円周上の点による最大面積多角形

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

今回の問題は，2000 年 11 月につくばで開催されたアジア地区予選の問題 G, Telescope である．

半径 1 の円周上に n 個の点が与えられる．そのうちの m 個を選んで作られる多角形（m 角形）のうちの最大面積を求めるのが問題である．n, m はあらかじめ与えられ，さらに各点の位置は $0 \leq p < 1$ なる実数値として与えられる（1 は一周，すなわち 360 度を表す）．問題の規模は $3 \leq m \leq n \leq 40$ となっている．図**-1** に問題の解答例を示す．

最も単純に考えると，n 個の点から m 個を取り出す組合せをすべて生成し，その面積を求めて最大値をとればよい．しかしそのような組合せは ${}_nC_m = n!/(m!(n-m)!)$であり，これはつまり n の指数オーダの時間を要することになる．実際，この問題の規模では ${}_{40}C_{20} \simeq 1.4 \times 10^{11}$ だから現在の計算機でそのまま計算するのは時間的に少し苦しいことになる（問題の規模さえ十分小さければ，こういった力任せの方法も悪くはない）．

このような指数オーダの時間を要する組合せ問題の計算量をいかに減らすか，が今回のテーマである．

■プログラムの準備

実際の問題では，入力データの形式が決まっており，それに合うようにプログラムすることも重要な課題である．しかし，本稿では本質的部分に集中できるように，入力データはすでに変数に格納されているとする．

```
#define NMAX 40
#define PI 3.14159265358979323846
int n;          /* 点の総数 */
int m;          /* 使用する点の個数 */
double p[NMAX]; /* 各点の円周上の位置 */
```

また，2 点 i, j が与えられた時に，円の中心とそれらの点から構成される二等辺三角形の面積を求める関数

```
double area(int i, int j)
```

も定義してあるとする．この面積の求め方は親切にも問題文中に示されており，

```
sin((p[j]-p[i])*2*PI)/2
```

である（標準ライブラリ関数 `sin` の引数の単位はラジアンである）．

この準備があると，3 点 i, j, k（反時計回り順）で囲まれる三角形の面積は

```
area(i,j)+area(j,k)+area(k,i)
```

として求めることができる．ここで注意しなければならないのは，この三角形が円の中心を含まない場合である．そのときには上式の 3 項のうちいずれか 1 つが負になるが，その正当性は図**-2** を見れば明らかである．ところで，問題文中に示された面積の求め方では，2 点のなす角が 180 度未満の場合のみを紹

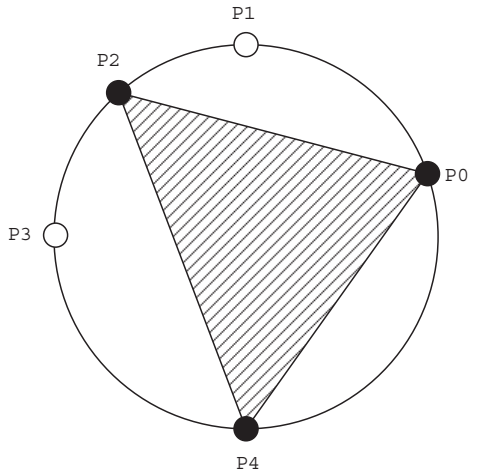

図**-1**　問題の解答例

n=5, m=3 のときの解の例を示す．黒丸は採用した点，白丸は採用しなかった点を表す．

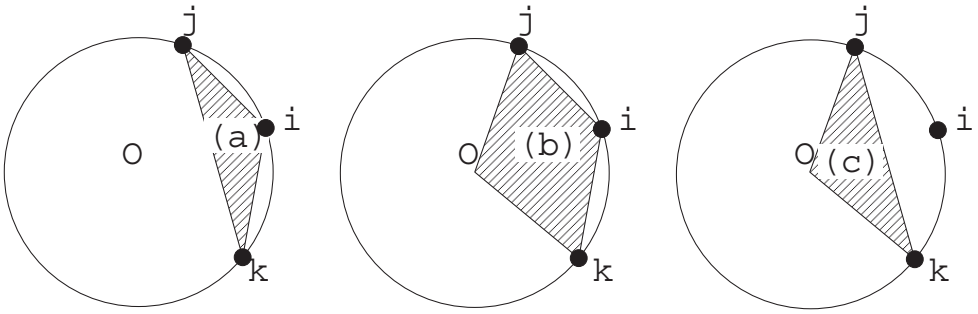

図-2　三角形 ijk の面積

求める多角形 ijk の面積(a)は，Oij, Ojk, Oki の面積の和となるが，Ojk の面積(c)は j, k のなす角が 180 度以上あるため負となり，符号なしで考えると Oij+Oki の面積(b)から(c)を引いたものが(a)となる．

介してあるだけで，このように 180 度を越える場合には面積が負になることがあることには触れていない．このことはのちのち重要な意味を持ってくるのだが．

作成目標である最大面積を求める関数は

```
double calc(void)
```

とすることにしよう．

なお，本稿では「*n* 番目」とある場合，先頭を 0 番目と数えることとする．また，以下のプログラムでは，最大値，最小値，絶対値を求める MAX，MIN， ABS なるマクロを使っているがこれらはしかるべく定義されているものとする．

■プログラム 1 ：生成 & テストタイプ

まず，最も単純にすべての組合せを生成しては面積計算を行うプログラムを示す．

```
int nodes[NMAX];
/* mi 番目の採用点として
   点 ni を検討する */
double place(int mi, int ni)
{
  double r;
  if(mi==m){
    /* 採用点が揃ったら
       多角形の面積計算 */
    int i;
    r = area(nodes[m-1], nodes[0]);
    for(i=0; i<m-1; i++)
      r += area(nodes[i], nodes[i+1]);
  } else if(ni<n) {
    double r1,r2;
    /* 点 ni をスキップ*/
    r1 = place(mi, ni+1);
    /* 点 ni を採用 */
    nodes[mi] = ni;
    r2 = place(mi+1, ni+1);
    r = MAX(r1,r2);
  } else r = 0;
  return r;
}
double calc(void)
{
  return place(0, 0);
}
```

関数 place(mi,ni) は，*mi* 番目の採用点として *ni* 以降を候補とした時の最大面積を返す関数である．内部では，点 *ni* を見送るか採用するかで 2 通りの再帰呼出に分岐し，そのうちの大きい方を返すことにしている．各点の採用状況は大域配列 nodes[] に記録していき，再帰末端の面積計算で利用する．

このプログラムは明らかに指数オーダを必要としている．

■プログラム 2 ：単純再帰タイプ

前章のプログラムは大域配列を使用しており引数だけでは値が定まらず，このままでは改良が難しい．そのため，指数オーダには違いがないのだが別の書き方のプログラムを導入する．

mi 番目の採用点 *ni* までの採用状況が定まっていてその次の採用点を選ぶ状況を考える．あと $(m-mi-1)$ 個の採用点を選べばよい．次の採用点 *nj*

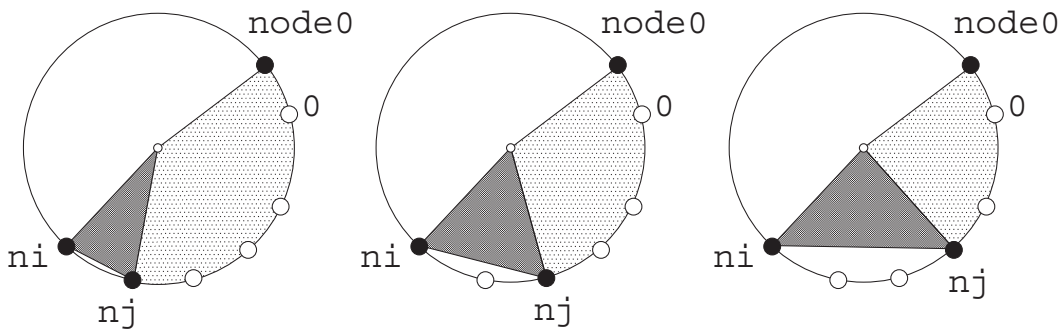

図-3　単純再帰方式

点 *ni* から終端点 *node*0 までの最大面積を求める．*ni* とその次の採用点 *nj* の部分（色の濃い三角形）と，*nj* から先の部分（色の薄い扇形）に分け，後者は再帰呼出によって最大面積を求める．それらのうちで和が最大面積を与える *nj* を採用する．

を，*ni* のすぐ次(*ni*+1)から始めて順次あとの点を候補とする．*nj* から先は再帰的に最大面積が求まるとして，*ni* と *nj* の一区間の面積と，再帰的に求まった *nj* から先の面積の和が *ni* からの面積となり，これを最大化すればよいことになる（図-3）．

この問題では，最初の採用点から始めて一周する形で計算を進めるので，区間の終端点はすなわち最初の採用点となり，これは計算を通じて固定できる．そこでこれを node0 なる大域変数に保持することにする．

関数 place は，*mi* 番目の採用点として *ni* を選んだ時の最大面積を返す関数である．最初の採用点 node0 は最も外側のループで制御する．

```
int node0;
/* mi 番目の採用点として
   点 ni を検討する */
double place(int mi, int ni)
{
  double max_s;
  if(mi==m-1){
    /* 一周したら最後の部分の面積 */
    max_s = area(ni, node0);
  } else {
    double s;
    int nj;
    max_s = -PI;
    for(nj=ni+1; nj<n; nj++){
      /* 次の採用点候補を変えつつ
         多角扇形の面積を求めて */
      s = area(ni, nj) + place(mi+1, nj);
      /* その中の最大値をとる */
      max_s = MAX(max_s, s);
    }
  }
  return max_s;
}
double calc(void)
{
  double s, max_s;
  max_s = 0;
  /* 最初の採用点を変えるループ */
  for(node0=0; node0<n; node0++){
    s = place(0, node0);
    max_s = MAX(max_s, s);
  }
  return max_s;
}
```

このプログラムでは，最大面積の初期値として，0 ではなく -PI を与えている．これは求めているのが部分面積であるから前述のとおり負になることがあり得るためで，ここを 0 としてしまうと間違った答になってしまう．

■プログラム 3 ：関数の結果を記憶

前章のプログラムの実行効率が悪い原因は，同じ関数を何度も評価していることにある．実際，引数である *mi*, *ni* はそれぞれ *m*, *n* で抑えられるのであるから，異なる呼出は高々 *node*0 ごとに *n*×*m* 種類しかない．この関数の値は，引数で完全に決定されるので，関数の呼出結果を表に記憶することを考える．2 次元配列 double memo[NMAX][NMAX] を用意し，初期値としては「未計算」を表す特別な値を入れて

おく．

計算関数 place が呼ばれると，まず引数をキーとして memo 配列の該当する場所をチェックし，すでにその引数の組について計算済みであればその値を直ちに返す．「未計算」の値が見つかった場合には前項の処理を正直に行って値を計算し，memo 配列に格納する．

この方式によれば，place 関数の呼出は引数の組に対して高々 n 回だけになり，全体の計算量は $n^3 \times m$ で抑えられることになる（引数の組が $n \times m$ だけあり，1 回の呼出によって下請け関数が n 回呼び出される．さらに，初の採用点 node0 の選び方が n 通りある）．

こうした手法を**索表計算 tabulation** と呼び，古典的な高速化手法の 1 つである．

■プログラム 4 ：動的計画法

単純再帰プログラムでは，

$$place(mi, ni) = \max_{ni<nj<n} (area(ni, nj) + place(mi+1, nj)) \quad (1)$$

のように再帰的な定義を行った．ここで，前項と同様に関数呼出し $place(mi, ni)$ を配列要素 $memo[mi][ni]$ に対応させれば，これは関数呼出の関係ではなく，配列要素間の関係を示す漸化式とみることができる．$memo[mi][ni]$ の値そのものは前項と同じ値となるが，ここではさらにその具体的な計算順序が与えられたことになる．漸化式

$$memo[mi][ni] = \max_{ni<nj<n} (area(ni, nj) + memo[mi+1][nj]) \quad (2)$$

にしたがって，mi の最大値 $m-1$ から始めて 0 に向かって計算していけばよい．この場合にも，プログラム全体の計算量は $n^3 \times m$ で済む．

この問題の場合，さらに空間的な計算量を減らすことが可能である．式 (2) から，memo の第一の添字に注意すると，1 つ先の行だけを使って計算していることが分かり，ある行の計算が終了すると先の行は不要になる．さらに，1 行の中で ni の要素を求めるには $ni+1$ から先だけを使っている．以上をまとめると，mi を $m-1$ の行から始めて，行内では ni を最も小さい値から始めていけば，memo 配列は 1 次元で済むことになる．

以上を反映させたプログラムを次に示す．

```
double memo[NMAX];
double calc(void)
{
  int node0, ni, nj, mi;
  double s, max_s, result;
  result = 0;
  /* 出発点についてのループ */
  for(node0=0; node0<n; node0++){
    /* 表の末端行 m = mi-1 の処理 */
    for(ni=node0; ni<n; ni++)
      memo[ni] = area(ni, node0);
    /* 行を大きい方から埋めていく */
    for(mi=m-2; mi>=0; mi--)
      /* 行内は小さい方から */
      for(ni=node0; ni<n; ni++){
        max_s = -PI;
        for(nj=ni+1; nj<n; nj++){
          s = area(ni, nj) + memo[nj];
          max_s = MAX(max_s, s);
        }
        memo[ni] = max_s;
      }
    result = MAX(result, memo[node0]);
  }
  return result;
}
```

このように，再帰的な関数の結果を順序よく表に収めていくことによって計算を効率化する手法を**動的計画法 dynamic programming** と呼ぶ．

動的計画法が適用できる問題の特徴は，与えられた問題を小さいサイズの問題に分解した時に，その親問題の解が子問題のいずれかから得られる点である．

この問題のような組合せ最適化の問題では，動的計画法が適用できるかどうかを判断するのが重要である．その意味では，プログラム 2 を思いついた段階で動的計画法の要件を満たしているということができる．

この問題について，動的計画法の適用に失敗する考え方をあげよう．面積最大 m 角形を作る際に，面積最大 $m-1$ 角形を出発点とし，それにいずれか 1 つの新しい頂点を加えることで面積最大 m 角形としようとするのである．この考え方がうまくいかないのは，正方形と正三角形が重なった点が与えられた場合，正三角形に新しい 1 点を加えることによっては正方形は作れないことを考えると分かる．親問題の解（正方形）には，子問題の解（正三角形）は含まれ

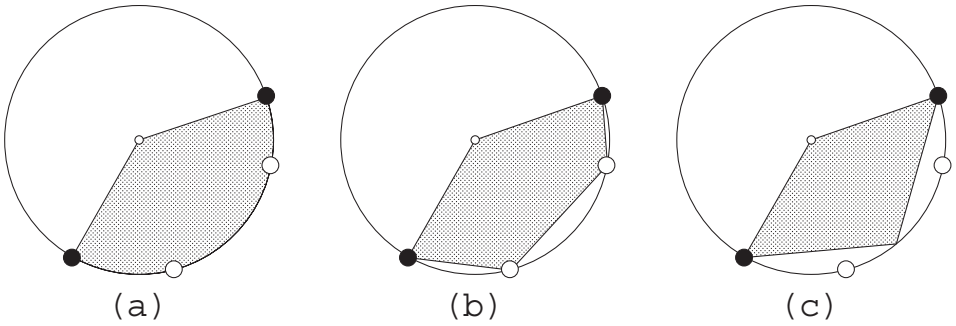

図-4　上界の例
ni も含めた残りの点の総数 *n*−*ni* は 3，採用すべき点の数 *m*−*mi* は 2 の場合.

ていないのである.

■プログラム 5 ：分枝限定法

この問題については，前章の動的計画法が最適な解であり，出題者もおそらくそれを想定していると思う.

しかし，ここではさらに別の手法によって，動的計画法を用いずに単純再帰を高速化することを考えよう．こうした探索の高速化は，状態空間（関数に与えられる引数の組）が大きすぎて索表や動的計画法が適用できない場合などに使われる.

まず，分枝限定法 **branch-and-bound method** を適用する．これは状態空間を全探索する場合に，「明らかに見込みのない枝」を探索候補から除外していくことで探索の高速化をはかるものである．枝を刈ることによって，その先に展開される膨大な（意味のない）計算を省けるので，状況によってはきわめて有効な方法である.

「明らかに見込みがない枝」とは，その枝の先でいかに良い結果が出たとしてもそれまでの最高記録（暫定解）を更新し得ないということで，ある種の上界を計算することになる．また，この判断は十分高速に（すなわち，探索によることなく）行う必要がある.

この上界をどう（きびしく）定めるかがポイントで，この問題の場合，*mi* 番目の採用点を *ni* とした時の，その先の最大面積の上界として，たとえば
(1) *ni* から終点までの円弧による扇形（図-4 (a)）
(2) *ni* から終点までのすべての点から作られる (*n*−*ni*)角扇形（図-4 (b)）
(3) *ni* から終点までの正(*m*−*mi*)角扇形（図-4 (c)）
などが考えられる．明らかに (1) は (2)，(3) に比べて大きい（甘い）上界であって，採用する必要はない．(2)，(3) はデータによってどちらがより小さい（きびしい）かが決まるので，両方を計算してその小さい方を採用するのが妥当だろう.

このような枝刈り **pruning** を導入すると，今度は探索の順序が重要になる．結果の良い枝を先に探索しておくと，早期によい暫定解を手に入れることができるので，それ以降の枝を有効に刈りとることが可能になる．プログラム 2 では，次の採用点 *nj* を現在点 *ni* の次から順に探索していたが，これを「より見込みのある」ものを先にするように並べ替えるのである．この問題では，正多角扇形が面積最大であることに着目して，残り角度の平均値に近い *nj* から探索するのがよいだろう.

```
int node0;
/* 点 k から残り r 個の点による面積の上界 */
double upper_bound(int k, int r)
{
  double r1, r2;
  int i;
  /* 正多角扇形 */
  r1 = sin((p[node0]+1-p[k])*PI*2/r)/2*r;
  /* 残り点すべてによる面積 */
  r2 = area(n-1, node0);
  for(i=k; i<n-1; i++)
    r2 += area(i, i+1);
  return MIN(r1, r2);
}
/* 点 i+1 から x 個を，最初の r 等分点に近い順に
   ソートする */
void rearrange(int *vec, int i, int x, int r)
{
  double av;
```

```
  int c, d, k, tmp;
  /* 最初の r 等分点 */
  av = p[i] + (p[node0]+1 - p[i])/r;
  for(c=0; c<x; c++){
    d = i+1+c;
    for(k=0; k<c; k++)
      if(ABS(p[vec[k]]-av)>ABS(p[d]-av)){
        tmp = d;
        d = vec[k];
        vec[k] = tmp;
      }
    vec[c] = d;
  }
}
/* mi 番目の採用点として，点 ni を検討する．
   同レベルでのこれまでの最良値が cur_max */
double place(int mi, int ni, double cur_max)
{
  double s, max_s;
  int nj;
  if(mi==m-1){
    max_s = area(ni, node0);
  } else {
    int v[NMAX];
    int nj_i, jcand;
    max_s = cur_max;
    /* 並べ替えの候補数 */
    jcand = n - ni - m + mi + 1;
    rearrange(v, ni, jcand, m-mi);
    /* 有望な候補から順に */
    for(nj_i=0; nj_i<jcand; nj_i++){
      double s1, s2;
      nj = v[nj_i];
      s1 = area(ni, nj);
      s2 = upper_bound(nj, m-mi-1);
      if(s1+s2 > max_s){
        /* 見込みがあれば 実際に計算 */
        s = s1 + place(mi+1, nj, max_s-s1);
        max_s = MAX(max_s, s);
      }
    }
  }
  return max_s;
}
double calc(void)
{
  double s, max_s;
  max_s = 0;
  for(node0=0; node0+m<=n; node0++){
    s = place(0, node0, max_s);
    max_s = MAX(max_s, s);
  }
  return max_s;
}
```

これらの手法の効果を，コンテストの問題に付加されているサンプルデータについて計測してみる．n=30, m=15 の問題に対して，単純再帰のプログラムにおける `place` の呼出回数は 614429671 であったが，枝刈りを導入することで 60916 に激減し，さらに探索順序の変更によって 15119 にまで減少している．

しかし残念ながら，この上界では n=40 というコンテストの最大サイズでは計算が終了せず，より精密な上界の評価が必要である．

■おわりに

動的計画法を頂点とする，組合せ的な探索問題の最適解を求めるためのアプローチを紹介してきた．これとは別に，最適性を犠牲にしても短時間でそれなりの解を求めるアルゴリズムが各種存在する．状態空間が巨大すぎる場合や，解の最適性にそれほどの意味がない場合などにはそうした方法が適用可能である．

プログラミングコンテストの観点から見れば，問題のサイズによってどのようなアプローチをとるべきかがほぼ決まってくるので，そこを見抜くのが 1 つの鍵となるであろう（現実の問題ではかならずしもそれほど単純ではあり得ないが...）．

（平成 14 年 5 月 10 日受付）