信州大学
大学院総合理工学研究科

修士論文

A^* アルゴリズムを用いた順序保存性のない最適な算術符号の探索

指導教員　西新 幹彦 准教授

専攻　工学専攻
分野　電子情報システム工学分野
学籍番号　18W2027A
氏名　打田 尚大

2020 年 02 月 21 日

目次

1　はじめに

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，効率の良い通信を行うために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 [1] がある．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．この精度を大きくすると遅延が大きくなることが知られている [2]．そこで，本研究では遅延を抑えるために精度が 2 ビットの算術符号を考える．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，本研究では算術符号の順序保存性という制約を緩和し，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．したがって，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，符号アルファベットを 2 元とし，情報源アルファベットを再帰的に繰り返し 2 分割することによって，情報源シンボルの順序と符号語を表現する．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，経路探索アルゴリズムの一種である A^* アルゴリズム [3] を応用することで，最適な算術符号を探索することを提案する．

本研究で取り扱う算術符号の精度は 2 ビットである．すると，本研究の算術符号は AIFV 符号 [4] と等価となる．AIFV 符号は複数の符号木を用いて，ハフマン符号よりも小さい符号化レートを達成できる符号として知られている．最適な AIFV 符号は反復アルゴリズムで構成できるが [5]，その内部アルゴリズムとして動的計画法を用いる方法 [6] や A^* アルゴリズムを用いる方法 [7] が提案されている．本論文の提案は定常分布を考慮することで，2 つの符号木を同時に最適化している．そのため，本提案アルゴリズムは反復を必要としない．

本論文では 2 章で本研究で扱う算術符号について説明し，3 章で順序保存性のない最適な算術符号の探索アルゴリズムを提案する．4 章ではその最適性の証明，提案アルゴリズムの計算量の見積もり，順序保存性を緩和したことによる平均符号語長の改善の確認を行う．5 章では算術符号と AIFV 符号の関係を，6 章では計算量の削減について考察する．最後に 7 章で本論文をまとめる．

2　算術符号

算術符号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．本章では，文献 [2] に基づいて算術符号の原理を述べた後，本研究で扱う算術符号の動作や状態遷移について説明する．

2.1 原理

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 $\boldsymbol{x}$ とその語頭 $\boldsymbol{x'}$ に対して，$\boldsymbol{x}$ に対応する区間は $\boldsymbol{x'}$ 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 $\boldsymbol{x} \in \mathcal{X}^*$ に対して $I_s(\boldsymbol{x}) \subset I_c(\boldsymbol{y})$ となるような符号語系列 $\boldsymbol{y} \in \mathcal{Y}^*$ を求めることである．一般にそのような $\boldsymbol{y}$ 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 $\boldsymbol{x}$ の先頭から順に文字を見ることによって，$I_s(\boldsymbol{x})$ 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 $\boldsymbol{y}$ 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 $\boldsymbol{y}$ に対して $I_c(\boldsymbol{y}) \subset I_s(\boldsymbol{x})$ となるような情報源系列 $\boldsymbol{x}$ を求めることである．符号器同様，復号器も符号語系列 $\boldsymbol{y}$ の先頭から順に文字を見ることによって，情報源系列 $\boldsymbol{x}$ 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．

2.2 精度と冗長度

算術符号を現実的な場面で使用する際の，区間の精度について説明する．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 w ビットであれば全区間は 2^w 個に分割され，$2^w - 1$ 個の分割点をもつ．したがって，全区間を $[0, 2^w)$ と定義した場合，分割点は整数値をとる．

ここで算術符号の冗長度について考える．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．このとき式 (1) を満たす写像 I_s は一般には存在しない．そこで，情報源の確率と符号化確率の間のダイバージェンスが小さくなるように分割点を選ぶことが重要になる．かつてはアルファベットサイズの大きい情報源を符号化するためには分割点は多くなければならないと考えられていた．しかし，情報源アルファベットのサイズによらず（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，演算精度が低くても符号化は可能である [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．また，精度を大きくすると遅延が大きくなることが知られている [2]．本研究では遅延を小さくするために 2 ビットの精度をもつ算術符号を考え，平均符号語長を最小化する．

2.3 状態遷移

算術符号では符号化をする過程で区間が複数回分割される．本研究では情報源アルファベットを 2 分するグループを段階的に作成する．まず，情報源アルファベット $\mathcal{X}$ を互いに素な任意の集合 $\mathcal{X}_0$ と $\mathcal{X}_1$ に分割する．次に，現在の区間を確率 $P(\mathcal{X}_0)$, $P(\mathcal{X}_1)$ に基づいて 2 つの区間に分割する．さらに，その 2 つの集合から 1 つの集合 $\mathcal{X}_s$ を選ぶ．$\mathcal{X}_s$ が単一のシンボルになるまで同様に $\mathcal{X}_s$ を $\mathcal{X}_{s0}$ と $\mathcal{X}_{s1}$ に分割し，$P(\mathcal{X}_{s0}|\mathcal{X}_s)$, $P(\mathcal{X}_{s1}|\mathcal{X}_s)$ に基づいて，新しい区間の分割を繰り返し行う．このように，情報源アルファベットを 2 分するグループを繰り返し作成する際の遷移を図 1 に示す．区間の右に示したアルファベットは遷移先である．本研究では算術符号の遅延を抑えるために精度を 2 ビットに固定していることから，全区間を $[0,4)$ とする．図 1(a) の区間 $[0,4)$ は 3 つの分割点を持ち，それに応じて 3 通りの分割方法がある．同様に図 1(b), 図 1(c) の区間 $[0,3)$, $[1,4)$ は分割点を 2 つ持ち，2 通りの分割方法がある．さらに図 1(d) の区間 $[1,3)$ は分割点が 1 つだけとなり，分割方法は 1 通りだけである．このように分割点に基づいて 4 つの遷移先が決定される．符号化はこの遷移に基づいている．情報源アルファベットのグループを段階的に 2 分割し，遷移するたびに出力される符号語を図 1 の矢印の横に示す．グループが単一のシンボルになったときシンボルの符号語が決定する．ただし，単一のシンボルが区間 $[0,3), [1,3), [1,4)$ に割り当てられたとき長さ 0 の符号語が出力される．

ここで，図 1 の 4 つの遷移先は図 2 の 2 つの遷移先で十分であることに注意する．なぜなら，図 1(a) の分割点 1 で分割する方法は情報源アルファベットの集合を入れ替えることで，符号語長を変えずに分割点 3 で分割する方法にできるからである．また，図 1(d) の区間の状

態より図 1(a) の区間の状態でシンボルに割り当てられる符号語が長くなることはないため，図 1(d) の区間の状態への遷移を考える必要はない．そこで，図 1(b) の分割点は，図 1(d) の区間の状態に遷移しないように分割点 2 で分割する．同様に図 1(c) の分割点も，分割点 2 で分割する．このとき，図 2(a)，図 2(b) のような区間の状態をそれぞれ区間の状態 T_0, T_1 と呼ぶことにする．

3　最適な算術符号の探索アルゴリズム

本研究で扱う順序保存性のない精度 2 の算術符号において平均符号語長を最小にする．つまり，再帰的 2 分割の中から最適な分割を選ぶことで最小の平均符号語長を得る．しかし，あらゆる分割の中からそのまま探索することは現実的ではない．そこで，経路探索アルゴリズムの一種である A^* アルゴリズム [3] を応用して最適な算術符号の探索を提案する．

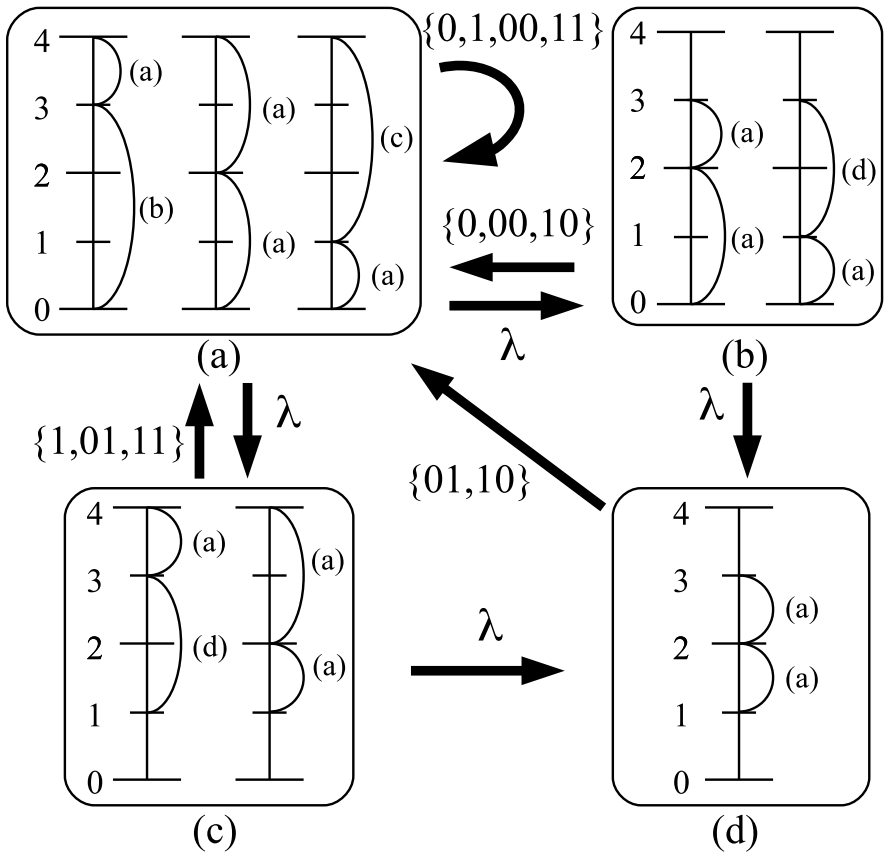

図 1　区間の状態遷移

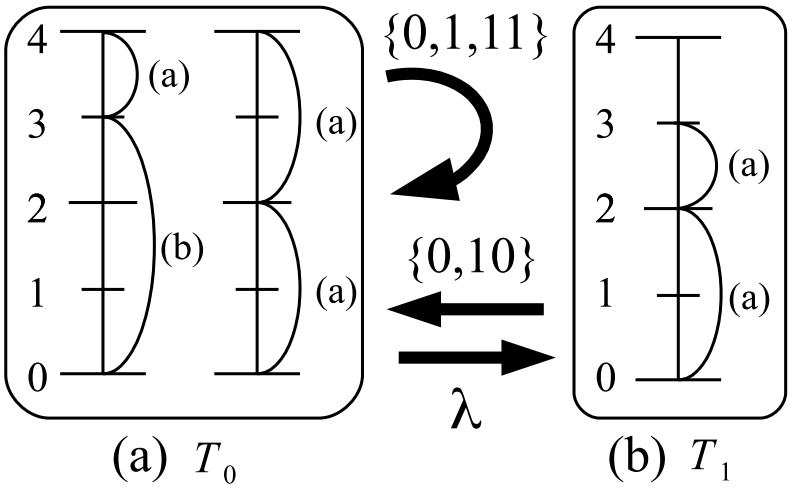

図 2　区間の状態遷移（順序保存性なし）

3.1　A^* アルゴリズムの概要

A^* アルゴリズム [3] とはグラフ探索アルゴリズムのひとつである．スタートノードから複数あるゴールノードまでの経路の中から最短の経路を探索する．最短経路を探索するために評価関数 f を使う．これは，関数 g と関数 h の合計で表される．関数 g はスタートノードから現在のノードまでの正確な距離を表す．関数 h はヒューリスティック関数と呼ばれ，現在のノードからゴールノードまでの推定距離を表す．したがって，g と h の合計である評価関数 f は，現在のノードを経てスタートノードからゴールノードまでの距離の推定値を表す．

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．

A^* アルゴリズムはスタートノードを OPEN に追加することから始まる．次に，評価関数 f の推定値が最小のノードを OPEN から取り出す．このノードを親ノードと呼ぶ．そして，親ノードから繋がるすべてのノードを子ノードと呼ぶ．すべての子ノードは OPEN に追加される．同様に，OPEN の中から f の推定値が最小の新たな親ノードを取り出す．この動作を繰り返し，f の推定値がより正確な値に更新される．子ノードの推定値が親ノードの推定値以上となるように評価関数 f を設計することで，OPEN から最初に取り出したゴールノードまで

の経路が最短経路になる．以下ではこの A^* アルゴリズムを応用して最適な算術符号の探索を提案する．

3.2 探索アルゴリズムの概要

本提案アルゴリズムではスタートノードからゴールノードまでの経路上でシンボルが部分区間を次々に獲得する．つまり，親ノードから子ノードへの枝上でひとつのシンボルにひとつの区間が割り当てられる．基本的な考え方としては，確率の大きいシンボルから順に大きい区間が割り当てられる．スタートノードではすべてのシンボルに区間が割り当てられていない．一方で，ゴールノードではすべてのシンボルに区間が割り当てられる．本提案アルゴリズムにおいて，スタートノードからゴールノードまでの距離は平均符号語長を表している．つまり，すべてのシンボルに区間が割り当てられている複数のゴールノードから最小の平均符号語長のゴールノードを探索する．

本研究で扱う算術符号は区間の拡大によってより小さい幅の区間をシンボルに割り当てることができる．よって，シンボルに割り当てる区間の候補は無限個ある．つまり，親ノードには無限個の子ノードが繋がることになる（図３）．しかし，A^* アルゴリズムを適用するためには親ノードに繋がる子ノードは有限個である必要がある．そこで，親ノードから情報源シンボルに区間を割り当てるかどうかを決める子ノードを作成し，２進木のグラフを構成する（図４）．以下では各ノードの構造と２進木の作成方法を説明する．

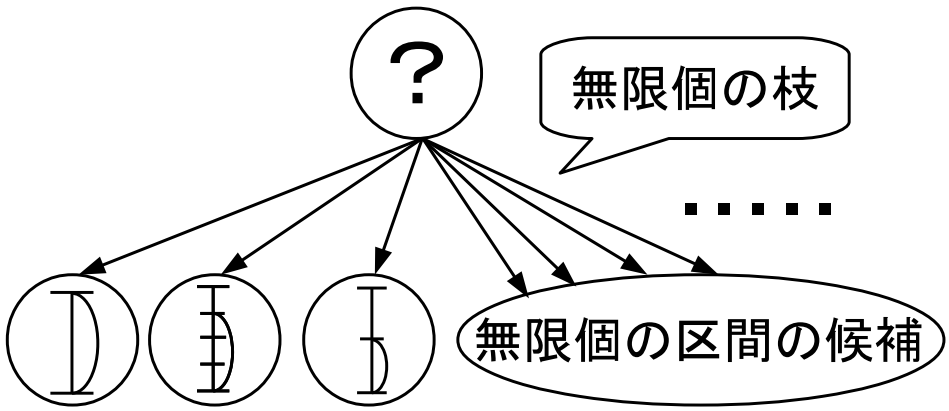

図３　無限個の区間の候補による無限木

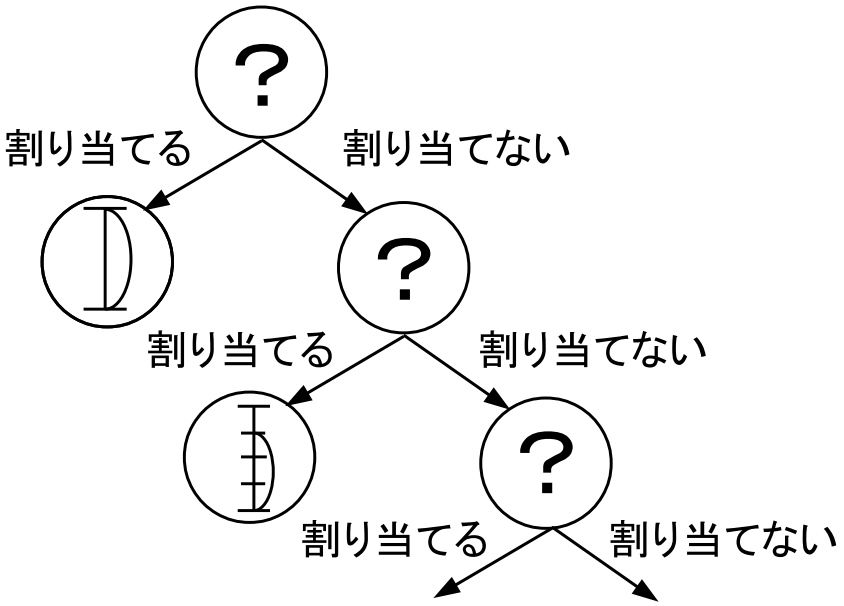

図 4　提案アルゴリズムの 2 進木

3.3　ノードの構造と作成

各ノードを形式的に区間の状態 T_k に対応した $\{(C_k, A_k)\}_{k=0,1}$ のように表す．C_k と A_k はそれぞれ，シンボルに割り当てられた区間のリストと，シンボルに割り当てる区間の候補を示している．スタートノードの A_0 は T_0 の区間の候補なので全区間 $[0, 4)$ のすべての部分区間が含まれる（図 5 ）．一方で，スタートノードの A_1 は T_1 の区間の候補なので区間 $[0, 3)$ のすべての部分区間が含まれる．本研究の算術符号の精度は 2 ビットのため，分割点は 3 つしかない．しかし，シンボルは区間の拡大によって，より小さい区間を割り当てることができる．これを図 5 の“ × ”を用いて表す．例えば，$[0, 2) \times [0, 3)$ は $[0, \frac{3}{2})$ を意味する．

次に子ノードの作成方法について説明する．OPEN から取り出した親ノードを修正することで 2 つの子ノードを作成することができる．まだ区間を割り当てていない最も確率の大きいシンボルを次に区間を割り当てるシンボルとする．ひとつの子ノードは，そのシンボルに区間の候補 A_k の中から最も大きい区間を割り当てる．また，区間の候補 A_k は割り当てた区間とその区間と重なる区間のすべてを消去する．もうひとつの子ノードでは，そのシンボルに区間

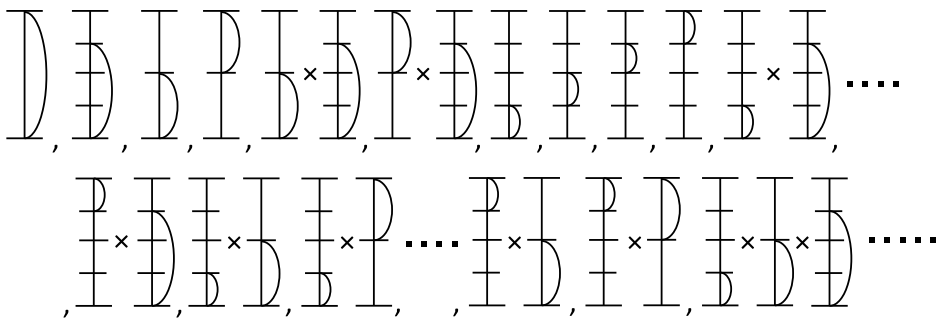

図 5　区間の候補 A_0

を割り当てず，区間の候補 A_k の中から最も大きい区間を消去する．これらの 2 つの子ノードは OPEN に追加される．ただし，シンボルの数に対して区間の候補が不足した場合，算術符号として成立しないため，そのようなノードは消去する．最終的に親ノードは CLOSED に入れられるが，本提案アルゴリズムは木構造のため消去できる．

3.4　平均符号語長の推定値

A^* アルゴリズムのための評価関数 $f = g + h$ を定義する．本研究では評価関数 f として符号化確率を用いて平均符号語長を推定する．T_0 では，全区間 $[0, 4)$ の部分区間をシンボルに割り当てるため，符号化確率は区間の幅を $1/4$ 倍したものである．一方，T_1 では，区間 $[0, 3)$ の部分区間を割り当てるため，符号化確率は区間の幅を $1/3$ 倍したものである．

まだ区間を割り当てていないシンボルの符号化確率を推定するために，区間の候補 A_k から大きい区間から順に割り当てると仮定する．この仮定したまだ割り当てていないシンボルの符号化確率は後で実際に割り当てられる値より大きく推定されていることに注意する．ここで，$a \in \mathcal{X}$ の確率を $p(a)$，符号化確率を $q_k(a)$ と表し，T_k の平均符号語長の推定値 l_k を

$$l_k \triangleq -\sum_{a \in \mathcal{X}} p(a) \log_2 q_k(a) \tag{3}$$

のように定義する．右辺の合計は関数 g と h の合計に対応している．つまり，確定値である g の値はすでに区間を割り当てたシンボルに対する値，推定値である h の値はまだ区間を割り当てていないシンボルに対する値に対応する．ここで h の値は実際の値より小さく推定されていることに注意する．

さらに，T_0 と T_1 の定常分布を求めるために遷移確率を計算する必要がある．区間の状態の遷移先はシンボルの割り当てが $\times[0, 3)$ で終わる場合，T_1 に遷移する．それ以外の場合は T_0 に遷移する．また，まだ区間を割り当てていないシンボルの区間の状態は平均符号語長の推定値 l_k が小さい T_k に遷移すると仮定する．そうすることで全てのシンボルの区間の状態の遷移

先を決めることができ，状態遷移確率 $Q(1|0), Q(0|1)$ を決めることができる．$Q(1|0), Q(0|1)$ はそれぞれ T_0 から T_1 への遷移，T_1 から T_0 への遷移を示している．これより定常分布 Q_0, Q_1 は

$$Q_0 = \frac{Q(0|1)}{Q(0|1) + Q(1|0)}, \tag{4}$$

$$Q_1 = \frac{Q(1|0)}{Q(0|1) + Q(1|0)} \tag{5}$$

となり，

$$L = Q_0 l_0 + Q_1 l_1. \tag{6}$$

のように T_k の平均符号語長の推定値 l_k を定常分布で重み付けすることで符号としての平均符号語長の推定値（ノードの評価値）が得られる．すべてのシンボルが区間に割り当てられたとき，この推定値は符号の平均符号語長として正確な値となる．

ここで Algorithm 1 に本提案アルゴリズムの基本構造を示す．

Algorithm 1 順序保存性のない最適な算術符号の探索アルゴリズム

Input: 情報源シンボルの数 n(アルファベットサイズ)，各情報源シンボルの確率 $p_i, (i = 1, 2, \ldots, n)$.

Output: 順序保存性のない最適な算術符号

1: スタートノード S を初期化する．：$C_0(S) = \phi$, $C_1(S) = \phi$, $A_0(S)$, $A_1(S)$.
　評価関数 $L = Q_0 l_0 + Q_1 l_1$ を計算する．
　スタートノード S を OPEN に追加する．
2: **do**
3:　　OPEN から親ノードとしてノード P（始めはノード S）を取り出す．
4:　　親ノード P から 2 つの子ノード X, Y を作成する．
5:　　**if** $A_k(P) = \phi$ かつ $C_k(X)$ のアルファベットサイズ $< n$ **then**
6:　　　　ノード X を消去する．
7:　　**end if**
8:　　子ノードの評価関数 $L = Q_0 l_0 + Q_1 l_1$ を計算する．
9:　　子ノードを OPEN に追加する．
10: **while** $C_0(P)$ のアルファベットサイズ $\neq n$ または $C_1(P)$ のアルファベットサイズ $\neq n$.
11: 最適な符号として $C_0(P)$ と $C_1(P)$ を出力する．

4　提案アルゴリズムの評価

以上で説明した提案アルゴリズムの最適性を証明し，計算量についての実験結果を示す．また，算術符号の順序保存性を緩和したことによる平均符号語長の改善を確かめる．

4.1　最適性

提案アルゴリズムによって得られた符号が最小の平均符号語長となることを説明する．A^* アルゴリズム の特徴から子ノードの評価値が親ノードの評価値以上となることを証明すればよい．

まず，各 T_k における平均符号語長の推定値 l_k について考える．子ノードの l'_k が親ノードの l_k 以上となることを示す．l_k は式 (3) より確率分布 $p(a)$ と符号化確率 $q_k(a)$ で表す．また，l'_k は確率分布 $p(a)$ と子ノードの符号化確率 $q'_k(a)$ で表される．本提案アルゴリズムではシンボルに割り当てる区間は幅の大きいものから割り当てるため，子ノードで割り当てた区間の符号化確率は親ノードの符号化確率と比べて増加しない．つまり，$q_k(a) \geq q'_k(a)$ という関係から，

$$\begin{aligned} l'_k &= -\sum_{a\in\mathcal{X}} p(a)\log_2 q'_k(a) \\ &\geq -\sum_{a\in\mathcal{X}} p(a)\log_2 q_k(a) \\ &= l_k. \end{aligned} \tag{7}$$

となることが示せる．

次に，定常分布を考慮する．区間の割り当てを決めていないシンボルの区間の状態の遷移先を仮定したことにより，T_k への遷移は増えない．つまり，$Q(k|0) \geq Q'(k|0), Q(k|1) \geq Q'(k|1)$ となる．そこで，遷移確率は

$$Q'(k|1-k) \leq Q(k|1-k), \tag{8}$$

$$Q'(1-k|k) \geq Q(1-k|k). \tag{9}$$

を満たす．この時，定常分布は

$$\begin{aligned} Q'_k &= \frac{Q'(k|1-k)}{Q'(k|1-k)+Q'(1-k|k)} \\ &\leq \frac{Q(k|1-k)}{Q(k|1-k)+Q'(1-k|k)} \\ &\leq \frac{Q(k|1-k)}{Q(k|1-k)+Q(1-k|k)} \\ &= Q_k. \end{aligned} \tag{10}$$

を満たす．

以上で示した $l_k \leq l_{1-k}$, $l'_k \geq l_k$, $l'_{1-k} \geq l_{1-k}$, $Q'_{1-k} \geq Q_{1-k}$ より

$$\begin{aligned} L' &= Q'_k l'_k + Q'_{1-k} l'_{1-k} \\ &\geq Q'_k l_k + Q'_{1-k} l_{1-k} \\ &= Q'_k l_k + (Q'_{1-k} - Q_{1-k}) l_{1-k} + Q_{1-k} l_{1-k} \\ &\geq Q'_k l_k + (Q'_{1-k} - Q_{1-k}) l_k + Q_{1-k} l_{1-k} \\ &= Q'_k l_k + (Q_k - Q'_k) l_k + Q_{1-k} l_{1-k} \\ &= Q_k l_k + Q_{1-k} l_{1-k} \\ &= L. \end{aligned} \tag{11}$$

となる．このことから，子ノードの評価値が親ノードの評価値以上となることがわかる．さらに，A^* アルゴリズムの特徴から最初に取り出したゴールノードよりも平均符号語長が小さいノードはすでに削除されている．そのため，最初に取り出したゴールノードが最適な符号を表している．

4.2 計算量

本提案アルゴリズムの計算量を測定した．情報源として一様分布の情報源シンボルを用いた．まず，時間の尺度として，このアルゴリズムが実行されている最中に OPEN から親ノードを選ぶ回数を数えた．また，使用メモリの尺度として，このアルゴリズムが終了した時に，OPEN に残っているノードの数を数えた．その結果を図 6 に示す．このアルゴリズムは 2 進木で，最大でも 2 つの子ノードが 1 つの親ノードで構成されているため，残りのノード数と親ノードを選ぶ回数はほぼ等しいという結果になった．図 6 から本提案アルゴリズムの計算上の複雑さは指数関数的であると考えられる．

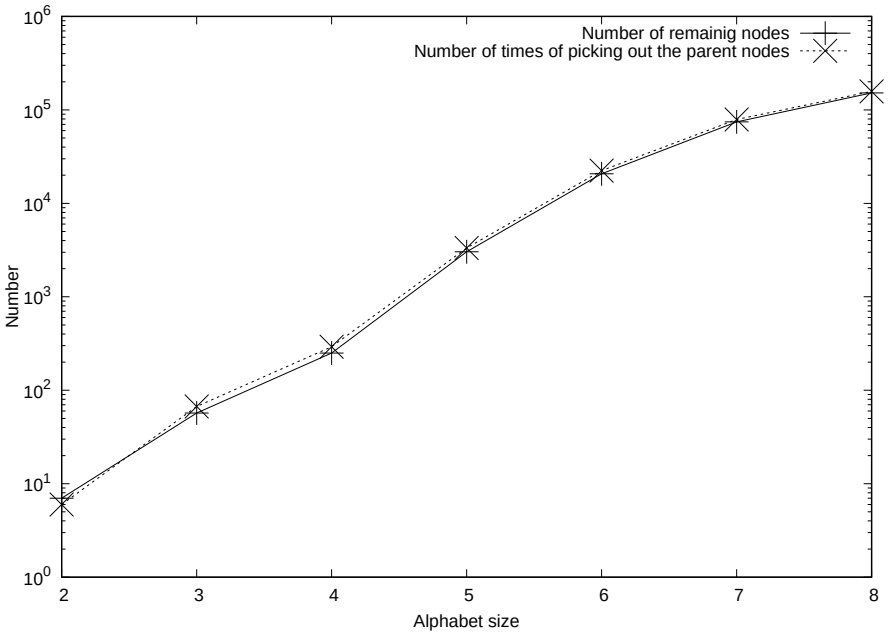

図 6　提案アルゴリズムの計算量

4.3　順序保存性による平均符号語長の比較

本提案アルゴリズムにより得られた順序保存性のない最適な算術符号と従来の順序保存性のある算術符号の平均符号語長を比較した．両者の精度は 2 ビットで固定した．その結果を表 1 に示す．情報源が一様分布であれば，平均符号語長に変化はなかった．なぜなら，情報源シンボルを入れ替えても確率分布が変わらないからである．また，情報源に偏りがある場合，情報源シンボルの順序によって従来の算術符号よりも小さい平均符号語長を得られる．このとき，平均符号語長に変化がないものは最適な情報源シンボルの順序であるといえる．また，確率が 0.4, 0.4, 0.1, 0.1 の情報源のように大きい確率のシンボルから並べるだけでは最適にはならない．情報源シンボルの順序によって，図 1 (d) のような区間の状態でシンボルが区間に割り当てられるため，冗長度が大きくなったと考えられる．

5　AIFV 符号との関係

本研究で取り扱う算術符号は AIFV 符号 [4] と等価となる．まず，AIFV 符号の概要を説明し，算術符号と AIFV 符号の関係を述べる．

表 1　順序保存性による算術符号の平均符号語長の比較

	平均符号語長	
情報源	順序保存性あり	順序保存性なし
0.25, 0.25, 0.25, 0.25	2.000	2.000
0.2, 0.2, 0.2, 0.2, 0.2	2.400	2.400
0.4, 0.4, 0.1, 0.1	1.800	1.743
0.8, 0.05, 0.05, 0.05, 0.05	1.155	1.155
0.05, 0.8, 0.05, 0.05, 0.05	1.638	1.155
0.9,0.025, 0.025, 0.025, 0.025	0.826	0.826
0.025, 0.9, 0.025, 0.025, 0.025	1.484	0.826

5.1　AIFV 符号の概要

AIFV 符号 [4] 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．m 個の符号木を使用するとき，AIFV-m 符号とする．ここで，AIFV-2 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（AIFV-2 符号）

1. AIFV-2 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（AIFV-2 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（AIFV-2 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 AIFV-2 符号の探索アルゴリズムとして文献 [5]，[6]，[7] が提案されている．AIFV-2 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．もうひとつは，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 の定常分布が大きくなる．つまり，このふたつはトレードオフである．ここで，文献 [5]，[6]，[7] ではコストという不確定な値を用いて評価値を計算し，コスト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．

5.2 算術符号と AIFV-2 符号の対応

本研究で扱う算術符号と AIFV-2 符号は等価となる．そこで，本研究の算術符号と AIFV-2 符号の定義を比較する．AIFV-2 符号は 2 個の符号木からなる．本研究の算術符号の精度は 2 ビットのため，区間の状態 T_0, T_1 で区間 $[0,4)$, $[0,3)$ それぞれの部分区間をシンボルに割り当てる．割り当てに応じた符号木が AIFV-2 符号の 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，区間の状態 T_1 の候補 A_1 において $[3,4)$ の区間と重なる区間は候補となりえないことに対応している．また，AIFV-2 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，区間 $[0,2)$, $[2,4)$ に割り当てたシンボルが符号木の葉に，区間 $[0,3)$ に割り当てたシンボルが符号木の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，区間 $[0,3)$ に割り当て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区間 $[3,4)$ に必ず割り当てることに対応している．

次に，具体例を示す．$P(a) = 0.45$, $P(b) = 0.3$, $P(c) = 0.2$, $P(d) = 0.05$ の分布を持つ情報源 $\mathcal{X} = \{a, b, c, d\}$ に対して，最適な AIFV-2 符号の符号木を図 7 に，順序保存性のない最適な算術符号の符号木を図 8 に示す．AIFV-2 符号の T_0 で c が，T_1 で a と b が主ノードに割り当てられている．また，主ノードと繋がるシンボルは 00 で繋がる．一方で，算術符号の T_0 でも c が，T_1 でも a と b が区間 $[0,3)$ に割り当てられる．このとき他のシンボルが全区間を埋めるように区間 $[3,4)$ に割り当てられる．これを符号木で示すと図 8 のように主ノードと繋が

るシンボルは 11 で繋がる．さらに，符号語シンボルの反転はあるが，符号語長は等しい．両者の平均符号語長は 1.738 である．ちなみに，情報源のエントロピーは約 1.720 である．

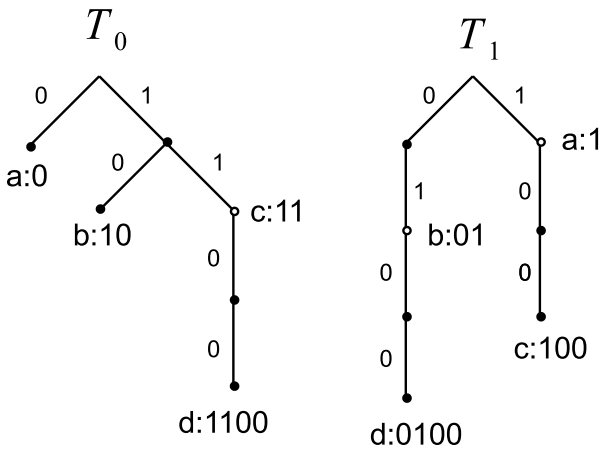

図 7　最適な AIFV-2 符号の例

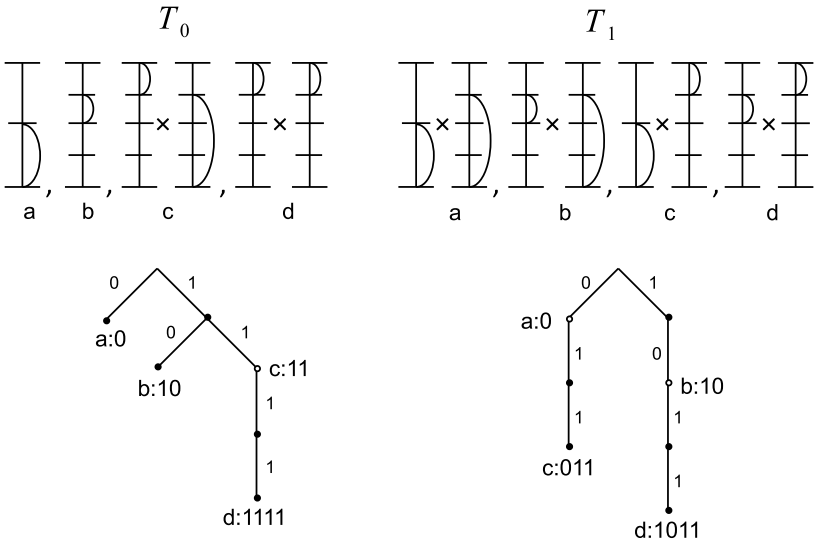

図 8　順序保存性のない最適な算術符号の例

5.3 復号遅延と符号木

AIFV-m 符号の復号遅延は m ビットであることが知られている [4]．つまり，AIFV-2 符号の復号遅延は 2 ビットである．また，本研究で扱った精度 2 の算術符号の復号遅延は 2 ビットである．まず，図 7, 8 から復号遅延の例を示す．AIFV-2 符号と算術符号の最初の入力が a のとき，どちらも符号語 0 が出力される．これは復号器で a を復号するための情報として十分である．一方で，最初の入力が c のとき，どちらも符号語 11 が出力される．これは復号器で c を復元するための情報として不十分である．ここで c を復号するためには，さらに 0 または 10 の符号語を受け取る必要がある．このことから復号遅延は高々 2 ビットである．

AIFV 符号の符号木数と算術符号の区間の状態数を考察する．復号遅延が 2 ビットの AIFV-2 符号の符号木は 2 つからなるのに対し，同様に復号遅延が 2 ビットの算術符号は図 1 の区間の状態からなる符号木は 4 つである．4 つの区間の状態に対応した符号木を定義すれば復号遅延が 2 ビットの AIFV 符号を 4 つの符号木で構成できる．しかし，算術符号と同様に順序保存性を緩和することで 2 つの符号木で構成している．また，復号遅延が 3 ビットの AIFV-3 符号は符号木は 3 つからなる．同様に復号遅延が 3 ビットの算術符号の区間の状態を図 9 に示す．精度 3 の全区間 $[0, 8)$ から段階的に 2 分割した際に，図 9 の 16 通りの区間の状態で区間は拡大されずに次のシンボルの入力を待つ必要がある．ここで，順序保存性を緩和することで区間の状態を制限することができる．その例として，区間 $[0, 8), [0, 7), [0, 6), [2, 8), [0, 5), [3, 8), [2, 6), [2, 5), [3, 6)$ の 9 通りに制限できる．つまり，AIFV-3 符号は 16 個の符号木で構成でき，それを 9 個の符号木に制限することができる．AIFV-3 符号の 3 つの符号木は $[0, 8), [0, 7), [0, 6)$ の区間の状態に対応しているが，その他の 6 通りの区間の状態に対応した符号木を無視している．そこで，算術符号の区間の状態と比較したとき，AIFV-3 符号は符号木を少なく見積もっていると考えられる．さらに，復号遅延の増加に対して AIFV 符号の符号木は線形的に増加するが，算術符号の符号木は指数的に増加すると考えられる．つまり，AIFV-m 符号は復号遅延 m ビットで得られる符号を制限して簡単化した符号であるとみなすことができる．

5.4 コストの考察

本提案アルゴリズムと文献 [5] で提案されている最適な AIFV 符号の関係を考察する．最適な AIFV 符号はまず，T_0, T_1 それぞれで最適な符号を作成する．評価関数には不確定なコストという値を用いる．コストの値は主ノードを使用したときに，評価値に加算される値である．不確定なコストを確定するために探索アルゴリズムを繰り返し，コストが確定したとき最適な AIFV 符号が得られる．また，コストの値は以下のように定義されている．それぞれの符号木

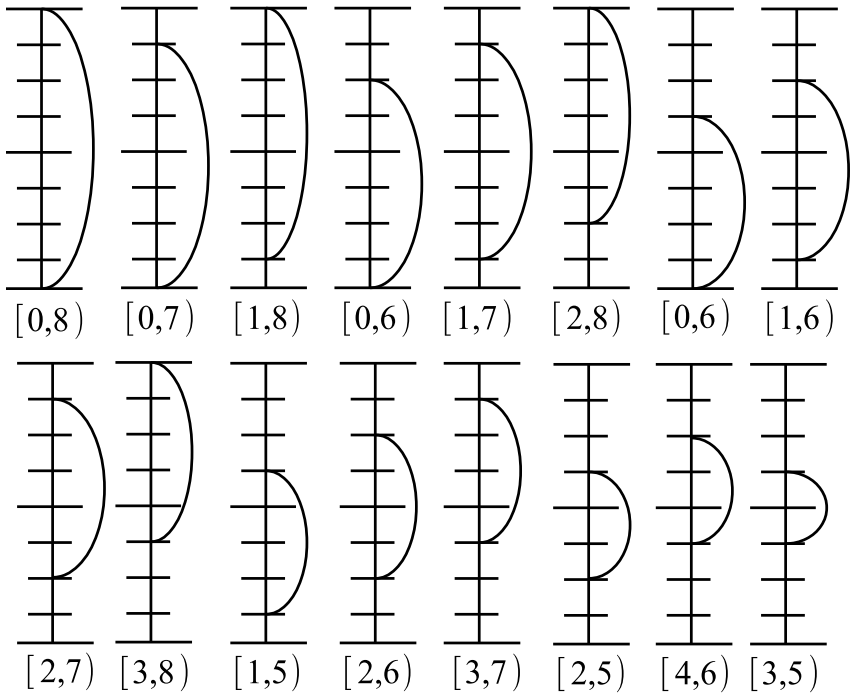

図 9　精度 3 の算術符号における区間の状態

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 $Q(1|0) = q_{10}, Q(0|1) = q_{01}$ と表記する．このときコスト c を

$$c = \frac{l_1 - l_0}{q_{10} + q_{01}} \tag{12}$$

と表し，値が確定するまでアルゴリズムを繰り返し行う．一方で，本提案アルゴリズムでは T_0, T_1 の定常分布を考慮して評価関数を作成した．つまり，文献 [5]，[6]，[7] で提案されているようなアルゴリズムの繰り返しは必要としない．

ここで，文献 [5] などのコスト c が本研究の算術符号においてどのような値となるか考察する．AIFV 符号の T_k の平均符号長 l_k は $a \in \mathcal{X}$ の確率 $p(a)$，符号語長 $l_k(a)$ に対して，

$$l_k = \sum_{a \in \mathcal{X}} p(a) l_k(a) \tag{13}$$

である．ここで，区間の状態 T_0, T_1 に対して $a \in \mathcal{X}$ の区間の幅を $t_0(a), t_1(a)$，符号化確率を

$q_0(a), q_1(a)$ とすると，

$$q_0(a) = \frac{1}{4} t_0(a), \tag{14}$$

$$q_1(a) = \frac{1}{3} t_1(a) \tag{15}$$

となる．符号語長 $l_0(a), l_1(a)$ は符号化確率を用いて，シンボルに区間 $[0, 3)$ を割り当てる場合とその他の場合で分けると，

$$l_0(a) = \begin{cases} -\log_2 \frac{1}{4} t_0(a) = -\log_2 q_0(a) & (otherwise) \\ -\log_2 (\frac{1}{4} t_0(a) \cdot \frac{4}{3}) = -\log_2 q_0(a) - \log_2 \frac{4}{3} & (assign[0, 3)) \end{cases}, \tag{16}$$

$$l_1(a) = \begin{cases} -\log_2 (\frac{1}{3} t_1(a)) = -\log_2 q_1(a) & (assign[0, 3)) \\ -\log_2 (\frac{1}{3} t_1(a) \cdot \frac{3}{4}) = -\log_2 q_1(a) + \log_2 \frac{4}{3} & (otherwise) \end{cases} \tag{17}$$

となる．式 (16)(17) を式 (13) に代入すると，

$$l_0 = -\sum_{a \in \mathcal{X}} p(a) \log_2 q_0(a) - q_{10} \log_2 \frac{4}{3}, \tag{18}$$

$$l_1 = -\sum_{a \in \mathcal{X}} p(a) \log_2 q_1(a) + q_{01} \log_2 \frac{4}{3} \tag{19}$$

とかける．ここで，式 (12) に式 (18), (19) を代入すると，

$$c = \frac{l_1 - l_0}{q_{10} + q_{01}} \tag{20}$$

$$= \frac{1}{q_{10} + q_{01}} \left(-\sum_{a \in \mathcal{X}} p(a) \log_2 q_1(a) + \sum_{a \in \mathcal{X}} p(a) \log_2 q_0(a) + (q_{01} + q_{10}) \log_2 \frac{4}{3} \right) \tag{21}$$

$$= \frac{1}{q_{10} + q_{01}} \left(\sum_{a \in \mathcal{X}} p(a) \log_2 \frac{p(a)}{q_1(a)} - \sum_{a \in \mathcal{X}} p(a) \log_2 \frac{p(a)}{q_0(a)} \right) + \log_2 \frac{4}{3} \tag{22}$$

$$= \frac{1}{q_{10} + q_{01}} \{ D\left(p(a) || q_1(a)\right) - D\left(p(a) || q_0(a)\right) \} + \log_2 \frac{4}{3} \tag{23}$$

となる．式 (23) のように AIFV 符号のコストの値は本研究における T_0, T_1 それぞれのシンボルの確率分布と符号化確率のダイバージェンスを含んだ式でかける．ここで AIFV 符号のコストの初期値は 0.415 が最適値のよい近似であるとされている．この値は式 (23) の T_0 と T_1 のダイバージェンスが等しいときの $\log_2 \frac{4}{3}$ と同じ値である．2 つの T で冗長度に差があるとき，冗長度が大きい T ばかりに遷移すると符号の冗長度が大きくなってしまう．そこでコストの更新，または定常分布の考慮が必要となる．

6　計算量の削減に向けて

本章では最適な算術符号の探索アルゴリズムの計算量を削減するための考察を行う．本研究の算術符号と AIFV 符号には対応があるため，最適な AIFV 符号の探索アルゴリズムについて考察する．本提案アルゴリズムの計算量は指数関数的であるのに対して文献 [6] で提案されている内部アルゴリズムの計算量はアルファベットサイズ n で $O(n^5)$ の時間，$O(n^3)$ のメモリを使用する．文献 [6] の内部アルゴリズムはコストが確定するまで繰り返し行う．以下で文献 [6] のアルゴリズムについて説明する．

文献 [6] のアルゴリズムは符号木の深さを追ってアルゴリズムが動く．AIFV 符号の符号木は葉，完全内部ノード，不完全内部ノードからなる．1 つだけの子をもつ不完全内部ノードは主ノードと副ノードに分けられ，主ノードの子は副ノードでなくてはならない．このアルゴリズムは，各深さで 3 つの整数の組 (n_0, n_1, n_2) で表されるシグネチャーと呼ばれる情報をもっている．n_0 は符号木の深さ i までの葉または主ノードの数を示す．また，n_1 は深さ $i+1$ までの，n_2 は深さ $i+2$ までの葉や主ノードに決めていないノード数を表している．この n_1, n_2 のノードを未定ノードと呼ぶ．探索の始まりは木の深さが 0 から始まり，シグネチャーの初期状態は T_0 で $(0,1,0)$，T_1 で $(0,1,1)$ とする．次に，深さを 1 進めると未定ノードを葉または主ノード，完全内部ノードにする 3 通りの木を構成する．アルファベットサイズに対して葉または主ノードの数が足りないときは，さらに深さを下げる．同様に，それぞれの未定ノードを葉または主ノード，完全内部ノードに決定して木を構成する．以上のように木の深さを追って，葉または主ノードの数がアルファベットサイズになるすべての木のなかで，平均符号語長が最小なものが最適である．

図 10 に例を示す．葉または主ノードを $\bullet$ で，完全内部ノードを $\circ$ で，副ノードを $\blacksquare$ で，未定ノードを $\square$ で表す．T_0 の初期状態は $(0,1,0)$ で，深さ 0 までの葉または主ノードは 0 個で，深さ 1 までの未定ノードは 1 個である．深さを 1 進めると図 10 のように，未定ノードを葉または主ノード，完全内部ノードにする．それぞれのシグネチャーは $(1,0,0)$, $(1,0,1)$, $(0,2,0)$ となる．さらに深さを 1 進めたときにそれぞれの未定ノードを決める．つまり，図 10 の $(0,2,0)$ は深さ 2 で 6 通りの木ができ，未定ノードに対して構成可能な木を全探索していることになる．

このアルゴリズムの計算量が多項式オーダーの理由は，符号語の反転を気にせず符号語長だけに注目できるからだと考える．本研究で提案した探索アルゴリズムへの適用を考えると，区間の幅が同じで符号語長が同じなら区別する必要がなくなるため，アルゴリズムの冗長を減らせる．ただし，文献 [6] のアルゴリズムは内部アルゴリズムの計算量を示しており，本研究の繰り返しを必要としないアルゴリズムとの純粋な計算量の比較をするためには，外側のアルゴリズムを考慮した計算量を計測する必要がある．また，文献 [6] では全探索であったが，この

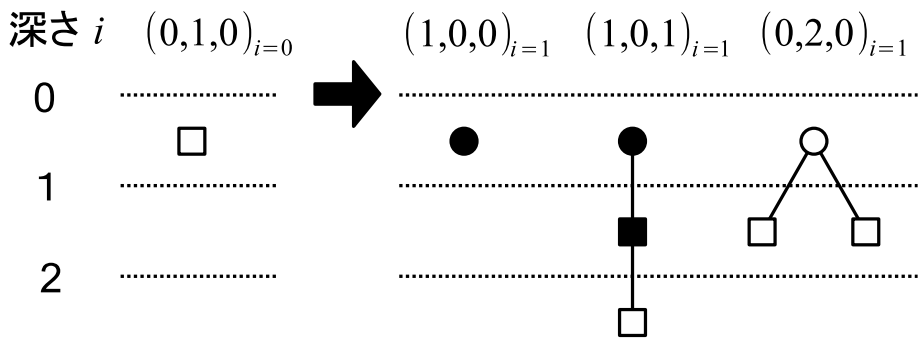

図 10　シグネチャーの例

アルゴリズムに A^* アルゴリズムを適用することによる計算量の影響については，さらなる研究を必要とする．課題として評価関数の定義，ノードに割り当てていないシンボルの仮の割り当て方，削除可能なノードの考慮などがある．

7　まとめ

本研究では精度 2 の順序保存性のない算術符号において平均符号語長が最小となる最適な算術符号を探索するアルゴリズムを提案した．本提案アルゴリズムは A^* アルゴリズムの評価関数として定常分布を用いることで 2 つの区間の状態を一度で探索している．また，このアルゴリズムの計算量は指数オーダーであると考えられる．

本研究で扱った算術符号は AIFV-2 符号と等価となる．そこで算術符号と AIFV 符号の関係を考察した．また，最適な AIFV 符号の探索アルゴリズムを参考に本提案アルゴリズムの計算量を削減できるか検討していく．

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．また，本研究内容についてご議論をいただいた山本博資東京大学名誉教授と岩田賢一福井大学准教授に深く感謝申し上げます．

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

[4] 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.

[5] Ryusei Fujita, Ken-ichi Iwata, and Hirosuke Yamamoto, “An Optimality Proof of the Iterative Algorithm for AIFV-m Codes,” IEEE International Symposium on Information Theory, pp.2187–2191, June 2018.

[6] Ken-ichi Iwata, Hirosuke Yamamoto, “A Dynamic Programming Algorithm to Construct Optimal Code Tree of AIFV Codes,” ISITA2016, Monterey, California, USA, pp.672–676, 2016.

[7] 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.

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 <time.h>

#define MAX_CODEWORDLEN 32

int alphabet_size;
double *prob;

struct state_tree {
        int fixed; //選択済みの数
        char **codeword;   //文字列
        double cost;       //単独のコスト
        double trans;      //遷移確率
};

struct state {
        struct state_tree t_name[2];
        double s_prob1;    //定常分布
        double total_cost;  //総合コスト
};

void
printstate(struct state *p)
{
        int i;
        printf("T0:");
        for (i = 0; i < p->t_name[0].fixed; i++) {
                printf("%s ", (p->t_name[0].codeword[i]) ? p->t_name[0].codeword[i] : "(NULL)");
        }
        putchar('|');
        for (; i < alphabet_size; i++) {
                printf(" %s", (p->t_name[0].codeword[i]) ? p->t_name[0].codeword[i] : "(NULL)");
        }
        printf("\nT1:");
        for (i = 0; i < p->t_name[1].fixed; i++) {
                printf("%s ", (p->t_name[1].codeword[i]) ? p->t_name[1].codeword[i] : "(NULL)");
        }
        putchar('|');
        for (; i < alphabet_size; i++) {
                printf(" %s", (p->t_name[1].codeword[i]) ? p->t_name[1].codeword[i] : "(NULL)");
        }
        printf("\n%f,%f,%f,%f\n", p->s_prob1, p->t_name[0].cost, p->t_name[1].cost, p->total_cost);
        return;
}

/*******************************エラー処理*********************************************************************/

void *
malloc_ed(size_t size, int line)
{
        void *ret = malloc(size);
        if (ret == NULL) {
                printf("error in %d\n", line);
                exit(1);
        }
        return(ret);
}

char *
```

```
strdup_ed(char * str, int line)
{
        char *ret = strdup(str);
        if (ret == NULL) {
                printf("error in %d\n", line);
                exit(1);
        }
        return(ret);
}

/*****************************次の文字列を生成*********************************************************************/

char *
nextword(char *w)
{
        static char ret[MAX_CODEWORDLEN];
        char *r;
        char *zero;
        char *term;
        for (zero = NULL, term = w; *term != '\0'; term++) {
                if (*term == '0') {
                        zero = term;
                }
        }
        if (zero) {
                if (term - w > MAX_CODEWORDLEN - 1) {
                        printf("error in %d\n", __LINE__);
                        exit(1);
                }
                for (r = ret; w != zero; r++, w++) {
                        *r = *w;
                }
                *r++ = '1';
                *w++;
        }
        else {
                if (term - w > MAX_CODEWORDLEN - 2) {
                        printf("error in %d\n", __LINE__);
                        exit(1);
                }
                r = ret;
                *r++ = '0';
        }
        for (; w != term; r++, w++) {
                *r = '0';
        }
        *r = '\0';
        return(ret);
}

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

int
isdisjoint(char *a, char *b)
{
        char f = *a;

        do {
                a++, b++;
        } while (*a != '\0' && *b != '\0' && *a == *b);
        if (*a != '\0') {
                return(1);
        }
        if (f == '1' && b[0] == '1' && b[1] == '1') {
                return(1);
        }
        return(0);
}
```

```
void
replace_codeword(struct state state, int i_name)//区間の候補（符号語）の補充
{
        int i, j;
        char *can;

        for (i = alphabet_size - 1; state.t_name[i_name].codeword[i] == NULL; i--) {
                ;
        }
        if (i == alphabet_size - 1) {
                return;
        }
        for (i++; i < alphabet_size; i++) {
                can = state.t_name[i_name].codeword[i - 1];
        NEXT_CANDIDATE:
                can = nextword(can);
                if (i_name == 1) {
                        while (can[1] == '1' && can[2] != '0') {
                                can = nextword(can);
                        }
                }
                for (j = 0; j < state.t_name[i_name].fixed; j++) {//選択済みの区間と作成した区間の比較
                        if (!isdisjoint(state.t_name[i_name].codeword[j], can)) {
                                goto NEXT_CANDIDATE;
                        }
                }
                state.t_name[i_name].codeword[i] = strdup_ed(can, __LINE__);
        }
        return;
}

int
kraft(struct state_tree *p, int i_name)
{
        int maxlen = 2;
        int remain = (i_name == 0) ? 4 : 3;
        int i;

        for (i = 0; i < p->fixed; i++) {
                int dep = strlen(p->codeword[i]) + 1;
                if (dep > maxlen) {
                        remain *= 1 << (dep - maxlen);
                        maxlen = dep;
                }
                remain -= (1 << (maxlen - dep)) * (p->codeword[i][0] == '0') ? 4 : 3;
        }
        return(remain);
}

/*****************************割り当てない場合*********************************************************************/
struct state
        removecandidate(struct state *pstate, int i_name)
{
        struct state nstate;
        int i;
        if (pstate == NULL) {
                printf("[%d]\n", __LINE__);
                exit(1);
        }
        nstate.t_name[i_name].codeword = (char **)malloc_ed(sizeof(char*) * alphabet_size, __LINE__);
        nstate.t_name[i_name].fixed = pstate->t_name[i_name].fixed;//コピーの作成
        for (i = 0; i < nstate.t_name[i_name].fixed; i++) {
                nstate.t_name[i_name].codeword[i] = strdup_ed(pstate->t_name[i_name].codeword[i], __LINE__);
        }
        for (; i < alphabet_size - 1; i++) {
                nstate.t_name[i_name].codeword[i] = strdup_ed(pstate->t_name[i_name].codeword[i + 1], __LINE__);
        }
        nstate.t_name[i_name].codeword[alphabet_size - 1] = NULL;
        replace_codeword(nstate, i_name);
```

```
        nstate.t_name[1 - i_name].codeword = (char **)malloc_ed(sizeof(char*) * alphabet_size, __LINE__);
        nstate.t_name[1 - i_name].fixed = pstate->t_name[1 - i_name].fixed;//コピーの作成
        for (i = 0; i < alphabet_size; i++) {
                nstate.t_name[1 - i_name].codeword[i] = strdup_ed(pstate->t_name[1 - i_name].codeword[i], __LINE__);
        }

        return(nstate);
}

/*********************************割り当てる場合*********************************************************************/
struct state
        selected_interval(struct state *pstate, int i_name)
{
        struct state nstate;
        int i, j;

        nstate.t_name[i_name].codeword = (char **)malloc_ed(sizeof(char*) * alphabet_size, __LINE__);
        nstate.t_name[i_name].fixed = pstate->t_name[i_name].fixed + 1;//コピーの作成
        for (i = 0; i < nstate.t_name[i_name].fixed; i++) {
                nstate.t_name[i_name].codeword[i] = strdup_ed(pstate->t_name[i_name].codeword[i], __LINE__);
        }
        for (; i < alphabet_size; i++) {
                nstate.t_name[i_name].codeword[i] = NULL;
        }
        for (i = j = nstate.t_name[i_name].fixed; i < alphabet_size; i++) {
                if (isdisjoint(nstate.t_name[i_name].codeword[nstate.t_name[i_name].fixed - 1],
                        pstate->t_name[i_name].codeword[i])) {
                        nstate.t_name[i_name].codeword[j++] = strdup_ed(pstate->t_name[i_name].codeword[i], __LINE__);
                }
        }

        nstate.t_name[1 - i_name].codeword = (char **)malloc_ed(sizeof(char*) * alphabet_size, __LINE__);
        nstate.t_name[1 - i_name].fixed = pstate->t_name[1 - i_name].fixed;//コピーの作成
        for (i = 0; i < alphabet_size; i++) {
                nstate.t_name[1 - i_name].codeword[i] = strdup_ed(pstate->t_name[1 - i_name].codeword[i], __LINE__);
        }

        if (nstate.t_name[i_name].fixed < alphabet_size) {//区間を超えないかの確認
                if (kraft(&nstate.t_name[i_name], i_name) <= 0) {
                        nstate.t_name[i_name].fixed += alphabet_size + 1;
                        return(nstate);
                }
                replace_codeword(nstate, i_name);
        }

        if (nstate.t_name[i_name].fixed == alphabet_size - 1) {
                nstate.t_name[i_name].fixed = alphabet_size;
        }

        return(nstate);
}
/*****************************コスト計算*************************************************************/

double c_cost(struct state *pstate) {
        int i;
        double len0, len1, q0, q1;
        int best;

        pstate->t_name[0].cost = 0;
        pstate->t_name[1].cost = 0;
        for (i = 0; i < alphabet_size; i++) {
                len0 = strlen(pstate->t_name[0].codeword[i]) - 1;
                if (pstate->t_name[0].codeword[i][0] == '1') {
                        len0 += 0.41503749927884381854626105605218; /* log2(4/3) */
                }
                pstate->t_name[0].cost += prob[i] * len0;
                len1 = strlen(pstate->t_name[1].codeword[i]) - 1;
                if (pstate->t_name[1].codeword[i][0] == '1') {
```

```
                        len1 += 0.41503749927884381854626105605218;
                }
                pstate->t_name[1].cost += prob[i] * len1;
        }
        pstate->t_name[1].cost -= 0.41503749927884381854626105605218;

        best = (pstate->t_name[0].cost < pstate->t_name[1].cost) ? 0 : 1;
        pstate->t_name[0].trans = 0;
        pstate->t_name[1].trans = 0;
        for (i = 0; i < pstate->t_name[0].fixed; i++) {
                if (pstate->t_name[0].codeword[i][0] == '1') {
                        pstate->t_name[0].trans += prob[i];
                }
        }
        if (best == 1) {
                for (; i < alphabet_size; i++) {
                        pstate->t_name[0].trans += prob[i];
                }
        }
        if (pstate->t_name[0].trans > 0) {
                for (i = 0; i < pstate->t_name[1].fixed; i++) {
                        if (pstate->t_name[1].codeword[i][0] == '0') {
                                pstate->t_name[1].trans += prob[i];
                        }
                }
                if (best == 0) {
                        for (; i < alphabet_size; i++) {
                                pstate->t_name[1].trans += prob[i];
                        }
                }
                q1 = (pstate->t_name[0].trans) / ((pstate->t_name[0].trans) + (pstate->t_name[1].trans));
                q0 = 1.0 - q1;
        }
        else {
                q0 = 1.0;
                q1 = 0.0;
        }
        pstate->s_prob1 = q1;

        pstate->total_cost = (q0 * (pstate->t_name[0].cost)) + (q1 * (pstate->t_name[1].cost));
        return(pstate->total_cost);
}

/*****************************************************************************************/
void
initlist(struct state *p)//初期設定
{
        int i;
        char *can;
        p->t_name[0].fixed = 0;//一番最初の state に入力していく↓
        p->t_name[0].codeword = (char **)malloc_ed(sizeof(char *) * alphabet_size, __LINE__);
        can = "0";
        for (i = 0; i < alphabet_size; i++) {
                can = nextword(can);
                p->t_name[0].codeword[i] = strdup_ed(can, __LINE__);
        }

        p->t_name[1].fixed = 0;//一番最初の state に入力していく↓
        p->t_name[1].codeword = (char **)malloc_ed(sizeof(char *) * alphabet_size, __LINE__);
        can = "1";
        for (i = 0; i < alphabet_size; i++) {
                can = nextword(can);
                while (can[1] == '1' && can[2] != '0') {
                        can = nextword(can);
                }
                p->t_name[1].codeword[i] = strdup_ed(can, __LINE__);
        }
        return;
}
```

```
/*****************************************************************************************/

#define LISTBOUND 10000000
struct state list[LISTBOUND];
int num_state; //state の数

void
state_free(struct state state)
{
        int i;

        for (i = 0; i < alphabet_size; i++) {
                free(state.t_name[0].codeword[i]);
                free(state.t_name[1].codeword[i]);
        }
        free(state.t_name[0].codeword);
        free(state.t_name[1].codeword);
        return;
}

void
add_list(struct state *pstate)
{
        int i;
        if (num_state >= LISTBOUND) {
                printf("error in %d\n", __LINE__);
                exit(1);
        }
        for (i = num_state; i > 0; i--) {
                if (pstate->total_cost < list[i - 1].total_cost) {
                        break;
                }
                list[i] = list[i - 1];
        }
        list[i] = *pstate;
        num_state++;
        return;
}

void
find_AC()
{
        int i_name;
        int i = 0;
        initlist(list);
        list[0].s_prob1 = 0;
        c_cost(list);
        num_state = 1;
        while (list[num_state - 1].t_name[0].fixed < alphabet_size ||
                   list[num_state - 1].t_name[1].fixed < alphabet_size) {
                i = i + 1;
                struct state state0, state1;
                double cost = list[num_state - 1].total_cost;

                if (list[num_state - 1].t_name[0].fixed == alphabet_size) {
                        i_name = 1;
                }
                else if (list[num_state - 1].t_name[1].fixed == alphabet_size) {
                        i_name = 0;
                }
                else {
                        i_name = (list[num_state - 1].t_name[0].cost < list[num_state - 1].t_name[1].cost) ? 0 : 1;
                }

                state0 = removecandidate(&list[num_state - 1], i_name);
                state1 = selected_interval(&list[num_state - 1], i_name);
                state_free(list[num_state - 1]);
                num_state--;
```

```
                if (state0.t_name[0].fixed <= alphabet_size && state0.t_name[1].fixed <= alphabet_size) {
                        c_cost(&state0);
                        if (state0.total_cost < cost) {
                                printf(">>> error!! <<< in %d\n", __LINE__);
                                exit(1);
                        }
                        add_list(&state0);
                }
                else {
                        state_free(state0);
                }
                if (state1.t_name[0].fixed <= alphabet_size && state1.t_name[1].fixed <= alphabet_size) {
                        c_cost(&state1);
                        if (state1.total_cost < cost) {
                                printf(">>> error!! <<< in %d\n", __LINE__);
                                exit(1);
                        }
                        add_list(&state1);
                }
                else {
                        state_free(state1);
                }
        }
        return;
}

int
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);
                        }
                }
        }
        find_AC();
        printstate(&list[num_state - 1]);
        printf("[END]\n");
        return(0);
}
```