BATCHED DYNAMIC SOLUTIONS TO
DECOMPOSABLE SEARCHING PROBLEMS

Herbert Edelsbrunner
Mark H. Overmars

RUU-CS-83-8
March 1983

Rijksuniversiteit Utrecht

Vakgroep informatica

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

vakgroep informatica R.U. Utrecht

BATCHED DYNAMIC SOLUTIONS TO DECOMPOSABLE SEARCHING PROBLEMS

Herbert Edelsbrunner

Mark H. Overmars

Technical Report RUU-CS-83-8

March 1983

Department of Computer Science

University of Utrecht

P.O. Box 80.002

3508 TA Utrecht

the Netherlands

BATCHED DYNAMIC SOLUTIONS TO DECOMPOSABLE SEARCHING PROBLEMS*

Herbert Edelsbrunner[(1)] and Mark H. Overmars[(2)]

Abstract. The batched static version of a searching problem asks for performing a given set of queries on a given set of objects. All queries are known in advance. The batched dynamic version of a searching problem is the following: given a sequence of insertions, deletions and queries, perform them on an initially empty set. We will develop methods for solving batched static and batched dynamic versions of searching problems which are in particular applicable to decomposable searching problems. The techniques show that batched static (dynamic) versions of searching problems can often be solved more efficiently than by using known static (dynamic) data structures. In particular, a technique called "streaming" is described that reduces the space requirements considerably. The methods have also a number of applications on set problems. E.g., the k intersecting pairs in a set of n orthogonal hyper-rectangles in d dimensions can be reported in $O(n \log^{d-1} n + k)$ time and only $O(n)$ space.

Keywords and phrases. Decomposable searching problems, batched static, batched dynamic, streaming, concrete complexity.

----

(1) Institutes for Information Processing, Technical University of Graz, Schiesstattgasse 4^a, A-8010 Graz, Austria.

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

* Research reported in this paper was done while the second author visited the Technical University of Graz. The first author was supported by the Austrian Fonds zur Foerderung der Wissenschaftlichen Forschung. The second author was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).

1. Introduction.

In the past few years, a lot of research has been done on solving searching problems. A searching problem is a problem in which a question (query) is asked about a (query)object x with respect to a set of objects (often called points) V. A wellknown example is the member searching problem in which we ask whether the query object x is in the set V. Numerous searching problem arise particularly in the area of "Computational geometry" that deals with problems about sets of points, lines, etc. in multi-dimensional space. A prime example is the range searching problem: given a set of points V in d-dimensional space and a orthogonal hyper-rectangle (i.e., a d-dimensional rectangle with sides parallel to the coordinate-axes) called a range, report all points in V that lie within the range.

There are a number of different ways of solving searching problems. A static solution to a searching problem consists of a data structure to store the set of points (together with an algorithm to construct such a data structure) such that queries with different query objects can be answered. The efficiency of such a data structure is measured by three quantitees: the amount of time needed for building the structure, the amount of time required for performing a query, and the amount of storage required for storing the structure. A dynamic solution to a searching problem consists of a data structure that allows for queries, insertions of new objects in the set, and deletions of existing objects. The efficiency of a dynamic data structure is measured by the query time, the amount of time required for performing an insertion or deletion, and the amount of storage required. Recently, plenty of work has been devoted to the design of general methods for turning static solutions to searching problems into dynamic solutions by means of general techniques. (See Overmars [15] for an overview of the methods used.)

In this paper we will consider two different ways of solving searching problems that appear in particular in an off-line (i.e., batched) environment. The batched static version of a searching problem is the following: given a set of points V and a set of n_q query objects $x_1,\dots,x_{n_q}$, perform these queries on V, i.e., for each x_i compute the answer to the query with x_i. Hence, we are not interested in keeping low the time required for individual queries but in minimizing the overall runtime. We are not bound to a specific order of the queries nor is there a need to perform them one after the other. Starting with V and the set of queries, we are only interested in obtaining the set of answers in its totality. Clearly, the batched static version of a searching problem can be solved using a static data structure for the problem, but in a number of cases one can do better. The batched dynamic version of a searching problem is the following: given a sequence of insertions, deletions and queries, report all answers to the queries when the sequence of actions is performed (in the given order) on an initially empty set. We are again only interested in the overall runtime. There is no need for actually performing the updates and queries in the given order, as long as it is made certain that queries are performed for the proper sets of points. Clearly, a dynamic data structure can be used for solving the problem, but often one can do better. The study of batched static and batched dynamic versions of searching problems is interesting not only for their use in an off-line environment. There are also numerous problems that can be formulated as batched (static or dynamic) versions of searching problems. A frequently considered example is the planar rectangle intersection problem that asks for all intersecting pairs in a set of orthogonal rectangles in the plane. The intersecting pairs can be determined by solving the batched dynamic version of the interval intersection searching problem (the problem that asks for all intervals in a set which intersect a query interval).

We will give a number of general techniques which can be used for solving batched (static or dynamic) versions of searching problems. They are in particular applicable to decomposable searching problems. Let PR be a searching problem then we denote by $PR(x,V)$ the answer for PR with respect to a set V and a query object x.

Definition. A searching problem PR is called decomposable if for any partition $A \cup B$ of the set V and for each query object x

$$PR(x,V) = \square(PR(x,A),PR(x,B))$$

for some constant time computable operator $\square$.

For example, the member searching problem is decomposable. When we know whether x is in A and whether x is in B we can compute in $O(1)$ time whether x is in $V = A \cup B$ using the or-function for $\square$. Numerous other searching problems are decomposable as well. The class of decomposable searching problems was introduced by Bentley [1] who showed how static data structures for decomposable searching problems can be turned into efficient dynamic data structures by applying a general method. His work was generalized in a number of ways, resulting in a very general and, in some sense, optimal dynamization method for decomposable searching problems (see Overmars and van Leeuwen [16]). For decomposable searching problems there is no need for storing the set of objects in one massive data structure. One can split the set in a number of disjoint subsets and build a data structure for each of the subsets. The answer to a query for the total set can be derived from the answers to the same query for the subsets using the composition operator $\square$.

In Section 2, we consider batched static solutions to searching problems. Some examples of batched static solutions to searching problems that are better than known static solutions will be given. In particular the plane-sweep technique (see e.g. Shamos and Hoey [18] or Bentley and Wood [3]) will turn out to be a powerful technique for solving batched static versions of a number of searching problems. This will be shown by applying the technique to a problem posed by McCreight [12]. Next, a general method is described for solving the batched static version of decomposable searching problems for which only structures exist with a large discrepancy between query time and preprocessing time. We will show that such data structures can be transformed into structures in which the query- and building time are more "balanced" and that have much better perspectives for use in a batched environment.

In Section 3, we give a general method for solving the batched dynamic version of decomposable searching problems for which static data structures exist. One of the applications shows that the batched dynamic version of the nearest neighbor searching problem (that asks for a point in a set of points in the plane that is nearest to a given query point) can be solved in $O(n \log^2 n)$ time, where n is the length of the sequence of updates and queries.

In Section 4, we show how a technique, called "streaming", can be used for reducing the space requirements for batched (static and dynamic) versions of searching problems. As a consequence, the batched static version of the d-dimensional rectangle intersection searching problem can be solved in $O(n \log^{d-1} n)$ time using $O(n)$ storage.

Section 5 shows how better results for the batched static version of a number of searching problems can be obtained by changing the query objects into set objects and the set objects into query objects and solving a kind of "reversed" problem.

In Section 6 and 7 we give some extensions, conclusions and directions for further research.

Throughout this paper we use the following notations:

(i) for a data structure S:

n = the number of points in the structure,

$Q_S(n)$ = the amount of time required for performing a query on S,

$P_S(n)$ = the amount of time required for building (preprocessing) S,

$U_S(n)$ = the amount of time required to perform an update (insertion or deletion) on S,

$M_S(n)$ = the amount of storage (memory) required for S.

(ii) in the batched static case:

n_s = the number of set objects,

n_q = the number of queries,

$n = n_s + n_q$,

$P^s(n)$ = the amount of time required for solving the batched static version of a searching problem (where n_s and n_q are implicitly understood),

$M^s(n)$ = the amount of storage required for solving the batched static version of a searching problem.

(iii) in the batched dynamic case:

N = the number of updates,

n_q = the number of queries,

$n = N + n_q$ (i.e., the length of the sequence of actions),

m = the maximum number of points once present in the set,

$P^d(n)$ = the amount of time required for solving the batched dynamic version of a searching problem,

$M^d(n)$ = the amount of storage required for solving the batched dynamic version of a searching problem.

We say that a function $F(n)$ is in $O(f(n))$ if there exists a positive constant c such that $F(n) \leq c.f(n)$, for all n. All bounds $F(n)$ except for $Q_S(n)$ are assumed to be at least linear, that is, $n = O(F(n))$. Furthermore, all bounds $F(n)$ state the requirements in worst-case and are assumed to be smooth, that is, $F(\theta(n)) = \theta(F(n))$, where a function $f(n)$ is in $\theta(F(n))$ if $f(n) = O(F(n))$ and $F(n) = O(f(n))$.

2. Batched static solutions.

Given a static data structure S for solving a searching problem PR, one can solve the batched static version of PR by, first, storing all points of the set in an instance of S and, next, performing all queries on the structure. This leads to a solution for the batched static version of PR with

$$P^S(n) = O(P_S(n_s) + n_q \cdot Q_S(n_s)),$$
$$M^S(n) = O(M_S(n_s)).$$

However, one can do better for a number of searching problems. Consider for example the two-dimensional <u>rectangle containment problem</u>: given a set V of orthogonal rectangles in the plane and another such rectangle x, report all rectangles in V that are contained in x. The best known static solution to the problem yields a query time of $O(\log^3 n + k)$, where k denotes the number of reported answers, a building time of $O(n \log^3 n)$ and uses $O(n \log^3 n)$ storage (see Lee and Wong [10] or Edelsbrunner and Overmars [7]). Using this data structure for solving the batched static version of the rectangle containment searching problem yields

$$P^S(n) = O((n_s+n_q)\log^3 n_s + k'),$$
$$M^S(n) = O(n_s \log^3 n_s),$$

where k' denotes the total number of reported answers. Lee and Preparata [9] have shown that the problem that asks for all pairs of rectangles (r_1,r_2) in a set V such that r_1 is contained in r_2, can be solved within time $O(n \log^2 n + k)$ using only $O(n)$ storage ($n=|V|$). Their solution can easily be adapted to solving the problem of reporting all pairs of rectangles (r_1,r_2) such that r_1 is contained in r_2 where r_1 is in a set V_1 and r_2 is in another set V_2. Choosing V_1 to be the set of rectangles and V_2 to be the set of query rectangles, this method solves the batched static version of the two-dimensional rectangle containment searching problem with

$$P^S(n) = O((n_s+n_q)\log^2(n_s+n_q) + k') = O(n \log^2 n + k'),$$
$$M^S(n) = O(n_s+n_q) = O(n).$$

A technique that is often useful for solving the batched static version of two-dimensional searching problems (i.e., searching problems dealing with objects in the plane) is the so-called "plane sweep" technique (see e.g. Shamos and Hoey [18], Bentley and Wood [3] or Nievergelt and Preparata [13]). As an example, consider the following searching problem (proposed by McCreight [12]): given a set V of vertical line segments in the plane and a query point x, determine the leftmost segment s in V we encounter when moving x horizontally to the right (see fig. 1). s is called the <u>(immediate)</u> <u>obstacle</u> of x. The batched static version of

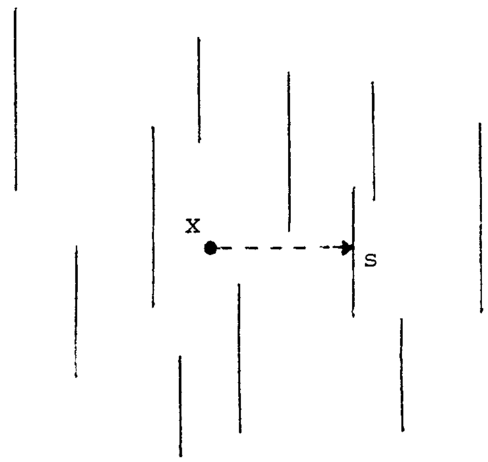

figure 1.

the problem is the following: given a set V_1 of n_s vertical line segments and a set V_2 of n_q points, determine for each point x in V_2 the immediate obstacle in V_1. To solve the problem, a vertical scan line l is moved from left to right over the plane. With l we keep a balanced search tree T that contains all points we have already passed but for which we did not pass the corresponding obstacle yet. T stores the points ordered with respect to the y-coordinate (i.e., in vertical direction). See figure 2 for an illustration. When we pass a point we insert it into the tree. When we pass a line segment $s = [y_1 : y_2]$ (where $y_1 \leq y_2$ are

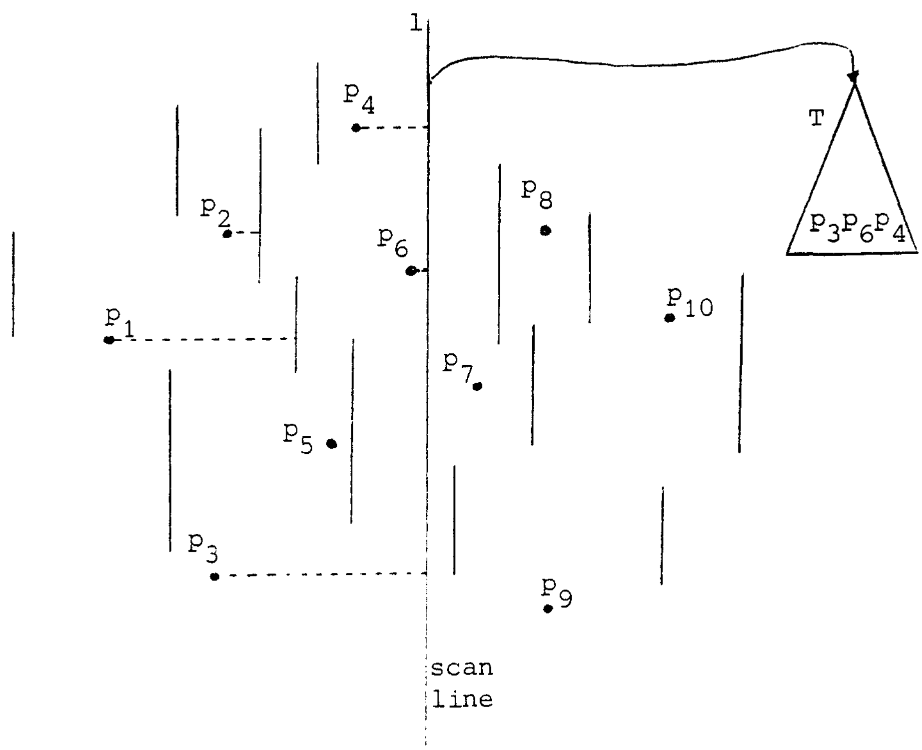

figure 2.

the y-coordinates of the endpoints) we search for all points in T with y-coordinate in $[y_1:y_2]$. These are exactly the query points that have s as obstacle. Since T contains the points sorted in vertical direction, $O((k+1)\log n_q)$ time suffices to determine the k points whose y-coordinates are in $[y_1:y_2]$, and to delete them from T. To be able to locate the next point or line segment the scan line passes, we have to sort both sets V_1 and V_2 with respect to x-coordinate. This takes $O(n_s \log n_s + n_q \log n_q)$ time. Each point in V_2 is once inserted into T and at most once deleted. As the size of T is bounded by n_q, the total amount of time required for inserting and deleting points is bounded by $O(n_q \log n_q)$. For each segment in V_1 we have to perform a query which costs $O(\log n_q)$ time per segment and per point determined. As each point in V_2 is found at most once, the time for queries is bounded by $O((n_s+n_q)\log n_q)$. One easily verifies that the amount of storage is bounded by $O(n_s+n_q)$. Hence, we have

$$P^S(n) = O((n_s+n_q)\log n_q + n_s \log n_s) = O(n \log n),$$
$$M^S(n) = O(n_s+n_q) = O(n).$$

Batched static versions of numerous other (two-dimensional) searching problems can also be solved using the plane-sweep technique.

When a static data structure S is used for solving the batched static version of a searching problem, it is important that the time needed for queries and the time needed for building the structure are "in balance", that is, $P_S(n_s)$ is about the same as $n_q Q_S(n_s)$. For a number of static data structures proposed in the literature, $P_S(n_s)$ is very large compared with $Q_S(n_s)$. As n_q is often $\theta(n_s)$, we had better look for static solutions with better trade-offs, i.e., with a lower building time and a higher query time. For decomposable searching problems this can be achieved by applying a general transformation as we will show. The technique is adapted from Maurer and Ottmann [11].

For decomposable searching problems, the answer to a query for the total set can be computed in $O(n_s)$ time from the answers to the queries for the individual elements. Computing the answer to a query for one element clearly takes constant time (assuming that the problem is solvable).

It follows that the batched static version of a decomposable searching problem can be solved with

$$P^S(n) = O(n_q.n_s),$$
$$M^S(n) = O(n_s).$$

In this way we reduce the building time to $O(n_s)$ (just to store the points of the set) at an increase of the query time to $O(n_s)$. To obtain other trade-offs between query and building time of a static data structure we split the set in a number of almost equally sized disjoint subsets and build a static data structure for each subset. To perform a query all subsets are queried separately and the answers are combined using the composition operator $\square$ for the decomposable searching problem at hand.

<u>Theorem</u> 2.1. Given a static data structure S for a decomposable searching problem PR and some integer function f(n), with $1 \leq f(n) \leq n$, there exists a static data structure S' for PR with

$$Q_{S'}(n) = O(f(n).Q_S(n/f(n))),$$
$$P_{S'}(n) = O(f(n).P_S(n/f(n))),$$
$$M_{S'}(n) = O(f(n).M_S(n/f(n))).$$

<u>Proof</u>

Split the set V in f(n) subsets of size at most $\lceil n/f(n) \rceil$ and build an instance of S for each subset. The bounds follow trivially.

$\square$

<u>Corollary</u> 2.2. Given a static data structure S for a decomposable searching problem PR and some integer function f(n), with $1 \leq f(n) \leq n$, the batched static version of PR can be solved with

$$P^S(n) = O(f(n_s).P_S(n_s/f(n_s)) + n_q.f(n_s).Q_S(n_s/f(n_s))),$$
$$M^S(n) = O(f(n_s).M_S(n_s/f(n_s))).$$

Let us consider some applications of the presented technique. We will assume that $n = \theta(n_q) = \theta(n_s)$.

Applications.

a) Fixed radius near neighbor searching.

The fixed radius near neighbor searching problem asks for all elements of a set of points in the plane that lie within some fixed distance ε from a given query point. Bentley and Maurer [2] describe a static solution S for solving the problem with

$$\begin{aligned} Q_S(n) &= O(\log n + k) \\ P_S(n) &= O(n^3), \\ M_S(n) &= O(n^3), \end{aligned}$$

where k denotes the number of reported answers. (Preparata [17] states $O(n^2 \log n)$ bounds on the preprocessing time and the amount of storage required but his bounds are not quite correct as he does not count time and storage required for storing partial answers.) The problem is clearly decomposable. Hence, we can apply Theorem 2.1. and, choosing $f(n) = \lceil n^{2/3}/\log^{1/3} n \rceil$, obtain a data structure S' such that

$$\begin{aligned} Q_{S'}(n) &= O(n^{2/3} \log^{2/3} n + k), \\ P_{S'}(n) &= O(n^{5/3} \log^{2/3} n), \\ M_{S'}(n) &= O(n^{5/3} \log^{2/3} n). \end{aligned}$$

Using this structure we obtain a solution to the batched static version of the fixed radius near neighbor searching problem with

$$\begin{aligned} P^S(n) &= O(n^{5/3} \log^{2/3} n + k'), \\ M^S(n) &= O(n^{5/3} \log^{2/3} n), \end{aligned}$$

where k' denotes the total number of reported answers.

b) Half-planar range counting.

The half-planar range counting problem asks for the number of elements of a set of points in the plane that lie below (or above) a given query line. The problem can be solved using a slight modification of a structure of Edelsbrunner, Kirkpatrick and Maurer [5] such that

$$\begin{aligned} Q_S(n) &= O(\log n + k), \\ P_S(n) &= O(n^2 \log n), \\ M_S(n) &= O(n^2). \end{aligned}$$

One easily verifies that the problem is decomposable. Applying Corollary 2.2. with $f(n) = \lceil \sqrt{n} \rceil$, we obtain a batched static solution to the problem with

$$P^S(n) = O(n\sqrt{n} \log n + k'),$$
$$M^S(n) = O(n\sqrt{n}).$$

Theorem 2.1. (Corollary 2.2.) can also be applied to data structures for numerous other decomposable searching problems like e.g. on structures for polygon retrieval [5,20], line segment intersection searching [5] and polygonal intersection searching [5]. For all these problems batched static solutions can be obtained with $P^S(n) = o(n^2)$.

3. A general batched dynamic solution.

Once we have a fully dynamic data structure S for a searching problem PR, the batched dynamic version of PR can be solved by performing the sequence of insertions, deletions, and queries in the right order on an initially empty instance of S. This clearly leads to a solution for the problem with

$$P^d(n) = O(n_q.Q_S(m) + N.U_S(m)),$$
$$M^d(n) = O(M_S(m)).$$

From now on, we assume that only a static data structure S is available for solving PR and that PR is a decomposable searching problem. We will show that in this case an efficient solution to the batched dynamic version of PR exists as well. In many cases, this solution is significantly more efficient than the one obtained by using a dynamic data structure. In addition, actual implementations of dynamic data structures tend to be rather involved while the solution to be described is less complicated.

An instance of the batched dynamic version of a searching problem consists of a sequence of n actions $act_1, \ldots, act_n$ where each action act_i is either an insertion of a set object (that is assumed to be not yet present in the set), a deletion of a set object (that is assumed to be present) or a query with a query object. For an action act_i we call i the moment at which the action is performed. For each point p that ever belongs to the set there is a moment mi_p at which p is inserted and possibly another moment md_p at which it is deleted (i.e., act_{mi_p} is the insertion of p and act_{md_p} is the deletion of p). When a point is reinserted at some later moment we treat it as a separate point. When p is not deleted we take $md_p = n+1$. Hence with each set object p we can associate an interval $[mi_p:md_p]$ during which p is present. We call this interval the <u>existence interval</u> of p. As a running example consider the following sequence of actions (where $INS(p_i)$ denotes the insertion of p_i, $DEL(p_i)$ denotes the deletion of p_i and $QRY(x_i)$ denotes a query with object x_i):

$act_1 = INS(p_1)$	$act_7 = INS(p_4)$
$act_2 = INS(p_2)$	$act_8 = DEL(p_1)$
$act_3 = INS(p_3)$	$act_9 = QRY(x_3)$
$act_4 = QRY(x_1)$	$act_{10} = INS(p_5)$
$act_5 = DEL(p_2)$	$act_{11} = QRY(x_4)$
$act_6 = QRY(x_2)$	

Figure 3 shows the existence intervals of the points $p_1,\dots,p_5$.

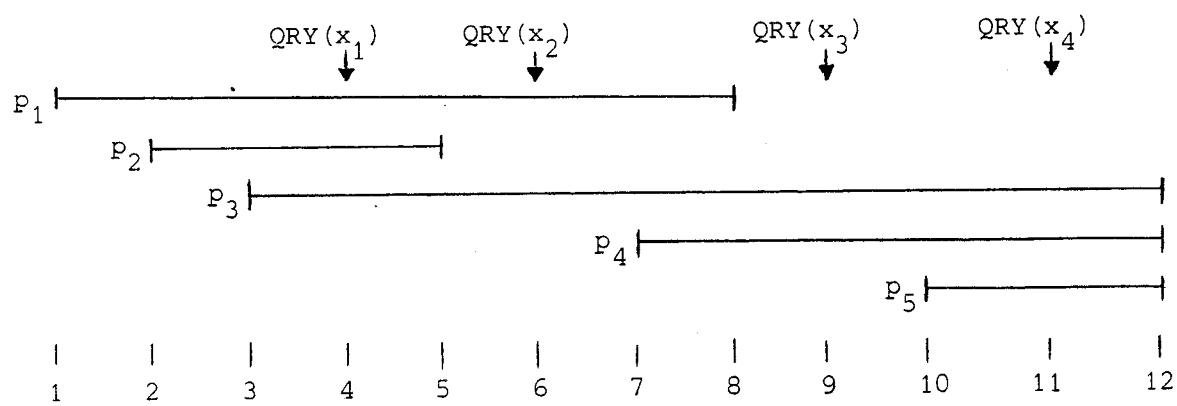

figure 3.

When we perform a query at some moment i (i.e., act_i is a query) we perform it for those points that are present in the set at moment i, i.e., those points that are inserted before moment i and deleted after moment i. These are the points whose existence intervals contain i. For example, the query with x_3 (action act_9) in our example has to be performed on the points p_3 and p_4. Thus, before performing the actual query act_i, we determine the points that are present at moment i, i.e., whose existence intervals contain i. This (pre-)query is called a <u>point enclosure query</u>. This problem can be solved using a segment tree. We will shortly describe this structure. For details see e.g. Bentley and Wood [3] or van Leeuwen and Wood [19].

To store the existence intervals, the total time interval $[1:n+1]$ is divided into so-called <u>atomic segments</u> that are the largest segments that do not contain a begin or endpoint of an existence interval in their interior. In other words, the atomic segments are the segments between consecutive begin and/or endpoints. In our example, the atomic segments are [1:2], [2:3], [3:5], [5:7], [7:8], [8:10] and [10:12]. These atomic segments correspond to the leaves of a balanced binary tree, such that the leaf of an atomic segment s_1 is to the left of the leaf of an atomic segment s_2, if and only if s_1 is to the left of s_2. Each internal node α corresponds to the total segment spanned by the leaves in the subtree rooted at α. In an ordinary segment tree each node α is associated with the list of all intervals that cover the segment corresponding to α but do not cover the segment corresponding to the father of α. In our application we store with a node α the points whose existence intervals cover the segment corresponding to α but do not cover the segment corresponding to the father of α. See figure 4 for the structure we get for our running example. The points associated with an internal node α are stored in an instance S_α of the data structure S for the decomposable searching problem at hand. To perform a query with object x at moment i we search with i in the tree to locate the atomic segment i lies in. (Note that

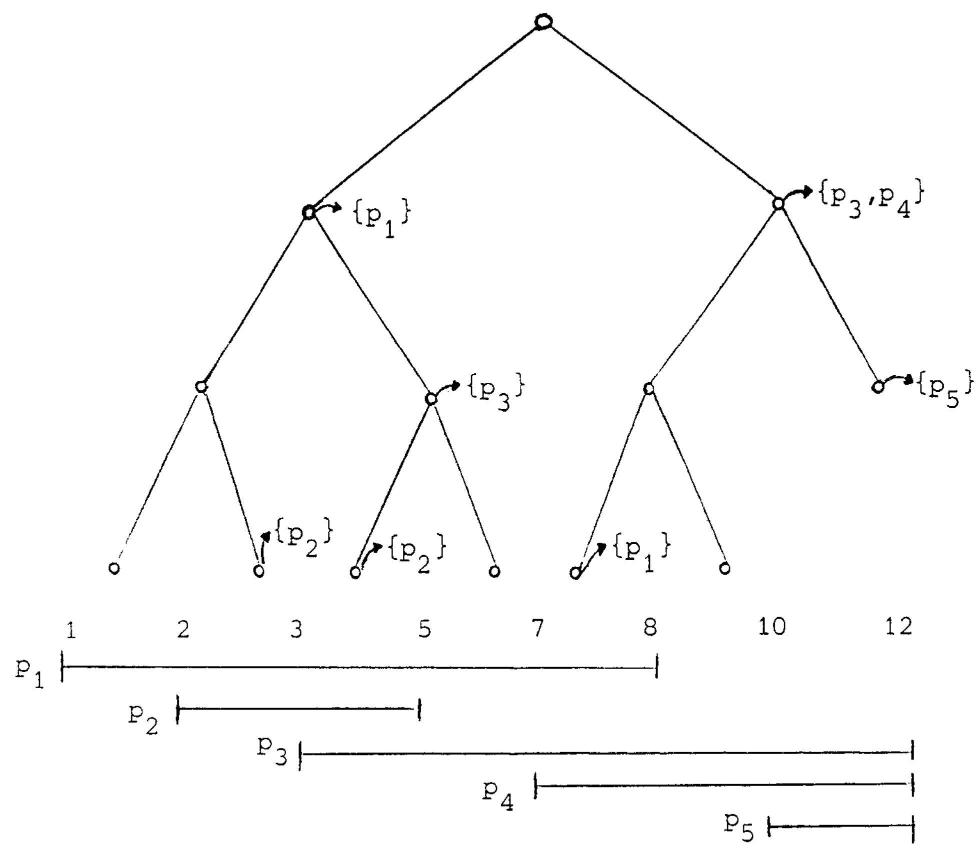

figure 4.

i never lies at the boundary of a segment). The structures S_α associated to the nodes α on the search path together contain each point present at moment i exactly once and contain only these points. Hence, we can solve the query by performing a query with x on each of these structures and combining the answers using the composition operator □. This leads to the following result:

Theorem 3.1. Given a static data structure S for solving a decomposable searching problem PR, the batched dynamic version of PR can be solved such that

$$P^d(n) = O(n_q.\log N.Q_S(m) + \log N.P_S(N)),$$
$$M^d(n) = O(n_q + \log N.M_S(N)).$$

Proof

Let us analyze the space requirements first. One easily verifies that at each level of the segment tree each point of the set occurs at most twice. As the number of points is bounded by N, it follows that the total amount of storage required for storing associated structures on one level is bounded by $O(M_S(N))$ (because M_S is assumed to be at least linear). As the depth of the segment tree is bounded by $O(\log N)$ and the tree itself uses only $O(N)$ storage, the bound on the amount of storage required follows (we need $O(n_q)$ space for storing the queries).

The amount of time required for solving the batched dynamic version of PR can be divided into two parts: the amount of time required for building the structure, and the amount of time required for performing the queries. Computing the existence intervals of the points in the set can be done in $O(n_q + N \log N)$ time. The construction of the segment tree (without the associated structures) takes $O(N \log N)$ time and the amount of time needed for constructing the associated structures can be estimated by $O(P_S(N))$ per level of the tree by the same arguments as used for estimating the amount of storage required. As each associated structure contains at most m points, the time needed per query is bounded by $O(\log N.Q_S(m))$ and, hence, the total amount of time needed for performing queries is bounded by $O(n_q.\log N.Q_S(m))$. The bound on P^d follows.

□

In certain cases, the factor log N in Theorem 3.1. needs not be paid. Exploiting the fact that the number of leaves descending from a node α is an upperbound on the number of points associated to α, one easily verifies

$$P^d(n) = O(n_q \cdot Q_S(N) + \log N . P_S(N)) \text{ when } Q_S(n) = \Omega(n^{\varepsilon}) \text{ for some } \varepsilon>0,$$
$$P^d(n) = O(n_q . \log N . Q_S(m) + P_S(N)) \text{ when } P_S(n) = \Omega(n^{1+\varepsilon}) \text{ for some } \varepsilon>0,$$
and
$$M^d(n) = O(n_q + M_S(N)) \qquad \text{when } M_S(n) = \Omega(n^{1+\varepsilon}) \text{ for some } \varepsilon>0.$$

For some decomposable searching problems, the amount of time needed for constructing the segment tree together with the associated structures can be improved by presorting the points. This method is particularly useful if the presorting improves the time required to construct an instance of S.

It is not strictly necessary to have a static data structure S available. We can also use the method described for solving the batched dynamic version of a decomposable searching problem when only a batched static solution is known. To this end we perform all queries simultaneously. For each internal node we collect all queries that have to be performed on the associated set of points and next solve the searching problem batched statically. This idea is exploited in the next section.

Let us now look at some applications of Theorem 3.1. For simplicity, we assume that $n = \theta(n_q) = \theta(N) = \theta(m)$.

Applications.

a) Nearest neighbor searching.

Given a set of points in the plane, the nearest neighbor searching problem asks for the point in the set nearest to a query point x (using the Euclidean metric). The best dynamic data structure currently know for the problem achieves a query time of $O(\sqrt{n \log n})$, an insertion time of of $O(\log n)$ and a deletion time of $O(\sqrt{n \log n})$, using $O(n \log \log n)$ storage (see Overmars and van Leeuwen [16]). Using this dynamic data structure for solving the batched dynamic version of the nearest neighbor searching problem, we obtain

$$P^d(n) = O(n\sqrt{n \log n}),$$
$$M^d(n) = O(n \log \log n).$$

On the other hand, a static data structure for the nearest neighbor searching problem is known with $Q_S(n) = O(\log n)$, $P_S(n) = O(n \log n)$ and $M_S(n) = O(n)$ (see Kirkpatrick [8]). Applying Theorem 3.1. to this structure we obtain a batched dynamic solution with

$$P^d(n) = O(n \log^2 n),$$
$$M^d(n) = O(n \log n).$$

b) Fixed radius near neighbor searching.

As shown in Section 2, a static data structure for the fixed radius near neighbor searching problem exists with $Q_S(n) = O(n^{2/3} \log^{2/3} n + k)$, $P_S(n) = O(n^{5/3} \log^{2/3} n)$ and $M_S(n) = O(n^{5/3} \log^{2/3} n)$. Applying Theorem 3.1. to this structure we obtain a batched dynamic solution to the fixed radius near neighbor searching problem with

$$P^d(n) = O(n^{5/3} \log^{2/3} n + k'),$$
$$M^d(n) = O(n^{5/3} \log^{2/3} n).$$

Hence, we obtain exactly the same bounds for the batched dynamic as for the batched static version.

c) Half-planar range counting.

As shown in Section 2, a static solution to the half-planar range counting problem exists with $Q_S(n) = O(\sqrt{n} \log n)$, $P_S(n) = O(n\sqrt{n} \log n)$ and $M_S(n) = O(n\sqrt{n})$. Applying Theorem 3.1. we obtain a batched dynamic solution to the problem with

$$P^d(n) = O(n\sqrt{n} \log n),$$
$$M^d(n) = O(n\sqrt{n}).$$

4. Streaming.

In this section we will describe a technique, called "streaming", for reducing the amount of storage required for solving batched (static or dynamic) versions of searching problems. The idea of streaming is the following. Rather than performing the queries one after the other on the data structure used, we perform them simultaneously. This is possible because all queries are known beforehand. To this end the data structure is traversed only once and hence, there is no need to have the complete structure available at the moment we start performing queries. At any stage of the process of performing the queries we only need that part of the data structure we are working on, i.e., the part the queries have come to. Hence, while performing the queries we build parts of the structure we need and discard parts that we do not need anymore. We will first consider the application of streaming to batched static solutions for decomposable searching problems.

Theorem 2.1. showed a simple technique for "balancing" the query time and building time of data structures for decomposable searching problems, by splitting the set of n points in f(n) subset and building a structure for each subset. When such a structure is used for solving the batched static version of a searching problem, there is no need for constructing all $f(n)$ structures in advance. We can proceed as follows. Build the first structure, compute the answers to the queries on this structure, store these partial answers and discard the first structure. Next, build the second structure, perform the queries and combine the answers with the stored answers using the composition operator $\square$. Repeat this for all $f(n)$ structures. It follows that at each moment there is at most one structure available. This reduces the storage requirements considerably. On the other hand, we have to store partial answers. Let the answer to a query for a set of n points take $A(n)$ storage. The technique described leads to the following result:

Theorem 4.1. Given a static data structure S for a decomposable searching problem PR and an integer function $f(n)$ with $1 \leq f(n) \leq n$, the batched static version of PR can be solved such that

$$P^S(n) = O(f(n_s).P_S(n_s/f(n_s)) + n_q.f(n_s).Q_S(n_s/f(n_s))),$$
$$M^S(n) = O(n_s + M_S(n_s/f(n_s)) + n_q.A(n_s)).$$

For a number of searching problems there is no need for storing partial answers. For example, in the range searching problem we can immediately report answers found. In such cases one can take $A(n_s) = 1$. (We always need n_q storage to store the query objects.)

Let us consider some applications, again assuming that $n = \theta(n_q) = \theta(n_s)$.

Applications.

a) Fixed radius near neighbor searching.

Taking $f(n) = \lceil n^{2/3}/\log^{1/3} n \rceil$ in Theorem 4.1., we obtain a solution to the batched static version of the fixed radius near neighbor searching problem with

$$P^S(n) = O(n^{5/3} \log^{2/3} n + k'),$$
$$M^S(n) = O(n \log n).$$

b) Half-planar range searching.

Applying Theorem 4.1. to the half-planar range counting problem with $f(n) = \lceil \sqrt{n} \rceil$, we obtain a batched static solution to the problem with

$$P^S(n) = O(n\sqrt{n} \log n),$$
$$M^S(n) = O(n).$$

Streaming can also be used for reducing the space requirements when using other types of data structures for solving the batched static version of searching problems. This will be demonstrated on a structure for range searching. We will first consider the two-dimensional case. A static data structure S is known with $Q_S(n) = O(\log n + k)$, $P_S(n) = O(n \log n)$ and $M_S(n) = O(n \log n)$ (see Willard [21]). It immediately yields a batched static solution with $P^S(n) = O(n \log n_s + k')$ and $M^S(n) = O(n_s \log n_s)$. We will show how streaming can be used to reduce the space requirements to $O(n)$.

Let V_s be the set of points and let V_q be the set of query ranges. We order the points in V_s with respect to their y-coordinates and order the ranges in V_q with respect to the y-coordinates of their lower borders. The algorithm is best described by the following recursive procedure:

<u>procedure</u> RANGE (<u>set of points</u> V_s, <u>set of ranges</u> V_q);

<u>step 1</u>: Determine a vertical line l that splits the set V_s of points into two nearly equal-sized subsets V_s^L and V_s^R of points to the left and to the right of l, respectively.

<u>step 2</u>: Determine the following subsets of V_q:

V_q^{L*} : contains all ranges whose x-intervals (i.e., projection on the x-axis) contain all x-values of points in V_s^L.

V_q^L : contains all ranges that lie partially to the left of l but are not contained in V_q^{L*}.

V_q^{R*} : similar to V_q^{L*} but for V_s^R.

V_q^R : similar to V_q^L but for V_s^R.

(Note that a range might come in more than one subset.)

<u>step 3</u>: Determine for each range in V_q^{L*} the points in V_s^L it contains. Do the same for V_q^{R*} and V_s^R. These actions can be done efficiently as the sets are ordered in a convenient way. Details will be described below.

<u>step 4</u>: Call recursively RANGE(V_s^L, V_q^L) and RANGE(V_s^R, V_q^R).

<u>end</u> of RANGE;

<u>Lemma</u> 4.3. A point p lying in a range r is reported exactly once for that range.

Proof

During the splitting of V_s there is a moment at which the x-interval of r contains all x-values of the points in the subset p is in. At this recursive call of RANGE p will be reported. Assume w.l.o.g. that p is in V_s^L. As r is in V_q^{L*} it cannot be in V_s^L. Hence, p is not reported later again for range r.

□

Lemma 4.2. shows that the algorithm works correctly. To estimate the amount of time and storage required we make the following observations:

Observation 1. At each moment, a point is in precisely one V_s set. A range is always in at most three sets of ranges.

It is worthwhile to note here that the order of the computations performed by the procedure RANGE is crucial. Only slight changes may violate Observation 1 and thus increase the space requirements.

Observation 2. The splitting of V_s (step 1) can be done in time $O(|V_s|)$. The splitting of V_q (step 2) can be done in time $O(|V_q| + |V_s|)$.

Observation 3. The level of nesting of recursive calls is bounded by $O(\log n_s)$.

Lemma 4.3. Step 3 can be performed in $O(n')$ time (plus the number of reported answers), where $n' = |V_s| + |V_q^{L*}| + |V_q^{R*}|$.

Proof

We will examine the actions taken for V_s^L and V_q^{L*} only. As the x-intervals of the ranges in V_q^{L*} contain all x-values of the points in V_s^L, we are only interested in the y-intervals and y-values of the ranges and the points. Hence, we have to solve a one-dimensional range searching problem. Both V_s^L and V_q^{L*} are ordered with respect to the y-coordinate. The queries are performed during a simultaneous walk along both sets. Let r_1 be the first range in V_q^{L*}. We walk along V_s^L until we find the first point p_i that lies in or past r_1. If p_i lies in r_1 we report this point and following points (as answers to the query with r_1) until we come to a point that lies past r_1. In this way we find all points in V_s^L that lie in r_1. Next we take r_2 and start the process at p_i (preceding points can never lie in r_2). In this way we continue with all ranges. One easily verifies that the amount of time required is bounded by $O(|V_s^L| + |V_q^{L*}|)$ plus the number of reported answers. Similar, we can perform the queries in V_q^{R*} on the points in V_s^R in time $O(|V_s^R| + |V_q^{R*}|)$.

□

It follows from Observations 2 and 3 and Lemma 4.3. that the total amount of time required for RANGE is bounded by $O(n_q+n_s)\log n_s + k')$, where k' denotes the total number of reported answers. As the ordering of V_s and V_q takes $O(n_q \log n_q + n_s \log n_s)$, the batched static version of the two-dimensional range searching problem can be solved in time $O(n \log n_s + n_q \log n_q + k')$. From Observation 1 it follows that the amount of storage required is bounded by $O(n)$.

To solve the batched static version of the d-dimensional range searching problem we use exactly the same procedure, except that we replace step 3 by

<u>step 3'</u>: Solve the batched static version of the d-1 dimensional range searching problem with the points in V_s^L (V_s^R), restricted to their last d-1 coordinates, as set of points and the ranges in V_q^{L*} (V_q^{R*}), restricted to their last d-1 coordinates, as set of query ranges.

This easily leads to the following result:

<u>Theorem</u> 4.4. The batched static version of the d-dimensional range searching problem can be solved such that

$$P^S(n) = O(n \log^{d-1} n_s + n_q \log n_q + k'),$$
$$M^S(n) = O(n).$$

In essentially the same way one can apply streaming to the RI-tree of Edelsbrunner [4] (see also Edelsbrunner and Maurer [6]) to solve the batched static version of the d-dimensional rectangle intersection searching problem. It yields the following result:

<u>Theorem</u> 4.5. The batched static version of the d-dimensional rectangle intersection searching problem can be solved such that

$$P^s(n) = O(n \log^{d-1} n_s + n_q \log n_q + k'),$$
$$M^s(n) = O(n).$$

One can use the batched static version of the rectangle intersection searching problem for solving the rectangle intersection problem, i.e., the problem of determining all intersecting pairs among a set V of n rectilinearly oriented hyper-rectangles in d-dimensional space. To this end we take the rectangles in V both as set and as query objects. In this way each intersecting pair would be reported twice but one can easily take care that this will not happen.

Corollary 4.6. The d-dimensional rectangle intersection problem can be solved in time $O(n \log^{d-1} n + k)$ using $O(n)$ space, where k is the number of reported intersecting pairs.

This result improves the best known solutions for the problem (see e.g. Edelsbrunner and Maurer [6]) that use $O(n \log^{d-2} n)$ space.

The idea of streaming can also be used for reducing the amount of space required for solving the batched dynamic version of decomposable searching problems. We essentially use the structure described in Section 3 but do not build the structure at once. The structure is rather built and destroyed node by node, performing all queries simultaneously. As an important side-benefit we only need a batched static solution for the decomposable searching problem rather than a static solution.

Let V_s denote the set of points and let with each point its existence interval be given. The endpoints of the existence intervals are sorted from left to right. Let V_q be the set of query objects and let with each query object the moment at which it is performed be given. We order V_q with respect to these moments. As we will perform queries simultaneously we have to store partial answers (although this may not be necessary for all decomposable searching problems). For this task we use an array ANSW that stores for each query object the answer for the part of the set examined up to now. Each time we compute the answer to a query for a part of the set, we combine it with the corresponding answer in ANSW using the composition operator $\square$. The algorithm is best described by the following recursive procedure:

<u>procedure</u> BATCHDYN (<u>set of points</u> V_s, <u>set of queries</u> V_q);

<u>step 1</u>: Determine a time moment t such that half of the time moments of queries in V_q is before t. Partition V_q into V_q^L and V_q^R such that V_q^L (V_q^R) contains the queries in V_q whose time moments are before (after) t.

<u>step 2</u>: Determine the following subsets of V_s:

V_s^{L*}: contains all points whose existence intervals contain all time moments of queries in V_q^L.

V_s^L : contains all points, not in V_s^{L*}, whose existence intervals lie partially before t.

V_s^{R*}: similar to V_s^{L*} but for V_q^R.

V_s^R : similar to V_s^L but for V_q^R.

(Note that a point might come in more than one set.)

<u>step 3</u>: Solve the batched static version of the decomposable searching problem for V_s^{L*} and V_q^L and for V_s^{R*} and V_q^R, combining the answers with the corresponding answers in ANSW.

<u>step 4</u>: Call recursively BATCHDYN(V_s^L,V_q^L) and
BATCHDYN(V_s^R,V_q^R).

<u>end</u> of BATCHDYN;

Clearly, step 1 and 2 take $O(|V_q| + |V_s|)$ time and step 3 takes $O(P^S(n'))$ time where $n' = |V_s^{L*}| + |V_s^{R*}| + |V_q|$. One easily verifies that the depth of the recursion is bounded by $O(\log n_q)$ and (hence) that the total amount of time required is bounded by $O(\log n_q\ P^S(n) + n_q \log n_q + N \log n_q)$ plus $O(N \log N + n_q)$ for computing the existence intervals of the points and constructing the initial sets V_s and V_q. Beside the amount of storage required for ANSW, the algorithm takes $O(M^S(n) + n) = O(M^S(n))$ storage.

<u>Theorem</u> 4.7. Given a batched static solution to a decomposable searching problem PR, the batched dynamic version of PR can be solved such that

$$P^d(n) = O(P^S(n)\log n_q + N \log N),$$
$$M^d(n) = O(M^S(n) + n_q.A(m)),$$

where A(m) denotes the amount of space required for storing the answer over a set of m points.

One easily verifies that $P^d(n) = O(P^s(n))$ when $P^s(n) = \Omega(n^{1+\varepsilon})$ for some $\varepsilon>0$.

Let us consider some applications. We assume that $n = \theta(n_q) = \theta(N) = \theta(m)$.

Applications.

a) Nearest neighbor searching.

Applying Theorem 4.7. to the structure for nearest neighbor searching of Kirkpatrick [8] we obtain a solution to the batched dynamic version of the nearest neighbor searching problem with

$$P^d(n) = O(n \log^2 n),$$
$$M^d(n) = O(n).$$

b) Range searching and rectangle intersections searching.

Applying Theorem 4.7. to the batched static solution for the d-dimensional range searching problem given in Theorem 4.4. we obtain a batched dynamic solution with

$$P^d(n) = O(n \log^d n + k'),$$
$$M^d(n) = O(n).$$

Applying Theorem 4.7. to Theorem 4.5. we obtain a batched dynamic solution for the d-dimensional rectangle intersection searching problem with

$$P^d(n) = O(n \log^d n + k'),$$
$$M^d(n) = O(n).$$

c) immediate obstacle searching problem.

Applying Theorem 4.7. to the batched static solution for the problem of McCreight [12] given in Section 2, we obtain a batched dynamic solution with

$$P^d(n) = O(n \log^2 n),$$
$$M^d(n) = O(n).$$

5. Reversing searching problems.

For a number of searching problems one can obtain efficient batched static or dynamic solutions by viewing the query objects as set objects and vice versa. We will demonstrate the idea by applying it to the triangular range searching problem. The triangular range searching problem is the following: given a set of points in the plane, report all points that lie within a given query triangle. Some data structures are known for the problem. Willard [20] describes a solution with

$Q_S(n) = O(n^{\log_6 4} + k)$, $P_S(n) = O(n^2)$ and $M_S(n) = O(n \log n)$ where k is the number of reported answers. Edelsbrunner, Kirkpatrick and Maurer [5] solve the problem in $Q_S(n) = O(\log n + k)$, $P_S(n) = O(n^7)$ and $M_S(n) = O(n^7)$. Both structures are quite inappropriate for solving the batched static version of the problem, even when we apply Theorem 4.1. The batched static version can be formulated as follows: given a set of triangles and a set of points in the plane report for each triangle the points it contains. In other words, report all pairs (triangle, point) with the point contained in the triangle. But to report these pairs we can as well ask for each point in what triangles it lies. Hence, we obtain the following searching problem: given a set of triangles in the plane, report all triangles that contain a given query point. One can easily give a static solution for this problem with $Q_S(n) = O(\log n)$, $P_S(n) = O(n^3)$ and $M_S(n) = O(n^3)$, based on a data structure for point location in a planar subdivision due to Kirkpatrick [8]. Applying Theorem 4.1. with $f(n) = \lceil n^{2/3}/\log^{1/3} n \rceil$ yields a batched static solution to the problem with

$$P^s(n) = O(n^{5/3} \log^{2/3} n),$$
$$M^s(n) = O(n \log n),$$

assuming that $n = \theta(n_s) = \theta(n_q)$. Hence, we can solve the batched static version of the triangular range searching problem within these bounds. Applying Theorem 4.7. we obtain a solution to the batched dynamic version of the problem such that

$$P^d(n) = O(n^{5/3} \log^{2/3} n),$$
$$M^d(n) = O(n \log n).$$

It is hard to give general constraints a problem should satisfy to be "reversible" (i.e., in which query and set objects can be interchanged). One class of problems that can be reversed is the class of so-called "set independent" problems.

Definition 5.1. A searching problem PR is called set independent if and only if there exist some function $f(p)$ that maps points into answers and a relation $R(p,x)$ between points and query objects such that for every set of points V

$$PR(x,V) = \{f(p) \mid p \in V \text{ and } R(p,x)\}.$$

Hence, the answer to a set independent searching problem consists of a set of answers $f(p)$ for those points p in V that satisfy the relation $R(p,x)$. Whether $f(p)$ is reported or not is independent of the other elements of the set. Clearly, a set independent problem is decomposable. Some examples of set independent problems are the range searching problem, the rectangle intersection searching problem, the fixed radius near neighbor searching problem and the triangular range searching problem described above.

Theorem 5.2. A set independent searching problem PR is reversible.

Proof

The answer to the batched static version of PR consists of a number of pairs $(x,f(p))$, where x is one of the query objects, p is a point in the set V and $R(p,x)$ holds. We can compute the pairs by solving for each p in V the searching problem $PR'(p,V_q)$ where V_q is the set of query objects and

$$PR'(p,V_q) = \{(x,f(p)) \mid x \in V_q \text{ and } R(p,x)\}.$$

□

The interchange of set and query objects does not give better results for all set independent searching problems. An example for which we do get better results is the circular range searching problem. The circular range searching problem is the following: given a set of points in the plane, report those points that lie in a given query circle (of arbitrary size). The reversed problem asks for those circles in a set that contain a given query point. Using the planar point location algorithm of Preparata [17] this problem can be solved within $Q_S(n) = O(\log n + k)$, $P_S(n) = O(n^3)$ and $M_S(n) = O(n^3)$. Applying Theorem 4.1. with $f(n) = \lceil n^{2/3}/\log^{1/3} n \rceil$ we can solve the batched static version of this problem, and hence, the batched static version of the circular range searching problem within

$$P^S(n) = O(n^{5/3} \log^{2/3} n),$$
$$M^S(n) = O(n \log n),$$

assuming that $n = \theta(n_q) = \theta(n_s)$.

6. Extensions.

When the number of updates is not of the same order as the number of queries we can tune Theorem 3.1. and Theorem 4.7. to obtain better time bounds for the batched dynamic solution. We will only show how to improve Theorem 3.1. The results for Theorem 4.7. follow in a direct way.

Let us first consider the case that the number of queries is much larger than the number of updates. Again we use an augmented segment tree but rather than using a binary tree as underlying structure a $f(N)$-ary tree is used, i.e., a tree in which each internal node has $f(N)$ sons for some integer function f depending on the number of updates. To each internal node α we again associate an instance S_α of the static structure S containing all points whose existence intervals cover the whole interval below α but not the whole interval below the father of α. The depth of such a segment tree is clearly bounded by $\lceil \log N/\log f(N) \rceil$. One easily verifies that each point is contained in at most $O(f(N) \log N/\log f(N))$ associated structures. Hence, the building time is bounded by

$$O(\frac{f(N)}{\log f(N)}.\log N.P_S(N)).$$

To query the structure we have to query at most $\log N/\log f(N)$ associated structures. Hence, each query takes at most

$$O(\frac{1}{\log f(N)}.\log N.Q_S(m) + \log N)$$

time (the extra $\log N$ comes in for querying the segment tree itself). So the total amount of time required for performing queries is bounded by

$$O(\frac{1}{\log f(N)}.n_q.\log N.Q_S(m) + n_q \log N).$$

One easily verifies that the amount of storage required is bounded by $O(f(N).M_S(N))$ per level, and hence, the total amount of storage required is bounded by

$$O(\frac{f(N)}{\log f(N)}.\log N.M_S(N)).$$

This leads to the following refined version of Theorem 3.1.:

Theorem 6.1. Let $f(n)$ be an integer function with $2 \leq f(n) \leq n$. Given a static data structure S for solving a decomposable searching problem PR, the batched dynamic version of PR can be solved such that

$$P^d(n) = O(\frac{1}{\log f(N)}.n_q.\log N.Q_S(m) + n_q.\log N + \frac{f(N)}{\log f(N)}.\log N.P_S(N)),$$

$$M^d(n) = O(\frac{f(N)}{\log f(N)}.\log N.\ M_S(N)).$$

Hence, at the cost of an increase of $f(N)/\log f(N)$ in the amount of time required for building the structure we obtain a decrease with a factor $\log f(N)$ in the amount of time required for performing the queries. As an example, consider the nearest neighbor searching problem and assume that $n_q = N \log N$ and $m=N$. Theorem 3.1. yields a solution to the batched dynamic version with

$$P^d(n) = O(N \log^3 N + N \log^2 N) = O(N \log^3 N).$$

Using Theorem 6.1. with $f(N) = \log N$ we obtain a solution with

$$\begin{aligned} P^d(n) &= O(N \log^3 N/\log\log N + N \log^2 N + N \log^3 N/\log\log N) \\ &= O(N \log^3 N/\log\log N). \end{aligned}$$

The fact that the amount of storage is increased is not relevant as it can be reduced to $O(M_S(N))$ using streaming.

Let us now assume that the number of updates is much larger than the number of queries. Again we use a $f(N)$-ary segment tree but we associate structures in a different way. Let $son_1,\ldots,son_{f(N)}$ be the sons of some internal node α. Rather than associating a structure with each son_i we construct structures $S_{i,j}$ ($1 \leq i \leq j \leq f(N)$ but not $i=1$ and $j=f(N)$) that contain all points whose existence internals cover all intervals below $son_i,\ldots,son_j$ but not the intervals below other sons. These are $O(f(N)^2)$ structures. One easily verifies that at each level of the tree each point is in at most two associated structures. It follows that the building time per level is bounded by $O(P_S(N))$ and hence, that the total building time is bounded by

$$O(\frac{1}{\log f(N)}.\log N.P_S(N))$$

plus $O(N \log N)$ for building the segment tree and computing the existence intervals. When we perform a query and the query-path goes through son_k, we have to perform a query on all structures $S_{i,j}$ with $i \leq k$ and $j \geq k$. There can be $\theta(f(N)^2)$ such structures. Hence, the total number of structures we have to query is bounded by

$$O(f(N)^2.\frac{\log N}{\log f(N)}).$$

It follows that the total amount of time required for performing all queries is bounded by

$$O(\frac{f(N)^2}{\log f(N)}.n_q.\log N.Q_S(m)).$$

One easily verifies that at each level of the tree we need at most $O(M_S(N))$ storage. Hence, we can refine Theorem 3.1. to obtain:

Theorem 6.2. Let $f(n)$ be an integer function with $2 \leq f(n) \leq \sqrt{n}$. Given a static data structure S for solving a decomposable searching problem PR, the batched dynamic version of PR can be solved such that

$$P^d(n) = O(\frac{f(N)^2}{\log f(N)}.n_q.\log N.Q_S(m) + \frac{1}{\log f(N)}.\log N.P_S(N) + N \log N),$$

$$M^d(n) = O(\frac{1}{\log f(N)}.\log N.M_S(N)).$$

Using streaming, the amount of storage required can be reduced to $O(M_S(N))$. As an example consider again the nearest neighbor searching problem and let $n_q = \frac{N}{\log N}$ and $m=N$. Theorem 3.1. would yield a solution with

$$P^d(n) = O(\frac{N}{\log N}.\log^2 N + \log N.N \log N) = O(N \log^2 N).$$

Applying Theorem 6.2. with $f(N) = \sqrt{\log N}$, we obtain a solution with

$$P^d(n) = O(\frac{\log N}{\log\sqrt{\log N}}.\frac{N}{\log N}.\log^2 N + \frac{1}{\log\sqrt{\log N}}.\log N.N \log N + N \log N)$$

$$= O(N \log^2 N/\log \log N).$$

7. Concluding remarks.

We have given a number of techniques that can be used for solving batched static and batched dynamic versions of decomposable searching problems. In the batched static case, the plane sweep technique proved to be a powerful instrument for solving a number of (planar geometrical) problems. A general transformation showed how data structures with a large difference between query and building time can be turned into structures with better trade-offs, leading to better time bounds when used in a batched environment. Next, it was shown how the batched dynamic version of a searching problem could be transformed into the addition of inverse range restrictions. It resulted in a general method for solving the batched dynamic version of decomposable searching problems that is applicable once a static (or batched static) solution to the problem is known. It was also shown that the space requirements of the method can be reduced considerably by performing all queries simultaneously and building only those parts of the data structure we are busy performing the queries on. We believe that this technique of "streaming" is applicable in numerous other problems as well. For example, it can be used (as shown in Section 4) to solve the d-dimensional rectangle intersection problem in time $O(n \log^{d-1} n + k)$ using only $O(n)$ storage, where n is the number of rectangles and k the number of reported intersecting pairs. It was also demonstrated how for a number of searching problems better results can be obtained by viewing set objects as query objects and vice versa.

A number of open problems remain. In Section 6 we showed how different trade-offs can be obtained between the amount of time required for building the structure and the amount of time required for performing the queries. Other trade-offs might exist. No lowerbounds are known for the efficiency of the transformations. More over, we only considered decomposable searching problems. General methods for solving batched versions for other classes of problems, e.g. the order decomposable set problems (Overmars [14]) might exist as well.

References.

[1] Bentley, J.L., Decomposable searching problems, Inform. Proc. Lett. 8 (1979) 244-251.

[2] Bentley, J.L. and H.A. Maurer, A note on Euclidean near neighbor searching in the plane, Inform. Proc. Lett. 8 (1979) 133-136.

[3] Bentley, J.L. and D. Wood, An optimal worst case algorithm for reporting intersections of rectangles, IEEE Trans. on Comp. C-29 (1980) 571-577.

[4] Edelsbrunner, H., Dynamic data structures for orthogonal intersection queries, Report F59, Inst. f. Informationsverarbeitung, TU Graz, 1980.

[5] Edelsbrunner, H., D.G. Kirkpatrick and H.A. Maurer, Polygonal intersection searching, Inform. Proc. Lett. 14 (1982) 74-79.

[6] Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects, Inform. Proc. Lett. 13 (1981) 177-181.

[7] Edelsbrunner, H. and M.H. Overmars, On the equivalence of some rectangle problems, Inform. Proc. Lett. 14 (1982) 124-127.

[8] Kirkpatrick, D.G., Optimal search in planar subdivisions, SIAM J. Computing 12 (1983) 28-35.

[9] Lee, D.T. and F.P. Preparata, An improved algorithm for the rectangle enclosure problem, J. of Algorithms 3 (1982) 218-224.

[10] Lee, D.T. and C.K. Wong, Finding intersections of rectangles by range search, J. of Algorithms 2 (1981) 337-347.

[11] Maurer, H.A. and T.A. Ottmann, Dynamic solutions of decomposable searching problems, in: U. Pape (ed.), Discrete structures and algorithms, Hanser Verlag, Wien, 1979, 17-24.

[12] McCreight, E., <problem 81-8>, J. of Algorithms 2 (1981) 314.

[13] Nievergelt, J. and F.P. Preparata, Plane-Sweep algorithms for intersecting geometric figures, Comm. of the ACM 25 (1982) 739-747.

[14] Overmars, M.H., Dynamization of order decomposable set problems, J. of Algorithms 2 (1981) 245-260.

[15] Overmars, M.H., The design of dynamic data structures, Ph.D. Thesis, Dept. of Computer Science, University of Utrecht, 1983. (to appear in Lect. Notes in Comp. Science, Springer Verlag)

[16] Overmars, M.H. and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, Inform. Proc. Lett. 12 (1981) 168-173.

[17] Preparata, F.P., A new approach to planar point location, SIAM J. Computing 10 (1981) 473-482.

[18] Shamos, M.I. and D. Hoey, Geometric intersection problems, Proc. 17^{th} Annual IEEE Symp. on Foundations of Computer Science, 1976, 208-215.

[19] van Leeuwen, J. and D. Wood, The measure problem for rectangular ranges in d-space, J. of Algorithms 2 (1981) 282-300.

[20] Willard, D.E., Polygon retrieval, SIAM J. Computing 11 (1982) 149-165.

[21] Willard, D.E., New data structures for orthogonal queries, Techn. Rep. TR-22-78, Aiken Computation Lab., Harvard University, 1978. (revised version to appear in SIAM J. Computing)