FOCS - 82

A Theorem on Polygon Cutting with Applications

Bernard Chazelle[*]

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Let P be a simple polygon with N vertices, each being assigned a weight $\in \{0,1\}$, and let C, the weight of P, be the added weight of all vertices. We prove that it is possible, in $O(N)$ time, to find two vertices a,b in P, such that the segment ab lies entirely inside the polygon P and partitions it into two polygons, each with a weight not exceeding $2C/3$. This computation assumes that all the vertices have been sorted along some axis, which can be done in $O(N\log N)$ time. We use this result to derive a number of efficient divide-and-conquer algorithms for:

1. Triangulating an N-gon in $O(N\log N)$ time.
2. Decomposing an N-gon into (few) convex pieces in $O(N\log N)$ time.
3. Given an $O(N\log N)$ preprocessing, computing the shortest distance between two arbitrary points *inside* an N-gon (i.e., the *internal* distance), in $O(N)$ time.
4. Computing the longest internal path in an N-gon in $O(N^2)$ time.

In all cases, the algorithms achieve significant improvements over previously known methods, either by displaying better performance or by gaining in simplicity. In particular, the best algorithms for Problems 2,3,4, known so far, performed respectively in $O(N^2)$, $O(N^2)$, and $O(N^4)$ time.

1. Introduction

Lipton and Tarjan's planar separator theorem [LT77,LT77] is a notable example of a systematic technique for introducing a computational tool, i.e., *divide-and-conquer*, into a whole class of related problems, i.e., *planar graph problems*. Drawing its inspiration from this philosophy, this paper presents a theoretical result on polygon decomposition which can be applied to derive a number of efficient algorithms for geometric problems, in particular, problems of convex decompositions, triangulation, visibility, and internal distance.

[*] The author's present address is Department of Computer Science, Brown Univ., Providence, R.I. 02912. This research was supported by DARPA, Order No. 3597, Contract F33615-81-K-1539.

2. The Polygon-cutting Theorem

Let P be a simple (i.e., not self-intersecting) polygon with vertices $v_1,\ldots,v_N$ in clockwise order. Let OXY be an orthogonal system of reference. Wlog, we can always assume that no two vertices have the same X-coordinate. We define a (vertical) relation between edges of P as follows: two edges of P are said to be vertically related iff there exists a line segment parallel to the Y-axis which intersects both edges without intersecting any other edge of P (fig.1) Note that it is easy to set up the list LV of all pairs of related edges in O(Nlog N) time, using a method developed in [SH76]. Sweep a vertical line L from left to right, maintaining the vertical order of the edges L currently intersects in a dynamic balanced search tree (e.g., AVL-tree). The line L starts at the leftmost vertex and proceeds to visit all of them in ascending X-order. If the current vertex is the left endpoint of an edge, this edge is inserted into the tree, otherwise it is deleted. We may omit the details.

Throughout this paper, we will assume that along with a description of the boundary of P, provided by a doubly-linked list LP, we have available a doubly-linked list LH of all the vertices sorted by X-coordinates [KN73]. The preprocessing involved in setting up these lists requires $O(N\log N)$ time and $O(N)$ space. The decomposition algorithm which we will describe later on runs in linear time, with this preprocessing in hand. Note that it is legitimate to separate both tasks, since in the applications which we will mention, the decomposition algorithm will be called recursively several times, while the preprocessing will be needed initially, once and for all. The goal of this section is to prove the following theorem:

Theorem 1: *The Polygon-cutting Theorem.* Let P be a simple polygon with N vertices $v_1,\ldots,v_N$, each assigned a weight c_i ($c_i=0,1$). Let $C(P)$ denote the total weight of P, defined as the sum $c_1+\ldots+c_N$, and assume that $C(P)>2$. With the lists LP,LH,LV in hand, it is possible to find, in $O(N)$ time, a pair of vertices v_i,v_j such that the segment v_iv_j lies entirely inside the polygon P and partitions it into two

0272-5428/82/0000/0339$00.75 © 1982 IEEE

simple polygons P_1, P_2 satisfying:

$$C(P_1) \leq C(P_2) \leq 2C(P)/3$$

The weights of the vertices in P_1 and P_2 are the same as in P, except for v_i and v_j, for which we will assume that in both P_1 and P_2, these weights become 0. This assumption is made only for the sake of simplicity, and other conventions (e.g., keeping the same weights c_i, c_j in both P_1 and P_2) are indeed acceptable, if we are ready to add a term $+2$ to $2C(P)/3$ in the inequality of Theorem 1. To facilitate our task, in a first stage, we will prove the theorem with slightly relaxed requirements.

2.1. An existence theorem

To begin with, we prove the existence, not of two *vertices*, but of two *points* on the boundary of P, satisfying the inequalities of Theorem 1.

> Theorem 2: Same assumptions as Theorem 1. There exists a pair of points A,B on the boundary of P, such that the segment AB is parallel to the Y-axis, lies entirely in the polygon P, and partitions it into two simple polygons P_1, P_2 satisfying:
>
> $$C(P_1) \leq C(P_2) \leq 2C(P)/3$$

If A and B are not vertices of P, for consistency, we assign them a 0-weight. We introduce a distance function $d(A,B)$, defined between two points A,B on the boundary of P as the minimum path weight between A and B. More precisely, let v_iv_{i+1} (resp. v_jv_{j+1}) be the edge of P containing A (resp. B). If A (resp. B) is a vertex of P, it is assumed to be v_i (resp. v_j). We introduce the function h, defined as follows: (*arithmetic on indices done mod N*)

$$h(A,B) = c_{j+1} + c_{j+2} + \ldots + c_i,$$

from which we can define $d(A,B)$:

$$d(A,B) = \min \{ h(A,B), C(P) - h(A,B) \}$$

Starting at the edge v_1v_2, we label each edge of P recursively, as follows: $\lambda(v_1v_2) = c_1$

$$\lambda(v_iv_{i+1}) = \lambda(v_{i-1}v_i) + c_i$$

Note that this labeling gives us an alternate way of defining the distance between two boundary points A,B:

$$d(A,B) = \min \{ |\lambda(v_iv_{i+1}) - \lambda(v_jv_{j+1})|, C(P) - |\lambda(v_iv_{i+1}) - \lambda(v_jv_{j+1})| \}$$

We are now in a position first to prove the existence of the segment AB, as defined in Theorem 2, then to describe an efficient method for finding it. As we will see, the first step is not superfluous; it is an essential ingredient in ensuring the correctness of the algorithm. Choose the leftmost point of P as the starting point of the left-to-right sweep of a vertical segment $S = AB$ (A below B). S will always stretch vertically so as to keep its endpoints A,B in permanent contact with the boundary of P. It will thus be able to move continuously to the right, until it must either expand (fig.2) or split (fig. 3). At any time during the course of the motion, S will be assigned a value $\Delta = h(A,B)$ to indicate how close it is to being the desired segment. We observe that initially, $\Delta = C(P)$, and that as long as S moves continuously, Δ decreases monotonously by unit steps. When either situation depicted in Figures 2,3 arises, we can always write (1) $\Delta = h(A,B) = h(A,C) + h(C,B)$, from which we can derive a decision procedure for redefining S.

> Starting from the leftmost vertex of P, move S from left to right, stretching or shrinking this segment so that it entirely lies in P, and its endpoints always lie on the boundary of P. As long as the motion of S is continuous, check whether $\Delta > 2C(P)/3$, in which case continue, else stop. When falling in either case of fig.2,3 reset S to AC if $h(A,C) > h(C,B)$, or to CB otherwise. Note that if S is reset to CB in the case of fig.2 , the motion must reverse its direction.

Relation (1) shows that every discontinuity causes Δ to decrease by at most half, while otherwise Δ decreases at most by unit steps, since we have assumed that no two vertices may lie on the same vertical line. Since Δ eventually vanishes and $C(P) > 2$, it must take on some value in the interval $[C(P)/3, 2C(P)/3]$, at which point the procedure will stop and return the desired segment AB. This completes the proof of Theorem 2. □

2.2. A relaxed version of the polygon-cutting theorem

Unfortunately, Theorem 2 falls short of providing an efficient algorithm for computing AB. We can, however, graft to it a binary search-like structure to improve the performance of a naive implementation. The purpose of this section is thus to prove the following result:

> Theorem 3: Same assumptions as Theorem 1. It is possible, in $O(N)$ time, to find a pair of points A,B on the boundary of P, such that the segment AB lies entirely in the polygon P, and partitions it into two simple polygons P_1, P_2 satisfying:
>
> $$C(P_1) \leq C(P_2) \leq 2C(P)/3$$

Since the edges supporting A and B must be vertically related, we can try out all pairs of LV, checking for the inequalities to hold and for the proper orientation of the edges relative to the interior of P. Theorem 2 ensures that we will thus find the desired pair of edges, hence AB.

2.3. Completing the proof of the polygon-cutting theorem

We may now turn our attention back to Theorem 1. Let $v_i v_{i+1}$ and $v_j v_{j+1}$ be the edges of P that contain the points A and B of Theorem 3, respectively. To prove the desired result, one may be tempted to slide A and B towards the endpoints of $v_i v_{i+1}$ and $v_j v_{j+1}$ respectively, until one of the configurations $v_i v_j$, $v_i v_{j+1}$, $v_{i+1} v_j$, or $v_{i+1} v_{j+1}$ has been reached. Unfortunately, obstacles may prevent this from ever happening (fig.4), so our next step will be to take a closer look at these possible obstacles. Since the quadrilateral $v_i v_{i+1} v_j v_{j+1} v_i$ contains the segment AB, it is a simple polygon, and AB partitions it into two polygons $Q_1 = A v_{i+1} v_j BA$ and $Q_2 = AB v_{j+1} v_i A$. As a corollary of the Jordan Curve Theorem, which states that a closed curve in the plane partitions the plane into two connected regions [HS55], it appears that the only obstacles encountered in Q_1 (resp. Q_2) are vertices in the set $\{v_{i+1}, v_{i+2}, \dots, v_j\}$ (resp. $\{v_{j+1}, v_{j+2}, \dots, v_i\}$). Moreover, the segment $S = AB$ can encounter only vertices on the convex hull H_1 (resp. H_2) of the vertices of P lying inside Q_1 (resp. Q_2) (fig.4), as is shown in following result.

Lemma 4: The segment AB intersects any edge of P (outside of A or B) if and only if it intersects the boundary of either H_1 or H_2.

Proof: Since H_1 (resp. H_2) lies entirely inside Q_1 (resp. Q_2), AB intersects any edge of P lying in Q iff the infinite line passing through AB does, hence iff AB lies outside of H_1 and H_2. □

The next task is to compute the convex hulls H_1 and H_2. We only give the details of the algorithm for H_1, the other case being strictly similar. Since we cannot afford to use a standard $O(N \log N)$ algorithm to simply compute the convex hull of the vertices of P in Q_1, we must exploit the fact that these vertices lie on a polygonal line in order to achieve linear time. To begin with, let us give an informal description of the algorithm. The goal is, in a first stage, to produce a polygon K_1 which lies entirely in Q_1, and whose convex hull is exactly H_1. Let a *polygonal chain* be a non-intersecting connected sequence of segments. K_1 consists essentially of polygonal chains L_i made of consecutive edges from the set:

$$L = \{ v_{i+1}v_{i+2}, v_{i+2}v_{i+3}, \dots, v_{j-1}v_j \}$$

Each chain has the property that it lies in Q_1 and intersects $v_{i+1}v_j$ in two points, U_k and V_k. Moreover, no two segments $U_k V_k$ overlap (fig.5.1). To compute these chains, we must distinguish between two types of edges in L. An edge $v_k v_{k+1}$ is said to be *entering* (resp. *exiting*) if it intersects $v_{i+1}v_j$, and v_{k+1} lies inside (resp. outside) Q_1. The algorithm proceeds as follows:

Wlog, assume that there is at least one vertex v_{i+2} from v_{i+1} to v_j in clockwise order. Traverse L from $v_{i+1}v_{i+2}$ to $v_{j-1}v_j$, stopping at entering and exiting edges and taking the following actions. If the current edge is *entering*, it may be the endpoint U_k of a new chain L_k. To decide on this, look at the next *exiting* edge in L; if it intersects $v_{i+1}v_j$ in a point V_k on the segment $U_k v_j$, we have indeed a new chain L_k from U_k to V_k. Otherwise, not only don't we have a new chain L_k, but the chain just visited may enclose previously computed chains, which must then be deleted. For that purpose, we use a stack to hold the pairs $(U_k V_k, L_k)$, so that deletions may be done efficiently. The algorithm is straightforward, so we omit the details.

```
Initially, the stack is empty, and the current edge $e$ is the first
*entering* edge of $L$.
begin
Let $f$ be the next *exiting* edge in $L$ following $e$, and let $U, V$ be
respectively the intersections of $e$ and $f$ with $v_{i+1}v_j$.

if $V$ lies on $Uv_j$
then
    Let $L_u$ be the chain between $e$ and $f$ in $L$.
    Push $(UV, L_u)$ onto stack.
    Go to next *entering* edge $e'$ in $L$, whose intersection
    with $v_{i+1}v_j$ lies on $Vv_{j+1}$, then iterate.

else
    Go to next *entering* edge $e'$ whose intersection $U'$
    with $v_{i+1}v_j$ lies on $v_{i+1}V$.
    Pop all pairs $(U_k V_k)$ off the stack as long as $V_k$ lies on $U'v_j$.
    Iterate.
end
```

To prove the correctness of the algorithm, we begin by observing that the intersection of L with Q_1 consists of chains whose endpoints lie on $v_{i+1}v_j$, and that K_1 consists of exactly all the *maximal* chains. A chain is said to be *maximal* if it does not lie in the enclosure of any other chain with $v_{i+1}v_j$. A maximal chain is also characterized by the fact that the segment formed by its endpoints does not lie inside any other such segments.

From the Jordan Curve Theorem [HS55], it follows that a maximal chain from U to V, in clockwise order, has its endpoint V lying above U (i.e., on the segment Uv_j). In consequence, only the chains which *move* towards v_j are candidates for being part of K_1, which justifies the selection criterion of the algorithm. On the other hand, we can also show that a non-maximal chain which moves towards v_j is necessarily enclosed by another chain moving away from v_j. This explains the deletion rule. Finally, the last observation to make is that a chain from U to V which moves away from v_j (i.e., U lies on Vv_j) must be enclosed by a subsequent maximal chain, therefore since maximal chains are computed "towards" v_j, we may skip directly to the next *entering* edge in L that intersects $v_{i+1}v_j$ at a point U' below V (i.e., U' lies on $v_{i+1}V$) - see illustration of the various cases in fig.5.1. This completes the proof of correctness, and shows that all the chains L_k may be computed in sorted order along the segment $v_{i+1}v_j$, all these computations requiring $O(N)$ time. The final step in computing K_1 is to connect all the chains L_k together in the order in which they appear in the stack. To do so, we borrow segments from $v_{i+1}v_j$, as shown in fig.5.2. We may now apply any standard linear convex hull algorithm for simple polygons [LE80] in order to obtain H_1 in $O(N)$ time.

Assuming that both H_1 and H_2 are available, we are now in a position to give an algorithm for finding the two vertices of Theorem 1. The idea is to connect the vertices of H_1 with those of H_2, so as to triangulate the polygon H^* defined as the area between H_1 and H_2 containing AB. We claim that at least one of the edges of the triangulation will provide the desired pair of vertices, with the property of Theorem 1. The algorithm proceeds as follows:

Let $h_1,...,h_p$ and $k_1,...,k_q$ be the vertices of H_1 and H_2, respectively, as we traverse them from $v_{i+1}v_i$ to v_jv_{j+1}, i.e., $h_1=v_{i+1}$, $h_p=v_j$, $k_1=v_i$, $k_q=v_{j+1}$. We maintain two pointers, h on H_1 and k on H_2, moving them from h_1 to h_p and k_1 to k_q, respectively, and computing the triangulation on the fly. Note that, at all times, the segment hk intersects H_1 and H_2 only at its endpoints (fig.6.1).

The simplest way of describing the algorithm is recursively. Initially, hk is $v_{i+1}v_i$; the algorithm terminates with $hk = v_jv_{j+1}$.

Let $hk = h_tk_u$, and consider the quadrilateral $hh_{t+1}k_{u+1}kh$. For consistency, we define h_{p+1} and k_{q+1} as v_{j+1} and v_j, respectively. We will show in Lemma 5 that at least one of its diagonals, hk_{u+1} or kh_{t+1}, connects H_1 and H_2 without intersecting these polygons outside of its endpoints, i.e., lies entirely in H^*. Moreover, this diagonal can be found in constant time. We may then determine that diagonal, add it to the triangulation, set hk to it, and iterate.

The algorithm clearly runs in linear time. Also, the assurance that it effectively produces a triangulation of H^* comes from the fact that it keeps only edges which lie entirely in H^*, and that the pointers h and k pass a vertex only after a diagonal has been assigned to it. Thus there only remains to prove the following lemma:

Lemma 5: If the segment h_tk_u connects H_1 and H_2 and lies entirely inside H^*, so does one of the diagonals h_tk_{u+1} or k_uh_{t+1}. Moreover, this diagonal can be found in constant time.

Proof: Consider the line passing through hk, oriented from h to k. If a point lies to the right (resp. left) of this line, we will say that it lies *below* (resp. *above*) hk. Since H_1 and H_2 are convex, at least one of the vertices v_j or v_{j+1} lies above hk, therefore it is impossible that both h_{t+1} and k_{u+1} lie below hk. Indeed this would involve the existence of at least three intersection points between a line and a convex boundary, leading to a contradiction. If only one segment, say h_{t+1} lies above hk, it can be easily determined in constant time, and since in that case, all of H_2 lies below hk, the diagonal $h_{t+1}k$ does not intersect H_2 (nor H_1 either) outside of its endpoints, and may thus be chosen as the next segment of the triangulation (fig.6.2). If, on the other hand, both h_{t+1} and k_{u+1} lie above hk, the quadrilateral $hh_{t+1}k_{u+1}k$ is a simple polygon, therefore it contains at least one of its diagonals entirely (fig.6.3), and this diagonal can be found in constant time. Note that, because of its convexity, H_1 (resp. H_2) lies totally on one side of the line passing through hh_{t+1} (resp. kk_{u+1}), therefore the whole quadrilateral, hence the chosen diagonal, lies inside the polygon H^*, which completes the proof. □

The purpose of triangulating the polygon H^* will become apparent with the following result.

Lemma 6: There exists an edge uv in the triangulation of H^* which satisfies the relation: $C(P)/3 \leq h(u,v) \leq 2C(P)/3$.

Proof: From Theorem 3, we know that AB partitions P into two polygons with weights between $C(P)/3$ and $2C(P)/3$, which gives the relations

$$C(P)/3 \leq \min(\, h(A,B), h(B,A)\,) \leq \max(\, h(A,B), h(B,A)\,) \leq 2C(P)/3$$

As a result, any pair of vertices a,b on H_1 (resp. H_2), with b following (resp. preceding) a in the list $\{h_1,...,h_p\}$ (resp. $\{k_1,...,k_q\}$) satisfies the relation (1) $h(b,a) \leq 2C(P)/3$. On the other hand, each triangle abc of the triangulation has one side ab on the boundary of either H_1 or H_2, with the two others ac, bc constructed by the triangulation algorithm. Wlog, let ac be the segment of the triangle constructed first (i.e., ac lies below bc). We always have (2) $h(c,a) = h(b,a) + h(c,b)$. Now we can show that a simple upward scan through the faces of the triangulation, i.e., starting at the triangle adjacent to v_iv_{i+1} and ending at the triangle adjacent to v_jv_{j+1}, will inevitably lead to the desired segment of Theorem 1. To see that, we may obviously assume that none of the edges ab of H_1 or H_2 satisfies the relations:

$$C(P)/3 \leq h(b,a) \leq 2C(P)/3\,,$$

otherwise, we have achieved our goal. In that case, Relation (1) shows that for any triangle abc visited, the edge on the boundary of H^*, say ab, satisfies the stronger inequality $h(b,a) < C(P)/3$, which, combined with Relation (2), leads to

$$h(c,b) > h(c,a) - C(P)/3.$$

Since $h(v_i,v_{i+1}) = C(P) - c_{i+1}$ and $h(v_{j+1},v_j) = c_{j+1}$, it follows that if $a_1b_1, a_2b_2, ...$ is the sequence of interior edges visited in the traversal of the triangulation, with the points a_m (resp. b_m) on H_1 (resp. H_2), the sequence $h(a_1,b_1), h(a_2,b_2),...$ is monotonously decreasing from $C(P)-c_{i+1}$ to c_{j+1} by jumps of at most $C(P)/3$. In consequence, it must take on at least one value in the interval $[C(P)/3\,,\,2C(P)/3]$, which can then be chosen as the pair u,v. □

The proof of Theorem 1 is now complete. Computing H_1 and H_2 definitely constitutes the most difficult part of the algorithm to implement. We may observe, however, that this overhead will often be unnecessary since, in practice, it may be seldom the case that the segment AB of Theorem 3 is prevented from sliding towards the endpoints of its supporting edges.

One final remark: Assume that P is triangulated, and consider the graph connecting adjacent faces. As can (and will) be shown easily, this graph is a tree whose centroid corresponds roughly to the cutting edge.

3. Applications to polygon decomposition problems

It is intuitive that the polygon-cutting theorem should lead to efficient methods for partitioning a polygon into convex pieces. We will examine two instances of this problem: in one, what is desired is a partition of the polygon into a *small* number of convex pieces, while in the other, only a triangulation of the polygon is sought, without consideration of optimality. Applying a quality criterion to the triangulation of a polygon is common practice in numerical analysis, where an area distribution or a shape function is often to be optimized. There are many good reasons, however, for making the availability of *any* triangulation desirable, as we will see later on.

3.1. Convex decompositions

Given a simple, non-convex polygon P, find a minimum number of convex, pairwise disjoint polygons, whose union is P.

This problem has been well-studied [CH80, GJ78, SC78], and several algorithms have been discovered for producing minimal or near-minimal decompositions. Here we consider only decompositions which do not introduce new points, i.e., all the vertices of the polygons are vertices of P. In connection with the previous section, we will assign to each vertex v_i of P a weight $c_i=1$ if its adjacent edges form a reflex angle (in which case, v_i is called a *notch*), and a weight $c_i=0$ otherwise. Thus we can apply the polygon-cutting theorem (Theorem 1) iteratively to decompose P into smaller polygons. Note that since, with our convention, the endpoints of the splitting segment lose their weights, the number of notches C_1,C_2 of the two parts is each bounded by $2C/3+2$, where C is the number of notches in the original polygon. As a result, we must stop the iteration when the algorithm ceases to reduce the number of notches, i.e., when all the parts have a number of notches satisfying: $C \leq 2C/3+2$, i.e., $C \leq 6$. Finally, to resolve the remaining reflex angles, we consider each of them in turn, proceeding as follows:

> Let Q be the polygon (with at most 6 reflex angles), and v be the notch exhibiting the reflex angle to be resolved. Let L (resp. R) denote the *ray* (i.e., semi-infinite line) starting at v in the direction of the edge ending (resp. starting) at v (fig.7). Compute the intersection(s) of L (resp. R) with the boundary of Q, and keep only the intersection point A (resp. B) closest to v. If A and B lie on different edges, there exists at least one vertex on the part of the boundary of P between A and B which can be joined to v, so as to resolve the reflex angle at v (fig.7.1). For example, we can choose the vertex w between A and B that minimizes the angle (vB,vw), and lies in the triangle formed by v and the edge supporting B. If, on the other hand, A and B lie on the same edge v_Av_B (fig.7.2), we compute the vertex a of the list $(v,\ldots,v_A)$, given in clockwise order, which lies in the triangle vAv_A and minimizes the angle (vA,va). Similarly, we compute the vertex b which lies in vBv_B and minimizes the angle (vb,vB). Both of these operations can be executed in linear time. Note that minimizing the angles ensures that both va and vb lie entirely in Q. It is also easy to show that the combination of these two segments resolves the reflex angle at v by splitting Q into 3 polygons (note that in most cases, a and b will be v_A and v_B, respectively).

The decomposition algorithm thus consists of a recursive "cutting" phase which relies on the algorithm given for the polygon-cutting theorem. The recursion stops when the polygon currently examined has fewer than 7 notches, at which point the procedure just described is called upon to finish off the decomposition. We observe that if either the vertex v_i or v_j, say v_i, in Theorem 1 is a notch, i.e., $c_i=1$, v_i appears in both of the resulting polygons P_1 and P_2, but is a notch for at most one of them. Therefore if, by extension, we let $C(N)$ denote the weight of P and $C(N_1)$ (resp. $C(N_2)$) be the weight of P_1 (resp. P_2), we can write: (1) $C(N) \geq C(N_1) + C(N_2)$ and (2) $C(N_1) \leq C(N_2) \leq 2C(N)/3 +2$. Note that $C(N_1)$ and $C(N_2)$ are actual weights, i.e., they count exactly the number of notches in P_1 and P_2, as opposed to the weights of P_1 and P_2 as defined in Theorem 1, which did not account for the endpoints of the splitting segment. It is easy to see that, in the worst case, we will end up with $C(N)/6$ polygons with, each, 6 notches, and the final phase will use 2 cuts for the resolution of each reflex angle. This will result in $13C(N)/6$ convex pieces, which is to be compared with the minimum number of convex pieces, shown to be always greater than or equal to $C(N)/2+1$ in [CH80]. Next we turn to the complexity of our decomposition algorithm. While it clearly needs $O(N)$ space, evaluating its run-time $T(N)$ calls for further investigation. If we neglect the preprocessing phase for the time being, we have the relations:

$$(3)\ T(N) = T(N_1) + T(N_2) + O(N), \text{ if } C(N) > 6$$
$$(4)\ N_1 + N_2 = N + 2$$
$$(5)\ T(N) = O(N), \text{ if } C(N) \leq 6$$

Consider the recursion tree, and label each node with the number p of vertices in the corresponding polygon. At the leaves, we have $C(P) \leq 6$, while for their ancestors, $C(P) > 6$. Relations (3) and (5) show that up to within a constant factor, $T(N)$ is equal to the sum of the labels in the tree, while from Relation (4), it follows that if $L(i)$ counts the sum of all labels at level i, we have $L(0)=N$ and $L(i) \leq L(i-1) + 2^i$, which gives $L(i) \leq N + 2^{i+1}$. If k is the height of the tree, we easily find that

$$L(0) + \ldots + L(k-1) \leq Nk + 2^{k+1},$$

and since, from (2), we have $k = O(\log C(N)) = O(\log N)$, including the $O(N\log N)$ preprocessing in the running time, we can conclude:

> Theorem 7: In $O(N\log N)$ time and with $O(N)$ space, it is possible to decompose a simple N-gon P into fewer than 4.333...xOPT convex pieces, without introducing new vertices, where OPT is the minimum number of convex pieces necessary to partition P.

3.2. Triangulation

When all the pieces of a convex decomposition are triangles and no new vertices are introduced, the decomposition is called a *triangulation* of the polygon. An $O(N\log N)$ algorithm for computing a triangulation of a simple polygon has been given in [GJ78]. Another method consists of using a strategy based on the polygon-cutting theorem. We may choose to assign a weight $=1$ to each vertex of P and apply the polygon-cutting theorem recursively, until the polygon under consideration has fewer than 7 vertices, at which point it is straightforward to complete the triangulation. We omit the details. An alternative consists of computing a convex decomposition of P as described in the previous section, then triangulate each convex polygon. To do so, pick any vertex of the polygon and join it to every other. In both cases, a triangulation of P can be explicitly computed in $O(N\log N)$ time, which matches the performance of [GJ78]. We recall that it is yet unknown whether $kN\log N$ is optimal for this problem.

Theorem 8: Using the polygon-cutting theorem, it is possible to triangulate a simple N-gon in $O(N\log N)$ time and $O(N)$ space.

4. Visibility problems

A problem which arises frequently in graphics concerns the elimination of hidden lines from a two- or three-dimensional scene [NS79]. In two dimensions, the problem reduces to computing the sets of points that are visible from a given point inside a polygon P. Linear algorithms for this problem already exist [CH80,EA81], but they involve complicated stack manipulations. Instead, we can use the polygon-cutting theorem for this problem, which can be formulated as follows:

Given a simple polygon P and a point M inside P, the locus of points V such that the segment MV lies entirely in P is a simple polygon $V(M)$. Compute a clockwise description of the boundary of $V(M)$.

We will assume that we have available a triangulation T of P, as produced, for example, by the algorithm of the previous section. We can regard the triangles of T as forming the nodes of a graph G, whose edges join the pairs of triangles with a common edge (i.e., an interior edge) (fig.8). As shown in Lemma 9, the absence of interior faces ensures that the graph G is actually a tree.

Lemma 9: G is a tree.

Proof: It suffices to show that for any pair of triangles t_1,t_2 in the triangulation, there exists a unique path between t_1 and t_2 in G. The triangle t_1 partitions P into 4 parts. One is the triangle t_1 itself, the others being polygons adjacent to the edges of t_1 (note that some of these polygons may be reduced to a single edge). At any rate, exactly one of the three polygons contains the triangle t_2. Call U this polygon, letting u denote its edge adjacent to t_1 and t be the triangle of T adjacent to u and lying in U. Since the triangulation of P also provides a triangulation of U, and its associated graph G_u is a subgraph of G, we can see that if there is a unique path in G_u from t to t_2, there is also a unique path in G from t_1 to t_2. Therefore we can prove the lemma by induction on the number of vertices. □

Let e be any interior edge of the triangulation, and let M be any point inside P. Letting t denote the triangle of T which contains M, we can define $G(M,e)$ as the unique subtree of G emanating from e, which does not contain t (fig.9). We are now in a position to give an algorithm for computing the visibility polygon $V(M)$. To facilitate our task, we introduce the function VISIB, defined as follows: let e^* be any segment lying entirely on the edge e. Remove from T all the triangles which do not belong to $G(M,e)$, and call Q the resulting polygon. We define VISIB (M,e^*) as the part of Q which is visible from M *through* the window e^*. More precisely, VISIB (M,e^*) is the set of points u in Q such that the only intersection of Mu with the boundary of Q takes place at e^* (fig.10). Let a,b,c be the vertices of the triangle in Q adjacent to e with $e=ab$ and $e^*=a^*b^*$. We define A (resp. B) as the intersection of the polygonal line $\{bc,ca\}$ with the infinite line passing through Ma^* (resp. Mb^*). It is now straightforward to compute the function VISIB recursively.

VISIB (M,e^*)

```
if $e^*$ lies on the boundary of $P$
    then
        return ($\{e^*\}$)
else
    Determine the points $c,A,B$.
    if $c$ lies between $A$ and $B$
        then
            $V \leftarrow$ VISIB $(M,Bc)$
            $V \leftarrow V \cup$ VISIB $(M,cA)$
        else
            $V \leftarrow$ VISIB $(M,AB)$
    return ($V$) (fig.10)
```

To complete the computation of $V(M)$, it suffices to determine the triangle of T where M lies - which can be done in $O(N)$ time - then apply the previous procedure with respect to its three edges.

VISIBILITY (P,M)

```
Let $e_1,e_2,e_3$ be the edges in clockwise order of the
triangle of $T$ which contains $M$. Initially $V(M) = \emptyset$.
for i = 1,2,3
    begin
    $V(M) \leftarrow V(M) \cup$ VISIB $(M,e_i)$
    end
```

Note that, as described, the procedure reports the boundary of V(M) in clockwise order, except for the *ray-edges* of V(M), i.e., the segments collinear with M, which are omitted. A single pass through the list V(M), however, will be sufficient to add the missing segments, and we need not elaborate. Using a standard (e.g., DCEL) representation of the triangulation ensures that each recursive step can be executed in constant time, from which we can conclude:

Theorem 10: Given a simple N-gon P along with an arbitrary triangulation of P, it is possible to compute the visibility polygon from any point M inside P, in O(N) time.

The main advantage of this algorithm is that it avoids the complicated stack manipulations of [CHS0] and [EA81]. The reader may convince himself/herself/itself that the algorithm could be rewritten without greater difficulty in order to deal directly with a more general convex decomposition (i.e., without first converting it into a triangulation). This may be an interesting alternative if one is willing to exploit the fact that searching among the edges of a convex polygon can be done in logarithmic time, using a Fibonacci search-based strategy [CH80.CD80].

5. Applications to internal distance problems

5.1. The car-racing problem

What is the shortest trajectory of a racing car on a given circuit? More precisely, the problem which we address in this section can be expressed as follows:

Given a simple polygon P and two arbitrary points A and B in P, find the shortest path inside P between A and B (fig.11).

This shortest path is called the *internal path* between A and B, denoted IP(A,B), and its length, |IP(A,B)|, is called the *internal distance* between A and B (fig.11). To have a visual representation of IP(A,B), one can imagine a rubber band inside P tightly stretched between A and B. In [SM77], Shamos suggests an O(N^2) algorithm for computing IP(A,B). The method consists essentially of computing all pairs of vertices visible from each other, in O(N^2) time, so as to form the so-called *viewability graph* of P. We next add weights to the graph by associating to each edge the Euclidean distance between its endpoints. Computing an internal path is now equivalent to finding the shortest path between two vertices of a graph with N vertices, which can be done in O(N^2) time. Of course, we assume in this case that both A and B are vertices of P. We will next show how the use of a triangulation permits us to compute the internal path in O(N) time, without even having to restrict the points to be on the boundary of P. Note that since we know how to compute a triangulation of an N-gon in O(Nlog N) time, this result constitutes a significant improvement.

For the time being, we will assume that both A and B are vertices of P. We will see later on how we can easily dispense with this requirement. If A and B are vertices of the same triangle of T, it is clear that IP(A,B)$=AB$, so we may assume that this is not the case. In the following, we will say that an interior edge of T is *AB-crossing* if its endpoints u,v are such that A,u,B,v appear this order around the boundary of P. Let P^* be the polygon resulting of the removal from T of all the edges that are not AB-crossing (fig.12). We first prove a few technical lemmas.

Lemma 11: The internal path between A and B in P is identical to the internal path between A and B in P^*.

Proof: It suffices to show that IP(A,B) can only intersect AB-crossing edges. To see that, suppose that it intersects an interior edge e which is not AB-crossing. Since e partitions P into two polygons, one of them does not contain B, therefore IP(A,B) crosses e at least twice (once in each direction). If A^* (resp. B^*) is the first (resp. second) intersection, going from A to B, replacing the part of IP(A,B) from A^* to B^* by the segment A^*B^* will shorten the length of IP(A,B), which leads to a contradiction. □

Lemma 12: The internal path between A and B intersects every interior edge of P^* exactly once, and intersects no other edge in T.

Proof: The proof of Lemma 11 shows that IP(A,B) cannot intersect any interior edge more than once. On the other hand, we can easily prove by induction that since every interior edge of P^* partitions this polygon into two parts, neither of which contains both A and B, it must intersect IP(A,B) at least once. Putting this result together with Lemma 11 completes the proof. □

It is easy to compute P^* in O(N) time. To do so, consider every interior edge of T in turn, and if it is not AB-crossing, remove it from T along with the dangling sub-polygon, just created, that does not contain A or B. Let $L = \{a_1b_1,\ldots,a_pb_p\}$ be the interior edges of P^*, as they appear from A to B (fig.12), i.e., in the order in which they intersect IP(A,B) (Lemma 12). Note that it is straightforward to obtain L in O(N) time, once P^* has been computed. From now on, the term IP(x,y), with x,y, vertices of P^*, refers to the internal path between x and y with respect to either P or P^*. This is legitimate since the two paths are identical, as a simple generalization of Lemma 11 readily shows.

Lemma 13: For any i; $1 \le i \le p$, there exists a vertex v of P^* such that IP(A,a_i)$=$IP(A,v)$\cup U$ and IP(A,b_i)$=$IP(A,v)$\cup W$, where U and W are two convex, non-intersecting polygonal lines turning their convexity against each other, and running from v to a_i and b_i, respectively (fig.13).

Proof: Let C_1 and C_2 be two oriented curves originating at the same point. To carry the analogy with internal paths, we may further assume that neither is self-intersecting; we say that C_1 and C_2 have a *proper crossing* if, as we follow C_1 from its starting point, we encounter a point where C_2 intersects C_1, and actually switches from one side to the other. Fig.14.1 (but not fig.14.2) shows an example of a proper crossing. We next prove that for any three points

A,B,C in P, the two paths $IP(A,B)$ and $IP(A,C)$ never have any proper crossings. Suppose that they did; let a be the first point (starting at A) where $IP(A,B)$ and $IP(A,C)$ cease to coincide, and let b denote the next intersecting point. Since $IP(A,B)$ and $IP(A,C)$ take distinct paths from a to b, we may re-route either one to the other, since they must have exactly the same length. Iterating on this process will eventually cause all proper crossings to disappear, which proves the above fact. We can now establish Lemma 13 by induction on i. The initial case being trivial, we may directly assume that the lemma is true for all indices from 1 to i. Since the $a_m b_m$'s are triangulation-edges, we necessarily have $a_i = a_{i+1}$ or $b_i = b_{i+1}$, say, $a_i = a_{i+1}$, wlog. Thus, considering the path $IP(A,b_{i+1})$, we observe that since it does not have any proper crossings with either $IP(A,a_i)$ or $IP(A,b_i)$,

1. It must pass through their common point v.

2. Its vertices between v and b_{i+1} are vertices of U and W.

From 1, it results that we may concentrate on the path $IP(v,b_{i+1})$ instead of $IP(A,b_{i+1})$, since we obviously have $IP(A,b_{i+1}) = IP(A,v) \cup IP(v,b_{i+1})$. Next, we strengthen proposition 2 by proving that the vertices of $IP(v,b_{i+1})$ are vertices of U or W, but never of both at the same time. Indeed, suppose wlog that starting at v, the vertices of $IP(v,b_{i+1})$ are $t_1,t_2,\dots$, with t_1 through t_m lying on U and t_{m+1} on W. It follows that the angle $(t_m t_{m+1}, t_m t_{m-1})$ is under 180 degrees, therefore there is an obvious shortcut for $IP(v,b_{i+1})$, avoiding t_m (fig.15), which leads to a contradiction. Thus there are now two basic cases to consider, depending on whether $IP(v,b_{i+1})$ takes its vertices in U or W. In the former case, v will be relocated further ahead on U, whereas it will stay unchanged in the latter. The details are straightforward, so we may consider the proof of the lemma as complete. □

We are now ready to proceed with the algorithm for computing $IP(A,B)$. The method involves computing $IP(A,a_i)$ and $IP(A,b_i)$, for $i=1,\dots,p$, which we can do iteratively by using the results of Lemma 13. The procedure being trivial for $i=1$, we turn to the general step directly. As already mentioned, we have either $a_i = a_{i+1}$ or $b_i = b_{i+1}$, and we can assume wlog that $a_i = a_{i+1}$. Let $u_1,\dots,u_\alpha$ (resp. $w_1,\dots,w_\beta$) be the vertices of U (resp. W) from v to a_i (resp. b_i). The half-plane delimited by $a_i b_i$ on the side where b_{i+1} lies is partitioned into $\alpha+\beta+1$ regions, themselves delimited by the lines passing through

$$w_{\beta-1}u_\beta,\dots,w_1w_2,vw_1,vu_1,u_1u_2,\dots,u_{\alpha-1}u_\alpha$$

With this order, the regions appear sorted along the segment $a_i b_i$ from b_i to a_i, so that we can find the region which contains b_{i+1} by testing each of them in turn in this order, until we are successful (fig.16). This corresponds to unfolding W and possibly folding over U. If b_{i+1} lies in a pencil of the kind $(w_{k-1}w_k, w_k w_{k+1})$, we must simply remove $w_{k+1},\dots,w_\beta$ from W and reset β to $k+1$ and w_β to b_{i+1} (fig.16.1). If b_{i+1} lies in the pencil $(u_{j-1}u_j, u_j u_{j+1})$, however, we must set W to $u_j b_{i+1}$, remove $(v,u_1,\dots,u_{k-1})$ from U and finally set v to u_j (fig.16.2). All the other cases are similar and call for no further explanation. Since none of the vertices removed in these operations will ever be examined again (Lemma 13), both $IP(A,a_p)$ and $IP(A,b_p)$, hence $IP(A,B)$, will be computed in $O(N)$ time.

We generalize this result by allowing both A and B to lie anywhere inside P, and not only on the boundary. Let R (resp. S) be the triangle where A (resp. B) lies. If $R=S$, the problem is solved since $IP(A,B)=AB$. Otherwise, we can compute the chain of triangles P^* in exactly the same way as described above. Next, let $v_i v_j$ be the interior edge of R which $IP(A,B)$ crosses. We can replace R by the triangle $v_i v_j A$ without altering the path $IP(A,B)$. Applying the same treatment to S will make A and B become vertices of P^*, which allows us to call on the procedure described earlier to compute $IP(A,B)$. In conclusion, we can state our main result:

> **Theorem 14:** Let P be a simple N-gon, and assume that any triangulation of P is available. For any pair of points A,B in P, it is possible to compute $IP(A,B)$, the internal path between A and B, in $O(N)$ time, which is optimal in the worst case.

5.2. The all-internal-paths problem

The problem is to preprocess the polygon P so that a batch of queries of the kind:

What is the internal path between A and B?

can be answered optimally. The method described in the previous section grants an attractive balance between execution and preprocessing time, when only a few queries have to be handled at any given time. It is worst-case optimal, but not optimal in the strictest sense of the term, since all the vertices of P must always be examined for every query. As a result, the precomputation of all possible internal paths between vertices entails a prohibitive $O(N^3)$ cost. The goal which we set forth here is to preprocess P so that the computation of $IP(A,B)$ for any pair of vertices (A,B) requires only time proportional to the size of the output, i.e., the number of vertices in $IP(A,B)$. To achieve this goal, we use the concept of visibility introduced earlier. Let $V(A)$ be the visibility polygon of A. If $IP(A,B)=AB$, B is a vertex of $V(A)$, otherwise $V(A)$ has a ray-edge (i.e., an edge vw such that v lies on Aw), with the property that vw separates A from B by intersecting $IP(A,B)$. More precisely, vw is the unique edge of $V(A)$ such that either A,v,B,w or A,w,B,v occur in clockwise order (fig.17). Since $V(A)$ is star-shaped, and vw is a ray-edge which is traversed by $IP(A,B)$, v must be the first vertex of $IP(A,B)$ after A. Indeed, there would be a shortcut if $IP(A,B)$ cut vw at any other point. Consequently, we have the relation $IP(A,B)= Av \cup IP(v,B)$ This motivates the introduction of the function $F(A,B)=B$, if $IP(A,B)=AB$, and $F(A,B)=v$ otherwise. Theorem 10 shows that if a triangulation of P is available, the visibility polygon $V(A)$ of each vertex A of P can be obtained in $O(N)$ time. The knowledge of $V(A)$ permits us to set up the array

$$D(A) = \{ F(A,v_i);\ i=1,\dots,N \}$$

in $O(N)$ time, with $O(N)$ storage, from which we conclude:

Theorem 15: Let P be a simple polygon with N vertices. It is possible to preprocess P in $O(N^2)$ time, using $O(N^2)$ space, so that for any pair of vertices A,B, the path $IP(A,B)$ can be computed optimally, i.e., in time proportional to the size of the output.

Proof: Compute the N arrays $D(v_1),...,D(v_N)$, forming an $N \times N$ matrix $\{F(v_i,v_j\}$, so that $IP(A,B)$ can be computed by retrieving $F(A,B)$ in constant time, and computing $IP(F(A,B),B)$ recursively. □

5.3. The internal-length problem

Imagine that an island with only inland communications is to be serviced by some utility (water tank, power station, fire house, police station, hospital, etc...). An interesting piece of information which may be needed is an upper bound on the internal path length between any pair of points. Let A^*,B^* be the two vertices of P which form the longest path $IP(A^*,B^*)$. We call $|IP(A^*,B^*)|$ the *internal length* of P. It is easy to determine A^* and B^* by trying out all possible pairs of vertices and using the matrix F of the previous section, given that the longest path can always be assumed to be found between two vertices of the polygon. This leads to an $O(N^3)$ running time, which we can cut down to $O(N^2)$ by proceeding as follows: Let $D(A,B) = |IP(A,B)|$. We will compute $D(A,B)$ iteratively by summing up partial distances obtained from F. In order to avoid duplicating computations, as soon as $D(A,B)$ is available, we backtrack along the path just followed in F to record the partial results. This ensures that, on average, one value $D(A,B)$ will be computed at every other step, which leads to an $O(N^2)$ algorithm.

INTDIST

- Initially, each $D(A,B)$ is set to -1 for $A \neq B$, and to 0 for $A = B$.

```
for all i ($1 \le i \le N$)
    for all j ($1 \le j \le N$)
        begin
            $Q \leftarrow \{v_i\}$
            $x \leftarrow v_i$
            while $D(x,v_j) = -1$
                begin
                    $x \leftarrow F(x,v_j)$
                    $Q \leftarrow Q \cup \{x\}$
                end
            if $Q$ has more than one element
                then
                    Let $Q = \{x_1,...,x_p\}$
                    $L \leftarrow D(x_p,v_j)$
                    for k = p-1,...,1
                        begin
                            $L \leftarrow L + |x_k x_{k+1}|$
                            $D(x_k,v_j) \leftarrow L$
                        end
    end
```

$D(A^*,B^*) = \text{Max} (D(v_i,v_j) \mid \text{all pairs of vertices } v_i,v_j)$

Since we can compute a triangulation of P in $O(N\log N)$ time, we may conclude:

Theorem 16: It is possible to determine the internal length of a simple N-gon as well as the corresponding internal path in $O(N^2)$ time.

6. Conclusions and future research

The decomposition principle in geometry expresses the feasibility of local treatments for the solution of general problems on arbitrary figures. The polygon-cutting theorem presented in this paper asserts the applicability of this principle in the case of simple polygons, and by doing so, leads to efficient, simple divide-and-conquer methods for solving a variety of geometric problems. The merit of this approach lies primarily in the versatility of its applications as well as in the increased efficiency which it affords. The most immediate open question is whether sorting the vertices in preprocessing is indeed required. This boils down to one of the most crucial problems of computational geometry: Is it possible to triangulate an N-gon in less than $(N\log N)$ time? In crude words, does the knowledge of a simple path among N points buy us anything? The answer is *yes* for computing convex hulls, but is still unknown for other problems like the one at hand. In this paper, we have deliberately chosen simplicity and practicality over generality by restricting the weights attached to the vertices to take on the values 0,1. There is no difficulty, however, in extending the theorem to a more general weight function.

REFERENCES

[CH80] Chazelle, B.M.
Computational geometry and convexity, PhD thesis, Yale University, 1980. Also available as CMU Tech. Rept. CMU-CS-80-150, July 1980.

[CD80] Chazelle, B.M., Dobkin, D.P.
Detection is easier than computation, Proc. 12th SIGACT Symp., Los Angeles, 1980.

[EA81] El Gindy, H., Avis, D.
A linear algorithm for computing the visibility polygon from a point, Journal of Algorithms, 2, 186-197 (1981).

[GJ78] Garey, M.R., Johnson, D.S., Preparata, F.P., and Tarjan, R.E.
Triangulating a simple polygon, Info. Proc. Lett., Vol. 7(4), June 1978, pp. 175-179.

[HS55] Hall, D.W., Spencer, G.
Elementary topology, Wiley, New York, 1955.

[KN73] Knuth, D.E.

The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Mass., 1973.

[LE80] Lee, D.T.

On finding the convex hull of a simple polygon, Tech. Report# 80-03-FC-01, Dept. of EE&CS, Northwestern University, 1980.

[LT77] Lipton, R.J., Tarjan, R.E.

A separator theorem for planar graphs, Waterloo Conference on Theoretical Computer Science, August 1977, pp. 1-10.

[LT77] Lipton, R.J., Tarjan, R.E.

Applications of a planar separator theorem, Proc. 18th IEEE FOCS Symp., Providence, RI, 1977, pp. 162-170.

[NS79] Newman, W.M., Sproull, R.F.

Principles of interactive computer graphics, McGraw-Hill, 2nd ed., 1979.

[SC78] Schachter, B.

Decomposition of polygons into convex sets, IEEE Trans. on Computers, Vol. C-27, 1978, pp. 1078-1082.

[SM77] Shamos, M.I.

Problems in computational geometry, Carnegie-Mellon University, 1977.

[SH76] Shamos, M.I., Hoey, D.
Geometric intersection problems, Proc. 17th Annual IEEE Symp. on Foundations of Computer Science, 1976.

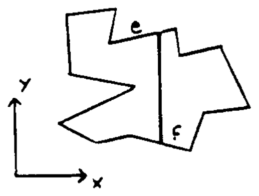

Figure 1

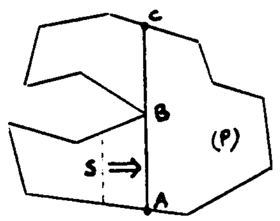

Figure 2

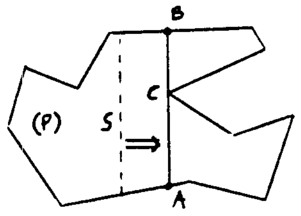

Figure 3

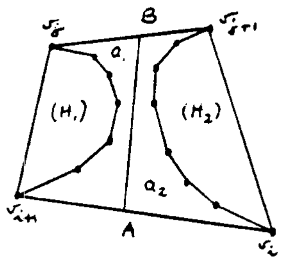

Figure 4

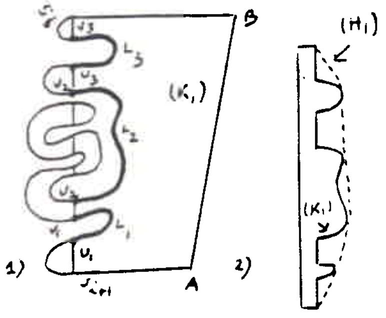

Figure 5

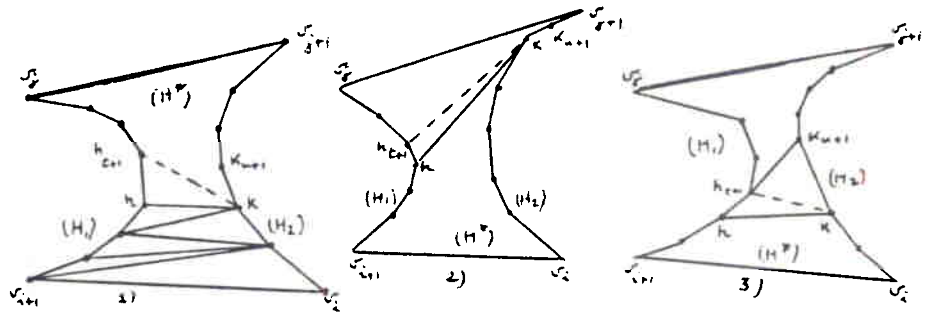

Figure 6

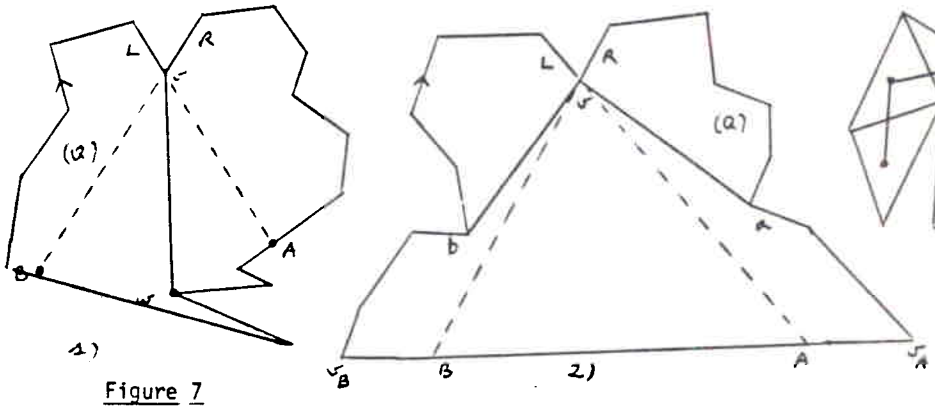

Figure 7

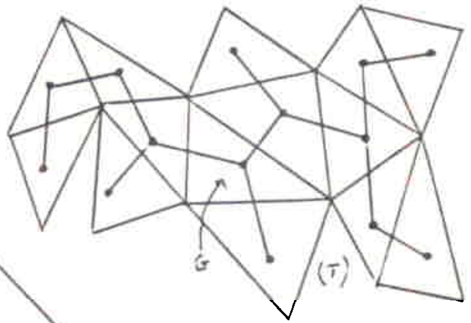

Figure 8

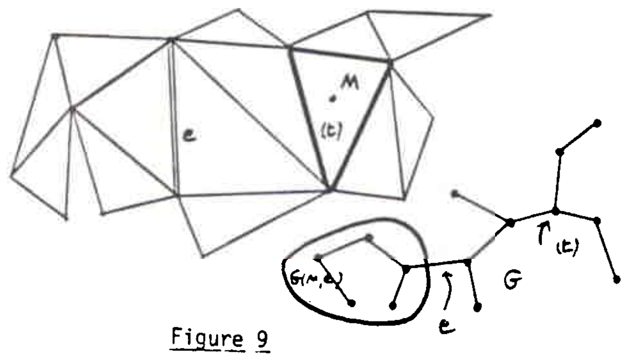

Figure 9

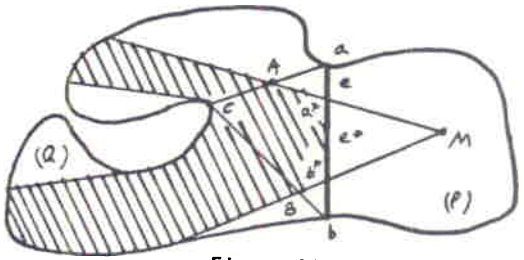

Figure 10

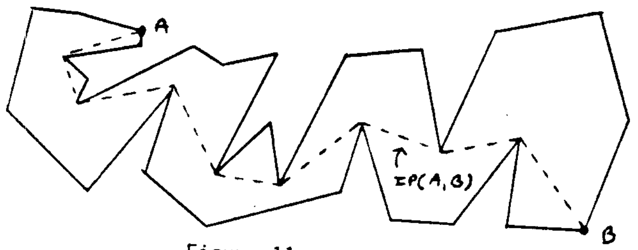

Figure 11

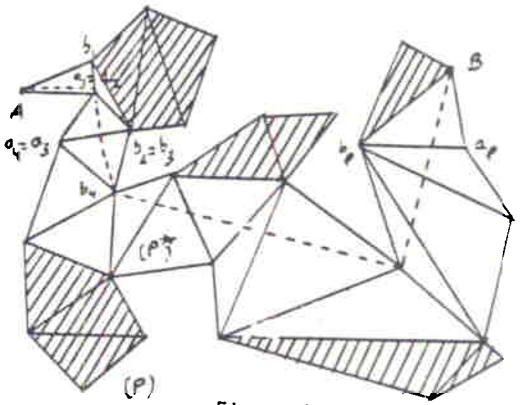

Figure 12

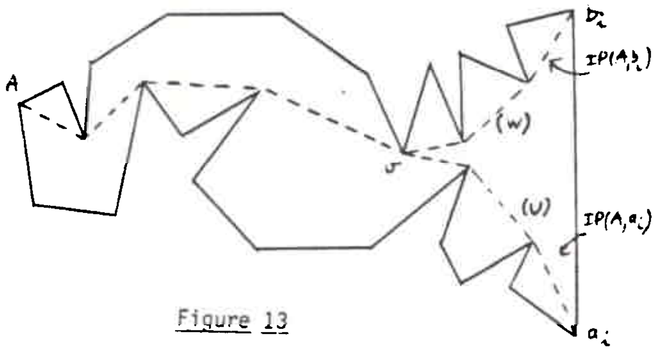

Figure 13

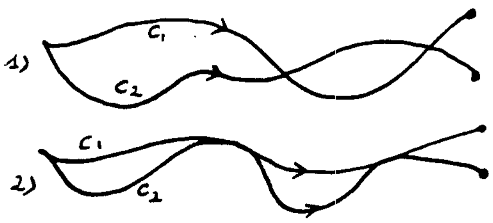

Figure 14

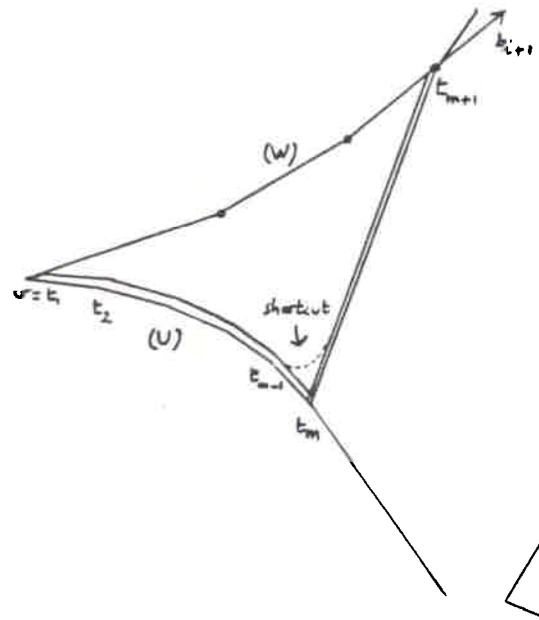

Figure 15

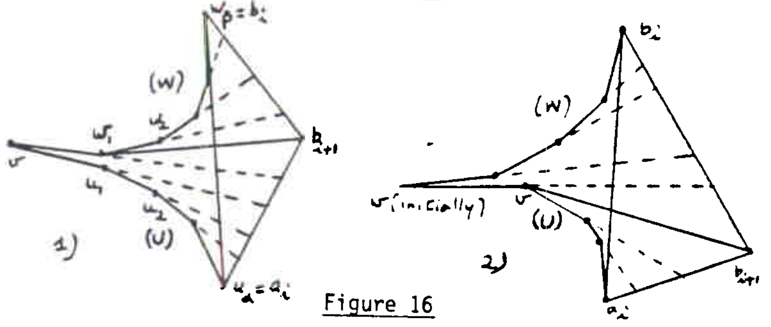

Figure 16

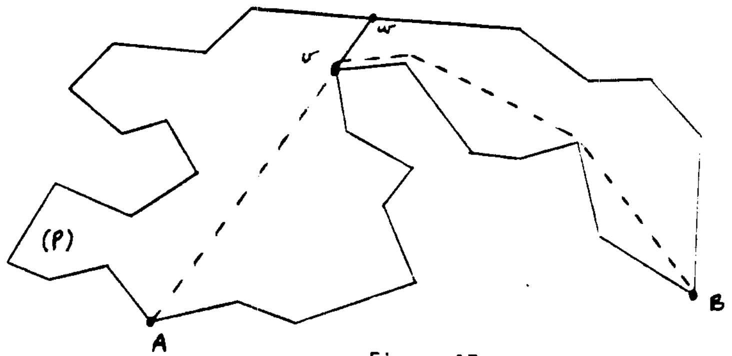

Figure 17