Electronic Colloquium on Computational Complexity, Report No. 187 (2013)

Property Testing on Linked Lists

Peyman Afshani
MADALGO*, Aarhus University
peyman@cs.au.dk

Kevin Matulef†
Aarhus University
kmatulef@cs.au.dk

Bryan T. Wilkinson‡
MADALGO*, Aarhus University
btw@cs.au.dk

November 11, 2013

Abstract

We define a new property testing model for algorithms that do not have arbitrary query access to the input, but must instead traverse it in a manner that respects the underlying data structure in which it is stored. In particular, we consider the case when the underlying data structure is a linked list, and the testing algorithm is allowed to either sample randomly from the list, or walk to nodes that are adjacent to those already visited. We study the well-known *monotonicity testing* problem in this model, and show that $\Theta(n^{1/3})$ queries are both necessary and sufficient to distinguish whether a list is sorted (monotone increasing) versus a constant distance from sorted. Our bound is strictly greater than the $\Theta(\log n)$ queries required in the standard testing model, that allows element access indexed by rank, and strictly less than the $\Theta(\sqrt{n})$ queries required by a weak model that only allows random sampling.

1 Introduction

Over the last 15 years, property testing has emerged as an important model for studying decision problems that are solvable in sublinear time (see, for instance, the surveys of [11, 15, 16]). In the testing model, the input is a large object—perhaps a large graph, or a function on some large domain. The testing algorithm must decide whether the object has a property or is "far" from having the property, while minimizing the portion of the object that the algorithm accesses.

A fundamental assumption of most property testing algorithms is that they can access the object by querying it arbitrarily. For instance, when testing functions, one typically assumes that the algorithm can query the function on any point in its domain. And when testing dense graphs, one typically assumes that the algorithm can query the graph's adjacency matrix to determine whether any particular (i, j) is an edge. The assumption of query access is rarely called into question, even when it does not reflect reality.

*MADALGO is a center of the Danish National Research Foundation.

†Supported by the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which part of this work was performed; and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed.

‡Supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

ISSN 1433-8092

One notable exception to the assumption of arbitrary query access is the so-called "sparse graph model," in which the large object is a sparse graph represented with adjacency lists, rather than an adjacency matrix [12]. A tester in this model can query for the identity of a neighbor of a given vertex, but cannot query for the existence of an edge between arbitrary (i, j). Since many large graphs in the real world, such as social networks, are sparse and stored as adjacency lists, this model is more plausible for practical settings.

The sparse graph model is an example of testing with only *limited* query access to the input, where the limitation comes from the underlying format in which the data is stored. Unfortunately, outside of sparse graphs (and hybrid graph models, such as in [13]), the property testing literature has largely ignored such real-world constraints. The purpose of this work is to initiate a study of property testing on inputs that cannot be accessed arbitrarily, but instead must be accessed in a way that respects the underlying data structure in which they are stored.

To illustrate this approach, we consider the classic problem of testing *sortedness* (a.k.a, *monotonicity* in one dimension). In the well-known version of this problem, the input is modeled as a function $f : [n] \to \mathbb{R}$, and the testing algorithm must minimize the number of queries it makes to f. In this setting, it is known that $\Theta(\epsilon^{-1} \log n)$ queries are both necessary and sufficient [8, 9].

The classic version of testing sortedness models the situations where the input is stored as a large array, since querying f is akin to accessing an array element. But in reality, not all data is stored in arrays. What if the input is stored as a linked list instead? Then we no longer have the ability to query $f(i)$ for any particular i, and the known upper bounds, which crucially rely on this ability, break down.

In this work, we define a model for property testing on linked lists. In the *linked list model*, a testing algorithm can access a random element of the list, or walk to elements that are adjacent to those it has already seen, but cannot access a specific element at some arbitrary rank i. Our main technical results are tight upper and lower bounds for testing monotonicity in this model.

Theorem 1.1. *For $\epsilon < 1/3$, any adaptive, order-based algorithm (with full rank access) for testing monotone versus ϵ-far from monotone in the linked list model has query complexity $\Omega(n^{1/3})$.*

Theorem 1.2. *There is a non-adaptive, order-based algorithm (with relative rank access) for testing monotonicity in the linked list model which makes $O(\epsilon^{-1} n^{1/3})$ queries.*

These results show that the linked list model is strictly weaker than the standard model which allows element access by rank, since, as previously mentioned, testing monotonicity in the standard model only requires $O(\log n)$ queries. However, the linked list model is also strictly stronger than one which only allows random samples, i.e. one in which the algorithm only learns the value of $f(i)$ for indices i chosen uniformly at random. Testing monotonicity in such a weak model requires $\Omega(\sqrt{n})$ queries, which is tight in terms of dependence on n.[1]

1.1 The Linked List Model

The basic premise of our model is simple: since linked lists do not support accessing elements at arbitrary positions, we consider algorithms that can either sample random elements in the list, or walk to elements that are adjacent to those it has already seen. Such a model has been studied previously in the algorithms literature. For instance, Chazelle et. al. [5] consider this model in their study of sublinear geometric algorithms.

[1] An upper bound of $O(\sqrt{n/\epsilon})$ follows from the work of [10], and a lower bound of $\Omega(\sqrt{n})$ from the following (folklore) argument: take a sorted increasing list and "swap" consecutive pairs of elements, e.g. $2, 1, 4, 3, 6, 5, ..., n, n-1$. Then a violation is only detected if two elements in the same pair are discovered, and with only random samples, a standard birthday paradox argument implies that $\Omega(\sqrt{n})$ samples are required to get two elements in the same pair.

However, when defining this model in the context of property testing, a subtlety arises. The issue is whether the algorithm knows the *rank* of the elements it randomly samples. If the algorithm asks for a random element of the list, and learns the value stored there, does it also know that it is receiving the 101st element of the list? Or the 59th, etc.?

It is not hard to show that if the algorithm has no knowledge of ranks, then efficient monotonicity testing is not possible. To briefly sketch the argument, consider, for instance, a list of n nodes consisting of one third 0's, followed by one third 1's, followed by one third 0's (i.e. $0, ..., 0, 1, ..., 1, 0, ..., 0$). It is easy to see that such a list is $(1/3)$-far from sorted, and an algorithm with access to the ranks can discover this with only a constant number of random samples (it just needs to sample a single 1 from the middle block, and a single 0 from the last block). But an algorithm without knowledge of the ranks cannot distinguish this from the list $0, ..., 0, 0, ..., 0, 1, ..., 1$ unless it finds the single "drop" from 1 to 0. Without the ranks, all the 1's are indistinguishable from each other, so finding the "drop" with high probability requires walking through $\Omega(n)$ list elements.

In our model, we require that the testing algorithm has at least partial knowledge of the ranks. We say that an algorithm has *full rank access* if it learns the exact ranks of each of the elements it queries. We say that an algorithm has *relative rank access* if it only learns the relative ranks of the elements it queries (for instance, if it queries two elements and learns that the first occurs before the second in the list, without learning exactly in which position either occurs). We note that with respect to these two definitions, our upper and lower bounds are as tight as possible. The upper bound of Theorem 1.2 only requires relative rank access, while the lower bound of Theorem 1.1 applies even to algorithms with full rank access.

The rank access assumptions do not rule out the applicability of the model to real-world situations. We note that it is well-known that linked lists can be augmented to answer relative rank queries with only $O(1)$ worst-case overhead (see the result of Dietz and Sleator [6], a solution to the order-maintenance problem).

Even when a linked list is not explicitly designed to answer rank queries, our model may still be applicable. For instance, suppose the list elements are records in a database. In some databases, the records are not stored in contiguous memory blocks, but instead in the structure of a linked list. Elements can be accessed (roughly) uniformly by jumping to different memory regions, but cannot be accessed at arbitrary positions. If the records are known to be ordered by a particular field, for instance a timestamp-based primary key, then we can use the values in that field to answer relative rank queries, while testing one of the other fields for sortedness. This loosely models a common situation in MongoDB, for instance, where collections of records are, by default, ordered by an increasing, timestamp-based key. They can be traversed in order of increasing key, but not accessed by their overall position [14].

1.2 Related Work

Monotonicity testing is one of the most studied problems in property testing, making it a natural candidate for study in our model. The one-dimensional case was studied in [8], and the lower bound was subsequently improved for adaptive algorithms in [9]. Functions on more general domains, such as the Boolean cube, were considered in [10, 7, 1]. Tight upper and lower bounds of $O(\epsilon^{-1} d \log n)$ are known for testing monotonicity over the hypergrid (i.e., functions of the form $f : [n]^d \to \mathbb{R}$) [3, 4]. The special case of functions with Boolean inputs and output (i.e., $f : \{-1, 1\}^d \to \{-1, 1\}$) is still an open problem; the best upper bound of [2] has an $O(d^{7/8})$ dependence on d , while the best lower bound (for one-sided error, nonadaptive algorithms) in [10] has an $\Omega(\sqrt{d})$ dependence.

1.3 Outline

In Section 2 we formally define our testing model. In Section 3 we prove the lower bound, Theorem 1.1. In Section 4 we prove the upper bound, Theorem 1.2.

2 Preliminaries

We define a *linked list* of length n as a tuple $L = (S, \mathbf{value}, \mathbf{rank}, \mathbf{next})$, where S is collection of n *labels*, **value** is a function from $S \to \mathbb{R}$, **rank** is a bijection from $S \to [n]$, and **next** is the function from $S \to S$ that maps the label $s \in S$ such that $\mathbf{rank}(s) = i$ to the label $s' \in S$ such that $\mathbf{rank}(s') = i + 1$ (if $\mathbf{rank}(s) = n$, we define $\mathbf{next}(s) = s$). That is, a linked list consists of n nodes with unique identifiers, where each node is located at a specific rank in the linked list and stores some specific value (here we are using real-numbered values, though they could be elements from any totally ordered set). One can think of S as a set of memory addresses for the nodes of the linked list.

For a given linked list L, we define the function $f_L : [n] \to \mathbb{R}$ which maps ranks to values such that $f_L(i) = \mathbf{value}(\mathbf{rank}^{-1}(i))$. For notational convenience, we will often drop the subscript and simply use f to refer to the linked list itself. We define the *distance* between two lists L_1 and L_2 as the fraction of i in $[n]$ such that $f_{L_1}(i) \neq f_{L_2}(i)$. We consider two lists *equivalent* if they have distance 0.

A *property* P is a set of linked lists that are defined by a family of functions $\mathcal{F}$ from $[n] \to \mathbb{R}$. Specifically, for such a family $\mathcal{F}$, a list L is in the property defined by $\mathcal{F}$ if and only if $f_L \in \mathcal{F}$. Note that this implies linked list properties are independent of the labels in S. For instance, the property of *monotonicity* is the subset of linked lists that satisfy $f(i) \leq f(j)$ for all $i \leq j$. We say that a linked list L is ϵ-far from property P if its distance from every list in P is at least ϵn.

A *tester* for a linked list property P is a randomized algorithm that can perform arbitrary computation on the information that it has accessed, but is limited in how it can access information. Initially, a tester has no labels with which to identify any of the linked list's nodes. However, the tester may call a **rand** oracle which returns a label uniformly at random from S. Once the tester has some labels, it can also obtain new labels via the linked list's **next** function.

When the tester has some labels with which to identify linked list nodes, it can get information about the ranks and values stored at those nodes. We define different variants of the model depending on the type of access the tester has to this information. Let the Boolean functions $\mathbf{less} : S \times S \to \{-1, 1\}$ and $\mathbf{before} : S \times S \to \{-1, 1\}$ compare the values and ranks, respectively, of two nodes given their labels (in other words, $\mathbf{less}(x, y)$ returns 1 if and only if $\mathbf{value}(x) < \mathbf{value}(y)$, and $\mathbf{before}(x, y)$ returns 1 if and only if $\mathbf{rank}(x) < \mathbf{rank}(y)$). We say that a tester has *full rank access* if it can directly query **rank**, but only *relative rank access* if it can only query **before**. Similarly, we say that a tester is *value-based* if it can directly query **value**, but only *order-based* if it can only query **less**.

The tester must output **Accept** with probability at least 2/3 if the list is in P and **Reject** with probability at least 2/3 if it is ϵ-far from P. Here the probability is taken over both the random coin flips of the algorithm, and the uniformly random output of **rand**.

The *query complexity* of a tester is defined as the maximum number of distinct labels of S returned by **rand** or **next** over any run of the algorithm.

The above restricted definition of a tester applies to our upper bound. We prove our lower bound for a stronger class of testers. We want our lower bound to subsume the setting in which the linked list nodes are stored contiguously in memory but not necessarily in rank order, as in [5]. In this setting, the labels are memory addresses and nodes can be accessed arbitrarily by memory

address. Memory addresses may be dereferenced and tested for equality. Therefore, the additional features that we need in our lower bound model are an array A containing the labels of S in some arbitrary order and a Boolean function **same** which checks two labels for equality. In this case, the query complexity of the tester also includes labels read from A. Note that these stronger testers do not need the **rand** oracle as they can simulate it by randomly reading a cell of the array containing S. For the lower bound, we also allow our testers full rank access. Note that the **before** function is thus redundant in this case.

As in the standard property testing model, we say an algorithm makes *one-sided error* if it always accepts lists which have the property. We say than an algorithm is *nonadaptive* if it decides all queries to make before learning the results of any queries.

A *violation* of sortedness is a pair of nodes (s_1, s_2) such that $\mathbf{rank}(s_1) < \mathbf{rank}(s_2)$ and $\mathbf{value}(s_2) < \mathbf{value}(s_1)$. A *local violation* is a violation (s_1, s_2) such that $\mathbf{next}(s_1) = s_2$.

3 Lower Bound

We define an *arbitrary-access* tester to be a tester with access to the **next**, **less**, **rank**, and **same** functions as well as an array A containing the elements of S in arbitrary order. We define a *random-access* tester to be a tester with access to the **rand**, **next**, **less**, **rank**, and **same** functions. The following lemma shows that arbitrary-access testers are not any more powerful than random-access testers.

Lemma 3.1. *Given an arbitrary-access tester T_a with query complexity $q \leq n/2$, there exists a random-access tester T_r with query complexity $O(q)$.*

Proof. We define the random-access tester T_r by simulating the arbitrary-access tester T_a. In particular, we simulate the array A with an injective and surjective partial function M from indices into A to labels in S that we build incrementally throughout the simulation. Initially, M's domain is empty. When T_a requests access to some cell $A[i]$ and i is already in M's domain, T_r returns $M[i]$. Otherwise, T_r calls **rand** until it finds a label s that is not already in M's range (we need the **same** function here). Finding s requires only a constant expected number of calls to **rand** since $q \leq n/2$. T_r then adds the mapping from i to s into M. When T_a requests $\mathbf{next}(u)$ for some label u, T_r also calls $\mathbf{next}(u)$ to obtain a label v. If v is already in M's range, T_r simply returns v. Otherwise, it arbitrarily chooses an index i into A that is not yet in M's domain, adds the mapping from i to v into M, and finally returns v. In this way, T_r uses M to simulate A for some random ordering of the labels in A. Since the correctness of T_a holds for any ordering of the labels in A, T_r is also correct. □

Lemma 3.1 implies that a lower bound on the query complexity of random-access testers is also a lower bound on the query complexity of arbitrary-access testers. So we now restrict our attention to random-access testers. In fact, we further restrict our attention to *reasonable* testers. A tester is reasonable if it computes the rank of every node that it visits and immediately outputs **Reject** if it sees a violation—that is, if it calls $\mathbf{less}(s_1, s_2)$ for $s_1, s_2 \in S$ such that $\mathbf{rank}(s_2) < \mathbf{rank}(s_1)$. We can assume without loss of generality that the tester we are lower bounding is reasonable, since the calls to **rank** do not affect the query complexity, and making more queries after seeing a violation can only increase the tester's query complexity.

We will prove our lower bound via the typical approach of using Yao's minimax lemma. We define two distributions, $\mathcal{D}_{YES}$ and $\mathcal{D}_{NO}$, over inputs that are monotone and ϵ-far from monotone, respectively. We then show that any deterministic algorithm making too few queries cannot distinguish the two distributions with probability great than $1/3$.

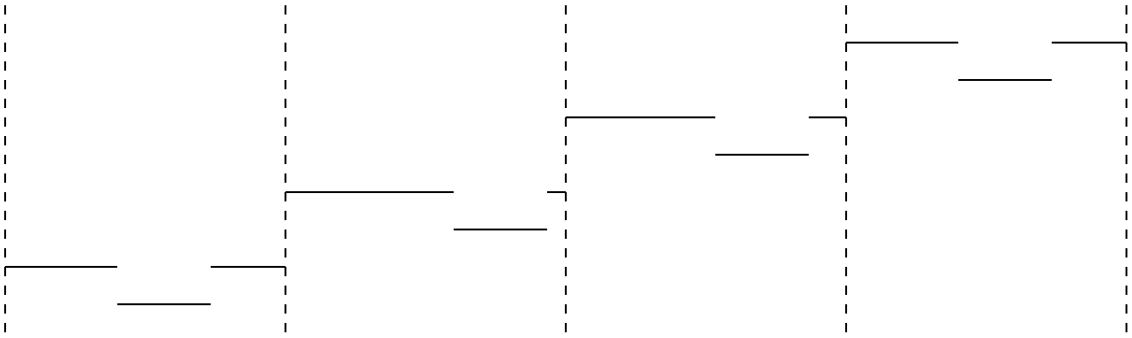

Figure 1: A representation of four blocks of a hard input where the x-axis represents node ranks and the y-axis represents node values.

The distribution $\mathcal{D}_{YES}$ is over a single monotone list defined as follows. Let $b = n^{2/3}$ and define an input on b "blocks," each of length $\frac{n}{b}$, where all the values in block $i \in [b]$ are equal to $2i$.

The distribution $\mathcal{D}_{NO}$ is defined similarly, except every block i has a contiguous sub-block of length $\frac{n}{3b}$ containing the value $2i-1$. See Figure 1 for an example. Note that there is only one local violation of sortedness (a node with value $2i$ pointing to a node with value $2i-1$) in each block. The distribution $\mathcal{D}_{NO}$ is defined by choosing the location of each local violation independently and uniformly at random amongst the nodes with ranks $\frac{n}{3b}$ to $\frac{2n}{3b}$ within the sublist defined by the block. Note that there are $\frac{n}{3b} = \frac{1}{3}n^{1/3}$ different possible locations for each local violation and there are no violations of sortedness whatsoever between blocks. It is easy to see that every input in $\mathcal{D}_{NO}$ is $1/3$-far from sorted.

The correctness of our lower bound, Theorem 1.1, then follows from the following lemma:

Lemma 3.2. *Fix an arbitrary, order-based tester $\mathcal{A}$ making at most $q \leq cn^{1/3}$ queries (where c is some absolute constant to be specified later). Then $\mathcal{A}$ cannot distinguish between $\mathcal{D}_{YES}$ and $\mathcal{D}_{NO}$ with probability greater than $1/3$.*

Proof. Assume $\mathcal{A}$ makes t calls to **rand**. Let the ranks of the resulting nodes be $r_1, r_2, \ldots, r_t$. We call this sequence of ranks given by **rand** the rank sequence. Fix an arbitrary rank sequence.

Although $\mathcal{A}$ may be adaptive, since it is reasonable and order-based, for a fixed rank sequence its behavior is identical for all inputs, unless it discovers two nodes in the same block, one with value $2i$ and the other with value $2i-1$. We will bound the probability that this happens.

The nodes that $\mathcal{A}$ visits can be specified by $K_1, K_2, \ldots, K_t$, where K_i is one plus the number of times $\mathcal{A}$ iteratively calls **next** starting from the node with rank r_i. Thus, $q = \sum_i K_i$ is the query complexity of $\mathcal{A}$.

First we analyze the probability that all the nodes $\mathcal{A}$ visits via **rand** calls lie in different blocks. Note that each block has size $\frac{n}{b}$, so if the ranks of the nodes returned by **rand** differ pairwise by at least that amount, this implies that they all lie in different blocks.

$$\begin{aligned}
\Pr[\forall_{i,j}|r_i - r_j| > n/b'] &= 1 - \Pr[\exists_{i,j}|r_i - r_j| \leq n/b] \\
&\geq 1 - \sum_{i<j} \Pr[|r_i - r_j| \leq n/b] \\
&= 1 - \sum_{i<j} 1/b \\
&\geq 1 - t^2/2b \\
&\geq 1 - q^2/2b \\
&\geq 1 - c^2/2
\end{aligned}$$

By setting c sufficiently small, we can make this probability arbitrarily close to 1 (say at least 0.9).

We now assume that each r_i is in a different block. In this case the only way that $\mathcal{A}$ may encounter a violation of sortedness is if one of the walks K_i crosses a local violation. Observe that the walk from the node with rank r_i can only reach the local violation in its block. Since the location of each such violation is chosen randomly from a set of size $\frac{1}{3}n^{1/3}$, the probability that the violation in the ith block overlaps with the walk K_i is at most $3n^{-1/3}K_i$. The probability that *none* of the violations overlap with any of the walks is at least $\prod_i(1 - 3n^{-1/3}K_i) \geq 1 - 3n^{-1/3}\sum_i K_i = 1 - 3c$. Again, we can make this probability arbitrarily close to 1 (say at least 0.9) by setting c sufficiently small. Thus, the overall probability that $\mathcal{A}$ never encounters a violation, and hence behaves the same on $\mathcal{D}_{YES}$ and $\mathcal{D}_{NO}$, is at least $0.8 > 2/3$. □

4 Upper Bound

The capabilities of the linked list model suggest two naive strategies for finding violations. One strategy is to perform walks along the linked list from random starting points, checking for local violations. The pseudocode for such a tester is given in Algorithm 1. Another simple strategy is to randomly sample a set of nodes and check them pairwise for violations of sortedness. The pseudocode for such a tester is given in Algorithm 2.

Our algorithm is a hybrid tester that uses both of these strategies in two phases. In the first phase, it repeats the walking strategy $3/\delta$ times with walks of length ℓ (where $\delta = \epsilon/2$). If no violation is found, it continues to the second phase. In the second phase, it performs the sampling strategy with a sample size of $\frac{100\sqrt{n/\ell}}{\epsilon}$. The pseudocode for our hybrid tester is given in Algorithm 3.

The query complexity of our algorithm is $O(\frac{\ell}{\epsilon} + \frac{\sqrt{n/\ell}}{\epsilon})$, which is minimized to $O(\frac{n^{1/3}}{\epsilon})$ when $\ell = n^{1/3}$.

We now analyze our algorithm for correctness. By the definition of the algorithm, it is clear that the algorithm has one-sided error, since it always accepts sorted lists. Thus, to prove Theorem 1.2, we need only show that the hybrid tester rejects lists that are ϵ-far from sorted with probability at least 2/3. To show this, our overall strategy will be to show that if a list is ϵ-far from sorted, either there are many local violations of sortedness, in which case the walking tester will detect one, or the list consists of few long *runs* (i.e., sorted sub-lists). In the latter case, we prove a structural theorem, showing that portions of the runs can be "paired up," in such a way that any two nodes, one from each member of a pair, form a violation. In that case, if we sample a node in each member of a pair, we have found a violation, and we can use a birthday paradox-style argument to reason about the number of random elements we must sample before we hit both members of a single pair.

Algorithm 1 Walking Tester

```
function WALK(ℓ, δ)
    for all i ∈ [3/δ] do
        u ← rand()
        for all j ∈ [ℓ] do
            v ← next(u)
            if less(v, u) then
                return Reject
            end if
            u ← v
        end for
        return Accept
    end for
end function
```

Algorithm 2 Sampling Tester

```
function SAMPLE(r)
    U ← ∅
    for all i ∈ [r] do
        U ← U ∪ {rand()}
    end for
    if U contains a violation then
        return Reject
    else
        return Accept
    end if
end function
```

Algorithm 3 Hybrid Tester

```
function MONOTONICITYTEST(ε, n)
    ℓ ← n^{1/3}
    if WALK(ℓ, ε/2) = Reject then
        return Reject
    else
        return SAMPLE(100√(n/ℓ) / ε)
    end if
end function
```

To prove our structural theorem, first we must argue about lists that do not have many local violations of sortedness. We define the notion of a *run*, or a sorted sub-list of a list L. A run of length ℓ is simply a set of ℓ elements with consecutive ranks that do not violated sortedness.

Lemma 4.1. *If the Walking Tester accepts a list of length n with probability at least $1/10$, then there exists a set of disjoint runs, all of length at least ℓ, covering at least $(1-\delta)n$ points in the list.*

Proof. For a list L, we say that a rank $i \in [n]$ is *bad* if a walk of length ℓ starting at rank i discovers a violation of L's sortedness. Let $B \subseteq [n]$ be the set of bad ranks. By definition, for a single walk performed by the walk tester, the probability that it outputs **Reject** is exactly $|B|/n$.

If $|B| > \delta n$, then after $3/\delta$ random walks, the probability that the tester accepts is at most $(1-\delta)^{3/\delta} < e^{-3} < 1/10$. Thus, we assume $|B| \leq \delta n$.

Define $G := [n] \setminus B$ to be the set of *good* ranks. By definition, all the points in G lie at the beginning of runs of length at least ℓ. So if we analyze the set $G' := \{r + i \mid r \in G \text{ and } i \in [\ell]\}$, it is clear that G' is the union of runs of length at least ℓ, and moreover $|G'| \geq |G| \geq (1-\delta)n$. □

Next we argue that if a list consists of many long runs, but is still far from sorted, then portions of these runs can be paired up in such a way that each pair includes a large number of violations.

We define a *balanced pair decomposition* as a set of pairs of sets of nodes, (A_1, B_1), (A_2, B_2), …, (A_t, B_t), such that all the A_i's and B_i's are disjoint, and each pair (A_i, B_i) satisfies the following properties:

- each $(u, v) \in A_i \times B_i$ is a violation of sortedness
- $|A_i| = |B_i|$

The size of a balanced pair decomposition is t, the number of balanced pairs, and the weight of a balanced pair decomposition is $\sum_{i=1}^{t}(|A_i| + |B_i|)$.

Lemma 4.2. *If L has distance d from sorted and consists of m maximal runs, then there is a balanced pair decomposition of L of size at most m and weight at least d.*

Proof. We proceed by induction on m. In the base case, $m = 1$ and a vacuous decomposition is sufficient as d must then be 0.

Otherwise, consider the first maximal run R. Let R_i be the ith node from the end of R. Let M_i be the set of nodes s such that $\mathbf{rank}(s) > \mathbf{rank}(R_1)$ and $\mathbf{value}(s) < \mathbf{value}(R_i)$. Let x be the largest index such that $x \leq |M_x|$. We create a pair (A, B) such that A contains $R_1, \ldots, R_x$ and B contains the x nodes with smallest values in M_x. We create a list L' by deleting the nodes of A and B from L.

If x happens to be the last index into R, deleting A from L deletes an entire maximal run. So, L has $m' < m$ maximal runs and distance $d' \geq d - 2x$ from sorted. By the induction hypothesis, there is a balanced pair decomposition of L' of size at most m' and weight at least $d' \geq d - 2x$. We obtain our desired decomposition by adding pair (A, B) to the decomposition of L'.

It remains to handle the case in which x is not the last index into R. Observe that by our selection of x, it must be that $x + 1 > |M_{x+1}|$ and $|M_{x+1}| \leq x$. So, B must contain all of M_{x+1}. Thus, the nodes of R that remain in L' have lesser values than all other nodes in L'. This means that the remainder of R fuses with the next maximal run remaining in L'. Again the number of maximal runs has reduced by at least one so the same inductive argument holds. □

Finally, we put the pieces together:

Proof of Theorem 1.2. It is obvious that if L is sorted, the tester accepts with probability 1. So we assume L is ϵ-far from sorted.

If the Walking Tester rejects with probability at least 9/10, we are done. So suppose the walking tester accepts with probability at least 1/10. In this case, we invoke Lemma 4.1 (with $\delta = \frac{\epsilon}{2}$) and conclude that there exists a sub-list L' of size at least $(1-\frac{\epsilon}{2})n$ such that L' consists of disjoint runs of length at least ℓ. Furthermore, since L was ϵ-far from sorted and L' was defined by removing at most $\frac{\epsilon n}{2}$ points from L, we know L' is $\frac{\epsilon}{2}$-far from sorted.

We now invoke Lemma 4.2 (with L' playing the role of L and $d = \frac{\epsilon}{2}$). Note that the runs all have length at most ℓ, so there are $m \leq \frac{n}{\ell}$ of them. Thus Lemma 4.2 implies that there is a decomposition of L' of size at most m and weight at least $\frac{\epsilon n}{2}$.

Write the decomposition of L' as (A_1, B_1), $\ldots$, (A_t, B_t) where $t \leq m$. Suppose we draw k random samples uniformly from L'. Conditioned on two samples falling in the same pair (A_i, B_i), the probability that we detect a violation is at least 1/2 (since A_i and B_i have the same size). Note also that, conditioned on k samples falling within *some* A_i or B_i, the probability that two of those samples fall into the same pair is minimized when $|A_i| + |B_i|$ is the same for all i (i.e., the uniform distribution minimizes collision probability, which can be proven using Schur-convexity), so we assume that each $|A_i| + |B_i|$ is equal for all i and that $t = m$. Using the fact that the weight of the decomposition is at least $\frac{\epsilon n}{2}$, we have that $|A_i| + |B_i| \geq \frac{\epsilon n}{2m} = \frac{\epsilon \ell}{2}$ for all i.

Now suppose the Sampling Tester makes r calls to **rand**. Note that the expected number of samples that fall in *some* A_i or B_i is at least $\frac{r\epsilon}{2}$. Let F be the event that fewer than $\frac{r\epsilon}{4}$ samples fall in some A_i or B_i, and let C be the event that no two samples fall in the same (A_i, B_i) pair. Then we have

$$\begin{aligned}
\Pr[C] &\leq \Pr[F] + \Pr[C|\bar{F}] \\
&\leq e^{-O(n)} + \prod_{i=0}^{r\epsilon/4}(1 - \frac{i}{m}) \\
&\leq e^{-O(n)} + e^{\sum_{i=0}^{r\epsilon/4} \frac{-i}{m}} \\
&\leq e^{-O(n)} + e^{-(r\epsilon/4)(r\epsilon/4-1)/m} \\
&\leq e^{-O(n)} + e^{-(r\epsilon/4-1)^2/m}
\end{aligned}$$

Setting $r > \frac{100\sqrt{m}}{\epsilon}$ makes $\Pr[C] < 0.1$. Thus the probability of detecting a violation is at least $\frac{1}{2}(1 - \Pr[C])$ and repeating the sampling test a constant number of times can boost this probability to 0.9. Finally, since the Walking Test and the Sampling Test both fail to find a violation with probability at most 0.1, by a union bound the total error probability is at most $0.2 < 1/3$.

□

Acknowledgements

We thank Constantinos Tsirogiannis for early discussions that led to this research, and Kristoffer Arnsfelt Hansen for the example illustrating the need for rank access. Kevin Matulef would also like to thank the IIIS at Tsinghua University for hosting him during the development of this work.

References

[1] E. Blais, J. Brody, and K. Matulef. Property testing lower bounds via communication complexity. *Comput. Complexity*, 21(2):311–358, 2012.

[2] D. Chakrabarty and C. Seshadhri. An o(n) monotonicity tester for boolean functions over the hypercube. In *Proc. 45th Annual ACM Symposium on the Theory of Computing*, pages 411–418, 2013.

[3] D. Chakrabarty and C. Seshadhri. Optimal bounds for monotonicity and Lipschitz testing over hypercubes and hypergrids. In *Proc. 45th Annual ACM Symposium on the Theory of Computing*, pages 419–428, 2013.

[4] D. Chakrabarty and C. Seshadhri. An optimal lower bound for monotonicity testing over hypergrids. In P. Raghavendra, S. Raskhodnikova, K. Jansen, and J. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization*, volume 8096 of *Lecture Notes in Computer Science*, pages 425–435. Springer Berlin Heidelberg, 2013.

[5] B. Chazelle, D. Liu, and A. Magen. Sublinear geometric algorithms. *SIAM J. Comput.*, 35(3):627–646, 2005.

[6] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual ACM Symposium on the Theory of Computing*, pages 365–372, 1987.

[7] Y. Dodis, O. Goldreich, E. Lehman, S. Raskhodnikova, D. Ron, and A. Samorodnitsky. Improved testing algorithms for monotonicity. In *Proc. 3rd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 97–108, 1999.

[8] F. Ergun, S. Kannan, R. Kumar, R. Rubenfeld, and M. Viswanathan. Spot-checkers. *J. Comput. Syst. Sci.*, 60:717–751, 2000.

[9] E. Fischer. On the strength of comparisons in property testing. *Info. and Comput.*, 189(1):107–116, 2004.

[10] E. Fischer, E. Lehman, I. Newman, S. Raskhodnikova, R. Rubinfeld, and A. Samorodnitsky. Monotonicity testing over general poset domains. In *Proc. 34th Annual ACM Symposium on the Theory of Computing*, pages 474–483, 2002.

[11] O. Goldreich. Introduction to testing graph properties. In O. Goldreich, editor, *Property Testing*, volume 6390 of *Lecture Notes in Computer Science*, pages 105–141. Springer Berlin Heidelberg, 2011.

[12] O. Goldreich and D. Ron. Property testing in bounded degree graphs. In *Proc. 29th Annual ACM Symposium on the Theory of Computing*, pages 406–415, 1997.

[13] T. Kaufman, M. Krivelevich, and D. Ron. Tight bounds for testing bipartiteness in general graphs. *SIAM J. Comput.*, 33(6):1441–1483, 2004.

[14] C. A. Kvalheim. Understanding MongoDB for performance and data safety. http://www.slideshare.net/mongodb/mongodb-london-2013understanding-mongodb-storage-for-performance-and-data-safety-by-christian-kvalheim-10gen.

[15] D. Ron. Property testing: A learning theory perspective. *Foundations and Trends in Machine Learning*, 1(3):307–402, 2008.

[16] R. Rubinfeld and A. Shapira. Sublinear time algorithms. *SIAM J. Disc. Math.*, 25(4):1562–1588, 2011.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de