# New methods for computing visibility graphs

Extended abstract

Mark H. Overmars and Emo Welzl

RUU-CS-88-7
March 1988

Rijksuniversiteit Utrecht

**Vakgroep informatica**

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

# New methods for computing visibility graphs*

Extended abstract

Mark H. Overmars† Emo Welzl‡

## Abstract

Let $S$ be a set of $n$ non-intersecting line segments in the plane. The visibility graph $G_S$ of $S$ is the graph that has the endpoints of the segments in $S$ as nodes and in which two nodes are adjacent whenever they can "see" each other (i.e., the open line segment joining them is disjoint from all segments or is contained in a segment). Two new methods are presented to construct $G_S$. Both methods are very simple to implement. The first method is based on a new solution to the following problem: given a set of points, for each point sort the other points around it by angle. It runs in time $O(n^2)$. The second method uses the fact that visibility graphs often are sparse and runs in time $O(m \log n)$ where $m$ is the number of edges in $G_S$. Both methods use only $O(n)$ storage.

## 1 Introduction.

Given a set $S$ of $n$ non-intersecting closed line segments, the *visibility graph* $G_S$ of $S$ is the undirected graph that has the endpoints of the segments as nodes and in which two nodes (endpoints) are adjacent whenever they "see" each other, i.e., the open line segment joining them is disjoint from all segments in $S$ or is contained in a segment.

Visibility graphs have a number of applications; for example, visibility graphs have been used several times as an approach to shortest path problems in the plane (see e.g., Lozano-Perez and Wesley[7]).

The trivial method for constructing a visibility graph takes time $O(n^3)$. The first efficient algorithm was due to Lee[6] and Sharir and Schorr[9] and runs in time $O(n^2 \log n)$. Later Asano et al.[1] and Welzl[10] gave an $O(n^2)$ algorithm which is optimal in the worst-case. Edelsbrunner and Guibas[2] improved the working storage of this method to $O(n)$. Recently, Ghosh and Mount[3] designed an optimal $O(m + n \log n)$ algorithm where $m$ is the size of $G_S$. Their method uses $O(m)$ storage.

Although the latest method is optimal in time, the authors themselves state that the method will be very hard to implement and implementations might be slow due to high constants.

In this paper we present two new methods for computing the visibility graph. Although not optimal both methods are simple to implement and do run efficiently. Moreover, both methods use only $O(n)$ working storage. The techniques used are new and might have interesting other applications as well.

The first method we propose is based on the technique of [10]. As a basic step it uses a solution to the following problem: Given a set of $n$ points, for each point $p$ compute the order of the other points by angle around $p$. This problem has already been solved in optimal $O(n^2)$ time and $O(n)$ storage by Edelsbrunner and Guibas[2] using dualization and topologically sweeping. We obtain the same complexity in a different way, without using dualization. The method is very simple. It took us less than 2 hours to program and the resulting code has less than 100 lines. It also handles all special cases, like multiple points on a line, without any problems. This technique will

*Research of the second author was supported by the Austrian ministry for "Wissenschaft und Forschung", project "Sichtbarkeitsgraphen".

†Dept. of Computer Science, University of Utrecht, P.O.Box 80.012, 3508 TA Utecht, the Netherlands.

‡Dept. of Mathematics, Free University Berlin, Arnimallee 2-6, 1000 Berlin 33, West Germany. Research associate of: IIG, Inst. f. Information Processing, Technical University Graz, Austria.

have many more applications as well (see [2]). The basic idea is to maintain a tree on the set of points during one rotational sweep.

The second method uses the fact that visibility graphs are often sparse, i.e., the number of edges is $o(n^2)$. The method again uses a tree on the endpoints of the line segments during a rotational sweep but, using some simple extra structure, we take care that only edges of the visibility graph will be considered. The method runs in time $O(m \log n)$ where $m$ is the size of the visibility graph. It uses $O(n)$ storage. Also this method was easy to implement and again it handled all special cases.

The paper is organized as follows. In Section 2 we describe the technique used for sorting points by angle around each point. In Section 3 we use this technique to find the visibility graph. In Section 4 we describe the second method, reducing the time bound for sparse visibility graphs. In Section 5 we make some concluding remarks and describe some extensions and open problems.

In this extended abstract some proofs are omitted or only briefly indicated. However, the methods are described in detail. To simplify the proofs and description (not the methods) general position is assumed. In the full paper these restrictions will be removed. (Note that we already implemented the methods without these restrictions.) A preliminary version of part of this work appeared in [8].

## 2 Rotation trees.

In this section we will describe our new method for the following problem: given a set $V$ of $n$ points, determine for each point $p$ of $V$ the other points ordered around $p$. We will give a simple solution that works in optimal time $O(n^2)$ and optimal $O(n)$ space. In fact, the method we present will report for each point $p$ only the points to the right of $p$ sorted by angle. For the application of visibility graph (and other applications) this is exactly what is needed. The method can easily be adapted to give all points.

First we describe a method that should provide some intuition behind the actual algorithm. We add two points $p_\infty$ and $p_{-\infty}$ to the pointset $V$ which represent points with $y$-coordinate $+\infty$ ($y$-coordinate $-\infty$, resp.) and $x$-coordinate larger than all points in $V$. For a direction $d$ between $-\pi/2$ and $\pi/2$ we define the *rotation tree $G_d$ following direction $d$* as the ordered tree with node set $V \cup \{p_\infty, p_{-\infty}\}$ where every point point $p$ (except for $p_\infty$) has exactly one outgoing edge to the point $q$ with the following property: $q$ is the first point encountered when looking from

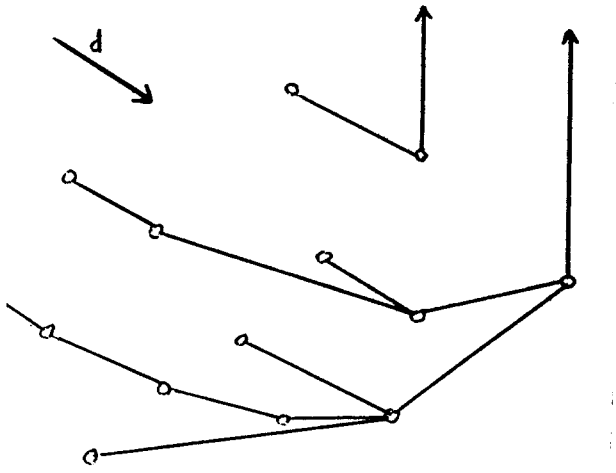


Figure 1: A rotation tree following direction $d$.

$p$ in direction $d$ and rotating counter clockwise. In other words, the directed line segment $\overline{pq}$ is the one with smallest slope not smaller than $d$. So $p_\infty$ is always the root of the tree and no edge in the tree has slope larger than $\pi/2$. The ingoing edges of a point are ordered by slope such that the edge with smallest slope is the leftmost one. (Note that whenever we talk about a tree, the edges are directed towards the root.) See figure 1 for an example.

For $d = -\pi/2$, every point in $V$ has $p_{-\infty}$ as its father, the point in the set with largest $x$-coordinate in $V$ being the leftmost son; $p_{-\infty}$ has $p_\infty$ as its father. For $d = +\pi/2$, all points are sons of $p_\infty$, the point with smallest $x$-coordinate as leftmost son.

The basic idea is to start with $G_d$ for $d = -\pi/2$ and increase $d$ while updating the tree until $d = +\pi/2$. During this process, whenever an edge $\overline{pq}$ is replaced (which happens when $d$ sweeps over the slope of $\overline{pq}$) then we report this edge. (From now on slope($\overline{pq}$) denoes the slope of $\overline{pq}$.)

As mentioned before, what we just described is only the underlying intuition of what we are going to do. The reason for this is that in order to find the edge to change in the process we need a priority queue which would increase the complexity.

Instead we will always replace the edge of the rightmost leaf in the tree that is leftmost son of its father. This edge will be called the *rightmost left leaf-edge*. The clue is that this will give us still the points sorted around each point and it allows us to find the edge to replace in constant time.

Of course, if we proceed like this, the tree will not keep such a clear geometric interpretation as the rotation tree following a direction $d$. Nevertheless, it will share crucial properties which we will keep as an invariant.

**Property 1** *(Basic properties)*

1. $p_\infty$ *is the root of the tree.*
2. *The incoming edges of a node are ordered by slope such that the edge with smallest slope is leftmost.*
3. *For every edge* $\overline{pq}$, $-\pi/2 \leq \overline{pq} \leq \pi/2$.
4. *If* $\overline{pq}$ *and* $\overline{qr}$ *are edges then slope*$(\overline{pq}) \leq$*slope*$(\overline{qr})$.

Hence, every path from a point in the tree to the root is a convex curve of increasing slope.

**Property 2** *(Order property) Let p, q and r be nodes in the tree such that* $\overline{pq}$ *is an edge. If slope*$(\overline{pq}) <$ *slope*$(\overline{pr}) \leq \pi/2$, *then r preceeds p in the preorder of the tree, or, in other words, either r lies on the path from p to the root or it is in a left subtree of this path. If* $-\pi/2 \leq$ *slope*$(\overline{pr}) <$ *slope*$(\overline{pq})$ *then r succeeds p in the preorder of the tree.*

The order property tells us that if we remove an edge $\overline{pq}$ from the tree, then the next point $z$ in order around $p$ can be found on the path to the root or in a left subtree of this path. We will see later on that $z$ can be specified in a more precise way.

**Property 3** *(Cone property) Let* $\overline{pq}$ *and* $\overline{rs}$ *be edges in the tree such that slope*$(\overline{pq}) \leq$ *slope*$(\overline{pr}) <$ *slope*$(\overline{rs})$. *Then no point in V lies in the cone which is the intersection of the open halfplane to the right of* $\overline{rs}$ *and the closed halfplane to the left of* $\overline{pr}$.

We define now a *rotation tree on V* as a tree which satisfies the three properties described above. It is not hard to prove that any rotation tree following a direction $d$, $-\pi/2 \leq d \leq +\pi/2$ satisfies the three properties and, thus, it is a rotation tree on $V$.

The first crucial lemma is:

**Lemma 2.1** *Let G be a rotation tree on V and let* $\overline{pq}$ *be the rightmost left leaf-edge in G. If* $\overline{pq}$ *is replaced by* $\overline{pz}$ *where z is the next point (after q) in order around p then the resulting graph is a rotation tree.*

In order to specify the next point $z$ around $p$, we introduce the following notion.

**Definition 2.1** *A* chain *in a rotation tree is a maximal sequences of edges* $e_0, e_1, ..., e_i$ *such that each* $e_{j+1}$ *starts where* $e_j$ *ends and all edges except for* $e_i$ *are rightmost incoming edges of their endpoint.*

**Lemma 2.2** *Let G be a rotation tree on V and let* $\overline{pq}$ *be the leftmost edge of q* ($q \neq p_\infty$). *Let r be the father of q and let z' be the left brother of q, provided it exists. Then the next point z around p (after q) is the tangent (from the right) from p to the chain ending in* $\overline{z'r}$, *or, if z' does not exist,* $z = r$.

Note that the lemma is not only formulated for the rightmost left leaf-edge. That is, if we replace an arbitrary edge then we find the next point correctly. However, the resulting graph after the replacement is not necessarily a rotation tree and later replacements in the process may become faulty. (Such examples can be constructed.)

The two lemmas determine the algorithm. The only things we have to be able to do is to determine the rightmost leaf that is leftmost son of its father and we have to be able to determine the tangent point. For the first task we use a simple stack that will contain all leaves that are leftmost son of their father. The second task is simply performed by traversing the chain backwards.

We will now describe the algorithm precisely. Each point in the set will be stored as a record that has fields for the $x$- and $y$-coordinate of the point and four pointers to its father to its left and right brother and to its rightmost son. We use the following procedures on the points that can all be implemented in O(1) time in a few lines of code.

AddRightmost($p, q$):
    Adds $p$ as rightmost son to $q$.
AddLeftOf($p, q$):
    Adds $p$ as left brother of $q$.
Remove($p$):
    Removes $p$ from the tree.
RightBrother($p$):
    Returns the right brother of $p$.
LeftBrother($p$):
    Returns the left brother of $p$.
Father($p$):
    Returns the father of $p$.
RightmostSon($p$):
    Returns the rightmost son of $p$.

As initialization we need a procedure Sort($V$) that sorts the set of points by decreasing $x$-coordinate. (For equal $x$-coordinates we sort by decreasing $y$-coordinate.) To compare angles we use a procedure LeftTurn($p, q, r$). It returns **true** if $r$ lies to the left of the directed line $\overline{pq}$. Otherwise (on the line or to the right) it returns **false**. When $r = p_\infty$ we have to be a bit more careful. TurnLeft($p, q, p_\infty$) will be **true** when $p.x < q.x$ or $p.x = q.x$ and $p.y < q.y$. (Note that none of the points with which TurnLeft is called can be $p_{-\infty}$ nor can $p$ or $q$ be $p_\infty$.) The routine Handle will report a pair $pq$ to be used by the application.

Finally we use a stack that contains all the leaves that are leftmost sons of their fathers. Always the rightmost such leaf is on top of the stack. We have the usual operations Pop, Push, Top, InitStack and

EmptyStack that can be carried out in time O(1).

Below a precise description of the algorithm is given.

```
0  begin
1    Sort(V);
2    AddRightmost(p_{-∞}, p_∞);
3    for i:=0 to n − 1 do AddRightmost(p_i, p_{-∞}) end;
4    InitStack; Push(p_0);
5    while not EmptyStack do
6      p:=Pop;
7      p_r:=RightBrother(p);
8      q:=Father(p);
9      if q ≠ p_{-∞} then Handle(p, q) end;
10     z:=LeftBrother(q);
11     Remove(p);
12     if z=nil or not LeftTurn(p,z,Father(z)) then
13       AddLeftOf(p,q)
14     else
15       while RightmostSon(z)≠ nil
16           and LeftTurn(p,RightmostSon(z),z) do
17         z:=RightmostSon(z)
18       end;
19       AddRightmost(p,z);
20       if z=Top then z:=Pop end
21     end;
22     if LeftBrother(p)=nil and Father(p) ≠ p_∞ then
23       Push(p)
24     end;
25     if p_r ≠ nil then Push(p_r) end
26   end
27 end;
```

The algorithm works as follows. In line 1 the set of points is sorted by decreasing $x$-coordinate. In lines 2-3 the initial tree is built. In line 4 the stack is initialized and the leftmost leaf (i.e., the rightmost point) is pushed on it. Lines 5-26 form the main loop that moves the nodes in the tree. $p$ becomes the point to move. We report the edge in line 9. Now there are two cases. The first case occurs when $p$ has to be added as a son to the father of his father. This occurs in line 13. Otherwise, in lines 15-18 we walk along the chain to find the tangent point from $p$. $p$ is added to this new father in line 19. Line 20 is a test you easily forget. When the new father of $p$ is on the stack it should be removed from it because it is no longer a leaf. Finally we adapt the stack. If $p$ does not have a left brother $p$ is added in line 23. When $p$ had a right brother this brother is added in line 25.

The correctness of the algorithm follows from above lemmas.

**Theorem 2.3** *Given a set of $n$ points in the plane, in time $O(n^2)$ and storage $O(n)$ we can for each point $p$ sort the other points by angle around $p$.*

**Proof.** The bound on the amount of storage is immediately clear. We have to prove that the method takes time $O(n^2)$. Except for the **while** loop in lines 15-18 all other steps are performed only $\binom{n}{2}+n$ times. This follows from the fact that every time the main loop is performed an edge will be reported, except when the edge runs to $p_{-\infty}$ which happens $n$ times. The key observation is that for every step of the **while** loop a point that was to the left of $p$ in the tree will become on the path towards $p$. It can easily be seen that points that lie on the path toward $p$ or to the right of $p$ can never move back to the left of $p$. Hence, the **while** loop is executed in total $\Theta(n^2)$ times. (In fact, for each line of the algorithm one can determine the exact number of times it is executed. As a result, the runtime of the algorithm is basically the same, independent of the set of points.) □

# 3 Visibility graphs.

We will now use the method described in the previous section to determine the visibility graph of a set of non-intersecting line segments. Let $S$ be the set of line segments. We will perform the above procedure on the endpoints of the line segments. For each point $p$ we maintain VIS($p$) which is the line segment $p$ sees in the direction just before the direction of its current outgoing edge in the tree. If there is no such line segment, VIS($p$)=**nil**.

To initialize the structure we determine for each point $p$ the line segment immediately below it. We can do this in $O(n \log n)$ time using a scanline technique. (Since the method will require $O(n^2)$ time we can as well check each point with each line segment and do the initialization in $\Theta(n^2)$ time.)

Next we run the algorithm described in the previous section. We only have to specify the routine Handle. Clearly we should report an edge $\overline{pq}$ only if $p$ can see $q$, i.e., if $q$ lies nearer to $p$ than VIS($p$) or $q$ is an endpoint of VIS($p$). Moreover, we might have to adapt VIS($p$).

For line segments we need the following operations that can be performed in O(1) time:

Segment($p$):
: Returns the segment $p$ is an endpoint of.

Other($p$):
: Returns the other endpoint of the segment of $p$.

Before($p, q, e$):
: **true** if $q$ lies nearer to $p$ than segment $e$.

Report($p, q$):
: Reports $pq$ as an edge of $G_S$.

The routine Handle is described below.

```
0 procedure Handle(p, q);
1 if q=Other(p) then
2   Report(p, q)
3 elsif Segment(q)=VIS(p) then
4   VIS(p):=VIS(q);
5   Report(p, q)
6 elsif Before(p,q,VIS(p)) then
7   VIS(p):=Segment(q);
8   Report(p, q)
9 end;
```

(It should be noted that the procedure is only correct when the points are in general position. If this is not the case some minor changes have to be made in both the procedure Handle and the routine in the previous section.) The correctness of the method follows from [10].

**Theorem 3.1** *Given a set $S$ of $n$ non-intersecting line segments in the plane, its visibility graph $G_S$ can be constructed in time $O(n^2)$ and storage $O(n)$.*

# 4 Sparse visibility graphs.

The method in the previous section works well when the number of edges is large. But in many cases the number of edges is $o(n^2)$. For such sparse visibility graphs we will devise a new method. The idea of the method is the same as in the previous sections. We again construct a tree on the endpoints of the line segments and maintain this tree while rotating it from pointing downwards to pointing upwards. The difference is that at any moment we will take care that the tree only contains edges of $G_S$. To achieve this we have to process the tree in a bit different way. We will again assume that no three points lie on a line. In fact we even require the more stronger restriction that no two lines connecting endpoints are parallel. These restriction can easily be removed without complicating the algorithm.

We first define the notion of visibility graph following a direction. Let $S' = S \cup \{p_\infty\}$. (We don't need the point $p_{-\infty}$.) Let $d$ be a direction between $-\pi/2$ and $\pi/2$.

**Definition 4.1** *The* visibility graph following direction $d$, *denoted as $G_d$ is a subgraph of $G_{S'}$ with the same node set, such that each node $p$, except for $p_\infty$, has only the one outgoing edge with smallest slope $\geq d$.*

See figure 2 for an example. $G_d$ is the tree we will maintain while rotating the direction from $-\pi/2$ to $\pi/2$. It is immediately clear that any edge of $G_S$ will be in $G_d$ for some direction $d$. We will report an edge at the moment we delete it from $G_d$ (during the rotation). In this way each edge of $G_S$ is reported exactly once. We will give some properties of $G_d$. For proofs see the full paper.

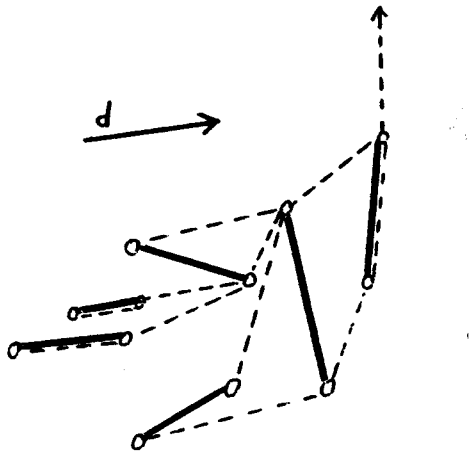


Figure 2: A visibility graph following direction $d$.

**Lemma 4.1** *No two edges in $G_d$ intersect.*

The edges in $G_d$ will be divided into two groups, arriving and departing edges.

**Definition 4.2** *Let $e = \overline{pq}$ be an edge of $G_d$. We call $e$* arriving *iff Other(q) does not lie to the right of $e$. We call $e$* departing *iff Other(q) does not lie to the left of $e$.*

Informally, an edge is arriving when it just arrived at a line segment in the rotational sweep. It is departing when it will depart from a line segment. Note that edges along line segments are both arriving and departing. For each point we keep the incoming edges sorted by slope and, hence, we can speak about the first arriving or departing edge (with smallest slope) and the last arriving or departing edge. Again we define chains of edges.

**Definition 4.3** *A* chain *is a maximal sequence $e_0, e_1, ..., e_i$ of edges such that each $e_{j+1}$ starts at the point where $e_j$ ends and all edges except for $e_i$ are last arriving edges.*

**Lemma 4.2**
1. *Each edge is part of exactly one chain.*
2. *The edges of a chain appear sorted by increasing slope.*
3. *The edge with smallest slope is the first edge of some chain.*

**Definition 4.4** *Let $e = \overline{pq}$ be an edge of $G_d$.* Previous($e$) *is the last edge ending at $q$ of the same type as $e$ (arriving or departing) and slope $<$ slope of $e$.* Visibility($p$) *is the current visibility of $p$, i.e., the point $p$ is pointing to.*

As we said before we will maintain $G_d$ will $d$ moves from $-\pi/2$ and $\pi/2$. Clearly, $G_d$ only changes when $d$ is the slope of the edge in $G_d$ with smallest slope. There is only one such edge (because of our assumptions). So we have to remove this edge and replace it by a new edge. The following lemmas show that there are only a limited number of candidates for this new edge. $\overline{pq}$ is the edge with minimal slope $d$ that will be replaced by a new edge.

**Lemma 4.3** *If $\overline{pq}$ is not a departing edge then the next visible point from $p$ is on the chain ending at the last departing edge at Other($q$) or, if there is no such chain, it is Other($q$).*

**Proof.** (Sketch) Let $z$=Other($q$). Consider the triangle $pqz$. When this triangle in empty $p$ can see $z$ and, hence, $z$ is the new visibility for $p$. If there is a chain $z$ lies on this chain. So if the triangle is empty the lemma holds. Otherwise, let $r_0$ be the new visibility for $p$. $r_0$ must lie inside the triangle $pqz$. Consider $r_1$=Visibility($r_0$). $r_1$ must lie inside the triangle or is equal to $z$. This follows from the fact that $\overline{r_0 r_1}$ cannot intersect $\overline{pq}$ or $\overline{qz}$ and if it would intersect $\overline{pz}$ its slope is larger than the slope of $\overline{r_0 z}$ which is impossible. It can easily be seen that either $\overline{r_0 r_1}$ is a last arriving edge or $r_1 = z$. If $p_1 = z$ we are done, otherwise we repeat the argument with $r_2$=Visibility($r_1$), etc. □

When the new visibility lies on the chain it is the point $r$ such that $\overline{pr}$ is a tangent to the chain. In a similar way we can proof:

**Lemma 4.4** *If $\overline{pq}$ is a departing edge then the next visible point from $p$ is on the chain ending at Previous(qVisibility($q$)) or, if there is no such chain, it is Visibility($q$).*

Now that we know where to find the new visibility, the only thing that remains is to show how chains change. As remarked above, when we remove an edge, it is the first edge of a chain. So this is a simple operation. When we add the new edge, either a new chain is created or the chain we search on is split into two chains and the new edge is added as first edge to one of them.

For the implementation we need three data structures:

- $G_d$ stores the visibility graph following the current direction $d$. It supports the following operations:

  Departing($e$):
  Returns whether $e$ is a departing edge.
  LastDeparting($p$):
  The departing edge of $p$ with largest slope.
  Previous($e$):
  The previous edge at the endpoint of $e$.
  Visibility($p$):
  The current visibility of point $p$.
  Other($p$):
  The other endpoint of the segment of $p$.
  Remove($e$):
  Removes $e$ from the structure.
  Add($e$):
  Adds $e$ to the structure.

  All these operation can be carried out in time O(1) because when we add an edge we have a pointer to its neighbor.

- A priority queue $Q$ that stores all the edges currently in $G_d$. It supports the following operations:

  DeleteMin:
  Returns and removes edge with smallest slope.
  Insert($e$):
  Inserts edge $e$.

  The operations can be carried out in O($\log n$) time.

- CHAIN stores the different chains. It supports the following operations.

  Chain($e$):
  The chain of which $e$ is the last edge.
  DeleteInChain($e$):
  Deletes $e$ from the chain starting at $e$.
  InsertInChain($ch, e$):
  Adds $e$ as first edge to chain $ch$.
  CreateChain($e$):
  Creates a new chain consisting of $e$ only.
  SearchInChain($ch, p$):
  Returns the tangent point from $p$ to $ch$.
  SplitChain($ch, p$):
  Splits $ch$ into two chains at point $p$.

  All operations can easily be performed in time O($\log n$) when implementing each chain as a concatenable queue. But for sake of simplicity of the method another approach can be taken. Each chain is stored as a simple linked list with extra

pointers between the last and first element. All operation, except for SearchInChain can now be carried out in time O(1). To perform SearchInChain we start at the first element of the chain and walk along the list until we find the tangent point. (Note that we walk along the chain in the reversed order as in section 2.) This might take a lot of work. But note that all points we pass on the chain are visible from $p$. We charge the work to these visibilities. In the full paper it will be shown, using Davenport-Schinzel sequences[4], that in this way each visibility is charged at most $O(\alpha(n))$ work.

Below a precise description of the algorithm is given.

```
0  begin
1    Initialise for d = -π/2;
2    loop
3      pq:=DeleteMin;
4      if Slope(pq)=π/2 then Ready end;
5      if not Departing(pq) then
6        z:=Other(q); e:=LastDeparting(z)
7      else
8        z:=Visibility(q); e:=Previous(qz)
9      end;
10     DeleteInChain(pq); Remove(pq); Report(p,q);
11     if e=nil then
12       CreateChain(pz)
13     else
14       z:=SearchInChain(Chain(e),p);
15       Splitchain(Chain(e),z);
16       InsertInChain(Chain(e),pz)
17     end;
18     Add(pz); Insert(pz)
19   end
20 end;
```

The algorithm works as follows. In line 1 the structure is initialized for the starting direction. In fact, this is probably the most complicated step of the algorithm. It can be performed in time $O(n \log n)$ by moving a scanline from right to left over the set of line segments, determining for each point the nearest edge below it and next performing one step of the algorithm to determine the correct touching point. See the full paper for details. The loop in lines 2-19 does all the work. It takes the edge with smallest slope (in line 3) checks whether we are done (in line 4) and, if not ready, removes it and replaces it with the new visibility for the starting point $p$. Lines 5-9 distinguish the two different cases corresponding to Lemmas 4.3 and 4.4. Line 10 removes the edge (and reports it). Next it is checked whether a chain exitsts that has to be searched and if so, this is done in lines 14-16. Finally, in line 18 the new edge is added to the structures.

The correctness of the method follows from the lemmas proven above.

**Theorem 4.5** *Given a set $S$ of $n$ non-intersecting line segments in the plane, its visibility graph $G_S$ can be constructed in time $O(m \log n)$ and storage $O(n)$ where $m = |G_S|$.*

**Proof.** Each time the loop is performed an edge is reported. Except for the call to SearchInChain all operation require $O(\log n)$ time. As indicated above, the calls to SearchInChain together take no more than $O(m\alpha(n))$ time. □

# 5 Conclusions and extensions.

In this paper we presented two algorithms for computing the visibility graph of a set of non-intersecting line segments in the plane. Both methods use only linear working storage and are easy to implement.

The first method is based on a solution to the problem of computing for each point in the set the other points sorted around it by angle. The method runs in optimal $O(n^2)$ time and $O(n)$ storage. It is an alternative for the topologically sweeping technique of [2]. Hence, it can also be used to solve many of the other problems mentioned in [2]. The method is based on a rotational sweep of a tree on the set of points.

The second method is based on a similar idea. But in this case care is taken that only edges of the visibility graph will be considered and, hence, all work done can be charged to edges. The method runs in time $O(m \log n)$ where $m$ is the size of the visibility graph. This time bound is only caused by the fact that we need to maintain a priority queue on the edges in the tree. At the moment we are investigating whether the use of the priority queue can be avoided which would improve the bound to $O(m\alpha(n) + n \log n)$.

A number of open problems do remain. The main question is to improve the time bound to $O(m + n \log n)$ requiring only $O(n)$ storage and without making the method more complicated. Secondly, one could consider visibility graphs of more complicated sets of objects, like polygons, circles or intersecting line segments. Finally, we think that the idea of rotation trees might have other interesting applications.

# References

[1] Asano, T., T. Asano, L. Guibas, J. Hershberger

and H. Imai, Visibility of disjoint polygons, *Algorithmica* 1 (1986), 49-63.

[2] Edelsbrunner, H., and L. Guibas, Topologically sweeping in an arrangement, *Proc. 18th Symp. on Theory of Computing,* 1986, 389-403.

[3] Ghosh, S.K. and D.M. Mount, An output sensitive algorithm for computing visibility graphs, *Proc. 28th Symp. on Foundations of Computer Science,* Los Angeles, 1987, 11-19.

[4] Hart, S., and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica* 6 (1986), 151-177.

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

[6] Lee, D.T., *Proximity and reachability in the plane,* Ph.D. thesis, Dept. of Electrical Engeneering, University of Illinois at Urbana-Champaign, 1979.

[7] Lozano-Perez, T. and M.A. Wesley, An algorithm for planning collision-free paths among polyhedral obstacles, *Comm. ACM* 22 (1979), 560-570.

[8] Overmars, M.H. and E. Welzl, *Construction of sparse visibility graphs,* Techn. Rep. RUU-CS-87-9, Dept. of Computer Science, University of Utrecht, 1987.

[9] Sharir, M., and A. Schorr, On shortest paths in polyhedral spaces, *Proc. 16th Symp. on Theory of Computing,* 1984, 144-153.

[10] Welzl, E., Constructing the visibility graph for $n$ line segments in $O(n^2)$ time, *Inform. Proc. Let.* 20 (1985), 167-171.