A Worst-Case Algorithm for Semi-Online Updates on Decomposable Problems *

Michiel Smid

A 03/90

FB Informatik, Universität des Saarlandes, 6600 Saarbrücken, Federal Republic of Germany

Abstract: An alternative description, and a more straightforward analysis, of Dobkin and Suri's algorithm to maintain the maximum of a symmetric bivariate function is given. Furthermore, the amortized update time of this algorithm is made worst-case. As an application of the resulting algorithm, it follows e.g. that the diameter (or closest pair) of a planar point set can be maintained under semi-online updates in $O((\log n)^2)$ time per update in the worst case. A simplified version of the algorithm is applied to obtain a data structure for decomposable searching (resp. set) problems, in which semi-online updates can be performed efficiently, even in the worst case. The amortized version of this algorithm is an extension of Bentley's logarithmic method.

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

A Worst-Case Algorithm for Semi-Online Updates on Decomposable Problems

Michiel Smid
Fachbereich 10 - Informatik
Universität des Saarlandes
D-6600 Saarbrücken
West-Germany

February 13, 1990

Abstract

An alternative description, and a more straightforward analysis, of Dobkin and Suri's algorithm to maintain the maximum of a symmetric bivariate function is given. Furthermore, the amortized update time of this algorithm is made worst-case. As an application of the resulting algorithm, it follows e.g. that the diameter (or closest pair) of a planar point set can be maintained under semi-online updates in $O((\log n)^2)$ time per update in the worst case. A simplified version of the algorithm is applied to obtain a data structure for decomposable searching (resp. set) problems, in which semi-online updates can be performed efficiently, even in the worst case. The amortized version of this algorithm is an extension of Bentley's logarithmic method.

1 Introduction

In the past decade, much work has been done on the design of general techniques to dynamize static data structures, that do not allow objects to be inserted or deleted efficiently, into dynamic ones, that do allow such operations. This research was initiated by the work of Bentley [2], who introduced the logarithmic method. This method transforms any static data structure that solves a decomposable searching problem, into a structure in which objects can be inserted efficiently, at the cost of a slight increase in query time. A decomposable searching problem is a problem that can be solved for a set V in constant time, if the problem is solved for sets A and B that partition V. The strength of this technique lies in the fact that it applies to any structure that solves a query problem of this type. Later, many other techniques were designed for decomposable searching problems. See Bentley and Saxe [3], Overmars [11] for an overview of these techniques.

The first of these general techniques resulted in data structures in which only insertions can be performed, and the insert times are amortized. Later, it was shown that many of these data structures can be transformed into structures in which objects can be inserted efficiently even in the worst case. See [11], where many of these worst-case algorithms are given.

For the class of decomposable searching problems, however, it is still difficult to design fully dynamic data structures, in which also objects can be deleted efficiently. For some subclasses, general techniques are available. An example of such a class are the decomposable

counting problems. (See [3]). For problems such as the nearest neighbor searching problem—which is clearly decomposable—there is, however, no data structure known, in which objects can be inserted and deleted in $o(n)$ time, and in which simultaneously queries can be solved in polylogarithmic time.

Other examples of problems, for which it is not known whether fully dynamic data structures exist in which arbitrary updates can be performed in $o(n)$ time, are the problems of maintaining the diameter and the closest pair of a planar point set.

Therefore, one possible direction is to consider restricted types of updates. For example, Edelsbrunner and Overmars [7] give general techniques to design data structures for decomposable searching problems in which all queries, insertions and deletions are known at the start of the algorithm.

Another type of updates was recently introduced by Dobkin and Suri [5]. They give a general technique to design a data structure for maintaining the maximum of a symmetric bivariate function, in which *semi-online* updates can be performed. Here, a sequence of updates is called semi-online, if the insertions are on-line—i.e., they arrive in an unknown order—but when an object is inserted, we are told how long it will stay, or that it will remain present forever. That is, with each inserted object, we are told how many updates from the moment of insertion, the object will be deleted. As an application of their method, they obtain a data structure that maintains the diameter (or closest pair) of a planar point set in amortized $O((\log n)^2)$ time per update. A simplified version of their data structure can be applied to decomposable searching problems. In this way, they get data structures for these problems, in which semi-online updates can be performed efficiently.

In this paper, an alternative description of Dobkin and Suri's algorithm is given. This description has the advantage, that the runtime of the algorithm can be analyzed in a more straightforward way. (Dobkin and Suri bound the runtime of their algorithm using recurrence relations.) This algorithm has an efficient amortized time complexity, and Dobkin and Suri conjecture that this amortized complexity can be made worst-case, in case all insertions and deletions are known before the algorithm starts. In the present paper, this conjecture is proved, even for semi-online updates. It follows e.g. that the diameter (or closest pair) of a set of points in the plane can be maintained under semi-online updates, at the cost of $O((\log n)^2)$ time per update, in the worst case.

Just as in [5], the algorithm is adapted to decomposable searching problems. The amortized version of this algorithm is an extension of Bentley's logarithmic method [2]: if we use the algorithm in case there are only insertions and no deletions, we get exactly Bentley's algorithm. The worst-case version of the adapted algorithm leads to a data structure for decomposable searching problems in which semi-online updates can be performed efficiently, at the cost of a logarithmic increase in query time. For example, we get a data structure in which nearest neighbor queries can be solved in $O((\log n)^2)$ time, and in which semi-online updates take $O((\log n)^2)$ time in the worst case.

As a new application, we adapt the algorithm, such that the answer of a decomposable set problem can be maintained efficiently, when semi-online updates are performed. This leads e.g. to a data structure of size $O(n)$, that maintains the convex hull of n points in three-dimensional space, at the cost of $O(n)$ time per semi-online update. At present, the best data structure for this problem has size $O(n \log\log n)$ and also linear update time, for arbitrary updates. So by restricting ourselves to semi-online updates, we improve upon the space complexity.

The rest of this paper is organized as follows. In Section 2, we introduce the decomposability properties of symmetric bivariate functions. In Section 3, the amortized algorithm

to maintain the maximum of such functions is presented. In Section 4, we transform this amortized algorithm into a worst-case algorithm. In Sections 5, 6 and 7, we apply the given algorithms to the problems of maintaining the diameter and closest pair of a point set, to general decomposable searching problems, and to decomposable set problems. Finally, in Section 8, we give some concluding remarks.

2 The decomposability properties

Let T be a set of objects, and let $f : T \times T \rightarrow \mathcal{R}$ be a symmetric function, i.e. $f(x, y) = f(y, x)$ for all x and y in T. The problem to be studied in the following sections, is the following: we are given a subset V of T, and we are asked to maintain the maximum value of f w.r.t. this set V, under insertions and deletions of objects in V.

We introduce some notations. If $p \in T$, and if V is a subset of T, then

$$f(p, V) := \max_{q \in V} f(p, q).$$

If A and B are subsets of T, then

$$f(A, B) := \max_{p \in A} f(p, B) = \max_{p \in A} \max_{q \in B} f(p, q).$$

The following two lemmas introduce the decomposability properties that enable us to design efficient data structures for maintaining the maximum of f under semi-online updates.

Lemma 1 *Let $f : T \times T \rightarrow \mathcal{R}$ be a symmetric function, and let p be an object in T. If $V_1, \ldots, V_m$ are pairwise disjoint subsets of T, then*

$$f(p, V_1 \cup \ldots \cup V_m) = \max(f(p, V_1), \ldots, f(p, V_m)).$$

Proof: To compute the maximum of $f(p, q)$, where q ranges over $V_1 \cup \ldots \cup V_m$, it is sufficient to compute the maximum of $f(p, q)$ for $q \in V_i$, for each i, and then to take the maximum of all these maxima. □

Lemma 2 *If $f : T \times T \rightarrow \mathcal{R}$ is a symmetric function, and if $V_1, \ldots, V_m$ are pairwise disjoint subsets of T, the union of which is equal to V, then*

$$\max_{p,q \in V} f(p, q) = \max_{1 \leq i \leq m} f(V_i, V_i \cup \ldots \cup V_m). \tag{1}$$

Proof. It is clear that the left-hand side of (1) is at least equal to the right-hand side.

Choose x and y such that $f(x, y)$ is maximal over all pairs in V. Let $1 \leq j, k \leq m$, such that $x \in V_j$ and $y \in V_k$. Since f is symmetric, we may assume that $j \leq k$. Then

$$f(V_j, V_j \cup \ldots \cup V_m) = f(x, y) = \max_{p,q \in V} f(p, q).$$

Hence, the right-hand side of (1) is at least equal to the left-hand side. □

As an example, suppose we want to maintain the *diameter* of a planar point set, i.e., the largest distance between any pair of points in this set. Let T be the set of all points in the plane. For p and q points in T, let $f(p, q)$ be the distance between p and q. Then the maximum of f over all pairs of points in V, is equal to the diameter of V. Also, $f(p, V)$

denotes the maximal distance between p and any point in V, and $f(A, B)$ is equal to the maximal distance between points in A and B.

Other examples of symmetric functions that occur frequently in the theory of algorithms and data structures, are given in [5]. Note that we can also consider the above notions if we replace all occurrences of "max" by "min". In general, we then want to maintain the minimum value of f among all pairs of *distinct* points of V. In this case, the algorithms to be presented have to be modified slightly. (See Section 5.)

3 An amortized algorithm for maintaining the maximum of a symmetric bivariate function

In this section, we give an alternative description of Dobkin and Suri's algorithm for maintaining the maximum value of a symmetric bivariate function, when semi-online updates are performed.

Recall that a sequence of insertions and deletions is called *semi-online*, if the insertions arrive online, but when an object is inserted, we are told how long it will be present. That is, when an object is inserted, we are told how many updates from that moment the object will be deleted, or that the object will remain present forever.

Let $f : T \times T \to \mathcal{R}$ be a symmetric function, and let V be a subset of T, for which we want to maintain the maximum of f. Assume we have a data structure $DS(V)$ that stores the set V, such that queries of the form "compute $f(p, V)$ for $p \in T$" can be computed efficiently. Let $S(n), P(n)$ and $Q(n)$ denote the size of the data structure $DS(V)$, the time needed to build it, and the time needed to answer a query of the above form, respectively. Here, n denotes the cardinality of the set V.

We assume that these three functions are non-decreasing and smooth, where a function g is called *smooth* if $g(O(n)) = O(g(n))$. Furthermore, we assume that $S(n)/n$ and $P(n)/n$ are non-decreasing.

The theorem to be proved is the following:

Theorem 1 (Dobkin and Suri) *There exists a data structure of size $O(S(n))$, such that the maximum value of the symmetric function f can be maintained under semi-online updates in*

$$O\left(\frac{P(n)}{n}\log n + Q(n)\log n + (\log n)^2\right)$$

amortized time per update.

Before we can give the data structure and the corresponding algorithm, we need some definitions. Let m be a positive integer, and assume that we perform a sequence of 2^m updates. We number these updates from 1 to 2^m.

Definition 1 *Let $0 \leq i \leq m$. A sequence of 2^i consecutive updates is called a* block at level i, *if the first of these updates has number $j\,2^i + 1$, for some $0 \leq j < 2^{m-i}$.*

So for each i, the sequence of 2^m updates is partitioned into blocks at level i, where the first block consists of update 1 to 2^i, the second block consists of update $2^i + 1$ to 2^{i+1}, etc.

Definition 2 *Suppose we are processing update k. Let $0 \leq i \leq m$. Then the* current block at level i *is the block at level i, that starts at update $j\,2^i + 1$, where $j = \lfloor (k-1)/2^i \rfloor$. The* next block at level i *is the block at level i, that follows immediately after the current block at level i. The* previous block at level i *is defined similarly.*

So the current block at level i is the unique block at level i to which k belongs.

The data structure : Let V be a subset of T of cardinality n. Let $m = \lfloor \log(n/2) \rfloor$, i.e., the integer m satisfies $2^m \leq n/2 < 2^{m+1}$. We maintain the maximum of f under a sequence of 2^m semi-online updates. These updates are numbered $1, 2, \ldots, 2^m$.

1. At each moment, the set V is partitioned into subsets $V_0, V_1, \ldots, V_m$. For $1 \leq i \leq m$, the set V_i consists of (not necessarily all) objects in V that are still present after the current block at level i is completed. Furthermore, $V_0 = V \setminus \bigcup_{i=1}^{m} V_i$. (Some of the V_i's may be empty.)

2. Each set V_i is stored in a data structure $DS(V_i)$ and in a list, that we call for simplicity V_i.

3. Each object p in V contains a pointer to a list containing the values $f_j(p) := f(p, V_j)$, for $j = i, \ldots, m$, in this order. Here i is such that $p \in V_i$. Also, with the list of p, we store the maximum value $\max(p)$ of the $f_j(p)$'s. By Lemma 1, we have

$$\max(p) = f(p, V_i \cup \ldots \cup V_m).$$

4. There is an array $A(0 : m)$ that contains the values $A(i) := \max_{p \in V_i}(\max(p))$, for $i = 0, 1, \ldots, m$. Hence, $A(i)$ is equal to $f(V_i, V_i \cup \ldots \cup V_m)$. Finally, we store the maximum value $\max(f)$ of this array. By Lemma 2, we have

$$\max(f) = \max_{p,q \in V} f(p, q).$$

The algorithm to be presented below, maintains this partition of V, such that the set V_i has size at most $c\,2^i$, for $0 \leq i \leq m$, where c is a constant independent of i. (We do not prove the upper bound on the size of the V_i's. The proof is similar to that of Lemma 7, where we prove the more difficult case for the worst-case algorithm.) It follows that if an object is to be deleted—by construction, this object is in the set V_0—it is stored in a small data structure. Furthermore, a set V_i is fixed for the current block at level i, so no objects are deleted from this set. Therefore, the data structure $DS(V_i)$ does not have to be changed during the current block at level i, i.e., during a block of 2^i updates.

The basic idea of the algorithm is the following: New objects are inserted at level 0. Then, the object gradually moves to higher levels, the structures at which remain fixed for an ever increasing amount of updates. At the moment that an object is to be deleted soon, this object gradually moves down to level 0, where it arrives at the moment of its deletion.

Initialization: First, we walk along the set V, and select all objects that are still present after the sequence of 2^m updates. These objects are put in the set V_m. Also during this walk, we select the object (if it exists) that will be deleted in the first update, and put it in V_0. Then, we walk along the remaining list. If p is in this list, and if it will be deleted in the d-th update, then p is put in set V_i, where $i = \lceil \log d \rceil - 1$.

Now we store each set V_i in a list—that we call V_i—and we build a data structure $DS(V_i)$ for it.

Next, we do the following for $i = 0, \ldots, m$: For each object p in V_i, compute $f_j(p) := f(p, V_j)$ using the data structure $DS(V_j)$, for $j = i, \ldots, m$, and store these values in a list. Also, compute the maximum $\max(p)$ of the $f_j(p)$'s, and store it with the list of p. Then, compute the maximum of the values $\max(p)$, where p ranges over all objects in V_i, and set its value to $A(i)$.

Finally, we compute the maximum of the array A, and set its value to $\max(f)$.

Lemma 3 *The initialization of the data structure takes $O(P(n) + n\,Q(n))$ time.*

Proof: First consider the partitioning of the set V. Clearly, this can be done in $O(n)$ time. We prove that the partition is correct. Suppose object p is in set V_i. If p is still present after 2^m updates, then $i = m$. If p will be deleted in the first update, then $i = 0$. Otherwise, $i = \lceil \log d \rceil - 1$, where p will be deleted in the d-th update. Since $i < \log d$, we have $d > 2^i$. Hence p is still present after the first block at level i. It follows that the sets V_i satisfy condition 1 of our data structure.

It is not difficult to prove that each subset V_i has size at most $c\,2^i$, where the constant c is independent of i.

Now consider the building of the structures $DS(V_i)$ and the lists V_i. These lists can be built in $O(n)$ time. The building of the data structures takes an amount of time that is bounded above by

$$\sum_{i=0}^{m} P(|V_i|) = O\left(\sum_{i=0}^{m} 2^i \frac{P(2^i)}{2^i}\right) = O\left(\frac{P(n)}{n} \sum_{i=0}^{m} 2^i\right) = O(P(n)).$$

Here we have used the facts that $P(n)$ is smooth, and that $P(n)/n$ is non-decreasing.

Let $p \in V_i$. The time needed to compute $f(p, V_j)$ is bounded by $Q(|V_j|)$. This value is computed for $j = i, \ldots, m$, and we also compute the maximum of these values. Finally, we compute the maximum of all these maxima, where p ranges over V_i. Altogether, this takes an amount of time that is bounded above by $|V_i| \sum_{j=i}^{m} Q(|V_j|)$. Varying i from 0 to m, it follows that the computation of all f-values, and their maxima, takes an amount of time that is bounded above by

$$\begin{aligned}
\sum_{i=0}^{m} |V_i| \sum_{j=i}^{m} Q(|V_j|) &\leq Q(n) \sum_{i=0}^{m} |V_i|(m - i + 1) \\
&= O\left(Q(n) \sum_{i=0}^{m} 2^i\,(m - i + 1)\right) \\
&= O\left(Q(n)\,2^m\right) \\
&= O(n\,Q(n)).
\end{aligned}$$

The final computation of the maximum of the array A takes an extra amount of $O(m) = O(\log n)$ time. It follows that the total initialization time is bounded by $O(n + P(n) + n\,Q(n) + \log n) = O(P(n) + n\,Q(n))$.

It is clear that the given algorithm builds a correct data structure. This proves the lemma. □

The update algorithm: Suppose we perform the k-th update, where $1 \leq k \leq 2^m$. Write k in the binary number system, i.e., $k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_l}$, where $0 \leq i_1 < i_2 < \ldots < i_l \leq m$.

Then after update k has been carried out, the current blocks at levels $0, 1, \ldots, i_1$ are all completed, and these are the only completed blocks.

If the k-th update is an insertion of object q, we insert q in the list V_0, we compute the values $f_j(q) := f(q, V_j)$, for $j = i_1 + 1, \ldots, m$, and we store these values in a list. If the update is a deletion, then the object to be deleted is stored in the list V_0. In this case, we delete it from this list.

Now let W be the union of all lists $V_0, \ldots, V_{i_1}$. Then we use the first part of the above initialization algorithm to partition the set W into new subsets $V_0, \ldots, V_{i_1}$, and we build new lists—which we call again $V_0, \ldots, V_{i_1}$—and new structures $DS(V_0), \ldots, DS(V_{i_1})$ for them. (The set V_{i_1} consists of all objects in W that are still present after the next block at level i_1 is completed.)

Next, we do the following for $i = 0, \ldots, i_1$: For each object p in (the new) V_i, we compute the values of $f_j(p) := f(p, V_j)$, for $j = i, \ldots, i_1$. These values replace the old values of $f_j(p)$, $j \leq i_1$, that are stored at the beginning of p's list. The values of $f_j(p)$, $i_1 < j \leq m$ are not changed, and they remain present in p's list. Next, we compute the maximum of all values $f_j(p)$, $i \leq j \leq m$, and we set its value to $\max(p)$. Then we compute the maximum of all these values $\max(p)$, where p ranges over all objects in V_i, and we store the result in the array-entry $A(i)$.

Finally, we compute the maximum of the array $A(0 : m)$, and we set its value to $\max(f)$. This value is the new maximum of f over all pairs of objects that are present at this moment.

Lemma 4 *The above update takes an amount of time that is bounded by*

$$O(Q(n) \log n + P(2^{i_1}) + 2^{i_1} Q(2^{i_1}) + 2^{i_1} \log n),$$

where the constant factor is independent of i_1.

Proof: In case of an insertion of object q, it takes

$$\sum_{j=i_1+1}^{m} Q(|V_j|) \leq \sum_{j=i_1+1}^{\log n} Q(n) = O(Q(n) \log n)$$

time to compute the values $f_j(q)$ for $j = i_1 + 1, \ldots, m$. Since $|W| \leq \sum_{j=0}^{i_1} c\, 2^j = O(2^{i_1})$, it follows from Lemma 3, that it takes $O(P(2^{i_1}) + 2^{i_1} Q(2^{i_1}))$ time to build the lists $V_0, \ldots, V_{i_1}$, the structures $DS(V_0), \ldots, DS(V_{i_1})$, and the values $f_j(p)$ for $j = i, \ldots, i_1$ and $p \in V_i$, where $i = 0, \ldots, i_1$. Furthermore, it takes

$$O\left(\sum_{i=0}^{i_1} |V_i|(m - i)\right) = O\left(\sum_{i=0}^{i_1} 2^i (m - i)\right) = O\left(2^{i_1} \log n\right)$$

time to compute the maxima of the lists $f_i(p), \ldots, f_m(p)$, for all $p \in V_i$, and the maximum of all these maxima over all $p \in V_i$, where $i = 0, \ldots, i_1$. Finally, the update of the maximum of the array $A(0 : m)$ takes $O(m) = O(\log n)$ time. This proves the time bound.

It remains to prove that the update algorithm is correct. Let $0 \leq i \leq m$. Then the integer $k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_l}$ is divisible by 2^i if and only if $0 \leq i \leq i_1$. Hence, indeed the current blocks at levels $0, 1, \ldots, i_1$ are the only ones that are completed. Clearly, after the update, the lists, the structures, and the various values of f, and their maxima, for the levels $0, 1, \ldots, i_1$ have the correct values. Also, the array segment $A(0 : i_1)$ has the correct value. Since at levels $i_1 + 1, \ldots, m$ no blocks are completed, nothing has to be changed for

the structures and the values of f corresponding to these levels. This proves the correctness of the update algorithm. □

Proof of Theorem 1: First we bound the time needed to perform a sequence of 2^m updates as described above. By Lemma 3, it takes $O(P(n)+n\,Q(n))$ time to build the data structure.

The complexity of the k-th update depends on the least significant bit in the binary representation of k. (See Lemma 4.) Let k_i be the number of times that during the updates $1, 2, \ldots, 2^m$, the least significant bit of k is equal to i. It is easy to see that k_i is equal to the total number of blocks at level $i+1$. Hence, $k_i = 2^{m-i-1}$ for $0 \leq i < m$, and $k_m = 1$. Therefore, $k_i \leq 2^{m-i} \leq n/2^i$ for $0 \leq i \leq m$.

It follows from Lemma 4, that the total amount of time needed to perform 2^m updates is bounded above by

$$\begin{aligned}
& P(n) + n\,Q(n) + \sum_{i=0}^{m} k_i \left(Q(n)\log n + P(2^i) + 2^i\,Q(2^i) + 2^i \log n \right) \\
&= O\left(P(n) + n\,Q(n) + \sum_{i=0}^{m} \frac{n}{2^i} \left(Q(n)\log n + P(2^i) + 2^i\,Q(n) + 2^i \log n \right) \right) \\
&= O\left(P(n) + n\,Q(n) + n\,Q(n)\log n + P(n)\log n + n\,Q(n)\log n + n(\log n)^2 \right) \\
&= O\left(P(n)\log n + n\,Q(n)\log n + n(\log n)^2 \right).
\end{aligned}$$

So, amortized, we spend an amount of time per update during this sequence, that is bounded by

$$O\left(\frac{P(n)\log n}{2^m} + \frac{n\,Q(n)\log n}{2^m} + \frac{n(\log n)^2}{2^m} \right) = O\left(\frac{P(n)}{n}\log n + Q(n)\,\log n + (\log n)^2 \right),$$

where we have used that $n/2 < 2^{m+1}$.

Since $2^m \leq n/2$, the number of objects is always between $n/2$ and $3n/2$. It follows—because our complexity measures are assumed to be smooth—that the amortized amount of time per update during this sequence of 2^m updates is bounded by $O((P(N)/N)\log N + Q(N)\log N + (\log N)^2)$, if N is the number of objects at the moment of the update.

After this sequence of 2^m updates, we choose a new value of m, depending on the number of objects at that moment, we build a new data structure, and we proceed in the same way. This proves the bound on the amortized update time.

We are left with the bound on the size of the data structure. The data structures $DS(V_i)$ and the lists V_i together have a size that is bounded by $O(n) + \sum_{i=0}^{m} S(|V_i|)$. Since $S(n)/n$ is non-decreasing, and since $|V_i| \leq c\,2^i$ for some constant c, it follows that this sum is bounded by $O(S(n))$.

If p is an object in set V_i, then it contains the values $f_i(p), \ldots, f_m(p)$, and $\max(p)$. It follows that the total size of these values for set V_i is bounded by $O(2^i(m-i+2))$. Therefore, all these values for all objects of V together have size at most $O(\sum_{i=0}^{m} 2^i(m-i+2)) = O(2^m) = O(n)$. The array A and the value $\max(f)$ have size $O(m) = O(\log n)$. Hence, the entire data structure has size at most $O(S(n)+n) = O(S(n))$. Since the current number of objects is $\Theta(n)$, and since $S(n)$ is smooth, we have proved that the size of the data structure is bounded by $O(S(N))$, if N is the current number of objects. This proves Theorem 1. □

4 The worst-case algorithm

We turn the amortized bound of Theorem 1 into a worst-case bound. If in the algorithm of the preceding section the value of i_1 is high, we spend a large amount of time in that update. For such updates, we divide this large amount of time over a number of updates. In this way, the worst-case update time turns out to be small.

Let $f : T \times T \to \mathcal{R}$ be a symmetric function, and let V be a subset of T of cardinality n, for which we want to maintain the maximum of f. Let $DS(V)$ be a data structure storing V, that has size $S(n)$, and that can be built in $P(n)$ time, such that for $p \in T$, the value of $f(p, V)$ can be computed in $Q(n)$ time. Again, we assume that these three functions are non-decreasing and smooth, and that $S(n)/n$ and $P(n)/n$ are non-decreasing.

The data structure: Let m be an integer, such that $2^m \leq n/2 \leq 2^{m+2}$. We maintain the maximum of f under a sequence of 2^m semi-online updates. Again, we number these updates $1, 2, \ldots, 2^m$.

1. At each moment, the set V is partitioned into subsets $V_2, V_3, \ldots, V_{m-1}$. For $3 \leq i \leq m-1$, the set V_i consists of objects that are still present after the current block at level i is completed. Furthermore, $V_2 = V \setminus \bigcup_{i=3}^{m-1} V_i$. (Some of the V_i's may be empty.)

2. For each $3 \leq i \leq m-1$, there is a data structure $DS_i := DS(V_i)$, and there are lists $V_i^2, V_i^3, \ldots, V_i^i$, where V_i^j stores the set V_j. So each level i contains the objects at its own level, and all objects at the lower levels.

3. For $i = 2$, there is a list V_2^2 that contains the set V_2.

4. Each object p in V contains a pointer to a list containing the values $f_j(p) := f(p, V_j)$, for $j = i, \ldots, m-1$, in this order. Here i is such that $p \in V_i$. Also, with the list of p, we store the maximum value $\max(p)$ of the $f_j(p)$'s.

5. There is an array $A(2 : m-1)$ that contains the values $A(i) := \max_{p \in V_i}(\max(p))$. Hence, $A(i)$ is equal to $f(V_i, V_i \cup \ldots \cup V_{m-1})$. Finally, we store the maximum value $\max(f)$ of this array. By Lemma 2, $\max(f)$ is the maximum of f over all pairs of points in V.

Consider a sequence of 2^m updates. In what follows, we speak about the next block at level $i+1$. Later, we guarantee that the value of i is such that this next block exists. (Note that after 2^m updates, the value of n, and hence that of m, might be changed. Therefore, if the value of m is decreased to $m-1$, and if $i = m-2$, then during the last block at level i, there is no next block at level $i+1$.)

Let $3 \leq i \leq m-2$. We describe how a block at level i—having length 2^i—is processed. We split this block in 8 parts, each of length 2^{i-3}. (Note that $2^{i-3} \geq 1$.)

Part 1 of level i: This first part consists of updates $1, 2, \ldots, 2^{i-3}$. During these updates, we make two lists F_i^{i+1} and F_i^{i-1}, where F_i^{i+1} are all objects in V_i^i that are still present after the current and next blocks at level $i+1$ are completed, and $F_i^{i-1} = V_i^i \setminus F_i^{i+1}$.

Later, we prove that $|V_i^i| = O(2^i)$. Therefore, this partitioning takes $O(2^i)$ time. With each update, we do an amount of $O(2^i)/2^{i-3} = O(1)$ work.

Comments: Since $F_i^{i-1} \subseteq V_i^i$, we see that all objects in F_i^{i-1} are still present after the current and next blocks at level $i-1$ are completed.

During Part 1 of the processing of the current block at level $i-1$, we have computed a list F_{i-1}^i of objects that are still present after the current and next blocks at level i are completed. This list F_{i-1}^i is available after update 2^{i-4}, hence surely at the start of Part 2 of level i. (If $i=3$, we guarantee later, that the list F_2^3 is available after the first update of the current block at level 2, hence at the start of Part 2 of the current block at level 3.)

Part 2 of level i: The second part consists of the next 2^{i-3} updates. During this part, we copy the list F_{i-1}^i into a list H_i, we merge the lists F_i^{i+1} and F_{i-1}^i in a list N_i^i, and we build a structure $DS(N_i^i)$. We also make for each $j=i+1,\ldots,m-1$, a copy N_j^i of the list N_i^i.

Later, we prove that the size of the sets F_i^{i+1} and F_{i-1}^i are both bounded by $O(2^i)$. Therefore, it takes $O(P(2^i))$ time to build $DS(N_i^i)$, and $O(2^i(m-i)) = O(2^i \log n)$ time to make the lists $H_i, N_i^i, N_{i+1}^i, \ldots, N_{m-1}^i$. With each update, we do an amount of $O(2^i \log n + P(2^i))/2^{i-3} = O(\log n + P(2^i)/2^i)$ work.

Comments: After Part 4 of level i, the new structure $DS(N_i^i)$ takes over the role of the structure DS_i. Therefore, after Part 4, each object p that is at one of the levels $2,\ldots,i$ at that moment, must have the value $f(p,N_i^i)$ in its list. At the beginning of Part 3 of level i, the lists $V_i^2,\ldots,V_i^i$ contain all objects at levels $2,\ldots,i$. After Part 4 of level i, these lists have been changed, but together they contain more or less the same objects. (Of course, objects have been deleted, and new objects have been inserted in the mean time. During Parts 3 and 4 of level i, however, no objects from levels $> i$ move to these lists.)

Part 3 of level i: The third part consists of the next 2^{i-3} updates. During this part, we compute for each object p in the lists $V_i^2, V_i^3 \ldots, V_i^i$, the value of $f_i'(p) := f(p,N_i^i)$, using the structure $DS(N_i^i)$.

This takes altogether an amount of time that is bounded by

$$Q(|N_i^i|)\sum_{j=2}^{i}|V_i^j| = O\left(Q(2^i)\sum_{j=2}^{i}2^j\right) = O(2^i Q(2^i)).$$

With each update, we do $O(2^i Q(2^i))/2^{i-3} = O(Q(2^i))$ work.

Note that during Part 3 of level i, several blocks at lower levels are completed. Hence, during Part 3 of level i, some of the lists V_i^j will change. If a list V_i^j is replaced by a new one, we keep the old list—we call it O_i^j—and upon computing the values of $f_i'(p)$ for objects at level j, we let p vary over the list O_i^j. Afterwards, we discard all old lists O_i^j. This increases the time and space bounds by at most a constant factor.

Also, if the actual update is an insertion of object q, we compute the value of $f_i'(q) := f(q,N_i^i)$.

Part 4 of level i: This part consists of the next 2^{i-3} updates. During this part, we give each object p in the list N_i^i, a new list $f_i(p) := f_i'(p)$, $f_{i+1}(p)$, $f_{i+2}(p)$, $\ldots$, $f_{m-1}(p)$. We also compute the maximal value of this new list, which we call $\max(p)$. Finally, we compute the maximum $\max_i$ of the values $\max(p)$ over all $p \in N_i^i$.

This takes $O(|N_i^i|(m-i)) = O(2^i(m-i)) = O(2^i \log n)$ time. With each update, we do an amount of $O(2^i \log n)/2^{i-3} = O(\log n)$ work.

Again, if the actual update is an insertion of object q, we compute the value of $f_i'(q) := f(q,N_i^i)$.

After Part 4 of level i: After Part 4, we have processed 2^{i-1} updates. Now we set $V_i := N_i^i$ (note that V_i is a set of objects that we do not store implicitly; we only need it for reference); we set $V_j^i := N_j^i$, for $j=i,\ldots,m-1$; we set $DS_i := DS(N_i^i)$; and we set

$A(i) := \max_i$. Hence, $A(i)$ is equal to $f(V_i, V_i \cup \ldots \cup V_{m-1})$. Finally, we replace F_i^{i-1} by an empty list F_i^{i-1}. All of this can be done in $O(m-i) = O(\log n)$ time.

Comments: Let p be an object that is at a level $\leq i$, after Part 4 of level i is completed. First suppose that p was inserted during Part 3 or 4 of level i. Then we have computed the value of $f_i'(p) = f(p, N_i^i)$. Otherwise, p was present already at the start of Part 3 of level i. Then, at the start of this Part 3, object p was already at a level $\leq i$. That is, at that moment, p was in one of the lists $V_i^2, \ldots, V_i^i$. It follows that with p, there is a value $f_i'(p) = f(p, N_i^i)$. If p is at level $j < i$ after Part 4 of level i is completed, then during the processing of Part 8 of the current block at level j—which is completed at the same time as Part 4 of the current block at level i is completed—we take care that the list of p gets the value $f_i'(p)$. (If $j = 2$, we take care that during the last update of the current block at level 2, the list of p gets $f_i'(p)$.) Hence, each object p that is at one of the levels $2, \ldots, i$, after Part 4 of level i is completed, contains the value of $f(p, N_i^i)$ in its list.

Next, note that the objects that are present after Part 4 of level i is completed, are still partitioned into subsets $V_2, \ldots, V_{m-1}$. In particular, these sets are still pairwise disjoint: At level i, the objects in F_i^{i-1} (here we mean the list F_i^{i-1} that was made during Part 1 of level i) have disappeared. At this moment, these objects are, however, stored in the data structure DS_{i-1}. (This structure was built during Part 6 of level $i-1$, and it took over the role of the old DS_{i-1} after Part 8 of level $i-1$. The end of Part 8 of level $i-1$ coincides with the end of Part 4 of level i. If $i = 3$, the list F_3^2 is part of the current list V_2^2.) Also, at the end of Part 4 of level i, the objects of the set F_{i-1}^i are stored in DS_i. These objects were contained in the old structure DS_{i-1} of level $i-1$, which is replaced at the end of Part 8 of level $i-1$ (hence at the end of Part 4 of level i) by a new structure DS_{i-1} that does not contain objects of F_{i-1}^i.

It follows that all structures and variables corresponding to this value of i are correct.

Part 5 of level i: This part consists of the next 2^{i-3} updates. During these updates, nothing happens at level i.

Comments: Suppose that the current block at level i coincides with the first half of the current block at level $i+1$. Then during Part 1 of the current block at level $i+1$ (which is completed after Part 2 of the current block at level i) we have computed a list F_{i+1}^i of objects that are still present after the current and next blocks at level i are completed. Otherwise, the current block at level i coincides with the second half of the current block at level $i+1$. Then, after Part 4 of the current block at level $i+1$ (which is completed at the start of Part 1 of the current block at level i) there is an empty list F_{i+1}^i.

Similarly, during Part 1 of the current block at level $i-1$ (which coincides with the first half of Part 5 of the current block at level i) we have computed a list F_{i-1}^i of objects that are still present after the current and next blocks at level i are completed. (If $i = 3$, we guarantee later that the list F_2^3 is available after the first update of the current block at level 2, hence at the start of Part 6 of the current block at level i.)

Hence, at the start of Part 6 of the current block at level i, the lists F_{i+1}^i and F_{i-1}^i are available.

Part 6 of level i: During this part—which consists of the next 2^{i-3} updates—we merge the lists H_i, F_{i+1}^i and F_{i-1}^i in a list N_i^i, and we build a structure $DS(N_i^i)$ for it. We also make for each $j = i+1, \ldots, m-1$, a copy N_j^i of the list N_i^i. Just as in Part 2, we do with each update an amount of $O(\log n + P(2^i)/2^i)$ work.

Comments: After Part 8 of level i, the new structure $DS(N_i^i)$ takes over the role of the structure DS_i. Therefore, after Part 8, each object p that is at one of the levels $2, \ldots, i$ at that moment, must have the value $f(p, N_i^i)$ in its list.

Part 7 of level i: This part consists of the next 2^{i-3} updates. During this part, we compute for each object p in the lists $N_i^i, V_i^2, \ldots, V_i^{i-1}$, the value of $f_i'(p) := f(p, N_i^i)$, using the structure $DS(N_i^i)$. (During Part 7 of level i, several blocks at lower levels are completed, and, hence, during Part 7 of level i, some of the lists V_i^j will change. If a list V_i^j is replaced by a new one, we keep the old list—we call it O_i^j—and upon computing the values of $f_i'(p)$ for objects at level j, we let p vary over the list O_i^j. Afterwards, we discard all old lists O_i^j.) Just as in Part 3, we do with each update an amount of $O(Q(2^i))$ work.

Also, if the actual update is an insertion of object q, we compute the value of $f_i'(q) := f(q, N_i^i)$.

Part 8 of level i: This part consists of the final 2^{i-3} updates of the current block at level i. After this block is completed, there are blocks at other levels that are also completed. Suppose that after Part 8 of level i, all blocks at levels $2, 3, \ldots, i_1$ are completed. (The value of i_1 can be determined in the same way as in the update algorithm of Section 3.) Let $j = \min(i_1 + 1, m - 1)$.

Let $p \in N_i^i$. During Part 3 of level j, object p was at one of the levels $\leq j$, or it was inserted during that part. Therefore, there is a value $f_j'(p) = f(p, N_j^j)$, where N_j^j becomes the new set V_j after Part 4 of level j (hence after Part 8 of level i) is completed. Similarly, during Part 7 of levels $a = i, \ldots, j-1$, object p was at one of the levels $\leq a$. So there is a value $f_a'(p) = f(p, N_a^a)$, where N_a^a becomes the new set V_a after Part 8 of level a (hence after Part 8 of level i) is completed.

During Part 8 of level i, we give each object p in N_i^i, a new list $f_i(p) := f_i'(p)$, $f_{i+1}(p) := f_{i+1}'(p)$, $\ldots$, $f_j(p) := f_j'(p)$, $f_{j+1}(p)$, $f_{j+2}(p)$, $\ldots$, $f_{m-1}(p)$. (If $j = m-1$, there are only f'-values in this sequence, and no f-values.) We also compute the maximal value of this new list, which we call $\max(p)$. Finally, we compute the maximum $\max_i$ of the values $\max(p)$ over all $p \in N_i^i$.

This takes $O(|N_i^i|(m-i)) = O(2^i(m-i)) = O(2^i \log n)$ time. With each update, we do an amount of $O(\log n)$ work.

Again, if the current update is an insertion of object q, we compute the value of $f_i'(q) := f(q, N_i^i)$.

After Part 8 of level i: After Part 8 is completed, we set $V_i := N_i^i$; we set the lists $V_j^i := N_j^i$, for $j = i, \ldots, m-1$; we set $DS_i := DS(N_i^i)$; and we set $A(i) := \max_i$. Hence, $A(i)$ is equal to $f(V_i, V_i \cup \ldots \cup V_{m-1})$. All of this can be done in $O(m-i) = O(\log n)$ time.

Comments: Let p be an object that is at a level $\leq i$, after Part 8 of level i is completed. Then in the same way as in the comments after Part 4 of level i, it follows that there is a value $f_i'(p) = f(p, N_i^i)$ for object p. If p is at level $a < i$ after Part 8 of level i is completed, then during the processing of Part 8 of the current block at level a—which is completed at the same time as Part 8 of the current block at level i is completed—we take care that the list of p gets the value $f_i'(p)$. (If $a = 2$, we take care that during the last update of the current block at level 2, the list of p gets $f_i'(p)$.) Hence, each object p that is at one of the levels $2, \ldots, i$, after Part 8 of level i is completed, contains the value of $f(p, N_i^i)$ in its list.

Again, the objects that are present after Part 8 of level i, are still partitioned into subsets $V_2, \ldots, V_{m-1}$. In particular, these sets are still pairwise disjoint: this can be proved in the same way as in the comments after Part 4 of level i.

It follows that all structures and variables corresponding to this value of i are correct.

This concludes the update algorithm for a block at a level $3 \leq i \leq m-2$. Next we describe how a block at level 2 is processed.

Block at level 2: During the first update, we make two lists F_2^3 and F_2^2, where F_2^3

are all objects in V_2^2 that are still present after the current and next blocks at level 3 are completed, and $F_2^2 = V_2^2 \setminus F_2^3$.

Afterwards, we perform the actual update: If the update is an insertion of object q, we insert q in the set V_2 and in the lists $F_2^2, V_2^2, V_3^2, \ldots, V_{m-1}^2$; we compute the values of $f_i(q) := f(q, V_i)$ using the list V_2^2 if $i = 2$, and using the structures DS_i if $3 \leq i \leq m-1$; we store these values in a list; and we compute the maximum $\max(q)$ of this list. If the update is a deletion of the object q, we delete q from the set V_2 and from the lists $F_2^2, V_2^2, V_3^2, \ldots, V_{m-1}^2$ (we know that q is in all these lists); and we discard all further information corresponding to q. After this insertion or deletion, we compute for each $p \in V_2^2$ the value of $f_2(p) := f(p, V_2)$ using the list V_2^2, and we replace the first element of p's list by this new value $f_2(p)$; we compute for each $p \in V_2^2$ the maximal value $\max(p)$ in its list; we compute the maximum of the values $\max(p)$, where p ranges over V_2^2, and we set $A(2)$ to this maximum.

In the second and third update, we only perform the actual update as just described.

In the fourth—and final—update, we first insert or delete the object in the list F_2^2. Next, we set $V_2 := F_2^2 \cup F_3^2$; we merge the lists F_2^2 and F_3^2 in a list V_2^2; and we copy this list into lists $V_3^2, \ldots, V_{m-1}^2$. (These lists V_j^2 replace the old ones. Note that the list F_3^2 has been computed during Part 1 of level 3, in case the current block at level 2 coincides with the first half of the current block at level 3. Otherwise, F_3^2 is empty. So these lists are indeed available if the fourth update of the current block at level 2 is processed.) Let i_1 be the highest level at which the current block is completed, after the current block at level 2 is completed. Let $j = \min(i_1 + 1, m-1)$. If the current update is an insertion of object q, we compute $f_2(q) := f(q, V_2)$, and $f_a(q) := f(q, V_a)$ for $a = j+1, \ldots, m-1$. Note that for all objects p in V_2 (also for $p = q$) the values of $f'_a(p) = f(p, N_a^a)$, $a = 3, \ldots, j$, have been computed during Part 3 of level j, and during Part 7 of levels $3, \ldots, j-1$. Now we give each object p in V_2 the list $f_2(p)$, $f_3(p) := f'_3(p)$, $f_4(p) := f'_4(p)$, $\ldots$, $f_j(p) := f'_j(p)$, $f_{j+1}(p), \ldots, f_{m-1}(p)$, and we compute the maximum $\max(p)$ of this list. Finally, we set $A(2)$ to the maximum of all values $\max(p)$, where p ranges over V_2.

Comments: Since the size of V_2^2 is bounded above by a constant, each update of a block at level 2 takes an amount of time that is bounded by

$$O\left(\log n + \sum_{i=2}^{m-1} Q(|V_i|)\right) = O\left(\log n + \sum_{i=2}^{m-1} Q(2^i)\right).$$

The first block at level m − 1: The first block at level $m-1$ is processed similarly as a block at level i. Again, this block is split in 8 parts, each of length 2^{m-4}. During Part 1 of level $m-1$, we make two lists F_{m-1}^{m-1} and F_{m-1}^{m-2}, where F_{m-1}^{m-1} are all objects in V_{m-1}^{m-1} that are still present after the current and next blocks at level $m-1$ are completed, and $F_{m-1}^{m-2} = V_{m-1}^{m-1} \setminus F_{m-1}^{m-1}$. We divide this work among the 2^{m-4} updates, each update spending an amount of $O(1)$ time.

During Part 2, we copy the list F_{m-2}^{m-1} into a list H_{m-1}, we merge the lists F_{m-1}^{m-1} and F_{m-2}^{m-1} in a list N_{m-1}^{m-1}, and we build a structure $DS(N_{m-1}^{m-1})$. This takes an amount of $O(2^m + P(2^m)) = O(P(2^m))$ time, that we divide among the 2^{m-4} updates of Part 2. With each update, we do $O(P(2^m)/2^m)$ work.

During Part 3, we compute for each object p in the lists $V_{m-1}^2, V_{m-1}^3, \ldots, V_{m-1}^{m-1}$, the value of $f'_{m-1}(p) := f(p, N_{m-1}^{m-1})$, using the structure $DS(N_{m-1}^{m-1})$. (If one of these lists V_{m-1}^j changes—because a block at a lower level is completed—we keep the old list in a list O_{m-1}^j,

as before.) This takes an amount of time that is bounded by

$$Q(|N_{m-1}^{m-1}|) \sum_{j=2}^{m-1} |V_{m-1}^{j}| = O\left(Q(2^m) \sum_{j=2}^{m-1} 2^j \right) = O(2^m\, Q(2^m)).$$

With each update, we do $O(Q(2^m))$ work. Also, if the actual update is the insertion of object q, we compute the value of $f'_{m-1}(q) := f(q, N_{m-1}^{m-1})$.

During Part 4, we set $f_{m-1}(p) := \max(p) := f'_{m-1}(p)$ for each object p in the list N_{m-1}^{m-1}. So p becomes a new list consisting of only one element $f_{m-1}(p)$. Then, we compute the maximum $\max_{m-1}$ of the values $\max(p)$ over all $p \in N_{m-1}^{m-1}$. This takes $O(2^m)$ time. With each update, we do an amount of $O(1)$ work. Also, if the actual update is an insertion of object q, we compute the value of $f'_{m-1}(q) := f(q, N_{m-1}^{m-1})$.

After Part 4 of level $m-1$, we set $V_{m-1} := N_{m-1}^{m-1}$; we set the list $V_{m-1}^{m-1} := N_{m-1}^{m-1}$; we set $DS_{m-1} := DS(N_{m-1}^{m-1})$; and we set $A(m-1) := \max_{m-1}$. Hence, $A(m-1)$ is equal to $f(V_{m-1}, V_{m-1})$. Finally, we initialize an empty list F_{m-1}^{m-2}. All of this can be done in $O(1)$ time.

During Part 5, nothing happens at level $m-1$. During Part 6, we merge the lists H_{m-1}, F_{m-1}^{m-1} and F_{m-2}^{m-1} in a list N_{m-1}^{m-1}, and we build a structure $DS(N_{m-1}^{m-1})$ for it. Just as in Part 2, we do with each update an amount of $O(P(2^m)/2^m)$ work.

During Part 7, we compute for each object p in the lists $N_{m-1}^{m-1}, V_{m-1}^{2} \ldots, V_{m-1}^{m-2}$, the value of $f'_{m-1}(p) := f(p, N_{m-1}^{m-1})$, using the structure $DS(N_{m-1}^{m-1})$. (Again, if a list V_{m-1}^{j} is replaced by a new one, we keep the old list in list O_{m-1}^{j}.) Just as in Part 3, we do with each update an amount of $O(Q(2^m))$ work. Also, if the actual update is the insertion of object q, we compute the value of $f'_{m-1}(q) := f(q, N_{m-1}^{m-1})$.

During Part 8, we set $f_{m-1}(p) := \max(p) := f'_{m-1}(p)$ for each object p in the list N_{m-1}^{m-1}, and we compute the maximum $\max_{m-1}$ of the values $\max(p)$ over all $p \in N_{m-1}^{m-1}$. This takes $O(2^m)$ time. With each update, we do an amount of $O(1)$ work. Again, if the actual update is the insertion of object q, we compute the value of $f'_{m-1}(q) := f(q, N_{m-1}^{m-1})$.

After Part 8 of level $m-1$, we set $V_{m-1} := N_{m-1}^{m-1}$; we set the list $V_{m-1}^{m-1} := N_{m-1}^{m-1}$; we set $DS_{m-1} := DS(N_{m-1}^{m-1})$; and we set $A(m-1) := \max_{m-1}$. All of this can be done in $O(1)$ time.

We saw already that after the entire sequence of 2^m updates, the value of m—and hence the number of levels—might have to be changed. In order to guarantee that the number of levels is correct, we have to make new levels, or we have to decrease the number of levels. Since this will take a considerable amount of time, we have to divide this work among a number of updates. Therefore, after the first block at level $m-1$ is completed, we already choose a new value for m, based on the total number of objects that are present at that moment.

Let n_0 be the total number of objects that are present after the first 2^{m-1} updates have been processed. Then we take $m' := \lfloor \log(n_0/3) \rfloor$. After the final 2^{m-1} updates have been processed, there will be levels $2, 3, \ldots, m'-1$. We will prove in Lemma 5, that $m-1 \leq m' \leq m+1$. Hence, the number of levels decreases by one, does not change, or increases by one.

We have to describe the processing of the blocks at levels $2, 3, \ldots, m-1$ during the final 2^{m-1} updates.

For $2 \leq i \leq \min(m-2, m'-2)$, a block at level i is processed as described above. Note that for these values of i, the notion of "next block at level $i+1$" indeed has sense. Also,

the list F^i_{i+1} that we need during Part 6 of level i, is available at the right moment. Only for the last block at level i, there are some differences: During Part 6 of this last block at level i, we make for each $j = i+1, \ldots, m'-1$, a copy N^i_j of the list N^i_i. (So we replace m by m'.) During Part 8 of the last block at level i, we give each object p in N^i_i, a new list $f_i(p) := f'_i(p), \ldots, f_j(p) := f'_j(p), f_{j+1}(p), \ldots, f_{m'-1}(p)$, where $j = \min(i_1 + 1, m' - 1)$. (For the notation, see the description of the processing of Part 8 of a block at level i. Note that if $m' \neq m$ and $j = m' - 1$, we guarantee later that the values of $f'_{m'-1}(p)$ are indeed available.) After Part 8, we set $V_i := N^i_i$; $V^i_j := N^i_j$ for $j = i, \ldots, m' - 1$; $DS_i := DS(N^i_i)$; $A(i) := \max_i$.

If $m' = m$, we process the second block at level $m-1$ in the same way as we processed the first block at level $m-1$.

If $m' = m+1$, we process the second block at level $m-1$ in a similar way as we processed a block at level i, for $3 \leq i \leq m-2$. Now, level $m-1$ "passes" objects to levels $m-2$ and m, and it "receives" objects from level $m-2$. It does not, however, "receive" objects from level m. The objects that are passed to level m are built in a data structure DS_m, and a list N^m_m. Also, for each object p, we add at the end of its list the value of $f_m(p) := f(p, N^m_m)$, and the maximal value $\max(p)$ of this new list. Finally, we replace the array $A(2 : m-1)$ by an array $A(2 : m)$. In this new array, the first $m-2$ entries coincide with the old array, and $A(m)$ is the maximum of the values $\max(p)$ over all $p \in N^m_m$.

We are left with the case $m' = m - 1$. During second the block at level $m-1$, we pass objects down to level $m-2$, and the other objects stay at level $m-1$ (for only the first half of this block at level $m-1$). During the final 2^{m-1} updates, there are two blocks at level $m-2$. The first of these blocks is processed as a block at level i (for $3 \leq i \leq m-2$) as described above. So during this first block, level $m-2$ "receives" objects from level $m-1$ that are still present after the current and next blocks at level $m-2$ are completed. During the last block at level $m-2$, this level receives objects from level $m-3$, and it passes objects to level $m-3$. During this last block at level $m-2$, we make out of the objects of level $m-2$, the objects of level $m-1$ that are not passed down during the first half of the last block at level $m-1$, and the objects that are received from level $m-3$, a new data structure at level $m-2$, and new lists and f-values. Finally, we replace the array $A(2 : m-1)$ by an array $A(2 : m-2)$, the value of which is obvious.

Now we are ready to give the final update algorithm.

The update algorithm: With each update, we perform an update at the current blocks at all levels, $2, 3, \ldots, m-1$, as described above. After this has been completed, we compute the maximum $\max(f)$ of the array $A(2 : m-1)$. This maximum is equal to the maximum of f over all pairs of objects that are present at this moment. After the first block at level $m-1$ is completed, we choose the value of m', as described above.

Let n' be the number of objects, after 2^m updates have been processed. It is shown in Lemma 6 that $2^{m'} \leq n'/2 \leq 2^{m'+2}$. Hence, we are in the same situation as the one we started with. Therefore, we can proceed performing updates in this way.

In the rest of this section, we prove some lemmas and the final theorem, in which the worst-case update time bound is proved. The correctness of the update algorithm follows from the comments that were given during the description of the update algorithm for the various levels.

Lemma 5 *The integers m and m', as defined above, satisfy $m - 1 \leq m' \leq m + 1$.*

Proof: Recall that at the start of the algorithm, there are n objects, and the integer m satisfies $2^m \leq n/2 \leq 2^{m+2}$. After 2^{m-1} updates, there are n_0 objects, and we set

$m' = \lfloor \log(n_0/3) \rfloor$. It follows that

$$\frac{n_0}{3} \leq \frac{n + 2^{m-1}}{3} \leq \frac{2^{m+3} + 2^{m-1}}{3} = \frac{17}{3} 2^{m-1} < 2^{m+2}.$$

Hence, $m' \leq \log(n_0/3) < m + 2$, which proves that $m' \leq m + 1$. Similarly, we have

$$\frac{n_0}{3} \geq \frac{n - 2^{m-1}}{3} \geq \frac{2^{m+1} - 2^{m-1}}{3} = 2^{m-1}.$$

This proves that $\log(n_0/3) \geq m - 1$. Since $m' = \lfloor \log(n_0/3) \rfloor$ is an integer, it follows that $m' \geq m - 1$. □

Lemma 6 *Let n' be the number of objects that are present after 2^m updates have been processed. Then*

$$2^{m'} \leq n'/2 \leq 2^{m'+2}.$$

Proof: Since n_0 is the number of objects after 2^{m-1} updates, and since $2^m \leq n/2$, we have

$$n_0 \geq n - 2^{m-1} \geq n - n/4 = 3n/4.$$

It follows that

$$n' \leq n_0 + 2^{m-1} \leq n_0 + n/4 \leq n_0 + n_0/3 = 4n_0/3.$$

Hence,

$$n'/2 \leq 2n_0/3 = 2^{1+\log(n_0/3)} \leq 2^{2+\lfloor \log(n_0/3) \rfloor} = 2^{m'+2}.$$

Similarly we have

$$n' \geq n_0 - 2^{m-1} \geq n_0 - n/4 \geq n_0 - n_0/3 = 2n_0/3.$$

Hence,

$$2^{m'} = 2^{\lfloor \log(n_0/3) \rfloor} \leq 2^{\log(n_0/3)} = n_0/3 \leq n'/2.$$

This proves the lemma. □

Next, we prove the upper bound on the sizes of the sets V_i.

Lemma 7 *Let $2 \leq i \leq m - 1$, and let V_i be the set of objects that are stored at level i. This set has size at most $18 \cdot 2^i$.*

Proof: The set V_{m-1} can contain all objects that are present. Since during a sequence of 2^m updates, the number of objects is at most $n + 2^m$, and since $n \leq 2^{m+3}$, it follows that, at any moment, the size of V_{m-1} is at most $2^{m+3} + 2^m = 18 \cdot 2^{m-1}$.

Now let $2 \leq i \leq m - 2$. Consider the current block at level i. Let V_i be the set of objects that are at level i at the start of this current block. Let $p \in V_i$. There are three possible cases.

Case 1: At some moment, p has moved from level j to level $j - 1$, for some $j \geq 3$.

Suppose that $j \geq i+2$. At the start of the current block at level i, object p is in V_i. Since objects only move one level up or down, it follows that—at some moment—p has moved from level $i + 1$ to level i. Therefore, we may assume that $j \leq i + 1$.

So, at some moment, object p belongs to the set F_j^{j-1} of objects that are not present anymore after the—at that moment—current and next blocks at level $j + 1$ are completed. That is, p is deleted during one of the 2^{j+2} updates of the—at that moment—current and

next blocks at level $j+1$. Since $j \leq i+1$, it follows that the moment that p is deleted, is in one of the—at this moment—current and next blocks at level $i+2$. Hence, p is deleted in one of the next 2^{i+3} updates. This proves that there are at most 2^{i+3} objects p in V_i that satisfy Case 1.

Case 2: After the moment of its insertion, p has moved monotonically from level 2 to level i.

At the start of the current block at level i, object p is at level i. Hence, during the previous block at level i—during Part 2 or 6 of that block—p was built in a structure $DS(N_i^i)$ (because p was in the list F_{i-1}^i). Therefore, either at the start of the previous block at level $i-1$, or at the start of the block at level $i-1$ that precedes this previous block, object p was at level $i-1$. It follows that there are two possible blocks at level $i-1$, such that at the start of such a block, p was at level $i-1$. For each of these two blocks at level $i-1$, there are two possible blocks at level $i-2$, such that at the start of such a block, p was at level $i-2$. Hence, in total there are 2^2 possible blocks, such that at the start of one of these, p was at level $i-2$. Continuing in this way, it follows that there are 2^{i-2} possible blocks at level 2, such that at the start of one of these blocks, p was at level 2.

If p is at level 2 at the start of block b at level 2, and if p moves monotonically to higher levels, then p was inserted during the block at level 2 that immediately precedes block b. So, for this block b, there are at most 4 possible objects p, that satisfy Case 2.

Altogether, there are at most $4 \cdot 2^{i-2} = 2^i$ objects p in V_i that satisfy Case 2.

Case 3: Otherwise, p has never moved from level j to level $j-1$, for any $j \geq 3$, and after its insertion, p has not moved monotonically from level 2 to level i.

Then, at the start of the algorithm, p was contained already at one of the levels $2, 3, \ldots, i$. The data structure can be built using the initialization algorithm of Section 3. (We have to adapt this algorithm slightly, because now there are only levels $2, 3, \ldots, m-1$, and there are lists V_a^b.) If at the start of the algorithm, p was at level j, for some $3 \leq j \leq m-2$, then p will be deleted in update d, for some d such that $j = \lceil \log d \rceil - 1$. There are 2^j such d's, namely $d = 2^j + 1, 2^j + 2, \ldots, 2^{j+1}$. Hence, at the start of the algorithm, there are at most 2^j objects at level j. Similarly, at the start of the algorithm, there are at most 8 objects at level 2.

It follows that there are at most $8 + \sum_{j=3}^{i} 2^j = 2^{i+1}$ objects p that are at one of the levels $2, 3, \ldots, i$, at the start of the algorithm. Hence, there are at most 2^{i+1} objects p in V_i that satisfy Case 3.

To summarize, we have shown that V_i has size at most

$$2^{i+3} + 2^i + 2^{i+1} = 11 \cdot 2^i,$$

if V_i is the set of objects that are at level i, at the start of the current block at level i.

Now let $3 \leq i \leq m-2$. If the first half of the current block at level i is completed, then level i gets a new set $V_i = F_i^{i+1} \cup F_{i-1}^i$. Here, F_i^{i+1} consists of objects that were at level i at the start of the current block at level i, and F_{i-1}^i consists of objects that were at level $i-1$ at the start of the—at that moment—current block at level $i-1$. It follows from the above that

$$|V_i| = |F_i^{i+1}| + |F_{i-1}^i| \leq 11 \cdot 2^i + 11 \cdot 2^{i-1} = \frac{33}{2} \cdot 2^i.$$

We have shown that for $3 \leq i \leq m-2$, the set V_i has—at any moment—size at most

$$\max(11 \cdot 2^i, \frac{33}{2} \cdot 2^i) \leq 18 \cdot 2^i.$$

We are left with the case $i = 2$. We saw that at the start of the current block at level 2, the set V_2 has size at most $11 \cdot 2^2$. During the processing of the current block at level 2, at most 4 objects are inserted in V_2. Hence, at any moment, the size of V_2 is at most $11 \cdot 2^2 + 4 \leq 18 \cdot 2^2$. This proves the lemma. □

Now we are ready to prove the final result:

Theorem 2 *There exists a data structure of size $O(S(n))$, such that the maximum value of the symmetric function f can be maintained under semi-online updates at the cost of*

$$O\left(\frac{P(n)}{n}\log n + Q(n)\log n + (\log n)^2\right)$$

time per update in the worst case.

Proof: Carefully inspecting the update algorithm, we see that at each level $3 \leq i \leq m-1$, we spend an amount of time per update that is bounded by

$$O\left(\log n + P(2^i)/2^i + Q(2^i)\right).$$

At level 2, we spend an amount of time per update that is bounded by

$$O\left(\log n + \sum_{i=2}^{m-1} Q(2^i)\right).$$

Finally, we spend $O(\log n)$ time to compute the maximum of the array A. It follows that the entire update algorithm spends an amount of time per update that is bounded by

$$\begin{aligned} & O\left(\log n + \sum_{i=2}^{m-1} Q(2^i) + \sum_{i=3}^{m-1}\left(\log n + P(2^i)/2^i + Q(2^i)\right)\right) \qquad (2) \\ = \; & O\left(Q(n)\log n + \sum_{i=3}^{m-1}\left(\log n + P(n)/n + Q(n)\right)\right) \\ = \; & O\left(Q(n)\log n + (P(n)/n)\log n + (\log n)^2\right). \end{aligned}$$

Just as in the proof of Theorem 1, it follows that, at any moment, the current number of objects is $\Theta(n)$. Therefore, each update takes $O((P(N)/N)\log N + Q(N)\log N + (\log N)^2)$ time in the worst case, if N is the current number of objects. Using Lemma 7, we can prove the bound on the size of the data structure in the same way as in Theorem 1. □

Remark: In the above estimates, we use the facts that $Q(n)$ and $P(n)/n$ are non-decreasing. If $P(n)/n^{1+\varepsilon}$ and/or $Q(n)/n^{\delta}$ are non-decreasing for some $\varepsilon, \delta > 0$, these estimates are too pessimistic. In this case, a smaller upper bound for Equation (2), and hence a smaller update time bound in Theorem 2, can be derived:

$$\sum_{i=2}^{m-1} Q(2^i) = \sum_{i=2}^{m-1} \frac{Q(2^i)}{2^{i\delta}}\, 2^{i\delta} \leq \frac{Q(n)}{n^{\delta}} \sum_{i=2}^{m-1} 2^{i\delta} = O(Q(n)).$$

Similarly, we get

$$\sum_{i=3}^{m-1} P(2^i)/2^i = O(P(n)/n).$$

In Section 5, we will see an example, where this case occurs.

5 Applications

Dobkin and Suri give several applications of their algorithm. In this section, we repeat two of these.

Let V be a set of n points in the plane. Suppose we want to maintain the *diameter* of V, i.e., the maximal distance between any two points in V. For planar points p and q, let $f(p,q)$ be the euclidean distance between these two points. Then the maximal value of f over all pairs of points in V is equal to the diameter of V. In order to apply Theorem 2, we need a data structure in which queries of the form "compute the value of $f(p,V)$" can be solved efficiently. Such a query is equal to "compute the maximal distance between p and any point in V", i.e., we want the furthest neighbor of p in V.

So let DS be the data structure that stores the furthest-point Voronoi diagram of V. Using point location techniques, this structure can be implemented using $O(n)$ space, such that furthest-neighbor queries can be solved in $O(\log n)$ time. Also, this structure can be built in $O(n \log n)$ time. (See e.g. Kirkpatrick [9], Edelsbrunner [6].) Applying Theorem 2, leads to:

Theorem 3 *There exists a data structure of size $O(n)$, such that the diameter of a set of n points in the plane can be maintained under semi-online updates at the cost of $O((\log n)^2)$ time per update in the worst case.*

Next consider the same problem, but now in three dimensions. Dobkin and Suri show that there exists a data structure for the furthest neighbor searching problem of size $O(n^{3/2} \log n)$, that can be built in $O(n^{3/2} \log n)$ time, and in which queries can be solved in $O(\sqrt{n} \log n)$ time. In this case, the remark made after Theorem 2 applies, and we get:

Theorem 4 *There exists a data structure of size $O(n^{3/2} \log n)$, such that the diameter of a set of n points in three-dimensional space can be maintained under semi-online updates at the cost of $O(\sqrt{n} \log n)$ time per update in the worst case.*

Remark: Dobkin and Suri did not notice that in this case, the upper bounds of Theorems 1 and 2 are too pessimistic. They obtain an amortized update time of $O(\sqrt{n}(\log n)^2)$ for maintaining the diameter in three dimensions. In fact, for many more problems considered in [5], the remark made after Theorem 2 applies. In all these cases, the amortized update times given there can be decreased.

As another example, suppose we want to maintain the *closest pair* of a set V of n points in the plane. For p and q points in the plane, let $f(p,q)$ be the euclidean distance between these two points. Then the pair of points that minimizes the value of f over all pairs of *distinct* points in V is equal to the closest pair of V. We apply the previous results, where all occurrences of "max" are replaced by "min".

In this case, we need a data structure in which nearest neighbor queries—i.e., given a point p, compute the nearest neighbor of p in V—can be solved. So let DS be the closest-point Voronoi diagram of V. This structure can be implemented in $O(n)$ space, such that nearest neighbor queries can be solved in $O(\log n)$ time. Also, this structure can be built in $O(n \log n)$ time. (See [9,6].)

Theorem 5 *There exists a data structure of size $O(n)$, such that the closest pair of a set of n points in the plane can be maintained under semi-online updates at the cost of $O((\log n)^2)$ time per update in the worst case.*

Proof: If we apply the algorithm of Section 4, we maintain the value of $\min_{p,q \in V} f(p,q)$. But this value is equal to zero because we are allowed to take p and q equal. So we have to adapt the algorithm in order to correctly maintain the closest pair. (It seems that this has been overlooked by Dobkin and Suri.)

During the algorithm, we compute values $f(p, N_i^i)$, where p ranges over all points that are in the lists $V_i^2, \ldots, V_i^i$, or that are in the lists $N_i^i, V_i^2, \ldots, V_i^{i-1}$. (See Parts 3 and 7 of level i.) If p is in one of the lists V_i^j, where $2 \leq j \leq i-2$, then $f(p, N_i^i)$ is equal to the nearest neighbor of p in N_i^i, and this nearest neighbor is not equal to p. If p is in one of N_i^i, V_i^{i-1} and V_i^i, the nearest neighbor of p in N_i^i can be p itself. Therefore, in Parts 3 and 7, we only compute $f(p, N_i^i)$ for p in V_i^j, where $j = 2, \ldots i-2$. The closest pairs between points in $V_i^{i-1} \cup V_i^i$ (resp. $N_i^i \cup V_i^{i-1}$) and N_i^i are computed during Part 2 (resp. 6) of level i. Note that this can be done in $O(a \log a)$ time, if a is the cardinality of $V_i^{i-1} \cup V_i^i$ (resp. $N_i^i \cup V_i^{i-1}$). (See Preparata and Shamos [12].) Here, $a = O(2^i)$. So in Parts 2 and 6, we spend an extra amount of $O(2^i \log n)$ time. Since we spend already the same amount of time, the time complexity increases by at most a constant factor. □

6 Decomposable searching problems

In a searching problem, we are given a set V of objects and a *query object* p, and we are asked some question $f(p, V)$—the query—about p and V. For example, in the *nearest neighbor searching problem*, V is a set of planar points. In a query, we get a planar point p, and we are asked to find a point in V that minimizes the distance to p. In this way, we can view a searching problem as a function $f : T_1 \times P(T_2) \to T_3$.

Definition 3 (Bentley [2]) *A searching problem $f : T_1 \times P(T_2) \to T_3$ is called* decomposable, *if there is a function $\square : T_3 \times T_3 \to T_3$, such that for any partition $V = A \cup B$ of any subset of T_2, and for any query object p in T_1, we have*

$$f(p, V) = \square(f(p, A), f(p, B)),$$

where the function $\square$ can be computed in constant time.

For example, the nearest neighbor searching problem is decomposable, because the nearest neighbor of p in the set $V = A \cup B$ can be computed in constant time from the nearest neighbors of p in A and B, using the function $\square = \min$.

The techniques of the preceding sections can also be applied to decomposable searching problems. In fact, for this case, the algorithms become easier. Let f be a decomposable searching problem, and let DS be a (static) data structure of size $S(n)$, that can be built in $P(n)$ time, such that queries "compute $f(p, V)$" can be solved in $Q(n)$ time. Then we make the following data structure that is capable of handling semi-online updates:

Again, the set V is partitioned into $O(\log n)$ subsets V_i of size $O(2^i)$, where V_i contains objects that are still present after the current block at level i is completed. The set V_i with the smallest index i contains all other objects. Each set V_i is stored in a data structure DS_i and in a list V_i^i.

So this structure is essentially the same one as that of Sections 3 and 4, except that now we do not need the lists containing the function-values and the array A.

Theorem 6 *For the decomposable searching problem f, there exists a data structure of size $O(S(n))$, such that semi-online updates can be performed at the cost of $O((P(n)/n) \log n)$ time in the worst case, and such that queries can be performed in $O(Q(n) \log n)$ time.*

Proof: In the above structure, we maintain a collection of data structures, the objects of which partition the set V. A query is performed by querying each structure DS_i separately, and by combining the answers using the function $\square$. Since there are $O(\log n)$ such structures, the query time is bounded by $O(Q(n)\log n)$. Semi-online updates are processed as in Section 4. Of course, the algorithm is adapted to the above data structure. The bounds on the size of the data structure and the update time follow from Theorem 2. Note that there are no terms $Q(n)\log n$ and $(\log n)^2$ in the update time, because these terms bound the time needed to update the various function-values in Section 4. In the present section, there are no function-values. $\square$

Remark: Suppose that we process a sequence of semi-online updates in which no deletions occur. Then, the amortized version of the above algorithm is the same one as Bentley's logarithmic method for decomposable searching problems. Also, the worst-case version of the algorithm is similar to the worst-case version of Bentley's algorithm. (See [2,3,11] for Bentley's method.)

As an application of this theorem, consider the planar nearest neighbor searching problem. We quoted already in Section 5, that there exists a static data structure for this problem of size $O(n)$, that can be built in $O(n\log n)$ time, such that queries can be answered in $O(\log n)$ time. Note that now the problem we had with the closest pair problem—namely of computing the nearest neighbor of p in set V' if $p \in V'$—does not occur, because we do not have to maintain the function values. Applying Theorem 6, leads to the following:

Theorem 7 *For the planar nearest neighbor searching problem, there exists a data structure of size $O(n)$ such that semi-online updates can be performed at the cost of $O((\log n)^2)$ time in the worst case, and such that queries can be answered in $O((\log n)^2)$ time.*

For other examples of decomposable searching problems for which Theorem 6 gives efficient data structures that can handle semi-online updates, see [3,5,11].

7 Decomposable set problems

In this section, we give a new application of the algorithms of the preceding sections.

In a set problem, we are given a set V of objects, and we are asked some question about this set. For example, the problems we considered in Section 5, are set problems. We can consider a set problem as a mapping $f : P(T_1) \rightarrow T_2$. Overmars [10] studied set problems the answers of which can be merged efficiently, provided these answers correspond to sets that are separated. Here, we introduce set problems, such that their answers can always be merged.

Definition 4 *A set problem $f : P(T_1) \rightarrow T_2$ is called $C(n)$-*decomposable*, if there is a function $\square : T_2 \times T_2 \rightarrow T_2$, such that for any partition $V = A \cup B$ of any subset of T_1, we have*

$$f(V) = \square(f(A), f(B)),$$

where the function $\square$ can be computed in $C(n)$ time, if n is the cardinality of V.

As an example, the problem of constructing the Voronoi diagram of a set of n planar points, is $O(n)$-decomposable: Kirkpatrick [8] and Chazelle [4] have shown that two arbitrary Voronoi diagrams can be merged in linear time. As another example, the problem

of computing the three-dimensional convex hull of n points, is $O(n)$-decomposable, as was shown by Chazelle [4].

Also for decomposable set problems, the techniques of the preceding sections apply. Let f be a $C(n)$-decomposable set problem, and let V be a set of cardinality n, for which we want to maintain the answer $f(V)$. Using divide-and-conquer, this answer $f(V)$ can be computed in time

$$P(n) := \sum_{i=0}^{\log n} 2^i C(n/2^i).$$

The following data structure maintains the answer $f(V)$ under semi-online updates:

As before, the set V is partitioned into $O(\log n)$ sets V_i of size $O(2^i)$, where V_i contains objects that are still present after the current block at level i is completed. The set V_i with the smallest index i contains all other objects. Each set V_i is stored in a list V_i^i. Also, for each set V_i, there is an answer $f(V_i)$ of this set to the set problem. Finally, there is the complete answer $f(V)$ to the set problem.

Again, this structure is essentially the same one as that of Sections 3 and 4.

Theorem 8 *Let f be a $C(n)$-decomposable set problem, and let $A(n)$ be the maximal size of any answer to this problem for a set of n objects. Then there exists a data structure of size $O(n + \sum_{i=0}^{\log n} A(2^i))$, that maintains the answer $f(V)$, at the cost of*

$$O\left(\frac{P(n)}{n}\log n + \sum_{i=0}^{\log n} C(2^i)\right)$$

time per semi-online update in the worst case.

Proof: Semi-online updates are performed on the lists V_i^i and the partial answers $f(V_i)$ as in Section 6. Afterwards, we merge all answers $f(V_i)$ to obtain the final answer $f(V)$.

The first part of this update algorithm takes $O((P(n)/n)\log n)$ time. The merging of the answers takes $\sum_{i=0}^{\log n} C(2^i)$ time. The bound on the size of the data structure follows easily. □

We mentioned above that the Voronoi diagram problem in two dimensions is $O(n)$-decomposable. For this problem, we have $A(n) = O(n)$ and $P(n) = O(n \log n)$. Applying Theorem 8 gives:

Theorem 9 *There exists a data structure of size $O(n)$, that maintains the Voronoi diagram of a set of n points in the plane, at the cost of $O(n)$ time per semi-online update in the worst case.*

Aggarwal et al. [1] have shown that arbitrary deletions in a Voronoi diagram can be performed in linear time. It follows from the definition of $O(n)$-decomposability, that insertions in a Voronoi diagram also take linear time. Therefore, the result of the above theorem even holds for arbitrary updates. So the result of Theorem 9 is not new. It follows, however, from a general technique, and it gives an alternative data structure for this problem.

Another interesting example is the problem of constructing the convex hull of a set of n points in three-dimensional space. We mentioned that this problem is also $O(n)$-decomposable. Again, $A(n) = O(n)$ and $P(n) = O(n \log n)$. Applying Theorem 8 leads to:

Theorem 10 *There exists a data structure of size $O(n)$, that maintains the convex hull of a set of n points in three-dimensional space, at the cost of $O(n)$ time per semi-online update in the worst case.*

At present, the best fully dynamic data structure that maintains the convex hull of n points in three dimensions, has size $O(n \log\log n)$ and a worst-case update time of $O(n)$. (See Overmars [10,11].) So the result of Theorem 10 improves upon the space requirement, if we restrict ourselves to semi-online updates.

8 Concluding remarks

We have given an alternative description of Dobkin and Suri's algorithm to maintain the maximum of a symmetric bivariate function. The analysis of the performances of the algorithm is more straightforward than originally given by them. We have also turned the amortized update time bound into a worst-case time bound. There are many interesting examples of symmetric functions, to which the algorithms can be applied. For example, the diameter and the closest pair in a set of n planar points can be maintained under semi-online updates at the cost of $O((\log n)^2)$ time in the worst case. Simplified versions of the algorithms can be used for dynamizing static data structures that solve decomposable searching problems. The amortized simplified algorithm is an extension of Bentley's logarithmic method: if we apply the algorithm in case there are only insertions and no deletions, we get Bentley's algorithm. The worst-case algorithm applied in case there are no deletions is also similar to the worst-case version of Bentley's algorithm. As a new application, we have adapted the algorithm, such that the answer to a decomposable set problem can be maintained efficiently under semi-online updates.

There remain some open problems. In the logarithmic method for decomposable searching problems, an insertion is performed as follows: We merge structures that store $2^0, 2^1, \ldots, 2^i$ objects, together with the object to be inserted, in a new structure representing 2^{i+1} objects. The time needed for this operation is in general bounded by $O(P(2^{i+1}))$, where $P(\cdot)$ is the time needed to build the data structure. For some problems, however, this merging can be done faster. Consider, for example, the case of Voronoi diagrams. It takes $O(n \log n)$ time to build a Voronoi diagram from scratch. It was shown by Kirkpatrick [8] and Chazelle [4], that two Voronoi diagrams can be merged in time that is linear in the total number of points. Therefore, Voronoi diagrams of size $2^0, \ldots, 2^i$ can be merged in $O(2^i)$ time. Hence, we may apply Bentley's theorem with $P(n) = n$, instead of $P(n) = n \log n$. It follows that there exists a data structure for the nearest neighbor searching problem in which points can be inserted in $O(\log n)$ time, such that queries can be solved in $O((\log n)^2)$ time. It is an open problem whether for semi-online updates the same complexity can be achieved.

Of course, the basic open problem is whether the techniques for semi-online updates can be extended to online updates.

References

[1] A. Aggarwal, L.J. Guibas, J. Saxe and P.W. Shor. *A linear-time algorithm for computing the Voronoi diagram of a convex polygon.* Discrete Comput. Geom. **4** (1989), pp. 591-604.

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

[3] J.L. Bentley and J.B. Saxe. *Decomposable searching problems I: static to dynamic transformations.* J. of Algorithms **1** (1980), pp. 301-358.

[4] B. Chazelle. *An optimal algorithm for intersecting three-dimensional convex polyhedra.* Proc. 30-th Annual IEEE Symp. on Foundations of Computer Science, 1989, pp. 586-591.

[5] D. Dobkin and S. Suri. *Dynamically computing the maxima of decomposable functions, with applications.* Proc. 30-th Annual IEEE Symp. on Foundations of Computer Science, 1989, pp. 488-493.

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

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

[8] D.G. Kirkpatrick. *Efficient computations of continuous skeletons.* Proc. 20-th Annual IEEE Symp. on Foundations of Computer Science, 1979, pp. 18-27.

[9] D.G. Kirkpatrick. *Optimal search in planar subdivisions.* SIAM J. Comput. **12** (1983), pp. 28-35.

[10] M.H. Overmars. *Dynamization of order decomposable set problems.* J. of Algorithms **2** (1981), pp. 245-260.

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

[12] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.