Using persistent data structures for adding range restrictions to searching problems*

Hans-Peter Lenhof Michiel Smid

Max-Planck-Institut für Informatik

W-6600 Saarbrücken, Germany

Abstract

The problem of adding range restrictions to decomposable searching problems is considered. First, a general technique is given that makes an arbitrary dynamic data structure partially persistent. Then, a general technique is given that transforms a partially persistent data structure that solves a decomposable searching problem into a structure for the same problem with added range restrictions. Applying the general technique to specific searching problems gives efficient data structures, especially in case more than one range restriction, one of which has constant width, is added.

1 Introduction

In the theory of data structures, we want to design structures that store a given set of objects in such a way that specific questions (queries) about these objects can be answered efficiently. A well-known example is the *member searching problem*, where we are given a set V of objects. To answer a member query, we get an object q, and we are asked whether or not q is an element of V. Another example is the *(orthogonal) range searching problem*, where we are given a set V of points in d-dimensional space, and an axis-parallel

*This work was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

hyper-rectangle $q = ([a_1 : b_1], \ldots, [a_d : b_d])$, and we are asked to determine all points $p = (p_1, \ldots, p_d)$ in V, such that $a_1 \leq p_1 \leq b_1, \ldots, a_d \leq p_d \leq b_d$, i.e., all points of V that are in the hyper-rectangle q. As a third example, in the *nearest neighbor searching problem*, we are given a set V of points in the plane and a query point q, and we are asked to find a point in V that is nearest to q.

Many general techniques are known to design static and dynamic data structures that solve such searching problems. See e.g. Bentley [1, 2], Overmars [11].

There is a special class of searching problems that has received much attention, the so-called decomposable searching problems. Let $PR(q, V)$ denote the answer to a searching problem PR with query object q and object set V.

Definition 1 (Bentley [1]) *A searching problem PR is called* decomposable*, if*

$$PR(q, V) = \Box(PR(q, A), PR(q, B)),$$

for any partition $V = A \cup B$ and any query object q, where the function $\Box$ can be computed in constant time.

For example, the member searching problem is decomposable with $\Box = \vee$, the range searching problem is decomposable with $\Box = \cup$, and the nearest neighbor searching problem is decomposable with $\Box =$ "minimal distance".

In this paper, we consider the problem of transforming searching problems into other problems by *adding range restrictions*. This transformation was introduced by Bentley [1], and was subsequently investigated by Willard and Lueker [17] and Scholten and Overmars [14].

If we add a range restriction to a searching problem for a set V, then we give each object x in V an additional parameter k_x. We assume that these new parameters are real numbers. In the transformed searching problem, we only query objects in V that have their parameter in a given range. We define this more precisely:

Definition 2 *Let PR be a searching problem for a set V of objects. To add a* range restriction*, we associate with each object x in V a real number k_x. In the transformed searching problem TPR, a query consists of a query object q together with an interval $[a : b]$, and*

$$TPR(q, [a : b], V) := PR(q, \{x \in V : a \leq k_x \leq b\}).$$

To illustrate the notion of adding a range restriction, consider the nearest neighbor searching problem. Interpret each point in the set V as a city. For each city x, define k_x as the size of its population. Then in the transformed problem we may ask for the city having a population between, say, 100,000 and 500,000, that is nearest to q.

A trivial example is the d-dimensional range searching problem, which is obtained by adding a range restriction to the $(d-1)$-dimensional range searching problem.

Bentley [1] gives a general technique for solving searching problems that arise by adding a range restriction to decomposable searching problems. More precisely, suppose we have a (static) data structure DS for a decomposable searching problem PR having query time $Q(n)$, size $S(n)$, and that can be built in $P(n)$ time. Then there exists a data structure for the transformed problem TPR for which these three complexity measures increase by a factor of $O(\log n)$. Willard and Lueker [17] show that if the data structure DS is dynamic, then the transformed structure can also be made dynamic. Scholten and Overmars [14] give a general technique that gives trade-offs between the query time and the size of the transformed structure. They present both static and dynamic solutions.

Gabow, Bentley and Tarjan [8] considered problems that can be viewed as adding several range restrictions. An example of the problems they consider is the problem of range searching for maximum. (See also Sections 5 and 7.)

In this paper, we give new techniques for adding range restrictions. The main part of the paper considers the problem of adding range restrictions of constant width. That is, in the transformed query problem TPR, we only query with a range restriction on the associated parameter of the form $[a : a + c]$, where a is an arbitrary real number, but c is fixed for all queries. We give a general technique that transforms a data structure for a problem PR into another structure in which the transformed problem TPR can be solved efficiently.

Applying this technique gives e.g. an optimal solution to the planar fixed height range searching problem. (This result was known already.) We also generalize the technique such that several range restrictions, only one of which has constant width, can be added. This technique leads to interesting results, such as a data structure for the d-dimensional fixed "height" range searching problem.

In the transformations, we use partially persistent data structures, which

are dynamic structures in which we cannot only query the current version, but also old versions, that arise by inserting objects. The main idea in the transformation is to consider the values of the additional parameters k_x as moments in time. Note that this idea is not new. Sarnak and Tarjan [13] use it to obtain an optimal solution for the planar point location problem.

The rest of this paper is organized as follows. In Section 2, we recall some results on partially persistent data structures. Moreover, we use a Van Emde Boas Tree—implemented using dynamic perfect hashing—to make arbitrary data structures partially persistent, at the cost of a slight increase in complexity, and at the cost of introducing randomization. This technique was mentioned by Tarjan at the FOCS 88 conference during the presentation of Dietzfelbinger et al. [5]. Since—as far as we know—the technique has not been published yet, and since there are some non-trivial details in the proof, we present it here in full detail.

In Section 3, we prove that in order to add range restrictions of constant width, it suffices to have fast transformations that add half-infinite range restrictions of the form $(-\infty : b]$ and $[a : \infty)$. Section 4 gives the transformation for adding a range restriction of constant width. This transformation is illustrated in Section 5 with some examples. We obtain e.g. an optimal solution for the fixed height range searching problem.

Of course, the transformation can be repeated to add several range restrictions. In Section 6, however, we give another transformation that gives better results. This transformation is based on range trees and fractional cascading. We illustrate it in Section 7 with some examples. One of the results is a data structure for the d-dimensional fixed "height" range searching problem, having size $O(n(\log n)^{d-2})$, in which queries can be solved in $O((\log n)^{d-2} \log\log n + k)$ time in the worst case, where k is the size of the output.

In Section 8, we show how the results can be extended to arbitrary range restrictions. We finish the paper in Section 9 with some concluding remarks.

In the rest of this paper, we use the notations $Q(n)$, $I(n)$, $S(n)$ and $P(n)$ to denote the query time, insertion time, size and building time of a data structure, respectively. We assume that all these functions are non-decreasing. Furthermore, the functions $S(n)/n$ and $P(n)/n$ are assumed to be non-decreasing.

We often state that an operation takes "expected and amortized time $O(f(n))$". This means that a sequence of n such operations takes $O(n\, f(n))$

expected time, where the randomization is based on choices made by the algorithm and is independent of the sequence of operations.

2 Partially persistent data structures

In general, data structures are *ephemeral*, in the sense that queries and updates can only be performed in the current version of the structure. In this paper, we need data structures in which we can query the current and old versions, and such that we can perform updates in the current version. Data structures in which this is possible are called *partially persistent*. (This in contrast to fully persistent data structures in which we can also update old versions.) See Driscoll et al. [6], Sarnak and Tarjan [13] for these notions.

Driscoll et al. [6] showed that ephemeral data structures of bounded in-degree can be transformed into partially persistent data structures, having the same complexity. For binary search trees, their result is as follows.

Theorem 1 ([6]) *There exists a partially persistent binary search tree, in which we can search for the largest (resp. smallest) element that is at most (resp. at least) equal to a given element x, in an arbitrary version, in $O(\log n)$ worst-case time. Also, one-dimensional range queries can be performed in an arbitrary version, in $O(\log n+k)$ worst-case time, if k is the size of the output. An element can be inserted or deleted in the current version of the structure, in $O(\log n)$ worst-case time. The size of the data structure is bounded by $O(n)$. Here, n is the maximal number of elements that have been present so far.*

Unfortunately, the technique in [6] only applies to data structures of bounded in-degree. For example, it does not apply to Van Emde Boas Trees [15, 16] which have in-degree $\sqrt{n}$. In the rest of this section, we show how an arbitrary ephemeral data structure, in which the current version can be queried, and in which objects can be inserted and deleted, can be transformed into a partially persistent structure. As mentioned already, this technique was mentioned by Tarjan during the presentation of [5]. As far as we know, a complete description of the method and a full proof of the complexity bounds have not appeared yet.

In order to give this transformation, we need a randomized version of the Van Emde Boas Tree in which we insert elements only in increasing

order. This randomized version uses dynamic perfect hashing as given by Dietzfelbinger et al.[5] and is partially due to Mehlhorn and Näher [10]:

Theorem 2 *Let n be a prime number. A randomized version of the Van Emde Boas Tree, that stores m integers in the range $[1:n]$ has size $O(m)$. In this structure, we can search for the largest element that is at most equal to a given integer x, in $O(\log\log n)$ worst-case time. In this structure, we can insert a new maximal element in expected and amortized $O(1)$ time. We can find the current maximal element in $O(1)$ worst-case time. The structure for the empty set can be initialized in $O(1)$ worst-case time.*

Proof. Mehlhorn and Näher [10] first give a so-called stratified tree. They use the dynamic perfect hashing strategy of [5] to implement this tree. If the set of elements has size m and if these elements are integers in $[1:n]$, this stratified tree has size $O(m\log\log n)$. Searches for arbitrary integers in $[1:n]$ take $O(\log\log n)$ worst-case time. The maximal element stored in the tree can be found in $O(1)$ worst-case time. An arbitrary integer in $[1:n]$ can be inserted in expected and amortized $O(\log\log n)$ time. Finally, this structure can be initialized for the empty set in $O(1)$ worst-case time.

As in [10, 15], we apply the method of pruning to this stratified tree. That is, we divide the current set of m elements into buckets of size $\lfloor\log\log n\rfloor$, such that the elements of the first bucket are less than those of the second one, which in turn are less than those in the third bucket, etc. We store the minimal element of each bucket in a stratified tree. Each bucket is stored as an (unordered) linked list.

To search for an element, we first search in the stratified tree to locate the appropriate bucket. Then we do a linear search in this bucket. Clearly, the searching time is bounded by $O(\log\log n)$ in the worst case.

To insert a new maximal element, we first use the stratified tree to find the "maximal" bucket, i.e., the bucket storing the largest elements. Then, we add the new element at the end of the list belonging to this bucket. As soon as the maximal bucket has size $2\lfloor\log\log n\rfloor$, we split it in two buckets of equal size, such that the elements in the first bucket are smaller than those in the second bucket. We insert the minimal element of the bucket containing the largest elements into the stratified tree.

If no splitting is necessary, an insertion takes $O(1)$ worst-case time. Otherwise, we need $O(\log\log n)$ time to split the bucket (using a linear time

median algorithm), and expected and amortized $O(\log\log n)$ time to insert the minimal element of the new bucket into the stratified tree. Since this splitting occurs only once every $\lfloor \log\log n \rfloor$ insertions, it follows that the expected and amortized time to insert a new maximal element is bounded by $O(1)$.

It will be clear that the current maximum can be maintained such that it can be found in $O(1)$ worst-case time.

Since the stratified tree stores only $m/\log\log n$ elements, it has size $O(m)$. Also, the lists of the buckets together have size $O(m)$. Hence, the entire data structure has size $O(m)$. This finishes the proof. □

We take the Random Access Machine (RAM) as our model of computation. The memory of a RAM consists of an array, the entries of which have unique addresses. Such a memory location can be accessed in constant time if this address is known.

We assume that the total number of updates that has to be carried out in the persistent data structure is known in advance. Call this number n. If n is not a prime, then we apply Theorem 2 with p the smallest prime that is at least equal to n. It is known that $n \leq p \leq 2n$. Therefore, all complexity bounds of the form $O(f(p))$ are asymptotically equivalent to $O(f(n))$. We assume that this prime p is given with n as part of the input. For practical applications, n has at most, say, 10 decimal digits. Therefore, for these values of n, the corresponding prime p can be found in tables.

Remark: It is not a restriction to assume that the number of updates is known in advance. (Note that in our application, this number is known!) If this number is not known, then we can use the standard doubling method: Assume we start with an empty data structure. Then we start with $n = 2$. In general, after n updates, we double n, discard the data structure and start anew by performing the first n updates again. If $U(m)$ is the update time of the data structure, then the amortized update time becomes $1/n \sum_{i=1}^{\log n} 2^i U(n) = O(U(n))$.

Because of this remark, we assume from now on, that if we apply Theorem 2, n is a prime number.

Suppose we have an ephemeral data structure DS for some searching problem PR. Let $Q(m)$, $U(m)$ and $S(m)$ denote the query time, the time to

perform an update and the size of DS, respectively. Let $C(m)$ be the number of memory locations that are changed during an update. (Both $U(m)$ and $C(m)$ may be amortized.) We transform DS into a partially persistent data structure.

The partially persistent data structure: Before we give the structure, note that in the ephemeral case the (current version of the) data structure DS would have been stored in certain memory locations of the RAM.

The persistent structure looks as follows. In each memory location i that ever would have been used by the ephemeral structure, we store a pointer to a randomized Van Emde Boas Tree T_i of Theorem 2, in which we maintain the history of this memory location. In such a tree T_i, we store all "time stamps" at which the content of memory location i was changed. These time stamps are integers in the range $[1 : n]$, and represent the numbers of the updates. With each time stamp t we store the (changed) content of cell i at time t.

The update algorithm: We assume that we start with an empty structure, and we number the updates $1, 2, \ldots$. Consider the t-th update. So suppose that we perform the t-th update in the current version of the persistent data structure. Then we simulate the update algorithm of the ephemeral structure: If we want to read the content of memory cell i, then we follow the pointer to the Van Emde Boas Tree T_i, and we search for the maximal element that is stored in this tree. With this maximum, we find the current content of memory cell i. Given this content, we take the appropriate action.

If during this simulation, we want to change the content of memory location i, then we insert the time stamp t in the corresponding Van Emde Boas Tree T_i. With this time stamp, we store the new content of memory location i. If in the simulated structure we would have to rewrite a memory location i more than once, then we find out that the time stamp t is already present in T_i. In that case, we rewrite the content of location i that is stored with time stamp t.

If the simulated structure would have needed a new memory location i, then we initialize an empty Van Emde Boas Tree T_i, and we insert the time stamp t together with the content of location i in T_i. In location i itself, we store a pointer to T_i. (If a memory location is not needed any more, we keep in it the pointer to the corresponding Van Emde Boas Tree.)

The query algorithm: Suppose we want to perform a query in the version of DS as it was at "time" t, i.e., after exactly t updates have been carried out. Then we simulate the query algorithm of the ephemeral data structure: If we want to read the content of memory cell i, we follow the pointer to the corresponding Van Emde Boas Tree T_i, and we search in T_i for the largest time stamp t' that is at most equal to t. With this time stamp, we find the content of memory cell i at time t. Given this content, we take the appropriate action.

Theorem 3 *Suppose we start with an empty data structure DS and consider a sequence of n updates. In the partially persistent version of DS, we can query arbitrary versions in worst-case time $O(Q(n) \log\log n)$, and we can update the current version in expected and amortized time $O(U(n))$. The expected size of the persistent structure is bounded by $O(S(n) + n\,C(n))$.*

Proof. Since we perform n updates, the maximal number of objects that are represented by the data structure is at most n.

To prove the bound on the query time, observe that if we want to know the content of memory cell i at time t, we search in the Van Emde Boas Tree T_i for t. By Theorem 2, this takes $O(\log\log n)$ time. Therefore, since we need the content of at most $Q(n)$ memory locations, the query time increases by a factor of $O(\log\log n)$.

To perform an update, we have to insert time stamps in several Van Emde Boas Trees. Note that these time stamps are new maximal elements, because we only allow updates to be performed in the current version of the data structure.

The time to perform an update in the current version of the data structure is proportional to the update time of the ephemeral structure plus the number of memory locations in which the content is changed multiplied by the time needed to update the corresponding Van Emde Boas Trees.

Note that in the insertion algorithm of the structure of Theorem 2 we perform $\lfloor \log\log n \rfloor - 1$ insertions at constant cost, and then one insertion at an expected and amortized cost of $O(\log\log n)$.

Now consider a sequence of n updates, starting with an empty structure. It takes $\sum_{j=1}^{n} U(j)$ time to simulate the update algorithm of the ephemeral structure. If during the sequence of updates there are n_i changes in memory location i, then the total expected time needed to update the Van Emde

Boas Tree T_i is bounded by

$$O(n_i) + \left\lfloor \frac{n_i}{\lfloor \log\log n \rfloor} \right\rfloor \times O(\log\log n) = O(n_i).$$

It follows that the total expected time for this sequence of n updates is bounded by

$$O\left(\sum_{j=1}^{n} U(j)\right) + O\left(\sum_{i=1}^{S(n)} n_i\right),$$

where the variable i in the second summation ranges over all memory locations that would ever have been used by the ephemeral structure. The first summation is bounded by $O(n\,U(n))$. To bound the second summation, note that it is equal to the total number of memory changes that occur during the n updates. Since during n updates we can change at most $n\,U(n)$ memory locations, the second summation is also bounded by $O(n\,U(n))$. This proves that the amortized expected update time of the partially persistent structure is bounded by $O(U(n))$.

Since there at most n elements present, the ephemeral structure would use at most $S(n)$ memory cells. Hence, there are at most $S(n)$ pointers to Van Emde Boas Trees. So we are left with the total size of the randomized Van Emde Boas Trees. If we change the content of a memory location during an update, we add an expected and amortized amount of $O(1)$ information to the corresponding Van Emde Boas Tree, because we spend expected and amortized $O(1)$ time. (Note that such a tree has a size that is linear in the number of stored elements, *not* in the size of the universe!) Therefore, in a sequence of n updates, the expected amount of information we add to all these trees is bounded by $O(n\,C(n))$. Hence, the expected size of these trees together is bounded by $O(n\,C(n))$. This completes the proof. □

We can apply Theorem 3 with DS a randomized Van Emde Boas Tree. If this data structure stores a set of m integers in the range $[1 : n]$, then we can search for the predecessor of an element in $O(\log\log n)$ time, we can insert or delete an element in $O(\log\log n)$ amortized and expected time, an update causes an expected and amortized number of $O(\log\log n)$ memory modifications, and the structure uses $O(m)$ space.

Hence, Theorem 3 applied to this data structure gives a partially persistent version that uses $O(m \log\log n)$ space, such that we can query any

version in $O((\log\log n)^2)$ time. For this special data structure, however, Dietz and Raman [4] prove that if only insertions are performed, a better solution is possible:

Theorem 4 ([4]) *There exists a partially persistent version of the Van Emde Boas Tree having size $O(m)$ if it stores m integers in the range $[1:n]$. In this structure, we can insert an element in the current version in $O(\log\log n)$ amortized and expected time. Moreover, we can solve a one-dimensional range query in an arbitrary version in $O(\log\log n + k)$ time, if k is the size of the output.*

Proof. In [4], the bounds on the space and update time are proved. It is also proved there, that we can search in an arbitrary version for the smallest resp. largest element that is at least resp. at most equal to a query element, in $O(\log\log n)$ time. In order to support range queries, we maintain a partially persistent list keeping the current elements in sorted order. This list is obtained by applying the method of [6]. Each element in the Van Emde Boas Tree contains a pointer to its occurrence in this list. Then a range query in the t-th version with query interval $[a:b]$ is solved by searching in the t-th version of the Van Emde Boas Tree for the smallest element, say x, that is at least equal to a. Then, we follow the pointer from x to the list, walk along the t-th version of this list and report all elements that are at most equal to b. In this way, we solve a range query in $O(\log\log n + k)$ time. □

3 It suffices to consider half-infinite range restrictions

As we said already, we restrict ourselves to decomposable searching problems, and we only add range restrictions of constant width. In this section, we show that it is sufficient to consider the case where we add half-infinite range restrictions, i.e., range restrictions of the form $(-\infty:b]$ and $[a:\infty)$.

Let PR be a decomposable searching problem for a set V of objects. Let each object x in V have an additional parameter k_x. Let TPR be the searching problem that is obtained by adding a range restriction to PR.

Suppose we have a data structure DS that stores the set V, such that queries $TPR(q,[a:\infty),V)$ can be solved in $Q(n)$ time. Let $S(n)$ and $P(n)$ be the size and the building time of the structure DS, respectively.

It is clear that by replacing the values k_x by $-k_x$, we can store V in a structure DS' of the same type as DS, such that queries $TPR(q, (-\infty : b], V)$ can be solved in time $Q(n)$. Also, the size resp. building time of DS' is $S(n)$ resp. $P(n)$.

We give a data structure that solves queries $TPR(q, [a : a + c], V)$, where c—the width of the range restriction—is a fixed real number.

Partition the reals into intervals $I_i := [ic, (i + 1)c)$ of length c, where i ranges over the integers. Next, partition the set V into subsets $V_i := \{x \in V : k_x \in I_i\}$.

The data structure: For each index i, for which V_i is not empty, there are two data structures DS_i and DS'_i, both storing V_i. In the structure DS_i, we can answer queries with a range restriction $[a : \infty)$. In the other structure DS'_i, we can answer queries with range restriction $(-\infty : b]$.

There is a balanced binary search tree T that contains all indices i for which V_i is not empty. Each such index i contains pointers to the structures DS_i and DS'_i.

The query algorithm: To answer a query $TPR(q, [a : a+c], V)$, do the following. Let $i := \lfloor a/c \rfloor$. Then search in the tree T for i. If i is not present, then there are no objects in V having their additional parameter in the interval $[ic, (i+1)c)$. Otherwise, if i is present, do a query $TPR(q, [a : \infty), V)$ in the structure DS_i. Next, search for $i+1$ in the tree T. If it is not present, we are finished. Otherwise, do a query $TPR(q, (-\infty : a + c], V)$ in the structure DS'_{i+1}.

Finally, use the merge operator $\square$ belonging to PR to combine the answers obtained by these two queries, to obtain the final answer to the query.

Theorem 5 *In the above data structure, queries with range restriction of constant width c can be answered in $O(Q(n) + \log n)$ time. The size resp. building time of the data structure is bounded by $O(S(n))$ resp. $O(n \log n + P(n))$.*

Proof. It is clear that the range restriction $[a : a + c]$ overlaps only two intervals I_i and I_{i+1}, where $i = \lfloor a/c \rfloor$. Therefore, we only have to consider objects in V_i and V_{i+1}. Since $I_i \cap [a : a + c] = [a : (i + 1)c)$, and since all objects in V_i have additional parameters that are less than $(i + 1)c < a + c$,

it suffices to query all objects in V_i with range restriction $[a : \infty)$. Similarly, it suffices to query the objects in V_{i+1} with range restriction $(-\infty : a + c]$. This proves that the query algorithm is correct.

The time for a query is bounded by $O(\log n + Q(|V_i|) + Q(|V_{i+1}|))$. Since both V_i and V_{i+1} have size at most n, and since we assumed that the query time is non-decreasing, the bound on the query time follows.

The size of the data structure is bounded by

$$O\left(n + \sum_{i:V_i \neq \emptyset} S(|V_i|)\right).$$

We assumed that $S(n)/n$ is non-decreasing. Therefore,

$$S(|V_i|) = |V_i| \frac{S(|V_i|)}{|V_i|} \leq |V_i| \frac{S(n)}{n}.$$

It follows that the size of the data structure is bounded by

$$O\left(n + \sum_{i:V_i \neq \emptyset} |V_i| \left(S(n)/n\right)\right) = O(n + S(n)) = O(S(n)).$$

The bound on the building time follows in a similar way. Here, the $O(n \log n)$ term is the building time of the tree T. □

4 Adding a range restriction of constant width

We now use the results of the preceding sections to transform a data structure that solves a decomposable searching problem into a structure that solves the same problem with an added range restriction of constant width. By Theorem 5, it suffices to consider the case where the range restriction is half-infinite.

Let PR be a decomposable searching problem for a set V of n objects. Let DS be a partially persistent data structure that stores V. Let $Q(n)$ be the time needed to query an arbitrary version, let $I(n)$ be the time needed to insert an object in the current version, and let $S(n)$ be the size of the data structure.

Each element x of V gets an additional parameter k_x. Let TPR be the searching problem obtained by adding a range restriction to PR.

We give a data structure TDS that solves the problem TPR for range restrictions of the form $(-\infty : b]$.

The data structure: Order the objects in V according to their additional parameters k_x. (Equal parameters are ordered in arbitrary order.) Let $k_{x_1} \leq k_{x_2} \leq \ldots \leq k_{x_n}$ be this order. Store the parameters k_x for $x \in V$, in this order, in an array $V(1:n)$.

Then, insert the objects, one after another, into the initially empty partially persistent data structure DS, in the order $x_1, \ldots, x_n$.

The query algorithm: To answer a query $TPR(q, (-\infty : b], V)$, do the following. Search for b in the array V. Let i be the position of b, i.e., $V(i) \leq b < V(i+1)$.

We have to solve the searching problem PR for all objects in V that have an additional parameter that is at most equal to b. But these are exactly the objects that were present in DS after i insertions, i.e., at "time" i. Therefore, query the persistent structure DS at time i.

Theorem 6 *The data structure TDS allows queries with range restrictions of the form $(-\infty : b]$ to be solved in $O(\log n + Q(n))$ time, has size $O(S(n))$, and can be built in $O(n \log n + n\, I(n))$ time.*

Proof. It is clear that the query algorithm is correct. The time to locate b in the array V is bounded by $O(\log n)$. Once we have located b, we know its position i and we can query the structure DS at time i, in $Q(n)$ time. The size of the data structure is bounded by $O(n)$ for the array V, plus $S(n)$ for the structure DS. Because $S(n)$ is at least linear, the size of the entire data structure is bounded by $O(S(n))$. To build the structure, we order the objects in V in order of increasing additional parameter, and we build the array V. This takes $O(n \log n)$ time. Finally, we insert the objects in increasing order in the structure DS. This takes $\sum_{i=1}^{n} I(i)$ time. Since we assumed the function $I(\cdot)$ to be non-decreasing, this sum is bounded by $n\, I(n)$. This completes the proof. □

Applying Theorem 5 leads to the main result of this section.

Theorem 7 *For the searching problem TPR, there exists a data structure that allows queries with range restrictions of constant width to be solved in $O(\log n + Q(n))$ time, that has size $O(S(n))$, and that can be built in $O(n \log n + n\, I(n))$ time.*

5 Examples

In this section, we give some applications of Theorem 7. As a first application, consider the *fixed height range searching problem.* Here we are given a set V of points in the plane. A query consists of an axis-parallel rectangle $[a_1 : b_1] \times [a_2 : a_2 + c]$ of fixed height c, and we have to find all points $p = (p_1, p_2)$ in V that are in this rectangle, i.e., satisfy $a_1 \leq p_1 \leq b_1$ and $a_2 \leq p_2 \leq a_2 + c$. Clearly, this is an example of a one-dimensional range searching problem with the addition of a constant width range restriction.

So, in the above terminology, let PR be the one-dimensional range searching problem. Let DS be the partially persistent search tree of Theorem 1. This structure allows one-dimensional range queries in arbitrary versions to be solved in $Q(n) = O(\log n + k)$ time in the worst case, elements can be inserted in the current version in $I(n) = O(\log n)$ time in the worst case, and it has size $S(n) = O(n)$.

Use this partially persistent structure DS for the first coordinates of the points in V. Each first coordinate x has as additional parameter k_x the corresponding second coordinate. In this way, the problem TPR—with a constant width range restriction—is the fixed height range searching problem.

Applying Theorem 7 to DS gives the following result:

Theorem 8 *For the fixed height range searching problem, there exists a data structure having a worst-case query time of $O(\log n + k)$, where k is the number of answers to the query, that has size $O(n)$, and that can be built in $O(n \log n)$ time.*

This gives an optimal solution for the fixed height range searching problem. Note that Klein et al. [9] even give a fully dynamic solution to this problem.

The above solution can be generalized to higher dimensions. Let V be a set of points in d-dimensional space. A query consists of a hyper-rectangle

$[a_1 : b_1] \times \ldots \times [a_{d-1} : b_{d-1}] \times [a_d : a_d + c]$ of fixed "height" c, and we have to find all points of V that are in the hyper-rectangle. Take the data structure of Theorem 8 for the last two dimensions, and add the first $d-2$ dimensions using the technique of Bentley [1]. This leads to the following result:

Theorem 9 *Let $d \geq 2$. For the d-dimensional range searching problem, where the query rectangles have constant width in one fixed dimension, there exists a data structure having a worst-case query time of $O((\log n)^{d-1} + k)$, where k is the number of answers to the query, that has size $O(n(\log n)^{d-2})$, and that can be built in $O(n(\log n)^{d-1})$ time.*

Next, we consider the problem of *range searching for minimum.* Let V be a set of n points in the plane. A query consists of a "rectangle" $[a_1 : \infty) \times [a_2 : a_2 + c]$ for some fixed c. We have to find a point in V that is lying in this rectangle, with minimal first coordinate. That is, among all points $p = (p_1, p_2)$ in V, such that $a_1 \leq p_1$, and $a_2 \leq p_2 \leq a_2 + c$, find one for which p_1 is minimal. (This is a two-dimensional generalization of a problem that was considered by Gabow, Bentley and Tarjan [8]. They consider the problem of searching for the point in the region $(-\infty : \infty) \times [a_2 : b_2] \times \ldots \times [a_d : b_d]$ having minimal first coordinate. In Section 7, we consider the generalization of this higher dimensional problem.)

In this case, let PR be the following searching problem. We are given a set of real numbers. A query consists of a real number a, and we have to find the smallest p in the set that is at least equal to a. Note that PR is decomposable. Again, we take for DS the partially persistent structure of Theorem 1. This structure solves problem PR with complexity $Q(n) = O(\log n)$ in the worst case, $I(n) = O(\log n)$ in the worst case, and $S(n) = O(n)$.

We use this structure DS for the first coordinates of the points in V. Each first coordinate x of these points has as additional parameter k_x the point's second coordinate. In this way, the problem TPR is the problem of range searching for minimum.

Applying Theorem 7 leads to the following result:

Theorem 10 *For the problem of range searching for minimum, where the query regions have constant height, there exists a data structure having a worst-case query time of $O(\log n)$, that has size $O(n)$, and that can be built in $O(n \log n)$ time.*

We can generalize Theorem 10 to higher dimensions in the same way as we did in Theorem 9. Then, for each dimension we add, we get an additional factor of $O(\log n)$ in the complexity bounds.

6 Adding several range restrictions

We now generalize the results to the case where more than one range restriction is added. We assume that one of the restrictions has constant width, say the last one.

We first consider the case, where we add two range restrictions. Let PR be a decomposable searching problem for a set V. Let DS be a partially persistent data structure that stores V. Let $Q(n)$ be the time needed to query an arbitrary version, let $I(n)$ be the time needed to insert an object in the current version, and let $S(n)$ be the size of the data structure.

Each element x of V gets two additional parameters k_x and l_x, taken from the real numbers. We consider the set V as a planar point set, where the k-values resp. l-values are the first resp. second coordinates. Let TPR be the transformed searching problem that is obtained by adding two range restrictions, i.e.,

$$TPR(q, [a_1 : b_1] \times [a_2 : b_2], V) := PR(q, \{x \in V : a_1 \leq k_x \leq b_1, a_2 \leq l_x \leq b_2\}).$$

We give a data structure that solves the problem TPR for range restrictions in which the second one has the form $[a_2 : a_2 + c]$ for a fixed c. It can be shown in exactly the same way as in Section 3 that it suffices to consider range restrictions in which the last one is half-infinite, say of the form $(-\infty : b_2]$.

The data structure: Store the points of V in a range tree. See [1, 17]. We briefly repeat the notion of this data structure. The points of V are stored in the leaves of a balanced binary search tree—called the *main tree*—ordered according to their first coordinates k_x. Each node w in this main tree contains an *associated structure*, defined as follows. Let V_w be the points of V that are in the subtree rooted at w. Then node w contains a pointer to an array V_w that stores the points of V_w ordered according to their second coordinates l_x.

We apply the technique of fractional cascading (see Chazelle and Guibas [3]) to this structure: Each array entry in an associated structure contains—

besides a point of V—two pointers. Let x be the point of V that is stored in such an array location, and let w be the node of the main tree whose associated structure we are considering. Then one pointer points to the point having largest second coordinate that is at most equal to l_x and that is stored in the associated structure of the left son of w. The other pointer has the same meaning for the right son of w.

Finally, for each node w of the main tree, there is a partially persistent data structure DS_w, that is obtained as follows. The objects of V_w are inserted in the initially empty structure DS_w, one after another, in order of non-decreasing second coordinate, i.e., in order of non-decreasing l-parameter.

The query algorithm: To answer a query $TPR(q, [a_1 : b_1] \times (-\infty : b_2], V)$, do the following. Find $O(\log n)$ nodes $w_1, w_2, \ldots$ in the main tree, such that the sets V_{w_i} partition all points in V that have their first coordinates in the interval $[a_1 : b_1]$. Then search in the associated structure of the root of the main tree for b_2. By following pointers between associated structures, find in the associated structure of each w_i the point with largest second coordinate that is at most equal to b_2.

This gives $O(\log n)$ positions in the associated structures of the w_i's. For each such position we have to solve the searching problem PR for all points that have a second coordinate that is at most equal to b_2. If the position in such an associated structure—of, say node w_i of the main tree—is t_i, then we query the partially persistent data structure DS_{w_i} at "time" t_i.

Finally, use the merge operator $\square$ belonging to PR to combine the answers obtained by these $O(\log n)$ queries, to get the final answer to the query.

Theorem 11 *For the searching problem TPR with two range restrictions, the second one of which has constant width, there exists a data structure that allows queries to be solved in $O(Q(n) \log n)$ time, that has size $O(S(n) \log n)$, and that can be built in $O(n\, I(n) \log n)$ time.*

Proof. We saw already that it suffices to prove this theorem for range restrictions of the form $[a_1 : b_1] \times (-\infty : b_2]$.

It is clear that the above query algorithm is correct. The time to locate the $O(\log n)$ positions in the associated structures is bounded by $O(\log n)$. Once these locations have been located, we can query the persistent structures DS_{w_i} in $Q(n)$ time per query. This proves that the query time is bounded by $O(Q(n) \log n)$.

The size of the data structure is bounded by $O(n \log n)$ for the range tree, plus the total sizes of all persistent data structures. Consider a fixed level in the main tree, and let $u_1, \ldots, u_m$ be the nodes on this level. The total size of the persistent structures at this level is equal to $\sum_{i=1}^{m} S(|V_{u_i}|)$. Since the function $S(n)/n$ is assumed to be non-decreasing, this sum is bounded above by

$$\sum_{i=1}^{m} |V_{u_i}| \, (S(n)/n).$$

But the sets V_{u_i} partition the set V. Therefore, the above sum is equal to $S(n)$. Hence, each level of the main tree contributes an amount of $S(n)$ to the size of the complete data structure. This proves that the total amount of space is bounded by $O(n \log n + S(n) \log n) = O(S(n) \log n)$, because $S(n) = \Omega(n)$.

To build the structure, we first build the range tree in $O(n \log n)$ time. (Use *presorting*, see Bentley [2].) Then for each node w of the main tree, we insert the objects of V_w in order of non-decreasing l-value in the structure DS_w. (Note that the objects in V_w are ordered already according to these l-parameters.) For each level of the main tree, there are exactly n insertions, each taking at most $I(n)$ time. Therefore, the time needed to build all persistent structures is bounded by $O(n \, I(n) \log n)$. This proves the theorem. □

We now generalize Theorem 11. Let PR be a decomposable searching problem for a set V. Let DS be a partially persistent data structure that stores V. Let $Q(n)$, $I(n)$ and $S(n)$ be the time needed to query an arbitrary version, the time needed to insert an object in the current version, and the size of DS, respectively. Each element x in V gets d parameters $k_1^x, \ldots, k_d^x$, taken from the real numbers. We consider V as a point set in d-dimensional space.

Let TPR be the transformed searching problem, i.e.,

$$TPR(q, \prod_{i=1}^{d} [a_i : b_i], V) := PR(q, \{x \in V : a_i \leq k_i^x \leq b_i, i = 1, \ldots, d\}).$$

We give a data structure that solves TPR for range restrictions in which the last one has the form $[a_d : a_d + c]$ for a fixed c. As before, it suffices to consider the case where the last restriction is half-infinite, say of the form $(-\infty : b_d]$.

If $d = 2$, we take the structure given above. Otherwise, if $d > 2$, we store the set V in the above structure, taking only the parameters k^x_{d-1} and k^x_d into account. Then we use the technique of Bentley [1] to add the first $d-2$ additional parameters. For each added parameter, the complexity bounds increase by a factor of $O(\log n)$.

The result is given in the following theorem, the proof of which is left to the reader.

Theorem 12 *Let $d \geq 2$. For the searching problem TPR with d range restrictions, the last one of which has constant width, there exists a data structure that allows queries to be solved in $O(Q(n)(\log n)^{d-1})$ time, that has size $O(S(n)(\log n)^{d-1})$, and that can be built in $O(n\, I(n)(\log n)^{d-1})$ time.*

7 Examples in higher dimensional space

In this section, we consider the d-dimensional versions of the problems we considered in Section 5.

First we consider the *fixed "height" range searching problem.* Let V be a set of n points in d-dimensional space. To answer a query, we get a hyper-rectangle $[a_1 : b_1] \times \ldots \times [a_{d-2} : b_{d-2}] \times [a_{d-1} : a_{d-1} + c] \times [a_d : b_d]$, where c is a fixed real number. We have to find all points of V that are in this rectangle.

We "normalize" the set V as follows. Store the points of V in the leaves of a balanced binary search tree, ordered according to their d-th coordinates. With each point, we store its rank in this order. That is, for each d-th coordinate p_d of any point p in V, we store the number of points that have a d-th coordinate that is at most equal to p_d.

To answer a query, we search in this search tree for the values a_d and b_d of the last range restriction. This gives us the ranks $r(a_d)$ and $r(b_d)$ of these numbers in the set of d-th coordinates of points in V. It is clear that if we now query the normalized set, i.e., the set where the d-th coordinates are replaced by their ranks, with the rectangle $[a_1 : b_1] \times \ldots \times [a_{d-2} : b_{d-2}] \times [a_{d-1} : a_{d-1} + c] \times [r(a_d) : r(b_d)]$, we get the right answer to the query.

So, we may assume that the d-th coordinates of the points in V have integer values in the range $[1 : n]$. We give a data structure that solves this normalized problem.

In Theorem 4, we saw a partially persistent version of the Van Emde Boas Tree. This structure stores a set of m integers in the range $[1 : n]$ in $O(m)$ space. A one-dimensional range query in an arbitrary version can be solved in $O(\log\log n + k)$ time, k being the size of the output. Moreover, an element can be inserted in the current version in $O(\log\log n)$ amortized and expected time.

We apply Theorem 12, as follows. We take for *PR* the one-dimensional range searching problem for a set of integers in the range $[1 : n]$. Let *DS* be the partially persistent Van Emde Boas Tree. Use this structure to store the normalized d-th coordinates of the points in V. Then we add $d - 1$ range restrictions. The first $d - 2$ are for the first $d - 2$ coordinates, the $(d - 1)$-th restriction—this one has constant width—is for the $(d - 1)$-th coordinates.

The transformed problem *TPR* is exactly the normalized fixed height range searching problem. Applying Theorem 12 gives:

Theorem 13 *Let $d \geq 2$. For the d-dimensional range searching problem, where the query rectangles have constant width in one fixed dimension, there exists a data structure*

1. *having a worst-case query time of $O(\log n + (\log n)^{d-2} \log\log n + k)$, where k is the number of answers to the query,*
2. *that has size $O(n(\log n)^{d-2})$,*
3. *and that can be built in expected time $O(n(\log n)^{d-2} \log\log n + n \log n)$.*

Proof. The proof follows from Theorem 12. The first term in the query time is the time needed to find the ranks of the endpoints of the d-th range restriction. The third term in the query time is k, because the sets of answers that are reported by the various persistent structures are disjoint. The $O(n \log n)$ term in the building time is needed to cover the case that $d = 2$. □

A similar result is present implicitly in Overmars [12]. He gives a deterministic data structure that stores a set of n points of the integer grid $[1 : n]^2$, that has size $O(n)$ and in which range queries with query regions $[a_1 : b_1] \times [a_2 : \infty)$ can be solved in $O(\log\log n + k)$ time. If we combine this result of Overmars with the technique of normalizing, Theorem 5 and

Bentley's method of adding range restrictions, then we get a deterministic data structure for the d-dimensional fixed height range searching problem, having the same size and query time as in Theorem 13.

Next, we consider the problem of *range searching for minimum*. As mentioned already, this is a generalization of a problem considered by Gabow, Bentley and Tarjan [8]. Let V be a set of n points in d-dimensional space. A query consists of a region $[a_1 : b_1] \times \ldots \times [a_{d-2} : b_{d-2}] \times [a_{d-1} : a_{d-1} + c] \times [a_d : \infty)$, and we have to find a point of V in this region having minimal d-th coordinate.

Again, we normalize the d-th coordinates of the points in V, i.e., we replace each d-th coordinate p_d by its rank. Clearly, if we query the normalized set, we get the right answer.

We apply Theorem 12 as follows. We take for PR the one-dimensional problem for a set of integers in the range $[1 : n]$, that asks for finding a minimal element that is at least equal to a given query value a. Let DS be the partially persistent version of the Van Emde Boas Tree with universe $[1 : n]$. (See Theorem 4.) This structure solves PR. A query can be solved in an arbitrary version in $O(\log\log n)$ time in the worst case, an element can be inserted in $O(\log\log n)$ amortized and expected time, and the size of the structure is bounded by $O(m)$ if it stores a set of size m.

We use DS for the normalized d-th coordinates of the points in V. Then we add $d-1$ range restrictions as in the previous example. This leads to the following result:

Theorem 14 *Let $d \geq 2$. For the d-dimensional problem of range searching for minimum, where the query regions have constant width in one fixed dimension, there exists a data structure*

1. *in which queries can be solved in worst-case time $O((\log n)^{d-2} \log\log n + \log n)$,*
2. *that has size $O(n(\log n)^{d-2})$,*
3. *and that can be built in expected time $O(n(\log n)^{d-2} \log\log n + n \log n)$.*

8 Adding arbitrary range restrictions

Until now we added range restrictions, one of which was of constant width. We now show that once a data structure for half-infinite range restrictions is available, we can get a structure for arbitrary range restrictions. The method is due to Edelsbrunner [7], who used it to get an efficient data structure for the range searching problem.

Let PR be a decomposable searching problem for a set V. Each object x in V has an additional parameter k_x. Let TPR be the searching problem that is obtained by adding a range restriction to PR.

Suppose we are given a data structure DS storing the set V, such that queries $TPR(q, [a : \infty), V)$ can be solved in $Q(n)$ time. Let $S(n)$ and $P(n)$ be the size and the building time of the structure DS, respectively. Let DS' be the data structure of the same type, in which queries $TPR(q, (-\infty : b], V)$ can be solved. This structure has the same complexity measures $Q(n)$, $S(n)$ and $P(n)$.

We give a data structure that solves queries $TPR(q, [a : b], V)$, where the range restriction is arbitrary.

The data structure: There is a balanced binary search tree T that stores the objects of V in its leaves, ordered according to their additional parameters k_x. Each non-root node w in T contains a pointer to an associated structure that is defined as follows. Let V_w be the subset of V that is stored in the subtree of w. If w is a left son, then this associated structure is a data structure DS_w storing the set V_w that allows queries with range restrictions $[a : \infty)$ to be solved. Otherwise, if w is a right son, the associated structure is a structure DS'_w for the set V_w that allows queries with range restrictions $(-\infty : b]$ to be solved.

The query algorithm: To answer a query $TPR(q, [a : b], V)$, do the following. Search in the tree T for a and b. Let u be that node in T for which the search path to a proceeds to its left son v, and the search path to b proceeds to its right son w. Then do a query $TPR(q, [a : \infty), V)$ in the structure DS_v, and do a query $TPR(q, (-\infty : b], V)$ in the structure DS'_w. Use the merge operator $\square$ belonging to PR to combine the answers obtained by these two queries, to obtain the final answer to the query.

Theorem 15 *In the above data structure, queries with an arbitrary range restriction can be answered in $O(Q(n) + \log n)$ time. The size resp. building time of the data structure is bounded by $O(S(n) \log n)$ resp. $O(P(n) \log n)$.*

Proof. The proof is of the same form as the previous ones, and is, therefore, left to the reader. In [7], the theorem is proved for the case where PR is the range searching problem. □

To illustrate Theorem 15, consider the d-dimensional range searching problem. In Theorem 13, we have given a data structure for this problem, in case one range restriction is half-infinite. Applying Theorem 15 to this structure gives the following result:

Theorem 16 *Let $d \geq 2$. For the d-dimensional range searching problem, there exists a data structure*

1. *having a worst-case query time of $O(\log n + (\log n)^{d-2} \log\log n + k)$, where k is the number of answers to the query,*
2. *that has size $O(n(\log n)^{d-1})$,*
3. *and that can be built in expected time $O(n(\log n)^{d-1} \log\log n)$.*

This result is similar to a result of Overmars [12]. He has, however, a deterministic data structure.

Finally, we apply Theorem 15 to the problem of range searching for minimum. Now, we combine Theorems 14 and 15, to obtain the following result:

Theorem 17 *Let $d \geq 2$. For the d-dimensional problem of range searching for minimum, there exists a data structure*

1. *in which queries can be solved in worst-case time $O((\log n)^{d-2} \log\log n + \log n)$,*
2. *that has size $O(n(\log n)^{d-1})$,*
3. *and that can be built in expected time $O(n(\log n)^{d-1} \log\log n)$.*

9 Concluding remarks

We have introduced new techniques for adding range restrictions to decomposable problems. The techniques give especially interesting results if we add several range restrictions, one of which has constant width. The techniques show the close relation between partially persistent data structures and structures for problems with constant width range restrictions.

We have given a general technique to make arbitrary data structures partially persistent. In Overmars [11, pages 158-159], another technique is given that works for static data structures that solve decomposable searching problems. The structures that result from this technique are almost identical to Bentley's structure for adding a half-infinite range restriction to decomposable searching problems. (See [1].) Hence, the relation between adding range restrictions and partially persistent data structures was already present in the literature.

We have given only few applications of our transformations. It should be possible to find other applications. Especially for problems that can be "normalized", new results may be obtained.

The most interesting results are obtained for range restrictions, one of which has constant width. In case arbitrary range restrictions are added, the space requirement increases by a factor of $O(\log n)$. (See Theorem 15.) An interesting direction for further research is to investigate whether in this case the space requirement can be reduced.

Finally, the data structures presented here are static. An interesting research direction is to investigate whether the techniques can be adapted to obtain dynamic structures.

Acknowledgement

We thank Paul Dietz for communicating the ideas of Section 2 to us, and Peter van Emde Boas for suggestions with respect to the proof of Theorem 2. Christian Schwarz is thanked for giving his comments on an earlier version of this paper. We thank Rajeev Raman for communicating the result of Theorem 4.

References

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

[2] J.L. Bentley. *Multidimensional divide and conquer.* Comm. ACM **23** (1980), pp. 214-229.

[3] B. Chazelle and L.J. Guibas. *Fractional cascading I: a data structuring technique.* Algorithmica **1** (1986), pp. 133-162.

[4] P.F. Dietz and R. Raman. *Persistence, amortization and randomization.* Tech. Report 353, University of Rochester, 1991.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R.E. Tarjan. *Dynamic perfect hashing.* Proc. 29-th Annual IEEE Symp. on Foundations of Computer Science, 1988, pp. 524-531.

[6] J.R. Driscoll, N. Sarnak, D.D. Sleator and R.E. Tarjan. *Making data structures persistent.* J. Comput. System Sci. **38** (1989), pp. 86-124.

[7] H. Edelsbrunner. *A note on dynamic range searching.* Bull. of the EATCS **15** (1981), pp. 34-40.

[8] H.N. Gabow, J.L. Bentley and R.E. Tarjan. *Scaling and related techniques for geometry problems.* Proc. 16-th Annual ACM Symp. on Theory of Computing, 1984, pp. 135-143.

[9] R. Klein, O. Nurmi, T. Ottmann and D. Wood. *A dynamic fixed windowing problem.* Algorithmica **4** (1989), pp. 535-550.

[10] K. Mehlhorn and S. Näher. *Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space.* Inform. Proc. Lett. **35** (1990), pp. 183-189.

[11] M.H. Overmars. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.

[12] M.H. Overmars. *Efficient data structures for range searching on a grid.* J. of Algorithms **9** (1988), pp. 254-275.

[13] N. Sarnak and R.E. Tarjan. *Planar point location using persistent search trees.* Comm. ACM **29** (1986), pp. 669-679.

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

[15] P. van Emde Boas. *Preserving order in a forest in less than logarithmic time and linear space.* Inform. Proc. Lett. **6** (1977), pp. 80-82.

[16] P. van Emde Boas, R. Kaas and E. Zijlstra. *Design and implementation of an efficient priority queue.* Math. Systems Theory **10** (1977), pp. 99-127.

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