Program Promenade

連載

点の密集区域

石畑　清（明治大学理工学部）
ishihata@cs.meiji.ac.jp

今回は，幾何の問題を取り上げる．2004 年 7 月の国内予選の問題 D「Circle and Points」である（http://www.ehime-u.ac.jp/ICPC/problems/domestic/d2004/ 参照）．非常に大雑把に言うと，点が最も密集している場所を見つけることを要求している．

今まで何回も幾何の問題を題材にしてきたが，今回の問題はそれらと比べて易しい部類に属する．コンテストで幾何の問題に出くわしたときにどういう考え方をすればよいかのサンプルとして参考にしてもらえたらと思う．

■問題

xy 平面の上に n 個の点がある．これらの点それぞれの座標値が与えられている．半径 1 の円を xy 平面上の適当な位置に置いて，円の中になるべく多くの点を囲むようにしたい．一番多くの点を囲むようにしたとき，何個の点を囲めるかを答えよ．

図 -1 を見てほしい．この図の中には 6 個の点がある．円を A の位置に置くと，4 個の点を囲んだことになる．一方，B の位置に置くと，5 個の点が囲まれる．この例の場合は，これが最善（6 個の点を囲むような円はない）で，5 が答ということになる．

入力データは，点の個数 n と，n 個の座標値（x と y の組）だけである．n は 1 以上 300 以下の整数と決められている．点の座標は，x, y とも 0 以上 10 以下の実数で，しかも小数点以下 5 桁の小数で与えられることになっている．小数点以下 5 桁という情報に意味がないわけではないが，問題を解く方法に影響を与えるようなものではない．無視してよい．

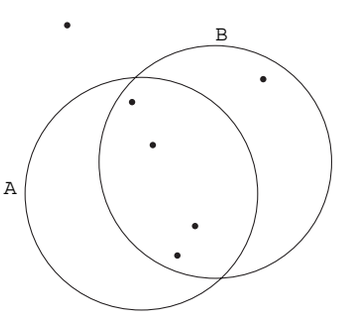

図-1　点を囲む円

この問題は，点の位置関係に制約がないと，非常に解きにくいものになる．計算の途中で生じる誤差によって，答が変わる恐れが生じるからである．このような心配をしなくてよいように，出題者は非常に丁寧な制約を用意してくれている．

- 2 つの点の距離が 0.0001 より小さくなることはない．
- 2 つの点の距離が 1.9999 以上 2.0001 以下になることもない．
- 3 つの点への距離がすべて 0.9999 以上，1.0001 以下になるような地点を xy 平面上に見つけることはできない．別の言い方をすると，3 つの点の円周からの距離が 0.0001 以下になるような半径 1 の円は存在しない．

2 番目の制約の意味を考えてみよう．2 つの点の距離がちょうど 2 だったとする．この 2 つの点の両方を囲む半径 1 の円を作ることができる（「囲む」という言葉は点がちょうど円周上に乗っている場合を含む）が，それはこの 2 点を直径の両端とするような円 1 つだけである．ある計算方式によって，この円を求めることが可能だとしても，その計算の途中で誤差が紛れ込むかもしれない．最終的に求まる円は，2 点を直径の両端とする理想の円から，ほんの少し違ったものになるだろう．すると，この円は 2 点のうちの一方しか囲んでいないことになる．

つまり，入力値から理論的に得られる答が 2（2 点を囲む円が作れる）であったとしても，現実の計算の

結果は1（1点を囲む円しか作れない）となって，間違いになってしまうであろう．

こういう心配があったのでは，プログラムを書けない．2番目の制約によって，距離が2に近い2点はないと保証されるので，このような心配は不要になる．

3番目の制約の意味も同様である．ある性質を持つ円を計算で求めたとする．理論的にその円が囲むはずの点の集合と，計算の結果その円が囲んでいると判定される点の集合が違っていては困る．3点をきわどくかすめるような半径1の円があると，このような危険なケースの心配が生じるが，3番目の制約はこのような円の存在を禁止している．計算の誤差が0.0001より大きくならないのであれば，心配不要である．倍精度浮動小数点数を使って普通に計算していれば，誤差が 1.0×10^{-4} より大きくなることは，まずあるまい．

以下では，入力した点の個数と位置が次のような変数に記録されていると仮定する．

```
#define Max_Points    300
typedef struct { double x, y; } pair;

int  n;
pair point[Max_Points];
```

pairは，xとyの2つの実数値の組を表現するための型である．基本的には，点の2次元座標値を記録するために使うが，ベクトル（2つの座標値の差）に流用する場合もある．変数nに点の個数，配列pointのpoint[0]～point[n-1]に各点の位置を記録する．

■2つの点を円周上に持つ円を調べる

幾何の問題の例にもれず，この問題にはいろいろな解き方がある．これは「点の集合を包含する球」（2002年9月号）や「丸い紙吹雪」（2003年6月号）などと共通している．適用可能な戦略の種類も，これらの問題に似ていると言えるかもしれない．しかし，細かい点になると，問題ごとに個性があり，今までの問題で使った解法の単なる応用というわけにはいかない．以下，具体的な解法を調べていこう．

初めに，出題者が正解として想定していた解法を示す．この方法に気づくことができれば，プログラミングは簡単だし，計算量も大したことにならず，易しい問題だという評に賛成してもらえるだろう．若干の幾何的センスが必要なことは確かだが，このくらいは気づいてもらいたいものだと思う．

n 個の点のうち2つを選び，それらの両方を円周上に持つ円を作る．2つの点の距離が2より小さければ，このような円が2つ作れるはずである．2つの点の選び方が $n(n-1)/2$ とおりあるので，全体では最大 $n(n-1)$ 個の円が作れることになる．これらの円それぞれが何個の点を囲むかを勘定し，その中で最大個数の点を囲むものを答とすればよい．

なぜ，この種の円だけを調べればよいのだろうか．それは次のような理屈による．

最大個数の点を囲む円が作れたと仮定する．囲まれている点のどれか1つにぶつかるまで，この円を平行移動していく．すると，点の囲み具合（どの点が円の中で，どの点が円の外か）を変えないまま，点のうちの1つが円周上に乗るようにできる．さらに，その点を中心に回転移動すれば，同様に点の囲み具合を変えないまま，もう1つの点が円周上に乗るようにできる．一般の場合だと，平行移動または回転移動の途中で，今まで円の外にあった点が円の中に入ることがあるのだが，元の円が最大個数の点を囲んでいたという仮定から，この現象は起こらない．もし起こったとしたら，より多くの点を囲む円が得られた，つまり仮定が正しくなかったということになるからである．

以上から，最大個数の点を囲む円の中には，それらの点のうちの2つをちょうど円周上に持つものが必ずある（囲まれる点の最大個数が1の場合は別）．したがって，2つの点を円周上に持つ円をすべて調べれば，その中に正解が必ず含まれている．

計算量を評価しよう．円を1つ決めたとき，n 個の点それぞれが円に囲まれるか否かを判定するのに $O(n)$ の計算量が必要である．円は最大 $n(n-1)$ 個あるから，全体の計算量は $O(n^3)$ ということになる．n は300以下なので，さほどの時間はかからない．

プログラムは次のようなものになる．

```
int  find_max_cover(void)
{
  int  i, j, k, s, count, max_cover;
  int  sign[2] = { +1, -1 };
  double d, e;
  pair m, v, p;

  max_cover = 1;
  for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++) {
      d = hypot(point[i].x-point[j].x,
                point[i].y-point[j].y);
      if (d > 2.0)
        continue;
      m.x = (point[i].x+point[j].x)/2.0;
      m.y = (point[i].y+point[j].y)/2.0;
      v.x = (point[j].x-point[i].x)/d;
      v.y = (point[j].y-point[i].y)/d;
      e = sqrt(1.0-d*d/4.0);
```

```
      for (s = 0; s < 2; s++) {
        p.x = m.x+sign[s]*e*v.y;
        p.y = m.y-sign[s]*e*v.x;
        count = 0;
        for (k = 0; k < n; k++)
          if (k == i || k == j ||
              sq(point[k].x-p.x)+
              sq(point[k].y-p.y) <= 1.0)
            count++;
        if (count > max_cover)
          max_cover = count;
      }
    }
  return (max_cover);
}
```

これ以降，どの解法プログラムについても，関数 find_max_cover として定義することにする．この関数の返り値が，囲まれる点数の最大である．

プログラム中で sq という名の関数を使っているが，これは二乗を求めるためのもので，次のようにマクロ定義してある．

```
#define sq(x)    ((x)*(x))
```

プログラムの中に難しい個所があるとしたら，2 つの点 point[i] と point[j] の両方を円周上に持つ円の中心を求める部分だろう．ここは，図 -2 と各変数の意味から理解してもらいたい．m は 2 点の中点，v は point[i] から point[j] に向かう方向の単位ベクトル（大きさ 1 のベクトル）である．2 点の中点 m から求める円の中心（変数 p，図 -2 では×印）への方向ベクトルは，v の x と y を交換して，一方の符号を逆転したものになる．円の中心さえ求まれば，後の計算は単純明快である．

このようにプログラム自体は簡単なのだが，いくつかの罠が潜んでいる．まず，変数 max_cover の初期値を 1 としている点に注意してほしい．最大値を求めるプログラムの通例にならって 0 とすると間違いである．点がバラバラに存在していて，2 つの点を囲む円が作れないような状況があり得る．この場合の正解は 1（1 個の点を囲む円は常に作れる）なのだが，最大値の初期値を 0 とすると，0 が答になってしまう．

2 つの点を通る円の中心と n 個の点それぞれの距離が 1 以内かどうか調べているループにも注意が必要である．

```
for (k = 0; k < n; k++)
  if (k == i || k == j ||
      sq(point[k].x-p.x)+
      sq(point[k].y-p.y) <= 1.0)
    count++;
```

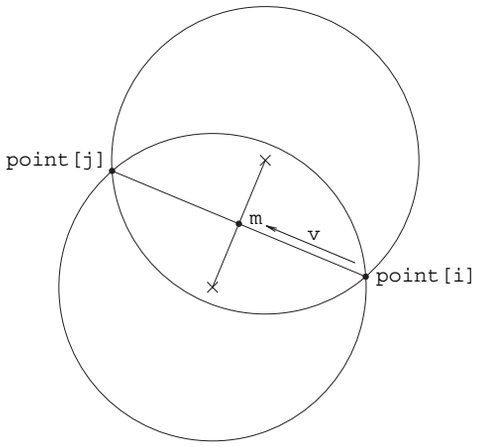

図-2　2点を通る円

元の 2 つの点 point[i] と point[j] だけは別扱いにしている．これは，距離の計算の途中で生じる誤差への対策である．point[i] や point[j] から円の中心までの距離は正確に 1 なのだが，誤差のせいで，1 よりわずかに大きくなったり，小さくなったりするかもしれない．2 点を特別扱いせず，単純に距離だけで判定していると，この 2 つを勘定に入れ損なう恐れがある．

点 point[k] が円の中心 p から距離 1 以内にあるかどうかは，距離の二乗を 1 と比較することによって調べている．これはスピードを考えてのことである．同じプログラムの中でも使っているライブラリ関数 hypot を使えば，距離そのものを簡単に計算できるのだが，それでは平方根の計算を余計にすることになる．距離が 1 より大きいか小さいかだけが分かればよいので，距離の正確な値は不要であり，より計算の速い距離の二乗を使うべきである．距離の二乗の代わりに距離そのものを調べるようにプログラムを変えてみたら，計算時間が 7 倍ほどになってしまった．

■ 1 つの点を中心とする円を調べる

次に示すのは，最初の解法に思い至ることができなかった場合，こういう方法もあるだろうという程度のものである．一般的に言えば，時間がかかりすぎる．実際にこの方法で挑戦したチームもあったらしいが，残念ながら正解に到達したチームはなかったようだ．

n 個の点のうちの 1 つを中心とする半径 1 の円を想定し，その円周を小さな刻みで切っていく．たとえば，円周の 50000 等分のようなことをする．元の点が n 個あるので，円周上の点が全部で $50000 \times n$ 個得られる．こうして得られたそれぞれの点を中心とする円

が何個の点を囲むかを調べる．調べた中の最大個数を問題の答とする．

点のうちの1つを円周上に持つ円だけ調べればよいという理屈は，最初の解法で述べたのと同様の考察で裏付けられる．2つの点を円周上に持つケースよりは考えやすいだろうか．最初の解法と違って，円周上のどこかまでは決められないので，細かく切って片端から調べるという強引な手法である．

プログラムは次のようになる．

```
int  find_max_cover(void)
{
  int  i, k, count, max_cover;
  double a;
  pair p;

  max_cover = 1;
  for (i = 0; i < n; i++)
    for (a = 0.0; a < 2.0*M_PI;
         a += Delta_Angle) {
      p.x = point[i].x+cos(a);
      p.y = point[i].y+sin(a);
      count = 0;
      for (k = 0; k < n; k++)
        if (k == i ||
            sq(point[k].x-p.x)+
            sq(point[k].y-p.y) <= 1.0)
          count++;
      if (count > max_cover)
        max_cover = count;
    }
  return (max_cover);
}
```

変数aは，点point[i]から見た円周上の点の角度を表している．この値を0から2πまで小さな刻みで増やす．角度aが決まれば，円周上の点pは三角関数の簡単な計算で求まる．

この方法で問題となるのは，定数Delta_Angleすなわち角度の刻みの決め方である．筆者のプログラムでは，Delta_Angleを次のように定義している．

```
#define Delta_Angle  0.006
```

刻み幅をこの値に決めれば，審判団が用意した判定用データに対して正しい答を得ることができる．計算時間も最初のプログラムよりははるかに遅いが，十分に耐えられる範囲内である（筆者のやや古い計算機で90秒弱）．

しかし，実際のコンテストの際，刻み幅をこれほど大きくできるだろうか．問題文に与えられている条件だけに頼るなら，この方法の刻み幅は0.0001まで小さくしなければならないはずである．すると，計算時間は60倍になって，コンテストの時間内に2組のデータを処理し終えられるかどうかきわどいことになる．日本における国内予選では，プログラムの実行時間に制限を設けていないが，3時間のうちに2組のデータに正しい答を返さなければ，正解したとは認められない．刻み幅0.0001だと難しいところだろう．プログラミングが非常に簡単なので，絶対に無理だとは言い切れないのだが．

また，筆者の計算機より速い機械が使える場合は，余計なことを考えずに挑戦してみる価値があるかもしれない．計算の量は，$(2\pi/0.0001) \times n^2$と簡単に概算できるので，あながち無謀な挑戦とも言い切れない．

審判の立場から見ると，刻み幅0.006程度で正解が得られるというのは残念なことである．もっとデータを工夫して，小さな刻み幅でないと正解にならないようにしたいところだ．しかし，問題の項で述べた制約（特に3番目の制約）に違反しないようにデータを作ることは至難の技であり，この程度で妥協せざるを得なかった．

■水平分割法

本連載2003年6月号の「丸い紙吹雪」で解説されている水平分割法をそのまま利用することによって解くことも可能である．ただし，この問題の解法としては，やや大げさに過ぎ，プログラミングが複雑になるので，実用的とは言えない．こんな考え方もあるという程度の解説にとどめることにしよう．

半径1の円によって何個点を囲むことができるかは，各点を中心とする半径1の円がいくつ重なり合うかと等価である．つまり，各点を中心とする半径1の円が一番多く重なり合う場所を見つけられれば，元の問題を解いたことになる．円の重なり具合は，いくつかの円からなる図形全体を水平線で切り，左からスキャンすることによって調べられる．円の左半分に交わったときは重なりを1増やし，右半分に交わったときは重なりを1減らすだけのことである．

水平線は，円の重なり具合の変化する位置ごとに1本ずつ調べるだけで済む．具体的には円の上下端および円と円の交点のy座標をすべて求めて，それらをソートし，2つのy座標の間に1本ずつ水平線を設定すればよい．

筆者は実際にこの方法でプログラムを書いてみた．計算時間は90秒強程度だった．実際のプログラムの紹介は省略する．

■分割統治法

幾何の問題は，分割統治法で解けることが多い．「点の集合を包含する球」でも「丸い紙吹雪」でも，分割統治法による解法を紹介している．

分割統治法は，今回の問題にも適用可能である．しかも，今までの問題と違って，分割統治法が非常によい結果を生む．最初の解法よりさらに数倍速く，筆者の知る限り最も高速なのがこれから紹介する解法である．筆者は，2人のエキスパートに独立にこの方法を教えてもらった．

基本的な考え方は，求める円（最大個数の点を囲む円）の中心が存在し得る領域を徐々に絞り込んでいくことである．最初は，点すべてを含む大きな領域から出発し，領域を分割しながら，だんだん小さくしていく．その途中で，最大個数を得られる見込みがないと判明した領域は捨てる．

具体的には次のようにする．上の説明で述べた領域として，ここでは正方形を採用する．プログラムは，求める円の中心が存在し得る正方形を常時保持することにする．こうした候補の正方形は，一般には複数個存在する．

図-3を見てほしい．まず，領域（正方形）の中心点から半径1の円を作り，その中に囲まれる点の個数を数える．これがそれまでに得られた最大個数を超える場合は，最大個数を更新する．

このほかに，正方形の中心から四隅までの距離に1を加えた半径を持つ円を作り，その中に囲まれる点の個数を数える．これを大円個数と呼ぶことにしよう．こちらは，その領域をそれ以上詳しく調べる必要があるか否かの判定用である．正方形の中に中心を持つ半径1の円は，この大きな円に必ず包含されている．したがって，大円個数は，領域の中に中心を持つ円が囲む点の個数の上限を与える．正方形の中のどこに円の中心を持ってきても，大円個数より多くの点を囲むことはできない．

大円個数がそれまでに得られた最大個数以下の場合，その正方形領域の中をいくら調べても新記録が得られる可能性はない．したがって，その領域は捨ててよい．

大円個数が最大個数より多い場合は，その領域の中により多くの点を囲む円を作れる可能性がある．この場合，正方形領域を縦横半分ずつ，合計4つの領域に分ける．新しくできるそれぞれの領域もまた正方形である．このそれぞれの領域に対して，これまでに述べた処理を再帰的に適用する．計算幾何の分野で，4分木（quadtree）と呼ばれる技法と同様である．

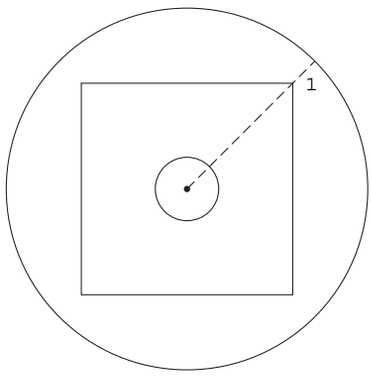

図-3　正方形領域

以上の方法で解が求まるのだが，実はこの手順を単純な再帰呼出しで記述したのではうまくいかない．計算時間がかかりすぎるのである．そこで，候補として残っている領域の中で一番よさそうな領域，つまりより多くの点を囲む円が作れそうな領域から先に調べるという工夫を加える．具体的には，大円個数が多い領域を優先すればよい．最良優先探索の一種であり，A*アルゴリズムの適用例と表現することもできる．

この工夫を加えると，手順の比較的早い段階で，最大個数の点を囲む円が見つかる．それ以外の領域は，この円に劣ることが簡単に分かって，詳しく調べることがほとんど不要になる．

計算時間の下限を理論的に保証することは困難だが，実際にプログラムを走らせてみると，非常に速い．最初に述べた方法より何倍も速く，この問題の解法としては最高速と評してよさそうである．

実際のプログラムを示そう．まず，次のようなデータ構造を用意する．

```
#define Max_Entries    10000

int  max_cover;
int  entry_count;

struct {
  int  u_bound;
  double x, y, size;
} a[Max_Entries];
```

配列aに，候補として残っている正方形領域を記録する．a[i]のメンバの意味は次のとおりである．u_boundにその領域の大円個数を入れる．上に述べたとおり，これがその領域中に円の中心を置いたときに得られる個数の上限になる．xとyは正方形の左下隅の座標，sizeは正方形の一辺の長さである．

```
void save(double x, double y,
          double size)
{
  int  i, count, u_bound;
  double d;
  pair center;

  center.x = x+size/2.0;
  center.y = y+size/2.0;
  count = 0;
  u_bound = 0;
  for (i = 0; i < n; i++) {
    d = hypot(center.x-point[i].x,
              center.y-point[i].y);
    if (d <= 1.0)
      count++;
    if (d <= 1.0+0.7072*size)
      u_bound++;
  }
  if (count > max_cover)
    max_cover = count;
  if (u_bound > max_cover) {
    if (entry_count >= Max_Entries)
      error("too many entries");
    a[entry_count].x = x;
    a[entry_count].y = y;
    a[entry_count].size = size;
    a[entry_count].u_bound = u_bound;
    entry_count++;
  }
}
```

関数 save は，引数で与えられた正方形領域を配列 a に登録する仕事を受け持つ．まず，正方形の中心位置を求め，ここから半径 1 および半径（1+ 正方形の中心と四隅の間の距離）の 2 つの円に囲まれる点の個数を数える．前者は新記録の更新に使うだけである．後者つまり大円個数が新記録以下であれば，この領域はデータ構造に保存する価値がない．大円個数が新記録より大きいときに限って，配列 a に追加する．

```
int  find_max_cover(void)
{
  int  u_max, p, q;
  double x, y, size, half;

  max_cover = 1;
  entry_count = 0;
  save(0.0, 0.0, 10.0);
  while (entry_count > 0) {
    u_max = a[0].u_bound;
    p = 0;
    for (q = 1; q < entry_count; q++)
      if (a[q].u_bound > u_max) {
        u_max = a[q].u_bound;
        p = q;
      }
    x = a[p].x;
    y = a[p].y;
    size = a[p].size;
    entry_count--;
    if (p < entry_count)
      a[p] = a[entry_count];
    if (u_max > max_cover) {
      half = size/2;
      save(x, y, half);
      save(x+half, y, half);
      save(x, y+half, half);
      save(x+half, y+half, half);
    }
  }
  return (max_cover);
}
```

関数 find_max_cover は，最初に空間全体を表す領域 [0,10] × [0,10] をデータ構造に入れてから，領域を 1 つずつ取り出しては調べるループに入る．ループの中の処理の前半は，大円個数が最大であるような領域を求める操作である．単純に配列を前から順に調べることによって最大値を見つけている．それが終わったら，見つかった領域を配列 a から取り除き，分割操作を加える価値があるかどうか調べる．分割操作そのものは簡単で，関数 save を呼び出して，size を半分にした 4 つの正方形を登録するだけである．

このプログラムで配列 a に対して加えている操作は，新しいデータの追加と最大値の取出しの 2 種類だけである．これは優先順位付き待ち行列（priority queue）に相当する．周知のとおり，ヒープなどのデータ構造を使えば，高速な優先順位付き待ち行列を実現することが可能で，プログラムのスピードアップが期待できる．

しかし，この問題の場合，そこまで頑張る必要はないと思われる．配列 a に登録される領域の総個数を調べてみると，1000 を超えることは滅多にない．ある時点で配列に入っている領域の個数の最大も，100 程度にしかならないようだ．これなら，ヒープを使わず，単純に配列に詰め込んで，いちいち最大値を求める方法でも十分間に合う．

単純な配列に対するヒープの優位性が重要な意味を持つようであれば，言語として C を使うのは愚策になるところだった．C++ や Java なら，高速な優先順位付き待ち行列がライブラリによって提供されているからである．こういう結論にならなかったのは，筆者のような原始的 C プログラマにとって幸いなことであった．

（平成 16 年 10 月 13 日受付）