

The Smallest Pair of Noncrossing Paths in a Rectilinear Polygon

**A dissertation submitted in partial fulfillment
of the requirements of M.Tech.(Computer Science)
degree of Indian Statistical Institute, Kolkata
by**

Gautam Kumar Das

under the supervision of

**Dr. S. Sur-Koley and Dr. S. C Nandy
Advance Computing & Microelectronics Unit**

**Indian Statistical Institute
203, Barrackpor Trunk Road
Kolkata-700 108.**

19th July 2002

Indian Statistical Institute


203, Barrackpore Trunk Road,

Kolkata-700 108.

Certificate of Approval

This is to certify that this thesis titled "**The Smallest Pair of Noncrossing Paths in a Rectilinear Polygon**" submitted by **Gautam Kumar Das** towards partial fulfillment of requirements for the degree of M. Tech in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under my supervision.

Dr. S. Sur-Koley,
Advanced Computing and
Microelectronic Unit,
Indian Statistical Institute,
Kolkata-700 108.


Dr. S. C. Nandy,
Advanced Computing and
Microelectronic Unit,
Indian Statistical Institute,
Kolkata-700 108.

Acknowledgements

I take pleasure in thanking Dr. S. Sur-Koley and Dr. S. C. Nandy for their friendly guidance throughout the dissertation period. Their pleasant and encouraging words have always kept my spirits up.

I take the opportunity to thank my classmates, friends and my family for their encouragement to finish this work.

Gautam Kumar Das

Abstract

Smallest rectilinear paths are rectilinear paths with a minimum length and a minimum number of bends simultaneously. Given two pairs of terminals within a rectilinear polygon, we describe an algorithm to find a pair of noncrossing rectilinear paths within the polygon such that the total number of lengths and total number of bends are both minimized. But such a smallest pair may not exist for some problem instance. In that case, the algorithm presented will find, among all non crossing paths with a minimum total length, a pair whose total length is the shortest, or find, among all noncrossing paths with a minimum total length, a pair whose total number of bends is minimized. We describe a simple linear time and space algorithm based on the fact that there are only a limited number of configurations of such a solution pair.

Contents

1	Introduction	2
2	Problem Formulation and some Definitions	4
3	Overview of Algorithm	9
4	Implementation Details	11
4.1	Data Structures	11
4.2	Shortest Path Algorithm	11
4.3	Detection of crossing of two given paths :	18
4.4	Detour of the Paths :	20
4.5	Canonical path	21
5	Experimental Results	26
6	Conclusion	29

Chapter 1

Introduction

Given a set of obstacles and two distinguished points in the plane, the problem of finding a collision-free path subject to a certain optimization function is a fundamental problem that arises in many fields, such as motion planning in robotics, wire routing in VLSI and in different logistics of operations research. In this thesis we emphasize its applications to VLSI design and limit ourselves to the rectilinear domain in which the goal path to be computed and the underlying obstacles are all rectilinearly oriented, i.e., the segments are either horizontal or vertical. We provide a survey of results pertaining to routing in VLSI according to different routing environments, optimization criteria and problem solving approach.

In VLSI layout, a basic function unit or circuit module is represented by a rectilinear polygon, whose shape may or may not be (rectilinearly) convex. The *pins* or *terminals* of the same net that lie on different modules need to be interconnected by wires that mostly run either horizontally or vertically. In a single layer model, the total *length* of the wires in the interconnection is often used as an objective function, as it affects cost of wiring and total delay. On the other hand, the *number of bends* on a path affects the resistance and hence the accuracy of expected timing and voltage in chips. Unfortunately, they can not be both optimized simultaneously in general. Therefore a *best path* can be categorized by either minimizing each of these measures individually, giving them different optimizing priorities, or fixing one at a certain bound and minimizing the other. In addition to these optimization factors of rectilinear paths, the routing models and the *types* of obstacles also affect the complexity of the problems.

In VLSI routing, an important problem is finding collision free rectilinear paths[2] and [3] in the presence of rectilinear obstacles. Traditionally, one is interested in finding a *shortest* rectilinear path connecting a given pair of points, or a path with a minimum number of bends, called a *minimum-bend path*. Here we address the problem, taking both factors, the number of bends and length, into account. For instance, among the minimum-bend paths, find one whose total length is minimized or among the minimum length paths, find one whose total number of bends is minimized. Within a rectilinear polygon, McDonald and Peters[5] have shown that there

always exists a path connecting two given points, whose total length and total number of bends are both minimum. This path is referred to as the smallest path. In[5], McDonald and Peters presented a linear time algorithm to find such a smallest path in a rectilinear polygon. In the restricted two-layer routing model, where horizontal and vertical wires must go on separate layers, the number of bends corresponds to the number of vias used([4]). Here a smallest path minimizes both wire length and the number of vias.

Algorithms for finding between two terminals are adopted in practice as a basic routine module for iteratively routing multiple two-terminal nets. But if more than one paths can be found at a time minimizing the total cost of these paths would yield better overall results. Here, we consider the problem of finding two noncrossing paths within a rectilinear polygon with an aim to minimize both total length and total number of bends. This is our an initial attempt of attacking the general $k \geq 2$ pairs of noncrossing paths in a simply connected routing area represented as a rectilinear polygon. Two paths are called noncrossing if they either are disjoint or may overlap, but never cross each other. Finding noncrossing paths has several applications in layout design in VLSI. (In robot motion planning this problem can be mapped to the layout of multiple robots in the same room).

Chapter 2

Problem Formulation and some Definitions

A *rectilinear path* is a path consisting of only vertical and horizontal segments. The *length* of a *rectilinear path* is the sum of the lengths of all its segments. An obstacle is said to be *rectilinear* if all its boundary edges are either horizontal or vertical. The point adjacent to a horizontal segment and a vertical segment is called a *bend*. Let $d(\pi)$ and $b(\pi)$ denote, respectively, the length and number of bends of a path π . The *rectilinear distance* between two points p and q equals $|p.x - q.x| + |p.y - q.y|$ where, $p.x$ and $p.y$ are abscissa and the ordinate of point p , respectively.

From now onwards, a *path* refers to a rectilinear path and all the obstacles are rectilinear unless otherwise specified.

Given a set of disjoint obstacles, a source point s and a destination point t , we define the following problems:

Shortest path (SP) Problem: Find a shortest (collision-free) path from s to t .

Minimum Bend Path (MBP) Problem: Find a path from s to t with minimum number of bends.

Smallest Path (SMALLP) Problem: Find a path from s to t such that it is of minimum length and has a minimum number of bends simultaneously. Such a path is called *smallest path*.

Minimum Bend Shortest path (MBSP) Problem: Find a path with minimum number of bends among all the shortest paths from s to t . Such a path is called *minimum bend shortest path*.

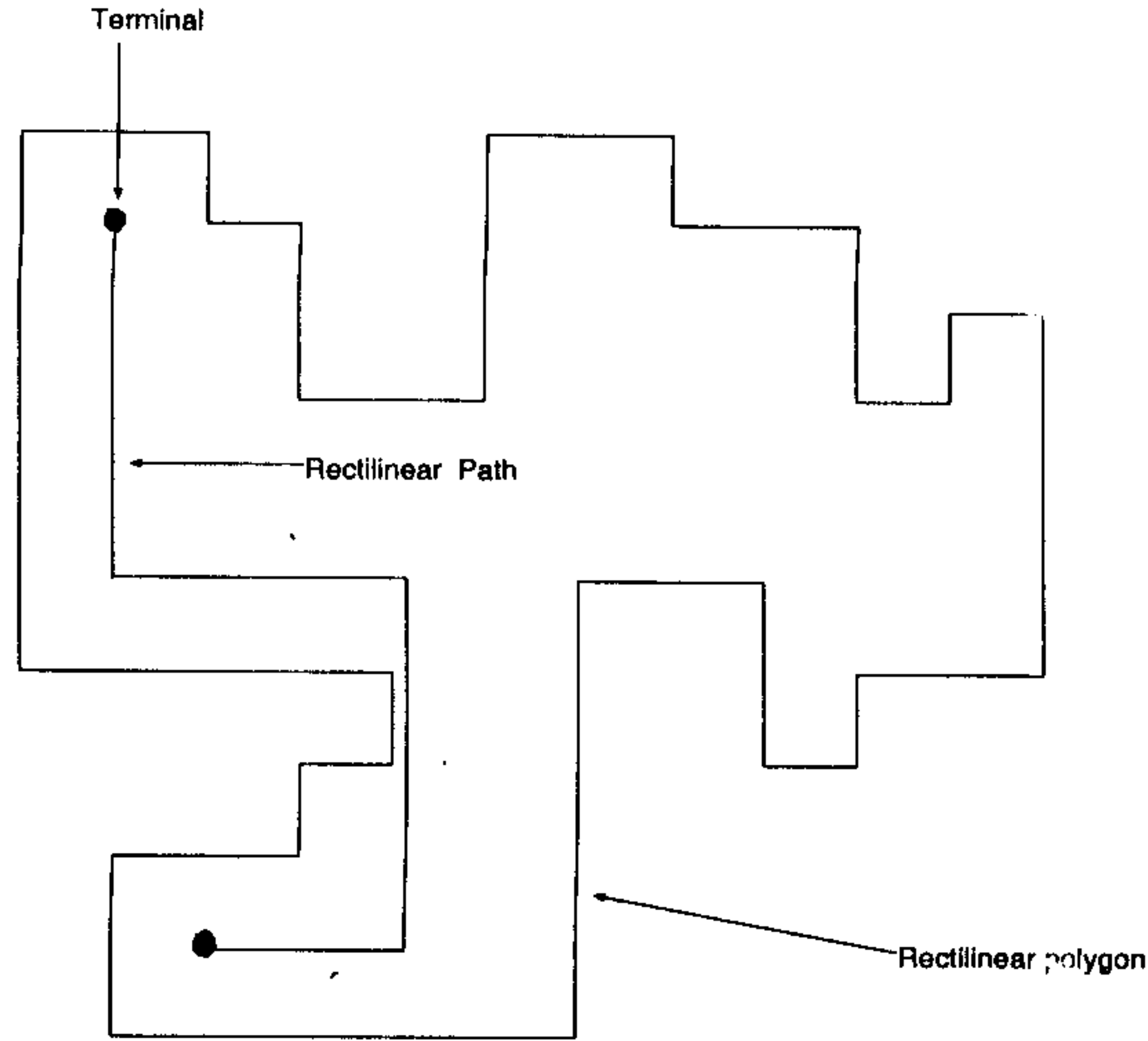


Figure 2.1: A rectilinear path inside an isothetic polygon

Shortest Minimum Bend Path (SMBP) Problem: Find a shortest path among all the minimum bend paths from s to t . Such a path is called *shortest minimum bend path*.

Minimum Cost Path (MCP) Problem: Find a minimum cost path from s to t , where the cost is a non-decreasing function of the length and the number of the bends of a path.

Bounded Bend Shortest path (BBSP) Problem: Find a shortest path among all the paths from s to t with the number of bends no greater than a given bound. Such a path is called *bounded bend shortest path*.

Bounded Length Minimum Bend path (BLMBP) Problem: Find a path with the minimum number of bends among all the paths from s to t with length no greater than a given bound. Such a path is called a *bounded length minimum bend path*.

Here, we consider the following problem : given a simple rectilinear polygon and two pair of terminals within the polygon find a pair of noncrossing paths among the corresponding terminals within the polygon such that the total number of bends and total length are both minimized. If simultaneously both are not possible then compromise one with the other..

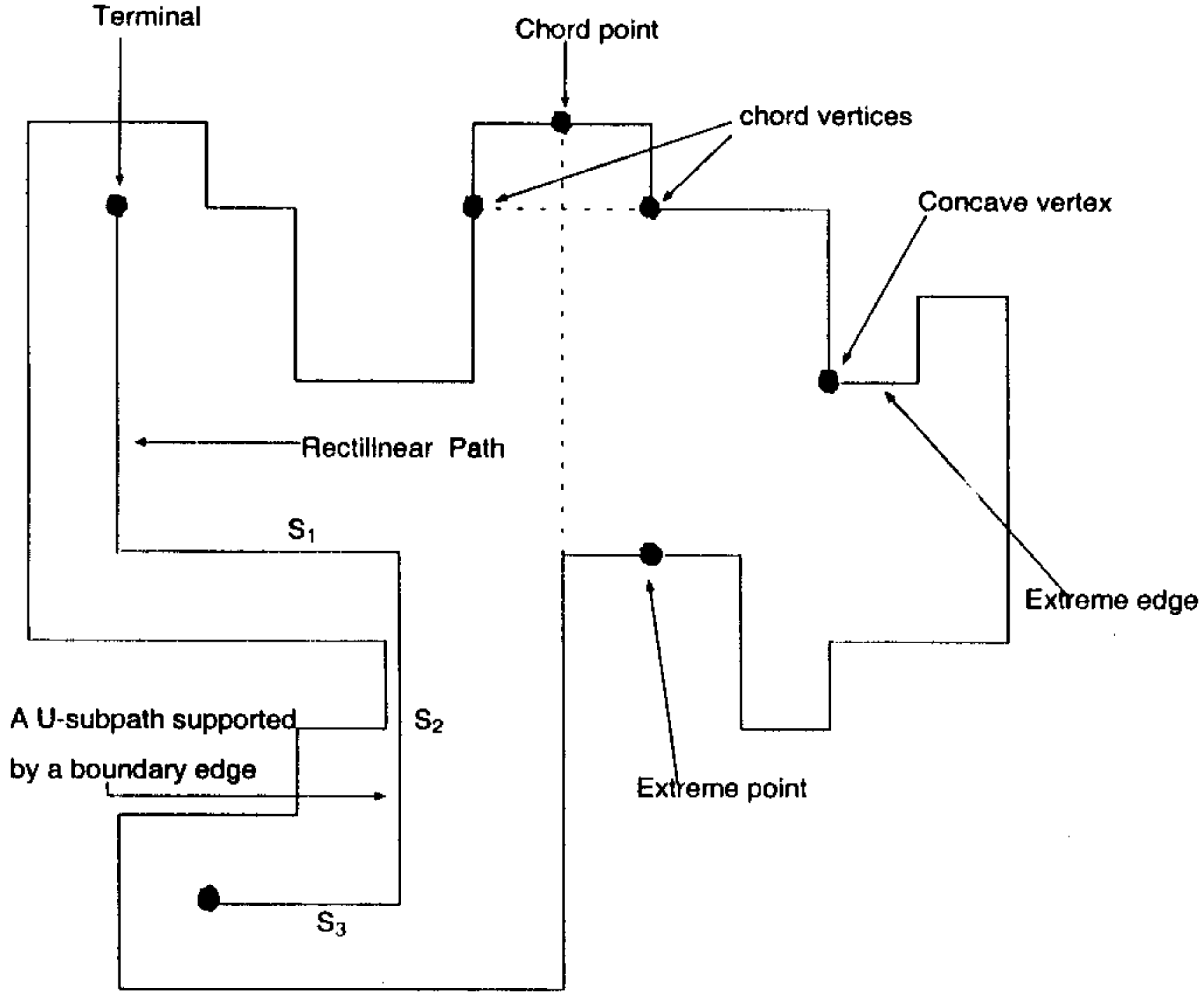


Figure 2.2: Illustration of concave vertex, extreme edge, extreme point, chord point, chord vertex, and U-subpath

Definition 1 A Rectilinear Path connecting two points u and v consists of a sequence of horizontal and vertical segments, represented as $\{s_1 = \overline{p_1, p_2}, s_2 = \overline{p_2, p_3}, \dots, s_k = \overline{p_k, p_{k+1}}\}$, where $\{p_2, p_3, \dots, p_k\}$ are the end points of the segments, and $u = p_1, v = p_{k+1}$.

Definition 2 A Simple Rectilinear Polygon is a rectilinear path whose two endpoints coincide and whose segments do not self-intersect, except at the end point between consecutive segments.

Definition 3 Two paths, π_1 and π_2 , are said to cross each other if there is an overlap between two segments $s_1, s_2 \in \pi_1$ and $s_3, s_4 \in \pi_2$ can be cyclically ordered as (s_1, s_2, s_3, s_4) . They are said to be noncrossing otherwise. If the total number of incident segment associated with the overlap is fewer than four, they are noncrossing.

Definition 4 A terminal is an end point of a path. A chord is a horizontal or a vertical line segment from a point on δP to another point on δP without crossing any boundary segment

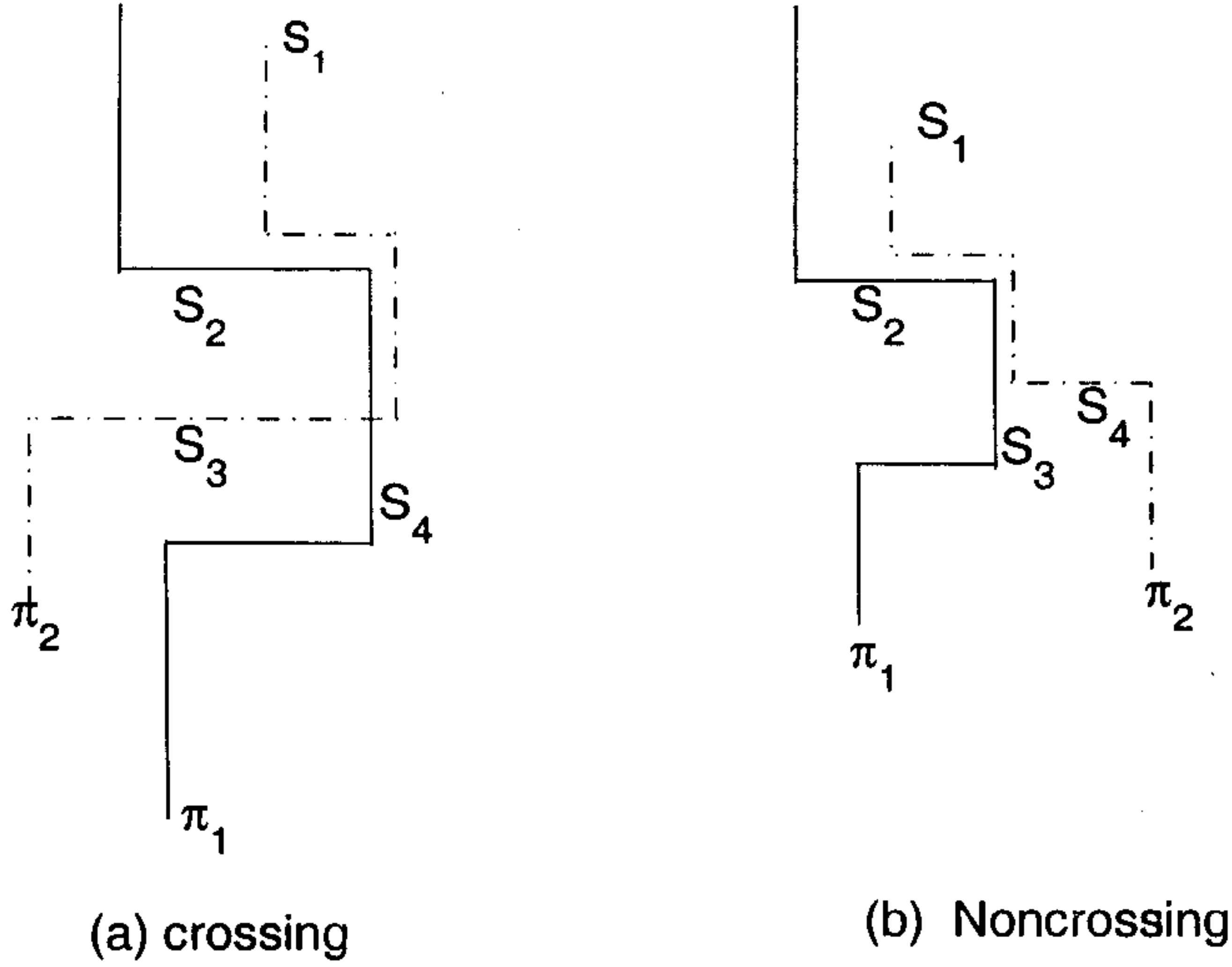


Figure 2.3: Example of crossing and non-crossing paths

of P . (where P is the name of the polygon and δP is the boundary of the polygon). An edge-to-vertex visible pair refers to an edge and a vertex that can be connected by a chord. For an edge-to-vertex visible pair, the endpoint of the chord on the edge is called chord point corresponding to the vertex. A vertex-vertex visible pair refers to two vertices that can be connected by a chord. The corresponding vertices are called chord vertices.

Definition 5 A vertex p_i is concave if the interior angle $\angle p_{i-1}p_ip_{i+1}$ has a measure of 270° (we assume $p_0 = p_n$ and $p_{n+1} = p_1$.) An edge $\overline{p_ip_{i+1}}$ is an extreme edge if both p_i and p_{i+1} are concave. An extreme point refers to the middle point of an extreme edge.

Definition 6 A U-subpath of a path is a three-segment subpath, namely s_1, s_2 , and s_3 ; the segments s_1 and s_3 lie on the same side of the line L_{s_2} , containing the segments s_2 . In a U-subpath, if segment s_2 coincides with an extreme edge or passes through a terminal, then we say that the U-subpath is supported by the extreme edge or the terminal, respectively.

Definition 7 An XY path from one endpoint p (starting point) to another end point q (ending point) is a path that only goes in $+X$ or $+Y$ directions. $X(-Y)$, $(-X)Y$ and $(-X)(-Y)$ paths are defined symmetrically. They are all referred to as staircase paths.

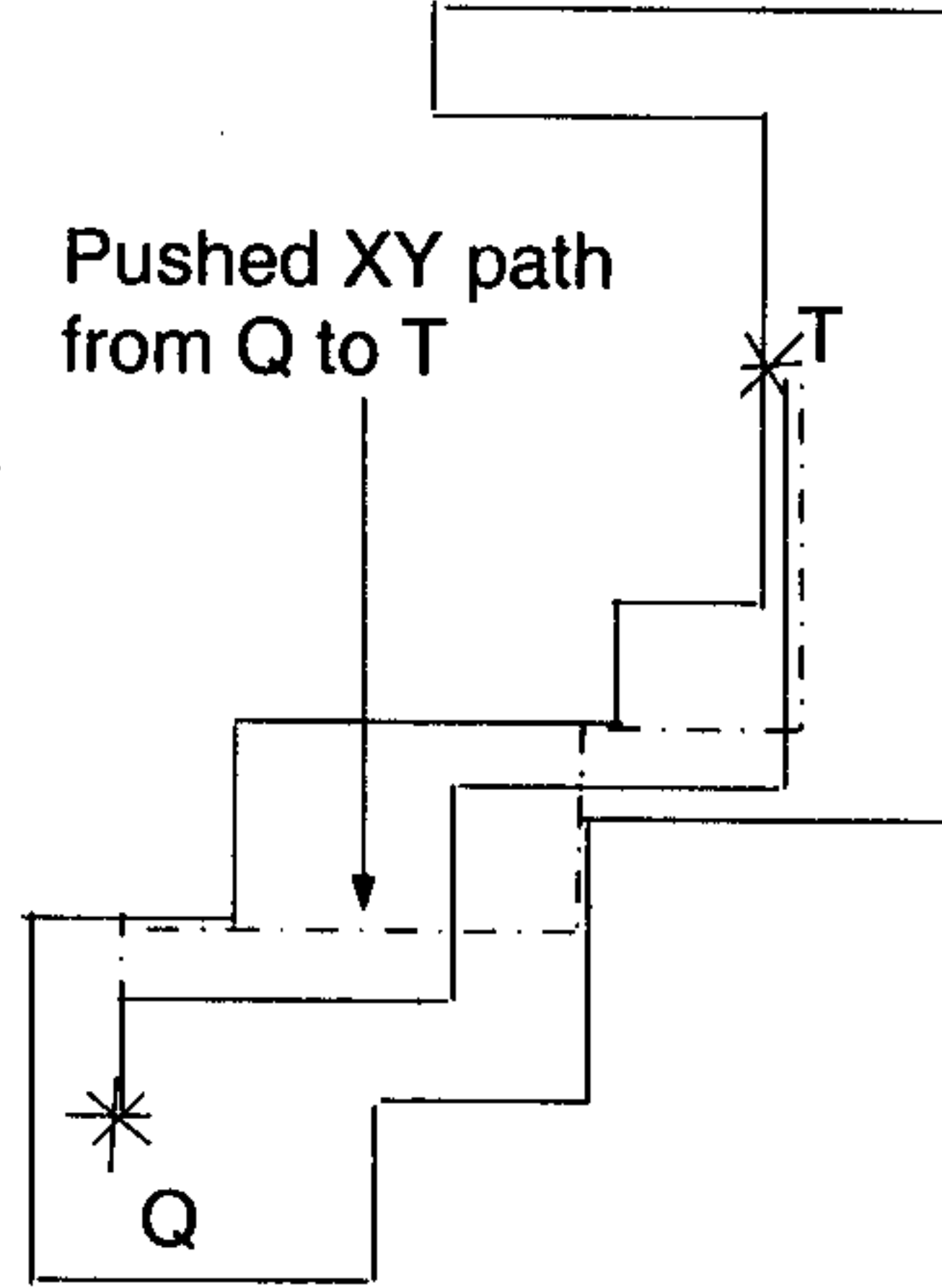


Figure 2.4: Example of *pushed-path*: $\pi_{QT} = \text{pushed } XY\text{-path}$, and $\pi_{TQ} = \text{pushed } (-X)(-Y)\text{-path}$

Note that an XY path from p to q is a $(-X)(-Y)$ path from q to p and, similarly, an $X(-Y)$ path from p to q is a $(-X)Y$ path from q to p .

Definition 8 Given a path π between two terminals p and q , the *extreme sequence* of π , denoted as $EX(\pi)$, is the sequence of points that starts with p , followed by the sequence of the extreme points supporting all the U -subpaths on π , and ends with q .

Definition 9 A path π_{pq} is a *pushed XY path* from p to q if the following three conditions hold

- (1) It is an XY path.
- (2) For every horizontal segment $\overline{h_1 h_2}$ on path π_{pq} , $h_1.x < h_2.x$.
- (3) For every vertical segment $\overline{v_1 v_2}$ on path π_{pq} , $v_1.y < v_2.y$.

Definition 10 A *canonical path* from p to q is a path composed of a sequence of pushed staircase subpath between e_i and e_{i+1} $0 \leq i < t$, where $e_0 = p$ and $e_t = q$, and (e_0, e_1, \dots, e_t) is the extreme sequence of path.

Chapter 3

Overview of Algorithm

The overview of the algorithm for finding a pair of noncrossing paths $\pi_{aa'}$ and $\pi_{bb'}$, within a rectilinear polygon is given below. Here $\pi_{aa'}$ is the target path from a to a' and $\pi_{bb'}$ is the target path from b to b' . These two paths will be referred to as *goal pair*.

Let S_{xy} denote any shortest path between two points x and y ((x, y) may be (a, a') or (b, b')), disregarding the crossing with the other path, if any. Now, any of the following five configurations may yield the *goal pair*.

- (1) There is no U-subpath on $\pi_{aa'}$ or $\pi_{bb'}$ supported by a terminal. Therefore, $EX(\pi_{aa'})$ equals $EX(S_{aa'})$ and $EX(\pi_{bb'})$ equals $EX(S_{bb'})$.
- (2) There is no U-subpath on $\pi_{bb'}$ supported by a . Therefore, $EX(\pi_{aa'})$ equals $EX(S_{aa'})$ and $EX(\pi_{bb'})$ equals $EX(S_{ba}) || EX(S_{ab'})$.
- (3) There is no U-subpath on $\pi_{bb'}$ supported by a' . Therefore, $EX(\pi_{aa'})$ equals $EX(S_{aa'})$ and $EX(\pi_{bb'})$ equals $EX(S_{ba'}) || EX(S_{a'b'})$.
- (4) There is no U-subpath on $\pi_{aa'}$ supported by b . Therefore, $EX(\pi_{bb'})$ equals $EX(S_{bb'})$ and $EX(\pi_{aa'})$ equals $EX(S_{ab}) || EX(S_{ba'})$.
- (5) There is no U-subpath on $\pi_{aa'}$ supported by b' . Therefore, $EX(\pi_{bb'})$ equals $EX(S_{bb'})$ and $EX(\pi_{aa'})$ equals $EX(S_{ab'}) || EX(S_{b'a'})$.

Algorithm 2NCP

1. For each of the five configurations described above, do the following:
 - 1.1 Find the smallest paths that correspond to the regular paths defined in the configuration. (There are two regular paths in the first configuration and three regular paths in the remaining four configurations.)

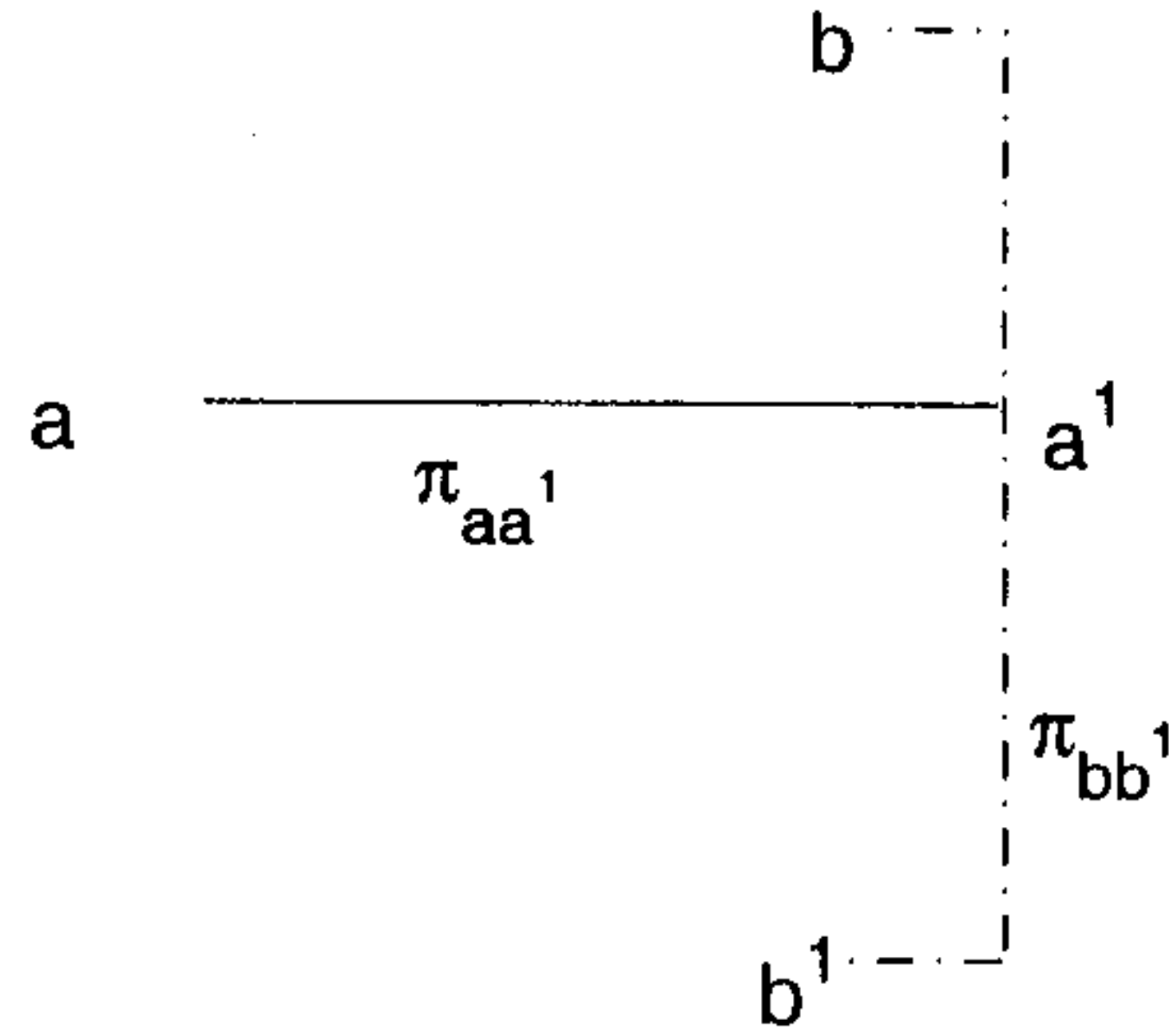


Figure 3.1: U-subpath on $\pi_{bb'}$ supported by a' , where a' is the terminal of $\pi_{aa'}$

- 1.2 Find a canonical smallest path, π_1 , from one of the two smallest paths, denoted π_1 and π_2 , or from the inner path if there are three paths, denoted π_1 , π_2 and π_3 . Call it the first path.
- 1.3 Compute the canonical smallest path from another smallest path, π_2 . Call it the second path. While computing the second path, we detect and record any interaction between the first and second paths.
- 1.4 If the solution is not of the first configuration, find the third canonical smallest path, π_3 (called the third path if the configuration is not the first one), and detect and record any interactions between the first and third path.
- 1.5 Decide the types of all the interactions and resolve them according to their types.
- 1.6 The optimal solution of the current configuration is the set of paths composed of all individual optimal staircase subpaths after resolving all the interactions (by pushing them into pushed paths).
2. If the goal pair is of the first configuration, then it is a smallest pair. Otherwise, compare the resultant paths of the remaining four configurations, and return a smallest pair, if it exists, or a minimum-bend shortest pair or a shortest minimum-bend pair as desired.

Chapter 4

Implementation Details

4.1 Data Structures

We store the polygon in a doubly link circular list, called P , and the paths in two doubly link lists, say L_1 and L_2 . Each element of the link list contains a point $c(x_c, y_c)$ in 2D, and two pointers namely, $next$ and $prev$. The $next$ field of an element in the link list P (corresponding to the polygon) contains the vertex of the polygon next to point c in anti-clockwise order, and the $prev$ field contains the vertex of the polygon previous to the point c in clockwise order. For the path L_1 (L_2), the $next$ field of an element contains the next point on the path from source point to the sink point, and the $prev$ field contains the next point on the same path if observed from sink to the source.

4.2 Shortest Path Algorithm

We use P to denote the header of doubly connected circular link list representing the polygon P ; L_1 and L_2 denotes the header of the respective link lists. Let p and q denote the source and destination points of the path; both p and q are inside the polygon. The algorithm, stated below, returns the shortest path from p to q inside the rectilinear polygon.

To find a shortest path between two points p and q in a polygon P , start by drawing a vertical line down from p until a boundary of P is encountered, let this point be s . And drawing another vertical line down from q until a boundary of P is encountered, let this point be t . Then our initial path is from p , then follow the boundary of P from s until t will be encountered, followed by q . This is called starting path from p to q .

There are five different types of transformations which can be used to improve a starting path. These transformation are shown graphically in the figure 4.1. Transformations Vertex_Edge visible and Vertex_Vertex visible reduce both distance and number of bends. A Vertex_Edge visible transform replaces an arbitrary simple path with a line segment between a vertex and

an edge. Transformation Vertex_Vertex visible similarly. Transformation U reduces distance by ensuring that U-turns which go around an intruding part of the exterior of the polygon do so in the shortest possible way. Transformation Z reduces the number of bends by removing unnecessary horizontal and vertical shifts in Z-shaped(or S shaped) parts of a path. Transformation C is a "cleanup" operation that eliminates unnecessary vertices after applications of the other types of transformations. Note that all transformation (and their reflections) can be applied in any of the four possible rectilinear orientations, not just the orientations shown in the figure 4.1.

An *ending path* is a simple path that is obtained from a starting path by repeatedly applying transformations from figure 4.1 until no further transformations can be made (by maintaining the property that all intermediate paths that occur during this transformation process are simple). Note that transformations Vertex_Edge visible, Vertex_Vertex visible, C all reduce the number of segments in the path, so they can not be applied more than $O(n)$ times. To show that transformation U can be applied at most $O(n)$ times, first observed that there are at most $O(n)$ potential locations for a U-transformation in the starting path and that an application of transformation U cannot create a "new" location for a U transformation. Each of the other four transformations introduces at most a constant number of "new" locations for a U transformation because at most two of the segments in a path resulting from one of these transformations is a new segment or has a different length than before the transformation. It follows that transformation U cannot be applied more than $O(n)$ times. The ending path is a shortest path.

Function *Shortest_Path_in_a_Simple_Rectilinear_Polygon*

Begin

 call *initial_path*(P, p, x_1, y_1, n).

 call *final_path*(P, p, x_1, y_1, n).

End.

Algorithm *initial_path*(P, L_1, p, q, n)

/* Input parameters are the pointer P , the points p and q , and the number of vertices of the polygon n */

/* Output parameter is the header L_1 of the link list representing the path */

Begin

 insert p in L_1 ;

$m = \text{just_down_point}(P, p, a_1)$; (* a_1 is an element of the path L_1 *)

 insert a_1 in the link list L_1 ;

$m = \text{just_down_point}(P, q, b_1)$; (* b_1 is also an element of the path L_1 *)


```

     $t_1 = \text{set\_previous\_point}(P, a_1);$ 
     $t_1 = t_1 \rightarrow \text{next};$ 
     $t_2 = \text{set\_previous\_point}(P, b_1);$ 
    while ( $t_1 \neq t_2$ ) do
        insert  $t_1$  in  $L_1$ ;
         $t_1 = t_1 \rightarrow \text{next};$ 
    Endwhile
    insert  $t_1$  in  $L_1$ ;
    insert  $b_1$  in  $L_1$ ;
    insert  $q$  in  $L_1$ ;
    return  $L_1$ ;
End

Function set\_previous\_point( $P, a_1$ )
/* Input parameters are the pointer  $P$ , a point  $a_1$  on the boundary of the polygon. */
/* This function returns a vertex  $v$  of the polygon such that  $a_1$  lies on the segment  $\overline{v, v \rightarrow \text{next}}$  */
*/

Begin
     $t_1 \leftarrow P;$ 
     $t_2 \leftarrow (t_1 \rightarrow \text{next});$ 
     $\text{found} \leftarrow \text{FALSE};$ 
    while ( $\text{found} = \text{FALSE}$ ) do
        if ( $(t_1 \rightarrow x_c = t_2 \rightarrow x_c)$  and ( $a_1 \rightarrow x_c = t_2 \rightarrow x_c$ )
            and ( $t_1 \rightarrow y_c < a_1 \rightarrow y_c < t_2 \rightarrow y_c$  or  $t_1 \rightarrow y_c > a_1 \rightarrow y_c > t_2 \rightarrow y_c$ )) then
             $\text{found} = \text{TRUE};$ 
            return  $t_1$ ;
        Endif
        if ( $(t_1 \rightarrow y_c = t_2 \rightarrow y_c)$  and ( $a_1 \rightarrow y_c = t_2 \rightarrow y_c$ )
            and ( $t_1 \rightarrow x_c < a_1 \rightarrow x_c < t_2 \rightarrow x_c$  or  $t_1 \rightarrow x_c > a_1 \rightarrow x_c > t_2 \rightarrow x_c$ )) then
             $\text{found} = \text{TRUE};$ 
            return  $t_1$ ;
        Endif
         $t_1 = t_2;$ 
         $t_1 = t_2 \rightarrow \text{next};$ 
    Endwhile
End

Function just\_down\_point( $P, a_1$ )
/* Input parameters are the pointer  $P$ , a point  $a_1$  inside the boundary of the polygon. */
/* This function returns a vertex  $v$  on the boundary of the polygon  $P$   $v$  is the nearest point of
the polygon from  $a_1$  in the downward direction */

```

```

Begin
  distance = 1000; m = 0;
  t1 ← P;
  t2 ← (t1 → next);
  while (t2 ≠ P) do
    if (t1 → yc = t2 → yc and t1 → xc ≤ a1 → xc and t2 → xc ≥ a1 → xc)
      if (t1 → yc ≤ a1 → yc and distance ≥ |a1 → yc - t1 → yc |)
        set point v such that a1 → xc as x component and t1 → yc as y component;
        m = 1; distance = |a1 → yc - t1 → yc |
      Endif
      t1 ← t2; t2 ← (t1 → next);
    Endif
    t1 ← t2; t2 ← (t1 → next);
  Endwhile
  if (t1 → yc = t2 → yc and t1 → xc ≤ a1 → xc and t2 → xc ≥ a1 → xc )
    if (t1 → yc ≤ a1 → yc and distance ≥ |a1 → yc - t1 → yc |)
      set point v such that a1 → xc as x component and t1 → yc as y component;
      m = 1; distance = |a1 → yc - t1 → yc |
    Endif
  Endif
  return m;
End

```

Similar Algorithms for

- 1) just_up_point
- 1) just_right_point
- 1) just_left_point

Function final_path

*/*Input parameters are the pointer P, the pointer p (header of the path), a pointer E(header of the extreme sequence of the path), n */*
/ This function returns nothing */*

Begin

```

  call pre_final_path(P, p, n);
  for (every vertex of path p) do
    call horizontal_vertex_edge_visible_transform(P, p, n);
  Endfor
  for (every vertex of path p) do

```

```

        call vertical_vertex_edge_visible_transform( $P$ ,  $p$ ,  $n$ );
    Endfor
    call c_transformation( $p$ );
    while(path  $p$  change)do
        call u_transformation( $P$ ,  $p$ ,  $E$ ,  $n$ );
    Endwhile
    while(path  $p$  change)do
        call z_transformation( $P$ ,  $p$ ,  $E$ ,  $n$ );
    Endwhile
    call c_transformation( $p$ );
End

```

Function *pre_final_path*

*/*Input parameters are the pointer P , the pointer p (header of the path), n */*
/ This function returns nothing */*

Begin

*For each of the vertex namely first and last vertex of the path,
 find the visibility in the direction left and right of the path by
 calling the routines just_left_point and just_right_point and if
 possible, take one which minimizes the path over another.*

Adjust (by minimizing path length) the path accordingly.

End

Function *horizontal_vertex_edge_visible_transform*

/ Input parameters are the pointer P , the pointer p , t_1 (a vertex
 whose visibility we found), n */*
/ This function returns nothing */*

Begin

*For the vertex t_1 , find the visibility in the direction left and right (which
 applicable for the point t_1) of the path by calling the routine just_left_point;
 or just_right_point (accordingly);*

If possible, adjust the path by minimizing the path length;

End

Similar Function for vertical_vertex_edge_visible_transform.

Function *c_transformation*

/ Input parameters are the pointer p */*
/ This function returns nothing */*

Begin

Let t_1, t_2 and t_3 are three consecutive vertex of the path p , initially t_1 is the header of p ;
change = 0;

while($t_3 \neq p$)**do**

if(t_1, t_2 and t_3 are in a straight line)**then**

remove t_2 from p ;

change = 1;

Endif

if(change = 1)**then**

$t_2 = t_3$;

$t_3 = t_2 \rightarrow \text{next}$;

change = 0

else

$t_1 = t_2$;

$t_2 = t_3$;

$t_3 = t_2 \rightarrow \text{next}$;

End if

Endwhile

End

Function *u_transformation*

*/*Input parameters are the pointer P , the pointer p (header of the path), a pointer E (header of the extreme sequence of the path), n */*

/ This function returns 0, if path not change by this transformation and 1 otherwise. */*

Begin

Let t_1, t_2, t_3 and t_4 are four consecutive vertex of the path p , where initially t_1 is the header of p ;

change = 0;

while($t_3 \neq P$)**do**

if(t_1, t_2, t_3 and t_4 form a *u*-subpath)

find the visible points p_1 and p_2 from t_1 and t_4 respectively in the direction t_4 and t_1 respectively

```

    find the nearest points1 between  $p_1$  to  $p_2$  on the polygon from  $t_2$ ;
    make the distance 0 between  $s_1$  and  $t_2$  by changing  $t_2$  and  $t_3$  approately;
    if(  $t_2$  change)then
        change = 1;
    Endif
    insert the extreme point (if exists) in  $E$ ;
Endif
 $t_1 = t_2$ ;
 $t_2 = t_3$ ;
 $t_3 = t_4$ ;
 $t_4 = t_3 \rightarrow \text{next}$ ;
Endwhile
return change;
End

Function z_transformation
/*Input parameters are the pointer  $P$ , the pointer  $p$  (header of the path) */
/* This function returns 0, if path not change by this transformation and 1 otherwise. */

Begin
    Let  $t_1, t_2, t_3$  and  $t_4$  are three consicutive vertex of the path  $p$ , initially
     $t_1$  is the header of  $p$ ;
    change = 0;
    while( $t_4 \neq p$ )do
        if( $t_1, t_2, t_3$  and  $t_4$  form the z-subpath)then
            check the visibility from  $t_1, t_3$  &  $t_2, t_4$  (by checking which set applicable) and
            check the validity of decreasing one bend of the path. If decreasing possible
            then decrease the appropriate bend of the path  $p$ , by taking appropriate action;
            change = 1;
        Endif
         $t_1 = t_2$ ;
         $t_2 = t_3$ ;
         $t_3 = t_4$ ;
         $t_4 = t_4 \rightarrow \text{next}$ ;
    Endwhile
    return change;
End

```

This is the end of finding Smallest path algorithms.

4.3 Detection of crossing of two given paths :

Now we can find smallest paths using the algorithms given in section 4.2 for given polygon and a pair of terminals. So we, first find a pair of paths using the above algorithm, then check whether they are crossing or not. If they are cross then we will find detour. So now, our task is to checking, after finding a pair of paths, they are cross or not.

To detect shortest paths $S_{aa'}$ and $S_{bb'}$ cross or not, We do the following. We first compute the the extrem sequences of paths $E\pi_{aa'} = (a, a_1, \dots, a_k, a')$ and $E\pi_{bb'} = (b, b_1, \dots, b_l, b')$ (in the time of U-transformation), where $k, l \geq 0$. Call the staircase paths from a to a_1 and a_k to a' the leading part and trailing part of $S_{aa'}$, respectively, and the canonical path from a_1 to a_k , the main part of $S_{aa'}$. It is possible that the main part of $S_{aa'}$ is empty, in which case, $S_{aa'}$ contains just one ($k = 0$) or two ($k = 1$) staircase subpaths. Now we divide detecting crossing in two parts. First part check crossing between two main parts of the paths and second part consist of checking leading and trailing part of $S_{aa'}$ with $S_{bb'}$ and leading and trailing part of $S_{bb'}$ with $S_{aa'}$.

To check the first part we check $k, l \geq 0$ or not. If $k, l < 0$ then checking this part is not required. If $k, l \geq 0$ then consider a homeomorphic transformation of δP (boundary of polygon) in to a unit circle, and extreme point mapped to the unit circle accordingly (shown in figure 4.2). Then map the first and last extreme points of the both paths on that unit circle. Then check (by linear scanning the extreme sequence of polygon) the four extreme points on that unit circle are came in alternate from both paths or not. If they are in alternate then the paths are cross each other, otherwise not.

In second part of checking crossing, we check the crossing of aa_1 and $a_k a'$ with each of $bb_1, b_1 b_2, \dots, b_l b'$ and check the crossing of bb_1 and $b_l b'$ with each of $aa_1, a_1 a_2, \dots, a_k a'$. To check the crossing between $a_i a_{i+1}$ and $b_j b_{j+1}$, we take the intersection rectangle R defined by a_i, a_{i+1} and b_j, b_{j+1} . If the intersection points of $a_i a_{i+1}$ with R and intersection points of $b_j b_{j+1}$ with R alternative when we round the rectangle in any direction then the portions $a_i a_{i+1}$ and b_j, b_{j+1} are cross with each other, otherwise not.

Function find_poly_extrempoint

/* Input parameters are the pointer P , EP (adress of the header of extreme sequence of the polygon) */

/* This function returns 0, if the number of extreme point is none and 1, otherwise. */

Begin

Let t_1, t_2, t_3 and t_4 are three consicutive pointers to the vertices of the path P , initially t_1 is the adress of the header of P ;

while ($t_4 \neq P$) **do**

```

    if( $\overline{t_2 t_3}$  is an extreme edge)then
        insert the middle point of the edge  $\overline{t_2 t_3}$  in EP;
    Endif
     $t_1 = t_2$ ;
     $t_2 = t_3$ ;
     $t_3 = t_4$ ;
     $t_4 = t_4 \rightarrow \text{next}$ ;
Endwhile
End

```

Function *check_cross_e_extrem*

/ Input parameters are the pointers EP, E1, E2 (pointers to the headers of extreme sequences of the polygon, path_1 and path_2 respectively) */*
/ This function returns 1, if within the extreme sequence two paths are crossed and 0, otherwise. */*

Begin

Let t_1 & t_2 are the pointers of first & last extreme point respectively of the path_1 and t_3 t_4 are the pointers of first & last extreme point respectively of the path_2. Also let p_1 is the pointer of first extreme point of the polygon;
 $\text{cross} = 0$;

while(p_1 not traverse the all extreme points in EP)**do**

 check whether p_1 takes alternative from $\{t_1, t_2\}$ and $\{t_3, t_4\}$ or not;

if(alternative) **then**

$\text{cross} = 1$;

break;

Endif

$p_1 = p_1 \rightarrow \text{next}$;

Endwhile

return cross;

End

Function *check_cross_e_simple*

/ Input parameters are the pointers EP, E1, E2 (pointers to the headers of extreme sequences of the polygon, path_1 and path_2 respectively) and the pointers p1 & p2 (pointers to the headers of path_1 and path_2 respectively) */*

/ This function returns 1, if leading part or trailing part of path-1 cross with path-2 or if leading part or trailing part of path-2 cross with path-1 and 0, otherwise. */*

Begin

Let $A = \{p_2, p_3, \dots, p_{k-1}\}$ and $B = \{q_2, q_3, \dots, q_{r-1}\}$ be the extreme sequence from source to destination of the path-1 and path-2 respectively and p_1 & p_k are respectively source and destination of path-1 and q_1 & q_r are respectively source and destination of path-2.

$cross = 0;$

for(every segment from q_i to $q_{i+1}, i = 1, 2, \dots, r - 1$)**do**

draw two rectangles with diagonal $\overline{p_1 p_2}$ and $\overline{q_i q_{i+1}};$

find the intersection rectangle $R;$

Let t_1 and t_2 are two intersection point of path-1 with R . Also let t_3 and t_4 are two intersection point of path-2 with $R;$

Now traverse the rectangle in any direction;

if(points come alternatively from the set $\{t_1, t_2\}$ and $\{t_3, t_4\}$)**then**

$cross = 1;$

Endif

Endfor

Similar for loop for rectangles with diagonal $\overline{p_{k-1} p_k}$ and $\overline{q_i q_{i+1}}, i = 1, 2, \dots, r - 1;$

Similar for loop for rectangles with diagonal $\overline{q_1 p_2}$ and $\overline{q_i q_{i+1}}, i = 1, 2, \dots, k - 1;$

Similar for loop for rectangles with diagonal $\overline{p_{r-1} p_r}$ and $\overline{q_i q_{i+1}}, i = 1, 2, \dots, k - 1;$

return $cross;$

End

4.4 Detour of the Paths :

Let $[a, a^1]$ and $[b, b^1]$ be two smallest paths inside the polygon. We detour paths, if paths are crossing. Let $[a, a^1]$ and $[b, b^1]$ are crossed path. Our goal is to find the noncrossing optimal paths. So, we detour the paths a, a^1 and b, b^1 . There are four type of detour can be possible, namely,

- one path is $[a, a^1]$ and another path is the union of two paths, namely, $[b, a]$ and $[a, b^1]$.
- one path is $[a, a^1]$ and another path is the union of two paths, namely, $[b, a^1]$ and $[a^1, b^1]$.
- one path is $[b, b^1]$ and another path is the union of two paths, namely, $[a, b]$ and $[b, a^1]$.
- one path is $[b, b^1]$ and another path is the union of two paths, namely, $[a, b^1]$ and $[b^1, a^1]$.

In each of the above type of detour, we first find the smallest path by the algorithms, given in section 4.2. Clearly these paths will be declared as noncrossing, if we check by the algorithms described in section 4.3. But always this can not happen in the figure 4.3 So, we need to find canonical paths, which we will describe in next section. After finding canonical paths, we find which of the four type of paths, is optimal with respect to our optimal criteria and we report accordingly.

4.5 Canonical path

Definition of a Canonical path is already defined in Chapter 2. Here we describe the algorithm of constructing a canonical path from a smallest path.

Function *canonicalpath*

/ Input parameters are pointers P, p, E, n(where P, p, E, points to header of polygon, path, extreme sequence of the path and n is the number of vertices of the polygon */*
/ This function returns to the pointer p */*

Begin

Let $\{t_2, t_3, \dots, t_{k-1}\}$ be the sequence of extreme points of the path p. Also let t_1 and t_k are respectively source and destination points of the path p;

for (each t_i, t_{i+1} ($i = 1, 2, \dots, k - 1$))do

if($(t_i.x \geq t_{i+1}.x$ and $t_i.y \geq t_{i+1}.y$) || $(t_i.x \leq t_{i+1}.x$ and $t_i.y \leq t_{i+1}.y$))then

call canonical_pushed_xy(P, p, t_i , t_{i+1} , n);

else

call canonical_pushed_cross_xy(P, p, t_i , t_{i+1} , n);

Endif

Endfor

return p;

End

Function *canonical_pushed_xy*

/ Input parameters are pointers P, and p. Pointers e_1 and e_2 pointing two points, between which we find canonical path */*

/ This function returns nothing */*

Begin

Let t_1 and t_2 are respectively just previous point of e_1 and e_2 , but in the case of source point of p, $t_1 = e_1$ and in the case of destination point of p, $t_2 = e_2$. Let $\{v_1, v_2, \dots, v_r\}$ be the sequence of vertices of the path p from t_1 to t_2 ;

if($e_1.x \geq e_2.x$ and $e_1.y \geq e_2.y$)then

for (each $\overline{v_i v_{i+1}}$ $i = 2, \dots, r - 1$) do

if($v_i.y = v_{i+1}.y$)then

see the visibility of the points v_i and v_{i+1} in downward direction on the polygon.

Let m and n be two visible point. Find the maximum height of the vertices from m to n on the polygon;
 if possible decrease the height of $\overline{v_i v_{i+1}}$ by changing the value of y co-ordinate of v_i and v_{i+1} ;
 else
 see the visibility of the points v_i and v_{i+1} in left direction on the polygon. Let m and n be two visible point. Find the maximum x co-ordinate of the vertices from m to n on the polygon;
 if possible decrease the value of x co-ordinate of $\overline{v_i v_{i+1}}$ by changing the value of x co-ordinate of v_i and v_{i+1} ;
 Endif
 Endfor
 else
 for (each $\overline{v_i v_{i+1}}$ $i = 2, \dots, r - 1$) do
 if($v_i.y = v_{i+1}.y$) then
 see the visibility of the points v_i and v_{i+1} in upward direction on the polygon. Let m and n be two visible point. Find the minimum y co-ordinate of the vertices from m to n on the polygon;
 if possible increase the height of $\overline{v_i v_{i+1}}$ by changing the value of y co-ordinate of v_i and v_{i+1} ;
 else
 see the visibility of the points v_i and v_{i+1} in right direction on the polygon. Let m and n be two visible point. Find the minimum x co-ordinate of the vertices from m to n on the polygon;
 if possible increase the value of x co-ordinate of $\overline{v_i v_{i+1}}$ by changing the value of x co-ordinate of v_i and v_{i+1} ;
 Endif
 Endfor
 Endif
 End

- Similar algorithm for the routine *canonical_pushed_cross_xy* except the type of paths. Routine *canonical_pushed_xy* compute canonical path of type XY or $(-X)(-Y)$ from the same type, but routine *canonical_pushed_cross_xy* compute canonical path of type $(-X)Y$ or $(X)(-Y)$ from the same type.

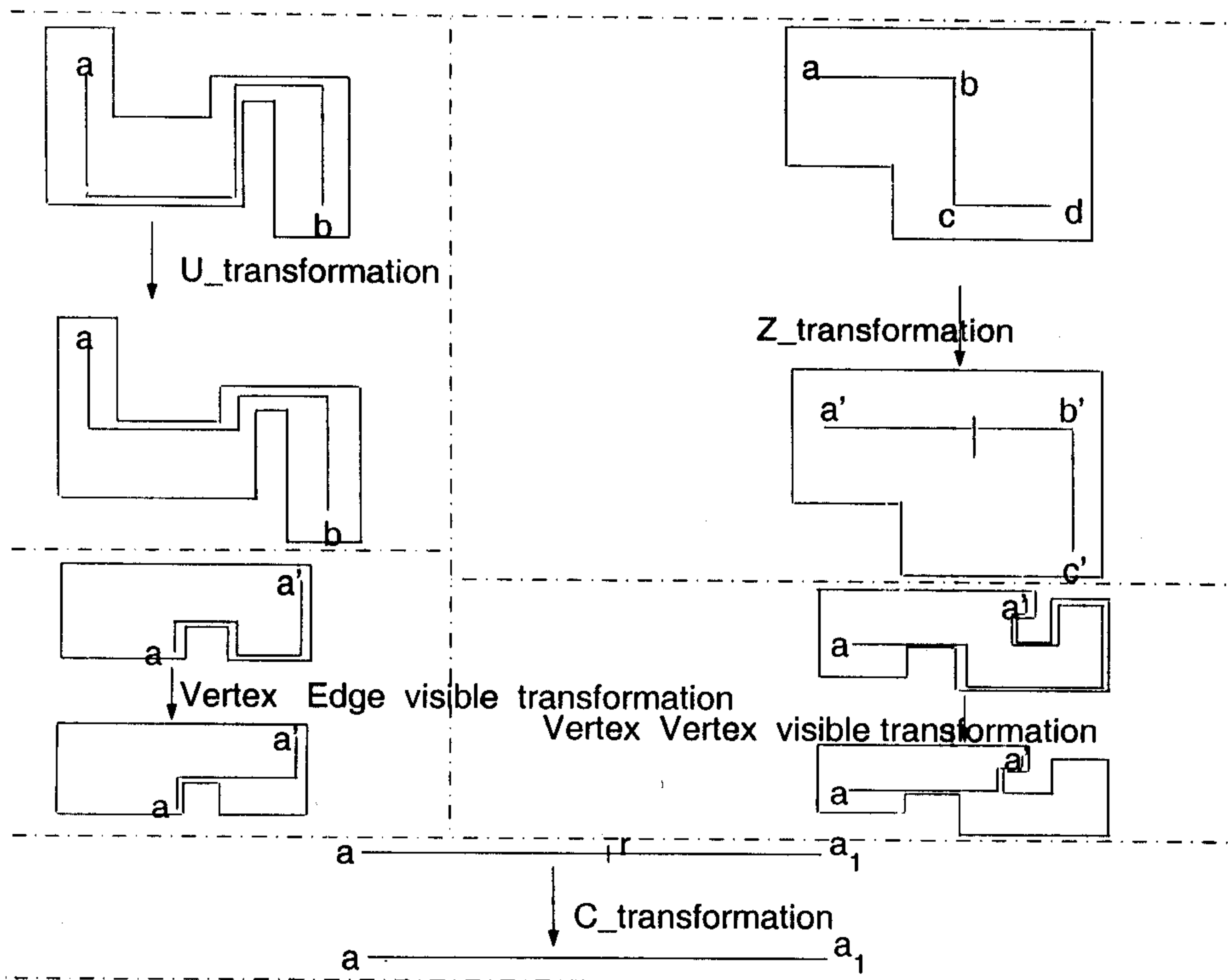


Figure 4.1: 3 types of transformations used in finding shortest path algorithm

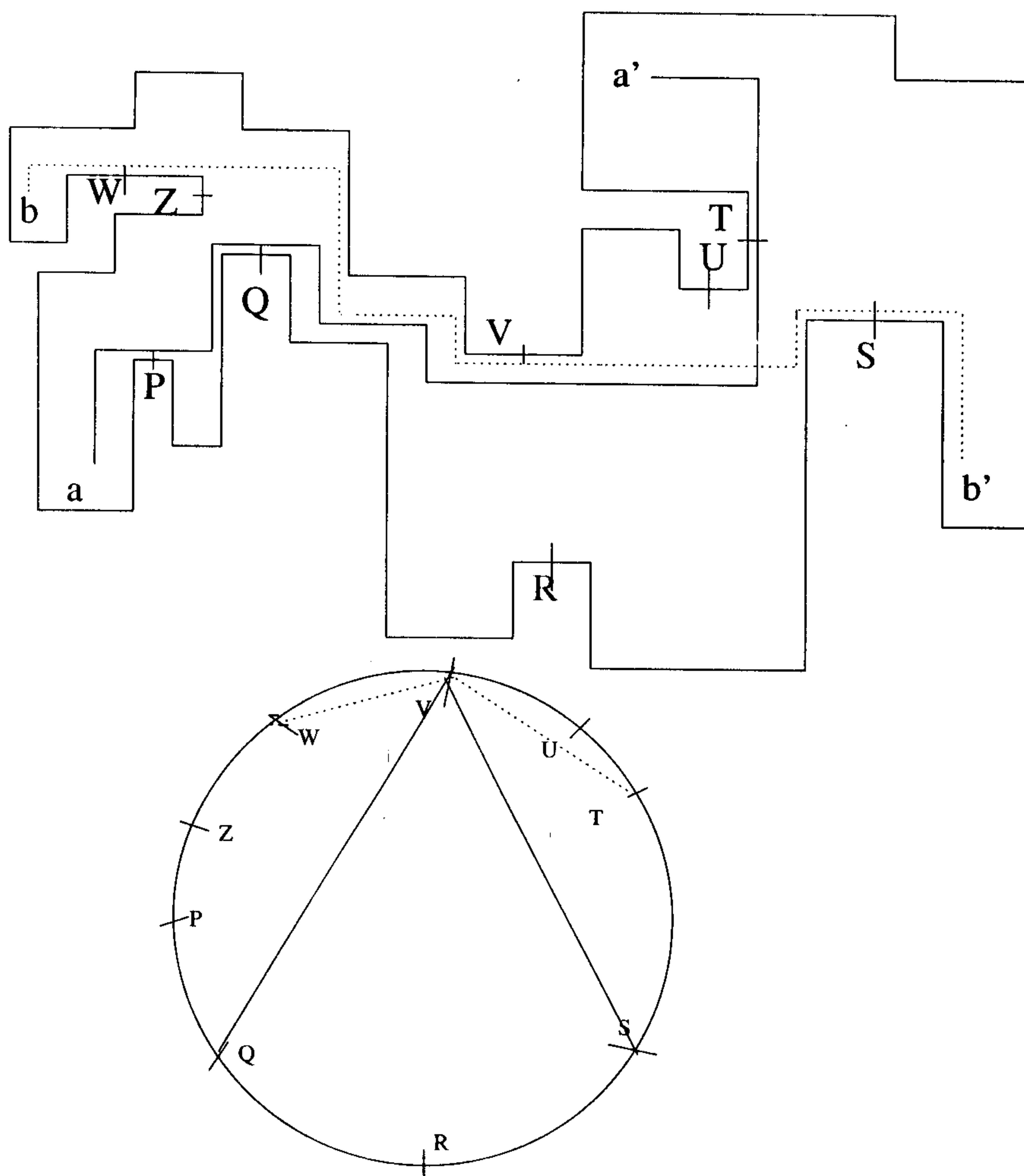


Figure 4.2: Homeomorphic transformation of the boundary of polygon to a circle and the mapping of extreme points to the circle

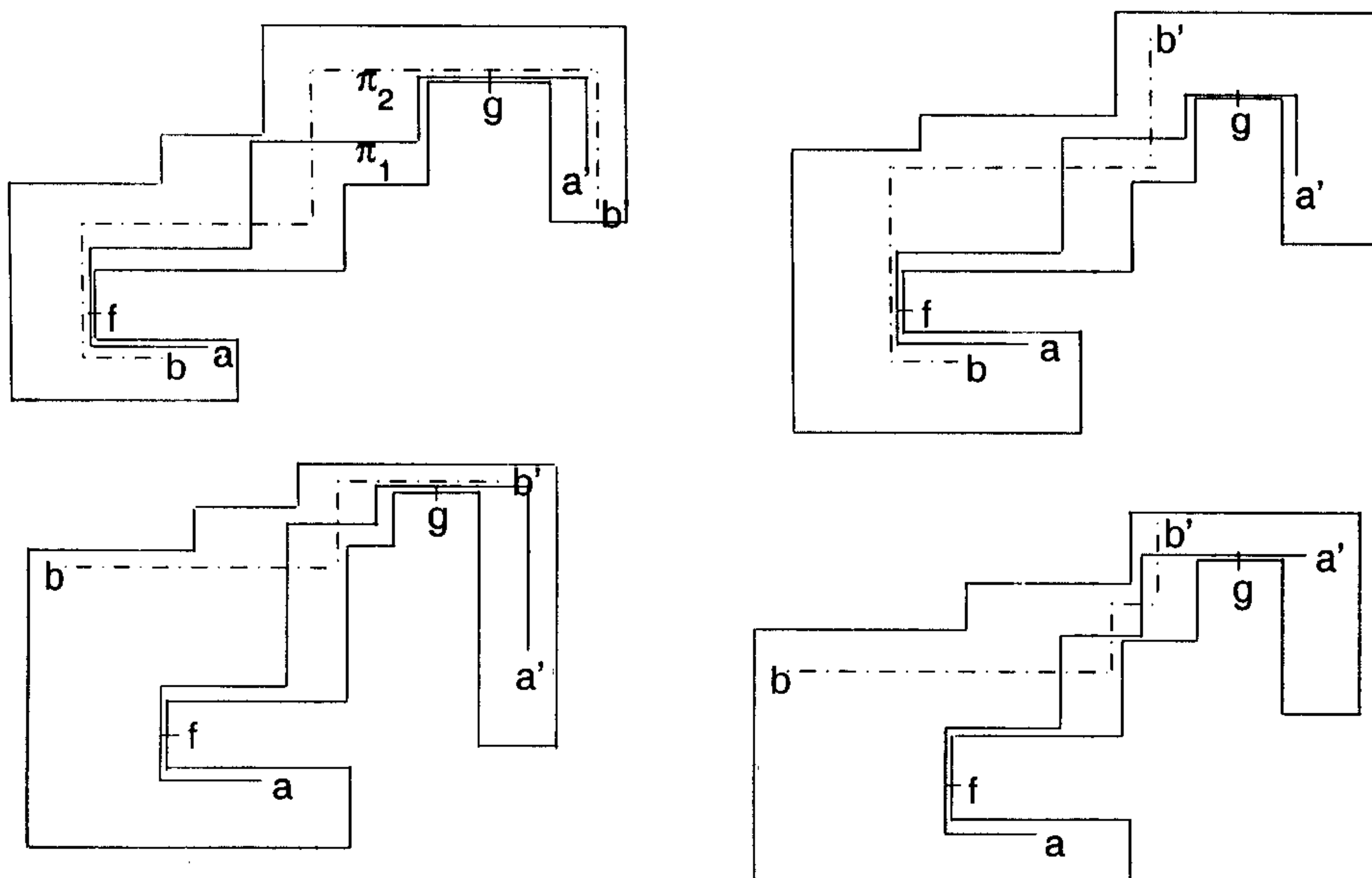


Figure 4.3: Interaction of two $X - Y$ paths

Chapter 5

Experimental Results

Here we give two results, details of these results are in tables and in figure shown below.

	# of vertices	# of extreme point	Lengths	Bends	CPU Time
<i>Polygon</i> <i>figure 5.1</i>	46	8	210	46	0.01 sec
<i>Paths[aa':bb']</i>	15	6	159	15	
<i>Polygon</i> <i>figure 5.2</i>	48	9	231	48	0.002
<i>Paths[aa':bb']</i>	17	5	132	17	

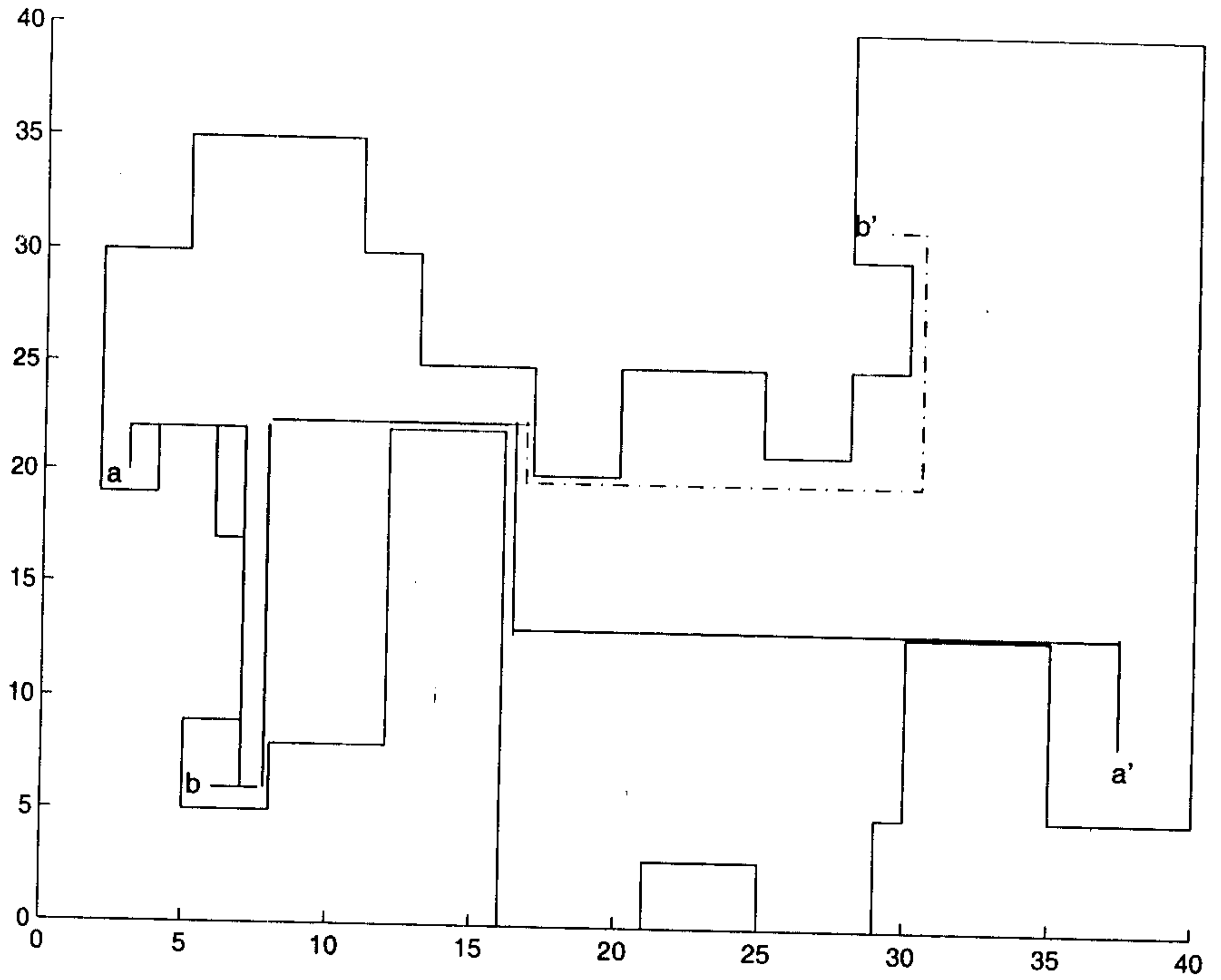


Figure 5.1: One of the experimental result

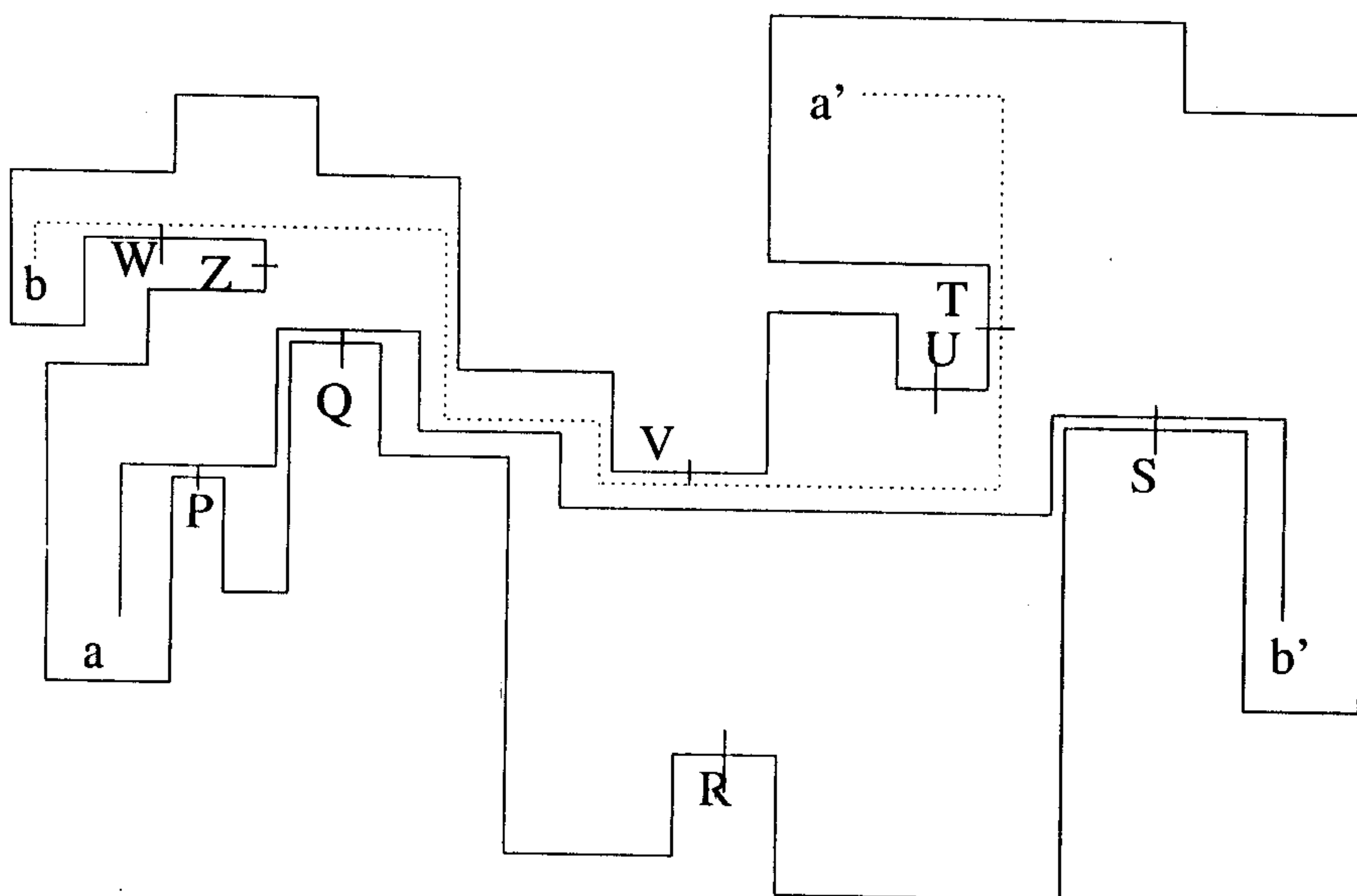


Figure 5.2: One of the experimental result

Chapter 6

Conclusion

We have presented a $O(n^2)$ algorithm for finding a pair of noncrossing rectilinear paths connecting two distinct pairs of terminals within a simple rectilinear polygon. We have shown that the noncrossing pairs of paths that have a minimum total length and minimum total number of bends may not exist. The $O(n^2)$ time algorithm computes such an optimal pair if it exists, and computes an optimal pair with either the length or the number of bends as the primary measure and the other is the secondary measure. The algorithm can also be modified to report an optimal pair of paths whose combined cost, an increasing function of both length and the number of bends, is minimum. the query mode in which two pairs of terminals are given as a query, and the best pair based on a prespecified optimization criterion is being investigated. Complexity of our algorithms is $O(n^2)$ due to find the visibility point, in[1], they find triangulation of a simple polygon in $O(n)$ and using this result in[6], they find the visibility in $O(n)$ time. The complexity $O(n)$, is theoretically possible, but in implementation issue it is imposible to do the result in $O(n)$ time. Generalization to $k > 2$ case of this problem remains open.

Bibliography

- [1] B. Chazelle. Triangulating a simple polygon in linear time. *DISCRETE AND COMPUTATIONAL GEOMETRY*, 6:485–524, 1991.
- [2] K. L. Clarkson, S. Kapoor, and P. M. Vaidya. Rectilinear shortest paths through polygonal obstacles in $o(n \log^{3/2} n)$ time. unpublished manuscript, 1988.
- [3] P. J. de Rezende, D. T. Lee, and Y. F. Wu. Rectilinear shortest paths with rectilinear barriers. In *1st ACM Symp. Computational Geometry*, pages 204–213, 1985.
- [4] D. T. Lee, C. D. Yang, and C. K. Wong. Rectilinear paths among rectilinear obstacles. In *DISCRETE APPLIED MATHEMATICS*, volume 70, pages 185–215. Elsevier Science B. V., 1996.
- [5] Kenneth M. McDonald and Joseph G. Peters. Smallest paths in simple rectilinear polygons. *IEEE TRANSACTION ON COMPUTER-AIDED DESIGN*, 11(7):864–875, 1992.
- [6] C. D. Yang and D. T. Lee. The smallest pair of noncrossing paths in a rectilinear polygon. *IEEE TRANSACTION ON COMPUTERS*, 46(8):930–941, 1995.