最適配置決定のアルゴリズムとシミュレーション・プログラムの実装

大　角　盛　広

概要：本論文では，需要点が離散的に分布する市場において，競合する2企業が獲得購買力の最大化を目的として交互に施設を配置する問題を扱う。このタイプの競合施設配置問題の解を求めることは理論的に困難な場合が多いため，コンピュータでシミュレートすることも現実的な手段として有効である。ここでは，そのためのシミュレーション・プログラムの実装技法について考察した。ただし，特殊な場合のみに適用できるようなコーディングは避け，一般的な問題に適用できるように，汎用性についても考察を加えた。

キーワード：配置問題，非協力ゲーム，アルゴリズム，シミュレーション

1　はじめに

本論文では，需要点が離散的に分布する市場において，競合する2企業が獲得購買力の最大化を目的として交互に施設を配置する問題を扱い，これを定式化するとともに，この配置ゲームをコンピュータでシミュレートする手法について考察する。

競合環境の下での施設の配置問題は，直線上での競合を扱った Hotelling [1]の研究に始まった。Hakimi[2]はネットワーク市場に2企業が順に施設を配置する場合のシュタッケルベルグ均衡を扱い，先手企業の最適配置を求める問題が NP-hard であることを示した。Drezner[3] は Hakimi のモデルを平面上に拡張した。

先手後手のある競合施設配置問題では，先手は後手が最善手を採ることを考

慮に入れた上で自らの戦略を決定せねばならない。需要点の分布に特に仮定を設けず一般の場合でこれを理論的に解くことは非常に難しいため，これらの問題に対してシミュレーション・プログラムの開発を行うことには実用的な意義がある。また，シミュレーションにより解の振る舞いを調べることで，解の満たす性質が解明されることもあろう。

本論文では，獲得購買力と利得とを同等と仮定し，購買力の到達範囲に限界がある場合と限界のない場合，および平面上の競合と直線上の競合とについて，C言語によるプログラムの実装手法を論じる。ここで実装した平面走査法やminimax 探索における分枝限定法は，他の問題にも応用の利くものである。

なお，プログラムが煩雑になりすぎないように，本来動的にメモリを割り当てるべき箇所であっても，特に仕掛けをほどこさず単純に配列を利用して提示している箇所がある。

2　購買力の到達範囲に限界距離のある場合

施設配置問題では，客は，施設までの距離の遠近によってどの施設を利用するかを決定する，と仮定することが多い。このとき，空港や専門店や病院のように，施設までの距離が遠くても出かけていく施設と，ファーストフード店や自動販売機のように，施設までの距離が遠いとわざわざ出向かない施設とが考えられる。

購買力の到達範囲に限界距離がない場合には，どの客の購買力も競合する施設のいずれかによって獲得されることになるから，先手後手ともに相手の利得を減らすことが自己の利得の最大化と同等になる。

限界距離がある場合には，最初に出店する先手企業が，その場面で最も多く取れる場所に置かずに，後手企業に取り分を残しておいた方が最終的に利得を多くできる場合がある。この具体的な例については，大角・塩出 他[6]で述べられている。

本節では，平面上に n 個の需要点が存在する市場に，2企業A，Bが順に1

つずつ施設を配置する問題を扱う。市場に関して以下の仮定を設ける。

1. 客は一定の限界距離以上には出かけない
2. 客は限界内の施設は均等に利用する
3. すべての客の限界距離は等しい

需要点の位置を p_i，その重み（購買力）を w_i とする。任意の位置 x, y の間の距離は距離関数 $d(x, y)$ で表されるとし，限界距離を r とする。このとき，p_i の客が出かける限界領域 $D(p_i)$ とその境界 $B(p_i)$ は，

$$D(p_i)=\{x \mid d(x, p_i) \leq r\},\ B(p_i)=\{x \mid d(x, p_i)=r\}$$

と表せる。一方，先手Aの施設の位置を a，後手Bの施設の位置を b とし，A，B側から客の位置をみると，A，Bが客を取り込める勢力圏 $D(a), D(b)$ は次のように表せる。

$$D(a)=\{x \mid d(x, a) \leq r\},\ D(b)=\{x \mid d(x, b) \leq r\}$$

メジアノイド問題は a が与えられたとき，その a に対するBの最適反応を求める問題である。すなわち，利得を購買力の総和そのものとして，Bの利得関数を f_b とすると，

$$f_b(b \mid a)=\frac{1}{2}\sum_{p_i \in D(a) \cap D(b)} w_i+\sum_{p_i \in \overline{D(a)} \cap D(b)} w_i$$

を最大にする b を求める問題である。この解を $b^*(a)$ とする。なお，a に対して f_b を最大にする b^* が複数存在する場合は，任意の代表点を b^* と定めるものとする。企業Bが後からB自身に最適な位置に施設を置くことを考慮に入れた上で，先手Aの採る最適配置を求めるのがセントロイド問題である。すなわち，Aの利得関数を f_a とすると，

$$f_a(a)=\frac{1}{2}\sum_{p_i \in D(a) \cap D(b^*(a))} w_i+\sum_{p_i \in D(a) \cap \overline{D(b^*(a))}} w_i$$

を最大にする a を求める問題である。

ここで，市場の総購買力を W とすると，両方の施設から r より遠く離れている需要点が存在することがあるため，$f_a(b \mid a)+f_b(a) \leq W$ である。従って必ずし

も相手の購買力を減らすことが自己の購買力の増加につながらない。

以上は一般の距離関数で定式化したが，以降では直角距離を用いることにする。すなわち，$x=(x_1, x_2)$ と $y=(y_1, y_2)$ の距離は $d(x, y)=|x_1-y_1|+|x_2-y_2|$ で表されるものとする。このとき，$B(p_i)$ や $D(a)$ は視覚的には 45° 傾いた矩形となる。直角距離は，都市部ではユークリッド距離よりも良い近似を与えることが知られている。

3　平面上での解の探索方法とプログラムの実装

購買力の到達範囲に限界距離のある場合，施設の最適解のうちの少なくともひとつは客の限界領域の境界に存在することが，大角・塩出 他[6]によって示されている。同様に，限界距離にあいまいさを取り入れた場合も，最適解のうちの少なくともひとつは，客が必ず近いと感じるような限界領域の境界に存在することが大角・塩出[7]によって示されている。

従って，最適解をひとつ見つければ充分な場合，境界 $B(p_i)$ のみをチェックすれば良い。また，特殊な状況を除いて限界領域同士が交わることを考えると，境界同士の交点を解の候補点として列挙すれば良いことになる。すなわち，施設配置の実行可能領域は平面上の任意の点であるが，解の候補としては離散点のみを考えれば良いことになる。

境界の交点を求めるには，計算幾何学の手法を用い，多角形の交点を $O(n \log n)$ で列挙する平面走査法の考え方が応用できる[9]。

ただし，ここでは，ただ交点を列挙すればよいのではなく，ある場所での利得関数 f_a, f_b の値を具体的に求めて，他の場所での場合と比較する必要がある。すなわち，どの交点の位置でいくつの需要点を勢力圏に入れることができるかという問題となり，領域の重なりの数は最悪の場合で n^2 に比例するからこれを $O(n \log n)$ のアルゴリズムで求めることはできない。

まず最初に行うことは，$B(p_i), B(p_j)$ $(i \neq j)$ の交点を列挙することと，その点がどの需要点の限界距離内に入っているかをリストに記録することである。

平面走査法を適用する前処理として，座標変換によりすべての点を時計回りに 45° だけ回転させる。以後，変換後の座標を用いることにする。

矩形 $B(p_i)$ を管理するデータ構造を以下のように定義する。需要点の位置 p_i は $B(p_i)$ の左下の頂点の位置と1対1に対応しているので，取り扱いを簡便にするため，ここでは頂点の座標を用いる。

```
struct rectangle {          /* 限界範囲を表す構造体 */
  int      id;              /* 需要点の番号 */
  double   x;               /* 矩形の左下頂点のｘ座標 */
  double   y;               /* 矩形の左下頂点のｙ座標 */
  double   w;               /* 重み(需要量) */
  double   remaining_w;     /* 残されている重み */
} Rect [ N_OF_RECT ];
```

後手Ｂが施設を配置するときに，ある需要点から獲得できる需要量を remaining_w で表す。扱える需要点の数は N_OF_RECT で表し，実際に使われているデータ数は大域変数 NumOfRect が保持するものとする。ここでは Rect は配列として定義するが，データとして与えられる点の数の範囲が具体的な問題ごとに大きく変わる場合は，リスト構造にすることも考えられる。以下で定義する配列に関しても同様である。

平面走査法では，前処理として，ｙの昇順に整列した Rect をさらにｘの昇順に整列する。その後，各 $B(p_i)$ の x 軸に平行な2本の辺を無限に伸ばして，平面を帯状のスラブ領域に分割する。このスラブ領域を，y 座標の小さい順に0番から昇順に番号を付けて次のデータ構造で保持する。

```
struct slab {
  double   y;               /* スラブのｙ座標(上側) */
  int      *idp;            /* このスラブと交わる矩形の id のリスト */
} Slab[ N_OF_RECT * 2 ];
```

ここも，実際に使っているデータ数は大域変数 NumOfSlab が保持するものと

する。

yに入る値は，スラブの上側の辺の y 座標値とする。このスラブは，$B(p_i)$ の縦の辺によって横切られているので，その縦の辺を管理するデータ構造を以下のように定義する。

```
struct edge {
  int      id;  /* この辺の属する矩形のid */
  double   x;   /* この辺のx座標 */
  boolean left; /* この辺が矩形の左側の辺なら TRUE，右なら FALSE */
} Edge[ N_OF_RECT * 2 + 1 ];  /* EOD 用に1つ使う */
```

ここで，boolean 型は typedef enum {FALSE=0, TRUE} boolean; で定義されているものとする。EOD は End Of Data を表す記号定数である。

矩形 $B(p_i)$ の重なりによって分割された平面の領域を管理する構造体を以下のように定義する。ただし，$B(p_i)$, $B(p_j)$ $(i \neq j)$ の交点によって領域を代表させることができるから，プログラム上では交点を領域として扱う。

```
struct domain {
  int       *idp;/* この領域を構成する矩形番号のリスト(空にはならない)
                 */
  double     w;    /* この領域で獲得できる購買力 */
} Domain[ N_OF_DOMAIN ];
```

idp には，この領域（交点）を内側に含んでいる $D(p_i)$ の点番号 i のリストが記録される。

以上でデータ構造が用意できたので，まず以下の手続きでスラブsと交わる矩形 id を Slab[s].idp に列挙することから始める。

```
find_intersection(void)
 {
  int  i, j, s, r;
  int  list[ N_OF_RECT ];
```

```
  int  *p;

  for (s=0; s<NumOfSlab; s++) {    /* スラブ番号 s を走査 */
    i = 0;
    for (r=0; r<NumOfRect; r++) { /* 左側下の矩形から右上に向かってチ
                                     ェック */
      if (Rect[r].y <= Slab[s].y && Rect[r].y + R_WIDTH>=Slab[s+1].y) {
        list[i] = r;              /* スラブと交わる Rect の添字を保存 */
        i++;
      }
    }
    Slab[s].idp = p =(int *)memalloc( sizeof(int) * i + 1 );
    for (j=0; j<i; j++) {
      *p = Rect[ list[j] ].id;    /* 確保したメモリ領域に需要点番号を
                                     保存 */
      p++;
    }
    *p = EOD;                     /* End Of Data を入れておく */
  }
}
```

前半部分ではスラブとすべての Rect との交わりを判定しているが，明らかに交わらない範囲にある Rect との比較を除外すると平均的な効率が向上する。ただし最悪の場合の効率は変わらない。

ここでは矩形の1辺の長さを R_WIDTH$=\sqrt{2}r$で表している。R_WIDTH が可変の場合も同様の方法で判定できるが，その場合は Rect に半径のデータを持たせ，前処理として x, y 軸に平行な2本の辺ごとに別々に昇順に整列しておく必要がある。

C言語では標準ライブラリ関数 malloc を使用して動的なリスト構造を実現することが多いが，この場合，1度の malloc によって保存すべき値は需要点番号だけであるから，本格的なリスト構造を用いると malloc のオーバーヘッド

が大きすぎる。このような場合は，OS のメモリ割り当て長の整数倍に揃えたブロック単位で malloc すると，メモリの使用効率が良くなる。さらに free で解放した後の再利用率も高くなることが期待でき，フラグメンテーションの回避にも貢献する。ここでは，下請け関数 memalloc がそれを行っているものとする。

次は，スラブ Slab[s] 内に含まれる矩形の id をもとに，スラブを走査して横切る線分を列挙し，その線分が矩形の左側か右側かの情報も併せて構造体の配列 Edge に保存していく。

```
scan_nth_slab(int s)
 {
  int    i, n, *p;
  struct rectangle   *rp;

  p = Slab[s].idp;      /* スラブと交わる矩形 id リストの先頭アドレス */
  i = 0;
  while ( *p != EOD ) {
    rp = find_rect(*p);     /* *p で示される id 番号を持つ矩形を探す */
    Edge[i].id   = *p;      /* 矩形 id をセット */
    Edge[i].x    = rp->x;   /* 左側辺の x 座標 */
    Edge[i].left = TRUE;
    i++;                    /* 右側辺を別 Edge として記録 */
    Edge[i].id   = *p;      /* 矩形 id をセット */
    Edge[i].x    = rp->x + R_WIDTH; /* 右側辺の x 座標 */
    Edge[i].left = FALSE;
    i++; p++;
  }
  Edge[i].id = EOD;         /* End Of Data を入れる */
  qsort(Edge,i,sizeof(struct edge),comp_edge); /* 全てを x 座標の昇順に整
                                                  列 */
}
```

scan_nth_slab の引数を 0 から NumOfSlab-1 まで 1 ずつ変えながら呼び出して Edge データを作成する。find_rect は，指定された id 番号から該当する Rect 構造体の先頭アドレスを返す関数である。comp_edge は，a の x 座標が b の x 座標より大きいと 1 を返す，小さいと −1 を返す関数である。

scan_nth_slab を呼び出した直後，スラブ内を左から右に走査して $D(p_i)$ の重なり領域を列挙していく必要がある。そのためには，走査中の領域がどの矩形の内部にあるかを記録する表が必要となるので，それを配列 Inside に書き込むことにする。プログラムが煩雑になるのを避けるため，この配列に対する各種の手続きはプログラム中のコメントとして概要のみを示す。

```
boolean Inside[ N_OF_RECT ]; /* 現領域が id 矩形の内部なら Inside[id]=
                                TRUE */
int     Dp = 0;                 /* Domain の添字 */

enum_intersection(int s)
 {
  int i;

  clear_flag();                 /* Inside 配列の内容をクリアする */
  for (i=0; Edge[i].id != EOD; i++) {
                                /* スラブを横切る辺を左から右に走 査 */
    if ( Edge[i].left ) {       /* 矩形の左側の辺を通過すると矩形内 */
      Inside[ Edge[i].id ] = TRUE;
                                /* 矩形番号 id の内部にあることを記録 */
      if (! already_exist_domain()) {
                                /* Inside 配列から列挙済みかを判定 */
        Domain[Dp].idp = dup_d();
                                /* Inside 配列を圧縮し確保領域に保存 */
        Dp++;
      }
    } else {
```

```
        Inside[ Edge[i].id ] = FALSE;    /* 矩形の内部から抜けた */
        if ( ! check_empty() ) {         /* Inside 配列が空でなければ */
          if ( ! already_exist_domain() ) {
            Domain[Dp].idp = dup_d();
            Dp++;
          }
        }
      }
    }
  }
```

同じ矩形 id のみで形成されている領域があれば，すでに列挙したものとして無視すべきである。すでに列挙済みか否かを判定する関数が already_exist_domain である。

配列 Domain が求められると，各 Domain に a, b が存在するときの利得関数値 $f_b(b|a), f_a(a)$ を求めることができる。ここで，最悪の場合でも最大の重みを持つ領域で重みを半分ずつ分け合うという解があるので，最大値の半分未満しか取れない領域が解になることはありえない。従って，最適解の探索において，これを下界として探索を打ち切ることができる。

すなわち，解を求める基本的な方法は，以下のようになる。

1. ある領域 (Domain[ap]) に先手Aが置いたとき，$D(a)$ 内にある需要点の重みを半分にして，後手Bが最大の重みを取れる領域 (Domain[bp]) を探索する。これがメジアノイド問題の解となる。
2. Bの配置が Domain[bp] に決まったとして，最終的にAの取れる重み $f_a(a)$ を再計算する。それが以前に求めた $f_a(a)$ よりも大きければ，Domain[ap] をセントロイド問題の解の候補として残す。
3. 最大の重みの半分の重みの領域までを走査し終えたとき，Domain[ap] がセントロイド問題の解である。

上の方法を疑似コードで書くと，次のようになる。

```
search_location(void)
 {
  int      i, ap, bp;        /* 先手後手の位置 */
  double   fa;               /* Aの利得 */

  Domain[i].wをソートキーとして配列 Domain を降順にソートする
  for (i=0; Domain[i].w >= Domain[0].w/2; i++ ) {
    ap = 0; fa = 0.0;
    すべての Rect について remaining_w をwにする
    Domain[s] と交わる id の Rect[id].remaining_w を半分にする
    すべての Domain についてwを再計算する
    再計算した Domain の中で最大のwを持つ Domain の添字を bp とする
    すべての Rect について remaining_w をwにする
    Domain[bp] と交わる id の Rect[id].remaining_w を半分にする
    Domain[s].w を計算し，fa より大であれば fa=Domain[s].w とし，ap=s
      とする
  }
}
```

この処理において，最終的に Domain[ap] にセントロイド問題の解が求められている。また，途中の bp には，その時の ap に対してのメジアノイド問題の解が求められている。ただし，ここでは最適配置を1点求めれば良いものとしたので，全てを列挙する場合には異なった解の組み合わせも現れうる。

ここで使った方法は，基本的にバックトラックを用いた探索であるから，非常に応用範囲が広い。非ゼロサムゲームのように互いの利得の間に一定の関係がない場合をシミュレートするのに役立つ。

4　購買力の到達範囲に限界距離のない場合

前節では平面上の非ゼロサムゲームを扱ったが，ここでは直線上に需要点が離散的に分布する市場について，購買力に到達限界のない場合，すなわちゼロサムゲームを扱う。

また，前節では1店ずつの配置競争を扱ったが，ここでは交互に何店でも配置できるという多段階状況を考える。ただし，既存施設と同じ点には配置できず，また，設置コスト等の何らかの要因により有限回数で配置競争が終了するものとする。

A，Bの施設の位置を $q_1, \cdots, q_n$ で表す。q_i は i が奇数のときはAの施設，i が偶数のときはBの施設を表す。このとき，q_i のボロノイ領域は，

$$V(q_i)=\{x \mid d(x, q_i)<d(x, q_j),\ \text{for all } j, j\neq i\}$$

と表せる。添字が奇数の q_i を改めて a_k 添字が偶数の q_i を改めて b_k と連番を振り直し，$a=(a_1, a_2, \cdots)$，$b=(b_1, b_2, \cdots)$ とすると，A，Bの勢力圏は，$V(a)=\cup V(a_k)$，$V(b)=\cup V(b_k)$ と表せる。A，Bのいずれかの施設から等距離の点は $B(a, b)$ と表すことにすると，A，Bの利得関数 f_a, f_b は

$$f_a(a, b)=\frac{1}{2}\sum_{p_i\in B(a, b)} w_i+\sum_{p_i\in V(a)} w_i$$

$$f_b(a, b)=W-f_a(a, b)$$

と表せる。ただし，$W=\Sigma w_i$ である。

5　直線上での minimax 探索とプログラムの実装

直線上に離散的に需要点が分布する市場で，客が距離によって施設を選ぶ場合，多段配置問題であっても，施設の位置は需要点上で最適値を取ることが大角・塩出 他[8]によって示されている。従って，線分上の無限の点を走査することなく，需要点のみを走査すれば最適解を見つけられる。

限界距離のない場合には，ゼロサムゲームとなり，相手の取る利得の最大値を最小にするという戦略を採る minimax 問題を解くことになる。すなわち，各自の手番において，先手はゲームの値 f_a を最大にしようとし，後手はゲームの値 f_a を最小にしようとする。

先手後手の最適戦略を求めようとすると，ゲームの木を構成して盤面の先を読む必要があるが，このとき，分枝限定法を用いると効率が向上する。このよ

うな分枝限定法は，プログラミング用語では α-β カットと表現されることが多い。

ここでは，最適解を求めるために再帰的に呼び出して使用する関数 min_max を設計し，そこに α-β カットを組み込むことにした。min_max は先手の利得を返す関数で，先手はこの関数の値が最大になるような戦略を，後手は最小になるような戦略を採る。

この関数を呼び出す前処理として，Vi[0], Vi[1], Vi[2], … に需要点の番号が重みの降順に入れられているものとする。また，変数 list には施設を配置済みの需要点番号が設置順に保存されているものとする。push_one はこのリストに要素を追加，pop_one は追加した要素を削除する関数である。

```
double min_max(int *list, int level, double alpha, double beta)
 {
  int       i, k;
  double  eval, mm_eval;
  double  evlist[ N_OF_VERTEX ];
  double  a, b;

  if ( level >= MaxLevel ) {    /* 読みを打ち切るレベル */
    return( eval_stage(list) );  /* 現状を先手の立場から評価する */
  }
  /* ミニマックス値の初期化(先手番と後手番で異なる) */
  mm_eval = is_leader(level) ? MIN_VALUE : MAX_VALUE;
  for (k=0; i=Vi[k], k<NofV; k++) { /* 最大重みの点から順に */
    evlist [i]  = NON_VALUE;        /* i番の点に置いた場合を初期化 */
    if ( ! is_member(list,i) ) {    /* 未使用点を対象に */
      push_one(list,i);             /* i 番の点に置いたと仮定する */
      /* 次のレベルで使うアルファ値とベータ値を更新 */
      a = is_leader(level) ? mm_eval : alpha;
      b = is_leader(level) ? beta    : mm_eval;
      /* 現状が list であるときの次の手のミニマックス値を求める */
```

```
        evlist[i] = eval = min_max(list,level+1,a,b);
        /* 今までより良い値が見つかれば mm_eval を更新 */
        if ( ( is_leader(level) && eval > mm_eval) ||
             (!is_leader(level) && eval < mm_eval) ) {
          mm_eval = eval;
        }
        pop_one(list);      /* i番の点に置いた仮定を外す */
        if ( ( is_leader(level) && eval > beta ) ||
             (!is_leader(level) && eval < alpha) ) {
          break; /* アルファベータカットで探索を打ち切る */
        }
      }
    }
    return( mm_eval );
}
```

この関数は，現在の盤面の状況を list，ゲーム木の現在の探索レベル（手番）を level，α-β カットに使用する値を alpha と beta という引数で受け取る。

min_max は，ゲームの木を深さ優先探索し，先読みの限界 MaxLevel に到達するかまたは α-β カットが起こると親ノードに戻る。なお，ここではC言語の特性と合わせるため，手番を0から順に数えているので，最初に呼び出す際には min_max(list,0,MIN_VALUE,MAX_VALUE); として呼び出すことになる。

MaxLevel を大きくすれば，最後まで読み切ることもできる。ただし，コンピュータの能力と資源の制約から，最後まで読み切ることができる点の数や手番には限りがある。大規模な問題でシミュレーションを行う際には，局面が1手進むごとに先手や後手が min_max 関数を呼び出して，次の最善手を得るという使い方をする。その場合，例えば先手の役割を人間が行うなど何らかの都合で最善手を選ばなかった場合でも，このコーディングで正しく機能する。

また，α-β カットによる打ち切りを行わなければ，すべての可能な手につい

ての評価値を列挙し，その中で適当な手を選ぶという使い方もできる。

ここで使われている α-β カットの仕組みは以下の通りである。

いま，レベル k の先手番でゲームが終了もしくは読みが打ち切られるとする。レベル $k-1$ で後手がある戦略 s_i を採ったとき，k で先手が採りうる戦略の集合が S_i であったとする。ゲームの木を考えると，ノード s_i の子のノードの集合が S_i である。S_i の各要素を走査し，先手の利得の最大値を β とする。このとき，ノード s_i のゲームの値を β とする。

次に同じレベル $k-1$ で後手が s_j を採ったとき，レベル k で先手が採りうる戦略の集合を S_j とする。S_j の各要素を走査し，先手の利得の最大値を求めて行くが，途中に1つでも先に求めた β を超える値が出てくれば，ただちに S_j の他の要素の探索を打ち切って深さ優先探索を上に戻れば良い。なぜなら，レベル $k-2$ は後手番であるから，s_i よりも s_j の方が先手を有利にすることがわかれば，後手が最善手を採る限り戦略 s_j が選ばれることはないからである。これが β カットである。同様に，最小の値より小さい値で探索を打ち切る場合が α カットである。

α-β カットの効果を充分に引き出すには，各レベルで最も有望そうな手から順に探索すると良い。ここでは，前処理によって点の重みを降順に整列して，未使用の点のうち最も重い点から順に探索することにしている。

min_max は関数値としては1つの値しか返せないが，利得の値だけではなく，どの場所でその利得が得られるかという位置情報も返す方が良い。最大値を与える点が1カ所とは限らないので，点番号をリストなどへ保存してそこへのポインタを引数を通じて返すと，複数の点位置情報を呼び出し側に返すことができる。具体的には，min_max 関数の引数に点番号リストへのポインタを表す int ＊plist などの引数を増やしておくことが考えられる。

min_max は再帰呼び出しを行っているが，各レベルにおいて，どの戦略を採択したか，すなわちどの位置に置いたときの利得がどのくらいかを記録する表が必要である。点番号 i に置いた場合の min_max 関数の値を記録しているの

が配列 evlist である。これは auto 変数として確保しているため，1回の呼び出しごとに N_OF_VERTEX に比例した量のスタックを消費してしまう。ここではプログラムが煩雑になるのを避けるために配列を使用したが，スタックの消費を減らすためには，配置可能な場所および実際に利得を求めた場所に限って，前節の find_ntersection と同様に動的にメモリを確保して記録していく方が効率的である。

6 ま と め

多段階競合施設配置問題を理論的に解くことは困難であるが，とにかく解くことが現実に必要であるような場合には，コンピュータ・シミュレーションが有効と言える。ここでは，GA などを用いた近似解ではなく，厳密解を得るためのコンピュータプログラムの実装法を考察した。本論文の第3節で示した平面走査法と領域探索の手法は，平面上の競合配置問題において汎用的に使用できるものである。また，第5節で示した minimax 探索における α-β カットの手法についても，ゼロサムゲーム全般に利用できるものである。一般に実装にはさまざまな困難がつきまとうため，これらの適用範囲の広い技法について，実際に動作するプログラムを提示することは，理論的に困難な問題をシミュレーションにより解こうとする場合に礎になると思われる。

参 考 文 献

[1] H. Hotelling, "Stability in Competition", The Economic Journal Vol. 30, pp. 41-57, 1929.

[2] S. L. Hakimi, "On Locating New Facilities in a Competitive Environment", European Journal of Operational Research Vol. 12, pp. 29-35, 1983.

[3] Z. Drezner, "Competitive Location Strategies for Two Facilities", Regional Science and Urban Economics Vol. 12, pp. 485-493, 1982.

[4] Hiroshi IMAI, "Finding Connected Components of An Intersection Graph of Squares in The Euclidean Plane", Information Processing Letters, Vol. 15, pp. 125-128, 1982

[5] 塩出省吾，"競合状態下の配置問題"，第4回 RAMP シンポジウム論文集，pp.

105-114, 1992.
[6] 大角盛広，塩出省吾，石井博昭，寺岡義伸，“施設の競合配置問題”，日本生産管理学会論文誌 Vo. 4, No. 2, pp. 153-160, 1997.
[7] 大角盛広，塩出省吾，“ファジィ限界距離を考慮した競合施設配置問題”，日本経営システム学会二十周年記念論文誌 pp. 265-283, 2001.
[8] 大角盛広，塩出省吾，石井博昭，寺岡義伸，“A Competitive Facility Location Problem with Establishing Cost”, Mathematica Japonica Vol. 48, No. 1, pp. 19-26, 1998.
[9] 杉原厚吉，“計算幾何工学”，培風館，1997.