信州大学大学院理工学系研究科

修士論文

最適なリバーシブル可変長符号探索アルゴリズムの高速化の提案

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　12TM245A
氏名　深瀬 拓巳

2014 年 2 月 24 日

目次

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，平均符号語長を小さ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．

現在 RVLC について多くの論文が発表されている．例えば S.Yekhanin は符号語長に関するクラフト和 K が $K \leq 5/8$ となる時 RVLC が存在することを証明し，また符号語長のひとつが 1 で，且つ $K \leq 3/4$ となる時も RVLC が存在することを証明した [1][2]．Ali Kakhbod らは RVLC のエントロピーの上界と下界を求めた [3]．このように RVLC は様々な視点から研究が行われているような注目すべき符号である．

RVLC の構成法についても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 [4]．本研究では，より高速な RVLC 探索アルゴリズムの提案，考察を行う．

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，及び概要の説明を行う．3 ～ 4 章では最適 RVLC 探索アルゴリズムの従来法について，5 章では最適 RVLC 探索アルゴリズムの提案法についての説明を行う．6 ～ 8 章で従来法と提案法の RVLC 構成までの速度比較の実験結果を示し，最後に 9 章にてまとめを行う．

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，接尾辞及び接頭辞フリー符号，接尾辞フリー符号についての説明を記述する．

2.1　接頭辞と接尾辞

任意のアルファベット $\mathcal{A}$ 上の長さ n の記号列 $\boldsymbol{x} = x_1x_2\cdots x_n$ に対し ($x_i \in \mathcal{A}$, $1 \leq i \leq n$)，$\boldsymbol{x}$ 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 $\boldsymbol{x}$ の接頭辞といい，接頭辞とは逆に $\boldsymbol{x}$ 自身を含む末尾部分 (x_n，$x_{n-1}x_n$，$x_{n-2}x_{n-1}x_n$，$\cdots$，$x_2\cdots x_n$，$x_1x_2\cdots x_n$) を接尾辞という．長さ 0 の文字列も $\boldsymbol{x}$ の接頭辞であり，接尾辞である．

2.2　接頭辞フリー符号，接尾辞フリー符号及び RVLC

接頭辞フリー符号とは各々の符号語が他の符号語の接頭辞となっていない符号のことを言う．また接尾辞フリー符号は接頭辞フリー符号とは逆に，各々の符号語が他の符号語の接尾辞となっていない符号のことを言う．

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．

例をあげると，{11，101，1011} は接頭辞フリー符号となっているが 11 が 1011 の接尾辞となっているので接尾辞フリー符号ではない．また {11，110，1101} は接尾辞フリー符号であるが 11 が 1101 の接頭辞となっているので接頭辞フリー符号ではない．これに対し {11，101，1001} は各々の符号語がそれぞれの接頭辞及び接尾辞となっていないので接頭辞フリー符号且つ接尾辞フリー符号である．従って {11，101，1001} は RVLC である．

2.3　本研究の問題設定

一般にシンボル数 N の情報源を考える．シンボルの生起確率 (p_1，p_2，$\cdots$，p_N) は ($p_1 \geq p_2 \geq \cdots \geq p_N$) となっていると仮定する．この情報源に対して最適な RVLC を構成する方法を考える．ここで RVLC が最適であるとは平均符号語長が最も小さいということである．このような問題に対して Huang 等は最適な RVLC を構成するアルゴリズムを提案している [4]．また，文献 [5] は Huang 等が提案した最適な RVLC を構成するアルゴリズムの高速化を行っている．本研究では最適な RVLC 構成するまでの時間の更なるを短縮を目的としている．

3　最適 RVLC 探索アルゴリズム (従来法 1)

本章では Huang 等が提案した最適 RVLC 探索アルゴリズム [4](以降従来法 1 と呼ぶ) についての説明を行う．

3.1　従来法 1 の概要

従来法 1 では，符号語の候補と選択された符号語の 2 つを区別して最適符号の探索を行う．符号語の候補は長さごと辞書順に並んだ文字列の並びであり，確率の大きなシンボルから順に符号語を割り当てていく．探索の初期状態は，符号語が一つも選択されていない状態である．従来法 1 では 1 つの状態から 2 つの「次の状態」が生成される．生成される「次の状態」の内 1 つは候補の列の先頭を RVLC の符号語として採用する場合，もう 1 つは採用しない場合である．例えば初期状態から 2 つの「次の状態」を生成することについて考えると，候補の列は長さごと辞書順に並んでいるので，長さが 1 の文字列の中で辞書順の先頭に当たる 0 を選択する場合，選択しない場合の 2 通りの状態をつくるということである．従来法 1 ではこの 2 つの状態をどちらも生成して保持する．したがって従来法 1 では複数の探索状態を同時に保持する．そして保持している状態の中で最も平均符号語長が小さくなる可能性が高い状態を用いて「次の状態」を生成する．これを繰り返していき，最初に N 個の文字列を候補から RVLC の符号語として選択した符号が最適な RVLC となる．最初に見つかった RVLC が最適であることの理由は後述する．また，以上のことから，この探索アルゴリズムは状態をノードとする 2 進木上の幅優先探索に相当する．

以降の節では，従来法 1 をより詳しく説明する．

3.2　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．この木における各ノードには RVLC の符号語として選択済み符号語，RVLC の符号語として選択する符号語の候補の情報が入っている．また，状態の関係を 2 進木で表わす場合，左側の子ノードは符号語を増やした状態，右側は候補を削除した状態を表わすことにする．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 の符号語として選んだ符号語の列であり，A_W，A_X，A_Y はノード W，X，Y の長さごと辞書順に並んだ符号語の候補の列である．

符号語の候補 A_W 及び RVLC の符号語として選択済みの符号語の列 C_W が入ったノード W から 2 つの子ノード X，Y を作るには次のような手順に従う．まず左側 (X 側) の子ノードを作る場合は A_W の一番左の候補 a_1 を選択された符号語として C_W の一番右側に加えて

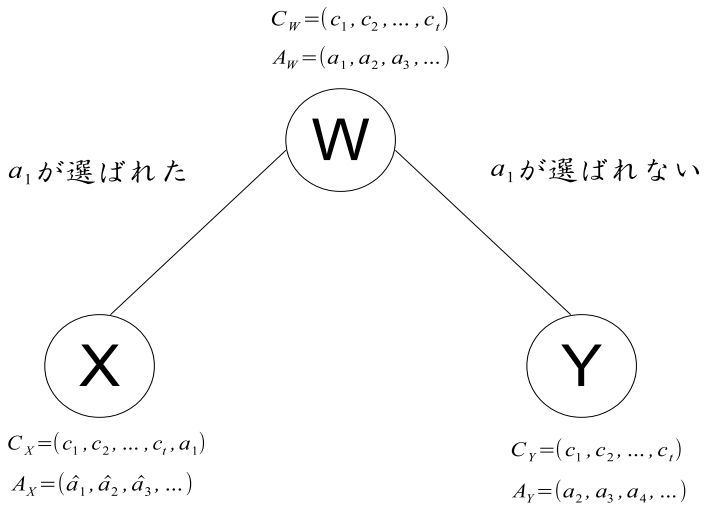

図 1　W からの分岐

C_X とする．このとき C_X 内の符号語の順番が長さの昇順になることに注意されたい．これはシンボルの確率が降順だからである．次に，選択した符号語 a_1 及び a_1 が接頭辞または接尾辞となっているような A_W 内の文字列は RVLC の符号語とはなりえないので，これらを A_W から消去し，A_X とする．A_W から符号語を選択する際，原則として最初に選択する符号語は 0 から始まるものとし，これに反するノードは消去する．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列のみを得ることができればよい．そのため各符号語の列を反転させた符号語を探索する必要は無く，最初の符号語は 0 から始まるものに限定する．

次に右側 (Y 側) の子ノードを作る場合は 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，C_W はそのままの状態で C_Y としたノード Y を生成する．上記の子ノードを生成する手順が終わったら，この子ノードを生成する際に用いた親ノードは保持する必要がないので消去する．

このとき RVLC の符号語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．

以上がノードの生成及び消去の手順である．このようにしてノードの生成を行ったら次に生成済みのノードの中で最も平均符号語長の小さい RVLC を作り得るものを選択し，そのノードを親ノードとして子ノードを生成する．ここで複数あるノードからどのノードを選択して処理するかについて説明する．本研究では「良い符号が見つかる可能性」の指針としてとしてコ

ストというものを用いる．コストとは各ノードから生成出来る RVLC の平均符号語長を見積もった値のことである．図 1 を例にコストの計算方法についての説明をする．まずノード W において既に選択済みの符号語のコスト $g(W)$ を

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

とする．ここで t は RVLC の符号語として選択済みの符号語の数を指し，$l(c_i)$ は i 番目の選択済みの符号語の符号語長である．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 $h(W)$ を

$$h(W) = \sum_{i=t+1}^{N} p_i \times l(a_{i-t}) \tag{2}$$

とする．ノード W のコスト $m(W)$ を

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

と定める．このようにノード W におけるコストは選択済みの符号のコスト $g(W)$ と候補の符号のコスト $h(W)$ を足したものである．従って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．

上記の手順でコストを算出し，最もコストが小さいものを親ノードとして次々と新たなノードを生成していく．そして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．その理由は次の通り．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い．したがって最初に完成した RVLC の平均符号語長は他の任意の RVLC の平均符号語長を超えることはない．

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．

step1. 必要な RVLC の符号語の個数 N と各符号語の生起確率 $(p_1,\ p_2,\ \cdots,\ p_N)$ を入力する．

step2. $C_R = \phi,\ A_R = (a_1,\ a_2,\ a_3,\ a_4,\ a_5,\ a_6, \cdots) = (0,\ 1,\ 00,\ 01,\ 10,\ 11, \cdots)$ 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．

step3. 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 のサイズが N と等しいならば，最適な RVLC として $C_W = (c_1,\ c_2, \cdots,\ c_N)$ が出力されて終了．

step4. ノード W から 2 つの子ノード X と Y を作り，C_X，A_X，C_Y，A_Y を作る．もし $A_X = \phi$ 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．また，C_X の先頭の符号語が 1 から始まる符号語だった場合ノード X を消去する．残りの子ノードのコストを計算する．

step5. 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 step3 に移る．

図 2　従来法 1 の探索アルゴリズム

3.3　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する場合がある．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 [4] 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 RVLC の平均符号語長を用いる．ただし用いる平均符号語長が最適とは限らない．また，事前に少なくとも 1 つの RVLC を見つけなければならない．しきい値を設けたとき，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．

3.4　コスト計算の改善

図 2 のアルゴリズムの step4 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 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 [4]．これを説明するにあたって図 2 の step4 におけるある 1 つのノード Z のコストを算出するときを考える．ノード

Z 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 $h(Z)$ を求めるとき，式 (3) に従って，候補 A_Z 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 では，まずノード Z をルートノードとする仮の木を考える．ここで仮の木のルートノード W' の状態は $C_Z = C_{W'}$，$A_Z = A_{W'}$ である．この仮の木において図 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，保持し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．h-estimate 内でのコスト算出は全て式 (3) を用いて行う．ここでこの分岐の回数 M は図 2 の step1 において定数として定めておく．そして M 回分岐を行った時点で仮の木で生成したノードの中で最小のコストを持つノード V のコストを $m(V)$ とする．$m(V)$ は Z を実際に分岐させた場合のコストなので $m(Z)$ より $m(V)$ の方が Z のコストとして正確である．従って $h(Z) = m(V) - g(Z)$ のように候補のコスト $h(Z)$ を定めた方が正確である．

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，全体として探索時間を短縮することができる．また，無駄なノード生成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 はコストの計算をより正確なものに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，4.3 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 U_{bound} 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．

4 従来法 1 の改良 (従来法 2)

従来法 1 では，図 3 において作り，保持されていたサブリストのノード (状態) はコストの算出が終わった際に全て消去されており，h-estimate を行うその都度新たに生成している．文献 [5] では図 3 において生成し，保持されたノードを消さずに親ノードとリンクさせておく方法が提案されている．ただし [5] では h-estimate で先読みする深さ $M = 1$ の場合だけが考察されている．即ち h-estimate を行う都度図 3 のノード Z には $1 \sim 2$ 個の子ノードが紐付けされることになる．そして図 2 において子ノードが紐付けされているノードが最もコストが小さいノードとなった場合その子ノードをリストに入れてソートする．この時親ノードは消去される．これは紐付けされている子ノードは実際に親ノードを分岐させることによって生成したものであることから可能である．

step1. 図 2 におけるリストとは別に仮に生成したノードを保持するリスト (サブリストとする) を用意する．h-estimate によるコスト算出を行いたいノード Z を入力し，ノード Z とその C_Z，A_Z をコピーし，サブリストに入れる．ここで $i = M$ とする．

step2. サブリストの先頭のノード W'(始めは $W' = Z$) をサブリストから取り出す．もし $C_{W'}$ のサイズが $N - 1$ と等しければ $m(W) = m(V)$ として step4 へ移動．ここで $m(X') < U_{\text{bound}}$ ならば $U_{\text{bound}} = m(X')$ とし図 2 のリストからコストが U_{bound} より大きいのノードを全て消去する．取り出したノード W' の 2 つの子ノード 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．また，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時もノード X' を消去する．ここで親ノードである W' を消去する．

step3. サブリスト内のノードをコストが小さい順に並べ替える．この時のコスト算出方法は式 (3) で示した方法を用いる．$i = i - 1$ とし，$i \neq 0$ ならば step2 に移る．

step4. サブリストの先頭のノードを V とする．ノード Z の新たな候補のコスト $h(Z) = m(V) - g(Z)$ を返して終了．

図 3　より正確に候補のコスト h を求めるためのアルゴリズム (h-estimate)

これを用いることで無駄な計算を省くことができ，高速化が出来ることが文献 [5] で報告されている．

5　符号語長先決め方式の最適 RVLC 探索アルゴリズム (提案法)

従来法 1，従来法 2 では h という推定値を用いており，これは実際に完成する最適 RVLC の平均符号語長と比べてかなり小さく見積もられている．従って選択済みの符号語が多いノードに比べて選択済みの符号語が少ないノードの方がコストが小さくなるケースが多く，選択済みの符号語が少ないノードを優先的に分岐することが増える．これが場合によっては余計なノードの生成となり，処理時間増加につながってしまう．そこで本研究では最適 RVLC の各符号語長と成り得る長さの列を次々と用意していき，それに RVLC の関係を満たす符号語を適用出来るか否かを繰り返し，始めに RVLC が完成したものを最適 RVLC とするアルゴリズムを考えた．以降これを提案法と呼ぶ．提案法では従来法 1，従来法 2 で用いている h という推定値は用いておらず，処理時間短縮が期待できると考えた．この章では提案法についての詳しい説明を行う．

5.1 提案法の概要

提案法では，初期設定として N 個のシンボルに対して $\sum_{i=1}^{N} 2^{-l_i} = 1(l_1 \leq l_2 \leq \cdots \leq l_N)$ を満たすような符号語長の列 $(l_1, \cdots, l_N)$ を全て生成し，保持する．RVLC はクラフト和が 1 以下であることに注意されたい．符号語長の列はまずクラフト和が 1 である列を初期とし，後述するように符号語長を 1 ずつ長くしていき，RVLC を構成できる列を探索する．

初期設定が終了したら，次に保持している長さの列の中で最も平均符号語長が小さい列を取り出し，符号語長にしたがって符号語の割り当てを行う．具体的な割り当て方については後述する．符号語の割り当ては成功するとは限らない．もし割り当てに成功し，RVLC が完成したならばそれが最適 RVLC となり，完成しなかった場合は取り出した列の各長さを 1 ずつ大きくしたものをそれぞれ 1 つの長さの列として生成する．そして $(l_1 \leq l_2 \leq \cdots \leq l_N)$ を満たすものだけを保持する．そして再び最も平均符号語長の小さい長さの列を取り出し，最適 RVLC を見つけるまで試行を繰り返していく．

以上が提案法の概要であり，このアルゴリズムを図 4 に示す．

step1. 必要な RVLC の符号語の個数 N と各符号語の生起確率 $(p_1, p_2, \cdots, p_N)$ を入力する．$\sum_{i=1}^{N} 2^{-l_i} = 1(l_1 \leq l_2 \leq \cdots \leq l_N)$ を満たすような符号語長の列を全て生成する．

step2. 生成済みの符号語長の列の中で最も平均符号語長の小さいものを取り出し，5.3 節の法則に基づき符号を割り当てる．ここで適切に符号語が割り当てられたならばそれを最適 RVLC として出力して終了する．割り当てられなかったならば step3 へ移動する．

step3. step2 において取り出した符号語長の列から 5.4 節の法則に基づいて新たな符号語長の列を生成し，step2 において取り出した符号語長の列は消去する．

step4. step2 へ．

図 4　提案法の探索アルゴリズム

5.2 葉の深さが昇順になるような完全木を全て生成するアルゴリズム

この節では，葉の数が N 個となるような完全木を全て生成するアルゴリズムについての説明を行う．

まず始めに要素数が N であり，長さ $l_1 = 1$ かつ $l_2, l_3, \cdots, l_N$ の長さが全て 0 であるような長さの列を用意する．また初期設定として $i = 2$ とする．

最初の試行として i 番目 (つまり初期状態では 2 番目) の要素 l_i を $l_i = l_i + 1$ とする．

もし $\sum_{j=1}^{i} 2^{-l_j} > 1$ ならば i 番目より後の葉の深さをどのようにしても完全木とはなり得ないので $i = i - 1$ として最初の試行に戻る．また $i = 0$ となったならばアルゴリズムは終了

となる．

ここで $i = N$ かつ $\sum_{i=1}^{N} 2^{-l_i} = 1$ であったならばクラフト和が 1 となるのは完全木であり，また葉の深さが昇順になっているためこの列を採用し，保持する．

次に $l_{i+1} \sim l_N$ の長さが仮に全て l_i となった場合を考える．これは l_i までの長さの値が決まった状態で最もクラフト和が大きくなる場合について考えている．この仮に値を入れた状態でもし $\sum_{i=1}^{N} 2^{-l_i} < 1$ となった場合，これも i 番目より後の葉の深さをどのようにしても完全木とはなり得ないので $i = i - 1$ として最初の試行に戻る．また $i = 0$ となったならばアルゴリズムは終了となる．

上の条件を全て抜けた場合 $i = i + 1$ として最初の試行に戻る．

以上が葉の数が N 枚となるような完全木を全て生成するアルゴリズムの流れである．図 5 にアルゴリズムを示す．

step1. 必要な RVLC の符号語の個数 N を入力値とし，$(l_1 = 1, l_2 = 0, \cdots, l_N = 0)$ のような長さの列を用意する．また $x = 2$ とする．

step2. もし $\sum_{i=1}^{N} 2^{-l_i} = 1$ 且つ $x = N$ ならばこの時の $(l_1, l_2, \cdots, l_N)$ を符号語長の候補の列としてリストに登録する．

step3. $l_x = l_x + 1$ とし，この l_x を $l_{x+1} \sim l_N$ の全てに代入する．もし $x \neq N$ 且つ $\sum_{i=1}^{x} 2^{-l_i} \geq 1$ または，$\sum_{i=1}^{N} 2^{-l_i} < 1$ ならば $x = x - 1$ とし，$x = 0$ ならば終了し，そうでなければ step3 の始めに戻る．

step4. $x = x + 1$ として step2 へ．

図 5　完全木の生成

5.3　符号語の割り当て方

この節では図 4 の step2 において取り出した符号語長の列に符号語を割り当て，RVLC となるのかを判定するアルゴリズムについて説明を行う．

初期設定として長さの列 $(l_1, l_2, \cdots, l_N)$ と割り当てる符号 $(c_1, c_2, \cdots, c_N)$ について，例えば $l_1 = 1$ ならば $c_1 = 0$，$l_1 = 2$ ならば $c_1 = 00$ のように各長さ l に応じて符号語 $(c_1, c_2, \cdots, c_N)$ にシンボル 0 のみからなる符号語を割り当て，また $i = 2$ とする．

まず，c_i(初期状態では c_2) を辞書順で次の符号語に更新する (例えば 00 ならば 01 に 101 ならば 110 となる)．初期状態で $i = 2$ からなのは，このアルゴリズムでは符号語の割り当てを行うが，各符号語の長さは始めから決まっているので，例えば c_1 が 00 である場合と 01 である場合どちらも符号語を割り当てて RVLC になるとしても平均符号語長は同じである．従って辞書順で $c_1 = 00$ から始まる符号から先に探すことにしている．

ここで更新した符号語 c_i が $(\cdots, c_{i-2}, c_{i-1})$ と RVLC になっているかを確認する．

もし RVLC の関係になっていないならば RVLC の関係になるまで更新を繰り返す．長さ 2 において 00 を更新していき 11 になるというようにもう更新が出来ない状態になった時は c_i は再びシンボル 0 からのみからなる符号語に戻し，$i = i - 1$ として再び c_i の符号語の更新及び RVLC の関係になっているかの確認を繰り返していく．もし c_1 が 1 から始まる符号語になった場合は以降更新を行っても全て 1 から始まる符号語となり，この提案法でも 3 章の従来法 1 と同じく先頭の符号語が 1 から始まるものは考えないとしているので，符号語の割り当てを行っていた長さの列からは RVLC を生成できなかったということになる．

もし RVLC の関係になっていたならば，$i = i + 1$ として再び c_i の符号語の更新及び RVLC の関係になっているかの確認を繰り返していく．c_N まで符号語の割り当てを行い，全ての符号語が RVLC の関係を満たしているならばアルゴリズム終了となる．

以上が長さの列に符号語の割り当てを行う方法である．図 6 にアルゴリズムを示す．

step1. 符号語長の候補の列 $(l_1 \cdots l_N)$ を用意する．$i = 2$ とし，$l_1 \sim l_N$ まで各長さの数だけ 0 を割り当て $c_1 \sim c_N$ を構成する．また $c_1 \sim c_n$ に対応して $x_1 = 2^{l_1} - 1 \sim x_n = 2^{l_N} - 1$ とおく．

step2. c_i を $c_1 \sim c_{i-1}$ に割り当てた各符号語と RVLC の関係になるまで $+1$ し，その度に x_i を -1 する．もし $x_i = 0$ となり，且つ c_i が $c_1 \sim c_{i-1}$ と RVLC の関係になっていないならば c_i に再び長さの数だけ 0 を割り当て，もし $i \neq 0$ ならば i を -1 し step2 へ．$i = 0$ ならば RVLC の構成に失敗して終了．

step3. $i = i + 1$ とする．この時 $i = N + 1$ となったならば RVLC の構成が完了して終了．

step4. step2 へ．

図 6　符号語を割り当てるアルゴリズム

5.4　新たな長さの列の生成

図 4 の step3 における新たな符号語長の列の生成は次のように行う．

まず，長さの列 $(l_1, l_2, \cdots, l_N)$ を用意する．用意した列について各長さに 1 を足したものをそれぞれ 1 つの長さの列とする．つまり $(l_1, l_2, \cdots, l_N + 1)$, $(l_1, \cdots, l_{N-1} + 1, l_N) \cdots (l_1, l_2 + 1, \cdots, l_N)$, $(l_1 + 1, l_2, \cdots, l_N)$ のように列がいくつか生成される．次にこの生成された列の中で $(l_1 \leq l_2 \leq \cdots \leq l_N)$ の関係を満たさないものを消去し，残った列は全て保持する．

これによって始めに用意した列から生成される可能性のある RVLC を全て網羅することが出来る．

6 実験

本論文にて提案した提案法が従来法 1 及び従来法 2 と比較して最適 RVLC 探索に要する時間がどのようになるのかを調べる実験を行った．シンボル数 N の値は処理時間の関係で 2 から 10 とした．

6.1 実験結果

従来法 1 及び従来法 2 に対する提案法の最適 RVLC を構成するまでの処理時間の比を図 7 に示し，シンボル数 2 ～ 6 までのみを表示したものを図 8 に示す．処理時間の比を求めるには，各プログラムがそれぞれ処理を 1000 回繰り返す時間を測定し，その時間の比を取った．また図 7，図 8 の横軸はシンボル数，縦軸は時間比である．但し，図 7 の縦軸は対数スケールで表示している．また凡例は equal が符号語の出現率が全て等しい場合を表し，$1 \times (0.9)^n$ というのは符号語の出現率が 1，0.9，0.81$\cdots$ のような幾何分布を表し，他も同様である．また，A は従来法 1 との比較，B は従来法 2 との比較をしたことを示している．図 8 を見て分かるように符号語数が 2 ～ 4 の時にはほぼ倍率が 1 倍を下回っており，提案法のほうが従来法 1 及び従来法 2 のアルゴリズムに比べて最適 RVLC 生成までに要する時間が短いことが分かる．しかし，図 7 を見ると符号語数が 5 を超えてからは部分的に例外も見られるが概ねどの確率分布であっても処理時間の比が指数的に増加していることが見て取れる．このことより提案法は符号語数が少ない時にのみ非常に効果的であると言える．これは A と B に関して若干の違いは見られたが共通して言えることである．

図 8 のように効果が見られた理由を符号語数 4 を例に考察を行う．符号語数 4 の時に図 4 の step1 において生成される符号語長の列は (1，2，3，3) と (2，2，2，2) の 2 個だけである．図 4 では平均符号語長が小さくなる列に優先的に符号語の割り当てを行うが，例えば (2，2，2，2) の方が平均符号語長が小さい場合これに符号語の割り当てを行うと (00，01，10，11) が割り当てられ，アルゴリズムは終了する．(1，2，3，3) の方が平均符号語長が小さい場合 (1，2，3，3) からは RVLC 生成は不可能であるが (1，2，3，3) から構成される新たな列 (1，2，3，4)，(2，2，3，3)，(1，3，3，3) の内 (1，2，3，4) には (0，11，101，1001)，(2，2，3，3) には (01，10，000，111) のように符号語が割り当てられ，RVLC を生成出来る．このように符号語数 4 では図 4 の step1 において生成される符号語長の列が少なく，また早い段階で多数の RVLC が生成可能な符号語長の列がリスト内に存在することになるので処理速度が向上したと考えられる．符号語数 2，3 に関しても同様のことが言える．

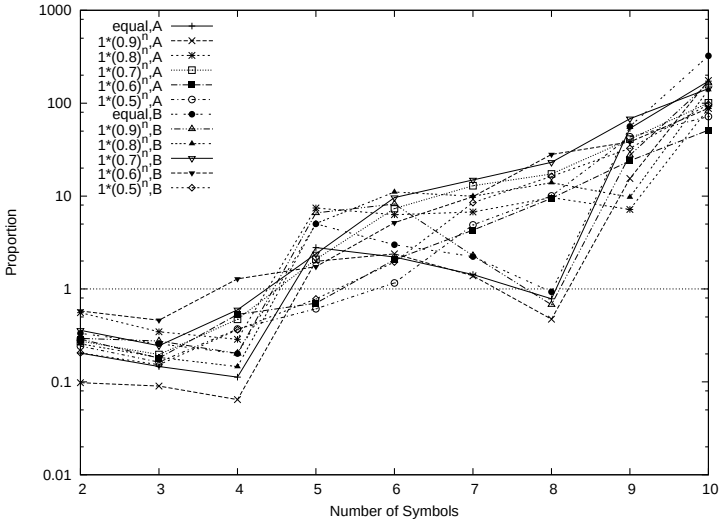

図 7　従来法 1 及び従来法 2 に対する提案法の処理時間の比

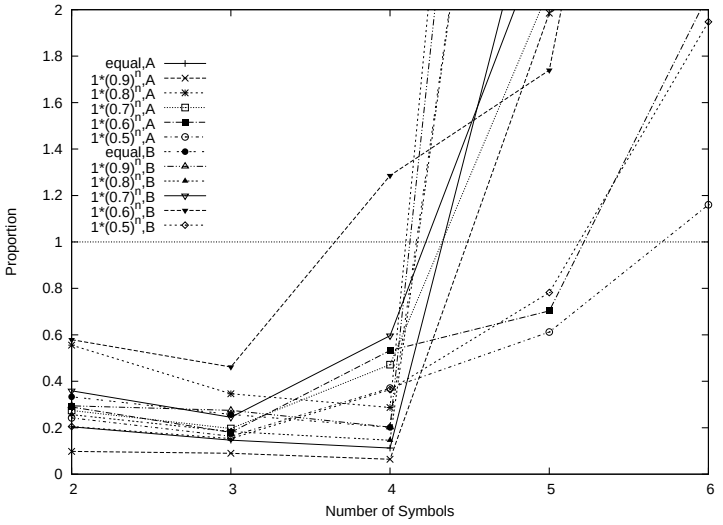

図 8　従来法 1 及び従来法 2 に対する提案法の処理時間の比 (シンボル数 2 ∼ 6)

7　生成する完全木の数

図 7 における速度低下の原因の 1 つが符号語数 N の増加により図 4 の step1 において生成する木が増加したためではないかと考え，符号語数の増加による生成する木の数の増加特性について実験的に確認した．この図 5 を実装したプログラム及びカタラン数の増加特性を実装したプログラムは非常に処理速度が早く，また符号語数 N に対する木の個数は一定なので処理は各 1 回で済むので時間的に余裕が出来る．従って前述の実験では符号語数は 10 としたがここではより特性を分かりやすくするために符号語数 20 で実験を行った．図 9 にその結果を示す．横軸が符号語数，縦軸が生成する完全木の数となっている．$(l_1 \leq l_2 \leq \cdots \leq l_N)$ という制約がない場合の完全木の数はカタラン数 [6] で表わされることが知られている．図 9 には比較のためカタラン数もプロットしている．図 9 を見ると $(l_1 \leq l_2 \leq \cdots \leq l_N)$ という制約がある場合，生成する完全木の数 N_1 は $N_1 \approx 10^{0.254\times N-0.843}$ となっており，また制約無しの場合生成する完全木の数 N_2 は $N_2 \approx 10^{0.574\times N-1.622}$ となっている．これより制約をもたせることにより大きく探索範囲を減らすことが出来ていることも見て取れる．しかし制約を設けることによりカタラン数より小さく抑えられてはいるが，指数的に増加しているので符号語数 N の増加に伴い処理速度も大きく低下してしまっていると考えられる．

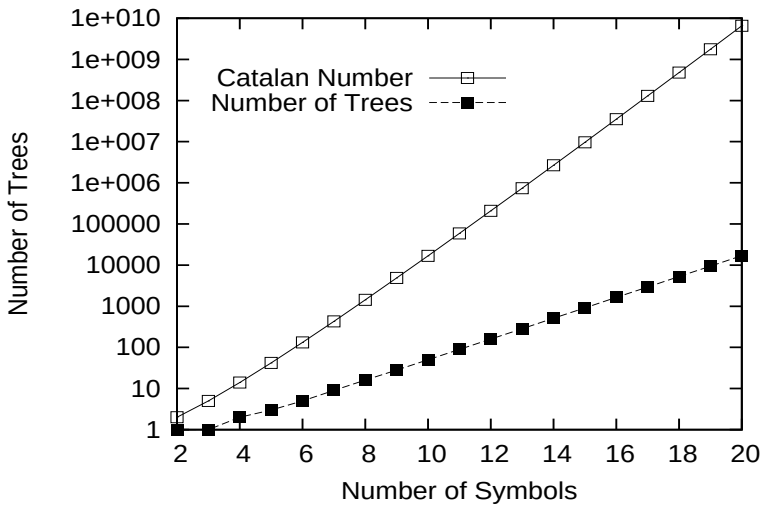

図 9　生成する完全木の数の増加特性

8　ノードを保持するデータ構造としてのヒープ

これまでの実験に用いたアルゴリズムでは，新しく生成したノードをリストに加える際に，挿入ソートを用いてリストの内容をソートされた状態に保っていた．ところが実際には，コストが最小のノードが分かれば十分なので，リストの内容がソートされている必要はない．常に最小値を取り出せる状態にしながら要素の出し入れができるデータ構造としてヒープがある[6]．ヒープは要素全体をソートするわけではないので，リストよりも高速にノードを管理できる可能性がある．そこで，実際に従来法 2 及び提案法のプログラムにヒープを実装して実験を行った．ここで従来法 1 は既に従来法 2 よりも速度が劣ることが分かっているので比較は行わなかった．図 10 が従来法 2 に挿入ソートを用いる場合に対するヒープを用いる場合の処理時間の比，図 11 が提案法に挿入ソートを用いる場合に対するヒープを用いる場合の処理時間の比となっている．これより，従来法 2 は処理時間が改善されることもあるがされないことが多く，ヒープの効果は薄いと考えられる．次に提案法に関しては一部例外もあるが概ね処理時間が短縮されていることから他の分布でもある程度の処理時間短縮の期待が出来る．また符号語数が小さい時も改善されていることから従来法 1 と比較するとかなりの処理時間短縮が出来たといえる．従来法 2 に対してヒープの効果が小さかったのは提案法は符号語数 N が少ない時でも初期設定として $10^{0.254\times N-0.843}$ 個程度のノードを始めから保持するのに対し，従来法 2 は比較的保持するノード数は多くなりづらく，また場合によってはリスト内のノード数が非常に少ない状況で最適 RVLC が見つかることがある．そのため従来法 2 では符号語数 N が少ない時にはヒープの処理を行うよりも単純に挿入ソートをした方が早くなる場合が多いためヒープの効果が出にくいと考えられる．

9　まとめ

符号語数が 4 ～ 5 までに関してはヒープを実装した提案法を，それ以上の場合はヒープを実装していない従来法 2 を適用する方法が最も処理時間を短縮出来る期待値が高いと考えられる．今後処理速度を向上するためにはメモリ内にいくつかの RVLC の割り当てが不可能な列を保存しておき，ハッシュタグをつけるなどして円滑に検索を出来るようにするといった方法が考えられる．

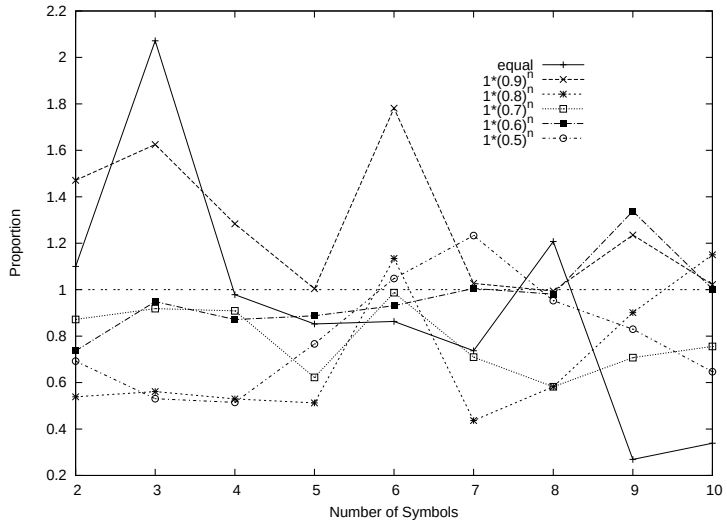

図 10　従来法 2 に挿入ソートを用いる場合に対するヒープを用いる場合の処理時間の比

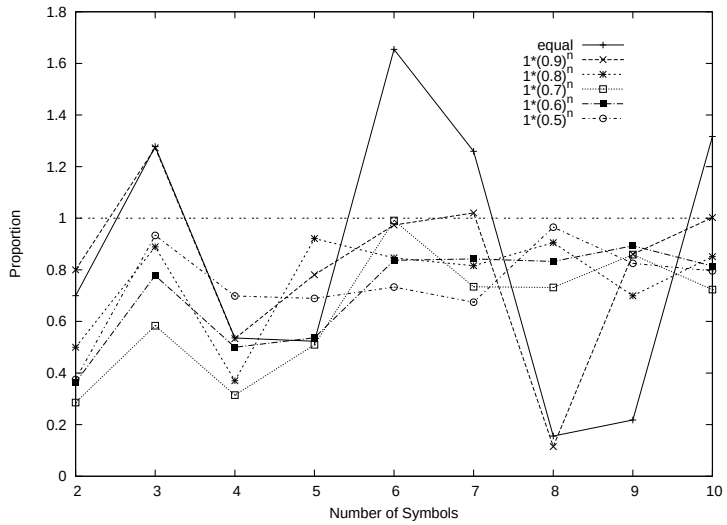

図 11　提案法に挿入ソートを用いる場合に対するヒープを用いる場合の処理時間の比

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] S.Yekhanin，“Sufficient conditions of existence of fix-free codes，” Proc. Int. Symp. On Information Theory，Washington D.C.，pp. 284，June 2001.

[2] S. Yekhanin，“Improved upper bound for the redundancy of fix-free codes，” IEEE Trans. Inform. Theory，vol. 50，no. 11，pp. 2815-2818，Nov. 2004.

[3] Ali Kakhbod，Ali Nazari，and Morteza Zadimoghaddam，“Some Notes On Fix-free Codes，”Information Sciences and Systems, 2008. CISS 2008. 42nd Annual Conference on 19-21 March 2008.

[4] Yuh-Ming Huang, 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.

[5] 深瀬拓巳，西新幹彦，“最適リバーシブル可変長符号の幅優先探索アルゴリズムの高速化の提案”，平成 24 年度 電子情報通信学会信越支部大会 IEEE 信越支部セッション 講演論文集，p5.

[6] R.L.Graham，D.E.Knuth，O.Patashnik，(有澤誠，安村通晃，萩野達也 訳)，コンピュータの数学，共立出版株式会社，1993.

付録 A　ソースコード

A.1　従来法 1 のプログラム

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


struct state {
    int fixed;//選択済みの数
    char **codeword;
    double cost;
};


int alphabet_size,EN;
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;
}

/*****************************次の文字列を生成*****************************************************************/
char *nextword(char *can, int size)
{
    int i;

    for (i = 0; can[i] == '1'; i++){
        ; //何も入れないのもあり
    }
    if (can[i] == '\0'){
        if (i + 1 >= size){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        can[i + 1] = '\0'; /* 1 文字長くなる */
        for (; i >= 0; i--){
            can[i] = '0';
        }
        return(can);
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    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);
}

/*****************************選択しない場合*************************************************************/

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){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    if (pstate == NULL){
        printf("[%d]",__LINE__);
        exit(1);
    }
    nstate.fixed = pstate->fixed;//コピーの作成

    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++){//選択済みの符号と作成した符号の比較
        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);
}

/*********************************************************************************************/

/*****************************選択した場合*****************************************************/

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){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate.fixed = pstate->fixed + 1;//コピーの作成
    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
            ;
        } else {
            nstate.codeword[k] = strdup(pstate->codeword[i]);
            if (nstate.codeword[j] == NULL){
                printf("error in %d\n", __LINE__);
                exit(1);
            }
```

```
            /*最初に選ぶのが 1 ならば*/
            if(j==1 && **nstate.codeword == '1'){
                nstate.fixed += alphabet_size +1;
                return(nstate);
            }
            k++;
        }
    }
    if (j < alphabet_size){
        //クラフト和
        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;
            //free(nstate.codeword);
            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);
}

/*****************************h-estimate*************************************************************/

#define LOCALLISTBOUND 1000/*別のリストを作成*/

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

```
    int l_num_state,i,k;
    double cost = 0;

    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 && 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--;
    }
    cost = l_list[l_num_state - 1].cost;
    for(i = 0;i < l_num_state; i++){
        for(k = 0;k < alphabet_size; k++){
            free(l_list[i].codeword[k]);
        }
        free(l_list[i].codeword);
    }

    return(cost);
}

/*****************************************************************************************/


#define LISTBOUND 1000000

void
    initlist(struct state *p)//初期設定
{
    int i;
```

```
    char can[32];
    p->fixed = 0;//一番最初の state に入力していく↓

    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;
}


struct state list[LISTBOUND];


void
    findrvlc()
{
    struct state state, state1, state2;//state1,state2 は子ノード
    int num_state;//state の数
    int i,j,k,x,y;
    initlist(list);
    list[0].cost = c_cost(list);
    num_state = 1;
    while (list[num_state - 1].fixed < alphabet_size){
        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);

            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){
            state2.cost = -1;
            //printstate(&state2);
            return;
        }

        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 (k = 0; k < num_state; k++){//Ubound と比較して大きいものは消去
            if (list[k].cost <= Ubound){
                break;
            }
        }

        if (k == num_state){
            printf("error in %d\n", __LINE__);
            exit(1);
        }

        x = k;
        y = k;

        if(k > 0){
            for(;k > 0;k--){
                for(i = 0;i > alphabet_size;i++){
                    free(list[k - 1].codeword[i]);
                }
                free(list[k - 1].codeword);
            }

            for(j = 0;j < num_state - y;x++,j++){
                list[j] = list[x];
            }
            num_state = num_state - y;
        }
    }
}


int
    main()
{
    double prob_c = 0;
    int i;
    clock_t start, end;
    //printf("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(prob[i] > 1){
            printf("確率は 1 以下");
            printf("\n");
            return(0);
        }
        if(i > 0){
            if(prob[i] > prob[i - 1]){
                printf("確率は降順で");
                printf("\n");
                return(0);
            }
        }
    }

    EN = 1;

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

    Ubound = 1e+300;

    start = clock();
    for(i = 0; i < 1000; i++){
    findrvlc();
    }
    end = clock();

    free(prob);

    printf( "%d %d\n", alphabet_size,end - start );

    return(0);
}
```

A.2 従来法 1 に heap を実装したプログラム

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


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

#define LOCALLISTBOUND 1000

#define LISTBOUND 1000000

struct state list[LISTBOUND];


int alphabet_size,EN,r_num;
int num_state;
```

```
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++){
        ; //何も入れないのもあり
    }
    if (can[i] == '\0'){
        if (i + 1 >= size){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        can[i + 1] = '\0'; /* 1 文字長くなる */
        for (; i >= 0; i--){
            can[i] = '0';
        }
        return(can);
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    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);//nstate.codeword  に サ イ ズ
alphabet_size の領域を確保
    if (nstate.codeword == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    if (pstate == NULL){

        exit(1);
    }
    nstate.fixed = pstate->fixed;//コピーの作成

    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++){//選択済みの符号と作成した符号の比較
        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){//エラーチェック
        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
            ;
        } else {
            nstate.codeword[k] = strdup(pstate->codeword[i]);
            if (nstate.codeword[j] == NULL){
                printf("error in %d\n", __LINE__);
                exit(1);
            }
            /*最初に選ぶのが 1 ならば*/
            if(j==1 && **nstate.codeword == '1'){
                nstate.fixed += alphabet_size +1;
                return(nstate);
            }
            k++;
        }
    }
    if (j < alphabet_size){
        //クラフト和
        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;
            //free(nstate.codeword);
            return(nstate);
        }
        //クラフト和
        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);
}

/*****************************heap 処理*********************************************************************/

void
    heapsort(struct state input){

        int i = 0,j = 0;
        struct state sub;

        if(num_state == 1){
        list[0] = input;
        }

        list[0] = input;

        while(list[0].cost == list[1].cost || list[0].cost == list[2].cost){

            list[0].cost = list[num_state - 1].cost;

            list[num_state - 1].cost = 0;

            num_state--;

            if(num_state == 1){
            break;
            }
        }

        while(list[2 * i + 1].cost != 0 || list[2 * i + 2].cost != 0){
            if(list[2 * i + 1].cost == 0 || list[2 * i + 2].cost == 0){
                if(list[2 * i + 1].cost == 0){
                    j = 2 * i + 2;
                }else{
                    j = 2 * i + 1;
                }
            }

            if(j == 0){
                if(list[2 * i + 1].cost <= list[2 * i + 2].cost){
                    j = 2 * i + 1;
                }else{
```

```
                j = 2 * i + 2;
            }
        }

        if(list[j].cost > list[i].cost){
            break;
        }

        sub = list[j];
        list[j] = list[i];
        list[i] = sub;

        i = j;
        j = 0;

    }

    return;
}

void
    heapsort2(struct state input){

        int i,j = 0;
        struct state sub;

        i = num_state;

        list[2 * i + 1].cost = 0;

        list[2 * i + 2].cost = 0;

        list[i] = input;

        num_state++;

        while(list[((i - 1) / 2)].cost > list[i].cost){
            sub = list[((i - 1) / 2)];
            list[((i - 1) / 2)] = list[i];
            list[i] = sub;

            i = (i - 1) / 2;

            if(i == 0){
                break;
            }
        }
        return;
}

void
    sort(struct state a){

        //a.cost = c_cost(a);

        if(r_num == 0){
            heapsort(a);
        }else{
            heapsort2(a);
        }

        r_num++;

        return;
}

/*****************************************************************************************/

double
```

```
    h_estimate(struct state state, int en)
{
    struct state l_list[LOCALLISTBOUND];
    struct state state1, state2;
    int l_num_state,i,k;
    double cost = 0;

    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;


        state1 = removecandidate(&state);

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

        }

        state2 = selected_word(&state);


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

        if(state1.cost <= state2.cost){
        cost = state1.cost;
        }else{
        cost = state2.cost;
        }

    return(cost);
}



void
    initlist(struct state *p)//初期設定
{
    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;
}


void
    findrvlc()
{
    struct state state, state1, state2;

    int i,k,j,x = 0,y;
    r_num = 0;
    initlist(list);
    list[0].cost = c_cost(list);
    num_state = 1;
    while (list[num_state - 1].fixed < alphabet_size){

        r_num = 0;

        state = list[0];//一番後ろの配列=コストが最も低いもの
        //num_state--;//始め 0 一つ減らして二つ増やす

        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);

            sort(state1);

        }

        state2 = selected_word(&state);

        if(state2.fixed == alphabet_size){
            state2.cost = -1;
            //printstate(&state2);
            return;
        }

        if (state2.fixed < alphabet_size){
            if (num_state >= LISTBOUND){
                printf("error in %d\n",__LINE__ );
                exit(1);
            }

            state2.cost = h_estimate(state2, EN);


            sort(state2);

        }
    }
}


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

```
    //printf("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(prob[i] > 1){
            printf("確率は 1 以下");
            printf("\n");
            return(0);
        }
        if(i > 0){
            if(prob[i] > prob[i - 1]){
                printf("確率は降順で");
                printf("\n");
                return(0);
            }
        }
    }

    EN = 1;

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



    start = clock();
    for(i = 0; i < 1000; i++){
    findrvlc();
    }
    end = clock();

    free(prob);

    printf( "%d %d\n", alphabet_size,end - start );

    return(0);
}
```

A.3 従来法 2 のプログラム

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


struct state {
    int fixed;
    char **codeword;
    double cost;
    struct state *children[2];
};
```

```
int alphabet_size,EN = 1;
double *prob,Ubound;

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

    if (p == NULL){
        printf("null state\n");
        return;
    }

    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++){
        ; //何も入れないのもあり
    }
    if (can[i] == '\0'){
        if (i + 1 >= size){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        can[i + 1] = '\0'; /* 1 文字長くなる */
        for (; i >= 0; i--){
            can[i] = '0';
        }
        return(can);
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    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 j;
    int tmp = 0;
    //double power,len,kraft;
    nstate = (struct state *)malloc(sizeof(struct state));
    if (nstate == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->children[0] = NULL;
    nstate->children[1] = NULL;
    nstate->codeword = (char **)malloc(sizeof(char*) * alphabet_size);

    if (nstate->codeword == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->fixed = pstate->fixed;//コピーの作成
    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++){//選択済みの符号と作成した符号の比較
        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);
    }
    nstate->cost = -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 = (struct state *)malloc(sizeof(struct state));
    if (nstate == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->children[0] = NULL;
    nstate->children[1] = NULL;
    nstate->codeword = (char **)malloc(sizeof(char*) * alphabet_size);

    if (nstate->codeword == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->fixed = pstate->fixed + 1;//コピーの作成

    for (i = 0; i < nstate->fixed; i++){

        nstate->codeword[i] = strdup(pstate->codeword[i]);//pstate~nstate にコピー

        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->fi
            ;
        } else {
            nstate->codeword[k] = strdup(pstate->codeword[i]);//pstate~nstate にコピー
            if (nstate->codeword[j] == NULL){
                printf("error in %d\n", __LINE__);
                exit(1);
            }

            /*最初に選ぶのが 1 ならば*/
            if(j==1 && **nstate->codeword == '1'){
                return(NULL);
            }
            k++;
        }
    }

    if (j < alphabet_size){
```

```
        //クラフト和
        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)){
            return(NULL);
        }
        //クラフト和
        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);
        }
    }
    nstate->cost = -1;
    return(nstate);
}


void
    initlist(struct state *p)//初期設定
{
    int i;
    char can[32];

    p->fixed = 0;

    p->codeword = (char **)calloc(alphabet_size,sizeof(char *));
    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);
        }
    }
    p->cost = -1;
    p->children[0] = NULL;
    p->children[1] = NULL;

    return;
}
```

```
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);
}

double
    h_estimate(struct state *state,int EN){
        double l_cost,l_cost1,l_cost2;

        if (state->children[0] != NULL){
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (state->children[1] != NULL){
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        state->children[0] = removecandidate(state);
        l_cost1 = c_cost(state->children[0]);

        state->children[1] = selected_word(state);

        if(state->children[1] == NULL){
            return(l_cost1);
        }

        l_cost2 = c_cost(state->children[1]);
        if(l_cost1 <= l_cost2){
            l_cost = l_cost1;
        } else {
            l_cost = l_cost2;
        }


        if(Ubound > l_cost && state->children[1]->fixed == alphabet_size - 1){
            Ubound = l_cost;
        }

        return(l_cost);
}


#define LISTBOUND 1000000
struct state *list[LISTBOUND];

void frvlc()
{
    struct state *state1,*state2;
    int i,k,x,y,j,num_state = 0;

    list[0] = (struct state *)malloc(sizeof(struct state));
    if (list[0] == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    initlist(list[0]);
    num_state++;

    list[0]->cost = h_estimate(list[0],EN);

    while(list[num_state - 1]->fixed < alphabet_size){
        state1 = list[num_state - 1]->children[0];
        state2 = list[num_state - 1]->children[1];
```

```
num_state--;


for(i=alphabet_size - 1;i>=0;i--){
    free(list[num_state]->codeword[i]);
}
free(list[num_state]->codeword);
free(list[num_state]);


state1->cost = h_estimate(state1, EN);

if (state2 != NULL){
    if (alphabet_size == state2->fixed){
        //printstate(state2);
        break;
    }
    state2->cost = h_estimate(state2, EN);
}


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++;
if (num_state >= LISTBOUND){
    printf("error in %d\n",__LINE__ );
    exit(1);
}


if(state2 != NULL){
    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++;
    if (num_state >= LISTBOUND){
        printf("error in %d\n",__LINE__ );
        exit(1);
    }
}

for (k = 0; k < num_state; k++){//Ubound と比較して大きいものは消去
    if (list[k]->cost <= Ubound){
        break;
    }
}

if (k == num_state){
    printf("error in %d\n", __LINE__);
    exit(1);
}

x = k;
y = k;

if(k > 0){
    for(;k > 0;k--){
        for(j = 0;j < alphabet_size;j++){
```

```
                    free(list[k - 1]->children[0]->codeword[j]);
                    if(list[k - 1]->children[1] != NULL){
                        free(list[k - 1]->children[1]->codeword[j]);
                    }
                }

                free(list[k - 1]->children[0]->codeword);
                free(list[k - 1]->children[0]);

                if(list[k - 1]->children[1] != NULL){
                    free(list[k - 1]->children[1]->codeword);
                    free(list[k - 1]->children[1]);
                }
                for(i = 0;i < alphabet_size;i++){
                    free(list[k - 1]->codeword[i]);
                }
                free(list[k - 1]->codeword);
            }
            for(j = 0;j < num_state - y;x++,j++){
                list[j] = list[x];
            }
            num_state = num_state - y;
        }
        /*
        for(i = num_state - 1;i >= 0;i--){
        printstate(list[i]);
        }
        printf("\n");
        */
    }

    return;
}


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

    //printf("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++){

        scanf("%lf",&prob[i]);

        if(i > 0){
            if(prob[i] > prob[i - 1]){
                //printf("確率は降順で");
                //printf("\n");
                return(0);


            }
        }
    }
    //printf("確率 %d を入力",i + 1);
    for( i = 0; alphabet_size > i; i++){
        prob_c += prob[i];
    }
```

```
    Ubound = 1e+300;

    start = clock();
    for(i = 0; i < 1000; i++){
    frvlc();
    }

    end = clock();

    free(prob);

    printf( "%d %d\n",alphabet_size,end - start);

    return(0);
}
```

A.4 従来法 2 に heap を実装したプログラム

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


struct state {
    int fixed;
    char **codeword;
    double cost;
    struct state *children[2];
};


#define LISTBOUND 1000000
struct state *list[LISTBOUND];


int alphabet_size,EN,num_state,r_num;
double *prob;

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

    if (p == NULL){
        printf("null state\n");
        return;
    }

    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++){
        ; //何も入れないのもあり
    }
    if (can[i] == '\0'){
        if (i + 1 >= size){
            printf("error in %d\n", __LINE__);
            exit(1);
        }
        can[i + 1] = '\0'; /* 1 文字長くなる */
        for (; i >= 0; i--){
            can[i] = '0';
        }
        return(can);
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    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 j;
    int tmp = 0;
    //double power,len,kraft;
    nstate = (struct state *)malloc(sizeof(struct state));
    if (nstate == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->children[0] = NULL;
    nstate->children[1] = NULL;
    nstate->codeword = (char **)malloc(sizeof(char*) * alphabet_size);

    if (nstate->codeword == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->fixed = pstate->fixed;//コピーの作成
    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++){//選択済みの符号と作成した符号の比較
        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);
    }
    nstate->cost = -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 = (struct state *)malloc(sizeof(struct state));
    if (nstate == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->children[0] = NULL;
    nstate->children[1] = NULL;
    nstate->codeword = (char **)malloc(sizeof(char*) * alphabet_size);
```

```
    if (nstate->codeword == NULL){//エラーチェック
        printf("error in %d\n", __LINE__);
        exit(1);
    }
    nstate->fixed = pstate->fixed + 1;//コピーの作成

    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->fi
            ;
        } else {
            nstate->codeword[k] = strdup(pstate->codeword[i]);
            if (nstate->codeword[j] == NULL){
                printf("error in %d\n", __LINE__);
                exit(1);
            }

            /*最初に選ぶのが 1 ならば*/
            if(j==1 && **nstate->codeword == '1'){
                return(NULL);
            }
            k++;
        }
    }

    if (j < alphabet_size){
        //クラフト和
        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)){
            return(NULL);
        }
        //クラフト和
        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);
        }
    }

    nstate->cost = -1;
    return(nstate);
}


void
    initlist(struct state *p)//初期設定
{
    int i;
    char can[32];

    p->fixed = 0;

    p->codeword = (char **)calloc(alphabet_size,sizeof(char *));
    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);
        }
    }
    p->cost = -1;
    p->children[0] = NULL;
    p->children[1] = NULL;

    return;
}


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);
}

void
    heapsort(struct state *input){

        int i = 0,j = 0;
        struct state *sub;

        if(num_state == 1){
        list[0] = input;
        }

        list[0] = input;

        while(list[0]->cost == list[1]->cost || list[0]->cost == list[2]->cost){
```

```
            list[0]->cost = list[num_state - 1]->cost;

            list[num_state - 1]->cost = 0;

            num_state--;

            if(num_state == 1){
            break;
            }
        }

        while(list[2 * i + 1]->cost != 0 || list[2 * i + 2]->cost != 0){
            if(list[2 * i + 1]->cost == 0 || list[2 * i + 2]->cost == 0){
                if(list[2 * i + 1]->cost == 0){
                    j = 2 * i + 2;
                }else{
                    j = 2 * i + 1;
                }
            }

            if(j == 0){
                if(list[2 * i + 1]->cost <= list[2 * i + 2]->cost){
                    j = 2 * i + 1;
                }else{
                    j = 2 * i + 2;
                }
            }

            if(list[j]->cost > list[i]->cost){
                break;
            }

            sub = list[j];
            list[j] = list[i];
            list[i] = sub;

            i = j;
            j = 0;

        }

        return;
}

void
    heapsort2(struct state *input){

        int i,j = 0;
        struct state *sub;

        i = num_state;

        list[2 * i + 1] = (struct state*)malloc(sizeof(struct state));

        list[2 * i + 1]->children[0] = NULL;

        list[2 * i + 1]->children[1] = NULL;

        list[2 * i + 1]->cost = 0;

        list[2 * i + 2] = (struct state*)malloc(sizeof(struct state));

        list[2 * i + 2]->children[0] = NULL;

        list[2 * i + 2]->children[1] = NULL;

        list[2 * i + 2]->cost = 0;

        list[i] = input;
```

```
        num_state++;

        while(list[((i - 1) / 2)]->cost > list[i]->cost){
            sub = list[((i - 1) / 2)];
            list[((i - 1) / 2)] = list[i];
            list[i] = sub;

            i = (i - 1) / 2;

            if(i == 0){
                break;
            }
        }
        return;
}

void
    sort(struct state *a){

        a->cost = c_cost(a);

        if(r_num == 0){
            heapsort(a);
        }else{
            heapsort2(a);
        }

        r_num++;

        return;
}


double
    h_estimate(struct state *state,int EN){
        double l_cost,l_cost1,l_cost2;

        if (state->children[0] != NULL){
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (state->children[1] != NULL){
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        state->children[0] = removecandidate(state);
        l_cost1 = c_cost(state->children[0]);

        state->children[1] = selected_word(state);

        if(state->children[1] == NULL){
            return(l_cost1);
        }

        l_cost2 = c_cost(state->children[1]);
        if(l_cost1 <= l_cost2){
            l_cost = l_cost1;
        } else {
            l_cost = l_cost2;
        }

        return(l_cost);
}

void frvlc()
{
    struct state *state1,*state2;
    int i,k,x,y,j;
```

```
EN = 1;
num_state = 0;
r_num = 0;

list[0] = (struct state *)malloc(sizeof(struct state));

list[0]->children[0] = NULL;

list[0]->children[1] = NULL;

list[0]->cost = 0;

    list[1] = (struct state*)malloc(sizeof(struct state));

    list[1]->children[0] = NULL;

list[1]->children[1] = NULL;

    list[1]->cost = 0;

    list[2] = (struct state*)malloc(sizeof(struct state));

    list[2]->children[0] = NULL;

list[2]->children[1] = NULL;

    list[2]->cost = 0;


initlist(list[0]);
num_state++;

list[0]->cost = h_estimate(list[0],EN);

while(list[num_state - 1]->fixed < alphabet_size){
    state1 = list[0]->children[0];
    state2 = list[0]->children[1];

    r_num = 0;

    state1->cost = h_estimate(state1, EN);

    if (state2 != NULL){
        if (alphabet_size == state2->fixed){
            //printstate(state2);
            break;
        }
        state2->cost = h_estimate(state2, EN);
    }

    sort(state1);

    if (num_state >= LISTBOUND){
        printf("error in %d\n",__LINE__ );
        exit(1);
    }

    if(state2 != NULL){

        sort(state2);

        if (num_state >= LISTBOUND){
            printf("error in %d\n",__LINE__ );
            exit(1);
        }
    }

    //
```

```
        //for(i = 0;i < num_state; i++){
        //printstate(list[i]);
        //}
        //printf("\n");

    }

    return;
}


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

    //printf("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++){

        scanf("%lf",&prob[i]);

        if(i > 0){
            if(prob[i] > prob[i - 1]){
                //printf("確率は降順で");
                //printf("\n");
                return(0);


            }
        }
    }
    //printf("確率 %d を入力",i + 1);
    for( i = 0; alphabet_size > i; i++){
        prob_c += prob[i];
    }

    start = clock();
    for(i = 0; i < 1000; i++){
    frvlc();
    }
    end = clock();

    free(prob);

    printf( "%d %d\n",alphabet_size,end - start);

    return(0);
}
```

A.5 提案法のプログラム

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

```
#include <time.h>

struct state{
    int *l_can;
    double cost;
};

struct w_state{
    char *word;
    int n_count;
};

int alphabet_size,num_state = 0;
int num;
double *prob;

#define LISTBOUND 1000000
struct state *list[LISTBOUND];

int
    kraft_sum(int *a,int b){
        int i;
        int sum = 0;

        for(i = 0; i < b; i++){
            sum += 1<<(a[b - 1] - a[i]);
        }

        return(sum);
}

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

    for (i = 0; can[i] == '1'; i++){
        ; //何も入れないのもあり
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    for (i--; can[i] == '1'; i--){
        can[i] = '0';
    }
    can[i] = '1';
    return(can);
}

double
    c_cost(int *c){
        int i;
        double r_cost = 0;

        for(i = 0; i < alphabet_size; i++){
            r_cost += prob[i] * c[i];
        }

        return(r_cost);
}

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);
}


/*****************************符号語の割り当て*****************************************************************/

int
    assign_w(int *len_can){
        int i,j,k,x,y,z;
        struct w_state *w_state;

        w_state = (struct w_state*)malloc(sizeof(struct w_state) * alphabet_size);
        if (w_state == NULL){//エラーチェック
            printf("error in %d\n", __LINE__);
            exit(1);
        }

        for(i = 0; i < alphabet_size; i++){
            w_state[i].word = (char*)malloc(sizeof(char) * (len_can[i] + 1));
            if (w_state[i].word == NULL){//エラーチェック
                printf("error in %d\n", __LINE__);
                exit(1);
            }
        }
        //初期設定　すべてに０を入れて終端入れる
        for(j = 0; j < alphabet_size; j++){
            for(i = 0; i < len_can[j]; i++){
                w_state[j].word[i] = '0';
            }
            w_state[j].word[i] = '\0';
        }

        for(i = 0; i < alphabet_size; i++){
            w_state[i].n_count = 2<<len_can[i] - 1;
        }

        for(i = 1; i < alphabet_size; i++){
        /*  printf("\n");
            for(z = 0; z< alphabet_size; z++){
            printf("[%s]",w_state[z].word);
        }printf("\n");*/

            if(len_can[i] == len_can[i - 1]){
                for(j = 0; j < len_can[i]; j++){
```

```
                w_state[i].word[j] = w_state[i - 1].word[j];
            }
            w_state[i].n_count = w_state[i - 1].n_count;//追加
        }

        for(k = 0,x = 0,y = 0; k < i; k++){
                x += isprefix(w_state[k].word,w_state[i].word);
                y += issuffix(w_state[k].word,w_state[i].word);
            }

one_more:
        while(x >= 1||y >= 1){
            if(w_state[i].n_count == 0){
                for(j = 0; j < len_can[i]; j++){
                    w_state[i].word[j] = '0';
                }//戻す
                w_state[i].n_count = 2<<len_can[i] - 1;
                i--;
                goto one_more;
            }
            nextword(w_state[i].word,len_can[i]);
            w_state[i].n_count--;//追加
            if(w_state[0].word[0] == '1'){
                //printf("1");
                return(1);
            }
            for(k = 0,x = 0,y = 0; k < i; k++){
                x += isprefix(w_state[k].word,w_state[i].word);
                y += issuffix(w_state[k].word,w_state[i].word);
            }
        }
    }

    /*for(z = 0; z< alphabet_size; z++){
        printf("[%s]",w_state[z].word);
    }printf("\n");*/
    //printf("0");
    return(0);
}


/*****************************************************************************************/


/*****************************リストへの登録*************************************************************/
void
    registry(){
        int i,j,k = 0;
        struct state *sub;

        list[num_state]->cost = c_cost(list[num_state]->l_can);

        for(i = num_state; i > 0; i--){
            if(list[i]->cost == list[i - 1]->cost){
                for(j = 0; j < alphabet_size; j++){
                    if(list[i]->l_can[j] != list[i - 1]->l_can[j]){
                        break;
                    }
                }
                if(j == alphabet_size){
                    //free
                    for(; i < num_state; i++){
                        list[i] = list[i + 1];
                    }
                    k++;
                    break;
                }
            }
```

```
            if(list[i]->cost > list[i - 1]->cost){
                sub = list[i - 1];
                list[i - 1] = list[i];
                list[i] = sub;
            }else{
            break;
            }
        }

        if(k != 1){
            num_state++;
        }

        return;
}


/*****************************************************************************************/


/*****************************初期設定*****************************************************/

void
    init(){
        int i,j = 0;//j は何個入ってるか
        int k;
        int *sub;
        struct state *sub2;

        sub = (int*)malloc(sizeof(int) * alphabet_size);

        sub[0] = 1;

        do{
            j++;
            while(kraft_sum(sub,j) >= 1<<(sub[j - 1])){
                sub[j - 1]++;//sub[j]++ ではない
            }
NEXT_STEP:
            for(k = j; k < alphabet_size; k++){
                sub[k] = sub[j - 1];
            }

        }while(kraft_sum(sub,alphabet_size) > 1<<(sub[alphabet_size - 1]));

        if(kraft_sum(sub,alphabet_size) == 1<<(sub[alphabet_size - 1])){
            //登録
            list[num_state] = (struct state*)malloc(sizeof(struct state));

            list[num_state]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

            for(i = 0; i < alphabet_size; i++){
                list[num_state]->l_can[i] = sub[i];
            }

            list[num_state]->cost = c_cost(list[num_state]->l_can);


            for(i = num_state; i > 0; i--){
                if(list[i]->cost < list[i - 1]->cost){
                    break;
                }
                sub2 = list[i];
                list[i] = list[i - 1];
                list[i - 1] = sub2;

            }
            num_state++;
        }
```

```
        if(j > 1){
            j--;
            sub[j - 1]++;//sub[j]++ ではない
            goto NEXT_STEP;
        }

        /*for(i = num_state; i > 0; i--){
            for(k = 0; k < alphabet_size; k++){
                printf("[%d]",list[i - 1]->l_can[k]);
            }
            printf("\n");
        }*/

        return;
}


/*****************************************************************************************/

void
    c_len(){
        int i,j;
        struct state *sub;

        init();

        /*list[0] = (struct state*)malloc(sizeof(struct state));
        list[0]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        for(i = 0; i < alphabet_size; i++){
            list[0]->l_can[i] = 1;
        }

        num_state++;*/
        /*RVLC が出来るかを判定し出来なければ新たな符号語長の列を生成し，登録する*/
        while(assign_w(list[num_state - 1]->l_can) == 1){
            /*for(i = num_state - 1; i >= 0; i--){
                for(j = 0; j < alphabet_size; j++){
                    printf("[%d]",list[i]->l_can[j]);
                }printf("\n");
            }
            printf("\n");*/

            sub = list[num_state - 1];

            num_state--;

            for(i = 0; i < alphabet_size - 1; i++){
                if(sub->l_can[i] + 1 <= sub->l_can[i + 1]){
                    list[num_state] = (struct state*)malloc(sizeof(struct state));
                    list[num_state]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

                    for(j = 0; j < alphabet_size; j++){
                        list[num_state]->l_can[j] = sub->l_can[j];
                    }
                    list[num_state]->l_can[i] = sub->l_can[i] + 1;

                    registry();

                }
            }

            list[num_state] = (struct state*)malloc(sizeof(struct state));
            list[num_state]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

            for(j = 0; j < alphabet_size; j++){
                list[num_state]->l_can[j] = sub->l_can[j];
            }
```

```
            list[num_state]->l_can[alphabet_size - 1]++;

            registry();

        }
        return;
}

int
    main(){
        int i;
        clock_t start, end;


        //printf("要素数を入力");
        scanf("%d",&alphabet_size);

        prob = (double*)malloc((sizeof(double)) * alphabet_size);
        if (prob == NULL){//エラーチェック
            printf("error in %d\n", __LINE__);
            exit(1);
        }

        for(i = 0; i < alphabet_size; i++){
            //printf("確率 %d を入力",i + 1);
            scanf("%lf",&prob[i]);
            if(i != 0 && prob[i] > prob[i - 1]){
                printf("確率は昇順");
                return(0);
            }
        }

        start = clock();
        for(i =0; i < 1000; i++){
        c_len();
        }
        end = clock();

        free(prob);

        printf( "%d %d\n",alphabet_size,end - start);

        return(0);
}
```

A.6 提案法に heap を実装したプログラム

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

struct state{
    int *l_can;
    double cost;
};

struct w_state{
    char *word;
    int n_count;
};

int alphabet_size,num_state;
int num,r_num;
```

```
double *prob;

#define LISTBOUND 1000000
struct state *list[LISTBOUND];

double
    c_cost(int *c){
        int i;
        double r_cost = 0;

        for(i = 0; i < alphabet_size; i++){
            r_cost += prob[i] * c[i];
        }

        return(r_cost);
}

/*****************************heap処理*********************************************************************/

void
    heapsort(double input){

        int i = 0,j = 0;
        struct state *sub;

        list[0]->cost = input;

        while(list[0]->cost == list[1]->cost || list[0]->cost == list[2]->cost){

            list[0]->cost = list[num_state - 1]->cost;

            list[num_state - 1]->cost = 0;

            num_state--;

            if(num_state == 1){
            break;
            }
        }

        while(list[2 * i + 1]->cost != 0 || list[2 * i + 2]->cost != 0){
            if(list[2 * i + 1]->cost == 0 || list[2 * i + 2]->cost == 0){
                if(list[2 * i + 1]->cost == 0){
                    j = 2 * i + 2;
                }else{
                    j = 2 * i + 1;
                }
            }

            if(j == 0){
                if(list[2 * i + 1]->cost <= list[2 * i + 2]->cost){
                    j = 2 * i + 1;
                }else{
                    j = 2 * i + 2;
                }
            }

            if(list[j]->cost > list[i]->cost){
                break;
            }

            sub = list[j];
            list[j] = list[i];
            list[i] = sub;

            i = j;
            j = 0;

        }
```

```
        return;
}

void
    heapsort2(double input){

        int i,j = 0;
        struct state *sub;

        i = num_state;

        list[2 * i + 1] = (struct state*)malloc(sizeof(struct state));

        list[2 * i + 1]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        list[2 * i + 1]->cost = 0;

        list[2 * i + 2] = (struct state*)malloc(sizeof(struct state));

        list[2 * i + 2]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        list[2 * i + 2]->cost = 0;

        list[i]->cost = input;

        num_state++;

        while(list[((i - 1) / 2)]->cost > list[i]->cost){
            sub = list[((i - 1) / 2)];
            list[((i - 1) / 2)] = list[i];
            list[i] = sub;

            i = (i - 1) / 2;

            if(i == 0){
                break;
            }
        }

        return;
}

void
    sort(struct state *a){

        a->cost = c_cost(a->l_can);

        if(r_num == 0){
            heapsort(a->cost);
        }else{
            heapsort2(a->cost);
        }

        r_num++;

        return;
}

/*****************************************************************************************/

int
    kraft_sum(int *a,int b){
        int i;
        int sum = 0;

        for(i = 0; i < b; i++){
            sum += 1<<(a[b - 1] - a[i]);
        }
```

```
        return(sum);
}

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

    for (i = 0; can[i] == '1'; i++){
        ; //何も入れないのもあり
    }

    for (; can[i] != '\0'; i++){ /* 終端を探す */
        ;
    }
    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);
}

int
    assign_w(int *len_can){
        int i,j,k,x,y,z;
        struct w_state *w_state;

        w_state = (struct w_state*)malloc(sizeof(struct w_state) * alphabet_size);
        if (w_state == NULL){//エラーチェック
            printf("error in %d\n", __LINE__);
            exit(1);
        }
```

```
        for(i = 0; i < alphabet_size; i++){
            w_state[i].word = (char*)malloc(sizeof(char) * (len_can[i] + 1));
            if (w_state[i].word == NULL){//エラーチェック
                printf("error in %d\n", __LINE__);
                exit(1);
            }
        }
        //初期設定  すべてに０を入れて終端入れる
        for(j = 0; j < alphabet_size; j++){
            for(i = 0; i < len_can[j]; i++){
                w_state[j].word[i] = '0';
            }
            w_state[j].word[i] = '\0';
        }

        for(i = 0; i < alphabet_size; i++){
            w_state[i].n_count = (2<<len_can[i]) - 1;
        }

        for(i = 1; i < alphabet_size; i++){
        /*  printf("\n");
            for(z = 0; z< alphabet_size; z++){
            printf("[%s]",w_state[z].word);
        }printf("\n");*/

            if(len_can[i] == len_can[i - 1]){
                for(j = 0; j < len_can[i]; j++){
                    w_state[i].word[j] = w_state[i - 1].word[j];
                }
                w_state[i].n_count = w_state[i - 1].n_count;//追加
            }

            for(k = 0,x = 0,y = 0; k < i; k++){
                    x += isprefix(w_state[k].word,w_state[i].word);
                    y += issuffix(w_state[k].word,w_state[i].word);
                }

one_more:
            while(x >= 1||y >= 1){
                if(w_state[i].n_count == 0){
                    for(j = 0; j < len_can[i]; j++){
                        w_state[i].word[j] = '0';
                    }//戻す
                    w_state[i].n_count = (2<<len_can[i]) - 1;
                    i--;
                    goto one_more;
                }
                nextword(w_state[i].word,len_can[i]);
                w_state[i].n_count--;//追加
                if(w_state[0].word[0] == '1'){
                    //printf("1");
                    return(1);
                }
                for(k = 0,x = 0,y = 0; k < i; k++){
                    x += isprefix(w_state[k].word,w_state[i].word);
                    y += issuffix(w_state[k].word,w_state[i].word);
                }
            }
        }

        /*for(z = 0; z< alphabet_size; z++){
            printf("[%s]",w_state[z].word);
        }printf("\n");*/
        //printf("0");
        return(0);
}

void
    registry(){
```

```
        int i,j,k = 0;
        struct state *sub;

        list[num_state]->cost = c_cost(list[num_state]->l_can);

        for(i = num_state; i > 0; i--){
            if(list[i]->cost == list[i - 1]->cost){
                for(j = 0; j < alphabet_size; j++){
                    if(list[i]->l_can[j] != list[i - 1]->l_can[j]){
                        break;
                    }
                }
                if(j == alphabet_size){
                    //free
                    for(; i < num_state; i++){
                        list[i] = list[i + 1];
                    }
                    k++;
                    break;
                }
            }

            if(list[i]->cost > list[i - 1]->cost){
                sub = list[i - 1];
                list[i - 1] = list[i];
                list[i] = sub;
            }else{
            break;
            }
        }

        if(k != 1){
            num_state++;
        }

        return;
}

void
    init(){
        int i,j = 0;//j は何個入ってるか
        int k;
        int *sub;

        sub = (int*)malloc(sizeof(int) * alphabet_size);

        sub[0] = 1;

        list[0] = (struct state*)malloc(sizeof(struct state));

        list[0]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        list[0]->cost = 0;

        list[1] = (struct state*)malloc(sizeof(struct state));

        list[1]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        list[1]->cost = 0;

        list[2] = (struct state*)malloc(sizeof(struct state));

        list[2]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        list[2]->cost = 0;

        do{
            j++;
            while(kraft_sum(sub,j) >= 1<<(sub[j - 1])){
```

```
                sub[j - 1]++;//sub[j]++ ではない
            }
NEXT_STEP:
            for(k = j; k < alphabet_size; k++){
                sub[k] = sub[j - 1];
            }

        }while(kraft_sum(sub,alphabet_size) > 1<<(sub[alphabet_size - 1]));

        if(kraft_sum(sub,alphabet_size) == 1<<(sub[alphabet_size - 1])){
            //登録
            if(num_state > 2){
            list[num_state]->l_can = (int*)malloc(sizeof(int) * alphabet_size);
            }

            for(i = 0; i < alphabet_size; i++){
                list[num_state]->l_can[i] = sub[i];
            }

            list[num_state]->cost = c_cost(list[num_state]->l_can);

            sort(list[num_state]);

            if(num_state == 0)
                num_state++;
        }

        if(j > 1){
            j--;
            sub[j - 1]++;//sub[j]++ ではない
            goto NEXT_STEP;
        }

        /*for(i = 0; i < num_state; i++){
            for(k = 0; k < alphabet_size; k++){
                printf("[%d]",list[i]->l_can[k]);
            }
            printf("\n");
        }*/

        r_num = 0;

        return;
}

void
    c_len(){
        int i,j,*sub;

        num_state = 0;
        r_num = 0;

        init();

        /*list[0] = (struct state*)malloc(sizeof(struct state));
        list[0]->l_can = (int*)malloc(sizeof(int) * alphabet_size);

        for(i = 0; i < alphabet_size; i++){
            list[0]->l_can[i] = 1;
        }

        num_state++;*/

        while(assign_w(list[0]->l_can) == 1){
            /*for(i = num_state - 1; i >= 0; i--){
                for(j = 0; j < alphabet_size; j++){
                    printf("[%d]",list[i]->l_can[j]);
                }printf("\n");
            }
```

```
            printf("\n");*/

            r_num = 0;


            //sub はコピーをつくる
            sub = (int*)malloc(sizeof(int) * alphabet_size);

            for(i = 0; i < alphabet_size; i++){
                sub[i] = list[0]->l_can[i];
            }


            for(i = 0; i < alphabet_size - 1; i++){
                if(sub[i] + 1 <= sub[i + 1]){

                    for(j = 0; j < alphabet_size; j++){
                        if(r_num != 0){
                        list[num_state]->l_can[j] = sub[j];
                        }
                    }
                    if(r_num == 0){
                    list[0]->l_can[i]++;
                    }else{
                    list[num_state]->l_can[i] = sub[i] + 1;
                    }

                    if(r_num == 0){
                    sort(list[0]);
                    }else{
                    sort(list[num_state]);
                    }

                }
            }

            for(j = 0; j < alphabet_size; j++){
                list[num_state]->l_can[j] = sub[j];
            }
            list[num_state]->l_can[alphabet_size - 1]++;

            sort(list[num_state]);

        }
        return;
}

int
    main(){
        int i;
        clock_t start, end;


        //printf("要素数を入力");
        scanf("%d",&alphabet_size);

        prob = (double*)malloc((sizeof(double)) * alphabet_size);
        if (prob == NULL){//エラーチェック
            printf("error in %d\n", __LINE__);
            exit(1);
        }

        for(i = 0; i < alphabet_size; i++){
            //printf("確率 %d を入力",i + 1);
            scanf("%lf",&prob[i]);
            if(i != 0 && prob[i] > prob[i - 1]){
                printf("確率は昇順");
                return(0);
            }
```

```
        }

        start = clock();
        for(i =0; i < 1000; i++){
        c_len();
        }
        end = clock();

        free(prob);

        printf( "%d %d\n",alphabet_size,end - start);

        return(0);
}
```