4

Interactive Proofs

The first interactive proof that we cover is the sum-check protocol, due to Lund *et al.* [186]. The sum-check protocol has served as the single most important "hammer" in the design of efficient interactive proofs. Indeed, after introducing the sum-check protocol in Section 4.1, the remaining sections of this section apply the protocol in clean (but non-trivial) ways to solve a variety of important problems.

4.1 The Sum-Check Protocol

Suppose we are given a v-variate polynomial g defined over a finite field $\mathbb{F}$. The purpose of the sum-check protocol is for prover to provide the verifier with the following sum:

$$H := \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_v \in \{0,1\}} g(b_1, \ldots, b_v). \tag{4.1}$$

Summing up the evaluations of a polynomial over all Boolean inputs may seem like a contrived task with limited practical utility. But to the contrary, later sections of this section will show that many natural problems can be directly cast as an instance of Equation (4.1).

Remark 4.1. In full generality, the sum-check protocol can compute the sum $\sum_{b \in B^v} g(b)$ for any $B \subseteq \mathbb{F}$, but most of the applications covered in this survey will only require $B = \{0, 1\}$.

What does the verifier gain by using the sum-check protocol? The verifier could clearly compute H via Equation (4.1) on her own by evaluating g at 2^v inputs (namely, all inputs in $\{0, 1\}^v$), but we are thinking of 2^v as an unacceptably large runtime for the verifier. Using the sum-check protocol, the verifier's runtime will be

$$O(v + [\text{the cost to evaluate } g \text{ at a single input in } \mathbb{F}^v]).$$

This is much better than the 2^v evaluations of g required to compute H unassisted.

It also turns out that the prover in the sum-check protocol can compute all of its prescribed messages by evaluating g at $O(2^v)$ inputs in $\mathbb{F}^v$. This is only a constant factor more than what is required simply to compute H without proving correctness.

For presentation purposes, we assume for the rest of this section that the verifier has oracle access to g, i.e., $\mathcal{V}$ can evaluate $g(r_1, \ldots, r_v)$ for a randomly chosen vector $(r_1, \ldots, r_v) \in \mathbb{F}^v$ with a single query to an oracle.[1] A self-contained description of the sum-check protocol is provided in the codebox below. This is followed by a more intuitive, recursive description of the protocol.

Description of the Start of the Protocol. At the start of the sum-check protocol, the prover sends a value C_1 claimed to equal the true answer (i.e., the quantity H defined in Equation (4.1)). The sum-check

[1] This will not be the case in the applications described in later sections of this section. In our applications, $\mathcal{V}$ will either be able to efficiently evaluate $g(r_1, \ldots, r_v)$ unaided, or if this is not the case, $\mathcal{V}$ will ask the prover to *tell her* $g(r_1, \ldots, r_v)$, and $\mathcal{P}$ will subsequently prove this claim is correct via further applications of the sum-check protocol.

Description of Sum-Check Protocol.

- At the start of the protocol, the prover sends a value C_1 claimed to equal the value H defined in Equation (4.1).
- In the first round, $\mathcal{P}$ sends the univariate polynomial $g_1(X_1)$ claimed to equal

$$\sum_{(x_2,\ldots,x_v)\in\{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v).$$

  $\mathcal{V}$ checks that

$$C_1 = g_1(0) + g_1(1),$$

  and that g_1 is a univariate polynomial of degree at most $\deg_1(g)$, rejecting if not. Here, $\deg_j(g)$ denotes the degree of $g(X_1, \ldots, X_v)$ in variable X_j.
- $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and sends r_1 to $\mathcal{P}$.
- In the jth round, for $1 < j < v$, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $g_j(X_j)$ claimed to equal

$$\sum_{(x_{j+1},\ldots,x_v)\in\{0,1\}^{v-j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_v).$$

  $\mathcal{V}$ checks that g_j is a univariate polynomial of degree at most $\deg_j(g)$, and that $g_{j-1}(r_{j-1}) = g_j(0) + g_j(1)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and sends r_j to $\mathcal{P}$.
- In Round v, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $g_v(X_v)$ claimed to equal

$$g(r_1, \ldots, r_{v-1}, X_v).$$

  $\mathcal{V}$ checks that g_v is a univariate polynomial of degree at most $\deg_v(g)$, rejecting if not, and also checks that $g_{v-1}(r_{v-1}) = g_v(0) + g_v(1)$.
- $\mathcal{V}$ chooses a random element $r_v \in \mathbb{F}$ and evaluates $g(r_1, \ldots, r_v)$ with a single oracle query to g. $\mathcal{V}$ checks that $g_v(r_v) = g(r_1, \ldots, r_v)$, rejecting if not.
- If $\mathcal{V}$ has not yet rejected, $\mathcal{V}$ halts and accepts.

protocol proceeds in v rounds, one for each variable of g. At the start of the first round, the prover sends a polynomial $g_1(X_1)$ *claimed* to equal the polynomial $s_1(X_1)$ defined as follows:

$$s_1(X_1) := \sum_{(x_2,\ldots,x_v)\in\{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v). \tag{4.2}$$

$s_1(X_1)$ is defined to ensure that

$$H = s_1(0) + s_1(1). \tag{4.3}$$

Accordingly, the verifier checks that $C_1 = g_1(0) + g_1(1)$, i.e., the verifier checks that g_1 and the claimed answer C_1 are consistent with Equation (4.3).

Throughout, let $\deg_i(g)$ denote the degree of variable i in g. If the prover is honest, the polynomial $g_1(X_1)$ has degree $\deg_1(g)$. Hence g_1 can be specified with $\deg_1(g)+1$ field elements, for example by sending the evaluation of g_1 at each point in the set $\{0, 1, \ldots, \deg_1(g)\}$, or by specifying the $d+1$ coefficients of g_1.

The Rest of Round 1. Recall that the polynomial $g_1(X_1)$ sent by the prover in round 1 is claimed to equal the polynomial $s_1(X_1)$ defined in Equation (4.2). The idea of the sum-check protocol is that $\mathcal{V}$ will probabilistically check this equality of polynomials holds by picking a random field element $r_1 \in \mathbb{F}$, and confirming that

$$g_1(r_1) = s_1(r_1). \tag{4.4}$$

Clearly, if g_1 is as claimed, then this equality holds for all $r_1 \in \mathbb{F}$ (i.e., this probabilistic protocol for checking that $g_1 = s_1$ as formal polynomials is complete). Meanwhile, if $g_1 \neq s_1$, then with probability at least $1 - \deg_1(g)/|\mathbb{F}|$ over the verifier's choice of r_1, Equation (4.4) fails to hold. This is because two distinct degree d univariate polynomials agree on at most d inputs. This means that this protocol for checking that $g_1 = s_1$ by confirming that equality holds at a random input r_1 is sound, so long as $|\mathbb{F}| \gg \deg_1(g)$.

The remaining issue is the following: can $\mathcal{V}$ efficiently compute both $g_1(r_1)$ and $s_1(r_1)$, in order to check that Equation (4.4) holds? Since $\mathcal{P}$ sends $\mathcal{V}$ an explicit description of the polynomial g_1, it is possible for $\mathcal{V}$

to evaluate $g_1(r_1)$ in $O(\deg_1(g))$ time.[2] In contrast, evaluating $s_1(r_1)$ is not an easy task for $\mathcal{V}$, as s_1 is defined as a sum over 2^{v-1} evaluations of g. This is only a factor of two smaller than the number of terms in the sum defining H (Equation (4.1)). Fortunately, Equation (4.2) expresses s_1 as the sum of the evaluations of a $(v-1)$-variate polynomial over the Boolean hypercube (the polynomial being $g(r_1, X_2, \ldots, X_v)$ that is defined over the variables $X_2, \ldots, X_v$). This is exactly the type of expression that the sum-check protocol is designed to check. Hence, rather than evaluating $s_1(r_1)$ on her own, $\mathcal{V}$ instead *recursively* applies the sum-check protocol to evaluate $s_1(r_1)$.

Recursive Description of Rounds $2, \ldots, v$. The protocol thus proceeds in this recursive manner, with one round per recursive call. This means that in round j, variable X_j gets *bound* to a random field element r_j chosen by the verifier. This process proceeds until round v, in which the prover is forced to send a polynomial $g_v(X_v)$ claimed to equal $s_v := g(r_1, \ldots, r_{v-1}, X_v)$. When the verifier goes to check that $g_v(r_v) = s_v(r_v)$, there is no need for further recursion: since the verifier is given oracle access to g, $\mathcal{V}$ can evaluate $s_v(r_v) = g(r_1, \ldots, r_v)$ with a single oracle query to g.

Iterative Description of the Protocol. Unpacking the recursion described above, here is an equivalent description of what happens in round j of the sum-check protocol. At the start of round j, variables $X_1, \ldots, X_{j-1}$ have already been bound to random field elements $r_1, \ldots, r_{j-1}$. The prover sends a polynomial $g_j(X_j)$, and claims that

$$g_j(X_j) = \sum_{(x_{j+1}, \ldots, x_v) \in \{0,1\}^{v-j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_v). \tag{4.5}$$

The verifier compares the two most recent polynomials by checking

$$g_{j-1}(r_{j-1}) = g_j(0) + g_j(1), \tag{4.6}$$

[2] One may wonder, if the prover specifies g_1 via its evaluations at each input $i \in \{0, \ldots, \deg_1(g)\}$ rather than via its coefficients, how efficiently can the verifier evaluate $g_1(r_1)$? This is just Lagrange interpolation of a univariate polynomial (Section 2.4), which costs $O(\deg(g_1))$ field additions, multiplications, and inversions. In practical applications of the sum-check protocol, g will often have degree at most 2 or 3 in each of its variables, and hence this is very fast.

and rejecting otherwise (for round $j = 1$, the left hand side of Equation (4.6) is replaced with the claimed answer C_1). The verifier also rejects if the degree of g_j is too high: each g_j should have degree at most $\deg_j(g)$, the degree of variable x_j in g. If these checks pass,, $\mathcal{V}$ chooses a value r_j uniformly at random from $\mathbb{F}$ and sends r_j to $\mathcal{P}$.

In the final round, the prover has sent $g_v(X_v)$ which is claimed to be $g(r_1, \ldots, r_{v-1}, X_v)$. $\mathcal{V}$ now checks that $g_v(r_v) = g(r_1, \ldots, r_v)$ (recall that we assumed $\mathcal{V}$ has oracle access to g). If this check succeeds, and so do all previous checks, then the verifier is convinced that $H = g_1(0) + g_1(1)$.

Example Execution of the Sum-Check Protocol. Let $g(X_1, X_2, X_3) = 2X_1^3 + X_1X_3 + X_2X_3$. The sum of g's evaluations over the Boolean hypercube is $H = 12$. When the sum-check protocol is applied to g, the honest prover's first message in the protocol is the univariate polynomial $s_1(X_1)$ equal to:

$$\begin{aligned} &g(X_1, 0, 0) + g(X_1, 0, 1) + g(X_1, 1, 0) + g(X_1, 1, 1) \\ &\quad = (2X_1^3) + (2X_1^3 + X_1) + (2X_1^3) + (2X_1^3 + X_1 + 1) \\ &\quad = 8X_1^3 + 2X_1 + 1. \end{aligned}$$

The verifier checks that $s_1(0) + s_1(1) = 12$, and then sends the prover r_1. Suppose that $r_1 = 2$. The honest prover would then respond with the univariate polynomial

$$s_2(X_2) = g(2, X_2, 0) + g(2, X_2, 1) = 16 + (16 + 2 + X_2) = 34 + X_2.$$

The verifier checks that $s_2(0) + s_2(1) = s_1(r_1)$, which amounts in this example to confirming that $34 + (34 + 1) = 8 \cdot (2^3) + 4 + 1$; indeed, both the left hand side and right hand side equal 69. The verifier then sends the prover r_2. Suppose that $r_2 = 3$. The honest prover would respond with the univariate polynomial $s_3(X_3) = g(2, 3, X_3) = 16 + 2X_3 + 3X_3 = 16 + 5X_3$, and the verifier confirms that $s_3(0) + s_3(1) = s_2(r_2)$, which amounts in this example to confirming that $16 + 21 = 37$. The verifier picks a random field element r_3. Suppose that $r_3 = 6$. The verifier confirms that $s_3(6) = g(2, 3, 6)$ by making one oracle query to g.

Completeness and Soundness. The following proposition formalizes the completeness and soundness properties of the sum-check protocol.

Proposition 4.1. Let g be a v-variate polynomial of degree at most d in each variable, defined over a finite field $\mathbb{F}$. For any specified $H \in \mathbb{F}$, let $\mathcal{L}$ be the language of polynomials g (given as an oracle) such that

$$H = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_v \in \{0,1\}} g(b_1, \ldots, b_v).$$

The sum-check protocol is an interactive proof system for $\mathcal{L}$ with completeness error $\delta_c = 0$ and soundness error $\delta_s \leq vd/|\mathbb{F}|$.

Proof. Completeness is evident: if the prover sends the prescribed polynomial $g_j(X_j)$ at all rounds j, then $\mathcal{V}$ will accept with probability 1. We offer two proofs of soundness, the first of which reasons in a manner analogous to the iterative description of the protocol, and the second of which reasons in a manner analogous to the recursive description.

Non-Inductive Proof of Soundness. One way to prove soundness conceptually follows the iterative description of the sum-check protocol. Specifically, if $H \neq \sum_{(x_1,\ldots,x_v) \in \{0,1\}^v} g(x_1, x_2, \ldots, x_v)$, then the only way the prover can convince the verifier to accept is if there is at least one round i such that the prover sends a univariate polynomial $g_i(X_i)$ that does not equal the prescribed polynomial

$$s_i(X_i) = \sum_{(x_{i+1},\ldots,x_v) \in \{0,1\}^{v-i}} g(r_1, r_2, \ldots, r_{i-1}, X_i, x_{i+1}, \ldots, x_v),$$

and yet $g_i(r_i) = s_i(r_i)$. For every round i, g_i and s_i both have degree at most d, and hence if $g_i \neq s_i$, the probability that $g_i(r_i) = s_i(r_i)$ is at most $d/|\mathbb{F}|$. By a union bound over all v rounds, the probability that there is any round i such that the prover sends a polynomial $g_i \neq s_i$ yet $g_i(r_i) = s_i(r_i)$ is at most $dv/|\mathbb{F}|$.

Inductive Proof of Soundness. A second way to prove soundness is by induction on v (this analysis conceptually follows the recursive description of the sum-check protocol). In the case $v = 1$, $\mathcal{P}$'s only message specifies a degree d univariate polynomial $g_1(X_1)$. If $g_1(X_1) \neq g(X_1)$, then because any two distinct degree d univariate polynomials can agree on at most d inputs, $g_1(r_1) \neq g(r_1)$ with probability at least

$1 - d/|\mathbb{F}|$ over the choice of r_1, and hence $\mathcal{V}$'s final check will cause $\mathcal{V}$ to reject with probably at least $1 - d/|\mathbb{F}|$.

For $v \geq 2$, assume by way of induction that for all $v-1$-variate polynomials of degree at most d in each variable, the sum-check protocol has soundness error at most $(v-1)d/|\mathbb{F}|$. Let

$$s_1(X_1) = \sum_{x_2,\ldots,x_v \in \{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v).$$

Suppose $\mathcal{P}$ sends a polynomial $g_1(X_1) \neq s_1(X_1)$ in Round 1. Then because any two distinct degree d univariate polynomials can agree on at most d inputs, $g_1(r_1) = s_1(r_1)$ with probability at most $d/|\mathbb{F}|$. Conditioned on this event, $\mathcal{P}$ is left to prove the false claim in Round 2 that $g_1(r_1) = \sum_{(x_2,\ldots,x_v)\in\{0,1\}^{v-1}} g(r_1, x_2, \ldots, x_v)$. Since $g(r_1, x_2, \ldots, x_v)$ is a $(v-1)$-variate polynomial of degree at most d in each variable, the inductive hypothesis implies that $\mathcal{V}$ will reject at some subsequent round of the protocol with probability at least $1 - d(v-1)/|\mathbb{F}|$. Therefore, $\mathcal{V}$ will reject with probability at least

$$\begin{aligned}
&\Pr[s_1(r_1) \neq g_1(r_1)] - (1 - \Pr[\mathcal{V} \text{ rejects in some Round} \\
&\qquad j > 1 | s_1(r_1) \neq g_1(r_1)]) \\
&\geq \left(1 - \frac{d}{|\mathbb{F}|}\right) - \frac{d(v-1)}{|\mathbb{F}|} = 1 - \frac{dv}{|\mathbb{F}|}.
\end{aligned}$$

□

Discussion of Costs. There is one round in the sum-check protocol for each of the v variables of g. The total prover-to-verifier communication is $\sum_{i=1}^{v} (\deg_i(g) + 1) = v + \sum_{i=1}^{v} \deg_i(g)$ field elements, and the total verifier-to-prover communication is v field elements (one per round).[3] In particular, if $\deg_i(g) = O(1)$ for all j, then the communication cost is $O(v)$ field elements.[4]

[3] More precisely, the verifier does not need to send to the prover the random field element r_v chosen in the final round. However, when the sum-check protocol is used as a "subroutine" in a more involved protocol (e.g., the GKR protocol of Section 4.6), the verifier will often have to send that last field element to the prover to "continue" the more involved protocol.

[4] In practical applications of the sum-check protocol, $\mathbb{F}$ will often be a field of size between 2^{128} and 2^{256}, meaning that any field element can be specified with between

The running time of the verifier over the entire execution of the protocol is proportional to the total communication, plus the cost of a single oracle query to g to compute $g(r_1, \ldots, r_v)$.

Determining the running time of the prover is less straightforward. Recall that $\mathcal{P}$ can specify g_j by sending for each $i \in \{0, \ldots, \deg_j(g)\}$ the value:

$$g_j(i) = \sum_{(x_{j+1},\ldots,x_v)\in\{0,1\}^{v-j}} g(r_1, \ldots, r_{j-1}, i, x_{j+1}, \ldots, x_v). \tag{4.7}$$

An important insight is that the number of terms defining the value $g_j(i)$ in Equation (4.7) falls geometrically with j: in the jth sum, there are only $(1+\deg_j(g))\cdot 2^{v-j}$ terms, with the 2^{v-j} factor due to the number of vectors in $\{0,1\}^{v-j}$. Thus, the total number of terms that must be evaluated over the course of the protocol is $\sum_{j=1}^{v}(1+\deg_j(g))2^{v-j}$. If $\deg_j(g) = O(1)$ for all j, this is $O(1)\cdot\sum_{j=1}^{v} 2^{v-j} = O(1)\cdot(2^v-1) = O(2^v)$. Consequently, if $\mathcal{P}$ is given oracle access to g, then $\mathcal{P}$ will require just $O(2^v)$ time.

In all of the applications covered in this survey, $\mathcal{P}$ will not have oracle access to the evaluation table of g, and the key to many of the results in this survey is to show that $\mathcal{P}$ can nonetheless evaluate g at all of the necessary points in close to $O(2^v)$ total time.

The costs of the sum-check protocol are summarized in Table 4.1. Since $\mathcal{P}$ and $\mathcal{V}$ will not be given oracle access to g in applications, the table makes the number of oracle queries to g explicit.

Remark 4.2. An important feature of the sum-check protocol is that the verifier's messages to the prover are simply random field elements, and hence entirely independent of the input polynomial g. In fact, the only information $\mathcal{V}$ needs about the polynomial g to execute its part of the protocol is an upper bound on the degree of g in each of its v variables, and the ability to evaluate g at a random point $r \in \mathbb{F}^v$.[5]

16 and 32 bytes. These field sizes are large enough to ensure very low soundness error of the sum-check protocol, while being small enough that field operations remain fast.

[5] And $g(r)$ is needed by $\mathcal{V}$ only in order for the verifier to perform its final check of the prover's final message in the protocol. All other checks that $\mathcal{V}$ performs on the messages sent by $\mathcal{P}$ can be performed with no knowledge of g.

Table 4.1: Costs of the sum-check protocol when applied to a v-variate polynomial g over $\mathbb{F}$. Here, $\deg_i(g)$ denotes the degree of variable i in g, and T denotes the cost of an oracle query to g.

Communication	**Rounds**	$\mathcal{V}$ **time**	$\mathcal{P}$ **time**
$O\left(\sum_{i=1}^{v} \deg_i(g)\right)$ field elements	v	$O\left(v + \sum_{i=1}^{v} \deg_i(g)\right) + T$	$O\left(\sum_{i=1}^{v} \deg_i(g) \cdot 2^{v-i} \cdot T\right)$ $= O\left(2^v \cdot T\right)$ if $\deg_i(g)$ $= O(1)$ for all i

This means that $\mathcal{V}$ can apply the sum-check protocol *even without knowing the polynomial g to which the protocol is being applied*, so long as $\mathcal{V}$ knows an upper bound on the degree of the polynomial in each variable, and later obtains the ability to evaluate g at a random point $r \in \mathbb{F}^v$. In contrast, the prover does need to know the precise polynomial g in order to compute each of its messages over the course of the sum-check protocol.

Preview: Why multilinear extensions are useful: ensuring a fast prover. We will see several scenarios where it is useful to compute $H = \sum_{x \in \{0,1\}^v} f(x)$ for some function $f\colon \{0,1\}^v \to \mathbb{F}$ derived from the verifier's input. We can compute H by applying the sum-check protocol to any low-degree extension g of f. When g is itself a product of a small number of multilinear polynomials, then the prover in the sum-check protocol applied to g can be implemented extremely efficiently. Specifically, as we show later in Lemma 4.5, Lemma 3.6 (which gave an explicit expression for $\widetilde{f}$ in terms of Lagrange basis polynomials) can be exploited to ensure that enormous cancellations occur in the computation of the prover's messages, allowing fast computation.

Preview: Why using multilinear extensions is not always possible: ensuring a fast verifier. Although the use of the MLE $\widetilde{f}$ typically ensures fast computation for the prover, $\widetilde{f}$ cannot be used in all applications. The reason is that the verifier has to be able to evaluate $\widetilde{f}$ at a random point $r \in \mathbb{F}^v$ to perform the final check in the sum-check protocol, and in some settings, this computation would be too costly.

Lemma 3.8 gives a way for $\mathcal{V}$ to evaluate $\tilde{f}(r)$ in time $O(2^v)$, given all evaluations of f at Boolean inputs. This might or might not be an acceptable runtime, depending on the relationship between v and the verifier's input size n. If $v \leq \log_2 n + O(\log \log n)$, then $O(2^v) = \tilde{O}(n)$,[6] and the verifier runs in quasilinear time.[7] But we will see some applications where $v = c \log n$ for some constant $c > 1$, and others where $v = n$ (see, e.g., the #SAT protocol in Section 4.2). In these settings, $O(2^v)$ runtime for the verifier is unacceptable, and we will be forced to use an extension g of f that has a succinct representation, enabling $\mathcal{V}$ to compute $g(r)$ in much less than 2^v time. Sometimes $\tilde{f}$ itself has such a succinct representation, but other times we will be forced to use a higher-degree extension of f. See Exercise 4.2 and Exercise 4.3 (Parts (d) and (e)) for further details.

4.2 First Application of Sum-Check: #SAT $\in$ IP

Boolean Formulas and Circuits. A Boolean formula over variables $x_1, \ldots, x_n$ is a binary tree with each leaf labeled by a variable x_i or its negation, and each non-leaf node computing the AND or OR of its two children. Each node of the tree is also called a *gate*. The root of the tree is the output gate of the formula. The size S of the formula is the number of leaves of the tree. Note that many leaves of the formula may be labeled by the same variable x_i or its negation, so S may be much larger than n (see Figure 4.1 for an example).

A Boolean formula is identical to a Boolean *circuit*, except that in a formula, non-output gates are required to have fan-out 1, while in a circuit the fan-out of each gate can be unbounded. Here, the fan-out of a gate g in a circuit or formula refers to the number of other gates that g feeds into, i.e., the number of gates for which g is itself an input. See Figure 4.2 for an example of a Boolean circuit.[8]

[6] The notation $\tilde{O}(\cdot)$ hides polylogarithmic factors. So, for example, $n \log^4 n = \tilde{O}(n)$.

[7] Quasilinear time means time $\tilde{O}(n)$; i.e., at most a polylogarithmic factor more than linear.

[8] By convention, variable negation in Boolean circuits is typically depicted via explicit NOT gates, whereas in formulas, variable negation is depicted directly at the leaves.

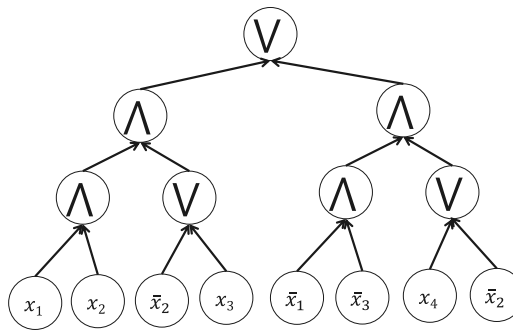

Figure 4.1: A Boolean formula ϕ over 4 variables of size 8. Here, $\vee$ denotes OR, $\wedge$ denotes AND, and $\overline{x}_i$ denotes the negation of variable x_i.

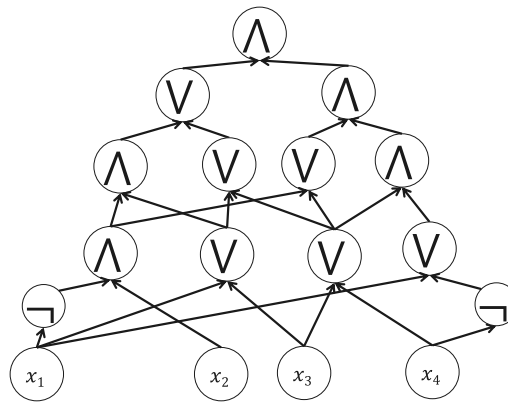

Figure 4.2: A Boolean circuit over 4 input variables. Here, $\vee$ denotes OR, $\wedge$ denotes AND, and $\neg$ denotes negation.

This means that circuits can "reuse intermediate values", in the sense that the value computed by one gate can be fed to multiple downstream gates. Whereas if a formula wants to reuse a value, it must recompute it from scratch, owing to the requirement that every AND and OR gate have fan-out 1. Visualized graph-theoretically, formulas have a binary-tree wiring pattern, while circuits can be arbitrary directed acyclic graphs. See Figure 4.2 for an example of a Boolean circuit.

The #SAT Problem. Let ϕ be any Boolean formula on n variables of size $S = \text{poly}(n)$.[9] Abusing notation, we will use ϕ to refer to both the formula itself and the function on $\{0,1\}^n$ that it computes. In the #SAT problem, the goal is to compute the number of satisfying assignments of ϕ. Equivalently, the goal is to compute

$$\sum_{x \in \{0,1\}^n} \phi(x). \tag{4.8}$$

#SAT is believed to be a very difficult problem, with the fastest known algorithms requiring time exponential in the number of variables n. This means that known algorithms do not do much better than the "brute force" approach of spending time $O(S)$ to evaluate the formula gate-by-gate at each of the 2^n possible assignments to the inputs. Even determining whether there exists one or more satisfying assignments to the formula is widely believed to require exponential time.[10] Nonetheless, there is an interactive proof protocol for #SAT in which the *verifier* runs in polynomial time.

The Interactive Proof for #SAT. Equation (4.8) sums up the evaluations of ϕ over all vectors in $\{0,1\}^n$. This is highly reminiscent of the kind of function that Lund *et al.* [186] designed the sum-check protocol to compute, namely the sum of g's evaluations over $\{0,1\}^n$ for some *low-degree polynomial* g. In order to apply the sum-check protocol to compute Equation (4.8), we need to identify a polynomial extension g of ϕ of total degree $\text{poly}(S)$ over a suitable finite field $\mathbb{F}$. The fact that g extends ϕ will ensure that $\sum_{x \in \{0,1\}^n} g(x) = \sum_{x \in \{0,1\}^n} \phi(x)$.[11] Moreover, we need the verifier to be able to evaluate g at a random point r in

[9] $S = \text{poly}(n)$ means that S is bounded above by $O(n^k)$ for some constant $k \geq 0$. We will assume $S \geq n$ to simplify statements of protocol costs—this will always be the case if ϕ depends on all n input variables.

[10] Readers familiar with the notion of **NP**-completeness will recognize formula satisfiability as an **NP**-complete problem, meaning that it has a polynomial time algorithm if and only if $\mathbf{P} = \mathbf{NP}$.

[11] More precisely, if the field $\mathbb{F}$ is of some prime size p, then $\sum_{x \in \{0,1\}^n} g(x)$ will equal the number of satisfying assignments of ϕ modulo p. There are a number of ways to address this issue if the exact number of satisfying assignments is desired. The simplest is to choose p larger than the maximum number of possible satisfying assignments, namely 2^n. This will still ensure a polynomial time verifier, as elements

polynomial time. Together with the fact that g has total degree $\text{poly}(S)$, this will ensure that the verifier in the sum-check protocol applied to g runs in time $\text{poly}(S)$. We define g as follows.

Let $\mathbb{F}$ be a finite field of size at least, say, S^4. In the application of the sum-check protocol below, the soundness error will be at most $S/|\mathbb{F}|$, so the field should be big enough to ensure that this quantity is acceptably small. If $|\mathbb{F}| \approx S^4$, then the soundness error is at most $1/S^3$. Bigger fields will ensure even smaller soundness error.

We can turn ϕ into an *arithmetic* circuit ψ over $\mathbb{F}$ that computes the desired extension g of ϕ. Here, an arithmetic circuit $\mathcal{C}$ has input gates, output gates, intermediate gates, and directed wires between them. Each gate computes addition or multiplication over a finite field $\mathbb{F}$. The process of replacing the Boolean formula ϕ with an arithmetic circuit ψ computing an extension polynomial of ϕ is called *arithmetization*.

For any gate in ϕ computing the AND of two inputs y, z, ψ replaces $\text{AND}(y, z)$ with multiplication of y and z over $\mathbb{F}$. It is easy to check that the bivariate polynomial $y \cdot z$ extends the Boolean function $\text{AND}(y, z)$, i.e., $\text{AND}(y, z) = y \cdot z$ for all $y, z \in \{0, 1\}$. Likewise, ψ replaces any gate computing $\text{OR}(y, z)$ by $y + z - y \cdot z$. Any formula leaf of the form $\bar{y}$ (i.e., the negation of variable y) is replaced by $1 - y$, This transformation is depicted in Figures 4.3 and 4.4. It is easy to check that $\psi(x) = \phi(x)$ for all $x \in \{0, 1\}^n$, and that the number of gates in the arithmetic circuit ψ is at most $3S$.

For the polynomial g computed by ψ, $\sum_{i=1}^{n} \deg_i(g) \leq S$.[12] Thus, the total communication cost of the sum-check protocol applied to g

of a field of this size can be written down and operated upon in time polynomial in n. Similar overflow issues arise frequently when designing proof systems that work over finite fields yet are meant to capture integer arithmetic. For other examples, see Footnote 19 and Sections 6.5.4.1 and 6.6.3.

[12]Here is an inductive proof of this fact. It is clearly true if ϕ consists of just one leaf. Suppose by way of induction that it is true when ϕ consists of at most $S/2$ leaves, and suppose that the output gate of ϕ is an AND gate (a similar argument applies if the output gate is an OR gate). The two in-neighbors of the output gate partition ϕ into two disjoint subformulas ϕ_1, ϕ_2 of sizes S_1, S_2 such that $S_1 + S_2 = S$ and $\phi(x) = \text{AND}(\phi_1(x), \phi_2(x))$. By the induction hypothesis, arithmetizing the two subformulas yields extension polynomials g_1, g_2 such that for $j = 1, 2$, $\sum_{i=1}^{n} \deg_i(g_j) \leq S_j$. The arithmetization of ϕ is $g = g_1 \cdot g_2$, which satisfies $\sum_{i=1}^{n} \deg_i(g) \leq S_1 + S_2 = S$.

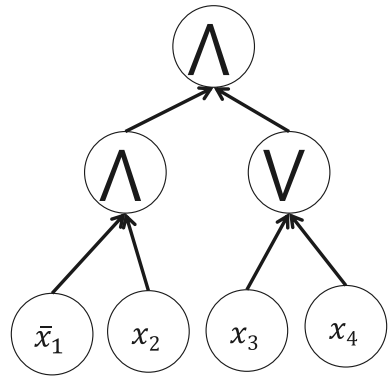

Figure 4.3: A Boolean formula ϕ over 4 variables of size 4. Here, $\vee$ denotes OR, $\wedge$ denotes AND, and $\overline{x}_1$ denotes the negation of variable x_1.

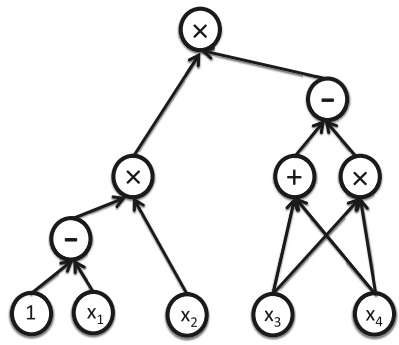

Figure 4.4: An arithmetic circuit ψ computing a polynomial extension g of ϕ over a finite field $\mathbb{F}$.

is $O(S)$ field elements, and $\mathcal{V}$ requires $O(S)$ time in total to check the first $n-1$ messages from $\mathcal{P}$. To check $\mathcal{P}$'s final message, $\mathcal{V}$ must also evaluate $g(r)$ for the random point $r \in \mathbb{F}^n$ chosen during the sum-check protocol. $\mathcal{V}$ can clearly evaluate $g(r)$ gate-by-gate in time $O(S)$. Since the polynomial g has n variables and $\sum_{i=1}^{n} \deg_i(g) \leq S$, the soundness error of the sum-check protocol applied to g is at most $S/|\mathbb{F}|$.

As explained in Section 4.1, the prover runs in time (at most) $2^n \cdot T \cdot (\sum_{i=1}^{n} \deg_i(g))$, where T is the cost of evaluating g at a point. Since g can be evaluated at any point in time $O(S)$ by evaluating ψ

Table 4.2: Costs of the #SAT protocol of Section 4.2 when applied to a Boolean formula $\phi : \{0,1\}^n \to \{0,1\}$ of size S.

Communication	**Rounds**	$\mathcal{V}$ **time**	$\mathcal{P}$ **time**
$O(S)$ field elements	n	$O(S)$	$O\left(S^2 \cdot 2^n\right)$

gate-by-gate, the prover in the #SAT protocol runs in time $O(S^2 \cdot 2^n)$. The costs of this protocol are summarized in Table 4.2.

IP = PSPACE. The above #SAT protocol comes quite close to establishing a famous result, namely that **IP** = **PSPACE** [186], [231].[13] That is, the class of problems solvable by interactive proofs with a polynomial-time verifier is exactly equal to the class of problems solvable in polynomial space. Here, we briefly discuss how to prove both directions of this result, i.e., that **IP** ⊆ **PSPACE** and that **PSPACE** ⊆ **IP**.

To show that **IP** ⊆ **PSPACE**, one needs to show that for any constant $c > 0$ and any language $\mathcal{L}$ solvable by an interactive proof in which the verifier's runtime is at most $O(n^c)$ there is an algorithm $\mathcal{A}$ that solves the problem in space at most, say, $O(n^{3c})$. Since c is a constant independent of n, so is $3c$ (albeit a larger one), and hence the space bound $O(n^{3c})$ is a polynomial in n.

Note that the resulting space-$O(n^{3c})$ algorithm might be *extremely* slow, potentially taking time *exponential* in n. That is, the inclusion **IP** ⊆ **PSPACE** does *not* state that any problem solvable by an interactive proof with an efficient verifier necessarily has a *fast* algorithm, but does state that the problem has a reasonably small-space algorithm.

Very roughly speaking, the algorithm $\mathcal{A}$ on input x will determine whether $x \in \mathcal{L}$ by ascertaining whether or not there is a prover strategy that causes the verifier to accept with probability at least 2/3. It does this by actually identifying an optimal prover strategy, i.e., finding the prover that maximizes the probability the verifier accepts on input x, and determining exactly what that probability is.

[13] Here, **PSPACE** is the class of decision problems that can be solved by some algorithm whose memory usage is bounded by some constant power of n.

In slightly more detail, it suffices to show that for any interactive proof protocol with the verifier running in time $O(n^c)$, that (a) an optimal prover strategy can be computed in space $O(n^{3c})$ and (b) the verifier's acceptance probability when the prover executes that optimal strategy can also be computed in space $O(n^{3c})$. Together, (a) and (b) imply that **IP** $\subseteq$ **PSPACE** because $x \in \mathcal{L}$ if and only if the optimal prover strategy induces the verifier to accept input x with probability at least 2/3.

Property (b) holds simply because for any fixed prover strategy $\mathcal{P}$ and input x, the probability the verifier accepts when interacting with $\mathcal{P}$ can be computed in space $O(n^c)$ by enumerating over every possible setting of the verifier's random coins and computing the fraction of settings that lead the verifier to accept. Again, note that this enumeration procedure is *extremely* slow—requiring time exponential in n—but can be done in space just $O(n^c)$, because if the verifier runs in time $O(n^c)$ then it also uses space at most $O(n^c)$. For a proof of Property (a), the interested reader is directed to [176, Lecture 17].[14]

The more challenging direction is to show that **PSPACE** $\subseteq$ **IP**. The #SAT protocol of Lund *et al.* [186] described above already contains the main ideas necessary to prove this. Shamir [231] extended the #SAT protocol to the **PSPACE**-complete language TQBF, and Shen [232] gave a simpler proof. We do not cover Shamir or Shen's extensions of the #SAT protocol here, since later (Section 4.5.5), we will provide a different and quantitatively stronger proof that **PSPACE** $\subseteq$ **IP**.

[14] As stated in [176, Lecture 17], the result that **IP** $\subseteq$ **PSPACE** is attributed to a manuscript by Paul Feldman in a paper by Goldwasser and Siper [136], and also follows from the analysis in [136].

4.3 Second Application: A Simple IP for Counting Triangles in Graphs

Section 4.2 used the sum-check protocol to give an IP for the #SAT problem, in which the verifier runs in time polynomial in the input size, and the prover runs in time exponential in the input size. This may not seem particularly useful, because in the real-world an exponential-time prover simply will not scale to even moderately-sized inputs. Ideally, we want provers that run in polynomial rather than exponential time, and we want verifiers that run in *linear* rather than polynomial time. IPs achieving such time costs are often called *doubly-efficient*, with the terminology chosen to highlight that both the verifier and prover are highly efficient. The remainder of this section is focused on developing doubly-efficient IPs.

As a warmup, in this section, we apply the sum-check protocol in a straightforward manner to give a simple, doubly-efficient IP for an important graph problem: counting triangles. We give an even more efficient (but less simple) IP for this problem in Section 4.5.1.

To define the problem, let $G = (V, E)$ be a simple graph on n vertices.[15] Here, V denotes the set of vertices of G, and E denotes the edges in G. Let $A \in \{0,1\}^{n \times n}$ be the adjacency matrix of G, i.e., $A_{i,j} = 1$ if and only if $(i, j) \in E$. In the counting triangles problem, the input is the adjacency matrix A, and the goal is to determine the number of unordered node triples $(i, j, k) \in V \times V \times V$ such that i, j, and k are all connected to each other, i.e., (i, j), (j, k) and (i, k) are all edges in E.

At first blush, it is totally unclear how to express the number of triangles in G as the sum of the evaluations of a low-degree polynomial g over all inputs in $\{0,1\}^v$, as per Equation (4.1). After all, the counting triangles problem itself makes no reference to any low-degree polynomial g, so where will g come from? This is where multilinear extensions come to the rescue.

For it to make sense to talk about multilinear extensions, we need to view the adjacency matrix A not as a matrix, but rather as a function

[15] A simple graph is one that is undirected and unweighted, with no self-loops or repeat edges.

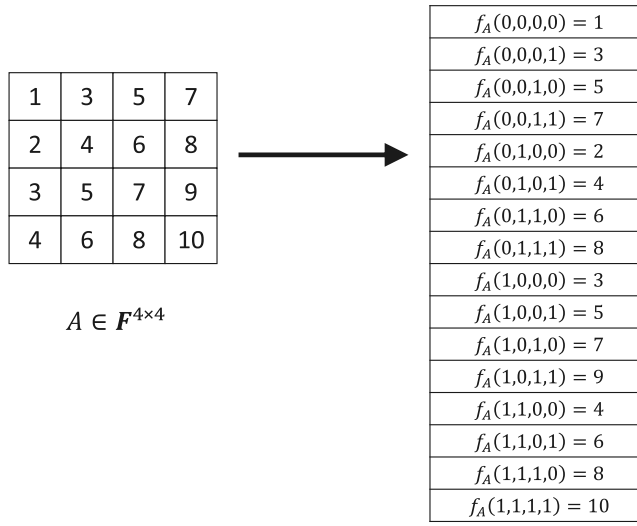

Figure 4.5: Example of how to view an $n \times n$ matrix A with entries from $\mathbb{F}$ as a function f_A mapping the domain $\{0,1\}^{\log_2(n)} \times \{0,1\}^{\log_2(n)}$ to $\mathbb{F}$, when $n = 4$. Note that there are n^2 entries of A, and n^2 vectors in $\{0,1\}^{\log_2(n)} \times \{0,1\}^{\log_2(n)}$. The entries of A are interpreted as the list of all n^2 evaluations of f_A.

f_A mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\{0,1\}$. The natural way to do this is to define $f_A(x,y)$ so that it interprets x and y as the binary representations of some integers i and j between 1 and n, and outputs $A_{i,j}$. See Figure 4.5 for an example.[16]

Then the number of triangles, Δ, in G can be written:

$$\Delta = \frac{1}{6} \sum_{x,y,z \in \{0,1\}^{\log n}} f_A(x,y) \cdot f_A(y,z) \cdot f_A(x,z). \tag{4.9}$$

To see that this equality is true, observe that the term for x, y, z in the above sum is 1 if edges (x,y), (y,z), and (x,z) all appear in G, and is 0 otherwise. The factor $1/6$ comes in because the sum over *ordered* node triples (i,j,k) counts each triangle 6 times, once for each permutation of i, j, and k.

[16]Figure 4.5 depicts a matrix A with arbitrary entries from some field $\mathbb{F}$. In the counting triangles problem as defined above, each entry of A is either 0 or 1, not an arbitrary field element.

Let $\mathbb{F}$ be a finite field of size $p \geq 6n^3$, where p is a prime, and let us view all entries of A as elements of $\mathbb{F}$. Here, we are choosing p large enough so that 6Δ is guaranteed to be in $\{0, 1, \ldots, p-1\}$, as the maximum number of triangles in any graph on n vertices is $\binom{n}{3} \leq n^3$. This ensures that, if we associate elements of $\mathbb{F}$ with integers in $\{0, 1, \ldots, p-1\}$ in the natural way, then Equation (4.9) holds even when all additions and multiplications are done in $\mathbb{F}$ rather than over the integers. (Choosing a large field to work over has the added benefit of ensuring good soundness error, as the soundness error of the sum-check protocol decreases linearly with field size.)

At last we are ready to describe the polynomial g to which we will apply the sum-check protocol to compute 6Δ. Recalling that $\widetilde{f}_A$ is the multilinear extension of f_A over $\mathbb{F}$, define the $(3 \log n)$-variate polynomial g to be:

$$g(X, Y, Z) = \widetilde{f}_A(X, Y) \cdot \widetilde{f}_A(Y, Z) \cdot \widetilde{f}_A(X, Z).$$

Equation (4.9) implies that:

$$6\Delta = \sum_{x,y,z \in \{0,1\}^{\log n}} g(x, y, z),$$

so applying the sum-check protocol to g yields an IP computing 6Δ.

Example. Consider the smallest non-empty graph, namely the two-vertex graph with a single undirected edge connecting the two vertices. There are no triangles in this graph. This is because there are fewer than three vertices in the entire graph, and there are no self-loops. That is, by the pigeonhole principle, for every triple of vertices (i, j, k), at least two of the vertices are the *same* vertex (i.e., at least one of $i = j$, $j = k$, or $i = k$ holds), and since there are no self-loops in the graph, these two vertices are not connected to each other by an edge. In this example, the adjacency matrix is

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

In this case,

$$\widetilde{f}_A(a, b) = a \cdot (1 - b) + b \cdot (1 - a),$$

and g is the following 3-variate polynomial:

$$g(X,Y,Z) = (X \cdot (1-Y) + Y \cdot (1-X))\,(Y \cdot (1-Z) \\ +Z \cdot (1-Y))\,(X \cdot (1-Z) + Z \cdot (1-X))\,.$$

It is not hard to see that $g(x,y,z) = 0$ for all $(x,y,z) \in \{0,1\}^3$, and hence applying the sum-check protocol to g reveals that the number of triangles in the graph is $\frac{1}{6} \cdot \sum_{(x,y,z)\in\{0,1\}^3} g(x,y,z) = 0$.

Costs of the Protocol. Since the polynomial g is defined over $3\log n$ variables, there are $3\log n$ rounds. Since g has degree at most 2 in each of its $3\log n$ variables, the total number of field elements sent by the prover in each round is at most 3. This means that the communication cost is $O(\log n)$ field elements ($9\log n$ elements sent from prover to verifier, and at most $3\log n$ sent from verifier to prover).

The verifier's runtime is dominated by the time to perform the final check in the sum-check protocol. This requires evaluating g at a random input $(r_1, r_2, r_3) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$, which in turn requires evaluating $\widetilde{f}_A(r_1, r_2)$, $\widetilde{f}_A(r_2, r_3)$ and $\widetilde{f}_A(r_1, r_3)$. Each of these 3 evaluations can be computed in $O(n^2)$ field operations using Lemma 3.8, which is linear in the size of the input matrix A.

The prover's runtime is clearly at most $O(n^5)$. This is because, since there are $3\log_2 n$ rounds of the protocol, it is sufficient for the prover to evaluate g at $O(n^3)$ inputs (see Table 4.1), and as explained in the previous paragraph, g can be evaluated at any input in $\mathbb{F}^{3\log n}$ in $O(n^2)$ time. In fact, more sophisticated algorithmic insights introduced in the next section can bring the prover runtime down to $O(n^3)$, which is competitive with the naive unverifiable algorithm for counting triangles that iterates over every triple of vertices and checks if they form a triangle. We omit further discussion of how to achieve prover time $O(n^3)$ in the protocol of this section, as Section 4.5.1 gives a different IP for counting triangles, in which the prover's runtime is much less than $O(n^3)$.

A Bird's Eye View. Hopefully the above protocol for counting triangles gives a sense of how problems that people care about in practice can be expressed as instances of Equation (4.1) in non-obvious ways. The general paradigm works as follows. An input x of length n is viewed as a

function f_x mapping $\{0,1\}^{\log n}$ to some field $\mathbb{F}$. And then the multilinear extension $\widetilde{f_x}$ of f_x is used in some way to construct a low-degree polynomial g such that, as per Equation (4.1), the desired answer equals the sum of the evaluations of g over the Boolean hypercube. The remaining sections of this section cover additional examples of this paradigm.

4.4 Third Application: Super-Efficient IP for MatMult

This section describes a highly optimized IP protocol for matrix multiplication (MatMult) from [237]. While this MatMult protocol is of interest in its own right, it is included here for didactic reasons: it displays, in a clean and unencumbered setting, all of the algorithmic insights that are exploited later in this survey to give more general IP and MIP protocols.

Given two $n \times n$ input matrices A, B over field $\mathbb{F}$, the goal of MatMult is to compute the product matrix $C = A \cdot B$.

4.4.1 Comparison to Freivalds' Protocol

Recall from Section 2.2 that, in 1977, Freivalds [115] gave the following verification protocol for MatMult: to check that $A \cdot B = C$, $\mathcal{V}$ picks a random vector $x \in \mathbb{F}^n$, and accepts if $A \cdot (Bx) = Cx$. $\mathcal{V}$ can compute $A \cdot (Bx)$ with two matrix-vector multiplications, which requires just $O(n^2)$ time. Thus, in Freivelds' protocol, $\mathcal{P}$ simply finds and sends the correct answer C, while $\mathcal{V}$ runs in optimal $O(n^2)$ total time. Today, Freivalds' protocol is regularly covered in introductory textbooks on randomized algorithms.

At first glance, Freivalds' protocol seems to close the book on verification protocols for MatMult, since the runtimes of both $\mathcal{V}$ and $\mathcal{P}$ are optimal: $\mathcal{P}$ does *no* extra work to prove correctness of the answer matrix C, $\mathcal{V}$ runs in time linear in the input size, and the protocol is even non-interactive ($\mathcal{P}$ just sends the answer matrix C to $\mathcal{V}$).

However, there is a sense in which it is possible to improve on Freivalds' protocol by introducing interaction between $\mathcal{P}$ and $\mathcal{V}$. In many settings, algorithms invoke MatMult, but they are not really interested in the full answer matrix. Rather, they apply a simple post-processing

step to the answer matrix to arrive at the quantity of true interest. For example, the best-known graph diameter algorithms repeatedly square the adjacency matrix of the graph, but ultimately they are not interested in the matrix powers—they are only interested in a single number. As another example, discussed in detail in Section 4.5.1, the fastest known algorithm for counting triangles in dense graphs invokes matrix multiplication, but is ultimately only interested in a single number, namely the number of triangles in the graph.

If Freivalds' protocol is used to verify the matrix multiplication steps of these algorithms, the actual product matrices must be sent for each step, necessitating $\Omega(n^2)$ communication. In practice, this can easily be many terabytes of data, even on graphs G with a few million nodes. Also, even if G is sparse, powers of G's adjacency matrix may be dense.

This section describes an interactive matrix multiplication protocol from [237] that preserves the runtimes of $\mathcal{V}$ and $\mathcal{P}$ from Freivalds' protocol, but avoids the need for $\mathcal{P}$ to send the full answer matrix in the settings described above—in these settings, the communication cost of the interactive protocol is just $O(\log n)$ field elements per matrix multiplication.

Preview: The Power of Interaction. This comparison of the interactive MatMult protocol to Freivalds' non-interactive one exemplifies the power of interaction in verification. Interaction buys the verifier the ability to ensure that the prover correctly materialized intermediate values in a computation (in this case, the entries of the product matrix C), without requiring the prover to explicitly materialize those values to the verifier. This point will become clearer later, when we cover the counting triangles protocol in Section 4.5.1. Roughly speaking, in that protocol, the prover convinces the verifier it correctly determined the squared adjacency matrix of the input graph, without ever materializing the squared adjacency matrix to the verifier.

Preview: Other Protocols for MatMult. An alternate interactive MatMult protocol can be obtained by applying the GKR protocol (covered later in Section 4.6) to a circuit $\mathcal{C}$ that computes the product C of two input matrices A, B. The verifier in this protocol runs in $O(n^2)$

time, and the prover runs in time $O(S)$, where S is the number of gates in $\mathcal{C}$.

The advantage of the MatMult protocol described in this section is two-fold. First, it does not care how the prover finds the right answer. In contrast, the GKR protocol demands that the prover compute the answer matrix C in a prescribed manner, namely by evaluating the circuit $\mathcal{C}$ gate-by-gate. Second, the prover in the protocol of this section simply finds the right answer and then does $O(n^2)$ extra work to prove correctness. This $O(n^2)$ term is a low-order additive overhead, assuming that there is no linear-time algorithm for matrix multiplication. In contrast, the GKR protocol introduces at least a constant factor overhead for the prover. In practice, this is the difference between a prover that runs many times slower than an (unverifiable) MatMult algorithm, and a prover that runs a fraction of a percent slower [237].

4.4.2 The Protocol

Given $n \times n$ input matrix A, B, recall that we denote the product matrix $A \cdot B$ by C. And as in Section 4.3, we interpret A, B, and C as functions f_A, f_B, f_C mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\mathbb{F}$ via:

$$f_A(i_1, \ldots, i_{\log n}, j_1, \ldots, j_{\log n}) = A_{ij}.$$

As usual, $\widetilde{f}_A$, $\widetilde{f}_B$, and $\widetilde{f}_C$ denote the MLEs of f_A, f_B, and f_C.

It is cleanest to describe the protocol for MatMult as a protocol for evaluating $\widetilde{f}_C$ at any given point $(r_1, r_2) \in \mathbb{F}^{\log n \times \log n}$. As we explain later (see Section 4.5), this turns out to be sufficient for application problems such as graph diameter and triangle counting.

The protocol for computing $\widetilde{f}_C(r_1, r_2)$ exploits the following explicit representation of the polynomial $\widetilde{f}_C(x, y)$.

Lemma 4.2. $\widetilde{f}_C(x, y) = \sum_{b \in \{0,1\}^{\log n}} \widetilde{f}_A(x, b) \cdot \widetilde{f}_B(b, y)$. Here, the equality holds as formal polynomials in the coordinates of x and y.

Proof. The left and right hand sides of the equation appearing in the lemma statement are both multilinear polynomials in the coordinates of x and y. Since the MLE of C is unique, we need only check that the left and right hand sides of the equation agree for all *Boolean*

vectors $i, j \in \{0,1\}^{\log n}$. That is, we must check that for Boolean vectors $i, j \in \{0,1\}^{\log n}$,

$$f_C(i,j) = \sum_{k \in \{0,1\}^{\log n}} f_A(i,k) \cdot f_B(k,j). \tag{4.10}$$

But this is immediate from the definition of matrix multiplication. □

With Lemma 4.2 in hand, the interactive protocol is immediate: we compute $\widetilde{f}_C(r_1, r_2)$ by applying the sum-check protocol to the $(\log n)$-variate polynomial $g(z) \coloneqq \widetilde{f}_A(r_1, z) \cdot \widetilde{f}_B(z, r_2)$.

Example. Consider the 2×2 matrices $A = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$ over $\mathbb{F}_5$. One can check that

$$A \cdot B = \begin{bmatrix} 0 & 4 \\ 2 & 0 \end{bmatrix}.$$

Viewing A and B as a functions mapping $\{0,1\}^2 \to \mathbb{F}_5$,

$$\widetilde{f}_A(x_1, x_2) = (1 - x_1)x_2 + 2x_1(1 - x_2) = -3x_1x_2 + 2x_1 + x_2,$$

and

$$\widetilde{f}_B(x_1, x_2) = (1-x_1)(1-x_2)+4x_1x_2 = 5x_1x_2 - x_1 - x_2 + 1 = 1 - x_1 - x_2,$$

where the final equality used the fact that we are working over $\mathbb{F}_5$, so the coefficient 5 is the same as the coefficient 0.

Observe that

$$\sum_{b \in \{0,1\}} \widetilde{f}_A(x_1, b) \cdot \widetilde{f}_B(b, x_2) = \widetilde{f}_A(x_1, 0) \cdot \widetilde{f}_B(0, x_2) + \widetilde{f}_A(x_1, 1) \cdot \widetilde{f}_B(1, x_2)$$
$$= 2x_1 \cdot (1 - x_2) + (-x_1 + 1) \cdot (-x_2) = -x_1x_2 + 2x_1 - x_2. \tag{4.11}$$

Meanwhile, viewing C as a function f_C mapping $\{0,1\}^2 \to \mathbb{F}_5$, we can calculate via Lagrange Interpolation:

$$\begin{aligned} \widetilde{f}_C(x_1, x_2) &= 4(1 - x_1)x_2 + 2x_1(1 - x_2) = -6x_1x_2 + 2x_1 + 4x_2 \\ &= -x_1x_2 + 2x_1 - x_2, \end{aligned}$$

where the final equality uses that $6 \equiv 1$ and $4 \equiv -1$ when working modulo 5. Hence, we have verified that Lemma 4.2 indeed holds for this particular example.

4.4.3 Discussion of Costs

Rounds and Communication Cost. Since g is a $(\log n)$-variate polynomial of degree 2 in each variable, the total communication is $O(\log n)$ field elements, spread over $\log n$ rounds.

$\mathcal{V}$'s Runtime. At the end of the sum-check protocol, $\mathcal{V}$ must evaluate $g(r_3) = \widetilde{f}_A(r_1, r_3) \cdot \widetilde{f}_B(r_3, r_2)$. To perform this evaluation, it suffices for $\mathcal{V}$ to evaluate $\widetilde{f}_A(r_1, r_3)$ and $\widetilde{f}_B(r_3, r_2)$. Since $\mathcal{V}$ is given the matrices A and B as input, Lemma 3.8 implies that both evaluations can be performed in $O(n^2)$ time.

$\mathcal{P}$'s Runtime. Recall that in each round k of the sum-check protocol $\mathcal{P}$ sends a quadratic polynomial $g_k(X_k)$ claimed to equal:

$$\sum_{b_{k+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,k-1}, X_i, b_{k+1}, \ldots b_{\log n}),$$

and to specify $g_k(X_k)$, $\mathcal{P}$ can just send the values $g_i(0), g_i(1)$, and $g_i(2)$. Thus, it is enough for $\mathcal{P}$ to evaluate g at all points of the form

$$(r_{3,1}, \ldots, r_{3,k-1}, \{0,1,2\}, b_{k+1}, \ldots, b_{\log n}) : (b_{k+1}, \ldots, b_{\log n}) \in \{0,1\}^{\log n - k}. \tag{4.12}$$

There are $3 \cdot n/2^k$ such points in round k.

We describe three separate methods to perform these evaluations. The first method is the least sophisticated and requires $\Theta(n^3)$ total time. The second method reduces the runtime to $\Theta(n^2)$ per round, for a total runtime bound of $\Theta(n^2 \log n)$ over all $\log n$ rounds. The third method is more sophisticated still—it enables the prover to *reuse work* across rounds, ensuring that $\mathcal{P}$'s runtime in round k is bounded by $O(n^2/2^k)$. Hence, the prover's total runtime is $O(\sum_k n^2/2^k) = O(n^2)$.

Method 1. As described when bounding $\mathcal{V}$'s runtime, g can be evaluated at any point in $O(n^2)$ time. Since there are $3 \cdot n/2^k$ points at which $\mathcal{P}$ must evaluate g in round k, this leads to a total runtime for $\mathcal{P}$ of $O(\sum_k n^3/2^k) = O(n^3)$.

Method 2. To improve on the $O(n^3)$ runtime of Method 1, the key is to exploit the fact that $3 \cdot n/2^k$ points at which $\mathcal{P}$ needs to evaluate

g in round k are not arbitrary points in $\mathbb{F}^{\log n}$, but are instead highly structured. Specifically, each such point z is in the form of Equation (4.12), and hence the trailing coordinates of z are all Boolean (i.e., $\{0,1\}$-valued). As explained below, this property ensures that ***each entry*** A_{ij} ***of*** A ***contributes to*** $g(r_{3,1},\ldots,r_{3,k-1},\{0,1,2\},b_{k+1},\ldots,b_{\log n})$ ***for only one tuple*** $(b_{k+1},\ldots,b_{\log n}) \in \{0,1\}^{\log n-k}$***, and similarly for each entry of*** B_{ij}***.*** Hence, $\mathcal{P}$ can make a single pass over the matrices A and B, and for each entry A_{ij} or B_{ij}, $\mathcal{P}$ only needs to update $g(z)$ for the three relevant tuples z of the form $(r_{3,1},\ldots,r_{3,k-1},\{0,1,2\},b_{k+1},\ldots,b_{\log n})$.

In more detail, in order to evaluate g at any input z, it suffices for $\mathcal{P}$ to evaluate $\widetilde{f}_A(r_1,z)$ and $\widetilde{f}_B(z,r_2)$. We explain the case of evaluating $\widetilde{f}_A(r_1,z)$ at all relevant points z, since the case of $\widetilde{f}_B(z,r_2)$ is identical. From Lemma 3.6 (Lagrange Interpolation), $\widetilde{f}_A(r_1,z) = \sum_{i,j\in\{0,1\}^{\log n}} A_{ij}\chi_{(i,j)}(r_1,z)$. For any input z of the form $(r_{3,1},\ldots,r_{3,k-1},\{0,1,2\},b_{k+1},\ldots,b_{\log n})$, notice that $\chi_{i,j}(r_1,z) = 0$ unless $(j_{k+1},\ldots,j_{\log n}) = (b_{k+1},\ldots,b_{\log n})$. This is because, for any coordinate ℓ such that $j_\ell \neq b_\ell$, the factor $(j_\ell b_\ell + (1-j_\ell)(1-b_\ell))$ appearing in the product defining $\chi_{(i,j)}$ equals 0 (see Equation (3.1)).

This enables $\mathcal{P}$ to evaluate $\widetilde{f}_A(r_1,z)$ in round k at all points z of the form of Equation (4.12) with a single pass over A: when $\mathcal{P}$ encounters entry A_{ij} of A, $\mathcal{P}$ updates $\widetilde{f}_A(z) \leftarrow \widetilde{f}_A(z) + A_{ij}\chi_{i,j}(z)$ for the three relevant values of z.

Method 3. To shave the last factor of $\log n$ off $\mathcal{P}$'s runtime, the idea is to have $\mathcal{P}$ reuse work across rounds. Very roughly speaking, the key fact that enables this is the following:

Informal Fact. If two entries $(i,j),(i',j') \in \{0,1\}^{\log n} \times \{0,1\}^{\log n}$ agree in their last ℓ bits, then $A_{i,j}$ and $A_{i',j'}$ contribute to the same three points in each of the final ℓ rounds of the protocol.

The specific points that they contribute to in each round $k \geq \log(n) - \ell$ are the ones of the form

$$z = (r_{3,1},\ldots,r_{3,k-1},\{0,1,2\},b_{k+1},\ldots,b_{\log n}),$$

where $b_{k+1}\ldots b_{\log n}$ equal the trailing bits of (i,j) and (i',j'). This turns out to ensure that $\mathcal{P}$ can treat (i,j) and (i',j') as a single entity

thereafter. There are only $O(n^2/2^k)$ entities of interest after k variables have been bound (out of the $2\log n$ variables over which $\widetilde{f}_A$ is defined). So the total work that $\mathcal{P}$ invests over the course of the protocol is

$$O\left(\sum_{k=1}^{2\log n} n^2/2^k\right) = O(n^2).$$

In more detail, the **Informal Fact** stated above is captured by the proof of the following lemma.

Lemma 4.3. Suppose that p is an ℓ-variate multilinear polynomial over field $\mathbb{F}$ and that A is an array of length 2^ℓ such that for each $x \in \{0,1\}^\ell$, $A[x] = p(x)$.[17] Then for any $r_1 \in \mathbb{F}$, there is an algorithm running in time $O(2^\ell)$ that, given r_1 and A as input, computes an array B of length $2^{\ell-1}$ such that for each $x' \in \{0,1\}^{\ell-1}$, $B[x'] = p(r_1, x')$.

Proof. The proof is reminiscent of that of Lemma 3.8. Specifically, we can express the multilinear polynomial $p(x_1, x_2, \ldots, x_\ell)$ via:

$$p(x_1, x_2, \ldots, x_\ell) = x_1 \cdot p(1, x_2, \ldots, x_\ell) + (1 - x_1) \cdot p(0, x_2, \ldots, x_\ell). \quad (4.13)$$

Indeed, the right hand side is clearly a multilinear polynomial that agrees with p at all inputs in $\{0,1\}^\ell$, and hence must equal p by Fact 3.5. The algorithm to compute B iterates over every value $x' \in \{0,1\}^{\ell-1}$ and sets $B[x'] \leftarrow r_1 \cdot A[1, x'] + (1 - r_1) \cdot A[0, x']$.[18] □

Lemma 4.3 captures Informal Fact because, while inputs $(0, x')$ and $(1, x')$ to p both contribute to $B[x']$, they contribute to no other entries of the array B. As we will see when we apply Lemma 4.3 repeatedly to compute the prover's messages in the sum-check protocol, once $B[x']$ is computed, the prover only needs to know $B[x']$, not $p(0, x')$ or $p(1, x')$ individually.

[17] Here, we associate bit-vectors x of length ℓ with indices into the array A of length 2^ℓ in the natural way.

[18] As in the statement of the lemma, here we associate bit-vectors x of length ℓ with indices into the array A of length 2^ℓ in the natural way, and similarly bit-vectors x' of length $\ell - 1$ with indices into the array B of length $2^{\ell-1}$.

Lemma 4.4. Let h be any ℓ-variate multilinear polynomial over field $\mathbb{F}$ for which all evaluations of $h(x)$ for $x \in \{0,1\}^\ell$ can be computed in time $O(2^\ell)$. Let $r_1, \ldots, r_\ell \in \mathbb{F}$ be any sequence of ℓ field elements. Then there is an algorithm that runs in time $O(2^\ell)$ and computes the following quantities:

$$\{h(r_1, \ldots, r_{i-1}, \{0,1,2\}, b_{i+1}, \ldots, b_\ell)\}_{i=1,\ldots,\ell;\ b_{i+1},\ldots,b_\ell \in \{0,1\}} \tag{4.14}$$

Proof. Let

$$S_i = \{h(r_1, \ldots, r_{i-1}, b_i, b_{i+1}, \ldots, b_\ell)\}_{b_i,\ldots,b_\ell \in \{0,1\}}.$$

Given all values in S_i, applying Lemma 4.3 to the $(\ell - i + 1)$-variate multilinear polynomial $p(X_i, \ldots, X_\ell) = h(r_1, \ldots, r_{i-1}, X_i, \ldots, X_\ell)$ implies that all values in S_{i+1} can be computed in time $O(2^{\ell-i})$.

Equation (4.13) further implies

$$\begin{aligned} h(r_1, \ldots, r_{i-1}, 2, b_{i+1}, \ldots, b_\ell) &= 2 \cdot h(r_1, \ldots, r_{i-1}, 1, b_{i+1}, \ldots, b_\ell) \\ &\quad - h(r_1, \ldots, r_{i-1}, 0, b_{i+1}, \ldots, b_\ell), \end{aligned}$$

and hence the values

$$\{h(r_1, \ldots, r_{i-1}, 2, b_{i+1}, \ldots, b_\ell)\}_{b_i,\ldots,b_\ell \in \{0,1\}}$$

can also be computed in $O(2^{\ell-i})$ time given the values in S_i.

It follows the total time required to compute all values in Equation (4.14) is $O(\sum_{i=1}^{\ell} 2^{\ell-i}) = O(2^\ell)$. □

Lemma 4.5. (Implicit in [103, Appendix B], see also [237], [251]) Let $p_1, p_2, \ldots, p_k$ be ℓ-variate multilinear polynomials. Suppose that for each p_i there is an algorithm that evaluates p_i at all inputs in $\{0,1\}^\ell$ in time $O(2^\ell)$. Let $g = p_1 \cdot p_2 \cdot \cdots \cdot p_k$ be the product of these multilinear polynomials. Then when the sum-check protocol is applied to the polynomial g, the honest prover can be implemented in $O(k \cdot 2^\ell)$ time.

Proof. As explained in Equation (4.12), the dominant cost in the honest prover's computation in the sum-check protocol lies in evaluating g at points of the form referred to in Lemma 4.4 (see Equation (4.14)). To obtain these evaluations, it clearly suffices to evaluate $p_1, \ldots, p_k$ at each one of these points, and multiply the results in time $O(k)$ per point.

Lemma 4.4 guarantees that each p_i can be evaluated at the relevant points in $O(2^\ell)$ time, yielding a total runtime of $O(k \cdot 2^\ell)$. See Figure 4.6 for a depiction of the honest prover's computation in the case $\ell = 3$. □

In the matrix multiplication protocol of this section, the sum-check protocol is applied to the $(\log_2 n)$-variate polynomial $g(X_3) = \widetilde{f}_A(r_1, X_3) \cdot \widetilde{f}_B(X_3, r_2)$. The multilinear polynomial $\widetilde{f}_A(r_1, X_3)$ can be evaluated at all inputs in $\{0,1\}^{\log n}$ in $O(n^2)$ time, by applying Lemma 4.4 with $h = \widetilde{f}_A$, and observing that the necessary evaluations of $\widetilde{f}_A(r_1, X_3)$ are a subset of the points in Equation (4.14) (with i in Equation (4.14) equal to $\log n$, and $(r_1, \ldots, r_{\log n})$ in Equation (4.14) equal to the entries of r_1). Similarly, $\widetilde{f}_B(X_3, r_2)$ can be evaluated at all inputs in $\{0,1\}^{\log n}$ in $O(n^2)$ time. Given all of these evaluations, Lemma 4.5 implies that the prover can execute its part of the sum-check protocol in just $O(n)$ additional time.

This completes the explanation of how the prover in the matrix multiplication protocol of this section executes its part of the sum-check protocol in $O(n^2)$ total time. All costs of the protocol are summarized in Table 4.3.

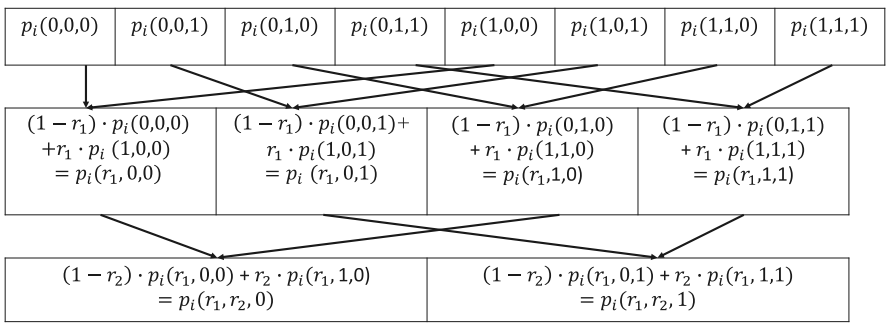

Figure 4.6: Depiction of the round-by-round evolution of the honest prover's internal data structure devoted to the polynomial p_i in Lemma 4.5 in the case $\ell = 3$ (recall this lemma considers the sum-check protocol applied to compute $\sum_{x \in \{0,1\}^\ell} p_1(x) \cdot \cdots \cdot p_k(x)$ when each p_i is multilinear). The top row is used by the prover to compute its prescribed message in the first round, the middle row for the second round, and the bottom row for the third round.

Table 4.3: Costs of the MatMult protocol of Section 4.4 when applied to $n \times n$ matrices A and B. Here, T is the time required by $\mathcal{P}$ to compute the product matrix $C = A \cdot B$.

Communication	Rounds	$\mathcal{V}$ time	$\mathcal{P}$ time
$O(\log n)$ field elements	$\log n$	$O\left(n^2\right)$	$T + O(n^2)$

4.5 Applications of the Super-Efficient MatMult IP

Why does an IP for computing $\widetilde{f}_C(r_1, r_2)$ rather than the full product matrix $C = A \cdot B$ suffice in applications? This section answers this question via several examples. With the exception of Section 4.5.5, all of the protocols in this section enable the honest prover to run the best-known algorithm to solve the problem at hand, and then do a low-order amount of extra work to prove the answer is correct. We refer to such IPs as *super-efficient* for the prover. There are no other known IPs or argument systems that achieve this super-efficiency while keeping the proof length sublinear in the input size.

4.5.1 A Super-Efficient IP For Counting Triangles

Algorithms often invoke MatMult to generate crucial *intermediate values* compute some product matrix C, but are not interested in the product matrix itself. For example, the fastest known algorithm for counting triangles in dense graphs works as follows. If A is the adjacency matrix of a simple graph, the algorithm first computes A^2 (it is known how to accomplish this in time $O(n^{2.3728639})$ [178]), and then outputs (1/6 times)

$$\sum_{i,j \in \{1,\ldots,n\}} (A^2)_{ij} \cdot A_{ij}. \tag{4.15}$$

It is not hard to see that Equation (4.15) quantity is six times the number of triangles in the graph, since $(A^2)_{i,j}$ counts the number of common neighbors of vertices i and j, and hence $A^2_{ij} \cdot A_{ij}$ equals the number of vertices k such that (i, j), (j, k) and (k, j) are all edges in the graph.

Clearly, the matrix A^2 is not of intrinsic interest here, but rather is a useful intermediate object from which the final answer can be quickly derived. As we explain in this section, it is possible to give an IP for counting triangles in which $\mathcal{P}$ essentially establishes that he correctly materialized A^2 and used it to generate the output via Equation (4.15). Crucially, $\mathcal{P}$ will accomplish this with only logarithmic communication (i.e., without sending A^2 to the verifier), and while doing very little extra work beyond determining A^2.

The Protocol. As in Section 4.3, let $\mathbb{F}$ be a finite field of size $p \geq 6n^3$, where p is a prime, and let us view all entries of A as elements of $\mathbb{F}$. Define the functions $f_A(x,y), f_{A^2}(x,y)\colon \{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \mathbb{F}$ that interprets x and y as the binary representations of some integers i and j between 1 and n, and outputs $A_{i,j}$ and $(A^2)_{i,j}$ respectively. Let $\widetilde{f}_A$ and $\widetilde{f}_{A^2}$ denote the multilinear extensions of f_A and f_{A^2} over $\mathbb{F}$.

Then the expression in Equation (4.15) equals $\sum_{x,y \in \{0,1\}^{\log n}} \widetilde{f}_{A^2}(x,y) \cdot \widetilde{f}_A(x,y)$. This quantity can be computed by applying the sum-check protocol to the multi-quadratic polynomial $\widetilde{f}_{A^2} \cdot \widetilde{f}_A$. At the end of this protocol, the verifier needs to evaluate $\widetilde{f}_{A^2}(r_1,r_2) \cdot \widetilde{f}_A(r_1,r_2)$ for a randomly chosen input $(r_1,r_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$. The verifier can evaluate $\widetilde{f}_A(r_1,r_2)$ unaided in $O(n^2)$ time using Lemma 3.8. While the verifier cannot evaluate $\widetilde{f}_{A^2}(r_1,r_2)$ without computing the matrix A^2 (which is as hard as solving the counting triangles problem on her own), evaluating $\widetilde{f}_{A^2}(r_1,r_2)$ is exactly the problem that the MatMult IP of Section 4.4.2 was designed to solve (as $A^2 = A \cdot A$), so we simply invoke that protocol to compute $\widetilde{f}_{A^2}(r_1,r_2)$.

Example. Consider the example from Section 4.3, in which the input matrix is

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

In this case,

$$A^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

One can check that

$$\widetilde{f}_A(X,Y) = X \cdot (1-Y) + Y \cdot (1-X),$$

and

$$\widetilde{f}_{A^2}(X,Y) = X \cdot Y + (1-Y) \cdot (1-X).$$

The counting triangles protocol in this section first applies the sum-check protocol to the following bivariate polynomial that has degree 2 in both of its variables:

$$\begin{aligned}\widetilde{f}_{A^2}(X,Y) \cdot \widetilde{f}_A(X,Y) = &(X \cdot (1-Y) + Y \cdot (1-X)) \\ &\cdot (X \cdot Y + (1-X) \cdot (1-Y)).\end{aligned}$$

It is easy to check that this polynomial evaluates to 0 for all four inputs in $\{0,1\}^2$, so applying the sum-check protocol to this polynomial reveals to the verifier that $\sum_{(x,y)\in\{0,1\}^2} \widetilde{f}_{A^2}(x,y) \cdot \widetilde{f}_A(x,y) = 0$.

At the end of the sum-check protocol applied to this polynomial, the verifier needs to evaluate $\widetilde{f}_{A^2}$ and $\widetilde{f}_A$ at a randomly chosen input $(r_1, r_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$. The verifier evaluates $\widetilde{f}_A(r_1, r_2)$ on its own. To compute $\widetilde{f}_{A^2}(r_1, r_2)$, the matrix multiplication IP is invoked. This protocol applies the sum-check protocol a second time, to the univariate quadratic polynomial

$$\begin{aligned}s(X) \coloneqq &\widetilde{f}_A(r_1, X) \cdot \widetilde{f}_A(X, r_2) = (r_1(1-X) + (1-r_1)X) \cdot (X(1-r_2) \\ &+ r_2(1-X)).\end{aligned}$$

This reveals to the verifier that

$$\widetilde{f}_{A^2}(r_1, r_2) = s(0) + s(1) = r_1 r_2 + (1-r_1)(1-r_2).$$

At the end of this second invocation of the sum-check protocol, the verifier needs to evaluate $s(r_3)$ for a randomly chosen $r_3 \in \mathbb{F}$. To do this, it suffices to evaluate $\widetilde{f}_A(r_1, r_3)$ and $\widetilde{f}_A(r_3, r_2)$, both of which the verifier computes on its own.

Costs of the Counting Triangles Protocol. The number of rounds, communication size, and verifier runtime of the IP of this section are all identical to the counting triangles protocol we saw earlier in Section 4.3 (namely, $O(\log n)$ rounds and communication, and $O(n^2)$ time verifier). The big advantage of the protocol of this section is in prover time: the prover in this section merely has to compute the matrix A^2 (it does not matter how $\mathcal{P}$ chooses to compute A^2), and then does $O(n^2)$ extra work

to compute the prescribed messages in the two invocations of the sum-check protocol. Up to the additive $O(n^2)$ term, this matches the amount of work performed by the fastest known (unverifiable) algorithm for counting triangles. The additive $O(n^2)$ is a low-order cost for $\mathcal{P}$, since computing A^2 with the fastest known algorithms requires super-linear time.

Communication and Rounds. In more detail, the application of sum-check to the polynomial $\widetilde{f}_{A^2} \cdot \widetilde{f}_A$ requires $2 \log n$ rounds, with 3 field elements sent from prover to verifier in each round. The matrix multiplication IP used to compute $\widetilde{f}_{A^2}(r_1, r_2)$ requires an additional $\log n$ rounds, with 3 field elements sent from the prover to verifier in each round. This means there are $3 \log n$ rounds in total, with $9 \log n$ field elements sent from the prover to the verifier (and $3 \log n$ sent from the verifier to the prover). This round complexity and communication cost is identical to the counting triangles protocol from Section 4.3.

Verifier runtime. The verifier is easily seen to run in $O(n^2)$ time in total–it's runtime is dominated by the cost of evaluating $\widetilde{f}_A$ at three inputs in $\mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$, namely (r_1, r_2), (r_2, r_3), and (r_1, r_3). This too is identical to the verifier cost in the counting triangles protocol from Section 4.3.

Prover runtime. Once the prover knows A^2, the prover's messages in both the sum-check protocol applied to the polynomial $\widetilde{f}_{A^2} \cdot \widetilde{f}_A$, and in the matrix multiplication IP of Section 4.4.2, can be derived in $O(n^2)$ time. Specifically, Method 3 of Section 4.4.3 achieves an $O(n^2)$ time prover in the matrix multiplication IP, and the same techniques show that, if $\mathcal{P}$ knows all of the entries of the matrix A^2, then in $O(n^2)$ time $\mathcal{P}$ can compute the prescribed messages when applying the sum-check protocol to the polynomial $\widetilde{f}_{A^2} \cdot \widetilde{f}_A$.

4.5.2 A Useful Subroutine: Reducing Multiple Polynomial Evaluations to One

In the counting triangles protocol just covered, at the end of the protocol the verifier needs to evaluate $\widetilde{f}_A$ at *three* points, (r_1, r_2), (r_2, r_3), and (r_1, r_3). This turns out to be a common occurrence: the sum-check

protocol is often applied to some polynomial g such that, in order to evaluate g at a single point, it is necessary to evaluate some other multilinear polynomial $\tilde{W}$ at multiple points.

For concreteness, let us begin by supposing that $\tilde{W}$ is a multilinear polynomial over $\mathbb{F}$ with $\log n$ variables, and the verifier wishes to evaluate $\tilde{W}$ at just two points, say $b, c \in \mathbb{F}^{\log n}$—we consider the case of three or more points at the end of this section. We cover a simple one-round interactive proof with communication cost $O(\log n)$ that reduces the evaluation of $\tilde{W}(b)$ and $\tilde{W}(c)$ to the evaluation of $\tilde{W}(r)$ for a *single* point $r \in \mathbb{F}^{\log n}$. What this means is that the protocol will force the prover $\mathcal{P}$ to send claimed values v_0 and v_1 for $\tilde{W}(b)$ and $\tilde{W}(c)$, as well as claimed values for many other points chosen by the verifier $\mathcal{V}$ in a specific manner. $\mathcal{V}$ will then pick r at random from those points, and it will be safe for $\mathcal{V}$ to believe that $v_0 = \tilde{W}(b)$ and $v_1 = \tilde{W}(c)$ so long as $\mathcal{P}$'s claim about $\tilde{W}(r)$ is valid. In other words, the protocol will ensure that if either $v_0 \neq \tilde{W}(b)$ or $v_1 \neq \tilde{W}(c)$, then with high probability over the $\mathcal{V}$'s choice of r, it will also be the case that the prover makes a false claim as to the value of $\tilde{W}(r)$.

The protocol. Let $\ell : \mathbb{F} \to \mathbb{F}^{\log n}$ be some canonical line passing through b and c. For example, we can let $\ell \colon \mathbb{F} \to \mathbb{F}^{\log n}$ be the unique line such that $\ell(0) = b$ and $\ell(1) = c$. $\mathcal{P}$ sends a univariate polynomial q of degree at most $\log n$ that is claimed to be $\widetilde{W} \circ \ell$, the restriction of $\widetilde{W}$ to the line ℓ. $\mathcal{V}$ interprets $q(0)$ and $q(1)$ as the prover's claims v_0 and v_1 as to the values of $\widetilde{W}(b)$ and $\widetilde{W}(c)$. $\mathcal{V}$ picks a random point $r^* \in \mathbb{F}$, sets $r = \ell(r^*)$, and interprets $q(r^*)$ as the prover's claim as to the value of $\tilde{W}(r)$.

A picture and an example. This technique is depicted pictorially in Figure 4.7. For a concrete example of how this technique works, suppose that $\log n = 2$, $b = (2, 4)$, $c = (3, 2)$, and $\widetilde{W}(x_1, x_2) = 3x_1x_2 + 2x_2$. Then the unique line $\ell(t)$ with $\ell(0) = b$ and $\ell(1) = c$ is $t \mapsto (t + 2, 4 - 2t)$. The restriction of $\widetilde{W}$ to ℓ is $3(t+2)(4-2t) + 2(4-2t) = -6t^2 - 4t + 32$. If $\mathcal{P}$ sends a degree-2 univariate polynomial q claimed to equal $\widetilde{W} \circ \ell$, the verifier will interpret $q(0)$ and $q(1)$ as claims about the values $\widetilde{W}(b)$ and $\widetilde{W}(c)$ respectively. The verifier will then pick a random $r^* \in \mathbb{F}$, set

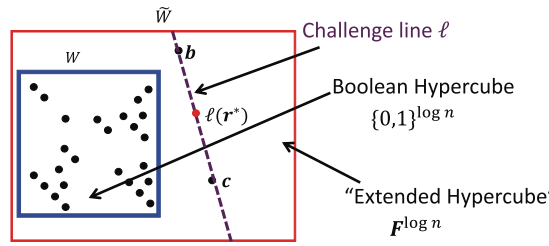

Figure 4.7: Schematic of how to reduce verifying claims about the values of $\tilde{W}(b)$ and $\tilde{W}(c)$ to a single claim about the value of $\tilde{W}(r)$. Here, $\tilde{W}$ is the multilinear extension of W, ℓ is the unique line passing through b and c, and $r = \ell(r^*)$ is a random point on ℓ.

$r = \ell(r^*)$, and interpret $q(r^*)$ as the claimed value of $\widetilde{W}(r)$. Observe that $\ell(r^*) = (r^* + 2, 4 - 2r^*)$ is a random point on the line ℓ.

The following claim establishes completeness and soundness of the above protocol.

Claim 4.6. Let $\tilde{W}$ be a multilinear polynomial over $\mathbb{F}$ in $\log n$ variables. If $q = \widetilde{W} \circ \ell$, then $q(0) = \tilde{W}(b)$, $q(1) = \tilde{W}(c)$, and $q(r^*) = \tilde{W}(\ell(r^*))$ for all $r^* \in \mathbb{F}$. Meanwhile, if $q \neq \widetilde{W} \circ \ell$, then with probability at least $1 - \log n/|\mathbb{F}|$ over a randomly chosen $r^* \in \mathbb{F}$, $q(r^*) \neq \tilde{W}(\ell(r^*))$.

Proof. The first claim is immediate from the fact that $\ell(0) = b$ and $\ell(1) = c$. For the second claim, observe that both q and $\widetilde{W} \circ \ell$ are univariate polynomials of degree at most $\log n$. If they are not the same polynomial, then the Schwartz-Zippel Lemma (even its simple special case for univariate polynomials) implies that when r^* is chosen at random from $\mathbb{F}$, $q(r^*) \neq \tilde{W}(\ell(r^*))$ with probability at least $1 - \log(n)/|\mathbb{F}|$. □

Reducing Three or More Evaluations to One. If the verifier needs to evaluate $\tilde{W}$ at more than two points, a similar protocol still applies. For example, suppose the verifier needs to know $\widetilde{W}(a)$, $\widetilde{W}(b)$, $\widetilde{W}(c)$. This time, let ℓ be a canonical *degree-two curve* passing through a, b, and c. For concreteness, we can let ℓ be the unique degree-2 curve with $\ell(0) = a$ and $\ell(1) = b$ and $\ell(2) = c$. For example, if $a = (0, 1)$, $b = (2, 2)$ and $c = (8, 5)$, then $\ell(t) = (2t^2, t^2 + 1)$.

Then $\mathcal{P}$ sends a univariate polynomial q of degree at most $2 \log n$ that is claimed to be $\widetilde{W} \circ \ell$. $\mathcal{V}$ interprets $q(0)$, $q(1)$ $q(2)$ as the prover's claims as to the values of $\widetilde{W}(a)$, $\widetilde{W}(b)$, and $\widetilde{W}(c)$. $\mathcal{V}$ picks a random point $r^* \in \mathbb{F}$, sets $r = \ell(r^*)$, and interprets $q(r^*)$ as the prover's claim as to the value of $\tilde{W}(r)$. Compared to the protocol for reducing two evaluations of $\widetilde{W}$ to one, the degree of q doubled from $\log n$ to $2 \log n$, and hence the prover-to-verifier communication increased by a factor of roughly 2, but remains $O(\log n)$. The protocol remains perfectly complete, and the soundness error increases from $1 - \log(n)/|\mathbb{F}$ to $1 - 2\log(n)/\mathbb{F}$.

This protocol could be applied at the end of both of the counting triangles protocols that we have covered, with $\tilde{W}$ equal to $\tilde{f}_A$, to reduce the number of points at which $\mathcal{V}$ needs to evaluate $\tilde{f}_A$ from three to one. As these evaluations are the dominant cost in $\mathcal{V}$'s runtime, this reduces $\mathcal{V}$ time by a factor of essentially 3. In the matrix powering protocol of the next section, the technique will be used to obtain more dramatic improvements in verification costs, and it will recur in the GKR protocol for circuit evaluation of Section 4.6.

4.5.3 A Super-Efficient IP for Matrix Powers

Let A be an $n \times n$ matrix with entries from field $\mathbb{F}$, and suppose a verifier wants to evaluate a single entry of the powered matrix A^k for a large integer k. For concreteness, let's say $\mathcal{V}$ is interested in learning entry $(A^k)_{n,n}$, and k and n are powers of 2. As we now explain, the MatMult IP of Section 4.4 gives a way to do this, with $O(\log(k) \cdot \log(n))$ rounds and communication, and a verifier that runs in $O(n^2 + \log(k)\log(n))$ time.

Clearly we can express the matrix A^k as a product of smaller powers of A:

$$A^k = A^{k/2} \cdot A^{k/2}. \tag{4.16}$$

Hence, letting g_ℓ denote the multilinear extension of the matrix A^ℓ, we can try to exploit Equation (4.16) by applying the MatMult IP to compute $(A^k)_{n,n} = g_k(\mathbf{1}, \mathbf{1})$.

At the end of the MatMult IP applied to two $n \times n$ matrices A', B', the verifier needs to evaluate $\widetilde{f}_{A'}$ and $\widetilde{f}_{B'}$ at the respective points (r_1, r_2) and (r_2, r_3), both in $\mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$. In the invocation of the MatMult IP

above, both A' and B' equal $A^{k/2}$. Hence, at the end of the MatMult IP, the verifier has to evaluate the polynomial $f_{A^{k/2}} = g_{k/2}$ at the two points (r_1, r_2) and (r_2, r_3). Unfortunately, the verifier cannot do this since she doesn't know $A^{k/2}$.

Reducing Two Points to One. Via the one-round interactive proof of Section 4.5.2 (see Claim 4.6 with $\tilde{W}$ equal to $g_{k/2}$), the verifier reduces evaluating a polynomial $g_{k/2}$ at the two points to evaluating $g_{k/2}$ at a single point.

Recursion to the Rescue. After reducing two points to one, the verifier is left with the task of evaluating $g_{k/2}$ at a single input, say $(r_3, r_4) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$. Since $g_{k/2}$ is the multilinear extension of the matrix $A^{k/2}$ (viewed in the natural way as a function $f_{A^{k/2}}$ mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \mathbb{F}$), and $A^{k/2}$ can be decomposed as $A^{k/4} \cdot A^{k/4}$, the verifier can recursively apply the MatMult protocol to compute $g_{k/2}(r_3, r_4)$. This runs into the same issues as before, namely that to run the MatMult protocol, the verifier needs to evaluate $g_{k/4}$ at two points, which can in turn be reduced to the task of evaluating $g_{k/4}$ at a single point. This can again be handled recursively as above. After $\log k$ layers of recursion, there is no need to recurse further since the verifier can evaluate $g_1 = \tilde{f}_A$ at any desired input in $O(n^2)$ time using Lemma 3.8.

4.5.4 A General Paradigm for IPs with Super-Efficient Provers

Beyond algorithms for counting triangles, there are other algorithms that invoke MatMult to compute some product matrix C, and then apply some post-processing to C to compute an answer that is much smaller than C itself (often the answer is just a single number, rather than an $n \times n$ matrix). In these settings, $\mathcal{V}$ can apply a general-purpose protocol, such as the GKR protocol that will be presented in Section 4.6, to verify that the post-processing step was correctly applied to the product matrix C. As we will see in Section 4.6, at the end of the application of the GKR protocol, $\mathcal{V}$ needs to evaluate $\widetilde{f}_C(r_1, r_2)$ at a randomly chosen point $(r_1, r_2) \in \mathbb{F}^{\log n \times \log n}$. $\mathcal{V}$ can do this using the MatMult protocol described above.

Crucially, this post-processing step typically requires time linear in the size of C. So $\mathcal{P}$'s runtime in this application of the GKR protocol will be proportional to the size of (a circuit computing) the post-processing step, which is typically just $\tilde{O}(n^2)$.

As a concrete example, consider the problem of computing the diameter of a directed graph G. Let A denote the adjacency matrix of G, and let I denote the $n \times n$ identity matrix. Then the diameter of G is the least positive number d such that $(A+I)^d_{ij} \neq 0$ for all (i,j). This yields the following natural protocol for diameter. $\mathcal{P}$ sends the claimed output d to V, as well as an (i,j) such that $(A+I)^{d-1}_{ij} = 0$. To confirm that d is the diameter of G, it suffices for $\mathcal{V}$ to check two things: first, that all entries of $(A+I)^d$ are nonzero, and second that $(A+I)^{d-1}_{ij}$ is indeed zero.[19]

The first task is accomplished by combining the MatMult protocol with the GKR protocol as follows. Let d_j denote the jth bit in the binary representation of d. Then $(A+I)^d = \prod_j^{\lceil \log d \rceil} (A+I)^{d_j 2^j}$, so computing the number of nonzero entries of $D_1 = (A+I)^d$ can be computed via a sequence of $O(\log d)$ matrix multiplications, followed by a post-processing step that computes the number of nonzero entries of D_1. We can apply the GKR protocol to verify this post-processing step, but at the end of the protocol, $\mathcal{V}$ needs to evaluate the multilinear extension of D_1 at a random point (as usual, when we refer to the multilinear extension of D_1, we are viewing D_1 as a function mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \mathbb{F}$ in the natural way). $\mathcal{V}$ cannot do this without help, so $\mathcal{V}$ outsources even this computation to $\mathcal{P}$, by using $O(\log d)$ invocations of the MatMult protocol described above.

The second task, of verifying that $(A+I)^{d-1}_{ij} = 0$, is similarly accomplished using $O(\log d)$ invocations of the MatMult protocol—since $\mathcal{V}$ is only interested in one entry of $(A+I)^{d-1}$, $\mathcal{P}$ need not send the matrix $(A+I)^{d-1}$ in full, and the total communication here is just polylog(n).

[19] If the interactive proof works over field $\mathbb{F}_p$, one does need to be careful that $(A+I)^{d-1}_{ij}$ is not positive and divisible by p. One technique for dealing with this is to have the verifier, after the prover sends (i,j), choose p to be a random prime in an appropriate interval. We omit further details for brevity.

Ultimately, $\mathcal{V}$'s runtime in this diameter protocol is $O(m \log n)$, where m is the number of edges in G. $\mathcal{P}$'s runtime in the above diameter protocol matches the best known unverifiable diameter algorithm up to a low-order additive term [225], [254], and the communication is just polylog(n).

4.5.5 An IP for Small-Space Computations (and IP = PSPACE)

In this section, we use the matrix-powering protocol to re-prove the following important result of Goldwasser *et al.* [135]: all problems solvable in logarithmic space have an IP with a quasilinear-time verifier, polynomial time prover, and polylogarithmic proof length.

The basic idea of the proof is that executing any Turing Machine M that uses s bits of space can be reduced to the problem of computing a single entry of A^{2^s} for a certain matrix A (A is in fact the configuration graph of M). So one can just apply the matrix-powering IP to A to determine the output of M. While A is a huge matrix (it has at least 2^s rows and columns), configuration graphs are highly structured, and this enables the verifier to evaluate $\tilde{f}_A$ at a single input in $O(s \cdot n)$ time. If s is logarithmic in the input size, then this means that the verifier in the IP runs in $O(n \log n)$ time.

The original paper of GKR proved the same result by constructing an arithmetic circuit for computing A^{2^s} and then applying a sophisticated IP for arithmetic circuit evaluation to that circuit (we cover this IP in Section 4.6 and the arithmetic circuit for computing A^{2^s} in Section 6.4). The approach described in this section is simpler, in that it directly applies a simple IP for matrix-powering, rather than the more complicated IP for the general circuit-evaluation problem.

Details. Let M be a Turing Machine that, when run on an m-bit input, uses at most s bits of space. Let $A(x)$ be the adjacency matrix of its *configuration graph* when M is run on input $x \in \{0,1\}^m$. Here, the configuration graph has as its vertex set all of the possible states and memory configurations of the machine M, with a directed edge from vertex i to vertex j if running M for one step from configuration i on input x causes M to move to configuration j. Since M uses s bits of space, there are $O(2^s)$ many vertices of the configuration graph. This

means that $A(x)$ is an $N \times N$ matrix for some $N = O(2^s)$. Note that if M never enters an infinite loop (i.e., never enters the same configuration twice), then M must trivially run in time at most N.

We can assume without loss of generality that M has a unique starting configuration and a unique accepting configuration; say for concreteness that these configurations correspond to vertices of the configuration graph with labels 1 and N. Then to determine whether M accepts input x, it is enough to determine whether there is a length-N path from vertex 0 to vertex N in the configuration graph of M. This is equivalent to determining the $(1, N)$'th entry of the matrix $(A(x))^N$.[20]

This quantity can be computed with the matrix power protocol of the previous section, which uses $O(s \cdot \log N)$ rounds and communication. At the end of the protocol, the verifier does need to evaluate the MLE of the matrix $A(x)$ at a randomly chosen input. This may seem like it should take up to $O(N^2)$ time, since A is a $N \times N$ matrix. However, the configuration matrix of any Turing Machine is highly structured, owing to the fact that at any time step, the machine only reads or writes to $O(1)$ memory cells, and only moves its read and write heads at most one cell to the left or right. This turns out to imply that the verifier can evaluate the MLE of A in $O(s \cdot m)$ time (we omit these details for brevity).

In total, the costs of the IP are as follows. The rounds and number of field elements communicated is $O(s \log N)$, the verifier's runtime is $O(s \log N + m \cdot s)$ and the prover's runtime is $\text{poly}(N)$. If $s = O(\log m)$, then these three costs are respectively $O(\log^2 m)$, $O(m \log m)$, and $\text{poly}(m)$. That is, the communication cost is polylogarithmic in the input size, the verifier's runtime is quasilinear, and the prover's runtime is polynomial.

Note that if $s = \text{poly}(m)$, then the verifier's runtime in this IP is $\text{poly}(m)$, recovering the famous result of LFKN [186] and Shamir [231] that **IP** = **PSPACE**.

[20] Since the configuration graph of M is acyclic (except for all halting states having self-loops), the entries of any power of $A(x)$ are all 0 or 1. This means that, unlike in Footnote 19 that discussed computing the diameter of general graphs, one does not need to worry about the possibility that $(1, N)$'th entry of $(A(x))^N$ is nonzero but divisible by the size p of the field over which the IP is defined.

Additional Discussion. One disappointing feature of this IP is that, if the runtime of M is significantly less than $N \geq 2^s$, the prover will still take time at least N, because the prover has to explicitly generate powers of the configuration graph's adjacency matrix. This is particularly problematic if the space bound s is superlogarithmic in the input size m, since then 2^s is not even a polynomial in m. Effectively, the IP we just presented forces the prover to explore all possible configurations of M, even though when running M on input x, the machine will only enter a tiny fraction of such configurations. A breakthrough complexity-theory result of [214] gave a very different IP that avoids this inefficiency for P. Remarkably, their IP also requires only a constant number of rounds of interaction.

4.6 The GKR Protocol and Its Efficient Implementation

4.6.1 Motivation

The goal of Section 4.2 was to develop an interactive proof for an intractable problem (such as #SAT [186] or TQBF [231]), in which the verifier ran in polynomial time. The perspective taken in this section is different: it acknowledges that there are no "real world" entities that can act as the prover in the #SAT and TQBF protocols of earlier sections, since real world entities cannot solve large instances of **PSPACE**-complete or #**P**-complete problems in the worst case. We would really like a "scaled down" result, one that is useful for problems that can be solved in the real world, such as problems in the complexity classes **P**, or **NC** (capturing problems solvable by efficient parallel algorithms), or even **L** (capturing problems solvable in logarithmic space).

One may wonder what is the point of developing verification protocols for such easy problems. Can't the verifier just ignore the prover and solve the problem without help? One answer is that this section will describe protocols in which the verifier runs much faster than would

be possible without a prover. Specifically, $\mathcal{V}$ will run linear time, doing little more than just reading the input.[21, 22]

Meanwhile, we will require that the prover not do much more than solve the problem of interest. Ideally, if the problem is solvable by a Random Access Machine or Turing Machine in time T and space s, we want the prover to run in time $O(T)$ and space $O(s)$, or as close to it as possible. At a minimum, $\mathcal{P}$ should run in polynomial time.

Can the TQBF and #SAT protocols of prior sections be scaled down to yield protocols where the verifier runs in (quasi-)linear time for a "weak" complexity class like **L**? It turns out that it can, but the prover is not efficient.

Recall that in the #SAT protocol (as well as in the TQBF protocol of [231]), $\mathcal{V}$ ran in time $O(S)$, and $\mathcal{P}$ ran in time $O(S^2 \cdot 2^N)$, when applied to a Boolean formula ϕ of size S over N variables. In principle, this yields an interactive proof for any problem solvable in space s: given an input $x \in \{0,1\}^n$, $\mathcal{V}$ first transforms x to an instance ϕ of TQBF (see, e.g., [9, Section 4] for a lucid exposition of this transformation, which is reminiscent of Savitch's Theorem [222]), and then applies the interactive proof for TQBF to ϕ.

However, the transformation yields a TQBF instance ϕ over $N = O(s \cdot \log T)$ variables when applied to a problem solvable in time T and space s. This results in a prover that runs in time in time $2^{O(s \cdot \log T)}$. This is superpolynomial (i.e., $n^{\Theta(\log n)}$), even if $s = O(\log n)$ and $T = \text{poly}(n)$. Until 2007, this was the state of the art in interactive proofs.

[21] The protocols for counting triangles, matrix multiplication and powering, and graph diameter of Sections 4.3–4.5 also achieved a linear-time verifier. But unlike the GKR protocol, those protocols were not general-purpose. As we will see, the GKR protocol is general-purpose in the sense that it solves the problem of arithmetic *circuit evaluation*, and any problem in **P** can be "efficiently" reduced to circuit evaluation (these reductions and the precise meaning of "efficiently" will be covered in Section 6).

[22] Another answer is that interactive proofs for "easy" problems can be combined with cryptography to turn them into succinct non-interactive arguments of knowledge (SNARKs), which allow the prover to establish that it knows a witness satisfying a specified property. In such SNARKs, the interactive proof only needs to solve the "easy" problem of checking that a purported witness satisfies the specified property.

4.6.2 The GKR Protocol and Its Costs

Goldwasser *et al.* [135] described a remarkable interactive proof protocol that does achieve many of the goals set forth above. The protocol is best presented in terms of the (arithmetic) *circuit evaluation* problem. In this problem, $\mathcal{V}$ and $\mathcal{P}$ first agree on a *log-space uniform* arithmetic circuit $\mathcal{C}$ of fan-in 2 over a finite field $\mathbb{F}$, and the goal is to compute the value of the output gate(s) of $\mathcal{C}$. A log-space uniform circuit $\mathcal{C}$ is one that possesses a succinct implicit description, in the sense that there is a logarithmic-space algorithm that takes as input the label of a gate a of $\mathcal{C}$, and is capable of determining all relevant information about that gate. That is, the algorithm can output the labels of all of a's neighbors, and is capable of determining if a is an addition gate or a multiplication gate.

Letting S denote the size (i.e., number of gates) of $\mathcal{C}$ and n the number of variables, the key feature of the GKR protocol is that the prover runs in time poly(S). We will see that $\mathcal{P}$'s time can even be made *linear* in S [102], [237], [251]. If $S = 2^{o(n)}$, then this is much better than the #SAT protocol that we saw in an earlier section, where the prover required time exponential in the number of variables over which the #SAT instance was defined.

Moreover, the costs to the verifier in the GKR protocol is $O(d \log S)$, which grows linearly with the *depth* d of $\mathcal{C}$, and only logarithmically with S. Crucially, this means that $\mathcal{V}$ can run in time *sublinear* in the size S of the circuit. At first glance, this might seem impossible—how can the verifier make sure the prover correctly evaluated $\mathcal{C}$ if the verifier never even "looks" at all of $\mathcal{C}$? The answer is that $\mathcal{C}$ was assumed to have a succinct implicit description in the sense of being log-space uniform. This enables $\mathcal{V}$ to "understand" the structure of $\mathcal{C}$ without ever having to look at every gate individually. The costs of the protocol are summarized in Table 4.4.

Application: An IP for Parallel Algorithms. The complexity class **NC** consists of languages solvable by parallel algorithms in time polylog(n) and total work poly(n). Any problem in **NC** can be computed by a

Table 4.4: Costs of the original GKR protocol [135] when applied to any log-space uniform layered arithmetic circuit $\mathcal{C}$ of size S and depth d over n variables defined over field $\mathbb{F}$. Section 4.6.5 describes methods from [102] for reducing $\mathcal{P}$'s runtime to $O(S \log S)$, and reducing the polylog(S) terms in the remaining costs to $O(\log S)$. It is now known how to achieve prover runtime of $O(S)$ for arbitrary layered arithmetic circuits $\mathcal{C}$ (see Remark 4.5).

Communication	Rounds	$\mathcal{V}$ Time	$\mathcal{P}$ Time	Soundness error
$d \cdot$ polylog(S) field elements	$d \cdot$ polylog(S)	$O(n + d \cdot$ polylog$(S))$	poly(S)	$O(d \log(S)/\lvert\mathbb{F}\rvert)$

log-space uniform arithmetic circuit $\mathcal{C}$ of polynomial size and polylogarithmic depth. Applying the GKR protocol to $\mathcal{C}$ yields a polynomial time prover and a linear time verifier.

4.6.3 Protocol Overview

As described above, $\mathcal{P}$ and $\mathcal{V}$ first agree on an arithmetic circuit $\mathcal{C}$ of fan-in 2 over a finite field $\mathbb{F}$ computing the function of interest. $\mathcal{C}$ is assumed to be in layered form, meaning that the circuit can be decomposed into layers, and wires only connect gates in adjacent layers (if $\mathcal{C}$ is not layered it can easily be transformed into a layered circuit $\mathcal{C}'$ with at most a factor-d blowup in size).[23] Suppose that $\mathcal{C}$ has depth d, and number the layers from 0 to d with layer d referring to the input layer, and layer 0 referring to the output layer.

In the first message, $\mathcal{P}$ tells $\mathcal{V}$ the (claimed) output(s) of the circuit. The protocol then works its way in iterations towards the input layer, with one iteration devoted to each layer. We describe the gates in $\mathcal{C}$ as having values: the value of an addition (respectively, multiplication) gate is set to be the sum (respectively, product) of its in-neighbors. The purpose of iteration i is to reduce a claim about the values of the gates at layer i to a claim about the values of the gates at layer $i + 1$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. This reduction is accomplished by applying the sum-check protocol.

[23] Recent work gives a variant of the GKR protocol that applies directly to non-layered circuits [255], avoiding a factor-d blowup in prover time.

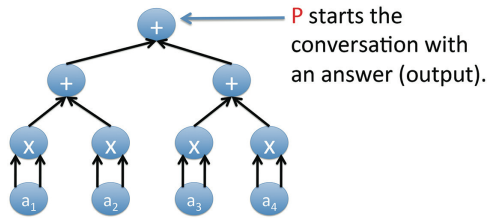

Figure 4.8: Start of GKR Protocol.

More concretely, the GKR protocol starts with a claim about the values of the output gates of the circuit, but $\mathcal{V}$ cannot check this claim without evaluating the circuit herself, which is precisely what she wants to avoid. So the first iteration uses a sum-check protocol to reduce this claim about the outputs of the circuit to a claim about the gate values at layer 2 (more specifically, to a claim about an evaluation of the multilinear extension of the gate values at layer 2). Once again, $\mathcal{V}$ cannot check this claim herself, so the second iteration uses another sum-check protocol to reduce the latter claim to a claim about the gate values at layer 3, and so on. Eventually, $\mathcal{V}$ is left with a claim about the inputs to the circuit, and $\mathcal{V}$ can check this claim without any help. This outline is depicted in Figures 4.8–4.11.

4.6.4 Protocol Details

Notation. Suppose we are given a layered arithmetic circuit $\mathcal{C}$ of size S, depth d, and fan-in two ($\mathcal{C}$ may have more than one output gate). Number the layers from 0 to d, with 0 being the output layer and d being the input layer. Let S_i denote the number of gates at layer i of

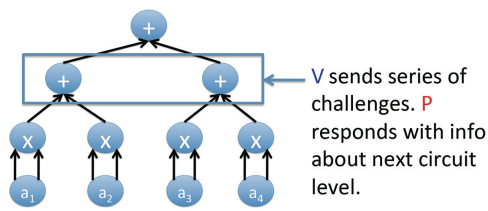

Figure 4.9: Iteration 1 reduces a claim about the output of $\mathcal{C}$ to one about the MLE of the gate values in the previous layer.

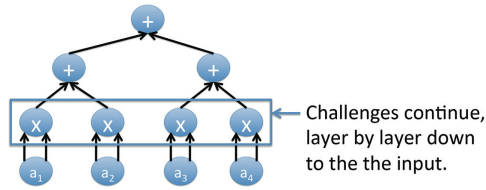

Figure 4.10: In general, iteration i reduces a claim about the MLE of gate values at layer i, to a claim about the MLE of gate values at layer $i+1$.

the circuit $\mathcal{C}$. Assume S_i is a power of 2 and let $S_i = 2^{k_i}$. The GKR protocol makes use of several functions, each of which encodes certain information about the circuit.

Number the gates at layer i from 0 to $S_i - 1$, and let $W_i : \{0,1\}^{k_i} \to \mathbb{F}$ denote the function that takes as input a binary gate label, and outputs the corresponding gate's value at layer i. As usual, let $\widetilde{W}_i$ denote the multilinear extension of W_i. See Figure 4.12, which depicts an example circuit $\mathcal{C}$ and input to $\mathcal{C}$ and describes the resulting function W_i for each layer i of $\mathcal{C}$.

The GKR protocol also makes use of the notion of a "wiring predicate" that encodes which pairs of wires from layer $i+1$ are connected to a given gate at layer i in $\mathcal{C}$. Let $\text{in}_{1,i}, \text{in}_{2,i} \colon \{0,1\}^{k_i} \to \{0,1\}^{k_{i+1}}$ denote the functions that take as input the label a of a gate at layer i of $\mathcal{C}$,

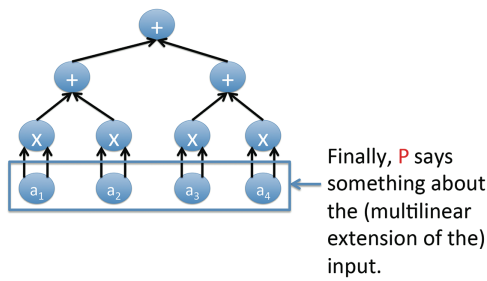

Figure 4.11: In the final iteration, $\mathcal{P}$ makes a claim about the MLE of the input (here, the input of length n with entries in $\mathbb{F}$ is interpreted as a function mapping $\{0,1\}^{\log_2 n} \to \mathbb{F}$. Any such function has a unique MLE by Fact 3.5). $\mathcal{V}$ can check this claim without help, since $\mathcal{V}$ sees the input explicitly.

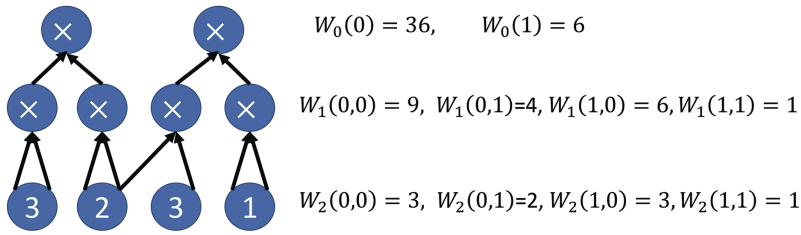

Figure 4.12: Example circuit $\mathcal{C}$ and input x, and resulting functions W_i for each layer i of $\mathcal{C}$. Note that $\mathcal{C}$ has two output gates.

and respectively output the label of the first and second in-neighbor of gate a. So, for example, if gate a at layer i computes the sum of gates b and c at layer $i+1$, then $\text{in}_{1,i}(a) = b$ and $\text{in}_{2,i}(a) = c$.

Define two functions, add_i and mult_i, mapping $\{0,1\}^{k_i + 2k_{i+1}}$ to $\{0,1\}$, which together constitute the wiring predicate of layer i of C. Specifically, these functions take as input three gate labels (a,b,c), and return 1 if and only if $(b,c) = (\text{in}_{1,i}(a), \text{in}_{2,i}(a))$ and gate a is an addition (respectively, multiplication) gate. As usual, let $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ denote the multilinear extensions of add_i and mult_i.

For an example, consider the circuit depicted in Figure 4.12. Since the circuit contains no addition gates, add_0 and add_1 are the constant 0 function. Meanwhile, mult_0 is the function defined over domain $\{0,1\} \times \{0,1\}^2 \times \{0,1\}^2$ as follows. mult_0 evaluates to 1 on the following two inputs: $(0,(0,0),(0,1))$ and $(1,(1,0),(1,1))$. On all other inputs, mult_0 evaluates to zero. This is because the first and second in-neighbors of gate 0 at layer 0 are respectively gates $(0,0)$ and $(0,1)$ at layer 1, and similarly the first and second in-neighbors of gate 1 at layer 0 are respectively gates $(1,0)$ and $(1,1)$ at layer 1.

Similarly, mult_1 is a function on domain $\{0,1\}^2 \times \{0,1\}^2 \times \{0,1\}^2$. It evaluates to 0 on all inputs except for the following four, on which it evaluates to 1:

- $((0,0),(0,0),(0,0))$.
- $((0,1),(0,1),(0,1))$.

- $((1,0),(0,1),(1,0))$.
- $((1,1),(1,1),(1,1))$.

Note that for each layer i, add_i and mult_i depend only on the circuit $\mathcal{C}$ and not on the input x to $\mathcal{C}$. In contrast, the function W_i *does* depend on x. This is because W_i maps each gate label at layer i to the value of the gate when $\mathcal{C}$ is evaluated on input x.

Detailed Description. The GKR protocol consists of d iterations, one for each layer of the circuit. Each iteration i starts with $\mathcal{P}$ claiming a value for $\widetilde{W}_i(r_i)$ for some point in $r_i \in \mathbb{F}^{k_i}$.

At the start of the first iteration, this claim is derived from the claimed outputs of the circuit. Specifically, if there are $S_0 = 2^{k_0}$ outputs of $\mathcal{C}$, let $D\colon \{0,1\}^{k_0} \to \mathbb{F}$ denote the function that maps the label of an output gate to the claimed value of that output. Then the verifier can pick a random point $r_0 \in \mathbb{F}^{k_0}$, and evaluate $\tilde{D}(r_0)$ in time $O(S_0)$ using Lemma 3.8. By the Schwartz-Zippel lemma, if $\tilde{D}(r_0) = \widetilde{W}_0(r_0)$ (i.e., if the multilinear extension of the claimed outputs equals the multilinear extension of the correct outputs when evaluated at a randomly chosen point), then it is safe for the verifier to believe that $\tilde{D}$ and $\tilde{W}_0$ are the same polynomial, and hence that all of the claimed outputs are correct. Unfortunately, the verifier cannot evaluate $\widetilde{W}_0(r_0)$ without help from the prover.[24]

The purpose of iteration i is to reduce the claim about the value of $\widetilde{W}_i(r_i)$ to a claim about $\widetilde{W}_{i+1}(r_{i+1})$ for some $r_{i+1} \in \mathbb{F}^{k_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. To accomplish this, the iteration applies the sum-check protocol to a specific polynomial derived from $\widetilde{W}_{i+1}$,

[24] Throughout this survey, a statement of the form "if $p(r) = q(r)$ for a random r, then it is safe for the verifier to believe that $p = q$ as formal polynomials" is shorthand for the following: if $p \neq q$, then the former equality fails to hold with overwhelming probability over the random choice of r, i.e., the prover would have to "get unreasonably lucky" to pass the check.

$\widetilde{\text{add}}_i$, and $\widetilde{\text{mult}}_i$. Our description of the protocol actually makes use of a simplification due to Thaler [238].

Applying the Sum-Check Protocol. The GKR protocol exploits an ingenious explicit expression for $\widetilde{W}_i(r_i)$, captured in the following lemma.

Lemma 4.7.

$$\tilde{W}_i(z) = \sum_{b,c \in \{0,1\}^{k_{i+1}}} \widetilde{\text{add}}_i(z,b,c)\left(\tilde{W}_{i+1}(b) + \tilde{W}_{i+1}(c)\right) + \widetilde{\text{mult}}_i(z,b,c)\left(\tilde{W}_{i+1}(b) \cdot \tilde{W}_{i+1}(c)\right) \tag{4.17}$$

Proof. It is easy to check that the right hand side is a multilinear polynomial in the entries of z, since $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ are multilinear polynomials. (Note that, just as in the matrix multiplication protocol of the Section 4.4, the function being summed over is *quadratic* in the entries of b and c, but this quadratic-ness is "summed away", leaving a multilinear polynomial only in the variables of z).

Since the multilinear extension of a function with domain $\{0,1\}^{k_i}$ is unique, it suffices to check that the left hand side and right hand side of the expression in the lemma agree for all $a \in \{0,1\}^{k_i}$. To this end, fix any $a \in \{0,1\}^{s_i}$, and suppose that gate a in layer i of $\mathcal{C}$ is an addition gate (the case where gate a is a multiplication gate is similar). Since each gate a at layer i has two unique in-neighbors, namely $\text{in}_1(a)$ and $\text{in}_2(a)$;

$$\text{add}_i(a,b,c) = \begin{cases} 1 & \text{if } (b,c) = (\text{in}_1(a), \text{in}_2(a)) \\ 0 & \text{otherwise} \end{cases}$$

and $\text{mult}_i(a,b,c) = 0$ for all $b, c \in \{0,1\}^{k_{i+1}}$.

Hence, since $\widetilde{\text{add}}_i$, $\widetilde{\text{mult}}_i$, $\widetilde{W}_{i+1}$, and $\widetilde{W}_i$ extend add_i and mult_i, W_{i+1}, and W_i respectively,

$$\sum_{b,c\in\{0,1\}^{k_{i+1}}} \widetilde{\text{add}}_i(a,b,c)$$
$$\left(\tilde{W}_{i+1}(b)+\tilde{W}_{i+1}(c)\right)+\widetilde{\text{mult}}_i(a,b,c)\left(\tilde{W}_{i+1}(b)\cdot\tilde{W}_{i+1}(c)\right)$$
$$=\widetilde{W}_{i+1}(\text{in}_1(a))+\widetilde{W}_{i+1}(\text{in}_2(a))=W_{i+1}(\text{in}_1(a))$$
$$+W_{i+1}(\text{in}_2(a))=W_i(a)=\tilde{W}_i(a).$$

□

Remark 4.3. Lemma 4.7 is actually valid using any extensions of add_i and mult_i that are multilinear in the first k_i variables.

Remark 4.4. Goldwasser *et al.* [135] use a slightly more complicated expression for $\tilde{W}_i(a)$ than the one in Lemma 4.7. Their expression allowed them to use even more general extensions of add_i and mult_i. In particular, their extensions do not have to be multilinear in the first k_i variables.

However, the use of the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ turns out to be critical to achieving a prover runtime that is nearly *linear* in the circuit size S, rather than a much larger polynomial in S as achieved by [135] (cf. Section 4.6.5 for details).

Therefore, in order to check the prover's claim about $\widetilde{W}_i(r_i)$, the verifier applies the sum-check protocol to the polynomial

$$f_{r_i}^{(i)}(b,c)=\widetilde{\text{add}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b)+\tilde{W}_{i+1}(c)\right)$$
$$+\widetilde{\text{mult}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b)\cdot\tilde{W}_{i+1}(c)\right). \tag{4.18}$$

Note that *the verifier does not know the polynomial* $\tilde{W}_{i+1}$ (as this polynomial is defined in terms of gate values at layer $i+1$ of the circuit, and unless $i+1$ is the input layer, the verifier does not have direct access to the values of these gates), and hence the verifier does not actually know the polynomial $f_{r_i}^{(i)}$ that it is applying the sum-check protocol to. Nonetheless, it is possible for the verifier to apply the sum-check protocol to $f_{r_i}^{(i)}$ because, until the final round, the sum-check protocol does not require the verifier to know anything about the polynomial other than its degree in each variable (see Remark 4.2). However, there

remains the issue that $\mathcal{V}$ can only execute the final check in the sum-check protocol if she can evaluate the polynomial $f_{r_i}^{(i)}$ at a random point. This is handled as follows.

Let us denote the random point at which $\mathcal{V}$ must evaluate $f_{r_i}^{(i)}$ by (b^*, c^*), where $b^* \in \mathbb{F}^{k_{i+1}}$ is the first k_{i+1} entries and $c^* \in \mathbb{F}^{k_{i+1}}$ the last k_{i+1} entries. Note that b^*, and c^* may have non-Boolean entries. Evaluating $f_{r_i}^{(i)}(b^*, c^*)$ requires evaluating $\widetilde{\text{add}}_i(r_i, b^*, c^*)$, $\widetilde{\text{mult}}_i(r_i, b^*, c^*)$, $\widetilde{W}_{i+1}(b^*)$, and $\widetilde{W}_{i+1}(c^*)$.

For many circuits, particularly those whose wiring pattern displays repeated structure, $\mathcal{V}$ can evaluate $\widetilde{\text{add}}_i(r_i, b^*, c^*)$ and $\widetilde{\text{mult}}_i(r_i, b^*, c^*)$ on her own in $O(k_i + k_{i+1})$ time as well. For now, assume that $\mathcal{V}$ can indeed perform this evaluation in $\text{poly}(k_i, k_{i+1})$ time, but this issue will be discussed further in Section 4.6.6.

$\mathcal{V}$ cannot however evaluate $\widetilde{W}_{i+1}(b^*)$, and $\widetilde{W}_{i+1}(c^*)$ on her own without evaluating the circuit. Instead, $\mathcal{V}$ asks $\mathcal{P}$ to simply provide these two values, say, z_1 and z_2, and uses iteration $i+1$ to *verify* that these values are as claimed. However, one complication remains: the precondition for iteration $i+1$ is that $\mathcal{P}$ claims a value for $\widetilde{W}_{i+1}(r_{i+1})$ for a single point $r_{i+1} \in \mathbb{F}^{k_{i+1}}$. So $\mathcal{V}$ needs to reduce verifying both $\widetilde{W}_{i+1}(b^*) = z_1$ and $\widetilde{W}_{i+1}(c^*) = z_2$ to verifying $\widetilde{W}_{i+1}(r_{i+1})$ at a single point $r_{i+1} \in \mathbb{F}^{k_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to accept the claimed values of $\widetilde{W}_{i+1}(b^*)$ and $\widetilde{W}_{i+1}(c^*)$ as long as the value of $\widetilde{W}_{i+1}(r_{i+1})$ is as claimed. As per Section 4.5.2 this is done as follows.

Reducing to Verification of a Single Point. Let $\ell \colon \mathbb{F} \to \mathbb{F}^{k_{i+1}}$ be the unique line such that $\ell(0) = b^*$ and $\ell(1) = c^*$. $\mathcal{P}$ sends a univariate polynomial q of degree at most k_{i+1} that is claimed to be $\widetilde{W}_{i+1} \circ \ell$, the restriction of $\widetilde{W}_{i+1}$ to the line ℓ. $\mathcal{V}$ checks that $q(0) = z_1$ and $q(1) = z_2$ (rejecting if this is not the case), picks a random point $r^* \in \mathbb{F}$, and asks $\mathcal{P}$ to prove that $\widetilde{W}_{i+1}(\ell(r^*)) = q(r^*)$. By Claim 4.6, as long as $\mathcal{V}$ is convinced that $\widetilde{W}_{i+1}(\ell(r^*)) = q(r^*)$, it is safe for $\mathcal{V}$ to believe that q does in fact equal $\widetilde{W}_{i+1} \circ \ell$, and hence that $\widetilde{W}_{i+1}(b^*) = z_1$ and $\widetilde{W}_{i+1}(c^*) = z_2$ as claimed by $\mathcal{P}$. See Section 4.5.2 for a picture and example of this sub-protocol.

This completes iteration i; $\mathcal{P}$ and $\mathcal{V}$ then move on to the iteration for layer $i+1$ of the circuit, whose purpose is to verify that $\widetilde{W}_{i+1}(r_{i+1})$ has the claimed value, where $r_{i+1} \coloneqq \ell(r^*)$.

The Final Iteration. At the final iteration d, $\mathcal{V}$ must evaluate $\widetilde{W}_d(r_d)$ on her own. But the vector of gate values at layer d of $\mathcal{C}$ is simply the input x to $\mathcal{C}$. By Lemma 3.8, $\mathcal{V}$ can compute $\widetilde{W}_d(r_d)$ on her own in $O(n)$ time, where recall that n is the size of the input x to $\mathcal{C}$.

A self-contained description of the GKR protocol is provided in Figure 4.13.

4.6.5 Discussion of Costs and Soundness

$\mathcal{V}$'s Runtime. Observe that the polynomial $f_{r_i}^{(i)}$ defined in Equation (4.18) is a $(2k_{i+1})$-variate polynomial of degree at most 2 in each variable, and so the invocation of the sum-check protocol at iteration i requires $2k_{i+1}$ rounds, with three field elements transmitted per round. Thus, the total communication cost is $O(S_0 + d \log S)$ field elements where S_0 is the number of outputs of the circuit. The time cost to $\mathcal{V}$ is $O(n + d \log S + t + S_0)$, where t is the amount of time required for $\mathcal{V}$ to evaluate $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at a random input, for each layer i of $\mathcal{C}$. Here the n term is due to the time required to evaluate $\widetilde{W}_d(r_d)$, the S_0 term is the time required to read the vector of claimed outputs and evaluate the corresponding multilinear extension, the $d \log S$ term is the time required for $\mathcal{V}$ to send messages to $\mathcal{P}$ and process and check the messages from $\mathcal{P}$. For now, let us assume that t is a low-order cost and that $S_0 = 1$, so that $\mathcal{V}$ runs in total time $O(n + d \log S)$; we discuss this issue further in Section 4.6.6.

$\mathcal{P}$'s Runtime. Analogously to the MatMult protocol of Section 4.4, we give two increasingly sophisticated implementations of the prover when the sum-check protocol is applied to the polynomial $f_{r_i}^{(i)}$.
Method 1: $f_{r_i}^{(i)}$ is a v-variate polynomial for $v = 2k_{i+1}$. As in the analysis of Method 1 for implementing the prover in the matrix multiplication protocol from Section 4.4, $\mathcal{P}$ can compute the prescribed method in round j by evaluating $f_{r_i}^{(i)}$ at $3 \cdot 2^{v-j}$ points. It is not hard to see that

Description of the GKR protocol, when applied to a layered arithmetic circuit $\mathcal{C}$ of depth d and fan-in two on input $x \in \mathbb{F}^n$. Throughout, k_i denotes $\log_2(S_i)$ where S_i is the number of gates at layer i of $\mathcal{C}$.

- At the start of the protocol, $\mathcal{P}$ sends a function $D\colon \{0,1\}^{k_0} \to \mathbb{F}$ claimed to equal W_0 (the function mapping output gate labels to output values).
- $\mathcal{V}$ picks a random $r_0 \in \mathbb{F}^{k_0}$ and lets $m_0 \leftarrow \tilde{D}(r_0)$. The remainder of the protocol is devoted to confirming that $m_0 = \tilde{W}_0(r_0)$.
- **For** $i = 0, 1, \ldots, d-1$:
  - Define the $(2k_{i+1})$-variate polynomial
$$f^{(i)}_{r_i}(b,c) := \widetilde{\text{add}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b) + \tilde{W}_{i+1}(c)\right) + \widetilde{\text{mult}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b)\cdot\tilde{W}_{i+1}(c)\right).$$
  - $\mathcal{P}$ claims that $\sum_{b,c\in\{0,1\}^{k_{i+1}}} f^{(i)}_{r_i}(b,c) = m_i$.
  - So that $\mathcal{V}$ may check this claim, $\mathcal{P}$ and $\mathcal{V}$ apply the sum-check protocol to $f^{(i)}_{r_i}$, up until $\mathcal{V}$'s final check in that protocol, when $\mathcal{V}$ must evaluate $f^{(i)}_{r_i}$ at a randomly chosen point $(b^*, c^*) \in \mathbb{F}^{k_{i+1}} \times \mathbb{F}^{k_{i+1}}$. See Remark (a) at the end of this codebox.
  - Let ℓ be the unique line satisfying $\ell(0) = b^*$ and $\ell(1) = c^*$. $\mathcal{P}$ sends a univariate polynomial q of degree at most k_{i+1} to $\mathcal{V}$, claimed to equal $\tilde{W}_{i+1}$ restricted to ℓ.
  - $\mathcal{V}$ now performs the final check in the sum-check protocol, using $q(0)$ and $q(1)$ in place of $\tilde{W}_{i+1}(b^*)$ and $\tilde{W}_{i+1}(c^*)$. See Remark (b) at the end of this codebox.
  - $\mathcal{V}$ chooses $r^* \in \mathbb{F}$ at random and sets $r_{i+1} = \ell(r^*)$ and $m_{i+1} \leftarrow q(r_{i+1})$.
- $\mathcal{V}$ checks directly that $m_d = \widetilde{W}_d(r_d)$ using Lemma 3.8.

  Note that $\widetilde{W}_d$ is simply $\widetilde{x}$, the multilinear extension of the input x when x is interpreted as the evaluation table of a function mapping $\{0,1\}^{\log n} \to \mathbb{F}$.

Remark a. Note that $\mathcal{V}$ does not actually know the polynomial $f^{(i)}_{r_i}$, because $\mathcal{V}$ does not know the polynomial $\tilde{W}_{i+1}$ that appears in the definition of $f^{(i)}_{r_i}$. However, the sum-check protocol does not require $\mathcal{V}$ to know anything about the polynomial to which it is being applied, until the very final check in the protocol (see Remark 4.2).

Remark b. We assume here that for each layer i of $\mathcal{C}$, $\mathcal{V}$ can evaluate the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at the point (r_i, b^*, c^*) in polylogarithmic time. Hence, given $\tilde{W}_{i+1}(b^*)$ and $\tilde{W}_{i+1}(c^*)$, $\mathcal{V}$ can quickly evaluate $f^{(i)}_{r_i}(b^*, c^*)$ and thereby perform its final check in the sum-check protocol applied to $f^{(i)}_{r_i}$.

Figure 4.13: Self-contained description of the GKR protocol for arithmetic circuit evaluation.

$\mathcal{P}$ can evaluate $f_{r_i}^{(i)}$ at any point in $O(S_i + S_{i+1})$ time using techniques similar to Lemma 3.8. This yields a runtime for $\mathcal{P}$ of $O\left(2^v \cdot (S_i + S_{i+1})\right)$. Over all d layers of the circuit, $\mathcal{P}$'s runtime is bounded by $O(S^3)$.

Method 2: Cormode *et al.* [102] improved on the $O(S^3)$ runtime of Method 1 by observing, just as in the matrix multiplication protocol from Section 4.4, that the $3 \cdot 2^{v-j}$ points at which $\mathcal{P}$ must evaluate $f_{r_i}^{(i)}$ in round j of the sum-check protocol are highly structured, in the sense that their trailing entries are Boolean. That is, it suffices for $\mathcal{P}$ to evaluate $f_{r_i}^{(i)}(z)$ for all points z of the form: $z = (r_1, \ldots, r_{j-1}, \{0,1,2\}, b_{j+1}, \ldots, b_v)$, where $v = 2k_{i+1}$ and each $b_k \in \{0,1\}$.

For each such point z, the bottleneck in evaluating $f_{r_i}^{(i)}(z)$ is in evaluating $\widetilde{\text{add}}_i(z)$ and $\widetilde{\text{mult}}_i(z)$. A direct application of Lemma 3.8 implies that each such evaluation can be performed in $2^v = O(S_{i+1}^2)$ time. However, we can do much better by observing that the functions add_i and mult_i are *sparse*, in the sense that $\text{add}_i(a,b,c) = \text{mult}_i(a,b,c) = 0$ for all Boolean vectors $(a,b,c) \in \mathbb{F}^v$ except for the S_i vectors of the form $(a, \text{in}_{1,i}(a), \text{in}_{2,i}(a))$: $a \in \{0,1\}^{k_i}$.

Thus, by Lagrange Interpolation (Lemma 3.6), we can write $\widetilde{\text{add}}_i(z) = \sum_{a \in \{0,1\}^{k_i}} \chi_{(a,\text{in}_{1,i}(a),\text{in}_{2,i}(a))}(z)$, where the sum is only over addition gates a at layer i of $\mathcal{C}$, and similarly for $\widetilde{\text{mult}}_i(z)$ (recall that the multilinear Lagrange basis polynomial $\chi_{(a,\text{in}_{1,i}(a),\text{in}_{2,i}(a))}$ was defined in Equation (3.2) of Lemma 3.6). Just as in the analysis of Method 2 for implementing the prover in the matrix multiplication protocol of Section 4.4, for any input z of the form $z = (r_1, \ldots, r_{j-1}, \{0,1,2\}, b_{j+1}, \ldots, b_v)$, it holds that $\chi_{(a,\text{in}_{1,i}(a),\text{in}_{2,i}(a))}(z) = 0$ unless the last $v - j$ entries of z and $(a, \text{in}_{1,i}(a), \text{in}_{2,i}(a))$ are equal (here, we are exploiting the fact that the trailing entries of z are Boolean). Hence, $\mathcal{P}$ can evaluate $\widetilde{\text{add}}_i(z)$ at all the necessary points z in each round of the sum-check protocol with a single pass over the gates at layer i of $\mathcal{C}$: for each gate a in layer i, $\mathcal{P}$ only needs to update $\widetilde{\text{add}}_i(z) \leftarrow \widetilde{\text{add}}_i(z) + \chi_{(a,\text{in}_{1,i}(a),\text{in}_{2,i}(a))}(z)$ for the three values of z whose trailing $v - j$ entries equal the trailing entries of $(a, \text{in}_{1,i}(a), \text{in}_{2,i}(a))$.

Round Complexity and Communication Cost. By direct inspection of the protocol description, there are $O(d \log S)$ rounds in the GKR protocol, and the total communication cost is $O(d \log S)$ field elements.

Soundness Error. The soundness error of the GKR protocol is $O(d \log(S)/|\mathbb{F}|)$. The idea of the soundness analysis is that, if the prover begins the protocol with a false claim as to the output value(s) $\mathcal{C}(x)$, then for the verifier to be convinced to accept, there must be at least one round j of the interactive proof in which the following occurs. The prover sends a univariate polynomial g_j that differs from the prescribed polynomial s_j that the honest prover would have sent in that round, yet $g_j(r_j) = s_j(r_j)$, where r_j is a random field element chosen by the verifier in round j. For rounds j of the GKR protocol corresponding to a round within an invocation of the sum-check protocol, g_j and s_j are polynomials of degree $O(1)$, and hence if $g_j \neq s_j$ then the probability (over the random choice of r_j) that $g_j(r_j) = s_j(r_j)$ is at most $O(1/|\mathbb{F}|)$.

In rounds j of the GKR protocol corresponding to the "reducing to verification of a single point" technique, g_j and s_j have degree at most $O(\log S)$, and hence if $g_j \neq s_j$, the probability that $g_j(r_j) = s_j(r_j)$ is at most $O(\log(S)/|\mathbb{F}|)$. Note that there are at most d such rounds over the course of the entire protocol, since this technique is applied at most once per layer of $\mathcal{C}$.

By applying a union bound over all rounds in the protocol, we conclude that the probability there is *any* round j such that $g_j \neq s_j$ yet $g_j(r_j) = s_j(r_j)$ is at most $O(d \log(S)/|\mathbb{F}|)$.

Additional Intuition and Discussion of Soundness. In summary, the GKR protocol prover begins by sending the claimed values of the output gates, thereby specifying the vector of output values W_0, and the verifier evaluates $\tilde{W}_0$ at a random point. Similarly, at the end of the ith iteration of the protocol, the prover is forced to make a claim about a single randomly chosen evaluation of $\tilde{W}_i$. In this way, the prover gradually transitions from making a claim about (one evaluation of the multilinear extension of) the output layer to an analogous claim about the input layer, which the verifier can check directly in linear time.

A common source of confusion is to suspect that "checking the prover's claim" about a random evaluation of $\tilde{W}_i$ is the same as *selecting a random gate* at layer i at confirming that the prover evaluated that one gate correctly (e.g., if the gate is a multiplication gate, checking that the prover indeed assigned a value to the selected gate that is equal to the product of the values assigned to the gate's inputs). If this interpretation were accurate, the protocol would not be sound, because a cheating prover that "alters" the value of a single gate in the circuit would only be caught by the verifier if that gate happens to be the one selected at random from its layer.

The above interpretation is inaccurate: these two processes would only be equivalent if each entry of r_i were chosen at random from $\{0, 1\}$, rather than at random from the entire field $\mathbb{F}$.

Indeed, if even a *single* gate value of layer i is corrupted, then by the Schwartz-Zippel lemma, *almost all* evaluations of $\tilde{W}_i$ must change.[25] By "spot-checking" the *multilinear extension encoding* of the gate values of each layer of the circuit, the GKR verifier is able to detect even tiny deviations of the prover from correct gate-by-gate evaluation of the circuit. See Figure 4.14 for a depiction.

4.6.6 Evaluating $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ Efficiently

The issue of the verifier efficiently evaluating $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at a random point $\omega \in \mathbb{F}^{k_i+2k_{i+1}}$ is a tricky one. While there does not seem to be a clean characterization of precisely which circuits have $\widetilde{\text{add}}_i$'s and $\widetilde{\text{mult}}_i$'s that can be evaluated in $O(\log S)$ time, most circuits that exhibit any kind of repeated structure satisfy this property. In particular, the papers [102], [237] show that the evaluation can be computed in $O(k_i + k_{i+1}) = O(\log S)$ time for a variety of common wiring patterns and specific circuits. This includes specific circuits computing functions such as MatMult, pattern matching, Fast Fourier Transforms, and various problems of interest in the streaming literature, like frequency moments and distinct elements (see Exercise 4.4). In a similar vein, Holmgren and Rothblum [152, Section 5.1] show that as long as add_i

[25] So long as the field size is significantly larger than the logarithm of the number of gates at layer i of the circuit.

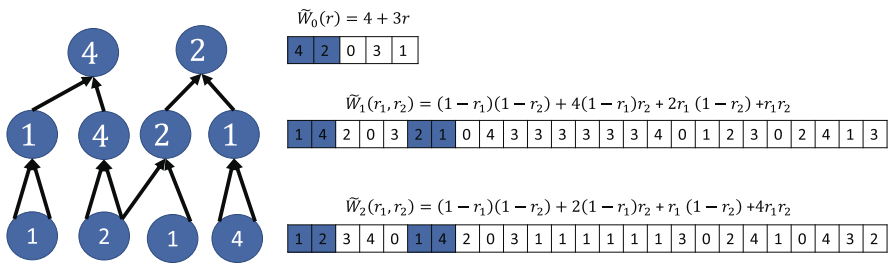

Figure 4.14: Depiction of a circuit over $\mathbb{F}_5$ consisting entirely of multiplication gates, and the multilinear extension encodings $\tilde{W}_i$ of each layer i when the circuit is evaluated on the length-4 input $(1, 2, 1, 4)$ (see Figure 3.2). Due to there being two outputs, $\tilde{W}_0$ is a univariate polynomial, and hence its evaluation table consists of $|\mathbb{F}_5| = 5$ values. The other two layers have four gates each, and hence $\tilde{W}_1$ and $\tilde{W}_2$ are bivariate polynomials, the evaluations tables of which each contain $5^2 = 25$ values, indexed from $(0, 0)$ to $(4, 4)$. Entries of the multilinear extension encodings indexed by Boolean vectors are highlighted in blue. In the GKR protocol applied to this circuit on this input, the prover begins by sending the claimed values of the two output gates, thereby specifying W_0, and the verifier evaluates $\tilde{W}_0$ at a random point. Then at the end of each iteration i of the for loop in Figure 4.13, the prover is forced to make a claim about a single (randomly chosen) evaluation of $\tilde{W}_i$.

and mult_i are computable within a computational model called read-once branching programs, then $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated at any desired point in logarithmic time, and observe that this condition indeed captures common wiring patterns. Moreover, we will see in Section 4.6.7 that $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated efficiently for any circuit that operates in a *data parallel* manner.

In addition, various suggestions have been put forth for what to do when $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ cannot be evaluated in time $O(\log S)$. For example, as observed by Cormode *et al.* [102], these computations can always be done by $\mathcal{V}$ in $O(\log S)$ *space* as long as the circuit is log-space uniform. This is sufficient in streaming applications where the space usage of the verifier is paramount [102]. Moreover, these computations can be done offline before the input is even observed, because they only depend on the wiring of the circuit, and not on the input [102], [135].

An additional proposal appeared in [135], where Goldwasser *et al.* considered the option of outsourcing the computation of $\widetilde{\text{add}}_i(r_i, b^*, c^*)$ and $\widetilde{\text{mult}}_i(r_i, b^*, c^*)$ themselves. In fact, this option plays a central role

in obtaining their result for general log-space uniform circuits. Specifically, GKR's results for general log-space uniform circuits are obtained via a two-stage protocol. First, they give a protocol for any problem computable in (non-deterministic) logarithmic space by applying their protocol to the canonical circuit for simulating a space-bounded Turing machine. This circuit has a highly regular wiring pattern for which $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated in $O(\log S)$ time.[26] For a general log-space uniform circuit $\mathcal{C}$, it is not known how to identify low-degree extensions of add_i and mult_i that can be evaluated at ω in polylogarithmic time. Rather, Goldwasser *et al.* outsource computation of $\widetilde{\text{add}}_i(r_i, b^*, c^*)$ and $\widetilde{\text{mult}}_i(r_i, b^*, c^*)$ themselves. Since $\mathcal{C}$ is log-space uniform, $\widetilde{\text{add}}_i(r_i, b^*, c^*)$ and $\widetilde{\text{mult}}_i(r_i, b^*, c^*)$ can be computed in logarithmic space, and the protocol for logspace computations applies directly.

A closely related proposal to deal with the circuits for which $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ cannot be evaluated in time sublinear in the circuit size S leverages cryptography. Specifically, later in this monograph we introduce a cryptographic primitive called a *polynomial commitment scheme* and explain how this primitive can be used to achieve the following. A trusted party (e.g., the verifier itself) can spend $O(S)$ time in pre-processing and produce a short *cryptographic commitment* to the polynomials $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ for all layers i of $\mathcal{C}$. After this pre-processing stage, the verifier $\mathcal{V}$ can apply the IP of this section to evaluate $\mathcal{C}$ on many different inputs, and $\mathcal{V}$ can use the cryptographic commitment to force the prover to accurately evaluate $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ on its behalf. Due to its use of cryptography, this proposal results in an argument system as opposed to an interactive proof. Argument systems that handle pre-processing in this manner are sometimes called *holographic*, or referred to as using *computation commitments*. See Sections 10.3.2 and 16.2 for details.

[26] In [135], Goldwasser *et al.* actually use higher degree extensions of add_i and mult_i obtained by arithemetizing a Boolean formula of size polylog(S) computing these functions (see Remark 4.4). The use of these extensions results in a prover whose runtime is a large polynomial in S (i.e., $O(S^4)$). Cormode *et al.* [102] observe that in fact the multilinear extensions of add_i and mult_i can be used for this circuit, and that with these extensions the prover's runtime can be brought down to $O(S \log S)$.

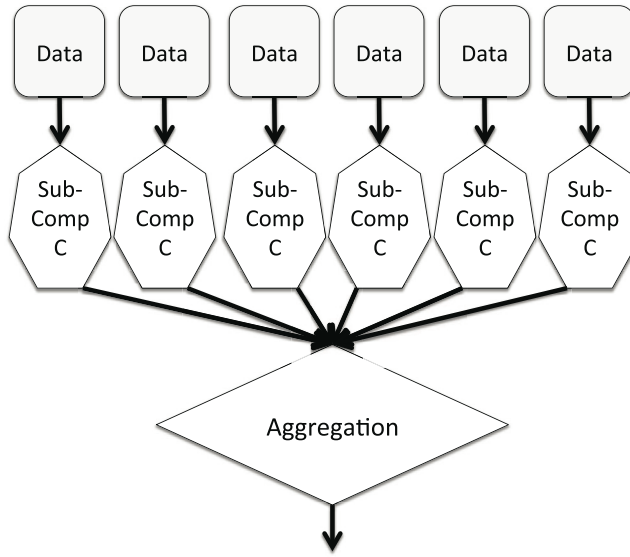

Figure 4.15: Schematic of a data parallel computation.

4.6.7 Leveraging Data Parallelism for Further Speedups

Data parallel computation refers to any setting in which the same sub-computation is applied independently to many pieces of data, before possibly aggregating the results. The protocol of this section makes no assumptions on the sub-computation that is being applied. In particular, it handles sub-computations computed by circuits with highly irregular wiring patterns, but does assume that the sub-computation is applied independently to many pieces of data. Figure 4.15 gives a schematic of a data parallel computation.

Data parallel computation is pervasive in real-world computing. For example, consider any *counting query* on a database. In a counting query, one applies some function independently to each row of the database and sums the results. For example, one may ask "How many people in the database satisfy Property P?" The protocol below allows one to verifiably outsource such a counting query with overhead that depends minimally on the size of the database, but that necessarily depends on the complexity of the property P. In Section 6.5, we will see that data

parallel computations are in some sense "universal", in that efficient transformations from high-level computer programs to circuits often yield data parallel circuits.

The Protocol and its Costs. Let C be a circuit of size S with an arbitrary wiring pattern, and let C' be a "super-circuit" that applies C independently to $B = 2^b$ different inputs before aggregating the results in some fashion. For example, in the case of a counting query, the aggregation phase simply sums the results of the data parallel phase. Assume that the aggregation step is sufficiently simple that the aggregation itself can be verified using the techniques of Section 4.6.5.

If one naively applies the GKR protocol to the super-circuit C', $\mathcal{V}$ might have to perform an expensive pre-processing phase to evaluate the wiring predicates $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ of C' at the necessary locations—this would require time $\Omega(B \cdot S)$. Moreover, when applying the basic GKR protocol to C' using the techniques of [102], $\mathcal{P}$ would require time $\Theta(B \cdot S \cdot \log(B \cdot S))$. A different approach was taken by Vu *et al.* [241], who applied the GKR protocol B independent times, once for each copy of C. This causes both the communication cost and $\mathcal{V}$'s online check time to grow linearly with B, the number of sub-computations, which is undesirable.

In contrast, the protocol of this section (due to [242], building on [237]) achieves the best of both worlds, in that the overheads for the prover and verifier have no dependence on the number of inputs B to which C is applied. More specifically, the preprocessing time of the verifier is at most $O(S)$, independent of B. The prover runs in time $O(BS + S \log S)$. Observe that as long as $B > \log S$ (i.e., there is a sufficient amount of data parallelism in the computation), $O(BS + S \log S) = O(B \cdot S)$, and hence the prover is only a constant factor slower than the time required to evaluate the circuit gate-by-gate with no guarantee of correctness.

The idea of the protocol is that although each sub-computation C can have a complicated wiring pattern, the circuit is maximally regular between sub-computations, as the sub-computations do not interact at all. It is possible to leverage this regularity to minimize the

pre-processing time of the verifier, and to significantly speed up the prover.

4.6.7.1 Protocol Details

Let $\mathcal{C}$ be an arithmetic circuit over $\mathbb{F}$ of depth d and size S with an arbitrary wiring pattern, and let $\mathcal{C}'$ be the circuit of depth d and size $B \cdot S$ obtained by laying B copies of C side-by-side, where $B = 2^b$ is a power of 2. We will use the same notation as in Section 4.6.4, using apostrophes to denote quantities referring to $\mathcal{C}'$. For example, layer i of $\mathcal{C}$ has size $S_i = 2^{k_i}$ and gate values specified by the function W_i, while layer i of $\mathcal{C}'$ has size $S'_i = 2^{k'_i} = 2^{b+k_i}$ and gate values specified by W'_i.

Consider layer i of C'. Let $a = (a_1, a_2) \in \{0,1\}^{k_i} \times \{0,1\}^b$ be the label of a gate at layer i of $\mathcal{C}'$, where a_2 specifies which "copy" of C the gate is in, while a_1 designates the label of the gate within the copy. Similarly, let $b = (b_1, b_2) \in \{0,1\}^{k_{i+1}} \times \{0,1\}^b$ and $c = (c_1, c_2) \in \{0,1\}^{k_{i+1}} \times \{0,1\}^b$ be the labels of two gates at layer $i+1$. The key to achieving the speedups for data parallel circuits relative to the interactive proof described in Section 4.6.4 is to tweak the expression in Lemma 4.7 for $\tilde{W}_i$. Specifically, Lemma 4.7 represents $\tilde{W}'_i(z)$ as a sum over $\left(S'_{i+1}\right)^2$ terms. In this section, we leverage the data parallel structure of C' to represent $\tilde{W}'_i(z)$ as a sum over $S'_{i+1} \cdot S_{i+1}$ terms, which is smaller than $\left(S'_{i+1}\right)^2$ by a factor of B.

Lemma 4.8. Let h denote the polynomial $\mathbb{F}^{k_i \times b} \to \mathbb{F}$ defined via

$$h(a_1, a_2) := \sum_{b_1, c_1 \in \{0,1\}^{k_{i+1}}} g(a_1, a_2, b_1, c_1),$$

where

$$g(a_1, a_2, b_1, c_1) := \widetilde{\text{add}}_i(a_1, b_1, c_1) \left(\widetilde{W}'_{i+1}(b_1, a_2) + \widetilde{W}'_{i+1}(c_1, a_2)\right) + \widetilde{\text{mult}}_i(a_1, b_1, c_1) \cdot \widetilde{W}'_{i+1}(b_1, a_2) \cdot \widetilde{W}'_{i+1}(c_1, a_2).$$

Then h extends W'_i.

Proof Sketch. Essentially, Lemma 4.8 says that an addition (respectively, multiplication) gate $a = (a_1, a_2) \in \{0,1\}^{k_i+b}$ of $\mathcal{C}'$ is connected to gates

$b = (b_1, b_2) \in \{0,1\}^{k_{i+1}+b}$ and $c = (c_1, c_2) \in \{0,1\}^{k_{i+1}+b}$ of $\mathcal{C}'$ if and only if a, b, and c are all in the same copy of $\mathcal{C}$, and a is connected to b and c within the copy. □

The following lemma requires some additional notation. Let $\beta_{k_i'}(a,b) : \{0,1\}^{k_i'} \times \{0,1\}^{k_i'} \to \{0,1\}$ be the function that evaluates to 1 if $a = b$, and evaluates to 0 otherwise, and define the formal polynomial

$$\widetilde{\beta}_{k_i'}(a,b) = \prod_{j=1}^{k_i'} \left((1-a_j)(1-b_j) + a_j b_j\right). \tag{4.19}$$

It is straightforward to check that $\widetilde{\beta}_{k_i'}$ is the multilinear extension $\beta_{k_i'}$. Indeed, $\widetilde{\beta}_{k_i'}$ is a multilinear polynomial. And for $a, b \in \{0,1\}^{k_i'}$, it is easy to check that $\widetilde{\beta}_{k_i'}(a,b) = 1$ if and only if a and b are equal coordinate-wise.

Lemma 4.9. (Restatement of [216, Lemma 3.2.1].) For *any* polynomial $h\colon \mathbb{F}^{k_i'} \to \mathbb{F}$ extending W_i', the following polynomial identity holds:

$$\tilde{W}_i'(z) = \sum_{a \in \{0,1\}^{k_i'}} \widetilde{\beta}_{k_i'}(z,a) h(a). \tag{4.20}$$

Proof. It is easy to check that the right hand side of Equation (4.20) is a multilinear polynomial in z, and that it agrees with W_i' on all Boolean inputs. Thus, the right hand side of Equation (4.20), viewed as a polynomial in z, must be the (unique) multilinear extension $\tilde{W}_i'$ of W_i'. □

Intuitively, Lemma 4.9 achieves "multi-linearization" of the higher-degree extension h. That is, it expresses the *multilinear* extension of any function W_i' in terms of *any* extension h of W_i', regardless of the degree of h.

Combining Lemmas 4.8 and 4.9 implies that for any $z \in \mathbb{F}^{k_i'}$,

$$\tilde{W}_i'(z) = \sum_{(a_1,a_2,b_1,c_1) \in \{0,1\}^{k_i+b+2k_{i+1}}} g_z^{(i)}(a_1,a_2,b_1,c_1), \tag{4.21}$$

where

$$g_z^{(i)}(a_1, a_2, b_1, c_1) := \widetilde{\beta}_{k_i'}(z, (a_1, a_2)) \cdot \Big[\widetilde{\text{add}}_i(a_1, b_1, c_1) \times \Big(\widetilde{W}'_{i+1}(b_1, a_2) + \widetilde{W}'_{i+1}(c_1, a_2)\Big) + \widetilde{\text{mult}}_i(a_1, b_1, c_1) \cdot \widetilde{W}'_{i+1}(b_1, a_2) \cdot \widetilde{W}'_{i+1}(c_1, a_2)\Big].$$

Thus, to reduce a claim about $\tilde{W}'_i(r_i)$ to a claim about $\tilde{W}'_{i+1}(r_{i+1})$ for some point $r_{i+1} \in \mathbb{F}^{k'_{i+1}}$, it suffices to apply the sum-check protocol to the polynomial $g_{r_i}^{(i)}$, and then use the "Reducing to Verification of a Single Point" protocol from Section 4.5.2. That is, the protocol is the same as in Section 4.6.4, except that, at layer i, rather than applying the sum-check protocol to the polynomial $f_{r_i}^{(i)}$ defined in Equation (4.18) to compute $\tilde{W}'_i(r_i)$, the protocol instead applies the sum-check protocol to the polynomial $g_{r_i}^{(i)}$ (Equation (4.21)).

Costs for $\mathcal{V}$. To bound $\mathcal{V}$'s runtime, observe that $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated at a random point in $\mathbb{F}^{k_i+2k_{i+1}}$ in pre-processing in time $O(S_i)$ by enumerating the in-neighbors of each of the S_i gates at layer i in order to apply Lemma 3.8. Adding up the pre-processing time across all iterations i of our protocol, $\mathcal{V}$'s pre-processing time is $O(\sum_i S_i) = O(S)$ as claimed. Notice this pre-processing time is independent of B, the number of copies of the subcircuit.

Outside of pre-processing, the costs to the verifier are similar to Section 4.6.5, with the main difference being that now the verifier needs to also evaluate $\widetilde{\beta}_{k_i}$ at a random point at each layer i. But the verifier can evaluate $\widetilde{\beta}_{k_i}$ at any input with $O(\log S_i)$ additions and multiplications over $\mathbb{F}$, using Equation (4.19). This does not affect the verifier's asymptotic runtime.

Costs for $\mathcal{P}$. The insights that go into implementing the honest prover in time $O(B \cdot S + S \log S)$ build on ideas related the Method 3 for implementing the prover in the Matrix Multiplication protocol of Section 4.4, and heavily exploit the fact that Equation (4.21) represents $\tilde{W}'_i(z)$ as a sum over just $S'_{i+1} \cdot S_{i+1}$ terms, rather than the $(S'_{i+1})^2$ terms in the sum that would be obtained by applying Equation (4.17) to $\mathcal{C}'$. The costs of the protocol are summarized in Table 4.5.

Table 4.5: Costs of the IP of Section 4.6.7 when applied to any log-space uniform arithmetic circuit $\mathcal{C}$ of size S and depth d over n variables, that is applied B times in a data parallel manner (cf. Figure 4.15).

Communication	**Rounds**	$\mathcal{V}$ **Time**	$\mathcal{P}$ **Time**
$O(d \cdot \log (B \cdot S))$ field elements	$O(d \cdot (\log (B \cdot S)))$	online time: $O(B \cdot n + d \cdot (\log(B \cdot S)))$ pre-processing time: $O(S)$	$O(B \cdot S + S \cdot \log(S))$

Remark 4.5. Recent work [251] has shown how to use Lemma 4.5 to implement the prover in the IP of Section 4.6.4 in time $O(S)$ for *arbitrary* arithmetic circuits of size S (not just circuits with a sufficient amount of data parallelism as in Section 4.6.7).[27] For brevity, we do not elaborate here upon how to achieve this result. The same result in fact follows (with some adaptation) from Section 8.4 in Section 8, where we explain how to achieve an $O(S)$-time prover in a (two-prover) interactive proof for a *generalization* of arithmetic circuits, called *rank-one constraint systems* (R1CS).

4.6.8 Tension Between Efficiency and Generality

The GKR protocol and its variants covered in this section is an example of a *general-purpose* technique for designing VC protocols. Specifically, the GKR protocol can be used to verifiably outsource the evaluation of an arbitrary arithmetic circuit, and as we will see in the next section, arbitrary computer programs can be turned into arithmetic circuits. Such general-purpose techniques are the primary focus of this survey.

However, there is often a tension between the generality and efficiency of VC protocols. That is, the general-purpose techniques should sometimes be viewed as heavy hammers that are capable of pounding arbitrary nails, but are not necessarily the most efficient way of hammering any particular nail.

[27] To clarify, this does not address the issue discussed in Section 4.6.6 that for arbitrary arithmetic circuits, the verifier may need time linear in the circuit size S to evaluate $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ as required by the protocol.

This point was already raised in Section 4.4.1 in the context of matrix multiplication (see the paragraph "Preview: Other Protocols for Matrix Multiplication"). That section described an interactive proof for matrix multiplication that is far more concretely efficient, especially in terms of prover time and communication cost, than applying the GKR protocol to any known arithmetic circuit computing matrix multiplication. As another example, the circuit depicted in Figures 4.8–4.11 computes the sum of the squared entries of the input in $\mathbb{F}^n$. This is an important function in the literature on streaming algorithms, called the second frequency moment. Applying the GKR protocol to this circuit (which has logarithmic depth and size $O(n)$) would result in communication cost of $\Theta(\log^2 n)$. But the function can be computed much more directly, and with total communication $O(\log n)$, by a single application of the sum-check protocol. Specifically, if we interpret the input as specifying a function $f\colon \mathbb{F}^{\log n} \to \mathbb{F}$ in the natural way, then we can simply apply the sum-check protocol to the polynomial $\left(\tilde{f}\right)^2$, the square of the multilinear extension of f. This requires the verifier to evaluate $\left(\tilde{f}\right)^2$ at a single point r. The verifier can compute $\left(\tilde{f}\right)^2(r)$ by evaluating $\tilde{f}(r)$ in linear or quasilinear time using Lemma 3.7 or Lemma 3.8, and then squaring the result.

To summarize, while this survey is primarily focused on general-purpose VC protocols, these do not represent the most efficient solutions in all situations. Those interested in specific functionalities may be well-advised to consider whether less general but more efficient protocols apply to the functionality of interest. Even when using a general-purpose VC protocol, there are typically many optimizations a protocol designer can identify (e.g., expanding the gate set within the GKR protocol from addition and multiplication gates to other types of low-degree operations tailored to the functionality of interest, see for example [102, Section 3.2], [251, Section 5], and [30]).

4.7 Exercises

Exercise 4.1. Recall that Section 4.3 gave a doubly-efficient interactive proof for counting triangles. Given as input the adjacency matrix A

of a graph on n vertices, the IP views A as a function over domain $\{0,1\}^{\log_2 n} \times \{0,1\}^{\log_2 n}$, lets $\widetilde{A}$ denote its multlinear extension, and applies the sum-check protocol to the $(3 \log n)$-variate polynomial

$$g(X, Y, Z) = \widetilde{A}(X, Y) \cdot \widetilde{A}(Y, Z) \cdot \widetilde{A}(X, Z).$$

A 4-cycle in a graph is a quadruple of vertices (a, b, c, d) such that (a, b), (b, c), (c, d), and (a, d) are all edges in the graph. Give a doubly-efficient interactive proof that, given as input the adjacency matrix A of a simple graph, counts the number of 4-cycles in the graph.

Exercise 4.2. Here is yet another interactive proof for counting triangles given as input the adjacency matrix A of a graph on n vertices: For a sufficiently large prime p, define $f\colon \{0,1\}^{\log_2 n} \times \{0,1\}^{\log_2 n} \times \{0,1\}^{\log_2 n} \to \mathbb{F}_p$ via $f(i, j, k) = A_{i,j} \cdot A_{j,k} \cdot A_{k,i}$, where here we associate vectors in $\{0,1\}^{\log_2 n}$ with numbers in $\{1, \ldots, n\}$ in the natural way, and interpret entries of A as elements of $\mathbb{F}_p$ in the natural way. Apply the sum-check protocol to the multilinear extension $\widetilde{f}$. Explain that the protocol is complete, and has soundness error at most $(3 \log_2 n)/p$.

What are the fastest runtimes you can give for the prover and verifier in this protocol? Do you think the verifier would be interested in using this protocol?

Exercise 4.3. This question has 5 parts.

- (Part a) Section 4.2 gave a technique to take any Boolean formula $\phi\colon \{0,1\}^n \to \{0,1\}$ of size S and turn ϕ into a polynomial g over field $\mathbb{F}$ that extends ϕ (the technique represents g via an arithmetic circuit over $\mathbb{F}$ of size $O(S)$).

  Apply this technique to the Boolean formula in Figure 4.16. You may specify the resulting extension polynomial g by drawing the arithmetic circuit computing g or by writing out some other representation of g.

- (Part b) Section 4.2 gives an interactive proof for counting the number of satisfying assignments to ϕ by applying the sum-check protocol to g. For the polynomial g you derived in Part a that extends the formula in Figure 4.16, provide the messages sent

by the honest prover if the random field element chosen by the verifier in round 1 is $r_1 = 3$ and the random field element chosen by the verifier in round 2 is $r_2 = 4$. You may work over the field $\mathbb{F}_{11}$ of integers modulo 11.

- (Part c) Imagine you are a cheating prover in the protocol of Part b above and somehow you know at the start of the protocol that in round 1 the random field element r_1 chosen by the verifier will be 3. Give a sequence of messages that you can send that will convince the verifier that the number of satisfying assignments of ϕ is 6 (the verifier should be convinced regardless of the random field elements r_2 and r_3 that will be chosen by the verifier in rounds 2 and 3).

- (Part d) You may notice that the extension polynomial g derived in Part a is *not* multilinear. This problem explains that there is a good reason for this.

  Show that the ability to evaluate the *multilinear* extension $\tilde{\phi}$ of a formula ϕ at a randomly chosen point in $\mathbb{F}^n$ allows one to determine whether or not ϕ is satisfiable. That is, give an efficient randomized algorithm that, given $\tilde{\phi}(\mathbf{r})$ for a randomly chosen $\mathbf{r} \in \mathbb{F}^n$, outputs SATISFIABLE with probability at least $1 - n/|\mathbb{F}|$ over the random choice of $\mathbf{r}$ if ϕ has one or more satisfying assignments, and outputs UNSATISFIABLE with probability 1 if ϕ has no satisfying assignments. Explain why your algorithm achieves this property.

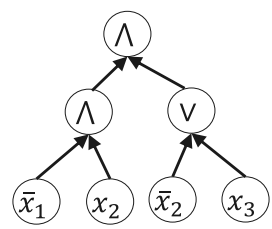

Figure 4.16: A Boolean formula ϕ over $n = 3$ variables.

- (Part e) Let $p > 2^n$ be a prime, and as usual let $\mathbb{F}_p$ denote the field of order p. This question establishes that the ability to evaluate $\widetilde{\phi}$ at a certain specific input implies the ability not only to determine whether or not ϕ is satisfiable, but in fact to *count* the number of satisfying assignments to ϕ. Specifically, prove that

$$\sum_{x \in \{0,1\}^n} \phi(x) = 2^n \cdot \widetilde{\phi}(2^{-1}, 2^{-1}, \ldots, 2^{-1}).$$

  Hint: Lagrange Interpolation.

Exercise 4.4. One of the more challenging notions to wrap one's head around regarding the GKR protocol is that, when applying it to a circuit $\mathcal{C}$ with a "nice" wiring pattern, the verifier never needs to materialize the full circuit. This is because the only information about the circuit's wiring pattern of $\mathcal{C}$ that the verifier needs to know in order to run the protocol is to evaluate $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at a random point, for each layer i of $\mathcal{C}$. And $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ often have nice, simple expressions that enable them to be evaluated at any point in time logarithmic in the size of $\mathcal{C}$. (See Section 4.6.6).

This problem asks you to work through the details for a specific, especially simple, wiring pattern. Figures 4.8–4.11 depict (for input size $n = 4$) a circuit that squares all of its inputs, and sums the results via a binary tree of addition gates.

Recall that for a layered circuit of depth d, the layers are numbered from 0 to d where 0 corresponds to the output layer and d to the input layer.

- Assume that n is a power of 2. Give expressions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ for layers $i = 1, \ldots, d-2$ such that the expressions can both be evaluated at any point in time $O(\log n)$ (layer i consists of 2^i addition gates, where for $j \in \{0, 1, \ldots, 2^i - 1\}$, the jth addition gate has as its in-neighbors gates $2j$ and $2j+1$ at layer $i+1$).

- Assume that n is a power of two. Give expressions for $\widetilde{\text{add}}_{d-1}$ and $\widetilde{\text{mult}}_{d-1}$ that can both be evaluated at any point in time $O(\log n)$. (This layer consists of $n = 2^{d-1}$ multiplication gates, where the jth multiplication gate at layer $d-1$ has both in-neighbors equal to the jth input gate at layer d).

Exercise 4.5. Write a Python program implementing the prover and verifier in the interactive proof for counting triangles from Section 4.3 (say, over the prime field $\mathbb{F}_p$ with $p = 2^{61} - 1$). Recall that in this interactive proof, the message from the prover in each round i is a univariate polynomial s_i of degree at most 2. To implement the prover $\mathcal{P}$, you may find it simplest for $\mathcal{P}$ to specify each such polynomial via its evaluations at 3 designated inputs (say, $\{0, 1, 2\}$), rather than via its (at most) 3 coefficients. For example, if $s_i(X) = 3X^2 + 2X + 1$, it may be simplest if, rather than sending the coefficients 3, 2, and 1, the prover sends $s_i(0) = 1$, $s_i(1) = 6$ and $s_i(2) = 17$. The verifier can then evaluate $s_i(r_i)$ via Lagrange interpolation:

$$s_i(r_i) = 2^{-1} \cdot s_i(0) \cdot (r_i - 1)(r_i - 2) - s_i(1) \cdot r_i(r_i - 2) + 2^{-1} \cdot s_i(2) \cdot r_i(r_i - 1).$$