信州大学工学部

学士論文

算術符号のための順序保存性のない最適な区間分割を探索するアルゴリズムの提案

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　14T2017B
氏名　打田 尚大

2018 年 2 月 16 日

目次

1　はじめに

1.1　研究背景

現代の情報社会では通信で送られるデータ量は膨大であり，データの圧縮が必要不可欠となっている．データ圧縮とは広い意味では情報源系列の符号化のことであるが，その汎用的な方法のひとつとして算術符号がある．算術符号は圧縮効率が高いことが知られている．また，算術符号は順序保存性のある符号として知られているが，本研究では順序保存性のない算術符号を考える．順序保存性という制約を緩和することによって算術符号のクラスは拡大することになる．したがって，情報源シンボルの順序を符号の設計パラメーターに取り入れることによって，従来の算術符号よりも性能の良いものを設計できる可能性が出てくる．具体的には，遅延が小さくなるような有限精度の 2 元算術符号に対して，情報源アルファベットの再帰的 2 分割を適用することによって，任意の情報源アルファベットに対する算術符号を考える．したがって，アルファベットの分割が情報源シンボルの順序と符号語を表現している．原理的には，あらゆる再帰的 2 分割の中から最適なものを選ぶことで，最適な算術符号を構成することができる．しかし，これをそのまま実行することは現実的ではない．本研究では，順序保存性のない最適な算術符号の構成に向けて，状態毎に冗長度の最小となる符号の構成法を提案する．提案法は経路探索アルゴリズムの一種である A^* アルゴリズムの符号探索への応用になっている．

1.2　論文の構成

本論文は次のような構成をとる．第 2 章では算術符号の概要を説明し，基礎的考察を述べる．第 3 章では本研究で提案する最適な区間分割の探索アルゴリズムを説明し，最適性を証明する．第 4 章では AIFV 符号との関係性を考察する．最後に 5 章でまとめを行う．

2　算術符号

2.1　算術符号の概要

算術符号 [1] は情報源系列を符号化する汎用的な方法のひとつである．ここでは文献 [2] に基づいて算術符号の概要を説明する．符号化法の原理は，文字列から半開区間への写像に基づいており，入力される情報源系列と出力される符号語系列はそれぞれに対応する区間を介して対応している．文字列から区間への写像は次の特徴をもつ：(1) 長さゼロの文字列に対応する区間は固定（例えば [0,1)）である．これを全区間と呼ぶことにする．(2) 同じ長さの異なる文

字列に対して，それらの区間は互いに素である．(3) 文字列 x とその語頭 x' に対して，x に対応する区間は x' に対応する区間に含まれる．

算術符号は文字列から区間への写像を 2 つもつ．それぞれ情報源系列用と符号語系列用である．それぞれの写像を I_s, I_c と表す．情報源アルファベットと符号アルファベットをそれぞれ $\mathcal{X}$, $\mathcal{Y}$ とする．

符号器の仕事は，与えられた情報源系列 $x \in \mathcal{X}^*$ に対して $I_s(x) \subset I_c(y)$ となるような符号語系列 $y \in \mathcal{Y}^*$ を求めることである．一般にそのような y は複数存在するが，最も長いものが唯一あり，他はそれの語頭になっている．写像 I_s の特徴から，情報源系列 x の先頭から順に文字を見ることによって，$I_s(x)$ を逐次的に求めていくことができる．すると，I_c の特徴から，符号語系列 y を先頭の文字から逐次的に決定していくことができる．

復号器の仕事は符号器と逆で，与えられた符号語系列 y に対して $I_c(y) \subset I_s(x)$ となるような情報源系列 x を求めることである．符号器同様，復号器も符号語系列 y の先頭から順に文字を見ることによって，情報源系列 x を先頭の文字から逐次的に決定していくことができる．

符号器と復号器が逐次的であることから，算術符号は逐次符号に分類することができる．また，上記の説明から，復号器の出力が符号器への入力の語頭になることは明らかであり，両者の長さの違いが逐次符号としての符号化遅延となる．情報源シンボルの確率分布 p が与えられた場合，写像 I_s として

$$p(a) = \frac{|I_s(xa)|}{|I_s(x)|}, \quad x \in \mathcal{X}^*, a \in \mathcal{X} \tag{1}$$

を満たすものを考えるのが通常である．なぜなら，式 (1) の両辺が表す分布の間のダイバージェンスが符号の冗長度を決めるからである．ここで一般に，$\mathcal{X}$ 上の 2 つの分布 p, q の間のダイバージェンスは

$$D(p||q) \triangleq \sum_{a \in \mathcal{X}} p(a) \log_2 \frac{p(a)}{q(a)} \tag{2}$$

と定義される．式 (1) の右辺はシンボル a の符号化確率と呼ばれる．符号化確率は区間の分割点に基づく分布であるということができる．

算術符号を現実的な場面で使用するには，区間の精度について考えなければならない．これは区間の端点を何桁の 2 進数で表現するかということである．言い換えると，区間の精度が k ビットであれば全区間は 2^k 個に分割され，$2^k - 1$ 個の分割点をもつ．このとき式 (1) を満たす写像 I_s は一般には存在しない．データ圧縮の目的からすると，情報源シンボルの確率分布と符号化確率分布の間のダイバージェンスを小さくしたい．そのためには区間の精度は高い方がよい．情報源系列のシンボルを先頭から順に見るごとに区間は狭くなっていくことから，区間の幅を広く保つための方法が複数提案されている．

まず基本的な対処として，符号語系列の文字がひとつ決まったときに，情報源系列の区間も

符号語の区間も同じだけ拡大させることによって，区間の精度を維持する方法がある．しかし，情報源系列の文字を見ても符号シンボルを決めることができず，情報源系列の区間だけがどんどん狭くなっていく場合がある．

区間の幅を広く保つのは符号化レートをよくするためだけでなく，符号化を継続するのにも必要であると考えられている．つまり，とても狭い区間を表現する場合，精度の問題で丸め誤差が生じ，結果的に区間の幅がゼロになってしまう場合がある．この場合，符号化を継続することは不可能である．これを避けるためには区間の幅を広く保てばよい．

ところが，情報源アルファベットのサイズが 2 より大きい場合（可算無限の場合を含む），情報源シンボルを一度 2 元符号化し，その結果得られた 2 元系列に算術符号を適用すれば，他に特段の工夫をすることなく，区間の幅がゼロになることを避けることができる [2]．言い換えれば，情報源シンボルを直接符号化するのではなく，情報源アルファベットを 2 分するグループを作成し，段階的に符号化すればよい．本研究では，この考え方に基づき，小さい精度で任意のサイズのアルファベットを符号化する算術符号のための区間分割を考察する．

2.2 算術符号と状態遷移

算術符号ではひとつの入力シンボルを符号化をする過程で区間が複数回分割される．ひとつの分割はひとつの状態に対応しており，入力されたシンボルにしたがって状態は遷移する．ここで，精度 2 の算術符号における符号化方法と状態の遷移を図 1, 2 に示す具体例を用いて説明する．3 つの情報源アルファベットと 2 つの符号アルファベットをそれぞれ $\mathcal{X} = \{a, b, c\}$, $\mathcal{Y} = \{0, 1\}$ とする．情報源シンボルの確率をそれぞれ 0.1，0.4，0.5 とする．この例では，情報源アルファベットを $\{a, c\}$ と $\{b\}$ の 2 つのグループに分け符号化する．また，分割点は情報源シンボルの確率分布と符号化確率分布の間のダイバージェンスが最小となるように選ぶ．

まず，全区間を $[0, 4)$ とし，これを初期状態とする．すると，$\{a, c\}$ と $\{b\}$ による分割は図 1(1) のようになる．ここでシンボル b が入力されたとすると，[2,4) の区間で符号化され，図 1(1) の状態に戻る．次に，図 1(1) の状態からシンボル a または c が入力されたとすると，a か c かを区別せずに図 1(2) の状態に移る．ここで，図 1(2) の状態の [0,4) を見てシンボルが a ならば [0,1) の区間で符号化され，図 1(1) の状態に戻る．シンボルが c ならば，[1,4) の区間となるが，符号シンボルは決定されずに図 1(3) の状態に移る．次に，図 1(3) の状態で [1,4) の区間を見て，情報源シンボルのグループ化を行う．ここで，シンボル b が入力されたとすると，[1,2) の区間で符号化され図 1(1) の状態に戻る．また，シンボル a または c が入力されたとすると，両者を区別せずに図 1(2) の状態に戻る．以上より，この例では算術符号の区間の状態は図 1 に示した 3 つによって網羅されることが分かる．さらに，それぞれの状態の動きをまとめたものが図 2 の状態遷移図である．

図 1(1) と図 1(3) では次の状態に遷移するために情報源シンボルを入力する必要がある．一

方，図 1(2) では情報源シンボルを入力する必要はなく，図 1(1) または図 1(3) の状態で入力されたシンボルに基づいて次の状態に遷移する．図 1(1), 図 1(3) のような状態を入力待ちの状態と呼ぶことにする．

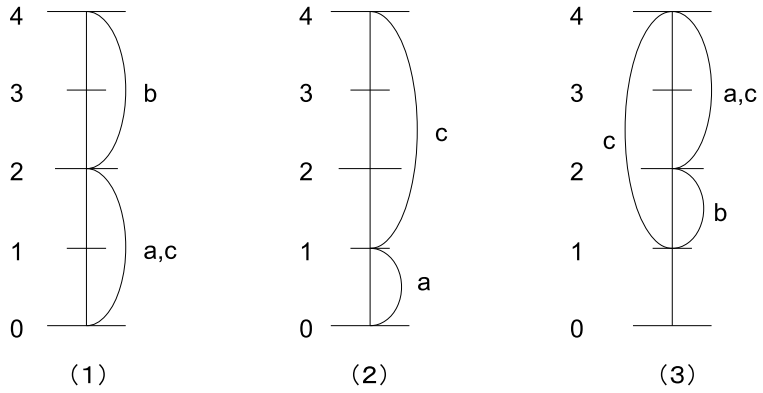

図 1　算術符号の状態

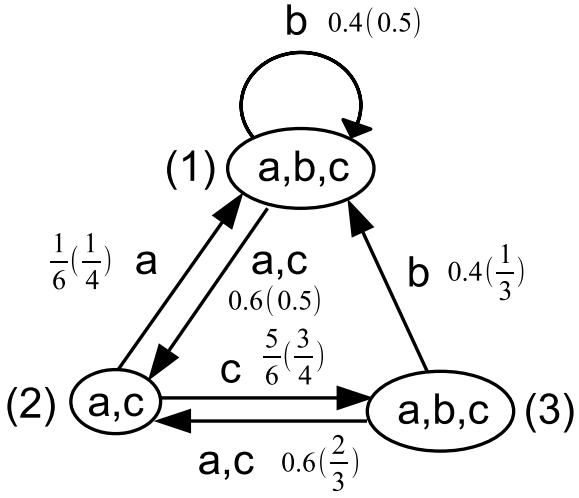

図 2　算術符号の状態遷移

2.3 状態遷移と冗長度

前節のように符号化したときの冗長度について考える．木構造を用いて情報源のエントロピーを求めることができることが知られている [3]．また，木構造の代わりに状態遷移を用いてエントロピーを求めることができる．さらに，エントロピーをダイバージェンスに替えることで符号の冗長度を求めることができる．具体的には，各状態の冗長度と定常確率から符号の冗長度が求められる．各状態の冗長度は情報源シンボルの真の確率と符号化確率の間のダイバージェンスで表される．図 2 では，情報源シンボルの真の確率と括弧付きで符号化確率を示している．状態 i の定常確率とダイバージェンスをそれぞれ Q_i, $D(p_i||q_i)$ とする．入力待ちの状態の定常確率の合計を平均入力長 Q' とすると，符号の冗長度は

$$\frac{1}{Q'}\sum_i Q_i D(p_i||q_i) \tag{3}$$

と表される．図 2 の例では冗長度は 0.039bit になる．以上のように，符号の冗長度は各状態のダイバージェンスから求めることができる．情報源シンボルのグループ化を選択することによって，各状態においてダイバージェンスが最小となる分割点を見つけることができる．次章では，入力待ちの状態のダイバージェンスが最小となる分割点を探るための探索アルゴリズムを提案する．

3 A^* アルゴリズムを用いた区間分割の探索

3.1 A^* アルゴリズムの概要

A^* アルゴリズム [4] とはグラフ探索アルゴリズムのひとつである．A^* アルゴリズムの概要を説明する．まず，関数 g とヒューリスティック関数と呼ばれる h を準備する．また，g と h の合計を関数 f とする．関数 g は初期ノードから現在のノードに到達するためのコストを表す．関数 h は現在のノードから目標ノードに到達するための追加コストの推定値である．したがって，g と h の合計である関数 f は，現在作成したノードを経て初期ノードから目標ノードに到達するためのコストの推定値を表す．

このアルゴリズムでは一般に，それぞれ OPEN, CLOSED と呼ばれる 2 つのノードリストが使われる．OPEN には，引き続き探索が必要なノードが入れられる．また，CLOSED には探索済みとみなされたノードが入れられる．一般のグラフでは，CLOSED の中のノードが再び OPEN に入る場合があるが，グラフが木構造である場合はそのようなことは起きない．したがって，木構造のグラフでは CLOSE というノードリストは必要ない．

初期ノードから目標ノードへのコストの推定値は探索の過程でより正確な値に更新され，そ

れによって最短経路のコスト f^* が求められる．以下に，この考え方を用いた入力待ちの各状態における最適な区間分割の探索アルゴリズムを提案する．

3.2 探索アルゴリズム

本論文で提案するアルゴリズムでは，符号の探索過程を木構造で表し，平均符号語長の推定値に基づいて A^* アルゴリズムを適用することによって，最適な区間分割の探索を行う．区間の探索過程では，確率の大きい情報源シンボルから順に算術符号の区間も大きい順で割り当てていく．どの情報源シンボルにも区間を割り当てていない状態を探索の初期状態とする．初期状態から各情報源シンボルに区間を「割り当てる」か「割り当てない」かの 2 つの状態を生成し，探索を進める．つまり，2 進木の木構造探索である．したがって，CLOSED は必要ない．生成された状態は OPEN に入れられる．また，各状態に対してコストという値を計算する．この値が小さいほど，平均符号語長が小さくなる可能性が高い．OPEN 内のコストが小さい状態から探索を進め，新しい状態を生成する．これを繰り返して，最初に n 個の情報源シンボルに区間を割り当てたときの区間分割が最適となる．最初に見つかった区間分割が最適であることの理由は後述する．

さらに詳しく状態の関係を図 3 を用いて説明する．生成する状態は 2 進木のノードで表される．提案アルゴリズムによって生成するノードの関係を図 3 に示す．それぞれのノードには情報源シンボルに割り当てた区間 C，算術符号として取り扱う区間の候補 A，A^* アルゴリズムにおける初期状態から目標状態に到達するためのコストが入っている．提案アルゴリズムにおけるコストについての詳細は後述する．図 3 において，左側の子ノードは区間の候補を情報源シンボルに割り当てた状態，右側の子ノードは区間の候補を削除した状態を表している．また，A_W, A_X, A_Y は各ノード W, X, Y の算術符号として取り扱う区間の候補を降順に並べている．さらに，C_W, C_X, C_Y は各ノード W, X, Y の情報源シンボルに割り当てた区間の列である．

区間の候補 A について，算術符号の精度 2 における具体的な区間の候補を図 4 に示す．これらの区間の候補はそれらの幅の降順で並んでいる．そのことによって，確率の大きい情報源シンボルから順に算術符号の区間も大きい順で割り当てることが容易にできる．ただし，$[1,4)$ という区間は使用しない．なぜなら，$[0,3)$ と $[1,4)$ という区間はアルゴリズムを進める上で互いに同様の経路をたどり，得られる結果も等しいためである．どちらの区間を使用してもよいが本論文では $[0,3)$ の区間を使用する．また，精度 2 では全区間を 4 分割しかできないので「$\times$」を用いてより細かい区間を表している．例えば，$[0,2)\times[0,3)$ の区間は $[0,\frac{3}{2})$ の区間を指している．この区間を情報源シンボルに割り当てるということは，精度 2 のため算術符号では $[0,2)$ の区間を $[0,4)$ の区間に拡大したもとで $[0,3)$ の区間に情報源シンボルを割り当てると解釈する．

ここで，入力待ちの各状態に対して最適な区間分割を探索していることに注意されたい．区間の候補の列は入力待ちの状態毎に準備する必要がある．図 4 の区間の候補は $[0,4)$ の区間での入力待ちに対する区間の候補である．他にも精度 2 では $[0,3)$ の区間で入力待ちがあり，この状態に対しては $[3,4)$ と共通部分をもつ区間は候補とはしない．

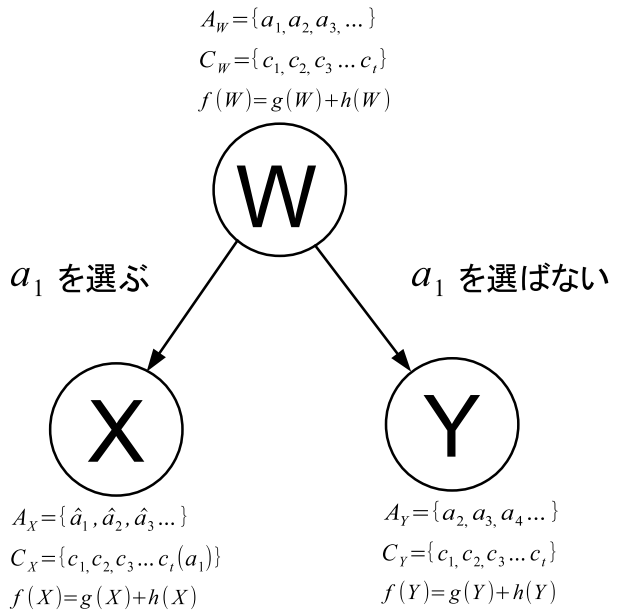

図 3　W からの分岐

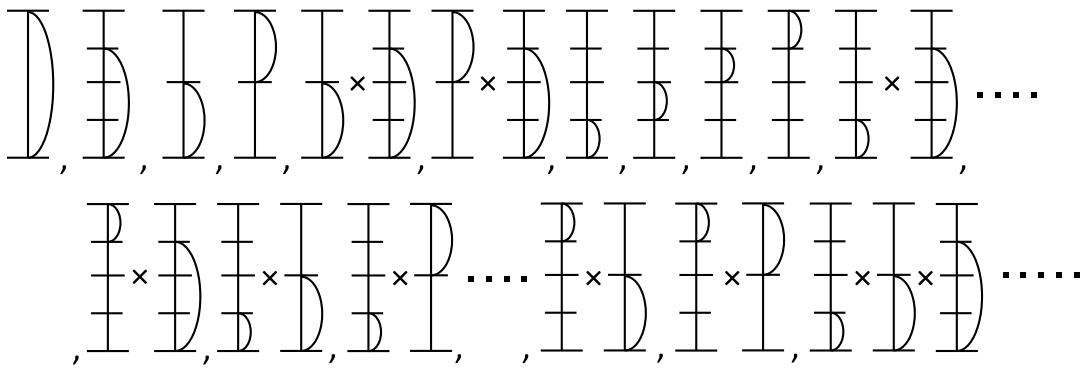

図 4　算術符号の精度 2 における区間の候補 A

算術符号として取り扱う区間の候補 A_W 及び情報源シンボルに割り当てた区間の列 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，X 側の子ノードを作る場合は A_W の一番左の候補 a_1 を選択した区間として C_W の一番右側に加えて C_X とする．このとき C_X 内の順番は区間の長さの降順になることに注意されたい．これは情報源シンボルの生起確率が降順だからである．また，選択した区間 a_1 及び a_1 の区間と重なるような A_W 内の区間は算術符号の区間としては取り扱えないので，これらを A_W から消去し，A_X とする．次に，Y 側の子ノードを作る場合は A_W の一番左の候補 a_1 を選択せずに消去して A_Y とする．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．

上記の生成した 2 つの子ノードは OPEN に入れる．一方，子ノードを生成する際に用いた親ノードは探索済みとなったので消去する．このとき区間が割り当てられていない情報源シンボルが残っているにも関わらず，必要な残りの区間の候補が不足している子ノードは，さらに探索を進めても有効な符号は見つからないので OPEN から消去する．

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 OPEN 内のノードで平均符号語長の小さい算術符号にたどり着く可能性が高いものを選択し，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 OPEN に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．本提案アルゴリズムでは最適な区間分割を見つける指針としてコストという値を用いる．この値は小さいほど平均符号語長が小さくなる可能性が高い．2.3 節で述べたように，入力待ちの状態の冗長度は情報源シンボルの真の確率と符号化確率の間のダイバージェンスで表される．このダイバージェンスの値が最適な区間分割を探る指針となる．ダイバージェンスの式を

$$D(p||q) = \sum_{i=1}^{n} p_i \log_2 \frac{p_i}{q_i} \tag{4}$$

$$= -\sum_{i=1}^{n} p_i \log_2 q_i - \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \tag{5}$$

$$= -\sum_{i=1}^{n} p_i \log_2 q_i - H \tag{6}$$

$$D(p||q) + H = -\sum_{i=1}^{n} p_i \log_2 q_i \tag{7}$$

と変形する．q が符号化確率を表すならば，式 (7) は平均符号語長を表しているとみなせる．いま情報源のエントロピー H は定数なので，冗長度を小さくすることと平均符号語長を小さくすることは等価である．本提案アルゴリズムでは式 (7) に基づいてコストを定義する．図 3 を例にコストの計算方法について説明する．まず，初期ノードからノード W までに既に情報

源シンボルに割り当てた区間のコスト $g(W)$ を

$$g(W) = -\sum_{i=1}^{t} p_i \log_2 q(c_i) \tag{8}$$

とする．ここで t は算術符号において情報源シンボルに割り当てた区間の数を指す．また，p_i は i 番目の情報源シンボルの生起確率，$q(c_i)$ は i 番目の情報源シンボルに割り当てた区間の符号化確率である．さらに，区間の候補 A_W の左から順番に残りの区間を選択したとして，ノード W から目標ノードまでの追加コストの推定値 $h(W)$ を

$$h(W) = -\sum_{i=t+1}^{n} p_i \log_2 q(a_{i-t}) \tag{9}$$

とする．$q(a_{i-t})$ は i 番目の情報源シンボルに仮に割り当てる $i-t$ 番目の区間の候補の符号化確率である．ノード W における初期ノードから目標ノードまでのコストの推定値 $f(W)$ を

$$f(W) = g(W) + h(W) = -\sum_{i=1}^{t} p_i \log_2 q(c_i) - \sum_{i=t+1}^{n} p_i \log_2 q(a_{i-t}) \tag{10}$$

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区間のコスト $g(W)$ と区間の候補のコストの推定値 $h(W)$ を足したものである．先に述べたようにコストは平均符号語長を表しているとみなせる．

ここまでで，ノードの生成と消去，コストの計算方法を説明した．ここで，生成されて消去されずに残ったノードを OPEN に入れる．そして，OPEN から最小コストのノードを取り出して子ノードを生成し，コストの計算を行う．再び残ったノードを OPEN に入れ，コストが最も小さいノードを取り出し探索を進める．以上の探索を繰り返し，最初に n 個の情報源シンボルに n 個の区間を割り当てたとき，最適な区間分割が出力され，アルゴリズムは終了する．

図 5 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．

3.3 算術符号における区間表現の検討

算術符号として取り扱える区間の候補 A について検討する．図 4 のように示した区間の状態は特別な記号として (0), (1) を用いて 0, 1 の文字列で表現することができる．精度 2 の算術符号の区間の候補と文字列の対応を図 6 に示す．

まず，一番区間の大きい $[0,4)$ の全区間を $(0)\lambda$ と表している．二番目に大きい $[0,3)$ の区間を $(1)\lambda$ と表している．ここで λ は空列を表す．次に大きい $[0,2),[2,4)$ の区間は $[0,2)$ を 0 で，$[2,4)$ を 1 で符号化する．以降においても，$[0,3)$ の区間が存在しない区間では $[0,2)$ と $[2,4)$ の区間の組み合わせで表現でき，0, 1 で符号化できる．これらの区間を表す符号語たちには (0) という記号を用いることとする．これは $(0)\lambda$ で表される $[0,4)$ の区間をか

step1.	必要な算術符号の区間の個数 n（情報源シンボルの数）と各情報源シンボルの生起確率 $(p_1, p_2, \cdots, p_n)$ を入力する．
step2.	$C_S = \phi$, $A_S = (a_1, a_2, a_3, \cdots)$ を図 4 のようにおき，$g(S) = 0, f(S) = h(S)(= -\sum_{i=1}^{n} p_i \log_2 q(a_i))$ であるようなノード S を OPEN リストの先頭に入れる．
step3.	OPEN リストの先頭に入っているノード W(始めは $W = S$) を取り出す．もし C_W のサイズが n と等しいならば，最適な区間分割として $C_W = (c_1, c_2, \cdots, c_n)$ が出力されて終了する．
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 $A_W = \phi$ 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．残りの子ノードのコストを計算する．
step5.	残りのノードを OPEN リストに入れ，コストの小さい順に並びかえて step3 に移る．

図 5　提案アルゴリズムの基本構造

け合わせていると考えることもできる．一方で，$[0, 3)$ の区間が組み合わさっている区間においては 0, 1 だけでの符号化はできない．ここで $(1)\lambda$ と表される $[0, 3)$ の区間がかけ合わされていると考えると (1) という記号を用いることができる．図 6 を見ると作成した符号語は $(0), (1)$ の記号でそれぞれ昇順になっている．ここで精度 2 の算術符号における状態は $(0), (1)$ のふたつで表される．よって，算術符号における区間の候補は漏れなく降順に並んでいる．

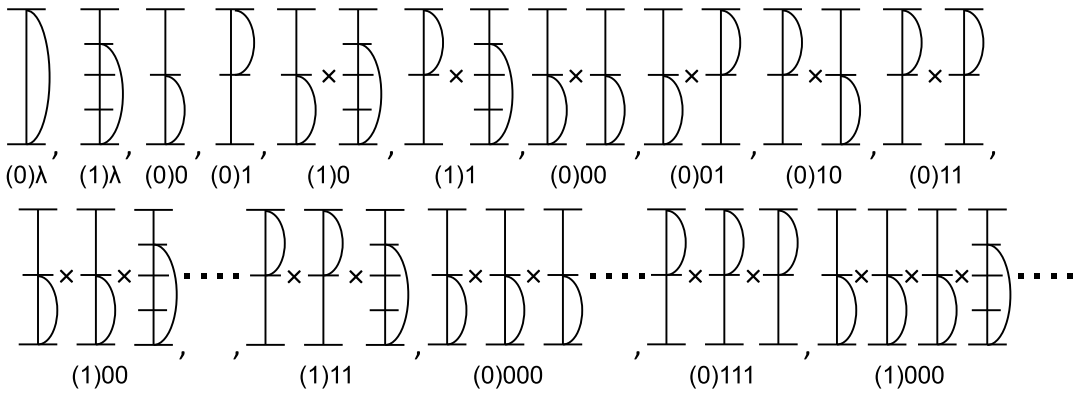

図 6　区間の候補 A と作成した符号語の対応

3.4 提案アルゴリズムの最適性の証明

本提案アルゴリズムでは最初に情報源シンボル n 個の区間を割り当てた区間分割が入力待ちの各状態において最適となる．ここで，提案アルゴリズムで定まる最終ノードのコストの値が最小となることについて証明する．まず，どのノードを見ても子ノードのコストは親ノードのコスト以上の値となることを示す．図 3 を見て，ノード W から子ノード X を作るとき，

$$\begin{aligned} f(W) &= -\sum_{i=1}^{t} p_i \log_2 q(c_i) - \sum_{i=t+1}^{N} p_i \log_2 q(a_{i-t}) \\ &\leqq \left[-\sum_{i=1}^{t} \{p_i \log_2 q\,(c_i)\} - p_{t+1} \log_2 q(a_1) \right] - \sum_{i=t+2}^{N} p_i \log_2 q(a'_{i-t-1}) \\ &= f(X) \end{aligned}$$

となる．同様にノード W から子ノード Y を作るとき，

$$\begin{aligned} f(W) &= -\sum_{i=1}^{t} p_i \log_2 q(c_i) - \sum_{i=t+1}^{N} p_i \log_2 q(a_{i-t}) \\ &\leqq -\sum_{i=1}^{t} p_i \log_2 q\,(c_i) - \sum_{i=t+1}^{N} p_i \log_2 q(a_{i-t+1}) \\ &= f(Y) \end{aligned}$$

となる．したがってどの状態でも子ノードのコストは親ノードのコスト以上の値になる．提案アルゴリズムが止まるときは，$n-1$ 個の算術符号としての区間が選択されているノード W でコストが一番小さい．このときノード W でコストが一番小さいため，W からの子ノードである区間の候補 a_1 を選んだ子ノード X のコストと等しい．子ノードのコストは親ノードのコスト以上の値となるから，そのときできたノード X のコストより低いコストをもつノードは今後現れない．

以上の説明より，まず子ノードのコストは親ノードのコスト以上の値となる．次に提案アルゴリズムが止まるときは，$n-1$ 個の情報源シンボルに区間を割り当てたノードのコストが一番小さく，その子ノードである n 個の情報源シンボルに区間を割り当てたノードのコストと等しくなる．先に述べた子ノードのコストは親ノードのコスト以上の値となることから，提案アルゴリズムが止まり，最初に情報源シンボル n 個すべてに算術符号の区間を割り当てたとき，コストが最小となる．ここで，コストの値からエントロピーの値を引くとダイバージェンスとなるため，最小コストで最小のダイバージェンスであると言える．また，算術符号の精度 2 では提案アルゴリズムを用いて $[0,4)$，$[0,3)$ の区間で探索を始める．つまり，提案アルゴリズム

が止まり，最初に見つかる区間分割が入力待ちの状態毎に最小のダイバージェンスとなり，最小の冗長度であり最小の平均符号語長である．

しかし，定常確率を考慮したときに，入力待ちの各状態に対する最適性を犠牲にしても符号の平均符号語長が小さくなる場合がある．次章では，最適な AIFV 符号の探索方法を参考にそのような場合の探索方法を考察する．

4 AIFV 符号との関係

4.1 AIFV 符号の概要

AIFV 符号 [5] とは複数の符号木を用い，より符号化レートを小さくする符号化法である．これは情報源シンボルを各符号木の葉だけではなく内部ノードにも割り当てることで，より小さい符号化レートを達成している．2 元 AIFV 符号の定義と符号化，復号化の手続きは以下のとおりである．

定義（2 元 AIFV 符号）

1. 2 元 AIFV 符号は 2 個の符号木からなる．それらを T_0, T_1 と表す．
2. 1 つだけの子をもつ不完全な内部ノードは主ノードと副ノードに分けられる．主ノードの子は副ノードでなくてはならない．主ノードはその孫へラベル 00 で繋がっている．
3. T_1 のルートノードは 2 つの子をもち，00 でつながる孫をもたない．T_1 のルートノードからラベル 0 で繋がる子ノードは副ノードである．
4. 情報源シンボルは符号木の葉または主ノードに割り当てられる．

手続き（2 元 AIFV 符号による符号化）

1. 最初の情報源シンボルは T_0 によって符号化する．
2. 情報源シンボルが葉（または主ノード）によって符号化されたら，次のシンボルの符号化には T_0（または T_1）を用いる．

手続き（2 元 AIFV 符号による復号化）

1. 符号化と同様に最初の符号語系列は T_0 によって復号化する．
2. 復号に用いる符号木のルートから観測した符号シンボルの順番に枝をたどり，次のの符号シンボルが割り当てられた枝をたどる方法がない場合は，今指しているノードに割り当てられた情報源シンボルを復号する．今指しているノードに情報源シンボルが割り当てられていない場合は親ノードに割り当てられた情報源シンボルを復号する．
3. 符号化と同様に符号語系列が葉（または主ノード）によって復号化されたら，次の符号語系列の復号化には T_0（または T_1）を用いる．

また，最適な 2 元 AIFV 符号の探索アルゴリズムとして文献 [6] が提案されている．AIFV 符号において，平均符号語長を小さくするために 2 つの方法が挙げられる．ひとつは，T_0 の定常確率を大きくし，T_1 の定常確率を小さくする．これは，T_0 より T_1 の方が平均符号語長が大きくなるからである．ふたつめは，T_0, T_1 それぞれにおいて平均符号語長を小さくする．これは，可能な限り主ノードを使用することで小さくなる．しかし，主ノードを使用すると T_1 に移るため，T_1 の定常確率が大きくなる．つまり，ひとつめとふたつめはトレードオフである．ここで，文献 [6] ではロスという不確定な値を用いてコストを計算し，ロスの値を確定するためにアルゴリズムを繰り返し行っている．

4.2 提案アルゴリズムと AIFV 符号との比較

本研究で提案した各状態における最適な算術符号の探索と最適な AIFV 符号の探索を比較し，定常確率を考慮した探索方法を考察する．

まず，提案アルゴリズムと AIFV 符号の定義を比較する．2 元 AIFV 符号は 2 個の符号木からなるが，提案アルゴリズムにおいて精度 2 では $[0,4)$, $[0,3)$ のそれぞれの区間で探索を始め，完成する木が T_0, T_1 に対応する．これは，T_1 のルートノードが 00 でつながる孫をもたないことと，$[0,3)$ の区間の候補において (0)11 で表せる $[3,4)$ の区間が消去されることからもわかる．また，AIFV 符号は情報源シンボルを各符号木の葉と内部ノードに割り当てているが，図 6 で作成した符号語において (0) が葉に，(1) が内部ノードに対応している．これは，主ノードがその孫へラベル 00 でつながっていることと，$[0,3)$ の区間を選択すると重なる区間が候補から消去されるため，残りの区間を埋めるように (0)11 で表せる $[3,4)$ の区間が必ず選ばれることからもわかる．

次に，本提案アルゴリズムと文献 [6] で提案されている最適な 2 元 AIFV 符号を比較する．最適な AIFV 符号では不確定なロスの値を確定するために，探索アルゴリズムを繰り返し行っている．ロスの値は主ノードを使用したときに，コストに加算される値である．不確定なロスの値を使用しているためコストも不確定となる．ロスの値を以下のように定義している．それぞれの符号木を T_0, T_1 とし，平均符号語長をそれぞれ l_0, l_1, 状態遷移確率を $Q(T_1|T_0) = q_0$, $Q(T_0|T_1) = q_1$ と表記する．このときロスの値 s を

$$s = \frac{l_1 - l_0}{q_0 + q_1} \tag{11}$$

と表し，値が確定するまでアルゴリズムを繰り返し行う．このように，算術符号においても状態遷移確率を導入したロスという値をコストとして用いることで定常確率を考慮できると考えられる．また，ロスの値は更新しながら繰り返す必要があるが，そうではないアルゴリズムが存在するかどうか今後研究を進めていく．

ここで，図 7, 8 で文献 [6] で使用された例を用いて最適な 2 元 AIFV 符号の結果と本提案

アルゴリズムを用いた最適な区間分割の結果を示す．また，提案アルゴリズムによって得られた結果から符号木を作成した．4 つの情報源シンボルをそれぞれ 0.45, 0.3, 0.2, 0.05 とする．図 7, 8 より 0, 1 の符号の逆転を除けば，木の形は等しくなった．図 7 に示した最適な 2 元 AIFV 符号の平均符号語長は T_0 と T_1 の定常確率と平均符号語長がそれぞれ $Q(T_0) = \frac{5}{9}$, $Q(T_1) = \frac{4}{9}$, $l_{T_0} = 1.65$, $l_{T_1} = 1.85$ であるため，$L_{AIFV} = 1.738$ となる．図 8 に示した各状態における最適な算術符号の平均符号語長は最適な 2 元 AIFV 符号と同様の定常確率と各平均符号語長となるため，$L_{AC} = 1.738$ となる．ちなみに，情報源のエントロピーは約 1.720 である．以上のように，両者の平均符号語長が同じになる場合がある．

一方で，両者の平均符号語長が異なる例も存在する．そのような例の符号木を図 9, 10 に示す．10 個の情報源シンボルをそれぞれ 0.45, 0.25, 0.15, 0.13, 0.0075, 0.005, 0.0025, 0.0025, 0.002, 0.0005 とする．本提案アルゴリズムでは $[0, 4)$, $[0, 3)$ それぞれの状態における最小な平均符号語長は $l_0 = 1.931$, $l_1 = 2.061$ となる．定常確率はそれぞれ $Q(0) = \frac{281}{560}$, $Q(1) = \frac{279}{560}$ であるため，算術符号の平均符号語長は $L_{AC} = 1.996$ となる．最適な 2 元 AIFV 符号の平均符号語長は T_0 と T_1 の定常確率と平均符号語長がそれぞれ $Q(T_0) = \frac{1721}{2000}$, $Q(T_1) = \frac{279}{2000}$, $l_{T_0} = 1.931$, $l_{T_1} = 2.381$ であるため，$L_{AIFV} = 1.994$ となる．ちなみに，情報源のエントロピーは 1.969 である．本提案アルゴリズムと最適な AIFV 符号の各状態の平均符号語長 l_1, l_{T_1} を比較すると本提案アルゴリズムの方が小さい．しかし，定常確率を考慮したときに各状態の平均符号語長を犠牲にしても，最適な AIFV 符号の方が平均符号語長が小さくなる．先に述べたように，定常確率を考慮した最適な算術符号の探索方法を今後探っていく．

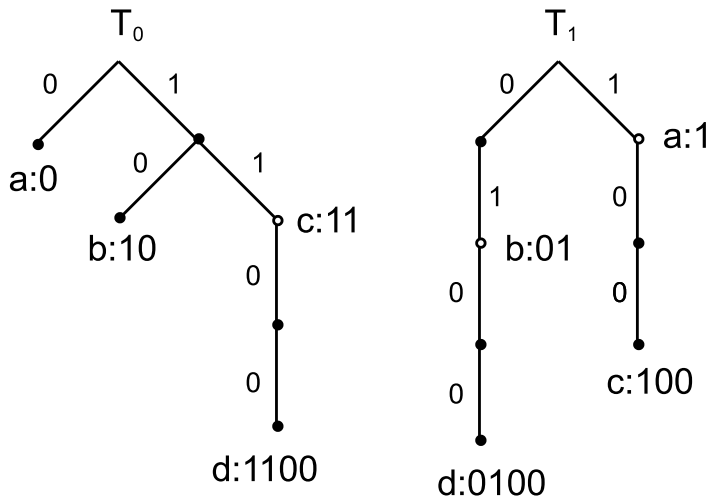

図 7　最適な 2 元 AIFV 符号の例 1

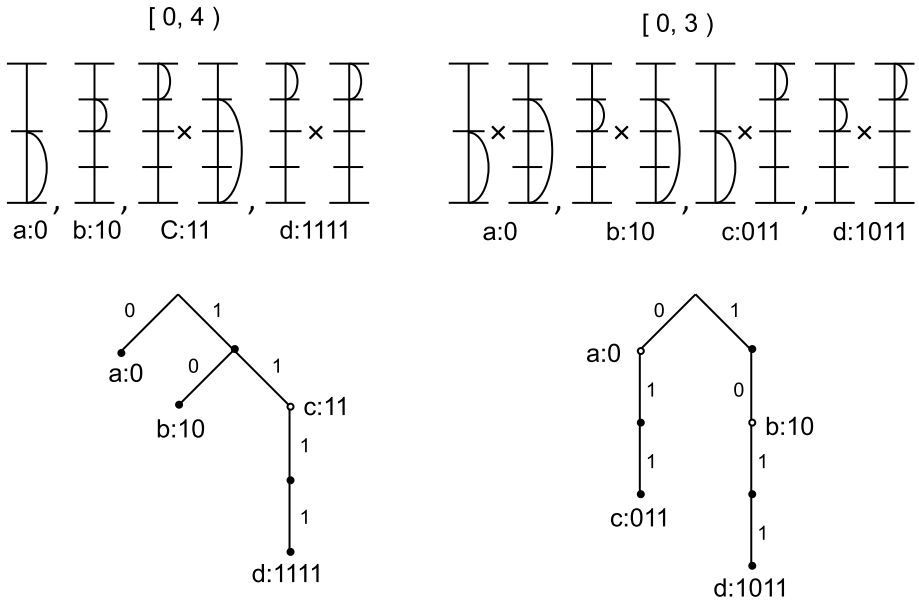

図 8　最適な区間分割の例 1 の符号木

5　まとめ

本研究では有限精度で順序保存性のない算術符号における平均符号語長が最小となる最適な算術符号を模索した．まず，算術符号の状態遷移より冗長度を求めた．次に，A^* アルゴリズムを用いて入力待ちの状態のダイバージェンスが最小になる分割点を探るための探索アルゴリズムを提案した．子ノードのコストが親ノードのコスト以上になることから，最小のコストが選ばれることが確認できた．また，ダイバージェンスが最小になるとき，入力待ちの各状態で冗長度が最小になることと平均符号語長が最小になることは同義である．さらに，定常分布を考慮するために本提案アルゴリズムと AIFV 符号を比較した．最適な AIFV 符号の探索方法におけるロスの値を本提案アルゴリズムで適用することで定常分布を考慮できるか考えた．また，ロスの値を用いずに定常分布を考慮できるか今後研究を進めていく．

謝辞

本研究を進めるにあたり，多くの助言，細かな指導を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．

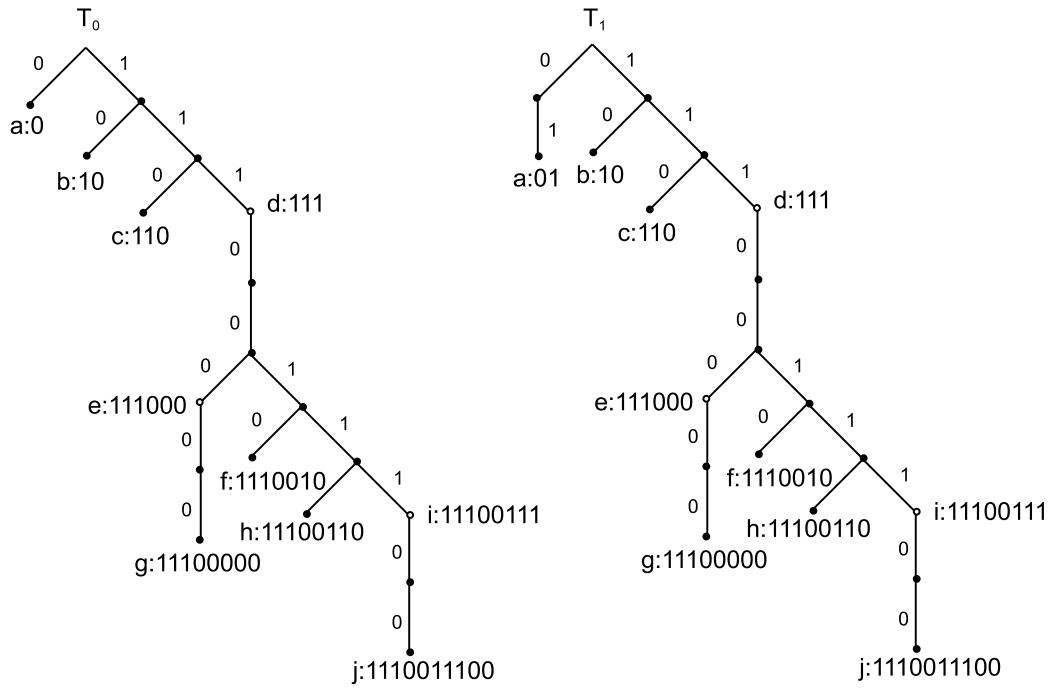

図 9　最適な 2 元 AIFV 符号の例 2

参考文献

[1] 情報理論とその応用学会編，情報源符号化—無歪みデータ圧縮，培風館，1998.

[2] 西新幹彦，「算術符号の演算精度と状態数と遅延に関する考察」，第 8 回シャノン理論ワークショップ (STW13)，pp.35–40，2013 年 10 月.

[3] Raymond W. Yeung, A First Course in Information Theory, Springer, pp.54–59, 2002.

[4] E. Rich and K. knight, Artificial Intelligence, New York: McGraw-Hill, 1993.

[5] Hirosuke Yamamoto, Masato Tsuchihashi, and Junya Honda, “Almost Instantaneous Fixed-to-Variable Length Codes,” IEEE Transactions on Information Theory, vol.61, no.12, pp.6432–6443, December 2015.

[6] 山川誠史，西新幹彦，「最適な 2 元 AIFV 符号の幅優先探索アルゴリズム」，信学技報，vol.117，no.120，IT2017-30，pp.79–83，2017 年 7 月.

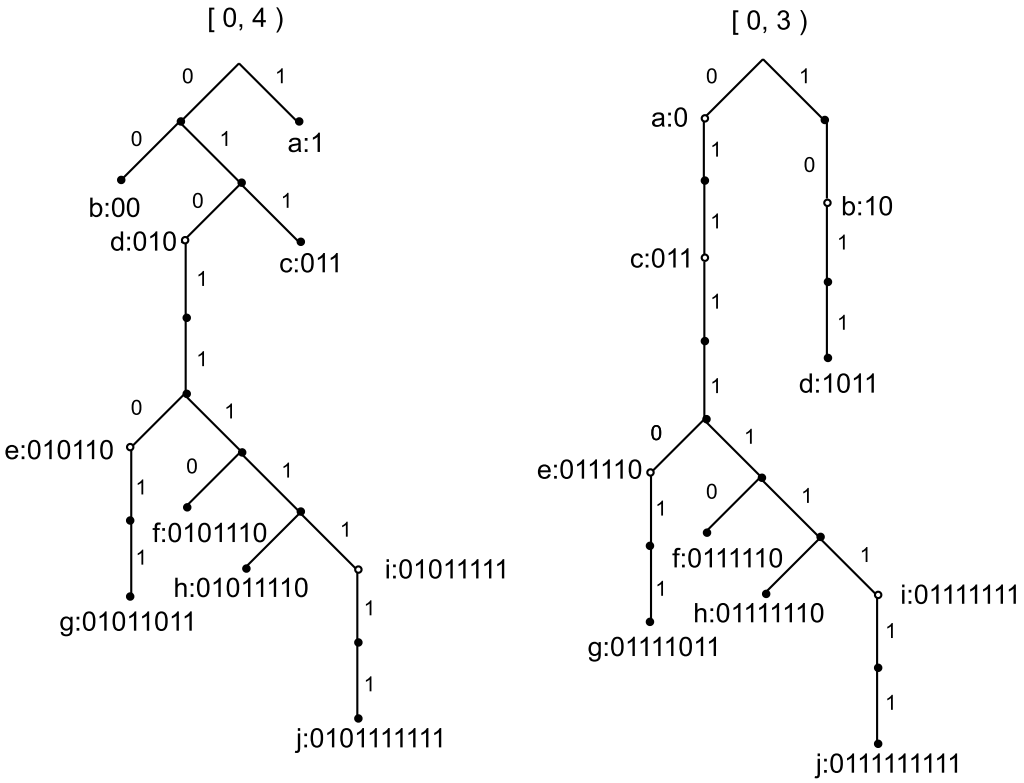

図 10　最適な区間分割の例 2 の符号木

付録 A　提案アルゴリズムのソースコード

```
#define _CRT_NONSTDC_NO_DEPRECATE
#define _CRT_SECURE_NO_WARNINGS

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

#define MAX_CODEWORDLEN 32

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

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("\n コスト:%f\n", p->cost);
        return;
}

/*****************************次の文字列を生成*********************************************************************/
char *nextword(char *can, int interval_name)
{
        int i;
NEXT:
        for (i = 0; can[i] == '1'; i++) {
                ; //何も入れないのもあり
        }
        if (can[i] == '\0') {
                if (i + 1 >= MAX_CODEWORDLEN) {
                        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';
                }
                if (interval_name == 1 && strlen(can) >= 3 && can[1] == '1' && can[2] == '1') {
                        goto NEXT;
                }
                if (interval_name == 1 && strlen(can) == 2 && can[1] == '1') {
                        goto NEXT;
                }
                return(can);
        }
        for (; can[i] != '\0'; i++) { /* 終端を探す*/
                ;
        }
        for (i--; can[i] == '1'; i--) {
```

```
                can[i] = '0';
        }
        can[i] = '1';
        if (interval_name == 1 && strlen(can) >= 3 && can[1] == '1' && can[2] == '1') {
                goto NEXT;
        }
        if (interval_name == 1 && strlen(can) == 2 && can[1] == '1') {
                goto NEXT;
        }
        return(can);
}

int
isprefix(char *word1, char *word2) {//word2 が word1 の子か
        int len1, len2, i;
        len1 = strlen(word1);
        len2 = strlen(word2);
        if (len1 > len2) {
                return(0);
        }
        for (i = 1; i < len1; i++) {
                if (word1[i] != word2[i]) {
                        return(0);
                }
        }
        return(1);
}

int
isoverlapping(char *word1, char *word2) {//word1 と word2 が重なっているか
        char interval_0[MAX_CODEWORDLEN];
        char interval_10[MAX_CODEWORDLEN];
        strcpy(interval_0, word1);
        strcpy(interval_10, word1);
        strcat(interval_0, "0");
        strcat(interval_10, "10");
        if (isprefix(interval_0, word2) || isprefix(interval_10, word2) || isprefix(word2, interval_10)) {
                return(1);
        }
        else {
                return(0);
        }
}

void
replace_codeword(struct state state, int interval_name)//区間の候補（符号語）の補充
{
        int i, j;
        char can[MAX_CODEWORDLEN];

        for (i = alphabet_size - 1; state.codeword[i] == NULL; i--) {
                ;
        }
        if (i == alphabet_size - 1) {
                return;
        }
        strcpy(can, state.codeword[i]);
        for (i++; i < alphabet_size; i++) {
        NEXT_CANDIDATE:
                switch (interval_name) {
                case 0:
                        nextword(can, 0);
                        break;
                case 1:
                        nextword(can, 1);
                        break;
                default:
                        printf("error in %d\n", __LINE__);
                        exit(1);
```

```
                        break;
                }
                for (j = 0; j < state.fixed; j++) {//選択済みの区間と作成した区間の比較
                        switch (state.codeword[j][0]) {
                        case '1':
                                if (isoverlapping(state.codeword[j], can)) {
                                        goto NEXT_CANDIDATE;
                                }
                                break;
                        case '0':
                                if (isprefix(state.codeword[j], can)) {
                                        goto NEXT_CANDIDATE;
                                }
                                break;
                        default:
                                printf("error in %d\n", __LINE__);
                                exit(1);
                                break;
                        }
                }
                state.codeword[i] = strdup(can);
                if (state.codeword[i] == NULL) {
                        exit(1);
                }
        }
        return;
}

void
confirm_descendant_masternode(struct state *state) {//全ての [0,3] が [3,4) の区間を持てるか確認
        int i, j, l, m;
        l = 0;//[0,4) の数
        m = 0;//[3,4) を持つ [0,3) の数
        for (i = 0; i < state->fixed; i++) {
                if (state->codeword[i][0] == '0') {
                        l++;
                        continue;
                }
                if (i == alphabet_size - 1) {
                        state->fixed += alphabet_size + 1;
                        return;
                }
                for (j = i + 1; j < state->fixed; j++) {
                        if (isprefix(state->codeword[i], state->codeword[j])) {
                                m++;
                                break;
                        }
                }
        }
        if (alphabet_size - state->fixed < state->fixed - m - l) {//区間の候補の数 < [3,4) がない [0,3) の数
                state->fixed += alphabet_size + 1;
                return;
        }
        return;
}
/*****************************[0,4) で選択しない場合*********************************************************************/
struct state
        removecandidate_t0(struct state *pstate)
{
        struct state nstate;
        int i;
        if (pstate == NULL) {
                printf("[%d]", __LINE__);
                exit(1);
        }
        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);
                }
        }
        nstate.codeword[alphabet_size - 1] = NULL;
        replace_codeword(nstate, 0);

        return(nstate);
}
/*****************************************************************************************/

/*****************************[0,4) で選択した場合*****************************************************************/
struct state
        selected_interval_t0(struct state *pstate)
{
        struct state nstate;
        int i, j;
        double len1, len2;
        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;
        for (i = nstate.fixed; i < alphabet_size; i++) {
                switch (nstate.codeword[nstate.fixed - 1][0]) {
                case '0':
                        if (!isprefix(nstate.codeword[nstate.fixed - 1], pstate->codeword[i])) {
                                nstate.codeword[j] = strdup(pstate->codeword[i]);
                                if (nstate.codeword[j] == NULL) {
                                        printf("error in %d\n", __LINE__);
                                        exit(1);
                                }
                        }
                        else {
                                continue;
                        }
                        j++;
                        break;
                case '1':
                        if (!isoverlapping(nstate.codeword[nstate.fixed - 1], pstate->codeword[i])) {
                                nstate.codeword[j] = strdup(pstate->codeword[i]);
                                if (nstate.codeword[j] == NULL) {
                                        printf("error in %d\n", __LINE__);
                                        exit(1);
                                }
```

```
                        }
                        else {
                                continue;
                        }
                        j++;
                        break;
                default:
                        printf("error in %d\n", __LINE__);
                        exit(1);
                        break;
                }
        }

        if (nstate.fixed < alphabet_size) {//[0,4) の区間を超えないかの確認
                len2 = 0;
                for (i = 0; i < nstate.fixed; i++) {
                        len1 = 1;
                        if (nstate.codeword[i][0] == '1') {
                                len1 *= 0.75;
                        }
                        if (strlen(nstate.codeword[i]) > 1) {
                                len1 *= pow(0.5, strlen(pstate->codeword[i]) - 1);
                        }
                        len2 += len1;
                }
                if (len2 > 1) {
                        nstate.fixed += alphabet_size + 1;
                        return(nstate);
                }
                if (len2 >= 1 && nstate.fixed < alphabet_size) {
                        nstate.fixed += alphabet_size + 1;
                        return(nstate);
                }
        }
        replace_codeword(nstate, 0);
        confirm_descendant_masternode(&nstate);

        return(nstate);
}
/*********************************************************************************************/

/*****************************[0,3) で選択しない場合*****************************************************************/
struct state
        removecandidate_t1(struct state *pstate)
{
        struct state nstate;
        int i;
        if (pstate == NULL) {
                printf("[%d]", __LINE__);
                exit(1);
        }
        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);
```

```
                }
        }

        nstate.codeword[alphabet_size - 1] = NULL;
        replace_codeword(nstate, 1);

        return(nstate);
}
/*****************************************************************************************/

/*****************************[0,3)で選択した場合*****************************************************************/
struct state
        selected_interval_t1(struct state *pstate)
{
        struct state nstate;
        int i, j;
        double len1, len2;
        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;
        for (i = nstate.fixed; i < alphabet_size; i++) {
                switch (nstate.codeword[nstate.fixed - 1][0]) {
                case '0':
                        if (!isprefix(nstate.codeword[nstate.fixed - 1], pstate->codeword[i])) {
                                nstate.codeword[j] = strdup(pstate->codeword[i]);
                                if (nstate.codeword[j] == NULL) {
                                        printf("error in %d\n", __LINE__);
                                        exit(1);
                                }
                        }
                        else {
                                continue;
                        }
                        j++;
                        break;
                case '1':
                        if (!isoverlapping(nstate.codeword[nstate.fixed - 1], pstate->codeword[i])) {
                                nstate.codeword[j] = strdup(pstate->codeword[i]);
                                if (nstate.codeword[j] == NULL) {
                                        printf("error in %d\n", __LINE__);
                                        exit(1);
                                }
                        }
                        else {
                                continue;
                        }
                        j++;
                        break;
                default:
                        printf("error in %d\n", __LINE__);
                        exit(1);
                        break;
                }
        }
```

```
        if (nstate.fixed < alphabet_size) {//[0,3) の区間を超えないかの確認
                len2 = 0;
                for (i = 0; i < nstate.fixed; i++) {
                        len1 = 1;
                        if (nstate.codeword[i][0] == '1') {
                                len1 *= 0.75;
                        }
                        if (strlen(nstate.codeword[i]) > 1) {
                                len1 *= pow(0.5, strlen(pstate->codeword[i]) - 1);
                        }
                        len2 += len1;
                }
                if (len2 > 0.75) {
                        nstate.fixed += alphabet_size + 1;
                        return(nstate);
                }
                if (len2 >= 0.75 && nstate.fixed < alphabet_size) {
                        nstate.fixed += alphabet_size + 1;
                        return(nstate);
                }
        }
        replace_codeword(nstate, 1);
        confirm_descendant_masternode(&nstate);

        return(nstate);
}
/*********************************************************************************************/

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

double c_cost(struct state *pstate) {
        int i;
        double cost, len; //コスト, 区間の分布

        cost = 0;
        for (i = 0; i < alphabet_size; i++) {
                len = 1;
                if (pstate->codeword[i][0] == '1') {
                        len *= 0.75;
                }
                if (strlen(pstate->codeword[i]) > 1) {
                        len *= pow(0.5, strlen(pstate->codeword[i]) - 1);
                }
                cost -= (prob[i] * log2(len));
        }
        return(cost);
}

/*********************************************************************************************/
void
initlist_t0(struct state *p)//[0,4) 用の初期設定
{
        int i;
        char can[MAX_CODEWORDLEN];

        p->fixed = 0;//一番最初の state に入力していく↓
        p->codeword = (char **)calloc(alphabet_size, sizeof(char *));
        if (p->codeword == NULL) {
                exit(EXIT_FAILURE);
        }
        can[0] = '\0';
        p->codeword[0] = strdup(can);
        for (i = 0; i < alphabet_size; i++) {
                nextword(can, 0);
                p->codeword[i] = strdup(can);
                if (p->codeword[i] == NULL) {
                        printf("error in %d\n", __LINE__);
                        exit(1);
                }
```

```
        }
        return;
}

void
initlist_t1(struct state *p)//[0,3) 用の初期設定
{
        int i;
        char can[MAX_CODEWORDLEN];

        p->fixed = 0;//一番最初の state に入力していく↓
        p->codeword = (char **)calloc(alphabet_size, sizeof(char *));
        if (p->codeword == NULL) {
                exit(EXIT_FAILURE);
        }
        can[0] = '\0';
        nextword(can, 1);
        for (i = 0; i < alphabet_size; i++) {
                nextword(can, 1);
                p->codeword[i] = strdup(can);
                if (p->codeword[i] == NULL) {
                        printf("error in %d\n", __LINE__);
                        exit(1);
                }
        }
        return;
}

#define LISTBOUND 1000000
struct state list[LISTBOUND];
struct state state, state1, state2_t0, state2_t1;//state1 は選択しない子,state2 は選択する子

void
state_free(struct state state)
{
        int i;

        for (i = 0; i < alphabet_size; i++) {
                free(state.codeword[i]);
        }
        free(state.codeword);
        return;
}

void
find_t0()
{
        int num_state;//state の数
        int i;
        initlist_t0(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_t0(&state);
                if (state1.fixed <= alphabet_size) {
                        if (num_state >= LISTBOUND) {
                                printf("error in %d\n", __LINE__);
                                exit(1);
                        }
                        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++;
                }
                else {
                        state_free(state1);
                }
                state2_t0 = selected_interval_t0(&state);
                if (state2_t0.fixed == alphabet_size) {
                        state2_t0.cost = c_cost(&state2_t0);
                        return;
                }
                if (state2_t0.fixed < alphabet_size) {
                        if (num_state >= LISTBOUND) {
                                printf("error in %d\n", __LINE__);
                                exit(1);
                        }
                        state2_t0.cost = c_cost(&state2_t0);
                        for (i = num_state; i > 0; i--) {
                                if (state2_t0.cost <= list[i - 1].cost) {
                                        break;
                                }
                                list[i] = list[i - 1];
                        }
                        list[i] = state2_t0;
                        num_state++;
                }
                else {
                        state_free(state2_t0);
                }
                state_free(state);
        }
}

void
find_t1()
{
        int num_state;//state の数
        int i;
        initlist_t1(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_t1(&state);
                if (state1.fixed <= alphabet_size) {
                        if (num_state >= LISTBOUND) {
                                printf("error in %d\n", __LINE__);
                                exit(1);
                        }
                        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++;
                }
                else {
                        state_free(state1);
                }
                state2_t1 = selected_interval_t1(&state);
                if (state2_t1.fixed == alphabet_size) {
                        state2_t1.cost = c_cost(&state2_t1);
                        return;
                }
                if (state2_t1.fixed < alphabet_size) {
                        if (num_state >= LISTBOUND) {
```

```
                                printf("error in %d\n", __LINE__);
                                exit(1);
                        }
                        state2_t1.cost = c_cost(&state2_t1);
                        for (i = num_state; i > 0; i--) {
                                if (state2_t1.cost <= list[i - 1].cost) {
                                        break;
                                }
                                list[i] = list[i - 1];
                        }
                        list[i] = state2_t1;
                        num_state++;
                }
                else {
                        state_free(state2_t1);
                }
                state_free(state);
        }
}
main() {
        int i;
        printf("アルファベットサイズ=");
        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);
                        }
                }
        }
        for (i = 0; i < alphabet_size; i++) {
                printf("確率 %d=%f\n", i, prob[i]);
        }
        find_t0();
        find_t1();
        printf("区間 [0,4)\n");
        printstate(&state2_t0);
        printf("区間 [0,3)\n");
        printstate(&state2_t1);
        return(0);
}
```