The Query Complexity of Program Checking by Constant-Depth Circuits

V. Arvind* K. V. Subrahmanyam † N. V. Vinodchandran ‡

Abstract

We consider deterministic and randomized AC^0 program checkers for standard P-complete and NC^1-complete problems. Our focus is on the *query complexity* of the checker: i.e. the number of queries made by the checker to the program. We show that $\Omega(n^{1-\epsilon})$ is a lower bound on the query complexity of deterministic AC^0 checkers for these problems, for each $\epsilon > 0$, and inputs of length n. On the other hand, we design randomized AC^0 checkers of *constant* query complexity for these problems.

1 Introduction

In this paper, we study program result checking (in the sense of Blum and Kannan [BK95]) with AC^0 circuits as checkers and we focus on the number of queries made by the checker to the checked program. We term this parameter as the *query complexity* of the checker for the given problem. The query complexity appears to be an important parameter in the design of efficient program checkers, because a large query complexity can be a serious bottleneck for a checker that may otherwise be efficient.

We first formally define program checkers introduced in [BK95].

Definition 1 [BK95] *Let A be a decision problem, a program checker for A, C_A, is a (probabilistic) oracle algorithm that, given as oracle any program P (supposedly for A) that halts on all instances, and for any input instance x of A, and any positive integer k (the security parameter) presented in unary the following holds:*

1. *If P is a correct program, that is, if $P(x) = A(x)$ for all instances x, then with probability 1, $C_A^P(x,k)$=Correct.*
2. *If $P(x) \neq A(x)$ then with probability $\geq 1 - 2^{-k}$, $C_A^P(x,k)$=Incorrect.*

The probability is computed over the sequences of coin flips that C_A could have tossed.

*Institute of Mathematical Sciences, Chennai 600113, India (email: arvind@imsc.ernet.in)

†Chennai Mathematical Institute, Chennai 600 017, India (email: kv@cmi.ac.in)

‡University of Nebraska-Lincoln, Lincoln NE 68588-0115, U.S.A (email: vinod@cse.unl.edu). Work done while at Institute of Mathematical Sciences, Chennai and BRICS, Aarhus.

When we speak of AC^0 checkers we mean that the checker C_A is described by a (uniform) family of AC^0 circuits $\{C_n\}_{n\geq 0}$, where C_n is the checker for input size n. We will also consider the (stronger) notion of deterministic checkability. The decision problem A is said to be *deterministically* checkable if C_A in the above definition is a deterministic algorithm. Next we define the query complexity of AC^0 checkers.

Definition 2 *Let L be a decision problem that is* AC^0 *checkable. The* AC^0 *checker defined by the circuit family $\{C_n\}_{n\geq 0}$ is said to have* query complexity $q(n)$ *if $q(n)$ bounds the number of queries made the checker circuit C_n for any input $x \in \Sigma^n$.*

The seminal paper of Blum and Kannan [BK95] already initiates the study of parallel checkers. They give a deterministic CRCW PRAM constant-time (i.e. AC^0) program checker for the P-complete problem *LFMIS* (lex. first maximal independent set problem for graphs). Rubinfeld in [Rub96] makes a comprehensive algorithmic study of parallel program checkers: parallel checkers are designed in [Rub96] for various problems with emphasis on analyzing parallel time and processor efficiency. In particular, an AC^0 checker for the P-complete problem of evaluating straight-line programs is described in [Rub96]. However, in the context of the present paper, we note that the above-mentioned AC^0 checkers for P-complete problems described in [BK95, Rub96] have large query complexity (the number of queries is proportional to the input size). Indeed the AC^0 checkers in [BK95, Rub96] are deterministic, and as shown in the present paper, deterministic AC^0 checkers for standard P-complete problems must necessarily have large query complexity.

Regarding query complexity, we note that constant query checkers are mentioned in [AHK95] as a notion of practical significance. For instance, it is shown in [AHK95] that GCD has a constant query checker.

Blum, Luby, and Rubinfeld, in their paper on self-testing and self-correction [BLR93], define and study following notions of efficiency of self-testers and correctors. These notions are also applicable to program checkers. We state their definition as applied to program checkers and discuss how it is related to the query complexity of checkers.

Definition 3 [BLR93] *Let A be a decision problem and C_A be a program checker for A. Given a purported program P for A, and input x the* incremental time *is the running time of C_A not counting the time for oracle calls to the program P. The* total time *of the checker C_A given input x and program P is the running time of C_A including the time for calls to the program P. The checker C_A is called* different *if the incremental time is faster than the running time for any correct program for A. The checker C_A is called* efficient *if the total time is linear in the running time of P.*

In the above definition of efficiency the number of queries made by C_A to P is not directly measured. It is interesting to note that even when the checker C_A is both *incremental* and *efficient*, it could be making a non-constant number of queries to P.

Apart from these related studies, to the best of our knowledge, there is no explicit mention of query complexity of program checkers in the existing literature.

Our motivation for studying AC^0 checkers is two-fold. First, in a complexity-theoretic sense AC^0 represents the easiest model of parallel computation, and one aspect of program

checking is to try and make the checker as efficient as possible. In this sense AC^0 checkers can be seen as constant-time parallel checkers since they essentially correspond to constant-time CRCW PRAM algorithms. The second motivation rests on the main goal of this paper, namely, to study the query complexity of program checkers. It turns out that AC^0 yields a model of program checking that is amenable to lower bound techniques. The problems we consider are different suitably encoded versions of the *Circuit Value Problem* (henceforth CVP). Different versions of the CVP are known to be complete for different important complexity classes: the unrestricted version is P-complete and the CVP problem for circuits that are formulas (i.e. fanout of each gate is one) is NC^1-complete.

We obtain nontrivial lower bounds on the query complexity of deterministic AC^0 checkers for these circuit value problems: we show the lower bounds by first noting for the Parity problem that, by a result of Ajtai [Ajt83], $\Omega(n^{1-\epsilon})$ is a lower bound on the query complexity of deterministic AC^0 checkers for Parity. Since Parity is AC^0 many-one reducible to each considered circuit-value problem, the same lower bounds on query complexity can be shown to carry over. Thus, we get that for each $\epsilon > 0$, $\Omega(n^{1-\epsilon})$ is a lower bound on the query complexity of deterministic AC^0 checkers for these circuit-value problems.

In contrast, we design randomized AC^0 checkers of *constant* query complexity for these circuit-value problems. We outline the ideas involved: consider a deterministic AC^0 checker for the given circuit-value problem (e.g. the one described in [Rub96] for general straight-line programs). The query complexity of this checker is roughly the number of gates in the input circuit. Our randomized *constant query* AC^0 checkers for the circuit-value problems use ideas from the $PCP(n^3, 1)$ protocol for satisfiability [ALM+92]. Intuitively, it turns out that the number of probes into an NP proof by a PCP protocol corresponds to the number of queries that the checker needs to make for a given instance of the circuit-value problem. A difficulty in the checker setting (which does not arise in the $PCP(n^3, 1)$ protocol) is that the AC^0 checker needs to compute unbounded GF(2) sums. It cannot directly do this because Parity is not computable in AC^0. But we can get around this difficulty by using Rubinfeld's parallel checker for Parity [Rub96] which we note is already an AC^0 constant-query checker. Since Parity is AC^0 many-one reducible to the considered circuit-value problems, we can use the tested program for CVP to compute Parity and use Rubinfeld's checker as subroutine to check that the returned answer is correct. Thus we are able to design a constant-query AC^0 checker for CVP.

Since CVP is logspace many-one equivalent to checking feasibility of linear programs FLP, our result also implies the existence of a randomized NC^2 constant-query checker for FLP. In fact, the same is true for all P-complete problems that are logspace many-one equivalent to CVP.

As explained in [ALM+92], the PCP theorem has evolved from interactive proofs [GMR89] and program checking [BK95, BLR93]. In particular, ideas from the area of self-correcting programs play a role in the $PCP(n^3, 1)$ protocol for 3-SAT [ALM+92]. It is not surprising, therefore, that ingredients of the $PCP(n^3, 1)$ protocol find application in our result on program checking. Our emphasis on the query complexity of program checkers leads naturally to ideas underlying probabilistically checkable proofs.

We now describe the CVP problems and the encodings of their instances. Let C denote a boolean circuit over the standard base (of NOT, AND, and OR gates). We consider circuits

of fanin bounded by two. We will encode the circuit C as 4-tuples (g_1, g_2, g_3, t) where t is a constant number of bits to indicate the type of the gate labeled g_1, and g_2 and g_3 are the gates whose values feed into the gate labeled g_1. For uniformity, we can assume that NOT gates are also encoded as such 4-tuples, except that $g_2 = g_3$. Furthermore, we insist that in the encoding, the gate labels g_i be *topologically sorted* consistent with the *DAG* underlying the circuit C. Thus, in each 4-tuple (g_1, g_2, g_3, t) present in the encoding of C, it will hold that $g_1 > g_2$ and $g_1 > g_3$. This stipulation ensures that checking whether an encoding indeed represents a circuit can be done in AC^0. For a circuit C with n inputs let $C_g(x_1, x_2, \ldots, x_n)$ denote the value of the circuit C at gate g designated as the output gate. We now define the *circuit value problem* which is the decision problem that we shall be mainly concerned with in this paper: $\{(C, g, x_1, x_2, \ldots, x_n) \mid C_g(x_1, x_2, \ldots, x_n) = 1\}$. The circuit C is encoded as described above.

The above circuit value problem is known to be P-complete under projection reducibility (see [GHR95] for details). We denote this by CVP. If the input circuit is a formula (i.e. each gate of the input circuit has fanout at most 1) the corresponding circuit value problem is known to be NC^1-complete under projection reducibility (see [GHR95]); we denote this problem by FVP (for formula value problem). Notice that an AC^0 circuit can check if a given circuit is a formula or not. Finally, we make another important stipulation on the circuits that are valid inputs for all the circuit value problems that we consider: we insist that the *fanout* of each gate is bounded by two. Notice that this last restriction on the input circuits does not affect the fact that CVP remains P-complete (in fact, such a restriction already holds for the CVP in the standard P-completeness proof by simulating polynomial-time Turing machines). Also, observe that this extra stipulation on the input circuits can be easily tested in AC^0.

2 Deterministic AC^0 checkers

We first recall deterministic AC^0 checkers for Parity and the circuit value problem. These are essentially implicit in [BK95, Rub96], although not in the form stated below in which we emphasize the query complexity $n/\log^{O(1)} n$.

Proposition 4 *For each constant k,* $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ *has a deterministic* AC^0 *checker of query complexity $n/\log^k n$.*

Proof. Let P be an alleged program for the Parity function. First observe that the AC^0 checker can make parallel queries to P for $\mathrm{Parity}(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$. In order to verify that the program's value of $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ is correct the checker just has to verify that the answers to the queries for $\mathrm{Parity}(x_1, x_2, \ldots, x_i)$ are all locally consistent: $P(x_1, x_2, \ldots, x_{i+1}) = x_{i+1} \oplus P(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n-1$. The above verification can be easily done in parallel in AC^0 since query answers $P(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$ are available. This yields an AC^0 checker which makes $n-1$ queries.

In order to design a checker with the number of queries scaled down to $n/\log^k n$ notice that we can compute the parity of $\log n$ boolean variables in AC^0 by brute force. Thus, we can group the n input variables into $n/\log n$ groups of $\log n$ variables each, compute

the parity of each group again by brute force, and the problem boils down to checking the program's correctness for the parity of $n/\log n$ variables, which we can do as before with $n/\log n$ queries to the program. To reduce the query complexity to $n/\log^2 n$, we iterate the above process. We group the $n/\log n$ variables whose parity is to be checked, into groups of $\log n$ variables and compute the parity of each such group of $\log n$ variables in AC^0. We are then left with the task of checking the program's correctness for the parity of $n/\log^2 n$ variables, which we can do as before with $n/\log^2 n$ queries to the program. Iterating this process k times we get a query complexity of $n/\log^k n$. However in the process of reducing query complexity, the depth and size of the AC^0 checker increases. ■

Remark: Notice that the above result applies to checking iterated products over arbitrary finite monoids. The proof and construction of the checker is similar.

Indeed, the above idea of doing piece-wise "local testing" in order to check global correctness is already used in the deterministic CRCW PRAM checker in [BK95] for an encoding of the P-complete problem *LFMIS*, and also in [Rub96] for checking straight-line programs. Notice that the checker for general straight-line programs includes checking the circuit-value problem as a special case. We include a proof sketch below since it is the starting point for the main results of this paper.

Theorem 5 [Rub96] CVP *has a deterministic* AC^0 *checker.*

Proof. Let P be an alleged program for CVP and let $(C, g, x_1, \ldots, x_n)$ be an input instance for program P. The AC^0 checker first queries in parallel the program for $P(C, g, x_1, \ldots, x_n)$ for all gates $g \in C$. Next, for each tuple (g_1, g_2, g_3, t) in the circuit description C the checker verifies that the program's answers are consistent with the gate type. This is again done in parallel for each tuple. The checker must also validate the input by verifying that the tuples that describe the circuit indeed describe an acyclic digraph. This is made sure as described in our encoding of the instances of the CVP. It suffices to check that $g_1 > g_2$ and $g_1 > g_3$ for each tuple (g_1, g_2, g_3, t), which can be done in AC^0. This completes the proof. ■

It can be shown similarly that FVP has a deterministic AC^0 checker. We now turn to lower bounds on the query complexity of AC^0 checkers for CVP and FVP. We first observe the following property of languages L having deterministic AC^0 checkers.

Lemma 6 *Let L be a decision problem that is deterministically* AC^0 *checkable and has an* AC^0 *checker of query complexity $q(n)$. Then, for each $n > 0$, there is a nondeterministic* AC^0 *circuit that takes n input bits and $q(n)$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff $x \in L^{=n}$ (where $L^{=n}$ denotes the set of strings in L of length n.)*

The proof of the above lemma is clear from the definition of checking and so we omit it.

Observe that, by symmetry, such nondeterministic AC^0 circuits also exist for $\overline{L}$. We next recall a result due to Ajtai [Ajt83] on lower bounds for AC^0 circuits approximating Parity.

Theorem 7 [Ajt83] *For all constants k, c, and $\epsilon > 0$, there is no depth k circuit of size n^c that can compute* $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ *for more than a $1/2 + 2^{-n^{1-\epsilon}}$ fraction of the inputs. Thus, no nondeterministic* AC^0 *circuit with $O(n^{1-\epsilon})$ nondeterministic bits can compute* $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$.

Lemma 8 Parity *does not have deterministic* AC^0 *checkers that make* $O(n^{1-\epsilon})$ *queries for any* $\epsilon > 0$.

Proof. Assume that Parity has AC^0 checkers that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. From Lemma 6 it follows that for each $n > 0$, there is a nondeterministic AC^0 circuit that takes n input bits and $O(n^{1-\epsilon})$ nondeterministic bits and accepts $x \in \Sigma^n$ iff x has odd parity. This is impossible since it contradicts Ajtai's result (Theorem 7 above). ∎

Theorem 9 CVP *(likewise* FVP*) does not have deterministic* AC^0 *checkers that make* $O(n^{1-\epsilon})$ *queries for any* $\epsilon > 0$.

Proof. We prove it just for CVP. Notice that we can design an AC^0 circuit (call it C') such that given an instance $x \in \Sigma^n$ of Parity the AC^0 circuit produces an instance $(C, x_1, x_2, \ldots, x_n, g)$ of CVP such that $\mathrm{Parity}(x_1, x_2, \ldots, x_n) = 1$ iff it holds that $(C, x_1, x_2, \ldots, x_n, g) \in \mathrm{CVP}$. Moreover, the size of $(C, x_1, x_2, \ldots, x_n, g)$ is $O(n \log n)$, since C encodes the linear-sized circuit for Parity in the 4-tuple encoding we are using for CVP instances. Assume that CVP has an AC^0 checker that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. Combining the nondeterministic circuit given by Lemma 6 with the AC^0 circuit C', we get a nondeterministic AC^0 circuit that takes n input bits and $O(n^{1-\delta})$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff x has odd parity, for some suitable $\delta > 0$. This contradicts Lemma 8 and hence completes the proof. ∎

3 A constant query randomized AC^0 checker for CVP

We first recall the relevant definition and results from [BLR93] concerning the linearity test.

Definition 10 [BLR93] *Let* F *be* $\mathrm{GF}(2)$ *and* f, g *be functions from* F^n *to* F. *The relative distance* $\Delta(f, g)$ *between* f *and* g *is the fraction of points in* F^n *on which they disagree. If* $\Delta(f, g) \leq \delta$ *then* f *is said to be* δ*-close to* g.

Theorem 11 [BLR93] *Let* F *be* $\mathrm{GF}(2)$ *and* f *be a function from* F^n *to* F *such that when we pick* y, z *randomly from* F^n, $Prob[f(y) + f(z) = f(y + z)] \geq 1 - \delta$, *where* $\delta < 1/6$. *Then* f *is* 3δ*-close to some linear function.*

The theorem gives a linearity test that needs to evaluate f at only a constant number of points in F^n, where the constant depends on δ. If f passes the test then the function is guaranteed to be 3δ-close to some linear function. Given a function f guaranteed to be δ-close to a linear function $\hat{f}$, and x in the domain of f, let us denote by SC-$f(x)$ the value $f(x + r) - f(r)$, for a randomly chosen r in the domain of f. Then the above theorem guarantees that with high probability SC-$f(x)$ is equal to $\hat{f}(x)$.

Next we recall another lemma from [ALM+92] (as stated in [MR95, Lemma 7.13]). Given $\vec{a}$ in $\mathrm{GF}(2)^n$, the *outer product* $\vec{b} = \vec{a} \otimes \vec{a}$ is an $n \times n$ matrix over $\mathrm{GF}(2)$ such that $\vec{b}_{ij} = \vec{a}_i \vec{a}_j$.

Lemma 12 [ALM+92] *Let* $\vec{a} \in \mathrm{GF}(2)^n$ *and* $\vec{b}$ *be an* $n \times n$ *matrix over* GF(2). *Suppose* $\vec{b} \neq \vec{a} \otimes \vec{a}$, *then* $\mathrm{Prob}[\vec{r}^t(\vec{a} \otimes \vec{a})\vec{s} \neq \vec{r}^t\vec{b}\vec{s}] \geq 1/4$ *where* $\vec{r}$ *and* $\vec{s}$ *are randomly chosen from* $\mathrm{GF}(2)^n$.

A randomized AC^0 checker for Parity is sketched in [Rub96] based on the fact that Parity is random self-reducible. Additionally, we observe that this checker has constant query complexity and we will use it as subroutine in the checker for CVP. We can also design a constant query AC^0 checker for Parity (without using random self-reducibility). This checker is very similar to part of the Graph Isomorphism checker. Since our checker is entirely different (and [Rub96] gives only a sketch), we include a proof here to make the paper self-contained.

Theorem 13 [Rub96] Parity *has a randomized* AC^0 *checker of constant query complexity.*

Proof. Our AC^0 checker is very similar to the standard IP protocol for Graph Nonisomorphism. Let P be a purported program for Parity and x be an input from $\{0,1\}^n$. Under coordinate-wise GF(2) addition, notice that $\{0,1\}^n$ is an abelian group of size 2^n. The set $\mathcal{E}$ of even parity vectors in $\{0,1\}^n$ forms a subgroup of size 2^{n-1}. Now, observe that $\mathcal{E}$ has a standard group action on the whole of $\{0,1\}^n$: $\vec{a} \in \mathcal{E}$ maps $\vec{x} \in \{0,1\}^n$ to $\vec{a} + \vec{x}$. Fix two vectors $E = 0^n \in \mathcal{E}$ and $O = 10^{n-1} \notin \mathcal{E}$. The two equivalence classes of $\{0,1\}^n$ under this action are the sets $\mathcal{E}$ and $\mathcal{E} + O$ of even parity vectors and odd parity vectors respectively.

Notice that the input x has even parity iff E and x are $\mathcal{E}$-equivalent, and x has odd parity iff O and x are $\mathcal{E}$-equivalent.

Now, let P be a purported program for parity. If the output $P(x)$ is "odd", it is the same as the program asserting that x and E are not $\mathcal{E}$-equivalent. We can check this by making a single query with probability 1/2 of catching the error as follows: pick either x or E at random and call it $\vec{y}$. Pick a random element $\vec{a} \in \mathcal{E}$ and compute $\vec{a} + \vec{y}$. The checker queries the program for $\vec{a} + \vec{y}$ and outputs "pass" if and only if P answers "odd" if $y = x$ and "even" if $y = E$. Notice that this is exactly like the standard IP protocol for Graph Nonisomorphism.

If the output $P(x)$ is "even" then the checker follows the same algorithm as above with O replacing E and other appropriate changes.

It remains to describe a randomized AC^0 algorithm for generating elements of $\mathcal{E}$ uniformly at random. Seen as a vector space, $\mathcal{E}$ has a basis consisting of the following $n-1$ vectors, $b_1 = \{1,1,0,\ldots,0\}$,$b_2 = \{0,1,1,0,\ldots,0\}$,...,$b_{n-1} = \{0,0,\ldots,1,1\}$. In order to generate a random element of $\mathcal{E}$, the checker picks up a random vector k in $\{0,1\}^{n-1}$ and computes $\Sigma_i k_i b_i$. Although this involves a parity computation for each coordinate, notice that for each coordinate there are at most two non-zero terms (in known positions) in the parity sum to be computed. Thus, such a parity computation can be easily done in AC^0.

This completes the proof. ∎

Remark. The above result can be generalized to the equivalence problem under group actions, where the group can be sampled in AC^0 and there are a constant number of equivalence classes.

We are now ready to design the constant query checker for the CVP problem (also for FVP). We make use of ideas in the PCP$(n^3, 1)$ protocol for 3-SAT from [ALM+92]. A crucial point of departure from [ALM+92] is when the checker needs to compute the parity of a multiset of input variables and a product of input variables. To do this we use as subroutine the checker of Theorem 13. Another point to note is that all queries have to be valid instances of CVP (or FVP as the case may be), and they need to be generated in AC^0. The starting point is the deterministic AC^0 checker for CVP described in Theorem 5. Recall that given instance $(C, g, x_1, \ldots, x_n)$ the deterministic AC^0 checker queries the program for $(C, g_i, x_1, \ldots, x_n)$, for each gate g_i of C. Then it checks that the query answers are locally consistent for each gate. Let $y_1, y_2, \ldots, y_m$ be the query answers by P for the queries $(C, g_i, x_1, \ldots, x_n)$, $1 \leq i \leq m$, where C has m gates. The *unique correct* vector $y_1, y_2, \ldots, y_m$ is a satisfying assignment to the collection of all the gate conditions (each of which is essentially a 3-literal formula). The idea is to avoid querying explicitly for y_i's. Instead, using randomness the checker will make fewer queries for other inputs that encode the y_i's. More precisely, we need to encode $y_1, y_2, \ldots, y_m$ in a way that making a constant number of queries to the program (which is similar to a constant number of probes into a proof by a PCP protocol) can convince the AC^0 checker with high probability that $y_1, y_2, \ldots, y_m$ is consistent with all the gate conditions.

Theorem 14 *The* P*-complete problem* CVP *(likewise the* NC^1*-complete problem* FVP*) has a randomized* AC^0 *checker of constant query complexity.*

Proof. We describe the checker only for CVP (the checker for FVP is similar). Let P be a program for CVP and $(C, g, x_1, \ldots, x_n)$ be an input. Suppose C has m gates $g_1, \ldots, g_m$ w.l.o.g. assume $g_m = g$. The deterministic checker of Theorem 5 queries P for $(C, g_i, x_1, \ldots, x_n)$, for each g_i. It then performs a local consistency check to test the validity of $(C, g_m, x_1, \ldots, x_n)$. The new checker must avoid querying explicitly for each $y_i := (C, g_i, x_1, \ldots, x_n)$. Let t_i denote a GF(2) polynomial corresponding to the ith gate of the circuit C, where t_i is a polynomial in at most three variables (these three variables are from $\{x_1, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_m\}$). Define t_i such that it is zero iff the corresponding variables are consistent with the gate type of g_i. More precisely, let g be an AND gate with output a and inputs b and c then the polynomial corresponding to g is $a + bc$. Similarly, for an OR gate with output a and inputs b and c the polynomial is $a + b + c + bc$, and for a NOT gate with output a and input b it is $a + b + 1$.

The checker

The checker first computes $y_m = P(C, g, x_1, \ldots, x_n)$ by querying the program on input $(C, g, x_1, \ldots, x_n)$. It suffices for the AC^0 checker to verify that the following linear function $f(\vec{x}, \vec{y}, \vec{z}) = \Sigma_{i=1}^{m} t_i(\vec{x}, \vec{y}) z_i$ in the new variables $z_i, 1 \leq i \leq m$ is the zero linear function.

Notice that if this function is nonzero then the linear function $f(\vec{x}, \vec{y}, \vec{z})$ evaluated at a randomly chosen $\vec{z} := \langle z_1, \ldots, z_m \rangle$ would be nonzero with probability 1/2. If it were the zero function then it must be zero with probability 1.

Recall that the checker must compute this value by making a series of CVP queries that are generated in AC^0. Towards this end, we rewrite the above expression: $f(\vec{x}, \vec{y}, \vec{z}) = p(\vec{x}, \vec{z}) + q(\vec{y}, \vec{z}) + r(\vec{y}, \vec{z})$, where $p(\vec{x}, \vec{z}) = \Sigma_{i=1}^{n} \Sigma_{k=1}^{m} \chi_i^k x_i + \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} \Sigma_{k=1}^{m} \chi_{ij}^k x_i x_j$, $q(\vec{y}, \vec{z}) = \Sigma_{k=1}^{m} c_i * y_i$, and $r(\vec{y}, \vec{z}) = \Sigma_{(i,j) \in [m] \times [m]} c_{ij} * y_i y_j$.

Since the gates in the circuit have input at most 2, notice that if we write $f(\vec{x}, \vec{y}, \vec{z})$ as a sum of terms, then any term is either independent of the y_i's, a linear function of the y_i's, or is quadratic in the y_i's. Observe that we have, accordingly, written $f(\vec{x}, \vec{y}, \vec{z})$ in three parts. The first part $p(\vec{x}, \vec{z})$ is independent of $\vec{y}$. The part $q(\vec{y}, \vec{z})$ is linear in the y_i's, and the part $r(\vec{y}, \vec{z})$ is quadratic.

We have similarly written $p(\vec{x}, \vec{z})$ as a function that is quadratic in the x_i's. Here, χ_i^k is defined to be 1 if x_i appears in the polynomial t_k and if z_k is 1, and defined to be zero otherwise. Likewise χ_{ij}^k, is defined to be 1 if $x_i x_j$ appears in the polynomial t_k and z_k is 1, and defined to be zero otherwise. It follows that an $nm + n^2 m$ length Boolean vector representing each term in p can be obtained in AC^0.

Next, observe that coefficients c_i and c_{ij} in q and r depend upon gates g_i and g_j, the constant number of gates they feed into, the elements of $\vec{z}$ corresponding to these gates and a constant number of input bits. So computing each of these coefficients involves computing the parity of a *constant* number of Boolean variables. This can be done directly in AC^0.[1]

The checker will work as follows: It first picks a random $\vec{z}$. Then, using polynomially many random bits, the checker will construct a constant number of queries for the given program P. From the answers given by P the checker will compute bits $\tilde{p}$, $\tilde{q}$, and $\tilde{r}$. To complete the checking the checker evaluates $\tilde{p} + \tilde{q} + \tilde{r}$ and accepts P as correct on its input if and only if this sum is zero.

Finally we will argue that if P is incorrect on its input then $\tilde{p} + \tilde{q} + \tilde{r} \neq 0$ with constant nonzero probability.

We now describe below how the checker computes $\tilde{p}$, $\tilde{q}$, and $\tilde{r}$.

Computing $\tilde{p}$ Note that $p(\tilde{x}, \tilde{z})$ is the parity of $nm+n^2m$ Boolean variables. As noted above the value of these variables can be obtained in AC^0 given $\vec{x}$ and $\vec{z}$. The checker constructs a description of a canonical circuit for the parity of $nm + n^2 m$ variables, and queries the program on this input. Next the AC^0 checker checks the answer of P using the checker of Theorem 13 as subroutine. If the answer is wrong then with high probability the subroutine checker will reject the program as incorrect. Thus the AC^0 checker computes a value $\tilde{p}$ which is equal to $p(\tilde{x}, \tilde{z})$ with high (constant) probability. In the process of this computation, only a constant number of queries are made to P.

Notice that, unlike in [ALM+92], we have to deal with both $y_1, \ldots, y_m$ and $x_1, x_2, \ldots, x_n$ which occur in the polynomials t_i. The crucial difference between the x_j's and y_j's is that $x_1, x_2, \ldots, x_n$ are *bound* to the input values. Thus, computing the value of $\tilde{p}$ is a *parity computation* which the checker requires to get done. As explained above, this is done using the checker of Theorem 13 as subroutine.

Computing $\tilde{q}$ and $\tilde{r}$

To compute $\tilde{q}$ and $\tilde{r}$ the checker goes through the following steps.

1. It builds a circuit C_1 with new inputs $r_1, r_2, \ldots, r_m$ that computes the function $\Sigma_{i=1}^m y_i r_i$. Recall that y_i is the output of the ith gate of the input circuit C on input $x_1, x_2, \ldots, x_n$. Clearly, the encoding of C_1 can be generated by an AC^0 circuit from the encoding of C.

[1] It is easier to first conceive of a constant time CRCW PRAM algorithm for this task.

2. Similar to the above step, the checker builds a circuit C_2 with new inputs $r_{ij}, 1 \leq i, j \leq m$ that computes $\Sigma_{i=1}^{m}\Sigma_{j=1}^{m} y_i y_j r_{ij}$. It is clear that an encoding for C_2 can also be generated by an AC^0 circuit.

3. The checker verifies that the program's behavior on C_1 is a function that is δ-close to a linear function in the variables r_i, and the program's behavior on C_2 is a function that is δ-close to a linear function in the variables r_{ij}. This can be done as described in the previous section using Theorem 11. If either of the tests fails, the checker rejects the program as being incorrect.

4. As in the PCP protocol the checker performs a consistency check: Let $\Sigma_{i=1}^{m}\Sigma_{j=1}^{m} b_{ij} r_{ij}$ be the linear function to which C_2 is δ-close. The checker does a constant query test, and ensures with high probability that the matrix b_{ij} is the tensor product of $\vec{y}$ with itself.

   To do this the checker employs the test given by Lemma 12. For two randomly chosen vectors r_1, r_2 of length m it verifies that SC-$C_1(r_1) * $ SC-$C_1(r_2) =$ SC-$C_2(r_1 \otimes r_2)$

   Note that the tensor product can be computed in AC^0 and the checker needs to make a constant number of queries to P.

Having performed the linearity and consistency tests the checker evaluates q and r by self-correction. Let $\vec{c}$ denote the m-vector $c_1, c_2, \ldots, c_m$ and let $\vec{d}$ denote the $m \times m$-vector consisting of $c_{ij}, 1 \leq i, j \leq m$. The checker sets $\tilde{q}$ to SC-$C_1(\vec{c})$ and $\tilde{r}$ to SC-$C_2(\vec{d})$.

Correctness.

If P is correct for all inputs then with probability 1 the checker will pass P as correct. Suppose that P is incorrect on input $(C, g, x_1, \ldots, x_n)$ and let $P(C, g, x_1, \ldots, x_n) = b$. Let F be the event that the checker fails to detect the program as incorrect. It suffices to show that Prob$[F]$ is bounded by a constant strictly smaller than 1. Since the (constant query AC^0) checker has one sided error, using standard techniques we can amplify the success probability to any constant.

Let h_1 and h_2 be the designated output gates of circuits C_1 and C_2. Let $\hat{C}_1(r_1, \ldots, r_n)$ denote $P(C_1, h_1, r_1, \ldots, r_n)$, and $\hat{C}_2(r_{11}, \ldots, r_{nn})$ denote $P(C_2, h_2, r_{11}, \ldots, r_{nn})$. Let $\delta > 0$ be a small constant to be suitably chosen later. Denote by T the event that $\hat{C}_1(r_1, \ldots, r_n)$ is δ-close to a unique linear function $\Sigma_{i=1}^{m} \hat{y}_i r_i$ and $\hat{C}_2(r_{11}, \ldots, r_{nn})$ is δ-close to the unique linear function $\Sigma_{(i,j)\in[m]\times[m]} \hat{y}_i \hat{y}_j * r_{ij}$. Let $\hat{y} = \langle \hat{y_1}, \ldots, \hat{y_m} \rangle$. Denote by w the entire string of random bits used by the checker; in particular, $\vec{z}$ is a substring of w.

Since T is independent of w, in order to bound Prob$_w[F]$ it suffices to bound each of Prob$_w[F|T]$ and Prob$_w[F|\neg T]$ by a constant smaller than 1.

Case (i) Suppose T is false. Notice that Prob$_w[F|\neg T]$ is bounded above by the probability that the checker fails to reject P in steps 3 and 4. Now, by Theorem 11 and Lemma 12, this probability can be made smaller than any constant $\epsilon > 0$ for a suitably small constant δ. This is achieved by the checker repeating the tests in steps 3 and 4 a constant number of times.

(ii) Next, suppose T is true. We must now bound $\text{Prob}_w[F|T]$. Given that P is incorrect on its input, the function $f(\vec{x}, \hat{y}, \vec{z}) = \Sigma_{i=1}^{m} t_i(\vec{x}, \hat{y}) z_i$ is a nonzero linear function of the z_i's. Hence, $\text{Prob}_w[f(\vec{x}, \hat{y}, \vec{z}) = 1 \mid T] = \frac{1}{2}$. Now,

$$\begin{aligned}
\text{Prob}_w[F \mid T] &= \text{Prob}_w[F \ \& \ f(\vec{x}, \hat{y}, \vec{z}) = 1 \mid T] + \text{Prob}_w[F \ \& \ f(\vec{x}, \hat{y}, \vec{z}) = 0 \mid T] \\
&\leq \text{Prob}_w[F \ \& \ f(\vec{x}, \hat{y}, \vec{z}) = 1 \mid T] + 1/2 \\
&= \text{Prob}_w[f(\vec{x}, \hat{y}, \vec{z}) = 1 \mid T] * \text{Prob}_w[F \mid f(\vec{x}, \hat{y}, \vec{z}) = 1 \ \& \ T] + 1/2 \\
&= 1/2 * \text{Prob}_w[F \mid f(\vec{x}, \hat{y}, \vec{z}) = 1 \ \& \ T] + 1/2.
\end{aligned}$$

Now, observe that

$$\text{Prob}_w[F \mid f(\vec{x}, \hat{y}, \vec{z}) = 1 \ \& \ T] \leq \text{Prob}_w[\tilde{p} \neq p \mid T] + \text{Prob}_w[\tilde{q} \neq q \mid T] + \text{Prob}_w[\tilde{r} \neq r \mid T].$$

Since $\tilde{q}$ and $\tilde{r}$ are obtained by self-correction, we have $\text{Prob}_w[\tilde{q} \neq q \mid T] \leq 2\delta$, and $\text{Prob}_w[\tilde{r} \neq r \mid T] \leq 2\delta$. By Theorem 13, $\text{Prob}_w[\tilde{p} \neq p \mid T] \leq 1/2$. Thus, we get $\text{Prob}_w[F \mid f(\vec{x}, \hat{y}, \vec{z}) = 1 \ \& \ T] \leq 1/2 + 4\delta$.

Putting it all together, we finally get

$$\text{Prob}_w[F \mid T] \leq 1/2 * (1/2 + 4\delta) \ + \ 1/2.$$

Choosing $\delta < 1/16$ we can bound $\text{Prob}_w[F \mid T]$ by $7/8$.

This completes the proof. ∎

4 Concluding remarks and open problems

In this paper we have studied the query complexity of deterministic and randomized AC^0 program checkers for some standard P-complete and NC^1-complete problems and have shown a big difference in the respective query complexities for the considered problems. Notice that it is crucial in the proof of Theorem 14 to be working with a circuit-value problem whose instances have the following closure property: if $(C, g, x_1, x_2, \ldots, x_n)$ is an instance of the problem and C has m gates $g_1, \ldots, g_m$, then the circuit C' with m new boolean inputs $r_1, \ldots, r_m$ that computes $\oplus_{i=1}^{m} g_i r_i$ by putting a parity gate on top to evaluate the sum and fanin two AND gates to evaluate each $g_i r_i$ is an instance of the same circuit-value problem. This property clearly holds for the standard P-complete problem CVP and the NC^1-complete problem FVP.

On the other hand, consider the complexity class NL. It is known that the circuit value problem with instances $(C, g, x_1, x_2, \ldots, x_n)$ such that C is a *skew* circuit (i.e. C has all fanin two gates, and every AND gates has at least one of the x_i's as input) is NL-complete under constant-depth reductions [Ven88]. In the light of the above discussion, modify this problem to now consist of instances $(C, g, x_1, x_2, \ldots, x_n)$, such that C has a formula on top

whose inputs are either the x_i's or skew circuits with input $x_1, x_2, \ldots, x_n$. Call this circuit-value problem $\mathrm{CVP}_{\mathrm{nl}}$. Clearly, $\mathrm{CVP}_{\mathrm{nl}}$ is NL-hard under constant-depth reductions. Since $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$ it follows that $\mathrm{CVP}_{\mathrm{nl}}$ is in NL. Thus, it is NL-complete under constant-depth reductions. From the previous paragraph it is clear that the proof of Theorem 14 will go through for $\mathrm{CVP}_{\mathrm{nl}}$. Therefore, $\mathrm{CVP}_{\mathrm{nl}}$ is an NL-complete problem with a constant query AC^0 checker.

We next give an example of a suitable constant query checker for a P-complete problem that is not a circuit-value problem. We consider feasibility of linear programs (FLP), which is a P-complete problem. First we make a simple observation (which is a version of Beigel's trick [BK95]).

Observation 15 *If π_1 and π_2 are AC^0 many-one equivalent problems and if π_1 has a constant-query AC^0 checker then so does π_2. If π_1 and π_2 are NC^1 many-one equivalent and if π_1 has a constant-query AC^0 checker then π_2 has a constant-query NC^1 checker.*

Now, CVP is NC^1-reducible to FLP [GHR95]. Since CVP is P-complete under projections, it is clear that CVP and FLP are many-one equivalent under NC^1 reductions. Thus, from the above observation we get the following result.

Theorem 16 *The* P*-complete problem* FLP *has a randomized* NC^1 *checker of constant query complexity.*

Along similar lines it is possible to derive checkers for other complete problems for P, NL, and NC^1.

Finally, our main focus in this paper has been on query complexity rather than on the size of the circuits defining the checker itself. It would be interesting to investigate trade-offs between the time/space complexities of the checker and its query complexity.

Acknowledgments. We thank Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh for interesting discussions on AC^0 lower bounds for parity. For useful remarks we are grateful to Samik Sengupta and D. Sivakumar. We are grateful to the referees for their comments and suggestions that have helped improve the presentation.

References

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

[Ajt83] M. Ajtai. Σ_1^1 formulas on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the intractability of approximation problems. *Journal of the Association of Computing Machinery*, 45(3):501–555, 1992.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the Association of Computing Machinery*, 42:269–291, 1995.

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

[GHR95] R. Greenlaw, J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation.* Oxford University Press, New York, Oxford, 1995.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, New York, 1995.

[Rub96] R. Rubinfeld. Designing checkers for programs that run in parallel. *Algorithmica*, 15(4):287–301, 1996.

[Ven88] H. Venkateswaran. Circuit definitions of nondeterministic complexity classes. *Proc. 8th FSTTCS, Lecture Notes in Computer Science*, 338:175–192, 1988.