Maintenance of Geometric Extrema*

David Dobkin
Department of Computer Science
Princeton University
Princeton, NJ 08544

Subhash Suri
Bell Communications Research
445 South Street
Morristown, N.J. 07960

January 2, 1990

Abstract

Let S be a set, $f : S \times S \rightarrow R^+$ a bivariate function, and $f(x, S)$ the *maximum* value of $f(x, y)$ over all elements $y \in S$. We say that f is *decomposable* with respect to the maximum if $f(x, S) = \max\{f(x, S_1), f(x, S_2), \ldots, f(x, S_k)\}$ for any decomposition $S = \cup_{i=1}^{i=k} S_i$. Computing the maximum (minimum) value of a decomposable function is inherent in many problems of computational geometry and robotics. In this paper, we present a general technique for updating the maximum (minimum) value of a decomposable function as elements are inserted into and deleted from the set S. Our result holds for a *semi-online* model of dynamization: when an element is inserted, we are told how long it will stay. Applications of this technique include efficient algorithms for *dynamically* computing the diameter or closest pair of a set of points, minimum separation among a set of rectangles, smallest distance between a set of points and a set of hyperplanes, and largest or smallest area (perimeter) rectangles determined by a set of points. These problems are fundamental to application areas such as robotics, VLSI masking, and optimization.

1 Introduction

A variety of problems in computational geometry and robotics involve computing the maximum (minimum) value of a bivariate function $f : S \times S \rightarrow R^+$, defined over a set of geometric objects. Generally, S is a collection of primitive objects such as points, lines, rectangles, hyperplanes, or spheres, and f is an elementary function such as L_p distance, area or perimeter. For instance, given a set of points S in the d-dimensional Euclidean space, the diameter of S is the maximum value of the distance function $f(x, y)$, for $x, y \in S$. Other examples include the closest pair in a set of points, largest or smallest area (perimeter) rectangles determined by a planar set of points, minimum distance between a set of points and a set of hyperplanes, and smallest separation among a set of orthogonal rectangles.

The computational geometry literature contains numerous efficient algorithms for solving such problems in the static mode, that is, for the case of a fixed input set. Not much is

*A preliminary version of this paper appeared in the *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, 1989,* under the title "Dynamically Computing the Maxima of Decomposable Functions, with Applications

available, however, in the way of dynamic algorithms. Often one wants to solve a problem for a sequence of "related" sets of input, rather than just one fixed set of input, in which case executing a static-case algorithm for each input set separately is likely to be inefficient. A dynamic algorithm simulates a sequence of related problem instances by allowing insertions and deletions into an underlying set. The strength of a dynamic algorithm lies in its ability to "update" the solution after an insertion or deletion, rather than having to recompute it.

Given their versatility, it is no surprise that dynamic algorithms are often significantly more difficult to obtain than static ones. In fact, despite the fundamental nature of the problems such as the diameter and closest pair, no efficient dynamic algorithms were known for them. In this paper, we obtain efficient dynamic algorithms for diameter, closest pair, and the other problems mentioned in the abstract. But let us first review some of the previous work on dynamic algorithms.

1.1 Previous Results

A systematic study of dynamization in the realm of computational geometry was enunciated by Bentley and Saxe [2]. They focused on the important class of problems, called *decomposable search problems*, and showed that under mild assumptions one can convert a static data structure into a semi-dynamic one, allowing insertions but no deletions, with a logarithmic increase in cost. The conversion cost in their scheme is generally quite high if deletions are also allowed. Bentley and Saxe, however, were mainly interested in data structures for query-answering (i.e., *univariate* functions), and consequently their scheme is not amenable to the kind of problems we are interested in, namely, the pairwise maxima in a set.

Overmars and van Leeuwen [17] extended the basic approach of Bentley and Saxe to dynamize problems related to configurations rather than just points. They developed efficient algorithms for dynamically computing convex hulls, maximal elements and intersection of halfspaces. But even these extended techniques do not seem to apply to geometric maximization (minimization) problems such as diameter and closest pair.

In [16], Overmars considered *order decomposable set problems*, a class of problems especially amenable to divide-and-conquer. Among the problems considered by him is the closest-pair problem in two dimensions. He obtained an $O(n)$ time algorithm for updating the closest-pair after each insertion or deletion. The best bound for dynamically maintaining the diameter of a planar set of points was also $O(n)$ per insertion or deletion. In fact, no sublinear-time update procedure was known for any of the problems mentioned in the abstract.

1.2 Our Main Results

Our main result is an efficient algorithm for updating the maximum (minimum) value of a decomposable function $f : S \times S \rightarrow R^+$ as elements are inserted into and deleted from S. Our result holds for a *semi-online* model of dynamization: when an element is inserted, we are told how long it will stay. Thus, the semi-online model should be viewed as a "cross" between the offline and the online models, which essentially mixes online insertions with offline deletions. The exact form of the semi-online model can vary depending on how one chooses to specify the sequence of deletions. In the model stated above, we have chosen to disclose the deletion-sequence "on the fly." In another variation, we may specify the complete order in which the elements will be deleted. Notice that specifying the order does not disclose the time step at which a deletion actually occurs since deletions are intermixed with online insertions. In the paper, we will focus exclusively on the first definition of the semi-online model, although all our results hold for the second version as well. We point out that our semi-online model is similar to a model used by Gabow and Tarjan [13] where they obtain a linear-time algorithm for the disjoint set union problem under the assumption that the sequence of "unions" is known in advance but "finds" are online.

We believe that a semi-online model may be an interesting middle ground when one of the operations is more difficult to perform online than others. In our case, deletion is the harder operation and, in fact, for most of the problems considered in this paper, efficient algorithms for handling even the online deletions (without any insertions) are not known. While the goal of obtaining efficient online algorithms for these problems remains elusive at this time, our results show that mixing online insertions with offline deletions is not much more difficult to handle than the insertions alone.

Our central theorem can be stated as follows. Suppose that we are given a set S, a data structure $D(S)$, and a decomposable function f such that (1) [*Preprocessing*] $D(S)$ can be computed in time $P(n)$, and (2) [*Query*] for any x, $f(x,S)$ can be determined from $D(S)$ in time $Q(n)$, where $|S| \leq n$. Then we can process a semi-online sequence of n insertions and deletions and update the maximum (minimum) value of f at an amortized cost of

$$O\left(\frac{P(n)\log n}{n} + Q(n)\log n\right)$$

per insertion or deletion. This theorem is presented in Section 2.

We then apply this result to derive sublinear-time update procedures for a variety of problems in computational geometry, robotics and VLSI masking. These applications are described in Section 3, and summarized in the table below; d denotes the dimension of the space.

Problem	*Amortized Update Time*	*Best Static Time*
Diameter	$O(\log^2 n)$ for $d=2$ $O(\sqrt{n}\log^2 n)$ for $d=3$ $O(n^{1-\frac{1}{d(d+3)+4}}\log^2 n)$ for $d \geq 4$	$O(n\log n)$ $O(n^{3/2}\log n)$ $O((n\log n)^{2-\frac{2}{d(d+3)+4}})$
Closest Pair	$O(\log^2 n)$ for $d=2$ $O(\sqrt{n}\log^2 n)$ for $d=3$ $O(n^{1-\frac{1}{d(d+3)+4}}\log^2 n)$ for $d \geq 4$	$O(dn\log n)$
Minimum Separation of Rectangles	$O(\log^2 n)$ for $d=2$	$O(n\log n)$
Minimum Distance between Points and Hyperplanes	$O(n^{1/2}\log^{3/2} n)$ for $d=2$ $O(n^{2/3}\log^{7/3} n)$ for $d=3$ $O(n^{1-\frac{1}{d(d+3)+4}}\log^2 n)$ for $d \geq 4$	$O(n^{1.41})$ $O(n^{5/3}\log^2 n)$ $O((n^{2-\frac{2}{d(d+3)+4}}\log^2 n)$
Largest and Smallest Rectangles	$O(\log^4 n)$ for $d=2$	$O(n\log^2 n)$

In Section 4 we consider extensions of the main theorem. We obtain a uniform result for dynamically solving a number of decomposable search problems, with deletions. We also discuss the (limited) applicability of our result to *non-decomposable* problems. Specifically, we show that the largest area (perimeter) *empty* rectangle in a set of points can be maintained in sublinear amortized time. Conclusions and open problems are discussed in Section 5.

2 The Main Theorem

Given a set S and a symmetric bivariate function $f : S \times S \rightarrow R^+$, we consider the problem of updating the maximum value of f as elements are inserted into and deleted from S. Equivalently, we compute the maximum value of f for a sequence of subsets of S where adjacent subsets differ by exactly one element. If $x \in S$ is an element and $T \subseteq S$ a subset, then we use the (slightly abusive) notation

$$f(x, T) \;=\; \max_{y \in T} f(x, y).$$

The following definitions formalize the notions of decomposability and computability of f.

- f is *decomposable with respect to the maximum* if $f(x, S) \;=\; \max_{1 \leq i \leq k} f(x, S_i)$, for any $x \in S$ and $S = \cup_{i=1}^{i=k} S_i$.
- f is *(P, Q)-computable with respect to the maximum* if for all $T \subseteq S$ there exists a data structure $D(T)$ such that

  [*Preprocessing*] $D(T)$ can be constructed in time $P(|T|)$, and

  [*Query*] for any $x \in S$, $f(x, T)$ can be determined from $D(T)$ in time $Q(|T|)$.

We assume that P and Q are monotone nondecreasing functions, with $P(n) = \Omega(n)$ and $Q(n) = \Omega(\log n)$. These assumptions are easily satisfied by the type of problems considered in this paper.

We call a sequence of insert and delete instructions *semi-online* if the instructions are received online but the duration of each element is announced at the time of its insertion. That is, if x is inserted at time t and deleted at time t', then t' becomes known at time t. Times t and t' actually denote indices of these updates in the sequence rather than "real" times. An element is *active* from the time of its insertion to the time of its deletion. The *active set* S_t consists of all the elements active at time t.

Given a bivariate function f and a semi-online sequence L of inserts and deletes, the *maximal history of f with respect to L* is the sequence of values

$$V_t \stackrel{\text{def}}{=} \max_{x, y \in S_t} f(x, y), \qquad t = 1, 2, \ldots, |L|.$$

Similarly, we can define the *minimal* history of f. Our result is stated in the following theorem.

Theorem 1 *Let f be a symmetric bivariate function that is (P, Q)-computable and decomposable with respect to the maximum (minimum), and let L be a semi-online sequence of n insert and delete instructions. Then we can compute the maximal (minimal) history of f with respect to L in time $O(P(n)\log n + nQ(n)\log n)$, for an amortized cost of*

$$C(n) \ = \ O\left(\frac{P(n)\log n}{n} \ + \ Q(n)\log n\right)$$

per insertion and deletion.

The rest of this section outlines a proof of Theorem 1. Our proof consists of an algorithm and its analysis. Without loss of generality, we consider the maximal history; the proof for the minimal history is identical. In the following discussion, we assume that the length of the update sequence, namely, n is known in advance. This is not a substantial restriction since we can always guess a value for n, say, $n = m$ and when m updates have been processed, re-initialize the algorithm with $n = 2m$. It is easy to see that this does not affect the asymptotic complexity of our algorithm. We begin with a key definition.

Given a subsequence $L' \subseteq L$, an element x is called a *long insertion of L with respect to L'* if x remains active during the entire time-span of L'; that is, x is inserted before the first insertion of L' and it is deleted after the last deletion of L'. The main idea behind the proof is to partition L into a hierarchy of subsequences so as to isolate large groups of long insertions.

2.1 Hierarchy

We begin by splitting L into α_1 blocks, each of length $|L|/\alpha_1 = n/\alpha_1$; the optimal value of α_1 will be determined later. A block is just a consecutive sequence of instructions of L. These α_1 blocks are processed in their sequential order. Consider one of these blocks, say, L_1. Let F_1 be the set of long insertions of L with respect to L_1. The key observation is that the set F_1 remains "frozen" during the processing of L_1 and that $|F_1| \le |L|$. In time $P(F_1)$ we compute the data structure $D(F_1)$, after which $f(x, F_1)$ can be determined in time $Q(|F_1|)$ for any element x. To process the instructions of L_1, we subdivide L_1 into α_2 subblocks, each of length $|L_1|/\alpha_2 = n/\alpha_1\alpha_2$. Each of these subblocks are then processed recursively.

We continue to subdivide L in this manner, and build a hierarchy of frozen data structures $D(F_1)$, $D(F_2)$, …, $D(F_k)$. At the ith level of the hierarchy, each block of insert and delete instructions has size

$$|L_i| \ = \ \frac{n}{\alpha_0\alpha_1\alpha_2\cdots\alpha_i},$$

where $\alpha_0 = 1$ and $L_0 = L$. The left half of Figure 1 illustrates the hierarchy. There are altogether α_i blocks at the ith level corresponding to each block L_{i-1} of the level $i-1$. F_i will denote the current frozen set at level i; the intended time step will be clear from the context.

Suppose that the hierarchy consists of k levels, where an optimal value of k will be determined later. A block of instructions at the kth level is processed directly, i.e., without making further recursive calls. These instructions operate on a set F_{k+1}.

As the processing of various blocks is completed, the hierarchy is updated accordingly. In particular, when a blocks of level i, say, L_i is completed, the topmost $k-i+1$ levels of the hierarchy are rebuilt. This involves recomputing the frozen sets and their associated data structures. We do this by the (upward) recursive subdivision of the next-in-line blocks at level i. In the following, we explain how to process an insert and delete instruction and update the hierarchy.

2.2 Updates

We start with the updating of frozen sets. During the processing of L_k, the current block of instructions at level k, all sets except F_{k+1} remain fixed. The set F_{k+1} and its associated data structure $D(F_{k+1})$ are recomputed after every insertion and deletion of L_k. Consider the time step t when L_k is completed. The completion of L_k generally implies the completion of several other blocks of instructions at lower levels of the hierarchy. (Recall that every time α_j blocks of instructions are completed at level j, one block of instructions gets completed at level $j-1$.) Let i, for $1 \leq i \leq k$, be the lowest level in the hierarchy where a block of instructions is completed at time t. Then we need to rebuild the hierarchy at time t for all the levels between i and k.

Let L_i be the block of instructions just completed at the ith level and let $L_i^{'}$ be the next block of instructions to be carried out. Suppose further that the parent sequence of L_i and $L_i^{'}$ is L_{i-1} (at the level $i-1$). The new frozen set for the ith level, denoted by $F_i^{'}$, is the set of long insertions of L_{i-1} with respect to $L_i^{'}$. Let $A(L_i)$ denote the set of elements that are inserted but not deleted during L_i, and let $B(L_i^{'})$ denote the set of elements in $F_i \cup A(L_i)$ that are scheduled for deletion during $L_i^{'}$. Then it is easily seen that

$$F_i^{'} \;=\; (F_i \cup A(L_i)) - B(L_i^{'}). \tag{1}$$

The set $B(L_i^{'})$ is propagated upward, where its elements are partitioned to form frozen sets of levels j, for $j > i$. $L_i^{'}$, the new block of instructions at the level i, is split into α_i subblocks. Let L_{i+1} be the first among these subblocks. Then the new frozen set at level $i+1$ consists of precisely those elements of $B(L_i^{'})$ that are *not scheduled* for deletion by L_{i+1}. The remaining elements of $B(L_i^{'})$ move on to level $i+2$, where they are either placed into the frozen set or passed further to the next higher level, and so on.

This completes the discussion of the updating of the frozen sets in the hierarchy as blocks of instructions are completed. A straightforward consequence of this hierarchical scheme is that the frozen sets form a partition of the current active set, as the following proposition states.

Proposition 1 *At any time t, $S_t = \bigcup_{i=1}^{k+1} F_i$. The set S_t in fact is a disjoint union of all the frozen sets.*

Proof. Let L_k be the block of instructions being processed at time t and level k. Consider the "genealogy" tree of L_k. This tree is a directed path whose nodes are blocks of instructions $L_0, L_1, \ldots, L_{k-1}, L_k$, where L_{i+1}, a block at level $i+1$, is a child (subblock) of L_i, a block at level i. Then it is easy to see that an element x is placed in the frozen set of the lowest level j for which the condition "x is a long insertion of L_{j-1} with respect to L_j"

holds. Thus, an element of S_t belongs to exactly one frozen set at time t, which completes the proof. ▮

Next, we describe how to update the maximum value of f as a new insertion or deletion is processed. We introduce the notation

$$W_t(F_i) \stackrel{\text{def}}{=} \max_{x \in F_i} \max_{1 \le j \le i} f(x, F_j), \tag{2}$$

so that $W_t(F_i)$ is the maximum value of f achieved between the elements of F_i and any F_j, for $j \le i$, at time t. Our algorithm computes the history of f by maintaining the values of $W_t(F_i)$'s.

Proposition 2 *At any time t, $V_t = \max_{1 \le i \le k+1} W_t(F_i)$.*

Proof. By Proposition 1, if the maximum value of f at time t is achieved by the pair of elements $x, y \in S_t$, then both x and y belong to some frozen sets. Suppose that $x \in F_i$ and $y \in F_j$, where $i \le j$. Then, $W_t(F_j) = f(x, y)$. ▮

Our algorithm maintains the values of $W_t(F_i)$'s as insertions and deletions cause the frozen sets to be changed. The set F_i is updated every time a block of instructions L_i is completed at level i. The new value of $W_t(F_i)$ is calculated as indicated by its definition in (2). In particular, the value of $W_t(F_{k+1})$ is updated after every insertion and deletion. In the following subsection, we show that these updates can be implemented within the time bound stated in Theorem 1.

2.3 Analysis

For notational convenience, we write

$$n_i = \frac{n}{\alpha_0 \alpha_1 \alpha_2 \cdots \alpha_{i-1}},$$

so that $|F_i| \le |L_{i-1}| = n_i$. The following proposition establishes the complexity of updates at the kth level.

Proposition 3 *Following an insertion or deletion at time t, we can update $D(F_{k+1})$ and $W_t(F_{k+1})$ in time $O((k+1)[P(n_{k+1}) + n_{k+1}Q(n)])$.*

Proof. Let z be the element inserted or deleted at time step t. If the instruction is to delete z, then our decomposition scheme ensures that z currently belongs to F_{k+1}. We therefore update F_{k+1} by either adding or deleting z, and then compute the new data structure $D(F_{k+1})$, at the cost of $P(n_{k+1})$. If the instruction is to insert z, then the new value of $W_t(F_{k+1})$ is computed as[1]

$$W_t(F_{k+1}) := \max \{W_{t-1}(F_{k+1}),\ f(z, S_t)\}.$$

[1]Notice that F_{k+1} on the left-hand side of the equation represents the set at time t while the one on the right-hand side represents the set at time $t-1$ (i.e., before the insertion or deletion of z).

Since S_t is the union of all frozen sets and f is decomposable, we have the following equality:

$$f(z, S_t) \;=\; \max_{1 \le i \le k+1} f(z, F_i).$$

We can determine the value of $f(z, S_t)$ by querying data structures $D(F_i)$'s, for $i = 1, 2 \ldots, k+1$, at the cost of at most $Q(n)$ each. Thus, updating $D(F_{k+1})$ and $W_t(F_{k+1})$ following an insertion takes $O(P(n_{k+1}) + (k+1)Q(n))$ time.

If the instruction is to delete z, then the new value of $W_t(F_{k+1})$ is computed as

$$W_t(F_{k+1}) \;:=\; \max_{x \in F_{k+1}} f(x, S_t).$$

In this case, we compute $f(x, S_t)$ for each of the elements of F_{k+1}. Since $|F_{k+1}| \le n_{k+1}$ and each $f(x, S_t)$ can be computed in time $(k+1)Q(n)$, the cost of updating $D(F_{k+1})$ and $W_t(F_{k+1})$ following a deletion is $O(P(n_{k+1}) + n_{k+1}(k+1)Q(n))$. (Notice that if z is not involved in the previous maximum value of $W_t(F_{k+1})$, then this work is unnecessary; the value of $W_t(F_{k+1})$ remains unchanged.) This completes the proof. ▮

Next, we analyze the cost of maintaining $D(F_i)$ and $W_t(F_i)$ at an arbitrary level of the hierarchy. Since F_i remains fixed during an ith level block of instructions L_i, the updates to $D(F_i)$ and $W_t(F_i)$ occur only once per block of instructions L_i.

Proposition 4 *Following the completion of a block of instructions L_i at time t and level i, we can update $D(F_i)$ and $W_t(F_i)$ in time $O(P(n_i) + n_i Q(n))$.*

Proof. Observe that computing the data structure $D(F_i)$ only takes time $P(n_i)$ once the new frozen set has been determined. We show that F_i and $W_t(F_i)$ both can be updated in time $O(n_i Q(n))$. Our algorithm maintains the following invariant:

(*) *Each element $x \in F_i$ stores with it the set of values $f(x, F_j)$ for $j = 1, 2, \ldots, i$.*

The invariant is initialized for an element at the time of its insertion; Proposition 3 takes into account the cost of computing $f(z, F_j)$ for $j = 1, 2, \ldots, k+1$. Thus, associated with each element $x \in S_t$ is a linear array that stores these $k+1$ values. Given these values, we can compute $W_t(F_i)$ in time $O(kn_i)$. Since we later choose $k < \log n$ (see Proposition 5), this cost is dominated by $O(n_i Q(n))$; recall that $Q(n) = \Omega(\log n)$. We break up the analysis into two cases depending upon whether or not i is the lowest level in the hierarchy where a block of instructions is completed at time t.

First, assume that i is the lowest level in the hierarchy where a block of instructions is completed at time t. Suppose that after the completion of L_i, the next block of instructions to be carried out at level i is $L_i^{'}$. Furthermore, let L_{i-1} (at level $i-1$) be the parent block of instructions for both L_i and $L_i^{'}$. Then the new frozen set for the ith level, denoted by $F_i^{'}$, is computed as

$$F_i^{'} \;=\; (F_i \cup A(L_i)) - B(L_i^{'}),$$

where $A(L_i)$ is the set of elements that are inserted but not deleted during L_i, and $B(L_i^{'})$ is the set of elements in $F_i \cup A(L_i)$ that are scheduled for deletion by $L_i^{'}$. (Recall equation (1) from the previous subsection.) Since L is semi-online, we can determine the set $B(L_i^{'})$ by simply looking at the deletion-times of all the elements in $F_i \cup A(L_i)$; this is where the semi-online property of L is crucial. So, the new frozen set $F_i^{'}$ can be found by making a single pass through $F_i \cup A(L_i)$. Inductively, assume that the invariant (*) holds for both F_i and $A(L_i)$. Then, in the new set $F_i^{'}$, every element x has the correct set of values $f(x, F_j)$ for $j < i$. We complete the invariant by computing $f(x, F_i^{'})$ for all $x \in F_i^{'}$, which can be done in time $O(n_i Q(n))$ by querying the data structure $D(F_i^{'})$. Finally, we also compute the values $f(x, F_i^{'})$ for all $x \in B(F_i^{'})$, at the total cost of $O(n_i Q(n))$. These values are used for updating the W_t values of frozen sets of the higher levels, which will be constructed by partitioning $B(F_i^{'})$. This completes the discussion of the case when i is the lowest level for which a block of instructions has just finished.

Next, assume that i is not the lowest level where a block of instructions is completed. We can illustrate all aspects of the update procedure but simplify the discussion by assuming that $i - 1$ is the lowest level in the hierarchy to have a block of instructions completed at time t. By the first half of the proof, after constructing the new frozen set at level $i - 1$, the algorithm propagates a set of elements B to level i; using the earlier notation, this set is $B(F_{i-1}^{'})$. Let L_i be the first subblock of instructions to be carried out at the level i after time t. Then the frozen set of level i consists of precisely those elements of B that are *not scheduled* for deletion by L_i. Thus, the new frozen set F_i can be determined in time $O(n_i)$, again using the semi-online property of L. To compute the new value of $W_t(F_i)$, observe that we know the set of values $f(x, F_j)$ for all $x \in B$ and $j < i$, implying that the set of values $f(x, F_j)$ is correctly known for all $x \in F_i$ and $j < i$. We complete the invariant by computing $f(x, F_i)$ for all $x \in F_i$ in time $O(n_i Q(n))$. The elements of B not placed in F_i are propagated upward to level $i + 1$, where a similar construction takes place. This completes the discussion of the case when i is not the lowest level to have a block of instructions completed at time t.

The total time spent in either case is $O(P(n_i) + n_i Q(n))$. This completes the proof of the proposition. ▌

We are now ready to prove the time bound stated in Theorem 1. Let T_i denote the time needed to process a block of instructions L_i at level i. To simplify our analysis, we assume that functions $P(n)$ and $Q(n)$ have been suitably scaled to incorporate the constant factors absorbed by the "big Oh" notation above. Recalling that $|L_k| = n_{k+1}$, Proposition 3 gives

$$T_k \leq n_{k+1}(k+1)[P(n_{k+1}) + n_{k+1}Q(n)]. \tag{3}$$

The bounds for T_i, $i = k-1, \ldots, 2, 1$, are given by Proposition 4:

$$\begin{array}{rcl} T_{k-1} & \leq & (P(n_k) + n_k Q(n) + T_k)\alpha_k \\ T_{k-2} & \leq & (P(n_{k-1}) + n_{k-1} Q(n) + T_{k-1})\alpha_{k-1} \\ \vdots & & \\ T_1 & \leq & (P(n_2) + n_2 Q(n) + T_2)\alpha_2 \\ T_0 & \leq & (P(n_1) + n_1 Q(n) + T_1)\alpha_1 \end{array} \tag{4}$$

(See the right half of Figure 1.) The expression for T_{i-1} has the following components: the term $P(n_i) + n_i Q(n)$ represents the cost of updating $D(F_i)$ and $W_t(F_i)$; T_i represents the cost of processing a block of instructions at the level i; and the final term α_i is the number of subblocks L_i into which a block of level $i-1$ is split. The final time complexity of our algorithm for processing the sequence L is expressed by T_0.

Proposition 5 *The set of inequalities (4) admits the solution*

$$T_0 = O(P(n)\log n \; + \; nQ(n)\log n).$$

Proof. We proceed from top to bottom, balancing the terms $P(n_i) + n_iQ(n)$ and T_i at each step to find optimal values of α_i's and T_i's. The choice of

$$\alpha_i \;=\; (n_i)^{1/(k-i+2)} \tag{5}$$

enables us to prove by induction that the following is an admissible solution to the set of inequalities (4):

$$T_i \;\leq\; (2k-i+1)\,(P(n_{i+1}) + n_{i+1}Q(n))\,(n_{i+1})^{1/(k-i+1)}$$

We notice that at $i = k$, this agrees with equation 3, establishing the basis case of the induction. Next, we observe that

$$\alpha_i P(n_{i+1}) \leq P(n_i), \tag{6}$$

which follows from

$$\alpha_i n_{i+1} = n_i \tag{7}$$

and the assumption that $P(n)$ grows at least linearly with n. For the inductive step, we verify that if the solution holds for T_i then it also holds for T_{i-1}.

$$\begin{aligned}
T_{i-1} &\leq (P(n_i) + n_iQ(n) + T_i)\,\alpha_i \\
&= \alpha_i\,(P(n_i) + n_iQ(n)) + \alpha_i T_i \\
&= \alpha_i\,(P(n_i) + n_iQ(n)) \;+\; \alpha_i(2k-i+1)\,(P(n_{i+1}) + n_{i+1}Q(n))\,(n_{i+1})^{1/(k-i+1)} \\
&\leq \alpha_i\,(P(n_i) + n_iQ(n)) \;+\; \alpha_i(2k-i+1)\left(\frac{P(n_i)}{\alpha_i} + \frac{n_iQ(n)}{\alpha_i}\right)\left(\frac{n_i}{\alpha_i}\right)^{1/(k-i+1)} && [(6,\,7)] \\
&= \left(\alpha_i \;+\; \frac{\alpha_i(2k-i+1)(n_i)^{1/(k-i+1)}}{(\alpha_i)^{1+1/(k-i+1)}}\right)(P(n_i) + n_iQ(n)) \\
&= \left(\alpha_i \;+\; \frac{\alpha_i(2k-i+1)(n_i)^{1/(k-i+1)}}{(\alpha_i)^{(k-i+2)/(k-i+1)}}\right)(P(n_i) + n_iQ(n)) \\
&= (\alpha_i \;+\; \alpha_i(2k-i+1))\,(P(n_i) + n_iQ(n)) && [(5)] \\
&= (2k-(i-1)+1)\,(P(n_i) + n_iQ(n))\,(n_i)^{1/(k-(i-1)+1)} && [(5)]
\end{aligned}$$

This completes the induction step. The time complexity of the algorithm follows by setting $k = (\log n - 1)$ in the expression for T_0:

$$T_0 \;=\; O(P(n)\log n + nQ(n)\log n)$$

This completes the proof of the proposition. ▮

This also completes the proof of Theorem 1. Observe that our proof sets $\alpha_i = 2$, for $i = 1, 2, \ldots, k$, thus demonstrating that maintaining the frozen structures in a binary hierarchy yields the best time bound. In the next section, we discuss applications of this theorem.

3 Applications of the Main Theorem

In this section, we explore applications of Theorem 1 to a variety of problems that involve computing the extremum value of a decomposable bivariate functions. The underlying universe is a collection of primitive geometric objects such as points, lines, rectangles, hyperplanes, spheres, and the function f is some elementary measure such as distance, area or perimeter. We apply Theorem 1 to compute maximal or minimal histories of these functions with respect to a semi-online sequence, thus obtaining efficient dynamic algorithms for various problems. In all cases, our method gives a substantial improvement over any previous known results.

3.1 Diameter

Let $S \subset E^d$ be a set of n points in d dimensions and let $f(x, y)$ be the L_p distance ($p \geq 1$) between the points x and y. The *diameter* of S is defined as

$$Diameter(S) = \max_{x,y \in S} f(x, y).$$

Our result is a sublinear-time algorithm for updating the diameter of a point-set in any dimension. (Since it is easy to obtain a constant or logarithmic time bound for one dimension, we concentrate on higher dimensions; also see Final Remarks.) The distance function f is clearly decomposable with respect to the maximum. The (P, Q)-computability of f is established by the following proposition.

Proposition 6 *With respect to the maximum, the distance function f is*

1. *$(n \log n,\ \log n)$-computable in dimension $d = 2$,*
2. *$(n^{3/2} \log n,\ n^{1/2} \log n)$-computable in dimension $d = 3$, and*
3. *$(n^{2-\beta_d},\ n^{1-\beta_d} \log n)$-computable in dimensions $d \geq 4$, where $\beta_d = 1/(d(d+3)+4)$.*

Proof. Our data structure consists of the furthest-point Voronoi diagram of S, preprocessed to allow efficient point location. The maximum distance between a query point q and the set S is computed by determining which cell of the Voronoi diagram of S contains q. If the cell belongs to the point $s \in S$, then $f(q, S) = f(q, s)$. In two dimensions, the furthest-point Voronoi diagram of n points can be computed and preprocessed in $O(n \log n)$ time, following which a point-location query can be answered in time $O(\log n)$ (see e.g. Edelsbrunner, Guibas and Stolfi [11] and Lee [14]). This proves the proposition for $d = 2$.

In three dimensions, the furthest-point Voronoi diagram of n points can be constructed and preprocessed in $O(n^2)$ time, after which point-location queries can be answered in time

$O(\log^2 n)$ (see e.g. Chazelle [4], Edelsbrunner [10] and Seidel [19]). We balance the cost of constructing a Voronoi diagram against the cost of searching it by partitioning S into $\sqrt{n}/\log n$ subsets of size $\sqrt{n}\log n$ each. We construct the Voronoi diagram of each of the subsets and preprocess it for point-location. The total cost of preprocessing is

$$O(\frac{\sqrt{n}}{\log n}(\sqrt{n}\log n)^2) \;=\; O(n^{3/2}\log n).$$

A point-location query is answered in $O(n^{1/2}\log n)$ time by searching each of the $\sqrt{n}/\log n$ Voronoi diagrams, at the cost of $O(\log^2 n)$ each. This proves the proposition for $d = 3$.

In dimensions $d \geq 4$, the best bounds for constructing and searching Voronoi diagrams are given by Chazelle and Friedman [6]. They show that n hyperplanes in d dimensions can be preprocessed in $O(n^{d(d+3)/2+2}/\log n)$ time, following which a point-location query can be answered in $O(\log n)$ time. To obtain the bounds stated in the proposition, we balance the preprocessing and query costs, by splitting S into $n^{1-\beta_d}$ subsets of size n^{β_d} each. This completes the proof. ▮

Theorem 1 and Proposition 6 lead to the following result.

Theorem 2 *The history of the diameter in d dimensions can be computed at an amortized cost of $C(n)$, where*

$$\begin{aligned} C(n) &= O(\log^2 n) \quad \textit{for dimension} \quad d = 2, \\ C(n) &= O(\sqrt{n}\log^2 n) \quad \textit{for dimension} \quad d = 3, \\ C(n) &= O(n^{1-\beta_d}\log^2 n) \quad \textit{for dimensions } d \geq 4, \quad \textit{where} \quad \beta_d = \frac{1}{d(d+3)+4}. \end{aligned}$$

To contrast Theorem 2 with previous results, we note that the online version of the two-dimensional case was considered by Overmars [16]), and he obtained $O(n)$ bound on the update time per insert or delete. A comparison of our bounds with the best static-case time bounds is presented in the table of results given in the introduction. Indeed, our bound for computing the *entire history* of the diameter during a sequence of n inserts and deletes is quite close to the best upper bounds known for solving a *single* instance of the problem by a static algorithm.

3.2 Closest Pair

Let $S \subset E^d$ be a set of n points in d dimensions and let $f(x, y)$ be the L_p distance ($p \geq 1$) between the points x and y. The *closest pair* of S is a pair of points $s_1, s_2 \in S$ such that

$$f(s_1, s_2) \;=\; \min_{x,y \in S} \{f(x, y) \mid x \neq y\}.$$

The closest pair of n points in E^d can be computed in time $O(n \log n)$ for any fixed dimension d (Bentley and Shamos [3]), which is optimal in the worst case. We present algorithms for maintaining the closest pair in sublinear time in all dimensions. The function f is clearly decomposable with respect to the minimum. Its (P, Q)-computability is established by the following proposition.

Proposition 7 *With respect to the minimum, the distance function f is*

1. $(n \log n,\ \log n)$*-computable in dimension* $d = 2$,
2. $(n^{3/2} \log n,\ n^{1/2} \log n)$*-computable in dimension* $d = 3$, *and*
3. $(n^{2-\beta_d},\ n^{1-\beta_d} \log n)$*-computable in dimensions* $d \geq 4$, *where* $\beta_d = 1/(d(d+3)+4)$.

Proof. The proof is identical to the proof of Proposition 6, except that we use the *closest-point Voronoi* diagram instead of the furthest-point Voronoi diagram. ▮

Theorem 1 and Proposition 7 yield the following result.

Theorem 3 *The history of the closest pair in d dimensions can be computed at an amortized cost of $C(n)$, where*

$$\begin{aligned} C(n) &= O(\log^2 n) \quad \textit{for dimension} \quad d = 2, \\ C(n) &= O(n^{1/2} \log^2 n) \quad \textit{for dimension} \quad d = 3, \\ C(n) &= O(n^{1-\beta_d} \log^2 n) \quad \textit{for dimensions } d \geq 4, \quad \textit{where} \quad \beta_d = \frac{1}{d(d+3)+4}. \end{aligned}$$

3.3 Minimum Separation of Rectangles

Let S be a set of n disjoint axes-parallel rectangles in the plane. Given a pair of rectangles $s = [x_1, x_2] \times [y_1, y_2]$ and $s' = [x_1', x_2'] \times [y_1', y_2']$, the *separation* of s and s' is defined as the minimum y-distance between them if $[x_1, x_2] \cap [x_1', x_2'] \neq \emptyset$, and as the minimum x-distance if $[y_1, y_2] \cap [y_1', y_2'] \neq \emptyset$; the separation is assumed to be infinite if the rectangles do not overlap on either coordinate.[2] Let $f(s, s')$ denote the separation of the rectangles s and s'. The minimum separation of S is defined as

$$MinSeparation(S) \;=\; \min_{s,s' \in S} \{f(s,s') \mid s \neq s'\}$$

Minimum separation problem arises in VLSI layout design, where masks used in the fabrication of integrated circuits are frequently expressed as a collection of axes-parallel rectangles. The separation function f is decomposable with respect to the minimum, and its (P, Q)-computability is proved by the following proposition.

Proposition 8 *The separation function f is $(n \log n, \log n)$-computable with respect to the minimum.*

Proof. We show how to preprocess a set S of n rectangles in time $O(n \log n)$ so that the minimum *vertical* separation between a query rectangle q and any rectangle of S can be computed in $O(\log n)$ time; the minimum horizontal separation is computed by a similar data structure.

[2]The assumption of infinite separation in the case of non-overlapping projections is not limiting. By observing that the minimum distance between two non-overlapping rectangles is achieved between two vertices, it is easy to extend our results to the case where the separation is defined as the minimum distance between any two points of the rectangles.

We construct a balanced binary tree H whose leaves represent the $2n$ abscissas of the rectangles of S, sorted in the nondecreasing order. For a nonleaf node w, let $S(w)$ denote the set of rectangles whose abscissas are represented by the descendant leaves of w. We store with w a sorted list of the ordinates of the rectangles of $S(w)$. Let x_l and x_r denote the left and right abscissas of the query rectangle q, and let y_t and y_b denote the top and bottom ordinates of q. There is a canonical decomposition of H into $O(\log n)$ subtrees rooted at, say, $w_1, w_2, \ldots, w_m$ such that $\bigcup_{i=1}^{i=m} S(w_i)$ is precisely the set of rectangles whose horizontal-spans have a nonempty intersection with $[x_l, x_r]$. The rectangle with a minimum y-separation from q is found by searching the lists stored at w_i's by the ordinate-keys of q, namely y_t and y_b.

The tree H and the sorted lists stored with its nodes can be constructed easily in $O(n \log n)$ time. The canonical decomposition of H with respect to q is found in $O(\log n)$ time by tracing the paths from the root to the leaves with the maximum x-coordinate less than x_l and the minimum x-coordinate greater than x_r. A naive method of searching the $O(\log n)$ sorted lists stored with w_i's takes $O(\log^2 n)$. We can save a factor of $\log n$ by making use of the technique of fractional cascading. This allows each of these binary searches to be performed in constant time, after the first binary search has been conducted at the full cost of $O(\log n)$ (see Chazelle and Guibas [7] for details). This completes the proof. ▮

Theorem 1 and Proposition 8 imply the following result.

Theorem 4 *The history of the minimum separation of a set of disjoint axes-parallel rectangles can be computed at an amortized cost of $O(\log^2 n)$.*

3.4 Points and Hyperplanes

We consider the following problem, which is a generalization of a problem posed by Hopcroft in the context of collision avoidance. Given a set of points S and a set of hyperplanes T in E^d, determine the minimum distance between a point of S and a hyperplane of T. In Hopcroft's original problem, we only need to detect if there is any incidence between the elements of S and T (the case of minimum distance being zero). The current best solution for the latter in E^2 is an $O(n^{1.41})$ time algorithm due to Cole, Sharir and Yap [8]. We develop dynamic algorithms of sublinear time complexity for the generalized problem in all dimensions.

Let $f : (S \cup T) \times (S \cup T) \to R^+$ be the distance function between points and hyperplanes, defined as follows. $f(x, y)$ is the distance between elements x and y if x is a point and y a hyperplane, or vice versa. Otherwise (when x and y are either both points or hyperplanes), $f(x, y) = +\infty$. It is easily seen that f is symmetric and decomposable with respect to the minimum. The (P, Q)-computability of f is established by the following proposition.

Proposition 9 *With respect to the minimum, the point-hyperplane distance function f is*

1. *$(n^{3/2} \log^{1/2} n,\ n^{1/2} \log^{1/2} n)$-computable in dimension $d = 2$,*
2. *$(n^{5/3} \log^{4/3} n,\ n^{2/3} \log^{4/3} n)$-computable in dimension $d = 3$,*
3. *$(n^{2-2\beta_d},\ n^{1-2\beta_d} \log n)$-computable in dimensions $d \geq 4$, where $\beta_d = 1/(d(d+3)+4)$.*

Proof. We show that given a set S of n points (resp. hyperplanes) in E^d, we can compute a data structure in time $P(n)$ so that given a query hyperplane (resp. point) q, the minimum distance between q and the set S can be determined in time $Q(n)$, where $P(n)$ and $Q(n)$ are as stated in the proposition.[3] The two problems, namely computing the point closest to a query hyperplane and computing the hyperplane closest to a query point, are equivalent under the standard point-hyperplane duality (see e.g. Chazelle [4]). Thus, we only need to describe our data structure for the "primal" case: a set of points and the hyperplane queries. We use known results on neighbor-searching in E^d.

In two dimensions, we use the following result obtained independently by Cole and Yap [9] and Lee and Ching [15]: A set S of n points in E^2 can be preprocessed in time $O(n^2)$ so that, given a query line ℓ, the point of S closest to ℓ can be determined in time $O(\log n)$. We balance the preprocessing and query costs by partitioning S into $\sqrt{n/\log n}$ subsets of $\sqrt{n\log n}$ points each. We then preprocess each of the subsets to construct the data structure of [9] or [15]. The total cost of preprocessing is

$$O\Big(\sqrt{\frac{n}{\log n}}\,(\sqrt{n\log n})^2\Big) \;=\; O(n^{3/2}\log^{1/2} n).$$

To answer the query "which point of S is closest to the line ℓ?," we search the $\sqrt{n/\log n}$ structures, at the cost of $O(\log n)$ each. This proves the proposition for the dimension $d = 2$.

In three dimensions, we use a similar result of Chazelle [4]: A set S of n points in E^3 can be preprocessed in time $O(n^3)$ so that, given a plane h, the point of S closest to h can be determined in time $O(\log^2 n)$. To obtain our bounds, we partition the set S into $(n/\log n)^{2/3}$ subsets of size $n^{1/3}\log^{2/3} n$ each. For dimensions $d \geq 4$, the data structure of Chazelle and Friedman [6] achieves the preprocessing and query costs $O(n^{d(d+3)/2+2})$ and $O(\log n)$, respectively. To obtain the bounds stated in the proposition, we balance the preprocessing and query costs by splitting S into $n^{1-2\beta_d}$ subsets of size $n^{2\beta_d}$ each. This completes the proof. ∎

Theorem 1 and Proposition 9 lead to the following result.

Theorem 5 *The history of the minimum distance between a set of points and a set of hyperplanes in d dimensions can be computed at an amortized cost of $C(n)$, where*

$$\begin{aligned}
C(n) &= O(n^{1/2}\log^{3/2} n) && \textit{for dimension}\;\; d = 2,\\
C(n) &= O(n^{2/3}\log^{7/3} n) && \textit{for dimension}\;\; d = 3,\\
C(n) &= O(n^{1-2\beta_d}\log^2 n) && \textit{for dimensions}\; d \geq 4, \quad \textit{where}\;\; \beta_d = \frac{1}{d(d+3)+4}.
\end{aligned}$$

Results similar to Theorem 5 are also possible for computing the minimum distance between points and spheres in two and three dimensions.

[3] Recall that in our algorithm we use this data structure to perform queries on the frozen sets. In the present situation, a frozen set will maintain two different data structures: one for the points and one for the hyperplanes.

3.5 Largest and Smallest Rectangles

Given a set of points S in the plane, we consider the problem of maintaining the largest (or smallest) axes-parallel rectangle whose opposite corners lie at points of S. Every pair of points in S determines a unique rectangle of this kind. Let $f(x, y)$ denote the area of the rectangle determined by the points $x, y \in S$. The function f is decomposable with respect to the maximum (as well as the minimum). In order to obtain a data structure with required (P, Q)-computability features, we utilize a result of Chazelle, Drysdale and Lee [5], concerning the LL-diagram of a set of points.

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the plane. Let x_{min} and y_{min} denote the minimum x- and y-coordinate among the points of S. Let

$$Z(S) \ = \ (-\infty, x_{min}] \times (-\infty, y_{min}].$$

The LL-diagram of S is a partition of $Z(S)$ into a set of regions $\{V(p_1), V(p_2), \ldots, V(p_n)\}$, where

$$V(p_i) \ = \ \{p \in Z(S) \mid f(p, p_i) \geq f(p, p_j) \ \text{ for all } \ j = 1, 2, \ldots, n\}.$$

(The LL-diagram in the original paper of Chazelle et al. [5] is defined for a region somewhat larger than $Z(S)$; however, we find the above definition more convenient for our application.) Observe that only the maximal points of S can have nonempty regions in the LL-diagram of S; recall that a point p is maximal if no other point in the set dominates p simultaneously on both the coordinates. Chazelle et al. [5] prove the following result.

Proposition 10 ([5]) *The LL-diagram of a set of n points can be computed in $O(n \log n)$ time.*

We introduce some notation. The *x-ordered tree* of S, denoted $X(S)$, is the complete binary tree whose leaves are the points of S sorted by their x-coordinates. The y-ordered tree, denoted $Y(S)$, is defined similarly. Given a node $w \in X(s)$, $S(w)$ denotes the set of points associated with the leaves of the subtree rooted at w, and $M(w)$ denotes the set of maximal points of $S(w)$. We find it convenient to assume that the elements of $M(w)$ are listed by increasing x-coordinates, and thus decreasing y-coordinates; consequently, we also refer to $M(w)$ as a *maximal chain.*

Given a query point $q = (x(q), y(q))$, the *maximal chain of S with respect to q*, denoted $M(q, S)$, is the set of maximal points of S in the region $[x(q), +\infty) \times [y(q), +\infty)$. Finally, we use $N[x(q), X(S)]$ to denote the *canonical representation* of the points of S having x-coordinates greater than $x(q)$:

$$N[x(q), X(S)] = \{w_1, w_2, \ldots, w_m\}, \ \ m \leq \lceil \log n \rceil,$$

where w_i's are nodes of $X(S)$ such that $\bigcup_{i=1}^{i=m} S(w_i)$ is precisely the set of points whose x-coordinate is greater than $x(q)$. We can obtain the canonical representation by removing from $X(S)$ the edges of the path between the root and the leaf containing the maximum x-coordinate smaller than $x(q)$. We are now ready to prove the following proposition.

Proposition 11 *The rectangular area function f is $(n \log^2 n, \log^3 n)$-computable with respect to the maximum.*

Proof. We describe a data structure for computing the largest area rectangle determined by the query point q and the points of S, where q forms the *lower-left* corner of the rectangle. The other three cases are treated similarly. Our data structure has three levels:

1. The top level of the data structure consists of the x-ordered tree $X(S)$.
2. The second level contains, for each node $w \in X(S)$, the y-ordered tree $Y(M(w))$ storing the maximal chain $M(w)$.
3. The third level contains, for each nonleaf node $v \in Y(M(w))$, the maximal chain of the set of points associated with the leaves of the subtree rooted at v. The LL-diagram of the set of points in this maximal chain is also stored with v.

Given a query point $q = (x(q), y(q))$, we assemble the maximal chain $M(q, S)$ by concatenating $O(\log^2 n)$ precomputed chains that are stored with the nodes of our data structure. The corresponding LL-diagrams are used to compute the largest area rectangle. The details are as follows.

We start by computing the canonical representation $N[x(q), X(S)] = \{w_1, w_2, \ldots, w_m\}$, where we assume that the descendants of w_i have x-coordinates smaller than those of w_j for $i < j$. We process w_i's in the decreasing order of index, and compute two numbers l_i and h_i for $i = m, m-1, \ldots, 1$, as follows:

$$
\begin{aligned}
l_{m+1} &:= y(q), \qquad h_{m+1} := y(q), \\
l_i &:= \text{the minimum } y\text{-coordinate in } M(w_i) \text{ greater than } h_{i+1}, \\
h_i &:= \text{the maximum } y\text{-coordinate in } M(w_i) \text{ greater than } h_{i+1}.
\end{aligned}
$$

(If l_i and h_i do not exist, we artificially set them to l_{i+1} and h_{i+1}.) Observe that $M(q, S)$ is the union over all i of $M(w_i)$ restricted to the range $[l_i, h_i]$. We now use the y-ordered trees of our data structure to retrieve $M(w_i)$ restricted to $[l_i, h_i]$. Let

$$N[l_i, Y(M(w_i))] = \{w_{i1}, w_{i2}, \ldots, w_{ik_i}\}$$

be the canonical representation of points of $M(w_i)$ having y-coordinates greater than l_i. Then, the chain $M(w_i)$ restricted to $[l_i, h_i]$ is simply the concatenation of the maximal chains $M(w_{ij})$ stored with w_{ij}'s for $j = 1, 2, \ldots, k_i$. The LL-diagrams of these chains are available by precomputation.

Thus, the maximal chain $M(q, S)$ is obtained as the union of $O(\log^2 n)$ precomputed subchains $M(w_{ij})$. The largest rectangle formed by q and some point of $M(w_{ij})$, with q at the lower-left corner, is found by determining the region of the LL-diagram of $M(w_{ij})$ containing q. The final answer is obtained by choosing the maximum over all $M(w_{ij})$.

The total cost of answering the query is $O(\log^3 n)$: assembling the maximal chain $M(q, S)$ takes $O(\log^2 n)$ time, $O(\log n)$ time per node w_i; and locating the query point q in the $O(\log^2 n)$ LL-diagrams takes $O(\log^3 n)$ time, $O(\log n)$ per diagram. To estimate the preprocessing cost, observe that, for a node $w \in X(S)$, computing $M(w)$ and the associated LL-diagram takes $O(m \log m)$ time, where $m = |S(w)|$. By building the LL-diagram from bottom to top, the LL-diagrams for all descendants of w in the tree $Y(M(w))$ can also be computed within the same time bound. Since each point $s \in S$ belongs to $O(\log n)$

subsets $S(w)$, the overall cost of building the data structure is $O(n \log^2 n)$. This completes the proof. ▮

Theorem 1 and Proposition 11 lead to the following theorem.

Theorem 6 *The history of the largest area rectangle in a two-dimensional set of points can be computed at an amortized cost of* $O(\log^4 n)$.

Theorem 6 also holds for the minimum area rectangles. For the perimeter function, observe that the perimeter of the largest (resp. smallest) rectangle determined by points of S is twice the L_1 diameter (resp. the L_1 distance between the closest pair) of S. Thus, extremal rectangles under the perimeter measure can be computed at an amortized cost of $O(\log^2 n)$ by using Theorems 2 and 3.

4 Extensions

The applications cited in Section 3 illustrate the wide applicability of Theorem 1. Indeed, other applications are possible; for instance, we can replace points, lines and rectangles with more complicated objects such as convex polygons with a fixed number of sides. Rather than discuss more of these applications, we consider generalizations of our technique to other kinds of problems. In particular, we consider two extensions of Theorem 1. The first utilizes the hierarchy of Section 2 to obtain a uniform result for decomposable search problems. The second applies the technique to a non-decomposable problem.

4.1 Decomposable Search Problems

A search problem consists of a query function f, a query element q, and a set S. The answer to the query is denoted by $f(q,S)$. The query function f is called *decomposable* if, for any query element q and decomposition $S = S_1, S_2, \ldots, S_k$,

$$f(q,S) \;=\; \Box\{f(x,S_1), f(x,S_2), \ldots, f(q,S_k)\},$$

where $\Box$ is an elementary operator such as *min*, *max* or $+$. The function f is called (P,Q)*-computable* if, for all $S' \subseteq S$, there is a $P(|S'|)$ time computable data structure that computes $f(q,S')$ in time $Q(|S'|)$ for all q.

We want to compute f during a semi-online sequence of insertions and deletions. The query at time t is to be answered with respect to the set of elements active at time t, S_t; we assume that the set is initially empty. We prove the following theorem.

Theorem 7 *Let* f *be a decomposable query function that is* (P,Q)*-computable. We can process a semi-online sequence of* n *inserts and deletes at an amortized cost of*

$$O\left(\frac{P(n)\log n}{n}\right)$$

such that, at any time t*, a query* $f(q,S_t)$ *can be answered in time* $O(Q(n)\log n)$.

Proof. We process the sequence L by constructing the same hierarchy as that used in proving Theorem 1. The present problem is simpler in that we are merely concerned with maintaining the frozen sets at all levels; no function value needs to be computed after each insertion or deletion. The cost per insertion and deletion, therefore, follows immediately from Theorem 1, except now there is no $Q(n)\log n$ term. Recall that the hierarchy uses $\log n$ levels, with one frozen set F_i per level such that, at any time t,

$$S_t \;=\; F_1 \cup F_2 \cup \cdots \cup F_{\log n}.$$

Since f is decomposable, we can compute $f(q, S)$ by searching all the frozen sets, at the cost of $Q(n)$ each. This establishes the bound for answering a query, and completes the proof. ▮

We illustrate Theorem 7 by giving two examples: the Post-Office problem and the Extremal Query problem.

Post-office Problem: Preprocess a set of points S in E^d so that, given a query point q, the point in S closest to q can be determined efficiently.

Extremal Query: Preprocess a set of points S in E^d so that, given a hyperplane h that is free to rotate about a $(d-2)$-dimensional flat contained in h, the point of S first hit by h can be determined efficiently. (All points of S are initially assumed to lie on one side of h.)

Theorem 7 allows these problems to be solved dynamically. The data structures employed are the same as those used to solve the static versions of these problems. In particular, for the Post-Office problem, we use the Voronoi diagram, preprocessed to allow efficient point-location. The time bounds are $P(n) = O(n\log n)$, $Q(n) = O(\log n)$ in two dimensions and $P(n) = O(n^{3/2}\log n)$, $Q(n) = O(n^{1/2}\log n)$ in three dimensions (cf. Subsection 3.2). For the Extremal Query problem, we use the convex hull of S. Given a set S of n points in two or three dimensions, the convex hull of S can be preprocessed in time $O(n\log n)$, following which an extremal query can be answered in time $O(\log n)$ (see e.g. Edelsbrunner [10]). Thus, Theorem 7 leads to the following results.

Theorem 8 *We can process a semi-online sequence of n inserts and deletes, and answer Post-Office queries, at an amortized cost of $O(\log^2 n)$ per operation in two dimensions, and $O(n^{1/2}\log^2 n)$ per operation in three dimensions.*

Theorem 9 *We can process a semi-online sequence of n inserts and deletes, and answer Extremal queries, at an amortized cost of $O(\log^2 n)$ per operation in dimensions $d \leq 3$.*

In the three-dimensional Post-Office problem, we can alternatively obtain the query time of $O(\log^3 n)$, at the expense of making insertions and deletions cost $O(n\log n)$. Extensions of Theorems 8 and 9 to higher dimensions are possible along the lines of Theorems 2 and 3.

Although similar bounds have been obtained by others, our method has a few key advantages. First, Theorem 7 offers a uniform bound for decomposable search problems with deletions. Second, our method works for semi-online sequences rather than offline. Third, our method may be easier to implement than, for instance, the method of Edelsbrunner and Overmars [12], which uses a more complicated segment-tree based approach.

4.2 Non-decomposable Problems

We consider the problem of computing the largest empty rectangle determined by a set of points. The problem is as follows.

> *Largest Empty Rectangle*: Given a set of points S in the plane, compute the largest axes-parallel rectangle whose two opposite corners are at points of S and that does not contain any point of S in its interior.

Observe that Theorem 6 is not applicable here because the largest empty rectangle problem is not decomposable. In particular, if $f(x, y)$ denotes the area of the rectangle determined by the points $x, y \in S$, then we are not interested in just computing the $max\ f(x, y)$, rather the maximum among those rectangles that do not contain any other point of S. To this end, let $f(q, S)$ denote the largest empty rectangle determined by q and some point of S. Then, even for a decomposition into two sets, $S = S_1 \cup S_2$, the equality $f(q, S) = max\{f(q, S_1), f(q, S_2)\}$ does not always hold. This is so because a rectangle determined by q and S_1 may very well contain points of S_2. Nevertheless, we show in the following that by using a shallower version of our hierarchy, we can compute the history of the largest empty rectangle at a sublinear amortized cost. In the static case, the best algorithm for computing the largest empty rectangle takes $O(n \log^2 n)$ time (see Aggarwal and Suri [1]).

Given a point $q \in Z(S)$, the rectangle formed by q and $p \in S$ is empty if and only if p does not dominate any other point of S; recall from Subsection 3.5 that $Z(S)$ denotes the region $(-\infty, x_{min}] \times (-\infty, y_{min}]$. Therefore, in the following, we focus our attention on the minimal points of S; a point $p \in S$ is called *minimal* if p does not dominate any other point of S. The following proposition states the analogue of Proposition 11 for the largest empty rectangle.

Proposition 12 *Let S be a set of n points in the plane. We can compute a data structure in time $O(n \log^2 n)$ so that, given a query point q, determining the largest empty rectangle formed by q and some point of S takes $O(\log^3 n)$ time.*

Proof. Our data structure is identical to the one used in Proposition 11, except that instead of storing maximal chains and their LL-diagrams, we store minimal chains and their LL-diagrams.Given a query point $q = (x(q), y(q))$, we assemble $M(q, S)$, the minimal chain of S with respect to q, as the union of $O(\log^2 n)$ precomputed chains. The largest empty rectangle formed by q and some point of S is determined by querying $O(\log^2 n)$ precomputed LL-diagrams, each at the cost of $O(\log n)$. The details are identical to those in the proof of Proposition 11 and we shall not repeat them here. ▮

Although f is not decomposable, Proposition 12 can be used to compute f over two sets, as follows.

Proposition 13 *Let S_1 and S_2 be sets of n_1 and n_2 points, respectively, where $n_1 \leq n_2$. After $O(n_2 \log^2 n_2)$ time preprocessing, we can compute $f(q, S_1 \cup S_2)$, area of the largest rectangle formed by a query point q and some point of $S_1 \cup S_2$, in time $O(n_1 \log^3 n_2)$.*

Proof. We store S_2 into the data structure of Proposition 12; this takes $O(n_2 \log^2 n_2)$ time. Given a query point $q = (x(q), y(q))$, we compute $M(q, S_1)$, the minimal chain of S_1 with respect to q, and the LL-diagram of $M(q, S_1)$; this takes $O(n_1 \log n_1)$ time. Next, we compute $N[x(q), X(S_2)]$, the canonical representation of points in S_2 having x-coordinates greater than $x(q)$. Let $N[x(q), X(S_2)] = \{w_1, w_2, \ldots, w_m\}$. Recall that $M(q, S_2)$, the minimal chain of S_2 with respect to q, is a subset of $\bigcup_{i=1}^{i=m} M(w_i)$. We "superimpose" $M(q, S_1)$ onto the minimal chains $M(w_i)$, $i = 1, 2, \ldots, m$. This involves clipping portions of $M(w_i)$'s and $M(q, S_1)$ that are hidden from q by the other.

This refinement splits $\bigcup_{i=1}^{i=m} M(w_i)$ into $O(n_1 + \log n_2)$ fragments. We glue these fragments together by marching from left to right to obtain a chain that consists of precisely those points of S_2 with which q can form an empty rectangle. (In general, the set of points in this chain will be a proper subset of $M(q, S_2)$.) Since $M(w_i)$ are stored also as y-ordered trees, we can use the canonical representation by y-coordinates to represent the final chain as the union of $O(n_1 \log^2 n_2)$ precomputed minimal chains. The largest empty rectangle formed by q and some point of S_2, with q at the lower left corner, can be found by searching the LL-diagrams associated with these $O(n_1 \log^2 n_2)$ minimal chains, and choosing the maximum. The largest empty rectangle formed by q and some point of S_1 is determined by a direct computation with each of the points left in $M(q, S_1)$ after the refinement. The time for constructing and merging the minimal chains is $O(n_1 \log^2 n_2)$ and the time to do rectangle computations is $O(n_1 \log^3 n_2)$. This completes the proof. ∎

We use this proposition to prove the following result.

Theorem 10 *The history of the largest empty rectangle in a two-dimensional set of points can be computed at an amortized cost of* $O(n^{2/3} \log^3 n)$.

Proof. We build a hierarchy similar to that used in proving Theorem 1, but this time we restrict ourselves to only two levels. We set

$$\alpha_1 = n^{2/3} \quad \text{and} \quad \alpha_2 = n^{1/3},$$

which means that the sequence L is split into $n^{2/3}$ blocks of length $n^{1/3}$ each, and each block L_1 is further split into $n^{1/3}$ subblocks of length 1 each. The frozen sets at the two levels have sizes $|F_1| \leq n$ and $|F_2| \leq n^{1/3}$. Each frozen set is organized into a data structure of Proposition 12. We maintain the following invariants:

1. For each point $x \in F_2$, the value of $f(x, F_1 \cup F_2)$ is known.

2. The area of the largest empty rectangle determined by two points both in F_1 is known.

Upon the insertion of an element z, we use Proposition 13 to compute $f(z, S_t)$ in time $O(n^{1/3} \log^3 n)$, where $S_t = F_1 \cup F_2$ is the current active set. The new largest empty rectangle is the winner of the old rectangle and the rectangle achieving $f(z, F_1 \cup F_2)$. Upon the deletion of an element z, we just choose the maximum rectangle from the existing $F_1 \cup F_2$; this is available from the two invariants above. Thus, the new maximum after a deletion can be reported in just constant time if we keep track of the largest empty rectangle formed by points of $F_1 \cup F_2$.

The time for maintaining the invariants is estimated as follows. Since a block of instructions at the second level consists of just one instruction, F_2 is recomputed after every instruction. Computing the new data structure for F_2 takes $O(n^{1/3} \log^2 n)$ time, by Proposition 12, and computing $f(x, F_1 \cup F_2)$ for all $x \in F_2$ takes $O(n^{2/3} \log^3 n)$ time, by Proposition 13. Thus, the cost of maintaining the two invariants at the second level of the hierarchy is $O(n^{2/3} \log^3 n)$ per insert and delete instruction.

Updates to F_1 occur every $n^{1/3}$ instructions. To restore the invariant for F_1, we need to compute the largest empty rectangle determined by two points both in F_1. This can be accomplished in $O(n \log^3 n)$ time by an easy modification of the static-case algorithm of Chazelle, Drysdale and Lee [5]. Thus, the amortized cost of maintaining the invariant at the first level also is $O(n^{2/3} \log^3 n)$. This proves the theorem. ▮

5 Final Remarks and Open Problems

We have presented an efficient method for updating the extremal value of a decomposable function after the insertion or deletion of an element. The various applications cited in the paper demonstrate the usefulness of our result. The main appeal of our approach lies in its generality, and we expect to see further applications and extensions of our technique. Our work continues on generalizations of Theorem 1. In particular, we are pursuing the following research directions that arise from relaxing the restrictions of the current model.

First, instead of bivariate functions, we may consider multivariate functions, which correspond to geometric objects determined by multiple points. Let $f(x_1, x_2, \ldots, x_k)$ denote the area (perimeter) of the polygon whose vertices are $x_1, x_2, \ldots, x_k$. Then maintaining the largest or smallest k-gon in a dynamic set of points is equivalent to computing the extremal history of f during a sequence of inserts and deletes. The problem appears to be difficult even for small values of k, say, $k = 3$. Questions can also be asked about the maintenance of other structures such as the minimum spanning circle or the largest empty inscribed circle; these also are ordinarily determined by three points.

Second, we may consider histories of more complex predicates than just *max* or *min*. For instance, consider the *discrete* minimum spanning circle problem. This problem asks for the smallest circle centered at one of the data points that contains all other points in its interior. Determining such a circle reduces to the computation of $min_x max_y\ f(x, y)$.

Our results clearly hold for the offline model, where the entire sequence of inserts and deletes is known in advance. In fact, we conjecture that in the offline model the amortized time complexity of Theorem 1 can be made worst-case. The semi-online model appears to be strictly more powerful than online, even in one dimension. Specifically, given a sequence of n inserts and deletes, we can show that computing the history of diameter in one dimension requires $\Omega(n \log n)$ time for the online model, but in the semi-online model it can be computed in $O(n)$ time. To what extent can our results be extended to the online model remains the foremost open problem.

Acknowledgment

Research of the first author was supported in part by NSF Grant Number CCR87-00917 and a Guggenheim Fellowship. We thank an anonymous referee for helpful suggestions.

References

[1] A. Aggarwal and S. Suri, Fast algorithms for computing the largest empty rectangle, (extended abstract) *Proc. of 3rd Annual Symposium on Computational Geometry*, 278-290, 1987.

[2] J. Bentley and J. Saxe, Decomposable searching problems I. Static-to-dynamic transformation, *J. of Algorithms*, 1, 301-358, 1980.

[3] J. Bentley and M. Shamos, Divide-and-conquer in multidimensional space, *Proc. of Eighth ACM Symposium on Theory of Computing*, 220-230, 1976.

[4] B. Chazelle, How to search in history, *Information and Control,* 64, 77-99, 1985.

[5] B. Chazelle, R. Drysdale and D. T. Lee, Computing the largest empty rectangle, *SIAM J. of Computing*, 15, 300-315, 1986.

[6] B. Chazelle and J. Friedman, A deterministic view of random sampling and its use in geometry, *Proc. of 29th IEEE Symposium on Foundations of Computer Science,* pp. 539-549, 1988.

[7] B. Chazelle and L. Guibas, Fractional Cascading: I and II, *Algorithmica*, 1, 133-191, 1986.

[8] R. Cole, M. Sharir and C. Yap, On k-hulls and related problems, *SIAM J. of Computing,* 16, 61-77, 1987.

[9] R. Cole and C. Yap, Geometric retrieval problems, *Information and Control,* 63, 39-57, 1984.

[10] H. Edelsbrunner, *Algorithms in Combinatorial Geometry,* Springer-Verlag, 1987.

[11] H. Edelsbrunner, L. Guibas and J. Stolfi, Optimal point location in monotone subdivision, *SIAM J. of Computing*, 15, 317-340, 1986.

[12] H. Edelsbrunner and M. Overmars, Batched dynamic solutions to decomposable searching problems, *J. of Algorithms*, 6, 515-542, 1985.

[13] H. Gabow and R. Tarjan, A linear-time algorithm for a special case of disjoint set union, *J. of Computer and System Sciences*, 30, 209-221, 1985.

[14] D. T. Lee, Two dimensional Voronoi diagram in the L_p metric, *J. ACM*, 27, 604-618, 1980.

[15] D. T. Lee and Y. Ching, The power of geometric duality revisited, *Information Processing Letters*, 21, 117-122, 1985.

[16] M. Overmars, Dynamization of order decomposable set problems, *J. of Algorithms,* 2, 245-260 1981.

[17] M. Overmars and J. van Leeuwen, Maintenance of Configurations in the Plane, *J. of Computer and System Science*, 23, 166-204, 1981.

[18] F. Preparata and M. Shamos, *Computational Geometry,* Springer Verlag, New York, NY, 1985.

[19] R. Seidel, A convex hull algorithm optimal for points in even dimensions, Report 81-14, Computer Science, University of British Columbia, Vancouver, 1981.

[20] A. C. Yao, On constructing minimum spanning trees in k-dimensional space and related problems, *SIAM J. of Computing,* 11, 721-736, 1982.