A Robust and Efficient Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem

Ulrike Bartuschka
Fachbereich Mathematik und Informatik, Universität Halle
Halle, Germany
e-mail: ulrike@informatik.uni-halle.de

Kurt Mehlhorn
Max-Planck-Institut für Informatik
Saarbrücken, Germany
e-mail: mehlhorn@mpi-sb.mpg.de

and

Stefan Näher
Fachbereich Mathematik und Informatik, Universität Halle
Halle, Germany
e-mail: naeher@informatik.uni-halle.de

ABSTRACT

We describe a robust and efficient implementation of the Bentley-Ottmann sweep line algorithm [1] based on the LEDA platform of combinatorial and geometric computing [9, 8]. The program computes the planar graph G induced by a set S of straight line segments. The nodes of G are all endpoints and all proper intersection points of segments in S. The edges of G are the maximal relatively open subsegments of segments in S that contain no node of G. The algorithm runs in time $O((n+s)\log n)$ where n is the number of segments and s is the size of the graph G. The implementation makes use of the basic geometric types *rat_point* and *rat_segment* of LEDA. These types realize two-dimensional points and segments with rational coordinates; they use exact arithmetic for the realization of all geometric primitives. The overhead of exact arithmetic is reduced by means of a floating point filter (cf. [4, 7]). The source of the full paper including the complete C++code is available from http://www.informatik.uni-halle.de/~naeher/geo.html.

1. Introduction

Let S be a set of n straight-line segments in the plane and let $G(S)$ be the graph induced by S. The vertices of $G(S)$ are all endpoints of segments in S and all proper intersection points between segments in S. The edges of G are the maximal relatively open and connected subsets of segments in S that contain no vertex of $G(S)$. Figure 1 shows an example. Note that the graph $G(S)$ contains parallel edges if S contains segments that overlap.

Bentley and Ottmann [1] have shown how to compute $G(S)$ in time $O((n+k)\log n)$ where k is the number of pairs of intersecting segments in S. The algorithm is based on the plane-sweep paradigm. We refer the reader to [5, section VIII.4], [10, section 7.2.3], and [3, section 35.2] for a discussion of the plane sweep paradigm.

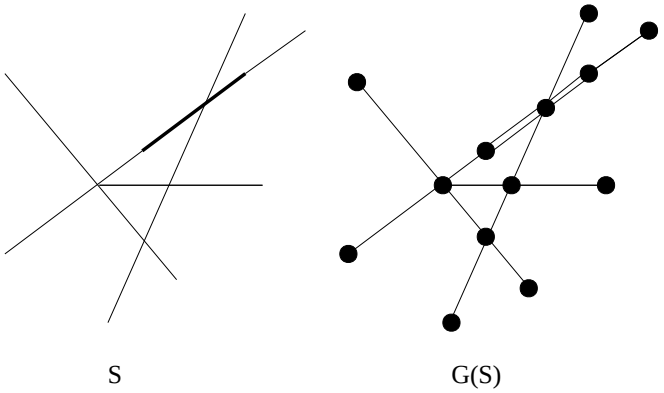

Figure 1: A set S of segments and the induced planar graph.

In this paper we describe an implementation of the Bentley-Ottmann algorithm. More precisely, we define a function

sweep_segments(*list*<*rat_segment*> L, *GRAPH*<*rat_point*, *rat_segment*>& G, *bool embed*)

that takes a list L of *rat_segment*s, i.e., segments whose vertices have rational coordinates, and computes the graph G induced by it. For each vertex v of G it also computes its position in the plane, a *rat_point* (= a point with rational coordinates), and for each edge e of G it computes the *rat_segment* containing it.

If *embed* = *true* the algorithm turns G into a planar map, i.e., G is made bidirected and the adjacency lists are sorted according to the geometric embedding in clockwise order.

We want to stress that the implementation makes no assumptions about the input, in particular, segments may have length zero, may be vertical or may overlap, several segments may intersect in the same point, endpoints of segments may lie in the interior of other segments, ...

We have achieved this generality by following two principles.

1. We treat degeneracies as first class citizens and not as an afterthought [2]. In particular, we reformulated the plane-sweep algorithm so that it can handle all geometric situations. The details will be given in section 2. The reformulation makes the description of the algorithm shorter since we do not distinguish between three kinds of events but have only one kind of event and it also makes the algorithm faster. The algorithm now runs in time $O((n+s)\log n)$ where s is the size of G. Note that $s \leq 3(n+k)$ and that k can be as large as s^2. The only previous algorithm that could handle all degeneracies is due to Myers [6]. Its expected running time for random segments is $O(n \log n + k)$ and its worst case running time is $O((n+k)\log n)$.

2. We evaluate all geometric tests exactly. This is achieved by using the LEDA types *rat_point* and *rat_segment*. These types use arbitrary precision integer arithmetic for all geometric computations. So all tests are computed exactly. Of course, there is an overhead of arbitrary precision integer arithmetic. In order to keep the overhead small LEDA uses a floating point filter following the suggestion of Fortune and van Wyk [4], i.e., all tests are first performed using floating point arithmetic and only if the result of the floating point computation is inconclusive the costly exact computation is performed (cf. [7] for details).

2. The Algorithm

In the sweep-line paradigm a vertical line is moved from left to right across the plane and the output (here the graph $G(S)$) is constructed incrementally as it evolves behind the sweep line. One

maintains two data structures to keep the construction going: The so-called *Y-structure* contains the intersection of the sweep line with the scene (here the set S of line segments) and the so-called *X-structure* contains the events where the sweep has to be stopped in order to add to the output or to update the X- or Y-structure. In the line segment intersection problem an event occurs when the sweep line hits an endpoint of some segment or an intersection point. When an event occurs, some nodes and edges are added to the graph $G(S)$, the Y-structure is updated, and maybe some more events are generated. When the input is in general position (no three lines intersecting in a common point, no endpoint lying on a segment, no two endpoints or intersections having the same x-coordinate, no vertical lines, no overlapping segments, ...) then at most one event can occur for each position of the sweep line and there are three clearly distiguishable types of events (left endpoint, right endpoint, intersection) with easily describable associated actions, cf. [5, section VII.8]. We want to place no restrictions on the input and therefore need to proceed slightly differently. We now describe the required changes.

We define the sweep line by a point $p\_sweep = (x\_sweep, y\_sweep)$. Let ϵ be a positive infinitesimal (readers not familiar with infinitesimals may think of ϵ as an arbitrarily small positive real). Consider the directed line L consisting of a vertical upward ray ending in point $(x\_sweep+\epsilon^2, y\_sweep+\epsilon)$ followed by a horizontal segment ending in $(x\_sweep-\epsilon^2, y\_sweep+\epsilon)$ followed by a vertical upward ray. We call L the *sweep line*. Note that no endpoint of any segment lies on L, that no two segments of S intersect L in the same point except if the segments overlap, and that no non-vertical segment of S intersects the horizontal part of L. All three properties follow from the fact that ϵ is arbitrarily small but positive. Figure 2 illustrates the definition of L and the main data structures used in the algorithm: The Y-structure, the X-structure, and the graph G.

The Y-structure contains all segments intersecting the sweep line L ordered as their intersections with L appear on the directed line L. Overlapping segments will be ordered by their *ID_numbers*.

The X-structure contains all endpoints that are to the right of the sweep line and also some intersection points between segments in the Y-structure. More precisely, for each pair of segments adjacent in the Y-structure their intersection point is contained in the X-structure (if it exists and is to the right of the sweep line). The X-structure may contain other intersection points. The graph G contains the part of $G(S)$ that is to the left of the sweep line.

Initially, the Y-structure and the graph G are empty and the X-structure contains all endpoints of all input segments. The events in the X-structure are then processed in left to right order. Events with the same x-coordinate are processed in bottom to top order.

Assume that we need to process an event at point p and that the X-structure and Y-structure reflect the situation for $p\_sweep = (p.x, p.y - 2\epsilon)$. Note that this is true initially, i.e., before the first event is removed from the X-structure. We now show how to establish the invariants for $p\_sweep = p$. We proceed in seven steps.

1. We add a node v at position p to our graph G.
2. We determine all segments in the Y-structure containing the point p. These segments form a possibly empty subsequence of the Y-structure.
3. For each segment in the subsequence we add an edge to the graph G.
4. We delete all segments ending in p from the Y-structure.
5. We update the order of the subsequence in the Y-structure. This amounts to moving the sweep line across the point p.
6. We insert all segments starting in p into the Y-structure.
7. We generate events for the segments in the Y-structure that become adjacent by the actions above and insert them into the X-structure.

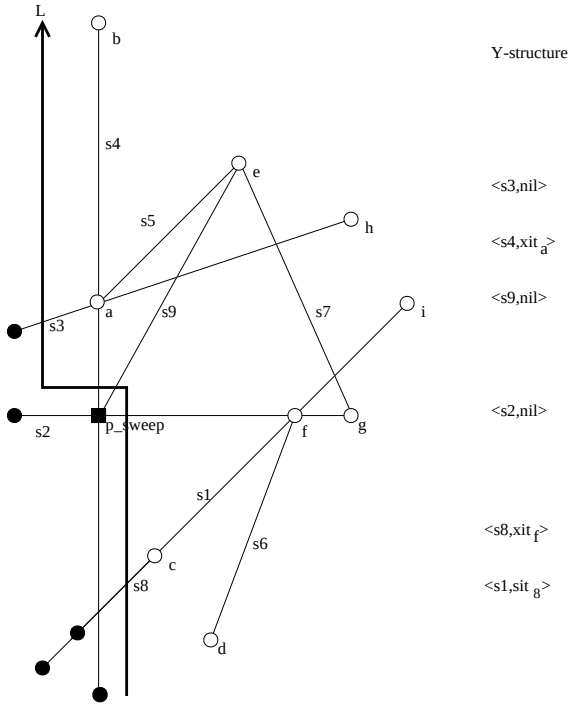

Figure 2: A scene of 9 segments. The segments s_1 and s_8 overlap. The Y-structure contains segments s_1, s_8, s_2, s_9, s_4, and s_3 and the X-structure contains points a, b, c, d, e, f, g, h, and i. An item in the X-structure containing point p is denoted xit_p and an item in the Y-structure containing segment s_i is denoted sit_i. The vertices of the graph G are shown as full circles.

This completes the description of how to process the event p. The invariants now hold for $p\_sweep = p$ and hence also for $p\_sweep = (p'.x, p'.y - 2\epsilon)$ where p' is the new first element of the X-structure.

3. The Implementation

The implementation follows the algorithm closely. Our program has the following structure.

⟨*⟩≡
⟨*geometric primitives*⟩

```
void sweep_segments(const list<rat_segment>& S,
                    GRAPH<rat_point, rat_segment>& G, bool embed)
{ ⟨local declarations⟩
  ⟨initialization⟩
  ⟨sweep⟩
}
```

We use sorted sequences, maps, priority queues, graphs, points and segments with rational coordinates, and integers of arbitrary precision from LEDA and have to include the corresponding header files. Let us briefly explain these types; for a detailed discussion we refer the reader to the

LEDA manual [9]. We also discuss the use of these types in our program.

The type *integer* realizes arbitrary precision integers.

The types *rat_point* and *rat_segment* realize points and segments in the plane with rational coordinates. A *rat_point* is specified by its homogeneous coordinates of type *integer*. If p is a *rat_point* then $p.X(\,)$, $p.Y(\,)$, and $p.W(\,)$ return its homogeneous coordinates. If x, y, and w are of type *integer* with $w \neq 0$ then $rat\_point(x, y)$ and $rat\_point(x, y, w)$ create the *rat_point* with homogeneous coordinates $(x, y, 1)$ and (x, y, w) respectively. Two points are equal (*operator*==) if they agree in their cartesian coordinates. A *rat_segment* is specified by its two endpoints; so if p and q are *rat_point*s then $rat\_segment(p, q)$ is the directed segment with source p and target q. If s is a *rat_segment* then *s.source*( ) and *s.target*( ) return the source and target of s respectively. LEDA provides a set of basic geometric primitives on *rat_points* and *rat_segments* (see [9] and [7] for details). In the sweep program we use the following primitives:

- *int compare*(*rat_point* p, *rat_point* q)
  defines the default linear order on *rat_point*s: it is the lexicographic ordering of the cartesian coordinates of the points.
- *int orientation*(*rat_point* p, *rat_point* q, *rat_point* r)
  computes the orientation of points p, q, and r in the plane i.e., the sign of the determinant

$$\begin{vmatrix} p_x & p_y & p_w \\ q_x & q_y & q_w \\ r_x & r_y & r_w \end{vmatrix}$$

- *int orientation*(*rat_segment* s, *rat_point* p)
  computes $orientation(s.source(\,), s.target(\,), p)$.
- *int cmp_slopes*(*rat_segment s1*, *rat_segment s2*)
  compares the slopes of *s1* and *s2*. If one of the segments is degenerate, i.e., has length zero, the result is zero. Otherwise, the result is the sign of $slope(s1) - slope(s2)$.
- *bool intersection_of_lines*(*rat_segment s1*, *rat_segment s2*, *rat_point*& p)
  returns *false* if segments *s1* and *s2* are parallel or one of them is degenerate. Otherwise, it computes the point of intersection of the two straight lines supporting the segments, assigns it to the third parameter q, and returns *true*.

Our program maintains its own set of segments which we call *internal segments* or simply segments. They are directed from left to right; vertical segments are directed upwards and are stored in list *internal*. We have a map <*rat_segment*, *rat_segment*> original that associates an input segment with every internal segment. More precisely, we have:

- For each internal segment s the segment $original[s]$ is an input segment; s is contained in $original[s]$ and the endpoints of s are endpoints of some input segments.
- For each input segment of non-zero length there is an internal segment with the same endpoints.

Note that the internal segments together with the zero length input segments define exactly the same graph as the input segments. This follows since every non-zero length input segment has an equivalent internal segment and since the additional internal segments are contained in input segments and introduce no new endpoints.

The type *sortseq*<K, I> realizes sorted sequences of pairs in $K \times I$; K is called the key type and I is called the information type of the sequence. K must be linearly ordered, i.e., the function *int compare*(*const* K&, *const* K&) must be defined for the type K and the relation $<$ on K defined

by $k_1 < k_2$ iff $compare(k_1, k_2) < 0$ is a linear order on K. An object of type *sortseq*<K, I> is a sequence of items (type *seq_item*) each containing a pair in $K \times I$. We use <k, i> to denote an item containing the pair (k, i) and call k the key and i the information of the item. The keys in a sorted sequence $< k_1, i_1 >, < k_2, i_2 >, \ldots, < k_m, i_m >$ form an increasing sequence, i.e., $k_l < k_{l+1}$ for $1 \leq l < m$.

Let S be a sorted sequence of type *sortseq*<K, I> and let k and i be of type K and I respectively. The operation $S.lookup(k)$ returns the item $it =< k, . >$ in S with key k if there is such an item and returns *nil* otherwise. If $S.lookup(k) == nil$ then $S.insert(k, i)$ adds a new item <k, i> to S and returns this item. If $S.lookup(k) == it$ then $S.insert(k, i)$ changes the information in the item it to i. If $it =< k, i >$ is an item of S then $S.key(it)$ and $S.inf(it)$ return k and i respectively and $S.succ(it)$ and $S.pred(it)$ return the successor and predecessor item of it respectively; the latter operations return *nil* if these items do not exist. The operation $S.min(\ )$ returns the first item of S, $S.empty(\ )$ returns true if S is empty and false otherwise. Finally, if *it1* and *it2* are items of S with *it1* before *it2* then $S.reverse\_items(it1, it2)$ reverses the subsequence of S starting at item *it1* and ending at item *it2*. The requirement for this operation is that the sequence S is sorted with respect to the current compare function after the operation.

In our implementation the X-structure has type *sortseq*<*rat_point*, *seq_item*> and the Y-structure has type *sortseq*<*rat_segment*, *seq_item*>. The Y-structure has one item for each segment intersecting the sweep line. The information field in the Y-structure is used for cross-links with the X-structure. We will come back to this below. The Y-structure is ordered according to the intersections of the segments with the directed sweep line L.

It is important to observe that the compare function for segments changes as the sweep progresses. What does it mean then that the keys of the items in a sorted sequence form an increasing sequence? LEDA requires that whenever a lookup or insert operation is applied to a sorted sequence the sequence must be sorted with respect to the current compare function. The other operations may be applied even if the sequence is not sorted.

In the example of Figure 2 the sweep line intersects the segments s_1, s_8, s_2, s_9, s_4, and s_3. The Y-structure therefore consists of six items, one each for segments s_1, s_8, s_2, s_9, s_4, and s_3.

The X-structure contains an item for each endpoint of an input segment that is to the right of the sweep line and for each intersection point between segments that are adjacent in the Y-structure and that intersect to the right of the sweep line. It may also contain intersection points between segments that are not adjacent in the Y-structure.* The points in the X-structure are ordered according to the lexicographic ordering of their cartesian coordinates. As mentioned above this is the default order on *rat_point*s.

In the example of Figure 2 the X-structures contains items for the endpoints b, c, d, e, g, h, i and for intersections a and f. Here, a (f) is the intersection of adjacent segments s_4 and s_3 (s_1 and s_2).

The informations associated with the items of both structures serve as cross-links between the two structures: the information associated with an item in the X-structure is either nil or an item in the Y-Structure; the information associated with an item in the Y-structure is either nil, or an item of either structure. The precise definition follows: Consider first an item $< s, it >$ in the Y-structure and let s' be the segment associated with the successor item it' in the Y-structure. If s and s' overlap then $it = it'$. If s and s' do not overlap and $s \cap s'$ exists and lies to the right of the sweep line then it is an item in the X-structure with key $s \cap s'$. In all other cases we have $it = nil$.

Consider next an item $< p, sit >$ in the X-structure. If $sit \neq nil$ then sit is an item in the Y-structure and the segment associated with it contains p. Moreover, if there is a pair of adjacent segments in the Y-structure that intersect in p then $sit \neq nil$. We may $sit \neq nil$ even if there is no pair of adjacent segments intersecting in p.

*Our X-structure may contain intersection points between segments that are non-adjacent in the Y-structure and that are not endpoint of any segment. These events can be removed from the X-structure in order to guarantee a linear size of this structure. This, however, complicates the code significantly. Since the size of the X-structure is always bounded by the size of the output graph we do not apply this method.

In our example, the Y-structure contains the items $< s_1, sit_8 >$, $< s_8, xit_f >$, $< s_2, nil >$,$< s_9, nil >$,$< s_4, xit_a >$ and $< s_3, nil >$ where sit_8 is the item of the Y-structure with associated segment s_8 and xit_a and xit_f are the items of the X-structure with associated points a and f respectively. Let's turn to the items of the X-structure next. All items except $< d, nil >$ point back to the Y-structure. If sit_i denotes the item $< s_i$,...>, $i \in \{1,2,9,4,3\}$, of the Y-structure then the items of the X-structure are $< a, sit_4 >$,$< b, sit_4 >$,$< c, sit_1 >$,$< d, nil >$,$< e, sit_9 >$,$< f, sit_1 >$,$< g, sit_2 >$,$< h, sit_3 >$ and $< i, sit_1 >$.

The graph G to be constructed has type *GRAPH*<*rat_point*, *rat_segment*>, i.e., it is a directed graph where a *rat_point*, respectively *rat_segment*, is associated with each node, respectively edge, of the graph. The graph G is the part of $G(S)$ that is left of the sweep line. The point associated with a vertex defines its position in the plane and the segment associated with an edge is an input segment containing the edge. We use two operations to extend the graph G. If p is a *rat_point* then *G.new_node*(p) adds a new node to G, associates p with the node, and returns the new node. If v and w are nodes of G and s is a *rat_segment* then *G.new_edge*(v, w, s) adds the edge (v, w) to G, associates s with the edge, and returns the new edge. In order to facilitate the addition of edges we maintain a map <*rat_segment*, *vertex*> *last_node*:. It gives for each segment in the Y-structure the rightmost vertex lying on the segment.

Finally, we have a priority queue *seg_queue* of type *p_queue*<*rat_point*, *rat_segment*>. The LEDA data type *p_queue*<*P*,*I*> realizes priority queues with priority type P and information type I. P must be linearly ordered. Similar by sorted sequences and dictionaries priority queue is an item-based data type. Every item (of type *pq_item*) stores a pair (p, i) from $P \times I$, p is called the priority and i is called the information of the item. LEDA offers the usual operations on priority queues (*insert*, *delete_min*, *find_min*). The priority queue *seg_queue* contains all internal segments that are ahead of the sweep line. The segments are ordered according to their left endpoint. In particular, the first segment in *seg_queue* is always the segment that is encountered next by the sweep line.

In the local declarations section of function *sweep* we introduce the variables of the data types mentioned above.

⟨*local declarations*⟩≡
```
sortseq<rat_point,seq_item>    X_structure;
sortseq<rat_segment,seq_item>  Y_structure;
map<rat_segment,node>          last_node(nil);
map<rat_segment,rat_segment>   original;
list<rat_segment>              internal;
p_queue<rat_point,rat_segment> seg_queue;
```

3.1. Initialization

Next we come to the initialization of the data structures. We clear the graph G, we compute a coordinate *Infinity* that is larger than the absolute value of the coordinates of all endpoints and that plays the role of ∞ in our program, we insert the endpoints of all input segments into the X-structure, and we create for each input segment of non-zero length an internal segment with the same endpoints, insert this segment into *seg_queue* and link the input segment to it (through map original), we create two sentinel segments at $-\infty$ and $+\infty$ respectively and insert them into the Y-structure, we put the sweep line at its initial position by setting *p_sweep* to $(-\infty, -\infty)$, and we add a stopper point with coordinates $(+\infty, +\infty)$ to *seg_queue*. The sentinels avoid special cases and thus simplify the code. Finally, we introduce a variable *next_seg* that always contains the first segment in *seg_queue*.

⟨*initialization*⟩≡
```
/* program code omitted */
```

3.2. Processing of Events

We now come to the heart of procedure sweep: processing events. Let *event* = <p, *sit*> be the first event in the X-structure and assume inductively that our data structure is correct for $p\_sweep = (p.x, p.y - 2\epsilon)$. Our goal is to change p_sweep to p, i.e., to move the sweep line across point p. As long as the X-structure is not empty we perform the following actions.

We first extract the next event point p_sweep from the X-structure. Then, we handle all segments passing through or ending at p_sweep. Finally, we insert all segments starting at p_sweep into the Y-structure, check for possible intersections between pairs of segments now adjacent in the Y-structure, and update the X-structure.

⟨*sweep*⟩≡
```
while (!X_structure.empty())
{ ⟨extract next event from the X-structure⟩
  ⟨handle passing and ending segments⟩
  ⟨insert starting segments⟩
  ⟨compute new intersections and update X-structure⟩
 }
```

Extracting the next event is easily done by assigning the minimal key in the X-structure to p_sweep, adding a vertex v with position p_sweep to the output graph G, and deleting the minimal item from the X-structure.

⟨*extract next event from the X-structure*⟩≡
```
seq_item event = X_structure.min();
p_sweep = X_structure.key(event);
node v = G.new_node(p_sweep);
X_structure.del_item(event);
```

Now we handle all segments passing through or ending in point p. How can we find them?

Recall that the current event is <p_sweep, *sit*>. Assume first that $sit = nil$. Then there is no pair of adjacent non-overlapping segments in the Y-structure that intersects in p_sweep and hence there is at most one bundle of overlapping segments in the Y-structure that contains p_sweep. We can decide whether there is such a bundle and determine its topmost segment by locating the zero-length segment (p_sweep, p_sweep) in the Y-structure.

Assume next that $sit \neq nil$ originally. Then we know that the segment associated with *sit* contains p_sweep. Now we have either $sit = nil$ and then no segment in the Y-structure contains p_sweep or $sit \neq nil$ and then the segment associated with *sit* contains p_sweep. In the latter case, taking *sit* as an entry point into the Y-structure, we determine all segments incident to p_sweep from the left or from below. Note that the corresponding items form a subsequence of the Y-structure. We also compute the predecessor (*sit_pred*), successor (*sit_succ*), first (*sit_first*) and last (*sit_last*) item of this *bundle* of items. Note further that all items in the bundle have information equal to the current event item *event* or (in the case of overlapping segments) to the successor item in the Y-structure, except for the last item whose information is either *nil* or a different item in the X-structure resulting from an intersection with *sit_succ*.

Starting at *sit* we first walk up until *sit_succ* is reached. Then we walk down to *sit_pred*. During the downward walk we also start to update the data structures. For every segment s in the bundle we add an edge to G connecting *last_node*[s] and v and label it with s. The new edge gets its direction from the original segment containing it. Also, if the segment ends at p_sweep then we delete it from the Y-structure and if the segment continues through p_sweep then we change the intersection information associated with it to *nil* and set *last_node* to v.

The identification of the subsequence of segments incident to p_sweep takes constant time per element of the sequence. Moreover, the constant is small since the test whether p is incident to a segment involves no geometric computation but only identity tests between items. The code

is particularly simple due to our sentinel segments: *sit* can never be the first or last item of the Y-structure.

All segments remaining in the bundle pass through node v and moving the sweep line through *p_sweep* changes the order of these segments in the Y-structure. Let s and s' be two segments of this bundle and assume that s is located over s' in the Y-structure. Moving the sweep line through *p_sweep* inverses their order iff s and s' do not overlap.

If the bundle is non-empty we update its order as follow : First reverse all subsequences of overlapping segments and then reverse the entire bundle.

⟨*handle passing and ending segments*⟩≡

```
seq_item sit = X_structure.inf(event);

if (sit == nil)
{ sit = Y_structure.locate_pred(rat_segment(p_sweep,p_sweep));
  if( orientation(Y_structure.key(sit), p_sweep) ) sit = nil;
 }
seq_item sit_succ  = nil;
seq_item sit_pred  = nil;
seq_item sit_first = nil;

if (sit != nil)
{ // walk up
  while ( Y_structure.inf(sit) == event ||
          Y_structure.inf(sit) == Y_structure.succ(sit) )
     sit = Y_structure.succ(sit);

  sit_succ = Y_structure.succ(sit);

  // walk down
  bool topmost;
  do { topmost = false;
       s = Y_structure.key(sit);

       if( identical( s.source(), original[s].source()) )
            G.new_edge( last_node[s], v, s );
        else
            G.new_edge( v, last_node[s], s );

       if ( identical(p_sweep,s.target()) )
         { //ending segment
           seq_item it = Y_structure.pred(sit);
           if ( Y_structure.inf(it) == sit )
           { topmost = true;
             Y_structure.change_inf(it, Y_structure.inf(sit));
            }
           Y_structure.del_item(sit);
           sit = it;
          }
       else //passing segment
         { if ( Y_structure.inf(sit) != Y_structure.succ(sit) )
              Y_structure.change_inf(sit,seq_item(nil));
           last_node[s] = v;
           sit = Y_structure.pred(sit);
          }
      } while ( Y_structure.inf(sit) == event || topmost ||
                Y_structure.inf(sit) == Y_structure.succ(sit) );

    sit_pred = sit;
    sit = Y_structure.succ(sit_pred);
```

```
    sit_first = sit;
    // reverse subsequences of overlapping segments  (if existing)
    while (sit != sit_succ  && Y_structure.succ(sit) != sit_succ)
    { seq_item sub_first = sit;
      seq_item sub_last  = sub_first;
      while (Y_structure.inf(sub_last) == Y_structure.succ(sub_last))
         sub_last = Y_structure.succ(sub_last);
      if (sub_last != sub_first)
         Y_structure.reverse_items(sub_first, sub_last);
      sit = Y_structure.succ(sub_first);
     }
    // reverse the entire bundle
    if ( Y_structure.succ(sit_pred) != sit_succ )
          Y_structure.reverse_items(Y_structure.succ(sit_pred),
                                    Y_structure.pred(sit_succ));
}
```

3.3. Insertion of Starting Segments

The last step in handling the event point *p_sweep* is to insert all segments starting at *p_sweep* into the Y-structure and to test the new pairs of adjacent items (*sit_pred*,...) and (...,*sit_succ*) for possible intersections. If there were no segments passing through or ending in *p_sweep* then the items *sit_succ* and *sit_pred* still have the value *nil* and we have to compute them now.

We use the priority queue *seg_queue* to find the segments to be inserted. As long as the first segment in *seg_queue* starts at *p_sweep*, i.e. *next_seg.source*( ) is identical to *p_sweep*, we remove it from the queue and locate it in the Y-structure.

We insert *next_seg* into the Y-structure; this will add an item to the Y-structure. We associate this item with the right endpoint of *next_seg* in the X-structure; note that the point is already there but it does not have its link to the Y-structure yet. We also set *last_node*[*s*] to *v* and if *sit_succ* and *sit_pred* are still undefined we set them to the successor and predecessor of the new item respectively.

The new item of the Y-structure has to be associated with its successor item if the corresponding segments overlap. Analogous its predessor item has to be associated with the new item if the corresponding segments overlap.

⟨*insert starting segments*⟩≡

```
    while ( identical(p_sweep,next_seg.source()) )
      { seq_item s_sit = Y_structure.locate(next_seg);
        seq_item p_sit = Y_structure.pred(s_sit);
        sit = Y_structure.insert_at(s_sit, next_seg, seq_item(nil));
        X_structure.insert(next_seg.end(), sit);
        last_node[next_seg] = v;
        if ( sit_succ == nil )
          { sit_succ = Y_structure.succ(sit);
            sit_pred = Y_structure.pred(sit);
            sit_first = sit_succ;
          }
        s  = Y_structure.key(s_sit);
        if( !orientation(s, next_seg.start()) &&
            !orientation(s, next_seg.end() ) )
          Y_structure.change_inf(sit, s_sit);
```

```
    s = Y_structure.key(p_sit);
    if( !orientation(s, next_seg.start()) &&
        !orientation(s, next_seg.end() ) )
      Y_structure.change_inf(p_sit, sit);

    seg_queue.del_min();
    next_seg = seg_queue.inf(seg_queue.find_min());
  }
```

3.4. Computing new Intersections

If *sit_pred* still has value *nil* then *p_sweep* is an isolated point and we are done. Otherwise we compute possible intersections between *sit_succ* and its predecessor and between *sit_pred* and its successor and update the X-structure.

⟨*compute new intersections and update X-structure*⟩≡

```
if (sit_pred != nil)
{ compute_intersection(X_structure, Y_structure, sit_pred);
  sit = Y_structure.pred(sit_succ);
  if (sit != sit_pred)
    compute_intersection(X_structure, Y_structure, sit);
 }
```

3.5. Geometric Primitives

It remains to define the geometric primitives used in the implementation. We need two:

- a compare-function for segments which given two segments intersecting the sweep line L determines the order of the intersections on L. It defines the linear order used in the Y-structure.
- a function *compute_intersection* that decides whether two segments intersect and if so whether the intersection is to the right of the sweep line. If both tests are positive it also makes the required changes to the X- and Y-structure.

⟨*geometric primitives*⟩≡

```
/* program code omitted */
```

4. Experiments and Efficiency

We performed tests on three kinds of test data, namely random, difficult, and highly degenerate inputs, and with three different implementations of points and segments (floating point, long integer arithmetic with floating point filter, long integer arithmetic without floating point filter).

Note that it is very easy to change our sweep program to use different implementations of the basic geometric objects. To switch to floating point arithmetic just replace types *rat_point* and *rat_segment* by *point* and *segment*, respectively. The types *point* and *segment* support the same set of basic geometric primitives implemented, however, by double floating point arithmetic. The floating point filter in *rat_point* and *rat_segment* can be switched off by setting the flags *rat_point*::*use_filter* and *rat_segment*::*use_filter* to false. We now describe the three kinds of test data.

The experiments inidicate that the floating point filter reduces the running time of the program by a factor between 2 and 3. More experiments and a detailed analysis of the floating point filter used in the implementation of the LEDA data types *rat_point* and *rat_segment* are described in [7].

5. Conclusion

We have given an implementation of the Bentley-Ottmann plane sweep algorithm for line segment intersection. The implementation is complete and reliable in the sense that it will work for all input instances. It is asympotically more efficient than previous algorithms for the same task; its running time depends on the number of vertices in the intersection graph and not on the number of pairs of intersecting segments. It also achieves a low constant factor in its running time by means of a floating point filter. The use of LEDA makes the implementation of the algorithm short and elegant.

References

[1] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[2] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms*, pp. 16–23, 1994.

[3] T.H. Cormen and C.E. Leiserson and R.L. Rivest. Introduction to Algorithms. MIT Press/McGraw-Hill Book Company, 1990.

[4] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th ACM Symp. on Computational Geometry*, pp. 163–172, 1993.

[5] K. Mehlhorn. Data Structures and Efficient Algorithms. Springer Publishing Company, 1984.

[6] E. Myers. An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem. *SIAM J. Comput.*, pp. 625 –636, 1985.

[7] K. Mehlhorn and S. Näher. The Implementation of Geometric Algorithms. 13th World Computer Congress IFIP 94, Elsevier Science B.V., Vol. 1, pp. 223–231, 1994.

[8] K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. *CACM* Vol. 38, No. 1, pp. 96–102, 1995.

[9] K. Mehlhorn, C. Uhrig, S. Näher. The LEDA User Manual (Version 3.5), Technical Report. Max-Planck-Institut für Informatik, 1997. online: http://www.mpi-sb.mpg.de/LEDA/leda.html

[10] F. Preparata and M.I. Shamos. Computational Geometry: An Introduction. Springer Publishing Company, 1985

[11] Ch. Yap and Th. Dubé. The exact computation paradigm. In *Computing in Euclidian Geometry*, World Scientific Press, 1994. (To appear, 2nd edition).