Computing the Visibility Polygons of the Endpoints of a Set of Line Segments in Output Sensitive Time

Mark Keil
Department of Computational Science
University of Saskatchewan
Saskatoon, Canada
keil@cs.usask.ca
&
Stephen Wismath
Department of Mathematical Sciences
University of Lethbridge
Lethbridge, Alberta, Canada
wismath@hg.uleth.ca

Abstract

Given a set S of n non-intersecting line segments in the plane, we present an algorithm that computes the $2n$ visibility polygons of the endpoints of S, in output sensitive time. The algorithm relies on the ordered (endpoint) visibility graph information to traverse the endpoints in a spiral-like manner using a combination of Jarvis' March and depth-first search.

One extension of this result is an efficient (and practical) algorithm for computing the full visibility graph of S, in which vertices correspond to segments and a pair of vertices are joined by an edge if the corresponding line segments are somewhere visible.

1 Introduction

Problems involving the *visibility* of objects in a given domain have arisen in several areas of computer science, such as, VLSI design, graphics and motion planning. Visibility problems involving line segments in the plane are fundamental and many problems involving more general objects can be reduced (or approximated) by this case. Frequently, it is the underlying structure of the visibilities that is critical and a graph can be created that condenses this structural information. In this paper, various problems involving a set S of n non-intersecting line segments in general position, in the plane, are considered.

The **endpoint visibility graph** of S, denoted here as $G_e(S)$, is defined as a graph with $2n$ vertices corresponding to the *endpoints* of the line segments of S, and with an edge set representing the endpoint visibilities (i.e. two endpoints q and r are visible if the line segment joining them intersects no other line segment of S). This graph arises naturally in motion planning problems and has been extensively studied [4], [7], [1]. Let E represent the number of edges in $G_e(S)$. It is clear that E is $\mathcal{O}(n^2)$. The problem of computing $G_e(S)$ can be solved in output sensitive time, in particular, Ghosh and Mount [4], have presented an $\mathcal{O}(E + n\log n)$ time algorithm.

The visibility polygon from a given point q is denoted as VPoly(q) and is defined as the (star-shaped) polygon consisting of the portions of the line segments of S visible from q.[1] The main contribution of this paper is an optimal solution to the problem of computing the set of visibility polygons from the $2n$ endpoints of S. We present an output sensitive algorithm that solves the problem in $\mathcal{O}(E)$ time after the endpoint visibility graph, $G_e(S)$ is computed.

Define two line segments u and v as **visible** if there exists a point on u and a point on v such that the line segment joining them, intersects no other line segment of S. The **full visibility graph** of S, denoted as $G_f(S)$, is

[1] VPoly(q) may be unbounded, in which case we assume there are "points at infinity" which serve to close the polygon with artificial sides.

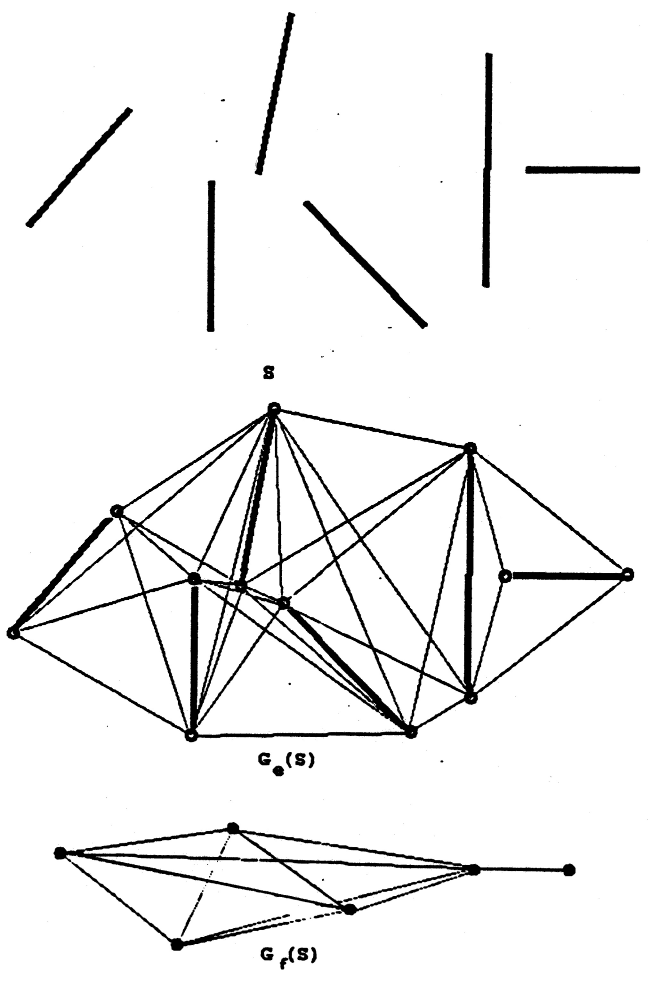

Figure 1: The visibility graphs of S

a graph whose n vertices correspond to the line segments of S and whose edges represent the visibility relation. See figure 1 for examples of the two visibility graphs. As noted by Ghosh and Mount[4], E, (i.e. the number of edges in $G_e(S)$), is asymptotically equal to the number of edges in $G_f(S)$ and furthermore, the graph $G_f(S)$ can also be computed in $\mathcal{O}(E + n \log n)$ time in the general case. Our second result is an efficient solution to the problem of computing $G_f(S)$ which outperforms the algorithm of Ghosh and Mount for special cases (for example, if the line segments of S form a simple polygon). Our algorithm relies on the information provided by the solution to the problem of computing the visibility polygons of the endpoints of S.

2 Visibility Polygons

In this section, we present an algorithm for computing the $2n$ visibility polygons of the endpoints of a set S of n non-intersecting line segments. As a preprocessing step, we assume that the ordered endpoint visibility graph $G_e(S)$, has been computed. The endpoint visibility graph information provides one half of the information needed to compute the visibility polygons. For a particular endpoint q, each neighbour r of q in $G_e(S)$ is in VPoly(q) but the following stabbing query must also be answered: determine the segment of S first encountered by the ray extending from r in the direction $\overline{qr}$. See Figure 2.

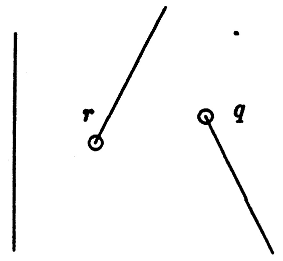

Figure 2: Stabbing Query

If the visibility polygon of r were known, then this stabbing query could be answered in $\mathcal{O}(\log d_q)$ time via binary search. Note that the visibility between q and r also generates a stabbing query in the opposite direction.

Given $G_e(S)$, the visibility polygons can be determined by solving (exactly) $2E$ "stabbing queries". The main contribution of this section is to order these queries so that the $2E$ answers are computed in $\mathcal{O}(E)$ time. The approach combines a modified Jarvis' March with depth-first search, to process the endpoints of S in a spiral-like manner.

The main algorithm to compute the $2n$ visibility polygons is as follows:

- compute $G_e(S)$.
- mark all vertices as unvisited.
- $v :=$ leftmost endpoint of S.
- $e :=$ the convex hull edge of S clockwise from v.
- **Spiral** (v, e).

The procedure *Spiral* is a modified depth first search routine which computes partial visibility polygons as it visits endpoints and then completes the visibility polygons as it returns from the recursion. Thus, there are two stages – a winding, and an unwinding stage. During the winding stage, a previously unvisited endpoint is partially processed by computing the "outer" portion of its visibility polygon and then this information is used to answer the stabbing queries of some of its neighbours, namely the visible endpoints on the "inside" of the spiral. During the unwinding step, the remainder (i.e. the "inside") of each endpoint's visibility polygon is computed and the related stabbing queries (i.e. to the "outside") are answered.

Let VPoly(v, a, b) denote the portion of the visibility polygon counterclockwise about vertex v from edge a to edge b. (If $a = b$ then the entire visibility polygon is meant). In order to compute VPoly(v, a, b) efficiently, it is critical that all the stabbing queries from v to r have been answered, for all r visible to v counterclockwise between a and b. The set of all stabbing queries *to* v from all vertices w that lie clockwise between a and b will be denoted as Stabs(v, a, b). The answer to each stabbing query will be stored on the associated edge in $G_e(S)$ to avoid "table lookup". Both VPoly(v, a, b) and Stabs(v, a, b) can be computed by a simple rotational sweep about v in $O(d_v)$ time, where d_v represents the degree of vertex v in $G_e(S)$.

The spiralling process is complicated slightly by the presence of "kinks", i.e. endpoints where the spiral actually forms a "right turn". In this case, the outside of the spiral is strictly less than 180 degrees and an insufficient number of stabbing answers would be generated if the (partial) visibility polygon were computed. The algorithm ignores such right turn endpoints by not marking them as "visited" (and hence visits them again). It will be shown that right turn endpoints do subsequently become left turn endpoints (or the recursion terminates at them) at which time they become marked and processed appropriately. The consequence of the existence of these right turn endpoints is that the spiral wraps around on itself at such endpoints - ***however it does not actually cross itself.*** See figures 3 and 5 for examples of right turn vertices.

We now present a more formal pseudocode description of the spiral procedure.

```
Procedure Spiral ($v$ : vertex; $e$ : edge)
known [$v$] := $e$
$e'$ := $e$
For each vertex $v'$ adjacent to $v$ (counterclockwise from $e$) do
    • $e'$ := $(v, v')$
    • if mark[$v'$] = unvisited then
        – if $e, e'$ forms a left turn then
            * mark[$v$] := visited;
            * compute VPoly($v$, known[$v$], $e'$)
            * report Stabs($v$, known[$v$], $e'$)
            * known[$v$] := $e'$
            * Spiral ($v', e'$)
          else {right turn – do not mark}
            * Spiral ($v', e'$)
            * return
{ all vertices adjacent to $v$ are marked – unwind}
if $e = e'$ then mark[$v$] := visited        { bottom out case}
compute VPoly($v$, known[$v$], $e$)        {VPoly($v$) is now complete}
report Stabs($v$, known[$v$], $e$)
end{Spiral}
```

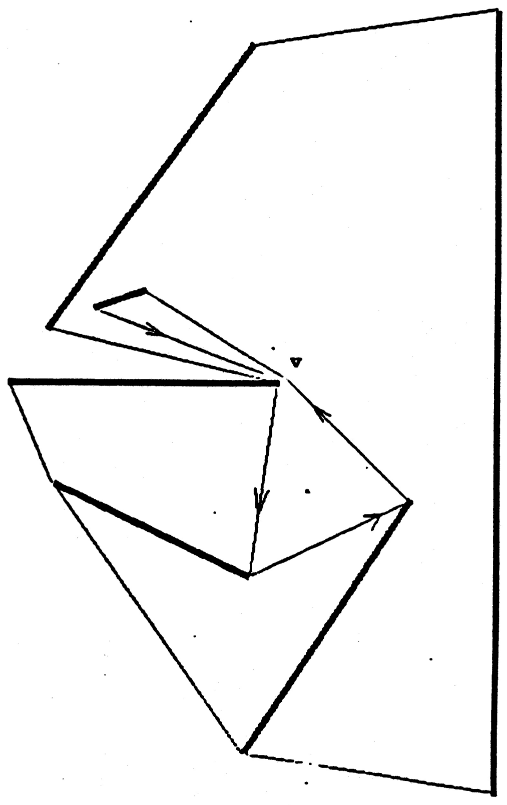

Figure 3: Right turns in spirals

Note that the array named ***known*** is used to keep track of the portion of the visibility polygon of endpoint v that has been computed thus far. Using known avoids recomputing part of Vpoly(v) in the case of v being a left turn vertex that when backed into by the recursion during unwinding, is discovered to have unvisited neighbours. For

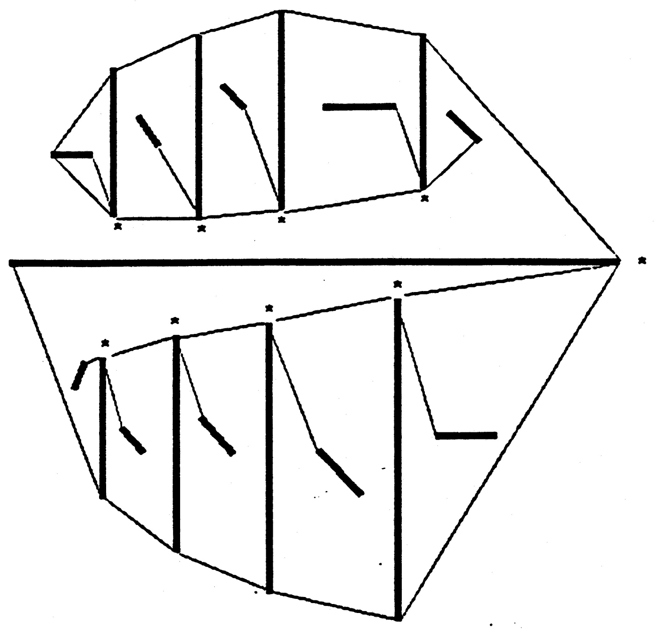

Figure 4: Backtracking through left turn endpoints

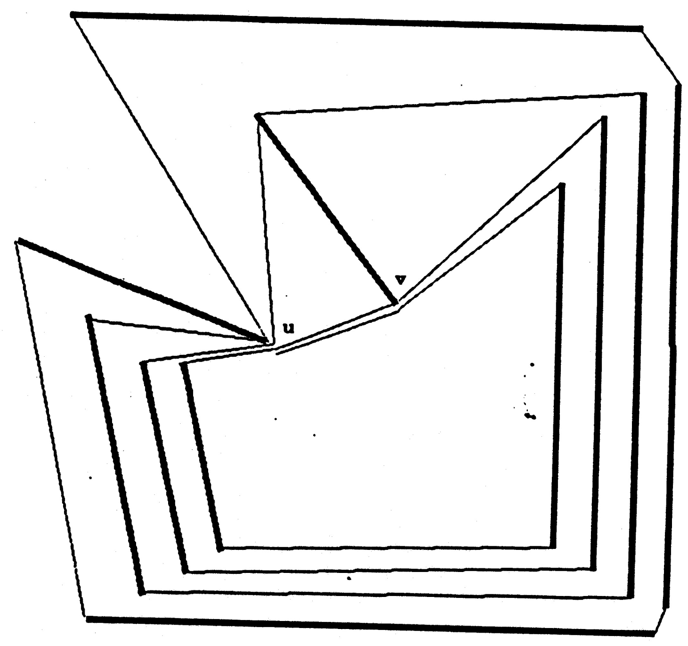

Figure 5: Two adjacent right turns in spirals

example, the endpoints marked with "*" in figure 4 are all left turn endpoints which have the above property.

2.1 Correctness

Since $G_e(S)$ is a connected graph, every endpoint of S is ultimately marked as "visited", because the algorithm is essentially a modified depth-first search.

The invariant exploited by the algorithm is that when a left turn endpoint v, is encountered and processed, all stabbing answers are available (each in constant time) from the (visible) endpoints on the outside of the spiral and thus VPoly(v, e, e') is computable in $O(d_v)$ time.

The correctness of the algorithm can be established by showing that the "spiral" does not actually pierce itself, and hence the notion of "outside" is well-defined. It is the existence of right turn endpoints on the spiral that complicates the spiralling process. The spiral-like path generated by the algorithm is not necessarily simple, it may self-intersect at a right turn endpoint, at an edge (between two right turn endpoints), or at a sequence of edges (between adjacent right turn endpoints). In figure 5 for example, the endpoints u and v are adjacent right turn endpoints on the spiral.

Lemma:

The spiral-like path generated by the algorithm does not cross itself.

Proof Sketch: (By contradiction) There are two cases to consider:

Assume the spiral crosses itself *at an endpoint* v, (refer to figure 6). Note that v must be a right turn endpoint in this case.

When the spiral first enters v along an edge e, the algorithm determines the first unvisited visible vertex from v counterclockwise from e - label this outgoing edge e'. If v is a right turn endpoint, then the spiral must reenter v at some later stage along an edge e'', however since there are no unvisited vertices counterclockwise between e and e', the outgoing edge associated with e'' can not lie in that region and hence the spiral can not actually cross itself at v. Note that this outgoing edge may create another right turn at v, in which case v will be entered yet again, however the argument is also still applicable.

Assume the spiral crosses itself at an edge as in figure 7. Let the endpoints creating the first such edge crossing on the spiral be a, b, c and d. When the endpoint a was processed, endpoint d was not marked as "visited" and hence there must exist a line segment u blocking the visibility between a and d (since b was chosen as the next endpoint on the spiral). Segment u can not block the visibility between a and b nor the visibility between c and d, and thus there is an endpoint v inside the triangle defined by adb, and visible to a. Since this endpoint v was not visited after a, it must have been previously visited:

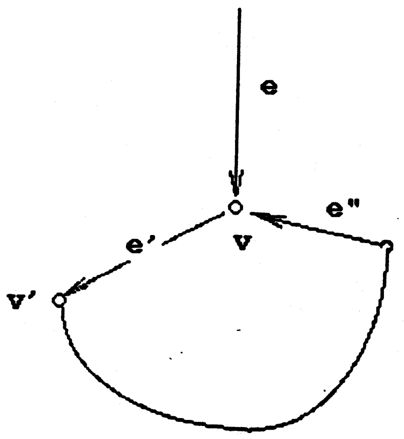

Figure 6: Endpoint crossing

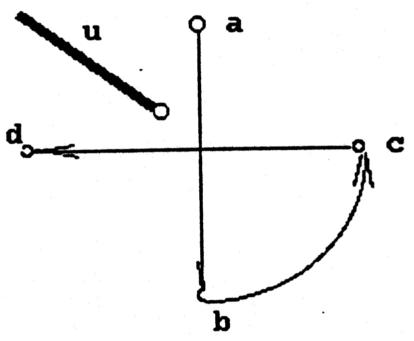

Figure 7: Edge crossing

- either v was a right turn endpoint (and hence unmarked), which contradicts the choice of b as the next visited endpoint on the spiral after a, or
- v was a left turn vertex, in which case there exists another edge crossing, earlier in the spiral, contradicting the assumption that (a, b) (c, d) was the first such crossing.

2.2 Analysis

After the preprocessing step of computing the endpoint visibility graph $G_e(S)$, the initializations in the main program can be performed in O(E) time.

Before analysing the procedure spiral, there are two modifications necessary. It is possible that the spiral enters a right turn endpoint more than once along a given edge (see figure 5 for example). If the spiral exitted such a right turn endpoint by a different edge on each visit, then it could be argued that each right turn endpoint v, is not entered more than d_v times. However, it is possible that a chain of consecutive right turn endpoints is created and the spiral may thus enter and exit some right turn endpoints entirely along previously used edges. To avoid this situation, we assume a double-ended queue is created whenever more than two consecutive right turn endpoints appear on the spiral. By maintaining this queue, it is possible to avoid passing through a right turn endpoint more than once without using a new edge.

The second modification to the procedure also involves right turn endpoints. since a right turn endpoint v, may be entered many times before it receives left turn status, it is important that the associated edge be found quickly - i.e. we can not afford to perform the same counterclockwise scan of visible *visited* vertices each time the endpoint is entered. The solution to this problem is to maintain a copy of the ordered visibility information about v and to actually delete the edges to visited neighbours as they are discovered. Thus, when a right turn endpoint is subsequently entered, the next visible unvisited endpoint is discovered without considering previously rejected edges.

Note that neither of these modifications affects the correctness of the algorithm; they are used merely to improve the efficiency. Given the above two modifications to the Spiral procedure, we now show that the running time is O(E).

Consider an endpoint v that is discovered to be a left turn on the spiral during the winding stage. Both VPoly(v, e, e') and Stabs(v, e, e') are easily computed with a single rotational sweep in O(d_v) time. Similarly, when v is encountered during the unwinding stage, the rest of the visibility polygon and associated stabbing queries can be computed in O(d_v) time. There is one complication to this argument. If during the unwinding stage, v is discovered to have further unvisited neighbours, a new winding stage is initiated (for example at the vertices marked with "*" in figure 4). In this situation, only a small slice of VPoly(v) is computed, namely from known[v] to e'. Thus, although v may be reentered several times, the *total* work to compute the visibility polygon of v is O(d_v).

Given the previous modifications to the Spiral procedure, consider the *total* time spent over the lifetime of the procedure at a given right turn endpoint v. The number of times v is encountered as a right turn endpoint times the amount of time spent determining the next endpoint on the spiral is O(d_v) in total, since each entry or exit to v must involve a new edge.

Summing over all endpoints yields the claimed O(E)

running time.

3 The Full Visibility Graph

Our original motivation for considering the visibility polygon problem for endpoints was in relation to the computation of the full visibility graph of a set of line segments. Recall that in the *full* visibility graph of S, $G_f(S)$, vertices correspond to the line segments of S, and a pair of vertices are adjacent iff the corresponding line *segments* are visible.

Although the algorithm of Ghosh and Mount [4], can be modified to compute the full visibility graph in optimal *output sensitive* time $\mathcal{O}(E + n \log n)$, the algorithm is not practical to implement as it requires sophisticated data structures (finger trees) to implement.

In [8], an $\mathcal{O}(n^2)$ time and space algorithm was presented, that relies on a result due to Asano, Asano, Guibas, Hershberger and Imai, [1] as a preprocessing step to compute the (ordered) visibility polygon of each endpoint of the line segments in S. However the full visibility algorithm does not require the full power of the Asano et al algorithm as only *endpoint* queries need be answered, not general queries.

In this section, we present a (practical) output sensitive algorithm for computing the full visibility graph *given the endpoint visibility graph*. By divorcing the two problems *any* endpoint visibility routine may be used as a preprocessing step.

This "modular" strategy enables one to take advantage of algorithms for the endpoint visibility graph problem that are, for example, dynamic [6], practical [5], or fast in a restricted domain [3].

We adapt the algorithm of Wismath [8] to compute the full visibility graph, $G_f(S)$, in $\mathcal{O}(E)$ time, as follows:

For each endpoint q of each segment u do

- compute the visibility polygon VPoly(q) as in section 2.
- report as visible u and each segment in VPoly(q).
- report the visibilities *around* q (i.e. the segments of S that are 180 degrees apart about q in VPoly(q)) with a rotational sweep technique.

4 Conclusion

The main contribution of this paper is an algorithm that computes the $2n$ visibility polygons of the endpoints of a set of line segments. The algorithm runs in $\mathcal{O}(E)$ time, after the endpoint visibility graph has been computed (in the general case this preprocessing step can be performed in $\mathcal{O}(E + n \log n)$ time [4] but can be contained to $\mathcal{O}(E)$ in special cases, for example, if the line segments form a simple polygon [3]).

One consequence of this result is that the full visibility graph of S can also be computed in $\mathcal{O}(E)$ time, given the endpoint visibility graph.

Acknowledgement

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, Visibility of Disjoint Polygons, *Algorithmica*, **1**, No. 1, (1986), 49-64.

[2] B. Chazelle, Triangulating a simple polygon in linear time, *Discrete and Computational Geometry* Vol. 6, No. 5 (1991), 485-524.

[3] J. Hershberger, Finding the visibility graph of a simple polygon in time proportional to its size, *Proceedings of the 3rd Symposium on Computational Geometry*, Waterloo, 1987, 11-20.

[4] S. Ghosh, D. Mount, An Output Sensitive Algorithm for Computing Visibility Graphs, *SIAM J. of Computing*, Vol. 20, No. 5, (1991), 888-910.

[5] M. Overmars, E. Welzl, New methods for computing visibility graphs, *Proceedings of the 4th Symposium on Computational Geometry*, Urbana-Champaign, 1988, 164-171.

[6] G. Vegter, Dynamically maintaining the visibility graph, Workshop on Algorithms and Data Structures 1991, Ottawa, *Springer Verlag Lecture Notes in Computer Science, #519*, 425-436.

[7] E. Welzl, Constructing the Visibility Graph for n-Line Segments in O(n^2) Time, *Information Processing Letters* 20 (1985), 167-171.

[8] S. Wismath, Computing the full visibility graph of a set of line segments, to appear in *Information Processing Letters*.