An introduction to property testing*

1 Introduction

In many applications one deals with large datasets: genome data, social networks, high resolution images. To investigate such data, it is useful to estimate properties by algorithms that run in sublinear time. In some cases accessing data might be hard, for example if data arrives through a congested network, or if data needs to be collected through expensive experiments. In such cases, one is interested to minimize the numbers of queries to the dataset. In this chapter, we will study algorithm that work in linear and constant time, and such algorithms can only use a small fraction of their input. We use the RAM-model for computation.

There are only a few natural problems that can be solved in sublinear time deterministically. One such problem is approximating the diameter of a set of points within a factor 2. In this problem the input is an $n \times n$ distance matrix, which is a matrix that represents the pairwise distances of n points in a metric space. Such a matrix is always symmetric, has zeros on the diagonal, and satisfies the triangle inequality: $D_{i,j} + D_{j,k} \geq D_{i,k}$, for all i, j, k in $[1, n]$. The task is solved by a very simple algorithm:

```
**Data**: An $n \times n$ distance matrix $D_{i,j}$
**Result**: $(k, \ell)$ such that $D_{k,\ell} \geq \frac{1}{2} \max_{i,j}\{D_{i,j}\}$
Return $(1, \ell)$ for an $\ell$ for which $D_{1,\ell}$ is maximal.
```

This algorithm uses only one column of the matrix, and runs in time $O(\sqrt{\text{input size}})$. Why is the answer never less than half of the maximal pairwise distance?

Most sublinear algorithms are probabilistic in nature. In these notes, we study *property testing*: given a large object, does it have a property or it needs to be modified in many places before it satisfies the property? For example, given a graph, can we check whether it is connected, or do we need to add many edges before it becomes connected? In case the graph is not connected, but close to be connected, we do not care about the answer.

Property testing can be useful when exact decision procedures are slow (sometimes even NP-hard). Property testing is useful in the following settings:

- If we are guaranteed that for the object, the property either holds or is far from true, then we don't need the exact decision procedure at all.

*Author: Bruno Bauwens. Lecture notes for the course: Theoretical Computer Science at Higher School of Economics, 2015-2016.
Initial version: May 7, 2016.

- In some cases it is enough to have an object which approximately satisfies the property, because other algorithms can somehow "fix" the object. (This was the case in the proof of NP $\subset$ PCP$(n, 1)$, see further.)
- In many cases, if the test rejects, it also presents a "witness" or "proof" that the property is false. In the case of a negative instance, a property testing algorithm can speed up a slow decision procedure.

We used the notion of property testing implicitly in the chapter about the PCP-theorem, when we proved that NP $\subset$ PCP$(n, 1)$. In this proof, we constructed a tester to check whether a function is linear or far from linear. These notes are helpful to better understand the proof of NP $\subset$ PCP$(n, 1)$.

In the next section definitions are presented. Then we study property testing of half-planes, sorted lists, monotone functions and connectedness of graphs. For this part we follow the slides from [7]. In the last part, we prove that there exists a testing procedure for each regular language.

2 Definition

Two strings x and y of equal length are said to be ε-*far* from each other if they differ in more than $\varepsilon|x|$ bits.

Definition 1. *Let L be a language over some alphabet. A* tester *for L is a probabilistic algorithm that either rejects or accepts its input (x, ε) with $\varepsilon > 0$ such that:*

- *If $x \in L$, then the algorithm accepts with probability at least $2/3$.*
- *If x is ε-far from y for all $y \in L$, then the algorithm rejects with probability at least $2/3$.*

If $x \notin L$ but x is close to some element in L, we don't care about the result. See figure 1. For example, let L be the set of bitstrings that only contain zeros. There exists a tester for L that runs in time $O(1/\varepsilon)$. The tester operates as follows.

```
Data: String $x$
Result: Accepts if $x$ only contains zeros.
        Rejects with probability $\geq 2/3$ if $x$ has more than a fraction $\varepsilon$ of ones.
  • Sample $s = \lceil 2/\varepsilon \rceil$ positions of $x$ uniformly and independently at random;
  • If a one is found, reject; otherwise, accept.
```

The above algorithm works:

- If $x \in L$, then the algorithm always accepts.
- If x contains at least a fraction ε of ones, then a randomly selected sample is zero with probability at most $1 - \varepsilon \leq \exp(-\varepsilon)$. s samples are zero with probability at most

$$(1 - \varepsilon)^s \leq \exp(-s\varepsilon) \leq \exp(-2) \leq 1/3.$$

  Thus, the algorithm accepts with probability at most $1/3$.

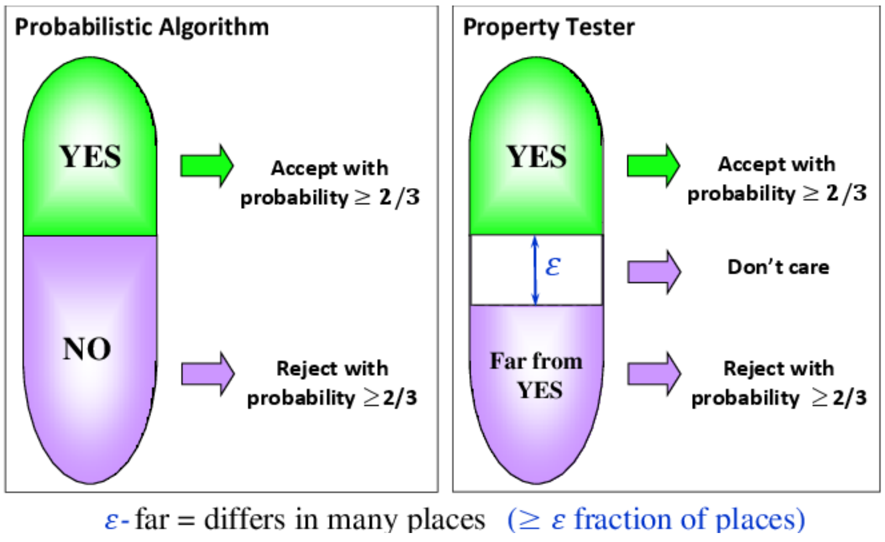

Probabilistic Algorithm
YES
Accept with probability ≥ 2/3
NO
Reject with probability ≥ 2/3
Property Tester
YES
Accept with probability ≥ 2/3
ε
Don't care
Far from YES
Reject with probability ≥ 2/3
ε- far = differs in many places (≥ ε fraction of places)

Figure 1: From [7].

The technique above will be used frequently and we refer to it through the following lemma.

Lemma 2 (Witness lemma)**.** *If an event happens with probability at least p, then in $\lceil 2/p \rceil$ independent repetitions of the experement, the probability that the event never happens is at most* $1/3$*.*

3 Testing properties of images

Imagine we are given a high resolution picture. We can only query a few bits of the picture. Can we detect whether the picture fits some template in constant time (regardless of the resolution)? See figure 2. We consider a special case, we test wheter a figure represents a halfplane.

The input is a picture of $n \times n$ pixels that are either black or white. A *half-plane* is such a picture for which a line can be drawn such that all black pixels are on one side of the line and all white pixels are on the other. See figure 3. A picture is ε-far from another picture, if at least a fraction ε of pixels needs to be changed to transform the first picture into the second.

Theorem 3 (Raskhodnikova [6])**.** *There exists a tester for half-planes that runs in time* $O(1/\varepsilon)$*.*

Proof. The idea is to first learn the rough structure of the picture by inspecting the corners and borders. Then, the interior is sampled to check whether this image respects this outline.

For each of the four borders of the picture, we can test whether the pixels in the corners have equal colour. The number of borders with different corners in a polygon is always even. In our case, this number can be 0, 2 or 4. The algorithm works as follows:

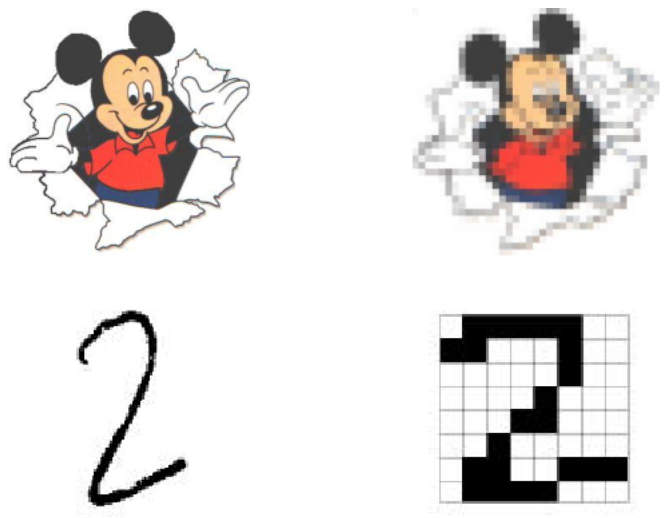


Figure 2: Does a picture fit some template? From [7].

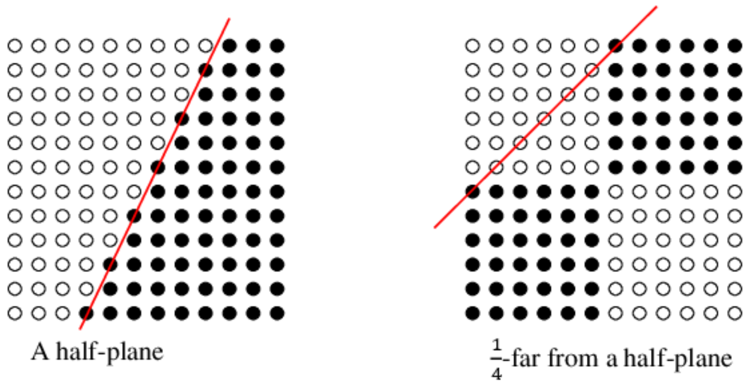


Figure 3: From [7].

Data: $\varepsilon > 0$ and an $n \times n$ picture
Result: Accepts if the picture is a half-plane.
Rejects with probability $\geq 2/3$ if it is ε-far from a half-plane.

1. If there are 4 sides with different corners, reject. [The image is not a half-plane.]

2. If all corners have the same colour, [the image is a half-plane iff all pixels have the same colour], sample $2/\varepsilon$ pixels and reject if one of them has a different colour, otherwise accept.

3. [Now, we have two sides with opposite colours.] On each such side, find two pixels of different colour within distance $\varepsilon/2$ using binary search.

4. The four pixels from the previous step define 2 regions B and W as indicated in figure 4. Sample $4/\varepsilon$ pixels from $B \cup W$ and reject if one of them has the wrong colour; otherwise accept.

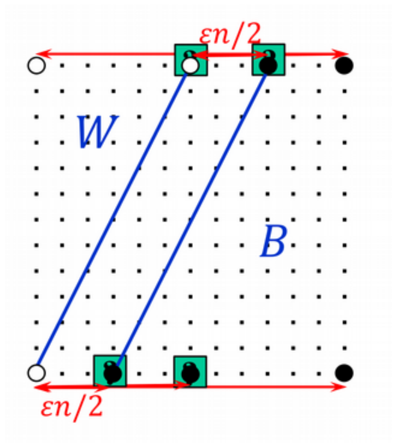


Figure 4: Testing whether a picture is a half-plane. From [7].

The binary search requires $O(\log(1/\varepsilon))$ time. The algorithm runs in time $O(1/\varepsilon)$. It is easily verified that if the picture is a half-plane, then the algorithm always accepts. It remains to show that if the picture is far from a half-plane, then the algorithm rejects. The case where all corners have the same colour is obvious, so we only need to bound the probability of acceptance in step 4. If the picture is far from any half-plane, then in particular it is far from a half-plane with some line through the area outside $B \cup W$. This area contains at most $\varepsilon n^2/2$ pixels. Because, the picture differs by at least εn^2 pixels from this half-plane, at least $\varepsilon n^2/2$ pixels differ in $B \cup W$. By the witness lemma, the algorithm will accept in step 4 with probability at most 1/3. □

Exercise 1. Let the picture be represented in the plain $\mathbb{R} \times \mathbb{R}$ such that neighbouring pixels are at distance 1. We say that a picture is *obtuse* if there is a polygon that contains precisely the black pixels, whose edges all have length more than 2 and whose (inner) angles are at least 90° and at most 180°. See figure 5. Show that there exists a tester for the set of obtuse pictures that runs in time $O(1/\varepsilon^2)$. (Hint: consider an algorithm that samples all points whose coordinates are multiples of $\varepsilon n/c$ for some c.)

Remark that any obtuse picture must be convex, in the sense that any triangle whose corners are black pixels can not contain white pixels. The result can be proven for convex pictures but the proof is more difficult, see [6].

4 Testing whether a list is sorted

Input: $\varepsilon > 0$ and a list $x_1, x_2, \ldots, x_n$ of integer numbers.
Question: is the list sorted: $x_1 \leq x_2 \leq \cdots \leq x_n$, or is it ε-far from sorted, i.e., more than a fraction ε of values x_i need to be changed to obtain a sorted list.

Attempts:

- Pick a random $i < n$ and verify that $x_i \leq x_{i+1}$.
  Fails for 111111000000, which is 1/2-far from sorted.

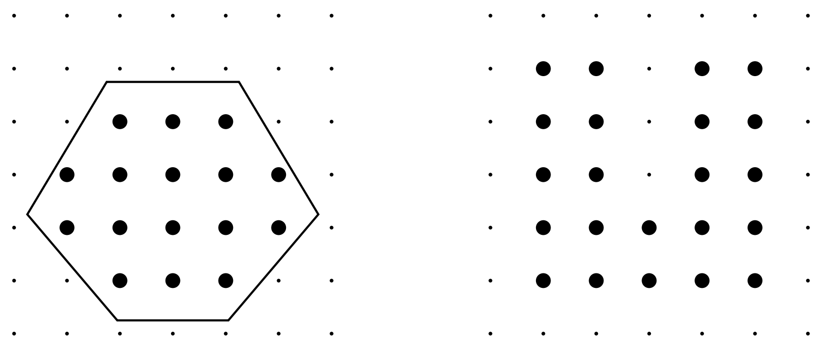


Figure 5: Left: an obtuse picture. Right: a picture that is not obtuse.

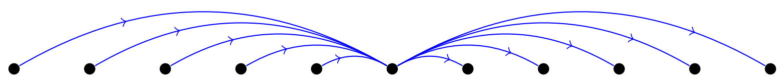
- Pick random $i \leq j \leq n$ and verify that $x_i \leq x_j$.
  Fails for 1032547698, which is 1/2-far from sorted.

Theorem 4 (Dodis et al. [2])**.** *There exists a tester for sorted lists that runs in time $O(\log n/\varepsilon)$.*

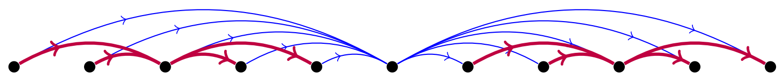
In the test, we use a directed graph $G = ([1, n], E)$ which is in some sense a 2-spanner:

- Edges leaving in a node i only go to nodes j with $j > i$.
- For all $i < j \leq n$ there is a path of length at most 2 from i to j.

Such a graph can be constructed by an inductive procedure. If the graph has 1 or 2 vertices, the construction is obvious. Now assume $n \geq 3$. Recall that $n/2 = \lfloor n/2 \rfloor$. First we connect all vertices strictly below $n/2$ to $n/2$, and $n/2$ to all vertices strictly above $n/2$.



Now, we need to connect all pairs of vertices below $n/2$ and all pairs of vertices above $n/2$. We repeat the procedure twice, once for the vertices $1 \ldots n/2 - 1$ and once for $n/2 + 1 \ldots n$.



We repeat this process recursively until $n \leq 2$. In total, we have $\log n$ levels of recursion, and the final graph has at most $n \log n$ edges. We are now ready to present the algorithm.

Data: $\varepsilon > 0$ and a list $x_1, \ldots, x_n$
Result: Accepts if $x_1 \leq x_2 \leq \cdots \leq x_n$.
Rejects with probability $\geq 2/3$ if the list is ε-far from sorted.

- Sample $4 \log n/\varepsilon$ edges (i, j) from the graph above.
- If $x_i \leq x_j$ for all these edges (i, j), then accept; otherwise reject.

Proof. We say that an edge (i, j) is *violated* if $x_i > x_j$. We say that a vertex i is *bad* if it is the start point or end point of some violated edge. Otherwise i is called *good.* The correctness of the algorithm follows from the two claims below.

1: *All good vertices are sorted.*

Indeed, let i and j be two good vertices such that $i < j$. Either they are connected, hence $x_i \leq x_j$. Otherwise, they are at distance 2 and connected through a vertex x_k. Because i and j are good, the edges are not violated, and $x_i \leq x_k \leq x_j$.

2: *If the list is ε-far from sorted, then more than $\varepsilon/(2 \log n)$ edges are violated.*

Indeed, if there were at least a fraction $1 - \varepsilon$ of good vertices, than the list is ε-close to be sorted. Hence, the list has more than a fraction ε of bad vertices. Every edge can be incident on at most 2 vertices, hence, the graph has more than $\varepsilon n/2$ bad edges. Recall that the graph has at most $n \log n$ edges. Hence, the ratio of bad edges is as in the claim and the theorem is proven. □

Exercise 2. We say that a list $x_1, \ldots, x_n$ is *Lipschitz* if $|x_j - x_i| \leq |j - i|$ for all $i, j \in [1, n]$. Show that there is a tester for the set of lists that are Lipschitz that runs in time $O((\log n)/\epsilon)$. (This is remarkable, because testing whether a circuit computes a Lipschitz function is NP-complete.)

5 Testing whether a function is monotone

A binary function on strings is *monotone* if for all strings a, b we have $f(a0b) \leq f(a1b)$.

Input: $\varepsilon > 0$ and a list of 2^n values describing a binary function on n-bit strings.
Question: is the function monotone or is it far from monotone, i.e., more than $\varepsilon 2^n$ values need to be changed to have a monotone function.

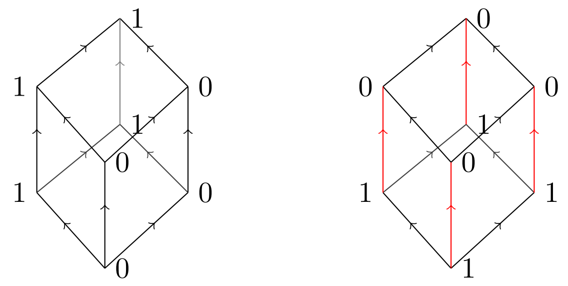


Figure 6: Left: monotone function; Right: 1/2-far from monotone

Theorem 5 ([3, 2])**.** *There exists a tester for monotone functions that runs in time $O(n/\varepsilon)$.*

Because the input has size 2^n, this is indeed a sublinear time algorithm.

Proof. We position the n-bit strings in a canonical way on the hypercube. A *(directed) edge* of the hypercube is any pair (x, y) where $x = a0b$ and $y = a1b$ for some strings a, b. A function is monotone if and only if for each edge (x, y) of the hypercube we have $f(x) \leq f(y)$. Otherwise we call the edge (x, y) *bad.*

Data: $\varepsilon > 0$, n and a list of 2^n values $f(x)$ for all n-bit x
Result: Accepts if f is monotone.
Rejects with probability $\geq 2/3$ if f is ε-far from monotone.

- Sample $2n/\varepsilon$ directed edges (x, y) from the hypercube.
- If $f(x) \leq f(y)$ for all these edges, then accept; otherwise reject.

Clearly, the algorithm always accepts if f is monotone. There are $n2^{n-1}$ edges in the hypercube. By the witness lemma, it suffices to show that if f is ε-far from monotone, then more then a fraction ε/n of edges are bad, i.e., at least $\varepsilon 2^{n-1}$ edges are bad. Let B be the set of bad edges. We show the contrapositive: if $|B| \leq \varepsilon 2^{n-1}$, then changing at most $\varepsilon 2^n$ values of f can make f monotone. This follows from Lemma 6.

Lemma 6. *If T edges of the hypercube are bad, then f can be made monotone by changing at most $2T$ values.*

Theorem 5 is proven except for Lemma 6. □

Proof of Lemma 6. Let B_i be the set of bad edges in direction $i \leq n$, i.e., the edges (x, y) that differ in the ith bit.

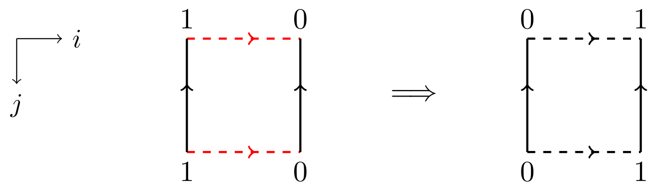
$$B = B_1 \cup B_2 \cup \cdots \cup B_n$$

Note that for each i, the edges in B_i are all disjoint. We modify the function f in n steps: in the ith step we swap the values of f on each edge in B_i. It only needs to be shown that this operation does not increase $|B_j|$ for all $j \leq n$, $i \neq j$.

An edge in B_i can be incident on at most two edges of B_j and these two edges of B_j are incident on at most two edges in B_i. These edges form a rectangle, and hence, we can check the claim for rectangles. More precisely, we prove that for a rectangle: *swapping values of f on bad edges in direction i does not increase the rectangle's number of bad edges in direction j.* We consider 4 cases.

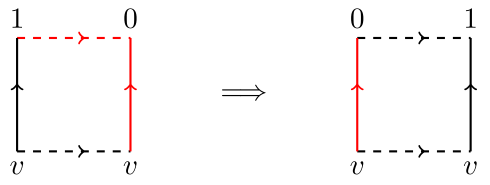
1. The rectangle has no bad edges in direction i. Nothing is swapped and the claim is trivially true.

2. The rectangle has two bad edges in direction i. Then the values of f on the rectangle are fixed:

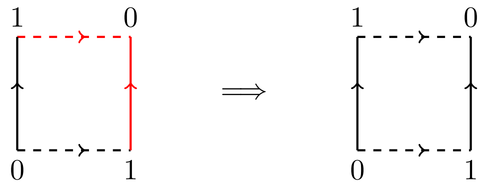


Swapping the values does not change the number of edges in direction j.

3. The rectangle has one bad edge in direction i and the values on the opposing edge are equal. Swapping the values on the bad edge also "swaps" the edges in direction j. The number of bad edges in direction j remains the same. (Below the picture for $v = 1$.)



4. The rectangle has one bad edge in direction i and the values on the opposing edge are different. This assumption fixes all the values of the rectangle, and swapping the values of f decreases the number of edges in B_j.



In all cases, the number of bad edges in direction j does not increase and the lemma is proven. □

6 Testing whether a graph is connected

We consider graphs of bounded degree. Many graphs are of this type, for example, the graph representing all links between web pages in the internet, or the graph representing social networks. We consider algorithms that have a graph $G = (V, E)$ as input and whose queries about G are of the form (v, i) where $v \in V$ and $i \in \mathbb{N}$. Upon such a query the ith neighbour of v is returned. If no such neighbour exists, a default value is returned.

Note that a list of all neighbours of all nodes has at most $2dn$ entries. We say that two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are ε-far if more than $\varepsilon 2dn$ entries differ in the lists corresponding to E_1 and E_2.

Theorem 7 (Goldreich [4])**.** *There exists a tester for the set of connected graphs of degree d that runs in time $O(1/(\varepsilon^2 d))$.*

Proof. The tester executes the following algorithm:

```
Data: $\varepsilon > 0$, $d$, $n$ and a graph $G$ with vertices $[1, n]$ and maximum degree $d$.
Result: Accepts if $G$ is connected.
        Rejects with probability $\geq 2/3$ if $G$ is $\varepsilon$-far from connected.
- If $n < 4/\varepsilon d$, check directly whether $G$ is connected.
- Otherwise, repeat $s = 8/\varepsilon d$ times:
    • Pick a random vertex $u$.
    • Check whether the connected component of $u$ is large:
      perform breath first search from $u$ and stop after finding $4/\varepsilon d$ vertices.
- Reject if a small component was found, otherwise accept.
```

Obviously, connected graphs are always accepted. The following two lemmas imply that if a graph is ε-far from being connected, it is rejected with probability at least $1/3$.

Lemma 8. *If G is ε-far from being connected, it has at least $\varepsilon dn/2$ connected components.*

Lemma 9. *If G has at least rn components for some $r > 0$, then G has at least $rn/2$ components of size at most $2/r$.*

The proof of the second lemma is an exercise. We now prove Lemma 8. We show that if G has less than $\varepsilon dn/2$ components, we can modify less than $\varepsilon 2nd$ entries of E to transform G into a connected graph of degree d. Indeed, adding an edge between two components requires to modify two entries of E. However, if all vertices of a component have already d neighbours, we first need to remove an edge in each of the two components we are connecting. We must do this carefully in order not to separate a component into two parts, and this is always possible. (Remove an edge that does not belong to a spanning tree.) In total we change at most 4 entries of E to connect two components. □

Exercise 3. Prove lemma 9. (A hint can be found at the end of these notes.)

7 Testing whether a string is accepted by an automaton

Before, we said that two strings are ε-close if one string could be transformed to the other by modifying at most an ε fraction of its bits. Besides modification of a symbol, we will now also consider the operations of inserting a single symbol, deleting a single symbol, and replacement of a substring, i.e., transforming a string $w = xaby$ to $xbay$, where x, a, b, y are all strings. The *edit distance* $d_E(x, y)$ between strings x and y is the minimal number of these operations needed to transform x to y.[1] For example, $d_E(0000, 0) = 3$. The strings 400022101010 and 401010100003 have distance at most 4 as witnessed by the following transformation (in fact they have distance 3):

400022101010	modify the 5th symbol to '3'
400032101010	delete the 6th symbol
40003101010	move symbols 2 to 5 to position 12
41010100003	insert a symbol '0' at position 2
401010100003	= final string

Observe that $d_E(x, y) = d_E(y, x)$ and that $d_E(x, y) + d_E(y, z) \geq d_E(x, z)$. In this section, we say that x is *ε-far* from y if $d_E(x, y) > \varepsilon|x|$. If $|x| = |y|$, then x is ε-far from y if and only if y is ε-far from x.

Input: $\varepsilon > 0$, a nondeterministic finite automaton $\mathcal{A}$, a string x
Question: Does $\mathcal{A}$ accept x or is x ε-far from any string accepted by $\mathcal{A}$?

Theorem 10 ([1, 5])**.** *There exists an algorithm that answers the question above using a number of bits of x that is polynomial in the size of $\mathcal{A}$ (and does not depend on $|x|$).*

[1]The term *edit distance*, can cover many different types of distances. It is often used with the operations insertion, deletion and modification of a single symbol. We will also use the movement of substrings.

Unfortunately, there exist $\mathcal{A}$ for which the algorithm has a computation time that is exponential in the size of $\mathcal{A}$. There exists a stronger version of the theorem that uses the same distance as from the previous sections (i.e. the hamming distance), for more details we refer to [1]. Before we prove Theorem 10, we first prove a special case. Recall that $\mathcal{P}(S)$ denotes the collection of all subsets of a set S. Let $\delta : Q \times A \to \mathcal{P}(Q)$ be the transition function of the automaton $\mathcal{A} = (Q, A, q_0, \delta, F)$. We define the extension $\delta' : Q \times A^* \to \mathcal{P}(Q)$ of δ over words by induction: $\delta(q, \text{empty word}) = q$ and for $a \in A$, $\delta'(q, ax) = \delta'(\delta(q, a), x)$.

Definition 11. *An automaton is* strongly connected *if for every two states q_1 and q_2, there exists a word x that brings the automaton from state q_1 to q_2, i.e., $\delta'(q_1, x) \ni q_2$.*

Proposition 12. *For each nondeterministic strongly connected finite automaton $\mathcal{A}$ there exists a tester for the set $L_{\mathcal{A}}$ of strings accepted by this automaton.*

Proof. We say that a string x is *feasible* if there exists a sequence of states that is consistent with the transitions of $\mathcal{A}$ on input x, more precisely, if there exists a $q \in Q$ such that $d'(q, x)$ is not empty. The tester for $\mathcal{A}$ operates as follows.

```
Data: $\varepsilon > 0$, a strongly connected automaton $\mathcal{A}$, $|x|$, and string $x$.
Result: Accepts if $\mathcal{A}$ accepts $x$.
        Rejects with probability $\geq 2/3$ if $x$ is $\varepsilon$-far from any string in $L_{\mathcal{A}}$.
  • Let $r = \varepsilon/2(m+1)$ where $m$ is the number of states of $\mathcal{A}$.
  • If $|x| < 2/r$, check directly whether $\mathcal{A}$ accepts $x$.
  • Select $4/r$ random substrings of $x$ of length $2/r$.
  • If all strings are feasible, then accept, otherwise reject.
```

Clearly, any string accepted by $\mathcal{A}$ is also accepted by the algorithm. Now suppose that a string is ε-far from any string in $L_{\mathcal{A}}$. The witness lemma and the following lemma imply that that algorithm accepts with probability at most $1/3$.

Lemma 13. *Let $r = \varepsilon/2(m+1)$. If x is ε-far from being accepted by $\mathcal{A}$, then x has at least $|x|r/2$ infeasible substrings of length $2/r$.*

To prove Proposition 12, it suffices to prove this lemma. □

Proof of Lemma 13. A *cut* of a string x is a list of strings $[w_1, w_2, \ldots, w_e]$ such that $x = w_1 w_2 \ldots w_e$. The lemma follows from the following two claims:

Claim 14. *If $d_E(x, L_{\mathcal{A}}) > (h+1)(m+1)$, then there exists a cut of x with at least h infeasible substrings.*

Claim 15. *If a cut of an n-bit string has at least rn infeasible substrings for some $r > 0$, it has at least $rn/2$ infeasible substrings of size less than $2/r$.*

Indeed, if $|x| < 1/r$, the lemma is trivially true. If $d_E(x, L_{\mathcal{A}}) > \varepsilon|x|$, then it has a cut for which the amount of infeasible strings is at least

$$\varepsilon|x|/(m+1) - 1 \geq 2r|x| - 1 \geq r|x|,$$

Now the lemma follows immediately from the second claim.

It remains to prove the claims. The second claim is proven in a similar way as Lemma 9. We prove the contra positive of the first claim: *If every cut of x has less than h infeasible strings, then $d_E(x, L_{\mathcal{A}}) \leq (h+1)(m+1)$.*

Construct a cut $[w_1, w_2, \ldots, w_{h'}]$ as follows: let w_1 be the minimal infeasible segment of x. Remove w_1 from x and let w_2 be the minimal infeasible initial segment of what remains, and so on. In this way, we obtain a list of $h'-1$ infeasible strings, (because the last part might be feasible). Hence, $h' \leq h$. By construction, removing the last symbol from a word w_i in the cut, makes the word feasible. Let $[v_1, v_2, \ldots, v_{h'}]$ be a list of feasible words obtained in this way.

We need to create a string in $L_{\mathcal{A}}$ that has small edit distance with $v_1 v_2 \ldots v_{h'}$. Let q_i and q'_i be the first and the last state of $\mathcal{A}$ in a sequence of states that witnesses that v_i is feasible (thus $\delta'(q_i, v_i) \ni q'_i$). Because $\mathcal{A}$ is strongly connected, there exists a string u_0 that transfers $\mathcal{A}$ from the initial state to q_1. We can assume that this string has length at most m. In a similar way, there exist strings $u_1, u_2, \ldots$ of length at most m that transfer $\mathcal{A}$ from q'_1 to q_2, from q'_2 to q_3, and so on. Let $u_{h'}$ be a string that connects $q_{h'}$ to an accept state. The string

$$u_0 v_1 u_1 v_2 \ldots v_{h'} u_{h'}$$

is accepted by $\mathcal{A}$ and can be obtained from $v_1 v_2 \ldots v_{h'}$ by at most $(h'+1)m$ insertions. In turn, this word can be obtained from $x = w_1 w_2 \ldots w_{h'}$ by at most e deletions. The total number of operations is bounded by $(h'+1)(m+1)$ and the claim follows because $h' \leq h$. $\square$

Proof of Theorem 10. Now, the automata is not strongly connected. We first present some definitions. Consider the directed graph G whose vertices are the states Q and for which there is an edge from q to each element in $\cup_{a \in A} \delta'(q, a)$, i.e., to all states that can be reached from q by reading at most one symbol of the alphabet A.

A strongly connected component of a vertex of a graph is the union of all cycles through this vertex. These components define a directed graph $\hat{G}$, where there is an edge from one component to another, if there exists a path from an element of the component to an element of the other component. An example is given in figure 7. Note that $\hat{G}$ has no directed cycles. This implies that there are at most 2^m different paths in $\hat{G}$. Why?

Let P be a path in $\hat{G}$, we say that a string x is *P-feasible* if x is feasible and the list of states that witnesses feasibility "follows" P in $\hat{G}$, i.e., the path $q_1, \ldots, q_\ell$ in G can be grouped in $|P|$ (possibly empty) sublists

$$[q_1, \ldots, q_a], [q_{a+1}, \ldots, q_b], \ldots, [q_{e+1} \ldots q_\ell]$$

such that subsequent sublists contain only states from subsequent components of P.[2]

Clearly, in Claim 15 we can replace *infeasible* by *P-infeasible* substrings for any path P in $\hat{G}$. For Claim 14 we need to increase the bound:

Claim 16. *If $d_E(x, L_{\mathcal{A}}) > (h+1)(m+1)^2$, then for every path P in $\hat{G}$ there exists a cut of x with at least h P-infeasible substrings.*

[2]This definition differs from [5].

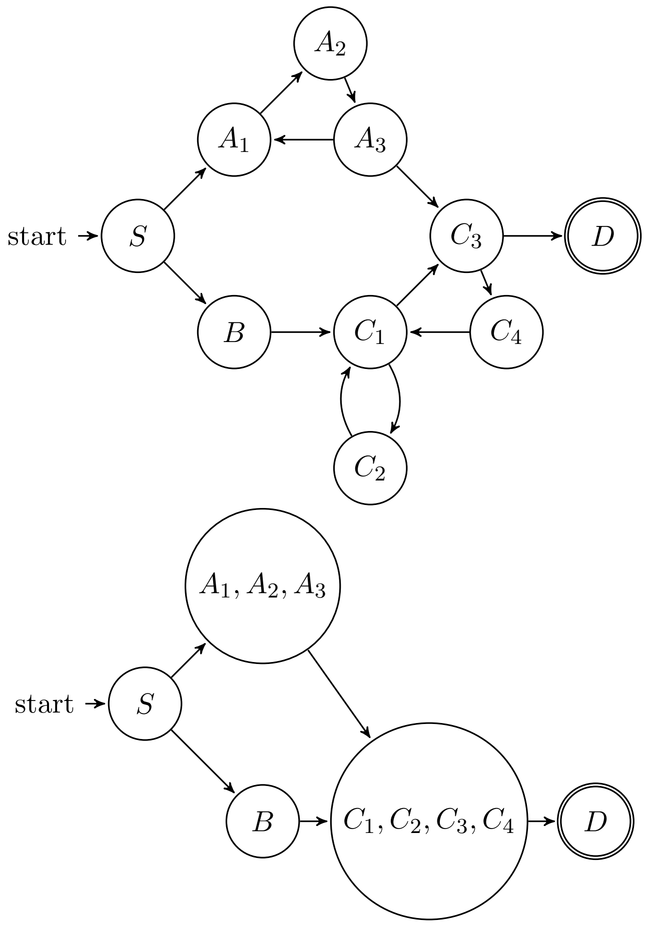


Figure 7: An automaton and its graph $\hat{G}$ of connected components

Proof of Claim 16. In the same way as above, we prove the counter positive. Assume that for some path P, every cut contains less than h P-infeasible substrings. We need to transform x to a string that is accepted by $\mathcal{A}$ using only a few insertions, deletions and block-movements. As before, we select a cut of x of minimal infeasible substrings. This cut has size $h' \leq h$. We make the substrings feasible by removing the last symbol, and in this way we obtain a list $[v_1, \ldots, v_{h'}]$ of at most h P-feasible substrings. Each v_i, contains $|P|$ parts, corresponding to each component in P, i.e., we can write each v_i as the concatenation of $|P|$ words $v_i = v_{i,1}v_{i,2}\ldots v_{i,|P|}$. Before we glue these components together, we move all blocks corresponding to each component together. This requires at most hm block movements (or even $(m-1)(h'-1)$ movements). Now we have to glue these hm blocks to each other, and this needs to be connected to the start state and some final state. In total the number of performed operations is at most

$$\overbrace{h}^{\text{make feasible}} + \overbrace{hm}^{\text{move blocks}} + \overbrace{hm^2}^{\text{glue blocks}} + \overbrace{2m}^{\text{start+accept}} \leq (h+1)(m+1)^2. \qquad \square$$

With this claim, we can generalize Lemma 13.

Lemma 17. *Let $r = \varepsilon/2(m+1)^2$ and let P be a path of $\hat{G}$. If x is ε-far from being accepted by $\mathcal{A}$, then x has at least $r|x|/2$ P-infeasible substrings of length $2/r$.*

```
Data: $\varepsilon > 0$, finite automaton $\mathcal{A}$, $|x|$ and string $x$.
Result: Accepts if $\mathcal{A}$ accepts $x$.
        Rejects with probability $\geq 2/3$ if $x$ is $\varepsilon$-far from any string in $L_{\mathcal{A}}$.
  1. Let $r = \varepsilon/2(m+1)^2$ where $m$ is the number of states of $\mathcal{A}$.
  2. If $|x| < 2/r$, check directly whether $\mathcal{A}$ accepts $x$.
  3. Select $4(m+1)/r$ random substrings of $x$ of length $2/r$.
  4. For each path $P$ in $\hat{G}$: if all the substrings are $P$-feasible, then accept.
  5. If the algorithm did not accept in the previous step, then reject.
```

Now it remains to explain why $d_E(x, L_{\mathcal{A}}) > \varepsilon|x|$ implies that the algorithm accepts with probability at most 1/3. Let us inspect step 4 of the algorithm for a fixed path P. By the same reasoning as above, we know that checking only $4/r$ randomly selected substrings, will lead to acceptance with probability at most 1/3. If we repeat this check $m+1$ times, (i.e., we check $4(m+1)/r$ random substrings), we will accept with probability at most $1/3^{m+1} \leq 2^{-m}/3$. There are at most 2^m paths in $\hat{G}$, hence by the union bound, the probability that one of these tests fails, is at most 1/3. The theorem is proven. □

Exercise 4. In this exercise we use again the string distance from the previous sections (we say that x is ε-far from y if x and y have different length or differ in more than $\varepsilon|x|$ places).

(1) Show Proposition 12 under the additional assumption that there is a state with a self loop. Thus, show that if $\mathcal{A}$ is strongly connected and has a state with a self loop, then there exists a tester for $L_{\mathcal{A}}$.

(2) Prove that for an automaton $\mathcal{A}$ for which every strongly connected component has a state with a self loop, there exists a tester for $L_{\mathcal{A}}$.

Exercise 5. Lemma 4 in the chapter about probabilistically checkable proofs presents a criterion for checking whether a function is close to linear. Write the resulting algorithm in detail and show that class of linear functions is testable. How many samples are used by the algorithm?

Hint for exercise 3. Note that the sum of the sizes of all components is at most n.

References

[1] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001.

[2] Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. In *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*, pages 97–108. Springer, 1999.

[3] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)*, 45(4):653–750, 1998.

[4] 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. ACM, 1997.

[5] Frédéric Magniez and Michel de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007.

[6] Sofya Raskhodnikova. Approximate testing of visual properties. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 370–381. Springer, 2003.

[7] Sofya Raskhodnikova. Sublinear algorithms. Technical report, Penn State University, 2012. Slides: www.cse.psu.edu/∼sxr48/sublinear-course/2012.html.