,1)=2$ とする．これをパターンの符号がなくなるまで繰り返す．出力関数は最後に遷移した状態が5であるので $output(5)=\{EC\}$ と定義する．他のパターンに対しては，初期状態からgoto関数によって遷移させ，遷移できなくなったときだけ新しい状態と辺を加える．パターン CD に対しても，同様な処理をする．こうして得られたgotoグラフを**パターン木**と呼ぶことにする．次に，このパターン木をHuffman木と結合して**図1** のようなgotoグラフを作成する．このとき，Huffman木の葉に当たる部分はパターン木の根である状態1へ遷移するようにする．図1において，太線の円は文字の切れ目の状態を示し，それ以外は中間状態を示す．

次にfailure関数の計算を行う．初めに初期状態1のfailure関数の値をHuffman木の根とする．つまり $failure(1)=11$ とする．あとはAho-Corasick法と同様にパターン木の各節点に対して幅優先順にfailure関数の値を計算する．また出力関数も同様に更新する．以上の操作によって作成されたpmmを**図2** に示す．破線矢印はfailure遷移を示す．

gotoグラフの作成の際に，パターン木にHuffman木を付加した．これは符号のずれ読みを防ぐためである．そのことを説明しよう．Huffman木を付け加えずにpmmを構成すると**図3** のようになる．ここで，テキスト $CCDB$ (001 001 01 0001) に対する2つのpmmの動作を，それぞれ，**図4** と**図5** に示す．実線矢印はgoto遷移を示し，破線矢印はfailure遷移を示す．図5において，入力を001 0まで読んだ状態9で

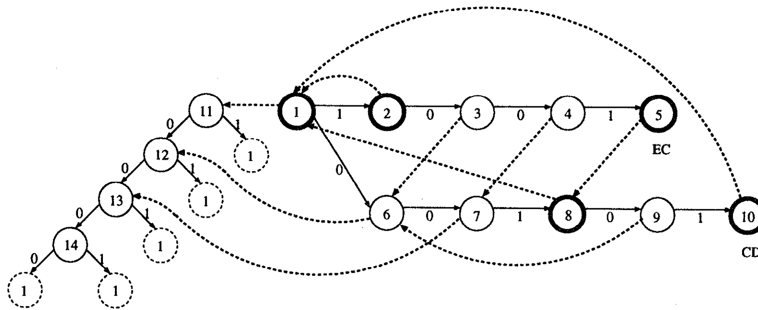


図2 Huffman 符号用 pmm
Fig. 2 pmm for Huffman code.

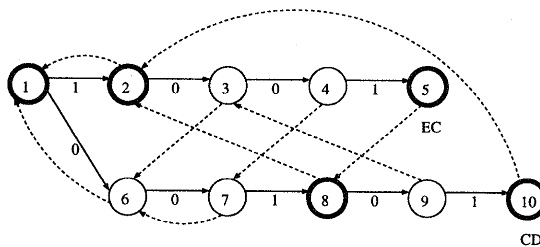


図3 パターンを誤検出する pmm
Fig. 3 pmm that leads misdetection of patterns.

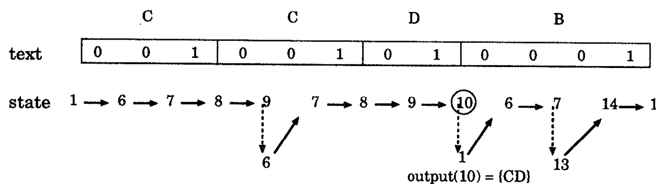


図4 図2の pmm の動作
Fig. 4 Behavior of pmm of Fig. 2.

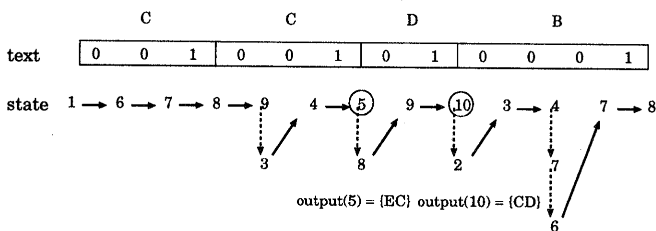


図5 図3の pmm の動作
Fig. 5 Behavior of pmm of Fig. 3.

の動作に注目しよう．次の入力は0であるので，状態3へ failure 遷移している．これまでの入力 001 0 は文字 C と0まで読んでいるのに対して，この状態3は文字 E と0まで読んでいると見なされる．つまり，Huffman 木なしの pmm では，failure 遷移で符号のずれが生じたために，パターン EC を誤検出している．一方，図4では，状態6へ failure 遷移しているがこの状態は0を読んだ状態であり，符号のずれは生じていない．このように，goto グラフに Huffman 木を付け加えて failure 関数を構成することにより誤検出のない pmm を作成できる．

また，この Huffman 木は符号語の集合を受理する決定性有限オートマトン（DFA）で置き換えることができる[16]．したがって，図6のように pmm 中で Huffman 木が占める状態数を減らすことができる．このような最小状態 DFA は Huffman 木の節点の個数に比例した時間で構成できる．

上で示した pmm の構成法は，Huffman 符号に限ら

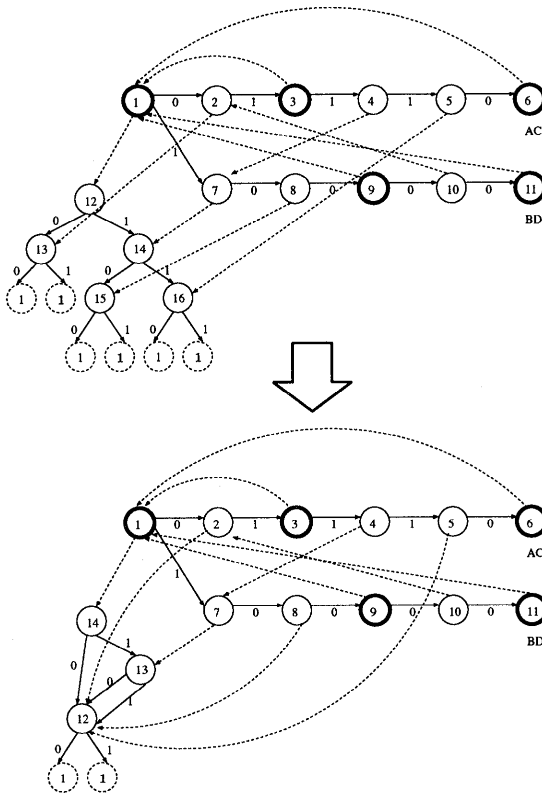


図6 Huffman 木から DFA への置き換え
Fig. 6 Replacement of Huffman tree with DFA.

ず語頭条件を満たすような任意の符号に対して有効である．ここで，符号が語頭条件を満たすとは，任意の符号語が他の符号語の接頭語とならないことをいう．

2.2 実 現 法

pmm を高速に走査させるためには，1 章で述べたように，failure 遷移を除去して決定性オートマトンへ変換し，その状態遷移関数を表一瞥法により実現すればよい[15]．しかし，Huffman 符号用の pmm は，圧縮テキストを 1 バイト走査するのに状態遷移と出力の有無の判定をそれぞれ 8 回ずつ行うため，1 章で述べた (2) の問題が生じ高速化されない．この問題を解決するためには，1 章で示したように 2 つの方法が考えられる．深町ら[3]による方法（方法 1）では，符号単位が 4 ビットの符号を用いるため，pmm は圧縮テキスト上を 1 バイト走査するのに状態遷移と出力の判定をそれぞれ 2 回ずつ行う．しかし，圧縮効率が下がるために圧縮による効果が低減するという欠点があった．

そこで本稿では，符号単位は 1 ビットとし pmm の状態遷移を t ビット単位で行う実現法（方法 2）を提案する．このような pmm を，**t ビットまとめ読みパターン照合機械**と呼ぶことにする．次章では，まとめ読みパターン照合機械の構成法について述べる．

3. まとめ読みパターン照合機械

Huffman 符号用の pmm の状態遷移表を変換して，t ビット分の状態遷移を 1 回の状態遷移で置き換えることを考える．pmm のある状態から t ビットに対する状態遷移は 2^t 通りある．そこで，t ビット先の状態を pmm のすべての状態に対してあらかじめ計算しておき，テキストを走査する際に t ビットずつまとめて状態遷移する．この際，t ビット分の状態遷移の途中でパターンを検出することがあるため，各状態からの遷移に対して，それぞれ出力を付けておく必要がある．

$\Sigma_t = \{0, 1, \ldots, 2^t - 1\}$ とおく．もとの Huffman 符号用 pmm において failure 遷移を除去[13]して得られる状態遷移関数を $\delta : Q \times \Sigma_1 \to Q$ とし，出力関数を $\lambda : Q \to 2^K$ とする．ただし，Q は pmm の状態の集合，K はパターンの集合とする．また，以下のようにして，δ を $Q \times \Sigma_1^*$ 上の関数に拡張する．

$$\delta(q, \varepsilon) = q$$
$$\delta(q, xa) = \delta(\delta(q, x), a)$$

ここで，$q \in Q$，$a \in \Sigma_1$，$x \in \Sigma_1^*$ である．t ビットまとめ読み pmm の状態遷移関数 $\bar{\delta} : Q \times \Sigma_t \to Q$ と出力関数 $\bar{\lambda} : Q \times \Sigma_t \to 2^{\{1,\ldots,t\} \times K}$ を次のように定義する．

$$\bar{\delta}(q, i) = \delta(q, i_1 i_2 \ldots i_t)$$

$$\bar{\lambda}(q, i) = \left\{ \langle j, x \rangle \;\middle|\; \begin{array}{l} 1 \le j \le t \text{ かつ} \\ x \in \lambda(\delta(q, i_1 i_2 \ldots i_j)) \end{array} \right\}$$

ここで，$i_1 i_2 \ldots i_t$ は i の 2 進表現とする．すなわち，$i = i_1 \cdot 2^0 + i_2 \cdot 2^1 + \cdots + i_t \cdot 2^{t-1}$ とする．このとき，t ビットまとめ読み pmm の状態遷移関数 $\bar{\delta}$ と出力関数 $\bar{\lambda}$ は，**図 7** に示すアルゴリズムにより，$O(|Q| \cdot 2^t)$ 時間で構成することができる．

図 8 に，図 2 に示した Huffman 符号用 pmm について，決定性に変換した pmm とそれを 4 ビットまとめ読み pmm へ変換する様子を示した．図 8 には，状態 4 からの遷移のみを示した．たとえば，状態 4 で入力が 0011 の場合，遷移先は 2 となり，出力は $\emptyset$ である．また，状態 4 で入力が 1011 のとき，遷移先は 2 となり，出力は $\{\langle 1, EC \rangle, \langle 3, CD \rangle\}$ となる．

4. 走査時間と領域量

pmm の走査時間はテキスト長に比例する．また，pmm の構成時間と領域量はいずれもパターン長の総和に比例する．しかし，この比例定数が問題である．この章では，以下の 3 種類の pmm の実現法について

pmm の走査時間と領域量を理論的に考察する．なお，pmm の構成時間については，5.2 節において実験結果をもとに考察する．

(A) 非圧縮テキストを t ビットずつ読んで状態遷移する pmm.

(B) 符号単位を t ビットとする符号化で圧縮したテキストを t ビットずつ読んで状態遷移する pmm.

(C) Huffman 符号で圧縮したテキストを t ビットずつ読んで状態遷移する pmm.

入力：状態遷移関数 $\delta : Q \times \Sigma_1 \to Q$
　　　出力関数 $\lambda : Q \to 2^K$
出力：t ビットまとめ読み状態遷移関数 $\bar{\delta} : Q \times \Sigma_t \to Q$
　　　t ビットまとめ読み出力関数
　　　　$\bar{\lambda} : Q \times \Sigma_t \to 2^{\{1,\ldots,t\}\times K}$
begin
　for each $q \in Q$ **do begin**
　　$w := 1$; $\bar{\delta}(q,0) := q$; $\bar{\lambda}(q,0) := \emptyset$;
　　for $i := 1$ **to** t **do begin**
　　　for $j := 0$ **to** $w-1$ **do begin**
　　　　$s := \bar{\delta}(q,j)$; $t := \bar{\lambda}(q,j)$;
　　　　$\bar{\delta}(q,j) := \delta(s,0)$;
　　　　$\bar{\delta}(q,j+w) := \delta(s,1)$;
　　　　$\bar{\lambda}(q,j) := t \cup \{\langle i,x\rangle \mid x \in \lambda(\bar{\delta}(q,j))\}$;
　　　　$\bar{\lambda}(q,j+w)$
　　　　　$:= t \cup \{\langle i,x\rangle \mid x \in \lambda(\bar{\delta}(q,j+w))\}$
　　　end;
　　　$w := 2 * w$
　　end
　end
end.

図7　パターン照合機械の変換アルゴリズム
Fig. 7　Algorithm for converting pattern matching machine.

4.1 走査時間

pmm の走査時間は，I/O 時間と CPU 時間の和と考えることができる．ここで CPU 時間とは，状態遷移に要するコストと出力の有無の判定に要するコストの和と考える．すなわち，パターンが検出された際の出力に関わるコストは考慮に入れないことにする．いま，データ転送コストを 1 ビットあたり d 秒とすると，非圧縮テキスト 1 文字あたりの I/O 時間コストは，(A)，(B)，(C) に対して，それぞれ，$8d$，$C_t d$，$C_1 d$ となる．ただし，C_n は符号単位 n ビットの符号化による平均符号長を表すものとする．一方，状態遷移表を用いて次状態を決定するコストを g，出力判定に要するコストを o とすると，テキスト 1 文字あたりの CPU 時間コストは，(A)，(B)，(C) に対して，それぞれ，$(8/t)g+o$，$C_t(g+o)/t$，$C_1(g+o)/t$ となる．ここで単位はいずれも秒である．非圧縮テキスト 1 文字あたりの I/O 時間と CPU 時間をあわせたコストを，それぞれ，$time_A$，$time_B$，$time_C$ とすると，次のようになる．

$$time_A = (8/t)g + o + 8d$$
$$time_B = C_t((1/t)(g+o)+d)$$
$$time_C = C_1((1/t)(g+o)+d)$$

いずれも，状態遷移の単位である t が大きいほど時間

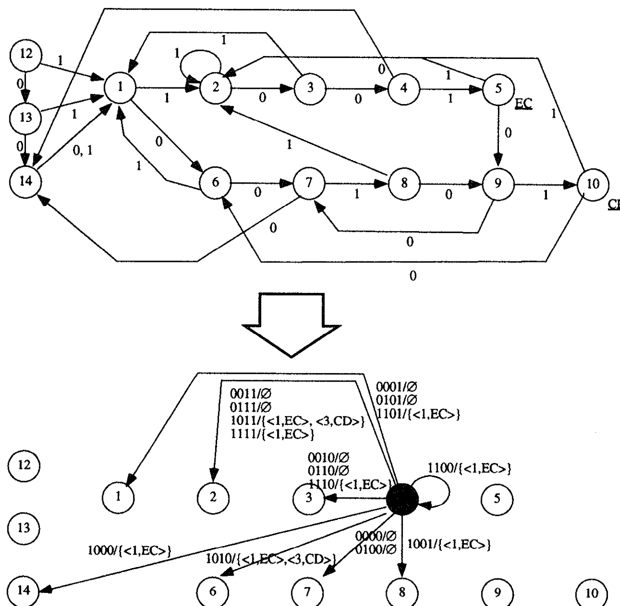


図8　状態遷移関数と出力関数の変換
Fig. 8　Conversion of state transition and output functions.

表2 状態遷移関数のための領域量（整数）
Table 2 Space required for the state-transition function (integers).

	(A)	(B)	(C)
$t=1$	$16(\ell+1)$	$2(C_1\ell+256)$	$(2+2)(C_1\ell+256)$
$t=2$	$16(\ell+1)$	$4((C_2/2)\ell+86)$	$(4+2)(C_1\ell+256)$
$t=4$	$32(\ell+1)$	$16((C_4/4)\ell+18)$	$(16+2)(C_1\ell+256)$
$t=8$	$256(\ell+1)$	$256(\ell+2)$	$(256+2)(C_1\ell+256)$

表3 出力関数のための領域量（ポインタ）
Table 3 Space required for the output function (pointers).

	(A)	(B)	(C)
$t=1$	$8(\ell+1)$	$C_1\ell+256$	$(3\cdot2+1)(C_1\ell+256)$
$t=2$	$4(\ell+1)$	$(C_2/2)\ell+86$	$(3\cdot4+1)(C_1\ell+256)$
$t=4$	$2(\ell+1)$	$(C_4/4)\ell+18$	$(3\cdot16+1)(C_1\ell+256)$
$t=8$	$\ell+1$	$\ell+2$	$(3\cdot256+1)(C_1\ell+256)$

は短縮される．また，$time_C/time_B = C_1/C_t$ が成り立つ．すなわち，t ビットごとに遷移を行う pmm では，(B) と (C) の走査時間の比は，そのまま平均符号長の比になっている．さらに，

$$time_C = (C_1/8)time_A + C_1(1/t - 1/8)o$$

であることから，$time_C$ は $time_A$ に圧縮率 $C_1/8$ を乗じた値より第2項の分だけ大きくなることが分かる．5章で示すように，代表的な英文コーパスである Brown コーパスを用いた実験では，圧縮率は61%であるのに対し，$t=4$ としたときの走査時間の比は，$time_C/time_A = 0.63$ であった．

4.2 領域量

pmm の領域量は，状態遷移関数の領域量と出力関数の領域量の和である．ただし，(C) の場合には，まとめ読み pmm に変換する前の pmm のための領域量も考慮しなければならない．

まず，状態遷移関数は，表一瞥法により，状態数 × アルファベットサイズの表で実現する．いま，探索するパターンの長さの総和を ℓ で表すことにする．(A) では，状態数はたかだか $(8/t)(\ell+1)$ である．(B) では，符号単位を t ビットとした場合，パターン木の節点数がたかだか $C_t\ell/t+1$ であり，符号木の内部節点数が $255/(2^t-1)$ であるため，状態数はたかだか $C_t\ell/t + 255/(2^t-1)+1$ となる．(C) の状態数は t の値によらずたかだか $C_1\ell+256$ である．以上により，状態遷移関数に必要な領域量は**表2**のようになる．(C) については，まとめ読み pmm へ変換する前の pmm の状態遷移関数のための $2(C_1\ell+256)$ を加えていることに注意されたい．

次に，出力関数の領域量について考える．(A)，(B) の場合，状態 s に対する出力関数の値 $\lambda(s)$ はパターンの集合 K の部分集合であるが，$\lambda(s) \neq \emptyset$ のとき，$\lambda(s)$ の最長のパターンを α とすると，$\lambda(s) = \{\beta \in K \mid \beta$ は α の接尾語$\}$ となる．すなわち，pmm の各状態は，本質的には，出力のうちの最長のパターンだけを記憶すればよい．したがって，(A)，(B) の場合は，状態数分のポインタで十分である．これに対して (C) では，出力の値は状態と入力の両方に依存するため，まず，状態数 × アルファベットサイズのポインタが必要である．さらに，t ビット分の遷移の途中での出現パターンリストのための領域も必要となる．そこで，細かな議論は省くが，最悪の場合に必要なポインタ数は，$3\cdot2^t(C_1\ell+256)$ となる．これに，変換前の pmm の出力関数のために必要な $C_1\ell+256$ を加えなければならない．以上をまとめると，**表3**のようになる．(C) に関しては，実際には，これよりかなり小さくなることが予想される．これについては，5.2節において，実験により領域量の比較を行う．表2と表3より，8ビットまとめ読み pmm は，4ビットまとめ読み pmm の16倍程度の領域を要することが分かる．したがって，ℓ が大きい場合にはまとめ読みの単位を $t=4$ とするのが現実的である．

5. 実験と評価

本稿で提案したまとめ読み pmm の有効性を評価するため，次に示す5つの pmm について，pmm の構成時間，領域量，および走査時間の比較実験を行った．なお，実験には，Sun Microsystems 社の SPARCstation2 を用いた．

- 非圧縮テキストに対して4ビットずつ状態遷移する pmm ((A) $t=4$)．
- 非圧縮テキストに対して8ビットずつ状態遷移する pmm ((A) $t=8$)．
- 符号単位4ビットで圧縮したテキストに対して4

表4 実験に使用したテキスト
Table 4 Texts used in the experiment.

テキスト	エントロピー (ビット/文字)	符号単位 (ビット)	平均符号長 (ビット)	圧縮率 (%)	圧縮効率 (%)
テキスト1	4.804	1	4.842	60.52	99.23
		4	5.151	64.39	93.27
テキスト2	4.475	1	4.497	56.21	99.50
		4	4.944	61.80	90.50
テキスト3	4.334	1	4.364	54.55	99.30
		4	4.964	62.05	87.30
テキスト4	2.602	1	2.666	33.33	97.59
		4	4.121	51.51	63.14

ビットずつ状態遷移する pmm ((B) $t=4$).

- Huffman 符号で圧縮したテキストに対して4ビット分まとめて状態遷移する pmm ((C) $t=4$).
- Huffman 符号で圧縮したテキストに対して8ビット分まとめて状態遷移する pmm ((C) $t=8$).

5.1 テキストと圧縮率

実験では，以下に示す4種類のテキストを用いた.

テキスト 1 Brown コーパス．大きさは 6.8 MB で，エントロピーは 4.80 である．

テキスト 2 シェークスピアの作品など．大きさは 7.1 MB で，エントロピーは 4.48 である．

テキスト 3 塩基配列データベース GenBank の一部．大きさは 8.3 MB で，エントロピーは 4.33 である．

テキスト 4 塩基配列データベース GenBank の一部から塩基配列と LOCUS，ACCESSION だけを抜き出したもので，a，c，g，t の4文字が大部分を占める．大きさは 17.1 MB で，エントロピーは 2.60 である．

これら4つのテキストに対して，符号単位を1ビット，4ビットとする符号化により圧縮を行った際の圧縮率と圧縮効率を**表4**に示す．各テキストはエントロピーがそれぞれ異なっており，エントロピーと走査速度の関係について調べることができる．テキスト4については，エントロピーが小さく符号単位を1ビットとした場合と4ビットとした場合の圧縮効率の差が，他のテキストに比べ大きいため，方法1と方法2で走査速度に大きな差が出ることが予想される．

5.2 構成時間と領域量の比較

pmm の構成時間は，パターン長の総和に比例すると考えられる．そこで，先に述べた5つの pmm の実現法に対して，パターン長の総和を変化させてそれぞれの構成時間を調査した．その結果を**図9**に示す．また，これらの pmm のサイズを**図10**に示す．なお，状態番号およびポインタはそれぞれ4バイトとした．

図9と図10から，(C) の $t=8$ の pmm，すなわち，8ビットまとめ読み pmm は，その他の pmm に比べて非常に多くの構成時間および領域量を要しており，傾きも大きいため，パターン長の総和が大きい場合には有効ではないことが分かる．また，ここで注目すべきは，4ビットまとめ読み pmm ((C) $t=4$) に比べ，非圧縮テキストを8ビットずつ走査する通常の pmm ((A) $t=8$) の方が多くの構成時間と領域量を要している点である．(A) の $t=8$ の pmm は，構成法は単純だが，状態遷移関数の表のサイズが状態数 × 256 と大きいため，構成時間と領域量が大きくなっている．一方，まとめ読み pmm の作成は，一見すると多くの時間と領域を要するように思われるが，$t=4$ とした場合には時間・領域とも十分小さく，構成時間は深町らの方法 ((B) $t=4$) と比べても2倍程度にすぎない．この程度の構成時間であれば，テキストが大きい場合には走査時間に比べ無視できる．このように，4ビットまとめ読み pmm は，構成時間および領域量の両面から実用的である．

5.3 走査時間の比較

pmm の走査時間は I/O 時間と CPU 時間の和である．4.1 節では，この CPU 時間を，状態遷移に要するコストと出力の有無の判定に要するコストの和として考察を行った．しかし，実際には，出力が検出された際の処理に要するコストも走査時間として考慮しなければならない．すなわち，走査時間はパターン検出の頻度に依存して変化する．そこで，次で定義するヒット率による走査時間の変化を調べた．

ヒット率
　　＝パターンの総生起回数/テキストの総文字数

実験では，テキストとして Brown コーパス（テキスト1）を用いた．その結果を**図11**に示す．図11から，8ビットまとめ読み pmm ((C) $t=8$) は，その他の pmm に比べ，パターンのヒット率が高くなるにつれ，走査時間が急激に増大することが分かる．これは，4.2 節で述べたように出力関数の表が大きいため，パ

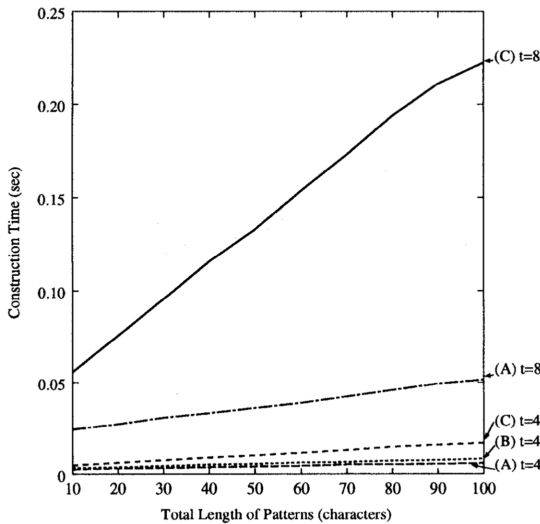


図 9 パターン長の総和と構成時間
Fig. 9 Total length of patterns and construction time.

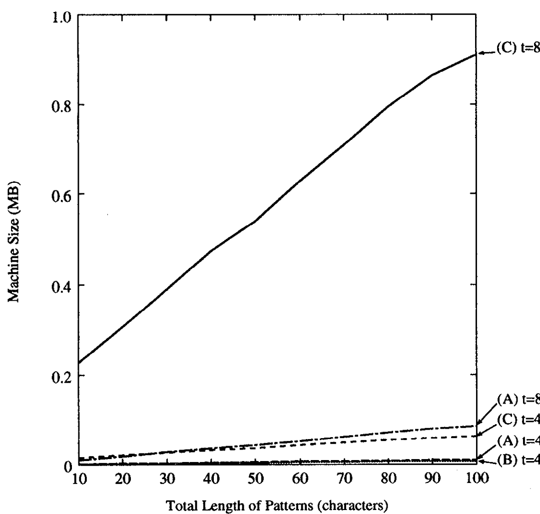


図 10 パターン長の総和と領域量
Fig. 10 Total length of patterns and machine size.

ターンが検出された際に出力関数の表を参照するコストが大きくなるためであると考えられる．8 ビットまとめ読み pmm は，ヒット率が 6%程度までであれば，他の pmm より高速である．すなわち，ヒット率が比較的小さいときに有効である．実用的には，テキスト長に比べパターンの生起回数はそれほど多くないのでヒット率が 1%を超えることは稀である．

次に，ヒット率が 0 となるようにパターン集合を選び，それぞれの pmm を 4 つのテキストに対して走査させ，次のように定義する走査速度を調べた．

走査速度 = テキストの総文字数/走査時間

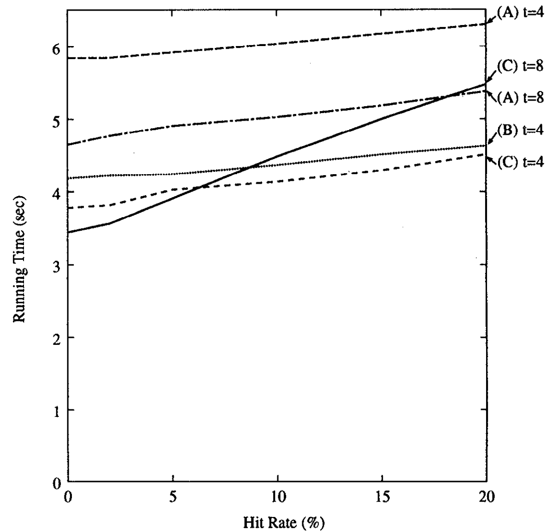


図 11 ヒット率と走査時間（Brown コーパス）
Fig. 11 Hit rate and running time (Brown corpus).

表 5 にその結果を示す．表 5 より，いずれのテキストに対しても，8 ビットまとめ読み pmm が最も高速であることが分かる．しかし，8 ビットまとめ読み pmm は，5.2 節で述べたように，pmm の構成時間および領域量が大きいため，パターン長の総和が大きいときには 4 ビットまとめ読み pmm が現実的である．そこで，表 5 には，4 ビットまとめ読み pmm の走査速度を 1 としたときの走査速度の値も示している．4 ビットまとめ読み pmm は，(A)，(B) の pmm に比べ，いずれのテキストに対しても最も高速である．深町ら[3]の方法である (B) の $t = 4$ の pmm に比べると，テキスト 1〜3 に対しては走査時間を 90%程度短縮できており，テキスト 4 に対しては 65%に短縮できていることが分かる．

5.4 実質走査速度

5.3 節では，テキストの圧縮によるパターン照合の高速化の度合を評価するために，走査速度，すなわち，単位時間あたりに処理した（もとのテキストでの）文字数を指標として用いた．しかし，この値はテキストによって変化してしまう．そこで，テキストに依存しない指標を与えるために，単位時間あたりに処理した実質の情報量を比較することを考えよう．すなわち，実質走査速度を以下のように定義する．

総情報量 = テキストの総文字数×エントロピー
実質走査速度 = 総情報量/走査時間
= 走査速度×エントロピー

表 6 に，5 つの pmm について，4 つのテキストに対する実質走査速度の値を示した．(C) のまとめ読み pmm

表5 ヒット率 = 0 のときの走査速度(文字/マイクロ秒)
Table 5 Running speed for hit rate = 0 (characters/μsec).

pmm の実現法	テキスト 1		テキスト 2		テキスト 3		テキスト 4	
(A) $t = 4$	1.165	(0.63)	1.187	(0.60)	1.179	(0.59)	1.187	(0.36)
$t = 8$	1.469	(0.80)	1.474	(0.74)	1.470	(0.73)	1.484	(0.45)
(B) $t = 4$	1.640	(0.89)	1.762	(0.89)	1.761	(0.87)	2.141	(0.65)
(C) $t = 4$	1.840	(1)	1.989	(1)	2.013	(1)	3.289	(1)
$t = 8$	2.120	(1.15)	2.242	(1.13)	2.313	(1.15)	4.080	(1.24)

表6 実質走査速度(ビット/マイクロ秒)
Table 6 Effective running speed for hit rate = 0 (bits/μsec).

pmm の実現法	テキスト 1	テキスト 2	テキスト 3	テキスト 4
(A) $t = 4$	5.597	5.310	5.108	3.089
$t = 8$	7.057	6.598	6.372	3.862
(B) $t = 4$	7.879	7.883	7.630	5.572
(C) $t = 4$	8.839	8.899	8.725	8.559
$t = 8$	10.184	10.032	10.025	10.617

の実質走査速度は，$t = 4$, 8 のいずれの場合も，テキストによらずほぼ同じ値である．これに対し，(A)，(B) の pmm では，テキストによって値がばらついている．特にテキスト 4 では，(A)，(B) の pmm の実質走査速度は，他のテキストと比べ大きく低下している．これは，(C) のまとめ読み pmm では，圧縮効率がつねに高いためにそのテキストが有する実質的な情報だけを処理しているのに対して，他の方法では圧縮効率が低い場合には冗長な情報までも処理しているからである．すなわち，本稿で提案したまとめ読み pmm の手法は，Huffman 符号による圧縮の効果を十分に引き出していることが分かる．

6. おわりに

本稿では，テキストの圧縮によるパターン照合の高速化を目的とし，Huffman 符号で圧縮したテキストを復号せずに走査する pmm の高速な実現法として，複数ビット分の状態遷移を 1 回の状態遷移に置き換えたまとめ読み pmm を提案した．この方式の有効性を検証するため，pmm の構成時間，領域量，および走査時間について，他の pmm の実現法との比較実験を行った．その結果，4 ビットまとめ読み pmm は，いずれの面からも十分実用的であることが示された．また，走査速度に関してテキストに依存しない指標を与えるために，単位時間あたりに処理した情報量である実質走査速度を導入した．まとめ読み pmm は，他の実現法に比べ実質走査速度の値が高く，その値はテキストに依存せずほぼ一定であることが分かった．すなわち，本稿で提案したこの方式は，Huffman 符号による圧縮の効果を十分に引き出して高速化できる．

走査時間をさらに短縮するためには，Markov モデルを用いた符号化により，英文テキストの圧縮率を向上させることが考えられる．Markov モデルのより一般的な形である有限状態モデルによって圧縮したテキストに対する pmm の構成アルゴリズムはすでに開発しており，その正当性の証明も与えている[17]．ただし，モデルの状態数が多くなると pmm のサイズが膨大になるため，状態数の小さい有限状態モデルでモデル化する必要がある．現在のところ，たとえば状態数 16 の有限状態モデルにより英文テキストを約 46%に圧縮でき，本稿で提案した 4 ビットまとめ読みの手法を適用することで走査時間を 47%に短縮できることが分かっている[18]が，これについては稿を改めて述べることにする．

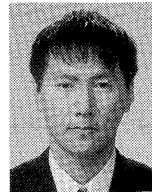
参考文献

1) Knuth, D.E., Morris, J.H. and Pratt, V.R.: Fast pattern matching in strings, *SIAM J. Comput*, Vol.6, No.2, pp.323–350 (1977).
2) Boyer, R.S. and Moore, J.S.: A fast string searching algorithm, *Comm. ACM*, Vol.20, No.10, pp.62–72 (1977).
3) 深町修一，篠原 武，竹田正幸：可変長符号圧縮データのための文字列パターン照合—ゲノム情報の高速検索技法，1992 年情報学シンポジウム講演論文集，pp.95–103 (1992).
4) Manber, U.: A text compression scheme that allows fast searching directly in the compressed file, *Proc. Combinatorial Pattern Matching*, Lecture Notes in Computer Science, Vol.807, pp.113–124, Springer-Verlag (1994).
5) 松本光崇，角田達彦，松本裕治：圧縮ファイル

への直接検索を可能にする符号化法の考案，情報処理学会研究報告，Vol.96, No.25, pp.41–48 (1996).

6) Ziv, J. and Lempel, A.: A Universal Algorithm for Sequential Data Compression, *IEEE Trans. Inform. Theory*, Vol.IT-23, No.3, pp.337–349 (1977).
7) Welch, T.A.: A Technique for High Performance Data Compression, *IEEE Comput.*, Vol.17, pp.8–19 (1984).
8) Amir, A., Benson, G. and Farach, M.: Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files, *Journal of Computer and System Sciences*, Vol.52, pp.299–307 (1996).
9) Gąsieniec, L., Karpinski, M., Plandowski, W. and Rytter, W.: Efficient algorithms for Lempel-Ziv Encoding, *Proc. 4th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, Vol.1097, pp.392–403, Springer-Verlag (1996).
10) Gąsieniec, L., Karpinski, M., Plandowski, W. and Rytter, W.: Randomized Efficient Algorithms for Compressed Strings: the Finger-Print Approach, *Proc. Combinatorial Pattern Matching*, Lecture Notes in Computer Science, Vol.1075, pp.39–49, Springer-Verlag (1996).
11) Farach, M. and Thorup, M.: String-matching in Lempel-Ziv compressed strings, *27th ACM STOC*, pp.703–713 (1995).
12) Kida, T., Takeda, M., Shinohara, A., Miyazaki, M. and Arikawa, S.: Multiple Pattern Matching in LZW Compressed Texts, *Proc. 8th Data Compression Conference*, pp.103–112, IEEE Computer Society (1998).
13) Aho, A.V. and Corasick, M.: Efficient string matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol.18, No.6, pp.333–340 (1975).
14) Knuth, D.E.: *The Art of Computer Programing, Sorting and Searching*, Vol.3, Addison-Wesley (1973).
15) Arikawa, S. and Shinohara, T.: A run-time efficient realization of Aho-Corasick pattern matching machines, *New Generation Computing*, Vol.2, pp.171–186 (1984).
16) 平田博明：パターン照合機械のためのハフマン木の縮退，九州工業大学情報工学部知能情報工学科卒業論文 (1995).
17) Takeda, M.: Pattern matching machine for text compressed using finite state model, Technical Report DOI-TR-CS-142, Department of Informatics, Kyushu University (1997).
18) 宮崎哲司：生成確率モデルに基づくデータ圧縮による文字列パターン照合の高速化に関する研究，修士論文，九州工業大学大学院情報工学研究科情報科学専攻 (1996).

(平成 9 年11月5日受付)
(平成10年 7 月3日採録)

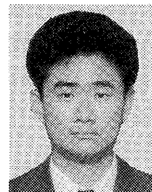


宮崎　正路（正会員）

1973 年生．1996 年九州工業大学情報工学部知能情報工学科卒業．1998 年九州大学大学院システム情報科学研究科情報理学専攻修士課程修了．同年日本電気（株）に入社．在学中，パターン照合アルゴリズムの研究に従事．

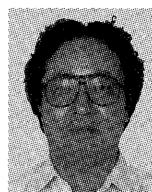
深町　修一（正会員）

1967 年生．1991 年九州工業大学情報工学部知能情報工学科卒業．1993 年同大学大学院情報工学研究科情報科学専攻修士課程修了．1995 年同専攻博士課程後期中退，同年より同大学情報工学部知能情報工学科助手，現在に至る．パターン照合アルゴリズム，情報検索などに興味を持つ．人工知能学会会員．



竹田　正幸（正会員）

1964 年生．1987 年九州大学理学部数学科卒業．1989 年同大学大学院総合理工学研究科情報システム学専攻修士課程修了，同年より同大学工学部電気工学科助手．1996 年より同大学大学院システム情報科学研究科情報理学専攻助教授，現在に至る．博士（工学）．パターン照合アルゴリズム，テキストデータマイニング，自然言語処理，情報検索などに興味を持つ．人工知能学会，日本ソフトウェア科学会各会員．



篠原　　武（正会員）

1955 年生．1980 年京都大学理学部数学科卒業．1982 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了，同年より同大学大型計算機センター助手．1987 年より九州工業大学情報工学部知能情報工学科助教授．1994 年より同学科教授，現在に至る．理学博士．帰納推論，PAC 学習，機械発見，情報検索，文字列パターン照合などに興味を持つ．人工知能学会会員．