The reconstruction of dynamic data structures

Michiel Smid, Leen Torenvliet, Peter van Emde Boas and Mark Overmars

RUU-CS-86-21
november 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

THE RECONSTRUCTION OF

DYNAMIC DATA STRUCTURES

by

Michiel H.M. Smid
Leen Torenvliet
Peter van Emde Boas

Department of Computer Science
University of Amsterdam

and

Mark H. Overmars

Department of Computer Science
University of Utrecht

abstract : In this paper the reconstruction problem for dynamic data structures is studied: Given a dynamic data structure, construct a shadow administration, which is stored in the safe secondary memory, such that in case of calamity the original data structure can be reconstructed. Some basic solutions to this problem are given. In the case of set problems, whose answers can be merged in linear time, a shadow administration is given, which is asymptotically more efficient than the shadow structures of the basic solutions.

November 1986

I. Introduction

I.1. Motivation

One of the main problems in the theory of data structures is the design of structures and algorithms, which can answer efficiently specific questions about large amount of data. One can think, e.g., of the efficient data structures to solve range queries in computational geometry.

These structures are used for the execution of a key-to-address-transform: a key uniquely identifies an object, and the algorithm that acts on the data structure, transforms this key into a physical memory address, where the additional information about the corresponding object is stored. In case of a range query, the key describes the range, and the algorithm gives the address of the set of objects which lie inside the range.

The efficiency of such data structures is often caused by the following properties:
(i) A large part of the data structure (often the entire structure) can be stored in main memory (in general, this is not possible for the complete data).
(ii) The data structures are dynamic, i.e., it is possible to adapt them to the alterations of the database.

Both nice properties pass away if we take the imperfection of computer systems into account. After a system crash, or as a result of errors in software, the contents of the main memory can get lost. Another case, in which the data structure in main memory can get lost, is the regular termination of the application program. In case of an application which is executed on a system which is also used by other persons, the copy of the data structure in core will get lost between two runs of the application program.
In both cases (system crash or regular termination) the data structure has to be reconstructed from the information stored in secondary memory.

There are several possibilities to solve this problem.
(i) Keep in secondary memory the set of objects represented by the data structure. After an update, just the information about the update is transported to secondary memory. The data structure is reconstructed from the set of objects in secondary memory.
(ii) After each update, a copy of the complete data structure is transported to secondary memory.
(iii) Periodically, a copy of the data structure is transported to secondary memory. After an intermediate update, the information about the update is transported.

The first solution has as a disadvantage that reconstruction from the set of objects stored in secondary memory can take as much time as building the original data structure from scratch. However, after an update, just the information about the update has to be transported. In the second solution, the entire data structure has to be transported after each update, whereas reconstruction

takes little time.
So we have a trade-off situation between reconstruction time and update time. The third solution shows that intermediate solutions exist between the extreme situations in this trade-off.

This leads us to the following problem:
Try to improve upon the above trade-off situation by using properties of the data structure in question. Ideally, the information which has to be transported after an update is local and compact. Furthermore, the time needed to reconstruct the data structure from this compact backup information is much lower than the time needed to build the data structure from scratch.

In 1983, Torenvliet and van Emde Boas [10] gave an example in which an improvement, as suggested above, is possible. It concerned there a hash-trie index structure, suggested by Litwin, in which a data base, having a set of lexicographically ordered strings as keys, distributed over a number of buckets, can be searched efficiently. It was shown that in this case the amount of storage for the backup information can be reduced by a factor 1/3, whereas the reconstruction time is reasonable.

It seems that this problem, which certainly arises in practical applications, never has been investigated (with the exception of the above example). The reason for this phenomenon can just be guessed. Maybe the gap between theory and practice is concerned here. Maybe there are just few applications where the data structure is too big to be saved frequently, but small enough to be stored in main memory. Another possibility is the fact that often no asymptotic improvement is possible (but see Chapter III). In the above example the constants in the complexity are improved, but not the asymptotic order.

The problem arises, e.g., in the following areas:
(i) The theory of data bases.
(ii) Computational geometry. Since in this area often data structures are used requiring more than linear space, it might be possible to improve asymptotically upon the storage requirements.
(iii) A system in which several processors on distinct times execute distinct tasks, and which communicate through message passing, might be even more sensitive for crashes than a uniprocessor system. To protect a calculation against the failure of processors, so-called checkpoints are built in on several places of the calculation. If such a checkpoint is reached, the complete state of all processors, and the interconnection pattern, is transported to secondary memory. If the system crashes, then the calculation can be continued at the last reached checkpoint. It is clear that much time and space can be saved by efficiently storing the information from each processor.

This paper, which is just an introduction to the reconstruction problem, is organized as follows.
In Section I.2, searching problems, data structures and their complexity are introduced.
In Section I.3, complexity measures are introduced, in which the efficiency of solutions to the reconstruction problem will be expressed.
In Chapter II, the three basic solutions, as suggested above, are worked out, and their complexity are given.
Chapter III considers solutions for the case of data structures, solving set problems, whose answers can be merged efficiently. It appears, that if the answers can be merged in linear time, then an asymptotic improvement of the basic solutions is possible.
Finally, in Chapter IV, some concluding remarks are given.

I.2. Searching problems

A searching problem is a problem in which a question (called a query) is asked about an object (the query object) with respect to a set of objects. More formally, let T_1, T_2 and T_3 be sets, and let $P(T_2)$ be the power set of T_2 (i.e., the set consisting of all subsets of T_2). A searching problem is a mapping $PR : T_1 \times P(T_2) \rightarrow T_3$.
An example is the member searching problem : Given a query object x and a set V, decide whether or not x is an element of V. In this case $T_1 = T_2$, $T_3 = \{\text{true, false}\}$, and $PR(x,V) = (x \in V)$.
Given a set V in $P(T_2)$, a solution to the searching problem PR consists of a data structure DS representing V, that answers queries (i.e., PR(x,V) for x in T_1) efficiently.
If the set V is given beforehand and does not change, then the data structure is called static. This in contrary to dynamic data structures, where elements can be inserted and deleted.
Let n=|V| be the cardinality of the set V. The complexity of data structure DS is given by the following functions of n: $P_{DS}(n)$, the preprocessing time required to build DS; $S_{DS}(n)$, the amount of storage required to store DS; $Q_{DS}(n)$, the time required to answer a query using DS; and in case DS is a dynamic data structure, $I_{DS}(n)$, the time required to insert an element into DS; and $D_{DS}(n)$, the time required to delete an element from DS. If the data structure DS is obvious from the context, then we write just P(n), S(n), etc.
As an example, a dynamic solution to the member searching problem, where T_1 and T_2 are the integers, is the AVL tree. Here P(n)=O(n log n), S(n)=O(n), and Q(n),I(n),D(n) are O(log n).
There exists extensive literature on searching problems. The reader is referred, e.g., to Overmars [7], and to the references given there.

Let PR : T_1 x $P(T_2)$ $\rightarrow$ T_3 be a searching problem, and let PA be a preprocessing algorithm, that builds for each V in $P(T_2)$ a data structure, that solves PR(x,V), $x \in T_1$.
Two data structures DS_1 and DS_2 are called equivalent, if they both represent the same set in $P(T_2)$, and if they both can be built by the algorithm PA (so if the represented set is V, then we can use either DS_1 or DS_2 to compute PR(x,V)).
Note that (in general) the data structure built by PA depends on the order in which the set V is read.

I.3. The reconstruction problem

The reconstruction problem can be formulated as follows.
Let DS be a dynamic data structure, solving the searching problem PR. We assume that the structure DS is entirely kept in core. To be able to reconstruct a data structure equivalent to DS in case of calamity, we keep in the safe secondary memory information about DS, the so-called shadow administration.
The problem is how to construct such a shadow administration, requiring of course little storage and which can be updated efficiently, and how to design a fast reconstruction algorithm.

Suppose e.g. that the data structure is an AVL tree T, representing the set V (a set of integers). Then we can take as a shadow administration an array which contains V in sorted order. So we do not have to store the pointers of the tree T in the secondary memory.
Reconstruction of an AVL tree equivalent to T can be done in O(n) time. However, the difficulty in this case is how to update the shadow administration efficiently after an insertion or a deletion.

As we saw in the preceding section, we use two measures to indicate the complexity of a data structure: time and space. For the complexity of a shadow administration we shall use also the space measure, but we shall use two different time measures.

(i) The first measure is the computing time. This is just the usual time measure, which is used, e.g., to indicate the time required to build a tree, to sort integers, etc.
(ii) Since the data structures we consider are dynamic, the shadow administration changes continuously. Therefore, in order to update the shadow structure, we have to transport data from core to secondary memory, and possibly vice versa.
Let T(S(n)) be the time required to transport an amount of S(n) data from core to secondary memory, or vice versa. We assume that $T(S(n)) = c_1 + c_2 S(n)$, where c_1 is the time to start up the transport process, and $c_2 S(n)$ is the proper transport time (c_1 and c_2 are constants). Since c_2 will be very small, compared to the constants in computing time, we shall treat the transport time as a

separate time measure.
We have to make one remark here. Often, the shadow administration will contain parts of the data structure itself. To update the shadow administration, one of the things we do is copy these parts of the data structure, and transport them to secondary memory. We assume that copying is part of the transport process itself. Hence the time required to copy parts from the data strucure is already contained in the transport time.

Now the complexity of a shadow administration will be given by the amount of storage it requires, the time to update it, and the time to reconstruct a data structure equivalent to the original one from it, where the times are split into two parts as described above.

II. Some basic solutions

In this chapter we give some basic solutions to the reconstruction problem.

Let PR : T_1 x $P(T_2) \rightarrow T_3$ be a searching problem, and let V be a subset of T_2. Let DS be a dynamic data structure solving PR(x,V) for x in T_1, with performances P(n), S(n), Q(n), I(n) and D(n), where n=|V| (cf.Section I.2).

We assume that the set V requires G(n) space to store (e.g., G(n)=O(d× n) if V is a set of n vectors in d-dimensional space). Furthermore, we assume that G(n)=Ω(n) (to store V, we have to store all its elements).

II.1. Low storage administrations

The first idea is to write to the secondary memory all information about the updates: if element p is inserted or deleted, we transport to the secondary memory, p together with information whether p is inserted or deleted. So an update of the shadow administration requires O(G(1)) transport time. Suppose we have a sequence of i insertions and d deletions (we start with an empty structure), and suppose that after these N=i+d updates a system crash occurs. To reconstruct the data structure, we transport the sequence of updates to core, we determine in some way the elements which are present in V, and we build (from scratch) a data structure from these elements. This reconstruction algorithm requires O(N× G(1)) transport time, and O(N log N + P(n)) computing time (n is the cardinality of V).

Furthermore, the shadow administration requires O(N× G(1)) storage.

The above method has as a disadvantage that we have to determine the elements which are present in V. Therefore we should like to store the elements of V in a search structure.
We assume that each element of T_2 contains a primary key (which is one word long) from an ordered set. So we can order the set V, according to this key, in O(n log n) time.
Now we take as a shadow administration the information about the updates as above, and a balanced binary search tree, which contains in its nodes the primary keys of the elements of V. Also, each node contains the location in the secondary memory of the corresponding element of V.

The storage required by this shadow administration is $O(N \times G(1))$ for the sequence of updates, and O(n) for the tree (again N is the number of updates, and n is the cardinality of V). So the total storage requirement is $O(N \times G(1))$.
Suppose element p is inserted into V. Then the shadow administration is updated as follows. Transport p to the secondary memory; transport the search tree to core; insert the primary key of p, together with the location of p at secondary memory, into this tree; and transport the new tree to the secondary memory.
This insertion procedure takes O(G(1)+O(n)) transport time, and O(log n) computing time.
The deletion procedure is similar. Suppose p is deleted from V. Then we transport the search tree to core; we delete from this tree the node which contains the primary key of p; and we transport the resulting tree to the secondary memory. This procedure takes O(n) transport time, and O(log n) computing time. Note that element p remains stored in secondary memory. We do not assume that we can use the location where p is stored, to store a new inserted element.
After a system crash, we transport the shadow administration to core, and we build from scratch a data structure from the set V. So reconstruction requires $O(N \times G(1))$ transport time, and O(P(n)) computing time. Note that this computing time can be much smaller, since we have the elements of V ordered with respect to their primary keys (in many preprocessing algorithms we first order the elements according to some order).

The above shadow administrations require little storage. However, the reconstruction times are large. In the next section we shall consider the opposite case: large storage and low reconstruction time.

II.2. A large storage administration

A very simple solution is to take as a shadow administration just a copy of the data structure DS. So the shadow structure requires S(n) storage.
After an update, we discard the old shadow administration, we copy the data structure, and we transport the copy to the secondary memory. This takes O(S(n)) transport time.
After a system crash, we transport the copy of the data structure to core. So reconstruction takes O(S(n)) transport time.

II.3. A mixed method

Let k be a fixed positive integer. We make a shadow administration as follows.
After every k-th update, we copy the data structure, and we transport the copy to the secondary memory (the old copy is discarded). Between update ik and (i+1)k (i=0,1,2,...) we transport the information about the update to the secondary memory, as in the beginning of Section II.1.
So each k-th update requires O(S(n)) transport time; all other updates require O(G(1)) transport time.
Suppose we have a sequence of N updates. Let q and r be integers, such that $N=qk+r$, $0 \leq r < k$.
Let m be the number of elements in the set V after the qk-th update.

Then our shadow administration requires $O(S(m)+r \times G(1))$ storage.
Reconstruction of the data structure after these N updates can be done as follows.
First transport the shadow administration to core. Then we have the data structure as it was after the qk-th update. Next we perform the last r updates.

Suppose, for simplicity, that the last r updates are insertions. Then the reconstruction algorithm takes $O(S(m) + r \times G(1))$ transport time, and $\sum_{i=1}^{r} I(m+i)$ computing time.

Now compare the performances of this shadow administration with those of Section II.2:

(i) The storage requirements are of the same order.

(ii) The average update time in this section is much lower than that of Section II.2.

(iii) In this section we have an increase in reconstruction time of $\sum_{i=1}^{r} I(m+i)$ computing time.

What value we choose for k depends on the goal we want to achieve: a low reconstruction time at the cost of a higher average update time, or a low average update time at the cost of a higher reconstruction time.

III. Order decomposable set problems

III.1. Introduction

In this chapter we restrict ourselves to a subclass of the searching problems, the so-called set problems. A set problem is a mapping PR : $P(T_2) \rightarrow T_3$ (we can consider this as a searching problem PR(x,V) by using a dummy query object x).
An example is the maximum problem: Given a subset V of a linearly ordered set, determine its maximal element.
In particular, we shall consider set problems, whose answers can be merged efficiently.

A set problem PR : $P(T_2) \rightarrow T_3$ is called C(n)-order decomposable, if there is an order ORD on T_2, and a function $\square : T_3 \times T_3 \rightarrow T_3$, such that for each set $V = \{p_1 < p_2 < ... < p_n\}$, ordered according to ORD, and for each i, $1 \leq i < n$, we have

$$PR(\{p_1,p_2,...,p_n\}) = \square (PR(\{p_1,...,p_i\}) , PR(\{p_{i+1},...,p_n\})),$$

where the function $\square$ takes C(n) time to compute.

For example, the maximum problem is O(1)-order decomposable (note that in this case we do not need the order ORD). A more interesting example is the set problem "compute the Voronoi diagram of a set of n points in the plane". As was shown by Shamos [9], this problem is O(n)-order decomposable (cf. Section III.4).

By using the divide-and-conquer technique, we see that the answer to a C(n)-order decomposable set problem can be computed in O(ORD(n) + F(n)) time, where ORD(n) is the time needed to order n points according to ORD, and F(n) is the solution of the recurrence $F(n) = 2\,F(n/2) + O(C(n))$. If we do not need the order ORD, then we take ORD(n)=O(1).
It is well known that the above recurrence has as a solution $F(n) = O(\sum_{i=0}^{\log n} 2^i\, C(n / 2^i))$.

III.2. A dynamic data structure

Let PR be a C(n)-order decomposable set problem. We make a dynamic data structure that solves this problem (cf. Overmars [7]).

Let V be a set of n points for which we want to solve PR. For simplicity we make the following assumptions. The set V can be stored in O(n) space, and the answer PR(V) takes O(C(n)) storage (these assumptions are not essential, cf. Overmars [7]).

First we order the set $V=\{p_1<p_2<...<p_n\}$ according to ORD (if we do not need the order ORD, then we order V lexicographically). We may assume that this can be done in O(n log n) time.

Partition V into sets $V_1=\{p_1,...,p_{f(n)}\}$, $V_2=\{p_{f(n)+1},...,p_{2f(n)}\}$,..., where $f(n)=\lceil n / \log n \rceil$.
We store each set V_i in a balanced binary search tree T_i. Let r_i be the root of T_i. We say that r_i represents the set V_i. The roots r_i are ordered according to $r_1<r_2<r_3<...$
Now we store these roots in an augmented BB(α) leaf search tree T, where $2/11< \alpha \leq 1-1/2\sqrt{2}$.
Let k be a node of T. Suppose the subtree of T with root k has $r_i,r_{i+1},...,r_j$ as its leaves. Then node k represents the set $V_i \cup V_{i+1} \cup ... \cup V_j$.
We store in node k:
(i) a pointer to its father (if k has a father);
(ii) pointers to its left and right son (if these sons exist);
(iii) the largest value (according to ORD) in the subtree of its left son;
(iv) the answer to PR of the subset of V represented by node k.

Let $A(n) = \sum_{i=0}^{\log \log n} 2^i C(n/2^i)$, and $B(n) = \sum_{i=0}^{\log \log n} C(n/2^i)$.

Then $A(n) = O(C(n))$ if $C(n)=\Omega(n^{1+\varepsilon})$ for some $\varepsilon>0$; $A(n) = O(C(n / \log n) \log n)$ if $C(n)=O(n^{\varepsilon})$ for some $0<\varepsilon<1$; and $A(n) = O(C(n) \log \log n)$ otherwise.

Also, $B(n) = O(C(n))$ if $C(n)=\Omega(n^{\varepsilon})$ for some $\varepsilon>0$; and $B(n) = O(C(n) \log \log n)$ otherwise.

The proof of the following theorem can be found in Overmars [7].

Theorem 1 : The performances of the above data structure are given by

$S(n) = O(n + A(n))$;
$P(n) = O(n \log n + F(n / \log n) \log n + A(n))$, where F(n) is as in Section III.1;
$Q(n) = O(1)$;
I(n) and D(n) are both $O(\log n + F(n / \log n) + B(n))$.

If we apply this theorem to the Voronoi diagram problem (then $C(n)=O(n)$, $F(n)=O(n \log n)$, $ORD(n)=O(n \log n)$), then we get a dynamic data structure with performances $S(n)=O(n \log \log n)$, $P(n)=O(n \log n)$, $Q(n)=O(1)$, $I(n)=O(n)$, $D(n)=O(n)$.
In the next section we shall see that there is a shadow administration for the Voronoi diagram structure, which is asymptotically more efficient than those of Chapter II.

III.3. A shadow administration

Consider again the dynamic data structure of the preceding section. We take the following shadow administration for this data structure:

(i) The trees T_i;
(ii) The answers $PR(V_i)$ to the set problem PR of the sets V_i.

Theorem 2 : This shadow administration has performances:
(i) The storage requires $S'(n) = O(n + C(n / \log n) \log n)$ space.
(ii) An update requires $O(S'(n))$ transport time.
(iii) Reconstruction takes $O(S'(n))$ transport time, and $O(A(n))$ computing time, where $A(n)$ is as in Section III.2.

Proof : (i) Because of our assumptions (cf. Section III.2), the trees T_i together take $O((n / \log n) \log n)=O(n)$ storage, and the answers $PR(V_i)$ take $O(C(n / \log n) \log n)$ storage.
(ii) Since our shadow administration consists of parts of the data structure itself, it can be updated as follows. First, we discard the old shadow structure. Then we copy the relevant parts from the data structure, and we transport these parts to the secondary memory. Hence an update takes $O(S'(n))$ transport time.
(iii) To reconstruct the original data structure, we transport the shadow administration to core. This takes $O(S'(n))$ transport time. We assume that the secondary memory is organized in such a way, that we have the roots of the trees T_i in sorted order. The only thing we have to do is to build the tree T: in every node of T, we copy the answers to PR of its sons, and we merge them. Because of our assumptions, this takes $O(\sum_{i=0}^{\log \log n} 2^i C(n / 2^i))$ computing time for the entire tree T.

□

In the rest of this section we assume that $C(n)=\theta(n)$.
According to Theorem 1, there is a dynamic data structure that solves PR, with performances $S(n)=O(n \log \log n)$, $P(n)=O(n \log n)$, $Q(n)=O(1)$, $I(n)=O(n)$, and $D(n)=O(n)$.
Now Theorem 2 gives

Theorem 3 : There is a shadow administration for the dynamic data structure of a $\theta(n)$-order decomposable set problem, with performances:
(i) The storage takes $O(n)$ space.
(ii) An update takes $O(n)$ transport time.
(iii) Reconstruction takes $O(n)$ transport time, and $O(n \log \log n)$ computing time.

Compare these performances to those of the second method of Section II.1:
(i) There the storage takes $\theta(N)=\Omega(n)$ space, where N is the number of updates.
(ii) An update takes O(n) transport time, and O(log n) computing time.
(iii) Reconstruction takes $\theta(N)=\Omega(n)$ transport time, and O(n log n) computing time.

So the shadow administration of this section is asymptotically more efficient than that of Section II.1 (note that the shadow structures of Sections II.2 and II.3 both require O(n log log n) space).

III.4. Examples

In this section we shall give some examples of $\theta(n)$-order decomposable set problems. As we saw in the preceding section, there are shadow administrations for these problems, which improve asymptotically on those of Chapter 2.

A first example is the set problem "compute the Voronoi diagram of a set of n points in the plane". Voronoi diagrams are used to solve the nearest neighbor searching problem: Given a set V of n points in the plane, and a query point x in the plane, determine a point in V, that is nearest to x, with respect to the Euclidean distance.
The Voronoi diagram of V is obtained by dividing the plane in areas of constant answer, i.e., if x and y are points in the same area, then they have the same nearest neighbor in V. If p is a point in V, then the area in which the answer to the nearest neighbor query is p, is a (possibly unbounded) convex polygon containing p. The total number of edges of these polygons is O(n). Hence the Voronoi diagram of V can be stored in O(n) space (cf. the assumptions in Section III.2).
It was shown by Shamos [9], that the Voronoi diagram problem is $\theta(n)$-order decomposable, where ORD is the order according to x-coordinate. Later, Kirkpatrick [5] showed that we do not need this order: Voronoi diagrams of two arbitrary disjoint sets can be merged in linear time. Hence we can apply Theorem 3 to get an efficient shadow administration.

As stated, we use Voronoi diagrams to solve the nearest neighbor searching problem: To determine the nearest neighbor of a point x, we determine which Voronoi area contains x. With this area, a point p in V is associated, which is the nearest neighbor of x.
Kirkpatrick [6] has shown that a data structure, requiring O(n) storage, can be built in O(n) time from the Voronoi diagram of the set V, such that the nearest neighbor of a given point can be determined in O(log n) time. Since the Voronoi diagram of a set of n points can be built in O(n log n) time (cf. Section III.1), we see that there is a static data structure that solves the nearest neighbor searching problem in the plane, with performances S(n)=O(n), P(n)=O(n log n), and Q(n)=O(log n).
Now we put this static data structure, together with the structure of Theorem 1 (applied to the Voronoi diagram problem), into one data structure. Then using the above result from Kirkpatrick, it is easy to see that we get a dynamic data structure that solves the nearest neighbor searching

problem, with performances S(n)=O(n log log n), P(n)=O(n log n), Q(n)=O(log n), I(n)=O(n), and D(n)=O(n).
The following theorem shows that there is a shadow administration for this data structure, which improves asymptotically on the results from Chapter II.

Theorem 4 : There is a shadow administration for the dynamic nearest neighbor structure, with performances:
(i) The storage takes O(n) space.
(ii) An update takes O(n) transport time.
(iii) Reconstruction takes O(n) transport time, and O(n log log n) computing time.

Proof : The shadow administration is just the same administration we used in Section III.3, i.e., a shadow structure consisting of the trees T_i and Voronoi diagrams of the sets V_i. Now (i) and (ii) follow from Theorem 3, and (iii) follows from Theorem 3 and from Kirkpatrick [6].

□

The next example is the set problem "compute the convex hull of a set of n points in three dimensional space". As was shown by Preparata and Hong [8], this problem is θ(n)-order decomposable.
A third example is the set problem "find the intersection of a set of n halfspaces in three dimensions". Using a result of Brown [3], it can be shown that this problem is θ(n)-order decomposable.
Finally, according to Edelsbrunner, Overmars and Wood [4], the set problem "compute the view of a set of n line segments in the plane from some fixed direction" is θ(n)-order decomposable.
For all these problems we can apply Theorem 3, to get a shadow administration with the following performances. The storage takes O(n) space. An update requires O(n) transport time.
Reconstruction of the data structure requires O(n) transport time, and O(n log log n) computing time.

IV. Concluding remarks

We finish this paper by giving some concluding remarks, and some ideas that are being worked out presently.

Many dynamic data structures are obtained from static structures by using one of the so-called dynamization techniques (see Bentley [1], Bentley and Saxe [2], Overmars [7]). Of course, these techniques can also be applied to shadow administrations. So the idea is the following. We are given a static data structure DS, that solves some searching problem. Now we make a shadow administration for this static structure. Then we apply a dynamization technique to both the data structure DS and the shadow structure, to get a shadow administration for the dynamic version of the data structure.

In Section II.3, the integer k was a constant. An idea is to let k depend on n. What performances can we get in this way?

In Chapter III, we saw a class of dynamic data structures, for which shadow administrations exist, which are asymptotically more efficient than those of Chapter II. It might be interesting to have more examples of shadow structures with performances that improve asymptotically on the results of Chapter II.
Suppose we have a dynamic data structure that has P(n) preprocessing time, and that requires S(n) storage. Now suppose we have a shadow administration for this data structure, that takes S'(n) storage, such that the original data structure can be reconstructed in R(n) computing time. Then this shadow administration is asymptotically more efficient than those of Chapter II, if e.g. $S'(n)=o(S(n))$, and $R(n)=o(P(n))$.
It is reasonable to assume that $S'(n)=\Omega(n)$ (the shadow administration contains at least the points represented by the data structure). So $S'(n)=o(S(n))$ implies $n=o(S(n))$.
To reconstruct the data structure, we have to walk through S(n) storage, and hence $R(n)=\Omega(S(n))$. So if $R(n)=o(P(n))$, then $S(n)=o(P(n))$.
So shadow administrations with performances, that improve asymptotically on those of the shadow structures of Chapter II, might exist for dynamic data structures that have S(n) storage and P(n) preprocessing time, where $n=o(S(n))$ and $S(n)=o(P(n))$.

References

1. J.L.Bentley, Decomposable Searching Problems, Inform.Proc.Lett. 8 (1979), pp.244-251.
2. J.L.Bentley and J.B.Saxe, Decomposable Searching Problems I : Static to Dynamic Transformations, J.of Algorithms 1 (1980),pp.301-358.
3. K.Q.Brown, Geometric Transforms for Fast Geometric Algorithms, Techn.Rep.CMU-CS-80-101, Dept.of Computer Science, Carnegie-Mellon University, 1980.
4. H.Edelsbrunner, M.H.Overmars, and D.Wood, Graphics in Flatland: A Case Study, in: F.P.Preparata (ed.), Advances in Computing Research, Vol.1, Computational Geometry, J.A.I. Press, London, 1983, pp.35-59.
5. D.G.Kirkpatrick, Efficient Computation of Continuous Skeletons, Proc.20th Annual IEEE Symp.on Foundations of Computer Science, 1979, pp.18-27.
6. D.G.Kirkpatrick, Optimal Search in Planar Subdivisions, SIAM J.Computing 12 (1983), pp.28-35.
7. M.H.Overmars, The Design of Dynamic Data Structures, Springer Lecture Notes in Computer Science, Vol.156, Springer Verlag, 1983.
8. F.P.Preparata and S.J.Hong, Convex Hulls of Finite Sets of Points in Two and Three Dimensions, Comm.of the ACM 20 (1977), pp.87-93.
9. M.I.Shamos, Computational Geometry, Ph.D.Thesis, Dept.of Computer Science, Yale University, 1978.
10. L.Torenvliet and P.van Emde Boas, The Reconstruction and Optimization of Trie Hashing Functions, Proc. 9th International Conf.on Very Large Databases (1983), pp.142-156.