International Journal of Computational Geometry & Applications

VISIBILITY STABS AND DEPTH-FIRST SPIRALLING ON LINE SEGMENTS IN OUTPUT SENSITIVE TIME

MARK KEIL
Dept. of Computer Science, University of Saskatchewan, Saskatoon, Sask., Canada. *

and

DAVID M. MOUNT
Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, USA. †

and

S. K. WISMATH
Department of Mathematics and Computer Science, University of Lethbridge, 4401 University Drive, Lethbridge, AB, Canada, T1K 3M4; E-mail: `wismath@cs.uleth.ca`. ‡

Received 1 September 1995
Revised 1 February 1998
Communicated by D. Kirkpatrick

ABSTRACT

Given a set S of n non-intersecting line segments in the plane, we present a new technique for efficiently traversing the endpoint visibility graph of S to solve a variety of visibility problems in output sensitive time. In particular, we develop two techniques to compute the $2n$ visibility polygons of the endpoints of S, in output sensitive time. Depth-first spiralling is a technique that relies on the ordered endpoint visibility graph information to traverse the endpoints of S in a spiral-like manner using a combination of Jarvis' March and depth-first search. It is a practical method and has been implemented in C++ using LEDA.

Keywords: visibility graphs, visibility polygons, depth-first spirals.

1. Introduction

Problems involving the *visibility* of objects in a given domain arise 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

*The author gratefully thanks N.S.E.R.C. for financial assistance.
†The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged.
‡The author gratefully thanks N.S.E.R.C. for financial assistance.

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. The text by O'Rourke [10] is the standard reference in the area. In this paper, certain 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.[6,17,1,12] Note that the endpoint visibility graph can be specified as either an abstract graph, or as a particular embedding, in which the clockwise *ordering* of the visibilities about each vertex is also provided. Here, we assume the latter specification.

Let E represent the number of edges in $G_e(S)$. It is clear that the maximum value of E is $n(n-1)/2$, and furthermore, Shen and Edelsbrunner [14] have shown that $E \geq 5n-4$ (see also Cambell and Higgins [2]). The problem of computing $G_e(S)$ *in output sensitive time* was first considered by Overmars and Welzl [11] who presented an $O(E \log n)$ time algorithm. This was improved by Ghosh and Mount, [6] who presented an $O(E + n \log n)$ time algorithm. This algorithm is primarily of theoretical interest, since it relies on rather complicated data structures. Pocchiola and Vegter [12,13] presented a more output-sensitive algorithm also running in $O(E \log n)$ based on a structure called the visibility complex.

Clearly, $G_e(S)$ does not capture all the visibility information of a set of line segments; for example, the view from a particular endpoint cannot be determined from $G_e(S)$. Let q and r be a pair of adjacent vertices in $G_e(S)$. We denote by **Stab(qr)** the segment visible to r in the direction $\vec{qr}$. For example, in Figure 1 Stab(qr) is S_1, and Stab(rq) is S_2. The stabbing ray may intersect no segment of S, in which case the symbol ∞ is returned by the function Stab(). This stabbing information is crucial in computing a more complete visibility picture.

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.[¶] In this paper, we present an optimal output sensitive algorithm that computes the set of visibility polygons from the $2n$ endpoints of S in $O(E)$ time after the endpoint visibility graph, $G_e(S)$ is computed. ¿From the visibility polygons, which require $O(E)$ space in total, several visibility problems can be solved.

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 a graph whose n

[§] The terms *endpoint* and *vertex* are used interchangeably for the remainder of the paper.

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

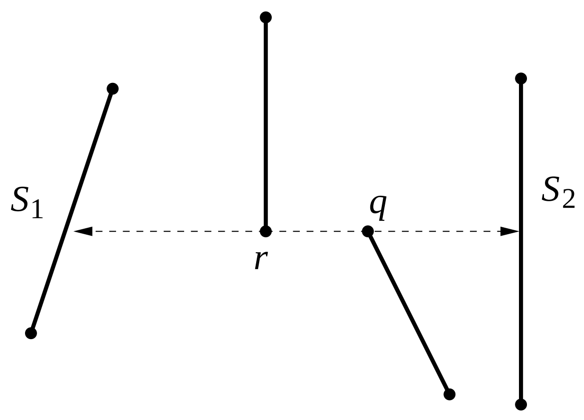

Fig. 1. Stabbing information

vertices correspond to the line segments of S and whose edges represent the visibility relation. See Figure 2 for examples of the two visibility graphs. As noted by Ghosh and Mount,[6] 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 $O(E + n \log n)$ time in the general case. Our second result is an efficient and practical solution to the problem of computing $G_f(S)$ which is superior to the algorithm of Ghosh and Mount for certain 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 and again is readily implementable.

One of the contributions of this paper is a new technique, called **depth-first spiralling**, for traversing the endpoint visibility graph to obtain efficient and practical, output sensitive solutions to several visibility problems. Depth-first spiralling sweeps through the endpoint visibility graph in a spiral fashion, backtracking as necessary and exploiting the inherent ordering information of $G_e(S)$, to visit each endpoint and to process the edges in an effective manner. We use this technique to compute the visibility polygons of all the endpoints in output sensitive time, from which the full visibility graph is easily obtained.

2. Computation of the Visibility Polygons

In this section, we present an $O(E)$ time algorithm for computing the $2n$ visibility polygons of the endpoints of a set S of n non-intersecting line segments, assuming that the (ordered) endpoint visibility graph $G_e(S)$ has already been computed.

The endpoint visibility graph information provides only part of the information needed to compute the visibility polygons. Consider an endpoint q. Each of its

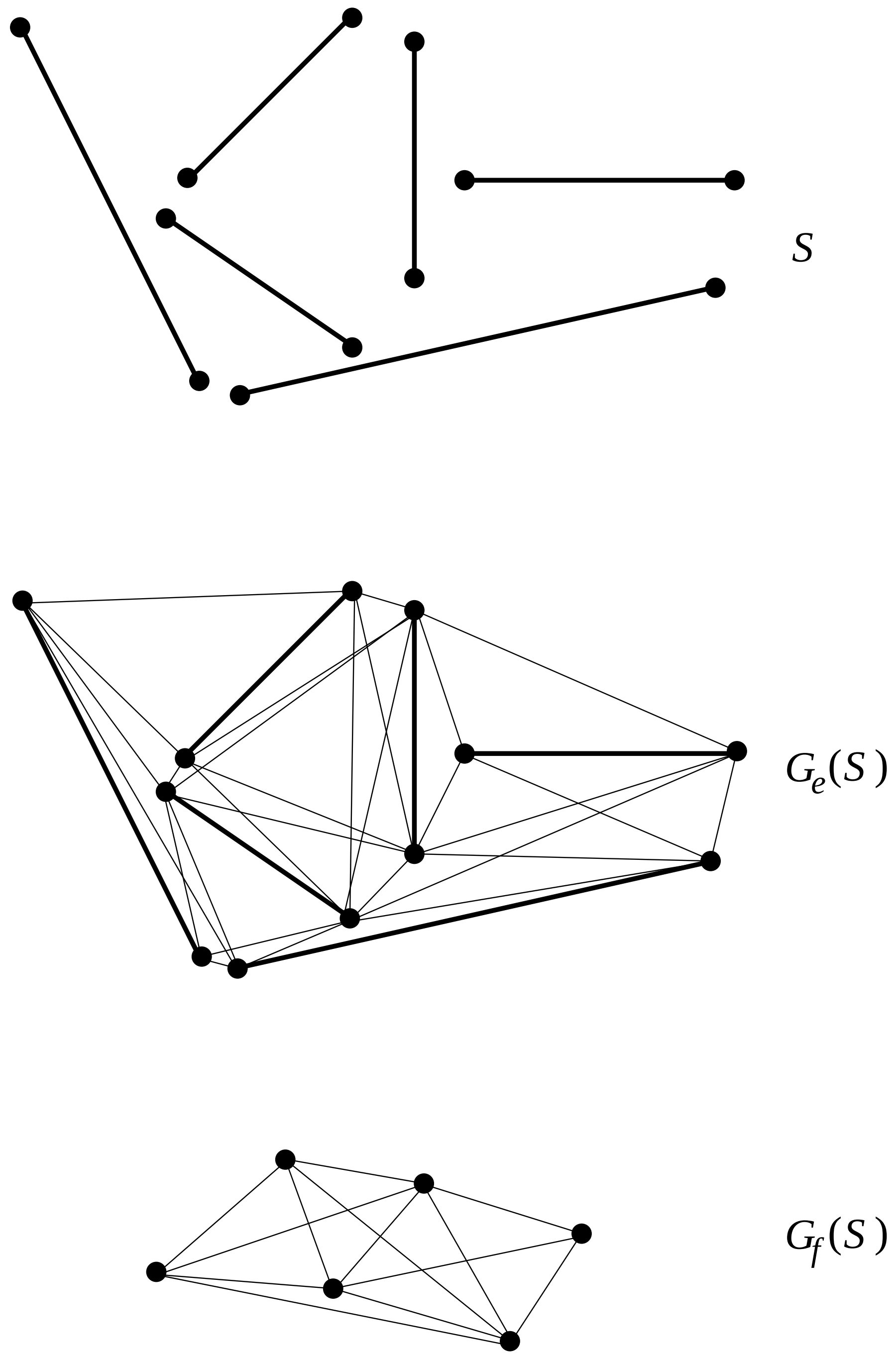

Fig. 2. The visibility graphs of S

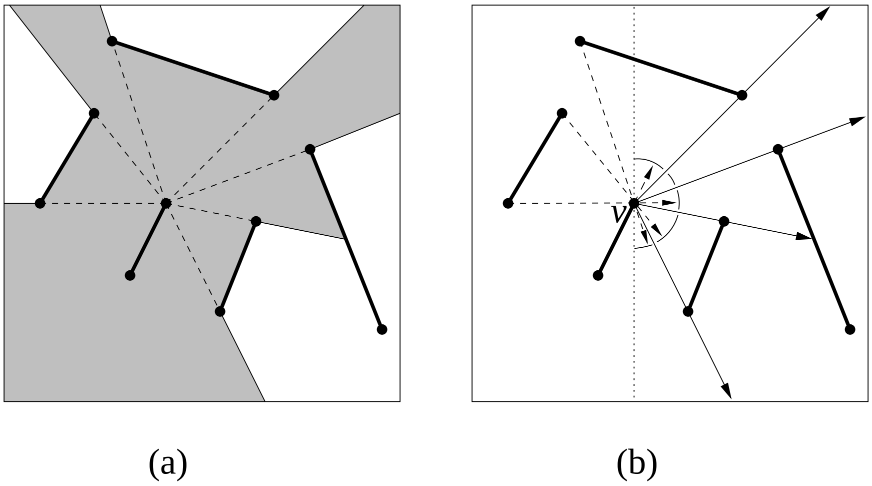

Fig. 3. Constructing visibility polygons

neighbours r in $G_e(S)$ contributes a vertex to $\mathrm{VPoly}(q)$. The remaining vertices of the visibility polygon are determined by the point at which a ray shot from q through r first strikes some segment of S (or it will be a vertex at infinity if the ray does not strike any segment). See Figure 3(a). In order to compute these vertices, it suffices to know $\mathrm{Stab}(qr)$, for each visible endpoint r. Recall Figure 1. Once the results of these stabbing queries are known for all of q's visible vertices, then it is an easy matter to traverse the cyclically ordered set of visible neighbours (as is provided by $G_e(S)$) to form $\mathrm{VPoly}(q)$. We assume that each undirected edge of the visibility graph is stored as two directed edges. The result of $\mathrm{Stab}(qr)$ is stored with the directed edge (q, r).

Thus it suffices to show that given just the endpoint visibility graph, $G_e(S)$, all of these $2E$ stabbing queries for all the endpoints can be computed in $O(E)$ time. Our algorithm to do this operates in two phases. The first traverses endpoints from right to left, resolving stabbing queries directed from left to right, and the second phase traverses endpoints from left to right and resolves stabbing queries directed from right to left. Given two endpoints v and w, we say that w lies to the right of v if the x-coordinate of w is greater than the x-coordinate of v, or if the two x-coordinates are equal and the y-coordinate of w is greater than the y-coordinate of v. We say that the stabbing query $\mathrm{Stab}(vw)$ is directed to the right. Otherwise w is to the left of v and the stabbing query is directed to the left.

Let us consider just the first phase, since the second phase is symmetrical. The algorithm begins by constructing a directed version of $G_e(S)$, in which all edges are directed to the right. This can be done in $O(E)$ time through any traversal of the graph. The resulting directed graph is acyclic. Sort the vertices of this graph in reverse topological order, and process them in this order. This implies that whenever a vertex v is processed, all of its visible vertices to the right have already been processed. This can be done in $O(E)$ time given the right-directed visibility graph by applying a standard depth-first search, and then processing each vertex as the depth-first recursion is backing out of the vertex.[4]

For each vertex v, assume inductively that all its right-directed stabbing queries have already been answered. (The basis case involves the rightmost vertex, whose right-directed stabbing queries are trivial.) Consider the edges of $G_e(S)$ joining v to vertices to its right. These edges partition the right half-circle about v into angular sectors. See Figure 3(b). For each such sector, all stabbing queries passing through v and entering into this sector have the same (known) answer.

To answer the stabbing queries that are entering v from the left, consider the visibility edges of $G_e(S)$ lying locally to the left of v, merge this angularly sorted list with the sorted list of sectors (after adding π to each angle). This merge can be performed in time proportional to the degree of v. As a result of the merger, each visible neighbour u lying to the left of v knows the sector into which the ray $\vec{uv}$ enters, and hence the result of this stabbing query is known. These left-side visibility edges are then labeled with the answers to their stabbing queries.

The running time of this procedure is bounded by the time to perform depth-first search on the visibility graph (for the topological sort) plus the merging time, which is just the sum of degrees. So the total time is $O(E)$. As mentioned above, given the answers to these queries, the visibility polygons can all be constructed in $O(E)$ time.

2.1. Depth-First Spiralling

In this section, we develop a second algorithm to compute the visibility polygons in $O(E)$ time. The technique is similar to the previous algorithm in that partial visibility polygons are computed and the associated stabbing information is reported. However, the order of processing is significantly different. The approach combines a modified Jarvis' March with depth-first search, to process the endpoints of S in a spiral-like manner. This depth-first spiralling is a technique which may prove to be useful in a broader context.

An early algorithm developed by Jarvis, [8] for computing the convex hull of a set of points, marches around the convex hull by initially choosing a point on the convex hull and then iteratively determining the next convex hull point by sorting all the remaining points by polar angle about the current convex hull point. If the algorithm is modified so that points are removed from consideration as they are visited, then all points are ultimately visited and the points are processed in a spiral-like manner. This is the basic paradigm used by our algorithm, albeit for the endpoints of line segments.

Consider the effect of a Jarvis march around the endpoints of a set of line segments. Choose the left-most endpoint as an initial point and proceed "wrapping" counter-clockwise, iteratively choosing the next *visible* unvisited endpoint. Again, the endpoints will be visited in a spiral-like fashion[‖], however when the algorithm terminates at an endpoint with no visible neighbours, some endpoints may not

[‖]The turns are not necessarily monotonically to the left.

have been visited. In order to ensure that all endpoints are ultimately visited, the algorithm will perform in depth-first search mode, backtracking along the spiral until an endpoint with unvisited visible neighbours is discovered at which stage, the spiralling process will resume. The effect will be to spawn spiral branches off the main spiral (see Figure 6 for example).

Jarvis' convex hull algorithm is not optimal in the worst case - too much time is expended (re)sorting at each stage. Note however that for our problem, the endpoint visibility graph $G_e(S)$, contains exactly the endpoints visible to each endpoint in sorted polar order.

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 when 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.

The algorithm computes partial visibility polygons based on previously answered stabbing queries. Let **VPoly**(v, a, b) denote the portion of the visibility polygon counterclockwise about vertex v from edge a to edge b **. 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 endpoints r visible to v counterclockwise between a and b. Once VPoly(v, a, b) is computed, the stabbing queries *to* v that are answerable via this portion of the visibility polygon of v must also be reported. These are exactly the endpoints visible to v that lie in the region π radians about v from VPoly(v, a, b). Let α and β denote the respective polar angles of edges a and b about v. Let **Stabs**(v, a, b) denote the set of stabbing queries from v to its visible vertices w that lie in the angular sector about v counterclockwise from $\alpha + \pi$ to $\beta + \pi$. See Figure 4. Once computed, the result of each stabbing query must be accessible in $O(1)$ time for a given edge. This is easily accomplished by storing the result directly with the edge in the data structure. Both VPoly(v, a, b) and

**If $a = b$ then the entire visibility polygon is meant.

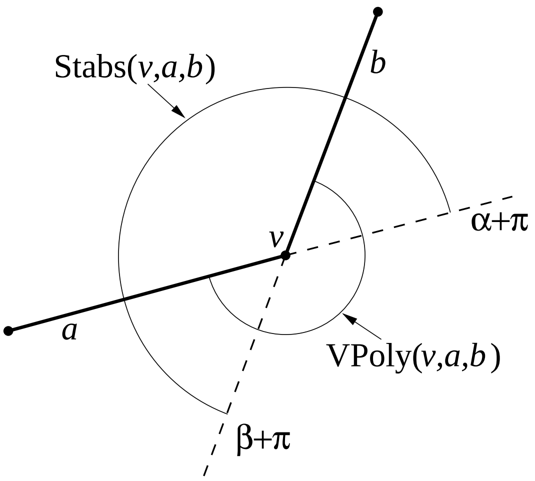

Fig. 4. VPoly and Stabs

Stabs(v, a, b) can be computed by a simple rotational sweep about v in $O(d_v)$ time, where d_v denotes 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 are marked as *visited* 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 5 and 7 for examples of right turn vertices.

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

`Procedure Spiral` (v : `vertex`; e : `edge`)

e' : `edge`

v' : `vertex`

`known` [v] $:= e$

$e' := e$

`For each vertex` v' `adjacent to` v `(counterclockwise after` 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'`)`

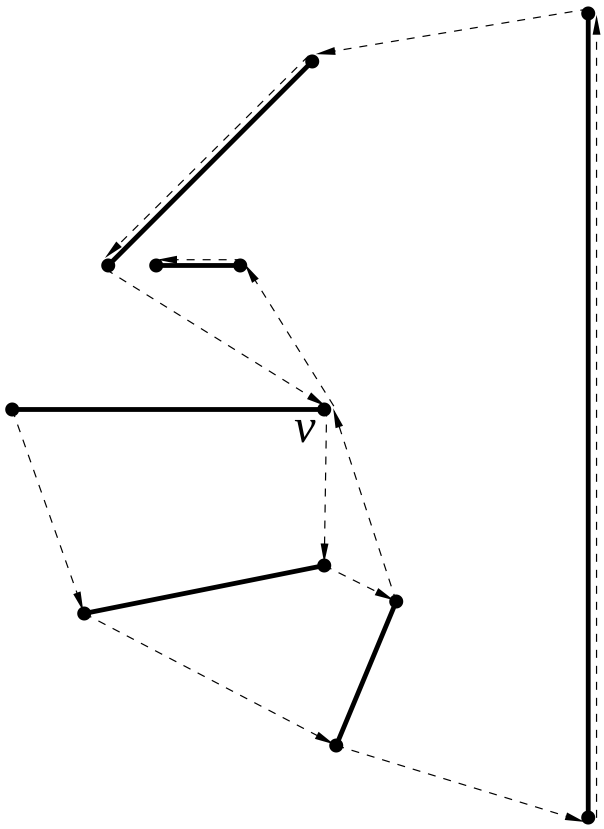

Fig. 5. Right turns in spirals

```
      * 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 -- now 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}
```

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 still have

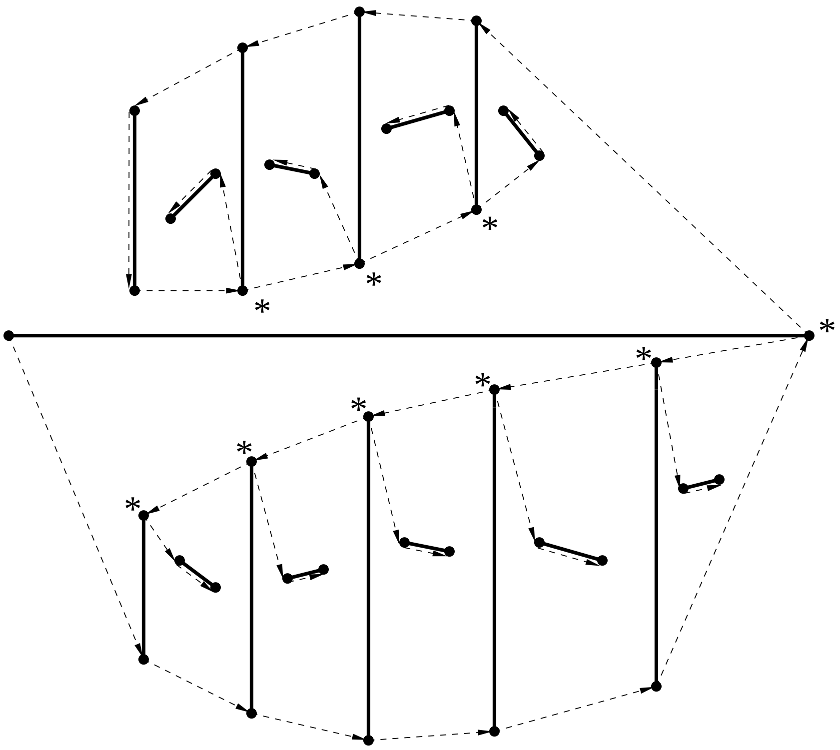

Fig. 6. Backtracking through left turn endpoints

unvisited neighbours. For example, the endpoints marked with "*" in Figure 6 are all left turn endpoints with this property.

2.2. Correctness

As will be seen, it is critical that the "spiral" generated by the algorithm 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 7 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: (By contradiction) There are two cases to consider:

Assume the spiral crosses itself *at an endpoint* v, (refer to Figure 8). Note that v must be a right turn endpoint in this case since left turn endpoints are only visited once. 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

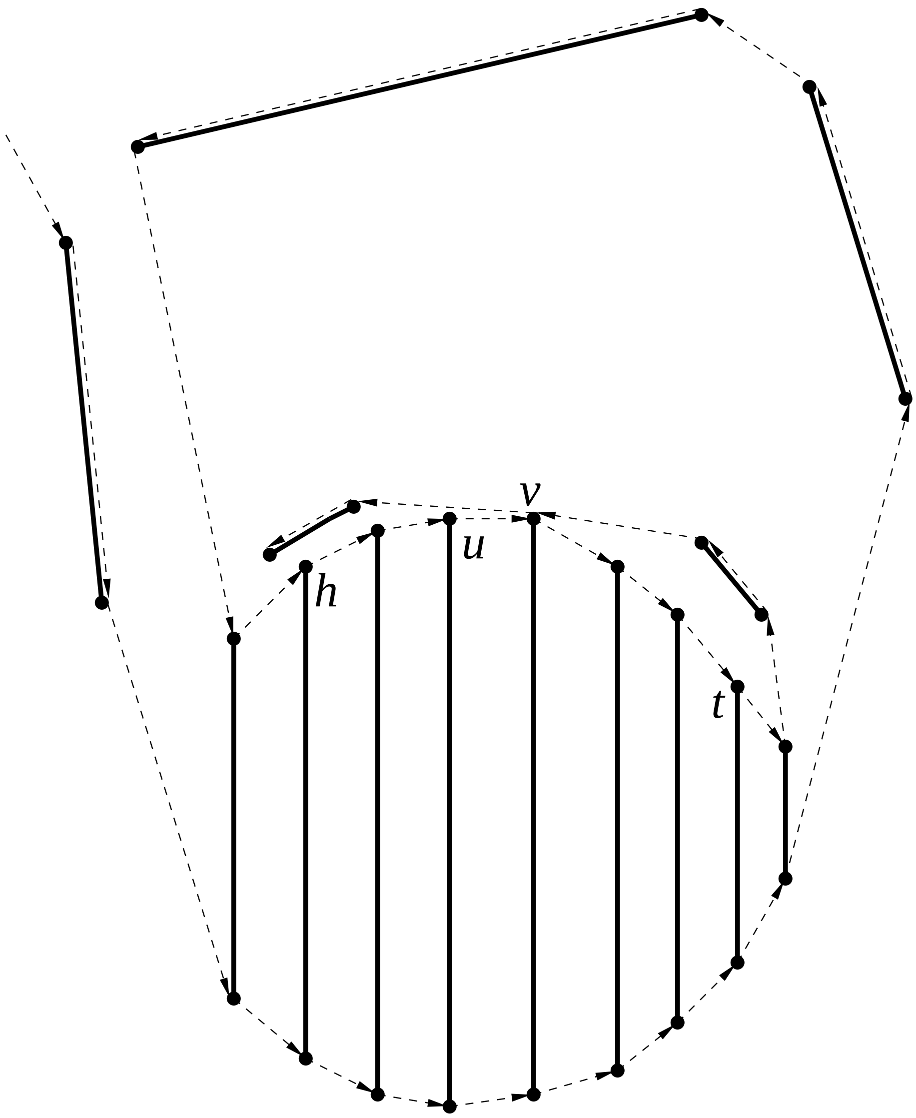

Fig. 7. Adjacent right turns in spirals

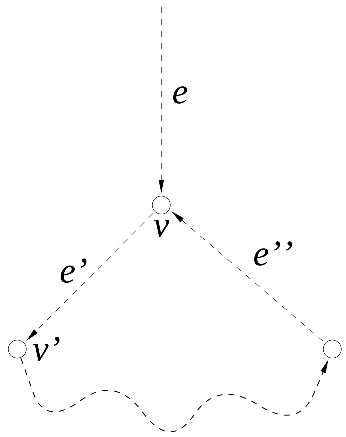

Fig. 8. Endpoint crossing

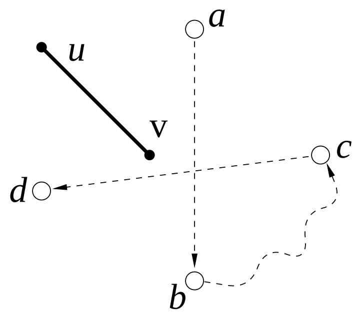

Fig. 9. Edge crossing

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'' cannot lie in that region and hence the spiral cannot actually cross itself at v [††].

Assume the spiral crosses itself at an edge as in Figure 9. Let the endpoints creating the first such edge crossing on the spiral be in order, 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 cannot 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:

- either v was a right turn endpoint (and hence unmarked), which would contradict the choice of b as the next visited endpoint on the spiral after a, or

[††] In fact, if v is a right turn vertex, then the line segment containing v must lie strictly on the outside of the spiral at v (i.e. counterclockwise between e and e').

- v was a left turn vertex. In this case, the spiral through v must cross (a, b) or (c, d), thus contradicting the assumption that (a, b) (c, d) formed the first spiral crossing.

A similar argument handles the case when the edge crossing occurs in a clockwise manner. □

The correctness of the algorithm can now be established.

Theorem 1 *The algorithm correctly computes the* $2n$ *visibility polygons of the endpoints of a set of* n *non-intersecting line segments.*

Proof: 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 (i.e. between e and e') and thus $\mathrm{VPoly}(v, e, e')$ and $\mathrm{Stabs}(v, e, e')$ are computable in $O(d_v)$ time. The previous lemma ensures that there are no unprocessed visible neighbours of v counterclockwise between e and e'. □

2.3. 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 7 for example). If the spiral exited 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. Unfortunately, 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 along previously used edges more than once. 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, by jumping to the tail of the queue. Whenever the edge on the front of the queue is encountered, the spiral must follow the same chain of right turn vertices as on the previous trip - only the head and tail vertices will be modified. Note that the queue may be entered in the middle (as in Figure 7 for example), in which case the queue must be split if an interior vertex becomes marked as *visited*. This splitting operation can be performed in constant time as follows.

Consider a queue Q of edges between right turn endpoints with a head pointer h, and a tail pointer t. If an endpoint in the queue is subsequently encountered by the spiral at some interior endpoint v and v becomes a left turn endpoint, then Q must be "split" into two pieces: an active queue from h to u (where u is the vertex just before v in the queue), and a (temporarily) inaccessible piece from v to t. Now, since this latter queue is inactive, the queue need not actually be split. Rather, a stack containing head-tail pairs can be maintained. In this case, the value of t for the tail and v for the head are put on the stack and t is assigned the value of u, thus creating the active queue from h to u. Note that all the endpoints in the active portion of the queue will become marked as *visited* before the inaccessible endpoints in the inactive portion of the queue are processed. Since the spiral does not cross itself, these inaccessible endpoints are in fact processed exactly when v is backed into by the recursion, at which point the head-tail stack is popped and the previously inactive queue is restored and processed appropriately. This "splitting" or hiding operation can be performed in constant time since the queue pointers are not actually destroyed. Although Q may be split an arbitrary number of times, at any stage, there is only a single active head and tail pair available (in constant time) to the active endpoints in each queue.

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 outgoing edge be found quickly - i.e. the algorithm cannot afford to perform the same counterclockwise scan of visible *visited* vertices each time the endpoint v 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, the running time can now be shown to be $O(E)$.

Theorem 2 *Procedure Spiral runs in $O(E)$ time and requires $O(E)$ space.*

Proof: 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. 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 6). 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 $O(d_v)$.

Now, 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 $O(E)$ bound on the running time.

Static space is required for the ordered visibility graph $G_e(S)$, and the arrays *known* and *mark*. At run time, the total size of the dequeues is bounded by $O(N)$ and the recursion is no deeper than E. Thus, the space requirements of the algorithm can be contained to $O(E)$.

□

3. Implementations

Simple versions of both the spiral and topological algorithms for computing the endpoint visibility polygons have been implemented in C++ using LEDA (Library of Efficient Data types and Algorithms). The LEDA package, which is available from the Max-Planck-Institut fuer Informatik,[9] is a comfortable environment for implementing geometric algorithms and provided the basic data structures and X11 windowing facilities used to construct the visibility polygons of the endpoints of the line segments.

Our implementations of these and other routines are available free of charge for research and teaching purposes as part of an ongoing visibility project. Refer to: `http://www.cs.uleth.ca/~wismath/vis.html`. The binary version runs on SPARC workstations under Solaris, and source code is also obtainable for porting to other environments. A more detailed account of the implementation is described in Ref. [7].

4. The Full Visibility Graph

The 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 if, and only if, the corresponding line *segments* are visible.

Although the algorithm of Ghosh and Mount,[6] can be modified to compute the full visibility graph in optimal *output sensitive* time $O(E+n\log n)$, the algorithm is not practical to implement as it requires sophisticated data structures (finger trees) to store funnel sequences and contains an extremely involved SPLIT procedure to update the visibility information.

In Ref. [18], a worst case optimal $O(n^2)$ time and space algorithm was presented, that computes the full visibility graph, $G_f(S)$. The algorithm 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. This initial preprocessing step truly requires $O(n^2)$ time and space as an arrangement of lines must be maintained in order to answer visibility queries in linear time. However the algorithm to compute $G_f(S)$ in Ref. [18] 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.

The advantages of this "modular" strategy are:

- practicality: the algorithm of Overmars and Welzl,[11] for example, which runs in $O(E \log n)$ time has been implemented and runs efficiently.

- versatility: In restricted domains, the computation of the endpoint visibilities can be performed faster than for the general case. For example, Hershberger,[5] has shown that for polygons, the vertex visibility graph can be obtained in $O(n + E)$ time[‡‡].
- dynamization: Vegter [16] has shown how the endpoint visibility graph can be maintained to accommodate insertions and deletions of line segments.

The algorithm to compute the full visibility graph, $G_f(S)$, as described by Wismath [18] is as follows:

- Preprocess S to construct an arrangement of lines using $O(n^2)$ time and $O(n^2)$ space, as outlined in Ref. [1].
- For each endpoint q of each segment u do
  - compute the visibility polygon VPoly(q) as described in Ref. [1].
  - report as visible u and each segment in VPoly(q).
  - report the visibilities *around* q (i.e. the segments of S that are π radians apart about q in VPoly(q)) with a rotational sweep technique.

The correctness of this algorithm is shown in Ref. [18]. Here we note that the algorithm actually runs in $O(E)$ time after the visibility polygons have been constructed. Combining this algorithm with the algorithm of the previous section yields an $O(E)$ time solution to the problem of computing the full visibility graph, after the endpoint visibility graph has been computed.

4.1. Dynamic Maintenance of the Full Visibility Graph

When segments are inserted or deleted in a dynamic fashion, the full visibility graph may be updated by maintaining the endpoint visibility graph. Vegter [16] has shown how to update the endpoint visibility graph in $O(\log^2 n + K \log n)$ time, where K represents the number of edges of the visibility graph that are changed. Although Vegter's algorithm, which performs in dual space, could be modified to update the full visibility graph directly, once again, separating the two problems may be beneficial. If the insertions and deletions have a special form, (for example, if the line segments form a polygon), then perhaps both the endpoint and full visibility graphs can be maintained efficiently by exploiting the underlying structure. We leave as an open problem the dynamic maintenance of the full visibility graph, given the endpoint visibility graph update information.

5. Conclusion

The endpoint visibility graph $G_e(S)$, of a set of line segments can be computed in $O(E + n \log n)$ time in the general case, [6] but in special cases, can be computed in $O(E)$ time. Given $G_e(S)$, the main contributions of this paper are

‡‡The linear time triangulation algorithm of Chazelle [3] is required.

- an efficient algorithm to compute all $O(E)$ ray shooting queries between pairs of visible endpoints in $O(E)$ time.
- a new and practical technique, called depth-first spiralling, for traversing $G_e(S)$ in output sensitive time.
- an algorithm for computing the $2n$ visibility polygons of the endpoints of a set of line segments in $O(E)$ time.
- an algorithm to compute the full visibility graph of S in $O(E)$ time.

All the algorithms have been implemented in C++ (using LEDA) and are available via the WWW.

6. 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. D. Campbell, J. Higgins, Minimal visibility graphs, *Information Processing Letters* 37 (1991), 49-53.
3. B. Chazelle, Triangulating a simple polygon in linear time, *Discrete and Computational Geometry* Vol. 6, No. 5 (1991), 485-524.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA (1990).
5. J. Hershberger, An Optimal Visibility Graph Algorithm for Triangulated Simple Polygons, *Algorithmica* **4** (1989), 141-155.
6. S. Ghosh, D. Mount, An Output Sensitive Algorithm for Computing Visibility Graphs, *SIAM J. of Computing*, Vol. 20, No. 5, (1991), 888-910.
7. L. Jackson, H. Pinto, S. Wismath, VisPak: A Package of Visibility Algorithms Written in Leda, U. of Lethbridge technical report, CS-95-1, 1995.
8. R. A. Jarvis, On identification of the convex hull of a finite set of points in the plane, *Information Processing Letters* 2 (1973), 18-21.
9. K. Mehlhorn, S. Naher, LEDA: A platform for combinatorial and geometric computing, *Communications of the ACM*, January 1995, Vol 38, No. 1, 96-102.
10. J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press (1987).
11. M. Overmars, E. Welzl, New methods for computing visibility graphs, *Proceedings of the 4th Symposium on Computational Geometry*, Urbana-Champaign, 1988, 164-171.
12. M. Pocchiola, G. Vegter, Computing the Visibility Graph via Pseudo-triangulations, *Proceedings of the* 11^{th} *ACM Symposium on Computational Geometry*, 1995, 248-257.
13. M. Pocchiola, G. Vegter, The visibility complex, *Internat. J. Comput. Geom. Appl.*, 6 (1996), 279-308.
14. X. Shen, H. Edelsbrunner, A tight lower bound on the size of visibility graphs, *Information Processing Letters* 26 (1987/88), 61-64.
15. G. Toussaint, Shortest Path Solves Edge-to-Edge Visibility in a Polygon, *Pattern Recognition Letters*, 4 (1986), 165-170.

16. 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.

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

18. S. Wismath, Computing the full visibility graph of a set of line segments, *Information Processing Letters* 42 (1992), 257-261.