パズルで鍛える

アルゴリズム力

第5回

覆面算を解いて、作る!

これまでさまざまなパズルを解くことをとおして、グラフ探索アルゴリズムの考え方に馴染んできました。今回は趣向を変えて、覆面算パズルを「解く」だけではなく「作る」ことに焦点を当ててみます。

Author けんちょん(大槻 兼資、Kensuke Otsuki) 株式会社NTTデータ数理システム **E-mail** dr.ken.1215@gmail.com **Twitter** @drken1215

覆面算とは

覆面算は、足し算や引き算の筆算において、0から9の各数値がそれぞれ対応する別の文字(ひらがなやアルファベットなど)に置き換えられたものが与えられ、もとの筆算を復元するパズルです。復元に際してのルールは次のとおりです。

- 同じ文字には同じ数値が入り、異なる文字には異なる数値が入る
- 最上位の桁の文字に0は入らない

たとえば、図1の覆面算を実際に解いてみましょう。まず、2桁の整数と2桁の整数の和は最大でも198であることから、「い」=1と確定します(図2)。次に、「か」+「1」の部分で繰り上がりが発生する必要があることから、「か」=9と確定します(「か」=8となる可能性もありますが、調べていくと矛盾することがわかります)。最後に、「わ」と「る」を埋めると、図3のように求められます。以上より、図1の覆面算の解は「98 + 11 = 109」であり、ただ1つに決まることがわかりました。このように、覆面算の作品は通常、一意解となるように作られています[注1]。

さて、本記事では、まず覆面算を解くアルゴリズムについて検討します。その後、覆面算を作る方法を考えます。アルゴリズムが、パズルを「解く」だけではなく「作る」のにも役立つことを示していきます。なお、本記事のソースコードはC++で記述していますが、基本的な機能のみを用いているため、ほかの多くの言語でも同様の処理を実現できます。

▼図1 覆面算の例

```
  か わ
+ い い
-------
い る か
```

▼図2 図1の覆面算の「い」に1を入れた様子

```
  か わ
+ 1 1
-------
1 る か
```

▼図3 図1の覆面算の答え

```
  9 8
+ 1 1
-------
1 0 9
```

注1) 一般にパズルにおいて、条件を満たす解がただ1つに決まるとき、一意解であると言います。覆面算に限らず、数独など多くのペンシルパズルでは、その作品に一意解であることを課しています。

覆面算の例

覆面算を解くアルゴリズムや作るアルゴリズムを考える前に、おもしろい覆面算作品をいくつか紹介します。

ワード覆面算

覆面算のうち、意味を持った単語や文章で作られたものをワード覆面算と呼びます。**図4**は、1924年にH. A. Dudeneyによって発表された作品「SEND MORE MONEY」です。覆面算を紹介する文献では必ずと言ってよいほど掲載されている有名な作品です。この作品をきっかけとして、覆面算が世界中に普及していったと言っても過言ではないでしょう。本記事では、のちほど「SEND + MORE = ?????」という形をした覆面算として、ほかにどのようなものが考えられるかを検討します。

テーマ覆面算

何らかのテーマに関する単語のみを用いて作られる覆面算もおもしろいものです。たとえば、日本の都道府県名は47個あります。「都道府県名」+「都道府県名」=「都道府県名」という形をした覆面算を考えてみましょう。これは、**図5**に示した3つの作品が知られています[注2]。なお、2つめ(ながの + ながさき = とうきょう)と3つめ(みやぎ + みやざき = とうきょう)は本質的に同一のものです。

▼図4　H. A. Dudeney作「SEND MORE MONEY」

```
  S E N D
+ M O R E
---------
M O N E Y
```

数詞覆面算

テーマ覆面算のうち、各単語が数値を表していて、かつその数値の計算も正しく成立しているものを数詞覆面算と呼びます。**図6**は、1947年にA. Wayneによって作られた有名な作品です。40 + 10 + 10 = 60となっており、この足

▼図6　FORTY+TEN+TEN=SIXTY

```
  F O R T Y
      T E N
+     T E N
-------------
  S I X T Y
```

▼図5　都道府県名で作られる覆面算

```
  おおいた        ながの        みやぎ
+おおさか     +ながさき     +みやざき
---------     ---------     ---------
  さいたま      とうきょう      とうきょう
```

注2) この3個以外には一意解となる組み合わせが存在しないことも言えます。

し算が正しいうえに覆面算としても一意解となっています。

幾何模様覆面算

文字の配列に趣向を凝らし、幾何的に美しい模様を作り出した覆面算を、幾何模様覆面算と呼ぶことがあります。図7は1980年に高木茂男によって発表された作品です。虫食算・覆面算を多数収録した書籍『虫食算パズル700選[注3]』においても、問482として掲載されています。本記事でも、幾何模様覆面算を1つ作ります。

覆面算を解く

それではまず、覆面算ソルバーを実装しましょう。ここでは、一意解ではない入力が与えられたとしても、それに対して解をすべて求めるアルゴリズムを実装します。

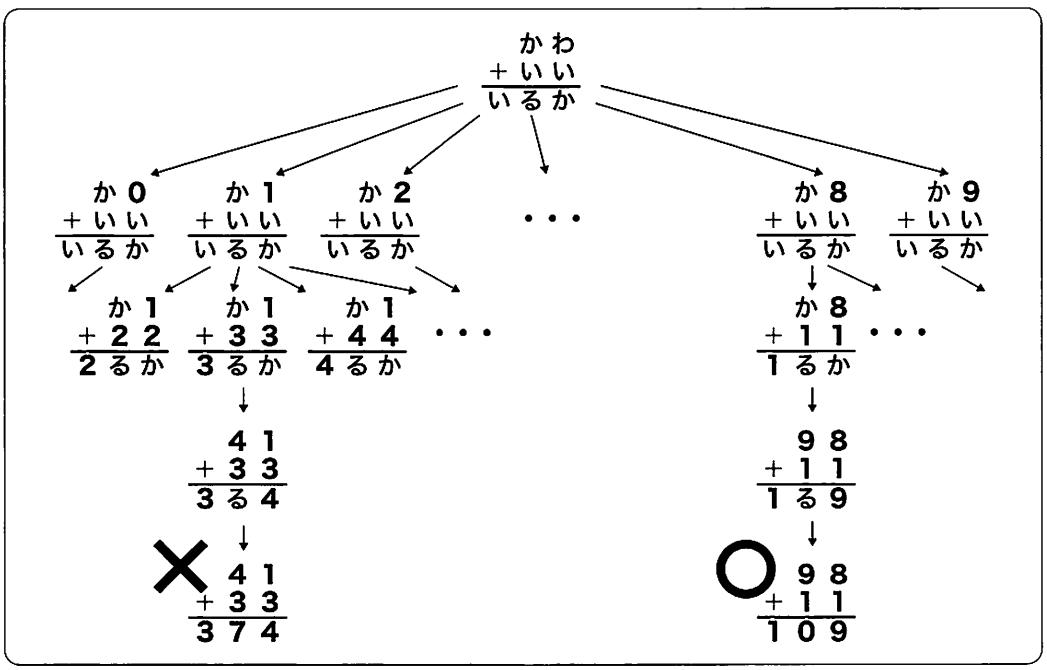
覆面算を表すグラフ上の探索

覆面算を解く探索過程は、図8のようなグラフで表せます[注4]。このグラフをリスト1のような再帰関数を用いて探索することで、覆面算を

▼図7　幾何模様覆面算の例

```
      D E F
    C D E F
    C D E F
  B C D E F
+ B C D E F
-----------
A B C D E F
```

▼図8　覆面算を解く探索過程を表すグラフ



注3）大駒 誠一、武 純也、丸尾 学 著、共立出版 刊、1985年
注4）本誌2020年8月号では、数独を解く探索過程をグラフとして表して、そのグラフ上のバックトラッキング法を実行することで数独を解きました。覆面算もまったく同様の方法によって解くことができます。

▼リスト1　覆面算を解く探索アルゴリズム

```
// 覆面算fuのi段めのj桁めに数値を入れることを考える
void rec(Fukumenzan &fu, int i, int j) {
    if (すべての文字に矛盾なく数値が入ったら) {
        解を格納する;
        return;
    }

    if (i段め、j桁めの文字にすでに数値が入っている) {
        rec(fu, 次の段, 次の桁);
    }

    // バックトラッキング
    for (int v = 0; v <= 9; ++v) {
        if (fu.set_val(i, j, v)) {
            rec(fu, 次の段, 次の桁);
            fu.reset_val(i, j, v);
        }
    }
}
```

解くことができます[注5]。なお、整数の一の位、十の位、百の位、……を順に0桁め、1桁め、2桁め、……と呼ぶことにします。また、覆面算に対して次のメソッドが実装されているものとします。

- fu.set_val(i, j, v)：覆面算fuのi段め、j桁めの文字をxとして、xをすべて数値vで置き換える関数。ただし置き換えによって矛盾が生じる場合には、もとに戻したうえでfalseを返す
- fu.reset_val(i, j, v)：覆面算fuのi段め、j桁めの文字をxとして、xに入っている数値を取り除く関数

具体的な実装については、次節で示します。

覆面算ソルバー

本記事では、覆面算を解く過程を管理するクラスFukumenzanを、リスト2のように実装しておきます[注6]。また、覆面算の各文字に割り当てた数値によって矛盾が生じていないかどうか

▼リスト2　覆面算を表すクラス

```
class Fukumenzan {
 public:
    // 覆面算の問題を表す情報
    std::vector<std::string> problem;

    // 覆面算を解く過程を表す情報
    std::vector<std::vector<int>> field;

    // 使用された数値
    std::set<int> used;

 public:
    // コンストラクタ
    Fukumenzan(const std::vector<std::string> &input) {
        for (int i = 0; i < input.size(); ++i) {
            std::string str = input[i];
            std::reverse(str.begin(), str.end());
            problem.emplace_back(str);
            int N = (int)str.size();
            field.emplace_back(std::vector<int>(N, -1));
        }
    }

    // 覆面算の段数を返す
    size_t size() { return field.size(); }

    // 覆面算のi段めの桁数を返す
    size_t digit(int i) { return field[i].size(); }

    // r段め、d桁めに数値が入っているとき：その数値を返す
    // 文字が数値に置き換わっていないとき：−1
    int get_val(int r, int d) {
        if (d >= (int)field[r].size()) return 0;
        else return field[r][d];
    }

    // 現時点で数値に矛盾が生じていないかを確認する
    bool is_valid();

    // r段め、d桁めの文字を数値vで置き換える
    // 置き換え不可の場合は何もせずにfalseを返す
    bool set_val(int r, int d, int v);

    // r段め、d桁めの文字に入っていた数値をリセットする
    void reset_val(int r, int d, int v);

    // fieldの様子を出力
    void print() {
        for (int i = 0; i < field.size(); ++i) {
            for (int j = field[i].size() - 1; j >= 0; --j)
                std::cout << get_val(i, j) << ", ";
            std::cout << std::endl;
        }
    }
};
```

注5）覆面算を解くだけならば、次のようにするほうが簡単です。文字は高々10種類しか登場しないことから、各文字への数値の割り当て方法は高々10!通り以下となります。したがって、これらをすべて試すことで覆面算を解くことができます。しかしながら、のちほどワイルドカードを利用して覆面算を作るときには、本記事で紹介するアルゴリズムを用いたほうが扱いやすくなります。

注6）リスト2〜5ではシンプルな実装をしていますが、あらかじめ各文字の位置情報を持っておくなど、さまざまな高速化の工夫が考えられます。

▼リスト3　覆面算の各文字に割り当てた数値が矛盾を起こさないかどうかを確認する関数

```
bool Fukumenzan::is_valid() {
    // 最上桁に0がない
    for (int i = 0; i < field.size(); ++i) {
        if (field[i].back() == 0) return false;
    }

    // 一の位から順に計算していく
    int kuriagari = 0;
    for (int j = 0; j < field.back().size(); ++j)
    {
        bool all_set = true;
        int sum = kuriagari;
        for (int i = 0; i + 1 < field.size(); ☑
++i) {
            int val = get_val(i, j);
            if (val == -1) {
                all_set = false;
                break;
            }
            sum += val;
        }
        if (!all_set) return true;
        kuriagari = sum / 10;
        int amari = sum % 10;
        int answer = get_val(field.size()-1, j);
        if (answer != -1 && amari != answer)
            return false;
    }
    if (kuriagari > 0) return false;
    else return true;
}
```

▼リスト4　覆面算クラスのset_val()とreset_val()の実装

```
bool Fukumenzan::set_val(int r, int d, int v) {
    // ワイルドカードに関する処理
    if (problem[r][d] == '?') {
        field[r][d] = v;
        if (is_valid()) return true;
        else {
            reset_val(r, d, v);
            return false;
        }
    }

    // すでに使用済みの数値は不可
    if (used.count(v)) return false;

    // 文字を置き換えていく
    used.insert(v);
    for (int i = 0; i < field.size(); ++i) {
        for (int j = 0; j < field[i].size(); ++j) {
            if (problem[i][j] == problem[r][d])
                field[i][j] = v;
        }
    }

    // 置き換え不可の場合はもとに戻す
    if (!is_valid()) {
        reset_val(r, d, v);
        return false;
    }
    else return true;
}

void Fukumenzan::reset_val(int r, int d, int v) {
    // ワイルドカードに関する処理
    if (problem[r][d] == '?') {
        field[r][d] = -1;
        return;
    }

    // 文字を戻していく
    used.erase(v);
    for (int i = 0; i < field.size(); ++i) {
        for (int j = 0; j < field[i].size(); ++j) {
            if (problem[i][j] == problem[r][d])
                field[i][j] = -1;
        }
    }
}
```

を確認する関数Fukumenzan::is_valid()を、リスト3のように実装します。さらに、前節で述べたメンバ関数set_val()、reset_val()をリスト4のように実装します。なおこれらは、のちほど解説する「ワイルドカード文字を用いて覆面算を作る方法」を意識した実装となっています。ワイルドカード文字は「?」で表しており、次の性質を表すものです。

・「?」にはどのような数値を入れてもよい(最上位の桁の0を除く)
・ほかの文字と同じ数値を入れてもよい

さて、リスト2〜4の実装を用いて、探索アルゴリズムを具体的に実装してみましょう。リスト5のように書くことができます。具体的な覆面算として、リスト6の入力「SEND + MORE = MONEY」を与えると、結果は図9のようになりました。確かに一意解であることが見て取れます。

覆面算を作る

いよいよ、覆面算を解くアルゴリズムを有効活用して、覆面算を作る方法を考えてみましょ

▼リスト5　覆面算ソルバー全体の実装

```
#include <iostream>
#include <vector>
#include <string>
#include <set>
#include <algorithm>

// 覆面算fuのi段め、j桁めに数値を入れることを考える
// res：条件を満たす解をすべて格納する変数
void rec(Fukumenzan &fu, int i, int j,
         std::vector<Fukumenzan> &res) {
    if (i == 0 && j == fu.digit(fu.size() - 1)) {
        res.emplace_back(fu);
        return;
    }

    // 次のマス
    int ni = i + 1, nj = j;
    if (i + 1 == fu.size()) ni = 0, nj = j + 1;

    // すでに数値が入っている場合は即座に次のマスへ
    int val = fu.get_val(i, j);
    if (val != -1) {
        rec(fu, ni, nj, res);
        return;
    }

    // バックトラッキング
    for (int v = 0; v <= 9; ++v) {
        if (fu.set_val(i, j, v)) {
            rec(fu, ni, nj, res);
            fu.reset_val(i, j, v);
        }
    }
}

int main() {
    int N;
    std::cin >> N;
    std::vector<std::string> input(N);
    for (int i = 0; i < N; ++i) std::cin >> input[i];

    // 再帰的に解く
    Fukumenzan fu(input);
    std::vector<Fukumenzan> res;
    rec(fu, 0, 0, res);

    // 解を出力
    for (int i = 0; i < res.size(); ++i) {
        std::cout << i << " th result:" << std::endl;
        res[i].print();
    }
}
```

う。さまざまな方法が考えられますが、ここではワイルドカードを用いる方法を紹介します[注7]。

▼リスト6　覆面算「SEND＋MORE＝MONEY」の入力

```
3
SEND
MORE
MONEY
```

▼図9　覆面算「SEND＋MORE＝MONEY」の解

```
0 th result:
9, 5, 6, 7,
1, 0, 8, 5,
1, 0, 6, 5, 2,
```

ワイルドカード

ワード覆面算を作るとき、すでにある程度文章のテーマが決まっていることが多いです。そこで、いったん「どのような数値を入れてもよい」ということを表すワイルドカード文字を用いた覆面算を考えます。たとえば覆面算「SEND + MORE = MONEY」の代わりに、「SEND + MORE = ?????」という形を考えます。本節で紹介するアルゴリズムは、これが一意解となるように「?」を文字で置き換える方法をすべて求めるものです。

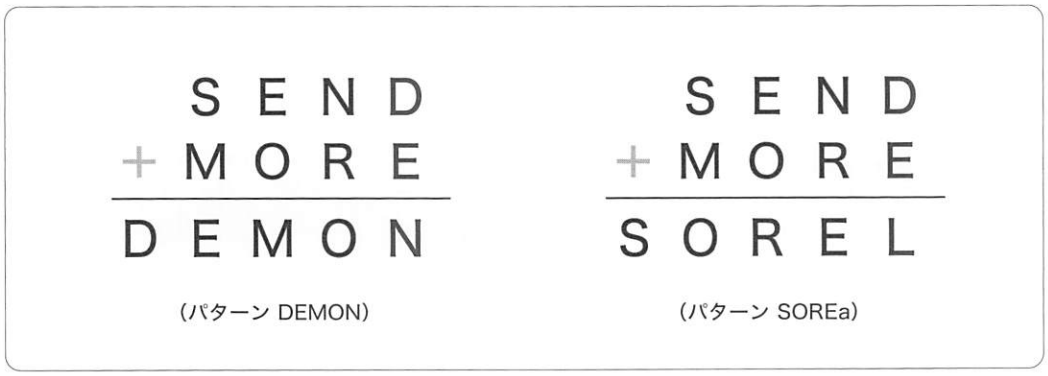
まず、「?」にはどのような数値を入れてもよいものとして、覆面算を解きます。そして出力される各解について、「?」を次のルールに則って文字で置き換えます。置き換えてできる文字列を「パターン」と呼ぶことにします。

> 各「?」に対して、順に次の処理を行う
> ・「?」で登場した数値が、すでにほかの文字に入った数値と一致するならば、「?」をその文字で置き換える
> ・「?」で登場した数値が、ほかのどの文字に入った数値とも一致しないならば、「?」を新たな文字で置き換える

ここでは「新たな文字」として、英小文字a、b、c、……を順に割り当てていくこととします。そしてすべての解を、パターンごとに分類します。このとき、ただ1つの解が属するパターン

注7）ほかの作り方としては、たとえば都道府県名を列挙して、それらの組み合わせであって一意解となるものを探索する方法が考えられます。

▼図10 「SEND＋MORE＝?????」の形をした覆面算



については、それ自体が一意解の覆面算となっていることがわかります！ こうして、ワイルドカード文字を用いることで、さまざまな覆面算作品を作れることがわかりました。

以上の一連の手続きは、**リスト7**のように実装できます。入力として、「SEND + MORE = ?????」を与えてみましょう。このとき、一意解となるパターンは合計368個出力されます。そのうちの1つに「MONEa」があります。これに対して文字aにYを当てはめることで、覆面算「SEND + MORE = MONEY」ができあがります。同様に、**図10**のようなオリジナル覆面算も作れます。

2桁+1桁=3桁の覆面算

桁数の小さな覆面算であれば、あり得る覆面算のパターンをすべて求めることもできます。ここではワイルドカード法を活用して、「2桁」+「1桁」=「3桁」の形となる覆面算をすべて求めてみましょう。筆算すべてを「?」で表した入力「?? + ? = ???」を与えて解きます。そしてパターンが一意になるものを抽出すると、**図11**に示す6通りの覆面算が生成されます。

これらに文字を当てはめることでワード覆面算を作ることもできます。たとえば**図11**の右下の覆面算において、「a」=「は」、「b」=「い」、「c」=「ま」、「d」=「す」とすると、「ははは+は=います(母はいます)」というワード覆面算ができます。

▼リスト7 ワイルドカード法で覆面算を生成する

```
// r段めについて、解をパターン化する
std::string Fukumenzan::pattern(int r) {
    // 各数値がどの文字に割り当てられているか
    std::map<int, char> assign;
    for (int i = 0; i < field.size(); ++i) {
        for (int j = 0; j < field[i].size(); ++j) {
            if (problem[i][j] == '?') continue;
            assign[field[i][j]] = problem[i][j];
        }
    }

    // r段めの各文字を置き換えていく
    std::string str = problem[r];
    char new_char = 'a';
    for (int j = 0; j < str.size(); ++j) {
        if (assign.count(field[r][j])) {
            str[j] = assign[field[r][j]];
        }
        else {
            assign[field[r][j]] = new_char;
            str[j] = new_char;
            ++new_char;
        }
    }

    // 一の位が右側に来るように反転して返す
    std::reverse(str.begin(), str.end());
    return str;
}

int main() {
    // 解を求める部分は略

    // パターンごとにカウントする
    int wildcard_row = 2;
    std::map<std::string, int> num;
    for (int i = 0; i < res.size(); ++i) {
        num[res[i].pattern(wildcard_row)]++;
    }
    for (auto it : num) {
        if (it.second != 1) continue;
        std::cout << it.first << std::endl;
    }
}
```

▼図11　2桁+1桁=3桁の覆面算のすべて

```
   b a        b a        b a
 +   b      +   b      +   a
 -----      -----      -----
 a c c      c d c      d c c

   a a        a a        a a
 +   b      +   b      +   a
 -----      -----      -----
 b c c      c d c      d c b
```

▼図12　幾何模様覆面算の製作例

```
  A B C D E        A B C D E
  B B C D E        B B C D E
  C C C D E        C C C D E
  D D D D E        D D D D E
+ E E E E E      + E E E E E
-----------      -----------
? ? ? ? ? ?      F U T U R E
```

（パターン dbcbaE）

幾何模様覆面算

ワイルドカード法は、幾何模様覆面算と相性良好です。一例として、図12の左側に示す入力を与えてみましょう。一意解となるように「??????」を埋める方法は6,587通りありますが、その中の1つに「dbcbaE」があります。a、b、c、dの部分にそれぞれR、U、T、Fを当てはめることで、図12右側の覆面算作品が作れました。

まとめ

今回は、パズル作品を「解く」だけではなく、「作る」ほうにも焦点を当ててみました。覆面算はパズルとしておもしろいだけでなく、メッセージ性の高い見た目を追求する芸術としての側面もあり、作品を作るアルゴリズムを考える題材として格好のものです。

そして覆面算以外のパズルにおいても、作品を作る方法を考えることはアルゴリズム力を鍛えるのに有効です。パズルによってさまざまな難しさがあります。本連載では「汎用性の高いアルゴリズム手法を解説する」というスタンスを採ることが多くありました。しかしながら、各問題に固有の構造を見抜く能力を磨くことも重要です。本記事をとおして、パズルの性質を洞察し、それを活かしたアルゴリズムを考えることのおもしろさを感じていただけたならば、大きな喜びです。SD