Fast Algorithms for 3-D Dominance Reporting and Counting*

Qingmin Shi and Joseph JaJa
Institute for Advanced Computer Studies,
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742
{*qshi,joseph@umiacs.umd.edu*}

Abstract

We present in this paper fast algorithms for the 3-D dominance reporting and counting problems, and generalize the results to the d-dimensional case. Our 3-D dominance reporting algorithm achieves $O(\log n/\log\log n + f)$[1] query time using $O(n\log^{\epsilon} n)$ space , where f is the number of points satisfying the query and $\epsilon > 0$ is an arbitrarily small constant. For the 3-D dominance counting problem (which is equivalent to the 3-D range counting problem), our algorithm runs in $O((\log n/\log\log n)^2)$ time using $O(nlog^{1+\epsilon}n/\log\log n)$ space.

1 Introduction

Let S be a set of n points in $\Re^d$. The *dominance reporting* problem is to store S in a data structure so that the subset Q of points in S that dominate an arbitrarily given point q can be reported efficiently. Given two points $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$, we say p *dominates* q if and only if $p_i \geq q_i$ for all $1 \leq i \leq d$. Another related problem is the *dominance counting* problem. In this problem, the size k of Q only needs to be reported. Many computational geometry problems, such as the rectangle intersection problems, can be reduced to the dominance search problem.

In this paper, we provide faster algorithms than previously known for both 3-D dominance reporting and counting problems. Our model of computation is the RAM model as modified by Fredman and Willard [8]. In this model, it is assumed that each word is of size w and that the number of data elements n never exceeds 2^w, that is, $w \geq \log_2 n$. In addition, arithmetic and bitwise logical operations take constant time.

Before outlining our results, we give a brief overview of related literature. The 3-D dominance reporting is a special case of the 3-D orthogonal range search problem. Hence results for orthogonal range search apply immediately to our problem. In particular, a class of

*Supported in part by the National Science Foundation through the National Partnership for Advanced Computational Infrastructure (NPACI), DoD-MD Procurement under contract MDA90402C0428, and NASA under the ESIP Program NCC5300.

[1] In this paper, we always assume that the logarithmic operations are to the base two.

algorithms for orthogonal range reporting uses constant time for each reported point and polylogarithmic search time (see for example, [3, 4, 13, 15, 1].) The best known result for 3-D orthogonal range search is given by Alstrup and Brodal, which achieves $O(n \log^{1+\epsilon} n)$ space and $O(\log n + f)$ query time [1]. Several other data structures use less space but require larger query time. In [4], Chazelle gives two data structures, one with query time $O(\log^2 n + f \log\log(4n + f))$ and using space $O(n \log n \log\log n)$, and the other with query time $O(\log^2 n + f \log^{\epsilon}(2n/f))$ and using $O(n \log n)$ space. The best known dominance reporting algorithm is due to Makris and Tsakalidis [11], which follows the approach of Chazelle and Edelsbrunner [5] and achieves linear space and $O(\log n + f)$ query time.

Unlike dominance reporting, which seems to be inherently simpler than general orthogonal range reporting, the dominance counting problem is equivalent to the orthogonal range counting problem, if the dimension d is assumed to be constant. Indeed, it is easy to show that if a data structure of size $O(s(n))$ exists for answering any d-dimensional dominance counting query in $O(t(n))$ time, then any d-dimensional range counting query can be answered with $O(2^d s(n))$ space and using $O(2^d t(n))$ query time. Not many results are known for the multidimensional dominance counting problem. In [4], Chazelle uses a compressed form of the range tree [2] to achieve $O(\log n)$ query time and $O(n)$ space for the 2-D dominance counting problem. This result easily leads to a solution to the 3-D dominance counting with $O(n \log n)$ space, and $O(\log^2 n)$ query time. On the other hand, we can reduce the counting query problem to the aggregation range query problem in the same dimension by assigning a weight of 1 to each point. Willard in [15] shows how to combine the fusion tree [7] and its variant, called the q^*-heaps [8], and the fractional cascading technique [6] to achieve $O(n \log^{2+\epsilon} n)$ space and $O((\log n/\log\log n)^2)$ query time to handle 3-D aggregation range queries, where $\epsilon > 0$ is arbitrarily small constant.

In this paper, we establish the following results that provide faster algorithms for 3-D dominance reporting and counting:

- An algorithm for three-sided 2-D range reporting that achieves $O(n)$ space and $O(\log n/\log\log n + f)$ query time. This result is similar to Willard's modification of priority tree [15] but the algorithm is much simpler. This algorithm plays an important role in our 3-D dominance reporting algorithm.
- An algorithm for 3-D dominance reporting that uses $O(n \log^{\epsilon} n)$ space and $O(\log n/\log\log n + f)$ query time, where f is the number of points satisfying the query.
- An algorithm for 2-D dominance counting problem, which uses $O(n \log^{\epsilon} n)$ space and runs in $O(\log n/\log\log n)$ query time. This algorithm can be seen as an improvement on the query time over Chazelle's algorithm at the expense of $O(\log^{\epsilon} n)$ additional space.
- An extension of the 2-D dominance counting algorithm and 3-D dominance reporting algorithm to multidimensional dominance search, which leads to an $O((\log n/\log\log n)^{d-1})$ query time and $O(n \log^{\epsilon} n(\log n/\log\log n)^{d-2})$ space algorithm for the counting case and an $O((\log n/\log\log n)^{d-2} + f)$ query time and $O(n \log^{\epsilon} n(\log n/\log\log n)^{d-3})$ space algorithm for the reporting case, where d is the number of dimensions and $d \geq 2$ for the counting case and $d \geq 3$ for the reporting case.

In Section 2, we briefly discuss several known techniques, and, in the case of fractional cascading, some modifications to them. These techniques are used heavily in this paper. The three-sided 2-D range reporting algorithm is described in Section 3 while the description of our 3-D dominance reporting algorithm is given in Section 4. Our 2-D dominance counting algorithm is described in Section 5. Results in Sections 4 and 5 are extended to the multidimensional case in Section 6.

2 Preliminaries

To avoid tedious details in describing the algorithms, we assume for the rest of this paper that no two points have the same coordinate in any dimension. For simplicity, we call the point with the largest x-coordinate smaller than or equal to a real number r the *x-predecessor* of r and the one with the smallest x-coordinate larger than or equal to r the *x-successor* of r. The y- and z-predecessors(successors) are defined similarly.

2.1 Cartesian Trees

Cartesian trees, defined over a finite set of 2-D points, were first introduced by Vuillemin [14]. Let $p_1, p_2, ..., p_n$ be a set of n 2-D points sorted by their x-coordinates. The corresponding Cartesian tree C is defined recursively as follows. Let p_i be the point with the largest y-coordinate. Then p_i is associated with the root r of C. The left child of r is the root of the Cartesian tree built on $p_1, \ldots, p_{i-1}$ and the right child of r is the root of the Cartesian tree built on $p_{i+1}, \ldots, p_n$. The left(right) child does not exist if $i = 1(n)$. We call such a Cartesian tree an (x, y)*-Cartesian tree.*

An important property of the Cartesian tree is given by the following observation [9]:

Observation 1 *Consider a set S of 2-D points and the corresponding (x, y)-Cartesian tree C. Let $x_1 < x_2$ be the x-coordinates of two points in S, and let α and β be their respective vertices. Then the point with the largest y-coordinate among those whose x-coordinates are between x_1 and x_2 is stored in the nearest common ancestor of α and β.*

2.2 Fusion Trees

Fusion trees, developed by Fredman and Willard [7, 8], achieve sublogarithmic search time of one-dimensional data, and can be exploited to asymptotically speed up many algorithms as shown in [15]. In essence, this strategy achieves sublogarithmic search time by increasing the degree of the search tree as a function of n, where n is the size of the data. Typically, the degree c of the fusion tree satisfies $\log c = \Theta(\log \log n)$. As a result the depth of the tree is reduced to $\log n / \log \log n$. Using compressed key representation, the fusion tree allows the correct child of a node to be determined in constant time. The following Lemma, which we make use of in our results, is a simplified version of Corollary 3.2 in [15].

Lemma 1 *Assume that in a database of n elements, we have available the use of precomputed tables of size $o(n)$. Then it is possible to construct a data structure of size $O(n)$ space, which has a worst-case time $O(\log n / \log \log n)$ for performing member, predecessor and rank operations.*

Closely related to fusion trees is the notion of a *Q-heap* [8] that supports insert, delete, and search operations in constant time for small sets. Its main properties is given in the following lemma (the version presented here is taken from [15]).

Lemma 2 *Suppose S is a subset of cardinality $m < \log^{1/5} n$ lying in a larger database consisting of n elements. Then there exists a Q-heap data structure for representing S such that the Q-heap uses $O(m)$ space and enables insertions, deletions, member, and predecessor queries into the subset S to run in constant worst-case time, provided access is available to a precomputed lookup table of size $o(n)$.*

We note that the lookup table of size $o(n)$ can be shared by many Q-heaps built on subsets of S.

2.3 Fractional Cascading

Suppose we have a tree T of bounded degree c and rooted at node r such that each node v contains a list $L(v)$ of elements sorted increasingly by their values. Let n be the total size of all the lists stored in T, and let F be an arbitrary forest with p nodes consisting of subtrees of T determined by some of the children of r. F may be specified on line, i.e. not necessarily known during the preprocessing step. The following lemma is a direct derivation from the one given by Chazelle and Guibas [6] for identifying all the *successors* of a value x in the lists stored in F. The successor of x in a list $L(v)$ is defined as the first element of $L(v)$ whose value is greater than or equal to x.

Lemma 3 *There exits a linear size fractional cascading data structure that can be used to determine the successors of a given item x in the lists stored in F in $O(p \log c + t(n))$ time, where $t(n)$ is the time it takes to identify the successor of x in $L(r)$.*

The main component of a fractional cascading structure is the notion of the *augmented lists*. At each node v in T, in addition to the original list $L(v)$, we store another augmented list $A(v)$, which is a superset of $L(v)$ and contains additional copies of elements from the augmented lists associated with its parent and children. Each element h in $A(v)$ is coupled with a pointer to its successor $\sigma_{L(v)}(h)$ in $L(v)$. Since $A(v)$ is a superset of $L(v)$, we have $\sigma_{L(v)}(g) = \sigma_{L(v)}(\sigma_{A(v)}(g))$. Note that the elements in an augmented list $A(v)$ form a multiset $S(v)$, that is, a single element can appear multiple times in an augmented list. The elements in an augmented list are chained together to form a double linked list.

Let u and v be two neighboring nodes in T, say u is the parent of v. There exists a subset $B(u, v)$ of $S(u) \times S(v)$ such that each pair of elements $(g, h) \in B(u, v)$ have the same value. The pair of elements (g, h) is called a *bridge*. We store with the element g a pointer to h, and similarly we associate with h a pointer to g. We will call g a *down-bridge*, and h an *up-bridge*, associated with the edge (u, v). It is important to point out that each element in an augmented list can serve as at most one up-bridge or one down-bridge. Bridges respect the ordering of equal-valued elements and thus do not "cross". This guarantees that $B(u, v)$ can be ordered and hence the concept of *gap* presented next is well defined. In an ordered set $B(u, v)$, the bridge (g, h) appears after the (g', h') if and only if g appears after g' in $A(u)$. A *gap* $G_{(u,v)}(g, h)$ of bridge (g, h) is defined as the multiset of elements from both

$A(u)$ and $A(v)$ that are strictly between two bridges (g, h) and (g', h'), where (g', h') is the bridge that appears immediately before (g, h) in $B(u, v)$. Accordingly, we define the *up-gap* (*down-gap*) $G_{(u,v)}(g)$ ($G_{(u,v)}(h)$) as the subset of $G_{(u,v)}(g, h)$ containing elements from $A(u)$ ($A(v)$), preserving their orders in the respective augmented lists.

The fractional cascading structure maintains the invariant that the size of any gap cannot exceed $6c - 1$. Chazelle and Guibas provide an algorithm that can in $O(n)$ time construct such a data structure; moreover, the size of the data structure is $O(n)$.

Given a parent-child pair $(u, v) \in E$, suppose we know the successor $\sigma_{A(u)}(x)$ of a value x in $A(u)$, we follow $A(u)$ along the direction of increasing values to the next down-bridge g connecting u and v, cross it to the corresponding up-bridge h, and scan $A(v)$ in the opposite direction until the successor of x in $A(v)$ is encountered. $\sigma_{A(v)}(x)$ is guaranteed to be found in this process because the value associated with the up-bridge before h is smaller than x. The constraint of the gap size ensures that the number of comparisons required is $O(c)$. (A modification of the above data structure described in [6] achieves the result in Lemma 3. But this simpler one suffices for our purposes.)

By incorporating the Q-heap technique of Fredman and Willard [8], we can achieve constant search time per node (independent of c) for trees whose degree c is bounded by $\log^{\epsilon} n$, where n is the total size of all the lists, and ϵ is any positive constant smaller than $1/5$, at the cost of increasing the storage by a factor of c. We call this variation of fraction cascading structure *fast fractional cascading* and the original linear-space fractional cascading structure of Chazelle and Guibas *compact fractional cascading*.

We augment the original fractional cascading structure by adding two types of components to each augmented list $A(v)$. First, we store c additional pointers $p_1(g), p_2(g), \ldots, p_c(g)$ with each element g in $A(v)$ such that $p_i(g)$ points to the next down-bridge (possibly g itself) connecting v to w_i, where w_i is the ith child of v. Second, we build for each up-gap $G_{(u,v)}(h)$ a Q-heap structure $Q(h)$, containing elements in $G_{(u,v)}(h)$ with distinct values (choosing the first one whenever multiple elements have the same value). For large enough n and $c = o(\log^{1/5} n)$, we have $6c - 1 < \log^{1/5} n$; and therefore Lemma 2 is applicable. We have added c pointers to each element in the augment lists, whose overall size is $O(n)$. In addition, a global look-up table of size $o(n)$ is used to support the constant time operations on all the Q-heaps. And finally, since no two up-gaps in an augmented list overlap (because they correspond to the same edge in T), the Q-heaps consume a total of no more than $O(n)$ space.

Now suppose we have found $g = \sigma_{A(u)}(x)$ in $A(u)$. Let v be the ith child of u. By following the pointer $p_i(g)$, we can reach in constant time the next down-bridge in u and then its companion up-bridge h in v. Using $Q(h)$, we can find the successor of x in $G_{(u,v)}(h)$ in constant time.

Lemma 4 *If the degree c of T is bounded by $O(\log^{\epsilon} n)$, where ϵ is any positive constant smaller than $1/5$, then the fast fractional cascading structures allows the identification of the successors of a given value x in the list stored in F in $O(p + t(n))$ time, where $t(n)$ is time it takes to identify the successor of x in $L(r)$. This structure requires $O(cn)$ space.*

We show in a forthcoming paper that the storage of our modified fractional cascading data structure can be reduced to linear. However the above result is sufficient for our purposes in this paper.

3 Fast Algorithm for Three-sided 2-D Range Reporting

A three-sided range reporting query looks for the two-dimensional points $p = (p_x, p_y)$ in a data set such that $x_1 \leq p_x \leq x_2$ and $p_y \geq y$. Using McCreight's priority search tree [12], this problem can be solved in $O(\log n)$ time using $O(n)$ space. In [15], Willard improves this algorithm by increasing the degree of the priority search tree to $\sqrt{\log n}$ and applying the Q^*-heap structure. A global table is required in order to avoid the access of the tree nodes that do not have any point to report. We describe below another algorithm that achieves the same bound but is much simpler.

Instead of using a balanced tree, we construct an (x,y)-Cartesian tree C and chain its nodes together in increasing order of the x-coordinates using a double linked list L. Given a query (x_1, x_2, y), if we know the x-successor of x_1 and the x-predecessor of x_2 and their corresponding nodes α and β in the Cartesian tree C, then the nearest common ancestor γ of α and β stores the point with the largest y-coordinate among those points whose x-coordinates are between x_1 and x_2. By transforming C into the structure $\mathcal{D}(C)$ using the techniques of Harel et al [10], the node γ can be found in $O(1)$ time. After the node γ is located, we check whether the point stored there should be reported. If not, we stop exploring the subtree of γ. Otherwise, we find the nearest common ancestor γ' of α and the predecessor of γ in L, and the nearest common ancestor γ'' of β and the successor of γ in L, both of which can be identified in constant time. Then we recursively access the subtrees of γ' and γ''.

Lemma 5 *Let C be an (x, y)-Cartesian for a set of n two-dimensional points. Given a three-sided two-dimensional range query given as (x_1, x_2, y), with the two pointers to the left- and right-most nodes of C whose x-coordinates fall within the range $[x_1, x_2]$, then the query can be answered in $O(f)$ time using $\mathcal{D}(C)$ of size $O(n)$.*

The x-successor of x_1 and x-predecessor of x_2 can be found in $O(\log n/\log\log n)$ time if we build a fusion tree to index the nodes of the Cartesian tree according to the increasing order of the x-coordinates of their associated points. Thus we have an algorithm that can handle three-sided two-dimensional range queries in $O(\log n/\log\log n + f)$ time, using linear space.

4 Fast Algorithm for 3-D Dominance Reporting

The 3-D dominance reporting problem involves the determination of all the points that dominate the query point (q_x, q_y, q_z). In this section, we describe a faster algorithm than the one presented by Makris and Tsakalidis [11] at the expense of a factor of $\log^{\epsilon} n$ additional space. Our algorithm is inspired by Willard's improvement [15] on the priority search tree. We first give a brief overview of the algorithm of Makris and Tsakalidis.

4.1 The Makris-Tsakalidis Algorithm

This algorithm is based on the linear space data structure of Chazelle and Edelsbrunner [5], which handles queries in $O(\log^2 n + f)$ time. The primary data structure used is a binary search tree T built on the points in decreasing order of the z-coordinate. Each internal node v stores the *maximal set* $M(v)$ of the points stored in the subtree rooted at v and which are not stored in one of its ancestors. A maximal set of a point set S consists of the points in S whose projections onto the xy-plane are not dominated by any other projection.

At each node v, $M(v)$ is represented by three data structures. The first two are lists $L_1(v)$ and $L_2(v)$ that store the points in $M(v)$ in increasing order of the x-coordinates and decreasing order of the y-coordinates respectively. The third structure $D(v) = \mathcal{D}(C(v))$ is the derivation of the (x, z)-Cartesian tree $C(v)$ built on $M(v)$, and ignoring the y-coordinates. Lists L_1 and L_2 associated with all the primary tree nodes are chained together separately using fractional cascading to facilitate constant time search of these lists. Elements of $L_1(v)$, $L_2(v)$ and $D(v)$ that correspond to the same point are also linked together.

Given a query (q_x, q_y, q_z), the algorithm works as follows. First, we identify in $O(\log n)$ time the path Π from the root to the leaf node whose corresponding point has the smallest z-coordinate that is larger than or equal to q_z. Then we perform a pre-order traversal of the tree performing the following operations:

- For each node v visited, determine the points in $M(v)$ that need to be reported.
- If v does not have any point to be reported and is not on Π, do not visit any of its children.
- If v and its left child are both on the path Π, do not visit its right child.

The above restrictions guarantee that the number of nodes visited is bounded by $O(\log n + f)$.

For a node v that is visited but not on Π, we find the x-successor of q_x in $L_1(v)$ in $O(1)$ time, except when v is the root, which requires $O(\log n)$ time. If this point does not satisfy the query, then we are done with this node and its descendants. Otherwise, we follow $L_1(v)$ until a point not in Q is encountered and recursively visit the two children v.

At each node v on Π, we can easily in $O(1)$ time find left most and right most leaf nodes α and β of $C(v)$, the x- and y-coordinates of whose points satisfy the query. By Lemma 4, $D(v)$ allows us to report points in $M(v)$ in $O(f(v))$ time, where $f(v)$ is the number of points reported in $M(v)$.

4.2 Our Algorithm

To reduce the query complexity, we increase the degree of the primary tree T used in the previous section from 2 to $c = \log^{\epsilon} n$, where ϵ is an arbitrary positive constant smaller than $1/5$. As a result, the height of the tree as well as the length of the path Π is reduced to $O(\log n / \log\log n)$.

Each node v contains $c+4$ auxiliary structures. First, a Q-heap $K(v)$ is used to organize the keys separating the z-ranges of its children. Second, two lists $L_x(v)$ and $L_y(v)$ represent of the points in $M(v)$ sorted according to the x and y order respectively. The L_x-, L_y-lists of all the nodes are chained together separately using the fast fractional cascading structure.

Also we have the structure $D(v)$, which is the same as described in Section 4.1, associated with the maximal set of points $M(v)$ stored at v. In addition, $\log^\epsilon n$ (x, y)-Cartesian trees $C_1(v), C_2(v), \ldots, C_c(v)$ are built. $C_i(v)$ contains the points stored in the first i children of v, starting from the left. The leaf nodes of $C_i(v)$ are in decreasing order of the x-coordinates of their associated points; and each internal node stores the point whose y-coordinate is the largest among the points stored in its subtree. Actually, we are not storing the Cartesian trees themselves in v but rather the transformations $D_i(v) = \mathcal{D}(C_i(v))$ so that each such structure can be used to answer two-sided two-dimensional range queries in constant time. And finally, one fusion tree is built for each of the three lists stored at the root, and hence a member lookup in each of them can be performed in $O(\log n / \log\log n)$ time.

It is obvious that the total size of all the sorted lists and their corresponding fractional cascading structures is $O(n \log^\epsilon n)$. Each point can appear in at most one (x, z)-Cartesian tree and c (x, y)-Cartesian trees, and hence the overall storage cost of these Cartesian trees is $O(n \log^\epsilon n)$.

As we have seen before, we do not need to access the children of a node v if it contains no points to be reported. Also, the combination of the three fast fractional cascading structures and the (x, z)-Cartesian trees guarantees that $O(1 + f(v))$ time will be spent on each node v on Π. However, since the degree of our primary tree is no longer a constant, we cannot afford to access each child of v even if v has contributed some points. Doing so would result in a $f \log^\epsilon n$ term in the overall query time. We have to be sure at least one point will be reported during the visit of a node v before it is actually visited. This is where the (x, y)-Cartesian trees come into play.

Let v be the node being visited. Suppose v is on Π. (We always start by searching the root, which is always visited and always on Π.) We first use $D(v)$ to report $M(v) \cap Q$. If v is a leaf node, we are done. Otherwise, we find in $O(1)$ time, using $K(v)$, the child u of v that is also on Π. Suppose it is the ith child from the left. If $i = 1$, then we recursively search the subtree of u. If not, $D_{i-1}(v)$ is accessed as described in Section 3 to answer the query $(q_x \leq p_x, q_y \leq p_y)$. Since all the points p in the subtrees of the first $i-1$ children already satisfy $p_z \geq q_z$, we can be confident that the points reported in the search of $D_{i-1}(v)$ using the x- and y-coordinates do belong to Q. Note that the right-most node of $C_{i-1}(v)$ that satisfies $q_x \leq p_x$ should be identified in constant time (and hence Lemma 4 to be applicable). This condition can be satisfied by augmenting the fractional cascading structure for the L_x-lists to include the lists of tree nodes of $D_i(v), i = 1, 2, \ldots, \log^\epsilon n$ as well. This will not asymptotically increase the storage cost.

To ensure we reach in constant time each child of v that has at least one point to be reported, we maintain a vector of c bits initialized as zeros, and set the jth bit to 1 whenever a point from the subtree of the jth child of v is reported. After this process, the 1-bits correspond to the children of v, in addition to the child of v already identified on Π, which needs to be accessed. Let k be the number of such children. In order to find these children in $O(k)$ time, we maintain a table of size 2^c. The dth entry of this table stores a list of integers $(l, I_1, I_2, \cdots, I_l)$, where l is the number of 1-bits in the binary representation of d, and I_i is the position of the ith 1-bit in the integer d. Since $c = \log^\epsilon n$, each of these integers can be encoded in $O(\log\log n)$ bits. Therefore, each entry uses at most $O(\log^\epsilon n \cdot \log\log n) = O(\log n)$ bits; and thus each entry can be stored in one word and the size of the table is $O(n)$. Note that this single table is used for the entire searching process.

Now suppose v is not on Π. We then simply use $D(v)$ to report $M(v) \cap Q$, visit $D_c(v)$ to determine the children of v to be visited, and recursively access these children. The description of our algorithm is now complete.

It should be clear from the above discussion that the number of primary tree nodes visited is $O(\log n/\log\log + f)$ and $O(f(v))$ time is spent at each node v. Moreover, each point in Q will be reported at most twice. We therefore have the following theorem:

Theorem 1 *A set of n three-dimensional points can be stored in a data structure of size $O(n \log^{\epsilon} n)$, where ϵ is an arbitrary positive constant smaller than 1/5, such that any three-dimensional dominance reporting query can be answered in $O(\log n/\log\log n + f)$ time.*

5 An Improved Algorithm for 2-D Range Counting

In this section, we describe a fast algorithm for handling the two-dimensional dominance range counting problem.

Our solution uses ideas similar to the ones used in [4]. Instead of using a binary search tree as the skeleton of our data structure, we use a tree of degree $c = \log^{\epsilon} n$. Points are stored in the leaf nodes in order by decreasing x-coordinates. Each internal node v corresponds to an x-range $[x_1, x_2]$, where x_1 and x_2 are the x-coordinates of the points stored in its leftmost and rightmost leaf descendants. We call x_2 the *key* of node v denoted as $k(v)$. Each internal node v of the tree stores the list of keys of its c children and the pointers to them in a Q-heap $K(v)$. In addition, v also contains an auxiliary data structure to be used to gain information about the y-coordinates of those points in v's subtree. We will describe this structure soon. In addition to our tree structure, we have a fusion tree that occupies $O(n)$ space and performs rank searches on the n points sorted on decreasing y-coordinates in $O(\log n/\log\log n)$ time.

The auxiliary data structure of each internal node v consists of two parts. First, similar to Chazelle's bit vector, a vector is used to record, for each point in the subtree rooted at v in order of decreasing y-coordinates, which of the c child subtrees it belongs to. Obviously, we need $\log c$ bits to encode this information. We call these $\log c$ bits a *microword* and the vector of microwords a *router*, denoted as $r(v)$. Since a word can accommodate $\log n/\log c$ microwords, $m(v) = \lceil n(v)\log c/\log n \rceil$ words are needed to store $r(v)$ if v has $n(v)$ leaf descendants. In addition, we count, for each i between 1 and $m(v) - 1$ and j between 1 and c, the number of microwords stored in the first i words of $r(v)$ whose values are between 0 and $j - 1$, and store them in a $(m(v) - 1) \times c$ two-dimensional array, called a *counter*, denoted as $c(v)$.

Now, we analyze the storage cost of the overall data structure. The tree itself is clearly of size $O(n)$. Since each microword corresponds to a point, which is represented once at each of the $\log_c n$ levels of the tree, there are a total number of $n \log n/\log c$ microwords. The word cost of all the routers is thus $O(n)$. Let T denote the set of internal nodes. The total size of all the counters is $\sum_{v \in T}(m(v) - 1)c = O(cn)$. The overall size of our data structure is thus $O(n \log^{\epsilon} n)$.

A query given as (q_x, q_y) is answered by recursively exploring the tree. We will show that, only a constant time is spent at each level of the tree, and hence the complexity of our query algorithm is $O(\log n/\log\log n)$. First, we use the fusion tree to find the rank i of the successor of q_y. This implies that the points that correspond to the first i microwords in $r(h)$ have their y-coordinates no less than q_y, where h is the root of the tree. If the x-range of the root h is included in $[q_x, \infty)$, then i is the answer. Otherwise, suppose the jth child v of h is the rightmost one such that $k(v) \geq q_x$ (the value of j can be decided in $O(1)$ time using $K(v)$). The number of points in the subtrees rooted the first j children of v that should be counted are those whose y-cooridnates are no less than q_y. This number can be computed by counting how many points that correspond to the first i microwords in $r(h)$ go to its first j children. Let $g = \lceil i \log c / \log n \rceil - 1$. The number of points that correspond to the first $g \log n / \log c$ microwords can be immediately found in $c(v)[g][j]$. The number of points corresponding to the remaining $i - g \log n \log c$ microwords that should be counted can be obtained by looking up a table, which costs only a constant amount of storage. The number of remaining points to be counted are stored in the subtree of the $(j+1)$th child v of h. Hence we repeat the above procedure on node v. The only difference is that, we already know the rank of q_y in $r(h)$. By Lemma 4, we can compute the rank of q_y in $r(v)$ in constant time without asymptotically increasing the storage cost. It is now clear the overall complexity of our algorithm is $O(\log n / \log\log n)$.

Theorem 2 *There exist an algorithm that solves two-dimensional dominance counting query and range counting query $O(\log n / \log\log n)$ time. This algorithm uses $O(n \log^{\epsilon} n)$ space.*

6 Fast Algorithms for Multidimensional Dominance Searching

Using the results obtained for the two-dimensional dominance counting case, we can achieve a better query complexity for the three-dimensional dominance counting problem. We build a search tree of degree $c = \log^{\epsilon/2} n$ on the points sorted on decreasing z-values. Let $D(S)$ denote the data structure described in the previous section for answering two-dimensional dominance counting queries. For each internal node v, we construct c data structures $\{D(S(i)) | i = 1, \cdots, c\}$, where $S(i)$ is the set of points stored in the subtrees rooted at the first i children of v. To answer a query given as (q_x, q_y, q_z), we identify $O(\log n / \log\log n)$ two-dimensional data structures, one at each level, each corresponding to a node whose z-range is completely covered by $[q_z, \infty)$. This takes $O(\log n / \log\log n)$ time. We then obtain the answers to the two-dimensional dominance query given by (x, y) by searching each of the two-dimensional data structures identified. The complexity of this algorithm is $O((\log n / \log\log n)^2)$. Let $s_1, s_2, \cdots, s_l$ be the size of the point sets stored at the l two-dimensional data structures at a certain level. Then the storage cost of that level is $O(\sum_{i=1,\cdots,l} s_i \log^{\epsilon/2} s_i) = O(\log^{\epsilon/2} n \sum_{i=1,\cdots,l} s_i) = O(n \log^{\epsilon} n)$. Thus the overall storage cost is $O(n \log^{1+\epsilon} n / \log\log n)$. Generalization of this result to higher dimensions is straightforward.

Similar technique can be used for to obtain fast algorithms for the multi-dimensional dominance reporting case, in which we use the data structure described in Section 4 to handle the lowest three dimensions.

Theorem 3 *There exist data structures such that any d-dimensional dominance counting query can be answered in $O((\log n / \log\log n)^{d-1})$ time using $O(n \log^{\epsilon} n \; (\log n / \log\log n)^{d-2})$*

space for $d \geq 2$, where ϵ is an arbitrarily positive constant smaller than 1/5. Similarly, an arbitrary d-dimensional reporting query can be answered in $O((\log n/\log\log n)^{d-2} + f)$ time using $O(n \log^{\epsilon} n(\log n/\log\log n)^{d-3})$ space for $d \geq 3$.

References

[1] S. Alstrup and G. S. Brodal. New data structures for orthogonal range searching. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.

[3] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, Aug. 1986.

[4] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–463, June 1988.

[5] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 3:113–126, 1987.

[6] B. Chazelle and L. J. Guibas. Fractional Cascading: I. A data structure technique. *Algorithmica*, 1(2):133–162, 1986.

[7] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[8] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[9] H. N. Gabow, J. L. Bently, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[10] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[11] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6), 1998.

[12] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[13] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.

[14] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[15] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion three. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.