TRANSFORMING SEMI-DYNAMIC DATA STRUCTURES INTO DYNAMIC STRUCTURES

Mark H. Overmars

RUU-CS-81-10

June 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

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

TRANSFORMING SEMI-DYNAMIC DATA STRUCTURES INTO DYNAMIC STRUCTURES

Mark H. Overmars

Technical Report RUU-CS-81-10

June 1981

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

TRANSFORMING SEMI-DYNAMIC DATA STRUCTURES INTO DYNAMIC STRUCTURES*

Mark H. Overmars

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

Abstract. Data structures are called semi-dynamic if they support updates (insertions and/or deletions) in some weak sense only. It is shown that semi-dynamic data structures can be transformed into fully dynamic structures without essential loss in efficiency of queries and updates. A number of applications is given to demonstrate the power of the notions and techniques presented, including a structure for range queries that is better than any structure known before.

---

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

1. Introduction

Searching problems have received considerable attention during the past few years. A searching problem is a problem in which some question is asked about an object x (called the query object) with respect to a set of objects (usually called points) V. The following two examples of searching problems are of prime interest to us: (i) MEMBER SEARCHING in which we want to know whether or not the query object x is an element of the set V, and (ii) RANGE SEARCHING in which we want to know what elements (or how many) of a multi-dimensional pointset V lie within a given range x (i.e., a rectilinear oriented hyper-rectangle). Solving a searching problem consists of devising a data structure S to represent the set of objects V such that queries with different query objects x can be answered efficiently. Such a data structure S can be static or dynamic. A structure is called static if it is once built for some pointset and from that moment on is only used to answer queries. A structure is called dynamic if it also supports insertion and/or deletions of points in the set. Recently, much work has been done in devising general techniques for turning static data structures for searching problems into dynamic structures. This process is called DYNAMIZATION. Usually, these dynamization techniques are applicable to searching problems with some special property. A first such class of searching problems, named "decomposable searching problems" was given by Bentley [2] who devised a method for turning static solutions for decomposable searching problems into half-dynamic structures (i.e. structures that only support insertions) with good average update time bounds. Much work has been done to extend the results of Bentley [2]. The most general, and in fact optimal, dynamization scheme for decomposable searching problems was very recently given by Overmars and van Leeuwen [10]. Another class of searching problems, named "order decomposable set problems", for which a general dynamization method exists was given by Overmars [8].

In this paper we will not use a property of the searching problem itself, but a property of the data structure known for the searching problem, to be able to dynamize it.

Definition 1.1. Deletions and/or insertions are called "weak" if the routines to carry them out on an admissible structure of n points merely guarantee that after αn deletions ($\alpha < 1$) and/or βn insertions the query time on the resulting structure of m points is no more than a factor of $k_{\alpha\beta}$ worse than the query time of a structure of m points on which no weak updates have been performed, for some constant $k_{\alpha\beta}$ depending solely on α and β.

Weak updates can be viewed as updates that do disturb balance but not to drastically.

Definition 1.2. A data structure S is called semi-dynamic if it allows for weak insertions and weak deletions.

Notation. Let S be a data structure for some searching problem Q, containing n points.

$Q_S(n)$ = the time required to perform a query on S,
$P_S(n)$ = the time required to build S,
$S_S(n)$ = the storage required for S,
$I_S(n)$ = the time required to perform an insertion in S (when applicable),
$D_S(n)$ = the time required to perform a deletion in S (when applicable),
$WI_S(n)$ = the time required to perform a weak insertion in S (when applicable),
$WD_S(n)$ = the time required to perform a weak deletion in S (when applicable).

All bounds are assumed to be worst-case bounds. We assume that all functions are nondecreasing, P_S and S_S are at least linear and all functions are smooth. (A function F is called smooth if $F(O(n)) = O(F(n))$.)

The notion of weak updates was first introduced in Overmars and van Leeuwen [10]. It was shown that semi-dynamic data structures could be transformed into fully dynamic structures, but this result was only used for dynamizing decomposable searching problems. In section 2 we will recall this dynamization result. Next, in section 3, we will show that this theorem has a number of applications on its own (i.e., not using the notion of decomposability). We will consider three important applications. First, we show that all structures known for member searching can very easily be transformed into structures that allow for weak deletions. Hence, once we have a half-dynamic data structure for member searching, we can transform it into a fully dynamic structure without essential loss in query and update time. Secondly, we concentrate on range searching. It is shown that structures known for range searching (Lueker [6], see also Willard [11]) can be adapted to allow for weak deletions. It will lead to a remarkable structure for d-dimensional range searching yielding a query time of $O(\log^d n)$ (+ the number of reported answers), an insertion time of $O(\log^d n)$ and a deletion time of only $O(\log^{d-1} n)$. As a third example we consider quad-trees (Finkel and Bentley [5]) and k-d trees (Bentley [1]). Both structures do allow for updates but only good expected time bounds can be obtained. Recently, Overmars and van Leeuwen [9] modified quad-trees into pseudo quad-trees (k-d trees into pseudo k-d trees) that do allow for insertions and deletions in

$O(\log^2 n)$ average time. We will show that one can perform weak deletions in pseudo quad-trees (pseudo k-d trees) in $O(\log n)$ worst-case time. It leads to a structure with $O(\log^2 n)$ average insertion and $O(\log n)$ worst-case deletion time.

2. The main theorem

Semi-dynamic data structures can be transformed into fully dynamic data structures without essential loss in efficiency of query answering.

Theorem 2.1. [10] 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 taken 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 then $n_0 + \frac{1}{6}n_0 = \frac{7}{6}n_0$ 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

$$w + t\lceil P_S(n_0)/\tfrac{1}{6}n_0 \rceil + d_2 WD_S(\tfrac{7}{6}n_0) + i_2 WI_S(\tfrac{7}{6}n_0) \geq$$
$$\geq d_1 WD_S(\tfrac{7}{6}n_0) + i_1 WI_S(\tfrac{7}{6}n_0) + d_2 WD_S(\tfrac{7}{6}n_0) + i_2 WI_S(\tfrac{7}{6}n_0)$$

, where the right-hand side accounts for the maximum of time required to process all buffered transactions. It follows that $w + t.\lceil P_S(n_0)/\frac{1}{6}n_0 \rceil \geq P_S(n_0) + w - \lceil s\lceil P_S(n_0)/\frac{1}{6}n_0 \rceil$, thus $(s + t)\lceil P_S(n_0)/\frac{1}{6}n_0 \rceil \geq P_S(n_0)$. Since $s \leq \frac{1}{6}n_0$ (by inspecting the first phase), we certainly have $s + t \leq \frac{1}{6}n_0$ and MAIN must be able to take over within the number of updates claimed. Since the new MAIN started at size n_0, the $\frac{1}{6}n_0$ 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 $\frac{1}{2}n_1 + \frac{1}{6}n_0 \leq \frac{3}{4}n_1$ updates, assuming its initial size was n_1. Further details are easy. □

It easily follows from the proof that the $O(P_S(n)/n)$ work need not be paid for deletions when they are clean (i.e., when $WD_S = D_S$) nor for insertions when they are clean (i.e.,when $WI_S = I_S$). In some applications the work for building a new MAIN is much less when we use information from OLDMAIN. For example it might be possible to get points ordered in some way (see section 3).

3. Applications

Theorem 2.1. has a number of applications. Especially weak deletions are important. The following lemma is useful in determining whether deletions are weak.

Lemma 3.1. Deletions that do not increase the query time are weak.

Proof

When we perform, on an admissable structure of n points, αn ($\alpha < 1$) such deletions and βn (weak) insertions we know that the query time is bounded by $k_\beta Q((1 + \beta)n)$ for some constant k_β depending on β only. The number of points currently in the set is $m = (1 + \beta - \alpha)n$. $k_\beta Q((1 + \beta)n) = k_\beta Q(\frac{1+\beta}{1+\beta-\alpha} m) =$ $= O(k_\beta Q(m))$ because $\frac{1+\beta}{1+\beta-\alpha} > 0$ and Q is smooth. Hence there is a constant $k_{\alpha\beta}$ depending on α and β only, such that the query time is bounded by $k_{\alpha\beta} Q(m)$. □

As will follow from the examples below, weak deletions often are deletions that do not increase the query time.

Member searching

A lot of data strcutures have been devised for solving the member searching problem, yielding a query time of O(log n). Most of these structures are dynamic yielding O(log n) update time bounds. (See Olivie [7] for an overview of known structures.) Data structures for member searching can easily be adapted to allow for weak deletions. To each set-element in the structure we add a boolean that tells whether or not the element is present. To perform a query with an element x we search for x in the structure. If it is not present the answer to the query is "no". Otherwise we check the boolean to see whether or not x is present. To perform a weak deletion of an object p we search for p in the structure and set the boolean to false. It follows that $WD(n) = O(Q(n))$.

Theorem 3.2. Given a half-dynamic structure S for member searching, there exist a structure S' for member searching with

$$Q_{S'}(n) = O(Q_S(n))$$
$$I_{S'}(n) = O(I_S(n))$$
$$D_{S'}(n) = O(Q_S(n) + P_S(n)/n)$$

As stated above, in general $Q_S(n) = I_S(n) = O(\log n)$. Moreover the construction of a new MAIN structure in general takes only O(n), because we have the points ordered in OLDMAIN. In that case $D_{S'}(n) = O(Q_S(n))$.

A wellknown data structure for member searching is the AVL-tree. AVL-trees are dynamic yielding an update time of O(log n). AVL-trees are kept balanced by means of local rotations. An important property of AVL-trees is that insertions take at most one rotation. This is especially relevant when the structure is augmented by associating information to internal nodes. Often this information has to be recomputed when rotations take place. Unfortunately, deletions in AVL-trees can cause local rotations at all nodes along the path

towards the deleted point. Using the technique described above this is no longer the case.

Another example are αBB-trees as introduced by Olivié [7]. For most α, O(log n) insertion algorithms are known but efficient deletion methods do not exist. The above method remedies this problem.

In a number of applications one already has a pointer to the appropriate node when a point needs to be deleted. In this case, the weak deletion time is bounded by O(1) and hence, we get a structure S' yielding $Q_{S'}(n) = O(\log n)$, $I_{S'}(n) = O(\log n)$ and $D_{S'}(n) = O(1)$.

Range searching

Efficient dynamic data structures for the range searching problem have recently been devised independently by Lueker [6] and Willard [11]. Their structure yields a query time of $O(\log^d n)$ and an update time of $O(\log^d n)$, where d is the dimension of the problem. Each point of the set occurs $O(\log^{d-1} n)$ times in the structure. One can perform a weak deletion in the structure by removing a point at all its occurences. Hence a weak deletion takes $O(\log^{d-1} n)$ time, provided we know the occurences of the point. To locate these occurences efficiently, we add to the structure a dictionary DICT in which we maintain this information for all points currently in the set. Insertions we perform in the usual way, except that we have to update DICT as well, and weak deletions we perform by searching for the point in DICT, following the pointers to its occurences in the structure and removing the point there. Next we delete the point from DICT. In this way we get a structure S with $Q_S(n) = O(\log^d n)$, $I_S(n) = O(\log^d n)$ and $WD_S(n) = O(\log^{d-1} n)$. It can be shown (see [6, 11]) that the structure can be built in $P_S(n) = O(n \log^{d-1} n)$. We can now apply theorem 2.1. to get a structure for range searching that is better than any known structure.

Theorem 3.3. There exists a structure S for d-dimensional range searching such that

$$Q_S(n) = O(\log^d n) \text{ (+ the number of answers)}$$
$$I_S(n) = O(\log^d n)$$
$$D_S(n) = O(\log^{d-1} n)$$

Quad- and k-d trees

Quad-trees were introduced by Finkel and Bentley [5] as a suitable data structure for answering various types of queries about sets of points in multi-

dimensional space. It is possible to update quad-trees, but only good expected time bounds exist. Recently Overmars and van Leeuwen [9] modified quad-trees into pseudo quad-trees. We will describe here only the 2-dimensional case. Hence we are given a two dimensional pointset V. A pseudo quad-tree of V is built in the following way. An arbitrary point p, somewhere in between the points of the set is chosen. This point divides the plane into four quadrants and hence, the set into four subsets. p becomes the root of the pseudo quad-tree and the four subsets become its four sons. In each quadrant we again choose a point that splits the quadrant into four subquadrants etc. We repeat the splitting until each subquadrant contains at most one point of the set. The points become the leaves of the pseudo quad-tree. See figure 1 for an example of a pseudo quad-tree (p_1, ..., p_{12} are the points from the set and h_1, ..., h_5 are the chosen splitting points.) Overmars and van Leeuwen [9] show that one can maintain a pseudo quad-tree in $O(\log^2 n)$ average update time.

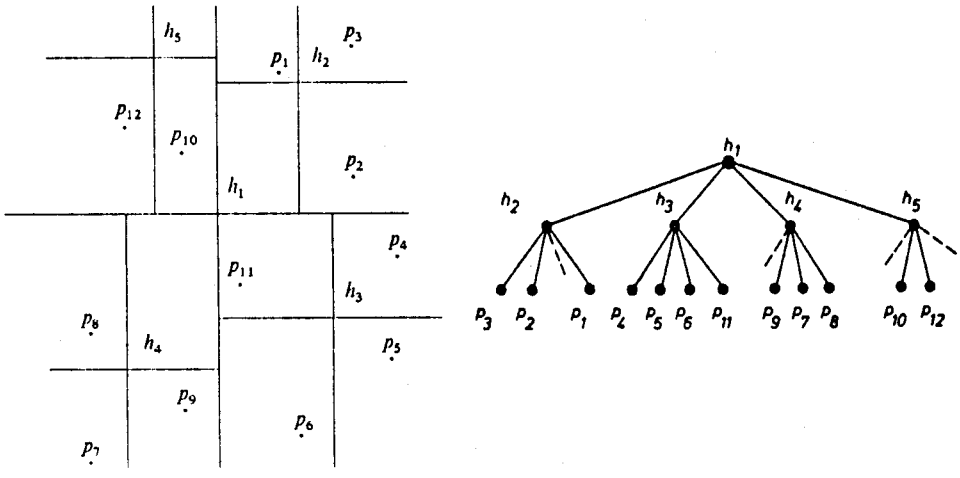

Fig. 1

It is possible to perform weak deletion in pseudo quad-trees in O(log n) time by just searching for the point and removing it. Such a deletion does not increase the depth and, as the query time for most queries depends on the depth, it does not increase the query time. There is only one problem. We cannot immediately apply theorem 2.1. because we assumed there that we know beforehand how much time a buffered insertion will take. In this case we only have an average insertion time bound and hence some insertions might be very "expensive" while others are cheap. It can be proven that when we perform m

insertions on a new built pseudo quad-tree of n points, the total time needed will be bounded by $O(m.\log^2(n+m))$. (This average result is stronger than the one stated in [9].) Refering to the proof of theorem 2.1., it follows that the $i_1 + i_2$ insertions in BUF will take a total of at most $(i_1 + i_2)\log^2(\frac{7}{6}n_0)$ time and hence, if we do with each insertion $\log^2(\frac{7}{6}n_0)$ work, the new MAIN will be ready in time. This leads to the following result:

Theorem 3.4. Pseudo quad-trees can be maintained at the cost of $O(\log^2 n)$ average insertion time and $O(\log n)$ worst-case deletion time.

Similar results can be obtained for k-d trees (Bentley [1]).

Decomposable searching problems

Dynamization of decomposable searching problems is an area of research that received considerable attention during the past few years. A number of techniques have been devised for transforming static data structures for decomposable searching problems into dynamic structures. Most of these techniques can be formulated much more easily as transforms that turn static structures into semi-dynamic structures (in which both insertions and deletions are weak). Next, applying theorem 2.1. yields dynamic structures for the problems. The application of theorem 2.1. for obtaining dynamizations for decomposable searching problems is described in Overmars and van Leeuwen [10].

Other applications

Already a number of other applications of theorem 2.1. are known. For example theorem 2.1. can be used to obtain efficient deletion methods for structures for rectangle intersection searching as described in Edelsbrunner [3] and Edelsbrunner and Maurer [4]. The notion of weak updates and theorem 2.1. might be useful in obtaining dynamic data structures for a number of other searching problems.

4. References

[1] Bentley, J.L., Multidimensional binary search trees used for associated searching, Comm. of the ACM 18 (1975) 509-517.

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

[3] Edelsbrunner, H., Dynamic data structures for orthogonal intersection queries, Bericht 59, Inst. f. Informationsverarbeitung, TU Graz, 1980.

[4] Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects, Bericht 60, Inst. f. Informationsverarbeitung, TU Graz, 1980.

[5] Finkel, R.A. and J.L. Bentley, Quad-trees: a data structure for retrieval on composite keys, Acta Informatica 4 (1974) 1-9.

[6] Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep. #129, Dept of Inform. and Computer Science, University of California, Irvine, 1978.

[7] Olivié, H., A study of balanced binary trees and balanced one-two trees, Ph.D.Thesis, Dept. of Mathematics, University of Antwerp (UIA), 1980.

[8] Overmars, M.H., Dynamization of order decomposable set problems, Techn. Rep. RUU-CS-80-9, Dept of Computer Science, University of Utrecht, 1980. (To appear in J. of Algorithms.)

[9] Overmars, M.H. and J. van Leeuwen, Dynamic multi-dimensional data structures based on quad- and k-d trees, Techn. Rep. RUU-CS-80-2, Dept. of Computer Science, University of Utrecht, 1980.

[10] 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, 1980. (To appear in Inform. Proc. Lett.).

[11] Willard, D.E., The super B-tree algorithm, TR-03-79, Aiken Comput.-Lab., Harvard University, Cambridge, 1979.