COMPUTATIONAL GEOMETRY, G.T. Toussaint (Editor)

OPTIMAL CONVEX DECOMPOSITIONS

by

Bernard Chazelle and David P. Dobkin

Abstract:

The problem of decomposing a non-convex simple polygon into a minimum number of convex polygons is solved. The decomposition allows for the introduction of Steiner points. Two algorithms are proposed. The first verifies that the problem is doable in polynomial time. The second provides an efficient method. Along the way, numerous results of independent interest in pure geometry as well as geometric complexity are stated.

1. Introduction

The problem of decomposing a simple polygon into its basic components has been a recurrent theme in computational geometry. Interest in this problem comes from its central location in the study of object representation. In the same way that English words benefit "greatly"

This research was supported in part by NSF grants MCS 83-03925 and MCS 83-03926, the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786.

Authors' addresses: Bernard Chazelle is with the dept. of Computer Science, Brown University, Providence, RI 02912. David Dobkin is with the dept. of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08544.

from their expressibility in a 26-letter alphabet, complex geometric structures are more easily handled when decomposed into simpler structures. Think of tool designers pressed for simple, modular designs, for instance. Another good example is the recognition of Chinese characters by matching text data against building blocks, as described in [8]. This operation involves the decomposition of polygonal shapes into convex pieces. For further motivation on decomposition problems, see [8,9,18,20]. The problem we consider is:

Given a simple polygon P, what is the minimum number of convex polygons which form a partition of P ?

This is called the OCD problem (for "optimal convex decomposition"). We will briefly review the known results. The first breakthrough on the OCD problem appeared in the proceedings of the 1979 STOC Symp. [5]. There, these authors proved that the problem was polynomial, thereby frustrating widespread suspicion that it was NP-hard. Interestingly enough, this finding was followed by a stream of NP-hardness results for similar problems [12—17]. For example, it was shown by Lingas that the presence of holes in the polygon P was sufficient to make the OCD problem NP-hard [14].

Variants of the OCD problem were considered in [12,13], where the requirement was made that no new vertices should be introduced in the partition of P. In [15,16] the objective function to minimize was no longer the number of pieces but the total edge-length. For other work, consult [1,3,4,8,9,14,18]. Following the original paper of Chazelle and Dobkin [5], the OCD problem was thoroughly treated in the former author's PhD thesis [3], where an $O(n+c^3)$ time decomposition algorithm was presented (n is the number of vertices of P and c is the number of reflex angles). This result represents a quantitative (but not qualitative) improvement over the $O(n^6)$ time algorithm of [5]. Of course it can be argued that for small values of c, the algorithm in [3] is linear or quasi-linear. Unfortunately this statement must be tempered by the rather intricate nature of the algorithm, which makes it an unlikely candidate for efficient implementation.

The purpose of this paper is to present the main ideas of the algorithms in [5] and [3]. We recognize that the interest of our results is primarily theoretical, so we will devote most of our effort to proving that the OCD problem is polynomial. The rest of the paper will outline

the most interesting points of the $O(n + c^3)$ time algorithm, referring to [3] for details and complementary information. The paper will be organized as follows: in this section, we give a brief overview of our method. In the next section, we present the basic geometric facts for the study of the OCD problem. In Section 3, we consider the algorithmic aspect of the problem and describe the polynomial algorithm for the OCD problem. In the following section, we address implementation issues and outline new lines of attack to speed up the algorithm. Finally we give conclusions and outline directions for further research in Section 5.

One methodological note is in order. Given the intricacy of our $O(n+c^3)$ algorithm for the OCD problem, the presentation will follow a top-down approach. We present the main ideas of the method first, and then fill in the blanks left. Our rationale is to separate the essential components of the algorithm and the parts which only contribute to its efficiency.

Two simple facts bound all algorithms for this problem. First, each notch (i.e. vertex displaying a reflex angle) *can* be removed by the addition of a polygon to the decomposition. Second, *at most* two notches can be removed through the addition of a single polygon. Hence, the minimum number of convex parts always lies between $\lceil c/2 \rceil + 1$ and $c + 1$. To extend these simple observations, however, is a difficult mathematical problem. To form minimal decompositions additional (Steiner) points must be introduced as vertices of newly generated polygons. This removes the obvious finiteness of the problem and makes simple enumerative procedures impossible. Furthermore the problem cannot be treated in a local manner. These observations led to the conjecture that the problem was NP-hard.

To circumvent these difficulties, we introduce X-patterns, from which minimal decompositions can be generated. An X_k-pattern is a particular interconnection of k notches which removes all reflex angles at the k notches and creates no new notches. A decomposition obtained by applying p patterns of type $X_{i_1}, \ldots, X_{i_p}$ along with straight-line segments to remove the remaining notches can be shown to yield $c + 1 - p$ convex parts. Clearly, decompositions with the most X-patterns also minimize the number of convex polygons. This can be viewed as a generalized matching problem which might lend itself to a dynamic programming approach. Simple examination shows that there is exactly one type of each X_k-pattern for $k = 2, 3$. Were

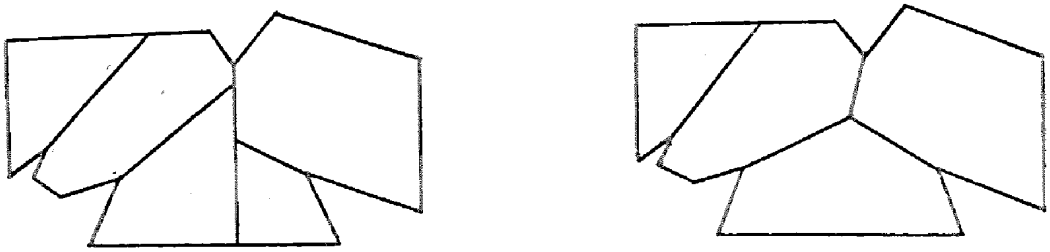


Figure 1 The naive decomposition and an improvement.

this the case for larger k, a polynomial-time algorithm based upon X-patterns would be possible. Unfortunately, determining whether a given set of notches can be interconnected via an X-pattern appears too involved to handle directly.

One solution is to constrain X-patterns in such a way that their detection becomes tractable. This leads to the introduction of Y-patterns, which we can regard as X-patterns endowed with some structural property. We show that with the exception of X_4-patterns any X-pattern can be advantageously replaced by a Y-pattern. Since Y-patterns can be constructed in polynomial time via dynamic programming, we can achieve our first goal, which is to show that the OCD problem is in P. As is shown later on, further geometric analysis leads to substantial gains in the efficiency of the original algorithm.

2. The Geometric Ingredients

In this section, we introduce our notation and show that the OCD problem can be reduced to a form of generalized matching problem. Let P be a simple polygon with n vertices, $p_1, \ldots, p_n$, in clockwise order. As previously mentioned, the vertices of P which display a reflex angle, called *notches*, will play a crucial role in the following. Let $v_1, \ldots, v_e$, be the list of notches in P, given in clockwise order. Throughout this paper, we will use the following convention on the representation of angles. Let ab and ac be two non-collinear line segments. $\angle(ab, ac)$ denotes the angle between 0 and 360 degrees swept by a counterclockwise rotation from ab to ac.

A *decomposition* of P is a set of polygons, $P_1, \ldots, P_k$, whose union gives P, and such that the intersection of any two if non-empty consists totally of edges and vertices. A decomposition is said to be *convex* if all its polygons are convex. We define an *optimal convex decomposition* of P, or OCD for short, as any convex decomposition of minimum cardinality.

We define the *naive decomposition of* P as the set of polygons obtained by removing each notch in turn by means of a simple line segment *naively* drawn from the notch. To be more precise, a naive decomposition of P is obtained by going through each notch $v_1, \ldots, v_e$ in turn, extending a line segment from v_i until we first hit another line already in the decomposition.

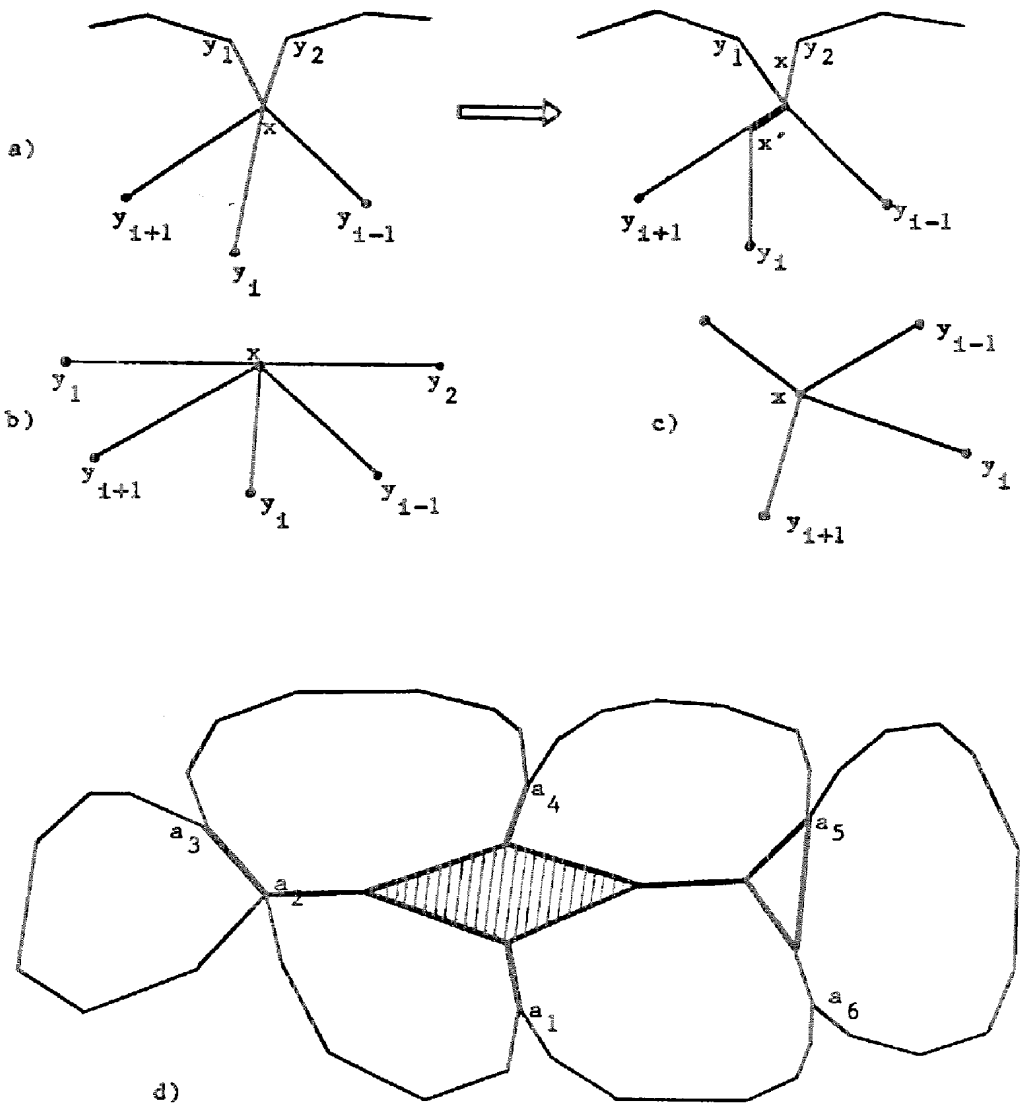


Figure 2 a,b,c) Satisfying degree requirements for X-decompositions.
d) Removing interior polygons.

Of course, the extended line should remove the reflex angle at v_i. Note that in this way we will also guarantee that all lines drawn always lie entirely inside P. To simplify the ensuing analysis, we will ensure that the degree of each vertex in the decomposition does not exceed 3 and that no segment in the naive decomposition connects two notches. These conditions are trivially always satisfiable. Figure 1 illustrates the notion of naive decomposition and shows in particular that it is not always minimal. We have the following (trivial) result.

Lemma 1. Any naive decomposition of P produces exactly $c + 1$ convex parts.

Next we wish to characterize a convenient class of decompositions to which we will restrict our attention in the following. We say that a polygon is *interior* to P if it lies inside P and at most a finite number of its points lie on the boundary of P.

Definition 1. An *X-decomposition* is any convex decomposition containing no interior polygons and such that no vertex is of degree greater than 3, except for the notches, which may be of degree at most 4.

Lemma 2. The class of X-decompositions always contains an OCD.

Proof: Consider an OCD which is not an X-decomposition. We transform its edges to yield an X-decomposition. First, we show how to satisfy the degree requirements. Since this process may introduce interior polygons, we show how to remove interior polygons without increasing the degree of any vertex.

1. We regard the decomposition as a planar graph consisting of *boundary edges* and *added edges* (an edge is said to be added if it does not lie on the boundary of P). Let x be a notch of degree greater than 4 and $y_1, \ldots, y_m$ $(m > 4)$ be its adjacent vertices, with $y_1, y_2 \in$ boundary of P. It is trivial to show that there exists $i > 2$ such that $\angle(xy_{i+1}, xy_{i-1}) < 180$ (Fig.2-a). We can then move xy_i along xy_{i-1} or xy_{i+1} to form a new segment $x'y_i$ with x' chosen close enough to x so as to preserve convexity. We iterate on this process until the notch x becomes of degree 4. The other cases to consider are depicted in Fig.2-b,c. In case b) the vertex x is not a notch but still lies on the boundary of P (it may or may not be a

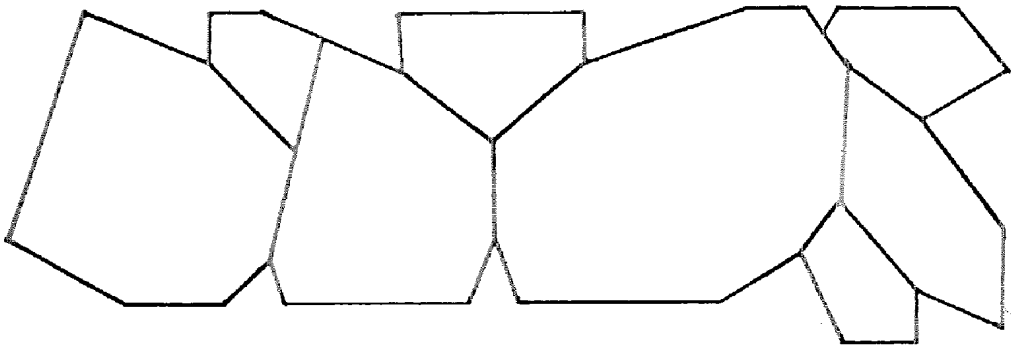


Figure 3 An X-decomposition involving an X3- and an X4-pattern.

vertex of P). In case c) x is a vertex of degree > 3 which does not lie on the boundary of P. The same method used in case a) will reduce the degrees accordingly.

2. Consider the subgraph H formed by the added edges of the OCD under consideration. Pick an interior polygon of the OCD and let G be the connected component in H to which the edges of this polygon belong (Fig.2-d). Let $a_1, \ldots, a_k$ denote the vertices of G (in clockwise order) on the boundary of P and let K be the graph obtained by removing G from the graph of the OCD. Since G lies in P and is a connected component of H, it lies entirely in one face of K, denoted Q. We observe that all the a_i's lie on the boundary of Q. Since G is connected, it determines at least k faces in K, aside from the interior polygon(s). This shows that the OCD has at least $k+1$ faces in Q. The polygon Q may not be convex, but since we had a convex decomposition of P before removing G, all the notches of Q must be notches of P, i.e. must be some of the a_i's. Now, instead of keeping the convex decomposition of Q induced by the OCD, we apply the naive decomposition to it. This will yield at most $k+1$ polygons (exactly $k+1$, actually, since we are dealing with an OCD, and consequently no transformation can *improve* the decomposition). Since the boundary of each of the created polygons contains at least one of the points a_i as a vertex, and each a_i belongs to at most two polygons, none of these polygons can be interior to P. Furthermore it follows from the definition of the naive decomposition that the desired degree constraints will be preserved. Iterating on this process for each of the remaining interior polygons completes the proof. ∎

We are now ready to introduce the important notion of ***X-pattern***. Once again we regard the added edges of an X-decomposition as forming a subgraph of the total decomposition. From the definition of X-decompositions, it follows that the subgraph is a forest of trees with each node having degree 1 or 3 except for the notches which may have degree 1 or 2. We will pay special attention to those trees where all vertices of degree 1 or 2 are notches of P.

Definition 2. A planar embedding of a tree lying inside P is called an X-pattern if it is not self-intersecting and:

1. All vertices are of degree 1, 2 or 3.
2. Any vertex of degree 1 or 2 coincides with a notch of P, and its (1 or 2) adjacent edges remove its reflex angle.
3. None of the 3 angles around any vertex of degree 3 is reflex.

An X-pattern with k vertices of degree 1 or 2 is called an X_k-pattern. Vertices of degree 1,2,3 are called N1, N2, N3-nodes, respectively. For simplicity we refer to the vertices of degree 1 or 2 as the notches of the X-pattern. Informally, an X_k-pattern is an interconnection of k notches used to remove them while introducing $k-1$ additional polygons into the decomposition. Figure 3 gives an example of an X-decomposition with one X_3-pattern and one X_4-pattern. We justify the introduction of X-patterns with the following observation.

Lemma 3. An X-decomposition with p X-patterns has at least $c+1-p$ convex parts.

Proof: Let S, t, k be respectively the number of polygons, trees, and tree-vertices lying on the boundary of P. We prove the relation

$$S = k - t + 1 \tag{1}$$

by induction on t. The case $t = 1$ is trivial, so we may assume that the introduction of $t-1$ trees involves k_1 vertices on the boundary of P and creates $S_1 = k_1 - (t-1) + 1$ polygons. Introducing the last tree into the decomposition will account for exactly $k - k_1 - 1$ additional polygons, leading to a total of $S = S_1 + k - k_1 - 1 = k - t + 1$ polygons and proving (1). Each of the $t - p$ trees which are not X-patterns has at least one distinct vertex which lies on the boundary of P and is not a notch. This implies that $t - p \leq k - c$, which alongside (1) establishes the lemma. ∎

Lemma 3 suggests that using p X-patterns saves at most p polygons over the naive decomposition. This leads to the definition of *compatible* X-patterns. A set of X-patterns is said to be *compatible*, if no pair of edges taken from two distinct patterns intersect. For example, the X-patterns in any X-decomposition always form a compatible set. Conversely, we can show that any set of compatible X-patterns can be used to produce a decomposition. The following result is complementary to Lemma 3.

Lemma 4. Any compatible set of p X-patterns can be used to produce an X-decomposition with exactly $c + 1 - p$ convex parts.

Proof: Start the decomposition with the p X-patterns. This will produce a certain number of polygons. If any of them is not convex, apply the naive decomposition to it. Straightforward analysis shows that the final number of polygons will be exactly $c + 1 - p$. ∎

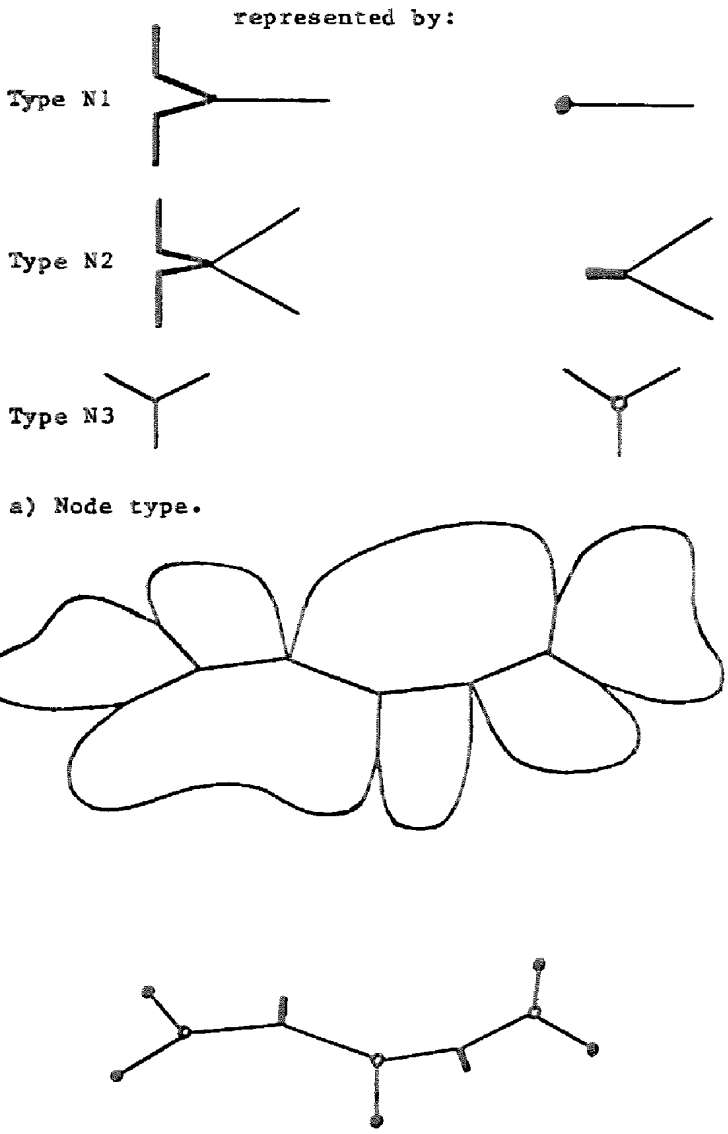
From Lemmas 2, 3 and 4, we are able to express the original OCD problem as a generalized matching problem.

Lemma 5. Let p be the maximum number of compatible X-patterns. An OCD can be obtained by first applying the p X-patterns and then applying the naive decomposition to any remaining non-convex polygon.

Since any X-pattern has at least two notches, the previous result shows that an OCD consists of at least $1 + \lceil c/2 \rceil$ polygons. Lemma 5 suggests a new line of attack for the problem at hand —the sufficiency of computing a maximum set of compatible X-patterns. Unfortunately to do so seems beyond reach, given the excessive number of candidates we might have to consider in the process. X-patterns allow Steiner points (i.e. vertices not on the boundary of P) to be adjacent. Looking at any X-pattern as a mechanical system of extendible arms and joints, this corresponds to a system which is not strongly constrained. We show next that X-patterns can be in general *reduced* to maximally constrained X-patterns, called Y-patterns. The next step will be to prove that Y-patterns (being maximally constrained) can be computed in polynomial time. A rigorous definition of these notions is now in order.

Definition 3. A Y_k-pattern is an X_k-pattern such that

1. no edge joins two nodes of type N3.
2. in any path containing three consecutive nodes of respective type N2,N3,N2, the N2-nodes lie on opposite sides, i.e. the two pairs of edges of P which emanate from the N2-nodes lie on opposite side of the path.



a) Node type.

b) a Y7-pattern and its representation.

Figure 4

We will use a special representation for N1, N2 and N3-nodes (Fig.4-a). Note that the N3 nodes of a Y-pattern are its Steiner points. A Y_7-pattern and its representation are given in Fig.4-b. To help visualize Condition 2, observe that if the two N2-nodes were pointing downward, we would not have a Y-pattern.

We next show that all X_k-patterns ($k \neq 4$) can be transformed into Y-patterns. This allows us to limit attention to X-decompositions with only Y and X_4-patterns. The transformations, called *reductions*, involve the stretching, shrinking, and rotating of lines in the original pattern. Reductions involve sequences of steps with each step translating an N3-node from one point to another. All edges in the pattern except for the three edges adjacent to the N3-node remain fixed. Reductions stop before an angle in the pattern becomes reflex or an edge in the pattern strictly intersects an edge on the boundary of P. During a reduction, a tree remains an X-pattern. It may however gain or lose vertices in the process. For example, Figure 5 shows the reduction of an X_3-pattern, which might correspond to one step in the reduction of a more complex X-pattern.

In Figure 5, we see how an X_k-pattern may be reduced to an X_l-pattern ($l < k$). The final tree is still an X_3-pattern; it can also be viewed as an X_2-pattern augmented with a segment provided by the naive decomposition. The X_3-pattern loses a notch thereby being reduced to an X_2-pattern. This is legitimate since from lemma 5 we know that numbers (rather than types) of compatible X-patterns matter. In Fig.5, observe that the notches a and b cannot possibly be interconnected by an X_2-pattern in *any* X-decomposition. However, it is true that an OCD can be obtained by considering the X_2-pattern as the single element of a maximal set of compatible X-patterns. In this regard, the X_2-pattern is of interest to us. If X-patterns can lose notches, they can also gain some, as will be soon shown. This should not surprise us, however, since the previous idea of applying X-patterns and then completing the process with the naive decomposition may also augment the X-patterns with additional edges (while not increasing the number of patterns).

Lemma 6. In an X-decomposition, any X-pattern which is not reducible to an X_4-pattern can be reduced to a Y-pattern.

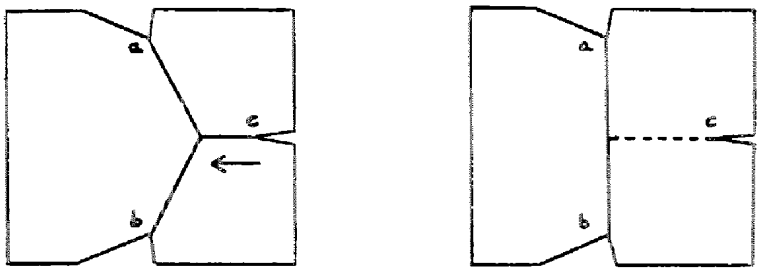
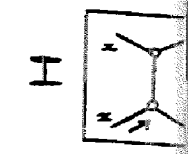
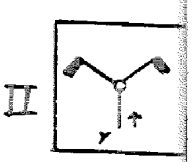
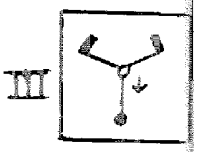


Figure 5 Reductions on X-patterns.







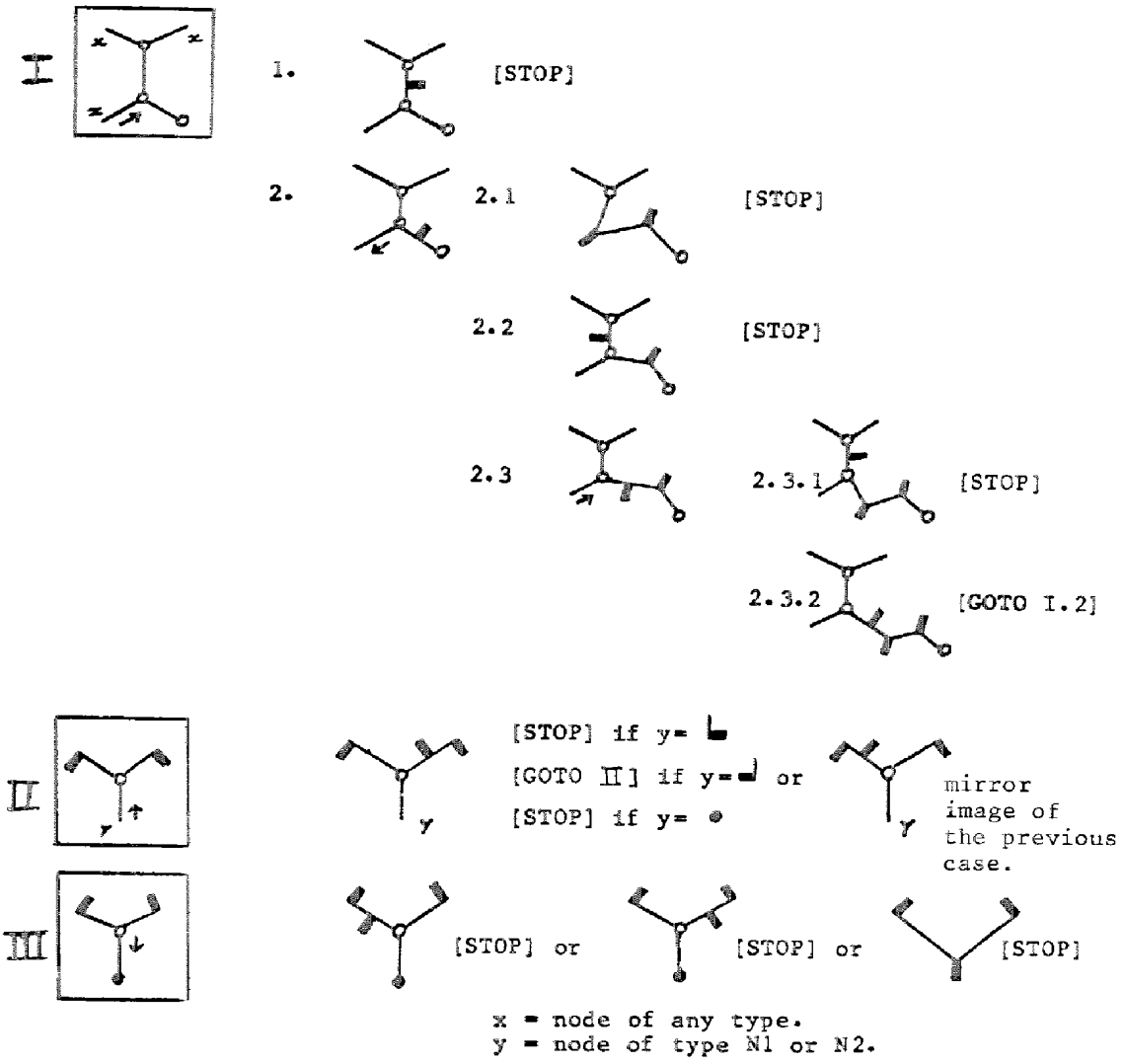


Figure 6 The proof of Lemma 6.

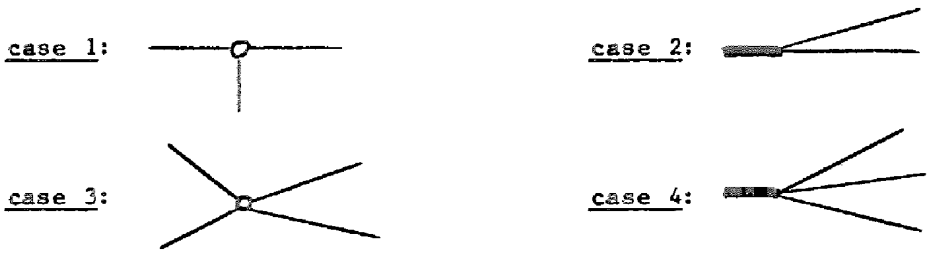


Figure 7 Extreme instances of X-pattern vertices.

Proof: Befo
into Y-patte
patterns in
If a reductio
not on the
definition of
will not occ

We are
condition 1

Condition 1

Figure
is not of typ
types in Fig
arrow. The
boundary or
being that w
done. Other
process until
current redu
Convergence

Note th
of X-pattern
representing
the reflex an
edge from th
all angles.
will be remo
patterns doe

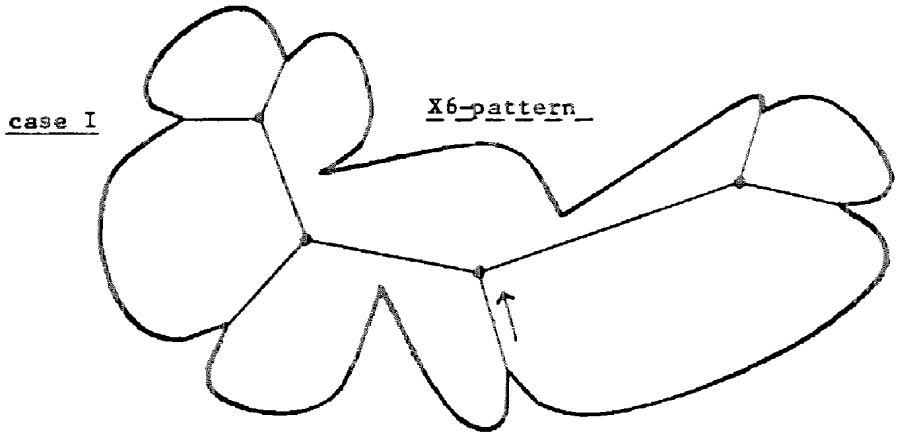
Proof: Before describing the appropriate sequence of reductions which will turn X-patterns into Y-patterns, we must ensure that reductions can be carried out freely without merging two patterns in the process, which might increase the number of polygons in the decomposition. If a reduction brings an edge of an X-pattern in contact with any edge of the decomposition not on the boundary of P a reflex angle will have necessarily resulted beforehand. Since the definition of a reduction precludes intersections with the boundary and reflex angles, this case will not occur.

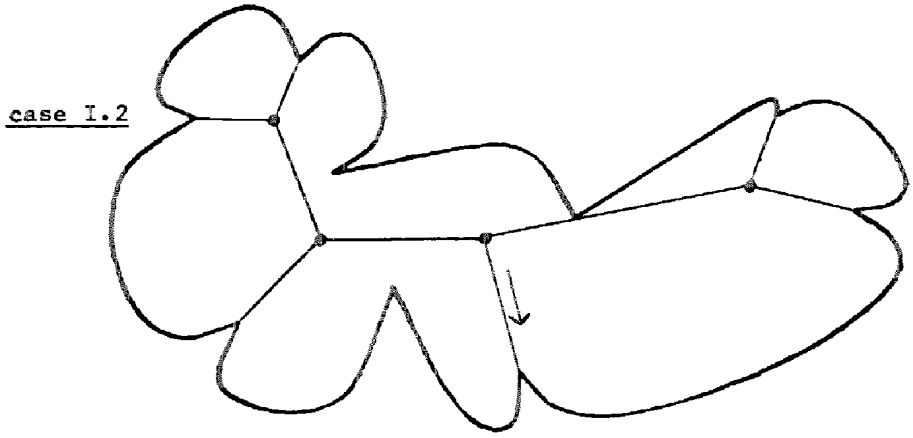
We are now ready to describe the reductions of undesirable X-patterns. We first ensure condition 1 of Definition 3.

Condition 1.

Figure 6-I) illustrates the sequence of actions. As indicated, we assume that the pattern is not of type X_4 but has two N3-nodes adjacent to each other. Recall the notation for node types in Fig.4. We move one of the N3-nodes by translation in the direction indicated by the arrow. The translation continues until either an N2-node results from intersection with the boundary or we fall into one of the "extreme" instances depicted in Fig.7. Assume for the time being that we are in the first case. If the N2-node occurs between the N3-nodes (case 1) we are done. Otherwise (case 2), another reduction leads to 2.1), 2.2) or 2.3). We then iterate on this process until no pair of N3-nodes are adjacent (the label STOP is meant to indicate that the current reduction step is over and that we should check again if more reductions are necessary). Convergence is guaranteed since each step adds a distinct N2-node.

Note that the figure investigates all cases except for those representing extreme instances of X-patterns. These extreme cases are illustrated in Fig.7. Case 2 is to be understood as representing two edges emanating from the same notch, one of which is sufficient to remove the reflex angle. To handle all four cases, we observe that in each of them we can prune one edge from the pattern along with the adjoining subtree and still preserve the non-reflexivity of all angles. Recall that the notches attached to the removed piece, although now unresolved, will be removed later on via the naive decomposition. The crucial observation is that pruning patterns does not affect the overall number of patterns in the decomposition. The pruning





(Figure 8 .../...)

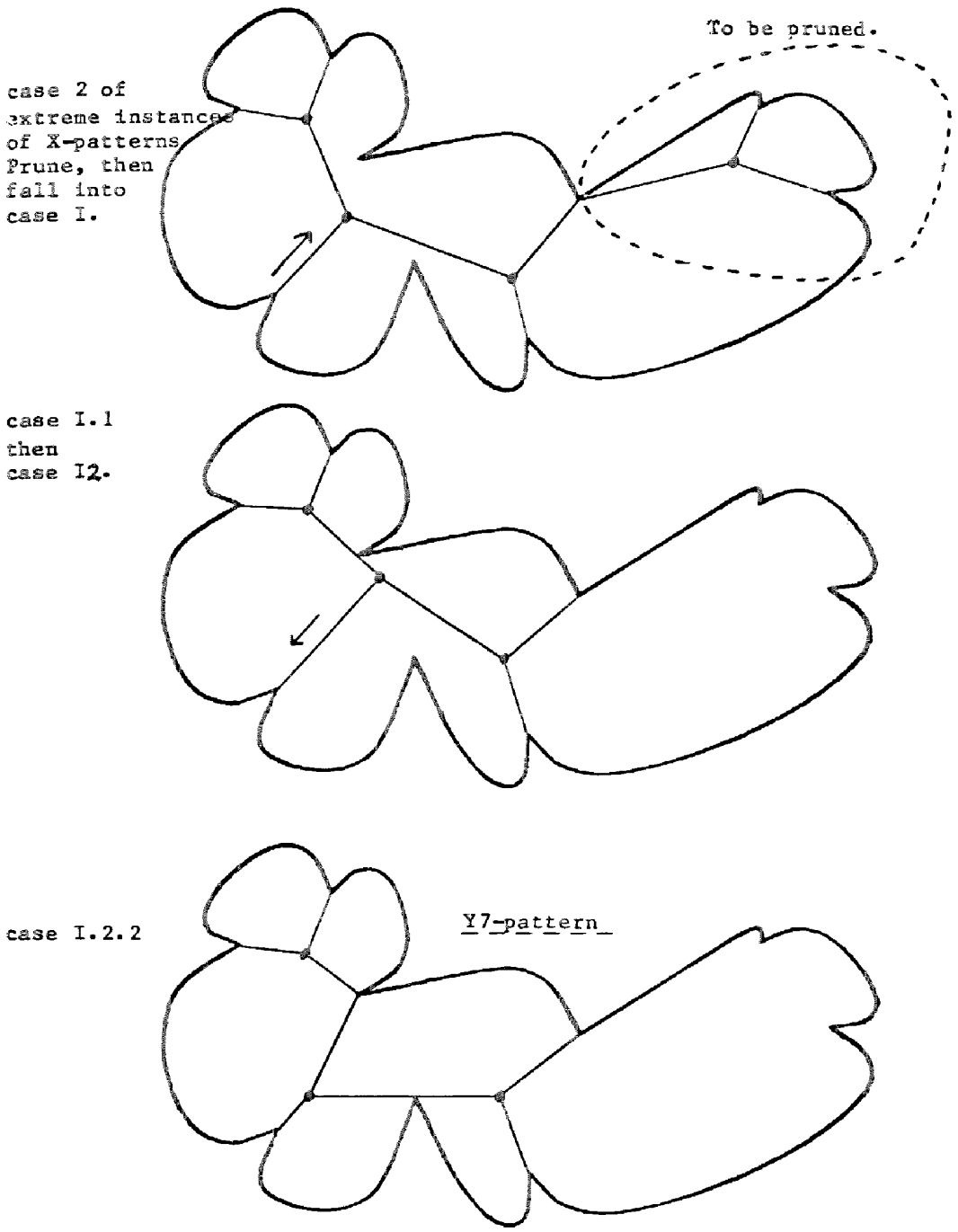


Figure 8 Example of reductions turning an X6-pattern into an Y7-pattern.

process will converge since every removal decreases the current number of N1-nodes by at least one. Unfortunately this statement is a little hasty, since case 2 of Fig.7 surely removes at least one N1-node but, unfortunately, also adds one. To establish convergence, we will have to show that when in case 2 the subtree pruned involves a single notch x, this notch can never be re-introduced by subsequent reductions. Let xy be the unique edge thus pruned. A simple observation shows that no further reduction of the pattern will ever bring any of its edges across the segment xy. For this reason x can never become an N2-node for the pattern. But being an N2-node is a prerequisite for becoming an N1-node again, therefore x is safely *lost* for the pattern once and for all —see [3] for a detailed proof of this fact.

Condition 2.

Once Condition 1 holds, we satisfy Condition 2 by following the instructions outlined in Fig.6-II,III). Proving convergence follows the lines given above and we do not elaborate on it.

The reductions shown in Fig.6 are to be applied iteratively to each X_k-pattern ($k \neq 4$) with adjacent N3-nodes. Convergence is straightforward. We have illustrated the complete reduction of an X_8-pattern in Fig.8. ∎

3. The Polynomial Time Algorithm

The previous section proved the existence of an OCD consisting solely of Y and X_4-patterns. In this section, we present a polynomial time algorithm for constructing such an OCD. For the sake of clarity, we will ignore efficiency issues, merely showing that every routine used in the algorithm runs in polynomial time. Later we will discuss efficient implementation.

We begin by constructing an oracle to answer questions of the form : "*Does there exist a pattern connecting k given notches ?*" in polynomial time. This oracle will be used by the decomposition algorithm.

Let v be an N3-node of a Y-pattern. Remove the three edges vv_i, vv_j, vv_k adjacent to v; the Y-pattern becomes a disconnected set of three subtrees, for which the removed edges play the role of an X_3-pattern. This leads us to introduce the notion of *extended* X-pattern. An

extended X_l-pattern ($l = 2, 3$) is either an X_l-pattern by itself or an X_l-pattern appearing as a subgraph of an X_m-pattern ($m > l$). It is clear that an algorithm for testing the possibility of an X_l-pattern between a given set of notches can also be used to determine the possibility of an extended X_l-pattern, as long as the angles formed at the notches by the subtrees of the X_m-pattern are known in advance.

To see this, we must define the notion of *extended notch* and *extended range*. Let v_i be a notch of the X_l-pattern, and let W be a wedge centered at v_i. We define the *extended range* of v_i as the set of points u such that v_iu lies entirely within both P and W. In essence, the extended range is the intersection of W with the visibility-polygon at v_i [7]. In general the wedge W will be taken as the locus of rays (i.e. half-lines) which emanate from v_i and remove the reflex angle at v_i. When dealing with ordinary X-patterns, the wedge W is simply determined by the edges of P adjacent to v_i. In this case the extended range is simply referred to as *range* of v_i, since it is then defined only with respect to v_i and P. In the case of extended patterns, however, the wedge will take into account the other edges already adjacent to v_i, and will thus be smaller than in the previous case. Later, we consider cases where the wedge W is taken as the entire plane. This is done if we do not wish to remove a reflex angle at a particular notch. In all cases, anyhow, we say that we *extend* the notch according to certain angular specifications.

Lemma 7. Checking for the possibility of an (extended) X_l-pattern between l given notches can be done in polynomial time (for $l \leq 4$).

Proof: For $l = 2, 3$ it is clear that an X_l-pattern will be possible if and only if 1) ($l = 2$) the (extended) range associated with each notch contains the other notch; 2) ($l = 3$) the (extended) ranges associated with the l notches have a common intersection point forming three non-reflex angles with respect to the notches. Computing each range can be done in $O(n)$ time by first computing the visibility-polygon [7], and then clipping it along the corresponding wedge. Next we compute the intersections of all ranges, which can be done naively in $O(n^2)$ time. To handle X_4-patterns, we first observe that two different kinds must be considered, as shown in Fig.9. Assume wlog that the two notches v_i and v_j are adjacent to the same N3-node. We successively apply the reductions shown in Fig.9-b-c, with respect to A then B. Assuming wlog that this

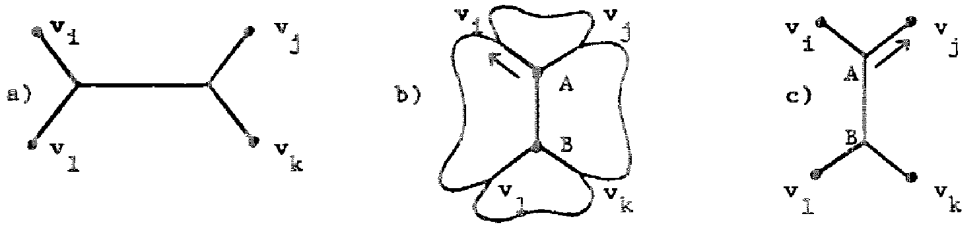


Figure 9 A simple method for computing X4-patterns.

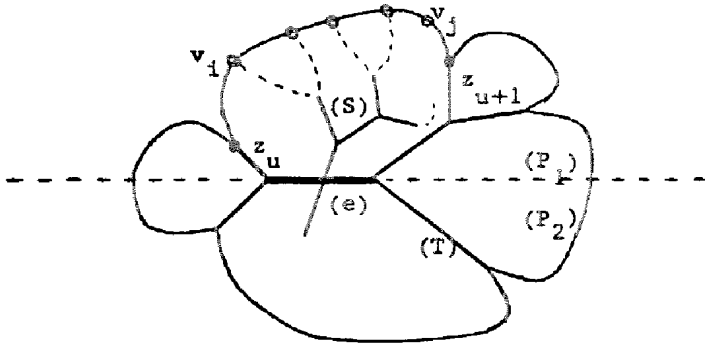


Figure 10 The interaction between X-patterns.

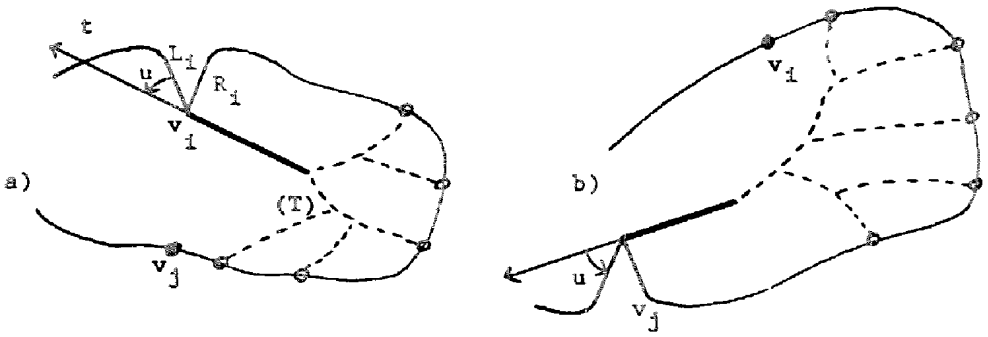


Figure 11 Defining B(i,j) and F(i,j).

does not create an X_3 or a Y_5-pattern with an N2-node between A and B, we simply observe that the four edges emanating from the notches will now be collinear with edges of P. This puts the total number of configurations within $O(c^4)$. We can therefore check whether any of them is feasible, which will lead to an $O(nc^4)$ time algorithm. ∎

We can now turn to the decomposition algorithm. The procedure for determining a maximum compatible set of X-patterns is based on dynamic programming. We rely upon the observation that if a certain X_4 or Y_4-pattern belongs to an OCD of P, it decomposes P into k subpolygons, $P_1, \ldots, P_k$, so that finding an OCD for each P_i yields an OCD for P. We compute maximal compatible sets of patterns for each P_i. Since the notches of P_i are also notches of P, any X-pattern of P_i is also an X-pattern of P. Conversely, we want to show that any X-pattern of P involving only notches in P_i is also an X-pattern of P_i. This is quite important. Dynamic programming proceeds bottom-up, so a maximal set of patterns involving notches of P_i must be found before we know the shape of P_i.

To solve this problem, we define $V(i,j)$ as the set of notches between v_i and v_j in clockwise order, so $V(i,j) = \{v_i, v_{i+1}, \ldots, v_j\}$, with index arithmetic taken modulo c. Let $z_1, \ldots, z_k$ be the notches of an X-pattern, T, given in clockwise order around the boundary of P, and let $V(i,j)$ be the notches of P between z_u and z_{u+1} in clockwise order ($z_u = v_{i-1}, z_{u+1} = v_{j+1}$). We will show that no X-pattern with all its notches in $V(i,j)$ can intersect T. Assume that an X-pattern S intersects an edge e of T. Consider the shortest segment which is collinear with e and has both of its endpoints on the boundary of P. This segment partitions P into two polygons P_1 and P_2 (Fig.10). Since the path of T between z_u and z_{u+1} is a convex polygonal line, it lies entirely in P_1 or P_2 (say, P_1). Since all the notches in $V(i,j)$ are notches of P_1, S must have notches in P_2, a contradiction. This independence result can be understood in combinatorial terms. It states that two X-patterns are intersection-free if and only if their notch sequences are not intermixed, i.e. one sequence falls completely between two consecutive elements of the other sequence.

We next define $S(i,j)$, for every pair of notches v_i, v_j, as a maximum compatible set of X_4 or Y-patterns in $V(i,j)$. To achieve our ultimate goal, i.e. evaluating $S(1,c)$, we compute all

values $S(i,j)$ from $\{S(k,l) \mid V(k,l) \subset V(i,j)\}$ using dynamic programming. This can be done directly if v_i and v_j are not connected to the same pattern. We simply test all combinations $\{S(i,k), S(k+1,j)\}$ for all $v_k \in V(i,j-1)$. Otherwise we have to distinguish whether v_i and v_j should be connected together via an X_4 or a Y-pattern.

To handle the latter case, we compute all Y-patterns which might belong to an OCD via dynamic programming. We compute Y-subtrees (i.e. subtrees of Y-patterns) as well as Y-patterns by patching Y-subtrees together. To prevent the number of computations from blowing up, however, we keep only the Y-subtrees that are candidates for belonging to an OCD. A Y-subtree is considered *not* to be a candidate if at the time it is computed we are ensured of the existence of at least one OCD which does not use this Y-subtree (although we may not know this OCD explicitly yet). As a shorthand we say that a pattern or a Y-subtree lies in $V(i,j)$ if all its notches do. It now remains to formalize the intuition given here and describe the polynomial time algorithm.

Consider a Y-pattern which is used in an OCD and has at least one N2-node, v_i. This node splits the Y-pattern into two Y-subtrees, so there exists an index j such that

1. One of the Y-subtrees lies in $V(i,j)$ whereas the other lies in $V(j+1,i)$.
2. All the other patterns in the OCD lie totally either in $V(i,j)$ or in $V(j+1,i)$.

We will consider the candidacy of the Y-subtree in $V(i,j)$ immediately after $S(i,j)$ has been computed. We first observe that if $v_i, v_{i_1}, \ldots, v_{i_m}$ is a list of its notches in clockwise order, we may dismiss the candidacy of the subtree if the following equality is not satisfied:

$$|S(i,j)| = |S(i+1, i_1 - 1)| + \ldots + |S(i_{m-1} + 1, i_m - 1)| + |S(i_m + 1, j)|, \qquad (2)$$

where $|S(k,l)|$ represents the number of patterns in $S(k,l)$.

Note that the last term in the right-hand side is to be ignored if $i_m = j$. If Relation (2) is not satisfied, the right-hand side is strictly smaller than $|S(i,j)|$. When considering candidate Y-subtrees, we have the idea in mind that only one pattern will have notches in both $V(i,j)$ and $V(j+1,i)$. It would then be unreasonable to use any Y-subtree which does not satisfy (2), since the patterns of $S(i,j)$ would provide a better decomposition altogether.

This simple fact will be consequential in achieving a polynomial time algorithm. We still have to cope, however, with the prospect of keeping an excessive number of candidate subtrees for each pair (i,j). Another geometric observation is in order. Let L_i (resp. R_i) denote the edge of P adjacent to v_i and preceding (resp. following) v_i in clockwise order. Whenever L_i or R_i is used in designating an angle, it is understood as directed outwards with respect to v_i. Let t be the edge adjacent to v_i in the Y-subtree lying in $V(i,j)$. The edge t is called the *arm* of the Y-subtree. When the arm of a Y-subtree enters the expression of an angle, we assume that it is directed towards the notch (here t is directed towards v_i). Among all the Y-subtrees in $V(i,j)$ for which v_i is an N2-node, (2) is true and $u = \angle(L_i,t) < 180$, we may keep the Y-subtree T which minimizes the angle u as the only candidate with respect to $V(i,j)$ (Fig.11-a).

We define $B(i,j)$ as a pointer to the arm of T. If there is no such subtree, $B(i,j)$ is 0. Patterns and subtrees will be represented by linked lists, so $B(i,j)$ will provide access to the entire Y-subtree T whenever necessary. Carrying out the same reasoning counterclockwise in $V(i,j)$ with now v_j as an N2-node, we define $F(i,j)$ in a similar fashion (Fig.11-b).

Having established our notation, we are now in a position to present the decomposition algorithm. We assume a function $\langle ARG \rangle$ for assembling Y-subtrees when computing $S(i,j)$. ARG is in general a pair of Y-subtrees taken from $B(u,v)$ or $F(u,v)$. If the two subtrees can be patched together and form a Y-pattern T, the function $\langle\rangle$ returns (C,T), where C is the maximum number of compatible patterns which can be applied in $V(i,j)$ including T. We return to a discussion of this function after a presentation of the algorithm. Before proceeding with a formal description, a brief overview might be helpful.

After all necessary preprocessing in STEP 1, we use nested loops to implement the dynamic programming scheme. Each step involves computing $S(i,j)$ for a given value of i and j. We start by computing the best Y-pattern which connects v_i and v_j (STEP 2). This involves patching precomputed candidate Y-subtrees. STEP 3 computes a maximum set of compatible patterns in $V(i,j)$, denoted L, assuming that v_i and v_j do not belong to the same pattern. Then we compute M, defined similarly, with the difference that we now allow the presence of an X_4-pattern connecting v_i and v_j. Finally the Y-pattern of STEP 2 (if any) is used to compute N, so the maximal set among L, M, N is finally chosen as $S(i,j)$. STEP 4 computes

the Y-subtrees which lie in $V(i,j)$ and are considered as candidates. These subtrees are to be used in further iterations through STEP 2. Once a maximum compatible set of patterns for P has been determined, we finish off the decomposition using the naive decomposition (STEP 5).

Procedure *ConvDec*(P)

STEP 1:

The preprocessing involves checking that P is simple and non-convex. We make a list of the notches $v_1, \ldots, v_c$, and we initialize all $B(i,i)$ and $F(i,i)$ to 0.

for $d = 1, \ldots, c-1$

  for $i = 1, \ldots, c$

    j:=i+d [mod c]

    do STEPS 2,3,4

STEP 2:

Compute the best Y-pattern connecting v_i and v_j as follows:

For each k; $v_k \in V(i+1, j-1)$, compute the set $Q = \bigcup_{1 \le i \le 4} Q_i$, where

$Q_1 = \{(F(i,k), B(k,j))\}$ /* N2-node on path */

$Q_2 = \{(B(i,k-1), F(k,j))\} \bigcup \{(B(i,j-1), F(j,j))\}$ /* no N2 or N3 nodes on path */

$Q_3 = \{(B(i,k-1), B(k,j-1))\}$ /* N3-node on path */

$Q_4 = \{(F(i+1,k), F(k+1,j))\}$ /* N3-node on path */

The elements of Q are pairs of the form (C,T). Let T be the Y-pattern which has the maximum

C value in Q.

STEP 3:

Let $S(i,j)$ be the maximum of L, M, N with respect to cardinality, where (max is taken with respect to cardinality)

$L = \max_{v_k \in V(i,j-1)} \{S(i,k) \bigcup S(k+1,j)\}$

/* corresponds to a patching together of Y-patterns */

$M = \max\{\{x_{i,a,b,j}\} \bigcup S(i+1,a-1) \bigcup S(a+1,b-1) \bigcup S(b+1,j-1)\}$

for all X_4-patterns $x_{i,a,b,j}$ connecting v_i, v_a, v_b, v_j, with $v_a, v_b \in V(i,j)$.

/* corresponds to the use of an X_4-pattern */

$N =$ {the Y-pattern T of STEP 2} $\bigcup S(i+1, i_1 - 1) \bigcup \ldots \bigcup S(i_{p-1}+1, i_p - 1) \bigcup S(i_p + 1, j-1)$

where $v_i, v_{i_1}, \ldots, v_{i_p}, v_j$ are the notches of T in clockwise order.

/* corresponds to the use of the Y-pattern T */

STEP 4:

Compute $B(i,j)$ and $F(i,j)$.

STEP 5:

Finish off the decomposition using the naive decomposition, i.e. adding one polygon for each remaining notch.

The remaining of this section is devoted to explaining the various steps of the algorithm and analyzing its complexity.

1. *Patching Y-subtrees* (STEP 2)

The function $\langle ARG \rangle$ takes two Y-subtrees and constructs a Y-pattern if these two subtrees can be patched together. ARG is any argument of the form: $(F(i,k), B(k,j))$, $(B(i,k-1), F(k,j))$, $(B(i,k-1), B(k,j-1))$ or $(F(i+1,k), F(k+1,j))$, with v_i, v_k, v_j occurring in clockwise order.

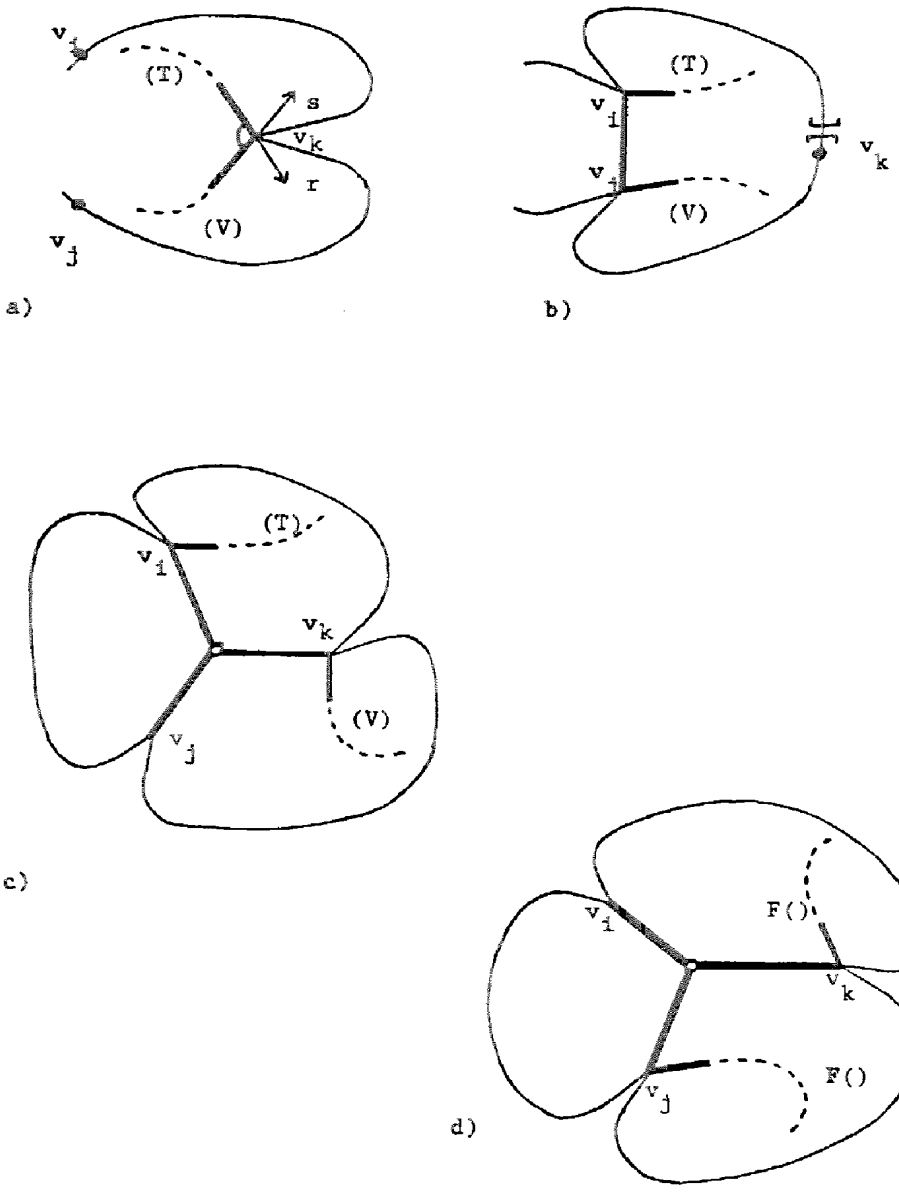


Figure 12 Computing the function <ARG>.

Case 1. $\langle F(i,k), B(k,j)\rangle$ (Fig.12-a).

Let $F(i,k) = T$ and $B(k,j) = V$, with r and s their respective arms. If $\angle(r,s) < 180$ and $T \neq 0$ and $V \neq 0$, then set $\langle F(i,k), B(k,j)\rangle = (|S(i,k)| + |S(k,j)| + 1, Y\text{-pattern: } T \bigcup V)$, else $\langle F(i,k), B(k,j)\rangle = 0$.

Case 2. $\langle B(i,k-1), F(k,j)\rangle$ (Fig.12-b).

Let $B(i,k-1) = T$ and $F(k,j) = V$. If an extended X_2-pattern is possible between v_i and v_j

then set $\langle B(i,k-1), F(k,j)\rangle = (|S(i,k-1)| + |S(k,j)| + 1, Y\text{-pattern: } \{v_iv_j\} \bigcup T \bigcup V)$, else $\langle B(i,k-1), F(k,j)\rangle = 0$.

Case 3. $\langle B(i,k-1), B(k,j-1)\rangle$ (Fig.12-c).

Let $B(i,k-1) = T$ and $B(k,j-1) = V$. If an extended X_3-pattern W is possible between v_i, v_j, v_k, then

$\langle B(i,k-1), B(k,j-1)\rangle = (|S(i,k-1)| + |S(k,j-1)| + 1, T \bigcup V \bigcup W)$,

else $\langle B(i,k-1), B(k,j-1)\rangle = 0$.

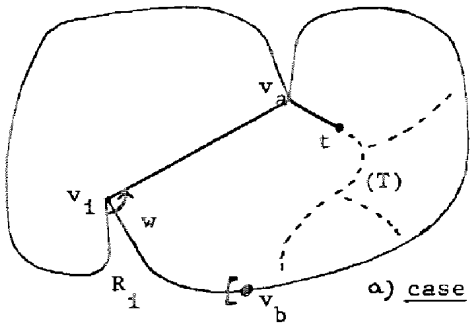
Case 4. $\langle F(i+1,k), F(k+1,j)\rangle$ (Fig.12-d).

Let $F(i+1,k) = T$ and $F(k+1,j) = V$. If an extended X_3-pattern W is possible between v_i, v_j, v_k, then

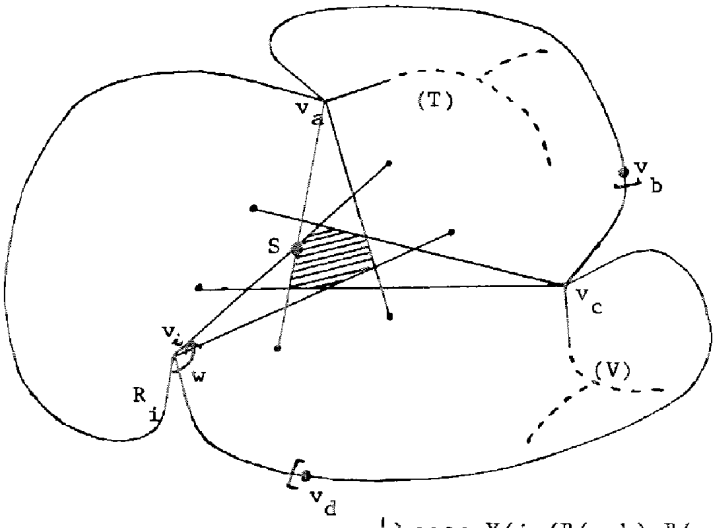
$\langle F(i+1,k), F(k+1,j)\rangle = (|S(i+1,k)| + |S(k+1,j)| + 1, T \bigcup V \bigcup W)$,

else $\langle F(i+1,k), F(k+1,j)\rangle = 0$.

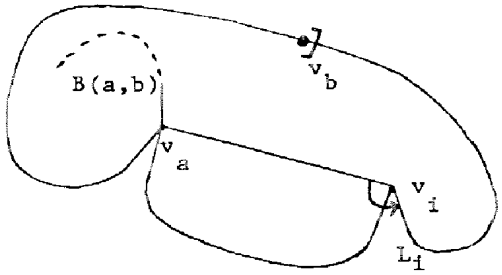
Because of Relation (2), it is clear that STEP 2 computes the Y-pattern connecting v_i and v_j (if any is to be found) such that the number of compatible patterns which can be applied in $V(i,j)$ is maximum. All we have to check is that all cases are indeed handled in STEP 2. Consider the path from v_i to v_j in any such Y-pattern. If it contains an N2-node, it will be detected in Q_1. Otherwise one N3-node may appear on this path and all these candidates will be reported in Q_3 and Q_4. The final case, handled by Q_2, assumes that the path from v_i to v_j is free of N2 and N3-nodes.



a) case Y(i,B(a,b))



b) case Y(i,(B(a,b),B(c,d)))



c) case Y'(i,B(a,b))

Figure 13 Computing Y(i,ARG) and Y'(i,ARG).

2. *Computing* $S(i,j)$ (STEP 3)

Assume by induction that $S(k,l)$ has been computed for all $v_k, v_l \in V(i,j)$ (except for $S(i,j)$). The algorithm investigates the three following cases in turn:

1. Disallow the presence of any pattern having both v_i and v_j as vertices.
2. Consider the possibility of an X_4-pattern connecting v_i and v_j.
3. Consider the possibility of a Y-pattern connecting v_i and v_j.

Lemma 7 shows that STEP 2 and STEP 3 can be accomplished in polynomial time. Correctness follows directly from previous discussion.

3. *Constructing* Y*-subtrees* (STEP 4)

We compute $B(i,j)$ and $F(i,j)$ by iteratively patching Y-subtrees together via two functions, $Y(i, ARG)$ and $Y'(i, ARG)$. ARG is an argument of the form $B(a,b)$ or $(B(a,b), B(c,d))$ (or the same with F). We describe these functions with respect to B's only, all other cases being similar.

Case 1. $Y(i, B(a,b))$ (Fig.13-a)

The vertices v_a, v_b, v_i occur in clockwise order. Let $T = B(a,b)$. Extend the notch at v_a to take into account the arm of T. Extend the notch at v_i by making its associated wedge be the entire plane. If an extended X_2-pattern is possible between v_a, v_i and if $w = \angle(R_i, v_i v_a) < 180$, set $Y(i, B(a,b)) = (Y\text{-subtree: } \{v_i v_a\} \bigcup T)$, else $Y(i, B(a,b)) = 0$.

Case 2. $Y(i, (B(a,b), B(c,d))$ (Fig.13-b)

The vertices v_a, v_b, v_c, v_d, v_i occur in clockwise order. Let $T = B(a,b)$ and $V = B(c,d)$. Extend the notch at v_a (resp. v_c) to take into account the arm of T (resp. V). Extend the notch at v_i by making its associated wedge be the entire plane. If an extended X_3-pattern is possible between v_a, v_c, v_i, compute the locus of its N3-node. Let S be the point in the locus which maximizes the angle $w = \angle(R_i, v_i S)$.

If $w < 180$, set $Y(i, (B(a,b), B(c,d)) = (Y\text{-subtree:} \{Sv_i\} \bigcup \{Sv_a\} \bigcup \{Sv_c\} \bigcup T \bigcup V)$,

else $Y(i, (B(a,b), B(c,d)) = 0$.

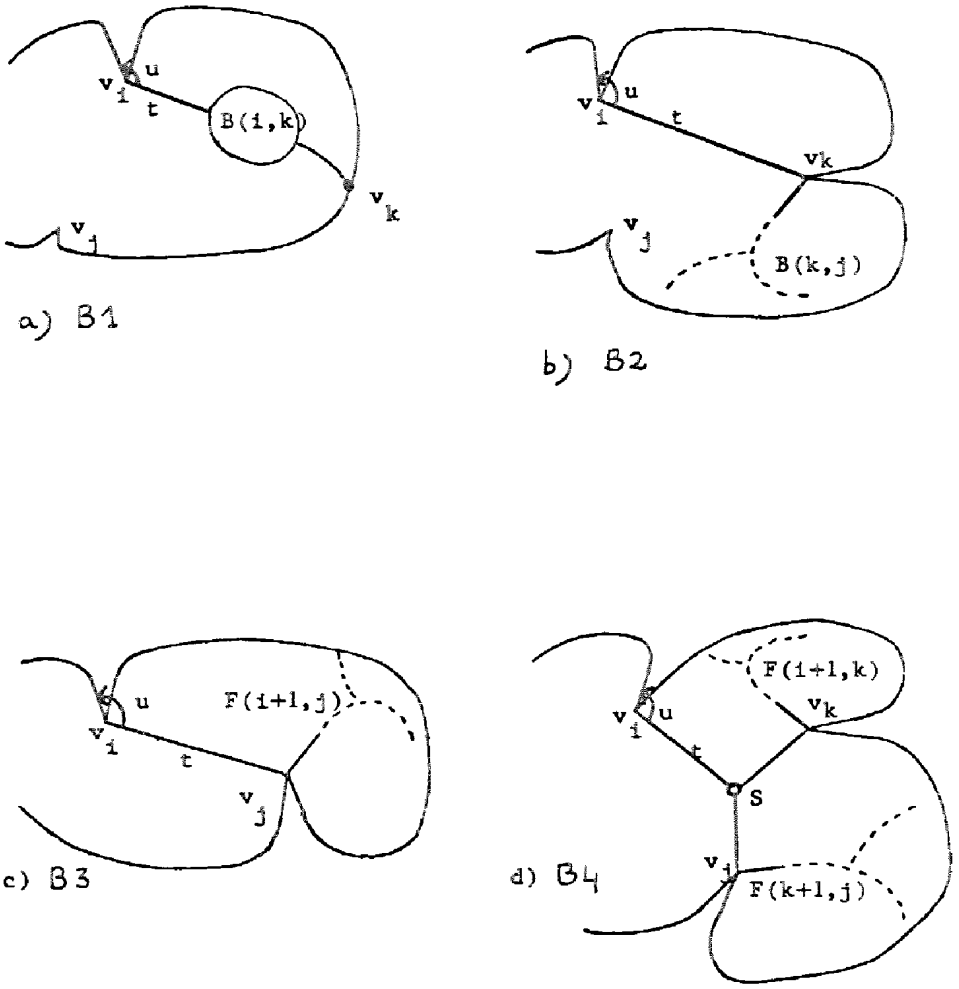


Figure 14 Computing B(i,j).

We define $Y'(i, B(a,b))$ in the same way as $Y(i, B(a,b))$, with $\angle(R_i, v_i v_a)$ replaced by $\angle(v_i v_a, L_i)$ (Fig.13-c). $Y'(i, F(a,b))$ is defined similarly, so we omit the details. We are now ready to implement STEP 4 of the decomposition algorithm. We will only describe the computation of $B(i,j)$, since the case of $F(i,j)$ is strictly similar. We begin by computing the four sets B_1, B_2, B_3, B_4. Let C be the value of $|S(i,j)|$ computed in Step 3.

$B_1 = \{$ Y-subtree of $B(i,k)\}$, for all $v_k \in V(i, j-1)$ such that $|S(i,k)| + |S(k+1,j)| = C$.

/* v_j is not a notch of the Y-subtree */

$B_2 = \{ Y'(i, B(k,j))\}$, for all $v_k \in V(i+1, j-1)$ such that $|S(i+1,k-1)| + |S(k,j)| = C$.

/* v_i's neighbor is an N2-node */

$B_3 = \{ Y'(i, F(i+1,j))\}$, if $|S(i+1,j)| = C$.

/* v_i's neighbor is an N2-node */

$B_4 = \{ Y'(i, (F(i+1,k), F(k+1,j)))\}$, for all $v_k \in V(i+1, j-1)$

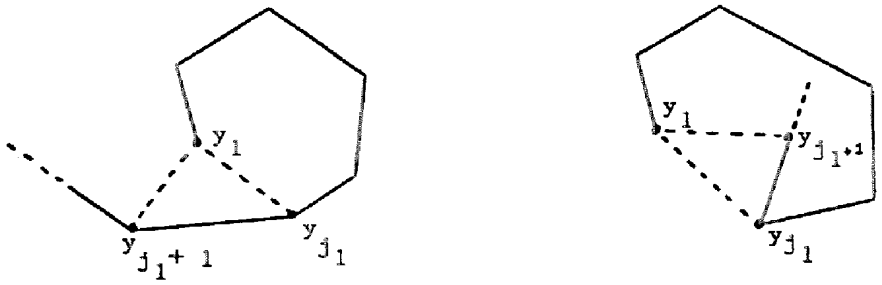
such that $|S(i+1,k)| + |S(k+1,j)| = C$.

/* v_i's neighbor is an N3-node */

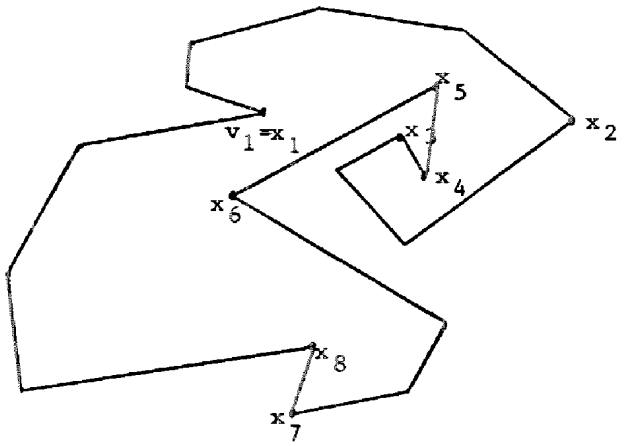
Let T be the Y-subtree of $B_1 \bigcup B_2 \bigcup B_3 \bigcup B_4$ which maximizes the angle $u = \angle(t, L_i)$, where t is understood here as the arm of T directed outward from v_i (Fig.14). We define $B(i,j)$ as a pointer to the arm of T (now understood to be directed towards v_i).

Computing $B(i,j)$ can be clearly done in polynomial time, according to Lemma 7. Note that Y-subtrees can be merged in constant time by linking their respective arms together. B_1 through B_4 evaluate all candidate Y-subtrees adjacent to v_i and lying in $V(i,j)$, and keep a single candidate, i.e. the subtree which has maximum angle u. We can show by induction that it is sufficient to consider only the Y-subtrees in the B's and F's. B_1 considers all subtrees which do not have both v_i and v_j as notches (Fig.14-a). B_2 and B_3 compute the subtrees whose vertex adjacent to v_i is an N2-node. Note that B_1 and B_2 may share common subtrees. The two possible configurations are illustrated in Fig.14-b,c. Finally B_4 detects all candidate subtrees such that the vertex adjacent to v_i is an N3-node (Fig.14-d).

4. *Completing the OCD* (Step 5)



a) The two cases for defining a pseudo-notch.



b) The pseudo-notches of a polygon.

Figure 15

The last step of procedure *ConvDec* consists of removing the remaining notches with the naive decomposition. This can be done in polynomial time, leading to the main result of this section.

Theorem: An optimal convex decomposition of a simple polygon can be computed in polynomial time.

4. Towards an Efficient Implementation

We will not make any attempt to evaluate the exponent in the polynomial expression bounding the time complexity of the previous algorithm. Clearly, it is prohibitively high. To produce a substantial savings in the running time of the algorithm, we first identify routines which need to be made more efficient. We essentially have three items to examine:

1. The naive decomposition.
2. Computing (extended) X_2 and X_3-patterns.
3. Computing X_4-patterns.

As we will see, improving the first two routines can be done without otherwise altering procedure *ConvDec*. Unfortunately, following the exact prescriptions of the procedure would lead to computing all possible X_4-patterns. Since we may have on the order of c^4 such patterns, discovering structural facts to limit the number of candidates to examine is in order. To begin, we consider the implementation of the naive decomposition. This allows us to introduce most of the tools used later on.

1. *The Naive Decomposition*

Recall that the naive decomposition involves removing each notch in turn by means of a simple line segment *naively* drawn from the notch. For simplicity we will choose a segment collinear with one of the edges of P adjacent to the notch. The degree requirements we made in the original definition of the naive decomposition can be relaxed here, since their only motivation was the simplification of subsequent proofs. Each line segment extends from a notch to the first intersection with the current decomposition. This can be easily accomplished in $O(n+c)$ time, but this is still too slow for our purposes.

The improvement which we propose involves $O(n)$ time preprocessing at the outset of the computation. Before describing this preprocessing, a few more definitions are in order. A *convex polygonal line* is a sequence of vertices $\{a_1, \ldots, a_p\}$ such that $\angle(a_i a_{i-1}, a_i a_{i+1}) \leq 180$, for each i such that $1 < i < p$. The sequence is called a *convex chain* if $\{a_1, \ldots, a_p, a_1\}$ forms a convex polygon. Note that in this case $a_1, \ldots, a_p$ corresponds to a clockwise traversal of the polygon. It is known from [3,6] that if the vertices of a convex n-gon are stored in a linear array it is possible to compute its intersection with any line in $O(\log n)$ time. Unfortunately, between each pair of notches v_i, v_{i+1}, the boundary of P certainly is a convex polygonal line but not necessarily a convex chain. This motivates the following preprocessing.

Partition the boundary of P between two consecutive notches into contiguous convex chains. Let $L_i = \{y_1, \ldots, y_p\}$ be the convex polygonal line given in clockwise order, with $y_1 = v_i$ and $y_p = v_{i+1}$. If neither the angle $\angle(y_1 y_k, y_1 y_2)$ nor the angle $\angle(y_k y_{k-1}, y_k y_1)$ is reflex for any $2 \leq k \leq p$, L_i is a convex chain and remains unchanged. Otherwise, let j_1 be the smallest k such that $\{y_1, \ldots, y_k, y_{k+1}, y_1\}$ is a non-convex polygon, i.e. such that either $\angle(y_1 y_{k+1}, y_1 y_2)$ or $\angle(y_{k+1} y_k, y_{k+1} y_1)$ is reflex (Fig.15-a). We define C_1 as the convex chain $\{y_1, \ldots, y_{j_1}\}$. Next we apply the same procedure recursively on the remaining part of L_i. This leads to defining C_2 as $\{y_{j_1}, \ldots, y_{j_2}\}$, with j_2 being the smallest $k > j_1$ such that $(y_{j_1} y_{k+1}, y_{j_1} y_{j_1+1})$ or $(y_{k+1} y_k, y_{k+1} y_{j_1})$ is reflex. We iterate on this process until we reach y_p, thus partitioning L_i into t consecutive convex chains $C_1, \ldots, C_t$.

We apply the same treatment to each pair of notches (v_i, v_{i+1}) and renumber the chains accordingly. This leads to a partitioning of the whole boundary of P into m consecutive convex chains, $C_1, \ldots, C_m$, in clockwise order. Letting x_i, x_{i+1} be the endpoints of C_i in clockwise order (with $x_1 = x_{m+1} = v_1$), we call the x_i the *pseudo-notches* of P. Note that all notches are pseudo-notches but the converse is in general not true (Fig.15-b). This preprocessing requires $O(n)$ operations. We next show that the number of convex chains is of the same order of magnitude as the number of notches.

Lemma 8. $m \leq 2(1 + c)$.

Proof: Consider the vertices p_i of P which are *not* notches of P and let U be the sum of all the angles $\angle(p_ip_{i+1}, p_{i-1}p_i)$. Similarly for all vertices p_j which are notches of P, let V be the sum value of all the angles $\angle(p_{j-1}p_j, p_jp_{j+1})$. It is a classical result of geometry that

$$U - V = 360. \tag{3}$$

For each convex chain $C_i = \{a_1, \ldots, a_p\}$ such that a_p is not a notch of P, let a_{p+1} be the vertex of P adjacent to a_p in clockwise order. We define U_i as the sum of all angles $\angle(a_ja_{j+1}, a_{j-1}a_j)$, for all $2 \leq j \leq p$. Let $U_{i_1}, \ldots, U_{i_t}$ be the values thus obtained. By construction the polygon $\{a_1, \ldots, a_p, a_{p+1}, a_1\}$ has a reflex angle either at a_{p+1} or at a_1. It follows that if c (resp. d) is the angle $\angle(a_{p+1}a_1, a_pa_{p+1})$ (resp. $\angle(a_1a_2, a_{p+1}a_1)$) measured (exceptionally) between -180 and $+180$ degrees, negative if there is a reflex angle at a_{p+1} (resp. a_1), positive otherwise, we have

$$U_i = 360 - (c + d) \geq 180. \tag{4}$$

Since none of the U_i's accounts for the reflex angles of P, we have $\sum_{1 \leq j \leq t} U_{i_j} \leq U$. Also if, between a pair of consecutive notches, P consists of a single convex chain, no U_i is defined on this portion of P, whereas if it consists of p chains, $p - 1$ U_i's are defined. This implies that $t = m - c$. Combining this fact with (4) we derive

$$180(m - c) \leq \sum_{1 \leq j \leq t} U_{i_j} \leq U,$$

and from (3)

$$U = 360 + V \leq 180(2 + c),$$

which completes the proof. ∎

This leads to the following result.

Lemma 9. In procedure *ConvDec*, STEP 5 can be accomplished in time $O(n + c^2 \log n)$.

Proof: We assume that P has been preprocessed as described above, with each convex chain stored in a linear array. For each remaining notch v, in turn, let t be the ray emanating from v in a direction removing the reflex angle at v (for example the direction of one of the edges

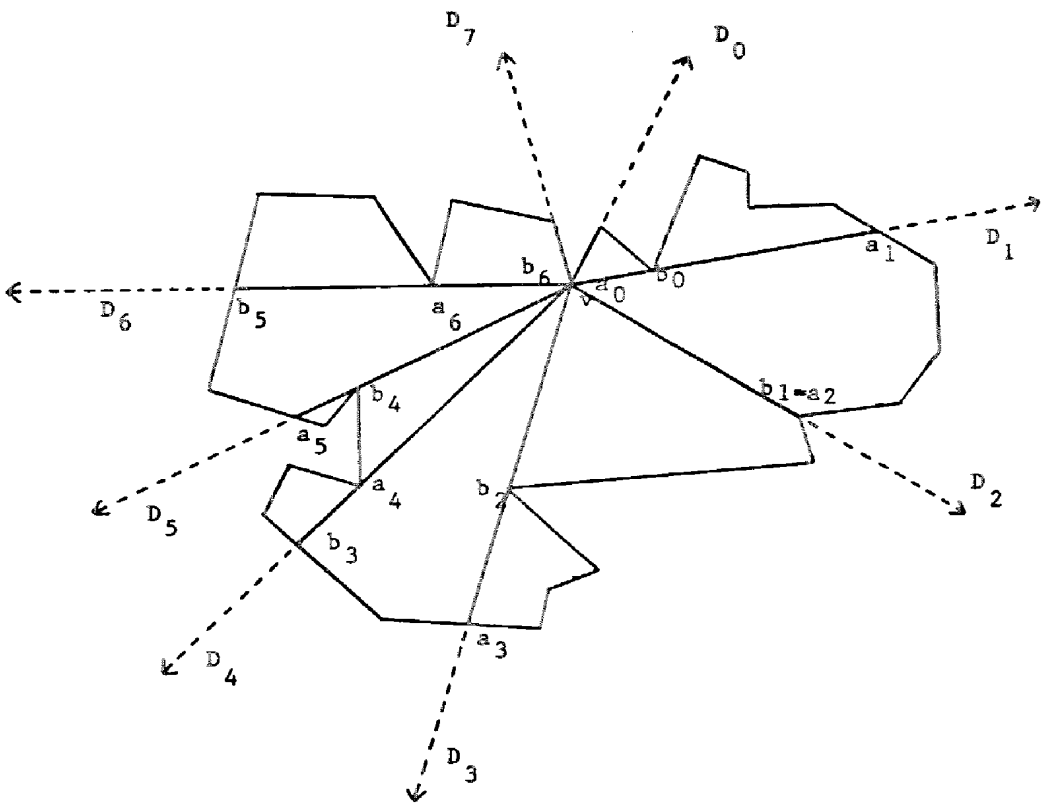


Figure 16 The superrange of a notch v.

adjacent to v). Among the intersection-points of t with the edges of the current decomposition, we must determine the closest to v. To do so, we compute all the intersections between t and the *internal edges* of the decomposition, i.e. the edges not on the boundary of P. It is easy to show by induction that there are $O(c)$ such edges, hence $O(c)$ intersection-points to compute. Next we compute the intersection of t with each convex chain of P in turn, using the fast algorithm of [6]. The running time of this method will be $O(n + c^2 \log n)$, including preprocessing. ∎

2. *Computing Extended Patterns*

We refine the preprocessing described above. Since the computation of X-patterns is intimately based on the notion of ranges, we precompute the visibility-polygon with respect to each notch at an overall cost of $O(c^2 \log n)$ time and space. This may seem a paradox since storing all these polygons may require as much as $\theta(cn)$ storage. The crux is that only significant vertices of the visibility-polygons will be stored. This economical description of the visibility-polygon of a notch v is called the ***superrange*** of v. In the next paragraphs we describe how to compute superranges and then show how to use these new structures efficiently.

Let v be a notch of P and $t_1, \ldots, t_p$ be the list in clockwise order of all the pseudo-notches visible from v. Note that scanning $t_1, \ldots, t_p$ corresponds to a clockwise traversal of the boundary of P as well as a clockwise sweep around v. Let D_i be the ray emanating from v with the direction from v to t_i and let D_0 (resp. D_{p+1}) be the ray passing through the edge of P starting from v in clockwise (resp. counterclockwise) order. The set of rays $D_0, \ldots, D_{p+1}$ partitions the region of P visible from v into $p+1$ simple polygons, all adjacent to v. Typically a polygon is comprised between D_i, D_{i+1} and a convex polygonal line on the boundary of P. Let a_i and b_i be the endpoints of this convex line (with b_i following a_i in clockwise order Fig.16). For each notch v, we define the ***superrange*** of v, denoted $\mathrm{SR}(v)$, as the ordered list

$$\mathrm{SR}(v) = \{(a_0, b_0), \ldots, (a_p, b_p)\}.$$

The next result states that superranges can be computed very efficiently.

Lemma 10. An n-gon can be preprocessed in $O(n)$ time so that the superrange of any of its notches can be computed in $O(c \log n)$ time.

Proof: The proof of this result is too lengthy to be included here. We refer the reader to [3] for the details of the proof. In [7] El-Gindy and Avis present a linear algorithm for computing the visibility-polygon from any point in P. Although their algorithm cannot be used here since it is too slow for our purposes, we can still use it as a basis to give intuition for our algorithm. El-Gindy and Avis's algorithm can be seen as an extension of a Graham scan. It essentially involves traversing the boundary of P in one direction, occasionally backing up but never more than once per vertex. Our algorithm is conceptually similar to [7]; the only difference comes from the traversing scheme used. Instead of going from one vertex to the next, indeed, we go from one convex chain to the next. In this regard, the data type "edge" in [7] becomes, in our algorithm, the data type "convex chain". An important feature of the former data type which we lose in our algorithm is the property that a ray scanning an edge moves either totally clockwise or totally counterclockwise. A convex chain, instead, can change directions at most twice. However, whenever entering a new chain, we can use the fast algorithm of [6] to compute the changes of direction in $O(\log n)$ time. This means that we can rewrite the entire algorithm of [7], now taking convex chains instead of edges as our basic objects. The price to pay will be a factor $\log n$ in every step of [7]'s algorithm. Since there are only $O(c)$ convex chains, however, the $O(n)$ algorithm of [7] now becomes an $O(c \log n)$ algorithm. ∎

The notion of superrange can be of great use for many geometric problems and is thus interesting in its own right. To appreciate its usefulness to our specific decomposition problem, we need introduce a function of two arguments, $R(v, D)$, where v is a notch of P and D is a ray emanating from v. Let D_i, D_{i+1} be the two rays (introduced earlier in the definition of superranges) between which D lies. If $\angle(vb_i, va_i)$ is reflex, $R(v, D)$ is set to 0, otherwise it is set to the segment vy, where y is the intersection of D with a_ib_i. Clearly, $R(v, D)$ is still well-defined if D is a directed segment emanating from v instead of a ray. The next two results give motivation for these definitions.

Lemma 11. Once the superrange of each notch has been computed, $R(v, D)$ can be computed in $O(c)$ time for any notch v.

Proof: Trivial. ∎

Lemma 12. If v is a notch of P and vx is the edge of an X-pattern, x lies on the segment $R(v, vx)$.

Proof: The result is obvious if x lies on the boundary of P, since it is then a notch of P and $R(v, vx)$ is exactly vx. Suppose that x is an N3-node and vx contains $R(v, vx)$ (Fig.17). Let (a_i, b_i) be the pair of $\mathrm{SR}(v)$ such that vx intersects a_ib_i. The segment a_ib_i partitions P into two polygons, P_1 and P_2, with say P_1 containing x. Since the portion of the boundary of P between a_i and b_i is a convex chain, P_1 is a convex polygon, therefore the X-pattern cannot have notches in P_1, which is in contradiction with the fact that all the angles formed by an X-pattern are non-reflex. ∎

We next show how to use these results to compute extended X_2 and X_3-patterns efficiently. For simplicity we will first consider the cases where the patterns are standard, i.e. not extended. It will then be easy to generalize the results obtained to extended patterns.

– *Detecting X_2-patterns*

Lemma 13. With $O(n + c^3 + c^2 \log n)$ preprocessing it is possible to check for the possibility of an X_2-pattern between any two notches in constant time.

Proof: The preprocessing involves computing the superrange of each notch as well as the segments $R(v_i, v_iv_j)$, for all pairs of notches v_i, v_j. Lemmas 10 and 11 show that this can be done in $O(c^3 + c^2 \log n)$ time. From Lemma 12, it then follows that an X_2-pattern between v_i and v_j is possible if and only if $R(v_i, v_iv_j) = v_iv_j$ and the segment v_iv_j removes the reflex angle at both v_i and v_j. ∎

– *Detecting X_3-patterns*

Computing X_3-patterns is somewhat more complicated. We need some additional preprocessing which we next describe. Recall that R_i and L_i are the *directed* segments from v_i to the next vertices of P respectively following and preceding v_i in clockwise order. This notion of direction allows us to define $R(v_i, R_i)$ and $R(v_i, L_i)$ without ambiguity. Similarly to give full meaning to angles of the form $\angle(v_ix, R(v_i, D))$, the segment $R(v_i, D)$ will be understood as assuming the same direction as D.

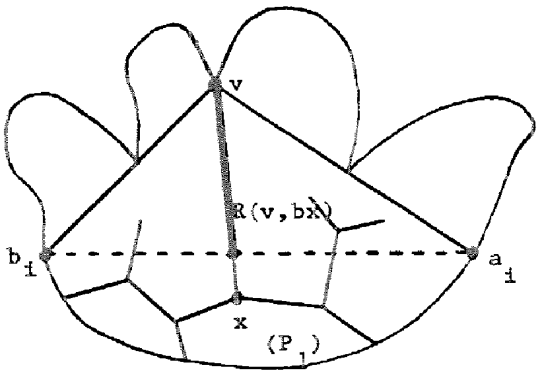


Figure 17 R(v,D) expresses the longest edge vx with direction D of an X-pattern.

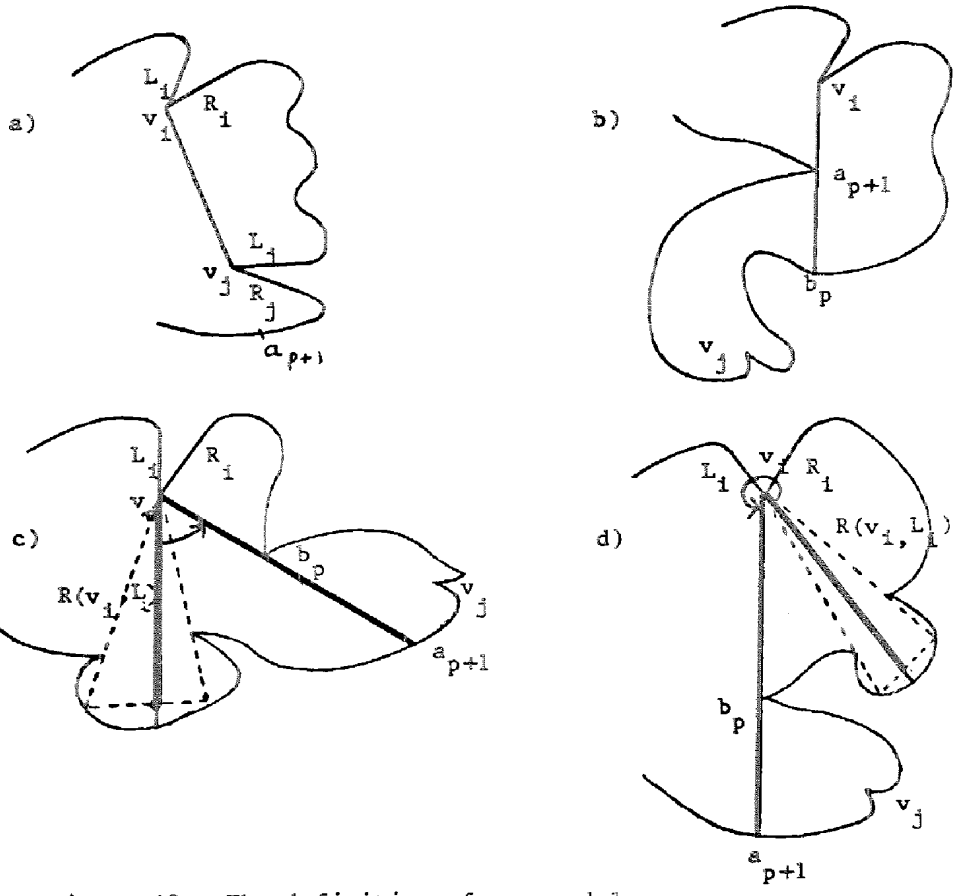


Figure 18 The definition of r_{ij} and l_{ji}.

For each pair of notches v_i, v_j, we define the two points r_{ij} and l_{ij} as follows: $v_i r_{ij}$ is the rightmost segment in the range of v_i, visible from v_j. More precisely, if both R_i and L_j lie on the same side of the line passing through $v_i v_j$ and $\angle(v_i v_j, R_i) < 180$ (Fig.18-a), we determine the pairs (a_p, b_p) and (a_{p+1}, b_{p+1}) of $\mathrm{SR}(v_i)$ such that v_j occurs between b_p and a_{p+1} in a clockwise traversal of the boundary of P. We assume that a_{p+1} does not lie on the segment $v_i b_p$ (unlike in Fig.18-b). We may have $v_j = b_p = a_{p+1}$, however. Let t denote the segment $R(v_i, L_i)$ if $\angle(R(v_i, L_i), v_i a_{p+1}) < 180$ (Fig.18-c), and the segment $v_i a_{p+1}$ otherwise (Fig.18-d). If we actually have $\angle(v_i v_j, t) < 180$, we define t as $R(v_i, v_i v_j)$. Finally if $\angle(t, R_i) < 180$, we define r_{ij} as the endpoint of t ($\neq v_i$). If any of the above conditions fails, r_{ij} is 0.

We repeat the same process on v_j with respect to v_i. If R_i and L_j lie on the same side of the line passing through $v_i v_j$, we first determine the pairs (a_p, b_p) and (a_{p+1}, b_{p+1}) from $\mathrm{SR}(v_j)$ such that v_i occurs between b_p and a_{p+1} in clockwise order. We will suppose that b_p does not lie strictly between v_j and a_{p+1}. Let t be $R(v_j, R_j)$ if $\angle(v_j b_p, R(v_j, R_j)) < 180$ or $v_j b_p$ otherwise. Similarly, if $\angle(t, v_j v_i) < 180$, t is reset to $R(v_j, v_j v_i)$ so that we can define l_{ji} as the endpoint of t other than v_j if $\angle(L_j, t) < 180$. In all other cases, l_{ji} is set to 0.

With the superrange of each notch at our disposal, we can compute each r_{ij} and l_{ji} in $O(c)$ time, which yields an $O(n + c^3 + c^2 \log n)$ overall preprocessing time. We are now ready to describe the computation of X_3-patterns.

Lemma 14. With $O(n + c^3 + c^2 \log n)$ preprocessing it is possible to check for the possibility of an X_3-pattern between any three notches in constant time.

Proof: Let v_i, v_j, v_k be three notches of P. We wish to give a set of necessary and sufficient conditions for v_i, v_j, v_k to form an X_3-pattern. Let $C_1 = v_i r_{ij} \cap v_j l_{ji}$, $C_2 = v_i r_{ij} \cap v_k r_{ki}$ and $C_3 = v_j l_{ji} \cap v_k l_{kj}$. One of the following is true: 1) C_2 is further from v_i than C_1 is $(\angle(v_k l_{kj}, v_k r_{ki}) + \angle(v_k r_{ki}, v_k C_1) = \angle(v_k l_{kj}, v_k C_1))$; 2) C_3 is further from v_j than C_1 is $(\angle(v_k C_1, v_k l_{kj}) + \angle(v_k l_{kj}, v_k r_{ki}) = \angle(v_k C_1, v_k r_{ki}))$; 3) C_1 is further from v_i than C_2 is and further from v_j than C_3 is $(\angle(v_k l_{kj}, v_k C_1) + \angle(v_k C_1, v_k r_{ki}) = \angle(v_k l_{kj}, v_k r_{ki}))$. In case 1, set $C = C_2$; in case 2, set $C = C_3$, and in case 3 set $C = C_1$. Similarly we introduce the points A and B, defined like C with respect to (v_j, v_k) and (v_k, v_i) respectively. From now on, we will

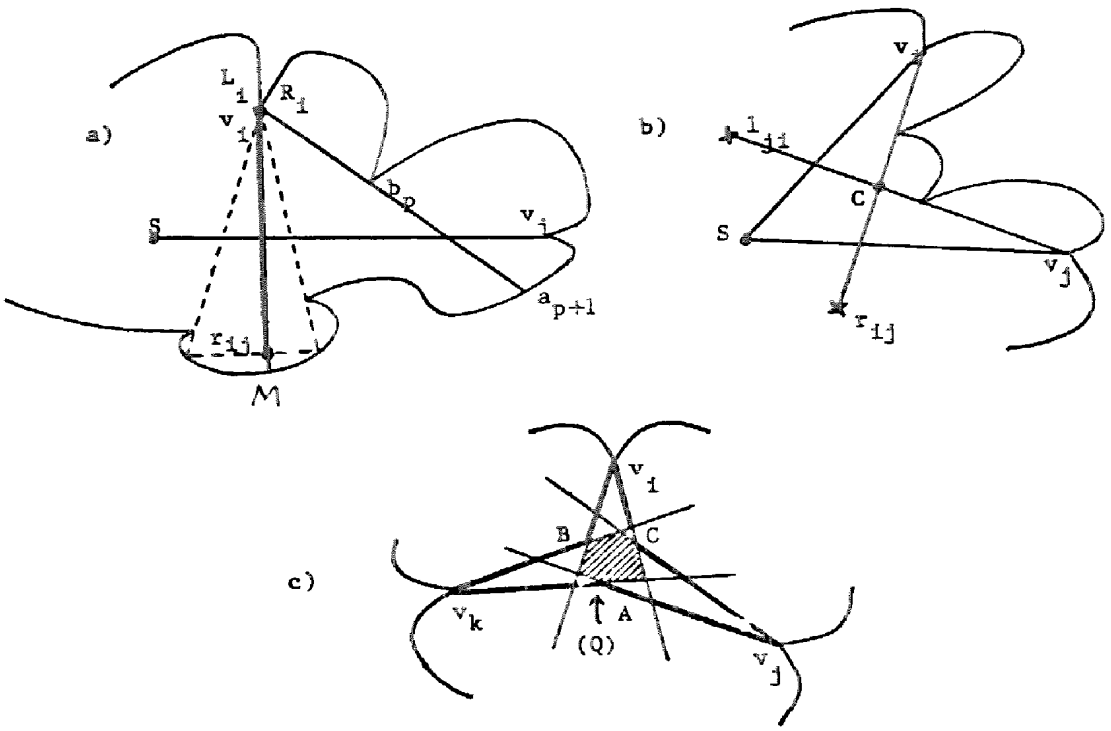


Figure 19 Detecting X3-patterns.

assume that A, B and C fall in case 3. The other cases are treated in a similar way and may thus be omitted. We can show that v_i, v_j, v_k are the notches of an X_3-pattern if and only if:

1. v_i, v_j, v_k occur in clockwise order on the triangle $v_i v_j v_k$.
2. $r_{ij}, r_{jk}, r_{ki}, l_{ik}, l_{kj}, l_{ji}$ are all distinct from 0.
3. The points A, B, C are well-defined.
4. The polygon $Q = v_i C v_j A v_k B v_i$ is simple and has a non-empty kernel (recall that the kernel of a polygon is the region visible from every point in the polygon).

All these conditions can be easily checked in constant time with the preprocessing described above.

We say that a point is *range-visible* from a notch v if it lies in its range, i.e. if the segment lies totally within P and removes the reflex angle at v. We define the *wedge* $W(ax, ay)$ as the region swept by a ray pivoting in clockwise order around a from ax to ay. Let S be the N3-node of an X_3-pattern between v_i, v_j, v_k. To prove that the second condition is also necessary, we show that $r_{ij} \neq 0$, all of the other cases being similar. Since the three edges of the pattern must lie in the triangle v_i, v_j, v_k, the first requirement (illustrated in Fig.18-a) is obvious. Considering the pairs (a_p, b_p) and (a_{p+1}, b_{p+1}) it is equally clear that the configuration of Fig.18-b cannot lead to an X_3-pattern since we must have $\angle(Sv_j, Sv_i) < 180$, where S is the N3-node. Indeed, Sv_j must intersect $b_p a_{p+1}$ with possibly $v_j = b_p$, since S must be visible from both v_i and v_j. This remark shows that not only are the configurations of Fig.18-c,d the only ones possible, but also that S cannot lie in the wedge $W(v_i a_{p+1}, R_i)$. The other conditions to satisfy in order to define r_{ij} express the fact that S lies in the triangle $v_i v_j v_k$ as well as the range of v_i. Also, since we must have $\angle(v_i S, v_i v_j) < 180$, it is legitimate to set t to $R(v_i, v_i v_j)$, if $\angle(v_i v_j, t) < 180$. Finally if $\angle(t, R_i) > 180$ no point visible from v_j can be range-visible from v_i, so we can set r_{ij} to 0. Thus, when an N3-node exists, all these conditions will be satisfied and S cannot lie in the wedge $W(v_i r_{ij}, R_i)$.

As mentioned earlier, the points $a_0, b_0, a_1, b_1, \ldots, a_p, b_p$ occur in clockwise order around the boundary of P, therefore we must have $\angle(v_i l_{ik}, v_i r_{ij}) < 180$ if $l_{ik} \neq 0$ and $r_{ij} \neq 0$, since v_i, v_j, v_k occur in clockwise order. It follows that if A, B, C exist, the polygon Q must be simple.

To prove that these points are well-defined, we first show that Sv_j intersects v_ir_{ij}. As we have seen that Sv_j intersects v_ia_{p+1}, thus implying that v_ir_{ij} is not defined as $R(v_i, v_iv_j)$, we only have to show that Sv_j intersects v_ir_{ij} whenever this segment is defined as $R(v_i, L_i)$. Since S cannot lie in $W(R(v_i, L_i), R_i)$, Sv_j must intersect v_iM (Fig.19-a). Also Sv_j cannot intersect Mr_{ij}, since S would then belong to a convex polygon where no N3-node point can lie (Lemma 12). This proves our claim, and shows that r_{ij} (as well as l_{ji} by a similar reasoning) lies outside the triangle Sv_iv_j (Fig.19-b). Finally, as we know that S cannot lie in the wedges $W(v_ir_{ij}, R_i)$ and $W(L_j, v_jl_{ji})$, we derive $\angle(v_iS, v_ir_{ij}) < 180$ and $\angle(v_jl_{ji}, v_jS) < 180$ which, combined with the previous result, establishes that v_ir_{ij} and v_jl_{ji} intersect. This proves the existence of the point C as well as points A and B, by symmetry. Since S must lie in $W(v_il_{ik}, v_ir_{ij})$, the same reasoning applied to v_j and v_k shows that S lies in the kernel of Q.

The four conditions having proven necessary, we next show that they are sufficient. Assume that they are all satisfied (Fig.19-c). Since v_ir_{ij} is range-visible from v_i, and so is v_jl_{ji} from v_j, Condition 3 shows that C is range-visible from both v_i and v_j. It follows that the boundary of P cannot intersect strictly with v_iC or v_jC, and by symmetry, cannot intersect with the edges of Q. Therefore, any point of its kernel is range-visible from v_i, v_j, v_k and is the N3-node of a possible X_3-pattern. Note that all three angles around the Steiner point are ensured to be < 180 since the kernel of Q lies within the triangle $v_iv_jv_k$. ∎

Our techniques for computing X_2 and X_3 patterns can be used to handle extended patterns as well. Patching together Y-subtrees in STEP 2 can be done along the same lines and no further explanation is necessary. The only remaining question to address concerns the computation of $Y(i, ARG)$ and $Y'(i, ARG)$ in STEP 4. The most general case corresponds to the computation of $Y(i, (B(a, b), B(c, d)))$ (Fig.13) (Y' is handled similarly). We extend the notches v_a, v_c, v_i accordingly, and (referring to Condition 4 in Lemma 14) compute the kernel of Q. In doing so, we drop all requirements involving R_i, since we do not have to remove reflex angles at v_i at this point. Once again, details are tedious but straightforward – see [3]. We conclude

Lemma 15. After $O(n + c^3 + c^2 \log n)$ preprocessing it is possible, in constant time, to

1. check for the possibility of an extended X_2-pattern between any two notches,

2. check for the possibility of an extended X_3-pattern between any three notches,
3. evaluate the functions Y and Y'.

3. *Computing X_4-patterns*

We now have to face the most difficult task in our quest for an efficient implementation. Improving the previous routines simply relied on increasing the amount of preprocessing without otherwise altering the basic nature of the algorithm. The excessive number of possible X_4-patterns require conceptual changes in our algorithm. These changes are based on the observation that of all the $\binom{n}{4}$ potential X_4-patterns, only $O(n^3)$ need be retained for consideration. Implementing the selection requires structural facts about the nature of X_4-patterns.

Definition 4. An X_4-pattern is said to be *loose* if it can be reduced so that each edge adjacent to a notch v_i is made to be collinear with either R_i or L_i (16 configurations should thus be achievable – Fig.20).

The term "reduced" is to be understood here in the sense of Lemma 6. The introduction of loose patterns finds justification in the following.

Lemma 16. Every X_4-pattern which is not reducible to an X_3-pattern or a Y_5-pattern can be reduced to a loose X_4-pattern.

Proof: Let $hull(T)$ designate the convex hull of all the points of an X-pattern T. It is clear that every X_4-pattern, T, can be reduced to an X_4-pattern V such that no further reduction of V can lead to another X_4-pattern lying strictly inside $hull(V)$, i.e. an X_4-pattern where at least one notch lies strictly in the interior of $hull(V)$. We show that if T cannot be reduced to a Y-pattern, V must be loose. Assume that one of the 16 configurations cannot be achieved for V; by applying the reductions shown in Fig.21 we can reach an X_3-pattern, a Y-pattern or an X_4-pattern lying strictly in $hull(V)$. Actually, another possibility is to reduce to the alternate case of Fig.21, which can arise only a finite number of times. ∎

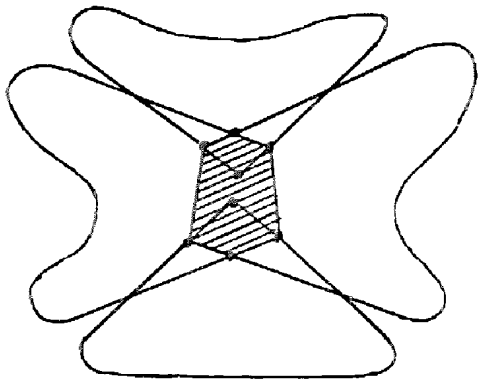


Figure 20 A loose X4-pattern with its 16 extreme configurations.

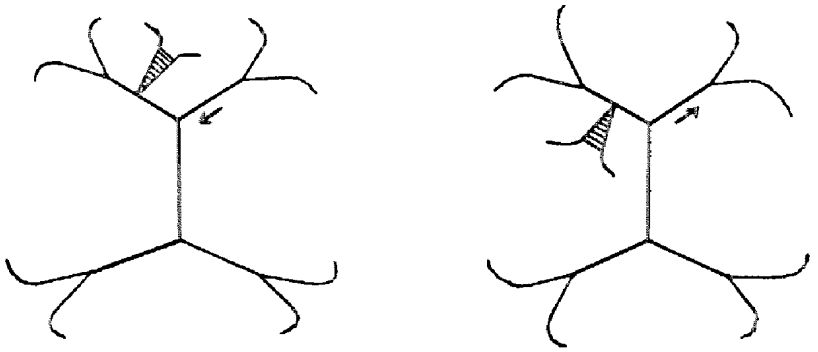


Figure 21 X4-patterns can be assumed to be loose.

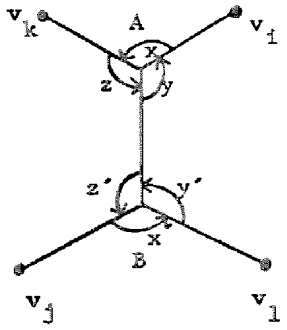


Figure 22 Characteristics of an X4-pattern

Because of this result we can assume that the X_4-patterns considered in STEP 3 are loose and collinear with the right edge of each notch involved. More precisely, if S is an N3-node adjacent to v_i, the edge R_i is collinear with Sv_i. From now on, all the X_4-patterns considered will be assumed to have this configuration. Before addressing the main problem, we give a convenient characterization of X_4-patterns which we will be using throughout.

Lemma 17. Let v_k, v_i, v_l, v_j be four notches in clockwise order around P, the tree in Fig.22 forms an X_4-pattern if and only if:

1. A and B are the only two intersections between edges.
2. Angles $\angle x, \angle y, \angle z, \angle x', \angle y', \angle z'$ are < 180 degrees.
3. No edge of the tree intersects with the boundary of P (except at the notches).

Proof: This characterization is fairly straightforward. It is important to notice, however, that if we consider the convex hull of the X_4-pattern, a clockwise order of its four vertices corresponds to a clockwise order of the notches on the boundary of P. This topological fact will be useful throughout. ∎

The fact that an X_4-pattern with the configuration of Fig.22 is possible between v_i, v_j, v_k, v_l, A, B will be expressed by the notation X4(v_k, v_i, v_l, v_j, A, B). We will often use this notation with nodes replaced by $*$ in order to represent the set of all possible X_4-patterns having the $*$-ed elements filled in. Next we introduce some operations to be added to the preprocessing at STEP 1. Let r_i be the segment $R(v_i, R_i)$ and (a_p, b_p) be the pair of SR(v_i) such that $a_p b_p$ intersects r_i (recall that this pair has to be determined in order to compute $R(v_i, R_i)$ – see Lemma 11). For all v_k between b_p and v_i in clockwise order (including b_p if it is a notch), compute A_{ik}, the intersection of r_i and r_k if it exists. Note that the intersection is undefined if one of the segments r_u is 0.

Fact 1. $\angle(A_{ik}v_i, A_{ik}v_k) < 180$, and $A_{ik}v_i$ and $A_{ik}v_k$ intersect the boundary of P at v_i and v_k, respectively.

Fact 2. For each v_i, the set of A_{ik}'s contains all possible N3-nodes adjacent to v_i in the loose X4($*, v_i, *, *, *, *$).

Next for each v_i the points A_{ik} are sorted along r_i and maintained in a sorted list. In the following, r_i will be viewed either as a geometric segment or as a list of sorted points. The data structure chosen for r_i should allow constant time access to A_{ik} as well as two-way scans through the list r_i (use a linear array for example). Note that the A_{ik} might not be defined for all pairs (i,k). All the segments r_i's can be computed in $O(n + c^2 \log n)$ time and setting up their respective lists can be done in $O(c^2 \log c)$ operations.

We wish to apply the idea of patching subtrees together to the construction of X_4-patterns. In the configuration of Fig.22, the edge $v_k A$ is to be patched with the rest of the pattern. To generate X_4-subtrees we extend the notion of F and B functions. We introduce the set $E(i,j)$ to store all the information needed to decide, in constant time, if for a given v_k there exists a v_l such that X4$(v_k, v_i, v_l, v_j, *, *)$. The set $E(i,j)$ will be computed immediately after $S(i,j)$. To do so, we will consider each notch v_l between v_i and v_j and determine all the v_k that can be patched to form an X_4-pattern. Since for each v_l we potentially have on the order of c notches of the form v_k, we must avoid going through each of them if our goal is to compute $E(i,j)$ in $O(c)$ time. Fortunately for each v_l we can express the corresponding set of v_k's in constant space after constant time computing time. It remains now to formalize the intuition given above.

$E(i,j)$ is defined as the set of pairs (A_{ik}, A_{jl}) obtained for all *distinct* values of k with the following properties:

1) X4$(v_k, v_i, v_l, v_j, A_{ik}, A_{jl})$; 2) if an OCD contains a loose X_4-pattern, X4$(v_k, v_i, *, v_j, *, *)$, then there exists an OCD containing X4$(v_k, v_i, v_l, v_j, A_{ik}, A_{jl})$, where $(A_{ik}, A_{jl}) \in E(i,j)$. This allows the set $E(i,j)$ to be used for our purposes without overlooking candidate X_4-patterns. We next show that such sets can be found satisfying this property and that each of them can be computed in $O(c)$ time with the previous preprocessing.

1. Computing $E(i,j)$

Recall that $E(i,j)$ is to be computed immediately after $S(i,j)$.

I) *Selecting candidates on r_i and r_j*

To begin with, we determine the points A_{ik} such that $v_k \in V(j+1, i-1)$ (note that if the pair (a_p, b_p) of $\mathrm{SR}(v_i)$ used to compute r_i is such that $b_p \in V(j, i)$, all the vertices in r_i will already satisfy this condition). Next we keep only the A_{ik} which lie on the other side of the infinite line passing through R_j than the edge L_j (repeat same operations with respect to r_j and A_{jl}). This ensures that:

Fact 3. v_j, v_k, v_i occur in clockwise order and the angle $\angle z' < 180$ (Fig.22).

Fact 4. v_i, v_l, v_j occur in clockwise order and $\angle y < 180$.

We now retain in r_j only the points A_{jl} for which

$$|S(i,j)| = |S(i+1, l-1)| + |S(l+1, j-1)|.$$

By doing this we keep only the candidates for N3-nodes of an OCD. Similar to the Y-subtrees occurring in $B(i,j)$, candidates must contribute a savings of $|S(i,j)|$ with respect to the removal of reflex angles in $V(i,j)$. Finally we update the lists r_i and r_j with the points just chosen, maintaining the sorted order. We rename the points of r_i (resp. r_j) from v_i (resp. v_j), $A_1, \ldots, A_p$ (resp. $B_1, \ldots, B_q$). This entire step can be done in $O(c)$ time with the preprocessing indicated earlier. We now have a list of all possible N3-nodes for candidate X_4-patterns. It is clear that $E(i,j)$ can be found satisfying the specifications given above. Each A_k will be paired with the point B_l such that A_k and B_l are the N3-nodes of the same optimal X_4-pattern forming the maximum angle $\angle(A_k B_l, A_k v_i)$ (Fig.23). We must now give a precise procedure for accomplishing this task.

II) *Computing a region of safety*

We compute the ***region of safety*** for added edges in order to ensure Condition 3 of Lemma 17. Note that $A_1 A_p B_1 B_q$ forms a convex quadrilateral (Facts 3,4). The next step is to compute two convex chains $C = \{c_1, \ldots, c_s\}$ and $D = \{d_1, \ldots, d_t\}$ running from r_i to r_j and r_j to r_i, respectively. These chains will have the property that a ***middle edge***, (i.e. the edge of an X_4-pattern between the two N3-nodes) from r_i to r_j lies totally in P if and only if it lies totally between the two chains (Fig.27-a). Informally C (resp. D) is the convex hull of the pieces of

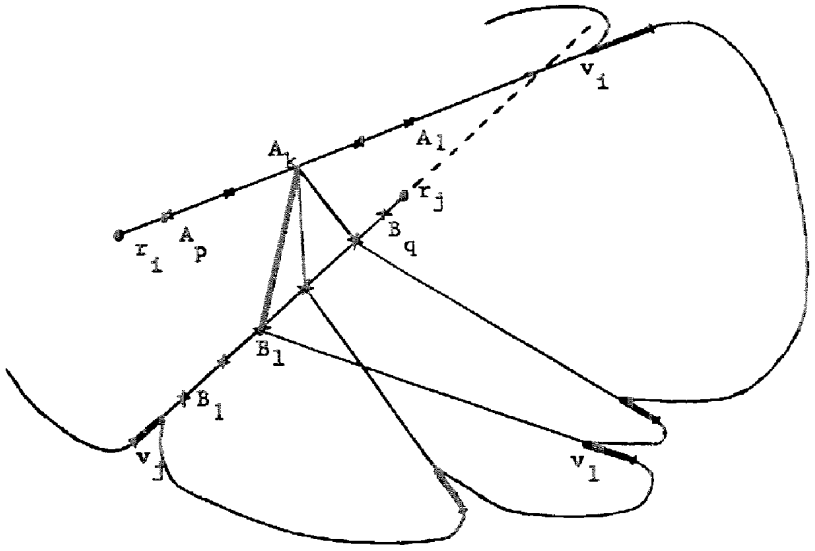


Figure 23 Computing E(i,j).

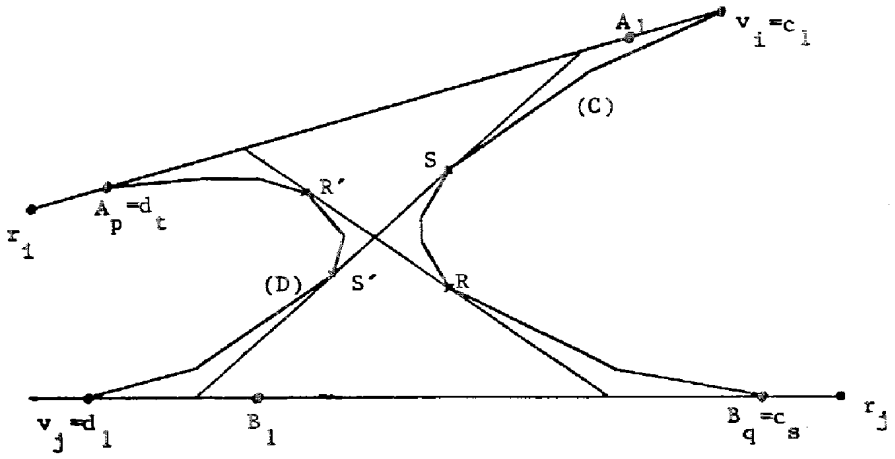


Figure 24 The limits on middle edged RR' and SS'.

the boundary of P which pass through A_1B_q (resp. A_pB_1), and delimits an area of safety for potential middle edges. Actually, as we will see later on, this definition will apply to the *parts* of C and D inside the quadrilateral $A_1A_pB_1B_q$. To preserve the flow of the presentation, we will prove the following result in the next section.

Lemma 18. The convex chains C and D can be computed in $O(c)$ time after $O(n + c^2 \log n)$ time preprocessing.

If the procedure of Lemma 18 determines that any segment drawn between A_1A_p and B_1B_q should intersect the boundary of P, it sets C or D to 0. Otherwise, it effectively returns two convex chains C and D, with the segment c_1d_t (resp. d_1c_s) containing A_1 and A_p (resp. B_1 and B_q). Also, as stated earlier, a segment joining A_1A_p and B_1B_q will intersect the boundary of P if and only if it intersects C or D.

It takes $O(c)$ operations to test whether C and D intersect since both have $O(c)$ vertices. If they do, no middle edge is possible and $E(i,j)$ is set to 0. Otherwise, we can compute RR' and SS' as the limits put upon the middle edges by the polygon P (Fig.24). SS' is computed by beginning at c_1 and d_1 and moving through D until all of D lies above the line passing through d_kc_1, then moving through C until all of C lies below the line passing through d_kc_1. We iterate on this process until termination, which will occur after $O(c)$ steps since no vertex of C or D is visited more than once. Here is a more formal description of the procedure. RR' is obtained in a similar manner.

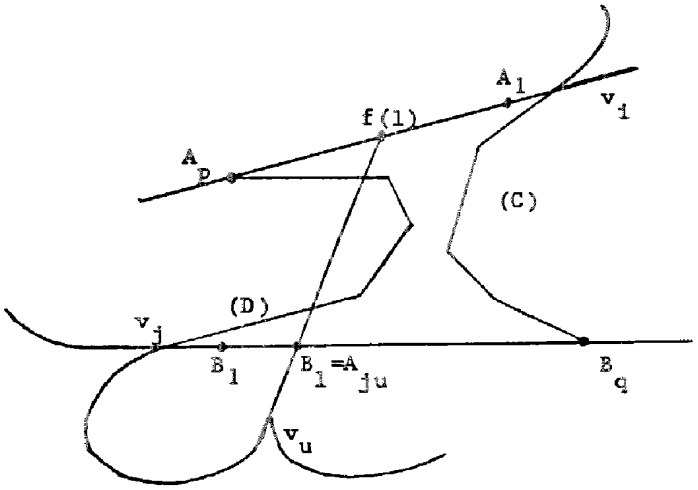


Figure 25 The function f.

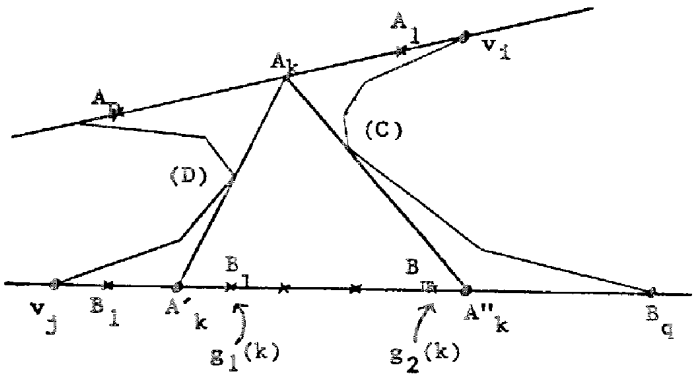


Figure 26 The functions g_1 and g_2.

Computing SS'

k:=l:=1

while $\angle(d_k d_{k+1}, d_k c_l) < 180$ or $\angle(c_l c_{l+1}, c_l d_k) < 180$

beg

while $\angle(d_k d_{k+1}, d_k c_l) < 180$

begin k:=k+1 **end**

while $\angle(c_l c_{l+1}, c_l d_k) < 180$

begin l:=l+1 **end**

end

$S := c_l$; $S' := d_k$

III) *Computing wedges for possible middle edges*

We are now in a position to compute the wedges where middle edges must lie. We begin by observing that any point in the list r_i not in the wedge formed by RR' and SS' can be discarded, which we can do in $O(e)$ time. For simplicity we still call $A_1, \ldots, A_p$ the elements of the list r_i.

For every $B_l \in r_j$ (recall that $B_l = A_{ju}$ for some u) we define $f(l)$ as the intersection of the line supporting r_i and the ray emanating from B_l, in the direction of R_u, and not passing through R_u. If this intersection does not exist, we compare the direction of R_u and r_i to decide if the "middle edge" wedge centered at B_l formed by the ray and r_j intersects the line supporting r_i or not. If yes, we set $f(l) = A_p$, else $f(l) = 0$ (Fig.25). Also, for each $A_k \in r_i$, let A'_k (resp. A''_k) denote the point on the segment r_j which is the closest to B_1 (resp. B_q) and such that the segment $A_k A'_k$ (resp. $A_k A''_k$) does not strictly intersect D (resp. C), i.e. the intersection consists of a segment or a single point. We now view r_j as a list of vertices and we find the two vertices B_l $(= g_1(k))$ and B_m $(= g_2(k))$ which lie on the segment $A'_k A''_k$ and are the closest to A'_k and A''_k, respectively (Fig.26). If B_l and B_m do not exist, we define $g_1(k)$ and $g_2(k)$ to be 0. At this stage we need two results whose proofs we postpone till the next section.

Lemma 19. After $O(c)$ preparation, the functions f, g_1, g_2 can be evaluated in constant time.

Lemma 20. Let $V = \{1, \ldots, q\}$ and $W = \{1, \ldots, p+q\}$, $W_1 \bigcup W_2 = W$, $|W_1| = q$, $|W_2| = p$ and let f, g_1, g_2 be defined with: $f \mid V \mapsto W_1$ a bijection and $g_1, g_2 \mid W_2 \mapsto V$ non-decreasing functions, with $g_1(i) \leq g_2(i)$ for each $i \in W_2$. Define $x \mid W_2 \mapsto V$ such that for each $i \in W_2$, $x(i)$ is the smallest integer in V with $g_1(i) \leq x(i) \leq g_2(i)$ and $i \leq f(x(i))$ if such an integer exists, and 0 otherwise. If f, g_1, g_2 are computable in constant time, then so is x after $O(p+q)$ preprocessing.

IV) *Computing $E(i,j)$*

Lemma 19 allows us to compute all the values of f, g_1, g_2 in $O(c)$ time. Note that if $f(l) = 0$ no middle edge adjacent to B_l is possible since it must lie in the wedge $W(B_l B_q, B_l f(l))$. Therefore we can eliminate those B_l from r_j. Once again we still represent the resulting list by $B_1, \ldots, B_q$. Similarly, if $g_1(k) = g_2(k) = 0$, A_k cannot be an N3-node and we eliminate all such A_k from r_i. Note that the values of g_1 and g_2 should be computed after the last selection on r_j. We will merge the points $f(l)$ with the remaining vertices A_k, thus forming the set W. Strictly speaking, f maps l not to a point on r_i but to the corresponding index in W. We can always assume that f is injective. Next we define V as the set of vertices left on r_j; instead of mapping to actual vertices of r_j, the functions g_1 and g_2 will map k to the corresponding indices in V. Because of the removals, g_1 and g_2 obey the two conditions of Lemma 20. Finally we define W_1 as the sublist of W corresponding to the $f(l)$ and W_2 as the complement in W, i.e. the indices corresponding to the A_k. It is easy to see that all removals, merges, and settings of functions can be done in $O(c)$ time. Moreover all the conditions of Lemma 20 have been met.

We can thus set up the function x in $O(c)$ time. The last step consists of keeping in $E(i,j)$ all the pairs $(A_k, B_{x(A_k)})$ such that $x(A_k) \neq 0$, with $A_k \in r_i$ and $\angle(A_k v_u, A_k B_{x(A_k)}) < 180$ $(A_k = A_{iu})$. Note that $x(A_k)$ is a shorthand for $x(t)$, with t the element in W_2 corresponding to A_k. Recall that if any of the previous computations fails, we have $E(i,j) = 0$. We can now state our main result:

Lemma 21. After $O(n + c^2 \log n)$ preparation, $E(i, j)$ can be evaluated in $O(c)$ time.

Proof: The preprocessing involves setting up the lists r_i which, as seen earlier, requires $O(n + c^2 \log n)$ time. For any i and j, $E(i, j)$ is then computed in $O(c)$ operations. We review the main phases of the procedure and establish its correctness. In the preprocessing stage, Fact 1 ensures that the angles $\angle x < 180$ and $\angle x' < 180$, and that Condition 1 of Lemma 17 is satisfied. Facts 3 and 4 show that $\angle y < 180$ and $\angle z' < 180$. Then, considering the savings, by an argument now standard, we eliminate the N3-nodes which are not candidates. Next we pair $A_k \in r_i$ with $B_l \in r_j$ $(l = x(A_k))$.

By definition, $x(A_k)$ is the smallest integer in V (i.e. the vertex of r_j that maximizes the angle $/\, y$) lying on the segment $(g_1(k), g_2(k))$ (i.e. ensuring Condition 3 of Lemma 17), and such that A_k lies on the segment $A_1 F$, with F the point on r_i corresponding to $f(x(A_k))$, i.e. ensuring $\angle y' < 180$. Finally, since $x(A_k)$ maximizes the angle $/\, y$ and $\angle(y + z)$ is a constant for all B_l, no vertex of r_j can be paired with A_k if $/\, z > 180$. If $\angle z \leq 180$, all the conditions of Lemma 17 are satisfied, and $A_k B_{x(A_k)}$ can be kept as the middle edge candidate to be adjacent to A_k. Since we know that this edge is indeed the middle edge of a loose X_4-pattern, only savings consideration will later decide whether this edge belongs to an OCD or not. This is, in essence, the only major difference with the Y-subtrees of $B(i, j)$ and $F(i, j)$, where both savings and geometry had to be tested at once in order to determine candidacy. ∎

2. The Proofs of the Lemmas Left Unresolved

We now justify our earlier claims and successively show how to compute the region of safety and set up the functions f, g_1, g_2, x, all in $O(c)$ time.

Lemma 18. The convex chains C and D can be computed in $O(c)$ time after $O(n + c^2 \log n)$ time preprocessing.

Proof: C and D are computed in the same manner, so we may concentrate on C exclusively. We assume that all the superranges have been precomputed, which requires $O(n + c^2 \log n)$ time, as was shown in Lemma 10. Let (a_p, b_p) be the pair of $\mathrm{SR}(v_j)$ such that the segment $a_p b_p$ intersects r_j. Recall that this pair is uniquely defined and must be computed in order to obtain

r_j. If a_p is a notch let w be a_p, otherwise let w be the notch of P next to a_p, counterclockwise. We may always assume that adequate preprocessing allows us to determine w in constant time, given a_p. If v_i lies between w and v_j in clockwise order, we simply define C as v_iB_q (Fig.27-c). Otherwise things are somewhat more complex, and before describing an algorithm formally we will give an overview of the method.

Let L be the line collinear with the edge of C adjacent to v_i (i.e. the first edge of C counterclockwise). L will essentially wrap around the obstacles created by the boundary of P in a counterclockwise motion (Fig.27-d). Let V be the polygonal line on the boundary of P between v_i and w in clockwise order. We will show that all the *obstacles* (which are the vertices of C) are notches of V. Consequently we can expect to wrap around C entirely in $O(c)$ operations if L can pivot around each vertex of C in constant time on the average.

Let x be a vertex of C with L_1 (resp. L_2) designating the line L before (resp. after) pivoting around x (Fig.27-d). We first locate L_1 in the superrange of x, then we scan SR(x) counterclockwise, until we hit a vertex b_k which lies on V. We can show that in general b_k is also the next vertex of C. Recall that locating L in SR(x) involves finding the pair (a_j, b_j) such that L intersects a_jb_j. To ensure an $O(c)$ running time, we cannot actually locate L_1 in SR(x). Instead we determine a notch y nearby which will serve the same purpose. This notch is to be determined at the time when L_1 is computed. Thus we define the function NEXT which maps (x, y) to (a_k, b_k). More generally, NEXT maps any pair of notches x, y ($x \in V$) to the pair (a_k, b_k) of SR(x), computed as follows:

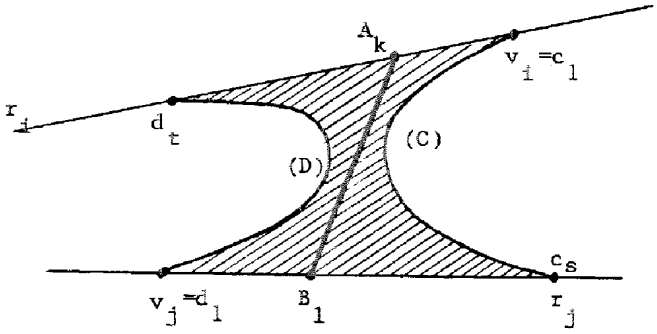
1. Find the two pairs (a_j, b_j) and (a_{j+1}, b_{j+1}) of SR(x) such that y lies between b_j and a_{j+1} in clockwise order.
2. Scan the pairs of SR(x) counterclockwise, starting at (a_j, b_j) (i.e. $(a_j, b_j), (a_{j-1}, b_{j-1}), \ldots$) and determine the pair (a_k, b_k) such that b_k is a notch of V and a_{k+1} lies outside of V. If we fail to find such a pair, return (0).
3. When NEXT is evaluated and a pair (a_k, b_k) is actually returned, the function sets a global variable *cnext* to a_{k+1} if it is a notch, else the notch of P next to a_{k+1}, counterclockwise (Fig.27-d).

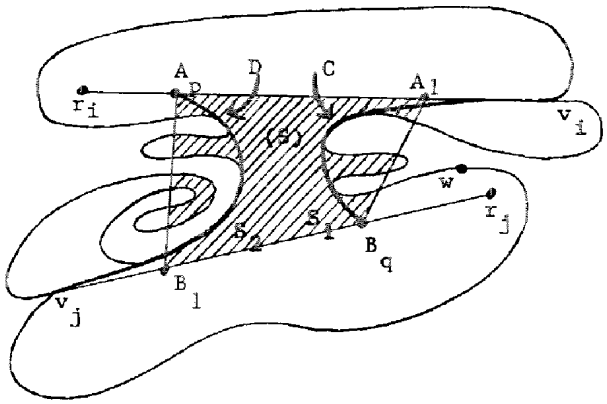
We first show that the function NEXT is well-defined and can be evaluated in time proportional to the number of pairs scanned in step **2**. Recall that in any superrange $\mathrm{SR}(x)$, the pairs (a_j, b_j) realize a partition of the set of all notches, and more precisely any notch y lies between b_j and a_{j+1} in clockwise order for some j. Thus, once $\mathrm{SR}(x)$ has been computed, we can extend the preprocessing to assign each notch of P to each corresponding pair b_j, a_{j+1}. A simple scan through the notches of P will do it in $O(c)$ time. Finally, noticing that we can test if a notch lies in V in constant time, and that *cnext* is also found in constant time, for the reasons seen above, we achieve our claims. We are now ready to set out the algorithm for computing C. Let $a_l b_l$ be the segment intersecting r_i with (a_l, b_l) in $\mathrm{SR}(v_i)$ and let t be the notch next to a_l counterclockwise. Let γ be the intersection $r_i \bigcap r_j$ if it is defined, or the endpoint of r_j $(\neq v_j)$ otherwise.

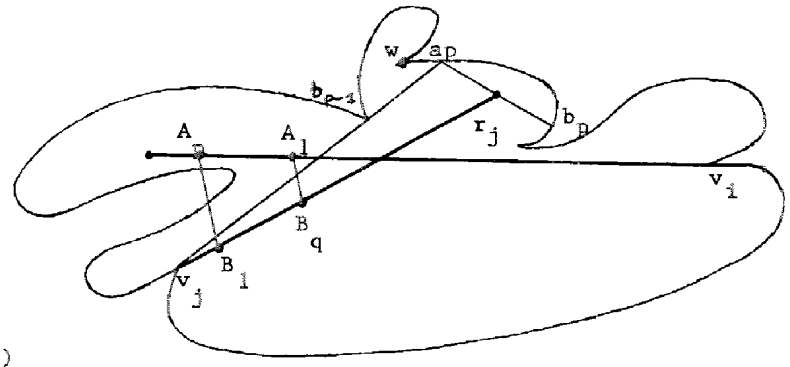
Computing C

if v_i lies between w and v_j
  then return $(C = \{v_i, \gamma, B_q\})$
C:=$\{v_i\}$, d:= v_i, e:= NEXT (v_i, t)
while $e \neq 0$
  begin
    if $\angle(dB_q, de) < 180$
      then return $(C := C \bigcup \{B_q\})$
    d:= e
    C:= $C \bigcup \{d\}$
    e:= NEXT $(e, cnext)$
  end
return $(C := 0)$

To see that the algorithm runs in $O(c)$ time, it suffices to note that the notch *cnext* moves counterclockwise on the boundary of P, so $O(c)$ pairs (a_j, b_j) will be examined in all the superranges considered by the function NEXT.







(Figure 27 .../...)

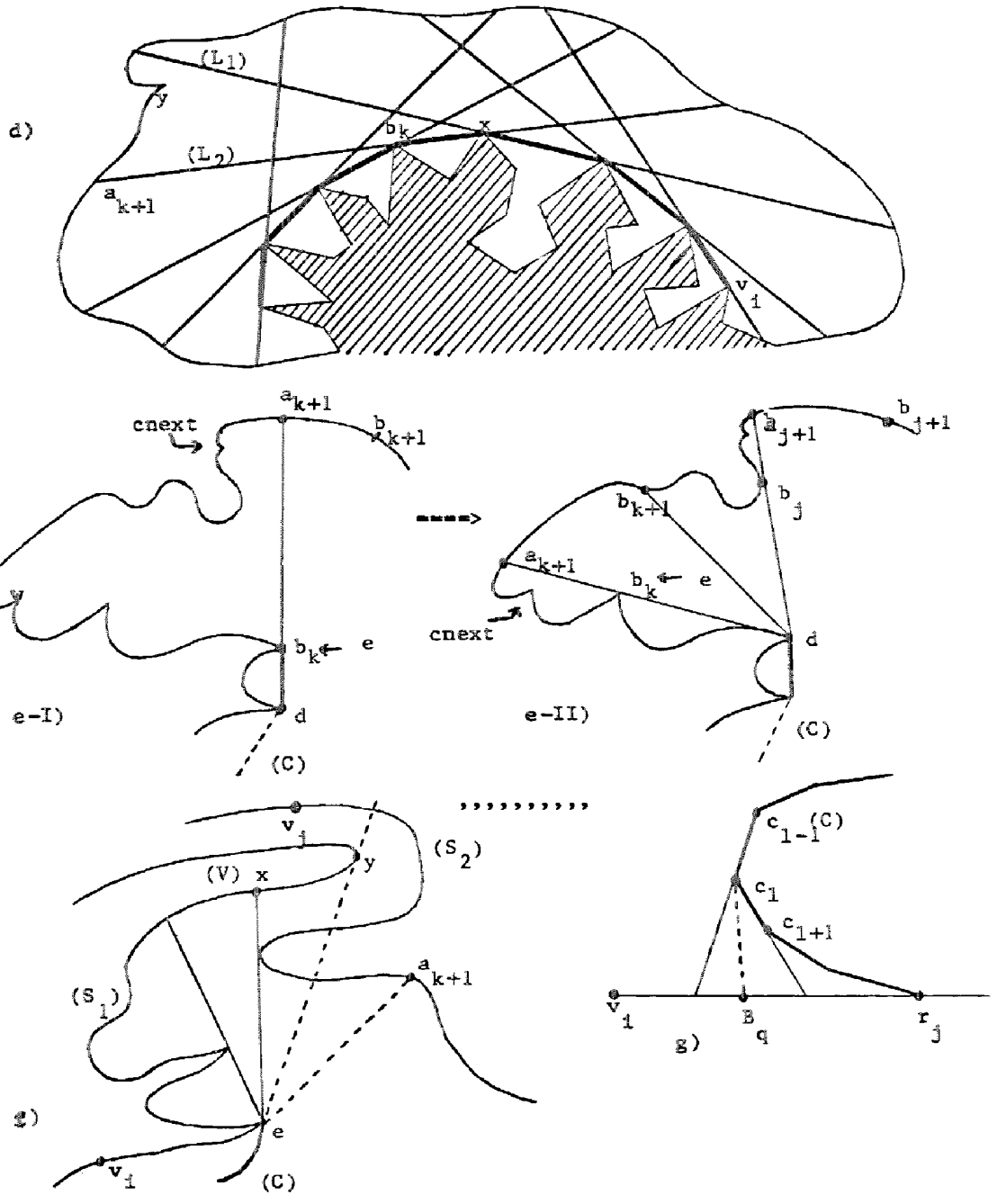


Figure 27 Computing the chains C and D.

We next show how a middle edge intersects the boundary of P if and only if it intersects C or D. In order to connect r_i and r_j via a middle edge, the intersection of P and the quadrilateral $A_1B_qB_1A_q$ must contain a polygon S with edges A_1A_p and B_1B_q. Note that this intersection may actually consist of several polygons. The boundary of S consists of these two segments joined by two polygonal lines, S_1 and S_2. S_1 has all its vertices in V and S_2 in V', where V' is defined as V, switching the roles of v_i and v_j (Fig.27-b).

To avoid computing S_1 and S_2 explicitly (they may have on the order of n vertices), we first notice that no middle edge can intersect the convex hull of S_1 without intersecting S_1. The same observation on S_2 leads to computing the convex hulls C_1 and C_2, respectively. For convenience, we can replace the vertex A_1 in S_1 (resp. B_1 in S_2) by v_i (resp. v_j) and still preserve the initial property that a middle edge lies totally in P if and only if it does not strictly cross C or D. We now turn to the actual computation of C and D.

The first case considered assumes that v_i lies between w and v_j in clockwise order. S is then reduced to the single intersection point of r_i and r_j and S can be set to v_iB_q (Fig.27-c). Note that we can always assume that in this case r_i and r_j intersect, otherwise the lists r_i and r_j would be empty. If v_i does not lie between w and v_j, V is not empty, and we will prove by induction that C is actually the convex hull of S_1 or 0 if no middle edge is possible. Fig.27-e illustrates the computation of the next edge of C. To ensure convexity, all of S_1 must lie on the same side of the line passing through this edge. Therefore the next vertex of C after the vertex labelled e in Fig.27-e-I) must be the point x of V, visible from b_k, which minimizes the angle $\angle(ea_{k+1}, ex)$. b_k and a_{k+1} are the vertices in $\mathrm{SR}(d)$ returned by the previous call on NEXT.

Since the endpoints of V are notches of P, x must be one of the vertices listed in $\mathrm{SR}(e)$ between which a_{k+1} lies, so we must start scanning $\mathrm{SR}(e)$. We only have to perform a counterclockwise scan since, by induction hypothesis, a_{k+1} does not lie in V, therefore the next vertex of C must be some a_l or b_l in $\mathrm{SR}(e)$ for $l \leq j$. Once again, the crux is that a counterclockwise scan in $\mathrm{SR}(e)$ corresponds both to a counterclockwise scan through the vertices of P and a counterclockwise angular sweep. Note that a_{k+1} is a point of $\mathrm{SR}(d)$ which has to be located in $\mathrm{SR}(e)$. Since a_{k+1} is not a notch in general this operation seems too complex.

Instead we can find the pair b_j, a_{j+1} of SR(e) between which *cnext* lies (recall that *cnext* is the first notch after a_{k+1} in a counterclockwise scan through the boundary of P). Since *cnext* is a notch, this operation can be done in constant time. Thus the function NEXT will return the next vertex of C. Note that if the point determined by NEXT is not a notch, NEXT returns 0. Indeed, this point could not be the next vertex of C and the actual next vertex y would not be visible from e. Consequently, C would intersect S_2 and no middle edge would then possible (Fig.27-f). We observe that if C is well-defined, there exists l such that B_q lies in the wedge $W(C_{l-1}C_l, C_lC_{l+1})$ (Fig.27-g). Thus the algorithm terminates by substituting B_q for $C_{l+1}, C_{l+2}, \ldots$ and the remaining vertices of C; this is legitimate since this last portion of C cannot have any effect on middle edges. ∎

Lemma 19. In $O(c)$ time, it is possible to precompute the functions f, g_1, g_2 so that any evaluation can be done in constant time.

Proof: $f(l)$ can be evaluated directly in constant time by intersecting the line passing through r_i with the ray emanating from B_l, collinear with R_u, yet not passing through R_u with $B_l = A_{ju}$. If there is no intersection, $f(l) = 0$. Next we show how to compute all the values of g_1 and g_2 in $O(c)$ time. We start by computing the intersection of r_i with all the lines (d_kd_{k+1}) for consecutive values of k. These points partition r_i into segments, and the previous computation provides a sorted list of their endpoints in $O(c)$ time. To each of these segments corresponds a unique vertex of D. Then, for each $A_1, \ldots, A_p$ in turn, we find the segment where A_k lies. Let d_m be the corresponding vertex of D. We compute A'_k by intersecting r_j with the line passing through A_kd_m (Fig.28). This also gives us a sorted list of the points A'_k, since D is convex. Finally we can merge the A'_k and B_l in $O(c)$ time, and in one scan through the list, find for each A'_k the nearest B_l on the same side as B_q. We then set $g_1(k)$ to 1. We iterate on this process with respect to C, defining $g_2(k)$ for each A_k on r_i. Finally, for each A_k, we check that $g_1(k) \leq g_2(k)$. If this is not the case, we set $g_1(k)$ and $g_2(k)$ to 0. All of this work clearly requires $O(c)$ time. ∎

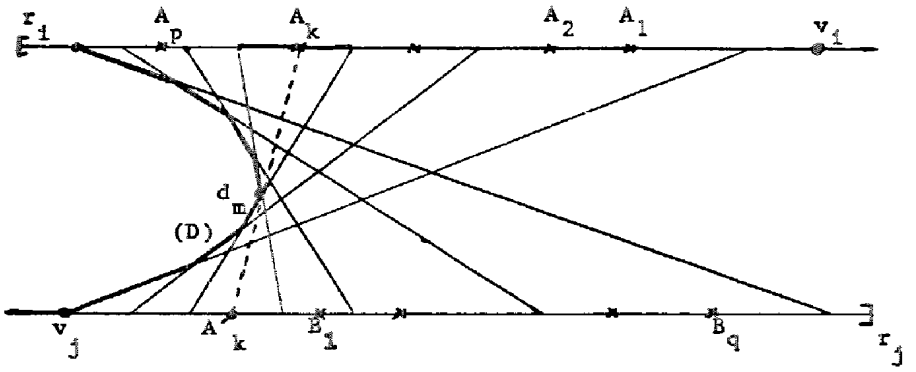


Figure 28 Computing g_1.

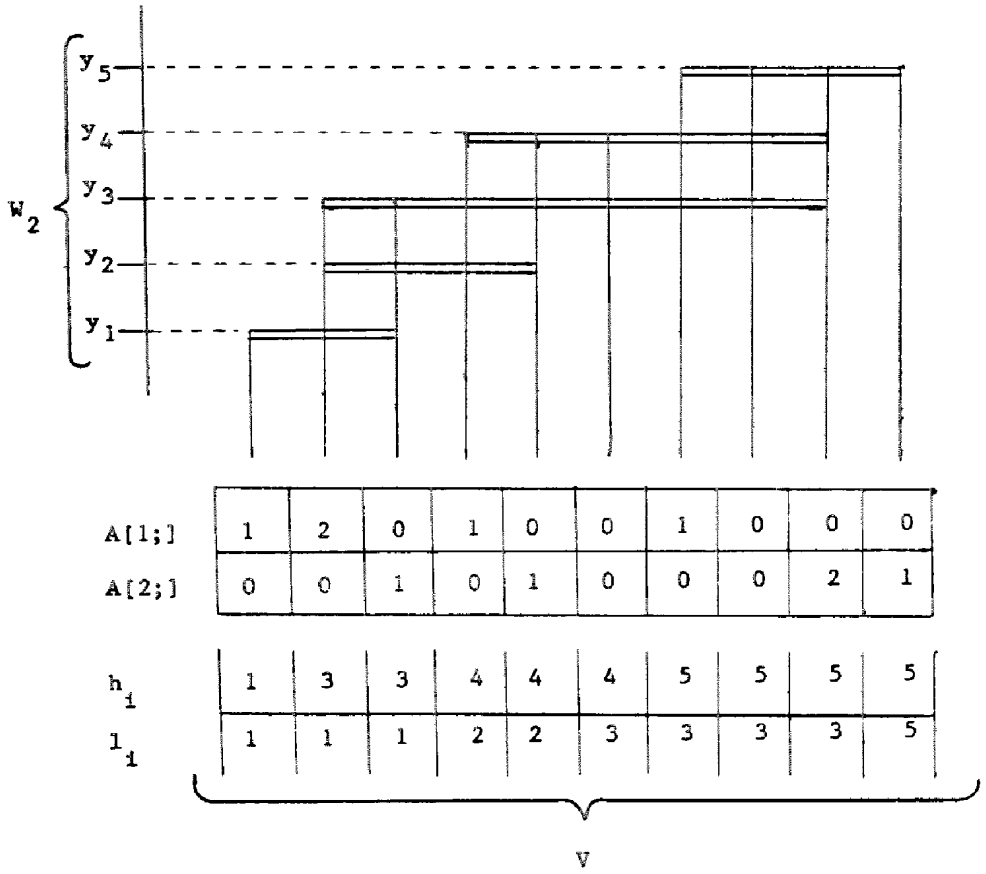


Figure 29 The setting of the function x.

Lemma 20. Let $V = \{1, \ldots, q\}$ and $W = \{1, \ldots, p+q\}$, $W_1 \bigcup W_2 = W$, $|W_1| = q$, $|W_2| = p$ and let f, g_1, g_2 be defined with: $f \mid V \mapsto W_1$ a bijection and $g_1, g_2 \mid W_2 \mapsto V$ non-decreasing functions, with $g_1(i) \leq g_2(i)$ for each $i \in W_2$. Define $x \mid W_2 \mapsto V$ such that for each $i \in W_2$, $x(i)$ is the smallest integer in V with $g_1(i) \leq x(i) \leq g_2(i)$ and $i \leq f(x(i))$ if such an integer exists, and 0 otherwise. If f, g_1, g_2 are computable in constant time, then so is x after $O(p+q)$ preprocessing.

Proof: Note that the naive method for computing all the values of x runs in $O(pq)$ time. We present an $O(p+q)$ time algorithm for achieving all these computations and establish its correctness. Let $y_1, \ldots, y_p$ be the elements of W_2 in increasing order ($1 \leq y_j \leq p+q$). First of all, we consider the set of $y \in W_2$ such that $g_1(y) \leq i \leq g_2(y)$ for a given i between 1 and q, and observe that it is a contiguous (possibly empty) subset of W_2 since g_1 and g_2 are non-decreasing. We compute the largest and smallest y, denoted y_{h_i} and y_{l_i} respectively, as follows (if there is no such y, we set (l_i, h_i) to 0).

Initialize an array A $(2 \times q)$ to 0.
for $i = 1, \ldots, p$
  begin
    A$[1, g_1(y_i)]$:= A$[1, g_1(y_i)]$+1
    A$[2, g_2(y_i)]$:= A$[2, g_2(y_i)]$+1
  end
l:= 1; h:= 0
for $i = 1, \ldots, q$
  begin
    h:= h+A$[1, i]$
    if $l \leq h$
      then (l_i, h_i):= (l, h)
      else (l_i, h_i):= 0
    l:= l+A$[2, i]$
  end

The algorithm clearly achieves a time bound of $O(p+q)$. To establish its correctness, we observe that the first loop sets $A[1,j]$ to the number of y in W_2 such that $j = g_1(y)$, i.e. the number of intervals $[g_1(y), g_2(y)]$, starting at j. Similarly, $A[2,j]$ counts the number of intervals finishing at j. Then since, as i increases, $g_1(y_i)$ and $g_2(y_i)$ cannot decrease or pass each other, we can derive (l_i, h_i) from (l_{i-1}, h_{i-1}) by counting the number of intervals which have to be added and removed. More precisely, the difference $h_i - h_{i-1}$ is exactly the number of y in W_2 such that $i-1 < g_1(y) \leq i$, which is equivalent to $g_1(y) = i$ and shows that this number is $A[1,i]$. Likewise, if $i-1 < g_2(y_{l_{i-1}})$ we have $l_i = l_{i-1}$. Else if $g_2(y_{l_{i-1}}) = i-1$, $l_i - l_{i-1}$ is the number of y such that $g_2(y) = i-1$, i.e. $A[2,i]$ (see example in Fig.29).

We are now ready to set up the function x by computing all its values. If $y_{l_1} \leq f(l) \leq y_{h_1}$, all $x(i)$ with i between y_{l_1} and $f(l)$ must be set to 1. Now if $y_{l_1} \leq y_{h_1} \leq f(l)$, all $x(i)$ with i between y_{l_1} and y_{h_1} must be set to 1. In both cases, no other i in W should have $x(i)$ equal to 1. Then we can carry out the same reasoning with 2, assigning this value only to the $x(i)$ which have not been set yet. Since, as i increases from 1 to q, l_i and h_i cannot decrease or pass each other (unless $(l_i, h_i) = 0$) a possible implementation is:

```
Initialize all x(i) to 0 for all i = 1,...,q
M:= 0
for i = 1,...,q
  begin
    a:= max(y_{l_i}, M)
    b:= min(f(i), y_{h_i})
    if a ≤ b and (l_i, h_i) ≠ 0
      then
        for j = a,...,b
        begin x(j):= i end
        M:= b + 1
  end
```

Note that if i belongs to W_1, the value assigned to $x(i)$ is meaningless. The algorithm runs in time $O(p+q)$, which completes the proof. ∎

3. The Cubic Algorithm

We are now prepared to use the information contained in $E(i,j)$ to produce an OCD. $E(i,j)$ may be computed in STEP 4 of the algorithm *ConvDec*, with the additional preprocessing described earlier. We can now replace the former computation of M in STEP 3 by the following:

Initialize M as the empty set.

For each $v_k \in V(i+1, j-2)$ such that $E(k,j)$ contains a pair (A_{ki}, B_{jl}), assign to M the maximum (with respect to cardinality) of M and

$$X4(v_i, v_k, v_l, v_j, A_{ki}, B_{jl}) \bigcup S(i+1, k-1) \bigcup S(k+1, l-1) \bigcup S(l+1, j-1).$$

For each $v_k \in V(i+2, j-1)$ such that $E(i,k)$ contains a pair (A_{ij}, B_{kl}), assign to M the maximum (with respect to cardinality) of M and

$$X4(v_j, v_i, v_l, v_k, A_{ij}, B_{kl}) \bigcup S(i+1, l-1) \bigcup S(l+1, k-1) \bigcup S(k+1, j-1).$$

Note that we investigate the two possible topologies of an X_4-pattern lying in $V(i,j)$ and adjacent to v_i and v_j (Fig.30). The procedure for computing M requires $O(c)$ time and, from Lemma 21, we know that the additional preprocessing requires $O(n + c^2 \log n)$ time. We are now ready to evaluate the complexity of each step of the decomposition algorithm:

- Preprocessing: $O(n + c^3 + c^2 \log n)$
- Step 2: $O(c^3)$
- Step 3: $O(c^3)$
- Step 4: $O(c^3)$
- Step 5: $O(n + c^2 \log n)$

The total running time of the decomposition algorithm is therefore $O(n + c^3 + c^2 \log n)$, which is easily shown to be also $O(n + c^3)$.

Main Theorem. It is possible to compute an optimal convex decomposition of a simple polygon with n vertices and c reflex angles in time $O(n + c^3)$.

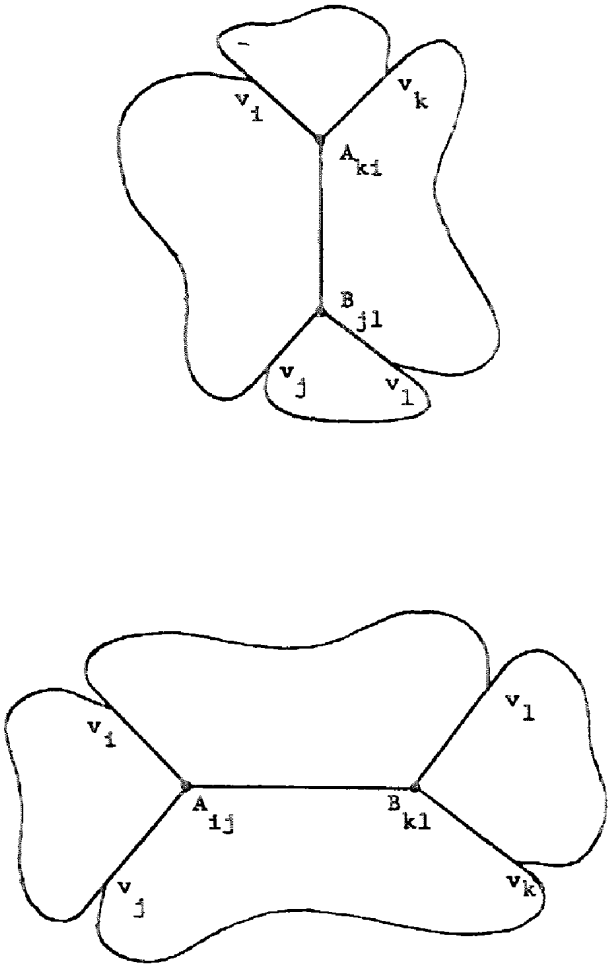


Figure 30 Computing B with the topologies of X4-pattern

We should note that in $S(i, i-1)$ the algorithm provides us with c optimal, yet not all necessarily identical, decompositions.

5. Concluding Remarks

The main result of this paper is an algorithm for decomposing a polygon into a minimum number of convex parts. The algorithm is linear in the number of vertices and cubic in the number of reflex angles. From a theoretical viewpoint, our main achievement has been to show that the decomposition problem was polynomial, even when Steiner points are allowed in the decomposition. Also one merit of our algorithm is to have its complexity expressed in the form $O(n + f(c))$. This is to our knowledge the only decomposition algorithm with this property; one interesting open problem is to decide whether the preprocessing used in our algorithm can be applied to the algorithms known for the case where no Steiner points are allowed in the decomposition. Indeed, Greene's $O(n^2c^2)$ and Keil's $(c^2n \log n)$ methods for this problem are less efficient than our algorithm.

On the practical side, the complexity of our algorithm might be acceptable, given the near-convexity of most polygons in practice. Unfortunately the algorithm seems inherently intricate and implementing it in its most elaborate form is certainly a formidable task. We might be willing, however, to sacrifice a little efficiency in order to achieve greater simplicity. Computing only X_2-patterns and doing away with superranges may often be found an acceptable compromise. Even the naive decomposition, when implemented efficiently, may turn out the best alternative if we can afford to miss an optimal solution by at worst a factor of two in the number of convex parts.

Of course, only the cubic algorithm reveals the genuine "beauty" of the problem. Its long development involves many subproblems, most of which are interesting in their own right (we believe). For example the concept of superranges might provide an effective means of dealing with visibility problems and its fast computation $(O(c \log n))$ makes it very appealing. This can be viewed as a first step towards adapting non-convexity to algorithms for convex designs.

Acknowledgment: We thank Diane Souvaine for reading this (long) manuscript very carefully and giving us many helpful comments and suggestions.

REFERENCES

[1] Asano, T., Asano, T. *Minimum partition of polygonal regions into trapezoids*, Proc. 24th Annual FOCS Symp., 1983, pp. 233–241.

[2] Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[3] Chazelle, B. *Computational geometry and convexity*, PhD Thesis, Yale University, 1980. Also available as Technical Report CMU–CS–80–150, Carnegie-Mellon University, Pittsburgh PA, 1980.

[4] Chazelle, B. *Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm*, SIAM J. on Computing, August 1984.

[5] Chazelle, B., Dobkin, D.P. *Decomposing a polygon into its convex parts*, Proc. 11th Annual SIGACT Symp., pp. 38–48, 1979.

[6] Chazelle, B., Dobkin, D. *Detection is easier than computation*, Proc. 12th Annual SIGACT Symp., pp. 146-153, 1980.

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

[8] Feng, H., Pavlidis, T. *Decomposition of polygons into simpler components: Feature generation for syntactic pattern recognition*, IEEE Trans. Comp., Vol. C-24, No. 6, June 1975, pp. 636-650.

[9] Ferrari, L., Sankar, P.V. and Sklansky, J. *Minimal rectangular partitions of digitized blobs*, Proc. 5th International Conference on Pattern Recognition, Miami Beach, Dec. 1981, pp. 1040-1043.

[10] 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.

[11] Graham, R.L. *An efficient algorithm for determining the convex hull of a finite planar set*, Info. Proc. Lett., 2, pp. 18-21, 1973.

[12] Greene, D.H. *The decomposition of polygons into convex parts*, Advances in Computing Research, Jay Press, pp. 235–259, 1983.

[13] Keil, M. *Decomposing polygons into simpler components*, PhD Thesis, University of Toronto, 1983.

[14] Lingas, A. *The power of non-rectilinear holes*, Proc. 9th Colloquium on Automata, Languages and Programming, Aarhus, 1982.

[15] Lingas, A. *Heuristics for minimum edge length rectangular decomposition*, Unpub. Man., MIT, Nov. 1981.

[16] Lingas, A., Pinter, R., Rivest, R. and Shamir, A. *Minimum edge length partitioning of rectilinear polygons*, Proc. 20th Annual Allerton Conference on Communication, Control, and Computing, Monticello, Ill., Oct. 1982.

[17] O'Rourke, J., Supowit, K.J. *Some NP-hard polygon decomposition problems*, IEEE Trans. Information Theory, 29(2), 1983, pp. 181–190.

[18] Schachter, B. *Decomposition of polygons into convex sets*, IEEE Trans. on Computers, C 27, pp. 1078–1082, 1978.

[19] Shamos, M.I. *Computational geometry*, PhD thesis, Yale University, 1978.

[20] Toussaint, G.T. *Pattern recognition and geometrical complexity*, Proc. 5th Int. Conf. on pattern recognition, Miami Beach, 1980, pp. 1324–1347.

[21] Toussaint, G.T. *Decomposing a simple polygon with the relative neighborhood graph*, Proc. 18th Annual Allerton Conf. on Communication, Control, and Computing, Monticello, Ill., Oct. 1980.