Time-Space Trade-off for Finding the k-Visibility Region of a Point in a Polygon*

Yeganeh Bahoo[1], Bahareh Banyassady[2] Prosenjit Bose[3], Stephane Durocher[1], and Wolfgang Mulzer[2]

[1] Department of Computer Science, University of Manitoba, Canada
[bahoo, durocher]@cs.umanitoba.ca
[2] Institut für Informatik, Freie Universität Berlin, Germany
[bahareh, mulzer]@inf.fu-berlin.de
[3] School of Computer Science, Carleton University, Canada
jit@scs.carleton.ca

Abstract. We study the problem of computing the k-visibility region 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 and for a given point $q \in P$, we say that a point p is inside the k-visibility region of q, if and only if the line segment pq intersects the boundary of P at most k times. Given a simple n-vertex polygon P stored in a read-only input array and a point $q \in P$, we give a time-space trade-off algorithm which reports the k-visibility region of q in P in $O(cn/s + n \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ words of workspace. Here $c \leq n$ is the number of critical vertices for q, i.e., the vertices of P where the visibility region may change. We also show how to generalize this result for polygons with holes and for sets of non-crossing line segments.

Keywords: memory-constrained model, k-visibility region, time-space trade-off

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 is provided 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, which is called the *workspace* of the algorithm. Here, $s \in \{1, \ldots, n\}$ is a parameter of the model. The output is written to a write-only array.

* This work was partially supported by DFG project MU/3501-2 and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Suppose we are given a polygon P and a query point $q \in P$. We say that the point $p \in P$ is *k-visible* from q if and only if the line 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 it is denoted by $V_k(P, q)$; see Figure 1. Visibility problems have played and continue to play a major role in computational geometry since the dawn of the field, leading to a rich history; see [16] for an overview. The concept of visibility through a single edge first appeared in [11]. Recently, k-visibility for $k > 1$ has been introduced and applied to model coverage of wireless devices whose radio signals can penetrate a given number k of walls [1, 13]. There are some other results in this context; for example see [4, 10, 12, 14, 15, 17, 20]. While the 0-visibility region consists of one connected component, the k-visibility region may be disconnected in general. A previous work [3] presents an algorithm for a slightly different variant of this problem, which computes the set of points in the plane which are k-visible from q in presence of a polygon P in $O(n^2)$ time using $O(n)$ space. In this case the k-visibility region is a single connected component.

The optimal classic algorithm for computing the 0-visibility region runs in $O(n)$ time using $O(n)$ space [18]. In the memory-constrained model using $O(1)$ workspace, there is an algorithm which reports the 0-visibility region of a point $q \in P$ in $O(n\bar{r})$ time, where $\bar{r}$ denotes number of reflex vertices of P in output [6]. This algorithm scans the boundary of P in counterclockwise order and it reports the maximal chains of adjacent vertices of P which are 0-visible from q. More precisely, it starts from a visible vertex v_{start}, and it finds v_{vis}, the next visible reflex vertex with respect to q, in $O(n)$ time. The first intersection of the ray qv_{vis} with the boundary of P is called the shadow of v_{vis}. Depending on the type of v_{vis}, the end vertex of the maximal visible chain starting at v_{start}, is either v_{vis} or its shadow, and in each case the other one is the start vertex of the next maximal visible chain. Thus, in each iteration the algorithm reports a maximal visible chain and it repeats this procedure $\bar{r}$ times, where $\bar{r}$ is the number of visible reflex vertices of P. This takes $O(n\bar{r})$ time using $O(1)$ workspace. When the workspace is increased to $O(s)$, such that $s \in O(\log r)$ and r is the number of reflex vertices of P with respect to q, they present $O(nr/2^s + n\log^2 r)$ time or $O(nr/2^s + n\log r)$ randomized expected time algorithm. The method is based on a divide-and-conquer approach which uses the previous algorithm as base algorithm and in each step of the recursion, it splits a chain into two subchains with roughly half of the visible reflex vertices of the chain. Due to differences between the properties of the 0-visibility region and the k-visibility region, there seems to be no straightforward way to generalize this approach. In [5] a general method for transforming *stack-based* algorithms into the memory-constrained model is provided, which can be used as an alternative method to obtain a time-space trade-off to compute the 0-visibility region.

Here, we look at the more general problem of computing the k-visibility region of a simple polygon P from $q \in P$ using a small workspace, and we establish a trade-off between running time and workspace. Unless stated otherwise, all polygons will be understood to be simple.

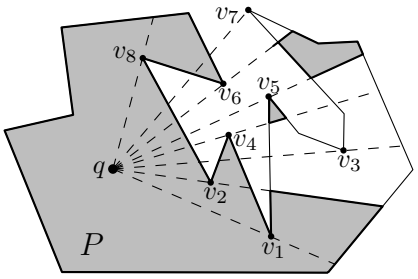

Fig. 1. The gray region is $V_2(P, q)$. The vertices $v_1, \ldots, v_8$ are critical for q. Here v_1, v_2, v_3 and v_6 are start vertices, while v_4, v_5, v_7 and v_8 are end vertices. The boundary of P is partitioned into 8 disjoint chains, i.e, the counterclockwise chain v_3v_5.

2 Preliminaries and definitions

We assume that our simple polygon P is given in a read-only array as a list of n vertices in counterclockwise (CCW) order along the boundary. This input array also contains a query point $q \in P$. The aim is to report $V_k(P, q)$, using $O(s)$ words of workspace. We assume that the vertices of P are in *weak general position*, i.e., the query point q does not lie on the line determined by any two vertices of P. Without loss of generality, assume that k is even and that $k < n$. If k is odd, we compute $V_{k-1}(P, q)$, which is by definition equal to $V_k(P, q)$, 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 a planar set U by ∂U.

Let $\theta \in [0, 2\pi)$, and let r_θ be the ray from q that forms a CCW angle θ with the positive-horizontal axis. An edge of P that intersects r_θ is called an *intersecting edge* of r_θ. The *edge list* of r_θ is defined as the sorted list of intersecting edges of r_θ, according to their intersection with r_θ (from q). The j^{th} member of the edge list of r_θ is denoted $e_\theta(j)$. When rotating r_θ around q in CCW order, the edge list of r_θ does not change unless r_θ stabs a vertex v of P. If r_θ stabs v, then the edge lists of $r_{\theta-\varepsilon}$ and of $r_{\theta+\varepsilon}$ differ, for any small $\varepsilon > 0$. The difference is caused only by edges incident to v. If these edges lie on opposite sides of r_θ, then the edge list of $r_{\theta+\varepsilon}$ is obtained from the edge list of $r_{\theta-\varepsilon}$ by exchanging the incident edge of v, which is in the edge list of $r_{\theta-\varepsilon}$, with the other incident edge of v. If both incident edges of v lie on the same side of r_θ, we call v a *critical* vertex; see Figure 1. If both incident edges of v lie on the right/left side of r_θ, then the edge list of $r_{\theta+\varepsilon}$ is obtained by removing/adding the two incident edges of v from/to the edge list of $r_{\theta-\varepsilon}$. For simplicity, if r_θ stabs a vertex v, we define the edge list of r_θ equal to the edge list of $r_{\theta+\varepsilon}$, for a small $\varepsilon > 0$. The number of critical vertices in P is denoted by c. 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. The critical vertex v is called an *end vertex*/a *start vertex* if both incident edges to v lie on the right/left side of r_θ. The name is due to the

fact that an end/start vertex shows the end/start of two chains in the edge list; see Figure 1. The *angle* of a vertex v which lies on the ray r_θ refers to θ.

Observation 2.1. *Suppose we are given an edge e of a chain C of P, and a ray r_θ. We can find the edge of C which intersects r_θ (if it exists) by scanning the chain C of P in $O(|C|)$ time using $O(1)$ words of workspace.*

The above observation implies that, any edge of a chain may be used as a proper representative of the chain and its other edges. Thus, in the edge list, each edge refers to its containing chain. Obviously, in direction θ, only the first $k+1$ members of the edge list of r_θ are k-visible from q, which leads us to focus on chains and their order in the edge list. As we explained before, when rotating 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 v. We will see that in this case a segment on r_θ may belong to $\partial V_k(P,q)$. Obviously, v is k-visible if its position on r_θ is not after $e_\theta(k+1)$.

Lemma 2.2. *If r_θ stabs a k-visible end (or start) vertex v, then the segment on r_θ between $e_\theta(k)$ and $e_\theta(k+1)$ (or $e_\theta(k+2)$ and $e_\theta(k+3)$), if these two edges exist, is an edge of $V_k(P,q)$.*

Proof. If v is an end vertex, then for small enough $\varepsilon > 0$, the edges $e_\theta(k)$ and $e_\theta(k+1)$ 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 $e_\theta(k)$ and $e_\theta(k+1)$ belongs to $\partial V_k(P,q)$. Similarly, if v is a start vertex, the segment between $e_\theta(k+2)$ and $e_\theta(k+3)$ belongs to $\partial V_k(P,q)$; see Figure 2. □

Lemma 2.2 leads to the following definition: for a ray r_θ that stabs a k-visible end (or start) vertex v, the segment between $e_\theta(k)$ and $e_\theta(k+1)$ (or $e_\theta(k+2)$ and $e_\theta(k+3)$), if they exist, is called the *window* of r_θ; see Figure 2.

Observation 2.3. *The boundary of $V_k(P,q)$ has $O(n)$ vertices.*

Proof. $\partial V_k(P,q)$ consists of part of ∂P and windows; thus, a vertex of $V_k(P,q)$ is either a vertex of P or an endpoint of a window. Since each critical vertex causes at most one window, the number of vertices of $V_k(P,q)$ is $O(n)$. □

Obviously, if P has no critical vertex, then no window exists, and $\partial V_k(P,q) = \partial P$. Thus, we assume that P has at least one critical vertex. From now on, $e_i(j)$ denotes the j^{th} intersecting edge of the ray qv_i, where v_i is a 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 2.1 to find $e_i(j)$. Therefore, we refer to any edge of the chain containing $e_i(j)$ by $e_i(j)$. In the following algorithms, for any critical vertex v_i, we determine $e_i(k+1)$ which helps to find the window of qv_i (if it exists), and also the part of ∂P which is in $\partial V_k(P,q)$. However, depending on how much workspace is available, we have different approaches for finding all $e_i(k+1)$. Details follow in the next sections.

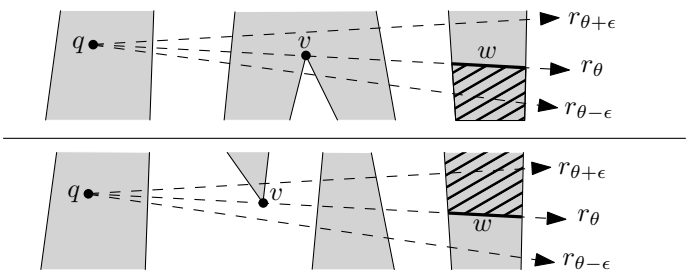

Fig. 2. The ray r_θ in the top/bottom figure stabs the end/start vertex v. The segment w is a window of 4-visible region. The tiled regions are not 4-visible for q.

3 A constant-memory algorithm

In this section, we assume that only $O(1)$ words of workspace is available. Suppose v_0 and v_1 are the critical vertices with respectively first and second smallest (polar) angles. We start from qv_0 and we find $e_0(k+1)$ in $O(kn)$ time using $O(1)$ words of workspace. Basically, we perform a simple *selection* subroutine as follows: pass over the input $k+1$ times, and in each pass, find 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, i.e., if v_0 is k-visible, we report the window of qv_0 (if it exists). Since the window is defined by $e_0(k)$ and $e_0(k+1)$ or by $e_0(k+2)$ and $e_0(k+3)$, it can be found in at most two passes over the input. Then we report the part of $\partial V_k(P,q)$ lying between qv_0 and qv_1 while scanning ∂P. In fact, for each edge $e \in P$ which is in the edge list of qv_0 and lies before $e_0(k+1)$ on qv_0, we report the segment of e which is between qv_0 and qv_1. We repeat the above procedure for v_1 except for determining $e_1(k+1)$ which is done in $O(n)$ time using $e_0(k+1)$ as follows: for $1 \leq i$, if v_i is an end or a start vertex the incident edges to v_i are respectively in the edge list of qv_{i-1} or qv_i and not in the other one; see Figure 3. Except for edges incident to v_i, all the other intersecting edges of qv_{i-1} intersect qv_i in the same order, and vice versa. Hence, if $e_{i-1}(k+1)$ lies before v_i on qv_i, then it defines $e_i(k+1)$. Otherwise, if v_i is an end/ a start vertex, then the second right/ left neighbour of $e_{i-1}(k+1)$ in the edge list of qv_{i-1}/ qv_i defines $e_i(k+1)$. However, in all cases the chain of $e_i(k+1)$ is found by at most two passes over the input; applying Observation 2.1, the edge $e_i(k+1)$ is obtained. Notice that here and in the following algorithms, if there are less than $k+1$ intersecting edges on qv_{i-1}, we store the last intersecting edge of qv_{i-1}, and the number of intersecting edges of qv_{i-1}. We this edge instead of $e_{i-1}(k+1)$, in the same procedure as above, to find $e_i(k+1)$ or the last intersecting edge of qv_i and the number of intersecting edges of qv_i. The algorithm repeats the above procedure until all critical vertices have been processed. The number of critical vertices is c, and processing each of them takes $O(n)$ time, except for the selection subroutine during processing v_0, which takes $O(kn)$ time. Thus, the running time of the algorithm is $O(kn+cn)$, using $O(1)$ words of workspace. This leads to the following theorem:

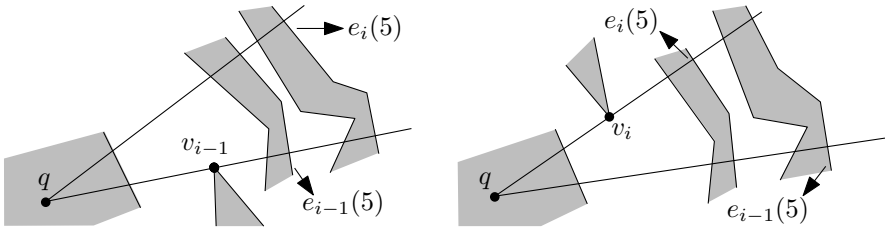

Fig. 3. Left: v_i is an end vertex and $e_i(5)$ is the second intersecting chain to the right of $e_{i-1}(5)$. Right: v_i is a start vertex and $e_i(5)$ is the second intersecting chain to the left of $e_{i-1}(5)$.

Theorem 3.1. *Given a simple polygon P with n vertices in a read-only array, a point $q \in P$, and a parameter $k \in \mathbb{N}$, there is an algorithm which reports $\partial V_k(P, q)$ in $O(kn + cn)$ time using $O(1)$ words of workspace, where c is the number of critical vertices of P.*

4 Memory-constrained algorithms

In this section, we assume that word of $O(s)$ workspace is available, and we show how to exploit the additional workspace to compute the k-visibility region faster. We provide two algorithms, the first one is presented only for better understanding of the second algorithm, which provides a better running time. In the first algorithm we process all the vertices in contiguous batches of size s in angular order. In each iteration we find the next batch of s vertices, and using the edge list of the last processed vertex, we construct a data structure which is used to find the windows of the batch. Using the windows we report $\partial V_k(P, q)$ in the interval of the batch. In the second algorithm we improve the running time by skipping the non-critical vertices. Specifically, in each iteration we find the next batch of s adjacent critical vertices, and similarly as the first algorithm, we construct a data structure for finding the windows, which requires a more involved approach to be updated. We first state Lemma 4.1, which is implicitly mentioned in [8] (see the second paragraph in the proof of Theorem 2.1)

Lemma 4.1. *Given a read-only array A of size n and an element $x \in A$, there is an algorithm that finds the s smallest elements in A, among those elements which are larger than x, in $O(n)$ time using $O(s)$ additional words of workspace.*

Proof. In the first step, insert the first $2s$ elements of A which 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 which are larger than x into memory and again find the median of the $2s$ elements in memory and remove the elements which are larger than the median. Repeat the latter step until all the elements of A are processed. Clearly, at the end of each

step, the s smallest elements of the ones which have been already processed, are in memory. Since the number of batches or steps is $O(n/s)$, the running time of the algorithm is $O(n)$ and it uses only $O(s)$ word of workspace. □

Lemma 4.2. *Given a read-only array A of size n and a parameter $k \in \mathbb{N}$, there is an algorithm that finds the k^{th} smallest element in A in $O(\lceil k/s \rceil n)$ time using $O(s)$ additional word of workspace.*

Proof. In the first step, apply Lemma 4.1 to find the first batch of s smallest elements in A and insert them into workspace 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, which plays the role of x in Lemma 4.1. In step i, apply Lemma 4.1 to find the i^{th} batch of s smallest elements in 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 and stop; 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 of A; therefore, we can find it in $O(\lceil k/s \rceil n)$ time using $O(s)$ word of workspace. □

In addition to our algorithm in Lemma 4.2 there are several other results on the selection problem in the read-only model; see Table 1 of [9]. There are $O(n \log\log_s n)$ expected time randomized algorithms for selection problem using $O(s)$ words of workspace in the read-only model [7, 19]. Depending on k, s and n we choose one of the latter algorithms or the algorithm of Lemma 4.2. In conclusion, the running time of selection in the read-only model using $O(s)$ words of workspace, which is denoted by $T_{\text{selection}}$, is $O(\min\{\lceil k/s \rceil n, n \log\log_s n\})$ expected time. Next we describe how to apply Lemmas 4.1 and 4.2.

4.1 Algo 1: processing all the vertices

First we find the critical vertex v_0 with smallest angle. We apply Lemma 4.1 to find the batch of s vertices with smallest angles after v_0, and we sort them in workspace memory in $O(s \log s)$ time. For qv_0 we use the selection subroutine (with $O(s)$ word of workspace) to find $e_0(k+1)$, and if v_0 is a k-visible vertex we report its window (if it exists).

Then, we apply Lemma 4.1 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. In other words, in T we store all $e_0(j)$, for $k+1-2s \leq j \leq k+1+2s$, sorted according to their intersection with qv_0. These edges are candidates for the $(k+1)^{th}$ 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 in Section 3, if $e_i(k+1)$ belongs to the edge list of qv_{i-1}, there is at most one edge between $e_{i-1}(k+1)$ and $e_i(k+1)$ in the edge list. Therefore, $e_i(k+1)$ is either an in the edge list 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; see Figure 4. More specifically, after creating T, we start from v_1, the next vertex with smallest angle after v_0, and according to the type of v_1, we update T: if v_1 is a non-critical vertex we change the incident edge to v_1 which is already in T with the other incident edge

to v_1; if v_1 is an end (start) critical vertex, we remove (insert) the two edges which are incident to v_1. In all 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. By updating T we can find $e_1(k+1)$ and the window of qv_1 (if it exists) using the position of $e_0(k+1)$ or its neighbours in T in $O(1)$ time.

In the same procedure for $1 \leq i \leq s$, using T and $e_{i-1}(k+1)$, we determine $e_i(k+1)$ and the window of qv_i, which take $O(s \log s)$ total time. Whenever we find and report a window, we insert its endpoints into a balanced search tree W in $O(\log s)$ time. In W the endpoints of windows are sorted according to the indices of the edges of P on which they lie. For reporting the part of $\partial V_k(P, q)$ between qv_0 and qv_s, we use W (as a sorted array) and also E which is the set of edges $e_i(k+1)$, $1 \leq i \leq s$. We know that, if there is no endpoint of a window on a segment, then the visibility of the segment is consistent on the entire segment. Using this, we walk on ∂P and for each edge e of P, we check if its endpoints, restricted to the interval of the batch, are k-visible or not (in $O(1)$ time using E). We also check if there is any endpoint of windows on e (in $O(|w_e|)$ time, where $|w_e|$ is the number of windows' endpoints on e). By having this information we report the k-visible segments of e restricted to the interval of the batch. Since the endpoints of windows are sorted according to their positions on ∂P, we do not check any member of W more than one time. It follows that the procedure of reporting the k-visible part of ∂P takes $O(n)$ time in each batch.

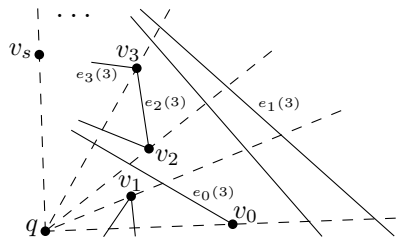

Fig. 4. The vertices $v_0, v_1, \ldots, v_s$ are the first batch of s vertices in angular order. The edge $e_1(3)$ is the second right neighbour of $e_0(3)$ because v_1 is an end vertex. The edge $e_2(3)$ is the second left neighbour of $e_1(3)$ which is inserted in T while processing v_2. The edge $e_3(3)$ is on the same chain as $e_2(3)$ because v_3 is a non-critical vertex.

After processing the first batch of vertices, we apply Lemma 4.1 to find the next batch of s vertices with smallest angle, and we sort them in memory in $O(s \log s)$ time. 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 4.1 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. Then similarly for each $s < i \leq 2s$ we find $e_i(k+1)$ and its corresponding window and we update T, W and E. Overall, finding a batch of s vertices, sorting and processing them, reporting the windows and the k-visible part of ∂P in the batch, take $O(n +$

$s \log s$) time. Moreover, we run the selection subroutine in the first batch. We repeat this procedure for $O(n/s)$ iterations, until all the vertices are processed. Thus, the running time of the algorithm is $O(n/s(n + s \log s)) + T_{\text{selection}}$. Since $T_{\text{selection}}$ is dominated, Theorem 4.3 is follows:

Theorem 4.3. *Given a simple polygon P with n vertices in a read-only array, a point $q \in P$ and a parameter $k \in \mathbb{N}$, there is an algorithm which reports $\partial V_k(P, q)$ in $O(n^2/s + n \log s)$ time using $O(s)$ words of workspace.*

4.2 Algo 2: processing only critical vertices

In this algorithm we process critical vertices in contiguous batches of size s in angular order. This algorithm works similarly as the algorithm in Section 4.1, but it differs in constructing and updating the data structure for finding the windows. In each iteration of this algorithm we find the next batch of s *critical* vertices with smallest angles and sort them in workspace memory in $O(s \log s)$ time. As in the previous algorithm, we construct a data structure T, which contains the possible candidates for the $(k+1)^{th}$ intersecting edges of the rays to critical vertices of the batch. In each step, we process a critical vertex, and we use T to find its corresponding window and we update T. For updating T efficiently, we use another data structure, which is called T_θ; see below. After finding the windows of the batch, we report the k-visible part of ∂P in the interval of the batch. We repeat the same procedure for the next s critical vertices.

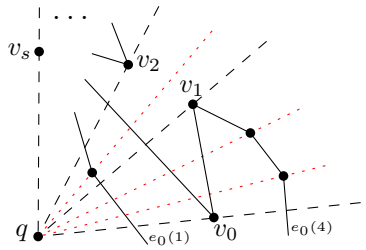

Fig. 5. The vertices $v_0, v_1, \ldots, v_s$ are the first batch of s critical vertices in angular order. The endpoint of the edge $e_0(1)$ is between qv_1 and qv_2, so $e_0(1)$ will be changed in T after processing v_1. The endpoint of $e_0(4)$ is between qv_0 and qv_1, so $e_0(4)$ will be changed in T after processing v_0.

In the first iteration, after computing $v_1, \ldots, v_s$, the angular sorted critical vertices after v_0, we 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 sort them and insert them in a balanced search tree T, which takes $O(n + s \log s)$ time. 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 edge in T should be updated to the other incident edge to the vertex; see Figure 5. 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 using $O(s)$ words of workspace.

Now for finding the $(k+1)^{th}$ intersecting edge of qv_1 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, so we change its incident edge, which has been already in T, with its other incident edge. Then we find the larger angle of the endpoints of the new edge and insert it in 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, with the same procedure as in the previous algorithm: if v_1 is an end (start) critical vertex, we remove (insert) the two edges which are incident to v_1, this can be done in $O(\log s)$ 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 all critical vertices in this batch. 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.

While processing the batch, we insert all $e_i(k+1)$, $1 \leq i \leq s$ into E. Also whenever we find a window we report it and we insert its endpoints, sorted according to the indices of the edges of P on which they lie, into a balanced search tree W in $O(\log s)$ time. After processing all the vertices of the batch, we use W (as a sorted array) and E to report the k-visible part of ∂P restricted to the interval of the batch. We know that, if there is no endpoint of a window on a chain, then the visibility of the chain is consistent on the entire chain. Using this, we walk on ∂P and for each chain C, we check if its endpoints, restricted to the interval of the batch, are k-visible or not (in $O(1)$ time using E). We also check whether there is any endpoint of windows on C (in $O(|w_e| + |C|)$ time using W, where $|w_e|$ is the number of windows' endpoints on the chain). Then we report the k-visible segments of C restricted to the interval of the batch. Since the endpoints of windows are sorted according to their positions on ∂P, we do not check any member of W more than one time. It follows that the procedure of reporting the k-visible part of ∂P takes $O(n)$ time in each batch.

In the next iteration, we repeat the same procedure for the next batch of critical vertices until 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 sorted critical vertices, constructing and updating the data structures and reporting $\partial V_k(P, q)$ take $O(cn/s + n \log s)$ total time, in addition to $T_{\text{selection}}$ in the first batch. This leads to the following theorem:

Theorem 4.4. *Given a simple polygon P with n vertices in a read-only array, a point $q \in P$ and a parameter $k \in \mathbb{N}$, there is an algorithm which reports $\partial V_k(P, q)$ in $O(cn/s + n \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ words of workspace, where c is the number of critical vertices of P.*

5 Variants and extensions

Our results can be extended in several ways; for example, computing the k-visibility region of a point q inside a polygon P, when P may have some holes, or computing the k-visibility region of a point q in presence of a set of non-crossing segments inside a bounding box in the plane, (the bounding box is only for bounding the k-visibility region). For the first problem, all the arguments in the algorithms for simple polygons hold for polygon with holes. The only remarkable issue is walking on ∂P to report the k-visible segments of ∂P. Here, after walking on the outer part of ∂P, we walk on the boundary of the holes one by one and we apply the same procedures for them. If there is no window on the boundary of a hole, then it is completely k-visible or completely non-k-visible. For such a hole, we check if it is k-visible and, if so, we report it completely. This leads to the following corollary:

Corollary 5.1. *Given a polygon P with $h \geq 0$ holes and n vertices in a read-only array, a point $q \in P$ and a parameter $k \in \mathbb{N}$, there is an algorithm which reports $\partial V_k(P, q)$ in $O(cn/s + n \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ words of workspace. Here c is the number of critical vertices of P.*

In the second problem for a set of n non-crossing segments inside a bounding box in the plane, the output is the part of the segments which are k-visible. Here, the endpoints of all segments are critical vertices and should be processed. In the parts of the algorithm where a walk on the boundary is needed, reading the input sequentially leads to similar results. Similarly, there may be some segments with no windows' endpoints on. For these we only need to check visibility of an endpoint to decide whether they are completely k-visible or completely non-k-visible. This leads to the following corollary:

Corollary 5.2. *Given a set of n non-crossing segments S in a read-only array which lie in a bounding box B, a point $q \in B$ and a parameter $k \in \mathbb{N}$, there is an algorithm which reports $V_k(S, q)$ in $O(n^2/s + n \log s)$ time using $O(s)$ words of workspace, where $V_k(S, q)$ is the k-visible subsets of segments in S from q.*

6 Conclusion

We have proposed algorithms for a class of k-visibility problems in the constrained-memory model, which provide time-space trade-offs for these problems. We leave it as an open question whether the presented algorithms are optimal. Also, it would be interesting to see whether there exists an output sensitive algorithm whose running time depends on the number of windows in the k-visibility region, instead of the critical vertices in the input polygon.

References

1. Aichholzer, O., Fabila Monroy, R., Flores Peñaloza, D., Hackl, T., Huemer, C., Urrutia Galicia, J., Vogtenhuber, B.: Modem illumination of monotone polygons. In: Proc. 25th EWCG. pp. 167–170 (2009)
2. Asano, T., Buchin, K., Buchin, M., Korman, M., Mulzer, W., Rote, G., Schulz, A.: Memory-constrained algorithms for simple polygons. CGTA 46(8), 959–969 (2013)
3. Bajuelos, A.L., Canales, S., Hernández-Peñalver, G., Martins, A.M.: A hybrid metaheuristic strategy for covering with wireless devices. J. UCS 18(14), 1906–1932 (2012)
4. Ballinger, B., Benbernou, N., Bose, P., Damian, M., Demaine, E.D., Dujmović, V., Flatland, R., Hurtado, F., Iacono, J., Lubiw, A., et al.: Coverage with k-transmitters in the presence of obstacles. In: Proc. 4th COCOA, pp. 1–15. Springer (2010)
5. Barba, L., Korman, M., Langerman, S., Sadakane, K., Silveira, R.I.: Space-time trade-offs for stack-based algorithms. Algorithmica 72(4), 1097–1129 (2015)
6. Barba, L., Korman, M., Langerman, S., Silveira, R.I.: Computing a visibility polygon using few variables. CGTA 47(9), 918–926 (2014)
7. Chan, T.M.: Comparison-based time-space lower bounds for selection. TALG 6(2), 26 (2010)
8. Chan, T.M., Chen, E.Y.: Multi-pass geometric algorithms. DCG 37(1), 79–102 (2007)
9. Chan, T.M., Munro, J.I., Raman, V.: Selection and sorting in the restore model. In: Proc. 25th SODA. pp. 995–1004. SIAM (2014)
10. Dean, A.M., Evans, W., Gethner, E., Laison, J.D., Safari, M.A., Trotter, W.T.: Bar k-visibility graphs: Bounds on the number of edges, chromatic number, and thickness. In: Proc. 13th GD. pp. 73–82. Springer (2005)
11. Dean, J.A., Lingas, A., Sack, J.R.: Recognizing polygons, or how to spy. The Visual Computer 3(6), 344–355 (1988)
12. Eppstein, D., Goodrich, M.T., Sitchinava, N.: Guard placement for efficient point-in-polygon proofs. In: Proc. 23rd SoCG. pp. 27–36. ACM (2007)
13. Fabila-Monroy, R., Vargas, A.R., Urrutia, J.: On modem illumination problems. In: Proc. 13th EGC (2009)
14. Felsner, S., Massow, M.: Parameters of bar k-visibility graphs. JGAA 12(1), 5–27 (2008)
15. Fulek, R., Holmsen, A.F., Pach, J.: Intersecting convex sets by rays. DCG 42(3), 343–358 (2009)
16. Ghosh, S.K.: Visibility algorithms in the plane. Cambridge University Press (2007)
17. Hartke, S.G., Vandenbussche, J., Wenger, P.: Further results on bar k-visibility graphs. SIAM J. on Discrete Mathematics 21(2), 523–531 (2007)
18. Joe, B., Simpson, R.B.: Corrections to Lee's visibility polygon algorithm. BIT Numerical Mathematics 27(4), 458–473 (1987)
19. Munro, J.I., Raman, V.: Selection from read-only memory and sorting with minimum data movement. TCS 165(2), 311–323 (1996)
20. O'Rourke, J.: Computational geometry column 52. ACM SIGACT News 43(1), 82–85 (2012)