Designing Programs That Check Their Work[1]

Manuel Blum

Sampath Kannan

Computer Science Division
University of California at Berkeley
94720

A *program correctness checker* is an algorithm for checking the output of a computation. This paper defines the concept of a program checker. It designs program checkers for a few specific and carefully chosen problems in the class P of problems solvable in polynomial time. It also applies methods of modern cryptography, especially the idea of a probabilistic interactive proof, to the design of program checkers for group theoretic computations. Finally it characterizes the problems that can be checked.

1. Introduction

Problem solving often involves two stages: calculating the result, and checking it. While computers have been put to good use in doing the calculation, checking the result has largely been left to the ingenuity of the programmer. It is hard to find any good reason for this state of affairs.

It is possible that the lack of a mathematically sound definition of checking has held back the mechanization of the checking stage. In this paper we provide a precise formulation of the concept of a checker.

While the notion of an algorithm to solve a problem dates back at least to Euclid, the idea of a mechanical procedure to check computation does not appear to have been given very serious thought. Perhaps this is because checking has been believed to be a matter of ingenuity, not doable on a computer. In this paper we hope to dispel this belief by providing several examples of problems for which mechanical checkers can be written. As an added bonus many of these checkers are 'simpler' (in a sense to be defined later) than the programs they check.

1.1. Verification, Checking, and Testing

Before we introduce our formalism it is useful to compare the notion of checking with the related notions of verification and testing.

Verification has to do with writing formal proofs of correctness for programs. Such proofs are an excellent way to ensure reliability, but they have not been constructed in practice for any but toy programs. The difficulty has to do in part with the somewhat arbitrary definitions and conventions that arise in real programming languages. Those conventions make proofs of correctness less satisfying intellectually than the usual mathematical proofs. Besides, theorems are generally hard (or impossible) to prove, and

---

[1] This paper extracts from work in Blum[B2] and Kannan[K2].

Support was provided in part by NSF grant CCR85-13926 and in part by the International Computer Science Institute at Berkeley.

Authors' e-mail addresses: blum@ernie.berkeley.edu and kannan@ernie.berkeley.edu

this includes theorems that assert a program's correctness.

Checking is concerned with the simpler task of verifying that a given program returns a correct answer on a given input rather than on all inputs. Checking is not as good as verification, but it is easier to do. It is important to note that unlike testing and verification, checking is done each time a program is run.

Testing is concerned with putting a program through its paces on samples of the expected input, in hopes that it will also work correctly on those inputs that arise in practice. Although there have been numerous attempts to put testing on a sound mathematical basis, the foundation for the field is still quite murky, and there is a chance it will always be so. Fortunately, checking looks like it can be put on the same sound basis as verification. Moreover, it is conceivable that writing checkers will be no harder for programmers to do than it is for them to write programs in the first place.

1.2. Brief History

The ideas in this paper arise from cryptography, probabilistic algorithms, and program testing. Particularly important for this work are the probabilistic interactive proofs of Goldwasser, Micali and Rackoff [GMR], and their spinoffs. As will be seen, several of the correctness checkers constructed in this paper use probabilistic interactive proofs as a kind of scaffolding. Equally important for this work are the papers on randomized algorithms of Rabin [R] and Freivald [F]. The latter, remarkably enough, includes excellent program checkers for integer, polynomial, and matrix multiplication. Finally, the works of Blum and Raghavan[BR], Budd and Angluin [BA], and Weyuker [W] are relevant in that they too seek to give program testing a rigorous mathematical basis.

2. Program Checkers

Let π denote a (computational) decision and/or search problem. For x an input to π, let $\pi(x)$ denote the output of π. Let P be a program (supposedly) for π that halts on all instances of π. We say that such a program P has a *bug* if for some instance x of π, $P(x) \neq \pi(x)$.

Define an (*efficient*) *program checker* C_π for problem π as follows: $C_\pi^P(I;k)$ is any probabilistic (expected-poly-time) oracle Turing machine that satisfies the following conditions, for any program P (supposedly for π) that halts on all instances of π, for any instance I of π, and for any positive integer k (the so-called "security parameter") presented in unary:

1. If P has no bugs, i.e., $P(x) = \pi(x)$ for all instances x of π, then with probability[2] greater or equal to $1 - 1/2^k$, $C_\pi^P(I;k) = CORRECT$ (i.e., $P(I)$ is *CORRECT*).

2. If $P(I) \neq \pi(I)$, then with probability greater or equal to $1 - 1/2^k$, $C_\pi^P(I;k) = BUGGY$ (i.e., P is *BUGGY*).

Some remarks are in order:

i. The running time of C above *includes* whatever time it takes C to submit inputs to and receive outputs from P, but *excludes* the time it takes for P to do its computations.

ii. In the above definition, if P has bugs but $P(I) = \pi(I)$, ie. buggy program P gives the correct output on input I, then $C_\pi^P(I;k)$ may output *CORRECT* or *BUGGY*.

Regarding this model for ensuring program correctness the question naturally arises: if one cannot be sure that a program is correct, how then can one be sure that its checker is correct? This is a very serious problem![3] One solution is

---

2 This probability is computed over the sample space of all finite sequences of coin flips that C could have tossed.

3 The first author was privileged to hear Dr. Warren S. McCulloch at the First Bionics Symposium, where he described how farmers at a county fair weigh pigs: "First they lay a plank across a rock and set a pig at one end. Then they heap rocks at the other end until the rocks balance the pig. Finally, they *guess* the weight of the rocks and *compute* the weight of the pig!" Our approach to program checking is similar: Instead of proving a program checker correct, we *test* it. Then we *prove* that a

to *prove* the checker *correct.* Sometimes, this is easier than proving the original program correct, as in the case of the *Extended GCD* checker of section 6. Another possibility is to try and make the checker to some extent independent of the program it checks. To this end, we make the following definition: Say that (probabilistic) program checker C has the *little oh* property with respect to program P if and only if the (expected) running time of C is little oh of the running time of P. We shall generally require that a checker have this little oh property with respect to any program it checks. The principal reason for this is to ensure that the checker is programmed *differently* from the program it checks.[4]

3. Example: Graph Isomorphism

The graph isomorphism decision problem is defined as follows:

Graph Isomorphism (GI):

Input: Two graphs G and H.

Output: YES if G is isomorphic to H; NO otherwise.

Our checker is an adaptation of Goldreich, Micali and Wigderson's [GMW] demonstration that Graph Isomorphism has interactive proofs. The [GMW] model relies on the existence of an all-powerful prover. The latter is replaced here by the program being checked. As such, ours is a concrete application of their abstract idea. Indeed, the following program checker is a sensible practical way to check computer programs for graph isomorphism. The checker, $C^P_{GI}(G, H; k)$, checks program P on input graphs G and H.

Begin
Compute $P(G, H)$.
If $P(G, H) = YES$, **then**

Use P (as if it were bug-free) to search for an 'isomorphism' from G to H.
(This is done by a standard self reduction as in Hoffmann [H].)

Check if the resulting correspondence is an isomorphism.

If not, **return** *BUGGY*; **if** yes, **return** *CORRECT*.

If $P(G, H) = NO$, **then**

Do k times:

Toss a fair coin.

If coin = heads **then**

generate a random[5] permutation G' of G.

Compute $P(G, G')$.

If $P(G, G') = NO$, **then return** *BUGGY*.

If coin = tails **then**

generate a random permutation H' of H.

Compute $P(G, H')$.

If $P(G, H') = YES$, **then return** *BUGGY*.

End-do

Return *CORRECT*.
End

The above program checker correctly tests *any* computer program *whatsoever* that is purported to solve the graph isomorphism problem. Even the most bizarre program designed specifically to fool the checker will be caught, when it is run on any input that causes it to output an incorrect answer. The following Theorem

(debugged) program checker will discover all output errors.

[4] A second reason (to ask that the checker have the "little oh" property) arises whenever the program checker is the type that runs the program P just once (to determine $O = P(I)$). In that case, P and C can be run consecutively without increasing the asymptotic running time over that of running just P.

[5] By a "random" permutation we mean that every permutation of G, i.e., every relabeling of the n nodes of G with the integers $1, \cdots, n$, is equally likely.

formally proves this:

Theorem: The program checker for Graph Isomorphism runs efficiently and works correctly (as specified). Formally:

C^P_{GI} is efficient.

Let P be any decision program (a program that halts on all inputs and always outputs YES or NO). Let G and H be any two graphs. Let k be a positive integer.

If P is a correct program for GI, ie. one without bugs, then $C^P_{GI}(G,H;k)$ will definitely output *CORRECT*.

If $P(G,H)$ is incorrect on this input, i.e., $P(G,H) \neq GI(G,H)$, then $prob\{C^P_{GI}(G,H;k) = CORRECT\}$ is at most $1/2^k$.

Proof: Clearly, C^P_{GI} runs in polynomial time.

If P has no bugs and G is isomorphic to H, then $C^P_{GI}(G,H;k)$ constructs an isomorphism from G to H and (correctly) outputs *CORRECT*.

If P has no bugs and G is not isomorphic to H, then $C^P_{GI}(G,H;k)$ tosses coins. It discovers that $P(G,G') = YES$ for all G', and $P(G,H') = NO$ for all H', and so (correctly) outputs *CORRECT*.

If $P(G,H)$ outputs an incorrect answer, there are two cases:

1. If $P(G,H) = YES$ but G is not isomorphic to H then C fails to construct an isomorphism from G to H, and so C (correctly) outputs *BUGGY*.

Finally, the most interesting case:

2. If $P(G,H) = NO$ but G is isomorphic to H, then the only way that C will return *CORRECT* is if $P(G, G' \text{ or } H') = YES$ whenever the coin comes up heads, *NO* when it comes up tails. But G is isomorphic to H. Since the permutations of G and H are random, G' has the same probability distribution as H'. Therefore, P correctly distinguishes G' from H' only by chance, i.e., for just 1 of the 2^k possible sequences of T's coin tosses.

Qed

4. Beigel's Trick

Richard Beigel [B1] has pointed out to us the following fundamental fact:

Theorem (Beigel's trick): Let π_1, π_2 be two polynomial-time equivalent computational (decision or search) problems. Then from any efficient program checker C_{π_1} for π_1 it is possible to construct an efficient program checker C_{π_2} for π_2.

Remark: The proof requires that π_1 be equivalent to π_2. It is not sufficient that one of the problems reduces to the other.

Problems that are polynomial-time equivalent to *Graph Isomorphism* include that of finding generators for the automorphism group of a graph, determining the order of the automorphism group of a graph, and counting the number of isomorphisms between two graphs. It follows from Beigel's trick that all these problems have efficient program checkers.

5. Checkers for Group Theoretic Problems

Many group theoretic problems have checkers resembling that for graph isomorphism. Subsection 5.1 shows this for two fairly general classes of examples. 5.2 gives a general approach to checker construction that works particularly well for group theoretic problems.

Why all the work on group theoretic problems? One way to show off the program checker concept is to apply it to a large interesting area in which there is substantial interest to get provably correct results. The group theoretic area, on account of both the enormous energy that has gone into the classification of finite simple groups and the interest of mathematicians to have credible computer generated output, seems a natural place to focus attention.

5.1. The Equivalence Search and Canonical Element Problems

The problems (and corresponding checkers) described in this subsection are all stated in terms of a set S of elements and a group G acting on S.

For a,b in S, define $a \equiv_G b$ if and only if

$g(a) = b$ for some g in G.

Let $ESP(S,G)$ denote the

Equivalence Search Problem

Input: a,b in S

Output: g such that $g(a) = b$ if $a \equiv_G b$; NO otherwise.

Proposition: Let $ESP(S,G)$ be the *Equivalence Search Problem*[6] for given S and G. Suppose there exists an efficient probabilistic algorithm to find a "random" g in G, where *random* means that all g in G are equally likely. Then there is an efficient program checker $C^P_{ESP(S,G)}$ for the problem $ESP(S,G)$.

Examples of the Equivalence Search Problem include graph Isomorphism, quadratic residuosity, a generalization of discrete log and games such as Rubic's cube. Other examples arise in knot theory, block designs, codes, matrices over GF (q), Latin Squares [L2, p. 32] and in applications of Burnside and Polya theorems [PR].

Canonical Element Problem (CEP)

Input: a in S

Output: (c,g) where c is a canonical element (the unique canonical element) in the equivalence class of a, and g in G satisfies $g(a) = c$.

Proposition: There is an efficient program checker for the canonical element problem, provided there is a probabilistic procedure to select a random g in G efficiently.

Remark: If the CEP program should fail by having two or more canonical elements in some class, then we define the (true) canonical element of that class to be the unique element, if any, to which more than half the elements of the class are mapped by the program.

5.2. The Vorpal Blade Went Snicker-Snack

A 1-2 approach is useful for constructing program checkers for group theoretic problems:

1. Design an interactive protocol (cf. Goldwasser, Micali, and Rackoff [GMR]) for proving correctness of an algorithm's output.
2. Modify said interactive protocol into a checker.

It is possible to design checkers for several group theoretic problems in this way. An example is the checker for

The Group Intersection Problem:

Input: Two permutation groups, G and H, specified by generators. The generators are presented as permutations of $[1, \cdots, n]$.

Output: Generators for $G \cap H$.

No probabilistic polynomial time algorithm is known for solving this problem, which is not surprising since graph isomorphism is polynomial time reducible to group intersection.

---

6 Related to the *Equivalence Search Problem* is the *Equivalence Decision Problem* defined by:

Equivalence Decision Problem (EDP)

Instance: a,b in S

Question: Is $a \equiv_G b$?

The search problem, not the decision problem, is required in the above theorem since the decision problem is not expected to be an *NP-Search-Solving-Decision* problem. Why? Recall that for N a positive integer, Z^*_N denotes the group of positive integers less than N that are relatively prime to N under multiplication mod N. For p a prime, let $S = Z^*_p$ and $G = Z^*_{p-1}$, where the action of g in G on a in S maps a to $a^g \bmod p$. Observe that $a \equiv_G b$ if and only if $b = a^g \bmod p$ for some g in Z^*_{p-1}. To find g is to solve the discrete log problem, which in cryptographic circles is believed to not be solvable in polynomial time, even given an oracle for factoring [A]. On the other hand, the *EDP is* solvable in polynomial time given an oracle for factoring. The proof consists in showing that $b = a^g \bmod p$ for some g if and only if order(b) | order(a). This is because $x^{\text{order}(a)} = 1 \bmod p$ has exactly order(a) solutions, namely $a, a^2, \cdots, a^{\text{order}(a)} \equiv 1$.

Finally, order(a) and order(b) can be determined from the factorization of $p - 1$.

IP Protocol

Begin

1. The prover sends the verifier a set of permutations of $[1, \cdots, n]$ which supposedly generate $G \cap H$.
2. The verifier checks that the elements sent by the prover actually lie in $G \cap H$. This involves testing membership in G and H, which the verifier can do [FHL]. As a consequence, the verifier is convinced that the elements sent by the prover either generate $G \cap H$ or a proper subgroup of it.

The next phase of the protocol is aimed towards giving the verifier a random element of $G \cap H$. Before going into it we introduce the following (standard) notation: with G and H as above GH represents the set of permutations $\{\pi \mid \pi = ab$ where $a \in G$ and $b \in H\}$. Here is the continuation of the protocol.

3. The verifier sends to the prover an element, π, of GH, which he obtains by selecting random elements of G and H and multiplying them together.
4. The prover sends back a 'factorization' of π as $a'b'$ where $a' \in G$ and $b' \in H$.

Lemma: $a^{-1}a'$ is a random element of $G \cap H$.

5. Now that the verifier has a random element of $G \cap H$, he tests it for membership in the group generated by the elements sent by the prover. If that group is a proper subgroup of $G \cap H$ its size is at most $|G \cap H|/2$. Hence the verifier has a probability of at least 1/2 of detecting this. If however the elements actually generate $G \cap H$ the verifier will be convinced of this after a number of trials.

End

Converting the IP Protocol into a Checker

The verifier in the above protocol asks the prover to 'factor' certain elements of GH. To convert this IP protocol into a checker presents a problem in that a checker is only allowed to run the program on various inputs to the program. If the Factorization Search Problem (FSP) were shown equivalent to the group intersection problem then using Beigel's trick we would have our desired checker.

Factorization Search Problem (FSP)

Input: Two permutation groups G and H specified by generators, and a permutation π.

Output: No, if π is not in GH.

a, b such that $a \in G$ and $b \in H$ and $ab = \pi$ otherwise.

The associated *Factoring Decision Problem (FDP)* is known to be equivalent to group intersection, cf. Hoffmann [H]. We state without proof the following lemma:

Lemma: FDP is equivalent to FSP.

The proof of this lemma relies on the notion of 'strong generators' for a group introduced in Furst, Hopcroft and Luks [FHL].

The above lemmas and Beigel's trick together imply

Theorem: There is a polynomial time checker for the group intersection problem.

6. Problems in P

In this section, some program checkers use their oracle just once (to determine $O = P(I)$) rather than several times. In such cases, instead of the program checker being denoted by $C_\pi^P(I;k)$, it will be denoted by $C_\pi(I, O;k)$. The latter notation has the advantage of clarifying what must be tested for. In cases where the checker is nonprobabilistic, it will be denoted by $C_\pi(I, O)$ instead of $C_\pi(I, O;k)$.

Many problems in P have efficient program checkers, and it is a challenge to find them. In what follows, we give a fairly complete description of program checkers for just three problems in P : *Extended GCD* (because it has one of the oldest nontrivial algorithms on the books), *Sorting* (because it is one of the most frequently run algorithms), and *Matrix rank* (because it is most unusual in that it seems to require a *multicall* checker with *two-sided* error).

6.1. A Program Checker for Extended GCD

Extended GCD

Input: Two positive integers a,b.

Output: $d = gcd\,(a,b)$, and integers u,v such that $a \cdot u + b \cdot v = d$.

Observe that *Extended GCD* is a specific computational problem, not an algorithm. In particular, it does not have to be solved by Euclid's algorithm. Problems *GCD* and *Extended GCD* both output $d = gcd(a,b)$, but *Extended GCD* also outputs two integers u,v such that $a \cdot u + b \cdot v = d$. We know how to check programs for *Extended GCD* but not *GCD*.

For our model of computation, we choose a standard RAM and count only arithmetic operations $+, -, \cdot, /$ as steps.

$C_{Extended\ GCD}$

Input: positive integers a,b; positive integer d and integers u,v.

Output: BUGGY if $d \nmid a$ or $d \nmid b$ or $a \cdot u + b \cdot v \neq d$,

CORRECT otherwise (the given program P computes *Extended GCD* correctly).

The desired input-output relationship above can be achieved by a checker that takes just 5 steps: 2 division steps + 2 multiplication steps + 1 addition step. This checker can also be proved correct easily.

6.2. A Program Checker for Sorting

Sorting

Input: An array of integers $X = [x_1, \cdots, x_n]$, representing a multiset.

Output: An array Y consisting of the elements of X listed in nondecreasing order.

A checker for *Sorting* must do *more* than just check that Y is in order. It must also check that $X = Y$ as multisets.

$C_{Sorting}$

Input: Two arrays of integers $X = [x_1, \cdots, x_n]$ and $Y = [y_1, \cdots, y_{n'}]$

Output: BUGGY if Y is *not* in order or if $X \neq Y$ as multisets.

CORRECT if the given program P correctly sorts.

This check is easy in a RAM model of computation, but a RAM does not reflect many sorting scenarios. The following one does.

Model of computation: The computer has a fixed number of tapes, including one that contains X and another that contains Y. X and Y each have at most n elements, and each such element is an integer in the range $[0,a]$. The random access memory has $O(lg\,n + lg\,a)$ words of memory, and each of its words is capable of holding any integer in the range $[0,a]$, in particular each can hold any element of $X \cup Y$.

Single precision operations: $+, -, \times, /, <, =$ each take one step. Here, $/$ denotes "integer divide."

Multi-precision operations: $+, -, <, =$ on integers that are m words long take m steps; $\times, /$ on integers that are m words long take m^2 steps.

In addition the machine can do the usual tape operations: Each shift of a tape, and each copy of a word on tape to a word in RAM or vice versa counts 1 step.

In this model of computation, it is easy to check that Y is in order in just $O(n)$ steps. To check that $X = Y$ as multisets can be done in probabilistic $O(n)$ steps, but the right method depends in general on the the comparative sizes of a and n. Here is a precise statement of what is wanted:

Multiset Equality Test

Input: Two arrays of integers X, Y; and a positive integer k.

Output: YES if X and Y represent the same multiset, NO otherwise, with probability of error $\leq 1/2^k$. The latter means that for any two sets X and Y selected by adversary, the probability of an incorrect output (measured over the distribution of coin-toss-sequences generated by the algorithm) should be at most $1/2^k$.

Method 1: This method (but not the specific and important choice of hash function) was first suggested by Wegman and Carter [WC]: Compute $n = |X|$ and check that $|Y| = n$. If so, select a hash function $h : Z \to \{0,1\}$ and compare $h(x_1) + \cdots + h(x_n)$ to $h(y_1) + \cdots + h(y_n)$.

If h is random and $X \neq Y$ then with probability at least 1/2 the above two sums will differ.[7]

Choosing an easy-to-compute random-enough hash function is difficult[8]. The Wegman and Carter hash function in particular requires a random access memory, so it cannot be implemented in the above tape model of computation. Here is a new approach that is guaranteed to work: Recall that $n = |X| = |Y|$. Let $m = n + 1$. a = largest possible value for any element of $X \cup Y$. Select prime p at random from the interval $[1, 3 \cdot a \cdot \log m]$. Set $h(x) = m^x \bmod p$. Observe that $X = Y$ if and only if $\sum m^{x_i} = \sum m^{y_i}$. Indeed, if $X = Y$, then $\sum (m^{x_i} \bmod p) = \sum (m^{y_i} \bmod p)$ for all primes p. If $X \neq Y$, then $\sum m^{x_i} \neq \sum m^{y_i}$ and so, as pointed out by Karp and Rabin [KR], $\sum m^{x_i} \not\equiv \sum m^{y_i} \bmod p$ for at least half of all primes in the interval $[1, 3 \cdot a \cdot \log m]$, and therefore $\sum (m^{x_i} \bmod p) \neq \sum (m^{y_i} \bmod p)$ for those primes. This proves that this h is a random-enough hash function.

Method 2: This idea was first suggested by Lipton [L3] and more recently by Ravi Kannan [K1]. Let $f(z) = (z - x_1) \cdots (z - x_n)$ and $g(z) = (z - y_1) \cdots (z - y_n)$. Then $X = Y$ as multisets if and only if $f = g$. Since f and g are polynomials of degree n, either $f(z) = g(z)$ for all z (if $f = g$) or $f(z) = g(z)$ for at most n values of z (if $f \neq g$). A probabilistic algorithm can decide if $f = g$ by selecting k values for z at random from a set of $2n$ (or more) possibilities, say from $[1, 2n]$, then comparing $f(z)$ to $g(z)$ for these k values. The computations can be kept to reasonable size by doing the subtractions and multiplications modulo randomly chosen (small) primes.

Comparison of methods 1 & 2 for checking multiset equality:

Recall that each multiset has at most n integers, each in the range $[0, a]$. In method 1, primes are $O(a \log n)$; in method 2, they are $O(n \log a)$. The lengths of the primes are respectively $O(\log a + \log\log n)$ and $O(\log n + \log\log a)$. If n is bigger than 2^a a simple bucket sort works in

---

[7] To see this, remove from X and Y any largest sub-multiset of elements that is common to both. The resulting X and Y are still the same size and their intersection is empty. Compute $\sum_{x \neq x_i} h(x_i)$ and compute $\sum_{y} h(y_i)$. If the two sums are equal, then setting $h(x_1) = 1$ will distinguish X from Y; if different, then setting $h(x_1) = 0$ will distinguish the two.

[8] A common way to construct hash functions is to select a prime p and integers a, b at random, then to set $h(x) = a \cdot x + b \bmod p$. This will not work for this problem. To see why not, suppose that addition of hash functions is done mod p. In that case, an adversary can choose $X \neq Y$ so that $\sum x_i = \sum y_i$, whence it follows that $\sum h(x_i) \equiv \sum h(y_i) \bmod p$. This hash function also fails when ordinary addition is used.

linear time and we don't need to use either of these two methods. For n less than 2^a the primes fit in a constant number of words in method 1. In method 2 the number of words to hold a prime is $k = O(\max(1, \log n / \log a))$. The running time of method 1 is $O(n \log a)$. (We need to perform $\log a$ multiplications to compute m^x each costing 1 time step.) The running time for method 2 is $O(n k^2)$ (Since we multiply numbers that are k words long).

To achieve the little oh property one must make the choice of method carefully. If $n < a$ method 2 runs in linear time since $k = 1$. We use method 2 as long as $n < a^{\log\log a}$. The running time of method 2 is then $O(n (\log\log a)^2)$ which is $O(n (\log\log n)^2)$. When n gets bigger than $a^{\log\log a}$ the running time of method 1 which is $n \log a$ becomes $o(n \log n)$.

6.3. A Program Checker for Matrix Rank

Matrix Rank

Input: An m x n matrix M with elements from a finite field F.

Output: An integer r, which is the rank of the matrix.

We assume in our model of computation that we can generate a random element of F in one time step. The difficulty in designing a checker for this problem is achieving the little oh property. For instance, a commonly used operation, that of forming random linear combinations of the columns of the matrix, takes time $O(n^2)$. So this cannot be done too many times. In this subsection we present an outline of the checker.

The checker is in two parts. The first part ensures that the rank of M is at least r and the second part verifies that the rank is at most r.

6.3.1. Checking that Rank is at least r

To check that the rank is at least r, use the program to obtain (by self reduction) an r by r submatrix, A, which is supposedly of full rank. It must be checked that A is really of full rank. The key fact used here is that any vector x in span(A), the span of the columns of A has a *unique* representation as a linear combination of the columns of A iff A is of full rank. In particular if A is not of full rank there is a column a_l of A such that any x in the span(A) can be expressed as a linear combination of the columns with any desired coefficient for a_l.

Create k (= the security parameter) random vectors, $y_1, \ldots, y_k$ in span(A). Let c_{ij} be the coefficient of column a_i of A in y_j. If A is not of full rank, then for column a_l as in the previous paragraph the following two cases are indistinguishable:

1. Setting $y_{lj} = y_j - c_{lj} \cdot a_l$ and using the program to find the rank of A without column a_l but with the vector y_{lj}.
2. Setting $y_{lj} = y_j - b \cdot a_l$ where b is a random element of F, $b \neq c_{lj}$ and using the program to find the rank of A without column a_l but with the vector y_{lj}.

However when A is of full rank these cases are clearly distinguishable to the program. Using these ideas one can prove

Lemma: Checking that a matrix is of full rank can be done in $O(n^2 k)$ time.

6.3.2. Checking that the rank is at most r

The program above obtained r independent columns such that all other columns are supposedly dependent on them. Let these independent columns be $a_1, \ldots, a_r$. These columns are vectors in F^n. One can augment this set of columns by random vectors in F^n so as to have n vectors in all.

Lemma: If $a_1, \ldots, a_r$ are independent, then with probability greater than a positive constant $(\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{7}{8} \cdots = .28\ldots)$, the n vectors obtained by augmenting $a_1, \ldots, a_r$ with random vectors $b_{r+1}, \ldots, b_n$ forms a basis for F^n.

Two other facts are used in the construction of this checker.

1. To check that the originally given $a_{r+1}, \ldots, a_m$ are dependent on $a_1, \ldots, a_r$, it is sufficient (probabilistically) to check that each of k random linear combinations of $a_{r+1}, \ldots, a_n$ is dependent.
2. If x is a vector independent of $a_1, \ldots, a_r$, then in its (unique) expression as

$$x = \sum_{i=1}^{r} c_i \cdot a_i + \sum_{i=r+1}^{n} c_i \cdot b_i,$$

each c_i for $r+1 \le i \le n$ has a probability $\ge$ 1/2 of being nonzero. Here the probability is over the possible choices of $b_{r+1},\ldots,b_n$.

Using these results one can prove

Theorem: There is a checker for Matrix Rank running in time $O(n^2k^3)$.

7. Checker Characterization Theorem

We take as our definition of IP (Interactive Proof-System) the definition appearing in Goldwasser, Micali, and Rackoff [GMR], except that we replace "for all sufficiently large x" in that definition by "for all x". This modification of GMR conforms with the commonly accepted definition of IP as it appears, for example, in Goldwasser and Sipser [GS], and Tompa and Woll [TW].

Define *function-restricted IP* (CO-IP) = the set of all decision problems for which there is an interactive probabilistic expected-polynomial-time proof system for YES-instances (NO-instances) of π satisfying the conditions that prover (= any honest prover) must compute the function π and $\overline{\text{prover}}$ (= any dishonest prover) must be a function from the set of instances of π to {YES, NO}. This restriction implies two things:

1. verifier may only ask questions that are instances of π, and
2. $\overline{\text{prover}}$ (and prover) must answer each of verifier's questions with an answer that is independent of $\overline{\text{prover}}$'s (prover's) previous history of questions and answers.

Because of restriction 1 on verifier and 2 on prover, it is not clear whether IP is contained in function-restricted IP. Because of restriction 2 on $\overline{\text{prover}}$, it is not clear whether function-restricted IP is contained in IP.

Theorem: An efficient program checker C_π exists for decision problem $\pi <=> \pi$ lies in *function−restricted IP* $\cap$ *function−restricted CO−IP*.

Let *NP-search* denote the class of problems π such that $\pi(x) = NO$ if x is a NO-instance; YES together with a *proof* that x is a YES-instance otherwise.

Corollary: Let π be an NP-search problem. An efficient program checker C_π exists for $\pi \Leftrightarrow \pi$ is in function-restricted co-IP.

The main purpose of the above corollary is to point out that if $NP \subseteq$ function-restricted co-IP, as seems likely, then there can be no efficient program checker C_π (in the above sense) for NP-complete problems!

8. Overview and Conclusions

The thrust of this paper is to show that in many cases, it is possible to check a program's output on a given input, thereby giving *quantitative mathematical evidence* that the program works correctly on that input. By allowing the possibility of an incorrect answer (just as one would if computations were done by hand), the program designer confronts the possibility of a bug and considers *what to do* if the answer is wrong. This gives an *alternative* to proving a program correct that may be achievable and sufficient for many situations.

A proper way to develop this theory would be to require that the program checker itself be proved correct. This paper, however, is about *pure* checking, meaning no proofs of correctness whatsoever. Instead, we require the checker C to be different from the program P that it checks in two ways: First, the input-output specifications for C are different from those for P (C gets P's output and it responds *CORRECT* or *BUGGY*). Second, we demand that the running time of the checker be $o(S)$, where S is the running time of the program being checked. This prevents a programmer from undercutting this approach, which he could otherwise do by simply running his program a second time and calling that a check. Whatever else the programmer does, he *must* think more about his problem.

References

[A] D. Angluin, "Lecture Notes on the Complexity of Some Problems in Number Theory," Yale Technical Report #243 (1982).

[B1] R. Beigel, personal communication.

[B2] M. Blum, "Designing Programs to Check Their Work", submitted for publication in CACM.

[BA] T.A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," Acta Informatica, v. 18, 31-45 (1982).

[BCG] E.R. Berlekamp, J.H. Conway, and R.K. Guy, "Winning Ways," Academic Press, (1982).

[BR] M. Blum and P. Raghavan, "Program Correctness: Can One Test for It?," IBM T.J. Watson Research Center Technical Report (1988).

[CFGMW] L. Carter, R. Floyd, J. Gill, G. Markowsky & M. Wegman, "Exact and Approximate Membership Testers" 10th ACM Symposium on Theory of Computing, 59-65 (1978).

[F] R. Freivalds, "Fast Probabilistic Algorithms," Springer Verlag Lecture Notes in CS #74, Mathematical Foundations of CS, 57-69 (1979).

[FHL] M. Furst, J.E. Hopcroft, E. Luks, "Polynomial-Time Algorithms for Permutation Groups," Proc. 21st IEEE Symposium on Foundations of Computer Science, 36-41 (1980).

[GJ] M.R. Garey and D.S. Johnson, "Computers and Intractability," Freeman, San Francisco, CA (1979).

[GMR] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof Systems," 17th ACM Symposium on Theory of Computing, 291-304 (1985).

[GMW] O. Goldreich, S. Micali and A. Wigderson, "Proofs that Yield Nothing but their Validity and a Methodologyof Cryptographic Design," Proc. 27th IEEE Symposium on Foundations of Computer Science, 174-187 (1986).

[GS] S. Goldwasser and M. Sipser, "Private Coins versus Public Coins in Interactive Proof Systems," 18th ACM Symposium on Theory of Computing, 59-68 (1986).

[H] C.M. Hoffmann, "Group-Theoretic Algorithms and Graph Isomorphism," V. 136 of the series "Lecture Notes in Computer Science," ed. by G. Goos and J. Hartmanis, Springer-Verlag, 311 pp. (1982).

[K1] R. Kannan, personal communication through S. Rudich.

[K2] S. Kannan, Ph.D. thesis to be submitted to the Computer Science Division of the University of California at Berkeley.

[KR] R.M. Karp and M.O. Rabin, "Efficient randomized pattern matching algorithms", IBM Journal of Research and Development, 31(2), 249-260.

[L1] J. Leech, "Computer Proof of Relations in Groups," in "Topics in Group Theory and Computation," edited by M.P.J. Curran, Academic Press, 38-61 (1977).

[L2] J.S. Leon, "Computing Automorphism Groups of Combinatorial Objects," in "Computational Group Theory," edited by M.D. Atkinson, Academic Press, 321-335 (1984).

[L3] R. Lipton, personal communication.

[PR] G. Polya and R.C. Read, "Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds," Springer-Verlag (1987).

[R] M.O. Rabin, "Probabilistic Algorithms," in "Algorithms and Complexity, Recent Results and New Directions," edited by J.F. Traub, Academic Press, 21-40 (1976).

[TW] M. Tompa and H. Woll, "Random Self-Reducibility and Zero Knowledge Interactive Proofs of Possession of Information," Proc. 28th IEEE Symposium on Foundations of Computer Science, 472-482 (1987).

[W] W.J. Weyuker, "The Evaluation of Program-Based Software Test Data Adequacy Criteria," Communications of the ACM, v. 31, no. 6, 668-675 (1988).

[WC] M.N. Wegman and J.L. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," J. of Computer and System Science, v. 22, no. 3, 265-279 (1981).