View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

東北大学機関リポジトリ
TOUR
Tohoku University Repository

エントロピー圧縮した文字列に対する検索アルゴリズムに関する研究

著者	柳町　修平
学位授与機関	Tohoku University
URL	http://hdl.handle.net/10097/39906

修士学位論文

エントロピー圧縮した文字列に対する検索アルゴリズムに関する研究

東北大学大学院 情報科学研究科

システム情報科学専攻 篠原研究室

博士課程前期二年

柳町 修平

2009年2月17日

目次

第1章　序論　1
1.1　研究の背景 . . . 1
1.2　本論文の構成 . . . 3
第2章　エントロピー符号化　4
2.1　Huffman 圧縮 . . . 4
2.1.1　圧縮と展開 . . . 4
2.2　KZ 圧縮 . . . 5
2.2.1　Fibonacci 数列 . . . 6
2.2.2　Zeckendorf の定理 . . . 6
2.2.3　圧縮と展開 . . . 7
2.3　圧縮率 . . . 8
2.3.1　経験エントロピー . . . 8
2.3.2　Shannon の第1符号化定理 . . . 8
2.3.3　Huffman 圧縮の圧縮率 . . . 9
2.3.4　KZ 符号の瞬時復号可能化 . . . 9
第3章　全文検索アルゴリズム　11
3.1　Aho-Corasick 法 . . . 11
3.1.1　AC オートマトン . . . 12
3.1.2　AC オートマトンの構築 . . . 14
3.1.3　AC オートマトンを用いた全文検索 . . . 15
3.1.4　AC オートマトンの決定性有限オートマトンへの等価変換 . . . 16
3.2　AC 法に基づく圧縮文字列検索 . . . 17

3.2.1 Huff-AC 法 18
3.2.2 KZ-AC 法 21
3.2.3 両手法の比較 21
3.2.4 t ビットまとめ読み法 22
3.3 各手法の走査に必要な計算量の比較 22
3.3.1 時間計算量 23
3.3.2 領域計算量 23

第 4 章 DC パターンを含むパターン集合に対する全文検索アルゴリズム 25

4.1 定義 25
4.2 DC パターンを含むパターン集合に対する全文検索 25
4.2.1 DC パターンの展開 26
4.2.2 アクティブなノード 27
4.2.3 AC 法を用いた DC パターンを含むパターン集合に対する全文検索の計算量 27
4.2.4 Huff-AC 法及び KZ-AC 法を用いた DC パターンを含むパターン集合に対する全文検索 31
4.3 提案手法 31
4.3.1 $\Sigma_{P,\ i}$と$\#_{P,\ i}$の導入 31
4.3.2 DC パターンの展開の定義の拡張 31
4.3.3 AC 法における $\#_i$ の実現手法 34
4.3.4 提案手法を実現するために要するノード数 36
4.3.5 提案手法の Huff-AC への応用 39
4.4 KZ-AC 法に特化した提案手法 40

第 5 章 実験 43

5.1 DC 文字の出現位置 44
5.1.1 DC 文字が 1 出現である場合の DC 文字の出現位置がノード数に与える影響 45

5.1.2 DC 文字が 2 出現である場合の DC 文字の出現位置がノード数に与える影響 48
5.2 パターン数が AC オートマトンのノード数に与える影響 50
5.3 エントロピー圧縮したパターンに対する検索オートマトンのノード数 . . 52
5.4 実験に対する考察 56

第 6 章　結論 **57**

参考文献 **59**

第1章

序論

1.1 研究の背景

テキスト文字列中から目的のパターンを探す文字列検索問題は，文字列に対する最も基本的な問題の一つである．この問題は，DNA配列の特定パターンの検出のようなバイオインフォマティクスの分野や，インターネット上での情報検索など幅広い応用範囲を持つ．これまでに，Knuth-Morris-Pratt法 [13] やBoyer-Moore法 [3] のように効率的に文字列検索を行うことが出来るアルゴリズムが数多く提案されている．これらの手法は，パターンとテキストをそれぞれ文字列と見なし，文字を逐次的に比較することによりパターンを見つけるパターン照合に基づく手法であり，本研究による提案手法もパターン照合に基づく．

一方，LarssonSadakane法 [14] やFM-Index [7] などに代表される，全文検索のために索引構造をあらかじめ用意する手法も存在する．索引を用いた全文検索は一般にパターン照合に基づく手法よりも高速に検索を行うことができるが，パターン照合では必要のなかった索引構造を構築するための計算時間や，索引構造のための領域が新たに必要となる．

ところで，文字列検索をする際に目的のパターンと一部が異なるパターンもまた有用なパターンである場合が多い．例えば，コンピュータウイルスの検出は文字列検索技術を応用している．あるウイルスが公開されると，その一部を改変したウイルスの亜種も多数出現することが多く，全ての亜種に対応した検出プログラムを作る必要がある．このように，目的のパターンだけではなく，目的のパターンとともにそれと似通ったパター

ンも同時に検索することは，難易度が高い問題ではあるものの，単純な文字列照合よりもはるかに応用範囲が広く，非常に有用である．このような問題を近似文字列照合問題と呼ぶ.

本論文では，近似文字列照合問題の一種として，パターン中に，どんな文字でも一致したと見なすようなメタ文字の出現を許して全文検索するような問題を扱っている．このようなメタ文字をDC文字といい，DC文字を含むパターンをDCパターンという.

一般に文字列検索問題の計算時間はテキスト長に依存している．これはテキストの全ての文字を少なくとも1度は参照しなくてはいけないからである．そのため，アルゴリズムの改良には限界があった．そこで，入力するテキストをあらかじめ圧縮し，展開することなく検索アルゴリズムに入力することにより，検索アルゴリズムへの入力テキスト長を短くして高速化を図る方式が提案されている．このような手法を圧縮文字列検索と呼ぶ.

これまでの圧縮アルゴリズムは主に圧縮率や圧縮・展開速度の向上を主な目的として研究されており，LZ77法 [20] やLZW法 [19] などが優れたアルゴリズムであるとして広く用いられている．しかし，圧縮文字列検索に適した圧縮アルゴリズムは，これらの評価基準に加え，文字列照合の高速化が出来る事が重要である．KZ77法やLZW法に対する圧縮文字列検索アルゴリズム [2] [6] の研究は圧縮された文字列から情報を高速に取り出すことを目的としてなされており，テキストを圧縮することによって文字列検索自体の高速化を目指しているわけではない.

それに対し，深町ら [8] や宮崎ら [15] によって提案された，静的Huffman圧縮した文字列に対する全文検索アルゴリズムは，圧縮文字列検索による文字列検索の高速化を目的としている．特に宮崎らは，圧縮せずに文字列検索を行った場合と比べて文字列検索の高速化ができたことを実験的に示した.

宮崎らはDCパターンを含まないパターン集合に対する全文検索アルゴリズムの提案をしているが，本論文ではこのアルゴリズムを拡張し，DCパターンを含むパターン集合に対しても効率的に処理する文字列検索アルゴリズムを提案する．また，KZ圧縮 [10] を用いた，アルファベットサイズの影響を受けにくいDCパターンを含むパターン集合に対する圧縮文字列検索アルゴリズムを提案する.

1.2 本論文の構成

本論文は 6 章からなり，以下にその構成を示す.

第 1 章は序論であり，本研究の背景及び目的について述べる．第 2 章は本研究で提案したアルゴリズムに採用したエントロピー圧縮方式の紹介をする．第 3 章では基本的な全文検索アルゴリズムをいくつか紹介し，改良したアルゴリズムを第 4 章で提案する．本研究では，提案手法のノード数削減の効果を調べるために実験を行っており，実験結果及びその考察を第 5 章でおこなっている．第 6 章は研究のまとめである．

第2章

エントロピー符号化

計算機上で情報を扱う場合，各情報にビットパターンを対応させる．この操作を符号化という．通常，文字の符号化には固定長のビットパターンが用いられる．しかしテキスト中の文字の出現頻度に偏りがある場合，出現頻度が高い文字に短いビットパターンを割り当て，反対に出現頻度が低い文字に長いビットパターンを割り当てることによってテキスト全体のビット長を少なくできる場合がある．このように出現頻度の偏りを利用して可変長符号を用い，情報圧縮することをエントロピー圧縮という．

本論文ではエントロピー圧縮したテキストに対し全文検索することを目的としており，本章では研究で用いた2つの圧縮方式について述べる．

2.1 Huffman圧縮

本節ではHuffmanが1952年に提案したエントロピー圧縮の1手法であるHuffman圧縮 [9] について述べる．

2.1.1 圧縮と展開

符号語を表した二分木を符号木という．また，Huffman圧縮に用いられる符号木を特にHuffman木と呼ぶ．Huffman圧縮は文字の出現頻度を調べた後，Huffman木を構築することにより実現できる．

Huffman木は以下のように構築する．

1. 各文字に対応する葉を作成する．

2. 出現頻度の低い２つの文字に対応する葉を取り出す.

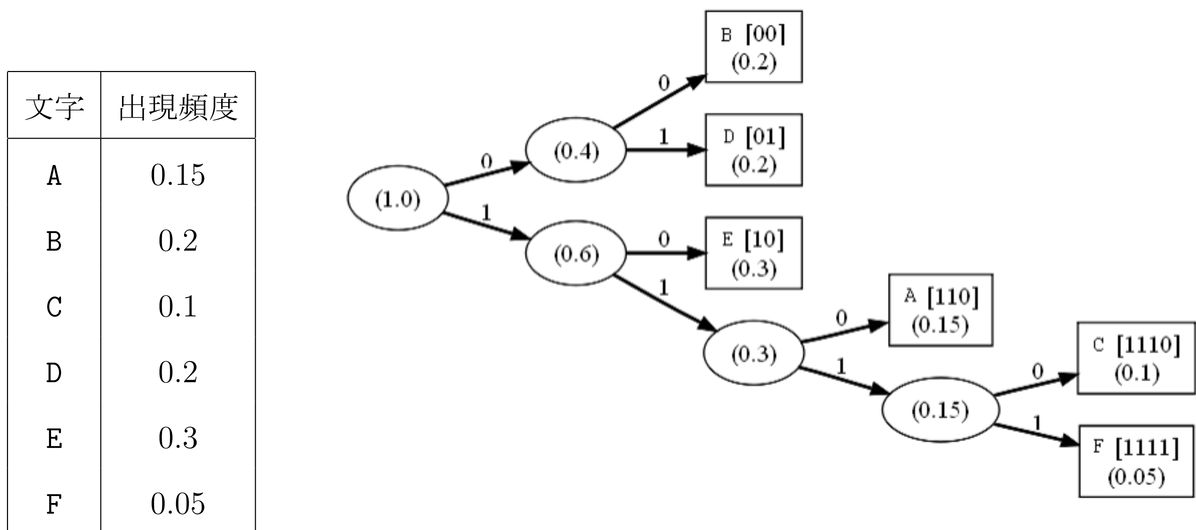
3. この２つの葉を子に持つ新たな節を作り加える．ただし，この節の出現頻度は２つの葉の出現頻度の和とする.

4. 残った葉と新たに作った節に対し，3 の操作を繰り返す.

   新たな節を作ることができなくなったら終了.

このように構築した木は二分木である．左右の枝にビット 0, 1 をそれぞれ対応させると，根から文字に対応する葉まで枝を辿ったときに得られる 0, 1 のビットパターンが圧縮後のその文字の符号である．圧縮データにはこうして得られたビットパターンと，復号のための Huffman 木が含まれる.

表 2.1 のような出現率の文字列の場合，Huffman 木は図 2.1 の通りになる.



文字	出現頻度
A	0.15
B	0.2
C	0.1
D	0.2
E	0.3
F	0.05

表 2.1: 文字の出現率

図 2.1: 表 2.1 に対応する Huffman 木

展開のためには，Huffman 木の根から圧縮されたビット列の先頭ビットから順に木を辿り，葉のところでそれに対応する文字を出力することを繰り返せばよい.

2.2 KZ 圧縮

本節では Kautz が 1964 年に提案した Kautz-Zeckendorf 圧縮 (KZ 圧縮) [10] について述べる.

KZ 圧縮は Fibonacci 数列の性質に基づくエントロピー符号化の 1 方式である．圧縮率は Huffman 圧縮に及ばないものの，符号語の先頭が必ず 110 から始まるという性質を持つ．

2.2.1 Fibonacci 数列

Fibonacci 数 [11] は以下の漸化式で定義される．

$$
\begin{aligned}
F_1 &= 1 \\
F_2 &= 1 \\
F_{i+2} &= F_i + F_{i+1} \qquad (i > 0)
\end{aligned}
$$

この漸化式を満たす数列 1, 1, 2, 3, 5, 8, 13, $\cdots$ が Fibonacci 数列である．

2.2.2 Zeckendorf の定理

Fibonacci 数列の性質を示す定理の 1 つに Zeckendorf の定理がある．

定理（Zeckendorf） [10] 全ての自然数は，隣り合わない Fibonacci 数の和で書き表すことが出来る．

証明

1. $i = 1, 2, 3$ の時はそれ自身が Fibonacci 数であるため，定理を満たす．
2. $j \geq 4$ を満たすある j 以下の全ての正の整数が，隣り合わない Fibonacci 数の和で表すことが出来ると仮定する．
   (a) $j+1$ が Fibonacci 数であれば $j+1$ は定理を満たす．
   (b) $j+1$ が Fibonacci 数でない場合，$F_{k_0} < j+1 < F_{k_0+1}$ なる k_0 が存在する．
   $0 < j+1-F_{k_0} < F_{k_0+1} - F_{k_0}$ となるが，Fibonacci 数を求める漸化式から $F_{k_0+1} - F_{k_0} = F_{k_0-1}$ なので，$j+1-F_{k_0} < F_{k_0-1}$ となる．$j+1-F_{k_0} < j+1$ なので，$j+1-F_{k_0}$ は隣り合わない Fibonacci 数の和で書き表すことが出来る．$j+1-F_{k_0} < F_{k_0-1}$ から，$j+1-F_{k_0}$ を表すために F_{k_0-1} は用いない．$j+1 = (j+1-F_{k_0}) + F_{k_0}$ と考えると，$j+1$ は定理を満たす．

よって数学的帰納法により Zeckendorf の定理は成り立つ．

2.2.3 圧縮と展開

まず，圧縮の方法について述べる．

Huffman 符号と同様に出現頻度を解析し，出現頻度の高い文字から順に 2 以上の自然数を割り当てる．2 以上の自然数を第 2 項以降の Fibonacci 数列の和で表し，和に用いる項の桁を 1，用いない項の桁を 0 としてバイナリ表現する．この時，自然数に対して項の選び方が一意に定まらない場合もあるが，Zeckendorf の定理を用いるためになるべく大きい項を選ぶようにする．このビット列の先頭に 1 を連結して，KZ 符号の符号語とする．

例えば文字 C が全体の 5 番目に頻出する文字であった場合，対応する自然数は 6 である．6 を Fibonacci 数列 1, 2, 3, 5, ··· の和で表す場合，$1+5$ と $1+2+3$ の 2 通りが考えられるが，Zeckendorf の定理を利用するため $1+5$ で表すことにする．これに対応する 2 進数は 1001 であり，このバイナリ列の先頭に 1 をつける．従って，文字 C の KZ 符号の符号語は 11001 である．

文字の出現頻度が表 2.1 の通りである場合，KZ 符号の符号語は表 2.2 の通りである．

文字	出現頻度	対応する自然数	符号語
A	0.15	5	11000
B	0.2	3	1100
C	0.1	6	11001
D	0.2	4	1101
E	0.3	2	110
F	0.05	7	11010

表 2.2: KZ 符号の符号語

Zeckendorf の定理から符号語の先頭は必ず 110 から始まり，1 が符号語の途中で連続して出ることがない．よって 110 が出現した場所が必ず符号語の先頭である．

Huffman 符号は，展開のために Huffman 木を保存する必要があったが，KZ 符号は文字の出現頻度の順のみ保存しておくことで，展開は容易に行うことが出来る．

2.3 圧縮率

本節ではエントロピー符号化の圧縮率について述べる．まず，符号語の平均符号長と密接な関係のある経験エントロピーを導入する．ついで符号語の平均符号長の下限を与えるShannonの第1符号化定理について述べ，さらにHuffman符号の圧縮率について考察する．

2.3.1 経験エントロピー

エントロピーは乱雑さを表す値としてShannonによって提唱され，次式で表される．

$$H(\delta) = -\sum_{c\in\Omega} \delta(c) \log \delta(c)$$

$\delta(c)$は事象cの発生確率であるため，この式は$c \in \Omega$なる事象cの選択情報量$-\log \delta(c)$の期待値である．

特に長さnの文字列Tに対し，$c \in \Sigma$なる事象について考えたエントロピーを0次経験エントロピーと呼ぶ．n_cをT中のcの出現数だとすると$\delta(c) = \frac{n_c}{n}$であることから，文字列$T$に対する0次経験エントロピーは次式のとおりである．

$$H(T) = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

本論文では以降，0次経験エントロピーを経験エントロピーまたはエントロピーと表すこととする．

2.3.2 Shannonの第1符号化定理

ある符号を受け取った時，その符号語を受け取った時点で語の終わりであることが分かり，復号可能である物を瞬時復号可能な符号であるという．

Shannonの第1符号化定理は瞬時復号可能な符号の符号長の限界を示した定理である．

定理（Shannonの第1符号化定理）[17] バイナリエンコーディングをする場合，平均符号長Lを$H(T)$より短くすることはできない．つまり，Lと$H(T)$は次式の関係を満たす．

$$H(T) \leq L$$

2.3.3 Huffman 圧縮の圧縮率

Huffman 符号は最短符号であることが知られている．最短符号とは，同じ文字列に対する平均符号長が，他のいかなる符号化のルールを用いたときの平均符号長と比較しても長くないことを意味する．Huffman 符号が最短符号であるのは，以下の理由による．

1. Huffman 木の構築アルゴリズムの中では，出現頻度の最も少ない 2 つの葉・節点を 1 つにまとめることにより，比較すべきノードの数を 1 つずつ減らしている．この操作を i 回繰り返した時，次に比較すべきノードの集合を U_i とおく．U_i を表現するための符号語を R_i とし，この符号語の平均符号長を L_i とする．R_{i-1} に含まれる出現頻度の最も少ない 2 つの葉・節点の出現確率をそれぞれ $\delta_{\alpha 0}$，$\delta_{\alpha 0}$ とすると，各操作でこのような節点に 0，1 ビットを割り当てていることから $L_{i-1} = L_i + \delta_{\alpha 0} + \delta_{\alpha 1}$ の関係が明らかに成り立つ．

2. R_i が U_i に対する最短符号であるとき，$\hat{L}_{i-1} < L_{i-1}$ なる平均符号長 $\hat{L}_{i-1}$ である符号語 $\hat{R}_{i-1}$ が U_{i-1} を表現するための符号語の集合として存在すると仮定する．$\hat{R}_{i-1}$ 中の最も出現頻度の小さい 2 つの符号語をまとめて $\hat{R}_i$ を作るとき，$\hat{R}_i$ の平均符号長を $\hat{L}_i$ と表すと，$\hat{L}_{i-1} = \hat{L}_i + \delta_{\alpha 0} + \delta_{\alpha 1}$ である．

   $\hat{L}_{i-1} < L_{i-1}$ であるためには $\hat{L}_i < L_i$ でなければならないが，仮定から R_i は U_i に対する最短符号であるため，$\hat{R}_i$ は存在しない．したがって背理法より R_i が U_i に対する最短符号であるとき，R_{i-1} は U_{i-1} に対する最短符号である事が証明できた．

このように，Huffman 符号は最短符号である．

2.3.4 KZ 符号の瞬時復号可能化

Kautz が提案した KZ 符号は 110 が符号語の先頭を示しているので一意に復号することは可能ではあるが，次の符号語の先頭の 110 を見つけるまで符号の終端が分からないため，瞬時復号可能ではない．これは KZ 符号が接頭符号ではないからである．

しかし，KZ 符号の各ビットを後ろから前に並び替えることで，ビットパターン 011 が出現した時点で復号可能である瞬時復号可能な符号となる．表 2.3 は表 2.2 の各符号語に対し，このような変換を施した符号語である．

文字	出現頻度	対応する自然数	符号語	変換後の符号語
A	0.15	5	11000	00011
B	0.2	3	1100	0011
C	0.1	6	11001	10011
D	0.2	4	1101	1011
E	0.3	2	110	011
F	0.05	7	11010	01011

表 2.3: 瞬時復号可能化した KZ 符号

2.3.3 節で述べたとおり Huffman 符号は最短符号なので，瞬時復号可能化した KZ 符号は Huffman 符号よりも平均符号長は長いといえる．KZ 符号を瞬時復号可能化する前後で符号語の長さは変わらないので，KZ 符号の平均符号長も Huffman 符号より長いと言える．

Canterbury コーパス [4] 及び，タンパク質を構成するアミノ酸の配列データ [5] より 3 つのテキストを選び，圧縮率の実験を行った．

表 2.4 は，これらのテキストの Huffman 圧縮と KZ 圧縮の圧縮率を示した表である．ただし，圧縮率は圧縮される前のデータサイズと圧縮後のデータサイズの比である．

テキスト	サイズ [KB]	アルファベットサイズ	エントロピー	平均符号長 [ビット/文字]		圧縮率 [%]	
				Huffman	KZ	Huffman	KZ
1	148	72	4.513	4.555	5.397	56.94	67.47
2	471	79	4.477	4.520	5.413	56.49	67.67
3	25	84	5.229	5.268	6.296	65.84	78.70
4	157	21	4.249	4.322	5.619	52.93	68.82

表 2.4: Huffman 符号と KZ 符号の圧縮率

この結果からも，Huffman 圧縮のほうが KZ 圧縮に比べ圧縮率が高いと言える．

第3章

全文検索アルゴリズム

アルファベット Σ 上の与えられたテキスト T 中からパターンとして与えられた文字列と完全に一致する部分を探し出すことを全文検索という．テキストとパターンを逐次的に比較することにより全文検索することを特に文字列照合と呼ぶ．本章では，文字列照合に基づく全文検索のためのアルゴリズムについて説明する．

3.1 Aho-Corasick 法

本研究では目的のパターン集合 P に含まれる複数のパターンを全文検索することを目的としている．Σ 上の長さ n のテキスト T 中から複数のパターンを検索する場合，それぞれのパターンについて独立に全文検索を実行することは非効率である．そのため提案アルゴリズムは 1975 年に Aho と Corasick が提案した Aho-Corasick 法 (AC 法) [1] をベースにしている．AC 法は 1 度のテキストの走査で複数パターンを同時に検索できるアルゴリズムである．

AC 法では P から検索のための ϵ 遷移付き有限オートマトンを構築する．検索のためのオートマトンを検索オートマトンという．AC 法によって生成される検索オートマトンを AC オートマトンと呼ぶ．AC オートマトンに T を入力することにより複数パターンに対する全文検索を実行することができる．AC 法は P に含まれるパターンの数によらずただ 1 回の走査で各パターンの出現位置をそれぞれ求めることができるという点で優れている．

3.1.1 AC オートマトン

AC オートマトンは $(S,\ \Sigma,\ s_0,\ goto,\ failure,\ A,\ output)$ の七つ組で定義できる．各要素は次の通りである．

- 状態の有限集合 S
- 入力アルファベット Σ
- 初期状態 $s_0 \in S$
- goto 関数 $goto\colon S \times \Sigma \to S \cup \{fail\}$
- failure 関数 $failure\colon S \to S$
- 受理状態の集合 $A \subseteq S$
- output 関数 $output\colon A \to 2^{\Sigma^+}$

goto 関数 $goto(s,\ char)$ は各状態において文字が入力されたときに，次の状態遷移先を示す．ただし，全ての文字について必ずしも状態遷移先が定義されておらず，未定義の場合は goto 関数は $fail$ を返す．goto 関数が $fail$ を返すことをミスマッチが起きると呼ぶ．failure 関数 $failure(state)$ はミスマッチが起きた時に呼ばれ，次の状態遷移先を示す．ただし，この時の遷移は ϵ 遷移である．output 関数 $output(state)$ は各受理状態で発見されるパターンの集合を返す．図 3.1 はパターン集合 $P = \{\texttt{ABC},\ \texttt{B}\}$ に対して構築した AC オートマトンである．黒の実線は goto 関数を表しており，赤の破線は failure 関数を表している．2 重丸で表したノードでは output 関数が設定されており，{} で囲ったパターンが発見できたことを示している．

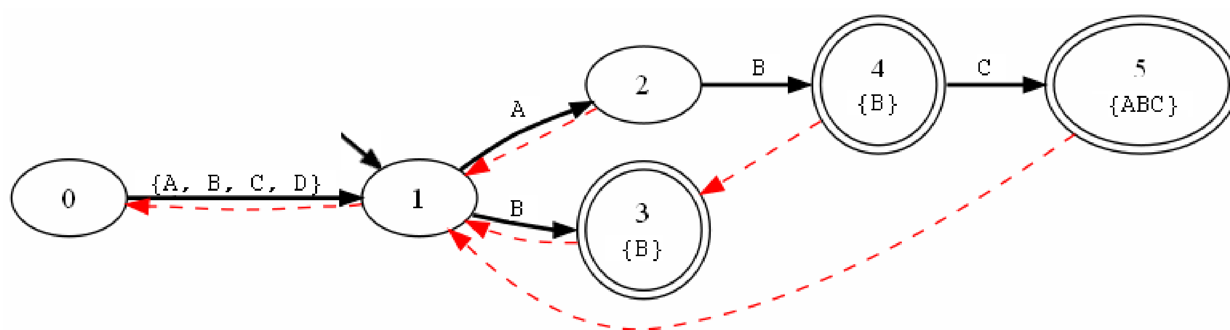


図 3.1: $P = \{\texttt{ABC}, \texttt{B}\}$ に対する AC オートマトン

3.1.2 AC オートマトンの構築

AC オートマトンを構築するアルゴリズムについて述べる．

まず，パターン集合から goto 関数と output 関数を計算し，goto 関数と output 関数のみからなるオートマトンを構築する．これが Algorithm 1 である．次に，このオートマトンの各ノードについて failure 関数を計算する．AC 法において failure 関数は，照合中のパターンの接尾辞と一致する，パターン集合 P に含まれるパターンの最長の接頭辞を求める関数である．failure 関数で遷移した先のノードで output 関数が定義されている場合，そこで発見されたパターンは failure 関数で遷移する前のノードの接尾辞でもあるため，このパターンを遷移元のノードの output 関数に加える．これが Algorithm 2 である．

AC オートマトンを構築するアルゴリズムは Algorithm 1 と Algorithm 2 を，Algorithm 3 のように組み合わせることによって実現できる．

Algorithm 1: goto 関数を計算するアルゴリズム

```
for i ← 0 to n − 1 do
    p ← p_i
    s ← 1
    for j ← 0 to |p| − 1 do
        if g(s, p[j]) == fail then
            newState ← newState + 1
            goto(s, p[j]) ← newState
            parent(newState) ← s
            s ← newState
        else
            s ← goto(s, p[j])
        end
    end
    output(s) ← p
end
```

Algorithm 2: failure 関数を計算するアルゴリズム

```
failure(1) ← 0
States ← {1}
while States ≠ φ do
    queue ← φ
    for all s in States do
        for all c in Σ do
            if goto(s, c) ≠ fail then
                queue ← queue ∪ goto(s, c)
            end
        end
    end
    for all s in queue do
        u ← failure(parent(s))
        while goto(u, char) == fail do
            u ← failure(parent(u))
        end
        failure(s) ← u
    end
    States ← queue
end
```

Algorithm 3: AC オートマトン構築アルゴリズム

Input : パターン集合 $P = \{p_0,\ p_1,\ \cdots,\ p_{n-1}\}$

Output : AC オートマトン

```
newState ← 1
for all c in Σ do
    goto(0, c) ← 1
end
Algorithm 1
Algorithm 2
```

3.1.3 AC オートマトンを用いた全文検索

Algorithm 4 は AC 法の擬似コードである.

Algorithm 4: AC 法による文字列照合アルゴリズム

Input : テキスト T, AC オートマトン

Output : パターン集合に含まれる各パターンの出現位置

```
s ← 1
for i ← 0 to n − 1 do
    while goto(s, T[i]) == fail do
        s ← failure(s)
    end
    s ← goto(s, T[i])
    if s ∈ A then
        report i, output(s)
    end
end
```

3.1.4 AC オートマトンの決定性有限オートマトンへの等価変換

AC オートマトン中には failure 関数として ϵ 遷移が含まれている．そのためテキストを 1 文字を読んだ際に複数回状態遷移するケースが考えられる．そこで高速化のために [1] で提案されている通り，AC オートマトンの構築後，決定性有限オートマトン（DFA）に変換する．DFA 化することで，1 文字の入力に対し 1 状態遷移するだけで検索を実現できるようになるため，計算時間の短縮が期待できる．

Algorithm 5 は AC オートマトンを DFA に変換するアルゴリズムである．

Algorithm 5: AC オートマトンから DFA への変換アルゴリズム

Input : AC オートマトン

Output : DFA

```
for all s in S do
    for all c in Σ do
        u ← s
        while goto(u, c) == fail do
            u ← failure(u)
        end
        goto(s, c) ← goto(u, c)
    end
end
```

図 3.2 は，Algorithm 5 を用いて図 3.1 の AC オートマトンを変換し生成した DFA である．図 3.2 中の緑矢印はこのアルゴリズムによって新たに生成された goto 関数を示している．説明のため failure 関数も併せて図中に示しているが，DFA 化した後であれば failure 関数を保持する必要はない．

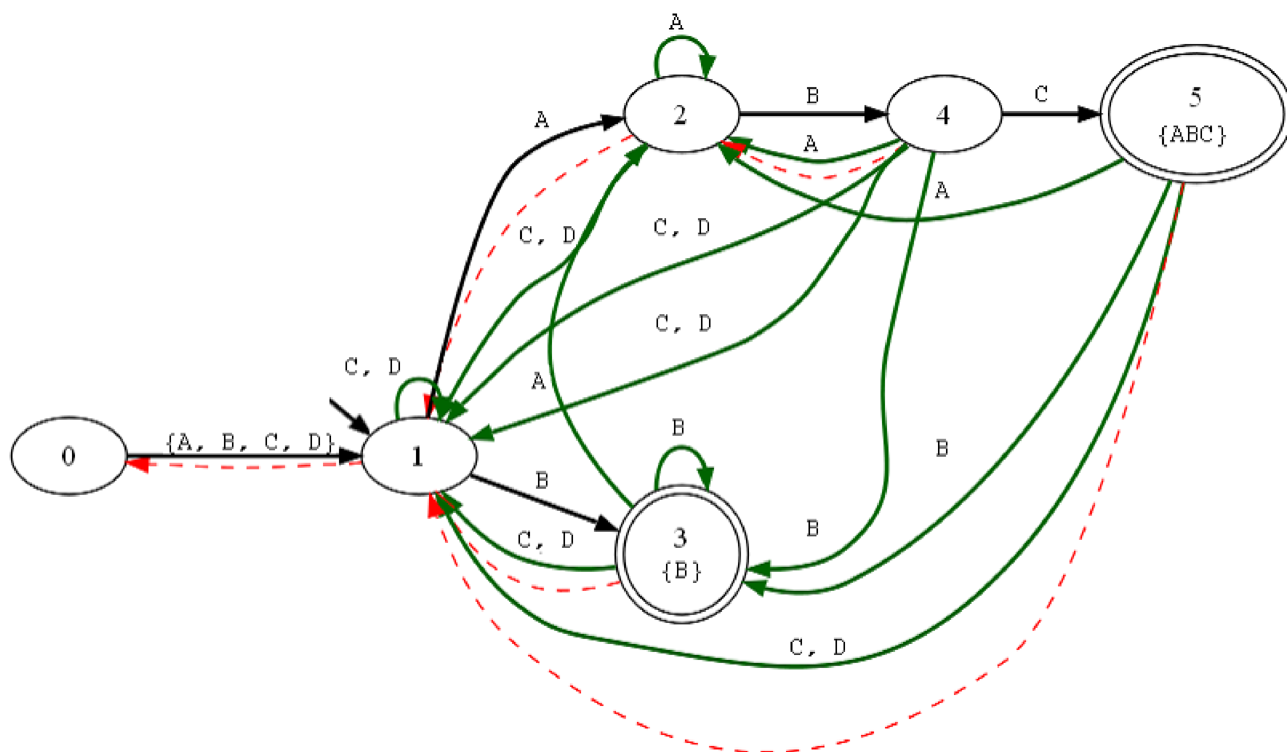


図 3.2: 図 3.1 の AC オートマトンを変換して生成した DFA

生成した goto 関数は表一瞥 (table-look-at) 法 [12] により実現する．

表一瞥法では，goto 関数は状態を表す整数と文字コードを引数とする 2 次元配列で実現している．これにより，次状態を表を 1 度参照するだけで決定することが出来る．

3.2 AC 法に基づく圧縮文字列検索

検索したいテキスト T の長さ n に対し，AC 法を用いた検索時間は $O(n)$ であり，テキスト長に依存している．そのため検索時間の短縮のためには圧縮文字列検索が有効である．

本節では，AC 法をベースとしエントロピー圧縮した文字列を全文検索するアルゴリズムを紹介・提案する．

3.2.1 Huff-AC 法

文字列及びパターンを Huffman 圧縮してから AC オートマトンを構築する方法が [8] によって提案されている．本論文ではこの手法を Huff-AC 法と呼ぶことにする．

Huff-AC 法では符号のずれ読みを防ぐ目的で，パターン木に Huffman 木を付加している．Huffman 木を付加せず AC オートマトンを構築した場合，AC オートマトンの性質上，符号語の途中でミスマッチが起こる場合が考えられ，その結果，誤検出が発生しうる AC オートマトンが生成される．例として，図 2.1 の Huffman 木を用い，パターン集合 $P = \{\mathtt{AE}, \mathtt{EB}\}$ に対して Huff-AC 検索オートマトンを構築する場合を考える．この時パターン AE，EB は Huffman 符号でそれぞれ 11010，1000 のように表される．このビット列について構築した AC オートマトンが図 3.3(a) である．このオートマトンを用いて $T = \mathtt{FDB}$ を検索しようとすると，以下ように誤検出が起きる．

FDB を Huffman 圧縮すると，バイナリ列 11110100 が得られる．図 3.3(b) は，図 3.3(a) のオートマトンを用いてこのバイナリ列を検索しようとしたときの状態遷移を示した図である．入力テキストから得られた 11110100 の部分文字列 11010 を，パターン AE が見つかったものとして誤認識していることがこの図から分かる．

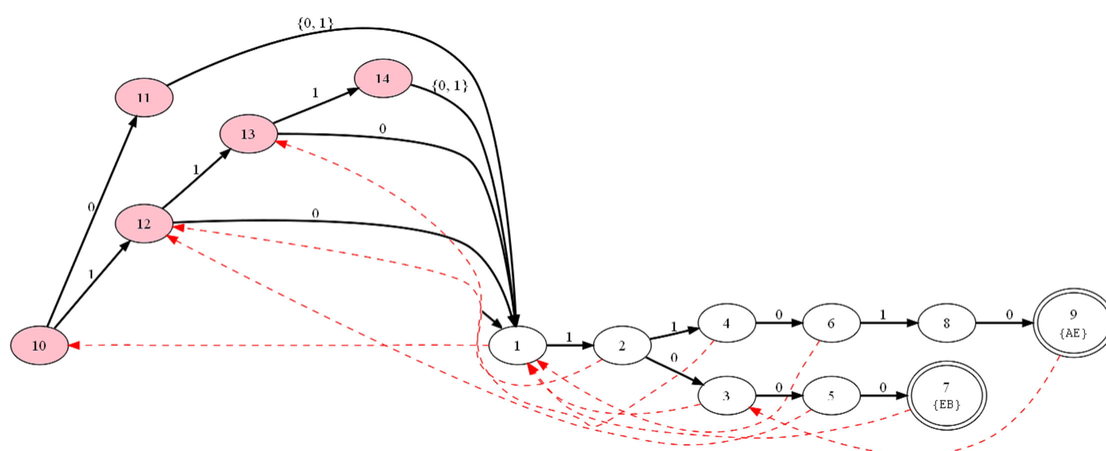
それに対し，図 3.4(a) のように goto 木に Huffman 木を付加してから AC オートマトンを構築することで，符号のずれを防ぐことが出来る．図 3.4(a) は図 3.3(a) のオートマトンを用いてこのバイナリ列を検索しようとしたときの状態遷移を示した図であるが，確かに誤認識を防ぐことが出来ている．

オートマトンに Huffman 木を付加し，パターン集合を Huffman 圧縮した後は，従来の AC オートマトン構築アルゴリズムを用いて Huff-AC オートマトンを構築することが出来る．

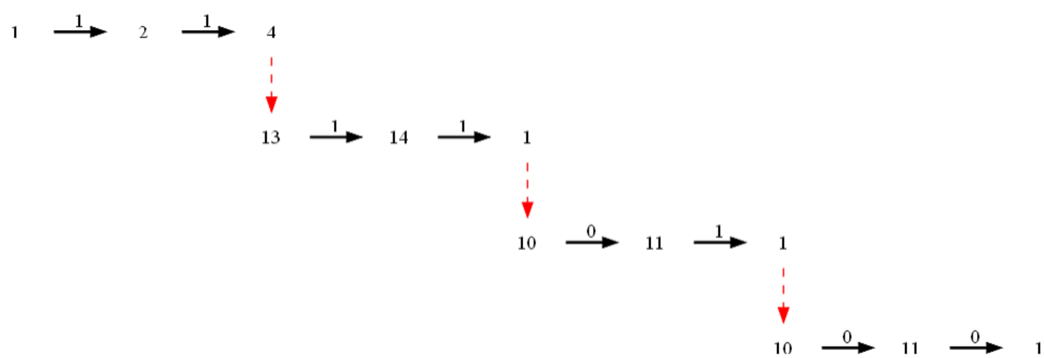
(a) 誤検出の起こりうる Huff-AC オートマトン

(b) 図 3.3(a) のオートマトンでテキスト FDB を検索したときの状態遷移

図 3.3: Huff-AC オートマトンで起こりうる誤検出



(a) 図 3.3(a) を誤検出が起こらないように修正した Huff-AC オートマトン



(b) 図 3.4(a) のオートマトンでテキスト `FDB` を検索したときの状態遷移

図 3.4: 誤検出を防ぐアイディア

3.2.2 KZ-AC 法

前節で述べた Huff-AC 法では Huffman 圧縮後のテキストとパターンを対象に AC 法を適用したが，Huffman 圧縮のかわりに 2.2 節で述べた同じエントロピー圧縮手法の一つである KZ 圧縮を用いる KZ-AC 法を提案する．KZ 圧縮には本論文で提案した瞬時復号可能化した KZ 符号を用いる．

Huffman 圧縮ではパターン途中でのミスマッチによるずれ読みを防ぐ目的でルートノードに Huffman 木を接続する必要があった．KZ 圧縮は Huffman 圧縮に比べて平均符号長が長くなるため，同じテキスト・パターン集合に対する計算時間は Huff-AC 法より遅くなる．しかし，KZ 符号は符号語が必ず 011 で終わる性質を持つため，Huffman 圧縮をベースにした AC オートマトンよりもずれ読みを防ぐための仕掛けがアルファベットによらず，また小さくてすむという利点がある．図 3.5 は，図 3.4(a) と同じ文字列・同じパターン集合に対して構築した KZ-AC オートマトンである．

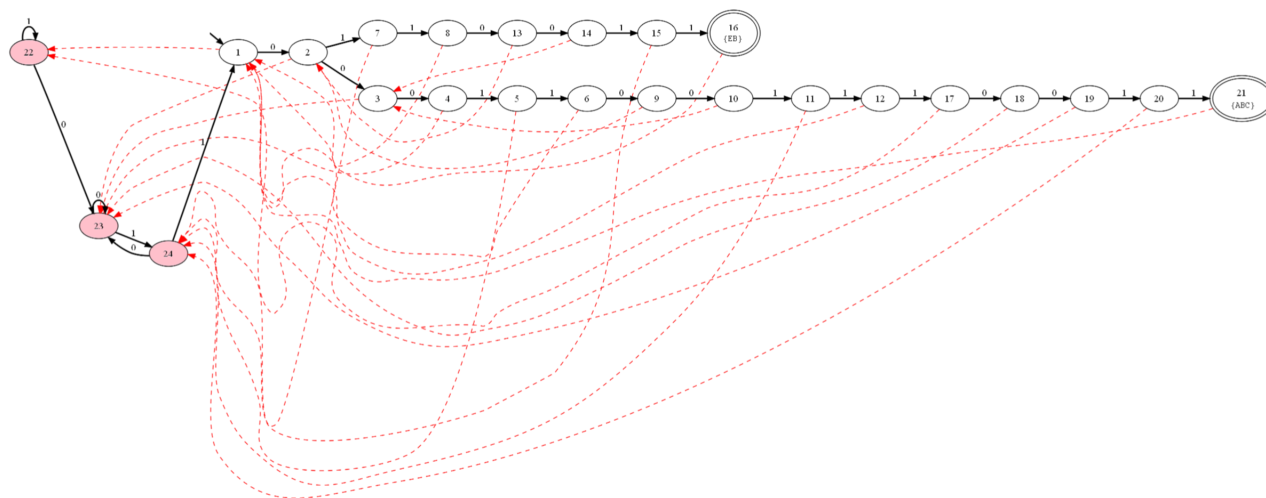


図 3.5: 図 3.4(a) と同じ文字列・同じパターン集合に対して構築した KZ-AC オートマトン

3.2.3 両手法の比較

本節では Huff-AC 法と KZ-AC 法の違いについて考察する．

Huffman 圧縮は KZ 圧縮よりも圧縮率が高い場合が多いことは前章で述べたとおりで

ある．つまり，パターン長も Huffman 圧縮を用いた法が短いビット数で表現できる事が多いため，パターン長が十分に長い場合は Huffman 圧縮の方がノード数が少なくなる．

一方 KZ-AC 法には，ずれ読みを防ぐための仕掛けが小さくてすむという長所がある．図 3.4(a)・図 3.5 中で色が付いているノードは，ずれ読みを防ぐために付け加えたノードである．Huff-AC 法ではここにハフマン木を接続する必要があるため，アルファベットサイズが大きいほどノード数が多くなる．これに対し KZ-AC 法においてはアルファベットサイズによらずノード数が 3 である．

まとめると，パターン長の総和がアルファベットサイズに対して非常に大きい場合は Huff-AC 法の法が少ないノード数で AC オートマトンを実現できるが，アルファベットサイズがパターン長の総和に対して比較的大きい場合，KZ-AC 法が Huff-AC 法よりも少ないノード数で AC オートマトンを構築できることもある．

3.2.4　t ビットまとめ読み法

一般に圧縮文字列検索は，入力するビット長が短くなることから高速化を実現することが出来る．しかし，AC 法に基づいた圧縮文字列検索において 1 ビットずつ状態遷移をする場合，広く用いられている 8 ビット固定長の符号化をした文字と比べ状態遷移の回数は未圧縮の場合よりも増えることから，圧縮文字列検索のほうが全体の計算時間が遅くなる．

そこで高速化のために， [15] で提案されている AC オートマトンを DFA 化する際に t ビットずつまとめ読みする DFA を構築する手法を導入する．この場合の状態遷移回数は 1 ビットずつ読み込む場合の $\frac{1}{t}$ であるため高速化することが出来る．

3.3　各手法の走査に必要な計算量の比較

本節では，圧縮文字列検索の有用性を示すために，各アルゴリズムで検索するための時間計算量・領域計算量について比較・考察する．

検索オートマトンはあらかじめ DFA に変換してあるものとする．

3.3.1　時間計算量

検索に必要な時間はテキストを読む時間とオートマトンの状態遷移にかかる時間の和であると考えることが出来る．

まず，テキストを読む時間について考察する．入力テキスト T は，長さが n 文字で 1 文字を 8 ビットで表している．エントロピー圧縮後の平均符号長を L すると，未圧縮テキストは $8n$ ビットで表現されており，この文字列をエントロピー圧縮すると Ln ビットで表現できる．テキストを読む時間はこの長さに依存する．

次に状態遷移の時間について考察する．状態遷移の時間は次の状態を検索する時間と，出力があるかどうか判定する時間に大別できるが，これはどちらも 1 状態遷移ごとに必要なので，状態遷移の回数に依存する．状態遷移の回数は未圧縮の場合は n 回，1 ビット遷移の圧縮文字列検索アルゴリズムでは Ln 回，t ビットまとめ読み遷移の圧縮文字列検索アルゴリズムでは $\frac{L}{t}n$ 回行う．

一般に状態遷移に必要な時間の方がテキストを読む時間に比べて長いため，エントロピー圧縮をして入力ビット長が短くしても，1 ビットごとの遷移では未圧縮のまま検索する場合とくらべて計算時間が遅くなる事が多い．しかし t ビットまとめ読みを行うことで状態遷移の回数を減らすことが出来るため，t を上手に設定することで高速化をすることが出来る．

また，Huffman 符号は KZ 符号よりも平均符号長が短いので，Huff-AC 法に比べ KZ-AC 法の方が計算時間は長くなる．

3.3.2　領域計算量

検索オートマトンを表現するためには，goto 関数と output 関数を計算できればよい．

goto 関数は前述の通り表一瞥法を用いて実装するものとする．また，output 関数が各ノードにおいて出力すべきパターンの全てが，発見された最長のパターンの接頭辞である．従って発見された最長のパターンのみ記憶すればよいので，各ノードにおいてパターン集合の要素を指し示すポインタが 1 つあれば実現可能である．

AC 法を用いて構築した検索オートマトンのノード数が N であるとする時，各手法の領域計算量を比較する．AC 法には goto 関数のために σN，output 関数のために N 個の

ポインタが必要である．

1ビット遷移の圧縮文字列検索アルゴリズム (Huff-AC法・KZ-AC法) では，パターンとテキストの文字の出現確率分布が同じである場合，goto関数のために $2LN$，output関数のために LN 個のポインタが必要である．ただし L は平均符号長である．Huffman符号はKZ符号よりも平均符号長が短いので，Huff-AC法に比べKZ-AC法の方が領域計算量は大きくなる．

t ビットまとめ読みの圧縮文字列検索アルゴリズムでは状態遷移の途中が文字の切れ目に当たる場合があるため，output関数 *output* を以下のように変更する．

- output関数 *output*: $S \times \Sigma \rightarrow b \times P,\ 0 \leq b \leq t-1,\ P \in 2^{\Sigma^+}$

従来の方法では，各受理状態に対し検出されるパターンの集合を定義していたが，変更した方法では，各状態遷移に対し検出されるパターンと，それが読み込んだ t ビットの何ビット目で検出されたかの組を返す.

表3.1は各手法に対する時間計算量と領域計算量をまとめた表である．

入力文字列 / 読み込みビット数	検索時間	goto関数のための領域 [bit]	output関数のための領域 [ポインタの個数]
非圧縮文字列 / 8ビット読み込み	$n(g+o+8d)$	σN_{Naive}	$n+1$
圧縮文字列 / 1ビット読み込み	$Ln(g+o+d)$	$2N_{Comp.}$	$N_{Comp.} + const.$
圧縮文字列 / 8ビット読み込み	$Ln(\frac{1}{8}(g+o)+d)$	$18N_{Comp.} + const.$	$49N_{Comp.} + const.$
圧縮文字列 / t ビット読み込み	$n(\frac{1}{t}(g+o)+d)$	$(2^t+2)N_{Comp.} + const.$	$(3 \cdot 2^t + 1)N_{Comp.} + const.$

表 3.1: AC法実現の為に必要となる各手法の計算量

この表からも明らかなように，t ビットまとめ読み法は領域計算量を多く使うことで計算時間の高速化を実現している．

第4章

DCパターンを含むパターン集合に対する全文検索アルゴリズム

本章ではメタ文字としてDC文字を導入し，DC文字を含むパターンに対する全文検索について述べる．

4.1 定義

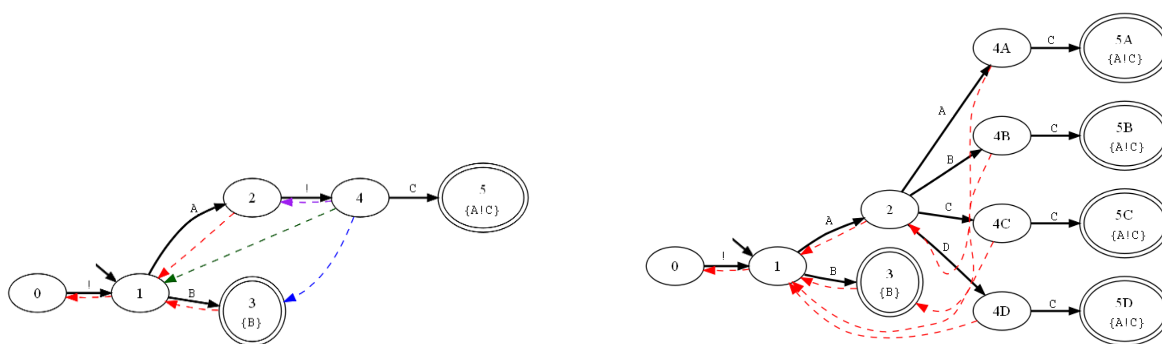
アルファベットΣ上の任意の1文字と一致するメタ文字！をDon't Care文字(DC文字)と定義する．また，DC文字を含む文字列パターンをDCパターンと呼ぶことにする．

4.2 DCパターンを含むパターン集合に対する全文検索

3.1章で述べたAC法を，DCパターンを含むパターン集合に対し拡張する．ただし，DC文字がDCパターンの第0字及び最後の文字であるような場合は考えないことにする．

$p \in P$ なるpの第i字がDC文字であるパターン集合Pに対しACオートマトンを構築することを考える．DC文字について特別な処理をしないままACオートマトンの構築を行った場合，pの第$i-1$字まで検索した状態をs_{i-1}とすると，goto関数$goto(s_{i-1}, !) = s_i$となる．このgoto関数は，状態s_{i-1}においてΣに含まれるどんな文字を入力したとしても状態s_iに遷移することを示している．

図4.1(a)は$\Sigma = \{$A, B, C, D$\}$上のパターン集合$P = \{$A!C, B$\}$を検索するためのACオートマトンを構築アルゴリズムAlgorithm 3により構築した図である．この例においても，DC文字によって同一状態s_4に遷移しているが，状態s_4は！に代入する文字によっ



(a) failure 関数を作ることが出来ない DC 文字を含む AC オートマトン

(b) 図 4.1(a) の FailureLink を修正した AC オートマトン (ノード 5A〜5D の failure 関数は省略)

図 4.1: DC パターンに対する AC オートマトン

て failure 関数が一意に定まらないため，これ以上 AC オートマトンを構築し続けることが出来ない．

このように，DC 文字は Σ 上の文字と同様に扱うことが出来ないため，従来の AC 法で AC オートマトンを構築できない場合があった．そこで本節では DC パターンの展開という概念を導入し，AC オートマトンに基づいた DC パターンに対する全文検索アルゴリズムを提案する．

4.2.1 DC パターンの展開

DC パターン p を Σ 上の文字のみの形に書き換えることを DC パターンの展開と定義する．

DC パターン p から，p 中に出現する全ての DC 文字 ! にアルファベット Σ 上の文字の全ての組み合わせを代入することで DC パターンの展開をすることが出来る．DC パターンを展開して出来たパターン集合を P とする．テキスト中から P を検索した結果と，p を検索した結果が等しくなることは定義より明らかである．与えられたパターン集合 P_0 を展開するためには，P_0 に含まれる全ての DC パターン p について，$P_{expanded} = (P_0 - \{p\}) \cup P$ の処理を行えばよい．

$P_{expanded}$ には DC パターンが含まれていないので， 3.1 章で述べた AC 法を適用す

ることが出来る．つまり，検索したいパターン集合 P_0 に DC パターン p が含まれている場合，従来の AC 法で P_0 を直接検索するのではなく，前処理として P_0 を展開してから $P_{expanded}$ を AC オートマトン構築アルゴリズム Algorithm 3 に入力すればよい．図 4.1(a) を，DC パターンを展開することによって修正した AC オートマトンが図 4.1(b) である．

4.2.2　アクティブなノード

パターン集合 P について構築した AC オートマトンについて考える．$p \in P$ なる p の第 i 字が見つかったときに遷移しうるノードをパターン p の第 i 字におけるアクティブなノードと定義する．

例えば，図 4.1(b) の 4A，4B，4C，4D はパターン $A!C$ の第 1 字に対するアクティブなノードである．

パターン p の第 $i+1$ 字を検索するためのノードを追加する場合，第 i 字を検索する時に遷移しうるどのノードに遷移した状態からでも第 $i+1$ 字の検索が出来なくてはならない．そのため，第 $i+1$ 文字を検索するためのノードは，第 i 字に対するアクティブなノードそれぞれに追加する必要がある．そのため，例においては第 2 字を検索するためのノードを 4A，4B，4C，4D の全てに追加している．

4.2.3　AC 法を用いた DC パターンを含むパターン集合に対する全文検索の計算量

前述したとおり，検索したいパターン集合 P_0 に DC パターン p が含まれている場合は，パターン集合に含まれる全ての DC パターンを展開した後に AC オートマトンを構築すればよい．これは p の第 $i+1$ 字が DC 文字である時，p の第 i 文字におけるアクティブなノード全てに，Σ の要素全てについて goto 関数を作ることを意味している．

本節では，このようにして構築した AC オートマトンのノード数についての定理とその証明のためにいくつかの補題について述べる．

補題（DC 文字を含む AC オートマトンのノード数） AC オートマトンのパターン p の

i 文字目までを検索するために必要なノード数 $N_{p,\ i}$ は以下の漸化式で表すことができる．

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ i} &= \begin{cases} 2N_{p,\ i-1} - N_{p,\ i-2} & (\text{if } T[i] \in \Sigma) \\ (N_{p,\ i-1} - N_{p,\ i-2})\sigma + N_{p,\ i-1} & (\text{if } T[i] = \,!) \end{cases} \end{aligned}$$

証明

1. $i = 0$ の時，パターンの最初の文字は DC 文字ではないので，必ず $N_{p,\ i} = 2N_{p,\ i-1} - N_{p,\ i-2}$ が選ばれる．

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ 0} &= 2N_{p,\ -1} - N_{p,\ -2} \\ &= 3 \end{aligned}$$

これが正しいことは明らかである．

2. $k \geq 1$ を満たす k について，$i = k$ までを検索するために必要なノード数が，

$$N_{p,\ k} = \begin{cases} 2N_{p,\ k-1} - N_{p,\ k-2} & (\text{if } T[k] \in \Sigma) \\ (N_{p,\ k-1} - N_{p,\ k-2})\sigma + N_{p,\ k-1} & (\text{if } T[k] = \,!) \end{cases}$$

であると仮定する．

p の第 k 字におけるアクティブなノード数は $(N_{p,\ k} - N_{p,\ k-1})$ である．

(a) $p[k+1]$ が DC 文字であるとき，$(N_{p,\ k} - N_{p,\ k-1})\sigma$ 個のノードが追加されるので，この場合のノード数は $N_{p,\ k+1} = (N_{p,\ k} - N_{p,\ k-1})\sigma + N_{p,\ k}$ である．

(b) $p[k+1]$ が DC 文字ではないとき，$N_{p,\ k} - N_{p,\ k-1}$ 個のノードが追加されるので，この場合のノード数は $N_{p,\ k+1} = 2N_{p,\ k} - N_{p,\ k-1}$ である．

よって，p の第 $k+1$ 字までを検索するために必要なノード数は，

$$N_{p,\ k+1} = \begin{cases} 2N_{p,\ k} - N_{p,\ k-1} & (\text{if } T[k+1] \in \Sigma) \\ (N_{p,\ k} - N_{p,\ k-1})\sigma + N_{p,\ k} & (\text{if } T[k+1] = \,!\,) \end{cases}$$

である．

1, 2 から数学的帰納法により，パターン p の i 文字目までを検索するために必要なノード数 $N_{p,\ i}$ は以下の漸化式

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ i} &= \begin{cases} 2N_{p,\ i-1} - N_{p,\ i-2} & (\text{if } T[i] \in \Sigma) \\ (N_{p,\ i-1} - N_{p,\ i-2})\sigma + N_{p,\ i-1} & (\text{if } T[i] = \,!\,) \end{cases} \end{aligned}$$

で表すことが出来る事が証明できた．

定理（DC 文字を含む AC オートマトンのノード数の期待値） DC 文字を含む AC オートマトンのノード数の期待値は，以下の不等式を満たす．

$$E[N] \leq \sum_{p \in P} \sum_{k=0}^{|p|-1} \left(2 + \sum_{k=0}^{i} (\tau - \tau\sigma + \sigma)^{k+1} \right)$$

証明 DC 文字を含む AC オートマトンのノード数に関する補題より，p の i 文字目までを検索するために必要なノード数 $N_{p,\ i}$ は以下の漸化式で表すことができる．

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ i} &= \begin{cases} 2N_{p,\ i-1} - N_{p,\ i-2} & (\text{if } T[i] \in \Sigma) \\ (N_{p,\ i-1} - N_{p,\ i-2})\sigma + N_{p,\ i-1} & (\text{if } T[i] = \,!\,) \end{cases} \end{aligned}$$

i 文字目が文字である確率を τ，！である確率を $1-\tau$ とすると，p の i 文字目までを検索

するために必要なノード数の期待値 $E[N_{p,\ i}]$ は,

$$
\begin{aligned}
E[N_{p,\ -2}] &= 1 \\
E[N_{p,\ -1}] &= 2 \\
E[N_{p,\ -i}] &= \tau(2E[N_{p,\ i-1}] - E[N_{p,\ i-2}]) + (1-\tau)((E[N_{p,\ i-1}] - E[N_{p,\ i-2}])\sigma + E[N_{p,\ i-1}])
\end{aligned}
$$

であり,この漸化式をとくと,

$$
E[N_p] = 2 + \sum_{k=0}^{i} (\tau - \tau\sigma + \sigma)^{k+1}
$$

である.展開した検索パターン集合 P_1 に対する AC オートマトンのノード数の期待値を $E[N]$ とする.P_1 に含まれるパターン間で検索に必要なノードが重複しうることを考慮すると,明らかに $E[N] \leq \sum_{p\in P} E[N_p]'$ なので,AC オートマトンのノード数の期待値 $E[N]$ は,

$$
\begin{aligned}
E[N] &\leq \sum_{p\in P} \sum_{k=0}^{|p|-1} E[N_{p,\ k}] \\
&= \sum_{p\in P} \sum_{k=0}^{|p|-1} \left(2 + \sum_{k=0}^{i} (\tau - \tau\sigma + \sigma)^{k+1} \right)
\end{aligned}
$$

である.

以上の結果から,DC 文字を含む AC オートマトンのノード数の期待値 $E[N]$ は,

$$
E[N] \leq \sum_{p\in P} \sum_{k=0}^{|p|-1} \left(2 + \sum_{k=0}^{i} (\tau - \tau\sigma + \sigma)^{k+1} \right)
$$

を満たす事が証明できた.

ただし,i 文字目までに含まれる DC 文字の個数の期待値は $\sum_{k=0}^{i} {}_iC_k \tau^{i-k}(1-\tau)^k k$ である.また,DFA 変換後に必要となる領域は $O(E[N]\cdot\sigma)$ である.

また,パターン集合中の DC 文字の数は検索時間に影響しない.

4.2.4 Huff-AC 法及び KZ-AC 法を用いた DC パターンを含むパターン集合に対する全文検索

目的のパターン集合 P_0 中の DC パターンをすべて展開した後のパターン集合 P_1 は DC パターンを含まないパターン集合なので， 3.1 章で述べた Huff-AC 法や KZ-AC 法のようなエントロピー圧縮後に AC オートマトンを構築する方法でも同様に DC パターンを検索することが出来ると言える．

4.3 提案手法

前節では，従来のアルゴリズムを自然に拡張した DC パターンを含むパターン集合に対する検索アルゴリズムついて述べた．しかし，このアルゴリズムでは 1 パターン中の DC 文字の出現数 n に対して $O(\sigma^n)$ のノード数が必要になるため，膨大な領域計算量が必要になる．

本節では領域計算量を節約しつつ AC オートマトンを構築するアルゴリズムを提案する．

4.3.1 $\Sigma_{P,\ i}$ と $\#_{P,\ i}$ の導入

パターン集合 P に含まれる各パターンの第 i 字までに出現する文字の集合を $\Sigma_{P,\ i}$ と定義する．ただし，パターン長が i よりも小さい場合は，パターンに含まれる全ての文字を $\Sigma_{P,\ i}$ に含める．

また，集合 $\hat{\Sigma}_{P,\ i} = \Sigma - \Sigma_{P,\ i}$ に含まれる任意の 1 文字と一致するメタ文字 $\#_{P,\ i}$ を導入する．

集合 $\Sigma_{P,\ i}$, $\hat{\Sigma}_{P,\ i}$, メタ文字 $\#_{P,\ i}$ は，パターン集合 P に依存するものの，文脈上そのことが明らかな場合は Σ_i，$\hat{\Sigma}_i$, $\#_i$ のように省略して表記することとする．特にメタ文字 $\#_{P,\ i}$ はパターン中の DC 文字の出現位置 i に依存することが文脈上明らかな場合があるので，その場合 # のように省略して表記することとする．

4.3.2 DC パターンの展開の定義の拡張

DC パターン中に出現する全ての DC 文字！について，その出現位置 i に対応した $\#_i$ と Σ_i に含まれる文字の全ての組み合わせを代入して出来るパターン集合を P' とする．P'

に含まれるパターン中に出現する全ての $\#_i$ に $\hat{\Sigma}_i$ に含まれる文字の全ての組み合わせを代入して出来るパターン集合を P とすると，$\#_i$ の定義より P はDCパターンを展開して出来るパターン集合そのものである．

この結果から，本節ではパターン集合の展開の定義を拡張し，DCパターン p を Σ 上の文字と $\#_i$ のみの形に書き換えることをDCパターンの展開であると再定義する．Algorithm 6 はDCパターンを展開するアルゴリズムである．

Algorithm 6: パターン展開アルゴリズム

Input : パターン集合 $P = \{p_0,\ p_1,\ \cdots,\ p_{n-1}\}$

Output : 展開したパターン集合 *expandedPatternSet*

```
expandedPatternSet ← {}
for all c in Σ do
    usedChar[c] ← fail
end
for i ← 0 to n − 1 do
    for j ← 0 to |p| do
        c ← p_i[j]
        if c == '!' then continue
        idx ← usedChar[c]
        if idx > j or idx == fail then
            usedChar[c] ← j
        end
    end
end
for i ← 0 to n − 1 do
    bufferSet ← {p_i}
    for j ← 0 to |p_i| − 1 do
        for all p in bufferSet do
            if p[j] ==' !' then
                bufferSet ← bufferSet − {p}
                for all c in Σ do
                    if usedChar[c] ≤ j then
                        p[j] ← c
                        bufferSet ← bufferSet ∪ {p}
                    end
                end
                p[j] ←' #'
                bufferSet ← bufferSet ∪ {p}
            end
        end
    end
    expandedPatternSet ← expandedPatternSet ∪ bufferSet
end
```

また，図 4.2 は，図 4.1(b) を，拡張した展開の定義を用いて修正したオートマトンである．

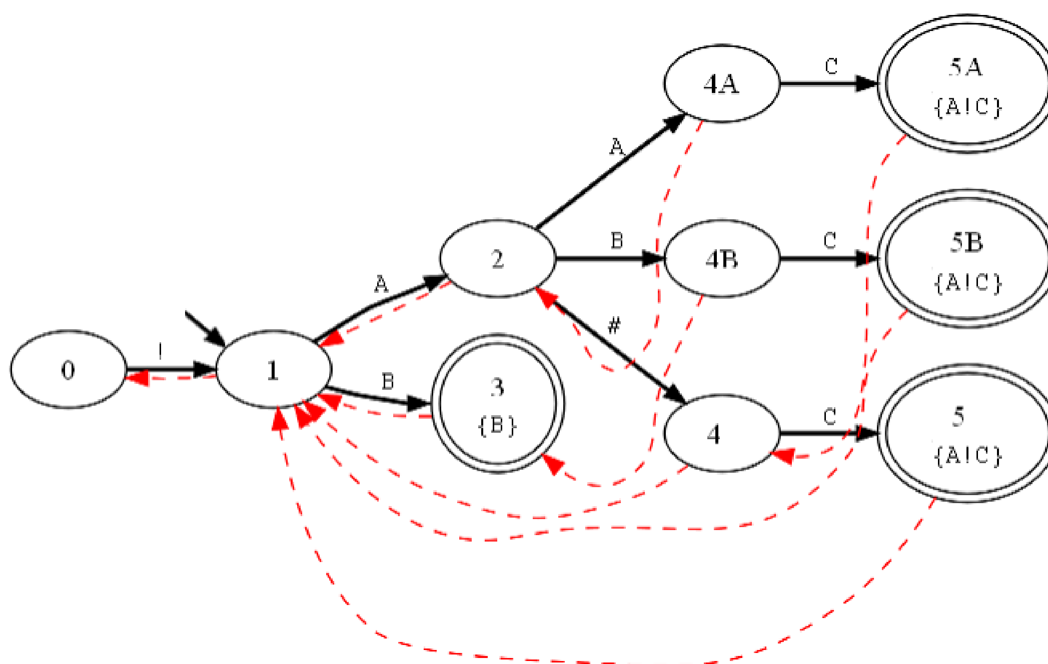


図 4.2: 図 4.1(b) を # を用いて修正した AC オートマトン

4.3.3 AC 法における $\#_i$ の実現手法

展開した DC パターン p から AC オートマトンを構築できることを示したい．そのためには，$\#_i$ で遷移した先の状態 s_i の failure 関数の値が $\hat{\Sigma}_i$ に含まれるどの文字で遷移したとしても常に同じであればよい．

1. DC パターンはパターンの先頭に DC 文字を許さないため，s_0 は存在しない．

2. failure 関数を計算するアルゴリズムより，DC パターンを含む任意のパターン集合に対して，$p \in P$ なる p について $failure(goto(1, p[0])) = 0$ である．

   よって $failure(s_1)$ は，

$$failure(s_1) \quad = \quad \begin{cases} goto(0,\ c) & (\text{if } c \in \Sigma_0 \cap \hat{\Sigma}_1) \\ 0 & (\text{otherwise}) \end{cases}$$

$\Sigma_0 \cap \hat{\Sigma}_0 = \phi$ かつ $\hat{\Sigma}_1 \subsetneqq \hat{\Sigma}_0$ であるので，$\Sigma_0 \cap \hat{\Sigma}_1 = \phi$ である．

$\mathit{failure}(s_1) = 0$ は常に成り立つので，AC オートマトンを構築することが出来る．

3. $i \leq k$ なるすべての i について，DC パターンを含む任意のパターン集合に対して AC オートマトンが構築できたと仮定する．

   failure 関数を計算するアルゴリズムより $\mathit{failure}(s_{k+1})$ を計算するためには，s_{k+1} の親ノードから $\mathit{goto}(s_j, \#_i) \neq \mathit{fail}$ なるノード s_j が見つかるまで failure 関数を繰り返し計算する．このような goto 関数が見つかるのは，以下の 2 つの場合のみである．

   (a) $\mathit{goto}(0, !) = 1$

   (b) $\mathit{goto}(s_j, \#_j) = s$

   (a) の場合はアルファベット上の任意の文字の入力に対し，遷移先がただ一つ定まるので，$\mathit{failure}(s_k) = 0$ といえる．また (b) の場合も，$j < k$ であるので $\hat{\Sigma}_k \subsetneqq \hat{\Sigma}_j$ は自明に成り立つ．$c \in \hat{\Sigma}_k$ なる任意の c について，$\mathit{goto}(s_j, c) = \mathit{goto}(s_j, \#_j)$ であるため，$\mathit{failure}(s_k) = \mathit{goto}(s_j, \#_j)$ である．

   以上の結果から，$i \leq k$ なるすべての i について，DC パターンを含む任意のパターン集合に対して AC オートマトンが構築できれば，$k+1$ の場合においても AC オートマトンを構築できる．

よって数学的帰納法により，展開したパターンから AC オートマトンは構築できる．

Algorithm 3 中で呼び出している Algorithm 1 を Algorithm 7 のように書き換えることで，機械的に DC 文字を含む AC オートマトンを構築できる．

Algorithm 7: Algorithm 1 を DC 文字の受理の為に書き換えたアルゴリズム

```
for i ← 0 to n − 1 do
    p ← p_i
    s ← 1
    for j ← 0 to |p| − 1 do
        if p[j] ==' #' then
            for all c in Σ do
                if usedChar[c] > j or usedChar[c] == fail then
                    character = c
                    break
                end
                character = c
            end
        else
            character = p[j]
        end
        if g(s, character) == fail then
            newState ← newState + 1
            if p[j] ==' #' then
                for all c in Σ do
                    if usedChar[c] > j or usedChar[c] == fail then
                        goto(s, c) ← newState
                    end
                end
            else
                goto(s, p[j]) ← newState
            end
            parent(newState) ← s
            s ← newState
        else
            s ← goto(s, character)
        end
    end
    output(s) ← p
end
```

4.3.4 提案手法を実現するために要するノード数

$|\Sigma_i| = \sigma_i$ と表す.

補題（DC 文字を含む提案手法を用いて構築した AC オートマトンのノード数） 提案手法を用いて構築した AC オートマトンの，パターン p の i 文字目までを検索するために必要なノード数 $N_{p,\ i}$ は以下の漸化式で表すことができる．

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ i} &= \begin{cases} 2N_{p,\ i-1} - N_{p,\ i-2} & (\text{if } T[i] \in \Sigma) \\ (N_{p,\ i-1} - N_{p,\ i-2})\sigma_i + N_{p,\ i-1} & (\text{if } T[i] = !) \end{cases} \end{aligned}$$

証明は 4.2.3 内の補題の証明とほとんど同じなので割愛する．

定理（DC 文字を含む提案手法を用いた AC オートマトンのノード数の期待値） DC 文字を含む提案手法を用いて構築した AC オートマトンのノード数の期待値 $E[N]$ は，次の不等式を満たす．

$$E[N] \ \le \ \sum_{p \in P} \left(2 + \sum_{j=0}^{i} \prod_{k=0}^{j} \left(\tau + (1-\tau) \left(\max E[\sigma_k] + \left(1 - \frac{\sigma - \max E[\sigma_k]}{\sigma} \right) \right) \right)^{k+1} \right)$$

である．

証明

DC 文字を含む提案手法を用いて構築した AC オートマトンのノード数に関する補題より，提案手法における p の i 文字目までを検索するために必要なノード数 $N_{p,\ i}$ は以下の漸化式で表すことができる．

$$\begin{aligned} N_{p,\ -2} &= 1 \\ N_{p,\ -1} &= 2 \\ N_{p,\ i} &= \begin{cases} 2N_{p,\ i-1} - N_{p,\ i-2} & (\text{if } T[i] \in \Sigma) \\ (N_{p,\ i-1} - N_{p,\ i-2})\sigma_i + N_{p,\ i-1} & (\text{if } T[i] = !) \end{cases} \end{aligned}$$

i 文字目が文字である確率を τ，！である確率を $1-\tau$ とすると，p の i 文字目までを検索

するために必要なノード数の期待値 $E[N_{p,\ i}]$ は，

$$
\begin{aligned}
E[N_{p,\ -2}] &= 1 \\
E[N_{p,\ -1}] &= 2 \\
E[N_{p,\ i}] &= \tau(2E[N_{p,\ i-1}] - E[N_{p,\ i-2}]) + (1-\tau)((E[N_{p,\ i-1}] - E[N_{p,\ i-2}])\sigma_i + E[N_{p,\ i-1}])
\end{aligned}
$$

この漸化式を解くと，

$$
E[N_{p,\ i}] = 2 + \sum_{j=0}^{i} \prod_{k=0}^{j} (\tau + (1-\tau)\sigma_k)^{k+1}
$$

である．

さらに，σ_i は次の漸化式で与えられる．

$$
\begin{aligned}
\sigma_1 &= 1 \\
\sigma_{k+1} &= \begin{cases} \sigma_k & (\text{if } p[k+1] \in \sigma_k) \\ \sigma_k + 1 & (\text{otherwise}) \end{cases}
\end{aligned}
$$

$p[i+1] \in \sigma_i$ である確率を p，σ_i の期待値を $E[\sigma_i]$ とすると，

$$
\begin{aligned}
E[\sigma_1] &= 1 \\
E[\sigma_{k+1}] &= p_k E[\sigma_k] + (1-p_k)(E[\sigma_k]+1) \\
&= E[\sigma_k] + (1-p_k)
\end{aligned}
$$

である．文字の出現に偏りがない場合に p は，$p_i = \frac{\sigma - \sigma_i}{\sigma}$ で最小になるため，このとき $E[\sigma_i]$ は最大値 $\max E[\sigma_i]_i$ をとる．

$$
\begin{aligned}
\max E[\sigma_1] &= 1 \\
\max E[\sigma_{k+1}] &= \max E[\sigma_k] + \left(1 - \frac{\sigma - \max E[\sigma_k]}{\sigma}\right)
\end{aligned}
$$

提案手法を用いて p を検索するために必要なノード数の期待値 $E[N_p]$ は，

$$
\begin{aligned}
E[N_p] &\leq 2 + \sum_{j=0}^{i} \prod_{k=0}^{j} (\tau + (1-\tau)\sigma_k)^{k+1} \\
&= 2 + \sum_{j=0}^{i} \prod_{k=0}^{j} \left(\tau + (1-\tau)\left(\max E[\sigma_k] + \left(1 - \frac{\sigma - \max E[\sigma_k]}{\sigma}\right)\right)\right)^{k+1}
\end{aligned}
$$

よって，パターン集合 P_2 に対する提案手法を用いた AC オートマトンのノード数の期待値 $E[N]$ は，

$$E[N_p] \leq \sum_{p \in P} \left(2 + \sum_{j=0}^{i} \prod_{k=0}^{j} \left(\tau + (1-\tau) \left(\max E[\sigma_k] + \left(1 - \frac{\sigma - \max E[\sigma_k]}{\sigma} \right) \right) \right)^{k+1} \right)$$

を満たす．

4.3.5 提案手法の Huff-AC への応用

提案手法を Huff-AC 法に対して適用する方法について述べる．Huff-AC 法において提案手法を実現するには `#`$_i$ が Huffman 符号化後のオートマトンに対しても構築可能であることを示せばよい．

p の第 $i-1$ 文字を読んだときに遷移するノードのうちの 1 つを s_{i-1} とする．DC 文字が第 i 字目に出現する場合，従来の手法と同様に DC 文字を Σ を用いて展開する．その後 $\hat{\Sigma}_i$ に含まれる全ての文字について，対応する検索ノードの末端のノードを 1 つに統合し，これを s_i とすると，s_{i-1} から s_i への遷移は `#`$_i$ で遷移していると言える．

図 4.3 は，$P = \{$`A!C`, `B`$\}$ を図 2.1 のハフマン木を用いて Huffman 圧縮したパターン集合に対し，提案手法を用いて構築した DC-AC オートマトンである．緑の枠で囲った部分

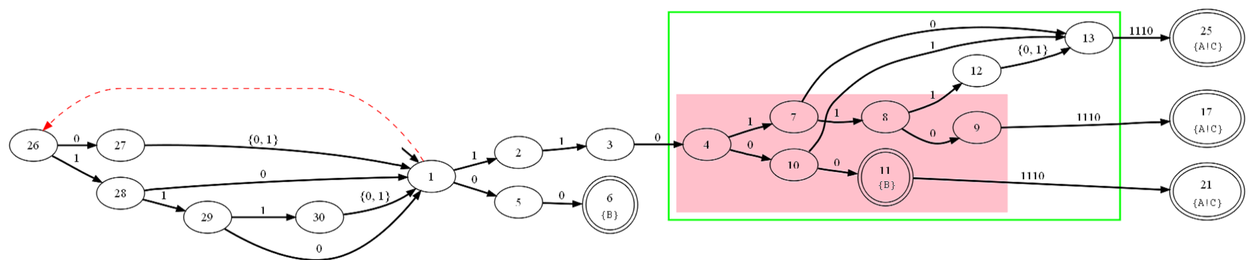


図 4.3: DC 文字を含むパターン集合に対する Huff-AC オートマトン（一部略記）

が DC 文字を受理する部分であり，これはハフマン木の葉ノードをいくつか統合した木である．また，網掛け部分は DC 文字として `A`，`B` を検索するためのノードであり，これ以外の文字を受理する部分はノード 13 に統合されている．

この例においてはアルファベットサイズが 6 であるため，従来の手法では DC 文字の部分で 6 分岐するはずであったが，DC 文字の出現位置の前には 2 種類の文字しか出現して

いないため，前節で述べたとおり，その他の文字による分岐は冗長である．よって，高々 3 つに分岐すれば検索可能なオートマトンを構築することが出来る．

このように冗長なハフマン木の分岐を無くすることにより，AC 法と同様，DC 文字によるアクティブなノードの増加を抑え，少ないノード数で AC オートマトンを実現している．

4.4 KZ-AC 法に特化した提案手法

KZ 圧縮は符号語末のみに 011 が出現するという性質がある．このため Huffman 圧縮とは異なり，任意の 1 文字を示すビットパターンを受理するオートマトンを少ないノード数で表現できる．本節ではこの性質を利用して，KZ-AC 法において DC 文字をより少ないノード数で表現する手法を提案する．

補題（DC 文字を検索するときの failure 関数） Algorithm 4 で，DC 文字を検索するときに failure 関数が呼ばれることはない．

証明 failure 関数が呼び出されるためには，$goto(state,\ !) = fail$ である必要がある．しかし，DC 文字の定義から goto 関数は全ての $c \in \Sigma$ について定義されているので，$goto(state,\ !) = fail$ とはなりえない．

よって，Algorithm 4 で，DC 文字を検索するときに failure 関数が呼ばれることはない．

補題（$\#_i$ の failure 関数を用いた実装） AC オートマトンで，パターン p の第 i 文字が DC 文字である時，$\#_i$ で遷移する goto 関数は failure 関数と ! で遷移する goto 関数によって実装することができる．

証明 p の第 $i-1$ 字まで検索した状態を s_{i-1} とする．$\#_i$ の failure 関数を用いた実装に関する補題から，状態 s_{i-1} の failure 関数 $failure(s_{i-1})$ は呼び出されることはない．そこで，新しい状態 s_{new0} を作り $failure(s_{i-1}) = s_{new0}$ とする．また，もう一つ新しい状態 s_{new1} を作り，$goto(s_{new0},\ !) = s_{new1}$ とする．$c \in \hat{\Sigma}$ なるすべての c について goto 関数 $goto(s_{i-1},\ c) = fail$ とすると，状態 s_{new0}，s_{new1} と遷移する．これは，$goto(s_{i-1},\ \#) = s_{new1}$ と同じ状態遷移を failure 関数と ! による遷移で表したことになる．

よって，AC オートマトンで，パターン p の第 i 文字が DC 文字である時，$\#_i$ で遷移する goto 関数は failure 関数と ! で遷移する goto 関数によって実装することができる．

この手法を KZ-AC に適用すると，以下の通りになる．パターン p の第 i 文字が DC 文字である時，p の第 $i-1$ 字まで検索した状態を s_{i-1} とすると，

1. $c \in \Sigma_i$ なる c に対応するビットパターンの goto 関数を構築する．
2. $failure(s_{i-1})$ の値を f_{temp} にコピーしてから，$\#_i$ の failure 関数を用いた実装に関する補題の通り，新たなノード s_{new} を作り，$f(s_{i-1}) = s_{new}$ と変更する．
3. ! を検索する DFA を s_{new} 以下に構築する．
4. 各ビットパターンの failure 関数を構築する．ただし，各ビットパターンの末端ノードの failure 関数を構築する時のみ，$failure(s_{i-1}) = s_{temp}$ であるとして計算する．

このように構築したオートマトンの DC 文字を検索しているときの挙動について述べる．このオートマトンの DC 文字を検索する部分に $c \in \Sigma_i$ なる c を入力した場合，それに対応するビットパターンの状態に状態遷移を続け，DC 文字の検索部分を抜ける．$c \in \hat{\Sigma}_i$ なる c を入力した場合，DC 文字の検索部分を抜ける前に状態遷移の過程のいずれかで失敗関数が呼ばれ，! を見つける DFA のいずれかに状態遷移する．その後，次の文字の先頭であるビットパターン 110 が見つかるまで DFA 内で状態遷移を続ける．

図 4.4 は，$P = \{$A!C, B$\}$ を表 2.3 を用いて KZ 圧縮したパターン集合に対し，提案手法を用いて構築した DC-AC オートマトンである．

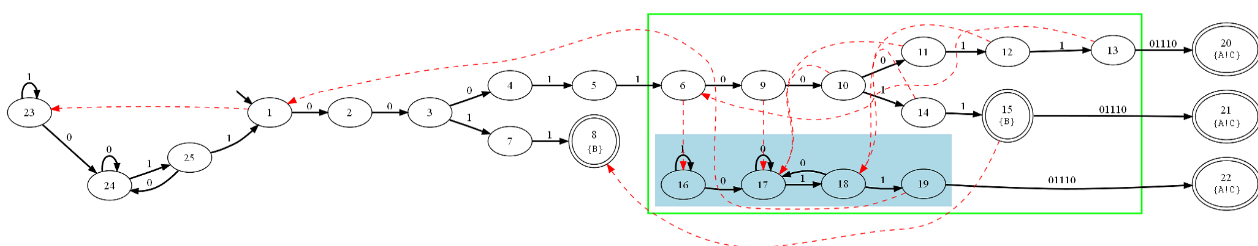


図 4.4: DC 文字を含むパターン集合に対する KZ-AC オートマトン（一部略記）

緑の枠で囲った部分が DC 文字を受理する部分であり，網掛け部分は DC 文字として今まで出現していない文字を受理する部分である．網掛け部分のノード数はアルファベットサイズに依らず 4 であるため，アルファベットサイズが大きい場合には相対的にノード数の削減効果が高くなる．

DC 文字を受理する部分のノード数は，DC 文字の出現位置の前に出る文字の種類によって決まるので，テキストで使用される文字の種類よりも DC 文字の出現位置より前に出る文字の種類が小さい場合は状態数削減の効果は非常に大きい．

第 5 章

実験

本章では前章までに提案したことについて実際に実装し，実験したその結果について述べる．

3.1.4 節で示したとおり，AC オートマトンは DFA にすることが出来る．また，DFA は Algorithm 8 を用いることによってノード数を最小化することが知られている．これは同じ能力を持つ検索オートマトンの中で，状態数が最小の場合であると考えることが出来る．そこで，提案手法が状態数をどの程度圧縮したかを調べるために，

- Naive　：従来の AC オートマトンを自然に拡張した方法
- Proposed: 提案手法により，ノード数を削減した検索オートマトン
- Minimize: Naive によって生成された AC オートマトンを DFA 化した後最小化した検索オートマトン

という，3 つの条件について以下の実験を行う．

Algorithm 8: DFA の最小化アルゴリズム

```
Input   : DFA
Output  : ノード数が最小である DFA

for i ← 0 to N do
    for j ← 0 to i do
        sameStates(i, j) ← (i, j)
        if output(p_i) ≠ output(p_j) then
            isDifferenrState(i, j) ← True
        else
            isDifferenrState(i, j) ← False
        end
    end
end
for i ← 0 to N do
    for j ← 0 to i do
        for all c in Σ do
            r ← goto(p_i, c)
            s ← goto(p_j, c)
            if isDifferenrState(r, s) == True then
                isDifferenrState(i, j) ← True
                for all statePair in sameStates(r, s) do
                    isDifferenrState(statePair) ← True
                end
                break
            else
                sameStates(r, s) ← sameStates(r, s) ∪ sameStates(i, j)
            end
        end
    end
end
```

5.1 DC 文字の出現位置

パターン中に DC 文字が出現すると，それ以降のアクティブなノード数が増えるためにノード数が多くなると予想される．それは，パターンの前のほうに DC 文字が出現す

るほど影響が大きいと考えられる．一方，提案手法においてはそれまでに出た文字の種類がアクティブなノード数に影響するため，パターンの後ろの方に DC 文字が出現するほど影響が大きいと考えられる．

このように，DC 文字のパターン中の出現位置は AC オートマトンのノード数に影響している．そこで以下の実験を行った．

5.1.1 DC 文字が 1 出現である場合の DC 文字の出現位置がノード数に与える影響

DC 文字がパターン中にただ 1 度出る場合について，DC 文字の出現位置を 1〜18 の間で変えノード数を数えた．以下の条件で検索パターンを発生させ，実験を行う．

- パターン長　　　　: 20
- パターン数　　　　: 1
- アルファベットサイズ: 52

本論文では複数のパターンを 1 度の検索で同時に検索することを目的とはしているものの，本実験は DC 文字の出現位置がノード数に与える影響を調べるための実験であるため，パターン数が 1 の場合のみ実験している．

図 5.1(a) はその実験結果であるが，従来の手法と比べ提案手法少ないノード数で AC オートマトンを構築することが出来た．

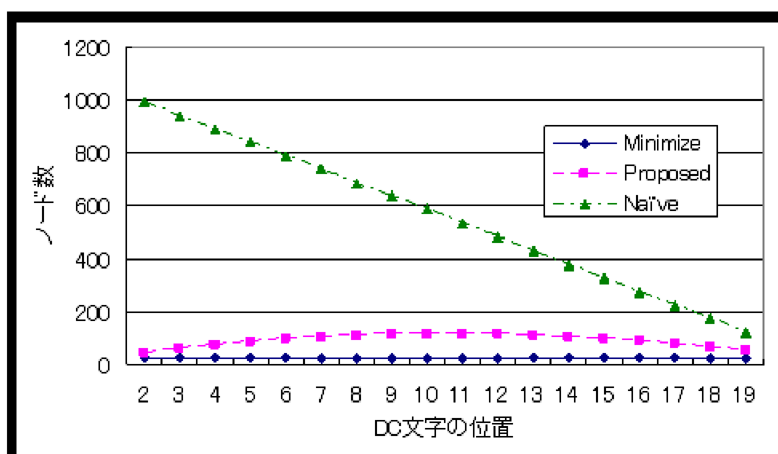
図 5.1(b) は図 5.1(a) を拡大したグラフである．従来の手法は DC 文字の出現位置が後ろになる程単調減少しており，最小化した場合はノード数がほとんど出現位置によらないのに対し，提案手法のグラフはパターン長の中程を頂点とした放物線を描いている．

この実験について，提案手法により構築された AC オートマトンのノード数 N の上限は，第 $i-1$ 字まで全て別な文字が出現した場合であり，次式で表される．

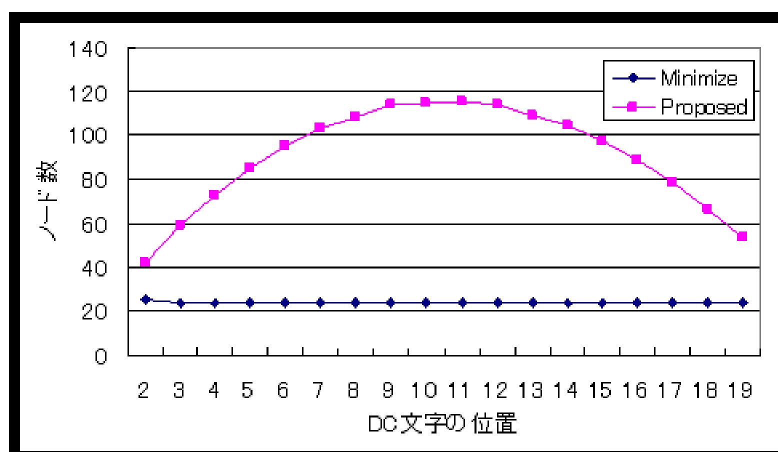
$$N_i = (20 - i) \cdot i + 2$$

この式はパターン長の半分の位置を頂点とした 2 次関数である．図 5.1(c) は，この式により導かれたノード数の上限と実験結果であるが，上限と実験結果はほとんど同じ形のグラフであった．

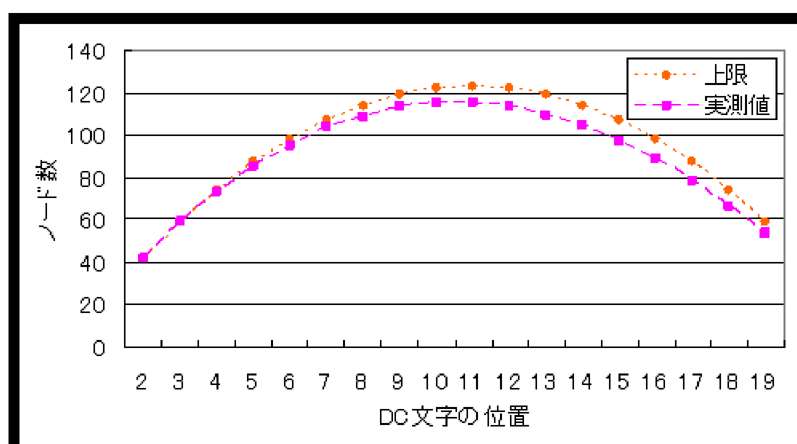
これらの結果から，提案手法によって AC オートマトンのノード数を減らすことが出来たと言える．



(a) DC 文字の出現位置とノード数



(b) DC 文字の出現位置とノード数（拡大図）



(c) 提案手法における DC 文字の出現位置によるノード数の上限

図 5.1: DC 文字の位置とノード数

5.1.2 DC 文字が 2 出現である場合の DC 文字の出現位置がノード数に与える影響

DC 文字が複数出現する場合のノード数の変化を知るために，DC 文字が 2 出現である場合の検索オートマトンのノード数を調べる．DC 文字は第 1 字には出現するものとして 2 つ目の DC 文字の出現位置を 2〜18 の間で変化させて実験する．その他の条件は 5.1.1 節と同じである．

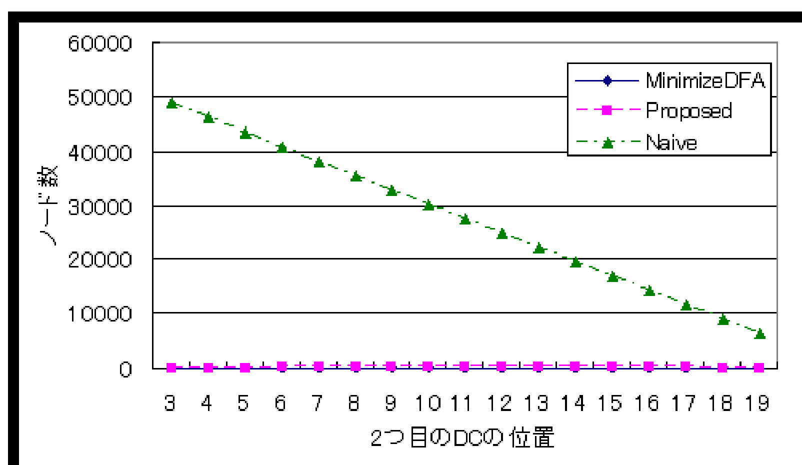
図 5.2 は実験結果である．

ただし，図 5.2(c) の上限を与える式は，

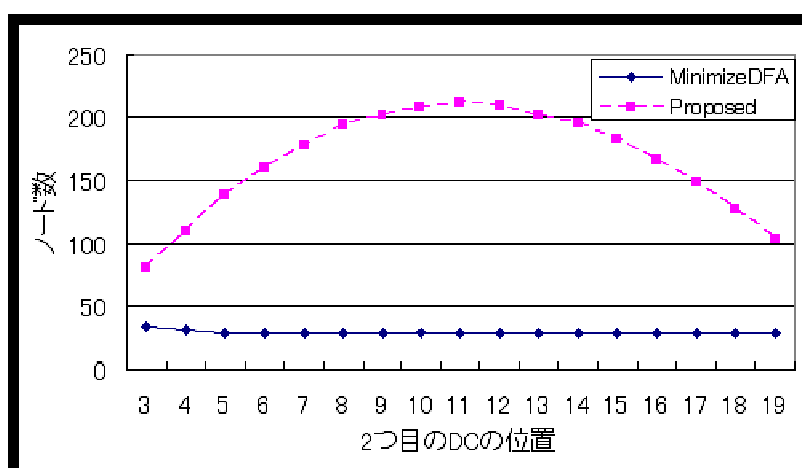
$$N_i = 4 + 2(i-1)(22-i)$$

である．

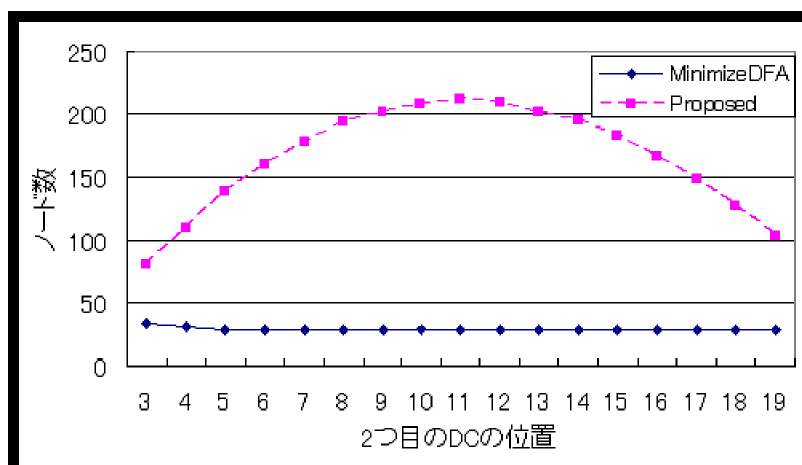
これらのグラフからも分かるように，1 出現の場合と同様に提案手法は少ないノード数で AC オートマトンを構築できると言える．さらに，従来の手法により構築される AC オートマトンのノード数は出現数の指数乗であるので，DC 文字の出現数が多いほど提案手法によるノード数の削減効果が大きいと言える．



(a) DC 文字の出現位置とノード数



(b) DC 文字の出現位置とノード数（拡大図）



(c) 提案手法における DC 文字の出現位置によるノード数の上限

図 5.2: 2 出現目の DC 文字の位置とノード数

5.2 パターン数がACオートマトンのノード数に与える影響

本論文では複数のパターンを1度の検索で同時に検索することを目的としている．そのため，DCパターンの数がACオートマトンのノード数に与える影響を調べる必要がある．

パターン数を1〜10の間に設定し，以下の条件で検索パターン集合を発生させて実験を行った．

- パターン長 ： 10
- アルファベットサイズ: 52
- パターン集合に含まれる全てのパターンの第1字がDC文字

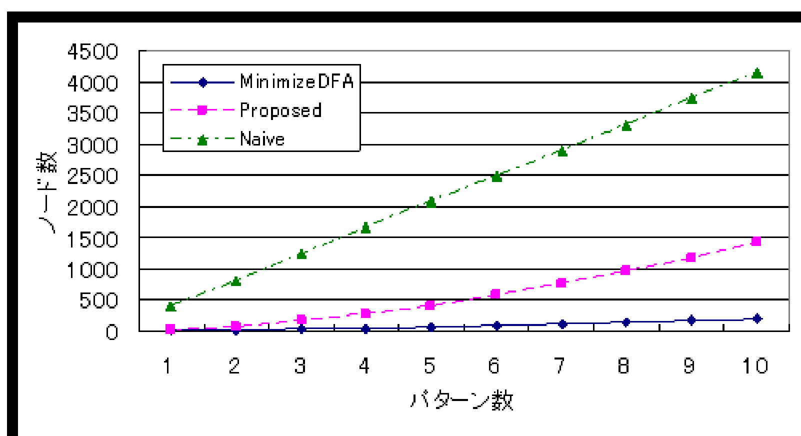
図5.3(a)はその実験結果であるが，パターン数が複数の場合でも，従来の手法と比べ提案手法はノード数が大きく改善していることが分かる．

図5.3(b)は図5.3(a)を拡大したグラフであるが，提案手法によって構築されたACオートマトンのノード数はパターン数の2乗に比例している．これは，パターン数が増えると単純に検索に必要なノード数が増えるだけでなく，第0字で使われている文字の種類が増えるため，DC文字を表現する際に必要なアクティブノードの数が増えることに起因している．

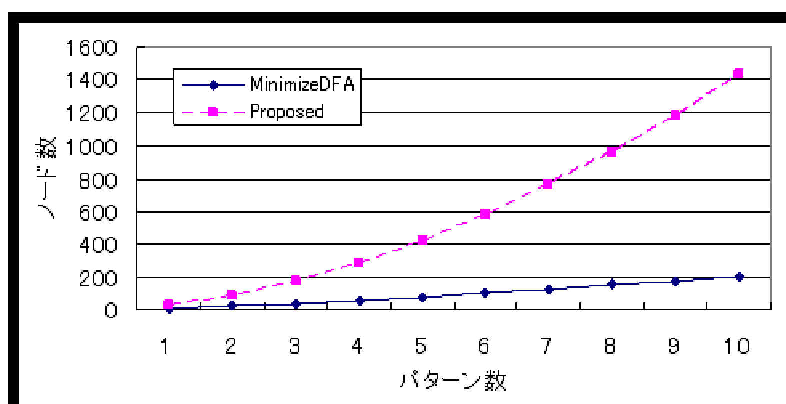
この実験の場合の提案手法により構築されたACオートマトンのノード数Nの上限は，各パターンの第0字が全て異なる場合であり，次式で表される．

$$N_{|P|} = 9|P|^2 + 11|P| + 2$$

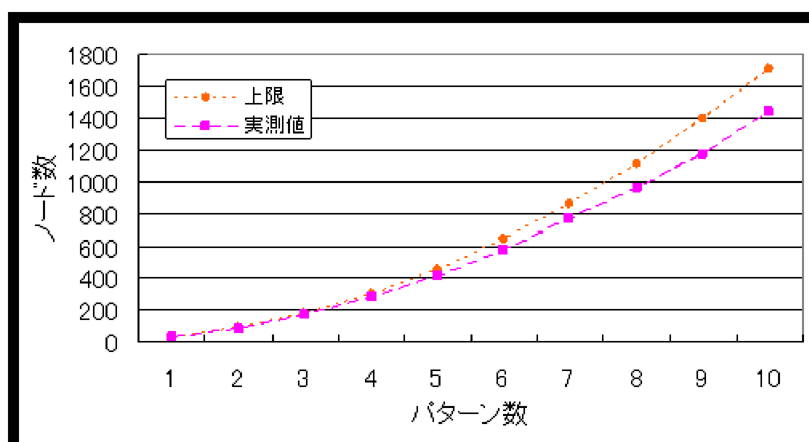
図5.3(c)は，提案手法による実験結果とともに，この式により導かれたノード数の上限を示したグラフであるが，計算によって求めた上限と実験結果はほとんど同じ形のグラフであった．



(a) DC 文字の出現位置とノード数



(b) DC 文字の出現位置とノード数（拡大図）



(c) 提案手法における DC 文字の出現位置によるノード数の上限

図 5.3: パターン数とノード数の関係

5.3 エントロピー圧縮したパターンに対する検索オートマトンのノード数

本節では Huffman 符号と KZ 符号に対する検索オートマトンのノード数の比較を実験的に示す．前述したとおり，通常は平均符号長の短い Huffman 符号の方が，KZ 符号を採用した場合と比べ少ないノード数で検索オートマトンを実現できそうである．しかし，ずれ読みを防止する機構を KZ 符号は定数サイズで実現できるのに対し，Huffman 符号はそれがアルファベットサイズに依存するため，アルファベットサイズが大きいほど KZ 符号が Huffman 符号よりも少ないノードで検索オートマトンを実現できると予想できる．また，DC 文字が多い場合も KZ 符号がノード数が少なくなるであろう事が予想できる．これらのことを確かめるため，以下の実験を行った．

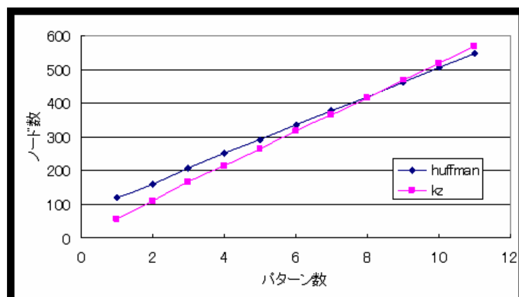
パターン数を 1〜11 の間で変化させ，アルファベットサイズの異なる 2 つのテキストについて，KZ 符号と Huffman 符号のそれぞれに対する AC オートマトンを構築し実験を行った．DC 文字の位置はランダムであり，100 回の実験結果の平均をこの実験の結果とする．

実験のその他の条件は以下の通りである．

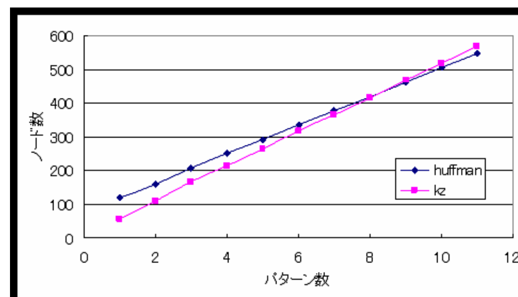
- パターン長: 10
- アルファベットサイズ:
    1. English （英文テキスト [4]）: 72
    2. Protein （アミノ酸配列 [5]）: 21

アルファベットサイズの小さい Protein では DC 文字の出現数やパターン数に関わらず，KZ 符号よりも Huffman 符号のほうがノード数が少なくオートマトンの構築が出来ている (図 5.4(b)，図 5.4(d)，図 5.4(f))．English は Protein よりもアルファベットサイズが大きいのであるが，パターン数が少ない場合で KZ 符号の方が Huffman 符号よりもノード数が少ない事象を観測することが出来た (図 5.4(b)，図 5.4(d)，図 5.4(f))．これはずれ読みを防止する部分や DC 文字を受理する部分が Huffman 符号に比べコンパクトに実現できている事に起因していると推定できる．DC 文字が 1 出現である場合，KZ 符号によって DC 文字を表現するために必要なノード数を調べたのが，図 5.3 である．

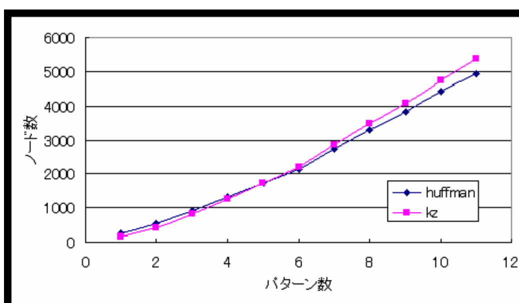
Huffman 符号において DC 文字はおよそ σ 個のノードで表されるので，図 5.3 から Protein では Huffman 符号にくらべ KZ 符号が DC 文字をコンパクトに表現出来ているとは言えない．それに対し，アルファベットサイズの大きな English においては KZ 符号を用いることで DC 文字をコンパクトに表現できている．つまり，このことがアルファベットサイズが大きいときに，KZ 符号が Huffman 符号と比べ検索オートマトンをコンパクトに表現しうる一因であるといえる．



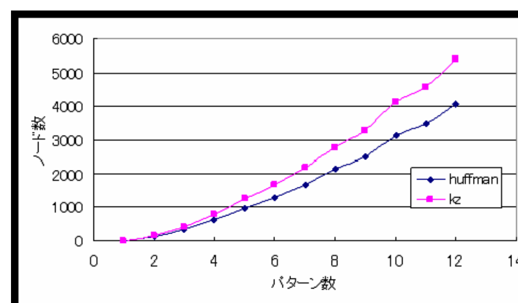
(a) English / No DC



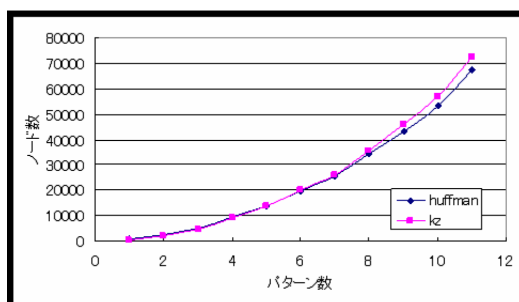
(b) Protein / No DC



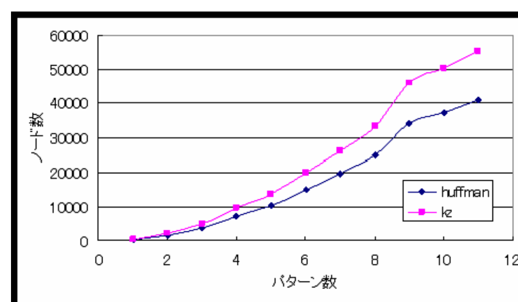
(c) English / 1DC



(d) Protein / 1DC



(e) English / 2DC



(f) Protein / 2DC

図 5.4: KZ 符号とハフマン符号のノード数比較

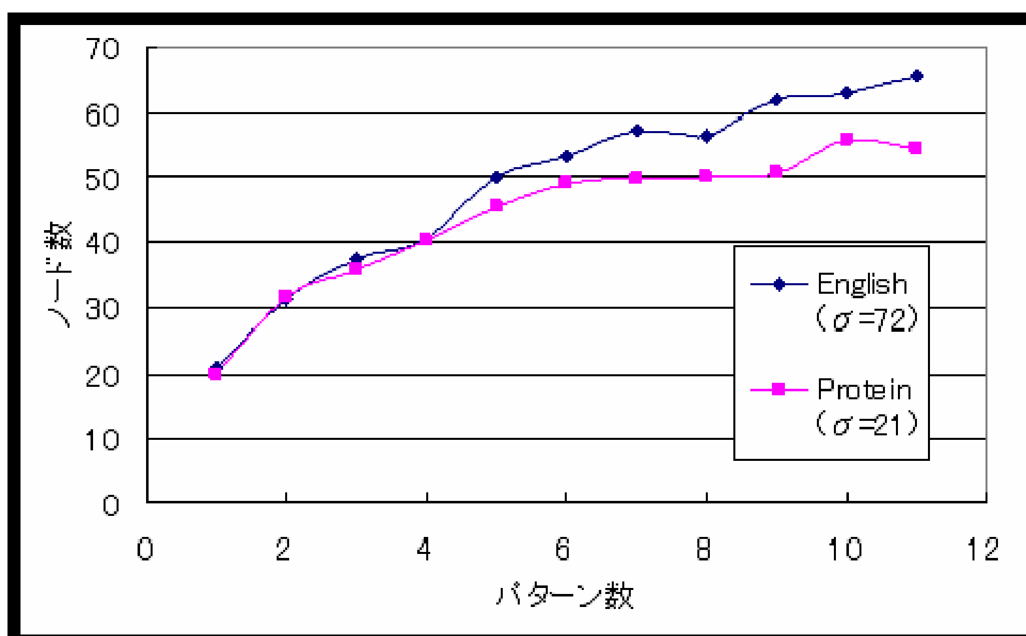


図 5.5: KZ 符号に基づく DC 文字を表現するために必要なノード数

5.4　実験に対する考察

理論的には前節までの定理によってノード数の観点から提案手法が優れていることが示されていたが，この実験から実用上も提案手法が優れていることが示された．ノード数のみから評価した場合は最小化した DFA がもっともノード数が少なかったが，DFA の最小化のために大きなコストを要するため，その点から見ても提案手法は優れている．

提案手法の優位性は DC 文字が出現した位置より前に出現した文字の種類がアルファベットサイズより小さいということに担保されているため，アルファベットサイズとくらべ，パターン数やパターン長が大きい場合，提案手法によるメリットは薄れる．

更に，エントロピー符号化後のビット列に対する検索オートマトンでもこの優位性は担保される．

第6章

結論

本研究において，DC パターンを含むパターン集合に対する全文検索を，領域計算量を抑えて実行するアルゴリズムを提案・実装し，その有用性を実験によって確かめた．さらに，このアルゴリズムが圧縮文字列検索に応用できることを示した．

特に本論文で提案した瞬時復号可能化した KZ 圧縮符号を用いて圧縮文字列検索をする場合，Huffman 圧縮を用いた場合と比べ計算速度は遅くなるものの，オートマトン上の DC 文字を少ないノード数で示すことが出来る点で有用性が極めて高い方式であるといえる．提案手法においては，DC 文字の出現位置より前に出てきた文字を他の文字と区別することで AC オートマトンの整合性を保つことに成功しているものの，区別する文字の数を減らすことで，ノード数を更に減らすことが期待できる．これは KZ 圧縮を用いるときには特に効果が高いと予想される．

少ないメモリで検索が実行できるため，パターンやアルファベットの選び方によっては，キャッシュよりも小さいサイズでオートマトンを表現できる場合も考えられ，その場合は高速化が期待されるが，未確認である．

また，本論文ではKZ圧縮を取り上げたが，Stopper Encoding [16] のような文字の切れ目が分かる他の圧縮方式でも，本論文で提案したアルゴリズムを適用可能であることが予想される．さらに，柴田らから BM 法に基づく検索オートマトンの構築法 [18] も既に提案されており，これに対しても同様のアプローチが可能であるかを調べることも，今後の課題である．

謝辞

本研究を行うにあたり，直接熱のこもった御指導をして頂きました篠原歩教授に，心より御礼申し上げます．また，御専門のお立場から的確なご助言をくださいました橋本和夫教授，堀口進教授に心から御礼申し上げます.

　最後に，日頃の研究室での活動全般にわたりご支援頂いた篠原研究室の皆様に心から感謝申し上げます.

参考文献

[1] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *communications of the ACM*, Vol. 18, No. 6, pp. 333–340, Jun. 1975.

[2] A. Amir, G. Benson, and M. Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *Journal of Computer and System Sciences.*, Vol. 52, pp. 299–307, 1996.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, Vol. 20, No. 10, pp. 62–72, 1977.

[4] Canterbury Copus. `URL: <http://corpus.canterbury.ac.nz/>`.

[5] DDBJ（日本ＤＮＡデータバンク）．`URL: <http://www.ddbj.nig.ac.jp/>`.

[6] M. Farach and M. Thorup. String-matching in Lampel-Ziv compressed string. *27th ACM STOC*, pp. 703–713, 1995.

[7] P. Ferragina and G. Manzini. Opportunistic data structure with applications. *FOCS*, pp. 390–398, 2000.

[8] 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ーゲノム情報の高速検索技法. 情報科学シンポジウム講演論文集, pp. 95–103, 1992.

[9] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings Of The I.R.E.*, Vol. 40, No. 9, pp. 1098–1101, Sep. 1952.

[10] W. H. Kautz. Fibonacci codes for synchronization control. *IEEE Trans. on Inf. Th.*, Vol. 11, pp. 284–292, Apr. 1965.

[11] D. E. Knuth. The Art of Computer Programing, Fundamental Algorithms. Vol. 1, , 1997.

[12] D. E. Knuth. The Art of Computer Programing, Fundamental Algorithms. Vol. 3, , 1997.

[13] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIMM J.comput*, Vol. 6, No. 2, pp. 323–350, 1977.

[14] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Technical Report LU-CS-TR:99-214*, 1999.

[15] 宮崎正路, 深町修一, 竹田正幸, 篠原武. 圧縮テキストに対するパターン照合機械の高速化. 情報処理学会論文誌, Vol. 39, No. 9, pp. 2638–2647, Sep. 1998.

[16] J. Rautio, J. Tanninen, and J. Tarhio. String Matching with Stopper Encoding and Code Splitting. *Proceeding of the 13th Annual Symposium on Combinatiorial Pattern Matching*, pp. 42–52, July 2002.

[17] C. E. Shannon. A mathematical theory of communictaion. *Bell Sys. Tech. Jour.*, Vol. 27, pp. 398–403, Jul. 1948.

[18] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore Type Algorithm for Compressed Pattern Matching. *Proc. 11th Annual Symposium on Combinatorial Pattern Matching, vol.1848 of Lecture Notes in Computer Science*, pp. 181–194, 2000.

[19] T.A. Welch. A Technique for High Performance Data Compression. *IEEE Comput.*, Vol. 17, pp. 8–19, 1984.

[20] J. Ziv and A. Lampel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inform. Theory*, Vol. IT-23, No. 3, pp. 337–349, 1977.