THE ART OF DYNAMIZING

Jan van Leeuwen and Mark H. Overmars

RUU-CS-81-8

April 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

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

THE ART OF DYNAMIZING

Jan van Leeuwen and Mark H. Overmars

Technical Report RUU-CS-81-8

April 1981

Department of Computer Science

University of Utrecht

P.O. Box 80.002

3508 TA Utrecht, the Netherlands

This paper will appear in the Proceedings of the 10th Int. Symp. on Mathematical Foundations of Computer Science, Štrbské Pleso, Aug. 31 - Sept. 4, 1981 (to be published by Springer Verlag).

THE ART OF DYNAMIZING

Jan van Leeuwen and Mark H. Overmars*
Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Abstract

A few years ago J. Bentley initiated a general approach to searching problems and their solution by means of dynamic data structures. As it is often easier to find a static solution first, his goal was to obtain efficient dynamic data structures by applying transformations to static data structures. This approach has become a paradigm (known as "dynamization") in current research in the design of efficient algorithms. We shall outline a number of general techniques that were developed for dynamizing decomposable searching problems and will discuss a recent solution to Bentley's original question to devise a dynamization method for such problems with worst-case optimal insertion and deletion routines.

1. Introduction

The design of efficient algorithms and data structures often aims at providing solutions to object-oriented searching problems. A searching problem is a problem in which a question (query) is asked about an object with respect to a set of other objects (points) or, sometimes, about a property of the set itself. To get a feel for the intricacy of searching problems, consider the following examples:

a) range searching

Given a finite set V of points in $\mathbb{R}^d$ and any "range" $([a_1 .. b_1], [a_2 .. b_2], \ldots, [a_d .. b_d])$, we wish to report, or just count, the $x \in V$ with $a_i \leq x_i \leq b_i$ $(1 \leq i \leq d)$.

b) rectangle intersection

Given a finite set V of recti-linearly oriented hyper-rectangles in $\mathbb{R}^d$ and

* The work of this author was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).

any query-rectangle x, report all $y \in V$ which intersect x (i.e., have at least one point in common with it).

c) nearest neighbor searching

Given a finite set V of points in $\mathbb{R}^d$ and any point x, we wish to report a $y \in V$ with smallest distance (in, say, the Euclidean metric) to x.

d) convex hull searching

Given a finite set V of points in $\mathbb{R}^d$ and any point x, we wish to determine whether x lies in the interior of the convex hull of V.

For d = 1 these problems can often easily be solved, even dynamically, by means of balanced search trees of some kind. For d > 1 the problems become much harder, as balancing criteria do not seem to carry over. Solutions tend to be complex pointer structures that are practical only for static sets. In some cases dynamic data structures have been found (see e.g. Lueker [9], Willard [24], Edelsbrunner [4], Overmars and van Leeuwen [16]), after a considerable effort of data structure engineering.

A few years ago J. Bentley (see [2]) initiated a general approach to searching problems and their solution by means of dynamic data structures. He observed that in certain cases efficient dynamic data structures may be obtained by applying suitable transformations to known static data structures. This approach to the design of dynamic data structures, termed "dynamization" by van Leeuwen and Wood [22], has led to a number of surprisingly powerful results that only require that an efficient static solution to a searching problem be known. We will restrict ourselves to the original class of searching problems distinguished by Bentley [2]. Let Q(x,V) denote the answer to a searching problem Q with object x and set V.

Definition. A searching problem Q is called decomposable if and only if for any partition $V_1 \cup V_2$ of V one has

$$Q(x,V) = \square(Q(x,V_1), Q(x,V_2))$$

for a constant time computable operator $\square$.

Note that examples a, b and c are decomposable searching problems, while example d is not.

For decomposable searching problems one can avoid storing all points in one large (and static) data structure. Instead, the set can be maintained as a "dynamic system" of disjoint subsets (blocks) of smaller sizes that are statically structured each. Queries can be performed by querying each of the blocks and combining the answers using $\square$, at only negligible extra costs. Observe that (i) maintaining a "large" number of small blocks will lead to low update- and high query times and (ii) maintaining a "small" number of large blocks will lead to high update- and low query times (compared to the update and query times of the original, static structure). Most dynamizations of decomposable searching problems tend to have their own way of decomposing (parti-

tioning) a set into blocks and of maintaining the decomposition as points are inserted or deleted, to try and strike an "optimal" balance between the overhead required for processing updates and queries, respectively. Bentley [2] and Saxe and Bentley [19] gave a variety of methods to just perform insertions, achieving different levels of efficiency. Soon after, several authors addressed the problem of efficiently supporting both insertions and deletions (Maurer and Ottmann [10], van Leeuwen and Wood [22], Overmars and van Leeuwen [14] and van Leeuwen and Maurer [21]). Working from the other end, Mehlhorn [11] (extending results of Saxe and Benley [19]) obtained good lower-bounds on the update- and query times and their trade-off in general dynamization methods and Mehlhorn and Overmars [12] devised a family of dynamizations that actually achieved the proven lowerbounds when only insertions are considered. The time bounds in these methods normally are averages. Recently, Overmars and van Leeuwen [17] developed a number of techniques to turn "averages" into "worst-case" bounds and in [18] they succeeded in putting all insights together to obtain an answer to Bentley's original question of devising a dynamization of decomposable searching problems with (general) worst-case optimal insertion and deletion routines.

In this paper we shall present a number of useful techniques for dynamizing decomposable structures and cite some main theorems and applications.

Notation : n = the current number of points in the set, N = the number of transactions that have occurred on an initially empty structure S, $Q_S(n)$ = the time needed to solve Q on a structure S of n points, $P_S(n)$ = the time required to build a (static) structure S of n points, $D_S(n)$ = the time needed to perform a deletion in a structure S of n points, $I_S(n)$ = the time needed to perform an insertion in a structure S of n points.

We shall assume that Q_S is nondecreasing, that P_S is at least linear and that both are "smooth" (i.e., are functions f with $f(c.n) = O(f(n))$ for every c). Bounds are normally worst-case bounds, unless explicitly superscripted with "a" (for average).

2. Techniques

We discuss three important ingredients of general dynamization schemes: (i) spreading work over time, (ii) the "equal blocks" method and (iii) the "logarithmic" method. It will show the main techniques that are used all over.

To appreciate the need for spreading work over time, imagine that a structure S for Q is known on which insertions and deletions can be processes in such a manner that S will only slowly get out of "balance". As long as S remains in reasonable shape we can use it for query-answering, but every once-in-a-while S must be fully rebuilt to restore its balance. We will show these "bursts" of work can be spread over time.

Definition. Insertions/deletions are called "weak" if the routines to perform them on an admissible structure S of n points only guarantee that after αn insertions and/or βn deletions ($\beta < 1$) the query time on the resulting set of m elements is bounded by $k_{\alpha\beta}\, Q_S(m)$, for some constant $k_{\alpha\beta}$ depending only on α and β.

Notation. $WD_S(n)$ = the time needed to perform a weak deletion on a structure S of n points, $WI_S(n)$ = the time needed to perform a weak insertion on a structure S of n points.

Theorem 1. ([18]) Given a structure S for a searching problem Q which allows for weak updates, there is a structure S_1 for Q with

$$Q_{S_1}(n) = O(Q_S(n))$$
$$D_{S_1}(n) = O(WD_S(n) + P_S(n)/n)$$
$$I_{S_1}(n) = O(WI_S(n) + P_S(n)/n)$$

Proof

S_1 will normally consist of just one S-structure MAIN. As insertions and deletions take place, MAIN will slowly grow out of balance. When the number of updates become equal to half its initial size (i.e., $\alpha + \beta = \frac{1}{2}$), MAIN is made into OLDMAIN and the construction of a new MAIN is started up. We omit the easy details of the necessary administration of elements. Assume MAIN had n_0 elements at this point. We shall see to it that the new MAIN can take over within $\frac{1}{6}n_0$ updates. Note that the incoming updates must be carried out on the new MAIN as well (after it is constructed). For some time we shall (i) continue to perform updates on OLDMAIN (so it remains of use for all query answering), (ii) spend $WD_S(\frac{7}{6}n_0) + \lceil P_S(n_0)/\frac{1}{6}n_0 \rceil$ time on building the new MAIN with every deletion and likewise $WI_S(\frac{7}{6}n_0) + \lceil P_S(n_0)/\frac{1}{6}n_0 \rceil$ time with every insertion and (iii) put each update on a queue BUF (to save it for performing it on the new MAIN later). Suppose MAIN gets finished while update s is processed. Suppose there have been d_1 deletions and i_1 insertions ($i_1 + d_1 = s$) and that w time is left unused of update s. Clearly

$$s.\lceil P_S(n_0)/\tfrac{1}{6}n_0 \rceil + d_1 WD_S(\tfrac{7}{6}n_0) + i_1 WI_S(\tfrac{7}{6}n_0) - w = P_S(n_0)$$

, thus

$$d_1 WD_S(\tfrac{7}{6}n_0) + i_1 WI_S(\tfrac{7}{6}n_0) = P_S(n_0) + w - s.\lceil P_S(n_0)/\tfrac{1}{6}n_0 \rceil.$$

Spend the w time left immediately on performing updates from BUF on the new MAIN. For the next period of time (until BUF is empty) we shall (i) still continue to perform incoming updates on OLDMAIN (for MAIN has not taken over yet), (ii) spend $WD_S(\frac{7}{6}n_0) + \lceil P_S(n_0)/\frac{1}{6}n_0 \rceil$ time with every deletion on processing updates from BUF on MAIN and likewise $WI_S(\frac{7}{6}n_0) + \lceil P_S(n_0)/\frac{1}{6}n_0 \rceil$ time with every insertion and (iii) store each update on BUF if BUF isn't empty yet (otherwise we can directly perform the update). Assuming MAIN is completely up-to-date within $\frac{1}{6}n_0$ updates, its size cannot be larger than $n_0 + \frac{1}{6}n_0 = \frac{7}{6}n_0$ as the buffered updates are carried out. Let BUF become empty

during transaction t and suppose there were d_2 deletions and i_2 insertions in this period (which did get buffered as well!). Clearly t is bounded by the smallest integer such that

$$w + t\lceil P_S(n_0)/\tfrac{1}{6}n_0\rceil + d_2WD_S(\tfrac{7}{6}n_0) + i_2WI_S(\tfrac{7}{6}n_0) \geq$$
$$\geq d_1WD_S(\tfrac{7}{6}n_0) + i_1WI_S(\tfrac{7}{6}n_0) + d_2WD_S(\tfrac{7}{6}n_0) + i_2WI_S(\tfrac{7}{6}n_0)$$

, where the right-hand side accounts for the maximum of time required to process all buffered transactions. It follows that $w + t.\lceil P_S(n_0)/\tfrac{1}{6}n_0\rceil \geq P_S(n_0) + w - s\lceil P_S(n_0)/\tfrac{1}{6}n_0\rceil$, thus $(s + t)\lceil P_S(n_0)/\tfrac{1}{6}n_0\rceil \geq P_S(n_0)$. Since $s \leq \tfrac{1}{6}n_0$ (by inspecting the first phase), we certainly have $s + t \leq \tfrac{1}{6}n_0$ and MAIN must be able to take over within the number of updates claimed. Since the new MAIN started at size n_0, the $\tfrac{1}{6}n_0$ updates performed on it cannot exceed "half its size" in number in the meantime. Thus the new MAIN will not have to be "rebalanced" before it is released from the construction. The structure that became OLDMAIN is eventually discarded after at most $\tfrac{1}{2}n_1 + \tfrac{1}{6}n_0 \leq \tfrac{3}{4}n_1$ updates, assuming its initial size was n_1. Further details are easy. □

Note that theorem 1 does not require that Q is decomposable.

The "equal blocks" method (Maurer and Ottmann [10], van Leeuwen and Wood [22], van Leeuwen and Maurer [21]) nicely illustrates (i) the flexibility of a partition in which the blocks are given some freedom to grow and shrink in size, (ii) the way to achieve good average time bounds by making reconstructions necessary only after "many" updates have occurred and (iii) the global argument used to go from good average to good worst-case bounds. Let f be a smooth and non-decreasing integer function with $1 \leq f(n) \leq n$. The equal blocks method keeps V partitioned into blocks $V_1, \ldots, V_k$ with (i) $f(\tfrac{1}{2}n) \leq k \leq f(2n)$ and (ii) $|V_j| \leq 2n/k$ $(1 \leq j \leq k)$. (This is only slightly more general than in [21], [22].) Queries are answered by querying each of the k blocks and composing the results through □. Deletions are processed by just searching for the element, deleting it and rebuilding the block without the element. Insertions always take place in the currently smallest block. When constraints (i), (ii) are violated after an update has taken place and V now has a total of n_0 points, the whole partition is built anew with $k = f(n_0)$ and $|V_j| = n_0/k$. Note that this gives the blocks a considerable amount of leeway before the structure is declared "out of balance" again. Thus the work to reconfigure V, estimated at $k.P_S(n_0/k) = n_0.P_S(n_0/k)/\tfrac{n_0}{k} \leq n_0.P_S(n_0)/n_0 = P_S(n_0)$, will not have to be spend very often. Suppose the next time V goes out of balance, it has n points. Consider the three possible ways constraints (i), (ii) can be violated:

Case I : $k < f(\tfrac{1}{2}n)$. Hence $k = f(n_0) < f(\tfrac{1}{2}n) \Rightarrow n_0 < \tfrac{1}{2}n \Rightarrow n > 2n_0$ (and n is the first such number). Thus at least $n_0 \simeq \tfrac{1}{2}n$ insertions must have taken place after the latest reconstruction.

Case II : $k > f(2n)$. Hence $k = f(n_0) > f(2n) \Rightarrow n_0 > 2n \Rightarrow n < \tfrac{1}{2}n_0$. Thus at least $\tfrac{1}{2}n_0 \simeq n$ deletions must have taken place after the latest reconstruction.

Case III: $|V_j| > 2n/k$, some j. Note that insertions always take place in blocks of smallest size and that such blocks must have size $\leq n/k$. (Also, at an insertion n is incremented by 1.) It follows that a block can only violate the size constraint as the result of "many" deletions that take place in other blocks. Suppose V had n_1 points when the last insertion took place on V_j. (Take $n_1 = n_0$ if there were no insertions into V_j.) At that moment $|V_j| \leq n_1/k$ and $|V_j|$ can only have shrunk since. Hence $2n/k < |V_j| \leq n_1/k \Rightarrow 2n < n_1 \Rightarrow n < \frac{1}{2}n_1$. Thus at least $\frac{1}{2}n_1 \simeq n$ deletions must have taken place.

It follows that in all cases at least $\frac{1}{2}n$ insertions and/or deletions must have taken place before the $O(P_S(n))$ work on reconfiguring V must be spend again. We formulate updates of blocks as weak updates (which could be as costly as $P_S(n/k)$ if indeed a block is completely reconstructed every time). Note that $n = \Theta(n_0)$.

Proposition 1. Given a (static) structure S for a decomposable searching problem Q and a smooth and non-decreasing integer function f with $1 \leq f(n) \leq n$, there is a structure S_1 for Q with

$$Q_{S_1}(n) = O(f(n)\ Q_S(n/f(n)))$$
$$D^a_{S_1}(n) = O(\log n + P_S(n)/n + WD_S(n/f(n)))$$
$$I^a_{S_1}(n) = O(\log n + P_S(n)/n + WI_S(n/f(n)))$$

The average bounds are not turned into worst-cases by working on the individual blocks, but by an application of theorem 1.

Proposition 2. The bounds given in proposition 1 are valid as worst-case bounds.

Proof

Always configure V initially as before. We no longer enforce the constraints strictly. Insertions and deletions are processed as before and turn out to satisfy the requirements for weak updates, as is easily seen. Apply theorem 1. □

The "logarithmic" method is explained well in Bentley [2]. In its simplest form, V is partitioned into blocks of size 2^i for appropriate values of i. Insertions are processed in very much the same way as in Vuillemin's binomial queues [23]. Clearly $Q_{S_1}(n) = O(\log n.Q_S(n))$ and one can show that $I^a_{S_1}(n) = O(\log n.P_S(n)/n)$ (with $n = N$). Overmars and van Leeuwen [14] proved that deletions can be incorporated in the method, by allowing each block of size 2^i a leeway of about 2^{i-1} in shrinkage and 2^i in growth. Van Leeuwen and Maurer [21] introduced the strategy of not cleaning up each block every time, but reconfiguring the set as a whole after the number of deletions has exceeded some fraction (say $\frac{1}{2}$) of its current size, with the elements marked as deleted included in the count. Overmars and van Leeuwen [17] finally showed how the logarithmic method

could be adapted to obtain the $O(\log n.P_S(n)/n)$ time for insertions and the time for deletions as worst-case rather than merely as average bounds.

The logarithmic method is just one example of a family of methods in which the set is partitioned into a collection of blocks of exponentially increasing sizes (see e.g. Overmars and van Leeuwen [15]). Mehlhorn and Overmars [12] systematically explored the possible trade-offs in update- and query-time, as the blocks and their sizes are varied. Let h be an increasing integer function with $h(n) \geq 2$. Let $2^k \leq n < 2^{k+1}$. V will be partitioned into (i) a large block of 2^k points and (ii) a set of blocks $B_j (j \geq 0)$ containing the remaining points, where each B_j contains some multiple of b^j points with $b = h(k)$. Writing $n - 2^k = \sum_{j\geq 0} a_j b^j$ with $0 \leq a_j < b$ (uniquely), there actually are two ways to organize the B_j:

Method I : B_j is a static structure of $a_j b^j$ points,

Method II: B_j consists of a_j subblocks $B_{j1}, \ldots, B_{ja_j}$ containing b^j points each.

Only insertions are initially considered. The "large block" is an effective way to keep a large chunk away from the active part of the structure. Only when n becomes 2^{k+1} are all points collected and is a new "large block" built, to last until n becomes 2^{k+2}. In the meantime, insertions only affect the small blocks. Let j_0 be the smallest integer with $a_{j_0} + 1 < b$ (a_j's as above). An insertion is processed as follows:

Method I : take $B_0, \ldots, B_{j_0}$ together with the new point and build them into one, new B_{j_0} with $1 + \sum_{j<j_0} (b-1)b^j + a_{j_0} b^{j_0} = (a_{j_0}+1)b^{j_0}$ points,

Method II: take all $B_{j,1}$ with $j < j_0$ together with the new point an build them as one block with $1 + \sum_{j<j_0} (b-1)b^j = b^{j_0}$ points that is added as $B_{j_0,a_{j_0}+1}$ to B_{j_0}.

Given these routines, the detailed average time analysis is relatively straightforward ([12]). The same techniques as used in Overmars and van Leeuwen [17] for the logarithmic method can be applied to incorporate deletions and to achieve worst-case instead of average bounds. It yields:

Proposition 3. Given a (static) structure S for a decomposable searching problem Q and a smooth and non-decreasing integer function g with $1 \leq g(n) \leq n$, there is a structure S_1 for Q with

$$
\begin{aligned}
Q_{S_1}(n) &= O(g(n).Q_S(n)) \\
D_{S_1}(n) &= O(\log n + P_S(n)/n + WD_S(n)) \\
I_{S_1}(n) &= \begin{cases} O(\log n/\log \frac{g(n)}{\log n}.P_S(n)/n) & \text{if } g(n) = \Omega(\log n) \\ O(g(n).n^{\frac{1}{g(n)}}.P_S(n)/n) & \text{if } g(n) = O(\log n) \end{cases}
\end{aligned}
$$

The equal blocks - and logarithmic methods can be combined into one dynamization that gives the best possible, general worst-case bounds (Overmars and van Leeuwen [18]). Optimality follows from Mehlhorn [11]. Let f and g be as above.

Theorem 2.([18]). Given a structure S for a decomposable searching problem Q, there is a structure S_1 for Q with

$$Q_{S_1}(n) = O((f(n) + g(n))\ Q_S(n/f(n)))$$
$$D_{S_1}(n) = O(\log n + P_S(n)/n + WD_S(n/f(n)))$$
$$I_{S_1}(n) = \left\{ \text{<as in proposition 3>} \right.$$

By choosing different f and g (e.g. $f(n) = g(n) = \log n$), theorem 2 can be tuned to many different applications.

3. Applications

Theorems 1 and 2 are the main tools for obtaining general dynamizations. For more specific types of (decomposable) searching problems there are additional results.

Definition. A decomposable searching problem Q is called a decomposable counting problem if and only if for any $V_1 \subset V$ one can synthesize $Q(x,V\backslash V_1)$ from $Q(x,V)$ and $Q(x,V_1)$ at only nominal extra costs.

Theorem 3.([17]) Given a (static) structure S for a decomposable counting problem Q, there is a structure S_1 for Q with

$$Q_{S_1}(n) = O(\log n.Q_S(n))$$
$$D_{S_1}(n) = O(\log n.P_S(n)/n)$$
$$I_{S_1}(n) = O(\log n.P_S(n)/n)$$

Definition.([14]) A decomposable searching problem Q, together with a fixed static structure S for solving it, is called an MD-searching problem if and only if it is possible to build S by spending $O(n \log n)$ time on sorting its n points and O(n) additional steps on its further construction.

Using the techniques of Overmars and van Leeuwen [14], [17] one can show the following result (note that one log-factor of the insertion time has disappeared):

Theorem 4. Given the (static) structure S of an MD-searching problem Q, there is a structure S_1 for Q with

$$Q_{S_1}(n) = O(\log n.Q_S(n))$$
$$D_{S_1}(n) = O(\log n + WD_S(n))$$
$$I_{S_1}(n) = O(\log n)$$

The dynamization results can be very powerful. In a number of instances they avoid the need for complex ad-hoc constructions and lead mechanically to good worst-case bounds.

a) range searching.

Bentley et.al. introduced a number of intrinsically static data structures for the range searching problem, including "range trees" (Bentley [2]) which achieve $P_S(n) = O(n \log^{d-1} n)$ and $Q_S(n) = O(\log^d n)$ for $d \geq 2$, where we ignore the time to print answers in the estimate for Q_S. Range trees allow weak deletions in time $WD_S = O(\log^{d-1} n)$ (cf. van Leeuwen and Maurer [21]). From theorem 2 we conclude the existence of a dynamic data structure S_1 for range searching with $Q_{S_1}(n) = O(\log^{d+1} n)$, $D_{S_1}(n) = O(\log^{d-1} n)$ and $I_{S_1}(n) = O(\log^d n)$. It is interesting to compare this with the first solution to dynamic range searching, due to Willard [24] (see also Lueker [9]), which achieves $P_S(n) = O(n \log^{d-1} n)$, $Q_S(n) = O(\log^d n)$, $D_S(n) = O(\log^d n)$ and $I_S(n) = O(\log^d n)$. The dynamization apparently traded a log-factor between Q_S and D_S. While the dynamization leads to a data structure with many "independent" static subblocks, Willard's structure is a giant BB[α]-tree with recursive structures attached to the internal nodes and a complex set of rebalancing rules. It can be shown that Willard's trees allow weak deletions in $WD_S = O(\log^{d-1} n)$ time by just removing links to an element everywhere it occurs. Applying theorem 1 to Willard's structure yields a dynamic data structure S_1 for range searching with the best upperbounds currently known: $Q_{S_1}(n) = O(\log^d n)$, $D_{S_1}(n) = O(\log^{d-1} n)$ and $I_{S_1}(n) = O(\log^d n)$.

b) rectangle searching.

In many rectangle searching problems it is required that a collection of segments (as intersections of the rectangles with a scanning line) is maintained such that one can always effectively determine e.g. to what segments a given point belongs. Bentley [1] (also Bentley and Wood [3]) introduced the "segment tree" for it, which has $P_S(n) = O(n \log n)$ and $Q_S(n) = O(\log n)$. Segment trees allow weak deletions with $WD_S(n) = O(\log n)$. By theorem 2 we conclude the existence of a dynamic structure S_1 for segment trees with $Q_{S_1}(n) = O(\log^2 n)$, $D_{S_1}(n) = O(\log n)$ and $I_{S_1}(n) = O(\log^2 n)$. Analysing the searching problem in detail, Edelsbrunner [4] found a direct method to maintain dynamic segment trees with $Q_{S_1}(n) = O(\log n)$, $D_{S_1}(n) = O(\log n)$ and $I_{S_1}(n) = O(\log n)$. Further applications of dynamization results to rectangle problems appear in Edelsbrunner and Maurer [5].

c) nearest neighbor searching in the plane.

Known static solutions use data structures with $P_S(n) = O(n \log n)$ and $Q_S(n) = O(\log n)$ (Shamos [20], Kirkpatrick [8]). Applying theorem 2 with $f(n) = g(n) = \sqrt{n}$ and $WD_S(n) = P_S(n)$ yields a dynamic structure S_1 for it with $Q_{S_1}(n) = O(\sqrt{n}.\log n)$, $D_{S_1}(n) = O(\sqrt{n}.\log n)$ and $I_{S_1}(n) = O(\log n)$. (Compare van Leeuwen and Maurer [21]). Overmars [13] recently obtained a (dynamic) data structure for nearest neighbor searching, based on the Voronoi diagram (cf. Shamos [20]), which achieves $Q_S(n) = O(\log n)$, $D_S(n) = O(n)$

and $I_S(n) = O(n)$. Applying theorem 2 to this structure yields a remarkable dynamic structure S_1 for the problem with $Q_{S_1}(n) = O(\sqrt{n.\log n})$, $D_{S_1}(n) = O(\sqrt{n \log n})$ and $I_{S_1}(n) = O(\log n)$.

Convex hull searching does not fit in the theory of decomposable searching problems as presented. Dynamic solutions for it and for related problems appear in Overmars and van Leeuwen [16]. See also Gowda [6] and Gowda and Kirkpatrick [7].

4. Epilogue

The theory of dynamization has provided a collection of powerful results and techniques for "mechanically" obtaining dynamic solutions from static solutions of decomposable searching problems, with competitive worst-case time bounds. Clearly, dynamization can never be expected to give a final answer if the detailed properties of the searching problem at hand are not explored further. Thus, the theory of dynamization merely serves as a tool-box for the algorithm designer in his search for efficient methods. Indications are that a suitable notion of "decomposability" often is the key to a fast dynamic structure. In this respect one can expect that any problem that admits a solution by "divide-and-conquer" can be dynamized in a non-trivial manner. This statement is made precise in Overmars [13].

5. References

1 Bentley, J.L., Algorithms for Klee's rectangle problems, unpubl. notes, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, 1977.

2 Bentley, J.L., Decomposable searching problems, Inf. Proc. Lett. 8 (1979) 244-251.

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

4 Edelsbrunner, H., Dynamic data structures for orthogonal intersection queries, Bericht 59, Inst. f. Informationsverarb., TU Graz, Graz, 1980.

5 Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects, Bericht 60, Inst. f. Informationsverarb., TU Graz, Graz, 1980.

6 Gowda, I.G., Dynamic problems in computational geometry, M.Sc.Thesis, Dept. of Computer Science, University of British Columbia, Vancouver, 1980.

7 Gowda, I.G. and D.G. Kirkpatrick, Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures, in: Proc. 18th Annual Allerton Conf. on Communication, Control and Computing, 1980.

8 Kirkpatrick, D.G., Optimal search in planar subdivisions, preprint, Dept. of Computer Science, University of British Columbia, Vancouver, 1979.

9 Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep., Dept. of Computer Science, University of California at Irvine, Irvine, 1978.

10 Maurer, H.A. and Th. Ottmann, Dynamic solutions of decomposable searching problems, Bericht 33, Inst. f. Informationsverarb., TU Graz, Graz, 1979.

11 Mehlhorn, K., Lower bounds on the efficiency of static to dynamic transforms of data structures, preprint, Fachbereich 10, University of Saarland, Saarbrücken, 1980.

12 Mehlhorn, K. and M.H. Overmars, Optimal dynamization of decomposable searching problems, preprint, Fachbereich 10, University of Saarland, Saarbrücken, 1980 (to appear in Inf. Proc. Lett.).

13 Overmars, M.H., Dynamization of order decomposable set problems, Techn. Rep. RUU-CS-80-9, Dept. of Computer Science, University of Utrecht, Utrecht, 1980 (to appear in J. Algor.).

14 Overmars, M.H. and J. van Leeuwen, Two general methods for dynamizing decomposable searching problems, Computing 26 (1981) 155-166.

15 Overmars, M.H. and J. van Leeuwen, Some principles for dynamizing decomposable searching problems, Inf. Proc. Lett. 12 (1981) 49-54.

16 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 (revised as Techn. Rep. RUU-CS-81-3, to appear in J. Comp. Syst. Sci.).

17 Overmars, M.H. and J. van Leeuwen, Dynamization of decomposable searching problems yielding good worst-case bounds, in: P. Deussen (ed.), Theoretical Computer Science (5th GI-Conf.), Lect. Notes in Comp. Sci. 104, Springer Verlag, Berlin, 1981, pp. 224-233.

18 Overmars, M.H. and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, Techn. Rep. RUU-CS-80-10, Dept. of Computer Science, University of Utrecht, Utrecht, 1980 (to appear in Inf. Proc. Lett.).

19 Saxe, J.B. and J.L. Bentley, Transforming static data structures into dynamic structures, Proc. 20th Ann. IEEE Symp. Found. of Comp. Sci., Mayaguez and Rio Piedras, Puerto Rico, 1979, pp. 148-168.

20 Shamos, M.I., Computational geometry, Ph.D. Thesis, Dept. of Computer Science, Yale University, New Haven, 1978 (to be published by Springer Verlag).

21 van Leeuwen, J. and H.A. Maurer, Dynamic systems of static data structures, Bericht 42, Inst. f. Informationsverarb., TU Graz, Graz, 1980.

22 van Leeuwen, J. and D. Wood, Dynamization of decomposable searching problems, Inf. Proc. Lett. 10 (1980) 51-56.

23 Vuillemin, J., A data structure for maintaining priority queues, C.ACM 21 (1978) 309-315.

24 Willard, D.E., The super B-tree algorithm, TR-03-79, The Aiken Comput. Lab., Harvard University, Cambridge (USA), 1979.