Near Optimal Line Segment Queries in Simple Polygons

Mojtaba Nouri Bygi[a], Mohammad Ghodsi[a,b]

[a] *Computer Engineering Department, Sharif University of Technology, Iran*
[b] *School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran*

Abstract

This paper considers the problem of computing the weak visibility polygon (*WVP*) of any query line segment pq (or *WVP*(pq)) inside a given simple polygon P. We present an algorithm that preprocesses P and creates a data structure from which *WVP*(pq) is efficiently reported in an output sensitive manner.

Our algorithm needs $O(n^2 \log n)$ time and $O(n^2)$ space in the preprocessing phase to report *WVP*(pq) of any query line segment pq in time $O(|WVP(pq)| + \log^2 n + \kappa \log^2(\frac{n}{\kappa}))$, where κ is an input and output sensitive parameter of at most $|WVP(pq)|$. We improve the preprocessing time and space of current results for this problem [11, 7] at the expense of more query time.

Keywords: Computational Geometry, Visibility, Line Segment Visibility

1. Introduction

Two points inside a polygon P are *visible* to each other if their connecting segment remains completely inside P. The *visibility polygon* (*VP*) of a point q inside P (or *VP*(q)) is the set of vertices of P that are visible from q. There have been many studies on computing *VP*'s in simple polygons. In a simple polygon P with n vertices, *VP*(q) can be reported in time $O(\log n + |VP(q)|)$ by spending $O(n^3 \log n)$ time and $O(n^3)$ of preprocessing space [2]. This result was later improved by [1] where the preprocessing time and space were reduced to $O(n^2 \log n)$ and $O(n^2)$ respectively, at the expense of more query time of $O(\log^2 n + |VP(q)|)$.

The visibility problem has also been considered for line segments. A point v is said to be *weakly visible* from a line segment pq if there exists a point $w \in pq$ such that w and v are visible to each other. The problem of computing the *weak visibility polygon* of pq (or *WVP*(pq)) inside P is to compute all points of P that are weakly visible from pq. If P is simple (with no holes), Chazelle and Guibas [3] gave an $O(n \log n)$ time algorithm for this problem. Guibas *et al.* [10] showed that this problem can be solved in $O(n)$ time if a triangulation of P is given along with P. Since any P can be triangulated in $O(n)$ [4], the algorithm of Guibas *et al.* always runs in $O(n)$ time [10]. Another linear time solution was obtained independently by [13].

The *WV* problem in the query version has been considered by few. It was shown in [2] that a simple polygon P can be preprocessed in $O(n^3 \log n)$ time and $O(n^3)$ space such that, given an arbitrary query line segment inside P, $O(k \log n)$ time is required to recover k weakly visible vertices. This result was later improved by [1] in which the preprocessing time and space were reduced to $O(n^2 \log n)$ and $O(n^2)$ respectively, at expense of more query time of $O(k \log^2 n)$. In a recent work, we presented an algorithm to report *WVP*(pq) of any pq in $O(\log n + |WVP(pq)|)$ time by spending $O(n^3 \log n)$ time and $O(n^3)$ space for preprocessing [11]. Later, Chen and Wang considered the same problem and, by improving the preprocessing time of the visibility algorithm of Bose *et al.* [2], they improved the preprocessing time to $O(n^3)$ [7]. Alternatively, they showed how to build a data structure of size $O(n)$ in time $O(n)$ which can answer weak visibility queries in $O(k \log n)$ time [8].

In this paper, we show that *WVP* of a line segment pq can be reported in near optimal time of $O(|WVP(pq)| + \log^2 n + \kappa \log^2(n/\kappa))$, after preprocessing the input polygon in time and space of $O(n^2 \log n)$ and $O(n^2)$ respectively, where κ is an input and output sensitive parameter of at most $|WVP(pq)|$. Compared to the algorithms of [11] and [7], the storage and preprocessing time has one fewer linear factor, at expense of more query time. Our approach is inspired by Aronov *et al.* algorithm for computing the visibility polygon of a point [1]. In Section 3, we first show how to compute the *partial weak visibility polygon* *PWVP*(pq). Then, in Section 4, we use a balanced triangulation to compute and report the final weak visibility polygon.

2. Preliminaries

In this section, we introduce some basic terminologies used throughout the paper. For a better introduction to these terms, we refer the readers to Guibas *et al.* [10], Bose *et al.* [2], and Aronov *et al.* [1]. For simplicity, we assume that no three vertices of the polygon are collinear.

Email addresses: nouribygi@ce.sharif.edu (Mojtaba Nouri Bygi), ghodsi@sharif.edu (Mohammad Ghodsi)

2.1. Visibility decomposition

Let P be a simple polygon with n vertices. Also, let p and q be two points inside P. The *visibility* of a point p is the cyclical sequence of vertices and edges of P that are visible from p. A *visibility decomposition* of P is to partition P into a set of *visibility regions*, such that, for each region, any point inside the region has the same visibility, which we call the visibility of the region. This partition is induced by the *critical constraint edges.* A critical constraint edge is a line segment inside and limited by the boundary of P, which passes through two vertices of P, and is *tangent* to P at one (or both) of these two vertices. A line is tangent to P at a vertex v if it passes through v and is on the same side of P before and after the pass.

In a simple polygon, the visibility of two *neighboring* visibility regions which are separated by an edge, differ only in one vertex. This fact is used to reduce the space complexity of maintaining the visibility of the regions [2]. This is done by defining the *sink regions.* A sink is a region with the smallest visibility compared to all of its adjacent regions. Therefore, it is sufficient to maintain the visibility of the sinks, from which the visibility of all other regions can be computed. By constructing a directed dual graph over the visibility regions (see Figure 1), one can maintain the difference between the visibility of the neighboring regions [2].

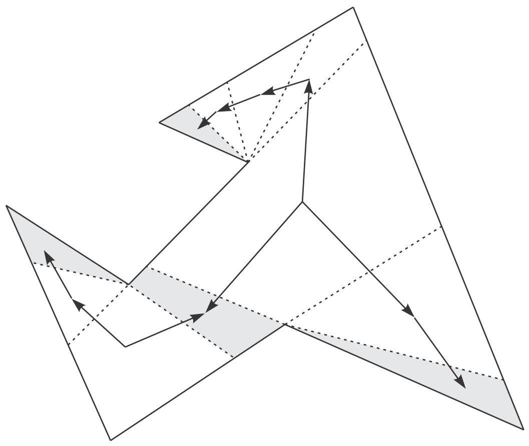


Figure 1: The visibility decomposition induced by the critical constraint edges and its dual graph . The sink regions are shown in gray.

In a simple polygon with n vertices, the number of visibility and sink regions are $O(n^3)$ and $O(n^2)$, respectively [2].

2.2. A linear time algorithm for computing WVP

Here, we present the $O(n)$ time algorithm of Guibas *et al.* for computing $WVP(pq)$ of a line segment pq inside P, as described in [6]. This algorithm is used in computing the partial weak visibility polygons in an output sensitive way, to be explained in Section 3.2. For simplicity, we assume that pq is a convex edge of P, but we will show that this can be extended for any line segment in the polygon.

Let $SPT(p)$ denote the shortest path tree in P rooted at p. The algorithm traverses $SPT(p)$ using a DFS and checks the turn at each vertex v_i in $SPT(p)$. If the path makes a right turn at v_i, then we find the descendant of v_i in the tree with the largest index j (see Figure 2). As there is no vertex between v_j and v_{j+1}, we can compute the intersection point z of v_jv_{j+1} and v_kv_i in $O(1)$ time, where v_k is the parent of v_i in $SPT(p)$. Finally the counter-clockwise boundary of P is removed from v_i to z by inserting the segment v_iz.

Let P' denote the remaining portion of P. We follow the same procedure for q. This time, the algorithm checks every vertex to see whether the path makes its first left turn. If so, we will cut the polygon at that vertex in a similar way. After finishing the procedure, the remaining portion of P' would be the $WVP(pq)$.

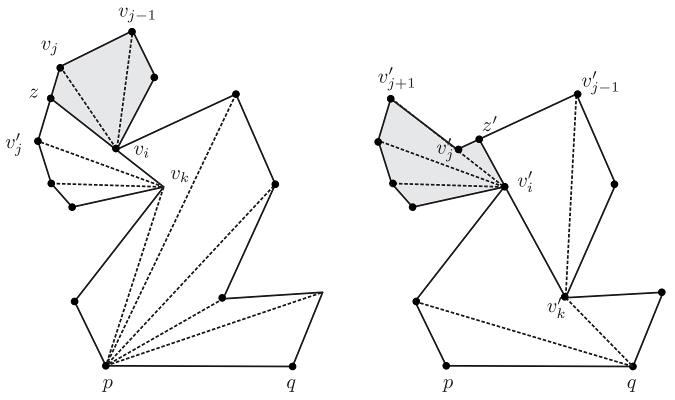


Figure 2: The two phases of the algorithm of computing $WVP(pq)$. In the left figure, the shortest path from p to v_j makes a first right turn at v_i. In the right figure, the shortest path from q to v'_j makes a first left turn at v'_i.

3. Computing the partial WVP

Suppose that a simple polygon P is divided by a diagonal e into two parts, L and R. Inspired by Aronov *et al.* algorithm for computing the visibility polygon of a point [1], for a query line segment $pq \in R$, we define the *partial weak visibility polygon* $WVP_L(pq)$ (or $PWVP_L(pq)$ for clarity) to be the polygon $WVP(pq) \cap L$. In other words, $WVP_L(pq)$ is the portion of P that is weakly visible from pq *through* e. In this section, we will show how to compute $WVP_L(pq)$. Later in Section 4, we will use this structure to compute $WVP(pq)$.

We will show how to use the algorithm of Guibas *et al.* [10] to compute $WVP_L(pq)$. To do this, we preprocess the polygon so that we can answer the visibility query in an output sensitive way. The idea is to compute the visibility decomposition of the polygon and, for each decomposition cell, compute the potential shortest path tree structures.

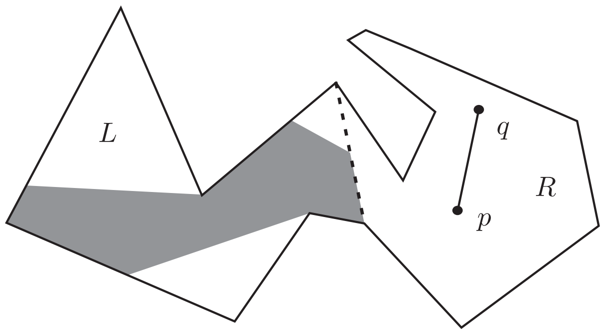


Figure 3: The partial weak visibility polygon of the segment pq is defined as the part of the sub-polygon L that is weakly visible from pq.

As the number of visibility regions is $O(n^3)$, the preprocessing cost of our approach would be high.

To overcome, we only consider the critical constraint edges that cut e. The number of such constraint edges is $O(n)$ and the complexity of the decomposition is reduced to $O(n^2)$. This decomposition can be computed in $O(n^2)$ time. We call this decomposition the *partial visibility decomposition* of P with respect to e. The remaining part of this section shows how to modify the linear algorithm of Guibas *et al.* [10] so that $WVP_L(pq)$ can be computed in an output sensitive way. First, we show how to compute the shortest path trees, and then present our algorithm for computing $WVP_L(pq)$.

3.1. Computing the partial $SPT_L(p)$

We define the *partial shortest path tree* $SPT_L(p)$ to be the subset of $SPT(p)$ that lead to a leaf node in L. In other words, $SPT_L(p)$ is the union of the shortest paths from p to all the vertices of L. In this section, we show how to preprocess the polygon P, so that for any given point $p \in R$, any part of $SPT_L(p)$ can be traversed in an output sensitive way. The shortest path tree $SPT_L(p)$ is composed of two kinds of edges: the *primary edges* that connect the root p to its direct visible vertices, and the *secondary edges* that connect two vertices of $SPT_L(p)$. Notice that if a point p crosses a critical constraint and that constraint does not cut e, then the structure of $SPT_L(p)$ would not change. Therefore, we can have *bent* primary edges that connect p to a visible vertex from e (see Figure 4).

For the secondary edges, we define two types of edges: the *1st type secondary edges* (1st type for short) are those secondary edges that are connected to a primary edge, and the *2nd type secondary edges* are the remaining secondary edges.

We can compute the primary edges of SPT_L by using Aronov's output sensitive algorithm of computing the partial visibility polygon [1]. More precisely, with a processing cost of $O(n^2 \log n)$ time and $O(n^2)$ space, giving a point p in query time, a pointer to the sorted list of the vertices that are visible to p can be computed in $O(\log n)$ time.

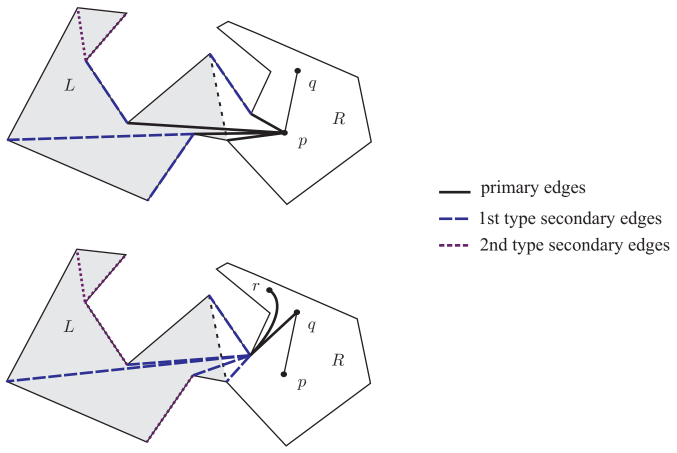


Figure 4: SPT_L for different points of R. Notice that as q and r are on the same visibility region w.r.t. L, $SPT_L(q)$ and $SPT_L(r)$ have the same structure.

It is also necessary to compute the list of the secondary edges of every vertex of SPT_L. Each vertex r in SPT_L have $O(n)$ possible 2nd type edges. Depending on the parent of r, a sub-list of these edges would appear in SPT_L. To store all the possible 2nd type edges of r, we compute and store this sub-list, or to be precise, the starting and ending edges of the list, for all the possible parents of r. As there are $O(n)$ possible parents for a vertex, these calculations can be performed for all the vertices of the polygon in total time of $O(n^2 \log n)$ and the data can be stored in $O(n^2)$ space. Having these data, we can, in the query time, access the list of the 2nd type edges of any vertex in constant time.

We build the same structure for the 1st type edges. The parent of a 1st type edge is the root of the tree. As the root can be in any of the $O(n^2)$ different visibility regions, computing and storing the starting and ending edges in the list of 1st type edges of a vertex cost $O(n^3 \log n)$ time and $O(n^3)$ space.

We can reduce the time and space needed to compute and store these structures, having this property that two adjacent regions have only $O(1)$ differences in their 1st type edges.

Lemma 1. *Consider a visibility region V in the polygon and suppose that the 1st type secondary edges are computed for a point p in this region. For a neighboring region that share a common edge with V, these edges can be updated in constant time.*

Proof. When a view point p crosses the border of two neighboring regions, a vertex becomes visible or invisible to p [2]. In Figure 5 for example, when p crosses the border specified by u and v, a 1st type secondary edge of u becomes a primary edge of p, and the 2nd type edges that start from v become 1st type secondary edges. It can be seen that no other vertex is affected by this movement. Processing these changes can be done in constant time, since it includes the following changes: removing a secondary edge of u (uv), adding a primary edge (pv), and moving an array pointer (edges of v) from the 2nd type edges of uv to the 1st type edges of pv. Note that we know the exact positions of these elements in their corresponding lists. Finally, the only edge which involves in these changes can be identified in the preprocessing time (the edge corresponding to the crossed critical constraint), so, the time we spent in the query time would be $O(1)$.

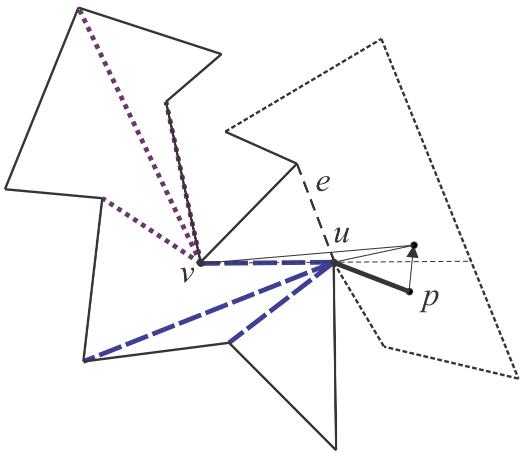


Figure 5: When p enters a new visibility region, the combinatorial structure of $SPT_L(p)$ can be maintained in constant time.

Having this fact and using a *persistent data structure*, e.g. *persistent red-black tree* [12], we can reduce the cost of storing the 1st type edges by a linear factor. A persistent red-black tree is a red-black tree that can remember all its intermediate versions. If a set of n linearly ordered items are stored in the tree and we perform m update into it, any version t, for $1 \leq t \leq m$, can be retrieved in time $O(\log n)$. This structure can be constructed in $O((m+n)\log n)$ time by using $O(m+n)$ space.

Theorem 2. *A simple polygon P can be processed into a data structure with $O(n^2)$ space and in $O(n^2 \log n)$ time so that for any query point p, the shortest path tree from p can be reported in $O(\log n + k)$, where k is the size of the tree that is to be reported.*

Proof. First, we use Aronov's algorithm for computing the partial visibility polygon of p. For this, $O(n^2)$ space and $O(n^2 \log n)$ time is needed in the preprocessing phase. For the secondary edges, $O(n^2 \log n)$ time and $O(n^2)$ space is needed to compute and store these edges. Also, a point location structure is built on top of the arrangement.

In the query time, the partial visibility region of p can be located in $O(\log n)$ to have the sorted list of the visible vertices from p. As the visible vertices from p correspond to the primary edges of SPT_L, we also have the primary edges of $SPT_L(p)$.

For the 1st type edges, a tour is formed to visit all the cells of the partial visibility decomposition. From Lemma 1, we can start from an arbitrary cell, walk along the tour, and construct a persistent red-black tree on the 1st type edges of SPT_L of a point in each cell. As there are $O(n^2)$ cells and, each cell has $O(n)$ 1st type edges, the structure takes $O(n^2)$ storage and can be built in $O(n^2 \log n)$ preprocessing time. Having this structure, the 1st type edges of the cell containing p can be retrieved from the persistent data structure in $O(\log n)$ time.

Finally, at each node of the tree, we have the list of 2nd type edges from that node. Therefore, the cost of traversing SPT_L is the number of visited nodes of the tree, plus the initial $O(\log n)$ time. In other words, the query time is $O(\log n+k)$, where k is the number of the traversed edges of the SPT_L.

3.2. Computing $WVP_L(pq)$

Now that we showed how to compute $SPT_L(p)$ for any point $p \in R$ in the query time, we can use the linear algorithm presented in Section 2.2 for computing WVP of a simple polygon and modify it to compute $WVP_L(pq)$ in an output sensitive way. As we can see in Figure 6, the algorithm can be extended to the cases that pq is not a polygonal edge.

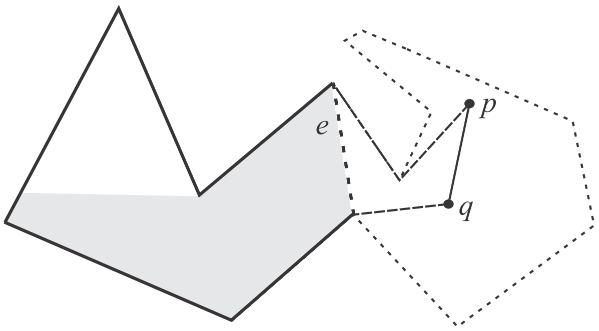


Figure 6: In computing $WVP_L(pq)$ we can assume pq to be a polygonal edge.

The idea is to change the algorithm of Section 2.2 and make it output sensitive. We store some additional information about the vertices of the polygon in the preprocessing time, so that we can somehow merge the two phases of the algorithm.

We say that a vertex $v \in L$ is *left critical* (LC for short) with respect to a point $q \in R$, if $SP(q, v)$ makes its first left turn at v or one of its ancestors. In other words, each shortest path from p to a non-LC vertex is a convex chain that makes only clockwise turns at each node. The *critical state* of a vertex is whether it is LC or not. If we have the critical state of all the vertices of L with respect to a point q, we say that we have the *critical information* of q. Note that as we check the right turns at the first phase of the algorithm, we do not need to store the right turn state of the vertices in the preprocessing time.

The outline of our algorithm is as follows: Having the line segment pq, in the first round, we traverse $SPT_L(p)$

using DFS. At each vertex of $SPT_L(p)$, we check whether this vertex is left critical with respect to q or not, using the critical information of q. If so, we are sure that the descendants of this vertex are not visible from pq. Therefore, we postpone its processing to the time it is reached from q, and check other branches of $SPT_L(p)$. Otherwise, we proceed with the algorithm and check whether $SPT_L(p)$ makes a right turn at this vertex. In the second phase, we traverse $SPT_L(q)$ and perform the normal procedure of the algorithm, and process the parts of the polygon that we reached in the first phase.

Remark 1. All the traversed vertices in $SPT_L(p)$ and $SPT_L(q)$ are vertices of $WVP_L(pq)$.

In the preprocessing phase, we compute the critical information of a point inside each region, and assign this information to that region. In the query time and upon receiving a line segment pq, we find the regions of p and q. Using the critical information of these two regions, the above algorithm can be applied to compute $WVP_L(pq)$.

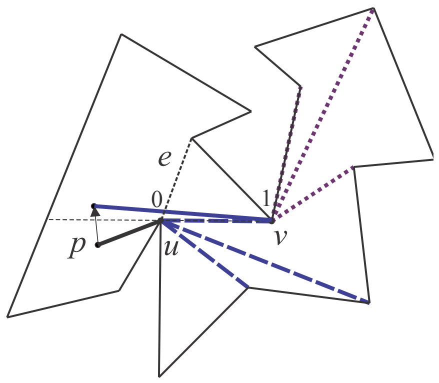
As there are $O(n^2)$ visibility regions in the partial visibility decomposition, $O(n^3)$ space is needed to store the critical information of the vertices of L. For each region, we compute SPT_L of a point and, by traversing the tree, we update the critical information of each vertex with respect to this region. An array of size $O(n)$ is assigned to each region to store these information. We also build the structure described in Section 3.1 to compute SPT in $O(n^3 \log n)$ time and $O(n^3)$ space. In the query time, we locate the visibility regions of p and q in $O(\log n)$ time. By Remark 1, when we proceed the algorithm in SPT_Ls of p and q, we only traverse the vertices of $WVP_L(pq)$. Finally, as the processing time spent in each vertex is $O(1)$, the total query time is $O(\log n + |WVP_L(pq)|)$.

To improve this result, we use the fact that any two adjacent regions have $O(1)$ differences in their critical information.

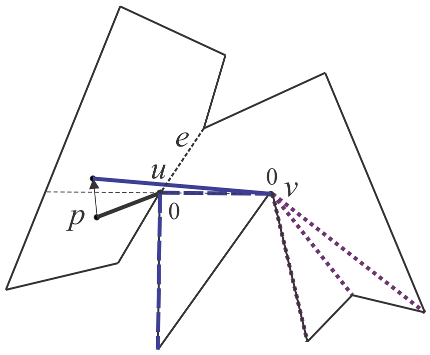
Lemma 3. *In the path between neighboring visibility regions, the changes of the critical information can be handled in constant time.*

Proof. Suppose that we want to maintain the critical information of p and p is crossing the critical constraint defined by uv, where u and v are two reflex vertices of P. Recall that by critical information, we mean whether the vertices of P are left critical w.r.t. p or not. As stated in Section 3.2, we do not store the right turn states of the vertices of P and the right turns will be checked at the query time.

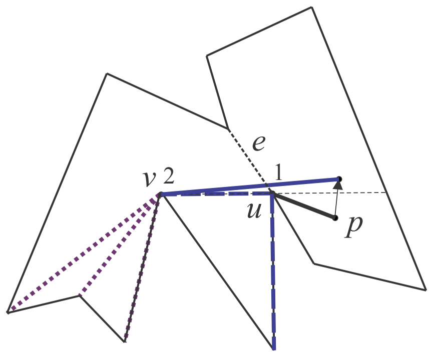
The only vertices that are affected directly by this change are u and v. Depending on the critical states of u and v w.r.t. p, four situations may occur (see Figure 7). In the first three cases, the critical state of v will not change. In the forth case, however, the critical state of v will change. Before the cross, the shortest path makes a left turn at u, therefore, both u and v are LC w.r.t. p.



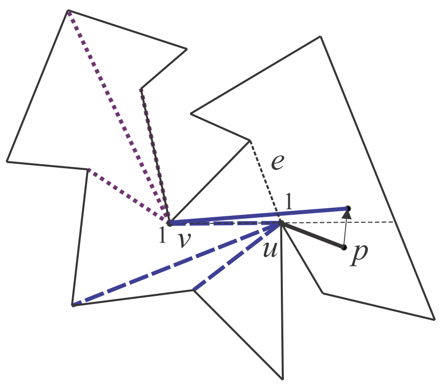
(a) v is LC but u is not.



(b) u and v are not LC.



(c) both u and v are LC.



(d) u is LC but v is not.

Figure 7: Changes in the critical state of v w.r.t. p, as p moves between the two regions.

However, after the cross, v is no longer LC. This means that the critical state of all the children of v in the $SPT_L(p)$ may change as well.

To handle these cases, we use a lazy updating method to propagate these changes across the tree. To do this, we modify the way that we store the critical information of each vertex w.r.t. p. At each vertex v, we store two additional values: the *critical number*, which is the number of LC vertices we met in the path $SP(p, v)$ from p, and the *debit number*, which is the critical number that is to be propagated in the vertex subtree. It is clear that if a vertex is LC, it means that its critical number must be greater than zero (see Figure 8). Also, if a vertex has a debit number, the critical numbers of all its children must be added by this debit number. Notice that computing and storing these numbers along the critical information will not change the time and space requirements. Also, in query time, we can update the critical number of a vertex in $O(1)$ time, while traversing the tree.

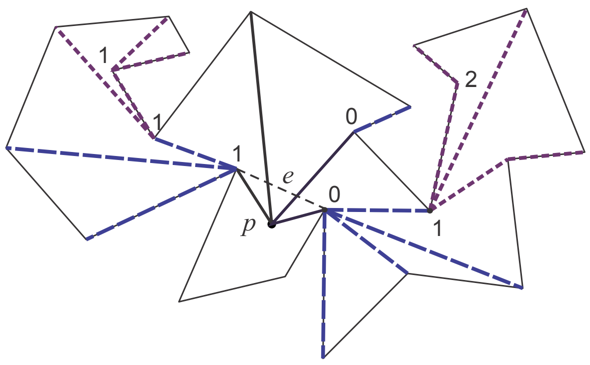


Figure 8: The critical number represents the number of the left critical vertices met from p in $SPT_e(p)$.

Having these new data, we must update these numbers in the third and forth cases in Figure 7. For example, let us consider the forth case. When v becomes visible to p, it is no longer LC w.r.t. p. Therefore, the critical number of v is changed to 0. However, instead of changing the critical numbers of all the children of v, we set the debit number of v to -1, indicating that the critical numbers of all the vertices of its subtree must be subtracted by 1. The actual propagation of this subtraction will happen at query time, when $SPT_L(p)$ will be traversed. Similarly, If p moves in the reverse path, i.e., when v becomes invisible to p, we handle the tree in a same way by adding 1 to the debit number, and propagating this addition in the query time.

A persistent data structure can be used to reduce the costs to $O(n^2 \log n)$ preprocessing time and $O(n^2)$ storage. We form a tour visiting all the cells and construct a persistent red-black tree on the critical information and the 2nd type edges of all the nodes. The structure takes $O(n^2)$ storage and can be built in $O(n^2 \log n)$ preprocessing time. In addition, we build a point location structure on top of the arrangement, which can be done in $O(n^2)$ time and $O(n^2)$ space [9].

Theorem 4. *Given a polygon P and a diagonal e which cuts P into two parts, L and R, and using $O(n^2 \log n)$ time, we can construct a data structure of size $O(n^2)$ so that, for any query line segment $pq \in R$, the partial weak visibility polygon $WVP_L(pq)$ can be reported in $O(\log n + |WVP_L(pq)|)$ time.*

3.3. Computing $WVP_L(pq)$ for Extended Line Segments

Here, we define the concept of the partial weak visibility polygon for an extended line segment. Assume that P is a polygon of size n and P' is a sub-polygon of P (see Figure 9). The size of P' is m and it is divided by diagonal e' to two sub-polygons L' and R'. Also, assume that the line segment pq is on the right side of e', and is cutting R'. As pq is not completely inside R', we call it an extended line segment for P'. We can also define the weak visibility polygon of and extended line segment in a similar way.

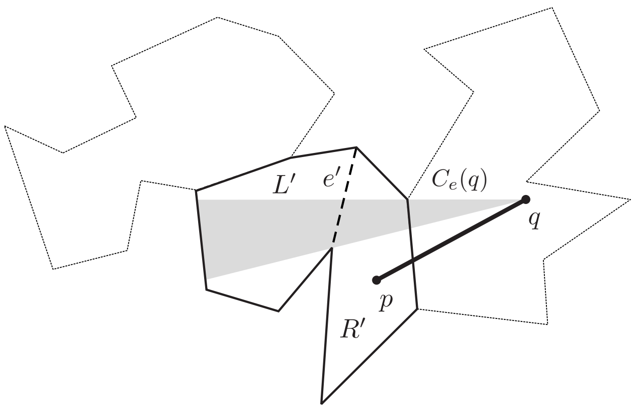


Figure 9: We can compute $WVP_{L'}(pq)$ for an extended line segment in a sub-polygon of size m, in time $O(\log m + k')$, with a preprocessing of time $O(m^2 \log m)$ and $O(m^2)$ space, by preprocessing the main polygon in $O(n)$ time and space.

It can be shown that by preprocessing the polygon P in $O(n)$ time and space, for any sub-polygon P', we can preprocess it in time $O(m^2 \log m)$ and build a data structure of size $O(m^2)$ space, so that we can answer the partial weak visibility queries for extended line segments in time $O(k' + \log m)$, where k' is the size of the output.

As the algorithm of Section 3.2 relies on $SPT(p)$ and $SPT(q)$, if we manage to compute them in the specified times, we can compute $WVP_{L'}(pq)$. On the other hand, computing $SPT(p)$ relies on the Aronov's algorithm of computing the partial visibility polygon [1]. This algorithm can be altered to accept an *extended point* p, which can be defined in a similar way as the extended line segments. In other words, if we build a data structure of size $O(n)$ in time $O(n)$, we can compute the $C_e(p)$ in $O(\log n)$ time, where $C_e(p)$ is the infinite cone with apex p and delimited by the endpoints of the visible portion of e'.

The other parts of our algorithm is based on the visibility decomposition of the sub-polygon L', which has $O(m^2)$ complexity.

This result will be used in the next section, when we want to compute the weak visibility polygon of a line segment.

4. Computing *WVP* by a balanced triangulation

There is always a diagonal e of a simple polygon that cuts P into two pieces, each having at most $2n/3$ vertices [5]. We can recursively subdivide and build a balanced binary tree, where the leaves are triangles and each interior node i corresponds to a subpolygon P_i and a diagonal e_i. Each diagonal e_i divides P_i into two subpolygons, L_i and R_i, which respectively correspond to the left and right subtrees of i (see Figure 10). We build the data structures described in Section 3 for L_i and R_i with respect to e_i.

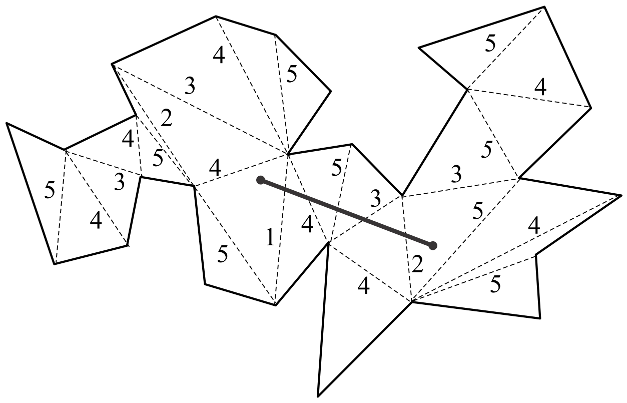


Figure 10: A balanced binary triangulation of the polygon is built so that the the weak visibility polygon can be computed recursively.

To compute $WVP(pq)$, p and q will be located among the leaf triangles. In the simplest case, both p and q belong to the same triangle (see Figure 11). First we explain this situation. We construct $PWVP_i(pq)$ for each i from the leaf to the root. Here, $PWVP_i(pq)$ is the partial weak visibility polygon of pq in P_i with respect to e_i. For the leaf node, it is the corresponding triangle, and for other nodes, it can be computed inductively. In each step, the merging of the computed polygons can be done in $O(\log n)$ time.

The space and time needed for building an exterior visibility decomposition of a simple polygon with m vertices are $O(m^2)$ and $O(m^2 \log m)$, respectively. Thus, the inductive procedure can be expressed as the following equations:

$$S(n) = \max_{n/3 \leq m \leq 2n/3} (S(m) + S(n-m)) + \Theta(n^2),$$

$$T(n) = \max_{n/3 \leq m \leq 2n/3} (S(m) + S(n-m)) + \Theta(n^2 \log n)$$

Therefore, $S(n) = \Theta(n^2)$, and $T(n) = \Theta(n^2 \log n)$. With the same analysis as in [1], we can calculate the query

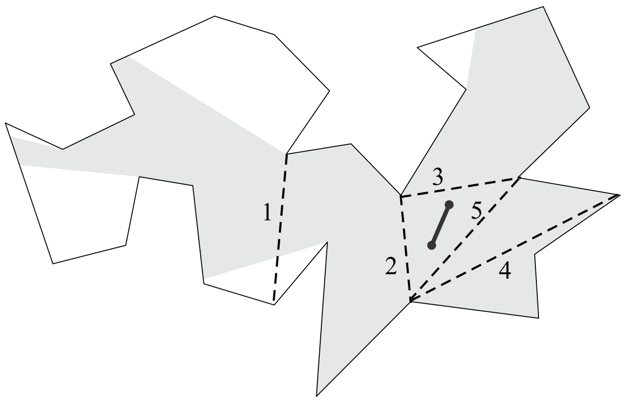


Figure 11: The base case in computing the weak visibility polygon a query line segment.

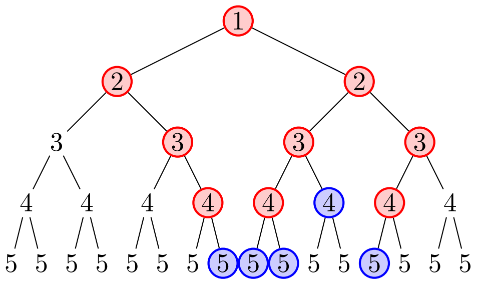
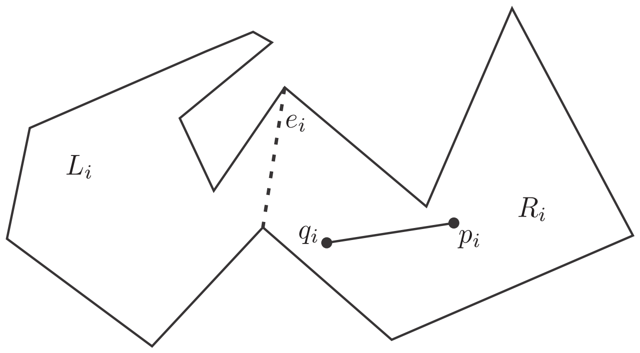


Figure 12: The specified nodes correspond to the computed partial *WVP*s in Figure 10.

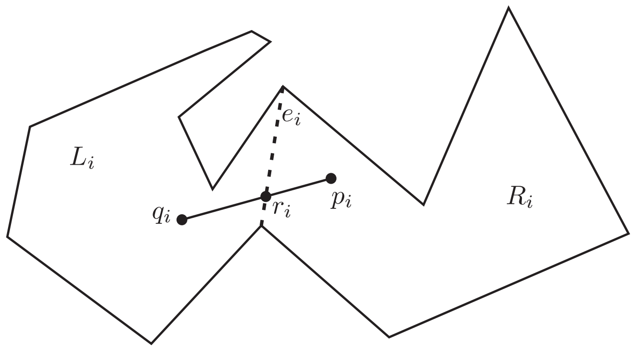
time. Two point locations can be done in $O(\log n)$ time. As the triangulation is balanced, any path from the root to a leaf node has $O(\log n)$ length. As we showed in Theorem 4, the time needed to compute $PWVP_i(pq)$ at step i is $O(\log n + |PWVP_i(pq)|)$. Also, the merging at each step can be done in $O(\log n)$ time. Therefore, the total query time is $O(\log n + \sum_i (\log n + |PWVP_i(pq|))$, or $O(\log^2 n + |WVP(pq)|)$.

The tricky part is when p and q are on different triangles. Assume that at step i, the query line segment is p_iq_i and it is in the sub-polygon P_i. The sub-polygon P_i is divided by diagonal e_i to two sub-polygons L_i and R_i. If p_iq_i does not intersect e_i, without loss of generality, assume that p_iq_i is located in R_i (see Figure 13a). We do the normal procedure of the algorithm and compute $PWVP_{L_i}(p_iq_i)$. We continue to recursively compute weak visibility polygon on R_i. In this case, the time needed by this step can be expressed as $T(n_i, p_iq_i) = T(n_i/2, p_iq_i) + O(\log n_i) + |PWVP_{L_i}(p_iq_i)|$.

On the other hand, if p_iq_i and e_i intersect at point r_i, without loss of generality, assume that p_i is in R_i and q_i is in L_i (see Figure 13b). We can express $WVP(p_iq_i)$ as the union of two weak visibility polygons on extended line segment p_iq_i (see Section 3.3): the weak visibility polygon of p_iq_i in R_i, and the weak visibility polygon of p_iq_i in L_i. In other words, we must compute two weak visibility subproblems. Having these two visibility polygons, the union of them can be merged in time $|WVP_{P_i}(p_iq_i)|$. According



(a) This case can be phrased as $WVP(p_iq_i) = PWVP_{L_i}(p_iq_i) + WVP_{R_i}(p_iq_i)$.



(b) This can be phrased as $WVP(p_iq_i) = WVP_{R_i}(p_iq_i) + WVP_{L_i}(p_iq_i)$.

Figure 13: The induction step can be categorized as one of these situations.

to the Theorem 4, the query time spent at step i can be expressed as: $T(n_i, p_iq_i) = 2T(n_i/2, p_iq_i) + O(\log n_i)$.

The preprocessing costs of the algorithm is the same as before. For the query time, a naive analysis leads to the recursive equation $T(n) = 2T(n/2) + O(\log n)$, or $T(n) = O(n)$. But we show how to express the query time in terms of the number of times that the line segment cuts the diagonals, by analyzing the recursion tree of the equations. If pq intersects the diagonals of the triangulation at κ steps, we have κ branches in the balanced binary tree associated with the triangulation. For a constant κ, the maximum cost will happen if these branches are near the root. Otherwise, if there is a binary node in the tree and it has an ancestor node with one child, we can replace these nodes and move the binary equation upward, and obtain a new tree that have more processing cost.

Figure 14 depicts this case. As you can see, it consists of a complete balanced tree with κ nodes, and its height is $\log \kappa$. There are also κ paths in the tree with length $\log n - \log \kappa = \log \frac{n}{\kappa}$. The first part of the tree corresponds to the equation $T(m) = 2T(m/2) + \log m$. Also, each path in the second part corresponds to the equation $T(m) = T(m/2) + \log m$, with the staring value of $m = n/\kappa$.

We can expand the recursion tree as the following equa-

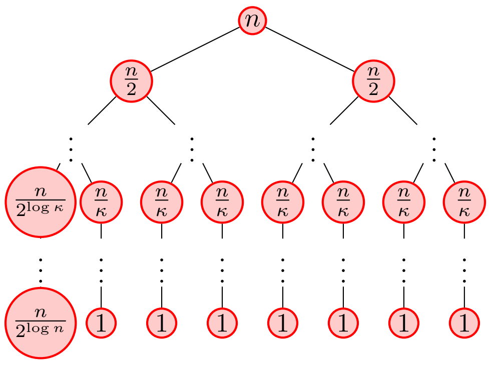


Figure 14: The worst case scenario in computing $WVP(pq)$. It consists of a complete balanced tree with height $\log \kappa$, and κ paths with length $\log n - \log \kappa = \log \frac{n}{\kappa}$.

tions:

$$\begin{aligned} T(n) &= \kappa \sum_{i=1}^{n/\kappa} \log i + \sum_{i=0}^{\log_2 \kappa} 2^i \log(n/2^i) \\ &= \kappa \log^2 \frac{n}{\kappa} + \log n \sum_{i=0}^{\log_2 \kappa} 2^i - \sum_{i=0}^{\log_2 \kappa} i2^i \end{aligned}$$

By plugging in the geometric sequence summation formula and the following formula

$$\sum_{k=1}^{n} ka^k = \frac{a(1-a^n)}{(1-a)^2} - \frac{na^{n+1}}{1-a}$$

we get

$$\begin{aligned} T(n) &\leq \kappa \log^2(n/\kappa) + \kappa \log n - \kappa \log \kappa \\ &= \kappa \log^2(n/\kappa) + \kappa \log(n/\kappa) \end{aligned}$$

As each intersection of the query line segment with a diagonal can be mapped to a unique vertex in $WVP(pq)$, we have $\kappa \leq |WVP(pq)|$. Therefore, in total, the query time is $O(|WVP(pq)| + \log^2 n + \kappa \log^2(n/\kappa))$.

In summary, we have the following theorem:

Theorem 5. *A simple polygon P can be processed in time $O(n^2 \log n)$ into a data structure of size $O(n^2)$ so that, for any query line segment pq, $WVP(pq)$ can be reported in time $O(|WVP(pq)| + \log^2 n + \kappa \log^2(n/\kappa))$ where $\kappa \leq |WVP(pq)|$.*

5. Conclusion

In this paper, we showed how to answer the weak visibility queries in a simple polygon with n vertices in an efficient way. In the first part of the paper, we defined the partial weak visibility polygon $WVP_e(pq)$ of a line segment pq with respect to a diagonal e and presented an algorithm to report it in time $O(\log n + |WVP_e(pq)|)$, by spending $O(n^2 \log n)$ time to preprocess the polygon and maintaining a data structure of size $O(n^2)$.

In the second part, we presented a data structure of size $O(n^2)$ which can be computed in time $O(n^2 \log n)$ so that the weak visibility polygon $WVP(pq)$ from any query line segment $pq \in P$ can be reported in time $O(|WVP(pq)| + \log^2 n + \kappa \log^2(n/\kappa))$. Here, κ is the number of triangles that intersect pq in a balanced triangulation of P, and $\kappa \leq |WVP(pq)|$.

References

[1] B. Aronov, L. J. Guibas, M. Teichmann and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete and Computational Geometry*, 27(4):461-483, 2002.

[2] P. Bose, A. Lubiw, J. I. Munro. Efficient visibility queries in simple polygons. *Computational Geometry: Theory and Applications*, 23(3):313-335, 2002.

[3] B. Chazelle and L. J. Guibas. Visibility and intersection problems in plane geometry. *Discrete and Computational Geometry*, 4(6):551-581, 1989.

[4] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485-524, 1991.

[5] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339-349, 1982.

[6] S. K. Ghosh. Visibility Algorithms in the Plane. *Cambridge University Press*, New York, NY, USA, 2007.

[7] D. Z. Chen and H. Wang. Weak visibility queries of line segments in simple polygons. In *23rd International Symposium, ISAAC*, pages 609–618, 2012.

[8] D. Z. Chen and H. Wang. Weak Visibility Queries of Line Segments in Simple Polygons. *CoRR* abs/1212.6039, 2012.

[9] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28-35, 1983.

[10] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209-233, 1987.

[11] M. Nouri Bygi and M. Ghodsi. Weak visibility queries in simple polygons. In *Proc. 23rd Canad. Conf. Comput. Geom.*, 2011.

[12] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669-679, 1986.

[13] G. T. Toussainta A linear-time algorithm for solving the strong hidden-line problem in a simple polygon. *Pattern Recognition Letters*, 4:449-451, 1986.

Appendix A. Analyzing the Term $\kappa \log^2(n/\kappa)$

The query time of our algorithm is $O(|WVP(pq)| + \log^2 n + \kappa \log^2(n/\kappa))$. κ is the number of intersections of the query line segment with the diagonals of the balanced triangulation. Although $\kappa = O(|WVP(pq)|)$, usually κ is much smaller than $|WVP(pq)|$.

For small values of κ, $\kappa \log^2(n/\kappa) = O(\log^2 n)$, and for large value of it, $\kappa \log^2(n/\kappa) = O(\kappa) = O(|WVP(pq)|)$. The diagram of Figure A.15 gives a sense of the behaviour of the term $\kappa \log^2(n/\kappa)$. The derivative of this term for a constant n, in terms of κ, is $\log^2 n + (-2\log\kappa - 2)\log n + \log^2\kappa + 2\log\kappa$. A simple calculation shows that the equation makes its maximum value at $\kappa = \exp^{-2} n$, and the maximum value is 4κ. As for the higher values of κ, $\kappa \log^2(n/\kappa)$ decreases, we can say that for $\kappa \geq \exp^{-2} n$, $\kappa \log^2(n/\kappa) = O(\kappa) = O(|WVP(pq)|)$. In other words, for $\kappa \geq \exp^{-2} n$, we can express the processing time of the algorithm simply as $O(|WVP(pq)| + \log^2 n)$.

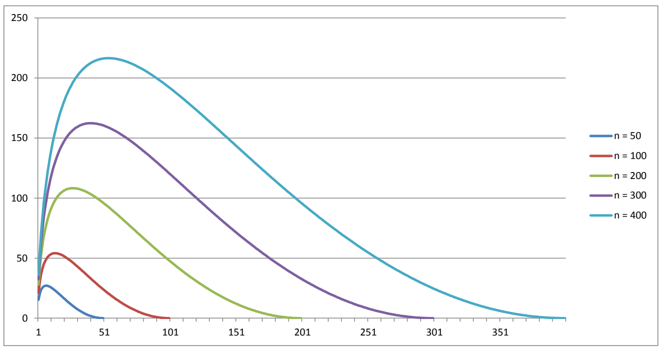


Figure A.15: Diagram of $\kappa \log^2(n/\kappa)$ for some values of n. The horizontal axis is κ and the vertical axis is the value of $\kappa \log^2(n/\kappa)$. At each curve, the maximum value is achieved at $\kappa = \exp^{-2} n$ and the maximum value at this point is 4κ.