Space–Query-time Tradeoff for Computing the Visibility Polygon

Mostafa Nouri and Mohammad Ghodsi
nourybay@ce.sharif.edu, ghodsi@sharif.edu

[1] Computer Engineering Department
Sharif University of Technology
[2] IPM School of Computer Science

Abstract. Computing the visibility polygon, VP, of a point in a polygonal scene, is a classical problem that has been studied extensively. In this paper, we consider the problem of computing VP for any query point efficiently, with some additional preprocessing phase. The scene consists of a set of obstacles, of total complexity $O(n)$. We show for a query point q, $VP(q)$ can be computed in logarithmic time using $O(n^4)$ space and $O(n^4 \log n)$ preprocessing time. Furthermore to decrease space usage and preprocessing time, we make a tradeoff between space usage and query time; so by spending $O(m)$ space, we can achieve $O(n^2 \log(\sqrt{m}/n)/\sqrt{m})$ query time, where $n^2 \leq m \leq n^4$. These results are also applied to angular sorting of a set of points around a query point.

Key words: Visibility polygon, logarithmic query time, space–query-time tradeoff, $(1/r)$-cutting.

1 Introduction

The visibility polygon, or simply VP, of a point is defined as the set of all points of the scene that are visible from that point. In this paper we consider the problem of computing VP of a query point when the scene is a fixed polygonal scene. We can use a preprocessing phase to prepare some data structures based on the scene, then for any query point, we use these data structures to quickly compute VP of that point.

The problem of computing VP when the scene is a polygonal domain, is more challenging than for simple polygon scene, which has been studied extensively. In these problems, a scene with total complexity of $O(n)$ consists of a simple polygon with h holes. Without any preprocessing $VP(q)$ can be computed in time $O(n \log n)$ by algorithms of Asano [3] and Suri and O'Rourke [15]. This can also be improved to $O(n+h \log h)$ with algorithm of Heffernan and Mitchell [10].

Asano *et al.* [4] showed that with a preprocessing of $O(n^2)$ time and space, $VP(q)$ can be reported in $O(n)$ time. It is remarkable that even though $|VP|$ may be very small, the query time is always $O(n)$. Vegter [16] showed that with $O(n^2 \log n)$ time and $O(n^2)$ space for preprocessing, we can report $VP(q)$ in

an output sensitive manner in time $O(|VP(q)|\log(n/|VP(q)|))$. Pocchiola and Vegter [13] then showed that if the boundary polygon of the scene and its holes are convex polygons, using visibility complex, $VP(q)$ can be reported in time $O(|VP(q)|\log n)$ after preprocessing the scene in time $O(n\log n + E)$ time and $O(E)$ space, where E is the number of edges in the visibility graph of the scene. Very recently Zarei and Ghodsi [17] proposed an algorithm that reports $VP(q)$ in $O((1+\min(h,|VP(q)|))\log n + |VP(q)|)$ query time. The preprocessing takes $O(n^3\log n)$ time and $O(n^3)$ space.

One of the problems with the above mentioned algorithms in the polygonal scene, is the large query time in the worst cases. For example when $h = \Theta(n)$ and $|VP(q)| = \Theta(n)$ the algorithms of Pocchiola and Vegter [13] and Zarei and Ghodsi [17] degrade to the algorithms without any preprocessing time. For Vegter's algorithm, we can also generate instances of the problem that use more time than the required time to report $VP(q)$. Another problem with these algorithms, as far as we know, is that they cannot efficiently report some other properties of the VP, like its size without actually computing it. For example in Asano *et al.* [4], we should first compute $VP(q)$ to be able to determine its size.

In this paper, we have introduced an algorithm which can be used to report $VP(q)$ in a polygonal scene in time $O(\log n + |VP(q)|)$, using $O(n^4\log n)$ preprocessing time and $O(n^4)$ space. The algorithm can be used to report the ordered visible edges and vertices of the scene around a query point or the size of $VP(q)$ in optimal time $O(\log n)$. The solution of the algorithm can also be used for further preprocessing to solve many other problems faster than before (e.g., line segment intersection with VP).

Because the space used in the preprocessing is still high, we have modified the algorithm to obtain a tradeoff between the space and query time. With this modifications, using $O(m)$ space and $O(m\log(\sqrt{m}/n))$ time in the preprocessing, where $n^2 \leq m \leq n^4$, we can find the combinatorial representation (to be described later) of $VP(q)$ in time $O(n^2\log(\sqrt{m}/n)/\sqrt{m})$. If we need to report the actual $VP(q)$, additional $O(|VP(q)|)$ time is required in query time.

The above algorithms are also described for the problem of angular sorting of some points around a query point and the same results are also established for it, i.e., we can return a data structure containing the set of n points sorted around a query point in time $O(n^2\log(\sqrt{m}/n)/\sqrt{m})$ using $O(m)$ space and $O(m\log(\sqrt{m}/n))$ preprocessing time where $n^2 \leq m \leq n^4$. The actual sorted points can then be reported using $O(n)$ additional time.

The remaining of this paper is organized as follows: In Section 2 we first describe how we can sort n points around a query point in logarithmic time using a preprocessing. We then use this solution to develop a similar result for computing VP. In Section 3 we obtain a tradeoff between space and query time for both problems. Section 4 describes some of immediate applications of the new algorithms, and finally, Section 5 summarizes the paper.

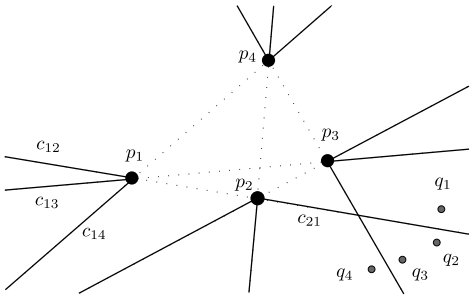


Fig. 1. Arrangement of critical constraints for $P = \{p_1, p_2, p_3, p_4\}$. q_1, q_2 and q_4 see the points in different angular orders, while q_2 and q_3 see in the same order.

2 Logarithmic query time

In this section we develop an algorithm with optimal query time for computing VP of a query point. Since the problem of angular sorting points is embedded in the computation of VP, we first consider it.

2.1 Angular sorting

Let $P = \{p_1, \ldots, p_n\}$ be a fixed set of n points and q, a query point in the plane. Let $\theta_q(p_i)$ denote the counterclockwise angle of the line connecting q to p_i with the positive y-axis. The goal is to find the sorted list of p_i in increasing order of $\theta_q(p_i)$. Let θ_q denote this sorted list.

The idea is to partition the plane into a set of disjoint cells, totally covering the plane, such that for all points in a cell r, the sorted sequences are the same. Let the sequence be denoted by θ_r. This partitioning is obtained by some rays, called *critical constraints*. A critical constraint c_{ij} lies on the line through the points p_i, p_j. It is the ray emanating from p_i in the direction that points away from p_j. That is, every line through two points contains two critical constraints (see Fig. 1). If q moves slowly, it can be shown that θ_q changes only when q crosses a critical constraint. When q crosses c_{ij}, p_i and p_j, which are adjacent in θ_q, are swapped. The following lemma, summarizes this observation, which is an adaptation of a similar lemma in [2] for our case.

Lemma 1. *Let $P = \{p_1, \ldots, p_n\}$ be a set of non-degenerate points in the plane, i.e., no three points lie on a line. Let q and q' be two points not on any critical constraint of points of P. The ordered sequences of points of P around q and q', i.e., θ_q and $\theta_{q'}$ differ iff they are on the opposite sides of some critical constraint.*

Proof. The lemma can be proved by a similar argument to that used by Aronov *et al.* [2] □

Let C denote the set of critical constraints c_{ij}, for $1 \leq i, j \leq n$, and $\mathcal{A}(C)$ denote the arrangement of the constraints in C. In Fig. 1 the arrangement of

critical constraints for a set of four points is illustrated. The complexity of $\mathcal{A}(C)$ is not greater than the complexity of the arrangement of $\binom{n}{2}$ lines, so $|\mathcal{A}(C)| = O(n^4)$. This implies the following corollary.

Corollary 1. *The plane can be decomposed into $O(n^4)$ cells, such that all points in a cell see the points of P in the same angular order.*

From the above corollary, a method for computing θ_q for a query point q can be derived. In the preprocessing phase, for each cell r in $\mathcal{A}(C)$, θ_r is computed and stored. Moreover $\mathcal{A}(C)$ is preprocessed for point location. Upon receiving a query point q, the cell r in which q lies is located, and θ_r is simply reported.

In spite of the simplicity of the algorithm, its space complexity is not satisfactory. The required space (resp. time) to compute the sorted sequences of points for each cell of $\mathcal{A}(C)$ is $O(n)$ (resp. $O(n \log n)$). So for computing θ_r for all the cells in the arrangement, $O(n^5)$ space and $O(n^5 \log n)$ time is required.

For any two adjacent cells r, r' in $\mathcal{A}(C)$, θ_r and $\theta_{r'}$ differ only in two (adjacent) position. To utilize this low rate of change and reduce the required time and space for the preprocessing, a persistent data structure, e.g., persistent red-black tree, can be used. A persistent red-black tree, or PRBT, introduced by Sarnak and Tarjan [14], is a red-black tree capable of remembering its earlier versions, i.e., after m updates into a set of n linearly ordered items, any item of version t of the red-black tree, for $1 \leq t \leq m$ can be accessed in time $O(\log n)$. PRBT can be constructed in $O((m+n)\log n)$ time using $O(m+n)$ space.

In order to use PRBT, a tour visiting all the cells of $\mathcal{A}(C)$ is needed. Using a depth-first traversal of the cells, a tour visiting all the cells and traversing each edge of $\mathcal{A}(C)$ at most twice is obtained. Initially, for an arbitrary cell in the tour, θ_r is computed from scratch and stored in PRBT. For the subsequent cells in the tour, we update PRBT accordingly by simply reordering two points. Finally, when all the cells of $\mathcal{A}(C)$ are visited, θ_r is stored in PRBT for all of them.

Constructing $\mathcal{A}(C)$ and the tour visiting all the cells, takes $O(n^4)$ time. Computing θ_r for the first cell needs $O(n \log n)$ time and $O(n)$ space, and subsequent updates for $O(n^4)$ cells need $O(n^4 \log n)$ time and $O(n^4)$ space. A point location data structure for $\mathcal{A}(C)$ is also needed that can be created in $O(n^4)$ time and space [5]. Totally the preprocessing phase is completed in $O(n^4 \log n)$ time using $O(n^4)$ space. The query time consists of locating the cell in which the query point q lies, and finding the related root of the tree for that cell. Each of these tasks can be accomplished in $O(\log n)$ time.

Theorem 1. *A set P of n points in the plane can be preprocessed in $O(n^4 \log n)$ time using $O(n^4)$ space, such that for any query point q, a pointer to a red-black tree which stores θ_q can be returned in time $O(\log n)$.*

2.2 Visibility polygon computation

In this section we consider the problem of computing VP in a polygonal scene. Let $O = \{o_1, \ldots, o_n\}$ be a set of line segments which may intersect each other

only at their end-points. These segments are the set of all edges of the polygonal scene. Let q be a query point in the scene. The goal is to compute $VP(q)$ efficiently using enough preprocessing time and space.

Since VP of each point is unique, the plane cannot be partitioned into cells in which all points have equal VP. But we can decompose it into cells such that in each cell all points have similar VP's, or formally *combinatorial structure*s of VP's, denoted by $\mathcal{VP}$, are equal. Combinatorial structure of $VP(q)$ is a circular list of visible edges and vertices of the scene in the order of their angle around q.

Let $P = \{p_1, \ldots, p_k\}$ be the set of the end-points of the segments in O, where $k \leq 2n$. With an observation similar to the one in the previous section, the arrangement of critical constraints (the rays emanating from p_i in the direction that points away from p_j for $1 \leq i, j \leq k$, $i \neq j$), determines the cells in which $\mathcal{VP}(q)$ is fixed and for each two adjacent cells these structures differ only in $O(1)$ edges and vertices.

Due to the changes in the scene, the critical constraints differ from what previously described. If a pair of points are not visible from each other, they do not produce any critical constraint at all. Some critical constraints may also be line segments, this happens when a critical constraint encounters an obstacle. Since for two points q and q' in the opposite sides of an obstacle $\mathcal{VP}(q) \neq \mathcal{VP}(q')$, the obstacles are also added to the set of critical constraints.

Since these differences do not increase the upper bound on the number of cells in the arrangement of critical constraints asymptotically, the argument about the complexity of the arrangement of critical constraints, which is called in this case *visibility decomposition*, remains valid, so we can rewrite the corollary as below.

Corollary 2. *In a polygonal scene of complexity $O(n)$, the plane can be decomposed into $O(n^4)$ cells, such that all points in a cell have equal $\mathcal{VP}$. Moreover $\mathcal{VP}$ of each two adjacent cells can be transformed to each other by $O(1)$ changes.*

Using the above result, the method in the previous section can be applied to compute $\mathcal{VP}(q)$ for any point q in $O(\log n)$ time, using $O(n^4)$ space and $O(n^4 \log n)$ time for preprocessing. If the actual $VP(q)$ (all the vertices and edges of VP) is to be computed, additional $O(|VP(q)|)$ time is needed. This can be done by examining the vertices and edges of $\mathcal{VP}(q)$ and finding non-obstacle edges and the corresponding vertices of $VP(q)$ (see Fig. 2).

Theorem 2. *A planar scene consists of n segment obstacles can be preprocessed in $O(n^4 \log n)$ time and $O(n^4)$ space, such that for any query point q, a pointer to a red-black tree which stores $\mathcal{VP}(q)$ can be returned in time $O(\log n)$. Furthermore the visibility polygon of the point, $VP(q)$, can be reported in time $O(\log n + |VP(q)|)$.*

It is easy to arrange a scene such that the number of combinatorially different visibility polygons is $\Theta(n^4)$ (e.g., see [17]). Therefore it seems likely that the above result is the best we can reach for optimal query time.

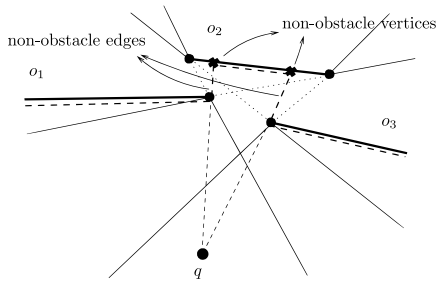


Fig. 2. Part of the visibility polygon of a point. The thick dashed lines denote the visibility polygon of q. Non-obstacle edges and non-obstacle vertices of the visibility polygon are marked in the figure.

3 Space–query-time tradeoff

In Section 2, an algorithm with logarithmic query time for computing VP is presented that uses $O(n^4)$ space and $O(n^4 \log n)$ preprocessing time. Asano *et al.* [4] have shown that we can compute VP of a point in a polygonal scene in $O(n)$ time by preprocessing the scene in $O(n^2)$ time using $O(n^2)$ space. In this section, these results are combined and a tradeoff between the memory usage and the query time for computing the visibility polygon is obtained.

3.1 Angular sorting

As in Section 2, we first consider the problem of angular sorting points around a query point. In this problem, P, $\theta_q(p_i)$ and θ_q are defined similar to the previous section. We assume the points are in the general position in the plane, so that no three points are collinear and for $1 \leq i \leq n$, $p_i q$ does not lie on a vertical line. In contrast to the previous section, here at most $o(n^4)$ space can be used.

For this problem the plane can be partitioned into disjoint cells, such that θ_q does not change *significantly* when q moves in each cell (similar to the method of Aronov *et al.* [2] to obtain space–query-time tradeoff). Unfortunately with this method, using $O(n^3)$ space, query time of $O(n \log n)$ is achieved which is not adequate.

Linear time algorithm for sorting points To be able to use the technique used by Asano *et al.* [4], let us review their method with some modifications. Consider the duality transform that maps the point $p = (p_x, p_y)$ into the line $p^* : y = p_x x - p_y$ and the line $l : y = mx + b$ into the point $l^* = (m, -b)$. This transformation preserves incidence and order between points and lines [8], i.e., $p \in l$ iff $l^* \in p^*$ and p lies above l iff l^* lies above p^*.

Let $\mathcal{A}(P^*)$ denote the arrangement of the set of lines $P^* = \{p_1^*, \ldots, p_n^*\}$. Line q^* can also be inserted in the arrangement in $O(n)$ time [6]. Let r_q denote the vertical upward ray from q and l_q its supporting line. When r_q rotates around q

counterclockwise 180°, it touches all the half-plane to the left of q, and l_q^* slides on line q^* from $-\infty$ to $+\infty$. Whenever r_q reaches a point p_i, l_q^* lies on p_i^*. Thus the order of all points p_i in the left half-plane of vertical line through q according to $\theta_q(p_i)$, is the same as the order of intersection points between q^* and the dual of those points according to x-coordinate. The same statement holds for the right half-plane of vertical line through q. Using this correspondence between orders, θ_q can be computed in linear time.

Lemma 2. [Asano *et al.*] *Using $O(n^2)$ preprocessing time and $O(n^2)$ space for constructing $\mathcal{A}(P^*)$ by an algorithm due to Chazelle et al. [6], for any query point q, we can find the angular sorted list of p_i, $1 \leq i \leq n$, in $O(n)$ time.*

Sublinear query time algorithm for sorting points Before describing the algorithm with space–query-time tradeoff, we should explain another concept. Let L be a set of lines in the plane. A $(1/r)$-cutting for L is a collection of (possibly unbounded) triangles with disjoint interiors, which cover all the plane and the interior of each triangle intersects at most n/r lines of L. Such triangles together with the collection of lines intersecting each triangle can be found for any $r \leq n$ in time $O(nr)$. The first (though not optimal) algorithm for constructing cutting is presented by Clarkson [7]. He showed that a random sample of size r of the lines of L can be used to produce a fairly good cutting. Efficient construction of $(1/r)$-cuttings of size $O(r^2)$ was given by Matoušek [11] and then improved by Agarwal [1].

As will be described later, a cutting that is effective for our purpose, has a specific property. To simplify the algorithm, in this section we describe the method using *random sampling* and give a simple (but inefficient) randomized solution. Later we show how an efficient cutting introduced by Chazelle [5] can be used with some modifications instead, to make the algorithm deterministic and also reduce the preprocessing and query time and space.

The basic idea in random sampling is to pick a random subset R of r lines from L. It can be proved [7] that with high probability, each triangle of *any* triangulation of the arrangement of R, $\mathcal{A}(R)$, intersects $O((n \log r)/r)$ lines of L. Let R be a random sample of size r of the set of lines $P^* = \{p_1^*, \ldots, p_n^*\}$. Let $\mathcal{A}(R)$ denote the arrangement of R and $\mathcal{R}$ denote a triangulation of $\mathcal{A}(R)$. According to the random sampling theory, with high probability, any triangle $t \in \mathcal{R}$ intersects $O((n \log r)/r)$ lines of P^*.

Lemma 3. *Any line l in the plane, intersects at most $O(r)$ triangles in $\mathcal{R}$.*

Proof. It is a trivial application of the Zone Theorem [9] and is omitted. □

The above lemma states that the line q^* intersects $O(r)$ triangles of $\mathcal{R}$ and for each such triangle t, the intersection segment $q^* \cap t$ is the dual of a double wedge w_t, with q as its center, in the primal plane. Therefore the primal plane is partitioned into $O(r)$ co-centered disjoint double wedges, totally covering all the plane. For computing θ_q, it is adequate to sort the points inside each double

wedge, and then join these sorted lists sequentially. Let each sorted sublist be denoted by $\theta_q(t)$. The sorted list $\theta_q(t)$ can be computed by first finding the sorted list of points whose dual intersect the triangle t, denoted by $P(t)$, and then confining the results to the double wedge w_t. In order to sort the points of $P(t)$ in each triangle $t \in \mathcal{R}$ efficiently, we can use the method of Section 2.

At the query stage, $\theta_q(t)$ is computed by performing four binary search in the returned red-black tree, to find the head and the tail of the two sublists that lie in the double wedge (one sublist for the left half-plane and one for the right half-plane). In this way, $O(\log |P(t)|)$ canonical subsets from the red-black tree are obtained whose union form the two sorted lists. This procedure is repeated for all triangles $t \in \mathcal{R}$ that are intersected by q^* and $O(r)$ sorted lists $\theta_q(t)$ are obtained which should be concatenated sequentially to make θ_q.

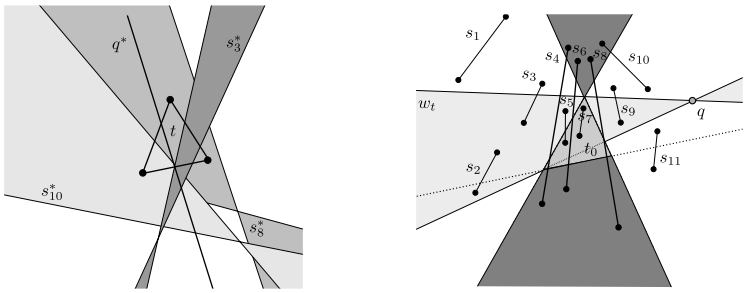
Constructing a $(1/r)$-cutting for P^* together with computing the lines intersected by each triangle uses $O(nr)$ time and space, for $1 \leq r \leq n$. Preprocessing a triangle t for angular sorting takes $O(|P(t)|^4)$ space and $O(|P(t)|^4 \log |P(t)|)$ time, which sum up to $O((n^4 \log^4 r)/r^2)$ space and $O((n^4 \log^4 r \log(n \log r/r))/r^2)$ time for all $O(r^2)$ triangles. At query time, each sorted subsequence is found in $O(\log((n \log r)/r))$ time, and the resulting canonical subsets are joined in $O(r \log((n \log r)/r))$ time.

Using efficient cutting Efficient cuttings that can be used for our approach have a special property: Any arbitrary line intersect $O(r)$ triangles of the cutting. We call this property the *regularity* of the cutting. Many algorithms have been proposed for constructing a $(1/r)$-cutting for a set of lines, but most of them do not fulfill regularity property. A good cutting which with some modifications satisfies the regularity was introduced by Chazelle [5]. Due to space limitations, we describe our modifications to make Chazelle's algorithm regular in the full version of the paper. We here only mention the final result of applying this efficient cutting.

Theorem 3. *A set of n points in the plane can be preprocessed into a data structure of size $O(m)$ for $n^2 \leq m \leq n^4$, in $O(m \log(\sqrt{m}/n))$ time, such that for any query point q, the angular sorted list of points, θ_q, can be returned in $O(n^2 \log(\sqrt{m}/n)/\sqrt{m})$ time.*

3.2 Space–query-time tradeoff for computing the visibility polygon

We can use the tradeoff between space and query-time for computing θ_q, to achieve a similar tradeoff for computing VP. The set O of the obstacle segments and the set P of the end-points of the segments are defined as before. Let $\mathcal{A}(P^*)$ denote the arrangement of dual lines of points of P. Based on this arrangement, we compute a regular $(1/r)$-cutting as described in Section 3.1 and decompose the plane into $O(r^2)$ triangles such that any triangle is intersected by at most $O(n/r)$ lines of P^*, and the dual line of any point q intersects at most $O(r)$ triangles of the cutting.



(a) Dual plane.

(b) Primal plane.

Fig. 3. (a) In the dual plane, the triangle t, dual of t_0, is intersected by q^*. The dual of some of obstacle segments are also shown. (b) The arrangement of t_0, q and obstacle segments in the primal plane.

As described in Section 3.1, the intersections of q^* with the triangles that are intersected by q^*, partition the primal plane into $O(r)$ co-centered disjoint double wedges, totally covering all the plane. Therefore it is enough to compute $\mathcal{VP}(q)$ bounded to each double wedge and join the partial $\mathcal{VP}$'s sequentially. Let the double wedge created by the triangle t be denoted by w_t and $\mathcal{VP}(q)$ bounded to w_t denoted by $\mathcal{VP}(q,t)$. $\mathcal{VP}(q,t)$ is computed very similar to the method described in Section 3.1 for computing $\theta_q(t)$. The only thing we should care is the possibility of the presence of some segments in $\mathcal{VP}(q,t)$ none of whose end-points lies in w_t.

In Fig. 3(a) the triangle t in the dual plane is shown which is intersected by q^*. The triangle t is also intersected by $O(n/r)$ lines which are dual to the end-points of $O(n/r)$ segments of O. We call these segments the *closed segments* of t, denoted by the set CS_t. At least the dual of one end-point of each segment in CS_t intersects t. There are also some segments in O which are not in CS_t, but may appear in $\mathcal{VP}(q)$, for example s_8 in Fig. 3. We call these segments the *open segments* of t, denoted by the set OS_t. OS_t consists of all the segments whose dual double wedge contains t completely.

The triangle t_0, whose dual is t, as in Fig. 3(b), partition the plane into three regions. A region, which is brighter in the figure, is the region that q and at least one end-point of each segment of CS_t lie (for example segments s_9 and s_{10}). In contrast, we have two regions, which is shaded in the figure, and contains at most one end-point of each segment of CS_t and both end-points of the other segments of O. If an end-point of a segment o_i lies in a shaded region and the other end-point in the other shaded region, o_i belongs to OS_t (for example segments s_4, s_6 and s_8).

By the above observations, and the fact that the segments of O may intersect each other only at their end-points, we conclude that the segments in OS_t partition the bright region, the region which contains q, into $|OS_t|+1$ subregions. Let

$R_t = \{r_t^1, \ldots, r_t^{|OS_t|+1}\}$ denote the set of subregions. In each region r_t^k, a set of segments $CS_t^k \subset CS_t$ is contained. Since $\mathcal{VP}(q,t)$ is limited to the subregion r_t^k in which q lies, for computing $\mathcal{VP}(q,t)$, we should first find r_t^k, and then compute $\mathcal{VP}(q)$ which is bounded to r_t^k and the double wedge w_t.

First assume we know where q lies, i.e., we know the region $r_t^k \in R_t$ that contains q, where $1 \leq k \leq |OS_t| + 1$. Any point inside r_t^k can see only the set of segments CS_t^k. If in the preprocessing phase, for each region r_t^k we use the method of Section 2.2 with CS_t^k as obstacles, we can find $\mathcal{VP}(q,t)$ bounded to w_t in time $O(\log(|CS_t^k|)) = O(\log(n/r))$. A trick is needed here to preprocess only non-empty regions of R_t, since otherwise an additional linear space is imposed, which may be undesirable when $r > O(n^{\frac{3}{4}})$. If we do not preprocess empty regions r_t^k, in which no closed segment lies, i.e., $CS_t^k = \emptyset$, the preprocessing time reduces to $O((n/r)^4 \log(n/r))$ and the space to $O((n/r)^4)$. Whenever a query point q is received, we first find the region r_t^k in which q lies. If $CS_t^k \neq \emptyset$, we can use the related data structures to compute $\mathcal{VP}(q,t)$, otherwise r_t^k consists of only two segments, $o_i, o_j \in OS_t$, and we can easily compute $\mathcal{VP}(q)$ bounded to w_t in $O(1)$ time.

There are three data structures, R_t, CS_t^k and CS_t that should be computed in the preprocessing phase. CS_t of complexity $O(n/r)$, which is already computed during the construction of the cutting $\mathcal{R}$, is the set of segments with an endpoint whose dual intersects t. However the process of computing R_t and CS_t^k is somewhat complicated and due to space limitations we give the details of the computation in the full version of the paper. Briefly, R_t of complexity $O(n)$ takes much space and time, so we cannot explicitly compute it for all the triangles of $\mathcal{R}$. We use a data structure to compute R_t of a triangle t based on $R_{t'}$ where t' is a neighbor triangle of t. By this data structure, we can also compute CS_t^k for each triangle t and $1 \leq k \leq |OS_t| + 1$. This procedure takes $O(rn \log(n/r))$ time and $O(rn)$ space.

At the query time r_t^k that contains q can be found in total $O(r \log(n/r))$ time for all the triangles that are intersected by q^*. For each triangle t, from r_t^k, we can easily compute $\mathcal{VP}(q)$ bounded by w_t in $O(\log(n/r))$ time. Summing up these amounts, we can compute $\mathcal{VP}(q)$ in $O(r \log(n/r))$ time.

In summary, the total preprocessing time and space are $O(n^4 \log(n/r)/r^2)$ and $O(n^4/r^2)$ respectively and the query time is $O(r \log(n/r))$. If the used space is denoted by m we can conclude the following theorem.

Theorem 4. *A planar polygonal scene of total complexity $O(n)$ can be preprocessed into a data structure of size $O(m)$ for $n^2 \leq m \leq n^4$, in $O(m \log(\sqrt{m}/n))$ time, such that for any query point q, in $O(n^2 \log(\sqrt{m}/n)/\sqrt{m})$ time $\mathcal{VP}(q)$ can be returned. Furthermore the visibility polygon of the point, $VP(q)$, can be reported in $O(n^2 \log(\sqrt{m}/n)/\sqrt{m} + |VP(q)|)$ time.*

4 Applications

In this section we use the previous results for computing the visibility polygon and apply them to some other related problems.

4.1 Maintaining the visibility polygon of a moving point

Let q be a point in a polygonal scene. The problem is to update $VP(q)$ as it moves along a line. We can use the technique previously used to compute VP of a query point, to maintain VP of a moving point in the plane. Due to space limitations, we only describe the final result.

Theorem 5. *A planar polygonal scene of total complexity n can be preprocessed into a data structure of size $O(m)$, $n^2 \leq m \leq n^4$, in time $O(m \log(\sqrt{m}/n))$, such that for any query point q which moves along a line, $VP(q)$ can be maintained in time $O((\frac{n^2}{\sqrt{m}} + k)(\log \frac{n^2}{\sqrt{m}} + \log \frac{\sqrt{m}}{n}))$ where k is the number of combinatorial visibility changes in $VP(q)$.*

It is remarkable that the moving path need not to be a straight line, but it can be broken at some points. In this case, the number of break points is added to the number of combinatorial changes in the above upper bounds.

4.2 The weak visibility polygon of a query segment

The weak visibility polygon of a segment rs is defined as the set of points in the plane, that are visible from at least a point on the segment. Using the previous result about maintaining the visibility polygon of a moving point, the weak visibility polygon of a query segment can be computed easily. We here only mention the last theorem and leave the proof for the extended version of the paper.

Theorem 6. *A planar polygonal scene of total complexity n can be preprocessed into a data structure of size $O(m)$, $n^2 \leq m \leq n^4$, in $O(m \log(\sqrt{m}/n))$ time, such that for any query segment rs, $VP(rs)$ can be computed in time $O((\frac{n^2}{\sqrt{m}} + |VP(rs)|)(\log \frac{n^2}{\sqrt{m}} + \log \frac{\sqrt{m}}{n}))$.*

4.3 Weak visibility detection between two query objects

In [12] Nouri *et al.* studied the problem of detecting weak visibility between two query objects in a polygonal scene. Formally, a scene consists of some obstacles of total complexity n, should be preprocessed, such that given any two query objects, we can quickly determine if the two objects are weakly visible. They proved that using $O(n^2)$ space and $O(n^{2+\epsilon})$ time for any $\epsilon > 0$, we can answer queries in $O(n^{1+\epsilon})$.

In their paper, they mentioned that the bottleneck of the algorithm is the time needed to compute the visibility polygon of a query point, and if it can be answered in a time less than $\Theta(n)$ in the worst case, the total query time will be reduced. Here we summarize the result of applying the new technique for computing $VP(q)$ and defer the details to the full version of the paper.

Theorem 7. *A planar polygonal scene of total complexity n can be preprocessed in $O(m^{1+\epsilon})$ time to build a data structure of size $O(m)$, where $n^2 \leq m \leq n^4$, so that the weak-visibility between two query line segments can be determined in $O(n^{2+\epsilon}/\sqrt{m})$ time.*

5 Conclusion

In this paper we studied the problem of computing the visibility polygon. We present a logarithmic query time algorithm, when we can use $O(n^4 \log n)$ time and $O(n^4)$ space in the preprocessing phase. As the algorithm requires much space, the algorithm was modified to get a tradeoff between space usage and query time. With this tradeoff, VP of a query point can be computed in time $O(n \log(\sqrt{m}/n)/\sqrt{m})$ using $O(m)$ space, where $n^2 \leq m \leq n^4$.

This problem may have many applications in other related problems. An interesting future work is to find more applications for the proposed algorithms. It is also an interesting open problem, whether our algorithms are optimal regarding space usage and preprocessing time.

References

1. P. K. Agarwal. Partitioning arrangements of lines, part I: An efficient deterministic algorithm. *Discrete Comput. Geom.*, 5(5):449–483, 1990.
2. B. Aronov, L. J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete Comput. Geom.*, 27(4):461–483, 2002.
3. T. Asano. Efficient algorithms for finding the visibility polygon for a polygonal region with holes. Manuscript, University of California at Berkeley.
4. T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1):49–63, 1986.
5. B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9:145–158, 1993.
6. B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.
7. K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2(2):195–222, 1987.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: algorithms and applications.* Springer-Verlag New York, Inc., 1997.
9. H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Comput.*, 22(2):418–429, 1993.
10. P. J. Heffernan and J. S. B. Mitchell. An optimal algorithm for computing visibility in the plane. *SIAM J. Comput.*, 24(1):184–201, 1995.
11. J. Matoušek. Construction of ϵ-nets. *Disc. Comput. Geom.*, 5:427–448, 1990.
12. Mostafa Nouri, Alireza Zarei, and Mohammad Ghodsi. Weak visibility of two objects in planar polygonal scenes. In *ICCSA (1)*, volume 4705 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007.
13. M. Pocchiola and G. Vegter. The visibility complex. *Int. J. Comput. Geometry Appl.*, 6(3):279–308, 1996.
14. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.
15. S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Symp. on Computational Geometry*, pages 14–23, 1986.
16. G. Vegter. The visibility diagram: A data structure for visibility problems and motion planning. In *Proc. of SWAT '90*, pages 97–110, 1990.
17. A. Zarei and M. Ghodsi. Efficient computation of query point visibility in polygons with holes. In *Proc. of the 21st symp. on Comp. geom.*, pages 314–320, 2005.