An $O(n)$-Space $O(\log n/\log\log n + f)$-Query Time Algorithm for 3-D Dominance Reporting [*]

(Technical Report CS-TR-4508 and UMIACS-TR-2003-77)

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 a linear-space algorithm for handling the *three-dimensional dominance reporting problem*: given a set S of n three-dimensional points, design a data structure for S so that the points in S which dominate a given query point can be reported quickly. Under the variation of the RAM model introduced by Fredman and Willard [8], our algorithm achieves $O(\log n/\log\log n + f)$ query time, where f is the number of points reported. Extensions to higher dimensions are also reported.

1 Introduction

Given a set S of d-dimensional points, we wish to store these points in a data structure so that, given a query point $q \in \Re^d$, the points in S that are dominated by q, which we will refer to as *proper* points, can be reported quickly. A point $p = (p_1, p_2, \ldots, p_d)$ is dominated by $q = (q_1, q_2, \ldots, q_d)$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, d$. Without loss of generality, we assume that no two points in S have the same x-, y-, or z-coordinates. A number of geometric retrieval problems involving iso-oriented objects can be reduced to this problem (see for example [6]). Solutions to this problem have also been used recently in dealing with the so-called *temporal range reporting queries* on time-series data [12].

We will use n to represent the input size, f to represent the output size, and ϵ to represent an arbitrarily small positive constant. The constant c is defined to be $\log^\epsilon n$. Given a set S of d-dimensional points $(x_1, x_2, \ldots, x_d)$, a point with the largest x_i-coordinate smaller than or equal to a real number α is called the x_i-predecessor of α and the one with the smallest x_i-coordinate larger than or equal to α is called the x_i-successor of α.

Our model of computation is the RAM model as modified by Fredman and Willard [8]. In this model, it is assume that each word consists of b bits and that the number n of data elements never exceeds 2^b, i.e., $b \geq \log n$[1]. In addition, arithmetic and bitwise logical operations take constant time.

[*]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.

In [4], Chazelle and Edelsbrunner proposed two linear-space algorithms to handle the 3-D dominance reporting problem. The first achieves $O(\log n + f \log n)$ query time and the second achieves $O(\log^2 n + f)$ query time. These two algorithms were later improved by Makris and Tsakalidis [10] to yield $O((\log\log n)^2 \log\log\log n + f \log\log n)$ and $O(\log n + f)$ query time respectively. In [11], we improved the query performance of the $O(\log n + f)$ time algorithm of Makris and Tsakalidis to $O(\log n / \log\log n + f)$ query time but at the expense of increasing the storage cost by a factor of $\log^\epsilon n$.

In this paper, we show how to reduce the space to linear while maintaining the same query performance as in [11], thus obtaining the fastest query time algorithm using linear space. In Section 2, we provide some previously known results that will be heavily used in this paper. We briefly review the non-linear space algorithm (that appeared in [11]) in Section 3 and describe the new linear-space algorithm in Section 4.

2 Preliminaries

2.1 Q-heaps and Fusion Trees

Q-heap and *fusion trees* achieve sublogarithmic search time on one-dimensional data. The following two lemmas are shown in [7] and [8] respectively.

Lemma 2.1. *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 fusion tree data structure of size $O(n)$ space, which has a worst-case $O(\log n / \log\log n)$ time for performing member, predecessor and rank operations.*

Lemma 2.2. *Suppose Q is a subset with cardinality $m < \log^{1/5} n$ lying in a larger database S consisting of n elements. Then three exists a Q-heap data structure of size $O(m)$ that enables insertion, deletion, member, and predecessor queries on Q to run in constant worst-case time, provided access is available to a precomputed table of size $o(n)$.*

Note that in Lemma 2.2, the look-up table of size $o(n)$ is shared among all the Q-heaps built on subsets of S.

2.2 Fast Fractional Cascading

Let T be a tree rooted at w and having a maximum degree of c at each node. A node v in T contains a sorted list $L(v)$ of elements. The total number of elements in all the lists is n. Such a tree is called a *catalog tree* [3]. An *iterative search* on T is defined as follows.

> Given a query item x, and an embedded tree F of T, which is rooted at w and has p nodes, find the predecessor of x in each of the lists associated with the nodes of F.

The fractional cascading technique [5] can be used to organize T and its associated lists so that an iterative search can be performed in $O(t(n) + p \log c)$ time, in which $t(n)$ is the time it takes to identify the predecessor of x in $L(w)$ and $\log c$ is the cost of finding the predecessor of x in each of the remaining $p - 1$ lists. This technique uses $O(n)$ space. Note that when c is not a constant, the time spent at each node is not a constant either.

In [11], we combined the Q-heap technique and the fractional cascading to achieve constant search time at each node when the maximum degree of T is polylogarithmic in n. The storage cost, however, is increased to $n \log^\epsilon n$. This result was improved in [13] by reducing the storage cost to linear while maintaining the same query performance. This result is summarized in the following lemma.

Lemma 2.3. *Let T be a catalog tree rooted at w with a total number n of items in its associated lists, and let $c = \log^{\epsilon} n$ be the maximum degree of a node in T. There exists a $O(n)$-space data structure derived from T such that an iterative search operation specified by a query item x and an embedded tree F with p nodes can be performed in $O(t(n) + p)$ time, where $t(n)$ is the time it takes to identify the predecessor of x in $L(w)$.*

We call this data structure the *fast fractional cascading* structure.

2.3 Handling 3-Sided 2-D Reporting Queries Using Cartesian Trees

A Cartesian tree [14] C is a binary tree defined on a finite set of 2-D points, say $p_1, p_2, \ldots, p_n$, sorted by their x-coordinates. The root of this tree is associated with the point p_i with the largest y-coordinate. Its left child is the root of the Cartesian tree built on $p_1, \ldots, p_{i-1}$, and its right child is the root of the Cartesian tree built on $p_{i+1}, \ldots, p_n$.

In [11], we explained how the Cartesian trees can be modified to efficiently handle the 3-sided 2-D reporting queries[2], i.e. to identify the points (x, y) that satisfy $a \leq x \leq b$ and $y \geq d$, where a, b, and d are three numbers provided by the query. We doubly link all the nodes in C according to an in-order traversal of C. Given a query (a, b, d), we first identify the two nodes α and β in C that correspond respectively to the x-successor p_i of a and the x-predecessor p_j of b. Then we find in constant time the nearest common ancestor γ of α and β using one of the algorithms provided in [9, 2]. If the y-coordinate of the point p_k stored at γ is greater than or equal to d, then we report that point and recursively search the subsequences $p_i, \ldots, p_{k-1}$ and $p_{k+1}, \ldots, p_j$. Note that the two nodes corresponding to p_{k-1} and p_{k+1} can be reached in constant time using the doubly linked list. If p_k does not satisfy the query, then we stop searching the subtree rooted at γ. Thus we have the following lemma.

Lemma 2.4. *Let C be the Cartesian tree for a set of n 2-D points and augmented with a doubly linked list. A 3-sided 2-D range query given as (a, b, d) can be handled in $O(t(n) + f)$ time using $\mathcal{D}(C)$ of size $O(n)$, where $t(n)$ is the time to identify the nodes corresponding to the successor of a and the predecessor of b, and $\mathcal{D}(C)$ is a transformation of C to support the nearest common ancestor search in constant time.*

Without causing confusion, in the rest of this paper, whenever we refer to a Cartesian tree C, we mean its transformation that is suitable for 3-sided 2-D range reporting queries. We can also use the Cartesian tree to index a set of d-dimensional points $(x_1, x_2, \ldots, x_d)$ based on any two of their dimensions. If the points are first sorted by the x_i-dimension and are recursively picked during the construction of the tree according to their x_j-coordinates, then the resulting Cartesian tree is called (x_i, x_j)-Cartesian tree.

3 An $O(n \log^{\epsilon} n)$-Space $O(\log / \log\log n + f)$-Query Time Algorithm

In this section, we briefly review the data structure proposed in [11], upon which our new algorithm is based. The skeleton of our data structure is a balanced search tree of degree $c = \log^{\epsilon} n$ (thus of height $O(\log n / \log\log n)$) on the *decreasing* z-coordinates. A Q-heap $K(v)$ is used to index the keys

[2]A similar idea is used by Alstrup et al. [1] in handling a special case of the 3-sided 2-D reporting queries when the points are taken from $[N] \times \Re$.

stored at each internal node v. Let $M(v)$ be the *maximal set* of points stored in the subtree rooted at v, excluding the points that are already associated with the ancestors of v (a maximal set of a point set R consists of the points in R whose projections onto the xy-plane are not dominated by any other projections). In addition to the Q-heap, each node v is associated with several Cartesian trees: an (x,z)-Cartesian tree $D(v)$ and c (x,y)-Cartesian trees $D_1(v), D_2(v), \ldots, D_c(v)$. The (x,z)-Cartesian tree $D(v)$ stores the *maximal set* of points associated with v; and $D_i(v)$ stores the union of the maximal sets associated with the leftmost i children of v. It is easy to realize that the storage cost of this data structure is $O(n \log^{\epsilon} n)$, since the tree T and the associated Q-heaps requires $O(n)$ space, and each point is stored in at most one (x, z)-Cartesian tree and c (x, y)-Cartesian trees.

To answer a 3-D dominance query specified by the point (x_0, y_0, z_0), we first identify, in $O(\log n/\log\log n)$ time using the Q-heaps, the path Π from the root to the leaf that corresponds to the z-successor of z_0. We then search the tree recursively, starting from the root. Note that we do not visit any node that is in a subtree rooted at the right sibling of a node on Π. For each node v visited, finding the proper points in $M(v)$ is equivalent to a 3-sided 2-D range query due to the properties of a maximal set (see [10] for more details) and thus can be handled in $O(f(v))$ time using $D(v)$ (assuming that we already know the leftmost and rightmost leaf nodes of $D(v)$ that are in the query range). Suppose the kth child of v from the left is on Π ($k = c + 1$ if v is not on Π). Note that we cannot afford to visit the each of the leftmost $k - 1$ child of v. Instead, we visit such a *proper* child u only if there is at least one proper point in $M(u)$. To decide which children to visit, we search $C_{k-1}(v)$ for proper points, and record in a vector of c bits, each corresponding to a child of v, which children need to be visited next. This vector is then converted using a look-up table of size $O(n)$ to a list of indices corresponding to the proper children.

We build a fusion tree on the x-coordinates to index the points stored in each of the $c + 1$ Cartesian trees associated with the root w. Furthermore, we connected all the Cartesian trees using a modified fractional cascading structure of size $O(n \log^{\epsilon} n)$. These additional data structures do not asymptotically increase the storage cost and allow each Cartesian tree associated with a non-root node to be searched in constant time, plus the time it takes to retrieve proper points.

4 A Linear-Space Algorithm with $O(\log n/\log\log n + f)$ Query Time

Two factors contributed to the non-linear space requirement of the data structure in [11]. First, the modified fractional cascading technique described there uses non-linear space. Second, with each node v, c (x, y)-Cartesian trees were needed to ensure that we can in constant time find the proper children. The first difficulty no longer exists since we now have at hand the fast fractional cascading structure [13]. Let $m(v)$ be the number of proper children, we show in the rest of this section that these children can be identified in $m(v)$ time without the (x,y)-Cartesian trees.

Let $u_1, u_2, \ldots, u_c$ be the children of v in T and let $M(u_i) = \{(x_{i,1}, y_{i,1}, z_{i,1}), (x_{i,2}, y_{i,2}, z_{i,2}), \ldots, (x_{i,n_i}, y_{i,n_i}, z_{i,n_i})\}$ for $i = 1, 2, \ldots, c$. Since $M(u_i)$ is maximal, we can assume without loss of generality that $x_{i,1} < x_{i,2} < \cdots < x_{i,n_i}$ and $y_{i,1} > y_{i,2} > \cdots > y_{i,n_i} > y_{i,n_i+1} = -\infty$. Now consider the projections of these points to the x-y plane. We define a set $G(v)$ of vertical segments in the x-y plane as follows: $G_i(v) = \{(x_{i,j}; y_{i,j+1}, y_{i,j}) | j = 1, \ldots, n_i\}$ and $G(v) = \bigcup_{i=1,\ldots,c} G_i(v)$. Let $N(v) = |G(v)|$. Figure 4 gives an example of such a set of segments, with segments from different children depicted using lines of different thicknesses.

Lemma 4.1. *Let $s = (x_0, +\infty; y_0)$ be a semi-infinite horizontal line. For each $i \in \{1, \ldots, c\}$, u_i contains at least one point whose projection to the x-y plane dominates (x_0, y_0) if and only if there exists a vertical segment $(x_{i,j}; y_{i,j+1}, y_{i,j}) \in G(v)$ that intersects s and furthermore, there is at most one such vertical segment.*

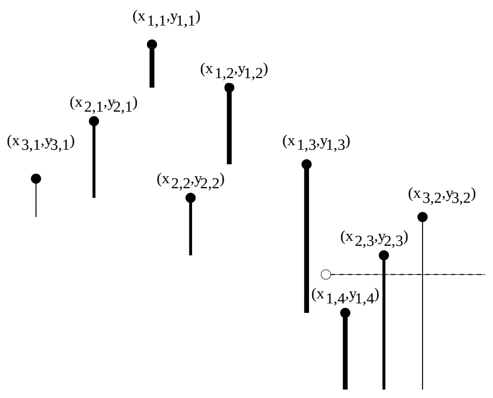

Figure 1: The dominance query.

Proof. If a segment $(x_{i,j}; y_{i,j+1}, y_{i,j}) \in G(v)$ intersects s, then by definition, $(x_{i,j}, y_{i,j})$ dominates (x_0, y_0). On the other hand, suppose $(x_{i,j}, y_{i,j})$ dominates (x_0, y_0). Then either the segment $(x_{i,j}; y_{i,j+1}, y_{i,j})$ intersects s, or $y_0 < y_{j+1} \neq -\infty$, which means $j < n_i$. Therefore, the point $(x_{i,j+1}, y_{j+1})$ also dominates (x_0, y_0). Repeating this process ensures that we can find one vertical segment corresponding to u_i which intersects s. Finally, since the projections of the vertical segments corresponding to u_i do not overlap, s can only intersect one of these segments. $\square$

As a result of Lemma 4.1, identifying the proper children of v in $O(m(v))$ time can be achieved if we design an indexing scheme on $G(v)$, such that given s and an integer k, the segments from $G_i(v)$, with $i = 1, 2, \ldots, k-1$, which intersect s can be reported in $O(m(v))$ time. Associating v with $G(v)$ dose not asymptotically increase the overall storage cost, as long as $G(v)$ can be indexed in linear space, since the points in a non-root node are replicated only once at its parent.

Note that this is not simply a segment intersection reporting problem, as we cannot afford to report every segment in $G(v)$ that intersects s. This is the case because some of them may come from other than the leftmost $k-1$ children of v. Nevertheless, we can solve this problem by performing a segment intersection *counting* query followed by a table look-up operation. This segment intersection counting query is defined as computing the number of segments in $G(v)$ that intersect s (these segments do not necessarily need to come from the leftmost $k-1$ children of v).

We first discuss the table look-up operation. The list of proper children of v with respect to a 3-D dominance reporting query can be represented as a vector $r = (m(v), I_1, I_2, \ldots, I_{m(v)})$, where I_i, with $i = 1, \ldots, m(v)$, is the index of a proper child. Obviously, the bit-cost of this vector is $O(c \log c)$. Once we obtain such a vector, we can retrieve from it the index of the proper children one by one in $O(m(v))$ time.

Lemma 4.2. *The vector r is uniquely defined by the y-rank g of y_0 in the set of endpoints of $G(v)$, the value of k, and the number h of segments in $G(v)$ which intersect s.*

Proof. Let $y_1, y_2, \ldots, y_{N(v)}$ be the list of y-coordinates of the points in $G(v)$ in sorted order. Consider two consecutive such y-coordinates y_j and y_{j+1}. It is easy to see that the list of segments in $G(v)$ sorted from left to right which intersect the query segment $s = (x_0, +\infty; y_0)$, with y_0 varying in the range $[y_j, y_{j+1})$ remains the same. Among these segments, the rightmost h intersects s. And, knowing k, we can uniquely remove those coming from the rightmost $c-k$ children of v. $\square$

Since only one segment from $G_i(v)$ could possibly intersect s, the value of h is bounded by c. The value of k is also bounded by c and the y-rank of y_0 is bounded by $N(v)$. Therefore, we can create a look-up table containing $N(v)$ words, each corresponding to a possible y-rank of y_0. The $\log n$ bits of each such word is sufficient to record for each possible combination of k and h, the vector r that has been uniquely determined ($c^3 \log c < \log n$ for large enough n).

Among the three indices g, k, and h, g can be computed in constant time by applying the fast fractional cascading technique on the y-coordinates of the points in $\bigcup_{i=1,\ldots,c} M(u_i)$, and k is known using the Q-heap associated with v. Thus we only need to show that the value of h can be computed in constant time. We first give the following lemma.

Lemma 4.3. *A 3-D dominance counting query on a set R of $m < \log^{\epsilon} n$ points can be handled in constant time using $O(m)$ space.*

Proof. An answer to such a query is uniquely decided by the ranks of the query point in R with respect to the x-, y-, and z-coordinates, which can computed by applying the Q-heap techniques in constant time and $O(m)$ space. These three ranks are used to index a $m \times m \times m$ look-up table to obtain the correct answer. Since $m < \log^{1/5} n$, any possible answer can be represented using only O(log log n) bits. Therefore all m^3 (not necessarily distinct) possible answers can be compacted into a single word ($m^3 \log\log n < \log n$ for large enough n). □

We now explain how to compute the value of h in constant time. We partition the endpoints of the segments in $G(v)$ into n/c horizontal stripes $P_1, \ldots, P_{n/c}$, each containing c endpoints. Let $B_1, \ldots, B_{n/c-1}$ be the boundaries such that B_i separates P_i and P_{i+1}. We associate with each boundary the maximal subset S_i of $G(v)$ such that every segment in S_i intersects B_i, and with each stripe P_i the maximal subset T_i of $G(v)$ such that every segment in T_i crosses the entire stripe P_i. We also denote the subset of segments in $G(v)$ that are completely inside P_i as R_i. Note that a segment can belong to up to $n/c - 1$ subsets associated with the boundaries and up to $n/c - 1$ subsets associated with the stripes. However, the size of each S_i or T_i is bounded by c. The total size of all the subsets associated with the boundaries is equal to the number of intersections between the segments in $G(v)$ and the $n/c - 1$ boundaries. Notice that each $G_i(v)$, with $i = 1, \ldots, c$, contributes at most $n/c - 1$ such intersections. Thus the total size of all the subsets associated with the boundaries is $O(N(v))$. Similarly, the total size of all the subsets associated with the stripes is also $O(N(v))$. And finally, $\sum_{i=1,\ldots,n/c} |R_i| = O(N(v))$.

Given a query segment $s = (x_0, +\infty; y_0)$, we can determine, using the fast fractional cascading structure, the stripe P_{j+1} within which it falls. Without loss of generality, suppose this is not the first nor the last stripe. Therefore, the two boundaries B_j and B_{j+1} exist. The number of segments in $G(v)$ that intersect s can be computed as $A + B - C + D$, where A and B are respectively the numbers of segments in S_j and S_{j+1} that intersect s, C is the number of segments in T_j that intersect s, and D is the number of segments in R_i that intersect s. Computing A and B is equivalent to a 2-D dominance counting query on the lower endpoints of the segments in S_j and the upper endpoints of the segments in S_{j+1} respectively; computing C is equivalent to a 1-D dominance counting query on the x-coordinates of the segments in T_{j+1}; and computing D is equivalent to a 3-D dominance counting query on the segments $(x; y_1, y_2)$ in R_{j+1} in the form $(x \geq x_0, y_1 \leq y_0, y_2 \geq y_0)$. Since the size of each of the sets involved is bounded by $O(\log^{\epsilon} n)$, by Lemma 4.3, these computation can be performed in $O(1)$ time and in linear space.

Theorem 4.1. *There exists a linear-space algorithm to answer the three-dimensional dominance reporting queries in $O(\log n / \log\log n + f)$ time.*

Using the techniques discussed in [11], we can extend the above results to higher-dimensions, increasing both the query and space bounds by a factor of $\log n / \log\log n$ for each addition dimension.

Theorem 4.2. *A d-dimensional dominance reporting query can be handled in $O((\log n/\log\log n)^{d-2}+f)$ time using $O(n(\log n/\log\log n)^{d-3})$ space, for $d \geq 3$.*

References

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

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of Latin American Theoretical Informatics*, pages 88–94, 2000.

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

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

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

[6] H. Edelsbrunner and M. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

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

[10] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[11] Q. Shi and J. JaJa. Fast algorithms for 3-d dominance reporting and counting. Technical Report CS-TR-4437, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[12] Q. Shi and J. JaJa. Fast algorithms for a class of temporal range queries. To appear in *Proceedings of Workshop on Algorithms and Data Structures (WADS'03)*, Ottawa, Canada, July, 2003.

[13] Q. Shi and J. JaJa. Fast fractional cascading and its applications. Technical Report CS-TR-4502, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

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