Weak Visibility Queries in Simple Polygons

Mojtaba Nouri Bygi [*]   Mohammad Ghodsi [†]

Abstract

In this paper, we consider the problem of computing the weak visibility (WV) of a query line segment inside a simple polygon. Our algorithm first preprocesses the polygon and creates data structures from which any WV query is answered efficiently in an output sensitive manner. In our solution, the preprocessing is performed in time $O(n^3 \log n)$ and the size of the constructed data structure is $O(n^3)$. It is then possible to report the WV polygon of any query line segment in time $O(\log n + k)$, where k is the size of the output. Our algorithm improves the current results for this problem.

1 Introduction

Two points inside a polygon are *visible* to each other if their connecting segment remains completely inside the polygon. The *visibility polygon* $VP(q)$ of a point q in a simple polygon P is the set of P points that are visible from q. A common approach to this problem is to decompose the polygon into the *visibility regions* in such a way that all points inside a region have equivalent visibility data [2]. Two visibility polygons are equivalent if they are composed of the same sequence of vertices and edges of the underlying polygon. If all the visibility regions and their corresponding visibility polygons are calculated in the preprocessing phase, for any point q, $VP(q)$ can then be obtained by refining the visibility polygon of the region that contains q.

In a simple polygon with n vertices, $VP(q)$ can be reported in time $O(\log n + |VP(q)|)$ by spending $O(n^3 \log n)$ preprocessing time and $O(n^3)$ space [2, 7]. An improvement was presented in [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* to 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* (or *WVP*) of pq inside a polygon P is to compute all points of P that are weakly visible from pq. If P is a polygon without holes, Chazelle and Guibas [3] gave an $O(n \log n)$ time algorithm for this problem. Guibas *et al.* [6] showed that this problem can be solved in $O(n)$ time if a triangulation of P is given along with P. Since P can be triangulated in $O(n)$ [4], the algorithm of Guibas *et al.* runs in $O(n)$ time [6]. Another linear time solution was obtained independently in [8].

The weak visibility problem in the query version has been considered by few. It is 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 the polygon, $O(k \log n)$ time is required to recover k weakly visible vertices. This result was later improved in [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 $O(k \log^2 n)$.

In this paper, we improve these results by showing that the weak visibility polygon of a line segment pq can be reported in an output sensitive time of $O(\log^2 n + k)$ after preprocessing the input in time and space of $O(n^3 \log n)$ and $O(n^3)$ respectively.

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.* [6], Bose *et al.* [2], and Aronov *et al.* [1]. For simplicity, we assume that no three vertices of the polygon are collinear.

2.1 Visibility Decomposition

Let P be a simple polygon with n vertices. Also let p and q be two points in the polygons. A *visibility decomposition* of P is to partition P into a set of *visibility regions*, such that for each region, the same sequence of vertices and edges of P are visible from any point inside the region.

Two visibility regions are *neighboring* if they are separated by an edge. In a simple polygon, two neighboring visibility regions differ only in one vertex in their visibility sequences. This fact is used to reduce the space complexity of maintaining the visibility sequences of the regions [2]. This is done by defining the *sink regions*. A sink is a region with the smallest visibility sequence

[*]Department of Computer Engineering, Sharif University of Technology, `nouribaygi@ce.sharif.edu`

[†]Computer Engineering Department, Sharif University of Technology, and Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. `ghodsi@sharif.edu`. This author's research was partially supported by the IPM under grant No: CS1389-2-01

compared to all of its adjacent regions. It is therefore sufficient to only maintain the visibility sequences of the sinks, from which the visibility sequences of all other regions can be computed. By constructing a directed dual graph (see Figure 1) over the visibility regions, one can maintain the difference between visibility sequences of neighboring regions[2].

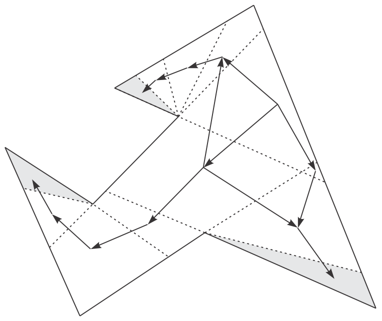
Figure 1: Decomposed visibility regions and its dual graph [2].

The number of visibility regions in a simple polygon is $O(n^3)$, and the number of sink regions is $O(n^2)$ [2].

2.2 Linear time algorithm for computing WVP

Here, we explain the $O(n)$ time algorithm of Guibas *et al.* [6] for computing $WVP(pq)$ of a line segment pq inside a simple polygon P with n vertices, as described in [5]. For any line segment pq, we can cut P into two polygons P_1 and P_2 along the supporting line of pq (see Figure 2). It can be seen that $WVP(pq)$ is the union of the WVPs of the two sub-polygons from pq. So here we assume that pq is an edge of P.

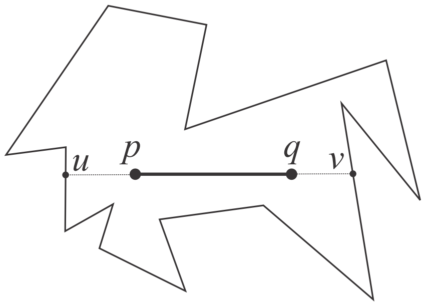

Figure 2: P is divided by uv into two sub-polygons.

Let $SPT(p)$ denote the shortest path tree in P rooted at p. We traverse $SPT(p)$ using a depth-first search and check the turn at every vertex v_i in $SPT(p)$. If the path $SP(p, v_j)$ 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 3). We compute the intersection point z of v_jv_{j+1} and v_kv_i, where v_k is the parent of v_i in $SPT(p)$, in $O(1)$ (because there is no vertex between v_j and v_{j+1}), and finally remove the counter-clockwise boundary of P 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, except that this time we check the turn at every vertex and see whether the path make its first left turn. After finishing the procedure, we output the remaining portion of P' as $WVP(pq)$.

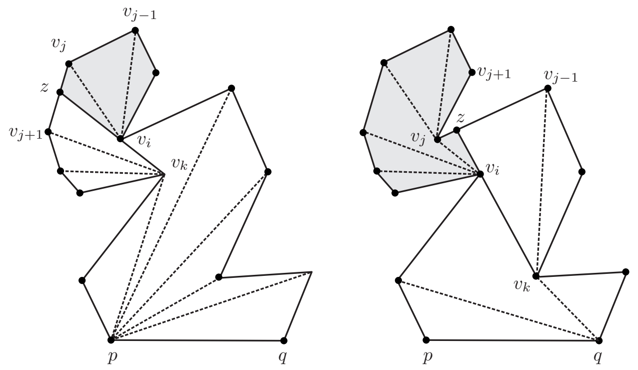

Figure 3: In both cases, the from the root makes its first right turn at v_j [5].

3 The Proposed Algorithm

In this section, we show how to modify the presented algorithm, so that the WVP can be computed efficiently in an output sensitive manner. First, we show how to compute the shortest path trees in an output sensitive manner, and then we present the first version of our algorithm. Finally, in Section 3.3, we improve this algorithm and present the final result.

3.1 Computing the shortest path trees

In our algorithm, we use both of the shortest path trees of p and q. In [6], it is shown how to compute the Euclidean shortest paths inside a simple polygon P of n vertices from a given point p to all other vertices in $O(n)$ time. But, this algorithm requires $O(n)$ of query time which is way beyond our goal. To overcome, we show how to preprocess a simple polygon, so that for any given point, we can compute any part of its shortest path tree in an output sensitive way.

The shortest path tree $SPT(p)$ is composed of two kinds of edges: the primary edges, which are from the root p to its direct visible vertices, and the secondary edges that connect other two vertices of polygons (see Figure 4).

We can compute the primary edges using the same output sensitive algorithm of computing the visibility polygon [2]. More precisely, with a processing cost of $O(n^3 \log n)$ time and $O(n^3)$ space, we can in query time of $O(\log n)$ have a pointer to the sorted list of the visible vertices from p in $O(\log n)$ time.

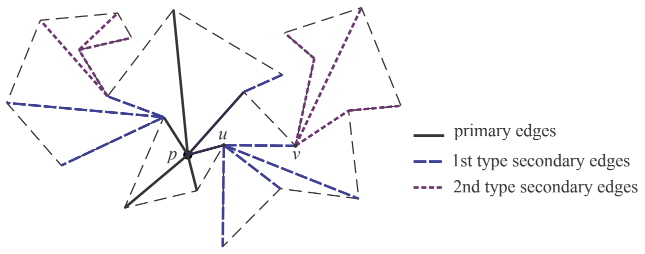

Figure 4: The shortest path tree from p and its different edge types.

We also need to access the list of the secondary edges of a node in constant time. For this, we compute all possible values of secondary edges of a vertex in preprocessing time and in query time, we detect the appropriate list without any further cost.

Depending on the number of possible parents of a vertex v, we recognize two kinds of secondary edges: The 1st type of secondary edges (1st type for short) are those connected to a primary edge, and the 2nd type are the ones that connect other two vertices of the polygon.

For the 2nd type edges, as there are $O(n)$ possible parents for v, and for each parent, there may be $O(n)$ edges emitting from v, we need $O(n^2)$ space to store all possible combinations of the 2nd type edges emitting from v. In total, we need $O(n^3)$ space to store all these edges. We can also compute a local shortest path tree in each case (a SPT with the parent of v as its root), and compute the sorted list of edges in $O(n \log n)$, or in total $O(n^3 \log n)$ time.

The parent of a 1st type edge is the root of the tree. As the root can be in any of the $O(n^3)$ different visibility regions, computing all possible combinations of the 1st type edges emitting from a vertex, requires to consider all these parents (remember that if the root of two SPTs are in the same region, then the combinatorial structure of the two trees are the same). We can compute the first type edges for each region in $O(n^4 \log n)$ time and store them in $O(n^4)$ space. In Section 3.3 we will show how to improve this result by a linear factor.

Theorem 1 *Given a simple polygon P, we can preprocess it into a data structure with $O(n^4)$ space and in $O(n^4 \log n)$ 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 Bose's algorithm for computing the visibility polygon of point p. For this, we need $O(n^3)$ space and $O(n^3 \log n)$ time in the preprocessing phase. For the secondary edges, we need $O(n^4 \log n)$ time and $O(n^4)$ space to compute and store the 1st type edges, and $O(n^3 \log n)$ time and $O(n^3)$ space to store the 2nd type ones.

In query time, we can locate the visibility region of p in $O(\log n)$ and have the sorted list of the visible vertices from p. Therefore, we can use the primary edges of $SPT(p)$ without paying any further costs (remember that each visible vertex from p corresponds to a primary edge in SPT).

As we have computed the 1st type edges of the SPT for all the regions, we can access a pointer to the sorted list of these edges in $O(1)$. Similarly, at any node of the tree, we have the list 2nd type edges from that node. Therefore, the cost of traversing the SPT would be the number of visited nodes of the tree, plus the initial $O(\log n)$ cost, i.e., $O(\log n + k)$, where k is the number of the traversed edges of SPT. □

3.2 Computing the query version of WVP

In this section, we use the linear algorithm of Guibas *et al.* [6] for computing WVP of a simple polygon and show how to compute the query version of this problem. We build the data structure explained in previous section, so that we can compute the SPT of any point inside the polygon in query time.

This algorithm is not output sensitive by itself. See the example of Figure 5. As stated in Section 2.2, first we traverse $SPT(p)$ using DFS and check the turn at every vertex of $SPT(p)$. Consider vertex v. As we traverse the shortest path from p to v, or $SP(p, v)$, we must check all the children of v and this checking can costs $O(n)$. But when we traverse $SPT(q)$, v would be omitted, therefore, the time we spend for processing its children would be useless.

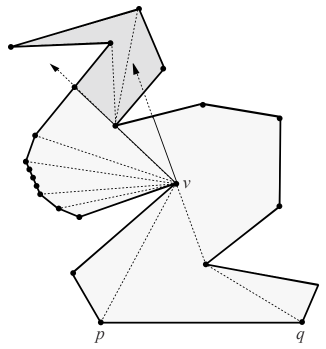

Figure 5: Processing vertex v imposes redundant $O(n)$ time.

To achieve an output sensitive algorithm, we store some additional information about the vertices of the polygon.

We say that a vertex v of a simple polygon is *left critical* (LC for short) with respect to a point p, if $SP(p, v)$ 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. Having the critical state of all vertices with respect to a point p, we say that we have the *critical information* with respect to p.

Having the critical information of p and q, we can change the algorithm for computing WVP as follows: In the first round, we traverse $SPT(p)$ using DFS. At each vertex, we check whether this vertex is left critical with respect to q. If so, we are sure that the descendants of this vertex are not visible from pq, so we postpone its processing to the time we reach it from q, and check the other branches of $SPT(p)$. Otherwise, we proceed with the algorithm and check whether $SPT(p)$ makes a right turn at this vertex. In the second round, we traverse $SPT(q)$ and perform the normal procedure of the algorithm.

Lemma 2 *All the vertices that we traverse in $SPT(p)$ and $SPT(q)$ are vertices of $WVP(pq)$.*

Proof. Assume that we meet v when we are traversing $SPT(p)$ and $v \notin WVP(pq)$. Also, assume that u is the parent of v in $SP(pv)$. Then, u or one of its ancestors must be LC with respect to q, otherwise the Guibas *et al.* algorithm will detect it as a WVP vertex. So, as one of the ancestors of v is LC, we would not reach v when traversing $SPT(p)$. The same argument applies to $SPT(q)$. $\square$

As the combinatorial structure of shortest path trees of all points in a visibility region are the same, we just need to compute the critical information of a point a in each region S, and use this information for all points of that region. In the preprocessing phase, for each visibility region we compute critical information of a point inside it, and assign this information to that region. In query time and upon receiving a line segment pq, we locate p and q. Using the critical information of their regions, we apply the above algorithm and compute $WVP(pq)$.

So far, we have assumed that pq is a polygon edge. The following lemma generalizes the position of pq in P.

Lemma 3 *If pq is a line segment inside the simple polygon P, we can decompose P into two sub-polygons P_1 and P_2, such that they both have pq as an edge. In addition, we can use the critical information and the secondary edges data of the visibility regions of P for these sub-polygons.*

Proof. We build the ray shooting structure in P, in $O(n)$ time and space [3]. In query time, we find the intersection points of the supporting line of pq with the border of P. Locating these intersection points among the vertices of P, we can create two simple polygons, P_1 and P_2, in $O(\log n)$ time . Each of these two polygons has pq on its edge. As the visibility regions of the generated polygons are a subset of the visibility regions of the original polygon, and we have computed the critical information and the SPT edges for all the regions of P, we have the needed data for p and q in both P_1 and P_2. See an example in Figure 6.

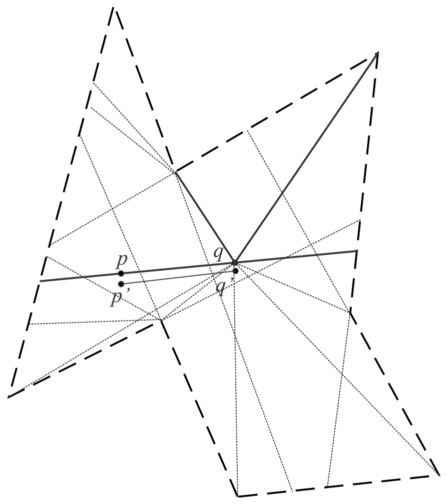

Figure 6: If the query line segment pq is inside the polygon, we split it along the supporting line of pq.

For simplicity, we can translocate pq a little higher (or lower) from its supporting line to $p'q'$. As the visibility regions of p and p' (also q and q') are the same, $WVP(pq)$ and $WVP(p'q')$ have the same combinatorial structures. We need to filter the primary edges originating from p and q to those that are in P_1 (or P_2). As we have the sorted list of these edges, this filtering can be done in $O(\log n)$ by a simple range searching. By traversing these primary edges at each vertex of P_1, we can use the stored critical information and secondary edges of that vertex. Depending on which side of the line pq we are on, we use the critical information of p or q in the weak visibility computations. $\square$

Now, we analyse the time and space of the above algorithm. As there are $O(n^3)$ visibility regions, we need $O(n^4)$ space to store the critical information of each vertex. For each region, we compute SPT of a point, and by traversing the tree, we update the critical information of each vertex with respect to this region. We assign an array of size $O(n)$ to each region to store these information. We also build the structure described in Section 3.1 for computing SPT in time $O(n^4 \log n)$ and $O(n^4)$ space. In query time, we locate the visibility regions of

p and q in $O(\log n)$. As we traverse SPTs of p and q, by Lemma 2, each vertex that we see is on $WVP(pq)$. Because the processing time we spend in each vertex is $O(1)$, the total query time is $O(\log n + |WVP(pq)|)$.

Theorem 4 *Using $O(n^4 \log n)$ time to preprocess a simple polygon P and maintain a data structure of size $O(n^4)$, it is possible to report $WVP(pq)$ in time $O(\log n + |WVP(pq)|)$.*

3.3 Improving the algorithm

To improve the result of Theorem 4, we will modify two parts of our algorithm. First, we show that it is sufficient to compute the critical information of the sink visibility regions (see Section 2.1), from which we can deduce the critical information of all other regions. Also, in computing SPT in Section 3.1, we will show that if we compute 1st type of secondary edges of the sink regions, we can compute these edges for the non-sink regions in query time. As there are $O(n^2)$ sinks in a simple polygon, the processing time and space of our algorithm would reduce to $O(n^3 \log n)$ and $O(n^3)$ respectively.

In query time, if both p and q belong to sink regions, we have critical information of both regions and we proceed the algorithm as stated before. On the other hand, if one of these points lie on a non-sink region, we show how to obtain the secondary edges and the critical information for that region in $O(\log n + |WVP(pq)|)$.

Lemma 5 *Consider two visibility regions that share a common edge. If we have the 1st type secondary edges of a region for each vertex visible from it, these edges are the same for its neighboring region, except for one edge.*

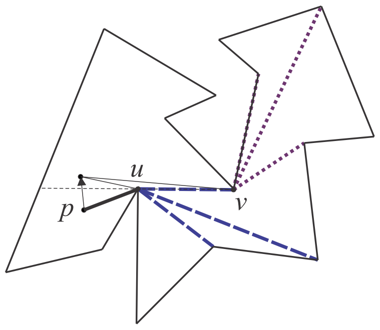

Figure 7: Combinatorial changes of SPT by moving between neighboring regions.

Proof. When we cross the border of two neighboring regions, a vertex becomes visible, or invisible [2]. In Figure 7 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 all edges of v become 1st type secondary edges. We can see that no other vertex would be 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 2nd type edges to 1st type edges. Note that we know the exact position of these elements, so we do not have the overhead time of finding them in their corresponding lists. Finally, we can identify the sole edge which involves with these changes in the preprocessing time (the edge corresponding to the crossed critical constraint), so, the time we spend in the query time would be $O(1)$. □

Lemma 6 *In the path from a sink to another visibility region, we can handle the changes of the critical information of the point in constant time.*

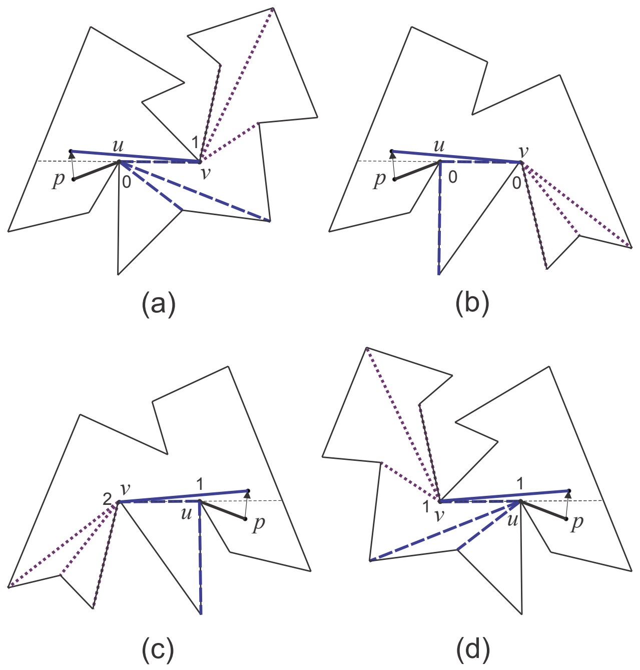

Figure 8: The critical information of v w.r.t p, as p moves between the two regions. a) v is LC but not u, b) u and v are not LC, c) both u and v are LC, d) u is LC but not v.

Proof. Suppose that we want to maintain the critical information of p and we are crossing the critical constraint defined by the edge uv. Depending on the critical status of u and v w.r.t. p, four possible situations may occur (see Figure 8). In the first three cases, the critical status of v will not change and no further action is required. In the forth case, however, the critical status of u will change. To handle this case, we modify the way we store the critical status of each vertex w.r.t. p. More precisely, at each vertex v we store the number of LC vertices we met, or *critical numbers*, in the path $SP(p,v)$ (see Figure 9). Computing and storing the critical numbers along the critical info will not change our time and space requirements. Now consider

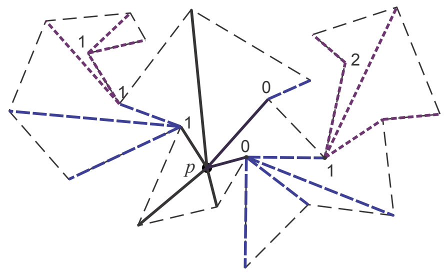

Figure 9: We store the number of LC vertices we met from p in $SPT(p)$.

the forth case in Figure 8. When v becomes visible to p, it is no longer LC w.r.t. p. So, we change the critical number of v to 0, but instead of changing the critical numbers of its children, we store -1 in v as its critical number, 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 when we are traversing $SPT(p)$. We also modify the query time algorithm to reflect this change. If we are computing $WVP(pq)$, and because v is LC w.r.t. q, we stopped at the path $SP(p,v)$, we store a pointer to this path at v. When we are traversing $SP(q,v)$ and we find out that v is no LC w.r.t. q, we resume the stored pass. $\square$

In the preprocessing time, we construct the dual planar graph of the visibility regions. We use the dual directed graph that was built by algorithm of Bose *et al.* [2] (Figure 1). In this graph, every node represents a visibility region, and an edge between two nodes corresponds to a gain of one vertex in the visibility set in one direction, and a loss in the other. By Lemma 5 and 6, we also know that these two neighboring regions have the same critical information and secondary edges, except for one vertex. We associate this vertex with the edge. We also compute the critical information and 1st type secondary edges of all the sink regions.

In query time, we locate the region containing point p, and follow any path from this region to a sink. As each arc represents one vertex seen by the query point p and therefore seen by pq, the number of arcs that we pass would be $O(|WVP(pq)|)$. When traversing the path from sink back to the region of p, we update the critical information and the secondary edges of the visible vertices in each region. Upon coming back to the original region, we would have the critical information and the secondary edges of this region. We perform the same procedure for q. Having the critical information and the 1st type edges of p and q, we can compute $WVP(pq)$ with the algorithm of Section 3.2. Putting all together, we have the following result

Theorem 7 *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 the polygon, it takes $O(\log n + |WVP(pq)|)$ time to list all vertices of $WVP(pq)$.*

4 Conclusion

In this paper, we showed how to answer weak visibility queries in a simple polygon in an efficient way. We presented an algorithm to report $WVP(pq)$ of any line segment pq in $O(\log n + |WVP(pq)|)$ time by spending $O(n^3 \log n)$ time to preprocess the polygon and maintaining a data structure of size $O(n^3)$.

Currently, we are working on a different approach for the same problem, to construct 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 $O(\log^2 n + |WVP(pq)|)$ time. Also, we are investigating whether our techniques can be extended to the cases of polygons with holes.

Acknowledgement

The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] B. Aronov, L. 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, and 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*, 4:551-581, 1989.

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

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

[6] 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.

[7] L. Guibas, R. Motwani, and P. Raghavan. The robot localization problem in two dimensions. *SIAM J. Comput.*, 26(4):11201138, 1997.

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