Journal of Algorithms **37,** 538–569 (2000)
doi:10.1006/jagm.2000.1092, available online at http://www.idealibrary.com on IDEAL®

Safety Zone Problem

Subhas C. Nandy[1, 3] and Bhargab B. Bhattacharya

Indian Statistical Institute, Calcutta 700 035, India
E-mail: nandysc@isical.ac.in, bhargab@isical.ac.in

and

Antonio Hernández-Barrera[2, 3]

Computer Science Department, Faculty of Mathematics and Computer Science, Havana University, Cuba
E-mail: tonyhdz@matcom.uh.cu

Received March 5, 2000

Given a simple polygon P, its *safety zone* S (of width δ) is a closed region consisting of straight line segments and circular arcs (of radius δ) bounding the polygon P such that there exists no pair of points p (on the boundary of P) and q (on the boundary of S) having their Euclidean distance $d(p, q)$ less than δ. In this paper we present a linear time algorithm for finding the minimum area safety zone of an arbitrarily shaped simple polygon. It is also shown that our proposed method can easily be modified to compute the Minkowski sum of a simple polygon and a convex polygon in $O(MN)$ time, where M and N are the number of vertices of both the polygons.

Key Words: polygon triangulation; convex hull; Minkowski sum; resizing of VLSI circuits; algorithm; complexity.

1. INTRODUCTION

In this paper, we introduce a new problem called the *safety zone problem*, which is as follows: Given a simple polygon P and a fixed parameter δ, the safety zone (of width δ) of the polygon P is a closed

[1] To whom correspondence should be addressed.
[2] The research of the third author was supported by JSPS Fellowship No. P97029.
[3] This work was done when these authors were visiting Japan Advanced Institute of Science and Technology, Japan.

0196-6774/00 $35.00

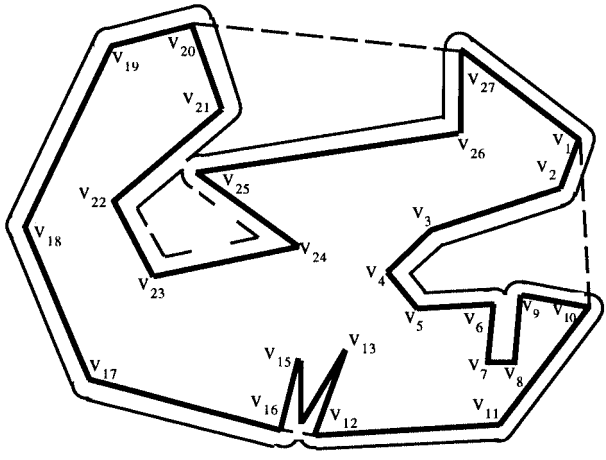


FIG. 1. Safety zone of a simple polygon.

region S of minimum area such that P is completely inside S and there exists no pair of points p and q, where p is on the boundary of P and q is on the boundary of S, such that $d(p, q)$, the Euclidean distance between p and q, is less than δ. Here S is not a polygonal region. Its boundary is composed of straight line segments and circular arcs of radius δ, where each straight line segment is parallel to an edge of the polygon at a distance δ from that edge, and each circular arc (of radius δ) is centered at a unique vertex of the polygon. The boundary of the safety zone describes a *simple* region in the sense that no two edges (straight line segment or circular arc) on its boundary intersect in their interior. It is easy to observe that for every point q on the boundary of the *safety zone* S of the polygon P there exists at least one point p on P such that $d(p, q) = \delta$. The safety zone of a simple polygon is demonstrated in Fig. 1. The problem originates in the context of VLSI resizing as described in the next section.

In this context, it is worth mentioning that, given two polygons P and Q in $\mathscr{R}^2$, their *Minkowski sum* is defined as $P \oplus Q = \{p + q : p \in P, q \in Q\}$, where $p + q$ denotes the vector sum of the vectors p and q, i.e., if $p = (p_x, p_y)$ and $q = (q_x, q_y)$, then we have $p + q = (p_x + q_x, p_y + q_y)$. The *safety zone* (of width δ) of a convex polygon P is surely obtained by taking the Minkowski sum of the polygon P and a circle C of radius δ. But for a simple nonconvex polygon, the *safety zone* is a super set of the region A obtained by taking the Minkowski sum of the polygon P and the circle C. Here the area indicated by the Minkowski sum may be composed

of the *safety zone* of the polygon P with some holes inside it such that the boundaries of the holes also satisfy the safety zone property.

The combinatorial complexity of the Minkowski sum of two arbitrary simple polygons P and Q is $O(M^2N^2)$ [3], where N and M are the number of vertices of these two polygons respectively. In particular, if one of these two polygons is convex, the size of the Minkowski sum reduces to $O(MN)$. In [6], an algorithm of time complexity $O(MN \log(MN))$ is proposed in the context of the polygon containment problem. The problem they have studied is deciding whether a convex polygon Q can be translated to fit within an arbitrary polygon P. In [2], a number of results are proposed on the Minkowski sum problem when one of the polygons is monotone.

If Q is convex and P is monotone, the size of the Minkowski sum as well as the time and space complexities of the algorithm are $O(MN)$.

If both P and Q are monotone, the size of the Minkowski sum is $O(MN\alpha(\min(M, N)))$, where $\alpha(\cdot)$ is the inverse of the Ackermann function. The time complexity of the proposed algorithm is $O(MN \log(MN))$.

If Q is monotone and P is any arbitrary simple polygon, the time complexity of the proposed algorithm is $O((k + MN)\log(MN))$, where k is the size of the Minkowski sum. The value of k may be $O(M^2N)$ in the worst case.

An algorithm for finding the outer face of the Minkowski sum of two simple polygons is presented in [11]. It uses the concept of convolution, and the running time of the algorithm is $O((k + (M + N)\sqrt{l})\log^2(M + N))$. Here M and N are the number of vertices of the two polygons; k and l represent the size of the convolution and the number of cycles in the convolution respectively. In the worst case, k may be $O(MN)$. If one of the polygons is convex, the algorithm runs in $O(k \log^2(M + N))$ time. To the best of our knowledge, no algorithm exists which can compute the boundary defined by the Minkowski sum of an arbitrary simple polygon and a circle or a convex polygon in time linear in the worst case size of output (combinatorial complexity) of the problem.

In this paper, we present an algorithm for finding the boundary of the minimum area safety zone of an arbitrarily shaped simple polygon. Using Chazelle's linear time triangulation algorithm [1], we have shown that the time complexity of our algorithm is $O(N)$, where N is the number of vertices of the polygon. The space complexity of the algorithm is $O(N)$. In this context, one may argue in two ways:

Chazelle's polygon triangulation is fairly complex. To avoid this difficulty, one may use a simple randomized algorithm [12] of time complexity $O(N \log^*N)$ for the practical implementation purpose.

One may draw the boundary of the safety zone of a simple polygon from its medial axis, which can be obtained in linear time [5]. But the above algorithm is fairly complex; it involves splitting the polygon into different forms of histograms. Moreover, it also uses Chazelle's linear time polygon triangulation algorithm. Surely, one may get a simple algorithm for finding the medial axis of a simple polygon whose expected time complexity is $O(N)$ [8]. But after getting the medial axis (Voronoi diagram), drawing the boundary of the safety zone or the Minkowski sum is also not very straightforward. It is as complex as drawing the boundary of the safety zone after triangulation of the polygon, using our algorithm. So, one level of computational complexity (drawing the medial axis) can easily be avoided by adopting our method for solving this problem.

Although we have used Chazelle's triangulation algorithm, after the triangulation step, our algorithm is very easy to implement. It uses only linear link lists and a binary tree as data structures, which can be tackled easily in a software program.

Next, we show that the technique of drawing the boundary of the safety zone is also applicable for finding the Minkowski sum of an arbitrary simple polygon and a rectangle in linear time. The same technique is then used to compute the Minkowski sum of a simple polygon and a convex polygon in $O(MN)$ time, where N and M are the number of vertices of both of the polygons respectively.

The paper is organized as follows. In the next section we shall discuss some important applications of this problem. Some preliminary concepts will be introduced in Section 3. In Section 4 we shall concentrate on describing the process of obtaining the boundary of the safety zone inside a notch of a nonconvex simple polygon. The stepwise description of the algorithm and its complexity analysis is given in Section 5. In Sections 6 and 7 we use the same technique to compute the Minkowski sum of a simple polygon and a convex polygon. Finally, we summarize our work in Section 8.

2. APPLICATIONS

In VLSI layout design, the circuit components must not be placed very closely in order to avoid electrical effects such as inductance, capacitance, etc., among the circuit components. The circuit components on a floor may be viewed as a set of polygonal regions on a two-dimensional plane. Each circuit component P_i is associated with a parameter δ_i such that a minimum clearance zone of width δ_i must be maintained around that circuit component. The regions representing the circuit components are in

general isothetic polygons, but may not always be limited to convex ones. The location of the safety zone (of specified width) for a simple polygon is a very important problem for resizing the circuit components [10, 13]. Given a set of isothetic nonoverlapping polygonal regions and a common resizing parameter δ, an $O(N \log N)$ time and $O(N)$ space algorithm for finding the safety zones of all polygons is already available in [10]. This outputs another set of closed regions resizing each polygon by an amount δ. If more than one polygon are closed enough, their safety zones overlap, indicating the violation of the minimum separation constraint among them. Note that, inside a notch of such a polygonal boundary, if a wide space is available which may accommodate some circuit component, we cannot use that location, as space for routing the connection wires from those circuit components to the other circuit components which are placed outside the notch is not available. Thus with respect to resizing problems in VLSI, this is the motivation of defining the safety zone of a polygon.

The *safety zone problem* finds another important application in the automatic monitoring of metal cutting tools. Here a metal sheet is given and the problem is to cut a polygonal region of specified shape from that sheet. The cutter is a *PEN* whose tip is a small ball of diameter δ, and it is monitored by a software program. If the holes inside the notch also need to be cut, our algorithm can easily be tailored to satisfy that requirement, too.

The Minkowski sum is an essential tool for computing the free configuration space of translating a polygonal robot. It also finds application in the polygon containment problem and in computing the buffer zone in geographic information systems, to name only a few.

3. PRELIMINARIES

First of all, we classify a vertex of the polygon P as *concave* or *convex* depending on whether the angle between its associated edges inside the polygon is greater than or less than 180°. Consider the convex hull $CH(P)$ of the polygon P. Choose a vertex v of $CH(P)$ and label the vertices of P as $v_1(=v), v_2, \ldots, v_N$, moving along its boundary in clockwise direction. Now consider a pair of *PEN*s, say PEN_1 and PEN_2, whose tips are always a distance δ apart. The tip of PEN_1 moves along the polygonal boundary from vertex v_1 in the clockwise direction, and the tip of PEN_2 draws the boundary of the safety zone, staying orthogonal to the direction of motion of PEN_1. From now on we shall refer to the boundary of the safety zone as the *safe boundary*. Our objective is to compute the safe boundary of a simple polygon. Note that, while drawing the safe boundary, (i) if PEN_1 finds a convex vertex then the direction of PEN_2 is changed by drawing a

circular arc with center at that convex vertex, and (ii) if PEN_1 residing at a point p_1 finds another point p_2 on the boundary of the polygon such that $p_1p_2 = 2\delta$ and p_1p_2 is completely outside the polygon, then it moves to p_2 and PEN_2 will remain in its present position. Then PEN_1 starts moving toward the higher numbered vertex attached to that edge; the movement of PEN_2 will be guided by that of PEN_1, as described earlier.

It is easy to observe that the safety zone of an n vertex convex polygon is a convex region, where the number of line segments as well as the number of circular arcs bounding this region are both N. The straight line segments of S are parallel to the arms of the polygon at a distance δ outside the polygon, and two consecutive line segments of S are joined by a circular arc of radius δ with its center at the corresponding vertex of the polygon. Surely, the time required for drawing the safe boundary of a convex polygon is $O(N)$.

We now concentrate on describing the method of drawing the safe boundary of nonconvex simple polygons. Given a polygon P, we first consider the convex hull $CH(P)$ of the polygon P and label the vertices as stated above. Each hull edge may be classified as any of the following two types: (i) if it coincides with some edge of P it is called a *solid hull edge*, and (ii) if it does not coincide with any of the edges of P it is called a *false hull edge*.

DEFINITION 1. A *notch* is a polygonal region outside the polygon P which is formed with a chain of edges of P initiating and terminating at two vertices of a *false hull edge*. Clearly the area $CH(P) - P$ consists of a number of disjoint *notches* outside the polygon P.

The safety zone of the polygon P is inscribed by the safe boundaries of the *solid hull edges*, the safe boundaries of all the *notches*, and circular arcs of radius δ centered at the hull vertices. These circular arcs at the convex vertices will also be referred to as their *safe boundaries*. As it has already been observed that the safe boundaries of *solid hull edges* and hull vertices are easy to obtain, our problem now is reduced to designing an efficient algorithm for drawing the safe boundary of a *notch*.

4. SAFE BOUNDARY OF A NOTCH

During traversal along the boundary of the convex hull, the PEN_1 identifies a *notch* when it encounters a *false hull edge*. Consider such a *notch* attached to a *false hull edge* v_iv_{i+n} having $n + 1$ vertices, labeled by $v_i, v_{i+1}, \ldots, v_{i+n}$. Here we should emphasize that *we need to process inside the notch even if the length of the false hull edge* v_iv_{i+n} *is less than* 2δ. Surely, the safe boundaries of v_i and v_{i+n} will intersect here, which implies that

we do not need to traverse the notch. However, inside the notch there may be other components of the polygon whose safe boundaries may affect the safe boundaries of v_i and v_{i+n}. In Fig. 1, a situation similar to this is displayed inside the *notch* $\{v_{12}, \ldots, v_{16}\}$.

First of all, we split the *notch* into triangles by using the linear time algorithm due to Chazelle [1]. An edge of a triangle is termed the *triangulation edge* if it is generated due to triangulation, otherwise it will be referred to as a *polygonal edge*. Each triangle must have at least one triangulation edge. For an n vertex polygon, the triangulation generates $(n - 2)$ triangles introducing $(n - 3)$ triangulation edges. Now consider the graph with nodes corresponding to the triangles; an edge between a pair of nodes indicates that the triangles representing those two nodes share a triangulation edge. Thus each edge of the graph can be mapped to a unique triangulation edge shared by a pair of triangles, and each triangulation edge corresponds to a unique edge of the graph. It is easy to show that the graph mentioned above is a tree [7], referred to as the *triangulation tree*. We classify the triangles into three categories, type A, type B, and type C, depending on whether the number of its triangulation edge(s) is one, two, or three. The *root* node of the tree corresponds to the triangle adjacent to a *false hull edge*, and directions are assigned to the edges by traversing the tree in depth-first manner. It is easy to observe that the type-A triangles correspond to the leaf nodes of the tree, and each internal node may have one or two out-degree(s) depending on whether the corresponding triangle is type B or type C. From now onward, a triangle having vertices v_i, v_j, and v_k will be referred to as $\Delta v_i v_j v_k$. The triangulation edge of $\Delta v_i v_j v_k$, through which the control reaches this triangle during the forward traversal will be referred to as the *incoming edge* of $\Delta v_i v_j v_k$. Its other triangulation edges (if any) will be referred to as the *outgoing edges*.

While drawing safe boundary of a notch, nodes of the tree are processed in post-order. After traversing the subtrees of a node, when the node is processed the safe boundaries for all of the elements (vertices and *polygonal edges*, if any) of the corresponding triangle are drawn. The safe boundary of an element c_i (a vertex or a polygonal edge) will be denoted as $SB(c_i)$; it is attached with a *count* field which is initially set to zero. Next, we compare a selected set of already drawn *SB*s to check for possible intersection among them inside that triangle. This procedure is referred to as a *merge pass* and is explained in Subsection 4.4. When a pair of *SB*s are compared to check for a possible intersection its *count* field is incremented. If an intersection among two *SB*s takes place, the portions of the two *SB*s to the left of the point of intersection are deleted. In such a case, the survived portion (if any) of the safe boundary of an element c_i will be referred to as $SB(c_i)$. The *count* field of the surviving portion of

$SB(c_i)$ is inherited from the original one. Now note that the *SB*s that survive up to the processing of the current triangle may be intersected by the *SB*s, which will be generated while processing the parent or the other sibling of the current node. So, after processing the current node, we need to propagate a subset of the *SB*s to the triangle corresponding to the parent of the current node in the tree. Below we describe the concept of a visibility list that will aid the merge pass inside a triangle and the propagation of *SB*s from the current triangle to another triangle attached to the parent of the current node in the tree.

4.1. *Visibility List*

Let $\Delta v_i v_j v_k$ be the triangle currently being processed, whose incoming edge is $v_i v_k$. In order to detect for possible intersection among the *SB*s which were generated after processing the tree rooted at the current node (triangle) with some other *SB*s which are generated during the processing of the parent or another sibling of the current node, we introduce the concept of the *visibility list* as follows.

Consider a pair of lines l and l', both parallel to $v_i v_k$ and at a distance δ from it (see Fig. 2). Let l be on the same side of v_j with respect to $v_i v_k$ and l' be on the opposite side of v_j. The safe boundary of an element, say *SB**, which is generated while the subtree rooted at the node corresponding to $\Delta v_i v_j v_k$ is processed, may intersect another *SB*, say *SB***, belonging to the triangle attached to the parent of the current node, if *SB** spans above l. The reason for this is that *SB*** cannot penetrate inside $\Delta v_i v_j v_k$ beyond l due to the width constraint δ. Surely, *SB** also cannot span above the line l', from the interior of $\Delta v_i v_j v_k$.

DEFINITION 2. The *active zone* of a triangulation edge $v_i v_k$ is a connected region inside the polygon bounded by l and l' and contains $v_i v_k$ in its interior. The active zone of the triangulation edge $v_i v_k$ is shown in Fig. 2.

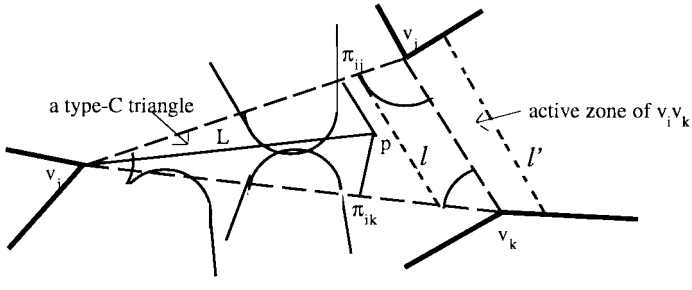


FIG. 2. Demonstration of merge pass.

A safe boundary, say SB^*, is said to be inside the active zone of a triangulation edge v_iv_k if it (or a part of it) spans inside that region. This can easily be tested by comparing SB^* with l or l' corresponding to that edge.

DEFINITION 3. The safe boundary of a vertex/edge c_i ($SB(c_i)$) is said to be *visible* to a *triangulation edge* (v_iv_k) if it spans the *active zone* of v_iv_k and if a line perpendicular to l', drawn from any point on it, cuts $SB(c_i)$ before cutting the safe boundary of some other vertex/edge or some edge of the polygon.

Each edge of the tree (i.e., the *triangulation edge*) is attached with a *visibility list* as defined below.

DEFINITION 4. The *visibility list V-LIST*(v_iv_k), attached to a *triangulation edge* v_iv_k, is a doubly connected link list containing a set of *SB*s which are drawn while processing the tree rooted at the current node and are *visible* to v_iv_k.

The *projection* of an element SB^* on a line is defined as the interval obtained by the foot of perpendiculars of the endpoints of SB^* on that line. For a pair of elements, say SB^* and SB^{**}, in the *V-LIST*(v_iv_k) their projections on v_iv_k are disjoint. The *SB*s in *V-LIST*(v_iv_k) are arranged in such a way that their projections are linearly ordered along v_iv_k.

It needs to be mentioned that for a *type-B* triangle, the *V-LIST* corresponding to its polygonal edge will consist of its *safe boundary*, which is a line segment parallel to that edge.

While processing a triangle $\Delta v_iv_jv_k$, it is assumed that the *V-LIST* of its outgoing edges v_iv_j and v_jv_k are already inherited from its successor(s) in the triangulation tree or are prepared by drawing the *SB*s of the elements (vertices/edges) of the triangle at the beginning of processing the triangle. These *V-LIST*s may contain the *SB*s that may intersect inside the current triangle. Now a merge pass among the elements of these two lists is performed to detect possible intersections.

It is now time to explicitly mention the data structures that need to be maintained during the execution of the algorithm. This will also help us to present our algorithm more clearly.

4.2. *Data Structure*

While processing a *notch*, the algorithm maintains the following data structures.

poly_chain — An array of vertices and edges of the notch stored in clockwise order.

$\mathscr{T}$ The triangulation tree of the notch. With each edge of this tree a *V-LIST* is maintained, as described in the previous subsection.

$\mathscr{L}$ A list containing *SB* for the elements (vertices and polygonal edges) inside a *notch*. Each element of $\mathscr{L}$ has a pointer to its neighboring *SB* in clockwise order. This will be recursively constructed while traversing the tree. At the time of processing a triangle, the *SB* of its edge(s) and vertices inside the triangle is created in $\mathscr{L}$ if it is not already present. During a merge pass, if an intersection between two *SB*s (say *SB** and *SB***) is detected, these *SB*s are updated by removing the portions to the left of the point of the intersection. An appropriate pointer is also established among *SB** and *SB***. Finally, after processing the root node, the list $\mathscr{L}$ gives the safe boundary of the *notch*.

4.3. *Processing of Triangles*

While processing a *type-A* triangle $\Delta v_i v_{i+1} v_{i+2}$, if the length of the triangulation edge $v_i v_{i+2}$ is greater than 2δ, the *SB*s for both of its *polygonal edges* $v_i v_{i+1}$ and $v_{i+1} v_{i+2}$ are drawn. Note that for a *type-A* triangle the merge pass essentially means finding the intersection of $SB(v_i v_{i+1})$ and $SB(v_{i+1} v_{i+2})$. We delete the portions of $SB(v_i v_{i+1})$ and $SB(v_{i+1} v_{i+2})$ to the left of the point of intersection. Next, we compute *SB*s of the two vertices v_i and v_{i+2}. Finally, the *SB*s of v_i, v_{i+2}, and the survived portions of $SB(v_i v_{i+1})$ and $SB(v_{i+1} v_{i+2})$ are connected using their bidirectional links and inserted in the $\mathscr{L}$ list. They are also inserted in the *V-LIST*$(v_i v_{i+2})$. If the length of the triangulation edge $v_i v_{i+2}$ is less than 2δ, $SB(v_i)$ and $SB(v_{i+2})$ will intersect and we need not compute $SB(v_i v_{i+1})$ and $SB(v_{i+1} v_{i+2})$.

While processing a node corresponding to a *type-B* triangle $\Delta v_i v_j v_{j+1}$, *SB*s corresponding to its *polygonal edge* $v_j v_{j+1}$ and the vertex v_{j+1} are created in the $\mathscr{L}$ list. Note that here $v_i v_j$ is the outgoing edge of $\Delta v_i v_j v_{j+1}$. The control reaches this triangle during the backtrack of the post-order traversal through $v_i v_j$. So, $SB(v_i)$ and $SB(v_j)$ are already drawn while processing the triangle attached to its only child. The pointers in $\mathscr{L}$ are established among (i) $SB(v_j)$ and $SB(v_j v_{j+1})$ and (ii) $SB(v_j v_{j+1})$ and $SB(v_{j+1})$. *V-LIST*$(v_j v_{j+1})$ is created with $SB(v_j v_{j+1})$ and $SB(v_{j+1})$. Next, a merge pass needs to be executed among the *SB*s in the *V-LIST*$(v_i v_j)$ and the *V-LIST*$(v_j v_{j+1})$. Finally, the *V-LIST* of the incoming *triangulation edge* $v_i v_{j+1}$ is prepared. It consists of the members of *V-LIST*$(v_i v_j)$ and *V-*

$LIST(v_jv_{j+1})$ which lie inside the active zone of v_iv_{j+1}. The control is then transferred to the parent of this node in $\mathcal{T}$.

Note that during the postorder processing of $\mathcal{T}$, when a *type-C* triangle $\Delta v_iv_jv_k$, $i < j < k$, is processed, both the children of the node corresponding to $\Delta v_iv_jv_k$ are already processed. Thus, $SB(v_i)$, $SB(v_j)$, and $SB(v_k)$ are already computed while the triangles attached to both of its children is processed; $V\text{-}LIST(v_jv_i)$ and $V\text{-}LIST(v_jv_k)$ are inherited from the successors of the current triangle in $\mathcal{T}$. So, here only a very careful merge pass needs to be executed among the $V\text{-}LIST(v_iv_j)$ and $V\text{-}LIST(v_jv_k)$ to detect any intersection(s) if they are present. Finally, the $V\text{-}LIST(v_iv_k)$ for its incoming edge v_iv_k is created before control is transferred to its parent in $\mathcal{T}$.

Thus we find that processing a triangle, apart from generating *SB*s of the polygonal edges and vertices present in that triangle, also involves efficient management of the *V-LIST*s, i.e., the merging of two *V-LIST*s corresponding to its two outgoing edges and creation of a new *V-LIST* corresponding to the incoming edge of that triangle. In the next two sections we shall elaborate on these two techniques.

4.4. *Merging a Pair of V-LISTs*

Let $\Delta v_iv_jv_k$ be a triangle under process whose incoming edge is v_iv_k and whose two outgoing edges are v_jv_i and v_jv_k (see Fig. 2). In order to detect the intersection(s) among the *SB*s present in $V\text{-}LIST(v_jv_i)$ and $V\text{-}LIST(v_jv_k)$, they are merged from their end corresponding to v_j with the help of two pointers which indicate the current elements of the respective lists. We fix a point π_{ji} on v_jv_i such that, during the merge pass inside $\Delta v_iv_jv_k$, a $SB \in V\text{-}LIST(v_jv_i)$ can never intersect with any *SB* of $V\text{-}LIST(v_jv_k)$ if the projection of the former one on v_jv_i lies completely outside the line segment $v_j\pi_{ji}$.

DEFINITION. During the processing of $\Delta v_iv_jv_k$, a $SB \in V\text{-}LIST(v_jv_i)$ is said to be *favorable* for merge pass if the projection of at least one end point on v_jv_i lies on the line segment $v_j\pi_{ji}$.

Similarly, we can locate a point π_{jk} on v_jv_k and identify a set of *SB* in $V\text{-}LIST(v_jv_k)$ which are favorable for merge pass inside $\Delta v_iv_jv_k$.

While processing $\Delta v_iv_jv_k$, our merge pass progresses along $V\text{-}LIST(v_jv_i)$ and $V\text{-}LIST(v_jv_k)$ starting from their ends corresponding to v_j. This merge pass terminates as soon as it finds a *SB* in either $V\text{-}LIST(v_jv_i)$ or $V\text{-}LIST(v_jv_k)$ which is not *favorable* for a merge pass inside $\Delta v_iv_jv_k$. The choice of π_{ji} and π_{jk} follows from the following observation:

Observation 1. Consider the bisector L of the $\angle v_iv_jv_k$ and choose a point p on it such that the length of the perpendiculars on v_jv_i and v_jv_k

from p is equal to δ. Let the foot of the perpendiculars of p on v_jv_i and on v_jv_k be α and β, respectively. Now the following cases arise.

Case 1. Both of α and β fall on the closed segment v_jv_i and v_jv_k respectively. In this case, a *SB* of *V-LIST*(v_jv_i) (*V-LIST*(v_jv_k)) becomes favorable if the projection (foot of perpendicular) of at least one end point on v_jv_i (v_jv_k) is closer to v_j than α (β) (see Fig. 3a). Thus, in this case, $\pi_{ji} = \alpha$ and $\pi_{jk} = \beta$.

Case 2. Both α and β fall outside the closed segments v_jv_i and v_jv_k, respectively, and $\Delta v_iv_jv_k$ is an acute angle triangle. Here $SB(v_i)$ and $SB(v_k)$ will intersect. But there may exist some *SB* who can intersect both of $SB(v_i)$ and $SB(v_k)$. So, a merge pass inside this triangle is needed, and $\pi_{ji} = v_i$ and $\pi_{jk} = v_k$ in this case (see Fig. 3b).

Case 3. α lies inside the closed segment v_jv_i but β is outside the closed segment v_jv_k. Here $SB(v_k)$ must participate in the merge pass. Now consider a line l perpendicular to v_iv_k which touches (not intersects) $SB(v_k)$ inside $\Delta v_iv_jv_k$. Let it meet v_jv_i at a point γ. Note that, as the vertex v_k lies on one side of the line v_jv_i, and the centers of all the *SB*s in *V-LIST*(v_jv_i) lie on the other side of v_jv_i, if any one of them intersects $SB(v_k)$ then its projection on v_jv_i must overlap the line segment $v_j\gamma$ (as shown in Fig. 3c). So, in this case a $SB \in$ *V-LIST*(v_jv_i) remains favorable if its projection on v_jv_i overlaps the line segment $v_j\gamma$. Thus, we have $\pi_{ji} = \gamma$ and $\pi_{jk} = v_k$ in this case. A similar case arises if α lies outside the closed segment v_jv_i but β is inside the closed segment v_jv_k.

Case 4. Both α and β fall outside the closed segments v_jv_i and v_jv_k respectively, and $\Delta v_iv_jv_k$ is an obtuse angle triangle whose $\angle v_jv_kv_i > 90°$. As $SB(v_k)$ must participate in the merge pass, here $\pi_{jk} = v_k$ and π_{ji} is chosen in a manner similar to that used in the earlier two cases.

During the merge inside $\Delta v_iv_jv_k$ let $SB^* \in$ *V-LIST*(v_jv_i) be a favorable candidate. First of all, we test whether SB^* penetrates the active zone of v_jv_k or not. In the case of a negative answer, we skip SB^* and consider the

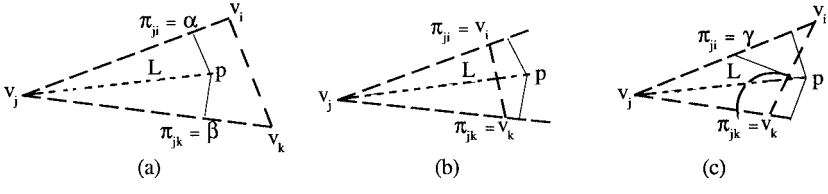


FIG. 3. Selection of the members of two V-LISTs corresponding to the outgoing edges of a triangle which are favorable for the merge pass inside the triangle.

next element of *V-LIST*(v_jv_i). But in the case of an affirmative answer, we need to check *SB** with the members of *V-LIST*(v_jv_k) for possible intersection. We draw perpendiculars from the end points of *SB** on L which hit L at the points a_1 and a_2. Now, the favorable members of *V-LIST*(v_jv_k), whose projections on L overlap a_1a_2 will be considered one by one to detect for possible intersection(s) with *SB**, if any. The cost of each comparison, excepting the last one, is charged by incrementing the *count* field of the participating element of *V-LIST*(v_jv_k). For the last comparison, we charge its cost to *SB**. The merge pass then proceeds considering the next element of *V-LIST*(v_jv_i). As soon as a member in either *V-LIST*(v_jv_i) or *V-LIST*(v_jv_k) is reached which is not *favorable* for a merge pass inside $\Delta v_iv_jv_k$ the merge pass terminates.

During the merge pass inside a triangle, the intersection (if any) which is observed most recently is preserved. Let the participating members be *SB** $\in$ *V-LIST*(v_jv_i) and *SB*** $\in$ *V-LIST*(v_jv_k) at the end of the merge pass. We update *SB** and *SB*** by deleting the portions to the left of the point of intersection and we establish an appropriate pointer among *SB** and *SB*** in the $\mathscr{L}$ list.

The processing of the current triangle ends by decrementing the *count* field of one of the last two compared *SB*s whose *count* field was incremented in the last comparison during the merge pass inside the current triangle. Conceptually, the cost of the last comparison is charged to the triangle itself. As the merge pass inside a triangle is performed at most once, such a charging to a triangle may also be done at most once during the entire execution process of the *notch*.

LEMMA 1. *The merge pass inside a triangle requires time linear to the number of elements in the V-LISTs of both of its outgoing edges.*

Proof. While processing a triangle $\Delta v_iv_jv_k$ during the backtrack of the post-order traversal in $\mathscr{T}$, let v_jv_i and v_jv_k be the two outgoing edges whose *V-LIST*s need to be merged. The proof of the lemma follows from the fact that the projections of the endpoints of the *SB*s belonging to the *V-LIST* of the edge v_jv_i (v_jv_k) are linearly ordered on the line L bisecting $\angle v_iv_jv_k$. ∎

LEMMA 2. *If* $\angle v_iv_jv_k > 90°$, *members of V-LIST*(v_jv_i) *and of V-LIST*(v_jv_k) *will not intersect.*

Proof. Let us draw the projections of the point $p \in L$ on v_jv_i and v_jv_k (as defined in the first paragraph of this section) which touch the respective lines at α and β respectively. As $\angle pv_jv_i$ ($\angle pv_jv_k$) $\geq 45°$, both $v_j\alpha$ and $v_j\beta$ will be less than δ. So all the *SB*s of *V-LIST*(v_jv_i) (*V-LIST*(v_jv_k)) spanned over α (β) have been covered by *SB*(v_j) and are not present in the respective *V-LISTs*. Again, as π_{ji} and π_{jk} are determined by α and β

respectively in this case (see Fig. 4), the merge pass need not be executed in such a triangle. Hence the result follows. ▮

The above lemma says that if $\angle v_i v_j v_k > 90°$, a merge pass of *V-LIST*$(v_j v_i)$ and *V-LIST*$(v_j v_k)$ need not be performed. In the next subsection we explain the creation of the *V-LIST* for the incoming edge of a triangle as the last step of the processing of a triangle.

4.5. *Creation of New V-LIST*

After the completion of the merge pass inside a triangle $\Delta v_i v_j v_k$, the *V-LIST* of its incoming edge $v_i v_k$ is created by the selected members of *V-LIST*$(v_j v_i)$ and *V-LIST*$(v_j v_k)$. We need to consider the two distinct cases which depend on whether $\Delta v_i v_j v_k$ is an obtuse angle triangle or an acute angle triangle.

Observation 2. If $\Delta v_i v_j v_k$ is an obtuse angle triangle, then one of the following situations holds.

(a) $\angle v_i v_j v_k > 90°$. Here no merge pass is needed inside $\Delta v_i v_j v_k$ (by Lemma 2). The *V-LIST* of the incoming edge $v_i v_k$ is obtained by concatenating *V-LIST*$(v_j v_i)$ and *V-LIST*$(v_j v_k)$.

(b) $\angle v_j v_k v_i > 90°$. By Lemma 2, here the members of *V-LIST*$(v_j v_k)$ will not intersect with the *SB*s generated inside the triangle attached to the other sibling or the parent of the current node. Thus no element of *V-LIST*$(v_j v_k)$ need be propagated to *V-LIST*$(v_i v_k)$. Only a few members in *V-LIST*$(v_j v_i)$, which extends inside the active zone of $v_i v_k$, will form *V-LIST*$(v_i v_k)$. These sets of *SB*s are obtained as follows.

Consider a pair of elements *SB** and *SB*** $\in$ *V-LIST*$(v_j v_i)$ which is favorable with respect to the merge pass inside $\Delta v_i v_j v_k$. Let *SB** appear after *SB*** during the above merge pass. The question of the propagation of *SB*** to *V-LIST*$(v_i v_k)$ arises if *SB*** is inside the active zone of $v_i v_k$. Surely, *SB** will also be inside the active zone of $(v_i v_k)$ in this case and will be compared with *SB*(v_k). Now, if *SB** intersects *SB*(v_k), then *SB*** will immediately be deleted from the $\mathscr{L}$ list. Otherwise, by Case 3 of Observa-

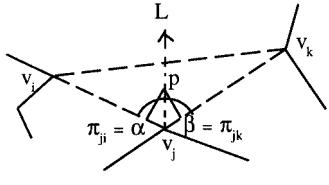


FIG. 4. Proof of Lemma 2.

tion 1, as π_{ji} is the point of intersection of v_jv_i and a tangent of $SB(v_k)$ which is perpendicular on v_iv_k, the orthogonal visibility of SB^{**} from v_iv_k is lost (see Fig. 3c). So, SB^{**} cannot belong to $V\text{-}LIST(v_iv_k)$. Thus, during the merge pass inside $\Delta v_iv_jv_k$, we need to identify $SB^* \in V\text{-}LIST(v_jv_i)$ which is compared last with $SB(v_k)$. $V\text{-}LIST(v_iv_k)$ will be formed with SB^* and all the SBs in $V\text{-}LIST(v_jv_i)$ which are not considered during this merge pass.

(c) $\angle v_jv_iv_k > 90°$. This situation is similar to Case (b), stated above.

Observation 3. If $\Delta v_iv_jv_k$ is an acute angle triangle, then as $\angle v_iv_jv_k < 90°$, the merge pass has already been performed inside $\Delta v_iv_jv_k$ and $SB' \in V\text{-}LIST(v_jv_i)$ has participated in the last comparison in the merge pass inside $\Delta v_iv_jv_k$. Now,

- If SB' is outside the active zone of v_iv_k, then surely all of the elements which are compared during the merge pass inside $\Delta v_iv_jv_k$ need not be propagated to $V\text{-}LIST(v_iv_k)$.
- In contrast, if SB' is inside the active zone of v_iv_k then there exists some element(s) of $V\text{-}LIST(v_jv_i)$ which may have been considered in the merge pass inside $\Delta v_iv_jv_k$ and may need to be propagated to $V\text{-}LIST(v_iv_k)$.

As mentioned earlier, during the progress of the merge pass inside a triangle $\Delta v_iv_jv_k$, two pointers are maintained corresponding to the $V\text{-}LIST$s of its two outgoing edges v_jv_i and v_jv_k. At the end of the merge pass each of them will contain the address of an element of the respective $V\text{-}LIST$ which has been compared last during the current merge pass. Let $SB' \in V\text{-}LIST(v_jv_i)$ and $SB'' \in V\text{-}LIST(v_jv_k)$ be the above two elements. The $V\text{-}LIST$ of the incoming edge v_iv_k will be constructed by following the steps described below.

- If projections of both the endpoints of SB' are outside the active zone of v_iv_k, the pointer along $V\text{-}LIST(v_jv_i)$ advances to get its first member, say SB^*, which lies inside the active zone of v_iv_k.
- If SB' is the only member of $V\text{-}LIST(v_jv_i)$ which lies inside the active zone of v_iv_k, then the first member which lies in the active zone of v_iv_k is SB' itself.
- If there exist some other elements in $V\text{-}LIST(v_jv_i)$ in addition to SB', the pointer along $V\text{-}LIST(v_jv_i)$ needs to backtrack to get its first member SB^* lying inside the active zone of v_iv_k.
- Any one of the aforesaid three situations may appear regarding the selection of SB^{**}, the first member of $V\text{-}LIST(v_jv_k)$, which lies inside the active zone of v_iv_k.

• Finally, *SB** and *SB*** are connected using bidirectional pointers. Now, the *V-LIST* of the incoming edge v_iv_k is a list of *SB*s whose two terminal members are the last element of the *V-LIST*(v_jv_i) toward v_i and the last element of the *V-LIST*(v_jv_k) toward v_k respectively.

Now note that the elements of *V-LIST*(v_jv_i) (*V-LIST*(v_jv_k)) which are encountered during the backtrack have been considered during the current merge pass and their *count* fields have been incremented. But the *count* field of *SB′* (*SB″*) remains unchanged as it was considered last during the current merge pass.

From earlier discussions we note that while processing a node corresponding to a *type-B* or a *type-C* triangle $\Delta v_iv_jv_k$, there may exist some situation when some *SB* of *V-LIST*(v_jv_i) may be considered in the merge passes in both of $\Delta v_iv_jv_k$ and in the triangle attached to the parent of the current node. This gives an impression that after the generation of a *SB* it may participate in the merge pass of different triangles arising along a path of the triangulation tree, which indicates an $O(n^2)$ time complexity of our algorithm. A trivial solution is reducing the depth of the tree. Using geodesic triangulation proposed in [4], the depth can be reduced to $O(\log N)$, and it leads to an $O(N \log N)$ time algorithm. So, we shall not proceed toward the complication of geodesic triangulation. Rather we shall show that a *SB*, after its generation, may participate in the merge pass of at most two triangles during a backtrack along the triangulation tree.

THEOREM 1. *After the generation of a SB, its count field may be incremented to at most* 2 *during its propagation while backtracking along a path during the post-order traversal in* $\mathcal{T}$.

Proof. In order to prove this result, we have to study two triangles along a path of $\mathcal{T}$. We assume that currently the control is in the successor triangle and that the merge pass has already been performed inside the current triangle. Our aim is to prove that a *SB* whose *count* field is incremented during the current merge pass will participate in at most one more merge pass inside some other predecessor triangle incrementing its *count* field. For the sake of simplicity in the proof, we assume that the two triangles under consideration share a common triangulation edge.

Let $\Delta v_iv_jv_k$ be the triangle corresponding to the current node under consideration and let $\Delta v_iv_kv_l$ be the triangle attached to the parent of the current node. If $\angle v_iv_jv_k > 90°$, then by Lemma 2 the merge pass need not be performed inside $\Delta v_iv_jv_k$. So, we assume $\angle v_iv_jv_k < 90°$. Surely, at least one of the other three angles of the quadrilateral, formed by concatenating

$\Delta v_i v_j v_k$ and $\Delta v_i v_k v_l$, must be greater than 90°. This gives birth to three different cases as follows:

Case 1. $\angle v_j v_i v_l > 90°$. Here, by Lemma 2, the members of *V-LIST*$(v_j v_i)$ will not be able to enter in the active zone of the edge $v_i v_l$. Thus the members of *V-LIST*$(v_j v_i)$ will either not be compared with the members of *V-LIST*$(v_i v_l)$ or they will not be propagated to *V-LIST*$(v_i v_l)$, depending on whether $v_k v_l$ or $v_i v_l$ is the incoming edge of $\Delta v_i v_k v_l$. Thus in either case their *count* fields will remain unchanged inside $\Delta v_i v_k v_l$.

Case 2. $\angle v_i v_l v_k > 90°$. By case (b) of Observation 2, the two subsets of members of *V-LIST*$(v_i v_k)$, which will participate in the merge pass inside $\Delta v_i v_k v_l$ and which need to be propagated to the *V-LIST* of the incoming edge of $\Delta v_i v_k v_l$, may have at most one element in common, whose *count* field will not be incremented during the merge pass inside $\Delta v_i v_k v_l$. Thus if there exists any member(s) of *V-LIST*$(v_j v_i)$ which also need(s) to be propagated to the *V-LIST* of the incoming edge of $\Delta v_i v_k v_l$, its *count* field will be unchanged (not incremented) during the merge process of $\Delta v_i v_k v_l$.

Case 3. $\angle v_l v_k v_j > 90°$. Note that $\angle v_l v_k v_j = \angle v_i v_k v_j + \angle v_i v_k v_l$. To analyze this case, we need to consider the following three situations separately.

Case 3.1. $\angle v_i v_k v_l \geq 90°$. If $v_i v_l$ is the incoming edge, the elements of *V-LIST*$(v_i v_k)$ will not participate in the merge pass with *V-LIST*$(v_k v_l)$ (by Lemma 2). So, the elements of *V-LIST*$(v_i v_k)$ which are inherited from *V-LIST*$(v_j v_i)$ and need to be propagated to *V-LIST*$(v_i v_l)$ will have their *count* field unchanged inside $\Delta v_i v_k v_l$.

If $v_k v_l$ is the incoming edge, the elements of *V-LIST*$(v_i v_k)$ will not be propagated to *V-LIST*$(v_k v_l)$.

Case 3.2. $\angle v_j v_k v_i \geq 90°$. Here the elements of *V-LIST*$(v_i v_k)$ which have been propagated from *V-LIST*$(v_j v_i)$ will have their *count* field unchanged inside $\Delta v_i v_j v_k$ (by Case (b) of Observation 2). Among them, if some one participates in the merge pass inside $\Delta v_i v_k v_l$, then its *count* field may be incremented to at most 2. In Case 3.3 we show that the *SB*s of $v_j v_i$ whose *count* field has been incremented during the merge pass inside two adjacent triangles, say $\Delta v_i v_j v_k$ and $\Delta v_i v_k v_l$, will not be propagated further. The same result is true if the *count* field of a *SB* is incremented during the merge pass of two triangles, which are not necessarily adjacent.

Case 3.3. Both $\angle v_j v_k v_i$ and $\angle v_i v_k v_l$ are less than 90°. As we are discussing the nature of the propagation of the *SB*s of *V-LIST*$(v_j v_i)$, we assume that $\angle v_j v_i v_k < 90°$.

Case 3.3.1. The incoming edge of $\Delta v_i v_k v_l$ is $v_k v_l$. In order to analyze this case, let us assume that the triangulation edge $v_i v_k$ is replaced by $v_j v_l$

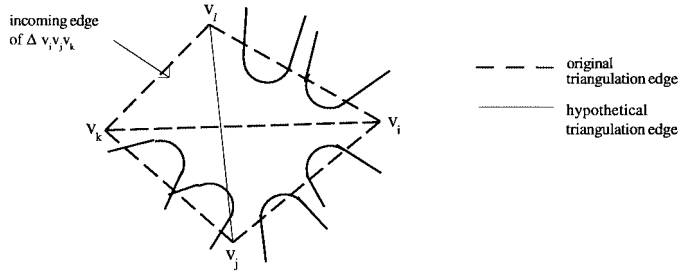


FIG. 5. Demonstration of Case 3.3.1 in the Proof of Theorem 1.

in the triangulation of the current notch (see Fig. 5). Its adjacent triangles are $\Delta v_j v_i v_l$ and $\Delta v_j v_k v_l$, where the former one is the predecessor and the latter is its successor in the triangulation tree.

Now, consider the processing of the triangle $\Delta v_j v_i v_l$. After the merge pass among *V-LIST*$(v_j v_i)$ and *V-LIST*$(v_i v_l)$ inside $\Delta v_j v_i v_l$, the elements in *V-LIST*$(v_j v_l)$ will be linearly ordered on $v_j v_l$. Now arguing in a manner similar to that in Case 2 of this proof, the elements in *V-LIST*$(v_j v_l)$ can be split into two subsets. One of these subsets will participate in the merge pass with *V-LIST*$(v_j v_k)$ and the other subset will be propagated to *V-LIST*$(v_k v_l)$. Among these two subsets we may have at most one element in common, whose *count* field will not be incremented during the merge pass of the triangle $\Delta v_j v_k v_l$. This leads to the conclusion that there may exist at most one element of *V-LIST*$(v_j v_i)$ which will be considered in the merge passes of both of the *actual* triangles $\Delta v_i v_j v_k$ and $\Delta v_i v_k v_l$.

Case 3.3.2. The incoming edge of $\Delta v_i v_k v_l$ is $v_i v_l$. Let $SB^* \in$ *V-LIST*$(v_j v_i)$ be compared last during the merge pass inside $\Delta v_i v_j v_k$. It may have been compared with $SB(v_k)$ or some element preceding $SB(v_k) \in$ *V-LIST*$(v_j v_k)$. The merge pass inside $\Delta v_i v_k v_l$ starts with $SB(v_k)$ as the first element of *V-LIST*$(v_k v_l)$. So, the subset of members (if any) of *V-LIST*$(v_j v_i)$ which has been propagated to *V-LIST*$(v_i v_k)$ and the subset of *V-LIST*$(v_j v_i)$ that has participated in the merge pass inside $\Delta v_i v_j v_k$ may have at most one element in common, which is SB^*. The reason is that since $\angle v_j v_k v_l > 90°$, at most one element of *V-LIST*$(v_j v_i)$ can enter in the active zone of both $v_j v_k$ and $v_k v_l$. Also note that the *count* field of SB^* has not been incremented during the merge pass inside $\Delta v_i v_j v_k$. ∎

Thus, Theorem 1 suggests that if the *count* field of an *SB* is set to 2 while processing a triangle, it will not appear in the *V-LIST* of the incoming edge of that triangle, which implies that it will not propagate further toward the root.

Based on the observations stated above regarding the merging and creation of visibility lists, we now present the stepwise description of our algorithm and its complexity analysis in the next section.

5. ALGORITHM AND COMPLEXITY

5.1. *Algorithm*

ALGORITHM SAFE_BOUNDARY.

Step 1. Compute the convex hull of the given polygon.

Step 2. For each notch do:

Step 2.1. Triangulate the notch to prepare the triangulation tree $\mathscr{T}$.

Step 2.2. (*Traverse the tree in post-order manner *).

Call Post_Order(root).

(* This procedure recursively traverses the tree $\mathscr{T}$. At each node after the traversal of both its children, it calls the procedure **Process_Triangle** which is thematically described in detail in Section 4. *)

Step 3. Draw the safe boundaries for the *solid hull edges* and for the *hull vertices* of the convex hull.

Step 4. Connect the *SB* of each *solid hull edge* to the *SB* of its attached hull vertices.

Step 5. For each notch do

Merge each end of the $\mathscr{L}$ list of that notch to the *SB* of the hull vertex attached to its *false hull edge*.

Step 6. Traverse the $\mathscr{L}$ list from any element to output the safe boundary of the polygon.

Stop.

5.2. *Proof of Correctness*

Before analyzing the complexity of our algorithm, we need to prove its correctness. For a convex polygon the method described is very simple and its correctness is obvious. The following theorem summarizes the justification of correctness of our algorithm for drawing the safe boundary inside a notch.

THEOREM 2. *Our proposed method of obtaining the safe boundary inside a notch is correct*.

Proof. The correctness of the algorithm follows from (i) the fact that the generation of the *SB* for all the edges and vertices of the notch is done and (ii) the following recursive argument.

• During the backtrack of the post-order traversal of the triangulation tree $\mathcal{T}$ of the notch, when a triangle corresponding to a nonleaf node is reached, let the safe boundary of the portion(s) of the *notch* indicated by the children of that node be correct. Now when the triangle corresponding to that node will be processed, first of all, the *SB*s of the vertices/edges of the current triangle are added to the $\mathcal{L}$ list and also to the respective *V-LIST*s. Then the merge pass is initiated which detects whether any pair of *SB*s intersect inside the triangle. If intersection(s) takes place, the latest one (i.e., the one closest to the incoming edge of the triangle) is considered and the portions to the left of the point of intersection are deleted. Thus the *notch* rooted at the current triangle is correctly processed.

• The *SB*s whose *count* field is less than 2 and which belong to the *active zone* of the incoming edge of the triangle are propagated to its predecessor triangle through *V-LIST* to check for possible intersections with the *SB*s generated inside its other sibling and its predecessor triangle. But, as suggested in Theorem 1, a *SB* having a value 2 in its *count* field need not be propagated further as it will not intersect further with any *SB* during the backtrack along the path from that node up to the root of $\mathcal{T}$. ▮

5.3. *Complexity Analysis*

The convex hull of an N vertex simple polygon can be drawn in $O(N)$ time using the algorithm proposed in [9]. Let it give birth to K *notches* where the ith *notch* is assumed to have N_i vertices. Below we describe a few results related to the analysis of the time complexity of processing the *notches*.

LEMMA 3. *Time required for drawing the safe boundary of a notch of N_i vertices is $O(N_i)$.*

Proof. The triangulation of a *notch* of N_i vertices can be done in $O(N_i)$ time [1]. It gives birth to $(N_i - 2)$ triangles. The total number of vertices and polygonal edges considering all the triangles are $3(N_i - 2)$ and $(N_i - 1)$ respectively. So, the time required for the drawing of safe boundaries of $O(N_i)$ vertices and edges of the *notch* is $O(N_i)$. By Lemma 1, the time required to conduct the merge pass in each triangle is linear in the number of *SB*s attached to *V-LIST*s of its outgoing edges. But a $SB(c_j)$ corresponding to an element c_j of the notch may participate in the merge pass of more than one triangle. Each time it is considered, either its *count* field is incremented or the triangle inside which the merge pass is

conducted is charged. Theorem 1 shows that for any *SB* generated inside a notch, its *count* field may be incremented to at most 2. Moreover, a triangle is charged at most once, provided a merge pass takes place inside it. Thus, apart from generation of *SB*s, the total time required for the merge passes of all the triangles in the notch is also $O(N_i)$. ∎

LEMMA 4. *At any point of time, the total space occupied by the V-LISTs of all the triangulation edges is* $O(N)$.

Proof. Note that the *V-LIST* of a *triangulation edge* corresponding to a triangle is generated when the control backtracks through that edge. It consists of some of the *SB*s of the *V-LIST*s of its outgoing edges. After generation of the new *V-LIST*, corresponding to the incoming edge of a triangle, those of its outgoing edges are deleted. Thus at any instant of time the sets of *SB*s present in the *V-LIST*s of different *triangulation edges* are disjoint. This proves the lemma. ∎

Lemmas 3 and 4 lead to the fact that the time required for the generation of the *safe boundaries* of each notch is linear in the number of its vertices. Again, since the notches around the convex hull of the polygon are disjoint, the total time complexity for drawing safe boundaries of all the notches is also linear in the total number of vertices of all the notches.

The time required to draw the safe boundaries of all *solid hull edges* and the hull vertices may be $O(N)$ in the worst case. Next, we may have to concatenate the safe boundary of a notch to the safe boundaries of some other notch or some *solid hull edge* which are attached to the two vertices of its corresponding *false hull edge*. This requires $O(K)$ time, where K is the number of notches. A further pass is required for reporting the safe boundary of the polygon by traversing along the $\mathscr{L}$ list. This requires time linear to the number of *SB*s describing the safe boundary of the polygon. Thus we have the following theorem which states the complexity of our proposed algorithm.

THEOREM 3. *The time and space complexities of our proposed algorithm are both* $O(N)$, *where N is the total number of vertices of the polygon.*

6. MINKOWSKI SUM OF A SIMPLE POLYGON AND A RECTANGLE

In this section we show that our SAFE_BOUNDARY algorithm can easily be modified to compute the Minkowski sum of a simple polygon P and a rectangle Q. Needless to say, the boundary of the Minkowski sum of the polygon P and the rectangle Q, denoted by $P \oplus Q$, can be obtained by translating the reference point of Q along the boundary of P (see Fig. 6).

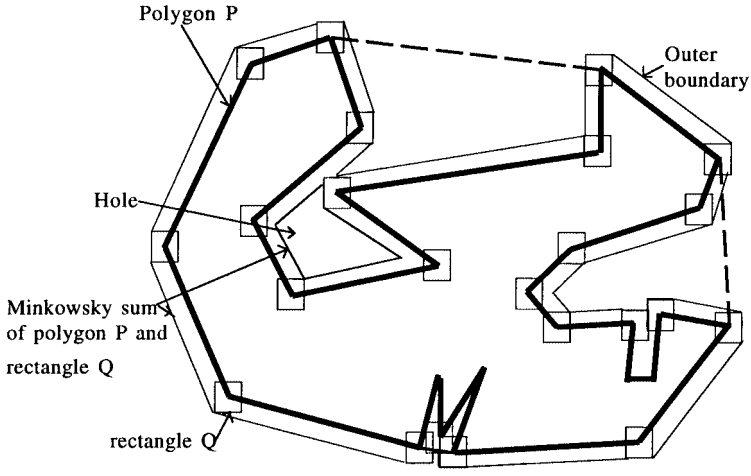


FIG. 6. Minkowski sum of a simple polygon and a rectangle.

This will help to show that our technique solves the problem of finding the Minkowski sum of an arbitrary simple polygon and a convex polygon in $O(MN)$ time, where M and N denote the number of vertices of both the polygons respectively. Note that the worst case size of the output for the above problem may also be $O(MN)$. As in the earlier problem, we detect the notches by finding the convex hull of the simple polygon. These notches are then processed one by one. The definitions and notations will remain the same as in the earlier sections unless otherwise specified.

6.1. *Recapitulation of Necessary Definitions*

The boundary of the Minkowski sum of a *notch* and a rectangle Q is formed with a set of disjoint chains of line segments. These line segments will form *components of the Minkowski sum* ($CMS(c_i) = c_i \oplus Q$) for different elements c_i (vertices and edges) of P. Needless to say, $CMS(c_i)$ serves the role of $SB(c_i)$ if the rectangle is replaced by the circle C as stated in the earlier sections. The list $\mathscr{L}$, containing the *CMS*s is constructed recursively and is maintained as a doubly connected link list. During the execution of a notch, if a closed cycle in $\mathscr{L}$ is observed it indicates a hole inside the area defined by the outer boundary of the Minkowski sum, and it is reported immediately. At the end of processing a notch, $\mathscr{L}$ contains a chain of *CMS*s which describes the outer boundary of the Minkowski sum of the notch. Finally, the $\mathscr{L}$ lists of different notches are concatenated with the *CMS*s of solid hull edges and hull vertices to get the outer

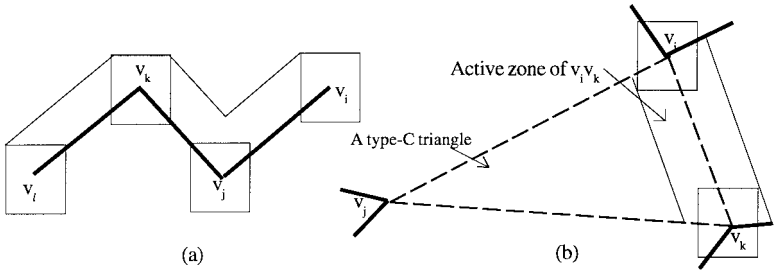


FIG. 7. Demonstration of (a) Minkowski sum of edges and vertices inside a notch, and (b) the active zone of a triangulation edge.

boundary of the Minkowski sum of P. Below we mention the nature of the $CMS(c_i)$ for different components c_i of the polygon (see Fig. 7a for an illustration).

If c_i is an edge of P, the exposed part of $(CMS(c_i))$ outside the polygon P will be a straight line segment parallel to c_i, which is the common tangent of two rectangles, obtained by placing Q at the two end vertices of c_i.

If c_i is a concave vertex of P, the portion of $CMS(c_i)$ outside P will be subsumed by the Minkowski sums of the edges adjacent to that vertex. So, for these vertices the $CMS(c_i)$ need not be computed.

If c_i is a convex vertex of P, a portion of $CMS(c_i)$ must be exposed as the boundary of $P \oplus Q$. It may be a portion of an edge of $c_i \oplus Q$ or portions of two adjacent edges along with the corresponding corner.

We process the notch by traversing the triangulation tree $\mathcal{T}$ in post order. Here also, the processing of a triangle corresponding to a node in $\mathcal{T}$ involves a merge pass and a propagation of the relevant *CMS*s from one triangle to its predecessor in the triangulation tree $\mathcal{T}$. But these two key operations greatly depend on the concept of the *active zone* of a triangulation edge, which needs to be appropriately modified to suit our present purpose. Our Definition 2 of the *active zone* of a triangulation edge remains valid with the following changed definition of the pair of parallel lines l and l' describing the active zone.

Consider a triangle $\Delta v_i v_j v_k$ whose $v_i v_k$ is a triangulation edge. Let Q^i and Q^k be the placement of the rectangle Q at v_i and v_k respectively. The translation of Q on $v_i v_k$ generates two common non-intersecting tangents, Q^i and Q^k. They are parallel to $v_i v_k$; one will pass through $\Delta v_i v_j v_k$ and the other will belong outside $\Delta v_i v_j v_k$. These two tangents are considered as l and l' respectively for defining the *active zone* of $v_i v_k$ (see Fig. 7b).

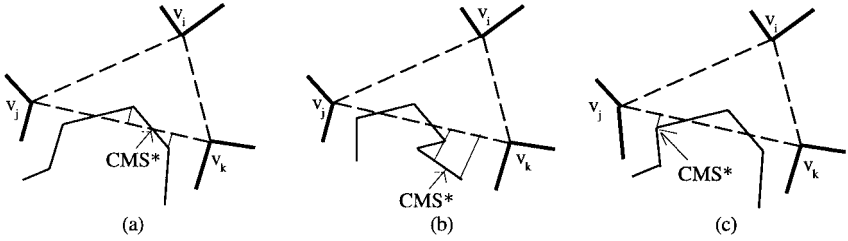


FIG. 8. Demonstration of the visibility of a *CMS* to a triangulation edge v_jv_k. The visible portion of *CMS** is indicated by its projection on v_jv_k.

Similar to the earlier problem, a *component of the Minkowski sum* (*CMS*) is said to be *visible* to a *triangulation edge* v_iv_k (which is the incoming edge of $\Delta v_iv_jv_k$) if it satisfies the following:

- It is generated during the processing of the subtree rooted at $\Delta v_iv_jv_k$.
- It spans in the *active zone* of v_iv_k.
- A line drawn from any point on v_iv_k and perpendicular to it meets (cuts) the *CMS* before cutting any other component of the Minkowski sum or some edge of the polygon.

In Fig. 8 we demonstrate three situations with respect to a component of the Minkowski sum, say *CMS**, and a triangulation edge v_iv_k of $\Delta v_iv_jv_k$: (a) all the points of the *CMS** are visible, (b) a part of the *CMS** is visible, and (c) only a single point of the *CMS** is visible. Without loss of generality, the *visibility list* of a triangulation edge v_iv_k is denoted as *V-LIST*(v_iv_k) and it contains the portions of all the *CMS*s that are visible from v_iv_k.

6.2. *Merging a Pair of V-LISTs*

Let $\Delta v_iv_jv_k$ be the triangle under current processing. Its incoming edge is v_iv_k and the two outgoing edges are v_jv_i and v_jv_k. The intersection(s) among *CMS*s present in *V-LIST*(v_jv_i) and *V-LIST*(v_jv_k) are detected by merging them from their end corresponding to v_j with the help of two pointers indicating the current element of the respective lists.

If more than one intersection is observed, it indicates a hole inside the area defined by the Minkowski sum. To facilitate the detection of holes, we attach a single bit field, called the *status*, to each triangle. It contains 0 or 1 to indicate the current status during the merge pass inside the triangle. The implications of two different values of the *status* bit of a triangle are illustrated below.

• Initially, all the triangles at the leaf level will have 0 in their *status* bit.

• The triangle corresponding to a nonleaf node inherits the value of its *status* bit from its successor(s).

• After encountering an intersection, the *status* bit of the node corresponding to that triangle is toggled.

• During traversal along a path, when the *status* bit is set to 1 it indicates that the portions of the participating *CMS*s above the intersection (toward its predecessor) are exposed as the boundary of the Minkowski sum. This may correspond to a hole or to the outer boundary of the Minkowski sum.

• As soon as the next intersection along that path is noted the *status* bit is toggled to 0 and it indicates that a hole has been generated to the left of the point of intersection (towards the vertex from where the merge pass has been initiated). Portions to the right of the point of intersection will not be exposed as the boundary of the Minkowski sum until the next intersection is found.

During the post-order traversal, when a triangle is processed the line L guiding the merge pass will not necessarily remain the bisector of $\angle v_i v_j v_k$ as in the previous problem. Consider the bounding lines of the active zone of two outgoing edges $v_i v_j$ and $v_j v_k$ inside $\Delta v_i v_j v_k$ which intersect at a point p. Here, the line L is obtained by joining p and v_j.

Suppose $CMS^* \in V\text{-}LIST(v_i v_j)$ needs to be checked with the members of $V\text{-}LIST(v_j v_k)$. Let (a_1, a_2) be the projections of the (portion of) CMS^* on L (Note that a_1 and a_2 may degenerate to a single point). If $\min(v_j a_1, v_j a_2) > v_j p$ the merge pass terminates inside this triangle. Otherwise, we need to test whether the CMS^* intersects with some member of $V\text{-}LIST(v_j v_k)$ by considering their projections on L. As soon as an intersection is noticed with $CMS^{**} \in V\text{-}LIST(v_j v_k)$, the *status* bit of the triangle is toggled. If the resulting *status* bit is 0, it indicates the presence of a hole to the left of the point of intersection. The portions of CMS^* and CMS^{**} to the right of the point of intersection is removed, the hole is reported traversing the $\mathscr{L}$ list starting from CMS^* and ending at CMS^{**}, and $\mathscr{L}$ is set to *null*. If the resulting *status* bit is 1, it indicates the start of either a hole or the outer boundary of the Minkowski sum of $P \oplus Q$. We initialize the $\mathscr{L}$ list with CMS^* and CMS^{**}, after removing the portions to the left of the point of intersection. Also they are connected in the $\mathscr{L}$ list using bidirectional pointers. A *count* field is attached to each *CMS*; the law of incrementing the *count* field will remain the same as that of the earlier problem.

LEMMA 5. *After processing a triangle inside a notch, the resulting $\mathscr{L}$ list contains a chain of CMSs, each of whose consecutive pairs are touching at their common endpoint. This chain of straight line segments defines the outer boundary of the Minkowski sum of the polygonal chain starting and ending at the two vertices of the incoming edge of the aforesaid notch.*

Proof. The first part follows from the manner of constructing the $\mathscr{L}$ list. The second part follows from the discussions in the preceding two paragraphs related to the reporting of holes and establishing connections among two *CMS*s that intersect just after reporting a hole. ▮

Now it remains to discuss (i) the creation of the *V-LIST* of the incoming edge of a triangle during backtrack and (ii) the inheritance of the *status* bit by a nonleaf node from its successor(s) in the triangulation tree $\mathscr{T}$ during the backtrack of the post-order traversal. The creation of *V-LIST* will be discussed in the next subsection. Below we discuss the other aspect.

6.3. *Propagation of Status Bit*

It is easy to follow that the value of the *status* bit that needs to be propagated from a node to its predecessor is

either inherited from its children, if no intersection is observed during the merge pass inside the current triangle, or

set when the last intersection is observed in the merge pass inside the triangle corresponding to the current node.

LEMMA 6. *If the value of the status bit of a triangle is observed to be* 0 *after its processing, it implies that the CMSs of the vertices adjacent to its incoming edge overlap each other during the processing of the triangle. Moreover, the CMS of one vertex adjacent to its incoming edge will not be present in the V-LIST of the incoming edge and hence will not be propagated to its predecessor triangle.*

Proof. The first part of the lemma follows trivially. To prove the second part, let $\Delta v_i v_j v_k$ be the triangle under consideration whose edge $v_i v_k$ is the incoming edge, and let $CMS(v_i)$ and $CMS(v_k)$ overlap. Note that the comparison of $CMS(v_i)$ and $CMS(v_k)$ will be the last comparison during the merge pass inside $\Delta v_i v_j v_k$. Now if the parts of both $CMS(v_i)$ and $CMS(v_k)$ are present in $V\text{-}LIST(v_i v_k)$, then their intersection will set the *status* bit of $\Delta v_i v_j v_k$ to 1 (see Fig. 9a). Otherwise, if the merge pass terminates with a 0 in the *status* bit of $\Delta v_i v_j v_k$ then the $V\text{-}LIST(v_i v_k)$ will contain a single *CMS* which is a part of either $CMS(v_i)$ or $CMS(v_k)$ (see Fig. 9b). ▮

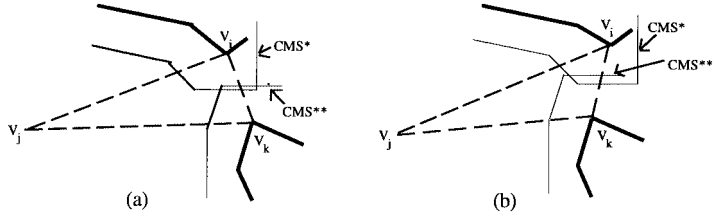


FIG. 9. Proof of Lemma 6.

A nonleaf node corresponding to a *type-B* triangle inherits the value of its *status* bit from its children. If the node under current processing corresponds to a *type-C* triangle, then the following two situations may occur:

(i) The *status* bits of both of the children are set to 1, in which case the *status* bit of the current node is unambiguously set to 1 and the merge proceeds in the current node.

(ii) The status bit of at least one of the two children of the current node is set to 0. In that case, by Lemma 6, the *CMS* of the element adjacent to at least one of its vertices will not be exposed in the boundary of $P \oplus Q$. So, we can easily initiate the merge pass inside the current triangle by setting the value of its *status* bit to 0.

It needs to be mentioned that if only the outer boundary of the Minkowski sum needs to be reported as in the previous problem, the *status* bit is not required. Here as soon as a new intersection is observed the entries corresponding to the participating *CMS*s need to be updated by removing the portions to the left of the point of intersection, and these two *CMS*s are connected in the $\mathscr{L}$ list.

6.4. *Creation of New V-LIST*

The creation of new *V-LIST* for the incoming edge of a triangle can be done in a manner exactly similar to that of the previous problem. To ensure the linear time complexity, we need to prove that the *count* field of a *CMS* may be incremented in the merge pass of at most two triangles. Let us consider our axes of reference, which are parallel to the two orthogonal edges of the rectangle Q. We shall use two terms, "vertical visibility" and "horizontal visibility," with respect to that.

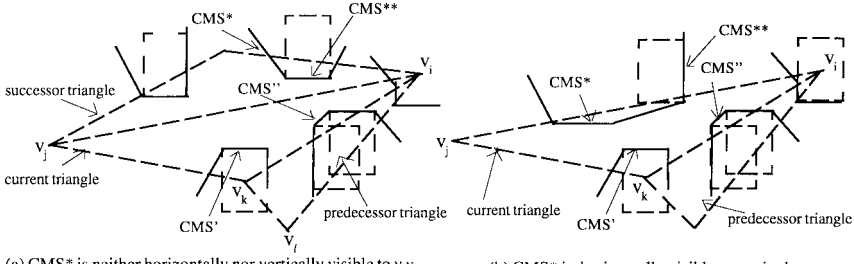
Let $\mathscr{A} \subset V\text{-}LIST(v_iv_j)$ be the set elements that are considered in the merge pass inside $\Delta v_iv_jv_k$ and are propagated to its predecessor triangle (i.e., the triangle attached to its parent in $\mathscr{T}$) during a backward pass. The *count* fields of all the members of $\mathscr{A}$ are incremented except for the last

one, say *CMS***, since it may be visible to some other edge of its predecessor triangles in the same direction (horizontal/vertical) as that from v_jv_k, and a *CMS* belonging to the *V-LIST* of that edge may intersect it. Conventionally, the cost of considering *CMS*** in the merge pass inside the current triangle $\Delta v_iv_jv_k$ is charged to $\Delta v_iv_jv_k$. Let us consider two exhaustive and mutually exclusive subsets of $\mathscr{A} - CMS^{**}$, say $\mathscr{A}_1$ and $\mathscr{A}_2$, such that the *count* field of the elements of $\mathscr{A}_1$ and $\mathscr{A}_2$ are 1 and 2 respectively at the end of a merge pass inside the current triangle.

LEMMA 7. *The count field of the members of $\mathscr{A}_1$ may be incremented in the merge pass of at most one more triangle during the backtrack up to the root of $\mathscr{T}$, and that of $\mathscr{A}_2$ will not be considered in the merge pass of any other triangle.*

Proof. Let *CMS** be an element in *V-LIST*(v_iv_j) which is compared with *CMS*′ $\in$ *V-LIST*(v_jv_k) in the current merge pass. Let them be visible to each other vertically. As *CMS** is not the last element considered during the current merge pass, it will not be vertically visible to any other edge of its predecessor triangles (see Fig. 10b). Some other *CMS*, say *CMS*″, may intersect *CMS** if *CMS** and *CMS*″ are visible to each other horizontally. Now, if *CMS** $\in \mathscr{A}_1$ this may be possible; but if *CMS** $\in \mathscr{A}_2$ its *count* field is already set to 2. This indicates that it has already been compared with some CMS (say *CMS*** in Fig. 10a) during the processing of one of the children of the current node in $\mathscr{T}$, which spans on it horizontally and destroys the horizontal visibility from any other triangulation edge. ∎

It should be noted here that this intersection may not be observed in the immediate predecessor of $\Delta v_iv_jv_k$. So the propagation of a *CMS* continues



(a) CMS* is neither horizontally nor vertically visible to v_iv_t in the predecessor of current triangle

(b) CMS* is horizontally visible to v_iv_t in the predecessor of current triangle

CMS** is the last compared element of V-LIST(v_iv_k) with the members of V-LIST(v_jv_k) in both (a) and (b)

FIG. 10. Proof of Lemma 7.

until it fails to belong in the active zone of the incoming edge of the current triangle or its *count* field attains a value 2.

Thus Lemma 7 leads to the following theorem stating the complexity results of the problem considered in this section.

THEOREM 4. *The time and space complexities of our proposed algorithm for computing the Minkowski sum of an arbitrary simple polygon and a rectangle are both* $O(N)$, *where* N *is the total number of vertices of the polygon.*

7. MINKOWSKI SUM OF A SIMPLE POLYGON AND A CONVEX POLYGON

Next, we show that the same technique efficiently computes the Minkowski sum of a simple polygon and a convex polygon. According to the prior convention, we refer to the above two polygons as P and Q respectively. Here, the most crucial thing is to appropriately redefine l and l' which describe the active zone of a triangulation edge, say v_iv_k. Let Q_i and Q_k be two copies of Q, placed at v_i and v_k respectively. Now, l and l' are a pair of tangents of Q_i and Q_k, parallel to v_iv_k.

LEMMA 8. *The time required for computing the active zones of a triangulation edge of a notch may be* $O(\log M)$ *in the worst case.*

Proof. We may assume that the ordered list of vertices of the convex polygon Q is stored in a circular array. Now, the tangents of Q parallel to a given straight line can easily be found using a binary search. ∎

Here we add two additional fields to each edge of $\mathcal{T}$. At the beginning of the execution of a notch, we compute l and l' of each triangulation edge and attach them to the corresponding node in $\mathcal{T}$. The time required to compute the active zones of all triangulation edges of a notch is $O(N_i \log M)$ in the worst case, where N_i is the number of vertices of the notch. Thus considering all the notches, this additional preprocessing step may consume $O(N \log M)$ time in the worst case.

Next, it needs to be mentioned that, for a given edge or a vertex c_i of P, the complexity of $c_i \oplus Q$ depends on the number of vertices of Q. So, unlike the problem discussed in the previous section, here each straight line segment of $c_i \oplus Q$ will be referred to as a *component of the Minkowski sum* (*CMS*). The generation of *CMS*s, the merging of a pair of *V-LIST*s while processing a triangle, and the propagation of *CMS*s from one triangle to its predecessor will remain the same as we have used in the previous sections. But in order to claim the desired time complexity, we

need to prove Lemma 9, stated below, on the basis of the following observation.

Observation 4. Two straight-line segments λ_i and λ_j intersect if the axes-parallel rectangles with λ_i and λ_j as diagonals also intersect. Note that the converse may not always be true.

LEMMA 9. *After the generation of a CMS, its count field may be incremented in at most two merge passes during its propagation toward the root of* $\mathcal{T}$.

Proof. In order to prove this result, we choose a pair of orthogonal axes and define the vertical/horizontal visibility in reference to that.

While merging a pair of *V-LIST*s, let us consider the axes-parallel rectangles of all the participating *CMS*s. Now the following observations are important.

A pair of *CMS*s are compared during this merge if the corresponding axes-parallel rectangles are mutually visible in either the horizontal or the vertical direction.

If a *CMS*, belonging to the *V-LIST* of an outgoing edge of the current triangle, participates in the comparisons during the merge pass, its corresponding rectangle will partially or completely lose its visibility in one (horizontal/vertical) direction.

Particularly, if the *count* field of a *CMS* is incremented due to a comparison during this merge, it implies that one side of the corresponding rectangle completely loses its (horizontal or vertical) visibility from its predecessor triangle.

However, if the *count* field of a *CMS* is not incremented due to a comparison during this merge, this implies that the corresponding *CMS* participated in the last comparison of the current merge pass. So, the visibility of the rectangle corresponding to that *CMS* is partially lost from the direction (horizontal/vertical) under consideration during the current merge pass. So, a rectangle corresponding to some other *CMS*, present in some of its predecessor triangle(s), may span over it from the same direction. This may cause an intersection between the aforesaid two *CMS*s.

If a *CMS* does not participate in the current merge pass, no question of incrementing its *count* field arises.

Now the proof of this lemma follows from the fact that Lemma 7 remains valid for rectangles of unequal aspect ratios also. ∎

Using Lemmas 8 and 9, and arguing as in Section 5.3, we may state the last result of our work as follows.

THEOREM 5. *The time and space complexities of our proposed scheme for computing the Minkowski sum of an arbitrary simple polygon and a convex polygon are both $O(MN)$, where N and M denote the number of vertices of the above two polygons respectively.*

8. CONCLUSION

In this paper we introduce the problem of locating the safety zone of a simple polygon and present a linear time algorithm for drawing the boundary of the safety zone which uses Chazelle's algorithm for triangulating simple polygons in linear time. The safety zone of width δ for a simple polygon is a simple closed area defining the outer boundary of the Minkowski sum of the polygon and a circle of radius δ. In order to get the holes generated by the Minkowski sum inside the safety zone, our algorithm can easily be tailored, keeping time and space complexities invariant. The same technique works for computing the Minkowski sum of an arbitrary simple polygon and a convex polygon in time proportional to the product of the number of vertices of both the polygons.

ACKNOWLEDGMENT

We acknowledge the two anonymous referees for their valuable suggestions regarding the presentation of the paper.

REFERENCES

1. B. Chazelle, Triangulating a simple polygon in linear time, *Discrete Comput. Geom.* **6** (1991), 485–524.
2. A. Hernández-Barrera, Computing the Minkowski sum of monotone polygons, *IEICE Trans. Inform. Systems* **E80-D**, No. 2 (1996), 218–222.
3. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, "Computational Geometry: Algorithms and Applications," Springer-Verlag, Berlin, 1997.
4. B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink, Ray shooting in polygons using geodesic triangulation, *in* "Proceedings, 18th International Colloquium on Automata Language Programming," Lecture Notes in Computer Science, Vol. 510, pp. 661–673, Springer-Verlag, New York/Berlin, 1991.
5. F. Chin, J. Snoeyink, and C.-A. Wang, Finding the medial axis of a simple polygon in linear time, *in* "Proceedings, 6th Annual International Symposium on Algorithms and Computation (ISAAC 95)," Lecture Notes in Computer Science, Vol. 1004, pp. 382–391, Springer-Verlag, New York/Berlin, 1995.
6. S. Fortune, A fast algorithm for polygon containment by translation, *in* "Proceedings, 12th International Colloquium on Automata Language Programming," Lecture Notes in Computer Science, Vol. 194, pp. 189–198, Springer-Verlag, New York/Berlin, 1985.

7. M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, Triangulating a simple polygon, *Inform. Process. Lett.* **7** (1978), 175–179.
8. R. Klein and A. Lingas, A linear-time randomized algorithm for the bounded Voronoi diagram of a simple polygon, *in* "Proceedings, 9th Annual ACM Symposium on Computational Geometry," pp. 124–132, Association for Computing Machinery, New York, 1993.
9. D. T. Lee, On finding the convex hull of a simple polygon, *Internat. J. Comp. Inform. Sci.* **12**, No. 2 (1983), 87–98.
10. T. Ohtsuki, "Layout Design and Verification," North-Holland, Amsterdam, 1986.
11. G. D. Ramkumar, An algorithm to compute the Minkowski sum outer face of two simple polygon, *in* "Proceedings, 12th Annual ACM Symposium on Computational Geometry," pp. 234–241, Association for Computing Machinery, New York, 1996.
12. R. Seidal, A simple and fast incremental randomized algorithm for computing the trapezoidal decompositions and for triangulating polygons, *Comput. Geom.* **1** (1991), 54–61.
13. N. Sherwani, "Algorithms for VLSI Physical Design Automation," Kluwer Academic, Boston, 1993.