国際大学対抗プログラミングコンテストの問題から学ぶアルゴリズム(2) —探索（基本編）—

Basic Search Algorithm from Problems of International Collegiate Programming Contest

ネットワーク情報学部　松永賢次
School of Network and Information　Kenji MATSUNAGA

Keywords: Backtrack, Depth First Search

1　はじめに

第 11 号より，International Collegiate Programming Contest（国際大学対抗プログラミングコンテスト, ICPC）でよく出題されるアルゴリズムを，既出問題を例題として解説してきている [1]。今回は，最もよく出題されるアルゴリズムである「探索」を取り上げる。

特定の問題向けのアルゴリズムとは異なり，探索は汎用的に利用できるアルゴリズムである。そのためプログラマは個別の問題ごとに，探索アルゴリズムを問題に適切にあてはまるようカスタマイズしていかなければならない。そのとき，問題をいかにデータとして表現していくのか，効率よく計算していくにはどのように計算順序を決めていけばよいのか，といった意思決定がプログラマに求められる。プログラミングコンテストの出題側の立場に立つと，問題設定を少し変えるだけで，難易度を微妙に変えることができるので，出題しやすいアルゴリズムだと言える。

実際，国内予選におけるアジア地区予選進出ボーダーラインの問題（3 問解けると通過できる可能性が高いので，やさしい方から 3 問目という意味）に，探索問題が出題される確率が高い。アジア地区予選でも，探索問題が解けると入賞できるというケースが多い。プログラミングコンテストで，良い成績を収めるためには探索問題をマスターすることは必須である。

本稿では，まず 2 節で探索アルゴリズムの基本的な考え方を，縦型探索を中心として示す。3, 4 節では縦型探索を利用する ICPC の過去問を解説していく。5 節では，縦型探索の特徴をまとめ，縦型探索を利用することが適当な問題について述べる。最後に，自分で解いてみたい人向けの問題の紹介，及びより高度な問題の解説を紹介する。

2　探索アルゴリズムの基本的な考え方

総数，総和，最大，最小を求めるときの問題に探索が使われることが多い。探索アルゴリズムがなぜ必要となるのか理解できるようにするため，例を示しながら述べていく。

2.1　単純な Generate and Test アルゴリズム

例題として，「4 桁以下の正の整数の中で，各桁の値の合計が 10 になる数がいくつあるのか（総和）を求める」ことを考えよう。最も単純な考え方は，1 から 9999 までの数に対して順に，各桁の値の合計が 10 になるのか調べていくことである。C 言語で記述すると，プログラム 1 のようになる。

```
/* プログラム 1 */
int check(unsigned int n);
int main(void)
{
  unsigned int n;
  unsigned int num = 0; /* 総和を求めるカウンター */
  for(n = 1; n <= 9999; n ++ ) {
    if(check(n)) {
      num++;
    }
  }
  printf("%d\n", num); /* 結果の出力 */
  return 0;
}

int check(unsigned int n)
{
  const int answer = 10;
  int num = 0;
  while(n > 0) {
    num += (n % 10);
    if(num > answer) return 0; /* 失敗 */
    n /= 10;
  }
  return num == answer;
}
```

4 桁程度だとこのプログラムはすぐに答えを返すが，もし 9 桁以下ということになると，1 から 999999999 まで調べなければならないため，途端に遅くなって 40 秒ほどか

かってしまう[1]。「失敗」とコメントに記述している行がなければ70秒かかる。このようなプログラムは，すべての候補を生成(generate)して，条件に合致しているかテストをするのでGenerate and Test（生成検査）と呼ばれる。

2.2 生成の工夫による効率化

プログラム1では,「x桁以下の正の整数」すべてを生成している。しかし問題領域の知識をうまく活用すれば，無用な候補を生成せずに済み，計算を早く終わらせることができる。

具体的な例をあげて説明する。整数19の各桁の合計は10である。19にさらに上位の桁へ値が追加された整数（例えば119, 519）は，桁の合計が10を上回ってしまう。正の数を加えても合計値が減ることはない。このことを知っていると，1の桁から上位の桁に向かって順に数をあてはめていき，10以上になったところでやめる，という考えが浮かぶ。以下のプログラムはこの考えに基づいて記述したものである。各桁の値を入れる変数は，a, b, c, dなどとせずに配列にしてみた。

```
/* プログラム2 */
unsigned int d[10]; /* 各桁の値を格納する配列 */
unsigned int v[10]; /* d[0] から d[i] の値の合計を v[i]
に格納する */

int main(void)
{
  unsigned int num = 0; /* 総和を求めるカウンター */
  for(d[0] = 0; d[0] <= 9; d[0]++){/* 1の桁 */
    v[0] = d[0];
    for(d[1] = 0; d[1] <= 9; d[1]++){/* 10の桁 */
      v[1] = d[1] + v[0];
      if(v[1] == 10) { /* 成功 */
        num ++;
      }
      if(v[1] >= 10) break; /* 成功又は失敗 */
      for(d[2] = 0; d[2] <= 9; d[2]++){/*100の桁*/
        v[2] = d[2] + v[1];
        if(v[2] == 10) { /* 成功 */
          num ++;
        }
        if(v[2] >= 10) break; /* 成功又は失敗 */
        for(d[3] = 0; d[3] <= 9; d[3]++){/*1000*/
          v[3] = d[3] + v[2];
          if(v[3] == 10) { /* 成功 */
            num ++;
          }
          if(v[3] >= 10) break; /* 成功又は失敗 */
        }
      }
    }
  }
  printf("%d\n", num); /* 結果の出力 */
  return 0;
}
```

プログラム2は4桁以下の正の整数を対象とするため，4重ループで構成されている。コメントで「成功又は失敗」と書かれているところが，このプログラムを効率よく動作させる肝のところである[2]。各桁の合計が10以上になったところで，ループを脱出している。先ほど19には，桁を追加する必要がないと述べた。そのことを忠実にプログラムにするならば，`continue`（次の値にスキップ）とすべきであるが，実は，29, 39, ... と10の位の値をこれ以上大きくしても失敗することが明白なので，`break`（ループを脱出）と記述している[3]。

このように，対象領域の知識をうまく利用することで，無駄な生成を抑えることができる。もしループを`for(d[0] = 9; d[0] >= 0; d[0]--)`のように逆順に回したらどうだろうか？これだと無駄な生成を抑えることはできない。生成の順序が，無駄な生成を抑えるために重要な役割を演じているということである。

2.3 縦型探索

プログラム2は4桁では有効だったが，9桁以下の整数に対して求めるにはどのようにしたら良いだろうか？プログラム1では，1箇所変更するだけで対応できたが，プログラム2では9重ループへと大幅に追加が必要となる。ループの内容を関数とし，その関数を再帰呼び出しする方法をとることで，桁数が変わっても変更をほとんど必要としないプログラムにできる。

```
/* プログラム3 */
unsigned int d[10]; /* 各桁の値を格納する配列 */
unsigned int num = 0; /* 総和を求めるカウンター */
unsigned int len = 9; /* 最大桁数 */
void  dfs(const int digit, const int sum);

int main(void)
{
  num = 0;
  dfs(0, 0);
  printf("%d\n", num);
  return 0;
}

void  dfs(const int digit, const int sum)
/* digit: 何桁目, sum: それまでの合計 */
{
  if(digit >= len) return; /* 失敗 */
  for(d[digit] = 0; d[digit] <= 9; d[digit]++) {
    /* digit桁目の数の生成 */
    int newsum = sum + d[digit];
    if(newsum == 10) { /* 成功 */
      num ++;
    }
    if(newsum >= 10) {/* 枝刈り（計算を戻す） */
      return;
    }
    dfs(digit + 1, newsum); /*再帰呼び出しで次に */
  }
  return;
}
```

[1] ネットワーク情報学部の学生が追体験しやすいよう，ここでの実行時間は，www.ne.senshu-u.ac.jpのgccを使用してコンパイルしたものを示している。

[2] 先をさらに調べるのをやめることから「枝刈り」と呼ばれる。

[3] さらに，各桁の合計が10なったところでループを脱出していれば，11以上になることはないので，`num++`と書かれた行の後に`break`を追加してもよい。

0 → 00 → 000 → 0000 → 00000 ×（桁オーバー）
→ 10000 ×（桁オーバー）
…
… → 9000 → 19000 ×（桁オーバー）
→ 29000 ×（桁オーバー）
…
… → 100 → 0100 → 01000 ×（桁オーバー）
→ 11000 ×（桁オーバー）
…
… → 9100 ○成功
枝刈りするので09100以降は試さない
→ 200 → 0200 → 01000 ×（桁オーバー）
→ 11000 ×（桁オーバー）
…
… → 8200 ○成功
枝刈りするので08200以降は試さない
枝刈りするので9200以降は試さない
→ 300
以下同様に継続

図 1: プログラム 3 を最大 4 桁として実行したときの計算過程

プログラム 3 では len に代入する値を代えるだけで，最高桁数を変えることができる。9 としたときの実行時間は 1 秒もかからない。プログラム 1 では 40 秒かかっていたのだから，このプログラムの効率が良いのは明らかである。図 1 に，プログラム 3 を最高桁数 4 のときに実行したときに，どのように計算が進むか示してある[4]。最高桁数 4 のときにはわかりにくいが，もし最高桁数が大きくなれば，9 桁すべてを生成しなくてもよい場合がかなりあることが想像できるであろう。

このプログラムは，縦型探索と呼ばれる探索アルゴリズムとなっている[5]。縦型探索アルゴリズムの基本的な形は次のようになる。これを問題毎にカスタマイズしてプログラムを作成していくことになる[6]。

```
/* 縦型探索の一般形 */
State state; /* 状態を表す State 型の変数 state */

void  dfs(const int depth) /* 必要があれば引数として状態データを渡す */
{
  if(成功したか?) {
    成功時の処理;
    return ;
  }
  if(失敗することが確実か?) return;
  if(depth が最大深さを超えたら) {
    /* 必要があれば成功かどうか判断して，成功時と失敗時の処理をする */
    return;
  }
  次の可能性たちそれぞれに対して {
    状態を次の可能性に変更する;
    dfs(depth + 1); /*次に進む */
    状態を元に戻す;
  }
  return;
}
```

「状態」とは，計算途中の状況を示すデータ構造である。これまでの例では，d[], v[] といった配列が相当する[7]。

縦型探索を使って問題を解くということは，対象領域の知識を利用して，

1. 次の候補の生成の方法
2. 成功の条件
3. 失敗の判断

[4]プログラム 3 では，各桁を配列に格納している。図 1 では，下の桁から順に求めていると仮定しているが，上の桁から求めていると考えてもよい。

[5]深さ優先探索 (Depth First Search)，バックトラック (Backtrack) とも呼ばれる

[6]これまでの例題では「成功」,「失敗」の判断を生成の繰り返し内で行っているが，ここで示した一般形は関数の先頭で行っている。これは，最初に関数に入ったときに，成功，失敗を判断する必要がある問題かどうかに依存している。

[7]再起呼び出しを使ったプログラムでは，関数の引数で渡される値も状態を示す値となる。配列 v[] が不要になっているのは，関数引数がその役割を負っているからである。

4. 枝刈りの条件
5. 状態の表現方法

を考えることである。以下3節，4節では，ICPCの過去問を利用して，具体的に縦型探索を使って問題を解き，プログラムを作成するのか示す[8]。

3 例題1：単位分数への分割

2004年アジア地区予選愛媛大会国内予選問題C "Unit Fraction Partition" を例に解説する。

3.1 問題の概要

この節の問題記述は，http://www.acm-japan.org/past-icpc/domestic2004/C.jp/C.html に記載されているものである[9]。

問題記述

分子が1であり，分母が正の整数である分数を単位分数と呼ぶ。正の有理数 $\frac{p}{q}$ を有限個の単位分数の和で表現したものを，$\frac{p}{q}$ の単位分数への「分割」と呼ぶ。例えば，$\frac{1}{2}+\frac{1}{6}$ は $\frac{2}{3}$ の単位分数への分割である。足し算の順序の違いは無視する。例えば，$\frac{1}{6}+\frac{1}{2}$ を $\frac{1}{2}+\frac{1}{6}$ と区別せず，同一のものとして考える。

与えられた四つの正整数 p, q, a, n に対して，$\frac{p}{q}$ の単位分数への分割で，以下の二つの条件を満たすものの個数を数えなさい。

- 分割は n 個以下の単位分数の和である。
- 分割に含まれる単位分数の分母の積は a 以下である。

例えば，$(p, q, a, n) = (2, 3, 120, 3)$ ならば，4と出力しなければならない。なぜならば，

$$\begin{aligned}\frac{2}{3} &= \frac{1}{3}+\frac{1}{3}\\ &= \frac{1}{2}+\frac{1}{6}\\ &= \frac{1}{4}+\frac{1}{4}+\frac{1}{6}\\ &= \frac{1}{3}+\frac{1}{6}+\frac{1}{6}\end{aligned}$$

で，条件を満たす分割が尽くされるからである。

入力は，200個以下のデータセットの並びで，その後に終端行が続く。

データセットは四つの正の整数 p, q, a, n を含む行で，$p, q \leq 800$ かつ $a \leq 12000$ かつ $n \leq 7$ を満たす。それらの整数は空白で区切られる。

終端行は，空白で区切られた四つのゼロを含む1行で構成される。これは入力データの一部ではなく，入力終了を表す印（しるし）である。

入力例

```
2 3 120 3
2 3 300 3
2 3 299 3
2 3 12 3
2 3 12000 7
54 795 12000 7
2 3 300 1
2 1 200 5
2 4 54 2
0 0 0 0
```

入力例に対する出力結果

```
4
7
6
2
42
1
0
9
3
```

3.2 考え方

この問題は，分割する単位分数の個数 n の最大が7であるので，7重ループを使って解くことも可能であるが，縦型探索を使って解いてみる。

生成の方法

$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \cdots$ と分母の値を一つずつ増やしながら，単位分数を生成していく。問題文の第一段落に，足し算の順序の違いを無視し，$\frac{1}{2}+\frac{1}{6}$ と $\frac{1}{6}+\frac{1}{2}$ を区別しない，とある。このように生成すると，$\frac{1}{6}$ の後に $\frac{1}{2}$ は生成されない（$\frac{1}{6}$ の後には，$\frac{1}{6}, \frac{1}{7}, \frac{1}{8}, \cdots$ と生成していくので）[10]。単位分数の生成の終了条件は，問題文にある「分割に含まれる単位分数の分母の積は a 以下である」にしたがえばよい。

成功の条件

単位分数の合計が，$\frac{p}{q}$ と一致したとき。

[8] 3節，4節をどちらから読んでも構わない。わかりやすい対象問題から先に読むと良いであろう。

[9] 日本のICPCの過去問は，ACM Japan Chapterの Webページ (http://www.acm-japan.org/) から，「Programming Contest」を選択し，「過去の大会」を選ぶと見ることができる。また最近の問題は，Peking University JUDGE ONLINE FOR ACM/ICPC(http://acm.pku.edu.cn/JudgeOnline) で，Problemを選択し，そのページで，Sourceとして"Japan"を入力して検索すると見ることができる。また，作成したプログラムがあっているかどうか判定してもらうこともできる。

[10] 単位分数の和は，大きい分数（分母が小さい）から小さい分数（分母が大きい）順に加算したもののみを考えると，一つの形（正規形）に決まる。このように正規形を決めて生成するとよい。

失敗の条件

1. 単位分数の数（探索の深さ）が n を超えたとき。
2. 分母の積が a を超えたとき。
3. 単位分数の合計が $\frac{p}{q}$ を超えたとき。

枝刈り

失敗条件 2 の場合，さらに単位分数の分母の値を大きくした場合でも，同様に失敗する。これ以上，分母の値を大きくしたものを試さない。

状態表現

分子は常に 1 と決まっているので，使用した分母を記憶しておけばよい。この問題では，生成を開始する分母値として，一つ前の分母だけを覚えておけばよいので，関数の引数 (old) として受け渡すことにする[11]。また，右辺の計算結果の値も覚えておかなければならない。右辺の分母と分子を，約分せずに覚えることにする。これは，単位分数の積が a 以下かを判定するためであり，左辺と右辺の値の比較のためには，約分をしなくても，分母と分子をたすき掛けで掛け算したもの同士を比較すればよい。変数の上限を考慮して，分母と分子を掛け合わせても，int 型の範囲で表される上限値を超えないからである。この二つの値も，一つ前だけ覚えていればよいので，関数の引数 (bunbo, bunshi) として受け渡す。

3.3 プログラム

3.2 節で示した考え方に基づいて記述した C 言語のプログラムを示す。

```
/* 単位分数への分割を解くプログラム */
#include <stdio.h>

int num; /* 総和 */
int p, q, a, n;
int data[16]; /* 状態：単位分数の分母 */

int compare(const int lbunshi, const int lbunbo,
            const int rbunshi, const int rbunbo)
{
  int a = lbunshi * rbunbo,  b = lbunbo * rbunshi;
  if( a > b ) return 1;
  if(a == b) return 0;
  return -1;
}

void  dfs(const int depth, const int old,
       const int bunbo, const int bunshi)
{
  int kouho;
  if (depth > n) return; /* 失敗条件 1 */
```

[11] 総和を数えろではなくて，すべて出力せよだと配列にして覚えておく必要がある。

$\frac{1}{1}$　失敗条件 3
$\frac{1}{2}$
$\frac{1}{2}+\frac{1}{2}$　失敗条件 3
$\frac{1}{2}+\frac{1}{3}$　失敗条件 3
$\frac{1}{2}+\frac{1}{4}$　失敗条件 3
$\frac{1}{2}+\frac{1}{5}$　失敗条件 3
$\frac{1}{2}+\frac{1}{6}$　成功
$\frac{1}{2}+\frac{1}{7}$
$\frac{1}{2}+\frac{1}{7}+\frac{1}{7}$　失敗条件 3
$\frac{1}{2}+\frac{1}{7}+\frac{1}{8}$　失敗条件 3
$\frac{1}{2}+\frac{1}{7}+\frac{1}{9}$　失敗条件 2（枝刈り）
$\frac{1}{2}+\frac{1}{8}$
$\frac{1}{2}+\frac{1}{8}+\frac{1}{8}$　失敗条件 2（枝刈り）
$\frac{1}{2}+\frac{1}{9}$
$\frac{1}{2}+\frac{1}{9}+\frac{1}{9}$　失敗条件 2（枝刈り）
…（途中省略）
$\frac{1}{2}+\frac{1}{61}$　失敗条件 2（枝刈り）
$\frac{1}{3}$
$\frac{1}{3}+\frac{1}{3}$　成功
$\frac{1}{3}+\frac{1}{4}$
$\frac{1}{3}+\frac{1}{4}+\frac{1}{4}$　失敗条件 3
$\frac{1}{3}+\frac{1}{4}+\frac{1}{5}$　失敗条件 3
以下同様に計算が進んでいく

図 2: 3.3 節のプログラムでの $(p, q, a, n) = (2, 3, 120, 3)$ のときの計算過程

```
  for(kouho = data[depth-1]; ; kouho ++ ) {
    int result, newbunbo, newbunshi;
    newbunbo = bunbo * kouho; /* 右辺の分母の計算 */
    if(newbunbo > a) /* 失敗条件 2 */
       return; /* ループを脱出して枝刈り */
    newbunshi = kouho * bunshi + bunbo * 1; /* 右 辺
の分子の計算 */
    result = compare(p, q, newbunshi, newbunbo);
    if(result == 1) { /* p/q が小さい */
      dfs(depth + 1, kouho, newbunbo, newbunshi);
    } else if (result == 0) { /* 成功 */
      num ++;
    } /* 失敗条件 3 */
  }
}

int main(void)
{
  while(1) {
    scanf("%d %d %d %d\n", &p, &q, &a, &n);
    if( p == 0 && q == 0 && a == 0 && n == 0) break;
    num = 0; /* 結果の初期化 */
    dfs(1, 1, 1, 0); /* 縦型探索 */
    printf("%d\n", num); /* 結果の出力 */
  }
  return 0;
}
```

このプログラムを，$(p, q, a, n) = (2, 3, 120, 3)$ として実行したときの計算過程を図 2 に示す。

この問題の解説は [2] にもあるので参照されたい。

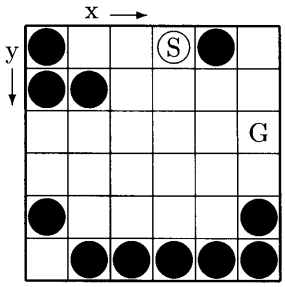

図 3: カーリング盤の例 (S はスタート，G はゴール)

4 例題：カーリングゲーム

2006 年アジア地区予選東京大会国内予選問題 D "Curling 2.0" を例に，一人ボードゲームの問題を解説する。

4.1 問題の概要

問題記述

この節の問題記述は，http://www.acm-japan.org/past-icpc/domestic2006/contest/all_ja.html#section_D に記載されているものである。

カーリングのようなゲームが与えられる。図 3 のように，2 次元のマスからなる盤面を考える (x 方向のマスの個数と y 方向のマスの個数が決まっている)。マスの中には，障害物（黒丸）が置いてある場所がある。石（白丸）が置かれたマス（スタート S）から目的のゴール (G) まで，次のように移動できるとして，その最短移動回数を求める。

- ゲームの開始時に，石はスタート地点で止まっている。
- 石の動きは x, y 方向に限る。ななめには動けない。
- 止まっている石は，滑らせることによって動き出す。その時の方向は，すぐ隣のマスに障害物がない方向ならどちらでもよい (図 4(a))。
- 動き出した石は，次のいずれかが起こるまで，同じ方向に動き続ける。
  - 障害物にぶつかった場合 (図 4(b), (c))。
    - 石は障害物の一つ手前のマスで止まる。
    - 障害物は消滅する。
  - 盤外に出た場合。
    - 失敗でゲームは終る。
  - ゴールの上に来た場合。
    - そこで石は止まり，成功でゲームは終わる。
- 1 回のゲーム中の滑らせる回数の最大は 10 である。この回数でゴールに石を到達させることができない場合，失敗でゲームは終わる。

図 3 に示す初期配置では 4 回で石をスタート位置からゴール位置に動かすことができる。そのときの経路を図 5(a) に示す。なお，石がゴールに到達した時点での盤面は図 5(b) のように変化している。

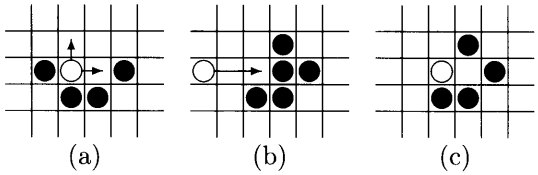

図 4: カーリングの石の移動

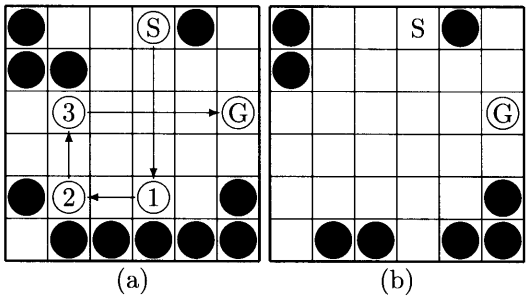

図 5: 図 3 の解とそのときの最終盤面配置

入力

入力はデータセットの並びである。入力の終わりは，一つの空白で区切られた二つのゼロからなる行で示される。データセット数が 100 を超えることはない。

各データセットは次のような形式をしている。

```
盤の幅 (=w) 盤の高さ (=h)
盤面の 1 行目
...
盤面の h 行目
```

盤の幅と高さは，$2 \leq w \leq 20, 1 \leq h \leq 20$ の条件を満たす。

盤面の各行には，w 個の数字が空白一つをはさんで並んでいる。その数字は対応するマス目の状態を示している。

```
0  何もないマス
1  障害物
2  スタート位置
3  ゴール位置
```

図 3 に対応するデータセットの記述は以下のようになる。

```
6 6
1 0 0 2 1 0
1 1 0 0 0 0
0 0 0 0 0 3
0 0 0 0 0 0
1 0 0 0 0 1
0 1 1 1 1 1
```

出力

各データセットが指定する盤面について，スタート位置にある石をゴール位置に到達させるまでに滑らせる回数の最小値を，十進の整数値でそれぞれ1行に出力せよ。そのような移動ができない場合には，-1 を出力せよ。

4.2 考え方

この問題は，最短を求めなければならない。縦型探索を用いて最短を求めるためには，最悪の入力値の場合，すべての可能性を調べつくさなければならないので，決して計算効率はよくない[12]。次の理由から，全部調べても大きく計算時間がかかることはない。このゲームでは，移動の可能性は四つしかない（石は障害物にぶつかるか，外に出るまで進むという点が重要である。途中で止まれるのであれば，移動の可能性は多くなる）。したがって，調べなければならない状態数は，最大移動回数が10回なので，どんなに多くとも4の10乗，すなわち2の20乗でおよそ100万である。100万程度ならば縦型探索で調べつくすことは，コンテストで許されている計算時間内でできる。

このような判断をするのは，プログラミングコンテストが，プログラミング時間を短く，なおかつ計算時間が所定内に終わるようにすることを目的としているからである。計算時間のベストを求めるために，プログラミング時間が大幅にかかっては損をするからである。ベストを求めるのか，ベターを求めるのかコンテストの最中に適切に判断するためにも，正確なアルゴリズムの知識とプログラミングの知識が必要となる。

生成の方法

石は，四つの方向（x 方向，$-x$ 方向，y 方向，$-y$ 方向）に進むことができ，障害物にぶつかるまで移動できる。また，障害物にぶつからずに外に出た場合は，失敗なので次の候補としては生成しない。

成功の条件

ゴールを通過したとき。

失敗の条件

- 移動回数が10を超えたとき。
- 外へ出たとき。

[12]最短移動回数が小さい入力値に対しては，横型探索を用いる方が，縦型探索よりは計算効率がよい可能性が高い。もし解がない入力値に対しては，調べる状態数は同じになるので，プログラムが単純な縦型探索の方が，横型探索よりは計算が速いであろう。

枝刈り

それまで得られた最短と同じ移動回数になった場合，それ以上調べても，最短を更新することはできないので，枝刈りをする。

状態表現

マス目上のゲームなので，2次元配列を使って状態を表現する。2次元のゲームを表現するときに，ボードの周囲を一マス分余計に確保しておくことがよく行われる。これは場外の判定を簡略化するテクニックである。通常，場外かどうかを判定するためには，x と y が範囲に収まっているかどうか，四つの条件すべて調べなければならない。場外用のマスを作成しておき，そこに場外を意味する数字を入れておけば，マスの内容を判定する一つの条件のみで，場外かどうか判断できる。

4.3 プログラム

4.2で述べた考え方に基づいて記述した，C言語のプログラムを示す。その前に，導入したいくつかのプログラミングテクニックを説明しておく。

1. マス目の中身を意味する整数（例えば障害物の1）は，プログラムでわかりやすいように定数宣言をしている。ここでは `enum` を使ってみたが，`#define` でマクロ宣言する方法でもよい。
2. ボード面を初期化する関数 `init` では，最初にマス目を「縁として」埋めている。後で，ボードデータが読み込まれるので，読み込まれるデータにないものを，最初に埋めておくとよい。
3. 関数 `dfs` の中で，四つの方向（`x` 方向，`-x` 方向，`y` 方向，`-y` 方向）を試すとき，それぞれ別個に並べて記述するとプログラムが長くなる。また，もしプログラムに誤りが見つかったときに，複数箇所修正することになりかねない。そこで，4回ループすることで試せるように工夫している。

```
/* x 方向の移動距離配列 */
static int dir_x[ ] = {1, -1, 0, 0};
/* y 方向の移動距離配列 */
static int dir_y[ ] = {0, 0, 1, -1};
```

のように，単位移動距離を配列に入れることで，そのようなことができるようになっている。

4. 関数 `dfs` を再帰呼び出しして戻ってきたときに，ボード状態 `space[][]` の内容を元に戻す操作をしている。

```
space[x][y] = VACANT; /* BLOCK を取り除く */
dfs(depth + 1, x - dir_x[d], y - dir_y[d]);
space[x][y] = BLOCK; /* 元の状態に戻す */
```

再帰呼び出しする前には space[x][y] は石があった(値が BLOCK)。それを取り除いて（値を VACANT にして）再帰呼び出ししたので，関数から戻ってきた場合に，石を戻さないとならない（値を BLOCK にする）[13]。先ほどの例題のように，状態表現が整数の場合は，関数へ引数として値を渡して再帰呼び出しをしている。C 言語では，関数引数は値渡しになるので，関数から戻ってくると変数の値は元に戻ることになっている。この例題の二次元配列 space[][] は，グローバル変数なので，自分で変数の値を元に戻す必要がある。同じように，関数引数として渡すことはできないのか，という疑問があると思うが，二次元配列のデータを，関数呼び出しのたびにコピーすることになってしまい，その結果として計算時間がかかってしまう，というマイナスがある。

```
/* カーリング 2.0 を解くプログラム */
#include <stdio.h>

#define MAX 128
#define MAX_DEPTH 10

enum {
  VACANT = 0, /* 氷 */
  BLOCK  = 1, /* 障害物 */
  START  = 2, /* スタート */
  GOAL   = 3, /* ゴール */
  EDGE   = 4 /* 縁 */
};

int min_depth;
int space[MAX][MAX];

void init(void)
{
  /* 最初にすべて縁に初期化する  */
  int x, y;
  for(x = 0; x < MAX; x ++ ) {
    for(y = 0; y < MAX; y ++ ) {
      space[x][y] = EDGE;
    }
  }
  return;
}
void dfs(const int depth,
         const int current_x, const int current_y)
{
  int i, d, x, y, step;
  /* x 方向の移動距離配列 */
  static int dir_x[ ] = {1, -1, 0, 0};
  /* y 方向の移動距離配列 */
  static int dir_y[ ] = {0, 0, 1, -1};
  if(depth > MAX_DEPTH) return; /* 失敗 */
  if(depth >= min_depth) return; /* 枝刈り */
  for(d = 0; d < 4; d ++ ) {
    x = current_x; y = current_y;
    for(i = 0; ; i ++) {
      x = x + dir_x[d]; y = y + dir_y[d];
      if(space[x][y] == GOAL) { /* 成功 */
        step = depth + 1;
        if(step < min_depth) min_depth = step;
        return;
      } else if (space[x][y] == EDGE) break; /* 石が
場外へ */
      else if (space[x][y] == BLOCK) {
        if(i > 0) { /* すぐ隣が BLOCK 以外のとき */
          space[x][y] = VACANT; /* BLOCK  を取り除
く */
          dfs(depth + 1, x - dir_x[d], y - dir_y[d]);
          space[x][y] = BLOCK; /* 元の状態に戻す */
        }
        break;
      }
    }
  }
  return;
}

void solve (const int size_x, const int size_y)
{
  int x, y, start_x, start_y, kind;
  /*  盤情報の読み込み */
  for(y = 1; y <= size_y; y ++ ) {
    for(x = 1; x <= size_x; x ++ ) {
      scanf("%d", &kind);
      space[x][y] = kind;
      switch (kind) {
      case START: start_x = x; start_y = y;
                  space[x][y] = VACANT; break;
      }
    }
  }
  min_depth = MAX_DEPTH + 1; /* 結果の初期値 : あり得な
い最大値 */
  dfs(0, start_x, start_y);
  if (min_depth == MAX_DEPTH + 1) printf("%d\n", -1);
  else printf("%d\n", min_depth);
}

int main(void)
{
  while(1) {
    int width, height;
    scanf("%d %d", &width, &height);
    if (width == 0 && height == 0) break;
    init();
    solve(width, height);
  }
  return 0;
}
```

[13]いわゆる「待った」をして元に戻すことと同じような操作をする。

5 おわりに

5.1 縦型探索の問題点

本稿では，最も基本的な探索アルゴリズムである縦型探索について述べた。縦型探索は万能ではなく，様々な限界がある。アルゴリズムを選択するとき，その特長と限界を理解した上で，縦型探索が適当な場合に選択できるようにしなければならない。

探索では最悪，それぞれの状態で a 個の候補が生成でき，探索の深さが n のときに a^n の状態を調べなければならなくなる。プログラミングコンテストの計算時間制限においては，数百万程度の候補数が限界である。枝刈りなどの方法を利用して，調べる状態数を減らすことはできる

が，深さの最大が10程度が，コンテストもおいて縦型探索で現実に解ける問題と思われる。

これ以上の深さを調べなければならない場合は，状態の同一性チェックをして，状態数を抑制することが考えられる。[1] で紹介した動的計画法は，この考えに基づいている。

縦型探索は，最大深さが設定されていたり，進行するにしたがって必ずある深さで計算が終わることが保証されている問題には適しているが，そうでない場合は，計算が終わらない可能性がある。プログラミングコンテストにおいて，総数，最大（最長）を求める問題は，網羅しなければならないために，縦型探索でも解けるはずである。一方，最小（最短）を求める問題は，一つでもそのようなものが見つかればよいので，縦型探索を使うと計算が終わらない可能性がある。そのような場合には，横型探索などの別の探索アルゴリズム，動的計画法，グラフの最短経路アルゴリズムが有力である。これらのアルゴリズムは，稿を改めて今後，紹介していくことにする。

5.2 さらに学習したい人のために

これまでの説明がわかりにくいと感じる人には，アルゴリズムの教科書（例えば，[3] の「バックトラック法」）を読んで補うとよい。

本稿で説明できなかった，縦型探索を使って解けるICPCの過去問を，やさしい問題から難しい問題の順に列挙する。これらを解いてみることで，縦型探索アルゴリズムを理解することができるであろう。情報処理学会が発行している情報処理に，解説があったものは，それを示したので，読んでみるとさらに理解が深まるであろう。

1. 2004 年愛媛大会国内予選 Problem B “Red and Black”（[2] に解説あり）
2. 2000 年つくば大会国内予選 Problem B “Patience”
3. 2001 年函館大会国内予選 Problem C “Jigsaw Puzzles for Computers”（[4] に解説あり）
4. 2001 年アジア地区予選函館大会 Problem D “77377”
5. 2002 年アジア地区予選金沢大会 Problem F “Shredding Company”
6. 2004 年アジア地区予選愛媛大会 Problem F “Dice Puzzle”（[5] に解説あり）
7. 1998 年アジア地区予選東京大会 Problem B “Lattice Practices”（[6] に解説あり）
8. 2006 年アジア地区予選横浜大会 Problem F “Polygons on the Grid”

参考文献

[1] 松永賢次：国際大学対抗プログラミングコンテストの問題から学ぶアルゴリズム (1) —動的計画法—，専修ネットワーク&インフォメーション, No. 11(Mar. 2007), pp. 41–50.

[2] 湯淺太一：国内予選を突破せよ，情報処理，Vol. 45, No. 12(Dec. 2004), pp. 1279–1283.(See also http://www.ipsj.or.jp/07editj/promenade/4512.pdf)

[3] 石畑清：アルゴリズムとデータ構造，岩波書店，1989.

[4] 和田英一：計算機用ジグソーパズル，情報処理，Vol. 43, No. 8 (Aug. 2002), pp. 894-902. (See also http://www.ipsj.or.jp/07editj/promenade/4308.pdf)

[5] 寺田実：サイコロパズル，情報処理，Vol. 46, No. 1 (Jan. 2005), pp. 75-79. (See also http://www.ipsj.or.jp/07editj/promenade/4601.pdf)

[6] 田中哲郎：組木細工，情報処理，Vol. 45, No. 8 (Aug. 2004), pp. 848-853. (See also http://www.ipsj.or.jp/07editj/promenade/4508.pdf)