An Optimal Algorithm Computing Edge-to-Edge Visibility in a Simple Polygon

Mikkel Abrahamsen*†

Abstract

Let $\mathcal{P}$ be a simple polygon with n vertices. We present a new $O(n)$-time algorithm to compute the visible part of one edge from another edge of P. The algorithm does not alter the input and only uses $O(1)$ variables and is therefore a constant-workspace algorithm. The algorithm can be used to make a constant-workspace algorithm for computing the weak visibility polygon from an edge in $O(mn)$ time, where m is the number of vertices of the resulting polygon, and a constant-workspace algorithm for computing a minimum link path between two points inside a simple polygon in $O(n^2)$ time.

1 Introduction

Much research has been done on visibility problems in the plane. See the book by Ghosh [8] for an overview of the most important problems and results.

Let $\mathcal{P}$ be a simple polygon with vertices $v_0v_1 \dots v_{n-1}$ in counterclockwise (CCW) order, and let $v_n = v_0$. A point $q \in \mathcal{P}$ is said to be *visible* from v_jv_{j+1} if there exists a point $p \in v_jv_{j+1}$ such that the segment pq is contained in $\mathcal{P}$. In this paper we show how to compute the visible part of an edge v_iv_{i+1} from the edge v_jv_{j+1}. Without loss of generality, we assume that $j = 0$ for the rest of this paper. The algorithm uses $O(n)$ time is therefore optimal. The input is given in read-only memory and only $O(1)$ variables are needed in the workspace, each consisting of $O(\log n)$ bits. Therefore, the algorithm is a *constant-workspace algorithm*.

The problem of computing visibility between two edges was first addressed by Toussaint [13], who gave a linear-time query algorithm deciding if two edges are visible to each other if a triangulation of $\mathcal{P}$ is provided. Later, Avis et al. [4] described an $O(n)$-time algorithm to compute the visible part of one edge from another which does not require a triangulation or other involved data structures, but uses $\Omega(n)$ variables in the workspace. De et al. [7] claimed to present an $O(n)$-time algorithm using constant workspace. However, their algorithm has a fault, as we shall see.

One of the best-known constant-workspace algorithms for a geometric problem is *Jarvis' march* [10] for the computation of the convex hull of n points in the plane in $O(hn)$ time, where h is the number of points on the hull. Recently, Asano et al. [2], Asano et al. [3], and Barba et al. [5] gave constant-workspace algorithms solving many elementary tasks in planar computational geometry. The research presented in this paper is part of a master's thesis [1], which contains more details and space-efficient solutions to some other planar visibility problems.

1.1 Notation and definitions

Given two points a and b in the plane, the line segment with endpoints a and b is written ab. Both endpoints are included in segment ab. If s is a line segment, the line containing s which is inifinite in both directions is written $\overleftrightarrow{s}$. The *half-line* $\overrightarrow{ab}$ is a line infinite in one direction, starting at a and passing through b. The *right half-plane* $\mathrm{RHP}(ab)$ is the closed half plane with boundary $\overleftrightarrow{ab}$ lying to the right of ab. The *left half-plane* $\mathrm{LHP}(ab)$ is just $\mathrm{RHP}(ba)$.

If $\mathcal{P}$ is a simple polygon, the boundary of $\mathcal{P}$ is written $\partial\mathcal{P}$. Let $\mathcal{P}(p, q)$ for two points $p, q \in \partial\mathcal{P}$ be the set of points on $\partial\mathcal{P}$ we meet when traversing $\partial\mathcal{P}$ CCW from p to q, both included. A *chain* of $\mathcal{P}$ is such a set $\mathcal{P}(p, q)$ for some points $p, q \in \partial\mathcal{P}$. We use the general position assumption that no three vertices of $\mathcal{P}$ are collinear.

Consider the edge v_0v_1 of a simple polygon $\mathcal{P}$. A *beam* emanating from v_0v_1 is a segment pq where $p \in v_0v_1$ and pq is contained in $\mathcal{P}$. Thus, a point q is visible from v_0v_1 if and only if there exists a beam pq emanating from v_0v_1. A *right support* of the beam pq is a reflex vertex v of $\mathcal{P}$ such that $v \in pq$ and the edges meeting at v are both contained in $\mathrm{RHP}(pq)$. A *left support* is defined analogously. Since no beam emanates from a point to the left of v_0, we use the convention that v_0 is a left support of any beam v_0q. Likewise, v_1 is a right support of any beam v_1q. A *support* is a right support or a left support.

The edge v_iv_{i+1} is *totally facing* the edge v_jv_{j+1} if both of the points v_j and v_{j+1} are in $\mathrm{LHP}(v_iv_{i+1})$. Notice that v_iv_{i+1} can be totally facing v_jv_{j+1} even though no point on v_jv_{j+1} is visible from v_iv_{i+1}. Edge v_iv_{i+1} is *partially facing* v_jv_{j+1} if excactly one of the points v_j

*Department of Computer Science, University of Copenhagen, mikkel.abrahamsen@gmail.com

†Autodesk ApS, Havnegade 39, DK-1058 Copenhagen K, Denmark

and v_{j+1} is in $\mathrm{LHP}(v_iv_{i+1})$ and *not facing* v_jv_{j+1} if none of the points are in $\mathrm{LHP}(v_iv_{i+1})$. We say that v_iv_{i+1} is *facing* v_jv_{j+1} if v_iv_{i+1} is partially or totally facing v_jv_{j+1}. It follows from the definitions that v_iv_{i+1} is either totally facing, partially facing or not facing v_jv_{j+1}. That gives 9 different combinations of how v_iv_{i+1} is facing v_jv_{j+1} and how v_jv_{j+1} is facing v_iv_{i+1}. However, only 8 of the cases are possible when v_iv_{i+1} and v_jv_{j+1} are edges of a simple polygon, since they cannot both partially face each other. That would imply that they intersect each other properly. All of the remaining 8 cases are possible. See for instance the paper of Avis et al. [4].

2 Visibility Between Two Edges of a Polygon

2.1 Point-to-point and point-to-edge visibility

If the edge v_iv_{i+1} is not facing edge v_0v_1, the only point on v_iv_{i+1} that can be visible from v_0v_1 is one of the endpoints v_i or v_{i+1}. Likewise, if v_0v_1 is not facing v_iv_{i+1}, the only point on v_0v_1 that can possibly see v_iv_{i+1} is one of the endpoints v_0 or v_1 by means of beams contained in $\mathrm{RHP}(v_0v_1)$. In such cases, the problem of computing the visible part of v_iv_{i+1} is reduced to point-to-point and point-to-edge visibility.

Point-to-point visibility is the problem of determining if ab is contained in $\mathcal{P}$ for two given points a and b. That can easily be tested in $O(n)$ time using constant workspace by traversing all edges of $\partial\mathcal{P}$, seeing if $\partial\mathcal{P}$ crosses ab somewhere.

Point-to-edge visibility is the slightly more complicated task of computing the visible part of an edge from a point p. This can also be done using constant workspace and $O(n)$ time by traversing all edges of $\partial\mathcal{P}$ once while keeping track of the vertices shadowing the largest part of the edge in each of the ends [1].

We now turn our attention to the more interesting case of computing the visible part of v_iv_{i+1} from v_0v_1 if the edges are facing each other. We motivate the development of a new algorithm by giving a counterexample to the constant-workspace algorithm of De et al. [7]. The authors are aware of the error [11]. The reader who has not consulted their paper can skip this section.

2.2 Counterexample to the algorithm proposed by De et al. [7]

The textual description and the pseudocode in [7] do not agree. Figure 1 is an example of a polygon where the algorithm computes a wrong result in both cases. After *PASS1*(), the line segment L is still $p_{i+1}p_{j+1}$. After *PASS2*(), L is θp_{j+1}. The text says that *PASS3*() is to check if a vertex on $\mathcal{P}(p_{j+1}, p_i)$ is to the right of L. All the vertices are to the left, so the algorithm returns that the rightmost visible point on p_jp_{j+1} from p_ip_{i+1} is p_{j+1}, which is wrong. The pseudocode gives another definition of *PASS3*(), according to which we also check if a vertex on $\mathcal{P}(p_{i+1}, p_j)$ is to the left of L. Vertex v is, so the algorithm reports that nothing of p_jp_{j+1} is visible. That is clearly also wrong.

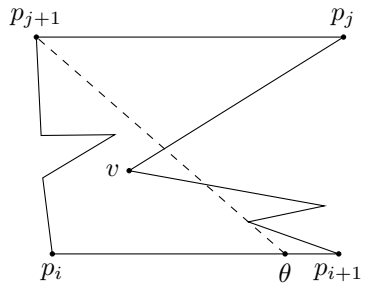

Figure 1: The algorithm from [7] reports the wrong visible part of p_jp_{j+1} from p_ip_{i+1} in this polygon.

2.3 Computing visibility between edges facing each other

Assume for the rest of this section that the edges v_0v_1 and v_iv_{i+1} are facing each other. We want to compute the part of v_iv_{i+1} containing v_{i+1} that is not visible from v_0v_1. The main idea is to consider the edges in the right side chain $\mathcal{P}(v_1, v_i)$ and the left side chain $\mathcal{P}(v_{i+1}, v_0)$ alternately, changing side after each edge. When an edge in one side is found that causes more of v_iv_{i+1} to be invisible from v_0v_1, we retract the search in the other chain to the last interfering edge in that chain. This will be made more precise in the following.

Let $\square = \square v_0v_1v_iv_{i+1}$ be the quadrilateral with vertices $v_0v_1v_iv_{i+1}$ in that order. The possible beams from v_0v_1 to v_iv_{i+1} are all contained in $\square$, so when computing the visible part of v_iv_{i+1}, we are only concerned about the edges of $\mathcal{P}$ that are (partially) in $\square$. A beam pq is a *proper beam* if $pq \subset \mathrm{LHP}(v_0v_1)$ and $pq \subset \mathrm{LHP}(v_iv_{i+1})$. An *improper beam* is a beam that is not proper. Each beam pq where p is an interior point on v_0v_1 and q is an interior point on v_iv_{i+1} is necessarily proper. Therefore, if pq is improper, $p = v_0$, $p = v_1$, $q = v_i$, or $q = v_{i+1}$. The visibility due to improper beams can be computed using point-to-edge visibility, so in this section, we focus on the visibility due to proper beams only. We leave out the proof of the following lemma due to limited space [1].

Lemma 1 *Let* $v_R \in \mathcal{P}(v_1, v_i) \cap \square$ *and* $v_L \in \mathcal{P}(v_{i+1}, v_0) \cap \square$. *Every proper beam* pq *from* v_0v_1 *to* v_iv_{i+1} *satisfies* $p \in LHP(v_Rv_L)$ *and* $q \in RHP(v_Rv_L)$. *In particular, if* $v_0v_1 \cap LHP(v_Rv_L) = \emptyset$ *or* $v_iv_{i+1} \cap RHP(v_Rv_L) = \emptyset$, *then no proper beam from* v_0v_1 *to* v_iv_{i+1} *exists.*

Assume that there are some proper beams from v_0v_1

to v_iv_{i+1}. We say that the beam pq is the *rightmost beam* from v_0v_1 to v_iv_{i+1} if p is as close to v_1 as possible and q is as close to v_{i+1} as possible among all proper beams. Similarly, pq is the *leftmost beam* from v_0v_1 to v_iv_{i+1} if p is as close to v_0 as possible and q is as close to v_i as possible. If v_0v_1 and v_iv_{i+1} are totally facing each other, all beams from v_0v_1 to v_iv_{i+1} are proper, so the visible part of v_iv_{i+1} is the points between the endpoints of the leftmost and rightmost beams. If one of the edges is only partially facing the other, the visible part of v_iv_{i+1} can be computed using the leftmost and rightmost beams in combination with point-to-edge visibility.

If pq is a beam from v_0v_1 to v_iv_{i+1}, a *generalized left support* of pq is v_{i+1} if $q = v_{i+1}$ or a left support of pq otherwise. The following lemma characterizes rightmost beams by means of their supports. The proof is given in [1].

Lemma 2 *Let pq be a proper beam from v_0v_1 to v_iv_{i+1}. The beam pq is a rightmost beam if and only if pq has a right support v_R and a generalized left support v_L and $v_L \in v_Rq$.*

If the edges v_0v_1 and v_iv_{i+1} are totally facing each other and no edge obstructs the visibility between the edges, then the rightmost beam is $pq = v_1v_{i+1}$ and it has supports $v_R = v_1$ and $v_L = v_{i+1}$.

Algorithm 1 returns the indices (R, L) of the supports of the rightmost beam if it exists. The algorithm iteratively computes the correct value of R and L, taking the edges into consideration one by one. Initially, R is set to 1 and L is set to $i + 1$, as if no edges obstructs the visibility between the edges. The points p and q on v_0v_1 and v_iv_{i+1}, respectively, are always defined such that the segment pq contains v_R and v_L. The algorithm alternately traverses $\mathcal{P}(v_1, v_i)$ and $\mathcal{P}(v_{i+1}, v_n)$ one edge at a time using the index variables r and l. The variable *side* is 1 when an edge in $\mathcal{P}(v_1, v_i)$ is traversed and -1 when an edge in $\mathcal{P}(v_{i+1}, v_n)$ is traversed. Each time an edge $v_{r-1}v_r$ or $v_{l-1}v_l$ is found that crosses pq, the value of R or L is updated to r or l, respectively. If the value of R is updated, we reset l to L, since it is possible that there are some edges on $\mathcal{P}(v_L, v_n)$ that did not intersect the old segment pq, but intersect the updated one. Likewise, when L is updated, we reset r to R. Although segment pq is changed when R or L is updated, $\mathcal{P}(v_1, v_R)$ or $\mathcal{P}(v_{i+1}, v_L)$ does not cross pq after the update. That is because pq is rotated clockwise (CW) away from the chains.

All our figures illustrate the case where v_0v_1 and v_iv_{i+1} are totally facing each other, but that assumption is not used in any of the proofs. If v_iv_{i+1} is partially facing v_0v_1 such that $v_0 \in \mathrm{LHP}(v_iv_{i+1})$, then v_i might be the right support of the rightmost beam from v_0v_1 to v_iv_{i+1}. Likewise, if v_0v_1 is partially facing v_iv_{i+1} such that $v_i \in \mathrm{LHP}(v_0v_1)$, v_0 can be the left support of the rightmost beam.

Algorithm 1: `FindRightmostBeam`(i)

Input: A polygon $\mathcal{P}$ defined by its vertices $v_0, v_1, \ldots, v_{n-1}$ in CCW order and an index i such that v_0v_1 and v_iv_{i+1} are facing each other.

Output: If no proper beam from v_0v_1 to v_iv_{i+1} exists, `NULL` is returned. Otherwise, a pair of indices (R, L) is returned such that the rightmost beam from v_0v_1 to v_iv_{i+1} has right support v_R and generalized left support v_L.

```
1  R ← 1,   L ← i + 1
2  r ← R,   l ← L
3  p ← v_1,   q ← v_{i+1}
4  side ← 1   (* 1 is right side, −1 is left side *)
5  while r < i or l < n
6      if side = 1
7          if r < i
8              r ← r + 1
9              if v_{r−1}v_r enters LHP(pq) ∩ □
10                 if v_{r−1}v_r intersects v_Lq
11                     return NULL
12             R ← r,   l ← L
13     else   (* side = −1 *)
14         if l < n
15             l ← l + 1
16             if v_{l−1}v_l enters RHP(pq) ∩ □
17                 if v_{l−1}v_l intersects v_Rp
18                     return NULL
19             L ← l,   r ← R
20     Let p be the intersection point between v_Lv_R→ and v_0v_1
21     Let q be the intersection point between v_Rv_L→ and v_iv_{i+1}
22     if p or q does not exist
23         return NULL
24     side ← −side
25 if pq ⊂ LHP(v_0v_1) ∩ LHP(v_iv_{i+1})
26     return (R, L)
27 else
28     return NULL
```

Lemma 3 *Assume that Algorithm 1 terminates after k iterations of the loop at line 5. Let R_j, L_j, p_j, and q_j be the values of R, L, p, and q, respectively, in the beginning of iteration j, $j = 1, 2, \ldots, k + 1$, where the values when the algorithm terminates have index $k + 1$. Then $R_j = R_{j+1}$ or $L_j = L_{j+1}$ for*

$j = 1, \ldots, k$. *$p_1, p_2, \ldots, p_{k+1}$ is a sequence of points moving monotonically along v_0v_1 from v_1 towards v_0. Likewise, $q_1, q_2, \ldots, q_{k+1}$ is a sequence of points moving monotonically along v_iv_{i+1} from v_{i+1} towards v_i. Let a_j be the CW angle from $\overleftrightarrow{v_{R_{j-1}}v_{L_{j-1}}}$ to $\overleftrightarrow{v_{R_j}v_{L_j}}$. Then $\sum_{j=2}^{k+1} a_j < 180°$. In particular, $a_j < 180°$ for each $2 = 1, \ldots, k+1$.*

Proof. See Figure 2. It is clear that at most one of R and L changes in iteration j, since the lines 12 and 19 cannot both be executed. Therefore, $R_j = R_{j+1}$ or $L_j = L_{j+1}$. If R is redefined in iteration j, then $\overleftrightarrow{v_Rv_L}$ is rotating around v_L and the new value of R, namely R_{j+1}, satisfies $v_{R_{j+1}} \in \mathrm{LHP}(v_{R_j}v_{L_j})$. Therefore, p_{j+1} is on the segment v_0p_j and q_{j+1} is on the segment q_jv_i. The same is true if L is updated. Hence, $p_1, \ldots, p_{k+1}$ is monotinically moving along v_0v_1 from v_1 towards v_0 and $q_1, \ldots, q_{k+1}$ is monotonically moving along v_iv_{i+1} from v_{i+1} towards v_i. Because of the monotonicity, the angles are additive, so that the CW angle from $\overleftrightarrow{v_{R_1}v_{L_1}}$ to $\overleftrightarrow{v_{R_{k+1}}v_{L_{k+1}}}$ is $\sum_{j=2}^{k+1} a_j$. If v_0v_1 is totally facing v_iv_{i+1}, every q_j is contained in $\mathrm{LHP}(v_0v_1)$. Otherwise v_iv_{i+1} is totally facing v_0v_1 so that every p_j is contained in $\mathrm{LHP}(v_iv_{i+1})$. In either case, $\sum_{j=2}^{k+1} a_j$ is bounded by 180°. That bound cannot be reached, since it would require that v_0v_1 or v_iv_{i+1} was infinitely long in both directions. □

Lemma 4 *Algorithm 1 correctly computes the rightmost beam from v_0v_1 to v_iv_{i+1} as specified. The algorithm is a constant-workspace algorithm.*

Proof. First, consider the cases where the algorithm returns `NULL`. In line 11, we have found an intersection point x between $\mathcal{P}(v_R, v_i)$ and v_Lq. That means that $\mathcal{P}(v_1, v_i)$ intersects pq properly at x, since no three vertices are collinear. Lemma 1 establishes that the only possible proper beams from v_0v_1 to v_iv_{i+1} are of the form $p'q'$, where $p' \in v_0p$ and $q' \in qv_i$. At the same time, if we use Lemma 1 with v_0v_1 and v_iv_{i+1} interchanged by each other and using x as 'v_L' and v_L as 'v_R', we get that $p'q'$ satisfies $p' \in pv_1$ and $q' \in v_{i+1}q$. Therefore, $p' = p$ and $q' = q$, but pq is not a beam. Hence, there are no proper beams from v_0v_1 to v_iv_{i+1}. The case in line 18 is analogous.

Due to Lemma 3, we know that p is moving monotonically from v_1 towards v_0 and q is moving monotonically from v_{i+1} towards v_i. The case of line 23 happens if p has moved outside v_0v_1, so that $v_0v_1 \cap \mathrm{LHP}(v_Rv_L) = \emptyset$, or q has moved outside v_iv_{i+1}, so that $v_iv_{i+1} \cap \mathrm{RHP}(v_Rv_L) = \emptyset$. In each of these cases, it follows from Lemma 1 that there are no proper beams from v_0v_1 to v_iv_{i+1}.

The test at line 25 is to ensure that pq is a proper beam, which is not always the case if v_0v_1 is only partially facing v_iv_{i+1}.

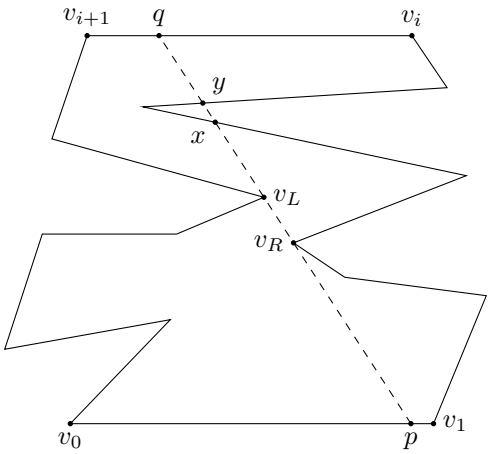

Figure 3: Case 2 in the proof of Theorem 4.

Now, assume that the algorithm returns (R, L), but that pq is not a beam since some edge obstructs the visibility from p to q. Assume that $\mathcal{P}(v_1, v_i)$ intersects pq properly, and let x be the intersection point closest to p. $\mathcal{P}(v_1, v_i)$ enters $\mathrm{LHP}(pq) \cap \square$ at x. Let y be the first point where $\mathcal{P}(x, v_i)$ crosses pq from left to right. Then $y \in xq$. We have two cases: $x \in \mathcal{P}(v_1, v_R)$ (case 1) and $x \in \mathcal{P}(v_R, v_i)$ (case 2). Assume that we are in case 2, see Figure 3. Assume that the final value of R is defined in a later iteration of the loop at line 5 than the final value of L. After R is defined in line 12, every edge $v_{r-1}v_r$ in $\mathcal{P}(v_R, v_i)$ is traversed and it is checked in line 10 if some edge intersects pq. In particular the edges in $\mathcal{P}(x, y)$ are traversed, in which case the algorithm either returns `NULL` or updates R, which is a contradiction. If R is defined in an earlier iteration than L, then r is reset to R in line 19 when L is defined, and it is checked if some edge in $\mathcal{P}(v_R, v_i)$ intersects pq, so that cannot happen either.

Assume that we are in case 1, i.e. $x \in \mathcal{P}(v_1, v_R)$. Consider the first iteration, say iteration j, at the beginning of which $\mathcal{P}(v_1, v_R)$ intersects pq properly, and let x' be the intersection point closest to p. Let y' be the first point where $\mathcal{P}(x', v_i)$ crosses pq from left to right. Then $y' \in x'q$ (x' and y' might not be the same as x and y, since R and L can change before the algorithm terminates). We must have $v_R \in \mathcal{P}(y', v_i)$. There are three possible cases to consider: $v_R \in px'$ (case 1.1), $v_R \in x'y'$ (case 1.2), and $v_R \in y'q$ (case 1.3).

Assume case 1.3. Let R_k, L_k, p_k, and q_k be defined as in Lemma 3 for each iteration k. Either R or L is redefined in iteration $j-1$ due to the minimality of j. Therefore, $R_{j-1} \neq R_j$ or $L_{j-1} \neq L_j$ (case 1.3.1 and 1.3.2, respectively). First, assume $R_{j-1} \neq R_j$ but $L_{j-1} = L_j$. Again, there are three cases to consider: $v_{R_{j-1}} \in \mathcal{P}(v_1, x')$ (case 1.3.1.1), $v_{R_{j-1}} \in \mathcal{P}(x', y')$ (case 1.3.1.2), and $v_{R_{j-1}} \in \mathcal{P}(y', v_{R_j})$ (case 1.3.1.3). As-

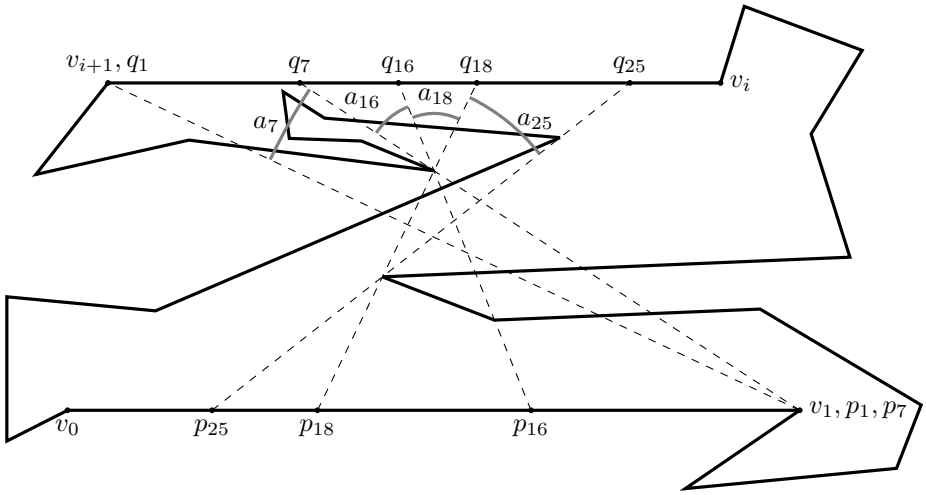

Figure 2: Illustration for Lemma 3. The points p_j, q_j are shown with the number j of the first iteration where they occur. The segments p_jq_j are drawn dashed. The angles $a_j > 0$ are indicated with grey arcs.

sume case 1.3.1.3, see Figure 4(a). According to Lemma 3, the CW angle between $\overleftrightarrow{v_{R_{j-1}}v_{L_{j-1}}}$ and $\overleftrightarrow{v_{R_j}v_{L_j}}$ is less than 180°. Therefore, a subset of $\mathcal{P}(x', y')$ would also be contained in $\mathrm{LHP}(v_{R_{j-1}}v_{L_{j-1}}) \cap \square$. That implies that $\mathcal{P}(v_1, v_{R_{j-1}})$ intersects $p_{j-1}q_{j-1}$, a contradiction because of the choice of j. $v_{R_{j-1}}$ cannot be in $\mathcal{P}(x', y')$ (case 1.3.1.2), because then v_{R_j} would be in $\mathrm{RHP}(v_{R_{j-1}}v_{L_{j-1}})$, so R would not have been redefined to R_j in iteration $j-1$. Finally, if $v_{R_{j-1}}$ was a vertex in $\mathcal{P}(v_1, x')$ (case 1.3.1.1), $\mathcal{P}(x', y')$ would be contained in $\mathrm{LHP}(v_{R_{j-1}}v_{L_{j-1}}) \cap \square$, and therefore v_R would be redefined to a vertex in $\mathcal{P}(x'y')$ when the edges of that chain was traversed. Hence, v_R would not be redefined to v_{R_j} in iteration $j-1$, which is a contradiction.

Now, assume $L_{j-1} \neq L_j$ (case 1.3.2), see Figure 4(b). The CW angle between $\overleftrightarrow{v_{R_{j-1}}v_{L_{j-1}}}$ and $\overleftrightarrow{v_{R_j}v_{L_j}}$ is less than 180°. Therefore, a part of $\mathcal{P}(x', y')$ is also in $\mathrm{LHP}(v_{R_{j-1}}v_{L_{j-1}})$. That implies that $\mathcal{P}(v_1, v_{R_{j-1}})$ intersects $p_{j-1}q_{j-1}$, which contradicts the choice of j.

The case where $v_R \in x'y'$ (case 1.2) can be eliminated in a similar way. Consider case 1.1, i.e. $v_R \in px'$. The chain $\mathcal{P}(p, x')$ and the segment $x'p$ forms a simple, closed curve, because x' is the intersection point between $\mathcal{P}(v_1, v_i)$ and pq closest to p. The curve can, for instance, be seen in Figure 4(a). Consider the region of $\mathcal{P}$ enclosed by the curve. In order to get to v_R, $\mathcal{P}(y', v_i)$ has to cross $x'p$ to get into the region. That contradicts that x' was the intersection point closest to p.

If we assume that $\mathcal{P}(v_{i+1}, v_n)$ intersects pq, we get a contradiction in an analogous way.

The conclusion is that if (R, L) is returned, v_R and v_L defines a proper beam pq with right support v_R and generalized left support v_L in that order. Therefore, pq must be the rightmost beam from v_0v_1 to v_iv_{i+1} according to Lemma 2.

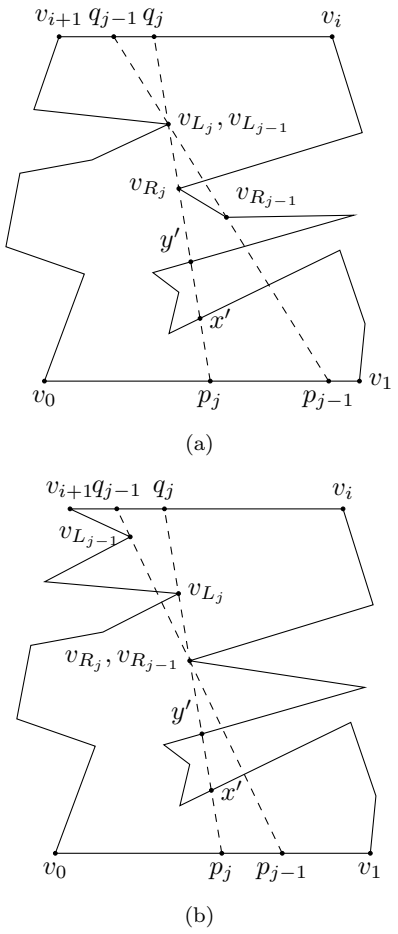

Figure 4: Cases in the proof of Theorem 4. (a) Case 1.3.1.3. (b) Case 1.3.2.

Observe that the vertices of $\mathcal{P}$ are not altered. Hence the input is read only. In addition to that, we only use the variables R, L, r, l, p, q, and *side*. The computations of intersections and containment at lines 9, 10, 16, 17, 20, 21, and 25 are easily implemented using constant workspace. □

Even though we reset l to L in line 12 or r to R in line 19, the running time is linear since the other variable is not reset, so half of the traversed edges are never traversed again, as the following lemma explains.

Lemma 5 *There are at most $2n-6$ iterations of the loop at line 5 of Algorithm 1.*

Proof. Let $N(n)$ be the maximal number of edge visits for a polygon with n vertices. Consider the first time line 12 or 19 is executed. Assume it is line 12. There have been made $2(r-1)-1 < 2(r-1)$ iterations, because $\mathcal{P}(v_1, v_i)$ is traversed every second time, beginning with the first. The $r-1$ edges in $\mathcal{P}(v_1, v_r)$ are never traversed again. Therefore, N satisifies the recurrence $N(n) \leq 2k + N(n-k)$, where $k = r-1$. A similar bound holds for some $k \geq 1$ if line 19 is executed first. We know that $N(4) = 2$, so induction yields that $N(n) \leq 2n-6$ is an upper bound. □

It is clear that an algorithm to compute a leftmost beam from v_0v_1 to v_iv_{i+1} can be constructed symmetrically. That gives us the following theorem:

Theorem 6 *The visible part of an edge v_iv_{i+1} from v_0v_1 in a simple polygon can be computed in $O(n)$ time using constant workspace.*

3 Weak Visibility Polygons and Minimum Link Paths

The weak visibility polygon of the polygon $\mathcal{P}$ from the edge v_0v_1 consists of all the points in $\mathcal{P}$ visible from v_0v_1. Guibas et al. [9] presented an $O(n)$-time algorithm to compute the weak visibility polygon if a triangulation of $\mathcal{P}$ is provided, where n is the number of vertices of $\mathcal{P}$. Later, Chazelle [6] described an $O(n)$-time deterministic triangulation algorithm, implying that the weak visibility polygon can be computed in $O(n)$ time using $O(n)$ space. Using Algorithm 1, one can make a $O(mn)$-time algorithm using constant workspace, where m is the number of edges of the weak visibility polygon [1]. It is well-known that $m = O(n)$.

A *minimum link path* between two points s and t in a simple polygon is a polygonal path from s to t which is contained in $\mathcal{P}$ and which consists of as few segments as possible. Suri [12] showed how to compute a minimum link path using $O(n)$ time if a triangulation of $\mathcal{P}$ is provided. Using the algorithm to compute the weak visibility polygon, it is possible to devise an $O(n^2)$-time algorithm to compute a minimum link path using constant workspace. The algorithm does not use a triangulation of $\mathcal{P}$. The details are given in [1].

Acknowledgements

We would like to thank Jyrki Katajainen, Ashwini Joshi, Kristian Mortensen, and the anonymous reviewers for suggesting many improvements to the present paper.

References

[1] M. Abrahamsen. Constant-workspace algorithms for visibility problems in the plane. Master's thesis. University of Copenhagen, Department of Computer Science, 2013. Available at `http://www.diku.dk/forskning/performance-engineering/Mikkel/thesis.pdf`.

[2] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, to appear.

[3] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *Journal of Computational Geometry*, 2(1):46–68, 2011.

[4] D. Avis, T. Gum, and G. Toussaint. Visibility between two edges of a simple polygon. *The Visual Computer*, 2(6):342–357, 1986.

[5] L. Barba, M. Korman, S. Langerman, and R. Silveira. Computing the visibility polygon using few variables. In *Proceedings of the 22nd International Symposium on Algorithms and Computation*, volume 7014 of *Lecture Notes in Computer Science*, pages 70–79. Springer, 2011.

[6] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(1):485–524, 1991.

[7] M. De, A. Maheshwari, and S. Nandy. Space-efficient algorithms for visibility problems in simple polygon. E-print arXiv:1204.2634, 2012.

[8] S. Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

[9] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1):209–233, 1987.

[10] R. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[11] A. Maheshwari. Private communication.

[12] S. Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics, and Image Processing*, 35(1):99–110, 1986.

[13] G. Toussaint. Shortest path solves edge-to-edge visibility in a polygon. *Pattern Recognition Letters*, 4(3):165–170, 1986.