Optimally computing a shortest weakly visible line segment inside a simple polygon

Binay K. Bhattacharya a, Gautam Das b, Asish Mukhopadhyay c, Giri Narasimhan d

a School of Computer Science, Simon Fraser Univ., Burnaby, BC, Canada, V5A 1S6
b Microsoft Research, One Microsoft Way, Microsoft Corp., Redmond, WA 98052, USA
c School of Computer Science, University of Windsor, Windsor, ON, Canada, N9B 3P4
d School of Computer Science, Florida International University, Miami, FL 33199, USA

Received 29 November 2000; accepted 24 August 2001
Communicated by S. Suri

Abstract
A simple polygon is said to be weakly internally visible from a line segment lying inside it if every point on the boundary of the polygon is visible from some point on the line segment. In this paper, we present an optimal linear-time algorithm for the following problem: Given a simple polygon, either compute a shortest line segment from which the polygon is weakly internally visible, or report that the polygon is not weakly internally visible.

The algorithm presented is conceptually simple. This paper also incorporates a significant improvement over the linear-time algorithm for the same problem, presented in a preliminary version [12], in the sense that it eliminates the need for using two complicated preprocessing tools: Chazelle’s linear-time triangulation algorithm [7], and the algorithm for computing single-source-shortest-paths from a specified vertex in a triangulated polygon [16], thus making the algorithm practical. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Simple polygons; Weak visibility; Triangulations; Weakly internally visible

A preliminary version of parts of this paper by the second and fourth authors appeared in the Proc. of the 10th ACM Symp. on Computational Geometry, 1994.

E-mail addresses: binay@cs.sfu.ca (B.K. Bhattacharya), GautamD@microsoft.com (G. Das), asishm@cs.uwindsor.ca (A. Mukhopadhyay), giri@cs.fiu.edu (G. Narasimhan).

1 Supported in part by NSF Grant CCR-930-6822; Part of this author’s work was done at The University of Memphis, Memphis, TN 38152, USA.
2 Research supported by NSERC Grant 227693-00.
3 Supported in part by NSF Grants INT-911-5870 and CCR-940-9752; Part of this author’s work was done at The University of Memphis, Memphis, TN 38152, USA.
1. Introduction

Polygonal visibility problems arise naturally in such diverse areas as robotics (path planning, motion planning), computer graphics (hidden-line and hidden-surface removal), image processing (hamiltonian triangulations). The notion has been extant in the mathematical literature [5,29] long before it was introduced into Computational Geometry. Research into the **computational aspects** of visibility was initiated by the well-known art-gallery problem, posed by Klee (see [24]), which is the problem of determining the minimum number of guards sufficient to cover the interior of a polygonal art-gallery.

A visibility problem in its most abstract form can be formulated thus:

Given a scene composed of a finite number of geometrical objects, a viewpoint or a set of viewpoints, and a notion of visibility, compute the scene as viewed.

A concrete example of this abstract formulation is the following: Given a point (i.e., viewpoint) lying inside a simple polygon (where the scene consists of only the polygon), compute the part of the polygon visible from this point [15] (where two points are considered visible if the straight line segment joining them lies entirely within this polygon).

When there is a set of viewpoints (instead of a single viewpoint), the appropriate notion of visibility that is useful is that of **weak visibility**, which was introduced by Avis and Toussaint [2] (they also introduced other kinds of visibility). An object is said to be weakly visible from a set of viewpoints if **every** point of the object is visible from some viewpoint. Weak visibility has received much attention from a number of researchers [1,2,4,8,10,11,13,14,17,18,26,28]; also see the survey article by O’Rourke [25].

This brings us to the notion of interest in this paper, namely that of a weakly visible line segment in the interior of a simple polygon. If we replace a point by a line segment lying inside the polygon we have a set of viewpoints instead. If every point of the polygon is thus visible, it is said to be weakly **internally visible** (**wiv** from now on) from this line segment.

The problem we consider in this paper is to find a shortest internal line segment of a given polygon \( P \) from which it is **wiv** or else report that the polygon is not **wiv**. An appealing reformulation of this problem is in terms of the illumination paradigm: if we think of the line segment as a linear light source, then the problem can be thought of as that of computing the shortest light source that completely illuminates the interior of the polygon, whenever it is possible to do so. A related problem is that of computing the shortest line segment from which the exterior of a simple polygon is weakly visible; an optimal linear-time algorithm was presented for this problem by Bhattacharya et al. [4].

The shortest illuminating line segment in a polygon can also be thought of as the shortest straight line path that a watchman could patrol along in order to watch over a polygonal art gallery. There have been a number of papers on the **shortest watchman tour** problem [9,21]. The algorithm in this paper finds the shortest straight-line watchman tour, if one exists.

In terms of related work, Sack and Suri [26] presented a linear time solution to determine whether a given polygon is **wiv** from any edge of the polygon. Doh and Chwa [14] presented an \( O(n \log n) \)-time algorithm to compute a **wiv** segment, if one exists. Djidjev et al. [13] showed how to find the link center in \( O(n \log n) \) time. Attempts to efficiently compute the shortest **wiv** segment include the one presented in Ke [19]. Aleksandrov et al. [1] pointed out that Ke’s proposed \( O(n \log n) \)-time algorithm [19] was incorrect. In fact, Aleksandrov et al. [1] provide an \( O(n \log n) \)-time algorithm for computing the shortest central link segment; note that a central line segment in a simple polygon specifies to an illuminating line
segment if the polygon is weakly visible from some line segment. In this paper we present an optimal linear time algorithm for computing the shortest illuminating line segment.

An interesting related problem is that of computing a **single** weakly-visible line segment in a simple polygon. This problem is solved by Das et al. [11], who presented a linear-time algorithm for this problem. However, an improved linear-time algorithm due to Bhattacharya and Mukhopadhyay [3] avoids the use of two tools that had rendered the algorithm by Das et al. [11] impractical: (a) the linear-time triangulation algorithm [7], and (b) the linear-time algorithm to compute shortest paths in a triangulated polygon [16,22].

In this paper we combine ideas from Bhattacharya and Mukhopadhyay [3] and from Das and Narasimhan [12] and present a linear-time algorithm to compute the **shortest** weakly-visible segment in a polygon. The algorithm avoids the two tools mentioned above, thus significantly improving the linear-time algorithm for the same problem presented in a preliminary version of this paper by Das and Narasimhan [12].

The results in this paper build on some of our previous work on optimal linear-time algorithms for weak-visibility problems in polygons. The linear-time algorithms for computing **all LR-visible pairs of points** [10] and for computing **all weakly-visible chords** [11] output a mass of information related to visibility within a polygon. Our present algorithm shows how to exploit this wealth of information to answer more interesting questions related to weak visibility in polygons. We achieve our results by studying the structure of **minimal** weakly-visible segments and identifying the bounding chords for such segments. As described later, one of the by-products of our algorithm in this paper is a linear-time algorithm to generate all minimal weakly-visible segments. These techniques were also used in [10] to obtain a linear-time recognition of $L_2$-convexity of simple polygons.

The paper is organized as follows. In the next two sections we introduce all the preliminaries, geometric and otherwise. Section 4 gives an overview of the algorithm. Sections 5, 6, and 7 provide details of the algorithm. In Section 8 we discuss an extension of our algorithm to a slightly more general problem. Finally, we conclude with open problems in the last section.

2. Notations

Let $P$ be a simple polygon on $n$ vertices. We shall denote its interior by $int(P)$ and its boundary by $bdy(P)$. Despite this distinction, we shall sometimes use simply $P$ to refer to a polygon plus its interior. The exact usage should be clear from the context. We also make the usual general position assumptions that no three vertices of $P$ are collinear, and no three of its edges have a common vertex.

The line segment joining two points $x$ and $y$ is denoted by $xy$. Two points $x,y \in P$ are mutually **visible** (or co-visible) if $xy$ lies entirely in $P$. We let $\bar{r}(x,y)$ represent the ray rooted at $x$ towards point $y$. Informally, the ray shot from a point $x \in P$ in direction of point $y$ consists of “shooting” a “bullet” from $x$ towards $y$. The first point where this shot hits $P$ is called the **hit point** of the ray shot.

A **polygonal chain** is a concatenation of line segments. If $x$ and $y$ are points on $bdy(P)$, then $P_{CW}(x,y)$ ($P_{CCW}(x,y)$) is the subchain of $bdy(P)$, obtained by going clockwise (counterclockwise) from $x$ to $y$. Let $v$ be a reflex vertex of $P$. Let $v^−$ and $v^+$ be the vertices that precede and succeed $v$ with respect to a counterclockwise vertex order on $P$. Let $\bar{r}(v^−,v)$ and $\bar{r}(v^+,v)$ when extended meet the polygon again at $v'$ and $v''$, respectively. The subchain $P_{CW}(v,v')$ is called the **clockwise component** of $v$ (see Fig. 1), while $P_{CCW}(v,v'')$ is called the **counterclockwise component** of $v$. Crucial to our algorithm is the concept
of a non-redundant component. A component is redundant if it is a superset of another component. All other components are non-redundant components.

The clockwise component of $v$ also defines a subpolygon called the clockwise $C$-polygon of $v$, which is the subpolygon of $P$ bounded by the polygonal chain $P_{CW}(v, v')$ and the chord $vv'$. The clockwise $C$-polygon of $v$ is shown as a shaded region in Fig. 1. The counterclockwise $C$-polygons are defined in a similar fashion. For a clockwise $C$-polygon of vertex $v$, $v'$ will be referred to as its clockwise endpoint (and $v$ its counterclockwise endpoint). Similarly, for a counterclockwise $C$-polygon of vertex $v$, $v''$ will be referred to as its counterclockwise endpoint (this time, $v$ is its clockwise endpoint). Given a $C$-polygon (or an intersection of a set of $C$-polygons) denoted by $P_A$, its envelope is defined as the convex polygonal chain bounding $P_A$ in the interior of $P$ (except for its endpoints) and connecting points $u$ and $v$, which lie on the boundary of $P$ (and $P_A$). Note that the envelope of the $C$-polygon in Fig. 1 is simply the straight-line segment (chord) $vv'$. A (clockwise or counterclockwise) $C$-polygon of some reflex vertex $v$ is called redundant (non-redundant, respectively) if its corresponding component is redundant (non-redundant, respectively).

Two subsets $X$ and $Y$ of $P$ are said to be weakly visible from each other if every point in $X$ is visible from some point of $Y$, and vice versa. A polygon $P$ is said to be LR-visible with respect to a pair of points $x$ and $y$ on its boundary, if the chains $P_{CW}(x, y)$ and $P_{CCW}(x, y)$ are weakly visible from each other. A polygon is said to be $L_2$-convex if for every pair of points in the polygon, there exists another point from which the first two are visible.

3. Preliminaries

A chord $xy$ of the polygon $P$ is a line segment connecting two visible points $x$ and $y$ on $\text{bdy}(P)$. A weakly-visible chord is a chord from which the polygon is weakly visible. A weakly-visible segment is simply any line segment in $P$ from which $P$ is weakly visible. A minimal weakly-visible segment is a weakly-visible line segment, no subsegment of which is weakly visible from $P$.

In this section we describe some of the geometric properties of a weakly-visible line segment. It was noted in [17] that the family of non-redundant components completely determines LR-visibility of $P$, since a pair of points $s$ and $t$ admits LR-visibility if and only if each non-redundant component of $P$ contains either $s$ or $t$. A similar result from Das et al. [11] states that the family of non-redundant components also determines all weakly-visible chords, since a chord $st$ is weakly-visible if and only if each non-redundant component of $P$ contains either $s$ or $t$. We first prove Lemma 1, which describes a simple property satisfied by all weakly-visible segments in $P$. We then show in Lemma 2 that the family of non-redundant components also determines the family of weakly-visible segments.
Lemma 1. If $P$ is weakly visible from a line segment $l = \overline{uv}$, then the chord $l'$, obtained by extending $l$ in both directions until it hits $\text{bdy}(P)$ is a weakly-visible chord; furthermore, the endpoints of $l'$ form a LR-visible pair of points with respect to $P$.

Proof. The first part is trivial. The second follows from Lemma 5 of [10]. 

Lemma 2. $P$ is weakly visible from a line segment $l (= \overline{uv})$ iff $l$ intersects every non-redundant C-polygon of $P$.

Proof. If $l$ does not intersect a C-polygon, then it cannot see all the points on the edge of $P$ that is used to generate the corresponding component. Hence the only if part is proved.

For the if part, let us assume that there is a point $x$ on $P$ that is not visible from $l$, i.e., all rays shots emanating from $x$ miss $l$. This implies that there exists a ray shot from $x$ that is tangent to $P$ at some vertex $z$ and that brings the ray closest to the one of the endpoints of $l$. But then, there exists a C-polygon associated with the reflex vertex $z$ that does not intersect $l$. A contradiction!

The obvious implication of Lemma 1 is that a polygon has at least one weakly-visible chord iff it has at least one weakly-visible segment and consequently a shortest weakly-visible segment.

Before giving an overview of the algorithm, we describe the peculiar output of the $O(n)$-time algorithm for computing all weakly-visible chords of a polygon (this algorithm is described in [11] and will henceforth be referred to as the chords algorithm), since this algorithm is used by our scheme. The chords algorithm generates $k = O(n)$ pairs of the form $(A_i, B_i)$ along with two linear functions $L_i(x)$ and $R_i(x)$. Each $A_i$ is a subedge of $P$ with all $A_i$'s being disjoint (except at their endpoints) line segments; each $B_i$ is a subchain of $P$ with the $B_i$'s possibly overlapping each other. For a given point $p \in A_i$ every line segment joining $p$ and any point on a specified subchain $B_p \subseteq B_i$ forms a weakly-visible chord. In order to describe this succinctly, a parameter $x \in [0, 1]$ is used. Let $A_i(x)$, for $x \in [0, 1]$, denote the points of $A_i$. Similarly, let $B_i(x)$, for $x \in [0, 1]$, denote the points of $B_i$. For example, $B_i(0)$ and $B_i(1)$ refer to the left and right endpoints of $B_i$, and $A_i(0)$ and $A_i(0.5)$ refer to the left endpoint and the mid point of segment $A_i$. For each value of $x \in [0, 1]$, the linear functions $L_i(x)$ and $R_i(x)$ correspond to the endpoints of the polygonal subchain of $B_i$ which form weakly visible chords with $A_i(x)$. In other words, for each value of $x \in [0, 1]$, the chord joining $A_i(x)$ and $B_i(y)$ is weakly visible for $y \in [L_i(x), R_i(x)]$. It may be helpful to point out that no component has one of its endpoints in the interior of $A_i$, for any $i$.

4. Overview of algorithm

We first compute all the non-redundant components of $P$ or determine that the polygon is not weakly internally visible; this is described in detail in Section 5. If the intersection of the C-polygons corresponding to all the non-redundant components is non-empty, then we stop since the smallest weakly-visible segment is simply a point. Note that finding the intersection of the C-polygons is described in detail as part of another step in Section 6, after which picking an arbitrary point in this region solves the problem. If the polygon is weakly internally visible, with the non-redundant components as input we run the LR-visibility algorithm of Das et al. [10]. The third step of our algorithm is to run the chords algorithm. If the polygon has no weakly-visible chords, then the algorithm stops and declares that there are no weakly-visible segments either.
Henceforth by components we shall mean non-redundant components; similarly, by C-polygons we shall mean the C-polygons corresponding to the non-redundant components. For every $A_i$ output by the chords algorithm, let $\alpha_i$ denote the envelope of the intersection of the C-polygons that contain $A_i$. Let $\beta_i$ denote the envelope due to the intersection of the remaining C-polygons.

It is clear that any line segment of $P$ that touches both $\alpha_i$ and $\beta_i$ for some $i$, must intersect every C-polygon and by Lemma 2 must be a weakly-visible segment. However, the converse is not so obvious. In Lemma 3 below, we establish this for the shortest weakly-visible segment. This vital property is necessary to make our algorithm work in linear time. It is also noteworthy that the components that contain $A_i$ correspond to a subsequence of the sorted (with respect to the order of appearance along the polygon boundary) sequence of components. What Lemma 3 proves is that only such subsequences (and not an arbitrary subset) of non-redundant components need be considered for computing the shortest weakly-visible segment.

**Lemma 3.** If $\overline{st} = l$ is a shortest weakly-visible segment, then $s$ must lie on $\alpha_i$ and $t$ must lie on $\beta_i$, for some $i = 1, \ldots, k$.

**Proof.** Consider the polygon $P$ of Fig. 2, drawn with smooth curves for simplicity. We extend the shortest weakly-visible line segment $l$ to meet $\text{bdy}(P)$ at the points $p$ and $q$. So the chord $l' = \overline{pq}$ contains $l$.

By Lemma 2, the segment $l$ intersects every C-polygon. Thus every C-polygon completely contains either $\overline{ps}$ or $\overline{qt}$. Consider the components corresponding to the C-polygons that contain $\overline{ps}$. From the minimality of $l$, one of them must contain exactly $\overline{ps}$ or equivalently that $s$ must lie on its bounding chord (or envelope). We conclude that $s$ lies on the envelope of the intersection of all the C-polygons that contain $\overline{ps}$. A similar argument proves that $t$ lies on the envelope of the intersection of all the C-polygons that contain $\overline{qt}$. □

The above lemma suggests the following skeleton for our algorithm, which will be refined later. For every $i = 1, \ldots, k$, construct the envelopes $\alpha_i$ and $\beta_i$, and then compute the shortest line segment joining a point on $\alpha_i$ and a point on $\beta_i$. Then compute the shortest of these segments.

Note that since both $\alpha_i$ and $\beta_i$ are convex polygonal chains, computing the shortest line segment connecting them can be computed in time $O(|\alpha_i| + |\beta_i|)$, where $|\alpha_i|$ and $|\beta_i|$ are the lengths of the two chains. However, each of the $|\alpha_i|$ and $|\beta_i|$ could be $O(n)$, and thus performing this computation in a naive fashion could take a total time of $O(n^2)$. Fortunately, in general, there may be considerable overlap between $\alpha_i$ and $\alpha_{i+1}$, as well as between $\beta_i$ and $\beta_{i+1}$. For the $i$th iteration, instead of simply finding the shortest line segment that joins $\alpha_i$ and $\beta_i$, the algorithm finds the shortest line segment that has at least one endpoint on the portion of $\alpha_i$ that is not part of $\alpha_{i+1}$ or on the portion of $\beta_i$ that is not part of $\beta_{i+1}$. The assumption is that the rest of the portions of the two envelopes will be scanned as part of a later iteration. Repetitious scanning of the polygonal chains is thus prevented by delaying the scanning of overlapping portions as much as possible.

In Section 6.1, we precisely characterize how $\alpha_i$ changes to become $\alpha_{i+1}$, and correspondingly, how $\beta_i$ changes to become $\beta_{i+1}$. In Section 6.2, we describe a data structure that stores $\alpha_i$, $i = 1, \ldots, k$, and another identical structure that stores $\beta_i$, $i = 1, \ldots, k$. We also prove that the total size of the union of $\alpha_i$’s and the total size of the union of $\beta_i$’s (for $i = 1, \ldots, k$) is $O(n)$. 


Fig. 2. The endpoints of the shortest weakly-visible segment must lie on the chains $\alpha_i$ and $\beta_i$ for some $1 \leq i \leq k$.

Fig. 3. The shortest line segment connecting $\alpha_i$ and $\beta_i$ may not lie entirely in $P$.

One problem with the skeleton algorithm described above is that the shortest line segment joining $\alpha_i$ and $\beta_i$ for some $i$ may not lie entirely within $P$. In Fig. 3, the segment $st$, which is the shortest illuminating line segment joining $\alpha_i$ and $\beta_i$, does not lie entirely within $P$. This happens because even though the line segment when extended may hit $A_i$, it might not hit $B_i$ because of obstruction from the rest of $P$, i.e., the extended line is not a weakly-visible chord. In this case, if there does exist a weakly-visible chord connecting $A_i$ and $B_i$, then the shortest weakly-visible segment joining $\alpha_i$ and $\beta_i$ would touch a vertex of $P$. In Fig. 3, such a segment is $uv$.

This suggests that our algorithm needs to deal with two main cases. The first case is when the shortest illuminating segment does not touch a vertex of $P$ except possibly at its endpoints; the second case is when it touches a vertex of $P$ in its interior. If the first case occurs, the algorithm briefly described earlier will output the shortest illuminating segment. The details of this case are described in the Section 6. The second case is handled separately in Section 7. The algorithm for the second case is a modification of our earlier algorithm for computing all weakly-visible chords of a polygon [11]. If a weakly-visible segment does not touch a vertex of $P$ it is referred to as a non-tangential weakly-visible segment; otherwise it is referred to as a tangential weakly-visible segment.

By putting all the pieces together, we show a linear-time algorithm to obtain the shortest non-tangential weakly-visible segment, and a linear-time algorithm to compute the shortest tangential weakly-visible segment. The shortest of the two segments is the shortest weakly-visible segment in a polygon, thus giving us the desired algorithm.

5. Computing all non-redundant components

In this section we show how to compute the set of all non-redundant components, $NR$, of $P$, or report that $P$ is not weakly internally visible. In the former case, the set $NR$ is input to the chords algorithm from [11]. We adopt the following nomenclature: we label the vertices of the polygon from $1 \ldots n$ in
counterclockwise order, while an edge whose endpoints are \( i \) and \( i + 1 \) (mod \( n \)) is labeled \( i \). With reflex vertex \( i \), we maintain the ordered triplet \((i, j, k)\), where \( j \) and \( k \) are respectively the labels of the edges that are hit by ray shots \( \tilde{r}(i-i) \) and along \( \tilde{r}(i+i) \). We use the special symbol \( \text{null} \) in place of \( j \) or \( k \) if the corresponding component has been identified to be redundant. Initially, \( j \) and \( k \) are set to \( \text{null} \) for all the reflex vertices.

We also use the concept of a critical polygon, which is defined as a subpolygon of \( P \) enveloped by a chord and the boundary of \( P \) that does not wholly contain a component and that every weakly visible segment must penetrate. Note that every non-redundant C-polygon of \( P \) is critical; however, the converse is not true as can be seen from the simple example of Fig. 4 in which the shaded subpolygon is critical, but is not a C-polygon.

The overview of the algorithm in this section is as follows:

**Step 1** Split the boundary of \( P \) into two, three or four polygonal chains (denoted by \( C_1, C_2, C_3 \) and \( C_4 \)) such that no component is wholly contained in any one of them. Furthermore, the endpoints of \( C_1 \) (and \( C_2 \), if it is non-empty) are co-visible and the corresponding chord envelopes a critical polygon.

**Step 2** For each possible value of \( i \) and \( j \), compute a superset of all non-redundant components that have endpoints on \( C_i \) and \( C_j \). For a given \( i \) and \( j \), this superset will not contain any component that is non-redundant with respect to another component of the same orientation (clockwise or counterclockwise) and with endpoints on \( C_i \) and \( C_j \). Note that \( i \) and \( j \) may be equal.

**Step 3** At the end, a clean-up phase is carried out to eliminate components that have endpoints on \( C_i \) and \( C_j \), but are rendered redundant by components of a different orientation with endpoints on \( C_i \) and \( C_j \), or by component that have endpoints on \( C_k \) and \( C_l \) with either \( i \neq k \) or \( j \neq l \).

5.1. **Step 1. Finding the polygonal chains**

We now describe how to compute the four polygonal chains \( C_1, C_2, C_3 \) and \( C_4 \) required by Step 1 of the algorithm.

Set \( NR \) to empty. We search the boundary of \( P \) to find a reflex vertex, say \( p \). If none exists, we return \( NR \) and quit; else, we consider the clockwise ray shot from this reflex vertex, and determine, by brute force, the first point \( p' \) where this ray intersects the boundary of \( P \). This gives us a component \( P_{\text{CW}}(p, p') \) with a corresponding C-polygon denoted by \( P_i^{p} \) enveloped by the chord \( \overline{pp'} \).
Using the algorithm of Avis and Toussaint [2], we check if \( P'_1 \) is weakly visible from the chord \( pp' \). If it is, then we \( C_1 \) is set equal to \( P_{CW}(p, p') \) and we proceed to compute the other three chains. If not, we use the procedure described below in Section 5.1.1 to compute a critical polygon inside \( P'_1 \). As defined above, the critical polygon is bounded by a chord. We denote the critical polygon by \( P_1 \) and, for the sake of convenience, we relabel the bounding chord as \( pp' \). Thus, \( C_1 \) is set equal to \( P_{CW}(p, p') \) and we proceed to compute the other three chains.

Next we check if \( P - P_1 \) is weakly visible from \( pp' \) using the algorithm of Avis and Toussaint [2]. If it is, then \( C_2 \) is set equal to \( P_{CCW}(p, p') \), the other two chains \( C_3 \) and \( C_4 \) are set to empty and we proceed to Step 2 of the algorithm. If not, once again we use the procedure described below in Section 5.1.1 to compute a critical polygon \( P_2 \) inside \( P - P_1 \). The critical polygon is bounded by a chord, which we denote by \( qq' \). We set \( C_2 \) equal to \( P_{CW}(q, q') \). \( C_3 \) and \( C_4 \) are now set equal to the two left over portions of \( P \), namely \( P_{CW}(q', p) \) and \( P_{CCW}(p', q) \), respectively. Note that we are left with checking whether \( C_3 \) and \( C_4 \) (if non-empty) have any components wholly contained in them. Before proceeding further, we present the procedure to compute a critical polygon inside \( P'_1 \) and \( P - P_1 \).

5.1.1. Finding a critical polygon

In what follows, we assume that a given subpolygon (of \( P \)) denoted by \( P'_1 \) is not weakly visible from its bounding chord \( pp' \). We show how to compute a critical subpolygon \( P_1 \subseteq P'_1 \).

By Lemma 2, \( P'_1 \) contains a non-redundant component of \( P \). By definition, a critical polygon is enveloped by some bounding chord and does not wholly contain a component. As described below, the critical polygon we find may either be a non-redundant C-polygon or may be a subpolygon that is bounded by a chord passing through a reflex vertex and that contains the intersection of the clockwise and the counterclockwise C-polygon of that reflex vertex. Clearly, every line segment from which the polygon \( P \) is weakly internally visible must intersect this critical subpolygon. The following lemma is therefore an easy consequence and is stated without proof. We remark that it generalizes Lemma 5 from Das et al. [10] which states that if a simple polygon has three disjoint components, then it is not LR-visible (and, consequently, cannot have any weakly visible chords and hence cannot have any weakly visible segments).

Lemma 4. If a simple polygon \( P \) has three disjoint critical polygons then it is not weakly visible from any line segment.

First, a definition. A clockwise (respectively counterclockwise) Restricted Shortest Path between two vertices \( u \) and \( v \) of \( P \) is the shortest path (not necessarily restricted to remain within \( P \)) that only makes left (respectively right) turns and that does not intersect the polygonal chain \( P_{CW}(u, v) \) (respectively \( P_{CCW}(u, v) \)); it is denoted by \( RSP_{cw}(u, v) \) (respectively \( RSP_{ccw}(u, v) \)). Note that \( RSP_{cw}(u, v) \) (respectively \( RSP_{ccw}(u, v) \)) may be different from the actual shortest path between \( u \) and \( v \) inside \( P \) because it ignores any obstructions from the rest of the polygon, i.e., it ignores obstructions from \( P_{CCW}(u, v) \) (respectively \( P_{CW}(u, v) \)). Another way to think of \( RSP_{cw}(u, v) \) is that it is the shortest path between \( u \) and \( v \) assuming that the initial and final edges of the polygonal chain from \( u \) to \( v \) is extended indefinitely.

Next we incrementally compute the \( RSPs \) from the endpoints of \( P'_1 \) (\( p \) and \( p' \)) to all the intermediate vertices of \( P'_1 \) and use them to compute a critical polygon in \( P'_1 \). We now take a closer look at the way the (counterclockwise) restricted shortest paths are constructed as we make a counterclockwise
swEEP starting from \( p \). The counterclockwise scan is reminiscent of the linear-time “Graham scan” for computing convex hulls, in that we move forward with right turns and backtrack on left turns. This is a standard procedure employed in several algorithms (for example, see [3,4]). It can be implemented in time linear in the number of nodes of \( P'_{1} \) because every vertex of \( P'_{1} \) is inserted in some RSP from \( p \) exactly once and is deleted exactly once. Thus, incrementally computing the counterclockwise RSP to the next counterclockwise vertex on \( P'_{1} \) is straightforward. However, as we compute the RSPs, our goal is to compute a critical polygon.

Suppose that \( RSP_{\text{ccw}}(p,q) \) has been computed for all vertices \( q \in P_{\text{CCW}}(p,p_{\text{cur}}) \), where \( p_{\text{cur}} \) denotes the current vertex. Note that \( P_{\text{CCW}}(p,p_{\text{cur}}) \) consists of only right turns. The invariant maintained by the scan is that there are no clockwise components contained in \( P_{\text{CCW}}(p,p_{\text{cur}}) \). Let \( e_{\text{last}} \) be the last edge on this path, while \( e_{\text{new}} = p_{\text{cur}}p_{\text{next}} \) and \( e_{\text{old}} \) are the two edges of \( P'_{1} \) incident on \( p_{\text{cur}} \). Assuming that \( e_{\text{old}} \) and \( e_{\text{last}} \) are distinct, one of the following three situations can arise when we try to extend the path to \( p_{\text{next}} \).

(A) \( e_{\text{new}} \) makes up a left turn with \( e_{\text{last}} \) (Fig. 5(A));
(B) \( e_{\text{new}} \) makes up a right turn with both \( e_{\text{old}} \) and \( e_{\text{last}} \) (Fig. 5(B));
(C) \( e_{\text{new}} \) makes up a left turn with \( e_{\text{old}} \) and a right turn with \( e_{\text{last}} \) (Fig. 5(C)).

To see that all cases are covered, note that \( e_{\text{new}} \) can make a right turn or left turn with \( e_{\text{last}} \) and a right turn or a left turn with \( e_{\text{old}} \). Two of these possibilities are covered by case (A), while the other two are covered by cases (B) and (C).

In case (A), we scan backwards from \( p_{\text{cur}} \) until we find the point of tangency (call it \( p_{t} \)) from \( p_{\text{next}} \) to the path \( RSP_{\text{ccw}}(p,p_{\text{cur}}) \). Now \( RSP(p,p_{\text{next}}) \) is obtained by concatenating edge \( (p_{t},p_{\text{next}}) \) to the portion of the path \( RSP_{\text{ccw}}(p,p_{\text{cur}}) \) until \( p_{t} \). We then continue on the counterclockwise sweep. No critical polygon is located yet, but the invariant is clearly maintained.

In case (B), since \( e_{\text{new}} \) makes a right turn with \( e_{\text{last}} \) and \( RSP_{\text{ccw}}(p,p_{\text{cur}}) \) only involves right turns, the clockwise ray shot along \( e_{\text{new}} \) cannot hit the traversed part of \( P'_{1} \) (i.e., \( P_{\text{CCW}}(p,p_{\text{cur}}) \)) and hence cannot generate a clockwise component wholly contained in \( P'_{1} \). Now \( RSP_{\text{ccw}}(p,p_{\text{next}}) \) is computed easily by simply augmenting \( RSP_{\text{ccw}}(p,p_{\text{cur}}) \) with the edge \( e_{\text{last}} = (p_{\text{cur}},p_{\text{next}}) \). Thus the invariant that there are no clockwise components completely contained in \( P_{\text{CCW}}(p,p_{\text{cur}}) \) is maintained and we continue on the counterclockwise sweep without locating a critical polygon.

Case (C) guarantees that the clockwise ray shot along \( e_{\text{new}} \) hits the polygon inside \( P'_{1} \) since otherwise \( e_{\text{new}} \) would not have a right turn with \( e_{\text{last}} \). Assuming that the vertex \( p_{\text{next}} \) that follows \( p_{\text{cur}} \) is a reflex vertex, case (C) captures a necessary condition for the generation of a clockwise component (wholly contained in \( P'_{1} \)) by a clockwise ray shot along \( e_{\text{new}} \). Fig. 6(a) shows an example where case (C) is satisfied and results in a clockwise component. (Fig. 6(b) shows an example of a counterclockwise component that
may be generated on a symmetric clockwise sweep starting from \( p' \). This condition is only a necessary one because \( p_{\text{next}} \) need not be a reflex vertex, and, even when it is, the clockwise ray shot along \( e_{\text{new}} \) can be obstructed by the as yet unexamined part of the boundary chain \( P_{\text{CW}}(p_{\text{next}}, p') \). Also, even when \( p_{\text{next}} \) is reflex and the ray shot along \( e_{\text{new}} \) is unobstructed, the component may be redundant by way of containing a counterclockwise component. Notwithstanding, for all the possibilities that may be true when case (C) is detected, we can compute a critical polygon or a non-redundant component.

If case (C) is detected at \( p_{\text{cur}} \), then the ray shot \( \vec{r}(p_{\text{cur}}, p_{\text{next}}) \) hits the chain \( P_{\text{CW}}(p_{\text{next}}, p_{\text{cur}}) \) if it is not obstructed by \( P_{\text{CCW}}(p_{\text{next}}, p') \). We denote the chain \( P_{\text{CW}}(p_{\text{next}}, p_{\text{cur}}) \) by \( C_{\text{tail}} \) and the chain \( P_{\text{CCW}}(p_{\text{next}}, p') \) by \( C_{\text{front}} \). We first find the point, \( p'_{\text{next}} \), where the ray shot along edge \( e_{\text{new}} \) hits \( C_{\text{tail}} \) if unobstructed by \( P_{\text{CCW}}(p_{\text{next}}, p') \). We then proceed to test if the ray shot \( \vec{r}(p_{\text{cur}}, p_{\text{next}}) \) is obstructed by \( P_{\text{CCW}}(p_{\text{next}}, p') \). The polygon defined by the polygonal chain \( P_{\text{CW}}(p_{\text{next}}, p'_{\text{next}}) \) and the segment \( p_{\text{next}} p_{\text{cur}} \) will be referred to as a “pocket” and will be denoted by \( \text{pkt} \). Next, we traverse \( C_{\text{front}} \) (traversal may be clockwise or counterclockwise) to determine if this chain dips into the pocket, \( \text{pkt} \). Simultaneously, we keep track of the vertex \( p_{\text{v}} \) that causes the largest angle between the segments \( p_{\text{next}} p_{\text{cur}} \) and \( p_{\text{next}} p_{\text{cur}} \). In particular, note that in the event that \( p_{\text{next}} \) is not a reflex vertex, then \( C_{\text{front}} \) clearly dips into the pocket (at \( p_{\text{next}} \) itself), and we proceed by keeping track of \( p_{\text{v}} \) in exactly the same way. This takes care of one of the possibilities in case (C) mentioned previously.

Our immediate goal is to check if \( C_{\text{front}} \) dips into the pocket. If \( C_{\text{front}} \) has not dipped into \( \text{pkt} \), then \( \text{pkt} \) is the required subpolygon of \( P'_{1} \) that does not wholly contain another clockwise component (because of the invariant). If it does dip into \( \text{pkt} \), then we argue that \( \text{pkt} \) (and consequently, \( P'_{1} \)) is guaranteed to completely contain a clockwise component (i.e., the clockwise component at \( v \)), in which case we compute a critical polygon inside \( \text{pkt} \). If \( C_{\text{front}} \) has dipped into the pocket then \( p_{\text{v}} \) lies in that pocket (see Fig. 7), and we find the two consecutive intersection points \( p_{1} \) and \( p_{2} \) of the ray \( \vec{r}(p_{\text{next}}, p_{\text{v}}) \) with \( C_{\text{tail}} \) that are separated by \( p_{\text{v}} \). (Note that an entire edge may be supported; however, there is no loss of generality in assuming that a support point exists.) The chord \( p_{1} p_{2} \), together with \( P_{\text{CW}}(p_{2}, p_{1}) \) gives us a subpolygon which is input to the next step (in order to verify that it does not wholly contain a counterclockwise component). Fig. 7 illustrates this situation.

The subpolygon obtained from the above description may yet contain a counterclockwise component (although it cannot contain a clockwise one). So, we test this subpolygon for weak visibility from its bounding chord. If it is weakly visible, we return this subpolygon as \( P_{1} \), the critical subpolygon not containing any components. Otherwise, we repeat the above process a second time for this subpolygon.
Fig. 7. Finding a critical subpolygon.

(instead of the subpolygon $P'_1$ as was done above), this time to detect a necessary condition corresponding to a counterclockwise component (Fig. 6(b)), by traversing the polygonal boundary in clockwise order. The subpolygon returned by this repeat step is the required critical polygon and is denoted by $P_1$. Since the above process was repeated at most twice, the time complexity of what has been described so far is only linear in the length of the processed polygonal chain. This completes the description of the computation of a critical polygon $P_1$ from a subpolygon $P'_1$ that is not weakly visible from its bounding chord $pp'_1$.

**Remark 5.** We make the following observation, since we will have occasion to use it later on in this paper. If $P'_1$ had been weakly visible from $pp'_1$ to start with (i.e., $P'_1$ does not wholly contain a non-redundant component), and we had followed the above algorithm to compute the $RSP$ from $p$ to each vertex on $P'_1$, then case (C) would never have occurred during the scan from $p$ to $p'$, and we would have been able to “maintain” $RSP_{ccw}(p, x)$, as we traverse with $x$ from $p$ to $p'$ (or from $p'$ to $p$) in time linear in the number of nodes on $P_{CCW}(p, p')$. We also need the following generalization of the above observation: Given that $P'_1$ is weakly visible from $pp'_1$, and a point $a$ anywhere on $P$ along with $RSP_{ccw}(a, p)$, we can “maintain” $RSP_{ccw}(a, x)$, as we traverse with $x$ from $p$ to $p'$ in (total) time linear in the number of nodes on $P_{CCW}(p, p')$ and $RSP_{ccw}(a, p)$.

Continuing with step 1 of the algorithm for computing all non-redundant components, we have shown how to identify at most two disjoint critical polygons and how to split the polygon boundary into at most four chains, as illustrated in Fig. 8. We now proceed to check two more conditions that are necessary for $P$ to be weakly internally visible, i.e., verify that the polygonal chains $C_3$ and $C_4$ (if non-empty) do not wholly contain a component. This is achieved by computing the restricted shortest paths ($RSP$s) from the endpoints of $C_3$ to every point on it (as described in Section 5.1.1). We perform a similar procedure with $C_4$. If case (C) of the $RSP$ computation does not occur, then the $RSP$ between the endpoints of $C_3$ (and between the endpoints of $C_4$) forms a convex envelope denoted by $C_3'$ (and $C_4'$, respectively). The chains $C_3$ (thick line) and $C_3'$ (dotted line) are shown in Fig. 9. If case (C) does occur, then one of $C_3$ and $C_4$ must wholly contain a component. Thus there must exist a critical polygon disjoint from the two identified earlier (i.e., $P_1$ and $P_2$), in which case, by Lemma 4 we can stop and report that $P$ has
three disjoint critical polygons and is thus not weakly internally visible. We also check if the chains $C_3'$ and $C_4'$ intersect. If they intersect, then $C_1$ and $C_2$ are not visible to each other at all, implying that no segment inside $P$ can touch both $P_1$ and $P_2$. Therefore, we can quit after reporting that the polygon is not weakly internally visible. If $C_3'$ and $C_4'$ do not intersect, we continue with the next step. We point out that if we have not quit until now, then for the four chains ($C_1$ through $C_4$), the RSPs are identical to the corresponding shortest paths.

It is clear that Step 1 can be implemented in linear time. It may be noted that in spite of all the checks made so far, $P$ may still be not weakly internally visible. However, barring any evidence that $P$ is not weakly internally visible, we proceed to the next step.

5.2. Step 2. Computing a superset of non-redundant components

Unless we have determined that $P$ is not weakly internally visible, we now proceed to compute a superset of all the non-redundant components. Each component in this set is generated by a ray shot that emanates from a chain $C_i$ and that terminates on a chain $C_j$. Note that since one or two of the chains may be empty, the values of $i$ and $j$ that need to be considered depend on the actual situation. Also note that any counterclockwise (clockwise) component that is reported does not wholly contain a counterclockwise (clockwise, respectively) component inside it with its endpoints on $C_i$ and $C_j$. We also remark that whenever possible, we avoid reporting non-redundant components that wholly contain critical polygons (not just components). However, it would be simple to modify the algorithm so that such components are also reported.

We differentiate between the case when $i \neq j$ and when $i = j$. Sections 5.2.1 and 5.2.2 deal with the two cases.

5.2.1. Components with endpoints on different chains

Two types of queries need to be answered to facilitate this computation:

QUERY A: Does the clockwise (respectively, counterclockwise) ray shot from a reflex vertex $v \in C_i$ hit $C_j$?
QUERY B: Given a reflex vertex $v \in C_i$, and a point $x \in C_j$, does the counterclockwise (respectively, clockwise) ray shot from $v$ hit counterclockwise of $x$?
Before explaining in detail how QUERY A and QUERY B are answered, we note that in order to report a superset of all non-redundant counterclockwise components that start on \( C_i \) and terminate on \( C_j \), we traverse \( C_i \) and \( C_j \) in clockwise order. For the first reflex vertex \( v_1 \in C_i \), if the ray shot hits \( C_j \) (QUERY A), we traverse \( C_j \) until the hit point \( v'_1 \in C_j \) is computed. This component is then reported, and \( x \) marks this hit point \( v'_1 \). As we continue to traverse \( C_j \) in clockwise order, for every reflex vertex \( v \in C_j \), we check if the counterclockwise ray shot \( \vec{r}(v, v^-) \) hits counterclockwise of \( x \) (QUERY B). If it is so, then this component is discarded as being redundant. Otherwise, we compute the actual hit point (by continuing the clockwise traversal of \( C_j \)), report this component, reset \( x \) to mark this new hit point, and continue the traversal.

We traverse \( C_i \) again in counterclockwise order to identify clockwise components. It is clear that reporting the components (for pair of chains \( C_i \) and \( C_j \)) takes time that is linear in the length of \( C_i \) and \( C_j \).

A similar procedure is repeated for all pairs of chains and the resulting collection of minimal components is output as the required superset of the non-redundant components of \( P \). Also, in the output, all reflex vertices whose components are in this set have been appropriately flagged, along with a label of the edge(s) that the ray shot(s) from this vertex hits (hit). In Section 5.3, we describe a final clean-up pass to output all the non-redundant components from the given superset of the non-redundant components. Note that redundancies occur because a component obtained for one pair of chains may render redundant a component obtained for another pair of chains. In fact, even for a single pair of chains, a clockwise component may render a counterclockwise component as redundant (or vice versa).

The example in Fig. 10 suggests how we solve QUERY A. We first compute \( RSPs \) from the endpoints \( a, b \) of the chain \( C_j \) to each vertex \( v \in C_i \). For vertex \( v \in C_i \), any ray shot that is clockwise (respectively counterclockwise) of the first edge on \( RSP_{cw}(v, a) \) (respectively \( RSP_{cw}(v, b) \)), as indicated by direction 1 (respectively direction 3) in Fig. 10, will not hit \( C_j \); all other ray shots correspond to direction 2 in Fig. 10 and will hit \( C_j \). QUERY A can be easily answered if for each \( v \in C_i \) we store the edges of \( RSP_{cw}(v, a) \) and \( RSP_{cw}(v, b) \) that are incident on \( v \) (i.e., only the first edges on the \( RSPs \)). Note that we are only interested in the counterclockwise ray shot along \( \vec{r}(v^-, v) \) and the clockwise ray shot along \( \vec{r}(v^+, v) \). QUERY A can be easily answered by inspecting the directions of \( \vec{r}(v^-, v) \) (\( \vec{r}(v^+, v) \)) and the direction of the first edge along \( RSP_{cw}(v, a) \) (respectively \( RSP_{cw}(v, b) \)).

To answer QUERY B, we assume that as we traverse with \( v \) on chain \( C_j \), we maintain \( RSP_{cw}(v, a) \) and \( RSP_{cw}(v, b) \) from the endpoints \( a \) and \( b \) of chain \( C_j \). We also assume that we maintain \( RSP_{cw}(a, x) \) and \( RSP_{cw}(b, x) \) as we traverse with \( x \) along chain \( C_j \). Maintaining the \( RSPs \) is achieved as described in Remark 5. Finally, we assume that we maintain \( RSP_{cw}(v, x) \) and \( RSP_{cw}(v, x) \). An example of these paths is shown in Fig. 11, in which \( RSP_{cw}(v, a), RSP_{cw}(v, b), RSP_{cw}(v, x) \) and \( RSP_{cw}(v, x) \) are shown as dashed polygonal chains, while \( RSP_{cw}(a, x) \) and \( RSP_{cw}(b, x) \) are shown as dotted polygonal chains. Note that in the above example, \( RSP_{cw}(v, x) \) is simply the line segment joining \( v \) and \( x \), while \( RSP_{cw}(v, x) \) is the polygonal chain that passes through \( c \).

To answer QUERY B, we assume that as we traverse with \( v \) on chain \( C_j \), we maintain \( RSP_{cw}(v, a) \) and \( RSP_{cw}(v, b) \) from the endpoints \( a \) and \( b \) of chain \( C_j \). We also assume that we maintain \( RSP_{cw}(a, x) \) and \( RSP_{cw}(b, x) \) as we traverse with \( x \) along chain \( C_j \). Maintaining the \( RSPs \) is achieved as described in Remark 5. Note that the fact that none of the chains wholly contain a component is necessary for maintaining the \( RSPs \). First we check whether the counterclockwise ray shot from \( v \) hits \( C_j \) by checking whether the direction is within the wedge formed by directions of the first edges on the paths \( RSP_{cw}(v, a) \) and \( RSP_{cw}(v, b) \). If it is, then the component is discarded as redundant. Otherwise, we compute the actual hit point (by continuing the clockwise traversal of \( C_j \)), report this component, reset \( x \) to mark this new hit point, and continue the traversal.

The example in Fig. 10 suggests how we solve QUERY A. We first compute \( RSPs \) from the endpoints \( a, b \) of the chain \( C_j \) to each vertex \( v \in C_i \). For vertex \( v \in C_i \), any ray shot that is clockwise (respectively counterclockwise) of the first edge on \( RSP_{cw}(v, a) \) (respectively \( RSP_{cw}(v, b) \)), as indicated by direction 1 (respectively direction 3) in Fig. 10, will not hit \( C_j \); all other ray shots correspond to direction 2 in Fig. 10 and will hit \( C_j \). QUERY A can be easily answered if for each \( v \in C_i \) we store the edges of \( RSP_{cw}(v, a) \) and \( RSP_{cw}(v, b) \) that are incident on \( v \) (i.e., only the first edges on the \( RSPs \)). Note that we are only interested in the counterclockwise ray shot along \( \vec{r}(v^-, v) \) and the clockwise ray shot along \( \vec{r}(v^+, v) \). QUERY A can be easily answered by inspecting the directions of \( \vec{r}(v^-, v) \) (\( \vec{r}(v^+, v) \)) and the direction of the first edge along \( RSP_{cw}(v, a) \) (respectively \( RSP_{cw}(v, b) \)).
and $RSP_{cw}(v, b)$. If not, then this component is ignored (since it will be handled for a different pair of values of $i$ and $j$). Otherwise, we proceed to decide whether the hit point is counterclockwise of $x$.

Finally, we assume that we also maintain $RSP_{cw}(v, x)$ and $RSP_{cw}(v, x)$; this last pair of RSPs can be maintained because at any one time either we traverse with $v$ on $C_i$ or we traverse with $x$ on $C_j$, and in either case, the incremental computations for the two RSPs are exactly the same as for the RSP computations described earlier. Furthermore, the cost is linear in the number of vertices traversed with $v$ or traversed with $x$. If $v$ and $x$ are co-visible, then $RSP_{cw}(v, x)$ and $RSP_{cw}(v, x)$ are both equal to the straight line joining $v$ and $x$. In this case, answering if the hit point is counterclockwise of $x$ is a simple matter of deciding whether the ray shot is counterclockwise of the ray from $v$ to $x$. On the other hand, if $v$ and $x$ are not co-visible, then since $C_j$ does not have any components wholly contained in it, either the first edge of $RSP_{cw}(v, x)$ or the first edge of $RSP_{cw}(v, x)$ will equal the straight line joining $v$ and $x$, and the first edge of $RSP_{cw}(v, x)$ will be counterclockwise of the first edge of $RSP_{cw}(v, x)$. An example of all the paths required to answer QUERY B are shown in Fig. 11. If, as is shown in the example, we assume that the first edge of $RSP_{cw}(v, x)$ is equal to straight line joining $v$ and $x$, then any ray shot that is directed in between the two first edges will hit $RSP_{cw}(v, x)$ before it hits $RSP_{cw}(v, x)$ and, therefore, the ray shot will hit counterclockwise of $x$. If, on the other hand, we assume that the first edge of $RSP_{cw}(v, x)$ is equal to straight line joining $v$ and $x$, then the ray shot will hit clockwise of $x$.

A final note about finding the actual hit point. Once we have determined that the ray shot will hit counterclockwise of $x$, we traverse counterclockwise with $x$ along $P$ until we locate the edge on which the hit point is located. Finding the actual hit point is a trivial matter of finding the intersection between the counterclockwise ray shot from $v$ and the edge in question.

5.2.2. Components with both endpoints on same chain

At this point we know that $C_1$ is non-empty and weakly visible from its bounding chord $pp'$. Therefore, any component with both endpoints on $C_2$, $C_3$ or $C_4$ will wholly contain the critical polygon corresponding to $C_1$ and need not be reported. If $C_3$ or $C_4$ is non-empty, then $C_2$ must be critical, and by a similar argument we need not report components with both endpoints on $C_1$ either (in which case, we proceed to step 3). Therefore, we assume that $C_2$ is not critical.
As usual, we will have two scans (a counterclockwise one and a clockwise one) of $C_1$ to report a superset of all non-redundant components. We only describe the counterclockwise scan. The algorithm involves visiting the reflex vertices of $C_1$ in counterclockwise order and deciding whether or not to report the corresponding clockwise component at that reflex vertex. In fact, we make two counterclockwise traversals of $C_1$ with two pointers $x$ and $y$ starting from $p$. The scan with $x$ visits all the reflex vertices. The scan with $y$ helps to decide whether the location of the hit point of the clockwise ray shot from the reflex vertex at $x$ causes a redundant component. For the scan with $x$, we also maintain $RSP_{cw}(p', x)$, computed in a manner as described earlier.

Initially, we traverse simultaneously with $x$ and $y$ until both reach the first reflex vertex. After that we go through iterations. In iteration 1, the first reflex vertex $v_1$ encountered along the scan is dealt with differently from the others. In each iteration, one component is reported or the algorithm stops. We first describe the processing in iteration 1.

Our first task is to find the hit point of the clockwise ray shot at $v_1$, if it lies in $P_{ccw}(p, p')$. Note that the hit point cannot lie on $P_{ccw}(p, v_1)$ since this would contradict the assumption that $C_1$ does not wholly contain a component. We traverse with $y$ ($x$ is stationary at $v_1$) along $P_{ccw}(v_1, p')$ until we reach the first point of intersection (denoted by $v'_i$) of $P_{ccw}(v_1, p')$ with the clockwise ray shot from $v_1$. If the point does not exist (i.e., we reach $p'$ without finding it), then we quit and report no components with both endpoints on $C_1$. However, if it exists, the point $v'_i$ must also be the hit point of the clockwise ray shot from $v_1$. If $v'_i$ is not the hit point, then it must be due to obstruction from $P_{ccw}(v_1', p')$, in which case a reflex vertex from within this obstruction will necessarily cause a component to wholly lie within $C_1$, which is a contradiction.

If we have not quit so far, then we can report the component $P_{ccw}(v_1, v'_i)$ as a component with both its endpoints on $C_1$. At this point $x$ is at $v_1$ and $y$ is at $v'_i$. We now traverse again with $x$ while maintaining $RSP_{cw}(v'_i, x)$.

We now describe the $(k + 1)$st iteration, which starts just after we have reported $k$ components having both endpoints on $C_1$. The invariant at the start of the $(k + 1)$st iteration is as follows: $x$ is at $v_k$, which is the reflex vertex of the $k$th component reported so far; $y$ is at $v'_k$, which is the hit point of the $k$th component; we have maintained $RSP_{cw}(p', x), RSP_{cw}(v'_i, x), RSP_{ccw}(v'_1, v'_k)$ and $RSP_{ccw}(v'_k, y)$; finally, we also maintain the “tangent” point $\tau(x)$ where the common tangent between $RSP_{cw}(v'_1, x)$ and $RSP_{cw}(v'_i, v'_k)$ touches the chain $RSP_{ccw}(v'_1, v'_k)$.

In iteration $k + 1$, $x$ moves to the next reflex vertex on $P_{ccw}(v_k, p')$. During this traversal, $RSP_{cw}(p', x)$ and $RSP_{cw}(v'_i, x)$ are maintained as described earlier. Furthermore, we maintain the tangent point $\tau(x)$, which will monotonically move along $RSP_{cw}(v'_1, v'_k)$. This situation is shown in the example in Fig. 12. It shows the polygonal chain $C_1 = P_{ccw}(p, p')$ with its bounding chord $pp'$ (thick line). The clockwise components reported in the first $k$ iterations are shown as the dashed lines $v_1, v_1, \ldots, v_k, v'_k$. The three restricted shortest paths – $RSP_{cw}(p', x), RSP_{cw}(v'_i, x)$ and $RSP_{ccw}(v'_1, v'_k)$ are shown as dotted polygonal chains. The common tangent between the polygonal chains $RSP_{cw}(v'_1, x)$ and $RSP_{ccw}(v'_1, v'_k)$ is shown as a thick line. The figure also shows the tangent point $\tau(x)$, where the common tangent terminates on $RSP_{ccw}(v'_1, v'_k)$. The point $y$ traverses on the polygonal chain $P_{ccw}(v'_k, p')$. The point $y$ and $RSP_{ccw}(v'_k, y)$ are not shown in the figure.

At the reflex vertex reached by $x$, firstly, if the clockwise ray shot from $x$ is counterclockwise of the last edge along $RSP_{cw}(p', x)$, then it is discarded since the clockwise component at $x$ does not have both its endpoints on $C_1$ (and we move with $x$ to the next reflex vertex). If it does, then if the clockwise ray shot from $x$ is clockwise of the last edge along $RSP_{cw}(v'_1, x)$, then it is discarded since it is rendered redundant.
Fig. 12. Components with endpoints on same chain.

by the component $P_{CCW}(v_1, v'_1)$. If not, then if the clockwise ray shot from $x$ is clockwise of the direction of $\vec{r}(x, \tau(x))$, then the ray shot hits $P_{CCW}(v'_1, v''_1)$ and it is discarded since it is rendered redundant by the component $P_{CCW}(v_k, v'_k)$. Note that if the component at $x$ is discarded, then we move with $x$ to the next reflex vertex and continue with iteration $k + 1$. If not, then we need to report a component. In that case, we label $x$ as $v_{k+1}$, and traverse with $y$ from $v'_k$ until we find the hit point $v'_k$ on $P_{CCW}(v'_1, v'_k)$.

As we traverse with $y$, $RSP_{cw}(v'_k, y)$ is maintained. After reaching $v'_{k+1}$, we compute $RSP_{cw}(v'_1, v'_{k+1})$ by merging $RSP_{cw}(v'_1, v'_k)$ and $RSP_{cw}(v'_k, v'_{k+1})$ in time that is proportional to the number of nodes on $RSP_{cw}(v'_k, v'_{k+1})$ and the number of nodes on $RSP_{cw}(v'_1, v'_k)$ that are not on $RSP_{cw}(v'_k, v'_{k+1})$. Finally, we update the tangent point $\tau(v_{k+1})$ to be the point $v'_{k+1}$.

The process is stopped as soon as we reach $p'$ on the traversal with $y$. Some complications may arise here. What if we reach $v'_1$ with $x$ before we reach $p'$ with $y$? Then, let $v_k$ be the last reflex vertex from which a component was reported. If $v_k$ is different from $v_1$, then we relabel the vertices $v_k$ and $v'_k$ as the vertices $v_1$ and $v'_1$ and we restart with iteration 2. Note that the invariants required for iteration 2 are satisfied (i.e., $RSP_{cw}(v'_1, x)$ and $RSP_{cw}(p', x)$) and we can continue without any more processing. If the last component reported was from reflex vertex $v_1$, then we restart with iteration 1 and identify a new reflex vertex $v_1$ (as described in the processing for iteration 1 above).

To analyze the time complexity, we note that all the $RSP$ computations have a total time that is linear in the number of nodes on $C_1$. All the tests related to checking directions only take $O(1)$ time. To see that all the tangent point computations also take time that is linear in the number of nodes on $C_1$, it is sufficient to note that for any two points $x'$ and $x''$ such that $x'' \in P_{CCW}(x', v_1)$, $\tau(x'')$ cannot lie on the scanned portion of $RSP_{cw}(v'_1, v'_k)$ between $\tau(x')$ and $v'_1$.

Once again, our discussions imply that the entire Step 2 described above can be implemented in linear time.
5.3. Step 3. Clean-up phase

This step is described in the LR-visibility algorithm by Das et al. [10] (see the start of Section 4 of that paper). We summarize it here for completeness. The idea is to obtain a sorted list of all the endpoints of the components in the superset by performing a few simple traversals of the list. Once this is done, we can think of the output of step 2 as a collection of circular arcs from which one simple traversal will ensure that all the redundant components are eliminated.

Suppose we have a set of clockwise components which contains a superset of non-redundant components. As we traverse $P$ in clockwise order, we encounter a beginning point and an ending point of each component. Since the beginning points are vertices of $P$, they can be sorted in linear time. Suppose we traverse $P$ twice counterclockwise. Each time we encounter a beginning point, we compare the ending point of the component to the ending point of the previous component; if the current component contains the previous component, then the current component is redundant and therefore is deleted from the list of components. We must traverse $P$ twice since one of the first components considered may be redundant with respect to one of the last ones. After an analogous procedure is performed for counterclockwise components, we have two lists of components, each in sorted order, which can be merged and pruned of redundant components in linear time to obtain a sorted list of all non-redundant components.

In this section we have described an algorithm to output all non-redundant components of a polygon in linear time. As per the overview of the entire algorithm presented in Section 4, this can be used to output all LR-visible pairs of points and all weakly-visible chords of the polygons in linear time.

We point out that the algorithms in [10] (for computing all LR-visible pairs of points) and [11] (for computing all weakly-visible chords) use the list of non-redundant components as input and run in linear time. It is significant to note that the algorithms in the two papers (which we use here in the following sections), do not require the expensive triangulation algorithm of Chazelle [7] or the shortest path algorithm of Guibas et al. [16] once they are already supplied with a list of non-redundant components as input.

6. Case 1. Non-tangential weakly-visible segment

As mentioned earlier, this case corresponds to the situation when the shortest weakly-visible segment does not touch any vertex of the polygon except possibly at its endpoints. For each $i = 1, \ldots, k$, let $SN_i$ be the shortest non-tangential weakly-visible segment (if one exists) that joins $\alpha_i$ and $\beta_i$ with at least one endpoint on $\alpha_i - \alpha_i + 1$ or $\beta_i - \beta_i + 1$. The shortest of the segments $SN_i$, $i = 1, \ldots, k$, is the shortest non-tangential weakly-visible segment that joins $\alpha_i$ and $\beta_i$, $i = 1, \ldots, k$.

6.1. Structure of $\alpha_i$ and $\beta_i$

As mentioned earlier, $\alpha_i$ is the envelope of the intersection of a set of C-polygons that contain $A_i$ and is denoted by $CA_i$; $\beta_i$ is the envelope of the intersection of a set of C-polygons that do not contain $A_i$ and is denoted by $CB_i$. Hence it is clear that both of them are convex polygonal chains. It may be possible that $\alpha_i = \alpha_i + 1 = \alpha_i + 2 = \cdots = \alpha_i + p$. This simply means that no component starts or ends on the portion of $P$ covered by $A_i, A_i + 2, \ldots, A_i + p$.
We now describe the structural differences between $\alpha_i$ and $\alpha_{i+1}$ (in case they do differ), and the corresponding differences between $\beta_i$ and $\beta_{i+1}$. The main purpose of studying this structure is to identify the polygonal chains $\alpha_i - \alpha_{i+1}$ and $\beta_i - \beta_{i+1}$ so that they can be processed in the $i$th iteration. Clearly, if $\alpha_i = \alpha_{i+1}$, then no processing is required in iteration $i$.

Assume that $\alpha_i \neq \alpha_{i+1}$. From Das et al. [11] we know that $A_i$ and $A_{i+1}$ are disjoint line segments. On closer inspection of their algorithm, we observe that there are various events that trigger the chords algorithm to go from iteration $i$ to iteration $i+1$, thus outputting pairs $(A_i, B_i)$ and pairs $(A_{i+1}, B_{i+1})$.

One such event occurs if a component starts or ends between $A_i$ and $A_{i+1}$ (such as the point $p_2$ in Fig. 13(b) where a component starts, or the point $p_1$ in Fig. 13(a) where a component terminates). The other possible events (which result in $\alpha_i = \alpha_{i+1}$) have to do with changes in the points of tangency for the boundaries of the weakly-visible chords. This happens because one could obtain a weakly-visible chord that is tangential to the polygon at some vertex. As we rotate this chord, it could continue to be weakly-visible while remaining tangential to the polygon at the same vertex. However, as we rotate more, the point of tangency could change, triggering an event that the chords algorithm needs to deal with (since the “compact” description of the chords changes with this event).

The chain $\alpha_i$ is different from $\alpha_{i+1}$ only when a component starts or ends between $A_i$ and $A_{i+1}$. For the next three paragraphs we will assume that the counterclockwise end for any polygonal chain is the front end, while the clockwise end is the tail end.

If a component $c$ starts between $A_i$ and $A_{i+1}$, the changes from $\alpha_i$ to $\alpha_{i+1}$ are as shown in Fig. 13(b). Note that the C-polygon corresponding to component $c$ lies to the left of the segment $p_2q_2$ and that the

![Fig. 13. Changes in the structure of $\alpha_i$ and $\beta_i$.](image-url)
component \( c \) consists of the polygonal chain \( P_{CCW}(p_2, q_2) \). \( A_i \) lies to the right of \( p_2 \), while \( A_{i+1} \) lies to the left of \( p_2 \). \( \alpha_i \) consists of the chain from \( a_i \) to \( s_2 \) to \( a_f \), while \( \alpha_{i+1} \) consists of the chain from \( p_2 \) to \( s_2 \) to \( a_f \), i.e., a portion of the tail of \( \alpha_i \) gets replaced by a portion of the ray shot corresponding to the component \( c \). At the same time, as shown in Fig. 13(b), \( \beta_i \) has a portion of its front replaced by a new polygonal chain. \( \beta_i \) consists of the chain from \( b_t \) to \( t_2 \) to \( q_2 \), while \( \beta_{i+1} \) consists of the chain from \( b_t \) to \( t_2 \) to \( b_f \). In other words, \( CA_i \) shrinks at its tail end, and \( CB_i \) grows at its front end, while both their envelopes remain convex. Note that \( \alpha_i - \alpha_{i+1} \) comprises of the polygonal chain from \( a_i \) to \( s_2 \), while \( \beta_i - \beta_{i+1} \) comprises of the segment from \( q_2 \) to \( t_2 \).

Note that in Figs. 13(a) and (b), the region \( CA_i \cap CA_{i+1} \) (as well as the region \( CB_i \cap CB_{i+1} \)) have been shown as a filled region. The area occupied by \( CA_{i+1} \) (but not by \( CA_i \)) is indicated as a dot-filled region, while the area occupied by \( CA_i \) and \( CB_i \) is left blank.

By a similar argument, if a component \( c \) ends between \( A_i \) and \( A_{i+1} \), the portion of the ray shot corresponding to \( c \) at the front (right end or the counterclockwise end) of \( \alpha_i \) gets replaced by a new polygonal chain, causing \( CA_i \) to grow in the front. As shown in Fig. 13(a), \( \beta_i \) has a portion of its tail (right end or counterclockwise end) replaced by a portion of the ray shot corresponding to \( c \), thus causing \( CB_i \) to shrink at its tail end. In this case note that \( \alpha_i - \alpha_{i+1} \) comprises of the segment from \( p_1 \) to \( s_1 \), while \( \beta_i - \beta_{i+1} \) comprises of the chain from \( b_t \) to \( t_1 \).

The above description elucidates the changes that take place to the \( \alpha \) and \( \beta \) chains while moving from the \( i \)th iteration to the \((i + 1)\)st iteration.

### 6.2. Data structure for storing the \( \alpha \) and \( \beta \) chains

We now describe the process of constructing the data structure to store all the chains \( \alpha_i \) and \( \beta_i \).

We first describe how \( \alpha_1 \) is computed and stored. Let \( CA_1 \) be the intersection of the C-polygons \( C_1, C_2, \ldots, C_p \) (i.e., the C-polygons that contain basic interval \( A_1 \)) listed in clockwise order of their clockwise endpoints (you may also use the counterclockwise endpoint). Let \( C_i \) have \( l_i \) as its clockwise endpoint and \( m_i \) as its counterclockwise endpoint. To start with, \( C_1 \) is a C-polygon whose envelope is a segment consisting of a chord of the polygon. Assume that the intersection of the C-polygons \( C_1, C_2, \ldots, C_j \), for some \( j < p \) has been computed and its envelope is stored as a linked list of segments, \( e_1 = r_1l_2 \), \( e_2 = r_2l_3 \), \( e_3 = r_3l_4 \), \ldots, \( e_q = r_qm_{q+1} \). Note that \( q \leq i \). We show how to add the C-polygon \( C_{i+1} \). Note that \( C_{i+1} \) is formed by the chord \( c = l_{i+1}m_{i+1} \). To determine the intersection of \( C_1, \ldots, C_i, C_{i+1} \), we find the intersection of the chain \( e_1, \ldots, e_q \) with the chord \( c \). This is done by scanning the sequence \( e_1, \ldots, e_q \) in reverse order and checking each of the segments for intersection with \( c \). Let segment \( e_j, j \leq q \), intersect chord \( c \) at point \( d \). Now the current linked list \( e_1, \ldots, e_q \) is updated to \( e_1, \ldots, e_{j-1}, e'_j, e_{j+1} \), where \( e'_j = r_jd \), i.e., the subsegment of \( e_j \) that ends at \( d \), and \( e_{i+1} = dm_{i+1} \), i.e., the subsegment of the chord \( c \) starting from \( d \). The old linked list from \( e_j, \ldots, e_q \) is not physically deleted; instead it is pushed to the background. In this sense, this could be thought of as a persistent structure for linked lists. It is much simpler than the generalized persistent structure for trees, as presented by Sarnak and Tarjan [27], since the set of operations to be performed on this structure are much simpler (as shown later). Later it will become necessary to delete the C-polygons \( C_{i+1}, \ldots, C_p \) (in precisely the reverse order), in which case, the old linked list will become the current envelope of the region of intersection of the C-polygons \( C_1, \ldots, C_i \). When all the C-polygons \( C_1, \ldots, C_p \) are processed, \( \alpha_1 \) is stored as a linked list, which is really the leftmost path of a tree structure. As described later, \( \alpha_1 \) can be thought of as a concatenation of two chains. The first one is stored as described above, while the second chain is initially empty and is...
stored in an empty tree structure; as $CA_i$, for $i > 1$ is computed, it may become non-empty. Note that $CB_1$ is the intersection of all C-polygons not included in the processing for $\alpha_1$, and thus $\beta_1$ can be computed in a fashion similar to that of $\alpha_1$.

An example of the computations of $\alpha_1$ is shown in Fig. 14. After the first four C-polygons are processed, $\alpha_1$ consists of four segments $e_1, e_2, e_3, e_4$, where $e_1 = m_1r_2, e_2 = r_2r_3, e_3 = r_3r_4$ and $e_4 = r_4m_4$. The line with points $a$ and $b$ is a simplified picture of a subchain of the input polygon $P$ where the relevant subchain is shown as a straight line edge. The corresponding data structure at this point is shown in Fig. 15(a). When the fifth C-polygon is processed, $\alpha_1$ consists of $e_1, e_2, e_3'$, where $e_3' = r_3r_5$ and $e_5 = r_5m_5$. The corresponding changes to the data structure are reflected in Fig. 15(b), where the leftmost path stores the current value of $\alpha_1$. After the sixth C-polygon is processed, $\alpha_1$ consists of $e_1, e_2', e_6$, where $e_2' = r_2r_6$ and $e_6 = r_6m_6$. Note again that the leftmost path of the data structure shown in Fig. 15(c) stores the final value of $\alpha_1$.

For the $i$th iteration, we describe how to compute and store $\alpha_{i+1}$ assuming that $\alpha_i$ has been computed.

As we move counterclockwise from $A_i$ to $A_{i+1}$, either (1) a component that contained $A_i$ does not contain $A_{i+1}$ but contains $B_{i+1}$, or (2) a component that did not contain $A_i$ now contains $A_{i+1}$ (and may or may not contain $B_i$). We inductively assume that instead of storing $\alpha_i$ as the leftmost path of a single tree structure, it is stored as the concatenation of two paths $\alpha_i'$ and $\alpha_i''$, which are subpaths of the leftmost path of two different tree structures denoted by $T'$ and $T''$. We also separately store their point of intersection $z_i$ (if it exists), thus making it easy to derive the chain $\alpha_i$ whenever necessary.

![Fig. 14. An example of the envelope $\alpha_1$.](image1)

![Fig. 15. The changes to the data structure that stores $\alpha_i$'s.](image2)
Let $c$ be the non-redundant component that contains all of the segments $A_1, A_2, \ldots, A_k$ output by the chords algorithm (see [11]). The chain $\alpha'_i$ (respectively $\alpha''_i$) is defined as the envelope of the region of intersection of all C-polygons whose corresponding components have their tail (respectively front) end inside $c$ and within the subchain spanned by $A_1, \ldots, A_i$.

Every component $c'$ must satisfy one of the following conditions:

(a) $c'$ is disjoint from $c$;
(b) $c'$ has only its front endpoint in $c$;
(c) $c'$ has only its tail endpoint in $c$;
(d) $c'$ has both endpoints in $c$.

Assume that $c'$ is the component with an endpoint between $A_i$ and $A_{i+1}$ (and, therefore, encountered in iteration $i$). Thus case (a) is impossible. If case (b) holds, $c'$ must have already been considered for the computation of the first chain $\alpha_1$; it is thus part of $\alpha'_1$ (initial tree $T'$), but is deleted from $T'$ in iteration $i$. If case (c) holds, $c'$ is considered for addition to $\alpha''_i$ in iteration $i$; this is achieved by inserting it into $T''$. Finally if case (d) holds, the front end of $c'$ must be encountered before the tail end, since otherwise it would render the component $c$ redundant. In this case $c'$ must be part of the initial tree $T'$; it is deleted from $T'$ in iteration $i$ and is finally inserted into $T''$ when its tail end is encountered in a later iteration.

Deleting a set of C-polygons from $T'$ is always done in the reverse order in which they were added to create the structure for $\alpha_1$ – this is because the right endpoints of the components are encountered in the same order as their left endpoints. These deletions are easy since they are a simple reversal of the process described earlier for adding a new C-polygon.

In contrast, adding a C-polygon is handled in a different manner. When adding C-polygons, they are added to $T''$, which is initially empty. Hence, $\alpha'_1 = \alpha_1$ and $\alpha''_1$ is empty; thus $z_1$ is simply the endpoint of $\alpha'_1$. Also, since the additions are done in counterclockwise order, this process is similar to the additions done in the computation for $\alpha_1$. The point $z_1$ is marked and stored on both the parallel structures for $\alpha'_1$ and $\alpha''_1$. The chain $\alpha_i$ is simply the concatenation of two subchains, namely, the subchains of $\alpha'_i$ and $\alpha''_i$ ending at $z_i$. Both the chains $\alpha'_i$ and $\alpha''_i$ are stored in the leftmost path of the two tree structures, $T'$ and $T''$. The tree $T'$ starts with $\alpha_1$ stored in its leftmost path, with $z_1$ at its leftmost leaf vertex. The second structure starts out empty, and at any instant has $\alpha''_1$ stored in its leftmost path. Thereafter, the first structure only has C-polygons deleted from it, while the second structure only has C-polygons added to it. Note that in moving from $A_i$ to $A_{i+1}$, only one of the two structures undergoes change. The idea of the chain $\alpha_i$ being a combination of two chains is similar to a scheme used by [20] in his algorithm for computing the envelope of a set of lines.

Note that the computation for $\beta_i$ is different from that described for $\alpha_i$. We now discuss the time complexities of the computations described above. Every time a C-polygon is added to one of the structures, the leftmost path may change and one of the vertices on that path may acquire a new left child. Thus each of the $O(n)$ additions involves traversing the current tree structure from its leftmost leaf, until the intersecting segments are reached. This pushes a portion of the leftmost path into the background. This portion of the path remains in the background until the C-polygon added last is deleted, at which time it once again becomes the current chain. Furthermore, it is easy to see that the point $z_{i+1}$ can be computed from the point $z_i$ by a monotonic movement in the two tree structures. This is justified as follows. C-polygons are only added to $T''$, and only deleted from $T'$. In each iteration, there is a change in either the leftmost path of $T''$ or of $T'$. If this change takes place (in say, $T''$) below (farther from
the root of the tree) the current location of \( z_i \), then \( z_{i+1} \) does not change from \( z_i \). If the change in \( T' \) takes place above (closer to the root) the current location of \( z_i \), then a fresh sweep is started from the new leaf on \( T' \) along its leftmost path (towards the root) until \( z_{j+1} \) is located. Also, we sweep from the current location of \( z_i \) on \( T'' \) towards the root to locate \( z_{j+1} \). Because of the planarity of the two parallel structures, both of them are of size \( O(n) \). Every vertex on both the tree structures is encountered once when it is created, once when it is pushed into the background, and once when it is deleted. Clearly, the total amount of processing of each vertex with regard to the creation of the data structures is constant. Furthermore, it is easy to see that the point \( z_{i+1} \) can be computed from the point \( z_i \) by a monotonic movement in the two parallel tree structures. Finally, the total change in the envelopes is of size \( O(n) \). Hence all the computations described above can be performed in \( O(n) \) time.

6.3. Computing \( SN_i \)

As mentioned earlier, the algorithm goes through \( k \) iterations. In the \( i \)th iteration, the chains \( \alpha_i - \alpha_{i+1} \) and \( \beta_i - \beta_{i+1} \) are identified, and the shortest segment that joins \( \alpha_i \) and \( \beta_i \) with one endpoint on \( \alpha_i - \alpha_{i+1} \) or \( \beta_i - \beta_{i+1} \) is computed. Let this segment be \( SN_i = s_j \), if it exists. Note that identifying \( \alpha_i - \alpha_{i+1} \) simply involves maintaining the point where \( \alpha_i \) and \( \alpha_{i+1} \) diverge. The algorithm needs two pointers to store this point since two parallel structures store the \( \alpha \) chains.

Given any two convex polygonal chains \( \alpha \) and \( \beta \), there is a simple sweep algorithm to find the shortest line segment that joins the two chains. In this case, \( \alpha \) and \( \beta \) are two chains that form the convex envelope of two disjoint polygons. The algorithm involves sweeping the two chains, one from its clockwise end and in counterclockwise order, the other from its counterclockwise end in clockwise order. For each vertex on \( \alpha \), the sweep algorithm finds the closest point on \( \beta \). Similarly, for each vertex on \( \beta \), the sweep algorithm finds the closest point on \( \alpha \). Finally, the closest of the pairs is reported. Informally speaking, the sweep algorithm works because of three simple facts: (1) for a fixed point \( a \in \alpha \), its distance to visible points \( b \in \beta \) is unimodal, (2) as point \( a \) moves monotonically on \( \alpha \), its closest point on \( \beta \) moves monotonically on \( \beta \), (3) for points \( a \in \alpha \), its shortest distance to \( \beta \) (i.e., the distance to its closest point on \( \beta \)) is unimodal. Intuitively speaking, fact (3) states that the local minimum is also the global minimum for that particular iteration.

If \( SN_i \) has one endpoint on \( \alpha_i - \alpha_{i+1} \) and the other on \( \beta_i - \beta_{i+1} \), then this will be discovered by the algorithm in iteration \( i \). If \( SN_i \) has neither endpoint on \( \alpha_i - \alpha_{i+1} \) and \( \beta_i - \beta_{i+1} \), then it will be discovered by the algorithm in a later iteration, i.e., \( SN_i = SN_j \) for some \( j > i \). A subtle complication is introduced by the possibility that \( SN_i \) may have one endpoint on \( \beta_i - \beta_{i+1} \) and another endpoint on \( \alpha_i \cap \alpha_i+1 \) (instead of \( \alpha_i - \alpha_i+1 \)). An example of such a situation is shown in Fig. 16. This would be detected by the algorithm since the sweep algorithm (described at the start of this subsection) for finding the shortest line segment joining two convex polygonal chains would reach the end of one of the chains without hitting a local minimum. For example, assume that the end of \( \alpha_i - \alpha_{i+1} \) is reached before reaching the end of \( \beta_i - \beta_{i+1} \) and before a minimum was encountered. In this case, our algorithm continues sweeping on \( \beta_i - \beta_{i+1} \), while continuing the sweep on \( \alpha_i \cap \alpha_i+1 \). Our algorithm needs to be modified to ensure that this portion of \( \alpha_i \cap \alpha_i+1 \) is not swept again during iteration \( j \) (for some \( j > i \)). In this case, we claim that \( SN_j \) cannot have an endpoint on this portion of \( \alpha_i \cap \alpha_i+1 \) and hence need not be considered in any later iteration. This claim is proved in Lemma 6 below. The relevant portion of \( \alpha_i \cap \alpha_i+1 \) is marked visited so that a sweep in a later iteration can skip over this portion of the chain. This is simply implemented by storing skip...
pointers in the data structure. The entire arguments in this paragraph could have been carried out with \( \alpha \) replaced by \( \beta \) and vice versa.

The above arguments are clarified by the example in Fig. 16. In the example, let \( \alpha_i \) and \( \beta_i \) be the boundaries of the intersection of the \( C \)-polygons that contain (respectively do not contain) the point \( x \) on the polygon. Then, \( \alpha_i \) is given by the chain \( l_1, l_2, l_3, l_4, l_5 \) and \( \beta_i \) is given by the chain \( k_1, k_2, k_3, k_4, k_5, k_6 \). The portion of the polygon under \( \alpha_i \) and \( \beta_i \) are simplified as straight line segments shown in the figure as \( \overline{\alpha_1\alpha_2} \) and \( \overline{\beta_1\beta_2} \). As we move counterclockwise along the polygon from \( x \) to \( x' \), \( \alpha_i \) changes to \( \alpha_{i+1} \) while \( \beta_i \) changes to \( \beta_{i+1} \). In this example, \( \alpha_{i+1} \) consists of the chain \( l_1, l_2, l_3, l_4, l_5, l_6 \), while \( \beta_{i+1} \) consists of the chain \( k_1, k_2, k_3, k_4, k_5, k_6 \). Now \( \alpha_i - \alpha_{i+1} \) consists of the chain \( l_3, l_4, l_5, l_6 \), while \( \beta_i - \beta_{i+1} \) consists of the chain \( k_7, k_2 \). Also, \( SN_i \), which is the shortest line segment joining \( \alpha_i \) and \( \beta_i \) with one endpoint on \( \alpha_i - \alpha_{i+1} \) or \( \beta_i - \beta_{i+1} \) is the line segment \( \overline{sT_i} \). The point \( s_i \) lies on \( \alpha_i - \alpha_{i+1} \), while \( t_i \) does not lie on \( \beta_i - \beta_{i+1} \). The algorithm sweeps the chains \( \alpha_i \) and \( \beta_i \) starting from \( l_4 \) and \( k_7 \), respectively. The sweep along \( \beta_i \) reaches \( k_2 \) when it is recognized that \( SN_i \) does not join a point on \( \alpha_j - \alpha_{j+1} \) and a point on \( \beta_j - \beta_{j+1} \). If the sweep along \( \alpha_i \) had reached \( l_6 \), then the search for \( SN_i \) would have been abandoned and left for a later iteration. However, the points \( s_i \) and \( t_i \) are discovered before reaching \( l_6 \) on \( \alpha_i \). Note that the nearest point from a point on \( \alpha_i \cap \alpha_{i+1} \) has to lie on the chain \( t_i, k_4, k_5, k_6 \) due to the monotonicity properties. In other words, point \( t_i \) will be closer to a \( \alpha_j \) chain \((j > i)\) than any point on the subchain from \( k_2 \) to \( t_j \). Hence the portion of \( \beta_j \) between \( k_2 \) and \( t_j \) need not be processed in iteration \( j \) for any \( j > i \) for computing \( SN_j \).

These arguments are formalized in Lemma 6.

**Lemma 6.** If in iteration \( i \), a portion of \( \alpha_i \cap \alpha_{i+1} \) (or \( \beta_i \cap \beta_{i+1} \)) was traversed to compute \( SN_i \), then this portion need not be traversed again for any iteration \( j > i \) to compute \( SN_j \).

**Proof.** We use the notation \( CP(p, \alpha) \) to denote the point on a convex polygonal chain \( \alpha \) closest to point \( p \).

For iteration \( i \), either \( \alpha_i - \alpha_{i+1} \) is a straight line segment and \( \beta_i - \beta_{i+1} \) is a convex polygonal chain, or \( \alpha_i - \alpha_{i+1} \) is a convex polygonal chain and \( \beta_i - \beta_{i+1} \) is a straight line segment. W.l.o.g. we assume the former, implying that the component corresponding to some chord \( l_i \) must terminate between \( A_j \) and \( A_{j+1} \) (as shown in the example in Fig. 13(a)). Note that the latter case would have implied that a component started between \( A_j \) and \( A_{j+1} \) (see Fig. 13(b)). The chord \( l_i \) may or may not intersect \( \beta_i \). If it does not, then either \( \beta_i - \beta_{i+1} = \beta_i \) or \( \beta_i - \beta_{i+1} = \emptyset \). The case of \( \beta_i - \beta_{i+1} = \emptyset \) implies that \( \beta_i \cap \beta_{i+1} = \emptyset \) and \( \beta_i \cap \beta_{i+1} = \emptyset \) are empty, implying that \( \alpha_i \cap \alpha_{i+1} \) cannot be repeatedly scanned.

If \( \beta_i - \beta_{i+1} = \emptyset \), then the algorithm does nothing in iteration \( i \), implying that the premises of the lemma do not apply. (For the sake of clarity, we remark that since \( \beta_i \) and \( \beta_{i+1} \) must extend between two points on \( P \), if \( \beta_i - \beta_{i+1} = \emptyset \) is empty then it must imply that \( \beta_i = \beta_{i+1} \). But then, since \( CA_i \subseteq CA_{i+1} \), and since \( \alpha_i - \alpha_{i+1} \) lies in the interior of \( CA_{i+1} \) (except for one of its endpoints), for any point \( p \in \beta_i \), \( p \) must be closer to \( CP(p, \alpha_{i+1}) \) than to \( CP(p, \alpha_i) \). Thus the results from some iteration \( j > i \) would supersede that of iteration \( i \) in any case. Therefore, not doing anything in iteration \( i \) is justified.)

So we assume that \( l_i \) does intersect both \( \alpha_i \) and \( \beta_i \). Let the two points of intersection be \( p_i' \) and \( q_i'' \), respectively. Note that \( \alpha_i - \alpha_{i+1} \) terminates at \( p_i' \). Due to the convexity of \( \beta_i \), it is easy to see that for any \( q \in \beta_i \), \( CP(q, \alpha_i - \alpha_{i+1}) \) must equal \( p_i' \), implying that the rest of \( \alpha_i - \alpha_{i+1} \) is irrelevant for the search for an endpoint of \( SN_i \). Let \( q_i' = CP(p_i', \beta_i) \). If \( q_i' \notin \beta_i - \beta_{i+1} \), then iteration \( i \) would terminate after having reached \( q_i'' \) and without having traversed any portion of \( \alpha_i \cap \alpha_{i+1} \) or \( \beta_i \cap \beta_{i+1} \). Consequently, the lemma would be trivially true. If \( q_i' \notin \beta_i - \beta_{i+1} \), then consider the portion of the chain \( \beta_i \) between \( q_i' \) and \( q_i'' \). Let
$p''_i = \text{CP}(q''_i, \alpha_i \cap \alpha_{i+1})$. Some portion of the chain $\alpha_i \cap \alpha_{i+1}$ from $p'_i$ to $p''_i$ may be traversed in iteration $i$. Our goal now is to prove that this subchain will not be traversed in a later iteration. Note that for this case, we may assume that $\beta_i \cap \beta_{i+1}$ is not traversed in iteration $i$.

This situation is shown by an example in Fig. 17. The dashed polygonal chain through $p'_i$ and $p''_i$ is $\alpha_i + 1$, while portion of the chord $l_i$ terminating at $p'_i$ is $\alpha_i - \alpha_{i+1} + 1$. The chain $\beta_i$ is shown as a dashed chain through $q'_i$ and $q''_i$ and also includes a portion of another chord $l_j$ $(j > i)$. Finally, $\beta_i + 1 - \beta_i$ is the portion of $l_j$ terminating at $q''_i$.

The point $q''_i$ is the intersection of two chords, $l_i$ and $l_j$. Note that $\text{CP}(p, \alpha_i \cap \alpha_{i+1})$ cannot lie on the subchain of $\alpha_i \cap \alpha_{i+1}$ from $p'_i$ to $p''_i$. This is because one could construct a hypothetical convex chain that consists of the subchain of $\beta_i$ between $q'_i$ and $q''_i$ concatenated with the segment $q''_i p$, and then it is easy to see that $\text{CP}(p, \alpha_i \cap \alpha_{i+1})$ cannot lie between $p'_i$ to $p''_i$. We have thus shown that no repeated traversal of the chains occur between iterations $i$ through $j$. For any iteration $l > j$, the chain $\alpha_i$ does not intersect any portion of $\alpha_i$ between $p'_i$ and $p''_i$, since it is required to lie in the C-polygon corresponding to chord $l_j$. (Note that chord $l_j$, by assumption, passes through $q''_i$ and has a clockwise endpoint counterclockwise of the counterclockwise endpoint of $l_i$ and thus its C-polygon cannot intersect any portion of $\alpha_i$ between $p'_i$ and $p''_i$.)

That completes the proof of this lemma. \(\square\)

Once a local minimum for $SN_i$ is found in the $i$th iteration with one of the endpoints of $SN_i$ on $\alpha_i - \alpha_{i+1}$ or $\beta_i - \beta_{i+1}$, the algorithm verifies if it is a global minimum for the shortest segment between $\alpha_i$ and $\beta_i$. If the endpoints of $SN_i$ on $\alpha_i - \alpha_{i+1}$ and on $\beta_i - \beta_{i+1}$ are not the endpoints of either of $\alpha_i - \alpha_{i+1}$ or $\beta_i - \beta_{i+1}$, then clearly the global minimum for the shortest segment between $\alpha_i$ and $\beta_i$ must be the segment $SN_i$. Otherwise, a simple test can check whether the global minimum has been reached or not. This can be done by doing infinitesimal movements (in both directions) on one of the chains to see if the shortest segment from that point is shorter or longer than $SN_i$. If it is not a global minimum, then $SN_i$ can be ignored since the shortest segment between $\alpha_i$ and $\beta_i$ connects points that are not on $\alpha_i - \alpha_{i+1}$ as well as $\beta_i - \beta_{i+1}$. Since such a segment would connect $\alpha_{i+1}$ and $\beta_{i+1}$ it
will be encountered in a later iteration. The algorithm with the minor modifications mentioned above is guaranteed to sweep every portion of the \( \alpha \) and \( \beta \) chains exactly once and hence achieves the claimed linear-time complexity.

As mentioned in the overview in Section 4, it is possible that the segment \( \overline{s_it} \) discovered by the algorithm in the \( i \)th iteration, may not lie entirely within \( P \). To identify this situation, we exploit the fact that given a point \( x \) on \( P \), the chords algorithm has already identified which directions from \( x \) give rise to weakly-visible chords. Hence to check whether \( \overline{s_it} \) lies in \( P \), the algorithm computes the endpoint of the chord (as described in the next paragraph) generated when the line segment \( \overline{s_it} \) is extended towards \( \alpha_i \). Let the endpoints be \( p_i \) and \( q_i \). Using the output of the chords algorithm our algorithm checks whether the chord in the direction \( \overrightarrow{p_iq_i} \) is a weakly-visible chord. If the chord is not weakly visible, then the segment \( \overline{s_it} \) is ignored, and will be handled by the second phase of the algorithm (corresponding to Case 2). Otherwise the segment is returned as \( SN_i \), a potential candidate for the shortest illuminating line segment.

How is the endpoint \( p_i \) of the segment \( \overline{s_it} \) generated? It should be pointed out that it is possible that \( p_i \) may not lie on \( A_i \), but may lie on some other segment \( A_j \). Since both \( j < i \) and \( j \geq i \) are possibilities, we check if the line obtained by extending segment \( \overline{s_it} \) intersects \( A_i \). If it does, then the intersection point is the required endpoint \( p_i \). This is due to the fact that the subpolygon \( CA_i \) within which \( p_i \) lies, does not wholly contain a component. If the extension of \( \overline{s_it} \) does not intersect \( A_i \), then we can easily determine if \( j < i \) or \( j > i \) by checking whether the endpoints of \( A_i \) are clockwise or counterclockwise of the line. Once this direction is determined, the algorithm traverses from \( A_i \) to \( A_j \) along \( P \) (in the clockwise or counterclockwise direction, as the case may be) to locate \( p_i \). To understand the \( O(n) \) time complexity, we will show that this portion of \( P \) is not traversed again for this purpose. This is proved in Lemma 7 below.

The intuition behind the claim is that if \( p_i \) lies on \( A_j \) then \( SN_i \) is also the shortest segment between \( \alpha_j \) and \( \beta_j \) as well as between \( \alpha_i \) and \( \beta_i \) for all values of \( l \) between \( i \) and \( j \).

**Lemma 7.** If \( SN_i = \overline{s_it} \) lies inside \( P \) and on the chord \( \overrightarrow{p_iq_i} \) with \( p_i \in A_j \) for some \( i \neq j \), then \( SN_i = SN_l \) (i.e., it is also the shortest segment between \( \alpha_i \) and \( \beta_i \)) for all values of \( l \) between \( i \) and \( j \).

**Proof.** We first prove that under the above assumptions \( s_i \) lies on \( \alpha_j \) and that \( t_i \) lies on \( \beta_j \). It is clear that if \( p_i \in A_j \) then \( q_i \in B_j \), since the chord \( \overrightarrow{p_0q_i} \) is a weakly-visible chord. Assume for the sake of contradiction that \( s_i \) does not lie on \( \alpha_j \). Since \( \overrightarrow{p_0q_i} \) is a weakly-visible chord, it must intersect \( \alpha_j \). Let the intersection point be \( p \). Let \( s \) be a segment of \( \alpha_j \) on which \( p \) lies. Let the corresponding chord be \( c \), and the corresponding C-polygon be \( C \). If \( p \) lies on \( \overrightarrow{p_0s} \), then \( C \) does not intersect \( SN_i \), which contradicts the assumption that it is a weakly-visible segment. Hence \( p \) must lie on \( \overrightarrow{p_0s} \). \( s_i \) lies on \( \alpha_i \). Let the segment of \( \alpha_i \) on which \( s_i \) lies be \( s' \), with the corresponding chord and C-polygon being \( c' \) and \( C' \), respectively. Clearly \( C' \) contains \( A_j \), but does not contain \( p \), which is a contradiction, since \( p \) is supposed to lie in the intersection of all C-polygons that contain \( A_j \). Hence \( s_i \) must lie on \( \alpha_j \), which implies that \( s_i \) lies on \( \alpha_j \) for every value of \( l \) between \( i \) and \( j \). Similarly we prove that \( t_i \) lies on \( \beta_i \). Since the \( \alpha \) and \( \beta \) chains are convex, it is clear that \( SN_i \) must be the shortest segment joining \( \alpha_i \) and \( \beta_i \), for all values of \( l \) between \( i \) and \( j \). \( \square \)

The above lemma guarantees that in each iteration once \( SN_i \) is computed, it takes only linear (over all iterations) time to compute the intersection of the extensions of \( SN_i \) with the polygon \( P \). The next step is to check whether the directions specified by \( \overrightarrow{p_0s} \) gives rise to a weakly-visible chord. This is done by scanning through the linear-sized output of the chords algorithm, which again takes linear time over all iterations.
7. Case 2. Shortest tangential weakly-visible segment

This case occurs when the interior of the shortest weakly-visible segment in the polygon touches a vertex of the polygon. However, in this case, the corresponding weakly-visible chord obtained by extending the segment is also a tangential chord, i.e., it touches a vertex of the polygon in its interior. The crucial point to observe is that these are exactly the weakly-visible chords that are output by the linear-time chords algorithm [11]. A suitable modification of the chords algorithm can output all tangential weakly-visible segments, of which the shortest can be computed.

As is detailed in [11], the chords algorithm uses the following strategy. It traverses along the polygon in a counterclockwise direction with a point \( x \). When \( x \) is on \( A_i \), the points \( y(x) \) and \( z(x) \) on \( B_i \) corresponding to the other endpoints of the two tangential chords from \( x \) are computed. The points \( y(x) \) and \( z(x) \) move monotonically on \( P \); so do the points of tangency for the tangential chords, namely \( s(x) \) and \( t(x) \). The points of tangency \( s(x) \) and \( t(x) \) lie on the convex envelopes of the side chains \( D_i \) and \( E_i \). Note that the side chains are the chains left over if \( A_i \) and \( B_i \) are removed from \( P \). As \( x \) moves on \( A_i \), there are several possible events that can take place, which would change the description of the tangents: the point \( y(x) \) (or \( z(x) \)) could move to a vertex of \( P \); the point \( s(x) \) (or \( t(x) \)) could move to a vertex of \( P \). These events cause a recomputation of the equations of the tangential chords as a function of \( x \). In [11] it was shown that the number of these events are \( O(n) \), thus resulting in a linear-time algorithm.

The modification for computing the tangential weakly-visible segments is as follows. During iteration \( i \), the chains \( \alpha_i \) and \( \beta_i \) are computed. When the point \( x \) is on \( A_i \), the points of intersection of the tangential chords with \( \alpha_i \) and \( \beta_i \) are also maintained (call them \( a_1(x), a_2(x), b_1(x), b_2(x) \)). The segment from \( a_1(x) \) to \( b_1(x) \) and the segment from \( a_2(x) \) to \( b_2(x) \) are the two tangential weakly-visible segments with respect to \( x \). The situation is described in Fig. 18. There are, however, an additional number of events that could cause a change in the description of the tangential weakly-visible segments: the points \( a_1(x) \) or \( a_2(x) \) (or \( b_1(x) \) or \( b_2(x) \)) could move to a vertex of \( \alpha_i \) (or \( \beta_i \)). This would cause additional recomputations of the equations as well as the lengths of the tangential segments. The crucial point is that in between events, the length of the tangential segments can be computed in terms of \( x \), from which the minimum can be computed for that interval in constant time. Das et al. [11] showed that the points \( s(x) \) and \( t(x) \) move

![Fig. 18. Case 2: determining tangential shortest weakly-visible segments.](image-url)
monotonically along the envelopes of the side chains. Consequently, \( a_1(x), b_1(x), a_2(x) \) and \( b_2(x) \) also move monotonically on the \( \alpha \) and \( \beta \) chains. Each event caused by the tangential chord passing over a vertex of the \( \alpha \) and \( \beta \) chains is such that a particular tangential chord passes over each vertex of the envelope only once over the entire algorithm. Since there are \( O(n) \) vertices on the envelopes overall, the total number of events encountered is \( O(n) \).

This completes the description of all the pieces of the algorithm for computing in linear time the shortest weakly internally visible line segment of a simple polygon (if one exists).

8. All minimal weakly-visible segments algorithm

One of the by-products of our algorithm is a linear-time algorithm to generate all minimal weakly-visible segments of a polygon. This algorithm is a modification of the algorithm described in Section 7 for computing the shortest tangential weakly-visible segment. It outputs a set of pairs \((U_i, V_i), i = 1, \ldots, m\). Here \( U_i \) and \( V_i \) are subchains of the polygonal chains \( \alpha \) and \( \beta \), and any segment joining points \( u \in U_i \) and \( v \in V_i \) is a minimal weakly-visible segment. One note of caution is that \( U_i \) and \( V_i \) have left and right endpoints that are linear functions of a parameter \( x \) in a spirit similar to that of the endpoints of the chain \( B_i \) that is output by the chords algorithm. For a point \( x \) on \( A_i \), the polygonal chains \( \alpha_i \) and \( \beta_i \) can be computed along with the points \( a_1(x), a_2(x) \in \alpha_i \) and \( b_1(x), b_2(x) \in \beta_i \). The output of the algorithm consists of \((U_i, V_i) = ((a_1(x), a_2(x)), (b_2(x), b_1(x)))\). The discussion at the end of Section 7 can also be used to show that the number of these pairs produced is \( m = O(n) \). Lemma 1 can be used to show that these segments are minimal in the sense that any subsegment of these segments is not weakly visible.

9. Conclusion and open problems

We show optimal linear-time algorithms to compute the shortest weakly-visible segment and all minimal weakly-visible segments in a given simple polygon. One extension of this problem that has been solved is that of finding the shortest watchman route [6] in a simple polygon in polynomial time.

Some interesting open questions are:

- Can the exhaustive sweeping techniques from this paper be used to solve other weak visibility problems efficiently? For example, are there linear-time algorithms for the all-pairs version of any of the 2-guard walk problems (see [10])?
- Ntafos [23] introduced the notion of \( d \)-visibility, where an observer’s visibility is limited to distance \( d \). Can the shortest illuminating segment be computed efficiently under \( d \)-visibility?
- Can the central link segment be computed in \( O(n) \) time, as conjectured by Aleksandrov et al. [1]?

Acknowledgements

The authors thank the diligent referees for pointing out many errors in an earlier draft and for helpful suggestions that made the paper more readable.
References