Space Complexity vs. Query Complexity

Oded Lachish[1], Ilan Newman[2], and Asaf Shapira[3]

[1] University of Haifa, Haifa, Israel, loded@cs.haifa.ac.il.
[2] University of Haifa, Haifa, Israel, ilan@cs.haifa.ac.il. *
[3] School of Computer Science, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv, Israel. asafico@tau.ac.il. **

Abstract. Combinatorial property testing deals with the following relaxation of decision problems: Given a fixed property and an input x, one wants to decide whether x satisfies the property or is "far" from satisfying it. The main focus of property testing is in identifying large families of properties that can be tested with a certain number of queries to the input. Unfortunately, there are nearly no general results connecting standard complexity measures of languages with the hardness of testing them. In this paper we study the relation between the space complexity of a language and its query complexity. Our main result is that for any space complexity $s(n) \leq \log n$ there is a language with space complexity $O(s(n))$ and query complexity $2^{\Omega(s(n))}$. We conjecture that this exponential lower bound is best possible, namely that the query complexity of a languages is at most exponential in its space complexity.

Our result has implications with respect to testing languages accepted by certain restricted machines. Alon et al. [FOCS 1999] have shown that any regular language is testable with a constant number of queries. It is well known that any language in space $o(\log \log n)$ is regular, thus implying that such languages can be so tested. It was previously known that there are languages in space $O(\log n)$ which are not testable with a constant number of queries and Newman [FOCS 2000] raised the question of closing the exponential gap between these two results. A special case of our main result resolves this problem as it implies that there is a language in space $O(\log \log n)$ that is not testable with a constant number of queries, thus showing that the $o(\log \log n)$ bound is best possible. It was also previously known that the class of testable properties cannot be extended to all context-free languages. We further show that one cannot even extend the family of testable languages to the class of languages accepted by single counter machines which is perhaps the weakest (uniform) computational model that is strictly stronger than finite automata.

1 Introduction

Basic Definitions: Combinatorial property testing deals with the following relaxation of decision problems: given a fixed property $\mathcal{P}$ and an input x, one

* Research was supported by the Israel Science Foundation (grant number 55/03)
** Research supported in part by a Charles Clore Foundation Fellowship.

wants to decide whether x satisfies $\mathcal{P}$ or is "far" from satisfying the property. This notion was first introduced in the work of Blum, Luby and Rubinfeld [5], and was explicitly formulated for the first time by Rubinfeld and Sudan [16]. Goldreich, Goldwasser and Ron [8] have started a rigorous study of what later became known as "combinatorial property testing". Since then much work has been done, both on designing efficient algorithms for specific properties, and on identifying natural classes of properties that are efficiently testable. For detailed surveys on the subject see [6, 15].

In this paper we focus on testing properties of strings, or equivalently languages [4]. In this case a string of length n is ϵ-far from satisfying a property $\mathcal{P}$ if at least ϵn of the string's entries should be modified in order to get a string satisfying $\mathcal{P}$. An ϵ-tester for $\mathcal{P}$ is a randomized algorithm that given ϵ and the ability to query the entries of an input string, can distinguish with high probability (say $2/3$) between strings satisfying $\mathcal{P}$ and those that are ϵ-far from satisfying it. The query complexity $q(\epsilon, n)$ is the maximum number of queries the algorithm makes on any input of length n. Property $\mathcal{P}$ is said to be testable with a *constant number of queries* if $q(\epsilon, n)$ can be bounded from above by a function of ϵ only. For the sake of brevity, we will sometimes say that a language is *easily testable* if it can be tested with a constant number of queries [5].

If a tester accepts with probability 1 inputs satisfying $\mathcal{P}$ then it is said to have a 1-sided error. If it may err in both directions then it is said to have 2-*sided error*. A tester may be *adaptive*, in the sense that its queries may depend on the answers to previous queries, or *non-adaptive*, in the sense that it first makes all the queries, and then proceeds to compute using the answers to these queries. All the lower bounds we prove in this paper hold for the most general testers, namely, 2-sided error adaptive testers.

Background: One of the most important questions in the field of property testing is to prove general testability results, and more ambitiously to classify the languages that are testable with a certain number of queries. While in the case of (dense) graph properties, many general results are known (see [2] and [3]) there are not too many general results for testing languages that can be decided in certain computational models. Our investigation is more related to the connection between certain classical complexity measures of languages and the hardness of testing them, which is measured by their query complexity as defined above. A notable result in this direction was obtained by Alon et al. [1] were it was shown that any regular language is easily testable. In fact, it was shown in [1] that any regular language can be tested with an optimal constant number of queries $\Theta(1/\epsilon)$ (the hidden constant depends on the language). It has been long known (see exercise 2.8.12 in [14]) that any language that can

[4] It will sometimes be convenient to refer to properties $\mathcal{P}$ of strings as languages L, as well as the other way around, where the language associated with the property is simply the family of strings that satisfy the property.

[5] We note that some papers use the term *easily testable* to indicate that a language is testable with $poly(1/\epsilon)$ queries.

be recognized in space [6] $o(\log \log n)$ is in fact regular. By the result of [1] this means that any such language is easily testable. A natural question is whether it is possible to extend the family of easily testable languages beyond those with space complexity $o(\log \log n)$. It was (implicitly) proved in [1] that there are properties in space $O(\log n)$ that are not easily testable, and Newman [13] raised the question of closing the exponential gap between the $o(\log \log n)$ positive result and the $\Omega(\log n)$ negative result. Another natural question is whether the family of easily testable languages can be extended beyond those of regular languages by considering stronger machines. Newman [13] has considered *non-uniform* extensions of regular languages and showed that any language that can be accepted by read-once branching programs of constant width is easily testable. Fischer and Newman [7] showed that this can not be further extended even to read twice branching programs of constant width. For the case of *uniform* extensions, it has been proved in [1] that there are context-free languages that are not easily testable.

In this paper we study a relation between the space complexity and the query complexity of languages. As a special case of this relation we resolve the open problem of Newman [13] concerning the space complexity of the easily testable languages. We also show that the family of easily testable languages cannot be extended to essentially no family of languages accepted by (uniform) machines stronger than finite state automata.

Main Results: As we have discussed above there are very few known connections between standard complexity measures and query complexity. Our first and main investigation in this paper is about the relation between the space complexity of a language and the query complexity of testing it. Our main result shows that in some cases the relation between space complexity and query complexity may be at least exponential. As we show in Theorem 2 below, it can be shown that there are languages, whose space complexity is $O(\log n)$ and whose query complexity is $\Omega(n)$. Also, as we have previously noted, languages whose space complexity is $o(\log \log n)$ can be tested with $\Theta(1/\epsilon)$ queries. Therefore, the interesting space complexities $s(n)$ that are left to deal with are in the "interval" $[\Omega(\log \log n), O(\log n)]$. For ease of presentation it will be easier to assume that $s(n) = f(\log \log n)$ for some integer function $x \leq f(x) \leq 2^x$. As in many cases, we would like to rule out very "strange" complexity functions $s(n)$. We will thus say that $s(n) = f(\log \log n)$ is *space constructible* if the function f is space constructible, that is, if given the *unary* representation of a number x it is possible to generate the *binary* representation of $f(x)$ using space $O(f(x))$.

[6] Throughout this paper we consider only deterministic space complexity. Our model for measuring the space complexity of the algorithm is the standard Turing Machine model, where there is a read only input tape, and a work tape where the machine can write. We only count the space used by the work tape. See [14] for the precise definitions. For concreteness we only consider the alphabet $\{0, 1\}$.

Note that natural functions, such as $s(n) = \log^2 \log n$ and $s(n) = \sqrt{\log n}$ are space constructible [7].

Theorem 1 (Main Result). *For any (space constructible) function $s(n)$ there is a language in space $O(s(n))$, whose query complexity is $2^{\Omega(s(n))}$.*

We believe it will be interesting to further study the relation between these two measures. Specifically, we raise the following conjecture claiming that the lower bound of Theorem 1 is best possible:

Conjecture 1. Any language in space $s(n)$ can be tested with query complexity $2^{O(s(n))}$.

As we have mentioned above, as one of the steps of the proof of Theorem 1 we prove the following result that may be of independent interest.

Theorem 2. *There is a language in space $O(\log n)$, whose query complexity is $\Omega(n)$.*

To the best of our knowledge, the lowest complexity class that was previous known to contain a language, whose query complexity is $\Omega(n)$, is **P** (see [9]). If Conjecture 1 is indeed true then Theorem 2 is essentially best possible.

As an immediate application of Theorem 1 we deduce the following corollary, showing that the class of easily testable languages cannot be extended from the family of regular languages even to the family of languages with space complexity $O(\log \log n)$ thus answering the problem raised by Newman in [13] concerning the space complexity of easily testable languages.

Corollary 1. *For any $k > 0$, there is a language in space $O(\log \log n)$, whose query complexity is $\Omega(\log^k n)$.*

Corollary 1 rules out the possibility of extending the family of easily testable languages from regular languages, to the entire family of languages, whose space complexity is $O(\log \log n)$.

We turn to address another result, ruling out another possible extension of regular languages. As we have mentioned before, it has been shown in [1] that there are context-free languages that are not testable. Hence, a natural question is whether there exists a uniform computational model stronger than finite state machines and weaker than stack machines such that all the languages that are accepted by machines in this model are testable. Perhaps the weakest uniform model within the class of context-free languages is that of a *deterministic single-counter automaton* (also known as one-symbol push-down automaton). A deterministic single-counter automaton is a finite state automaton equipped with a counter. The possible counter operations are increment, decrement and do nothing, and the only feedback from the counter is whether it is currently 0 or positive (larger than 0). Thus, such an automaton, running on a string ω reads an

[7] Note that when $s(n) = \log^2 \log n$ we have $f(x) = x^2$ and when $s(n) = \sqrt{\log n}$ we have $f(x) = 2^{x/2}$. We also note that we use the standard notion of space constructibility, see e.g. [14].

input character at a time, and based on its current state and whether the counter is 0, jumps to the next state and increments/decrements the counter or leaves it unchanged. Such an automaton accepts a string ω if starting with a counter holding the value 0 it reads all the input characters and ends with the counter holding the value 0. It is quite obvious that such an automaton is equivalent to a deterministic push-down automaton with one symbol stack (and a read-only bottom symbol to indicate empty stack). This model of computation can recognize a very restricted subset of context free languages. Still, some interesting languages are recognized by such an automaton, e.g. D_1 the first Dyck language, which is the language of balanced parentheses. Formal definition and discussion on variants of counter automata can be found in [18].

In this paper we also prove the following theorem showing that the family of testable properties cannot be extended even to those accepted by single-counter automata.

Theorem 3. *There is a language that can be accepted by a single-counter automaton and whose query complexity is* $\Omega(\log\log n)$*.*

Combining Theorem 3 and Corollary 1 we see that the family of testable properties cannot be extended beyond that of the regular languages in two natural senses.

Organization: The rest of the paper is organized as follows. In Section 2 we prove the exponential relation between space complexity and query complexity of Theorem 1. An important step in the proof is Theorem 2 that we also prove in this section. Section 3 contains the proof of Theorem 3 showing that there are languages accepted by counter machines that are not easily testable. Section 4 contains some concluding remarks and open problems. Due to space limitations several proofs are omitted and will appear in the full version.

2 Space Complexity vs. Query Complexity

In this section we prove that languages in space $s(n)$ may have query complexity exponential in $s(n)$. We start with an overview containing the important details of the proof of Theorem 2 stating that there are languages in space $O(\log n)$ that have query complexity $\Omega(n)$. We then show how to use Theorem 2 in order to prove the general lower bound of Theorem 1.

Overview of the Proof of Theorem 2: The construction of the language L in Theorem 2 is based on dual-codes of asymptotically good linear codes over $GF(2)$, which are based on Justesen's construction [10]. We begin with some brief background from Coding Theory (see [12] for a comprehensive background). A *linear code* C over $GF(2)$ is just a subset of $\{0,1\}^n$ that forms a linear subspace. The (Hamming) *distance* between two words $x, y \in C$, denoted $d(x, y)$, is the number of indices $i \in [n]$ for which $x_i \neq y_i$. The *distance* of the code, denoted $d(C)$ is the minimum distance over all pairs of distinct words of C, that is

$d(C) = \min_{x \neq y \in C} d(x,y)$. The *size* of a code, denoted $|C|$ is the number of words in C. The *dual-code* of C, denoted $C^{\perp}$ is the linear subspace orthogonal to C, that is $C^{\perp} = \{y : \langle x, y \rangle = 0 \text{ for all } x \in C\}$, where $\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i \pmod 2$ is the dot product of x and y over $GF(2)$. The *generator matrix* of a code C is a matrix G whose rows span the subspace of C. Note, that a code is a family of strings of fixed size n and our interest is languages containing strings of unbounded size. We will thus have to consider families of codes of increasing size. The following notion will be central in the proof of Theorem 2:

Definition 1. *An infinite family of codes $\mathcal{C} = \{C_1, C_2, \ldots\}$, where $C_n \subseteq \{0,1\}^n$, is said to be* asymptotically good *if there exist positive reals d and r such that $\liminf_{n\to\infty} \frac{d(C_n)}{n} \geq d$ and $\liminf_{n\to\infty} \frac{\log(|C_n|)}{n} \geq r$.*

We turn to discuss the main two Lemmas needed to prove Theorem 2. The first is the following:

Lemma 1. *Suppose $\mathcal{C} = \{C_1, C_2, \ldots\}$ is an asymptotically good family of linear codes. Then, for any infinite $S \subseteq \mathcal{N}$, the language $L = \bigcup_{n\in S} C_n^{\perp}$ has query complexity $\Omega(n)$.*

Lemma 1 is essentially a folklore result. Its (simple) proof relies on the known fact that if C is a code with distance t then $C^{\perp}$ is a *t-wise independent* family, that is, if one uniformly samples a string from $C^{\perp}$ then the distribution induced on any t coordinates is the uniform distribution. Such families are sometimes called in the coding literature *orthogonal array of strength* t, see [12]. The fact that the codes in $\mathcal{C}$ satisfy $\frac{\log(|C_{n_i}|)}{n_i} \geq r$ implies that a random string is with high probability far from belonging to $C_{n_i}^{\perp}$. These two facts allow us to apply Yao's principle to prove that even *adaptive* testers must use at least $\Omega(n)$ queries in order to test L for some fixed ϵ_0. As pointed to us by Eli Ben-Sasson, Lemma 1 can also be proved by applying a general non-trivial result about testers for membership in linear codes (see Theorem 3.3 in [4] for more details).

A well known construction of Justesen [10] gives an asymptotically good family of codes. By exploiting the fact that for appropriate prime powers n, one can perform arithmetic operations over $GF(n)$ in space $O(\log n)$, one can use the main idea of [10] in order to prove the following:

Lemma 2. *There is an asymptotically good family of linear codes $\mathcal{C} = \{C_1, C_2, \ldots\}$ and a space $O(\log n)$ algorithm, with the following property: Given integers n, i and j, the algorithm generates entry i, j of the generator matrix of C_n.*

Apparently this result does not appear in any published paper. However, most details of the construction appear in Madhu Sudan's lecture notes [17]. Lemma 2 immediately implies that the language $C^{\perp} = \cup C_i^{\perp}$ is recognizable in $O(\log n)$ space. Theorem 2 will follow by applying the above two lemmas. The proofs of the above Lemmas will appear in the full version of the paper.

Proof of Theorem 1: In this subsection we apply Theorem 2 in order to prove Theorem 1. To gain intuition for the construction, let us consider the case

$s(n) = \log\log n$. Consider the following language L_s: a string $x \in \{0, 1, \#\}^n$ is in L_s if it is composed of $n/\log n$ blocks of size $\log n$ each, separated by the # symbol, such that each block is a word of the language of Theorem 2. It can be shown that the query complexity of testing L_s is $\Omega(\log n)$. As the language of Theorem 2 is in space $O(\log n)$ it is clear that if the blocks of an input are indeed or length $O(\log n)$, then we can recognize L_s using space $O(\log\log n)$; we just run the space $O(\log n)$ algorithm on each of the blocks, whose length is $O(\log n)$. Of course, the problem is that if the blocks are not of the right length then we may be "tricked" into using too much space. We thus have to add to the language some "mechanism" that will allow us to check if the blocks are of the right length. This seems to be difficult as we need to initiate a counter that will hold the value n, but we need to do so without using more than $O(\log\log n)$ space, and just holding the value n requires $\Theta(\log n)$ bits.

The following language comes to the rescue: consider the language $\mathcal{B}$, which is defined as follows: for every integer $r \geq 1$, the language $\mathcal{B}$ contains the string $s_r = \text{bin}(0) * \text{bin}(1) * \ldots * \text{bin}(2^r - 1)*$, where $\text{bin}(i)$ is the binary representation of the integer i. Therefore, for every r there is precisely one string in $\mathcal{B}$ of length $(r + 1)2^r$. This language is the standard example for showing that there are languages in space $O(\log\log n)$ that are not regular (see [14] exercise 2.8.11). To see that $\mathcal{B}$ can indeed be accepted by a machine using space $O(\log\log n)$ note that again the wrong way to do so is to verify that the strings between consecutive $*$ symbols are the binary representations of consecutive integers, because if the string does not belong to $\mathcal{B}$ we may be tricked into using space $\Omega(\log n)$. In order to (safely) verify that a string belongs to $\mathcal{B}$ we check in the i^{th} step that the last i bits in the blocks form an increasing sequence modulo 2^i and that all blocks are of length i. We also check that in the last step we get an increasing sequence. This way in each step we use space $O(i)$, and it is easy to see that we never use more than $O(\log\log n)$ space.

Note that after verifying that a string $x \in \mathcal{B}$ we have an implicit representation of a number very close to $\log(|x|)$: this is just the number of entries before the first $*$ symbol. This also gives us a value close to $\log\log n$, which we needed in the previous example. The main idea for the proof of Theorem 1 is to "interleave" the language $\mathcal{B}$ with a language consisting of blocks of length $2^{s(n)}$ of strings from the language of Theorem 2. For ease of presentation the language we construct to prove Theorem 1 is over the alphabet $\{0, 1, \#, *\}$. It can easily be converted into a language over $\{0, 1\}$ with the same asymptotic properties by encoding each of the 4 symbols using 2 bits. The details follow.

Let L_2 be the language of Theorem 2 and let $s(n)$ satisfy $s(n) = f(\log\log n)$ for some space constructible function $n \leq f(n) \leq 2^n$ (recall the discussion before the statement of Theorem 1). In what follows we will use the notation L^k to denote the strings of some language L whose length is k, that is $L^k = L \cap \{0, 1, \#, *\}^k$. Given the function f, we define a language L_f that we need in order to prove Theorem 1 as the union of families of strings X_r of length $n(r)$, where for any $r \geq 1$ we define

$$n(r) = 2(r + 1)2^r.$$

A string $x \in \{0, 1, \#, *\}^{n(r)}$ belongs to X_r if it has the following two properties:

1. The odd entries of x form a string from $\mathcal{B}$ (thus the odd entries are over $\{0, 1, *\}$).
2. In the even entries of x, substrings between consecutive # symbols [8] form a string from L_2^k, where $k = 2^{f(\lfloor \log r \rfloor)}$. The only exception is the last block for which the only requirement is that it would be of length at most k (thus the even entries are over $\{0, 1, \#\}$).

Note that the words from L_2, which appear in the even entries of strings belonging to X_r all have length $2^{f(\lfloor \log r \rfloor)}$. We now define

$$L_f = \bigcup_{r=1}^{\infty} X_r \ . \tag{1}$$

and

$$K_f = \{2^{f(\lfloor \log r \rfloor)} : r \in \mathcal{N}\}. \tag{2}$$

Observe that the words from L_2, which appear in the even entries of strings belonging to L_f, all have lengths that belong to the set K_f. With a slight abuse of notation we now define the language L_2^f as the subset of L_2 consisting of words with lengths from K_f. By Theorem 2, when taking K_f as the set S in the statement of the theorem, we get the following claim:

Claim 1. *For some $\epsilon_0 > 0$, every ϵ_0-tester of L_2^f has query complexity $\Omega(n)$.*

We now turn to prove the main claims needed to obtain Theorem 1.

Claim 2. *The language L_f has space complexity $O(s(n)) = O(f(\log \log n))$.*

Proof: To show that L_f is in space $O(f(\log \log n))$ we consider the following algorithm for deciding if an input x belongs to L_f. We first consider only the odd entries of x and use the $O(\log \log n)$ space algorithm for deciding if these entries form a string from $\mathcal{B}$. If they do not we reject and if they do we move to the second step. Note, that at this step we know that the input's length n is $2(r+1)2^r$ for some $r \leq \log n$. In the second step we initiate a binary counter that stores the number $\lfloor \log r \rfloor \leq \log \log n$. Observe, that the algorithm can obtain r by counting the number of odd entries between consecutive $*$ symbols, and that we need $O(\log \log n)$ bits to hold r. We then construct a counter that holds the value $k = 2^{f(\lfloor \log r \rfloor)}$, using space $O(f(\lfloor \log r \rfloor))$ by exploiting the fact that f is space constructible [9]. We then verify that the number of even entries between consecutive # symbols is k, besides the last block for which we check that the

[8] The first # symbol is between the first block and the second block.

[9] More precisely, given the binary encoding of $\lfloor \log r \rfloor$ we form an unary representation of $\lfloor \log r \rfloor$. Such a representation requires $O(\log \log n)$ bits. We then use the space constructibility of f to generate a binary representation of $f(\lfloor \log r \rfloor)$ using space $O(f(\lfloor \log r \rfloor))$. Finally, given the binary representation of $f(\lfloor \log r \rfloor)$ it is easy to generate the binary representation of $2^{f(\lfloor \log r \rfloor)}$ using space $O(f(\lfloor \log r \rfloor))$.

length is at most k. Finally, we run the space $O(\log n)$ algorithm of L_2 in order to verify that the even entries between consecutive # symbols form a string from L_2 (besides the last block).

The algorithm clearly accepts a string if and only if it belongs to L_f. Regarding the algorithm's space complexity, recall that we use an $O(\log\log n)$ space algorithm in the first step (this algorithm was sketched at the beginning of this section). Note, that after verifying that the odd entries form a string from the language $\mathcal{B}$, we are guaranteed that $r \leq \log n$. The number of bits needed to store the counter we use in order to hold the number $k = 2^{f(\lfloor \log r \rfloor)}$ is $f(\lfloor \log r \rfloor) \leq f(\log\log n)$ as needed. Finally, as each block is guaranteed to be of length $2^{f(\lfloor \log r \rfloor)}$, the $O(\log n)$ algorithm that we run on each of the blocks uses space $O(\log(2^{f(\lfloor \log r \rfloor)})) = O(f(\lfloor \log r \rfloor)) = O(f(\log\log n))$ as needed. ■

Claim 3. *The language L_f has query complexity $2^{\Omega(f(\log\log n))} = 2^{\Omega(s(n))}$.*

Proof: By Claim 1, for some fixed ϵ_0 every ϵ_0-tester for L_2^f has query complexity $\Omega(n)$. We claim that this implies that every $\frac{\epsilon_0}{3}$-tester for L_f has query complexity $2^{\Omega(f(\log\log n))}$. Consider any $\frac{\epsilon_0}{3}$-tester T_f for L_f and consider the following ϵ_0-tester T_2 for L_2^f: Given an input x, the tester T_2 immediately rejects x in case there is no integer r for which $|x| = 2^{f(\lfloor \log r \rfloor)}$. Recall that the strings of L_2^f are all taken from K_f as defined in (2). In case such an integer r exists, set $n = 2(r+1)2^r$. The tester T_2 now *implicitly* constructs the following string x' of length n. The odd entries of x' will contain the unique string of $\mathcal{B}$ of length $(r+1)2^r$. The even entries of x' will contain repeated copies of x separated by the # symbol (the last block may contain some prefix of x). Note that if $x \in L_2^f$ then $x' \in L_f$. On the other hand, observe that if x is ϵ-far from L_2^f then x' is $(\frac{\epsilon}{2} - o(1))$-far from L_f, because in the even entries of x', one needs to change an ϵ-fraction of the entries in the substring between consecutive # symbols, in order to get a words from L_2^f (the $o(1)$ term is due to the fraction of the string occupied by the # symbols that need not be changed). This means that it is enough for T_2 to simulate T_f on x' with error parameter $\frac{\epsilon_0}{3}$ and thus return the correct answer with high probability. Of course, T_2 cannot construct x' "for free" because to do so T_2 must query all entries of x. Instead, T_2 only answers the oracle queries that T_f makes as follows: given a query of T_f to entry $2i-1$ of x', the tester T_2 will supply T_f with the i^{th} entry of the unique string of $\mathcal{B}$ of length $(r+1)2^r$. Given a query of T_f to entry $2i$ of x', the tester T_2 will supply T_f with the j^{th} entry of x, where $j = i \pmod{|x|+1}$. To this end, T_2 will have to perform a query to the entries of x.

We thus get that if L_f has an $\frac{\epsilon_0}{3}$-tester making t queries on inputs of length $2(r+1)2^r$, then L_2^f has an ϵ_0-tester making t queries on inputs of length $2^{f(\lfloor \log r \rfloor)}$. We know by Claim 1 that the query complexity of any ϵ_0-tester of L_2^f on inputs of length $2^{f(\lfloor \log r \rfloor)}$ is $\Omega(2^{f(\lfloor \log r \rfloor)})$. This means that the query complexity of T_2 on the inputs x' we described must also be $\Omega(2^{f(\lfloor \log r \rfloor)})$. The lengths of these inputs is $n = 2(r+1)2^r$. This means that $r = \log n - \Theta(\log\log n)$ and therefore the query complexity on these inputs is $\Omega(2^{f(\lfloor \log r \rfloor)}) = \Omega(2^{f(\log\log n - 2)}) = 2^{\Omega(f(\log\log n))}$, where in the last equality we used the fact that $f(x) \leq 2^x$. ■

Proof of Theorem 1: Take the language L_f and apply Claims 3 and 2. ■ ■

3 Testing Counter Machine Languages May Be Hard

In this section we define a language $\mathcal{L}$ and sketch an $\Omega(\log\log n)$ lower bound on the query complexity of *adaptive* testers for testing membership in $\mathcal{L}$. We start with defining the language $\mathcal{L}$.

Definition 2. *$\mathcal{L}$ is the family of strings $s \in \{0,1\}^*$ such that $s = 0^{k_1}1^{k_1}\dots 0^{k_i}1^{k_i}$ (The integers k_i are arbitrary). For every integer n we set $\mathcal{L}_n = \mathcal{L} \cap \{0,1\}^n$.*

We proceed with the proof of Theorem 3. First note that one can easily see that $\mathcal{L}$ can be accepted by a single-counter automaton as defined in Subsection 1. What we are left with is thus to prove the claimed lower bound on testing $\mathcal{L}$. Note that any adaptive tester of a language $L \subseteq \{0,1\}^*$ with query complexity $q(\epsilon, n)$ can be simulated by a non-adaptive tester with query complexity $2^{q(\epsilon,n)}$. Therefore, in order to prove our $\Omega(\log\log n)$ lower bound, we may and will prove an $\Omega(\log n/\log\log n)$ lower bound that holds for *non-adaptive* testers. To this end we apply Yao's minmax principle, which implies that in order to prove a lower bound of $\Omega(\log n/\log\log n)$ for non-adaptive testers it is enough to show that there is a distribution $\mathcal{D}$ over legitimate inputs (that is, inputs from $\mathcal{L}_n$ and inputs that are $\frac{1}{120}$-far from $\mathcal{L}_n$), such that for any non-adaptive *deterministic* algorithm Alg, which makes $o(\log n/\log\log n)$ queries, the probability that Alg errs on inputs generated by $\mathcal{D}$ is at least 1/3.

One of the key ingredients needed to construct $\mathcal{D}$ are the following two pairs of strings:

$$BAD_\ell = \left\{ \begin{array}{l} 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 0^\ell\ 1^\ell, \\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 1^\ell \end{array} \right\}$$

$$GOOD_\ell = \left\{ \begin{array}{l} 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell\ 0^\ell\ 0^\ell\ 1^\ell\ 1^\ell, \\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell\ 0^\ell\ 1^\ell \end{array} \right\}$$

where ℓ is a positive integer. Note, that each of the 4 strings is of length 12ℓ. We refer to strings selected from these sets as 'phrase strings'. We view the phrase strings as being composed of 12 disjoint intervals of length ℓ, which we refer to as 'phrase segments'. By the definition of the 'phrase strings' each 'phrase segment' is an homogeneous substring (i.e. contains equivalent symbols).

Note that for any ℓ, the 4 strings in BAD_ℓ and $GOOD_\ell$ have the following two important properties: (i) The 4 strings have the same (boolean) value in phrase segments $1, 4, 5, 8, 9$ and 12. (ii) In the other phrase segments, one of the strings in BAD_ℓ has the value 0 and the other has value 1, and the same applies to $GOOD_\ell$. The idea behind the construction of $\mathcal{D}$ and the intuition of the lower bound is that in order to distinguish between a string chosen from BAD_ℓ and a string chosen from $GOOD_\ell$ one must make queries into 2 distinct phrase segments. The reason is that by the above observation, if all the queries belong to segment $i \in [12]$, then either the answer is identical and is known in advance (in case $i \in \{1, 4, 5, 8, 9, 12\}$) or they are identical and have probability 0.5 to be either 0 or 1, *regardless* of the set from which the string was chosen.

In the construction of the distribution $\mathcal{D}$ we select with probability 1/2 whether the string we choose will be a positive instance or a negative instance. We select a positive instance by concatenating a set of strings uniformly and independently selected from $GOOD_\ell$ with strings of the form 0^t1^t. We construct negative instance in the same manner except that we replace the selection of strings from $GOOD_\ell$, by selecting strings from BAD_ℓ. Thus, the only way to distinguish between a positive instance and a negative instance is if at least two queries are located in the same phrase string, but in different phrase segments. The distribution $\mathcal{D}$ will be such that if the number of queries that is used is $o(\log n/ \log\log n)$, then with high probability there will be no two queries in two different phrase segments that belong to the same phrase string. As each phrase string is selected independently this makes it impossible for the tester to know whether the string is a positive instance or a negative one.

We assume henceforth that $n \geq 16$. Let $\mathcal{D}_N$ be a distribution over $\{0, 1\}^n$ that is defined by the following process of generating a string $\alpha \in \{0, 1\}^n$:

1. Uniformly select an integer $s \in [1, \lfloor \log n \rfloor - 3]$ and set $\ell = 2^s$.
2. Independently and uniformly select integers $b \in [6\ell]$, until the first time that the integers $b_1, \ldots, b_r$ selected satisfy $\sum_{i=1}^{r}(2b_i + 12\ell) \geq n - 24\ell$.
3. Independently and uniformly select r strings $\beta_1, \ldots, \beta_r \in BAD_\ell$.
4. For each $i \in [r]$ set $B_i = 0^{b_i}1^{b_i}\beta_i$. We refer to B_i as the i^{th} ‘block string‘. We refer to the substring $0^{b_i}1^{b_i}$ as the ‘buffer string‘ and β_i as the ’phrase’.
5. Set $\alpha = B_1 \cdots B_r 0^t 1^t$, where $t = (n - \sum_{i=1}^{r} |B_i|)/2$.

Let $\mathcal{D}_P$ be a distribution over $\{0, 1\}^n$ that is defined in the same manner as $\mathcal{D}_N$ with the exception that in the third stage we select independently and uniformly r strings $\beta_1, \ldots, \beta_r \in GOOD_\ell$. In the full version of the paper we use these two distributions to prove the required lower bound on testing $\mathcal{L}$.

4 Concluding Remarks and Open Problems

Our main result in this paper gives a relation between the space complexity and the query complexity of a language, showing that the later may be exponential in the former. We also raise the conjecture that this relation is tight, namely that the query complexity of a language is at most exponential in its space complexity. The results of this paper further show that the family of easily testable languages cannot be extended beyond that of the regular languages in terms of two natural senses; the space complexity of the accepting machine or the minimal computational model in which it can be recognized.

An intriguing related question is to understand the testability of languages with sublinear number of queries, that is $poly(\log n)$ or even just $o(n)$ queries. In particular, an intriguing open problem is whether all the context free languages can be tested with a sublinear number of queries. Currently, the lower bounds for testing context-free languages are of type $\Omega(n^{\alpha})$ for some $0 < \alpha < 1$. It seems that as an intermediate step towards understanding the testability of context-free languages, it will be interesting to investigate whether all the languages acceptable by single-counter automata can be tested with $o(n)$ queries. We note that the language we constructed in order to prove Theorem 3 can be tested with $poly(\log n, \epsilon)$ queries. See [11] for the full details.

Acknowledgements: The authors would like to thank Noga Alon, Madhu Sudan and Eli Ben-Sasson for helpful discussions.

References

1. N. Alon, M. Krivelevich, I. Newman and M. Szegedy, Regular languages are testable with a constant number of queries, SIAM J. on Computing 30 (2001), 1842-1862.
2. N. Alon and A. Shapira, A characterization of the (natural) graph properties testable with one-sided error, Proc. of FOCS 2005, 429-438.
3. N. Alon, E, Fischer, I. Newman and A. Shapira, A combinatorial characterization of the testable graph properties: it's all about regularity, Proc. of STOC 2006, 251-260.
4. E. Ben-Sasson, P. Harsha and S. Raskhodnikova, Some 3-CNF properties are hard to test, Proc. of STOC 2003, 345-354.
5. M. Blum, M. Luby and R. Rubinfeld, Self-testing/correcting with applications to numerical problems, JCSS 47 (1993), 549-595.
6. E. Fischer, The art of uninformed decisions: A primer to property testing, The Computational Complexity Column of The Bulletin of the European Association for Theoretical Computer Science 75 (2001), 97-126.
7. E. Fischer, I. Newman and J. Sgall, Functions that have read-twice constant width branching programs are not necessarily testable, Random Struct. and Alg., in press.
8. O. Goldreich, S. Goldwasser and D. Ron, Property testing and its connection to learning and approximation, JACM 45(4): 653-750 (1998).
9. O. Goldreich and L. Trevisan, Three theorems regarding testing graph properties, Random Structures and Algorithms, 23(1):23-57, 2003.
10. J. Justesen, A class of constructive asymptotically good algebraic codes, IEEE Transcations on Information, 18:652-656, 1972.
11. O. Lachish and I. Newman, Languages that are Recognized by Simple Counter Automata are not necessarily Testable, ECCC report TR05-152.
12. F. MacWilliams and N. Sloane, **The Theory of Error-Correcting Codes**, North-Holland, Amsterdam, 1997.
13. I. Newman, Testing of functions that have small width branching programs, Proc. of 41^{th} FOCS (2000), 251-258.
14. C. Papadimitriou, **Computational Complexity**, Addison Wesley, 1994.
15. D. Ron, Property testing, in: *Handbook of Randomized Computing*, Vol. II, Kluwer Academic Publishers, 2001, 597–649.
16. R. Rubinfeld and M. Sudan, Robust characterization of polynomials with applications to program testing, *SIAM J. on Computing* 25 (1996), 252–271.
17. M. Sudan, Lecture Notes on Algorithmic Introduction to Coding Theory, available at http://theory.lcs.mit.edu/∼madhu/FT01/scribe/lect6.ps.
18. L.G. Valiant, M. Paterson, Deterministic one-counter automata, Journal of Computer and System Sciences, 10 (1975), 340–350.