DYNAMIZATION OF ORDER DECOMPOSABLE SET PROBLEMS

Mark H. Overmars

RUU-CS-80-9
oktober 1980

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

DYNAMIZATION OF ORDER DECOMPOSABLE SET PROBLEMS

Mark H. Overmars

Technical Report RUU-CS-80-9
oktober 1980

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

Proposed running head:

DYNAMIZATION OF SET PROBLEMS

All correspondence to:

Mark H. Overmars
Dept. of Computer Science
University of Utrecht
Princetonplein 5
P.O. Box 80.002
3508 TA Utrecht
the Netherlands

DYNAMIZATION OF ORDER DECOMPOSABLE SET PROBLEMS*

Mark H. Overmars

Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Abstract. We define a general class of problems concerning sets of objects called "order decomposable set problems". We give a method to perform insertions and deletions in order decomposable set problems with low worst-case bounds. Examples include the maintenance of the 2-dimensional Voronoi diagram of a set of points, and of the convex hull of a dynamically changing 3-dimensional pointset in linear time. We show that there is a strong connection between the general dynamization method given and the well-known technique of divide-and-conquer used for solving many problems concerning static sets.

Keywords and phrases. Order decomposable set problems, dynamization, divide and conquer, convex hull, halfspaces, maximal elements, line segments, closest point problems, Voronoi diagram.

1. Introduction.

In the past few years, much effort was made to develop general techniques for dynamizing problems (i.e. for turning static solutions into dynamic ones). Bentley [1] first noted that such a general approach was especially relevant to the class of so-called decomposable searching problems. He gave a dynamization method for these problems that supported insertions in low average time bounds without sacrificing much of the efficiency of query answering. Soon after, a number of other dynamization methods for decomposable searching problems were developed ([10, 13, 19]), some also able to handle deletions for special subclasses of decomposable searching problems with good average bounds ([9, 11, 22, 23]). More recently it was shown that good worst-case bounds

---

* This work was supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

can be obtained as well ([16]). It was noted ([19]) that, although the number of decomposable searching problems was considerable, there were still many problems of a decomposable flavour that were not included.

Overmars and van Leeuwen [12, 14, 15] showed that some of these problems could be dynamized using a different technique. The three main problems they considered were the maintenance of the convex hull of a dynamically changing 2-dimensional pointset, the maintenance of the intersection of a set of half-spaces in the plane and the maintenance of the contour of maximal elements of a 2-dimensional point set. Their maintenance algorithms resulted in $O(\log^2 n)$ worst-case insertion and deletion times. A number of interesting applications was offered as well, including the maintenance of the feasible region of a 2-dimensional linear programming problem.

For all three problems considered in [12, 14, 15] a similar technique was used to obtain the maintenance algorithms. First, for each of the problems an appropriate notion of decomposability was defined. The decomposability consisted of a method to derive the answer of the problem for the total set from the answer of two, in some way separated, subsets. After this method was devised, the technique used to dynamize the three problems was the same. In this paper we exploit this technique further. We define a general class of problems called "order decomposable set problems" for which some notion of decomposability exists. The class contains the three problems considered by Overmars and van Leeuwen [12, 14, 15], but it contains many more problems also (see Section 4). We show that the technique described in [12, 14], when suitably modified and extended, can be used to obtain efficient dynamic solutions to all these problems. It appears that there is a strong connection between the existence of a divide-and-conquer algorithm for a set problem and the notion of order decomposability for the problem as defined here. More precisely, the notion of order decomposability of a set problem always leads to an easy divide-and-conquer algorithm for the static version of the problem. Also, most set problems for which a solution, based on a divide and conquer strategy, exists are order decomposable and can therefore be dynamized by the method described in this paper.

In Section 2 we define the class of order decomposable set problems and show the connection with the divide-and-conquer technique. In Section 3 we adapt the method of [12, 14] to obtain a general dynamization technique for all order decomposable set problems. In Section 4 we give a number of examples of order decomposable set problems and show how the dynamization technique of Section 3 can be used. It shows maintenance algorithms for the convex hull of a 3-dimensional point set, the closest pair of points in a set and the Voronoi-diagram of a 2-dimensional point set with linear insertion and deletion times. We also show how to maintain the union (and intersection) of a set of line segments in $O(\log^2 n)$ steps per transaction.

2. Order decomposable set problems.

For a number of set problems it is possible to derive the answer for the total set from the answer for two in some way separated "halves" of the set, sometimes using some kind of preconditioning of the data. These set problems we call "order decomposable set problems". More precisely,

Definition. A set problem P is called order decomposable if and only if there exists an ordering ORD, a "preconditioning" PRE and two "functions" $\square$ and Δ such that for each set $\{a_1, \ldots, a_n\}$ ordered according to ORD

(i) $\forall_{1 \le i < n}\ P(a_1, \ldots, a_n) = \square\ (P(a_1, \ldots, a_i), P(a_{i+1}, \ldots, a_n), PRE(a_1, \ldots, a_i), PRE(a_{i+1}, \ldots, a_n))$

(ii) $\forall_{1 \le i < n}\ PRE(a_1, \ldots, a_n) = \Delta\ (PRE(a_1, \ldots, a_i), PRE(a_{i+1}, \ldots, a_n))$

where $\square$ leaves $PRE(a_1, \ldots, a_i)$ and $PRE(a_{i+1}, \ldots, a_n)$ unchanged. Let C(n) denote the maximal time needed by $\square$ and Δ when the set contains n points.

We assume that C(n) is "smooth" and nondecreasing. The PRE-"structures" may be empty. Condition (i) states that once we have the answer for two consecutive halves (according to ORD), together with some extra information (in the form of preconditioning), we can obtain the answer for the total set in C(n) steps. Condition (ii) is needed to be able to concatenate (or split) PRE-structures, to avoid that we have to build them each time again. Note that the use of the word "answer" to a set problem is a bit confusing. Often this answer will consist of entire configurations or structures. There is a slight problem with the definition of order decomposable set problems. Careful reading of it results in the notion that every set problem is order decomposable, namely in the following sense. Choosing for ORD an arbitrary ordering and for PRE the total set then, clearly, Δ exists. Yet we can take for $\square$ a static solution to P using the information in the PRE-structures only. In this way we do not make use of $P(a_1, \ldots, a_i)$ and $P(a_{i+1}, \ldots, a_n)$. Of course C(n) will be large. There is no way of avoiding this problem (except maybe by stating that C(n) must be smaller than any known static solution of P). All theorems below also hold for this kind of order decomposable set problems, but give rise to awful bounds in this case.

A well-known method for solving a number of static set problems is the divide-and-conquer technique ([2, 3, 5, 17, 20]). There is a strong connection between the notion of order decomposability and divide-and-conquer. The divide-and-conquer technique asks for some kind of decomposability, often very similar to the one described in the definition of order decomposable set problems. It follows that most of the problems using divide-and-conquer are order decomposable . On the other hand, for each order decomposable searching problem one

can develop a divide-and-conquer algorithm quite easily (assuming we have algorithms for $\square$ and Δ).

Theorem. Let P be an order decomposable set problem. There exists a divide-and-conquer solution to P that takes $O(n + ORD(n))$ steps when $C(n) = O(n^{\varepsilon})$ for $0 < \varepsilon < 1$ and $O(\log n.C(n) + ORD(n))$ steps otherwise, where n is the number of points in the set and ORD(n) is the time required to order n points according to ORD.

Proof

Let the point set for which we want to solve P be $S = \{p_1, \ldots, p_n\}$. We first order S according to ORD, thus obtaining an ordered set $S' = \{a_1, \ldots, a_n\}$ ($a_i = p_j$ for some j, $a_i \neq a_j$ for $i \neq j$). To solve P we split S' in two halves, solve P and build the PRE-structures for the two halves and concatenate the structures using $\square$ and Δ. The algorithm is best described by the following recursive procedure.

```
proc ALG(S', P, PRE);
  {S' contains an ordered part of the point set. Let S = {a_1, ..., a_u}.
   ALG will return the answer to S' in P and the preconditioning
   for S' in PRE.}
begin
   if l = u
      then {S' contains only one point}
           build P and PRE directly
      else m := ⌊(l + u)/2⌋;
           {split S' in two halves and obtain the answers to the
            two halves by recursive calls of ALG....}
           ALG({a_l, ..., a_m}, P_1, PRE_1);
           ALG({a_{m+1}, ..., a_u}, P_2, PRE_2);
           {... and build the answer for S' out of the answers for
            the halves}
           P := □ (P_1, P_2, PRE_1, PRE_2);
           PRE := Δ (PRE_1, PRE_2)
   fi
end of ALG;
```

We call the procedure as ALG($\{a_1, \ldots, a_n\}$, P, PRE) and the answer will be delivered in P.

The ordering of S takes time ORD(n) by definition. To obtain a bound for the cost of ALG note that, beside the recursive calls, ALG takes O(C(n)) steps

(n the number of points it is working on). Hence we get the following recurrence

$$T(n) = 2T(\tfrac{1}{2}n) + C(n)$$

One easily verifies that $T(n) = O(n)$ when $C(n) = O(n^{\varepsilon})$ for some $0 < \varepsilon < 1$ and that $T(n) = O(\log n).C(n)$ otherwise. The bounds follow.

□

Often $ORD(n) = O(n \log n)$ but there are order decomposable set problems for which □ and Δ need no ordering at all (for example MAX that asks for the largest element in the set). In this case we can take $ORD(n) = O(1)$.

3. Dynamization of order decomposable set problems.

As we have shown in the previous section there exist efficient algorithms to solve static order decomposable set problems. But often one would like to maintain the answer to an order decomposable set problem while inserting and/or deleting points in the set. In this section we give a method to maintain these answers efficiently. To achieve this we keep track of how the divide-and-conquer algorithm splits the set and how it "merges" answers for subsets. The structure we use for this is quite similar to the one described by Overmars and van Leeuwen [12]. It keeps track of all points currently in the set and keeps the answer P up to date, while insertions and deletions are processed. The main part of the structure consists of a balanced search tree T in which we keep points at the leaves, ordered by ORD. (When ORD consists of no ordering at all we choose a lexicografic ordering, just to be able to find points.) T indicates the splittings in the divide-and-conquer algorithm. At each internal node α of the tree we would like to have the complete P and PRE structures of the points below it. Let α be a node of T with sons γ and δ It follows from the definition of order decomposability that, once we have the P and PRE-structures at γ and δ, we can obtain the P and PRE structure at α in $O(C(n))$ time by using □ and Δ (in that order) (see figure 1). There is only one problem. There is no garantee in the definition of order decomposable set problems that □ and Δ leave the P and PRE structures they work on unchanged. In most of the examples offered in Section 4 these structures are indeed largely destroyed. Copying the P and PRE structures before using them may take much more than $O(C(n))$ time. Therefore we have to develop another strategy.

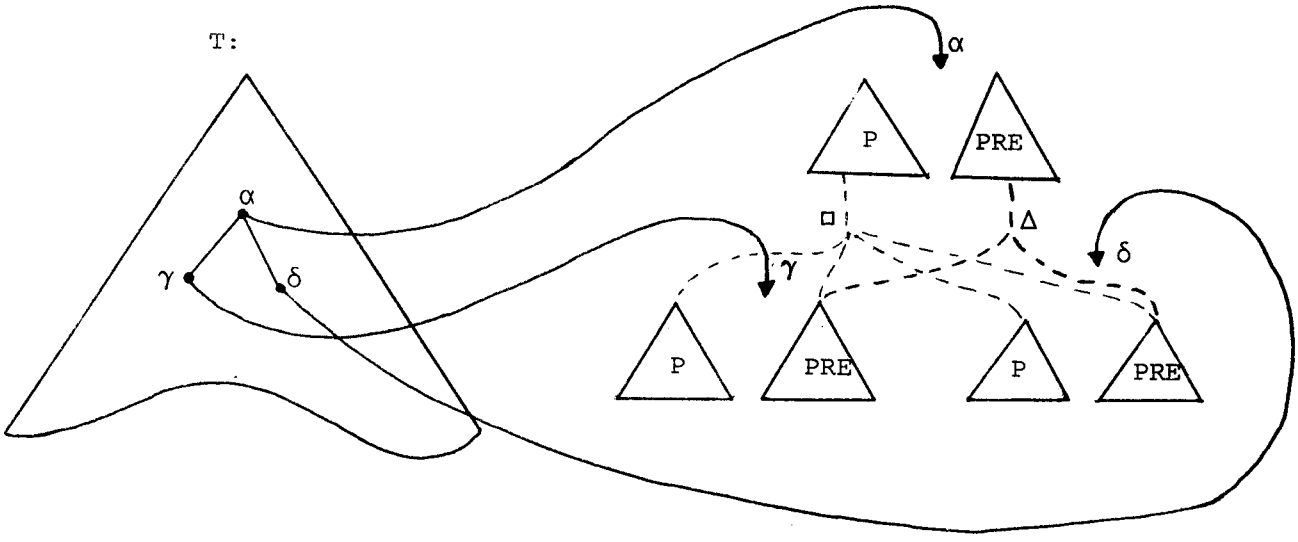


figure 1

Let P_α and PRE_α denote the P and PRE structures at α (build from the points at the leaves of T below α). When we want to build P_α and PRE_α out of the structures at the sons γ and δ of α, we keep track (at α) of how we did this and at γ and δ we save the pieces of P_γ, PRE_γ, P_δ and PRE_δ that are left over. This can be done in $O(C(n))$ time. It follows that at some later time we can rebuild P_γ, PRE_γ, P_δ and PRE_δ out of P_α and PRE_α, using the stored information. Only at the root of T we keep the complete P and PRE structures of the entire set (and therefore the answer to the problem). It follows that at each internal node α of T we have the following associated information.

(i) $f(\alpha)$ = a pointer to the father of α (if any).

(ii) $lson(\alpha)$ = a pointer to the leftson of α.

(iii) $rson(\alpha)$ = a pointer to the rightson of α.

(iv) $max(\alpha)$ = the largest value (according to ORD) in the subtree of $lson(\alpha)$.

(v) $P^*(\alpha)$ = the left-over piece of P_α.

(vi) $PRE^*(\alpha)$ = the left-over piece of PRE_α.

(vii) $I(\alpha)$ = the information needed to obtain $P_{lson(\alpha)}$, $PRE_{lson(\alpha)}$, $P_{rson(\alpha)}$ and $PRE_{rson(\alpha)}$ from P_α and PRE_α (i.e., a record of actions).

(i) to (iv) are needed to let T function as a search tree and (v) to (vii) contain the associated information. When α is the root $P^*(\alpha) = P_\alpha$ and $PRE^*(\alpha) = PRE_\alpha$. We will first describe the procedure DOWN to reconstruct the full P_β and PRE_β structures at an arbitrary node β of T. DOWN also reconstructs the full P_β and PRE_β structures at each node γ bordering the search path towards β. These are needed to be able to return from β to the root of the

tree and to reassemble structures when we do so. Later β will be the father of some leaf that needs to be updated and the search will be guided by the usual decision criterion for search trees. We will not spell out this detail in the description of DOWN. DOWN works its way down the tree to β by disassembling structures at the node visited and reassembling the structures at its sons.

```
proc DOWN (α,β);
{α is the node we just reached on our way towards β. P*(α) and PRE*(α)
 contain the full P_α and PRE_α structures.}
    begin
      if α = β
        then goal reached
        else build P_lson(α), PRE_lson(α), P_rson(α) and PRE_rson(α) using
             P*(α), PRE*(α), the left-over pieces at lson(α) and rson(α)
             and the information in I(α);
             P*(lson(α)) := P_lson(α);
             P*(rson(α)) := P_rson(α);
             PRE*(lson(α)) := PRE_lson(α);
             PRE*(rson(α)) := PRE_rson(α);
             {continue the search}
             if β below lson(α)
               then DOWN (lson(α), β)
               else DOWN (rson(α), β)
             fi
      fi
    end of DOWN;
```

The procedure is called as DOWN (root, β). Let the number of points in the set be n.

Lemma 3.1. DOWN reaches its goal after $O(C(n))$ steps when $C(n) = \Omega(n^{\varepsilon})$ for $\varepsilon > 0$ and $O(\log n.C(n))$ steps otherwise.

Proof

The expensive part of DOWN is the construction of $P_{lson(\alpha)}$, $PRE_{lson(\alpha)}$, $P_{rson(\alpha)}$ and $PRE_{rson(\alpha)}$ out of P_α, PRE_α and $I(\alpha)$. At the i^{th} recursive call of DOWN the number of points below α is at most $O(\frac{n}{2^i})$ because T is balanced. So the total cost (except the recursive call) is bounded by $O(C(\frac{n}{2^i}))$. Because β can never lie deeper than $O(\log n)$ the total cost for DOWN is bounded by

$$O(C(n) + C(\tfrac{n}{2}) + C(\tfrac{n}{4}) + \ldots) = O(\sum_{i=0}^{\lfloor \log n \rfloor} C(\tfrac{n}{2^i}))$$

For $C(n) = \Omega(n^{\varepsilon})$ $(\varepsilon > 0)$ this adds up to $O(C(n))$. Otherwise it is bounded by $O(\log n.C(n))$.

□

We will use the procedure DOWN to reach the father of a leaf we want to update. After we have performed the action we want to climb back to the root, meanwhile assembling P and PRE structure. For this task we use the procedure UP. The assembling of structures can be done quite easily using □ and Δ but there is one problem. Because we have updated the tree (by means of an insertion or a deletion) T may have gotten out of balance. To be able to rebalance T efficiently we will restrict our possible choices of T to search trees that can be balanced by means of local rotations (like AVL-trees or BB[α] trees). In the description of UP we delegate this task to a procedure BALANCE. So the procedure UP is the counterpart of DOWN. It starts at β and works its way back to the root, meanwhile assembling structures and performing local rotations when needed.

```
proc UP(α);
{α is the node we most recently reached on our path from β to the root of
 the tree. The P* and PRE* structures at the sons of α contain the complete
 structures.}
    begin
        P*(α) := □ (P*(lson(α)), P*(rson(α)), PRE*(lson(α)), PRE*(rson(α)));
        PRE*(α) := Δ (PRE*(lson(α)), PRE*(rson(α)));
        meanwhile filling in the information in I(α) and leaving the left-over
        pieces of the structures at the sons;
        if out of balance then BALANCE(α) fi;
        if α = root
           then goal reached
           else UP(f(α))
        fi
    end of UP;
```

The procedure is called as UP(f(β)) (when β is not the root).

Lemma 3.2. UP always reaches its goal after $O(C(n) + R)$ steps when $C(n) = \Omega(n^{\varepsilon})$ and $O(\log n.C(n) + R)$ steps otherwise, where R is the total time needed for rebalancing.

Proof

The cost for rebalancing is R by definition. The costs for building $P(\alpha)$ and $PRE(\alpha)$ and collecting the information in $I(\alpha)$ add up to $O(C(n))$ when $C(n) = \Omega(n^{\varepsilon})$ and $O(\log n.C(n))$ otherwise, by the same arguments as in the proof of lemma 3.1.

□

To achieve a bound for R we have to look more carefully at the actions that need to be performed for rebalancing. In AVL-trees and BB[α] trees this rebalancing consists of local rotations only. We consider only the case in which a single rotation is needed, as double rotations can be handled in a similar way. When we have to perform a single rotation at node α (see figure 2) we first do one iteration of DOWN to reconstruct the P and PRE structures

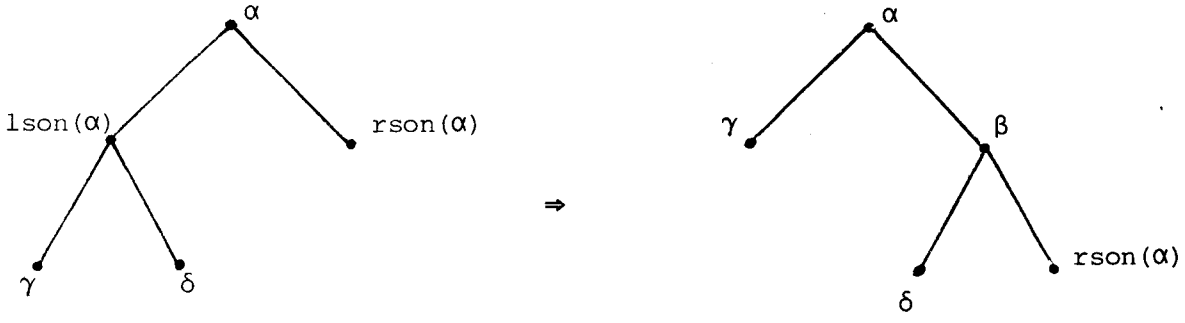


figure 2

at $lson(\alpha)$ and $rson(\alpha)$. When we had noticed the need for rebalancing at the beginning of UP this step was not needed. (This modification is left to the reader.) Another iteration of DOWN is done at $lson(\alpha)$ to obtain the full structures at γ and δ. Next we perform the rotation and start UP again at β.

Lemma 3.3. $R = O(C(n))$ when $C(n) = \Omega(n^{\varepsilon})$ and $R = O(\log n.C(n))$ otherwise.

Proof

A single rotation consists at most 2 iterations of DOWN and 1 iteration of UP. Similar results can be obtained for double rotations. Because at each level there is at most once a need for rebalancing, the bound follows from the bounds for DOWN and UP.

□

We are now ready to prove our main result, namely the dynamization of order decomposable set problems.

Theorem 3.4. Given an order decomposable set problem P, we can dynamize it to yield an insertion time $I(n) = O(C(n))$ and a deletion time $D(n) = O(C(n))$ when $C(n) = \Omega(n^{\varepsilon})$ and $I(n) = D(n) = O(\log n.C(n))$ otherwise, where n is the current number of points in the set.

Proof

We keep the points of the set, ordered with respect to ORD, in a tree structure T as described. In T we have the answer to the problem at the root. When we want to insert (or delete) a point p we use DOWN to reach the father of the leaf where p needs to be inserted (or deleted) according to ORD. After performing the action we climb back to the root using the procedure UP, meanwhile reassembling the updated answers and preconditionings. UP also takes care of the rebalancing operations needed. When we have reached the root of T we have the updated answer to the total set present. The bounds follow from the bounds of DOWN, UP and BALANCE.

□

Hence, once there are "cheap" merging algorithms □ and Δ for a set problem, one can always dynamize it efficiently.

4. Examples.

In this section we will consider a number of typical examples of order decomposable set problems.

a) convex hull in 1, 2 and 3 dimensions.

A first example of an order decomposable set problem is the determination of the convex hull of a set of points in 1-, 2- and 3-dimensional space. In one dimension the convex hull of a set of points consists of the smallest and largest point in the set. One easily verifies that the problem is order decomposable with the usual ordering and with $C(n) = O(1)$ (we need no preconditioning). Hence the dynamization method yields $I(n) = D(n) = O(\log n)$. In the two dimensional case the problem is more difficult. We need some "merging" routine for two in some way separated convex hulls (see figure 3). This merging consists of finding the two common tangents. Overmars and van Leeuwen [15] have given a method to do this in $O(\log n)$ steps when we maintain the convex hulls in

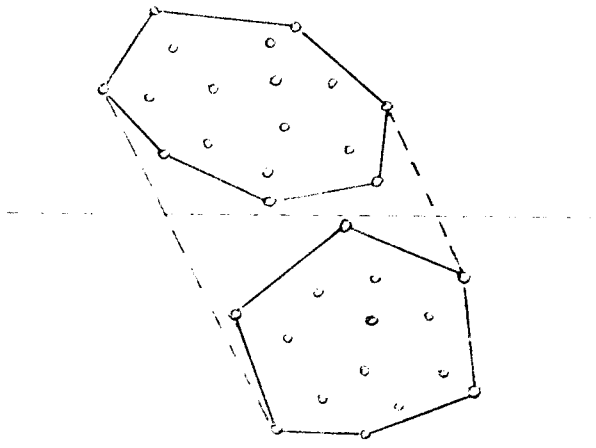


figure 3

some appropriate data structure. As ordering they use the ordering by y-coordinate of the points. (PRE is again empty.) Hence the problem is order decomposable with $C(n) = O(\log n)$ and we can use the scheme of Section 3 to obtain a structure to maintain the two-dimensional convex hull of a set of points with a transaction time of $O(\log^2 n)$. In 3-dimensional space the merging of convex hulls is much harder. Preparata and Hong [17] offer a method for merging separated convex polygons in $O(n)$ time. Thus the 3-dimensional convex hull problem is again order decomposable but with the higher bound of $C(n) = O(n)$, and we can dynamize it yielding an insertion and deletion time of $O(n)$. (Bentley and Shamos [5] state that it is not necessary for the method of Preparata and Hong [17] that the convex polygons are separated. Hence we can choose any ordering we like for ORD.) In d-dimensional space ($d > 3$) no merging algorithms for convex polygons are known. Hence we cannot give dynamizations for the d-dimensional convex hull problem.

b) intersection of halfspaces in 1, 2 and 3 dimensions.

A somewhat related problem asks for the intersection of a set of halfspaces. In 1-dimensional space it can easily be shown that the problem is order decomposable with $C(n) = O(1)$ and hence we can achieve $O(\log n)$ insertion and deletion time. In the two dimensional case the common intersection of a set of halfspaces consists of a (possibly open) convex polygonal region. Instead of maintaining the whole set we split the set in the set of halfspaces "looking" leftwards and the set of halfspaces looking rightwards (the l-halfspaces and r-halfspaces respectively). Overmars and van Leeuwen [15] have shown that the problem of finding the intersection of l-halfspaces (and likewise r-halfspaces) is decomposable with $C(n) = O(\log n)$. As ordering they use the angle of the bordering line (see figure 4). In this case, finding the total intersection from intersections of two halves consist of finding the one point of inter-

section of the two convex arcs. It follows that we can maintain the intersection of a set of l-halfspaces in $O(\log^2 n)$ transaction time (likewise for

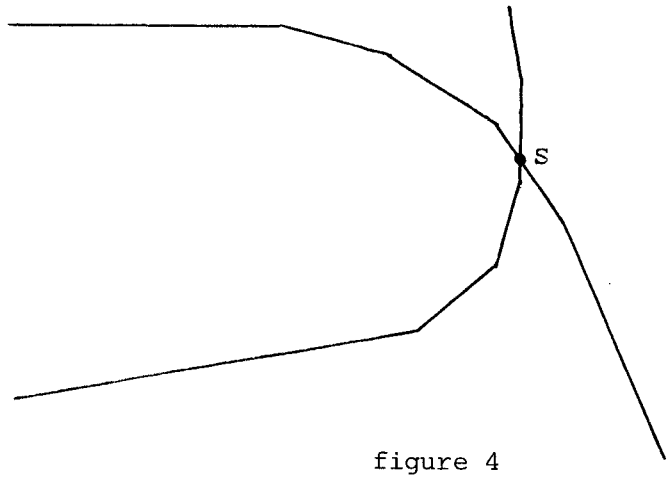


figure 4

r-halfspaces) Overmars and van Leeuwen [12] show that we can find the intersection of the total set out of the intersections of the l-halfspaces and r-halfspaces in only $O(\log^2 n)$ additional steps. Hence we can maintain the total intersection at a cost of $O(\log^2 n)$ per transaction. In three dimensions the problem is again more complex. Brown [6, 7] and Preparata and Muller [18] have shown that there is a strong connection between the intersection of a set of halfspaces and the convex hull of a set of points (by means of dualization). Using these results one can develop an O(n) maintenance algorithm for the intersection of halfspaces in 3-dimensional space.

c) maximal elements in 1, 2 and higher dimensions.

Given a set of points we can ask for the maximal elements in the set. An element x is called maximal if there is no y in the set such that $y > x$. In one dimension the problem is trivial (yielding O(log n) bounds). In the two-dimensional case Overmars and van Leeuwen [12] have given a method for merging the contours of maximal elements of horizontally separated subsets in O(log n) time (see figure 5). Hence the problem is order decomposable and

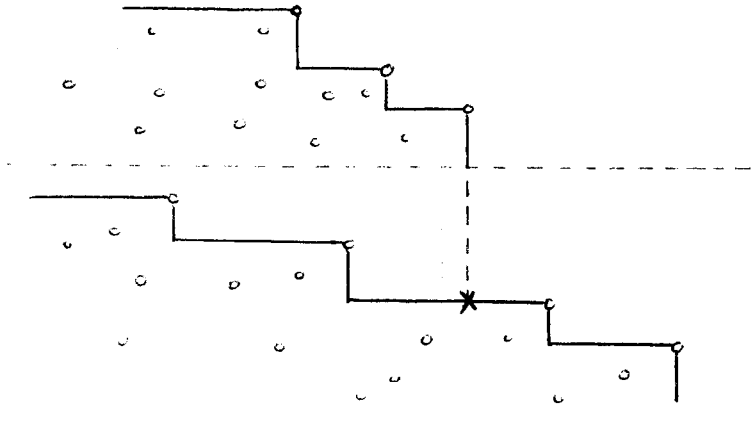


figure 5

can be dynamized yielding $O(\log^2 n)$ bounds. In the higher dimensional case we can use a trivial dynamization of the ECDF searching problem (see Bentley and Shamos [4]) to achieve O(n) bounds for the problem.

d) union and intersection of line segments.

Consider the following problem: Given a set of segments on a line, determine the union. Clearly the union again consists of a set of segments (see figure 6).

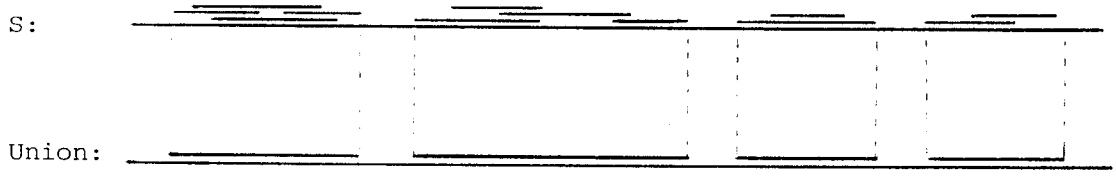


figure 6

The set problem is decomposable in the following sense:

Theorem 4.1. Let $S = \{s_1, \ldots, s_n\}$ be a set of segments ordered by starting point (= left point). If the union of $\{s_1, \ldots, s_i\}$ and of $\{s_{i+1}, \ldots, s_n\}$ is known we can determine the union of the entire set in O(log n).

Proof

Let p be a point between the starting point of s_i and s_{i+1}. The union of $\{s_1, \ldots, s_i\}$ consists of a number of segments but at most one of these segments can stretch to the right of p (see figure 7). When there is no such segment, the union of the total set consists of the unions of the two parts. Otherwise,

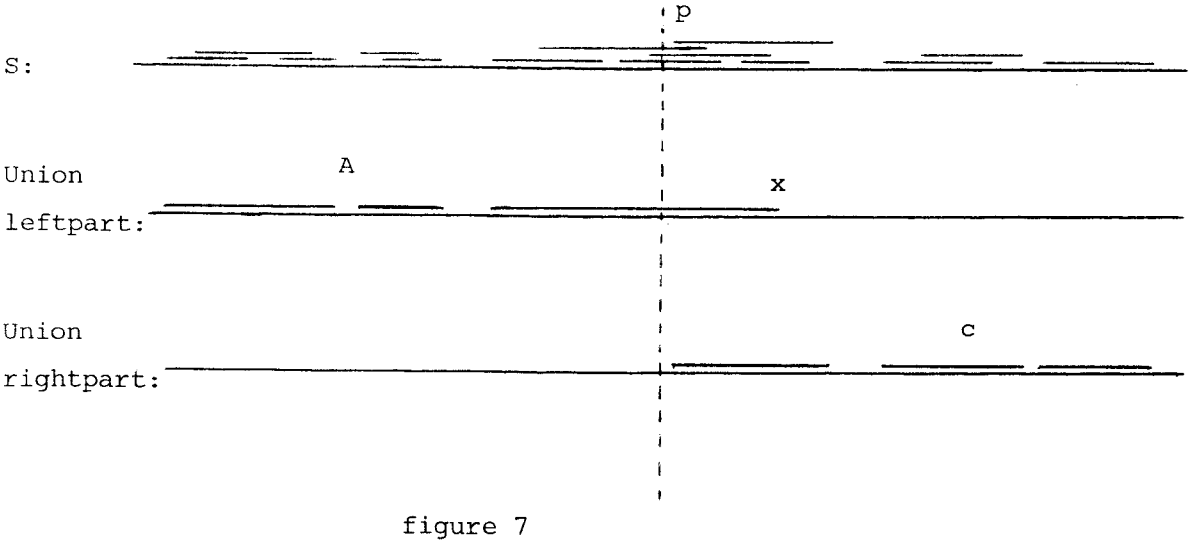
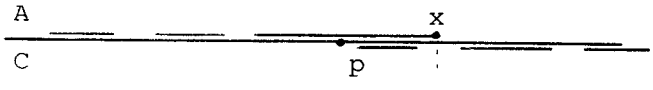


figure 7

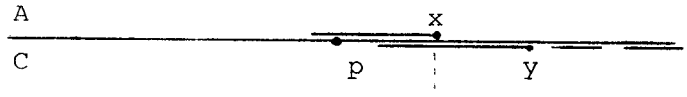
let x be the rightmost end point of that segment. Let the union of $\{s_1, \ldots, s_i\}$ be A and the union of $\{s_{i+1}, \ldots, s_n\}$ be C. We search for x in C. The following two cases can occur

Case a: x lies between two segments in C (or before or behind C).



In this case we just throw the part of C left from x away. The total union consists of A plus the rest of C.

Case b: x lies within a segment in C.



In this case we extend the last segment of A up to y and throw away the part of C up to y.

It follows that, when we use a proper datastructure for A and C, we can find the union in O(log n) steps.

□

It follows that the problem is decomposable with C(n) = O(log n). Hence we can dynamize it by the methods described yielding I(n) = D(n) = $O(\log^2 n)$. Extending this result to higher dimensional space give rise to tremendous problems.

To determine (and maintain) the intersection of a set of line segments is easier. Such an intersection consists of at most one segment.

Theorem 4.2. Given a set $S = \{s_1, \ldots, s_n\}$ of segments and the intersection of $\{s_1, \ldots, s_i\}$ and of $\{s_{i+1}, \ldots, s_n\}$, the intersection of S can be determined in O(1) steps.

Proof

The intersection of $\{s_1, \ldots, s_i\}$ consists of (at most) one segment, as does the intersection of $\{s_{i+1}, \ldots, s_n\}$. To find the total intersection we just have to intersect these two segments, which takes O(1).

□

Hence, the problem is decomposable with C(n) = O(1) and we can dynamize it in I(n) = D(n) = O(log n). To extend the result to d-dimensional space, just replace "segment" by "rectilinearly oriented hyperrectangle" and everything carries

over. (More precisely $C(n) = O(2^d)$ and $I(n) = D(n) = O(2^d \log n)$.)

e) closest points problem.

The closest points problem we consider asks for two points in a set that have smallest distance. For this problem there exists an algorithm based on a divide-and-conquer strategy (Bentley and Shamos [3] and Bentley [2]). It is a good example of how the PRE-structure can be used. To find the closest pair of points we split the set by a vertical line into two halves and determine the closest pair in each half. Let δ be the minimum of the two distances. The only points that can lie closer than δ must lie in the slab of size 2δ around the dividing line (see figure 8). When we have the points in these slab ordered

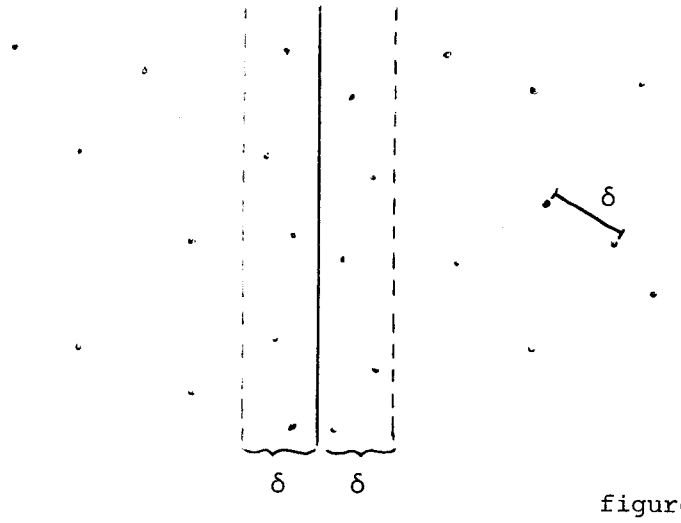


figure 8

by y-coordinate we can determine the closest pair among them by one walk along them. It can be shown (see [2] and [3]) that with each point in the slab we have to do only a constant number of comparisions with other points. Hence, the conquer step of the algorithm takes at most $O(n)$, provided we have the points ordered by y-coordinate. This is where our PRE-structure comes in. In this structures we keep all the points ordered by y-coordinate. It follows that Δ takes at most $O(n)$. Sifting the points in the slab from the PRE-structure can also be done in $O(n)$. It follows that $C(n) = O(n)$. Hence we can dynamize the problem obtaining $I(n) = D(n) = O(n)$.

f) the Voronoi diagram.

A structure that has received much attention in the last few years is the Voronoi diagram. Shamos [20] and Shamos and Hoey [21] first showed how the Voronoi diagram could be used to answer a number of closest point problems. In a Voronoi diagram we keep with each point of the set the region of points that lie nearest to the particular point (see figure 9). Each region is a convex

polygon. To build the Voronoi diagram of a set of points, Shamos [20] divides

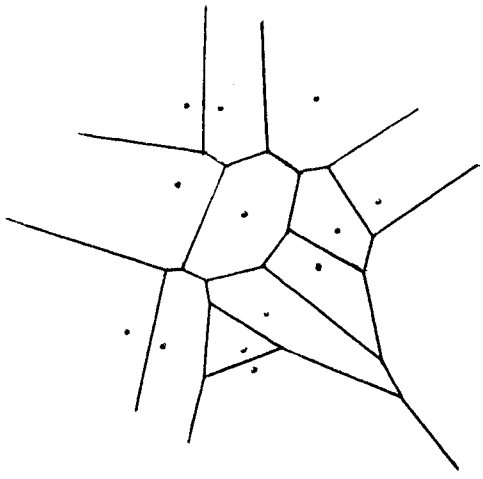


figure 9

the set by a vertical line into two equal size subsets, builds the Voronoi diagrams of both halves and then "concatenates" them. The merging of the two diagrams takes O(n). (Kirkpatrick [8] has shown that the separation is not necessary.) It follows that the problem of finding the Voronoi diagram of a set of points is order decomposable with C(n) = O(n). Hence we can dynamize it yielding I(n) = D(n) = O(n). There are many problems that can be solved in linear time using the Voronoi diagram (see Shamos and Hoey [21]) which can thus be dynamized in linear time, including the all closest points problem, the maintenance of a minimum-weight triangulation, the largest empty circle problem and the maintenance of a minimum spanning tree.

Acknowledgement.

I would like to thank Jan van Leeuwen for helpful discussions.

References.

[1] Bentley, J.L., Decomposable Searching Problems. Inform. Proc. Lett. 8 (1979) pp. 244-251.

[2] Bentley, J.L., Multidimensional Divide-and-Conquer, Comm. of the ACM 23 (1980) pp. 214-229.

[3] Bentley, J.L. and M.I. Shamos, Divide-and-Conquer in Multidimensional Space, Proc. of the 8th Annual ACM Symp. on Theory of Computing, 1976, pp. 220-230.

[4] Bentley, J.L. and M.I. Shamos, A Problem in Multivariate Statistics: Algorithm, Data structure and Applications, Proc. of the 15th Annual Allerton Conference on Commercial, Control and Computing, 1977, pp. 193-201.

[5] Bentley, J.L. and M.I. Shamos, Divide and Conquer for Linear expected time, Inform. Proc. Lett. 7 (1978) pp. 87-91.

[6] Brown, K.Q., Fast intersection of Half Spaces, Techn. Rep. CMU-CS-78-129, Dept. of Computer Science, Carnegie-Mellon University, 1978.

[7] Brown, K.Q., Geometric Transforms for Fast Geometric Algorithms, Techn. Rep. CMU-CS-80-101, Dept. of Computer Science, Carnegie-Mellon University, 1980.

[8] Kirkpatrick, D.G., Efficient Computation of Continuous Skeletons, Proc. of the 20th Annual IEEE Symp. on Foundations of Computer Science, 1979, pp. 18-27.

[9] Maurer, H.A. and Th. Ottmann, Dynamic Solutions of Decomposable Searching Problems, in: Pape, U.(ed.), Discrete Structures and Algorithms, Carl Hanser, 1979, pp. 17-24.

[10] Mehlhorn, K. and M.H. Overmars, Optimal Dynamization of decomposable Searching Problems (in preparation).

[11] Overmars, M.H. and J. van Leeuwen, Two general methods for Dynamizing Decomposable Searching Problems, Techn. Rep. RUU-CS-79-10, Dept. of Computer Science, University of Utrecht, 1979. (To appear in Computing.)

[12] Overmars, M.H. and J. van Leeuwen, Maintenance of Configurations in the Plane, Techn. Rep. RUU-CS-79-9, Dept. of Computer Science, University of Utrecht, 1979/1980.

[13] Overmars, M.H. and J. van Leeuwen, Some Principles for Dynamizing Decomposable Searching Problems, Techn. Rep. RUU-CS-80-1, Dept. of Computer Science, University of Utrecht, 1980. (To appear in Inform. Proc. Lett.)

[14] Overmars, M.H. and J. van Leeuwen, Dynamically Maintaining Configurations in the plane, Proc. of the 12th Annual ACM Symp. on Theory of Computing, 1980, pp. 135-145.

[15] Overmars, M.H. and J. van Leeuwen, Notes on Maintenance of Configurations in the Plane, Techn. Rep. RUU-CS-80-5, Dept. of Computer Science, University of Utrecht, 1980.

[16] Overmars, M.H. and J. van Leeuwen, Dynamizations of Decomposable Searching Problems yielding Good Worst-Case Bounds, Techn. Rep. RUU-CS-80-6, Dept. of Computer Science, University of Utrecht, 1980.

[17] Preparata, F.P. and S.J. Hong, Convex Hulls of Finite Sets of Points in Two and Three Dimensions, Comm. of the ACM 20 (1977) pp. 87-93.

[18] Preparata, F.P. and D.E. Muller, Finding the Intersection of n Half-spaces in Time O(n log n), Theoretical Computer Science 8 (1979) pp 45-55.

[19] Saxe, J.B. and J.L. Bentley, Transforming Static Data Structures into Dynamic Structures, Proc. of the 20th IEEE Symp. on Foundations of Computer Science, 1979, pp. 148-168.

[20] Shamos, M.I., Computational Geometry, to be published by Springer-Verlag.

[21] Shamos, M.I. and D. Hoey, Closest-Point Problems, Proc. of the 16th Annual IEEE Symp. on Foundations of Computer Science, 1976, pp. 151-162.

[22] van Leeuwen, J. and H.A. Maurer, Dynamic Systems of Static Data-Structures, Report 42, Inst. für Informationsverarbeitung, Technical University Graz, 1980.

[23] van Leeuwen, J. and D. Wood, Dynamization of Decomposable Searching Problems, Inform. Proc. Lett. 10 (1980) pp. 51-56.