CONQUERING CONTOURS

EFFICIENT ALGORITHMS FOR COMPUTATIONAL GEOMETRY

Dissertation

zur Erlangung des Grades des

Doktors der Naturwissenschaften

der Universität Dortmund
an der Abteilung Informatik

von

RALF HARTMUT GÜTING

Dortmund

1983

Tag der mündlichen Prüfung:	27. Juni 1983
Dekan:	Prof. Dr. Bernd Reusch
Gutachter:	Prof. Dr. Armin B. Cremers
	Prof. Dr. Thomas Ottmann
	Prof. Dr. Derick Wood

Abstract

Motivated by applications in e.g. VLSI-design, computer graphics, databases, computational geometry is concerned with studying the computational complexity of elementary geometric problems. This thesis is a rather broad study of one specific problem of this kind, the contour problem: Given a set of iso-oriented rectangles in the plane, compute the boundary of their union.

The thesis consists of three main parts. In the first part we present the first time-optimal solution of the contour problem, achieved by means of a line-sweep algorithm. Furthermore we investigate a more general problem: The i-area is defined as the region of the plane covered by exactly i rectangles. We describe a number of line-sweep algorithms to compute i-contours (boundaries of i-areas) and i-measures (sizes of i-areas). Some of those algorithms are time- and/or space-optimal.

The algorithmic paradigm of divide-and-conquer so far had typically not been applied to sets of planar objects other than points. Line-sweep appeared to be the only suitable method to treat for instance sets of rectangles efficiently. We overcome the conceptual difficulties of divide-and-conquer by introducing a new idea called separational representation of planar objects. The new technique permits efficient divide-and-conquer solutions for a wide range of problems based on orthogonal planar objects. We prove this by describing time-optimal algorithms for five nontrivial problems including the contour problem.

So far, all the problems studied were based on orthogonal objects. In general for those objects much more efficient solutions can be obtained than for arbitrarily oriented objects (polygons). To bridge this gap in complexity we introduce in the third part of the thesis the notion of c-oriented polygons, that is polygons, whose edges are oriented in only a constant number of previously defined directions. We obtain a solution of the contour problem for c-oriented polygons which is of much higher complexity than the one for rectangles, but still better than the solution for arbitrary polygons. For other problems, however, there exist quite efficient solutions. We show this for the 'stabbing number' problem and the polygonal intersection searching problem.

Acknowledgements

Most of the research of this thesis was done during a one-year stay at McMaster University supported by the DAAD (Deutscher Akademischer Austauschdienst) under the excellent guidance of Derick Wood. He turned my interest to computational geometry and initiated my study of the contour problem. By discussing problems and posing interesting questions he provided a really stimulating research environment. For all his support and friendship I am very thankful to him.

After my return to Dortmund, Armin B. Cremers guided me through the second phase of my work. To him I wish to express my gratitude for his support and encouragement, also for all the time he spent in reading and correcting draft versions of the thesis. His appreciation of my work was a great help to me.

For refereeing the thesis quickly and carefully, I owe many thanks to Armin Cremers, Thomas Ottmann and Derick Wood.

Thanks to my wife Edith for being willing to go to Canada with me, for her support during the time I worked on the thesis and in particular for the beautiful job she did in typing the manuscript. Finally, thanks to God, who gave me strength and ideas necessary for this work.

Table of contents

1. *INTRODUCTION.*

This thesis is a case study of a specific problem in computational geometry, the contour problem. Before stating the problem let us briefly look at computational geometry in general. Following a definition by Shamos [Sh], computational geometry is the study of the computational complexity of elementary geometric problems. The problems considered are, for instance:

- Given a set of points in 2-space, determine all points inside a specified rectangle.
- Given a set of line segments in 2-space, find all intersecting pairs.
- Given a set of intersecting rectangles in the plane, compute the total size of the 'covered' area of the plane.

There exist different motives for the study of this kind of problems. In our opinion the most important among them are:

- From a theoretical point of view the problems are appealing since they can be stated in a very simple but precise manner while efficient solutions often require the most refined data structures and sophisticated algorithms. Thus the study sheds light on the power of different algorithmic techniques and contributes to 'a theory of algorithms and data structures'.

- Often real-world problems, which seem to have no connection to geometry at all, can be formulated in an abstract manner as geometric problems. We give a few examples:

  (1) In databases information is frequently stored in the form of multi-attribute records. The attribute values are often drawn from some

ordered domain, which can be interpreted as a one-dimensional space. Hence a record corresponds to a point in multidimensional space. The first geometric problem above can be interpreted as retrieving information from a database fulfilling certain conditions.

(2) Certain problems in operating systems, namely safety and deadlock-freeness of locked transaction systems can be related to geometric problems based on sets of rectilinearly oriented rectangles. A locked transaction system is safe iff the closure of the corresponding set of rectangles consists of one connected region. The geometric solution provides the most efficient algorithm known to solve the mentioned problems [SoW].

(3) In speech recognition a word uttered by a speaker can be analyzed by some input device and decomposed into a number of features. The collection of features can again be interpreted as a point p in multi-dimensional space. The vocabulary of the recognizer can be given as a set of points S in k-space. Finding the word most likely spoken then corresponds to finding the point in S closest to p (the well-known 'nearest-neighbour-problem'). This example is due to Bentley [Be1].

Stating the problems in geometric terms strongly supports human intuition and thus leads to efficient algorithms.

- In many applications geometric problems play an important role in their own right. For instance the nearest-neighbour problem in 2-space can also be interpreted as searching for the nearest post-office for a given location. Further examples of application areas are given below.

The contour problem.

For a set of iso-oriented (=sides parallel to the coordinate axes) rectangles the contour is the boundary between the 'free' and the 'covered' area of the plane. The contour problem is to compute this boundary (a more precise problem formulation is given in Section 3).

The contour problem is interesting for a number of reasons. First, there exist numerous applications based on sets of iso-oriented rectangles. In VLSI-design sets of rectangles are used to define chip layouts for complex integrated circuits [MeC], [La1]. This has motivated in particular an extensive study of the problem of finding all intersecting pairs in a set of rectangles. Sets of rectangles play an important role in architectural databases [EaL] and in geography [BeS]. They also have applications in computer graphics.

Second, there was no optimal algorithm known when we began our work. Lipski and Preparata [LiP] posed and first attacked the problem, coming up with an $O(n \log n + p \log(2n^2/p))$ time solution where n is the number of rectangles and p the size of the contour. In this thesis we present time-optimal algorithms, achieving a worst-case time of $O(n \log n + p)$.

Third, the problem has some inherent difficulty which makes it a challenge for the algorithm designer. Let us explain what we mean by this. The contour problem is in a certain sense non-decomposable. That is, we cannot afford to compute the contour for some subset A of a given set of rectangles R and hope to modify or complete the solution later by looking at the rectangles in R - A. There might exist a rectangle $r \in (R-A)$ which encloses all rectangles in A, making all the previous work superfluous. This is in contrast to the rectangle intersection problem, for instance, where intersections found in a subset remain 'valid' under all circumstances, whatever the complete set looks like.

However, the scope of this thesis is not limited to the contour problem. Rather, we used it as a guide to direct our research. Studying the contour problem quite a number of related problems came into view and some of them were solved.

Let us now briefly sketch the 'historic' development of our research. We started by improving the line-sweep algorithm of Lipski and Preparata [LiP] for the contour problem by means of a more sophisticated supporting data structure called a contracted segment tree, resulting in a time-optimal algorithm. We then looked at a generalization of the contour concept due to Wood [Wo] called i-contours. The i-contour is the boundary between areas of the plane covered (i-1) and i times, respectively. We devised a number of algorithms to compute i-contours and i-measures (the i-measure is the size of the area covered i times). All this was done by use of line-sweep algorithms.

The algorithmic paradigm of divide-and-conquer is an interesting alternative to line-sweep. However it had typically not been used for problems in planar geometry except for those based on point sets. It seemed that objects like line segments, rectangles, polygons etc. could not efficiently be treated by divide-and-conquer.

This impression proved wrong when we found a time-optimal divide-and-conquer solution of the rectangle intersection problem. Since this problem is somewhat easier we used it to gain some knowledge about planar divide-and-conquer before attacking the contour problem. In this study the central idea of 'separational representation' (see Section 4) emerged. - A second preparatory step was the design of a divide-and-conquer algorithm to compute the measure (the size) of the area covered by a set of rectangles. The measure problem is related to the contour problem because both measure and contour are properties of the <u>union</u> of a set of rectangles. In fact, the contour algorithm which was finally discovered first constructs an abstract description of the union (which can also be used to solve the measure problem) and then computes the contour from this description. This algorithm is also time-optimal and simpler than the line-sweep contour algorithm. However, its space-requirements are somewhat higher.

All the work done so far was restricted to orthogonal objects. An interesting question is how much the complexity of the problem increases if the objects are allowed to be oriented in for instance three (instead of two) possible directions. This led to the definition of c-oriented objects, that is objects whose defining line segments are oriented in at most a constant number of possible directions. Unfortunately we did not discover a really nice algorithm to compute the contour of a set of c-oriented polygons. The algorithm found is only slightly more efficient than the known algorithm for arbitrary polygons. The increased effiency is due to the fact that intersections can be found more easily among c-oriented than among arbitrary line segments.

However, the notion of c-oriented objects proved useful because some other problems based on them could be solved efficiently. We found an optimal algorithm for the stabbing number problem (Given a set of c-oriented polygons S and a query point p, how many polygons in S contain p?). We also developed an efficient solution of the c-oriented polygonal intersection searching problem. That is, given a set of c-oriented polygons S and a c-oriented query polygon q, find all polygons in S intersecting q.

The thesis is structured as follows: We start with a short preliminary section containing some basic definitions and notations which we use freely throughout the thesis (Section 2). Section 3 contains line-sweep algorithms, namely the optimal contour algorithm (Section 3.1) and a collection of algorithms for 'higher' contour and measure problems (Section 3.2). - Section 4 is devoted to divide-and-conquer algorithms, that is the algorithm for finding all intersections in a set of rectangles (Sections 4.1 - 4.3; the problem is solved by combining the solutions of two subproblems) and the combined measure/contour algorithm (Section 4.4). - Section 5 shows how to cope with (restricted) non-orthogonality; apart from describing a not completely satisfying contour algorithm for c-oriented polygons we give nicer algorithms for the stabbing number problem in Section 5.1 and the polygonal intersection searching problem in Section 5.2. - In Section 6 we try to evaluate the work done and identify the questions arising from it. This includes a list of open problems encountered in the various phases of the work.

2. *DEFINITIONS AND NOTATIONS.*

In this section we give a few basic definitions and notations which are used throughout the thesis. The section may be used as a reference if unknown terms occur in one of the following sections. More specific definitions, however, appear in the text whenever they are needed and are not listed here.

1-space.

One-dimensional space, or the line, is given by the set of real numbers $\mathbb{R}$. An interval $[x_l, x_r]$ is a subset of 1-space.

Occasionally, especially in diagrams, we use the notation a-b for an interval [a, b].

The measure of an interval i, measure(i), is x_2-x_1 for $i = [x_1, x_2]$. We also denote measure(i) by $|i|$. Based on this, the measure of the union of a set of intervals is defined in the usual way.

We say a set of intervals I is based on a set of points P if each interval end-point is in P. On the other hand, a set of points P defines in a natural manner a partition of the line into a set of 'atomic' intervals, called fragments:

$$\text{partition}(P) := \{[x_1, x_2] \mid x_1, x_2 \in P \text{ and } \forall x \in P: x \leq x_1 \text{ or } x \geq x_2\}.$$

We say interval i_1 contains interval i_2, iff $i_2 \subseteq i_1$.

2-space.

Two-dimensional space, or the plane, is represented by the set $\mathbb{R} \times \mathbb{R}$.

A line segment is a subset of $\mathbb{R} \times \mathbb{R}$. If it is axis-parallel (i.e. horizontal or vertical) it is represented either by a point and an interval, that is as a pair (x, i_y) or (i_x, y), or simply by three coordinates (x, y_1, y_2) or (x_1, x_2, y), respectively. Obviously a pair (x, I_y), where x is an x-coordinate and I_y a set of y-intervals, defines a set of line segments. For emphasis we sometimes use the notation linesegments(x, I_y) for this set instead of the simpler (x, I_y).

An arbitrary line segment is given by its endpoints as a pair (p_1, p_2), p_i a point in 2-space, or as a quadruple (x_1, y_1, x_2, y_2).

A rectangle is also a subset of 2-space. It is given either by a quadruple (x_l, x_r, y_b, y_t) or alternatively by the product $i_x \times i_y$ of an x- and a y-interval.

For a set of rectangles R, union(R)$:= \bigcup_{r \in R} r$.

To be able to talk about the set of all rectangle coordinates (in x- or y-direction) we define

$$\text{x-set}(R) := \{x \in \mathbb{R} \mid \exists r \text{ in } R: x = x_l(r) \text{ or } x = x_r(r)\}$$

and y-set(R) analogously.

We project rectangles onto the axes by defining

$$\text{x-proj}(r) := [x_l, x_r] \quad \text{for } r = (x_l, x_r, y_b, y_t)$$

(similarly y-proj(r)). For a set of rectangles R x-proj(R) and y-proj(R) yield a set of x- and y-intervals, respectively.

The application of x-proj (or y-proj) to an arbitrary set of points $P \subset \mathbb{R} \times \mathbb{R}$ yields a set of x- (or y-) coordinates. However, we will interpret e.g. x-proj(P) as the set of disjoint x-intervals defined by the projection rather than as a set of solitary points on the x-axis.

For a set of tuples T we denote the projection onto the i-th component by $\text{proj}_i(T)$. The result is the set of all values of the i-th component occurring in tuples of T.

Trees.

The following notations for trees are used. Let p be a node of some tree T. Then T(p) is the subtree of T rooted in p. Refering to T(p) we also speak of p's subtree. Furthermore we use the notation λp and ρp for p's left and right son, respectively.

Model of computation, O-notation.

To be able to analyze algorithms we need some model of computation. For computational geometry the real RAM as described by Shamos [Sh] seems to be a suitable and sufficiently realistic model. Essentially it is the RAM model of Aho, Hopcroft and Ullman [AHU] with the additional assumption that a single storage cell is able to store a real number with unlimited precision. In other words, on the level of algorithm design described in this thesis we ignore the difficulties which might arise from the limited precision of the representation of real numbers in computers. However, these effects have to be taken care of in implementations.

We always analyze algorithms and data structures in this thesis with regard to the asymptotic worst-case time and space requirements. To denote bounds we use the well-known O-, Ω- and Θ-notation:

Let $f\colon \mathbb{N} \to \mathbb{N}$, $g\colon \mathbb{N} \to \mathbb{N}$. Then

(a) $f(n) = O(g(n))$ iff there exists a positive real number c such that $f(n) \leq c \cdot g(n)$ for all but finitely many $n \in \mathbb{N}$.

(b) $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.

(c) $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Often logarithms appear in these bounds which are in this thesis always to the base 2.

3. *LINE-SWEEP ALGORITHMS.*

This section contains a rather extensive study of line-sweep algorithms for the contour problem and related problems. The line-sweep paradigm was apparently introduced into computational geometry by Shamos and Hoey [ShH] and afterwards widely used to solve problems based on a set of objects in 2-space. The idea is to move a (say) vertical line from left to right through the set of objects. At any position the sweepline may intersect some of the objects which are represented by their one-dimensional projection onto the sweepline. The problem is solved by observing the interaction between the projected objects. In this manner a static problem in two dimensions is reduced to a dynamic problem in one dimension. - Of course the technique can be generalized to higher dimensions by sweeping a (k-1)-dimensional hyperplane through k-space; a nice example for this is [vLW].

In Section 3.1 we give a more precise formulation of the contour problem and reconsider an algorithm to solve it by Lipski and Preparata [LiP]. By replacing the data structure supporting the line-sweep (a counting segment tree) by a more sophisticated structure (called a contracted segment tree) we achieve time-optimal performance.

The measure problem is to determine the total size of the area covered by a set of rectangles. It is closely related to the contour problem since both measure and contour are properties of the <u>union</u> of a set of rectangles. A time- and space-optimal solution of the measure problem was given by Bentley [Be2]. In Section 3.2 we study generalized contour and measure problems based on the notion of an <u>i-area</u>. For a given set of rectangles the i-area ($i \in \mathbb{N}$) contains those points of the plane which are covered by exactly i rectangles. This induces the concepts of <u>i-measure</u> and <u>i-contour</u>. We develop efficient algorithms for a number of problems based on these concepts. In a final subsection (3.2.9) we prove lower bounds for all the problems considered which enables us to show that some of our algorithms are optimal.

Throughout Sections 3 and 4 we study only orthogonal objects. Hence speaking of a set of rectangles we always mean iso-oriented (=axis-parallel) rectangles.

The results of Section 3.1 have been published previously in [Gü1].

3.1. *Computing the contour of a set of rectangles.*

3.1.1. *The contour problem.*

We start by stating the contour problem more precisely:

> Let $R = \{r_1,...,r_n\}$ be a set of rectangles and let $F = r_1 \cup r_2 \cup ... \cup r_n$. The contour of the union F is defined as follows. Let EDGES be the set of horizontal and vertical line segments which separate the region F from the 'free' area of the plane, $\overline{F}$. Two elements of EDGES are connected, if they contain a common point. The transitive closure of the connectedness relation partitions EDGES into a collection of 'cycles'. Each of these cycles is a sequence of alternating horizontal and vertical edges (see Figure 3-1). This collection of cycles is called the contour. Now define an optimal algorithm to compute the contour, if R contains n rectangles and the contour consists of p edges.

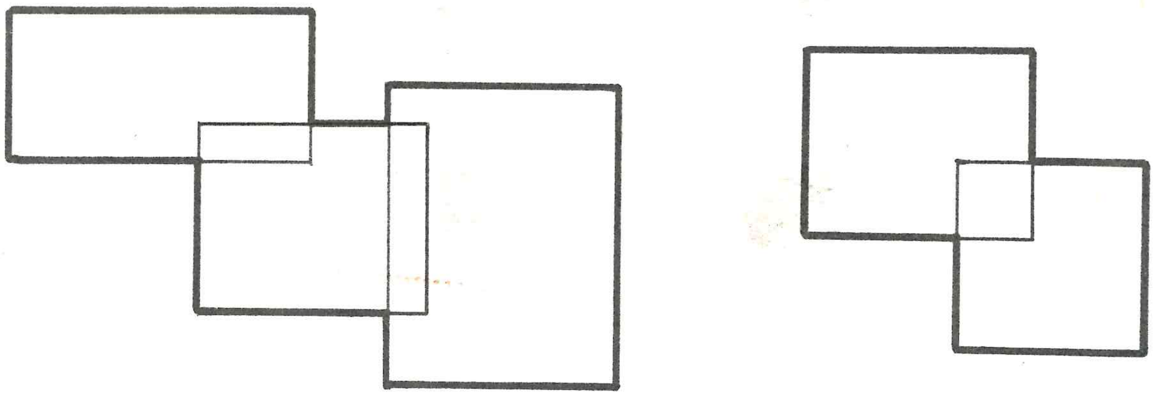


Figure 3-1

As mentioned before, this problem was recently attacked by Lipski and Preparata [LiP]. Their algorithm finds the contour in $O(n \log n + p \log (2n^2/p))$ time and $O(n + p)$ space. This performance is time-optimal for $p = O(n)$ and $p = O(n^2)$, but not in general ($p = O(n^\mu)$ for $1 < \mu < 2$, say). In the sequel we develop an optimal $O(n \log n + p)$-time algorithm.

We describe our solution of the contour problem in a top-down manner. The top level (Section 3.1.2) consists of an algorithm A performing the line-sweep which makes use of a supporting data structure SD. SD must have the following properties:

- It represents a set of intervals I.
- It allows two operations:
  - insertion-query: asks for those subintervals of an interval i which are free (that is, not covered by any interval in I) and inserts i into SD.
  - deletion-query: deletes an interval from SD and asks then for the subintervals of i free with respect to I.

Then we are already able to analyze the performance of algorithm A provided the operations of SD can be performed within certain time bounds.

On a second level (Section 3.1.3) we introduce a new data structure called a contracted segment tree to implement SD. Operations INSERTION-QUERY and DELETION-QUERY are implemented by algorithms which use some more basic operations LEFTDOWN, RIGHTDOWN, UP, LINK-OUT, LINK-IN and REPORT. Again we establish the required time bounds for INSERTION-QUERY and DELETION-QUERY provided the basic operations can be performed within certain time bounds.

The third and final level of description (Section 3.1.4) shows how to implement the six basic operations and proves that the implementation yields the time bounds required on the next higher level.

3.1.2. *The algorithm.*

The algorithm for computing the contour consists of two phases. In the first phase a vertical line is moved from left to right through the plane. During this process all vertical edges of the contour are found. In a second phase horizontal edges are created from the list of vertical edges and all edges are linked into the cycles which form the contour.

In [LiP] it is shown, that after an initial sorting of all rectangle coordinates the second phase of the algorithm takes $O(p)$ time and space, where p is the number of edges in the contour. Hence we will now concentrate on the first phase.

The line-sweep.

Let $R = \{r_1,\dots,r_n\}$ be the set of rectangles. We form the set of all vertical rectangle edges, representing the left and right side of a rectangle $r = (x_l, x_r, y_b, y_t)$ by $(x_l, \text{left}, y_b, y_t)$ and $(x_r, \text{right}, y_b, y_t)$, respectively. All edges are then sorted in lexicographical order assuming left < right (the reason for this is explained below).

Now the line-sweep is performed by scanning the ordered list of vertical rectangle edges. Let L be the sweepline and $F = \text{union}(R)$. At any position of L, each of the sets $L_c = L \cap F$ and $L_f = L - L_c$ contains a collection of disjoint intervals.

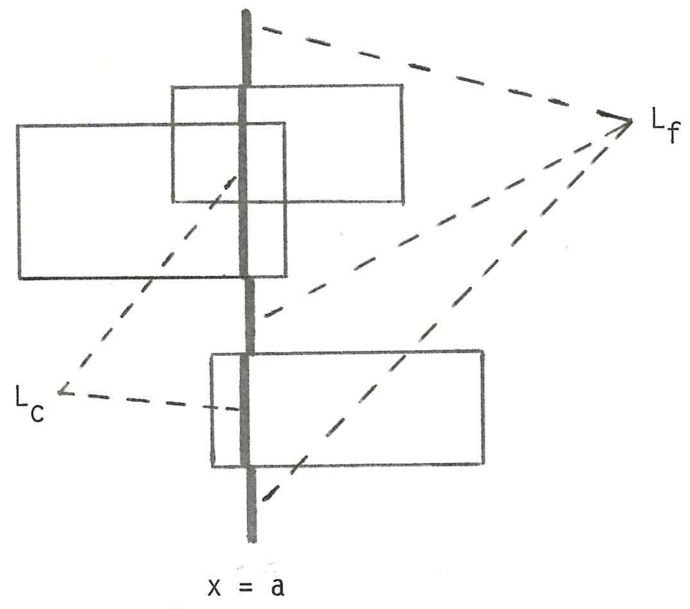


Figure 3-2

Let L be fixed at a certain position $x = a$. The present rectangles are those fulfilling $x_l \leq a$, $x_r \geq a$. The intervals in L_c are formed by the union of the y-intervals $[y_b, y_t]$ of all present rectangles (see Fig. 3-2).

L_c and L_f change only when a vertical side s of a rectangle is encountered. The contribution of s to the contour is $s \cap L_f$, where L_f is taken:

- before encountering s, if s is a left side
- after encountering s, if s is a right side.

Hence the line-sweep algorithm works as follows:

- Encountering the left side of a rectangle, the vertical interval is inserted into a supporting data structure SD, storing intervals, and the free parts of the line which become covered are reported.

- Encountering the right side of a rectangle, the vertical interval is deleted from SD and the parts of the line which become free are reported.

Now it becomes clear why we defined left < right. In this way we ensure that whenever one rectangle is 'closed' and another one 'opened' at the same x-coordinate, their intersection will not contribute to the contour (see Fig. 3-3).

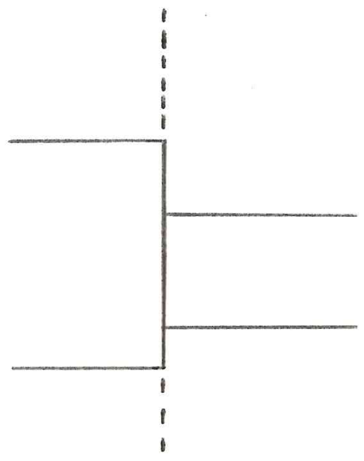


Figure 3-3

We have seen that we need a data structure which enables us to efficiently perform the two operations:

- insertion of an interval reporting intersected (up to now) free parts of the line.
- deletion of an interval reporting intersected (from now on) free parts of the line.

For this, Lipski and Preparata [LiP] use a variant of the segment tree which we call a <u>counting segment tree</u>. We briefly describe this data structure, since it provides a foundation and motivation for the contracted segment tree.

Using counting segment trees.

The segment tree was originally introduced by Bentley [Be2]. A detailed description can be found in [BeW]. We describe the counting segment tree used in [LiP].

The (counting) segment tree represents a set of intervals. It is semidynamic in the following sense: For a fixed set of intervals I the set of all endpoints of intervals E(I) defines a partition of the line into 'atomic' intervals. The segment tree S is a binary searchtree of minimal height representing the atomic intervals in its leaves. In a natural way each internal node p has an associated interval <u>interval(p)</u> which is the concatenation of the leaf intervals of its subtree. The structure described so far we call an <u>empty</u> segment tree <u>based on I</u>. It serves as a frame which allows easy insertion or deletion of an interval <u>from I</u>. Any interval i in I defines a collection of nodes from S, CN(i), by

$$p \in CN(i) \Leftrightarrow interval(p) \subseteq i \text{ and } interval(f) \nsubseteq i \text{ (where f is p's father).}$$

To insert or delete intervals from I into S we augment each node by an additional integer field <u>cover(p)</u>. For the empty segment tree holds cover(p) = 0 for all nodes. An interval i is inserted or deleted by increasing or decreasing, respectively, the cover-fields of all nodes in CN(i).

It is evident that for any interval $i \in I$ the nodes in CN(i) are attached to some 'forked path' (see Fig. 3-4):

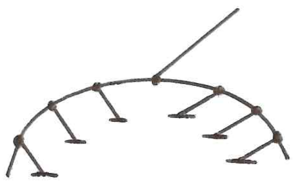


Figure 3-4

Therefore, if I has cardinality n then there are at most $O(\log n)$ nodes in CN(i) and $O(\log n)$ is also the time bound for an insertion or deletion.

Since for the contour problem the set of all y-intervals of rectangles is known in advance the segment tree may be constructed before the line-sweep begins.

This data structure is used in the algorithm of Lipski and Preparata. Inserting an interval i the nodes in CN(i) are located in $O(\log n)$ time. For each such node, the free parts of its subtree have to be reported. For this, an additional field status(p) is maintained for each node p, indicating whether p's subtree is full, empty or partially covered by intervals in this subtree. To report the free parts of a subtree $q \in CN(i)$, it is traversed, but subtrees of it which are full or empty are not entered.

A careful analysis of this algorithm shows that it uses $O(n \log n + p \log(2n^2/p))$ time. The subtree traversal for the nodes in CN(i) causes some inefficiency. Consider a node $q \in CN(i)$. If the ß free parts in q's subtree were attached to q as an interval list, then they could be reported in $O(ß)$ time, yielding a total time of $O(n \log n + p)$, provided the interval lists could also be maintained in $O(\log n)$ time per insertion/deletion. This time is optimal (see Section 3.2.9).

This observation leads to the development of the contracted segment tree. Instead of a list of free intervals, a tree is maintained for each node. This tree, however, has only $O(ß)$ nodes, if it contains ß gaps (free intervals), hence it can be traversed in $O(ß)$ time.

Using contracted segment trees.

First, let us see what we can gain by employing contracted segment trees as a supporting data structure in the line-sweep algorithm. In Section 3.1.3 we prove the following properties of contracted segment trees.

Property 1: In a contracted segment tree for n intervals insertion of an interval i reporting the γ free parts of the line which become covered as well as deletion of an interval i together with reporting the γ covered parts of the line which become free, can be done in $O(\log n + \gamma)$time (Theorem 3.4).

Property 2: A contracted segment tree for n intervals uses $O(n)$ space. The empty tree can be constructed in $O(n)$ time from the ordered list of interval-endpoints (Lemma 3.2).

This enables us to prove the main result:

Theorem 3.1: Given a set of n rectilinearly oriented rectangles, we can determine the contour in $O(n \log n + p)$ time, using $O(n + p)$ space, where p is the number of edges in the contour.

Proof: The algorithm follows [LiP] except for the use of a contracted segment tree as a supporting data structure.

Step 1: Initially the x- and y-coordinates of all rectangles are sorted separately. After this, rectangle coordinates are identified with their rank in the x- and y-ordering, respectively. From now on all sorting operations can be done by standard bucket sorting in $O(n)$ time. This step takes $O(n \log n)$ time.

Step 2: The empty contracted segment tree is built from the list of y-coordinates. This takes $O(n)$ time, due to Property 2.

Step 3: The vertical sides of all rectangles are ordered lexicographically in $O(n)$ time.

Step 4: Scan the list of vertical sides of rectangles. Insertion or deletion of the y-interval i_j of a rectangle side in the contracted segment tree together with reporting p_j free portions of the line takes $O(n \log n + p_j)$ time, due to Property 1. This yields a total time of $O(n \log n + \sum_{j=1}^{2n} p_j)$. Adjacent parts of the contour counted in p_i and p_{i+1} may be merged into one part, hence in the worst case $p = \sum_{j=1}^{2n} p_j - 2n$. So the total time for step 4 is $O(n \log n + p + 2n) = O(n \log n + p)$. The reported parts of the contour are stored in a list of vertical edges which uses $O(p)$ space.

Step 5: The list of vertical edges is scanned and horizontal edges are linked in (see [LiP]). This step uses $O(p)$ time and space.

All data structures including the contracted segment tree use $O(n)$ space, except for the list of contour edges which uses $O(p)$ space. This establishes the $O(n + p)$ bound. The time bound follows from adding up the time bounds in steps 1 through 5. <>

3.1.3. *The contracted segment tree: structure and operations.*

The structure.

Recall that the inefficiency of the counting segment tree was caused by the fact that to find the free subintervals of interval(p) for any node p in the tree large portions of p's subtree had to be traversed. The idea crucial to our solution is the following: We still traverse p's subtree to find free subintervals. However, by pruning away covered branches and further contraction we make the whole tree so small that any subtree contains only $O(\beta)$ nodes if it contains β disjoint free subintervals. Then the subtree traversal with reporting the β gaps takes only $O(\beta)$ time.

Since we cannot afford to destroy the structure of the tree by pruning away subtrees, we keep two structures: the primary tree which is the segment tree, as described before, and a copy of it which is cut into pieces which are furthermore 'contracted'. Each of these pieces is a tree and corresponds to some subtree of the primary tree. The collection of pieces we call secondary structure. Insertions and deletions are performed on the primary tree, updating the secondary structure, while traversing a subtree to report free subintervals is done on the secondary structure. Primary and secondary structure together we call a contracted segment tree.

In the sequel we describe how to obtain the secondary structure for a given segment tree storing some intervals. This seems to be the easiest way to explain the shape of the secondary structure. In practice, however, the secondary structure is constructed dynamically by insertion and deletion algorithms which we describe later.

So given a counting segment tree T storing some intervals we first produce a perfect copy of it which we call T' (to distinguish between a node p and its copy we also call the copy p'). Next we augment each node of T by a field <u>second</u> which contains a pointer. This field can be used to point to a part of T'. At the moment we initialize all second-pointers to nil. Furthermore let $ROOT_1$ point to T and $ROOT_2$ to T'.

We need a few definitions concerning counting segment trees:

For any node p of a counting segment tree we say it is <u>covered</u> iff cover(p) > 0. The <u>total coverage</u> of p is the sum of the cover-fields of all ancestors of p, including p. For any leaf q in p's subtree representing the atomic interval interval(q) we say interval(q) is <u>free with respect to p</u> if none of the nodes on the path from p to q (excluding p, including q) is covered. This defintion extends in a natural way to subintervals of interval(p) obtained by concatenation of some leaf intervals. In an analogous manner we define subintervals <u>covered with respect to p</u>. We call p's subtree <u>nontrivial</u> if it contains free <u>and</u> covered (with respect to p) subintervals.

To obtain the secondary structure, T' goes through two different contractions. We use an example to illustrate them. Fig. 3-5 (a) shows a segment tree whose nodes i, l and g are covered. Covering is indicated by horizontal bars. A rectangular box represents a leaf storing an interval (which is given below the box).

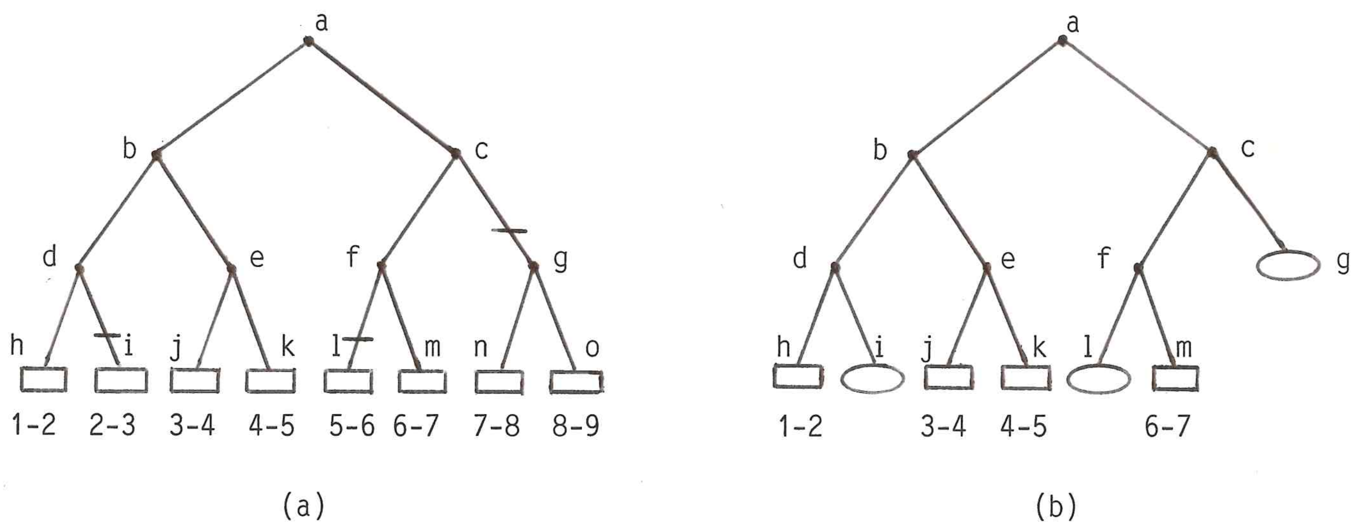


Figure 3-5

The first contraction disconnects any covered subtree from its father and replaces it by a leaf of a new type to represent the removed subtree. Note that this implies that from a removed subtree again some subtrees may be cut off. The whole process leaves us with a collection of fragments of T' each of which is a tree with two different kinds of leaves. To distinguish them we call the leaves representing removed subtrees black leaves, the other ones white leaves. Consider any such tree t with root node r. t represents a subinterval of the line, namely interval(r). Obviously a white leaf of t represents a subinterval of interval(r) which is free with respect to r. A black leaf represents a subinterval covered with respect to r.

Of course the disconnected fragment trees must be accessible in some way. For this purpose we provided the second-fields in the primary tree. If node q' is the root of a disconnected subtree, then we let second(q) point to q'.

Note that now only the fragment of T' connected to $ROOT_2$ may contain any absolutely free intervals (they are given by the white leaves of this tree). Therefore we call this particular subtree of T' the free tree (if the root node of T is covered then the first contraction has set $ROOT_2$ to nil, corresponding to the fact that there exist no free intervals). In Fig. 3-5 (b) only the free tree is shown.

After the first contraction the resulting trees may still be too large. We want to achieve that any fragment tree contains only $O(ß)$ nodes if it has ß free subintervals. But for instance the completely empty tree (having no covered subtrees) contains $O(n)$ white leaves though it represents only one single free interval!

The second contraction therefore compresses any empty subtree into a single white leaf. It also removes 'superfluous' black or white leaves, as we will see. The idea is to leave only nodes in the contracted tree which correspond to a boundary between a free and a covered subinterval. The second contraction compresses a tree 'bottom-to-top', starting from the leaves, according to the following rules:

1. If both sons of a node p are white leaves, replace p by a single white leaf (with the corresponding interval).

2. If both sons are black leaves, replace p by a black leaf.

3. If one son is a white and one a black leaf, create an ordinary leaf (represented by ◨ or ◧) and replace p by it. An ordinary leaf clearly represents a boundary between a free and a covered subinterval. The interval of the white leaf and the ordering black-white or white-black is kept in some way (details follow).

4. If one son of p is a black or white leaf, the other one an inner node or an ordinary leaf, then remove the black or white leaf, respectively, and collapse p with its remaining son (see Fig. 3-6).

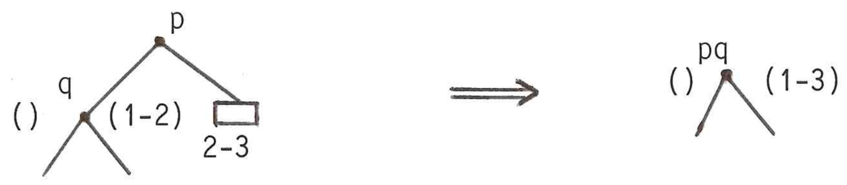


Figure 3-6

Again some information is kept: p's name and the interval of the pruned son of p (if it is a white leaf) is stored with the remaining son (see below).

5. If p is a leaf or if each son is either an inner node or an ordinary leaf, then do nothing.

For the example of Fig. 3-5 (b) we display the second contraction in two steps:

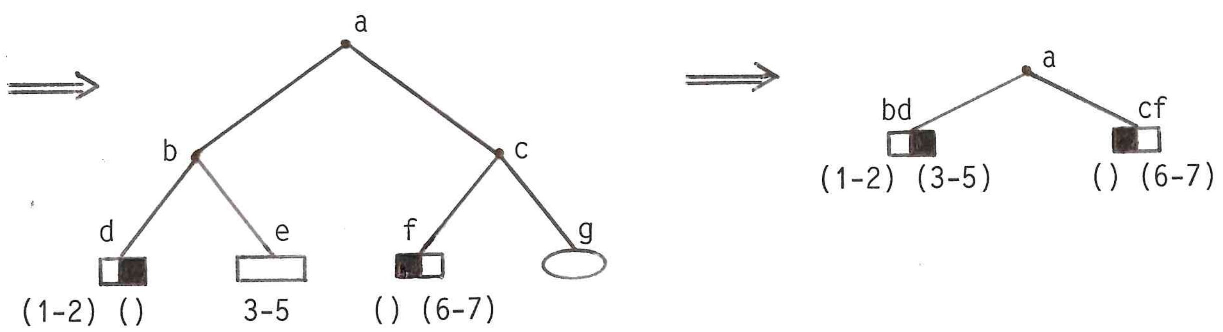


Figure 3-7

In general, after completing the second contraction, each of the resulting trees is either a white or a black leaf or a tree containing only inner nodes and ordinary leaves. This follows from the fact that, according to rules 1. - 4., a black or white leaf does not 'survive' the second contraction iff it has a brother.

To keep some information which would otherwise be lost during the second contraction we augment inner nodes and ordinary leaves by some fields names, leftfree, rightfree and whichwhite (the last one for leaves only) with the following meaning:

- names(p): a list of pairs (node name, type: black/white),

  contains the names of all predecessors of p which have been collapsed with p according to rule 4. For each name a boolean value tells us whether a black or a white leaf was eliminated.

- leftfree(p): a list of intervals,

  contains the intervals of all white leaves eliminated according to rule 4. leftfree contains those intervals for leaves left of p (they were left sons of some node q whose name is now in the list names(p)).

- rightfree(p): a list of intervals,

  contains the corresponding intervals for eliminated white leaves which were right sons.

Lists leftfree and rightfree contain the intervals in sorted order. Applying rule 4, a new interval may be added at the left end of list leftfree or at the right end of list rightfree, respectively. Furthermore it is checked whether an interval added is adjacent with its neighbour interval. In that case the two intervals are replaced by a single one.

Applying rule 3 an ordinary leaf is created. The interval of the white leaf is entered as the first item of the just created list leftfree or rightfree, respectively. The respective other list is initially empty. To keep the information whether the ordinary leaf represents two leaves in order black-white or white-black ordinary leaves have an additional boolean field.

- <u>whichwhite</u>(p): left/right.

We must emphasize again that we used a dynamic description to define the (static) structure of the contracted segment tree (consisting of a primary tree and the secondary structure). In fact, there is no component of our algorithm which takes a counting segment tree as input and constructs a secondary structure from it. The secondary structure is instead constructed by insertion and deletion algorithms which are described below.

We close this section by summarizing some properties of the structure thus obtained.

<u>Lemma 3.2</u>: A contracted segment tree based on a set of n intervals I requires O(n) space. The empty tree can be built in O(n) time from the sorted list of interval endpoints.

<u>Proof</u>: Both statements hold for ordinary segment trees. Since each node in the primary tree appears at most once in the secondary structure, the total space is still O(n). The secondary structure of an empty contracted segment tree

consists of just a single white leaf, which can be constructed in constant time. <>

We are now able to prove the property which we wanted to achieve from the beginning: the contracted copy of a subtree of the primary tree contains only O(ß) nodes if it represents ß disjoint free subintervals. Let the contracted copy be defined for any nontrivial subtree of the primary tree (that is a subtree containing free and covered parts). Let p be the root of such a subtree. Then p appears as a name in the name list of some node q' in the secondary structure. The contracted copy of p's subtree T(p') consists essentially of the subtree rooted in q'. However T(p') includes only the inner parts of the interval lists leftfree(q) and rightfree(q), that is, those intervals which belong to names following p in the name list (because the other intervals are not contained in p's subtree). In Section 3.1.4 it is shown how to obtain the reduced interval lists of T(p') before T(p') is traversed for reporting.

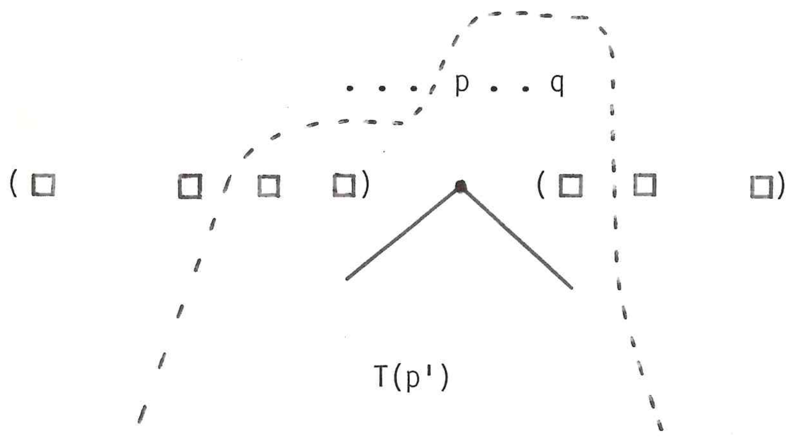


Figure 3-8

Lemma 3.3: Given a contracted segment tree, let p be any node of the primary tree with a nontrivial subtree and let ß be the number of disjoint intervals of T(p) which are free with respect to p. Then T(p') has O(ß) nodes and contains O(ß) intervals in its interval lists.

Proof: Since p has a nontrivial subtree T(p), T(p') is a tree consisting of inner nodes and ordinary leaves. Each ordinary leaf represents a boundary between a free (with respect to p) and a covered interval. Since T(p) contains ß disjoint free subintervals it may contain at most (ß+1) covered subintervals. Between all (2ß+1) disjoint intervals exist 2ß boundaries and therefore at most 2ß ordinary leaves. Hence the total number of nodes in T(p') is O(ß).

By construction of the interval lists, the intervals within any single list are disjoint. However, the last interval of a list and the first interval of the 'subsequent' list may be adjacent. In Fig. 3-9 some pairs of list entries which represent possibly adjacent intervals are shown.

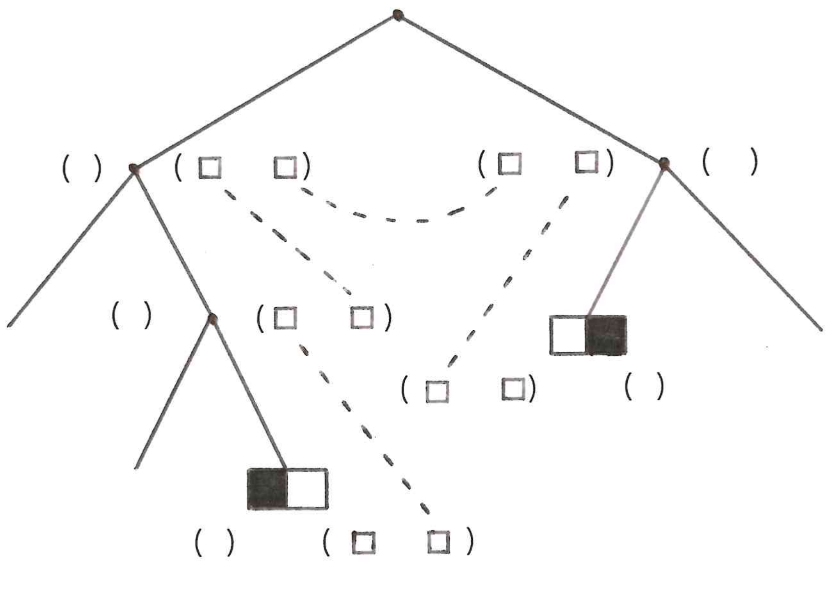


Figure 3-9

So we cannot be sure that the first or last item in a list represents an independent free subinterval. However, since $T(p')$ contains only $O(\beta)$ nodes and interval lists, there may exist at most $O(\beta)$ 'superfluous' list items. Hence the total number of intervals in interval lists is $O(\beta)$. <>

Operations insertion-query and deletion-query.

We now describe how to perform on this structure the two operations in which we are interested, namely insertion-query and deletion-query. There exists a standard algorithm for insertion and deletion of an interval in a counting segment tree (see for instance [vLW]). We apply this algorithm to the primary tree and augment it by operations which are performed simultaneously on the secondary structure.

Algorithm INSERTION-QUERY

Let i be the interval to be inserted and p a node of the primary tree. There exists a current position in the secondary structure which is moved by the algorithm. An INSERTION-QUERY is started by initializing this position to $ROOT_2$ and then calling INSERT(i, $ROOT_1$). Let λp and ρp denote p's left and right son, respectively.

```
INSERT(i, p) = if interval(p) ⊆ i
               then cover(p) ← cover(p) + 1
                    if cover(p) = 1  then LINK-OUT; REPORT fi
               else if interval(λp) ∩ i ≠ ∅
                    then LEFTDOWN
                         INSERT(i, λp)
                         UP
                    fi
                    if interval(ρp) ∩ i ≠ ∅
                    then RIGHTDOWN
                         INSERT(i, ρp)
                         UP
                    fi
               fi
```

<u>end of algorithm</u> INSERTION-QUERY.

Note that the algorithm is controlled by the fixed structure of the primary tree. Apart from the operations LINK-OUT, REPORT, LEFTDOWN, RIGHTDOWN and UP it is the normal insertion algorithm of the segment tree. Those operations manipulate the secondary structure.

The deletion algorithm is obtained from the insertion algorithm by replacing

```
        cover(p) ← cover(p) + 1
        if cover(p) = 1 then LINK-OUT; REPORT fi

                  by

        cover(p) ← cover(p) - 1
        if cover(p) = 0 then LINK-IN; REPORT fi
```

and every occurrence of 'INSERT' by 'DELETE'.

The strategy of the traversal of the secondary structure is to expand it on the way down along the search path, that is to make the second contraction locally undone. This is done by LEFTDOWN and RIGHTDOWN. LINK-OUT cuts off a subtree. More precisely, it replaces it by a black leaf and connects it to some second-pointer of the primary structure. The deletion algorithm uses the inverse operation LINK-IN. LINK-IN and LINK-OUT together perform the first contraction. UP recontracts the structure on the way up according to rules 1. through 5., that is, it performs the second contraction. REPORT traverses the current subtree to report all free subintervals provided this subtree is part of the free tree (see above).

In Section 3.1.4 these operations are described and the following properties are shown:

- Each of the operations LEFTDOWN, RIGHTDOWN, UP, LINK-IN and LINK-OUT takes only constant time (Lemma 3.5).
- The operation REPORT takes $O(1 + \beta)$ time if it is applied to a subtree with β disjoint (absolutely) free subintervals (Lemma 3.6).

Based on these properties we can prove:

Theorem 3.4: On a contracted segment tree based on a set of n intervals I each of the operations insertion-query and deletion-query takes $O(\log n + \gamma)$ time where γ is the number of subintervals which are to be reported.

Proof: As mentioned before, the insertion or deletion of an interval i in a counting segment tree - as defined by the algorithm above - is restricted to a forked path:

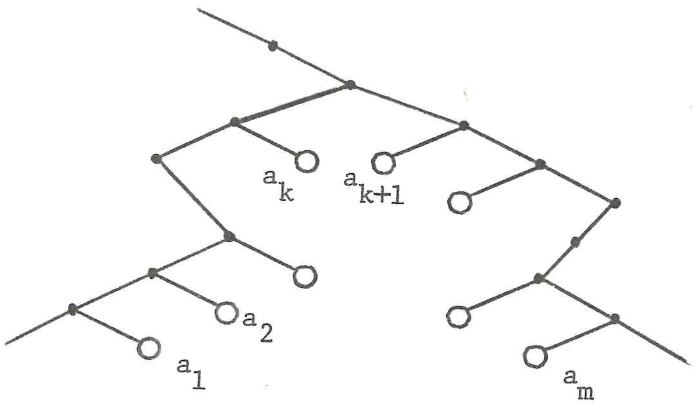


Figure 3-10

The total number of nodes on this path is $O(\log n)$. The algorithm visits the nodes $a_1 \dots a_m$ in this order. These nodes form together the set CN(i). Apart from calls of REPORT the algorithm clearly takes time $O(\log n)$, since the operations LEFTDOWN, RIGHTDOWN, UP, LINK-IN and LINK-OUT take only constant time per node.

For each of the $O(\log n)$ calls of REPORT the work is $O(1 + \beta_i)$, for $i = 1, \dots, m$. So the total work for these calls is $O(\log n + \sum_{i=1}^{m} \beta_i)$. Note that, since adjacent gaps in the subtrees of a_i and a_{i+1} are merged and only once counted in γ, we have

$$\gamma = \sum_{i=1}^{m} \beta_i - O(\log n).$$

It follows that the total work is

$$O(\log n + \sum_{i=1}^{m} \beta_i) = O(\log n + \gamma + \log n) = O(\log n + \gamma). \qquad \diamond$$

3.1.4. *Traversing and manipulating the secondary structure.*

The idea.

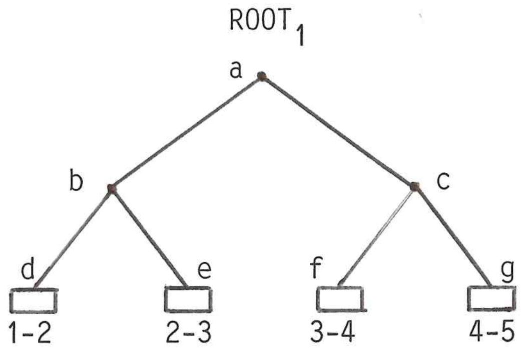
Before we describe the traversal of the secondary structure in detail, let's try to gain an overall understanding of it. The algorithm works recursively on the primary structure. Whenever it changes position there, it calls LEFTDOWN, RIGHTDOWN or UP to perform the appropriate movement in the secondary structure. (We assume there always exists one well defined current position in the secondary structure which is initially the root).

Since an inner node in the primary structure may be represented as a name or a leaf in the secondary structure, it is not obvious, how to go down from it. Our strategy is to expand names and leaves into inner nodes going down and to recontract the structure on the way up. Let's look at two examples:

Example 1: Interval [3, 4] is inserted into the empty tree

(We show the changes of the secondary structure. The current position is indicated by ∇):

Primary Tree



Secondary Structure

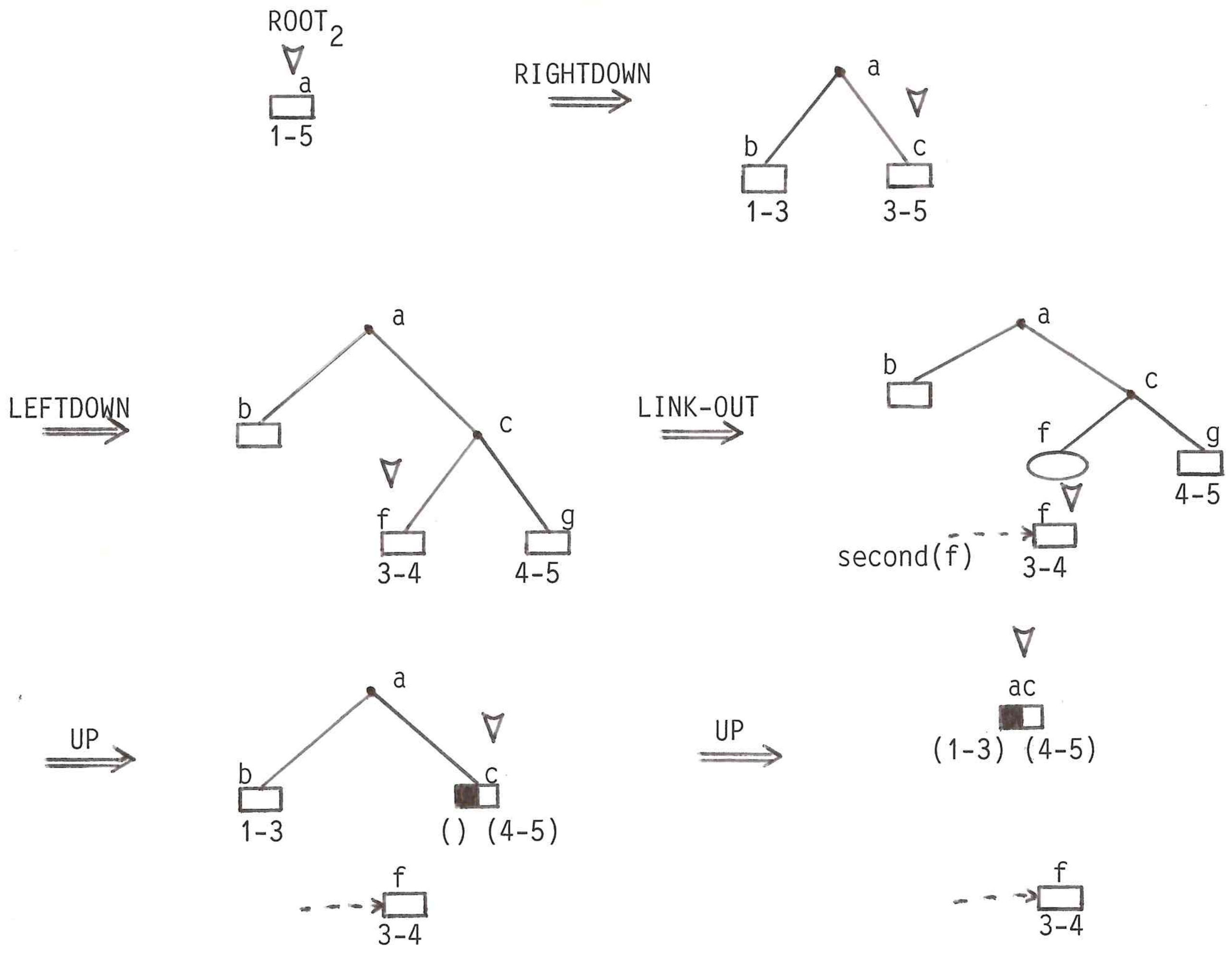


Figure 3-11

Example 2: Interval [3, 5] is deleted from the tree:

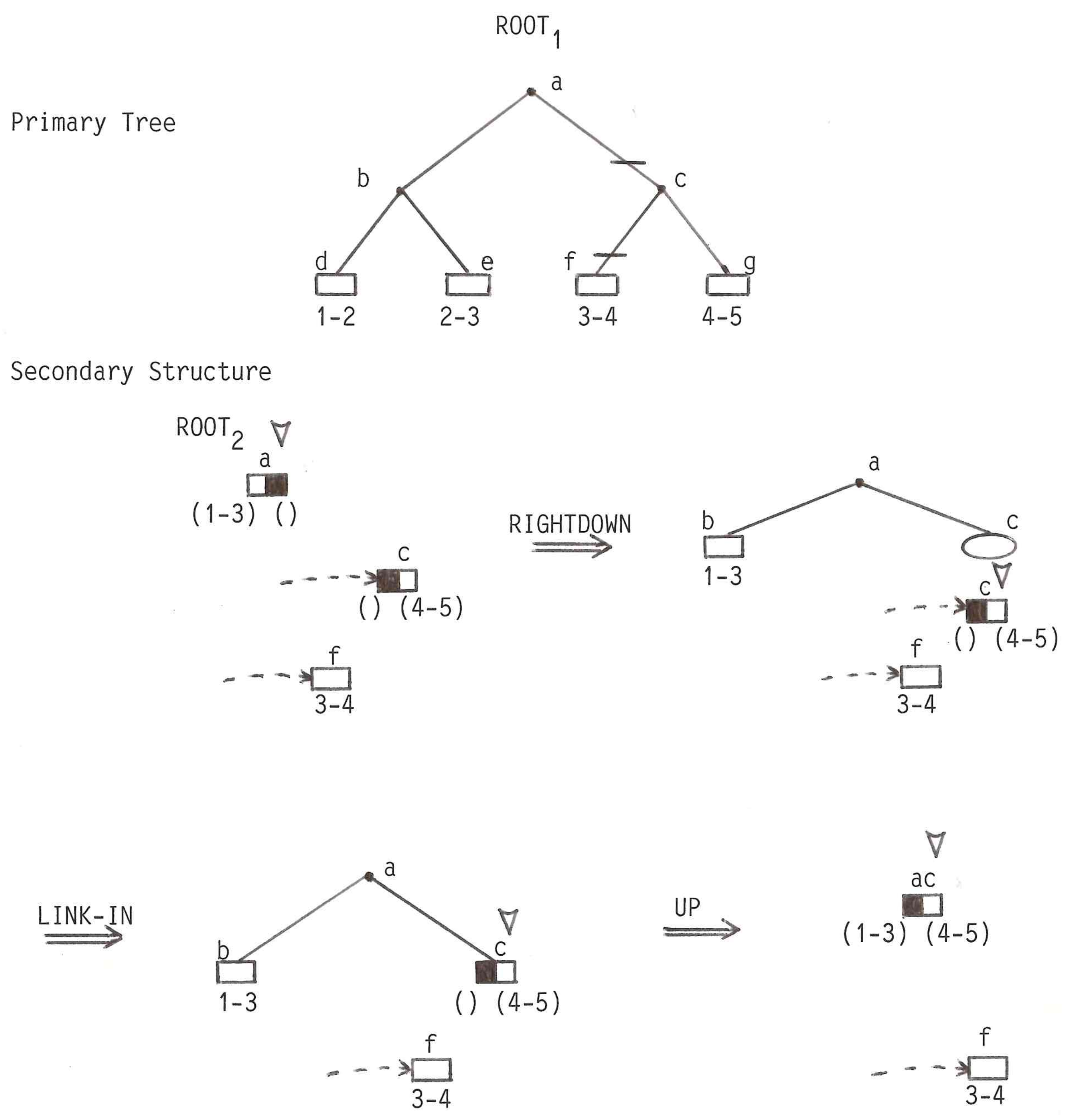


Figure 3-12

We omitted the calls of REPORT since they do not change the structure.

Observe once more the tasks of the different operations:

- LEFTDOWN, RIGHTDOWN move the current position from a node to its son, expanding names and leaves into inner nodes. These operations remove the second contraction along the search path.
- LINK-OUT cuts off a subtree, replaces it by a black leaf and connects the subtree to the primary tree. In this way it performs the first contraction.
- LINK-IN performs exactly the inverse operation, removing the first contraction.
- UP returns to the root performing the second contraction according to rules 1. through 5.

In the sequel we describe these operations in a little more detail and make some remarks concerning implementation.

Going down (LEFTDOWN, RIGHTDOWN).

Whenever one of these operations is called then the current position is one of the following:

- an inner node
- an ordinary leaf
- a white leaf
- a black leaf
- a name in a name list.

The current position is the address of a node in the secondary structure. If this node has a name list we assume the current position to refer to the first name in the list. Therefore if we say the current position is a name, we imply that the name list contains more than one element. If it has only one element then the current position is an inner node or an ordinary leaf.

Going down consists of two steps:

1. Expanding the current position into an inner node, if necessary.
2. Moving the current position to the son.

Expanding.

The rules for expanding are obviously inverse to the contraction rules 1. through 5. of Section 3.1.3. Let the current position be a node or name p.

1. If p is a white leaf, replace it by an inner node with two white leaves as sons (the intervals of those leaves can be seen from the primary tree).

2. If p is a black leaf, replace it by an inner node with two black leaves as sons.

3. If p is an ordinary leaf, replace it by an inner node with a white leaf and a black leaf as sons. Remove the interval of the white leaf from the interval list of p.

Example:

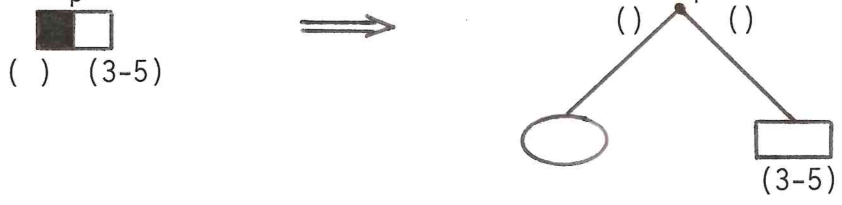


Figure 3-13

4. If p is a name in the name list of some node q (which implies that it is the first name and not the only one) then expand p into an inner node. Let one of the sons be a black or white leaf, according to the type stored in the name list. Remove name p from the name list of q. If p was expanded with a white leaf then remove also the interval of that leaf from the list leftfree or rightfree, respectively. By construction it must be (part of) the first item of leftfree or the last item of rightfree.

Example:

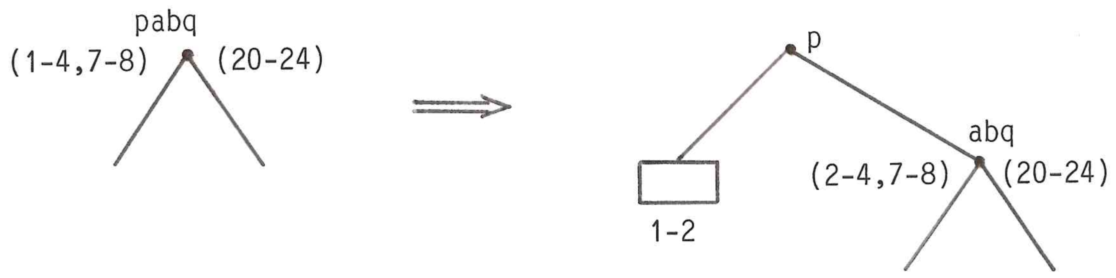


Figure 3-14

5. If p is an inner node, do nothing.

Moving the current position.

At this point we can be sure that the current position is an inner node. We then move the position to the son. If the son is an inner node or ordinary leaf then the new position refers to the first name in its name list. If the son is a white leaf, no problem arises. If it is a black leaf then we change the position by jumping to another part of the secondary structure, namely to second(λp) for LEFTDOWN or second(ρp) for RIGHTDOWN, respectively.

To be able to return along the search path we apply the usual technique of storing the position we leave on a stack. Each stack element consists of two components, however: the address and a boolean field free which tells us whether the position stored belongs to the free tree (see Section 3.1.3). This enables the operation REPORT to decide, whenever it is called, whether the subtree belonging to the current position may contain absolutely free subintervals or not.

Going up (UP).

There is not much to be said about going up because it is exactly the inverse operation to going down. Clearly it consists of the steps:

1. Moving the current position to the father.
2. Contracting the father according to the contraction rules 1. - 5.

Moving the position to the father is no problem since we pushed the father's address on the stack when going down. Contracting is then performed according to rules 1. - 5. of Section 3.1.3.

Moving subtrees (LINK-OUT, LINK-IN).

These two operations together perform and remove the first contraction (see Section 3.1.3).

LINK-OUT: The subtree rooted in the current node is cut off from its father p. That is, the father receives a black leaf as a son instead of the subtree. Furthermore second(λp) or second(ρp) is put on the current position.

Example:

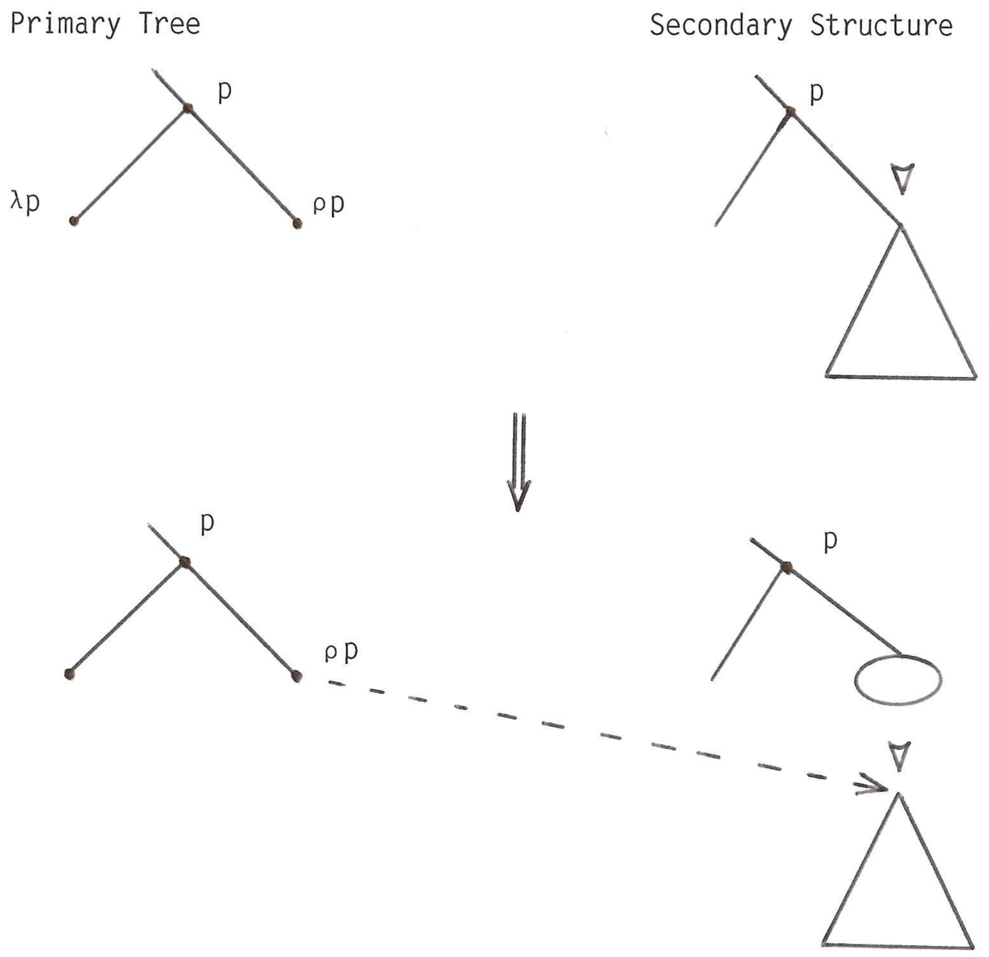


Figure 3-15

LINK-IN: inverse to LINK-OUT.

Lemma 3.5: Each of the operations LEFTDOWN, RIGHTDOWN, UP, LINK-OUT and LINK-IN takes only constant time.

Proof: This is easy to check from the description given in the corresponding subsections. All actions performed are local, that is restricted to the direct environment of the current position. <>

Reporting free subintervals (REPORT).

Let q be the node of the current position at the moment REPORT is called. REPORT first checks whether the current position belongs to the free tree and may therefore contain any absolutely free intervals (from the top stack element it can be seen whether the father belongs to the free tree etc.) If it does, REPORT calls a recursive algorithm report-intervals(q) to report all intervals contained in interval lists of q's subtree. The argument of report-intervals is a node of the secondary structure. It can be shown that report-intervals is never called for a black leaf.

```
REPORT = if 'q belongs to the free tree'
         then report-intervals(q) fi

report-intervals(p) =

         if 'p is a white leaf'
         then 'output p's interval'
         else if 'p is an ordinary leaf'
              then output all intervals in leftfree(p)
                   output all intervals in rightfree(p)
              else* p is an inner node*
                   output all intervals in leftfree(p)
                   report-intervals ($\lambda$p)
                   report-intervals ($\rho$p)
                   output all intervals in rightfree(p)
              fi
         fi.
```

In Section 3.1.3 (see Fig. 3-9) we have seen that different interval lists may contain adjacent intervals. Therefore we merge these intervals during reporting by always keeping the last previous interval and checking for adjacency.

Lemma 3.6: The operation REPORT takes $O(1 + \beta)$ time applied to a subtree with β disjoint (absolutely) free subintervals.

Proof: Obviously REPORT works in constant time if the subtree does not belong to the free tree (and therefore $\beta = 0$) or if it consists of a single white leaf ($\beta = 1$). Otherwise report-intervals is called for an inner node or ordinary leaf q. Let p be the first name in q's name list. It is crucial to note that now the tree with root q is precisely the tree T(p') defined in Section 3.1.3 and refered to in Lemma 3.3. The removing of superfluous intervals from q's interval lists (see Fig. 3-8) has been performed by LEFTDOWN and RIGHTDOWN. The work of report-intervals consists of visiting all nodes and scanning all interval lists. We may now apply Lemma 3.3 and conclude that the total work for this is $O(\beta)$ if β disjoint free intervals are reported. Note that the name lists are not scanned during reporting (which might spoil the time bound). <>

Final remarks.

With Lemma 3.6 we completed the proof of Theorem 3.4 which in turn completes the proof of Theorem 3.1. This finishes our description of an algorithm which computes the contour of a set of n iso-oriented rectangles in $O(n \log n + p)$ time and $O(n + p)$ space where p is the number of contour-pieces.

In Section 3.2 we study some related contour and measure problems. The contracted segment tree or variants of it will again prove useful. In Section 3.2.9 we prove lower bounds for all the problems considered including the one of this section. We show that $\Omega(n \log n + p)$ is also a lower bound for computing the contour. Hence our algorithm is time-optimal. We do not know whether it is space-optimal (see the remarks in Section 3.2.9).

In Section 4.4 we give a divide-and-conquer algorithm to compute the contour which achieves the same time bound, but has higher space complexity.

3.2. *I-contour and i-measure problems.*

3.2.1. *Problem description.*

We now study problems based on a generalization of the contour concept due to Wood [Wo]. Imagine each of the rectangles of a given set R to have a certain small 'thickness'. Laying out the intersecting rectangles in the plane then results in something like a 'hill landscape'; areas where many rectangles intersect are elevated. Obviously separate regions are formed on different elevation levels which have their own contours and measures. - Problems based on these concepts may have applications in VLSI-design, see the conclusions.

Throughout Section 3.2 we assume that all rectangles have distinct coordinates (that is a set of n rectangles defines 2n distinct x- and y-coordinates, respectively). This restriction allows us to present our algorithms free from unnecessary detail. We do not expect the adaptation to the general case to be a problem; in Section 4 we give some examples of how to modify an algorithm to obtain a solution for the general case.

We now formalize the 'hill landscape' concept:

Given a set of rectangles R in 2-space,

$$\underline{\text{i-area}(R)} := \{p \in \mathbb{R}^2 \mid \text{there are exactly } i \text{ rectangles in } R \text{ which contain } p\}.$$

This induces the notions of i-measures, i-contours and the height of a set of rectangles:

<u>i-measure(R)</u>:= measure(i-area(R))

<u>i-contour(R)</u>:= $\{p \in \mathbb{R}^2 \mid$ p is on the boundary of (i-1)-area and of i-area$\}$

where 'on the boundary' is defined as follows:

Let S, T be subsets of $\mathbb{R}^2$. $p \in \mathbb{R}^2$ is <u>in the closure of S</u> iff any circle around p with radius $r > 0$ contains points of S. p is <u>on the boundary of S and T</u> if it is in the closure of S and of T.

In this terminology the contour considered in the last section is called the 1-contour. Again any i-contour is a collection of contour-cycles each of which is a sequence of alternating horizontal and vertical contour-pieces.

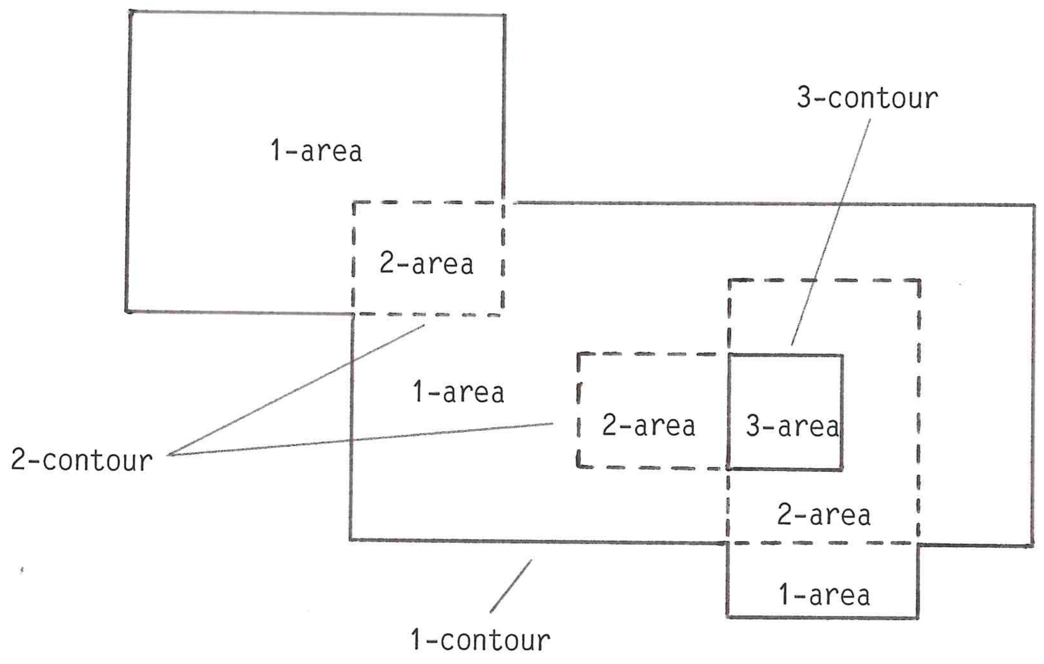


Figure 3-16

Finally we define

$$\underline{\text{height}(R)} := \max \{ i \in \mathbb{N}_0 \mid \text{there exists an } i\text{-area} \neq \emptyset \}.$$

It is easy to see that now $\text{measure}(R) = \sum_{i=1}^{\text{height}(R)} i\text{-measure}(R)$.

In the following sections we consider algorithmic solutions for the following problems:

- Compute height(R).
- Compute the height-measure and the height-contour (that is measure and contour for the area of highest elevation).
- Compute <u>all</u> i-measures and i-contours, respectively.
- Compute the i-measure (or the i-contour) for a given value of i. We will see that this is the most difficult task.

3.2.2. *Computing the height of a set of rectangles.*

We approach the task again with a line-sweep algorithm in mind. At any position the intersection of the sweeping line with the set of rectangles is a set of intervals. The set of intervals partitions the line into a set of fragments. Now we can associate with each fragment an 'elevation level':

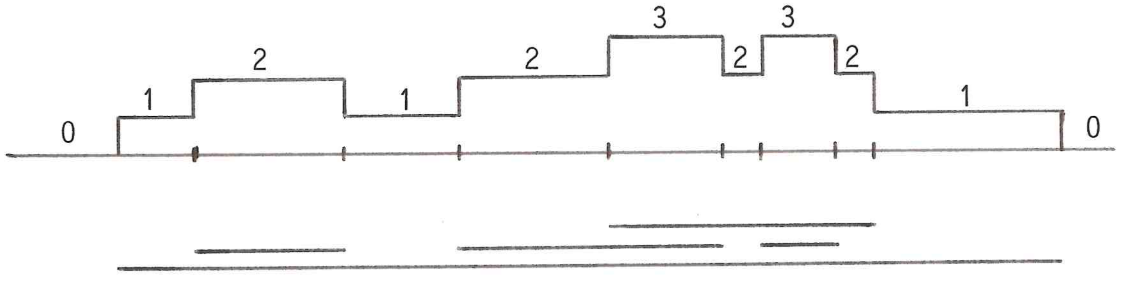


Figure 3-17

During the line-sweep the set of intervals changes by insertions and deletions and the set of fragments changes accordingly. Clearly the height of the set of rectangles is the maximum, taken over all sweepline positions, of the maximal elevation level occurring in a particular set of fragments (for a fixed sweep-line position).

We model the set of fragments in an abstract data structure called a weighted partition defined as follows (we use some terms defined in Section 2):

Let I be a set of intervals based on a set of points P. Then

$\underline{\text{weighted partition (I, P)}} := \{(i_y, c) \mid i_y \in \text{partition}(P)$ and

there exist exactly c intervals in I which contain $i_y\}$.

For this data structure two operations, namely insertion and deletion of an interval, are defined. Let W be weighted partition (I, P), i an interval based on P.

$W.\text{insert}(i) = \underline{\text{for all}}\ (i_y, c)$ in W $\underline{\text{with}}\ i_y \subseteq i : c \leftarrow c + 1.$

$W.\text{delete}(i) = \underline{\text{for all}}\ (i_y, c)$ in W $\underline{\text{with}}\ i_y \subseteq i : c \leftarrow c - 1.$

Furthermore a function maxc yields the maximal c-value occurring in W:

$W.\text{maxc} = \max(\text{proj}_2(W))$

(proj_2 yields the set of all c-values occurring in W, of which the maximum is taken).

Now it is easy to give an algorithm computing the height of a set of rectangles:

```
Algorithm HEIGHT

Input:    A set of rectangles R.
Output:   height(R), an integer value.

Step 1:   Let VX be the set of all vertical rectangle edges, sorted by x. Let
          P = y-set(R).

          W ← $\{(i_y, 0) \mid i_y \in \text{partition}(P)\}$
          h ← 0

Step 2:   Scan VX. Encountering

          a) a left edge $(x, y_1, y_2)$

             W.insert($[y_1,y_2]$)
             if W.maxc > h then  h ← W.maxc  fi

          b) a right edge $(x, y_1, y_2)$

             W.delete($[y_1,y_2]$)

Step 3:   HEIGHT ← h.

end of algorithm HEIGHT.
```

Of course we have not really solved the problem without giving an implementation of the abstract data structure weighted partition. However, this is not a difficult task: We use basically a counting segment tree (described in Section 3.1). Each element (i_y,c) of the weighted partition corresponds to a leaf p of the counting segment tree with represented interval interval(p) = i_y. The 'elevation level' (or coverage value) c is given by the sum of the cover-fields of all nodes on the path to p.

Creating weighted partition ($\emptyset$, P) is implemented by building an empty counting segment tree based on P. It is obvious that the normal insertion and deletion procedures of counting segment trees maintain a weighted partition correctly if c-values are computed as described above. We only have to add a mechanism implementing the function maxc.

To achieve this we augment each node p by an integer field maxcover(p), containing the highest coverage-value on any path below p.

$$\text{maxcover}(p) = \begin{cases} \text{cover}(p) & \text{if } p \text{ is a leaf} \\ \text{cover}(p) + \max\{\text{maxcover}(\lambda p), \text{maxcover}(\rho p)\} & \text{otherwise.} \end{cases}$$

Maxcover-fields are initially 0. Since a node's maxcover-value depends only on the maxcover-values of its sons (and its own cover-value) it is possible to update after an insertion or deletion of an interval I the maxcover-fields of all nodes on the forked search path, going up the tree, without increasing the O(log n) update time. A function call W.maxc is implemented by an access to maxcover(root).

Lemma 3.7: A weighted partition based on a set of n points can be stored in O(n) space with an update time of O(log n) and constant time for an evaluation of function maxc. The empty weighted partition can be built (from a sorted set of points) in O(n) time.

This immediately determines the time and space complexity of algorithm HEIGHT:

Theorem 3.8: Given a set of n rectangles R, height(R) can be computed in O(n log n) time and O(n) space.

3.2.3. *Computing the height-measure.*

This can be done by a simple modification of the previous algorithm. Given R, we divide the plane into a set of disjoint vertical stripes defined by the set of all sweepline positions. Let $h = \text{height}(R)$. Each stripe S_i defined by adjacent sweepline positions (x_i, x_{i+1}) may contain some part of the h-area which we call h-area$_i$. The size of h-area$_i$ is the product of the total length of fragments of the sweepline with coverage h and the x-extension of S_i, $(x_{i+1} - x_i)$.

Hence we have to maintain during the line-sweep the total length of the parts of the sweepline which are covered h times after each update. On the abstract level we add to the weighted partition data structure a function maxlength defined as follows:

$$W.\text{maxlength} = \sum_{\{(i_y, c) \in W \mid c = W.\text{maxc}\}} |i_y|$$

This is the modified algorithm:

Algorithm H-MEASURE

Input: A set of rectangles R.
Output: For $h = \text{height}(R)$, h-measure(R), a real number.

```
Step 1:   $h \leftarrow \text{HEIGHT}(R)$

Step 2:   Let VX be the set of all vertical rectangle edges, sorted by x.
          Let  $P = \text{y-set}(R)$.

          $W \leftarrow \{(i_y, 0) \mid i_y \in \text{partition}(P)\}$
          $sum \leftarrow 0$
```

```
Step 3:   Scan VX

          if encountering a left edge $(x, y_1, y_2)$
          then W.insert($[y_1,y_2]$)
          else  *a right edge $(x, y_1, y_2)$ is encountered*
                W.delete($[y_1,y_2]$)
          fi
          $x' \leftarrow$ the next sweepline position. If there is none, goto step 4.
          if W.maxc = h
          then sum $\leftarrow$ sum + W.maxlength $\cdot$ $(x' - x)$
          fi

Step 4:   H-MEASURE $\leftarrow$ sum

end of algorithm H-MEASURE.
```

We complete the description of the algorithm by giving an implementation of function maxlength of a weighted partition. We proceed in the same manner as in the implementation of function maxc. Each node of the counting segment tree is augmented by a further field maxlength. Again after each insertion or deletion the maxlength-fields of nodes on the search path are updated, going up the tree, and a function call W.maxlength is implemented by an access to maxlength(root). Maxlength-fields have the following meaning: For any node p in the tree, maxlength(p) contains the total length of subintervals of interval(p) which are covered maximally with regard to p's subtree, namely maxcover(p) times.

It is easy to see that the correct value of the maxlength-field of a node p can be computed from the fields of its sons λp and ρp (which is crucial for efficient updating) after insertions or deletions:

```
maxlength(p) = if p is a leaf
               then  $|\text{interval}(p)|$
               else  *p is an inner node*
                     if maxcover($\lambda$p) = maxcover($\rho$p)
                     then maxlength($\lambda$p) + maxlength($\rho$p)
                     else maxlength($\mu$p)
                     fi
               fi
```

where μp is the son of p with higher coverage:

```
($\mu$p = if maxcover($\lambda$p) > maxcover($\rho$p)
      then  $\lambda$p
      else  $\rho$p
      fi).
```

It is clear that the updating of maxlength-fields on the search path does not increase the time bound for an insertion or deletion, which is still $O(\log n)$. Accessing maxlength(root) takes only constant time. Hence the time complexity of steps 2 through 4 of algorithm H-MEASURE is the same as that of steps 1 through 3 of algorithm HEIGHT, namely $O(n \log n)$. Calling HEIGHT in step 1 takes additional time of $O(n \log n)$. The space bound is unchanged. Hence it follows:

Theorem 3.9: Given a set of n rectangles R of height h, h-measure(R) (the size of the area of highest elevation) can be computed in $O(n \log n)$ time and $O(n)$ space.

For simplicity we used two different scans to compute h = height(R) and afterwards the h-measure. It is no problem, however, to compute both numbers in a single scan by resetting sum to zero whenever a new maximal height-value is found during the scan.

3.2.4. *Computing the height-contour.*

Our task is now to determine the contour of the area of highest elevation. Recall how the 1-contour was found (Section 3.1): Whenever the sweepline encounters the left edge (x, y_1, y_2) of a rectangle, the interval $[y_1,y_2]$ is inserted into a data structure storing the currently present rectangles represented by their y-intervals. Furthermore the intersection of $[y_1,y_2]$ with (previously) free parts of the sweepline is reported as a collection of contour-pieces. In a similar manner contour-pieces are reported on deletion of an interval.

For h = height(R) those parts of a left rectangle edge contribute to the h-contour which intersect parts of the sweepline previously covered (h-1) times or covered h times now, respectively. Similarly the intersection of a right rectangle edge with the parts of the sweepline covered h times before encountering this edge or (h-1) times afterwards has to be reported as a set of contour-pieces.

On an abstract level it is easy to come up with an algorithm exploiting these facts. We add another function to our weighted partition data structure. Given an integer h and an interval i, W.maxinterval(i) yields a set of disjoint intervals containing the intersection of i with parts of the sweepline covered maximally (assuming the sweepline is represented by a weighted partition W). This can be described formally as follows:

Let W = weighted partition (I, P) and i an interval based on P.

```
W.maxintervals(i) =

    CPF ← {$i_y$ | there exists ($i_y$, c) ∈ W with $i_y \subseteq i$ and c = W.maxc}
                  (CPF = Contour Piece Fragments)
    h-intervals ← union(CPF).
```

The first operation yields the y-intervals of all fragments in W which are covered maximally. Since a single vertical contour-piece may extend over some fragments, the union operation concatenates adjacent fragment intervals.

As we know from [LiP] it is sufficient to compute the set of vertical contour-pieces, since from it the complete contour (as a collection of linked contour-cycles) can be computed in O(p) time and space, where p is the size of the contour. We used this fact already in Section 3.1. Hence our contour algorithm is complete:

<u>Algorithm</u> H-CONTOUR

<u>Input</u>: A set of rectangles R.
<u>Output</u>: For h = height(R), the set of vertical contour-pieces of the h-contour of R.

```
Step 1:   h ← HEIGHT(R)

Step 2:   Let VX be the set of all vertical rectangle edges, sorted by x.
          Let P = y-set(R).

          W ← {($i_y$,0) | $i_y$ ∈ partition(P)}
```

```
Step 3:   Scan VX. Encountering

          a) a left edge $(x, y_1, y_2)$

             W.insert $([y_1,y_2])$
             if W.maxc = h then C ← W.maxintervals $([y_1,y_2])$ else C ← ∅ fi
             output (line-segments(x, C))

          b) a right edge $(x, y_1, y_2)$

             if W.maxc = h then C ← W.maxintervals $([y_1,y_2])$ else C ← ∅ fi
             W.delete $([y_1,y_2])$
             output (line-segments(x, C))

end of algorithm H-CONTOUR.
```

Of course the real difficulty is to implement the weighted partition W in a way that allows efficient computation of function maxintervals(i). Fortunately we do not have to develop a completely new data structure, but can use a modification of the contracted segment tree, called an inverted contracted segment tree.

Recall that a contracted segment tree (CST) consists of a primary tree (which is a counting segment tree) and a secondary structure which is a collection of trees. Each of the secondary trees is the contracted copy of some subtree of the primary tree. The tree attached to $ROOT_2$ contains exactly the free intervals of the line.

An inverted contracted segment tree (ICST) consists again of a primary tree and a secondary structure. The primary tree is a counting segment tree with additional maxcover-fields, as used in Section 3.2.2. The secondary tree attached to $ROOT_2$ now represents the parts of the line maximally covered.

To obtain this structure insertion and deletion procedures of the CST have to be modified. Since the basic concepts are the same as before we will not describe the modified version in as much detail as in Section 3.1. However, we will try to clarify the main differences.

The basic idea of a CST is to 'link out' a subtree when its root node becomes covered. It can be illustrated by the following figure (node q becomes covered by insertion of an interval):

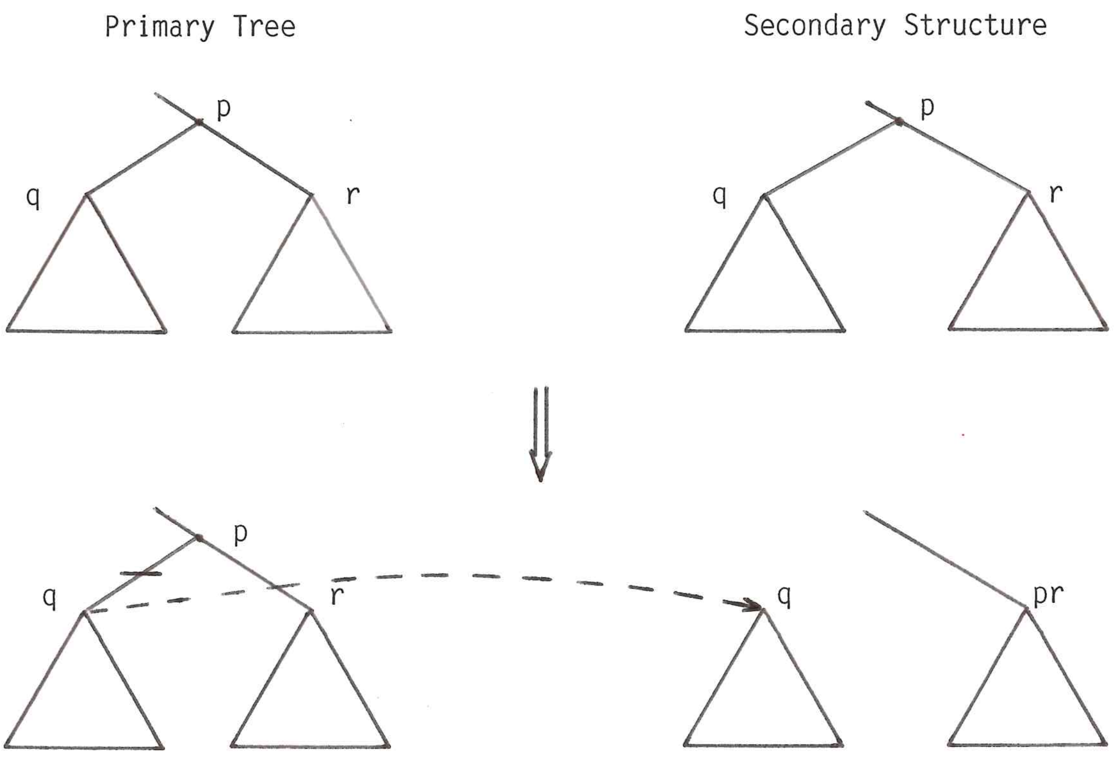


Figure 3-18

In an ICST, of two brother nodes the one with higher maximal coverage remains in the contracted (secondary) structure, the other one is linked out. Both sons are present if they have equal maximal coverage. Hence the result of covering q in the example above is the following:

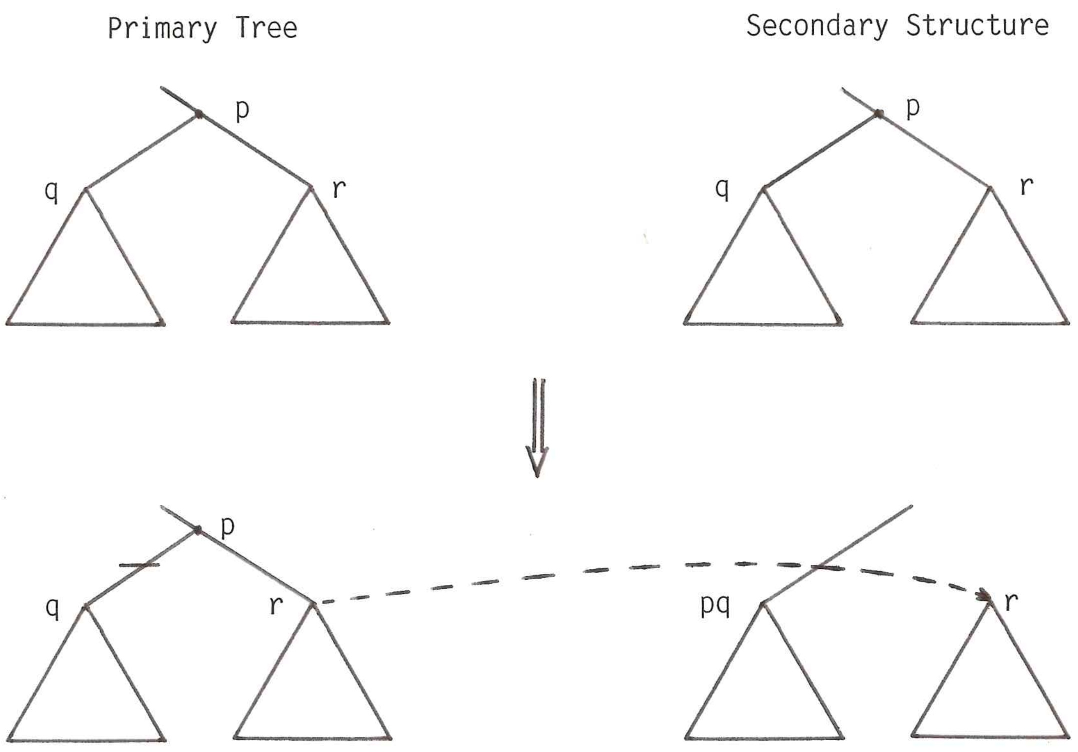


Figure 3-19

Recall that two kinds of contractions are performed to obtain secondary trees of a CST (and as well an ICST). The easiest way to illustrate them is to show how a (static) counting segment tree is transformed into its contracted copy though the secondary structure is constructed dynamically by insertion and deletion procedures. The following example shows the two contractions for ICSTs.

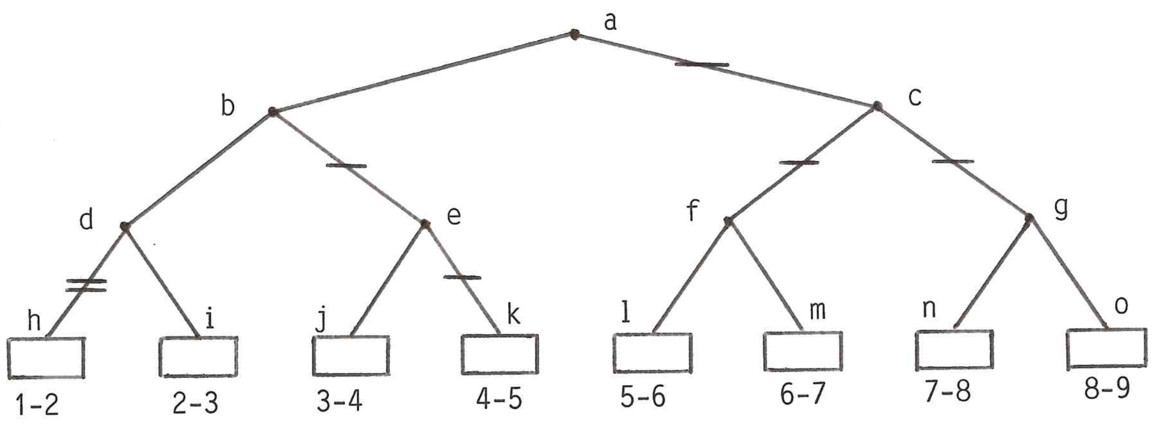


Figure 3-20

Covering is indicated by horizontal bars. The first contraction removes subtrees not maximally covered and replaces them by black leaves:

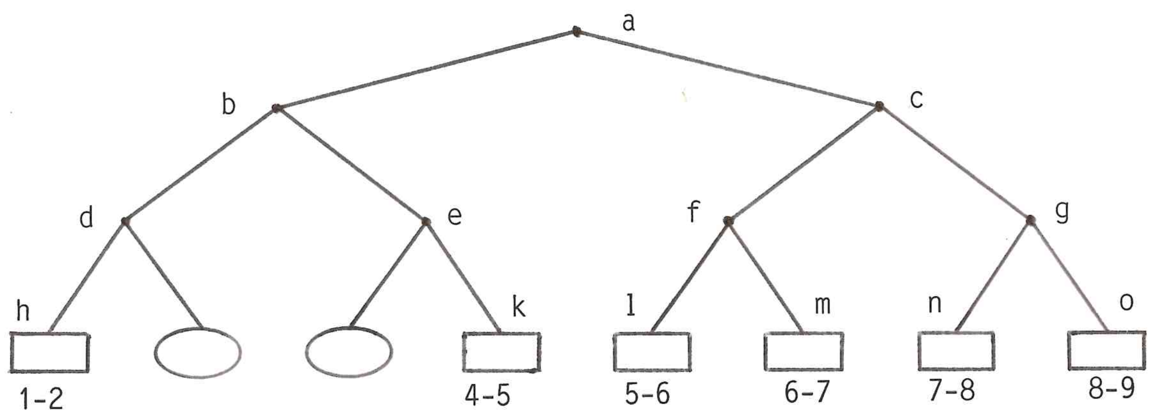


Figure 3-21

The second contraction merges adjacent intervals of the remaining tree to fulfil the condition that a contracted tree contains only O(ß) nodes if it represents ß disjoint intervals. It also creates ordinary leaves at the boundary of covered and free parts (that is from a white and a black leaf) etc. The rules for the second contraction are given in Section 3.1.3. The result is shown in Fig. 3-22:

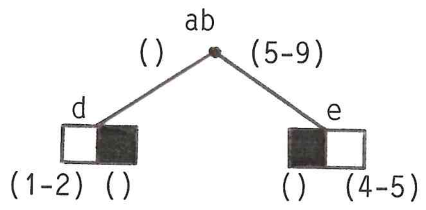


Figure 3-22

These contractions are performed dynamically by algorithms INSERT and DELETE. To remain consistent with algorithm H-CONTOUR we now use a separate algorithm

QUERY to implement the function maxintervals of a weighted partition.

In the sequel we give the modified algorithms INSERT, DELETE and QUERY to maintain and query an ICST. INSERT and DELETE use operations LEFTDOWN, RIGHTDOWN and UP. LEFTDOWN and RIGHTDOWN are unchanged from Section 3.1.4. UP now combines the tasks of the previous operations LINK-OUT, LINK-IN and UP, that is it performs the first <u>and</u> the second contraction.

```
INSERT(i,p)  (i an interval, p a node) =

   if interval(p) $\subseteq$ i
   then cover(p) $\leftarrow$ cover(p) + 1
        maxcover(p) $\leftarrow$ maxcover(p) + 1
   else if interval ($\lambda$p) $\cap$ i $\neq \emptyset$
        then LEFTDOWN
             INSERT(i,$\lambda$p)
             UP
        fi
        if interval($\rho$p) $\cap$ i $\neq \emptyset$
        then RIGHTDOWN
             INSERT(i,$\rho$p)
             UP
        fi
        maxcover(p) $\leftarrow$ cover(p) + max {maxcover($\lambda$p),maxcover($\rho$p)
   fi.
```

To obtain the algorithm DELETE(i,p) replace in INSERT '+1' by '-1' (in lines 2 and 3) and 'INSERT' by 'DELETE'.

Since LEFTDOWN and RIGHTDOWN are unchanged we only describe the operation UP. It consists of three steps:

1. Move the current position to the father.

2. Perform the first contraction, that is link in/out p's sons according to the following rule:

```
T(p') = if maxcover(λp) = maxcover(ρp)
        then (p',T(λp'),T(ρp'))
        else if maxcover(λp) > maxcover(ρp)
             then (p',T(λp'),black leaf)
             else (p',black leaf,T(ρp'))
             fi
        fi
```

(see Fig. 3-23).

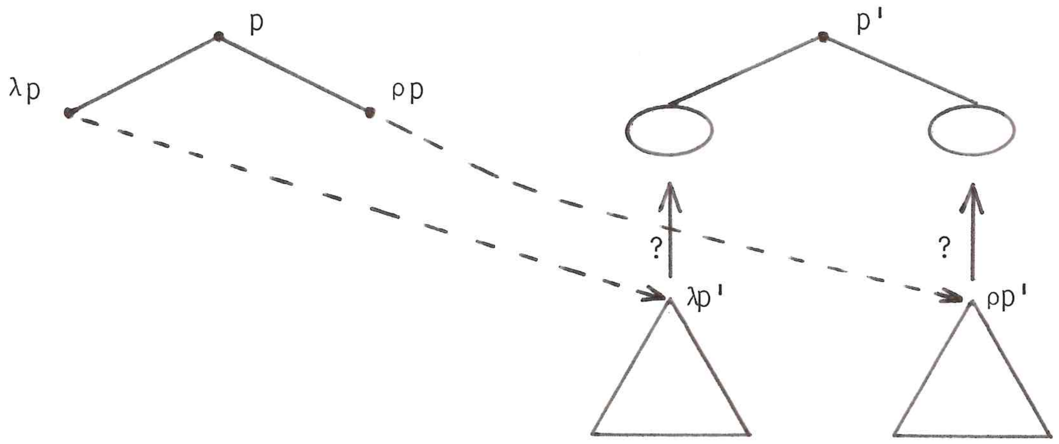


Figure 3-23

3. Perform the second contraction by merging two white leaves into one white leaf, a white and a black leaf into an ordinary leaf etc. as described in Section 3.1.

In this way after an insertion or deletion the following holds:

For each node p of an ICST, T(p') represents the set of maximally covered disjoint subintervals of interval(p), MI(p). If MI(p) has cardinality ß then T(p') has O(ß) nodes.

For completeness we also give the algorithm QUERY of an ICST which implements the function maxintervals of a weighted partition.

```
QUERY(i,p)  (i an interval, p a node) =

  if interval(p) ⊆ i
  then REPORT
  else if interval(λp) ∩ i ≠ ∅
       then LEFTDOWN
            QUERY(i,λp)
            SIMPLE-UP
       fi
       if interval(ρp) ∩ i ≠ ∅
       then RIGHTDOWN
            QUERY(i,ρp)
            SIMPLE-UP
       fi
  fi.
```

REPORT traverses a subtree reporting intervals in interval lists as described in Section 3.1.4. Now the reported intervals are the maximally covered ones. SIMPLE-UP is operation UP without step 2, which is superfluous since the coverage of subtrees has not changed.

With these modifications the basic properties of a CST hold again:

<u>Lemma 3.10</u>: In an inverted contracted segment tree based on n points an interval can be inserted or deleted in $O(\log n)$ time. The ß disjoint maximally covered subintervals of an interval can be reported in $O(\log n + ß)$ time. Space is $O(n)$ and the empty ICST can be constructed in $O(n)$ time.

For proof, see the corresponding Lemmas and Theorems of Section 3.1.

We now return to the abstract level to analyze algorithm H-CONTOUR. Implementing the weighted partition by an ICST we obtain

Theorem 3.11: For a set of n rectangles of height h, the h-contour can be computed in $O(n \log n + p_h)$ time and $O(n + p_h)$ space, where p_h is the size of the h-contour.

Proof: We show this by analyzing the time and space requirements of algorithm H-CONTOUR. As we know from Theorem 3.8 step 1 takes $O(n \log n)$ time. Step 2 involves sorting ($O(n \log n)$ time) and building an empty ICST ($O(n)$time). Step 3 is a loop executed 2n times for all vertical rectangle edges. For every edge an insertion or deletion is performed, taking $O(\log n)$ time and possibly a call of function maxintervals.This call, implemented by algorithm QUERY of the ICST, takes $O(\log n)$ time plus $O(p_j)$ time for reporting p_j contour-pieces. Hence the whole scan takes $O(n \log n)$ time plus the time for reporting the h-contour which is $O(p_h)$.

Space is $O(n)$ for the ICST and $O(p_h)$ to store the contour-pieces which are later by a sorting operation linked into contour-cycles in $O(p_h)$ time and space. Summing all the time and space requirements yields the bounds of the theorem. <>

3.2.5. *Computing all measures*.

So far we are able to compute i-measure and i-contour for the extremal values of the i-scale, that is, $i = 1$ and $i = \text{height}(R)$. It is easy to show that all the algorithms developed up to now are time-optimal (see Section 3.2.9). We now address the range of i-values between the extremes which is in general more difficult. In the next two sections we give algorithms which compute all measures or all contours, respectively, in a single line-sweep. The all measures problem can be solved by a simple extension of our previous methods, but the resulting algorithm is probably not optimal. Surprisingly the all contours problem permits a very simple optimal solution, reversing the normal situation where the contour problem is much more difficult than the measure problem. In Sections 3.2.7 and 3.2.8 we finally address the problem of computing the i-measure and the i-contour for any specific value of i which proves to be the hardest problem.

On the abstract level, the all measures problem can be solved by a trivial modification of algorithm H-MEASURE and the weighted partition data structure. We introduce a new function length(i), giving the total length of the sweepline parts which are covered i times, for $i = 1,\dots,\text{height}(R)$:

$$W.\text{length}(i) = \sum_{\{(i_y,c) \in W \mid c = i\}} |i_y|$$

The algorithm is then quite similar to algorithm H-MEASURE:

```
Algorithm ALL MEASURES

Input:    A set of rectangles R.
Output:   For  i = 1,...,height(R), i-measure(R), a real number.

Step 1:   h ← HEIGHT(R)

Step 2:   Let VX be the set of all vertical rectangle edges, sorted by x. Let
          P = y-set(R).
          W ← $\{(i_y,0) \mid i_y \in \text{partition}(P)\}$
          For i = 1,...,h:  area[i] ← 0

Step 3:   Scan VX.

          if encountering a left edge (x, $y_1$, $y_2$)
          then W.insert([$y_1$,$y_2$])
          else *a right edge (x, $y_1$, $y_2$) is encountered*
               W.delete([$y_1$,$y_2$])
          fi
          x' ← the next sweepline position. If there is none, goto step 4.
          for all  i ∈ {1,...,W.maxc}:   area[i] ← area[i] + (x'-x)· W.length(i)

Step 4:   For  i = 1,...,h:  output(area[i])

end of algorithm ALL MEASURES.
```

The obvious method to implement the weighted partition W with function length(i) is to use a counting segment tree where each node contains an array length. For a node p length[i](p), for i = 0,...,h, contains the total length of the subintervals of interval(p) which are covered i times. When a node is covered, a shift to the right is performed on its array length, also a shift to the left, when it is uncovered. The values of length(p) can be reconstructed from the values of length(λp) and length(ρp):

$$\text{length}[i](p) = \begin{cases} 0 & 0 \le i < \text{cover}(p) \\ \text{length}[i-\text{cover}(p)](\lambda p) & \text{cover}(p) \le i \le h \\ \quad + \text{length}[i-\text{cover}(p)](\rho p) & \end{cases}$$

This makes it possible to update the length-arrays of the nodes on the search path after an insertion or deletion. Of course now this isn't any more possible in constant time, but takes O(h) time per node. After updating, W.length(i) is given by length[i](root).

The time and space requirements of this algorithm are not difficult to analyze. Steps 1 and 2 take together O(n log n) time. In step 3, any insertion or deletion takes O(h log n) time and the updating of the array area takes O(h) time. Hence step 3 dominates the time complexity which is in total $O(n \cdot h \log n)$. Space is $O(h \cdot n)$ since each node stores an array of length h.

This approach was straightforward and easy to analyze. However by a more careful implementation we can gain some efficiency. We replace the array length belonging to each node by a linked list lengthlist. Each list element is a pair (i, l_i) (equivalent to $\text{length}[i] = l_i$). However the list contains only nonempty elements, that is, $l_i \neq 0$ for any list element (i, l_i).

Covering a node is now performed by scanning its list and increasing all i-fields by one, uncovering by decreasing them, in $O(\beta)$ time for a list of length β. It is again possible to construct lengthlist(p) from cover(p), lengthlist(λp) and lengthlist(ρp). This takes $O(l + r)$ time where l and r denote the size of lengthlist(λp) and lengthlist(ρp), respectively. To analyze time and space requirements it is crucial to find some bound on the sizes of lists which appear in the tree.

There are two facts limiting the list size. First, no list can have more than (h+1) items, for h = height(R). Second, if p is the root of a subtree with m leaves, then there cannot exist more than m different coverage values in this subtree and lengthlist(p) has at most m items. Clearly the size of lengthlist(p) is the minimum of h and m, in the worst case.

The first restriction will usually apply to nodes close to the root of the tree while the second applies to leaves and nodes on the lower levels. h has some value between 1 and n. Let us assume $h = 2^k$. (This simplification does not change the asymptotic complexities which we compute in the sequel). Then the list sizes on the different levels of the tree are in the worst case as follows:

	height	list size	number of nodes on the level
(log n - k) levels	log n	2^k	1
	(log n) - 1	2^k	2
	.	.	.
	k + 1	2^k	$2^{(\log n)-k} = n/2^k$
k levels	k	2^{k-1}	$n/2^{k-1}$
	.	.	.
	3	$2^2 = 4$	n/4
	2	$2^1 = 2$	n/2
	1	$2^0 = 1$	n

Figure 3-24

An insertion or deletion is restricted to a forked search path and the work done for each node is proportional to the size of this node's lengthlist. Hence we obtain the asymptotic update time by summing the list sizes on a search path. Let

$U(n,h)$ denote the worst-case update time.

$$\begin{aligned} U(n,h) &= O((\log n - k)\cdot 2^k + 2^k - 1) \\ &= O(h\cdot(\log (n/h) + 1)) \qquad (h = 2^k,\ k = \log h) \end{aligned}$$

Since the update time dominates the time complexity of the algorithm we have a total time complexity of $O(n\cdot h\ (\log (n/h) + 1))$. For $h \to n$ or $h \to 1$ this is $O(n^2)$ or $O(n \log n)$, respectively.

To compute the space-requirements we look again at the list sizes on the different levels of the tree (Fig. 3-24) and sum the products of list sizes and numbers of nodes on the level. Let $S(n,h)$ denote the worst-case space-requirements.

$$\begin{aligned} S(n,h) &= O(k\cdot n + 2^k\cdot(2^{\log n - k + 1}-1)) \\ &= O(n\cdot\log h + 2\cdot 2^k\cdot 2^{\log (n/h)}) \\ &= O(n \log h + h\cdot(n/h)) \\ &= O(n\ (1 + \log h)) \end{aligned}$$

Again we consider the cases $h \to n$ and $h \to 1$:

$$\begin{aligned} h \to n: &\quad S(n,h) \to O(n \log n) \\ h \to 1: &\quad S(n,h) \to O(n) \end{aligned}$$

We summarize these results in

<u>Theorem 3.12</u>: Let R be a set of n rectangles of height h. Then we can compute all i-measures of R together, for $i = 1,\dots,h$, in $O(n\cdot h\ (\log (n/h) + 1))$ time and $O(n\cdot(1 + \log h))$ space. Since $1 \le h \le n$, the time complexity lies between $O(n \log n)$ and $O(n^2)$ and the space requirements between $O(n)$ and $O(n \log n)$.

3.2.6. *Computing all contours.*

Surprisingly the all contours problem has the most simple solution of all problems we consider in Section 3. This is due to the fact that any fragment of a rectangle edge created by intersection with other rectangles is part of some i-contour. Since we have to report all those contour-pieces we are allowed to perform a constant amount of work for each occurring line segment intersection. In other words, because so much work has to be done anyway, there is no possibility to save some of it by sophisticated methods and a simple method will do to provide a time-optimal solution!

Recall that between any two sweepline positions the intersection of the sweepline with the set of rectangles is a set of intervals. This set of intervals induces a partition of the line into a set of fragment intervals, each of which has its own elevation level (see Fig. 3-17). For the first time we can now afford to store this collection of fragments basically as a simple linked list ordered by y-coordinate. Each list element is a pair (i_y,c) like the elements of a weighted partition. There is a difference to a weighted partition, however, because this abstract data structure is based on a set of points P which is static. That is, during the whole line-sweep the fragment intervals of a weighted partition do not change. This reflects the fixed structure of the segment tree which is normally used for implementation.

In contrast to this the linked list we use now represents the dynamic set of fragments induced by the intervals currently intersecting the sweepline. We might define a 'dynamic weighted partition' as an abstract representation, but in the present case this seems not necessary.

The new line-sweep algorithm proceeds as follows: Encountering the left edge of a rectangle (x,y_1,y_2) the list element (i_y,c) is located which contains point y_1 $(y_1 \in i_y)$. Obviously this list element has to be split into two parts with intervals $[\text{bottom}(i_y), y_1]$ and $[y_1, \text{top}(i_y)]$. Starting with the latter list element, the linked list is scanned. For each encountered list element (i'_y, c') the coverage value c' is increased by 1 and the line segment (x,i'_y) is reported as a piece of the c'-contour. The scan ends when the list element (i''_y,c'') with $y_2 \in i''_y$ is found. This list element has again to be split in the same way as the first list element. There exist some special cases (y_1 and y_2 may fall into the same fragment interval or on the boundary between two fragments) but there is no difficulty in treating these cases. - For a right rectangle edge (x,y_1,y_2), the same scan is performed decreasing the coverage values and merging the list elements adjacent to y_1 and y_2, if necessary. Again contour-pieces are reported during this scan.

The only remaining problem is how to locate the first list element efficiently. For this we superimpose a binary searchtree of which the list elements are the leaves. Any balanced tree scheme (AVL-trees, 2-3 trees etc.) with additional concatenation of the leaves can be used to implement this structure. For an explicit description of a leafsearch tree see for instance Comer [C] (B^+-tree). This structure allows to search for a leaf as well as splitting or merging leaves together with restructuring in $O(\log n)$ time (n the number of leaves).

As for the previous contour algorithms we only describe how to report the contour-pieces without linking them into contour-cycles. This can again be done in $O(p)$ time and space by the method given in [LiP], where p is the number of contour-pieces.

The time complexity of this algorithm is as follows: The preparatory sorting of the rectangle edges takes $O(n \log n)$ time. For each rectangle edge encountered during the line-sweep $O(\log n)$ work is done to locate the 'start' list element.

Scanning the linked list takes O(ß) time if ß contour-pieces are reported. Splitting or merging leaves (list elements) with restructuring the tree can also be done in O(log n) time. Hence the whole line-sweep takes O(n log n) time without the time for reporting, which is linear in the total size of all contours.

The leafsearch tree requires O(n) space. Since the contour-pieces have to be stored to link them into contour-cycles later, an additional O(p) space is used. Thus we have:

Theorem 3.13: Given a set of n rectangles R, all i-contours of R can be computed together in O(n log n + p) time and O(p) space, where p is the total size of all contours.

3.2.7. *Computing the i-measure.*

We must admit that we did not really have a new idea to solve this problem. Therefore this section will be quite short. Essentially we use the all measures algorithm to compute the i-measure. However we can save some work. Recall that all measures are computed by maintaining a segment tree during the line-sweep which stores the y-intervals of all rectangles currently intersecting the sweepline. Each node p of this tree has an associated interval interval(p) and contains some information about the total length of the subintervals of interval(p) which are covered i times, for $i = 0,\dots,h$. Let us for a moment return to the representation of this information by an array. From the definition

$$\text{length}[i](p) = \begin{cases} 0 & 0 \leq i \leq \text{cover}(p) \\ \text{length}[i-\text{cover}(p)](\lambda p) & \text{cover}(p) \leq i \leq h \\ \quad + \text{length}[i-\text{cover}(p)](\rho p) & \end{cases}$$

it follows that length[i](p) depends only on fields length[j](q) of nodes q in p's subtree with $j \leq i$. This implies that for the computation of the i-measure all arrays length in the tree need only have the fields 0 to i.

The same holds for the representation using list lengthlist; the analysis of time and space complexity remains the same if we replace h by i. Therefore the result of Section 3.2.5 applies:

Theorem 3.14: For a set of n rectangles R the i-measure can be computed in $O(n \cdot i\ (\log (n/i) + 1))$ time and $O(n\ (1 + \log i))$ space.

3.2.8. *Computing the i-contour.*

To solve the i-measure problem it was possible to use the algorithm for the all measures problem and to save some of the work done by it. Can we do the same for the i-contour problem? This seems not possible. The time for scanning the linked list can not be reduced, though we are interested only in the list elements with one specific coverage value. To solve the problem more efficiently we have to come up with a completely new algorithm.

We still intend to use a line-sweep algorithm which is even easy to describe on the abstract level:

```
Maintain a data structure representing the parts of the sweepline covered
(i - 1) times. Call it D.

   - Encountering a left rectangle edge  $v = (x, y_1, y_2)$:
       - Query D with interval  $[y_1, y_2]$  for subintervals covered (i-1) times.
         The answer (together with x) defines the contour-pieces created by v.
       - Insert  $[y_1, y_2]$  into D.
```

- Encountering a right edge $v = (x, y_1, y_2)$:
  - Delete $[y_1, y_2]$ from D.
  - Query D with $[y_1, y_2]$ for subintervals covered (i-1) times. Again the answer determines the contour-pieces created by v.

The difficulty is again to design a data structure supporting insertion, deletion and querying for subintervals covered (i-1) times. The first candidate for such a data structure is again the contracted segment tree, and we shall indeed use some variant of it called an <u>i-fold contracted segment tree</u>. Recall the structure of a normal CST: It consists of primary and secondary structures. The primary structure is a counting segment tree. The secondary structure is a collection of trees (contracted copies of subtrees of the primary tree) of which one is of special importance, namely the tree attached to $ROOT_2$. This tree, called the free tree, contains in its interval lists all the free intervals on the sweepline. The other trees are disconnected from $ROOT_2$ and attached to different nodes in the primary tree. We may say they are 'scattered' over the primary tree. The intervals represented in the <u>scattered trees</u> are covered one or more times.

Now any node p of the primary tree may have its copy either in the free tree or in one of the scattered trees. If p' belongs to the free tree, then the subtree of p' contains all free subintervals of interval(p) in its leaves and interval lists. Otherwise p has no free subintervals. Hence we may say that for each node p in the primary tree the free subintervals (that is, the subintervals covered 0 times) are <u>maintained</u> in the secondary structure.

In an i-fold contracted segment tree for each node p the secondary structure maintains the i sets of subintervals of interval(p) which are covered j times, for j = 0,...,(i-1). This is in analogy to the i-measure algorithm where for each node p all j-measures, for j = 0,...,i, are maintained by means of the list lengthlist(p). In a similar way it is now necessary to maintain the sets of j-covered subintervals for all j-values below i though we are only interested in the (i-1)-covered subintervals.

An i-fold contracted segment tree has the same primary structure as a normal CST, namely a counting segment tree. However, the secondary structure consists of i different trees T_0, T_1, ... , $T_{(i-1)}$ with roots $ROOT_0$, ... , $ROOT_{(i-1)}$ which take the role of the free tree. T_j contains exactly the j-covered intervals in its interval lists. Furthermore the secondary structure of an i-fold CST contains also a collection of scattered trees disconnected from any $ROOT_j$ (they are attached to primary nodes, as in a CST). The intervals represented in those trees are covered i or more times.

How is an i-fold CST maintained during the line-sweep? We have to describe how to insert and delete an interval and how to report for an interval the subintervals covered (i-1) times. The formal description of these algorithms is nearly the same as for an inverted contracted segment tree (see Section 3.2.4).

The algorithms for insertion and deletion read now:

```
INSERT(i,p) =                             DELETE(i,p) =
                                          (only differences to INSERT are shown)
if interval(p) ⊆ i
then cover(p) ← cover(p) + 1              cover(p) ← cover(p) - 1
    SHIFT-UP                              SHIFT-DOWN
else if interval(λp) ∩ i ≠ ∅
    then LEFTDOWN
        INSERT (i, λp)                    DELETE (i, λp)
        UP
    fi
    if interval(ρp) ∩ i ≠ ∅
    then RIGHTDOWN
        INSERT (i, ρp)                    DELETE (i, ρp)
        UP
    fi
fi.
```

Algorithm QUERY(i,p) is only minimally changed:

```
if interval(p) ⊆ i
then call REPORT for the current position in tree $T_{(i-1)}$
else ...
```

The remainder of the algorithm is unchanged.

The operations LEFTDOWN, RIGHTDOWN and UP now work in parallel on all trees $T_0, \dots, T_{(i-1)}$. In each tree exists a current position which is manipulated by these operations. For each single tree the operations work in the same way as for normal CSTs. That is, LEFTDOWN and RIGHTDOWN perform the going-down in secondary structures, expanding the tree, while UP moves the position upwards, recontracting the tree. Only the meaning of white and black leaves is changed. Recall that in a CST white leaves represent free subtrees of the primary tree, black leaves covered subtrees. Now in tree T_j, $j = 0,\dots,(i-1)$, a white leaf represents a subtree covered j times and a black leaf a subtree not containing any j-covered intervals.

Example: The empty tree has the structure

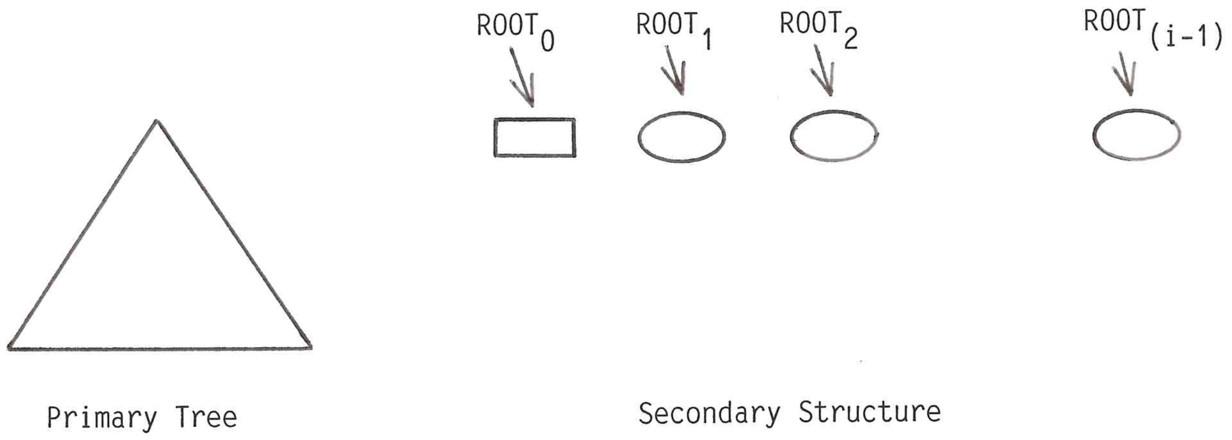


Figure 3-25

Contracting a tree (when going up after insertions or deletions) transforms white leaves into ordinary leaves or elements of interval lists. Hence after contraction the interval lists of tree T_j represent exactly the j-covered intervals in the i-fold CST.

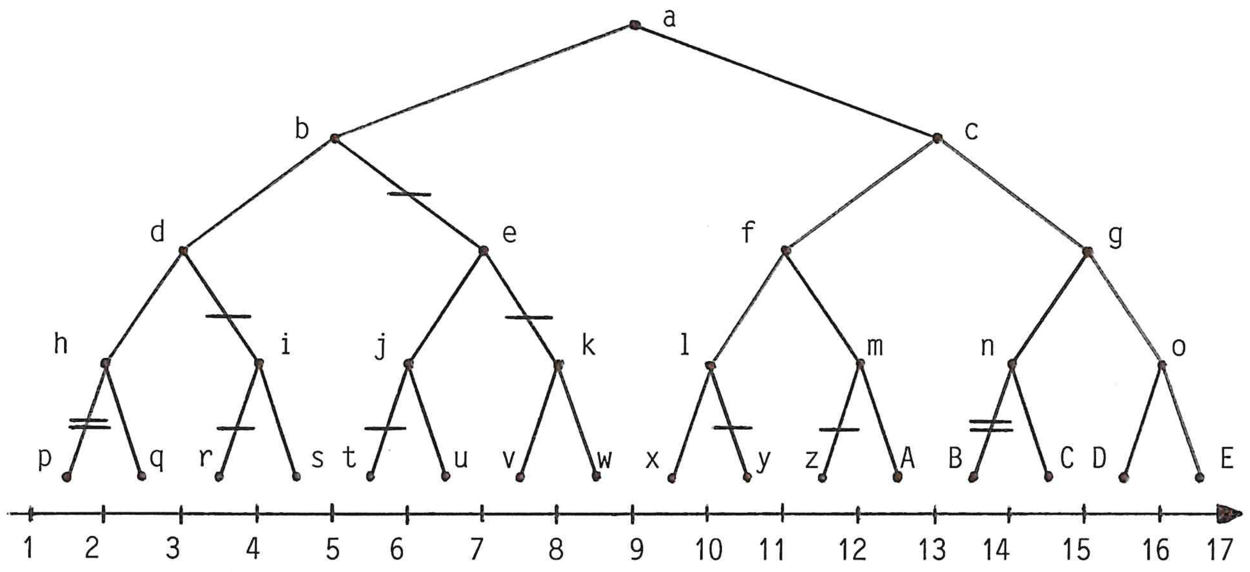
The algorithms INSERT and DELETE contain two new operations SHIFT-UP and SHIFT-DOWN. They update the secondary structure when a node's coverage is increased or decreased. Suppose node p of the primary tree becomes covered once more. Let us call p's subtree in the primary tree T(p). There exist contracted copies of T(p) in each tree $T_0,\dots,T_{(i-1)}$, which we call $T_0(p)$, $T_1(p)$, ... , $T_{(i-1)}(p)$. When SHIFT-UP is called, the actual positions in all trees are the roots of the trees $T_0(p)$, ... , $T_{(i-1)}(p)$. SHIFT-UP then moves all subtrees one position to the right. More precisely, $T_0(p)$ is replaced by a black leaf, $T_1(p)$ by $T_0(p)$, $T_2(p)$ by $T_1(p)$ etc. Finally $T_{(i-1)}(p)$ is attached to second(p).

Note that SHIFT-UP maintains correctly the condition that tree T_j represents the j-covered intervals: Before p received the additional covering $T_{(j-1)}(p)$ represented the (j-1)-covered subintervals of interval(p). Covering p, each of those subintervals becomes covered j times. Moving the subtree $T_{(j-1)}(p)$ into tree T_j makes sure that (the new) $T_j(p)$ represents the j-covered subintervals of interval(p). - Obviously $T_0(p)$ has to be replaced by a black leaf because the new $T_0(p)$ cannot contain any empty intervals after covering p.

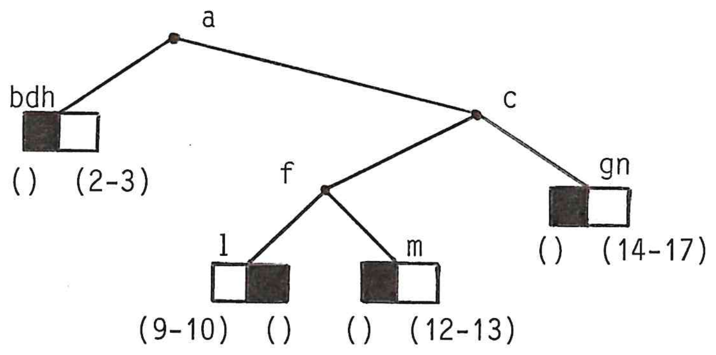
SHIFT-DOWN performs the inverse operation of moving subtrees left, into 'lower' trees, when uncovering a node.

We hope to clarify the issue by providing a rather detailed example. A 3-fold contracted segment tree is shown and the effect of inserting an interval covering just one node f.

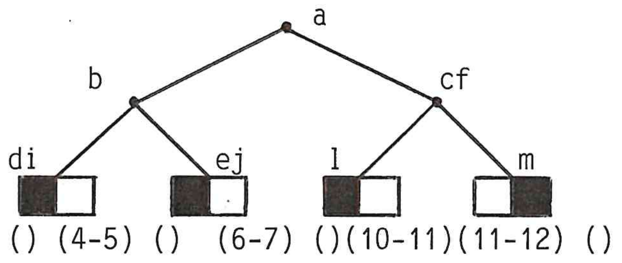
Initial state:



T_0:



T_1:



T_2:

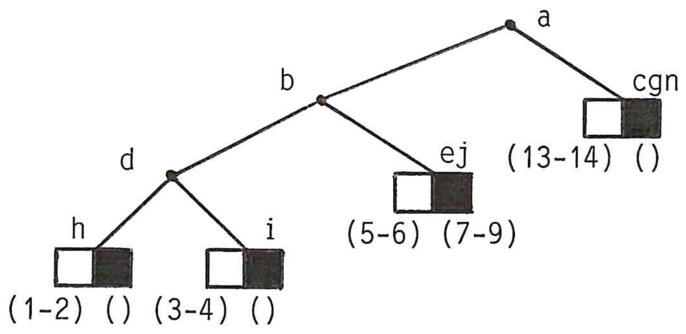


Figure 3-26

After going down, expanding (▽ indicates current position):

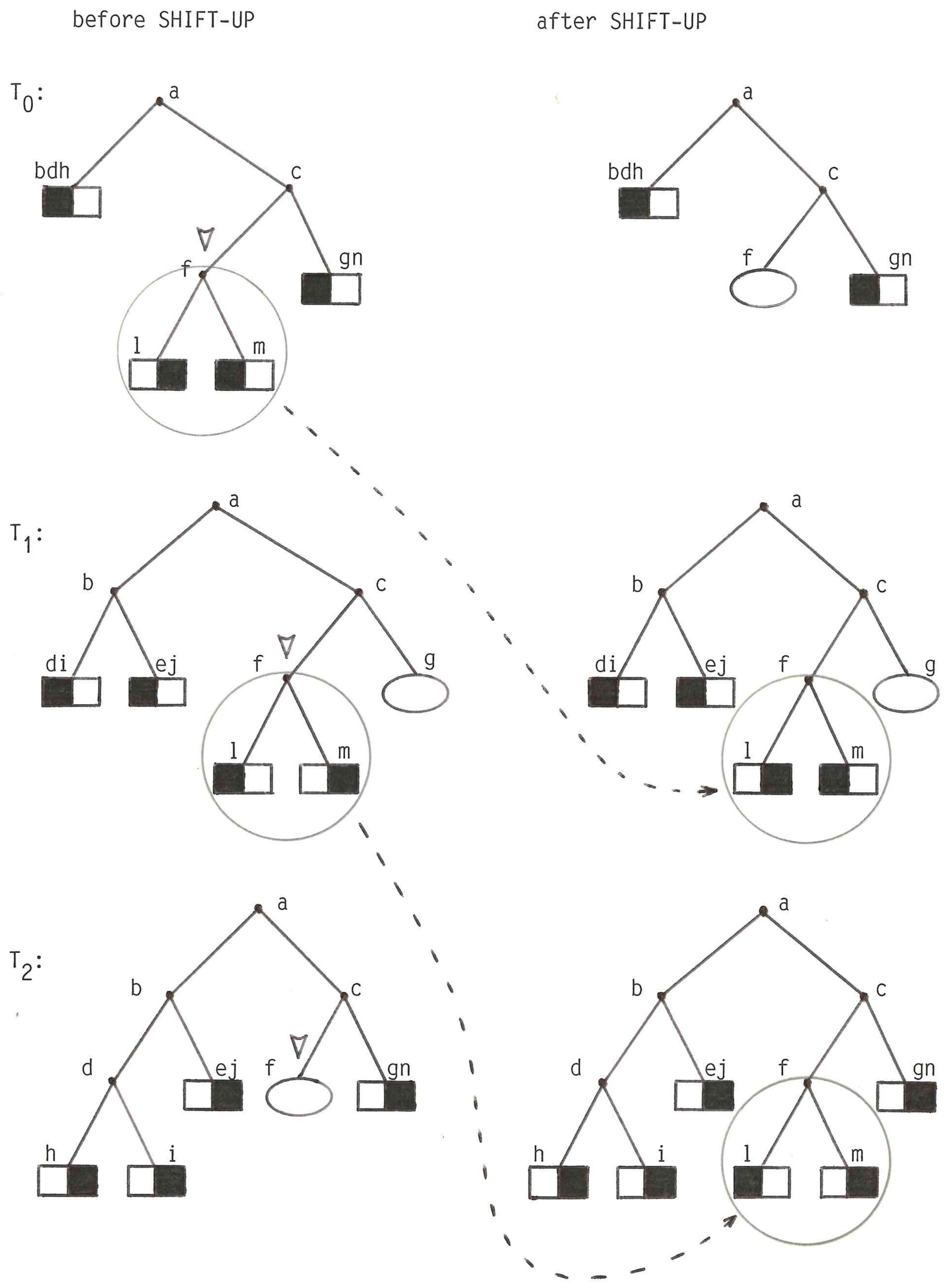


Figure 3-27

Final state after going up, recontracting:

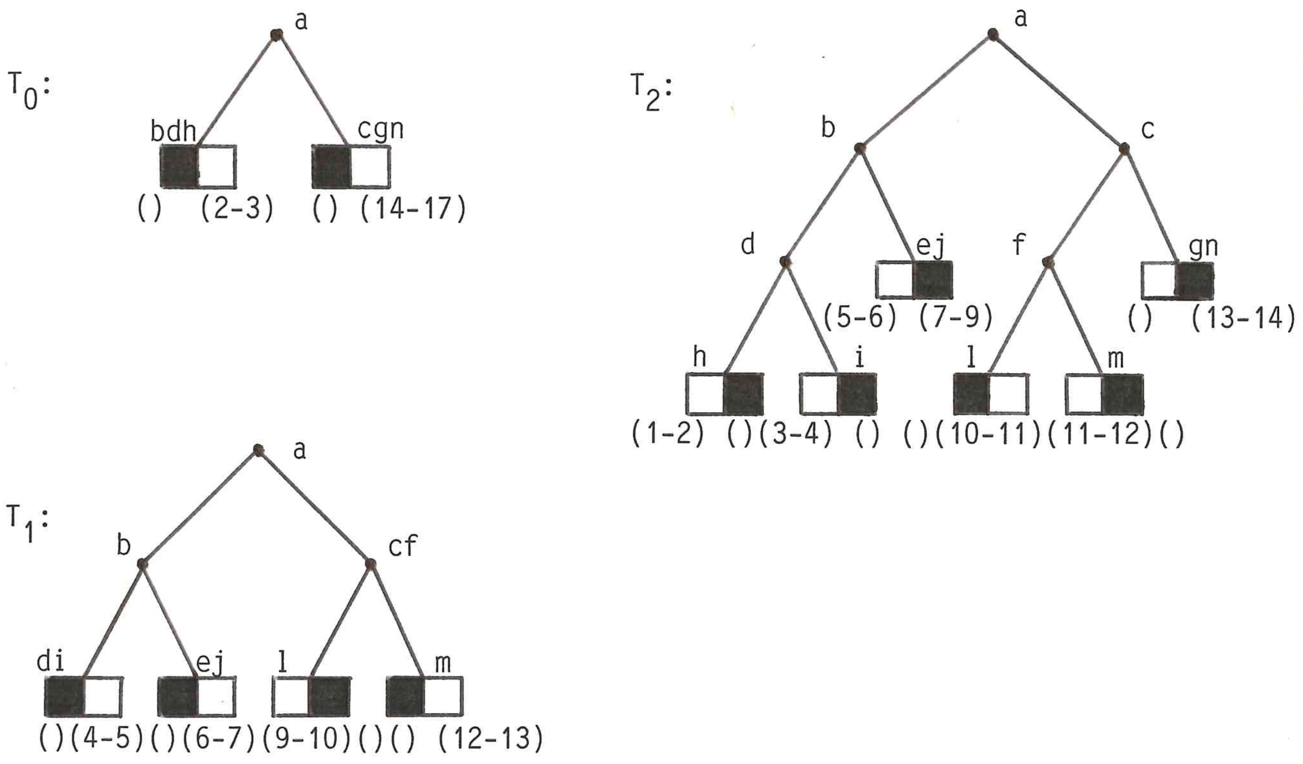


Figure 3-28

Reporting is done in the same way as for CSTs or ICSTs on a single secondary tree, namely $T_{(i-1)}$.

Let us now determine the time and space complexity of this algorithm. As before, the algorithms INSERT and DELETE produce O(log n) calls of any of the operations LEFTDOWN, RIGHTDOWN and UP. Those operations work in parallel on trees $T_0, \ldots, T_{(i-1)}$ and use constant time per tree. Since there are i trees, the total time for an insertion or deletion is O(i log n). Reporting takes O(log n + ß) time where ß is the number of reported pieces of the i-contour. Per vertical rectangle edge the algorithm requires therefore O(i log n + log n + ß) = O(i log n + ß) time. This yields a total time for computing the i-contour of

$O(n \cdot i \log n + p_i)$ where p_i is the size of the i-contour.

The i-fold contracted segment tree consists of primary tree and secondary structure, that is, a counting segment tree, the trees $T_0,\dots,T_{(i-1)}$ and scattered trees. The primary tree and the scattered trees require only O(n) space (each scattered tree is a copy of some part of the primary tree and the scattered trees are disjoint). For trees $T_0,\dots,T_{(i-1)}$ the analysis is a little more difficult. Observe that all trees together have at most as many leaves as there are fragment intervals (represented by leaves) in the primary tree. So the number of leaves of those trees is O(n). However, additional space is required for the name lists. How many names are stored in all trees? To analyze this, observe that the primary tree contains n paths from the root to a leaf and that each of those paths is stored in exactly one of the trees $T_0,\dots,T_{(i-1)}$.

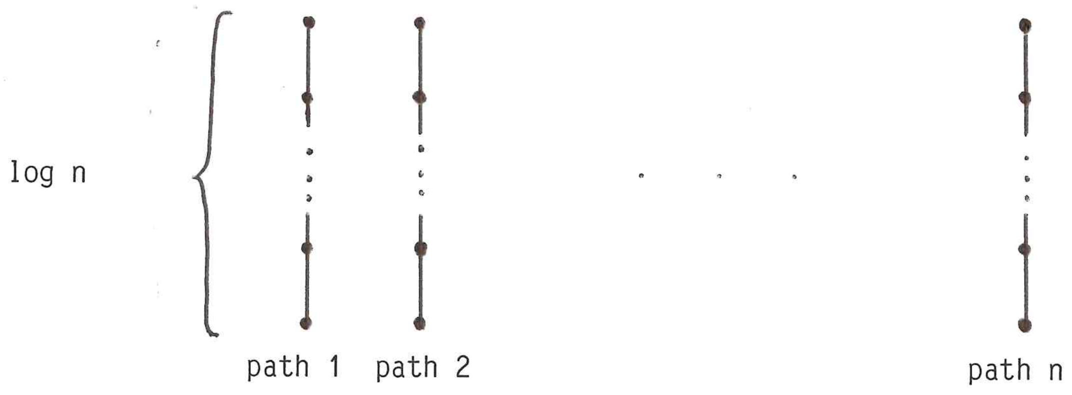


Figure 3-29

If all paths were stored separately, then O(n log n) space would be used. However, they are merged into trees such that for any two paths we consider, the top nodes may be stored only once.

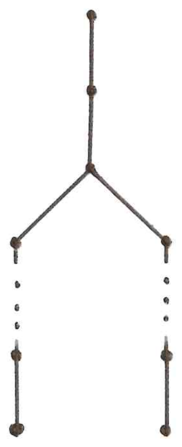


Figure 3-30

We do not know how the paths are distributed into trees. We look at two extreme situations to determine the worst case.

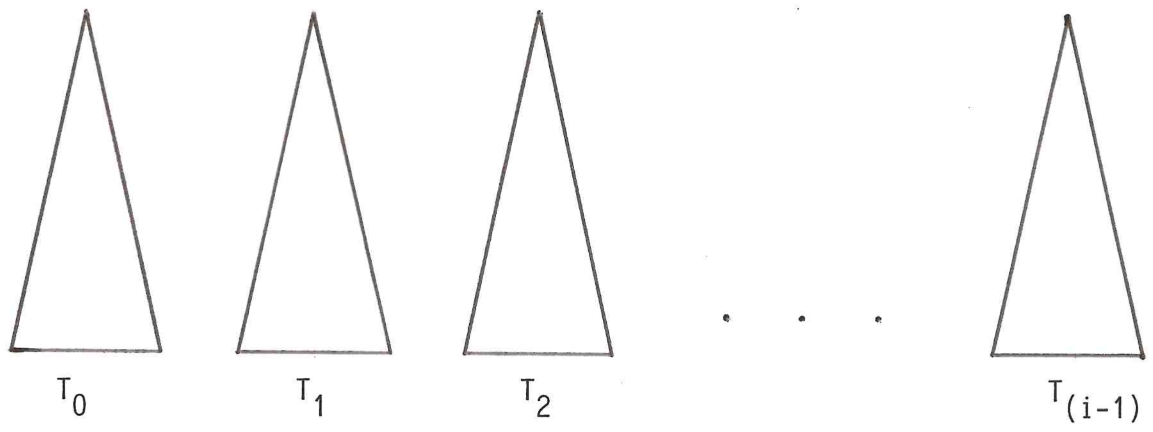


Figure 3-31

Fig. 3-31 represents the case that the paths are distributed equally over trees $T_0, \ldots, T_{i-1}$. Each tree has $O(n/i)$ leaves, therefore uses $O(n/i)$ space. The total space consumption is $i \cdot O(n/i) = O(n)$.

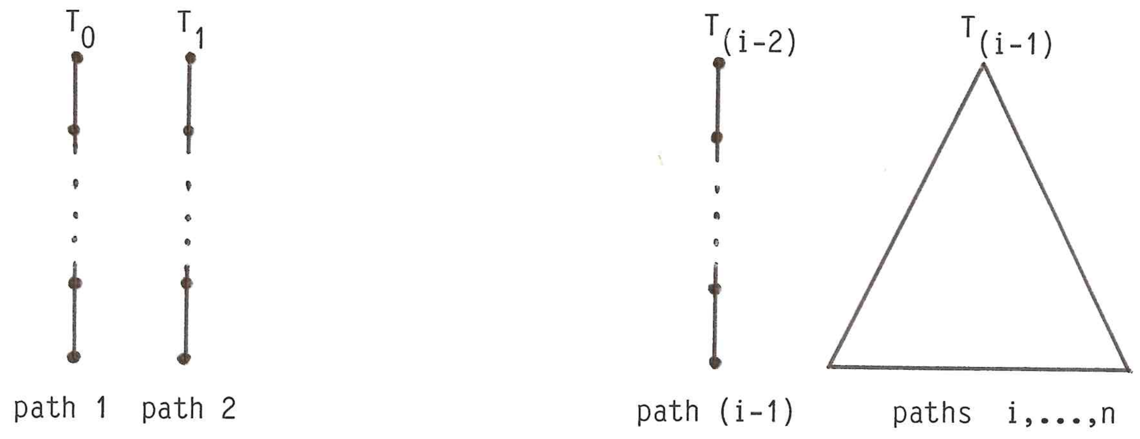


Figure 3-32

In Fig. 3-32 there are as many disjoint paths as possible, the other paths are merged into one tree. The total space required is $O((i-1)\cdot \log n) + O(n - i + 1) = O(n + (i-1)\cdot \log n)$. It is easy to see that this is indeed the worst case, because disjoint paths require more space than those merged into a tree.

Hence the space requirements of the i-fold contracted segment tree are $O(n + (i-1)\cdot \log n)$. As usual we need additional space of $O(p_i)$ to store the contour-pieces until they are linked into contour-cycles, and the same amount of time to construct the cycles. We summarize the results of this section in:

<u>Theorem 3.15</u>: For a set of n rectangles R the i-contour of R can be computed in $O(n\cdot i\cdot \log n + p_i)$ time and $O(n + (i-1)\cdot \log n + p_i)$ space, where p_i is the size of the i-contour.

Observe that for $i = 1$ the algorithm reduces to the algorithm of Section 3.1 and we obtain the same time and space complexity. Hence it is really a generalization of that algorithm.

3.2.9. *Summary of results, lower bounds.*

In the following table we summarize the performances of the line-sweep algorithms of Section 3.

Problem	Solution	
	Time complexity	Space requirements
Finding the contour of a set of iso-oriented rectangles R	$O(n \log n + p_1)$	$O(n + p_1)$
Determining height(R)	$O(n \log n)$	$O(n)$
Computing the height-measure	$O(n \log n)$	$O(n)$
Computing all measures	$O(n \cdot h\ (\log n/h + 1))$	$O(n(1 + \log h))$
Computing the i-measure	$O(n \cdot i\ (\log n/i + 1))$	$O(n(1 + \log i))$
Computing the height-contour	$O(n \log n + p_h)$	$O(n + p_h)$
Computing all contours	$O(n \log n + p)$	$O(n + p)$
Computing the i-contour	$O(n \cdot i \log n + p_i)$	$O(n + (i-1) \log n + p_i)$

Figure 3-33

In Fig. 3-33 n denotes the cardinality of R, h is the height of R, p_i the number of contour-pieces in the i-contour, for $i = 1, \dots, h$, and p the total number of contour-pieces in all contours $(p = \sum_{i=1}^{h} p_i)$.

Lower bounds.

At this point the obvious question is, whether the results presented in Fig. 3-33 are the best we can hope for or whether better algorithms might exist. Therefore we are interested in lower bounds on the complexity of the problems.

Time complexity.

From a naive point of view it seems quite clear that Ω (n log n) is a lower bound on the complexity of all problems listed in Fig. 3-33. One might argue that it is surely impossible to solve any of those problems without sorting the set of objects. To avoid sorting implies immediately not to use a line-sweep algorithm.

From a formal point of view we cannot be content with this kind of arguing. So we face the task of proving lower bounds. This is in general a quite difficult task for problems in computational geometry. Fortunately we are able to apply a result by Fredman and Weide [FredW]. They consider the ε-closeness problem:

> Given a set of n real numbers $x_1, \dots, x_n$, determine whether any two of them are within ε of each other (that is, $|x_i - x_j| < \varepsilon$).

Fredman and Weide show that under the usual decision tree model the complexity of this problem is Ω (n log n). In the sequel we will show for all of our problems that they are at least of the same complexity because any algorithm solving problem X (one of the problems in Fig. 3-33) can be used for solving the ε-closeness problem.

Theorem 3-16: Given a set of iso-oriented rectangles R, for any of the problems

(1) Finding the contour

(2) Determining the height

(3) Computing the height-measure

(4) Computing all measures

(5) Computing the i-measure

(6) Computing the height-contour

(7) Computing all contours

(8) Computing the i-contour

$\Omega(n \log n)$ is a lower bound on the number of comparisons required.

Proof:

(1) Finding the contour.

Lipski and Preparata [LiP] have shown that the problem of sorting n real numbers can be reduced in $O(n)$ time to computing the contour for some set of rectangles. Since sorting is of complexity $\Omega(n \log n)$, the same lower bound applies to the contour problem.

(2) Determining the height.

Suppose there exists an algorithm A computing the height in $T(n)$ time. We use it to solve the ε-closeness problem for $X = \{x_1, \dots, x_n\}$:

1. Construct a set of rectangles $R = \{r_1, \ldots, r_n\}$ where

   $$r_j = (x_j - \varepsilon, x_j, 0, 1) \text{ for } 1 \leq j \leq n.$$

   This can be done in $O(n)$ time.

2. Use A to compute height(R).
3. If height(R) = 1 then no two rectangles overlap and all points in X are more than ε apart. Otherwise (height(R) > 1) there exist two points closer together than ε.

In other words, A enables us to solve the ε-closeness problem in $T(n) + O(n)$ time. This implies $T(n) = \Omega(n \log n)$.

For the other problems we proceed in a similar way. We therefore just sketch steps 1. - 3. in the sequel.

(3) Computing the height-measure.

1. Construct R as in (2).
2. Apply A to compute height-measure(R).
3. If the result is exactly $n \cdot \varepsilon$ then no two points are within ε of each other, otherwise there exist two points closer together than ε.

(4) Computing all measures.

This includes computing the height-measure, hence we are done.

(5) Computing the i-measure.

1. Compute the minimal and maximal x-value occurring in X, $\underline{x}$ and $\overline{x}$, in $O(n)$ time. Construct R as in (2) and let R' be $R \cup \{r_1', \dots, r_{i-1}'\}$ where

   $$r_j' = (\underline{x} - \varepsilon, \overline{x}, 0, 1) \text{ for } j = 1, \dots, i-1.$$

   This takes $\Theta(i) = O(n)$ time.

2. Apply A to R'.
3. If the i-measure is exactly $n \cdot \varepsilon$ then the answer for the ε-closeness problem is negative (no two points ...) otherwise positive.

(6) Computing the height-contour.

1. Construct R as in (2).
2. Apply A to compute the height-contour of R.
3. Determine in $O(n)$ time whether the answer is a collection of n contour-cycles of which each represents a rectangle with an x-interval of length ε. If this is the case then the answer for the ε-closeness problem is negative, otherwise positive.

(7) Computing all contours.

The solution includes those of problems (1) and (6) and is therefore $\Omega(n \log n)$.

(8) Computing the i-contour.

We combine the techniques of (5) and (6), that is

1. Construct R' as in (5).
2. Apply A to compute the i-contour of R'.
3. Check the solution as in (6).

This completes the proof. <>

It follows immediately that our algorithms for computing the height and the height-measure are time-optimal. The algorithms for computing the contour (Section 3.1), the height-contour and all contours are also time-optimal because $\Omega(p_1)$, $\Omega(p_h)$ and $\Omega(p)$, respectively, are independently lower bounds for these problems.

Concerning the algorithms for computing all measures, the i-measure and the i-contour we do not know whether they are time-optimal or not. However, we conjecture that it is not possible to find a better line-sweep algorithm for any of them.

Space complexity.

There is not much to be said about space-complexity. Obviously $\Omega(n)$ space is necessary for the representation of the set of rectangles. Hence the algorithms for computing the height and the height-measure, respectively, are space-optimal.

The following remark holds for all our contour algorithms: Since in the line-sweep the vertical contour-pieces are stored to infer the horizontal contour-pieces afterwards, $O(p_1)$ (or p_h, p, p_i, respectively) space is required. We do not know whether it is possible to report the contour-pieces in the order of the contour-cycles by a different method, using only $O(n)$ space. Hence we don't know whether those algorithms are space optimal. If, however, the task is only to report all contour-pieces (in any order) then the respective algorithm may be used twice, in a left-to-right and a bottom-to-top line-sweep, to report all contour-pieces, not changing the asymptotic time bound. In that case the algorithms for computing the contour, the height-contour and all contours, respectively, require only $O(n)$ space and are space-optimal.

For the three remaining algorithms it is not clear whether they are space-optimal or not.

4. *DIVIDE-AND-CONQUER ALGORITHMS*.

In Section 3 we treated all the problems considered by means of line-sweep algorithms. In this section we study the applicability of a completely different algorithmic paradigm, divide-and-conquer, for problems in planar geometry. To achieve our goal of finding a divide-and-conquer solution for the contour problem we first look at some more basic problems. One of the most elementary problems conceivable is to find all intersections among a set of horizontal and vertical line segments. After solving it we study the point enclosure problem (given a mixed set of points and rectangles, which rectangles enclose which points?). The study of this problem is particularly interesting because its solution can be combined with a solution of the line segment intersection problem to solve the rectangle intersection problem (given a set of rectangles, report all intersecting pairs). In fact, Sections 4.1 - 4.3 contain a time- and space-optimal divide-and-conquer solution for this problem. Finally we describe a divide-and-conquer solution of the contour problem (Section 4.4). The high-level algorithm used for this is rather general and includes a solution of the measure problem (determine the size of the union of a set of rectangles).

To apply divide-and-conquer to a set S of planar objects we obviously have to find some partition of S (preferably into two parts of equal size). Of course we may choose an arbitrary division (for a set of n objects put 'the first' n/2 objects into set A, the other objects into set B). However, it soon becomes clear that algorithms based on arbitrary division cannot be very efficient since the division does not exploit the geometric properties of the problem.

A more appropriate idea seems to be the following: Choose an arbitrary line l, for instance a vertical one, splitting S into the sets A of objects left of l and B of objects right of l. This idea works nicely for a set of points in 2-space. However, if the objects have some x-extension then l usually intersects some objects introducing a third set C into the partition which contains the intersected objects. Two problems arise: First, it is hardly possible to find an equal-sized partition. Second, the objects of C interact with objects in A and in B and it is difficult to treat this interaction without loss of efficiency.

Apparently up to now these obstacles prevented the use of divide-and-conquer algorithms for planar objects other than points. Only recently Lee [Le] tried divide-and-conquer to solve the rectangle intersection problem using the mentioned 3-partition. The time complexity of his algorithm ($O(n \log n \log^* n + k)$ for n rectangles with k intersections) is quite good but a little less than optimal ($O(n \log n + k)$).

In the sequel we give time-optimal divide-and-conquer algorithms for all five problems we study. The efficiency of our algorithms is based on a new idea which is quite simple but important and which we call <u>separational representation</u> of orthogonal planar objects. Each object is represented by 'its left and right end'. For instance in case of rectangles this leaves us with a set of vertical line segments V. From now on each object in V is treated as an independent unit. The objects in V are characterized by a single x-coordinate, hence a dividing line splits V into only two subsets LEFT and RIGHT which can be 'conquered' independently. In the merge step the original objects (rectangles) are reconstructed though only on a conceptual level.

Looking back to line-sweep algorithms we realize that in fact the same principle is used there: objects are represented by their left and right ends. This seems appropriate since in a line-sweep algorithm the x-axis (for instance) is transformed into a time axis and the 'ends' of an object correspond to its 'appearing' and 'disappearing'. We will see, however, that separational representation proves also quite useful for divide-and-conquer algorithms. In fact, a divide-and-conquer algorithm may be perceived as a 'structured hierarchical line-sweep'.

A very interesting question arising from our results is how line-sweep and divide-and-conquer using separational representation compare with regard to the complexity of algorithms and data structures when applied to the same problem. Some first observations of this kind are discussed in the conclusions (Section 6).

The results of Section 4 have been published previously in [GüW] and [Gü3].

4.1. *Finding line segment intersections.*

The line segment intersection problem is:

> Given n horizontal and vertical line segments, report all pairwise intersections.

A line-sweep algorithm for this problem was found by Bentley and Ottmann [BeO]. In this section we first give an algorithm LINE SEGMENT INTERSECTION which makes use of a second algorithm LINSECT, then explain it and finally analyze its time and space requirements. LINSECT uses divide-and-conquer to accomplish its task.

For the sake of simplicity and clarity we initially assume all line segments to have distinct x- and y-coordinates. At the end of Section 4.1 we drop this restriction.

Algorithm LINE SEGMENT INTERSECTION

Input: A set of horizontal line segments H and a set of vertical line segments V.
Output: The set of all intersecting pairs (h,v) where $h \in H$ and $v \in V$.

Step 1: Let $\overline{H}$ be the set of endpoints defined by H. Let S be $\overline{H} \cup V$ sorted by x-coordinate. *Observe that sorting is possible because all objects in $\overline{H} \cup V$ are characterized by a single x-coordinate.*
Step 2: LINSECT(S)

end of algorithm LINE SEGMENT INTERSECTION.

Algorithm LINSECT (S, LEFT, RIGHT, VERT)

Input: An x-ordered set of objects S where each object is either a point (a left or a right endpoint of a horizontal line segment) or a vertical line segment.
Output: Three sets LEFT, RIGHT and VERT. LEFT and RIGHT contain the y-projections of left and right endpoints in S, respectively, whose partner is not in S. VERT is a set of intervals. It contains the y-projections of all vertical line segments in S.
Recursive invariant: On exit LINSECT has reported all edge intersections within S, that is all pairs (h,v) where h is a horizontal line segment in S (represented by its left or right endpoint) and v is a vertical line segment in S.

Case 1: S consists of one element p.

a) $p = (p_x,p_y)$ is the left endpoint of a horizontal line segment:

LEFT $\leftarrow \{p_y\}$, RIGHT $\leftarrow \emptyset$, VERT $\leftarrow \emptyset$ and return.

b) $p = (p_x,p_y)$ is the right endpoint of a horizontal line segment:

LEFT $\leftarrow \emptyset$, RIGHT $\leftarrow \{p_y\}$, VERT $\leftarrow \emptyset$ and return.

```
c) $p = (p_x, p_{y1}, p_{y2})$ is a vertical line segment:

      LEFT $\leftarrow \emptyset$, RIGHT $\leftarrow \emptyset$, VERT $\leftarrow \{[p_{y1}, p_{y2}]\}$ and return.


Case 2:   S contains more than one element.

  Divide:   Choose an x-coordinate $x_\mu$ dividing S into two subsets $S_1$ and $S_2$ of
            approximately equal size.

  Conquer:  LINSECT ($S_1$, $LEFT_1$, $RIGHT_1$, $VERT_1$);
            LINSECT ($S_2$, $LEFT_2$, $RIGHT_2$, $VERT_2$)

  Merge:    (Let LR = $LEFT_1 \cap RIGHT_2$)
            output (($LEFT_1$-LR) # $VERT_2$)                          (1)
            output (($RIGHT_2$-LR) # $VERT_1$)                         (2)
                 *the operator # is defined below*
            LEFT $\leftarrow$ ($LEFT_1$-LR) $\cup$ $LEFT_2$                          (3)
            RIGHT $\leftarrow$ $RIGHT_1$ $\cup$ ($RIGHT_2$-LR)                        (4)
            VERT $\leftarrow$ $VERT_1$ $\cup$ $VERT_2$          and return.             (5)

end of algorithm LINSECT.
```

The merge step needs some explanation. First we have to define the symbol '#'. Let P be a set of points on the line and I a set of intervals on the line. Then

$$P \# I := \{(p,i) \mid p \in P,\ i \in I \text{ and } i \text{ contains } p\}.$$

In the merge step intersections between horizontal and vertical line segments are reported. Since horizontal line segments are represented by their endpoints we have to examine the interaction between points and vertical line segments.

Any set S_i to which LINSECT is applied has an associated rectangular area $A(S_i)$ which is defined by the minimal and maximal x- and y-coordinates of objects in

S_i. For a left or right endpoint $p \in S_i$ let l_p be the horizontal line segment from which it is taken. Then the <u>partial segment</u> PS(p) is defined by

$$PS(p) := l_p \cap A(S_i)$$

<u>Example</u>:

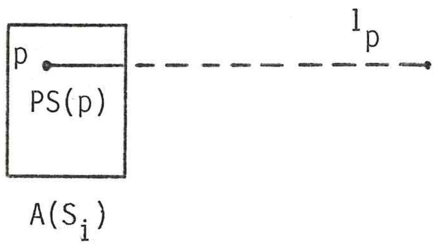


Figure 4-1

We know from the recursive invariant that after execution of LINSECT (S_i, ...) all intersections between partial segments and vertical line segments in S_i have been reported. Hence in the merge step of LINSECT (S, ...) we only have to find intersections between partial segments in S_1 and vertical line segments in S_2 and vice versa. In the merge step of LINSECT (S, ...) the following things can happen to a partial segment PS(p) (let us assume p is a left endpoint; the other case is symmetric):

a) p is in S_2.

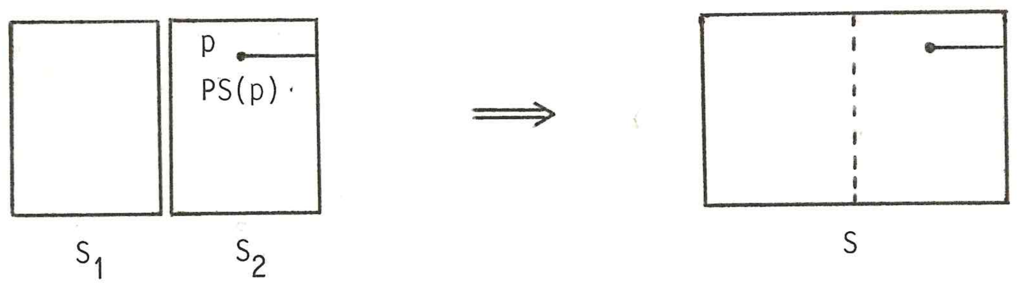


Figure 4-2

It remains as it is; no new intersections occur.

b) p is in S_1 and its partner is in S_2.

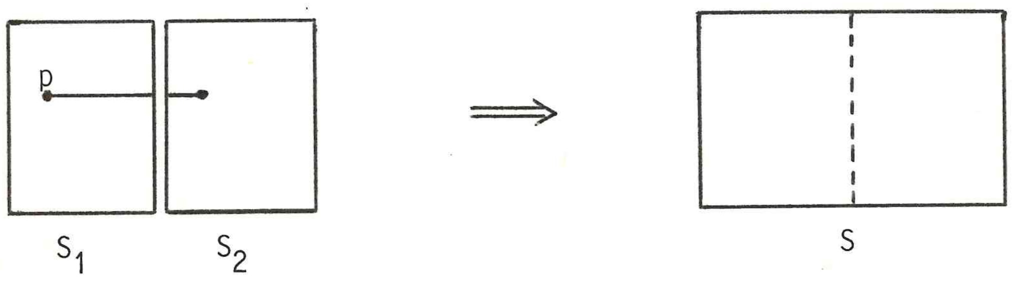


Figure 4-3

It is completed meeting its partner partial segment; no new intersections occur; both partial segments (or the complete segment) are removed from S.

c) p is in S_1 and its partner is not in S_2.

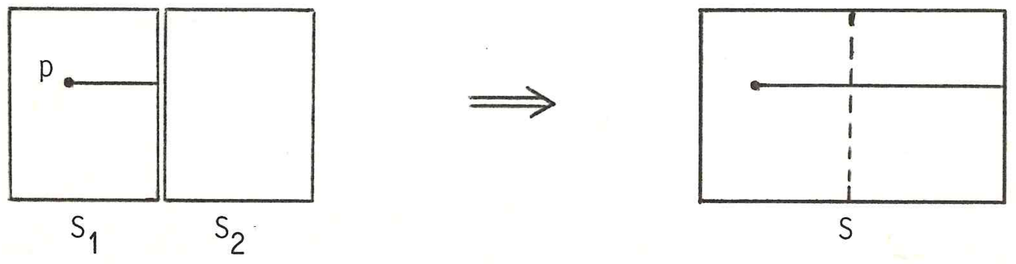


Figure 4-4

It is extended across $A(S_2)$. The crucial point is that all vertical line segments in S_2 which contain p's y-coordinate intersect l_p regardless of their x-coordinate. In other words l_p intersects a vertical line segment $v_j \in S_2$ if and only if p's y-projection (a point) is contained in v_j's y-projection (an interval).

Hence, since case c) describes the only way intersections can occur, we have reduced the two-dimensional line segment intersection problem to the one-dimensional point enclosure problem. The task of finding all intersections between extended partial segments from S_1 and vertical line segments from S_2 is exactly to compute $(LEFT_1\text{-}LR) \# VERT_2$. Therefore the two output statements (1) and (2) report exactly the intersections occurring by combining S_1 and S_2. Hence the recursive invariant holds once more. Lines (3) - (5) construct the sets LEFT, RIGHT and VERT in the obvious manner; subtraction of LR yields the removal of complete horizontal segments from S.

Time analysis.

LINE SEGMENT INTERSECTION:

Step 1: Constructing $\overline{H}$ takes linear time, sorting O(n log n) time, hence step 1 takes O(n log n) time.

Step 2: Let T(n) be the time complexity of applying LINSECT to a set S of cardinality n.

LINSECT:

Case 1: n = 1

All actions take constant time, establishing

$$T(1) = O(1) \qquad (1)$$

Case 2: $n > 1$

Divide: The x-ordered set S can be given as an array-subrange. Then dividing takes only constant time.

Conquer: The recursive calls yield two terms $T(n/2)$.

Merge: Finally we have to choose some representation of the sets LEFT, RIGHT and VERT. Surprisingly, simple y-ordered linked lists are sufficient to obtain an optimal solution. For VERT, each list item contains one interval $[y_b, y_t]$ and the list is ordered by the y_b-coordinates. Now all operations in the merge step can be realized by scanning some of the given lists in parallel. For instance, a first scan may operate on the lists (representing) $LEFT_1$, $RIGHT_1$, $LEFT_2$ and $RIGHT_2$ and construct lists LEFT and RIGHT. At the same time the elements of the set LR are removed from lists $LEFT_1$ and $RIGHT_2$. (This can be done by removing elements occurring in both $LEFT_1$ and $RIGHT_2$ at the same y-coordinate, because we assumed the y-coordinates of all rectangles to be distinct. If we permit multiple y-coordinates then we have to choose a different technique, see below). - A second scan operates on lists $VERT_1$ and $VERT_2$ and the reduced lists representing $LEFT_1$-LR and $RIGHT_2$-LR. During this scan VERT is constructed and the sets $(LEFT_1\text{-}LR) \# VERT_2$ and $(RIGHT_2\text{-}LR) \# VERT_1$ are computed and reported.

The computation of $P \# I$ where P and I are given as linked lists L_P and L_I is the only nontrivial (but not difficult) task. L_P and L_I are scanned in parallel. For each interval $i = [y_b, y_t]$ encountered in L_I this scan (called the main scan) pauses. From the current position in L_P (which corresponds to y_b) a report scan is started which reports a pair (p, i) for each point $p = y'$ encountered in L_P. The report scan terminates as soon as a y-coordinate $y' > y_t$ is encountered. Then the main scan is resumed.

Since the size of all lists is $O(n)$ the total time for scanning them (excluding report scans) is also $O(n)$. Hence the merge step contributes an $\underline{O(n)}$ term if we count the time for reporting separately.

Thus we have: $$T(n) = O(1) + 2\ T(n/2) + O(n) \tag{2}$$

It is well known that the recurrence equations (1) and (2) have the solution

$$T(n) = O(n \log n)$$

(see for instance [AHU]). Obviously the report scans require time linear in the number of reported pairs (which correspond to intersections). Hence, if k is the number of pairwise intersections between H and V, the total worst-case time required by the algorithm is $O(n \log n + k)$.

Space.

The space required is $O(n)$ since each of the six lists used by the algorithm contains at most n elements.

Multiple x- or y-coordinates.

We now drop the restriction that the x- and y-coordinates of all line segments have to be pairwise distinct. The task of our algorithm is to find all intersections between horizontal and vertical line segments. Finding intersections of type horizontal/horizontal or vertical/vertical is no problem (it amounts, for example, to sorting the horizontal line segments by y-coordinate and then for each subset with equal y-coordinate to find the intersections among a set of intervals) and can be done in a separate step in $O(n \log n)$ time (the line-sweep algorithm of Bentley and Ottmann [BeO] reports only intersections of type horizontal/vertical and would also require a separate step to report the other types).

To accomodate multiple x- or y-coordinates we modify the algorithm slightly. We first deal with multiple x-coordinates. After the initial sorting of all objects in $\overline{H} \cup V$ (points and vertical line segments) we are left with a set of groups $g_1, \dots, g_r$ of objects for which each group g_i contains one or more objects with the same x-coordinate.

Example:

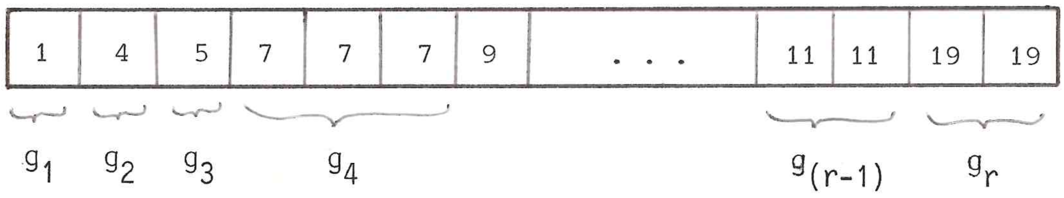


Figure 4-5

For each group g_i we construct the sets LEFT, RIGHT and VERT in advance and report the point enclosures (intersections) LEFT # VERT and RIGHT # VERT. Since those are one-dimensional problems the task only involves sorting the objects in the group by y-coordinate and can be done in $O(s \log s)$ time for a group with s elements. The total time for this step is $O(n \log n)$.

We now apply divide-and-conquer in the following way:

Let $S = \overline{H} \cup V$ contain the elements $x_1, \dots, x_m$ $(m \leq 2n)$. We select the median object $x_{\lfloor (n+1)/2 \rfloor}$. It belongs to some group g_i. We divide S into three subsets S_1, S_m and S_2 where

$$S_1 = \bigcup_{j=1}^{i-1} g_j, \quad S_m = g_i, \quad S_2 = \bigcup_{j=i+1}^{r} g_j.$$

We recursively apply the same algorithm to sets S_1 and S_2, constructing sets LEFT, RIGHT and VERT for each of them (remember that for S_m we constructed those sets beforehand).

After this S_1, S_m and S_2 are merged. This can either be done simultaneously or sequentially by first merging S_1 and S_m into S_{left}, then $S_{left}$and S_2 into S.

It is easy to see that this algorithm still has the same time complexity: By construction, each of the sets S_1 and S_2 contains less than n/2 elements. Merging S_1, S_m and S_2 takes $O(n)$ time. Hence the recurrence equations

$$T(1) = O(1)$$
$$T(n) \leq 2\,T(n/2) + O(n)$$

with solution $O(n \log n)$ hold again.

With multiple y-coordinates it becomes a problem to form the intersection of the sets $LEFT_1$ and $RIGHT_2$. We said previously that LEFT and RIGHT are represented by y-ordered point lists. If a point list contains many elements with identical y-coordinates then it becomes a problem to identify a pair of points in $LEFT_1$ and $RIGHT_2$ coming from the same horizontal line segment. However, there exists a simple way around this difficulty. Note that any set S to which LINSECT is applied contains all objects within a certain x-range. We pass this range as a parameter of LINSECT. Furthermore we add to the representation of a left (right) endpoint the x-coordinate of its right (left) partner endpoint. Merging for instance sets S_1 and S_m with associated x-intervals (a, b) and [b, b],say, it is then possible to decide for any point in $LEFT_1$ and $RIGHT_m$ whether its partner endpoint is in the other set. Hence it is once more possible to perform the subtraction of set LR (= $\text{LEFT}_1 \cap \text{RIGHT}_m$, in this case) in linear time.

In this way the time complexity of the algorithm is maintained for the general case which permits multiple x- or y-coordinates.

We summarize the results of this section in

Theorem 4.1: For a set of n horizontal and vertical line segments in 2-space with k intersections the line segment intersection problem can be solved by divide-and-conquer in $O(n \log n + k)$ time and $O(n)$ space.

4.2. *Finding point enclosures.*

The point enclosure problem is:

> Given n objects each of which is a point or a rectangle in the plane, report for each rectangle all points that lie within it.

A first time-optimal solution of this problem was given by Bentley and Wood [BeW]. Independently Edelsbrunner [Ed2] and McCreight [Mc] found time- and space-optimal solutions. All mentioned solutions use line-sweep algorithms.

Remember that the study of this problem is motivated by the fact that we can combine its solution with the solution of the line segment intersection problem to find all intersections among a set of rectangles. Again we use separational representation and divide-and-conquer and assume for simplicity at the moment that all objects have pairwise distinct coordinates.

The idea.

Separational representation applied to the given set reduces each rectangle to its left and right vertical edge and leaves the points unchanged. Hence we obtain a mixed set of points and vertical line segments which we sort by x-coordinate. To the resulting ordered set S the divide-and-conquer algorithm PENC is applied.

PENC splits a given set S into two subsets S_1 and S_2 and recursively computes y-ordered interval sets $LEFT_i$ (the y-projections of left rectangle edges), $RIGHT_i$ (the y-projections of right edges) and a set of y-coordinates $POINTS_i$ (y-projections of points) from S_i. The merge step which computes LEFT, RIGHT and POINTS from $LEFT_i$,

$RIGHT_i$ and $POINTS_i$ ($i = 1,2$) is once more the crucial step. It makes use of the following observation:

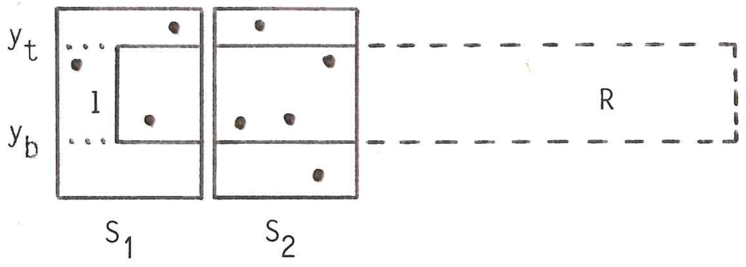


Figure 4-6

If a left vertical edge l in S_1 of a rectangle $R = (x_l, x_r, y_b, y_t)$ is not matched by its partner edge in S_2, then the bottom edge and the top edge of R extend through all of S_2. This means all points in the y-ordered set $POINTS_2$in the range $[y_b, y_t]$ are enclosed by R. $[y_b,y_t]$ is precisely the y-projection of l which is contained in $LEFT_i$.

To report all point enclosures within $S = S_1 \cup S_2$ we have to check whether or not the partner of each interval in $LEFT_1$ is in $RIGHT_2$. For each of the remaining intervals all enclosed points in $POINTS_2$ have to be reported (or rather the corresponding point-rectangle enclosure. Obviously the same has to be done for $RIGHT_2$ and $POINTS_1$. Using the notation $P \# I$ of Section 4.1 this is nothing other than computing and reporting $POINTS_2 \# (LEFT_1\text{-}LR)$ and $POINTS_1 \# (RIGHT_2\text{-}LR)$ where $LR = LEFT_1 \cap RIGHT_2$.

Hence the algorithm is a simple adaptation of LINSECT:

Algorithm POINT ENCLOSURE

Input: A set of points P and a set of rectangles R.

Output: The set of all pairs (p,r) where $p \in P$ and $r \in R$ and p lies inside r.

Step 1: Construct the sets L and R of left and right edges of rectangles in R, respectively. Sort $L \cup R \cup P$ by x-coordinate, resulting in the ordered set S.

Step 2: PENC(S, LEFT, RIGHT, POINTS)

end of algorithm POINT ENCLOSURE.

Algorithm PENC (S, LEFT, RIGHT, POINTS)

Input: An x-ordered set of points and left and right vertical rectangle edges S.

Output: Three sets LEFT, RIGHT and POINTS. LEFT and RIGHT contain the y-projections of all left and right rectangle edges from S, whose partner is not in S. POINTS contains the y-projection of $S \cap P$.

Recursive invariant: On exit, PENC(S, ...) has reported all point enclosures occurring within S, that is all pairs (p,r), where $p \in (P \cap S)$ and $r \in R$ and r is represented in S by its left or right vertical edge.

Case 1: S contains only a single object x.

Depending on the type of x (line segment or point) let LEFT or RIGHT contain a single y-interval or POINTS a y-coordinate, respectively, and let the other sets be $\emptyset$.

Case 2: S contains more than one object.

```
Divide:   Choose an x-coordinate dividing S into two subsets $S_1$ and $S_2$ of ap-
          proximately equal size.

Conquer:  PENC($S_1$, $LEFT_1$, $RIGHT_1$, $POINTS_1$);
          PENC($S_2$, $LEFT_2$, $RIGHT_2$, $POINTS_2$)

Merge:    (Let LR = $LEFT_1 \cap RIGHT_2$)
          output ($POINTS_2$ # ($LEFT_1$-LR))
          output ($POINTS_1$ # ($RIGHT_2$-LR))
          LEFT $\leftarrow$ ($LEFT_1$ - LR) $\cup$ $LEFT_2$
          RIGHT $\leftarrow$ $RIGHT_1$ $\cup$ ($RIGHT_2$ - LR)
          POINTS $\leftarrow$ $POINTS_1$ $\cup$ $POINTS_2$   and return.

end of algorithm PENC.
```

Time and space complexity.

We may choose the same linked-list representation for sets LEFT, RIGHT and POINTS as for LINSECT. In the merge step of PENC the same operations occur as in the merge step of LINSECT. Hence we know that they can be performed in linear time. The analysis is the same as for LINSECT and the algorithm may be adapted to multiple x- and y-coordinates by the same techniques. Hence we obtain

Theorem 4.2: For a mixed set of points and rectangles in 2-space with cardinality n, the point enclosure problem can be solved by divide-and-conquer in $O(n \log n + k)$ time and $O(n)$ space where k is the number of existing point enclosures.

4.3. *Finding rectangle intersections*.

The rectangle intersection problem is:

Given a set of n iso-oriented rectangles, report all pairwise intersections.

The first time-optimal solution for the problem was given by Bentley and Wood [BeW], that is $O(n \log n + k)$ time where k is the number of intersecting pairs. Edelsbrunner [Ed2] and McCreight [Mc] independently found time- and space-optimal solutions. Each of these algorithms is based on the line-sweep paradigm.

Algorithms solving the rectangle intersection problem have important applications in VLSI-design (see Baird [Ba], Lauther [La1] or Mead and Conway [MeC]). They are used to check a design of some VLSI-circuitry,specified in terms of rectangles, for consistency with a given set of 'design rules' (for more details see e.g. [BeW]). There exist other applications as well, for instance in architectural data bases [EaL].

We include this section for completeness, to show how our algorithms of Sections 4.1 and 4.2 can be combined to solve the rectangle intersection problem by divide-and-conquer. The idea to combine the solutions of the line segment intersecting problem and the point enclosure problem is due to Bentley and Wood [BeW]. Insofar this section does not contain new results.

Note that two rectangles intersect if either their edges intersect or one encloses the other one completely.

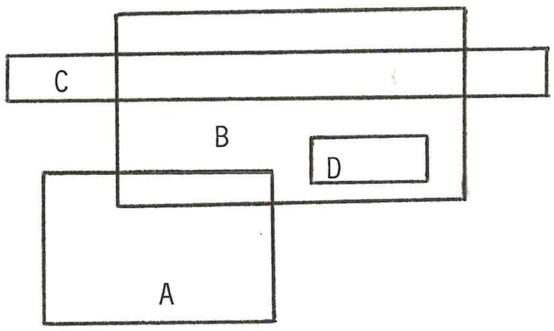


Figure 4-7

In Fig. 4-7 the pairs of rectangles (A,B) and (B,C) intersect by edge intersection, while (B,D) constitutes a rectangular enclosure. The two types of intersections can be found efficiently by treating them separately. We follow [BeW] in this approach. The result is a 2-stage algorithm which finds all edge intersections in the first step and all rectangular enclosures in the second.

In the first step we consider the set of all edges composing the rectangles, that is 4n edges, and solve the line segment intersection problem for them. In the second step we extend the set of rectangles to include an interior point for each rectangle. We then solve the point enclosure problem for the resulting set. Letting R be a rectangle and p_R be the chosen interior point for R, then it is important to realize that a report 'A encloses p_B' holds not only when B is enclosed by A, but may also hold when B is not enclosed by A. However, in this latter case B and A have a point in common and hence intersect (see [BeW] for more details).

Of course, when using the algorithms LINE SEGMENT INTERSECTION and POINT ENCLOSURE combined to solve the rectangle intersection problem, some simplifications may be introduced. The preparatory steps in both algorithms can be replaced by a single scan through the set of rectangles, creating all input sets as needed. Instead of line segment names and point names we store, of course, the names of the corresponding rectangles (in an implementation the 'name' is usually the address of some representation of the rectangle).

In an implementation some details have to be taken care of, such as avoiding multiple reporting (two rectangles whose edges intersect, have at least two different edge intersections) and not reporting the intersection of a rectangle with itself (two adjacent edges intersect, of course). However, these details do not affect the asymptotic complexity of the algorithm. They merely require careful programming.

The time and space requirements of the algorithm follow immediately from those of the component algorithms:

Let n be the number of rectangles, k the number of intersecting pairs of rectangles, k' the number of edge intersections and k" the number of point enclosures. The algorithm LINE SEGMENT INTERSECTION requires $O(n \log n + k')$ time. POINT ENCLOSURE requires $O(n \log n + k'')$ time. Since $k' = O(k)$ and $k'' = O(k)$ we have a total time bound of $O(n \log n + k)$. This time is known to be optimal, see [BeW]. Space is $O(n)$ for both component algorithms.

Theorem 4.3: For a set of n iso-oriented rectangles in 2-space the rectangle intersection problem can be solved by divide-and-conquer in $O(n \log n + k)$ time and $O(n)$ space where k is the number of pairwise rectangle intersections.

4.4. Computing measure and contour for a set of rectangles.

We are now ready to attack the most difficult of the problems considered in Section 4 which is to compute the contour of a set of rectangles. The contour problem has already been extensively discussed and solved by a line-sweep algorithm in Section 3.1. The reader may be surprised that we include the measure problem in this section. The measure problem is to compute the size (the two-dimensional measure) of the union of a set of rectangles and we discussed already a generalized version of it (i-measures) in Section 3.2. An optimal line-sweep solution for the measure problem was given by Bentley [Be2].

The reason for including the measure problem here is that both the measure and the contour are properties of the union of a set of rectangles. We will exploit this fact by developing an abstract description D of this union which makes the solution of both the measure and the contour problem quite easy.

The central part of this section is the description of a high-level divide-and-conquer algorithm A to compute D. To solve one of the specific problems a convenient representation of D is chosen and the abstract operations of A are implemented to compute this representation.

Hence our approach has two steps:

1.) Given a set of rectangles R, compute the description of their union D according to problem X, using algorithm A.
2.) Given D, solve problem X.

Since step 2 is quite simple for both problems, it is essentially the one powerful algorithm A which solves both problems.

This section is structured as follows: First we define the description D of the union of a set of rectangles (Section 4.4.1). We then show how each of the two problems can be solved once D is given (Section 4.4.2). In Section 4.4.3 we describe a fairly abstract high-level algorithm A which computes D by divide-and-conquer. Since it is not obvious how to represent D and implement the operations of A Section 4.4.4 deals with these issues. In that section we also analyse time and space requirements.

4.4.1. *An abstract description of the union of a set of rectangles.*

Now let R be the given set of rectangles. The description we choose is a partition of the plane into horizontal stripes. For technical reasons we want to exclude infinite stripes, hence we first choose arbitrarily a rectangular 'frame' f completely enclosing the given set of rectangles. The area of this frame is then divided into horizontal rectangular stripes. This partition is imposed by the horizontal edges of rectangles in R, that is, each edge coincides with a stripe's boundary (for simplicity we assume at present the x-coordinates and y-coordinates of all rectangles to be pairwise distinct):

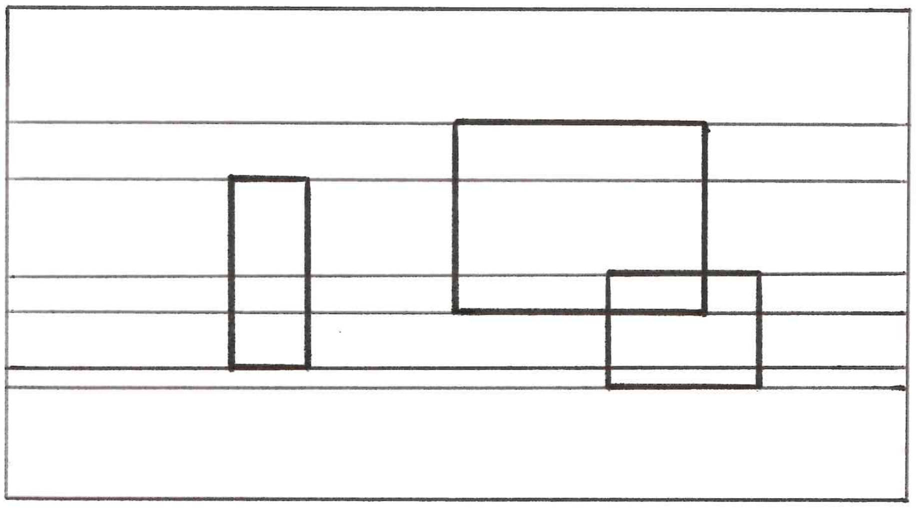


Figure 4-8

Note that the part of the union of the rectangles in R that falls into any particular stripe is completely defined by a set of disjoint x-intervals.

To describe this precisely we recall a few definitions from Section 2. y-set(R) is the set of all y-coordinates of bottom or top edges in R. For a set of points P in 1-space, partition(P) defines the set of 'atomic' intervals into which the line is divided by P (for formal definitions see Section 2.).

We then define the set of stripes:

Let R be a set of rectangles and $f = (x_1, x_2, y_1, y_2)$ a rectangle such that y-set(R) $\subset (y_1, y_2)$ (the latter denoting an open interval). Let RY be y-set(R) $\cup \{y_1, y_2\}$. Then

stripes(R,f) := $\{(i_x, i_y, \text{x-union}) \mid i_x = [x_1, x_2]$, i_y is in partition (RY) and $\underline{\text{x-union}} = \text{x-proj}\ ((i_x \times i_y) \cap \text{union}\ (R))\}$.

So every stripe is a triple $(i_x, i_y, \text{x-union})$ where $i_x \times i_y$ defines the rectangular area of the stripe and x-union is a set of disjoint intervals representing the part of union(R) inside the stripe.

Note that the definition is still valid if the x-interval of frame f does not contain the whole x-extension of R. Thus a frame 'cuts out' a vertical stripe from the set of rectangles.

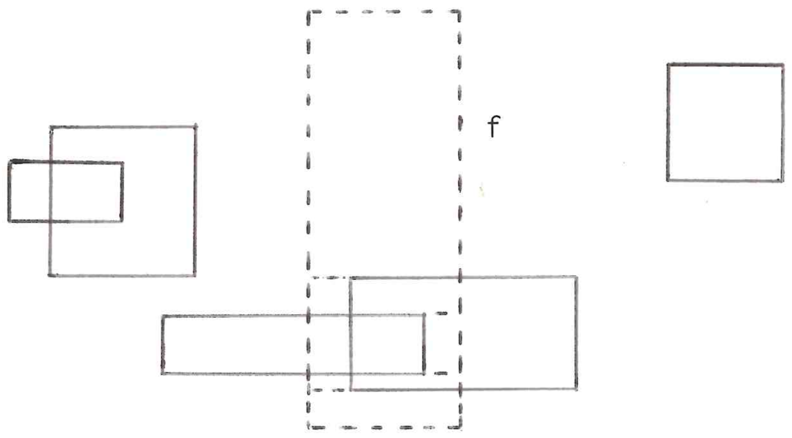


Figure 4-9

This is essential for the divide-and-conquer algorithm which merges two adjacent vertical stripes (frames) into a new one.

To provide some motivation we will now show that given a set of stripes the measure problem and the contour problem can easily be solved.

4.4.2. *Reducing the problems.*

The measure problem.

Given a set of rectangles R our task is to compute measure(R). First we choose a frame completely enclosing and not intersecting any of the rectangles in R. We then compute S = stripes(R,f) by the divide-and-conquer algorithm given in the next section.

Now each stripe s in S contains a set of disjoint intervals x-union(s). The only property of x-union(s) relevant to the measure problem is the total length of the contained intervals, that is measure(x-union(s)). Then

$$\text{measure}(R) \leftarrow \sum_{s \in S} \text{measure}(\text{x-union}(s)) \cdot i_y(s).$$

The contour problem.

Recall that for a set of iso-oriented rectangles R the contour is a collection of contour-cycles. Each contour-cycle is a sequence of alternating horizontal and vertical contour-pieces. Every contour-piece is a fragment of a rectangle edge from R.

Lipski and Preparata [LiP] have shown that it is sufficient to know the contour-pieces oriented in one direction (for instance the horizontal ones) to construct the complete contour efficiently. Our algorithm therefore only computes the horizontal contour-pieces.

Let HCP denote the set of horizontal contour-pieces and H the set of horizontal rectangle edges defined by R. For an edge $h \in H$ let CP(h) denote the set of contour-pieces generated by it. Given $S = \text{stripes}(R,f)$ (for some frame f) we can easily compute CP(h) for any $h \in H$. By construction of S, h coincides with the boundary between two stripes:

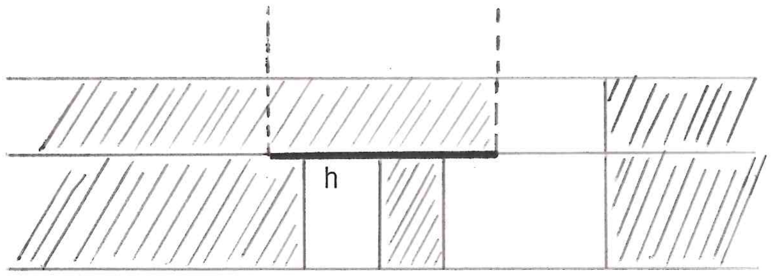


Figure 4-10

The contour-pieces generated by h are given by the intersection of h with 'free' area in an adjacent stripe. In one of the stripes there cannot exist any free area intersecting h because it is covered by the rectangle to which h belongs. Hence if h is a bottom edge the stripe below it has to be examined, if it is a top edge, the stripe above it.

Now we compute CP(h) in the following way:

Let $h = (x_1, x_2, y, \text{side})$ where side is in {bottom, top}.

```
if side(h) = bottom
then select $s \in S$ with $i_y(s) = [y',y]$
else select $s \in S$ with $i_y(s) = [y,y']$
fi;
$I \leftarrow [x_1, x_2] - ([x_i, x_2] \cap \text{x-union}(s))$;
$CP(h) \leftarrow (I,y)$
```

After selecting the correct stripe s, I is assigned the set of free intervals hit by h. Together with h's y-coordinate I defines a set of horizontal line segments. These are exactly the horizontal contour-pieces generated by h.

Since $HCP = \bigcup_{h \in H} CP(h)$, we are done.

4.4.3. *Computing the set of stripes.*

We now describe how to compute the set of stripes S = stripes(R,f). We have seen that, given S, the measure problem and the contour problem could be solved easily; hence the power of the algorithm is certainly to be found in this step.

We now describe in a top-down manner the algorithm which uses subalgorithms yet to be defined. We then explain the algorithm together with describing the subalgorithms.

The algorithm consists of two parts: a quite simple main algorithm RECTANGLE-DAC which only provides an environment for the recursive divide-and-conquer algorithm STRIPES.

Algorithm RECTANGLE-DAC

Input: A set of rectangles R.
Output: S = stripes(R,f), for some frame f.

Step 1: Let VR be the set of vertical rectangle edges defined by R. Let each vertical line segment v in VR be given as a quadruple $v = (x, y_b, y_t, \text{side})$ where side is in {left, right}. Let VRX be VR sorted by x.

To construct a frame f determine the minimal and maximal x- and y-coordinates in R, x_{min}, x_{max}, y_{min} and y_{max}. Select some arbitrary numbers x-, x+, y- and y+, such that $x- < x_{min}$, $x_{max} < x+$, $y- < y_{min}$ and $y_{max} < y+$. Let f = (x-, x+, y-, y+).

Step 2: STRIPES (VRX, x-, x+, S, LEFT, RIGHT, POINTS)

end of algorithm RECTANGLE-DAC.

(The parameters LEFT, RIGHT and POINTS of algorithm STRIPES are explained below).

Algorithm STRIPES (V, a, b, S, L, R, P)

Input: V - a set of vertical rectangle edges.

a, b - x-values bounding the line segments in V, that is $a < x(v) < b$ for all $v \in V$.

Output: L - a set of y-intervals, containing the y-projections of all left vertical edges in V whose partner (the corresponding right edge) is not in V.

R - symmetric to L (for right edges).

P - an ordered set of y-coordinates containing the y-projections of all endpoints of line segments in V plus the frame boundaries y-, y+.

S - a set of stripes: Let RECT(V) be the subset of rectangles in R which are represented in V by at least one vertical edge.
S = stripes (RECT(V), (a, b, y-, y+)).

Case 1: V contains only one vertical line segment $v = (x, y_1, y_2, s)$.

```
if s = left
then L ← {[$y_1$, $y_2$]} , R ← ∅
else L ←   ∅        , R ← {[$y_1$, $y_2$]}
fi;
P ← {y-, $y_1$, $y_2$, y+}
S ← PARTITION (P, a, b)
INITIALIZE (S,v)
and return.
```

```
Case 2:     V contains more than one vertical line segment.

  Divide:   Choose an x-coordinate $x_\mu$ dividing V into two approximately equal-
            sized subsets $V_1$ and $V_2$.
  Conquer:  STRIPES $(V_1, a, x_\mu, S_1, L_1, R_1, P_1)$;
            STRIPES $(V_2, x_\mu, b, S_2, L_2, R_2, P_2)$
  Merge:    Let $LR = L_1 \cap R_2$

            $L \leftarrow (L_1 - LR) \cup L_2$
            $R \leftarrow R_1 \cup (R_2 - LR)$
            $P \leftarrow P_1 \cup P_2$

            $S_{left} \leftarrow$ PARTITION $(P, a, x_\mu)$
            $S_{right} \leftarrow$ PARTITION $(P, x_\mu, b)$
            COPY $(S_1, S_{left})$; COPY $(S_2, S_{right})$
            BLACKEN $(S_{left}, R_2 - LR)$; BLACKEN $(S_{right}, L_1 - LR)$
            $S \leftarrow$ PARTITION $(P, a, b)$
            COMBINE $(S_{left}, S_{right}, S)$
            and return.

end of algorithm STRIPES.
```

Let us now look at the algorithm STRIPES in detail and define the used subalgorithms. Recall that the task of the algorithm is to compute for a set of rectangles R, which is given by its vertical edges V, the set S=stripes(R,f). The additional information only supports this.

The algorithm starts with the complete set of vertical rectangle edges V enclosed by some frame f. A vertical line is chosen which splits f into two smaller subframes and V into two subsets of equal size enclosed by the subframes. Recursively this process is continued yielding smaller and smaller frames (vertical stripes) until finally a frame contains only one vertical edge.

This is Case 1 of the algorithm which constructs L, R, P and S for this single edge. It is easy to check that after completion L, R and P fulfill the output specification, though we didn't explain yet the purpose of these sets. To construct S, two subalgorithms PARTITION and INITIALIZE are invoked.

PARTITION creates from a set of y-coordinates P and two x-coordinates a and b a set of stripes whose x-union components are initialized with the empty set (of x-intervals):

```
PARTITION (P,a,b) =
        $\{([a,b], i_y, \emptyset) \mid i_y$ is in partition(P)$\}$                    (A)
```

INITIALIZE is given a set of stripes S and a vertical edge v. S contains exactly one stripe whose y-interval equals that of v. This stripe's x-union field is then initialized.

```
INITIALIZE (S, v) =

    comment $v = (x, y_1, y_2, side)$;
    select $s \in S$ with $i_y(s)=[y_1,y_2]$
    if side(v)=left
    then x-union(s) $\leftarrow \{[x, b]\}$                              (B)
    else x-union(s) $\leftarrow \{[a, x]\}$                              (C)
    fi.
```

It is easy to verify that the algorithm computes S = stripes ($\{r\}$, (a, b, y-, y+)) where r is the rectangle from which v is taken. Hence in Case 1 the output specification, which serves as a recursive invariant, holds.

The resulting set of stripes can be represented (for instance in case of a left edge):

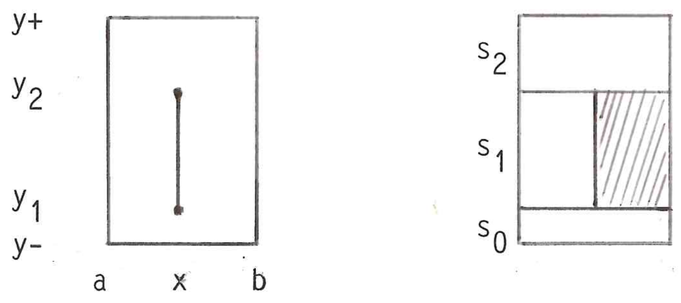


Figure 4-11

In Case 2, the algorithm starts with a set of vertical edges V located within some frame f.

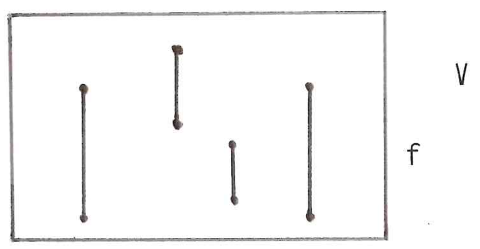


Figure 4-12

In the divide step this area is divided into two subareas f_1 and f_2, also splitting V into V_1 and V_2.

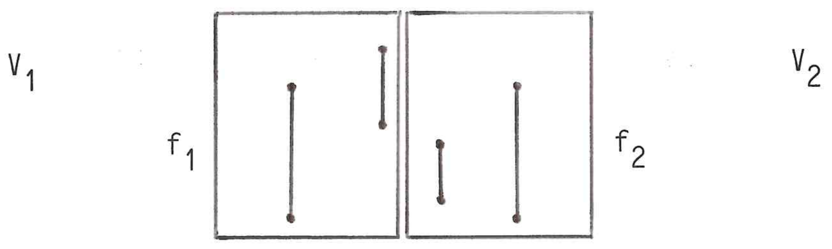


Figure 4-13

In the conquer step for each of these subsets and subareas a set of stripes is constructed:

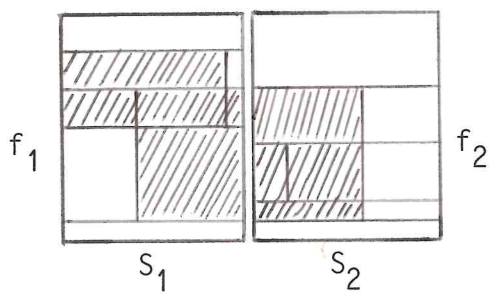


Figure 4-14

In the merge step the set of stripes S representing V is constructed from S_1 and S_2. Note that any horizontal partition occurring in S_1 or S_2 must also be a partition of S. Otherwise there would be horizontal edges within a stripe of S. The algorithm therefore first constructs two copies S_{left} and S_{right} of S_1 and S_2, respectively, which are based on the complete set of partition points $P = P_1 \cup P_2$.

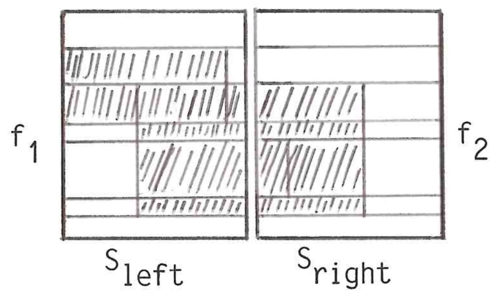


Figure 4-15

This is done by the statements

$S_{left} \leftarrow$ PARTITION(P,a,x_μ); $S_{right} \leftarrow$ PARTITION(P,x_μ,b)
COPY (S_1, S_{left}); COPY (S_2, S_{right})

where for two sets of stripes S, S'

```
COPY (S, S') =

    For all s' ∈ S':
        select s ∈ S such that $i_y(s') \subseteq i_y(s)$
        x-union(s') ← x-union(s)                                    (D)
```

Now consider an arbitrary edge v in V belonging to a rectangle r. Since $V = V_1 \cup V_2$, v is either in V_1 or in V_2. Without loss of generality assume it is in V_1 (the other case is symmetric). Then r is in RECT(V_1). Since S_1 = stripes(RECT(V_1), f_1), r is represented in S_1 and also in S_{left}. Therefore we might only have to update S_{right} with regard to the intersection of r with f_2. There are four cases:

a) v is a right edge.

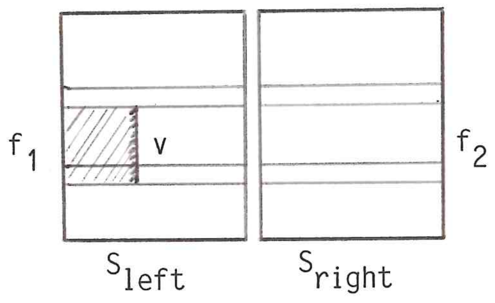


Figure 4-16

Then r does not intersect f_2. S_{right} does not have to be updated.

b) v is a left edge with partner in V_1.

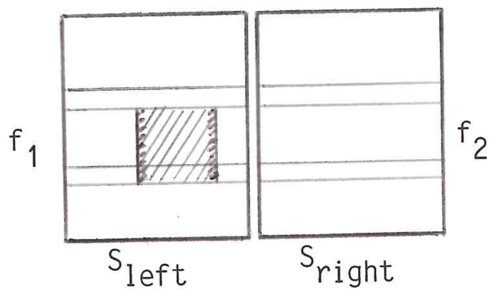


Figure 4-17

Again r does not intersect f_2 and S_{right} remains unchanged.

c) v is a left edge with partner in V_2.

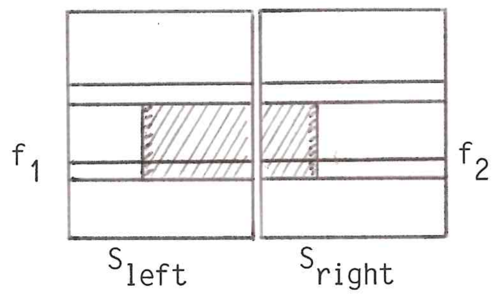


Figure 4-18

Then the intersection of r with f_2 is already represented in S_{right}, because the right edge participated in the construction of S_{right}. S_{right} does not have to be updated.

d) v is a left edge and the right edge of r is neither in V_1 nor in V_2.

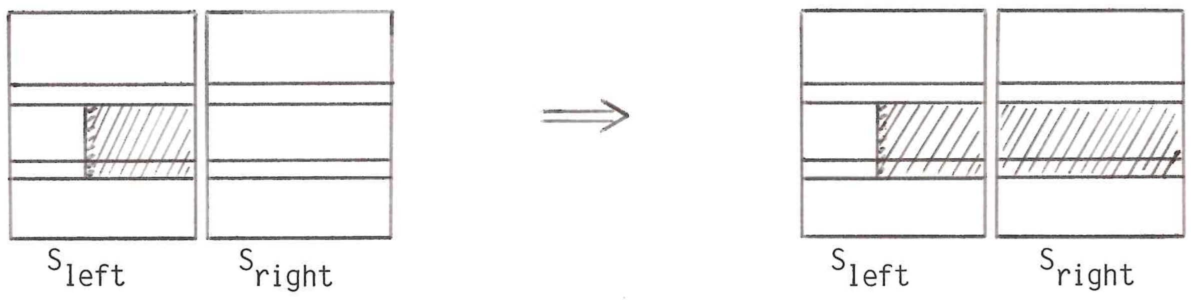


Figure 4-19

In this case S_{right} has to be updated. Since the right edge of r is not in V_2, rectangle r covers the whole x-extension of area f_2. This is the crucial point on which the whole efficiency of the algorithm depends! No updating is necessary <u>within</u> a stripe. Instead, any stripe in S_{right} either remains unchanged or becomes completely covered by r.

In this way the updating of S_{left} and S_{right} has to be performed for any unmatched left edge in V_1 and right edge in V_2, respectively. This is done by the statements

BLACKEN(S_{left}, R_2 - LR); BLACKEN(S_{right}, L_1 - LR).

The set L_1 - LR contains exactly the y-projections of unmatched left edges in V_1 corresponding to Case d). Any stripe in S_{right} whose y-interval is contained in at least one interval in L_1 - LR is completely covered by a rectangle and has therefore to be 'blackened':

```
BLACKEN (S, I) =

    (for a set of stripes S and a set of y-intervals I. Let [a, b] be the
    x-extension of S)

    For all s in S:
        if there exists i in I such that $i_y(s) \subseteq i$
        then x-union(s) $\leftarrow$ {[a, b]}                              (E)
        fi.
```

After this the stripes of S are obtained by simply concatenating the corresponding stripes of S_{left} and S_{right}:

```
    S $\leftarrow$ PARTITION(P, a, b)
    COMBINE($S_{left}$, $S_{right}$, S)
```

```
where COMBINE ($S_1$, $S_2$, S) =

    For all s in S:
        select $s_1$ from $S_1$ such that $i_y(s_1) = i_y(s)$
        select $s_2$ from $S_2$ such that $i_y(s_2) = i_y(s)$
        x-union(s) $\leftarrow$ x-union($s_1$) & x-union($s_2$)            (F)
```

The operation I & I' denotes the concatenation of two sets of intervals (basically the union, but two adjacent intervals at the boundary are merged into one).
We omit a formal definition. - It follows from the discussion, that now S = stripes (RECT(V), (a, b, y-, y+)). The construction of the sets L, R and P in the merge step is quite obvious. Hence the output specification is true again.

4.4.4. *Implementation.*

In Sections 4.4.2. and 4.4.3 we have shown how to solve the measure and the contour problem. Data and operations on them were described set-theoretically for precision and clarity. However, it is not obvious how to represent the data and implement the abstract operations. This we will describe now.

We have already seen in Section 4.4.2. that the set of stripes is represented in different ways for the measure and the contour problem. We first describe these representations. Later we look at the implementation of the whole algorithm and determine its time and space requirements.

The measure problem.

The question is for both the measure and the contour problem how to represent x-union(s). Recall that to solve the measure problem only measure(x-union(s)) is needed. Hence we simply represent x-union(s) by a real number x-measure(s).

In Section 4.4.3. all lines describing operations on x-union(s) are marked by capital letters. To solve the measure problem we simply replace these lines by the following:

```
(A)    PARTITION(P,a,b) = $\{([a, b], i_y, 0) \mid i_y \in \text{partition}(P)\}$
(B)    x-measure(s) ← (b-x)
(C)    x-measure(s) ← (x-a)
(D)    x-measure(s') ← x-measure(s)
(E)    x-measure(s) ← (b-a)
(F)    x-measure(s) ← x-measure($s_1$) + x-measure($s_2$)
```

It is easy to check that these operations maintain the x-measure of a stripe in the proper way.

The contour problem.

As we have seen in Section 4.4.2 the representation of x-union(s) must support a range query:

> Given an interval $[x_1, x_2]$ and a set of disjoint x-intervals I, report the 'free' intervals hit by $[x_1, x_2]$ (more formal: report $[x_1, x_2] - (I \cap [x_1, x_2])$).

The representation we choose is a binary searchtree storing the endpoints of the intervals in x-union(s) in its leaves. Hence each stripe contains a field contour-tree(s). The structure of the tree is defined as follows:

$$T = \begin{cases} \lambda & \text{if it is the empty tree} \\ (x, \text{side}, T_l, T_r) & \text{otherwise,} \end{cases}$$

where x is an x-coordinate,

$$\text{side} = \begin{cases} \text{u (undefined)} & \text{if T is not a leaf} \\ \text{left} & \text{if T is a leaf storing the left end of an interval} \\ \text{right} & \text{if T is a leaf storing the right end of an interval,} \end{cases}$$

and T_l and T_r denote the left and right subtree, respectively.

Again we describe operations (A) - (F):

(A) PARTITION(P,a,b) = $\{([a, b], i_y, \lambda) \mid i_y \in \text{partition}(P)\}$

(B) contour-tree(s) $\leftarrow$ (x, left, λ , λ)

(C) contour-tree(s) $\leftarrow$ (x, right, λ , λ)

(D) contour-tree(s') $\leftarrow$ contour-tree(s)

(this is to be understood as a pointer operation)

(E) contour-tree(s) $\leftarrow$ λ

(F) contour-tree(s) $\leftarrow$ (x_μ, u, contour-tree(s_1), contour-tree(s_2));

(Now for abbreviation let contour-tree(s) = T = (x_μ, u, T_l, T_r))

$$T \leftarrow \begin{cases} \lambda & \text{if } T_l = \lambda = T_r \\ T_l & \text{if } T_l \neq \lambda = T_r \\ T_r & \text{if } T_l = \lambda \neq T_r \\ T & \text{if } T_l \neq \lambda \neq T_r \end{cases}$$

(The second step prevents the construction of trees with empty subtrees).

From the point of view of implementation it is important to note that the field contour-tree of a stripe and also the fields T_l and T_r contain pointers. Especially the operations (E) and (F) do not copy trees but manipulate pointers. This has the effect that two trees belonging to different stripes may have common subtrees and in total a quite complicated interlinked structure is constructed. On the conceptual level, however, this doesn't matter and we can still maintain the view that each stripe has its own separate tree. - A similar construction of interlinked trees is made in [GüW] and is there explained in more detail.

Since contour-tree(s) is a searchtree, it is certainly possible to answer a range query with interval $[x_1, x_2]$ by visiting all nodes in the tree with x-values between x_1 and x_2. The additional information left/right in a leaf makes it possible to report a 'free' interval [a, b] hit by $[x_1, x_2]$ for every pair of subsequent leaves (a, right, λ , λ) and (b, left, λ , λ) found between x_1 and x_2. This takes $O(h + p)$ time where h is the height of the tree and p the number of reported 'free' intervals.

The whole algorithm and its complexity.

The initial set V can be represented by an array; subsets then correspond to array-subranges. All the sets S, L, R and P are implemented by linked lists ordered by y-coordinate. In case of the sets L and R, which contain intervals, each interval is represented by its bottom and top point (so L and R are represented by point lists). Associated with each point is a number +1, if it is a bottom point, and -1, if it is a top point. Scanning the list representing L, for instance, by summing the encountered numbers it is easy to determine whether the current position is covered by any interval in L or is free. This is needed for the BLACKEN operation. Matching points in both lists which represent L and R are omitted to perform the subtraction of the intersection set LR.

Time.

Let us now look at the time complexity of the algorithm STRIPES. Let T(n) denote the worst-case time STRIPES requires, applied to a set of cardinality n. Obviously all operations in Case 1 take only constant time, hence

$$T(1) = O(1) \qquad (1)$$

In Case 2, dividing can be done in constant time and the conquer step yields two terms T(n/2). All operations in the merge step can be performed by scanning linked lists in parallel. We leave it to the reader to check that the operations performed during those scans take only constant time per list element. Hence the operations of the merge step require linear time in total. Therefore

$$T(n) = O(1) + 2\,T(n/2) + O(n) \qquad (2)$$

We encountered the same recurrence equations already in Section 4.1, with solution

$$T(n) = O(n \log n).$$

Space.

Clearly the array representing S and the linked lists representing L, R and P require O(n) space. The space requirements of the data structure representing the set of stripes S are different for the measure and the contour problem.

In case of the measure problem the representation of each stripe takes only constant space, hence the whole set is stored in O(n) space.

In case of the contour problem we have to look at the way a set of stripes is constructed. Let SP(n) denote the worst-case space requirements of a set of stripes constructed by algorithm STRIPES when applied to a set V of cardinality n. Case 1 of the algorithm constructs a set S using constant space, hence

$$SP(1) = O(1) \qquad (1)$$

In Case 2 of the algorithm the set of stripes is constructed in the merge step. Observe that a new linked list is created with O(n) elements (each element representing a stripe) and that for each list element at most a new root node and two pointers are created. The tree structures belonging to stripes of S_1 and S_2 are retained. Hence

$$SP(n) = 2\ SP(n/2) + O(n). \tag{2}$$

These are the same recurrence equations as above with solution

$$SP(n) = O(n \log n).$$

We have to add a few words on the complexity of obtaining the actual solution for each problem from the data structure representing S. How to obtain this solution is described in Section 4.4.2. For the measure problem the linked list representing S is scanned, summing the 2-dimensional measure of each stripe. Clearly $O(n)$ time is sufficient. For the contour problem the horizontal rectangle edges are sorted by y-coordinate into a list HRE. This list is then scanned in parallel with the linked list of stripes. For each horizontal edge a range query is performed on the corresponding stripe's contour-tree. Since the height of this tree is $O(\log n)$, each range query takes $O(\log n + k)$ time (k the number of reported contour-pieces) yielding a total time of $O(n \log n + p)$ where p is the total size of the contour (the number of contour-pieces). Using the method of [LiP] the contour can then be constructed as a set of linked contour-cycles in $O(p)$ time and space.

So far we discussed the problems under the restriction that the x- and y-coordinates of all rectangles are distinct. The methods for modifying the algorithm for the general case are given in Section 4.1 and are not repeated here (in [Gü3] those methods are described for algorithm STRIPES).

We summarize the results of this section in:

Theorem 4.4: For a set of n rectangles the measure problem and the contour problem can be solved by divide-and-conquer. This takes O(n log n) time and O(n) space in case of the measure problem and O(n log n + p) time and space for the contour problem, where p is the size of the contour.

5. *NON-ORTHOGONAL PROBLEMS: EFFICIENT ALGORITHMS FOR C-ORIENTED POLYGONS.*

Up to now we restricted our attention to problems involving orthogonal objects in 2-space, that is points, horizontal and vertical line segments and axis-parallel rectangles. We thereby reflect a trend in computational geometry to study this kind of problems first. In fact, most of the research in computational geometry has so far been done for orthogonal objects. The reason for this is, of course, that the orthogonal case is much easier to treat algorithmically. That means solutions are easier to devise and the resulting algorithms tend to be much more efficient than those for the general case.

In the general case the objects naturally considered are points, (arbitrarily oriented) line segments and polygons. If we reformulate for instance the problems involving rectangles in terms of polygons, then very often solutions are not yet known. If they are known they are usually of higher time or space complexity. The gap in complexity can already be seen from one of the most basic problems: Finding all k intersections in a set of n line segments can be done in $O(n \log n + k)$ time ([BeO] or by divide-and-conquer, see Section 4.1) for axis-parallel line segments while it takes $O(n \log n + k \log n)$ time for arbitrarily oriented line segments [BeO].

A natural extension of our work on contour problems would be to develop an algorithm which computes the contour of (the union of) a set of polygons. Such an algorithm exists already as a special case of a more general algorithm for boolean mask operations due to Ottmann, Widmayer and Wood [OWW]. For a set of n polygons with k edge intersections it requires $O(n \log n + k \log n)$ time. Note that the bound depends on the number of edge intersections k rather than on the actual size of the contour p as it does in the orthogonal case

(O(n log n + p) time). Since p may be much smaller than k the algorithm also illustrates the gap in complexity between the orthogonal and the non-orthogonal case. Still it seeems to be the best we can hope for since in the general case apparently one cannot avoid to compute all edge intersections which immediately brings about the time bound.

In this situation the question arises whether it is possible to bridge the complexity gap in some way by considering objects more general than rectangles but still so restricted that efficient solutions are possible. This is the topic of Section 5. We define so-called c-oriented objects whose edges are oriented in only a constant number of previously defined directions. We show the usefulness of this notion by giving algorithms and data structures achieving bounds distinctly better than those for the general case and sometimes optimal.

It turns out that the resulting contour algorithm for c-oriented polygons is not very exciting since its time bound still depends on the total number of edge intersections instead of the size of the contour. However, the algorithm saves a factor of log n compared to the general one because the edge intersections can be found more efficiently. We postpone the description of the contour algorithm to Section 5.3 because it makes use of some elementary techniques developed in the preceding sections.

In Section 5.1 we consider two versions of the stabbing number problem. This is to determine how many (c-oriented) polygons from a given set contain a query point. We devise time-optimal solutions for both considered problems. In fact, they are of the same complexity as those for the orthogonal case.

In Section 5.2 we consider another searching problem called polygonal intersection searching. Given a set of polygons S and a query polygon p we ask for all polygons from S which intersect p. The problem has been studied extensively for orthogonal objects where it can be solved in optimal $O(\log n + k)$ time (for references see Section 5.2). Surprisingly the same time bound has been achieved for the general case. However, this latter solution uses prohibitive amounts of space. For the restricted case of c-oriented polygons we are able to achieve an $O(\log^2 n + k)$ time bound within still reasonable space ($O(n \log^2 n)$).

The results of Section 5.1 have been published previously in [Gü2].

5.1. *The stabbing number problem.*

In computational geometry the stabbing problem has been examined in various contexts. In one dimension stabbing involves a set of intervals and a query point q. We either want to determine the set of intervals containing q (the Stabbing Set Problem) or just the number of 'covering' intervals (the Stabbing Number Problem). The stabbing set problem was solved independently by Edelsbrunner [Ed1] and McCreight [Mc] in $O(\log n + k)$ time and $O(n)$ space for a set of n intervals of which k contain q. Recently Gonnet, Munro and Wood [GoMW] solved the problem of finding the stabbing number in $O(\log n)$ time and $O(n)$ space. For this problem also a simpler solution achieving the same time and space bounds is sketched in Edelsbrunner and Overmars [EdO] as a special case of rectangle intersection counting. The mentioned results support dynamic sets of intervals while both problems had been solved earlier with the same time bound by means of a segment tree for a static set of intervals (see [BeW]).

In two dimensions the problem may be formulated as follows: Given a set of n polygons in 2-space and a query point q, determine all polygons enclosing q (the set problem) or the number of enclosing polygons (the number problem). The only solution for this general case is by Edelsbrunner, Kirkpatrick and Maurer [EdKM]. It uses $O(\log n + k)$ time for the stabbing set problem and $O(\log n)$ time for the stabbing number problem, which is optimal. Unfortunately space and preprocessing costs are very high (all costs are at least $\Omega(n^2)$).

Restricting to orthogonal objects, that is iso-oriented rectangles, more space-efficient solutions are known. Given a set of n rectangles we can determine the subset enclosing a query point q, with cardinality k, in $O(\log^2 n + k)$ and determine the stabbing number in $O(\log^2 n)$ time by means of a segment-segment tree, see [SiW], [EdM]. This query time has been improved to $O(\log n + k)$ and $O(\log n)$, respectively [Va].

In two dimensions it also makes sense to consider the batched problem: Given a set of n objects where each object is a rectangle or a point, compute for each point the set of enclosing rectangles or the stabbing number. This can be done in $O(n \log n + k)$ and $O(n \log n)$ time, respectively, reducing it to the one-dimensional problem by means of a line-sweep algorithm [BeW].

In the sequel we discuss the problem for c-oriented polygons and try to extend the efficiency of the orthogonal case to this more general class of objects. For them we solve the batched stabbing number problem in $O(n \log n)$ time and $O(n)$ space (Section 5.1.1) and the stabbing number query problem in $O(\log n)$ time and $O(n \log n)$ space (Section 5.1.2). The methods used in this section do not solve the stabbing set problem.. This is due to the collection of operations that can be performed for answering this problem. Subtraction is allowed for the number problem, while too time-costly for the set problem. In fact, our methods extend easily to stabbing problems which require the sum of the stabbing objects, where the terms leading to the sum are in some commutative group. -

In Section 5.2, however, we will show how to solve the stabbing set problem by different methods though with a higher query time of $O(\log^2 n + k)$ and $O(n \log^2 n + k)$ space-requirements.

5.1.1. *The batched problem.*

Let C be a finite set of lines in 2-space. C must be of rather small cardinality to make our solutions efficient. A polygon is C-oriented if for each of its edges there is a parallel line in C. We also speak of a set of c-oriented polygons, if there exists a set C of cardinality c such that all polygons are C-oriented.

Consider the batched stabbing number problem:

> Given C, and a set of C-oriented polygons and of points where the total number of edges and points is n (the size of any single polygon is not restricted), determine for each point the number of polygons enclosing it.

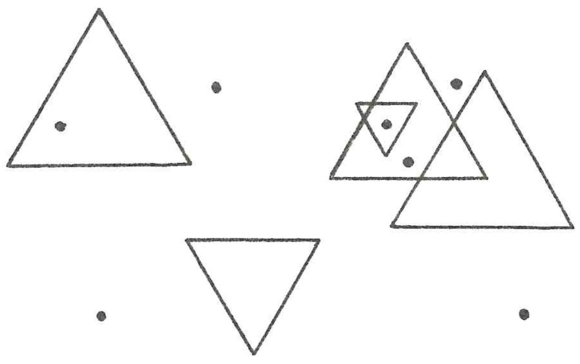


Figure 5-1: A mixed set of c-oriented triangles and points.

To determine the stabbing number we will use the following observation: Without loss of generality assume there is no vertical line in C (we can always rotate the coordinate axes to achieve this). Then every polygon edge can be classified as a top edge (having the polygon below) or a bottom edge. For a given polygon P and a point q let T_P and B_P be the number of top edges and bottom edges below P, respectively. Then P contains q iff $B_P - T_P = 1$. Otherwise q is outside P ($B_P - T_P = 0$).

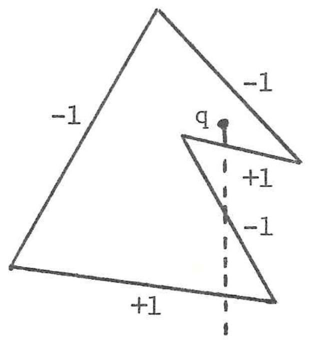


Figure 5-2: Add numbers below q to compute $B_P - T_P$.

Extending this to a set of polygons, let T and B be the total number of top and bottom edges below q, respectively. Then q's stabbing number is exactly B - T.

To solve the batched problem we use this observation in a line-sweep algorithm. At each position, the sweepline intersects the set of all polygon edges in some points $y_0,\ldots,y_m$. These are stored in a binary search tree where each node is augmented by an integer field <u>below(p)</u>. For a leaf p

$$\text{below}(p) = \begin{cases} +1 & \text{if p represents a bottom edge} \\ -1 & \text{if p represents a top edge.} \end{cases}$$

For an inner node p, below(p) is the sum of the below-fields of its sons. Whenever the sweepline encounters a point $q = (x_0, y_0)$, the tree is searched with argument y_0. On the way down the tree the below-fields of all nodes which are left sons of nodes on the search path are summed. In this way the total difference of bottom edges and top edges below q, namely B - T is computed. This is q's stabbing number.

However, there is one drawback of this approach: unlike the orthogonal case the points on the sweepline are not fixed when the sweepline is moved, but also move relatively to the sweepline and to each other, possibly changing their order. But to maintain the search tree it is necessary to maintain the total order of the line segments intersecting the sweepline. More specifically it means that we have to update the structure at each pairwise intersection of line segments. From [BeO] we know that this immediately increases the complexity of the algorithm to $O(n \log n + k \log n)$ where k is the number of intersections. Observe that k can be $\Theta(n^2)$.

At this point our restriction to c-oriented polygons becomes crucial. It enables us to split the set of all polygon edges into c subsets, one for each direction in C. For each subset we maintain a separate searchtree. Within each tree points still move relatively to the sweepline, but not relatively to each other. Now, of course, each node stores a function y(x) rather than a fixed y-value. But for a given query point q it is no problem to evaluate these functions when searching the tree in order to decide whether to go left or right.

We summarize this discussion by giving a description of:

Algorithm STAB

Input: A mixed set of c-oriented polygons and points.

Output: A sequence of pairs (p_i, n_i) where p_i is a point and n_i its stabbing number.

Step 1: Represent all polygon edges by their left and right endpoint. Sort these endpoints and the original points in the set by x-coordinate into a list X-LIST.

Step 2: Scan X-LIST. Encountering

a) a left endpoint of an edge

insert the edge into the searchtree corresponding to its direction, updating below-fields.

b) a right endpoint of an edge

delete the edge from the corresponding tree, updating below-fields.

c) a point q

search all trees with argument q and sum the results from each tree. Report the sum as q's stabbing number.

end of algorithm STAB.

Step 1 takes O(n log n) time. Each of the operations in step 2 takes only O(log n) time, using any balanced tree scheme. In case c) this is true because there is only a constant number of trees searched. Therefore the total time for step 2 as well as the time for the whole algorithm is O(n log n). As a result we obtain:

Theorem 5.1: Let S be a set of c-oriented polygons and points where the total number of polygon edges and points is n. Then there exists an algorithm which solves the batched stabbing number problem for S in O(n log n) time and O(n) space.

5.1.2. *The query problem.*

Now we consider the related problem:

> Given a set of c-oriented polygons, devise a data structure such that stabbing number queries (i.e. 'Given a query point q, how many polygons enclose it?') can be answered efficiently.

Often we can convert a solution for the batched problem which uses a line-sweep into a solution for the query problem. In principle, for a query point q = (x_0,y_0) we only have to look at the data structure employed in the line-sweep algorithm as it was at position x_0. We trivially keep a copy of this data structure after each update and answer a query by searching for the right structure and then using it for answering the query. Unfortunately this structure uses in most cases a prohibitive amount of storage, in our case $O(n^2)$.

In this situation the segment tree, a data structure introduced by Bentley [Be2] which we used already in Section 3, essentially helps to reduce storage costs. The idea is to recover the situation at time x_0 by combining the answers for a collection of data structures, of which some can be used for many different points in time. Which data structures we have to choose is controlled by the segment tree.

Recall that in a segment tree any interval i represented defines a collection of nodes CN(i) (the 'covered nodes') which are 'marked' in some way to represent i (for a definition of CN(i) see Section 3.1.2). In Section 3 we described counting segment trees for which the marking is done by increasing a counter associated to each node. In a segment tree, the marking is done by entering the interval's name into a name list associated to each node.

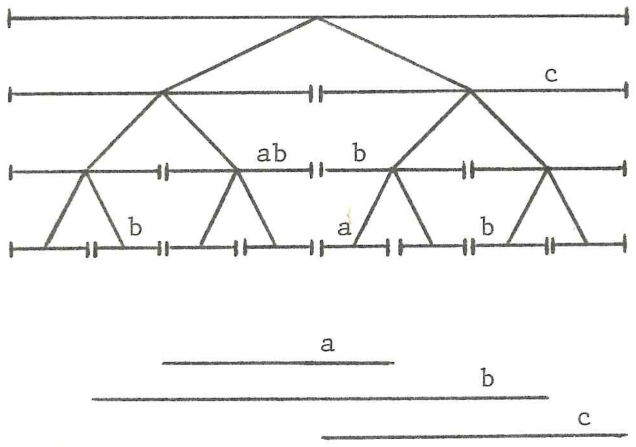


Figure 5-3: A segment tree, storing intervals a, b and c.

Furthermore a set of horizontal line segments in 2-space can be represented by a segment-range tree, see [SiW], [EdM]. The first level of this structure is a segment tree, where each node has an associated point set. A horizontal line segment (I_x, y_0) is stored by entering point y_0 into the point sets of all nodes in the set $CN(I_x)$. Now for a given node p its point set is stored as a binary search tree (called range tree). We may represent this structure as shown in Fig. 5-4.

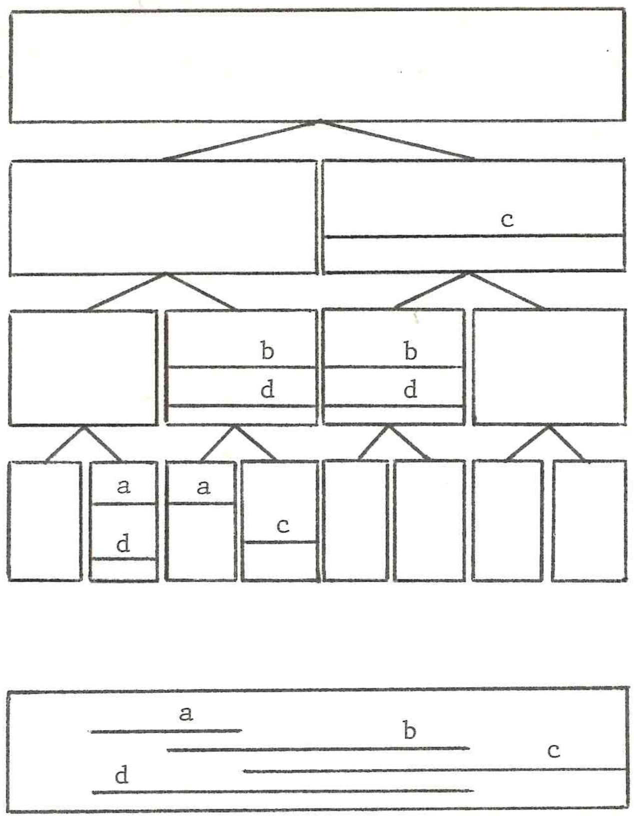


Figure 5-4: A segment-range tree, storing line segments a, b, c and d.

Now we introduce slanted segment-range trees observing that it does not cause any problems to store slanted line segments instead of horizontal ones, as long as they are parallel (and not vertical). The line segment pieces within a single node of the segment tree can be represented as shown in Fig. 5-5.

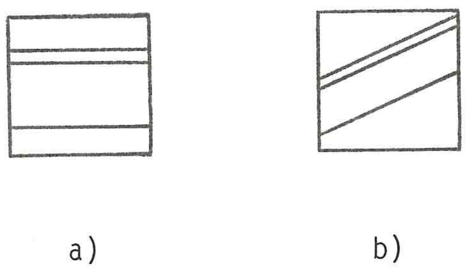


Figure 5-5: Line segment pieces stored in a node of

a) a segment-range tree

b) a slanted segment-range tree.

Of course we have to store the line segment pieces belonging to a segment tree node as a $y(x)$-function instead of a simple y-value. For this we use exactly the data structure employed in the line-sweep algorithm in Section 5.1.1. This justifies our remarks that the segment tree is here just used to control a number of these structures.

The solution for the query problem is now straightforward. From the set of all polygon edges we build a collection of slanted segment-range trees, one for each direction in C. Every polygon edge is stored in the tree corresponding to its direction. Since the structure is very similar to normal segment-range trees it is known that the total preprocessing time is $O(n \log n)$ and that $O(n \log n)$ space is sufficient, [SiW].

A query with point $q = (x_0, y_0)$ is answered by performing a search on each of the c slanted segment-range trees. In each tree the segment tree level is searched with argument x_0, determining $O(\log n)$ nodes in $O(\log n)$ time. For every node the associated tree (the structure described in Section 5.1.1) is searched and the balance $B - T$ of the line segments below computed in $O(\log n)$ time. These balance-values are added for all the $O(\log n)$ level-2 trees and then for all directions. The result is q's stabbing number. Its determination takes $O(\log^2 n)$ time for each slanted segment-range tree. This also determines the total query time since there are only a constant number for these trees.

There exist similar techniques for segment-segment trees [Va], segment-range trees [VaW] and range-range trees [Wi1] to interlink the leaves of the level-2 trees. In all these cases the query time is improved from $O(\log^2 n)$ to $O(\log n)$. It can be shown that these techniques can be adapted for the stabbing number problem (instead of of the stabbing problem) and for slanted segment-range trees yielding $O(\log n)$ search time. Preprocessing and space costs remain the same. It follows:

Theorem 5.2: Stabbing number queries can be answered in $O(\log n)$ time for a set of c-oriented polygons with n edges, using $O(n \log n)$ space and preprocessing time.

5.2. *Polygonal intersection searching*.

The polygonal intersection searching problem, as defined by Edelsbrunner, Kirkpatrick and Maurer [EdKM], is:

> Given a set of polygons P and a query polygon q, determine all polygons in P which intersect q.

This general case has so far been studied only in [EdKM]. In contrast, the special case of iso-oriented rectangles has been examined in quite a number of papers. For a summary see [EdM].From the study of the orthogonal case it is known that the problem can be solved by reducing it to three subproblems.

5.2.1. *The subproblems*.

There are three possible ways how a polygon $p \in P$ may intersect the query polygon q:

(1) q encloses p entirely.
(2) q is enclosed by p.
(3) q and p have an edge intersection.

Clearly those cases are not disjoint. However, if any $p \in P$ intersects q at all, then one of the criteria (1) - (3) is true. A polygonal intersection query may hence be answered in three steps:

1. Represent P by a set of points: for each polygon p in P choose an arbitrary point r_p which is inside p. Let R_p be the resulting point set. Find all points of R_p inside q. We call this a <u>polygonal range query</u>.

2. Represent q by an internal point r_q. Find all polygons in P enclosing r_q. Let us call this a <u>polygonal point enclosure query</u> (identical to the stabbing set problem of the last section; for rectangles also called inverse range searching).

3. Represent P by the set of all edges of polygons in P, E_p. Form the set E_q of all edges composing q. For each edge e in E_q perform a <u>line segment intersection query</u> on E_p.

The mentioned problems are all very basic for the study of non-orthogonal objects. Efficient solutions would have many applications in such areas as computer graphics, querying databases, especially geographical or architectural databases.

Let us review the solutions obtained so far in the general and the orthogonal case. Edelsbrunner, Kirkpatrick and Maurer [EdKM] solve all three subproblems by means of geometric transforms. Let P contain n simple polygons of which t intersect q. The polygons are assumed to be simple and bounded in size (the number of edges) by some constant. Then [EdKM] solve the polygonal intersection searching problem in optimal $O(\log n + t)$ time. However, the data structures employed require extreme amounts of space, namely $O(n^5)$ (see [Ed4, Section 4.3] for the space bound).

In contrast, restricting to rectangles, the intersection searching problem can be solved in $O(\log n + t)$ time and $O(n \log^2 n)$ space. This result is obtained by combining the solutions of the corresponding subproblems (references are given in the following three sections).

Observe again the complexity gap between the orthogonal and the general case which is manifested this time in the space-requirements. We now study the polygonal intersection searching problem for c-oriented polygons. To our knowledge so far only Edelsbrunner [Ed4, Section 4.1.2] considered similar problems. He reduces intersection searching for c-oriented polygons (which have to be convex and bounded in size by $2 \cdot c$ in contrast to our definition) to the corresponding problems for c-dimensional (hyper-)rectangles. This allows to solve the intersection searching problem in $O(\log^{(c-1)} n + t)$ time and $O(n \log^c n)$ space (see [EdM]). In the sequel we describe a much more efficient solution and thereby solve an open problem of Edelbrunner's thesis [Ed4, Section 4.4, problem (2)].

In Sections 5.2.2 through 5.2.4 we give solutions for the c-oriented versions of the three subproblems. Combining them we obtain:

<u>Theorem 5.3</u>: Given C, a set of n C-oriented polygons P (each of which must be simple and bounded in size by some constant) and a C-oriented query polygon q, the polygonal intersection searching problem can be solved in $O(\log^2 n + t)$ time, using $O(n \log^2 n)$ space and preprocessing, where t is the number of answers.

<u>Proof</u>: In the following three sections we prove that each of the subproblems can be solved within the time and space bounds of the theorem (see Theorems 5.4 - 5.6). There remains the problem of multiple reporting since pairs of

intersecting polygons found via point enclosure may also have edge intersections and for any two intersecting polygons several edge intersections may occur. However, since the polygons are bounded in the number of edges, each intersecting pair may be found only a constant number of times. In [Ed4, Section 4.3] it is described how to report each intersecting pair only once without increasing query time or space bound by means of a bit vector. <>

5.2.2. *Line segment intersection searching.*

The (c-oriented) line segment intersection searching problem is:

> Given C, a set of C-oriented line segments L and a C-oriented query line segment l_q, find all line segments in L which intersect l_q.

For arbitrarily oriented line segments it is possible to answer this query in $O(\log n + t)$ time and $O(n^5)$ space, where n is the cardinality of L and t the number of answers [EdKM]. For orthogonal, that is horizontal and vertical, line segments a query can be answered in $O(\log^2 n + t)$ time and $O(n \log n)$ space by means of a segment-range tree (see [EdM]). This result was improved by Vaishnavi and Wood [VaW] to a query time of $O(\log n + t)$ without increasing the space bound.

In Section 5.1.2 we used already slanted segment-range trees to solve the stabbing number query problem. The range trees used were augmented by below-fields. Now the (c-oriented) line segment intersection problem can be solved in a rather trivial manner using a collection of segment-range trees. In this case the range trees are normal binary search trees where for convenience the leaves are arranged as a linked list.

Preprocess L in the following way: For each pair of directions $a, b \in C$, $a \neq b$, construct a slanted segment-range tree storing all a-directed line segments to be searched with a b-directed line segment. This means that the coordinate axis of the segment tree level is the direction perpendicular to b which we call $\bar{b}$. In the $(\bar{b}, b)$-coordinate system a b-directed line segment is given by a tripel $(\bar{b}_0, b_1, b_2)$ corresponding to the description (x_0, y_1, y_2) of a vertical line segment in the (x, y)-system. An a-directed line segment is represented by a $\bar{b}$-interval $[\bar{b}_1, \bar{b}_2]$ and a function $b(\bar{b})$. Fig. 5-6 shows a node of a $(\bar{b}, b)$-segment-range tree storing a-directed line segment pieces and a b-directed query line segment $l_q = (\bar{b}_0, b_1, b_2)$.

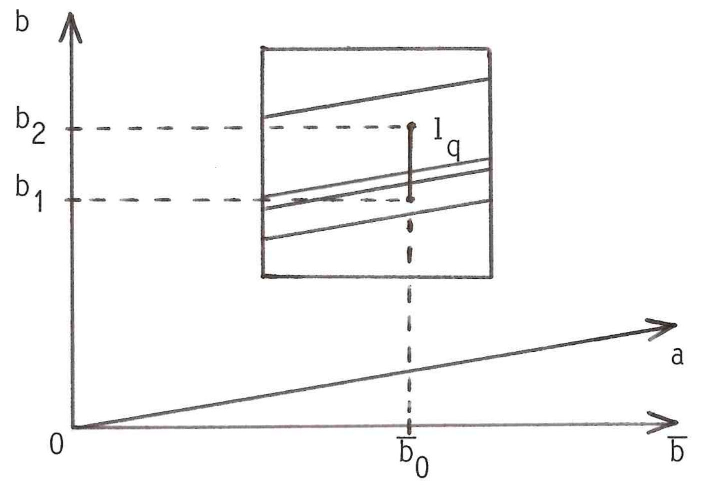


Figure 5-6

Now it is obvious how an a-directed line segment $(\bar{b}_1, \bar{b}_2, b(\bar{b}))$ is stored in a $(\bar{b}, b)$-segment-range tree for direction a: The interval $[\bar{b}_1, \bar{b}_2]$ uniquely identifies $O(\log n)$ nodes of the segment tree level, namely the nodes in $CN([\bar{b}_1, \bar{b}_2])$. Each of those nodes has an associated range tree storing the function $b(\bar{b})$ in one of its leaves (this is possible since the range tree contains only line segments with the same slope).

For $|C| = c$, the complete collection consists of $c \cdot (c - 1)$ trees. Each line segment appears in (c-1) different trees and takes $O(\log n)$ space per tree. Hence the total space required is $O(c \cdot n \log n)$.

A query with a b-directed line segment $l_q = (\bar{b}_0, b_1, b_2)$ is answered as follows: In each of the (c-1) $(\bar{b}, b)$-segment-range trees the 'stabbing coordinate' $\bar{b}_0$ identifies $O(\log n)$ nodes with their respective range trees. In each range tree in $O(\log n)$ time a leaf is found representing the first line segment piece above point $(\bar{b}_0, b_1)$. The linked list of leaves is then scanned, reporting all encountered line segments as intersecting l_q, until the 'top point' $(\bar{b}_0, b_2)$ is reached. Clearly the whole process takes $O(c \log^2 n + t)$ time in the worst case.

The technique of Vaishnavi and Wood [VaW] to improve the query time of segment-range trees can here be applied as well. The modified structure also requires $O(n \log n)$ space and can be built in $O(n \log n)$ time. Hence we obtain:

Theorem 5.4: For a set of n C-oriented line segments and a C-oriented query line segment the line segment intersection searching problem can be solved in $O(\log n + t)$ time, using $O(n \log n)$ space and preprocessing.

5.2.3. *Polygonal range searching.*

The (c-oriented) polygonal range searching problem is:

> Given C, a set of points in 2-space P and a C-oriented query polygon q, find all points of P inside q.

There exist two solutions for the general problem (with an arbitrarily oriented query polygon). The query can either be answered in optimal $O(\log n + t)$ time, using $O(n^5)$ space ([EdKM], [Ed4]) or in $O(n^{0.77} + t)$ time and $O(n)$ space. The latter solution is due to Willard [Wi2] who studies the problem in the context of information retrieval in databases, pointing out that certain types of queries may be interpreted as regions of 2-space bounded by straight lines.

The two mentioned solutions demonstrate quite well that the problem is indeed hard. Only when restricting to the orthogonal case efficient solutions are known. For a query rectangle it is possible to report the enclosed points in $O(\log n + t)$ time, using $O(n \log n)$ space, see Willard [Wi1].

Let us again see what kind of solution we obtain for the 'in between' case of c-oriented polygons. Our task is to find all points from P inside q. q is a simple polygon with at most k edges. First we decompose q into some standard form of regions which happen to be trapeziums:

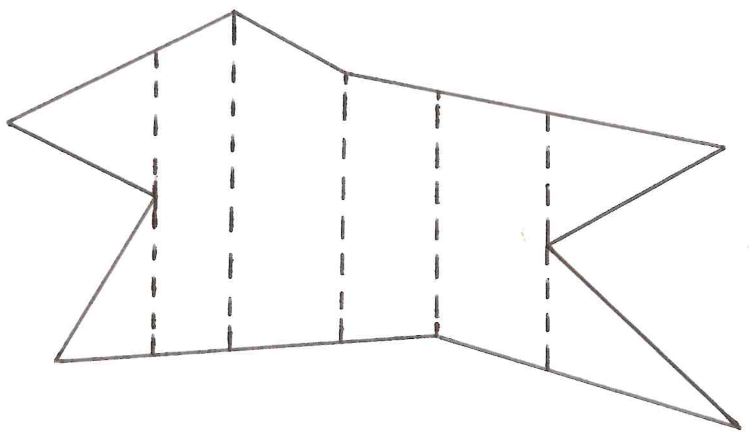


Figure 5-7

The 'paper-and-pencil' method to set up the subdivision is the following: From each vertex p of q draw the vertical lines which project p onto the edges immediately above and below p iff the respective projection line is inside q. Some of the resulting trapeziums may be triangles, that is, trapeziums with a vertical edge of length zero.

It can be shown that for a polygon with k edges the resulting subdivision consists of at most (k-1) trapeziums (of precisely (k-1) trapeziums if no two vertices have identical x-coordinates). The subdivision can be set up in O(k log k) time and O(k) space in the worst case. If no custom-tailored algorithm for trapeziums can be found (which should not be too difficult) use the algorithm of Garey, Johnson, Preparata and Tarjan [GaJPT] to triangulate the polygon and then split each resulting triangle into two (pseudo-) trapeziums. So for our purposes the subdivision can be set up in constant time and comprises only a constant number of regions.

The remaining task is to report all points from a given set inside a query trapezium t_q. t_q has two vertical edges, an a-directed bottom edge and a b-directed top edge, for some directions a, b $\in C$. If we introduce again coordinate axes $\bar{a}$ and $\bar{b}$ perpendicular to a and b, respectively, then t_q can be represented by a quadrupel $t_q = (x_1, x_2, \bar{a}_0, \bar{b}_0)$ (see Fig. 5-8).

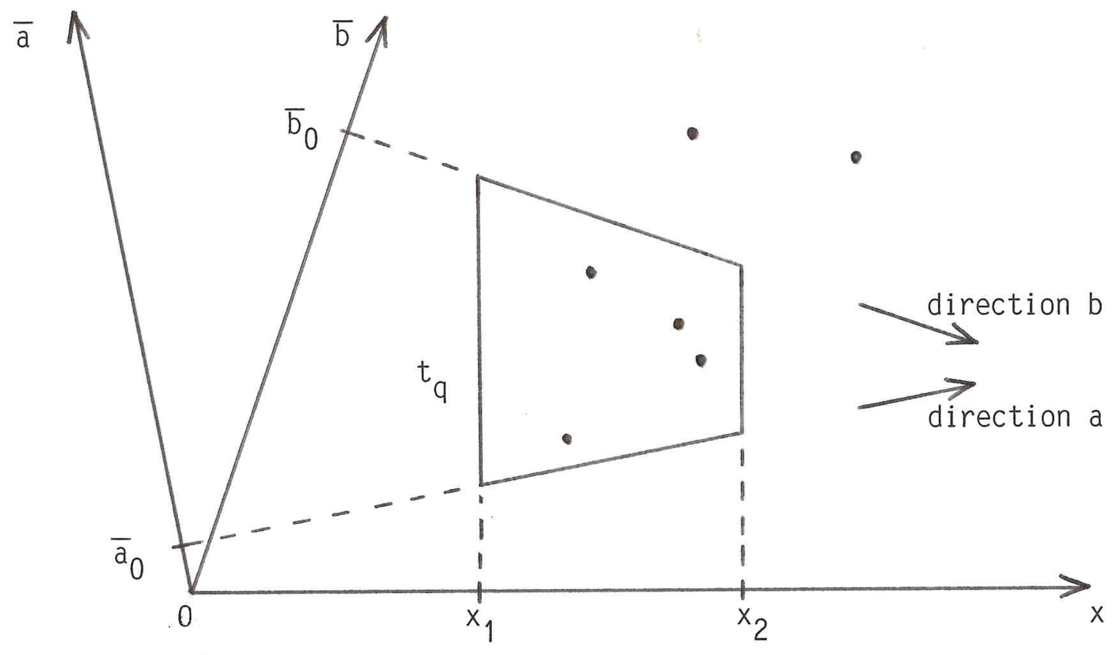


Figure 5-8: A query trapezium t_q in the $(x, \bar{a}, \bar{b})$-coordinate system.

Let us for a moment think about the representation of a point in 2-space. Usually it is given as a pair (x_0, y_0) based on the (x, y)-coordinate system. Clearly, once the origin is given, we may define a new coordinate system by giving a collection of, say, d different arbitrarily chosen directions and describe a point by the d-tuple of projections onto the new coordinate axes.

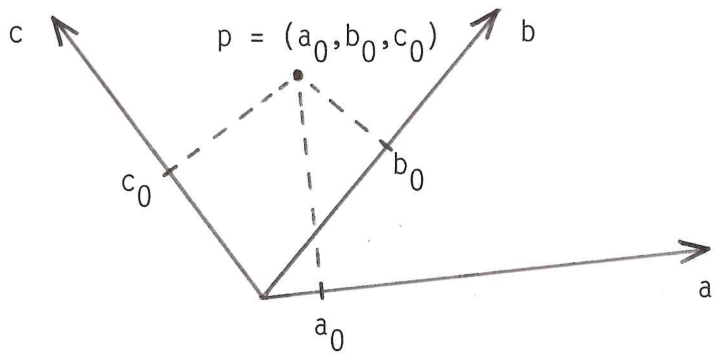


Figure 5-9

Hence any point $p \in P$ may be represented with respect to the $(x, \overline{a}, \overline{b})$-coordinate system by a triple $(x', \overline{a}', \overline{b}')$. Then

$$p \text{ inside } t_q \Longleftrightarrow x_1 \leq x' \leq x_2 \wedge \overline{a}_0 \leq \overline{a}' \wedge \overline{b}' \leq \overline{b}_0 \qquad (*)$$

if we make the convention that the $\overline{a}$ and $\overline{b}$ coordinate axes point upwards from the origin (that is e.g. $\overline{a}$-values increase with increasing y-coordinate).

Now we are able to retrieve all points from P fulfilling condition (*) efficiently if P is organised as a three-dimensional range tree (due to Bentley [Be3], see also Lueker [Lu]) whose three dimension levels are sorted by x-, $\overline{a}$- and $\overline{b}$-coordinate, respectively. Recall the structure of this tree: The top level is a binary searchtree representing the x-coordinates of all points in P. For each node q of this tree let SUB(q) denote the set of all points from P whose x-coordinate lies in q's subtree. Now q contains an additional pointer AUX(q) to a two-dimensional range tree ordered by $\overline{a}$- and $\overline{b}$-coordinate representing SUB(q). The top level of this tree is again a binary searchtree ordered by $\overline{a}$-coordinate with an additional field AUX(r) representing SUB(r) for each node r as a one-dimensional structure sorted by $\overline{b}$-coordinate. In our case it is sufficient to use two-way linked lists for the $\overline{b}$-level. Each list element contains a point from P.

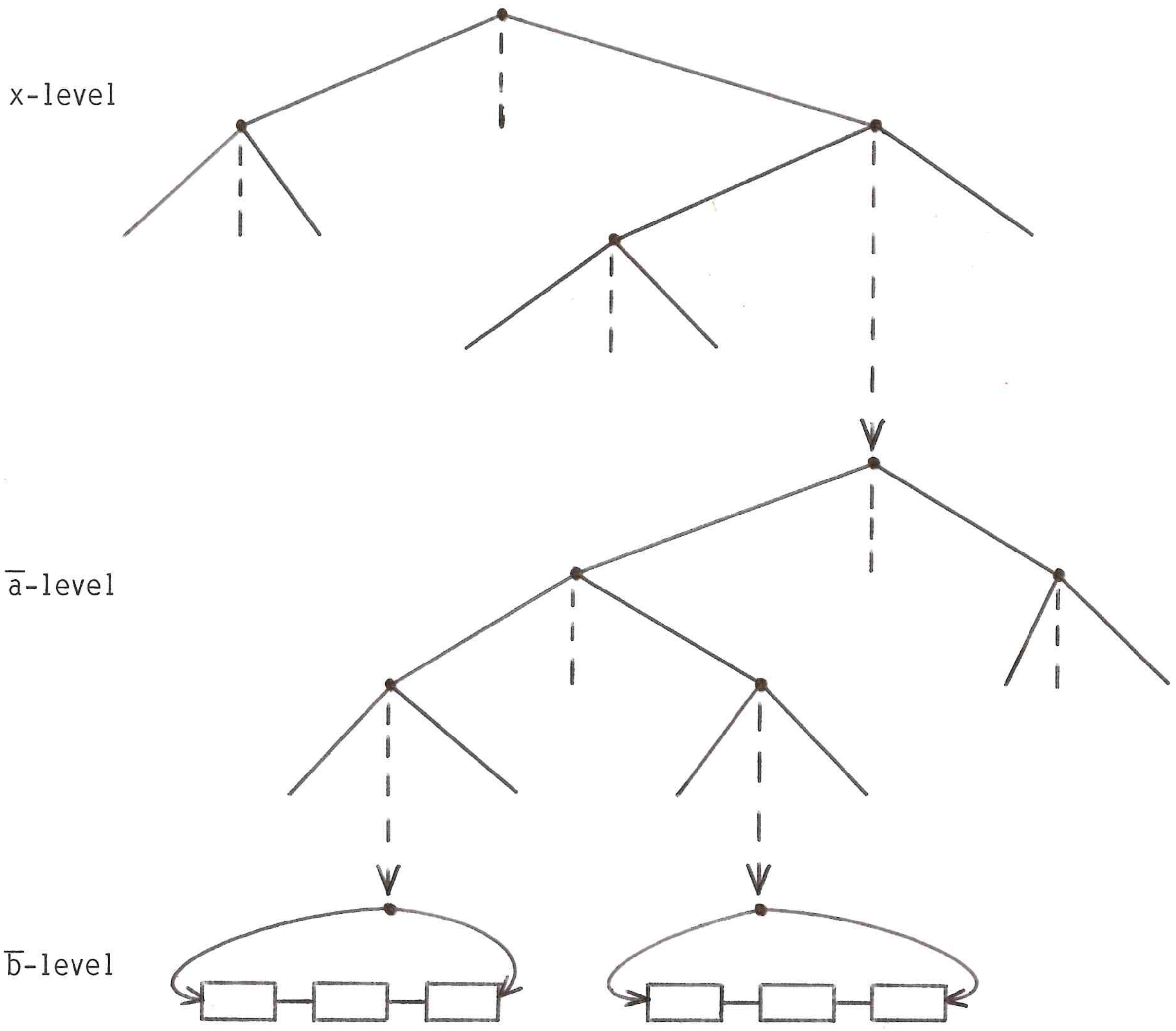


Figure 5-10: A slanted three-dimensional range tree.

A three-dimensional range tree for a set of n points requires $O(n \log^2 n)$ space and preprocessing time (see [Be3]). A range query with a 3-range $(x_1, x_2, y_1, y_2, z_1, z_2)$ can be answered in $O(\log^3 n + t)$ time in the worst case where t is the number of answers. In our special situation we can do better, because querying with $t_q = (x_1, x_2, \overline{a}_0, \overline{b}_0)$ we do not search with intervals on the $\overline{a}$- and $\overline{b}$-level but rather with half-spaces.

The query with t_q is answered in the following way: On the x-level we find the $O(\log n)$ roots of trees covered by $[x_1, x_2]$ (the nodes in $CN([x_1, x_2])$ in analogy to segment trees, see Section 3.1.2) in $O(\log n)$ time. For each node p in $CN([x_1, x_2])$ we search the tree AUX(p) with argument $\overline{a}_0$ and determine the $O(\log n)$ nodes in $CN([\overline{a}_0, \infty])$ in the same time. So far we have found the collection of all lists of the $\overline{b}$-level which may contain answers to the query, in $O(\log^2 n)$ time. Now we scan each list from the left end, reporting all encountered points as answers to the query, until value $\overline{b}_0$ is reached. The time for this is linear in the number of reported points, t. Hence answering a trapezium-query takes $O(\log^2 n + t)$ time in the worst case.

Note that the same tree may be used to answer queries with trapeziums having an a-directed top and a b-directed bottom edge (scanning lists from the right end etc.)

We summarize our solution of the c-oriented polygonal range searching problem in:

Theorem 5.5: Given C, a set of n points P in 2-space and a simple C-oriented polygon q with at most k edges. Then the t points from P inside q can be reported in $O(\log^2 n + t)$ time, with $O(n \log^2 n)$ space and preprocessing.

Proof: For $|C| = c$, preprocess P into the c^2 three-dimensional range trees corresponding to all pairs of directions $a, b \in C$. Given q, divide it in constant time into trapeziums. With each trapezium query the appropriate tree within the time bound of the theorem. <>

5.2.4. *Polygonal point enclosure searching.*

The (c-oriented) polygonal point enclosure searching problem is:

> Given C, a set of C-oriented polygons P and a query point r_q, find all polygons from P which enclose r_q.

The [EdKM]-solution of the general problem allows to find the answer in $O(\log n + t)$ time, but requires $O(n^3)$ space. In contrast, the rectangular version of this problem can be solved in $O(n \log^2 n)$ space retaining the optimal time bound [Va].

The point enclosure searching problem is dual to the range searching problem of the last section. Only the roles of query object and queried set are exchanged. This observation leads to the idea of 'turning around' the solution of the last section which we describe now.

Given the set of C-oriented polygons P we first split each polygon into the set of composing trapeziums (see Section 5.2.3) and consider from now on the set of all trapeziums T. Each trapezium in T is characterized by the two directions of its bottom and top edge. This induces a partition of T into c^2 (for $c = |C|$) disjoint subsets, one subset for each pair of directions in C. Of course some subsets may be empty. Let $T_{a,b}$ denote the subset corresponding to directions a,b.

The representation $(x_1, x_2, \overline{a}_0, \overline{b}_0)$ of a trapezium may be interpreted as a 3-range $([x_1, x_2], [\overline{a}_0, \infty], [-\infty, \overline{b}_0])$ in the $(x, \overline{a}, \overline{b})$-coordinate system. In fact we used this interpretation when querying the three-dimensional slanted range tree. For the present problem we therefore have to look for a data structure able to accomodate 3-ranges and to retrieve them by stabbing (or point enclosure) queries. Such a data structure is the three-dimensional segment tree (described for instance in [Ed3]). It allows to store n 3-ranges in $O(n \log^3 n)$ space and to answer a stabbing query in $O(\log^3 n + t)$ time where t is the number of answers. For all pairs a, b $\in C$ we now represent $T_{a,b}$ by a slanted three-dimensional segment tree. Again we are able to gain some efficiency due to the fact that two of the intervals composing a 3-range are halfspaces in our case. This makes it possible to replace the bottom level of the tree, which is normally a segment tree, by a simple linked list.

A slanted three-dimensional segment tree for directions a, b then looks as follows: The top level is a segment tree (see Fig. 5-3) for x-intervals. Each of its nodes contains a pointer to a segment tree storing $\bar{a}$-intervals. Each node of this (second level) segment tree again contains a pointer to some data structure of the third level which represents $\bar{b}$-intervals. Since any $\bar{b}$-interval occurring is either a left halfspace $[-\infty, \bar{b}_0]$ or a right halfspace $[\bar{b}_0, \infty]$, it is sufficient to maintain two $\bar{b}$-ordered linked lists L and R. A list item of L, for instance, is a pair $(\bar{b}', P')$ where P' is a list of polygon names (any polygon in P' contains a trapezium with $\bar{b}$-interval $[-\infty, \bar{b}']$). Fig. 5-11 illustrates the structure.

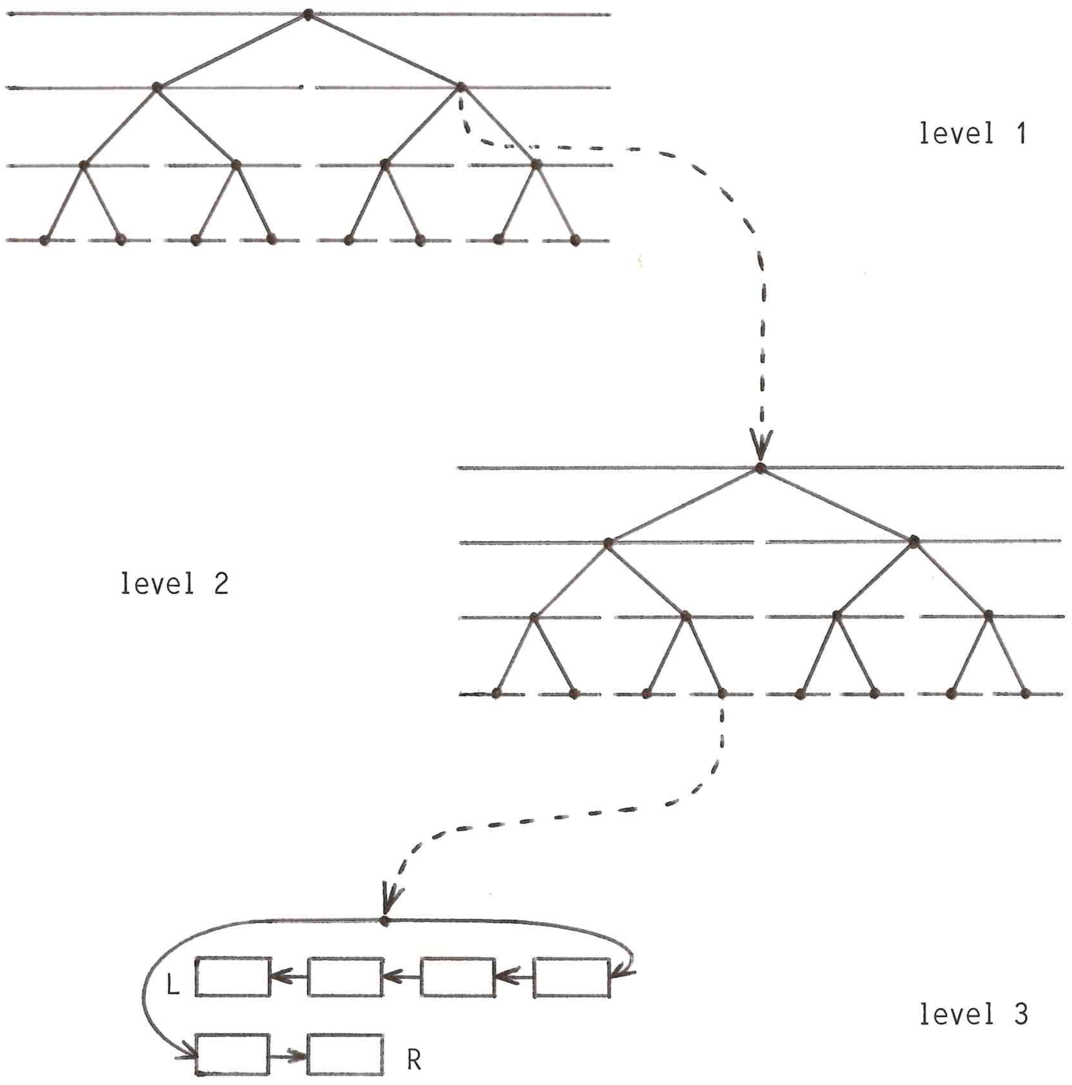


Figure 5-11: A slanted three-dimensional segment tree.

A trapezium $t = (x_1, x_2, \overline{a}_0, \overline{b}_0)$ coming from a polygon p is represented in this structure as follows: On level 1, the interval $[x_1, x_2]$ determines a collection of $O(\log n)$ nodes, the nodes in $CN([x_1, x_2])$. In each associated level-2 segment tree the interval $[\overline{a}_0, \infty]$ again selects $O(\log n)$ nodes. For each of those nodes there is an entry $(\overline{b}_0, P_0)$ in its L-list and P_0 contains the name p. Clearly the representation of a single trapezium requires $O(\log^2 n)$ space in the worst case. Since there are n polygons and each of them consists of a constant number of trapeziums, the total space required is $O(n \log^2 n)$. Choosing a suitable method for preprocessing (along the lines of Six and Wood [SiW, Section V]) the structure can be built in $O(n \log^2 n)$ time. The crucial idea is to first sort all trapeziums by $\overline{b}$-coordinate and then to build the segment tree structure.

So far we argued as if all trapeziums were stored in one single three-dimensional segment tree. In fact we use (at most) c^2 different trees. It is easy to see, however, that splitting into different structures does not increase the space or preprocessing time bound but on the contrary, improves it to $O(c^2 (n/c^2) \log (n/c^2))$.

How do we answer a point enclosure query with $r_q = (x_0, y_0)$? We have to query each of the c^2 structures. For each tree r_q has to be transformed into the appropriate triple of coordinates $(x_0, \overline{a}_0, \overline{b}_0)$. On each tree x_0 then stabs the intervals of $O(\log n)$ nodes of level 1. In each associated level-2 tree $\overline{a}_0$ again stabs $O(\log n)$ nodes. For each of them the L-list and R-list are scanned from the right and left end, respectively. For each encountered list item the names of its polygon list are reported as answers to the query. Finding the lists has taken $O(\log^2 n)$ time, scanning the lists $O(t)$ time for t reported polygons. Thus we obtain:

Theorem 5.6: For a set of n C-oriented simple polygons P (bounded in size by some constant) the polygonal point enclosure searching problem can be solved in $O(\log^2 n + t)$ time and $O(n \log^2 n)$ space and preprocessing, where t is the number of answers.

This completes the proof of Theorem 5.3.

5.3. *Computing the contour of a set of c-oriented polygons.*

In this final section we return to the problem studied primarily in this thesis, the contour problem. One of the motives in studying c-oriented polygons was to find a more efficient contour algorithm for this restricted case than seems possible for arbitrary polygons. As we will see, such an algorithm does exist; however, we do not achieve the efficiency possible for rectangles.

A solution of the general problem, that is computing the contour of a set of arbitrarily oriented polygons, was given by Ottmann, Widmayer and Wood [OWW]. In fact, their algorithm computes the set of contour-pieces; from it in a trivial sorting step the contour can be computed as a collection of contour-cycles (as in Sections 3 and 4) in $O(p \log p)$ time and $O(p)$ space, where p is the number of edges in the contour. In this section we also understand by 'computing the contour' just to report the set of contour-pieces.

The contour algorithm of [OWW] is a modification of the line-sweep algorithm of Bentley and Ottmann [BeO] which finds all intersections among a set of arbitrarily oriented line segments. Essentially in the contour algorithm the sweepline stops at each edge intersection point to update some structure representing the order of the line segments intersecting the sweepline as well as the 'coverage' of the regions between them. Stopping at each intersection point immediately implies a time bound of $O(n \log n + k \log n)$ (the time bound of [BeO]) for the contour algorithm (for n polygons of bounded size with k edge intersections).

It is important to realize that this time bound is indeed much worse than the one for rectangles ($O(n \log n + p)$ time) which does not depend on the number of intersections, but on the actual size p of the contour. An extreme example may illustrate this. Imagine a 'complicated' set of polygons with $\Theta(n^2)$ intersections enclosed by a single large polygon. Obviously the contour of the combined set is identical to the enclosing polygon. To compute this contour the polygonal contour algorithm requires $\Theta(n^2 \log n)$ operations. The algorithm for the analogous rectangular case uses only $\Theta(n \log n)$ time.

How difficult is the problem for c-oriented polygons? First, observe that the time bound for computing all edge intersections can trivially be reduced by a factor of log n in this case. Namely, divide the set of line segments into c subsets, one for each direction in C. Then compute intersections between all pairs of subsets using the (slightly modified) algorithm of Bentley and Ottmann [BeO] for horizontal and vertical line segments. The time bound is $O(n \log n + k)$.

The reduced complexity of finding intersections permits us also to reduce the time bound for computing the contour. We now give an alternative description of the edge intersection algorithm which is later modified to obtain a contour algorithm.

Let L be a set of n c-oriented line segments. Divide L into subsets of equal direction $L_1, \ldots, L_c$ with associated directions $d_1, \ldots, d_c$. For each set L_i, compute a coordinate transformation on all sets L_j, $j = 1, \ldots, c$, such that in the new coordinate system the line segments of L_i are vertical. Then perform a line-sweep in (the new) horizontal direction. Maintain a binary searchtree T_j for each direction d_j, $j \neq i$. For convenience we use leafsearch trees, that is the leaves are linked into a linear list. The line-sweep then works as follows: Encountering

- the left endpoint of a line segment l of direction d_j, $j \neq i$: insert l into T_j. l is represented by its $y(x)$ function, see Section 5.1.1.
- the right endpoint of a line segment l of direction d_j, $j \neq i$: delete l from T_j.
- a vertical line segment $l_q = (x_0, y_1, y_2)$ (that is a line segment of direction d_i): query all trees T_j, $j \neq i$ with the interval $[y_1, y_2]$ and report the line segments with $y_1 \leq y(x_0) \leq y_2$ as intersecting l_q (in each tree those line segments are represented by a sequence of adjacent leaves).

To analyze this algorithm, observe that the line-sweep stops at $O(n)$ positions. At each position either an insertion or deletion is performed in $O(\log n)$ time or all trees are queried in $O(\log n + t)$ time where t is the number of reported intersections. Hence the whole algorithm requires $O(n \log n + k)$ time. $O(n)$ space is sufficient.

To obtain a contour algorithm our strategy is to compute for each polygon edge those fragments of it which intersect free area. To this end we modify the edge intersection algorithm such that whenever a vertical polygon edge l_q is encountered we compute the parts of it intersecting free area. Let us ignore for a moment that l_q belongs itself to some polygon and consider the problem of determining for an 'independent' vertical line segment which parts of it intersect free area.

When the sweepline is moved through a set of polygons, at each position it is divided by the polygon edges into a sequence of fragment intervals. Each of those fragments is covered by polygons a certain number of times.

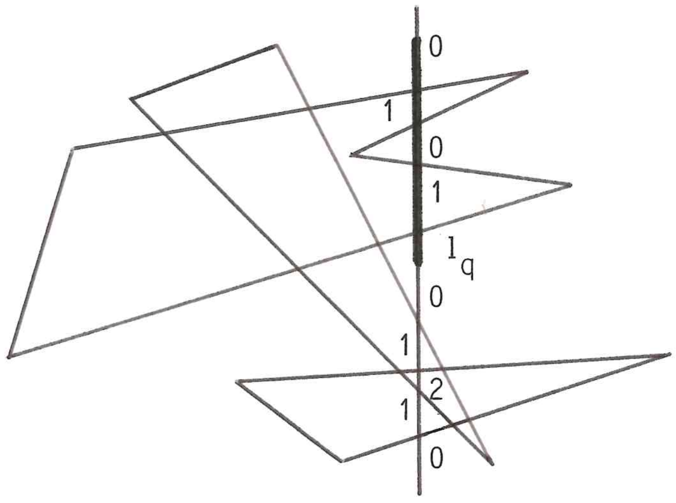


Figure 5-12

If we maintain this sequence of fragment intervals during the line-sweep as a binary searchtree (each leaf representing a fragment interval) then we are able to compute for a given vertical line segment $l_q = (x_0, y_1, y_2)$ the 'free' parts: We search for the fragment containing y_1 in $O(\log n)$ time. Then the leaves of the tree are scanned (suppose we use a leafsearch tree). Whenever the coverage of a fragment is zero, the corresponding part of l_q is reported as being 'free'. - Unfortunately to maintain the sequence of fragment intervals it is necessary to stop the line-sweep at each edge intersection point which is just what we want to avoid in order to improve the efficiency.

However, it is possible to use the same idea if we modify the edge intersection algorithm slightly: We replace the binary searchtrees employed by the trees augmented by below-fields of Section 5.1.1. This enables us to perform stabbing number queries during the line-sweep. The contour algorithm then works as follows: After forming the set of all polygon edges we apply the edge intersection algorithm as described above, except for the case when a vertical edge $l_q = (x_0, y_1, y_2)$ is encountered. In that case we search all trees with l_q's bottom point $\underline{l}_q = (x_0, y_1)$. The result is twofold: First, $\underline{l}_q$'s stabbing number is computed and assigned to a variable s. Second, in each tree's linked list of leaves the position of $\underline{l}_q$ is determined. We then scan those lists in parallel, computing the correct order of the edges intersecting l_q and by this, in fact, the sequence of fragment intervals, until we reach $l_q = (x_0, y_2)$. For each encountered edge we update s which represents the coverage of the current fragment. If a fragment $[y', y'']$ with coverage zero is found then the contour-piece (x_0, y', y'') is reported.

Up to now we ignored the fact that l_q itself belongs to some polygon p_l. Obviously we have to avoid to count p_l as covering l_q. This can be done by choosing the correct order of the three operations of inserting/deleting the neighbouring edges of p_l below and above l_q and querying with l_q. Those operations occur at the same position of the line-sweep and we did not yet specify their order.

The time complexity of this algorithm is clearly the same as that of the edge intersection algorithm: Computing the stabbing number takes no (asymptotic) extra time since each tree had to be searched for the scan's start position anyway. Also scanning the lists of leaves in parallel instead of separately adds only a constant factor of time. Since the space requirements are unchanged, we obtain:

<u>Theorem 5.7</u>: For a set of n c-oriented polygons, each of them simple and bounded in size by some constant, the contour can be computed in $O(n \log n + k)$ time and $O(n)$ space where k is the number of polygonal edge intersections.

6. *CONCLUSIONS*.

In this final section we try to evaluate the work done, discuss its significance and point out applications. We also identify directions of further research as well as interesting open problems.

First, let us summarize what we consider to be the main contributions of this thesis:

(1) The first time-optimal solutions of the contour problem are given. Furthermore variants of the contour problem are studied in considerable depth and efficient solutions are presented.

(2) We introduce the idea of separational representation to support divide-and-conquer. This permits efficient divide-and-conquer solutions for a wide range of problems concerning orthogonal planar objects for which previously only line-sweep seemed to be a suitable method. We prove this by describing time-optimal divide-and-conquer algorithms for five non-trivial problems.

(3) We introduce the notion of c-oriented objects to bridge the complexity gap between problems concerning orthogonal and those for arbitrarily oriented objects. We show that this is possible by giving efficient solutions for a number of problems based on c-oriented objects. In certain application areas (see below) problems involving only a small number of directions occur. Here c-oriented objects might prove to be a quite useful alternative between the orthogonal and the general case.

Let us now review the results obtained in more detail.

Contour problems.

Computing the contour of a set of rectangles is a quite basic problem in planar computational geometry. Its study is also motivated by applications in VLSI-design, geometric data bases etc. Lipski and Preparata [LiP] attacked the problem first, but did not find an optimal solution. They left it as an open problem whether an optimal $O(n \log n + p)$ time solution does exist (for n rectangles with a contour of size p). We answer this question affirmatively, describing two different algorithms achieving the time bound. Our line-sweep algorithm requires $O(n + p)$ space, provided the task is, to report the contour in the form of 'contour-cycles' (as does the algorithm of [LiP]). If only contour-pieces have to be reported then $O(n)$ space is sufficient. It is an open problem whether also the first task can be performed in optimal $O(n)$ space.

The efficiency of the line-sweep algorithm is due to a rather sophisticated data structure, the contracted segment tree (CST), which is a modification of the well-known segment tree. The CST provides a solution for the following one-dimensional searching problem:

> Represent a (semi-) dynamic (see below) set of n intervals I such that insertion and deletion of an interval is possible in $O(\log n)$ time and queries with an interval i for the intersection with the complement of I (the 'free' parts of the line) in $O(\log n + \beta)$ time, where β is the number of disjoint intervals in the answer.

We mention the searching problem because it might have applications in its own right. - The set I is semidynamic because only intervals whose endpoints are in a prespecified set of points are allowed for insertion, deletion and querying. An interesting open problem is whether a fully dynamic solution can be given.

Such a solution might improve the actual execution time of our algorithm though not the asymptotic time bound.

The more general problem behind the mentioned one is to dynamize the segment tree. While for the segment tree with nodes augmented by node lists no very efficient solution seems to exist (see Edelsbrunner [Ed3]), the counting segment tree might permit O(log n) update and query time. For two special applications of the counting segment tree (measure and stabbing number queries) Gonnet, Munro and Wood [GoMW] provide dynamic solutions by means of alternative tree structures. However, dynamizing the counting segment tree itself could make it possible to adapt easily to the fully dynamic case many applications of the counting segment tree already found.

There exist two interesting open problems arising from our work on contours which are strongly interrelated. The first is to give an algorithm computing the tree-dimensional contour of a set of orthogonal bricks in 3-space. The second is to maintain dynamically the two-dimensional contour on insertion and deletion of rectangles. Obviously when using a plane-sweep algorithm the first problem reduces to the second. First studies of the author indicate that it is impossible to maintain the isolated 1-contour in efficient worst-case update time, if deletions are permitted. This is due to the fact that after deleting a rectangle a complex contour of enclosed rectangles may become visible. Hence it is necessary to maintain all i-contours. - Apart from theoretical interest the mentioned problems might have applications in computer graphics and interactive design algorithms.

Another generalization of the contour problem is studied in depth in this thesis, namely the class of problems based on the notions of i-areas, i-measures and i-contours. Many efficient algorithms are described, some of them optimal.

The study illustrates the usefulness of the counting segment tree and shows that also the contracted segment tree is not a single-purpose structure but can be modified in different nontrivial ways to obtain efficient solutions for other interval searching problems. The results of Sections 3.2 show that it is possible to compute the height of a set of rectangles efficiently as well as measure and contour for the top and bottom end of the i-scale. For arbitrary intermediate values of i the computation becomes less efficient.

Again each of the problems discussed is solved by means of a line-sweep algorithm using a data structure which efficiently solves a semidynamic one-dimensional interval searching problem. For all of those problems fully dynamic solutions are not yet known and should be investigated. They would improve the respective line-sweep algorithms.

Since in the last few years many efficient data structures for searching for or with intervals have been developed it would be a quite useful enterprise to survey and classify them. This work would certainly exhibit interesting directions of further research.

Though the study of i-contour and i-measure problems is primarily of theoretical interest, there might exist applications, especially in VLSI-design. Recently problems have been investigated ([La2], [OWW]) which are based on the realistic assumption that not a single set of planar objects is given, but rather a collection of sets corresponding to the different layers of a VLSI cirquit layout. One particular problem motivated from practice is to compute 'boolean mask operations' between those layers. Consider the special case of computing 'A AND B' for two sets of rectangles A and B (that is to compute the contour of the area defined by union(A) $\cap$ union(B)). If each layer is given by a set of non-intersecting rectangles then simple application of algorithm HEIGHT-CONTOUR yields the desired result. Otherwise a possible (though not very efficient) way to compute A AND B is to compute the contours of union(A) and union(B) separately, to retransform the result into a set of disjoint rectangles and then to proceed as before. However, a more promising approach is to devise a custom-tailored adaptation of the contracted segment tree for the boolean mask operation problem. The divide-and-conquer methods of Section 4 might also be tried.

Divide-and-conquer.

Five problems concerning orthogonal planar objects are solved by divide-and-conquer:

(1) The line segment intersection problem.
(2) The point enclosure problem.
(3) The rectangle intersection problem, by combining (1) and (2).
(4) The measure problem.
(5) The contour problem.

All solutions are time-optimal. With exception of the contour algorithm they are also space-optimal. All problems had been solved by use of line-sweep algorithms before (problem (5) in this thesis). These are the first optimal divide-and-conquer solutions. While up to now line-sweep seemed to be the only possible method to solve this kind of problems efficiently, the divide-and-conquer solutions have become possible through the simple, yet powerful idea of separational representation of planar objects.

Since these results concern the methodology of the design of algorithms, they are of wider interest than the particular problems considered. Many natural questions arise from this work, like the following: Which of the two paradigms is more powerful for which kind of problems? Are there problems that can be solved definitely only by line-sweep or only by divide-and-conquer in optimal time or space? Are the two paradigms interchangeable for problems involving orthogonal objects in 2-space, as we conjecture?

Some insights can already be gained from the comparison of the respective line-sweep and divide-and-conquer algorithms for the problems we studied. We observe that in all cases a certain trade-off seems to take place: With divide-and-conquer, more complexity is put into the structure of the algorithm. As a result the data structures become more simple and easier to maintain. With line-sweep, the structure of the algorithm is more simple which has to be paid for by an increased complexity of the supporting data structures.

The crucial step in devising a line-sweep algorithm is to solve a dynamic one-dimensional searching problem. The crucial step in the design of a divide-and-conquer algorithm is to merge two sets of objects, extracting or maintaining the information relevant to the problem. Therefore we may characterize line-sweep as an <u>asymmetric</u> and divide-and-conquer as a <u>symmetric</u> technique: the former compares a single object (the query object) to a set of objects while the latter deals with two approximately equal-sized sets.

More specifically, if we wish to obtain solutions working in $O(n \log n)$ time for a set of n objects (as is the case for all problems we solve in this thesis) then it is necessary for the line-sweep approach to solve the searching problem in $O(\log n)$ update and query time. With divide-and-conquer, it is necessary to perform the merging of the two sets in linear time, that is, a constant amount of time may be consumed for each object.

It would be interesting to know which of those two tasks is in general the more difficult one if such a general statement makes sense at all. Looking at the problems solved so far, it seems that the segment tree used to solve for instance the measure problem was quite an invention while the divide-and-conquer counterpart, a linked list of numbers, is a very simple data structure. Also considerable effort was necessary to develop the contracted segment tree which is certainly a complicated structure. The divide-and-conquer counterpart, a linked list of stripes containing interconnected trees, is somewhat more complex, too, but still seems relatively simple compared to the CST. On the other hand for the line-sweep solution $O(n + p)$ space is sufficient while divide-and-conquer requires $O(n \log n + p)$ space. It is an open problem whether the space bound for divide-and-conquer can be reduced.

To summarize the comparison: divide-and-conquer seems to be a quite suitable alternative to line-sweep for problems based on orthogonal objects in 2-space. In some cases it leads to simpler solutions. As a consequence, the designer of algorithms should try both paradigms when dealing with new problems of this kind.

Concerning further research, a systematic comparison of both paradigms should be undertaken. The following questions are particularly interesting:

- Characterize the properties of a problem which permit or prevent a solution with either technique.
- Let LS and DAC denote the problem classes which are 'solvable' by line-sweep and divide-and-conquer, respectively. Here different definitions of 'solvable' may be considered, like solvable at all or with certain time/space bounds. Then define $LS \cap DAC$. That is, given one solution it is possible to describe the dual one. Develop a method to obtain the other solution.
- Study the relation between the corresponding data structures for problems in $LS \cap DAC$ (for the measure problem they are the segment tree and a

linked list of numbers). Is it possible to define transforms between them?

- Characterize LS - DAC and DAC - LS.
- In case of problems for which both paradigms achieve the same asymptotic time and space bounds, compare the solutions by implementation to gain some knowledge about the constants hidden in the O-notation.

Other open problems concern extensions of the divide-and-conquer technique:

- Generalize the method for higher dimensions ($d \geq 3$).
- Line-sweep algorithms have been used successfully to solve problems involving non-orthogonal objects (for instance the algorithm finding all intersections among a set of arbitrarily oriented line segments, see [BeO]). Can the applicability of divide-and-conquer be extended to this kind of problems?

Last not least, divide-and-conquer should be tried on problems for which time-optimal line-sweep algorithms could not yet be found.

C-oriented objects.

We introduce the notion of c-oriented objects which can be viewed either as a generalization of '2-oriented objects' (that is, orthogonal objects like rectangles) or as a restriction of the general case of arbitrary polygons. The goal is to extend the efficiency obtainable for the orthogonal case to this more general class of objects. For a number of problems we show that this is indeed possible: For two versions of the stabbing number problem time-optimal algorithms are presented. The restricted case of the polygonal intersection searching problem is solved with a time bound of $O(\log^2 n + t)$ and $O(n \log^2 n)$ space and preprocessing (for n polygons and an answer of size t). In contrast, the solution of the general case permits a query time of $O(\log n + t)$, but requires $O(n^5)$ space and preprocessing. Finally for c-oriented polygons the complexity of the contour problem is studied. A solution is given whose time bound is a little better than that of the general case but much worse than the one we obtain for rectangles.

The general approach to solve problems for c-oriented objects is to provide a separate data structure for each occurring direction (or pair of directions). The remaining problems are then how to decompose the given objects (polygons) according to those directions and how to reconstruct the information lost by this (apart from some representational issues). Sections 5.1 and 5.3 give two examples of how to deal with the second problem while in Section 5.2 the first problem is in the foreground.

Algorithms and data structures for c-oriented objects may have applications whenever a real-life problem involves only a small number of possible directions. We mention a few examples:

(1) In VLSI-design, besides orthogonal layouts sometimes '45° artwork' is used. Since only very few directions occur, algorithms designed for c-oriented polygons may be used with advantage instead of those for arbitrary polygons.
(2) It seems that for instance in architectural design very often only few directions occur. There might be similar design areas.
(3) A graphics editor might allow to compose pictures (e.g. figures) by combining certain basic symbols, which consist of line segments with only a few different orientations.

Concerning further work, it seems important to examine possible application areas and to identify the basic geometrical problems occurring. For computer graphics a c-oriented hidden line elimination algorithm might be useful. Furthermore the possibility of approximating arbitrary curves by c-oriented line segments seems worth being investigated. This idea is discussed in [Free].

References.

[AHU] Aho, A.V., J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms. Addison-Wesley, Reading, Mass., 1974.

[Ba] Baird, H.S., Fast algorithms for LSI artwork analysis. J. Design Automation and Fault-Tolerant Computing 2, 179-209, 1978.

[Be1] Bentley, J.L., Multidimensional binary search trees used for associative searching. Communications of the ACM 18, 509-517, 1975.

[Be2] Bentley, J.L., Solutions to Klee's rectangle problems. Carnegie-Mellon University, Department of Computer Science, unpublished manuscript, 1977 (described in [vLW]).

[Be3] Bentley, J.L., Decomposable searching problems. Information Processing Letters 8, 244-251, 1979.

[Be4] Bentley, J.L., Multidimensional divide-and-conquer. Communications of the ACM 23, 214-229, 1980.

[BeO] Bentley, J.L. and Th. Ottmann, Algorithms for reporting and counting geometric intersections. IEEE Transactions on Computers C-28, 643-647, 1979.

[BeS] Bentley, J.L. and M.I. Shamos, Optimal algorithms for structuring geographic data. In: Proceedings, Symposium on Topological Data Structures for Geographic Information Systems, Harvard University, 43-51, 1977.

[BeW] Bentley, J.L. and D. Wood, An optimal worst-case algorithm for reporting intersections of rectangles. IEEE Transactions on Computers C-29, 571-577, 1980.

[C] Comer, D., The ubiquitous B-tree. Computing Surveys 11, 121-137, 1979.

[EaL] Eastman, C.M. and J. Lividini, Spatial search. Carnegie-Mellon University, Institute of Physical Planning, Report 55, 1975.

[Ed1] Edelsbrunner, H., Dynamic rectangle intersection searching. Technical University of Graz, Institut für Informationsverarbeitung, Report F47, 1980.

[Ed2] Edelsbrunner, H., A time- and space-optimal solution for the planar all intersecting rectangles problem. Technical University of Graz, Institut für Informationsverarbeitung, Report F50, 1980.

[Ed3] Edelsbrunner, H., Dynamic data structures for orthogonal intersection queries. Technical University of Graz, Institut für Informationsverarbeitung, Report F59, 1980.

[Ed4] Edelsbrunner, H., Intersection problems in computational geometry. Technical University of Graz, Institut für Informationsverarbeitung, Report F93, Ph.D. Thesis, 1982.

[EdKM] Edelsbrunner, H., D.G. Kirkpatrick and H.A. Maurer, Polygonal intersection searching. Information Processing Letters 14, 74-79, 1982.

[EdLOW] Edelsbrunner, H., J. van Leeuwen, Th. Ottmann and D. Wood, Connected components of orthogonal geometrical objects. McMaster University, Unit for Computer Science, Report 81-CS-04, 1981.

[EdM] Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects. Information Processing Letters 13, 177-181, 1981.

[EdO] Edelsbrunner, H. and M.H. Overmars, On the equivalence of some rectangle problems. Information Processing Letters 14, 124-127, 1982.

[FredW] Fredman, M.L. and B.W. Weide, On the complexity of computing the measure of $\bigcup [a_i, b_i]$, Communications of the ACM 21, 540-544, 1978.

[Free] Freeman, H., Lines, curves, and the characterization of shapes. Information Processing 80, Proceedings of IFIP Congress 80, North-Holland Publishing Company, 629-639, 1980.

[GaJPT] Garey, M.R., D.S. Johnson, F.P. Preparata and R.E. Tarjan, Triangulating a simple polygon. Information Processing Letters 7, 175-179, 1978.

[GoMW] Gonnet, G.H., J.I. Munro and D. Wood, Direct dynamic structures for some line segment problems. McMaster University, Unit for Computer Science, manuscript, 1981.

[Gü1] Güting, R.H., An optimal contour algorithm for iso-oriented rectangles. McMaster University, Unit for Computer Science, Report 82-CS-03, 1982, to appear in Journal of Algorithms.

[Gü2] Güting, R.H., Stabbing c-oriented polygons. Information Processing Letters 16, 35-40, 1983.

[Gü3] Güting, R.H., Optimal divide-and-conquer to compute measure and contour for a set of iso-rectangles. Universität Dortmund, Lehrstuhl Informati VI, Report 141, 1982.

[GüW] Güting, R.H. and D. Wood, Finding rectangle intersections by divide-and-conquer. McMaster University, Unit for Computer Science, Report 82-CS-04, 1982.

[La1] Lauther, U., 4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits. J. Design Automation and Fault-Tolerant Computing 2, 241-247, 1978.

[La2] Lauther, U., An O(n log n) algorithm for boolean mask operations. Proceedings of the 18th Design Automation Conference, Nashville, 555-562, 1981.

[Le] Lee, D.T., Algorithms for rectangle intersection problems. Unpublished manuscript, 1982.

[LiP] Lipski, W. and F.P. Preparata, Finding the contour of a union of iso-oriented rectangles. Journal of Algorithms 1, 235-246, 1980.

[Lu] Lueker, G.S., A data structure for orthogonal range queries. Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, 28-34, 1978.

[Mc] McCreight, E.M., Efficient algorithms for enumerating intersecting intervals and rectangles. XEROX Palo Alto Research Center, Report CSL-80-9, 1980.

[MeC] Mead, C. and L. Conway, Introduction to VLSI-Systems. Addison-Wesley, Reading, Mass., 1980.

[OWW] Ottmann, Th., P. Widmayer and D. Wood, A fast algorithm for boolean mask operations. Universität Karlsruhe, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Report 112, 1982.

[Sh] Shamos, M.I., Computational geometry. Yale University, Ph.D. Thesis, 1978.

[ShH] Shamos, M.I. and D. Hoey, Geometric intersection problems. Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science, 208-215, 1976.

[SiW] Six, H.W. and D. Wood, Counting and reporting intersections of d-ranges. <u>IEEE Transactions on Computers C-31</u>, 181-187, 1982.

[SoW] Soisalon-Soininen, E. and D. Wood, Optimal algorithms to compute the closure of a set of iso-rectangles. University of Helsinki, Department of Computer Science, Report C-1982-31, 1982.

[Va] Vaishnavi, V.K., Computing point enclosures. <u>IEEE Transactions on Computers C-31</u>, 22-29, 1982.

[VaW] Vaishnavi, V.K. and D. Wood, Rectilinear line segment intersection, layered segment trees and dynamization. <u>Journal of Algorithms 2</u>, 160-176, 1981.

[vLW] van Leeuwen, J. and D. Wood, The measure problem for rectangular ranges in d-space. <u>Journal of Algorithms 2</u>, 282-300, 1981.

[Wi1] Willard, D.E., New data structures for orthogonal queries. Harvard University, Aiken Computer Laboratory, Report TR-22-78, 1978.

[Wi2] Willard, D.E., Polygon retrieval. <u>SIAM Journal on Computing 11</u>, 149-165, 1982.

[Wo] Wood, D., personal communication, 1981.