ユークリッドの書斎

和田 英一（IIJ 技術研究所）
wada@u-tokyo.ac.jp

■ e- 幾何学

今回は 2000 年 11 月，つくばでのアジア地区予選の問題 H, Ether Geometry（エーテル幾何学）を話題とする（問題文は http://icpc.score.is.tsukuba.ac.jp/ 参照）．この問題も例によってどうでもよい駄文から始まる．「アレキサンドリアに住む Euclid は幾何学に王道はないと確信しつつも，新聞記事を読みテレビ番組を視，その上，若き学徒に洗脳されると，e- 幾何学あるいはインターネット幾何学という学問分野が創設できるのではないかと思い始めた．e- 幾何学になると，彼は Pythagoras を始め多くの友人と幾何学上の問題をいつでも議論することができ，また彼の比類なき古典，幾何学原論の新版をウェブに公開できるに違いない．彼はすぐさまインターネットを始めようと決心した．彼はまず特異な形状の書斎で，ハブから端末までイーサーケーブルの敷設を始めた．ケーブルは最短距離をとるようにしたい．」まだ無線 LAN が登場する前の話らしい．

問題には例として下のような図がついている（**図 -1**）．A がハブ，B が端末．破線がケーブルを示す．Euclid に登場してもらったのは，柱，壁，ケーブルなど，平面幾何学的に抽象化したいからである．

図には

柱の位置：(0 0) (2 0) (2 2) (4 2) (4 0) (6 0) (6 2) (8 2) (8 0) (10 0)
(10 6) (8 6) (8 4) (6 4) (6 6) (4 6) (4 4) (2 4) (2 6) (0 6)
ハブ：(1 1)，端末：(9 5)

と指示があるが，コンテストのデータは柱の数 n, 以下の行に 1 番からの柱の座標，x_1, y_1, …, x_n, y_n, さらに改行して A，B の座標 a_x, a_y, b_x, b_y と，かっこなしに与えられる．

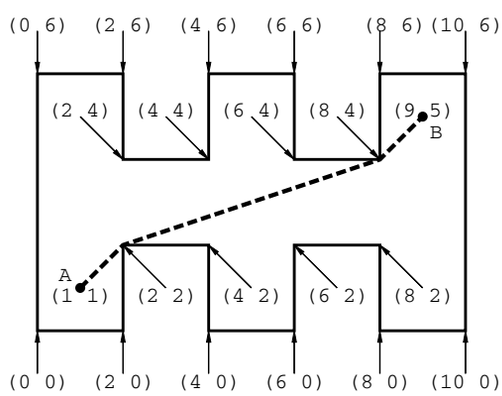

図-1

座標はすべて整数値で，壁は x 軸か y 軸に平行；書斎の内側を左手に見る順に座標が与えられている柱のところで，必ず右か左に直角に曲がる．

ハブから端末まで，両端を含めケーブルの曲がる位置の座標を答える．この例では

（1 1）（2 2）（8 4）（9 5）

と答えればよい（コンテストではかっこなしで答える）．

またこういう例も示してある（**図 -2, 3, 4, 5**）．図 -5 は書斎というより廊下であろう．図 -3 は，柱に接しただけで，ケーブルは曲がらないことを示している．

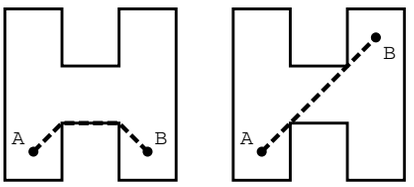

図-2

図-3

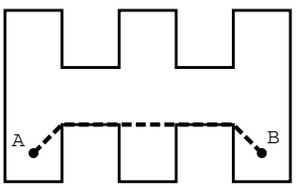

図-4

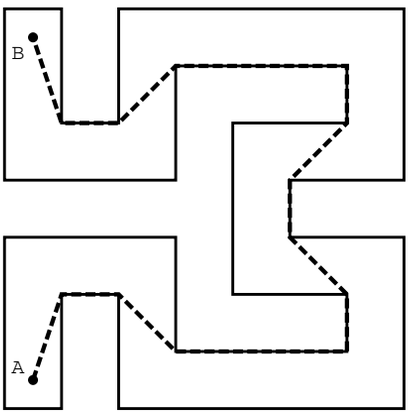

図-5

■ Dijkstra の最短経路探索法

図 -1 を見て，ケーブルが折れ曲がる可能性のある（凸の）角に C から順に名前をつける（**図 -6**）．それぞれの点の間の距離を表にしてみると**表 -1** のようになる（添字は後述する角の区別）．凹の角はケーブルに接しないことに注意しよう．

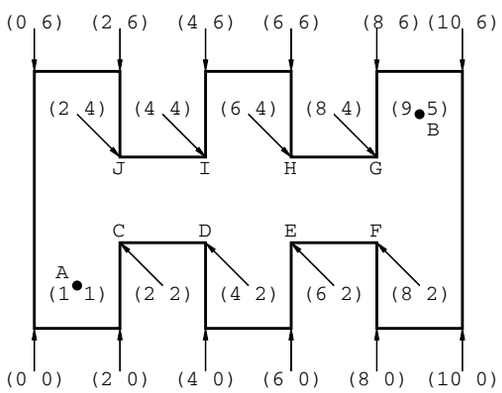

図-6

	A_{-1}	B_{-1}	C_3	D_0	E_3	F_0	G_1	H_2	I_1	J_2
A_{-1}	0		$\sqrt{2}$						$\sqrt{18}$	
B_{-1}		0			$\sqrt{18}$		$\sqrt{2}$			
C_3	$\sqrt{2}$		0	2		6	$\sqrt{40}$		$\sqrt{8}$	
D_0			2	0						$\sqrt{8}$
E_3		$\sqrt{18}$			0	2	$\sqrt{8}$			
F_0			6		2	0		$\sqrt{8}$		$\sqrt{40}$
G_1		$\sqrt{2}$	$\sqrt{40}$		$\sqrt{8}$		0	2		6
H_2						$\sqrt{8}$	2	0		
I_1	$\sqrt{18}$		$\sqrt{8}$						0	2
J_2				$\sqrt{8}$		$\sqrt{40}$	6		2	0

表 -1

この表-1をLispのデータにしたものを次に示す．−1は接続がないことである．プログラム言語には今回もSchemeを使う．

```
(define tab (list
  (list 0 -1 (sqrt 2) -1 -1 -1 -1 -1 (sqrt 18) -1)      ;a
  (list -1 0 -1 -1 (sqrt 18) -1 (sqrt 2) -1 -1 -1)      ;b
  (list (sqrt 2) -1 0 2 -1 6 (sqrt 40) -1 (sqrt 8) -1)  ;c
  (list -1 -1 2 0 -1 -1 -1 -1 -1 (sqrt 8))              ;d
  (list -1 (sqrt 18) -1 -1 0 2 (sqrt 18) -1 -1 -1)      ;e
  (list -1 -1 6 -1 2 0 -1 (sqrt 8) -1 (sqrt 40))        ;f
  (list -1 (sqrt 2) (sqrt 40) -1 (sqrt 8) -1 0 2 -1 6)  ;g
  (list -1 -1 -1 -1 -1 (sqrt 8) 2 0 -1 -1)              ;h
  (list (sqrt 18) -1 (sqrt 8) -1 -1 -1 -1 -1 0 2)       ;i
  (list -1 -1 -1 (sqrt 8) -1 (sqrt 40) 6 -1 2 0)))      ;j
```

Dijkstraの最短経路探索は次のように書くことができる[1]．今回はDijkstraの解法の解説ではないので，このプログラムは無駄な計算は容認し素直に書いてある．上のような`tab`をもらってくると，節点番号0から各節点への最短距離がリスト`l`に，またその場合，その節点へはどの節点から来たかがリスト`v`に得られる．

```
(define (dijk tab)
  (define (make-list i n v)     ; リスト((i v) (i+1 v) ... (n-1 v))を作る
    (if (>= i n) '()
        (cons (list i v) (make-list (+ i 1) n v))))
  (define (<< a b)
    (and (< a b) (> (/ (- b a) b) 0.000001)))
  (let ((l (cons (list 0 0) (make-list 1 (length tab) 999999)))
        (v (make-list 0 (length tab) -1)))
    (do ((i 0 (+ i 1))) ((= i (length tab)))
      (let ((j (car (list-ref l i))) (m (cadr (list-ref l i))))
        (let ((d (list-ref tab j)))
          (do ((k 0 (+ k 1))) ((= k (length tab)))
            (if (> (list-ref d k) 0)
              (let ((s (+ m (list-ref d k))))
                (if (<< s (cadr (assoc k l)))
                    (begin (set-cdr! (assoc k l) (list s))
                           (set-cdr! (assoc k v) (list j)))))))))
      (set! l (sort l (lambda (x y) (<< (cadr x) (cadr y))))))
    (list l v)))
```

上の`tab`でこのプログラムを走らせると

```
(((0 0) (2 1.4142135623730951) (3 3.414213562373095) (8 4.242640687119285)
  (9 6.242640687119285) (5 7.414213562373095) (6 7.738768882709854)
  (1 9.15298244508295) (4 9.414213562373096) (7 9.738768882709854))
 ((0 -1) (1 6) (2 0) (3 2) (4 5) (5 2) (6 2) (7 6) (8 0) (9 8)))
```

が得られる．2つのリストのうち，前のリスト`l`は節点0からの最短距離，後のリスト`v`は節点0からの最短距離を得るには，この節点にはどの節点から来たかを示す．Aは節点番号0，Bは節点番号1だから，Bにはどこから来たか見ると`(1 6)`により6から来たことが分かる；また`(6 2)`により6には2から来たことが分かる；`(2 0)`により2には0，すなわちAから来たことが分かる．つまり0(A)→2→6→1(B)がABの最短距離になる．その距離は上のリストに`(1 9.15298244508295)`とあるから，Aからは9.なにがしの距離で来たことが分かる．これは$\sqrt{2}$（ACの距離）+$\sqrt{40}$（CGの距離）+$\sqrt{2}$（GBの距離）である．

`(+ (sqrt 2) (sqrt 40) (sqrt 2))`→9.15298244508295

つまり`tab`のような距離行列が作れれば，問題は解けたことになる．

■距離行列の作り方—角の向きだけを考慮する

図 -5 に対して表 -1 のような距離行列を作ってみる．まず壁の角を向きにより区別しよう．北東，南東，南西，北西をそれぞれ 0, 1, 2, 3 で区別する．図 -5 の各角に名前と区別をつけたのが**図 -7** である．

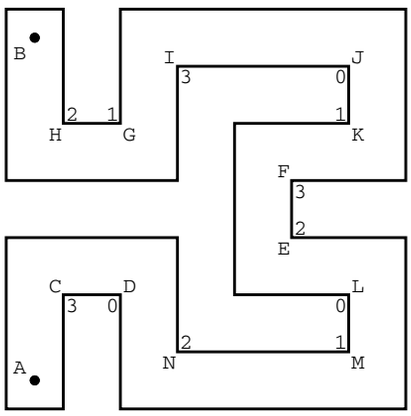

図-7

ある点 X から出発したケーブルが柱の角で曲がるのはいつかというと X を原点として第 1 象限にある相手はそれが 3 の角のとき（そこで右に曲がる）か 1 の角のとき（そこで左に曲がる）である．

他の場合も同様に考え，それらをまとめると

第 1 象限 : 1 の角（左折），3 の角（右折）
第 2 象限 : 0 の角（左折），2 の角（右折）
第 3 象限 : 3 の角（左折），1 の角（右折）
第 4 象限 : 2 の角（左折），0 の角（右折）
が相手になる（**図 -8**）．破線はケーブルのつもり．

同様にして x+ の方向に探すときは角 1（左折），角 0（右折）だけが候補である．これも他の場合とともにまとめると（**図 -9**），

x+: 1 の角（左折），0 の角（右折）
x−: 3 の角（左折），2 の角（右折）
y+: 0 の角（左折），3 の角（右折）
y−: 2 の角（左折），1 の角（右折）

一方，A や B のような孤立点を除き，他の角はその実体が 1 つの象限を占めているので，ケーブルが折れ曲がる方向にはそれぞれ制約がある（**図 -10**）．そこで

角 0 なら相手を探すのは i) 第 4 象限と ii)−y の方向と iii) 第 2 象限と iv)−x の方向
角 1 なら相手を探すのは i) 第 1 象限と ii)+y の方向と iii) 第 3 象限と iv)−x の方向
角 2 なら相手を探すのは i) 第 2 象限と ii)+y の方向と iii) 第 4 象限と iv)+x の方向
角 3 なら相手を探すのは i) 第 3 象限と ii)−y の方向と iii) 第 1 象限と iv)+x の方向
となる．

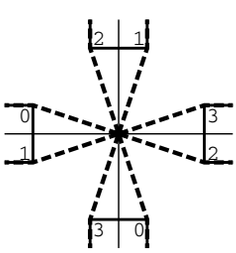

図-8

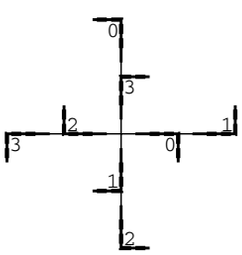

図-9

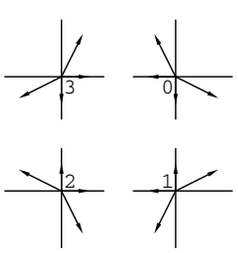

図-10

次のような考慮すべき座標のデータがあるとする（入力データから作れる）．各要素は角の名前（小文字で表示してある），x 座標，y 座標，角の区別（A, B は −1 とする）．角の名前は本質的ではないが，虫取りには有効である．

```
(define corners '(
 (a 1 1 -1) (b 1 13 -1) (c 2 4 3) (d 4 4 0) (e 10 6 2) (f 10 8 3) (g 4 10 1)
 (h 2 10 2) (i 6 12 3) (j 12 12 0) (k 12 10 1) (l 12 4 0) (m 12 2 1) (n 6 2 2))
```

この corners に対して次のプログラムを走らせる．q1 は第 1 象限の，x+ は x+ 方向の候補を探す．他も同様．

```
(define (connectablep here there)
  (let ((x0 (cadr here)) (y0 (caddr here)) (t0 (cadddr here))
        (x1 (cadr there)) (y1 (caddr there)) (t1 (cadddr there)))
    (define (q1) (and (> x1 x0) (> y1 y0) (or (= t1 -1) (odd? t1))))
    (define (q2) (and (< x1 x0) (> y1 y0) (or (= t1 -1) (even? t1))))
    (define (q3) (and (< x1 x0) (< y1 y0) (or (= t1 -1) (odd? t1))))
    (define (q4) (and (> x1 x0) (< y1 y0) (or (= t1 -1) (even? t1))))
    (define (x-) (and (< x1 x0) (= y1 y0) (or (= t1 -1) (>= t1 2))))
    (define (x+) (and (> x1 x0) (= y1 y0) (or (= t1 -1) (<= t1 1))))
    (define (y-) (and (= x1 x0) (< y1 y0) (or (= t1 -1) (= t1 1) (= t1 2))))
    (define (y+) (and (= x1 x0) (> y1 y0) (or (= t1 -1) (= t1 3) (= t1 0))))
    (or (and (= t0 0) (or (q4) (y-) (q2) (x-))) ; 図 -10 参照
        (and (= t0 1) (or (q1) (y+) (q3) (x-)))
        (and (= t0 2) (or (q2) (y+) (q4) (x+)))
        (and (= t0 3) (or (q3) (y-) (q1) (x+)))
        (and (= t0 -1) (or (q1) (q2) (q3) (q4) (x+) (x-) (y+) (y-))))))

(define (filter p l)    ;l の要素で述語 p に対して真なもののリストを返す
  (cond ((null? l) '())
        ((p (car l)) (cons (car l) (filter p (cdr l))))
        (else (filter p (cdr l)))))

(define (make-connectable corners)
  (map (lambda (here)
         (cons (car here)
               (map (lambda (c)
                      (car c))
                    (filter (lambda (there) (connectablep here there))
                            corners))))
       corners))
```

make-connectable は corners の各要素に対し，(lambda (here) ...) の関数を作用させる．here には corners の要素（たとえば (a 1 1 -1)）が次々と与えられる．filter が corners の各要素で (lambda (there) ...) の述語に対し真になったもののリストを返すと，内側の map はそれに (lambda (c) ...) の関数を作用させ，connectablep に合格した角の名前（(car c) で得られる）のリストを作る．それに出発点の角の名前を (cons (car here) ...) で前におく．

したがって (set! connectable (make-connectable corners)) で次の結果が connectable に得られる．各要素リストの先頭は出発点でそこから壁を無視してケーブルが引ける角のリストが続く．

```
((a b c f g i k m) (b a d e h j l n) (c a d f g i k l) (d b c h n)
 (e b f h l) (f a c e k) (g a c h i) (h b d e g k l n) (i a c g j n)
 (j b i k m) (k a c f h j) (l b c e h m) (m a j l n) (n b d h i m))
```

A から壁を無視してケーブルが引けるのは B, C, F, G, I, K, M であるのように読む．

■距離行列の作り方—壁を考慮する

次は壁で遮られる角を除去する番である．壁には図-11のように壁が北向き，東向き，南向き，西向きをそれぞれ0, 1, 2, 3で区別する．wall0（入力データから作れる）は北向き壁で各要素は壁のy座標，両端のx座標である．

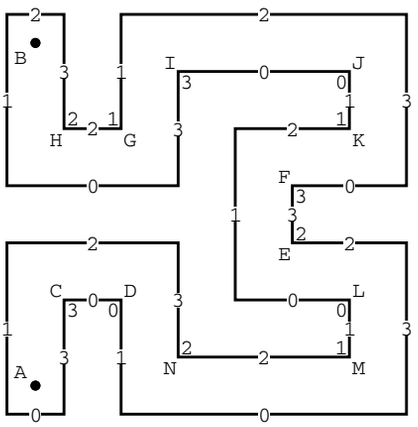

図-11

```
(define wall0 '(
 (0 0 2) (4 2 4) (0 4 14) (8 10 14) (8 0 6) (12 6 12) (4 8 12)))
(define wall1 '(
 (4 4 0) (4 14 10) (0 14 8) (12 12 10) (8 10 4) (12 4 2) (0 6 0)))
(define wall2 '(
 (6 14 10) (14 14 4) (10 4 2) (14 2 0) (10 12 8) (2 12 6) (6 6 0)))
(define wall3 '(
 (2 0 4) (14 0 6) (10 6 8) (14 8 14) (2 10 14) (6 8 12) (6 2 6)))
```

壁を定義し，今のconnectableに対して以下のプログラムを走らせる．

```
(define (visiblep here there)
  (let* ((x0 (cadr here)) (y0 (caddr here)) (t0 (cadddr here))
         (x1 (cadr there)) (y1 (caddr there)) (t1 (cadddr there))
         (d (* (- x1 x0) (- y1 y0))))
    (define (nothing p l)       ;lにpなるものが皆無のとき真を返す
      (cond ((null? l))
            ((p (car l)) '())
            (else (nothing p (cdr l)))))
    (define (side x2 y2)        ;三角形の分割にあったのと同様
      (- (+ (* x0 y1) (* x1 y2) (* x2 y0))
         (+ (* x2 y1) (* x0 y2) (* x1 y0))))
    (define (corner? x y)
      (nothing
        (lambda (c) (and (= (cadr c) x) (= (caddr c) y)
          (let ((t1 (cadddr c)))        ;corner上の点なら
            (cond ((> d 0) (odd? t1))   ;視線の向きと角の関係を見る
                  ((< d 0) (even? t1))
                  (else #t))))) corners))
    (define (xv)
      (define (crossx x y+ y-)
        (let ((s0 (side x y-)) (s1 (side x y+)))
            (cond ((= s0 0) (corner? x y-))
                  ((= s1 0) (corner? x y+))
                  (else (< (* s0 s1) 0)))))
      (cond ((< x0 x1)  ;(x0 < x < x1) にある3の壁を調べる
```

```
                (nothing
                  (lambda (w)
                    (let ((x (car w)) (y- (cadr w)) (y+ (caddr w)))
                      (and (< x0 x) (< x x1) (crossx x y+ y-)))) wall3))
               ((> x0 x1)  ;(x0 > x > x1) にある 1 の壁を調べる
                (nothing
                  (lambda (w)
                    (let ((x (car w)) (y- (cadr w)) (y+ (caddr w)))
                      (and (> x0 x) (> x x1) (crossx x y+ y-)))) wall1))
               (else #t)))
      (define (yv)
        (define (crossy y x+ x-)
          (let ((s0 (side x- y)) (s1 (side x+ y)))
              (cond ((= s0 0) (corner? x- y))
                    ((= s1 0) (corner? x+ y))
                    (else (< (* s0 s1) 0)))))
        (cond ((< y0 y1)  ;(y0 < y < y1) にある 2 の壁を調べる
               (nothing
                 (lambda (w)
                   (let ((y (car w)) (x- (cadr w)) (x+ (caddr w)))
                     (and (< y0 y) (< y y1) (crossy y x+ x-)))) wall2))
              ((> y0 y1)  ;(y0 > y > y1) にある 0 の壁を調べる
               (nothing
                 (lambda (w)
                   (let ((y (car w)) (x- (cadr w)) (x+ (caddr w)))
                     (and (> y0 y) (> y y1) (crossy y x+ x-)))) wall0))
              (else #t)))
    (and (xv) (yv))))

(define (make-visible connectable)
  (map (lambda (x)
         (let ((here (assoc (car x) corners)))
           (cons (car x)
                 (filter (lambda (y)
                           (let ((there (assoc y corners)))
                             (visiblep here there)))
                         (cdr x)))))
       connectable))
```

make-visible は connectable の要素（たとえば (a b c f g i k m)）を順に x にとり，出発点の名前 (car x)（たとえば a）から (assoc (car x) corners) で角のデータ（たとえば (a 1 1 -1)）を here に得る．次に (cdr x)（たとえば (b c f g i k m)）の各要素を y にとり，(assoc y corners) で y の角のデータを there にとって here から there が visiblep のものだけを filter して返す．それに (cons (car x) ...) で出発点の名前を前におく．

したがって (set! visible (make-visible connectable)) で visible に得られた結果は以下のとおり．

```
((a c) (b h) (c a d) (d c n) (e f l) (f e k) (g h i)
 (h b g) (i g j) (j i k) (k f j) (l e m) (m l n) (n d m))
```

A から見える（行ける）のは C, B から見える（行ける）のは H のように読む．

■距離行列の作り方—まとめる

最後は visible に対し次のプログラムを走らせる．

```
(define (make-tab visible)
  (define (dist x y)
```

```
  (let* ((here (assoc x corners)) (there (assoc y corners))
         (x0 (cadr here)) (y0 (caddr here))
         (x1 (cadr there)) (y1 (caddr there)))
  (+ (sqrt (+ (* (- x1 x0) (- x1 x0)) (* (- y1 y0) (- y1 y0))))
     0.00001)))
(map (lambda (x)
       (map (lambda (y)
              (cond ((eq? (car y) (car x)) 0)
                    ((member (car y) x) (dist (car x) (car y)))
                    (else -1)))
            corners))
visible))
```

こうして (set! tab (make-tab visible)) で得られた図-5 のための最終的な距離行列は

```
((0 -1 3.1622876601683796 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1)            ;a
 (-1 0 -1 -1 -1 -1 -1 3.1622876601683796 -1 -1 -1 -1 -1 -1)         ;b
 (3.1622876601683796 -1 0 2.00001 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1)   ;c
 (-1 -1 2.00001 0 -1 -1 -1 -1 -1 -1 -1 -1 2.8284371247461904)      ;d
 (-1 -1 -1 -1 0 2.00001 -1 -1 -1 -1 -1 2.8284371247461904 -1 -1)   ;e
 (-1 -1 -1 -1 2.00001 0 -1 -1 -1 -1 2.8284371247461904 -1 -1 -1)   ;f
 (-1 -1 -1 -1 -1 -1 0 2.00001 2.8284371247461904 -1 -1 -1 -1 -1)   ;g
 (-1 3.1622876601683796 -1 -1 -1 -1 2.00001 0 -1 -1 -1 -1 -1 -1)   ;h
 (-1 -1 -1 -1 -1 -1 2.8284371247461904 -1 0 6.00001 -1 -1 -1 -1)   ;i
 (-1 -1 -1 -1 -1 -1 -1 -1 6.00001 0 2.00001 -1 -1 -1)              ;j
 (-1 -1 -1 -1 -1 2.8284371247461904 -1 -1 -1 2.00001 0 -1 -1 -1)   ;k
 (-1 -1 -1 -1 2.8284371247461904 -1 -1 -1 -1 -1 -1 0 2.00001 -1)   ;l
 (-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2.00001 0 6.00001)              ;m
 (-1 -1 -1 2.8284371247461904 -1 -1 -1 -1 -1 -1 -1 -1 6.00001 0))  ;n
```

となる（各行の最後のコメントは後から人手で追加）．このプログラムでは connectable や visible に角の名前だけのリストを返すようにしたため，次のプログラムで assoc して corners の情報を取り戻さなければならず，少し面倒になっている．

参考文献

1）野崎昭弘：アルゴリズムと計算量，計算機科学／ソフトウェア技術講座，共立出版 (1987).

（平成 15 年 1 月 14 日受付）