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(3)

—探索(発展編)—

Various Search Algorithms
from 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: Breadth First Search Algorithm, Dijkstra's Shortest Path Algorithm

１．　はじめに

「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」というタイトルでこれまで，第１回に動的計画法[1]，第 2 回に探索手法の中から縦型探索（バックトラック）[2]をとりあげた。今回は前回[2]の続きを述べる。今回紹介する問題の一部では，[1]で述べた動的計画法でも解けるものもあり，関連した内容を取り上げていると考えられる。

[2]の 5.1 で述べたように，縦型探索は網羅的に調べるために，総数，あるいは最大（最長）を求める問題には向いている。同様に，最小（最短）を求める問題においても，縦型探索によってすべての可能性を調べた後，その中から最小のものを出力することができる。しかし，もし探索開始地点から，2 ステップのところにゴールがあるとすれば，開始地点から近い場所から順番に調べていくことで，ゴールに到達すればすぐに，それが最短であると判断できる。今回は，そのような近い場所から調べていく探索アルゴリズムについて述べる。

２．　一般的な探索アルゴズム

探索アルゴリズムの基本的な考え方は[2]で述べているので，ここではその内容は既知のものとして述べていく。

探索アルゴリズムでは，問題を「状態」で表現する。スタートから到達できる状態を次々と生成し，ゴール候補群に加えていく。候補群の中から１つの状態選び，それがゴールに達しているのか（成功しているのか）どうか調べ，もしゴールでなければ，選んだ状態のから到達できる状態すべてを生成し，さらに候補群に加えていくものである。

このときに，候補群の中からどの状態を選ぶのかという基準により,知られているアルゴリズムを分類できる。

1. 縦型探索。深さ優先探索という別名があるとおり，スタートからのステップ数が大きい状態から選んでいく。
2. 横型探索。幅優先探索という別名があるとおり，スタートからのステップ数が小さい状態から選んでいく。
3. ダイクストラ法。グラフの最短経路問題のアルゴリズムとして知られており，スタートからの経路長（コスト）が小さい状態から選んでいく。
4. A*アルゴリズム。将来のコスト予測関数を定義でき，かつその関数の値が，必ず実コスト以下関数の値にできるときに使用するアルゴリズム。A*アルゴリズムでは，スタートからその状態の実コストと，その状態からゴールまでの予測コストの和が小さい状態から選んでいく。

縦型探索は，バックトラックという別名があるが，これは再帰アルゴリズムによる実装方法を示した名前である。再帰計算により調べ尽くした後に，再帰関数から戻るために，このような名前がついている。

一方，再帰アルゴリズムを使用しなくても，候補群を順番に管理していくループで記述することもできる。この方法だと，上で示したすべてのアルゴリズムすべてを共通の一般化したアルゴリズムで記述することができる。

一般的な探索アルゴリズムでは候補群に加えて，ループ検出するために，既に検査済みの状態も管理する。候補群を管理するデータ構造を open，検査済みの状態を

管理するためのデータ構造を closed と呼ぶことが多い。

- open これから調べる状態が格納されているデータ構造で，優先順位にしたがって取り出せる必要がある。また，新たな状態を追加できる。
- closed これまでに調べた状態が格納されているデータ構造で，状態を生成したときに，このデータ構造に含まれているか調べられればよい。

一般的な探索アルゴリズムを擬似コードで書くと以下のようになる。

```
「スタート状態」を open に追加
while(open に状態が残っている) {
    s := open から優先順位の一番高い状態を取り出す
    if(s がゴール状態の条件を満たしていれば) 終了
    if(s が closed に存在しなければ) {
        s を closed に追加
        s から次に行ける状態すべてを生成し，
        それぞれを ss とする {
            cost = スタートから ss までのコスト
            cost と ss をペアにして open に追加
        }
}
```

３． C++言語のデータ構造

第 4 節以降，実際の問題に対する C++言語でのコードを紹介していく。データ構造は，C++の STL(Standard Template Library) にあるものを利用するので，この節ではそれについて紹介する。一般的なデータ構造の概念（配列，スタック，キュー，集合，木，ハッシュなど）[3] は既知のものとして，STL のデータ構造がどの概念と対応づけられるのか示す。

Java 言語でも同様のデータ構造を持っており，同様の記述が可能である。

4.1. vector

長さ可変の動的配列である。配列のように添字（インデックス）でアクセスできるとともに，末尾から追加・取り出しができるので，スタックとしても使用できる。
代表的な関数
[添字番号] :配列のように添字でデータを参照できる
push_back(データ) :末尾にデータを追加
back( ) :末尾のデータの参照
pop_back( ): 末尾のデータの削除
size( ) :要素数を得る
empty( ): 要素がないかどうか判定

4.2. deque

vector と同じく，長さ可変の動的配列であるが，末尾に加えて先頭からも追加・取り出しができるので，双方向キューとしても使用できる。
代表的な関数
vector で紹介した関数はすべて使用できる。先頭からのデータの出し入れのために，次の３つの関数が用意されている。
push_front(データ)
front( )
pop_front( )

4.3. set

集合のデータ構造。数学の集合の概念同様，同じ要素は１つしか格納されない。木構造を利用して実現されているので，等価関係，大小関係を格納するデータに記述する必要がある。
代表的な関数
insert(データ) :データを挿入する。
find(データ) :データを探す。本稿で記述するプログラムでは，データがあるかないかだけに関心がある場合がある。データが存在しない場合は，.end( )関数で返ってきた値と等価になる。

4.4. priority_queue

順序付きキュー。追加と取り出しができるが，取り出した際には，一番大きいものが取り出される。比較関数を与えることで，一番●●なものを取り出すことができる。
priority_queue の標準では，比較関数は「より小さい」になっており，取り出されるデータは「最も大きい」データとなっている。ダイクストラ法などでは，最も小さいデータを取り出さなければならないので，比較関数は「より大きい」を与えなければならない。
代表的な関数
push(データ) :データの追加
top( ) : 一番●●なデータを参照。
pop( ) :一番●●なデータを削除。

4.4. 構造体の比較関数

「より小さい」を示す二項演算'<'を定義する。二項演算 > は < を用いて示すことができる。等価演算 == は，< でも > でもないと定義されているので，改めて定義しなくてよい。

4.5. pair

二つ組みのデータ構造。構造体のようにメンバー名でアクセスでき，その名前は first, second となっている。pair の大小関係は，プライマリーキーが 1 つ目のデータ，セカンダリーキーが 2 つ目のデータと既に規定されている。
代表的な関数
make_pair(データ 1, データ 2) :データ 1 とデータ 2 に

よる pair を生成する。

4.5. map

連想配列のように，キーから値をアクセスするデータ構造。過去に訪問した状態を map のキーに，それまでの最短コストを値にするといった利用方法がある。様々なデータ型をキーとすることができるが，集合のときと同様，木構造に格納するため，大小関係が定義されていないといけない。

pair のように，キーへのアクセスが first，値へのアクセスが second となる。

4． 例 1（横型探索）

横型探索の簡単な例として，パソコン甲子園プログラミング部門（会津大学が主催している高校生対象のチームプログラミング大会）で，2008 年予選問題 07 として出題された「ふしぎな虫」を解説する。大会 Web ページ（http://web-ext.u-aizu.ac.jp/pc-concours/index.html）からプログラミング部門の過去問に，出題された問題文全体が掲載されている。

4.1. 問題概要

南の島で発見した虫は，接続された N 個の体節からなっている。体節の色は，赤(r) ，緑(g)，青(b)のいずれかで，1 秒ごとに一部の体幹の色を変化させる。その変化は，次のようになっている。

- あるときに色が変わるのは，隣り合った体節の色が異なる，一組（2 つ）の体幹である。
- 変化する一組（2 つ）の体幹は，その段階での 2 つの色とは同じではない色に変わる。（例えば，青と赤が隣り合った組であれば，両方とも緑に変わる。

最初の体節の色データが与えられる。そこから，すべての体節の色が同じ色になるように変化する可能性のうち，最も早い秒数を答える。そのようなことがない場合は”NA”を答える。

4.2. 考え方

問題文に記述されている例でもって考えてみる。最初に長さ N=4 で brbr という色をした虫がいたとする。この虫は，1 秒後の可能性 として，

1. 最初の br が gg に変化した ggbr
2. 真ん中の rb が gg に変化した bggr
3. 最後の br が gg に変化した brgg

と 3 つの可能性がある。

さらにその 1 秒後（開始 2 秒後）には，次の可能性がある。

- ggbr からの可能性として grrr と gggg
- bggr からの可能性として bgbb と rrgr
- brgg からの可能性として gggg と bbbg

これにより，

brbr→ggbr→gggg
brbr→brgg→gggg

の 2 つの変換経路で，2 秒ですべての体節が緑色(g)に変化できることがわかる。1 秒後のすべての可能性を調べた後に，2 秒後の可能性を調べているため，1 秒後には同じ色になることはないことがわかっている。そのため，2 秒で gggg になることがわかったとき，最短であることが保証される。

このように，色が変化する体節ペアは，最大 N-1 ある可能性があるため，すべての候補を確認して調べる探索アルゴリズムが適当である。1 秒に 1 回しか変化しないので，秒数をステップ数と考えることができる。そのため横型探索で解ける。

4.2.実装

この問題を解くための，実際のコードは次ページに示した通りである。ここでは，そのコードを理解するための補足を示す。

体節の長さ N の範囲は，2 以上 10 以下となっている。状態表現は文字列(C++の string クラス)を使う。色によって r, g, b の文字で表現する

スタートの状態は文字列として与えられる。ゴールの状態（すべて同じ色になる）は，3 種類用意できる。string クラスには，ある文字長ですべて同じ文字の文字列を生成する関数があるので，それを使用している。

変化した後の体幹の色を，簡単に計算できるように，あらかじめ r, g, b の 3 文字の文字コードの総和を求めている。変化前の 2 色の文字コードを総和から減じることにより，変化後の色の文字コードを得ることができる。

候補を格納する open のデータ構造は，横型探索の場合，キューということになっている。C++の deque を採用し，新たな候補を後から追加し，取り出すときには前から取り出している。open に格納するデータの内容は，経過秒数と状態文字列の pair としている。open の変数宣言では，deque で，格納される内容が pair<int, string> であることを示している。

訪問済みを格納する closed のデータ構造は集合(set)を用いている。横型探索の場合，closed に状態を入れるタイミングは，2 で示した一般的な探索手法よりも，はやくできる。1 ステップずつしか進まないので，展開したときに新たに見つかった状態は，最短であることが保証できるからである。

ゴール状態に到達したら，そのときの所要秒数を返す。候補がなくなった段階で，すべての可能性を調べ尽くしたことになる。open が空になったら，ループ(while 文)を抜け -1 を返す。このように見つからないという可能性があり，そのときには縦型探索を使用したときと計算時間は変わらない。横型探索は，ゴールに到達した場合，しかもそのゴールができるだけスタートから近いときに早く計算が終わる。

```
// ふしぎな虫
// #include マクロは省略
using namespace std;
int solve(const string &start)
{
        const int N = start.length( ); // 文字列長が体幹の長さ
        // 3 つのゴール状態文字列の生成。
        const string r_goal = string(N, 'r'); // 長さ N ですべて r の文字列
        const string g_goal = string(N, 'g'); // 長さ N ですべて g の文字列
        const string b_goal = string(N, 'b'); // 長さ N ですべて b の文字列
        const int color_sum = 'r' + 'g' + 'b'; // 3 色の文字の合計（変化する色を求めるとき使用）
        deque< pair<int, string> > open; // open は，〈秒数，状態〉のペアが入る，双方向キュー
        set<string> closed; // closed は状態(string)が入る集合

        open.push_back(make_pair(0, start));// 開始状態を open に入れる
        closed.insert(start);  // 最小コストは逆転されないのでここで入れる

        while(! open.empty( )) {
            // open の先頭要素の取り出し
            const int cost = open.front( ).first;
            const string state = open.front( ).second;
            open.pop_front( );
            // 状態がゴールならば秒数を返す
            if(state == r_goal || state == g_goal || state == b_goal) return cost;
            // 次の変化の可能性を生成
            for(int i = 1; i < N; i++) {
                if(state[i-1] != state[i]) { // 隣り合った体幹が違う色の場合
                    string nstate = state; // 新しい状態
                    // 現在の体幹とは違う色を求め，その色に変える
                    char color = color_sum - state[i-1] - state[i];
                    nstate[i-1] = color;
                    nstate[i] = color;
                    if(closed.find(nstate) == closed.end( )) { // 新しい状態が closed にないならば
                        // 秒数を 1 つ増やしたものと，新しい状態とのペアを open に追加
                        open.push_back(make_pair(cost + 1, nstate));
                        closed.insert(nstate); // 最小コストは逆転されないのでここで入れる
                    }
                }
            }
        }
        return -1; // すべての変化を調べつくした。-1, NA
}

// main 関数は省略
```

５．　例２（横型探索）

4 節と同じように横型探索を使用して解ける問題として，2010 年 ICPC（国際大学対抗プログラミングコンテスト）国内予選問題 B で出題された「迷図と命ず」を取り上げる。問題文は，

http://icpc2010.honiden.nii.ac.jp/domestic-contest/problems で見ることができる。

5.1. 問題記述

四角い部屋を縦横に並べた長方形の迷路を考える。迷路は横に w 個の部屋,縦には h 個の部屋が並んでおり，w × h 個の部屋からなっている。ある部屋から，縦，または横に隣接されている部屋に壁がある場合とない場合があり，壁がなければ隣の部屋に移動することができる。壁データが入力として与えられる。

迷路の左上がスタート，右下がゴールであるとき，最短でいくつの部屋を通れば，スタートからゴールまで行けるか答える。行けない場合は 0 を返す。

5.2. 考え方

迷路の部屋と壁情報は移動中，変わらないので，その情報は１つだけ持てばよい。

探索アルゴリズムでは，変化する状態を扱うため，ここでは移動中にいる部屋の x 座標と y 座標となる。

左上の座標位置である(0, 0)からスタートし，右下の座

標位置である(w-1, h-1)がゴールとなる。

部屋の 4 方向の壁の有無を見て，壁がないときには，その方向の部屋に移動できるとする。通った部屋数の最小を求めるので，1 ステップで 1 部屋移動する横型探索を用いる。

5.3. コード

この問題では，入力データから部屋と壁情報を構成する実装技術も問われているが，そこはアルゴリズムの本質ではないので，ここでは，3 次元配列（部屋の y 方向位置，部屋の x 方向位置，壁の方向を添字として，値として開いているかどうか示す bool 型）にすでに値が入っているとして説明を進める。

4 節と同じ横型探索を使う問題なので，そのコードとの差異についてここでは述べる。

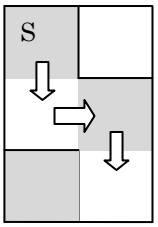
2 次元の移動を簡単に記述する典型的な方法として，移動方向別に番号をつけ，その番号を配列の添字とするものがある。それにより，同じようなコードを羅列することなく，繰り返しで処理を記述できるようになる。

0: 右（x を 1 増やす）方向
1: 下（y を 1 増やす）方向
2: 左（x を 1 減らす）方向
3: 上（y を 1 減らす）方向

この番号と，移動方向を格納している dy, dx 配列，壁があるかどうか示す wall 配列の添字をあわせるようにしている。

移動先の部屋の y 座標値，x 座標値が全体の迷路の範囲外になっていないのか，本来チェックしなければならないのだが，壁情報を生成するときに，そのような場所へ移動する壁は必ず存在するように作成してある。

4 節の問題は，状態表現が文字列のため set を用いて closed データ構造を表現した。この問題は，2 つの整数値(y, x 座標の値)で状態を管理しているため，2 次元配列を用いて closed データ構造を表現できる。C++の set は内部で木を用いているため，挿入，アクセスには log2n に比例した計算時間がかかる。配列は，データの大きさとは関係なく同じ時間で処理できるので，配列で記述できるのであれば，set よりも配列を使いたい。




```
// 迷図と命ず
// #include マクロは省略
using namespace std;
const int SIZE = 32;
typedef pair<int, int> State;
typedef pair<int, State> Node;
const int INF = 100000;
bool wall[SIZE][SIZE][4];
int solve(const int h, const int w)
{
        // y=0, x=0 がスタート
        int closed[SIZE][SIZE] = {false};
        closed[0][0] = true;
        deque<Node> open;
        open.push_back(make_pair(1, make_pair(0, 0))); // スタート位置は一部屋目
        // y=h-1, x=w-1 がゴール
        const int goal_y = h-1, goal_x = w-1;
        while(! open.empty()) {
                Node s = open.front(); open.pop_front();
                const int step = s.first, y = s.second.first, x = s.second.second;
                if(y == goal_y && x == goal_x) return step; // ゴールに到達した場合
                // 4 方向の壁を調べる
                const int dx[4] = {1, 0, -1, 0};
                const int dy[4] = {0, 1, 0, -1};
                for(int d = 0; d < 4; d++) {
                    if(wall[y][x][d] == false) { // 壁がないので移動できる
                        const int nx = x + dx[d], ny = y + dy[d];
                        if(closed[ny][nx] == false) {
                            closed[ny][nx] = true;
                            open.push_back(make_pair(step+1, make_pair(ny, nx)));
                        }
                    }
                }
        }
        return 0; // すべての移動先を調べ尽くした
}
// main 関数は省略
```

６．　例３（最短経路探索）

2008 年 ICPC（国際大学対抗プログラミングコンテスト）国内予選問題 D で出題された「ちょろちょろロボット」を取り上げる。問題文は，

http://sparth.u-aizu.ac.jp/icpc2008/d_problem.php?lang=jp で見ることができる。

6.1. 問題記述

正方形のマスを縦横に並べた長方形の床面を考える。床面の幅 w, 床面の縦 h であり，w * h 個のマスからなっている。ロボットが左上(0, 0)の位置にいて，右(x が正の方向)を前にしている。このロボットを最小コストで右下(w-1, h-1)のマスに移動させたい。

ロボットへの命令は 5 種類ある。（番号が命令番号）

0. 直進(Straight)：現在の進行方向のまま，次のマスに前進する。
1. 右折(Right)：現在の進行方向を 90 度右に変えて，次のマスに前進する。
2. 反転(Back)：現在の進行方向を 180 度変えて，次のマスに前進する。
3. 左折(Left)：現在の進行方向を 90 度左に変えて，次のマスに前進する。
4. 停止(Halt)：現在のマスにとまってゲームを終了する。

それぞれのマスには，命令が割り当てられていて，特に何もしなければ，ロボットはマスの命令を順番に実行していく。それではゴールに到達しないかもしれないので，人間がロボットに，人間が指定した命令を実行させることができる。命令の種類毎に，人間が 1 回命令を実行させるためのコストが決められている（正の整数）。ロボットをゴールにたどりつかせるための最小コストを求める。

6.2. 考え方

先ほどの 2 つの問題は，最短の秒数，部屋数を求める問題であったが，探索の 1 つのステップが「秒」,「部屋」となっていたので，横型探索で解くことができた。求める最小の単位が，探索のステップとあわない場合は，グラフ構造の代表的な問題である最短経路問題となる。最短経路問題の代表的なアルゴリズムであるダイクストラ法は，一般的な探索アルゴリズムで，スタートからコスト最小のものから取り出すものと考えることができる。

6.3. C++による実装コード

5 節の迷路の問題は，位置の座標を状態としたが，この問題ではそれに加えてロボットの向いている方向も状態にする。右，下，左，上の順に 0, 1, 2, 3 の整数値を方向に割り当てる。x 座標，y 座標，方向の 3 つ組みを状態とするために構造体を用意する。open のデータ構造としては priority_queue を使用し，その中には int 型のコストと状態の構造体のペアを格納する。

priority_queue の比較関数として，greater（大なり）を指定すると，最も小さいものが取り出せる。pair では，プライマリキーに first が利用され，この場合は int 型なので > は定義済みである。セカンダリーキーが状態の構造体であり，その大小関数を定義しなければならない。

状態の順序関係を活用することはなく，単に定義だけされていれば良い。3 つのメンバー変数を順序づけて，最初に決めた変数の大小関係が決まればそれで決め，もし同じであれば次の順序の変数で決める，というように記述する。

```
// ちょろちょろロボットデータ定義
// #include マクロは省略
using namespace std;
// 状態を表す構造体
struct State{
        int y, x; // 位置
        int d; // 向き
};
// 構造体の大小関係の定義
// < でも > でもないものが等価==なので，全ての値が同じときだけ == になるようにする
bool operator<(const State &a, const State &b)
{
    if(a.y < b.y) return true; if(a.y > b.y) return false;
    if(a.x < b.x) return true; if(a.x > b.x) return false;
    return a.d < b.d;
}
// > は，< の反対（以下は典型的な記述）
bool operator>(const State &a, const State &b) { return b < a; }
```

S	S	S	S	S	S	S	R
B	L	S	R	H	S	S	R
L	L	S	S	S	S	S	H

5 節のプログラム同様，ロボットの向きを，0, 1, 2, 3 の値に割り当てている。問題文では命令が番号で記述されており，向きに命令番号を加算した結果に，4 で割った余りが新たな向きとなる

新しく候補を生成したときに，横型探索では closed にあるものは，open に追加しなくてよい。横型探索の場合は，すでに訪問済みのもの方が，ステップ数が小さいか同じことが保証されているからである。一方，最小コスト問題の場合は，後から生成された候補の方が，以前に生成された同一状態の候補より，コストが小さい場合があり，その場合は open に追加しなければならない。おその判断をするため，closed には，すでに訪問したかどうかだけではなく，どのコストで訪問したか記録しなければならない。

2 次元の座標，向きの 3 つとも整数値なので，closed は 3 次元配列を用いることができる。ここでは，整数値以外の状態表現のケースの記述方法を紹介するために，あえて closed データ構造に map を使っている。map のキーとして構造体を使っている。C++の map では，木構造を用いているので，大小関係の演算が規定されていないとキーとして使用できない。

```
// ちょろちょろロボット続き
typedef pair<int, State> Node;

const int SIZE = 32;
int op[SIZE][SIZE]; // 床に書いてある命令
int standard_cost[4]; // 命令に対する標準コスト
int solve(const int h, const int w)
{
        map<State, int> closed;
        priority_queue< Node, vector<Node>, greater<Node> > open; // コスト最小を取り出すため
        State start = (State){0, 0, 0};
        open.push(make_pair(0, start));
        closed[start] = 0;
        const int goal_x = w-1, goal_y = h-1;
        while(!open.empty()) {
                // コスト最小を取り出す
                int cost = open.top().first;
                State cur = open.top().second;
                open.pop();
                // ゴールに到達しているかどうか判定
                if(cur.y == goal_y && cur.x == goal_x) return cost;
                // ロボットの移動
                const int dx[] = {1, 0, -1, 0};
                const int dy[] = {0, 1, 0, -1};
                for(int d = 0; d < 4; d++) {
                        const int nd = (d + cur.d) % 4;// ロボットの新しい向き
                        const int ny = cur.y + dy[nd];
                        const int nx = cur.x + dx[nd];
                        if(ny >= 0 && ny < h && nx >= 0 && nx < w) { // 新しい位置は枠内にある
                                int ncost = cost + (d == op[cur.y][cur.x] ? 0 : standard_cost[d]);
                                State nstate = (State){ny, nx, nd};
                                if(closed.find(nstate) == closed.end() // 既に訪問していない
                                        || ncost < closed[nstate]) { // 又は過去のコストより小さい
                                        closed[nstate] = ncost;
                                        open.push(make_pair(ncost, nstate));
                                }
                        }
                }
        }
        return -1; // ゴールに到達しない場合
}
```

7． おわりに

7.1. 留意点

縦型探索により問題解決する場合は，[2]で述べたような再帰によるバックトラックで実装することが普通であるが，再帰の深さが深くなると，実行時にスタックオーバーフローになる場合がある。そのようなときには，一般的な探索の枠組みで記述すると良い。deque の代わりに vector(stack)を使えば縦型探索になる。

ゴールに到達できないような場合，候補がなくなるまでの途中, open データ構造に数多くの候補が入ってくることがある。原理的に状態数は，状態を構成する変数のとりうる値の組み合わせになるので，最大状態数の見込みはたつ。しかし，それが大きい場合でも，スタートから到達可能できない状態が多くある場合もあり，あきらめる必要はない。計算時間がかかり過ぎて，実際のコンテストでは，所定の計算時間オーバー(Time Limit Exceeded)になる可能性がある。

縦型探索では，時間がかかり過ぎる問題の場合，ゴールの状態に行くことがない方向へ展開しないようにする「枝狩り」が重要であった。横型探索でもそのことは同様にあてはまるが，プログラミングコンテストでは，横型探索で枝狩りを必要とする問題は少ない。

プログラミングコンテストでは，両側探索が有効なことが多い。ゴールの状態が限定的な場合，スタートとゴールの両側から 1 ステップずつ横型探索を進めていき，合流したところを答えとするものである。2 つの横型探索は，答えとなるステップ数の半分のステップ数で合流する。合流したところで，それぞれのステップ数の総和が求める答えとなる。ステップ数が半分になったとき，open データ構造に格納される候補数は, 問題に依存しているが，同じ状態に戻らない場合には，平方根にまで少なくなる。

7.2. さらに学習したいひとのために

国際大学対抗プログラミングコンテストで出題されている問題やアルゴリズムについては，[4]でも概略が示されているので，あわせて読んでみるとよい。

今回解説した，横型探索，ダイクストラ法によって，解ける他の問題をあげておく。これらの問題は，Aizu Online Judge(http://judge.u-aizu.ac.jp/)を使って，作成したプログラムの正誤を判定してもらうことができるので，チャレンジしてみるとよい。

【横型探索又は両側探索を用いるもの】

1. 1999 年アジア地区予選京都大会 Problem G “Walking Ant”
2. 2000 年アジア地区予選つくば大会 Problem C “Push!” [5]に解説あり
3. 2009 年アジア地区予選東京大会 Problem B “Repeated Substitution with Sed”
4. 2007 年アジア地区予選東京大会 Problem G “The Morning after Halloween”
5. 2006 年アジア地区予選横浜大会 Problem C “Cubic Eight Puzzle”
6. 2003 年アジア地区予選会津大学 Problem F “Gap”
7. 2004 年アジア地区予選愛媛大会 Problem E “Confusing Login Name”
8. 2005 年アジア地区予選東京大会 Problem D “Organize Your Train”
9. 2006 年横浜大会国内予選 Problem C “六角沼の六角大蛇”

【ダイクストラ法を用いるもの】

10. 2007 年東京大会国内予選 Problem D “崖登り”
11. 2009 年東京大会国内予選 Problem D “離散的速度”
12. 2010 年アジア地区予選東京大会 Problem E “The Two Men of the Japanese Alps”

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] 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(2)――探索（基本編）―――, 専修ネットワーク＆インフォメーション, No. 14(Jan. 2009), pp. 41-50.

[3] 石畑清：アルゴリズムとデータ構造，岩波書店, 1989.

[4] 筧捷彦：目指せ!プログラミング世界一―大学対抗プログラミングコンテスト ICPC への挑戦，近代科学社, 2009.

[5] 田中哲郎：倉庫番パズル，情報処理，Vol. 43，No. 11(Nov. 2002). pp. 1253-1258. (See also http://www.ipsj.or.jp/07editj/promenade/4311.pdf)