Finding the k-Visibility Region of a Point in a Simple Polygon in the Memory-Constrained Model

Yeganeh Bahoo* Bahareh Banyassady† Prosenjit Bose‡ Stephane Durocher*
Wolfgang Mulzer†

Abstract

We study the problem of k-visibility in the memory-constrained model. In this model, the input resides in a randomly accessible read-only memory of $O(n)$ words with $O(\log n)$ bits each. An algorithm can read and write $O(s)$ additional words of workspace during its execution, and it writes its output to write-only memory. In a given polygon P, for a given point $q \in P$, we say a point p is inside the k-visibility region of q iff the segment pq intersects the boundary of P at most k times. Given a simple n-vertex polygon P stored in a read-only array and a point $q \in P$, we give a time-space trade-off algorithm which reports a suitable representation of the k-visibility region of q in $O(n^2/s+n \log s)$ time using $O(s)$ words of workspace.

1 Introduction

Memory constraints on mobile and distributed devices have led to an increasing concern among researchers to design algorithms that use memory efficiently. One common model to capture this notion is the *memory-constrained model* [2]. In this model, the input resides in a randomly accessible read-only array of $O(n)$ words with $O(\log n)$ bits each. There is an additional read/write memory consisting of $O(s)$ words of $O(\log n)$ bits each, called the *workspace* of the algorithm. Here, $s \in \{1, \ldots, s\}$ is a parameter of the model. The output is written to a write-only array.

For a given polygon P and a given point $q \in P$, the point $p \in P$ is *k-visible* from q iff the segment pq properly intersects the boundary of P at most k times (p and q are not counted toward k). The set of k-visible points of P from q is called the *k-visibility region* of q within P, and is denoted $V_k(P, q)$; see Figure 1. Visibility has a rich history in computational geometry and other fields; see [6] for an overview. While the 0-visibility region is a connected component, the k-visibility region may be disconnected. The k-visibility region of a point inside the plane in presence of a polygon can be computed in $O(n^2)$ time [3].

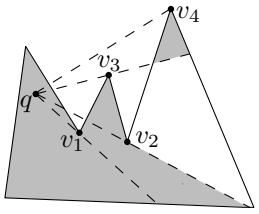

Figure 1: The gray region is $V_2(P, q)$. The vertices v_1, v_2, v_3 and v_4 are critical for q. ∂P is partitioned into chains v_2v_3, v_3v_1, v_1v_4 and v_4v_2.

Using constant workspace, the 0-visibility region of a point $q \in P$ can be computed in $O(n\bar{r})$ time, where $\bar{r}$ denotes the number of the reflex vertices of P in the output [4]. When the workspace is increased to $O(s)$, the running time decreases to $O(nr/2^s + n \log^2 r)$ or $O(nr/2^s + n \log r)$ randomized expected time, where $s \in O(\log r)$. Computing the 0-visibility region without workspace limitations takes $O(n)$ time [1].

We provide time-space trade-off algorithms for computing the k-visibility region of a simple polygon P from $q \in P$ using a small workspace.

2 Preliminaries and definitions

We have a simple polygon P in a read-only array as a list of n vertices in counterclockwise order along the boundary and a query point $q \in P$. The aim is to report a suitable representation of $V_k(P, q)$, using $O(s)$ words of workspace. We assume that the vertices of P are in *weak general position*, i.e., q does not lie on the line determined by any two vertices of P. W.l.o.g., assume that k is even and that $k < n$. If k is odd, we compute $V_{k-1}(P, q) = V_k(P, q)$, and if $k \geq n$, then P is completely k-visible. The boundary of $V_k(P, q)$ consists of part of the boundary of P and some chords that cross the interior of P to join two points on its boundary. We denote the boundary of planar set U by ∂U. Let $\theta \in [0, 2\pi)$, and let r_θ be the ray from q that forms an angle θ with the positive-horizontal axis. The j^{th} edge of P that intersects r_θ, starting from q, is denoted $e_\theta(j)$. Only the first $k + 1$ intersections of $r_\theta \cap \partial P$ are k-visible from q in direction θ.

*Department of Computer Science, University of Manitoba, {bahoo, durocher}@cs.umanitoba.ca

†Institut für Informatik, Freie Universität Berlin, {bahareh, mulzer}@inf.fu-berlin.de. Supported by DFG project MU/3501-2

‡School of Computer Science, Carleton University, jit@scs.carleton.ca

If r_θ does not stab any vertices of P, then the *edge lists*, i.e., the list of intersecting edges, of both $r_{\theta-\varepsilon}$ and $r_{\theta+\varepsilon}$, for a small enough $\varepsilon > 0$, are the same as the edge list of r_θ. However, if r_θ stabs a vertex v of P, then the edge lists of $r_{\theta-\varepsilon}$ and of $r_{\theta+\varepsilon}$ differ, for any small $\varepsilon > 0$. The difference is caused by the edges incident to v. If these edges lie on opposite sides of r_θ, then the edge list of $r_{\theta+\varepsilon}$ can be obtained from the edge list of $r_{\theta-\varepsilon}$ by exchanging the name of the corresponding edge. However, if both incident edges of v lie on the same side of r_θ, then there are two edges in the edge list of either $r_{\theta-\varepsilon}$ or $r_{\theta+\varepsilon}$ which are not in the edge list of the other. In this case, we call v a *critical* vertex; see Figure 1. The number of critical vertices in P is denoted by c. The *angle* of a vertex v refers to the angle between the ray qv and positive-horizontal axis. A *chain* is defined as a maximal sequence of edges of P which does not contain a critical vertex, except at the beginning and at the end. Thus, ∂P is partitioned into disjoint chains; see Figure 1.

Observation 1 *Let C be a chain on P. Suppose we are given an edge e of C, and a ray r_θ. We can find the edge $e_\theta \in C$ which intersects r_θ (if it exists) in $O(|C|)$ time using $O(1)$ workspace.*

When rotating the ray r_θ around q, the structure of the edge list of r_θ (i.e., the chains and their order) changes only when r_θ stabs a critical vertex. We will see that in this case a segment of r_θ may belong to $\partial V_k(P,q)$. A critical vertex v on r_θ is counted as both $e_\theta(j)$ and $e_\theta(j+1)$, if there are $j-1$ intersecting edges with r_θ between q and v. Obviously, v is k-visible if its position on r_θ is not after $e_\theta(k+1)$. A critical vertex v is called an *end vertex* if its edges lie on the right side of qv, and it is called a *start vertex* otherwise.

Lemma 1 *If r_θ stabs a k-visible critical vertex v, then the segment on r_θ between $e_\theta(k+2)$ and $e_\theta(k+3)$ (if they exist) is an edge of $V_k(P,q)$.*

Proof. If v is an end vertex, then for small enough $\varepsilon > 0$, the edges $e_\theta(k+2)$ and $e_\theta(k+3)$ are respectively $e_{\theta-\varepsilon}(k+2)$ and $e_{\theta-\varepsilon}(k+3)$, so they are not k-visible in direction $\theta-\varepsilon$. These edges are also $e_{\theta+\varepsilon}(k)$ and $e_{\theta+\varepsilon}(k+1)$, so they are k-visible in direction $\theta+\varepsilon$. Hence, the segment on r_θ between $e_\theta(k+2)$ and $e_\theta(k+3)$ belongs to $\partial V_k(P,q)$, and $V_k(P,q)$ lies on the side of the segment which has direction $\theta+\varepsilon$; see Figure 2. Similarly, if v is a start vertex, the same segment belongs to $\partial V_k(P,q)$; in this case, $V_k(P,q)$ lies on the side of the segment with direction $\theta-\varepsilon$. □

Lemma 1 leads to the following definition: for a ray r_θ that stabs a k-visible critical vertex v, the segment between $e_\theta(k+2)$ and $e_\theta(k+3)$ (if they exist) is called the *window* of r_θ. The window is *CCW* if $V_k(P,q)$ lies to the left of r_θ;(see Figures 2), and *CW*, otherwise.

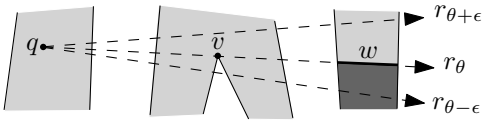

Figure 2: For the ray r_θ which stabs the end vertex v, the segment w is a *CCW* window of $V_4(P,q)$.

Each window is identified by its two endpoints, and each endpoint is represented by a triple (θ, j, type), where j is the index of either $e_\theta(k+2)$ or $e_\theta(k+3)$ in P (depending on the position of two endpoints of a window on these edges) and type $\in \{\text{CCW}, \text{CW}\}$ specifies the *type* of the window. The set of endpoints of windows of $V_k(P,q)$ is denoted by $W_k(P,q)$.

Observation 2 *$\partial V_k(P,q)$ has $O(n)$ vertices.*

Lemma 2 *If there exists an algorithm $A(P,q,k)$ in the memory-constrained model for computing $W = W_k(P,q)$ in $T_A(n)$ time using $S_A(n)$ workspace, where n is the number of vertices of P, then there exists an algorithm $A'(P,q,W)$ in the memory-constrained model that reports $\partial V_k(P,q)$ in $O(|W|T_A(n)+n)$ time using $O(S_A(n))$ workspace.*

Proof. The algorithm A' works as follows: start from a point $w_0 \in W$ and walk on ∂P in CCW direction until the next element $w_1 \in W$. If this walk is on the k-visible side of w_0 (which is specified by the type of w_0), report the visited edges of P; otherwise, report only the windows with endpoint(s) w_0 and/or w_1. Repeat this procedure until the entire boundary ∂P has been traversed. Specifically, in step i of A', run algorithm A and find $w_i = (\theta_i, j_i, \text{type}_i)$ which minimizes j_i, with $j_i > j_{i-1}$ for $i \neq 0$. If there is more than one element which minimizes j_i, choose the one among them that minimizes $|\theta_i - \theta_{i-1}|$ (minimizes θ_i for $i = 0$). Since the output of A is write-only, in each step i of A' we have to run A again to find w_i, requiring $O(|W|T_A(n))$ total time. Regarding the workspace, in step i of A' we store only w_{i-1} and w_i; however, for finding w_i we need as much workspace as A does. Thus, the workspace of A' is $O(S_A(n))$. □

Lemma 2 shows that given $W_k(P,q)$ and P, we can uniquely report $\partial V_k(P,q)$. This motivates us to focus on algorithms for computing $W_k(P,q)$. We assume that P has at least one critical vertex, if not, then $\partial V_k(P,q) = \partial P$. From now on, $e_i(j)$ denotes the j^{th} intersecting edge of the ray qv_i, where v_i is a critical vertex of P. However, instead of $e_i(j)$, it suffices to find an arbitrary edge of the chain containing $e_i(j)$ and then apply Observation 1 to find $e_i(j)$. Therefore, we refer to any edge of the chain containing $e_i(j)$ by $e_i(j)$. The following algorithms, for any critical vertex v_i, examine its position relative to $e_i(k+1)$ on qv_i and,

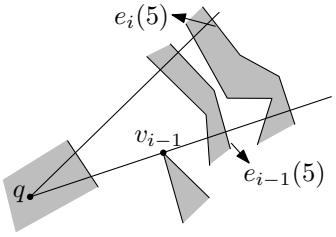

Figure 3: v_{i-1} is an end vertex. $e_i(5)$ is the second intersecting chain to the right of $e_{i-1}(5)$.

if it is k-visible, reports the segment on qv_i which is between $e_i(k+2)$ and $e_i(k+3)$ (if they exist).

3 A constant-memory algorithm

In this section, we assume that only $O(1)$ workspace is available. Suppose that v_0 is the critical vertex with smallest angle. The algorithm starts from qv_0 and finds $e_0(k+1)$ in $O(kn)$ time using $O(1)$ workspace. Basically, the algorithm passes over the input $k+1$ times, and in each pass, it finds the next intersecting edge of qv_0 until the $(k+1)^{\text{th}}$ one, $e_0(k+1)$. If v_0 does not lie after $e_0(k+1)$ on qv_0, in other words, if v_0 is k-visible, it reports the window of qv_0. Finding the window can be done in two passes by determining the first and the second intersecting edge after $e_0(k+1)$ on qv_0. Then, the algorithm finds the next critical vertex with smallest angle after v_0; we call it v_1. The algorithm determines $e_1(k+1)$, and if v_1 is k-visible, it reports the window of qv_1 (if it exists). However, for $1 \leq i$, we find $e_i(k+1)$ in $O(n)$ time by using $e_{i-1}(k+1)$. More precisely, if v_{i-1} is an end vertex, then the incident edges to v_{i-1} do not intersect qv_i; see Figure 3. If v_i is a start vertex, then the incident edges to v_i do not intersect qv_{i-1}. Except for these edges, all the other intersecting edges of qv_{i-1} intersect qv_i in the same order, and vice versa. Hence, if $e_i(k+1)$ intersects qv_{i-1}, then there is at most one other edge between $e_{i-1}(k+1)$ and $e_i(k+1)$ that intersects qv_{i-1}. Thus, $e_i(k+1)$ can be found in at most two passes over the input. More accurately, we have found only an edge of the chain of $e_i(k+1)$; applying Observation 1, the edge $e_i(k+1)$ can be obtained. The algorithm repeats the above procedure until all critical vertices have been processed. Since the number of critical vertices is c, and since processing each critical vertex takes $O(n)$ time, except for v_0, which takes $O(kn)$ time, the running time of the algorithm is $O(kn+cn)$, using $O(1)$ workspace. This leads to the following theorem:

Theorem 3 *Given a simple polygon P with n vertices in a read-only array, a point $q \in P$, and a constant $k \in \mathbb{N}$, there is an algorithm which reports $W_k(P,q)$ in $O(kn+cn)$ time using $O(1)$ workspace.*

4 Memory-constrained algorithms

In this section, we assume $O(s)$ workspace is available, and we show how to exploit this for a faster algorithm. The following lemma is implicitly mentioned in [5] (the second paragraph in the proof of Theorem 2.1)

Lemma 4 *Given a read-only array A of size n, $O(s)$ additional workspace and a specific element $x \in A$, there is an algorithm that finds the s smallest elements in A that are larger than x in $O(n)$ time.*

Proof. In the first step, insert the first $2s$ elements of A that are larger than x into workspace memory (without sorting them). Select the median of the $2s$ elements in memory in $O(s)$ time, and remove the elements which are larger than the median. In the next step, insert the next batch of s elements of A that are larger than x into memory and again find their median. Remove the elements larger than the median. Repeat the latter step until all elements of A are processed. Clearly, at the end of each step, the s smallest elements among those processed so far are in memory. Since the number of batches is $O(n/s)$, the running time is $O(n)$ using $O(s)$ workspace. □

Lemma 5 *Given a read-only array A of size n and $O(s)$ additional workspace, there is an algorithm that finds the k^{th} smallest element in A in $O(\lceil k/s \rceil n)$ time.*

Proof. In the first step, apply Lemma 4 to find the first batch of s smallest elements in A and to insert them into memory in $O(n)$ time. If $k < s$, select the k^{th} smallest element in memory in $O(s)$ time; otherwise, find the largest element in memory. In step i, apply Lemma 4 to find the i^{th} batch of s smallest elements of A and insert them into memory. If $k < i \cdot s$, select the $(k-(i-1)s)^{th}$ smallest element in memory in $O(s)$ time; otherwise, find the largest element in memory and repeat. The element being sought is in the $\lceil k/s \rceil^{th}$ batch of s smallest elements; therefore,the running time is $O(\lceil k/s \rceil n)$ using $O(s)$ workspace. □

There is an $O(n \log\log_s n)$ expected time randomized algorithm for the selection problem using $O(s)$ workspace in the read-only model [7]. Depending on k, s, and n, we choose the latter algorithm or the one in Lemma 5. In each iteration of the following algorithm, we find the next batch of s critical vertices with smallest angles and sort them in memory. Then, we construct a data structure T that contains the possible candidates for the $(k+1)^{\text{th}}$ intersecting edges of the rays from q to the critical vertices of the batch. In each step, when we process a critical vertex of the batch, we use T to find the window of the critical vertex, and we update T. For updating T efficiently, we use another data structure T_θ; see below. We repeat this procedure for the next batch of s critical vertices.

As in the constant-memory algorithm, we find the critical vertex v_0 with smallest angle. We apply Lemma 4 to find the batch of s critical vertices with smallest angles after v_0, and we sort them in memory. For qv_0, we apply Lemma 5 to find $e_0(k+1)$, and if v_0 is k-visible, we report the window (if it exists). Then, we apply Lemma 4 to find the two batches of $2s$ adjacent intersecting edges to the right and to the left of $e_0(k+1)$ on qv_0, we insert them in a balanced search tree T. Hence, T stores all $e_0(j)$, for $k+1-2s \leq j \leq k+1+2s$, in sorted order according to their intersection with qv_0. These edges are candidates for the $(k+1)^{\text{th}}$ intersecting edge of the next s rays in angular order or $e_i(k+1)$, for $1 \leq i \leq s$. This is because, as we explained before, if $e_i(k+1)$ intersects qv_{i-1}, then there is at most one other edge between $e_{i-1}(k+1)$ and $e_i(k+1)$ that intersects qv_{i-1}. Therefore, $e_i(k+1)$ is either an intersecting edge of qv_0, and in this case there are at most $2i-1$ edges between $e_0(k+1)$ and $e_i(k+1)$, or $e_i(k+1)$ is an edge which is inserted in T later. Then for each edge in T we determine the larger angle of its endpoints. This angle shows the position of the endpoint between the rays from q to the critical vertices. Specifically, if the edge is incident to a non-critical vertex, this angle determines the step in which the name of the edge in T should be updated to the other incident edge to the vertex. By traversing ∂P we determine these angles for the edges in T, and we insert them in a balanced search tree T_θ, whose entries are connected through cross-pointers to their corresponding edges in T. We construct T_θ in $O(n + s\log s)$ time.

After creating T and T_θ, we start from the next critical vertex with smallest angle after v_0, called v_1, and we update T so that it contains the edge list of qv_1: If there is any angle in T_θ which is smaller than the angle of v_1, we change the corresponding edge of the angle in T with its previous or next edge in P. In other words, we have found a non-critical vertex between qv_0 and qv_1 and so we change its incident edge, which has been already in T, with its other incident edge. Then we find the angle of the new edge and insert it into T_θ. These two steps take $O(1)$ and $O(\log s)$ time for each angle that meets the condition. By doing these steps, changes in the edge list which are caused by non-critical vertices between qv_0 and qv_1 are handled. Then we update T and consequently T_θ according to the type of v_1: if v_1 is an end (start) critical vertex, we remove (insert) the two edges which are incident to v_1. In both cases, we update T only if the incident edges to v_1 are in the interval of the $2s$ intersecting edges of qv_0 in T, this takes $O(\log s)$ time. Now T contains $2s$ intersecting edges of qv_1, and we can find $e_1(k+1)$ using the position of $e_0(k+1)$ and its neighbours in T in $O(1)$ time. We repeat this procedure for $1 \leq i \leq s$, and we determine $e_i(k+1)$ and the window of qv_i by using T and $e_{i-1}(k+1)$.

After processing the first batch, we apply Lemma 4 to find the next batch of s critical vertices with smallest angle, and we sort them in memory. The last updated T is not usable anymore, because it does not necessarily contain any right or left neighbours of $e_s(k+1)$. Applying Lemma 4 as before, we find the two batches of $2s$ adjacent intersecting edges to the right and to the left of $e_s(k+1)$ on qv_s and we insert them into T. We also update T_θ. Then similarly for each $s < i \leq 2s$, we find $e_i(k+1)$ and its corresponding window, and we update T and T_θ. In summary, updating T considering the changes that are caused by critical and non-critical vertices of the batch takes respectively $O(s\log s)$ and $O(n'\log s)$ time, where n' is the number of non-critical vertices that lie on the interval of the batch. In the next iteration, we repeat the same procedure for the next batch of critical vertices. We stop when all critical vertices are processed. Since the batches do not have any intersections, each non-critical vertex lies only on one batch. Thus, updating T in all batches takes $O(n\log s)$ time. All together, finding the batches of s critical vertices, constructing and updating the data structures and reporting the windows take $O(cn/s+n\log s)$ time for all the critical vertices, in addition to the running time of k-selection in the first batch.

Theorem 6 *Given a simple polygon P with n vertices in a read-only array, a point $q \in P$, and a constant $k \in \mathbb{N}$, there is an algorithm which reports $W_k(P,q)$ in $O(cn/s+n\log s+\min\{kn/s, n\log\log_s n\})$ time using $O(s)$ workspace.*

References

[1] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1-4):49–63, 1986.

[2] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *CGTA*, 46(8): 959–969, 2013.

[3] A. L. Bajuelos, S. Canales, G. Hernández-Peñalver, and A. M. Martins. A hybrid metaheuristic strategy for covering with wireless devices. *J. UCS*, 18(14):1906–1932, 2012.

[4] L. Barba, M. Korman, S. Langerman, and R. I. Silveira. Computing a visibility polygon using few variables. *JoCG*, 47(9):918–926, 2014.

[5] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *DCG*, 37(1):79–102, 2007.

[6] S. K. Ghosh. Visibility algorithms in the plane. *Cambridge University Press*, 2007.

[7] J. I. Munro and V. Raman. Selection from read-only memory and sorting with optimum data movement. *Theoretical Computer Science*, 165(2):311-323, 1996.