信州大学工学部

学士論文

推測を複数提示するマスターマインドにおける最適な推測集合を探索するアルゴリズムについて

指導教員　西新 幹彦 准教授

学科　電子情報システム工学科
学籍番号　16T2040D
氏名　小澤 泰雅

2019 年 3 月 12 日

目次

1 はじめに 1

2 マスターマインドのルール 2
2.1 本来のルール 2
2.2 ヒント 2
2.3 1 回の提示で秘密のコードを特定する 3

3 テストパターンの役割 3
3.1 本来のルールにおける戦略とテストパターン 3
3.2 テストパターンによるコードの分割 4
3.3 実質的に同じテストパターン 5

4 A*アルゴリズムを用いた探索とプログラム構造 7
4.1 A*アルゴリズムの概要 7
4.2 A*アルゴリズムのための推測数の見積もり 8
4.3 データ構造 9

5 問題点 12

6 まとめ 12

謝辞 13

参考文献 13

付録 A ソースコード 14
A.1 提示する複数の推測を探索するプログラム 14

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．戦略を決めると，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．ここで，解答者は 4 手目が終了した時点で出題者の秘密のコードが何か判明していることに注意する．

マスターマインドは，解答者が秘密のコードを当てるまで推測の提示をし，出題者はそれが秘密のコードか否かを解答者に提示することを繰り返すため，秘密のコードによって当てるまでの数が異なる．これは情報理論における可変長符号化の考え方に類似している．そこで本研究では，本来のマスターマインドのルールを変更し，一度に複数の推測を出題者に提示し，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，どのような推測を提示すれば出題者の秘密のコードが判明できるかについて考える．このようにルールを変更することによって，可変長符号としてのマスターマインドを固定長符号として捉え直す．本来のルールでは 4 手目が終了した時点で出題者の秘密のコードが何か判明しているので，変更後のルールでは 4 つの推測で秘密のコードを特定できるのかが大きな関心事となる．本研究では，A*アルゴリズムに基づいて推測の数が最小となるような，最適な推測の組み合わせを見つけるアルゴリズムを作成し，本来のマスターマインドの最悪手数と比較，考察することで，情報理論的な関係性を見出すことを目的とする．本論文では，探索アルゴリズムの提案を行う．

2　マスターマインドのルール

この章ではまず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，本研究における変更点を説明し，以後の議論に備える．

2.1　本来のルール

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．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 つ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の配置を推測して 4 つの数字を並べ出題者に提示する．推測した 4 つの並びを推測又はテストパターンと言うこととする。
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．黒ピンの数は数字と位置があっているピンの数で，白ピンの数は数字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．手数が少ないほど，良い推測を提示したことになる．

2.2　ヒント

秘密のコード $c_1c_2c_3c_4$ と推測 $q_1q_2q_3q_4$ に対して，両者の近さを表すヒントを $h(c_1c_2c_3c_4, q_1q_2q_3q_4)$ と書く．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．また，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．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．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　1 回の提示で秘密のコードを特定する

本研究では，解答者の推測の提示を一回とし，推測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基にして次に提示するテストパターンを決定できるが，本研究ではそれができないため，一回の提示でどのような組み合わせの推測を提示するかが重要となる．

このようにルールを変更し，秘密のコードが判明するのに必要な推測の組み合わせを探索する．

3　テストパターンの役割

この章では，従来研究と本研究における戦略の違いを説明する．

3.1　本来のルールにおける戦略とテストパターン

解答者は出題者の提示するヒントを基に次のテストパターンを決定する．この手順が記述されたものを戦略と呼ぶ．本来のルールにおける解答者の戦略は，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．

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いる．特にルートノードは何の手掛かりもない状態に対応していることから，コード全体が候補として対応している．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テストパターンも記述されており，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，テストパターン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必要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テストパターン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テストパターンを提示する．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．

3.2 テストパターンによるコードの分割

本研究は，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を正答できるような推測の組み合わせを探索するものである．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，本研究の戦略では提示する推測は動的ではない．そのため，実際のところ本研究には戦略的要素はない．具体的な推測の組み合わせの探索方法を以下に述べる．

解答者は，出題者の秘密のコードが何であるかを知るために，出題者の用意しうる秘密のコードの集合を分割したいと考える．ある推測を提示したときに，その推測に対応したヒントごとに，その集合を分割していくことを考える．ここで，秘密のコードの候補の集合 C の要素において，推測 $q_1q_2q_3q_4$ に対してヒント $a \in (r, w)$ を返すコードの集合

$$C(q_1q_2q_3q_4, a) \triangleq \{c \in C \mid h(c, q_1q_2q_3q_4) = a\} \tag{2}$$

を定義する．これにより，C は推測 $q_1q_2q_3q_4$ によって

$$C = \bigcup_{a \in (r,w)} C(q_1q_2q_3q_4, a) \tag{3}$$

と分割されることになる．1296 個のコードの集合を C_0 とし，式 (2) を用いて C_0 を分割すると，C_0 は高々 14 個の部分集合に分割される．例えば，C_0 に対し，推測 1122 を用いて分割すると，C_0 は図 1 のように分割される．次の推測を用いて C_0 を分割すると，C_0 内の高々 14 個の部分集合をそれぞれ，さらに高々 14 個の部分集合に分割する．つまり，推測を 1 つ用いて分割すると高々 14 個，2 つ用いて分割すると高々 14^2 個に分割される．一般に n 個の推測を用いて C_0 を分割すると，C_0 は高々 14^n 個の部分集合に分割される．C_0 を n 個の推測を用いて分割したものを C_n と表すことにする．部分集合は，秘密のコードが各推測に応じたヒントごとに分かれている．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，分割は終了し，今までに提示した推測が，解答者の提示すべき推測となる．

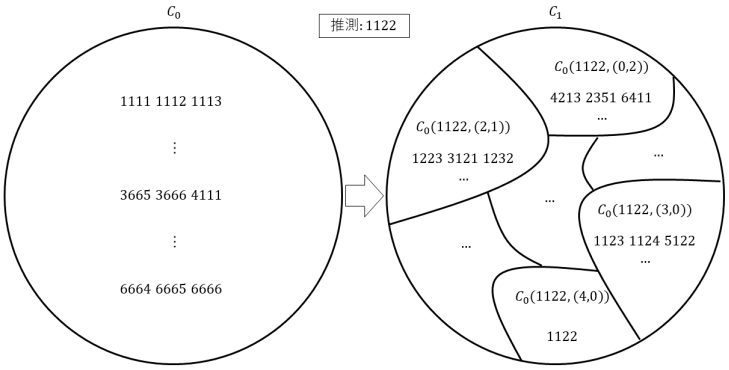

図 1: 1122 による C_0 の分割

このように，出題者に提示するべき推測の組み合わせを探索する．本研究では，この探索方法を使用して，出題者に提示する推測の数が最小となるようなアルゴリズムを考える．

3.3 実質的に同じテストパターン

前節でも述べたように，本研究ではいくつかの推測を同時に出題者に提示し，1296 個の推測を分割していくことが目的である．ここで 1 つ注意しなければいけないことが，分割に使う推測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，推測 $\{1111, 3332\}$ を用いて分割したとき，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．

これを踏まえて，提示するテストパターンの候補を絞り込む．解答者が出題者に提示する推測の候補は 1296 通り存在するが，その全てを考慮するのは効率的ではないので，次の考え方を用いて候補を減らす．例えば，推測 1122 と推測 3443 の 2 つの推測がある．この 2 つの推測のうち，3443 について，以下の操作を行う．

1. 数字の置換：推測に含まれる数字を，別の数字に置き換える
2. 場所の置換：推測の数字の位置を入れ替える

3443 についてこの操作を行うと，3 を 1 に，4 を 2 に置き換えて，場所を入れ替えると，1122 に置換できる．同様に，1144, 1155, 2211, 5656 なども，1122 に置換できる．このような推測の組を，1122 と実質的に同じ推測と呼ぶことにする．置換する際の数字や場所は任意に決定できる，つまり，3443 を 5522 や 1661 などに置換することも可能だが，ここでは，推

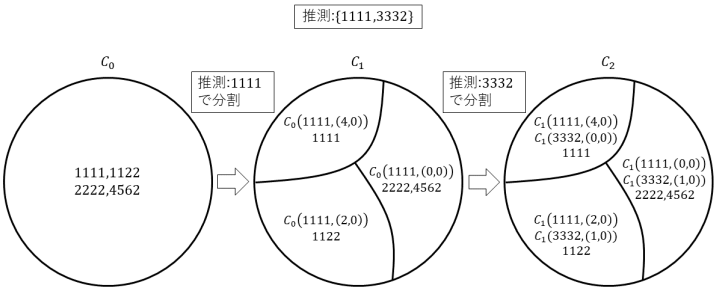

図 2: 1111,3332 の順で分割

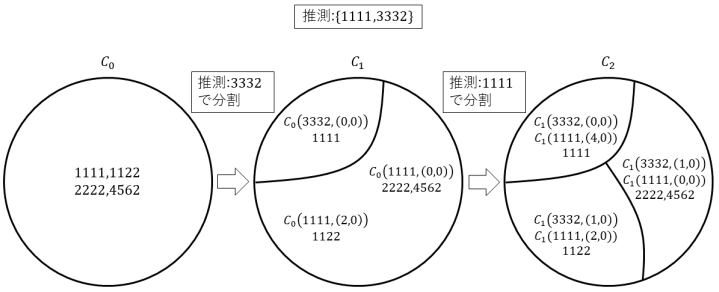

図 3: 3332,1111 の順で分割

測を 4 桁の整数とみなしたときに，推測が最も小さくなるように置換を行うこととする．1296 個の推測全てに数字と場所の置換を行い，置換後の推測が他の推測と同じになる場合，置換する前の推測を省略することができる．そのようにして残った推測が，出題者に提示する推測の候補である．この推測の候補を，代表的な推測と呼ぶ．例えば，1122 と実質的に同じ推測は，$\frac{4!}{2!2!} \times_6 C_2 = 90$ 個ある．

組み合わせのうちの 1 つ目の推測の候補について考えてみると，代表的な推測は表 1 に示す 5 つであることが分かる．この表は，1 列目に代表的な推測が，2〜15 列目に各推測を用いて分割した各部分集合の要素数が記載されている．1 行目に「ヒント」と書かれた行の下の行には 14 個のヒントが並んでいる．例えば，1122 に対してヒントが (0,0) と返ってくるような

コードの数は，1122 の行の 6 列目から，256 個であることが分かる．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．以後，同様に提示するテストパターンの候補を絞り込んでいく．1 つ目の推測として 1122 を選んだ場合，C_0 を 1122 で分割した状態 C_1 から，1296 個の推測から次に提示するテストパターンを絞り込んでいく．この絞り込みは，1296 個の推測の候補の先頭に，1122 を付けたもの，つまり，1122 から始まる 1296 個の 8 桁の整数の推測において，数字と場所の置換を行い，先頭 4 桁を取り除いたものが代表的な推測となるように行う．

少ない数の推測で秘密のコードが何かを特定するためには，各部分集合の要素数が重要になる．要素の多い部分集合があると，より細かく秘密のコードを区別するために必要な推測が多くなってしまう．そのため，解答者は，偏りなく C_0 を分割できる「最適な推測」を選択する必要がある．

4　A*アルゴリズムを用いた探索とプログラム構造

推測の最適な組み合わせを探索するために，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，その概要と具体的な探索方法を説明する．

4.1　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コスト」と「現在位置からゴールノードまでの推定コスト」の 2 つのコスト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解く．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

表 1: 初めに提示するテストパターンとヒント

	ヒント													
テストパターン	04	03	02	01	00	13	12	11	10	22	21	20	30	40
1111	0	0	0	0	625	0	0	0	500	0	0	150	20	1
1112	0	0	61	308	256	0	27	156	317	3	24	123	20	1
1122	1	16	96	256	256	0	36	208	256	4	32	114	20	1
1123	2	44	222	276	81	4	84	230	182	5	40	105	20	1
1234	9	136	312	152	16	8	132	252	108	6	48	96	20	1

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 にてコストを計算する．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 までの最短経路のコストを $g(n)$, ノード n からゴールノードまでの最短経路のコストを $h(n)$ とすると，ノード n を通る最短経路のコスト $f(n)$ は

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

と表せる．開始ノードからノード n までの最短経路のコストは計算できるが，ノード n からゴールノードまでの最短経路は実際に探索が終わるまでは分からないので，$h(n)$ が計算できず，他の経路とのコスト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経路」のコストを $h^*(n)$ とすると，ノード n を通る推定最短経路のコスト $f^*(n)$ は

$$f^*(n) = g(n) + h^*(n) \tag{5}$$

と表せる．この $h^*(n)$ をヒューリスティック関数という．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解を見つけるためには，$h^*(n)$ が常に楽観的な見積もりをする必要がある．すなわち，

$$0 \leq h^*(n) \leq h(n) \tag{6}$$

となる必要がある．このように $h^*(n)$ を見積もると，最短経路を見つけることができる．

現在探索可能な全てのノードにおいて $f^*(n)$ の値を計算して，最も値の小さいノードを選択していくことを，ゴールノード G にたどり着くまで繰り返す．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．

4.2 A*アルゴリズムのための推測数の見積もり

A*アルゴリズムを用いた具体的な探索方法を述べる. 解答者は，秘密のコードの集合 C_0 を，代表的な推測の中から 1 つ選択して分割していく．代表的な推測ごとにノードが与えられる．ゴールノードは，C_0 を n 個の推測を使用して分割したとき，分割の各要素数が 1 以下となるノードである．A*アルゴリズムにおける開始ノードを C_0 とすると，推測を 1 つ使用して C_0 を分割した C_1 が，開始ノードの次に探索するノードの候補である．探索するノードの候補全てにおいて，$f^*(n)$ の値を計算する．$f^*(n)$ は，ノード n を通る推定最短経路のコストであるが，本研究におけるコストは，推測の数である．このため，$f^*(n)$ における $g(n)$ を，そのノー

ドまでに使用した推測の数とする．また，$f^*(n)$ における $h^*(n)$ は，ノード n からゴールノードまでに必要だと推定される推測の数となる．そのため，現在ノードからあと何個の推測を使用すればゴールノードにたどり着くかを見積もる必要がある．C_0 を分割し全ての部分集合の要素数を 1 以下にしたいので，現在の部分集合のうち，要素数が最大なものが，分割に必要な推測を一番多く使うと予想される．また，$h^*(n)$ は常に楽観的な見積もりをする必要があり，一つの推測を使用して集合を高々 14 分割できる．このことから，C_1 の分割のうち，要素数が最大なものの要素数を c して，$h^*(n)$ を，$h^*(n) = \log_{14} c$ と定義する．このように $f^*(n)$ を計算することで，分割が終了するまでに使用する推測を楽観的に見積もれる．各ノードで $f^*(n)$ の値を計算したら，探索されていないノードの $f^*(n)$ の値を比較し，最も値の小さいノードを探索していく．これを，C_0 の分割の各部分集合の要素数が 1 以下になるまで，代表的な推測を絞り込み，探索を続けていく．

1 つ目のテストパターンの見積もりを例にして見ていく．表 2 は，1 つ目のテストパターンの候補と，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．$g(n)$ の値は，推測を一つしか使ってないので 1 となるため，$f^*(n)$ の値は表のようになる．$f^*(n)$ の値が最小なものを A*アルゴリズムでは探索するので，最初に推測 1122 が選択され，次の代表的なテストパターンを探索していく．

4.3 データ構造

A*アルゴリズムを用いて推測の最適な組み合わせを探索するプログラムを実装するために，ノードを表現するデータ構造を考える．

推測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，木の深さがテストパターンの数を表している．ここで，木の根を深さ 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，まず親ノードへのリンクがある．親ノードは，ひとつ前に選択した推測を示している．2 段目はノードに対応したテストパターンが格納されている．このテストパターンを用いて，C_n の分割や次の代表的なテストパターンの推測を行う．3 段目には，$f^*(n)$ の値が

表 2: 初めに提示するテストパターンと見積もり

テストパターン	部分集合の最大値	$h^*(n)$	$f^*(n)$
1111	625	$\log_{14} 625$	$\log_{14} 625 + 1$
1112	317	$\log_{14} 317$	$\log_{14} 317 + 1$
1122	256	$\log_{14} 256$	$\log_{14} 256 + 1$
1123	276	$\log_{14} 276$	$\log_{14} 276 + 1$
1234	312	$\log_{14} 312$	$\log_{14} 312 + 1$

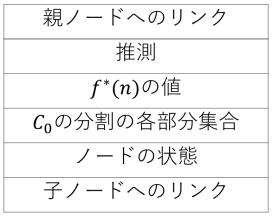

図 4: 各ノードの構造

格納されており，この値を他の探索済みでないノードと比較して最も値の小さいものを探索する．4 段目には C_0 を現在の推測で分割したものが格納されている．C_0 の分割を行う時は，親ノードのこの場所を参照して分割を行う．5 段目には，ノードが探索済みかどうかを表す状態が格納されている．初期状態は Open となっており，探索が終了したときに Close となる．6 段目には，次の代表的なテストパターンごとに子ノードへのリンクがあり，リンク先には，また同じ構造体が存在する．

木構造を用いたプログラムは以下の手順に沿って動く．

1. 木の根のノードを作成し，親ノードとする．木の根には，親ノードへのリンクが存在せず，推測，分割された C_0 の各部分集合，子ノードへのリンクは None である．$f^*(n)$ の値は $\log_{14} 1296$ と，C_0 が分割されていないことを示している．
2. 推測の絞り込みを行う．推測が絞り込めたら，絞り込んだ推測をそれぞれ，親ノードの子ノードとしてリンク付けして格納する．このとき，それぞれの子ノードにおいて，$f^*(n)$ の値を計算し，C_0 を分割して格納し，ノードの状態を Open にする．その他の状態は None となっている．
3. 親ノードの状態を Close にする．
4. 状態が Open のノードのうち，分割された C_0 の各部分集合の要素数が 1 以下のノードが存在すればそのノードがゴールノードであり，探索は終了する．探索によって見つかった推測の組み合わせは，木の根からゴールノードまでの経路上にあるノードに格納された推測によって表される．そうでない場合， ノードの状態が Close となっていないノードのうち，$f^*(n)$ の値が最も小さいノードを親ノードとし，手順 2 に戻って探索を続ける．

プログラムの具体的な動きを見ていく．まず，木の根の作成を手順 1 に従って作成する．次に，推測の絞り込みを行い，絞り込んだ推測をノードとして，それぞれを木の根の子ノードとしてリンク付けして格納する．推測の絞り込みは前節に示したように行うので，木の根の子ノードへのリンクは 5 つとなる．手順 2 に従って子ノードの作成が終了したら，木の根のノー

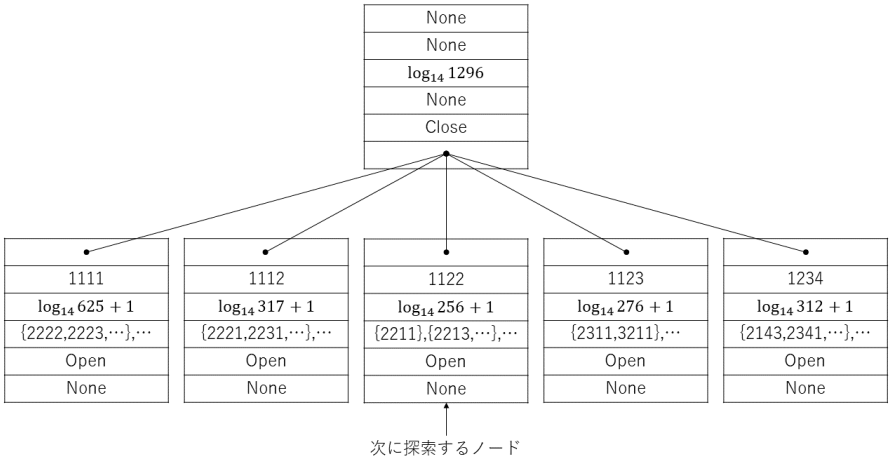

図 5: 最初の推測の候補

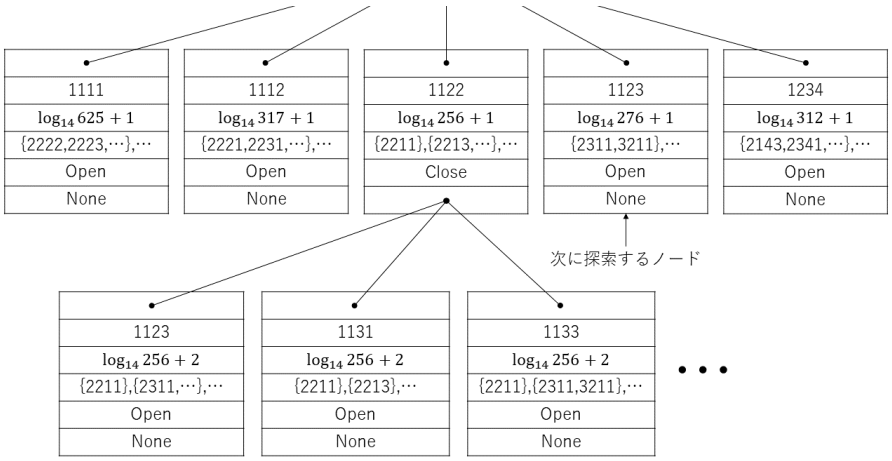

図 6: 次の推測の候補

ドの状態を Close にする．図 5 に，この状態の木を示す．$f^*(n)$ の値は，$h^*(n)$ の値に，木の深さである，$g(n) = 1$ が足された値となっている．そして，状態が Open のノードの $f^*(n)$ の値を比較し，最も値の小さいノードを探索していく．ここでは，推測 1122 が C_0 の分割の必要な推測の数の見積もりが最も少ないので，推測 1122 を探索していく．推測 1122 が選択されると状態が Open になり，手順 2 に戻って次の代表的なテストパターンが絞り込まれ，1122 の子ノードが生成される．子ノードの生成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．これを，Open のノードのうち，分割された C_0 の各部分集合の要素数が 1 以下になるノードが作成されるまで繰り返す．探索が終了すると，推測の最適な組み合わせが確定する．

5　問題点

前節にて述べたプログラムを，Python を用いて実装したが，使用メモリ量が多く，プログラムが実行中にメモリが足りなくなるという問題が起きている．使用メモリ量を減らすため，秘密のコードを 2 桁にしてプログラムを実行したところ，尤もらしい結果が出てきたため，秘密のコードが 4 桁の場合に結果が出ない要因は，次のように考えられる．

まず，分割された C_0 の各部分集合をノードに格納することである．作成したプログラムでは，部分集合の要素を配列にして各ノードに格納していたが，配列の大きさやノードの数が非常に多く，メモリ不足になってしまったと考えられる．実際，コンピュータのメモリが不足しているといったウィンドウが表示され，プログラムが停止した．

次に，代表的なテストパターンの絞り込みの効率の悪さである．テストパターンの絞り込みには，各桁の数字の，入れ替えと置換を用いて行う．例えば，2123 というコードが代表的なコードかどうか確かめるためには，2 桁目の 1 と 3 桁目の 2 を入れ替え，2213 という並びにする．そして，2 を 1 に，1 を 2 に置換すると，1123 というコードになる．1123 は 2123 よりも値が小さいため，2123 は代表的なテストパターンではないことが分かる．木の高さが 2 以降の代表的なテストパターンの絞り込みは，木の高さが増えるにつれて複雑な入れ替えと置換が必要になってくる．この入れ替えと置換のアルゴリズムを考えることができず，すべての入れ替えと置換を行って一番値が小さいものを代表的なテストパターンにするというアルゴリズムにしたため，計算時間やメモリ使用量が大幅に増えてしまったと考えられる．

最後に，Python はメモリの確保をユーザが指定せずにできるため，メモリ使用をユーザ側できめ細かく管理できないという点もあると考えられる．

これらの反省点を活かし，プログラムの改善を図る必要がある．

6　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拡張し，同時に推測を提示する場合に必要な推測の組み合わせを見つけるためのアルゴリズムを提案した．A*アルゴリズムが最短経路の探索に優れていることを利用し，より的確に秘密のコードを小さく分けていき，同時に提示する推測を発見できると考えられる．本研究では提案アルゴリズムに基づいてプログラムの実装も行い，解を求めようとしたが，代表的なテストパターンの絞り込みに時間がかかったり，C_0 の分割を保存しておくのに多くのメモリを消費したりと，計算時間が早くメモリ消

費の少ないプログラムを作成することができなかったため，結果の考察や従来研究との比較にまでは至っていない．前節で述べた問題点や，さらなる計算の高速化，メモリの効率の良い利用方法などを考案し，それを取り入れたプログラムを作成することが今後の課題である．

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, 2020 年 1 月閲覧．

付録 A　ソースコード

A.1　提示する複数の推測を探索するプログラム

```
# coding: utf-8

import math
import itertools
import numpy as np
import sys
import gc
import psutil

#2 桁
CODELEN = 2
DIVPAT = 6

P_list = [0,1]
Position = list(itertools.permutations(P_list))
N_list = ['1','2','3','4','5','6']
Number = list(itertools.permutations(N_list))
Num = '123456'
Code = [i+j for i in Num for j in Num]

"""
#3 桁
CODELEN = 3
DIVPAT = 10

P_list = np.array([0,1,2],dtype=np.int32)
Position = list(itertools.permutations(P_list))
N_list = np.array(['1','2','3','4','5','6'],dtype=object)
Number = list(itertools.permutations(N_list))
Num = '123456'
Code = np.array([i+j+k for i in Num for j in Num for k in Num],dtype=object)
"""

"""
#4 桁
CODELEN = 4
DIVPAT = 14

P_list = [0,1,2,3]
Position = list(itertools.permutations(P_list))
N_list = ['1','2','3','4','5','6']
Number = list(itertools.permutations(N_list))
Num = '123456'
Code = [i+j+k+l for i in Num for j in Num for k in Num for l in Num]
"""

#場所入れ替え (4!)
def permutation(List,Times,min):
    for i in range(len(Position)):
        c_code = np.array(['0'] * len(List),dtype=object)
        for j in range(Times):
            for k in range(CODELEN):
                c_code[Position[i][k]+(j*CODELEN)] = List[k+(j*CODELEN)]
        c_code = int(''.join(c_code))
        if c_code < min:
            min = c_code
    return min

#数字入れ替え (6!)
def change(List):
    min = int(List)
```

```
    if C_list.iterative_tree_search(min) == 1:
        return min
    Times = len(List)//CODELEN
    for i in range(len(Number)):
        c_code = np.array(['0'] * len(List),dtype=object)
        for j in range(Times):
            for k in range(CODELEN):
                c_code[k+(j*CODELEN)] = Number[i][int(List[k+(j*CODELEN)])-1]
        if sum(int(l) for l in c_code) <= sum(int(m) for m in List):
            c_code = int(''.join(c_code))
            if c_code < min:
                min = c_code
            min = permutation(str(c_code),Times,min)
    return min
```

#2 色木の作成

```
class Node:
def __init__(self,data):
     self.parent = None
     self.color = "Red"
     self.data = data
     self.left = None
     self.right = None

class T_nil(Node):
def __init__(self):
     self.parent = self.left = self.right = self.data = None
     self.color = "Black"

class BCT:
def __init__(self,node):
     self.root = None
     self.rb_insert(node)
   def rb_insert(self,data):
     n = self.root
     if n == None:
       self.root = Node(data)
       z = self.root
       z.parent = T_nil()
       z.parent.left = z
       z.parent.right = z
     else:
       while n.data != None:
         y = n
         if data < n.data:
           n = n.left
         else:
           n = n.right
       if data < y.data:
         y.left = Node(data)
         z = y.left
         z.parent = y
       else:
         y.right = Node(data)
         z = y.right
         z.parent = y
     z.right = T_nil()
     z.left = T_nil()
     z.right.parent = T_nil()
     z.left.parent = T_nil()
     self.rb_insert_fixup(z)
     return

   def rb_insert_fixup(self,z):
     while z.parent.color == "Red":
       if z.parent == z.parent.parent.left:
         y = z.parent.parent.right
         if y.color == "Red":
           z.parent.color = "Black"
```

```
            y.color = "Black"
            z.parent.parent.color = "Red"
            z = z.parent.parent
          else:
            if z == z.parent.right:
              z = z.parent
              self.left_rotate(z)
            z.parent.color = "Black"
            z.parent.parent.color = "Red"
            self.right_rotate(z.parent.parent)
        else:
          y = z.parent.parent.left
          if y.color == "Red":
            z.parent.color = "Black"
            y.color = "Black"
            z.parent.parent.color = "Red"
            z = z.parent.parent
          else:
            if z == z.parent.left:
              z = z.parent
              self.right_rotate(z)
            z.parent.color = "Black"
            z.parent.parent.color = "Red"
            self.left_rotate(z.parent.parent)
self.root.color = "Black"

#探索
   def iterative_tree_search(self,k):
     n = self.root
     while(n.data != None and k != n.data):
       if k < n.data:
         n = n.left
       else:
         n = n.right
     if n.data == None:
       return 0
     else:
       return 1

   #左回転
   def left_rotate(self,x):
     y = x.right
     x.right = y.left
     if y.left.data != None:
       y.left.parent = x
     y.parent = x.parent
     if x.parent.data == None:
       self.root = y
     elif x == x.parent.left:
       x.parent.left = y
     else:
       x.parent.right = y
     y.left = x
     x.parent = y

   #右回転
   def right_rotate(self,x):
     y = x.left
     x.left = y.right
     if y.right.data != None:
       y.right.parent = x
     y.parent = x.parent
     if x.parent.data == None:
       self.root = y
     elif x == x.parent.left:
       x.parent.left = y
     else:
       x.parent.right = y
     y.right = x
```

```
      x.parent = y

#多分木の作成
class MultipleNode:
   def __init__(self,data):
     self.parent = None
     self.data = data
     self.childlist = np.array([],dtype=np.int32)
     self.estimate = None
     self.status = None
     self.divid = None

class MT:
   def __init__(self,number_list):
     self.root = MultipleNode(pow(DIVPAT,CODELEN+1))
     self.root.estimate = pow(DIVPAT,CODELEN+1)
     self.root.divid = Code
     n = self.root
     if len(n.childlist) != 0:
       print("木の作成に失敗")
       sys.exit()
     for data in number_list:
       n.childlist = np.append(n.childlist,MultipleNode(data))
       n.childlist[len(n.childlist)-1].parent = n

   #挿入
   def insert(self,number_list,n):
     for data in number_list:
       n.childlist = np.append(n.childlist,MultipleNode(data))
       n.childlist[len(n.childlist)-1].parent = n
     return

   #見積もりを与える
   def estimatedata(self,est,n):
     n.estimate = est
     return

   #見積もりの最小値計算
   def minest(self,n):
     min = Tree.root.estimate
     new_n = Tree.root
     for i in range(len(n.childlist)):
       if n.childlist[i].estimate == None:
         pass
       else:
         if len(n.childlist[i].childlist) != 0:
           N = self.minest(n.childlist[i])
           if min > N.estimate:
             new_n = N
             min = N.estimate
         elif n.childlist[i].status == "Close":
           pass
         elif min > n.childlist[i].estimate:
           new_n = n.childlist[i]
           min = n.childlist[i].estimate
     return new_n

   #ノードの高さを調べる
   def height(self,n):
     hei = 0
     while n != None:
       hei += 1
       n = n.parent
     return hei

#コードの分割
def div_code(Pat,Code,index_table,max,div):
   divcode = [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
   prof = np.zeros(14,int)
```

```
    for code in Code:
      cc = np.zeros(CODELEN,int)
      ct = np.zeros(CODELEN,int)
      bh = 0
      wh = 0
      for i in range(CODELEN):
        if Pat[i] == code[i]:
          bh += 1
          cc[i] =ct[i] = 1

      for i in range(CODELEN):
        if cc[i]:
          continue
        for j in range(CODELEN):
          if ct[j]:
            continue
          if code[i] == Pat[j]:
            wh += 1
            ct[j] = 1
            break

      resp = bh * 5 + wh

      if (resp > 24 or resp < 0):
        resp = 24

     prof[index_table[resp]] += 1
      divcode[index_table[resp]].append(code)
    for i in range(len(prof)):
      if prof[i] > max:
        max = prof[i]

    for i in range(len(divcode)):
      div.append(divcode[i])
    return max,div

#見積もりを計算する関数
def calc_est(n):
    Pat = str(n.data)
    index_table = np.array([4,3,2,1,0,
                            8,7,6,5,-1,
                            11,10,9,-1,-1,
                            12,-1,-1,-1,-1,
                            13,-1,-1,-1,-1],dtype=np.int32)
    max = 0
    maxdiv = (0,[])
    if n.parent == Tree.root:
      maxdiv = div_code(Pat,Tree.root.divid,index_table,maxdiv[0],maxdiv[1])

    else:
      for i in range(len(n.parent.divid)):
        maxdiv = div_code(Pat,n.parent.divid[i],index_table,maxdiv[0],maxdiv[1])

    max = maxdiv[0] * pow(DIVPAT,Tree.height(n)-1)
    n.divid = np.array(maxdiv[1])
    return max

#見積もりの最大値を与える
def dataest(n):
    for i in range(len(n.childlist)):
      pat_n = n.childlist[i]
      max = calc_est(pat_n)
      Tree.estimatedata(max,pat_n)

#main
#最初の候補
Pattern = np.array([],dtype=np.int32)
Pattern = np.append(Pattern,int(Code[0]))
```

```
C_list = BCT(int(Code[0]))
for i in range(1,len(Code)):
   code = int(Code[i])
   min = change(Code[i])
   flag = C_list.iterative_tree_search(min)
   if flag == 0:
     Pattern = np.append(Pattern,code)
     C_list.rb_insert(int(code))

del C_list
gc.collect()

Tree = MT(Pattern)
maxheight = 2
#見積もりの最小値計算
n = Tree.minest(Tree.root)
data = n.data
n.status = "Open"

#見積もり計算
dataest(n)
n.status = "Close"

#1296*14
Tree.root.estimate *= DIVPAT
cnt = 0
while True:
   #見積もりの最小値計算
   n = Tree.minest(Tree.root)
   print(n.data)
   cnt += 1
   if cnt % 50 == 0:
       mem = psutil.virtual_memory()
       print(cnt,"個目のデータを分割中")
       print("メモリ使用量:",mem.used)
       print("メモリ使用率:",mem.percent)
       print("メモリ空き容量:",mem.available)
       print("CPU 使用率:",psutil.cpu_percent(interval=1))

   n.status = "Open"

   #掘り下げ
   Pattern = np.array([],dtype=np.int32)
   code = np.array([],dtype=object)
   code = np.append(code,str(n.data))
   code_n = n.parent
   while (code_n.parent != None):
     code = np.append(code,str(code_n.data))
     code_n = code_n.parent
   j = 0
   while n.data > int(Code[j]):
     j += 1
   code = np.append(code,Code[i])
   C_list = BCT(int(''.join(code)))
   code = code[:-1]

   for i in range(j+1,len(Code)):
     code = np.append(code,Code[i])
     data = ''.join(code)
     min = change(data)
     flag = C_list.iterative_tree_search(min)
     if flag == 0:
       Pattern = np.append(Pattern,int(Code[i]))
       C_list.rb_insert(int(data))
     code = code[:-1]

   Tree.insert(Pattern,n)

   #見積もり計算
```

```
    dataest(n)

    n.status = "Close"

    #高さ計算
    height = Tree.height(n)
    if maxheight < height:
      maxheight = height
      Tree.root.estimate *= DIVPAT
    #終了条件
    endest = pow(DIVPAT,height)
    flag = 0
    for i in range(len(n.childlist)):
      if n.childlist[i].estimate == endest:
        flag = 1
        n = n.childlist[i]
        print("END")
        break
    if flag == 1:
      break

print("同時にする質問は",end = "")
while(n.parent != None):
    print(n.data,",",end = "")
    n = n.parent
```