CMPS290A: Sublinear algorithms for graphs **Spring 2017**

Lecture 2: 4/15/17

Lecturer: C. Seshadhri *Scribe: Matthew Gray*

2.1 Monoticity Testing

Definition 2.1 *[GGLRS00] A monotonicity tester for function $f : D \to R$, where D is a partially order set and R is some range with total order, is a randomized Algorithm A with the following properties:*

1. *Given query access to function f and a proximity parameter $\epsilon > 0$*
   - *If f is monotone, A accepts with probability $> \frac{2}{3}$*
   - *If distance of f to monotonicity $> \epsilon$, A rejects with probability $> \frac{2}{3}$*
2. *A makes $o(|D|)$ queries*

Definition 2.2 *A one sided tester accepts monotone functions with probability 1. If distance of f to monotone is greater than ϵ it rejects with probability greater than $\frac{2}{3}$ and will give a pair that violated monotonicity.*

Definition 2.3 *A non-adaptive tester makes all it's queries in advance. It's queries do not depend on the results of the queries preceding them.*

2.1.1 Monotonicity Testing on the Line

A monotonicity tester A on a $f : [n] \to \mathbb{N}$ will act as follows.

1. If f is sorted, A accepts with probability $> \frac{2}{3}$
2. If distance of f to sorted $> \epsilon$ A rejects with probability $> \frac{2}{3}$. i.e.
   If you must change more than ϵ elements of f to make f sorted, it will reject w.p. $\frac{2}{3}$.

Theorem 2.4 *[EKK+] There is a $O(\frac{log(n)}{\epsilon})$-query monotonicity tester. This tester is one sided and non-adaptive*

That a $O(\frac{log(n)}{\epsilon})$-query monotonicity tester is optimal will be covered in future lectures.

An example algorithm would be to make $s = \frac{log(n)}{\epsilon}$ queries and then check that they are in sorted order. If we choose them randomly the average gap between two queries would is $\frac{n}{s}$. This will only catch places where there are areas far apart that violate monotonicity (remember s is much less than n when n is large).

If the function is increasing on a large scale but has small dips all over the place a random distribution will not work. So we will instead see if we can try every gap at once.

This approach is inspired by Binary Search and has two proofs, one is clean, the other is expressive. We'll start with the clean one.

Here's the algorithm

1. repeat $\frac{s}{\epsilon}$ times.
   (a) Pick $x \in [n]$ uniform at random
   (b) Perform Binary Search of $f(x)$ on f
   (c) If you fail to find it successfully REJECT
2. ACCEPT

Lemma 2.5 *If f is at least ϵ far from monotone, then for at lease ϵn elements in $[n]$, binary search fails.*

Suppose Lemma is true. If f is monotone, tester always accepts, because a sorted list will never fail a binary search. If the distance from f to monotonicity is $> \epsilon$, then by the Lemma there are $> \epsilon n$, where $n > 1$ elements where binary search fails. Which means that on a single run of the algorithm it fails with probability at least ϵ. So the probability that the tester accepts after $\frac{s}{\epsilon}$ runs is:

$$Pr[ACCEPT] < (1-\epsilon)^{s/\epsilon}$$
$$lim_{\epsilon \to 0}(1-\epsilon)^{1/\epsilon} = \frac{1}{e}$$
$$Pr[ACCEPT] < (\frac{1}{e})^s = e^{-s} < \frac{1}{3}$$

So when ϵ is small and f is at least ϵ far from monotone it fails with probability greater than $\frac{2}{3}$. We still need the lemma so lets prove it.

Proof: Lemma 2.5 Suppose that $x < y$ but $f(x) > f(y)$. Consider the tree generated by a binary search on the domain. x and y have a least common ancestor z. $x \leq z \leq y$ but $f(x) > f(y)$ therefor $f(x) > f(z)$ or $f(y) < f(z)$. Which means that binary search will go the wrong way and miss either $f(x)$ or $f(y)$.

Define x to be bad if binary search on x fails. Let $\mathcal{B}$ be the set of bad indices. Then $f : [n]|\mathcal{B}$, or f on the line without all bad indices, is monotone. $[n]|\mathcal{B}$ is an increasing subsequence with cardinality $|[n]|\mathcal{B}| \leq (1-\epsilon)n$ when $|\mathcal{B}| \geq \epsilon n$. ■

So that was cool and slick, but doesn't tell you too much about why the tester works. This other proof and algorithm gives you more insight into why monotonicity testers for the line work. Before we get to the algorithm we need some definitions and lemmas.

Definition 2.6 *The violation graph VG_f has vertex cover set $[n]$ and an edge (x, y) for all pairs $x < y, f(x) > f(y)$. Note: this definition works on posets as well as the line.*

Definition 2.7 *ϵ_f is the distance of f to monotonicity.*

Definition 2.8 *A Vertex Cover of graph G is a set of vertexes S such that for all edges(u,v) in G $\{u, v\} \cup S \neq \emptyset$.*

Lemma 2.9 *[FLN+02] The size of the minimum vertex cover of $VG_f = \epsilon_f|D|$ where D is the domain.*

Proof: Lemma 2.9 Take any vertex cover S of the violation graph VG_f. f evaluated on $D|S$ has no violations.

It's possible to "fill in" values in S, so that the resulting function g is monotone. Therefore $|S| \geq \epsilon_f * D$

By our previous argument, the number of bad indices (where binary search fails) form a vertex cover of VG_f thus the number of bad indices $\geq \epsilon_f * |D|$ ■

Here's the algorithm for the monotonicity tester:

1. Repeat $50(\frac{logn}{\epsilon})$ times
   (a) Pick uniform at random x $\in [n]$
   (b) Pick uniform at random k $\in [1, logn]$
   (c) Pick +1 or -1 with probability $\frac{1}{2}$
   (d) Pick uniform at random y $\in [x, x + 2^k]$ if 3 was +1 or $\in [x - 2^k, x]$ if 3 was -1
   (e) If (x,y) is a violation REJECT
2. ACCEPT

Definition 2.10 *x is heavy if there exists $x' > x$ such that at least half of the elements of $[x, x']$ are in violation with x. Analogously for $x' < x$.*

Lemma 2.11 *Heavy vertices form a vertex cover of VG_f*

Proof: Lemma 2.11 For $x < yf(x) > f(y)$ then every mid point $x < z < y$ is in violation with either x or y. Thus either x or y is heavy. ■

The powers of 2 give good approximations for the range x' that has $> \frac{1}{2}$ violations. Let k be the smallest integer k such that $2^k > |x - x'|$. 2^k is less than $2|x - x'|$ because 2^{k-1} was less than $|x - x'|$. So $|x - x'| < 2^k < 2|x - x'|$. So if x is heavy the range from x of 2^k in the correct direction will have at least $2^k \frac{1}{4}$ violations.

So the probability for finding a violation using the above algorithm is the product of making a good guess at each of the 4 steps.

1. Probability of guessing a heavy $x \geq \epsilon$
2. Probability of guessing a correct $k \geq \frac{1}{logn}$
3. Probability of guessing a correct direction $\geq \frac{1}{2}$
4. Probability of guessing a elem that is in violation $\geq \frac{1}{4}$

So the probability for finding a violation on one round is $\geq \frac{\epsilon}{8logn}$

Theorem 2.12 *The tester works.*

Proof: Let us lower bound the probability of rejection if f is ϵ far from monotonic.

For each heavy vertex x, there exists some power of 2 such that either $[x - 2^k, x]$ or $[x, x + 2^k]$ has at least a $\frac{1}{4}$ fraction of violations with x.

There are at least ϵ heavy x's because the heavy x's form a vertex cover of the violation graph. And removing the members of a violation graph would make the function monotonic.

So with the chance of the tester discovering a violation in one round is as shown above to be at least $\frac{\epsilon}{8logn}$, the chance of the algorithm giving a false positive on $50(\frac{logn}{\epsilon})$ runs is $(1 - \frac{\epsilon}{8logn})^{50(\frac{logn}{\epsilon})}$ which is small. ■

[GGLRS00] O. Goldreich, S. Goldwasser, E. Lehman, D. Ron, and A. Samordinsky, Testing Monotonicity, *Combinatorica* (2000), pp. 301-337. vol. 20

[FLN+02] E. Fischer E. Lehman I. Newman S. Raskhodnikova R. Rubinfeld, Monotonicity Testing over General Poset Domains, *Proceedings of the 34th Annual ACM Symposium on the Theory of Computing (STOC)* (2000), pp. 474-483.

[EKK+98] F. ErgnS. Kannan S. KumarR. Rubinfeld M. Viswanathan, Spot-checkers, *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC)* (1998), pp. 259-268