Geometric data structures for windowing queries

Jan Lönnberg
jlonnber@cs.hut.fi

Seminar presentation 16/3/2005
T-106.850 Geometric algorithms
Department of Computer Science and Engineering
Helsinki University of Technology

Abstract

Finding the contents of a specified region of a plane or space is an important part of many systems that process spatial data. In many cases, the contents of the region are line segments. In this report, some data structures are presented that enable fast enumeration of the line segments in a specified axis-parallel rectangular area. These data structures are *interval trees*, *priority search trees* and *segment trees*.

1 Introduction

Geometric data plays an important role in many applications that model the real world or objects in it. In such applications, it is often important to be able to quickly extract the information that describes a certain region of space. For example, a flight simulator needs to be able to determine which buildings, roads and terrain features are visible in order to generate a view of the surroundings. A CAD/CAM system must be able to find the visible parts in a complex design quickly.

Two-dimensional problems of this type also abound. Navigation systems need to be able to show maps at different levels of magnification and detail. Printed circuit boards contain a multitude of traces and components.

The most common version of this problem is to find the contents of a rectangular (or in three dimensions, cuboid) window; this is called a *windowing query*. Windowing queries are typically performed on line segments, polygons and curves. In this report, I will describe three data structures that can be used when performing windowing queries on certain types of line segment data: *interval trees*, *priority search trees* and *segment trees*.

This report is primarily based on the presentation of these data structures in [2].

2 Windowing for axis-parallel segments

Printed circuit boards usually contain traces that are essentially line segments that are either parallel to one of the sides of the (rectangular) board or at a 45 degree angle to the sides. This case can be simplified further by assuming that every line segment is parallel to one of the sides of the board. By aligning the x and y axes to the sides of the board, the segments become *axis-parallel*; they are parallel to either the x axis or the y axis.

The problem, then, is the following:

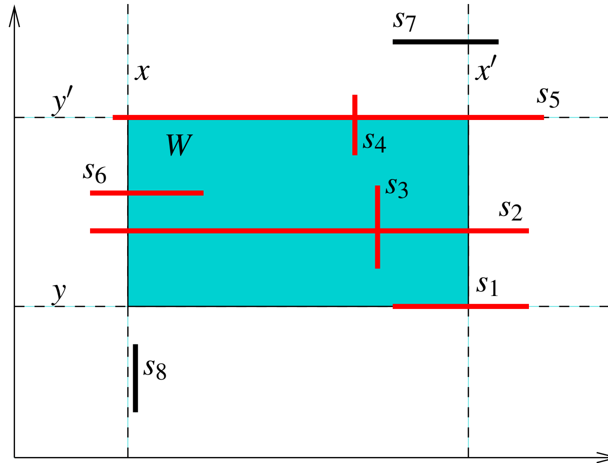

Figure 1: An example axis-parallel query

Input A set S of axis-parallel line segments, where $n = |S|$ and a query window $W = [x, x'] \times [y, y']$.

Output The set $S_W \subseteq S$ of line segments that intersect W; in other words, $s \in S_W \Leftrightarrow s \in S \wedge s \cap W \neq \emptyset$.

An example of a query of this type is shown in Figure 1. The segments $s_1, \ldots, s_8$ form S. The segments highlighted in red ($s_1, \ldots, s_6$) form S_W.

2.1 Problem analysis

If a line segment s intersects a query window W, one of the following is true:

1. Both the endpoints of s lie in W, which means that s lies entirely within W (for example, s_3 in Figure 1).

2. Exactly one of the endpoints of s lies within W, which means that s intersects the boundary of W once (e.g. s_4) or contains part of an edge of W (e.g. s_1).

3. Neither of the endpoints of s is within W, in which case s intersects the boundary of W twice (e.g. s_2) or contains an edge of W (e.g. s_5).

Testing for cases 1 and 2 is a matter of finding the k endpoints (of segments in S) in W. This can be done in $O(\log n + k)$ time and $O(n \log n / \log\log n)$ storage using a two-dimensional range tree [3].

Case 3 is harder. In this case s and W intersect, but neither of the endpoints of s is in W. Assume s is horizontal ($s = [x_1, x_2] \times [y_s, y_s]$) (the case where s is vertical is similar, but with the x and y axes swapped). If both endpoints of s are to the left or right of W, s and W obviously do not intersect. Likewise, they obviously do not intersect if $y_s < y$ or $y_s > y'$ (e.g. s_7). As neither endpoint is in W, $x_1 \leq x$ and $x_2 \geq x' \geq x$. Conversely, if $x_1 \leq x \leq x_2$ and $y \leq y_s \leq y'$, s and W intersect at point (x, y_s) on the left boundary of W.

In other words, one can find the horizontal segments of case 3 by finding the segments for which:

1. $x_1 \leq x \leq x_2$.

2. $y \leq y_s \leq y'$.

Vertical segments can, of course, be handled similarly, by switching the roles of the axes.

2.2 Interval trees

Interval trees (proposed independently by Edelsbrunner [5] and McCreight [7]) can be used to find segments that satisfy requirement 1. Interval trees are used to answer queries of the following type:

Input A set $I = \{[x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n]\}$ of closed real intervals and a query value q.

Output The set I_q such that $i \in I_q \Leftrightarrow i \in I \wedge q \in i$.

This type of query is known as a *stabbing query*.

The interval tree T corresponding to (non-empty) I is a binary tree such that:

- m_T is the median[1] of the $2n$ interval endpoints.
- The left subtree L_T of the tree is the interval tree corresponding to the set of intervals in I for which both endpoints are less than m_T.
- The right subtree R_T of the tree is the interval tree corresponding to the set of intervals in I for which both endpoints are greater than m_T.
- The root node of the tree contains the set of intervals $I'_T \subseteq I$ that contain m_T as a list sorted in ascending order on the left endpoint and a list sorted in descending order on the right endpoint.

The interval tree of $\emptyset$ is an empty leaf.

Constructing T from I can be done by applying the above rules recursively in $O(n \log n)$ time. The resulting tree has $O(\log n)$ depth and uses $O(n)$ storage (each interval appears in one node in the tree, and each internal node contains at least one interval). The query can be performed using the following algorithm:

`QueryIntervalTree`(T, q):
if T is a leaf **return** $\emptyset$
if $q < m_T$
  return $\{[a,b] \in I'_T : a \leq q\} \cup$ `QueryIntervalTree`(L_T, q)
else
  return $\{[a,b] \in I'_T : b \geq q\} \cup$ `QueryIntervalTree`(R_T, q)

At each node the query algorithm spends $O(1 + k')$ time, where k' is the amount of intervals found in I'_T to return (the sorted lists allow the relevant intervals to be enumerated from I'_T simply by iterating through the sorted list until a non-matching element is found). As the algorithm recurses at most $O(\log n)$ times and no interval can be found twice, the total query time is $O(\log n + k)$, where k is the amount of intervals in the result.

As an example, the interval tree for $I = \{i_1, i_2, i_3, i_4\}$ is shown in Figure 2, where $i_1 = [0,2]$, $i_2 = [4,5]$, $i_3 = [1,3]$, $i_4 = [5,7]$. The empty leaf nodes are not shown. For each node T the median m_T and I'_T sorted by ascending left endpoint and descending right endpoint are shown.

If, for example, the tree in Figure 2 is queried for the value 6, the query algorithm starts at the root node, notes that $6 > 3$, notes that $6 \notin i_3$ and proceeds into the right subtree where it notes that $6 > 5$, $6 \in i_4$ and $6 \notin i_2$. Thus, the output is $\{i_4\}$.

[1] For the purposes of this report, the median of $2n$ numbers is the nth-smallest number.

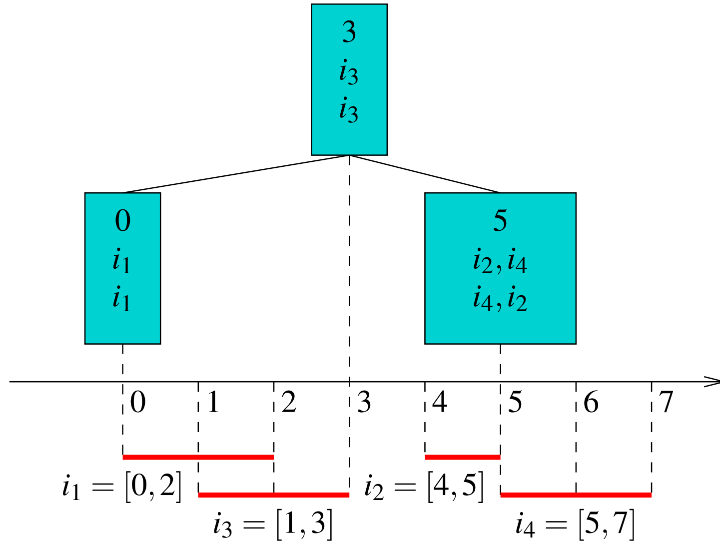

Figure 2: An example interval tree

2.3 Extending interval trees for windowing queries

The interval tree gives us a good way to find line segments that satisfy requirement 1 in case 3 in the axis-parallel line segment windowing query problem. However, requirement 2 is not addressed at all. Instead of returning all intervals in I'_T with the left/right endpoint less/greater than q (as applicable), only the intervals for which the attached y co-ordinate lies within the y bounds of the window should be returned. Luckily, this corresponds to the rectangular range query $]-\infty, q] \times [q_y, q'_y]$ on the set of left/right endpoints in I_m, which can be again be solved using a 2D range tree.

Thus, by replacing the two sorted lists in each interval tree node with two 2D range trees on the left and right endpoints (repectively) of the segments in I'_T and using a query on the corresponding range tree instead of the sorted list query, the interval tree query can be extended to complete the windowing query algorithm. The interval tree now uses $O(m \log m)$ space instead of $O(m)$ space for each node containing m segments, so the space requirements increase to $O(n \log n)$. Similarly, at each of the $O(\log n)$ nodes traversed during the search, $O(\log n + k')$ time is used (where k' is the number of intervals found in each node), so the total time used is $O(\log^2 n + k)$.

Based on the above, an algorithm can be constructed to answer windowing queries for axis-parallel line segments. The algorithm runs in $O(\log^2 n + k)$ time and uses $O(n \log n)$ storage.

2.4 Priority search trees

The structure for windowing queries described above uses range trees to find line segments that intersect two sides of the window. All of the range tree queries used for this purpose have a range that is unbounded on one side (the left side, for horizontal line segments). This property can be used to decrease the space requirements for the windowing data structure (although it is still $O(n \log n)$ due to the range tree used to find endpoints within the window). The specific problem addressed here can be described as:

Input A set P of points, where $n = |P|$ and a query range $R =]-\infty, q_x] \times [q_y, q'_y]$.

Output The set of k points in P that are in R.

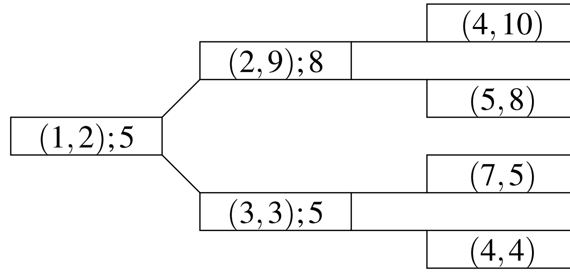

Figure 3: An example priority search tree

A more space-efficient data structure for range queries of the form $]-\infty, q_x] \times [q_y, q'_y]$ can be constructed from a heap in which the heap condition is that the x co-ordinate of a node is less than that of the nodes in the subtrees. In a heap, the division into left and right subtrees is arbitrary except that the subtrees should be (roughly) equal in size. Thus, the nodes can be divided into subtrees in such a way that a search on the y co-ordinate can be performed quickly.

The *priority search tree* (proposed by McCreight [8]) T for a non-empty set P of points is a binary tree with the following properties:

- p_T is the point in P with the smallest x co-ordinate.
- $P_T = P \setminus \{p_T\}$.
- m_T is the median of the y co-ordinates in P_T (if any).
- The root node of the priority search tree contains p_T and m_T.
- The left subtree is a priority search tree for $\{(x,y) \in P_T : y \leq m_T\}$.
- The right subtree is a priority search tree for $\{(x,y) \in P_T : y > m_T\}$.

If $P = \emptyset$, the priority tree is an empty leaf.

Co-ordinates are assumed to be distinct (in other words, no two points may share x or y co-ordinates). This can be achieved by using the other co-ordinate as a tie-breaker in comparisons[2].

An example of a priority search tree is shown in Figure 3. For each node T, p_T and (where applicable) m_T is shown.

The priority search tree can easily be built recursively in $O(n \log n)$ time (or even in $O(n)$ time, if the points are already sorted on the y co-ordinate).

The subtrees with y co-ordinate within the search range can be found as in range trees: by searching the tree for q_y and q'_y and reporting the nodes in the subtrees between the left and right boundaries formed by these searches. Due to the heap condition, the x co-ordinate grows for every step down into the tree one takes. Thus, within a subtree in which the y co-ordinate is known to be within the range, the values with the right x co-ordinate can be found by recursing downwards until the x co-ordinate is out of range.

The algorithm is thus:

[2]De Berg et al. [2] refer to using *composite numbers*, which is exactly the same thing, but they require an entire page to explain it.

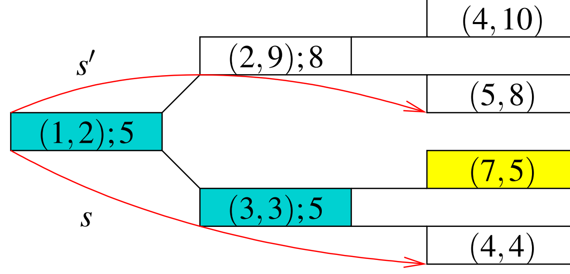

Figure 4: An example query on a priority search tree

`QueryPrioritySearchTree`(T, q_x, q_y, q'_y):
Search for q_y and q'_y in T, storing the search paths as s and s' respectively.
$P_R \leftarrow \emptyset$
for $T' \in s \cup s'$:
    if $p_{T'} \in]-\infty, q_x] \times [q_y, q'_y]$
        $P_R \leftarrow P_R \cup \{p_{T'}\}$
for $T' \in s \setminus s'$:
    if successor in s of T' is $L_{T'}$:
        $P_R \leftarrow P_R \cup$ `ReportInSubTree`$(R_{T'}, q_x)$
for $T' \in s' \setminus s$:
    if successor in s of T' is $R_{T'}$:
        $P_R \leftarrow P_R \cup$ `ReportInSubTree`$(L_{T'}, q_x)$
return P_R

`ReportInSubTree`(T, q_x):
if T not a leaf and $x_{P_T} \leq q_x$
    return $\{P_T\} \cup$ `ReportInSubTree`$(L_T, q_x) \cup$ `ReportInSubTree`(R_T, q_x)

The algorithm executes in $O(\log n + k)$ time and uses a tree that requires $O(n)$ storage and can be built in $O(n \log n)$ time.

Figure 4 shows the paths s and s' used when the query $]-\infty, 3] \times [2, 6]$ is performed on the example in Figure 3, as well as the nodes for which `ReportInSubTree` are called (in yellow) and the nodes whose points belong to the output (in cyan).

3 Windowing for arbitrarily oriented line segments

For applications such as road maps that contain arbitrarily oriented line segments, interval trees are inadequate. One way to adapt interval trees to this type of application is to replace each segment with its *bounding box* (the smallest rectangle with axis-parallel sides that contains the line segment). In many cases this may be an acceptable solution, but in the worst case, it may test all line segments even though none intersect the window.

The problem, then, is the same as in Section 2, except that one word has changed:

Input A set S of ~~axis-parallel~~ non-intersecting line segments, where $n = |S|$ and a query window $W = [x, x'] \times [y, y']$.

Output The set $S_W \subseteq S$ of line segments that intersect W; in other words, $s \in S_W \Leftrightarrow s \in S \wedge s \cap W \neq \emptyset$.

The reason for requiring non-intersecting segments will become apparent as the solution to the problem is explained.

3.1 Problem analysis

As in the previous cases, segments with an endpoint in the window can be reported using a range tree. To report segments without an endpoint in the window, the segments that intersect a vertical (or horizontal, which is similar) query segment must be found. In the case where the segments in S do not intersect, a *segment tree* can be used for this purpose.

3.2 Segment trees

The interval tree does not provide a satisfactory solution to the problem of windowing queries on arbitrarily oriented line segments. However, it can be replaced with a tree that uses a different approach to the problem of representing intervals. In other words, segment trees also address the stabbing query problem described in Subsection 2.2.

The segment tree (developed by Bentley [1]) is based on splitting the real line into *elementary intervals* that are bounded by the endpoints of the input intervals. For a set of endpoint x co-ordinates $P = \{p_1, p_2, \ldots, p_m\}$ (the endpoints of the intervals, sorted in ascending order with duplicates removed), the elementary intervals are (from left to right): $]-\infty, p_1[, [p_1, p_1],]p_1, p_2[, [p_2, p_2], \ldots,]p_{m-1}, p_m[, [p_m, p_m],]p_m, \infty[$.

The leaves of a segment tree (from left to right) correspond to the elementary intervals. The internal nodes of the segment tree correspond to the union of the intervals of their children. Each node T contains the interval I'_T corresponding to the node and the set $I_T \subseteq I$ of intervals i such that $I'_T \subseteq i$ and $I'_{\pi_T} \nsubseteq i$, where π_T is the parent of T. In other words, the intervals in I are stored in the highest nodes that correspond to an interval they contain.

A segment tree is a balanced binary search tree with at most $4n+1$ leaves, so its depth is $O(\log n)$. As each interval in I is stored in at most two nodes of the same depth, the storage requirement of the segment tree is $O(n \log n)$.

A segment tree can be constructed by creating a node for each elementary interval and then repeatedly combining as many nodes as possible into pairs and forming a new internal node as a parent for each pair of nodes and finally inserting all the intervals into the resulting tree structures:

`CreateSegmentTree`(I):
$p_1, \ldots, p_m \leftarrow$ all endpoints in I (sorted in ascending order, duplicates removed)
$Q \leftarrow$ empty queue
for each interval i in $]-\infty, p_1[, [p_1, p_1],]p_1, p_2[, [p_2, p_2], \ldots,]p_{m-1}, p_m[, [p_m, p_m],]p_m, \infty[$
    Create a node T
    $I'_T \leftarrow i$
    $I_T \leftarrow \emptyset$
    Add T to end of Q.
while $|Q| > 1$
    $Q' \leftarrow$ empty queue
    while $|Q| > 1$
        Remove T_1 and T_2 from the beginning of the queue
        Create a node T
        $I_T \leftarrow \emptyset$

```
        I'_T ← I'_{T_1} ∪ I'_{T_2}
        Add T to end of Q'
    Add the single element of Q (if any) to end of Q'
    Q ← Q'
T ← the single element of Q
for every interval i ∈ I
    InsertSegmentTree(T, i)
return T

InsertSegmentTree(T, i):
if I'_T ⊆ i
    I_T ← I_T ∪ {i}
else
    if I'_{L_T} ∩ i ≠ ∅
        InsertSegmentTree(L_T, i)
    if I'_{R_T} ∩ i ≠ ∅
        InsertSegmentTree(R_T, i)
```

The query algorithm is simple:

```
QuerySegmentTree(T, q_x):
if T is a leaf
    return ∅
if q_x ∈ I'_{L_T}
    return I_T ∪ QuerySegmentTree(L_T, q_x)
else
    return I_T ∪ QuerySegmentTree(R_T, q_x)
```

Again, the search algorithm uses $O(\log n + k)$ time.

Figure 5 is a segment tree containing the same intervals as the interval tree in Figure 2. For each node T, I'_T and I_T are shown.

3.3 Segment trees in windowing

In order to be useful for windowing, segment trees must be adapted to provide efficient searching in two dimensions. The basic segment tree query described above quickly finds the line segments that intersect a vertical line at a specified x co-ordinate. Checking that the intersection point is within the bounds of a vertical line segment (corresponding to the edge of a window) is not hard to add.

Instead of storing the list I_T of intervals in each node T, the corresponding segments S_T are stored. Where the algorithm above uses an interval, the x co-ordinate interval of the corresponding segment is used. The segments in S_T each span I'_T (but not I'_{π_T}) and do not intersect each other. Thus, their vertical ordering is the same in all of I'_T. By storing S_T as a binary search tree, the lines that intersect a vertical line segment can be found in $O(\log n + k_T)$ time, where k_T is the number of intersecting segments in the BST.

The resulting query algorithm is:

```
WindowSegmentQuery(T, q_x, q_y, q'_y):
if T is a leaf
```

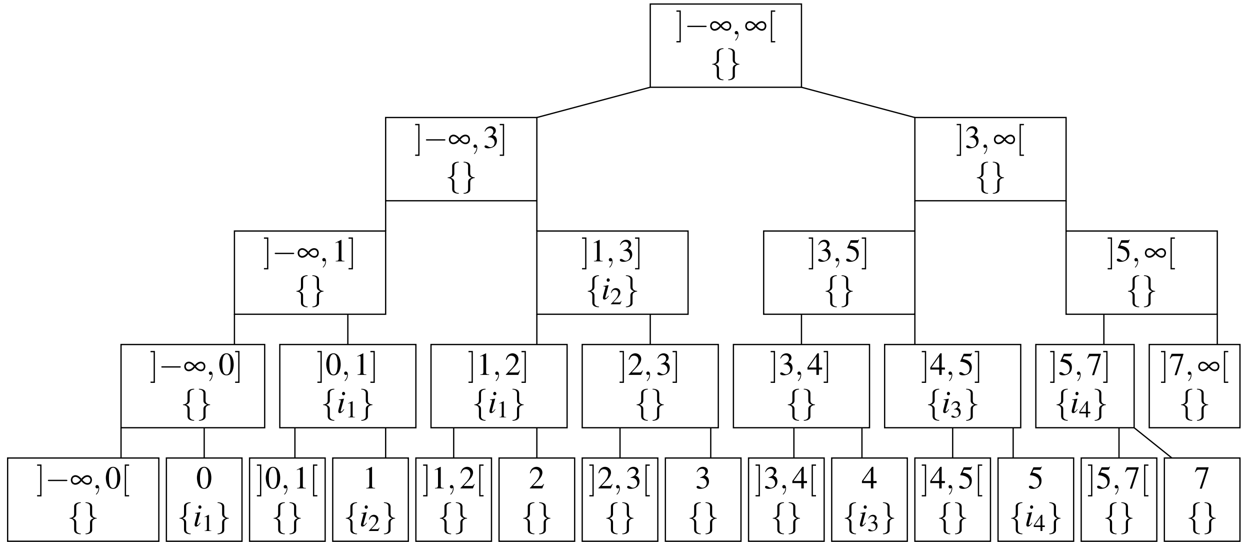

Figure 5: An example segment tree

    return $\emptyset$
if $q_x \in I'_{L_T}$
    return `SegmentBSTQuery`$(S_T, q_x, q_y, q'_y) \cup$ `WindowSegmentQuery`(L_T, q_x, q_y, q'_y)
else
    return `SegmentBSTQuery`$(S_T, q_x, q_y, q'_y) \cup$ `WindowSegmentQuery`(R_T, q_x, q_y, q'_y)

Here, `SegmentBSTQuery`(S, q_x, q_y, q'_y) is a one-dimensional query on the interval $[q_y, q'_y]$ in a binary search tree, where the y co-ordinate of the intersection point with the query segment at q_x must be calculated for each segment that is tested.

The algorithm runs in $O(\log^2 n + k)$ time and uses $O(n \log n)$ storage.

Segment trees can be generalised to d dimensions and then use $O(n \log^{d-1} n)$ storage and answer queries in $O(\log^d n + k)$ time. *Fractional cascading* can be used to lower query time by a logarithmic factor. Using an interval tree on the deepest level of nested trees lowers the storage bound by a logarithmic factor.

4 Dynamic windowing data structures

Allowing insertions and deletions of line segments requires changes to the data structures described here. Cormen et al. [4] describe an interval tree based on a red-black tree that supports insertion and deletion in $O(\log n)$ time. However, their interval tree is designed for a slightly different problem: to find a single interval in the tree that overlaps a query interval, instead of all intervals in the tree that contain a query value. It does this in $O(\log n)$ time.

Priority search trees can use the insertion and deletion operations of a balanced search trees that require only $O(1)$ rotations per update (such as red-black trees, described by e.g. Cormen et al. [4]) to allow insertion and deletion in $O(\log n)$ time. [8]

Van Kreveld and Overmars describe a dynamic segment tree that supports insertion in $O(\log n)$ time and deletion in $O(\log n \cdot \alpha(i,n))$ amortised time, where $\alpha(i,n)$ is the row-inverse of Ackermann's

function for a constant i. $\alpha(i,n)$ grows very slowly for $i \geq 2$ and can be considered to be a small constant for any sane value of n. [6]

5 Conclusions

In this report, I presented some data structures and algorithms that can be used to perform windowing queries efficiently on line segments in a plane.

Using interval trees, windowing queries on axis-parallel line segments in a plane can be performed using a data structure built in $O(n \log n)$ time and using $O(n \log n)$ storage in $O(\log^2 n + k)$ time (regardless of whether the interval trees use range trees or priority search trees, although the priority search trees are more space-efficient). Using segment trees, the same can be done on non-intersecting line segments; the time and space requirements are of the same magnitude as in the axis-parallel case.

These data structures allow practical windowing queries for many applications related to spatial data, such as CAD/CAM, navigation systems and PCB layout.

References

[1] J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, 1977.

[2] Mark de Berg, Marc van Kreveld, Mark Overmars and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000.

[3] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, volume 15, number 3, pages 703–724, August 1986.

[4] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Technische Universität Graz, 1980.

[6] Marc van Kreveld and Mark Overmars. Union-copy structures and dynamic segment trees. *Journal of the Association for Computing Machinery*, volume 40, number 3, pages 635–652, July 1993.

[7] Edward McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-9, Xerox PARC, June 1980.

[8] Edward McCreight. Priority search trees. *SIAM Journal on Computing*, volume 14, number 2, pages 257–276, May 1985.