Generalized static orthogonal range searching in less space

Christian Worm Mortensen

IT University Technical Report Series

TR-2003-33

ISSN 1600–6100 September 2003

ISSN 1600–6100

ISBN 87-7949-046-8

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone:	+45 38 16 88 88
Telefax:	+45 38 16 88 99
Web	`www.itu.dk`

Generalized static orthogonal range searching in less space

Christian Worm Mortensen
IT University of Copenhagen
E-mail: `cworm@itu.dk`

September 17, 2003

Abstract

We reduce the space usage on two problems related to generalized orthogonal range searching by almost a logarithmic factor. Our main result is that the generalized static orthogonal segment intersection reporting problem for n segment on an n times n grid can be solved in time $O(\log^2 \log n + k)$ for queries using space $O(n \log \log n)$. Here k is the number of reported segments.

1 Introduction

We define the generalized static orthogonal segment intersection reporting problem on a grid as follows. We are given a set S of n vertical and colored line segments with endpoints on a n times n grid. We must then preprocess the segments of S such that given a query $(x_1, x_2, y) \in [1 \dots n]^3$ [*] we can report the k colors represented among the segments from S which intersects the horizontal line segment between (x_1, y) and (x_2, y). In this paper we give a data structure for this problem which supports queries in time $O(\log^2 \log n + k)$ and uses space $O(n \log \log n)$. We also give a data structure which support queries in time $O(\log n \log^2 \log n + k)$ and uses space $O(n)$. In both cases the preprocessing time is $O(n \log \log n)$ w.h.p.[†] (w.h.p. because we use hashing [9]).

We define the generalized static orthogonal range reporting problem on a grid as follows. We are given a set P of n colored points on an n times n grid. We must then preprocess the points of P such that given a query $(x_1, x_2, y_1, y_2) \in [1 \dots n]^4$ with $x_1 \leq x_2$ and $y_1 \leq y_2$ we can report the set of colors among the points in $P \cap ([x_1 \dots x_2] \times [y_1 \dots y_2])$. Using a standard technique we convert our solutions for the generalized static orthogonal segment intersection reporting problem into data structures for this problem with the same query time but with an additional $O(\log n)$ factor on the space usage and preprocessing time (see table 2).

The model of computation is a unit-cost RAM with word size at least $\log n$ bits[‡]. We assume we have access to a sequence of truly random words.

1.1 A remark on problem variations

As mentioned, the solutions we provide are for n objects on an n times n grid. Previous solutions of the problems (see below) have either considered the case with coordinates in $\mathbb{R}^2$ or on a U times U grid for some $U \geq n$. Using a simple transformation our structures can be converted

[*]For integers $i \leq j$ we let $[i \dots j]$ denote the set of integers t for which $i \leq t \leq j$.

[†]We use w.h.p. as an abbreviation for with high probability. In a data structure with n elements we let high probability mean probability at least $1 - n^{-c}$ for any constant $c > 0$.

[‡]All logarithms in this paper are base 2.

Query time	Space usage	Source
$O(\log n + k)$	$O(n \log n)$	[13]
$O(\log^2 n + k)$	$O(n)$	[13]
$O(\log^2 \log n + k)$	$O(n \log \log n)$	New
$O(\log n \log^2 \log n + k)$	$O(n)$	New

Table 1: Solutions for the generalized static orthogonal segment intersection reporting problem.

Query time	Space usage	Source
$O(\log \log n + k)$	$O(n \log^2 n)$	[1]
$O(\log^2 n + k)$	$O(n \log n)$	[13]
$O(\log^2 \log n + k)$	$O(n \log n \log \log n)$	New
$O(\log n \log^2 \log n + k)$	$O(n \log n)$	New

Table 2: Solutions for the generalized static orthogonal range reporting problem.

into structures for $\mathbb{R}^2$ adding a term of $O(\log n)$ to the query time and a term of $O(n \log n)$ to the preprocessing time. Using another transformation we can instead get structures for an U times U grid if we add a term of $O(\log \log U)$ to the query time and a term of $O(n \log \log U)$ w.h.p. to the preprocessing time.

1.2 Comparison with other results

The generalized orthogonal range searching problems studied in this paper were introduced and motivated by Janardan and Lopez [13] together with other related problems, and these problems were further studied in [1, 12, 4]. The present paper builds on techniques used in these papers.

In [13] a solution to the generalized static orthogonal segment intersection reporting problem where segments were assumed to have endpoints in $\mathbb{R}^2$ was given. The solution had query time $O(\log n + k)$ and space usage $O(n \log n)$ or query time $O(\log^2 n + k)$ and space usage $O(n)$. The previously and now best known solutions for n segments on an n times n grid are summarized in table 1.

The generalized static orthogonal segment intersection reporting problem where all segments have different colors has been widely studied. Here the problem can be seen as reporting the set of segments intersecting a given query segment. Chazelle [5] considered the case where segments have endpoints in $\mathbb{R}^2$ and gave an optimal structure with query time $O(\log n + k)$ and space usage $O(n)$. In section 6.1 we remark, that the query time can be reduced to $O(\log^2 \log n + k)$ if endpoints lie on an n times n grid keeping the space usage on $O(n)$. Though I have not been able to find any references on this result it is not likely to be new.

In [13] a solution for the generalized static orthogonal range reporting problem with points in $\mathbb{R}^2$ was given. The solution had query time $O(\log n + k)$ and space usage $O(n \log^2 n)$ or query time $O(\log^2 n + k)$ and space usage $O(n \log n)$. Assuming points lie on a U times U grid Agarwal, Govindarajan and Muthukrishnan [1] improved the first result by reducing the query time to $O(\log \log U)$ keeping the space usage on $O(n \log^2 U)$. The previously and now best known solutions for n points on an n times n grid are summarized in table 2.

The generalized static orthogonal range reporting problem where all points have different colors has also been widely studied. Here the problem can be seen as reporting the set of points in a query rectangle. Chazelle [6] considered the case with points in $\mathbb{R}^2$ and gave a data structure with optimal query time $O(\log n + k)$ and a space usage $O(n \log^{\epsilon} n)$ for any constant $\epsilon > 0$. Alstrup, Brodal and Rauhe [3] considered the problem on an n times n grid and improved the query time to $O(\log\log n + k)$ keeping the same space usage.

The generalized static orthogonal range reporting problem where all points have the same color has also been studied. Here the problem can be seen as deciding if a given query rectangle is empty. Assuming points are in $\mathbb{R}^2$ Chazelle [6] has given an optimal solution for this problem using query time $O(\log n)$ and linear space $O(n)$.

1.3 Outline of paper

We start with preliminaries in section 2 where we also state lemma 1. This lemma is used in section 3 to give a dynamic one-dimensional structure. In section 4 this structure is used to make the structures from the introduction. In section 5 we review and reformulate some results on partial persistence and finally in section 6 we prove lemma 1.

2 Preliminaries

If T is a rooted tree and $v \in T$ is a node we let *height*(v) denote the height of v in T (leafs have height 0). For integers $i \leq j$ we let $X[i \dots j]$ denote an array indexed by $[i \dots j]$ and we let $X[i]$ denote the element of X at index i.

We let a VEB denote the data structure of van Emde Boas et. al. [16] combined with hashing [9]. Such a structure makes it possible to maintain a set $S \subseteq [1 \dots n]$ using time $O(\log\log n)$ w.h.p. for inserting or deleting an element and time $O(\log\log n)$ for predecessor queries. The space usage is $O(|S|)$.

Suppose we have a data structure supporting update and query operations where each update may perform a number of writes and reads on memory cells and queries may only perform reads. Such a structure is said to be *partial persistent* if each update increments a timestamp and each query takes a timestamp of the version of the data structure in which the query should be performed.

The results from the introduction builds on part 2 and 3 of lemma 1 below. We do not use part 1 but it may be of independent interest in connection with theorem 1 below. We defer the proof of the lemma to section 6. Suppose n is a power of two and that $B[1 \dots n]$ is an array of elements in $[0 \dots \log n]$ which are initially zero. Suppose further we can update B as follows. For $i \in [1 \dots n]$ and $y \in [0 \dots \log n]$ we can set $B[i] = y$. Further, given $i, j \in [1 \dots n]$ where $i \leq j$ and $y \in [1 \dots \log n]$ we can report the indices $e \in [i \dots j]$ for which $B[e] \geq y$.

Lemma 1. *Let k be the number of indices reported by a given query. Then:*

1. *We can maintain B with update time $O(\log\log n)$, query time $O(\log\log n + k)$ and space usage $O(n)$.*

Further let m be the number of updates performed. Then:

2. *We can make a partial persistent version of B with update time amortized $O(\log\log n)$ w.h.p., query time $O(\log^2 \log n + k)$ and space usage $O(m \log\log n)$.*

3. *We can make a partial persistent version of B with update time amortized $O(\log\log n)$ w.h.p., query time $O(\log n \log^2 \log n + k)$ and space usage $O(m)$.*

In all parts 1,2 and 3 a precomputed lookup table with $o(n)$ entries construct able in time $o(n)$ is needed.

3 A dynamic one-dimensional structure

In this section we describe how to maintain an array $A[1 \dots n]$ where each element in A has a color which is initially black. For given $i \in [1 \dots n]$ the array can be updated by assigning a color to $A[i]$. Further, for $i, j \in [1 \dots n]$ where $i \leq j$ we support a query (i, j) which returns the set of non-black colors among the elements $A[k]$ for which $i \leq k \leq j$. In this section we show:

Theorem 1. *The results for B in lemma 1 also applies to A except that in part 1 the update time becomes w.h.p..*

We assume w.l.o.g. (without loss of generality) that n is a power of two. We span a complete binary tree T over the elements of A from left to right. We will not distinguish between an element of A, its index in A and the corresponding leaf in T. For every node $v \in T$ we let $\mathit{span}(v)$ be the set of leafs descendant to v. We define $\mathit{left}(v)$ ($\mathit{right}(v)$) as the left-most (right-most) element in $\mathit{span}(v)$.

Below we describe how to make a data structure supporting queries of the form $(\mathit{left}(v), i)$ for $v \in T$ and $i \in \mathit{span}(v)$. A data structure supporting queries of the form $(i, \mathit{right}(v))$ can be made in a symmetric way. General queries (i, j) can be answered using these data structures as follows. If $i = j$ the query is easy so assume $i < j$. We identify the nearest common ancestor v of $A[i]$ and $A[j]$ in constant time. Let u be the left and w be the right child of v. The colors to report can be found by the two queries $(i, \mathit{right}(u))$ and $(\mathit{left}(w), j)$ (the same color may be reported by both queries and we filter such duplicate colors out).

What remains to describe is how to answer queries of the form $(\mathit{left}(v), i)$ for $i \in \mathit{span}(v)$. Again, if $\mathit{left}(v) = i$ the query is easy, so assume $\mathit{left}(v) < i$. For each internal node $v \in T$ we define (but do not store) the set $E(v)$ of non-black elements $e \in \mathit{span}(v)$ for which there is no element in $\mathit{span}(v)$ to the left of e with the same color as e. We observe that if $e \in E(v)$ then $e \in E(w)$ for the child w of v for which $e \in \mathit{span}(w)$ (because $\mathit{span}(w)$ is a subset of $\mathit{span}(v)$). It follows, that for each leaf $e \in T$ we can assign a y-coordinate $y(e)$ such that e is in $E(w)$ for exactly the ancestors w of e which is at distance at most $y(e)$ from e (if e is black we set $y(e) = 0$). The answer to the query is then the elements $e \in [\mathit{left}(v) \dots i]$ for which $y(e) \geq \mathit{height}(v)$. By maintaining an array B such that $B[i] = y(A[i])$ we can get these elements by performing a query in the structure provided by lemma 1. We will now show that each update of A modifies the y-coordinate of at most a constant number of element of A and that these can be found in time $O(\log \log n)$ w.h.p.. It follows that this is sufficient in order to prove theorem 1.

We first describe how to give a black element $e \in A$ a non-black color c. Let $v \in T$ be the nearest ancestor to e such that $\mathit{span}(v)$ contains an element different from e with color c and let e' be the leftmost such element. If v does not exists we set $y(e) = \log n$ and we are done. We note that we can find v (but not necessarily e') in time $O(\log \log n)$ if we for each color maintain a VEB containing the elements with that color. These VEBs are not used when answering queries and thus do not need to be made partial persistent. Further, it is these VEBs that make the update time w.h.p.. Let w be the child of v such that $e \in \mathit{span}(w)$. If w is the right child of v we set $y(e)$ to $\mathit{height}(w)$. If w is the left child of v we locate e' using the VEB we maintain for color c and then we first set $y(e)$ to $y(e')$ and next we set $y(e')$ to $\mathit{height}(w)$. We now argue for correctness by describing how the sets $E(u)$ change for $u \in T$ because of the update. Let $w' \neq w$ be the other child of v. Then $e' \in E(w')$ and further $e' \in E(v)$ before the update. We note that $\mathit{span}(w)$ contains no element with color c different from e and therefore e is inserted in $E(u)$ for the nodes $u \in T$ between e and w where $u \neq e$. If w is the right child of v then e is to the right of e' and thus no more changes occur. If w is the left child of v then e is to the left of e' and therefore e' is replaced by e in $E(u)$ for all ancestors u of v that contains e'.

We next describe how to color an element $e \in A$ with color c black. If e is black there is nothing to do so assume e is not black. Let v, w and e' be as before. If v does not exists or if w is the right child of v we set $y(e) = 0$. If w is the left child of v we locate e' using the VEB for color c and then we first set $y(e')$ to $y(e)$ and next we set $y(e)$ to 0. The argument of correctness is similar to before.

We finally note, that we can give a non-black element $e \in A$ a non-black color c by first coloring e black and then color e with c. As claimed it follows that changing the color of an element $e \in A$ changes the y-coordinate of at most a constant number of elements in A and these can be found in time $O(\log \log n)$ w.h.p..

4 Static two-dimensional structures

In this section we prove the results from the introduction by applying standard techniques to theorem 1. We first show how to solve the generalized static orthogonal segment intersection reporting problem on an n times n grid. Let S be the set of given vertical and colored segments. W.l.o.g. assume no segment in S is black and that all segments in S are disjoint. Let X be the structure from theorem 1 part 2 or 3 and let $Y[1 \dots n]$ be an array. We then enumerate the numbers in $[1 \dots n]$ in increasing order. Let y be an enumerated number. For each endpoint $(x, y) \in [1 \dots n]^2$ of a segment which does not have an endpoint with lower y-coordinate we set $X[x]$ to the color of the segment. After this we record the current timestamp of X in $Y[y]$. Next, for each endpoint $(x, y) \in [1 \dots n]^2$ of a segment which does not have an endpoint with higher y-coordinate we set $X[x]$ to black. The answer to a query $(x_1, x_2, y) \in [1 \dots n]^3$ in S where $x_1 \leq x_2$ can then be found by performing the query (x_1, x_2) in X at timestamp $Y[y]$. The time and space bounds from the introduction follows directly from theorem 1.

Next we consider the generalized static orthogonal range reporting problem. Let P be the set of given colored points. W.l.o.g. assume that no two different points in P have the same coordinates. First assume we only need to support *3-sided queries*, that is queries of the restricted form $(x_1, x_2, 1, y_2)$ for $x_1, x_2, y_2 \in [1 \dots n]$ and $x_1 \leq x_2$. Our structure for the generalized static orthogonal segment intersection reporting problem can be used for this as follows. We store the point $(x, y) \in P$ as the vertical segment between $(x, 1)$ and (x, y) in S. The answer to the query $(x_1, x_2, 1, y_2)$ in P is then the same as the answer to the query (x_1, x_2, y_2) in S.

We now describe how to convert the structure supporting 3-sided queries just described to one supporting general queries. We span a complete binary tree T over the y-axis of the grid from bottom to top. We will not distinguish between a leaf of T and its coordinate on the y-axis. Let $v \in T$ be a node in T and let $P_v \subseteq P$ be the points of P which have a y-coordinate descendant to v. We then in v store two structures supporting 3-sided queries. Given $x_1, x_2, y \in [1 \dots n]$ where $x_1 \leq x_2$ the first structure should support queries of the form $(x_1, x_2, 1, y)$ among the points in P_v and the second should support queries of the form (x_1, x_2, y, n) among the points in P_v (the queries in the second structure are indeed 3-sided if we turn things upside down). Now suppose we are given a general query $(x_1, x_2, y_1, y_2) \in [1 \dots n]^4$ in P with $x_1 \leq x_2$ and $y_1 \leq y_2$. To answer the query we first locate the nearest common ancestor v of y_1 and y_2 in T. If v is a leaf then $y_1 = y_2$ and the answer to the query can be found by performing a query (x_1, x_2, y_1, n) among the points in P_v. Suppose v is not a leaf and has lower child l and upper child u. Then the answer to the query can be found by performing a query (x_1, x_2, y_1, n) among the points in P_l and a query $(x_1, x_2, 1, y_2)$ among the points in P_u (the same color may be reported by both queries and we filter such duplicate colors out).

As described the structure has a large space usage because each 3-sided structure we store must support queries on a n times n grid even if it contains much less than n points. To overcome this problem we use the technique mentioned in section 1.1 which in case of m points allows us to reduce the grid size to m times m. This adds a term of $O(\log\log n)$ to the query time and a term of $O(n\log\log n)$ w.h.p. to the preprocessing time (coming from the usage of a VEB). But in our case this only changes the query and preprocessing time by constant factors. Since each point of P is stored in $O(\log n)$ structures supporting 3-sided queries it follows that we get an $O(\log n)$ factor on the space usage and preprocessing time as claimed.

5 Partial persistence

In this section we introduce a general way to make data structures partial persistent in the form of lemma 2 below. The section can be seen as a re formulation of known results and techniques. We refer the reader to Driscoll et. al. [10], Dietz [7] and Dietz and Raman [8] for more details.

Let D be an arbitrary deterministic data structure. We can model D as a set of nodes $v \in D$ which we can think of as allocated by the `new` operator in C++ or Java. The data in a node $v \in D$ is contained in an array $\mathit{array}(v)[1 \dots \mathit{size}(v)]$ where $\mathit{size}(v)$ is fixed on allocation of v. Each entry in $\mathit{array}(v)$ is a computer word and may contain a pointer to a node. In the RAM model $\mathit{size}(v)$ can be arbitrary whereas in the pointer machine model (which we do not consider) $\mathit{size}(v)$ must be constant. We assume D supports updates and queries as mentioned in the preliminaries and that queries always start in a fixed node of D. An update may in constant time per operation allocate a new node of any size or perform a read from or a write to an array in a node. A query may in constant time perform a read from an array in a node. We let n be an upper bound on the total number of writes to arrays and the total number of node allocations performed. On allocation of node v, the user must mark v as *big* or *small*. The user is only allowed to mark a node v as small if $\mathit{size}(v) = O(\log^c n)$ for a constant c.

Lemma 2. *Assume that for each small node $v \in D$ at most $\mathit{size}(v)$ pointers (the predecessor pointers) point to it. Then we can make D partial persistent using space $O(n)$. The time to update D is only increased by a constant factor but is made amortized and w.h.p.. The time to read an element in $\mathit{array}(v)$ in a query is $O(1)$ if v is small and $O(\log\log n)$ if v is big.*

Proof (sketch). Consider a node $v \in G$. We store in v both $\mathit{array}(v)$ and an additional array $\mathit{parray}(v)[1 \dots \mathit{size}(v)]$ using hashing [9]. For each value w written to $\mathit{array}(v)[i]$ at time t we store w with key t in a predecessor structure in $\mathit{parray}(v)[i]$. A read of $\mathit{array}(v)[i]$ at a specific time can then be performed by a predecessor query in $\mathit{parray}(v)[i]$.

Assume v is big. If we use a VEB as predecessor structure we get the properties we want except that in connection with updates we have to use time $O(\log\log n)$ w.h.p. to perform a write to an array in a node. To reduce this to $O(1)$ w.h.p. we use a standard trick. We group the writes into blocks with $\log\log n$ elements and from each block we only insert the element with the smallest timestamp in the VEB. Since the timestamps of the inserted elements are always increasing we can handle insertions in a block in constant time.

Assume now v is small. Since $\mathit{size}(v) = O(\log^c n)$ we can use the q*-heap of Fredman and Willard [17, 11] as predecessor structure. The q*-heap supports updates and predecessor queries in constant time per operation. For each element in a q*-heap in $\mathit{parray}(v)$ we place a pebble in v which can pay for a constant amount of time and space usage. When v contains $3\mathit{size}(v)$ pebbles we do as follows. We create a new copy v' of v with the elements of $\mathit{array}(v')$ initialized to $\mathit{array}(v)$ (and with $\mathit{parray}(v')$ initialized accordingly). Further, we take each of the at most $\mathit{size}(v)$ predecessor pointers of v and modify them to point to v' (we maintain in v the current set of predecessor pointers). $\mathit{size}(v)$ of the $3\mathit{size}(v)$ pebbles in v are used to pay for the work just described. $\mathit{size}(v)$ pebbles are used to place on v' and the remaining $\mathit{size}(v)$ pebbles are used to place on the small nodes in which we modify pointers to point to v' instead of v. It follows that the copying v to v' can be done amortized for free. □

6 Proof of lemma 1

This section is devoted to the proof of lemma 1. We prove each of the three parts of the lemma in different subsections.

6.1 Proof of lemma 1 part 3

In this section we prove part 3 of lemma 1. Görtz et. al. [2] have showed:

Lemma 3. *A VEB can be modified such that only $O(1)$ memory cells are written on each update.*

From this we get:

Lemma 4. *A VEB can be made partial persistent with update time amortized $O(\log \log n)$ w.h.p., query time $O(\log^2 \log n)$ and space usage $O(m)$ where m is the number of updates performed. Further, when we have made a query in a given version we can report the successors of the answer in that version in increasing order in constant time per element.*

Proof. We use the persistence technique of lemma 2. We put the VEB of lemma 3 into a single big node (we use a deterministic version of the VEB which do not use hashing). We then link the elements of the VEB together in order using small nodes which allows us to report the successors of a query answer in increasing order in constant time per element. □

Part 3 of lemma 1 follows from lemma 4: For each $y \in [1 \ldots \log n]$ we maintain a VEB of lemma 4 with the indices e of B for which $B[e] = y$. Given a query we can just search in each of the at most $\log n$ relevant VEBs. As another corollary to lemma 4 we get the following lemma announced in the introduction:

Lemma 5. *There exists a data structure for the generalized static orthogonal segment intersection reporting problem on a grid where all colors are different with query time $O(\log^2 \log n + k)$, space usage $O(n)$ and preprocessing time $O(n \log \log n)$ w.h.p.. Here k is the number of reported segments.*

Proof. We use the same algorithm as in section 4 except that we use the persistent VEB of lemma 4 instead of the structure from theorem 1. This is possible because all colors are different. □

6.2 Proof of lemma 1 part 1

In this section we give a proof sketch of part 1 of lemma 1. First assume B contains at most $O(\log^c n)$ elements different from 0 for a constant c. Using the variant of priority search trees of Willard [17] we can maintain B using constant time for updates, linear space and time $O(1+k)$ for queries where k is the number of reported elements. Plugging this structure with $c = 1$ into [14, lemma 4.5] (the lemma is implicit in [15]) gives a structure for B, without restriction on the number of elements different from 0 with update time $O(\log \log n)$, query time $O(\log \log n + k)$ and space usage $O(n \log n \log \log n)$. The space usage can be reduced to $O(n)$ using a standard trick: We group the elements of B into blocks with $O(\log^2 n)$ elements. For each block we insert a point with the largest y-coordinate into the structure just described and the points inside each block are kept in the variant of priority search tree of [17] with $c = 2$.

6.3 Proof of lemma 1 part 2

In this section we give a proof sketch of part 2 of lemma 1. The structure is obtained by applying lemma 2 to the structure described in section 6.2 except that we do not need the standard trick reducing the space usage. Looking into the structure of [14, lemma 4.5] we see, that all nodes in the structure can be made small in the terminology of lemma 2 with two exceptions: First, the structure uses (deterministic) VEBs to link elements together on the top level. But these VEBs are not used in connection with queries (only the links are used) and thus they do not need to be made partial persistent. Second, the structure contains a number of `bottom` arrays which need to be placed in big nodes. This is what increases the query time from $O(\log\log n + k)$ to $O(\log^2 \log n + k)$.

References

[1] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *In Proc. 10th European Sympos. Algorithms*, pages 17–28, 2002.

[2] Stephen Alstrup, Gerth Brodal, Inge Li Görtz, and Theis Rauhe. Time and space efficient multi-method dispatching. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 2002.

[3] Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 198–207. IEEE Computer Society Press, 2000.

[4] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Automata, Languages and Programming, 22th Colloquium*, pages 464–474, 1995.

[5] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Computing*, 15(3):703–724, August 1986.

[6] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.

[7] P. F. Dietz. Fully persistent arrays. In *Algorithms and Data Structures. Workshop (WADS '89)*, pages 67–74, Berlin - Heidelberg - New York, August 1989. Springer.

[8] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, 28–30 January 1991.

[9] M. Dietzfelbinger and Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In Michael S. Paterson, editor, *Automata, Languages, and Programming: 17th International Colloquium, Warwick University, England, July 16–20, 1990: Proceedings*, pages 6–19, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1990. Springer-Verlag.

[10] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[11] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994. See also FOCS'90.

[12] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, September 1995.

[13] R. Janardan and M. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry and Applications*, 3:39–69, 1993.

[14] Christian W. Mortensen. Fully-dynamic orthogonal range reporting on RAM. Technical Report TR-22, IT University of Copenhagen, April 2003.

[15] Christian W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[16] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[17] Dan E. Willard. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, June 2000.