# Coherent Functions and Program Checkers *
(extended abstract)

Andrew Chi-Chih Yao
Department of Computer Science
Princeton University
Princeton, New Jersey 08544

## ABSTRACT

Functions whose value at one point can be interpolated from values at other points have occupied a special place in cryptography. In this paper we will develop a general concept of "coherence" to capture this property and show its applications to several models of program checking. The concept and techniques also have applications in Hellman's time-memory tradeoff for inverting random functions.

## 1 Introduction

Functions with the property that the value at one point can be interpolated from values at other points have occupied a special place in cryptography. Concepts like "self-reducibility" (Angluin and Lichtenstein [AL], Abadi, Feigenbaum, and Kilian [AFK]), "randomly testable" (Lipton [L]), and other related notions (Blum, Luby, and Rubinfeld [BLR]) which were raised for various purposes have captured some aspects of this quality. In this paper we will develop a general concept of "coherence", which is in some sense the minimum requirement for a function to have the property mentioned above, and show its applications to program checking and other subjects.

*This research was supported in part by the National Science Foundation under Grant CCR-8813283.

In Section 2, we illustrate the concept of coherence by considering a simple puzzle. In Section 3 a direct application of this concept is given for a time-memory tradeoff problem of Hellman [H] for inverting one-way functions. In Sections 4 and 5, we extend and develop the coherence concept to Turing machine models and comparison tree models, and show the connection with program checking (Blum and Kannan [BK]); the connection gives an approach to prove that certain computations do not have efficient program checkers. Section 6 concludes with some open problems.

## 2 A Puzzle

Let $N, m$ be positive integers. Consider the following game to be played by $A$ and $B$. There are $N$ boxes with lids $\text{BOX}_1$, $\text{BOX}_2$, $\cdots$, $\text{BOX}_N$ each containing a boolean bit. In the preprocessing stage, Player $A$ will inspect the bits and take notes using an $m$-bit pad. Afterwards, Player $B$ will ask Player $A$ a question of the form "What is in $\text{BOX}_i$?". Before answering the question, Player $A$ is allowed to consult the $m$-bit pad and take off the lids of an adaptively chosen sequence of boxes not including $\text{BOX}_i$. The puzzle is, what is the minimum number of boxes $A$ needs to examine in order to find the answer?

One strategy that $A$ may adopt is the following. Divide the boxes into $m$ consecutive groups each containing no more than $\lceil N/m \rceil$ members. In the preprocessing stage, $A$ records for each $1 \leq j \leq m$ a bit

$a_j$ which is the parity of all the bits in the $j$-th group. To answer a query "what is in BOX$_i$", $A$ can lift the lids of all the boxes (except BOX$_i$) in the group containing BOX$_i$, say group $j$, and compute the parity $b$ of these bits; clearly, the bit in BOX$_i$ must be equal to $a_j \oplus b$. With this strategy, $A$ never needs to lift more than $\lceil N/m \rceil - 1$ lids. Is this the best possible strategy?

Let us define the model precisely. Let $F_N$ be the set of all functions $f : \{1, 2, \cdots, N\} \to \{0, 1\}$. For convenience we represent each of the $2^N$ possible bit patterns of the boxes as a function $f \in F_N$; $f(i) = 1$ if and only if BOX$_i$ contains 1. An algorithm $G$ consists of a partition $\{D_\alpha \mid \alpha \in \{0,1\}^m\}$ of $F_N$ and a collection of $N2^m$ decision trees $T_{\alpha,i}$, $\alpha \in \{0,1\}^m, 1 \leq i \leq N$. Each $T_{\alpha,i}$ is a binary tree, in which every internal node is associated with a query BOX$_k$=? $(k \neq i)$ and has its two outgoing edges labeled by 0 and 1; each leaf contains a boolean output bit. We further require $T_{\alpha,i}$ to yield the correct output when it is used in the standard way to answer Player $B$'s question "what is in BOX$_i$?" when $\alpha$ is the $m$-bit string recorded on the pad. Define the *cost* of the algorithm, *cost*$(G)$, to be the maximum height of all the trees $T_{\alpha,i}$. Let $s(m, N) = \min_G \operatorname{cost}(G)$. The following theorem shows that the strategy described earlier is an optimal one.

**Theorem 1** Any algorithm must have cost at least $\lceil N/m \rceil - 1$.

**Corollary** $s(m, N) = \lceil N/m \rceil - 1$.

**Proof** Without loss of generality we can assume that $1 \leq m < N$. Consider any algorithm $\{D_\alpha, T_{\alpha,i}\}$. We will prove that for each $\alpha \in \{0,1\}^m$

$$|D_\alpha| \leq 2^{Nt/(t+1)}, \qquad (1)$$

where $t$ is the cost of the algorithm.

Let $\ell = \lfloor N/(t+1) \rfloor t + \max(0, N - \lfloor N/(t+1) \rfloor (t+1) - 1)$. It is easy to verify that $\ell \leq Nt/(t+1)$. To prove (1), we only need to demonstrate that every $f \in D_\alpha$ can be uniquely specified by $\ell$ bits. Set $J \leftarrow \emptyset$. Repeat the following step $\lfloor N/(t+1) \rfloor$ times: take the smallest $i \notin J$, use the tree $T_{\alpha,i}$ to ask exactly $t$ queries $f(j) =?$ where $j \notin J \cup \{i\}$ and find the value of $f(i)$, then enlarge $J$ to include $i$ and all these $j$'s. At this point $t \cdot \lfloor N/(t+1) \rfloor$ queries have been answered and at least $(t+1) \cdot \lfloor N/(t+1) \rfloor$ values of $f(j)$ have been determined; it takes at most $\max(0, N - (t+1)\lfloor N/(t+1) \rfloor - 1)$ more queries to find out the rest of the values of $f(j)$. Since the answer to the queries determines $f$ completely, $\ell$ bits suffice to specify $f$. This proves (1).

It follows from (1) that

$$2^N \leq \sum_\alpha |D_\alpha| \leq 2^m \cdot 2^{Nt/(t+1)},$$

and hence $N/m \leq t+1$. This proves Theorem 1. The corollary follows immediately. □

Let us view the above discussion more abstractly. Call a function $f$ *coherent* if, when given some hints ($m$ bits) about $f$, one can interpolate its value at any $w$ from the value of $f$ at other points efficiently ($t$ points). There is an implicit assumption that one can discuss how to interpret hints *before* $f$ is chosen. (See Beaver and Feigenbaum [BeaF] for a discussion of this assumption in a different context.) Which functions are coherent depend not only on $m, t$, but also the interpretation agreed upon.

We will extend this concept to the Turing machine model in Section 4. A function $f$ will be called *coherent* if, when given some hints (a finite number of bits to specify a machine $S$), one can interpolate any $f(w)$ from the value of $f$ at other points efficiently (using $S$). In this case, the different possible interpretations of hints correspond to the different ways of specifying a universal Turing machine. The new ingredients to be added are the Turing-uniformity condition imposed on algorithms and the probabilistic setting needed for application purposes. Before doing that, we first discuss in the next section a question for which the simple model defined in the present section can be directly applied.

## 3 Inverting Random Functions

In the context of cryptanalysis, Hellman [H] developed a method for finding the inverse of any function. We will be interested in the question whether his method is the best possible among procedures of similar nature.

To explain Hellman's method, suppose we are given a one-cycle permutation $f : \{1,2,\cdots,N\} \rightarrow \{1,2,\cdots,N\}$ as a "blackbox"; that is, the only way to get information about $f$ is to feed the blackbox inputs $x$ and get the value $f(x)$ as outputs. The goal is to preprocess the information about $f$ so as to be able to find the inverse $f^{-1}(y)$ fast for inputs $y$. His strategy is as follows. Write $\{1,2,\cdots,N\}$ as $\{x_0, x_1, \cdots, x_{N-1}\}$ such that $f(x_i) = x_{i+1 \bmod N}$. In the preprocessing stage, we experiment with the blackbox and store the sequence $x_{jN/m}$, $0 \leq j \leq m-1$ using $m \log_2 N$ bits. Afterwards, if we are given $y$, we can find $f^{-1}(y)$ by first computing $y_1 = f(y), y_2 = f(y_1), \cdots$ until we recognize some $y_r$ to be $x_{i+1}$, and then compute $z_1 = f(x_i), z_2 = f(z_1), z_3 = f(z_2), \cdots$ until we notice some $z_{r'+1}$ to be the same as $y$; clearly, $z_{r'}$ is $f^{-1}(y)$. Note that $r + r' = N/m$. Thus, using $m \log_2 N$ bits of *memory*, we can invert $f$ in *time* $N/m$.

If we allow $f$ to be arbitrary functions from $\{1,2,\cdots,N\}$ into $\{1,2,\cdots,N\}$, then it is not always possible to partition the points into disjoint chains of uniform length. Hellman [H] considered a certain strategy of storing $2m$ numbers $x_i, x_i', 1 \leq i \leq m$ such that there is a chain $z_{j+1} = f(z_j), j = 1,2,\cdots$ leading from $x_i$ to $x_i'$ for each $i$, and a way of trying to find $f^{-1}(y)$ with this information. He showed that, if $f$ is a random function uniformly chosen from all functions and if $x$ is uniformly chosen from $\{1,2,\cdots,N\}$, then the probability of this algorithm successfully finding $x$ from the value $y = f(x)$ in time $O((N/m)^2)$ is at least 0.63.

Are these memory-time tradeoffs the best possible? If we restrict algorithms to those based on storing endpoints of chains, Shamir and Spencer [SS] showed that $t = O(N/m)$ and $t = O((N/m)^2)$ are the best achievable for random 1-cycle permutations and random functions. What if we remove the restriction of basing algorithms on the chain-chasing principle?

Let us consider a general model for answering this question. The problem is similar to that considered in Section 2 except for the probabilistic setting. Let $H_N$ denote the set of all functions $f : \{1,2,\cdots,N\} \rightarrow \{1,2,\cdots,N\}$, and $q_N$ be a probability distribution over $H_N$. An algorithm $G$ consists of a partition of $\{D_\alpha \mid \alpha \in \{0,1\}^m\}$ of $H_N$ and a collection of $N$-ary decision trees $T_{\alpha,i}$, $\alpha \in \{0,1\}^m, 1 \leq i \leq N$; the internal nodes of $T_{\alpha,i}$ ask queries of the form $f(j) =?$ $(j \neq i)$, and the leaves contain answers $1 \leq k \leq N$. Let *cost*$(G)$ be the maximum height of the trees. Denote by $\eta(G, q_N)$ the probability that, for a random $x \in \{1,2,\cdots,N\}$ and a random $f$ distributed according to $q_N$, the algorithm $G$ with $y = f(x)$ as input will return $z$ satisfying $f(z) = y$. The following theorem can be obtained by an extension of the proof of Theorem 1. Let $q_N$ be either the uniform distribution over all 1-cycle permutations or the uniform distribution over $H_N$. Let $0 < c < 1$ be any fixed constant.

**Theorem 2** Any algorithm $G$ with $\eta(G, q_N) \geq c$ must satisfy $\text{cost}(G) = \Omega((N \log N)/m)$.

**Proof** Omitted from this extended abstract. □

Thus, Hellman's scheme is nearly optimal for random 1-cycle permutations, even allowing probabilistic algorithms. It remains an interesting open question to bridge the gap between the upper and lower bounds for the random function case.

## 4 Coherent Functions

Let $f$ be a function, and $\mathcal{M}$ be a class of algorithms, called *examiners*, which for every input $w$ try to guess the value of $f(w)$ by asking a series of questions $f(x_1) =?$, $f(x_2) =?$, $\cdots$, where the $x_i$'s are not allowed to include $w$. Informally, $f$ is *coherent* with respect to $\mathcal{M}$ if there exists an examiner in $\mathcal{M}$ which can correctly predict $f(w)$ for all $w$. The precise nature of the functions $f$ and the model of algorithms will depend on the specific applications. Note that whether a function is coherent, i.e. intuitively easy to interpolate, is different from the difficulty of computing the function.

We will define below in detail the concept of *coherent functions* in the Turing machine model. Extensions to other models will be discussed in Section 5.2.

An *examiner* $S$ is a multitape probabilistic polynomial time Turing machine with a special query tape. Given an input string $w \in \{0,1\}^*$, the oracle machine $S^A$ with any oracle set $A$ operates in the standard way, except that oracle $A$ will never be used to query whether $w \in A$.

Let $f$: $\{0,1\}^* \rightarrow \{0,1\}$ be any boolean function. Define $A_f$ to be the set of $w$ such that $f(w) = 1$. Write $S(f, w)$ as the random variable whose value is 1 if $S^{A_f}$ accepts $w$ and 0 otherwise. Let $\alpha(S, f, w) = \Pr\{S(f, w) \neq f(w)\}$, and $\alpha(S, f) = \max_w \alpha(S, f, w)$.

**Definition** Let $\epsilon \geq 0$. We say that $S$ is $(1-\epsilon)$-*successful on* $f$ if $\alpha(S, f) \leq \epsilon$.

**Definition** A boolean function $f$ is said to be *coherent* (with respect to probabilistic polynomial time Turing machines) if for every $\epsilon > 0$ there exists an examiner $S$ that is $(1-\epsilon)$-successful on $f$.

**Remark** A main difference between between the definitions for coherent functions and self-reducible functions is that in the former case we allow an adaptive choice of queries in the interpolation process, while for self-reducible functions the choice of points for interpolation purpose is made in one pass.

**Theorem 3** There exists a language in $DSPACE(2^{n^{\log\log n}})$ that is not coherent.

**Sketch of Proof** To make the main idea clear, we will assume that checkers only ask adversary programs $P$ to return $P(x)$ for strings $x$ having the same length as the input string $w$. To simplify the matter further, we will only prove the existence of $f$ which have no efficient checkers. We will indicate at the end how to modify the arguments to remove these simplifying assumptions.

The general approach is as follows. For any examiner $S$, integer $n$, let $h_S(n)$ be the number of $n$-variable boolean functions for which $S$ interpolates successfully (for all $2^n$ possible inputs). If we can prove that $h_S(n)$ is $o(2^n)$, then we can use the diagonalization technique to construct an $f$ which will avoid being interpolated by any $M$. Now the arguments used in proving Theorem 1 almost tell us that $h_S(n)$ is $o(2^n)$, except that we now allow probabilistic algorithms with errors, which entails a more involved argument.

To study the behavior of $S$ on input strings of length $n$, we introduce some notations. An *$n$-boolean function* is a mapping $b : \{0,1\}^n \rightarrow \{0,1\}$. For any boolean function $f : \{0,1\}^* \rightarrow \{0,1\}$, let $f^{(n)}$ denote the induced $n$-boolean function when $f$ is restricted to $n$-bit inputs. An *$(n,m)$-tree* is a rooted decision tree $K$ such that the following conditions are met:
(a) The root has $2^n$ edges labeled by distinct $n$-bit strings,
(b) If $v$ is an internal node in the $w$-th subtree from the root (where $w \in \{0,1\}^n$), then $v$ contains a query of the form $b(x) =?$ where $x \neq w$,
(c) Each leaf $\ell$ contains an *output* $\mu(\ell)$, and
(d) The height of the tree is $m + 1$.

For any $n$-boolean function $b$, we can regard $K$ as a deterministic algorithm for guessing the value $b(w)$ for any input $w$. Given $w$, we first take the edge labeled by $w$ from the root, and then trace a path by asking queries and branching according to the answers until a leaf $\ell$ is reached; we then make the guess $b(w) = \mu(\ell)$. Clearly, for any $w$, the number of queries asked does not exceed $m$. Let $\mathcal{K}_{n,m}$ denote the set of all $(n,m)$-trees. We now define randomized algorithms.

An *$(n,m)$-tree examiner* $T$ is specified by a probability distribution $\tau$ over $\mathcal{K}_{n,m}$. As an algorithm for guessing values $b(w)$, we first pick a random $K$ according to $\tau$ and execute $K$. For any $w, b$, let $\beta(T, b, w)$ denote the probability that $T$ will guess $b(w)$ incorrectly; let $\beta(T, b) = \max_w \beta(T, b, w)$. For any $\epsilon$, let $\mathcal{B}_{T,\epsilon}$ denote the set of all $n$-boolean functions $b$ such that $\beta(T, b) \leq \epsilon$.

CLAIM 1 Let $0 < \epsilon < 1/2$. If there exists an examiner $S$ which is (1-$\epsilon$)- successful on $f$, then there exists a $k > 0$ and a sequence of $(n, n^k)$-tree examiners $T_n$

such that $\beta(T_n, f^{(n)}) \leq 4^{-n}$ for all $n$.

To prove the claim, let $S'$ be the examiner which, for any input $(w, P)$, employs $S$ $10\lceil n/(1/2-\epsilon)^2\rceil + 1$ times and takes the majority vote as the verdict. It is easy to verify that the probability that $S'$ has made a mistake is at most $4^{-n}$ for any input $(w, P)$ with $|w| = n$. Choose integer $k$ large enough so that, for all $n$, the running time of $S'$ is at most $n^k$ for input $(w, P)$ with $|w| = n$. The behavior of $S'$ when $|w| = n$ naturally gives an $n$-tree examiner $T_n$ satisfying the claim. We have thus proved CLAIM 1.

The next claim extends the argument used in Theorem 1 to the probabilistic case, and is central to the proof of Theorem 3. Let $k > 0$ be fixed. We will prove the following claim for all large enough $n$.

CLAIM 2 Let $\epsilon = 4^{-n}$. If $T$ is an $(n, n^k)$-tree examiner, then $|\mathcal{B}_{T,\epsilon}| \leq 2^{2^n(1-\frac{1}{2n^k})}$.

To prove CLAIM 2, let us label the elements of $\mathcal{B}_{T,\epsilon}$ as $K_1, K_2, K_3, \cdots$. Let $M = \lceil 2^n/(n^k+1)\rceil$, and $I_n$ be the set of all $M$-tuples of positive integers. For each $\tilde{a} = (a_1, a_2, \cdots, a_M) \in I_n$, consider the following deterministic procedure which, for any $n$-boolean function $b$, generates a binary string $\sigma(\tilde{a}, b)$.

```
Procedure encode
begin Set $J \leftarrow \emptyset$;
      for $j = 1$ to $M$ do
          begin take smallest $i \notin J$;
                use $K_{a_j}$ to ask exactly $n^k$ queries
                    $b(t)$=? $(t \neq i)$ and then
                    make a guess of $b(i)$;
                Set $\gamma_j \in \{0,1\}^{n^k}$ to be the
                    sequence of answers to the
                    $n^k$ queries asked;
                enlarge $J$ to include $i$ and
                    all the $t$'s
          end
      if all the $M$ guesses of $b(i)$ are correct
          then $\sigma(\tilde{a}, b) \leftarrow 0\gamma_1\gamma_2\cdots\gamma_M$;
      if any of the $M$ guesses of $b(i)$ is incorrect
          then $\sigma(\tilde{a}, b) \leftarrow 1$ followed by the $2^n$
                values of $b(w)$
end encode
```

In the above procedure it is understood that $i, t$ are $n$-bit strings, and the phrase "smallest $i$" means "lexicographically smallest".

Let $\tau$ be the probability distribution describing $T$ such that $\tau(i)$ is the probability that $T$ takes on the value $K_i$. We extend the notation $\tau$ by defining $\tau(\tilde{a})$ to be $\prod_{1\leq j\leq M}\tau(a_j)$, when $\tilde{a} = (a_1, a_2, \cdots, a_M) \in I_n$.

For any $\tilde{a}$, $\sigma(\tilde{a}, b)$ gives an encoding of all the $n$-boolean functions $b$. Let $\mu(\tilde{a})$ denote the average length of the encoding for a random $b \in \mathcal{B}_{T,\epsilon}$. Then

$$\begin{aligned}\sum_{\tilde{a}}\tau(\tilde{a})\mu(\tilde{a}) &= \sum_{\tilde{a}}\tau(\tilde{a})\frac{1}{|\mathcal{B}_{T,\epsilon}|}\sum_{b\in\mathcal{B}_{T,\epsilon}}|\sigma(\tilde{a},b)| \\ &= \frac{1}{|\mathcal{B}_{T,\epsilon}|}\sum_{b\in\mathcal{B}_{T,\epsilon}}\sum_{\tilde{a}}\tau(\tilde{a})|\sigma(\tilde{a},b)|(2)\end{aligned}$$

Now, for any fixed $b \in \mathcal{B}_{T,\epsilon}$, the probability for any of the $M$ guesses to be incorrect is bounded by $M\cdot\epsilon \leq 2^{-n}$, and hence for large $n$

$$\begin{aligned}\sum_{\tilde{a}}\tau(\tilde{a})|\sigma(\tilde{a},b)| &\leq (1+Mn^k) + 2^{-n}(1+2^n) \\ &\leq 2^n(1-\frac{1}{2n^k}). \qquad (3)\end{aligned}$$

It follows from (2) and (3) that there exists $\tilde{a}$ such that

$$\mu(\tilde{a}) \leq 2^n(1-\frac{1}{2n^k}).$$

Since any encoding in binary of the element in the set $\mathcal{B}_{T,\epsilon}$ must have an average length of $\log_2|\mathcal{B}_{T,\epsilon}|$ or more, it follows that

$$|\mathcal{B}_{T,\epsilon}| \leq 2^{2^n(1-\frac{1}{2n^k})}.$$

This completes the proof of CLAIM 2.

CLAIMs 1 and 2 allow us to construct by diagonalization a function $f$ which is not coherent, as we will show below. Let $S_1, S_2, \cdots$ be the sequence of all examiners. Let $\epsilon = 1/4$, and let $T_1^{[i]}, T_2^{[i]}, \cdots, T_n^{[i]}, \cdots$ be the sequence of tree examiners mentioned in CLAIM 1 for $S_i$. Let $\epsilon_n = 4^{-n}$. We can define a sequence of integers $\Lambda_i$ such that $\Lambda_1 < \Lambda_2 < \cdots < \Lambda_i < \cdots$ and, for each $j$,

$$\left|\cup_{1\leq i\leq j}\mathcal{B}_{T_n^{[i]},\epsilon_n}\right| \leq j\cdot 2^{2^n(1-1/n^{\log\log n})} < 2^{2^n}$$

for all $n \geq \Lambda_j$. As there are $2^{2^n}$ $n$-boolean functions, there exists a function $f$ such that $f^{(n)} \notin \bigcup_{1 \leq i \leq j} \mathcal{B}_{T_n^{[i]}, \epsilon_n}$ for all $n \geq \Lambda_j$. This means, for any $i$, $T_n^{[i]}$ will fail to interpolate $f^{(n)}$ with error $\epsilon_n$ for all sufficiently large $n$. By CLAIM 2 no examiner $S_i$ can be $(1 - \epsilon)$-successful on $f$.

This completes the proof of Theorem 3, with the assumptions stated at the beginning of the proof. To show that the function $f$ can be constructed in space $2^{n^{\log \log n}}$, one only needs to study the resources needed in carrying out the diagonalization argument. To remove the restriction that checkers can ask queries only of the same length as the input, we observe that the proof presented remains valid even if $n$ does not take on all possible integer values. Let $\mathcal{L}$ denote the family of languages that contain no strings of length not of the form $2^{2^{2^i}}$. Then the checkers have no need to ask queries longer than the input. The present proof then shows the existence of a language in $\mathcal{L}$ that is not coherent. The details will be described in the complete paper. □

**Remark** Theorem 3 remains true if we replace $\log \log n$ by any exponential-space-constructible monotonic function $k(n) \to \infty$.

**Remark** A result related to Theorem 3 was proved by Feigenbaum, Kannan, and Nisan [FKN] for self-reducible functions.

## 5 Program Checkers

### 5.1 Turing Machine Model

Program checking, which is a novel approach for checking the correctness of programs, was recently raised by Blum and Kannan [BK] (see also Blum, Luby, and Rubinfeld [BLR], Lipton [L]). Seemingly intractable computations such as graph isomorphism have simple *checkers* which run in polynomial time. The limitation of this approach is not well understood; in fact no computations have been shown to be hard to check.

Let $\mathcal{M}$ be the class of probabilistic polynomial time Turing machines. We will show that any boolean-valued function with efficient checkers must be coherent with respect to $\mathcal{M}$. As a corollary, this shows that there exist functions in $DSPACE(2^{n^{\log \log n}})$ for which no efficient checker exists.

Let $f$ be a computational problem. Let $P$ be a deterministic Turing machine that halts for all input strings $w$; call the output $P(w)$. Following the terminology in [BK], we say that $P$ *has a bug* (with respect to $f$) if $P(w) \neq f(w)$ for some $w$. An *efficent checker* for $f$ is a probabilistic polynomial time oracle Turing machine which, for any $P$ as the oracle and any $w$, will output either CORRECT or BUGGY such that the following conditions are satisfied:
(a) If $P$ has no bugs, then with probability at least $2/3$, $M$ will output CORRECT;
(b) If $P(w) \neq f(w)$, then with probability at least $2/3$, $M$ will output BUGGY.

Clearly, the quantity $2/3$ in the above definition can be substituted by any fixed constant $1/2 < c < 1$.

**Theorem 4** If $f$ has an efficient checker, then $f$ is coherent.

**Proof** Let $\epsilon > 0$, and $V$ be an efficient checker for $f$ with error probability at most $\epsilon$. For any $w \in \{0,1\}^*$, let $P_w$ denote the program which returns $P_w(w) = 0$ and $P_w(x) = f(x)$ for all $x \neq w$. Suppose $(w, P)$ is input to $V$. From the definition of program checker, if we make the guess $f(w) = 0$ when $V$ gives the CORRECT verdict, and 1 when $V$ outputs BUGGY, then the probability $r_V(w)$ of the guess being correct is at least $1 - \epsilon$.

Now turn $V$ into an examiner $S_V$ by simulating $V$, except omitting queries $f(w) =?$ where $w$ is the input and behaving as if an adversary returns the value $f(w) = 0$; at the end if $V$ gives a CORRECT verdict, then $S$ outputs the value 0, and if $V$ gives a BUGGY verdict, then $S$ outputs the value 1. Clearly, the probability for $S_V$ to output the correct value of $f(w)$ is exactly $r_V(w)$, which is at least $1 - \epsilon$. This proves the theorem. □

**Theorem 5** There exists $L \in \text{DSPACE}(2^{n^{\log\log n}})$ for which there is no efficient checker.

**Proof** Follows immediately from Theorems 3 and 4. □

**Remark** Theorem 5 can be strengthened to $DSPACE(2^{cn})$ for any $c > 1$ by a direct proof from the definition of program checkers.

## 5.2 Comparison Tree Models

Although program checking was defined originally for the Turing machine model in [BK], it is of interest to consider program checking in other models such as the comparison-based model, the bounded-degree algebraic tree model, or the arithmetic computation tree model. The merits are (a) these models have more realistic cost measure than the Turing machine model for certain types of problems such as geometric optimization, and (b) nonlinear lower bounds are known in these models, which might make it possible to compare the cost of optimal checkers versus optimal algorithms for a problem.

We take a step in this direction by considering two classes of ordering problems in a comparison-base model, namely, sorting-like problems as defined by Fredman [F] and partial order production problems as given by Schönhage [Sch] (also see [Y2]). We will show that these two classes of problems behave quite differently with respect to program checking. In fact, for almost all sorting-like problems, checking is as expensive as direct computing; while every partial order production problem has a linear time checker and hence checking is always cheap.

Before proceeding, we give an example of a well-known problem that has an efficient checker in the comparison tree model.

ELEMENT DISINCTNESS: Given real numbers $x_1, x_2, \cdots, x_n$, determine whether they are all distinct.

In the comparison tree model, a program can perform a sequence of tests of the form $x_i : x_j$ with $<$, $=$, or $>$ as possible outcomes, and the cost of the program is the number of comparisons needed. It is well known that the cost of any program is at least $\Omega(n \log n)$, even for probabilistic algorithms allowing error (Manber and Tompa [MT]). Let us consider probabilistic procedures $V$ which, when given a set $W$ of numbers and a program $P$, want to decide if the answer of $P$ to $W$ is reliable. Aside from performing comparisons, $V$ can call on $P$ to determine whether a subset (allowing repeated entries) of the input numbers has the all-distinctness property. We will say that $V$ is a *program checker* if for any $W$ and $P$, the final judgment made has probability at least 2/3 to be right. The cost of the program checker is the maximum expected number of comparisons plus the number of calls to $P$. Since we will describe an algorithm and not prove lower bounds for this problem, we will not describe the complexity model in detail. The model will become clear from the later discussion of sorting-like problems.

**Theorem 6** ELEMENT DISTINCTNESS has a program checker with $O(n)$ cost in the comparison tree model.

**Proof** We describe below a program checker $V$. Given an input $(W, P)$, where $W = \{x_1, x_2, \cdots, x_n\}$ is a set of real numbers and $P$ is a program, possibly faulty, for the element distinctness problem. Depending on the answer of $P$ to $W$, we distinguish two cases.

(A) If $P$ returns "no", then for $j = n, n-1, \cdots$, ask $P$ whether $\{x_1, x_2, \cdots, x_{j-1}\}$ contains all distinct elements, until either (a) in the round $j = k \geq 2$, $P$ returns a "yes" answer, or (b) up to and including round $j = 2$, $P$ has not returned a "yes" answer. In case (b) $V$ says BUGGY. In case (a) $V$ compares $x_k$ against every $x_i$, $1 \leq i < k$; if $x_k = x_i$ for some $i$ then says CORRECT, and otherwise gives the BUGGY verdict.

(B) If $P$ returns "yes", then $V$ randomly permutes $x_1, x_2, \cdots, x_n$ and partitions it into two sets $\{y_1, y_2, \cdots, y_{n/2}\}$ and $\{y_{n/2+1}, \cdots, y_n\}$. Then it constructs $n$ sets $W_j = \{y_1, y_2, \cdots, y_{n/2}\} \cup \{j\}, 1 \leq j \leq n$, and presents to $P$ in random order these sets. If $P$

returns "yes" for any $W_j$ with $1 \leq j \leq n/2$ or if $P$ returns "no" for any $W_j$ with $1 \leq j \leq n/2$, then $V$ says BUGGY; otherwise $V$ says CORRECT.

The above $V$ clearly runs in time $O(n)$. It is easy to see that $V$ always gives the right verdict in situation (A). In situation (B) if $P$ did not lie, $M$ clearly will give the right verdict. It remains to show that, when $x_1, x_2, \cdots, x_n$ have some $x_i = x_j$ but $P$ returns "yes", $M$ will catch $P$ with a BUGGY verdict with probability at least 1/4. However, this is true since with probability 1/2, $x_i$ and $x_j$ will be in different halves of the $y$'s list, and $P$ cannot distinguish between $S_i$ and $S_j$. This proves Theorem 6. □

**Remark** In Blum and Kannan [BK] and Lipton [L], $O(n)$-time program checkers were given for the element distinctness problem, but not in the comparison model.

We now turn to the discussion of sorting-like problems and partial order production problems. A *sorting-like problem* $g$ is a function from the set of all permutations on $n$ or fewer elements into $\{1, 2, \cdots, r\}$ where $n, r \geq 2$ are integers. Let $J_m$ denote the set of all $m$-tuples of distinct real numbers. For any $\tilde{x} = (x_1, x_2, \cdots, x_m) \in J_m$, let $\sigma_{\tilde{x}}$ be the permutation $(i_1, i_2, \cdots, i_m)$ such that $x_{i_1} < x_{i_2} < \cdots < x_{i_m}$. Given an input $\tilde{x} \in J_m$ where $m \leq n$, we want to determine the value of $g(\sigma_{\tilde{x}})$. For example, the standard sorting problem corresponds to the case $r = 1! + 2! + \cdots + n!$ with each integer $1 \leq j \leq r$ representing a linear ordering of $n$ or fewer elements. Let $G_{n,r}$ be the set of all sorting-like problems.

Below we give a rough description of the model. A detailed specification is given in the Appendix.

Let $\mathcal{F}_{n,r}$ be the set of all decision trees using binary comparisons $x_i : x_j$ as primitives. A decision tree $F \in \mathcal{F}_{n,r}$ is an *algorithm* for $g$ if it outputs the value of $g(\sigma_{\tilde{x}})$ for all inputs $\tilde{x}$.

A *deterministic checker* $H$ is a decision tree for which the inputs are triplets $(\tilde{x}, F, j)$; $F \in \mathcal{F}_{n,r}$ and $j$ is the output of $F$ when $\tilde{x}$ is given to $F$. The deterministic checker $H$ needs to either endorse the value $j$ as $g(\sigma_{\tilde{x}})$ or returns a BUGGY verdict. In addition to performing comparisons $x_i : x_j$, $H$ can make subroutine calls of $F$, each time supplying $F$ with a permuted tuple $(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ and then running $F$. In the spirit of [BK], each call is counted as one step. A *checker* $V$ is a randomized decision tree defined in the standard way [Y1]. We say that $V$ is a *checker with error* $\epsilon$ if, for every $\tilde{x}$ and $F \in \mathcal{F}_{n,r}$, the probability of $V$ returning the wrong verdict is at most $\epsilon$.

Clearly, for the standard sorting problem, there is a checker $V$ with error 0 and running time $n-1$. In fact, if the input is $(\tilde{x}, F, j)$ where $j$ is the encoding of $(i_1, i_2, \cdots, i_m)$, then $V$ can simply check the $m-1$ inequalities $x_{i_1} < x_{i_2} < \cdots < x_{i_m}$. (Our model is different from the one used in [BK] for sorting and has no need to check set equalities.) Since it is clear that any algorithm (even randomized and with error, say 1/3) for sorting must use $\Omega(n \log n)$ comparisons, it is easier to *check* than to *compute* for the sorting problem. We will show that this is an anomaly for sorting-like problems, and in fact almost all sorting-like problems are as hard to check as to compute. Before doing that, we define the concept of coherence in this model.

An *examiner* $S$ is a randomized decision tree for which the inputs are pairs $(\tilde{x}, g)$, where $\tilde{x} \in J_m, m \leq n$ and $g \in G_{n,r}$; the outputs are integers $1 \leq j \leq r$. It can make comparisons $x_i : x_j$ and ask queries $g(\sigma_{\tilde{y}}) =$ ? where $\tilde{y} = (x_{i_1}, x_{i_2}, \cdots, x_{i_k}) \neq \tilde{x}$; each comparison or query is counted as one step for the cost. Let $\alpha(S, g, \tilde{x})$ denote the probability that the output of $S$ is different from $g(\sigma_{\tilde{x}})$; let $\alpha(S, g) = \max_{\tilde{x}} \alpha(S, g, \tilde{x})$. For any $S$ and $g$, we will say that $S$ is $(1-\epsilon)$-*successful on* $g$ if $\alpha(S, g) \leq \epsilon$. Let $\mathcal{S}_{n,r}$ denote the set of all examiners.

**Definition** Let $r > 1$ be fixed. A sequence of sorting-like problems $\hat{g} = g_1, g_2, \cdots$ where $g_n \in G_{n,r}$ is said to be *coherent* (with respect to $O(n)$-cost decision trees) if, for every $\epsilon > 0$, there exists a sequence of examiners $S_1, S_2, \cdots$, where $S_n \in \mathcal{S}_{n,r}$, such that $S_n$ has $O(n)$ cost and is $(1-\epsilon)$-successful on $g_n$ for every $n$.

**Theorem 7** Let $\hat{g} = g_1, g_2, \cdots$ where $g_n \in G_{n,r}$. If there exist a sequence of checkers $V_n$ for $g_n$ with cost $O(n)$ and error 1/3, then $\hat{g}$ must be coherent.

**Proof** Similar to the proof of Theorem 4. □

**Theorem 8** Take a random sequence of sorting-like problems $g_1, g_2, \cdots$ where $g_n$ is uniformly chosen from $G_{n,r}$. Then with probability 1 $\hat{g}$ is not coherent.

**Proof** Omitted. □

It follows from Theorems 7 and 8 that, if we choose a random $g_n$ uniformly from $G_{n,r}$ for each $n$, then with probability 1, there cannot be any sequence of $O(n)$-cost checkers $V_n$ for $g_n$ with error 1/3. A stronger version of the result is given below.

**Theorem 9** Let $r > 1$ be fixed. Take a random $g$ uniformly chosen from $G_{n,r}$. Then with probability $1 - o(1)$ for large $n$, the following is true: any checker for $g$ with error probability less than 1/3 must have $\Omega(n \log n)$ running time.

**Proof** Omitted. □

The other class of decision tree problems we consider is the partial order production problem ([Sch][Y2]). Let $Q$ be a partial order on $n$ elements $z_1, z_2, \cdots, z_n$. Given $n$ distinct numbers $x_1, x_2, \cdots, x_n$, we want to find a permutation $i_1, i_2, \cdots, i_n$ such that $z_j <_Q z_k$ implies $x_{i_j} < x_{i_k}$. Define *checkers* as in the sorting-like problems case. The following result shows that, for any $Q$ with only a moderate number $\#Q$ of linear extensions, checking is easier than computing. The proof is an easy consequence of results in the literature originally considered in other contexts.

**Theorem 10** Any randomized algorithm for $Q$-production with error 1/3 must have running time $\Omega(\log(n!/\#Q))$. There is a checker for $Q$ with error 1/3 and running time $O(n)$.

The first statement is a straightforward extension of Schönhage's result [Sch] that any deterministic algorithm for $Q$-production must have running time $\Omega(\log(n!/\#Q))$. The second statement follows immediately from a recent result of Valerie King [K] that the partial order verification problem on $n$ elements can be done in $O(n)$ time probabilistically (improving on a previous result by Kenyon-Mathieu and King [KK]).

## 6 Concluding Remarks

There are many open problems in this subject. The broad direction is to seek other basic properties for classes of functions, study their relationships, and use them to resolve cryptographic questions. It seems that the randomness concept for functions is now well understood (see Goldreich, Goldwasser, and Micali [GGM]), but concepts concerning various non-randomness properties are not yet fully explored. Our develpment of the "coherence" notion is one step in this direction.

We list below a few concrete open questions which are immediately suggested by this work.

(a) Is there any language in the polynomial time hierarchy that is not coherent, under reasonable assumptions such as the noncollapsing of certain complexity hierarchies ? A result of this nature for the encrypted data computation was given by coherence, Feigenbaum, and Kilian [AFK]. How about assuming the existence of one-way functions? Some interesting results along this line have recently been obtained by Beigel and Feigenbaum [BeiF].

(b) Investigate program checking in the algebraic computation tree model. For example, can one prove that any checker for the problem of finding the closest pair of points in the plane must take $\Omega(n \log n)$ time?

(c) Is the time-memory tradeoff for inverting random functions achieved by Hellman's algorithm optimal?

## References

[AFK] M. Abadi, J. Feigenbaum, and J. Kilian, "On hiding information from an oracle," *Journal of Computer and System Sciences*, **39** (1989), 21-50.

[AL] D. Angluin and D. Lichtenstein, "Provable security of cryptosystems: a survey," Technical Report TR-288, Yale University, October 1983.

[BeaF] D. Beaver and J. Feigembaum, "Hiding instances in multioracle queries," *Proceedings of the 7th Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag LNCS 415, edited by C. Choffrut and T. Lengauer, 1990, 37-48.

[BeiF] R. Beigel and J. Feigenbaum, "On the complexity of coherent functions," private communication, February 1990.

[BK] M. Blum and S. Kannan, "Designing programs that check their work," *Proceedings 21st ACM Symposium on Theory of Computing* , May 1989, 86-97.

[BLR] M. Blum, M. Luby, and R. Rubinfeld, "Stronger checkers and general techniques for numerical problems," a talk presented in the *DIMACS Workshop on Cryptography and Distributed Computing*, Princeton, October 1989.

[FKN] J. Feigenbaum, S. Kannan, and N. Nisan, "Lower bounds on random-self-reducibility," to appear in *Proceedings of Structures 90.*

[F] M.L. Fredman, "How good is the information theory bound in sorting," *Theoretical Computer Science* **1**(1976), 355-361.

[GGM] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM* **33** (1986), 792-807.

[H] M.E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory* **26** (1980), 401-405.

[KK] C. Kenyon-Mathieu and V. King, "Verifying partial orders," *Proceedings 21st ACM Symposium on Theory of Computing* , May 1989, 367-374.

[K] V. King, "An optimal randomized algorithm for set-maxima," private communication, 1989.

[L] R.J. Lipton, "New directions in testing," manuscript and a talk presented in the *DIMACS Workshop on Cryptography and Distributed Computing*, Princeton, October 1989.

[MT] U. Manber and M. Tompa,"The complexity of problems on probabilistic, nondeterministic and alternating decision trees," *Journal of ACM* **32** (1985), 720-732.

[Sch] A. Schönhage, "The production of partial orders," *Asterisque* **38-39** (1976), 229-246.

[SS] A. Shamir and J. Spencer, private communication, 1979.

[Y1] A.C. Yao, "Probabilistic computations: towards a unified measure of complexity," *Proceedings 18th Annual IEEE Symposium on Foundations of Computer Science*, October 1977, 222-227.

[Y2] A.C. Yao, "On the complexity of partial order productions," *SIAM Journal on Computing* **18** (1989), 679-689.

## Appendix: Model for Sorting-Like Problems

Let $n, r$ be positive integers. Let $\Delta_n = \cup_{1 \leq m \leq n} \Gamma_m$, where $\Gamma_m$ is the set of all permutations of $\{1, 2, \cdots, m\}$. Following Fredman [F] with minor modifications, a *sorting-like problem Q* is a family of disjoint sets $(Y_1, Y_2, \cdots, Y_r)$ that partition $\Delta_n$. Given an $m$-tuple of distinct real numbers $\tilde{x} = (x_1, x_2, \cdots, x_m)$, we want to determine $f_Q(\tilde{x})$, the integer $j$ such that $Y_j$ contains the linear ordering satisfied by the $x_i's$.

A *comparison forest F* is a collection of $n$ comparison-based decision trees $A_1, A_2, \cdots, A_n$, where $A_m$ uses only comparisons of the form $x_i : x_j$ with $1 \leq i, j \leq m$; each leaf of $A_m$ contains an *output* $s_\ell \in \{1, 2, \cdots, r\}$. Let $\mathcal{F}$ denote the set of all comparison forests.

For $\tilde{x} = (x_1, x_2, \cdots, x_m)$ and $F \in \mathcal{F}$, let *cost*$(F, \tilde{x})$ denote the number of comparisons used when $A_m$ is used for the input $\tilde{x}$. We will say that $F$ *correctly computes* $Q$ for input $\tilde{x} = (x_1, x_2, \cdots, x_m)$, if for $A_m$ the leaf $\ell$ reached by $\tilde{x}$ satisties $s_\ell = f_Q(\tilde{x})$; otherwise, $F$ *incorrectly computes* $Q$ for $\tilde{x}$. Write $\nu(F, \tilde{x}) = 0$ in the former case and 1 in the latter case. Let $\mathcal{F}_Q$ denote the set of $F$ that correctly compute $Q$ for all inputs $\tilde{x}$.

A *randomized algorithm* $B$ for $Q$ is a probability distribution $p$ on $\mathcal{F}$. For input $\tilde{x}$, the *cost* of $B$ is defined to be $c(B, \tilde{x}) = \sum_{F \in \mathcal{F}} p(F) \operatorname{cost}(F, \tilde{x})$; the *error probability* $e_Q(B, \tilde{x})$ is, for a random $F$ distributed according to $p$, the probability that $F$ incorrectly computes $\tilde{x}$. Let $c(B) = \max_{\tilde{x}} c(B, \tilde{x})$. For any $0 \leq \epsilon \leq 1$, let $\mathcal{B}_{Q,\epsilon}$ denote the set of all randomized algorithms $B$ satisfying $e_Q(B, \tilde{x}) \leq \epsilon$. The *randomized complexity with error* $\epsilon$ is defined as $C_{Q,\epsilon} = \min \{ c(B) \mid B \in \mathcal{B}_{Q,\epsilon} \}$.

Clearly, every sorting-like problem $Q$ on $n$ elements has a comparison forest $F$ such that $F$ computes $Q$ correctly for every $\tilde{x}$ and uses $O(n \log n)$ comparisons. As a trivial corollary, $C_{Q,\epsilon} = O(n \log n)$ for all $0 \leq \epsilon \leq 1$.

A *deteministic checker* $H$ is a collection of $rn$ rooted trees $D_{k,1}, D_{k,2}, \cdots, D_{k,n}$, $1 \leq k \leq r$, where $D_{k,m}$ has two kinds of internal nodes: *comparison nodes* with tests $x_i : x_j$, and *subroutine nodes* specified with a $j$-tuple $\tilde{x}' = (x_{i_1}, x_{i_2}, \cdots, x_{i_j})$ where $1 \leq j \leq n$. Out of every comparison node there are two edges labeled with $<$ and $>$; out of each subroutine node, there are up to $r$ edges labeled with distinct integers between 1 and $r$. At each leaf $\ell$, a value $\mu(\ell) \in \{ \text{CORRECT}, \text{BUGGY} \}$ is stored. Let $\mathcal{H}$ denote the set of all comparison forest checkers.

As a deterministic checker, $H \in \mathcal{H}$ works as follows. Given any adversary program, i.e. a comparison forest $F \in \mathcal{F}_n$ and $\tilde{x} = (x_1, x_2, \cdots, x_m)$, suppose for input $\tilde{x}$ the output of $F$ is $k$. Then $H$ will try to *check* the result by tracing a path in $D_{k,m}$ until a leaf $\ell$ is reached; the rules for tracing the path are (a) when a comparsion node with test $x_i : x_j$ is encountered, perform the test and take the branch according to the result, and (b) when a subroutine node with $\tilde{x}' = (x_{i_1}, x_{i_2}, \cdots, x_{i_j})$ is encountered, ask $F$ to perform a computation with input $\tilde{x}'$, and take the edge labeled by $t$ if $t$ is the output from $F$. The value $\mu(\ell)$ is the result to be returned by the checker $H$. Define *cost*$(H, F, \tilde{x})$ as the path length from the root to $\ell$. Let $\delta(H, F, \tilde{x})$ be the boolean function whose value is 1 if and only if $H$ fails in checking the validity of $F$'s computing $Q$ for input $\tilde{x}$. Precisely, let $\delta(H, F, \tilde{x}) = 1$ if and only if either of the following conditions is true: (a) $(F \in \mathcal{F}_Q) \wedge (\mu(\ell) = \text{BUGGY})$, (b) $(\nu(F, \tilde{x}) = 1) \wedge (\mu(\ell) = \text{CORRECT})$.

A *checker* $V$ for $Q$ is a probability distribution $p$ on $\mathcal{H}$. For input $(\tilde{x}, F)$, the expected number of operations used by $V$ is clearly $c(V, F, \tilde{x}) = \sum_{H \in \mathcal{H}} p(H) \operatorname{cost}(H, F, \tilde{x})$; the *error probability* $\delta_Q(V, F, \tilde{x})$ is, for a random $H$ distributed according to $p$, the probability that $H$ fails as a checker, i.e. $\sum_{H \in \mathcal{H}} p(H) \delta(H, F, \tilde{x})$. The *cost* of $V$ is defined to be $\max_{F, \tilde{x}} c(V, F, \tilde{x})$. We say that $V$ is a *checker for* $Q$ *with error* $\epsilon$, if $\delta_Q(V, F, \tilde{x}) \leq \epsilon$ for all $F, \tilde{x}$.