CS 270 Algorithms **Spring 2021**

Lecture 25: Property Testing

Lecturer: Prasad Raghavendra Scribe: Bhaskar Roberts

A **property** is a statement about a given input that can be verified. For example, given a list of numbers, a property that the list might have is that it is sorted in increasing order. Given a graph, a property that the graph might have is that it is triangle-free or that it contains a maximal matching of size s (we'll define maximal matchings later).

In all of these examples, we can check whether the input has the desired property in time polynomial in the input size. For example, we can check if a list of n numbers is sorted in $O(n)$ time.

A logical question to ask next is: can we check whether a list is sorted while reading $o(n)$ bits of the input? The answer is no, if we want an algorithm that is correct on every input. If our algorithm queries $o(n)$ entries, then there are at least 2 entries of the list that will not be checked by the algorithm. The algorithm cannot distinguish between a list that is sorted and a list that is almost sorted except that those two entries are in the wrong order.

Let's make the problem easier: any list we give the algorithm is guaranteed to either be sorted or *far from sorted*. The way we define "far from sorted" depends on the context, but it describes lists that look very different from sorted lists, even to an algorithm that checks $o(n)$ bits of the list. Note that the input will never be close to sorted, such as a list that has just two entries out of order. In the next section, we give an algorithm to solve this problem that uses $O(\log n)$ queries and runtime.

Property testing algorithms solve the kind of problem we just described. They decide whether the input satisfies a property P or is far from satisfying P, and they do so by querying a constant or sublinear number of input bits. Finally, they have the following correctness guarantee:

- If the input satisfies P, then the algorithm accepts with probability $\geq .99$.
- If the input is far from satisfying P, then the algorithm accepts with probability $\leq .01$.

For the rest of lecture, we will cover two algorithms. The first one tests whether a list is sorted, which completes the example we started above. The second one tests the size of a maximal matching of a graph.

25.1 Testing List Sortedness

We want an algorithm that tests whether a list is sorted (in increasing order) or far from sorted. We say that a list is far from sorted if the longest increasing subsequence of the list is much shorter than n.

An **increasing subsequence** of a list is a sequence of elements, not necessarily adjacent to each other, that

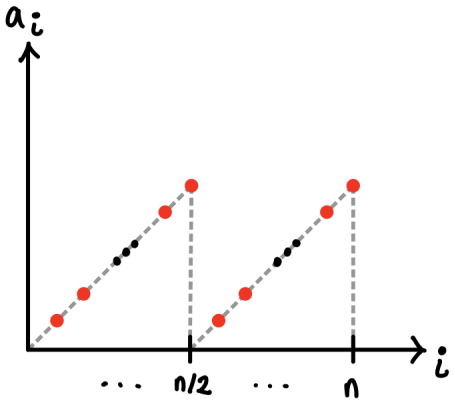

Figure 25.1: The sequence is .5-far from sorted, but our first proposed algorithm accepts it with probability $1 - 1/n$.

are in increasing order. For example, if the list is $A = (2, 1, 4, 3, 8, 6, 7)$, then one increasing subsequence is $S = (2, 4, 6, 7)$. The elements of S must have the same order that they did in A.

A list of n numbers is **ε-far from sorted** if the longest increasing subsequence has length at most $(1 - \varepsilon) \cdot n$. This means that we must remove at least $\varepsilon \cdot n$ of the entries to obtain a sorted subsequence. For example, $A = (2, 1, 4, 3, 8, 6, 7)$ is 3/7-far from sorted because the longest increasing subsequences have length 4.

25.1.1 Problem Statement

In more detail, here is what our property testing algorithm should do. Let A be a list of length n that is either sorted or ε-far from sorted, for some constant $\varepsilon \in (0, 1)$.

If A is sorted, then the algorithm accepts with probability $\geq .99$. If A is ε-far from sorted, then the algorithm accepts with probability $\leq .01$.

Below, we give an algorithm for this problem that uses $O\left(\frac{\log(n)}{\varepsilon}\right)$ time and query bits.

25.1.2 Two attempts

One idea for an algorithm is this: pick a random element in the list and check that it is $\leq$ the element that directly follows it. More precisely, let $A = (a_1, \ldots, a_n)$. Pick $i \in_R [n - 1]$, and check that $a_i \leq a_{i+1}$.

This does not work because there is a sequence, shown in figure 25.1, that is .5-far from sorted that nevertheless has a large probability $(1 - 1/n)$ of being accepted. Even if we repeat our algorithm many times and reject the list if any iteration rejects, we need $O(n)$ repetitions before the probability of acceptance is $\leq .01$.

Here is another idea for an algorithm: pick *two* random elements in the list and check which one is greater.

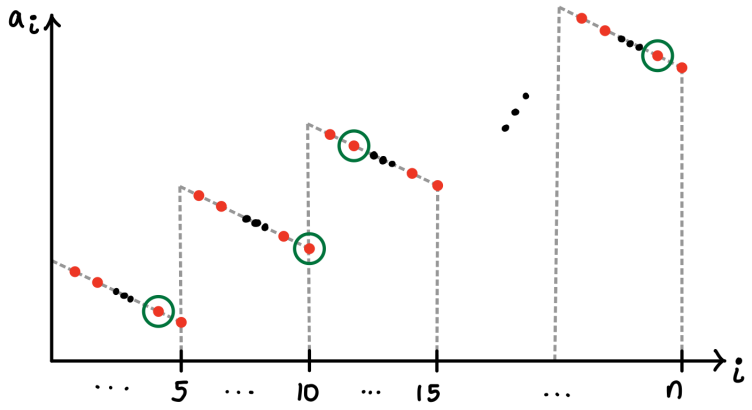

Figure 25.2: This sequence is .8-far from sorted, and our second proposed algorithm accepts it with probability $1 - 5/n$.
The elements circled in green form a longest increasing subsequence. This sequence has length $n/5$, so $\varepsilon = .8$. Our algorithm rejects this sequence if and only if i, j belong to the same chunk, which happens with probability $5/n$.

More precisely, pick $i, j \in_R [n]$, where $i \leq j$. Then check that $a_i \leq a_j$.

This idea also fails. There is a sequence, shown in figure 25.2, that is .8-far from sorted but which is accepted with probability $1 - 5/n$. If we repeat our algorithm many times, then we need $O(n)$ repetitions before the probability of acceptance is $\leq .01$. [1]

25.1.3 Algorithm

The algorithm that ultimately works is a more sophisticated version of the two previous ideas. We pick a random index $i \in_R [n]$ and search for a_i using binary search. $\mathsf{BinarySearch}(a_i)$ queries $O(\log n)$ entries of A in order to home in on a_i. We claim that if the list is far from sorted, then with high probability, we can tell that the elements we queried did not come from a sorted list.

We say that $\mathsf{BinarySearch}(a_i)$ is **consistent** with a sorted list if the elements we query along the way appear to come from a sorted list. More precisely, at any step of the search, let $L, R \in [n]$ be the left and right indices of the subarray we are searching. $\mathsf{BinarySearch}(a_i)$ is consistent if:

1. At every step, $a_L \leq a_{\lfloor (L+R)/2 \rfloor} \leq a_R$, and
2. $\mathsf{BinarySearch}(a_i)$ actually finds a_i.

At any step, $a_{\lfloor (L+R)/2 \rfloor}$ is the next element that we query, and its value must lie between the left and right values of the subarray.

[1] There is a way to use only $O(\sqrt{n})$ samples: sample $O(\sqrt{n})$ elements at once and check whether any pair of them is out of order. See [3] for more detail. $O(\sqrt{n})$ is a reasonably efficient number of queries since $O(\sqrt{n}) = o(n)$, but we can do better. We want to use only $O\left(\frac{\log}{\varepsilon}\right)$ queries.

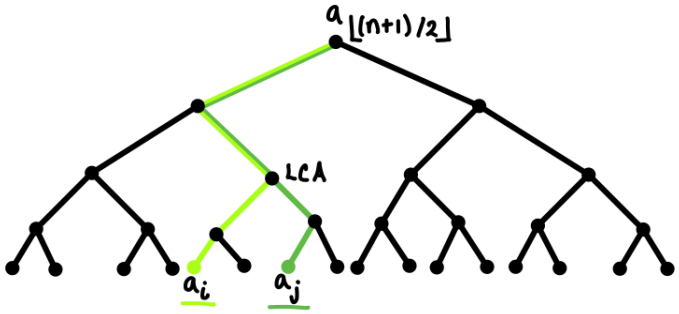

Figure 25.3: A binary search tree

Now we can state our algorithm.

Algorithm 1 Test the sortedness of A

Input: A

1. Sample $i \in_R [n]$.
2. Run $\mathsf{BinarySearch}(a_i)$. **Reject** the list if $\mathsf{BinarySearch}(a_i)$ is inconsistent with a sorted list.
3. Repeat the previous steps $100/\varepsilon$ times. If binary search is consistent on every iteration, then **accept** the list.

25.1.4 Analysis

This algorithm uses $O\left(\frac{\log n}{\varepsilon}\right)$ queries and time because there are $O(1/\varepsilon)$ iterations of binary search, each of which takes $O(\log n)$ queries/time.

If A is sorted, then binary search is always consistent, and the list is accepted with probability 1.

Theorem 25.1. *If A is ε-far from sorted, then A is accepted with probability $\leq .01$.*

Proof. Let a_i be "good" if $\mathsf{BinarySearch}(a_i)$ is consistent.

Lemma 25.2. *If a_i and a_j are good, and $i < j$, then $a_i \leq a_j$.*

Proof. To understand this proof, it helps to picture a binary search tree, shown in figure 25.3. $\mathsf{BinarySearch}(a_i)$ and $\mathsf{BinarySearch}(a_j)$ start by querying the same element, $a_{\lfloor (n+1)/2 \rfloor}$, and they may continue querying the same elements for a little while. The lowest common ancestor (LCA) of a_i and a_j is the last element that $\mathsf{BinarySearch}(a_i)$ and $\mathsf{BinarySearch}(a_j)$ both query. Since a_i and a_j are consistent with a sorted list, $a_i \leq LCA \leq a_j$.

□

Corollary 25.3. *The set of good elements form an increasing subsequence.*

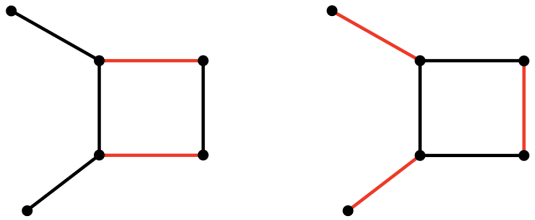

Figure 25.4: Two maximal matchings, shown in red, of the same graph. The first matching has size 2, while the second has size 3.

Proof. By lemma 25.2, any pair of good elements is ordered in increasing order: if $i < j$ then $a_i \leq a_j$. □

Next, the length of the longest increasing subsequence must be longer than the number of good elements. Since A is ε-far from sorted:

$$\begin{aligned} \#\text{ Good Elements} &\leq (1-\varepsilon)\cdot n \\ \mathbb{P}_i(a_i \text{ is good}) = \#\text{ Good Elements} \cdot \frac{1}{n} &\leq 1-\varepsilon \end{aligned}$$

Then A will be accepted with probability $\leq .01$:

$$\begin{aligned} \mathbb{P}(A \text{ accepted on a given iteration}) &= \mathbb{P}_i(a_i \text{ is good}) \leq 1-\varepsilon \\ \mathbb{P}(A \text{ accepted after } 100/\varepsilon \text{ iterations}) &= \mathbb{P}(A \text{ accepted on a given iteration})^{100/\varepsilon} \\ &\leq (1-\varepsilon)^{100/\varepsilon} \leq \frac{1}{e^{100}} \leq .01 \end{aligned}$$

□

25.2 Testing Maximal Matching Size

Given a graph G, we will estimate the size of a maximal matching of G. This is not a decision problem, so it is not strictly a property testing problem. But it is in the same spirit as property testing because we want to estimate some function by querying a small number of bits of the input.

Let $G = (V, E)$ be a graph. A **matching** of G is a subset of the edges, $M \subseteq E$, such that every vertex touches 0 or 1 of the edges in M. A **maximal matching** is a matching that cannot add any edges and still be considered a matching. More formally, $M \subseteq E$ is a maximal matching if $\forall e \in E \backslash M$, $M \cup \{e\}$ is not a matching.

Note that the maximal matchings of a graph don't necessarily all have the same size. Figure 25.4 shows two maximal matchings of different sizes for the same graph. Also note that *maximal* matchings are not the same as *maximum* matchings. Maximum matchings are matchings of the maximum possible size. They are a subset of the maximal matchings because if you add an edge to a maximum matching, it is no longer a matching.

25.2.1 Problem Statement

Given a maximal matching, we want to estimate its size in constant time.

Let G be a graph of maximum degree $\leq d = O(1)$. Let M be a random maximal matching of G, and let s be the size of M. Given G and M, we want an algorithm that produces an estimate $\tilde{s}$ of s such that with probability $\geq .99$, $\tilde{s} \in (1 \pm \varepsilon) \cdot s$, for some constant ε.

M will be represented in a particular form that makes it easier to design our algorithm. M is represented by a bijective function $\pi : E \to [|E|]$, along with its inverse. In this model, we can query π and π^{-1}.

M can be constructed from π^{-1}. First, π^{-1} defines an ordering of the edges: $\pi^{-1}(1), \ldots, \pi^{-1}(|E|)$. We will try to add each edge e to M in the order given by π^{-1}. e is actually added to M iff none of edges already added share a vertex with e. This is summarized by the following procedure:

1. Let $M = \{\}$.
2. For each $i \in \{1, \ldots, |E|\}$:
   (a) Add edge $\pi^{-1}(i)$ to M if $M \cup \{\pi^{-1}(i)\}$ is a matching.

This shows that M is uniquely determined by π, so π is a valid way to represent M.

Finally, if π is a uniformly random bijection, then our algorithm will run in expected time $2^{O(d)}/\varepsilon^2 = O(1)$.

25.2.2 Algorithm to Estimate Maximal Matching Size

Given G and M, expressed as (π, π^{-1}), we will estimate the size of M. Our algorithm samples random edges in the graph and checks whether they belong to M. Then the fraction of sampled edges that are in M is a good estimate of $|M|/|E|$.

Algorithm 2 Estimate the size of M

Input: G, M

1. Sample $400 \cdot d^2/\varepsilon^2$ random edges: $R := (e_1, \ldots, e_{400 \cdot d^2/\varepsilon^2})$.
2. For each $e \in R$, check whether $e \in M$ (using algorithm 3 below).
3. **Output:** $\tilde{s} = \frac{|E|}{|R|} \cdot \sum_{i=1}^{|R|} \mathbb{1}_{e_i \in M}$

Analysis:
We will analyze algorithm 3 in the next subsection, but here it suffices to know that algorithm 3 runs in expected time $2^{O(d)}$. Algorithm 2 makes $O(d^2/\varepsilon^2)$ calls to algorithm 3, so algorithm 2 runs in expected time $2^{O(d)} \cdot O(d^2/\varepsilon^2) = 2^{O(d)}/\varepsilon^2$.

Claim 25.4. $\mathbb{P}[(1-\varepsilon)s \leq \tilde{s} \leq (1+\varepsilon)s] \geq .99$.

Proof. This is a standard application of the Chernoff bound, so skip this proof if you are comfortable with the techniques. First,

$$\mathbb{E}[\tilde{s}] = \frac{|E|}{|R|} \cdot \sum_{i=1}^{|R|} \mathbb{P}_i(e_i \in M) = \frac{|E|}{|R|} \cdot |R| \cdot \frac{s}{|E|} = s$$

Second, $s \geq \frac{|E|}{2d}$ because M is maximal. Alternatively, if $s < \frac{|E|}{2d}$, then there would be $< \frac{|E|}{d}$ vertices touching an edge in M, and $< |E|$ edges touching one of those vertices. But since M is maximal, every edge must touch a vertex that touches an edge in M. So in fact, $s \geq \frac{|E|}{2d}$.

Third, by a Chernoff bound,

$$\begin{aligned}\mathbb{P}\big[(1-\varepsilon)s \leq \tilde{s} \leq (1+\varepsilon)s\big] &= 1 - \mathbb{P}\big[|\tilde{s} - \mathbb{E}(\tilde{s})| > \varepsilon \cdot \mathbb{E}(\tilde{s})\big] \\ &\geq 1 - 2\exp\left[-\frac{\varepsilon^2 \cdot s^2}{|R| \cdot (|E|^2/|R|^2)}\right] = 1 - 2\exp\left[-\frac{\varepsilon^2 \cdot s^2 \cdot |R|}{|E|^2}\right] \leq 1 - 2\exp\left[-\frac{\varepsilon^2 \cdot |R|}{4d^2}\right] \\ &= 1 - 2\exp(-100) > .99\end{aligned}$$

□

25.2.3 Algorithm to decide membership in M

Finally, we need an algorithm to decide whether a given edge is in M. Ours runs in expected time $2^{O(d)}$.

Recall the procedure for constructing M: we try to add each edge e in the order given by π. We fail to add e if M already contains an e' that shares a vertex with e. Let the **neighborhood** of e, denoted $N(e)$, be all the edges that share a vertex with e. Algorithm 3 searches over $N(e)$ for an e' that was added to M before we tried to add e.

Algorithm 3 Decide membership in M

Input: G, M, e

1. Compute $N(e)$.
2. For each $e' \in N(e)$:
   - **If** $\pi(e) < \pi(e')$, then do nothing. (Base case)
   - **Else** compute whether $e' \in M$. **If** $e' \in M$, then **reject** e. (Recursive case)
3. **Accept** e if it was not rejected on any iteration.

Analysis:
Algorithm 3 correctly decides whether $e \in M$. It checks every $e' \in N(e)$ and determines if any e' is in M and comes before e in π's ordering. $e \in M$ iff no such e' exists.

Algorithm 3 is recursive because it computes whether $e' \in M$ in order to decide whether $e \in M$. The recursion does halt because eventually it reaches the base case: $\pi(e) < \pi(e')$ for all of e's neighbors. In that case, we know that $e \in M$ without needing to make any recursive calls.

Finally,

Theorem 25.5. *If π is a uniformly random bijection, then algorithm 3 runs in expected time $2^{O(d)}$.*

Proof. Every sequence of recursive calls follows a path through the graph: $e, e^{(1)}, \dots, e^{(k-1)}$, where k is the length of the path. Furthermore, $\pi(e) > \pi(e^{(1)}) > \cdots > \pi(e^{(k-1)})$. On input e, the algorithm follows every path from e that has decreasing π-values.

For any k, there are $\leq d^{k-1}$ paths of length k that start with e. The probability that a path of length k has decreasing π-values is $1/k!$. Therefore, the expected number of recursive calls is:

$$\begin{aligned}\mathbb{E}(\#\text{ calls}) &\leq \sum_{\text{paths } P \text{ starting from } e} \text{length}(P) \cdot \mathbb{P}_\pi(\pi \text{ decreases along } P) \\ &\leq \sum_{k=1}^{\infty} d^{k-1} \cdot k \cdot \frac{1}{k!} = \sum_{k=1}^{\infty} \frac{d^{k-1}}{(k-1)!} \leq e^d = 2^{O(d)}\end{aligned}$$

The expected number of recursive calls is $2^{O(d)}$. Each call takes time $O(d)$, to check the neighbors of e, so the algorithm runs in expected time $O(d) \cdot 2^{O(d)} = 2^{O(d)}$. □

25.3 Final notes

Given a graph of constant degree, we can test other properties in constant time. For instance, there is a constant-time algorithm that tests whether a graph is triangle-free or far from triangle-free. However, the runtime is a tower of exponentials, $d^{d^{\cdots}}$, called a tetration. So it's not useful for large graphs.

We have an almost-complete characterization of which graph properties admit constant-time property testing algorithms.

Property testing algorithms are closely related to **probabilistically checkable proofs** (PCPs) from complexity theory. A PCP is a proof that some input satisfies a property P. The proof can be checked in poly(n) time by randomly querying a small fraction of its bits.

References

[1] Goemans, M. Chernoff bounds, and some applications, February 2015.

[2] Harvey, N. Lecture 21: Property testing, spring 2012.

[3] Sepehr Assadi, Daniel Bittner, J. D. Lecture 4: Property testing, February 2020.