パズルで鍛える

アルゴリズム力

第7回

虫食算と覆面算の融合!

前々回は覆面算を、前回は虫食算を扱いました。今回は、それらを融合させたパズルを見ていきます。また前回の虫食算ソルバーを改良し、超大型の虫食算も解けるようにしていきます。

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

虫食算と覆面算の融合

虫食算は、足し算や掛け算や割り算の筆算において、いくつかの数値が□で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。一方、覆面算は、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。

今回は、虫食算と覆面算とを融合したパズルを考えてみましょう。筆算を復元するためのルールを次のように定めます。なお、このようなパズルも広い意味で「虫食算」と呼ぶことが多々あります。本記事でも虫食算と呼ぶことにします。

- □や文字には0から9までのいずれかの数値が入る
- 最上位の桁の□や文字には0は入らない
- 同一の文字(□以外)には同じ数値が入る
- 異なる文字(□以外)には異なる数値が入る
- □にはほかの文字と同じ数値が入ってもよい

▼図1　虫食算と覆面算を融合したパズルの例

```
  4□F□
×  □O□
------
  F□U□
□□O□R
□□U□
------
□□□R□□□
```

▼図2　図1の虫食算の解

```
  4794
×  232
------
  9588
14382
 9588
------
1112208
```

図1は虫食算と覆面算とを融合したパズルの一例を示しています。□には、文字F、O、U、Rに入れる数値と同じ数値を入れてもよいです。さて、図1の虫食算の解は、図2で示すものに限られます。このように虫食算の作品は通常、一意解となるように作られています[注1]。

本記事では、このような覆面文字を含む虫食算を解く探索アルゴリズムを実装していきます。それだけではなく、探索アルゴリズムを工夫することによって、非常に高速なソルバーに仕上げます。具体的には、図3のような超大型虫食算[注2]を現実的な計算時間で解けるようにします。

虫食算の例

虫食算を解くアルゴリズムを考える前に、お

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。虫食算に限らず、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。

注2) 加藤徹氏による作品です。「虫食算パズル700選」(大駒誠一、武純粋也、丸尾学 著、共立出版 刊、1985年)において、問698として紹介されています。

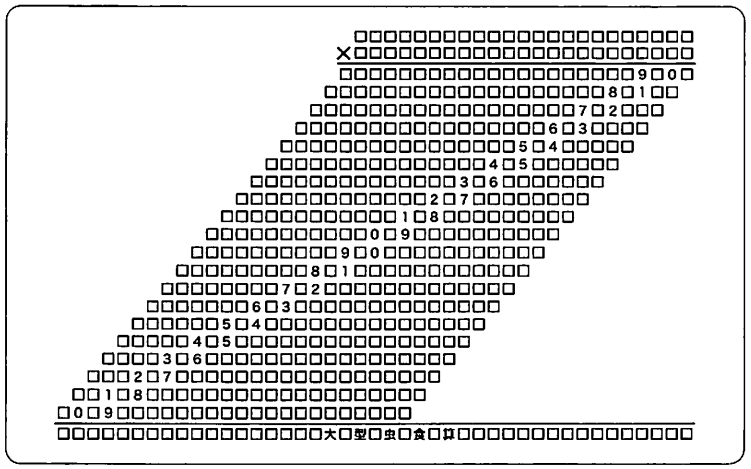
もしろい虫食算(覆面文字を含むもの)をいくつか紹介します。

図4は、カタカナ文字「ケ」を「ケ」の形状に表出した虫食算です。

図5はベンゼンの化学構造式をモチーフとした虫食算です。水素原子を表す文字「H」、炭素原子を表す文字「C」を覆面文字として活用しています。

図6は将棋における美濃囲いをモチーフとした虫食算です。将棋の各駒を表す文字「歩」「香」「桂」「銀」「金」「玉」をそれぞれ覆面文字として活用しています。

▼図3　本記事で目標とする超大型虫食算



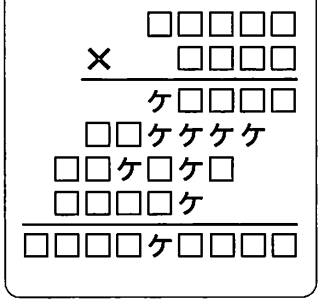
虫食算を解くアルゴリズム

それでは、虫食算を解くアルゴリズムを考えていきましょう。本記事では掛け算の筆算のみを扱いますが、割り算の筆算に対しても同様に考えることができます。また簡単のため、掛ける数(2段め)に0が入るものについては扱わないこととします[注3]。

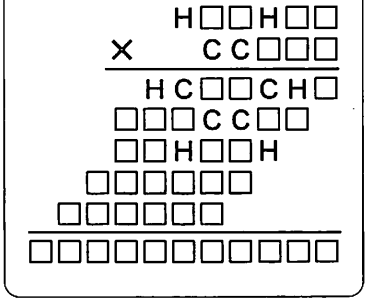
虫食算を表すグラフ上の探索

虫食算を解く探索過程は、**図7**のようなグラフで表せます。このグラフ上をバックトラッキング法により探索することで、虫食算を解くことができます。バックトラッキング法とは、グラフのより「深い」ところへと可能な限り潜っていき、これ以上深いところへは進めない状態になったら一歩戻って次の分岐を試す、という動きを繰り返す探索アルゴリズムです。バックトラッキング法を虫食算に適用した場合は、各桁の□に対して順に0から9までの数値を当てはめていき、

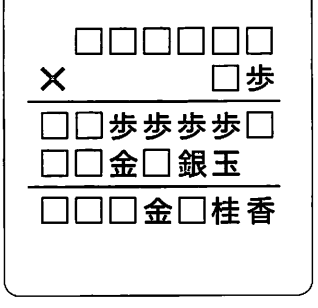
▼図4　カタカナ文字「ケ」を用いた虫食算



▼図5　ベンゼンをモチーフとした虫食算

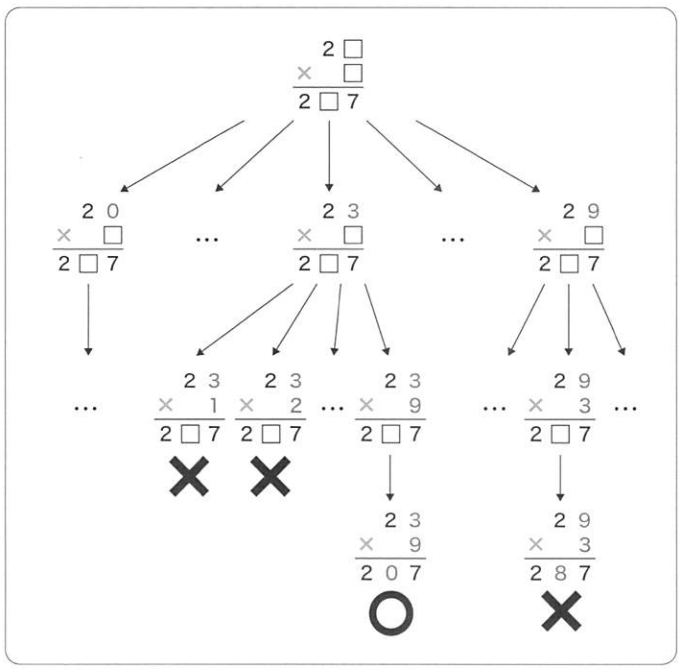


▼図6　将棋における美濃囲いをモチーフとした虫食算



注3) 2段め(掛ける数)に0が入るものについても、入力データの与え方を工夫することにより、本記事の虫食算ソルバーで解くことができます。

▼図7　虫食算を解く過程を表すグラフ



矛盾が生じたら一歩戻って次の選択肢を試していく探索アルゴリズムとなります。

探索順序の工夫

虫食算に限らず、バックトラッキング法に基づいて探索アルゴリズムを検討する際には、探索順序を工夫することが肝要です。

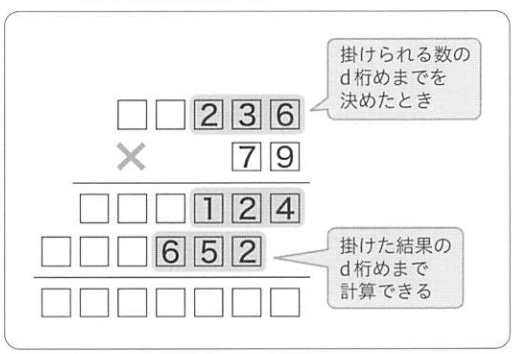
まず、掛け算の筆算においては、掛けられる数(1段め)と掛ける数(2段め)の値を決めれば、残りの部分の値がすべて決まることに注意しましょう。よって虫食算を解く方法として、次のような探索方法が考えられます[注4]。ここで、整数の一の位、十の位、百の位、……を順に0桁め、1桁め、2桁め、……と呼ぶことにします。

・掛けられる数については、すべての可能性を順に試す
・そのそれぞれについて、掛ける数の1桁め、2桁め、3桁め、……に順に数値を入れながら条件を満たすものを探索する

しかしこの方法では、たとえば図3のような超大型虫食算に対しては、掛けられる数として考えられる10^{23}通りの選択肢をすべて試すことになります。とても対処できる分量ではありません。

より効率的な方法を考えてみましょう。たとえば図8に示すように、掛けられる数のすべての□の数値を決めなくても、d桁めまでの数値を決めたならば、掛け算した結果のd桁めまでの数値も決まることに着目しましょう。よって、掛けた結果のd桁めまでに数値や文字のヒントがある場合、それを活用することで、探索の選択肢を絞ることができます。このように、解を導く見込みのない選択肢を切り捨てる考え方を「枝刈り」と呼びます。枝刈りは、実用的なアルゴリズムを設計するうえでたいへん有効な考え方です。

▼図8　掛けられる数の□をすべて埋めることなく、枝刈りするためのアイデア



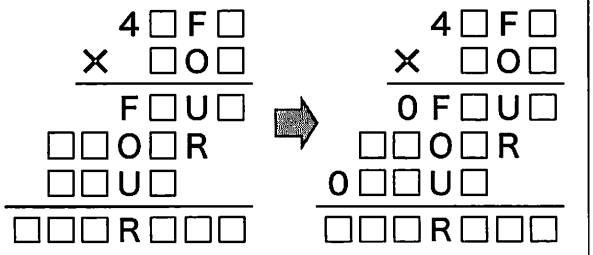
注4)　これは本誌先月号(2021年1月号)で実装した方法です。

▼図9　枝刈りの工夫を取り入れた虫食算ソルバー

掛けられる数の桁数をa、掛ける数の桁数をbとする

・For d1 = 0 to a - 1:
　・For v1 = 0 to 9:
　・可能ならば、掛けられる数のd1桁めの値をv1にする
　・For d2 = 0 to b - 1:
　　・掛けられる数のd2桁めに、すでに数値が入っている場合はスキップする
　　・掛けられる数のd2桁めに対応する計算結果のd1桁めに、数値や文字がない場合はスキップする
　　・For v2 = 1 to 9:
　　　・可能ならば、掛ける数のd2桁めの値をv2にする
・掛ける数の各桁のうち、まだ数値が入っていない桁については順に数値を入れていく
　・最後に整合性を確認し、条件を満たすならば、それを解とする

▼図10　虫食算の桁数を大きいほうに統一する



以上の考え方を取り入れることで、虫食算ソルバーは図9の探索アルゴリズムへと改良できます。次節以降で具体的に実装していきます。

虫食算ソルバーの実装

それではいよいよ、虫食算ソルバーを実装しましょう。虫食算の入力はリスト1のような形式で、標準入力で受け取るものとします。リスト1は、図1の虫食算を表しています。1行めには、掛けられる数の桁数(aとします)と、掛ける数の桁数(bとします)を記入します。続く2行は、掛けられる数の情報と、掛ける数の情報を表します。さらに続くb行は、筆算の過程を表します。そして最後の行は、筆算の結果を表します。また、虫食算の□に対応するところは文字*で表しています。

▼リスト1　虫食算の入力例(図1の例)

```
4 3
4*F*
*O*
F*U*
**O*R
**U*
***R***
```

各段の桁数に関する前処理

虫食算を解く前に、探索処理を簡潔にするための前処理を行います。さて、虫食算の各段(上2段を除く)の桁数については、それぞれ次の2通りの場合があります。

・最下段以外の各段については、a桁の場合と、a＋1桁の場合とがある
・最下段については、a＋b－1桁の場合と、a＋b桁の場合とがある

ここでは図10のように、もとの虫食算の各段の桁数が小さい場合は、便宜的に最上位の値が0であるものとして、桁数の大きな虫食算として解くことにします。以上の前処理を考慮して、虫食算を解くためのクラスMushikuizanおよび、そのコンストラクタをリスト2、3のように実装しておきます。各メンバ変数や各メンバ関数の意味と実装については次節以降で示していきます。

虫食算の覆面文字に関する扱い

虫食算を解く過程における覆面文字の扱いについて考えます。先述の探索アルゴリズムにおいては、特定の文字に特定の数値を置く場面が多々発生します。このとき、その文字にすでに数値が割り当てられているならば、整合性を確認する必要があります。数値が割り当てられていないならば、その文字に新たな数値を割り当てます(ここで、ほかの文字の数値と被らないように注意します)。以上の処理を実現するためのメンバ変数として、次のものを用意します。

- std::set<int> used_：使用済みの数値を表す
- std::map<char, int> num_：各文字に割り当てられた数値を表す
- std::map<char, std::pair<int, int>> put_：各文字に数値が割り当てられた瞬間の、虫食算の段と桁を表す

まず最初の変数used_は、異なる文字に同一の数値を割り当てるのを防ぐために用います。変数num_は、各文字への数値の割り当て状況を管理するために用います。また、虫食算を解くためのバックトラッキング法において、各文字への数値の割り当てが矛盾することがわかった場合には、その割り当てを解除する必要があります。変数put_は、各文字への数値の割り当てを解除するときに活用します。これらのメンバ変数を用いて、次のメンバ関数をリスト4のように実装します。

▼リスト2　虫食算を解くためのクラス

```
class Mushikuizan {
 private:
    // 掛けられる数と、掛ける数の桁数(a、bとする)
    int a_, b_;

    // 虫食算の問題を表す情報
    std::vector<std::string> problem_;

    // 虫食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_;

    // 各文字がどの数値を表すか
    std::map<char, int> num_;

    // 各文字に数値が置かれた瞬間がいつだったか
    std::map<char, std::pair<int, int>> put_;

 public:
    // コンストラクタ
    Mushikuizan(int a, int b,
                const std::vector<std::string> &input);

    // 掛けられる数と、掛ける数の桁数を取得する
    int get_a() { return a_; }
    int get_b() { return b_; }

    // r段めのd桁めに数値があるかどうか
    bool exist_val(int r, int d) {
        return (problem_[r][d] >= '0'
                && problem_[r][d] <= '9');
    }

    // r段めのd桁めに文字があるかどうか
    bool exist_char(int r, int d) {
        return (!exist_val(r, d)
                && problem_[r][d] != '*');
    }

    // 掛ける数のd桁めにすでに数値がある場合その値を返す
    // そうでない場合は-1を返す
    int already(int d) { return field_[1][d]; }

    // r段めのd桁めに値vを置く(矛盾が生じるときはfalse)
    bool set_val(int r, int d, int v);

    // r段めのd桁めの値をリセットする
    void reset_val(int r, int d);

    // r段めの値を計算する
    std::vector<int> calc(int r);

    // 最終段階において、最下段の値を計算する
    std::vector<int> calc_answer();

    // 解を出力する
    void print();
};
```

▼リスト3　虫食算を解くためのクラスのコンストラクタ（前処理を考慮する）

```
// コンストラクタ
Mushikuizan::Mushikuizan(int a, int b,
const std::vector<std::string> &input): a_(a), b_(b) {
    // 問題の情報を取得する
    for (int i = 0; i < input.size(); ++i) {
        std::string str = input[i];
        std::reverse(str.begin(), str.end());

        // strの桁数が小さい場合
        // 便宜上1桁増やして最上位の値を0とする
        if (i == input.size() - 1) {
            // 最下段の場合
            if (str.size() == a + b - 1) str += "0";
        } else if (i >= 2) {
            // それ以外の段の場合
            if (str.size() == a) str += "0";
        }
        problem_.emplace_back(str);
        field_.emplace_back(
            std::vector<int>(str.size(), -1));
    }
}
```

▼リスト4　虫食算の各マスへ数値を割り当てる関数と、解除する関数の実装

```
bool Mushikuizan::set_val(int r, int d, int v) {
    // 掛けられる数の最上位桁には0を置けない
    if (r == 0 && d == get_a() - 1 && v == 0) {
        return false;
    }

    // 最上位桁の条件があるときに、0が入るのはfalse
    if (d == problem_[r].size() - 1
        && problem_[r][d] != '0'
        && v == 0) {
        return false;
    }

    // もともと数値があって一致しない場合はfalse
    if (exist_val(r, d)
        && problem_[r][d] - '0' != v) {
        return false;
    }

    // もともと文字がある場合
    if (exist_char(r, d)) {
        if (num_.count(problem_[r][d])) {
            // すでに置かれた数値と一致しない場合はfalse
            if (num_[problem_[r][d]] != v)
                return false;
        } else {
            // ほかの文字にvが置かれている場合はfalse
            if (used_.count(v)) return false;

            // 文字に数値を置く
            used_.insert(v);
            num_[problem_[r][d]] = v;
            put_[problem_[r][d]] = s{r, d};
        }
    }

    // 数値を置いてtrueを返す
    field_[r][d] = v;
    return true;
}

void Mushikuizan::reset_val(int r, int d) {
    // 数値の置かれていない文字がある場合、文字情報を削除
    if (exist_char(r, d)
        && put_[problem_[r][d]].first == r
        && put_[problem_[r][d]].second == d) {
        used_.erase(num_[problem_[r][d]]);
        num_.erase(problem_[r][d]);
        put_.erase(problem_[r][d]);
    }
    field_[r][d] = -1;
}
```

・bool set_val(int r, int d, int v)：r段めのd桁めに値vを置く関数（矛盾が生じるときはfalseを返す）
・void reset_val(int r, int d)：r段めのd桁めの値を解除する関数

また、残りのメンバ関数calc(int r)、calc_answer()、print()については誌面の都合から詳細を省略しますが、関心のある方は筆者のGitHub[注5]をご覧ください。

バックトラッキング法の実装

以上の虫食算クラスMushikuizanを用いて、先述のバックトラッキング法はリスト5のように実装できます。ここで、

・rec1：虫食算の掛けられる数の各桁を再帰的に処理していく関数
・rec2：虫食算の掛ける数の各桁を再帰的に処理していく関数

注5）http://github.com/drken1215/mushikui_solver

▼リスト5　虫食算を解くバックトラッキング法の実装

```
// 前方宣言
void rec1(Mushikuizan &mu, int d,
          std::vector<Mushikuizan> &res);
void rec2(Mushikuizan &mu, int d1, int d,
          std::vector<Mushikuizan> &res);

// 最下段の値をvalとしてよいかどうかを再帰的に確認する
void check(Mushikuizan &mu,
           const std::vector<int> &val,
           int d, std::vector<Mushikuizan> &res) {
    // 終端条件
    if (d == val.size()) {
        res.emplace_back(mu);
        return;
    }

    // d桁めの値をval[d]として問題なければ次の桁に進む
    if (mu.set_val(mu.get_b() + 2, d, val[d])) {
        check(mu, val, d + 1, res);
        mu.reset_val(mu.get_b() + 2, d);
    }
}

// 虫食算の掛けられる数のd桁めを再帰的に処理していく
void rec1(Mushikuizan &mu, int d,
          std::vector<Mushikuizan> &res) {
    // 最後に、最下段を確認していく
    if (d == mu.get_a() + 1) {
        auto val = mu.calc_answer();
        check(mu, val, 0, res);
        return;
    }

    if (d == mu.get_a()) {
        // 掛けられる数がすべて埋まったならば、掛けられる数の
        // まだ埋まっていないところを順に埋めていく
        rec2(mu, d, 0, res);
    } else {
        for (int v = 0; v <= 9; ++v) {
            // 掛けられる数のd桁めにvを入れる
            if (mu.set_val(0, d, v)) {
                rec2(mu, d, 0, res);
                mu.reset_val(0, d);
            }
        }
    }
}

// 虫食算の掛ける数のd桁めを再帰的に処理していく
// 掛けられる数はd1桁めまで埋まっていることを仮定する
void rec2(Mushikuizan &mu, int d1, int d,
          std::vector<Mushikuizan> &res) {
    // 終端条件
    if (d == mu.get_b()) {
        rec1(mu, d1 + 1, res);
        return;
    }
```

↗右上につづく

```
    // 掛けられる数がすべて埋まっていない、かつ
    // d1桁めにヒントがない場合はスキップして次に進む
    if (d1 != mu.get_a()
        && !mu.exist_val(d + 2, d1)
        && !mu.exist_char(d + 2, d1)) {
        rec2(mu, d1, d + 1, res);
        return;
    }

    if (mu.already(d) != -1) {
        // すでに数値が入っている場合
        auto val = mu.calc(d + 2);
        if (mu.set_val(d + 2, d1, val[d1])) {
            rec2(mu, d1, d + 1, res);
            mu.reset_val(d + 2, d1);
        }
    } else {
        for (int v = 1; v <= 9; ++v) {
            // 掛ける数のd桁めにvを入れる
            if (mu.set_val(1, d, v)) {
                auto val = mu.calc(d + 2);
                if (mu.set_val(d + 2, d1, val[d1])) {
                    rec2(mu, d1, d + 1, res);
                    mu.reset_val(d + 2, d1);
                }
                mu.reset_val(1, d);
            }
        }
    }
}
```

と定義しています。これらの関数は相互再帰となっていますので、前方宣言が必要です。また、掛けられる数と掛ける数の数値がすべて決まった状態で、最下段の値を計算して整合性を確認する関数checkも実装しておきます。最後に、虫食算ソルバー全体(入力データの受け取りを含む)はリスト6のように実装できます。

虫食算ソルバーの活用

これまでに実装してきた虫食算ソルバーに、リスト1の入力を与えてみましょう。その結果は図11のようになります。確かに一意解であることが見て取れます。筆者の手元の環境[注6]では、0.20秒の計算時間で解が得られました。

注6) MacBook Air(13-inch, Early 2015)、プロセッサ：1.6 GHz Intel Core i5、メモリ：8GB

▼リスト6　虫食算ソルバー全体の実装

```
// 虫食算を解く
std::vector<Mushikuizan> solve(Mushikuizan &mu) {
    std::vector<Mushikuizan> res;
    rec1(mu, 0, res);
    return res;
}

int main() {
    int a, b;
    std::cin >> a >> b;
    std::vector<std::string> input(b + 3);
    for (int i = 0; i < b + 3; ++i)
        std::cin >> input[i];

    // 虫食算を解く
    Mushikuizan mu(a, b, input);
    std::vector<Mushikuizan> res = solve(mu);

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

▼リスト7　図3の超大型虫食算の入力データ

```
23 20
**********************
******************
******************9*0*
*****************8*1**
****************7*2***
***************6*3****
**************5*4*****
*************4*5******
************3*6*******
***********2*7********
**********1*8*********
*********0*9**********
*********9*0**********
********8*1***********
*******7*2************
******6*3*************
*****5*4**************
****4*5***************
***3*6****************
**2*7*****************
*1*8******************
0*9*******************
****************A*B*C*D*E****************
```

▼図11　リスト1(図1)の虫食算の解

```
0 th result:
-----------
4794
232
9588^14382
9588
1112208
```

▼図12　リスト7(図3)の超大型虫食算の解

```
504708514851473106930 69
249953758728968594 23
151412554455441932079207
100941702970294621386138
201883405940589242772276
454237663366325796237621
252354257425736553465345
403766811881178485544552
302825108910883864158414
454237663366325796237621
403766811881178485544552
100941702970294621386138
353295960396031174851483
403766811881178485544552
252354257425736553465345
353295960396031174851483
151412554455441932079207
252354257425736553465345
454237663366325796237621
454237663366325796237621
201883405940589242772276
100941702970294621386138
126153790349641171480451239036935559343918 7
```

さらに、図3の超大型虫食算も解いてみましょう。リスト7に示す入力を与えると、その結果は図12のようになります。たいへん大きなサイズの問題ですが、わずか0.38秒の計算時間で解が得られました。

まとめ

今回は、虫食算を解く探索アルゴリズムとして、高速なものを示しました。そのために用いた工夫は「バックトラッキング法において、探索順序を工夫する」という汎用的なものでした。この工夫は、虫食算に限らず、さまざまな探索アルゴリズムの効率化のために活用できます。

本記事を通して、探索アルゴリズムを設計することのおもしろさを感じていただたり、さらにそれらの知見を活かして実際のさまざまな問題の解決に役立てていただいたりしたならば、大きな喜びです。SD