Program Promenade
連載

怪物を閉じ込める

田中 哲朗（東京大学情報基盤センター）
ktanaka@tanaka.ecc.u-tokyo.ac.jp

■問題の定義

今回は，2003 年に行われたアジア予選会津大会の問題 H「Monster Trap」を取り上げる．

図 -1 のように怪物（M で表現されている）が平面上の原点にいる．魔法使いは魔法の杖でさまざまな長さの線分を引くことができる．怪物は線分を乗り越えることができないが，どんな小さい隙間があってもすり抜けてしまう．線分の集合が与えられたときに，怪物の封鎖に成功しているかどうかを判定するプログラムを作成するというのがこの問題である．

図 -1 のように原点を含む閉領域ができる場合は封鎖は成功であり，**図 -2** のように原点が外側につながっていたら封鎖は失敗である．

問題には以下のような条件が与えられている．

- 線分の数は 100 以下
- 線分の端点の座標は −50 以上 50 以下の整数
- 長さ 0 の線分はない
- 原点を通る線分はない
- 2 つの線分が共有する点は高々 1 点
- 3 つの線分が 1 点で交叉しない

ただし，図 -1，2 を見ると分かるが，線分が端点を共有することや，線分の端点がちょうど別の線分上にあることは禁止されていない．後から分かるように，これが問題を難しくしている要因の 1 つである．

■準備

今回も C++ 言語と STL（Standard Template Library）を使って解答を作成する．この章では，平面幾何の問題を解くための準備として，点や線分を表現する型や，線分間の関係を求める関数を導入する．

点の表現

この連載の以前の記事 [1] で，格子座標上の点を表現するのに，

```
typedef pair<int,int> Point;
```

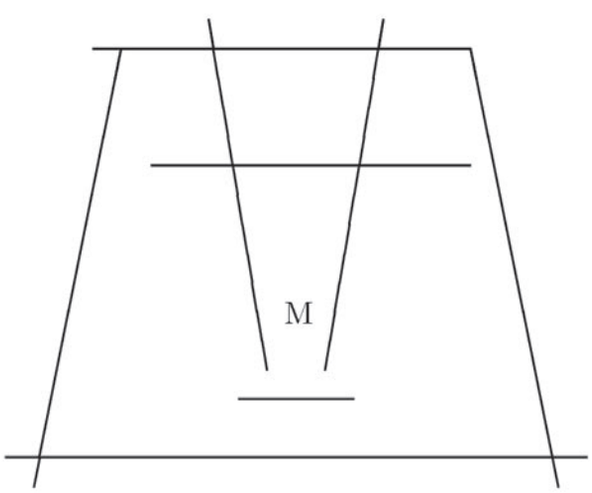

図 -1　閉じ込められた怪物

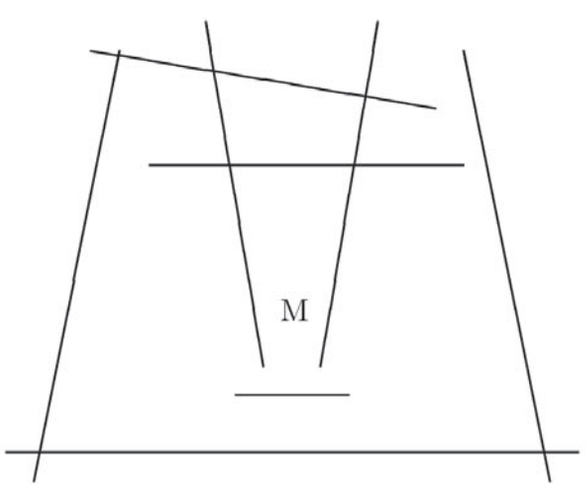

図 -2　脱出可能な怪物

のように定義したことがあるが，今回は複素数を表現するためのクラステンプレート complex を用いて，以下のような型で平面上の点を表現する．

```
typedef complex<double> Point;
```

よく知られているように，複素数は実数上の 2 次元ベクトルと対応づけることが可能であり，複素数上の演算は以下のように 2 次元ベクトルに対する操作に対応している．

- 演算子「+」,「-」がそのままベクトルとしての和，差になっている．
- abs(x) でベクトル x の長さが得られる．
- x*y を計算すると「ベクトル (1,0) がベクトル y と重なるように，回転して，拡大（縮小）する」という変換をベクトル x に施したものが得られる．
- x/y を計算すると「ベクトル y がベクトル (1,0) と重なるように，回転して，拡大（縮小）する」という変換をベクトル x に施したものが得られる．
- arg(x) でベクトル (1,0) を基準としたベクトル x の角度[☆1] が $-\pi$ から π の範囲で得られる．

ついでに，Point の vector である vPoint という型と，Point 間の演算子「<」も以下のように「X 座標の大小，ただし X 座標の値が等しいときは Y 座標の大小」と定義しておく．

```
typedef vector<Point> vPoint;

namespace std{
  bool operator<(Point const& p0,
                 Point const& p1){
    if(p0.real()==p1.real())
      return p0.imag()<p1.imag();
    return p0.real()<p1.real();
  }
}
```

線分の定義

コンテストの問題を速く解くには，自分でクラスを定義しないで，なるべく STL で定義した方が良いと，この連載で繰り返し書いてきた．それに従うと，線分を表す型 Line は，

```
typedef pair<Point,Point> Line;
```

と定義したくなるが，2 端点を添字で指定して配列風に使えるようにしておくと便利なので，以下のように vPoint を継承するクラスとして定義しておく．なお，Line の vector もよく使うので vLine という名前の型として定義しておく．

```
struct Line : public vPoint{
  Line(Point const& p0,
       Point const& p1) {
    push_back(p0);
    push_back(p1);
  }
};
typedef vector<Line> vLine;
```

線分間の関係

問題の条件内で，2 つの線分の間の関係は以下のように分類可能である．異なる線分はただ 1 点のみを共有する条件になっているので，たとえば (0,0)-(2,0) と (3,0)-(1,0) のように線分の一部を共有する場合や，完全に一致する場合，片方が他方に含まれる場合は考えなくてよい．

1．片方の線分の端点が，他方の線分の端点と一致する場合
2．片方の線分の端点が他方の線分上にある場合
3．双方が端点以外の点で交わる場合
4．交わらない場合

これらを判定する関数を定義していくことにする．

まず，片方の線分の端点が，他方の線分の端点と一致するかどうかを判定する関数 connect を定義する．以下のように 4 通りをチェックするだけでよい．

```
bool connect(Line const& l0,
             Line const& l1){
  for(int i=0;i<2;i++)
    for(int j=0;j<2;j++)
      if(l0[i]==l1[j])
        return true;
  return false;
}
```

次に，片方の線分の端点が他方の線分上にある場合を判定する関数 isOn を定義する．connect と定義が重ならないように，線分の端点を与えた場合は，偽となるようにする．

```
bool isOn(Line const& l,
          Point const& p){
  // 端点を共有する場合は偽と定義する
  if(l[0]==p || l[1]==p)
    return false;
  // l[0]を原点，l[1]を(1,0)となるように座標変換
  Point d=l[1]-l[0];
  Point p0=(p-l[0])/d;
  // X 軸上にあり，かつ(0,0)-(1,0)の間にあるときに真
  return p0.imag()==0.0 &&
         0.0<p0.real() &&
         p0.real()<1.0;
}
```

一般に点が線分上にあるかどうかは，誤差が少しで

☆1 以下では複素数の言葉を使って「偏角 (argument)」と書く．

も入ったら正しく判定できない．関数 isOn は，登場する点の座標が整数で表される場合は正しく判定できるが，それ以外の場合は正しく判定できないので注意が必要になる．

最後は，双方が端点以外の点で交わる場合である．これも定義の重複を避けるために，connect または isOn が成り立つときは偽となるように定義する．

また，交わる場合は参照として渡される引数 p に交点の座標を返すことにする．

```
bool intersect(Line const& l0,
               Line const& l1,
               Point& p){
  // 端点を共有する場合，端点が線分上にある場合は偽と定義する
  if(connect(l0,l1) ||
     isOn(l0,l1[0]) || isOn(l0,l1[1]) ||
     isOn(l1,l0[0]) || isOn(l1,l0[1]))
    return false;
  // l0[0]を原点，l0[1]を(1,0)となるように座標変換
  Point d=l0[1]-l0[0];
  Point p0=(l1[0]-l0[0])/d;
  Point p1=(l1[1]-l0[0])/d;
  // p0，p1がX軸の上下同じ側にある場合は交わらない
  if(p0.imag()*p1.imag()>=0)
    return false;
  double t=p0.imag()/
           (p0.imag()-p1.imag());
  // X軸との交点のX座標がsに入る．
  double s=p0.real()*(1-t)+p1.real()*t;
  if(s<=0.0 || 1.0<=s) return false;
  // 交点の座標(sが範囲内のとき)
  p=l0[0]+d*s;
  return true;
}
```

■アルゴリズム

問題に取り組むための準備ができたところで，問題を解くためのアルゴリズムを考える．高速化まで考慮するとさまざまな方法がありそうだが，素直に解くと以下のようになるだろう．

- 線分同士が交叉したり，線分の端点が別の線分上にある場合は，その交点で線分を分割する．この操作を繰り返して，すべての線分が端点以外の共有点を持たないようにする．
- 線分の集合を連結成分に分解する．各連結成分ごとに原点を閉じ込めているかどうかを別々に判定することが可能である．
- 各連結成分について，連結成分の外側を左手法で（壁を左手で触りながら）トレースして 1 周したパスを作成する．このパスが原点を含むなら連結成分が原点を閉じ込めていることが言える．
- 原点を閉じ込めている連結成分が 1 つでもあれば，回答は Yes となる．

それぞれのステップをどうプログラミングするか順に見ていこう．

単純化

線分の集合を交点を持たない形に分割するアルゴリズムは以下のように記述できる．

1．線分 A の端点が線分 B 上にあるかどうかをすべての線分の組合せに関してチェックする．線分上の端点が見つかった場合は，その点で線分 B を 2 つに分割して線分の集合を更新する．
2．線分の組合せすべてについて，2 つの線分が端点以外で交わるかどうかを調べる．見つかった場合は，それぞれの線分をその点で分割する．見つからなくなるまでこの操作を繰り返す．

線分の集合を，Line の vector で表現することにすると，(1) の部分の関数 removeOn は以下のように書ける．

```
void removeOn(vLine& ls){
  for(size_t i=0;i<ls.size();i++){
    for(size_t j=i+1;j<ls.size();j++){
      for(int k=0;k<2;k++){
        if(isOn(ls[i],ls[j][k])){
          ls.push_back(Line(ls[i][0],
                            ls[j][k]));
          ls[i][0]=ls[j][k];
          break;
        }
        else if(isOn(ls[j],ls[i][k])){
          ls.push_back(Line(ls[j][0],
                            ls[i][k]));
          ls[j][0]=ls[i][k];
          break;
        }
      }
    }
  }
}
```

(2) の部分の関数 simplify も同様に，以下のように書ける．

```
void simplify(vLine& ls){
  for(size_t i=0;i<ls.size();i++){
    Line li=ls[i];
    for(size_t j=i+1;j<ls.size();j++){
      Point cross;
      if(intersect(ls[i],
                   ls[j],cross)){
        ls.push_back(Line(ls[i][0],
                          cross));
        ls[i][0]=cross;
        ls.push_back(Line(ls[j][0],
```

```
                                cross));
        ls[j][0]=cross;
      }
    }
  }
}
```

isOn が成り立つケースと intersect が成り立つケースをまとめて扱うともっと簡単に書けそうな気がするが，残念ながらうまくいかない．一度，intersect が成り立ち，そこで線分を分割してしまうと，端点の座標が整数でない線分が生じてしまう．すると，isOn が成り立つかどうか誤差により正しく判定されないことがある．

なお，このアルゴリズムは分割した線分を後ろにつけていってしまっているが，その結果，すでに intersect しないことが判明している（ある線分が別の線分と intersect しない場合は，分割後の線分とも intersect しないことが明らか）ものも再びチェックしてしまっている．効率が問題になる場合は工夫が必要となるが，ここでは読みやすさを重視した．

連結成分への分解

連結成分への分解は線分の同値類を作成して，ある線分と別の線分が connect の関係にあるときに，それぞれの線分の属する同値類をマージする操作を繰り返すことで実現できる．

同値類を処理するためのクラス Equiv を以下のように定義する．線分の集合が vector で表現されていて，各線分には添字が対応しているので，Equiv は整数元のみを扱うようにしている．

```
struct Equiv{
  vector<int> links;
  // 大きさ1の同値類をn個作る
  Equiv(int n) :links(n){
    for(int i=0;i<n;i++)
      links[i]=i;
  }
  // xの属する同値類の代表元(最小の元)を返す
  // 副作用としてたどったリンクのショートカットを作成する
  int deref(int x){
    int xx=links[x];
    if(xx==x) return x;
    return links[x]=deref(xx);
  }
  // xの属する同値類と，yの属する同値類をマージする
  void unify(int x, int y){
    int xx=deref(x), yy=deref(y);
    if(xx<=yy) links[yy]=xx;
    else links[xx]=yy;
  }
};
```

ここの処理は，以前この連載で紹介された「嘘つき島の問題[2]」中の union-find 問題そのものなので，詳しい説明は省略する．

線分の集合 ls から連結成分の集合（vLine の vector として定義する型 vvLine で表現する）を返す関数 decomposeCC は以下のように定義できる．

```
typedef vector<vLine> vvLine;
vvLine decomposeCC(vLine const& ls){
  vvLine ret;
  Equiv equiv(ls.size());
  for(int i=0;i<(int)ls.size();i++){
    for(int j=0;j<i;j++){
      if(connect(ls[i],ls[j])){
        equiv.unify(i,j);
      }
    }
  }
  for(int i=0;i<(int)ls.size();i++){
    if(equiv.deref(i)==i){
      vLine newLs;
      newLs.push_back(ls[i]);
      for(int j=i+1;j<(int)ls.size();j++){
        if(equiv.deref(j)==i)
          newLs.push_back(ls[j]);
      }
      ret.push_back(newLs);
    }
  }
  return ret;
}
```

左手法でのトレース

左手法でのトレースは以下のようにする．

- 線分の端点全体の中で X 座標が一番小さい点を選んでトレースの開始点とする．この点は，確実に外側の点になる．次に移動する点は，開始点から見た角度が最も小さい（$-\pi$ に最も近い）隣接点とする．
- ある点にトレースが進んだときに，次に移動する点は，移動前の点への偏角を初期値として角度を増やしていったときに注目する点から見た角度が最初に一致する隣接点とする．隣接点が 1 つしかない場合は移動前の点を次の移動点とする．
- 移動を繰り返していって，開始点に戻ったら終了．

このアルゴリズムで，図 -3 の左のようなトレースした領域の面積が 0 の場合も，図 -3 の右のように開始点以外の点を複数回通る場合も正しく処理される．前者では，$P0-P1-P2-P1-P0$ とトレースされるし，後者では $P0-P1-P2-P3-P4-P5-P3-P2-P0$ となる．

このアルゴリズムを記述するために，点をキーとして偏角でソートされた隣接点の vector を得るデータ構造を STL の map を使って以下のように定義する．

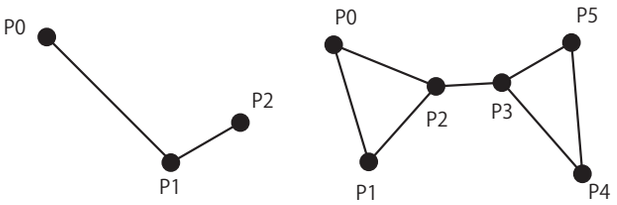

図-3　トレースの例

```
typedef pair<double,Point> doublePoint;
typedef vector<doublePoint> vDoublePoint;
typedef map<Point,vDoublePoint> AdjPoint;
```

こうすると，図 -4 では P0 をキーとして，{<-5/6π, *P*3>, <-1/4π, *P*4>, <1/6π, *P*1>, <3/4π, *P*2>} のような列が得られるわけである．P3 から P0 に来たときは次に P4 に行けばよいし，P4 から来たときには P1 に行けばよい，列の中で元の点を探して，次の点（一番最後の場合は，最初に戻る）が移動先の点となる．

トレース全体は以下のようになる．

```
// 移動前の点と隣接点のリストから移動先を決定
Point findNext(vDoublePoint const& vdp,
               Point const& p){
  for(int i=0;i<(int)vdp.size()-1;i++)
    if(p==vdp[i].second){
      return vdp[i+1].second;
    }
  return vdp[0].second;
}
// 左手法でトレースして通った点の列を返す.
vPoint trace(vLine const& ls){
  AdjPoint aPoint;
  // 隣接点(偏角付き)のリストを作成
  for(int i=0;i<(int)ls.size();i++){
    aPoint[ls[i][0]].push_back(
      doublePoint(
        arg(ls[i][1]-ls[i][0]),
        ls[i][1]));
    aPoint[ls[i][1]].push_back(
      doublePoint(
        arg(ls[i][0]-ls[i][1]),
        ls[i][0]));
  }
  // X座標最小の点の計算と，隣接点リストのソート
  Point minPoint(INT_MAX,INT_MAX);
  AdjPoint::iterator it;
  for(it=aPoint.begin();
      it!=aPoint.end();++it){
    Point p=it->first;
    minPoint=min(minPoint,p);
    vDoublePoint &vdp=it->second;
    sort(vdp.begin(),vdp.end());
  }
  // 返り値の初期化
  vPoint ret;
  ret.push_back(minPoint);
  // 最初の移動先
```

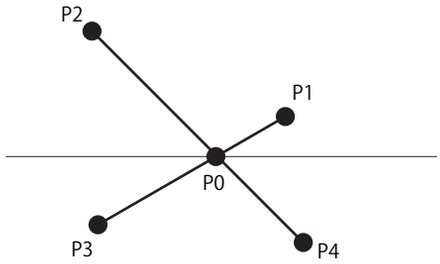

図-4　隣接点との関係

```
  Point curP=aPoint[minPoint][0].second;
  Point lastP=minPoint;
  // トレースの繰り返し
  while(curP!=minPoint){
    ret.push_back(curP);
    Point nextP=findNext(aPoint[curP],
                         lastP);
    lastP=curP;
    curP=nextP;
  }
  // 開始点を最後にも追加
  ret.push_back(minPoint);
  return ret;
}
```

パスが原点を含むかどうかの判定

自己交叉のない線分列（パス）からなる領域が内部に原点を含むかどうかを求めるアルゴリズムとしては，原点を端点とする半直線と交点の数を数えるという方法がよく知られている．図 -5 の左のように，この回数が奇数回ならば領域は原点を含み，図 -5 の右のように偶数回ならば含まない．

ただし，この半直線の選び方によっては問題が起きる可能性がある．たとえば，0 から X 軸に沿って正方向に伸びる半直線を選んでみる．交点がたまたま，X 軸上にあったとする．図 -6 の 1 番左のような場合は，交点の数を奇数個と判断したいし，2 番目の場合は偶数個としたい．この例ではローカルに判定すればよさそうだが，3 番目や 4 番目のような例まで考え始めると面倒になる．

対策はいくつかある．

1．半直線の方向を乱数で選べば，たまたま一致するようなことはないだろう．自信がなければ，乱数で選んだ方向を複数使って多数決をとる．
2．パスを構成する点すべての原点からの偏角を計算し，その間で十分が間が空いている角度に半直線を伸ばす．
3．パスを構成する点が乗ることがないと保証されている半直線を選ぶ．

実用的には 1. でも構わないと思うが，この問題の場合は 3. でもよい．元の線分の端点の X 座標，Y 座

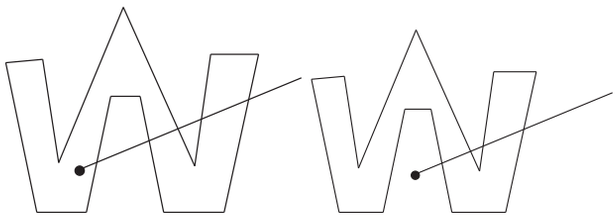

図-5　パスが原点を含むかの判定

図-6　パスと半直線の関係

標が −50 から 50 までの整数値なので，交点の座標は有理数となり，その分母は 20000 を超えない（これは過大な見積もり）ことが分かる．したがって，(0,0) から (51,1/20000) の方向に伸ばした半直線（プログラム上は線分で構わない）の上に交点が重なることはない．数値誤差的にも double なら許せるだろう．

```
bool isInside(vPoint const& vp){
  Line l0(Point(0,0),
          Point(51.0,1.0/20000.0));
  int count=0;
  for(int i=0;i<(int)vp.size()-1;i++){
    Point dummy;
    Line l1(vp[i],vp[i+1]);
    if(intersect(l0,l1,dummy))
      count++;
  }
  return (count&1)==1;
}
```

まとめ

以上のアルゴリズムをまとめた関数 trapAll は以下のように書ける．審判用データは 32 個の入力からなり，その中には図 -7 のような複雑なデータも含まれているが，作成したプログラムは実用的な時間で正解を返すことができる．

```
bool trapAll(vLine& ls){
  removeOn(ls);
  simplify(ls);
  vvLine cc=decomposeCC(ls);
  for(size_t i=0;i<cc.size();i++){
    if(isInside(trace(cc[i])))
      return true;
  }
  return false;
}
```

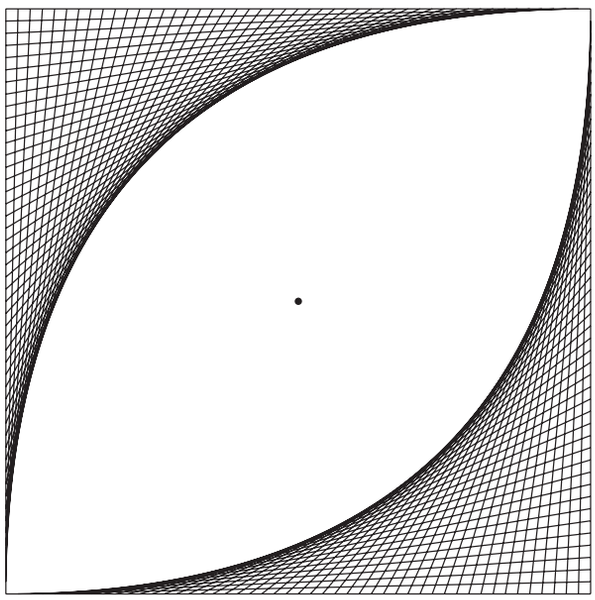

図-7　入力の例

■おわりに

この問題は見るからに手がかかりそうな問題であるが，実際に解いてみて，解答を見やすく書き直してみても，手間がかかった．2003 年の最後の問題だったということもあり，解けたチームはなく，submit したチームも 1 チームだけだったようだ．

せっかく作った問題なのに，解くチームがいないというのは，不本意だろう．限られた時間で解くというコンテスト用の問題としては，もう少し難易度を下げた方が良かったと思う．

難易度を下げる手段はいくつか考えられる．線分の途中に他の線分の端点が来ることがないという条件を加えれば，少しは易しくなるだろう．また，問題中の線分の集合全体が連結だとすると，連結成分に分解するプロセスが不要になる．個人的には，単純化された後の線分の集合を与える位の難易度が適当なのではないかと思う．

参考文献

1）田中哲朗: Program Promenade 倉庫番パズル, 情報処理学会誌, Vol.43, No.11, pp.1253-1258 (Nov. 2002).

2）石畑　清: Program Promenade 嘘つき島の問題, 情報処理学会誌, Vol.44, No.5, pp.539-545 (May 2003).

（平成 17 年 1 月 17 日受付）