信州大学
大学院総合理工学研究科

修士論文

同時質問マスターマインドにおける最適な質問

指導教員　西新 幹彦 准教授

専攻　工学専攻
分野　電子情報システム学分野
学籍番号　20W2019J
氏名　小澤 泰雅

2022 年 3 月 7 日

目次

1 はじめに 1

2 マスターマインドのルール 1
2.1 本来のルール 2
2.2 ヒント 2
2.3 本来のマスターマインドからの変更点 3

3 従来研究の戦略と本研究の解答 3
3.1 従来研究における戦略 3
3.2 本研究における質問 4

4 コードの役割 5
4.1 コードによる C_0 の分割 5
4.2 コードの候補の削減 6

5 A^* アルゴリズム 9
5.1 A^* アルゴリズムの概要 9
5.2 本研究でのコードの個数の見積もり 10

6 プログラム構造と高速化 11
6.1 プログラム構造 11
6.2 高速化 12

7 結果と考察 18
7.1 エントロピーの定義 18
7.2 質問の分析 19

8 まとめ 22

謝辞 22

参考文献 22

付録 A ソースコード 24
A.1 最適な質問を探索するプログラム 24

1　はじめに

本論文ではマスターマインドという 2 人でプレイする対戦型推理ゲームの最適戦略について論じる．

このゲームは古くから存在し，Bulls and Cows や Hit and Blow，そしてマスターマインドなど様々な名前で呼ばれている．1970 年代全範囲イギリスのインヴィクタ社がマスターマインドという商品名で発売し，その後アメリカでハスブロ社から発売されるなど，世界中で発売された．また，ゲーム番組としても楽しまれてきた．1946 年からはアメリカで Twenty Questions，日本ではそれをモデルにした二十の扉というゲーム番組が放送されてた．ただし，これらは単に推理ゲームというだけで，マスターマインドとの類似点は解答者がヒントによって対象を絞り込んでいくという点である．マスターマインドとほぼ同じルールのゲーム番組としては，2011 年から 2014 年までフジテレビで放送されていた NumerOn[1] がある．

マスターマインドは，出題者と解答者と呼ばれる 2 人のプレイヤーによって行われる．まず準備として，出題者が秘密のコードと言うものを決め，解答者に知られないようにする．解答者は秘密のコードを推測して出題者に提示する．出題者は，解答者の推測が秘密のコードにどれだけ近いかヒントを与える．上の一組の問答を 1 手とし，その手続きを解答者が秘密のコードを当てるまで続ける．解答者にとってコードを当てるまでの手数は小さいほうが良い．

解答者の振る舞いをあらかじめ記述したものを戦略と呼ぶ．解答者が戦略を決めると，出題者が決めた秘密のコードごとにそれを当てるまでの手数が決まる．戦略の手数に関して様々な研究が行われてきている．1976 年に D.E.Knuth[2] は，最悪手数が 5 手の戦略を一つ示し，最悪手数は 5 手より短くならないことを示している．

本研究では，本来のマスターマインドのルールを変更し，一度に複数のコードを出題者に提示し，それぞれに対応したヒントをまとめて受け取るとしたとき，どのようなコードの組み合わせを提示すれば出題者の秘密のコードが判明するかについて考える．ここで，本来のルールでは秘密のコードを当てることを目的としていたので，出題者に提示するコードを「推測」と呼んでいたが，本研究では秘密のコードを判明することが目的であり当てる必要がないため，出題者に提示するコードの組み合わせを「質問」と呼ぶことにし，質問に含まれるコードの数はなるべく小さくしたい．本研究では，A*アルゴリズムに基づいて最適な質問を見つけるプログラムを作成し，本来のマスターマインドの最悪手数と比較，考察する．

2　マスターマインドのルール

この章ではまず本来のマスターマインドのルールを説明する．次に，出題者が解答者に返すヒントについて説明する．そのうえで，本来のマスターマインドとの変更点を説明し，以後の

議論に備える．

2.1 本来のルール

本来のマスターマインドのルールを説明する．ボードゲームでは 6 種類の色のピンを使うが，本論文では色の種類を 1 から 6 までの数字で表す．数字の重複を許して数字を 4 つ並べたものをコードと呼ぶ．したがってコードは全部で $6^4 = 1296$ 種類存在する．プレイヤーは出題者と解答者に分かれて以下の手順に従って対戦する．

1. 出題者は解答者に分からないように 6 種類の数字の中から重複を許して 4 つ選び並べる．出題者が選んだコードを秘密のコードと呼ぶ.
2. 解答者は秘密のコードを推測して 4 つ数字を並べ出題者に提示する．推測した 4 つの数字の並びを推測と呼ぶ.
3. 出題者は解答者の推測がどの程度秘密のコードに近いかをヒントというもので示す．ヒントの厳密な定義は次節で与えるが，ボードゲームの言葉で簡単に言うとヒントは次のように決められる．ヒントは黒ピンと白ピンの数によって示される．黒ピンの数は色と位置があっているピンの数で，白ピンの数は色だけがあっているピンの数である．
4. 示されたヒントが黒ピン 4 本のとき，つまり推測が秘密のコードと一致したとき，ゲームは終了する．このことを秘密のコードを正答すると言う事とする．正答しなければ手順 2 に戻って解答者は次の推測を作る．

2 から 3 までの手順を 1 手と数え，この手順を繰り返した回数を手数と言うこととする．プレイヤーは，出題者と解答者の役割を交代して行い，秘密のコードを正答するまでの手数の小さいプレイヤーの勝ちとする．そのため，プレイヤーは互いに，ゲームに勝つために秘密のコードを正答するまでの手数を小さくしたいと考える．

2.2 ヒント

秘密のコード $c = c_1c_2c_3c_4$ と推測 $q = q_1q_2q_3q_4$ に対して，両者の近さを表すヒントを $h(c, q)$ と書く．秘密のコードと推測は入れ替えても同じヒントが得られる．ヒントは 2 つの整数 r, w を用いて (r, w) という形式で表される．ゲームをプレイするとき，r, w はそれぞれ黒ピン，白ピンの数を表している．この r と w は次のように決まる．

1. $c_j = q_j$を満たす j の数を r とする．

2. $c_j = i$ を満たす j の数を $m_i, q_j = i$ を満たす j の数を n_iとし，w を

$$w = \sum_{i=1}^{6} \min(m_i, n_i) - r \tag{1}$$

と定義する．例えば，$h(1123, 2413) = h(2413, 1123) = (1, 2)$ である．実際にヒントとして取りうる値は

$$(r, w) = (0, 4), (0, 3), (0, 2), (0, 1), (0, 0), (1, 3), (1, 2), (1, 1), (1, 0),$$
$$(2, 2), (2, 1), (2, 0), (3, 0), (4, 0)$$

の 14 通りである．特に，ヒント $(4, 0)$ が出題者から提示された場合，解答者は秘密のコードに正答したということを表している．以降，これら 14 種類のヒントからなる集合を H と表す．

2.3 本来のマスターマインドからの変更点

本研究では，解答者の質問の提示を一回とし，コードを複数個提示できるとする．本来のマスターマインドであれば，提示した推測とそれに応じたヒントを元にして次に提示する推測の候補を絞れるが，本研究のルールではそれができないため，解答者は，どのような秘密のコードでも何であるか判明できるように，複数のコードを提示することが重要となる．また，前述のとおり，質問に含まれるコードの数はできるだけ小さくする．

3 従来研究の戦略と本研究の解答

この章では，従来研究における戦略と本研究における質問の違いを説明する．

3.1 従来研究における戦略

D.E.Knuth の研究における戦略は，本来のマスターマインドのルールに則り，解答者が提示した推測とそれに対応して出題者が提示したヒントから，次に解答者が提示すべき最適な推測を決定するという構造になっている．具体的な戦略の構造を以下に述べる．

解答者は出題者の提示するヒントを基に次の推測を決定する．この手順が記述されたものを戦略と呼ぶ．ヒントは前節で述べた 14 通りであり戦略は基本的に 14 分木の構造をしている．各内部ノードには秘密のコードの候補としてコードの集合が対応してる．特にルートノードにはコード全体が対応している．さらに秘密のコードの候補が複数ある内部ノードに対しては推測も記述されており，そのときそのノードは高々 14 個の子ノードを持ち，子ノードへの枝に

は 14 通りのヒントの内いずれかがラベルとして重複なく振られている．子ノードに対応しているコードの集合は，親ノードが持つコードのうち，推測に対するヒントが枝のラベルと一致するもの全体である．

解答者はこの戦略を使用することによって，出題者の秘密のコードを正答するのに最適な推測を提示することができる．まず，解答者はルートノードに記載されている推測を出題者に提示する．出題者からヒントが返ってきたら，そのヒントに対応した子ノードを参照し，次にその子ノードに記載されている推測を提示する．これを，出題者からヒント (4,0) が返ってくるまで繰り返す．

従来研究において，Knuth の論文 [2] で正答するまでの最悪手数の最小値が 5 手であることが分かっており，その戦略も記載されている．つまり，出題者の秘密のコードが何か判明するまでの最悪手数の最小値は 4 手となる．

3.2 本研究における質問

本研究における質問は，従来の戦略のような，解答者の推測とそれに返ってきたヒントから次の推測が定まるような構造ではなく，出題者がどのような秘密のコードを用意してもそれが何であるか判明できるよう，複数のコードを提示するものである．つまり，従来の戦略では解答者が提示する推測が動的に決定されるが，本研究の質問は提示するコードが静的に決定される．具体的な質問の決定方法を以下に述べる．

解答者は，秘密のコードの集合を出題者に提示するコードを用いて分割していくことで，質問を決定できる．秘密のコードの集合の分割に際し，次を定義する．

秘密のコードの集合を C_n とする．添え字 n は分割が行われた回数を示している．C_n において，コード q に対してヒント (r, w) となる秘密のコードの集合を

$$C_n(q, (r, w)) \tag{2}$$

と定義する．

これを用いて，秘密のコードの集合の分割方法を述べる．出題者に提示するコードを一つ決める．解答者は，そのコードを用いて C_0 を分割する．分割は，提示するコードに対して返されるヒントを用いて行う．秘密のコードの集合 C_0 に対し，出題者に提示するコードが q のとき，分割後の秘密のコードの集合 C_1 は，$C_1 = \{C_0(q, (r, w)) : (r, w) = (0, 4), (0, 3), \cdots, (4, 0)\}$ となる．ヒントに応じて分割が行われるため，C_0 は高々 14 個の部分集合に分割される．例えば，提示するコードを 1122 とすると，分割は図 1 のように行われる．次のコードを解答者に提示すると，C_1 内の高々 14 個の部分集合をそれぞれ，高々 14 個の部分集合に分割する．つまり，コードを 1 つ提示すると，高々 14 個に分割され，2 つ提示すると，高々 14^2 個に分割さ

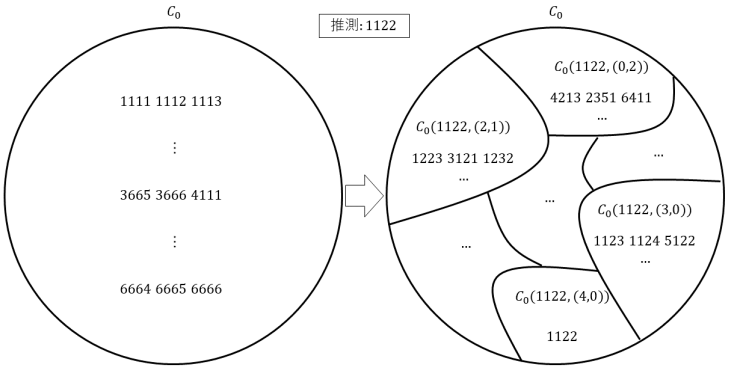


図 1: 1122 による C_0 の分割

れる．一般に，コードを n 個提示すると，C_0 は高々 14^n 個の部分集合に分割される．これを繰り返して，すべての部分集合の要素数が 1 以下になったとき，分割は終了する．これは，分割に使用したコードを全て提示すれば，出題者がどのような秘密のコードを用意しても，異なるヒントを返すコードの組み合わせを網羅したことを意味する．よって，分割に使用したコードが，解答者の提示すべき質問となる．

このように，出題者に提示する質問を，コードの数が最も少なくなるように探索するアルゴリズムを考える．

4 コードの役割

この章では，コードが C_0 を分割する際の特徴と，それを踏まえたコードの候補の絞り込みについて説明する．

4.1 コードによる C_0 の分割

前章でも述べたように，本研究ではいくつかのコードを同時に出題者に提示し，1296 個のコードの中から秘密のコードを特定することが目的である．ここで注意しなければいけないことが二つある．一つは，提示するコードによっては，秘密のコードの集合の分割のされ方が同じになるということである．表 1 は，要素数が 1296 個の C_0 を，1122 と 1133 の二つのコードを使ってそれぞれ分割したときの，各ヒントごとの要素数を表したものである．異なるコードを使用しても，分割後の各ヒントに対する要素数が変わらないため，この二つの分割は，実質的に同じであると言える．同様に，コード 1144 や，5566,3223 などでも実質的に同じ分割にな

表 1: 1122 と 1133 による C_0 分割後の要素数

	ヒント													
コード	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$	$(0,4)$	$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$	$(2,0)$	$(2,1)$	$(2,2)$	$(3,0)$	$(4,0)$
1122	256	256	96	16	1	256	208	36	0	114	32	4	20	1
1133	256	256	96	16	1	256	208	36	0	114	32	4	20	1

る．ここで注意したいことは，要素数は変わっていないが，要素は異なっている点である．例えば，ヒントが $(0,4)$ となる秘密のコードは，コード 1122 だと 2211, コード 1133 だと 3311 である．もう一つは，分割に使うコードの順番が変わっても，最終的な部分集合の要素は変わらないということである．図 2 と図 3 を見てほしい．例えば，$C_0 = \{1111, 1122, 2222, 4562\}$ というコードの集合に対し，コード $\{1111, 3332\}$ を提示したとき，C_0 の分割を 1111 で始めた時と，3332 で始めた時では，最終的な部分集合の要素は一致する．

4.2 コードの候補の削減

前節のことを考慮することで，C_0 を分割する際に使用するコードを次の様にして 1296 個から絞り込むことができる．絞り込まれたコードを，代表的な推測と呼ぶ.

実質的に同じ分割になるコードのうち，コードの数の並びを 4 桁の整数とみなしたときに最も値の小さいものを選択する．分割が一度もされていない状態から 1296 個のコードを実質的に同じ分割になるように分けると，5 つのグループに分かれる．5 つのグループの中から，4 桁の整数とみなしたときに，最も小さいコードは，1111,1112,1122,1123,1234 の 5 つとなる．解答者はこの代表的な推測から一つを選択し，C_0 を分割する．また，この 5 つのグループに，それぞれ① ～ ⑤ までの番号を割り当てる．分割が既に何度か行われている場合，代表的な推測は，既に分割に使用したコードを考慮して絞り込む必要がある．例えば，1122 を用いて既に分割している場合，1324 や 3124,3142 は実質的に同じ分割になる．

代表的な推測を絞り込むには，1 つのコードに対して，数字の置換と場所の置換を行い，元のコードが最も値が小さいかで行う．例えば，1133 というコードでは，$1122, 1144, \cdots, 3434, 3443, \cdots, 6655$ のように，1 を 2 に置換したり，1 と 3 の位置を入れ替えたりする．それらを全て列挙すると，その中に 1133 よりも小さい，1122 があるので，1133 は代表的な推測にはならないことが分かる．しかし，この数字と場所の置換には，数字の置換が 6! 回，場所の置換が 4! 回，コードの個数が n 個のとき，合計 $6!4!n!$ 回の置換をする必要があり，置換に時間がかかるので，任意のコードを代表的な推測に一度で変換できる仕組みを考える．

いくつかの例を用いて説明する．2343 を例にすると，このコードは，3 を 2 つ，2 と 4 を 1 つずつ使用したコードであり，数字と場所の置換を行うと，1123 という代表的な推測があ

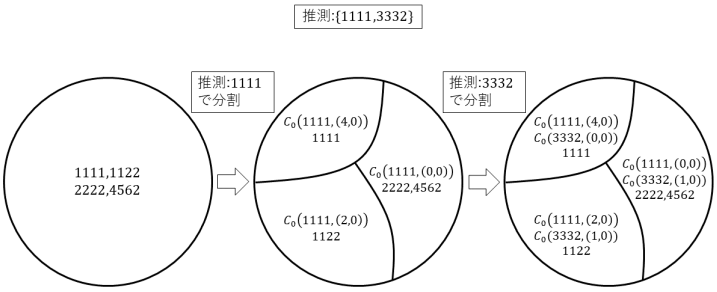


図 2: 1111,3332 の順で分割

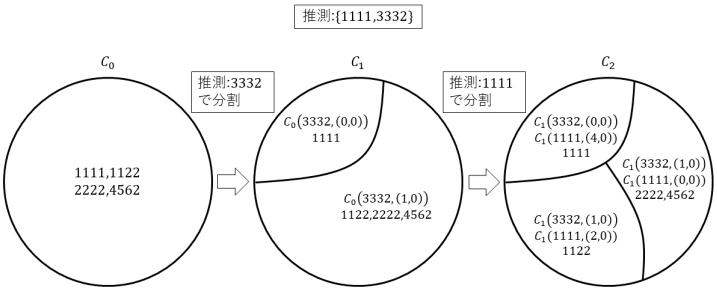


図 3: 3332,1111 の順で分割

ると考えられる．そこで，左から 1 番目の 2 と，右から 1 番目の 3 を入れ替え，3342 とした後，前から順に 1,2,3 と数字を置換すれば，1123 と変換できる．既に 1234 で分割された状態で，2222 というコードを例にすると，1234 と 2222 のコードは，左から 2 番目がどちらも 2 になっている．これを一番左になるように入れ替えると，2134 と 2222 となる．その後，2134 から，前から順に 1,2,3,4 と数字を置換する．2222 は，2134 の数字の置換で，2 を 1 に置換しているので，その置換を 2222 にも適用する．すると，1234 と 2222 が，1234 と 1111 と変換できる．この 1111 は，既に 1234 で分割された状態で，次に分割に使用する代表的な推測の候補の 1 つである．このように，使用されている数字の種類と個数を利用して，任意のコードを代表的な推測に変換できる．変換したコードを，標準形と呼ぶ．これにより，代表的な推

表 2: 数字と場所の置換と準標準形の絞りこみとの比較

	1111	1112	1122	1123	1234
6!4!n!	11	51	38	127	56
準標準形	11	51	38	127	66

測を短時間で絞り込むことができる．ただし，標準形に変換する方法はコードによりさまざまで，既に分割された状態で次に分割に使用するコードを絞り込む場合，非常に複雑な方法で変換するため，詳細は割愛する．また，標準形への変形は数字と場所の置換よりも不正確な絞り込みになる．表 2 は，コードが 2 つのとき，数字と場所の置換を行った際のコードの候補の数と，標準形に変形した際のコードの候補の数を比較したものである．分割を一度も行っていないときの代表的な推測は 5 つなので，この 5 つで C_0 の分割をそれぞれ行った後の，2 回目の分割に使用する代表的な推測の個数を示している．この表から，1 回目に 1234 を用いて分割を行った後の代表的な推測の個数が，数字と場所の置換をした場合と，標準形に変換した場合で異なっている．数字と場所の置換は代表的な推測を完全に絞り込めるが，標準形の変換には明確な方法がないため，不完全な絞り込みになってしまう．しかし，前述のように，数字と場所の置換は置換の回数が多いので，本研究では多少の重複を許して標準形を代表的な推測の絞り込みとして採用する．重複のある標準形を，準標準形と呼ぶことにする．

また，前節でも述べたように，複数のコードで分割する場合，そのコードが異なっていても，最終的な分割が実質的に同じ分割になる場合がある．前述した例を元にすると，1111 を用いて分割した後，1234 で分割する場合と，1234 を用いて分割した後，2222 を用いて分割する場合，後者のコードは，1234 と 1111 という標準形に変換することができるため，分割が同じになる．これは，標準形に変換する前のコードを用いて分割を行っても，実質的に同じ分割になる．つまり，1234 の次に分割に使用するコードの候補に 2222 があるが，この候補は削減できる．このことを活用して次のようにコードの候補を削減する．既に分割が行われている場合，既に分割に使用したコードと，次に分割に使用するコードの候補に，前述の① ～ ⑤の番号を振り分ける．その際に，既に分割に使用したコードに振り分けられた番号より若い番号が振り分けられたコードの候補は，実質的に同じ分割が存在するので，候補を削減できる．前述の例でも，1111 と 1234 と，1234 と 2222 に番号を振り分けると，前者は 1111①,1234⑤, 後者は 1234⑤,2222①となり，後者の 2222 は候補から削除できる．

以上のことを用いて，コードの候補を絞り込み，効率的に探索を行う．

5 A^* アルゴリズム

本研究における最適な解答を考えるために，A^* アルゴリズムという探索アルゴリズムをプログラム中で使用するため，その概要と探索に必要な要素を説明する．

5.1 A^* アルゴリズムの概要

A^* アルゴリズムは，Peter, Nils, and Bertram[6] によって提案されたグラフ探索アルゴリズムの一つである．「開始ノードから現在位置に至るまでの正確な距離」と「現在位置からゴールノードまでの推定距離」の 2 つの距離を用いて開始ノードからゴールノードまでの最短経路を探索する．このアルゴリズムは，グラフ探索において最短経路を効率的に探索するアルゴリズムである．

A^* アルゴリズムが実際にどのようなものか説明する．開始ノードからゴールノードまでの最短経路を探索する．開始ノードからゴールノードまでにいくつかのノードを経由するとし，経由するノード n にて距離を計算する．ここで，今最短経路上にあるノード n にいるとすると，最短経路は，「開始ノードからノード n までの最短経路」+「ノード n からゴールノードまでの最短経路」となる．開始ノードからノード n までの最短距離を $g(n)$, ノード n からゴールノードまでの最短距離を $h(n)$ とすると，ノード n を通る最短距離 $f(n)$ は

$$f(n) = g(n) + h(n) \tag{3}$$

と表せる．しかし実際には，開始ノードからノード n までの最短距離は計算できるが，ノード n からゴールノードまでの最短経路は探索が終わるまでは分からないので，$h(n)$ が計算できず，他の経路との距離の比較ができないので，推定値を使う必要がある．「ノード n からゴールノードまでの推定最短距離」を $h^*(n)$ とすると，ノード n を通る推定最短距離 $f^*(n)$ は

$$f^*(n) = g(n) + h^*(n) \tag{4}$$

と表せる．ノード n における $f^*(n)$ を計算し，その値が一番小さくなる経路を選択していくことで最短経路を求められる．

しかし，$h^*(n)$ の推定で見当違いな値を計算すると，最短経路を探索することが難しくなってしまう．A^* アルゴリズムが最適となるためには，$h^*(n)$ がもっともらしい値をとる必要がある．この $h^*(n)$ をヒューリスティック関数という．今，$h^*(n)$ がとるべき範囲を考えると，最小値は，ノード n がゴールノードだったとき，即ち $h^*(n) = 0$ のときである．また最大値は，ノード n からゴールノードまでの最短距離，即ち $h^*(n) = h(n)$ である．これは，A^* アルゴリズムは最短経路を探索するため，実際の最短経路よりも長い距離を見積もると最短経路

を効率的に探索できなくなるからである．以上から，ヒューリスティック関数がとるべき範囲は，

$$0 \leq h^*(n) \leq h(n) \tag{5}$$

となる．このように $h^*(n)$ を見積もることで，最短経路を見つけることができる．開始ノードから現在地までの距離は，探索したノードに到達するたびに計算し，これ以上探索出来ない場合や，推定距離が大きくなる経路にしか探索出来ない場合は，次に小さい推定距離を持つ経路を選択し，探索を続ける．ゴールノード G にたどり着いたとき，即ち $f(G) = g(G)$ となったとき，探索は終了し，それまでにたどってきた経路が最短経路となる．

5.2 本研究でのコードの個数の見積もり

本研究での A^* アルゴリズムにおける $g(n)$ および $h^*(n)$ について述べる．開始ノードからノード n までの最短距離 $g(n)$ を，選択したコードの数とし，ノード n からゴールノードまでの推定最短距離 $h^*(n)$ を，C_0 の部分集合の要素数のうち，最大なものを，14 を底とする対数を取るようにして，最短経路の推定距離を計算していく．本研究では，質問に含まれるコードの数が少なくなるように探索したいので，分割が終了するのに必要だと予想されるコードの数を見積もりとする．部分集合の要素数が一番多い部分集合に着目し，それに対し 1 つのコードで分割したときに，分割後の要素数が満遍なく分割したと考えると，分割後の要素数が全て 1 以下になるのに必要なコードの数は，要素数を 14 を底とする対数を取ることで見積もることができる．これを用いて，代表的な推測から最適なコードを選択する．

初めに提示するコードを例にして見ていく．表 3 は，初めに提示する代表的な推測の候補と，それを提示して C_0 を分割したときの部分集合の要素数の最大値，$h^*(n)$ の値,$f^*(n)$ の値をまとめたものである．$h^*(n)$ の最大値は，C_0 が分割されていない状態，即ち，$h(n) = \log_{14} 1296$ である．C_0 を分割し全ての部分集合の要素数が 1 以下になるようにしたいので，現在の部分集合のうち，要素数が最大なものが，分割に必要なコードを一番多く使うと予想される．一つ

表 3: 初めに提示するコード

コード	部分集合の最大値	$h^*(n)$	$f^*(n)$
1111	625	$\log_{14} 625$	$\log_{14} 625 \times 14$
1112	317	$\log_{14} 317$	$\log_{14} 317 \times 14$
1122	256	$\log_{14} 256$	$\log_{14} 256 \times 14$
1123	276	$\log_{14} 276$	$\log_{14} 276 \times 14$
1234	312	$\log_{14} 312$	$\log_{14} 312 \times 14$

のコードを使うと集合を高々 14 分割するため，$h^*(n)$ の値として妥当な見積もりとなる．表 3 において，$f^*(n)$ の値が最小なものを A*アルゴリズムでは探索するので，コード 1122 が選択され，次の代表的な推測を絞り込んでいく．

6 プログラム構造と高速化

前節で述べた A^* アルゴリズムを用いた探索行うためのプログラムを作成した．プログラム構造と実行のための高速化について説明する．

6.1 プログラム構造

コードの探索は木構造と A^* アルゴリズムを用いて行う．コードは木構造のノードに割り当てられており，選択した枝が経路，木の高さが手数を表している．ここで，木の根を高さ 0 とする．図 4 に，各ノードの構造を示す．上から見ていくと，1 段目には親ノードへのリンクがある．親ノードは，現在のコードで C_n を分割する前に $C_n - 1$ を分割するコードを示している．2 段目は現在のコードの値が格納されている．この値を用いて，C_n の分割や次の代表的な推測の絞り込みを行う．3 段目，4 段目には，それぞれ，$g(n), h^*(n)$ の値が格納されており，この値を用いて $f^*(n)$ の値を計算し，他の探索済みでないノードと比較しながら探索を行う．5 段目には，ノードの状態を表すステータスが格納される．状態とは，そのノードが探索されていない又は探索中の場合，Open，探索が終了している場合，Close が格納される．6 段目，7 段目には，現在のコードの次の代表的な推測へのリンクがある．次の代表的な推測の候補はほとんどの場合複数存在し，その候補数も可変なため，全てを子ノードとしてリンクさせるのは難しい．そのため，次のようにして二分木で表現する．代表的な推測の中から一つ選びそれを先頭子ノードとする．先頭子ノードは現在のコードの 6 段目に格納する．残りの代表的な推測から再び一つ選び，それを先頭子ノードの 7 段目に格納する．以後これを繰り返し，隣接子ノードの 7 段目にに次の代表的な推測を一つ，隣接子ノードとして格納する．各子ノードの親ノードは，現在のコードへのリンクを格納する．リンク先には，また同じ構造体が存在する．

プログラムがどのように動いていくかをより詳しく説明する．まず，木の根のノードを作成する．木の根には，親ノードへのリンクが存在せず，コードは NULL である．$g(n), h^*(n)$ の値は C_0 が分割されていないため，それぞれ，$0, \log_{14} 1296$ となる．状態は木の根が未探索のため Open となる．先頭，隣接子ノードへのリンクは，代表的な推測へのリンクが格納されるが，木の根を作成した時点では NULL である．

木の根を作成すると，代表的な推測が絞り込まれる．代表的な推測が判明すると，それらを各々用いて，C_0 の分割も行われる．初めに提示する代表的な推測は，前述したように 5 つある．この 5 つのコードごとにノードを作成するが，これは木の根に対する子ノードになるた

め，親ノードへのリンクは，木の根になる．コードには代表的な推測の値がそれぞれ格納され，$g(n)$ の値は，コードを一つ使用して C_0 を分割したため，1 を，$h^*(n)$ の値は，表 3 に示す値が格納される．状態は未探索のため Open となる．先頭子ノードへのリンクは，根を作成したときと同様に，NULL となる．また，子から親へのリンク付けがされるのと同時に，親から子へのリンク付けもされる．代表的な推測は，コードを 4 桁の整数とみなしたとき小さい順にソートされているため，木の根の先頭子ノードに 1111,1111 の隣接子ノードに 1112, 1112 の隣接子ノードに 1122, $\cdots$ といったように格納する．最後尾のノードの隣接子ノードは NULL となる．コードが格納されると，C_0 の分割も行われ，$f^*(n)$ も計算される．また，この時点では子ノードへのリンクが None となる．これにより，木の根の探索が終了するので，木の根の状態を Close に変更する．図 5 に，この状態の木を示す．

このプログラムでは，子ノードを持たず状態が Open であるノードの $f^*(n)$ の値を比較し，最も値の小さいノードを探索していくので，図 5 では，コード 1122 が選択される．コード 1122 を選択されると探索が始まり，次の代表的な推測の絞り込みをして，1122 の子ノードを生成する．絞り込みが完了するとノード 1122 の状態は Close となる．図 6 に，この状態の木を示す．1122 の子ノードの生成が完了すると，次に探索するノードを見つけるのだが，比較するノードは現在のノードと同じ高さ以外のノード，即ち，状態が Open になっている全てのノードで比較することに注意したい．図 6 では，次に探索するノードは，子ノードを持たず状態が Open の，高さ 1 のコード 1123 となる．これを繰り返して，分割された C_0 の各部分集合の要素数が全て 1 以下になったとき，探索は終了し，質問が確定する．質問は，探索の終了したノードから高さ 1 の祖先まで登っていくことで確認することができる．

6.2 高速化

探索アルゴリズムを考えプログラムを実装し実行したが，探索に時間がかかった．原因として，次に探索するノードを見つけることに時間がかかることが考えられる．A^* アルゴリズムは未探索のノードの中から，もっとも見通しの良いノードを見つけ，そのノードを探索するア

親ノードへのリンク
推測
$g(n)$の値
$h^*(n)$の値
状態
先頭子ノードへのリンク
隣接子ノードへのリンク

図 4: 各ノードの構造

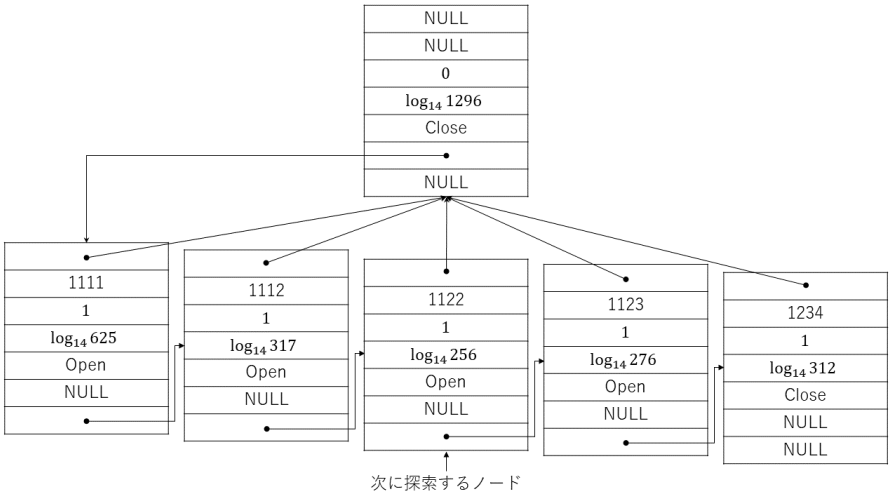


図 5: 最初のコードの候補

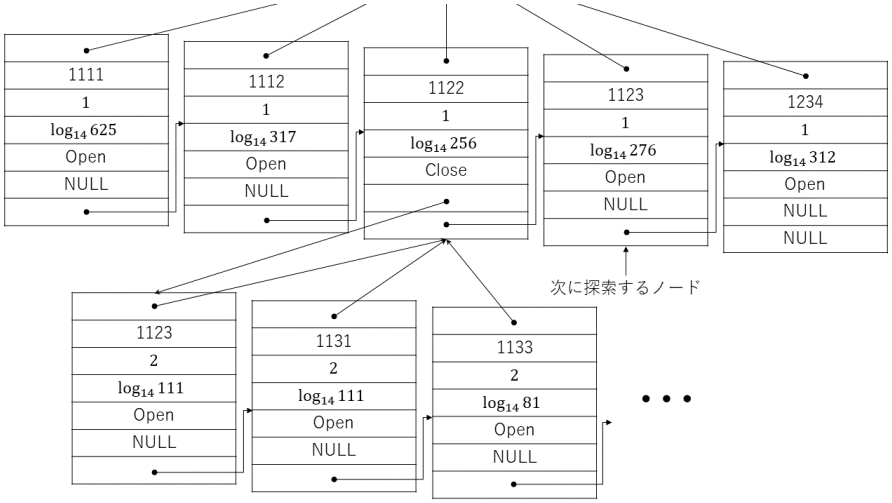


図 6: 次のコードの候補

ルゴリズムだが，本研究の問題の場合，探索が進められるにつれ未探索のノードが増える．見通しの良いノードは線形探索で見つけるため，未探索のノードが増えるほど，線形探索の時間が増えてしまう．これを改善するために，子ノードをヒープデータ構造に変更し，さらに A^* アルゴリズムを改良することで，実行時間を短くなるようにした．また，分割に必要なコードの数におおよその見通しをつけることで，A^* アルゴリズムで探索する高さに制限を設けた．

6.2.1 ヒープデータ構造

ヒープデータ構造とは，おおよそ完全二分木とみなすことができる配列オブジェクトである．木の最下位レベル以外の全てのレベルは完全に埋まっており，最下位レベルは左から順にある所まで埋まっている．ヒープを表現する配列は二つの属性を持つオブジェクトである．属性 $A.length$ は配列が含む要素数を表し，属性 $A.heap-size$ は配列 A に格納されているヒープの要素数を表す．ここで，$0 \leq A.heap-size \leq A.length$ である．木の根は $A[0]$ であり，節点の添え字 i から，以下のように，親，左の子，右の子を表す添え字を簡単に計算できる．

- 親の添え字：$\lfloor i-1 \rfloor$
- 左の子の添え字：$2i+1$
- 右の子の添え字：$2i+2$

二分木ヒープには，max ヒープと min ヒープの 2 種類がある．どちらも節点がヒープ条件を満たすが，条件はヒープによって異なる．max ヒープの場合，根以外の任意の節点 i が

$$A[Parent(i)] \geq A[i] \tag{6}$$

となる．ここで，$Parent(i)$ は添え字 i の親を表す添え字である．即ち，節点の値がその親の値以下であるという max ヒープ条件を満たすことが要請される．したがって，max ヒープは最大の要素を根に格納し，ある節点を根とする部分木が含む値はその節点自身の値を越えない．min ヒープの構成は逆である．根以外の任意の節点 i が min ヒープ条件，即ち

$$A[Parent(i)] \leq A[i] \tag{7}$$

を満たすことが要請される．min ヒープは最小の要素を根に格納する [8].

親ノードへのリンク
推測
$g(n)$の値
$h^*(n)$の値
子ノードの数
子ノードへのリンク

図 7: 改良後の各ノードの構造

6.2.2 ヒープデータ構造を用いた A^* アルゴリズムの改良

前節で述べたヒープデータ構造を用いて，A^* アルゴリズムによる探索を改良した．

前述のプログラム構造では，各ノードの子ノードが，二分木のように親ノードにリンクされていた．また，次に探索するノードは，Open になっている全てのノードを線形探索して選択していた．しかし，この線形探索が，実行時間の増加に影響していたため，子ノードが二分木になっていることを利用し，子ノードをヒープの配列となるよう変更した．

図 7 を見てほしい．これは，図 4 のノードの構造をヒープの配列で扱うために変更したものである．ヒープの配列では,min ヒープ条件を採用しており，min ヒープ条件は，ノードの $h^*(n)$ の値を用いて満たしている．これにより，子ノードの配列の根ノードを参照することで，その子ノードの中でもっとも見通しの良いノードを選択できるようになる．これにより，ノードの状態を気にする必要が無くなったため，状態の欄を削除している．また，子ノードは配列として親ノードにリンクさせるため，子ノードへのリンクを一つにまとめている．子ノードは配列なため，その要素数を 5 段目に格納して扱いやすくしている．

この新たな構造体を用いると，図 5 の木が図 8 のようになる．木の根の子ノードにヒープ構造の配列がリンクされており，ノードの $h^*(n)$ の値で min ヒープ条件を満たしている．配列には，代表的な推測の候補を 1 つずつ格納し，格納するごとに min ヒープ条件を満たすよう

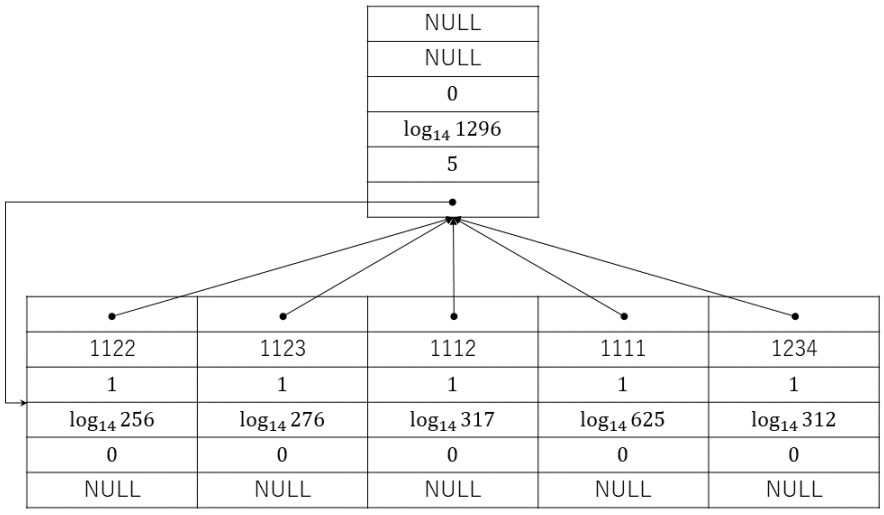


図 8: ヒープデータ構造を用いた最初のコードの候補

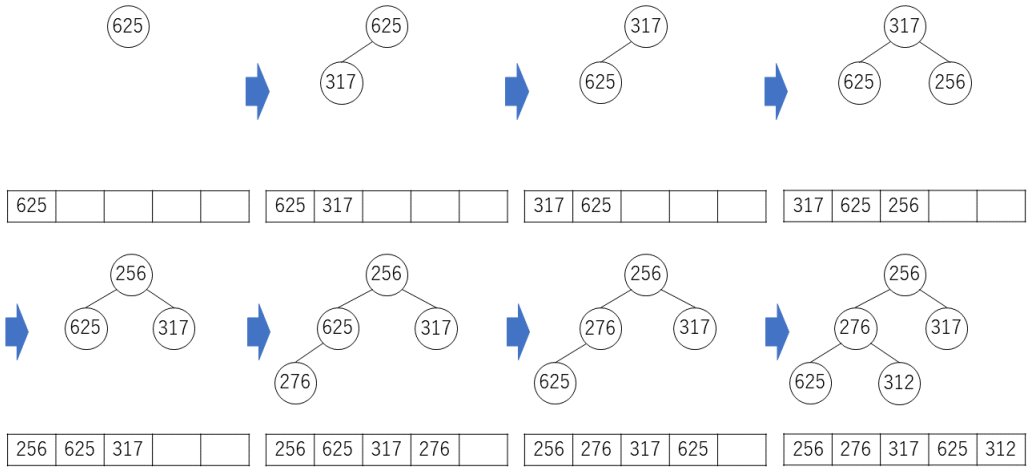


図 9: 最初のコードの候補の格納のようす

配列の要素の順番を入れ替える．最初のコードの候補を配列にひとつずつ格納するときのヒープ木と配列の並びを，図 9 に示す．ノードと配列に格納された値は便宜上 $h^*(n)$ の値になっているが，実際には図 7 が格納されている．前節で述べたように，min ヒープでは最小の要素が木の根，即ち配列の先頭に格納されるため，子ノードの配列の先頭を選択することで，見通しの良いコードを探索できる．

また，ヒープ構造の配列を活用し，A^* アルゴリズムも改良できる．図 10 は，図 8 により探索することが確定した，1122 のノードの先を探索した結果である．1122 の子ノードに，新たに配列がリンクされているが，これは 1122 で分割した次の代表的な推測の候補が格納された配列である．この配列も min ヒープ条件を満たしている．ここで，この配列の根ノードの $g(n), h^*(n)$ の値を，配列の親ノードである 1122 の $g(n), h^*(n)$ の値に書き換え，5 段目に配列の要素数を格納している．その後，最初の候補の配列を，min ヒープ条件を満たすよう並び替えている．このようにすることで，未探索全てのノードを比較して最も見通しの良いノードを決定する必要が無く，配列の先頭ノードが常に最適なノードとなる．よって，子ノードの配列が作成されたら，その親ノードの $g(n), h^*(n)$ の値を書き換え，min ヒープ条件を満たすよう並び替えることで，より高速に探索を行える．

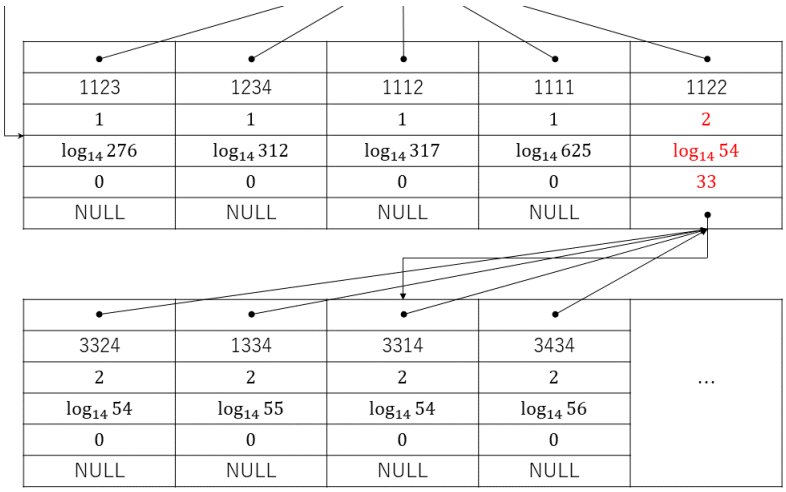


図 10: 最初の推測の候補の書き換え

6.2.3 分割に必要なコード数の見通し

A^* アルゴリズムは，$f^*(n)$ の値が最も小さいノードを探索するため，見積もりによっては無駄な高さまで探索を行っている可能性がある．そこで，A^* アルゴリズム以外の別の探索方法を用いて，分割に必要なコード数のおおよその見通しを求めた．

そもそも，出題者には，1296 個の全てのコードを提示すれば確実に出題者の秘密のコードが何であるか判明するが，その提示するコードの数ができるだけ小さくなるようにするのが本研究の目的である．最も少ないコード数を求められるのが A^* アルゴリズムであると考えられるが，別の探索方法でも，出来るだけ少ないコード数を求められるのではないかと考えた．これにより求めたコード数を A^* アルゴリズムに制限として与えることで，A^* アルゴリズムがそのコード数よりも多いコード数のノードを探索しないようにできると考えた．以下に別の探索方法と，それにより求まったコード数を述べる．

別の探索方法は，ノードの構造は A^* アルゴリズムとは変わらないが，一度探索した高さの他のノードは探索せず，常に h^* の値が最も小さいノードを探索する．つまり，深さ優先探索である．

深さ優先探索により，分割に必要なコードは，1122,1344,5135,2636,5441,3654,1112 の 7 つとなることが分かった．このコードの並びは，深さ優先探索で選択するコードの順に並んでおり，最後に 1112 を用いて C_6 を分割したところ，C_7 の全ての部分集合の要素数が 1 個以下

になったことが分かる．しかし，図 3 より，1112 は満遍なく分割することができないことが分かる．また，最初の分割で使用する 1122 と 1112 で分割することも，分割に偏りがあると考えられる．そこで，最初に分割に使用するコードを 1112 に固定したら，よりコード数を減らせるのではないかと考え，深さ優先探索を改善し，探索を行った．その結果，分割に必要なコードは，1112,3344,2455,6564,5313,2623 の 6 つとなった．よって，最悪でも 6 つのコードを質問として提示すれば，出題者の秘密のコードが何であるか判明することが分かったので，A^* アルゴリズムに，6 つよりも多いコード数になるノードを探索しないよう制限を設けることができた．

これらの変更を行ったアルゴリズムを用いることで, 最適な質問の探索を行うことができる．

7　結果と考察

前述のアルゴリズムを実装したが，最適な質問は現在探索中であるため，深さ優先探索によって探索した質問を分析した．この章では，質問の分析に使用する，エントロピーなどの一般的な定義を説明し，秘密のコードやヒントを確率変数としてエントロピー等を求めた結果を説明する．

7.1　エントロピーの定義

質問を分析するために，エントロピー，同時エントロピー及び条件付エントロピーを用いた．それらの定義を説明する．

エントロピーは，情報源 X において，n 個の情報が確率 $p_1, \cdots, p_n$ で起こるとすると，

$$H(X) = \sum_{i=1}^{n} p_i \log \frac{1}{p_i} \tag{8}$$

となる．

同時エントロピーは，情報源 X, Y において，この 2 つから同時に (x, y) の情報を得られる確率を $p(x, y)$ とすると，

$$H(X, Y) = \sum_{x,y} p(x, y) \log \frac{1}{p(x, y)} \tag{9}$$

となる．

条件付きエントロピーは，情報源 X, Y において，Y から情報 y を確率 $p(y)$ で得たとき，X から情報 x を得られる確率を $p(x|y)$ とすると，

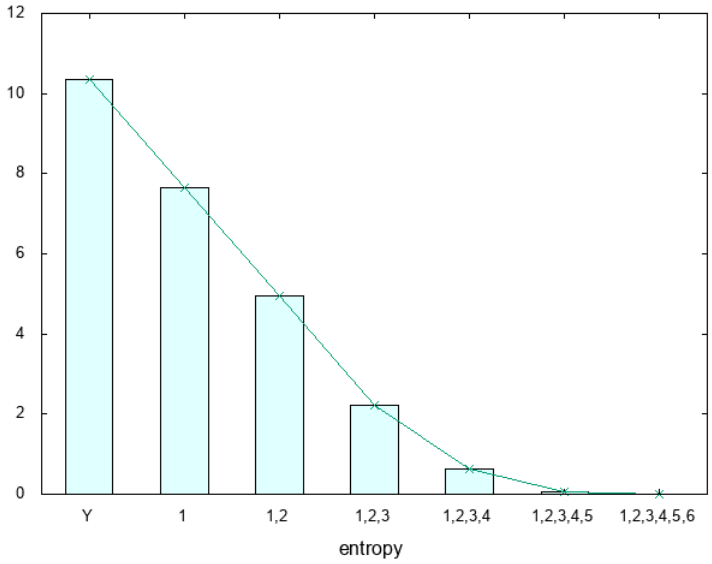


図 11: Y のエントロピーと分割後の Y の条件付きエントロピー

$$H(X|Y) = \sum_y p(y) \left(\sum_x p(x|y) \log \frac{1}{p(x|y)} \right) \tag{10}$$

となる．

エントロピー $H(Y)$ は，Y の曖昧さを示す量と言われている．また，条件付きエントロピー $H(Y|X)$ は X を観測したもとでの Y の曖昧さを表す．特に X を観測することで Y の値が確定するならば，$H(Y|X) = 0$ となる．

また，情報源 X, Y において，X と Y の依存度を表す尺度として，相互情報量がある．相互情報量は，エントロピーと条件付エントロピーを用いて

$$I(X;Y) = H(X) - H(X|Y) \tag{11}$$
$$= H(Y) - H(Y|X) \tag{12}$$

と表せる [9].

7.2 質問の分析

深さ優先探索で見つかった質問である，1112,3344,2455,6564,5313,2623 について，エントロピー等を求めた．このコードに番号を振り分けると，1112②,3344③,2455④,6564④,5313④,2623④

となり，確率変数 X_n を，1112 のヒントを X_1,3344 のヒントを $X_2, \cdots, 2623$ のヒントを X_6 とし，確率変数 Y を，秘密のコードとした．以下，グラフの横軸のメモリが数字のみの場合，その数字は情報源 X_n の添え字 n と対応している．

まず，Y のエントロピーと，$X_1, \cdots, X_n$ の順で分割をした時の Y の条件付きエントロピー，即ち $H(Y|X_1), \cdots, H(Y|X_1, X_2, X_3, X_4, X_5, X_6)$ を求めたものを図 11 に示す．一番左は，情報 X_n を得ていない，即ち，ヒントが何もないときのエントロピーを示している．グラフから分かるように，ヒントを得るごとにエントロピーが減少していき，6 つのヒントを得た時点でエントロピーが 0 になっている．また，折れ線グラフの傾きから，ヒントが少ないうちはエントロピーの減少量が大きいが，ヒントが多くなると，エントロピーの減少量が少なくなっていることが分かる．しかし，ヒントが多くなるとエントロピーが 0 に近づいており，ヒントが 5 個の時点でほぼ 0 になっていることから，6 個目のヒントは，部分集合を均等に分割することより，部分集合の要素数を 1 にするための分割と言える．

また，$H(Y)$ と $H(Y|X_1)$ の差について，(11),(12) 式より，

$$H(Y) - H(Y|X_1) = I(Y; X_1) \tag{13}$$
$$= H(X_1) - H(X_1|Y) \tag{14}$$

となる．ここで，$H(X_1|Y)$ は，秘密のコードの情報を得たときの X_1 の条件付きエントロピーであるため，$H(X_1|Y) = 0$ となる．よって

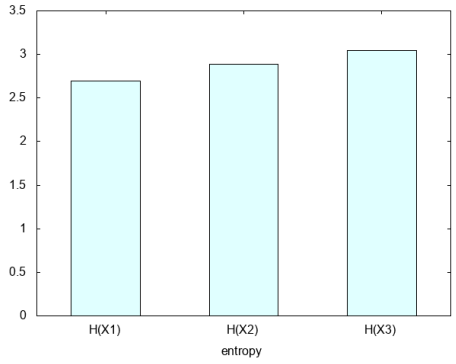
$$H(Y) - H(Y|X_1) = H(X_1) \tag{15}$$

となる．$H(Y)$ と $H(Y|X_1, X_2)$ についても同様に

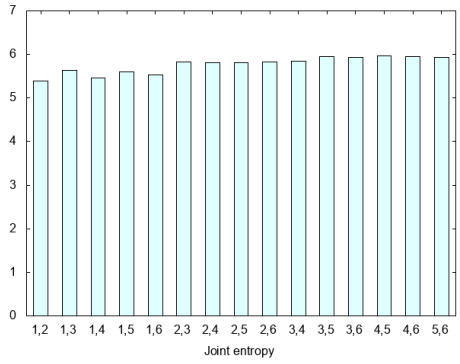
$$H(Y) - H(Y|X_1, X_2) = H(X_1, X_2) \tag{16}$$

となる．このことから，ヒントを得たときのエントロピーの差は，得られたヒントのエントロピーに関わってくる．

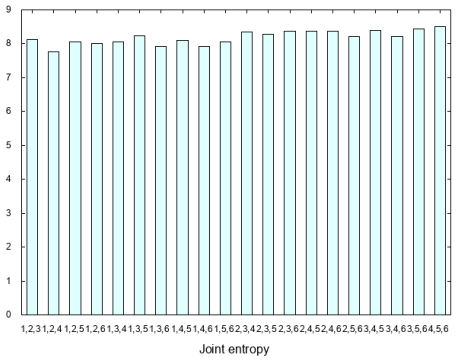
これにより，各ヒントのエントロピー，および各ヒントごとの同時エントロピーを求めた．図 12 に，その結果を示す．なお，図 12a は，本来 $H(X_1) \sim H(X_6)$ の 6 つが考えられるが，X_3, X_4, X_5, X_6 のコードはいずれも④のグループに属しているので，エントロピーが同じ値になるため，省略している．また，全てのヒントの同時エントロピーは，Y のエントロピーと等しくなるため，省略している．注目したい点は，(15),(16) 式より，エントロピーが大きいほど，条件付きエントロピーの値を小さくすることができる点である．このことから，図 12a を参照すると，$H(X_3)$ が最も値が大きいことが分かる．A^* アルゴリズムでは，コードが 1 つの



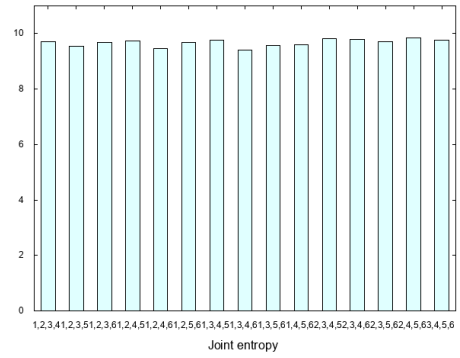
(a) ヒントが 1 つのとき



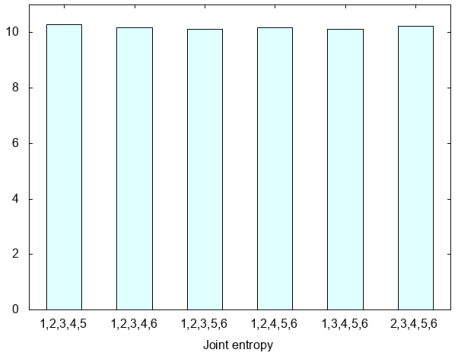
(b) ヒントが 2 つのとき



(c) ヒントが 3 つのとき



(d) ヒントが 4 つのとき



(e) ヒントが 5 つのとき

図 12: 各ヒントごとのエントロピー

とき，③のグループが，部分集合の要素数の最大値が最も小さいため，最初の探索は③のグループで行うが，エントロピーは④のグループの方が大きいため，エントロピーを見積もりとして探索することで，より高速な探索ができる可能性も有る．

コードが 2 つの場合，最もエントロピーが大きい組み合わせは，X_4 と X_5 のペアだった．コードを見ると，この 2 つは，単体でエントロピーが大きい④のグループに属する X_3, X_4, X_5, X_6 の中から 2 つ選んだとき，数字の種類と位置が最もばらけている組である．同時に提示するコードがばらけているほど，より満遍なく分割できると考えられる．他の組み合わせのエントロピーも，④のグループが含まれ，数字の種類と位置がばらけているほど，エントロピーが高くなる．エントロピーの観点から見ると，分割の効率の良いコードの組み合わせによって最適な質問を見つけられる可能性がある．

8 まとめ

本研究では，マスターマインドというゲームのルールを変更し，同時にコードを提示する場合に必要なコードを見つけるためのアルゴリズムを提案し，実装した．本研究では秘密のコードを効率良く分割し最適な質問を見つけたいため，優れた探索性能を持つ A^* アルゴリズムを採用した．探索に時間がかかるためより効率的に探索できるよう改善を続けてきたが，最適な質問を見つけることができなかったため，深さ優先探索にて見つけた質問を分析したところ，エントロピーに着目して探索方法を改善できる可能性がある．また，現在の探索時点で，4 つ以下のコード数で探索が終わっていない点と，深さ優先探索によって，6 つのコード数で探索が終了することから，最適なコード数は，5 つ又は 6 つであると考えられる．コードごとのばらつきに焦点を当て，より効率よく分割を行う方法を考案することで，探索にかかる時間を短縮できると期待できる．

謝辞

本研究にあたり，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]「ヌメロン-フジテレビ」，https://www.fujitv.co.jp/b_hp/numer0n/，2019 年 11 月閲覧．

[2] D. E. Knuth, “The Computer as Master Mind,” J.Recreational Mathematics, Vol.9(1), pp.1–6, 1976–77.

[3] 荒井航，「マスターマインドにおける最悪手数を最小化する戦略とそれに対する堅牢な出

題」，信州大学工学部卒業論文（指導教員：西新幹彦），2014 年 3 月．
[4] 荒井航，「マスターマインドにおける最適な戦略の数え上げに向けて」，信州大学大学院理工学系研究科修士論文（指導教員：西新幹彦），2015 年 3 月．
[5] 中村玲音，「マスターマインドにおける最適な戦略の再検証」，信州大学工学部卒業論文（指導教員：西新幹彦），2019 年 2 月．
[6] Peter E. Hart and Nils J. Nilsson and Bertram Raphael,“A Formal Basis for the Heuristic Determination of Minimum Cost Paths,”IEEE Transctions of Systems Science and Cybernetics 4,pp.100–107,1968.
[7] A*(A-star:エ ー ス タ ー) 探 索 ア ル ゴ リ ズ ム の 原 理 ，https://piyajk.com/archives/162, 2022 年 1 月閲覧．
[8] T. コルメン，C. ライザーソン，R. リベスト，C. シュタイン，世界標準 MIT 教科書 アルゴリズムイントロダクション 第 3 版，近代科学社，2012.
[9] Thomas M.Cover, Joy A.Thomas, 情報理論 基礎と広がり，共立出版株式会社，2013.

付録 A　ソースコード

A.1　最適な質問を探索するプログラム

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

#define CODELEN (4)
#define DIVPAT (14)
#define CODEKIND (6)
#define CODENUM (1296)

int Patlen; //Pattern の長さ
int cnt = 0; //探索の回数

//データの構造体
typedef struct p_tag{
    int data; //コード
    int estimate_g; //実績
    int estimate_h; //見積もり
    int childnum; //子の数
    struct p_tag *parent; //親
    struct p_tag *child; //子ども
}codedata;

//関数 div_code 用構造体
struct divid{
    int code;
    char hints[1296];
};

//cntlist 用構造体
struct cntlist{
    int cntlist1[7]; //1~6 が各行で何個使われているか
    int cntlist2[4]; //縦列で同じものが何個あるか
    int Codelist[4]; //コードを縦で見たもの
};

//新データの作成関数
codedata* makeNewNode(int code){
    codedata *pNewData;

    pNewData = (codedata *)malloc (sizeof(codedata));

    pNewData->data = code;
    pNewData->estimate_g = 0;
    pNewData->estimate_h = 0;
    pNewData->childnum = 0;
    pNewData->parent = NULL;
    pNewData->child = NULL;

    return pNewData;
}

//10 進数を 6 進数の並びに変換する関数
int dec_hex(int code){
    int num = 0;
    for (int j=0;j<CODELEN;j++){
        num = num + ((code % 6) + 1) * pow(10,j);
        code = code / 6;
    }
```

```
    return num;
}

//int から arr に変換する関数
void int_arr(int code,int pat[]){
    for (int i=0;i<CODELEN;i++){
        int num = code / pow(10,CODELEN-1-i);
        pat[i] = num;
        code = code - num * pow(10,CODELEN-1-i);
    }

}

//arr から int に変換する関数
int arr_int(int Pat[]){
    int code = 0;
    for (int i=0;i<CODELEN;i++){
        code = code + Pat[i] * pow(10,3-i);
    }
    return code;
}

//数を付け替える関数
void replace_num(int Save[][4],int lng){
    int numlist[6] = {0,0,0,0,0,0};
    int cnt = 1;

    for (int i=0;i<=lng;i++){
        for (int j=0;j<CODELEN;j++){
            if (numlist[Save[i][j]-1] == 0){
                numlist[Save[i][j]-1] = cnt;
                cnt ++;
            }
            Save[i][j] = numlist[Save[i][j]-1];
        }
    }
}

//比較関数 (1~5 までの種類わけ)
int countnum(int num){
    int code[4];
    int numlist[6] = {0,0,0,0,0,0};
    int Max = -1;
    int l,count = 0;
    int cnt = 3;
    int_arr(num,code);

    for (int i=0;i<CODELEN;i++){
        numlist[code[i]-1] = numlist[code[i]-1] + 1;
    }

    while (Max != 0){
        Max = -1;
        for (int i=0;i<CODEKIND;i++){
            if (numlist[i] > Max){
                Max = numlist[i];
                l = i;
            }
        }
        count = count + Max * pow(10,cnt);
        cnt --;
        numlist[l] = 0;
    }

    return count;

}

//コードの候補を分割する関数
```

```
void div_code(struct divid divlist[],int divplace){
    int Pat[4],Code[4];
    int code;
    int fourteenlist[21] = {0,1,2,3,4,5,6,7,8,-1,9,10,11,-1,-1,12,-1,-1,-1,-1,13};

    int_arr(divlist[divplace].code,Pat);

    for (int i=0;i<CODENUM;i++){

        code = dec_hex(i);
        int_arr(code,Code);

        int cc[4] = {};
        int ct[4] = {};
        int bh = 0;
        int wh = 0;
        for (int j=0;j<CODELEN;j++){
            if (Pat[j] == Code[j]){
                bh ++;
                cc[j] = 1;
                ct[j] = 1;

            }
        }
        for (int j=0;j<CODELEN;j++){
            if (cc[j]==1)
                continue;
            for(int k=0;k<CODELEN;k++){
                if (ct[k]==1)
                    continue;
                if (Code[j] == Pat[k]){
                    wh ++;
                    ct[k] = 1;
                    break;
                }
            }
        }

        int resp = bh * 5 + wh;

        if (resp > 24 || resp < 0){
            resp = 24;
        }

        divlist[divplace].hints[i] = fourteenlist[resp];
    }
}

//最悪値の計算
double culc_est(struct divid divlist[],int divplace){
    int Max = 0;
    int cntest[2][1296];
    int cnt = 0;
    for (int i=0;i<CODENUM;i++){
        int est = 0;
        for (int j=0;j<=divplace;j++){
            est = est * 14 + divlist[j].hints[i];
        }

        int flag = 0;
        int place = -1;

        for (int j=0;j<cnt;j++){
            if (cntest[1][j] == est){
                flag = 1;
                place = j;
            }
        }
        if (flag == 0){
```

```
            cntest[1][cnt] = est;
            cntest[0][cnt] = 1;
            cnt ++;
        }
        else{
            cntest[0][place] ++;
        }
    }

    //部分集合の最大値
    for (int i=0;i<cnt;i++){
        if (cntest[0][i] > Max){
            Max = cntest[0][i];
        }
    }

    return Max;
}

//子の並び替え
void Min_Heapify(codedata *code,int i,int Patlen){
    int l = i * 2;
    int r = i * 2 + 1;
    int largest = i;
    if ((l < Patlen) && (pow(14,code[l-1].estimate_g) * code[l-1].estimate_h < pow(14,code[i-1].estimate_g)
    * code[i-1].estimate_h)){
        largest = l;
    }
    if ((r < Patlen) && (pow(14,code[r-1].estimate_g) * code[r-1].estimate_h < pow(14,code[largest-1].estimate_g)
    * code[largest-1].estimate_h)){
        largest = r;
    }
    if (largest != i){
        codedata pChan = code[i-1];
        code[i-1] = code[largest-1];
        code[largest-1] = pChan;

        Min_Heapify(code,largest,Patlen);
    }
}

//見積もりの最大値を与える関数
void dataest(codedata *code,codedata *pTop,int Pattern[]){
    int Patlen = code->childnum;
    codedata *clist;
    codedata *pNow;
    clist = (codedata*)malloc(sizeof(codedata) * Patlen);

    if (clist == NULL){
        while (pTop->child != NULL){
            pTop =pTop->child;
            printf("%d\n",pTop->data);
        }
        printf("%s\n","error");
        exit(1);
    }


    for (int i=0;i<Patlen;i++){
        int divplace = 0;
        int cnt;
        struct divid divlist[7];
        codedata *pNew = code;

        divlist[divplace].code = Pattern[i];
        div_code(divlist,divplace);
        divplace ++;
        while (pNew != pTop){
            divlist[divplace].code = pNew -> data;
```

```
            div_code(divlist,divplace);
            divplace ++;
            pNew = pNew -> parent;
        }
        divplace --;

        cnt = culc_est(divlist,divplace);

        clist[i].data = Pattern[i];
        clist[i].estimate_g = divplace + 1;
        clist[i].estimate_h = cnt;
        clist[i].parent = code;
        clist[i].child = NULL;
    }

    code->child = clist;
    pNow = code->child;

    if (Patlen == 0){
        code->estimate_g = 7;
        code->estimate_h = 1;
    }
    else{
        for (int i=(Patlen)/2;i>=1;i--){
            Min_Heapify(pNow,i,Patlen+1);
        }
        code->estimate_g = pNow->estimate_g;
        code->estimate_h = pNow->estimate_h;
    }

    while (code != pTop){
        code = code -> parent;
        pNow = code->child;
        Patlen = code->childnum;
        for (int i=(Patlen)/2;i>=1;i--){
            Min_Heapify(pNow,i,Patlen+1);
        }
        code->estimate_g = pNow->estimate_g;
        code->estimate_h = pNow->estimate_h;
    }

}

//1 つのコードの並びを入れ替える関数
void changeone(int Code[],int i, int j){
    int save;
    save = Code[i];
    Code[i] = Code[j];
    Code[j] = save;
}

//コードの数を付け変える関数
int numchange(int Code[4]){
    int numlist[6] = {0,0,0,0,0,0};
    int cnt = 1;
    int nomal = 0;
    for (int i=0;i<CODELEN;i++){
        if (numlist[Code[i]-1] == 0){
            numlist[Code[i]-1] = cnt;
            Code[i] = numlist[Code[i]-1];
            cnt ++;
        }
        else{
            Code[i] = numlist[Code[i]-1];
        }
    }
    for (int i=0;i<CODELEN;i++){
        nomal = nomal + Code[i] * pow(10,CODELEN-1-i);
    }
```

```
    return nomal;
}

//1 桁の標準形を作成する関数（場所入れ替え）
int changenomal(int Code[4]){
    int i = -1;
    int j = 0;
    int nomal;
    while (i < CODELEN-1){
        int k = j + 1;
        while (k < CODELEN){
            if (Code[j] == Code[k]){
                if (i == j){
                    changeone(Code,i+1,k);
                    i ++;
                }
                else{
                    changeone(Code,i+1,j);
                    changeone(Code,i+2,k);
                    i = i + 2;
                }
                j = i;
                k = j + 1;
            }
            k ++;
        }
        j ++;

        if (j >= 3){
            i ++;
            j = i;
        }
    }
    nomal = numchange(Code);

    return nomal;

}

void nomal_change(int Save[][4],int i,int j,int L){
    int save[L+1];

    for (int k=0;k<L;k++){
        save[k] = Save[k][i];
    }
    for (int k=0;k<L;k++){
        Save[k][i] = Save[k][j];
        Save[k][j] = save[k];
    }
}

int changeflag(int Save[][4],struct cntlist *cntlist,int i,int j,int k,int lng){
    int flag;
    if (cntlist->cntlist1[Save[k][i]] < cntlist->cntlist1[Save[k][j]]){
        flag = 0;
    }
    else if (cntlist->cntlist1[Save[k][i]] > cntlist->cntlist1[Save[k][j]]){
        flag = 1;
    }
    else{
        if (k < lng){
            flag = changeflag(Save,cntlist,i,j,k+1,lng);
        }
        else{
            flag = 1;
        }
    }
    return flag;
```

```
}

//2 個で Ncntlist 作成
void Two_makeNcntlist(int d,int P,int st,int num,int Ncntlist[4]){
    struct cntlist cntlist;
    int data[4],Pat[4];

    int_arr(d,data);
    int_arr(P,Pat);

    for (int i=st;i<num;i++){
        cntlist.Codelist[i] = data[i] * 10 + Pat[i];
    }

    for (int i=0;i<7;i++){
        cntlist.cntlist1[i] = 0;
    }

    for (int i=st;i<num;i++){
        int cnt = 0;
        cntlist.cntlist1[data[i]] += 10;
        cntlist.cntlist1[Pat[i]] += 1;

        for (int j=st;j<num;j++){
            if (cntlist.Codelist[i] == cntlist.Codelist[j]){
                cnt ++;
            }
        }

        cntlist.cntlist2[i] = cnt;
    }

    for (int i=st;i<num;i++){

        Ncntlist[i] = cntlist.cntlist2[i] * 10000 + cntlist.cntlist1[data[i]] * 100 + cntlist.cntlist1[Pat[i]];
    }
}

//3 個以上で Ncntlist 作成
void N_makeNcntlist(int Save[7][4],int codelist[7],int st,int num,struct cntlist *cntlist,int lng){

    for (int i=0;i<4;i++){
        cntlist->Codelist[i] = 0;
    }
    for (int i=0;i<7;i++){
        cntlist->cntlist1[i] = 0;
    }

    for (int i=st;i<num;i++){
        for (int j=0;j<=lng;j++){
            cntlist->Codelist[i] = cntlist->Codelist[i] + Save[j][i] * pow(10,lng-j);
        }
    }

    for (int i=st;i<num;i++){
        int cnt = 0;
        for (int j=0;j<=lng;j++){
            cntlist->cntlist1[Save[j][i]] = cntlist->cntlist1[Save[j][i]] + pow(10,lng-j);
        }
        for (int j=st;j<num;j++){
            if (cntlist->Codelist[i] == cntlist->Codelist[j]){
                cnt ++;
            }
        }

        cntlist->cntlist2[i] = cnt;
    }

    return;
```

```
}

//2 個のコードの入れ替え関数
void Two_Patterncnt(int Save[2][4],int st,int num,int Ncntlist[4]){
    int i,j,cnt;

    //並び替え（数の大きい順）
    for (int i=st;i<num-1;i++){
        for (int j=i+1;j<num;j++){
            if (Ncntlist[i] < Ncntlist[j]){
                changeone(Ncntlist,i,j);
                nomal_change(Save,i,j,2);
            }
        }

    }

    //数付け替え
    replace_num(Save,1);

    i = st + 1;
    cnt = 0;

    //上段、一番前と同じ数を前に持っていく
    while (i < num){
        if (Save[0][cnt] == Save[0][i]){
            cnt ++;
            nomal_change(Save,cnt,i,2);
        }
        i ++;
    }

    //前に持って行った分だけ、下段ソート
    i = st;
    while (i < cnt){
        j = i + 1;
        while (j < cnt + 1){
            if (Save[1][i] > Save[1][j]){
                nomal_change(Save,i,j,2);
            }
            j ++;
        }
        i ++;
    }

    if (cnt < 2){
        i = cnt + 1;
        while (i < CODELEN-1){
            j = i + 1;
            while (j < CODELEN){
                if (Save[1][i] > Save[1][j]){
                    nomal_change(Save,i,j,2);
                }
                j ++;
            }
            i ++;
        }
    }

    //数付け替え
    replace_num(Save,1);
}

//3 個以上のコードの入れ替え関数
void N_Patterncnt(int Save[7][4],int st,int num,struct cntlist *cntlist,int lng){
    int i,j,cnt;

    cnt = -1; //確定位置の記憶
```

```
    i = 0; //固定してみる方
    j = 0; //動かしてみる方

    //並び替え（同じ縦列を前へ）
    while (i < 3){
        j = i + 1;
        while (j < 4){
            if (cntlist->cntlist2[i] < cntlist->cntlist2[j]){
                changeone(cntlist->cntlist2,i,j);
                nomal_change(Save,i,j,lng+1);
                cnt = i;
            }
            j ++;
        }
        i ++;
    }

    //並び替え（数の大きい順）
    if (cnt == -1){
        int flag;
        i  = 0;
        while (i < 3){
            j = i + 1;
            while (j < 4){
                flag = changeflag(Save,cntlist,i,j,0,lng);
                if (flag == 0){
                    nomal_change(Save,i,j,lng+1);
                }
                j ++;
            }
            i ++;
        }
    }

    //数付け替え
    replace_num(Save,lng);
}

int Two_changenomal(int codelist[7],int replist[1296]){
    int cnti,cntj,cnt;
    int data = codelist[0];
    int replen = 0;
    int temprep[1296][2];

    for (int i=0;i<CODENUM;i++){
        int Pat = dec_hex(i);

        if (data < Pat){
            int Ncntlist[4] = {0,0,0,0};
            int Save[2][4]; //2 つの標準形を格納する配列
            int Saveint[2]; //Save を数字に変換した者
            int d[4],P[4];

            int_arr(data,d);
            int_arr(Pat,P);

            for (int i=0;i<CODELEN;i++){
                Save[0][i] = d[i];
                Save[1][i] = P[i];
            }

            Two_makeNcntlist(data,Pat,0,4,Ncntlist);

            Two_Patterncnt(Save,0,4,Ncntlist);

            cnti = 0;
            cnt = 0;
            while (cnti < CODELEN-1){
                cntj = cnti + 1;
```

```
                while (cntj < CODELEN){
                    if (Ncntlist[cnti] == Ncntlist[cntj]){
                        cnt ++;
                    }
                    cntj ++;
                }
                if (cnti != cnt){
                    int Newd,NewP;
                    int NewNcntlist[4] = {0,0,0,0};
                    Newd = arr_int(Save[0]);
                    NewP = arr_int(Save[1]);

                    Two_makeNcntlist(Newd,NewP,cnti,cnt+1,NewNcntlist);

                    Two_Patterncnt(Save,cnti,cnt+1,NewNcntlist);
                }
                cnt ++;
                cnti = cnt;
            }

            for (int i=0;i<2;i++){
                int code;
                code = arr_int(Save[i]);
                Saveint[i] = code;
            }

            int flag = 0;

            for (int i=0;i<replen;i++){

                if ((Saveint[0] == temprep[i][0]) && (Saveint[1] == temprep[i][1])){
                    flag = 1;
                }
            }

            if (flag == 0){
                for (int i=0;i<2;i++){
                    temprep[replen][i] = Saveint[i];
                }

                replen++;
            }
        }
    }
    for (int i=0;i<replen;i++){
        replist[i] = temprep[i][1];
    }

    return replen;

}


int N_changenomal(int codelist[7],int replist[1296],int lng){
    int data = codelist[lng-1];
    int replen = 0;
    int temprep[1296][lng+1];
    int cnt = 0;

    for (int i=0;i<CODENUM;i++){
        int Pat = dec_hex(i);
        cnt ++;
        if (data < Pat){
            struct cntlist cntlist;
            int Save[7][4]; //2 つの標準形を格納する配列
            int Saveint[7]; //Save を数字に変換したもの

            codelist[lng] = Pat;
```

```
            for (int j=0;j<=lng;j++){
                int d[5];
                int_arr(codelist[j],d);
                for (int k=0;k<CODELEN;k++){
                    Save[j][k] = d[k];
                }
            }

            N_makeNcntlist(Save,codelist,0,4,&cntlist,lng);

            N_Patterncnt(Save,0,4,&cntlist,lng);

            for (int i=0;i<=lng;i++){
                int code;
                code = arr_int(Save[i]);
                Saveint[i] = code;
            }

            int flag = 0;
            int cnt = 0;
            int i = 0;

            while (i < replen){
                flag = 0;
                cnt = 0;
                int j = 0;
                while (j <= lng){
                    if (Saveint[j] == temprep[i][j]){
                        flag ++;
                    }
                    cnt ++;
                    j ++;
                }
                if (flag == cnt){
                    break;
                }
                i ++;

            }
            if (flag != cnt){
                flag = 0;
            }
            if (flag == 0){
                for (int i=0;i<=lng;i++){
                    temprep[replen][i] = Saveint[i];

                }
                replen++;

            }

        }

    }

    for (int i=0;i<replen;i++){
        replist[i] = temprep[i][lng];
    }
    return replen;
}


int main(void){

    int Pattern[1296]; //標準形を格納する配列

    //ポインタ変数として先頭を定義
    codedata *pTop = NULL;
    //新データ作成用と現在位置用ポインタ
```

```
codedata *pNew,*pNow;

//根の作成
pNew = makeNewNode(pow(DIVPAT,CODELEN+1));
pNew->estimate_g = CODELEN+1;
pNew->estimate_h = CODENUM;
pTop = pNew;
pNow = pTop;

Patlen = 0;
//1 段目の作成
for (int i=0;i<CODENUM;i++){
    int nomal; //標準形
    int flag = 0;
    int code = dec_hex(i); //推測の候補 (int)
    int Code[4]; //推測の候補 (arr)

    int_arr(code,Code);

    nomal = changenomal(Code);

    for(int i=0;i<Patlen;i++){
        if (Pattern[i] == nomal){
            flag = 1;
        }
    }

    if (flag == 0){
        Pattern[Patlen] = nomal;
        Patlen ++;
    }

}

pTop->childnum = Patlen;

/*子づくり、見積もりを与える*/
dataest(pTop,pTop,Pattern);

pNow = pTop;

int a = 0;
double time = 60;
double cpu_time;

while (1){
    int codelist[7]; //葉から根までの候補を格納する関数
    int replen; //replist の長さ
    int replist[1296];//標準形を格納する配列
    int lng = 0;
    pNow = pTop->child;

    if (pNow -> estimate_h == 1){
        cnt ++;

        printf("%s\n","END");

        while (pNow->child != NULL){
            printf("%d\n",pNow->data);
            pNow = pNow->child;
        }
        printf("%d\n",pNow->data);
        break;
    }

    cpu_time = clock() / CLOCKS_PER_SEC;

    if (cpu_time > time){
        printf("%d%s\n",a,"個終了");
```

```
            printf("%s%lf%s\n","CPU 時間:",cpu_time,"秒");
            while (pNow->child != NULL){
                printf("%d%s",pNow->data,",");
                pNow = pNow->child;
            }
            printf("%d\n",pNow->data);
            printf("\n");
            time = time + 3600;
        }

        while(pNow->child != NULL){
            codelist[lng] = pNow->data;
            lng ++;
            pNow = pNow->child;
        }

        codelist[lng] = pNow->data;
        lng ++;

        //推測 1 個なら Two,2 個以上なら N
        if (lng == 1){
            replen = Two_changenomal(codelist,replist);
        }
        else{
            replen = N_changenomal(codelist,replist,lng);
        }

        //子を挿入
        int insertlist[1296];
        int insertcnt = 0;
        for (int i=0;i<replen;i++){
            if (countnum(replist[i]) <= countnum(pNow->data)){
                insertlist[insertcnt] = replist[i];
                insertcnt ++;
            }

        }

        pNow->childnum = insertcnt;

        //見積もり計算
        dataest(pNow,pTop,insertlist);

        a++;

    }

    return (0);

}
```