COMPUTING THE VISIBILITY POLYGONS OF THE ENDPOINTS OF A SET OF LINE SEGMENTS IN OUTPUT SENSITIVE TIME *

MARK KEIL † AND STEPHEN WISMATH‡

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 of S in a spiral-like manner using a combination of Jarvis' March and depth-first search.

One extension of this result is an efficient 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 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 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, 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 [5], [11], [1]. 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 [8] 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 solved by Ghosh and Mount [5],who presented an $O(E + n \log n)$ 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 $O(E)$ time after the endpoint visibility graph, $G_e(S)$ is computed.

* THIS RESEARCH WAS SUPPORTED BY N.S.E.R.C.

† Dept. of Computational Science, University of Saskatchewan, Saskatoon, Canada

‡ Dept. of Math. and Computer Science, University of Lethbridge, Alberta, Canada

[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.

FIG. 1. *The visibility graphs of S*

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 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[5], 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 solution to the problem of computing $G_f(S)$ which outperforms 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.

2. Computation of the 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.

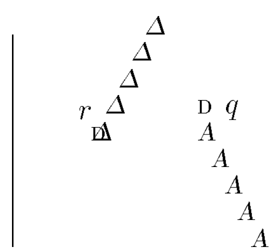

FIG. 2. *Stabbing Query*

If the visibility polygon of r were known, then this stabbing query could be answered in $O(\log d_r)$ time via binary search. If endpoint r is on the convex hull of S, then all the stabbing queries from r are in fact immediately known. Note that the visibility between q and r also generates a stabbing query in the opposite direction which is required for the computation of VPoly(r).

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

An early algorithm developed by Jarvis [6], 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" counterclockwise, iteratively choosing the next *visible* unvisited endpoint. Again, the endpoints will be visited spirally, however when the algorithm terminates at an endpoint with no visible neighbours, some endpoints may not 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 4 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 the previously answered stabbing queries. Let $\mathbf{VPoly}(v, a, b)$ denote the portion of the visibility polygon counterclockwise about vertex v from edge a to edge b [2]. 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. 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 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 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)
e' : edge
v' : vertex
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 -- now unwind}
if e = e' then mark[v] := visited                          { bottom out case}
compute VPoly(v, known[v], e)                         {VPoly(v) is now complete}
```

[2] If $a = b$ then the entire visibility polygon is meant.

FIG. 3. *Right turns in spirals*

`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 unvisited neighbours. For example, the endpoints marked with "*" in figure 4 are all left turn endpoints with this property.

2.1. Correctness. 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 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: (By contradiction) There are two cases to consider:

FIG. 4. *Backtracking through left turn endpoints*

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 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 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 [3].

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 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 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:

- 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

[3] 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').

FIG. 5. *Adjacent right turns in spirals*

FIG. 6. *Endpoint crossing*

FIG. 7. *Edge crossing*

in the spiral, contradicting the assumption that (a, b) (c, d) was the first such 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 2.1. *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 VPoly(v, e, e') and 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'. $\square$

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. 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 5 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 first vertex 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 can not 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 2.2. *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 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 $O(d_v)$.

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. Extensions. In this section, we consider extensions to the previous result: a searchlight scheduling problem, and the problems of computing and dynamically maintaining the full visibility graph of S.

3.1. 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 [5], 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 [12], 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 [12] 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 [7], 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 [4], has shown that for polygons, the vertex visibility graph can be obtained in $O(n+E)$ time[4].
- dynamization: Vegter [10], 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 [12] is as follows:

- Preprocess S to construct an arrangement of lines using $O(n^2)$ time and $O(n^2)$ space, as outlined in [1].
- For each endpoint q of each segment u do
  - compute the visibility polygon VPoly(q) as described in [1].
  - 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.

The correctness of this algorithm is shown in [12]. 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.

3.2. 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 [10] 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

[4] using the linear time triangulation algorithm of Chazelle [3]

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.

3.3. Search Light Scheduling. The depth first spiralling technique may also prove useful in other visibility problems where the endpoints of line segments must be ordered in some fashion. For example, consider the following search light scheduling problem. Suppose that each line segment represents a fixed barrier and each endpoint corresponds to a guard tower equipped with a rotatable search light. The problem is to devise a schedule for searching the plane in order to guarantee the illumination of a moving "invader". Communication between guards is permitted only via search lights. The layout of the barriers, and the location of the invader are unknown to the guards. We assume that the guard towers are transparent, that is, light passes through them. A single "initiator guard" located on the convex hull is allowed two beacons, one of which is fixed pointing to the outside and the second of which is swept from the first beam counterclockwise.

All guards now perform identically and at any stage each guard g is in one of four modes:

- *waiting*: The light is off and guard g is waiting to be illuminated by some neighbouring guard. When initially illuminated by some guard f, guard g points the search light in the direction of f and switches to *sweep* mode.
- *sweeping*: Guard g rotates the light counterclockwise until a guard tower h is encountered. If h is *waiting* (i.e. unlit but subsequently illuminated) then g *pauses*. If h is *pausing* (i.e. already lit) then g continues sweeping. Sweeping continues until a rotation of 360 degrees has been made at which point the guard is *done*.
- *pausing*: A paused guard g waits for the guard it has illuminated to be *done* and then reverts to sweeping mode.
- *done*: The beacon is permanently extinguished.

In this case, right turn vertices do not require the special treatment that they received in the Spiral algorithm; it is the counterclockwise ordering of the endpoints and the spirality of the search that is critical. Note that at any stage, there exists a spiral shaped outer region that is guaranteed to be invader-free and which is separated from the unsearched inner region(s) by a series of barriers and pausedly fixed lights.

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 $O(E)$ time, after the endpoint visibility graph has been computed (in the general case this preprocessing step can be performed in $O(E + n \log n)$ time [5] but can be contained to $O(E)$ in special cases, for example, if the line segments form a simple polygon [4]).

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

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] 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.

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

[6] 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.

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

[8] X. Shen, H. Edelsbrunner, A tight lower bound on the size of visibility graphs, *Information Processing Letters* 26 (1987/88), 61-64.

[9] G. Toussaint, Shortest Path Solves Edge-to-Edge Visibility in a Polygon, *Pattern Recognition Letters*, 4 (1986), 165-170.

[10] 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.

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

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