6.5240 Sublinear Time Algorithms	October 13, 2024

Lecture 13

Lecturer: Ronitt Rubinfeld *Scribe: Albert Tam*

Today, we consider **monotonicity testing**, which was one of the first properties to be studied in property testing.

1 Warmup: testing sortedness of a list

Definition 1 *A list of size n is ε-close to sorted, if we can delete at most εn values from the list.*

We want an algorithm that runs in sublinear time and:

- Passes all sorted lists
- Rejects ε-far lists with probability at least $3/4$.

A first stab at an algorithm might work by sampling elements, and checking if they are less than or equal to their neighbor on the right. But if we have a list of the form

$$1, 2, \ldots, i, 1, 2, \ldots, n - i,$$

then we will need a linear number of samples to find the "breakpoint" in the middle, and identify that this list is ε-far!

Another approach is to sample multiple random entries at a time, and check if they are all in the right order. That would fix the issue we had with our bad input above: we'd end up sampling some entries to the left of the breakpoint and some entries to the right of the breakpoint, and find that they are not in the right order. But consider the following bad input example:

$$2, 1, 4, 3, 6, 5, \ldots,$$

which consists of $n/2$ pairs of decreasing elements. This is ε-far from being sorted, since the largest monotone sequence has size $n/2$—but to identify this, we would need to sample two elements from the same pair, which requires $\Omega(\sqrt{n})$ samples. (It turns out that $\Theta(\sqrt{n})$ samples indeed suffices.)

But we want to test sortedness in fewer samples. How low can we go? It turns out that any property tester for sortedness must make $\Omega(\log n)$ samples—discussed in class. We will show that there indeed exists a property tester for sortedness that takes $O(\log n/\varepsilon)$ samples.

2 A property tester for sortedness

2.1 The algorithm

Before introducing the algorithm, we will first simplify the problem by making the list distinct. This is a common trick in parallel algorithms. We can do this by treating each element y_i as the pair (y_i, i), which induces a strict ordering on our elements.

Our algorithm is based on binary search. We repeat the following subroutine $O(1/\varepsilon)$ times on our input y:

1. Sample a random index i.
2. Binary search for the value of $y[i]$.
   (a) If we find inconsistencies, then **FAIL**
   (b) If y_i is not at location i, then **FAIL**

If we pass all of the subroutines, then we **ACCEPT**.

2.2 Analysis

The algorithm clearly has $O(\log n/\varepsilon)$ query and time complexity.

We'll now show that it is a valid property tester. We will call an index i *good* if the binary search for $y[i]$ works and does not reveal any inconsistencies. Our algorithm fails an array if it finds an index that is not good.

If the list is sorted, all i's are good. (This is why we need distinct elements: otherwise, the binary search might not return the index i for the value $y[i]$, and we might fail a sorted list.)

Now, suppose that our list is likely to pass the test. Then fewer than ε of the indices can be bad, so we must have $\geq (1-\varepsilon)n$ good indices.

Claim 2 *The good elements of y form an increasing sequence.*

Proof Suppose that i and j are both good indices ($i < j$). Consider the binary search for both y_i and y_j, and let y_k be the last midpoint that was considered in both searches. Then i must have been to the left of k, so $y_i < y_k$, and j was to the right of k, so $y_j > y_k$. This implies $y_i < y_j$. ■

Therefore, we can delete all the elements that are *not* good from y to create a sorted list. There are at most εn of these elements, so we can conclude that if our original list is likely to pass the test, then it is ε-close to being sorted.

3 Monotonicity on the boolean cube

3.1 Intro

We now move onto a new domain: testing the monotonicity of functions on the boolean cube.

The *boolean cube* comprises the set of all n-bit strings, denoted $\{0,1\}^n$. We can think of the boolean cube as a hypercube, where each vertex is an n-bit string, and two vertices are connected by an edge if they differ in exactly one bit. In particular, these edges are directed, pointing from the string with the 0 to the string with the 1 (e.g. $000 \to 100$).

This structure induces a partial order on the vertices of the boolean cube, defined by the relation $x \prec y$ if there is a directed path from x to y, i.e. $x_i \leq y_i$ for all i. The partial order makes analyzing this setting more complicated than just testing for sortedness.

Instead of testing for sortedness, we will test for *monotonicity* of boolean functions on the cube.

Definition 3 *A function $f : \{0,1\}^n \to \{0,1\}$ is **monotone** if $f(x) \leq f(y)$ whenever $x \prec y$.*

Naturally, a function f is **ε-close to monotone** if we can change at most $\varepsilon 2^n$ values of f to make it monotone.

Throughout our analysis, we will make reference to edges that are "violated" by a function f:

Definition 4 *An edge $x \to y$ is **violated** by a function f if $f(x) > f(y)$.*

3.2 Testing monotonicity

3.2.1 The algorithm

Today, we're going to look at a $O(n/\varepsilon)$ property tester for monotonicity of a function. (Recall that the size of our input is 2^n, so this is logarithmic in input size.)

(The current best testers in the literature make $\Theta(\sqrt{n}/\varepsilon^2)$ queries for nonadaptive tests, and $\Omega(n^{1/3})$ queries for adaptive tests.)

Based on the idea of violating edges, we're going to propose a very simple tester, which makes $O(n/\varepsilon)$ queries:

1. Sample $2n/\varepsilon$ edges (x, y) from the hypercube.

2. **Fail** if any edge (x, y) is violated, otherwise **accept**.

(As a sidenote: you can sample an edge uniformly at random by picking a random string and flipping a 0 bit to a 1.)

3.2.2 Analysis

If f is monotone, then there are no violating edges and this test always passes.

Now, suppose that f is likely to pass the test. This means that $\leq \varepsilon/n$ of the edges are violating; otherwise, we will sample a violating edge w.h.p.

How many violating edges does this give us? Each of the 2^n vertices has degree n (one edge for each bit), so there must be $n2^{n-1}$ edges in total. Therefore, there are at most $\varepsilon/n \cdot n2^{n-1} = \varepsilon 2^{n-1}$ violating edges for f.

We will make use of the following lemma:

Lemma 5 (Repair lemma) *Let $V(f)$ be the number of edges violated by f. Any function f on the boolean cube can be made monotone by changing $\leq 2V(f)$ values.*

By this lemma, we can make fewer than $\varepsilon 2^n$ changes to f to make it monotone. Therefore, if f is likely to pass, then f is ε-close to being monotone, and our tester is valid!

Now, we'll prove the repair lemma.

Proof The idea here is to "repair" violating edges one by one, by changing them from $1 \to 0$ to $0 \to 1$. This means 2 value changes for each violating edge, which will give us the $2V(f)$ bound.

We have to be careful, though, because we don't want to change a value later and get a *new* violated edge! To fix this, we will repair edges in one dimension at a time. (Here, on the n-dimensional boolean cube, the edges in dimension i are simply all the edges that differ in the ith bit.)

Let's show that this works.

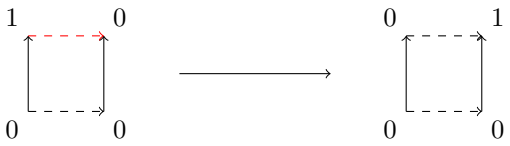
Claim 6 *Let V_j be the number of violated edges in dimension j. Repairing an edge in dimension i does not increase V_j for $j \neq i$.*

Proof Because we only consider two dimensions at a time, it suffices to prove the claim for squares of edges, where two of the edges are in dimension i and two are in dimension j.

If both of the edges in dimension i are violated, like in the figure below, then swapping them both will not affect the edges in dimension j.

If only one of the edges in dimension i is violated (WLOG the top one), then we can do casework on the labels of the bottom edge. The bottom edge could be $0 \to 1$, in which case we have the following:

or the bottom edge could be $0 \to 0$, in which case we have the following:



or the bottom edge could be $1 \to 1$:

In all of these cases, the number of violating edges in dimension j was either unchanged or decreased. ■

By the above claim, we can simply go through each dimension from 1 to n and repair all the violating edges in that dimension. Each repair will change at most 2 values, and we will have repaired all the violating edges. Therefore, we can make at most $2V(f)$ changes to f to make it monotone. ■

It turns out that if we're clever about it, we can get rid of the factor of 2 in the repair lemma, and only need to change $V(f)$ values to make f monotone.