信州大学工学部

学士論文

最適なリバーシブル可変長符号の幅優先探索アルゴリズムの計算量解析

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　08T2076D
氏名　深瀬 拓巳

2012 年 3 月 12 日

目次

1 序論 1

1.1 研究の背景と目的 1

1.2 本論文の構成 1

2 リバーシブル可変長符号 (RVLC) 2

2.1 接頭辞フリー符号 2

2.2 接尾辞フリー符号 2

2.3 RVLC の例 2

3 最適 RVLC 探索アルゴリズム 3

3.1 探索状態の生成とコスト 3

3.2 コストのしきい値の設定 5

3.3 コスト計算の改善 6

4 最適 RVLC 探索アルゴリズムの計算量解析 8

4.1 解析の探索範囲 8

4.2 h-estimate の計算量 9

4.3 しきい値とメモリの削減 11

4.4 EN の値と選択済みの符号語の数 12

5 まとめ 14

謝辞 15

参考文献 15

付録 A ソースコード 16

A.1 4.2 節におけるプログラム 1 16

1　序論

1.1　研究の背景と目的

今日の社会において可変長符号 (variable-length codes, VLCs）は，平均符号語長を短くすることができるという点から多くの圧縮に使われている．しかし，VLC は 1 ビットのエラーでさえも伝達に深刻なエラーを引き起こす要因となってしまい，正しい復号化が行われず，受信者に損失をもたらすということが起きてしまうことがある．リバーシブル可変長符号（reversible variable-length codes, RVLCs）はこの事態を防ぐために用いられる．VLC はエラーが発生した位置から後ろのデータが捨てられてしまうのに対して RVLC は始めからでも終わりからでも瞬時復号が可能であるという特徴があり，終わりからも復号が可能であるため本来捨てられてしまうデータもほぼ復元できる．

RVLC の構成法について多くの従来研究があるが，Yuh-Ming Huang, Ting-Yi Wu, and Yunghsiang S. Han は平均符号語長が最小となる RVLC の探索アルゴリズムを提案した [1]．本研究では，Huang らのアルゴリズムの計算量を解析することによって，高速化への知見を得る．

1.2　本論文の構成

本論文では次のような構成をとる．2 章ではリバーシブル可変長符号 (RVLC) の定義を説明し，3 章では Huang らの探索アルゴリズムを説明する．4 章では Huang らのアルゴリズムの計算量を解析と速度向上のための改善点の指摘をおこない，5 章においてまとめを示す．

2　リバーシブル可変長符号 (RVLC)

接頭辞フリー符号かつ接尾辞フリー符号であるような符号語の集合をリバーシブル可変長符号 (RVLC) という．下記に接頭辞フリー符号及び接尾辞フリー符号について説明する．

2.1　接頭辞フリー符号

接頭辞フリー符号を説明するにあたってまず接頭辞について説明する．任意のアルファベット $\mathcal{A}$ 上の長さ n の記号列 $\boldsymbol{x} = x_1x_2\cdots x_n$ があるとき ($x_i \in \mathcal{A}$, $1 \leq i \leq n$)，$\boldsymbol{x}$ 自身を含むその先頭部分 (具体的には，x_1，x_1x_2，$x_1x_2x_3$，$\cdots$，$x_1\cdots x_{n-1}$，$x_1\cdots x_{n-1}x_n$) を $\boldsymbol{x}$ の接頭辞という．また，長さ 0 の文字列も $\boldsymbol{x}$ の接頭辞である．

接頭辞フリー符号とはある符号語の集まりがあり，その各々の符号語が他の符号語の先頭と同じになっていない，即ち接頭辞となっていない符号語のことを言う．例えば {11，110，1011} は {11} が {110} の接頭辞となってしまっているため接頭辞フリー符号とは言えないのに対し，{11，101，1011} は各々の符号語が他の符号語の接頭辞となっていないため接頭辞フリー符号と言える．

2.2　接尾辞フリー符号

接尾辞フリー符号を説明するにあたってまず接尾辞について説明する．任意のアルファベット $\mathcal{A}$ 上の長さ n の記号列 $\boldsymbol{x} = x_1x_2\cdots x_n$ があるとき ($x_i \in \mathcal{A}$, $1 \leq i \leq n$)，$\boldsymbol{x}$ 自身を含むその末尾部分 (具体的には，x_n，$x_{n-1}x_n$，$x_{n-2}x_{n-1}x_n$，$\cdots$，$x_2\cdots x_n$，$x_1x_2\cdots x_n$) を $\boldsymbol{x}$ の接尾辞という．また，長さ 0 の文字列も $\boldsymbol{x}$ の接尾辞である．

接尾辞フリー符号とは，ある符号語の集まりがあり，その各々の符号語が他の符号語の末尾と同じになっていない，即ち接尾辞となっていない符号語のことを言う．例えば {11，110，1011} は {11} が {1011} の接尾辞となってしまっているため接尾辞フリー符号とは言えないのに対し，{11，110，1101} は各々の符号語が他の符号語の接尾辞となっていないため接尾辞フリー符号と言える．

2.3　RVLC の例

RVLC の例としては {11，101，1001} や {100，110，1011，11101} などがあり見ての通りこれらは接頭辞フリー符号且つ接尾辞フリー符号となっている．

3　最適 RVLC 探索アルゴリズム

次式は最小の平均符号語長を持つ符号語の計算式である．

$$\widehat{C} = arg \min_{C \in \Psi} \sum_{i=1}^{N} p_i \times l(c_i) \tag{1}$$

式 (1) では $\{s_1, s_2 \cdots s_n\}$ のように N 個の独立したソースシンボル S の個々の確率を $\{p_1, p_2 \cdots p_n\}$ としており，Ψ は全ての RVLC を指す．$C = \{c_1, c_2 \cdots c_N\} \in \Psi$ に対して C の中の各々の符号語長を $l(c_i)$，$1 \leq i \leq N$ と表す．最適な RVLC とは式 (1) を満たすような RVLC のことを指す．

Huang らの探索アルゴリズムでは，符号語の候補の中から符号語を取捨選択していくことで式 (1) を満たすような RVLC の $\hat{C}$ を見つけることを最終目標としている．アルゴリズムは符号語が何も選ばれていない状態から開始する．符号語の候補は長さと辞書順によって順位付けられており，候補の先頭を符号語として採用するか否かの 2 通りの選択肢がある．つまり次の状態として 2 通りが考えられるが，探索アルゴリズムでは 2 つの状態をどちらも生成して保持する．したがってアルゴリズムは複数の探索状態を同時に保持する．そして，性能の良い（平均符号語長の短い）符号が見つかりそうな状態に対して 2 つの「次の状態」を作成する．これは幅優先探索に相当する．また，以上のことから，生成される状態の関係は 2 進木で表現することができる．

この章の以降の節では，Huang らの探索アルゴリズムを詳しく説明する．

3.1　探索状態の生成とコスト

探索アルゴリズムによって生成される状態の関係を図 1 に示す．状態は 2 進木のノードで表される．Huang らのアルゴリズムでは，各ノードにコストと呼ばれる値を割り振り，「良い符号が見つかる可能性」の指針として使用する．この木における各ノードには RVLC として選択済み符号語，RVLC として選択する候補の情報が入っている．図 1 において C_W, C_X, C_Y はそれぞれノード W, X, Y において RVLC として選んだ符号語の集まりであり，A_W, A_X, A_Y はノード W, X, Y の符号語長ごと辞書順に並んだ符号語の候補の集まりである．

一般に，ノード W から 2 つの子供のノード X, Y を作るには次のような手順に従う．ある符号語の候補 A 及び RVLC として選択済みの符号語 C が入ったノード W から 2 つの子供のノード X，Y を作る．左側 (X 側) に進んだ場合は A_W の一番左の候補 (a_1) を選択したとして C_W の一番右側に加え C_X とし，A_W から選択した符号語 (a_1) の接頭辞または接尾辞となっている符号語を消去し A_X として更新する．また，右側 (Y 側) に進んだ場合は A_W の一番左側の候補 (a_1) は選択されなかったとして A_W の一番左の候補 (a_1) を消去して A_Y とし，

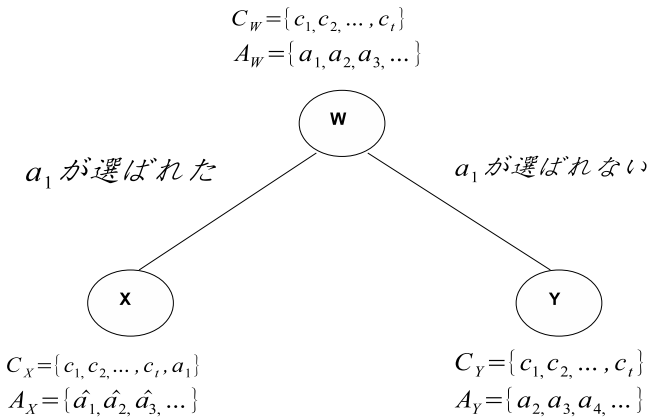

図 1　W からの分岐

C_W はそのままの状態で C_Y に更新する．

このとき RVLC として選択した符号語の数が必要な符号語の数に達しておらず，且つ必要な残りの符号語の数よりも候補の数が少ない場合はそのノードから子供のノードを作っていっても必要な符号語の数に達することは無いのでそのノードは消去する．また原則として最初に選ぶ符号語は 0 から始まるものとし，これに反するノードもまた消去する．これは例えば {0, 11, 101, 1001} とその 0 及び 1 を反転させた {1, 00, 010, 0110} の平均符号語長は同じであり，探索していった結果どちらかの符号語の集まりのみを得ることができればよい．そのため各符号語の集まりを反転させた符号語を探索する必要は無く，本論文では最初の符号語は 0 から始まるものに限定する．

次に全ての葉を各々に割り振られたコストの小さい順に並べ，再び最も小さいコストを持つノードを取り出し上の手順を繰り返し行うのだが，ここでコストの計算法について説明する．

図 1 を例にコストの計算法を説明する．まずノード W において既に選択されている符号語のコストを

$$g(W) = \sum_{i=1}^{t} p_i \times l(c_i) \tag{2}$$

とする．また，候補 A_W から左から順番に残りの符号語を選択していったとして考えた候補のコストを

$$h(W) = \sum_{i=t+1}^{N} p_i \times l(a_{i-t}) \tag{3}$$

とする．ノード W のコストを

$$m(W) = g(W) + h(W) = \sum_{i=1}^{t} p_i \times l(c_i) + \sum_{i=t+1}^{N} p_i \times l(a_{i-t}) \tag{4}$$

と定める．このようにノード W におけるコストは符号語のコスト $g(W)$ と候補のコスト $h(W)$ を足したものである．つまり，ノードのコストは仮の平均符号語長であると言える．また，未確定の符号語が確定するときには，候補と同じかより長い符号語が選ばれるので，コストは実際の平均符号語長を超えることはない．

また X のコストは

$$m(X) = g(X) + h(X) = [\sum_{i=1}^{t} p_i \times l(c_i)] + p_{t+1} \times l(a_i) + \sum_{i=t+2}^{N} p_i \times l(\hat{a}_{i-t-1}) \tag{5}$$

Y のコストは

$$m(Y) = g(Y) + h(Y) = \sum_{i=1}^{t} p_i \times l(c_i) + \sum_{i=t+1}^{N} p_i \times l(a_{i-t+1}) \tag{6}$$

となる．

このように定義されたコストを用いて上に示した試行を繰り返し，必要な数の符号語が選択されたノードが一つ出来たところで最適な RVLC が出来たと言える．その理由としては，このアルゴリズムでは先にも述べたように各ノードの持つコストはそのノードから分岐して実際に出来る RVLC の平均符号語長を超えることはない．また，このアルゴリズムではノードをコストが小さい順に並べ，最もコストが小さいノードを優先して分岐していく．更に残り 1 個の符号語が決まると必要数の符号語が揃うノードのコストとそのノードを分岐して完成する RVLC の実際の平均符号語長は等しい．そのためあるノードをリストの先頭から取り出し，それを分岐することで一番最初に完成した RVLC の平均符号語長は他のどのノードのコストを超えることはなく，他のどの RVLC の平均符号語長以下となる．

図 2 にこれまで説明した探索アルゴリズムの基本構造を示す．

3.2 コストのしきい値の設定

リストに入っているノードの中には明らかにコストが大きすぎてリストの先頭に来るとは考えられないものが存在し，それをリスト内に入れておくことはメモリの無駄遣いであり，探索

step1. まず $C_R = 0$,$A_R = \{a_1 = 0, a_2 = 1, a_3 = 00, a_4 = 01, a_5 = 10, a_6 = 11, \cdots\}$ 且つ $g(R) = 0$,$m(R) = h(R)(= \sum_{i=1}^{N} p_i \times\ l(a_i))$ であるようなノード R をリストの先頭に入れる．

step2. リストの先頭に入っているノード W(始めは $W = R$) を取り出す．もし $C_W \cup \{a_1, a_2, \cdots, a_N - t\}$ が妥当な RVLC ならば，最適な RVLC を得たと言える．

step3. ノード W から 2 つの子供のノード X と Y を作り，C_X, A_X, C_Y, A_Y を作る．もし A_X が空で C_X のサイズが N よりも小さかったらノード X を消去する．残りの子供のノードのコストを計算する．

step4. 残りのノードをリストの中に入れ，コストの小さい順に並び替えて step2 に移る．

図 2　基本探索アルゴリズム

範囲を広め，所要時間を長くしてしまうだけであると考えられる．そこで文献 [1] ではしきい値を設定して，それより大きいコストをもつノードをリストから消去することが提案されている．しきい値としては実際に存在する（最適とは限らない）RVLC の平均符号語長を用いる．（したがって事前に少なくとも 1 つの RVLC を見つけなければならない.）このとき，しきい値より大きいコストをもつノードから最適な RVLC は導かれないので，そのようなノードは消去して構わない．これにより探索範囲を狭め，メモリの使用量を減らす効果とプログラムの速度を向上させる効果が期待できる．

3.3　コスト計算の改善

図 2 のアルゴリズムの step3 において計算されるコストはその後定まる実際の平均符号語長よりかなり小さい値になることがある．これは候補のコスト $h(W)$ が小さくなることが原因で，step 2 で取り出すノードを不適切なものにする．そこで，候補のコストをより正確に算出するために，Huang らは h-estimate と呼ばれる計算方法も提案している [1]．これを説明するにあたって図 1 のようなあるノード W のコストを算出するときを考える．ノード W のコストを算出する際，図 2 のアルゴリズムにおいては候補のコスト $h(W)$ を求めるとき候補 A_W から左から順番に残りの符号語を選択していったときのコストを算出していた．これに対して h-estimate を用いて $h(W)$ を算出する際には，まずノード W をルートノードとした図 2 のアルゴリズムで作成した 2 進木とは別の 2 進木を考える．この別に作成した 2 進木において図 2 のアルゴリズムと同手順でノードを分岐させてはコストを算出し，葉となっているノードをコストの小さい順に並べ替え，コストが最も小さいものをまた分岐させるという作業を行う．ここでこの分岐の回数は任意の回数 EN 回とし，EN 回分岐を行った時点で最小のコストを持つ葉のコストを候補のコスト $h(W)$ とする．

これによってコストを算出するための計算量は増えるが，コストの値が正確になることによって適切なノードが取り出されるようになり，全体として計算速度を上げることができる．また，無駄な探索が削減されることから，メモリの使用量も減らすことができる．

h-estimate はコストの計算を改善するためのものであるが，計算の途中で実際に存在する RVLC の平均符号語長を見つける場合がある．このとき，3.2 節で説明したしきい値を更新することができる．具体的には h-estimate によってより深くまで探索することで残り 1 個の符号語を選択すれば RVLC が完成するノードをみつけたとすると，そのノードのコストと実際に完成する RVLC の平均符号語長は等しい．従ってそのノードのコストよりもコストが大きいノードは最適な RVLC を作り得ないと言える．つまり残り 1 個の符号語を選択すれば RVLC が完成するノードのコストをしきい値として設定できる．

図 3 に h-estimate のアルゴリズムを示す．アルゴリズム内の Ubound はしきい値を表し，このアルゴリズムにおける step3 内でしきい値の更新が行われている．

//Z は図 2 のアルゴリズムにおけるノード R，ノード X，またはノード Y とする//

step1. ノード Z とその C_Z, A_Z 及びアルゴリズム 1 で計算した $g(Z)$ をコピーし，リストに入れる．$i = 0$ とする．

step2. リストの先頭のノード W(始めは $W = Z$) をリストから取り出す．もし C_W のサイズが $N - 1$ と等しいならば，$C_Z = C_W \cup \{a_1\}$ であり，step5 に移る．
//$m(W)$ は全てのサブの木の中のノード Z からの RVLCs の最小のコストとなる．もし $i = EN$ ならば step5 へ移る．

step3. ノード W の 2 つの子供のノード X,Y 及び C_X, A_X, C_Y, A_Y を作る．もし A_X が空で C_X のサイズが N よりも小さいならばノード X を消去する．もし始めに C_X の中に選んだ符号語が 0 から始まるものでなかったならばノード X を消去する．もし C_X のサイズが $N - 1$ と等しく，且つ $m(X) < Ubound$ ならば $Ubound = m(X)$ とする．
//$C_X \cup \{a_1\}$ は妥当な RVLC であり，即ち $m(X)$ は新しい Ubound となる．もし X(または Y) のコストが Ubound よりも大きいならばノード X(または Y) を消去する．

step4. リストに子供のノードを入れ，コストの低い順に並べ替え，$i = i + 1$ とし，step2 へ移る．

step5. コストを返す．

図 3　h-estimate のアルゴリズム

4　最適 RVLC 探索アルゴリズムの計算量解析

文献 [1] では図 2 のアルゴリズムに加えて始めに深さ優先探索で 1 つの RVLC を見つけ出し，それをしきい値として設定している．しかし始めに深さ優先探索で一つの RVLC を作成したとしても深さ優先探索で見つけた RVLC は非常に平均符号語長が長いため，これをしきい値として設定したとしても消去できるノードの数は極僅かであり，始めに一つの RVLC を求めるという余計な計算に見合った働きをしない．ゆえに本研究では図 2 のように始めに一つの RVLC を作成しないアルゴリズムとし，h-estimate 内でしきい値の更新が起きるまではしきい値を無限とするようなアルゴリズムとした．

また文献 [1] では h-estimate の EN の値を 1 に固定して実験を行い，所要時間及びメモリの使用量を標準の方法でコスト計算を行うアルゴリズムの半分程度に出来たと報告されている．しかし本研究では EN の値を場合に応じて変動させればより所要時間を短縮できるのではないかと考え，そのために必要な知見を得るためにアルゴリズムの様々な計算量の解析を行った．

最適 RVLC 探索アルゴリズムの計算量を解析するにあたって以下のような実験を行った．

4.1　解析の探索範囲

符号語数や確率分布など全ての組み合わせで実験を行うことは不可能であるため，本研究では必要な符号語の数 (以降，要素数) 及び符号語の出現確率を固定し，また EN の値を変動させる範囲にも上限を設けた．下記に各値に設けた条件を示す．

まず始めに要素数と確率について述べる．図 4 は図 2 のアルゴリズムを用いたプログラムで各要素数による所要時間を測定したものであり，確率が等確率の時と確率が徐々に小さくなる時 (i 番目の符号語の確率を $(0.9)^{i-1}/\sum_{j=0}^{n-1} 0.9^j$($n$:符号語数) とした) の 2 種類を示した．図 4 を見ると等確率且つ要素数 25 の時が最も所要時間が長くなっていることが分かる．即ち最悪のケースが等確率且つ要素数 25 の時であり，この最悪のケースにおいて所要時間を最も短く出来るようなプログラムを作成すればどのようなケースでも最も所要時間が短くなると考え，要素数 25 で全ての符号語が等確率で出現する場合を解析することにした．

次に EN の値は上限を 25 とした．これは図 2 のアルゴリズムにおけるコストの計算を図 3 の h-estimate のアルゴリズムを用いて行い，また h-estimate 内でしきい値の更新も行うプログラムを用いて EN の値を 0〜900 まで変え，EN-時間特性を調べたところ図 5 のように EN の値が一定の値よりも大きくなると所要時間が長くなっていくことが分かったため，この後の実験では EN の値は所要時間が長くなりだす前の 25 までとして実験を行った．

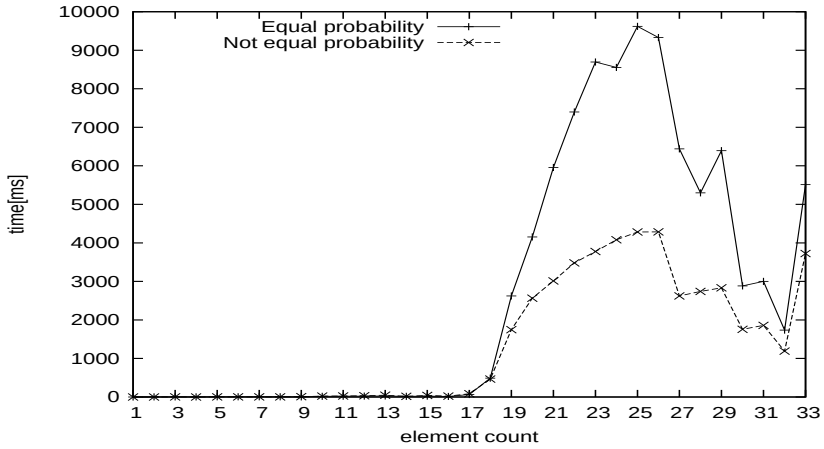

図 4　図 2 アルゴリズムの各確率における要素数-時間特性

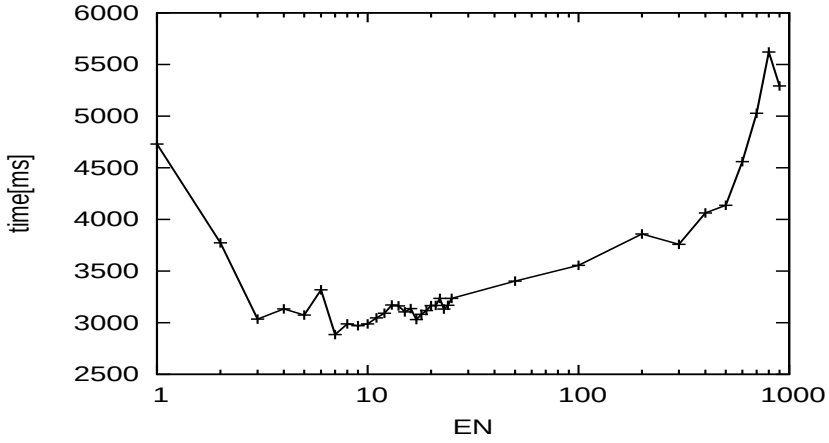

図 5　EN-時間特性

4.2　h-estimate の計算量

本研究ではプログラムを高速化させるにあたって主に h-estimate について解析，改良を行った．

文献 [1] では h-estimate を用いると基本探索アルゴリズムと比較して探索の所要時間が短縮されると述べているが，この要因として 2 つのことが考えられる．1 つ目に h-estimate ではコストを算出する際に EN の値だけ深く探索をすることによってより正確なコストを得るこ

とが出来るということであり，2 つ目はしきい値を更新することで探索範囲を狭められることである．EN の値の大小によってこれら 2 つの要因が所要時間等に与える影響が変化するのは明らかである．そこで，これら 2 つの要因と EN の関係性がそれぞれどれだけ所要時間に影響するかをおおまかに知るために表 1 に示す 3 種類のプログラムを用いて実験した．

表 1　各プログラムの概要

	コストの計算方法	しきい値の更新
プログラム 1	h-estimate	有り
プログラム 2	h-estimate	無し
プログラム 3	標準 (式 (4))	有り

表 1 におけるプログラム 1 は図 5 において使用したプログラムと同じプログラム．プログラム 2 はプログラム 1 からしきい値の更新機能を取り除いたプログラムであり，プログラム 1 と比較することで h-estimate のしきい値を更新する機能がどれだけ所要時間を短縮出来ているのかを把握できる．プログラム 3 はプログラム 1 のコストの計算を標準の方法 (式 (4)) で行うプログラムであり，プログラム 1 と比較することで h-estimate の正確なコストを算出する機能がどれだけ所要時間を短縮出来ているのかを把握できる．

図 6 にプログラム 1 とプログラム 2 の EN-時間特性を示し，図 7 にプログラム 1 とプログラム 3 の EN-時間特性を示した．ここで $EN = 0$ は図 2 の基本探索アルゴリズムを指すので考えない．

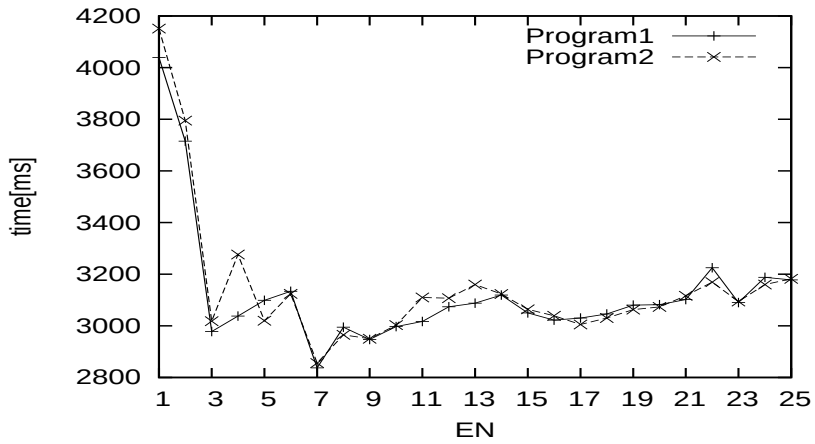

図 6　プログラム 1 とプログラム 2 の EN-時間特性

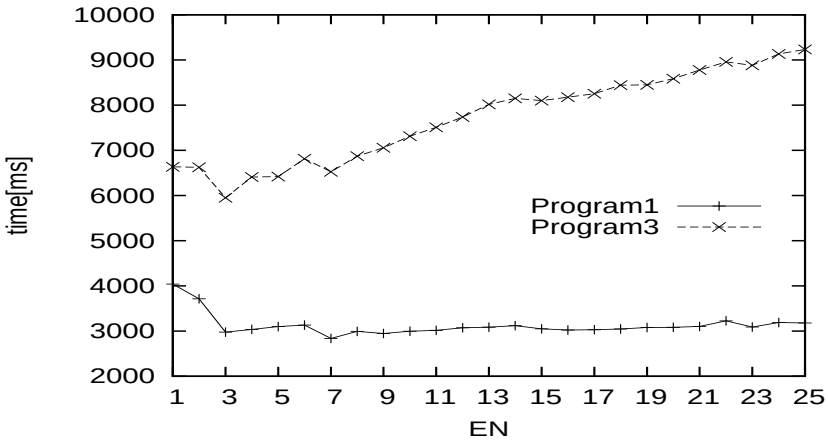

図 7　プログラム 1 とプログラム 3 の EN-時間特性

まず図 6 を見ると EN の値によってプログラム 1 よりもプログラム 2 のほうが所要時間が短い場合とプログラム 1 のほうがプログラム 2 よりも所要時間が短い場合があるがグラフ全体を通して大きな差は見られないことが分かる．つまり所要時間に関してしきい値が与える影響は小さいと言える．次に図 7 を見るとプログラム 1 に比べてプログラム 3 の所要時間が非常に長くなっていることが分かる．つまり正確なコストを見積もる機能は非常に大きく所要時間の短縮につながっていると言える．これらより h-estimate を用いることによる所要時間の短縮は主に正確なコストを見積もる機能によるものであることを確認した．

4.3　しきい値とメモリの削減

4.2 節においてしきい値が所要時間に与える影響は小さいことを確認した．そこでしきい値はメモリの削減に貢献しているという文献 [1] の指摘を確認するために使用されるメモリの最大値をリスト内に入っているノード (状態) の数の最大数とし，各 EN とメモリの最大数との関係を図 8 に示した．

図 8 より EN の値が大きいほどしきい値のメモリを削減する効果が大きくなっていることを確認した．つまり文献 [1] のように EN=1 という小さい値に固定する場合はしきい値は所要時間の短縮は期待できるがメモリの削減に関しては期待できないと言える．

ここまでの実験でしきい値が所要時間及びメモリの削減数に与える影響は大きくないことを確認したがしきい値を用いないプログラムに比べるとどちらも多少の効果は期待できるため以降の実験でもしきい値を用いたプログラムで実験を行う．

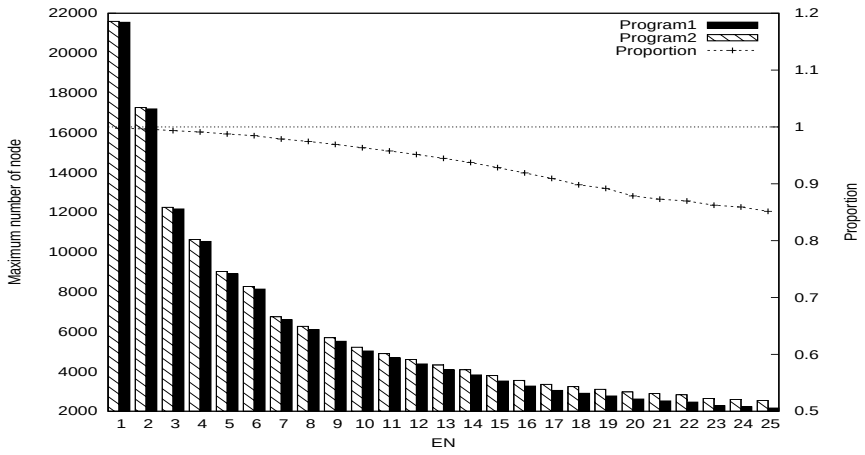

図 8　各 EN におけるリストに入っているノードの最大数

4.4　EN の値と選択済みの符号語の数

4.2 節の実験により探索時間は EN の値と大きく関係していることを確認した．次に所要時間の短縮をするにあたって EN の値は文献 [1] のように一定の値とするほうが良いのかそれとも選択済みの符号語の数に応じて変えたほうが良いのかを調べる．これを調べるにあたって下記のような実験を行った．

まず選択済みの符号語の数が 0〜8 個の時，9〜17 個の時，18〜25 個の時の 3 つの範囲に分ける．次に 3 つの範囲のうち 1 つの範囲の EN を 1，10，20 の 3 つの値にし，他の範囲を EN=0 として 3 つの場合の所要時間を測定する．これを各範囲ごとに行う．図 9 にその実験結果を示した．図は左から選択済みの符号語の数が 0〜8 個の時に EN を各値としその他の範囲では EN=0 とした時の各所要時間，選択済みの符号語の数が 9〜17 個の時に EN を各値としその他の範囲では EN=0 とした時の各所要時間，選択済みの符号語の数が 18〜25 個の時に EN を各値としその他の範囲では EN=0 とした時の各所要時間を示している．比較のために EN が常に 0(h-estimate 無し) のプログラムの所要時間も示した．

図 9 を見ると選択済みの符号語が 0〜9 の時以外は h-estimate を用いないときの所要時間と大きな差が無いため h-estimate の効果が薄いことが分かる．

また選択済みの符号語の数が 0〜8 個の時は EN の値がある程度大きいと所要時間が短くなり，選択済みの符号語の数が 9〜17 個の時は EN の値が小さい程所要時間が短くなり，選択済みの符号語の数が 18〜25 個時には EN の値にはほぼ依存しないということが分かった．これより選択済みの符号語の数が少ない内はある程度 EN の値を大きくとり，選択済みの符号語が

多いときは EN の値を小さくとればより時間を短縮できると推測できる．

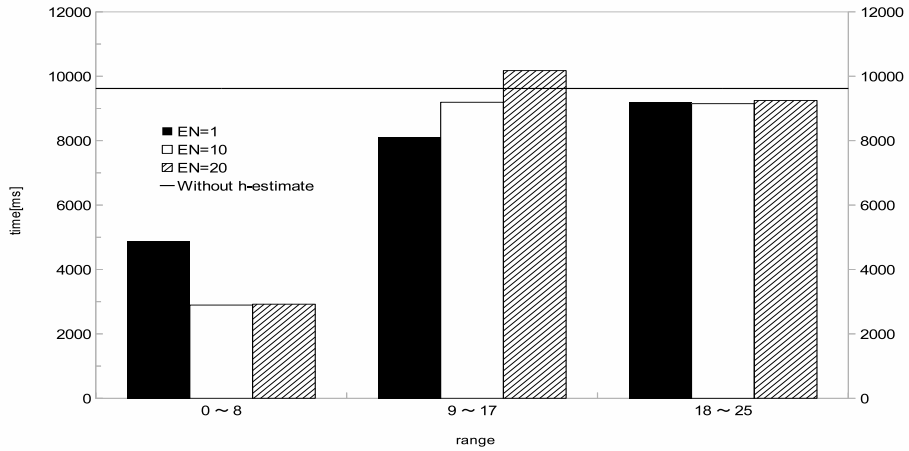

図 9　各場合と各 EN の値における所要時間の変化

上の結果より選択済みの符号語の数が少ない時には EN の値を小さくとると所要時間が短くなることがわかったが，より正確に EN の値を大きくとるべき範囲を調べる必要があり，そのために次の実験を行った．

まず選択済みの符号語の数が 0～3 個の時，0～6 個の時，0～9 個の時，0～12 個の時，0～15 個の時，0～18 個の時，0～21 個の時，0～25 個の時の 8 つの範囲に分ける．次に各範囲ごとにその範囲内のみ EN を適当な大きい値 (本論では EN=10 とした) とし範囲外は EN=0 として所要時間を測定する．その実験結果を図 10 に示した．

図 10 は左から選択済みの符号語の数が 0～3 個の範囲で EN=10 とし，その他の範囲では EN=0 とした時の所要時間，選択済みの符号語の数が 0～6 個の範囲で EN=10 とし，その他の範囲では EN=0 とした時の所要時間，選択済みの符号語の数が 0～9 個の範囲で EN=10 とし，その他の範囲では EN=0 とした時の所要時間，以下同様に 0～12 個の範囲の時，0～15 個の範囲の時，0～18 個の範囲の時，0～21 個の範囲の時，0～25 個の範囲の時である．

図 10 を見てわかるように選択済みの符号語の数が 0～9 個の間 EN=10 とした場合が最も所要時間が短縮されていることが見てとれる．従って 0～9 個の範囲で EN の値を大きい値にすれば速度の向上が期待できることがわかる．

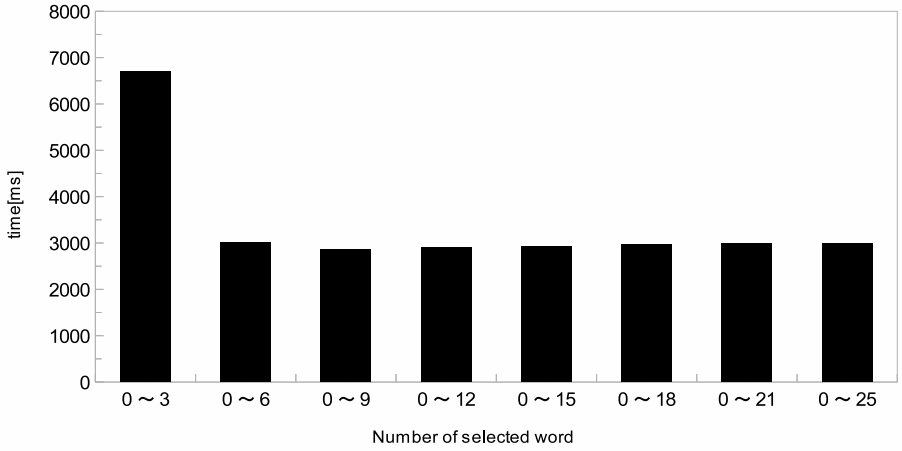

図 10　各範囲で EN=10 としたときの所要時間

5　まとめ

本実験を通してプログラムの実行時間は主に EN の値に依存しており，要素数 25 において EN は選択済みの符号語の数が 0〜9 個の間大きい値を用い，符号語の数が 10〜25 の間 EN=0 とするとプログラムを高速化できると考えられる．今回は要素数 25 を主として実験を行ったが要素数によって最適な EN の取り方があると考えられるので今後は様々な要素数についても実験していく必要がある．

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] Yuh-Ming Hung，Ting-Yi Wu，and Yunghsiang S .Han，“An A*-Based Algorithm for Constructing Reversible Variable Length Codes with Minimum Average Codeword Length，” IEEE Transactions on Communications, vol.58, no.11, pp.3175–3185, Nov. 2010.

付録 A　ソースコード

A.1　4.2 節におけるプログラム 1

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


struct state {
    int fixed;
    char **codeword;
    double cost;
};


int alphabet_size;
double *prob;


void
    printstate(struct state *p)
{
    int i;

    for (i = 0; i < p->fixed; i++){
        printf("%s ", (p->codeword[i])? p->codeword[i]: "(NULL)");
    }
    printf("|");
    for (; i < alphabet_size; i++){
        printf(" %s", (p->codeword[i])? p->codeword[i]: "(NULL)");
    }
    printf("\ncost: %f\n", p->cost);
    return;
}

char *nextword(char *can, int size)
{
    int i;

    for (i = 0; can[i] == '1'; i++){
        ;
    }
    if (can[i] == '\0'){
        if (i + 1 >= size){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        can[i + 1] = '\0';
        for (; i >= 0; i--){
            can[i] = '0';
        }
        return(can);
    }

    for (; can[i] != '\0'; i++){
        ;
    }
    for (i--; can[i] == '1'; i--){
        can[i] = '0';
    }
    can[i] = '1';
```

```
    return(can);
}


int
    isprefix(char *word1, char *word2){
        int len1, len2, i;
        len1 = strlen(word1);
        len2 = strlen(word2);
        if(len1 > len2){
            return(0);
        }

        for (i = 0; i < len1; i++){
            if(word1[i] != word2[i]){
                return(0);
            }
        }
        return(1);
}


int
    issuffix(char *word1, char *word2){
        int len1, len2, i;
        len1 = strlen(word1);
        len2 = strlen(word2);

        if(len1 > len2){
            return(0);
        }

        for(i = 1; i <= len1; i++){
            if(word1[len1 - i] != word2[len2 - i]){
                return(0);
            }
        }
        return(1);
}


struct state
    removecandidate(struct state *pstate)
{
    struct state nstate;
    char can[32];
    int i;
    int tmp = 0;
    nstate.codeword = (char **)malloc(sizeof(char*) * alphabet_size);
    if (nstate.codeword == NULL){
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate.fixed = pstate->fixed;
    for (i = 0; i < nstate.fixed; i++){
        nstate.codeword[i] = strdup(pstate->codeword[i]);
        if (nstate.codeword[i] == NULL){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
    }
    for (i = nstate.fixed; i < alphabet_size - 1; i++){
        nstate.codeword[i] = strdup(pstate->codeword[i + 1]);
        if (nstate.codeword[i] == NULL){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
    }
    strcpy(can, pstate->codeword[alphabet_size - 1]);
```

```
NEXT_CANDIDATE:
    nextword(can, 32);
    for (i = 0; i < nstate.fixed; i++){
        if (isprefix(nstate.codeword[i], can)){
            goto NEXT_CANDIDATE;
        }
        if (issuffix(nstate.codeword[i], can)){
            goto NEXT_CANDIDATE;
        }
    }
    nstate.codeword[alphabet_size - 1] = strdup(can);
    if (nstate.codeword[alphabet_size - 1] == NULL){
        printf("error in %d\n", __LINE__);
        exit(1);
    }

    return(nstate);
}


struct state
    selected_word(struct state *pstate)
{
    struct state nstate;
    char can[32];
    int i, j,k,len1,len2,power,kraft;
    nstate.codeword = (char **)malloc(sizeof(char*) * alphabet_size);
    if (nstate.codeword == NULL){
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate.fixed = pstate->fixed + 1;
    for (i = 0; i < nstate.fixed; i++){
        nstate.codeword[i] = strdup(pstate->codeword[i]);
        if (nstate.codeword[i] == NULL){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
    }
    for (; i < alphabet_size; i++){
        nstate.codeword[i] = NULL;
    }

    j = nstate.fixed;
    k = nstate.fixed;
    for (i = j; i < alphabet_size; i++){
        if (isprefix(nstate.codeword[nstate.fixed - 1],pstate->codeword[i])
                ||issuffix(nstate.codeword[nstate.fixed - 1], pstate->codeword[i])){
            ;
        } else {
            nstate.codeword[k] = strdup(pstate->codeword[i]);
            if (nstate.codeword[j] == NULL){
                printf("error in %d\n", __LINE__);
                exit(1);
            }
            if(j==1 && **nstate.codeword == '1'){
                nstate.fixed += alphabet_size +1;
                return(nstate);
            }
            k++;
        }
    }
    if (j < alphabet_size){
        kraft = 0;
        len1 = strlen(nstate.codeword[nstate.fixed - 1]);
        for( i = 0; j > i; i++){
            len2 = strlen(nstate.codeword[i]);
            power = 1<<(len1 - len2);
            kraft += power;
```

```
        }
        if(kraft == 1<<(len1)){
            nstate.fixed += alphabet_size +1;
            return(nstate);
        }
        strcpy(can, pstate->codeword[alphabet_size - 1]);
    }
    for (; k < alphabet_size; k++){

NEXT_CANDIDATE:
        nextword(can, 32);
        for (i = 0; i < nstate.fixed; i++){

            if (isprefix(nstate.codeword[i],can)||issuffix(nstate.codeword[i], can)){

                goto NEXT_CANDIDATE;

            }
        }
        nstate.codeword[k] = strdup(can);
        if (nstate.codeword[j] == NULL){
            exit(1);
        }
    }
    return(nstate);
}


double
    c_cost(struct state *pstate){
        int i;
        double cost;

        cost = 0;
        for (i = 0; i < alphabet_size; i++){
            cost += strlen(pstate->codeword[i]) * prob[i];
        }
        return(cost);
}

#define LOCALLISTBOUND 1000

double Ubound;
double
    h_estimate(struct state state, int en)
{
    struct state l_list[LOCALLISTBOUND];
    struct state state1, state2;
    int l_num_state,i;

    if (en > LOCALLISTBOUND){
        printf("error in %d\n", __LINE__);
        exit(1);
    }

    l_list[0] = state;
    l_list[0].cost = c_cost(l_list);
    l_num_state = 1;
    while (l_list[l_num_state - 1].fixed < alphabet_size - 1 && en > 0){
        state = l_list[l_num_state - 1];
        l_num_state--;

        state1 = removecandidate(&state);
        if (state1.fixed <= alphabet_size){
            state1.cost = c_cost(&state1);

            for (i = l_num_state; i > 0; i--){
                if (state1.cost < l_list[i - 1].cost){
                    break;
```

```
                }
                l_list[i] = l_list[i - 1];
            }
            l_list[i] = state1;
            l_num_state++;
        }

        state2 = selected_word(&state);
        if (state2.fixed <= alphabet_size){
            state2.cost = c_cost(&state2);

            if(state2.fixed == alphabet_size - 1 && state2.cost < Ubound){
                Ubound = l_list[i].cost;
            }

            for (i = l_num_state; i > 0; i--){
                if (state2.cost < l_list[i - 1].cost){
                    break;
                }
                l_list[i] = l_list[i - 1];
            }
            l_list[i] = state2;
            l_num_state++;
        }
        en--;
    }
    return(l_list[l_num_state - 1].cost);
}


#define LISTBOUND 1000000

void
    initlist(struct state *p)
{
    int i;
    char can[32];
    p->fixed = 0;

    if (p == NULL){
    }

    p->codeword = (char **)calloc(alphabet_size,sizeof(char *));

    if (p->codeword == NULL){
        exit(EXIT_FAILURE);
    }

    if (p->codeword == NULL){
        printf("error in %d\n", __LINE__);
        exit(1);
    }

    can[0] = '\0';

    for (i = 0; i < alphabet_size; i++){
        nextword(can, 32);
        p->codeword[i] = strdup(can);
        if (p->codeword[i] == NULL){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
    }
    return;
}

#define EN 1

struct state list[LISTBOUND];
```

```
void
    findrvlc()
{
    struct state state, state1, state2;
    int num_state;
    int i,k,x;
    int j = 0;
    initlist(list);
    list[0].cost = c_cost(list);
    num_state = 1;
    while (list[num_state - 1].fixed < alphabet_size){
        for (k = 0; k < num_state; k++){
            if (list[k].cost <= Ubound){
                break;
            }
        }
        if (k == num_state){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        if (k > 0){
            for (x = 0; k < num_state; x++, k++){
                list[x] = list[k];
            }
            num_state = x;
        }
        j++;
        state = list[num_state - 1];
        num_state--;
        state1 = removecandidate(&state);
        if (state1.fixed <= alphabet_size){
            if (num_state >= LISTBOUND){
                printf("error in %d\n", __LINE__);
                exit(1);
            }
            state1.cost = h_estimate(state1, EN);
            //state1.cost = c_cost(&state1);
            for (i = num_state; i > 0; i--){
                if (state1.cost < list[i - 1].cost){
                    break;
                }
                list[i] = list[i - 1];
            }
            list[i] = state1;
            num_state++;
        }

        state2 = selected_word(&state);

        if (state2.fixed <= alphabet_size){
            if (num_state >= LISTBOUND){
                printf("error in %d\n",__LINE__ );
                exit(1);
            }
            state2.cost = h_estimate(state2, EN);
            for (i = num_state; i > 0; i--){
                if (state2.cost < list[i - 1].cost){
                    break;
                }

                list[i] = list[i - 1];
            }
            list[i] = state2;
            num_state++;
        }

        for (i = 0; i < alphabet_size; i++){
        }
```

```
        free(state.codeword);
    }
    printstate(list + (num_state - 1));
    return;
}


    main()
{
    double prob_c = 0;
    int i;
    clock_t start, end;

    puts("要素数を入力");
    scanf("%d",&alphabet_size);
    if(alphabet_size>100){
        puts("要素数オーバー");
        return(0);
    }

    prob = (double *)calloc(alphabet_size,sizeof(double));//+

    for(i=0;i<alphabet_size;i++){
        printf("確率 %d を入力",i+1);
        scanf("%lf",&prob[i]);
        if(i > 0){
            if(prob[i] > prob[i - 1]){
                printf("確率は降順で");
                printf("\n");
                return(0);
            }
        }
    }

    for( i = 0; alphabet_size > i; i++){
        prob_c += prob[i];
    }

    Ubound = 1e+300;

    start = clock();

    findrvlc();

    end = clock();

    free(prob);

    printf( "処理時間:%d[ms]", end - start );

    return(0);
}
```