8.6 Program Checking

The discovery of the interactive protocol for $\#\mathsf{SAT}_{\mathrm{D}}$ was triggered by a research area called *program checking*, sometimes also called *instance checking*. Blum and Kannan initiated this area motivated by the observation that even though program verification —deciding whether or not a given program solves a certain computational task on all inputs— is undecidable, in many situations it would suffice to have a weaker guarantee of the program's "correctness" on an input-by-input basis. This is encapsulated in the notion of a *program checker*. A checker for a program P is another program that may run P as a subroutine. Whenever P is run on an input, C's job is to detect if P's answer is incorrect ("buggy") on that particular input. To do this, the checker may also compute P's answer on some other inputs. Formally, the checker C is a TM that expects to have the code of another program, which it uses as a black box. We denote by C^P the result of running C when it is provided P as a subroutine.

Definition 8.26 Let T be a computational task. A *checker* for T is a probabilistic polynomial time TM C that, given any program P that is a claimed program for T and any input x, has the following behavior:

1. If P is a correct program for T (i.e., $\forall y \;\; P(y) = T(y)$), then $P[C^P \text{ accepts } P(x)] \geq \frac{2}{3}$
2. If $P(x) \neq T(x)$ then $P[C^P \text{ accepts } P(x)] < \frac{1}{3}$ $\diamondsuit$

Note that checkers don't certify the correctness of a program. Furthermore, even in the case that P is correct on x (i.e., $P(x) = C(x)$) but the program P is not correct on inputs other than x, the output of the checker is allowed to be arbitrary.

Surprisingly, for many problems checking seems easier than actually computing the problem. Blum and Kannan suggested that one should build such checkers into the software for these problems; the overhead introduced by the checker would be negligible and the program would be able to automatically check its work.

Example 8.27 *(Checker for Graph Non-Isomorphism)*
The input for the problem of Graph Non-Isomorphism is a pair of labeled graphs $\langle G_1, G_2 \rangle$, and the problem is to decide whether $G_1 \cong G_2$. As noted, we do not know of an efficient algorithm for this problem. But it has an efficient checker. There are two types of inputs depending upon whether or not the program claims $G_1 \cong G_2$. If it claims that $G_1 \cong G_2$ then one can change the graph little by little and use the program to actually obtain a permutation π mapping G_1 to G_2 (or if this fails, finds a bug in the program; see Exercise 8.11). We now show how to check the claim that $G_1 \ncong G_2$ using our earlier interactive proof of Graph non-isomorphism.
Recall the IP for Graph Non-Isomorphism:

- In case prover admits $G_1 \ncong G_2$ repeat k times:
- Choose $i \in_R \{1, 2\}$. Permute G_i randomly into H
- Ask the prover whether G_1, H are isomorphic and check to see if the answer is consistent with the earlier answer.

Given a computer program P that supposedly computes graph isomorphism, how would we check its correctness? The program checking approach suggests we use an IP while regarding the program as the prover. Let C be a program that performs the above protocol using as prover the claimed program P.

Theorem 8.28 *If P is a correct program for Graph Non-Isomorphism then C outputs "correct" always. Otherwise, if $P(G_1, G_2)$ is incorrect then $P[C$ outputs "correct" $] \leq 2^{-k}$. Moreover, C runs in polynomial time.*

8.6.1 Languages that have checkers

Whenever a language L has an interactive proof system where the prover can be implemented using oracle access to L, this implies that L has a checker. Thus, the following theorem is a fairly straightforward consequence of the interactive proofs we have seen:

Theorem 8.29 *The problems Graph Isomorphism (*GI*), #*SAT_D *and True Quantified Boolean Formulae (*TQBF*) have checkers.* ♢

Similarly, it can be shown [Rub90] that problems that are random self-reducible and downward self-reducible also have checkers. (For a definition of downward self-reducibility, see Exercise 8.9.)

Using the fact that **P**-complete languages are reducible to each other via **NC**-reductions (in fact, even via the weaker logspace reductions), it suffices to show a checker in **NC** for one **P**-complete language (as was shown by Blum and Kannan) to obtain the following interesting fact:

Theorem 8.30 *For any* **P***-complete language there exists a program checker in* **NC**

Since we believe that **P**-complete languages cannot be computed in **NC**, this provides additional evidence that checking is easier than actual computation.

Blum and Kannan actually provide a precise characterization of languages that have checkers using interactive proofs, but it is omitted here because it is technical.

8.6.2 Random Self Reducibility and the Permanent

Most checkers are designed by observing that the output of the program at x should be related to its output at some other points. The simplest such relationship, which holds for many interesting problems, is random self-reducibility.

Roughly speaking, a problem is *random-self-reducible* if solving the problem on any input x can be reduced to solving the problem on a sequence of random inputs $y_1, y_2, \ldots$, where each y_i is uniformly distributed among all inputs. (The correct definition is more general and technical but this vague one will suffice here.) This property is important in understanding the average-case complexity of problems, an angle that is addressed further in Theorem 8.33 and Section 19.4.

Example 8.31
Suppose a function $f : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$ is linear, that is there exist coefficients $a_1, a_2, \ldots, a_n$ such that $f(x_1, x_2, \ldots, x_n) = \sum_i a_i x_i$.
Then for any $\mathbf{x}, \mathbf{y} \in \mathrm{GF}(2)^n$ we have $f(\mathbf{x}) + f(\mathbf{y}) = f(\mathbf{x} + \mathbf{y})$. This fact can be used to show that computing f is *random self-reducible*. If we want to compute $f(\mathbf{x})$ where x is aribitrary, it suffices to pick a random $\mathbf{y}$ and compute $f(\mathbf{y})$ and $f(\mathbf{x} + \mathbf{y})$, and both $\mathbf{y}$ and $\mathbf{x} + \mathbf{y}$ are random vectors (though not independently distributed) in $\mathrm{GF}(2)^n$ (Aside: this simple observation will appear later in a different context in the linearity test of Chapter 11.)

The above example may appear trivial, but in fact some very nontrivial problems are also random self-reducible. The *permanent* of a matrix is superficially similar to the determinant and defined as follows:

Definition 8.32 Let $A \in F^{n \times n}$ be a matrix over the field F. The permanent of A is:

$$\mathsf{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{16}$$

♢

The problem of calculating the permanent is clearly in **PSPACE**. In Chapter 17 we will see that computing the permanent is **#P**-complete, which means that it is essentially equivalent to $\#\mathsf{SAT}_{\mathrm{D}}$. In particular, if the permanent can be computed in polynomial time then $\mathbf{P} = \mathbf{NP}$. Here we show that the permanent is random self-reducible. The main observation used is that if we think of $\mathsf{perm}(A)$ as a function of n^2 variables (denoting the entries of the matrix A) then by (16) this function is a polynomial of degree n.

Theorem 8.33 *(Lipton* [Lip91]*) There is a randomized algorithm that, given an oracle that can compute the permanent on $1-\frac{1}{3n}$ fraction of the inputs in $\mathbb{F}^{n\times n}$ (where the finite field $\mathbb{F}$ has size $> 3n$), can compute the permanent on all inputs correctly with high probability.*$\diamondsuit$

Proof: Let A be some input matrix. Pick a random matrix $R \in_R \mathbb{F}^{n\times n}$ and let $B(x) = A + x \cdot R$ for a variable x. Notice that:

- $\mathsf{perm}(B(x))$ is a degree n univariate polynomial.
- For any fixed $a \neq 0$, $B(a)$ is a random matrix, and hence the probability that the oracle computes $\mathsf{perm}(B(a))$ correctly is at least $1 - \frac{1}{3n}$.

Now the algorithm for computing the permanent of A is straightforward. Fix any $n+1$ distinct points $a_1, a_2, \ldots, a_{n+1}$ in the field and query the oracle on all matrices $\{B(a_i)|1 \leq i \leq n+1\}$. According to the union bound, with probability of at least $1 - \frac{n+1}{n} \approx \frac{2}{3}$ the oracle will compute the permanent correctly on all matrices.
Recall the fact (see Theorem A.35 in Appendix A) that given $n+1$ (point, value) pairs $\{(a_i, b_i)|i \in [n+1]\}$, there exists a unique a degree n polynomial p that satisfies $\forall i \;\; p(a_i) = b_i$. Therefore, given that the values $B(a_i)$ are correct, the algorithm can interpolate the polynomial $B(x)$ and compute $B(0) = \mathsf{perm}(A)$. ■

The hypothesis of Theorem 8.33 be weakened so that the oracle only needs to compute the permanent correctly on a fraction of $\frac{1}{2} + \varepsilon$ for any constant $\varepsilon > 0$ of the inputs. This uses a stronger interpolation theorem; see Section 19.6.

8.7 Interactive proof for the Permanent

Although the existence of an interactive proof for the Permanent follows from that for $\#\mathsf{SAT}$ and TQBF, we describe a specialized protocol as well. This is both for historical context (this protocol was discovered before the other two protocols) and also because this protocol may be helpful for further research.

The protocol will use the random self-reducibility of the permanent and downward self-reducibility, a property encountered earlier in Chapter 2 in the context of SAT (see also Exercise 8.9). In case of permanent, this is the observation that

$$\mathsf{perm}(A) = \sum_{i=1}^{n} a_{1i}\mathsf{perm}(A_{1,i}),$$

where $A_{1,i}$ is a $(n-1)\times(n-1)$ sub-matrix of A obtained by removing the first row and i'th column of A (recall that the analogous formula for the determinant uses alternating signs). Thus computing the $n \times n$ permanent reduces to computing n permanents of $(n-1)\times(n-1)$ matrices.

For ease of notation, we assume the field $\mathbb{F}$ is equal to $\mathrm{GF}(p)$ for some prime $p > n$, and so $1, 2, \ldots, n \in \mathbb{F}$, and reserve a_{ij} for the (i,j)th element of the matrix. For every $n \times n$ matrix A, and $i \in [n]$, we define $D_A(i)$ to be the $(n-1)\times(n-1)$ matrix $A_{1,i}$. If $x \in \mathbb{F} \setminus [n]$, then we define $D_A(x)$ in the unique way such that for every $j, k \in [n-1]$, the function $(D_A(x))_{j,k}$ is a univariate polynomial of degree at most n. Note that since the permanent of an $(n-1)\times(n-1)$ matrix is a degree-$(n-1)$ polynomial in the entries of the matrix, $\mathsf{perm}(D_A(x))$ is a univariate polynomial of degree at most $(n-1)n < n^2$.

8.7.1 The protocol

We now show an interactive proof for the permanent. Specifically, define L_{perm} to contain all tuples $\langle A, p, k\rangle$ such that $p > n^4$ is prime, A is an $n \times n$ matrix over $\mathrm{GF}(p)$, and $\mathsf{perm}(A) = k$. We prove the following theorem:

Theorem 8.34 $L_{\mathsf{perm}} \in \mathbf{IP}$ ♢

PROOF: The proof is by induction— we assume that we have an interactive proof for matrices up to size $(n-1)$, and show a proof for $n \times n$ matrices. That is, we assume inductively that for each $(n-1) \times (n-1)$ matrix B, the prover can make the verifier accept the claim $\mathsf{perm}(B) = k'$ with probability 1 if it is true and with probability at most ϵ if it is false. (Clearly, in the base case when $n - 1 = 1$, the permanent computation is trivial for the verifier and hence $\epsilon = 0$ in the base case.) Then we show that for every $n \times n$ matrix A the prover can make the verifier accept the claim $\mathsf{perm}(A) = k$ with probability 1 if it is true and with probability at most $\epsilon + (n-1)^2/p$ if it is false. The following simple exchange shows this.

- *Round 1:* Prover sends to verifier a polynomial $g(x)$ of degree $(n-1)^2$, which is supposedly $\mathsf{perm}(D_A(x))$.

- *Round 2:* Verifier checks whether: $k = \sum_{i=1}^{m} a_{1,i} g(i)$. If not, it rejects at once. Otherwise, the verifier picks a random element of the field $b \in_R \mathbb{F}_p$ and asks the prover to prove that $g(b) = \mathsf{perm}(D_A(b))$. Notice, $D_A(b)$ is an $(n-2) \times (n-2)$ matrix over $\mathbb{F}_p$, and so now use the inductive hypothesis to design a protocol for this verification.

Now we analyze this protocol. If $\mathsf{perm}(A) = k$, then an all-powerful prover can provide $\mathsf{perm}(D_A(x))$ and thus by the inductive hypothesis make the verifier accept with probability 1.

On the other hand, suppose that $\mathsf{perm}(A) \neq k$. If in the first round, the polynomial $g(x)$ sent is the correct polynomial $\mathsf{perm}(D_A(x))$, then:

$$\sum_{i=1}^{m} a_{1,i} g(i) = \mathsf{perm}(A) \neq k,$$

and the verifier would immediately reject. Hence we only need to consider a prover that sends $g(x) \neq \mathsf{perm}(D_A(x))$. Since two polynomials of degree $(n-1)^2$ can only agree for less than $(n-1)^2$ values of x, the chance that the randomly chosen $b \in \mathbb{F}_p$ is one of them is at most $(n-1)^2/p$. if b is not one of these values, then the prover is stuck with proving an incorrect claim, which by the inductive hypothesis he can prove with conditional probability at most ϵ. This finishes the proof of correctness.

Unwrapping the inductive claim, we see that the probability that the prover can convince this verifier about an incorrect value of the permanent of an $n \times n$ matrix is at most

$$\frac{(n-1)^2}{p} + \frac{(n-2)^2}{p} + \cdots + \frac{1}{p} \leq \frac{n^3}{p},$$

which is much smaller than $1/3$ for our choice of p. ■

WHAT HAVE WE LEARNED?

- An *interactive proof* is a generalization of mathematical proofs in which the prover and polynomial-time probabilistic verifier interact.
- Allowing randomization and interaction seems to add significantly more power to proof system: the class **IP** of languages provable by a polynomial-time interactive proofs is equal to **PSPACE**.
- All languages provable by a *constant round* proof system are in the class **AM**: that is, they have a proof system consisting of the the verifier sending a single random string to the prover, and the prover responding with a single message.
- Interactive proofs have surprising connections to cryptography, approximation algorithms (rather, their nonexistence), and program checking.

Chapter notes and history

Interactive proofs were defined in 1985 by Goldwasser, Micali, Rackoff [GMR85] for cryptographic applications and (independently, and using the public coin definition) by Babai [Bab85]; see also Babai and Moran [BM88]. The private coins interactive proof for graph non-isomorphism was given by Goldreich, Micali and Wigderson [GMW87]. Simulations of private coins by public coins (Theorem 8.12) were given by Goldwasser and Sipser [GS87] (see [Gol08, Appendix A] for a good exposition of the full proof). It was influenced by earlier results such as $\mathbf{BPP} \subseteq \mathbf{PH}$ (Section 7.5.2) and the fact that one can approximate $\#\mathsf{SAT}_{\mathrm{D}}$ in $p^{\Sigma_2^p}$. *Multi-prover* interactive proofs were defined by Ben-Or et al [BOGKW88] for the purposes of obtaining zero knowledge proof systems for **NP** (see also Section 9.4) without any cryptographic assumptions.

The general feeling at the time was that interactive proofs are only a "slight" extension of **NP** and that not even $\overline{\mathsf{3SAT}}$ has interactive proofs. For example, Fortnow and Sipser [FS88] conjectured that this is the case and even showed an oracle O relative to which $\mathbf{coNP}^O \not\subseteq \mathbf{IP}^O$ (thus in the terms of Section 3.4, $\mathbf{IP} = \mathbf{PSPACE}$ is a *non-relativizing* theorem).

The result that $\mathbf{IP} = \mathbf{PSPACE}$ was a big surprise, and the story of its discovery is very interesting. In the late 1980s, Blum and Kannan [BK95] introduced the notion of program checking. Around the same time, manuscripts of Beaver and Feigenbaum [BF90] and Lipton [Lip91] appeared that fleshed out the notion of random self reducibility and the connection to checking. Inspired by some of these developments, Nisan proved in December 1989 that the permanent problem (hence also $\#\mathsf{SAT}_{\mathrm{D}}$) has *multiprover* interactive proofs. He announced his proof in an email to several colleagues and then left on vacation to South America. This email motivated a flurry of activity in research groups around the world. Lund, Fortnow, Karloff showed that $\#\mathsf{SAT}_{\mathrm{D}}$ is in **IP** (they added Nisan as a coauthor and the final paper is [LFKN90]). Then Shamir showed that $\mathbf{IP} = \mathbf{PSPACE}$ [Sha90] and Babai, Fortnow and Lund [BFL90] showed $\mathbf{MIP} = \mathbf{NEXP}$. This story —as well as subsequent developments such as the **PCP** Theorem— is described in Babai's entertaining surveys [Bab90, Bab94]. See also the chapter notes to Chapter 11.

The proof of $\mathbf{IP} = \mathbf{PSPACE}$ using the linearization operator is due to Shen [She92]. The question about the power of the prover is related to the complexity of decision versus search, as explored by Bellare and Goldwasser [BG94]; see also Vadhan [Vad00]. Theorem 8.30 has been generalized to languages within **NC** by Goldwasser et al. [GGH+07].

The result that approximating the shortest vector to within a $\sqrt{n/\log n}$ is in $\mathbf{AM}[2]$ and hence probably not **NP**-hard (as mentioned in the introduction) is due to Goldreich and Goldwasser [GG00]. Aharonov and Regev [AR04] proved that approximating this problem to within $\sqrt{n}$ is in $\mathbf{NP} \cap \mathbf{coNP}$.