Acta Informatica 21, 271–291 (1984)

Acta Informatica

© Springer-Verlag 1984

Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles*

Ralf Hartmut Güting

Lehrstuhl Informatik VI, Universität Dortmund, D-4600 Dortmund 50 (Fed. Rep.)

Summary. We reconsider two geometrical problems that have been solved previously by line-sweep algorithms: the measure problem and the contour problem. Both problems involve determining some property of the union of a set of rectangles, namely the size and the contour (boundary) of the union. We devise essentially a single time-optimal divide-and-conquer algorithm to solve both problems. This can be seen as a step towards comparing the power of the line-sweep and the divide-and-conquer paradigms. The surprisingly efficient divide-and-conquer algorithm is obtained by using a new technique called "separational representation", which extends the applicability of divide-and-conquer to orthogonal planar objects.

1. Introduction

Over the last few years much attention has been paid to problems based on a set of iso-oriented rectangles. (A set of rectangles is *iso-oriented* if the sides of all rectangles are parallel to the coordinate axes). The problems studied include finding all intersecting pairs in a set of rectangles [5, 7, 13], computing the measure of the union, constructing the contour of the union, finding connected components [8] and computing the closure of a set of rectangles [16].

The interest in these problems is due to the fact that sets of rectangles play an important role in VLSI-design (see for instance [14]) and that they have other applications as well, e.g. in geography [4], in maintaining architectural data bases [6], and in computer graphics.

In almost all cases these problems have been solved by the use of *line-sweep* algorithms. The line-sweep paradigm was apparently introduced into computational geometry by Shamos and Hoey [15] and afterwards widely used to solve problems based on a set of objects in the plane. The idea is to move a

* This work was partially supported by the DAAD (Deutscher Akademischer Austauschdienst) and by the DFG (Deutsche Forschungsgemeinschaft)

(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 projection onto the sweepline. The problem is solved by observing the interaction between the projected objects.

Until recently, line-sweep seemed to be the only way to solve problems based on sets of planar objects efficiently. In [11] Güting and Wood showed that the same efficiency can be obtained by using the well-known algorithmic paradigm of *divide-and-conquer*. There, a time- and space-optimal algorithm for the rectangle intersection problem is given. They key idea that makes efficient divide-and-conquer possible is called *separational representation* (of planar objects). In this paper we use the same technique to solve two other problems based on a set of rectangles. Both problems concern "free" and "covered" areas of the plane. (A point of the plane is called *covered* with respect to a set of rectangles R if there exists a rectangle in R containing it. Otherwise it is called *free* with respect to R. This definition extends in the obvious way to a set of intervals in 1-space).

The Measure Problem

Given a set of n iso-oriented rectangles in the plane, determine the measure of their union.
In other words we ask for the size of the covered area of the plane. This problem was solved by Bentley [2] by means of a line-sweep algorithm in optimal - i.e. $O(n \log n)$ - time.

The Contour Problem

For a set of iso-oriented rectangles in the plane, the *contour* is the boundary between the free and the covered area of the plane. In general it is a collection of disjoint "contour cycles", each of which is a sequence of alternating horizontal and vertical contour pieces.

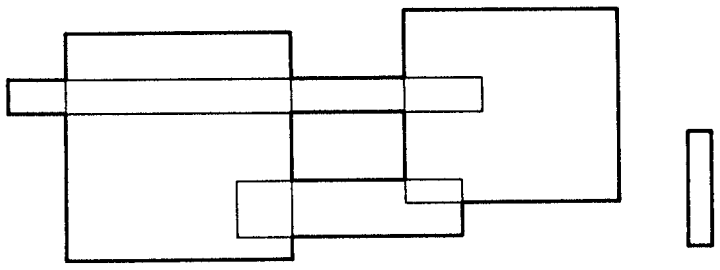

Fig. 1

The contour problem is to construct this collection of contour cycles. It was posed and first attacked by Lipski and Preparata [12]. They used a line-sweep algorithm with a segment tree as a supporting data structure to obtain an

$O(n \log n + p \log(2n^2/p))$ – time solution where n is the number of rectangles and p the number of contour pieces. Only recently a time-optimal, that is $O(n \log n + p)$ – time, solution was found by Güting [9]. Again a line-sweep algorithm was used, supported by a rather sophisticated data structure called a contracted segment tree.

In the sequel we use divide-and-conquer based on separational representation to develop time-optimal solutions for both problems. Apart from solving the specific problems our goal is to demonstrate:

1) *That* divide-and-conquer is a reasonable method to solve problems based on a set of planar objects.

2) *How* it is to be applied, namely, using separational representation.

Using a relatively abstract level of description we are able to provide the essential part of the solution of both problems in a single high-level algorithm. The paper is structured as follows: In Sect. 2 we reduce both problems to obtaining a solution from some given abstract description (called *stripes*) of the union of the rectangles. In Sect. 3 we give a divide-and-conquer algorithm *STRIPES* to compute this description. In Sect. 4 we show how to represent the description *stripes* and implement the abstract operations of algorithm *STRIPES*, and also analyze time- and space-requirements.

2. Reducing the Problems

At first glance the measure problem and the contour problem may appear rather different. However, both the measure and the contour are properties of the union of a set of rectangles. We exploit this fact by developing an abstract description of this union which makes the solution of both the measure and the contour problem relatively easy. Hence our approach has two steps: First, compute the description *stripes* of a set of rectangles using the divide-and-conquer algorithm *STRIPES*. Second, given *stripes*, solve the specific problem. Since the second step is relatively simple for both problems, it is essentially the one powerful algorithm *STRIPES* that solves both problems.

It must be noted, however, that the "abstract data structure" *stripes* is identical for both problems only on a high level of abstraction. When it comes to implementation we have to use different representations of *stripes* for the two problems to obtain efficient solutions. Thus in Sect. 4 a specialization of data structure *stripes* and algorithm *STRIPES* is introduced.

Let R be the given set of rectangles with cardinality n. At present we assume for simplicity the x- and y-coordinates of all rectangles to be pairwise distinct, that is, R defines $2n$ different x-coordinates and $2n$ different y-coordinates. This restriction is removed in Sect. 4. The description of R 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 and each (internal) stripe's boundary coincides with a horizontal edge:

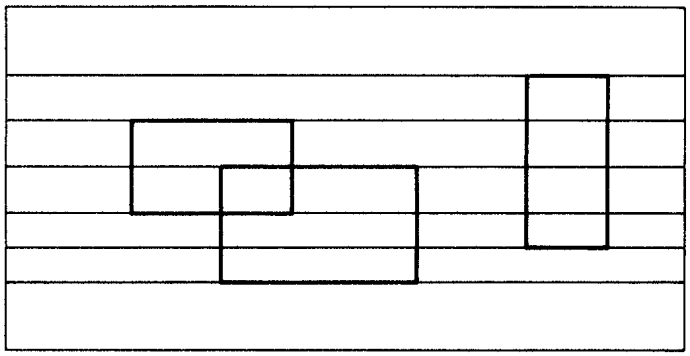

Fig. 2

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 formalize the collection of stripes and to be able to describe operations on it we need a few data type and function definitions. We use an algorithmic design notation to describe functions in the mathematical and in the computational sense in a uniform manner. Thus some of the constructs below serve for specification and cannot be implemented directly (since, for instance, they deal with infinite sets of objects).

type *coord* = $\mathbb{R} \cup \{-\infty, +\infty\}$
type *point* = (**coord** *x*, **coord** *y*)
type *interval* = (**coord** *bottom*, **coord** *top*)
type *line_segment* = (**interval** *int*, **coord** *coord*)
type *rectangle* = (**coord** *x_left*, **coord** *x_right*, **coord** *y_bottom*,
coord *y_ top*)
|(**interval** *x_interval*, **interval** *y_interval*)
(*rectangle* is a variant type, the alternatives are separated by "|")
type *edge* = (**interval** *int*, **coord** *coord*, **edgetype** *side*)
type *edgetype* = **atomic** {*left*, *right*, *bottom*, *top*}
type *stripe* = (**interval** *x_interval*, **interval** *y_interval*, **set of interval** *x_union*).

The only unusual object type is *stripe*. A stripe consists of a rectangular area defined by *x_interval* and *y_interval* and of a component *x_union* able to hold a set of intervals, which is used to represent the part of the union of the rectangles in R that falls into the stripe's area. The following definitions serve to specify the particular set of stripes defined by a given set of rectangles R.

Since a rectangle defines a set of points we may apply set operations to rectangles. *union* (R) defines the area of the plane covered with respect to R:

function *union* (**set of rectangle** R) **set of point**: $\bigcup_{r \in R} r$

y_set(R) is used below to define the partition of the plane induced by R:

function *y_set* (**set of rectangle** *R*) **set of coord**:
$\{y \in \textbf{coord} \mid \exists r \in R:\ y = r.y\_bottom \textbf{ or } y = r.y\_top\}$

function *partition* (**set of coord** *Y*) **set of interval**:
$\{[y_1, y_2] \mid y_1, y_2 \in Y \textbf{ and } y_1 < y_2 \textbf{ and } (\forall y \in Y:\ y \leqq y_1 \textbf{ or } y \geqq y_2)\}$

Furthermore we need

function *x_proj* (**set of point**) **set of coord**: ...

to project a set of points onto the x-axis. If we apply *x_proj* to a connected region of the plane we would like to interpret the result as an interval rather than a set of coordinates and therefore define a "type conversion" function

function *intervals* (**set of cord**) **set of interval**: ...

which performs just that transformation. Now we are able to define the set of stripes:

function *stripes* (**set of rectangle** *R*, **rectangle** *f*:
$(\forall y \in y\_set(R):\ f.y\_bottom < y < f.y\_top))$ **set of stripe**:
set of coord $Y := y\_set(R) \cup \{f.y\_bottom, f.y\_top\}$;
$\{(i_x, i_y, i\_set) \mid i_x = f.x\_interval$ **and**
$i_y \in partition\ (Y)$ **and**
$i\_set = intervals(x\_proj((i_x, i_y) \cap union(R)))\}$

The algorithm *STRIPES* of Sect. 3 computes *stripes* (R,f) for a frame f completely enclosing R, such that measure and contour can be computed from this description. Note that the definition is still valid if the *x_interval* of f does not contain the whole x-extension of R, like in Fig. 3:

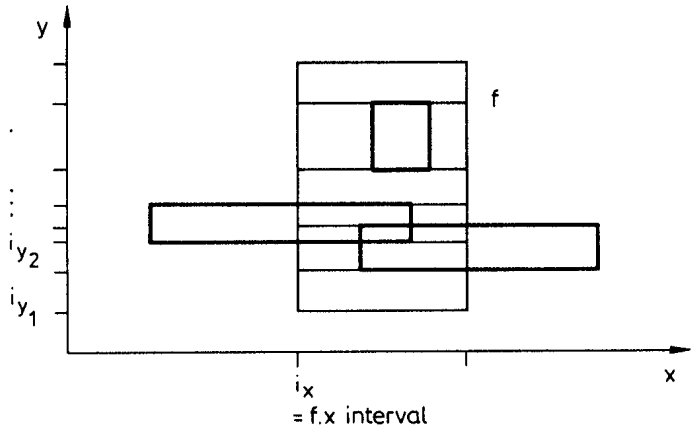

Fig. 3

Sets of stripes of this kind appear as intermediate results of the divide-and-conquer algorithm *STRIPES*.

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

The Measure Problem

Given a set of rectangles R our task is to compute *measure*(R). First we choose a frame f completely enclosing and not intersecting any of the rectangles in R. We then compute $S = stripes(R,f)$ by the divide-and-conquer algorithm *STRIPES* given in the next section. Now each stripe s in S contains a set of disjoint intervals $s.x\_union$. The only property of $s.x\_union$ relevant to the measure problem is the total length of the contained intervals, that is $measure(s.x\_union)$. Then the measure of $union(R)$ is given by

function *measure* (**set of stripe** S) **real:**

$$\sum_{s \in S} (measure(s.x\_union) * (s.y\_interval.top - s.y\_interval.bottom))$$

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 [12] have shown that it is sufficient to know the contour pieces oriented in one direction (e.g. the horizontal ones) to construct the complete contour efficiently. Our algorithm therefore only computes the horizontal contour pieces from the previously constructed data structure $S = stripes(R,f)$.

Since any horizontal contour piece is a fragment of some rectangle edge we consider the set H of horizontal rectangle edges defined by R. By construction of S any edge $h \in H$ coincides with the boundary between two stripes in S, with its rectangle either above or below it:

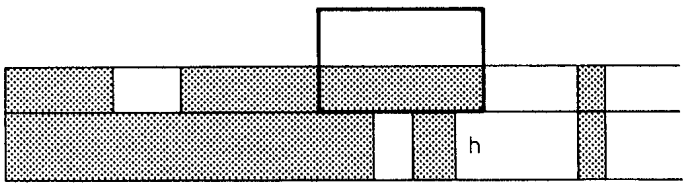

Fig. 4

The area within a stripe s free with respect to R is given by the complement of $s.x\_union$ (together with $s.y\_interval$). The contour pieces created by h are formed by the intersection of h with that adjacent stripe which is not covered by h's rectangle (the stripe below h in Fig. 4). This intersection is completely defined by $h.x\_interval$ and $s.x\_union$. The contour pieces created by h are given by:

function *contour_pieces* (**edge** h, **set of stripe** S) **set of line segment:**

if $h.side = bottom$

then select $s \in S$ **such that** $s.y\_interval.top = h.coord$
else select $s \in S$ **such that** $s.y\_interval.bottom = h.coord$
fi;
set of interval $J := intervals(h.x\_interval \setminus (h.x\_interval \cap union(s.x\_union)))$;
$\{(i, y) \mid i \in J$ **and** $y = h \cdot coord\}$

Here after selecting the stripe not covered by $h's$ rectangle the set J of $x\_intervals$ of contour pieces is formed. *union* applied to the set of intervals x_union yields the corresponding set of coordinates (defined analogously to *union* for a set of rectangles). Intersecting and forming the complement with $h.x\_interval$ we obtain the free (w.r.t. J) subintervals of $h.x\_interval$. Together with $h's$ y-coordinate this yields the contour pieces created by h.

The whole set of contour pieces is simply *contour* (H, S) defined by

function *contour* **(set of edge** H**, set of stripe** S**) set of line segment:**
$\bigcup_{h \in H} contour\_pieces(h, S)$

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 easily be solved; hence the following is the crucial part of the algorithms.

To compute S efficiently, we use divide-and-conquer. For a set of rectangles R the obvious way to do this is to choose a (say) vertical line which splits R into three subsets: the rectangles left and right of it, respectively, and those intersected by it. We may call the sets *LEFT*, *MIDDLE* and *RIGHT*. In this approach it seems difficult to treat the interaction between e.g. sets *LEFT* and *MIDDLE*.

Therefore we use *separational representation* in applying divide-and-conquer to a set of rectangles. The idea is to represent a rectangle by its left and right vertical edges which are treated by the algorithm as independent units. Hence the input for the algorithm is a set of vertical line segments instead of a set of rectangles. By a vertical line this set can be divided into only two equal-sized subsets *LEFT* and *RIGHT* which can be "conquered" independently.

We now describe in a top-down manner the algorithm which uses sub-algorithms yet to be defined. We then explain the algorithm together with describing the subalgorithms. The algorithm consists of two parts *RECTANGLE_DAC* and *STRIPES*; the quite simple main algorithm *RECTANGLE_DAC* only provides an environment for the recursive divide-and-conquer algorithm *STRIPES*.

Algorithm *RECTANGLE_DAC* (*RECT*, *S*)

Input: **set of rectangle** *RECT*: the x- and y-coordinates of all rectangles in *RECT* are pairwise distinct.

Output: **set of stripe** $S: S = stripes\ (RECT, (-\infty, +\infty, -\infty, +\infty))$

Step 1: Let *VRX* be the set of left and right vertical rectangle edges defined by *RECT*.

Step 2 *STRIPES* (*VRX*, $[-\infty, +\infty]$, *LEFT*, *RIGHT*, *POINTS*, *S*)

end of algorithm *RECTANGLE_DAC*.

(Here *LEFT*, *RIGHT* and *POINTS* are result parameters of algorithm *STRIPES*, which are not used in *RECTANGLE_DAC*).

In the following description of algorithm *STRIPES* two new terms occur. For a vertical edge of a rectangle r we call the respective other vertical edge its *partner*. For a set V of vertical edges $rect(V)$ denotes the set of those rectangles which are represented in V by at least one edge.

Algorithm $STRIPES(V, x\_ext, L, R, P, S)$

Input: **set of edge** V: a set of vertical rectangle edges.
interval x_ext: encloses the x-coordinates of all edges in V.

Output: **set of interval** L: contains the y-projections of all left edges whose partner is not in V.
set of interval R: symmetric to L (for right edges).
set of cord P: contains the y-projection of all endpoints of line segments in V plus the frame boundaries in y-direction, namely $-\infty$ and $+\infty$.
set of stripe S: $S = stripes(rect(V), (x\_ext, [-\infty, +\infty]))$.

Case 1: V contains only one edge v.

if $v.side = left$
then $L := \{v.y\_interval\}$; $R := \emptyset$
else $L := \emptyset$; $R := \{v.y\_interval\}$
fi;
$P := \{-\infty, v.y\_interval.bottom, v.y\_interval.top, +\infty\}$;
$S := \{(i_x, i_y, \emptyset) \mid i_x = x\_ext$ **and** $i_y \in partition(P)\}$; (A)
select $s \in S$ **such that** $s.y\_interval = v.y\_interval$;
if $v.side = left$
then $s.x\_union := \{[v.coord, x\_ext.top]\}$ (B)
else $s.x\_union := \{[x\_ext.bottom, v.coord]\}$ (C)
fi

Case 2: V contains more than one edge.

Divide: Choose an x-coordinate x_m dividing V into two approximately equal-sized subsets V_1 and V_2.

Conquer: $STRIPES(V_1, [x\_ext.bottom, x_m], L_1, R_1, P_1, S_1)$;
$STRIPES(V_2, [x_m, x\_ext.top], L_2, R_2, P_2, S_2)$;

Merge: **set of interval** $LR := L_1 \cap R_2$;

$L := (L_1 \setminus LR) \cup L_2$;
$R := R_1 \cup (R_2 \setminus LR)$;
$P := P_1 \cup P_2$;

set of stripe $S_{left} := copy(S_1, P, [x\_ext.bottom, x_m])$;
set of stripe $S_{right} := copy(S_2, P, [x_m, x\_ext.top])$;

$blacken(S_{left}, R_2 \backslash LR)$;
$blacken(S_{right}, L_1 \backslash LR)$;

$S := concat(S_{left}, S_{right}, P, x\_ext)$

end of algorithm *STRIPES*.

Let us now look at the algorithm *STRIPES* in detail and define the used subalgorithms *copy*, *blacken* and *concat*. The task of *STRIPES* is to construct sets L, R, P and S according to the output specification from a given set of vertical rectangle edges V. Although we are interested only in the set of stripes S as a final result, sets L, R and P are needed to support the merge step.

Initially *STRIPES* is called by *RECTANGLE_DAC* with the complete set of vertical rectangle edges. All edges are located within a "maximal" frame (a rectangle) $f=(-\infty, +\infty, -\infty, +\infty)$. A vertical line is chosen that splits f into two smaller subframes f_1 and f_2 and V into two subsets V_1 and V_2 of approximately equal size enclosed by f_1 and f_2, respectively. Recursively this process is repeated yielding smaller and smaller frames 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. S is constructed in two steps: First, an "empty" set of stripes is formed, that is, all *x_union* fields are assigned $\emptyset$. The single edge v leads to the construction of three stripes. Second, the stripe containing v is selected and assigned the x-interval describing its coverage by the rectangle r of which v is an edge. Hence now $S=stripes(\{r\}, (x\_ext, [-\infty, +\infty])$ and the output specification holds. The resulting set of stripes can be represented (for instance in case of a left edge):

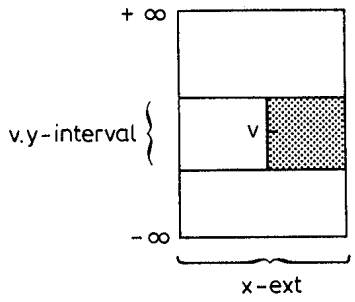

Fig. 5

In Case 2, the algorithm is given a set of vertical edges V located within some frame $f=(x\_ext, [-\infty, +\infty])$.

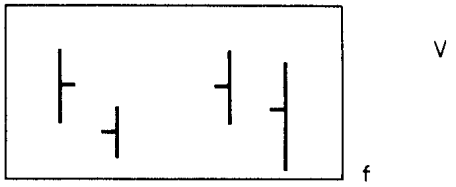

Fig. 6

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.

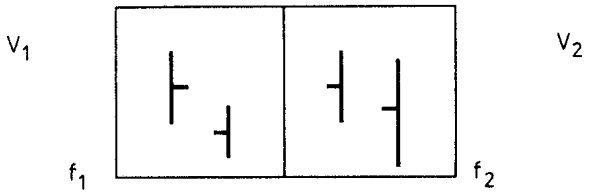

Fig. 7

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

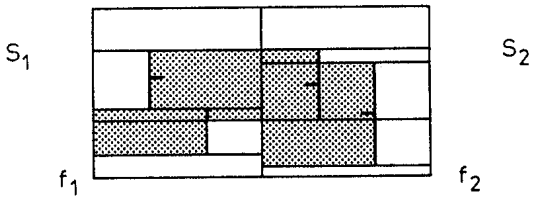

Fig. 8

In the merge step sets L, R, P and S are constructed from L_i, R_i, P_i and S_i, for $i=1,2$. For L, R and P this is done in the obvious way; the subtraction of set LR corresponds to removing rectangles that are completed in the merge step, that is rectangles whose left edge is in V_1 and whose right edge is in V_2. In this way L and R represent only edges of incomplete rectangles as required in the output specification.

The construction of the set of stripes S is a little more difficult. It is done in three steps *copy*, *blacken* and *concat*. We briefly illustrate these steps before explaining them in detail.

First, the different y-partitions of S_1 and S_2 are made equal, that is, stripe boundaries of S_1 are extended into S_2 and vice versa. In this step the coverage of area in f_1 and f_2, as given by *x_union* fields, is *not* changed. The copies of S_1 and S_2 are called S_{left} and S_{right}, respectively.

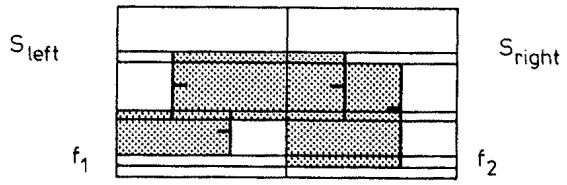

Fig. 9

Second, certain stripes in S_{left} and S_{right} are "blackened" (that is, their x_union fields are updated) to indicate that they are completely covered by a rectangle. Why it suffices to blacken complete stripes and how to select these stripes we shall explain below.

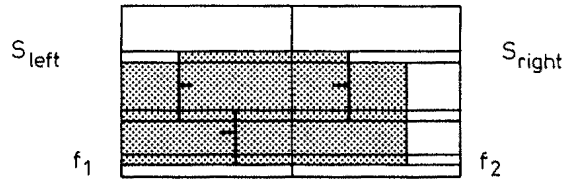

Fig. 10

Third, every two adjacent stripes in S_{left} and S_{right} (stripes with the same y-interval) are concatenated to form the stripes of S. Again in this step the coverage of area is *not* changed.

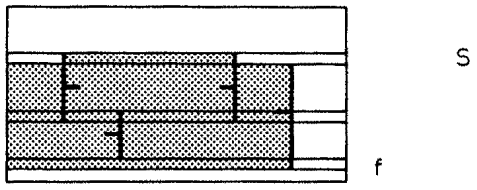

Fig. 11

This completes the construction of $S = stripes\ (rect(V), f)$ as required in the output specification.

Let us now examine the three steps *copy*, *blacken* and *concat* in more detail. In the *copy* step copies S_{left} and S_{right} are produced from S_1 and S_2, respectively, that are based on the set P containing *all* partition points occurring in either S_1 or S_2. This is done by the statements

$S_{left} := copy\ (S_1, P, [x\_ext.bottom, x_m])$;
$S_{right} := copy\ (S_2, P, [x_m, x\_ext.top])$

where

function *copy* (**set of stripe** S, **set of coord** P, **interval** x_int)
set of stripe:
set of stripe $S' := \{(i_x, i_y, \emptyset) \mid i_x = x\_int$ **and** $i_y \in partition(P)\}$;
forall $s' \in S'$:

select $s \in S$ **such that** $s.y\_interval \supseteq s'.y\_interval$;
$s'.x\_union := s.x\_union$; (D)
S'

We know from the output specification that $S_1 = stripes\ (rect(V_1), f_1)$ and $S_2 = stripes\ (rect(V_2), f_2)$. In other words S_1 represents the union of the rectangles represented in V_1 by at least one edge (as far as it intersects f_1), similarly S_2 those represented in V_2. Since the *copy* step only introduces a finer y-partition, S_{left} and S_{right} still represent the same union, respectively. To construct S representing the union of the rectangles represented by an edge in $V = V_1 \cup V_2$, in the *blacken* step S_{left} is updated with respect to rectangles represented in V_2, similarly S_{right} for those in V_1.

In the sequel we only describe the updating of S_{right} (for S_{left} it is symmetric).

Lemma 3.1. *Only those edges in V_1 that are represented by an interval in $L_1 \backslash LR$ may cause changes of S_{right}.*

Proof. Consider an arbitrary edge $v \in V_1$ belonging to some rectangle r. There are four cases:

a) v is a right edge

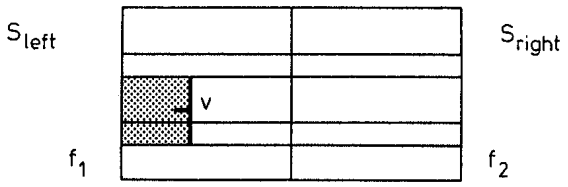

Fig. 12

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.

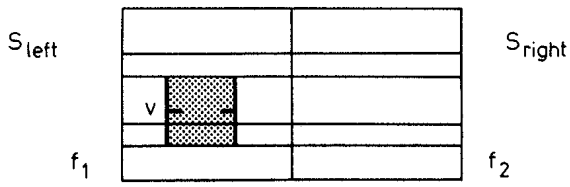

Fig. 13

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

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

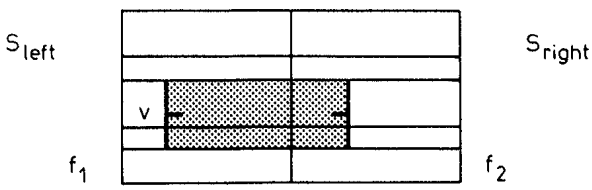

Fig. 14

Then the intersection of r with f_2 is already represented in S_{right}, because the right edge is in V_2 and 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.

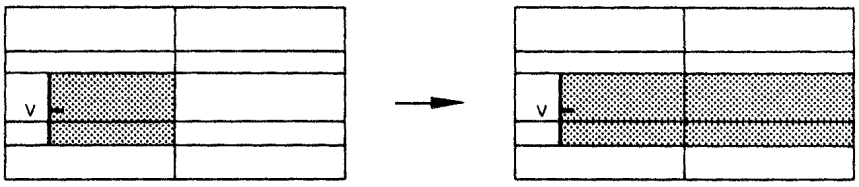

Fig. 15

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.

To summarize, only left edges in V_1, whose partner is neither in V_1 nor in V_2, may cause changes of S_{right}. Precisely those edges are represented by their y-intervals in $L_1 \backslash LR$. □

Lemma 3.2. *Any stripe in S_{right} is either completely covered by a rectangle represented in $L_1 \backslash LR$ or does not intersect such a rectangle.*

Proof. It suffices to observe that a stripe in S_{right} contains neither a horizontal nor a vertical edge of such a rectangle. It contains no horizontal edge because the y-partition of S_{right} is based on the endpoints of all edges in $V_1 \cup V_2$. It contains no vertical edge since for all rectangles represented in $L_1 \backslash LR$ the left edge is in V_1 (that is, left of f_2) and the right edge to the right of f_2.

Lemma 3.3. *Precisely those stripes in S_{right}, whose y-interval is enclosed by some y-interval in $L_1 \backslash LR$, are completely covered and have to be blackened.*

Proof. See the proof of Lemma 3.1, Case d). □

It follows that S_{left} and S_{right} are updated correctly by the two calls

blacken $(S_{left}, R_2 \backslash LR)$;
blacken $(S_{right}, L_1 \backslash LR)$

where

procedure *blacken* (**var set of stripe** S, **set of interval** J):

forall $s \in S$:
if exists $i \in J$ **such that** $s.y\_interval \subseteq i$
then $s.x\_union := \{s.x\_interval\}$ (E)
fi

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

$S := concat\ (S_{left}, S_{right}, P, x\_ext)$

where

function *concat* (**set of stripe** S_1, **set of stripe** S_2, **set of coord** P,
interval x_int) **set of stripe**:

set of stripe $S := \{(i_x, i_y, \emptyset) \mid i_x = x\_int$ **and** $i_y \in partition\ (P)\}$;
forall $s \in S$:
select $s_1 \in S_1$ **such that** $s_1.y\_interval = s.y\_interval$;
select $s_2 \in S_2$ **such that** $s_2.y\_interval = s.y\_interval$;
$s.x\_union := s_1.x\_union \;\&\; s_2.x\_union$; (F)
S

The operation $J \& J'$ 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.

4. Implementation

In Sects. 2 and 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 Sect. 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 $s.x\_union$. Recall that to solve the measure problem only *measure* ($s.x\_union$) is needed. Hence we simply represent $s.x\_union$ by a real number and define

type $stripe = ($**interval** $x\_interval$, **interval** $y\_interval$, **real** $x\_measure)$

In Sect. 3, all lines describing operations on $s.x\_union$ are marked by capital letters. To solve the measure problem we simply replace these lines by the following:

(A) $S := \{(i_x, i_y, 0) \mid i_x = x\_ext$ **and** $i_y \in partition\ (P)\}$
(B) $s.x\_measure := x\_ext.top - v.coord$
(C) $s.x\_measure := v.coord - x\_ext.bottom$
(D) $s'.x\_measure := s.x\_measure$
(E) $s.x\_measure := s.x\_interval.top - s.x\_interval.bottom$
(F) $s.x\_measure := s_1.x\_measure + s_2.x\_measure$

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

The Contour Problem

As we have seen in Sect. 2 the representation of $s.x\_union$ for the contour problem must support a *free subinterval query:*

Given a set of disjoint x-intervals J and a query interval $q=[x_1, x_2]$, report the subintervals of q that are free with respect to J. (These subintervals are given technically by *intervals* $(q \setminus (q \cap union(J)))$).

The representation we choose is a binary search tree storing the endpoints of the intervals in $s.x\_union$ in its leaves. A leaf also contains an indication whether the represented point is a left or a right endpoint. This leads to the definitions:

type *stripe* = (**interval** *x_interval*, **interval** *y_interval*, **ctree** *tree*)
type *ctree* = **empty** | (**real** *x*, **lru** *side*, **ctree** *lson*, **ctree** *rson*)
type *lru* = **atomic** {*left*, *right*, *undef*}

Again we describe operations (A)–(F):

(A) $S := \{(i_x, i_y, \textbf{empty}) \mid i_x = x\_ext \textbf{ and } i_y \in partition\ (P)\}$
(B) $s.tree := (v.coord, left, \textbf{empty}, \textbf{empty})$
(C) $s.tree := (v.coord, right, \textbf{empty}, \textbf{empty})$
(D) $s'.tree := s.tree$
(E) $s.tree := \textbf{empty}$
(F) $s.tree :=$ **if** $s_1.tree \neq \textbf{empty} \neq s_2.tree$ **then**
$(s_1.x\_interval.top, undef, s_1.tree, s_2.tree)$
$\square$ $s_1.tree \neq \textbf{empty} = s_2.tree$ **then** $s_1.tree$
$\square$ $s_1.tree = \textbf{empty} \neq s_2.tree$ **then** $s_2.tree$
$\square$ $s_1.tree = \textbf{empty} = s_2.tree$ **then empty fi**

From the point of view of implementation it is important to note that the field *tree* of a stripe contains a pointer to some structure. Especially the operations (D) and (F) do not copy trees but manipulate pointers. Hence the three operations *copy*, *blacken* and *concat* require only constant time per stripe. They can be illustrated as follows:

copy

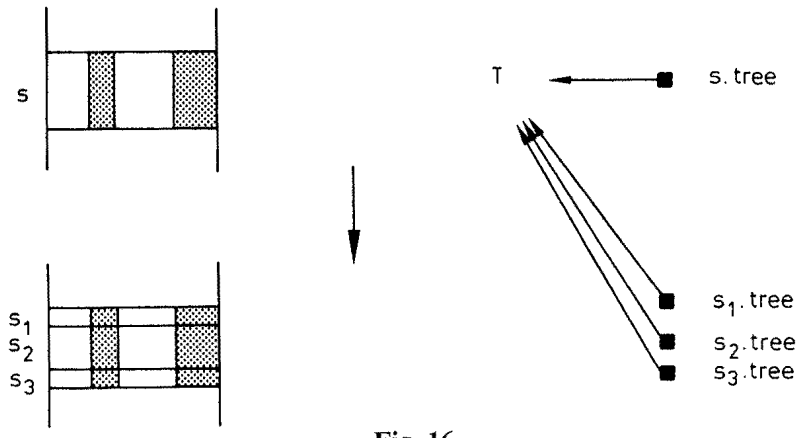

Fig. 16

blacken

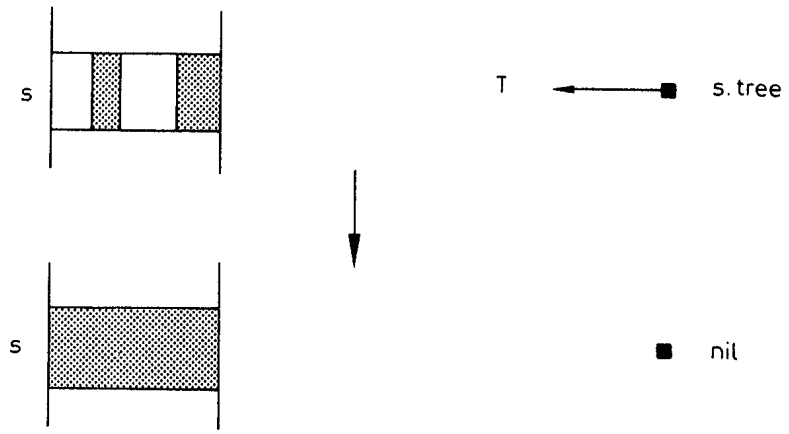

Fig. 17

concat

(Case: Both T_1 and T_2 are non-empty.)

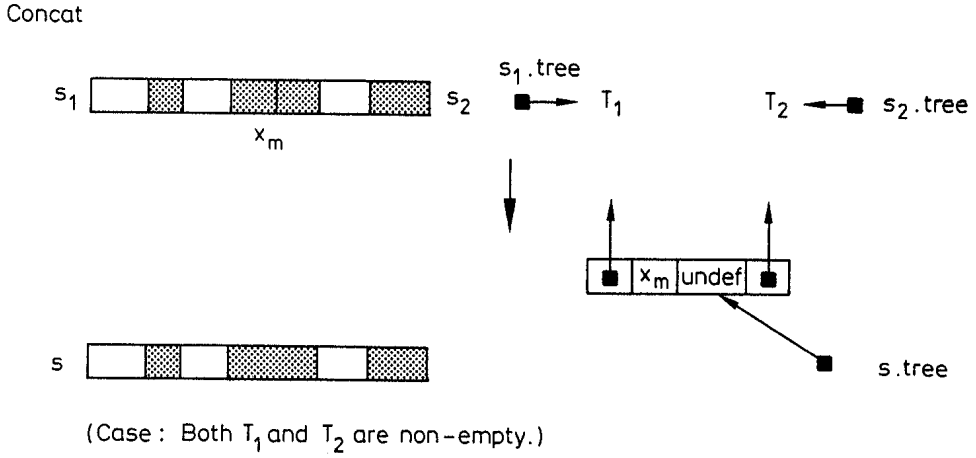

Fig. 18

The pointer operations have the effect that two trees belonging to different stripes may have common subtrees. On the conceptual level, however, this doesn't matter and we can still maintain the view that each stripe has its own separate tree. In fact, the collection of nodes accessible via the tree pointer of any particular stripe s is simply a binary search tree.

Since *s.tree* is a search tree, it is certainly possible to answer a free subinterval 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 $[a, b]$ within $[x_1, x_2]$ for every pair of subsequent leaves (a, *right*, **empty, empty**) and (b, *left*, **empty, empty**) found between x_1 and x_2. This takes $O(h+\alpha)$ time where h is the height of the tree and α 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 L, R, P and S 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). Scanning a list we keep a count of the number of currently present intervals by adding 1 when encountering a bottom point and subtracting 1 for an encountered top point. In this way 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

$$T(1)=O(1) \tag{1}$$

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 lists in parallel. We leave it to the reader to check that the operations during those scans take only constant time for each list element. Hence the operations in the merge step require linear time in total. Therefore

$$T(n)=O(1)+2\,T(n/2)+O(n) \tag{2}$$

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

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

(see for instance [1]).

Space

Clearly the array representing V 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 is 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 S can be stored in $O(n)$ space.

In case of the contour problem we have to look at the way a set of stripes is created. 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

$$SP(1)=O(1)$$

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

$$SP(n) = 2SP(n/2) + O(n)$$

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 Sect. 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 free subinterval query is performed on the corresponding stripe's tree. Since the height of this tree is $O(\log n)$, each query takes $O(\log n + \alpha)$ time (α 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 [12] the contour can then be constructed as a set of linked contour cycles in $O(p)$ time and space.

We summarize our results in:

Theorem 4.1. *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.*

Multiple x- or y-Coordinates

We now drop the initial restriction that the x- and y-coordinates of all rectangles have to be pairwise distinct. From now on we only assume the x- and y-coordinates of any particular rectangle to be distinct, that is, the rectangle does not degenerate to a line segment or a point.

To accomodate multiple x- or y-coordinates we slightly modify the algorithm or its implementation, respectively. Concerning multiple x-coordinates it is not obvious anymore how to split the set of edges V in the divide step since many edges may share an x-coordinate. We proceed as follows: In the initial sorting of all vertical rectangle edges (Step 1 of algorithm *RECTANGLE_DAC*) all left edges are put before all right edges with the same x-coordinate in the sorted order. Input for algorithm *STRIPES* is then a sorted sequence of edges $V = v_i \ldots v_j$ (initially $v_1 \ldots v_{2n}$) which is divided by selecting a median edge v_m and forming $V_1 = v_i \ldots v_m$ and $V_2 = v_{m+1} \ldots v_j$. The condition "left edges appear before right edges" ensures a correct functioning of the algorithm in situations like that depicted in Fig. 19a):

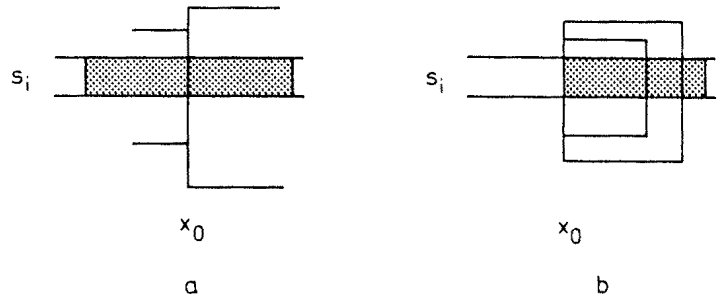

Fig. 19

In this case a left and right edge on the same x-coordinate x_0 remove each other in the *blacken* operation such that no interval boundary remains in stripe s_i. It is also easy to check that from a group of left edges sharing an x-coordinate (see Fig. 19b)) only one edge survives the *blacken* operations to form an interval boundary in stripe s_i.

The trees representing $s.x\_union$ constructed by the algorithm may now contain internal nodes and leaves with the same x-coordinate. However, since all leaves have distinct x-coordinates, at most two internal nodes and one leaf may have the same x-coordinate. It is not difficult to adapt the free subinterval search algorithm to this case.

Concerning multiple y-coordinates it becomes a problem to compute efficiently the intersection set $LR = L_1 \cap R_2$. Obviously it doesn't suffice anymore to look for list elements with the same y-coordinate in the lists (representing) L_1 and R_2. Therefore we initially assign a unique identification number to each rectangle (for instance the numbers $1, \ldots, n$). Each element of an interval list is augmented by the number of the rectangle from which it originates. The resulting pairs (y-coordinate, rectangle number) are kept in lexicographical order in the lists. That is, the merge step maintains this order which is given trivially in the initial lists built in Case 1 of algorithm *STRIPES*. – Since there exists again a total order on the list elements it is once more possible to form the intersection of L_1 and R_2 (in other words to detect common elements in both lists) in linear time.

Clearly the modified algorithm has the same time- and space-complexity and treats multiple x- and y-coordinates correctly.

5. Conclusions

In this paper we presented time-optimal divide-and-conquer solutions for the measure and the contour problem. The central part of the solution is algorithm *STRIPES* computing a compact representation *stripes* of the union of a set of rectangles.

Data structure *stripes* (in its specialization for the contour algorithm) essentially provides a partition of *union* (R) into disjoint rectangles. As such it might have other applications, for instance, based on *stripes* it is possible to compute efficiently integrals ranging over *union* (R). If several integrals are to be computed then it seems reasonable to maintain data structure stripes independently

from algorithm *STRIPES*. This could be appropriate in various situations where different properties of one set of rectangles are of interest.

The results of this paper were achieved using a new approach to divide-and-conquer called "separational representation" of planar objects. Together with a previous paper [11] the following problems have so far been solved using this technique:

1) Finding all intersections in a set of horizontal and vertical line segments.
2) Finding all point enclosures in a mixed set of points and rectangles.
3) The rectangle intersection problem, by combining 1) and 2).
4) The measure problem.
5) The contour problem.

This seems to be sufficient evidence to support the belief that for problems involving iso-oriented planar objects divide-and-conquer using separational representation is as powerful as the line-sweep paradigm. As a consequence, when dealing with new problems of this kind, it seems advisable to try both algorithmic paradigms. Perhaps there exist problems which can be solved more easily by divide-and-conquer than by line-sweep.

Comparing both paradigms, 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.

We suggest the following directions for further work:

1) Define a class of problems for which both paradigms are equivalent. That is, whenever a line-sweep algorithm exists to solve a problem in this class, then the corresponding divide-and-conquer algorithm can be given and vice-versa.
2) Study the relation between corresponding data structures. For instance for the measure problem the corresponding data structures are a linked list of numbers and the segment tree. Is it possible to give rules how to define the "partner" data structure, if one of them is given?
3) Line-sweep algorithms have been used successfully to solve problems involving non-orthogonal objects (for instance see the algorithm finding all intersections in a set of arbitrarily oriented line segments in 2-space [3]). Can the applicability of divide-and-conquer be extended to this kind of problems?
4) Try divide-and-conquer on problems for which time-optimal line-sweep algorithms could not yet be obtained.

Acknowledgements. I would like to thank Derick Wood for his encouragement during this work and Armin B. Cremers for his comments on the manuscript. Thanks also to two anonymous referees for many useful remarks that led in particular to a cleaner notation for algorithms and data structures.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms. Reading, MA: Addison-Wesley 1974

2. Bentley, J.L.: Solutions to Klee's rectangle problems. Carnegie-Mellon University, Department of Computer Science, 1977 (Unpublished)
3. Bentley, J.L., Ottmann, T.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. C-**28**, 643-647 (1979)
4. Bentley, J.L., Shamos, M.I.: Optimal algorithms for structuring geographic data. In: Proceedings, Symposium on Topological Data Structures for Geographic Information Systems, Harvard University, pp. 43-51, 1977
5. Bentley, J.L., Wood, D.: An optimal worst-case algorithm for reporting intersections of rectangles. IEEE Trans. Comput. C-**29**, 571-577 (1980)
6. Eastman, C.M., Lividini, J.: Spatial search. Carnegie-Mellon University, Institute of Physical Planning, Report 55, 1975
7. Edelsbrunner, H.: Dynamic rectangle intersection searching. Technical University of Graz, Institut für Informationsverarbeitung, Report F47, 1980
8. Edelsbrunner, H., van Leeuwen, J., Ottmann, T., Wood, D.: Connected components of orthogonal geometrical objects. RAIRO Theoretical Informatics 18, 171-183 (1984)
9. Güting, R.H.: An optimal contour algorithm for iso-oriented rectangles. McMaster University, Unit for Computer Science, Report 82-CS-03, 1982 (To appear)
10. Güting, R.H.: Conquering contours. Efficient algorithms for computational geometry. Universität Dortmund, Lehrstuhl Informatik VI, Ph. D. Thesis, 1983
11. Güting, R.H., Wood, D.: Finding rectangle intersections by divide-and-conquer. IEEE Trans. Comput. C-33, 671-675 (1984)
12. Lipski, W., Preparata, F.P.: Finding the contour of a union of isooriented rectangles. J. Algorithms **1**, 235-246 (1980)
13. McCreight, E.M.: Efficient algorithms for enumerating intersecting intervals and rectangles. XEROX Palo Alto Research Center, Report CSL-80-9, 1980
14. Mead, C., Conway, L.: Introduction to VLSI-Systems. Reading, MA: Addison-Wesley, 1980
15. Shamos, M.I., Hoey, D.: Geometric intersection problems. Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science, pp. 208-215, 1976
16. Soisalon-Soininen, E., Wood, D.: Optimal algorithms to compute the closure of a set of iso-rectangles. J. Algorithms 5, 199-214 (1984)

Received September 3, 1982 / June 18, 1984

Note Added in Proof

Recently another time-optimal line-sweep algorithm for the contour problem was found by Wood:

Wood, D., The contour problem for rectilinear polygons. University of Waterloo, Computer Science Department, Report CS-83-20, 1983

It can be seen as an improvement of [9] because the data structure supporting the line-sweep is simpler