Algorithms for Semi-Online Updates on Decomposable Problems

Michiel Smid[1]

1 Introduction

Let T be a set of objects, and let $f : T \times T \to \mathcal{R}$ be a symmetric function, i.e. $f(x,y) = f(y,x)$. The problem to be studied is the following: given $V \subseteq T$, maintain the maximal value of f w.r.t. V, when objects are inserted and deleted in V. (Of course, also the minimal value can be considered.) We introduce some notations. If $p \in T$, and if $V, A, B \subseteq T$, then

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

Such symmetric functions often appear in computational geometry. For example, let $T = \mathcal{R}^2$. For p and q planar points, let $f(p,q)$ be the distance between p and q. Then the maximum of f 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 extrema of symmetric functions are the closest pair in a point set, the minimal separation between rectangles, the largest or smallest rectangle determined by points, etc.

In general, it is difficult to maintain the maximum of f under arbitrary updates. For example, no algorithm is known that maintains the diameter of a point set in less than linear time per update. Therefore, we consider a restricted class of updates, introduced by Dobkin and Suri (see FOCS 89):

A sequence of updates is called *semi-online*, if the insertions are on-line, but with each inserted object, we are told how many updates from the moment of insertion, the object will be deleted.

Dobkin and Suri give a general technique to design data structures that can handle such updates. They conjecture that the amortized complexity of their algorithm 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.

In the full paper, the algorithms are adapted to decomposable searching problems and decomposable set problems. This leads to efficient algorithms for performing semi-online updates on such problems as the post-office problem, maintaining convex hulls, Voronoi diagrams, etc.

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

$$f(p, V_1 \cup \ldots \cup V_m) = \max(f(p,V_1), \ldots, f(p,V_m)), \text{ and}$$

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

2 The amortized algorithm

We give an alternative description of Dobkin and Suri's algorithm. This description, and the analysis of the algorithm, are easier to understand. Also, the algorithm as presented is a good basis for the worst-case algorithm. Let $f : T \times T \to \mathcal{R}$ be a symmetric function, and let $V \subseteq T$. Assume we have a data structure $DS(V)$ that stores V, such that queries of the form "compute $f(p,V)$ for $p \in T$" can be solved 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, where $n = |V|$.

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

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

[1]Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, West-Germany. This research was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

Let m be a positive integer, and assume that we perform a sequence of 2^m updates. We number these updates from 1 up 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 up to 2^i, the second block consists of update $2^i + 1$ up 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 $j = \lfloor (k-1)/2^i \rfloor$. (So the current block at level i is the unique block at level i to which k belongs.) The* next block at level i *is the block at level i, that follows immediately after the current block at level i.*

The data structure : Let $V \subseteq T$, $|V| = n$. Let $m = \lfloor \log(n/2) \rfloor$. We maintain the maximum of f under a sequence of 2^m 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 maximal 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 maximal value $\max(f)$ of this array. By Lemma 1, we have $\max(f) = \max_{p,q \in V} f(p, q)$.

Initialization: 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. During this walk, select the object (if it exists) that will be deleted in the first update, and put it in V_0. 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 $i = \lceil \log d \rceil - 1$.

Store each set V_i in a list—that we call V_i—and build a data structure $DS(V_i)$ for it. Next, 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, compute the maximum of the array A, and set its value to $\max(f)$.

The update algorithm: Consider the k-th update, where $1 \leq k \leq 2^m$. Write $k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_l}$, 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, insert q in the list V_0, compute the values $f_j(q) := f(q, V_j)$ for $j = i_1 + 1, \ldots, m$, and store these values in a list. If the update is a deletion, then the object to be deleted is stored V_0. Then, delete it from this list.

Now let W be the union of all lists $V_0, \ldots, V_{i_1}$. Use the above initialization algorithm to partition the set W into new subsets $V_0, \ldots, V_{i_1}$, and 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.)

For each object $p \in W$, compute the values of $f_j(p) := f(p, V_j)$, for $j = i, \ldots, i_1$. Here, i is such that p is in (the new) V_i. 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. Next, compute the maximum of the values $f_j(p)$, $i \leq j \leq m$, and set its value to $\max(p)$.

For $i = 0, \ldots, i_1$, compute the maximum of all values $\max(p)$, where p ranges over all objects in V_i, and store the result in the array-entry $A(i)$.

Finally, compute the maximum of the array $A(0:m)$, and 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.

Proof of Theorem 1: It takes $O(P(n)+nQ(n))$ time to build the data structure. An update, performed as described above, takes an amount of time that is bounded by (we write $i=i_1$) $O(Q(n)\log n + P(2^i) + 2^iQ(n) + 2^i\log n)$. 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. Then, $k_i = 2^{m-i-1}$ for $0 \le i < m$, and $k_m = 1$. Hence, the total amount of time needed to perform 2^m updates is bounded above by

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

Dividing by $2^m = \Theta(n)$ gives the amortised complexity. After this sequence of 2^m updates, we choose a new value of m, and we proceed in the same way. □

3 The worst-case algorithm

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

The data structure is almost the same as before. The differences are as follows: Now, the set V is partitioned into subsets $V_2, V_3, \ldots, V_{m-1}$. For each $3 \le i \le m-1$, V_i consists of objects that are still present after the current block at level i is completed. Furthermore, $V_2 = V \setminus \bigcup_{i=3}^{m-1} V_i$. For each $3 \le i \le m-1$, 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. For $i=2$, there is a list V_2^2 that contains the set V_2. The array A is only defined for $2 \le i \le m-1$. The rest of the data structure is the same as before.

The update algorithm: With each update, we perform an update at all current blocks at levels $2,3,\ldots,m-1$. Afterwards, we compute the maximum of the array A, which is the maximum of the function f. We only describe how a block at level i, for $3 \le i \le m-2$, is processed. We split this block in 8 parts, each of length 2^{i-3}:

Part 1 of level i: This first part consists of updates $1,2,\ldots,2^{i-3}$. 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 $F_i^{i-1} = V_i^i \setminus F_i^{i+1}$. This partitioning takes $O(2^i)$ time. With each update, do an amount of $O(2^i)/2^{i-3} = O(1)$ work.

Comments: Since $F_i^{i-1} \subseteq V_i^i$, 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.

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

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 $O(\log n + P(2^i)/2^i)$ work.

Comments: After Part 4 of level i, the new $DS(N_i^i)$ takes over the role of 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.

Part 3 of level i: The third part consists of the next 2^{i-3} updates. 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 $DS(N_i^i)$. With each update, do $O(Q(2^i))$ work. If the actual update is an insertion of object q, compute $f_i'(q) := f(q, N_i^i)$.

Part 4 of level i: This part consists of the next 2^{i-3} updates. Give each object p in the list N_i^i, a new list $f_i(p) := f_i'(p), f_{i+1}(p), f_{i+2}(p), \ldots, f_{m-1}(p)$. Compute the maximal value of this new list,

which we call $\max(p)$. Compute the maximum $\max_i$ of the values $\max(p)$ over all $p \in N_i^i$. With each update, do an amount of $O(\log n)$ work. If the actual update is an insertion of object q, compute $f_i^i(q) := f(q, N_i^i)$.

After Part 4 of level i: After Part 4, set $V_i := N_i^i$; $V_j^i := N_j^i$, for $j = i, \ldots, m-1$; $DS_i := DS(N_i^i)$; $A(i) := \max_i$; 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. Note that indeed $A(i) = f(V_i, V_i \cup \ldots \cup V_{m-1})$.

Comments: 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. All structures and variables corresponding to this value of i are correct after Part 4.

Part 5 of level i: These are the next 2^{i-3} updates. Nothing happens at level i.

Comments: At the start of Part 6, there are lists F_{i+1}^i and F_{i-1}^i available.

Part 6 of level i: During this part—which consists of the next 2^{i-3} updates—merge the lists H_i, F_{i+1}^i and F_{i-1}^i in a list N_i^i, and build a structure $DS(N_i^i)$ for it. For each $j = i+1, \ldots, m-1$, make a copy N_j^i of the list N_i^i. With each update, do 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. Compute for each 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 $DS(N_i^i)$. With each update, do an amount of $O(Q(2^i))$ work. If the actual update is an insertion of object q, compute $f_i'(q) := f(q, N_i^i)$.

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, all blocks at levels $2, 3, \ldots, i_1$ are completed. Let $j = \min(i_1 + 1, m-1)$. During Part 8, give each object p in N_i^i, a new list $f_i(p) := f_i'(p)$, $f_{i+1}(p) := f_{i+1}'(p)$, $\ldots$, $f_j(p) := f_j'(p)$, $f_{j+1}(p)$, $f_{j+2}(p)$, $\ldots$, $f_{m-1}(p)$. Compute the maximal value of this new list, which we call $\max(p)$. Compute the maximum $\max_i$ of the values $\max(p)$ over all $p \in N_i^i$. With each update, do $O(\log n)$ work. If the current update is an insertion of object q, compute $f_i'(q) := f(q, N_i^i)$.

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

Comments: Each 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. All information corresponding to this value of i is correct.

This concludes the update algorithm for a block at a level $3 \leq i \leq m-2$. The update algorithms for blocks at level 2 and $m-1$ are similar.

After the entire sequence of 2^m updates, the value of m—and hence the number of levels—might have to be changed. After the first block at level $m-1$, we choose a new value for m: Let n_0 be the number of objects, after the first 2^{m-1} updates have been processed. Let $m' := \lfloor \log(n_0/3) \rfloor$. After the entire sequence of 2^m updates, there will be levels $2, 3, \ldots, m'-1$. Since $m-1 \leq m' \leq m+1$, the number of levels decreases by one, does not change, or increases by one.

The blocks at levels $2, 3, \ldots, m-1$ during the final 2^{m-1} updates are processed similarly as before. If $m' = m+1$, we build a new level $m'-1$. If $m' = m-1$, we collapse the levels $m-1 = m'$ and $m-2 = m'-1$ into one new level $m'-1$. Let n' be the number of objects, after 2^m updates have been processed. Since $2^{m'} \leq n'/2 \leq 2^{m'+2}$, we are in the same situation as the one we started with. Therefore, we can proceed performing updates in this way.

Theorem 2 *There exists a data structure of size $O(S(n))$, such that the maximal value of the symmetric function f can be maintained in $O((P(n)/n)\log n + Q(n)\log n + (\log n)^2)$ time per semi-online update, in the worst case.*

Proof: The update time is bounded by

$$Q(n)\log n + \sum_{i=3}^{m-1} \left(P(2^i)/2^i + Q(2^i) + \log n\right) = O\left((P(n)/n)\log n + Q(n)\log n + (\log n)^2\right). \square$$