CHAPTER 8 Interactive proofs

What is intuitively required from a theorem-proving procedure? First, that it is possible to "prove" a true theorem. Second, that it is impossible to "prove" a false theorem. Third, that communicating the proof should be efficient, in the following sense. It does not matter how long must the prover compute during the proving process, but it is essential that the computation required from the verifier is easy.

– Goldwasser, Micali, and Rackoff, 1985

The standard notion of a mathematical proof is closely related to the certificate definition of **NP**. To prove that a statement is true one provides a sequence of symbols on a piece of paper, and the verifier checks that they represent a valid proof/certificate. A valid proof/certificate exists only for true statements. However, people often use a more general way to convince one another of the validity of statements: they *interact* with one another, where the person verifying the proof (called *verifier* from now on) asks the person providing it (called *prover* from now on) for a series of explanations before he is convinced.

It seems natural to try to understand the power of such interactive proofs from the complexity-theoretic perspective. For example, can one prove in a succinct way that a given formula is *not* satisfiable? This problem is **coNP**-complete, and hence is believed to not have a polynomial-sized proof in the traditional sense. The surprising fact is that it does have succinct proofs when the verifier is allowed to interact with the prover (Section 8.3), and in fact so does TQBF and every other problem in **PSPACE**. (We note that these succinct interactive proofs require that the verifier be randomized, and this is crucial; see Section 8.1.) Such facts alone make the study of interactive proofs very important. Furthermore, study of interactive proofs yields new insights into other issues—cryptographic protocols (see Remark 8.8 and Section 9.4); limits on the power of approximation algorithms (Section 8.5); program checking (Section 8.6); and evidence that some famous problems like *graph isomorphism* (see Section 8.1.3) and *approximate shortest lattice vector* (see chapter notes) are *not* **NP**-complete.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

8.1 INTERACTIVE PROOFS: SOME VARIATIONS

As mentioned, interactive proofs introduce *interaction* into the basic **NP** scenario. Instead of the prover sending a written proof to the verifier, the verifier conducts an interrogation of the prover, repeatedly asking questions and listening to the prover's responses. At the end, the verifier decides whether or not to accept the input. Of course, the message of each party at any point in the interaction can depend upon messages sent and received so far. The prover is assumed to be an all-powerful machine (see the notes following Definition 8.3), though, as we will see, it suffices to assume it is a **PSPACE** machine; see the remark after Definition 8.6.

We have several further choices to make in completing the definition: (a) Is the prover deterministic or probabilistic? (b) Is the verifier deterministic or probabilistic? (c) If we allow probabilistic machines, how do we define "accept" and "reject"? We saw in Chapter 7 several choices for this depending upon the type of error allowed (one-sided versus two-sided).

Let us explore the effect of some of these choices.

8.1.1 Warmup: Interactive proofs with deterministic verifier and prover

First, we consider interactive proofs with deterministic verifier and prover.

EXAMPLE 8.1

Let us consider a trivial example of such an interactive proof for membership in 3SAT. Proceeding clause by clause, the verifier asks the prover to announce the values for the literals in the clause. The verifier keeps a record of these answers, and accepts at the end if all clauses were indeed satisfied, and the prover never announced conflicting values for a variable.

Thus both verifier and prover are deterministic.

Of course, in this case we may well ask what the point of interaction is, as the prover could just announce values of all clauses in the very first round, and then take a nap from then on. In fact, we will soon see this is a subcase of a more general phenomenon: Interactive proofs with deterministic verifiers never need to last more than a single round.

First, let us clarify the word "interaction" in Example 8.1. By this we mean that the verifier and prover are two deterministic functions that at each round of interaction compute the next question/response as a function of the input and the questions and responses of the previous rounds.

Definition 8.2 *(Interaction of deterministic functions)* Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be functions and $k \geq 0$ be an integer (allowed to depend upon the input size). A *k-round interaction* of f and g on input $x \in \{0,1\}^*$, denoted by $\langle f, g \rangle(x)$ is the sequence of strings

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

$a_1, \ldots, a_k \in \{0,1\}^*$ defined as follows:

$$\begin{aligned} a_1 &= f(x) \\ a_2 &= g(x, a_1) \\ &\ldots \\ a_{2i+1} &= f(x, a_1, \ldots, a_{2i}) \qquad \text{for } 2i < k \\ a_{2i+2} &= g(x, a_1, \ldots, a_{2i+1}) \qquad \text{for } 2i+1 < k \end{aligned} \tag{8.1}$$

The *output* of f at the end of the interaction denoted $\mathtt{out}_f\langle f, g\rangle(x)$ is defined to be $f(x, a_1, \ldots, a_k)$; we assume this output is in $\{0,1\}$. ◇

Definition 8.3 *(Deterministic proof systems)* We say that a language L has a *k-round deterministic interactive proof system* if there's a deterministic TM V that on input $x, a_1, \ldots, a_i$ runs in time polynomial in $|x|$, and can have a k-round interaction with any function P such that

$$\begin{aligned} &\text{(Completeness)} && x \in L \Rightarrow \exists P : \{0,1\}^* \to \{0,1\}^* \ \mathtt{out}_V\langle V, P\rangle(x) = 1 \\ &\text{(Soundness)} && x \notin L \Rightarrow \forall P : \{0,1\}^* \to \{0,1\}^* \ \mathtt{out}_V\langle V, P\rangle(x) = 0 \end{aligned}$$

The class **dIP** contains all languages with a $k(n)$-round deterministic interactive proof system where $k(n)$ is polynomial in n. ◇

Notice, this definition places no limits on the computational power of the prover P; this makes intuitive sense, since a false assertion should *not* be provable, no matter how clever the prover. Note also that because we place no such limits, it does not matter that we allow the prover in the completeness and soundness conditions to depend on x (see also Exercise 8.2).

As hinted in Example 8.1, **dIP** actually is a class we know well.

Lemma 8.4 $\mathbf{dIP} = \mathbf{NP}$. ◇

Proof: Trivially, every **NP** language has a one-round deterministic proof system and thus is in **dIP**. Now we prove that if $L \in \mathbf{dIP}$ then $L \in \mathbf{NP}$. If V is the verifier for L, then a certificate that an input is in L is just a transcript $(a_1, a_2, \ldots, a_k)$ causing the verifier V to accept. To verify this transcript, one checks that indeed $V(x) = a_1$, $V(x, a_1, a_2) = a_3, \ldots$, and $V(x, a_1, \ldots, a_k) = 1$. If $x \in L$ then such a transcript exists. Conversely, if such a transcript $(a_1, \ldots, a_k)$ exists then we can define a prover function P to satisfy $P(x, a_1) = a_2$, $P(x, a_1, a_2, a_3) = a_4$, and so on. This deterministic prover satisfies $\mathtt{out}_V\langle V, P\rangle(x) = 1$, which implies $x \in L$. ■

8.1.2 The class IP: Probabilistic verifier

The message of Section 8.1.1 is that in order for interaction to provide any benefit, we need to let the verifier be *probabilistic*. This means that the verifier's questions will be

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

computed using a probabilistic algorithm. Furthermore, the verifier will be allowed to come to a wrong conclusion (e.g., accept a proof for a wrong statement) with some small probability. As in the case of probabilistic algorithms, this probability is over the choice of the verifier's coins, and we require the verifier to reject proofs for a wrong statement with good probability *regardless* of the strategy the prover uses. Allowing this combination of interaction and randomization has a huge effect: As we will see in Section 8.3, the set of languages that have such interactive proof systems jumps from **NP** to **PSPACE**.

EXAMPLE 8.5

As an intuitive example for the power of combining randomization and interaction, consider the following scenario: Marla has one red sock and one yellow sock, but her friend Arthur, who is color-blind, does not believe her that the socks have different colors. How can she convince him that this is really the case?

Here is a way to do so. Marla gives both socks to Arthur, tells him which sock is yellow and which one is red, and Arthur holds the red sock in his right hand and the yellow sock in his left hand. Then Marla turns her back to Arthur and he tosses a coin. If the coin comes up "heads" then Arthur keeps the socks as they are; otherwise, he switches them between his left and right hands. He then asks Marla to guess whether he switched the socks or not. Of course Marla can easily do so by seeing whether the red sock is still in Arthur's right hand or not. But if the socks were identical then she would not have been able to guess the answer with probability better than $1/2$. Thus if Marla manages to answer correctly in all of, say, 100 repetitions of this game, then Arthur can indeed be convinced that the socks have different colors.

The principle behind this "interactive proof system" actually underlies the systems for graph nonisomorphism and quadratic nonresiduosity that we will see later in this chapter (Section 8.1.3 and Example 8.9). In the sock example, the verifier, being color-blind, has less power (i.e., fewer capabilities) than the prover. In general interactive proofs, the verifier—being polynomial-time—also has less computational power than the prover.

Now we give a precise definition of an interactive proof with a *probabilistic* verifier. To extend Definition 8.2 to model an interaction between f and g where f is probabilistic, we add an additional m-bit input r to the function f in (8.1), that is, $a_1 = f(x,r)$, $a_3 = f(x,r,a_1,a_2)$, and so on. However, the function g is evaluated only on the a_i's and does not get r as an additional input. (This models the fact that the prover cannot "see" the verifier's coins but only his messages; for this reason, this is called the *private coins* model for interactive proofs, as opposed to the *public coins* model of Section 8.2.) The interaction $\langle f,g\rangle(x)$ is now a random variable over $r \in_R \{0,1\}^m$. Similarly the output $\mathtt{out}_f\langle f,g\rangle(x)$ is also a random variable.

Definition 8.6 *(Probabilistic verifiers and the class* **IP***)* For an integer $k \geq 1$ (that may depend on the input length), we say that a language L is in **IP**$[k]$ if there is a probabilistic polynomial-time Turing machine V that can have a k-round interaction with a function $P:\{0,1\}^* \to \{0,1\}^*$ such that

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

$$\text{(Completeness)} \qquad x \in L \Rightarrow \exists P \Pr[\mathtt{out}_V\langle V, P\rangle(x) = 1] \geq 2/3 \tag{8.2}$$

$$\text{(Soundness)} \qquad x \notin L \Rightarrow \forall P \Pr[\mathtt{out}_V\langle V, P\rangle(x) = 1] \leq 1/3 \tag{8.3}$$

where all probabilities are over the choice of r.
We define $\mathbf{IP} = \cup_{c \geq 1}\mathbf{IP}[n^c]$.

Now we study the robustness of this definition. First we show that the probabilities $2/3$ and $1/3$ in Definition 8.6 can be made arbitrarily close to 1 and 0, respectively, by using the same boosting technique we used for **BPP** (see Section 7.4.1).

Lemma 8.7 *The class* **IP** *defined in Definition 8.6 is unchanged if we replace the completeness parameter* $2/3$ *by* $1 - 2^{-n^s}$ *and the soundness parameter* $1/3$ *by* 2^{-n^s} *for any fixed constant* $s > 0$. ◇

Proof: The verifier repeats the entire protocol over and over again, say m times, and accepts at the end iff more than $1/2$ the runs resulted in an accept. If $x \in L$, then a prover that can make the verifier accept with probability $2/3$ in each repetition will at the end succeed with probability $1 - 2^{-\Omega(m)}$ by the Chernoff bound (Theorem A.14). If $x \notin L$, we have to argue that every prover strategy will fail with high probability. We claim that that the prover can succeed in each repetition of the protocol with probability only $1/3$—irrespective of what happened in earlier rounds. The reason is that even though the prover's responses in this repetition may depend arbitarily on its responses in the earlier repetitions, since the expression in (8.3) holds for all provers, it holds in particular for the prover that knows the questions of earlier rounds.

Thus Chernoff bounds again imply that the probability that the prover succeed in a majority of the repetitions only with probability $2^{-\Omega(m)}$. Choosing $m = O(n^s)$ completes the proof. ■

We now make several assertions about the class **IP**. Exercise 8.1 asks you to prove some of them.

1. Allowing the prover to be probabilistic, that is, allowing the answer function a_i to depend upon some random string used by the prover (and unknown to the verifier), does not change the class **IP**. The reason is that for any language L, if a probabilistic prover P can make a verifier V accept with some probability, then averaging implies that there is a deterministic prover that makes V accept with the same probability.
2. Since the prover can use an arbitrary function, it can in principle use unbounded computational power or even compute undecidable functions. However, we can show that given any verifier V, we can compute the optimum prover (which, given x, maximizes the verifier's acceptance probability) using $\text{poly}(|x|)$ space (and hence also $2^{\text{poly}(|x|)}$ time). Thus $\mathbf{IP} \subseteq \mathbf{PSPACE}$.
3. Replacing the constant $2/3$ with 1 in the completeness requirement (8.2) does not change the class **IP**. This is a nontrivial fact. It was originally proved in a complicated way but today can be proved using our characterization of **IP** in Section 8.3.
4. By contrast, replacing the constant $1/3$ with 0 in the soundness condition (8.3) is equivalent to having a deterministic verifier and hence reduces the class **IP** to **NP**.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

5. *Private Coins:* Thus far the prover functions do not depend upon the verifier's random strings, only on the messages/questions the verifier sends. In other words, the verifier's random string is *private*. Often these are called *private coin* interactive proofs. In Section 8.2 we also consider the model of *public-coin* proofs (also known as *Arthur-Merlin* proofs) where all the verifier's questions are simply obtained by tossing coins and revealing them to the prover.
6. The proof of Lemma 8.7 sequentially repeats the basic protocol m times and takes the majority answer. In fact, using a more complicated proof, it can be shown that we can decrease the probability without increasing the number of rounds using *parallel repetition*, where the prover and verifier will run m executions of the protocol in parallel (i.e., by asking all m questions in one go). The proof of this fact is easier for the case of *public-coin* protocols.

8.1.3 Interactive proof for graph nonisomorphism

We present another example of a language in **IP** that is not known to be in **NP**. The usual ways of representing graphs—adjacency lists, adjacency matrices—involve labeling each vertex with a unique number. We say two graphs G_1 and G_2 are *isomorphic* if they are the same up to a renumbering of vertices; in other words, if there is a permutation π of the labels of the nodes of G_1 such that $\pi(G_1) = G_2$, where $\pi(G_1)$ is the labeled graph obtained by applying π on its vertex labels. The graphs in Figure 8.1, for example, are isomorphic with $\pi = (12)(3654)$. (This is the permutation in which 1 and 2 are mapped to each other, 3 to 6, 6 to 5, 5 to 4, and 4 to 1.) If G_1 and G_2 are isomorphic, we write $G_1 \cong G_2$. The GI problem is the following: given two graphs G_1, G_2 decide if they are isomorphic.

Clearly GI $\in$ **NP**, since a certificate is simply the description of the permutation π. The graph isomorphism problem is important in a variety of fields and has a rich history (see [Hof82]). It is open whether GI is **NP**-complete, and, along with the factoring problem, it is the most famous **NP**-problem that is not known to be either in **P** or **NP**-complete. In Section 8.2.4, we show that that GI is not **NP**-complete unless the polynomial hierarchy collapses. The first step of this proof will be an interactive proof for the complement of GI: the problem GNI of deciding whether two given graphs are *not* isomorphic.

Protocol: *Private-coin* Graph Nonisomorphism

V: Pick $i \in \{1, 2\}$ uniformly randomly. Randomly permute the vertices of G_i to get a new graph H. Send H to P.

P: Identify which of G_1, G_2 was used to produce H. Let G_j be that graph. Send j to V.

V: Accept if $i = j$; reject otherwise.

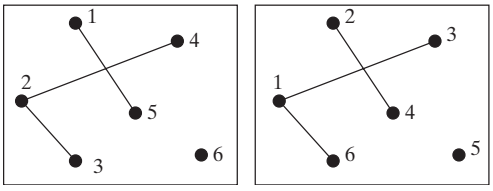


Figure 8.1. Two isomorphic graphs.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

To see that Definition 8.6 is satisfied by the protocol, note that if $G_1 \ncong G_2$ then there exists a prover such that $\Pr[V \text{ accepts}] = 1$ because if the graphs are nonisomorphic, an all-powerful prover can certainly tell which one of the two is isomorphic to H. On the other hand, if $G_1 \cong G_2$, the best any prover can do is to randomly guess because a random permutation of G_1 looks exactly like a random permutation of G_2. Thus in this case for every prover, $\Pr[V \text{accepts}] \leq 1/2$. This probability can be reduced to $1/3$ by sequential or parallel repetition.

Remark 8.8 (Zero-knowledge proofs)
Now we briefly touch upon *zero-knowledge proofs*, a topic related to interactive proofs that underlies a huge research effort in cryptography. Roughly speaking, a *zero-knowledge proof* system for membership in a language is an interactive proof protocol where the verifier is convinced at the end that the input x is in the language, but learns *nothing else*. How can we quantify that the verifier learns nothing else? We do this by showing that the verifier could have produced the transcript of the protocol in polynomial time with no help from the prover. We will see in Section 9.4 that the above protocol for graph nonisomorphism is zero-knowledge.

One can see why such a concept might be useful in cryptography. It raises the possibility of parties being able to prove things to each other without revealing any secrets (e.g., to prove that you hold the password without revealing the password itself). This was one of the original motivations for the invention of the notion of interactive proofs. Section 9.4 of contains a formal definition and some examples of zero-knowledge protocols. (That section does not depend on the other material of Chapter 9 and hence can be read in isolation from that chapter.)

EXAMPLE 8.9 *(Quadratic nonresiduosity)*

Here is another example for an interactive proof for a language not known to be in **NP**. We say that a number a is a *quadratic residue* mod$p$ if there is another number b such that $a \equiv b^2 (\text{mod } p)$. Such a b is called the *square root* of a (mod p). Clearly, $-b$ is another square root, and there are no other square roots since the equation $x^2 - a$ has at most two solutions over GF(p).

The language of pairs (a, p) where p is a prime and a is a quadratic residue mod p is in **NP**, since a square root constitutes a membership proof. Of course, the fact that p is a prime also has a short membership proof, and indeed primality can be tested in polynomial time; see Chapter 2.

In contrast, the language QNR of pairs (a, p) such that p is a prime and a is *not* a quadratic residue modulo p has no natural short membership proof and is not known to be in **NP**. But it does have a simple interactive proof if the verifier is probabilistic.

The verifier takes a random number r mod p and a random bit $b \in \{0, 1\}$ (kept secret from the prover). If $b = 0$ she sends the prover r^2 mod p and if $b = 1$ she sends ar^2 mod p. She asks the prover to guess what b was and accepts iff the prover guesses correctly.

If a is a quadratic residue, then the distribution of ar^2 and r^2 are identical; both are random elements of the group of quadratic residues modulo p. (To see this, note that every quadratic residue a' can be written as as^2 where s is a square root of a/a'.) Thus the prover has probability at most $1/2$ of guessing b.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

On the other hand, if a is a nonresidue, then the distributions ar^2 and r^2 are completely distinct: The first is a random nonresidue modulo p, and the second is a random quadratic residue modulo p. An all-powerful prover can tell them apart, and thus guess b with probability 1. Thus it can make the verifier accept with probability 1.

8.2 PUBLIC COINS AND **AM**

Our proof system for graph nonisormorphism and nonresiduosity seemed to crucially rely on the verifier's access to a source of *private random coins* that are not seen by the prover. Allowing the prover full access to the verifier's random string leads to the model of *interactive proofs with public coins*.

Definition 8.10 *(**AM**, **MA**)* For every k the complexity class **AM**[k] is defined as the subset of **IP**[k] (see Definition 8.6) obtained when we restrict the verifier's messages to be random bits, and not allowing it to use any other random bits that are not contained in these messages.

An interactive proof where the verifier has this form is called a *public coin* proof, sometimes also known as an *Arthur-Merlin* proof.[1] ◇

We denote by **AM** the class **AM**[2].[2] That is, **AM** is the class of languages with an interactive proof that consist of the verifier sending a random string, and the prover responding with a message, where the verifier's decision is obtained by applying a deterministic polynomial-time function to the transcript. The class **MA** denotes the class of languages with a two-round public-coin interactive proof with the prover sending the first message. That is, $L \in$ **MA** if there's a proof system for L that consists of the prover first sending a message, and then the verifier tossing coins and computing its decision by doing a deterministic polynomial-time computation involving the input, the prover's message and the coins.

Remark 8.11
We mention some properties of the class **AM**[k]:

1. Note that even in a public-coins proof, the prover doesn't get to see immediately all of the verifier's random coins, but rather they are revealed to the prover iteratively message by message. That is, an **AM**[k]-proof is an **IP**[k]-proof where the verifier's random tape r consists of $\lceil k/2 \rceil$ strings $r_1, \ldots, r_{\lceil k/2 \rceil}$, his ith message is simply the string

[1] According to an old legend, Arthur was a great king of medieval England and Merlin was his court magician. Babai [Bab85] used the name "Arthur-Merlin" for this model by drawing an analogy between the prover's infinite power and Merlin's magic. While Merlin cannot predict the coins that Arthur will toss in the future, Arthur has no way of hiding from Merlin's magic the results of the coins he tossed in the past.

[2] Note that **AM** = **AM**[2] while **IP** = **IP**[poly]. Although this is indeed somewhat inconsistent, it is the standard notation used in the literature. Some sources denote the class **AM**[3] by **AMA**, the class **AM**[4] by **AMAM**, and so on.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

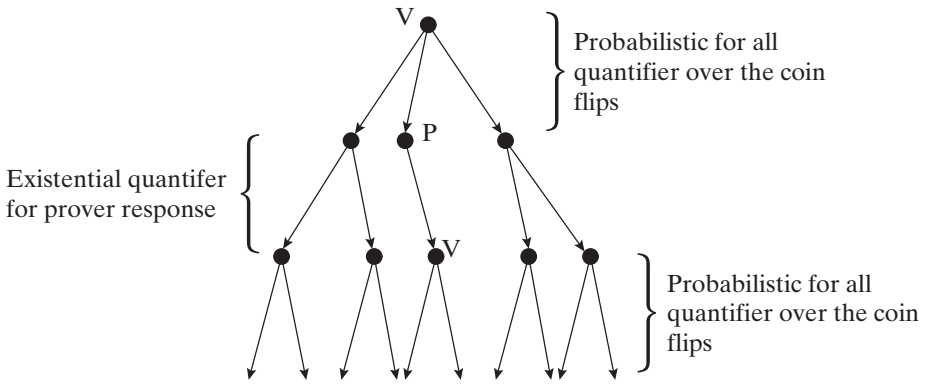


Figure 8.2. **AM**$[k]$ looks like $\prod_k^p$, with the $\forall$ quantifier replaced by probabilistic choice.

r_i, and the decision whether to accept or reject is obtained by applying a deterministic polynomial-time computable function to the transcript.

2. $\mathbf{AM}[2] = \mathbf{BP} \cdot \mathbf{NP}$ where $\mathbf{BP} \cdot \mathbf{NP}$ is the class in Definition 7.17. In particular it follows that $\mathbf{AM}[2] \subseteq \mathbf{\Sigma}_3^p$. (See Exercise 8.3.)
3. For constants $k \geq 2$ we have $\mathbf{AM}[k] = \mathbf{AM}[2]$ (Exercise 8.7). This "collapse" is somewhat surprising because $\mathbf{AM}[k]$ at first glance seems similar to **PH** with the $\forall$ quantifiers changed to "probabilistic $\forall$" quantifiers, where *most* of the branches lead to acceptance. See Figure 8.2.
4. It is an open problem whether there is any nice characterization of $\mathbf{AM}[\sigma(n)]$, where $\sigma(n)$ is a suitably slowly growing function of n, such as $\log \log n$.

8.2.1 Simulating private coins

Clearly for every k, $\mathbf{AM}[k] \subseteq \mathbf{IP}[k]$. The interactive proof for GNI seemed to crucially depend upon the fact that P cannot see the random bits of V. If P knew those bits, P would know i and so could trivially always guess correctly. Thus it may seem that allowing the verifier to keep its coins private adds significant power to interactive proofs, and so the following result should be quite surprising:

Theorem 8.12 *(Goldwasser-Sipser* [GS87]*)*
For every $k : \mathbb{N} \to \mathbb{N}$ with $k(n)$ computable in poly(n)*,*

$$\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$$

We sketch the proof of Theorem 8.12 in Section 8.12 after proving the next Theorem, which concerns the subcase of GNI.

Theorem 8.13 GNI $\in \mathbf{AM}[2]$. ◇

The proof of Theorem 8.13 is a good example of how nontrivial interactive proofs can be designed by recasting the problem. The key idea is to look at graph nonisomorphism in a different, more quantitative, way. Consider the following set of labeled

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

graphs $S = \{H : H \cong G_1 \text{ or } H \cong G_2\}$. Note that it is easy to certify that a graph H is a member of S, by providing the permutation mapping either G_1 or G_2 to H. An n vertex graph G has at most $n!$ equivalent graphs. For simplicity assume first that both G_1 and G_2 have each exactly $n!$ equivalent graphs. The size of S differs by a factor 2 depending upon whether or not G_1 is isomorphic to G_2.

$$\text{if } G_1 \ncong G_2 \text{ then } |S| = 2n! \tag{8.4}$$

$$\text{if } G_1 \cong G_2 \text{ then } |S| = n! \tag{8.5}$$

Now consider the general case where G_1 or G_2 may have less than $n!$ equivalent graphs. An n-vertex graph G has less than $n!$ equivalent graphs iff it has a nontrivial *automorphism*, which is a permutation π that is not the identity permutation and yet $\pi(G) = G$. Let $aut(G)$ denote the set of automorphisms of graph G. We change the definition of S to

$$S = \{(H, \pi) : H \cong G_1 \text{ or } H \cong G_2 \text{ and } \pi \in aut(H)\}$$

Using the fact that $aut(G)$ is a subgroup, one can verify that S satisfies (8.4) and (8.5). Also, membership in this set is easy to certify.

Thus to convince the verifier that $G_1 \ncong G_2$, the prover has to convince the verifier that case (8.4) holds rather than (8.5). This is done by using a *set lower bound protocol*.

8.2.2 Set lower bound protocol

Suppose there is a set S known to both prover and verifier, such that membership in S is easily certifiable, in the sense that given some string x that happens to be in S, the prover—using its superior computational power—can provide the verifier a certificate to this effect. (To put it more formally, S is in $\mathbf{BP} \cdot \mathbf{NP}$.) The *set lower bound protocol* is a public-coins protocol that allows the prover to *certify* the approximate size of S. Note that the prover—using its superior computational power—can certainly compute and announce $|S|$. The question is how to convince the verifier that this answer is correct, or even approximately correct. Suppose the prover's claimed value for $|S|$ is K. The protocol below has the property that if the true value of $|S|$ is indeed at least K, then the prover can cause the verifier to accept with high probability, whereas if the true value of $|S|$ is at most $K/2$ (the prover's answer is grossly on the high side), then the verifier will reject with high probability, no matter what the prover does. This protocol is called the *set lower bound protocol* and it clearly suffices to complete the proof of Theorem 8.13.

Tool: Pairwise independent hash functions

The main tool in the set lower bound protocol is a *pairwise independent hash function collection*, which has also found numerous other applications in complexity theory and computer science (see Note 8.16).

Definition 8.14 *(Pairwise independent hash functions)* Let $\mathcal{H}_{n,k}$ be a collection of functions from $\{0,1\}^n$ to $\{0,1\}^k$. We say that $\mathcal{H}_{n,k}$ is *pairwise independent* if for every $x, x' \in \{0,1\}^n$ with $x \neq x'$ and for every $y, y' \in \{0,1\}^k$, $\Pr_{h \in_R \mathcal{H}_{n,k}}[h(x) = y \wedge h(x') = y'] = 2^{-2k}$. ◇

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

An equivalent formulation is that for every two distinct but fixed strings $x, x' \in \{0,1\}^n$, when we choose h at random from $\mathcal{H}_{n,k}$, then the random variable $\langle h(x), h(x')\rangle$ is distributed according to the uniform distribution on $\{0,1\}^k \times \{0,1\}^k$.

We can identify the elements of $\{0,1\}^n$ with the *finite field* $\mathrm{GF}(2^n)$ containing 2^n elements (see Section A.4). Recall that the addition ($+$) and multiplication ($\cdot$) operations in this field are efficiently computable and satisfy the usual commutative and distributive laws, every element x has an additive inverse (denoted by $-x$) and, if nonzero, a multiplicative inverse (denoted by x^{-1}). The following theorem provides a construction of a family of *efficiently computable* pairwise independent hash functions (see also Exercise 8.4 for a different construction).

Theorem 8.15 *(Efficient pairwise independent hash functions) For every n, define the collection $\mathcal{H}_{n,n}$ to be $\{h_{a,b}\}_{a,b\in \mathrm{GF}(2^n)}$ where for every $a, b \in \mathrm{GF}(2^n)$, the function $h_{a,b} : \mathrm{GF}(2^n) \to \mathrm{GF}(2^n)$ maps x to $ax + b$. Then, $\mathcal{H}_{n,n}$ is a collection of pairwise independent hash functions.* ◇

Theorem 8.15 implies the existence of an efficiently computable family of pairwise independent hash functions $\mathcal{H}_{n,k}$ for every n, k: if $k > n$ we can use the collection $\mathcal{H}_{k,k}$ and extend n bit inputs to k bits by padding with zeros. If $k < n$, then we can use the collection $\mathcal{H}_{n,n}$ and reduce n bit outputs to k bits by truncating the last $n - k$ bits.

Proof: For every $x \neq x' \in \mathrm{GF}(2^n)$ and $y, y' \in \mathrm{GF}(2^n)$, $h_{a,b}(x) = y$ and $h_{a,b}(x') = y'$ iff a, b satisfy the equations:

$$a \cdot x + b = y$$
$$a \cdot x' + b = y'$$

These equations imply that $a = (y - y')(x - x')^{-1}$; this is well-defined because $x - x' \neq 0$. Since $b = y - a \cdot x$, the pair $\langle a, b\rangle$ is completely determined by these equations, and so the probability that this happens over the choice of a, b is exactly one over the number of possible pairs, which indeed equals $\frac{1}{2^{2n}}$. ■

The lower-bound protocol

The lower-bound protocol is as follows.

Protocol: Goldwasser-Sipser Set Lower Bound Protocol

Conditions: $S \subseteq \{0,1\}^m$ is a set such that membership in S can be certified. Both parties know a number K. The prover's goal is to convince the verifier that $|S| \geq K$ and the verifier should reject with good probability if $|S| \leq \frac{K}{2}$. Let k be an integer such that $2^{k-2} < K \leq 2^{k-1}$.

V: Randomly pick a function $h : \{0,1\}^m \to \{0,1\}^k$ from a pairwise independent hash function collection $\mathcal{H}_{m,k}$. Pick $y \in_R \{0,1\}^k$. Send h, y to prover.

P: Try to find an $x \in S$ such that $h(x) = y$. Send such an x to V, together with a certificate that $x \in S$.

V's output: If $h(x) = y$ and the certificate validates that $x \in S$ then accept; otherwise reject.

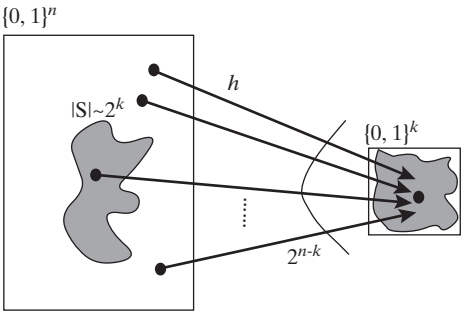
https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

NOTE 8.16 *(The Hashing paradigm* [CW77]*)*

In many computer programs, hash functions are used to create a *hash table*. The goal is to store a set $S \subseteq \{0,1\}^n$ so as to be able to efficiently answer *membership queries*, which ask whether or not a given element x is in S. The set S could change dynamically (i.e., elements may be added or deleted), but its size is guaranteed to be much smaller than 2^n, the number of all possible elements.

To create a hash table of size 2^k, we pick a hash function h mapping $\{0,1\}^n$ to $\{0,1\}^k$ and store $x \in S$ at location $h(x)$. If we ever need to later determine whether or not x is in S, we just compute $h(x)$ and go look for x at this location in the hash table. Notice, if $h(x) = h(y)$, then both x, y are stored in the same location; this is called a *collision*. Such collisions can be dealt with, but at a cost to efficiency, and hence we want to minimize them by choosing a sufficiently "random" hash function.

Instead of using a fixed hash function, it makes sense to use a random function from a *hash function collection*, such as the collection in Theorem 8.15. This will guarantee that that most elements of $\{0,1\}^k$ have roughly $|S|2^{-k}$ preimages in S (which is the expected number if h were a completely random function). In particular, if S has size roughly 2^k, then we expect the mapping to be one-to-one or almost one-to-one, and so the expected number of collisions is small. Therefore, the image of S under h should look like this:



Hash tables are a preferred solution in many cases because of the ease with which they handle sets that change dynamically. The foregoing analysis of the expectation continues to apply so long as the size of S remains less than 2^k and the hash function is picked from a collection using random bits that are *independent* of the set S.

In theoretical computer science, hash functions have found a variety of uses. One example is Lemma 17.19, which shows that if the collection is pairwise independent and $S \subseteq \{0,1\}^n$ has size roughly 2^k, then with good probability the value 0^k will have exactly one preimage in S. Another example is the *Leftover Hash Lemma* (Lemma 21.26) that shows that if S is larger than 2^k then a random element of S is mapped by h almost perfectly to a random element of $\{0,1\}^k$.

Pairwise independent hash functions are but one example of a hash function collection. Once can study other collections featuring various tradeoffs between efficiency and uniformity of output, including almost pairwise independence, k-wise independence, ϵ-biased, and more. See the survey by Luby and Wigderson [LW06].

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

Clearly, the prover (being all powerful) can make the verifier accept iff h, y happen to be such that an $x \in S$ exists satisfying $h(x) = y$. The following claim shows that, for $p^* = K/2^k$, there is a gap of $\frac{3}{4}p^*$ versus $p^*/2$ in the probability of this happening in the two cases we are interested in ($|S| \geq K$ versus $|S| < K/2$). While the claim is stated only for $|S| \leq K$, note that clearly a larger set only increases the probability that the prover can make the verifier accept.

Claim 8.16.1 *Let $S \subseteq \{0,1\}^m$ satisfy $|S| \leq \frac{2^k}{2}$. Then, for $p = |S|/2^k$*

$$p \geq \Pr_{h \in_R \mathcal{H}_{m,k}, y \in_R \{0,1\}^k}[\exists_{x \in S} : h(x) = y] \geq \frac{3p}{4} - \frac{p}{2^k}. \qquad \diamond$$

Proof: The upper bound on the probability follows trivially by noticing that the set $h(S)$ of y's with preimages in S has size that is at most $|S|$. We now prove the lower bound. In fact, we show the stronger statement that

$$\Pr_{h \in_R \mathcal{H}_{m,k}}[\exists_{x \in S} h(x) = y] \geq \tfrac{3}{4}p$$

for *every* $y \in \{0,1\}^k$. Indeed, for every $x \in S$ define E_x as the event that $h(x) = y$. Then, $\Pr[\exists\ x \in S\ : h(x) = y] = \Pr[\vee_{x \in S} E_x]$. By the inclusion-exclusion principle (Corollary A.2), this is at least

$$\sum_{x \in S} \Pr[E_x] - \tfrac{1}{2} \sum_{x \neq x' \in S} \Pr[E_x \cap E_{x'}]$$

However, by pairwise independence of the hash functions, if $x \neq x'$, then $\Pr[E_x] = 2^{-k}$ and $\Pr[E_x \cap E_{x'}] = 2^{-2k}$ and so (up to the low order term $|S|/2^{2k}$) this probability is at least

$$\frac{|S|}{2^k} - \frac{1}{2}\frac{|S|^2}{2^{2k}} = \frac{|S|}{2^k}\left(1 - \frac{|S|}{2^{k+1}}\right) \geq \frac{3}{4}p \ \blacksquare$$

Proving Theorem 8.13

The public-coin interactive proof system for GNI consists of the verifier and prover running several iterations of the set lower bound protocol for the set S as defined above, where the verifier accepts iff the fraction of accepting iterations is at least $5p^*/8$ (note that $p^* = K/2^k$ can be easily computed by the verifier). Using the Chernoff bound (Theorem A.14), it can be easily seen that a constant number of iterations will suffice to ensure completeness probability at least $2/3$ and soundness error at most $1/3$.

Finally, the number of rounds stays at 2 because the verifier can do all iterations in *parallel*: Pick several choices of h, y and send them all to the prover at once. It is easily checked that the above analysis of the probability of the prover's success is unaffected even if the prover is asked many questions in parallel. ■

Remark 8.17

Note that, unlike the private-coins protocol for GNI, the public-coins protocol of Theorem 8.13 does not have perfect completeness (i.e., the completeness parameter is not 1), since the set lower bound protocol does not satisfy this property. However, we can construct a public-coins set lower bound protocol with completeness parameter 1

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

(see Exercise 8.5), thus implying a perfectly complete public-coins proof for GNI. This can be generalized to show that every private-coins proof system (even one not satisfying perfect completeness) can be transformed into a perfectly complete public-coins system with a similar number of rounds.

8.2.3 Sketch of proof of Theorem 8.12

Our transformation of the private-coins protocol for GNI into a public-coins protocol suggests how to do such a transformation for every other private-coins protocol. The idea is that the public-coin prover demonstrates to the public-coin verifier an approximate lower bound on the size of the set of random strings which would have made the private-coin verifier accept in the original protocol.

Think how our public-coins protocol for GNI relates to the private-coin protocol of Section 8.1.3. The set S roughly corresponds to the set of possible messages sent by the verifier in the protocol, where the verifier's message is a random element in S. If the two graphs are isomorphic, then the verifier's message completely hides its choice of a random $i \in_R \{1,2\}$, whereas if they're not, then the message distribution completely reveals it (at least to a prover that has unbounded computation time). Thus roughly speaking in the former case the mapping from the verifier's coins to the message is 2-to-1, whereas in the latter case it is 1-to-1, resulting in a set that is twice as large. In fact we can think of the public-coin prover as convincing the verifier that the private-coin verifier would have accepted with large probability. The idea behind the proof of $\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$ is similar, but one has to proceed in a round-by-round fashion, and the prover has to prove to the verifier that certain messages are quite likely to be sent by the verifier—in other words, the set of random strings that make the verifier send these messages in the private-coin protocol is quite large.

8.2.4 Can GI be NP-complete?

As mentioned earlier, it is an open problem if GI is **NP**-complete. We now prove that if GI is **NP**-complete, then the polynomial hierarchy collapses.

Theorem 8.18 *(*[BHZ87]*) If* GI *is* **NP***-complete, then* $\mathbf{\Sigma_2} = \mathbf{\Pi_2}$. ◇

Proof: We show that under this condition, $\mathbf{\Sigma_2} \subseteq \mathbf{\Pi_2}$; this will imply $\mathbf{\Sigma_2} = \mathbf{\Pi_2}$ because $\mathbf{\Sigma_2} = \mathbf{co\Pi_2}$.

If GI is **NP**-complete, then GNI is **coNP**-complete, which implies that there exists a function f such that for every n variable formula φ, $\forall_y \varphi(y)$ holds iff $f(\varphi) \in$ GNI. Consider an arbitrary Σ_2 SAT formula

$$\psi = \exists_{x \in \{0,1\}^n} \forall_{y \in \{0,1\}^n} \varphi(x,y)$$

The formula ψ is equivalent to

$$\exists_{x \in \{0,1\}^n} g(x) \in \text{GNI}$$

where $g(x) = f(\varphi_{\restriction x})$, and $\varphi_{\restriction x}$ is the formula obtained from $\varphi(x,y)$ by fixing x.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

Using Remark 8.17 and the comments of Section 8.11, GNI has a two-round **AM** proof with perfect completeness[3] and (after appropriate amplification) soundness error less than 2^{-n}. Let V be the verifier algorithm for this proof system, and denote by m the length of the verifier's random tape and by m' the length of the prover's message. We claim that ψ is true if and only if

$$\forall_{r\in\{0,1\}^m}\exists_{x\in\{0,1\}^n}\exists_{a\in\{0,1\}^{m'}}(V(g(x),r,a)=1) \tag{8.6}$$

Indeed, if ψ is true, then perfect completeness clearly implies (8.6). If on the other hand ψ is false, this means that

$$\forall_{x\in\{0,1\}^n} g(x) \notin \mathrm{GNI}$$

Now, using the fact that the soundness error of the interactive proof is less than 2^{-n} and the number of x's is 2^n, we conclude (by the "probabilistic method basic principle") that there exists a *single* string $r \in \{0,1\}^m$ such that for every $x \in \{0,1\}^n$, the prover in the **AM** proof for GNI has no response a that will cause the verifier to accept $g(x)$ if the verifier's first message is r. In other words,

$$\exists_{r\in\{0,1\}^m}\forall_{x\in\{0,1\}^n}\forall_{a\in\{0,1\}^{m'}}(V(g(x),r,a)=0)$$

which is exactly the negation of (8.6). Since deciding the truth of (8.6) is in $\mathbf{\Pi_2}$ (as it is a statement of the form $\forall x\exists y P(x,y)$ for some polynomial-time computable predicate P), we have shown $\mathbf{\Sigma_2} \subseteq \mathbf{\Pi_2}$. ■

8.3 IP = PSPACE

It was an open question for a while to characterize **IP**, the set of languages that have interactive proofs. All we knew was that **NP** $\subseteq$ **IP** $\subseteq$ **PSPACE**, and there was evidence (e.g., the protocols for quadratic nonresiduosity and GNI) that the first containment is proper. Most researchers felt that the second containment would also be proper. They reasoned as follows. We know that interaction alone does not give us any languages outside **NP** (Section 8.1.1). We also suspect (see Chapter 7) that randomization alone does not add significant power to computation—researchers even suspect that **BPP** = **P**, based upon evidence described in Chapter 20. So how much more power could the *combination* of randomization and interaction provide? "Not much," the evidence up to 1990 seemed to suggest. For any fixed k, **IP**[k] collapses to the class **AM** = **AM**[2], which equals **BP** · **NP** as mentioned in Remark 8.11, and **BP** · **NP** seems not "much different" from **NP**.[4] Finally, there were simply no protocols known that required k to be superconstant, so **IP** = **IP**[poly(n)] did not seem much bigger than **IP**[$O(1)$]. The following result from 1990, giving a surprising characterization of **IP**, shows that this intuition was drastically wrong.

[3] It is possible to do the rest of this proof without relying on perfect completeness; we leave the details to the interested reader.

[4] In fact, under plausible complexity conjectures, **AM** = **NP**, see Exercise 20.7.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

Theorem 8.19 ([LFKN90, Sha90])
IP = PSPACE.

By our earlier remarks, we only need to show the nontrivial direction **PSPACE** $\subseteq$ **IP**, and for this it suffices to show TQBF $\in$ **IP**[$poly(n)$] because every $L \in$ **PSPACE** is polytime reducible to TQBF. We describe a protocol for TQBF that uses public coins and also has the property that if the input is in TQBF, then there is a prover that makes the verifier accept with probability 1.

Rather than tackle the job of designing a protocol for TQBF right away, let us first think about how to design one for $\overline{\text{3SAT}}$. How can the prover convince the verifier than a given 3CNF formula has no satisfying assignment? We show how to prove something even more general: The prover can prove to the verifier that the *number* of satisfying assignments is exactly K for some number K. That is, we give an interactive proof for membership in the following language.

Definition 8.20 (#SAT$_D$)

$$\#\text{SAT}_{\text{D}} = \{\langle \phi, K\rangle : \phi \text{ is a } 3CNF \text{ formula and it has exactly } K \text{ satisfying assignments}\}$$

◇

This clearly contains $\overline{SAT}$ as a special case (when $K = 0$). In Chapter 17 we will see that #SAT$_{\text{D}}$ is a complete problem for a powerful class called **#P**.

Note that the *set lower bound* protocol of Section 8.2.2 can tackle an approximation version of this problem, namely, prove the value of K within a factor 2 (or any other constant factor). The protocol takes only two rounds. By contrast, our protocol for #SAT$_{\text{D}}$ will use n rounds. The idea of *arithmetization* introduced in this protocol will also prove useful in our protocol for TQBF.

8.3.1 Arithmetization

The key idea will be to take an algebraic view of Boolean formulas by representing them as polynomials. Note that 0, 1 can be thought of both as truth values and as elements of some finite field $\mathbb{F}$. Thus $x_i \wedge x_j$ is true iff $x_i \cdot x_j = 1$ in the field, and $\neg x_i$ is true iff $1 - x_i = 1$.

Arithmetization refers to the following trick. Given any 3CNF formula $\varphi(x_1, x_2, \ldots, x_n)$ with m clauses and n variables, we introduce field variables $X_1, X_2, \ldots, X_n$. For any clause of size 3, we can write an equivalent degree 3 polynomial, as in the following example:

$$x_i \vee \overline{x_j} \vee x_k \longleftrightarrow X_i(1 - X_j)X_k$$

Let us denote the polynomial for the jth clause by $p_j(X_1, X_2, \ldots, X_n)$, where the notation allows the polynomial to depend on all n variables even though, as is clear in the previous example, each p_j only depends upon at most three variables. For every 0, 1 assignment to $X_1, X_2, \ldots, X_n$, we have $p_j(X_1, X_2, \ldots, X_n) = 1$ if the assignment satisfies the clause and $p_j(X_1, X_2, \ldots, X_n) = 0$ otherwise.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

Multiplying these polynomials, we obtain a multivariate polynomial $P_\varphi(X_1, X_2, \ldots, X_n) = \prod_{j \le m} p_j(X_1, \ldots, X_n)$ that evaluates to 1 on satisfying assignments and to 0 for unsatisfying assignments. This polynomial has degree at most $3m$. We represent such a polynomial as a product of all the above degree 3 polynomials without opening up the parenthesis, and so $P_\varphi(X_1, X_2, \ldots, X_n)$ has a representation of size $O(m)$. This conversion of φ to P_φ is called *arithmetization*. Once we have written such a polynomial, nothing stops us from substituting arbitrary values from the field $\mathbb{F}$ instead of just $0, 1$ and evaluating the polynomial. As we will see, this gives the verifier unexpected power over the prover.

8.3.2 Interactive protocol for #SAT$_D$

Now we prove the following result.

Theorem 8.21 #SAT$_D \in$ **IP**. ◇

Proof: Given input $\langle \phi, K \rangle$, where ϕ is a 3CNF formula of n variables and m clauses, we construct P_ϕ by arithmetization, as in Section 8.3.1. The number of satisfying assignments $\#\phi$ of ϕ satisfies

$$\#\phi = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{8.7}$$

The prover's claim is that this sum is exactly K, and from now on, we can forget about the formula and concentrate only on this claim about the polynomial P_ϕ.

To start, the prover sends to the verifier a prime p in the interval $(2^n, 2^{2n}]$. The verifier can check that p is prime using a probabilistic or deterministic primality testing algorithm. All computations described here are done in the field $\mathbb{F} = \mathbb{F}_p$ of integers modulo p. Note that since the sum in (8.7) is between 0 and 2^n, this equation is true over the integers iff it is true modulo p. Thus, from now on we consider (8.7) as an equation in the field $\mathbb{F}_p$. We'll prove the theorem by showing a general protocol, *Sumcheck*, for verifying equations such as (8.7).

Sumcheck protocol

Given a degree d polynomial $g(X_1, \ldots, X_n)$, an integer K, and a prime p, we show how the prover can provide an interactive proof for the claim

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, \ldots, X_n) \tag{8.8}$$

where all computations are modulo p. To execute the protocol the only property of g that the verifier needs is that it has a poly(n) size representation and thus for any assignment of values for the variables from the field GF(p), say $X_1 = b_1, X_2 = b_2, \ldots, X_n = b_n$, the verifier can evaluate $g(b_1, b_2, \ldots, b_n)$ in polynomial time. As noted earlier, this property is satisfied by $g = P_\phi$.

For each sequence of values $b_2, b_3, \ldots, b_n$ to $X_2, X_3, \ldots, X_n$, note that $g(X_1, b_2, b_3, \ldots, b_n)$ is a univariate degree d polynomial in the variable X_1. Thus the

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

following is also a univariate degree d polynomial:

$$h(X_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, b_2, \ldots, b_n) \tag{8.9}$$

If Claim (8.8) is true, then we must have $h(0) + h(1) = K$.

Consider the following protocol.

Protocol: Sumcheck Protocol to Check Claim (8.8)

V: If $n = 1$, check that $g(1) + g(0) = K$. If so accept; otherwise reject. If $n \geq 2$, ask P to send $h(X_1)$ as defined in (8.9).

P: Sends some polynomial $s(X_1)$ (if the prover is not "cheating," then we'll have $s(X_1) = h(X_1)$).

V: Reject if $s(0)+s(1) \neq K$; otherwise pick a random number a in GF(p). Recursively use the same protocol to check that

$$s(a) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots j, b_n)$$

Claim *If (8.8) is false, then V rejects with probability at least* $(1 - \frac{d}{p})^n$.

The claim implies the theorem since if (8.8) is true, then the prover can make the V accept with probability 1, and with our choice of p, the $(1 - \frac{d}{p})^n$ is roughly $1 - dn/p$ and is very close to 1.

Proof of claim: Assume that (8.8) is false. We prove the claim by induction on n. For $n = 1$, V simply evaluates $g(0), g(1)$ and rejects with probability 1 if their sum is not K. Assume the hypothesis is true for degree d polynomials in $n - 1$ variables.

In the first round, the prover P is supposed to return the polynomial h. If it indeed returns h, then since $h(0) + h(1) \neq K$ by assumption, V will immediately reject (i.e., with probability 1). So assume that the prover returns some $s(X_1)$ different from $h(X_1)$. Since the degree d nonzero polynomial $s(X_1) - h(X_1)$ has at most d roots, there are at most d values a such that $s(a) = h(a)$. Thus when V picks a random a,

$$\Pr_a[s(a) \neq h(a)] \geq 1 - \frac{d}{p} \tag{8.10}$$

If $s(a) \neq h(a)$, then the prover is left with an incorrect claim to prove in the recursive step. By the induction hypothesis, the prover fails to prove this false claim with probability at least $\geq \left(1 - \frac{d}{p}\right)^{n-1}$ Thus we have

$$\Pr[V \text{ rejects}] \geq \left(1 - \frac{d}{p}\right) \cdot \left(1 - \frac{d}{p}\right)^{n-1} = \left(1 - \frac{d}{p}\right)^n \tag{8.11}$$

This completes the proof of the claim and hence of Theorem 8.21. ■

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

8.3.3 Protocol for TQBF: proof of Theorem 8.19

We use a very similar idea to obtain a protocol for TQBF. Given a quantified Boolean formula $\Psi = \forall x_1 \exists x_2 \forall x_3 \cdots \exists x_n \phi(x_1, \ldots, x_n)$, we use arithmetization to construct the polynomial P_ϕ. Thus $\Psi \in$ TQBF if and only if

$$\prod_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \prod_{b_3 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \neq 0 \tag{8.12}$$

A first thought is that we could use the same protocol as in the $\#\mathsf{SAT}_\mathrm{D}$ case, except since the first variable x_1 is quantified with $\forall$, we check that $s(0) \cdot s(1) = K$, and do something analogous for all other variables that are quantified with $\forall$. There is nothing basically wrong with this apart from the running time. Multiplying polynomials, unlike addition, increases the degree. If we define $h(X_1)$ analogously as in (8.9) by making X_1 a free variable in (8.12), then its degree may be as high as 2^n. This polynomial may have 2^n coefficients and so cannot be transmitted to a polynomial-time verifier.

The solution is to observe that the claimed statement (8.12) only uses $\{0, 1\}$ values and for $x \in \{0, 1\}$, $x^k = x$ for all $k \geq 1$. Thus in principle we can convert any polynomial $p(X_1, \ldots, X_n)$ into a *multilinear* polynomial $q(X_1, \ldots, X_n)$ (i.e., the degree of $q(\cdot)$ in any variable X_i is at most one) that agrees with $p(\cdot)$ on all $X_1, \ldots, X_n \in \{0, 1\}$. Specifically, we define a *linearization* operator on polynomials where for any polynomial $p(\cdot)$ let $L_{X_i}(p)$ (or $L_i(p)$ for short) be the polynomial defined as follows:

$$\begin{aligned} L_{X_i}(p)(X_1, \ldots, X_n) = {} & X_i \cdot p(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n) \\ & + (1 - X_i) \cdot p(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n) \end{aligned} \tag{8.13}$$

Thus $L_i(p)$ is linear in X_i and agrees with $p(\cdot)$ whenever $X_i \in \{0, 1\}$. So $L_1(L_2(\cdots(L_n(p)\cdots)$ is a multilinear polynomial agreeing with $p(\cdot)$ on all values in $\{0, 1\}$.

We will also think of $\forall x_i$ and $\exists x_i$ as operators on polynomials where

$$\begin{aligned} \forall_{X_i} p(X_1, X_2, \ldots, X_n) = {} & p(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n) \\ & \cdot p(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n) \end{aligned} \tag{8.14}$$

$$\begin{aligned} \exists_{X_i} p(X_1, X_2, \ldots, X_n) = {} & p(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n) \\ & + p(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n) \end{aligned} \tag{8.15}$$

Thus the claim (8.12) may be rephrased as follows: If we apply the sequence of operators $\forall_{X_1} \exists_{X_2} \forall_{X_3} \cdots \exists_{X_n}$ (where $\exists_{X_n}$ is applied first and $\forall X_1$ is applied last) on the polynomial $P_\phi(X_1, \ldots, X_n)$, then we get a nonzero value K.

As observed, since this claim only concerns values taken when variables are in $\{0, 1\}$, it is unaffected if we sprinkle in any arbitrary sequence of the linearization operators in between. We will sprinkle in linearization operators so that the intermediate polynomials arising in our sum check protocol all have low degree. Specifically, we use the expression

$$\forall_{X_1} L_1 \exists_{X_2} L_1 L_2 \forall_{X_3} L_1 L_2 L_3 \cdots \exists_{X_n} L_1 L_2 L_3 \cdots L_n P_\phi(X_1, \ldots, X_n)$$

The size of the expression is $O(1 + 2 + 3 + \cdots + n) = O(n^2)$.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

Now we give an inductive description of the protocol. Suppose for some polynomial $g(X_1, \ldots, X_k)$ the prover has the ability to convince the verifier that $g(a_1, a_2, \ldots, a_k) = C$ with probability 1 for any $a_1, a_2, \ldots, a_k, C$ when this is true and probability less than ϵ when it is false. Let $U(X_1, X_2, \ldots, X_l)$ be any polynomial on l variables obtained as

$$U(X_1, X_2, \ldots, X_l) = \mathcal{O}g(X_1, \ldots, X_k)$$

where $\mathcal{O}$ is either $\exists_{X_i}$, or $\forall_{X_i}$ or L_{X_i} for some variable. (Thus l is $k-1$ in the first two cases and k in the third.) Let d be an upper bound (known to the verifier) on the degree of U with respect to x_i. (In our case, $d \leq 3m$.) We show how the prover can convince the verifier that $U(a_1, a_2, \ldots, a_l) = C'$ with probability 1 for any $a_1, a_2, \ldots, a_k, C'$ for which it is true and with probability at most $\epsilon + d/p$ when it is false.

By renaming variables if necessary, assume $i = 1$. The verifier's check is as follows.

Case 1: $\mathcal{O} = \exists_{X_1}$. The prover provides a degree d polynomial $s(X_1)$ that is supposed to be $g(X_1, a_2, \ldots, a_k)$.

Verifier checks if $s(0) + s(1) = C'$. If not, it rejects. If yes, it picks a random value $a \in \mathbb{F}_p$ and asks prover to prove $s(a) = g(a, a_2, \ldots, a_k)$.

Case 2: $O = \forall_{X_1}$. Same as above but use $s(0) \cdot s(1)$ instead of $s(0) + s(1)$.

Case 3: $O = L_{X_1}$. Prover wishes to prove that $U(a_1, a_2, \ldots, a_k) = C'$. Prover provides a degree d polynomial $s(X_1)$ that is supposed to be $g(X_1, a_2, \ldots, a_k)$.

Verifier checks if $a_1 s(0) + (1 - a_1)s(1) = C'$. If not, it rejects. If yes, it picks random $a \in \mathbb{F}_p$ and asks prover to prove $s(a) = g(a, a_2, \ldots, a_k)$.

The proof of correctness follows as in case of $\#\mathsf{SAT}_\mathrm{D}$, by using the observation that if $s(X_1)$ is not the right polynomial, then with probability $1 - d/p$ the prover is still stuck with proving an incorrect statement at the next round. ■

An alternative proof of Theorem 8.19 is outlined in Exercise 8.8.

8.4 THE POWER OF THE PROVER

A curious feature of many known interactive proof systems is that in order to prove membership in language L, the prover needs to do more powerful computation than just deciding membership in L. We give some examples.

1. The public-coin system for graph nonisomorphism in Theorem 8.13 requires the prover to produce, for some randomly chosen hash function h and a random element y in the range of h, a graph H such that $h(H)$ is isomorphic to either G_1 or G_2 and $h(x) = y$. This seems harder than just solving graph nonisomorphism (though we do not know of any proof that it is).
2. The interactive proof for $\overline{\mathsf{3SAT}}$, a language in **coNP**, requires the prover to at the very least be able to compute $\#\mathsf{SAT}_\mathrm{D}$, which is not known to be computable in polynomial time even if we have an oracle for $\overline{\mathsf{3SAT}}$. In fact we see in Chapter 17 that the ability to compute $\#\mathsf{SAT}_\mathrm{D}$ is **#P**-complete, which implies $\mathbf{PH} \subseteq \mathbf{P}^{\#\mathsf{SAT}_\mathrm{D}}$.

In both cases, it is an open problem whether the protocol can be redesigned to use a weaker prover. By contrast, the protocol for TQBF is different from the previous

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

protocols in that the prover requires no more computational power than the ability to compute TQBF—the reason is that, as mentioned, the prover's replies can be computed in **PSPACE**, which reduces to TQBF. This observation underlies the following result, which is in the same spirit as the Karp-Lipton results described in Chapter 6, except the conclusion is stronger since **MA** is contained in $\mathbf{\Sigma_2}$ (indeed, **MA**-proof system for L with perfect completeness trivially implies that $L \in \mathbf{\Sigma_2}$). See also Lemma 20.18 in Chapter 20 for a related result.

Theorem 8.22 *If* $\mathbf{PSPACE} \subseteq \mathbf{P_{/poly}}$ *then* $\mathbf{PSPACE} = \mathbf{MA}$. ◇

PROOF: If $\mathbf{PSPACE} \subseteq \mathbf{P_{/poly}}$, then the prover in our TQBF protocol can be replaced by a circuit of polynomial size. Merlin (the prover) can just give this circuit to Arthur (the verifier) in Round 1, and Arthur then runs the interactive proof using this "prover." No more interaction is needed. Note that there is no need for Arthur to put blind trust in Merlin's circuit, since the correctness proof of the TQBF protocol shows that if the formula is not true, then *no* prover can make Arthur accept with high probability. ■

8.5 MULTIPROVER INTERACTIVE PROOFS (MIP)

It is also possible to define interactive proofs that involve more than one prover. The important assumption is that the provers do not communicate with each other *during the protocol*. They may communicate *before* the protocol starts and, in particular, agree upon a shared strategy for answering questions. The analogy often given is that of the police interrogating two suspects in separate rooms. The suspects may be accomplices who have decided upon a common story to tell the police, but since they are interrogated separately, they may inadvertently reveal an inconsistency in the story.

The set of languages with multiprover interactive provers is called **MIP**. The formal definition is analogous to Definition 8.6. We assume there are two provers (allowing polynomially many provers does not change the class; see Exercise 8.12), and in each round the verifier sends a query to each of them—the two queries need not be the same. Each prover sends a response in each round.

Clearly, $\mathbf{IP} \subseteq \mathbf{MIP}$ since the verifier can always simply ignore one prover. However, it turns out that **MIP** is probably strictly larger than **IP** (unless $\mathbf{PSPACE} = \mathbf{NEXP}$). That is, we have the following theorem.

Theorem 8.23 *(*[BFL90]*)* $\mathbf{NEXP} = \mathbf{MIP}$. ◇

We will say more about this theorem in Chapter 11, as well as a related class called **PCP**. Intuitively, one reason why two provers are more useful than one is that the second prover can be used to force *nonadaptivity*. That is, consider the interactive proof as an "interrogation" where the verifier asks questions and gets back answers from the prover. If the verifier wants to ensure that the answer of a prover to the question q is a function only of q and does not depend on the previous questions the prover heard, the prover can ask the second prover the question q and accept only if both answers agree with one another. This technique was used to show that multiprover interactive proofs can

be used to implement (and in fact are equivalent to) a model of a "probabilistically checkable proof in the sky." In this model, we go back to an **NP**-like notion of a proof as a static string, but this string may be huge and so is best thought of as a huge table, consisting of the prover's answers to all the possible verifier's questions. The verifier checks the proof by looking at only a few entries in this table that are chosen randomly from some distribution. If we let the class $\mathbf{PCP}[r,q]$ be the set of languages that can be proven using a table of size 2^r and q queries to this table then Theorem 8.23 can be restated as follows:

Theorem 8.24 *(Theorem 8.23, restated)* $\mathbf{NEXP} = \mathbf{PCP}[\mathrm{poly}, \mathrm{poly}] = \cup_c \mathbf{PCP}[n^c, n^c]$. ⋄

It turns out Theorem 8.23 can be scaled down to to obtain $\mathbf{NP} = \mathbf{PCP}[\mathrm{polylog}, \mathrm{polylog}]$. In fact (with a lot of work) the following has been shown; see Chapter 11.

Theorem 8.25 *(The* **PCP** *Theorem,* [AS92, ALM+92]*)* $\mathbf{NP} = \mathbf{PCP}[O(\log n), O(1)]$. ⋄

This theorem, which is described in Chapter 11 and proven in Chapter 22, has had many applications in complexity, and in particular establishes that for many **NP**-complete optimization problems, obtaining an *approximately optimal* solution is as hard as coming up with the optimal solution itself. Thus, it seems that complexity theory has come full circle with interactive proofs: By starting with **NP** and adding interaction, randomization, and multiple provers to it, we get to classes as high as **NEXP**, and then end up with new and fundamental insights about the class **NP**.

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 $P[C^P \text{ accepts } P(x)] < \frac{1}{3}$. ⋄

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

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 nonisomorphism)

The input for the problem of graph nonisomorphism 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 nonisomorphism.

Recall the IP for graph nonisomorphism:

- 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 nonisomorphism, 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), $\#\mathsf{SAT}_D$ and True Quantified Boolean Formulas (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

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

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.)

Example 8.31 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 is 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

$$\mathtt{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{8.16}$$

◇

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

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

equivalent to #SAT_D. In particular, if the permanent can be computed in polynomial time then **P** = **NP**. Here we show that the permanent is random-self-reducible. The main observation used is that if we think of $\mathtt{perm}(A)$ as a function of n^2 variables (denoting the entries of the matrix A), then by (8.16) 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.* ◇

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

- $\mathtt{perm}(B(x))$ 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 $\mathtt{perm}(B(a))$ 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) that given $n + 1$ (point, value) pairs $\{(a_i, b_i)|i \in [n + 1]\}$, there exists a unique 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 $B(0) = \mathtt{perm}(A)$. ■

The hypothesis of Theorem 8.33 can 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 #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 in Chapter 2 in the context of SAT (see also Exercise 8.9). In the case of permanent, this is the observation that

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

where $A_{1,i}$ is a $(n-1) \times (n-1)$ submatrix of A obtained by removing the first row and ith 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.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

For ease of notation, we assume the field $\mathbb{F}$ is equal to 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, $\texttt{perm}(D_A(x))$ 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 $L_{\texttt{perm}}$ to contain all tuples $\langle A, p, k \rangle$ such that $p > n^4$ is prime, A is an $n \times n$ matrix over GF(p), and $\texttt{perm}(A) = k$. We prove the following theorem.

Theorem 8.34 $L_{\texttt{perm}} \in$ **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 $\texttt{perm}(B) = k'$ 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 $\texttt{perm}(A) = k$ 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 $\texttt{perm}(D_A(x))$.
- *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 $g(b) = \texttt{perm}(D_A(b))$. 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 $\texttt{perm}(A) = k$, then an all-powerful prover can provide $\texttt{perm}(D_A(x))$ and thus by the inductive hypothesis make the verifier accept with probability 1.

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

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

and the verifier would immediately reject. Hence we only need to consider a prover that sends $g(x) \neq \texttt{perm}(D_A(x))$. 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.

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

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, and 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 **BPP** $\subseteq$ **PH** (Section 7.5.2) and the fact that one can approximate $\#\mathsf{SAT}_{\mathrm{D}}$ in $\mathbf{P}^{\mathbf{\Sigma}_2^p}$. *Multiprover* 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 terminology of Section 3.4, **IP** = **PSPACE** is a *nonrelativizing* theorem).

The result that **IP** = **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, Beaver and Feigenbaum [BF90] and Lipton [Lip91] published papers appeared that fleshed out the notion of random-self-reducibility and the connection to checking. Inspired by some of these developments,

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

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, and 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 **IP** =**PSPACE** [Sha90] and Babai, Fortnow, and Lund [BFL90] showed **MIP** = **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 **IP** = **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 **AM**[2] and hence probably not **NP**-hard (as mentioned in the introduction) is due to Goldreich and Goldwasser [GG98]. Aharonov and Regev [AR04] proved that approximating this problem to within $\sqrt{n}$ is in **NP** $\cap$ **coNP**.

EXERCISES

8.1. Prove the assertions about IP made in Section 8.1. That is, prove:

(a) Let **IP**′ denote the class obtained by allowing the prover to be probabilistic in Definition 8.6. That is, the prover's strategy can be chosen at random from some distribution on functions. Prove that **IP**′ = **IP**.

(b) Prove that **IP** $\subseteq$ **PSPACE**.

(c) Let **IP**′ denote the class obtained by changing the constant 2/3 in (8.2) to 1. Prove that **IP**′ = **IP**.

H534

(d) Let **IP**′ denote the class obtained by changing the constant 1/3 in (8.3) to 0. Prove that **IP**′ = **NP**.

8.2. Let **IP**′ denote the class obtained by requiring in the completeness condition (8.2) that there exists a single prover P for every $x \in L$ (rather than for every $x \in L$ there is a prover). Prove that **IP**′ = **IP**.

8.3. Show that **AM**[2] = **BP** · **NP**.

8.4. Let $k \leq n$. Prove that the following family $\mathcal{H}_{n,k}$ is a collection of pairwise independent functions from $\{0,1\}^n$ to $\{0,1\}^k$: Identify $\{0,1\}$ with the field GF(2). For every $k \times n$ matrix A with entries in GF(2), and $\mathbf{b} \in \mathrm{GF}(2)^k$, $\mathcal{H}_{n,k}$ contains the function $h_{A,\mathbf{b}} : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^k$ defined as $h_{A,\mathbf{b}}(x) = Ax + \mathbf{b}$.

8.5. Prove that there exists a perfectly complete **AM**[$O(1)$] protocol for proving a lower bound on set size.

H534

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press

8.6. Prove that for every **AM**[2] protocol for a language L, if the prover and the verifier repeat the protocol k times in parallel (verifier runs k independent random strings for each message) and the verifier accepts only if all k copies accept, then the probability that the verifier accepts $x \notin L$ is at most $(1/3)^k$. (Note that you *cannot* assume the prover is acting independently in each execution.) Can you generalize your proof for every k?

8.7. (Babai-Moran [BM88]) Prove that for every constant $k \geq 2$, $\mathbf{AM}[k+1] \subseteq \mathbf{AM}[k]$.

H534

8.8. In this exercise we explore an alternative way to generalize the proof that **coNP** $\subseteq$ **IP** to show that **IP** = **PSPACE**.

(a) Suppose that φ is a QBF formula (not necessarily in prenex form) satisfying the following property: If $x_1, \ldots, x_n$ are φ's variable sorted according to their order of appearance, then every variable x_i there is at most a single universal quantifier involving x_j (for $j > i$) appearing before the last occurrence of x_i in φ. Show that, in this case, when we run the sumcheck protocol of Section 8.3.2 with the modification that we use the check $s(0) \cdot s(1) = K$ for product operations, the prover only needs to send polynomials of degree $O(n)$.

H534

(b) Show that we can transform every size n QBF formula ψ into a logically equivalent size $O(n^2)$ formula φ with the above property.

H534

8.9. Define a language L to be *downward-self-reducible* if there's a polynomial-time algorithm R that for any n and $x \in \{0,1\}^n$, $R^{L_{n-1}}(x) = L(x)$ where by L_k we denote an oracle that solves L on inputs of size at most k. Prove that if L is downward-self-reducible, then $L \in$ **PSPACE**.

8.10. Complete the proof in Example 8.27 and show that graph isomorphism has a checker. Specifically, you have to show that if the program claims that $G_1 \cong G_2$, then we can do some further investigation (including calling the programs on other inputs) and with high probability conclude that either (a) the program was right on this input or (b) the program is wrong on *some* input and hence is not a correct program for graph isomorphism.

8.11. Show that **MIP** $\subseteq$ **NEXP**.

8.12. Show that if we redefine multiprover interactive proofs to allow, instead of two provers, as many as $m(n) = \text{poly}(n)$ provers on inputs of size n, then the class **MIP** is unchanged.

H535

https://doi.org/10.1017/CBO9780511804090.011 Published online by Cambridge University Press