信州大学工学部

学士論文

Induced Sorting を用いた BW 変換の高速化に関する研究

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　11T2042B
氏名　佐藤 優輝

2015 年 3 月 23 日

目次

1 はじめに 1
- 1.1 研究の背景と目的 1
- 1.2 本論分の構成 1

2 BW 変換 1
- 2.1 可逆変換 2
- 2.2 BW 変換の性質 3
- 2.3 実装と計算量 3

3 Induced Sorting 4
- 3.1 接尾辞配列と BW 変換 4
- 3.2 Induced Sorting による接尾辞配列の構築 5
- 3.3 修正案 13

4 提案法の検証 13
- 4.1 入力ファイル別の特性 14
- 4.2 再帰の深さ別の特性 16

5 まとめ 18

謝辞 18

参考文献 18

付録 A ソースコード 20
- A.1 入力ファイルの BW 変換後の文字列を返すプログラム (提案法１) 20
- A.2 入力ファイルの BW 変換後の文字列を返すプログラム (提案法２) 24

1　はじめに

1.1　研究の背景と目的

膨大な量のデータを扱う現代の情報社会において，データを保存するための記憶領域の節約や，通信にかかるコストの削減が期待できるデータ圧縮技術は必要不可欠である．データ圧縮技術にも様々な方式が存在するが，本研究では BW 変換を用いたデータ圧縮法に注目した．BW 変換 (BWT，Burrows-Wheeler Transform) は，M. Burrows and D. J. Wheeler によって提案された文字列の可逆変換である．1994 年に発表された論文 [1] では，Block-sorting という名称で原理，及びアルゴリズムといくつかの実行結果がまとめられている．BW 変換は主に文字列を圧縮したい時に用いられる技術であるが，直接文字列を圧縮するわけではない．BW 変換を適用された文字列は，各文字の出現頻度はそのままながら同じ文字が連続して出現しやすいため，データ圧縮に向いた性質を持つ．これをデータ圧縮に適用するために，BW 変換後の文字列に Move-to-front coding(先頭移動法) を施し，小さい数字が出現しやすい数列に変換した後，ハフマン符号化または算術符号化をする方法が紹介されている [1]．また，BW 変換は全文検索やデータマイニングにも応用できることが知られ，広い分野で研究されている．ただし，BW 変換の基本アルゴリズムは実用上作業領域の確保やアルゴリズムの実行時間に難があり，このことは M. Burrows ら自身の論文 [1] でも指摘されている．

上記の問題点への対策として接尾辞配列を用いた方法が研究されており，その中でも優れた技術として，Induced Sorting と呼ばれるソート法で文字列の接尾辞配列を作り，それを利用して BW 変換をする方法が提案されている [2]．接尾辞配列の詳細については 3 章で述べる．本研究では Induced Sorting についてまとめられた論文 [3] を参考にこの方法を再現し，その特性を調べるとともに更なる高速化を目的とした改善案を示した．

1.2　本論分の構成

本論分は次のような構成をとる．2 章では BW 変換のアルゴリズムと性質，および実装について述べる．3 章では接尾辞配列を用いた BW 変換の実装法と Induced Sorting のアルゴリズムを説明し，本研究の提案法を示す．4 章では提案法の検証結果と考察を述べる．5 章では本研究のまとめを述べる．

2　BW 変換

データ圧縮において BW 変換は，文字列をより圧縮に向いた形に変換する役割を持つ．本章では BW 変換による文字列の変換と逆変換の基本アルゴリズムと BW 変換の性質，および

実装する上での注意点について述べる．

2.1 可逆変換

まずはじめに，文字列の変換アルゴリズムを示す．これ以降，入力文字列を S，文字列の長さを N とする．

1. 入力文字列 S を巡回シフトして，$N \times N$ の行列 M を作る．すなわち，$M[0,0],\ldots,M[0,N-1]$ は $S[0],\ldots,S[N-1]$ に，$M[1,0],\ldots,M[1,N-2],M[1,N-1]$ は $S[1],\ldots,S[N-1],S[0]$ に等しい．
2. 行列 M の各行を辞書順にソートする．
3. ソート後の行列 M の最後の列を出力文字列 L とする．すなわち，$L[0],\ldots,L[N-1]$ はソート後の $M[N-1,0],\ldots,M[N-1,N-1]$ に等しい．
4. ソート後の行列 M の中で元の文字列 S が現れている行番号を I とする．

入力文字列 S に対して，この変換は (L,I) のペアを出力する．例えば，入力文字列 $S=$shinshu に対して BW 変換を実行した時の出力ペアは，(L,I) = (sshiunh,4) である (図 1)．

行	行列 M						
0	s	h	i	n	s	h	u
1	h	i	n	s	h	u	s
2	i	n	s	h	u	s	h
3	n	s	h	u	s	h	i
4	s	h	u	s	h	i	n
5	h	u	s	h	i	n	s
6	u	s	h	i	n	s	h

辞書順ソート $\Longrightarrow$

行	行列 M						
0	h	i	n	s	h	u	s
1	h	u	s	h	i	n	s
2	i	n	s	h	u	s	h
3	n	s	h	u	s	h	i
4	s	h	i	n	s	h	u
5	s	h	u	s	h	i	n
6	u	s	h	i	n	s	h

(L,I) = (sshiunh,4)

図 1　BW 変換の実行例

次に，逆変換アルゴリズムを示す．出力されたペア (L,I) のみから元の文字列 S を完全に復元できることが，BW 変換の重要な性質である．

1. 逆変換する文字列 L を辞書順にソートし，文字列 F を得る．変換アルゴリズムにおける行列 M の作り方から，文字列 F はソート後の行列 M の最初の列であることが分かる．同時に，ソート後の行列 M の任意の行 i に注目すると，$F[i]$ と $L[i]$ は元の文字列 S において隣り合って (もしくは最初と最後に) 出現していることが分かる．

2. 逆変換する文字列 L の各文字が，辞書順ソートした文字列 F の何番目に対応しているかを調べる．この時，文字列 L の中に同じ文字が複数個含まれている場合は，その文字だけに注目した時の順序関係を保ったまま文字列 F に出現することが分かっている [1].
3. 変換アルゴリズムにおける出力ペア (L, I) の作り方から，復元したい文字列 S の最後の文字は $L[I]$ であることが分かる．従って，$L[I]$ から順に対応を追っていくことで元の文字列 S を復元できる．すなわち，$L[i]$ に対応する F の文字を $F[C[i]]$ とすると，$S[N-1] = L[I] = F[C[I]]$, 巡回シフトより $F[C[I]]$ の直前の文字は $L[C[I]]$ なので $S[N-2] = L[C[I]] = F[C[C[I]]]$,... というように復元できる (図 2).

$$
\begin{array}{ccccccccc}
L & = & s_1 & s_2 & h_1 & \mathrm{i} & \mathrm{u} & \mathrm{n} & h_2 \\
 & & & \uparrow & & & & & \uparrow \\
F & = & h_1 & h_2 & \mathrm{i} & \mathrm{n} & s_1 & s_2 & \mathrm{u}
\end{array}
$$

図 2　文字列の復元方法．u の前の文字は h, h の前は s, ... と追っていくことで, S =shinshu を復元できる．(下付の数字は，同じ文字の中での順位を表している．)

2.2　BW 変換の性質

BW 変換された文字列には同じ文字が連続して出現しやすく，このような現象が起こる理由には文脈が関係している．例えば英語の文脈では ‘he’ の前には ‘t’ が出現しやすく，これは変換アルゴリズムの行列 M において，‘he’ で始まる行の最後の文字が ‘t’ になりやすいことを意味する．よって，ソートされ ‘he’ で始まる行が並んだブロックの最後の列には ‘t’ が連続して出現しやすくなる．別の言い方をすれば，入力文字列の k 次経験エントロピーが低い場合に BW 変換後の文字列は同じ文字が連続して出現しやすくなる．

ここまでは入力がテキストファイルであると想定して話を進めてきたが，BW 変換はバイナリファイルに対しても適用することができる．コンピュータで用いられている ASCII コードでは，1 文字を 1 バイトで表現している．従って，1 バイトを基本単位として処理を行えば入力ファイルの種類に関係なく BW 変換をすることができる．

2.3　実装と計算量

BW 変換を実装する上で注意すべき点を述べる．先に示した基本アルゴリズムでは入力長 N に対して N^2 のサイズの行列を作るため，N の大きさによっては非常に多くの作業領域が必要になってしまい，メモリが足りなくなる恐れがある．これを回避する手段としてまず考え

られるのは，巡回シフトして得られた各文字列の先頭文字の，入力文字列 S の中での位置情報のみを記憶する方法である．つまり，行列 M の i 行目と j 行目の比較動作は，入力文字列 S の $S[i]$ から $S[i+N-1 \bmod N]$ までと $S[j]$ から $S[j+N-1 \bmod N]$ までとを比較することに等しい．

ただし，上記の方法を用いたとしても，辞書順にソートする際の計算量，すなわち比較回数が問題となってくる．例えば，高速なソート法として知られているクイックソートを用いれば文字列の比較回数は $O(N \log N)$ であるが，一部の最悪なケース (全てが同じ文字で構成されている文字列 $S=$aaa... など) の場合は 2 つの文字列の比較に毎回 N 回の文字の比較が必要なため，全体で $O(N^2 \log N)$ 回比較を行うことになる．これは N の大きさによっては非常に多くの比較処理が必要となり，それだけアルゴリズムの実行時間が長くなってしまうことに繋がる．これらの問題点への対策を次章で述べる．

3 Induced Sorting

与えられた文字列から行列を作る代わりに，接尾辞配列という概念を用いて BW 変換動作を実現できることが知られている [2]．ただし，その接尾辞配列の構築にもソート動作が必要であり，長い文字列にも対応できるように高速で処理できるアルゴリズムが望まれる．Induced Sorting は $O(N)$ の時間で接尾辞配列を構築できるアルゴリズムであり，Ge Nong らによって 2011 年に発表された [3]．

本章では，はじめに接尾辞配列の概念と BW 変換への応用を述べた後に Induced Sorting のアルゴリズムを紹介し，最後に実行時間に注目した場合の修正案を述べる．

3.1 接尾辞配列と BW 変換

文字列 S の任意の i 番目から最後までの文字を切り取った部分文字列を S の接尾辞と言い，$\mathrm{suf}(S,i)$ で表す．接尾辞配列とは，全ての接尾辞を辞書順に並べ替えた時の位置情報を格納した配列である．その例を図 3 に示した．

本研究において接尾辞配列を作る際は，文字列の最後に他のどの文字よりも辞書順の小さい特殊文字 '\$' を付け加える．これより先，'\$' を含む長さを N とする．本章で説明する Induced Sorting ではこの '\$' があることにより，効率的に接尾辞配列を構築することができる．任意の接尾辞 $\mathrm{suf}(S,i)$ において，その先頭文字の文字列 S の中での位置 i を，$\mathrm{suf}(S,i)$ の開始位置と呼ぶことにする．接尾辞配列を構成する情報は各接尾辞の開始位置 $0,\ldots,N-1$ を並べ替えた数列であるため，接尾辞配列は $N \log N$ バイトで表現できる．

接尾辞配列を SA とすれば，$S[SA[i]]$ は 2 章の変換アルゴリズムにおける $M[i][0]$ に等しい．

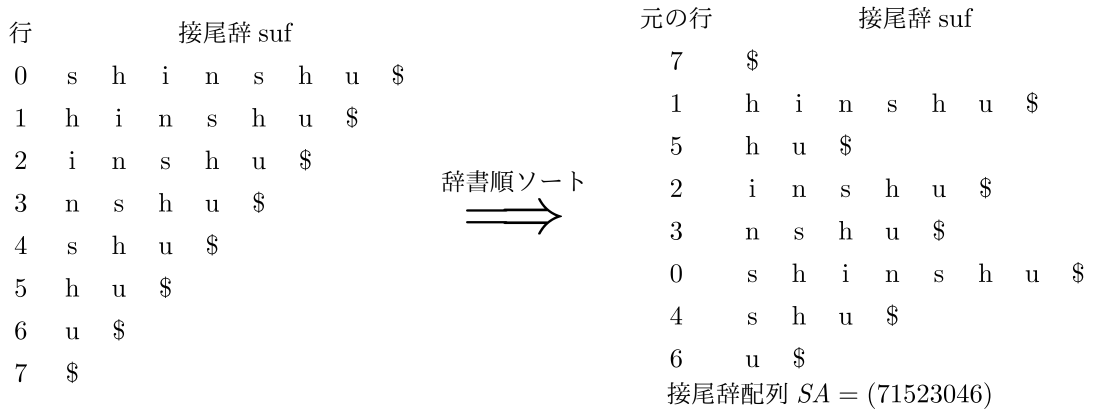

図 3　接尾辞配列構築の例

従って，BW 変換動作は

$$L[i] = \begin{cases} S[SA[i]-1] & i \neq 0 \\ S[N-1] & i = 0 \end{cases}$$

で実現できる．

3.2　Induced Sorting による接尾辞配列の構築

3.2.1　接尾辞のタイプ

Induced Sorting のアルゴリズムを説明するにあたり，各接尾辞のタイプを次のように定義する．

定義 1 すべての接尾辞を，次のようにタイプ分けする．

- Type-S: $\mathrm{suf}(S,i) < \mathrm{suf}(S,i+1)$ を満たす接尾辞．　または，$\mathrm{suf}(S,N-1) =$‘$’.
- Type-L: $\mathrm{suf}(S,i) > \mathrm{suf}(S,i+1)$ を満たす接尾辞．

ただし，不等号は辞書式順序の大小関係を表す．

例えば，$S=$shinshu$において，$\mathrm{suf}(S,0) =$ shinshu$ $>$ $\mathrm{suf}(S,1) =$ hinshu$ であるため，$\mathrm{suf}(S,0)$ は Type-L に分類される．また，$\mathrm{suf}(S,i)$ の先頭文字 $S[i]$ に対してもタイプ分けを定義し，$\mathrm{suf}(S,i)$ と同じタイプを割り当てるとする．例を図 4 に示した．

これらのタイプは，文字列 S を後ろから一度だけ走査することで分類できる．すなわち，

- $\mathrm{suf}(S,N-1)$ は Type-S.
- 次のどちらかを満たすなら，$\mathrm{suf}(S,i)$ は Type-S.

1. $S[\mathrm{i}] < S(i+1)$ である．
2. $S[\mathrm{i}] = S(i+1)$ かつ，$\mathrm{suf}(S, i+1)$ が Type-S である．

- 次のどちらかを満たすなら，$\mathrm{suf}(S, i)$ は Type-L.
  1. $S[\mathrm{i}] > S(i+1)$ である．
  2. $S[\mathrm{i}] = S(i+1)$ かつ，$\mathrm{suf}(S, i+1)$ が Type-L である．

とすれば良い．そして，このタイプ分けにより次が言える．

補題 2 Type-S である接尾辞 $\mathrm{suf}(S, i)$ と Type-L である接尾辞 $\mathrm{suf}(S, j)$ について，先頭の文字が同じであった場合，辞書順では常に $\mathrm{suf}(S, j)$ が先に来る．すなわち，

$$S[i] = S[j] \Rightarrow \mathrm{suf}(S, i) > \mathrm{suf}(S, j)$$

が成り立つ．

証明　$\mathrm{suf}(S, i)$ と $\mathrm{suf}(S, j)$ の先頭文字を，$S[i] = S[j] = c$ と置く．この 2 つの接尾辞の辞書順は，先頭文字が同じであるため 2 文字目以降である $\mathrm{suf}(S, i+1)$ と $\mathrm{suf}(S, j+1)$ との間での比較結果により決定される．

まず，$\mathrm{suf}(S, i)$ は Type-S より $\mathrm{suf}(S, i) < \mathrm{suf}(S, i+1)$ なので，$\mathrm{suf}(S, i+1)$ は先頭が c より大きな文字，もしくは c が 1 回以上連続した後に c より大きな文字が続く．

一方，$\mathrm{suf}(S, j)$ は Type-L より $\mathrm{suf}(S, j) > \mathrm{suf}(S, j+1)$ なので，$\mathrm{suf}(S, j+1)$ は先頭が c より小さな文字，もしくは c が 1 回以上連続した後に c より小さな文字が続く．

従って，常に $\mathrm{suf}(S, i+1) > \mathrm{suf}(S, j+1)$ であるため，$\mathrm{suf}(S, i) > \mathrm{suf}(S, j)$ となる．　□

このとき，接尾辞配列において共通の先頭文字 c を持つ接尾辞の並んだ部分を c のバケットと呼ぶことにする．これにより，各文字のバケットには，最初に Type-L の接尾辞が並んだ後に Type-S の接尾辞が並ぶことが分かる．

最後に，Type-S の特別な場合を次のように定義する．

定義 3 Type-S の接尾辞 $\mathrm{suf}(S, i)$ のうち，$\mathrm{suf}(S, i-1)$ が Type-L であるものを特に，Type-S^*とする．

接尾辞のタイプを順に並べると，Type-S の固まりと Type-L の固まりが交互に並び，Type-S^*は連続する Type-S の先頭に位置する (図 4).

以上のタイプ分けを利用して，Induced Sorting は次の手順で接尾辞配列を構築する．

- Step0 : 各接尾辞のタイプを判別する．
- Step1 :Type-S^*の接尾辞をソートする．
- Step2 :Type-S^*の順序から，Type-L の位置を決定する．

i	0	1	2	3	4	5	6	7
$S[i]$	s	h	i	n	s	h	u	$
Type	L	$S*$	S	S	L	$S*$	L	$S*$

図 4　タイプ分けの例

- Step3 :Type-L の位置から，Type-S の位置を決定する．

各ステップの実行時間は入力長 N に依存し，全て $O(N)$ である．これよりどのようにして各ステップを実行するかの説明に入るが，先に Step1 までが完了したとして Step2 以降について延べ，最後に Step1 を説明する．図 5 に Step2 および Step3 の実行中の例を示した．

3.2.2　Step2:Type-L の接尾辞の位置の決定

ソートされた Type-$S*$の接尾辞を元に，接尾辞配列における Type-L の接尾辞の位置を決定する．まずはじめに，入力文字列 S 中に出現する各文字の出現回数を調べておく．'$' を含む文字列のアルファベットサイズを σ としたときに，$bkt[c]$ が S 中の c の出現回数となるような大きさ σ の配列を bkt とすれば，接尾辞配列の各バケットの先頭位置を $bkt[0]$ から $bkt[c-1]$ の総和で，終端位置を $bkt[0]$ から $bkt[c]$ の総和で求めることができる．次に，ソートされた Type-$S*$の接尾辞を先頭から順に調べ，各接尾辞の開始位置をその先頭文字のバケットに右詰めで格納する．このとき，接尾辞配列 SA の各要素は先に -1 などで初期化しておく．これは Step1 の結果を利用しているため，図 5 では Step1 の欄に記してある．

Type-$S*$の接尾辞が辞書順で接尾辞配列 SA に格納されたら，SA を先頭から順に調べていく．現在調べている要素 $SA[i]$ に格納されている値を n_i とすると，各 $\mathrm{suf}(S, n_i)$ について，$\mathrm{suf}(S, n_i-1)$ が Type-L であれば補題 2 より，$\mathrm{suf}(S, n_i-1)$ は $S[n_i-1]$ から始まる接尾辞の中の，現在決定していないものの中で最も小さいことが分かる．よって，$\mathrm{suf}(S, n_i-1)$ の開始位置を $S[n_i-1]$ のバケットの先頭に格納し，その後先頭位置をひとつ後ろへずらす．図 5 の例では，SA の先頭要素 $SA[0]$ には $\mathrm{suf}(S, 11)$ の開始位置が格納されている．$\mathrm{suf}(S, 11-1) = \mathrm{suf}(S, 10)$ のタイプは L であるため，$S[10] =$'v' のバケットの先頭に $\mathrm{suf}(S, 10)$ の開始位置を格納している．

接尾辞配列 SA の要素を先頭から調べている途中で，値が初期値もしくは 0 の場合や，$\mathrm{suf}(S, n_i-1)$ の Type が S の場合は調べる対象を次の要素に移す．なお，値がすでに決定した Type-L の接尾辞の開始位置であっても，その直前の接尾辞 $\mathrm{suf}(S, n_i-1)$ が Type-L であれば同様に決定できる．図 5 では，現在調べている要素を '?'，それにより決定した値を '!' を下に付けて表している．

SA を先頭から終端まで調べ終えたら，全ての Type-L の接尾辞の位置が SA に正しく格納

されている．走査は一度で良いため，実行時間は $O(N)$ である．

3.2.3 Step3:Type-S の接尾辞の位置の決定

Step2 で決定した Type-L の接尾辞を元に，接尾辞配列における Type-S の接尾辞の位置を決定する．Step2 と同様の原理であるが，今度は SA を終端から順に調べていく．$\mathrm{suf}(S, n_i)$ について，$\mathrm{suf}(S, n_i - 1)$ が Type-S であれば補題 2 より，$\mathrm{suf}(S, n_i - 1)$ は $S[n_i - 1]$ から始まる接尾辞の中の，現在決定していないものの中で最も大きいことが分かる．よって，$\mathrm{suf}(S, n_i - 1)$ を $S[n_i - 1]$ のバケットの終端に格納し，その後終端位置をひとつ前へずらす．接尾辞配列 SA の要素を調べている途中で，値がすでに決定した Type-S の接尾辞の開始位置であっても，その直前の接尾辞 $\mathrm{suf}(S, n_i - 1)$ が Type-S であれば同様に決定できる．

なお，Step3 では Type-S の接尾辞の位置が決定した際に，その格納先にすでに別の値が格納されている場合がある．これは Step2 のはじめに格納した Type-S*の接尾辞の開始位置が残っているからであるが，上書きしてしまって問題ない．

SA を終端から先頭まで調べ終えたら，全ての Type-S の接尾辞の位置が SA に正しく格納されている．走査は一度で良いため，実行時間は $O(N)$ である．

3.2.4 Step1 :Type-S*の接尾辞のソート

このように，Step1 を除いた各ステップは，ソートされた Type-S*の接尾辞があれば合計で $O(N)$ の時間で完了できる．残りは Step1 での Type-S*の接尾辞のソート時間だが，Induced Sorting の優れている点はこのソートも $O(N)$ の時間で実現できることにあり，そのためにアルゴリズムを再帰的に実行している．ここでは，その手順について説明する．

はじめに，Type-S*の接尾辞に注目して以下を定義する．

定義 4 次のような S の部分文字列を，S*部分文字列と呼ぶ．

- $S[i]$ と $S[j]$ が Type-S*であり，間に Type-S*の文字を含まない部分文字列 $S[i \dots j]$．
- 特殊文字 '\$' 自身．

定義 5 各 S*部分文字列をその中での辞書的順位に置き換えて，文字列 S に出現する順番に並べて得られた文字列を，短縮文字列 $s1$ と呼ぶ．また，短縮文字列 $s1$ の接尾辞配列を $SA1$ とする．

例えば，S =shinshuuniv\$の S*部分文字列は 'hinsh', 'huuni', 'iv\$', '\$' の 4 つであり，それぞれの S*部分文字列内での辞書的順位である '1', '2', '3', '0' に置き換えられる．よって，短縮文字列 $s1$ は，$s1 = 1230$ となる．ただし，同じ S*部分文字列が複数個存在する場合はそれらを同一と見なし，同じ辞書的順位を割り当てる．Type-S*の文字列のソート結果は，S*部分文字列をソートすることで求められる．

	i	0	1	2	3	4	5	6	7	8	9	10	11
	$S[i]$	s	h	i	n	s	h	u	u	n	i	v	$
Step0	Type	L	S	S	S	L	S	L	L	L	S	L	S
	$S*$		*				*				*		*

	バケット	$	h		i		n		s		u		v
Step1	$SA[i]$	11	1	5	−1	9	−1	−1	−1	−1	−1	−1	−1
Step2	$SA[i]$	11 ?	1	5	−1	9	−1	−1	−1	−1	−1	−1	10 !
		11	1 ?	5	−1	9	−1	−1	0 !	−1	−1	−1	10
		11	1	5 ?	−1	9	−1	−1	0	4 !	−1	−1	10
		11	1	5	−1 ?	9 ?	8 !	−1	0	4	−1	−1	10
		11	1	5	−1	9	8 ?	−1	0	4	7 !	−1	10
		11	1	5	−1	9	8	−1 ?	0 ?	4 ?	7 ?	6 !	10
		11	1	5	−1	9	8	−1	0	4	7	6 ?	10 ?
Step3	$SA[i]$	11	1	5	−1	9 !	8	−1	0	4	7	6	10 ?
		11	1	5 !	−1	9	8	−1	0	4	7	6 ?	10
		11	1	5	−1	9	8	3 !	0	4 ?	7 ?	6	10
		11	1	5	2 !	9	8	3 ?	0 ?	4	7	6	10
		11	1 !	5	2 ?	9 ?	8 ?	3	0	4	7	6	10
		11 ?	1 ?	5 ?	2	9	8	3	0	4	7	6	10

図 5　Induced Sorting のアルゴリズム実行例

定義 4 より，文字列 S の最後に付け加えられた特殊文字 '\$' はそれ単独で S*部分文字列を成すため S*部分文字列内での辞書的順位は常に最小であり，短縮文字列 $s1$ では '0' に変換され再び特殊文字の役割を果たす．

また，短縮文字列 $s1$ の長さを $n1$ とすると，次が成り立つ．

補題 6 短縮文字列 $s1$ の長さは，元の文字列 S の長さのたかだか半分である．すなわち，

$$n1 \leq \lfloor N/2 \rfloor$$

が成り立つ．

以上を用いて，Type-S*接尾辞のソートは，次の手順で実現できる．

1. 文字列 S を先頭から末尾まで一度調べ，調べた Type-S*の文字の位置を接尾辞配列 SA のその文字のバケットに後ろから順に格納する．
   この時，同じ Type-S*の文字が複数個あった場合，SA のバケットに後ろから格納する順番は単に S での出現順であるため，必ずしも辞書順にはなっていない．
2. Step2 の手順で Type-L の接尾辞の開始位置を格納する．
3. Step3 の手順で Type-S の接尾辞の開始位置を格納する．
   この段階で SA には，S*部分文字列のみに注目すると，重複しているものを除いて開始位置が正しく辞書順で並んでいる．
4. 短縮文字列 $s1$ を作り，$s1$ がすべて異なる文字で構成されているかを調べる．
5. 重複がなかった場合，S*部分文字列は SA に正しい辞書順で格納されているため，Step2 へ移る．
6. 重複があった場合，$s1$ を入力として 1 に戻り，再帰的に処理を行う．

上記手順はすべて $O(N)$ の時間で実行できる．再帰があった場合でも，補題 6 より文字列は再帰する度に 1/2 以下の長さになり，なおかつ再帰は一本道のため合計の実行時間は $O(N)$ である．

このようにして，Induced Sorting は $O(N)$ の時間で接尾辞配列を構築する．アルゴリズムをフローチャートにまとめたものを図 6 に示した．

3.2.5 Induced Sorting の作業領域

BW 変換の基本アルゴリズムでは作業領域がもうひとつの問題となっていたが，Induced Sorting の作業領域は $O(N \log N)$ バイトと $N \log \sigma$ バイトの和であり，$O(N^2)$ のような N の増加による急激な増加の心配はない．その内訳は次のようになっている．

まず，入力文字列 S は 1 文字 1 バイトを基本単位としているため，N バイトである．次に接尾辞のタイプを記録するために最大 $N/4$ バイトを使用する．このうち必ず使用する領域

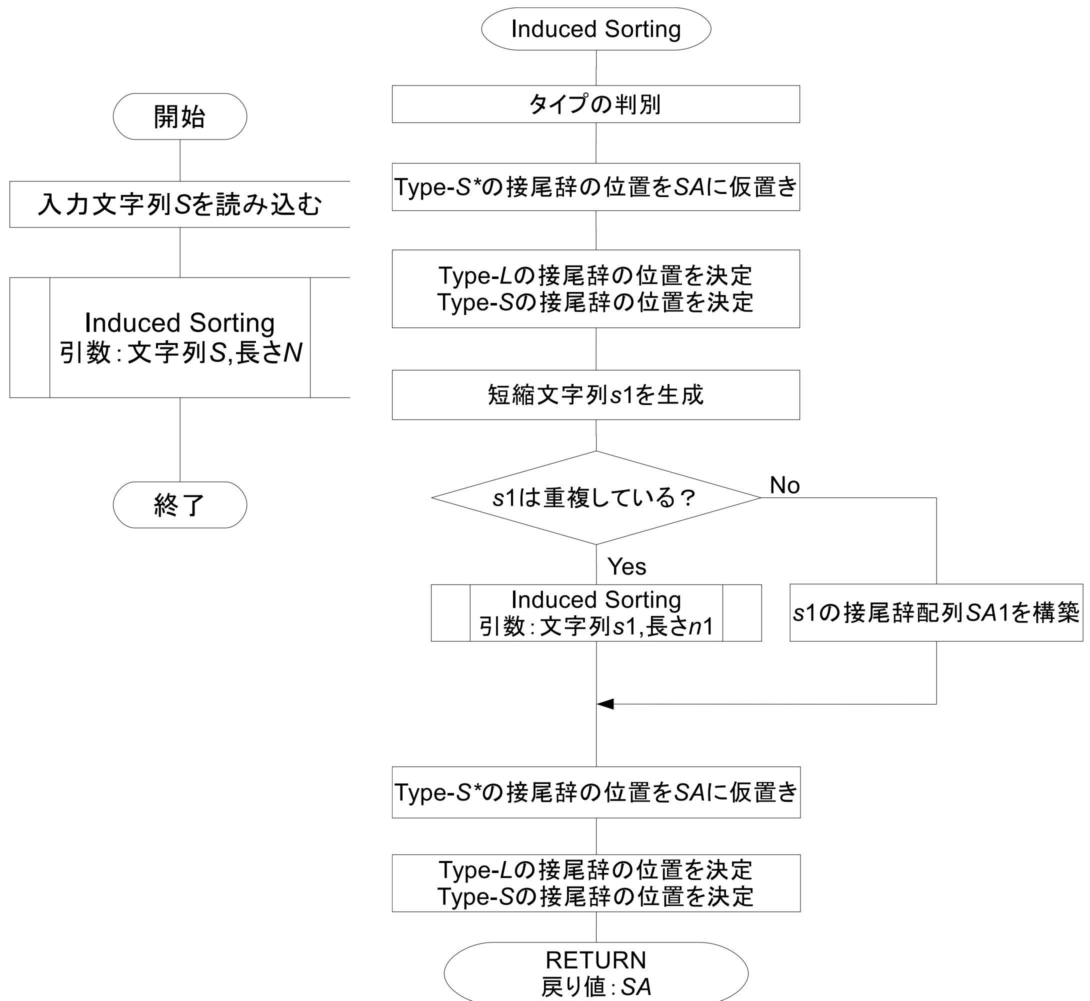

図 6　Induced Sorting のフローチャート

は，入力文字列 S のタイプを記録する N ビット $= N/8$ バイト である．各タイプについて，Type-L を '0' で，Type-S を '1' で表現し，Type-$S*$は定義 4 より，'1' が格納されている要素の中で直前の要素に '0' が格納されているものを探すことで区別している．これにより，1 つの接尾辞のタイプを 1 ビットで表現できる．再帰処理をした場合にはそれぞれの短縮文字列に対して同様の処理を行うが，補題 6 より再帰する度に長さは 1/2 以下になるため，合計で最大 $N/8$ バイトに収まる．これらを合わせて最大 $N/4$ バイトを使用する．また，接尾辞配列 SA は各要素に入る最大値が $N-1$ なので，$N \log N$ バイトを要する．SA は再帰した時でも同じ領域を再利用できるため，追加でメモリを取る必要はない (図 7)．

図 7　作業領域の再利用の例 (3 段目まで再帰した場合)．接尾辞配列 SA の領域を使いまわすことで，その後の処理に新たな領域を必要としない．

最後に，バケットの開始 (終了) 位置を求める時に用いる配列 bkt の使用領域は，文字列 S のアルファベットサイズ σ を用いて $N \log \sigma$ で表せる．ただし，短縮文字列のアルファベットサイズは σ より大きくなることがあるので，bkt は再帰した時にはメモリを開放して新たに取り直す．

なお，Induced Sorting を用いて最終的に BW 変換を行う場合は，BW 変換後の文字列 L を格納するために N バイトがさらに必要となる．

3.3　修正案

Induced Sorting を実行するコードは，Ge Nong らが公開している，論文 [3] のドラフト版 [4] に記載されている．本研究ではこのソースコードを BW 変換の実行プログラムに適用し，動作を高速化できるようアルゴリズムの実装法を修正した．以下にその修正点を記す．

修正案 1 バイナリファイルへの対応

Induced Sorting を実行するためには，入力文字列に特殊文字が入っていなければならない．しかし，特殊文字は入力文字列のどの文字とも異なっている必要があるため，00〜FF まですべて使用しているバイナリファイルには対応していない．

これに対して，特殊文字を含めた入力文字列の各文字は bkt に記録された出現回数で認識されるため，任意の文字 'c' の出現回数を $bkt[c+1]$ に記録し，$bkt[0]$ の値を 1 とすることで特殊文字が存在すると見せかけた．

修正案 2 接尾辞配列を作り直す手順の省略

図 7 に示した通り，ソースコードでは接尾辞配列 SA の領域を使いまわしているため，再帰の判断時に一度作った接尾辞配列を崩し，再帰がなければ作り直すという処理がある．実行時間の高速化を図るにはこの点は非効率であると考え，同じサイズの配列をもう一つ用意して使い分けることで処理を一部省略した．これにより増える作業領域は $N \log N$ バイトなので，オーダー自体に変化はない．図 8 に例を示した．

この案により省略できるのは，再帰の最下層での接尾辞配列を作り直す処理である．接尾辞配列の大きさは文字列の長さに合わせて再帰のたびに半分以下になっていくため，再帰が深いほど作り直しの手間が少なくなる．従って，再帰回数が少なく大きな接尾辞配列を構築している程効果が表れると予想できる．

上記修正案のうち，1 のみを適用して BW 変換に取り入れたプログラムを提案法 1，1 と 2 の両方を適用して BW 変換に取り入れたプログラムを提案法 2 と呼ぶことにとする．修正案 1 はファイル形式によらず正確な実行ができるように常に適用し，修正案 2 の有無による違いを検証する．

4　提案法の検証

複数の入力ファイルに対して入力長を次第に大きくした場合に，それぞれ提案法 1 と提案法 2 を実行した時の結果を比較し，実行時間の短縮率を調べた．その結果から，入力ファイルの違いによる Induced Sorting の性能の違いと提案法の効果を評価した．

実験には Calgary Corpus [5] を使用した．

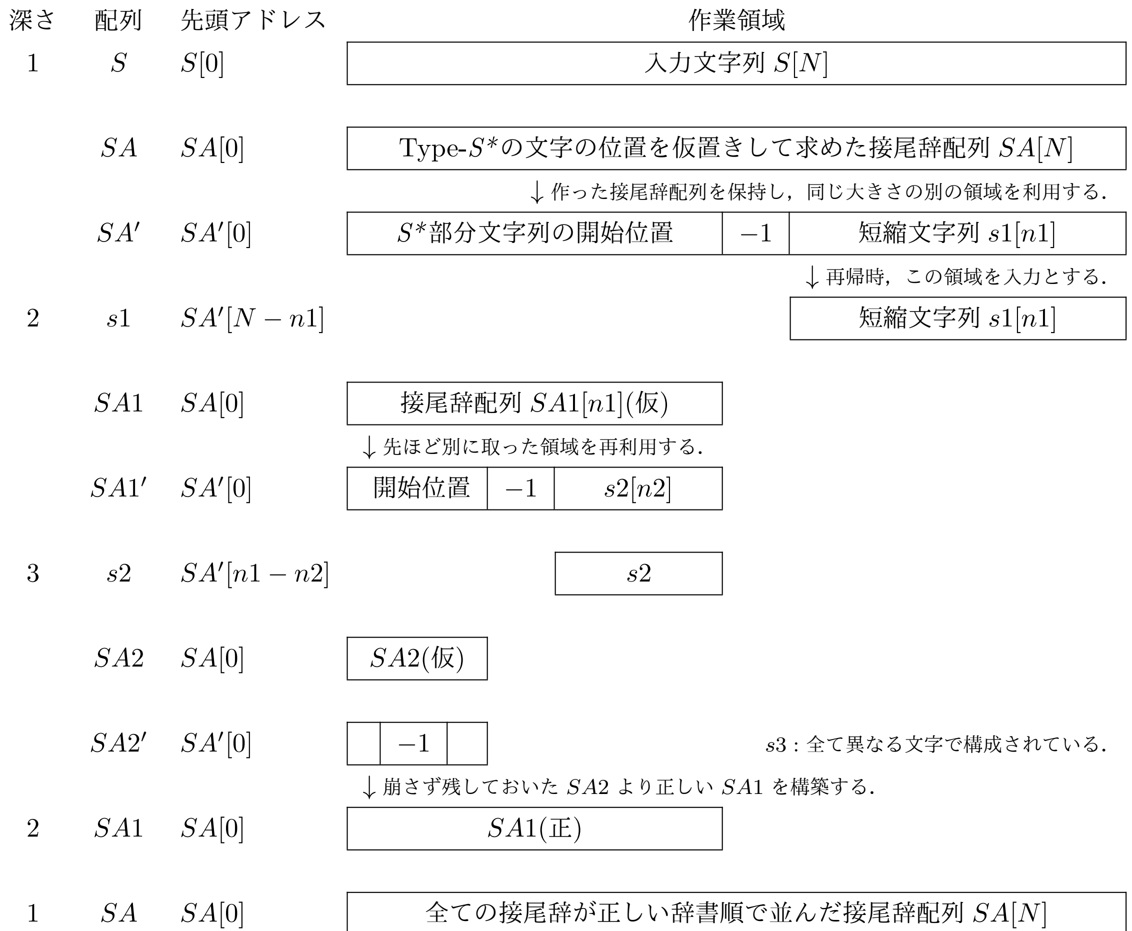

図 8　提案法による作業領域の再利用の例 (3 段目まで再帰した場合)．接尾辞配列を崩さず残しておくことで，再帰の最下層での処理を一部省略している．

4.1　入力ファイル別の特性

Calgary Corpus [5] の全 18 個のファイルに対して Induced Sorting を取り入れた BW 変換を繰り返し実行したところ，入力ファイルの種類によらず実行時間の平均値は入力長の増加に伴い線形的に増加し，Induced Sorting の実行時間が $O(N)$ であることが確認できた．提案法 1 と提案法 2 の実行結果を比較すると，どの場合でも提案法 2 の実行時間が提案法 1 の実

行時間を超えることはなく，

$$\text{短縮比率} = \frac{\text{提案法 1 の実行時間} - \text{提案法 2 の実行時間}}{\text{提案法 1 の実行時間}} \times 100[\%]$$

で定義した短縮比率は常に 0 以上だった．一例として全 18 個のファイルのうち，book1(テキストファイル), obj2(序盤にコメントがあるバイナリファイル), pic(バイナリファイル) の実行結果を図 9〜図 14 に示した．

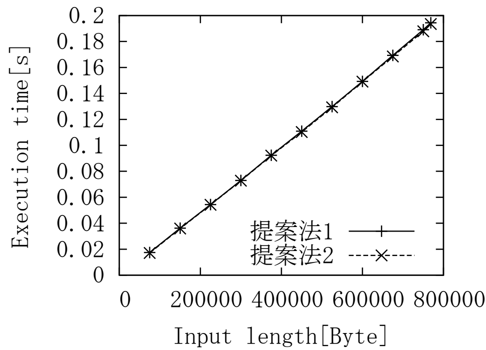

図 9　入力長-実行時間特性 (book1)

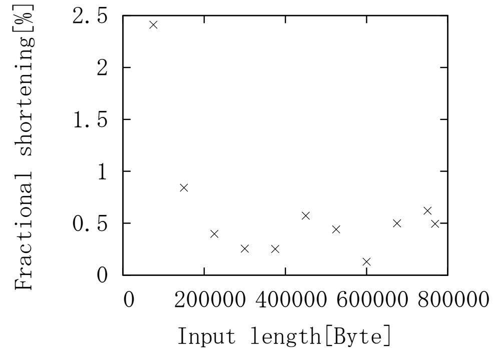

図 10　入力長-短縮比率特性 (book1)

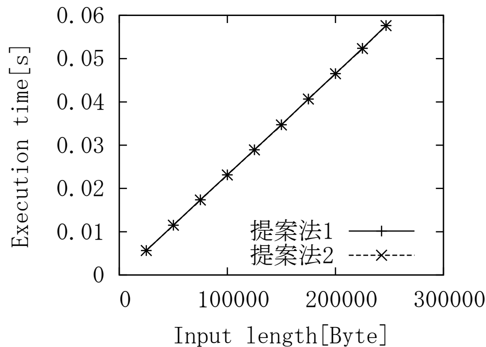

図 11　入力長-実行時間特性 (obj2)

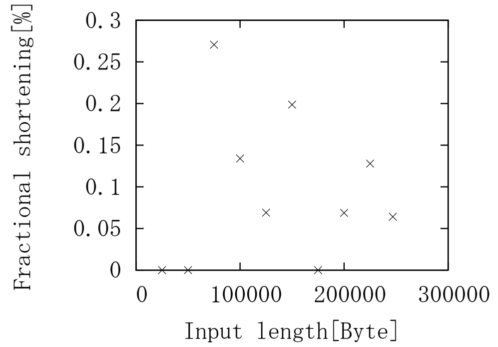

図 12　入力長-短縮比率特性 (obj2)

それぞれのファイルの入力長-実行時間特性を見るとどれも線形的に増加しているが，その傾きには book1 が約 2.54×10^{-7}, obj2 が約 2.34×10^{-7}, pic が約 1.60×10^{-7} と違いが見られた．特に pic の傾きは Calgary Corpus の他のファイルの傾き (平均約 2.33×10^{-7}) と比較しても際立って小さかった．これは，pic の文字の出現頻度が大きく偏っていることが原因と考えられる．pic は約 87.1[%] が ‘00’ で構成されているファイルである．ある 1 種類の文字が極

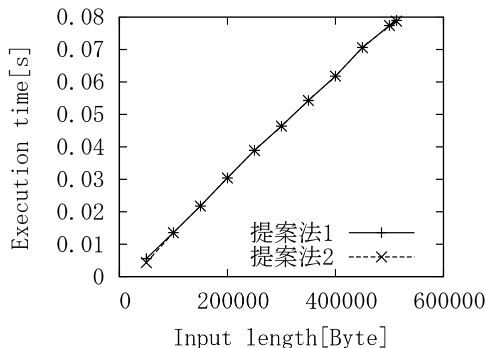

図 13　入力長-実行時間特性 (pic)

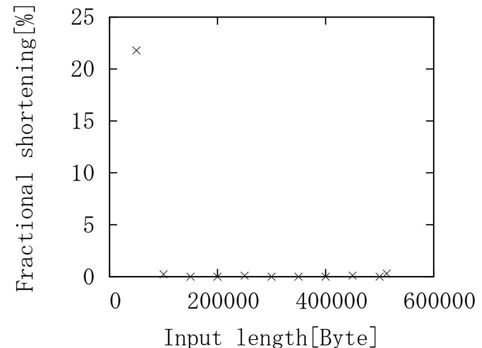

図 14　入力長-短縮比率特性 (pic)

端に多く出現する文字列を Induced Sorting すると，同じ文字が固まっている部分では最初のタイプ分けの際に同じタイプが連続することになり，Type-S^*の接尾辞が出現しにくい．従って短縮文字列が短くなりやすく，これにより再帰した後の処理が少なくなったと考えられる．

次に入力長-短縮比率特性に注目すると，obj2 は全体が 0[%] から約 0.27[%] の間にばらついていたが，book1 や pic では 1 部分だけ明らかに他より短縮比率が大きくなる様子が見られた．これは，同じファイルでも入力長により再帰深さが異なるため，深さが変わる入力長を境に短縮比率が大きく変化したためだと考えられる．いくつかの入力ファイルと入力長 N に対する再帰深さは，book1 では $N = 75000$ の場合に 3 でその他は 4，obj2 では全て 6，pic では $N = 50000$ 場合に 1 で $N = 100000$ で 4，その他で 5 だった．他よりも大きな短縮比率を示した条件を見ると，book1 は 75000，pic は 50000 であり，深さが浅い場合と一致している．なお，pic には深さが 4 と 5 の境が存在するが，$N = 100000$(深さ 4) での短縮比率が約 0.23[%] であるのに対し $N \geq 150000$(深さ 5) での短縮比率の平均が約 0.056[%] と，book1 の深さ 3 と 4 の場合の差ほど大きくはなかった．これより，予想通り深さが増すほど短縮比率は小さくなると考えられ，これは次節で示す深さ-平均短縮率特性からも確認できる．

4.2　再帰の深さ別の特性

入力ファイルの種類や入力長を変更することで，再帰の深さに違いが現れた．深さと実行時間の関係を調べるために，深さ別の入力長-短縮比率特性と深さ-平均短縮比率特性を求め，図 15〜図 19 に示した．実験した条件の中で深さが 1 となったパターンは 1 つしかなかったため，深さ 1 の時の入力長-短縮比率特性は省略した．また，深さが 2，および 7 以上となった場合はなかった．

深さ別の入力長-短縮比率特性 (図 15〜図 18) を見ると，入力長が短い間は比率にばらつき

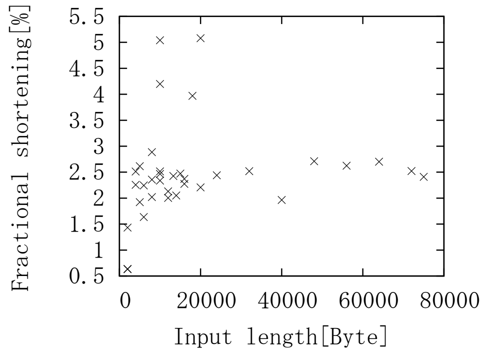

図 15　入力長-短縮比率特性 (深さ 3)

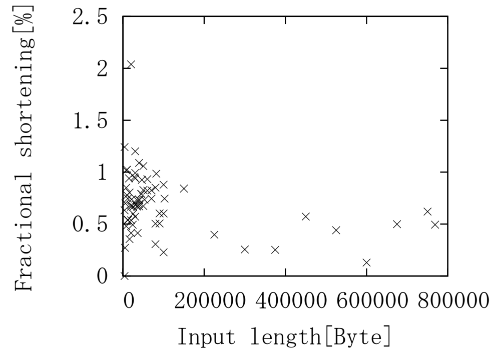

図 16　入力長-短縮比率特性 (深さ 4)

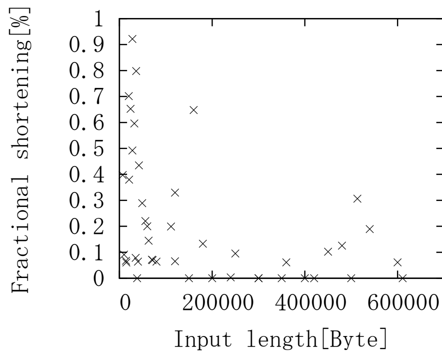

図 17　入力長-短縮比率特性 (深さ 5)

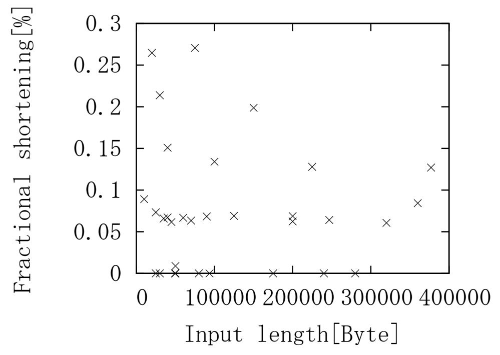

図 18　入力長-短縮比率特性 (深さ 6)

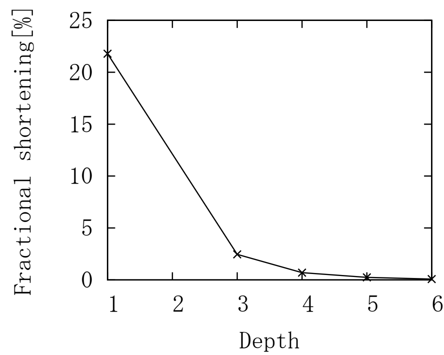

図 19　深さ-平均短縮比率特性

が見られるが，長くなるにつれてばらつきが小さくなることが読み取れる．これは，単純に入力長が短い条件での試行回数が多かった (長さの異なる 18 種類のファイルをそれぞれ約 10 等分して検証したため，入力が一番長い条件で正規化すると長さが短い部分に偏ることによる) ことと，入力長が短すぎると実行時間も非常に短くなり，わずかな差でも比率に大きく影響したためだと考えられる．例えば，実行時間が一番短くなった条件は obj1 の先頭 2000 文字を入力したときで，提案法 1 と 2 の平均実行時間はそれぞれ 349[μs] と 344[μs] だった．この場合，提案法 2 の平均実行時間がわずか 1[μs] ずれるだけでも短縮比率が約 0.29[%] 違ってくるため，実験環境の変化などによる測定誤差の影響を大きく受けてしまう．

深さ-平均短縮比率特性 (図 19) を見ると，再帰が深くなるにつれて短縮比率が小さくなることが読み取れ，再帰回数が少ないほど提案法 2 の効果が表れるという予想が裏付けられた．実際それゆえに，短縮比率は深さが増すほど 0 に漸近して小さくなり，深さ 6 では約 0.077[%] しかなく，入力ファイルによってはほとんど効果が得られない場合があると言える．

5　まとめ

本研究では，Induced Sorting を用いた BW 変換の動作の高速化を目指して，Induced Sorting の実装において，新たな作業領域を確保することで処理を一部省略できる修正を提案し，提案法の適用の有無による実行結果を比較して短縮率を調べた．また，その結果から Induced Sorting の特性を調べた．提案法を適用したプログラムはどの場合においても適用前より実行時間が長くなることはなく，提案法の有用性が確認できた．ただし，Induced Sorting の再帰が深くなるほど実行時間の差は小さくなり，ほとんど効果がなくなるケースも存在した．また，Induced Sorting は $O(N)$ 時間で接尾辞配列を構築するが，同じ長さであれば入力ファイルのアルファベットの出現頻度の偏りが大きいものが速く動作する傾向が見られた．

謝辞

本研究を行うにあたって，細かく指導してくださった指導教員の西新幹彦准教授に感謝の意を表する．

参考文献

[1] M. Burrows and D. J. Wheeler, “A Block-sorting Lossless Data Compression Algorithm,” SRC Research Report 124, Digital Systems Research Center, 1994.

[2] 岡野原大輔，高速文字列解析の世界，岩波書店，2012.

[3] Ge Nong, Sen Zhang, and Wai Hong Chan, “Two Efficient Algorithms for Linear Time

Suffix Array Construction," IEEE Trans. on Computers, vol. 60, no.10, pp.1471–1484, Oct. 2011.

[4] https://code.google.com/p/ge-nong/downloads/detail?name=Two%20Efficient%20Algorithms%20for%20Linear%20Time%20Suffix%20Array%20Construction.pdf&can=2&q= , 2015 年 1 月閲覧.

[5] Archive Comparison Test, http://compression.ca/act/act-files.html, 2015 年 1 月閲覧.

付録 A　ソースコード

A.1　入力ファイルの BW 変換後の文字列を返すプログラム (提案法 1)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

unsigned char mask[] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
// t[i] のタイプを返す処理.
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
// 各文字のタイプを定義する処理 (Type-L は 0,Type-S は 1).
#define tset(i, b) t[(i)/8]=(b)? (mask[(i) % 8] | t[(i) / 8]): ((~mask[(i) % 8])&t[(i) / 8])
#define chr(i) (cs==sizeof(int)? ((int*)s)[i]: ((unsigned char *)s)[i])
#define isLMS(i) (i>0 && tget(i) && !tget(i-1))

// 各バケットの開始位置または終端位置を求める関数.
void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
        int i, sum = 0;
        // バケットを初期化する.
        for (i = 0; i <= K; i++){
                bkt[i] = 0;
        }
        // 任意の文字‘c’ の出現数を，bkt[c+1] に格納する (bkt[0] には‘$’ の出現数である 1 を入れるため).
        for (i = 0; i < n - 1; i++){
                bkt[chr(i) + 1]++;
        }
        bkt[0] = 1;
        // 開始位置を求める場合 (end = false) は，bkt[i] に bkt[0],...,bkt[i-1] の総和を格納する.
        // 終端位置を求める場合 (end = true) は，bkt[i] に bkt[0],...,bkt[i] の総和を格納する.
        // これにより，文字‘c’ のバケットの開始位置は bkt[c+1]，終端位置は bkt[c+1]-1 で表せる.
        for (i = 0; i <= K; i++){
                sum += bkt[i];
                bkt[i] = end ? sum : sum - bkt[i];
        }
}

// Type-L の接尾辞の開始位置を格納する関数.
void induceSAl(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
        int i, j;
        // バケットの開始位置を求める.
        getBuckets(s, bkt, n, K, cs, end);
        for (i = 0; i < n; i++) {
                j = SA[i] - 1;
                if (j >= 0 && !tget(j)){
                        SA[bkt[chr(j) + 1]++] = j;
                }
        }
}

// Type-S の接尾辞の開始位置を格納する関数.
void induceSAs(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
        int i, j;
        // バケットの終端位置を求める.
        getBuckets(s, bkt, n, K, cs, end);
        for (i = n - 1; i >= 0; i--) {
        j = SA[i] - 1;
        if (j >= 0 && tget(j)){
                        SA[--bkt[chr(j) + 1]] = j;
                }
        }
}

// K 種類 (‘$’ を含めないで) の文字で構成された長さ n の文字列 s の接尾辞配列 SA を構築する関数 (Induced Sorting).
// 入力の条件は，s[n-1]=0 (最後の文字が‘$’) かつ n>=2 であること.
```

```
void SA_IS(unsigned char *s, int *SA, int n, int K, int cs, int* depth) {
	// 各文字のタイプをそれぞれ 1 ビットで表現するパート.
	unsigned char *t = (unsigned char *)malloc(n / 8 + 1);
	if (*t == NULL){
		printf("メモリエラー (%d)\n", __LINE__);
		return;
	}
	int i, j;
	//各文字のタイプを判断する.
	tset(n - 2, 0);		// ‘$’ の前の文字は必ず Type-L.
	tset(n - 1, 1);		// ‘$’ 自身は Type-S.
	for (i = n - 3; i >= 0; i--){
		tset(i, (chr(i) < chr(i + 1) || (chr(i) == chr(i + 1) && tget(i + 1) == 1)) ? 1 : 0);
	}

	// 短縮文字列 s1 を作るパート.
	// バケットを求める配列を用意する.
	int *bkt = (int *)malloc(sizeof(int)*(K + 1));
	if (*bkt == NULL){
		printf("メモリエラー (%d)\n", __LINE__);
		return;
	}
	// バケットの終端位置を求める.
	getBuckets(s, bkt, n, K, cs, true);
	// 接尾辞配列 SA を-1 で初期化する.
	for (i = 0; i < n; i++){
		SA[i] = -1;
	}
	// ‘$’ のバケットには n-1 が格納される.
	SA[--bkt[chr(n - 1)]] = n - 1;
	// 文字 s[i] が Type-S*であれば，s[i] のバケットに終端位置から順にインデックス i を格納する.
	for (i = 1; i < n - 2; i++){
		if (isLMS(i)){
			SA[--bkt[chr(i) + 1]] = i;
		}
	}
	// Type-L と Type-S の接尾辞の開始位置を格納する.
	induceSAl(t, SA, s, bkt, n, K, cs, false);
	induceSAs(t, SA, s, bkt, n, K, cs, true);
	free(bkt);
	// SA の最初の n1 個の要素に，並べ替えられた Type-S*の文字のインデックスを先頭から順に再格納する.
	// このとき，n1 は n の半分以下の長さになる.
	int n1 = 0;
	for (i = 0; i < n; i++){
		if (isLMS(SA[i])){
			SA[n1++] = SA[i];
		}
	}
	// S*部分文字列を探し，その辞書順に名付け直す.
	// SA[n1],...,SA[n-1] を初期化する.
	for (i = n1; i < n; i++){
		SA[i] = -1;
	}
	// S*部分文字列の辞書順を比較する.
	int name = 0, prev = -1;
	for (i = 0; i < n1; i++) {
		int pos = SA[i];
		bool diff = false;
		// S[pos] から始まる接尾辞と S[prev] から始まる接尾辞を先頭から 1 文字ずつ比較する.
		for (int d = 0; d<n; d++){
			// 違う文字，または同じ文字だがタイプが異なる場合は，2 つの S*部分文字列は異なる.
			if (prev == -1 || chr(pos + d) != chr(prev + d) || tget(pos + d) != tget(prev + d)){
				diff = true;
				break;
			// 同じ文字とタイプの組が d 個続き，s[pos+d]=s[prev+d] が Type-S*であれば,
			// 2 つの S*部分文字列は等しい.
			}else if (d>0 && (isLMS(pos + d) || isLMS(prev + d))){
				break;
			}
```

```
        }
        // 2 つの S*部分文字列が異なる場合は，別の名前（辞書順）を付ける．
        // ループ終了後は，name には異なる S*部分文字列の数が入っている．
        if (diff) {
                name++;
                prev = pos;
        }
        // SA[n1]〜SA[n-1] の範囲に，辞書順に名付け直された S*部分文字列を，s での出現順を保ったまま格納する．
        pos = (pos % 2 == 0) ? pos / 2 : (pos - 1) / 2;
        SA[n1 + pos] = name - 1;
}
// 上記で求めた S*部分文字列の短縮形を SA[n-1-n1]〜SA[n-1] に詰め直し，短縮文字列 s1 を得る．
for (i = n - 1, j = n - 1; i >= n1; i--){
        if (SA[i] >= 0) SA[j--] = SA[i];
}

// 短縮文字列の接尾辞配列を構築するパート．
// 短縮文字列 s1 を構成する要素が独立でなければ再帰する．
int *SA1 = SA, *s1 = SA + n - n1;
// name が短縮文字列の長さ n1 より小さければ重複があるため，再帰する．
if (name < n1){
        *depth += 1;
        SA_IS((unsigned char*)s1, SA1, n1, name, sizeof(int), depth);
}
// name と n1 が等しければ S*部分文字列は独立なので，s1 の接尾辞配列を構築する．
else {
        for (i = 0; i < n1; i++){
        SA1[s1[i]] = i;
        }
}

// s1 の接尾辞配列を元に，s の接尾辞配列 SA を構築するパート．
// バケットを求める配列を用意する．
bkt = (int *)malloc(sizeof(int)*(K + 1));
if (*bkt == NULL){
        printf("メモリエラー (%d)\n", __LINE__);
        return;
}
// 全ての Type-S*の文字を，その文字のバケットに格納する．
// バケットの終端を探す．
getBuckets(s, bkt, n, K, cs, true);
// 短縮文字列 s1 の各文字を，短縮前の S*部分文字列の s での開始位置に対応させる．
for (i = 1, j = 0; i < n; i++){
        if (isLMS(i)){
                s1[j++] = i;
        }
}
// s1 と s の対応を元に，s の S*部分文字列の開始位置を s の S*部分文字列の辞書順に並べ替える．
for (i = 0; i < n1; i++){
        SA1[i] = s1[SA1[i]];
}
// SA の残りの要素を初期化する．
for (i = n1; i < n; i++){
        SA[i] = -1;
}
// S*部分文字列の開始位置を，その先頭文字のバケットに正しい辞書順に格納する．
for (i = n1 - 1; i >= 0; i--) {
        j = SA[i];
        SA[i] = -1;
        if (i == 0){
                SA[--bkt[chr(j)]] = j;
        }
        else{
                SA[--bkt[chr(j) + 1]] = j;
        }
}
// 正しい Type-S*の文字の順序を元に，Type-L と Type-S の接尾辞の位置を決定する．
induceSAl(t, SA, s, bkt, n, K, cs, false);
induceSAs(t, SA, s, bkt, n, K, cs, true);
```

```
	free(bkt); free(t);
}

// メイン関数 (入力ファイルの BW 変換後の文字列を返す関数)
// 引数は，ファイル名と入力長.
int main(int argc, char *argv[]){
	int size = atoi(argv[2]);
	// 入力文字列とその接尾辞配列，および BW 変換後の文字列を格納する配列を用意する.
	unsigned char* s = (unsigned char*)malloc(size);
	int* sa = (int*)malloc(sizeof(int)*size);
	unsigned char* L = (unsigned char*)malloc(size);
	if ((s == NULL) || (sa == NULL) || (L == NULL)){
		printf("メモリエラー (%d)\n", __LINE__);
		return 0;
	}
	int dep;
	int I;

	// 入力ファイルを size-1 文字読み込み，最後に‘$’ を追加する.
	FILE *fp = fopen(argv[1], "rb");
	if (fp == NULL){
		printf("open error\n");
		return -1;
	}
	if (fread(s, sizeof(char), size - 1, fp) < size - 1){
		printf("size error\n");
		return -1;
	}
	fclose(fp);
	s[size - 1] = '\0';

	// Induced Sorting を用いて文字列 s の接尾辞配列を構築する.
	dep = 1;
	SA_IS(s, sa, size, 256, sizeof(char), &dep);
	// s の BW 変換後の文字列 L と，初期文字列の位置 I を求める.
	int i;
	for (i = 0; i < size; i++){
		if (sa[i] == 0){
			L[i] = s[size - 1];
			I = i;
		}
		else{
			L[i] = s[sa[i] - 1];
		}
	}

	/*
	printf("\n 接尾辞配列 SA =\n");
	for (i = 0; i < size; i++){
		printf("%d ", SA[i]);
	}
	printf("\n");

	printf("\n 出力文字列 L =\n");
	for (i = 0; i < size; i++){
		printf("%02x ", L[i]);
	}
	printf("\n");
	*/

	/*
	FILE *fp1 = fopen("outSA1.txt", "wb");
	if (fp1 == NULL){
		printf("open error\n");
		return -1;
	}
	fwrite(sa, sizeof(char), size, fp1);
	fclose(fp1);
	*/
```

```
	FILE *fp2 = fopen("outL.txt", "wb");
	if (fp2 == NULL){
		printf("open error\n");
		return -1;
	}
	fwrite(L, sizeof(char), size, fp2);
	fclose(fp2);

	FILE *fp3 = fopen("outI.txt", "wt");
	if (fp3 == NULL){
		printf("open error\n");
		return -1;
	}
	fprintf(fp3, "%d", I);
	fclose(fp3);

	free(s); free(sa); free(L);
	return 0;
}
```

A.2 入力ファイルの BW 変換後の文字列を返すプログラム (提案法２)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

unsigned char mask[] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
#define tset(i, b) t[(i)/8]=(b)? (mask[(i) % 8] | t[(i) / 8]): ((~mask[(i) % 8])&t[(i) / 8])
#define chr(i) (cs==sizeof(int)? ((int*)s)[i]: ((unsigned char *)s)[i])
#define isLMS(i) (i>0 && tget(i) && !tget(i-1))

// 各バケットの開始位置または終端位置を求める関数.
void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
	int i, sum = 0;
	for (i = 0; i <= K; i++){
		bkt[i] = 0;
	}
	for (i = 0; i < n - 1; i++){
		bkt[chr(i) + 1]++;
	}
	bkt[0] = 1;
	for (i = 0; i <= K; i++){
		sum += bkt[i];
		bkt[i] = end ? sum : sum - bkt[i];
	}
}

// Type-L の接尾辞の開始位置を格納する関数.
void induceSAl(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
	int i, j;
	getBuckets(s, bkt, n, K, cs, end);
	for (i = 0; i < n; i++) {
		j = SA[i] - 1;
		if (j >= 0 && !tget(j)){
			SA[bkt[chr(j) + 1]++] = j;
		}
	}
}

// Type-S の接尾辞の開始位置を格納する関数.
void induceSAs(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
	int i, j;
	getBuckets(s, bkt, n, K, cs, end);
	for (i = n - 1; i >= 0; i--) {
```

```
			j = SA[i] - 1;
			if (j >= 0 && tget(j)){
				SA[--bkt[chr(j) + 1]] = j;
			}
		}
}

// 文字列 s の接尾辞配列 SA を構築する関数 (Induced Sorting).
void SA_IS(unsigned char *s, int *SA,int *SAC, int n, int K, int cs, bool *bottom, int *depth) {
	unsigned char *t = (unsigned char *)malloc(n / 8 + 1);
	if (*t == NULL){
		printf("メモリエラー (%d)\n", __LINE__);
		return;
	}
	int i, j;
	tset(n - 2, 0);
	tset(n - 1, 1);
	for (i = n - 3; i >= 0; i--){
		tset(i, (chr(i) < chr(i + 1) || (chr(i) == chr(i + 1) && tget(i + 1) == 1)) ? 1 : 0);
	}

	// 短縮文字列 s1 を作るパート.
	int *bkt = (int *)malloc(sizeof(int)*(K + 1));
	if (*bkt == NULL){
		printf("メモリエラー (%d)\n", __LINE__);
		return;
	}
	getBuckets(s, bkt, n, K, cs, true);
	for (i = 0; i < n; i++){
		SA[i] = -1;
	}
	SA[--bkt[chr(n - 1)]] = n - 1;
	for (i = 1; i < n - 2; i++){
		if (isLMS(i)) SA[--bkt[chr(i) + 1]] = i;
	}
	induceSAl(t, SA, s, bkt, n, K, cs, false);
	induceSAs(t, SA, s, bkt, n, K, cs, true);
	free(bkt);

	// 並べ替えられた Type-S*の文字のインデックスを, 別の配列 SAC の最初の n1 個の要素に格納する.
	int n1 = 0;
	for (i = 0; i < n; i++){
		if (isLMS(SA[i])) SAC[n1++] = SA[i];
	}
	for (i = n1; i < n; i++){
		SAC[i] = -1;
	}
	// SAC[n-1-n1]〜SAC[n-1] に短縮文字列 s1 を作る.
	int name = 0, prev = -1;
	for (i = 0; i < n1; i++) {
		int pos = SAC[i];
		bool diff = false;
		for (int d = 0; d<n; d++){
			if (prev == -1 || chr(pos + d) != chr(prev + d) || tget(pos + d) != tget(prev + d)){
				diff = true; break;
			}
			else if (d>0 && (isLMS(pos + d) || isLMS(prev + d))){
				break;
			}
			if (diff) {
				name++; prev = pos;
			}
			pos = (pos % 2 == 0) ? pos / 2 : (pos - 1) / 2;
			SAC[n1 + pos] = name - 1;
		}
	}
	for (i = n - 1, j = n - 1; i >= n1; i--){
		if (SAC[i] >= 0){
			SAC[j--] = SAC[i];
```

```
            }
        }

        // 再帰の判断
        // 再帰無しの場合は，SA の Type-S*の文字が正しい順序で並んでいるため，そのまま利用できる．
        int *SA1 = SA, *s1 = SAC + n - n1, *SAC1 = SAC;
        if (name < n1){
                *depth += 1;
                SA_IS((unsigned char*)s1, SA1, SAC1, n1, name, sizeof(int), bottom, depth);
        }

        // s1 の接尾辞を元に，s の接尾辞配列 SA を構築するパート．
        bkt = (int *)malloc(sizeof(int)*(K + 1));
        if (*bkt == NULL){
                printf("メモリエラー (%d)\n", __LINE__);
                return;
        }
        // 再帰が最下層でなければ S*部分文字列の開始位置を，その先頭文字のバケットに正しい辞書順に格納する．
        if (!*bottom){
                getBuckets(s, bkt, n, K, cs, true);
                for (i = 1, j = 0; i < n; i++){
                        if (isLMS(i)) s1[j++] = i;
                }
                for (i = 0; i < n1; i++){
                        SA1[i] = s1[SA1[i]];
                }
                for (i = n1; i < n; i++){
                        SA[i] = -1;
                }
                for (i = n1 - 1; i >= 0; i--) {
                        j = SA[i];
                        SA[i] = -1;
                        if (i == 0){
                                SA[--bkt[chr(j)]] = j;
                        }
                        else{
                                SA[--bkt[chr(j) + 1]] = j;
                        }
                }
        }
        induceSAl(t, SA, s, bkt, n, K, cs, false);
        // 再帰が最下層の場合は，すでに Type-S(S*を含む) の接尾辞の開始位置は正しい位置に格納されているので省略．
        if (!*bottom){
                induceSAs(t, SA, s, bkt, n, K, cs, true);
        }
        else{
                *bottom = false;
        }
        free(bkt); free(t);
}

// メイン関数 (入力ファイルの BW 変換後の文字列を返す関数)
int main(int argc,char *argv[]){
        int size = atoi(argv[2]);
        unsigned char* s = (unsigned char*)malloc(size);
        // 接尾辞配列 SA と，作業用配列 SAC の領域をまとめて取る．
        int* sa = (int*)malloc(sizeof(int)*size * 2);
        int* sac = sa + size;
        unsigned char* L = (unsigned char*)malloc(size);
        if ((s == NULL) || (sa == NULL) || (L == NULL)){
                printf("メモリエラー (%d)\n", __LINE__);
                return 0;
        }
        bool btm;
        int dep;
        int I = 0;

        FILE *fp = fopen(argv[1], "rb");
        if (fp == NULL){
```

```
		printf("open error\n");
		return -1;
	}
	if (fread(s, sizeof(char), size - 1, fp) < size - 1){
		printf("size error\n");
		return -1;
	}
	fclose(fp);
	s[size - 1] = '\0';

	// Induced Sorting を用いて文字列 s の接尾辞配列を構築する.
	btm = true;
	dep = 1;
	SA_IS(s, sa, sac, size, 256, sizeof(char), &btm, &dep);
	// s の BW 変換後の文字列 L と, 初期文字列の位置 I を求める.
	int i;
	for (i = 0; i < size; i++){
		if (sa[i] == 0){
			L[i] = s[size - 1];
			I = i;
		}
		else{
			L[i] = s[sa[i] - 1];
		}
	}

	/*
	printf("\n 接尾辞配列 SA =\n");
	for (i = 0; i < size; i++){
		printf("%d ", SA[i]);
	}
	printf("\n");
	printf("\n 出力文字列 L =\n");
	for (i = 0; i < size; i++){
		printf("%02x ", L[i]);
	}
	printf("\n");
	*/

	/*
	FILE *fp1 = fopen("outSA2.txt", "wb");
	if (fp1 == NULL){
		printf("open error\n");
		return -1;
	}
	fwrite(sa, sizeof(char), size, fp1);
	fclose(fp1);
	*/
	FILE *fp2 = fopen("outL.txt", "wb");
	if (fp2 == NULL){
		printf("open error\n");
		return -1;
	}
	fwrite(L, sizeof(char), size, fp2);
	fclose(fp2);
	FILE *fp3 = fopen("outI.txt", "wt");
	if (fp3 == NULL){
		printf("open error\n");
		return -1;
	}
	fprintf(fp3, "%d", I);
	fclose(fp3);

	free(s); free(sa); free(L);
	return 0;
}
```