Property-Preserving Data Reconstruction*

Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu

Dept. Comp. Sci., Princeton University, Princeton NJ 08544, USA
{nailon, chazelle, csesha, dingliu}@cs.princeton.edu

Abstract. We initiate a new line of investigation into *online property-preserving data reconstruction.* Consider a dataset which is assumed to satisfy various (known) structural properties; eg, it may consist of sorted numbers, or points on a manifold, or vectors in a polyhedral cone, or codewords from an error-correcting code. Because of noise and errors, however, an (unknown) fraction of the data is deemed *unsound*, ie, in violation with the expected structural properties. Can one still query into the dataset in an online fashion and be provided data that is *always* sound? In other words, can one design a filter which, when given a query to any item I in the dataset, returns a *sound* item J that, although not necessarily in the dataset, differs from I as infrequently as possible. No preprocessing should be allowed and queries should be answered online.

We consider the case of a monotone function. Specifically, the dataset encodes a function $f : \{1, \ldots, n\} \mapsto \mathbf{R}$ that is at (unknown) distance ε from monotone, meaning that f can—and must—be modified at εn places to become monotone.

Our main result is a randomized filter that can answer any query in $O(\log^2 n \log\log n)$ time while modifying the function f at only $O(\varepsilon n)$ places. The amortized time over n function evaluations is $O(\log n)$. The filter works as stated with probability arbitrarily close to 1. We also provide an alternative filter with $O(\log n)$ worst case query time and $O(\varepsilon n \log n)$ function modifications.

1 Introduction

It is a fact of (computing) life that massive datasets often come laden with varying degrees of reliability. Errors might be inherent to the data acquisition itself (faulty sensors, white/bursty noise, aliasing), or to data processing (roundoff errors, numerical instability, coding bugs), or even to intrinsic uncertainty (think of surveys and poll data). Classical error correction postulates the existence of exact data and uses redundancy to provide recovery mechanisms in the presence of errors. Mesh generation in computer graphics, on the other hand, will often deal with reconstruction mostly on the basis of esthetic criteria, while signal processing might filter out noise by relying on frequency domain models.

* This work was supported in part by NSF grants CCR-998817, 0306283, ARO Grant DAAH04-96-1-0181.

R. Fleischer and G. Trippen (Eds.): ISAAC 2004, LNCS 3341, pp. 15–26, 2004.

In the case of geometric datasets, reconstruction must sometimes seek to enforce structural properties. Early work on geometric robustness [5,9] pointed out the importance of topological consistency. For example, one might want to ensure that the output of an imprecise, error-prone computation of a Voronoi diagram is *still* a Voronoi diagram (albeit that of a slightly perturbed set of points). Geometric algorithm design is notoriously sensitive to structure: Dimensionality, convexity, and monotonicity are features that often impact the design and complexity of geometric algorithms. Consider a computation that requires that the input be a set of points in convex position. If the input is noisy, convexity might be violated and the algorithm might crash. Is there a filter that can be inserted between the algorithm (the client) and the dataset so that: (i) the client is always provided with a point set in convex position; and (ii) the "filtered" data differs as little as possible from the original (noisy) data? In an offline setting, the filter can always go over the entire dataset, compute the "nearest" convex-position point set, and store it as its filtered dataset. This is unrealistic in the presence of massive input size, however, and only online solutions requiring no preprocessing at all can be considered viable. We call this *online property-preserving data reconstruction.* Besides convexity, other properties we might wish to preserve include low dimensionality and angular constraints:

- Consider a dataset consisting of points on a low-dim manifold embedded in very high dimensional space. Obviously, the slighest noise is enough to make the point set full-dimensional. How to "pull back" points to the (unknown) manifold online can be highly nonobvious.
- Angle constraints are of paramount importance in industrial/architectural design. Opposite walls of a building have a habit of being parallel, and no amount of noise and error should violate that property. Again, the design of a suitable filter to enforce such angular constraints online is an interesting open problem.

In this paper we consider one of the simplest possible instances of online property-preserving reconstruction: monotone functions. Sorted lists of numbers are a requirement for all sorts of operations. A binary search, for example, will easily err if the list is not perfectly sorted. In this case of property-preserving data reconstruction, the filter must be able to return a value that is consistent with a sorted list and differs from the original as little as possible. (An immediate application of such a filter is to provide robustness for binary searching in near-sorted lists.)

We formalize the problem. Let $f : \{1, \ldots, n\} \mapsto \mathbf{R}$ be a function at an unknown distance ε from monotonicity, which means that f can (and must) be modified at εn places to become monotone. Figure 1 illustrates the filter in action. To avoid confusion, we use the term "query" to denote interaction between the client and the filter, and "lookup" to denote interaction between the filter and the dataset. Given a query x, the filter generates lookups $a, b, c, \ldots$ to the dataset, from which it receives the values $f(a), f(b), f(c), \ldots$, and then computes a value $g(x)$ such that the function g is monotone and differs from f in at most $c\varepsilon n$ places, for some c (typically constant, but not necessarily so). We note two things.

1. Once the filter returns $g(x)$ for some query x, it commits to this value and must return the same value upon future queries.
2. The filter may choose to follow a multi-round protocol and adaptively generate lookups to the dataset depending on previous results. The function $g(x)$ is defined on the fly, and it can depend on both the queries and on random bits. Therefore, after the first few queries, g might only be defined on a small fraction of the domain. At any point in time, if k distinct x_i's have been queried so far, then querying the remaining x_i's (whether the client does it or not) while honoring past commitments leads to a monotone function close enough to f.

It is natural to measure the performance of the filter with respect to two functions. A $(p(n,\varepsilon), q(n))$-filter performs $O(p(n,\varepsilon))$ lookups per query, and returns a function g that is at a distance of at most $q(n)\varepsilon$ from monotonicity, with high probability. The lookup-per-query guarantee can be either amortized or in worst case (the running times are deterministic). Ideally, we would like $p(n,\varepsilon)$ to depend only on n, and $q(n)$ to be constant. There is a natural tradeoff between p and q: we expect q to decrease as p increases. We will see an example of this in this work.

Theorem 1. *There exists a randomized* $(\log^2 n \log\log n, 2+\delta)$*-filter for any fixed* $\delta > 0$*. with a worst case lookups-per-query guarantee. The amortized lookups-per-query over* n *function evaluations is* $O(\log n)$*. The filter behaves as stated with probability arbitrarily close to* 1*.*

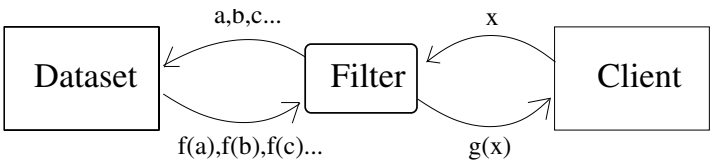

Fig. 1. *The property-preserving reconstruction filter:* g *is sound and differs from* f *in few places*

We also provide an alternative filter with a better lookups-per-query guarantee and a worse distance guarantee.

Theorem 2. *There exists a* $(\log n, O(\log n))$*-filter with a worst case lookups-per-query guarantee.*

It is important to note that, in this work, we think of the client as *adversarial*. That is, the filter's guarantees must hold for all sequences of client queries. However, in some cases it might be useful to assume the client's queries are drawn from some known probability distribution. We will see that the filter can take advantage of this.

Theorem 3. *Assuming the client draws the queries independently, uniformly at random, a* $(1, O(\log n))$*-filter can be devised.*

We are not aware of any previous work on this specific problem. Of course, property testing is by now a well studied area [4,8], with many nontrivial results regarding combinatorial, algebraic, and geometric problems [2,3,7]. More recent work [1, 6] has provided sublinear algorithms for estimating the distance of a function to monotone. We use ideas from [1] in this work.

2 The $(\log^2 n \log\log n, 2+\delta)$-Filter

We use the following notation in what follows. The distance between two functions f_1 and f_2 over the domain $\{1,..,n\}$ is defined as the fractional size of domain points on which they disagree. The function f and the domain size n which are the input to the problem (the dataset in Figure 1) are fixed. We use ε to denote the distance of f from monotonicity, and $\hat{f}$ to denote the monotone function closest to f. So the distance between f and $\hat{f}$ is ε, and $\hat{f}$ minimizes the distance between f and any monotone function. We use g to denote the function outputted by the filter.

2.1 Preliminaries

Proving Theorem 1 requires a few preliminaries, beginning with these definitions:

- *δ-bad and δ-good* : Given $0 < \delta < 1/2$, the integer i is called *δ-bad* if there exists $j > i$ such that

$$\left|\left\{ i \le k \le j \,|\, f(k) < f(i) \right\}\right| \ge (1/2 - \delta)(j - i + 1)$$

  or, similarly, $j < i$ such that

$$\left|\left\{ j \le k \le i \,|\, f(k) > f(i) \right\}\right| \ge (1/2 - \delta)(i - j + 1) \ .$$

  Otherwise the integer i is called *δ-good*.
- *a-light and a-heavy* : Let $\mathcal{D}$ be the joint distribution of m independent 0/1 random variables $x_1, \ldots, x_m$, which can be sampled independently. If $\mathbf{E}[x_i] \le a$ for all i, then $\mathcal{D}$ is called *a-light*; else it is *a-heavy*.

Lemma 1. *(*Ailon et al. **[1])** *Given any fixed $a < b$, if $\mathcal{D}$ is either a-light or b-heavy, then with probability 2/3 we can tell which is the case in $O(m)$ time. If $\mathcal{D}$ is neither, the algorithm returns an arbitrary answer.*

In the following we use the algorithm of Lemma 1 to test, with high probability of success, whether a given integer i is δ-bad or 2δ-good, for any fixed $\delta > 0$. Given an interval $[u, v]$, we define two 0/1 random variables $\alpha[u, v]$ and $\beta[u, v]$: given a random integer $j \in [u, v]$, $\alpha[u, v] = 1$ (resp. $\beta[u, v] = 1$) iff $f(u) > f(j)$ (resp. $f(j) > f(v)$). The algorithm `bad-good-test` (Fig. 2) tests if a given integer i is δ-bad or 2δ-good.

Lemma 2. *Given any fixed $\delta > 0$ and a parameter k, if i is either δ-bad or 2δ-good, then* `bad-good-test` *will tell which is the case in time $O(\log n \log k)$ and with probability at least $1 - 1/k$.*

Proof. If integer i is 2δ-good then the expectation of every $x^{(i)}_{2j-1}$ or $x^{(i)}_{2j}$ defined in `bad-good-test` is at most $1/2 - 2\delta$, and so the distribution $\mathcal{D}$ is $(1/2 - 2\delta)$-light. On the other hand, if i is δ-bad, then there exists some $x^{(i)}_{2j-1}$ or $x^{(i)}_{2j}$ with expectation at least $(1/2 - \delta)/(1 + \delta) \geq 1/2 - 3\delta/2$, and so $\mathcal{D}$ is $(1/2 - 3\delta/2)$-heavy. The algorithm from Lemma 1 distinguishes between $(1/2 - 2\delta)$-light and $(1/2 - 3\delta/2)$-heavy with probability $2/3$ in $O(\log n)$ time. Since we repeat it $c \log k$ times and take a majority vote, a standard Chernoff bound argument shows that `bad-good-test` fails with probability at most $1/k$. □

bad-good-test (f, i, δ, k)

`repeat the following` $c \log k$ `times for a big enough constant` c
  `for each` $1 \leq j \leq (2/\delta) \ln n$
    `define` $x^{(i)}_{2j-1} = \alpha[i, i + (1+\delta)^j]$ `and` $x^{(i)}_{2j} = \beta[i - (1+\delta)^j, i]$
  `let` $\mathcal{D}$ `be the distribution` $(x_1, x_2, \cdots)$
`if majority of above tests output` $(1/2 - 2\delta)$`-light`
  `then output` 2δ`-good`
  `else output` δ`-bad`

Fig. 2. Testing if an integer i is δ-bad or 2δ-good

Lemma 3. *There are at most $(2 + O(\delta))\varepsilon n$ δ-bad integers (*Ailon et al. **[1]**$)$*. Moreover, the monotone function $\hat{f}$ which is closest to f can be assumed to agree with f on δ-good integers.*

Finally, we let $\delta > 0$ denote an arbitrarily small positive real. Choosing a small enough δ will satisfy the distance guarantee of Theorem 1.

2.2 The Algorithm

We now describe the algorithm `monotonize`. Our goal, as described above, is: given a fixed $\delta > 0$, compute a function g online such that: (1) g is monotone; (2) g is $((2 + O(\delta))\varepsilon)$-close to f. Specifically, on query i, `monotonize` computes $g(i)$ in time $O(\log^2 n \log \log n)$. Whenever `monotonize` outputs a value $g(i)$, this value must be recorded to ensure consistency. The procedure will therefore hold an internal data structure that will record past commitments. The data structure can be designed to allow efficient retrievals, but we omit the details because we are mainly interested in the number of f-lookups it performs, and not the cost of other operations.

Given a query i, `monotonize` first checks whether i was committed to in the past, and returns that commitment in that case. If not, more work should be done. In virtue of Lemma 3, `monotonize` tries to keep the f values at δ-good integers and change the values for other queries. We will use `bad-good-test` to decide whether i is bad or good.

Suppose now we decide that i is δ-bad and hence $g(i)$ needs a value that might be different from $f(i)$. Ideally, we would like to find the closest δ-good integers l (to the left of i) and r (to the right of i) and assign $g(i)$ to some value between $f(l)$ and $f(r)$. Because of the sublinear time constraint, we slightly relax this condition. Instead, the idea is to find an interval I_0 around i such that the fraction of 2δ-good integers in I_0 is at least $\Omega(\delta)$, but their fraction in a slightly smaller interval is $O(\delta)$. This ensures that such an interval can be detected through random sampling and that there are not many 2δ-good integers between i and any 2δ-good integer in this interval (a relaxation of the closest condition).

We will search for a good interval within the interval determined by the closest committed values on the left and right of i. Denote this interval by $[l, r]$. Once such a good interval I_0 is found, we try to find a value x that is sandwiched between values of f evaluated at two δ-good points in I_0. Finding x is done in `find-good-value` (Figure 3). We commit to the value x on g restricted to I_0. If no good intervals are found, we spread the value of $g(l)$ on g in the interval $[l+1, r-1]$.

find-good-value (f, I, δ)

```
set L as an empty list
randomly select cδ^{-1} log n integers from I for a big enough constant c.
for each j in random sample
    if bad-good-test (f, j, δ, δ^{-2}) returns ''2δ-good''
        then append f(j) to L
return median value of L
```

Fig. 3. Finding a good value in an interval.

Lemma 4. *The procedure* `find-good-value` *returns, with probability* $1 - 1/n^3$*, a value* y *that is sandwiched between* $f(i_1)$ *and* $f(i_2)$*, where* $i_1, i_2 \in I$ *are* δ*-good. This requires the existence of at least a fraction of* δ 2δ*-good integers in* I*. The running time of* `find-good-value` *is* $O(\log^2 n)$*, for fixed* δ*.*

Proof. The expected number X of δ-bad samples for which `bad-good-test` returns "2δ-good" is at most $c\delta(1-\delta)\log n$, by Lemma 2. The expected total number Y of samples for which `bad-good-test` returns "2δ-good" is at least $c(1-\delta^2)\log n$. The probability that X exceeds $Y/2$ is at most $1/n^3$ if c is chosen large enough, using Chernoff bounds. Therefore, with probability at least

$1 - 1/n^3$, more than half the values that are appended to the list L are the function f evaluated at δ-good points ("good values"). By taking the median of values in L, in such a case, we are guaranteed to get a value sandwiched between two good values. The time bound follows from Lemma 2. □

```
monotonize (f, δ, i)

  if g(i) was already committed to then return g(i)
  if bad-good-test(f, i, δ, n^3) returns ''2δ-good''
      then commit to g(i) = f(i) and return g(i)

  let l be closest committed index on left of i (0 if none)
  let r be closest committed index on right of i (n + 1 if none)
  (*)
  set j_max = ⌊ln(i − l)/ln(1 + δ)⌋ and j_min = 0.
  while j_max − j_min > 1
      set j = ⌊(j_max + j_min)/2⌋, I = [i − (1 + δ)^j, i]
      choose random sample of size cδ^{-1} log n from I, for large c
      for each point i' in sample
          run bad-good-test(f, i', δ, c_1), for large c_1
      if number of "2δ-good" outputs is ≥ (3/2)c log n
          then set j_max = j
          else set j_min = j
  if j_max ≠ ⌊ln(i − l)/ln(1 + δ)⌋
      then set I_l = [i − (1 + δ)^{j_max}, i] and val_l = find-good-value(f, I_l, δ)
      else set val_l = g(l) and I_l = [l + 1, i]
  (**)
  repeat lines (*)...(**) for right side of i, obtaining val_r and I_r

  choose val_l < y < val_r and commit to y on I_l ∪ {i} ∪ I_r
  return y
```

Fig. 4. Computing a monotone function online

To find a good interval, we do a binary search among all the intervals of length $(1+\delta)^j$ $(j = 0, 1, \ldots)$ starting or ending at i, that is, $[i, i + (1+\delta)^j]$ and $[i - (1+\delta)^j, i]$. There are $O(\log n)$ such intervals, and thus the running time is $O(\log \log n)$ times the time spent for each interval. The overall algorithm `monotonize` is shown in Figure 4. The following claim together with a suitable rescaling of δ concludes the proof of the first part of Theorem 1.

Claim. Given any $0 < \delta < \frac{1}{2}$, with probability $1 - 1/n$, `monotonize` computes a monotone function g that is within distance $(2+\delta)\varepsilon$ to f. Given a query i, $g(i)$ is computed online in time $O(\log^2 n \log \log n)$, when δ is assumed to be fixed.

Proof. First we analyze the running time. The `bad-good-test` in line 3 takes $O(\log^2 n)$ time. If the algorithm determines that i is δ-bad, then the while-loops run $O(\log \log n)$ times. In one iteration of the while-loop, the algorithm calls `bad-good-test` $O(\log n)$ times. Each call takes $O(\log n)$ time by Lemma 2. Therefore, the time complexity of the while-loop is $O(\log^2 n \log \log n)$. By Lemma 4, the running time of the call to `find-good-integer` is $O(\log^2 n)$. The time complexity of the algorithm is therefore $O(\log^2 n \log \log n)$.

Let us first look at the while-loop. If I has more than 2δ-fraction of 2δ-good integers, then the number of "2δ-good" outputs is $< \frac{3}{2}c \log n$ with inverse polynomial probability. This can be shown through Chernoff bounds. On the other hand, if I has less than δ-fraction of 2δ-good integers, then the number of "2δ-good" outputs is $> \frac{3}{2}c \log n$ with inverse polynomial probability. Therefore, we can assume that -

- When `find-good-value` is called on interval I_l, I_l has at least a δ-fraction of 2δ-good integers.
- The interval $I_{min} = [i - (1+\delta)^{j_{min}}, i]$ has at most 2δ-fraction of 2δ-good integers.

Both these events hold with probability $1 - 1/n^c$, for some large constant c. These events occur totally at most a polynomial number of times. We can also assume that, during the execution of the algorithm, the first call of `bad-good-test` (in line 3) correctly distinguishes between δ-bad and 2δ-good.

As shown in Lemma 2, this holds with probability $1 - 1/n^3$. This is totally called at most n times. By a union-bound, all of the above events occur with probability $1 - 1/n^d$, for some positive constant d.

To show that the function g is monotone, we first note that if `bad-good-test` outputs "2δ-good" for i (leading to $g(i)$ being set to $f(i)$), then i is not δ-bad. If i is bad, then val_l lies between the value at two δ-good points in I_l. If val_l is assigned to all the values of g in I_l, then g would be monotone with respect to all the values at the δ-good points already committed to. Similarly, val_r can be assigned to the values of g in I_r without disturbing monotonicity. Therefore, since the algorithm assigns some value between val_l and val_r to $I_l \cup \{i\} \cup I_r$, g is remains monotone.

Finally we show that g is within distance $(2+\delta)\varepsilon$ to f. We earlier showed that for $I_{min} = [i - (1+\delta)^{j_{min}}, i]$, the fraction of 2δ-good integers in I_{min} is at most 2δ. Since by the end of the algorithm $j_{max} \leq j_{min} + 1$, the fraction of 2δ-good integers in $I_r = [i, i + (1+\delta)^{j_{max}}]$ (or $I_l = [i - (1+\delta)^{j_{max}}, i]$) is at most 4δ. In other words, each time we make a total of $|I_l \cup \{i\} \cup I_r|$ corrections to f at least a $(1-4\delta)$-fraction of these changes are made on 2δ-bad integers. By Lemma 2.4 in [1], the total number of 2δ-bad integers is at most $(2 + 10\delta)\varepsilon n$. So the total number of changes we made on f is at most $(2 + 10\delta)\varepsilon n/(1 - 4\delta) \leq (2 + c\delta)\varepsilon n$ for some constant c. This concludes the proof. □

2.3 Achieving Logarithmic Amortized Query Time

In this section we show how to modify the algorithm to achieve better amortized query time. The worst case query time for a single query remains the same. We need a technical lemma first.

Lemma 5. *For any $1/2 > \delta > 0$, let i be a δ-bad integer. Let l, r be two δ-good integers such that $l < i < r$. Then there is a witness to i's badness in the interval $[l, r]$.*

Proof. If $f(i) < f(l)$, then we claim that l is a witness to i's badness. In fact, since $f(l)$ and $f(i)$ is a violating pair, it is immediate that at least one of them is 0-bad with respect to the interval $[l, i]$. Since l is δ-good, i must be 0-bad (and hence δ-bad) with respect to $[l, i]$. In this case, l is a witness to i's badness. Similarly, r will be a witness if $f(i) > f(r)$. In the following we assume that $f(l) < f(i) < f(r)$.

Let w be a witness to i's badness. Without loss of generality, assume that $w < i$. If $w \geq l$ then we are done, so let $w < l$. Since i is δ-bad and l is δ-good, we know that: number of violations in $[w, l]$ with respect to l is $< (1/2-\delta)(l-w+1)$; number of violations in $[w, i]$ with respect to i is $\geq (1/2-\delta)(i-w+1)$. We also know that each violation in $[w, l]$ with respect to i is also a violation with respect to l, so the number of violations with respect to i in $[l+1, i]$ is $> (1/2-\delta)(i-l) = (1/2-\delta)(i-(l+1)+1)$. This shows that i has a witness to its badness in $[l+1, i]$. □

The improvement on amortized query time comes from the following strategy: each time the algorithm answers a client query, it also generates a new query by itself and answers that query. This self query is completely independent of all the client queries, and we call it an *oblivious* query.

The oblivious queries are generated based on the balanced binary tree on $[1, n]$. The root of this tree is $\lfloor n/2 \rfloor$. The left subtree of the root corresponds to the interval $[1, \lfloor n/2 \rfloor - 1]$, and similarly the right subtree corresponds to $[\lfloor n/2 \rfloor + 1, n]$. The two subtrees are then defined recursively. This tree is denoted by T.

The oblivious queries are generated according to the following order. We start from the root of T and scan its elements one by one by going down level by level. Within each level we scan from left to right. This defines an ordering of all integers in $[1, n]$ which is the order to make oblivious queries. This ordering ensures that, after the $(2^k - 1)$-th oblivious query, $[1, n]$ is divided by all the oblivious queries into a set of disjoint intervals of length at most $n/2^k$. Each oblivious query is either a δ-good integer itself in which case `monotonize` returns at line 6, or it causes two δ-good integers being outputted (val_l and val_r in `monotonize`). These two δ-good integers lie on the left and right side of the oblivious query, respectively. This shows that after the $(2^k - 1)$-th oblivious query, $[1, n]$ is divided by some δ-good integers into a set of smaller intervals each of length at most $n/2^k$.

Based on Lemma 5, whenever we call `bad-good-test` (in `find-good-integer` or `monotonize`) to test the badness of an integer i, we only need to search for a witness within a smaller interval $[l, r]$ such that l (resp. r) is the closest δ-good integer on the left (resp. right) of i. As explained above, these δ-good integers come as by-products of oblivious queries. This will reduce the running time of

`bad-good-test` to $O(\log n_i \log k)$ (to achieve success probability at least $1-1/k$), where $n_i = r-l+1$. Accordingly, the time spent on binary searching intervals in `monotonize` is reduced to $O(\log \log n_i)$. By the distribution of oblivious queries, for the j-th client query where $2^{k-1} \leq j < 2^k$, the running time of `monotonize` is now $O(\log n \log \frac{n}{2^k} \log \log \frac{n}{2^k})$. The same is true for the j-th oblivious query.

To bound the amortized running time, it suffices to focus on the smallest m such that all n distinct queries appear in the first m queries (including both client and oblivious queries). We can also ignore repetition queries (those that have appeared before) since each one only takes $O(\log n)$ time using standard data structure techniques. Therefore, without loss of generality, we assume that the first n client queries are distinct. The total query time for these n queries is:

$$\sum_{k=1}^{\log n} O(2^{k-1} \log n \log \frac{n}{2^k} \log \log \frac{n}{2^k}) \ .$$

It is simple to verify that this sum is $O(n \log n)$. The following claim concludes the proof of the second part of Theorem 1.

Claim. With probability $1 - 1/n$, `monotonize` computes a monotone function g that is within distance $(2 + O(\delta))\varepsilon$ to f. Each single evaluation of $g(i)$ is computed online in time $O(\log^2 n \log \log n)$, and the amortized query time over the first $m \geq n$ client queries is $O(\log n)$.

3 The $(\log n, O(\log n))$- Filter

We prove Theorem 2. To do this, we define a function g by a random process. The function is determined after some coin flipping done by the algorithm (before handling the client queries). Although the function g is defined after the coin flips, the algorithm doesn't explicitly know it. In order to explicitly calculate g at a point, the algorithm will have to do some f-lookups. Our construction and analysis will upper bound $\mathbf{E}[\text{dist}(f, g)]$ and the amount of work required for explicitly calculating g at a point.

As before, let $\hat{f}$ be a monotone function such that $\text{dist}(f, \hat{f}) = \varepsilon$. Let $B \subseteq [n]$ be the set of points $\{x | f(x) \neq \hat{f}(x)\}$. So $|B| = \varepsilon n$. For simplicity of notation, assume the formal values of $-\infty$ (resp. $+\infty$) of any function on $[n]$ evaluated at 0 (resp. $n+1$).

We build a randomized binary tree $T = \text{build-tree}(1, n)$ with nodes labeled $1..n$, where $\text{build-tree}(a, b)$ is defined as follows:

After constructing the randomized tree T, the function g at point i is defined as follows. If i is the root of the tree, then $g(i) = f(i)$. Otherwise, Let $p_1, .., p_j, i$ denote the labels of the nodes on the path from the root to node i, where p_1 is the root of the tree and p_j is the parent of i. Assume that g was already defined on $p_1, .., p_j$. Let $l = \max(\{0\} \cup \{p_k | p_k < i\})$ and $r = \min(\{n+1\} \cup \{p_k | p_k > i\})$. If $g(l) \leq f(i) \leq g(r)$, then define $g(i) = f(i)$, otherwise define $g(i)$ as an arbitrary value in $[g(l), g(r)]$. The function g is clearly monotone. The number of

build-tree(a, b)

```
if a > b then
    return empty tree
else
    return tree with
      root i chosen uniformly at random in [a, b]
      left subtree build-tree(a, i − 1)
      right subtree build-tree(i + 1, b)
```

f-lookups required for computing $g(i)$ is the length of the path from the root to i. We omit the proof of the following fact.

Lemma 6. *The expected length of this path for any i is at most $O(\log n)$.*

We show that $\mathbf{E}[\text{dist}(f, g)] = O(\varepsilon \log n)$. We first observe that for any i, if $\{p_1, .., p_j, i\} \bigcap B = \emptyset$, then it is guaranteed that $g(i) = f(i)$. Therefore, any i for which $f(i) \neq g(i)$ can be charged to some $b \in B$ on the path from the root to i. The amount of charge on any $b \in B$ is at most the size of the subtree b in T. We omit the proof of the following lemma.

Lemma 7. *The expected size of the subtree rooted at node i in T is $O(\log n)$ for any $i \in [n]$.*

Therefore, the expected total amount of charge is at most $O(|B| \log n) = O(n\varepsilon \log n)$. By Chebyshev's inequality, the total amount of charge is at most $O(n\varepsilon \log n)$ with high probability. The total amount of charge is an upper bound on the distance between f and g. This proves Theorem 2, except for the fact that the lookups-per-query guarantee is only on expectation, and not worse case (due to Lemma 6). We note without proof that the random tree T can be constructed as a balanced binary tree, so that Lemma 6 is unnecessary, and Lemma 7 is still true. So we get a worst case (instead of expected) guarantee of $O(\log n)$ on the length of the path from the root to i (and hence on the number of f-lookups per client query). This concludes the proof of Theorem 2.

To prove Theorem 3, where the client queries are assumed to be uniformly and independently chosen in $[n]$, we observe that the choices the client makes can be used to build T. More precisely, we can build T on the fly, as follows: The root r of T is the first client query. The left child of r is the first client query in the interval $[1, r-1]$, and the right child of r is the first client query in the interval $[r+1, n]$. In general, the root of any subtree in T is the first client query in the interval corresponding to that subtree. Clearly, this results in a tree T drawn from the same probability distribution as in build-tree$(1, n)$. So we still have Lemma 7, guaranteeing the upper bound on the expected distance between

g and f. But now we observe that for any new client query i, the path from the root of T to i (excluding i) was already queried, so we need only one more f-lookup, namely $f(i)$. This concludes the proof of Theorem 3. □

References

1. Ailon, N., Chazelle, B., Comandur, S., Liu, D.: Estimating the distance to a monotone Function. Proc. 8th RANDOM, 2004
2. Fischer, E.: The art of uninformed decisions: A primer to property testing. Bulletin of EATCS, 75: 97-126, 2001
3. Goldreich, O.: Combinatorial property testing - A survey. "Randomization Methods in Algorithm Design," 45-60, 1998
4. Goldreich, O., Goldwasser, S., Ron, D.: Property testing and its connection to learning and approximation. J. ACM 45 (1998), 653–750
5. Hoffmann, C.M., Hopcroft, J.E., Karasick, M.S.: Towards implementing robust geometric computations. Proc. 4th SOCG (1988), 106-117.
6. Parnas, M., Ron, D., Rubinfeld, R.: Tolerant property testing and distance approximation. ECCC 2004
7. Ron, D.: Property testing. "Handbook on Randomization," Volume II, 597-649, 2001
8. Rubinfeld, R., Sudan, M.: Robust characterization of polynomials with applications to program testing. SIAM J. Comput. 25 (1996), 647–668
9. Salesin, S., Stolfi, J., Guibas, L.J.: Epsilon geometry: building robust algorithms from imprecise computations. Proc. 5th SOCG (1988), 208-217