General Methods for Adding Range Restrictions to Decomposable Searching Problems

HANS W. SCHOLTEN AND MARK H. OVERMARS

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

(Received 3 February 1986)

In this paper we consider the problem of adding range restrictions to decomposable searching problems. Two classes of structures for this problem are described. The first class consists of structures that use little storage and preprocessing time but still have reasonable query time. The second class consists of structures that have a much better query time, at the cost of an increase in the amount of storage and preprocessing time. Both classes of structures can be tuned to obtain different trade-offs. First we only describe static structures. To dynamize the structures a new type of weight-balanced multiway tree (the MWB-tree) is introduced that is used as an underlying structure. The MWB-tree might be useful in other applications as well.

1. Introduction

Searching problems arise in many applications in Computer Science and many efficient algorithms and data structures have been developed for a wide variety of these problems. In a searching problem, we ask a question (query) about an object $x$ with respect to a whole set $V$ of objects. A common example of a searching problem is the so-called member searching problem: given a set $V$ of objects and an object $x$, determine whether $x$ belongs to $V$. Two other important examples are the nearest neighbor searching problem and the range searching problem. Let $V$ be a set of $n$ points in $d$-dimensional space. The nearest neighbor searching problem asks for a point in $V$ nearest to a given query point $x$ with respect to some prechosen metric. In the range searching problem the query object $x$ is an axis-parallel hyper-rectangle (range) in $d$-dimensional space. A hyper-rectangle in $d$-dimensional space can be represented as $[A_0 : B_0] \times [A_1 : B_1] \times \cdots \times [A_d : B_d]$. We now ask for all points $x = (x_0, \ldots, x_{d-1})$ in $V$ that lie in the range, i.e. with $A_0 \leq x_0 \leq B_0$ and $\ldots$ and $A_{d-1} \leq x_{d-1} \leq B_{d-1}$. Let $Q(x, V)$ denote the answer to a searching problem $Q$ with query object $x$ over the set $V$.

**Definition 1.1.** A searching problem $Q$ is called decomposable if and only if

$$Q(x, V) = \square [Q(x, A), Q(x, B)]$$

for any partition $A \cup B = V$ and any query object $x$, where $\square$ takes $O(1)$ time to compute.

Many searching problems are decomposable. For example, range searching is decomposable with $\square = "\text{union}"$, nearest neighbor searching is decomposable with $\square = "\text{minimal distance}"$ and member searching is decomposable with $\square = "\text{or}"$. Decomposability enables us to split the set in subsets, and derive the answer to a query over the whole set out of the answers to the same query over the subsets, at only nominal
extra cost. Bentley (1979) was the first to make this important observation and to use it for designing a general dynamization method for decomposable searching problems. Many other methods have been designed since. (See Overmars, 1983 for an overview.)

In this paper we consider the problem of adding range restrictions to decomposable searching problems. This problem was first investigated by Bentley (1979) and Bentley & Saxe (1979). Adding a range restriction means associating a new parameter with every object in the set \( V \). Queries will now be restricted to those objects that have this new parameter in a certain given range.

**Definition 1.2.** Let \( Q \) be a searching problem over a set \( V \). To add a range restriction to \( Q \) we associate a parameter \( k_p \) to each point \( p \) in the set \( V \). The new searching problem \( QR \) now becomes:

\[
QR(x, [A : B], V) = Q(x, \{ p \in V | A \leq k_p \leq B \}).
\]

Addition of range restrictions occurs in a number of cases. Suppose we have a nearest neighbor searching problem in 2-dimensional space, in which the points represent cities. Now assume that with each city we are given its population. Instead of asking the city nearest to a certain query point we ask for the city with population between, say, 100 000 and 200 000 that is nearest to the query point. Hence, we have added a range restriction to the nearest neighbor searching problem. Another example is the \( d \)-dimensional range searching problem itself. It is in fact the addition of range restrictions to the \((d - 1)\)-dimensional range searching problem.

Some notations are useful in comparing data structures.

**Definition 1.3.** Let \( S \) be a data structure for a searching problem, in which the set \( V \) contains \( n \) points.

\[
\begin{align*}
Q_S(n) &= \text{the time needed to perform a query on } S, \\
P_S(n) &= \text{the time needed to build } S \text{ (preprocessing time),} \\
I_S(n) &= \text{the time needed to perform an insertion on } S, \\
D_S(n) &= \text{the time needed to perform a deletion on } S, \\
M_S(n) &= \text{the amount of storage (memory) needed for } S, \\
U_S(n) &= \text{the time needed to perform an update on } S.
\end{align*}
\]

All bounds are worst-case bounds. For amortized bounds we add the superscript \( a \). We assume that all functions are non-decreasing and smooth and the \( P_S \) and \( M_S \) are at least linear.

Bentley (1979) showed how a static data structure \( S \) for a decomposable searching problem can be transformed into a structure \( R \) for the transformed problem \( QR \) achieving

\[
\begin{align*}
Q_R(n) &= O(\log n)Q_S(n), \\
M_R(n) &= O(\log n)M_S(n), \\
P_R(n) &= O(\log n)P_S(n).
\end{align*}
\]

Bentley and Saxe (1979) introduced some other transformations for adding range restrictions to decomposable searching problems, obtaining different trade-offs between
query time and storage required. Their results are static, i.e., they do not allow for updates. Willard and Luecker (1985) presented some dynamic solutions.

In this paper we will develop two new classes of structures for solving the problem of adding range restrictions to decomposable searching problems. These two classes will have a variety of trade-offs with all previous known results contained in them. In section 2 we will develop a class of \textit{M-structures} that use little memory and have reasonable query time and preprocessing costs. The class of \textit{Q-structures} developed in section 3 has a much better query time at the cost of an increase in storage and preprocessing time. Both methods use a multiway tree in which all internal nodes contain some $F(n)$ sons and associate structures to internal nodes of this tree. By varying $F$ we get different structures with different trade-offs for query time, storage and preprocessing time. Examples of these trade-offs are shown in section 4. M-structures and Q-structures are static. To make them dynamic we introduce a new weight-balanced multiway tree (the MWB-tree) in section 5. In section 6 we use the MWB-tree to obtain dynamic structures for the addition of range restrictions. Finally, in section 7 we will use the structures in both classes to solve the $d$-dimensional range searching problem. This is done by applying addition of range restrictions recursively. The results obtained contain many previously known bounds, e.g., the structures developed by Bentley & Friedman (1979) and by Bentley & Maurer (1980). The known bounds were obtained by using many different types of structures. In our approach we only have to vary $F(n)$. Moreover, all our results are dynamic.

2. M-structures

In this section we will define the class of M-structures that have low storage cost at an expense of increased query time. Let $F(n)$ be an integer function with $1 < F(n) \leq n$. Let $S$ be a static data structure for a decomposable searching problem $Q$. An M-structure of order $F(n)$ is a B-tree of order $F(n)$ in which we store the points ordered by added parameter. With each internal node $\beta$ with sons $\beta_1, \ldots, \beta_k$ we store $S$-structures $S_{1}, \ldots, S_{k}$, where $S_{i}$ contains all points in the subtree rooted at $\beta_i$. (We structure internal nodes as balanced binary trees in order to locate sons in $O(\log F(n))$ time.)

Let $R$ be such an M-structure. If we want to perform a query $Q(x, [A : B], V)$ we search with both $A$ and $B$ in $R$. For some time $A$ and $B$ will follow the same path and nothing needs to be done. But at some internal node $\beta$ (possibly the root) $A$ and $B$ will turn to different sons $\beta_a$ and $\beta_b$, respectively. All points that are contained in the subtrees rooted between $\beta_a$ and $\beta_b$ have their added parameter in the range, and we have to perform the query on them. This can easily be done by performing the query on the $S$-structures $S_{a+1}^{\beta_a}, \ldots, S_{k-1}^{\beta_a}$. Next we have to handle the points that belong to the subtrees $T_{\beta_a}$ and $T_{\beta_b}$ rooted at $\beta_a$ and $\beta_b$. We will only describe the actions in $T_{\beta_a}$, $T_{\beta_b}$ can be handled in a similar way. We continue our search in subtree $T_{\beta_a}$. Suppose our search has reached a node $\alpha$, and the search has to be continued in son $\alpha_a$. All points that belong to subtrees at the right of $\alpha_a$ have their added parameter in the range and we have to perform the query on them. This can easily be done by performing the query on the $S$-structures $S_{a+1}^{\alpha_a}, \ldots, S_{r(n)}^{\alpha_a}$. We continue our search with $A$ in $T_{\alpha_a}$. When the search for $A$ and $B$ reaches leaves of $R$, we have performed the query on all points that have their added parameter in the range $[A : B]$. Because the problem is decomposable we can combine the answers to the different queries using the composition operator $\Box$ for the problem.

**Theorem 2.1.** Let $F$ be an integer function such that $2 < F(n) \leq n$. Let $S$ be a data
structure for a decomposable searching problem $Q$. Let $QR$ be the transformed problem that adds a range restriction to $Q$. Then there exists a data structure $R$ that solves $QR$, achieving

$$Q_A(n) = O\left(\frac{\log n}{\log F(n)}\right)Q_S(n),$$

$$M_A(n) = O\left(\frac{\log n}{\log F(n)}\right)M_S(n),$$

$$P_A(n) = O\left(\frac{\log n}{\log F(n)}\right)P_S(n) + O(n \log n).$$

**Proof.** When we want to perform a query on $R$ with $[A:B]$ as range, we search with $A$ and $B$ as discussed above. There are $O\left(\frac{\log n}{\log F(n)}\right)$ levels in the tree, in each node the search time is $O(\log F(n))$, hence the total time needed for searching with $A$ and $B$ in the tree is $O(\log n)$. Note that at each level of $R$ we have to perform the original query on at most $2F(n)$ $S$-structures, each containing at most $n$ points. The bound for the query time follows.

At each level, each point is stored in exactly one $S$-structure. Because $M_S$ is at least linear all $S$-structures on the same level together take $O(M_S(n))$ storage. The bound follows. The bound on the time required for building an $M$-structure follows in a similar way. □

When $Q_S(n) = \Omega(n^\epsilon)$ for any positive $\epsilon$ it can be shown that $Q_A(n) = O(F(n))Q_S(n)^{\frac{\epsilon}{\epsilon}}$. Similar, $M_A(n) = O(M_S(n))$ when $M_S(n) = \Omega(n^{1+\epsilon})$ for any positive $\epsilon$ and $P_A(n) = O(P_S(n))$ when $P_S(n) = \Omega(n^{1+\epsilon})$ for any positive $\epsilon$.

3. $Q$-Structures

When query time rather than storage is our main concern, the $M$-structures are of no use. Hence, in this section we define a second class of structures, $Q$-structures, that have much better query times at the cost of an increase in the amount of storage and the preprocessing time.

Let $F(n)$ be an integer function with $4 < F(n) \leq n$. Let $G(n) = \lceil\sqrt{F(n)}\rceil$. Let $S$ be a static data structure for a decomposable searching problem $Q$. A $Q$-structure of order $F(n)$ is a B-tree of order $G(n)$ in which a node $\beta$ with sons $\beta_1, \ldots, \beta_k$ contains $S$-structures $S_{i,j}^\beta$ for all $1 \leq i \leq j \leq k$, where $S_{i,j}^\beta$ contains the points in the subtrees below $\beta_i \ldots, \beta_j$.

Let $R$ be such a $Q$-structure. If we want to perform a query $Q(x, [A:B], V)$, we search with both $A$ and $B$ through $R$. For some time $A$ and $B$ will follow the same path and nothing needs to be done. Then, at some internal node $\beta$ (possibly the root) $A$ and $B$ will turn to different sons $\beta_a$ and $\beta_b$, respectively. The points that are contained in subtrees, that are rooted at sons strictly between $\beta_a$ and $\beta_b$, belong to our range, and the query has to be performed on them. This can easily be done by performing the query on structure $S_{a+1,b-1}^\beta$. We still have to handle the points in the subtrees rooted at $\beta_a$ and $\beta_b$. We will only discuss the way we handle $\beta_a$. $\beta_b$ can be handled in a similar way. Continue the search with $A$ through the subtree rooted at $\beta_a$. Suppose our search is at node $\alpha$, where $A$ turns to son $\alpha'$. Now all points that are stored in subtrees at the right of $\alpha$, are in our range, and we have to perform the query on them. This can easily be done by performing the query on structure $S_{r+1,G(n)}^\alpha$. We continue our search with $A$ in the subtree rooted at
When A and B reach leaves of R we have performed the query on all points that have their range parameter in range \([A : B]\).

**Theorem 3.1.** Let \(F\) be an integer function such that \(4 < F(n) \leq n\). Let \(S\) be a data structure for a decomposable searching problem \(Q\). There exists a data structure \(R\) for solving \(QR\) achieving

\[
Q_R(n) = O\left(\frac{\log n}{\log F(n)}\right)Q_S(n) + O(\log n),
\]

\[
M_R(n) = O\left(\frac{\log n}{\log F(n)}\right)M_S(n),
\]

\[
P_R(n) = O\left(\frac{\log n}{\log F(n)}\right)P_S(n).
\]

**Proof.** It is easy to see that the time needed for searching with \(A\) and \(B\) is \(O(\log n)\). Note that at each level of \(R\) we have to perform the original query on at most two \(S\)-structures, each containing at most \(n\) points. As there are \(O(\log F(n))\) levels in the tree, the total query time is \(O\left(\frac{\log n}{\log F(n)}\right)Q_S(n) + O(\log n) = O\left(\frac{\log n}{\log F(n)}\right)Q_S(n) + O(\log n)\).

At each level every point is contained in \(O(G^2(n)) = O(F(n))\) \(S\)-structures. Each \(S\)-structure has size at most \(n\). Hence, because \(M_S(n)\) is at least linear, the \(S\)-structures at one level take at most \(O(F(n)M_S(n))\) storage. The bound follows.

To build the structure \(R\) we first order the \(n\) points with respect to the added parameter, and construct the B-tree. Next we construct the associated structures. This takes \(O(\frac{\log n}{\log F(n)}F(n))\) \(P_S(n)\), using the same arguments as for determining the amount of storage required. □

For any positive \(c\) it can be shown that \(Q_R(n) = O(Q_S(n))\) when \(Q_S(n) = \Omega(n^c)\), \(M_R(n) = O(F(n))M_S(n)\) when \(M_S(n) = \Omega(n^{1+c})\) and \(P_R(n) = O(F(n))P_S(n)\) when \(P_S(n) = \Omega(n^{1+c})\).

### 4. Examples

In this section we will give some examples of trade-offs we can get by choosing different functions \(F(n)\). The result of Theorem 2.1 can be written as

\[
Q_R(n) = O(B(n))Q_S(n),
\]

\[
M_R(n) = O(A(n))M_S(n),
\]

\[
P_R(n) = O(A(n))P_S(n) + O(n \log n),
\]

and the result of Theorem 3.1 as

\[
Q_R(n) = O(A(n))Q_S(n) + O(\log n),
\]

\[
M_R(n) = O(B(n))M_S(n),
\]

\[
P_R(n) = O(B(n))P_S(n),
\]

with

\[
A(n) = \frac{\log n}{\log F(n)}.
\]
and

\[ B(n) = \frac{\log n}{\log F(n)} F(n). \]

In Table 1 we show some possible choices that can be used to get different trade-offs. It contains all known trade-offs (see Bentley, 1979; Bentley & Saxe, 1979; Willard & Lueker, 1985) and many more.

**Table 1.** Some possible choices used to obtain different trade-offs

<table>
<thead>
<tr>
<th>F(n)</th>
<th>A(n)</th>
<th>B(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c log n</td>
<td>\log n</td>
<td>\log n</td>
</tr>
<tr>
<td>n^k</td>
<td>1</td>
<td>\sqrt{\log n} 2/\log n</td>
</tr>
<tr>
<td>\sqrt{\log n}</td>
<td>\log n</td>
<td>\log 1 + \delta n</td>
</tr>
<tr>
<td>\log^k n</td>
<td>\log n</td>
<td>\log \log n</td>
</tr>
</tbody>
</table>

---

5. The MWB-tree

In this section we develop a dynamic multiway weight-balanced search tree, the MWB-tree, that will be used in section 6 to transform the static M- and Q-structures into dynamic structures with reasonable insertion and deletion times. Let \( n_\beta \) denote the number of keys stored in the subtree rooted at \( \beta \). Let \( 0 < \alpha < 1/2 \). Let \( k > 2 \).

**Definition 5.1.** An MWB[\( \alpha \)]-tree of order \( k \) is a multiway tree, storing the keys in the leaves, with the following properties:

- Each internal node, except the fathers of the leaves have between \( \lceil k/2 \rceil \) and \( k \) sons.
- The fathers of the leaves have been 2 and \( k \) sons.
- For each internal node \( \beta \), \( n_\beta \leq \lceil \frac{k}{2} \rceil (1 - \alpha) n_{\text{father}(\beta)} \).

The MWB[\( \alpha \)]-tree is somewhere in between a BB[\( \alpha \)]-tree and a B-tree of order \( k \). But note that we do not force all leaves to be at the same level. As \( k \) will be replaced by \( F(n) \) in section 6, we do not treat \( k \) as a constant. \( \alpha \) will be treated as a constant. Splitting values in internal nodes are stored in a binary tree to speed up searching. The following lemma is obvious from the definition.

**Lemma 5.1.** The depth of an MWB[\( \alpha \)]-tree of order \( k \) is bounded by \( O(\log n / \log k) \).

**Definition 5.2.** Let \( \beta \) be a node in an MWB[\( \alpha \)]-tree of order \( k \). We call \( \beta \) perfectly balanced if each subtree rooted at a son of \( \beta \) contains at most \( \lceil \frac{k}{2} \rceil n_\beta \) keys, and \( \beta \) has exactly \( \lceil k/2 \rceil \) sons. We call an MWB[\( \alpha \)]-tree of order \( k \) perfectly balanced if the fathers of the leaves each contain at most \( \lceil k/2 \rceil \) sons (keys), and all other internal nodes are perfectly balanced.

**Lemma 5.2.** Given an ordered set of \( n \) keys, a perfectly balanced MWB-tree can be built in \( O(n) \) time.
PROOF. We cannot use the standard technique of building a B-tree by starting at the leaves because we cannot decide how many sons fathers of leaves should get. Hence, we use a different technique. We first build, from the root to the leaves, a skeleton tree without filling in the keys and internal splitting values. This can easily be done in $O(n)$ time. Next, working from the leaves upwards, we add the keys and the splitting values. It can easily be shown that this can be done in $O(n)$ time. The details are left as an exercise to the reader.

To perform insertions and deletions efficiently we will use two techniques from Overmars (1983). Insertion will use partial rebuilding. Deletions will be handled in a global way using global rebuilding. We will not describe these techniques in great detail. See Overmars (1983) for more details.

We will first only consider insertions. Suppose we want to insert a key $K$. First we search for $K$ in the tree. This will lead to the node $\beta$ in the tree where $K$ has to be inserted as a son. We insert $K$ here at the right place, find an appropriate splitting value and insert this splitting value in the balanced binary tree stored in $\beta$. If $\beta$ now has more than $k$ sons we rebuild the subtree rooted at $\beta$ as a perfectly balanced MWB-tree.

Next we walk back from $K$ towards the root, and adjust $n_{\beta}$ at each node $\gamma$ on the path. We also locate the highest node $\gamma$ that is out of balance, i.e., whose subtree contains too many keys. There are two possible ways of restoring the balance. If the father of $\gamma$ has less than $k$ sons, we reconstruct the subtree rooted at $\gamma$ as two perfectly balanced MWB-trees, each containing at most $[n_{\gamma}/2]$ keys, and replace $\gamma$ by these two subtrees. This clearly brings the tree back into balance. If the father of $\gamma$ has $k$ sons we rebuild the whole subtree rooted at the father of $\gamma$ as a perfectly balanced MWB-tree. Although this sometimes takes a lot of time we will show that the amortized insertion time will be low.

**Lemma 5.3.** The amortized insertion time in an MWB-tree of $n$ keys is $O(\log n)$.

**Proof.** When inserting a key we have to walk down and up the tree, sometimes rebuild the subtree rooted at the father of the inserted leaf, and sometimes rebuild the subtree rooted at some internal node. Clearly, the cost for walking up and down the tree is bounded by $O(\log n)$. Amortizing the cost for rebuilding subtrees is done in a similar way as described in chapter 4 of Overmars (1983) and will not be described in detail here.

Rebuilding the subtree rooted at the father of a leaf takes time $O(k)$. When this node was created, as part of a rebuilt perfectly balanced subtree, it had $[k/2]$ sons. Now it has $k + 1$ sons. Hence, we can charge the $O(k)$ work to a least $[k/2]$ keys that have been inserted since.

Assume a node $\beta$ went out of balance. Then $n_{\beta} > [k/2 \cdot (1 - 1/2)] = [k/2 \cdot n_{\text{father}(\beta)}]$. At the moment the subtree rooted at $\beta$ was rebuilt for the last time it contained $[k/2 \cdot n_{\text{father}(\beta)}]$ keys. Hence, there have been at least $\Omega(n_{\beta})$ insertions in the subtree since (see Overmars, 1983 for details). When the subtree rooted at $\beta$ has to be rebuilt this takes $O(n_{\beta})$ time (see Lemma 5.2). Charging this cost to the insertions that have taken place makes for $O(1)$ per insertion. When the father of $\beta$ has to be rebuilt it must have $k$ sons. At the moment this father of $\beta$ was rebuilt the last time it had $[k/2]$ sons. Hence $[k/2]$ times a son must have been split and, hence, there must have been $\Omega(n_{\text{father}(\beta)})$ insertions below sons of father($\beta$). Charging the rebuilding cost to these insertions is $O(1)$ per insertion. It can easily be shown (see Overmars, 1983) that each insertion is charged at most $O(\log k)$ times $O(1)$ work. The amortized insertion time bound follows.\[\square\]
To delete a key $K$ we search for $K$ in the tree. If $K$ is not present we are done. Otherwise we remove $K$ as a leaf. We don’t do any rebalancing of the tree. In this way, clearly, the tree will (slowly) go out of balance. On the other hand, deletions of this sort are weak in the sense that they do not increase the query time (See Overmars, 1983 for a precise definition of weak updates.) In Overmars (1983) it was shown that a structure $S$, that allows for weak deletions in time $WD_S(n)$, can be transformed into a structure that supports real deletions in time $O(WD_S(n) + P_S(n)/n)$, without any essential loss in the query time and amount of storage required. In our case $WD_S(n) = O(\log n)$. Hence, we obtain the following result:

**Theorem 5.4.** Insertions and deletions in an MWB-tree can be performed in $O(\log n)$ time. The insertion time bound is an amortized bound.

### 6. A Dynamic Solution for Adding Range Restrictions

To use the MWB-tree for adding range restrictions to decomposable searching problems we must take care that $k$ behaves like $F(n)$ (or $G(n)$). To this end, we sometimes rebuild the whole structure with $k = F(n)$ and do not change $k$ as long as the size of the set is less than $2n$ and more than $n/2$. When the size of the set becomes $2n$ or $n/2$ we rebuild the whole structure with the new $k$. Hence, at least $n/2$ updates take place between rebuildings of the structure. We can charge the rebuilding cost to these $n/2$ updates. This will only add a small amount of extra cost to the amortized update time, but guarantees that $k$ behaves like $F(n)$. This technique is a kind of global rebuilding Overmars (1983). We will from now on call such a structure an MWB-tree of order $F(n)$.

A dynamic $M$-structure is an MWB-tree of order $F(n)$ in which all points are stored in the leaves, ordered with respect to their added parameter. $S$-structures are associated with internal nodes in exactly the same way as in section 2.

To insert (or delete) a point $(p, k_p)$ we search with $k_p$ in the MWB-tree. For every node $\beta$ on the search path, where the search for $k_p$ turns to son $\beta$, we insert (delete) $p$ in $S\beta$. Next we insert (delete) $k_p$ in the MWB-tree and walk back to the root to find the highest node $\beta$ that is out of balance. To rebalance we rebuild subtrees in the way described in section 5.

**Theorem 6.1.** Let $F$ be an integer function such that $2 < F(n) \leq n$. Let $S$ be a dynamic data structure for a decomposable searching problem $Q$. There exists a data structure $R$ that solves $QR$, achieving

\[
Q_R(n) = O\left(\frac{\log n}{\log F(n)} F(n)\right) Q_S(n),
\]

\[
M_R(n) = O\left(\frac{\log n}{\log F(n)} M_S(n)\right),
\]

\[
P_R(n) = O\left(\frac{\log n}{\log F(n)} P_S(n) + O(n \log n)\right),
\]

\[
I_R^a(n) = O\left(\frac{\log n}{\log F(n)} (I_S(n) + \frac{\log n}{\log F(n)} P_S(n)/n)\right),
\]

\[
D_R^a(n) = O\left(\frac{\log n}{\log F(n)} (D_S(n) + P_S(n)/n)\right).
\]
Adding Range Restrictions to Decomposable Searching Problems

**Proof.** The depth of the dynamic $M$-structure is bounded by $O\left(\frac{\log n}{\log F(n)}\right)$. Hence, the query time, building time and storage follow in the same way as in the proof of Theorem 2.1.

The average update time can be split into four parts: (i) The time required for searching the MWB-tree and performing the insertion or deletion in this tree. This takes $O(\log n)$ time. (ii) The amount of time required for inserting or deleting points from associated structures. This clearly is bounded by $O\left(\frac{\log n}{\log F(n)} P_S(n)/n\right)$. (iii) The amortized amount of time required for rebuilding subtrees needed for rebalancing. By the same arguments described in section 5, using partial rebuilding, this can be estimated as $O\left(\frac{\log n}{\log F(n)} P_R(n)/n\right) = O\left(\frac{\log n}{\log F(n)} P_S(n)/n\right)$ per insertion. For deletions this amount is much less. Using global rebuilding it adds an average of $O\left(\frac{P_R(n)}{n}\right) = O\left(\frac{\log n}{\log F(n)} P_S(n)/n\right)$ per deletion. (iv) The amortized cost for rebuilding the whole structure when $k$ must be updated. It can easily be shown that this adds $O\left(\frac{P_R(n)}{n}\right) = O\left(\frac{\log n}{\log F(n)} P_S(n)/n\right)$ per update. □

Dynamic Q-structures are defined in exactly the same way. Again we use a MWB-tree of order $G(n)$ and associate structures in the way described in section 3. (Remember $G(n) = \sqrt{F(n)}$.) To insert or delete a point $(p, k_p)$ we search with $k_p$ in the MWB-tree. For every internal node $\beta$ where we have to continue the search in the son $\beta_i$ we insert or delete $p$ in all structures $S_{i,t}$ for all $s \leq i$ and $i \leq t$. Next we insert or delete $k_p$ in the tree, find the highest node that is out of balance and rebuild a subtree in the way described in section 5. It can easily be shown that this leads to the following result.

**Theorem 6.2.** Let $F$ be an integer function such that $4 < F(n) \leq n$. Let $S$ be a dynamic data structure for a decomposable searching problem $Q$. There exists a data structure $R$ that solves $QR$ achieving

\[
Q_S(n) = O\left(\frac{\log n}{\log F(n)}\right)Q_S(n) + O(\log n),
\]

\[
M_S(n) = O\left(\frac{\log n}{\log F(n)} F(n)\right)M_S(n),
\]

\[
P_S(n) = O\left(\frac{\log n}{\log F(n)} F(n)\right)P_S(n) + O(n \log n),
\]

\[
I_S(n) = O\left(\frac{\log n}{\log F(n)} \left(\frac{F(n)I_S(n)}{\log F(n)} + \frac{\log n}{\log F(n)} F(n)P_S(n)/n\right)\right),
\]

\[
D_S(n) = O\left(\frac{\log n}{\log F(n)} \left(\frac{F(n)D_S(n)}{\log F(n)} + F(n)P_S(n)/n\right)\right).
\]

In many practical cases the update time can be improved by making use of the fact that, when a subtree has to be rebuilt, the old subtree is available. Let $B_S(n)$ be the time required to build an $S$-structure from an ordered set of points (some prechosen ordering). If $B_S(n)$ is smaller than $P_S(n)$ it is easy to see that one can replace the $P_S(n)$ in the amortized insertion and deletion time bounds in Theorems 6.1 and 6.2 by $B_S(n)$. 

**7. d-dimensional Range Searching**

In this section we will use the $M$- and $Q$-structures recursively to solve the $d$-dimensional range searching problem. The 1-dimensional range searching problem can be solved in $O(\log n)$ query time, $O(\log n)$ update time and $O(n \log n)$ preprocessing time, using $O(n)$
Table 2. The query time, update time and amount of storage required for $d$-dimensional range searching for different choices of $F(n)$

<table>
<thead>
<tr>
<th>Structure</th>
<th>$F(n)$</th>
<th>$Q_{d}(n)$</th>
<th>$U_{d}(n)$</th>
<th>$M_{d}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>$n^d$</td>
<td>$n^d$</td>
<td>$\log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$M$</td>
<td>$\log^d n$</td>
<td>$\log^d n$</td>
<td>$\log^d n$</td>
<td>$\log^d n$</td>
</tr>
<tr>
<td>$M$</td>
<td>$c$</td>
<td>$\log^d n$</td>
<td>$\log^d n$</td>
<td>$n\log^d n$</td>
</tr>
<tr>
<td>$O$</td>
<td>$c$</td>
<td>$\log^d n$</td>
<td>$\log^d n$</td>
<td>$n\log^d n$</td>
</tr>
<tr>
<td>$Q$</td>
<td>$n^d$</td>
<td>$\log n$</td>
<td>$n^d$</td>
<td>$n^{1+c}$</td>
</tr>
</tbody>
</table>

storage. Clearly, $B_{d}(n) = O(n)$. Next we apply addition of range restrictions $d - 1$ times. In Table 2 the query time, update time and amount of storage required for $d$-dimensional range searching are given for different choices of $F(n)$.

Table 2 contains many of the known results on range searching (see e.g. Lueker, 1979; Willard & Lueker, 1985; Bentley & Friedman, 1979; Bentley & Maurer, 1980). But using the method presented in this paper they all follow in the same way, only varying $F(n)$. Moreover, all our results are dynamic. Finally, our method yields many more trade-offs.

8. Conclusions and Open Problems

In this paper we have presented two classes of structures for the addition of range restrictions to decomposable searching problems. These classes give us a wide variety of structures with different trade-offs for query time, memory cost and preprocessing time. Both static and dynamic structures have been considered. As an application we applied the results to the $d$-dimensional range searching problem. In this way we did obtain a whole class of structures, containing most of the known results and adding many new results.

Some open problems do remain. The transformations described in this paper only yield good amortized update time bounds. It is quite easy to change the deletion time bound to a worst-case bound (by applying techniques from Overmars, 1983), but the insertion time remains amortized. It is an interesting question whether this amortized bound can be turned into a worst-case bound as well. It is not clear whether the bounds in this paper are optimal. Other, better transformations might exist. Hence, there is a need for lower bounds on the efficiency of the transformations. Some extra techniques, like e.g. presorting, might be useful in reducing the bounds obtained in this paper further for some special subclasses of decomposable searching problems.

References