信州大学工学部

学士論文

文法圧縮におけるリダクションルールを削減するアルゴリズム

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　04T2100F
氏名　矢野 裕幸

平成 20 年 2 月 29 日

目次

1 序章

1.1 研究の背景と目的

データ圧縮は，大量のデータを扱うためのハードディスクの容量や，通信時のコストを削減するために重要である．データ列には，何度も使用される文字列が数多く存在するので，その文字列を別の変数に置換することでデータ量を圧縮できる．その一つに文法圧縮がある．文法圧縮は可逆圧縮の一つである．文法の大きさを小さくする手法として，Yang and Kieffer[1] が提案した 5 つのリダクションルールがある．文法圧縮に関しては，圧縮率は犠牲になるが使用メモリ量を減らすという省スペースな線形時間文法圧縮アルゴリズムの研究 [2] がこれまでになされている．

リダクションルールは大きく 3 つに分けることができる．生成規則の数を増やすルール 2 と 3，文字列の長さを短くするルール 4，そして生成規則を数を減らすルール 1 と 5 である．ここでルール 2,3,4 は圧縮するために欠かすことができない．ルール 1,5 は，ルール 2,3,4 によって圧縮された文法から必要ない生成規則を削除する後処理の役割を持つ．そのため，ルール 2,3,4 の組み合わせ方によってはルール 1,5 を必要としなくて済むのではないかと考えた．本研究では，ルール 2,3,4 の適用順を工夫することによってルール 1,5 を適用できる条件が出てこないようなアルゴリズムを提案しその正当性を検証することを目的とする．

1.2 本論文の構成

本論文では，次のような構成をとる．2 章では，文法について説明する．3 章では，アルゴリズムを提案する．4 章では，プログラムによる実装方法を示す．5 章では，提案法による評価結果を示す．6 章でまとめをする．

2 文法

本研究では，1 バイトのデータのことを終端文字と呼ぶ．終端文字とは別に変数と呼ばれる文字を用意し，終端文字または変数を単に文字と呼ぶ．文字の並びを文字列と呼ぶ．文法とは，変数から文字列への写像のことであり，記号 G で表す．

2.1 生成規則

生成規則 [3] とは，個々の変数 s_i と文字列 $G(s_i)$ の対応のことである．この関係を式 (1) のように表す．

$$s_i \to G(s_i) \qquad (i \geq 0) \tag{1}$$

この関係を「変数 s_i が文字列 $G(s_i)$ を指す」という．また，文字列 $G(s_i)$ に含まれる変数を生成規則によって再帰的に有限回変換することで終端文字列 α が得られるとき，「変数 s_i が終端文字列 α を意味する」という．本研究では，変数 s_0 がただ 1 つの終端文字列を意味するような文法を考える．

2.2 リダクションルール

コンピュータの中にあるデータを圧縮することを考える．まず，圧縮したいデータ列全体をメモリ上に展開して，変数 s_0 とデータ列全体を対応させ，この 1 つの生成規則から成る文法を作る．そして，s_0 が意味する終端文字列を変えずに，文法の大きさを小さく書き換える．これが文法変換による圧縮の基本である．ここで文法の大きさとは，生成規則の右辺にある文字列の長さの総和 $\sum_i |G(s_i)|$ である．

この文法の書き換え規則はリダクションルールと呼ばれ，本論文では省略する際には，単にルールと呼ばれる．Yang and Kieffer は 5 通りのルールを提案した．それらを以下に示す．

[ルール 1] 一度のみ現れる生成規則を削除する．

$$\begin{aligned} s_i &\to \alpha s_j \beta \\ s_j &\to \ \gamma \\ &\Downarrow \\ s_i &\to \alpha\gamma\beta \end{aligned}$$

この表記方法では，s_i のように s の右下に添え字が付く文字を変数とし，これ以外の文字は全て文字列とする．

[ルール 2] 1 つの生成規則から重複文字列を見つける．ここで β は 2 文字以上の文字列を表す．

$$\begin{aligned} s_i &\to \alpha_1\beta\alpha_2\beta\alpha_3 \\ &\Downarrow \\ s_i &\to \alpha_1 s_j \alpha_2 s_j \alpha_3 \\ s_j &\to \qquad \beta \end{aligned}$$

[ルール 3] 2 つの生成規則から重複文字列を見つける．ここで β は 2 文字以上の文字列を表す．

$$s_i \to \alpha_1\beta\alpha_2$$

$$
\begin{aligned}
s_j &\to \alpha_3\beta\alpha_4 \\
&\Downarrow \\
s_i &\to \alpha_1 s_k \alpha_2 \\
s_j &\to \alpha_3 s_k \alpha_4 \\
s_k &\to \quad \beta
\end{aligned}
$$

[ルール 4] 文字列の長さを短くする．ここで β は 2 文字以上の文字列を表す．

$$
\begin{aligned}
s_i &\to \alpha_1\beta\alpha_2 \\
s_j &\to \quad \beta \\
&\Downarrow \\
s_i &\to \alpha_1 s_j \alpha_2 \\
s_j &\to \quad \beta
\end{aligned}
$$

[ルール 5] 式 (1) より，2 つの生成規則の文字列 $G(s_i)$ が全く同じ終端文字列であるならば，片方の生成規則を削除する．

リダクションルール 1 から 5 のいずれも適用できないような文法を irreducible であるという．また，1 つの生成規則のみから成る文法からはじめて，ルール 1 から 5 を一定の手順で繰り返し適用することで，irreducible な文法を得るような変換を irreducible な文法変換という．さらに，定常エルゴード情報源に対し，その確率的性質をあらかじめ知らずに圧縮を行って，圧縮率が入力長に関して情報源のエントロピーレートに確率 1 で収束するような圧縮法をユニバーサルであるという．

Kieffer and Yang[4] は，irreducible な文法変換に関して次の 2 つの結果を示した．1 つは，文法の大きさ $|G|$ と文字列の長さ $|x|$ の比 $|G|/|x|$ は，文字列を限りなく長くすると，いかなる場合でも 0 に収束することである．言い換えると，

$$\max\{|G| : G(s_0) = x \text{ であるような irrebucible な文法, } x \text{ は長さ } n \text{ の終端文字列 }\} = O(n/\log n) \qquad (2)$$

が成り立つ．もう 1 つは irreducible な文法変換に基づく圧縮法はユニバーサルであることである．

3 アルゴリズムの提案

3.1 提案法

リダクションルール 1 と 5 を必要としないアルゴリズムを提案する．以下にその手順を示す．

1. ルール 2 または 3 を適用できるかを検索する．適用できる場所がなければ，終了する．
2. 重複部分の長い方のルールを適用する．
   長さが同じ場合はルール 2 を適用する．
3. これまでに作成された全ての生成規則に対して，ルール 4 を可能な限り適用する．
4. 手順 1 に戻る．

ルール 2,3,4 だけで，ルール 1,2,3,4,5 全てを使ったものと同じ能力が出せる．なぜなら，この提案法は irreducible な文法変換であるからである．そのことを以下で示し，また実際に検証する．

3.2 提案法の利点

リダクションルール 1 と 5 を必要としないことから，これらのルールが適用できる場所を検索する時間が不要となる．その結果，全体の処理時間を短縮できるといえる．

3.3 リダクションルール 1 と 5 が不要な理由

まず，ルール 5 が不要であることは，手順 3 におけるルール 4 の適用方法から証明できる．手順 2 よりできた新しい変数を s とする．手順 3 では $G(s)$ が他の変数の指す文字列の中にあれば，存在する限り書き換える．これより，手順 3 終了時では他の変数によって表されている文字列の中に $G(s)$ は現れない．提案法によって文法を作成すると，各変数が同じ終端文字列を意味することはないため，ルール 5 は不要となる．下にこの一例を示す．

$$
\begin{aligned}
s_0 \rightarrow\ & \alpha\beta\gamma 1 \alpha\beta\gamma 2 \alpha\beta\gamma 3 \\
s_1 \rightarrow\ & \beta\alpha\beta\gamma 4\alpha \\
& \Downarrow (\text{ルール } 2 \text{ を適用}) \\
s_0 \rightarrow\ & s_2 1 \alpha\beta\gamma 2 s_2 3 \\
s_1 \rightarrow\ & \beta\alpha\beta\gamma 4\alpha \\
s_2 \rightarrow\ & \alpha\beta\gamma \\
& \Downarrow (\text{ルール } 4 \text{ を適用}) \\
s_0 \rightarrow\ & s_2 1 s_2 2 s_2 3 \\
s_1 \rightarrow\ & \beta s_2 4\alpha \\
s_2 \rightarrow\ & \alpha\beta\gamma
\end{aligned}
$$

元々，$\alpha\beta\gamma$ という文字列が s_i と s_j の指す文字列の中にしか存在していないと仮定する．提案法により，s_k の指す文字列と別の生成規則の文字列は一致しないことがわかる．

次にルール 1 を考える．ルール 5 より生成規則が削除された結果，一度しか使われていない変数が現れる．

下にこの一例を示す.

$$
\begin{aligned}
s_0 &\rightarrow \quad \alpha\beta\gamma\beta\alpha \\
s_1 &\rightarrow \quad \alpha\beta s_3\alpha \\
s_2 &\rightarrow \quad \beta s_3 \\
s_3 &\rightarrow \quad \gamma\beta \\
&\Downarrow \text{(終端文字列に直す)} \\
s_0 &\rightarrow \quad \alpha\beta\gamma\beta\alpha \\
s_1 &\rightarrow \quad \alpha\beta\gamma\beta\alpha \\
s_2 &\rightarrow \quad \beta\gamma\beta \\
s_3 &\rightarrow \quad \gamma\beta \\
&\Downarrow \text{(ルール 5 を適用)} \\
s_0 &\rightarrow \quad \alpha\beta\gamma\beta\alpha \\
s_2 &\rightarrow \quad \beta\gamma\beta \\
s_3 &\rightarrow \quad \gamma\beta \\
&\Downarrow \text{(ルール 1 を適用)} \\
s_0 &\rightarrow \quad \alpha\beta\gamma\beta\alpha \\
s_2 &\rightarrow \quad \beta\gamma\beta
\end{aligned}
$$

ここで，変数 s_l は s_j と s_k の対応する文字列の中でしか存在しないと仮定する．ルール 5 を適用可能かどうかを確認するために終端文字列にすると s_i と s_j の対応する終端文字列が全く同じになる．これより，s_j の生成規則が削除され，s_l は s_k と対応する文字列の中でしか存在しなくなるため，s_l は一度しか使われない変数となる．最後にルール 1 を適用すると，生成規則は 2 つだけになる．

しかし，先に述べたようにルール 5 が不要であることがわかっているため，ルール 1 が必要になることはない．

したがって，提案法で得られる文法にはルール 1 から 5 のいずれも適用できないため，提案法は irreducible であるといえる．

4 実装

本提案法が irreducible であることを確認するために，アルゴリズムをプログラムによって実装する．

4.1 データ構造

このアルゴリズムを実装するために，生成規則の変数は配列のポインタ array[i] として用意し，文字列はメモリを動的確保して一文字ずつ格納する．ここで 1 つの配列の要素が 1 つのメモリの場所を指すことで生成規則を表現する．この生成規則の集合を文法として表現する．

終端文字は 1 バイトデータなので char 型で表現することができるが，文字は変数の場合もあるため char 型では表現できない．そこで，文字を int 型で表現することとする．このとき，文法の各生成規則の文字列は printf() では表示できないので，表示のための関数 printrule() を定義する．

char 型の値の範囲が 0 から 255 であることから int 型の 256 が文字列の末尾を表し，格納されている値 i が 257 以上のときは (i-256) 番目の変数を表すように設定している．printrule() は引数で与えられた変数 s が指す文字列を表示し，最後に 0*という表記をつけることで，視覚的に文字列の終わりを表す．文字列中の変数は，その後に*の記号をつけて，例えば，1 番目の変数は 1*などと表示される．終端文字であるならばそのまま表示し，改行文字等は文字コードを表示する．戻り値はない．

```
void printrule(int *s)
{
    int n;

    do {
        if (*s < 256){
            if (iscntrl((char)*s)){
                printf(" %02x", (char)*s);
            } else {
                printf("  %c", (char)*s);
            }
        } else {
            printf(" %d%c", *s - 256, '*');
        }
    } while (*s++ != 256);
    return;
}
```

次に変数が指す文字列の長さをカウントする関数 rulelen() を示す．この関数は，指定した変数 rule が指す文字列の長さを返す．この長さは動的メモリ確保する上で重要である．

```
int rulelen(int *rule)
```

```
{
    int num;

    for (num = 0; rule[num] != 256; num++){  /* 文字列の長さのカウント */
        ;
    }
    return(num);
}
```

文法の大きさを計算するプログラム部分を次に示す．各変数が指す文字列の長さの合計を変数 gram_len に格納することにより，どの程度文法圧縮されたかがわかる．

```
{
    int i, m = 0;
    int gram_len = 0;

    for(i = 0; array[i] != NULL; i++){
        m = rulelen(array[i]);
        gram_len += m;
    }
    printf("文法の大きさは %d である。\n", gram_len);
}
```

新しい生成規則を格納する場所を探す関数 empty_array() を示す．この関数は変数が指す文字列が NULL のとき，その変数を表す配列番号 i を返す．変数が指す文字列を順に見ていき，その文字列が NULL となれば，そこが次に使用する変数となる．

```
int empty_array()
{
    int i;
    int array_size;

    for(i = 1; i < array_size; i++){
        if(array[i] == NULL){
            break;
        }
    }
    return i;
}
```

変数が不足した時，追加確保する関数 increse_array() を示す．あらかじめ変数を有限個用意し，使い切れば変数の数を 2 倍に増やした新しい変数を用意する．各変数が指す文字列はそれぞれ新しい変数に格納し

直す.

```
void increse_array()
{
    int i;
    int **new_array;
    int array_size, new_array_size;

    new_array_size = array_size * 2;
    new_array = (int **) calloc(new_array_size, sizeof(int *));

    /* 新しい配列に古い配列をコピー */
    for(i = 0; i < array_size; i++){
        new_array[i] = array[i];
    }

    free(array);
    array = new_array;

    array_size = new_array_size;
    return;
}
```

4.2 各リダクションルールに関する処理

各リダクションルールに関する処理の実装を以下に示す.

4.2.1 ルール 1

本研究で提案するアルゴリズムではルール 1 は不要である. そのことを確認するための関数 check_no_rule1() を次に示す. この関数は変数 array[i] が指す文字列の中で使われている各変数の回数をカウントする. 各変数の使われている回数は 2 以上のはずである. 引数 output は, 各変数の回数を出力するファイルの名前である. 戻り値はない.

```
void check_no_rule1(char *output)
{
    int i, k;
    int count[100000];
    FILE *fp;

    fp = fopen(output, "w");
```

```
    for (i = 0; i < 100000; i++){
        count[i] = 0;
    }

    for(i = 0; array[i] != NULL; i++){
        for(k = 0; *(array[i] + k) != 256; k++){
            if(*(array[i] + k) > 256){
                count[*(array[i] + k) - 256] += 1;
            }
        }
    }

    for(i = 1; array[i] != NULL; i++){
        fprintf(fp, "%4d* : %2d 回\n", i, count[i]);
        if(count[i] < 2){
            fprintf(fp, "1 回のみ使われるルールが見つかりました\n");
        }
    }

    fclose(fp);
    return;
}
```

4.2.2 ルール 2 の検索

提案アルゴリズムではルール 2 とルール 3 のうち，より長い重複文字列が見つかった方を実際に適用する．したがって適用に先立って適用の候補となる場所を全て挙げておかなくてはならない．次に示す関数 same_pat2() は与えられた line が指す文字列に対して最も長い重複文字列を検索し，見つかった文字列の先頭位置を*x2，*y2，長さを*n に格納する．戻り値は重複文字列が見つかった場合は長さ，そうでない場合は-1 である．

ここでは文字列全体の長さに注目する．その文字列の中で，重複文字列が 2 箇所見つかるということは，この重複文字列の長さは最大でも文字列全体の長さの半分である．この文字列全体の半分の長さを距離 dist とし，文字列の先頭と dist だけ離れた文字から 1 文字ずつ照合する．ここで，dist はこの場合に見つかる重複文字の最大の長さでもあるため，dist を 1 ずつ小さくしていき，dist と見つかった重複文字列の長さが一致するか，もしくは dist が 2 になるまでこの処理を行う．同時に，先頭文字と文字列全体の長さから dist を引いた位置の文字からも照合することで全ての場合の照合を可能とする．重複文字列が見つかれば，その先頭文字の位置と長さを格納しておき，より長い重複文字列が見つかれば位置と長さを上書きする．

例として，banana123bana4banana という文字列から最も長い重複文字列 banana の検索方法を図 1 に示す．

```
                b a n a n a 1 2 3 b a n a 4 b a n a n a
dist 10   1)  *                 *                   → 無
          ------------------------------------------
dist  9   2)  *               *                     → bana
          3)  *                   *                 → 無
          ------------------------------------------
dist  8   4)  *             *                       → 無
          5)  *                     *               → ana
          ------------------------------------------
dist  7   6)  *           *                         → ana
          7)  *                       *             → 無
          ------------------------------------------
dist  6   8)  *         *                           → 無
          9)  *                           *         → banana（最長）
```

図 1　最も長い重複文字列の検索方法

記号 * のある位置から順番に照合していき，2 文字以上の重複文字列が見つかれば記録する．この重複文字列の長いものが見つかる度に上書きしていく．この文字列全体の長さは 20 であるため，dist の初期値は 10 となる．処理を続けていくと，dist が 6 のとき banana という 6 文字の重複文字列が見つかるので，これが最も長い重複文字列となる．

```
int same_pat2(int *line, int **x2, int **y2, int *n)
{
    int len = 0, i_len;
    int i_mem, dist_mem;
    int i, dist;

    for(dist = i_len / 2; dist > 1; dist--){
        for(i = 0; i < i_len - dist; i++){
            if(line[i] == line[dist + i]){
                len++;
                while (len > *n){
                    *n = len;          /* 重複部分の最大の長さ */
                    i_mem = i;         /* 重複文字の最後の場所 */
                    dist_mem = dist;   /* 2 箇所の重複部分の距離 */
                }
                if(len > dist)
                    break;
            } else {
```

```
                len = 0;
            }
            if(len == dist){
                break;
            } else if(i == i_len - dist - 1 && len > 1){
                ;
            }
        }
        if(*n == dist){
            break;
        }
        len = 0;
    }
    if(*n < 2){
        return (-1);
    }
    *x2 = &line[i_mem + 1 - *n];
    *y2 = &line[i_mem + 1 - *n + dist_mem];
    return (*n);
}
```

4.2.3 ルール 2 の適用

次の関数 reduct_2() はルール 2 の検索時に格納した重複文字列の先頭位置 x2，y2 と長さ n より，i 番目の変数が指す文字列の 2 箇所の重複部分を新しい j 番目の変数に置き換え，j 番目の変数が重複文字列を指すようにする．戻り値はない．

```
void reduct_2(int i, int j, int *x2, int *y2, int n)
{
    int k, l;
    int *s0, *s1;
    int i_len;

    i_len = rulelen(array[i]);
    s0 = (int *) calloc(i_len + 3 - 2 * n, sizeof(int));
    s1 = (int *) calloc(n + 1, sizeof(int));

    for(l = k = 0; l < i_len + 1; l++, k++){
        if (array[i] + l == x2 || array[i] + l == y2){
            s0[k] = 256 + j;
            l += n - 1;
```

```
        } else {
            s0[k] = *(array[i] + l);
        }
    }
    for(l = 0; l < n; l++){
        s1[l] = x2[l];
    }
    s1[n] = 256;

    free(array[i]);

    array[i] = s0;
    array[j] = s1;
    return;
}
```

4.2.4 ルール 3 の検索

次の関数 same_pat3() は 2 つの生成規則を力まかせに 1 文字ずつ比較していき，最長文字列を検索する．引数 i と j は任意の 2 つの生成規則の変数を指し，*x3 と*y3 にはそれぞれ見つかった重複文字列の先頭文字の位置が格納される．重複文字列が見つかった場合は，その長さを戻り値とし，見つからなかった場合は-1 を返す．

ルール 2 は各生成規則に対して検索すればよかったが，ルール 3 では生成規則の任意の 2 つの組合せ全てに対して検索しなければならない．ルール 2 の検索と同様に重複文字列が見つかれば，その先頭文字の位置と長さを記録し，より長い重複文字列が見つかる度に，位置と長さを上書きする．

```
int same_pat3(int i, int j, int **x3, int **y3)
{
    int pt, pp, pl;
    int i_len, j_len, samelen = 0;
    int max_pt, max_pp, maxl = 1;

    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);

    for(pl = 0; pl < i_len - j_len + 1; pl++){
        for(pt = pl, pp = 0; pt < i_len && pp < j_len; pt++, pp++){
            if(*(array[i] + pt) == *(array[j] + pp)){
                samelen++;
                if(maxl < samelen){
                    maxl = samelen;
```

```
                    max_pt = pt;
                    max_pp = pp;
                }
                if(pp == j_len - 1){
                    samelen = 0;
                }
            } else {
                samelen = 0;
            }
        }
    }
    if(maxl > 1){
        *x3 = array[i] + max_pt - maxl + 1;
        *y3 = array[j] + max_pp - maxl + 1;
        return(maxl);
    } else {
        return(-1);
    }
}
```

4.2.5 ルール 3 の適用

次の関数 reduct_3() はルール 3 を i 番目と j 番目の変数に対して検索した時に格納した重複文字列の先頭位置*x3, *y3 と長さ n より，重複部分を新しい k 番目の変数に置き換え，k 番目の重複文字列に関しては，新しく生成規則を作る．戻り値はない．

```
void reduct_3(int i, int j, int k, int *x3, int *y3, int n)
{
    int i_len, j_len;
    int *u0, *u1, *u2;
    int rl, rk;

    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);

    u0 = (int *) calloc(i_len - n + 2, sizeof(int));
    u1 = (int *) calloc(j_len - n + 2, sizeof(int));
    u2 = (int *) calloc(n + 1, sizeof(int));

    /* array[i] に関する部分 */
    if(j_len >= n){
```

```
        for(rl = rk = 0; rl < i_len + 1; rl++, rk++){
            if(array[i] + rl == x3){
                if(j_len == n){
                    u0[rk] = 256 + j;
                } else {
                    u0[rk] = 256 + k;
                }
                rl += n - 1;
            } else {
                u0[rk] = *(array[i] + rl);
            }
        }
    }
    /* array[j], array[k] に関する部分 */
    if(j_len > n){
        for(rl = rk = 0; rl < j_len + 1; rl++, rk++){
            if(array[j] + rl == y3){
                u1[rk] = 256 + k;
                rl += n - 1;
            } else {
                u1[rk] = *(array[j] + rl);
            }
        }
        for(rl = 0; rl < n; rl++){
            u2[rl] = y3[rl];
        }
        u2[n] = 256;

        free(array[j]);
        array[j] = u1;
        array[k] = u2;
    } else if(j_len == n){
        free(u2);
    }
    free(array[i]);
    array[i] = u0;
    return;
}
```

4.2.6 ルール 4 の検索

kmp_match() は j 番目の変数が指す文字列を i 番目の変数が指す文字列の中から検索する．j 番目の変数が指す文字列の長さは n である．戻り値は文字列が見つかった場合，先頭位置であり，そうでない場合は-1 である．

この関数の内部では文字列照合をするために KMP 法 [5] を用いる．KMP 法とは，テキスト文字列から照合文字列をうまく見つけ出して照合し直す位置を求め，なるべくパターン検索の移動を大きくしようというアルゴリズムである．このアルゴリズムでは，テキスト文字列を遡って読むことがないため，処理時間が短縮できる．

テキスト文字列 ABCABCABA から照合文字列 ABCABA を探すときは, まず左端を合わせて左から 1 文字ずつ比較する．

```
ABCABCABA
ABCABA
```

先頭文字を 0 番目とすると 5 番目で合わなくなる．ここで単純な方法では，照合文字列を右に 1 字ずらして, また照合文字列の最初から調べ直す．しかし，照合文字列の 3 番目と 4 番目の AB が照合文字列の先頭 AB と一致することを知っていれば，照合文字列の先頭の 2 文字は調べ直す必要がない．そこで，この先頭 2 文字を飛ばして 3 文字目の C とテキスト文字列の現在位置とを揃え，そこから調べ直せばよい．

```
ABCABCABA
   ABCABA
```

このことを能率よく行うためには, “ 照合文字列の位置 5 で一致しなくなったなら，照合文字列の位置 2 から調べ始めればよい ” ということを “ pat[5]=2 ” という形で表にまとめておけばよい．この表 pat[j] は，照合文字列パターンの先頭の j 文字について，その頭と尾が何文字一致するかを調べたものである．この照合文字列 ABCABA の照合開始の位置は表 1 のようになる．

表 1　照合文字列 ABCABA の照合開始の位置

A	B	C	A	B	A
0	0	0	1	2	1

```
int kmp_match(int i, int j, int n)
{
    int pt = 1, pp = 0;
    int i_len, j_len;
    int *skip;

    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);
```

```
    skip = (int *) calloc(j_len + 1, sizeof(int));
    /* 表の作成 */
    skip[0] = 0;
    skip[pt] = 0;
    while (*(array[j] + pt) != 256){
        if(*(array[j] + pt) == *(array[j] + pp)){
            skip[++pt] = ++pp;
        } else if (pp == 0){
            skip[++pt] = 0;
        } else {
            pp = skip[pp];
        }
    }

    for(pt = pp = 0; pt < i_len && pp < n; ){
        if(*(array[i] + pt) == *(array[j] + pp)){
            pt++; pp++;
        } else if(pp == 0){
            pt++;
        } else {
            pp = skip[pp];   /* ここがポイント */
        }
    }
    free(skip);
    if(pp == n){
        return (pt - pp);    /* 重複文字列の先頭の場所 */
    }
    return (-1);
}
```

4.2.7 ルール 4 の適用

次の関数 reduct_4() はルール 4 の検索時に記録した重複文字列の先頭位置 m より，i 番目の変数が指す文字列の重複部分を j 番目の変数に置き換える．戻り値はない．

```
void reduct_4(int i, int j, int m)
{
    int k, l;
    int i_len, j_len;
    int *t0;
```

```
    i_len = rulelen(array[i]);
    j_len = rulelen(array[j]);

    t0 = (int *) calloc(i_len - j_len + 2, sizeof(int));

    for(l = k = 0; l < i_len + 1; l++, k++){
        if (l == m){
            t0[k] = 256 + j;
            l += j_len - 1;
        } else {
            t0[k] = *(array[i] + l);
        }
    }
    free(array[i]);
    array[i] = t0;

    return;
}
```

4.2.8 **ルール** 5

ルール 5 が適用できるかどうかを調べるためには各変数が意味する終端文字列を見る必要がある．次の関数 printrule_rec() は引数 s が表す変数が意味する終端文字列を文字コード表示に直してファイル fp に出力する．文字コード表示にしたのは，後のソートのためである．戻り値はない．

```
void printrule_rec(FILE *fp, int *s)
{
    int n;

    for (; *s != 256; s++){
        if (*s < 256){
            fprintf(fp, " %02x", *s);
        } else {
            printrule_rec(fp, array[*s - 256]);
        }
    }
    return;
}
```

次の関数 check_no_rule5() は現在の文法に含まれる 1 番目以降の全ての変数に対し，それが意味する終端文字列の一覧を指定された名前 output のファイルに出力する．戻り値はない．終端文字列の出力には

printrule_rec() を用いる.

```
void check_no_rule5(char *output)
{
    int i;
    FILE *fp;

    fp = fopen(output, "w");
    if(fp == NULL){
        printf("file open err\n");
        exit(-1);
    }

    /* ルールを代入する */
    for(i = 1; a[i] != NULL; i++){
        fprintf(fp, "array[%5d] :", i);
        printrule_rec(fp, array[i]);
        fprintf(fp, "\n");
    }
    fclose(fp);

    return;
}
```

上記の check_no_rule5() で作成されたファイルを各行の辞書的順番でソートしてファイルに保存しておく．次の関数 check_sort_rule5() はソートされた終端文字列を上から順に比較していき，同じ終端文字列がないかをチェックし，結果を表示する．引数 output は check_no_rule5() で出力したファイルの名前である．戻り値はない.

```
void check_sort_rule5(char *output)
{
    int c, x[30000], y[30000];
    int i_len, j_len;
    int i, j, i_max;
    int same = 0, l;
    FILE *fp;

    fp = fopen(output, "r");
    if(fp == NULL){
        printf("file open err\n");
        exit(-1);
```

```
}

for(l = 0; l < i_max - 1; l++){ /* i_max : 生成規則の数から 2 を引いた値 */
    if(l == 0 || l %2 == 1){
        for(i_len = 0; (c = fgetc(fp)) != '\n'; i_len++){
            x[i_len] = c;
            printf("%c", x[i_len]);
        }
    }
    printf("\n");
    if(l == 0 || l %2 == 0){
        for(j_len = 0; (c = fgetc(fp)) != '\n'; j_len++){
            y[j_len] = c;
            printf("%c", y[j_len]);
        }
    }
    printf("\ni_len : %d j_len : %d\n", i_len - 10, j_len - 10);

    if(i_len == j_len){
        for(i = j = 10; i < i_len && j < j_len; i++, j++){
            if(x[i] == y[j]){
                same++;
                if(same == i_len){
                    printf("同じルールが見つかりました。\n");
                    exit(-1);
                }
            } else {
                printf("同じルールではない。\n");
                same = 0;
                break;
            }
        }
    } else {
        printf("スキップ\n");
    }
    /* 配列の初期化 */
    if(l == 0 || l %2 == 0){
        for(i_len = 0; i_len < 30000; i_len++){
            x[i_len] = 0;
        }
```

```
        } else if(l %2 == 1){
            for(j_len = 0; j_len < 30000; j_len++){
                y[j_len] = 0;
            }
        }
    }
    printf("同じルールは見つかりませんでした。\n");
    fclose(fp);
    return;
}
```

4.3 アルゴリズムの実装

3 章で提案したアルゴリズムを実装する．

アルゴリズムは，圧縮対象のファイルを 1 つ入力し，文法とその大きさを表すファイル，生成規則の文字列の中で各変数が使われている回数を表すファイル，そして文法に含まれる 1 番目以降の全ての変数が意味する終端文字列の一覧を表すファイルを出力する．

コマンド行の引数すなわちパラメータを，実行開始時にプログラムに渡す方法をとる [5]．最初の引数 argc は, このプログラムを呼び出したコマンド行の引数の個数である．2 番目の引数*argv は，引数を内容とする文字列の配列を指すポインタで，その文字列 1 つが引数 1 つに対応する．

プログラムの内部では，具体的には次のような動作をしている．まず，0 番目の変数 array[0] がファイル全体の文字列と対応する 1 つの生成規則をつくる．はじめは生成規則が 1 つしかないため手順 1 でルール 2 のみを検索する．この段階で，重複文字列が見つからない場合は圧縮不可能となる．ルール 2 が適用できたならば，生成規則が 2 つになったので提案法を適用する．ルール 2 の検索とルール 3 の検索で見つかった重複文字列の長さをそれぞれ rr2，rr3 とし，両者の長さを比較する．重複文字列が長い方のルールを適用し，その直後にルール 4 が適用できる場所がある限り，関数 reduct_4 を重複文字列が見つかる度，実行する．この処理を全ての生成規則の組み合わせに対して行う．最後に，gram_len で文法の大きさを計り，関数 check_no_rule1() と関数 check_no_rule5() よりルール 1,5 が不要となることを確認する．

```
int main(int argc, char *argv[]){
    int c, i, j, k, l, m = 0;
    int rr, rr2, rr3;
    int gram_len = 0, num_char;

    FILE *fp;
    fp = fopen(argv[1],"rb");
    for (num_char = 0; (c = fgetc(fp)) != EOF; num_char++){
        ;          /* 入力文字数のカウント */
    }
```

```
/* 文字数+1個のint型のメモリを動的に確保 */
line = (int *) calloc(num_char + 1, sizeof(int));
rewind(fp); /* ファイルの先頭に戻す関数 */

/* 確保したメモリ上に書き込む */
while((c = fgetc(fp)) != EOF){
    line[m] = c;
    m++;
}
line[num_char] = 256;
fclose(fp);

array_size = 5000;
array = (int **) calloc(array_size, sizeof(int *));
array[0] = line;

if(array[1] == NULL){
    if(same_pat2(a[0], &x2, &y2, &maxlen) > 1){
        reduct_2(0, empty_array(), x2, y2, maxlen);
    } else {
        printf("圧縮できません。\n");
        exit(-1);
    }
    while ((m = kmp_match(0, 1, maxlen)) > -1){
        reduct_4(0, 1, m);
    }
}

for(i = 0; i < empty_array() - 1; i++){
    for(j = i + 1; j < empty_array(); j++){
        rr2 = same_pat2(array[i], &x2, &y2, &maxlen);
        rr3 = same_pat3(i, j, &x3, &y3);

        if(rr2 >= rr3 && rr2 > 1){
            reduct_2(i, empty_array(), x2, y2, rr2);
            for(l = 0; l < empty_array() - 1; l++){
                while ((m = kmp_match(l, empty_array() - 1, rr2)) > -1){
                    reduct_4(l, empty_array() - 1, m);
                }
            }
```

```
            } else if(rr2 < rr3 && rr3 > 1){
                reduct_3(i, j, empty_array(), x3, y3, rr3);
                for(l = 0; l < empty_array() - 1; l++){
                    while ((m = kmp_match(l, empty_array() - 1, rr3)) > -1){
                        reduct_4(l, empty_array() - 1, m);
                    }
                }
            } else { /* rr2 = rr3 = -1 の場合 */
                ;
            }

            if(empty_array() == array_size){
                increse_array();
            }
        }
    }

    for(i = 0; array[i] != NULL; i++){
        printf("array[ %d] :", i); printrule(a[i]);
    }

    for(i = 0; array[i] != NULL; i++){
        m = rulelen(array[i]);
        gram_len += m;
    }
    printf("文法の大きさは %d である。\n", gram_len);

    check_no_rule1(argv[2]);
    check_no_rule5(argv[3]);

    return 0;
}
```

5 検証

本章では，前章での提案法について検証する．

提案法の評価には，カルガリーコーパス [6] のファイル (18 ファイル) を使用する．各ファイルを提案法によって文法圧縮し，得られる文法に対してリダクションルール 1 と 5 を適用する必要がないことを確認する．検証方法は，4.3 節でのプログラムを各ファイルに対して適用し，文法とその大きさを表すファイル 1 を作成し，ここから生成規則の文字列の中で各変数が使われている回数を表すファイル 2，またファイル 1 を終端文字列に直したファイル 3 を作成する．ファイル 3 を，コマンドプロンプトでソートした結果をファイル 4 に保存し，これを引数として関数 `check_sort_rule5()` を実行する．この結果よりルール 1,5 が不要であるかが確認できる．さらに，ファイル 2 の各変数の回数をコマンドプロンプトでソートして回数の多い変数を確認する．これより，どの文字列が多く適用されているかが確認できる．

検証した結果，表 2 に示すカルガリーコーパスのファイルに対し，各変数は最低でも 2 回適用されていることが確認できた．これより，リダクションルール 1 は不要であるといえる．表 2 の中で空欄となっているファイルについては処理時間が長すぎるため，実験を断念した．また，各変数が指す文字列を終端文字列に変換し，ソートしたところ，同じ終端文字列を意味する変数は見つからなかった．これより，リダクションルール 5 は不要であるといえる．よって，この提案法は irreducible であることが確認された．したがって，本提案法はユニバーサルである．

参考までに，実験に用いたデータと生成規則の数，文法の大きさ，処理にかかった時間を表 2 にまとめた．これより，生成規則の数が多くなると処理にかかる時間が長くなることが分かった．また各変数が使われている回数をカウントした結果，どのファイルにおいても “ in ”“ er ”“ re ”“ . 改行 ” などが比較的多く使われていることが分かった．

表 2　実験に用いたデータと生成規則数，文法の大きさ，処理時間

ファイル名	サイズ [Byte]	生成規則数	文法の大きさ	処理時間
bib	111261	5199	28180	5 時間 15 分
book1	768771			
book2	610856			
geo	102400	6499	52868	17 時間 27 分
news	377109			220 時間 36 分
obj1	21504	1241	9569	3 分
obj2	246814	10469	65761	42 時間 46 分
paper1	53161	3137	16707	55 分
paper2	82199	4741	24883	4 時間 10 分
paper3	46526	3053	16260	52 分
paper4	13286	1002	5579	1 分
paper5	11954	941	5196	1 分
paper6	38105	2342	12633	20 分
pic	513216	7486	44844	59 時間 57 分
progc	39611	2262	12438	20 分
progl	71646	2789	14978	47 分
progp	49379	1907	10598	13 分
trans	93695	3088	16985	53 分

6 まとめ

Yang and Kieffer によって提案された 5 つのリダクションルールのうち，3 つのルールを適用するだけで文法圧縮するアルゴリズムを提案した．そして，カルガリーコーパスのファイルを実験に用いて，提案法の正当性を検証した．この提案法では，実験したファイルに対してルール 1,5 を適用できる場所は存在しなかった．目的を満たすアルゴリズムであることを確認した．

本研究の目的とは異なるが，この実験より生成規則の数が多くなると処理にかかる時間が長くなることが分かった．提案法の欠点は，ファイル全体を一度に読みこんで文字列操作するため，処理時間が極端に長くなることである．

今後の課題は，処理時間の短縮化，文法を圧縮ファイルに変換して圧縮率等から性能評価すること等が挙げられる．本研究で提案するアルゴリズムを先頭部分からファイルを読んで文字列操作するような逐次アルゴリズムに変換できれば，処理速度の改善につながると考えられる．

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] En-hui Yang, John C. Kieffer, “ Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform -Part One: Without Context Models, ” IEEE Transactions on Information Theory, vol.46, no.3, pp.755–777, 2000.

[2] 喜田拓也，坂本比呂志，下薗真一，“ 省スペースな線形時間文法圧縮アルゴリズム, ” 電子情報通信学会，pp.1–7，2004.

[3] 湯淺太一，コンパイラ，昭晃堂, 2005.

[4] John C. Kieffer, En-hui Yang, “ Grammar-Based Codes: A New Class of Universal Lossless Source Codes, ” IEEE Transactions on Information Theory, vol.46, no.3, pp.737–754, 2000.

[5] 奥村晴彦，C 言語による最新アルゴリズム事典，技術評論社，2005.

[6] B.W. カーハニン，D.M. リッチー，プログラミング言語第 2 版，共立出版株式会社，2005.

[7] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus