Lecture 21

*Prof. Nick Harvey* *University of British Columbia*

# 1 Overview of Property Testing

**Property testing** is a research area in theoretical computer science that has seen a lot of activity over the past 15 years or so. A one-sentence description of this area's goal is: *design algorithms that, given a very large object, examine the object in very few places and decide whether it either has a certain property, or is "far" from having that property.*

As a simple example, let the object be the set of all people in Canada, and let the property be "does a majority like Stephen Harper?". The first algorithm that comes to mind for this problem is: sample a few people at random, ask them if they like Stephen Harper, then return the majority vote of the sample.

Does this algorithm have a good probability of deciding whether a majority of people likes Stephen Harper? The answer is no. Suppose there are 999,999 people in Canada. The sampling algorithm cannot reliably distinguish between the case that exactly 500,000 people like Stephen Harper (a majority) and exactly 499,999 people like Stephen Harper (not a majority).

But our algorithm is in fact a good property testing algorithm for this problem! The reason is that we have not yet discussed the important word "far". That word allows us to ignore these scenarios that are right "on the boundary" of having the desired property.

In our example, we could formalize the word "far" as meaning "more than 5% of the population needs to change their vote in order for a majority to like Stephen Harper". Equivalently, our algorithm only needs to distinguish two scenarios:

- At least 500,000 people like Stephen Harper, or
- Less than 450,000 people like Stephen Harper.

Our sampling algorithm has good probability of distinguishing these scenarios. The number of samples needed depends only on the fraction 5% and the desired probability of success, and does not depend on the size of the population. (This is easy to prove using Chernoff bounds.)

This example probably doesn't excite you very much because statisticians have studied these sorts of polling problems for centuries, and we know all about confidence intervals, etc. The field of property testing does not focus on these simple polling problems, but instead tends to look at problems of a more algebraic or combinatorial flavour. Some central problems in property testing are:

- Given a function $f : \{0,1\}^n \to \{0,1\}$, decide if it is either a linear function or far from a linear function. There is a property testing algorithm for this problem that uses only a constant number of queries. This algorithm is a key ingredient in proving the infamous PCP theorem.
- Given a graph $G = (V, E)$, decide if $G$ has no triangles, or is far from having no triangles. There is a property testing algorithm for this problem that uses only a constant number of queries, however the constant is a horrifying tower function (a.k.a., a tetration).

## 2 Testing Sortedness

Let $A_1, \ldots, A_n$ be a list of $n$ distinct numbers. The property we would like to test is "is the list sorted?". Our definition of "far" from sorted will be "we need to remove at least $\epsilon n$ numbers for the remaining list to be sorted". In other words, we wish to distinguish the following two scenarios:

- $A_1 < A_2 < \cdots < A_n$.
- The longest sorted subsequence has length less than $(1-\epsilon)n$.

We will give a randomized algorithm which can distinguish these cases with constant probability by examining only $O(\log(n)/\epsilon)$ entries of the list.

A natural algorithm to try is: pick $i$ at random and test whether $A_i < A_{i+1}$. Consider the input

$$1, 2, \ldots, n/2-1, n/2, 1, 2, \ldots, n/2-1, n/2.$$

We need to remove at least half of the list to make it sorted. But that algorithm will only find an unsorted pair if it picks $i = n/2$, which happens with low probability.

So instead we consider the following more sophisticated algorithm.

- For $t = 1, \ldots, 1/\epsilon$
  - Pick $i \in \{1, \ldots, n\}$ uniformly at random.
  - Search for the value $A_i$ in the list using binary search.
  - If we discover some unsorted elements during the binary search, return "No".
  - If the binary search does not end up at position $i$, return "No".
- Return "Yes".

**Theorem 1** *This algorithm has constant probability of correctly distinguishing between lists that are sorted and those that are far from sorted.*

Proof: If the given list is sorted, obviously every binary search will work correctly and the algorithm will return "Yes".

So let us suppose that the list is far from sorted. Say that an index $i$ is "good" if a binary search for $y_i$ correctly terminates at position $i$ (without discovering any unsorted elements). We claim that these good indices form a sorted subsequence. To see this, consider any two good indices $i$ and $j$ with $i < j$. Let $k$ be the last common index of their binary search paths. Then we must have $y_i \leq y_k \leq y_j$, which implies that $y_i < y_j$ by distinctness.

Since the list is far from sorted, there can be at most $(1-\epsilon)n$ good indices. So the probability of picking a good index in every iteration is $(1-\epsilon)^{1/\epsilon} \leq e^{-\epsilon(1/\epsilon)} = 1/e$. $\square$

## 3 Estimating Maximal Matching Size

Our next example deviates from the property testing model described above. Instead of testing whether an object has a property or not, we will estimate some real-valued statistic of that object.

Our example is estimating the size of a **maximal matching** in a bounded-degree graph. There is an important distinction here. A *maximum matching* is a matching in the graph such that no other matching contains more edges. A *maximal matching* is a matching for which it is impossible to get a bigger matching by adding a single edge. Whereas all maximum matchings have the same size, it is not necessarily true that all maximal matchings have the same size.

Here is a greedy algorithm for generating a maximal matching in a graph $G = (V, E)$ with $n = |V|$, $m = |E|$ and maximum degree $d$.

- Let $M = \emptyset$.
- Let $\pi : \{1, \ldots, m\} \to E$ be an arbitrary bijection.
- For $i = 1, \ldots, m$
  - If both endpoints of edge $\pi(i)$ are uncovered, add $\pi(i)$ to $M$.

**Fact 2** *The resulting matching $M$ is maximal. Furthermore $|M| \geq n/2d$.*

**Theorem 3** *Let $G$ be a graph with maximum degree $d$. There is an algorithm to estimate the size of the maximal matching to within a multiplicative factor of $1 + \epsilon$ in time $2^{O(d)}/\epsilon^2$.*

**Estimating given an oracle.** Suppose we have a oracle which, assuming some fixed maximal matching $M$, can answer queries about whether a given edge $e$ is covered by that matching $M$. We will use this oracle to estimate $|M|$.

The algorithm just involves simple sampling.

- Fix a maximal matching $M$.
- Set $k = 9d^2/\epsilon^2$.
- For $i = 1, \ldots, k$
  - Pick a random edge $e$.
  - Let $X_i = 1$ if $e$ is in $M$, otherwise $X_i = 0$.

Let $X = \sum_{i=1}^{k} X_i$ and $\mu = \mathrm{E}[X]$. The actual number of edges in $M$ is

$$|M| \;=\; \mathrm{E}[X_i]m \;=\; \tfrac{m}{k}\mu.$$

Our estimate for the size of $M$ is $\frac{m}{k}X$. Since $m \leq nd/2$, Fact 2 shows that $\mathrm{E}[X_i] \geq 1/d^2$, which implies that $\mu \geq k/d^2$. Therefore

$$\begin{aligned} \Pr\Big[\ (1-\epsilon)\tfrac{m}{k}\mu \leq \tfrac{m}{k}X \leq (1+\epsilon)\tfrac{m}{k}\mu\ \Big] \;&=\; \Pr\Big[\ (1-\epsilon)\mu \leq X \leq (1+\epsilon)\mu\ \Big] \\ &\geq\; 1 - 2\exp(-\epsilon^2\mu/3) \;\geq\; 0.9, \end{aligned}$$

by a Chernoff bound.

**Implementing the oracle.** We will not actually implement the oracle for an arbitrary $M$; we will require $M$ to be generated by the greedy algorithm above with a random ordering $\pi$.

Associate independent random numbers $r_e \in [0, 1]$ with each edge $e \in E$. They are all distinct with probability 1, and so they induce a uniformly random ordering on the edges. We let $M$ be the maximal matching created by the greedy algorithm above, using this ordering.

Our algorithm for implementing the oracle is as follows.

- Given an edge $e \in E$
- For every edge $e' \in E$ sharing an endpoint with $e$
  - If $r_{e'} < r_e$, recursively test if $e'$ is in the matching. If so, return "No".
- Return "Yes".

Clearly this algorithm is always correct: its logic for deciding whether $e \in M$ is equivalent to the greedy algorithm. The only question is: what is the running time of the oracle?

Consider an edge $f$ reachable by a path of length $k$ from the initial edge $e$. The recursion only considers the edge $f$ if the edges on this path have the $r_e$ values in decreasing order. The probability of that event is $k!$. The number of edges reachable by a path of length $k$ is less than $(2d)^k$. So the expected number of nodes explored is less than $\sum_{k=0}^{\infty}(2d)^k/k! = e^{2d}$. So the expected time required by an oracle call is $O(d) \cdot e^{2d} = 2^{O(d)}$.

Since the estimation algorithm calls the oracle $9d/\epsilon^2$ times, the expected total running time is $2^{O(d)}/\epsilon^2$.