SCHOOL OF OPERATIONS RESEARCH
AND INDUSTRIAL ENGINEERING
COLLEGE OF ENGINEERING
CORNELL UNIVERSITY
ITHACA, NEW YORK 14853

TECHNICAL REPORT NO. 881

December 1989

STRUCTURED VISIBILITY PROFILES WITH APPLICATIONS TO PROBLEMS IN SIMPLE POLYGONS

By

Paul J. Heffernan
Joseph S. B. Mitchell

Structured Visibility Profiles with Applications to Problems in Simple Polygons

Paul J. Heffernan*and Joseph S. B. Mitchell†

Cornell University
Ithaca, NY 14853

January 10, 1990

Abstract

A number of problems in computational geometry involving simple polygons can be solved in linear time once the polygon has been triangulated. Since the worst-case time bound for triangulating a general simple polygon is currently $O(t(n)) = O(n \log\log n)$, these algorithms are not linear time in the worst case. In this paper we define the *structured visibility profile* of a polygonal path and show how to compute it in linear time. We apply our result to solve many problems in linear (optimal) time that previously required triangulation. Our list of problems includes: translation separability of two simple polygons, computing the weak visibility region for a segment within a simple polygon, finding shortest monotone paths in a simple polygon, ray shooting from an edge, and the convex rope problem.

Our strategy is the same for each problem: we replace the polygon in question with a subpolygon such that the subpolygon can be triangulated in linear time, and the subpolygon contains all the information needed to answer the question for the larger polygon.

1 Introduction

There have been many algorithms in computational geometry that address various problems involving visibility questions within a simple polygon P. The problem of computing the visible portion of a polygon from a given point s within P can be solved in time $O(n)$, where n is the number of vertices defining P (see [Le], [EA], [JS]). The related problem of computing the portion of P that is illuminated by a line segment e within P has been solved in linear time *if a triangulation is given for* P. It is one of the deepest open questions in computational geometry to determine if triangulation of a simple polygon can be accomplished in linear time. The current best bound is given in the work of [TV].

In this paper, we introduce a new methodology of processing a simple polygon that facilitates answering many questions involving visibility, shortest paths, and separability in *linear* time, *without resorting to triangulation.* We apply our technique to answer a large catalogue of problems in linear time, improving previous time bounds that relied on triangulation. In particular, we improve the time bounds on four of the problems considered by Guibas, Hershberger, Leven, Sharir, and Tarjan [GHLST]:

*Supported by a NSF graduate fellowship. This author also wishes to thank DIMACS and Rutgers University for use of facilities.

†Partially supported by NSF grants IRI-8710858 and ECSE 8857642, and by a grant from Hughes Research Laboratories.

- (Weak edge visibility) Given an edge e within P, compute the subpolygon of P that is weakly visible from e. [GHLST] and [To1] give bounds of $O(t(n)+n)$ for this problem. We obtain an optimal bound of $\Theta(n)$.
- Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point q, determine the portion of e that q can see. [GHLST] give a bound of $O(t(n)+n)$ on the preprocessing, which we improve to $\Theta(n)$.
- (Ray shooting from an edge) Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point $q \in e$ and a direction d, determine the point on the boundary of P where a bullet would exit if fired from q in direction d. [GHLST] give a bound of $O(t(n)+n)$ on the preprocessing, which we improve to $\Theta(n)$.
- (Convex rope problem) Given a point s on the convex hull of P, determine those points on the boundary of P that admit "convex ropes" from s around the outside of P.

In addition to the above results, we solve the following problems in linear time:

- (Monotone paths in a simple polygon) Given two points s and t within P, determine if there exists a monotone path from s to t within P, and, if so, produce one and give the set of all directions d for which there exists a d-monotone path from s to t. In fact, if there exists a monotone path from s to t, we can produce both the (unique, "taut-string") shortest path from s to t and a (non-unique) minimum-link path from s to t. Previous results for this problem required a triangulation of P ([ACM], [GHLST]).
- (Translation separability) Given two disjoint simple polygons P and Q, determine if there exists a direction d such that polygon P can be translated in direction d an arbitrary distance without colliding with Q. Previous results for this problem required a triangulation of the region "between" P and Q in order to compute the relative convex hull ([BT], [To2]).
- (Illumination by a convex body) Given a convex k-gon Q within simple polygon P, determine the subpolygon of P that is weakly illuminated by a light source occupied by Q. Previous results required triangulation before obtaining a bound of $O(n+k)$ [Gh]. We obtain an optimal algorithm that runs in time $\Theta(n+k)$.
- (Link reachability) Given a simple polygon P and a point s within P, determine that portion of P that is at link distance $\leq L$ from s and triangulate it. We solve this problem in time $O(Ln)$, which for fixed L is optimal. (For arbitrary L one would use the algorithm of [Su] to compute the link distance shortest path map in time $O(t(n))$.) In fact, we can produce a triangulation of the set of points of P that are at link distance at most L from a convex k-gon in time $O(Ln+k)$.

The key idea behind our improvements is the notion of a *structured visibility profile* (with respect to a fixed direction d) of a simple polygonal path π from s to t. Basically, the structured visibility profile gives a description of the (polygonal) subset of the plane consisting of all points u such that any path from s to t avoiding π that passes through u must, at some point, be directed in the fixed direction d.

We show how the structured visibility profile can be computed in linear time, and how it naturally leads to a linear-time algorithm for detecting the existence of monotone paths between s and t. We

then apply our methodology to the catalogue of problems given above, in each case showing how the structured visibility profile provides us with enough information about P to avoid having to triangulate the entire polygon P.

We should emphasize that our linear-time algorithms are relatively simple and do not require complex data structures. They should be easy to implement, and the constants in the "Big-Oh" notation are small.

The structured visibility profile permits *limited exploration* of P and a *partial* triangulation that is rich enough to answer a variety of fundamental questions. We hope too that it sheds some light on the inherent complexity of problems defined in a simple polygon.

This paper is organized as follows. Section 2 defines and describes the structured visibility profile. Section 3 gives the details of our $O(n)$ algorithm for constructing the structured visibility profile. Section 4 applies the structured visibility profile to the problem of computing monotone paths within simple polygons and proves various important properties. Section 5 gives the details of the method of partitioning the interesting portion of the simple polygon, so that a triangulation can be obtained in linear time of this portion. Section 6 describes the application of these results to the translation separability problem, Section 7 describes the application to various weak visibility problems, and Section 8 describes the application to the link distance problem. Finally, Section 9 gives a brief conclusion.

2 The Structured Visibility Profile

In this section we introduce the *structured visibility profile*, the main technique of this paper. We begin with some basic notation.

For a point p in the plane, we let x_p and y_p denote the x-coordinate and y-coordinate, respectively, of p. We say that a triple of points (p, q, r) makes a *left turn* if $\vec{pq} \times \vec{qr} \geq 0$. (We similarly can define a *right turn.*) A *chain* $c = (p_1, \ldots, p_k)$ is a polygonal path joining (in order) the *vertices* p_i with line segments (called *edges*). A polygonal chain c whose first and last vertices are $s = p_1$ and $t = p_k$ is an (s,t)-*chain.* We will denote the subchain of c between two of its points p and q by $c(p,q)$. (Thus, $c = c(p_1, p_k) = c(s,t)$ for an (s,t)-chain c.) A chain whose first and last vertices are the same point is a *polygon*; if the chain does not cross itself, then the polygon is *simple.* If p is a vertex of a chain, $p.prev$ and $p.next$ denote the vertices of the chain preceeding and succeeding p. When we speak of a polygon P, we will not distinguish between the boundary of P (the chain that defines it) and the interior of P (which is well-defined for simple polygons); the meaning should be clear from the context. Finally, we denote the convex hull of polygon P by $CH(P)$.

Let c_1 be a simple (s,t)-chain. Throughout, we will assume without loss of generality that $y_s \leq y_t$. If c_2 is another simple (s,t)-chain that intersects c_1 only at s and t, then c_1 and c_2 form a bounded simple polygon P. As we traverse c_1 from s to t, the interior of P lies either to the left or right of c_1. If the interior lies to the right as we traverse c_1 from s to t, then we say that c_1 is a *left path* of P, or more precisely c_1 is an (s,t)-*left-path-subchain* of P. This implies that c_2 is a *right path* of P, since the interior lies to the left of each directed edge of c_2. We will discuss left paths, and note that analogous definitions and theorems can be stated for right paths. Our figures will show the interior side of a chain shaded gray, or will indicate the orientation (from s to t) of the chain with an arrow drawn on the interior side of the chain.

Let c_1 be a left path from s to t. (By this we mean that c_1 is an (s,t)-left-path-subchain of some

polygon P whose (s,t)-right-path-subchain c_2 is not specified.) Thus, the "interior side" of a directed edge of c_1 lies to the right of the edge. We define a *lid* of c_1 as follows:

Definition 1 *A* lid *of a left path c_1 from s to t is a horizontal segment $\overline{pq}$ that satisfies the following conditions:*

1. *p and q lie on path c_1;*
2. *the segment $\overline{pq}$ together with $c_1(p,q)$ define a bounded simple polygonal region P', with $c_1(p,q)$ being the (p,q)-left-path-subchain of P' and $\overline{pq}$ being the right path, and where P' lies below $\overline{pq}$;*
3. *the relative interior of $\overline{pq}$ and the interior of P' do not intersect c_1.*

The polygon P' defined by lid $\overline{pq}$ (together with $c_1(p,q)$) is called an *up-pocket* of c_1. (The term "up-pocket" is to suggest that the opening of the "pocket" is directed "up".) A *maximal up-pocket* is an up-pocket that is not a proper subset of any other up-pocket. We will be concerned only with maximal up-pockets, and for simplicity will refer to them simply as up-pockets.

We call a vertex $p \neq s,t$ of c_1 a *left-peak* if $y_p > y_{p.prev}$, $y_p > y_{p.next}$, and $(p.prev, p, p.next)$ is a left turn. We call a vertex p of c_1 a *left-valley* if $y_p < y_{p.prev}$, $y_p < y_{p.next}$, and $(p.prev, p, p.next)$ is a left turn. (We define *right-peak* and *right-valley* similarly.) Note that the lid of a maximal up-pocket will always have s, t, or a left-peak as one of its endpoints. Therefore we concern ourselves only with lids for which at least one endpoint is either s, t, or a left-peak. Examples of lids, up-pockets, and left-peaks are shown in Figure 1, where we have shaded gray the "interior" side of c_1.

Assume that every left-peak of c_1 is an endpoint of a lid. In this case, we define the *horizontal paring of c_1*, $l(c_1)$, to be the polygonal chain formed by letting each maximal lid replace the subchain of c_1 between its endpoints. The chain $l(c_1)$ is well-defined by the definition of lids and the fact that we use only lids of maximal up-pockets.

Note that $l(c_1)$ contains right-peaks and left-valleys, but no left-peaks or right-valleys. In this sense, we say that $l(c_1)$ "progresses right". More formally, if we write the chain c as a union of y-monotone subchains, $c = a_1, a_2, \ldots, a_k$, then we say that c *progresses right* if the following holds: any horizontal line that intersects c must do so in such a way that if $p \in a_i$ and $q \in a_j$ are two points of intersection with p left of q, then $i \leq j$.

Let S be the ray with root s in direction $\theta = \pi$ (due left), and T be the ray with root t in direction $\theta = 0$ (due right). The rays S and T do not intersect $l(c_1)$ (because of the "progresses right" property), so the infinite polygonal chain formed by S, $l(c_1)$, and T partitions the plane into two regions. Let $L(c_1)$ be the region below this infinite chain (see Figure 2).

Associated with the region $L(c_1)$ is a binary tree, $\mathcal{T}(c_1)$, called the *tree of down-pockets*, in which each node of the tree is associated with a subregion of $L(c_1)$ and a subchain of $S \cup l(c_1) \cup T$. The root node is associated with the entire region $L(c_1)$ and the entire chain $S \cup l(c_1) \cup T$. The tree is recursively defined as follows. Consider a node ν of the tree. If the associated subchain, $c(\nu)$, has no left-valleys, then ν is a leaf. Otherwise, the node ν has children, and we let w represent the left-valley of $c(\nu)$ with least y-coordinate. Imagine "firing bullets" horizontally left and right from w, allowing each bullet to go until it hits $l(c_1)$ (or allowing it to go until infinity, if necessary). If the left bullet stops at v_l, then the left child of the node is associated with the subchain of $l(c_1)$ from v_l to w, and with the subregion

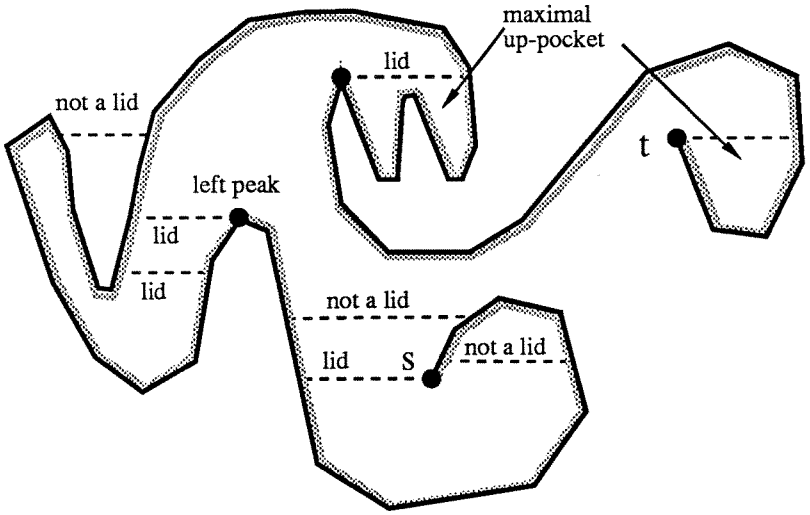


Figure 1. Examples of lids, up-pockets, and left peaks.

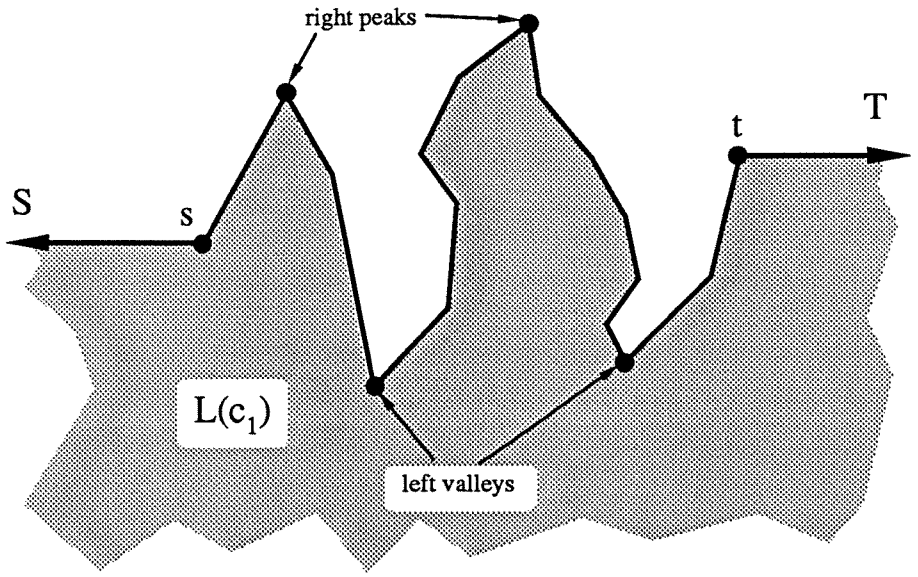


Figure 2. Definition of $L(c_1)$, S, T.

defined by its subchain plus $\overline{v_l w}$ (if v_l is at infinity, the subchain includes S). Similarly, the right child is associated with the subchain from w to v_r, the point where the bullet fired right from w hits $l(c_1)$. This procedure partitions all of $L(c_1)$ into horizontal slabs.

The regions associated with the nodes of the tree are called *down-pockets*, and the horizontal segment of the bullet path that forms the lower border of a down-pocket is called its *bottom*. A down-pocket minus its children, i.e. a horizontal slab, is called a *strict down-pocket*. Each node in the tree stores its bottom, and pointers to its parent, children, and sibling.

In Figure 3, $L(c_1)$ is shown with the tree of down-pockets represented as a directed, acyclic graph. There exists a path from point a to point b monotone with respect to the vertical up direction ($\theta = \pi/2$) if and only if there is a path in the directed graph from the strict down-pocket containing a to the strict down-pocket containing b.

We can now state the main definition of this section.

Definition 2 *Let c_1 be a polygonal path from s to t, and assume, without loss of generality, that $y_s \leq y_t$. We consider c_1 to be an (s,t)-left-path-subchain (for some polygon). Assume that every left-peak of c_1 is an endpoint of a lid (so that the chain $l(c_1)$ is defined for c_1). Then we define the* structured visibility profile of c_1 as a left path *(denoted by $svp_L(c_1)$) to be the ordered pair $(l(c_1), \mathcal{T}(c_1))$ of the chain $l(c_1)$ together with the associated tree of down-pockets. If not every left-peak of c_1 is an endpoint of a lid, then $svp_L(c_1)$ is undefined.*

The structured visibility profile of a chain *as a right path* (svp_R) is defined in a similar manner, for an (s,t)-chain with $y_s \leq y_t$ that is a right path, i.e. has "interior" to the left of each directed edge.

Structured visibility profiles derive their name from "visibility profiles". The visibility profile from direction θ of a polygonal chain c consists of all points of c visible from infinity in direction θ. Clearly the svp_L allows us to construct the visibility profile of $l(c_1)$ in the horizontal directions, $\theta = \pi$ and $\theta = 0$. The word "structured" refers to the additional information contained in $svp_L(c_1)$; namely, we can use $\mathcal{T}(c_1)$ to define a partitioning of $L(c_1)$ such that, given any query point p in $L(c_1)$, we can find the point of $l(c_1)$ visible from p to the right and to the left. Of course, in converting c_1 to $l(c_1)$, we lose visibility information about the subchains of c_1 that are deleted in making $l(c_1)$; however, we will see that this information is not necessary in our applications. (It is this loss of information that prevents $svp_L(c_1)$ from producing a full trapezoidization, which would imply a full triangulation in linear time.) Given $svp_L(c_1)$, we will be able to triangulate a down-pocket of $L(c_1)$ in linear time, since the set of all bottoms partitions the pocket into polygons each of which is monotone in the vertical direction.

3 Constructing the Structured Visibility Profile

We outline in this section a procedure ($\mathbf{SVP_L}(c)$) to compute the svp_L of a simple (s,t)-chain c in linear time. (Discussion of the procedure $\mathbf{SVP_R}$ is omitted because it is symmetric to $\mathbf{SVP_L}$.) The algorithm works in two "phases": **PhaseI** makes a first pass through the chain, removing maximal up-pockets by drawing the appropriate chords ("blue" and "red") from left-peaks. **PhaseI** either returns the answer *MustFaceLeft* (which implies that any right path from s to t that does not cross the chain c must face left), or it returns a nonsimple new chain, $\hat{l}(c)$, without up-pockets or left-peaks. If **PhaseI** returns *MustFaceLeft*, then we stop; otherwise, **PhaseII** takes the nonsimple chain $\hat{l}(c)$ and either con-

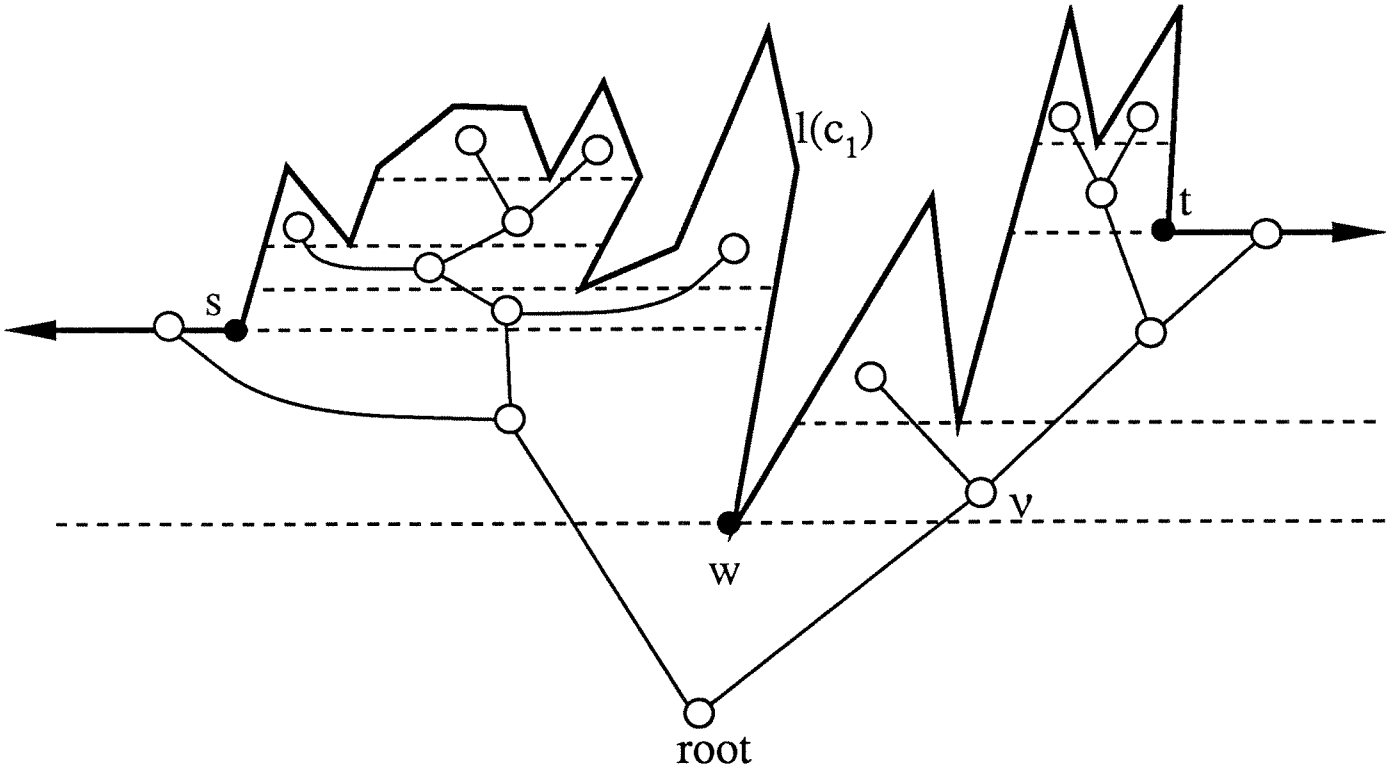


Figure 3. Tree of down-pockets.

cludes *MustFaceLeft* or produces the simple chain $l(c)$ by detecting self intersections and throwing away appropriate subchains of c. **PhaseII** also sets up the pointers that constitute the tree of down-pockets.

The global structure of $\mathbf{SVP_L}(c)$ is shown below.

Procedure $\mathbf{SVP_L}(c)$;
  Given a polygonal chain c from s to t with $y_s \leq y_t$.
  Input c to **PhaseI**:
    PhaseI either outputs chain $\hat{l}(c)$ or answers *MustFaceLeft*.
  If **PhaseI** outputs $\hat{l}(c)$, then input $\hat{l}(c)$ to **PhaseII**:
    PhaseII eithers outputs chain $l(c)$ or answers *MustFaceLeft*.
end $\mathbf{SVP_L}$.

3.1 Procedure PhaseI

PhaseI makes a single pass through chain c, calling one of two subprocedures (**Blue** or **Red**), depending on the local nature of the current point p of c:

Procedure **PhaseI**(c);
  Scan c from s to t. p will denote the current vertex of c.
  Initialize $p \leftarrow s$.
  Repeat
    $p \leftarrow p.next$;
    if p is a left-peak,
      then call **Blue(p)**;
    if p is a right valley,
      then call **Red(p)**;
  Until $(p = t.prev)$;
end **PhaseI**.

3.1.1 Procedure Blue(p)

Procedure **Blue**(p) detects lids of up-pockets whose right endpoint is a left-peak of c_1. We draw a "blue" ray, b, from p in direction $\theta = \pi$ (due left). We draw a "green" ray, g, from p in direction $\theta = 0$ (due right). Denote the subchain of c starting at p by $c(p)$. We maintain a "winding counter", w, which we initially set to 0.

Every time $c(p)$ intersects g from below, we decrement the winding counter, and every time $c(p)$ intersects g from above, we increment the winding counter. We traverse $c(p)$, keeping track of the rightmost intersection point of $c(p)$ and b (the one closest to p).

If $c(p)$ intersects b from below while the winding counter is 0, and this intersection point is closer to p than all previous intersections, then we do the following. Call the intersection point a, and let $(q.prev, q)$ be the edge of $c(p)$ on which it lies. Insert a into the chain c between $q.prev$ and q, and delete all vertices of c between p and a, exclusive. Return to the main scan of **PhaseI**, with $p \leftarrow q$. See Figure 4.

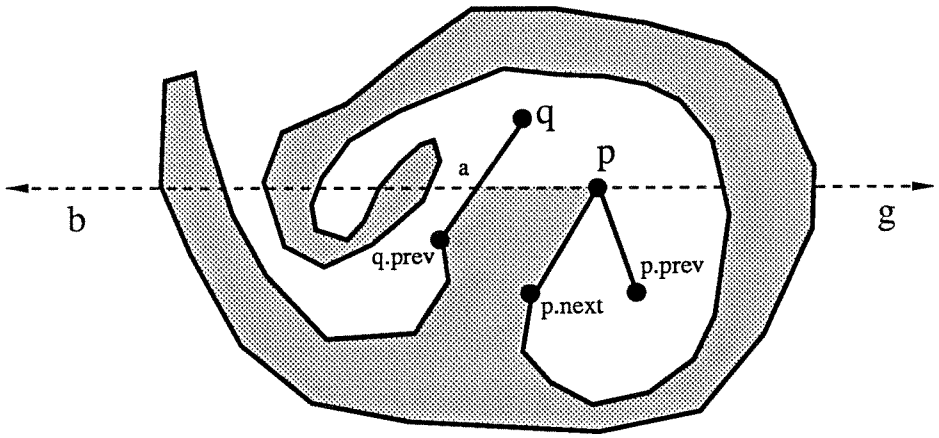


Figure 4. Procedure **Blue**.

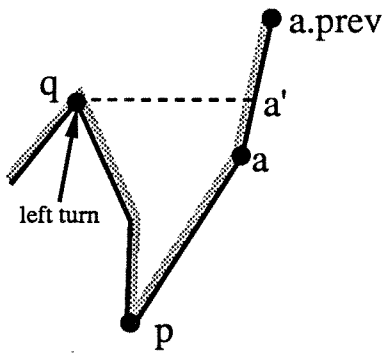


Figure 5. Procedure **Red**: insert a'.

If we traverse until t without finding a point a as described, then **STOP** the procedure and exit out of procedure $\mathbf{SVP_L}$ with the answer *MustFaceLeft*.

3.1.2 Procedure Red(p)

The procedure **Red**(p) detects lids of a second type of up-pocket: those whose left endpoint is a left-peak. It also detects the two special cases of lids: those with right endpoint s or with left endpoint t. We describe the procedure as follows.

- Traverse forward from p, keeping a winding counter, originally set to 0. Decrement the counter at each point of horizontal tangency that is a left turn (counterclockwise), and increment the counter at each point of horizontal tangency that is a right turn (clockwise).
- While traversing forward, keep a pointer on the current point, q. Also, traverse backwards from p, keeping a pointer on the point, a, where a has the same (approximate) y-coordinate as q. Occasionally we may start moving down in an attempt to update a; this is all right—just continue traversing and we will eventually reach the y-coordinate of the current q (or reach s).
- If, while not in a spiral (spirals are discussed below), the winding counter takes the value -1 at point q, or q reaches t, then find the point a' near a such that $y_{a'} = y_a$. Insert a' into the chain c, and delete all vertices between a' and q, exclusive. Set $p \leftarrow q$. Exit procedure **Red**. See Figure 5.
- If $y_p < y_s$, and if ever $y_q \geq y_s$ and the edge $(q.prev, q)$ lies to the left of s, then find q' near q such that $y_{q'} = y_s$. Insert q' into the chain between $q.prev$ and q, and delete all vertices between s and q'. Set $p \leftarrow q'$. Exit procedure **Red**. See Figure 6.
- When the winding counter takes the value $+1$ at a point q, we enter a spiral. Freeze the point a, and store the value y_q. Do not update a until the forward traversal point moves above y_q. When this happens, the winding counter is 0 and we have exited the spiral. Update q accordingly and continue the double traversal with q and a.
- If we reach t while in a spiral, re-set a to p, and repeat the backwards traversal until we reach a point a with the same y-coordinate as t. Traverse c from p to t, checking whether any edge intersects (t, a). If YES, **STOP**: procedure $\mathbf{SVP_L}$ answers *MustFaceLeft*. If NO, then insert a in c, and delete all vertices between a and t, exclusive; exit procedure **Red** (and **PhaseI**). See Figure 7.

3.2 Procedure PhaseII($\hat{l}(c)$)

We are given $\hat{l}(c)$, a polygonal chain from s to t with no "illegal" turns, i.e. all peaks are right-peaks and all valleys are left-valleys. The chain $\hat{l}(c)$ is not necessarily simple, and we will transform it into the simple chain $l(c)$. We will also build the tree of down-pockets for $l(c)$.

We will call the right-peaks *top-points*, and the left-valleys *bottom-points*. Points s and t are considered to be bottom-points.

We will traverse $\hat{l}(c)$ twice. The first traversal is from t to s, and it fixes all crossings involving red segments, i.e. horizontal segments drawn by procedure **Red** (except those with endpoint s). The

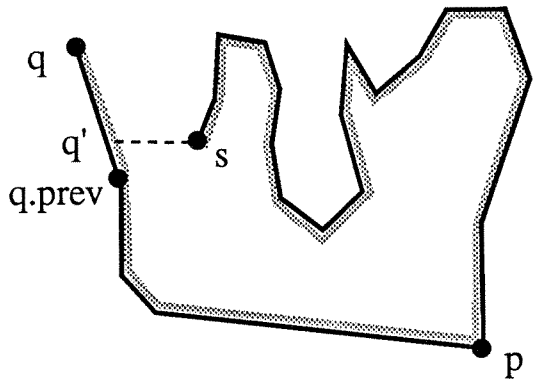


Figure 6. Procedure **Red**: insert q'.

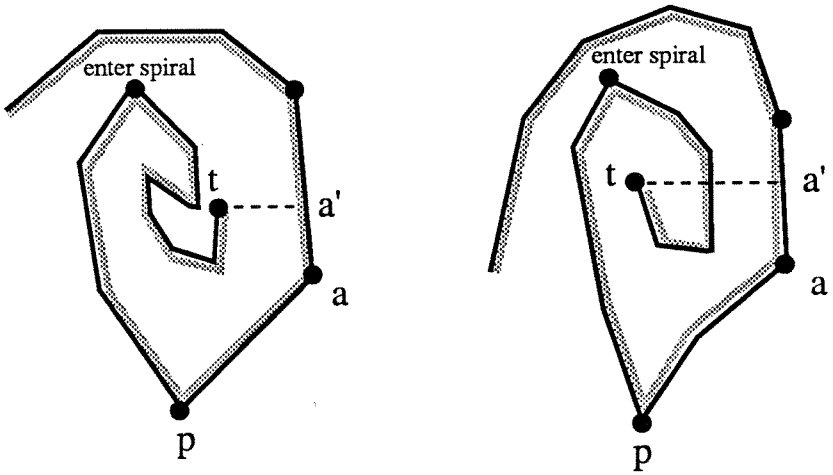


Figure 7. Procedure **Red**: reach t in a spiral.

second traversal is from s to t; it fixes all crossings involving blue segments (those with endpoint s or drawn by procedure **Blue**) and constructs the tree of down-pockets. The traversals are performed by the procedures **FixRed** and **FixBlue**.

3.2.1 Procedure FixRed($\hat{l}(c)$)

We traverse $\hat{l}(c)$ from t to s. We call a horizontal segment with left endpoint q a *red* segment if $q = t$ or if: (1) $q.prev$ is the right endpoint, and (2) $y_{q.next} < y_q < y_{(q.prev).prev}$. We keep a stack of red segments, initially empty, where e_{top} denotes the top element. For each segment e on the stack, we store $lf(e)$ and $rt(e)$, its left and right endpoints; y_e, the y-coordinate of $lf(e)$ and $rt(e)$; and $cand(e)$, the *candidate point* of e, initially $rt(e)$.

The traversal is always in one of three modes: *empty*, if the stack is empty, and either *above* or *below* if the stack is not empty, depending on whether the current traversal point is above or below e_{top}.

If the stack is empty, traverse until we find a red segment e, put e on the stack, and set $cand(e) \leftarrow rt(e)$; we are above the top segment.

If we are in the above mode, traverse until we encounter either a red segment or a point with y-coordinate of $y_{e_{top}}$. If we encounter a red segment e, we put it on the stack, and set $cand(e) \leftarrow rt(e)$; we are in the above mode.

If we encounter a point p with y-coordinate $y_{e_{top}}$, let $u = lf(e_{top})$ and consider two cases:

($x_p < x_u$) Pop the stack and make $cand(e_{top})$ the new right endpoint of the top segment, thereby deleting the subchain in between $cand(e_{top})$ and $rt(e_{top})$; we are in either the above or empty mode.

($x_p > x_u$) We are below the top segment; traverse until we reach a point p with $y_p = y_{e_{top}}$, and set $cand(e_{top}) \leftarrow p$; we are in the above mode.

Figure 8 illustrates the behavior of **FixRed**.

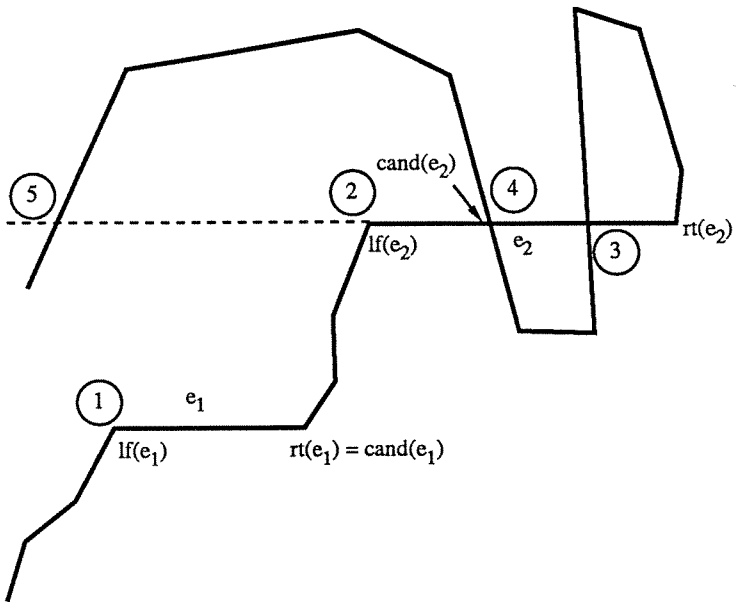
3.2.2 Procedure FixBlue

The input is the output of **FixRed**: a chain with no left-peaks or right-valleys, and with no crossings involving red segments. We will traverse the chain from s to t, keeping a pointer on the current point, p, and one on the current down-pocket, CDP. There are two starting cases. After the appropriate starting case calls procedure **Down**, **FixBlue** remains in **Down** until termination.

The starting cases are:

$y_s \leq y_{s.next}$: We initialize a down-pocket and make it CDP. Make s its left endpoint. Create a left sibling of the CDP, and make s both the left and right endpoints (this pocket is empty). Create a parent pocket for CDP and its sibling. Let p be the first top point in $\hat{l}(c)$. Call **Down**(p).

$y_s > y_{s.next}$: We traverse to the first bottom point, and call it q. Create a parent down-pocket with 2 children. The left child has lid $((-\infty, y_q), q)$, and the right child has left endpoint q. The right child is CDP. Let p be the first top point after q. Call **Down**(p).



(1) put e_1 on stack

(2) put e_2 on stack

(3) enter below mode

(4) update cand(e_2), enter above mode

(5) pop e_2 from stack

Figure 8. Behavior of **FixRed**.

3.2.3 Procedure Down(p, CDP)

The point p is the top point of what appears, locally at least, to be a down-pocket (whether or not it is a down-pocket will not be known until the end of **PhaseII**). Traversing the chain backwards from p, keeping a pointer on the current point l, gives the left side of this "down-pocket". Similarly, traversing forwards from p, keeping a pointer on the current point r, gives the right side. Perform a double-traversal down the two sides in leapfrog fashion to keep the two traversal points at the same approximate y-coordinate. Three cases can occur.

Case (1) The left side hits a bottom point, l. Refer to Figure 9.

Denote by r' the point near r on the right side such that $y_{r'} = y_l$. Make r' the right endpoint of CDP.

CDP is a right sibling. If $l \neq s$ and the left sibling of CDP has a left endpoint, l', not at infinity (Figure 9(a)), then set $l \leftarrow l'$, set CDP to the parent, and continue the double traversal.

If $l = s$ (Figure 9(b)), or if $l \neq s$ (Figure 9(c)) and the left sibling has a left endpoint at infinity, then let r' be the point on the right side near r such that $y_l = y_{r'}$, and let r' be the right endpoint of CDP. Move CDP to the parent, move r to the next bottom point, and let CDP have lid $((-\infty, y_r), r)$. Now create a parent and right sibling for CDP, and make r the left endpoint of the right sibling. Make the right sibling the new CDP. Let p be the first top point after r. Call **Down**(p, CDP).

Case (2) The right side hits a bottom point, r. Refer to Figure 10.

Create left and right children for CDP. If CDP already has children, then they become the children of the newly-created left child.

Make r the right endpoint of the left child and the left endpoint of the right child. Let l' be the point on the left side near l such that $y_{l'} = y_r$. Make l' the left endpoint of the left child.

If $r \neq t$ (Figure 10a)), then let CDP be the right child, let p be the first top point after r, and call **Down**(p, CDP).

If $r = t$ (Figure 10(b)), then $r = t$ is the right endpoint of the right child (i.e. the right child is empty). Exit **Down** and **PhaseII**.

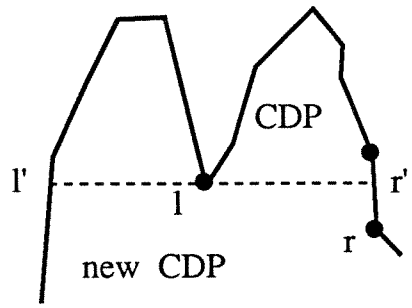
Case (3) The left side cuts horizontally right and intersects the right side. Refer to Figure 11.

Say it cuts to the right at point q; then $q.prev$ is the right endpoint of this horizontal segment. Continue to traverse forwards from r until we reach a point w (as in Figure 11(a)) such that $y_w \geq y_q$, and $(w.prev, w)$ lies to the right of the edge containing r that intersects $(q, q.prev)$. Denote by a the point on $(w.prev, w)$ that intersects $(q, q.prev)$. Insert a into the chain between $w.prev$ and w, and delete all vertices between $q.prev$ and a, exclusive. Delete any children of CDP. Let p be the first top point after a, and call **Down**(p, CDP).

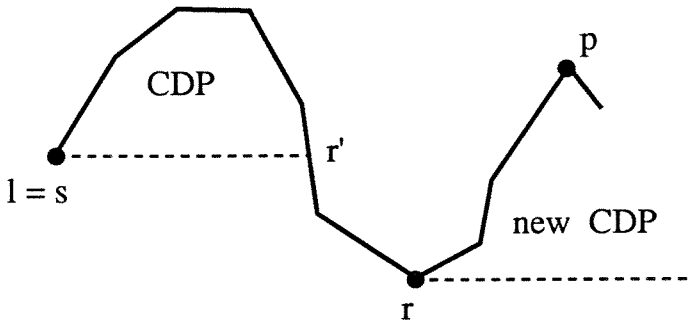
If we reach t before finding a point w as described (Figure 11(b)), **STOP**: procedure $\mathbf{SVP_L}$ answers *MustFaceLeft*.

4 Searching for Monotone Shortest Paths

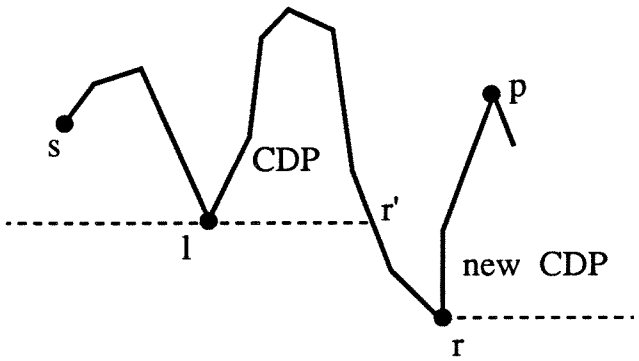
In this section the correctness and running time of procedure $\mathbf{SVP_L}$ are established, and we use structured visibility profiles to solve the following problem in linear time:



(a)



(b)



(c)

Figure 9. Procedure **Down**: Case (1).

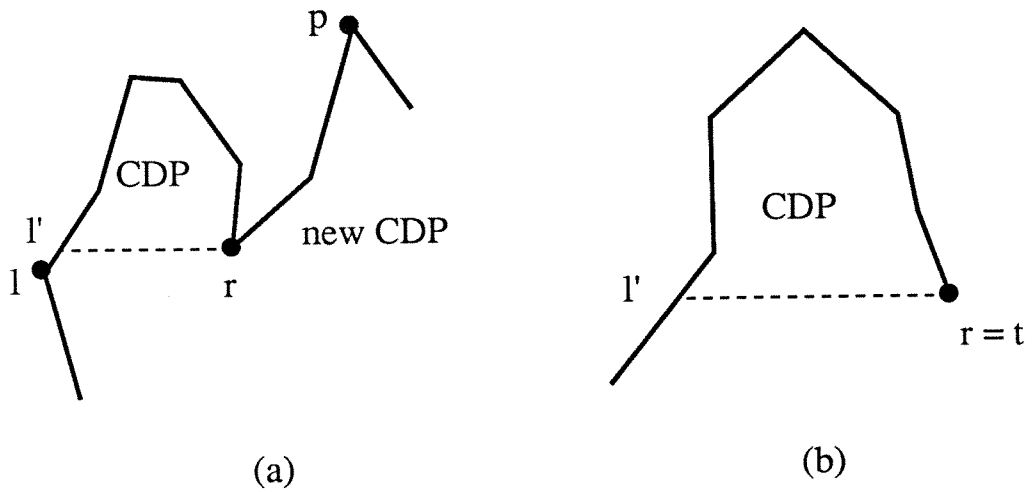


Figure 10. Procedure **Down**: Case (2).

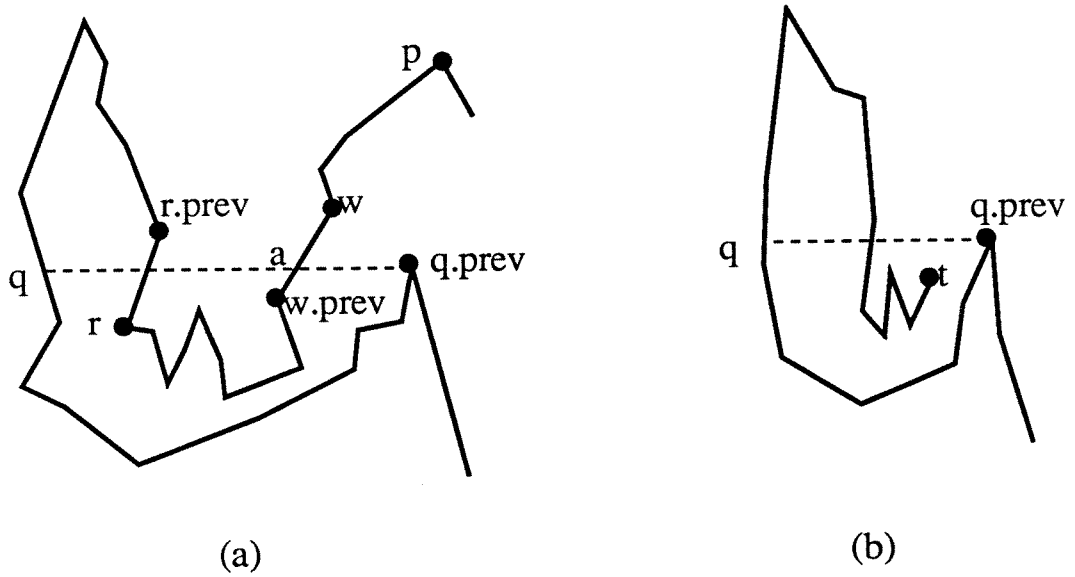


Figure 11. Procedure **Down**: Case (3).

(P1) Given two points s and t on or inside a polygon P, determine if there is a monotone (s,t)-path in P and, if so, produce one. Also produce the set of all directions for which there exists a monotone (s,t)-path. (This set is guaranteed to be a convex cone [ACM].)

Since there exists a monotone path between s and t if and only if the shortest path between s and t is monotone (see [ACM]), we will consider instead the following stronger problem:

(P2) Given two points s and t on or inside a polygon P, determine if the (unique) shortest path from s to t in P is monotone and, if so, produce it.

The set of all directions for which there exists a monotone (s,t)-path is known to be precisely the convex cone of directions for which the unique shortest path from s to t is monotone [ACM]. Since the set of directions for which a path is monotone can be found in linear time ([PrS]), solving problem (P2) in linear time implies a solution to (P1) in linear time. We can assume without loss of generality that s and t are vertices of P. If s is not a vertex but is on the boundary of P, simply insert it into the list of vertices. If s is in the interior of P, then shoot two rays from s in opposite directions, such that the rays are perpendicular to (s,t). For each ray, draw a segment from s to first point on the boundary of P hit by the ray. Together these two segments partition P into two subpolygons, one of which contains t. We can discard the subpolygon which does not contain t, since the shortest (s,t)-path will not enter it.

A polygonal path is *monotone in direction* θ if every line in direction $\theta + \pi/2$ intersects the path in a connected set (i.e., either in a point or a line segment). (If the intersection is always a single point, then we say that the path is *strictly* monotone in direction θ.) We refer the reader to [PrS] and [ACM]. Note that if the path is not simple, then it cannot be monotone in any direction; therefore we will limit our attention to simple (s,t)-paths.

We say that a path *faces left* if the headlamps of an automobile that "drives" along the path point due left at some point. (We imagine "driving" an automobile along the path, turning at vertices in such a way as to turn by less than π.) More formally, we say that a path faces left if either (1). it has an edge $\overline{pq}$ oriented in direction $\theta = \pi$ (due left), and it turns left (or right) at both p and q (as in Figures 12(a),(b)); or (2). it has a vertex p such that one of the following holds:

$(p.prev, p)$ is oriented in direction $\theta < \pi$, $(p, p.next)$ is oriented in direction $\theta > \pi$, and $(p.prev, p, p.next)$ is a left turn (Figure 12(c)); or,

$(p.prev, p)$ is oriented in direction $\theta > \pi$, $(p, p.next)$ is oriented in direction $\theta < \pi$, and $(p.prev, p, p.next)$ is a right turn (Figure 12(d)).

We define *faces right* in a similar manner. Clearly a path that both faces left and faces right cannot be monotone. This suggests that one approach to finding a monotone (s,t)-path would be to determine if there is an (s,t)-path in P that never faces left, or never faces right. Through the use of *svp*'s, we can determine this and more; namely, we can determine whether there exists an (s,t)-path that never faces left, and if there does we can triangulate in linear time a subpolygon of P that contains the shortest (s,t)-path in P.

We are given P with vertices s and t. Assume that P is oriented so that $y_s \leq y_t$. Let c_1 be the subchain obtained by traversing the boundary of P clockwise from s to t, and let c_2 be the counterclockwise (s,t)-subchain. Then c_1 is a left path and c_2 is a right path. Define c_1^R (c_2^R) to be the chain obtained by

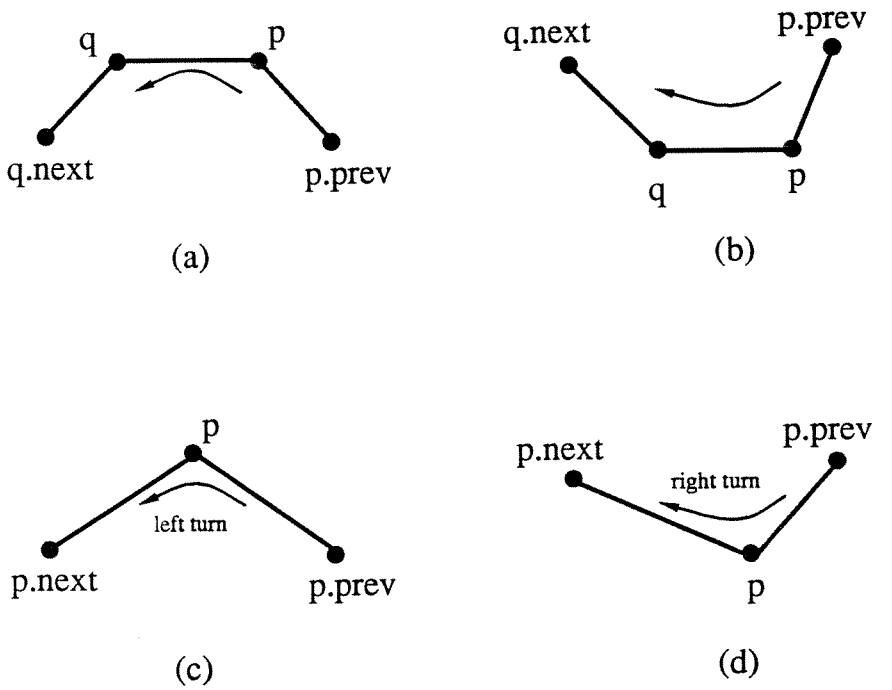


Figure 12. Definition of a path facing left.

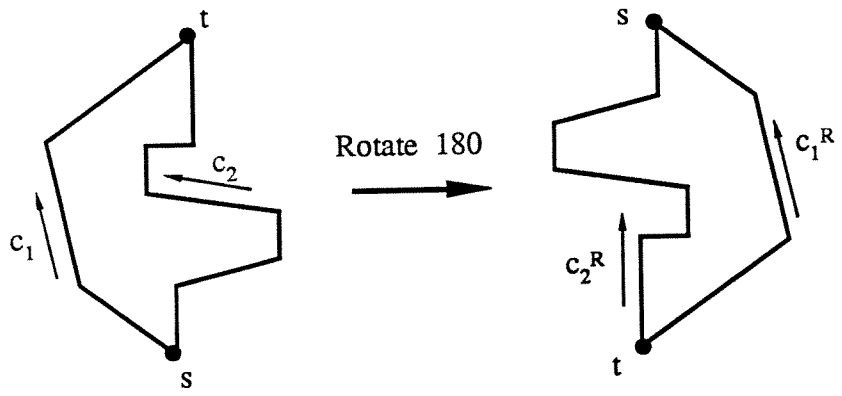


Figure 13. Paths c_1^R and c_2^R.

rotating c_1 (c_2) 180 degrees, and reversing the orientation of c_1 (c_2). Thus, c_1^R (c_2^R) is a right path (left path) from t to s, where $y_t \leq y_s$ (see Figure 13). We can, therefore, take the svp_R of c_1^R, denoted $r(c_1^R)$, and the svp_L of c_2^R, denoted $l(c_2^R)$. We will see how the two left-path-*svp*'s of P, $l(c_1)$ and $l(c_2^R)$, can be combined to determine whether the shortest path faces left, and similarly how the right-path-*svp*'s can determine whether it faces right.

The procedures $\mathbf{SVP_L}$ and $\mathbf{SVP_R}$ construct the svp_L and svp_R, respectively, of a polygonal chain. The input to $\mathbf{SVP_L}$ is c_1, a simple (s,t)-chain where $y_s \leq y_t$ and c_1 is considered to be a left-path-subchain of some polygon P. The output is $l(c_1)$ if $l(c_1)$ is defined for c_1, or *MustFaceLeft* if $l(c_1)$ is not defined. The following lemma establishes an important property of left-peaks.

Lemma 1 *If c_1 has a left-peak that is not an endpoint of a lid, then every simple (s,t)-path in P faces left somewhere.*

Proof. Consider a left-peak, p, of c_1. Let b be the ray with root p in direction $\theta = \pi$, and let g be the ray with root p in direction $\theta = 0$. Note that if a simple path crosses g and later crosses b, the path must face left somewhere. This is true by a form of the Mean Value Theorem for subgradients on continuous curves.

Assume p is not an endpoint of a lid. Then either

1. c_1 does not intersect g; or,
2. the intersection of c_1 and g that lies closest to p is a point q on $c_1(s,p)$; in this case, s lies in the polygon defined by segment (p,q) and the subchain between q and p; or,
3. the intersection of c_1 and g that lies closest to p is a point q on $c_1(p,t)$; in this case, s lies in the polygon defined by segment (p,q) and the subchain between p and q.

The above property, combined with a similar property for b, shows that in every case a simple (s,t)-path in P must face left. Some of these cases are illustrated in Figure 14. ∎

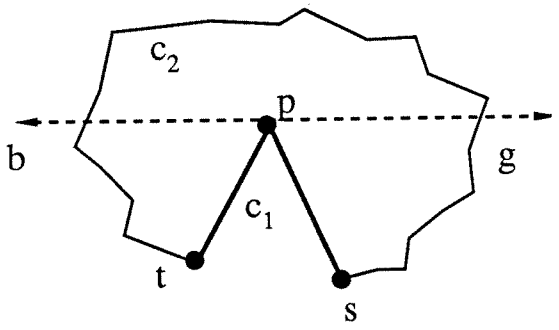
Lemma 2 *Given input c_1, $\mathbf{SVP_L}$ returns* MustFaceLeft *only if every simple (s,t)-path in P faces left.*

Proof. Suppose $\mathbf{SVP_L}(c_1)$ returns *MustFaceLeft*. It can generate this answer in one of three ways:

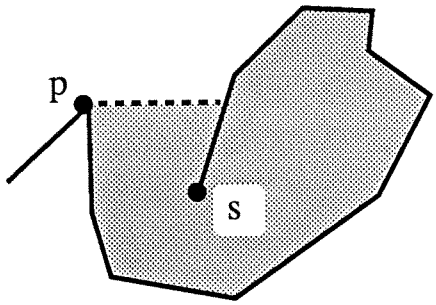
1. In procedure **Blue**(p), $\mathbf{SVP_L}(c_1)$ will return *MustFaceLeft* if the ray b is not intersected from below while the winding counter is 0 at a point that is closer to p than the previous intersection points. The point p cannot be the left endpoint of a lid, else **Blue**(p) would not have been called.

   Consider whether p is the right endpoint of a lid. Let q be the intersection point of b and c_1 closest to p. Suppose q lies on $c_1(p,t)$. If the winding counter at q is nonnegative, **Blue**(p) will draw a segment, either to q or a point to the right of q occuring earlier in $c_1(p,t)$, contradicting that **Blue**(p) returns *MustFaceLeft*. If the winding counter at q is negative, the region defined by (q,p) and the subchain between p and q and lying below (q,p) is unbounded, implying that p is not the the right endpoint of a lid (see Figure 15).

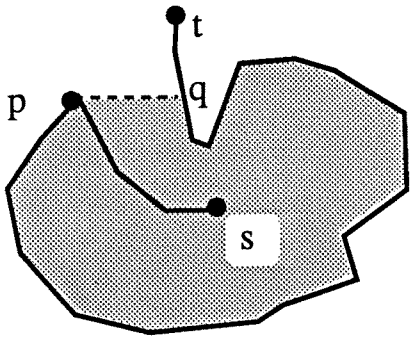
   Suppose q lies on $c_1(s,p)$. The region defined by (q,p) and the subchain between q and p either is unbounded or contains t, implying that p is not a right lid endpoint. Therefore **Blue**(p) returns *MustFaceLeft* only if p is not a lid endpoint, which by the previous lemma implies that every (s,t)-path in P faces left.



(1)

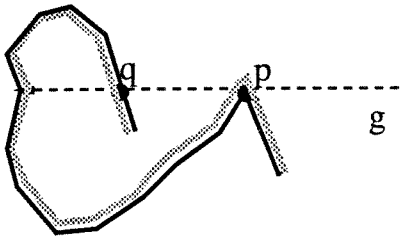


(2)

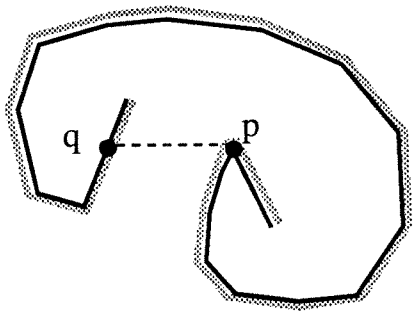


(3)

Figure 14. Proof of Lemma 1.



Winding counter at q nonnegative (0).



Winding counter at q is negative (-1).

Figure 15. Proof of Lemma 2: Case (1).

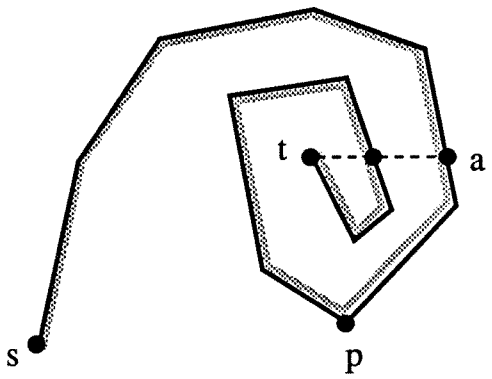


Figure 16. Proof of Lemma 2: Case (2).

2. In procedure **Red**, $\mathbf{SVP_L}$ returns *MustFaceLeft* if we are in a spiral, reach t, draw the segment (t, a), and find a point, say r, on the subchain from p to t that lies between t and a (see Figure 16). An (s,t)-path must cross (r,a) from above, since otherwise s is in the polygon formed by (r,a) and the subchain between r and a, which would imply that the red segment would have been drawn earlier when the condition ($y_q \geq y_s$, edge $(q.prev, q)$ lies left of s) was first met. We have that a simple (s,t)-path must cross a horizontal segment that lies to the right of t with the same y-coordinate as t—this implies the path faces left.

3. Procedure $\mathbf{SVP_L}(c_1)$ also returns *MustFaceLeft* in Case (3) of **PhaseII** if we reach t before crossing a horizontal segment whose right endpoint is $q.prev$. Since the horizontal segment was drawn by procedure **Blue** (unless $q.prev = s$, in which event Case (3) cannot return *MustFaceLeft*), $q.prev$ cannot be the left endpoint of a lid. If the intersection point of the horizontal segment and c_1 closest to $q.prev$ lies on $c_1(q.prev, t)$, then $q.prev$ is not a right endpoint since the polygon formed by $c_1(q.prev, a)$ and $(q.prev, a)$ contains t in its interior. If the closest intersection point lies on $c_1(s, q.prev)$, then $q.prev$ is not a right lid endpoint by arguments similar to those for procedure **Blue**. Since $q.prev$ is a left-peak, every simple (s,t)-path must face left.

■

It is not difficult to see by a case-analysis that if c_1 has a left-peak that is not a lid endpoint, $\mathbf{SVP_L}(c_1)$ will detect it by returning *MustFaceLeft*. The remaining theorems of this section assume that all left-peaks of c_1 and c_2^R are lid endpoints, implying that $l(c_1)$ and $l(c_2^R)$ are defined.

Lemma 3 *If c_1 is an (s,t)-left-path-subchain of a polygon P, procedure $\mathbf{SVP_L}$ correctly outputs $l(c_1)$ and its tree of down-pockets in $O(n)$ time, where c_1 has n vertices.*

Proof. It suffices to show that for each p and q such that $\overline{pq}$ is a maximal lid, $\mathbf{SVP_L}$ replaces $c_1(p,q)$ with $\overline{pq}$.

Suppose q is a left-peak, and **PhaseI** does not draw a red or blue segment with endpoint q. This occurs because **PhaseI** reaches q either in a call to **Blue** or a call to **Red**. If q is reached in a call to **Blue**(p), as in Figure 17(a), then any lid with q as an endpoint corresponds to a subchain that is eliminated when we exit **Blue**(p). A similar statement holds if q is reached in a call to **Red** (see Figure 17(b)). This implies that q is not an endpoint of a maximal lid.

The points s and t can be endpoints of maximal lids, and if they are, procedure **Red** draws the appropriate lid. Therefore, if any of s, t, or a left-peak q is an endpoint of a maximal lid, **PhaseI** will draw a segment with this point as an endpoint.

It is possible for **PhaseI** to draw a segment for a left-peak where the segment does not contain a maximal lid. This occurs when **Red** or **Blue** draws a segment (e_R or e_B), and later that segment is eliminated in **PhaseI** because it is backtracked by **Red** (Figure 18(a)), or is eliminated in **PhaseII** by **FixRed** (Figure 18(b)), or is eliminated in **PhaseII** by **FixBlue**, Case (3) of **Down** (Figure 18(c)). Since this segment is eventually eliminated, it does not matter that it was drawn in the first place.

We need to show that if any of s, t, or a left-peak q is an endpoint of a maximal lid, the segment produced by $\mathbf{SVP_L}$ is the maximal lid. If the relative interior of the segment drawn by **PhaseI** does not intersect c_1, then there will be no problems. Either the segment is a maximal lid, or it will be eliminated in one of the three cases discussed above (Figure 18).

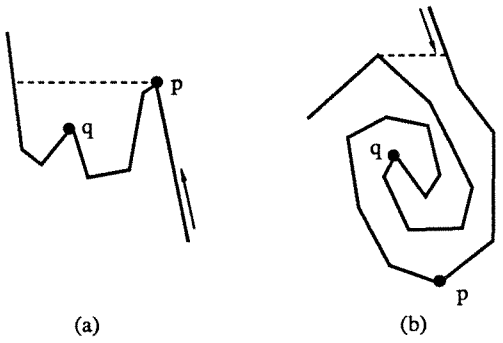


Figure 17. Left-peak q does not generate a call to **Red** or **Blue**.

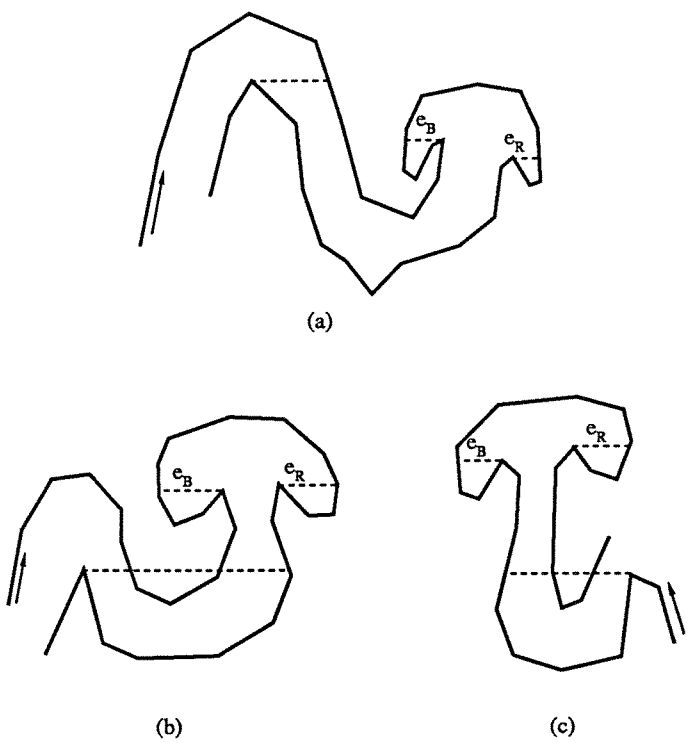


Figure 18. e_B, e_R lie on the chain eliminated in (a). **Red**, **PhaseI**; (b). **FixRed**, **PhaseII**; (c). case (3) of **Down**, **Fixblue**, **PhaseII**.

For purposes of this proof, we define the terms *red segment*, *blue segment*, *pre-chain*, and *post-chain*. A red segment is defined to be a horizontal segment with left endpoint q and right endpoint $q.prev$, where (i). $y_q < y_{(q.prev).prev}$, and (ii). either $y_q > y_{q.next}$ or $q = t$. Note that the set of red segments in $\hat{l}(c_1)$ is all (horizontal) segments of $\hat{l}(c_1)$ drawn by procedure **Red** with the exception of a segment with s as an endpoint. A blue segment is defined to be any horizontal segment drawn in **PhaseI** that is not a red segment. (These include segments drawn by **Blue** and the horizontal segment drawn by **Red** with s as an endpoint, if it exists.) An edge or vertex of a chain divides the chain into two subchains, which we call the pre-chain and post-chain.

If the segment drawn by **PhaseI** is intersected by c_1, then **PhaseII** will shorten the segment until it becomes the correct maximal lid. **PhaseII** begins with procedure **FixRed**, a backwards traversal of $\hat{l}(c_1)$ that considers all *red* segments. If a red segment e is intersected by its post-chain (as in Figure 19), the intersection will occur in a call to **Red**(p), and e will be eliminated by this call. Thus, no red segment of $\hat{l}(c_1)$ is intersected by its post-chain.

If a red segment e is intersected by its pre-chain, the intersection is detected in **FixRed**. Several important observations should be made about **FixRed**:

(1) a red segment e will be placed on the stack before **FixRed** encounters an edge that intersects e;

(2) **FixRed** will not encounter an edge intersecting e after e is popped from the stack;

(3) e is popped with its correct right endpoint; and

(4) at any time, the top element of the stack must be popped before an intersecting edge of any other element is detected.

Observation (1) holds because e is intersected only by its pre-chain. A counterexample to Observation (2) would imply the existence of a left-peak u in $\hat{l}(c_1)$, as in Figure 20, contradicting a property of $\hat{l}(c_1)$. Observation (3) follows from (2) and the fact that the left-most intersection of e and $\hat{l}(c_1)$ is returned as the right endpoint. Finally, since the elements of the stack are ordered from top to bottom by decreasing y-coordinate, (4) follows from the simplicity of c_1: a back traversal of c_1 must cross the gray line shown in Figure 21 before it can cross elements (e_1 or e_2) below the top element of the stack (e_3), since there can be no left peaks in $\hat{l}(c_1)$. These observations guarantee the correctness of **FixRed**. They also establish its linear run-time, since each red segment is put on and popped from the stack exactly once, and the number of red segments is bounded by the number of vertices of $\hat{l}(c_1)$.

FixRed outputs a chain in which the only crossings involve blue segments. **FixBlue** takes this chain as input and outputs the final chain $l(c_1)$, and its correctness follows from arguments similar to those for **FixRed**. While we do not explicitly maintain a stack, the double traversals of procedure **Down** essentially perform the task of detecting intersections until a blue segment is "popped", i.e. until it is backtracked by **Down** without being intersected by the right side of the down-pocket. We are able to maintain left visibility in the double traversals because no red segments are involved in intersections (note that this was not true in **FixRed**: since blue segments could have been involved in intersections, we would not have been able to maintain right visibility, and therefore were required to use a stack). Maintaining left visibility also allows the construction of the tree of down-pockets. **FixBlue** runs in time linear in the size of the input chain, since only a blue segment can be backtracked more than once–all

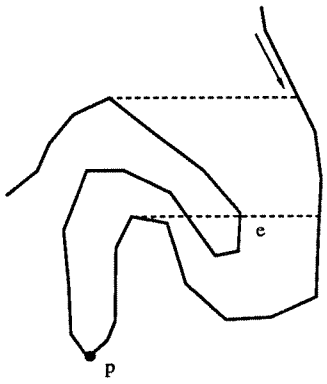


Figure 19. If e is intersected by its post-chain, e is eliminated in $\mathbf{Red}(p)$.

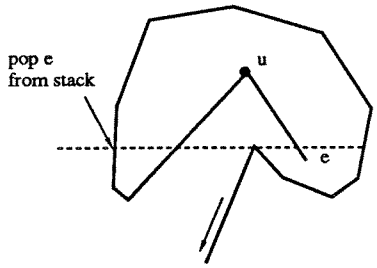


Figure 20. A counterexample to Observation (2).

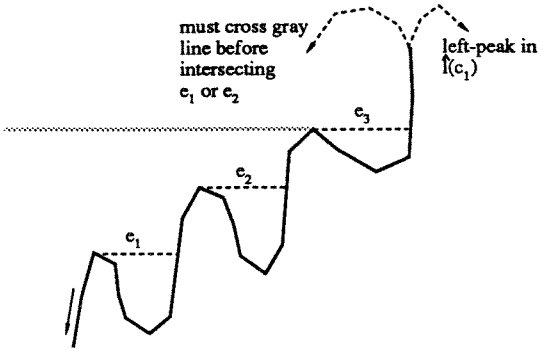


Figure 21. Observation (4).

but one of these events corresponds to detection of an intersection, so the work can be charged to an eliminated vertex.

The entire procedure $\mathbf{SVP_L}$ is $O(n)$ for c_1 with n vertices. **PhaseI** is clearly $O(n)$, and it returns chain $\hat{l}(c_1)$ of size $O(n)$. We have seen that both **FixRed** and **FixBlue** run in time proportional to the size of the input chain and output chains of size proportional to the input. Therefore $\mathbf{SVP_L}$ correctly outputs $l(c_1)$ in $O(n)$ time. ∎

Lemma 4 *If c_1 is an (s,t)-left-path-subchain of a polygon P, where $y_s \leq y_t$, then any simple (s,t)-path in P that enters the interior of an up-pocket of c_1 faces left somewhere.*

Proof. Since s and t do not lie in up-pockets of c_1, if π is a simple (s,t)-path in P that enters an up-pocket of c_1, then π must first enter the pocket at some point, p, and last exit the pocket at some point, q. The first-entry point cannot be t, since this would imply that π is not simple.

Denote the right endpoint of the up-pocket lid by u. Form a subpolygon Q of P consisting of $\overline{pu}$, $c_1(s,u)$, and $\pi(s,p)$ (see Figure 22). We must have that $t \neq u$, since t cannot be a right endpoint of a lid, and $t \neq p$, as mentioned above. By simplicity, $t \notin \pi(s,p)$, and clearly $t \notin c_1(s,u)$. Also, $t \notin \overline{pu}$ and t is not in the interior of Q. Since t is not on or in Q, and π is simple, π must last exit the up-pocket at some point q to the left of p in order to reach t. Therefore, π faces left somewhere between p and q. ∎

Suppose we take c_2^R, by rotating c_2 180 degrees and reversing its orientation. Then c_2^R can be input to $\mathbf{SVP_L}$ to obtain $l(c_2^R)$. Now rotate $l(c_2^R)$ 180 degrees and reverse its orientation, so that we once again have an (s,t)-chain with $y_s < y_t$. Rotating 180 degrees and reversing the orientation each result in switching the words "left" and "right" in the above lemma. Therefore, for $l(c_2^R)$ in this new form, the above lemma says that entering an up-pocket of $l(c_2^R)$ implies an (s,t)-path in P faces left. From now on we refer to this new form of $l(c_2^R)$, so that $l(c_1)$ and $l(c_2^R)$ are (s,t)-chains, with s and t the same for both chains. We will define P_l as the polygon formed by concatenating $l(c_1)$ and $l(c_2^R)$ (see Figure 23).

Lemma 5 *Given polygon P with (s,t)-subchains c_1 and c_2, $y_s \leq y_t$. The following are equivalent:*

(a) *Every simple (s,t)-path in P faces left.*

(b) *$l(c_1)$ and $l(c_2^R)$ intersect at some point other than s and t.*

(c) *An up-pocket of c_1 is intersected by c_2, or vice versa.*

Proof.

(a) ⇒ (b) Define P_l as the polygon formed by concatenating $l(c_1)$ and $l(c_2^R)$. By the definition of lids, each change between c_1 and $l(c_1)$, or between c_2^R and $l(c_2^R)$, is of the following type: a region which we call an up-pocket is moved from the right side of the path to the left side, i.e. from possibly being in the interior of P to definitely being in the exterior. Therefore P_l is a subpolygon of P. If $l(c_1)$ and $l(c_2^R)$ do not cross, then P_l is a simple polygon, and $l(c_1)$ is an (s,t)-path that does not face left (see Figure 24).

(b) ⇒ (c) Suppose that $l(c_1)$ and $l(c_2^R)$ intersect. Since the original chains c_1 and c_2 do not intersect, the svp_L's can only cross when an added horizontal segment is involved. Suppose without loss of generality that the horizontal segment is from $l(c_1)$. The segment it crosses cannot be horizontal and therefore must be an edge of c_2. Since the horizontal segment is the lid of an up-pocket, c_2 intersects an up-pocket of c_1.

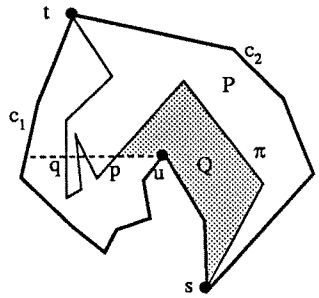


Figure 22. Proof of Lemma 4.

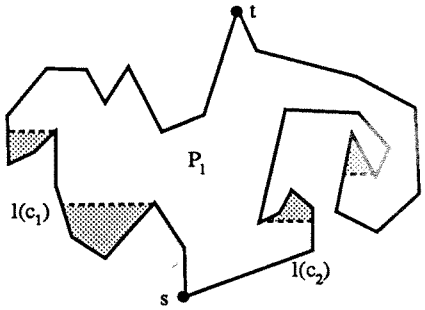


Figure 23. Definition of polygon P_l.

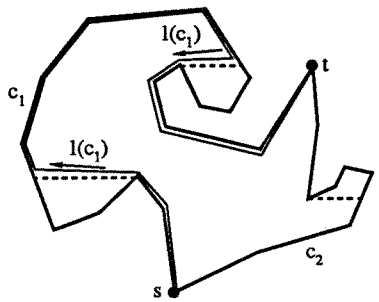


Figure 24. Proof of Lemma 5.

(c) ⇒ (a) Suppose we have a simple (s,t)-path, c, that never faces left. The path partitions P into two subpolygons, one containing c_1 and one containing c_2. If c_2, say, is intersected by a left-path-up-pocket of c_1, c also intersects the up-pocket, and by Lemma 4 faces left, a contradiction. Therefore no up-pockets are intersected. ∎

Corollary 6 *If $l(c_1)$ and $l(c_2^R)$ intersect, then the shortest (s,t)-path faces left.*

Theorem 7 *If $l(c_1)$ and $l(c_2^R)$ do not intersect, then the shortest (s,t)-path does not intersect any up-pockets. In other words, the shortest (s,t)-path in P lies in P_l.*

Proof. Because s and t are not in up-pockets, any (s,t)-path c that intersects an up-pocket must first enter the pocket at some point f and last exit the pocket at some point l. By replacing the portion of c between f and l with the segment (f,l), the path becomes shorter. Since $l(c_1)$ and $l(c_2^R)$ do not intersect, the up-pocket is not intersected, so the path remains in P. Therefore the shortest (s,t)-path does not enter any up-pockets. ∎

By the above fact, if in linear time we determine that $l(c_1)$ and $l(c_2^R)$ do not intersect and compute a triangulation of P_l, then we can find the shortest (s,t)-path in P in linear time (e.g., by [GHLST] or [LP]).

Since we have already seen that procedure $\mathbf{SVP_L}$ computes $l(c_1)$ and $l(c_2^R)$ in linear time, we need a linear time procedure, called **Partition**, which determines whether $l(c_1)$ and $l(c_2^R)$ intersect, and triangulates P_l if they do not. Since all the results proved above for left paths have analogous results for right paths, we would like **Partition** to determine if $r(c_1^R)$ and $r(c_2)$ intersect, and triangulate P_r if they do not. Assuming that we have such a procedure, we can describe the entire shortest path algorithm:

Call $\mathbf{SVP_L}(c_1)$ and $\mathbf{SVP_L}(c_2^R)$.
  If either procedure aborts, then return *MustFaceLeft*
  Else Call **Partition**$(l(c_1), l(c_2^R))$
    If **Partition**$(l(c_1), l(c_2^R))$ aborts (i.e. finds an intersection), then return *MustFaceLeft*
    Else **Partition**$(l(c_1), l(c_2^R))$ returns a triangulation of P_l
      Since P_l contains the shortest (s,t)-path in P,
      we apply the method of [LP] to the triangulated polygon P_l
      to find the shortest (s,t)-path in P.
  If *MustFaceLeft* is returned, then call $\mathbf{SVP_R}(c_1^R)$ and $\mathbf{SVP_R}(c_2)$; as above, we either find a triangulation of P_r where P_r contains the shortest (s,t)-path in P, or *MustFaceRight* is returned.
  If both *MustFaceLeft* and *MustFaceRight* are returned, then the shortest path is not monotone.
(end algorithm)

We now give a brief discussion of how **Partition** works. Details are given in the next section. Suppose we have $l(c_1)$ and $l(c_2^R)$; we wish to determine whether they intersect, and triangulate P_l if they do not. Let S be the ray with root s in direction $\theta = \pi$ (due left), and T be the ray with root t in direction $\theta = 0$ (due right). The rays S and T do not intersect $l(c_1)$ and $l(c_2^R)$ except at s and t, because $l(c_1)$ and $l(c_2^R)$ "progress right" as we traverse them from s to t. The region $L(c_1)$ is the area below the infinite chain formed by S, $l(c_1)$, and T, and $L(c_2)$ is the region above the chain of S, $l(c_2^R)$, and T. We have $P_l = L(c_1) \cap L(c_2^R)$.

Procedure **Partition** traverses $l(c_1)$ and $l(c_2^R)$ simultaneously from s to t, maintaining visibility between the current points of the scans. Because the procedure keeps the current points at the same approximate y-coordinate, it knows when a current point enters or leaves a strict down-pocket of the other chain. Because horizontal visibility is maintained and $l(c_1)$ and $l(c_2^R)$ have the "progresses right" property, an intersection of $l(c_1)$ and $l(c_2^R)$ will be detected during the simultaneous scan.

If $l(c_1)$ and $l(c_2^R)$ do not intersect, **Partition** returns a partitioning of P_l, where each polygon of the partitioning is *near-monotone*:

Definition 3 *Given a polygon A, and two vertices a and b. The points a and b partition A into two (a,b)-paths, l and r. Let l^* be the visibility profile of l from the right, and r^* the visibility profile of r from the left. If l^* and r^* do not intersect except at a and b, then A is* near-monotone for vertices a and b. *We call the polygon, A^*, formed by l^* and r^* the* monotone core *of A.*

The monotone core of a near-monotone polygon is a monotone polygon with respect to the vertical direction, but a near-monotone polygon is not necessarily monotone (see Figure 25). **Partition** produces only near-monotone polygons. We show this for a polygon generated by the **Moving-Up** subprocedure (see Figure 25). In **Moving-Up**, the point q is the current point of the traversal of $l(c_2^R)$, and p is the current point for $l(c_1)$. As we update p and q, we are traversing the left and right paths of A^*, the monotone core of the polygon being produced by this call to **Moving-Up**. Since we keep p to the left of q, A^* is simple; thus A is near-monotone. Similarly, **Moving-Down** produces only near-monotone polygons.

The maximal connected regions of $A \setminus A^*$ are down-pockets of either $l(c_1)$ or $l(c_2^R)$, since they are separated from A^* by their bottoms. Furthermore, a down-pocket can be triangulated in time linear in the number of vertices. This is true because each strict down-pocket is a polygon monotone in the vertical direction. A partition into strict down-pockets can be performed in time proportional to the size of the tree, by a single traversal of the tree, and clearly a tree cannot have more nodes than the root down-pocket has vertices in its chain. Therefore, since visibility profiles can be computed in linear time ([EA],[Le],[JS]), each polygon A produced by **Partition** can be partitioned into monotone polygons in linear time; this implies that A can be triangulated in linear time ([GJPT]).

Since the simultaneous scans are linear in the number of vertices of $l(c_1)$ and $l(c_2^R)$, **Partition** in linear time detects an intersection of $l(c_1)$ and $l(c_2^R)$ if one exists, and triangulates P_l otherwise. Both the simultaneous scans and the triangulation of the monotone polygons can be performed without complex data structures, and with small constants in the "Big-Oh" notation.

5 Details of Partition

The procedure **Partition** takes as input the structured visibility profiles of c_1 and c_2 as either left paths or right paths. We will describe the procedure for left paths; the case for right paths is similar.

The input is $svp_L(c_1) = (l(c_1), \mathcal{T}(c_1))$ and $svp_L(c_2^R) = (l(c_2^R), \mathcal{T}(c_2))$, the left-path structured visibility profiles of c_1 and c_2, respectively. The paths $l(c_1)$ and $l(c_2^R)$ are (s,t)-paths, and $y_s \leq y_t$. The procedure traverses $l(c_1)$ and $l(c_2^R)$ from s to t using pointers p and q, respectively. The pointer CPl points to the node of the strict down-pocket of $l(c_1)$ that contains p. Similarly, CPr is the strict down-pocket containing q.

The algorithm will find in order the points of horizontal tangency on the shortest (s,t)-path: $z_1,\ldots,z_{m-1}$. This gives a list $s = z_0, z_1, \ldots, z_{m-1}, z_m = t$. For each z_i, $i = 1,\ldots,m-1$, a horizontal segment in P_l is drawn from z_i to a point on the other path, and z_i is always the left endpoint of the segment. This partitions P_l into polygons $P^1,\ldots,P^m$. The procedure answers *MustFaceLeft* if it finds an intersection of $l(c_1)$ and $l(c_2^R)$ other than at s or t.

Since $l(c_1)$ and $l(c_2^R)$ have down-pockets but no up-pockets, we will simply refer to "pockets" in the algorithm. When we say "left endpoint" ("right endpoint") of a pocket, we mean the endpoint of the lid with lesser (greater) x-coordinate when the path is in its normal orientation, i.e. with $y_s \leq y_t$.

Let v_1 and v_2 point to the successors of s in chains $l(c_1)$ and $l(c_2^R)$, respectively. There are three initial cases:

1. $y_{v_1}, y_{v_2} \geq y_s$. Then s is the left endpoint of a pocket of $l(c_1)$. Set $p \leftarrow s, q \leftarrow s$, and initialize CPl and CPr accordingly. Call procedure **Moving-Up**, where a is initialized to be the right endpoint of CPl.

2. $y_{v_1}, y_{v_2} < y_s$. Then s is the left endpoint of a pocket of $l(c_2^R)$. Set $p \leftarrow s, q \leftarrow s$, and initialize CPl and CPr accordingly. Call procedure **Moving-Down**, where b is initialized to be the right endpoint of CPr.

3. $y_{v_1} \geq y_s, y_{v_2} < y_s$. Then s is the left endpoint of non-empty pockets of $l(c_1)$ and $l(c_2^R)$. If the pocket of $l(c_2^R)$ has a shorter lid than the pocket of $l(c_1)$, then call **Moving-Up**, as in **(1)** above. Otherwise call **Moving-Down**, as in **(2)** above.

Note that $y_{v_1} < y_s, y_{v_2} \geq y_s$ cannot occur.

5.1 Procedure Moving-Up

We start with z_{i-1}. We will find z_i, and draw a segment through z_i to form P^i.

We have p lying to the left of q, where p is the left endpoint of CPl. We will use another pointer, a, which is initialized to the right endpoint of CPl.

The general step of **Moving-Up** is to consider $q.next$:

Case (1) $y_{q.next} > y_q$.

We update p and a. Traverse forward from p up the left side of CPl until reaching a point p' above $q.next$. Similarly, traverse backwards from a, up the right side of CPl, until reaching a point a' above q. If during the traversal up the left side, p hits the left endpoint of the left child of CPl (equivalently, if during the traversal up the right side, a hits the right endpoint of the right endpoint of the right child), then determine whether $(q, q.next)$ intersects the bottom of the left or of the right child and update CPl and p or a accordingly. Once we reach points p' and a', we update $p \leftarrow p'$ and $a \leftarrow a'$.

When the traversals end, we have new points p and a, such that $y_{q.next} \leq y_p, y_a$. Set $q \leftarrow q.next$, and return to the general step of **Moving-Up**.

Case (2) $y_{q.next} = y_q$.

This means that $l(c_2^R)$ jumps left, i.e. $x_{q.next} < x_q$. Check whether $(q, q.next)$ intersects $(p, p.prev)$. If YES, **STOP**: procedure **Partition** answers *MustFaceLeft*, since $l(c_1)$ and $l(c_2^R)$ intersect. If NO, $q \leftarrow q.next$; return to the general step of **Moving-Up**.

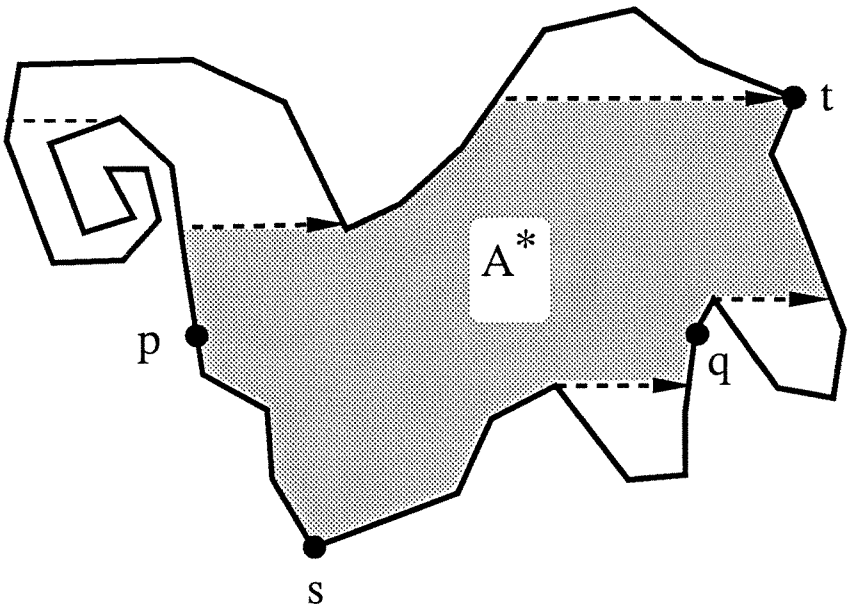


Figure 25. Definition of monotone core.

Case (3) $y_{q.next} < y_q$.

Then q is a right-peak. The point q is the right endpoint of CPr, and the left endpoint of the sibling of CPr. Let b be the right endpoint of the sibling (if b is at infinity, then replace segment (q, b) below with ray (q, b)).

Determine whether (q, b) intersects $(a, a.next)$. If NO, then $q \leftarrow b$; update CPr to be the parent, and return to the general step of **Moving-Up**.

If YES, then call the intersection point d. Set $z_i \leftarrow q$, and add the segment (q, d) to the partition of P_l to form P^i. Switch CPr to the sibling. Call procedure **Moving-Down**, initializing $p \leftarrow d, q \leftarrow q, b \leftarrow b$.

5.2 Procedure Moving-Down

We have q lying to the left of p, where q is the left endpoint of CPr. We will use another pointer, b, initialized to be the right endpoint of CPr.

The general step of **Moving-Down** is to consider $p.next$:

Case (1) $y_{p.next} < y_p$.

Then scan to update q and b, updating CPr if necessary, in a manner similar to Case (1) of **Moving-Up**. Set $p \leftarrow p.next$, and return to the general step of **Moving-Down**.

Case (2) $y_{p.next} = y_p$.

Then $l(c_1)$ jumps left, i.e. $x_{p.next} < x_p$. Check whether $(p, p.next)$ intersects $(q, q.prev)$. If YES, **STOP**: procedure **Partition** answers *MustFaceLeft*, since $l(c_1)$ and $l(c_2^R)$ intersect. If NO, then set $p \leftarrow p.next$, and return to the general step of **Moving-Down**.

Case (3) $y_{p.next} > y_p$.

Then p is the right endpoint of CPl, and also the left endpoint of the sibling of CPl. Let a be the right endpoint of the sibling of CPl (if a is at infinity, then replace segment (p, a) below with ray (p, a)). Determine whether (p, a) intersects $(b, b.next)$. If NO, then update CPl to the parent, set $p \leftarrow a$, and return to the general step of **Moving-Down**.

If YES, then call the intersection point d. Switch CPl to the sibling, and call procedure **Moving-Up**, initializing $q \leftarrow d, p \leftarrow p, a \leftarrow a$.

6 Translation Separability

We say that two polygons P and Q are *movably separable* if one of them can be moved arbitrarily far from the other without colliding with the other one. The translation separability problem asks for a direction (or the set of directions) for which Q can be translated away from P without collision:

(P3) Given two disjoint simple polygons P and Q, determine the cone of all directions in which Q can be translated arbitrarily far without colliding with P.

[BT] and [To2] give $O(t(n) + n)$ time algorithms for solving (P3). We obtain a $\Theta(n)$ time algorithm for (P3), by reducing it to problem (P2), which was solved in the previous section.

We define a *separator* of Q and P as an infinite, directed polygonal chain such that Q lies in the region to the right of the chain and P lies in the region to the left. We will use the following lemmas:

Lemma 8 *Polygon Q can be moved arbitrarily far in direction θ without colliding with P* iff *there exists a separator of Q and P monotone in direction $\theta + \pi/2$.*

Proof. This lemma follows from Theorem 7 of [To4]. ∎

Lemma 9 *If a path from s to t inside polygon Z is monotone in direction θ, then the shortest path from s to t in Z is monotone in direction θ.*

Take $CH(P \cup Q)$, the convex hull of $P \cup Q$. This can be done in $O(n)$ time ([AT],[GY],[Me]). Define a *bridge* to be an edge of $CH(P \cup Q)$ with one endpoint a vertex of P and the other a vertex of Q. Define Z to be the unique connected component of $CH(P \cup Q) \setminus (P \cup Q)$ that borders both P and Q (and hence will border any bridges). $CH(P \cup Q)$ has either 0 or 2 bridges (see [BT]), giving us two cases to consider.

6.1 $CH(P \cup Q)$ has 2 bridges

If there are two bridges, then label the endpoints of the bridges p_T, q_T, p_B, and q_B, as pictured in Figure 26. The two bridges, the subchain of P obtained by traversing from p_T to p_B clockwise, and the subchain of Q obtained by traversing from q_B to q_T clockwise, form a polygon which we call P. There are four ways to pair an endpoint of the top bridge, T, with an endpoint of the bottom bridge, B. In the next lemma we consider shortest paths in P for these pairs.

Lemma 10 *There exists a separator of Q and P monotone in direction θ* iff *the shortest path in Z from some endpoint of B to some endpoint of T is monotone in direction θ.*

Proof. Clearly a shortest path in P between points on different bridges can be extended to a separator of Q and P such that the separator is monotone in all directions for which the shortest path is monotone.

Conversely, suppose we have a separator of Q and P that is monotone in direction θ. The separator intersects T at some point, v. Replace the remainder of the separator with either the segment (v, p_T) or (v, q_T), so that the new chain is still monotone with respect to θ. Perform a similar operation at B, to obtain a path from an endpoint of B to an endpoint of T that is monotone in direction θ. By Lemma 9, the shortest path between these points is also monotone in θ. ∎

6.2 $CH(P \cup Q)$ has zero bridges

If there are no bridges, then one polygon is completely contained in a *pocket of concavity* of the other, so we will assume without loss of generality that $Q \subset CH(P)$. Denote the endpoints of the pocket containing Q by a and b. Find a point q on Q and a point p on (a, b) such that q and p are visible or determine that no such point exists. (This can be done in linear time as follows: take $q' \in Q$, determine the cone of directions in which q' can see infinitely far, and find $q \in Q$ in that cone that is closest to the lid.) By inserting the edge (q, p) twice, we obtain the *weakly-simple* polygon Z formed as follows: traverse the pocket of P from b to a, add (a, p) and (p, q), traverse Q counterclockwise starting from q, add (q, p) and (p, b). (see Figure 27).

Lemma 11 *There exists a separator of Q and P monotone in direction θ* iff *the shortest path from b to a in Z is monotone in direction θ.*

Proof. If the shortest path from b to a in Z is monotone in direction θ, it can be extended to a separator of Q and P monotone in θ by simply extending the first and last edges.

Conversely, suppose we have a separator of Q and P monotone in θ. It must intersect (b, p) at some point, which we will call b', and must intersect (p, a) at some point, a'. Replace the beginning of the separator until a' with the directed segment (a, a'), and replace everything after b' with (b', b). This gives us an (b, a)-path in Z. The only turns in this path not found in the separator are the ones at b' and a', but in order to travel from b' to a' in Z a path must sweep through these ranges of angles. Therefore the (b, a)-path we have constructed is monotone in every direction that the separator is monotone. By Lemma 9, this implies that the shortest (b, a)-path is monotone in θ. ∎

From these lemmas we see how to answer the separation problem in linear time. We construct $CH(P \cup Q)$, and determine whether it has two bridges or zero bridges. If it has zero bridges, we construct Z as described and find the shortest (b, a)-path in Z. The set of directions in which the path is monotone gives (by adding $\pi/2$) the set of directions in which Q can be translated arbitrarily far. If $CH(P \cup Q)$ has two bridges, we construct Z and find 4 shortest paths. The union of the directions in which these paths are monotone gives us the set of directions for translating Q.

7 Weak Visibility Problems

Our method can be applied to yield linear-time algorithms for a variety of weak visibility problems. We begin by solving the following problem:

(P4) Given a simple polygon P and a line segment e within P, find the subset of P that is illuminated if every point of e is a light source that casts light in every direction and the boundary of P is opaque.

We call the subset of P illuminated by e the *weak edge visible region* of P with respect to e, and we denote it by $WV(P, e)$. More precisely, $WV(P, e)$ is (the closure of) the set of all points w in P such that there exists an x in the interior of e such that $(x, w) \subset P$.

Weak visibility from a line segment has been the subject of several papers. [El], [LL], and [CG] found algorithms to compute $WV(P, e)$ in time $O(n \log n)$. More recently, [GHLST] and [To1] have obtained a bound of $O(t(n) + n)$ based on triangulating P. Also, [AT] show that in linear time one can check if all of P is illuminated by e (i.e., if $P = WV(P, e)$). Until now, it has been open to find a linear-time algorithm for computing $WV(P, e)$.

Note that, without loss of generality, we can assume that e is an edge on the boundary of P (otherwise, in linear time we can extend e until it hits the boundary of P and cut open P along this chord). Orient P so that e is horizontal, and the interior of P lies above e. Denote by s the left endpoint of e, and t the right endpoint of e. Let c_1 be the subchain of P from s to t obtained by traversing clockwise; therefore, it is an (s, t)-left-path-subchain of P and contains every edge of P except e.

Our algorithm begins by calling procedure $\mathbf{SVP_L}(c_1)$ to obtain $l(c_1)$. We claim that $\mathbf{SVP_L}$ cannot return *MustFaceLeft* on input c_1.

Lemma 12 *Assume that $e = \overline{st}$ is a horizontal edge on the boundary of P. Procedure* $\mathbf{SVP_L}$ *cannot return* MustFaceLeft *on c_1, the (s, t)-left-path-subchain of a polygon P.*

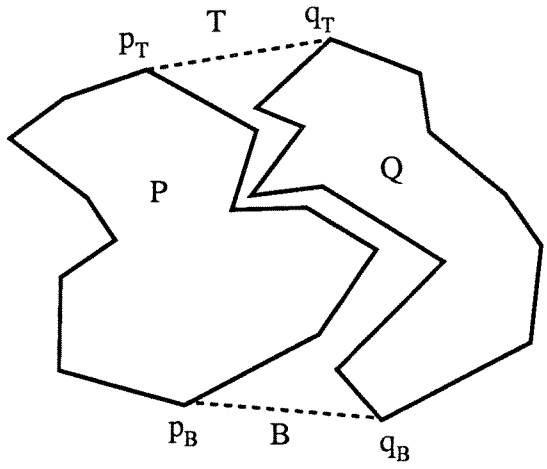


Figure 26. $CH(P \cup Q)$ has 2 bridges.

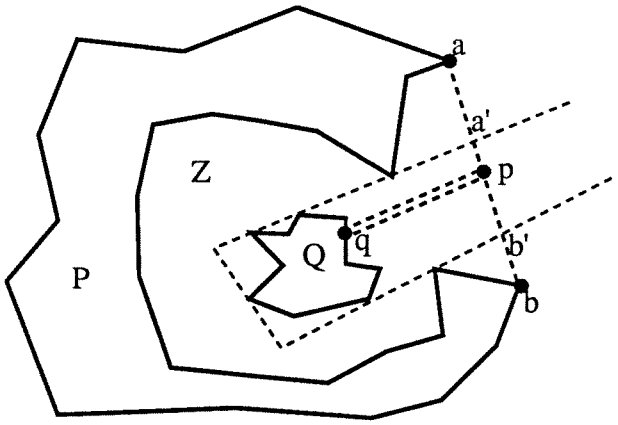


Figure 27. $CH(P \cup Q)$ has 0 bridges.

Proof. By Lemma 4, if $\mathbf{SVP_L}(c_1)$ returns *MustFaceLeft*, where c_1 is an (s,t)-left-path-subchain of a polygon P, then every (s,t)-path in P must face left. But in our case we have an obvious path from s to t that does not face left: just go from s to t along e. Therefore $\mathbf{SVP_L}$ cannot return *MustFaceLeft*. ∎

Denote by P_l the polygon formed by $l(c_1)$ and (s,t). (Note that, if we let c_2 be the edge (s,t), then $l(c_2^R) = (s,t)$, so P_l corresponds to our earlier notion of concatenating the svp_L's.)

Lemma 13 $WV(P,e) \subseteq P_l$.

Proof. No point with y-coordinate less than y_s can be in the weak edge visible region, by the manner in which the weak edge visible region is defined.

Suppose $w \in WV(P,e)$ is a point in P with $y_w \geq y_s$, and w lies in an up-pocket of c_1 (see Figure 28). All up-pockets of c_1 are subpolygons of P (since $l(c_1)$ and $c_2 = (s,t)$ do not intersect), separated from the rest of P by a horizontal lid (a subsegment of a segment drawn during procedure **Blue** or **Red**), and lying below the segment. Therefore, any path from a point of P outside the up-pocket to a point w inside the up-pocket must cross a horizontal segment from above. Since $y_u \leq y_w$ for any point u on e, the directed segment from x to w is in some direction in the range $[0,\pi]$. But we just saw that any (u,w)-path in P must somewhere face in a direction in the range $(\pi, 2\pi)$, so (w,u) is not contained in P, a contradiction. ∎

Having found P_l in linear time, we now note that we can triangulate P_l in linear time in the fashion discussed in procedure **Partition**. Since P_l is guaranteed to contain $WV(P,e)$, $WV(P_l,e) = WV(P,e)$, so we can now appeal to the existing methods of [To1] or [GHLST] to find $WV(P_l,e)$ in linear time. Note that we also have shown that polygons that are weakly visible from an edge can be triangulated in linear time.

We now describe linear-time solutions to three related problems that were solved by [GHLST] in linear time within a triangulated polygon.

(P5) Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point q, determine the portion of e that q can see.

We solve (P5) by constructing P_l in linear time and then noting that if q lies in $P \setminus P_l$, it sees no portion of e. Thus, given q, first determine whether q lies in $P \setminus P_l$ or P_l. (This can be done in $O(\log n)$ time since we can triangulate P_l in linear time, and we can also triangulate the *exterior* of P_l in linear time.) If q lies in $P \setminus P_l$, it sees none of e, and if it lies in P_l, we appeal to [GHLST].

(P6) (Ray shooting from an edge) Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point $q \in e$ and a direction d, determine the point on the boundary of P where a bullet would exit if fired from q in direction d.

We solve (P6) by noting that the answer to the query must be a point on the boundary of P_l (as defined above). (In fact, the answer must be a point on the boundary of $WV(P,e)$.) Triangulation of P_l in linear time implies the overall linear time bound.

(P7) (Convex rope problem) Given a point s on the convex hull of P, determine those points on the boundary of P that admit "convex ropes" from s around the outside of P.

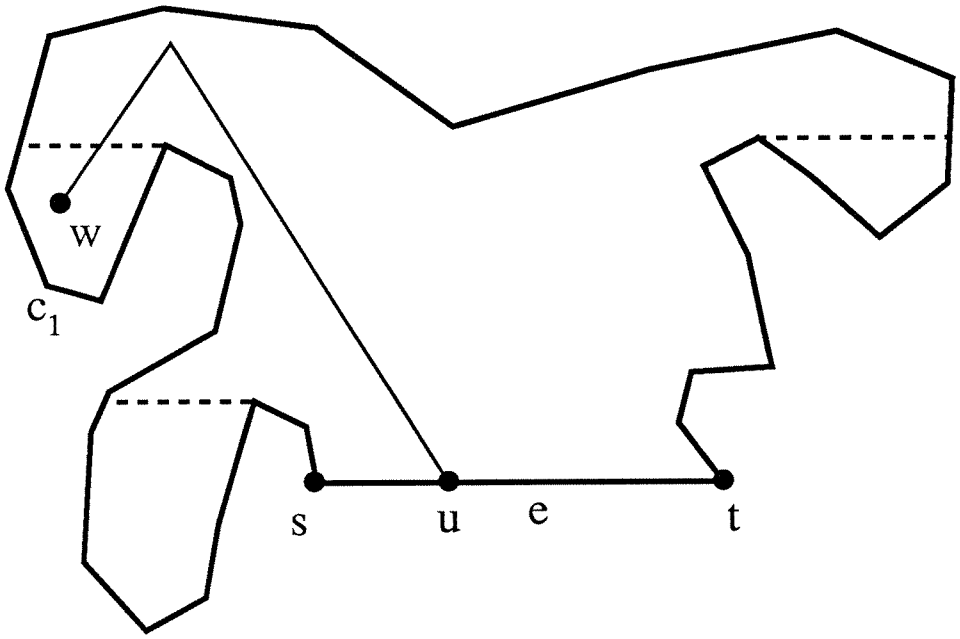


Figure 28. Proof of Lemma 13.

The *convex rope* problem was introduced in [PeS]. We make the problem definition more precise. For P a simple polygon, s a vertex of P lying on its convex hull, and v another vertex of P, the *clockwise convex rope* from s to v is the shortest simple polygonal path from s to v that does not enter the interior of P and is clockwise convex, i.e. has only "right turns". The counterclockwise convex rope is defined in a similar manner. The convex rope problem is to calculate the convex ropes for each vertex v that admits both clockwise and counterclockwise ropes from every vertex s on the convex hull.

We solve (P7) as follows. First we find the convex hull of P in linear time ([GY], [Me], [MA]), and take the collection of all "pockets of concavity". As described in [GHLST], only vertices in the pockets of concavity need to be considered. Each pocket, Q, together with its corresponding edge on the convex hull, e, forms a polygon. Any vertex v of Q not in the weak edge visible region of Q for e cannot admit convex ropes for all s, because the shortest paths from the endpoints of e to v must meet somewhere before v and turn in the same direction. Thus, we replace each pocket of concavity with its weak edge visible region from the pocket lid, and triangulate the weak edge visible regions. We then appeal to [GHLST], who complete the problem by finding shortest paths from a fixed source to all vertices in a triangulated polygon in linear time.

Another weak visibility problem asks us to find the subset of P that is illuminated by a convex subset Q of P.

(P8) (Illumination by a convex body) Given a convex k-gon Q within simple polygon P, determine the subpolygon of P that is weakly illuminated by a light source occupied by Q.

The subset of P illuminated by Q is called the *weak visible region* of P with respect to Q, and is denoted $WV(P,Q)$. More precisely, $WV(P,Q)$ is the set of all points w in P such that there exists an x in Q such that $\overline{wx}$ does not intersect the exterior of P. Ghosh [Gh] computes the weak visible region $WV(P,Q)$ in $O(n+k)$ time *given a triangulation* of P, where n and k are the number of vertices of P and Q, respectively. We give an optimal $\Theta(n+k)$ algorithm for computing $WV(P,Q)$.

Since Q is convex and inside P, the segment between any two points of Q lies in P. In other words, for any two points $x, y \in Q$, $ld(x,y) = 1$, where $ld(\cdot,\cdot)$ denotes link-distance in P. Let $x \in Q$ be an arbitrary point in Q. If $w \in P$ is in $WV(P,Q)$, then $ld(x,w) \leq 2$. This is true since $w \in WV(P,Q)$ implies $\overline{wy} \subset P$ for some $y \in Q$, and $x, y \in Q$ implies $\overline{yx} \subset P$, giving us a (w,x)-path of link-distance 2. Therefore, $WV(P,Q)$ lies in the subset of P within link-distance 2 of any fixed point x of Q.

We can obtain and triangulate this subset in $O(n)$ time. First take $VP(P,x)$, the visibility polygon of P from x. This can be done in linear time by a number of methods ([EA], [LL], [JS]), and $VP(P,x)$ can be triangulated in linear time because it is star-shaped. Each edge e of $VP(P,x)$ that is not an edge of P partitions the interior of P into two regions, where $VP(P,x)$ lies in one of the regions. Let P_e be the region not containing $VP(P,x)$. A point w in P_e has $ld(w,x) = 2$ if and only if w sees edge e, i.e. $w \in WV(P_e,e)$. We can compute and triangulate $WV(P_e,e)$ in linear time for each edge e of $VP(P,x)$ not an edge of P, and since the polygons P_e are disjoint this takes $O(n)$ time for all such edges e. We have now triangulated a region that contains $WV(P,Q)$, so by appealing to [Gh] we can compute $WV(P,Q)$ in $O(n+k)$ time.

As for weak edge visible regions, our method can be used to triangulate $WV(P,Q)$ in linear time. Also, $WV(P,Q) \setminus Q$ can be triangulated in linear time, since $VP(P,x) \setminus Q$ can be cut to form a weakly-simple polygon that is *radially monotone* about x (see [To3]).

We can, in fact, obtain and triangulate the region within link-distance L of Q in $O(Ln+k)$ time, by illuminating Q and then, in $L-1$ stages, illuminating each of the "window edges" on the boundary of the region illuminated so far, using the weak edge visibility algorithm. This solves the following problem in time $O(Ln+k)$ and space $O(n+k)$:

(P9) (Link reachability) Given a simple polygon P with n vertices and a convex k-gon Q within P, determine that portion of P that is at link distance $\leq L$ from Q and triangulate it.

8 Minimum Link Monotone Paths

In this section we solve the following in linear time:

(P10) (Monotone Min-Link Path) Given points s and t in a simple polygon P, determine if there is a monotone (s,t)-path in P, and if so produce a minimum link-distance (s,t)-path in P.

In Section 4 we described an algorithm for finding the shortest path between two points s and t in a polygon P, if the shortest path is monotone. The algorithm was shown to be correct by Corollary 6 and Theorem 7, which stated that if $l(c_1)$ and $l(c_2^R)$ intersect then the (Euclidean) shortest path faces left, and if $l(c_1)$ and $l(c_2^R)$ do not intersect then the (Euclidean) shortest path lies in P_l. Clearly a version of Corollary 6 for minimum link-distance paths hold by Lemma 5:

Lemma 14 *If $l(c_1)$ and $l(c_2^R)$ intersect, then every minimum link-distance path faces left.*

The following modification of Theorem 7 can be established using the fact that $l(c_1)$ and $l(c_2^R)$ have no left-peaks or right-valleys:

Theorem 15 *If $l(c_1)$ and $l(c_2^R)$ do not intersect, then there exists a minimum link-distance (s,t)-path that does not intersect any up-pockets, i.e. it lies in P_l.*

Proof. It suffices to show the following: given an (s,t)-path in P, it can be transformed into an (s,t)-path in P_l without increasing the link-distance.

Suppose c is an (s,t)-path in P, and c enters an up-pocket of c_2^R, with a blue lid. That is to say, the lid is a subsegment of a segment drawn by procedure **Blue**. Thus, when we look at P in its normal orientation, the up-pocket lies above the lid, and the left endpoint of the lid (call it a) is a right-valley (see Figure 29).

Assume $a \neq t$. Let g be the ray with root a in direction $\theta = \pi$. The chain $l(c_2^R)$ has been constructed in such a manner that every peak is a right-peak, which implies that c must intersect g for the first time from below (at point r, say) after last leaving the up-pocket (recall that t cannot lie *in* the up-pocket). If c first enters the up-pocket at point p and last leaves at q, then c visits p, q, and r in that order ($r = q$ iff $r = q = a$). Furthermore, the edges of c containing p and r face somewhere in the range $(0,\pi)$, while q lies on an edge facing somewhere in the range $(\pi, 2\pi)$. Therefore, there is at least one complete edge of c between p and r, so by replacing the subchain of c between p and r by the segment $\overline{pr}$, we do not increase the link-distance of the path. Also, this new path is in P_l, because the up-pocket is not intersected by $l(c_1)$, $\overline{ra}$ is not intersected by $l(c_1)$, and g is not intersected by $l(c_2^R)$.

If $a = t$, then replace the subchain of c from the point of first entry, p, with the $\overline{pt}$. Since p cannot be on the last edge of c, the new path has link-distance no greater than the original.

The other cases to consider are up-pockets of $l(c_2^R)$ with red lids, and up-pockets of $l(c_1)$ with either red or blue lids. However, the proofs are similar to the one above. ∎

With these two results, we can find minimum link-distance paths for s and t on the boundary of P whenever there exists a monotone path, using the same algorithm as in the Euclidean case. We can handle the case of s and/or t in the interior of P by shooting opposite rays as before, and proving in a fashion similar to the above proof that minimum link-distance paths need not enter the discarded region. Since the minimum link-distance path in a triangulated polygon can be found in linear time ([Su], [Gh]), our algorithm is linear time.

We have described how to find a minimum link-distance path from s to t whenever there exists a monotone (s,t)-path. In fact, the existence of a monotone (s,t)-path implies that there is a monotone minimum link-distance (s,t)-path. This is a consequence of the algorithm of Ghosh [Gh], which constructs a minimum link-distance path from the Euclidean shortest path, in such a manner that the cone of monotone directions is the same for both paths.

9 Summary and Conclusion

In summary, we have provided a method to solve a large catalogue of problems on simple polygons in optimal time. The method relies on building the *Structured Visibility Profile* of a polygonal path in linear time, and then using this data structure to obtain a partial triangulation of a polygon. We suspect that there are many other problems on simple polygons for which triangulation is a bottleneck in the running time and to which our methods can apply. We are currently investigating other problems.

For reference, we include below a summary of the problems solved in this paper. All problems are solved in linear (optimal) time $O(n)$, except for problem (P8), which is solved in optimal time $O(n+k)$, and problem (P9), which is solved in time $O(Ln+k)$.

(P1) Given two points s and t on or inside a polygon P, determine if there is a monotone (s,t)-path in P and, if so, produce one. Also produce the set of all directions for which there exists a monotone (s,t)-path. (This set is guaranteed to be a convex cone [ACM].)

(P2) Given two points s and t on or inside a polygon P, determine if the (unique) shortest path from s to t in P is monotone and, if so, produce it.

(P3) Given two disjoint simple polygons P and Q, determine the cone of all directions in which Q can be translated arbitrarily far without colliding with P.

(P4) Given a simple polygon P and a line segment e within P, find the subset of P that is illuminated if every point of e is a light source that casts light in every direction and the boundary of P is opaque.

(P5) Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point q, determine the portion of e that q can see.

(P6) (Ray shooting from an edge) Given an edge e, preprocess P so that the following query can be answered in $O(\log n)$ time: for a query point $q \in e$ and a direction d, determine the point on the boundary of P where a bullet would exit if fired from q in direction d.

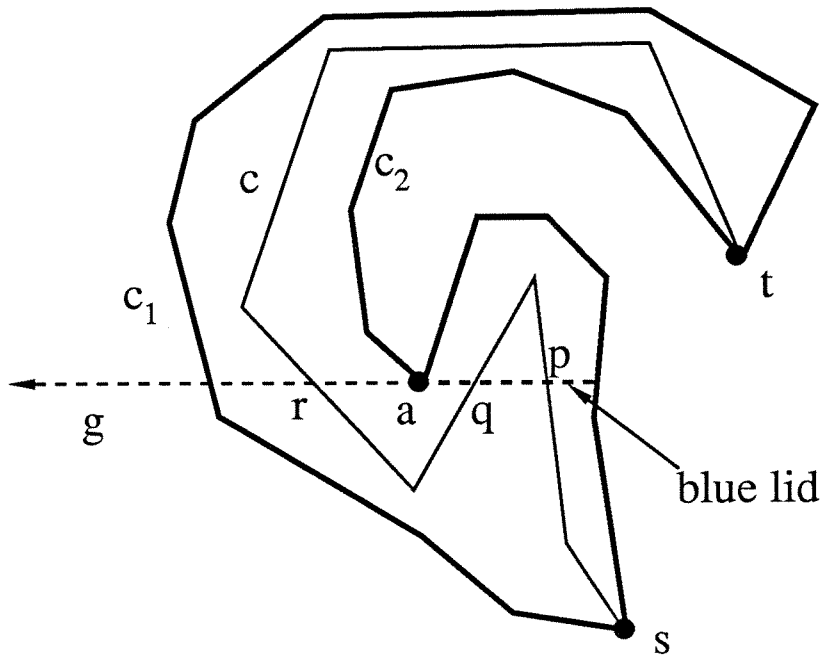


Figure 29. Proof of Theorem 15.

(P7) (Convex rope problem) Given a point s on the convex hull of P, determine those points on the boundary of P that admit "convex ropes" from s around the outside of P.

(P8) (Illumination by a convex body) Given a convex k-gon Q within simple polygon P, determine the subpolygon of P that is weakly illuminated by a light source occupied by Q.

(P9) (Link reachability) Given a simple polygon P with n vertices and a convex k-gon Q within P, determine that portion of P that is at link distance $\leq L$ from Q and triangulate it.

(P10) (Monotone Min-Link Path) Given points s and t in a simple polygon P, determine if there is a monotone (s,t)-path in P, and if so produce a minimum link-distance (s,t)-path in P.

Acknowledgement P. Heffernan is supported by an NSF graduate fellowship and thanks the support of DIMACS and Rutgers University for the use of facilities. J. Mitchell is supported by NSF grants IRI-8710858 and ECSE 8857642, and by a grant from Hughes Research Laboratories. We thank Jonathan Phillips and Godfried Toussaint for useful discussions and suggestions on this paper.

References

[ACM] E. Arkin, R. Connelly, and J.S.B. Mitchell, "On Monotone Paths Among Obstacles, with Applications to Planning Assemblies", Technical Report No. 838 School of Operations Research and Industrial Engineering, Cornell University, March, 1989. Appears in: *Proc. Fifth Annual ACM Symposium on Computational Geometry*, Saarbrücken, West Germany, June 1989, pp. 334-343.

[AT] D. Avis and G.T. Toussaint, "An Optimal Algorithm for Determining the Visibility of a Polygon from an Edge", *IEEE Transactions on Computers*, **C-30**, No. 12, (1981), 910-914.

[BT] B.K. Bhattacharya and G.T. Toussaint, "A Linear Algorithm for Determining Translation Separability of Two Simple Polygons", McGill Univ., School of Computer Science Report SOCS-86.1, Montreal, Quebec, Canada, 1986.

[CG] B. Chazelle and L.J. Guibas, "Visibility and Intersection Problems in Plane geometry", *Discrete and Computational Geometry*, **4** (1989), pp. 551-581.

[El] H.A. El Gindy, "Hierarchical Decomposition of Polygons with Applications", Ph.D. thesis, School of Computer Science, McGill University, May 1985.

[EA] H.A. El Gindy and D. Avis, "A Linear Algorithm for Computing the Visibility Polygon From a Point", *Journal of Algorithms*, Vol. 2 (1981), pp. 186-197.

[GJPT] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan, "Triangulating a simple polygon", *Information Processing Letters*, Vol. 7 (1978): 175–179.

[Gh] S.K. Ghosh, "Computing the Visibility Polygon from a Convex Set", CAR-TR-246, Center for Automation Research, University of Maryland, 1986.

[GY] R.L. Graham and F.F. Yao, "Finding the Convex Hull of a Simple Polygon", *Journal of Algorithms*, 4 (1983), pp. 324-331.

[GHLST] L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, "Linear Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons", *Algorithmica* **2** (1987), pp. 209-233.

[JS] B. Joe and R.B. Simpson, "Correction to Lee's Visibility Polygon Algorithm", *BIT*, **27** (1987), pp. 458-473.

[Le] D.T. Lee, "Visibility of a Simple Polygon", *Computer Vision, Graphics, and Image Processing*, Vol. 22 (1983), pp. 207-221.

[LL] D.T. Lee and A. Lin, "Computing the Visibility Polygon from an Edge", Tech. Report, Northwestern University, 1984.

[LP] D.T. Lee and F.P. Preparata, "Euclidean Shortest Paths in the Presence of Rectilinear Boundaries", *Networks*, **14** (1984), pp. 393-410.

[MA] D. McCallum and D. Avis, "A Linear Time Algorithm for Finding the Convex Hull of a Simple Polygon", *Information Processing Letters*, **8** (1979), pp. 201-205.

[Me] A. Melkman, "On-line Construction of the Convex Hull of a Simple Polyline", *Information Processing Letters*, **25** (1987), 11-12.

[PeS] M.A. Peshkin and A.C. Sanderson, "Reachable Grasps on a Polygon: The Convex Rope Algorithm", *IEEE Journal of Robotics and Automation*, **RA-2** (1986), pp. 53-58.

[PrS] F.P. Preparata and K.J. Supowit, "Testing a Simple Polygon for Monotonicity", *Information Processing Letters*, **12** (1981), pp. 161-164.

[Su] S. Suri, "A Linear Time Algorithm for Minimum Link Paths inside a Simple Polygon", *Computer Vision, Graphics, and Image Processing*, **35**, (1986), pp. 99-110.

[TV] R.E. Tarjan and C. Van Wyk, "An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon", *SIAM Journal on Computing*, **17** (1988), No. 1, pp. 143-178.

[To1] G.T. Toussaint, "A Linear-Time Algorithm for Solving the Strong Hidden-Line Problem in a Simple Polygon", *Pattern Recognition Letters*, **4** (1986), pp. 449-451.

[To2] G.T. Toussaint, "On Separating Two Simple Polygons by a Single Translation", *Discrete and Computational Geometry*, **4** (1989), pp. 265-278.

[To3] G.T. Toussaint, "Shortest Path Solves Translation Separability of Polygons", Technical Report SOCS-85-27, McGill School of Computer Science, October, 1985.

[To4] G.T. Toussaint, "Movable Separability of Sets", in *Computational Geometry*, ed. G.T. Toussaint, North-Holland, 1985, pp. 335-375.