Property Testing Course

Spring 2022

Matan Hamilis
Reichman University
matan@hamil.is

Contents

Preface

This is my scribes while taking the "Introduction to Property Testing" course at Reichman University during the Spring term of 2022. The course was given by Dr. Reut Levi. I have been trying to closely follow the contents and adding some explanations as I see fit, especially when I have personally found them helpful. If you find any mistake or have any question, feel free to contact me via email: matan@hamil.is I'm also available via other media as listed on my website https://hamil.is/contact/

Lecture 1

1.1 Introduction

Conceptually, we are used to consider algorithms as "efficient" if their execution steps are bounded by a polynomial $f(|x|)$ for input x. The best thing we can wish to achieve is an algorithm running in linear time, this is because we are used to think that we have to at least read the entire input to make so decision about the input itself.

In this course we will discuss a new kind of algorithms which we will call "efficient", but these kind of algorithms will be even more efficient in the sense that they are not even reading the entire input!

Those with some background in the realms of machine learning and optimization theory are probably familiar with the concept of "heuristics" in which we seek some solution to an optimizatioon problem without givigin laying solid theoretical proof for the efficiency of the method. In this course, however, we will not be dealing with such "heuristics", instead, we will lay these strong mathematical foundations and introduce the appropriate definitions, using these foundations we will prove certain bounds on our algorithms.

In classical computer science we were trying to save two main resources: time and space. In the realms of sublinear algorithms we will be mostly interested in the number of queryings used to access the input. In real world scenarios accessing data can be costly and therefore we may wish to optimize the number of queryies used to access the data.

The field of "property testing" is a subfield of sublinear algorithms, in which we ask "Yes/No" questions about some object. For example, given a graph, is it connected? Given a sequence of numbers, is it connected? and so on. These "Yes/No" questions are also known as **decision problems**. Besides decision problems, other known type of problems is **search problems** in which we seek for a specific instance of an object that satisfies a set of predefined constraints. For exmaple, given a graph G we can introduce these two kinds of problems:

1. Decision Problem: Does G contain an hamiltonian path? 2. Search Problem: Find a hamiltonian path in G, if such exists.

The general approach to property testing problems incorporates the relaxation of a decision problem. One kind of relaxation that is used in optimization problems comes in the form of *approximation algorithms* in which an NP-hard problem is taken an instead of finding a solution to the problem, we seek an approximate solution. For example, in the Vertex-Cover problem, instead of finding a cover of mimnimal size, we seek a cover whose size at most twice the size of the optimal solution, in sense providing a solution that is not "too far" from the optimal one.

In decision problems, where the output is binary (i.e. True/False), a relaxation can be done in a similar fashion. Given an instance to the problem we want to be eable to efficiently (i.e using small number of queries) to tell whether the instance holds the property or is at least ϵ-far from holding the property for some **proximity parameter** denoted ϵ.

We can think of a property as a subset of some general set of objects. For example, the property of a graph being connected implies the existence of some universal set of graphs $\mathcal{G}$ and a subset of $\mathcal{G}$ denoted $\mathcal{P}$ which includes all connected graphs. Therefore, for convenience, a property $\mathcal{P}$ will often be referred to as the set of all instances possessing these property.

A property testing algorithm for property $\mathcal{P}$ and proximity parameter ϵ is a **randommized** algorithm such that given some instance of a problem G:

- If $G \in \mathcal{P}$ the algorithm accepts with probability $\geq 2/3$. - If G is at least ϵ-far (w.r.t to some predetermined distance measure) from being in G the algorithm rejects with probability $\geq 2/3$.

Notice that by doing so we give absolutely zero guarantee about the output of our algorithm for instances that do not posses the property but are less than ϵ-far from holding it.

It is also important to mention that the constant of $2/3$ is arbitrary and any **constant** greater than $1/2$ would suffice because by the means of **amplification** we can increase it. The amplification can be done be repeating the experiment multiple times and taking the majority of the results response of the amplified algorithm, by using Chernoff Bound we can show that this repetition yields an algorithm with better probability guarantees. This comes with the obvious cost of greater query complexity, of course. This will be presented and defined in greater depth later.

1.2 Toy Example

1.2.1 The Problem

In this section we will solve a toy example for property testing which will examplify many aspects of proeprty testing.

- Input to the problem: A n-digit long binary string $w \in \{0,1\}^n$. - Output: Whether the string is all zeroes, i.e. whether $w = 0^n$

To solve the problem we will have to read the entire input because otherwise our algorithm may miss the appearance of "1" somewhere in the string. Instead, we will try to solve a relaxed form of the question: Is $w = 00\ldots0$ or are there at least $\epsilon \cdot n$ occurences of 1's in the string?

1.2.2 The Solution

The algorithm that solves the relaxed version of the problem will do the following (for a parameter ϵ): 1. Sample $s = 2/\epsilon$ positions uniformly. 2. If any of these positions has "1", reject, otherwise, accept.

We will show that this algorithm matches the definition of a property tester, i.e.: 1. If $w = 0^n$, then is accepts with probability of at least 2/3. 2. If w has at least ϵ-fraction of 1's, it rejects with probability of at least 2/3.

1.2.3 Solution Analysis

Notice that our algorithm has one-sided error. It will never reject when $w = 0^n$.

If w is ϵ-far from 0^n, then:

$$\begin{aligned} P[\text{error}] &= P[\text{no 1's found in the sample}] \\ &\leq (1-\epsilon)^s \\ &\leq e^{-\epsilon \cdot s} && (1 - x \leq e^{-x}) \\ &\leq e^{-2} && (s = 2/\epsilon) \\ &\leq 1/3 \end{aligned}$$

Notice that we can always **amplify** the tester. In general if a tester finds a witness (e.g finding 1 in vector w) with probability p, we can create a tester that finds a witness with probability $\geq 2/3$ by repeating the test $2/p$ times.

1.2.4 Modified Toy Example

In this subsection we discuss a modified version of our toy example in which we consider the following problem:

- Input: A n-characters long binary string $w \in \{0,1\}^n$. - Output: An estimate to the fraction of 1's in w.

It suffices to sample $s = \Theta(\epsilon^{-}2)$ positions and output the average to get the fraction of 1's $\pm\epsilon$ (i.e. with additive error of ϵ) with probability $\geq 2/3$.

This can be proven using Chernoff Bound:

Theorem 1.1 (Chernoff Bound)**.** *Let $X_1, X_2, \ldots, X_k$ be indentical, independent random variables in $[0,1]$ and let $p = E[X_1]$. Denote $X = \sum_{i=1}^{k} X_i$ (sum of the samples).*

Then for any $\epsilon \in (0,1]$ the following holds:

$$P\left[\left|\frac{X}{k} - p\right| \geq \epsilon\right] \leq 2 \cdot e^{-\epsilon^2 k/4}$$

This means that if we use those identical random variables and compute their average, then the average will deviate by more than ϵ from the expectation with only an exponentially small probability (depending on ϵ and k).

1.3 Testing if an Array is Sorted

In this section we will deal with a deeper problem, given an array, testing whether it is sorted or not. This problem, unlike previous one, had some academic work and papers published about.

Let's start with definition of the problem:

- Input: An array of length n: $x_1, x_2, \ldots, x_n$.
- Output: Accept if the array is sorted and reject otherwise.

We will try to solve the following **relaxed version**: Is the array sorted or at least ϵ-far from being sorted? We say an array is ϵ-far from being sorted if at least ϵ-fraction of its entries have to be modified for it to be sorted.

Results by [3] and [4] give lower and upper (and therefore a tight) bound on the efficiency of such tester with an upper bound of $O(\log n/\epsilon)$ and a lower bound of $\Omega(\log n)$.

1.3.1 Naive Approaches

Attempt 1

1. Select random i.

2. Check if $x_i > x_{i+1}$.

This may not work if the array is: $11\ldots100\ldots0$ (half 1s and half 0s), because the check will only succeed if we select $i = n/2$, which happens with low prorbability, this is despite this array is 1/2-far from being sorted.

From this failure we learn that checking the condition locally (i.e. on following items only) will not work with high probability.

Attempt 2

1. Select random i, j such that $i < j$.

2. Check if $x_i > x_j$.

This may not work if the array is: $1, 0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 5...$ (The series 1,2,3,4... interleaving with the series 0,1,2,3,...). This array is also 1/2-far from being sorted.

The tester will succeed only if $j = i + 1$ for odd j which happens with low probability.

From this failure we learn that checking the monotonicity of the array locally is also a must!

1.3.2 Correct Solution

The Algorithm

The algorithm works as follows:

1. Select $i \in [n]$ randomy and sample x_i.

2. Try to find the value x_i in the array by conducting a binary search.

Notice that the tester has one sided error! If the array is sorted it will always accept.

Soundness analysis

First, notice that we can assume, without loss of generality that all items are distinct, if this is not the case we can just transform x_i into the tuple (x_i, i) for all i.

We will now prove the soundness property of the tester.

Definition 1.1 (Good Location)**.** A location i in the array is **good** if the binary search finds x_i eventually.

Notice that in a sorted array all locations are good. Moreover, since we can assume each x_i is unique, if the binary search finds x_i it also finds it in location i.

We first prove the following lemma:

Lemma 1.1. *Let $i < j$ be two **good** locations, then $x_i < x_j$*

Proof. Let t be the first location in which the binary searches for x_i and x_j diverge. Since at this step we have stepped in two different directions, "left" towards x_i (smaller than x_t) to reach x_i and "right" towards x_j (bigger than x_t) and since both i and j are good we can tell then $x_t < x_j$ and $x_i < x_t$ and from these two it follows that $x_i < x_j$. □

Corollary 1.1. The array a when restricted to good points is sorted.

Notice that the rejection probability is exactly the fraction of non-good locations. We also know that if we can change all those non-good locations to obtain a sorted array, therefore the rejection probability is greater than ϵ where ϵ is the proximity parameter.

Therefore, to succeed with probability $\geq 2/3$ we will have to repeat the tes $\Theta(\epsilon^{-1})$ times yielding an overall complexity of $\Theta(\epsilon \cdot \log n)$.

Definition 1.2 (Adaptiveness)**.** We say a tester is **adaptive** if its queries depend on the results of some previous queries.

Notice that the tester we have shown for array sortedness is therefore adaptive, because the binary search depends on the values queried from the array.

If there is a promise that the array is sorted then given i, we know in advance which locations the binary search are going to be queried, thereby becoming non-adaptive. On the other hand, if i is not good then at least one of these locations will lead us in the wrong direction, being a witness for rejection. The probability to hit one of these witnesses is $\Omega(1/\log n)$.

1.4 Approximating the Number of Connected Components

In this section we deal with the problem of approximating the number of connected components in a graph in using sublinear number of queries. The input to the problem is a graph $G = (V, E)$ on n vertices. We assume that G is represented using an adjacency matrix. We also assume that the maximal degree of a vertex in the graph is d.

Achieving an exact answer will require $O(dn)$ time. Instead, we are insterested in find an approximate solution with an additive error of $\pm\epsilon \cdot n$ with probability $\geq 2/3$.

Best known result takes $O(\frac{d}{\epsilon^2})$ with a lower bound of $\Omega(\frac{d}{\epsilon^2})$ by [2]. In this section we will present a tester working in $O(\frac{d}{\epsilon^3})$. Notice that the number of queries is independent from the number of vertices n!

Let $G = (V, E)$ be a graph such that $n = |V|$. Let C denote the number of connected components in the graph. For every vertex u, define n_u to be the number of nodes in u's component. Notice that for each component A:

$$\sum_{u \in A} \frac{1}{n_u} = 1$$

The main idea behind our construction would be to estimate the sum $\sum_{u \in V} n_u^{-1} = C$ for a few random nodes. For each node u exactly one of the following holds:

- The component of u is small, in that case n_u^{-1} is large but we can compute it using BFS.
- The component of u is big, in that case n_u^{-1} is small so it doesn't affect much the total sum.
- We can stop the BFS after a few steps (like tester by [5]).

1.4.1 Estimating n_u

Our goal at this point is estimating n_u, the number of elements in u's component. Let $\hat{n}_u = \min\{n_u, 2/\epsilon\}$. Simply put, it means when u's component has less than $2/\epsilon$ nodes, $\hat{n}_u = n_u$. Otherwise, $\hat{n}_u = 2/\epsilon$, in that case:

$$\underbrace{0 \leq}_{\hat{n}_u < n_u} 1/\hat{n}_u - 1/n_u < 1/\hat{n}_u = \epsilon/2$$

So, their difference is in the range $(0, \epsilon/2)$.

1.4.2 Estimating the Number of Connected Components

Using $\hat{n}_u$, our estimation of n_u, we can estimate the number of connected components. Our estimation is denoted $\hat{C}$.

$$\hat{C} = \sum_{u \in V} \frac{1}{\hat{n}_u}$$

Algorithm 1: Approximate Number of Connected Components

Input : (G, d, ϵ), A undirected graph G of degree at most d and proximity parameter ϵ

Output: $\hat{C}$, an approximation to the number of connected components of G

begin

  repeat

    Pick random node u

    Compute $\hat{n}_u$ via BFS from u, stop after at most $2/\epsilon$ new nodes are revealed.

  until *done* $\Theta(\epsilon^{-2})$ *times*;

  return $\hat{C} = (\textit{average of the values } \hat{n}_u^{-1}) \cdot n$

It follows that:

$$\begin{aligned} |C - \hat{C}| &= \left| \sum_{u \in V} \frac{1}{n_u} - \sum u \in V \frac{1}{\hat{n}_u} \right| \\ &= \left| \sum_{u \in V} \left(\frac{1}{n_u} - \frac{1}{\hat{n}_n} \right) \right| \\ &\leq \sum_{u \in V} \frac{\epsilon}{2} \\ &= \frac{\epsilon n}{2} \end{aligned}$$

This still doesn't yield us an algorithm, since this estimation still has to be employed on each and every node in the graph to give the overall estimation $\hat{C}$, so we come up with another idea to avoid from sampling all nodes in the graph.

Instead, we will employ the approximation algorithm given in algorithm 1. The query complexity of the proposed algorithm is $\Theta(d/\epsilon^3)$ because we repeat $\Theta(\epsilon^{-2})$ times the BFS operation which costs us at most $\Theta(d/\epsilon)$ We can prove the soundness using Chernoff Bound like we did before.

1.5 Approximating the Weight of a Minimum-Spanning-Tree

In the Minimum Spanning Tree (MST) problem we take a graph with positively weighted edges as input and output a spanning subgraph with minimal total weight of its edges among all spanning subgraphs.

Algorithm 2: Kruskal's MST Algorithm

Input :

- A undirected graph $G = (V, E)$.
- Weight function $f : E \to \mathbb{N}$

Output: A tree $T \subseteq E$, spanning G with minimal weight

begin

Create a forest F from all vertices in the graph.

$F \leftarrow \{\{u\} \,|u \in V\}$ $S \leftarrow E$ **repeat**

Remove an edge $e = (u, v)$ with minimum weight from S

If u and v are on two different trees in the forest F, merge the trees into a single tree.

until *While $S \neq \emptyset$ and F isn't spanning the tree*;

if *F is spanning* **then return** F;

else return ERROR;

For exact answer we can use the deterministic algorithm by [1] take $O(m \log^* m)$ or the linear time randomized algorithm by [6].

If we only want to approximate it with sublinear number of queries we will have to take a different approach. In our problem we assume the given grapph $G = (V, E)$ is represented using an adjacency-list with maximum degree d and maximum allowed weight w. We will output a $(1 + \epsilon)$-approximation to the MST weight, w_{MST}.

Best known results are an upper bound of $\tilde{O}(dw/\epsilon^3)$ and lower bound of $\Omega(dw/\epsilon^2)$. In this section we will give a result bound by a small polynomial in $d, w, 1/\epsilon$. Note this is independent of n!

The approach is derived from Kruskal's algorithm for finding an MST given in algorithm 2. Now, let's begin by assuming there are only two possible weights in the tree, either 1 or 2. In that case we have:

$$\begin{aligned} w_{MST} &= (\#\text{ weight-1 edges in MST}) + 2 \cdot (\#\text{ weight-2 edges in MST}) \\ &= n - 1 + (\#\text{ weight-2 edges in MST}) \\ &= n - 1 + (\#\text{ of CCs induced by weight-1 edges}) - 1 \end{aligned}$$

The last transition follows from the fact that Kruskal's algorithm will first create a forest from the nodes of the graph using only edges with weight 1. Next, it will connect the resulting connected components in the forest using edges of weight 2. So, the last transition is correct since we need $C - 1$ to connect C connected components.

From this example case of weight of only 1 and 2, we can see that there is some connection between the weight of the MST and the number of connected components.

We will now generalize this approach to arbitrary number of weights. First, given a graph $G = (V, E)$ and a weight function $f : E \to \{1, ..., w\}$ denote by G_i to be the subgraph containing all edges of weight at most i. Denote by C_i the number of connected components of G_i. We know, from Kruskal's algorithm that the MST of G has at least $C_i - 1$ edges of weight $> i$.

Lemma 1.2. $w_{MST} = n - w + \sum_{i=1}^{w} C_i$

Proof. Let β_i be the number of edges of weight $> i$ in an MST.

$$\begin{aligned}
w_{MST}(G) &= \sum_{i=0}^{w-1} (i+1) \cdot (\#\text{edges in MST of weight } i+1) \\
&= \sum_{i=0}^{w-1} \underbrace{\sum_{j=i+1}^{w-1} (\#\text{edges in MST of weight } j+1)}_{\beta_i} \\
&= \sum_{i=0}^{w-1} \beta_i \\
&= \sum_{i=0}^{w-1} (C_i - 1) \\
&= -w + \sum_{i=0}^{w-1} C_i \\
&= n - w + \sum_{i=1}^{w-1} C_i
\end{aligned}$$

□

The algorithm follows directly from lemma 1.2 If all approximations $\tilde{C}_i$ are correct, then $\left|\tilde{C}_i - C_i\right| \leq \epsilon n/w$. Therefore, the approximation to the MST weight $\tilde{w}_{MST}$ isn't too far from the correct MST weight w_{MST}.

$$|\tilde{w}_{MST} - w_{MST}| \leq w \cdot \frac{\epsilon n}{w} = \epsilon n$$

In terms of probability, we get the correct approximation with probability $\geq 2/3$. By employing amplification we can get the correct approximation

Algorithm 3: Approximate MST Weight

Input : (G, w, d, ϵ)
A undirected graph $G = (V, E)$
A maximal weight of the edges w
A maximal degree of each vertex d
A proximity parameter ϵ

Output: An approximate weight of the MST $\tilde{w}_{MST}$

begin
  for $i = 1$ *to* w **do**
    $\tilde{C}_i \leftarrow \text{ApproxCC}(G_i, d, \epsilon/w)$
  return $\tilde{w}_{MST} = n - w + \sum_{i=1}^{w} \tilde{C}_i$

for the connected components of each graph with probability $\geq 1 - 1/3w$. From union bound we will get a good approximation to w_{MST} with probability $\geq 2/3$.

Lecture 2

Bibliography

[1] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[2] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34:1370–1379, 2005.

[3] Funda Ergün, Sampath Kannan, S Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. *Journal of Computer and System Sciences*, 60(3):717–751, 2000.

[4] Eldar Fischer. On the strength of comparisons in property testing. *Information and Computation*, 189(1):107–116, 2004.

[5] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 406–415, 1997.

[6] David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995.