Spot-Checkers*

Funda Ergün† Sampath Kannan† S Ravi Kumar‡
Ronitt Rubinfeld§ Mahesh Viswanathan†

Abstract

On Labor Day Weekend, the highway patrol sets up spot-checks at random points on the freeways with the intention of deterring a large fraction of motorists from driving incorrectly. We explore a very similar idea in the context of program checking to ascertain with minimal overhead that a program output is *reasonably* correct. Our model of *spot-checking* requires that the spot-checker must run asymptotically much faster than the combined length of the input and output. We then show that the spot-checking model can be applied to problems in a wide range of areas, including problems regarding graphs, sets, and algebra. In particular, we present spot-checkers for sorting, element distinctness, set containment, set equality, total orders, and correctness of group operations. All of our spot-checkers are very simple to state and rely on testing that the input and/or output have certain simple properties that depend on very few bits.

Our sorting spot-checker runs in $O(\log n)$ time to check the correctness of the output produced by a sorting algorithm on an input consisting of n numbers. We also show that there is an $O(1)$ spot-checker to check a program that determines whether a given relation is close to a total order. We present a technique for testing in almost linear time whether a given operation is *close to* an associative cancellative operation. In this extended abstract we show the checker under the assumption that the input operation is cancellative and leave the general case for the full version of the paper. In contrast, [RaS96] show that quadratic time is necessary and sufficient to test that a given cancellative operation is associative. This method yields a very efficient tester (over small domains) for all functions satisfying associative functional equations [Acz66]. We also extend this result to test in almost linear time whether the given operation is close to a group operation.

* This work was supported by ONR N00014-97-1-0505, MURI. The second author is also supported by NSF Grant CCR96-19910. The third author is also supported by DARPA/AF F30602-95-1-0047. The fourth author is also supported by the NSF Career grant CCR-9624552 and Alfred P. Sloan Research Award. The fifth author is also supported by ARO DAAH04-95-1-0092.

† Email: {fergun@saul, kannan@central, maheshv@gradient}.cis.upenn.edu. Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

‡ Email: ravi@almaden.ibm.com. IBM Almaden Research Center, San Jose, CA 95120.

§ Email: ronitt@cs.cornell.edu. Department of Computer Science, Cornell University, Ithaca, NY 14853.

1 Introduction

Ensuring the correctness of computer programs is an important yet difficult task. Program result checking [BK89] and self-testing/correcting programs [BLR93, Lip91] make runtime checks to certify that the program is giving the right answer. Though efficient, these methods often add small multiplicative factors to the runtime of the programs. Efforts to minimize the overhead due to program checking have been somewhat successful [BW94, Rub94, BGR96] for linear functions. Can this overhead be minimized further by settling for a weaker, yet nontrivial, guarantee on the correctness of the program's output? For example, it could be very useful to know that the program's output is reasonably correct (say, close in Hamming distance to the correct output). Alternatively, for programs that verify whether an input has a particular property, it may be useful to know whether the input is at least close to some input which has the property.

In this paper, we introduce the model of *spot-checking*, which performs only a small amount (sublinear) of additional work in order to check the program's answer. In this context, three prototypical scenarios arise, each of which is captured by our model. In the following, let f be a function purportedly computed by program P that is being spot-checked, and x be an input to f.

- *Functions with small output.* If the output size of the program is smaller than the input size, say $|f(x)| = o(|x|)$ (as is the case for example for decision problems), the spot-checker may read the whole output and only a small part of the input.

- *Functions with large output.* If the output size of the program is much bigger than the input size, say $|x| = o(|f(x)|)$ (for example, on input a domain D, outputting the table of a binary operation over $D \times D$), the spot-checker may read the whole input but only a small part of the output.
- *Functions for which the input and output are comparable.* If the output size and the input size are about the same order of magnitude, say $|x| = \Theta(|f(x)|)$ (for example, sorting), the spot-checker may only read part of the input and part of the output.

One naive way to define a weaker checker is to ask that whenever the program outputs an incorrect answer, the checker should detect the error with some probability. This definition is disconcerting because it does not preclude the case when the output of the program is very wrong, yet is passed by the checker most of the time. In contrast, our spot-checkers satisfy a very strong condition: if the output of the program is far from being correct, our spot-checkers output FAIL with high probability. More formally:

Definition 1 *Let $\Delta(\cdot,\cdot)$ be a distance function. We say that $\mathcal{C}$ is an* ϵ-spot-checker *for f with distance function Δ if*

1. *Given any input x and program P (purporting to compute f), and ϵ, $\mathcal{C}$ outputs with probability at least 3/4 (over the internal coin tosses of $\mathcal{C}$)* PASS *if $\Delta(\langle x, P(x)\rangle, \langle x, f(x)\rangle) = 0$ and* FAIL *if for all inputs y, $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) > \epsilon$.*
2. *The runtime of $\mathcal{C}$ is $o(|x| + |f(x)|)$*

The spot-checker can be repeated $O(\lg 1/\delta)$ times to get confidence $1 - \delta$. The choice of the distance function Δ is problem specific, and determines the ability to spot-check. For example, for programs with small output, one might choose a distance function for which the distance is infinite whenever $P(x) \neq f(y)$, whereas for programs with large output it may be natural to choose a distance function for which the distance is infinite whenever $x \neq y$.

OUR RESULTS. We show that the spot-checking model can be applied to problems in a wide range of areas, including problems regarding graphs, sets, and algebra. We present spot-checkers for sorting, element distinctness, set containment, set equality, total orders, and group operations. All of our spot-checker algorithms are very simple to state and rely on testing that the input and/or output have certain simple properties that depend on very few bits; the non-triviality lies in the choice of the distribution underlying the test. Some of our spot-checkers run much faster than $o(|x| + |f(x)|)$ — for example, our sorting spot-checker runs in $O(\lg |x|)$ time. All of our spot-checkers have the additional property that if the output is incorrect even on one bit, the spot-checker will detect this with a small probability. In order to construct these spot-checkers, we develop several new tools, which we hope will prove useful for constructing spot-checkers for a number of other problems.

One of the techniques that we developed for testing group operations allows us to efficiently test that an operation is associative. Recently in a surprising and elegant result, [RaS96] show how to test that operation $\circ$ is associative in $O(|D|^2)$ steps, rather than the straightforward $O(|D|^3)$. They also show that $\Omega(|D|^2)$ steps are necessary, even for cancellative operations. In contrast, we show how to test that $\circ$ is *close* (equal on most inputs) to some cancellative associative operation $\circ'$ over domain D in $\tilde{O}(|D|)$ steps[1]. We also show how to modify the test to accommodate operations that are not known to be cancellative, in which case the running time increases to $\tilde{O}(n^{3/2})$. Though our test yields a weaker conclusion, we also give a self-corrector for the operation $\circ'$, i.e., a method of computing $\circ'$ correctly for *all* inputs in constant time. This method yields a reasonably efficient tester (over small domains) for all functions satisfying associative functional equations [Acz66].

RELATIONSHIP TO PROPERTY TESTING. A number of interesting result checkers for various problems have been developed (cf., [BK89, BLR93, EKS97, KS96, AHK95, Kan90, BEG$^+$91, ABC$^+$93]). Many of the checkers for numerical problems have used forms of *property testing* (albeit under various names) to ensure that the program's output satisfies certain properties characterizing the function that the program is supposed to compute. For example, efficient property tests that ensure that the program is computing a linear function have been used to construct checkers. In [GGR96], the idea of using property testing *directly on the input* is first proposed. This idea extended the scope of property testing beyond numeric properties. In [GGR96, GR97], property testing is applied to graph problems such as bipartiteness and clique number. The ideas in this paper are inspired by their work.

For the purposes of this exposition, we give a simplified definition of property testing that captures the common features of the definitions given by [RS96, Rub94, GGR96]. Given a domain H and a distribution $\mathcal{D}$ over H, a function f is ϵ-*close* to a function g over $\mathcal{D}$ if $\Pr_{x\in\mathcal{D}}[f(x) \neq g(x)] \leq \epsilon$. $\mathcal{A}$ is a *property tester* for a class of functions $\mathcal{F}$ if for any given ϵ and function f, with high probability (over the coin tosses of $\mathcal{A}$) $\mathcal{A}$ outputs PASS if $f \in \mathcal{F}$ and FAIL if there is no $g \in \mathcal{F}$ such that g and f are ϵ-close.[2]

Our focus on the checking of program results motivates a definition of spot-checkers that is natural for testing input/output relations for a wide range of problems. All previous property testers used a "Hamming-like" distance func-

[1] The notation $\tilde{O}(n)$ suppresses polylogarithmic factors of n.

[2] In fact, the definition of property testing given by [GGR96] is much more general. For example, it allows one to separately consider two different models of the tester's access to f. The first case is when the tester may make queries to f on any input. The second case is when the tester cannot make queries to f but is given a random sequence of $\langle x, f(x)\rangle$ pairs where x is chosen according to $\mathcal{D}$. In our setting, the former is the natural model.

tion. Our general definition of a distance function allows us to construct spot-checkers for set and list problems such as sorting and element distinctness, where the Hamming distance is not useful. In fact, with a proper distance function, all property testers in [GGR96] can be transformed into spot-checkers. One must, however, be careful in choosing the distance function. For instance, consider a program which decides whether an input graph is bipartite or not. Every graph is close to a graph that is not bipartite (just add a triangle), so property testing for nonbipartiteness is trivial. Thus, unless the distance function satisfies a property such as $\Delta(\langle x,y\rangle,\langle x,y'\rangle)$ is greater than ϵ when $y \neq y'$, the spot-checker will have an uninteresting behavior.

2 Set and List Problems

2.1 Sorting

Given an input to and output from a sorting program, we show how to determine whether the output of the program is close in edit-distance to the correct sorting of the input, where the edit-distance $\rho(u,v)$ is the number of insertions and deletions required to change string u into v. The distance function that we use in defining our spot-checker is as follows: for all x, y lists of elements, $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle)$ is infinite if either $x \neq y$ or $|P(x)| \neq |f(y)|$ and otherwise is equal to $\rho(P(x), f(y))/|P(x)|$. Since sorting has the property that for all x, $|x| = |f(x)|$, we assume that the program P satisfies $\forall x,\ |x| = |P(x)|$. It is straightforward to extend our techniques to obtain similar results when this is not the case. We also assume that all the elements in our unsorted list are distinct. (This assumption is not necessary for testing for the existence of a long increasing subsequence.)

In Section 2.1.2, we show that the running time of our sorting spot-checker is tight.

2.1.1 The Test

Our 2ϵ-spot-checker first checks if there is a long increasing subsequence in $P(x)$ (Theorem 2). It then checks that the sets $P(x)$ and x have a large overlap (Lemma 8). If $P(x)$ and x have an overlap of size at least $(1-\epsilon)n$, where $n = |x|$, and $P(x)$ has an increasing subsequence of length at least $(1-\epsilon)n$, then $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) \leq 2\epsilon$.

For $m = O((1/c)\lg 1/\delta)$ and $n = O(\lg 1/\delta)$, the algorithm presented in the figure checks if an input sequence A has a long increasing subsequence by picking random pairs of indices $i < j$ and checking that $A[i] < A[j]$. An obvious way of picking i and j is to pick i uniformly and then pick j to be $i+1$. Another way is to pick i and j uniformly, making sure that $i < j$. One can find sequences, however, that pass these tests even though they do not contain long increasing subsequences. The choice of distribution on the pairs i, j is crucial to the correctness of the checker.

```
Procedure Sort-Check(c, δ)
repeat m times
   choose i ∈_R [1, n]
   for k ← 0 ... lg i do
      repeat n times
         choose j ∈_R [1, 2^k]
         if (A[i − j] > A[i]) then return FAIL
   for k ← 0 ... lg(n − i) do
      repeat n times
         choose j ∈_R [1, 2^k]
         if (A[i] > A[i + j]) then return FAIL
return PASS
```

Theorem 2 *Procedure* Sort-Check(c,δ) *runs in* $O((1/c)\lg n \lg^2 1/\delta)$ *time, and satisfies:*

- *If A is sorted,* Sort-Check(c,δ) = PASS.
- *If A does not have an increasing subsequence of length at least $(1-c)n$, then with probability at least $1-\delta$,* Sort-Check(c,δ) = FAIL.

To prove this theorem we need some basic definitions and lemmas.

Definition 3 *The graph* induced *by an array A, of integers having n elements, is the directed graph G_A, where $V(G_A) = \{v_1,\ldots,v_n\}$ and $E(G_A) = \{\langle v_i, v_j\rangle \mid i < j$ and $A[i] < A[j]\}$.*

We now make some trivial observations about such graphs.

Observation 4 *The graph G_A induced by an array $A = \{v_1, v_2, \ldots, v_n\}$ is transitive, i.e., if $\langle u,v\rangle \in E(G_A)$ and $\langle v,w\rangle \in E(G_A)$ then $\langle u,w\rangle \in E(G_A)$.*

We shall use the following notation to define neighborhoods of a vertex in some interval.

NOTATION. $\Gamma^+_{(t,t')}(i)$ denotes the set of vertices in the open interval between t and t' that have an incoming edge from v_i. Similarly, $\Gamma^-_{(t,t')}(i)$ denotes the set of vertices between t and t' that have an outgoing edge to v_i.

It is useful to define the notion of a *heavy* vertex in such a graph to be one whose in-degree and out-degree, in every 2^k interval around it, is a significant fraction of the maximum possible in-degree and out-degree, in that interval.

Definition 5 *A vertex v_i in the graph G_A is said to be* heavy *if for all k, $0 \leq k \leq \lg i$, $|\,\Gamma^-_{(i-2^k,i)}(i)\,| \geq \eta 2^k$ and for all k, $0 \leq k \leq \lg(n-i)$, $|\,\Gamma^+_{(i,i+2^k)}(i)\,| \geq \eta 2^k$, where $\eta = 3/4$.*

Theorem 6 *A graph G_A induced by an array A, that has $(1-c)n$ heavy vertices, has a path of length at least $(1-c)n$.*

The theorem follows as a trivial consequence of the following:

Lemma 7 *If v_i and v_j $(i < j)$ are heavy vertices in the graph G_A, then $\langle v_i, v_j\rangle \in E(G_A)$.*

Proof. Since G_A is transitive, in order to prove the above lemma, all we need to show is that between any two heavy vertices, there is a vertex v_k such that $\langle v_i, v_k\rangle \in E(G_A)$ and $\langle v_k, v_j\rangle \in E(G_A)$.

Let m be such that $2^m \leq (j-i)$, but $2^{(m+1)} \geq (j-i)$. Let $l = (j-i)-2^m$. Let I be the closed interval $[j-2^m, i+2^m]$ with $|I| = (i+2^m)-(j-2^m)+1 = 2^m - l + 1$. Since v_i is a heavy vertex, the number of vertices in I that have an edge from v_i is at least $\eta 2^m - (\ \{(j-2^m)-i\}\ ) = \eta 2^m - l$. Similarly, the number of vertices in I, that are adjacent to v_j is at least $\eta 2^m - (\ \{j-(i+2^m)\}\ ) = \eta 2^m - l$.

Now, we use the pigeonhole principle to show that there is a vertex in I that has an incoming edge from i and an outgoing edge to j. By transitivity that there must be an edge from i to j. This is true if $(\eta 2^m - l) + (\eta 2^m - l) \geq |I| = 2^m - l + 1$. Since $\eta = 3/4$, this condition holds if $l \leq 2^{m-1}$.

Now consider the case when $l > 2^{m-1}$. In this case we can consider the intervals of size 2^{m+1} to the right of i and to the left of j and apply the same argument based on the pigeonhole principle to complete the proof. □

Proof. [of Theorem 2] Clearly if the checker returns FAIL, then the array is not sorted.

We will now show that if the induced graph G_A does not have at least $(1-c)n$ heavy vertices then the checker returns FAIL with probability $1-\delta$. Assume that G_A has greater that cn light vertices. The checker can fail to detect this if either of the following two cases occurs: (i) the checker only picks heavy vertices, or (ii) the checker fails to detect that a picked vertex is light. A simple application of Chernoff bound shows that the probability of (i) is at most $\delta/2$.

By the definition of a light vertex, say v_i, there is a k such that $|\Gamma^+_{(i,i+2^k)}(i)|$ (or $|\Gamma^-_{(i,i-2^k)}(i)|$) is less than $(3/4)2^k$. The checker looks at every neighborhood; the probability that the checker fails to detect a missing edge when it looks at the k neighborhood (v_j such that $i \leq j \leq i \pm 2^k$) can be shown to be at most $\delta/2$ by an application of Chernoff's bound. Thus the probability of (ii) is at most $\delta/2$. □

In order to complete the spot-checker for sorting, we give a method of determining whether two lists A and B (of size n) have a large intersection, where A is presumed to be sorted.

Lemma 8 *Given lists A, B of size n, where A is presumed to be sorted. There is a procedure that runs in $O(\lg n)$ time such that if A is sorted and $|A \cap B| = n$, it outputs* PASS *with high probability, and if $|A \cap B| < cn$ for a suitable constant c, it outputs* FAIL *with high probability.*

Remark: The algorithm may also fail if it detects that A is not sorted or is not able to find an element of B in A.

Proof. Suppose A is sorted. Then, one can randomly pick $b \in B$ and check if $b \in A$ using binary search. If binary search fails to find b (either because $b \notin A$ or A is wrongly sorted), the test outputs FAIL. Each test takes $O(\lg n)$ time, and constant number of tests are sufficient to make the conclusion. □

2.1.2 A Lower Bound for Spot-Checking Sorting

We show that any comparison-based spot-checker for sorting running in $o(\lg n)$ time will either fail a completely sorted sequence or pass a sequence that contains no increasing subsequence of length $\Omega(n)$. We do this by describing sets of input sequences that presents a problem for such spot-checkers. We will call these sequences *3-layer-saw-tooth inputs.*

We define k-layer-saw-tooth inputs (k-lst's) inductively. k-lsts take k integer arguments, $(x_1, x_2, \ldots, x_k)$ and are denoted by $lst_k(x_1, x_2, \ldots, x_k)$. $lst_k(x_1, x_2, \ldots, x_k)$ represents the set of sequences in $\mathbb{Z}^{x_1 x_2 \ldots x_k}$ which are comprised of x_k blocks of sequences from $lst_{k-1}(x_1, x_2, \ldots, x_{k-1})$. Moreover, if k is odd, then the largest integer in the i^{th} block is less than the smallest integer in the $(i+1)^{th}$ block for $1 \leq i < x_k$. If k is even, then the smallest integer in the i^{th} block is greater than the largest integer in the $i(i+1)^{th}$ block for $1 \leq i < x_k$. Finally to specify these sets of inputs we need to specify the base case. We define $lst_1(x_1)$ to be the set of sequences in $\mathbb{Z}^{x_1}$ which are increasing.

An example $lst_3(3,3,2)$ is:

$$\overbrace{\underbrace{7\ 8\ 9}_{\in\, lst_1(3)}\ \ 4\ 5\ 6\ 1\ 2\ 3}^{\in\, lst_2(3,3)}\ 16\ 17\ 18\ 13\ 14\ 15\ 10\ 11\ 12$$

Note that the longest increasing subsequence in $lst_3(i,j,k)$ is of lentgth ik and can be constructed by choosing one $lst_1(i)$ from each $lst_2(i,j)$.

We now show that $o(\lg n)$ comparisons are not enough to spot-check sorting using any comparison-based checker (including that presented in the previous section). Suppose, for contradiction, that there is a checker that runs in $f(n) = \Theta(\lg n/\alpha(n))$ time where $\alpha(n)$ is an unbounded, increasing function of n. Without loss of generality, the checker generates $O(f(n))$ index pairs $(a_1, b_1), \ldots (a_k, b_k)$, where the $a_l < b_l$ for $1 \leq l \leq k$ and returns PASS if and only if, for all l, the value at position a_l is less than the value at position b_l.

Lemma 9 *A checker of the kind described above must either* FAIL *a completely sorted sequence or* PASS *a sequence that contains no increasing sequence of length $\Omega(n)$.*

Proof. Maintain an array consisting of $\log n$ buckets. For each (a_l, a_l) pair generated by the checker, put this pair in the bucket whose index is $\lfloor \lg(b_l - a_l)\rfloor$. It follows that there is a sequence of $c\alpha(n)$ buckets (for some $c < 1$) such that the probability (over all possible runs of the checker) that one of the pairs falls in one of these buckets is at most c. Let these $c\alpha(n)$ buckets range from p to q. In other words, $q = p + c\alpha(n)$ and there are very few pairs (a, b) such that $b - a$ is between 2^p and 2^q.

Consider an input from $lst_3(i, j, k)$ with $i = 2^{p+d}$ and $j = 2^{c\alpha(n)-d}$ for some constant d, and $k = n/(ij)$. In order to detect that this input is not sorted, the checker must compare elements coming from different lst_1 blocks but within the same lst_2 block. In other words, the checker has to generate either an (a, b) pair such that $b-a$ falls in the low probability range (i.e., between i and ij), or one where a and b belong to two consecutive lst_1 blocks such that $0 < b-a < 2^k$. To generate such a pair the checker must generate an a that is close to the end of an lst_1 block. The probability of this is at most 2^{-d}.[3] Thus, even though this input has no increasing sequence of length more than $n/(2^{\Omega(\alpha(n))})$, the probability that the checker will return FAIL is less than a constant. □

2.2 Element Distinctness

Given oracle access to a multiset A of size n, can we check in sublinear time if A has $O(n)$ distinct elements? The test we propose is simple: randomly pick m elements from A and if all the elements are distinct, declare A has $O(n)$ distinct elements. Otherwise, fail A. How big does m have to be for such a test to work?

Our distance function captures the number of elements of the input set that need to be changed in order to make the output correct. Given multisets A, B, let $\rho(A, B)$ be the minimum number of elements that need to be inserted to or deleted from A in order to obtain B. If the program says "not distinct", then since x is trivially close to a nondistinct set, the distance can be set. Then $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle)$ is infinite if $P(x) \neq f(y)$, and $\rho(x, y)/|x|$ otherwise. The following lemma states our result.

Lemma 10 *For a given constant ϵ, element distinctness can be ϵ-spot-checked in $\tilde{O}(\sqrt{n})$ time.*

Proof. Consider a set with k distinct elements and the proposed test (assume $k \mid n$). If x_i denotes the probability of picking the i-th element, noting that $\sum_{i=1}^{k} x_i^2$ is minimized when $x_1 = \cdots = x_k = n/k$, the worst-case is to assume that each element occurs n/k times.

If m elements are sampled uniformly with replacement, then the probability that all are distinct is

$$\prod_{i=0}^{m-1} \left(1 - \frac{i}{k}\right) \leq \prod_{i=0}^{m-1} \exp\left(\frac{-i}{k}\right) = \exp\left(\frac{-1}{k} \sum_{i=0}^{m-1} i\right)$$
$$= \exp\left(\frac{-m(m-1)}{2k}\right)$$

We want this to be less than some constant. Simple manipulations yield condition $m \geq \Omega(\sqrt{k})$. Thus, if we need $k = \Omega(n)$, we need $m \geq \Omega(\sqrt{n})$. We can sort this sample in order to tell whether all elements are distinct, which adds an extra $\log m$ factor. □

2.3 Set Equality

Given two sets A and B of size n, we show how to ensure that $A \cap B$ is of size $\Omega(n)$. We use this to spot-check programs for set-equality.

Given sets A, B, let $\rho(A, B)$ be the minimum number of elements that need to be inserted to or deleted from A in order to obtain B. Then $\Delta(\langle (x_1, x_2), P(x_1, x_2)\rangle, \langle (y_1, y_2), f(y_1, y_2)\rangle)$ is infinite if either $y_1 \neq x_1$ or $P(x_1, x_2) \neq f(y_1, y_2)$, and otherwise is $\rho(x_1, x_2)/|x_1|$. The following lemma shows that set-equality can be spot-checked very efficiently.

Lemma 11 *Given lists A, B of size n and constant ϵ, there is an ϵ-spot-checker for set equality that runs in $\tilde{O}(\sqrt{n})$ time.*

Proof. For a constant k to be determined, the checker simply chooses $k\sqrt{n}$ elements at random from each list and spot-checks that the intersection of the samples has cardinality "close" to k^2 where "close" will be defined in the sequel. Notice that by hashing the two samples this checker can be made to run in $O(\sqrt{n})$ time with high probability.

To analyze the checker, consider first the case where $|A \cap B| = n$ (i.e., B is a permutation of A). For each element x_i let the random variable X_i be the indicator of the event that x_i occurs in both samples. $\Pr[X_i = 1] = (k\sqrt{n}/n)^2 = k^2/n$. Thus, $E[X_i] = k^2/n$. Letting X be the sum of the X_i, $E[X] = k^2$. Since the X_i are independent random variables, we can use Chernoff bounds to establish $\Pr[k^2 - X \geq \delta k^2] \leq \exp(-k^2\delta^2/2)$.

Now if $|A \cap B| < cn$, we are summing over cn X_i's instead of n. Thus the expected value of X is $k^2 c$. Once again Chernoff bounds imply that $\Pr[X - k^2 c \geq \delta k^2 c] \leq \exp(-k^2 c\delta^2/3)$.

We now need to choose k, c, and the threshold at which the checker outputs PASS. For any desired constant $c < 1$ set the threshold to be $k^2\sqrt{c}$. Corresponding to this threshold, set $\delta = (1 - \sqrt{c})$ in both inequalities above. Finally, k should be chosen so as to make the probability of wrong classification a small constant. This is achieved by choosing k such that $k^2 c\delta^2/3$ is bigger than l in order to achieve an an error of at most e^{-l}. □

3 Total Orders

In this section we show how to test whether a given relation "$\prec$" on the set $\{a_i \mid 1 \leq i \leq n\}$ is close to a total order. We represent the relation as a directed graph $H_\prec$ with vertex set

[3] This analysis assumes that the checker generates a uniformly. If not, one can consider not only the lst_3's described, but also concatenations of an increasing sequence to an lst_3 structure. Thus, one can eliminate the dependence on a in the sampling procedure.

$[n] = \{1, \ldots, n\}$, where $a_i \prec a_j$ iff (i, j) is an edge in $H_\prec$. We assume that i and j we can query whether $i \prec j$ or $j \prec i$ in unit time. Note that $\prec$ is a total order iff $H_\prec$ is acyclic.

Given an input x (a relation assumed to be represented as a graph), let $f(x)$ return TOTAL ORDER if x represents a total order (alternately, x is a directed acyclic graph) and NOT TOTAL ORDER if x is not, and let P be a program purporting to compute f. The distance function is defined as follows: $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle)$ is infinite whenever $P(x) \neq f(y)$ and is equal to the fraction of edges that need to be reversed to change x into y otherwise.

Though the problem of testing that a given graph is close to an acyclic graph seems similar to testing that a list has a long increasing subsequence, we show that it can be accomplished in constant time!

For any permutation π of $[n]$ let $D_\pi(H_\prec)$ denote the number of edges $(\pi(i), \pi(j))$ of $H_\prec$ such that $\pi(i) > \pi(j)$. In other words, D counts the number of edges that go *backward* with respect to the order induced by π. We quantify how far $H_\prec$ is is from being acyclic (or, equivalently, how far $\prec$ is from being a total order) by the function $D^*(H_\prec) = \min_\pi D_\pi(H_\prec)$. We also let π^* denote an ordering which achieves D^*. Without loss of generality we assume that the vertices are numbered in the order defined by π^*. We say that an edge (i, j) is *bad* if $i > j$. Otherwise we will say that the edge is *good*. Note that due to the numbering of the vertices, the goodness or badness of an edge is defined with respect to π^*.

The following fact about $H_\prec$ shows that $H_\prec$ cannot have too many bad edges with respect to π^*.

Observation 12 *For each i and for each $k > i$, at least half the edges between i and vertices in the interval $[i+1, k]$ must be good edges. Similarly, for each i and for each $k < i$, at least half the edges between the interval $[k, i-1]$ and i must be good edges.*

The above fact follows from the optimality of π^*, because otherwise we could change the position of i and get a better order. This fact also implies that at most half the edges in $H_\prec$ can be bad with respect to the optimal order.

The following fact links bad edges to cycles of length 3.

Corollary 13 *If for $i < j$, the edge between i and j is a bad edge (i.e., from j to i), then there is a $k \in [i+1, j-1]$ such that the edges between i and k and between k and j are good edges. Hence the* triangle (i, j, k) witnesses *the fact that $H_\prec$ contains a cycle (of length 3).*

Strictly more than half of the edges between i and the vertices in the interval $[i+1, j-1]$ between j and $[i+1, j-1]$ are good, because at least half of the edges between i (resp. j) and the interval $[i+1, j]$ (resp. $[i, j-1]$) are good, and (j, i) is bad. The above corollary follows by an application of the pigeonhole principle, and yields an $O(n)$ spot-checker: The mapping from bad edges to witness triangles described above is injective. Thus, the checker picks $O(n)$ sets of three vertices at random and outputs PASS if and only if none of the triangles forms a cycle. We now show how to obtain a constant time spot-checker.

Let B_i denote the set of vertices in $[i+1, n]$ that have bad edges from i. Let $G_i = [i+1, n] \backslash B_i$. By Observation 12, $|G_i| \geq |B_i|$.

We are now ready to state our main theorem for spot-checking total orders. First we describe the spot-checker: The spot-checker outputs PASS if $P(H_\prec) =$ NOT TOTAL ORDER. Otherwise, it picks a random sample consisting of a constant number of vertices and checks that the graph induces by $H_\prec$ on this sample is acyclic. If it is, the checker outputs PASS. Otherwise, it outputs FAIL.

Theorem 14 *The spot-checker described above is an ϵ-spot-checker for the total order problem.*

Proof. Let $H_\prec$ be such that $P(H_\prec) =$ TOTAL ORDER. If $H_\prec$ is acyclic, the spot-checker outputs PASS. Conversely, suppose the fraction of bad edges is at least c. There is a constant $c' = c/(2-c)$, and a set S, with $|S| \geq (c/2)n$, such that for all $i \in S$, $|B_i| \geq c'n$.

Call (i, x, y) a *witness-triple*, where $i \in S, x \in B_i, y \in G_i$ and $(y, x) \in H_\prec$. Since we have $(x, i), (i, y) \in H_\prec$, locating a witness-triple is tantamount to causing the spot-checker to output FAIL.

For $i \in S$, we have $|B_i| \geq c'n$. We now consider the interaction between B_i and G_i for an $i \in_R S$. The outline of the argument is: first, if most edges between G_i and B_i in $H_\prec$ go from G_i to B_i, then the spot-checker detects witness-triples with constant probability. If this does not occur, then, most edges *must* go from B_i to G_i. We then argue that this scenario violates the optimality assumption of the order. Hence, the former case should indeed occur and thus witness-triples are detected with constant probability.

Suppose at least k_0 fraction of edges between G_i and B_i are pointed from G_i to B_i. (We will fix k_0 later.) The spot checker looks at a constant-sized sample of the vertices. Since $|S| \geq (c/2)n$, the probability that the spot-checker hits S is at least $c/2$. Since $|G_i| \geq |B_i| \geq c'n$, for each $i \in S$, the sample will also contain an $x \in B_i$ and a $y \in G_i$ with probability at least c'^2. Now, since k_0 fraction of edges go from G_i to B_i, and x and y are uniformly distributed in B_i and G_i respectively, with probability $k_0 c c'^2/2$, (i, x, y) is a witness-triple. (To boost the probability that the checker will pick a witness-triple by a factor of d, one has to increase the number of vertices proportional to $\log d$.)

Assume now that less than k_0 fraction of edges between G_i and B_i are pointing from G_i to B_i. Let $k_4 = |G_i|/|B_i|$. Thus, $1 \leq k_4 \leq 2(1 - 1/c)$. Fix k_2 to be $1/(48k_4)$, and pick k_1 such that $12/(1-k_2) < k_1 < (1-k_2)/(2k_2k_4)$. Finally let k_0 be such that $k_0 \leq c'^2 k_2/k_1$.

Call $x \in B_i$ *typical* if at most $|B_i|/k_1$ edges from G_i are directed to x. Observe that at least $(1 - k_2)$ fraction of the vertices of B_i are typical, for otherwise, the number of edges from G_i to B_i is at least $(k_2|B_i|) \cdot (|B_i|/k_1) \geq$

$(c'^2 k_2/k_1)n^2 \geq k_0 n^2$, which is a contradiction since it violates the assumption about the fraction of the edges between G_i and B_i that point from G_i to B_i.

In the list of vertices that succeed i in the optimal ordering, consider the vertex j such that there are $3|B_i|/k_1$ vertices from G_i between i and j. Let $1 - k_3$ be the fraction of typical vertices between i and j. The two cases are:

[$k_3 > 3/4$:] In this case, we claim that by moving all the vertices in B_i (without disrupting the ordering among them) ahead of all the vertices in G_i, we can cut down the number of bad edges, thus contradicting the optimality of the ordering. We now analyze the number of bad edges eliminated and added by this operation. This operation must add new bad edges from the following possibilities: (i) all the edges between G_i and $k_2|B_i|$ non-typical vertices could become bad; by counting, we have at most $k_2|B_i||G_i|$ of them, and (ii) for the $(1-k_2)|B_i|$ typical vertices, the edges that were originally pointed from G_i could turn bad; by counting, we have at most $(1-k_2)|B_i||B_i|/k_1$ of them. This operation may eliminate bad edges as per the following: for at least $k_3(1-k_2)|B_i|$ typical vertices, at least $2|B_i|/k_1$ of the edges that were originally bad (i.e., pointing from these typical vertices back to vertices in G_i that preceded them) turn good; by counting, the number of bad edges eliminated is at least $(2|B_i|/k_1) \cdot k_3(1-k_2)|B_i|$. By our choice of k_1, $k_1 \leq (1-k_2)/(2k_2k_4)$, and by our assumption that $k_3 > 3/4$ the new ordering has fewer bad edges.

[$k_3 \leq 3/4$:] In this case, we show that one can relocate i just after j to reduce the number of bad edges, contradicting the optimality of the ordering. The number of new bad edges added by this relocation is at most $3|B_i|/k_1$ while the number of bad edges eliminated is $\geq (1-k_3)(1-k_2)|B_i| \geq (1-k_2)|B_i|/2$. Since $k_1 \geq 12/(1-k_2)$, and $k_3 \leq 3/4$, the net change in the number of bad edges is negative. $\square$

4 Groups

Suppose we are given a program P purporting to compute a group operation f as follows. On input a finite set G, P and f output the tables for operations $\circ$ and $\diamond$ respectively on G. Let $P_G(x,y)$ (resp. $f_G(x,y)$) denote the (x,y) entry from the table produced by P (resp. by f) on G. Assume that $\circ$ is known to be cancellative i.e., $(a \circ c = b \circ c) \Rightarrow a = b$ and $(a \circ b = a \circ c) \Rightarrow b = c$. In particular, cancellativity is a necessary but not sufficient condition for an operation to be a group. This assumption is not required for our checker to work; however, it helps greatly to simplify the tests and the proofs. In Section 4.3 we sketch briefly how to handle the case when $\circ$ is not known to be cancellative. We use the following distance function: $\Delta(\langle G, P_G\rangle, \langle H, f_H\rangle)$ is infinite if $G \neq H$ and is $\Pr_{a,b\in G}[P_G(a,b) \neq f_H(a,b)]$ otherwise. We present a method for checking very efficiently whether $\circ$ is close to $\diamond$. Though the output of P is of size $O(|G|^2)$, for any given distance ϵ our checker runs in $\tilde{O}(|G|/\epsilon)$ time.

The most interesting and challenging part of checking whether a given operation is close to a group is checking for associativity. The first $o(|G|^3)$ algorithm for checking if $\circ$ is associative is given in In [RaS96]. In particular, their randomized algorithm runs in $O(|G|^2)$ steps for cancellative operations. They also give a lower bound which shows that any randomized algorithm required $\Omega(|G|^2)$ steps to verify associativity, even in the cancellative case. Despite this lower bound, we show that one can check if $\circ$ "close" to an associative function table — i.e., if there is an associative operation which agrees with $\circ$ on a large fraction of $G \times G$ — in only $\tilde{O}(|G|)$ steps.

4.1 The Test

We check the following: (i) $\circ$ is close to some cancellative associative operation $\circ'$, (ii) $\circ'$ has an identity element, (iii) each element in G has an inverse under $\circ'$, and (iv) $\circ'$ is close to $\diamond$. We will show a way of computing $\circ'$ in constant time by making calls to $\circ$ for testing properties (ii) through (iv).

Our test has two stages. If P passes the first stage, checking (i) through (iii), then one can show the existence of a group operation $\circ'$ that differs from $\circ$ on at most 4ϵ fraction of $G \times G$. In the second stage we test (iv), whether $\circ$ is computing a *specific* group operation $\diamond$. We assume that $\diamond$ is available to the checker in terms of the values of $g \diamond a$, $\forall g \in S_G, a \in G$, where S_G is a set of generators of G. We note below that this representation has size $\tilde{O}(|G|)$.

Observation 15 *G has a set S_G of generators of size* $\lg |G|$.

We then note that for cancellative operations $\circ$, each row of the table for $\circ$ is a permutation of elements in G. Let $a \in_R G$ denote that a is distributed uniformly in G, and let $a, b \in_R G$ denote that a and b are picked independently from G according to the uniform distribution.

Observation 16 *If $\circ$ is cancellative, then for any $b \in G$, if $\alpha \in_R G \Rightarrow \alpha \circ b \in_R G$.*

ASSOCIATIVITY. For (i), we describe our check that the table for $\circ$ is associative. To do this, the checker repeats each of following checks several times (the number to be determined later), and fails the program if any one fails. All of the elements come from G.

(a) Pick random β, γ; check for all a that $a \circ (\beta \circ \gamma) = (a \circ \beta) \circ \gamma$.

(b) Pick random α, β; check for all c that $\alpha \circ (\beta \circ c) = (\alpha \circ \beta) \circ c$.

(c) Pick random $\alpha_1, \alpha_2, \alpha_3$; check that $(\alpha_1 \circ \alpha_2) \circ \alpha_3 = \alpha_1 \circ (\alpha_2 \circ \alpha_3)$.

If $\circ$ passes this test, then with high probability it must have the following properties:

(1) $\Pr_{\beta,\gamma}[\forall a,\ a \circ (\beta \circ \gamma) = (a \circ \beta) \circ \gamma] \geq 1 - \epsilon$,

(2) $\Pr_{\alpha,\beta}[\forall c,\ \alpha \circ (\beta \circ c) = (\alpha \circ \beta) \circ c] \geq 1 - \epsilon$, and

(3) $\forall a, \Pr_{(\alpha_1 \circ \alpha_2) \circ \alpha_3 = a}[\alpha_1 \circ (\alpha_2 \circ \alpha_3) = a] \geq 1 - \epsilon$.

Since our definition of a result-checker includes a confidence parameter, and since we have $O(|G|)$ probabilistic tests for each (1) and (2), the overall confidence δ has to be apportioned. Using Chernoff bounds it can be shown that it is sufficient to repeat each test $O(\lg(|G|/\delta)) = \lg|G| + \lg(1/\delta)$ times. For (3), we need to make sure that we have a similar number of samples for each a. Assume that we have $|G|$ buckets, one for each $x_i \in G$, $i = 1 \ldots |G|$. Upon picking $\alpha_1, \alpha_2, \alpha_3$, we store them in the bucket corresponding to the result of the operation $\alpha_1 \circ (\alpha_2 \circ \alpha_3)$. Using a coupon-collector type argument, one can show that, with high (desired) probability, after $\tilde{O}(|G|)$ iterations, each bucket will contain at least a given constant number of triples.

The following theorem states that the above properties are sufficient to conclude that $\circ$ is close to being a group operation. We present the proof in the next section.

Theorem 17 *Let $\epsilon < 1/15$. If $\circ$ is cancellative and satisfies (1) through (3) above, then there is a cancellative associative $\circ'$ (computable using $O(1)$ evaluations of $\circ$) satisfying*

1. $\forall b, \Pr_\alpha[\alpha \circ' b = \alpha \circ b] \geq 1 - 4\epsilon$
2. $\forall a, \Pr_\beta[a \circ' \beta = a \circ \beta] \geq 1 - 4\epsilon$.

In fact, we know how to construct $\circ'$ efficiently and correctly with high probability. For $a, b \in G$, let

$$a \circ' b = \operatorname*{maj}_{\beta \circ \gamma = b} \{(a \circ \beta) \circ \gamma\}.$$

The intuition behind taking a majority vote is that if $\circ$ were associative, we would have $(a \circ \beta) \circ \gamma = a \circ (\beta \circ \gamma) = a \circ b$. By defining $\circ'$ to be a majority over all $\beta \circ \gamma = b$, the "non-associativity" in $\circ$ is amortized. To compute $\circ'$ efficiently, given any $b \in G$, we need to be able to produce many β and γ such that $\beta \circ \gamma = b$. We can assume that a sufficient (at least a required constant) number of such pairs are available to us at no extra cost. This is true because while performing test (c), the checker can, in a similar way to the bucket argument about the test, store α_1, α_2 pairs in bucket labeled $\alpha_1 \circ \alpha_2$. Using the coupon-collector argument again, the samples collected for check (c) will, with high probability, provide a sufficiently large sample for each $b \in G$ so that $\circ'$ can be computed from them correctly with high probability. In fact, this estimate of $\circ'$ that we are computing is the *self-corrector* for $\circ'$ in terms of the original operation $\circ$. Note that the overhead for each such computation is only a constant (we use a constant number of samples from each bucket). From now on, we can assume $\circ'$ is available.

IDENTITY AND INVERSE. The following procedure shows how to test whether $\circ'$ induces an identity. For any element a, by cancellativity, there is a b such that $a \circ' b = a$. Then, b should be the identity e, if G were to be a group. That e is an identity can be verified in $O(|G|)$ time. Note that the cancellativity of $\circ'$ implies that e is unique.

Now, since $\circ'$ is cancellative, for every $a \in G$, there is a $b \in G$ such that $a \circ' b = e$. In other words, each $a \in G$ has an inverse and (iii) follows.

EQUALITY. Finally, we have to check if $\circ'$ is the same as $\diamond$, the specific group operation (*equality testing*, [BLR93, Rub94]). To do this in $|G| \lg |G|$ steps, check $\forall b \in G, g \in S_G$ if $g \circ' b = g \diamond b$, where the latter is given. To see that this uniquely identifies the group, we induct on $|b|$, the length of the string when b is expressed in terms of $\diamond$ and elements from S_G. Suppose for $k > 1$, $a = g_1 \diamond \cdots \diamond g_k$. Then, by induction $a \circ' b = (a' \diamond g) \circ' b = (a' \circ' g) \circ' b = a' \circ' (g \circ' b) = a' \circ' (g \diamond b) = a' \diamond (g \diamond b) = (a' \diamond g) \diamond b = a \diamond b$, where $a' = g_1 \diamond \cdots \diamond g_{k-1}, g = g_k$, the claim follows.

The required number of repetitions for identity, inverse, and equality tests can be derived using a similar argument to that involving the associativity test, as a result of which the following theorem ensues.

Theorem 18 *For $\epsilon < 1/15$, groupness has a 4ϵ-spot-checker that runs in $\tilde{O}(|G|)$ time.*

4.2 Associativity

NOTATION. For random variables a, b, c, we reason about the probability that $a = c$ by using an intermediate variable b, using $\Pr[a = c] \geq \Pr[a = b = c] \geq 1 - \Pr[a \neq b] - \Pr[b \neq c]$.

The rest of this section is dedicated to proving Theorem 17.

Proof. [of Theorem 17] The following series of lemmas establish the theorem. Lemma 20 shows that $\circ'$ is well-defined and Lemma 21 shows $\circ'$ is cancellative. Then, Lemma 22 shows that $\circ'$ agrees with $\circ$ on a large fraction of $G \times G$. Lemma 23 proves an intermediate step that is used in Lemma 24, which finally eliminates all probabilistic quantifiers. $\square$

The following lemma is an easy consequence of (3):

Lemma 19 *If $\beta_1, \gamma_1 \in_R G$ and $\beta_2, \gamma_2, \delta$ are such that $\beta_1 \circ \beta_2 = b = \gamma_1 \circ \gamma_2$, and $\delta \circ \beta_2 = \gamma_2$, then $\Pr[(\gamma_1 \circ \delta) = \beta_1] \geq 1 - \epsilon$.*

Proof. Note that β_2, γ_2, and δ exist by the cancellativity of $\circ$. Then, $\beta_1 \circ \beta_2 = b = \gamma_1 \circ \gamma_2 = \gamma_1 \circ (\delta \circ \beta_2) = (\gamma_1 \circ \delta) \circ \beta_2$, with the last step true with probability $1 - \epsilon$ by (3). Since $\circ$ is cancellative, we have $\beta_1 = \gamma_1 \circ \delta$. $\square$

First, we show $\circ'$ is well-defined. For a given $b \in G, \beta_1, \gamma_1 \in_R G$, let $\beta_2, \gamma_2, \delta$ be such that $\beta_1 \circ \beta_2 = \gamma_1 \circ \gamma_2$ and $\delta \circ \beta_2 = \gamma_2$. Note that $\gamma_1, \delta, \beta_2$ are pairwise independent random variables. Using Lemma 19 and (1), we have for given $a, b \in G$, $\Pr[(a \circ \beta_1) \circ \beta_2 = (a \circ (\gamma_1 \circ \delta)) \circ \beta_2 = ((a \circ \gamma_1) \circ \delta) \circ \beta_2 = (a \circ \gamma_1) \circ (\delta \circ \beta_2) = (a \circ \gamma_1) \circ \gamma_2] \geq$

$1-3\epsilon$. Since $\circ'$ is defined to be the majority over $\beta\circ\gamma = b$ of $\{(a\circ\beta)\circ\gamma\}$, by the standard collision argument, we obtain the following lemma.

Lemma 20 *For all $a, b \in G$, $\Pr_{\beta_1}[a \circ' b = (a \circ \beta_1) \circ \beta_2$, where $\beta_1 \circ \beta_2 = b] \geq 1 - 3\epsilon$.*

The following lemma shows that $\circ'$ is cancellative. This will be useful for the rest of the discussion.

Lemma 21 *If $a \circ' b = a \circ' c$, then $b = c$. If $a \circ' c = b \circ' c$, then $a = b$.*

Proof. Let $\beta \in_R G$. Let $\alpha_1, \alpha_2 \in G$ be such that $\alpha_1 \circ \beta = b$, $\alpha_2 \circ \beta = c$ and $(a\circ\alpha_1)\circ\beta = a\circ' b = a\circ' c = (a\circ\alpha_2)\circ\beta$ holds. Note that such $\alpha_1, \alpha_2, \beta$ exist by Lemma 20. Now, by the cancellativity of $\circ$, we have first $a \circ \alpha_1 = a \circ \alpha_2$ and next $\alpha_1 = \alpha_2$, thus finally $b = c$.

Let $\beta_1 \in_R G$. Let $\beta_2 \in G$ be such that $\beta_1 \circ \beta_2 = c$ and $(a\circ\beta_1)\circ\beta_2 = a\circ' c = b\circ' c = (b\circ\beta_1)\circ\beta_2$ holds. Note that such β_1, β_2 exist by Lemma 20. Now, by the cancellativity of $\circ$, we have $a \circ \beta_1 = b \circ \beta_1$ and hence $a = b$. □

The following lemma proves part of Theorem 17 — that $\circ'$ agrees with $\circ$.

Lemma 22 $\forall b, \Pr_\alpha[\alpha\circ' b = \alpha\circ b] \geq 1-4\epsilon$. $\forall a, \Pr_\beta[a\circ'\beta = a\circ\beta] \geq 1 - 4\epsilon$.

Proof. Let $\beta_1 \in_R G$ and β_2 be such that $\beta_1 \circ \beta_2 = b$. We have $\alpha, \beta_1 \in_R G$. $\Pr_{\beta_1}[\alpha \circ' b = (\alpha \circ \beta_1) \circ \beta_2 = \alpha \circ (\beta_1 \circ \beta_2) = \alpha \circ b] \geq 1 - 4\epsilon$, where the first equality follows from Lemma 20 and the second equality follows using (2).

Similarly, $\Pr_{\beta_1}[a \circ' \beta = (a\circ\beta_1)\circ\beta_2 = a\circ(\beta_1\circ\beta_2) = a \circ \beta] \geq 1 - 4\epsilon$, where the first equality follows from Lemma 20 and the second equality follows using (1). □

The following is a useful step in proving Theorem 17

Lemma 23 $\forall b, c, \Pr_{\beta_1,\gamma_1}[b \circ' c = (\beta_1 \circ (\beta_2 \circ \gamma_1)) \circ \gamma_2] \geq 1-4\epsilon$*, where β_2, γ_2 are such that $\beta_1\circ\beta_2 = b$ and $\gamma_1\circ\gamma_2 = c$.*

Proof. Using Lemma 20 and (1), we have $\Pr_{\beta_1,\gamma_1}[b \circ' c = (b\circ\gamma_1)\circ\gamma_2 = ((\beta_1\circ\beta_2)\circ\gamma_1)\circ\gamma_2 = (\beta_1\circ(\beta_2\circ\gamma_1))\circ\gamma_2] \geq 1 - 4\epsilon$. □

Finally, the following lemma shows $\circ'$ is associative, completing the proof of Theorem 17.

Lemma 24 *If $\epsilon < 1/15$, for all $a, b, c \in G$, $a \circ' (b \circ' c) = (a \circ' b) \circ' c$.*

Proof. Let $\beta_1, \beta_2 \in_R G$ and $\beta_2, \gamma_2 \in G$ be such that $\beta_1 \circ \beta_2 = b, \gamma_1 \circ \gamma_2 = c$. Then, it follows that $\beta_1, \beta_2 \circ \gamma_1 \in_R G$ and $\beta_2, \gamma_1 \in_R G$. Using Lemma 23, (1), and Lemma 20, we have $\Pr_{\alpha_2}[a \circ' (b \circ' c) = a \circ' ((\beta_1 \circ (\beta_2 \circ \gamma_1)) \circ \gamma_2) = (a \circ (\beta_1 \circ (\beta_2 \circ \gamma_1))) \circ \gamma_2 = ((a \circ \beta_1) \circ (\beta_2 \circ \gamma_1)) \circ \gamma_2 = (((a \circ \beta_1) \circ \beta_2) \circ \gamma_1) \circ \gamma_2 = ((a \circ \beta_1) \circ \beta_2) \circ' c = (a \circ' b) \circ' c] \geq 1 - 15\epsilon > 0$. The lemma follows since the probabilistic assertion is independent of α_2. □

Our result can be used to show that a class of functional equations is useful for testing program correctness over small domains. The class of functional equations that our results apply to are those satisfying the the associativity equation $F[F[x, y], z] = F[x, F[y, z]]$, which characterize functions of the form $F[x, y] = f(f^{-1}(x) + f^{-1}(y))$ where f is a continuous and strictly monotone function [Acz66].

4.3 Discarding the Cancellativity Assumption

In this section we sketch additional tests required when $\circ$ is not known to be cancellative. The general intuition is that even if the table for $\circ$ is not cancellative, it should still contain a reasonably "even" distribution of the elements of G. To ensure that, we require the conditions below, which are similar to those given in [KR98], for a small enough ϵ'. Note that since ϵ' will not be part of the error, we need to change the parameters in the previous tests to accommodate errors due to ϵ' in addition to those due to ϵ.

We check the following conditions random sampling:

(1) $\forall a, |\{a \circ b \mid b \in G\}| \geq (1 - \epsilon')n$.

(2) $\Pr_\beta[|\{a \circ \beta \mid a \in G\}| = n] \geq 1 - \epsilon'$.

(3) $\Pr_\alpha[|\{\alpha \circ b \mid b \in G\}| = n] \geq 1 - \epsilon'$.

Checking the first condition involves using the element distinctness algorithm of Section 2.2, which increases the running time to $\tilde{O}(n^{3/2})$. Using the above conditions and the tests implied by them, and modifying the proofs to accommodate the bias in distribution of elements due the the small probability of non-cancellative behavior, our spot-checker can be made to work even in the non-cancellative case.

References

[Acz66] J. Aczel. *Lectures on Functional Equations and their Applications*. Academic Press, 1966.

[AHK95] L. M. Adleman, M-D. Huang, and K. Kompella. Efficient checkers for number-theoretic computations. *Information and Computation*, 121(1):93–102, 1995

[ABC+93] S. Ar, M. Blum, B. Codenotti, and P. Gemmell. Checking approximate computations over the reals. *Proc. 25th Symposium on Theory of Computing*, pp. 786–795, 1993.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. *Proc. 33rd Foundations of Computer Science*, pp. 14–23, 1992.

[AS92] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Proc. 33rd Foundations of Computer Science*, pp. 2–13, 1992.

[BGR96] M. Bellare, J. Garay, T. Rabin. Batch verification with applications to cryptography and checking. *Proc. Latin American Theoretical Informatics 98*, Springer LNCS 1830:267–288, 1998. By the same authors Fast Batch Verification for modular exponentiation and digital signatures. Proceedings of Eurocrypt 98, Springer-Verlag LNCS, Editor K. Nyberg, 1998, to appear.

[BEG+91] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Proc. 32nd Foundations of Computer Science*, pp. 90–99, 1991.

[BK89] M. Blum and S. Kannan. Designing programs that check their work. *Proc. 21st Symposium on Theory of Computing*, pp. 86–97, 1989.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. of Computing and System Sciences*, 47(3):549–595, 1993.

[BW94] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. *Proc. 35th Foundations of Computer Science*, pp. 382–392, 1994.

[BW94] M. Blum and H. Wasserman. Reflections on the Pentium division bug. *Proc. 8th Intl. Software Quality Week*, 1994.

[CR92] E. Castillo and M.R. Ruiz-Cobo. *Functional Equations and Modeling in Science and Engineering*. Marcel Dekker Inc., 1992.

[EKR96] F. Ergün, S. Ravi Kumar, and R. Rubinfeld. Approximate checking of polynomials and functional equations. *Proc. 37th Foundations of Computer Science*, pp. 592–601, 1996.

[EKS97] F. Ergün, S. Ravi Kumar, and D. Sivakumar. Self-testing without the generator bottleneck. *sicomp*, to appear.

[GLR+91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. *Proc. 23rd Symposium on Theory of Computing*, pp. 32–42, 1991.

[GGR96] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Proc. 37th Foundations of Computer Science*, pp. 339–348, 1996.

[GR97] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Proc. 29th Symposium on Theory of Computing*, pp. 406–415, 1997.

[Kan90] S. Kannan. *Program Result Checking with Applications*. PhD thesis, U. of California at Berkeley, 1990.

[KR98] S. Ravi Kumar and R. Rubinfeld. Testing Abelian Group Operations. Manuscript.

[KS96] S. Ravi Kumar and D. Sivakumar. Efficient self-testing/self-correction of linear recurrences. *Proc. 37th Foundations of Computer Science*, pp. 602–611, 1996.

[Lip91] R. Lipton. New directions in testing. *Proc. DIMACS Workshop on Distr. Comp. and Cryptography*, pp. 191–202, 1991.

[Rub94] R. Rubinfeld. Robust functional equations with applications to self-testing/correcting. *Proc. 35th Foundations of Computer Science*, pp. 288–299, 1994.

[RS92] R. Rubinfeld and M. Sudan. Testing polynomial functions efficiently and over rational domains. *Proc. 3rd Symposium on Discrete Algorithms*, pp. 23–43, 1992.

[RS96] R. Rubinfeld and M. Sudan. Robust characterizations of polynomials and their applications to program testing. *SIAM J. on Computing*, 25(2):252–271, 1996.

[RaS96] S. Rajagopalan and L. Schulman. Verifying identities. *Proc. 37th Foundations of Computer Science*, pp. 612–616, 1996.

[Vai93] F. Vainstein. *Algebraic Methods in Hardware/Software Testing*. PhD thesis, Boston University, 1993.