信州大学
大学院理工学系研究科

修士論文

順序保存符号の符号語を
分節木上へ配置した符号器の遅延特性

指導教員　西新 幹彦 准教授

専攻　電気電子工学専攻
学籍番号　13TM211A
氏名　荻野 真志

2015 年 3 月 13 日

目次

1　はじめに

近年，緊急地震速報の技術が発展してきている．これは地震観測所で地震の前兆を観測した時に，その地震観測所より通信が行われて携帯電話などの端末を通して私達の元にその地震の情報が届けられる技術である．この技術に欠かせない問題として通信の遅延があろう．深刻な災害を未然に防ぐために 1 秒でも早く通信によって情報を届ける必要がある．本研究では情報源符号化のしくみに着目し，通信の遅延の改善を試みている．緊急地震速報の例を挙げたように，遅延を小さくすることは重要な工学的問題といえる．このような目的の符号化はあまり研究されておらず，新たに問題に取り組む意義がある．よく研究されているのはデータサイズを小さくするための符号化だが，これはむしろ遅延を大きくすることによって効率を上げるので本研究の目的に反する．従来研究 [1] では分節木の葉に符号語を割り当てさらにその符号語の語頭を節点に割り当てた構造をもつ符号器が提案されており，この符号器によって遅延の算術平均が改善されることが報告されている．本研究では，このような方法で符号器を作るにあたって，どのような符号語を用いれば遅延特性がよくなるかを実験的に検証する．

本論文は次のような構成をとる．2 章では本研究で扱った通信システムの説明と遅延の定義を行う．3 章では符号器と符号器を構成する分節木の説明を行う．4 章では遅延特性改善のために従来研究 [1] で提案された再配置について説明する．5 章では再配置の効果を上げる符号として順序保存符号を指摘し，最適な順序保存符号 (Hu-Tucker 符号) を導入する．6 章では Hu-Tucker 符号を再配置した符号器の遅延特性を調べるために実験を行い，その結果と考察を述べる．

2　通信システムのモデルと遅延

本研究では，符号器と復号器の間の通信路で誤りは発生しないと仮定するが，1 秒あたりに送信できるシンボル数は限られていると仮定する．具体的には，符号語は 1 秒あたり 1 シンボルで送信する送信機によって復号器に届けられる．ただし送信機がシンボルを送信中に符号器から出力された符号シンボルを貯えておくために送信機の直前にキューをおく．キューのサイズは十分大きいとする．

符号器の前には情報源がある．簡単のために情報源アルファベットは $\mathcal{S} = \{a, b\}$ とする．情報源は定常無記憶で，シンボル a, b の生起確率はそれぞれ $p, 1-p$ とする．情報源の出力のタイミングは レートλ のポアソン過程に従うと仮定する．

以上のことをまとめると本研究で考える通信システムは図 1 のように書ける．情報源シンボルが情報源から出力されてから，そのシンボルが復号器から出るまでの時間をそのシンボルの遅延と定義する．

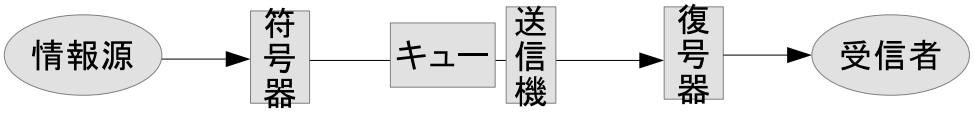

図 1　通信システムのモデル

3　符号器と分節木

符号器では分節で切り出された文字列が語頭フリー条件を満たすように情報源系列を分節し，切れ出された文字列に対応した符号語を出力すると仮定する．符号語全体は一意復号可能とする．符号器が分節で切り出す可能性のある文字列全体は語頭フリー条件を満たすので枝に情報源シンボルをラベル付けした木で表現できる．この木を符号器の分節木と呼ぶ．分節木の葉は，切り出される文字列を表しているので，符号器によって決まる対応した符号語を持つ．

葉に符号語が対応した分節木を用いて，符号器の動きは次のように説明できる．分節木の節点を符号器の状態とみなし，符号器は初め分節木の根にいるものとする (初期状態)．符号器にシンボルが入力されると，分節木ではシンボルとラベルとが一致した枝を進む．ただし 1 シンボルにつき，1 つ次の節点に移る．分節木で葉に到達した時，葉に配置された符号語を出力し，根に戻る．符号器にシンボルが次々に入力されることから，以上の動作は繰り返される．図 2 の分節木が与えられたとする．入力系列 $\mathbf{s} = abbaaa...$ を分節木用いて符号化することを考える．初め初期状態にある．系列 1 番目のシンボル a が入力されると，分節木で a のラベル付けされた枝を進む．次に 2 番目のシンボル b が入力されると，分節木で b のラベル付けされた枝

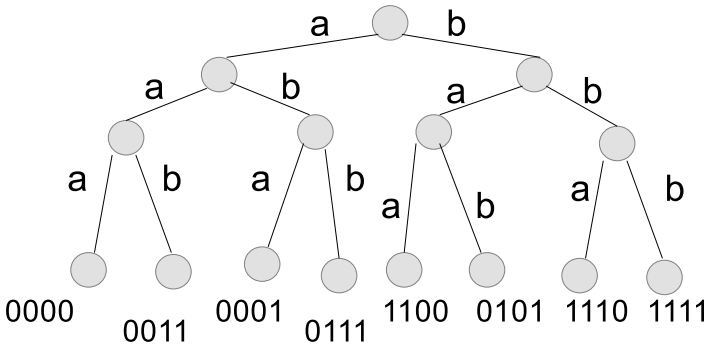

図 2　分節木の例

を進む．次に 3 番目のシンボル b が入力されると，分節木で b のラベル付けされた枝を進む．この時，分節木で葉であることから，その葉に配置された符号語 0111 を出力し，根に戻る．次に aaa が入力されると，分節木の根から a のラベル付けされた枝を 3 つ進み，葉に配置された符号語 0000 を出力し，根に戻る．以上から，出力系列は 01110000 となる．

復号器の構造も符号器と同様である．つまり，復号器もその入力系列を分節し，切り出された符号語に対応して元の情報源シンボルの並びを復元する．従って，復号器の動きも符号器と同じく分節木を用いて表すことができる．

4　符号語の再配置

符号器の仕組みを工夫することによって，遅延を改善することが本研究の目的である．このような問題に対して，従来研究 [1] で分節木の葉に配置された語頭符号の語頭を節点に再配置する方法によって，遅延が改善されることが報告されている．

この方法は次のようになる．共通の親を持つ子を兄弟と呼ぶ．分節木で兄弟の関係にある葉に配置された符号語が共通の語頭をもっている場合に，その語頭を抜き出し，親の節点に再配置する．再配置は葉に限らず節点の兄弟へも同様に適用される．例えば，図 3(図 2 と同じ) のような語頭符号を葉に配置した分節木があるとする．左から 1 番目と 2 番目の兄弟と 3 番目と 4 番目の兄弟と 7 番目と 8 番目の兄弟において共通した語頭が存在している．これらの語頭を親の節点に再配置すると，図 4 のような分節木になる．次にレベルが 2 の節点に注目すると，左から 1 番目と 2 番目の兄弟において共通した語頭が存在している．したがって，それらの語頭を親の節点に再配置する．すると，図 5 のような分節木となる． 図 5 の分節木ではどの兄弟にも共通した語頭は存在しない．

再配置を適用した符号器の動きは次のように説明できる．符号器は初め分節木で根にいる

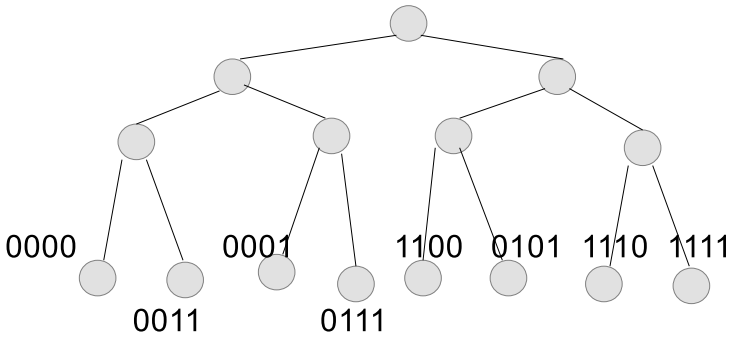

図 3　語頭の再配置その 1

(初期状態). 符号器にシンボルが入力されると，分節木ではシンボルとラベルとが一致した枝を進む. ただし 1 シンボルにつき，1 つ次の節点に移る. 節点に再配置された語頭が存在すれば，その語頭を出力する. 分節木で葉に到達した時，葉に配置された残りの符号語を出力し，根に戻る. 符号器にシンボルが次々に入力されることから，以上の動作は繰り返される. 例えば，入力系列

$$\mathbf{s} = abbaaa...$$

を図 5 の分節木で符号化することを考える. その様子を図 6 に表した. 1 番目の a で左の枝を進む. すると，節点に語頭 0 が再配置されているから，この 0 を出力する. 次に，2 番目の b で右の枝を進む. 次に，3 番目の b で右の枝を進む. すると，葉であるから，残りの符号語 111 を出力し，根に戻る. 次に，4 番目の a で左に進み，0 を出力する. 次に，5 番目の a で左に進み，0 を出力する. 6 番目の a で左に進み，葉であるから残りの符号語 00 を出力し，根に戻る. 以上から，出力系列は 01110000 となる. この出力系列は 3 章の例の出力系列と一致して

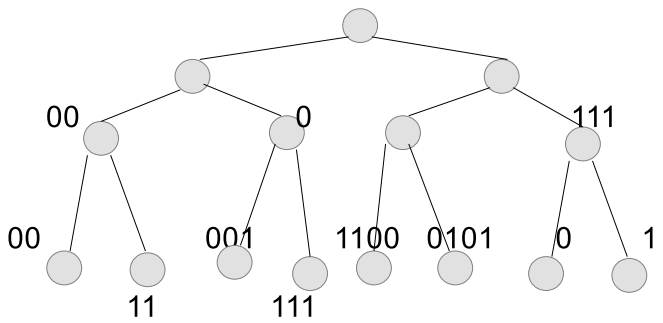

図 4　語頭の再配置その 2

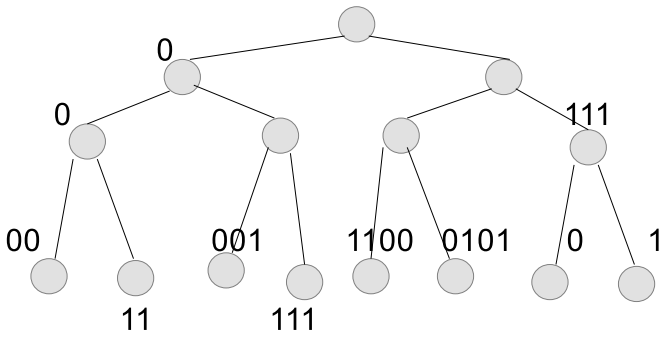

図 5　語頭の再配置その 3

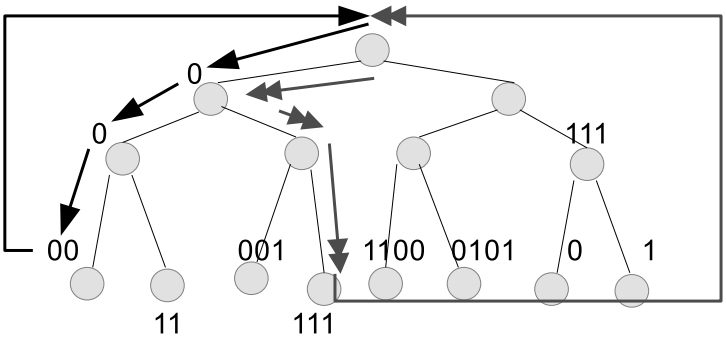

図 6　符号化

いる．このように，再配置する前とした後で出力系列は同一となる．

復号器に対しても，語頭の再配置は符号器と同様に適用される．

5　順序保存符号の導入

4 章で本研究の符号器の構造を定義した．本章では符号についての議論を行う．我々は遅延改善のために符号器の分節木の節点に符号語の語頭をより多く再配置することを考えた．語頭をより多く再配置するために使用したのが順序保存符号である．順序保存符号は符号語の並びが辞書順となる符号である．順序保存符号を使用すれば，分節木の左から右へ辞書順を保って符号語を配置することができる．これは葉の隣同士で符号語の語頭が揃っている状態である．よって，符号語の再配置を行うと，語頭が分節木の節点により多く再配置されることが期待される．

最適な順序保存符号は 1971 年に T. C. Hu と A. C. Tucker が発表した Hu-Tucker 符号である [3]．以下では Hu-Tucker 符号の構成方法の説明を行う．情報源シンボルの生起確率が与えられていることが前提となる．与えられた情報源シンボルの生起確率をもとに，木の葉の生起確率を求める．以降は木をつくる要領で符号を構成できる．まず，T-C[*1] について説明する．T-C とはある 2 つの葉または節点に注目した際，その 2 つの葉または節点の間の文字列が空または節点のみからなることを言う．図 7 と図 8 で四角は葉を，丸は節点を表す． 図 7 で，a と b，b と c は T-C となっているが，a と c は T-C でない．図 8 で，どの 2 つの組み合わせも T-C となっている．この T-C を使って，木を構築する．T-C の中から，生起確率の和が最小となる T-C を選ぶ．選んだ T-C を結合する．この結合の作業を節点が 1 つになるまで続け

[*1] T-C とは tentative-connecting の略語である．

図 7　T-C の例 1

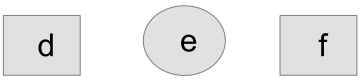

図 8　T-C の例 2

る．すると木ができあがる．この木は T' 木と呼ばれる．注意事項として，2 つの葉または節点を足し合わせて新たな節点を作り出す時に，新たな節点は左の子の節点の上に作る．

T' 木は T'_N 木と呼ばれる別の木へ変換される．T'_N 木は Hu-Tucker 符号を表す．T' 木ができれば，T' 木の葉から根までに辿る枝の数が分かる．T'_N 木へ変換するために，T' 木では他の枝と交差しないために枝を付け替える．以下で実際に生起確率から T'_N 木をつくってみる．生起確率 $\{\frac{6}{21}, \frac{5}{21}, \frac{4}{21}, \frac{3}{21}, \frac{2}{21}, \frac{1}{21}\}$ が与えられたとする．生起確率と葉を対応させる．そして，生起確率の和が最小となる T-C を選び出し，結合させる．結合を続けていくと，図 9 のような T' 木ができる．次に，枝の付け替えを行い，T'_N 木をつくる．根から葉までの枝の数はそのままに，枝同士が交差しないように枝を付け替える．すると，図 10 のような T'_N 木ができる．T'_N 木は Hu-Tucker 符号を表す．

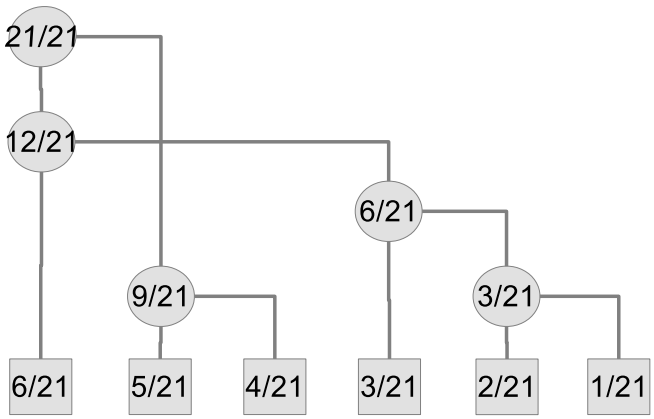

図 9　T'木 の例

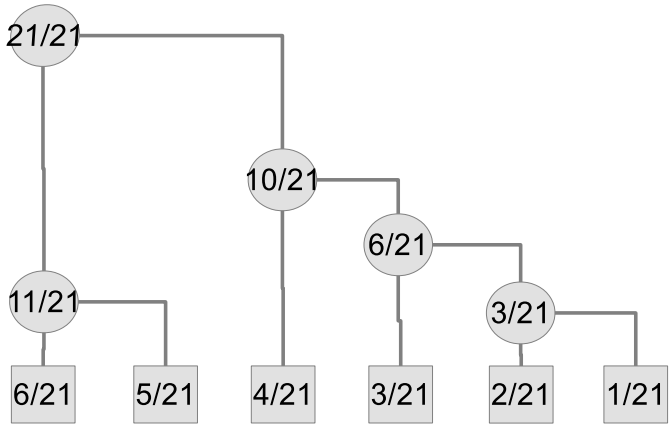

図 10　T'_N木 の例

6　符号器の遅延特性

Hu-Tucker 符号を配置・再配置した分節木を持つ符号器の遅延特性を調べるために，シミュレーション実験を行った．6.1 節で実験方法を説明し，6.2 節と 6.3 節でそれぞれ実験結果と考察を記す．

6.1　実験方法

図 11 で示した通信システムに基づいて遅延を測定するシミュレーション実験を行った．実際のプログラムを付録 A に記載した．情報源は定常無記憶な分布 $p, 1-p$ に従って 10 万個のシンボルを出力する．情報源シンボルの出力の時間間隔はレート λ のポアソン過程に従う．そのレート λ は

$$\lambda < \frac{1}{H} \tag{1}$$

となる．ただし H は情報源のエントロピーを示す．これはキューで待ち時間が発散しないための理論限界値である．符号器では，簡単のために，すべての葉のレベルは分節木の深さ d に等しいとした．その深さ d は $1 \leq d \leq 15$ とした．遅延特性の比較を行うために，無圧縮と Huffman 符号でも Hu-Tucker 符号と同様に実験を行った．その他の変数は表 1 にまとめた．

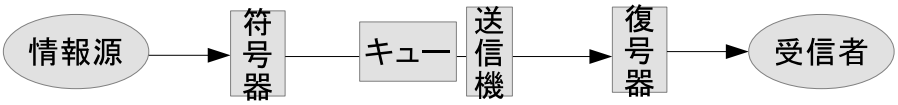

図 11　実験で使用した通信システムのモデル

表 1　実験諸元

名称	固定または可変	値
情報源アルファベット	固定	$\mathcal{S} = \{a, b\}$
情報源シンボルの生起確率	可変	$0.5 \leq p < 1$
ポアソン過程の到着レート	可変	$0 < \lambda < \frac{1}{H}$
分節木の深さ	可変	$1 \leq d \leq 15$
送信機の処理能力	固定	1 [symbol/sec]

6.2　実験結果

上記のシミュレーション実験により得られた結果から，無圧縮，Hu-Tucker 符号，Huffman 符号それぞれの平均遅延の各レートにおける最小値を算出して図にまとめた．横軸はポアソン過程のレート λ，縦軸は平均遅延とした．図 12 は生起確率 $p = 0.9$，図 13 は $p = 0.8$，図 14 は $p = 0.7$，図 15 は $p = 0.6$ の場合をそれぞれ表している．

図 12 について，無圧縮，Hu-Tucker 符号，Huffman 符号の平均遅延はいずれも単調増加となっており，レートを大きくするに従って平均遅延が急激に増加するレートが存在する．このレートは計算によって求めることができる．平均符号長を L とすれば，キューに入力される符号シンボルの到着レートは $\lambda L[symbol/sec]$ と計算できる．入力 λL が送信能力 $1[symbol/sec]$ に近づくにつれて平均遅延は大きくなり，$1[symbol/sec]$ を超えると発散する．つまり，レート λ が $\frac{1}{L}$ を超えると平均遅延は発散する．

最小な平均遅延はレートが小さい方より順に無圧縮，Hu-Tucker 符号，Huffman 符号の 3 つに場合分けできる．どのように場合分けできるかも図中に示した．レートが小さい場合では，無圧縮で通信することが平均遅延を最も小さくできる．次にレートを大きくしていくと，キューでの待ち時間を抑えるために符号器で情報圧縮を行う．情報圧縮のために本来は平均符号語長の最適な Huffman 符号を用いるが，本研究は遅延を改善することが目的であるために Hu-Tucker 符号を用いた．レートが小さいほうから順に Hu-Tucker 符号，Huffman 符号の並びになっている．レートが大きい場合では平均符号語長の小さい Huffman 符号の平均遅延が

小さくなっているが，レートが小さい場合では Hu-Tucker 符号の平均遅延の方が小さくなっている．これの原因は次のように考えられる．レートが小さい場合において，キューにおける平均遅延は Hu-Tucker 符号と Huffman 符号とで共に小さくなる．一方，符号器・復号器での平均遅延は Hu-Tucker 符号の方が小さくなる．なぜならば Hu-Tucker 符号の方が Huffman 符号よりも分節木の節点により多くの符号語を再配置できるからである．以上を総合すると Hu-Tucker 符号の平均遅延が小さくなったと考えられる．また，図中の Huffman 符号の範囲よりもさらにレートを大きくした時，平均遅延の発散を防ぐ語頭符号はない．

生起確率を $p = 0.8, 0.7, 0.6$ と変化させた場合についても，生起確率 $p = 0.9$ の場合と同様な結果が得られた．つまり，最小な平均遅延はレートによって小さい方から順に無圧縮，Hu-Tucker 符号，Huffman 符号の 3 つに場合分けできる．生起確率 $p = 0.9, ..., 0.6$ としていくと，そのエントロピー H は次第に増加する．したがって，$p = 0.9, ..., 0.6$ とすれば，式 (1) よりレート λ は上から押さえられることとなり，図 12，13，14，15 のように $-x$ 軸方向へ縮んだ形となる．

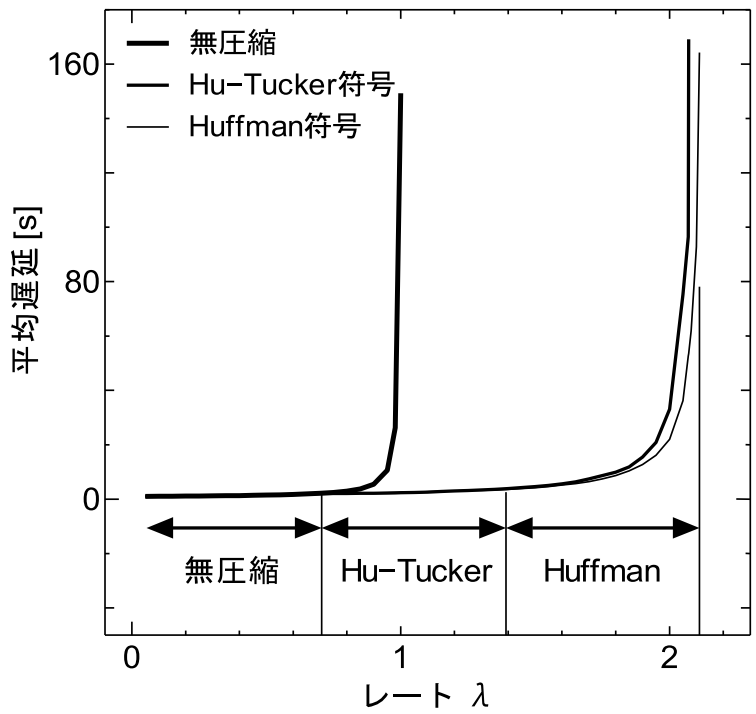

図 12　p=0.9 の場合のレート-平均遅延特性

6.3 考察

図 12, 13, 14, 15 でレートの小さい方から無圧縮, Hu-Tucker 符号, Huffman 符号の並びになった理由を考える. 通信の遅延は符号器・復号器またはキュー・送信機で生まれる. キュー・送信機で生じる遅延の影響は大きく, この遅延を小さくするために式 (1) を満たすレートで通信をするべきである. レートが式 (1) を満たすのであれば, キュー・送信機で生じる遅延を小さくすることが可能である. 次に, 通信の遅延に影響を与えるのは符号器・復号器で生じる遅延である. 符号器はレートが大きくなれば, キューでの待ち時間の発散を防ぐために情報圧縮を行う. この情報圧縮の度合いが大きくなるに従って, 符号器で生じる遅延も大きくなる.

レートが小さい場合, キュー・送信機で待ち時間が発散しないため, 符号器・復号器の遅延を 0 にできる無圧縮の平均遅延が最小になったと考える. 次にレートを大きくすると, 無圧縮ではキュー・送信機で待ち時間が発散を始めるために, 符号器は情報圧縮を行う. すると, Hu-Tucker 符号の平均遅延が最小になった. このことから Hu-Tucker 符号の方が符号器・復

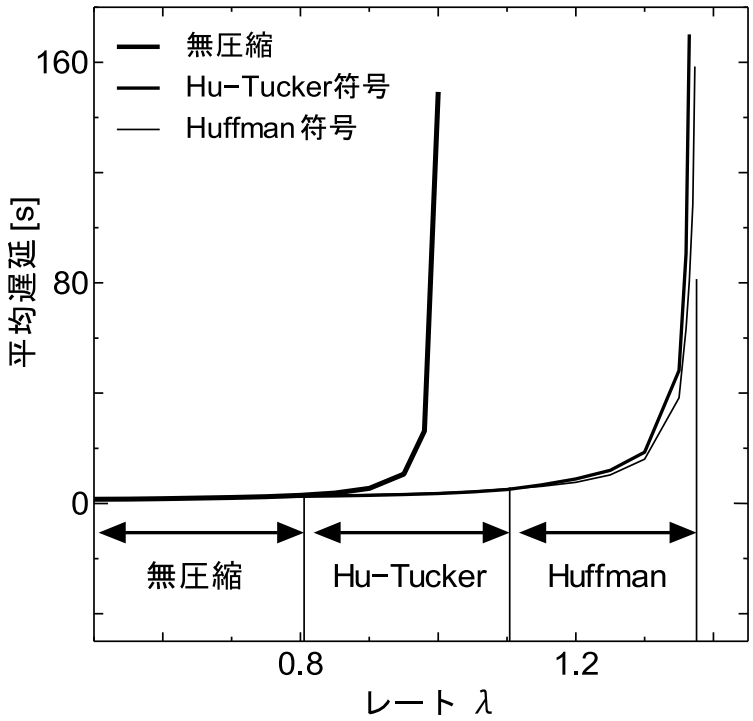

図 13　p=0.8 の場合のレート-平均遅延特性

号器で生じる遅延が小さくなると予想できる．さらにレートを大きくすると Huffman 符号の平均遅延が最小になった．この原因は Hu-Tucker 符号は Huffman 符号よりも平均符号語長が大きいために Hu-Tucker 符号の方が先に発散するためである．

7　まとめ

符号器の仕組みを工夫することによって通信の遅延を改善する問題に取り組んだ．本研究では従来研究 [1] で提案された分節木の節点に符号語を再配置する符号器に適する符号として順序保存符号 (Hu-Tucker 符号) を取り入れた．Hu-Tucker 符号を再配置した符号器の遅延特性を調べるために無圧縮と Huffman 符号との比較と実験を行った．実験の結果，平均遅延を最小にする場合は，レートによって 3 つに場合分けできることが分かった．調査した Hu-Tucker 符号が平均遅延を最小にする場合はとても限られていたが，それが存在することは示した．今後の課題は更なる遅延の改善のために符号器の分節木の形を決定することである．

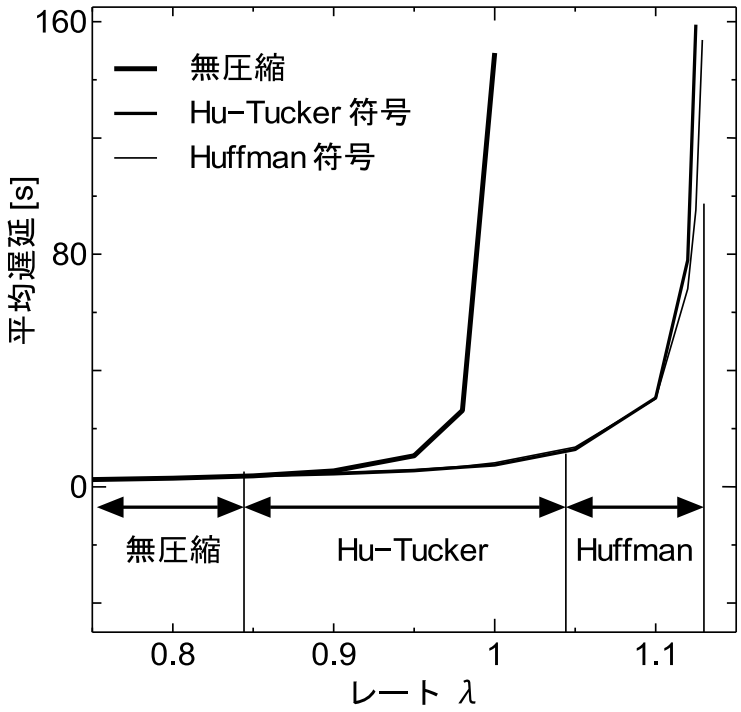

図 14　p=0.7 の場合のレート-平均遅延特性

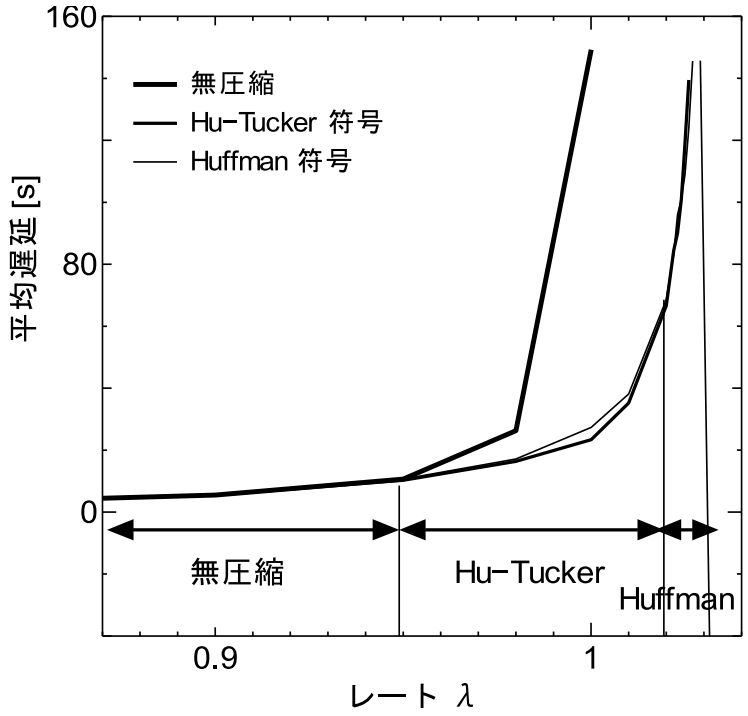

図 15　p=0.6 の場合のレート-平均遅延特性

謝辞

本研究を終えるにあたり, 終始一貫して丁寧なご指導を賜わりました指導教員の西新幹彦先生に心より感謝の意を申し上げます.

参考文献

[1] 荻野真志, 西新幹彦, 「遅延特性改善のための分節木上の符号語の配置に関する一考察」, 電気情報通信学会信越支部大会, 2013.

[2] 韓太舜, 小林欣吾, 情報と符号化の数理, 培風館, 1999.

[3] T. C. Hu, and A. C. Tucker, “Optimal computer search trees and variable-length alphabetical codes,” SIAM J. Appl. Math., Vol.21, No.4, pp.514–523, December 1971.

付録 A　プログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

//一様乱数の生成
#define MRND 1000000000L
static int jrand;
static long ia[56];

static void irn55(void)
{
    int i;
    long j;

    for(i=1;i<=24;i++){
        j=ia[i]-ia[i+31];
        if(j<0) j+=MRND;
        ia[i]=j;
    }
    for(i=25;i<=55;i++){
        j=ia[i]-ia[i-24];
        if(j<0) j+=MRND;
        ia[i]=j;
    }
}

void init_rnd(unsigned long seed)
{
    int i,ii;
    long k;

    ia[55]=seed;
    k=1;
    for(i=1;i<=54;i++){
        ii=(21*i)%55;
        ia[ii]=k;
        k=seed-k;
        if(k<0) k+=MRND;
        seed=ia[ii];
    }
    irn55(); irn55(); irn55();
    jrand=55;
}

long irnd(void)
```

```
{
    if(++jrand>55){irn55(); jrand=1;}
    return ia[jrand];
}

double rnd(void)
{
    return (1.0/MRND)*irnd();
}
//一様乱数→指数乱数を生成
double ramda_x = 0.5;       //ポアソンレートの宣言
double rand_exp(void)
{
return (-(1/ramda_x)*log(1-(rnd())));
}

#define CODEWORDLEN 40

struct node {
    int ch[2];
    char word[CODEWORDLEN + 1];
};

int num_en_node;
int num_de_node;
struct node encode_tree[256];
struct node decode_tree[256];

#define BUFFLEN 4096
int
load_en_tree(FILE *fp, int n)
{
    int isleaf = -1;
    int m;
    char buff[BUFFLEN];

    fgets(buff, BUFFLEN, fp);
    //printf("buff=%s", buff);
    sscanf(buff, "%d : %s\n" , &isleaf, encode_tree[n].word);
    switch (isleaf){
    case 0:
        m = n;
        encode_tree[m].ch[0] = n + 1;
        n = load_en_tree(fp, n + 1);
        encode_tree[m].ch[1] = n;
        n = load_en_tree(fp, n);
        break;
    case 1:
        encode_tree[n].ch[0] = -1;
        encode_tree[n].ch[1] = -1;
```

```
        n++;
        break;
    default:
        printf("[%d]", __LINE__);
        exit(1);
        break;
    }
    return(n);
}

void
sprout_de_tree(char *codeword, char *src)
{
    int i;
    int node;

    for (i = 0, node = 0; codeword[i] != '\0'; i++){
        if (codeword[i] != '0' && codeword[i] != '1'){
            printf("[%d] \n", __LINE__);
            exit(1);
        }
        if (decode_tree[node].ch[codeword[i] - '0'] < 0){
            node = decode_tree[node].ch[codeword[i] - '0'] = num_de_node++;
            decode_tree[node].ch[0] = -1;
            decode_tree[node].ch[1] = -1;
            decode_tree[node].word[0] = '\0';
        } else {
            node = decode_tree[node].ch[codeword[i] - '0'];
            if (decode_tree[node].word[0] != '\0'){
                printf("[%c]",decode_tree[node].word[0] );
                printf("[%d]", __LINE__);
                exit(1);
            }
        }
    }
    strcpy(decode_tree[node].word, src);
    return;
}

void
make_de_tree_(int node)
{
    static char path[CODEWORDLEN + 1];
    static int idx = 0;

    if (encode_tree[node].word[0] == '\0'){
        idx++;
        path[idx - 1] = 'a';
        make_de_tree_(encode_tree[node].ch[0]);
        path[idx - 1] = 'b';
```

```
        make_de_tree_(encode_tree[node].ch[1]);
        idx--;
    } else {
        path[idx] = '\0';
        // printf("sprout(\"%s\", \"%s\");\n", encode_tree[node].word, path);
        sprout_de_tree(encode_tree[node].word, path);
    }
    return;
}

void
make_de_tree(void)
{
    num_de_node = 1;
    decode_tree[0].ch[0] = -1;
    decode_tree[0].ch[1] = -1;
    decode_tree[0].word[0] = '\0';
    make_de_tree_(0);
    return;
}

void
extract_prefix(struct node tree[], int i)
{
    int j, k;

    if (tree[i].ch[0] < 0 || tree[i].ch[1] < 0){
        return;
    }
    extract_prefix(tree, tree[i].ch[0]);
    extract_prefix(tree, tree[i].ch[1]);

    /* ここで子供から共通の語頭を受け取る */
    if (tree[tree[i].ch[0]].word[0] == '\0' || tree[tree[i].ch[1]].word[0] == '\0'){
        tree[i].word[0] = '\0';
        return;
    }

    for (j = 0; tree[tree[i].ch[0]].word[j] == tree[tree[i].ch[1]].word[j]; j++){
        if (tree[tree[i].ch[0]].word[j] == '\0'){
            break;
        }
        tree[i].word[j] = tree[tree[i].ch[0]].word[j];
    }
    tree[i].word[j] = '\0';
    if (j > 0){
        for (k = 0; tree[tree[i].ch[0]].word[k + j] != '\0'; k++){
            tree[tree[i].ch[0]].word[k] = tree[tree[i].ch[0]].word[k + j];
        }
        tree[tree[i].ch[0]].word[k] = '\0';
```

```
        for (k = 0; tree[tree[i].ch[1]].word[k + j] != '\0'; k++){
            tree[tree[i].ch[1]].word[k] = tree[tree[i].ch[1]].word[k + j];
        }
        tree[tree[i].ch[1]].word[k] = '\0';
    }

    return;
}

void
extract_prefix_en(void)
{
    int i;

    for (i = 0; i <num_en_node; i++){
        if (encode_tree[encode_tree[i].ch[0]].word[0] != '\0' &&
        encode_tree[encode_tree[i].ch[1]].word[0] != '\0'){
            int j, k;

for (j = 0; encode_tree[encode_tree[i].ch[0]].word[j] == encode_tree[encode_tree[i].ch[1]].word[j]; j++){
                encode_tree[i].word[j] = encode_tree[encode_tree[i].ch[0]].word[j];
            }
            encode_tree[i].word[j] = '\0';
            if (j > 0){
                for (k = 0; encode_tree[encode_tree[i].ch[0]].word[k + j] != '\0'; k++){
                        encode_tree[encode_tree[i].ch[0]].word[k] =
encode_tree[encode_tree[i].ch[0]].word[k + j];
                }
                encode_tree[encode_tree[i].ch[0]].word[k] = '\0';
                for (k = 0; encode_tree[encode_tree[i].ch[1]].word[k + j] != '\0'; k++){
                        encode_tree[encode_tree[i].ch[1]].word[k] =
encode_tree[encode_tree[i].ch[1]].word[k + j];
                }
                encode_tree[encode_tree[i].ch[1]].word[k] = '\0';
            }
        }
    }
    return;
}

void
extract_prefix_de(void)
{
    int i;

    for (i = 0; i <num_de_node; i++){
        if (decode_tree[decode_tree[i].ch[0]].word[0] != '\0' &&
decode_tree[decode_tree[i].ch[1]].word[0] != '\0'){
            int j = 0, k;
            printf("[%d]", __LINE__);
```

```
            while (decode_tree[decode_tree[i].ch[0]].word[j] ==
decode_tree[decode_tree[i].ch[1]].word[j]){
                decode_tree[i].word[j] = decode_tree[decode_tree[i].ch[0]].word[j];
                j++;
                                printf("[%d]", __LINE__);
            }
            decode_tree[i].word[j] = '\0';
            if (j > 0){
                for (k = 0; decode_tree[decode_tree[i].ch[0]].word[k + j] != '\0'; k++){
                        decode_tree[decode_tree[i].ch[0]].word[k] =
 decode_tree[decode_tree[i].ch[0]].word[k + j];
                }
                decode_tree[decode_tree[i].ch[0]].word[k] = '\0';
                for (k = 0; decode_tree[decode_tree[i].ch[1]].word[k + j] != '\0'; k++){
                        decode_tree[decode_tree[i].ch[1]].word[k] =
decode_tree[decode_tree[i].ch[1]].word[k + j];
                }
                decode_tree[decode_tree[i].ch[1]].word[k] = '\0';
            }
        }
        printf("num_de_node=%d 子 1=%d 子 2=%d 復号語=%s\n",
 i, decode_tree[i].ch[0], decode_tree[i].ch[1], decode_tree[i].word);
    }
}

int symbol_queue_count = 0;
int symbol_queue_in = 0;
int symbol_queue_out = 0;
#define SYMBOL_QUEUE_MAX 1000000
double symbol_queue_time[SYMBOL_QUEUE_MAX];
char symbol_queue_symbol[SYMBOL_QUEUE_MAX];

void
put_symbol_queue(double time, char symbol)
{
    if (symbol_queue_count >= SYMBOL_QUEUE_MAX){
        printf("[%d]\n", __LINE__);
        exit(1);
    }
    symbol_queue_time[symbol_queue_in] = time;
    symbol_queue_symbol[symbol_queue_in] = symbol;
    symbol_queue_in = (symbol_queue_in + 1) % SYMBOL_QUEUE_MAX;
    symbol_queue_count++;

    return;
}

void
get_symbol_queue(double *time, char *symbol)
{
```

```
    if (symbol_queue_count == 0){
        printf("[%d]\n", __LINE__);
        exit(1);
    }
    *time = symbol_queue_time[symbol_queue_out];
    *symbol = symbol_queue_symbol[symbol_queue_out];
    symbol_queue_out = (symbol_queue_out + 1) % SYMBOL_QUEUE_MAX;
    symbol_queue_count --;
    return;
}

int server_queue_count = 0;
int server_queue_in = 0;
int server_queue_out = 0;
#define SERVER_QUEUE_MAX 1000000
char server_queue[SERVER_QUEUE_MAX];

void
put_server_queue(char symbol)
{
    if (server_queue_count >= SERVER_QUEUE_MAX){
        printf("[%d]\n", __LINE__);
        exit(1);
    }
    server_queue[server_queue_in] = symbol;
    server_queue_in = (server_queue_in + 1) % SERVER_QUEUE_MAX;
    server_queue_count++;

    return;
}

int
get_server_queue()
{
    char symbol;

    if (server_queue_count == 0){
        printf("[%d]\n", __LINE__);
        exit(1);
    }
    symbol = server_queue[server_queue_out];
    server_queue_out = (server_queue_out + 1) % SERVER_QUEUE_MAX;
    server_queue_count--;

    return (symbol);
}

double current_time;
double arrival_time;
double departure_time;
```

```
#define CUSTOMER_MAX 10000
int customers = 0;
int arrival_mode = 0;        /* 0:通常の到着 1:余分な到着 2:到着停止 */


void
encode()
{
    static int current_node = 0;
    int next_node;
    int length;

    if (rnd() >= 0.2){              //ここでソースシンボルの生起確率を変更できる
        next_node = encode_tree[current_node].ch[0];
        put_symbol_queue(current_time, 'a');
    //      printf("arrive: a\n");
    } else {
        next_node = encode_tree[current_node].ch[1];
        put_symbol_queue(current_time, 'b');
        //printf("arrive: b\n");
    }

    length = strlen(encode_tree[next_node].word);
    if (length > 0){
        int i;
        if (server_queue_count == 0){
            departure_time = current_time + 1.0;
        }
        for (i = 0; i < length; i++){
            put_server_queue(encode_tree[next_node].word[i]);
        //      printf("queue: %c\n", encode_tree[next_node].word[i]);
        }
    }

    if (encode_tree[next_node].ch[0] == -1) {
        current_node = 0;
        if (arrival_mode == 1){
            arrival_mode = 2;
        }
    } else {
        current_node = next_node;
    }

    customers++;
    if (arrival_mode == 0 && customers >= CUSTOMER_MAX){
        arrival_mode = 1;
    }
    if (arrival_mode <= 1){
        arrival_time = current_time + rand_exp();
    }
```

```
    return;
}


void
decode(FILE *fo)
{
    static int current_node = 0;
    double a_time;
    char symbol;
    int next_node;
    int i;
    double n;

    if (get_server_queue() == '0'){
        next_node = decode_tree[current_node].ch[0];
        //printf("server: 0 (%d)\n", next_node);
    } else {
        next_node = decode_tree[current_node].ch[1];
        //printf("server: 1 (%d)\n", next_node);
    }

    for (i = 0; decode_tree[next_node].word[i] != '\0'; i++){
        get_symbol_queue(&a_time, &symbol);
        //printf("'%c'->'%c' %f %f %f\n", symbol, decode_tree[next_node].word[i],
 a_time, current_time, current_time - a_time);
        if(fo){
        fprintf(fo, "%f \n",current_time - a_time);
        } else {
            printf("%d\n", __LINE__);
            exit(1);
        }

        n = current_time - a_time;
        if (decode_tree[next_node].word[i] != symbol){
            printf("%d\n", __LINE__);
            exit(1);
        }
    }


    if (decode_tree[next_node].ch[0] == -1 && decode_tree[next_node].ch[1] == -1){
        current_node = 0;
    } else {
        current_node = next_node;
    }

    if (server_queue_count > 0){
        departure_time = current_time + 1.0;
```

```
    }

    return;
}

void
print_tree(struct node tree[], int i)
{
    printf("i=%d ch_0=%d ch_1=%d word=[%s]\n", i, tree[i].ch[0], tree[i].ch[1], tree[i].word);
    if (tree[i].ch[0] > 0){
        print_tree(tree, tree[i].ch[0]);
    }
    if (tree[i].ch[1] > 0){
        print_tree(tree, tree[i].ch[1]);
    }
    return;
}

int
main(void)
{
    FILE *fp, *fo;
    char filename[80];
    int i;

    init_rnd(2345);

    printf("ファイル名=");
    gets(filename);

    if ((fp = fopen(filename, "r")) == NULL) {
        printf ("ファイルをオープンできません.\n");
        exit(1);
    }

        if ((fo = fopen("test20.ods", "w")) == NULL) {
        printf ("ファイルをオープンできません.\n");
        exit(1);
    }

    fscanf(fp, "%d ", &num_en_node);
    load_en_tree(fp, 0);
    fclose(fp);

    make_de_tree();
    extract_prefix(encode_tree, 0);
    extract_prefix(decode_tree, 0);


    for (i = 0; i < num_en_node; i++){
```

```
        printf("i=%d ch_0=%d ch_1=%d en_word=[%s]\n",
i, encode_tree[i].ch[0], encode_tree[i].ch[1], encode_tree[i].word);
    }
    for (i = 0; i < num_de_node; i++){
        printf("i=%d ch_0=%d ch_1=%d de_word=[%s]\n",
 i, decode_tree[i].ch[0], decode_tree[i].ch[1], decode_tree[i].word);
    }


    while (arrival_mode <= 1 || server_queue_count > 0){
        if (arrival_mode <= 1 && ((server_queue_count == 0) || (arrival_time <= departure_time))){
            current_time = arrival_time;
            encode();
        } else {
            current_time = departure_time;
            decode(fo);
        }
    }
    printf("%d, %d\n",symbol_queue_count, server_queue_count);
    fclose(fo);

    return(0);
}
```