Chapter: 3

Algorithms for Computing Two-Dimensional Guard Zones

3.1 Overview

Two-Dimensional (2D) guard zone computation problem undoubtedly occupies a majority of interest in the field of VLSI physical design automation and design of embedded systems, as resizing is an important problem in VLSI layout design as well as in embedded system design. If two or more subcircuits are close enough, it may result in a parasitic effect that disrupts the performance of the circuit. Thus, with respect to resizing problems in VLSI, this is the motivation for defining the safety zone of a polygon [60]. The width of the safety zone depends on the polygon, i.e. the type of the corresponding circuit and the free space required to perform its task safely. In this context, it is also essential to detect overlapping (if any) among the guard zonal regions and accordingly remove them to obtain the resultant guard zone. Through this procedure, we may also find a hole (if any) in the guard zone and sometimes these holes may be used to place a subcircuit in order to utilize the chip area compactly. In this chapter, we develop a number of sequential algorithms to compute guard zone of a simple 2D polygon including the detection and exclusion of the overlapped regions, if any.

To solve the guard zone computation problem, one of the wonted methods is Minkowski sum [44, 45, 66], which can be used to draw a parallel line with respect to a given polygonal edge. Actually, Minkowski sum between a line and a point with same x- and y-coordinates gives a line parallel to the given one. However, the question arises is whether the parallel line is inside or outside the polygon. Here the definition of Minkowski sum [32] can be reminded again as follows.

If $A$ and $B$ are subsets of $\mathbb{R}^n$, and $\lambda \in \mathbb{R}$, then $A+B = \{x+y \mid x \in A, y \in B\}$, $A-B = \{x-y \mid x \in A, y \in B\}$, and $\lambda A = \{\lambda x \mid x \in A\}$. Note that $A+A$ does not equal $2A$, and $A-A$ does not equal ‘zero’ in any sense.
An alternative method to solve the guard zone problem is the use of convolution [3, 32]. The convolution between a polygon and a circle of radius \( r \) gives us the desired solution. However, the circles need to be drawn in every possible point of the polygon and consequently the time complexity of the algorithm increases. Minkowski sum and convolution theories find broad applications in Mathematics, Computational Geometry, resizing of VLSI circuit components, and in many other problems.

### 3.2 Computing Guard Zone of a Simple Polygon

#### 3.2.1 Problem Formulation

In this section, we present a time optimal sequential algorithm for computing a boundary of guard zone that uses simple analytical and coordinate geometric concepts.

![Figure 3.1](image.png)

**Figure 3.1:** Part of a polygon with vertices \((x_1,y_1)\) through \((x_8,y_8)\), and edges \(a\) through \(g\); the dotted line specifies the inner portion of the polygon.

At the beginning of computing guard zone of a simple polygon, we are to characterize a simple polygon. In the case of a convex polygon, each polygonal edge has corresponding straight line segment as its guard zone, and at the convex vertex, the guard zone is circular in shape. The circular arc formed at the convex vertex must be of radius \( r \), the width of the guard zone that is provided in the problem specification. In general, a simple polygon may contain both convex and concave vertices in it. The vertices have been
defined as follows: A vertex $v$ of polygon $P$ is defined as convex (concave), if the angle between its associated edges inside the polygon, i.e. the internal angle formed at vertex $v$, is less than (greater than) 180°. Now consider Figure 3.1. Here angle $\theta_1$, between edges $a$ and $b$ at vertex $(x_2, y_2)$, is less than 180° whereas angle $\theta_2$, between edges $c$ and $d$ at vertex $(x_4, y_4)$, is greater than 180°. Clearly, we may observe that if an internal (external) angle formed at a vertex is less than (greater than) 180°, then the external (internal) angle formed at the said vertex is greater than (less than) 180°, which is actually convex (concave). Hence, in Figure 3.1, the polygonal vertices $(x_2, y_2)$, $(x_3, y_3)$, $(x_6, y_6)$, and $(x_7, y_7)$ are convex whereas polygonal vertices $(x_4, y_4)$ and $(x_5, y_5)$ are concave. The angle determination step (convex/concave) of a given simple polygon can eventually be computed using simple analytical and coordinate geometry only.

Now, let us consider a simple polygon $P$ with $n$ vertices and $n$ edges. A polygon $P$ is given implies that the coordinates of the successive vertices of the polygon are given, where no two polygonal edges cross each other; rather, two consecutive polygonal edges intersect only at a polygonal vertex. It can be assumed that a portion of such a polygon is as shown in Figure 3.1. To know whether an angle $\theta$, inside the polygon, is either less than 180° (or convex) or greater than 180° (or concave) at vertex $v$, the following constant-time computation is performed for determining the value of $\theta$ between two consecutive polygonal edges intersected at vertex $v$.

In $\mathbb{R}^2$, for a given set of three or more connected vertices that form a simple polygon, the orientation of the resulting polygon is directly related to the sign of the angle at any vertex of the convex hull of the polygon. For example, to determine the type of angle formed between edges $a$ and $b$ with coordinates $X_A (x_1, y_1)$, $X_B (x_2, y_2)$, and $X_C (x_3, y_3)$, the following equation is being used that takes constant time for finding out the angle whether it is convex or concave [30].

$$\det(O) = (X_B - X_A)(Y_C - Y_A) - (X_C - X_A)(Y_B - Y_A)$$

The sign of $\det(O)$ helps to identify the type of angle being formed at the polygonal vertex $X_B$. A positive value indicates a convex angle outside the polygon whereas a negative value indicates a concave angle outside the polygon, and a value zero indicates that the points $X_A$, $X_B$, and $X_C$ are co-linear. In this way, all the $n$ polygonal angles of $P$ are identified.
as convex or concave in $O(n)$ time and avoided the use of comparatively expensive (computationally) trigonometric functions.

At this point, it is fair to conclude that the guard zone is computed only with the help of $n$ straight line segments and $n$ circular arcs if all the $n$ polygonal vertices form convex (external) angles. However, for a given simple polygon, it may have concave angles as well, in $P$. Problems may arise in computing guard zone for these portions of polygon $P$ with concave (external) angles. In this context, the concept of the notch has been introduced as defined below.

A notch is a polygonal region outside polygon $P$ that is formed with a chain of edges of $P$ initiating and terminating at two vertices of a false hull edge [30]. A false hull edge is a hull edge introduced in obtaining a convex hull. A convex hull is a convex polygon (having no concave external angle) of the minimum area with all the points residing on the boundary or inside the polygon for a given set of arbitrary points on a plane.

**Figure 3.2:** A notch is formed inside (or below) the false hull edge formed by vertices $v_2$ and $v_8$, and a guard zone is obtained for this notch as shown by dotted lines and circular arcs outside of the polygon.
Apparently, if $P$ is a given simple polygon and $\text{CH}(P)$ denotes the convex hull of polygon $P$, then the area $\text{CH}(P) - P$ consists of a number of disjoint notches outside polygon $P$. According to this definition, a notch is formed outside the polygon in Figure 3.2, below (or inside) the dotted line $v_2v_8$ as this edge is a false hull edge.

Now for the sake of convenience, in order to develop our algorithm, we redefine notch as the following. A notch is a polygonal region outside the polygon, which is initiating and terminating between two consecutive (or nearest) convex vertices of the polygon that are not adjacent. That means, a notch starts with a convex vertex, goes through one or more concave vertices, and terminates with a convex vertex, along the edges (and vertices) of the polygon (in one particular direction). According to this definition, a notch is formed between polygonal vertices $v_3$ and $v_7$ as $v_4$, $v_5$, and $v_6$ are concave vertices whereas $v_3$ and $v_7$ are convex, as shown in Figure 3.2.

### 3.2.2 Development of Algorithm Guard Zone

A guard zone $G$ of a simple polygon $P$ can easily be obtained with the help of two pens, say $Pen_1$ and $Pen_2$, whose tips are always at distance $r$ apart, as stated below. Let the vertices of polygon $P$ are labelled as $v_1, v_2, \ldots, v_n$, moving along its boundary in a clockwise direction. Let the tip of $Pen_1$ moves along the polygonal boundary from vertex $v_1$ through $v_n$ (to $v_1$), and the tip of $Pen_2$ draws the boundary of the guard zone, staying orthogonal to the direction of motion of $Pen_1$. While drawing the guard zone, two cases arise.

(i) If $Pen_1$ finds a convex vertex (as at $(x_2,y_2)$ in Figure 3.3), then the direction of $Pen_2$ is changed by drawing a circular arc of radius $r$, with center at that convex vertex.

(ii) If $Pen_1$ residing at point $P_1$ (on the polygon) finds another point $P_2$ on the boundary of the polygon such that $P_1T + TP_2 = 2r$ (where $Pen_2$ resides at point $T$), and $P_1T$ and $TP_2$ are completely outside the polygon (see Figure 3.3), and then $Pen_1$ moves to $P_2$ and $Pen_2$ remains at its present position $T$. Then $Pen_1$ starts moving toward the next (higher numbered) vertex attached to that edge; the movement of $Pen_2$ has to be guided by that of $Pen_1$, as described earlier. This happens only when a concave vertex (as at $(x_3,y_3)$ in Figure 3.3) of the polygon is found, where $P_1P_2$, i.e. the distance between points $P_1$ and $P_2$, is less than $2r$. 


So, $Pen_1$ moves through the boundary of the polygon, shown by the firm line in Figure 3.3, and $Pen_2$ moves through the dotted line, always at a distance $r$ apart from (and outside) the polygon, which is actually the guard zone.

Now, in order to develop a sequential algorithm for computing a guard zone $G$ of a simple polygon $P$ is stated as follows. Usually, a guard zone contains line segments that are parallel to the edges of $P$. A circular arc-shaped portion of a guard zone is only obtained at a convex vertex of the polygon. Thus, in order to develop an algorithm, the edges have to be considered one after another and have to execute the following tasks.

Let us consider a simple polygon $P$ with $n$ vertices (and $n$ edges). A polygon $P$ is given implies that the coordinates of the successive vertices of the polygon are given, where no two polygonal edges cross each other, rather two consecutive polygonal edges intersect only at a polygonal vertex. We can assume that a portion of this polygon is as shown in Figure 3.1. To know whether an angle $\theta$, inside the polygon, is either convex or concave at vertex $v$, we do the following constant-time computation of determining the value of $\theta$ between two consecutive polygonal edges intersected at vertex $v$.

Let the slope of edge $a$ be $m_a$ and that of edge $b$ be $m_b$. Therefore, $m_a = (y_2-y_1) / (x_2-x_1)$ and $m_b = (y_3-y_2) / (x_3-x_2)$. Hence, $\tan \theta = (m_a-m_b) / (1+m_a m_b)$ gives $\theta = \tan^{-1}\left(\frac{m_a-m_b}{1+m_a m_b}\right)$. Alternatively, the external angles can also be calculated by determining the sign of determinant of the coordinates of any two adjacent edges, as

![Figure 3.3: Lines of a guard zone (dotted) corresponding to polygonal edges (firm), a convex vertex at $(x_2,y_2)$, and a concave vertex at $(x_3,y_3)$.](image-url)
described in previous section, which also essentially avoids the use of comparatively costly trigonometric functions, such as \( \tan() \). In this way, all \( n \) internal angles of \( P \) are identified as convex or concave in \( O(n) \) time. At this point, we can conclude that the guard zone is computed only with the help of \( n \) straight line segments and \( n \) circular arcs if all the \( n \) external angles of the polygon are convex.

Here we develop a time optimal sequential algorithm for computing a guard zone \( G \) of a simple polygon \( P \) as described below. Usually, a guard zone contains line segments that are parallel to the edges of \( P \). A circular arc-shaped portion of a guard zone is only obtained at a convex vertex of the polygon. Thus, in developing the algorithm, we consider edges one after another and do the following.

Consider the edge \( v_2v_3 \) in Figure 3.2. Here both the vertices \( v_2 \) and \( v_3 \) are convex. Therefore, a line segment parallel to \( v_2v_3 \) and of length same as \( v_2v_3 \), for \( G \) is computed at a distance \( r \) apart from the polygonal edge with a circular arc of radius \( r \) centered at vertex \( v_2 \). Clearly, the line segment parallel to \( v_2v_3 \) is a tangent to the circular arc centered at \( v_2 \).

Now consider the edge \( v_3v_4 \). Here \( v_3 \) is a convex vertex whereas \( v_4 \) is concave. So, the length of the line segment for \( G \) parallel to \( v_3v_4 \) is less than that of \( v_3v_4 \). Certainly, the line segment of \( G \) touches (as a tangent) the circular arc centered at \( v_3 \) at point \( p \) (in Figure 3.2), but on the other end (near \( v_4 \), which is a concave vertex) the line segment of \( G \) is prolonged up to point \( q \), as stated below. Point \( q \) is the intersection point of the bisection \( bs_1 \) of the angle between polygonal edges \( v_3v_4 \) and \( v_4v_5 \) (i.e. the angle at vertex \( v_4 \)), and the line segment of \( G \) passes through \( p \) and parallel to \( v_3v_4 \). The reason for doing this has been clearly explained in Figure 3.3, for a concave vertex at \((x_3, y_3)\), where \( P_1T = TP_2 = r \).

Now consider polygonal edges \( v_4v_5 \), where both polygonal vertices \( v_4 \) and \( v_5 \) are concave. Thus, it is not required to draw any circular arc here for this portion of \( G \). The line segment of \( G \) is obtained by computing a line going through \( q \) up to \( s \), where \( s \) is the intersection of the line segment of \( G \) parallel to \( v_4v_5 \) and the bisection \( bs_2 \) of the (internal) angle at vertex \( v_5 \). Definitely, the length of \( qs \) is less than that of \( v_4v_5 \).

It may so happen that the length of \( v_4v_5 \) is sufficiently small and the internal angles at \( v_4 \) and \( v_5 \) are large enough such that the distance between the intersection point \( f \) of \( bs_1 \) and \( bs_2 \) to \( v_4v_5 \) is less than the distance between the intersection point \( g \) of the two line
segments of $G$ parallel to $v_3v_4$ and $v_5v_6$ to $v_4v_5$ (see Figure 3.4). In this case, there is no line segment of $G$ parallel to polygonal edge $v_4v_5$. In general, instead of a single edge like $v_4v_5$, several edges or even several notches of a simple polygon may be there for which no guard zone is distinguishingly computed.

**Figure 3.4:** A portion of the guard zone without any line segment parallel to the polygonal edge $v_4v_5$.

In case of the “pen” algorithm, the human being or software module responsible for moving the pen must have identified such a scenario by introducing some preprocessing of the given input polygon even before the start of guard zone computation. This prior knowledge of the input polygon helps the human being or software module to determine how the pen needs to be moved to compute the desired guard zone with a certain degree of acceptable approximation necessary in the presence of such a notch (or any special case, if any). Moreover, in the case where software module is responsible for moving the pens then the same methodology as discussed in the previous paragraph, with a look-ahead mechanism preprogrammed within the software module can be utilized to control the movement of the pens without necessarily a preprocessing step, which is more applicable for human beings to be able to adjust the pen movements in the presence of such a notch.

Here we like to mention that all these computations could be performed with the help of simple analytical and coordinate geometry, and mathematics. Besides, all these computations required to obtain a portion of the guard zone for a polygonal edge or a polygonal vertex can also be performed in constant time. Hence, as has been devised, the
entire guard zone $G$ of a simple polygon $P$ can be computed in linear time. This result has been established as a theorem. Before that, we briefly state the steps of the algorithm $Guard\_Zone$, for computing a guard zone of a simple polygon, as given below.

**Algorithm $Guard\_Zone$**

**Input:** A simple polygon $P$.

**Output:** A guard zone of polygon $P$.

**Step 1:** Clockwise label the vertices $v_1, v_2, \ldots, v_n$ of polygon $P$.

**Step 2:** For $i = 1$ to $n-1$ do

  **Step 2.1:** If the internal angle at $v_i$ is convex, then
  
  **Step 2.1.1:** Draw a circular arc (outside the polygon) of radius $r$ centered at $v_i$.
  
  **Step 2.1.2:** Find the internal angle at $v_{i+1}$, and consider polygonal edge $(v_i, v_{i+1})$.
  
  **Step 2.1.3:** If the internal angle at $v_{i+1}$ is convex, then
  
  **Step 2.1.3.1:** Draw a circular arc (outside the polygon) of radius $r$ centered at $v_{i+1}$.
  
  **Step 2.1.3.2:** Draw a line parallel to $(v_i, v_{i+1})$ at a distance $r$ apart from the polygonal edge (outside the polygon) that is a simple common tangent to both the arcs drawn at $v_i$ and $v_{i+1}$.
  
  **Else**
  
  **Step 2.1.3.3:** Bisect the (internal) angle at $v_{i+1}$, denote the bisection $b_{S_{i+1}}$.
  
  **Step 2.1.3.4:** Draw a line parallel to $(v_i, v_{i+1})$ at a distance $r$ apart from the polygonal edge (outside the polygon) that is a tangent to the arc drawn at $v_i$ and intersects $b_{S_{i+1}}$ at a point, say $p_{i+1}$.

  **Step 2.1.4:** Assign $i \leftarrow i+1$.

  **Step 2.1.5:** If $v_i = v_n$ then $v_{i+1} = v_1$.

  **Else**

  **Step 2.1.6:** Bisect the (internal) angle at $v_i$, denote the bisection $b_{S_{i}}$.

  **Step 2.1.7:** Find the internal angle at vertex $v_{i+1}$, and consider polygonal edge $(v_i, v_{i+1})$.

  **Step 2.1.8:** If the internal angle at $v_{i+1}$ is convex, then
**Step 2.1.8.1:** Draw a circular arc (outside the polygon) of radius $r$ centered at $v_{i+1}$.

**Step 2.1.8.2:** Draw a line parallel to $(v_i, v_{i+1})$ at a distance $r$ apart from the polygonal edge (outside the polygon) that intersects $bs_i$ at a point, say $p_i$ and is tangent to the arc drawn at $v_{i+1}$.

*Else*

**Step 2.1.8.3:** Bisect the (internal) angle at $v_{i+1}$, denote the bisection $bs_{i+1}$.

**Step 2.1.8.4:** Draw a line parallel to $(v_i, v_{i+1})$ at a distance $r$ apart from the polygonal edge (outside the polygon) that intersects $bs_i$ at a point, say $p_i$ and also intersects $bs_{i+1}$ at a point, say $p_{i+1}$.

**Step 2.1.9:** Assign $v_i \leftarrow v_{i+1}$.

**Step 2.1.10:** If $v_i = v_n$, then $v_{i+1} = v_1$.

**End for**

**Step 3:** If two line segments or a line segment and a circular arc or two circular arcs of the guard zone intersect, then exclude the portions of the line segment(s) and/or the circular arc(s) that are at a distance less than $r$ apart from a polygonal edge or a polygonal vertex (outside the polygon).

### 3.2.3 Computational Complexity of Algorithm Guard Zone

It is straightforward to observe that the guard zone of an $n$-vertex convex polygon is a convex region with $n$ straight line segments and $n$ circular arcs only. The straight line segments of the guard zone are parallel to the edges of the polygon at a distance $r$ outside the polygon, and two consecutive line segments of the guard zone are joined by a circular arc of radius $r$ centered at the corresponding polygonal vertex. As a result, the computational time required for figuring the guard zone of a convex polygon is $O(n)$ [48, 50, 55].

The situation is a bit complicated if notches rather concave vertices belong to a simple polygon. Besides, for the presence of a concave polygonal vertex, we bisect the polygonal angle (instead of introducing a circular arc outside the polygon) and draw a line parallel to the polygonal edge under consideration. For a polygonal edge or an angle of the polygon, either convex or concave, all these computations take constant time. Therefore,
for a polygon of \( n \) vertices (and \( n \) edges), the overall worst case time required in computing a guard zone of a simple polygon is \( O(n) \).

**Figure 3.5:** Different types of probable intersections of adjacent guard zonal segments. (a) The intersection of two guard zonal circular arcs due to adjacent convex vertices. (b) The intersection of a guard zonal circular arc and a (guard zonal) straight line segment. (c) A polygon consisting of a number of convex and concave vertices whose guard zonal regions are likely to overlap.

### 3.3 An Algorithm to Detect and Exclude the Overlapped Regions in Guard Zone

#### 3.3.1 Formulation of the Problem

The difficulty arises in computing the guard zone in situations where there is some part(s) of the guard zone \( G \) that overlaps(s). Figures 3.5(a) and 3.5(b) depict two such possible overlapping among many others that might get encountered after the initial guard zone computation. Figure 3.5(a) depicts the probable overlapping between the guard zones of two convex regions, where the guard zone at convex vertex \( A \) may intersect the guard zone at \( B \), hence forming a hole inside. Again, it may happen between the guard zone of a convex vertex and the guard zone of a straight line segment as shown in Figure 3.5(b), where the guard zone at the convex vertex \( C \) may have an intersection with that of the line segment \( AB \). Here we refer the circular arc as the guard zone of the corresponding convex vertex and the line segments in the guard zone are referred to be guard zone of the corresponding polygonal edges. We can assume another case showing a portion of a polygon in Figure
3.5(c), where $B$ is a convex vertex and $A$ is concave, and the guard zones of these two regions may overlap. Again the guard zones of linear portions $BD$ and $BE$ can overlap the guard zones of linear portions $AC$ and $AF$. All the above cases may arise while computing the guard zone for a given simple polygon.

Figure 3.6: Recursive division of a convex polygonal angle formed at vertex $v$ wherefrom equal-length smallest possible chords are computed that in a group replaces the circular arc that is computed as a part of guard zone outside the polygon up to the desired level of precision of an angle that is formed at the convex vertex for each such smallest possible chord.

3.3.2 Development of the Algorithm

As our inclination in doing the task is exclusively by means of computational geometry, we like to exclude the part(s) of $G$ that overlap(s) using the concept of analytical and coordinate geometry. In the prior computation we have at most $O(n)$ straight line segments and $O(n)$ circular arcs in computing $G$. Thus, to find out all the intersection points and exclude the overlapped region(s) accordingly (in order to get the desired guard zone only, including holes, if any, as parts of $G$), we may execute an $O(n^2)$ time algorithm for each pair of such segments, among all straight line segments and circular arcs. Indeed, this algorithm is greatly expensive. Hence, while computing the initial guard zone $G$, we enclose $P$ by $G$, which is essentially a collection of $O(n)$ line segments only (and there is
no circular arc as part of $G$ that has been drawn so far for each of the convex polygonal vertex of $P$ [52, 53]; we elucidate it as follows with the help of Figure 3.6.

Presently, we explain how we replace a circular arc that has been drawn so far for each of the convex polygonal vertex $v$ of $P$ with the help of a collection of smaller straight line segments. Next, we claim that the desired guard zone $G$ is computed with the help of $O(n)$ straight line segments only. To show the first part under consideration, we take the help of Figure 3.6 that contains a convex polygonal vertex $v$ along with its associated polygonal edges $uv$ and $vw$. Here for the time being, we do not like to know whether $u$ and $w$ are convex or concave polygonal vertices, as we are only interested in considering a convex polygonal vertex $v$, whose guard zone is to be computed comprising a constant number of (smallest possible) straight line segments instead of a circular arc centering at $v$ with radius $r$.

Without loss of generality, we assume that both the polygonal vertices $u$ and $w$ are also convex. So what we do, we compute two straight line segments $u'v'$ and $v''w'$, where $u'v' \parallel uv$ and $u'v' = uv$, and also $v''w' \parallel vw$ and $v''w' = vw$, and the perpendicular distance between the parallel lines for both the pairs is same as $r$. Hence, we obtain two rectangles $uvv'u'$ and $vww'v''$, where $vv' = uu' = ww' = vv'' = r$, since $u'u$ (or $v'v$) is perpendicular to $uv$ and $w'w$ (or $v''v$) is perpendicular to $vw$. So, $u'v'$ and $v''w'$ are guard zones for the polygonal edges $uv$ and $vw$, respectively. Now we compute guard zone for the polygonal vertex $v$ as follows.

We have already mentioned that the guard zone for polygonal vertex $v$ is composed of a number of straight line segments that collectively replace the guard zonal circular arc (of radius $r$) that we usually draw at a convex polygonal vertex $v$ outside the polygon (where $u'v'$ and $w'v''$ are two tangents to that circular arc). Now we like to do this task recursively using a constant-time computation for each such polygonal vertex $v$, as the value of $\angle v'vv''$ is always less than $180^\circ$. In other words, we may state that we like to replace the circular arc that we usually compute as a part of $G$ for $v$ by exactly $2^p$ number of straight line segments (for some constant $p$) that are equal in length to each other.

In order to obtain the smallest possible line segments, we follow a recursive procedure that is binary in nature. First of all, we bisect $\angle v'vv''$ (or $\angle uvw$) by a bisector $b_{s1}$
whereon \( p_1 \) is a point outside the polygon such that \( v p_1 = r \). We join \( v'p_1 \) and \( p_1v'' \); so, \( v'p_1v'' \) is an approximated guard zone (for \( p = 1 \) of the circular arc we liked to draw. Next to make this approximation finer, we further bisect \( \angle v'v_1p_1 \) by a bisector \( b_{s2} \) whereon \( p_2 \) is a point outside the polygon such that \( v p_2 = r \) and bisect \( \angle p_1v_1v'' \) by a bisector \( b_{s3} \) whereon \( p_3 \) is a point outside the polygon such that \( v p_3 = r \). Then we join \( v'p_2, p_2p_1, p_1p_3 \), and \( p_3v'' \); so, \( v'p_2p_1p_3v'' \) is a finer approximation of the guard zone (for \( p = 2 \) than the previous one (i.e. \( v'p_1v'' \)) of the circular arc we usually draw.

Now, it is implied that for \( p = 3 \), we are supposed to bisect each of the angles \( \angle v'v_2p_2 \) through \( \angle p_3v_2v'' \), and obtain intermediate points \( p_4 \) through \( p_7 \) on each such bisection \( b_{s4} \) through \( b_{s7} \), outside the polygon such that \( v p_4 = v p_5 = v p_6 = v p_7 = r \), and even smaller line segments \( v'p_4 = p_4v_2 = p_2p_5 = p_5p_1 = p_1v_6 = p_6p_3 = p_3p_7 = p_7v'' \), and in due course we obtain an even finer approximation of the guard zone (for \( p = 3 \) than that we computed for \( p = 2 \), which is more closer to the circular arc we usually draw as a part of guard zone for a convex polygonal vertex.

This process of bisection is continued till the value of each bisectional angle becomes 0.50° or 0.25° or up to some precision of angle that makes the straight line segments as chords of the circular arc reasonably very small. At present, it is implicit that all the points \( p_i \) over the bisections outside the polygon are the points on the circular arc as part of \( G \), and each small line segment (whose length tends to zero for a smaller value of \( r \)) is an approximation of its associated arc for which it is the largest chord. So, further bisection of each of the bisected angles in subsequent levels of recursion and marking a point \( p_i \) on each of the bisections at distance \( r \) from \( v \) outside the polygon helps to achieve more points on the said circular arc that are consecutively equidistant and closer to each other. Hence, up to some desired level of precision, we may obtain a set of equal length straight line segments collectively that replaces the circular arc we liked to draw as a part of \( G \) for vertex \( v \).

In any case as the value of \( \angle v'v'' \) is always less than 180°, which is a constant, we claim that the number of bisections or the number of recursive calls to bisect \( \angle v'v'' \) is always a constant (up to an acceptable smallest precision of angle). Even then this method
may lack in finding a point of intersection of two circular arcs or a circular arc and an edge of $G$ where the segments are tangential (or almost tangential) to each other.

However, the probability of occurrences of such a circumstance significantly reduces as in general, the value of $r$ is appreciably small. If some smaller edge (or the initial or final chord or some intermediate chord $p_ip_{i+1}$ that approximates its coupled arc) is found, that intersects with other part(s) of $G$, which is also a line segment, then we may further bisect only that angle recursively to identify a more accurate point of intersection that in time may reduce many redundant computations.

In this context, we like to conclude that the total number of straight line segments as part of the computed guard zone $G$ is at most $O(n)$, as stated in the following lemma.

**Lemma 3.1:** The computed guard zone $G$ consists of $O(n)$ straight line segments for a given polygon $P$ with $n$ polygonal vertices (and $n$ polygonal edges).

**Proof:** The proof is straightforward. For each of the $n$ polygonal edges, we have to draw at most $n$ parallel guard zonal line segments at a distance $r$ apart, where one guard zonal line segment is computed for its associated polygonal edge outside the polygon. Thus, it takes $O(n)$ time. Next, we like to know all the polygonal vertices, whether they are convex or concave; this also takes time $O(n)$. If a polygonal vertex is concave, the only bisection of its angle is sufficient to draw, on which the adjacent guard zonal line segments (that are parallel to adjacent polygonal edges) coincide. This computation takes constant time. On the other hand, if a polygonal vertex is convex, a recursive step of a constant number of bisections of its angle is performed that computes a constant number of smaller straight line segments (as chords) at distance $r$ along each bisection outside the polygon, where each such smaller straight line segment forms an angle at the convex polygonal vertex up to a desired level of precision. In any case, all these computations take constant time, and the number of additional straight line segments for computing $G$ is also constant (for each polygonal vertex $v$). Hence, the total number of straight lines segments of the computed guard zone $G$ is $O(n)$. ♦

The following lemma proves that the number of bisections required to convert a circular arc into a set of line segments is a constant, i.e. bound within some limit independent of the number of edges of the polygon.
Lemma 3.2: A circular arc due to a convex polygonal vertex outside the polygon is subdivided $p$ times replacing the arc by a set of equal $p+1$ straight line segments, where $p$ is constant.

Proof: We consider two extreme cases, i.e. the circular arcs obtained for the convex polygonal vertices with two extreme values of the angles. If we can prove that $p$ is constant for these two cases, it will eventually be true for all the intermediate values possible for an external angle. The convex angle to be divided becomes the minimum when it tends to $0^\circ$, and becomes the maximum when it tends to $180^\circ$. Figures 3.7(a) and 3.7(b) depict these two cases. If the internal angle tends to $180^\circ - \delta$, then the external angle also tends to $180^\circ + \delta$, for some $\delta \to 0^\circ$. Hence, on the perpendicular lines drawn on two adjacent polygonal edges joining at the polygonal vertex $O$, we find two points $A$ and $B$ such that $OA = OB = r$, as shown in Figure 3.7(a). So, the value of $\angle AOB$ tends to $0^\circ$, and we get the line segment (or the sub-arc) $AB$ replacing the circular arc (by connecting $A$ and $B$). Thus, there is no need to bisect the angle for such a situation.

Figure 3.7: (a) The external angle tends to $180^\circ + \delta$; however, the circular arc to be subdivided is associated with an angle tends to $0^\circ$, as $\delta \to 0^\circ$. (b) The external angle tends to $360^\circ - \delta$; however, the circular arc to be subdivided is associated with an angle tends to $180^\circ - \delta$, for some $\delta \to 0^\circ$.

On the other hand, if the external angle tends to $360^\circ - \delta$, observing Figure 3.7(b), then the $\angle AOB$ tends to $(360^\circ - \delta) - (90^\circ + 90^\circ) = 180^\circ - \delta$ is to be subdivided. When a sub-arc makes an angle less than or equal to $4^\circ$ (at some convex polygonal vertex), we can consider the sub-arc to be a line segment. This is true because of the fact that an arc having
an angle less than or equal to 4º may be considered as a line segment as \( \sin \theta \approx \tan \theta \approx \theta \), if \( \theta \leq 4º \) [8]. Again, while subdividing the angle, \( \angle AOB \), if we follow a sequential algorithm, we need to perform the successive bisection operation for a maximum of 44 times, which is a constant, i.e. it does not depend on the value of \( n \), the number of vertices in the given polygon. Hence, \( p \) is always a constant between 0 and 44. ♦

Thus, currently, we have \( O(n) \) straight line segments of \( G \) that are not arbitrary as the given polygon \( P \) is a simple polygon. Whatever be the shape of the simple polygon, the number of intersections among the straight line segments of \( G \) never exceeds \( O(n) \). Now, the line sweep algorithm is applied where the input is a set of line segments associated with their starting and ending coordinates as event points. The sweep line is traversed through the sorted list of event points depending on x- or y-coordinates. Here starting point, ending point, and point of intersection (if any) are the three types of event points at which insertion, deletion, and update of neighbouring operations are performed, respectively, during the sweep line traversal.

![Figure 3.8: Computing the guard zone \( G \) of a simple polygon \( P \).](image)

Here the data structures used are of the highest importance to accomplish the task in \( O(n \log n) \) time. This algorithm maintains two binary search trees, one for storing the event points and another for the line segments to keep track of their neighbouring information, which is known as the query tree. All the three operations are performed on the query tree. As an intersection occurs only between the neighbours, this data structure reduces the search space for intersection detection to the set of neighbours of a line
segment. In addition, it is proved that the line sweep algorithm reports all the intersection points [51]. After getting all these intersection points, we traverse the original guard zone and depending on the intersection points we exclude the intersected or overlapped region(s), and report the outer and inner guard zones [55], accordingly.

### 3.3.3 Experimental Results

In this section, we cite a relative study based on the polygon shown in Figure 3.8, in terms of time and detecting intersections, between the naïve algorithm where the intersections are identified by checking each pair of segments and our proposed method on the line sweep algorithm after repetitive bisections of (external) convex polygonal angles replacing circular arcs by collections of smaller straight line segments.

**Table 3.1:** A study of comparisons between different guard zone computation algorithms for intersection detection among guard zonal straight line segments; the reference simple polygon is shown in Figure 3.8.

<table>
<thead>
<tr>
<th>Variables to be considered</th>
<th>Naive algorithm</th>
<th>Repetitive bisection and line sweep algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of edges of the polygon</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Number of convex vertices of the polygon</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Segments to be considered for checking intersections</td>
<td>14 (line segments) + 9 (circular arcs)</td>
<td>14 (original guard zonal line segments) + 9(p) (derived line segments from the circular arcs), where (p) is a constant</td>
</tr>
<tr>
<td>Computation required</td>
<td>((14+9)^2 = 529)</td>
<td>((23+I) \times (\log_2 23) \approx 104.042 + 4.524 \times I), where (I) is the number of intersections</td>
</tr>
<tr>
<td>Time complexity</td>
<td>(O(n^2))</td>
<td>(O(n \log n + I \log n))</td>
</tr>
</tbody>
</table>

Next, we study the contrast of detecting intersection between the naïve algorithm, where all pairs of guard zonal segments are considered and our proposed algorithm, where
first we draw the guard zone of a simple polygon using the process depicted in algorithm $Guard\_Zone$ and then apply the above method of recursive angular bisection, as shown in Table 3.1. Inevitably, our proposed method outperforms the other in many ways.

3.3.4 Computational Complexity

If the number of edges in the given polygon is $n$, then the number of edges after the preprocessing of the polygon by replacing all the circular arcs by a set of (smaller) line segments is also $O(n)$ as we require a constant number of operations for each of the convex vertices. The next step is to apply the line sweep algorithm taking the set of all line segments, original as well as derived segments by preprocessing. The algorithm starts by creating the event queue by sorting the starting and ending points of the line segments, which takes $O(n \log n)$ time using any standard sorting algorithm. Initializing the status structure takes constant time. The handling of status structure consists of three operations, insertion, deletion, and interchange of positions that takes $O(\log n)$ time each. Now, $m = n+I$, where $I$ is the number of intersection points. Hence, the complexity of line sweep algorithm is $O(m \log n)$ [51, 52].

We have already mentioned that the number of intersection points, $I = O(n)$ in the worst case. This is because the line segments as part of $G$ are not arbitrary line segments on a plane; they are placed one after another, and two adjacent line segments intersect at a guard zonal vertex of $G$ that is not a member of $I$, as the guard zone has been computed and elucidated above, for a given simple polygon $P$ of $n$ polygonal vertices (and $n$ polygonal edges). Here the computation of identifying the points of intersection is essential for non-adjacent guard zonal line segments only. Hence, the identification of all desired points of intersection of the guard zonal line segments computed for $G$ and thus excluding of overlapped regions formed by three or more guard zonal segments can be performed in time $O(n \log n)$. We conclude the result in the following theorem.

**Theorem 3.1:** Algorithm $2D\_Guard\_Zone$ computes a guard zone of a simple convex polygon of $n$ vertices in $\Theta(n)$ time. The worst case computational complexity of algorithm $2D\_Guard\_Zone$ is $O(n \log n)$, in the case of the simple non-convex polygon of $n$ vertices. This is due to the fact that, the running time of computing guard zone of each polygonal
component, either a vertex or an edge, takes constant time, and that of detecting the intersections among different portions of the computed guard zone is $O(n \log n)$.

In some cases, it may be optimal when each pair of line segments really intersects. Our objective is to have an algorithm that is faster in some situations where the number of intersecting line segments is considerably less than the total number of line segments. Line sweep algorithm [50] is such an algorithm whose running time depends not only on the number of segments in the input but also on the number of intersection points. For this reason, this algorithm is known as output sensitive algorithm, or, in this case, we may call it intersection-sensitive algorithm because the number of intersections determines the size of the output, or in other words, running time of the algorithm. Proceeding in this way we can use the line sweep algorithm as each of the circular arcs has been replaced by a set of line segments, taking all the line segments that are essentially in guard zone and which have been derived.

We yet do not know which pair of a circular arc(s) and/or line segment(s) may intersect. If we can somehow detect the probable intersecting regions, the number of checking can be reduced and thus we may apply the above procedure to a smaller set of regions that are really probable to intersect. In the next section, we develop an algorithm that searches the probable intersection region first and then detects original intersections depending on the information obtained in the first phase. However, our objective is to develop a new algorithm to solve the guard zone computation problem always keeping in mind the dictum of simplicity.

3.4. A Two-Phase Algorithm to Detect and Exclude the Overlapped Regions in Guard Zone

In the previous section, we develop an algorithm to detect and exclude the overlapping (if any) among the guard zonal regions, where we have used line sweep algorithm after recursively subdividing the circular regions up to an extent and then replacing the arc segments by straight line segments. The algorithm detects all the intersections (if any) and accordingly reports the resulting guard zone after removing the overlapping [50].
The computation procedure becomes more elegant if we somehow detect the probable intersection region(s) or the components that are most prone to overlap and afterwards we detect the final intersecting guard zonal component pair(s). Also, it may happen that originally there were no intersections, but the algorithm reports the intersection points and works only with that intersection information and finally reports the computed guard zone. The algorithm works in two phases. In the first phase, it detects the prone guard zonal components to be intersected and in the second phase, it computes the guard zone eliminating overlapping portions.

### 3.4.1 Development of the Two-Phase Algorithm

The main objective of this algorithm is to reduce the search space such that the applied line sweep algorithm becomes speedy. We notice that for each of the circular arcs, its two tangents, which are actually the line segments of the guard zone attached to that circular arc, can be extended. Thus, they meet at a point. If we apply this at each such circular arc, we get \(s\) number of points for \(s\) such circular arcs. As a result, the guard zone becomes a polygon that is not necessarily a simple one. We call it the extended or overestimated guard zone. By converting a guard zone into an extended or overestimated guard zone, we now do not have to consider the circular arc portions for repetitive bisection.

![Figure 3.9: Extending the guard zonal line segments \((u'v'\text{ and } w'v'')\) associated with a circular arc \((v'v'')\) to obtain the overestimated guard zonal region.](image)

For an example, the vertex \(p\) of the overestimated polygon, as shown in Figure 3.9, is formed by extending two neighbouring line segments of the circular guard zonal
component of the convex vertex \( v \) of the given polygon. Here the circular arc \( v'v'' \) is the guard zonal region of the convex vertex \( v \). Now the neighbours of this circular arc are \( u'v' \) and \( v''w' \). They are extended and meet at \( p \) that is considered to be a convex vertex of the overestimated polygon. Thus, all the circular arcs of the guard zone are now replaced by corresponding convex vertices of the extended polygon (assumed as an overestimated guard zone).

Now, if the extended guard zone that is actually a polygon also becomes a simple polygon, it means that there is no intersection among the guard zonal components, and there is no need for the second phase of the algorithm. Here the line sweep algorithm is applied where the input is a set of line segments (original guard zonal line segments and the derived line segments replacing the circular arcs) associated with their starting and ending coordinates as event points. The starting point, ending point, and point of intersection (if any) is the three types of event points. As usual, all the event points are sorted, and the sweep line is traversed through the sorted list of event points. At each event point, insertion (at start event point), deletion (at end event point), and update (at intersection point) of neighbouring operations are performed during the sweep line traversal. Here the data structures used are of the highest importance to accomplish the task in \( O(n \log n) \) time.

This algorithm maintains two binary search trees, one for storing the event points and another for the line segments to keep track of their neighbouring information. All the three operations are performed on the query tree. As an intersection occurs only between the neighbours, this data structure reduces the search space for intersection checking to the set of neighbours of a line segment. Furthermore, it is proved that line sweep algorithm reports all the intersection points [5]. After getting all these intersection points, we traverse the original guard zone and depending on the intersection points we exclude the intersected or overlapped region(s), and report the outer and inner guard zone [55], accordingly.

When applying line sweep algorithm [5] to a polygon, we need to remember that each pair of consecutive edges share one event point (starting and/or ending), and thus the algorithm needs a necessary modification such that an even point based on starting and/or ending of two adjacent polygonal edges are taken care of. At the beginning of the
algorithm, the sweep line is at the maximum or minimum event point; again, that point is the starting point of two line segments and hence arises the ambiguity in selecting the root of the query tree as both the line segments are claimers to be the root. We remove this confusion in the following way; let us consider $L1$ and $L2$ are such line segments competing to be the root.

First, we compare the end points of the line segments. If their y-coordinates (i.e. values) are different, the line segment having larger y-coordinate (say $L1$) is selected as the root, and another line segment (i.e. $L2$) has been selected as its left (right) child if the x-coordinate of $L2$ is less (greater) than that of $L1$. On the other hand, if their y-coordinates are same, the line segment having smaller x-coordinate (say $L1$) is selected as the root, and another line segment (i.e. $L2$) has been selected as its right child.

Another modification of line sweep algorithm lies in the fact that we have to deal with a closed polygon, and the query tree is empty only before starting and after termination of the algorithm, whereas the traditional line sweep algorithm might have an empty query tree before its termination. This may so happen when line segments are clustered, and one cluster is placed apart from a second one. Except for the above-cited modifications, the algorithm proceeds in the way of traditional line sweep algorithm identifying all the intersection points, if they exist [5].

Now we start with the second phase of the algorithm. In this phase, the algorithm deals with the part of the original guard zonal regions (comprising circular arcs) that are detected to be the probable or suspected regions of intersection by the first phase of the algorithm. As for example, any two line segments that are part of the extended guard zone may overlap and must be replaced by actual portions of guard zonal line segments and circular arcs in the second phase. Now, to detect the real intersections among the original guard zonal segments of the given polygon, we need a further checking by considering only the suspected segments or regions acquired from the first phase of the algorithm. Algorithm $Two\_Phase\_Intersection\_Detection$ which is a slightly modified version of the $Bentley\-Ottoman$ algorithm as discussed in [5], is depicted stepwise as follows.
Algorithm Two_Phase_Intersection_Detection

Input: A set $S$ of line segments in the plane consisting of both polygonal and guard zonal edges.

Output: The set of intersection points of the segments in $S$, with for each intersection point the segments that contain it (excluding the polygonal and guard zonal vertices connecting consecutive edges for the input polygon and the computed guard zone).

Step 1: Initialize an empty event queue $Q$. Next, insert the segment endpoints into $Q$; when an upper endpoint is inserted in $Q$, the corresponding segment should be stored with it.

Step 2: Initialize an empty status structure (query tree) $T$.

Step 3: While $Q$ is not empty do

Step 3.1: Determine the next event point $p$ in $Q$ and delete it.

Step 3.2: Process_Eventpoint($p$)

End while

Procedure Process_Eventpoint($p$)

Step 1: Let $U(p)$ be the set of segments whose upper endpoint is $p$; these segments are stored with the event point $p$. (For horizontal segments, the upper endpoint is by definition the left endpoint.)

Step 2: Find all segments stored in $T$ that contain $p$; they are adjacent in $T$. Let $L(p)$ denote the subset of segments found whose lower endpoint is $p$, and let $C(p)$ denote the subset of segments found that contain $p$ in their interior.

Step 3: If $L(p) \cup U(p) \cup C(p)$ contains more than one segment, then

Step 3.1: If $C(p)$ is Null AND cardinality $(L(p) \cup U(p)) = 2$ AND $(L(p) \cup U(p))$ contains either two consecutive polygonal edges OR two consecutive guard zonal edges, then

Step 3.2: Do not report $p$ as an intersection point, since $p$ is merely connecting two polygonal or guard zonal edges.
Else

Step 4: Report $p$ as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.

Step 5: Delete the segments in $L(p) \cup C(p)$ from $T$.

Step 6: Insert the segments in $U(p) \cup C(p)$ into $T$. The order of the segments in $T$ should correspond to the order in which they are intersected by a sweep line just below $p$.

If there is a horizontal segment, it comes last in all segments containing $p$.

Step 7: Deleting and re-inserting the segments of $C(p)$ reverses their order.

End if

Step 8: If $U(p) \cup C(p)$ is Null, then

Step 8.1: Let $s_l$ and $s_r$ be the left and right neighbours of $p$ in $T$.

Step 8.2: Search_Newevent($s_l, s_r, p$)

Else

Step 9: Let $s'$ be the leftmost segment of $U(p) \cup C(p)$ in $T$.

Step 9.1: Let $s_l$ be the left neighbour of $s'$ in $T$.

Step 9.2: Search_Newevent($s_l, s', p$)

Step 9.3: Let $s''$ be the rightmost segment of $U(p) \cup C(p)$ in $T$.

Step 9.4: Let $s_r$ be the right neighbour of $s''$ in $T$.

Step 9.5: Search_Newevent($s'', s_r, p$)

End if

Procedure Search_Newevent($s_l, s_r, p$)

Step 1: If $s_l$ and $s_r$ intersect below the sweep line, or on it and to the right of the current event point $p$, and the intersection is not yet present as an event in $Q$, then

Step 1.1: Insert the intersection point as an event into $Q$.

End if

We have already said that the suspected regions are composed of a number of line segments and circular arcs. Thus, in the second phase, we like to deal only with the guard zonal segments that are intersected (with their extended / overestimated information) in the first phase of the algorithm. Now we find the real portions of the guard zones that intersect, after replacing each circular arc by a collection of smaller line segments and applying the
line sweep algorithm over these smaller line segments and the remaining suspected guard zonal line segments. At the end of the second phase, we get the original intersections and depending on these points, the algorithm reports the outer guard zone as well as the inner guard zonal loop, if any. Two phases of the algorithm are now discussed with the help of a suitable example.

Let us consider a simple polygon $M$ whose vertices are stored in anticlockwise order as $a, b, c, d, e, f, g, h, i, j, k, l$, where $a, b, c, d, e, f, g, k$ are convex and $h, i, j$ are concave vertices (see Figure 3.10(a)). After computing the extended guard zone of this polygon, the algorithm is applied to detect and exclude the overlapping portions.

**Figure 3.10:** (a) A simple polygon $M$. (b) The extended guard zone $N$ of polygon $M$ (dotted polygon is $M$, and $N$ is the extended guard zonal polygon drawn by solid lines).

3.4.1.1 The First Phase of the Algorithm

As the pair of neighbouring line segments of each circular arc (guard zone of a convex vertex) is extended, they meet at a point that is again a convex vertex of the overestimated polygon. Now let us consider a overestimated polygon $N$ as shown in Figure 3.10(b), whose edges are labeled as $AB(2), BC(3), CD(4), DE(5), EF(6), FG(7), GH(8), HI(9), IJ(10), JK(11), KL(12), and LA(1)$. After sorting all the event points in the non-ascending $y$-value, we get the event list as $L, E, A, F, K, J, D, G, H, I, B, C$. At the beginning, this array only contains all the start and end points in their sorted sequence (according to their $y$-
coordinates), but subsequently checking of intersection introduces more event points as the intersection points are also considered as the event points. All the event points are handled by a query tree $T$, as explained next.

**Figure 3.11:**
(a) 1 and 12 have been inserted, 12 being the right child of 1, as the end point of 12 is at right side to that of 1.  
(b) 1 and 12 have been inserted, 12 being the left child of 1 and the resulting tree might be the alternative to the tree of (a).  
(c) 5 and 6 have to be inserted; 6 is inserted first as the end point of 6 is to the left of 5. After inserting 6 as the right child of 12, the tree becomes imbalanced.  
(d) The imbalanced tree in (c) is made height balanced through AVL rotation.  
(e) 5 is inserted as the right child of 6.  
(f) 1 is deleted, and 2 is inserted; 2 being the left child of 12.  
(g) The positions of 12 and 6 have been interchanged at the event point that is the intersection point of 12 and 6.  
(h) 6 is deleted, and 7 is inserted at the position of 6.  
(i) 12 is deleted, and 11 is inserted at the position of 12.  
(j) The positions of 7 and 11 have been interchanged.  
(k) 11 is deleted, and 10 is inserted at the position of 11.  
(l) The positions of 7 and 10 have been interchanged at the intersection point of 7 and 11.  
(m) 5 is deleted, and 4 is inserted at the position of 5.  
(n) 7 is deleted, and 8 is inserted at the position of 7.
Figure 3.11 (contd.): (o) The positions of 8 and 10 have been interchanged at their associated intersection point. (p) 8 is deleted, and 9 is inserted at the position of 8. (q) 9 and 10 are deleted resulting 4 to be the new root. (r) 9 and 10 are deleted resulting 2 to be the new root and the tree in (r) is selected as the x-coordinate of the end point of 2 is less than that of 4, though both of them have same y-coordinate. (s) 2 is deleted, and 3 is inserted at the position of 2.

Now the sweep line is adjusted, parallel to the x-axis at the maximum y-coordinate, i.e. at L. It is the starting point of two line segments 1 and 12. The x-coordinate of the end of 1 is less than the x-coordinate of the end of 12. Thus, in T, 1 is made the left neighbour of 12. Accordingly, this relationship is conveyed by both the trees as shown in Figures 3.11(a) and 3.11(b). Though at a glance, it seems to be ambiguous to choose the tree structure between these two, this does not introduce any indecisiveness in our algorithm.

We can always choose the root by comparing the y-coordinates of the two line segments whose start points are same. The line segment, whose end point is of greater y-coordinate between the two, is selected as the root. Now the other line segment is inserted into the tree as the right child or the left child of the root depending on the neighbouring relationship of this segment with the root. Here, 1 becomes the root and 12 becomes its right child as the y-coordinate of A, which is the end point of segment 1, is higher than that of K (i.e. the y-coordinate of A is greater than that of K), which is the end point of segment 12. Thus, the tree of Figure 3.11(b) is chosen as the desired one. Now 1 and 12 are two consecutive polygonal edges, and there is no need for checking their point of intersection. L is deleted from the event list. Hence, now the content of the event queue becomes E, A, F, K, J, D, G, H, I, B, C.

Now E becomes the next event point. It is the starting point of 6 and 5; hence, these two event points are to be inserted into the query tree. As E is a point with higher x-
coordinate than that of \(L\), 6 and 5 are inserted into the right subtree of 12. Here we compare the end points of 6 and 5 to make a decision regarding the immediate neighbour of 12. As the end point of 6 possesses lower x-coordinate than that of 5, 6 is inserted first as the right child of 12 as shown in Figure 3.11(c).

Now we observe that the tree we get is not a height balanced tree; thus, to make it an AVL tree, the necessary rotation is applied to it, and the final (AVL) tree we obtain is as shown in Figure 3.11(d). Now, checking for intersections between 6 and 12 is performed. It results in an intersection point \(Q\). The point is inserted into the intersection list with information \(Q(6,12)\) maintaining its order. Again, the intersection point \(Q\) is inserted into the event queue maintaining proper y-value sequence. Now, 5 becomes the right neighbour of 6. Accordingly, the tree is shown in Figure 3.11(e). \(E\) is deleted from the event queue, and now the content of event queue becomes \(A, Q, F, K, J, D, G, H, I, B, C\).

Now the sweep line moves towards the next event point \(A\), which is the end point of 1 and start point of 2. As 1 is a leaf node in \(T\) and the remaining tree does not become height imbalanced even after deleting the node \(A\), it is deleted without any additional alteration in the tree. As 1 has been deleted and \(A\) is of less x-coordinate than the point at which 12 cuts the sweep line, 2 is inserted, being the left neighbour of 12. Accordingly, the tree is shown in Figure 3.11(f). \(A\) is deleted from the event list. The event list is now \(Q, F, K, J, D, G, H, I, B, C\). Thus, now the event point, \(Q\) is to be handled. As it is an intersection point, the neighbouring information is updated in \(T\). This is performed by interchanging 12 and 6, and this, in turn, interchanges their set of neighbours; the tree is shown in Figure 3.11(g). Next, \(Q\) is deleted from the event list. Now, the event list becomes \(F, K, J, D, G, H, I, B, C\). Thus, during the execution of the algorithm the sweep line traverses all the points in the event list. In the meantime, it also handles the query tree \(T\). In Figure3.11, all the intermediate trees are shown. As we are applying the line sweep algorithm on a bounded region, i.e. a polygon, the query tree cannot be empty until all the event points are processed. Hence, the first phase of the algorithm is over when the query tree becomes empty. Table 3.2 shows the state of the intersection list and the event queue at each event point.
Table 3.2: Successive consideration of all the event points, and the contents of event queue and intersection list after processing of associated event points during the execution of the algorithm.

<table>
<thead>
<tr>
<th>Event points</th>
<th>Start line segment(s)</th>
<th>End line segment(s)</th>
<th>Intersecting line segments</th>
<th>Event queue</th>
<th>Intersection list</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>1, 12</td>
<td>-</td>
<td>-</td>
<td>E, A, F, K, J, D, G, H, I, B.</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>5, 6</td>
<td>-</td>
<td>-</td>
<td>A, Q, F, K, J, D, G, H, I, B.</td>
<td>Q(6,12)</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>Q, F, K, J, D, G, H, I, B, C</td>
<td>Q(6,12)</td>
</tr>
<tr>
<td>Q</td>
<td>-</td>
<td>-</td>
<td>6, 12</td>
<td>F, K, J, D, G, H, I, B, C</td>
<td>Q(6,12)</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>6</td>
<td>-</td>
<td>K, J, D, G, H, I, B, C</td>
<td>Q(6,12)</td>
</tr>
<tr>
<td>K</td>
<td>11</td>
<td>12</td>
<td>-</td>
<td>P, J, D, G, H, I, B, C</td>
<td>Q(6,12), P(7,11)</td>
</tr>
<tr>
<td>P</td>
<td>-</td>
<td>-</td>
<td>7, 11</td>
<td>J, D, G, H, I, B, C</td>
<td>Q(6,12), P(7,11)</td>
</tr>
<tr>
<td>J</td>
<td>10</td>
<td>11</td>
<td>-</td>
<td>D, R, G, H, I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10)</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
<td>-</td>
<td>R, G, H, I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10)</td>
</tr>
<tr>
<td>R</td>
<td>-</td>
<td>-</td>
<td>7, 10</td>
<td>G, H, I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10)</td>
</tr>
<tr>
<td>G</td>
<td>8</td>
<td>7</td>
<td>-</td>
<td>S, H, I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
<tr>
<td>S</td>
<td>-</td>
<td>-</td>
<td>7, 8</td>
<td>H, I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
<tr>
<td>H</td>
<td>9</td>
<td>8</td>
<td>-</td>
<td>I, B, C</td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
<tr>
<td>I</td>
<td>-</td>
<td>10, 9</td>
<td>-</td>
<td>B, C</td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>C</td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>3, 4</td>
<td>-</td>
<td></td>
<td>Q(6,12), P(7,11), R(7,10), S(7,8)</td>
</tr>
</tbody>
</table>

3.4.1.2 The Second Phase of the Algorithm

At the end of the first phase of the algorithm, we have acquired all the probable intersection points for the overestimated/extended guard zonal regions. Each intersection point can be on the actual guard zonal components or the extended portion of the guard zonal line segments. So, an intersection point thus obtained may be original or fake depending on
whether it is on the original guard zonal segment (or not), and thus, a further checking is required to identify the real ones. In doing so, for each such point, we have to check for intersection among the original guard zonal components (line segments and circular arcs) depending on the fact whether a (polygonal) line segment joins two convex/concave vertices or one concave vertex and one convex vertex. If it joins two concave vertices, then merely the guard zonal line segment (if it exists) of the polygonal edge is to be considered.

At this time for each circular arc, as described earlier, we perform repetitive bisection of a convex polygonal angle to convert each (guard zonal) arc into a collection of smaller line segments, and discuss that even when the convex angle tends to 360° we have to subdivide only 180° as we exclude two 90° angles before starting the subdivision. Thus, the subdivision of an arc \( p \) times results in \( p+1 \) sub-arcs (or smaller line segments) in a sequence, where two consecutive line segments share their starting and ending points on the guard zone. Therefore, the number of event points is \( p+2 \) for each of the circular arcs.

In our example (see Figure 3.10(b)), in the intersection list the first intersection point is \( Q \), which is in between two line segments 6 and 12 of the extended guard zone. Line 6 joins two convex points \( F \) and \( E \). Thus, we have to consider both the circular arcs corresponding to \( E \) and \( F \) and the line segment joining the allied circular arcs. Again, 12 also joins two convex points \( L \) and \( K \), and we consider both the circular arcs associated with \( L \) and \( K \) and the line segment joining these two circular arcs. Thus, we have four circular arcs and two line segments for a probable intersecting region that we have obtained from the first phase of the algorithm. At the second phase, after subdividing the arcs, we get a total of \( 4(p+1)+2 \) line segments as inputs to the line sweep algorithm. After applying the line sweep algorithm to the said input, no intersection has been detected among the original guard zonal components.

Now we discuss what the algorithm does when dealing with the intersection points considered from the example given. The intersection point \( P \) is obtained from the guard zonal segment 7 and 11. Line 7 joins two convex points \( F \) and \( G \). So, we have to consider both the circular arcs corresponding to \( F \) and \( G \) and the line segment joining these circular arcs. Again, vertex 11 joins a convex point \( K \) and a concave point \( J \); so we consider the
circular arc corresponding to $K$ and the line segment joining the circular arc and the concave point $J$. Hence, we have three circular arcs and two line segments for a probable intersecting region. At the second phase after subdividing the arcs, we get total $3(p+1)+2$ line segments for the line sweep algorithm. After applying the line sweep algorithm on the above-mentioned line segments, we find that actually there is no intersection among the original guard zonal components.

For intersection point $R$ that is between 7 and 10, it can be noticed that 7 joins two convex points $F$ and $G$. So, we consider both the circular arcs corresponding to $F$ and $G$ and the line segment joining these circular arcs. Again, the point 10 joins two concave points $J$ and $I$; hence, we consider only the line segment joining the concave points $J$ and $I$. Now we have two circular arcs and two line segments for a probable intersecting region. At the second phase after repetitive bisection, we get total $2(p+1)+2$ line segments as input for the line sweep algorithm. After applying the line sweep algorithm on the aforesaid segments, we find that there is one intersection point between the circular arc of the original guard zone (including overlapping) and the line segment $JI$. Let the point of intersection be $Y$ (as shown in Figure 3.12).

We denote each intersection point as a 3-tuple (intersection point, (two ends of one of the intersecting component), (two ends of another intersecting component)). Thus, for the intersection point $Y$, guard zonal information is $\langle Y, \text{arc}(G_1,G_2), (I,J) \rangle$. Hence, the guard zonal component list is updated by replacing $\text{arc}(G_1,G_2)$ by $\langle \text{arc}(G_1,Y), \text{arc}(Y,G_2) \rangle$ and $IJ$ by $\langle IY, YJ \rangle$.

The next intersection point is $S$, which is between line segments 8 and 10. 8 joins one convex point $G$ and one concave point $H$. So, we consider the circular arcs related to $G$ and the line segment joining the circular arc and the concave point $H$. Again, 10 joins two concave points $J$ and $I$; thus, we consider only the line segment joining the concave points $J$ and $I$. Hence, we have one circular arc and two line segments for a probable intersecting region. At the second phase after subdividing the arcs, we get a total of $(p+1) + 2$ line segments for the line sweep algorithm. After applying the line sweep algorithm on the abovementioned line segments, we find that in fact there is one intersection between the line segment joining the circular arc corresponding to $G$, the concave point $H$ and the
segment \(JI\). Let the point be \(X\), and the information that supposed to be updated in the guard zonal component list is \(\langle X, (G_2, H), (IJ)\rangle\). Now at the time of updating the guard zonal component list, as \(IJ\) has already been updated, we need to check whether the point \(X\) lies on \(IY\) or \(YJ\). Accordingly, the guard zonal information regarding this component is updated in the list. Here, for point \(X\), we update \(IY\) as \(IX\) and \(XY\), as \(X\) is on \(IY\).

After getting all the guard zonal information, the algorithm reports the computed guard zone excluding the overlapped portions. Thus, the output is reported in the form of a list of guard zonal line segments and guard zonal circular (arc) segments after removing intersecting region(s), if any. Now, here is an important observation; in the notch area, the overlapping among the components may create two types of guard zonal loops: outer guard zone and inner guard zone. For the latter case, it is free space and not a part of the polygon. Hence, detection of the inner guard zonal loop is of utmost importance for further utilization of this free space. However, both the loops are free from overlapping.

For the example given, as we traverse the guard zonal components in a counterclockwise manner, we can have each component in order. Thus, before updating the list, we need to make sure that the list must be in the following form: \(\text{arc}(A_1,A_2), A_2B_1, \text{arc}(B_1,B_2), B_2C_1, \text{arc}(C_1,C_2), C_2D_1, \text{arc}(D_1,D_2), D_2E_1, \text{arc}(E_1,E_2), E_2F_1, \text{arc}(F_1,F_2), F_2G_1, \text{arc}(G_1,G_2), G_2H, HI, IJ, JK_1, \text{arc}(K_1,K_2), K_2L_1, \text{arc}(L_1,L_2), L_2A_1\).

Now, as we have updated every such guard zonal component that has been identified for the intersection, the traversal also shows the new guard zonal component list. As our objective is to determine (excluding the overlapped portions) the guard zone, outer guard zonal loop and inner guard zonal loop, we focus only on the intersection points. According to the list, as we traverse the guard zone, we encounter the intersection points one after another, as it appears on the list. Once we encounter an intersection point, we change our path and continue the traversal through the other line segment in an anticlockwise manner. We can find the next line segment or arc following a counterclockwise direction, as the original guard zonal information in the form of a list is already stored. Thus, when we reach the starting point of the traversal, our job is completed. This process takes \(O(n)\) time if the number of vertices in the polygon is \(n\). As per our example, the said traversal results the outer guard zone as follows: \(\text{arc}(A_1,A_2), A_2B_1, \text{arc}(B_1,B_2),\)
$B2C1$, arc$(C1,C2)$, $C2D1$, arc$(D1,D2)$, $D2E1$, arc$(E1,E2)$, $E2F1$, arc$(F1,F2)$, $F2Y$, $YJ$, $JK1$, arc$(K1,K2)$, $K2L1$, arc$(L1,L2)$, $L2A1$, as we find it in Figure 3.12.

**Figure 3.12:** Original guard zone of a simple polygon consisting of line segments and circular arcs.

In this case, when we arrive at $Y$ after traversing segment $F2Y$, we check for the line segment that intersects at $Y$ other than $F1G2$. Here it is $IJ$, and the immediate next point from $Y$ in anticlockwise direction is $J$. So, we move to $J$ and report the segment $YJ$ as the next traversed line segment in the desired guard zone excluding the overlapped regions. Sometimes, there might have an overlapping at the notch region, and there is sufficient space inside the notch to place a subcircuit to utilize the area more efficiently. In this case, if we follow the above procedure we may fail to find out such a region that is detached from the guard zone and form a separate loop inside the notch. In such a scenario, we follow the procedure discussed below.

We start traversing the guard zonal line segments and circular arcs as said above. When we are at one of the intersection points, we traverse anticlockwise manner enlisting the line segments and circular (arc) segments including the intersection points. Thus, the list starting from one intersection point and circle back to the same point needs to be eliminated from the guard zone as it includes one (or more) loop(s) along with the intersection region. Then we get the resultant list for the guard zone. The loop can also be specified by sublist(s) of the initial list mentioned above. If there is one such cycle starting
from one intersection point and ending at the same point without having any other intersection point in between, then it assures that a loop is formed.

For the example polygon (in Figure 3.10(a)), we have obtained two 3-tuples \( \langle Y, (\text{arc}(F1,F2), \text{arc}(G1,G2)), (I,J) \rangle \) and \( \langle X, (\text{arc}(G1,G2), H), (I,J) \rangle \). Starting from \( \text{arc}(F1,F2) \) we get the list: \( \text{arc}(F1,F2), F2Y, YG1, \text{arc}(G1,G2), G2X, XH, HI, IX, XY, YJ \) as it covers all the end points of this tuple, and it is updated in the original guard zonal list. However, here is no inner guard zone starting from \( Y \) and ending at \( Y \) because in this cycle there is another intersection point \( X \). Before updating the original list, we remove the sublist starting from \( Y \) and ending at the point \( (X) \) on the line segment joining two intersection points \( (X \text{ and } Y) \). Thus, here we remove the following portion: \( YG1, \text{arc}(G1,G2), G2X, XH, HI, IX, XY, YJ \).

Starting from the \( \text{arc}(G1,G2) \) we get the list: \( \text{arc}(G1,G2), G2X, XH, HI, IX, XY, YJ \). Hence, the loop comprises the list \( XH, HI, IX \), and the outer guard zone is formed by \( \text{arc}(A1,A2), A2B1, \text{arc}(B1,B2), B2C1, \text{arc}(C1,C2), C2D1, \text{arc}(D1,D2), D2E1, \text{arc}(E1,E2), E2F1, \text{arc}(F1,F2), F2Y, YJ, JK1, \text{arc}(K1,K2), K2L1, \text{arc}(L1,L2), L2A1 \).

Therefore, at the beginning of the algorithm the list contains all the line segments and arcs without any prior knowledge of intersection region(s), if any, and after application of the algorithm, the original intersection points are identified, if they exist, and sublists are computed comprising points (over the line segment(s) and/or circular arc(s)) for the desired guard zone as well as loops.

At the time of reporting the guard zone excluding the overlapped regions, starting from the end point as listed in one of the intersection tuples, the computed guard zone is traversed anticlockwise enlisting the line segments and circular (arc) segments including the intersection points. Thus, the list starting from one intersection point and ending at the same point is to be eliminated from the guard zone as it includes the inner guard zone and intersection region. The resultant list for the guard zone is thus obtained. The inner guard zone can also be specified by sub-list of the above-said list. If there is one such cycle starting from one intersection point and ending at that point without having any other intersection point within it, it is the inner guard zone.

After finding the list of intersecting line segments and arcs, the information is updated in the original guard zonal list. There is no inner guard zone starting from one
intersection point (Y) and returning to that very intersection point (Y) if in this cycle there is no other intersection point X, where X ≠ Y; otherwise, there is an inner guard zone that needs to be reported separately. If there is any inner guard zone, before updating the original list the sub-list starting from Y and ending at the line segment joining two intersection points X, Y is removed. The remaining list is reported as the outer guard zone. Thus, we obtain the outer guard zone as well as the holes in the guard zone (if any) in O(n log n) time.

In this section, we have developed an algorithm for computing the guard zone of a given simple polygon, whose computational complexity has not been measured now. This is because the algorithm devised herein is tuned with a small but very important modification in the following section, thereafter we have computed the necessary time and space complexities of the algorithm and concluded with an associated theorem.

3.4.2 Computational Complexity of Two_Phase_Intersection_Detection

First of all, the event queue Q is initialized with all the segment endpoints (both starting and ending). The construction of the event queue Q takes O(n log n) time, due to the fact that the event queue has been built using balanced binary search tree data structure. The initialization of status structure or query tree T takes constant time. Each event is being handled by performing at most three operations on the event queue Q. The algorithm first deletes an event from the queue before any further processing related to that event by making a call to the procedure, Process_Evenpoint, whereas Search_Newevent is called once or twice, which results in the insertion of at most two new events into Q. The deletion and insertion operation on Q takes at most O(log n) time each. The operations performed on status structure / query tree T are insertions, deletions, and neighbour finding, each type of operation takes O(log n) time. If the cardinality of (L(p) ∪ U(p) ∪ C(p)) is denoted by m(p), which also represents the number of segments involved with the event, is linear with the number of operations performed. Now, if the sum of all m(p), over all the event points p is denoted by m, then the running time of the algorithm becomes O(m log n).

Clearly, m = O(n + k), where the size of the output is represented as k; whenever m(p) > 1, the algorithm reports all the segments in the event, the events involving only one segment are the end points of the segments. Now the intention is to prove m = O(n + l),
where \( l \) represents the number of intersection points. To illustrate this, interpret the set of segments as a planar graph embedded in the plane. The vertices of the planar graph are the end points and the intersection points of the segments, and its edges are the pieces of the segments connecting vertices.

Consider \( p \) is an event point, which is a vertex of the graph, and \( m(p) \) is bounded by the degree of the vertex. Therefore, \( m \) is bounded by the sum of the degrees of all vertices of the planar graph. Each edge of the graph contributes one each to the degree of exactly two vertices (as endpoints), so \( m \) is bounded by \( 2ne \), where \( ne \) represents the number of edges of the graph. Consider bounding \( ne \) in terms of both \( n \) and \( I \). Now by definition, the number of vertices \( nv \) cannot exceed \( 2n + I \). Considering the well-known fact regarding planar graphs, which is \( ne = O(nv) \), in turn, proves the claim. Now for the sake of completeness of the proof, consider the fact that every face of the planar graph is bounded by at least three edges, provided that there are at least three segments, whereas an edge can bind at most two different faces. Hence the number of faces \( nf \), cannot exceed \( 2ne/3 \). By applying Euler’s formula, which states that for any planar graph with \( nv \) vertices, \( ne \) edges, and \( nf \) faces, the following relation holds:

\[
_nv - ne + nf \geq 2.
\]

Equality holds if and only if the graph is connected. Plugging the bounds on \( nv \) and \( nf \) into this formula, we get

\[
2 \leq (2n + I) - ne + (2ne/3) = (2n + I) - ne/3.
\]

Thus, \( ne \leq 6n + 3I - 6 \), and \( m \leq 12n + 6I - 12 \), and the bound on the running time follows.

### 3.5 Enhancement over the Methods of Overestimation

#### 3.5.1 Objective of the Problem

In the two-phase algorithm as has been devised above, we derive the overestimated guard zone by extending the guard zonal line segments. This procedure finds all the intersection point(s) and reports the overlapped region(s) reducing the search space and meets our objective. There are some situations where we may find that this approach fails as the overestimation varies significantly with the angle at that convex vertex. According to the procedure of derivation of the overestimation, guard zonal region at a convex angle that
tends to $180^\circ$ has very small overestimation while at a convex angle tends to $0^\circ$ has an infinitely large overestimation. Hence, in the first phase of the algorithm, the extended part of a small convex vertex may result in a huge number of fake intersection points, i.e. some regions may be identified as intersection prone region while the corresponding guard zonal components have no chances for overlap.

Let us consider the case as depicted in Figure 3.13, where $NOPQRSTUV$ is a portion of a polygon and $N'O'P'Q'R'S'T'U'V'$ denotes its extended guard zone. As $\angle NOP$ tends to $0^\circ$, overestimation leads to a problem. Here $OO'$ is much larger than $r$, the width of the guard zone. Thus, there may be a greater chance to overlap with other extended guard zonal regions and thereby may produce a larger number of intersection points, which are ultimately proved to be fake intersection points in the second phase of the algorithm. Hence, in the second phase of the algorithm, we have to deal with a large number of intersection points unnecessarily which in turn enhances redundant computation, and thus the effectuality of the algorithm is reduced.

**Figure 3.13:** Extension of guard zone may report a large number of fake intersections.

This problem leads us to develop a more refined approach towards overestimation. In fact, such a situation can be avoided if we restrict the degree of overestimation to a certain limit for all convex vertices. In doing so, we try to incorporate each circular arc and its neighbouring line segments within the individual polygonal region, thereby reducing the amount of overestimation (based on a given value of $r$). In the following section, we address a little but significant modification of the algorithm which further speeds up the defined task.
3.5.2 Development of the Enhanced Two-Phase Algorithm

As we like to reduce the number of fake intersection points, the overestimation must be in a smallest possible region that covers the guard zone with only line segments [51]. Now, we take the extended guard zone instead of the original guard zone, and that is derived in the following way. For each convex angle we draw the bisector, say $bs$, and at a distance $r$ from the convex vertex outside the polygon we draw a perpendicular line that intersects the extended guard zonal line segments of polygonal adjacent edges $UV$ and $VW$, at points $A$ and $B$, respectively. Hence, $V'ABV''$ becomes a reduced overestimation of the circular arc $V'PV''$, as shown in Figure 3.14.

![Diagram of extended guard zone](https://via.placeholder.com/150)

**Figure 3.14:** $AB$ is the perpendicular on angle bisector $bs$ at point $P$, where the bisector cuts the circular arc; $V'ABV''$ is an overestimation over the circular arc segment.

**Algorithm Enhanced_Guard_Zone**

**Input:** A simple polygon $P$.

**Output:** A guard zone of polygon $P$ consisting of only line segments and represented as a set of vertices $\{p_1, p_2, \ldots, p_n\}$ that are the end points of the consecutive edges.

**Step 1:** Clockwise label the vertices $v_1, v_2, \ldots, v_n$ of polygon $P$.

**Step 2:** For $i = 1$ to $n-1$ do

**Step 2.1:** If the external angle at $v_i$ is convex, then
Step 2.1.1: Draw an angle bisector of the convex angle and a perpendicular to the
bisector, say $s$, at a distance $r$ from $v_i$ outside the polygon.

Step 2.1.2: Find the external angle at $v_{i+1}$, and consider polygonal edge $(v_i,v_{i+1})$.

Step 2.1.3: If the external angle at $v_{i+1}$ is convex, then

Step 2.1.3.1: Draw an angle bisector of this angle and a perpendicular to the
bisector, say $t$, at a distance $r$ from $v_{i+1}$ outside the polygon.

Step 2.1.3.2: Draw a line parallel to $(v_i,v_{i+1})$ at a distance $r$ apart from the
polygonal edge (outside the polygon) that intersects both $s$ and $t$ at $p_i$ and
$p_{i+1}$, respectively. Report $p_i$ and $p_{i+1}$.

Else

Step 2.1.3.3: Bisect the (internal) angle at $v_{i+1}$, denote the bisection $b_{s_{i+1}}$.

Step 2.1.3.4: Draw a line parallel to $(v_i,v_{i+1})$ at a distance $r$ apart from the
polygonal edge (outside the polygon) that intersects $s$ at $p_i$ and $b_{s_{i+1}}$ at a
point, say $p_{i+1}$. Report $p_i$ and $p_{i+1}$.

Step 2.1.4: Assign $i \leftarrow i+1$.

Else

Step 2.1.6: Bisect the (internal) angle at $v_i$, denote the bisection $b_{s_i}$.

Step 2.1.7: Find the external angle at vertex $v_{i+1}$, and consider polygonal edge
$(v_i,v_{i+1})$.

Step 2.1.8: If the external angle at $v_{i+1}$ is convex, then

Step 2.1.8.1: Draw an angle bisector of this angle and a perpendicular to the
bisector, say $t$, at a distance $r$ from $v_{i+1}$ outside the polygon.

Step 2.1.8.2: Draw a line parallel to $(v_i,v_{i+1})$ at a distance $r$ apart from the
polygonal edge (outside the polygon) that intersects $b_{s_i}$ at a point, say $p_i$ and
intersects $t$ at $p_{i+1}$. Report $p_i$ and $p_{i+1}$.

Else

Step 2.1.8.3: Do Step 2.1.3.3 and get the bisector $b_{s_{i+1}}$.

Step 2.1.8.4: Draw a line parallel to $(v_i,v_{i+1})$ at a distance $r$ apart from the
polygonal edge (outside the polygon) that intersects $b_{s_i}$ at a point, say $p_i$ and
also intersects $b_{s_{i+1}}$ at a point, say $p_{i+1}$. Report $p_i$ and $p_{i+1}$.

Step 2.1.9: Assign $v_i \leftarrow v_{i+1}$.
Step 2.1.10: If $v_i = v_n$, then $v_{i+1} = v_1$.

End for

Now, we get the guard zone of the given polygon comprising only a set of line segments instead of circular arcs which then solves the difficulty that was detected in the previous version of the algorithm while deriving the extended guard zone from the original guard zone. Now, the modified line sweep algorithm (for a polygon) is applied where the input is a set of line segments (extended guard zonal line segments $UA$ and $UB$, and the derived line segments $AB$ replacing the circular arcs) associated with their starting and ending coordinates as event points. The sweep line is traversed through the sorted list of event points depending on x- or y-coordinates. Here starting point, ending point, and point of intersection (if any) is the three types of event points at which insertion, deletion, and update of neighbouring operations are performed, during the sweep line traversal. As discussed earlier, the data structures used to accomplish this task is of utmost importance and ensures the running time to be bounded by $O(n \log n)$ time.

Let us consider the case depicted in Figure 3.15, where we observe that the previous algorithm reports a number of fake intersections in the first phase of the algorithm. On the other hand, in this version of the algorithm, we restrict the overestimation as has been discussed above and as a consequence, the number of fake intersections reported in the first phase is significantly reduced. Hence, the algorithm reduces the number of operational overhead in the second phase, which does not introduce any change in the asymptotic running time of the algorithm, but definitely enhances its efficacy.

Figure 3.15: The new method of extending guard zone at each convex vertex of a polygon improves the effectiveness of the algorithm by reducing the overhead of fake intersections.
We are to cite the algorithm now through an example instance. Let us consider the simple polygon \( P \) as depicted in Figure 3.16, whose vertices are \( A \) through \( N \) while traversing \( P \) in anticlockwise direction and its corresponding guard zone is \( G \). Now we draw the derived line segments using the method described above and the extended guard zone becomes a set of line segments only instead of a set of line segments and circular arcs. Hence, the arc for the convex vertex \( A \) is replaced by the line segment \( A'A'' \) and so on (where all the dotted segments in Figure 3.16 are the derived segments). Now we are able to apply the line sweep algorithm taking this set of all line segments in \( G \) as input, i.e. the coordinates of the start and end points of all the line segments, known a priori. The extended guard zone becomes a polygon, which may be simple or not. If this is a simple polygon, i.e. there is no intersection of the edges of this extended one, the original guard zone (to be computed) must not have any overlapping. Hence, the first phase of the algorithm searches for the regions that might intersect in \( G \) in a similar way that has been discussed in the previous section.

**Lemma 3.3:** Enhanced_Guard_Zone results in a guard zone \((G)\) of a polygon \((P)\) where there are no two points \( A \) and \( B \), such that \( A \in P \) and \( B \in G \), and the distance between \( A \) and \( B \) is less than \( r \), the width of the guard zone.

**Proof:** We prove this lemma through coordinate geometry point of view. In this context, we refer to Figure 3.14, where \( V \) denotes a convex vertex belonging to the given polygon, for which we draw the guard zonal segments. As \( V''W' \) and \( U'V' \) are two line segments that have been drawn parallel to the adjacent edges at \( V \) at a normal distance \( r \) from the polygonal edges, this lemma holds true for the polygonal points and any point on \( V''W' \) and \( U'V' \). Now consider \( p \) as a point on the angle bisector \( bs \) and at a distance \( r \) from \( V \). A perpendicular is drawn on the bisector through point \( p \) and the adjacent edges of the concerned convex vertex are extended. As a consequence, two intersection points are achieved between each of the parallel guard zonal edges and the perpendicular line drawn on the bisector.

If we imagine a circle of radius \( r \) taking \( V \) as the center, \( AB \) becomes a tangent to the circle at point \( p \). Hence, all the points on \( AP, BP, V''B \), and \( V'A \) are at a distance greater
than \( r \) from \( V \) (except points \( P, V'', \) and \( V' \), where \( VP = VV'' = VV' = r \)), and hence proves the lemma.

\[ \square \]

**Figure 3.16:** Polygon \( P \) with extended guard zone having solid lines as parts of the desired guard zonal segments and dotted line segments as the derived line segments after execution of the first phase of the algorithm.

### 3.5.3 Computational Complexity

If the number of edges in the given polygon is \( n \), then the number of edges in the overestimated polygon is also \( O(n) \). The algorithm starts by constructing the event queue by sorting the start and end points of the line segments, which takes \( O(n \log n) \) time. Initializing the status structure takes constant time, i.e. initially the event queue as well as the BST is made empty. The handling of event queue consists of three operations, insertion, deletion, and interchange of positions, which takes \( O(\log n) \) time each. Now consider, \( m = n + I \), where \( I \) is the number of intersection points for the polygon of \( n \) edges. The complexity of the line sweep algorithm becomes \( O(m \log_2 n) \) [5].

Again, as the number of intersections is \( I \), the maximum number of line segments that take part in the line sweep algorithm in the second phase is \( 4(p+1)+2 \) for each case, where \( p \) is the number of iterations required to perform the bisection of a convex polygonal vertex. Thus, the complexity becomes \( O(p \log_2 p) \), and including the number of intersection points (i.e. \( I \)), it becomes \( O(IP \log_2 p) \). Hence, for both the phases of the algorithm, the
overall computational complexity turns into \( O(n \log_2 n + Ip \log_2 p \log_2 n) \). Now as \( p \) is a predefined constant (and independent of \( n \)), \( p \log_2 p \) is also constant, and hence we conclude that the overall running time of the newly devised guard zone computation algorithm is \( O(n \log_2 n + cI \log_2 n) \), where \( c = p \log_2 p \).

Now to analyze the amount of storage used by the algorithm, we first consider the tree \( T \) that stores a segment at most once during the first phase of the algorithm and hence uses \( O(n) \) storage space. The size of the event queue is bounded by \( O(n+I) \) [50]. During the second phase, the algorithm uses the constant (in terms of \( p \)) amount of storage for every intersection point detected in the first phase if the algorithm and hence the overall storage requirement for the newly devised algorithm is \( O(n+I) \), which is linear. We now conclude the computational complexities of the algorithm devised for computing a guard zone of a given simple polygon in the form of the following theorem.

**Theorem 3.2:** If the number of edges in the given simple polygon is \( n \), the Enhanced Two-Phase Algorithm computes a guard zone of the polygon in \( O(n \log_2 n + cI \log_2 n) \) time, where the number of intersections is \( I \), \( c = p \log_2 p \), and \( p \) is the number of iterations required to perform the bisection of a convex polygonal vertex in its second phase. Moreover, the overall storage requirement for the algorithm is \( O(n+I) \).

### 3.6 The Bounded Box Algorithm for Detecting and Eliminating Guard Zonal Overlapping

#### 3.6.1 Development of Algorithm 2D_Bounded_Box

In the previous section, we have developed an algorithm that detects and removes overlapping among the guard zonal regions (if any) using line sweep algorithm with necessary modifications [51]. In all of the previously discussed algorithms, our job was to detect the suspected guard zonal regions, such that the operational overhead gets reduced by confining the search procedure within the regions. Again, the effectiveness of all the algorithms is solely dependent on this phase (of finding out suspected overlapped guard zonal regions); as we are more tightly bound the targeted regions, the unnecessary searching operation is also reduced.
Figure 3.17: (a) Bounded box for a (guard zonal) line segment parallel to neither x-axis nor y-axis. (b) Bounded box for a (guard zonal) line segment parallel to the x-axis. (c) Bounded box for a (guard zonal) line segment parallel to the y-axis.

On the other hand, all the algorithms are supposed to report all the suspected regions and ensure that no such regions left undetected. Towards our continuous goal of reducing the search space for intersection detection, we are going to introduce one more version of the algorithm for computing the desired guard zone. In this new algorithm, our goal is to divide the guard zonal region into a set of sub-regions, not necessarily disjoint, each containing one of the guard zonal segments (either line segment or circular arc) and to transform our purpose in finding the overlapped sub-regions among the derived sub-regions. Again, all the sub-regions need to be orthogonal, i.e. these are rectangular in shape, and the sides are either parallel to x-axis or y-axis. We call each such sub-region a bounded box or a bounded rectangle.

Now, we would like to formulate and develop this new algorithm towards solving the guard zone computation problem for a simple polygon. There are two issues to be considered: construction of the rectangles or bounded boxes such that their edges are either horizontal or vertical (with respect to a coordinate frame) and detection of overlapping among the bounded boxes. In the first phase itself, we deal with the construction of the bounded boxes and the detection of overlapping among the bounded boxes. The overlapping detection procedure remains transparent of the type of segments contained within each box. After finding the probable overlapping, we deal with the original segments to check for the actual overlapping. Subsequent discussions reveal the method, i.e. how we
construct a bounded box and detect the overlapping between two or more such bounded boxes.

To locate a bounded box, we find the maximum and the minimum for both x- and y-coordinates of a segment. We draw a pair of horizontal lines through the x- and y-coordinates for their individual minimum and maximum values for a guard zonal component under consideration. Thus, the four pairs of orthogonal straight lines cut each other and result in the bounded rectangle as shown in Figure 3.17(a). \( AB \) denotes a (guard zonal) line segment, for which two vertical and two horizontal lines are drawn, and those lines form a rectangle bounding the guard zonal segment. For line segments other than the ones parallel to x- or y-axis, it is straightforward to construct the bounded box. However, when the line segment, i.e. the guard zonal line segment becomes parallel to either x- or y-axis, the above procedure fails to make the line segment bounded within a rectangle. For each such case, we consider a constant \( r \), the width of the guard zone, and find their corresponding bounded rectangle as follows.

When the line segment is parallel to x-axis, two horizontal lines are drawn at \( r \) unit above and below the segment and two vertical lines are drawn at two end points, which result in a bounded box of height \( 2r \) as shown in Figure 3.17(b), where \( AB \) is a horizontal guard zonal edge (or component or line segment) and the dotted rectangle is the bounded box of height \( 2r \). Similarly, we can find the bounded box for a line parallel to y-axis by drawing two vertical lines on both left and right at \( r \) unit apart from the guard zonal line segment \( AB \) under consideration, as shown in Figure 3.17(c).

For circular arcs, finding bounded box is tricky as the end points of a circular arc may not always denote its max or min points. Thus, to find max and min of x and y, we proceed as follows. At first, two vertical straight lines are drawn at two end points of the polygonal edge \( AB \), which results in three different cases.

**Case I:** Two drawn lines do not cut the corresponding circle of the circular arc at points except the end points. For this case, these end points are the maximum and minimum values of x, as shown in Figure 3.18(a). Let \( ACB \) is the circular arc, where \( A \) and \( B \) be the two end points and the vertical lines drawn through \( A \) and \( B \) do not intersect any other point
of the associated circle. Hence, point A and point B are the coordinates with the maximum and minimum x values, respectively, as \( x_A \) is greater than \( x_B \) in terms of the x value only.

**Case II:** Both the drawn lines cut the corresponding circle of the circular arc at another point except the end points. We have to check whether the point lies on the concerned circular guard zonal segment or the remaining portions of the circle. From Figure 3.18(b), we find that if the endpoints are A and B, we draw vertical lines through A and B that cut the circle at D and C, respectively. Now to check whether C and D are on the circular guard zonal segment, we have to measure the angles \( \angle ACB \) and \( \angle ADB \), respectively. If the angle is an acute angle, the corresponding point does not lie on the circular guard zonal segment.

**Figure 3.18:** Finding maximum and minimum x- and y-coordinates of a circular arc. (a) End-points are the maximum and minimum x-coordinate points. (b) Some other points are maximum and/or minimum x-coordinate points. (c) End-points are the maximum and minimum y-coordinate points. (d) Some other points are maximum and/or minimum y-coordinate points.
Figure 3.19: Finding the maximum and minimum $x$- and $y$-coordinates of a circular arc. (a) One of the end points ($B$) is the minimum $x$-coordinate point. (b) One of the end points ($B$) is the maximum $x$-coordinate point. (c) One of the end points ($B$) is the maximum $y$-coordinate point. (d) One of the end points ($B$) is the minimum $x$-coordinate point.

On the other hand, if the angle is an obtuse angle, the associated point lies on the circular guard zonal segment. In Figure 3.18(b), $C$ is outside the circular arc while $D$ is on the arc. Hence, if the point lies outside the arc, the point through which the vertical line is drawn is an extreme (i.e. minimum or maximum) point of that segment; else, the extreme point lies on the circular segment other than the end points. In that case, we find this extreme point(s) directly by using some basic geometry as follows. Provided with the coordinate of its center, say $(a,b)$, and the radius, which is equal to the width of the guard zone, i.e. $r$, the coordinates of the points on the corresponding circle, possessing the maximum and minimum $x$ values are $(a+r,b)$ and $(a-r,b)$, respectively, while the coordinates of the points with the maximum and minimum $y$ values are $(a,b+r)$ and $(a,b-r)$, respectively.

Case III: One of the newly drawn lines cuts the related circle of the circular arc at another point except for the endpoints. To check whether the point lies on the circular
segment we consider the remaining portion of the circle, we follow the method as discussed in Case II, and accordingly we get the extreme points using essential basic geometry [8], i.e. minimum and maximum points as shown in Figures 3.19(a) and 3.19(b), where \( ADB \) is the circular segment to be considered and the vertical line through \( A \) cut the circle at \( C \), while the vertical line through \( B \) does not cut the circle at any other point. Hence, \( B \) is the minimum (in Figure 3.19(a)) and maximum (in Figure 3.19(b)) in the \( x \) value of the assumed circular arc.

To find extreme \( y \) points, we follow the above-mentioned cases considering the drawn lines to be horizontal instead of vertical as depicted in Figures 3.18(c), 3.18(d), 3.19(c), and 3.19(d). Thus, we obtain all the bounded boxes in terms of their maximum and minimum \( x \) and \( y \) values with reference to the coordinates on each circular arc and line segment of the guard zone. Each box is stored as a 5-tuple, \( (S, \text{max } x, \text{min } x, \text{max } y, \text{min } y) \), where \( S \) denotes the segment for which the box is stored.

Whenever the first phase is over, we have the bounded boxes and our objective is to find overlapping among the boxes, if any. Now each box contains a single guard zonal segment, either circular or linear. We may visualize the problem as there are only \( O(n) \) boxes and any pair of boxes may overlap. Again, if there are overlapping between two boxes, we cannot say with certainty that the corresponding guard zonal segments intersect. The information of bounded box overlapping means that there is a high probability of intersection between the corresponding guard zonal segments. Thus, our task is to retrieve the overlapping information of the bounded boxes and then check whether the guard zonal segments contained within the boxes, also intersect.

Now two boxes overlap only if there are overlapping between their \( x \)-span and \( y \)-span (in terms of their \( x \)- and \( y \)-coordinates). We index the boxes with natural numbers starting from the box containing the component of the leftmost or uppermost polygonal vertex. As a box is bounded by two pairs of horizontal and vertical line segments, each box may be identified by one horizontal line (i.e. \( h \) line) and one vertical line (i.e. \( v \) line). For detection of overlapping along \( x \)-span (\( y \)-span), the starting and ending point of the \( h \) line (\( v \) line) are considered to be the event points. All the event points are sorted on the basis of \( x \)-coordinate for \( h \) lines and \( y \)-coordinate for \( v \) lines in isolation.
Figure 3.20: (a) $h$ lines of the matching boxes in the search space. (b) $v$ lines of the associated boxes in the search space.

Now we describe how an overlapping (if any) along $x$-span as well as $y$-span is identified. The problem may be considered to be an instance of finding overlap among a set of parallel line segments. To solve this, an imaginary line is swept starting from the minimum $x$- (maximum $y$-) coordinate for the $h$ line ($v$ line) and the information of overlapping are recorded. Here also we use binary search tree (BST) data structure to handle the information throughout as we have done in the case of line sweep algorithm.

The BST is updated according to the event points faced by the sweep line. One $h$ line ($v$ line) is inserted into the tree at the starting point of a line and deleted at its ending point. We get the information whether the newly entered line has any overlapping along $x$-span ($y$-span) with all the existing $h$ line(s) ($v$ line(s)) in the tree. Thus, we obtain two separate overlapping lists for $x$-span and $y$-span. If a pair of lines (i.e. two $h$ lines in $x$-span overlapping list and two $v$ lines in the $y$-span overlapping list) is obtained in both the lists, the allied bounded boxes definitely overlap. This method may be visualized as there are two sets $A$ and $B$, and we need to find $A \cap B$. We can use the hash data structure to accomplish this in linear time as all the boxes, i.e. the $h$ lines and $v$ lines are indexed by
natural number for further references. Let us illustrate the procedure of finding the $h$ lines (or $v$ lines), whose x-spans or y-spans are overlapping.

![Figure 3.21](image)

**Figure 3.21:** (a) 1 is inserted into the empty tree, as it appears first when the sweep line moves from left to right (see Figure 3.20(a)). (b) 2 is inserted making it the right child of 1, as 2 starts next and it is on the right to 1. (c) 3 is inserted as the right child of 2 in a similar manner; thus, the tree becomes imbalanced. (d) Rotation is applied to make the tree in (c) a height-balanced (AVL) tree. (e) 1 is deleted, as it terminates as the sweep line moves to the right. (f) 4 is inserted making the right child of 3 again resulting an imbalanced tree. (g) Appropriate rotation is performed to make the tree in (f) a balanced one. (h) 3 is deleted. (i) 2 is deleted. (j) 4 is deleted as it terminates, but the sweep line moves further as all spans are yet to traverse; next 5 is inserted into the tree making it as the root of the tree.

In Figure 3.20(a), there are $5h$ lines that are denoted by natural numbers and the sorted event points are 1s, 2s, 3s, 1e, 4s, 3e, 2e, 4e, 5s, 5e. The sweep line is initially placed at the minimum x-coordinate, and it is moved through the event points until the maximum x-coordinate point is reached. At each starting event point, the associated $h$ line is inserted into the BST, and it is considered to be overlapped with all the existing $h$ lines in the BST. Thus, the overlapping information is updated, and one $h$ line is deleted from the tree while its end point is reached. At each event point, the BST and the updated overlapping information are shown in Figures 3.21(a) through 3.21(j).

At each insertion of an $h$ line, the overlapping list is updated depending on the newly inserted $h$ line and existing nodes that represent the other $h$ lines whose end points
have not yet been visited. Thus, when the first h line is inserted into the empty tree, there is no overlapping information. When the next line is inserted into the tree, we traverse the tree in an inorder manner and get the overlapped information between a pair of h (or v) lines, here the first pair is (1,2) for h lines, which is updated in the overlapping list (see Figures 3.20(a) and 3.21(b)). For each newly inserted line, we have a number of such pairs that are entered into the list holding overlapping information for the corresponding span (x or y). After completion of the traversal of the sweep line, we get all the overlapping information of pairs of h lines along an x direction; thus the set of overlapping pairs we obtain is \( A = \{(1,2), (1,3), (2,3), (2,4), (3,4)\} \). Similarly, taking the v lines and traversing the sweep line along y direction we get the set \( B = \{(1,3), (1,4), (1,5), (3,4), (3,5), (4,5)\} \).

After performing the intersection between sets \( A \) and \( B \), we get the actual pairs for which the bounded boxes overlap. In this case \( A \cap B = \{(1, 3), (3, 4)\} \) indicating the overlapping between boxes 1 and 3, and between boxes 3 and 4; thus, there is a possibility of overlapping between these pairs of associated guard zonal segments only.

The boxes that are overlapped are only considered for further processing where the segments like line-line, or line-circle, or circle-circle contained within the overlapped boxes are now deemed for intersection detection. In the case of the line-line intersection, we directly get the intersection point, if any, using simple coordinate geometry. However, when one of the segments is circular, the intersection is checked with the associated circle(s). If there is any intersection point, we further check whether the point is on the circular guard zonal segment using the logic described at the beginning of this section. The bounded boxes for two consecutive (disjoint) segments (that are either line-line or line-circular segments) are not considered to be overlapped. If we obtain such type of information in the first phase, we can straightway ignore them as they are successive, and there is no need to check for any intersection between them.

After getting all these intersection points, we traverse the original guard zone and depending on the intersection points we exclude the intersected or overlapped region(s), and report the outer guard zone and guard zonal holes [55] if they occur. Thus, the phase terminates after checking all the probable intersection points and reporting of the intersection point(s), if any.
Now we are about to explain our algorithm with the help of an example as shown in Figure 3.22, where we consider a portion of the guard zone of a simple polygon. Though the polygon is simple, the guard zone may not be a simple one (as its segments may cross each other). The guard zonal segments are traversed starting from $A$ through $J$, and bounded boxes are obtained one after the other as the segments appear. Afterwards, by following the steps of the algorithm, we detect the overlapped boxes. As per this example, the boxes containing circular arcs $BC$ and $FG$, and each of the bounded boxes containing guard zonal segments $DE$, $EF$, and $FG$ with the box containing such a line segment $CD$ overlap. Now we like to identify whether the guard zonal segments of the detected pairs of bounded boxes truly overlap. Thus, in the next phase, we check the corresponding guard zonal segments for their mutual intersection if any.

We may further note that for a pair of successive guard zonal segments there might have any overlapping, but as they are consecutive, there must not be any overlapping between the associated segments. As for example, the bounded boxes for line segments $CD$ and $DE$ overlap, while the bounded boxes for $DE$ and $EF$ do not. In both of these cases, there is no question of overlapping of guard zonal segments. Hence, for the only pair(s) of non-adjacent guard zonal segments, if any, we execute this phase of our algorithm to identify particular overlapping of guard zones.

**Figure 3.22:** A portion of the guard zone of a simple polygon and the bounded boxes of the guard zonal segments. Here guard zonal segments $BC$ and $FG$ intersect at $B'$. 
Here for the example guard zone under consideration, as shown in Figure 3.22, we are supposed to take care of only three pairs of bounded boxes, that are \((BC, FG)\), \((CD, FG)\), and \((CD, EF)\), which are non-adjacent to each other. As \(BC\) and \(FG\) are circular guard zonal segments, we check the associated circles if they intersect. Indeed, these two segments intersect at \(B'\), as shown in the figure. If we find intersection point(s), the point(s) is (are) further checked whether they lie on the concerned circular segment(s), or they lie on the remaining portion of the believed circles containing the circular arcs \(BC\) and \(FG\). Here \(B'\) is a point that lies on the said circular arcs. On the other hand, \(F\) is a point on \(CD\); thus, the intersection point is common to both the guard zonal line segment \(EF\) as well as guard zonal circular arc \(FG\).

Hence, the intersection points are detected to identify the loop(s) like \(FDEF\), and compute the desired guard zone traversing the points … \(ABB'GHIJ\) … after deleting the portions between the points of intersection \(B'\) and \(F\) (that are \(B'CF\) and \(FB'\)) excluding the points of intersection (rather, points of crossing \(B'\) and \(F\) of guard zonal segments), as has been shown in Figure 3.22.

### 3.6.2 Computational Complexity of Algorithm 2D\_Bounded\_Box

In order to calculate the computational complexity of the algorithm, we consider that the number of vertices in a simple polygon for which the guard zone has to be computed is \(n\). The algorithm begins by computing the guard zone of a simple polygon using the method as cited in [54] in linear time. The guard zone that computed in such a way may have overlapped regions. We then compute all the bounded boxes for all individual guard zonal segments (linear/circular). Finding out each bounded box requires constant time as for a segment to bind in a box we need to find only its extreme points using fundamental geometry as has been discussed in the algorithm development section. So, for \(n\) boxes, this phase requires \(O(n)\) time. Next, to detect overlapping among the bounded boxes (if any), the algorithm requires a preprocessing, where each end point of \(h\) lines, as well as \(v\) lines, are sorted in \(O(n \log n)\) time separately.

As we are using BST to detect the overlapping among the boxes, each insertion, update, and deletion operation take \(O(\log n)\) time, and for \(n\) boxes, it runs in \(O(n \log n)\)
time. When all the overlapping information among boxes is obtained, then checking for intersection among guard zonal segments requires $O(I)$ time, if $O(I)$ pairs for $O(I)$ bounded boxes are overlapped in the first phase. We traverse the original guard zone and depending on the intersection points, we exclude the overlapped region(s) and report the guard zone in $O(n)$ time [54]. Hence, the algorithm computes the guard zone and also reports the overlapped regions, if any, in $O(n \log n + I)$ time. The proposed algorithm is an output sensitive algorithm as the complexity depends not only on the number of vertices of the given input polygon but also on the number of the intersection points. For a simple polygon and a small $r$, the width of the guard zone, $I$ is not beyond $O(n)$. Hence, the algorithm detects and excludes all the intersections among the guard zonal segments and reports it in $O(n \log n)$ time [52]. We conclude this result based on the identification and exclusion of overlapped guard zonal regions for a given simple polygon in the form of a theorem as follows.

**Theorem 3.3:** Algorithm 2D_Bounded_Box computes a guard zone of a simple polygon with $n$ vertices in $O(n \log n + I)$ time, where $I$ is the number of overlapped bounded box pairs that are detected in the second phase of the algorithm. For a simple polygon and a small value of $r$, the width of the guard zone, $I$ is not beyond $O(n)$; hence, the overall time complexity of algorithm 2D_Bounded_Box becomes $O(n \log n)$ including the detection of intersections among different portions of the computed guard zone.

### 3.6.3 Experimental Results

In this section, we cite a relative study based on the polygon shown in Figure 3.8, in terms of time complexity for intersection detection, between the naive algorithm where the intersections are identified by checking each pair of segments and our proposed bounded box method where the first phase identifies the probable intersecting segments through the detection of overlapping among the bounded boxes and the second phase deals with the original guard zonal segments to find the actual intersection points. Here we denote the number of probable intersections resulted in the first phase as $I$. Note that the assumed polygon in Figure 3.8 containing 14 edges and 14 vertices out of which nine vertices are convex. A time sensitive comparative study has been included in Table 3.3.
### Table 3.3: A comparative study of various intersection detection methods.

<table>
<thead>
<tr>
<th>Variables to be considered</th>
<th>Naïve algorithm</th>
<th>Bounded Box based algorithm for detecting and excluding of overlapped guard zonal segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segments to be considered for checking intersections</td>
<td>14 (line segments) + 9 (circular arcs) = 23</td>
<td>23 bounded boxes, i.e. 23 horizontal line segments and 23 vertical line segments</td>
</tr>
<tr>
<td>Time complexity</td>
<td>O(n²)</td>
<td>O(n) (for computing the guard zone without excluding the overlapped region) + O(n) (for finding the bounded boxes) + O(n log n) (for sorting the starting and ending points of the h lines and v lines) + O(n log n) (for detecting the overlapped bounded boxes) + O(I) (for checking original guard zonal intersections among I pairs of overlapped bounded boxes) + O(n) (traversing and reporting the guard zone excluding overlapped regions) ≡ O(n log n), as for a given simple polygon I is not beyond O(n).</td>
</tr>
</tbody>
</table>

### 3.7 Summary

As discussed earlier, resizing of electrical circuits is an important problem in VLSI layout design as well as in embedded system design, while accommodating the (groups of) circuit components on a chip floor. This problem motivates us to compute a guard zone of a simple polygon. In this chapter, we have considered the problem of computing guard zone of a (2D) simple polygon and developed a number of sequential algorithms for computing the same that uses the concepts of analytical and coordinate geometry. First, we have developed an algorithm that only computes guard zone of a simple polygon in linear time,
hereby raising the key challenge issue to detect overlapped region(s) within the guard zone (if any) and accordingly exclude that region(s) to report the resulting outer guard zone.

Computing overlapped regions is of utmost importance as the information of the inner guard zonal loop(s) often enhances the placement phase of VLSI physical design automation. These algorithms can be customized in a modular fashion in computing the regions of width $r$ (as guard zonal distance) outside the polygon, and also inside the polygon, if necessary, which may find several applications in practice. In subsequent chapters, we have shown how a 2D guard zone computation algorithm devised in this chapter could be extended for developing its parallel counterpart and designing algorithms for computing the same in the case of 3D solid objects that are available in reality. For the sake of simplicity, we have assumed that the 3D objects are covered by only planar surfaces; thus, the intersection of two such adjacent planes is always a straight line segment. In this thesis, we have also devised sequential as well as parallel algorithms for guard zone computation for three-dimensional (3D) simple solid objects.