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

Yeganeh Bahoo† Bahareh Banyassady‡ Prosenjit K. Bose§
Stephane Durocher† Wolfgang Mulzer‡

Abstract

Let P be a simple polygon with n vertices, and let $q \in P$ be a point in P. Let $k \in \{0, \dots, n-1\}$. A point $p \in P$ is *k-visible* from q if and only if the line segment pq crosses the boundary of P at most k times. The *k-visibility region* of q in P is the set of all points that are k-visible from q. We study the problem of computing the k-visibility region in the limited workspace model, where the input resides in a random-access read-only memory of $O(n)$ words, each with $\Omega(\log n)$ bits. The algorithm can read and write $O(s)$ additional words of workspace, where $s \in \mathbb{N}$ is a parameter of the model. The output is written to a write-only stream.

Given a simple polygon P with n vertices and a point $q \in P$, we present an algorithm that reports the k-visibility region of q in P in $O(cn/s + c\log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ words of workspace. Here, $c \in \{1, \dots, n\}$ is the number of *critical vertices* of P for q where the k-visibility region of q may change. We generalize this result for polygons with holes and for sets of non-crossing line segments.

Keywords: Limited workspace model, k-visibility region, Time-space trade-off

1 Introduction

Memory constraints on mobile devices and distributed sensors have led to an increasing focus on algorithms that use their memory efficiently. One common approach to capture this notion is the *limited workspace model* [3]. Here, the input is provided in a random-access read-only array of $O(n)$ words. Each word has $\Omega(\log n)$ bits. Additionally, there is a read/write memory with $O(s)$ words, where $s \in \{1, \dots, n\}$ is a parameter of the

*A preliminary version appeared as Y. Bahoo, B. Banyassady, P. Bose, S. Durocher, and W. Mulzer. *Time-Space Trade-off for Finding the k-Visibility Region of a Point in a Polygon.* Proc. 11th WALCOM, 2017. This work was partially supported by DFG project MU/3501-2, ERC StG 757609, and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

†Department of Computer Science, University of Manitoba, {bahoo, durocher}@cs.umanitoba.ca
‡Institut für Informatik, Freie Universität Berlin, {bahareh, mulzer}@inf.fu-berlin.de
§School of Computer Science, Carleton University, jit@scs.carleton.ca

model. This is called the *workspace* of the algorithm. The output is written to a write-only stream.

Let P be a simple polygon with n vertices and n edges, and let q be a point in P. Let $k \in \{0, \ldots, n-1\}$. A point $p \in P$ is *k-visible* from q if and only if the line segment pq has at most k proper intersections with the boundary ∂P of P (p and q do not count toward the number of intersections).[1] The set of k-visible points in P from q is called the *k-visibility region* of q in P; see Figure 1. We denote it by $V_k(P, q)$. For $k = 0$, this notion corresponds to classic visibility in polygons.

Visibility problems have played a major role in computational geometry since the very beginning of the field. Thus, there is a rich history of previous results; see the book by Ghosh [17] for an overview. The concept of 1-visibility first appeared in a work by Dean *et al.* [12] as far back as 1988. In the related *superman problem* [20], we are given two polygons P and G such that $G \subseteq P$, and a point $p \in P \setminus G$. The goal is to find the minimum number of edges in P that need to be made opaque in order to make G invisible from p. More general k-visibility, for $k > 1$, is more recent. Since 2009, this variant of visibility has been explored more widely due to its relevance in wireless networks. In particular, it models the coverage areas of wireless devices whose radio signals can penetrate up to k walls [2, 14]. This makes the problem particularly interesting for the limited workspace model, since these wireless devices are typically equipped with only a small amount of memory for computational tasks and may need to determine their coverage region using the few resources at their disposal.

The notion of k-visibility has previously been considered in the context of art-gallery-style questions [5, 13, 16, 22] and in the definition of certain geometric graphs [11, 15, 18]. While the 0-visibility region is always connected, the k-visibility region may have several components. Bajuelos *et al.* [4] present an algorithm for a slightly different notion of k-visibility. It computes the region of the *plane* which is k-visible from q in the presence of a simple polygon P with n vertices, using $O(n^2)$ time and $O(n^2)$ space. In this setting, the k-visibility region is connected. We believe that our ideas are also applicable for this notion and lead to an improvement of their result.[2]

Related work. The optimal classic algorithm for computing the 0-visibility region needs $O(n)$ time and $O(n)$ space [19]. In the *constant-workspace model* (i.e., for $s = 1$), the 0-visibility region of a point $q \in P$ can be reported in $O(n\bar{r})$ time, where $\bar{r}$ is the number of *reflex* vertices of P that occur in the output, as shown by Barba *et al.* [7]. This algorithm scans the boundary ∂P in counterclockwise order, and it reports the maximal subchains of ∂P that are 0-visible from q. More precisely, this works as follows: we find a vertex v_{start} of P that is 0-visible from q. Walking from v_{start}, we then go until the next reflex vertex v_{vis} that is 0-visible from q, in counterclockwise direction. This takes

[1]For $k = n - 1$, the whole polygon is k-visible from q, so there is no reason to consider $k > n - 1$.

[2]The algorithm of Bajuelos *et al.* [4] essentially first computes a complete arrangement of quadratic size that encodes the whole visibility information, and then extracts the k-visible region from this arrangement. Our algorithms, on the other hand, use a plane sweep so that only the relevant parts of this arrangement are considered. Thus, when $O(n)$ words of workspace are available, we achieve a running time of $O(n \log n)$.

$O(n)$ time. The first intersection of the ray qv_{vis} with ∂P is called the *shadow* of v_{vis}. Now, the end vertex of the maximal counterclockwise visible chain starting at v_{start} is either v_{vis} or its shadow. In each case, the next maximal visible chain starts at the other of the two vertices (v_{vis} or its shadow). Thus, we can find a maximal visible chain and a new starting point in $O(n)$ time. The number of iterations is $\bar{r}$, the number of reflex vertices that are 0-visible from q. This gives an algorithm with $O(n\bar{r})$ running time and $O(1)$ workspace.

Now suppose that the number of reflex vertices in P with respect to q is r. If the available workspace is $O(s)$, for $s \in \{1, \dots, O(\log r)\}$, Barba *et al.* [7] show how to find the 0-visibility region of q in P in $O(nr/2^s + n\log^2 r)$ deterministic time or $O(nr/2^s + n\log r)$ expected time. Their method is recursive. It uses the previous algorithm as the base, and in each step of the recursion, it splits a chain on ∂P into two subchains that each contains roughly half of the visible reflex vertices of the original chain. Since the 0-visibility region and the k-visibility region of q for $k > 0$ have different properties, there seems to be no straightforward way to generalize this approach to our setting. Later, Barba *et al.* [6] provided a general method for obtaining time-space trade-offs for *stack-based* algorithms. This gives an alternative trade-off for computing the 0-visibility region: there is an algorithm that runs in $O(n^2 \log n/2^s)$ time for $s = o(\log n)$ and in $n^{1+O(1/\log s)}$ time for $s \geq \log n$.[3] Again, this approach does not seem to be directly applicable to our setting.

Abrahamsen [1] presents a constant workspace algorithm that computes the visible part of one edge from another edge in a simple polygon P in $O(n)$ time, where n is the number of vertices in P. This gives an algorithm that needs $O(mn)$ time and $O(1)$ words of workspace to compute the weak visibility region of one edge in P. The parameter m denotes the size of the resulting weak visibility polygon.

Our Results. We look at the more general problem of computing the k-visibility region of a simple polygon P for a given point $q \in P$. We give a constant workspace algorithm for this problem, and we establish a time-space trade-off. Our first algorithm runs in $O(kn + cn)$ time using $O(1)$ words of space, and our second algorithm requires $O(cn/s + c\log s + \min\{\lceil k/s \rceil n, n \log\log_s n\})$ expected time and $O(s)$ words of workspace. Here, $c \in \{1, \dots, n\}$ is the number of *critical vertices* of P for q, where the k-visibility region of q may change. A precise definition is given later.

We generalize this result for polygons with holes and for sets of non-crossing line segments. More precisely, we show that in a polygon P with h holes, we can report the k-visibility region of a point $q \in P$ in expected time $O(cn/s + c\log s + \min\{\lceil k/s \rceil n, n \log\log_s n\})$ using $O(s)$ words of workspace. In an arrangement of n pairwise non-crossing line segments, this takes $O(n^2/s + n\log s)$ deterministic time.

[3] The actual trade-off is more nuanced, but we simplified the bound to make it more digestible for the casual reader.

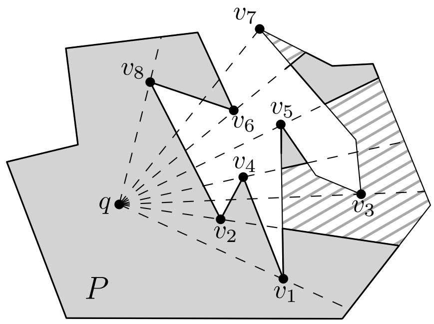


Figure 1: An example with $k = 2$. The hatched regions are not 2-visible for q. The vertices $v_1, \dots, v_8$ are critical for q. More precisely, v_1, v_2, v_3, v_6 are start vertices, and v_4, v_5, v_7, v_8 are end vertices. ∂P is partitioned into 8 disjoint chains, e.g, the counterclockwise chain $v_3 v_5$.

2 Preliminaries and Definitions

Let $s \in \{1, \dots, n\}$ be the amount of available workspace, measured in words. We assume that the input polygon P is given as a sequence of n vertices in counterclockwise (CCW) order along ∂P. The input also contains the query point $q \in P$ and the visibility parameter $k \in \{0, \dots, n-1\}$. The aim is to report $V_k(P, q)$, using $O(s)$ words of workspace. We require that the input is in *weak general position*, i.e., the query point q does not lie on any line through two distinct vertices of P. Without loss of generality, we assume that k is even: if k is odd, we can just compute $V_{k-1}(P, q)$, which is the same as $V_k(P, q)$, by definition. The boundary $\partial V_k(P, q)$ of $V_k(P, q)$ consists of pieces of ∂P and chords of P that connect two such pieces; see Figure 1.

We fix a coordinate system with origin q. For $\theta \in [0, 2\pi)$, we denote by r_θ the ray that emanates from q and has CCW-angle θ with the x-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 list of intersecting edges of r_θ, sorted according to their intersection with r_θ, in increasing distance from q. The j^{th} element of this list is denoted by $e_\theta(j)$. We also say that $e_\theta(j)$ has *rank* j in the edge list of r_θ, or simply on r_θ.

The *angle* of a vertex v of P refers to the angle $\theta \in [0, 2\pi)$ at which r_θ encounters v. Suppose r_θ stabs a vertex v of P. We call v a *critical* vertex if its incident edges lie on the same side of r_θ, and a *non-critical* vertex otherwise. We can check in constant time whether a given vertex of P is critical. We use c to denote the number of critical vertices in P. Let v be a critical vertex. We call v a *start vertex* if both incident edges lie counterclockwise of r_θ, and an *end vertex* otherwise; see Figure 1. A *chain* is a sequence of edges of P (in CW or CCW order along ∂P) which starts at a start vertex and ends at an end vertex and contains no other critical vertices. Note that every ray r_θ intersects each chain at most once. Thus, we will sometimes talk of *chains* that appear in the edge list of a ray r_θ.

Suppose we continuously increase θ from 0 to 2π. The edge list of r_θ only changes when r_θ encounters a vertex v of P. This change only involves the two edges incident to v. At a non-critical vertex v, the edge list is updated by replacing one incident edge of v with the other. The other edges and their order in the edge list do not change. At a critical vertex v, the edge list is updated by adding or removing both incident edges of v, depending on whether v is a start vertex or an end vertex. The other edges and their order in the edge list are not affected; see Figure 1. If r_θ stabs a start vertex of P, we define the edge list of r_θ to be the edge list of $r_{\theta+\varepsilon}$, for a small enough $\varepsilon > 0$. If r_θ stabs an end vertex or a non-critical vertex of P, we define the edge list of r_θ to be the edge list of $r_{\theta-\varepsilon}$, for a small enough $\varepsilon > 0$.

For any $\theta \in [0, 2\pi)$, only the first $k+1$ elements in the edge list of r_θ are k-visible from q in direction θ. While increasing θ, as long as r_θ does not encounter a critical vertex, the k-visible chains in direction θ do not change. However, if r_θ encounters a critical vertex v, then this may affect which chains are visible from q. This happens if at least one of the incident edges to v is among the first $k+1$ elements in the edge list of r_θ. In other words, if v is k-visible from q, which means that v does not lie after $e_\theta(k+1)$ on r_θ. The next lemma shows that in this case a segment on r_θ may occur on $\partial V_k(P, q)$.

Lemma 2.1. *Let $\theta \in [0, 2\pi)$ such that r_θ stabs a k-visible end or start vertex v. Then, the segment on r_θ between $e_\theta(k+2)$ and $e_\theta(k+3)$ is an edge of $V_k(P, q)$, provided that these two edges exist.*

Proof. Suppose that v is a k-visible end vertex. As mentioned above, right after r_θ encounters v, two consecutive edges are removed from the edge list of r_θ. Since v is k-visible, these edges are among the first $k+2$ entries in the edge list. Thus, right after v, the k-visibility region of q extends to $e_\theta(k+3)$ (recall that the indices refer to the situation just before v). Before v, the k-visibility region extends to $e_\theta(k+1)$. This means that the segment between $e_\theta(k+2)$ and $e_\theta(k+3)$ on r_θ belongs to $\partial V_k(P, q)$. In particular, this includes the case that $e_\theta(k+1)$ and $e_\theta(k+2)$ are incident to v. The situation for a k-visible start vertex v is symmetric. Note that in this case, the indices in the edge list refer to the situation just after v; see Figure 2. □

Lemma 2.1 leads to the following definition: let $\theta \in [0, 2\pi)$ such that r_θ stabs a k-visible end or start vertex v. The segment on r_θ between $e_\theta(k+2)$ and $e_\theta(k+3)$, if these edges exist, is called the *window* of r_θ; see Figure 2.

Observation 2.2. *The k-visibility region $V_k(P, q)$ has $O(n)$ vertices.*

Proof. The boundary $\partial V_k(P, q)$ consists of subchains of ∂P and of 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, since each window has two endpoints, and since there are at most n critical vertices, the total number of vertices of $V_k(P, q)$ is $O(n)$. □

3 A Constant-Memory Algorithm

First, we assume that a constant amount of workspace is available. If the input polygon P has no critical vertex, there is no window, and $V_k(P, q) = P$. This can be checked in

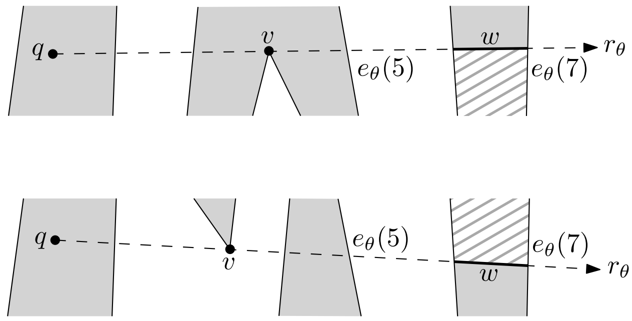


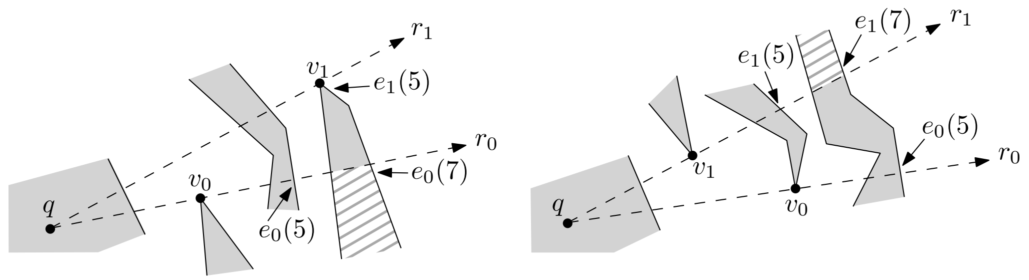
Figure 2: An example with $k = 4$. The hatched regions are not 4-visible for q. (a) The ray r_θ encounters the end vertex v. The 4-visibility region of q right before v extends to $e_\theta(5)$ and right after v extends to $e_\theta(7)$. (b) The ray r_θ encounters the start vertex v. The 4-visibility region of q right before v extends to $e_\theta(7)$ and right after v extends to $e_\theta(5)$. The segment w in both figures is the window of r_θ.

$O(n)$ time by a simple scan through the input. Thus, we assume that P has at least one critical vertex v_0. Again, v_0 can be found in $O(n)$ time with a single scan. We choose our coordinate system such that q is the origin and such that v_0 lies on the positive x-axis. We number the critical vertices of P as $v_0, v_1, \dots, v_{c-1}$ in the order that the ray r_θ encounters them. Let θ_i be the angle for v_i. We simplify our notation and write r_i instead of r_{θ_i}, and we let $e_i(j)$ denote the j^{th} entry in the edge list of the ray r_i.

We start with the ray r_0, and we find the edge $e_0(k+1)$ in $O(kn)$ time using $O(1)$ words of workspace. For this, we perform a simple *selection* subroutine as follows: we scan the input $k+1$ times, and in each pass, we find the next intersecting edge of r_0 until $e_0(k+1)$. If v_0 is k-visible, i.e., if it is not after $e_0(k+1)$ on r_0, we report the window of r_0, as given by Lemma 2.1 (if it exists). Since the window is defined by $e_0(k+2)$ and $e_0(k+3)$, it can be found in two more scans over the input.

Next, we find v_1 by a single scan of ∂P. Then, we determine $e_1(k+1)$. This can be done in $O(n)$ time by using $e_0(k+1)$ as a starting point: we know that if v_0 is an end vertex, the two incident chains of v_0 disappear in the edge list of r_1. If v_1 is a start vertex, the two incident chains of v_1 appear in the edge list of r_1. All other chains are not affected, and they intersect r_0 and r_1 in the same order. Using this, we first find the edge e' that has rank $k+1$ in the edge list of the ray $r_{\theta_0+\varepsilon}$ just after r_0. Depending on the type and position of v_0, e' is either $e_0(k+1)$ or $e_0(k+3)$, and it can be found in $O(n)$ time. Then, by scanning ∂P starting from e', we can find the edge e'' on the chain of e' that intersects the ray $r_{\theta_1-\varepsilon}$ just before r_1, again in $O(n)$ time. Depending on the type and position of v_1, the edge e'' is either $e_1(k+1)$ or $e_1(k+3)$. Thus, we can find $e_1(k+1)$ using e'' in $O(n)$ time; see Figure 3.

If v_1 is k-visible, we report the window of r_1 in $O(n)$ time, as described above. Finally, we report the subchains of $\partial V_k(P,q)$ between r_0 and r_1 by scanning ∂P. More precisely, we walk along ∂P in counterclockwise direction. Whenever we enter the counterclockwise cone between r_0 and r_1, we check whether the intersection between ∂P and r_0 or r_1 occurs



(a) (b)

Figure 3: Two cases for going from v_0 to v_1, with $k = 4$. (a) Both v_0 and v_1 are end vertices. We use $e_0(5)$ to find $e_0(7)$ and follow the chain until $e_1(5)$. (b) Both v_0 and v_1 are start vertices. We follow the chain of $e_0(5)$ until $e_1(7)$, and then use it to find $e_1(5)$. We report the window from $e_1(6)$ to $e_1(7)$.

at or before $e_0(k+1)$ or $e_1(k+1)$, respectively. If so, we report the subchain of ∂P until we leave the cone again.

We repeat this procedure until all critical vertices have been processed; see Algorithm 3.1. Here and in the following algorithms, if there are less than $k+1$ intersecting edges on r_i, we store the last intersecting edge together with its rank. We use this edge instead of $e_i(k+1)$, in the procedure above, to find $e_{i+1}(k+1)$ or the last intersecting edge of r_{i+1} and its rank. The number of critical vertices is c. For each of them, we spend $O(n)$ time. Additionally, the selection subroutine for v_0 takes $O(kn)$ time. This leads to the following theorem:

Algorithm 3.1: The constant workspace algorithm for computing $V_k(P, q)$

input: Simple polygon P, point $q \in P$, $k \in \mathbb{N}$
output: The boundary of the k-visibility region of q in P, $\partial V_k(P, q)$

```
if P has no critical vertex then
    return ∂P
v_0 ← a critical vertex of P
Find e_0(k+1) using selection
i ← 0
repeat
    if v_i lies on or before e_i(k+1) on r_i then
        Report the window of r_i (if it exists)
    v_{i+1} ← the next counterclockwise critical vertex after v_i
    Find e_{i+1}(k+1) using e_i(k+1)
    Report the part of ∂V_k(P, q) between r_i and r_{i+1}
    i ← i + 1
until v_i = v_0
```

Theorem 3.1. *Given a simple polygon P with n vertices, a point $q \in P$, and a parameter $k \in \{0, \ldots, n-1\}$, we can report the k-visibility region of q in P in $O(kn + cn)$ time using $O(1)$ words of workspace, where c is the number of critical vertices of P.*

4 Time-Space Trade-Offs

In this section, we assume that we have $O(s)$ words of workspace at our disposal, and we show how to exploit this additional workspace to compute the k-visibility region faster. We describe two algorithms. The first algorithm is simpler, and it is meant to illustrate the main idea behind the trade-off. Our main contribution is in the second algorithm, which is more complicated but achieves a better running time. In the first algorithm, we process the vertices in angular order in contiguous batches of size s. 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 that is used to output the windows of the batch. Using the windows, we report $\partial V_k(P, q)$ between the first and the last ray of the batch.[4] 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 as before, we construct a data structure for finding the windows. We need a more involved approach in order to maintain this data structure. The next lemma shows how to obtain the contiguous batches of vertices in angular order efficiently. The procedure is taken from the work of Chan and Chen [9] (see the second paragraph in the proof of Theorem 2.1 in [9]).

Lemma 4.1. *Suppose we are given a read-only array A with n pairwise distinct elements from a totally ordered universe and an element $x \in A$. For any given parameter $s \in \{1, \ldots, n\}$, there is an algorithm that runs in $O(n)$ time and uses $O(s)$ words of workspace and that finds the set of the first s elements in A that follow x in the sorted order.*

Proof. Let $A_{>x}$ be the subsequence of A that contains exactly the elements in A that are larger than x. The algorithm makes a single pass over $A_{>x}$ and processes the elements in batches. In the first step, we insert the first $2s$ elements of $A_{>x}$ into our workspace (without sorting them). We select the median of these $2s$ elements using $O(s)$ time and space, and we remove the elements which are larger than the median. In the next step, we insert the next batch of s elements from $A_{>x}$ into the workspace, and we again find the median of the resulting $2s$ elements and remove those elements that are larger than the median. We repeat the latter step until all the elements of $A_{>x}$ have been processed. Clearly, at the end of each step, the s smallest elements of $A_{>x}$ that we have seen so far reside in memory. Since the number of steps is $O(n/s)$ and since each step needs $O(s)$ time, the running time of the algorithm is $O(n)$. By construction, it uses $O(s)$ words of workspace. □

[4] We emphasize that $\partial V_k(P, q)$ is not necessarily reported in order, but we ensure that the union of the reported line segments constitutes the boundary of the k-visibility region.

Lemma 4.2. *Suppose we are given a read-only array A with n elements from a totally ordered universe and a number $k \in \{1, \ldots, n-1\}$. For any given parameter $s \in \{1, \ldots, n\}$, there is an algorithm that runs in $O(\lceil k/s \rceil n)$ time and uses $O(s)$ words of workspace and that finds the k^{th} smallest element in A.*

Proof. We again process the elements of A in batches. In the first step, we apply Lemma 4.1 to find the first batch with the s smallest elements in A and to put it into our workspace. This needs $O(n)$ time and $O(s)$ words of workspace. If $k \leq s$, we select the k^{th} smallest element in the workspace in $O(s)$ time; otherwise, we find the largest element x in the workspace, and we apply Lemma 4.1 to find the set of s elements following x. In step i, we apply Lemma 4.1 to find the i^{th} batch of s elements in the sorted order of A and to insert this set of elements into the workspace. If $k \leq i \cdot s$, we select the $(k-(i-1)s)^{\text{th}}$ smallest element in the workspace in $O(s)$ time and we output it; otherwise, we find the largest element in the workspace and we continue. The element being sought is in the $\lceil k/s \rceil^{\text{th}}$ batch. Therefore, we can find it in $O(\lceil k/s \rceil n)$ time using $O(s)$ words of workspace. □

In addition to the simple algorithm in Lemma 4.2, there are several other results on selection in the read-only model; see Table 1 of [10]. In particular, there is a $O(n \log \log_s n)$ expected time randomized algorithms for selection using $O(s)$ words of workspace in the limited workspace model [8, 21]. Depending on k, s, and n, we will choose one of the latter algorithms or the algorithm that we presented in Lemma 4.2. In conclusion, the running time of selection in the limited workspace model using $O(s)$ words of workspace, denoted by $T_{\text{selection}}$, is $O(\min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time.

4.1 First Algorithm: Processing All the Vertices

Let v_0 be some vertex of P. We choose our coordinate system such that q is the origin and such that v_0 lies on the positive x-axis. We apply Lemma 4.1 to find the batch of s vertices with the smallest positive angles, and we sort them in workspace in $O(s \log s)$ time. Let $v_1, \ldots, v_s$ denote these vertices in sorted order. We use the selection subroutine (with $O(s)$ words of workspace) to find $e_0(k+1)$ on r_0, and if v_0 is a k-visible vertex, i.e., if it does not occur after $e_0(k+1)$ on r_0, we report its window (if it exists). Recall that if there are less than $k+1$ intersecting edges on r_0, we store the last intersecting edge together with its rank.

Then, we apply Lemma 4.1 four times in order to find the at most $4s+1$ intersecting edges with ranks in $\{k-2s+1, \ldots, k+2s+1\}$ on r_0 (Lemma 4.1 can be applied, because we have $e_0(k+1)$ at hand). We insert these edges into a balanced binary search tree T, sorted according to their ranks on r_0. The edges in T are candidates for having rank $k+1$ on the next s rays $r_1, \ldots, r_s$. This is because, as we explained in Section 3, if $e_i(k+1)$ belongs to the edge list of r_{i-1}, there is at most one edge between $e_{i-1}(k+1)$ and $e_i(k+1)$ in the edge list of r_{i-1}. Therefore, if $e_i(k+1)$ appears in the edge list of r_0, there are at most $2i-1$ edges between $e_0(k+1)$ and $e_i(k+1)$ in the edge list of r_0.

Now the algorithm proceeds as follows: we go to the next vertex v_1, and we update T depending on the types of v_0 and v_1: if v_0 is a non-critical vertex, we may need to

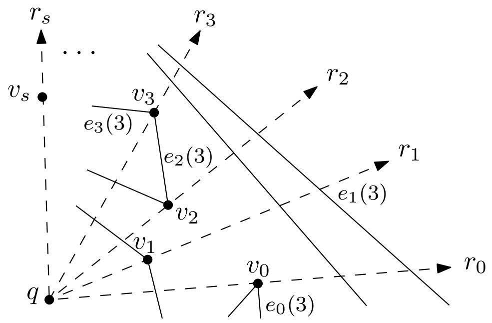


Figure 4: The first batch $v_0, v_1, \ldots, v_s$ of s vertices in angular order. The edge $e_1(3)$ is the second neighbor to the right of $e_0(3)$ on r_0, because v_0 is an end vertex. The edge $e_2(3)$ is the second neighbor to the left of $e_1(3)$ which is inserted in T before processing v_2. The edge $e_2(3)$ is exchanged with $e_3(3)$, after processing v_3, because v_3 is a non-critical vertex.

exchange one incident edge of v_0 with another in T; if v_0 is an end vertex, we may need to remove its incident edges from T; and if v_1 is a start vertex, we may need to insert its incident edges into T. In all other case, no action is necessary. The insertion and/or deletion is performed only for the edges whose ranks are between the smallest and the largest rank in T (with respect to r_1). The update of T takes $O(\log s)$ time. Afterwards, we can find $e_1(k+1)$ and the window of r_1 (if it exists) in $O(1)$ time, using the position of $e_0(k+1)$ or its neighbors in T, as explained in Section 3. See Figure 4 for an example.

We repeat this procedure for $v_2, \ldots, v_s$. We use, for $i = 2, \ldots, s$, the binary search tree T and the previous edge $e_{i-1}(k+1)$ in order to determine the next edge $e_i(k+1)$ and the window of r_i. This takes $O(s \log s)$ total time. Whenever we find and report a window, we insert its endpoints into a balanced binary search tree W. This takes $O(\log s)$ time per window. The endpoints in W are sorted according to their counterclockwise order along ∂P. For reporting the part of $\partial V_k(P, q)$ between r_0 and r_s, we use W and the sequence $E = e_0(k+1), e_1(k+1), \ldots, e_s(k+1)$ of edges of rank $k+1$.

For an edge e of P, the $0s$-*segment* of e is the subsegment of e that lies between r_0 and r_s. If a $0s$-segment does not contain an endpoint of a window, then it is either completely k-visible or completely not k-visible. Thus, we can walk along ∂P and, simultaneously, along the window endpoints in W. For each edge e of P, we can check if the endpoints of the $0s$-segment of e are k-visible or not. We can do this in $O(1)$ time using E. With the help of the parallel traversal of W, we can also check if there is a window endpoint on e. This takes $O(|w_e|)$ time, where $|w_e|$ is the number of window endpoints on e. With this information, we can report the k-visible subsegments of the $0s$-segment of e. Since there are $O(n)$ window endpoints by Observation 2.2, and since we check each window endpoint once, it follows that we need $O(n)$ time to report the k-visible part of ∂P between r_0 and r_s.

After processing $v_0, \ldots, v_s$, we apply Lemma 4.1 to find the next batch of s vertices following v_s in angular order. We sort them in $O(s \log s)$ time, using $O(s)$ words of workspace. The search tree T for the previous batch is not useful anymore, because it does not necessarily contain any right or left neighbor of $e_s(k+1)$ on r_s. Applying Lemma 4.1 four times as before, we find the at most $4s+1$ intersecting edges with ranks in $\{k-2s+1, \ldots, k+2s+1\}$ on r_s, and we insert them into T. Then, as before, for each $s < i \leq 2s$, we find $e_i(k+1)$ and its corresponding window while maintaining T, W, and E. After that, we report the k-visible part of ∂P between r_s and r_{2s}, where r_{2s} is the ray for the last vertex in the batch, in sorted order. If n is not divisible by s, the last batch wraps around, taking the indices modulo n, but we report only the part of $\partial V_k(P, q)$ before $r_n = r_0$; see Algorithm 4.1.

Algorithm 4.1: Computing $\partial V_k(P, q)$ using $O(s)$ words of workspace

input: Simple polygon P, point $q \in P$, $k \in \mathbb{N}$, $1 \leq s \leq n$
output: The boundary of k-visibility region of q in P, $\partial V_k(P, q)$

$v_0 \leftarrow$ a vertex of P
$E \leftarrow \langle e_0(k+1) \rangle$ (using the selection subroutine with $O(s)$ workspace)
T, $W \leftarrow$ an empty balanced binary search tree
$i \leftarrow 0$
repeat
    $v_{i+1}, \ldots, v_{i+s} \leftarrow$ sorted list of s vertices following v_i in angular order
    $T \leftarrow$ at most $4s+1$ edges with rank in $\{k-2s+1, \ldots, k+2s+1\}$ on r_i
    for $j = i$ *to* $i+s-1$ **do**
        if v_j *lies on or before* $e_j(k+1)$ *on* r_j **then**
            Report the window of r_j (if it exists)
            Insert the endpoints of the window into W (according to their position on ∂P)
        Update T according to the types of v_j and v_{j+1}
        E.append($e_{j+1}(k+1)$) (find it using $e_j(k+1)$ and T)
    Report the part of $\partial V_k(P, q)$ between r_i and $r_{\min\{i+s,n\}}$ (using W and E)
    $i \leftarrow i + s$
until $i \geq n$

Overall, we need $O(n + s \log s)$ time for a batch. We repeat this procedure for $O(n/s)$ iterations, until all vertices are processed. Moreover, we run the selection subroutine in the first batch. Thus, the running time of the algorithm is $O(n/s(n + s \log s)) + T_{\text{selection}}$. Since $T_{\text{selection}}$ is dominated by the other terms, we obtain the following theorem.

Theorem 4.3. *Let $s \in \{1, \ldots, n\}$. Given a simple polygon P with n vertices in a read-only array, a point $q \in P$ and a parameter $k \in \{0, \ldots, n-1\}$, we can report the k-visibility region of q in P in $O(n^2/s + n \log s)$ time using $O(s)$ words of workspace.*

4.2 Second Algorithm: Processing only the Critical Vertices

As in Section 4.1, we process the vertices in batches, but now we focus only on the critical vertices. The new algorithm is similar to the algorithm in Section 4.1, but it

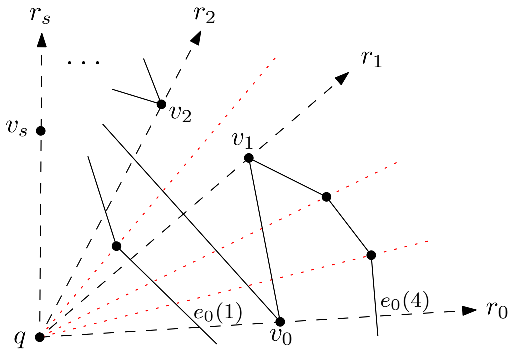


Figure 5: The first batch $v_0, v_1, \ldots, v_s$ of s critical vertices in angular order. The non-critical endpoint of $e_0(1)$ is between r_1 and r_2, so $e_0(1)$ will be replaced in T right before processing v_2. The non-critical endpoint of $e_0(4)$ is between r_0 and r_1, so $e_0(4)$ will be replaced in T right before processing v_1.

handles the data structure for the intersecting edges differently. In each iteration, we find the next batch of s *critical* vertices, and we sort them in $O(s \log s)$ time using $O(s)$ words of workspace. As in the previous algorithm, we construct a data structure T that contains the possible candidates for the edges of rank $k+1$ on the rays for the s critical vertices of the batch. In each step, we process the next critical vertex. We use T to find the corresponding window, and we update T. To update T efficiently, we use an auxiliary data structure T_{aux}; see below. After finding all the windows of the batch, we report the k-visible part of ∂P between the first and the last ray of the batch.

As in Section 3, if P has no critical vertex, then $V_k(P, q) = P$. This can be checked in $O(n)$ time by a simple scan through the input. Thus, we let v_0 be some critical vertex, and we choose our coordinate system such that q is the origin and such that v_0 lies on the positive x-axis. In the first iteration, we compute $v_1, \ldots, v_s$, the list of s critical vertices after v_0, sorted in angular order. Using Lemma 4.1 and a traditional sorting algorithm, this takes $O(n + s \log s)$ time and $O(s)$ words of workspace. We find $e_0(k+1)$ using our selection subroutine, and the at most $4s+1$ intersecting edges with rank in $\{k-2s+1, \ldots, k+2s+1\}$ on r_0. We insert them into a balanced binary search tree T, ordered according to their rank on r_0. This takes $T_{\text{selection}} + O(n + s \log s)$ time. Then, for each edge e in T, we determine whether it has a non-critical endpoint between r_0 and r_s. We insert all these non-critical endpoints into a balanced binary search tree T_{aux}, sorted according to their angle. The vertices in T_{aux} have cross-pointers to their corresponding edges in T. We can construct T_{aux} in $O(s \log s)$ time using $O(s)$ words of workspace. We use T_{aux} to determine which edges in T need to be updated between two critical vertices; see Figure 5.

Now, to find $e_1(k+1)$, we update T so that it contains the edge list of r_1. This is done as follows: for each non-critical vertex v in T_{aux} that lies between r_0 and r_1, we walk along the chain C containing v to find the edge e of C that intersects r_1. The edge e exists, since there is no critical vertex between r_0 and r_1 that could be the endpoint of the chain C. If the endpoint of e that lies after r_1 is non-critical, we insert it into T_{aux}. Furthermore, we replace the corresponding edge of v in T with e. This takes $O(s \log s + n_1)$ time, where n_1 is the number of non-critical vertices between r_0 and r_1. Then, we update T and T_{aux} according to the types of v_0 and v_1, as in the previous algorithm: if v_0 is an end vertex, we remove the two incident edges from T, and if v_1 is a start vertex, we insert the two incident edges of v_1 into T. This can be done in $O(\log s)$ time. Now, T contains at most $4s+1$ intersecting edges of r_1, and we can find $e_1(k+1)$ using the chain of $e_0(k+1)$ and its neighbors in T in $O(1)$ time. We repeat this procedure for all critical vertices in the batch. In total, processing the changes in T that are caused by critical and non-critical vertices of the batch takes $O(s \log s + n')$ time, where n' is the number of non-critical vertices that lie between r_0 and r_s.

While processing the batch, we insert all $e_i(k+1)$, $0 \leq i \leq s$, into E. Also, whenever we find and report a window, we insert its endpoints, sorted according to their counter-clockwise order along ∂P, into a balanced binary search tree W, in $O(\log s)$ time. After processing all the vertices of the batch, we use W and E to report the part of $\partial V_k(P, q)$ between r_0 and r_s, as in Section 4.1. The only difference is that now we keep track of the visibility of the whole chains between r_0 and r_s instead of individual edges. As before, this takes $O(n)$ time.

In the subsequent iteration, we repeat the same procedure for the next batch of s critical vertices. We repeat until all critical vertices are processed; see Algorithm 4.2. By construction, each non-critical vertex is handled in exactly one iteration. Since there are $O(c/s)$ iterations, updating T takes $O(c \log s + n)$ time in total. All together, we get a total running time of $O(cn/s + c \log s)$, in addition to $T_{\text{selection}}$ in the first batch. This leads to the following theorem:

Theorem 4.4. *Let $s \in \{1, \ldots, n\}$. Given a simple polygon P with n vertices in a read-only array, a point $q \in P$ and a parameter $k \in \{0, \ldots, n-1\}$, we can report the k-visibility region of q in P in $O(cn/s + c \log s + \min\{\lceil k/s \rceil n, n \log \log_s n\})$ expected time using $O(s)$ words of workspace, where c is the number of critical vertices of P for q.*

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, where P may have holes, or computing the k-visibility region of a point q in a planar arrangement of n non-crossing segments inside a bounding box (the bounding box is only for bounding the k-visibility region). Concerning the first extension, all the properties we showed to hold for the algorithms for simple polygons also hold for the case with holes. The only noteworthy issue is the use of ∂P to report the k-visible segments of ∂P. In the case of polygons with holes, after walking on the outer part of ∂P, we walk on the boundaries 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 either 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:

Algorithm 4.2: Computing $\partial V_k(P, q)$ using $O(s)$ words of workspace

input: Simple polygon P, point $q \in P$, $k \in \mathbb{N}$, $1 \leq s \leq n$

output: The boundary of k-visibility region of q in P, $\partial V_k(P, q)$

$v_0 \leftarrow$ a critical vertex of P

$E \leftarrow \langle e_0(k+1) \rangle$ (using the selection subroutine with $O(s)$ workspace)

T, T_{aux}, $W \leftarrow$ an empty balanced binary search tree

$i \leftarrow 0$

repeat

- $v_{i+1}, \ldots, v_{i+s} \leftarrow$ sorted list of s critical vertices following v_i in angular order
- $T \leftarrow$ at most $4s + 1$ edges with rank in $\{k - 2s + 1, \ldots, k + 2s + 1\}$ on r_i
- $T_{\text{aux}} \leftarrow$ for each edge in T, its non-critical endpoint between r_i and r_{i+s} (if it exists)
- **for** $j = i$ *to* $i + s - 1$ **do**
  - **if** v_j *lies on or before* $e_j(k+1)$ *on* r_j **then**
    - Report the window of r_j (if it exists)
    - Insert the endpoints of the window into W (according to their position on ∂P)
  - **for** *any* $v \in T_{aux}$ *between* r_j *and* r_{j+1} **do**
    - Find the edge e on v's chain that intersects r_{j+1}
    - Exchange the corresponding edge of v in T with e
    - If e has a non-critical endpoint between r_{j+1} and r_{i+s}, insert it into T_{aux}
  - Update T according to the types of v_j and v_{j+1}
  - E.append($e_{j+1}(k+1)$) (find it using $e_j(k+1)$ and T)
- Report the part of $\partial V_k(P, q)$ between r_i and $r_{\min\{i+s,n\}}$ (using W and E)
- $i \leftarrow i + s$

until $i \geq n$

Corollary 5.1. *Let $s \in \{1, \ldots, n\}$. 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 $k \in \{0, \ldots, n-1\}$, we can report the k-visibility region of q in P in $O(cn/s + c\log s + \min\{\lceil k/s \rceil n, n \log\log_s n\})$ expected time using $O(s)$ words of workspace. Here, c is the number of critical vertices of P for the point q.*

Concerning the second problem, for a planar arrangement of n non-crossing segments inside a bounding box, the output consists of the k-visible parts of the segments. All the segments endpoints are critical vertices and should be processed. In the parts of the algorithm where a walk on the boundary is needed, a sequential scan of the input leads to similar results. Similarly, there may be some segments with no window endpoints. 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. *Let $s \in \{1, \ldots, n\}$. Given a set S of n non-crossing planar segments in a read-only array that lie in a bounding box B, a point $q \in B$ and a parameter $k \in \{0, \ldots, n-1\}$, there is an algorithm that reports the k-visible subsets of segments in S from q in $O(n^2/s + n\log s)$ time using $O(s)$ words of workspace.*

6 Conclusion

We have proposed algorithms for a class of k-visibility problems in the limited workspace model, and we have provided 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.

Finally, our ideas are also applicable to the slightly different definition of k-visibility used by Bajuelos et al. [4]. Thus, our techniques can be used to improve their result, achieving $O(n \log n)$ running time if $O(n)$ words of workspace are available.

References

[1] M. Abrahamsen. An optimal algorithm computing edge-to-edge visibility in a simple polygon. In *Proc. 25th Canad. Conf. Comput. Geom. (CCCG)*, 2013.

[2] O. Aichholzer, R. Fabila Monroy, D. Flores Peñaloza, T. Hackl, C. Huemer, J. Urrutia Galicia, and B. Vogtenhuber. Modem illumination of monotone polygons. In *Proc. 25th European Workshop Comput. Geom. (EWCG)*, pages 167–170, 2009.

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

[4] A. L. Bajuelos, S. Canales, G. Hernández-Peñalver, and A. M. Martins. A hybrid metaheuristic strategy for covering with wireless devices. *Journal of Universal Computer Science*, 18(14):1906–1932, 2012.

[5] B. Ballinger, N. Benbernou, P. Bose, M. Damian, E. D. Demaine, V. Dujmovic, R. Y. Flatland, F. Hurtado, J. Iacono, A. Lubiw, P. Morin, V. Sacristán Adinolfi, D. L. Souvaine, and R. Uehara. Coverage with k-transmitters in the presence of obstacles. *Journal of Combinatorial Optimization*, 25(2):208–233, 2013.

[6] L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015.

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

[8] T. M. Chan. Comparison-based time-space lower bounds for selection. *ACM Transactions on Algorithms*, 6(2):26, 2010.

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

[10] T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the restore model. In *Proc. 25th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 995–1004, 2014.

[11] A. M. Dean, W. Evans, E. Gethner, J. D. Laison, M. A. Safari, and W. T. Trotter. Bar k-visibility graphs: Bounds on the number of edges, chromatic number, and thickness. In *Proc. 13th Int. Symp. Graph Drawing (GD)*, pages 73–82, 2005.

[12] J. A. Dean, A. Lingas, and J.-R. Sack. Recognizing polygons, or how to spy. *The Visual Computer*, 3(6):344–355, 1988.

[13] D. Eppstein, M. T. Goodrich, and N. Sitchinava. Guard placement for efficient point-in-polygon proofs. In *Proc. 23rd Annu. Sympos. Comput. Geom. (SoCG)*, pages 27–36, 2007.

[14] R. Fabila-Monroy, A. R. Vargas, and J. Urrutia. On modem illumination problems. In *Proc. 13th Encuentros de Geometría Computacional (EGC)*, 2009.

[15] S. Felsner and M. Massow. Parameters of bar k-visibility graphs. *J. Graph. Alg. Appl.*, 12(1):5–27, 2008.

[16] R. Fulek, A. F. Holmsen, and J. Pach. Intersecting convex sets by rays. *Discrete Comput. Geom.*, 42(3):343–358, 2009.

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

[18] S. G. Hartke, J. Vandenbussche, and P. Wenger. Further results on bar k-visibility graphs. *SIAM J. on Discrete Mathematics*, 21(2):523–531, 2007.

[19] B. Joe and R. B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27(4):458–473, 1987.

[20] N. Mouawad and T. C. Shermer. The superman problem. *The Visual Computer*, 10(8):459–473, 1994.

[21] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.

[22] J. O'Rourke. Computational geometry column 52. *ACM SIGACT News*, 43(1):82–85, 2012.