Concatenable structures for decomposable problems

Marc J. van Kreveld and Mark H. Overmars

RUU-CS-89-16
August 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Concatenable structures for decomposable problems

Marc J. van Kreveld and Mark H. Overmars

Technical Report RUU-CS-89-16
August 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

ISSN:0924-3275

Concatenable Structures for Decomposable Problems*

Marc J. van Kreveld Mark H. Overmars

August 21, 1989

Abstract

Given a data structure and an ordering on the objects it contains, we study methods for obtaining a modified data structure that allows for splits and concatenations with respect to that ordering. A general technique will be given, which works for all data structures for decomposable searching problems and order decomposable set problems. Furthermore, the results imply a new method for adding range restrictions to data structures. Applications include e.g. a version of an interval tree that allows for splitting and concatenating on the length of the intervals, a version of the d-dimensional k-d tree that allows for splitting and concatenating on all coordinates, and a data structure on points in the plane that allows for reporting the convex hull of the points in a given query rectangle.

1 Introduction

Data structures are used to answer certain questions about a set of objects efficiently. Many data structures also allow for efficient insertion of new objects in the set and deletion of objects from the set. They are called dynamic data structures. For data structures that are not dynamic, a trivial way to perform an insertion or deletion is to add the new object to the set or remove it from the set, and rebuild the entire data structure. But if many updates are performed, or the data structure stores many objects, this is too time consuming. Research on this topic has resulted in general techniques for making data structures dynamic (see [10] for an extensive treatment of this topic). A number of these techniques make use of the notion of *decomposability*, introduced by Bentley [2]. The main idea is that one doesn't have to keep all objects of the set in one big data structure, but one could also use several

*Authors address: Dept. of Computer Science, University of Utrecht, P.O.Box 80.089, 3508TB Utrecht, the Netherlands. This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

smaller data structures. In this case, an insertion or deletion only requires one of the (smaller) data structures to be rebuilt, thus saving time. A query (question to be answered) must be performed on all data structures, and the several answers composed to one answer for the query on the whole set. Clearly, this only makes sense if the searching problem allows for an efficient composition of answers. This is captured by the following

Definition 1 *A searching problem PR on a set S is called decomposable if and only if for any partition $A \cup B$ of the set S, and any query object x, $PR(x, S) = \Box(PR(x, A), PR(x, B))$ for a constant time computable operator $\Box$.*

For example, member searching is decomposable because if we know whether $x \in A$ and $x \in B$, then we can compute $x \in A \cup B$, with the logical or-funtion as operator. Another example is range searching, where the data structure represents a set of points in the plane, and a range query asks for all points in the set that lie within a given rectangle, the query object. We can simply report all answers in A first, and then report all answers in B. Notice that this is only correct because A and B form a partition of the set, and are thus disjoint. An example of a searching problem that is not decomposable is convex hull searching. Suppose a set of points in the plane is given, and we wish to know whether a given query point lies in the convex hull of the set of points. Given a partition $A \cup B$ of the set, if the query point lies in the convex hull of neither A nor B, we do not know if the query point lies in the convex hull of $A \cup B$.

Inserting and deleting objects is one way to change a set. Other ways include dividing the set in two sets, and uniting two sets to form one larger set. Suppose there is some total ordering $\prec$ on the objects. When a set S of objects is divided in two sets S_1 and S_2, such that S_1 contains all objects of S that are less than or equal to some object m, and S_2 contains all objects of S greater than m, then we say that S is *split* with *splitting object* m into S_1 and S_2. Furthermore, we say that two sets S_1 and S_2 are *concatenated* to a set S, if there exists an object m such that S_1 only contains objects less than or equal to m, S_2 only contains objects greater than m, and S contains all objects of $S_1 \cup S_2$. Note that split and concatenate are inverse operations. We call a data structure that allows for efficient splitting and concatenating a concatenable data structure.

A well-known concatenable data structure is the 2-3 tree, introduced by Hopcroft in [1]. On a 2-3 tree, member queries (among some other queries) can be performed efficiently, together with the set operations insert, delete, split and concatenate. More recently, a concatenable segment tree has been described in [5], which stores a set of intervals and allows for efficient stabbing queries (determining all intervals that contain a given query value), insertions, deletions, splits and concatenations. In [6], a new version of the k-d tree is described, on which range queries, insertions and deletions can be performed, together with splitting and concatenating on both coordinates.

In this paper we will study techniques for obtaining concatenable data structures from existing data structures. To this end, we use a new version of a dynamization technique called the equal block method. This technique applies to all data structures for decomposable searching problems, as described above. We will also show that, using concatenable data structures, range restrictions can be added to query problems. Unlike previous methods for adding range restrictions (see [14, 15]), our method also works for static data structures (and, in fact, makes them dynamic). Some applications of the results we obtain are the following: A concatenable interval tree that supports stabbing queries in $O(\sqrt{n}\log n + k)$ time (where k is the output size), insertions, deletions, splits and concatenations (on any ordering) in $O(\sqrt{n}\log n)$ time. A data structure for rectangular range searching on a set of n points in E^d, which supports queries in $O(n^{1-1/d}\log n + k)$ time (where k is the output size), insertions and deletions in $O(\log n)$ time, and splits and concatenations on all coordinates in $O(n^{1-1/d}\log n)$ time. A data structure for reporting the convex hull in a query rectangle in $O(\sqrt{n\log n}\log n + k)$ time (where k is the output size), on which insertions and deletions take $O(\log^2 n)$ time. All three structures use linear space and can be built in $O(n\log n)$ time.

The remainder of this paper is organized as follows.

Section 2 will present a new version of the equal block method, called the ordered equal block method. We will prove that this method is equally good in making data structures dynamic as the standard equal block method.

In section 3 we introduce the notion of concatenable environment, in which structures can be split and concatenated. The main result of this paper is stated at the end of the section. The result implies a new method for adding range restrictions to decomposable searching problems.

Applications of the results obtained are given in section 4, in which a concatenable version of the interval tree is given. Furthermore, a 2-3 tree storing points in the plane, which is concatenable on both first and second coordinate, is presented.

Section 5 shows that one can make a data structure concatenable on more than one total ordering.

In section 6 we use the result of the previous section to obtain structures for point sets in arbitrary dimensional space that can be split and concatenated on every coordinate. Secondly, a structure that stores points in the plane is obtained, from which the convex hull of the points in a query rectangle can be retrieved.

Finally, in section 7, conclusions and possibilities for further research are given.

Notation: The following notation will be used in the remaining sections. $I_T(n)$ is the time needed to insert an object in a data structure T storing n objects. Likewise, $D_T(n)$ is the deletion time, $S_T(n)$ is the split time, $C_{T_1,T_2}(n)$ is the concatenation time, $Q_T(n)$ is the query time, $P_T(n)$ is the construction time of T, and $M_T(n)$ is the amount of storage used by T. A function f is called *restricted* if and

only if there is a constant c such that for all $n > c$, $f(n) \in [0, n]$ and the derivate $f'(n) \in [0, 1]$. Typical restricted functions are n^{ε} and $n^{\delta}(\log n)^c$ (ε, δ, c constants, $1 \leq \varepsilon \leq 1$, $0 < \delta < 1$).

2 The ordered equal block method

In this section the ordered equal block method, a new version of the known equal block method for obtaining dynamic structures for decomposable searching problems, is introduced. This new version has two advantages. Firstly, for some query problems and some orderings, our method yields better query time bounds, and, secondly, it is suited for splitting and concatenating, whereas the original method is not.

The (unordered) equal block method was introduced by van Leeuwen and Maurer [7], van Leeuwen and Wood [8], and Maurer and Ottmann [9]. The idea is as follows. Partition a given set of objects in a number of smaller, equal-sized subsets, and for each subset, construct a (static) data structure of the type that solves the query problem. The data structure for a subset is called a block. An update requires to choose an appropriate block, on which the update is performed. This generally involves rebuilding the block with the updated subset. A query is performed on all blocks, and from the answers to the queries on every block, the answer to the query on the whole set is retrieved (this is possible when the searching problem is decomposable). Generally, having many blocks results in fast updates and slow queries, and having few blocks results in slow updates and fast queries. The trade-off is expressed by some restricted function f, which relates the number of blocks to the number of objects to be represented. It is required that the number of blocks does not become much greater than f (for a reasonable query time), and no block becomes too large (for a reasonable update time). For details we refer to [7, 8, 9, 10].

The above description also holds for the *ordered* equal block method, but additionally, we require that the blocks are mutually ordered with respect to some total ordering on the objects. Thus the objects in one block are not (necessarily) ordered, but the objects in different blocks are. We make this more precise.

Definition 2 *Let f be a restricted function and $\prec$ a total ordering on the objects. A set S of n objects is partitioned with the ordered equal block method into subsets $S_1, S_2, \ldots, S_m$ such that*

1. $f(\frac{1}{2}n) \leq m \leq 2f(n)$;
2. $\forall_{1 \leq i \leq m} : |S_i| \leq 3n/m$;
3. $\forall_{1 \leq i < j \leq m} \forall_{a \in S_i} \forall_{b \in S_j} : a \prec b$.

(Notice the asymmetry of the first constraint.)

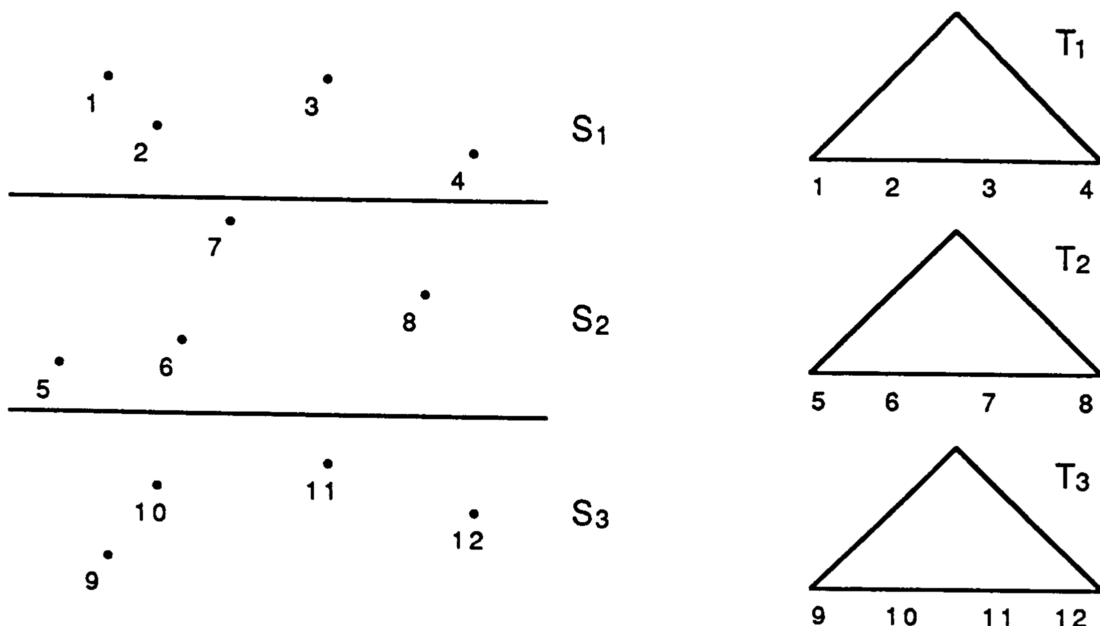

Figure 1: Example of the ordered equal block method

The first constraint guarantees that the number of blocks, representing the sets $S_1, S_2, \ldots, S_m$, will not become too small or too large. The second constraint (in combination with the first) guarantees that no block can become too large. Specifically, $m = \Theta(f(n))$ and $|S_i| = O(n/f(n))$. The third constraint imposes an ordering upon the blocks and will make it possible to split and concatenate using the total ordering $\prec$.

Each set S_i $(1 \leq i \leq m)$ is represented by a block T_i, which is a data structure that solves the query problem. In figure 1 an example is given of an ordered equal block structure of twelve points in the plane, where the points in a block are ordered on first coordinate, the points in different blocks are ordered on second coordinate, and $f(n) = \sqrt{n}$.

Next we consider how updates are performed, such that the set S remains partitioned with the ordered equal block method. To this end, we locate the block on which we must perform the update, so that the third constraint of the definition is not violated. Here the update is performed, which generally requires rebuilding the block. Then we must make sure that the first and second constraints still hold.

The constraints are maintained in the following way. Suppose at some time, the whole structure has been constructed with a set S of n_0 objects, using $m_0 = f(n_0)$ blocks of equal size. Then a mixed sequence of at most $\frac{1}{4}n_0$ insertions and at most $\frac{1}{4}n_0$ deletions are allowed, and then the whole structure is constructed again for the current set of objects. Now consider one block T_i. An update in T_i might cause it to grow, and when it becomes larger than $1\frac{1}{2}n_0/m_0$, its set of objects is divided in

two equal-sized halves (making use of the total ordering), and two new blocks are constructed.

We show that after the construction of the ordered equal block structure for a set S of n_0 objects, partitioned in $m_0 = f(n_0)$ blocks of size n_0/m_0, the first constraint will not be violated within $\frac{1}{4}n_0$ insertions or $\frac{1}{4}n_0$ deletions. Furthermore, any block will satisfy the second constraint, if any newly constructed block is allowed to grow with at most $\frac{1}{2}n_0/m_0$ objects.

Let n be the current number of objects in the set, and m the current number of blocks. Then $\frac{3}{4}n_0 \leq n \leq 1\frac{1}{4}n_0$, and $m \geq m_0$ (blocks are allowed to be empty, but they are not removed until the next reconstruction).

Lemma 1 *There will be at least* $f(\frac{1}{2}n)$ *blocks.*

Proof: The current number of blocks is at least $m_0 = f(n_0) \geq f(\frac{4}{5}n) \geq f(\frac{1}{2}n)$. □

Lemma 2 *There will be at most* $2f(n)$ *blocks.*

Proof: It takes $\frac{1}{2}n_0/m_0$ insertions in a block before it is divided, and after division, the two new blocks contain at most n_0/m_0 objects. Thus at most $\frac{1}{4}n_0/(\frac{1}{2}n_0/m_0) = \frac{1}{2}m_0$ extra blocks can appear before the next reconstruction. The current number of blocks m is at most $1\frac{1}{2}m_0 = 1\frac{1}{2}f(n_0) \leq 1\frac{1}{2}f(1\frac{1}{3}n) \leq 2f(n)$ (the two inequalities follow from the restrictedness of f). □

Lemma 3 *A block will have size at most* $3n/m$.

Proof: By the proof of lemma 2, $m_0 \geq \frac{2}{3}m$. The size of any block will be at most $1\frac{1}{2}n_0/m_0 \leq 1\frac{1}{2}(1\frac{1}{3}n)/(\frac{2}{3}m) = 3n/m$. □

Next we consider how much time it takes to construct an ordered equal block structure. Let S be a set of n objects, and let $\prec$ be the chosen total ordering on the objects. First the objects are sorted with respect to the total ordering, and partitioned in $m = f(n)$ subsets of size n/m (give or take 1). For each subset, a block T is built in time $P_T(n/m)$. Thus all blocks are built in time $m \cdot P_T(n/m) \leq P_T(n)$, because P_T is at least linear. The total construction time is $O(n \log n + P_T(n))$.

To divide a block, we first find the median of the objects it contains with respect to the total ordering $\prec$. Then the objects in the block are divided in two sets, one for the objects less than the median, and one for the other objects. For both sets a new block is constructed. The time taken is $O(n/m) + 2 \cdot O(P_T(n/m)) = O(P_T(n/m))$.

Summarizing, it takes $\frac{1}{4}n$ insertions or $\frac{1}{4}n$ deletions before $O(n \log n + P_T(n))$ time is spent for reconstruction, and it takes $\frac{1}{2}n/m$ insertions in a block before $O(P_T(n/m))$ time is spent for division of that block. This results in $O(\log n + P_T(n)/n)$ amortized reconstruction time per update.

Theorem 1 *Given a (static) data structure T for a decomposable searching problem PR and a restricted function f, then there exists a structure T′ for PR such that*

- $Q_{T'}(n) = O(f(n) \cdot Q_T(n/f(n)))$;
- $I_{T'}(n) = O(\log n + P_T(n)/n + I_T(n/f(n)))$ *amortized;*
- $D_{T'}(n) = O(\log n + P_T(n)/n + D_T(n/f(n)))$ *amortized;*
- $M_{T'}(n) = O(f(n) \cdot M_T(n/f(n)))$;
- $P_{T'}(n) = O(n \log n + P_T(n))$.

Proof: Let S be the set of objects T represents. Choose some total ordering $\prec$ on the objects, and construct an ordered equal block structure T' for S using this total ordering. Furthermore, a *top tree* is added to T', with its leaves associated to the blocks in an ordered way. For the internal nodes of the top tree, correct splitting values are chosen to guide searches.

To perform a query, we traverse the top tree and perform the query on all blocks. There are at most $2f(n)$ blocks, and each block has size at most $3n/f(\frac{1}{2}n)$. Because both f and Q_T are restricted functions, the time a query takes is bounded by $O(f(n) \cdot Q_T(n/f(n)))$, and it takes $O(f(n))$ time to combine the answers.

To perform an insertion, we locate the correct block in $O(\log(2f(n))) = O(\log n)$ time, we insert the object in at most $I_T(2n/f(\frac{1}{2}n)) = O(I_T(n/f(n)))$ time, and reconfiguration takes $O(P_T(n)/n)$ time amortized. The time bound for a deletion is proved similarly.

The amount of memory used by the structure is at most $2f(n) \cdot M_T(2n/f(\frac{1}{2}n))$, plus $O(f(n))$ for the top tree. This is bounded by $O(f(n) \cdot M_T(n/f(n)))$.

The construction time is $O(n \log n)$ for obtaining an ordered equal block partition and constructing a top tree, and $O(P_T(n))$ for building all blocks. □

Remark: The amortized insertion and deletion time bounds of theorem 1 can be changed into worst-case time bounds with the general technique of global rebuilding, if some care is taken. This technique was introduced in [12] by Overmars and van Leeuwen, and is also described in [10].

3 Concatenable environments

We have done some work to obtain an ordered version of the equal block method, to be able to split and concatenate a structure, using the ordering imposed on the structure by the division in blocks. It seems that we can simply concatenate two structures by taking their blocks together, or actually by concatenating their two top trees. Splitting can be done by splitting the top tree, and dividing the block in which the path through the top tree ended. (The top tree will be a 2-3 tree, which

can be split and concatenated efficiently, see e.g. [1].) This only involves dividing and rebuilding one block.

Unfortunately, things are not that simple. There seems to be no way to meet the requirements of the ordered equal block method after every split and concatenate operation, without having to rebuild the entire structure. Splitting tends to result in too few blocks, and concatenating may result in too many blocks.

Notice, for example, that if three equal-sized newly constructed ordered equal block structures (with $f(n) = \sqrt{n}$) are concatenated, then there would be too many blocks, thus reconstruction is needed immediatly. Furthermore, any newly constructed ordered equal block structure can be split in such a way that one of the resulting structures would require immediate reconstruction, caused by too few blocks.

To overcome these balancing problems we introduce a notion called *concatenable environment* (see also [6]). A concatenable environment consists of one or more data structures of some type, which can be split or concatenated with each other. Insertions and deletions are also possible. There is a global notion of balancedness, such that the time bounds for queries, insertions, deletions, splits and concatenations are expressed in the total number of objects in the concatenable environment.

Definition 3 *Given a decomposable searching problem PR, a restricted function f, sets $S_1, \ldots, S_k$ of objects, and a total ordering $\prec$ on the objects. Then a concatenable environment for $PR, f, S_1, \ldots, S_k$ and $\prec$ consists of the following (let N be $|S_1| + \cdots + |S_k|$, the total number of objects):*

- *A structure MAIN, which is the ordered equal block structure for the set $S_1 \cup \cdots \cup S_k$, the function f and the ordering $\prec$. Let the m blocks of MAIN be $B_1, \ldots, B_m$.*
- *Structures $T_1, \ldots, T_k$, representing $S_1, \ldots, S_k$. Every structure T_j $(1 \leq j \leq k)$ consists of m blocks $B_{1j}, \ldots, B_{mj}$, such that an object $x \in S_j$ is in a block B_{ij} if and only if x is in B_i of MAIN $(1 \leq i \leq m)$. Additionally, T_j has a top tree with the non-empty blocks of $B_{1j}, \ldots, B_{mj}$ in the leaves.*
- *With every block B_i of MAIN $(1 \leq i \leq m)$, an extra tree C_i is stored with a reference to every block B_{ij} $(1 \leq i \leq m)$ that is non-empty.*

The above definition states that the set of all objects is partitioned in blocks, and this partition superimposes a partition in blocks on the sets $S_1, \ldots, S_k$. Thus $B_i = B_{i1} \cup \cdots \cup B_{ik}$ for all $1 \leq i \leq m$. See figure 2 for an example of a concatenable environment.

The concatenable environment is balanced by the balancedness of MAIN, and the imposed balancedness on $T_1, \ldots, T_k$.

For the top trees of $T_1, \ldots, T_k$, a tree is chosen that allows for fast insertions, deletions, splits, concatenations and member (neighbor) queries. In a 2-3 tree these operations all take $O(\log n)$ time (see [1]).

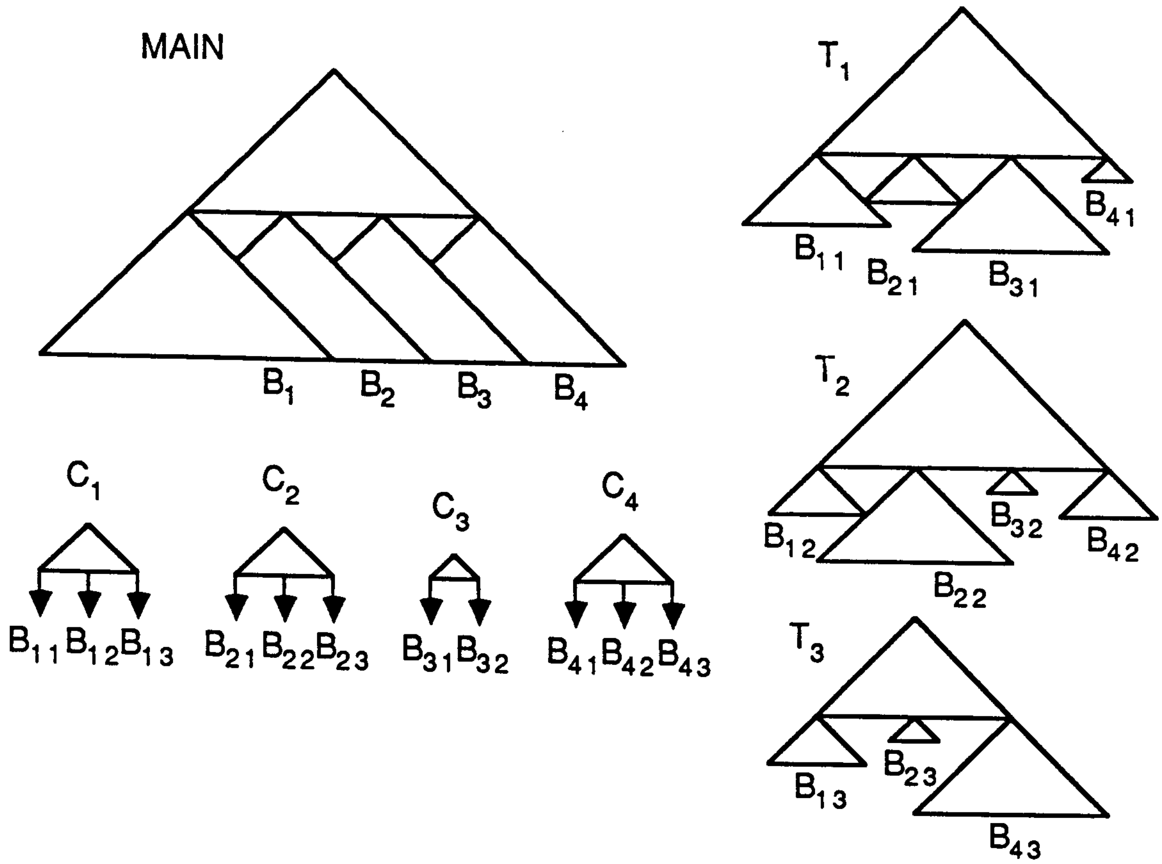

Figure 2: Example of a concatenable environment

Lemma 4 *A concatenable environment with N objects uses $O(f(N) \cdot M_T(N/f(N)))$ space to store, and can be constructed in time $O(N \log N + P_T(N))$.*

Proof: First, notice that the claimed bounds are the same as the bounds for MAIN alone. Furthermore, every block B_j of MAIN contains the same number of objects as $B_{j1}, \ldots, B_{jk}$ together, and as M_T and P_T are at least linear, these blocks cannot more than double the amount of memory space used. Furthermore, both the top trees of $T_1, \ldots, T_k$, and the extra trees $C_1, \ldots, C_m$ will not take more space than the blocks of $T_1, \ldots, T_k$. Consequently, the structure uses at most four times as much memory space as the MAIN structure alone. A similar argument can be used proving the construction time. □

Lemma 5 *A query on a structure in a concatenable environment with N objects takes $O(f(N) \cdot Q_T(N/f(N)))$ time.*

Proof: The time for a query on a structure can never be worse than the time for the same query on MAIN. □

To perform an insertion or deletion in a structure T_j, the blocks B_{ij} and B_i of T_j and MAIN, on which the update must be performed, are located using their top trees. On the two blocks, the update is performed.

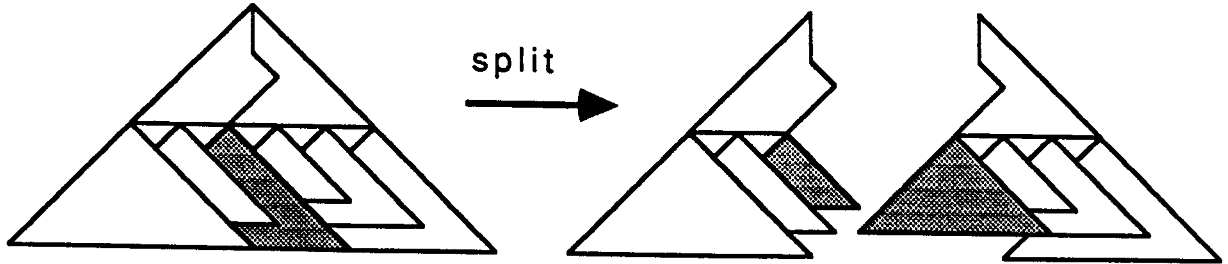

Figure 3: Splitting in a concatenable environment

On MAIN $\frac{1}{4}N_0$ insertions and $\frac{1}{4}N_0$ deletions are again allowed, before the whole concatenable environment is reconstructed. Furthermore, $\frac{1}{2}N_0/m_0$ insertions are allowed on a block B_i of MAIN before it is divided, and consequently, all non-empty blocks B_{ij} are divided (they are found efficiently using the extra tree C_i). It is easy to show that with this policy, the concatenable environment will satisfy the constraints of definition 3. We have not yet considered maintaining the extra trees $C_1, \ldots, C_m$. When an insertion is performed on an empty block B_{ij}, then a leaf is added to the extra tree C_i with a reference to this block, and when a deletion is performed on a block B_{ij} with only one object, then the leaf of C_i containing the reference to that block is removed.

Lemma 6 *An insertion in a structure in a concatenable environment with N objects takes $O(\log N + P_T(N)/N + I_T(N/f(N)))$ time amortized, and a deletion takes $O(\log N + P_T(N)/N + D_T(N/f(N)))$ time amortized.*

As splitting a structure T_j, or concatenating two structures T_j and T_l, does not introduce any new objects in the concatenable environment, such an operation cannot disturb the balance. In fact, MAIN is not visited at all.

A structure T_j is split in the following way. First the top tree of T_j is split with the standard algorithm for splitting 2-3 trees, resulting in two smaller top trees. The splitting path ends in a leaf of the top tree, corresponding to some block B_{ij}. The objects in B_{ij} are split in a brute force way to obtain two sets, one with the objects less than or equal to the splitting value, and one with the greater objects. For both sets a new block is built, and these are associated with two new leaves in the two top trees that resulted from the splitting (see figure 3). C_i is adapted by removing the reference to B_{ij} and adding references to the two newly constructed blocks.

Lemma 7 *A structure in a concatenable environment with N objects can be split with respect to the total ordering in $O(\log N + P_T(N/f(N)))$ time.*

Proof: Splitting the top tree with the standard algorithm takes $O(\log N)$ time, dividing and rebuilding a block of size $O(N/f(N))$ can be done in time bounded by $O(P_T(N/f(N)))$, and the insertions and deletion in the tree C_i take $O(\log N)$ time. □

Two structures T_j and T_l are concatenated in the following way (assume without loss of generality that for all x in T_j and y in T_l, $x \prec y$ holds). Let B_{pj} be the rightmost non-empty block of T_j, and B_{ql} the leftmost non-empty block of T_l. Then $p \leq q$. If $p < q$, then T_j and T_l are concatenated by simply concatenating their top trees with the standard algorithm for concatenating 2-3 trees. If $p = q$, then remove B_{pj} from T_j and remove B_{ql} from T_l. With the objects in these blocks, build one new block and add it as rightmost block to the top tree of T_j. Adapt C_i by removing the references to B_{pj} and B_{ql} and adding a reference to the new block. Then concatenate the two top trees with the standard algorithm.

Lemma 8 *Two structures in a concatenable environment with N objects can be concatenated with respect to the total ordering in $O(\log N + P_T(N/f(N)))$ time.*

Proof: This is analogous to the proof of lemma 7. □

In the following theorem we summarize the results obtained in this section.

Theorem 2 *Given k data structures $T_1, \ldots, T_k$ for a decomposable searching problem PR, which contain N objects together, and a restricted function f, then there exists a concatenable environment E for PR with structures $T'_1, \ldots, T'_k$ such that $(1 \leq j, l \leq k)$*

- $Q_{T'_j}(N) = O(f(N) \cdot Q_T(N/f(N)))$;
- $I_{T'_j}(N) = O(\log N + P_T(N)/N + I_T(N/f(N)))$ *amortized;*
- $D_{T'_j}(N) = O(\log N + P_T(N)/N + D_T(N/f(N)))$ *amortized;*
- $S_{T'_j}(N) = O(\log N + P_T(N/f(N)))$;
- $C_{T'_j,T'_l}(N) = O(\log N + P_T(N/f(N)))$;
- $M_E(N) = O(f(N) \cdot M_T(N/f(N)))$.

Proof: From the lemmas 4, 5, 6, 7 and 8. □

Being able to split and concatenate data structures has an interesting application. In Scholten and Overmars [14] and Willard and Lueker [15], the problem of *adding range restrictions* to searching problems is considered. Assume we have some query problem on a set S of objects. We add to each object p in S some value v_p. The

query is extended with two values a and b and we ask to answer the query over only those objects p in S for which $a \prec v_p \prec b$, with respect to some ordering $\prec$. Splitting and concatenating can be used to answer such queries.

Corollary 1 *Given a data structure T for a decomposable searching problem PR, storing a set S of n objects, a restricted function f, and some total ordering $\prec$ on the objects. Then there exists a dynamic data structure T', storing all objects in S, with some value v_p added to every object p in S. Given a query q of PR and two values a and b, then an answer to q in PR over the points p in S which satisfy $a \prec v_p \prec b$, can be reported efficiently. Specifically,*

- $Q_{T'}(n) = O(f(n) \cdot Q_T(n/f(n)) + \log n + P_T(n/f(n)))$;
- $I_{T'}(n) = O(\log n + P_T(n)/n + I_T(n/f(n)))$ *amortized;*
- $D_{T'}(n) = O(\log n + P_T(n)/n + D_T(n/f(n)))$ *amortized;*
- $M_{T'}(n) = O(f(n) \cdot M_T(n/f(n)))$.

Proof: The structure T' consists of only the MAIN structure of a concatenable environment, or equally, it is the structure obtained by applying the ordered equal block method on the set S, using f and $\prec$. A query is performed by first splitting T' with splitting value a, then with splitting value b, and then performing the query q on the resulting structure, which contains exactly those objects p of S that satisfy $a \prec p \prec b$. Then the original structure is reconstructed by concatenating the three parts that were obtained from the splitting. All bounds follow from the above theorem. □

The methods of Scholten and Overmars [14] and Willard and Lueker [15] generally obtain better query time bounds, but use more memory space. Furthermore, they both start out with dynamic data structures, whereas our method yields dynamic data structures automatically.

4 Applications

4.1 Interval trees

The interval tree was introduced by Edelsbrunner in [3] for solving orthogonal intersection queries. The interval tree is a static structure (although a complicated dynamic version does exist) for storing intervals, and uses linear space to store. It is suited for solving stabbing queries: given a real number (a point on the real line), report all intervals in the structure that contain the point. It is easy to see that the stabbing query problem is decomposable; to report all intervals in $A \cup B$ that

contain a query point, it suffices to report all answers in A first and then all answers in B.

As total ordering on which we intend to split and concatenate we choose the length of the interval. Thus $[b_1 : e_1] \prec [b_2 : e_2]$ if and only if $e_2 - b_2 > e_1 - b_1$, or $e_2 - b_2 = e_1 - b_1$ and $b_2 > b_1$. For the dynamic version of an interval tree T, storing n intervals, $P_T(n) = O(n \log n)$, $M_T(n) = O(n)$, $I_T(n) = D_T(n) = O(\log n)$ amortized, and $SQ_T(n) = O(\log n + k)$, where k is the number of intervals reported.

Theorem 3 *Given m interval trees $T_1, \ldots, T_m$, which contain N intervals together, then there exists a concatenable environment E for solving the stabbing query problem with structures $T'_1, \ldots, T'_m$ such that $(1 \leq j, l \leq m)$*

- $SQ_{T'_j}(N) = O(\sqrt{N} \log N + k)$, *where k is the number of intervals reported;*
- $I_{T'_j}(N)$, $D_{T'_j}(N) = O(\log N)$ *amortized;*
- $S_{T'_j}(N)$, $C_{T'_j,T'_l}(N) = O(\sqrt{N} \log N)$;
- $M_E(N) = O(N)$.

Proof: Choose $f(n) = \sqrt{n}$, then the bounds follow immediatly from theorem 2. □

Corollary 2 *Given a set S of n intervals, then there exists a structure T for the stabbing query problem on S, that only reports those stabbed intervals with length between two given query values, and*

- $Q_T(n) = O(\sqrt{n} \log n + k)$, *where k is the number of intervals reported;*
- I_T , $D_T(n) = O(\log n)$ *amortized;*
- $M_T(n) = O(n)$.

4.2 Two-dimensional 2-3 trees

A 2-dimensional 2-3 tree is a 2-3 tree that stores a collection of points in the plane, where the points are ordered on first coordinate, and with equal first coordinate on second coordinate (lexicographical ordering). On such a tree one can perform member queries, insertions and deletions in $O(\log n)$ time. Also, the split and concatenate algorithms on first coordinate can be done in $O(\log n)$ time. Suppose we want to make this tree concatenable on second coordinate as well. Therefore we assume that no two points have equal second coordinate, and we can apply theorem 2 immediatly. For a 2-dimensional 2-3 tree T, we have $I_T(n)$, $D_T(n)$, $Q_T(n) = O(\log n)$, $P_T(n) = O(n \log n)$ and $M_T(n) = O(n)$.

Theorem 4 *Given m 2-dimensional 2-3 trees $T_1, \ldots, T_m$, which contain N points together, then there exists a concatenable environment E with structures $T'_1, \ldots, T'_m$ such that $(1 \le j, l \le m)$*

- *a member query takes time $MQ_{T'_j}(N) = O(\log N)$;*
- *a range query takes time $RQ_{T'_j}(N) = O(\sqrt{N} \log N + k)$, where k is the number of answers reported;*
- *$I_{T'_j}(N)$, $D_{T'_j}(N) = O(\log N)$ amortized;*
- *$S^1_{T'_j}(N)$, $C^1_{T'_j,T'_l}(N)$, $S^2_{T'_j}(N)$, $C^2_{T'_j,T'_m}(N) = O(\sqrt{N} \log N)$;*
- *$M_E(N) = O(N)$.*

(The upper index of S and C denotes the coordinate on which we split or concatenate.)

Proof: Choose $f(n) = \sqrt{n}$, and the bounds for I, D, S^2, C^2 and M follow immediatly from theorem 2.

To perform a member query, we will not query every block, but instead we will use the top tree to determine the block in which the query point must lie, if it is in the structure. It will take $O(\log N)$ time to find this block, and $O(\log N)$ time to search in it, which proves the member query time bound.

To split (concatenate) a structure T_j on first coordinate, we must split (concatenate) every block in T_j. Every block is a 2-3 tree and can be split (concatenated) with the standard algorithm in $O(\log N)$ time. Also, we need to copy (remove a copy of) the top tree, which has size $O(\sqrt{N})$. Changing the extra trees C_i involves at most three insertions and deletions in all $O(\sqrt{N})$ extra trees. The total time taken is thus $O(\sqrt{N} \log N)$.

Last we prove the range query time. To perform a range query with $[a_1 : b_1] \times [a_2 : b_2]$ on T_j, we split the tree on first coordinate with a_1 and a_2, and on second coordinate with b_1 and b_2. Then all points in the resulting structure are reported, and all pieces are concatenated to form the original structure again. □

Remark: This result can be improved slightly, if we take $f(n) = \sqrt{n/\log n}$, and observe that a block can be rebuilt for a split and concatenation in linear time instead of $O(n \log n)$. This is because the points are already ordered on first coordinate. Now we can split and concatenate on first and second coordinate in $O(\sqrt{n \log n})$ time, and range queries take $O(\sqrt{n \log n} + k)$ time, where k is the number of answers. The structure obtained is basically the same as the divided k-d tree described in [6].

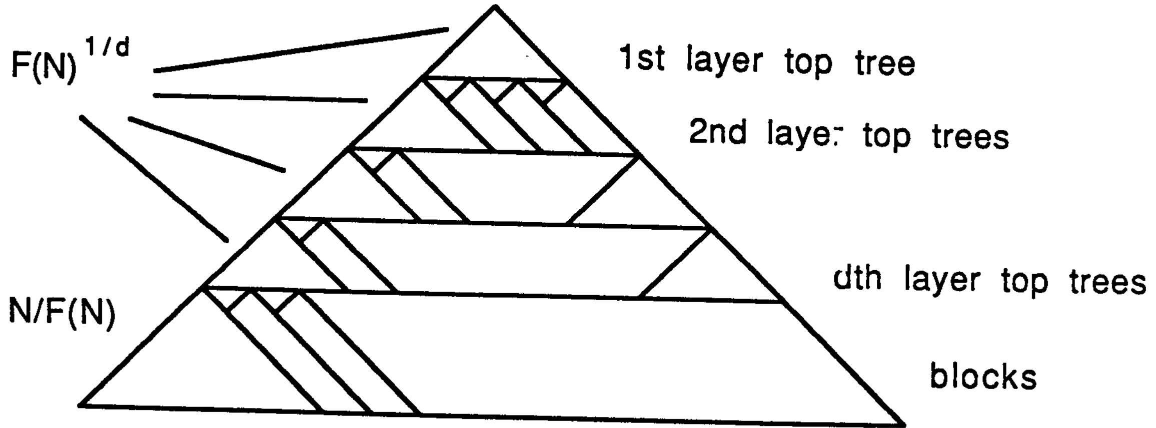

Figure 4: A structure concatenable on d orderings

5 Making structures concatenable on more than one ordering

In the last application of the previous section we obtained a structure that is concatenable on both first and second coordinate. Suppose we want to generalize this structure to higher dimensions, or, we want some other structure that is concatenable on more than one ordering. This can be done by repeatedly making use of theorem 2, since making a structure concatenable results in a new structure, which on its turn can be made concatenable on another total ording, using the same technique. Some care though has to be taken, as we do not have a single structure, but a concatenable environment.

Suppose k structures $T_1, \ldots, T_k$ are given, d total orderings (for a constant d) according to which the structures must be concatenable, and a restricted function F. Let N be the sum of the numbers of objects in $T_1, \ldots, T_k$.

A MAIN structure for the concatenable environment, representing $T_1, \ldots, T_k$, will have one top tree, with every leaf corresponding to some structure, which again has a top tree. This nesting continues until there are d layers of top trees below each other. The leaves of the lowest top trees correspond to the actual blocks, which contain $O(N/F(N))$ objects each. The highest top tree is called the first layer top tree, the top trees corresponding to its leaves the second layer top trees, and the lowest top trees are called the dth layer top trees. A top tree in the cth layer divides the set of objects below it according to the cth ordering. Any top tree of any layer has $O(F(N)^{1/d})$ leaves (see figure 4).

To construct the MAIN structure, functions $f_1, \ldots, f_d$ are computed using the following recurrence: $f_1(N) = F(N)^{1/d}$, $f_{i+1}(N/f_i(N)) = f_i(N)$ for $1 \leq i \leq d-1$. Then the equal block method is applied successively with the functions $f_d, \ldots, f_1$. It is a simple exercise to show that this leads to the structure described above.

In a concatenable environment, structures $T'_1, \ldots, T'_k$ are made, which divide their sets of objects in the same way as MAIN does.

Splitting (Concatenating) on the cth ordering is done by splitting (concatenating) all top trees in the cth layer.

Theorem 5 *Given k data structures $T_1, \ldots, T_k$ for a decomposable searching problem PR, which contain N objects together, d total orderings (for a constant d), and a restricted function F, then there exists a concatenable environment E for PR with structures $T'_1, \ldots, T'_k$ such that ($1 \leq j, l \leq k$)*

- $Q_{T'_j}(N) = O(F(N) \cdot Q_T(N/F(N)))$;
- $I_{T'_j}(N) = O(\log N + P_T(N)/N + I_T(N/F(N)))$ *amortized;*
- $D_{T'_j}(N) = O(\log N + P_T(N)/N + D_T(N/F(N)))$ *amortized;*
- $S_{T'_j}(N) = O(F(N)^{1-1/d} \cdot \log N + F(N)^{1-1/d} \cdot P_T(N/F(N)))$ *on any coordinate;*
- $C_{T'_j, T'_l}(N) = O(F(N)^{1-1/d} \cdot \log N + F(N)^{1-1/d} \cdot P_T(N/F(N)))$ *on any coordinate;*
- $M_E(N) = O(F(N) \cdot M_T(N/F(N)))$.

6 More applications

6.1 d-dimensional 2-3 trees

We wish to obtain a structure that stores points of d-dimensional space such that we can split and concatenate on every coordinate. As 1-dimensional 2-3 trees can be split and concatenated with the standard algorithm, we need to apply theorem 5 with $d-1$ total orderings. Choose $F(n) = n^{1-1/d}$.

Theorem 6 *Given m d-dimensional 2-3 trees $T_1, \ldots, T_m$ that contain N points together, then there exists a concatenable environment E with structures $T'_1, \ldots, T'_k$ such that ($1 \leq j, l \leq m$)*

- *a member query takes time* $MQ_{T'_j}(N) = O(\log N)$;
- *a range query takes time* $RQ_{T'_j}(N) = O(N^{1-1/d} \log N + k)$, *where k is the number of answers;*
- $I_{T'_j}(N)$, $D_{T'_j}(N) = O(\log N)$ *amortized;*
- $S_{T'_j}(N)$, $C_{T'_j, T'_l}(N) = O(N^{1-1/d} \log N)$ *on any coordinate;*
- $M_E(N) = O(N)$.

6.2 Convex hulls in the plane

Thusfar we have only considered decomposable searching problems. However, there is another interesting class of problems that allows for splitting and concatenating: order decomposable set problems. With these problems, the objective is to maintain some set property when insertions and deletions are performed. We will also allow splits and concatenations.

Definition 4 *A set problem PR is called $(C(n), \prec)$-decomposable if and only if there exists a function $\square$ such that for each set of objects $S = \{p_1, \ldots, p_n\}$, ordered according to the total ordering $\prec$,*

$$\forall_{1 \leq i < n}\, PR(\{p_1, \ldots, p_n\}) = \square(PR(\{p_1, \ldots, p_i\}), PR(\{p_{i+1}, \ldots, p_n\})),$$

where $\square$ takes at most $C(n)$ time to compute.

For example, the convex hull problem of a set of n points in the plane is an $(O(\log n), \prec)$-decomposable set problem if $\prec$ is the ordering on first coordinate (or second coordinate), because the convex hull of a set can be constructed from two convex hulls of any ordered partition in $O(\log n)$ time, by computing the supporting lines (also called bridges) (see e.g. [4, 13]).

For order decomposable set problems there exist dynamic structures, which are studied in detail in [10, 11]. From the structure it can easily be seen that $(C(n), \prec)$-decomposable set problems can be split and concatenated with respect to $\prec$ in time $O(C(n) \cdot \log n)$. Structures for order decomposable set problems can be made concatenable on any ordering (one or more) for which the set problem is order decomposable. For the convex hull problem this leads to the following result:

Theorem 7 *There exists a concatenable environment E with m structures $T_1, \ldots, T_m$ for the convex hull problem in the plane, containing n points together, such that the convex hull of the points in T_i $(1 \leq i \leq m)$ can be found in constant time, and $(1 \leq j, l \leq m)$*

- $I_{T_j}(n)$, $D_{T_j}(n) = O(\log^2 n)$ *amortized;*
- $S^1_{T_j}(n)$, $C^1_{T_j,T_l}(n)$, $S^2_{T_j}(n)$, $C^2_{T_j,T_l}(n) = O(\sqrt{n \log n} \log n)$*;*
- $P_E(n) = O(n \log n)$*;*
- $M_E(n) = O(n)$.

Proof: Let T be the structure for the convex hull problem, using the ordering on first coordinate. Choose $f(n) = \sqrt{n/\log n}$, and apply the ordered equal block method on T, using the ordering on second coordinate. Extend the top tree with the same information as used in T. We now have a structure that resembles the concatenable 2-dimensional 2-3 tree, but which stores extra information concerning

the convex hull both in the blocks and in the top tree. From theorem 6.3.6 in [10] and theorem 2 in this paper, the claimed bounds follow. □

Corollary 3 *There exists a structure for the convex hull problem of a set of n points in the plane with update time $O(\log^2 n)$, and in which one can obtain the convex hull of the points that lie in a given axis-parallel query rectangle in time $O(\sqrt{n \log n} \log n + k)$, where k is the size of the output. The structure takes linear space to store and can be built in $O(n \log n)$ time.*

7 Conclusions and further research

In this paper we have devised the ordered equal block method, a new version of the equal block method, which is used for making static structures for decomposable searching problems dynamic. Both methods are equally good, although in some cases the ordered variant is better for queries if the ordering in the top tree can be used for solving the queries. Next we introduced the notion of concatenable environment, in which structures can be split and concatenated using a total ordering on the objects. Also we extended the results to make structures concatenable on more than one ordering. Last we gave some applications in which the obtained results are used. These applications generally resulted in good storage and update bounds, and reasonable split and concatenate bounds at the cost of an increase in query bounds.

There are various possibilities for further research. First, it would be an improvement if split and concatenate bounds could be found that are expressed in the actual number of objects in the structure rather than the sum of the number of objects in all structures. Secondly, only data structures for decomposable searching problems and order decomposable set problems can be made concatenable with the techniques described. One could ask if there is a general technique for arbitrary data structures. As a last item for further research one could investigate specific data structures and orderings to obtain better bounds using special properties of the structure itself, as was done for segment trees in [5].

References

[1] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

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

[3] Edelsbrunner, H., *Intersection problems in computational geometry*, Techn. Rep. F93, Inst. f. Information Processing, TU Graz, 1982.

[4] Edelsbrunner, H. *Algorithms in combinatorial geometry*, Springer Verlag, 1987, pp. 145-147.

[5] van Kreveld, M. J., and M. H. Overmars, Concatenable Segment Trees (extended abstract), in: B. Monien and R. Cori (Eds.) *STACS 89, proceedings*, Lect. Notes in Comp. Science 349, Springer Verlag, pp. 493-504.

[6] van Kreveld, M. J., and M. H. Overmars, *Divided k-d trees*, Techn. Rep. RUU-CS-88-28, University of Utrecht, 1988.

[7] van Leeuwen, J., and H. A. Maurer, *Dynamic systems of static data structures*, Techn. Rep. 42, Inst. f. Informationsverarbeitung, TU Graz, 1980.

[8] van Leeuwen, J., and D. Wood, Dynamization of decomposable searching problems, *Inf. Proc. Lett. 10* (1980), pp. 51-56.

[9] 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, pp. 17-24.

[10] Overmars, M. H., *The design of dynamic data structures*, Lect. Notes in Comp. Science 156, Springer Verlag, 1983.

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

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

[13] Overmars, M. H., and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comp. Syst. Sci. 23* (1981), pp. 166-204.

[14] Scholten, H. W., and M. H. Overmars, General methods for adding range restrictions to decomposable searching problems, *J. Symbolic Computation 7* (1989), pp. 1-10.

[15] Willard, D. E., and G. S. Lueker, Adding range restriction capability to dynamic data structures, *J. ACM 32* (1985), pp. 597-617.