Computational Efficiency and Delegation

So far we have seen how much interactive proofs can do, and the significance of many of their parameters and resources – completeness and soundness errors, private randomness, and rounds. Today, we will look at another important resource – computational efficiency of the parties – and see an interesting application of interactive proofs that research into this had led to.

1 Doubly Efficient IPs

In general, the prover in any interactive proof can be implemented with a polynomial-space algorithm – this can be seen in the proof of $\mathsf{IP} \subseteq \mathsf{PSPACE}$ that we hinted at in the first lecture. Sometimes, it happens that the prover can be implemented with the same complexity as deciding the language in question – this was the case in the proof for GNI.

Question 1. *What properties are necessary and/or sufficient for a language to have an IP such that the prover can be implemented in polymonial time given access to an oracle that tells whether any given string is in the language?*

Today, we will be interested in IPs that are computationally very efficient – that have polynomial-time provers, and almost linear-time verifiers. Such IPs are said to be *doubly efficient*. Note that linear-time is the best possible complexity for the verifier (except in some special cases), so we will be asking that the verifier be almost optimal.

Definition 1.1. An interactive protocol (P, V) is *doubly efficient* if, given an input of length n, the prover P runs in time $\mathsf{poly}(n)$, and the verifier V runs in time $O(n \operatorname{polylog}(n))$ (denoted $\widetilde{O}(n)$). The set of languages that have doubly efficient IPs is denoted $\mathsf{DE\text{-}IP}$.

We will still require the soundness guarantee to hold against unbounded cheating provers. Limiting this requirement to only polynomial-time cheating provers leads to another interesting kind of proof system called an argument, which we shall see much later in the class.

It is easy to see that any language that has a doubly efficient IP is contained in BPP, simply because a randomised polynomial-time algorithm can simulate both the prover and verifier in the IP.

Theorem 1.1. $\mathsf{DE\text{-}IP} \subseteq \mathsf{BPP}$

Along the same lines as the proof that $\mathsf{IP} \subseteq \mathsf{PSPACE}$, it may also be verified that such a language is contained in $\mathsf{SPACE}[\widetilde{O}(n)]$. Specifically, an optimal prover for a doubly efficient IP can be implemented with $\widetilde{O}(n)$ space.

Theorem 1.2. $\mathsf{DE\text{-}IP} \subseteq \mathsf{SPACE}[\widetilde{O}(n)]$

In the other direction, what all can an efficient prover prove? Can all languages in P have a doubly efficient IP? This seems unlikely, as we don't expect P to be contained in $\mathsf{SPACE}[\widetilde{O}(n)]$. How about the class $\mathsf{P} \cap \mathsf{SPACE}[\widetilde{O}(n)]$, then? This still contain many interesting problems, and while we know doubly efficient IPs for large subclasses of it [RRR16], we still don't know such proofs for the entire class

Question 2. *Does every language in* $\mathsf{P} \cap \mathsf{SPACE}[\widetilde{O}(n)]$ *have a doubly efficient* IP*?*

We will see some examples of specific languages that have doubly efficient IPs in the next problem set. Today, we will see an important construction of such IPs for languages computed by low-depth circuits.

Delegation of Computation. One important motivation for studying doubly efficient IPs is their applications to verifiable delegation of computation. This refers to a process whereby a computationally weak client (say using a phone or a laptop) delegates the execution of a program it wishes to run to a computationally powerful server (say AWS or a supercomputer). The client provides the input and the program, which the server executes on its behalf and returns the output of. The client, though, wants proof that the server performed this computation correctly. This is what a doubly efficient interactive proof can enable.

The modelling of the prover as a polynomial-time machine and the verifier as quasilinear captures this situation well. Whereas the prover is considerably more powerful than the verifier, it still cannot perform exponentially hard computations. This makes the generality of the IP we will see today quite valuable. Improving the performance of the prover and verifier in doubly efficient protocols for more and more general classes of computations, so that they can be used for better verifiable delegation, is one of the largest areas of research in interactive proofs today.

2 IP for Low-Depth Arithmetic Circuits

Fix a finite field $\mathbb{F}$. An arithmetic circuit C is a circuit comprising of addition and multiplication gates, each of fan-in 2. The input to the circuit is a number of variables $x_1, \ldots, x_n$, and constants from a finite field (which we shall ignore for now). The *size* S of the circuit is the number of gates it contains, and its *depth* D is the length of the longest path from the output gate to an input. The evaluation of such a circuit given an assignment to the inputs $x_1, \ldots, x_n \in \mathbb{F}$, denoted $C(x_1, \ldots, x_n)$, proceeds in the natural manner, by adding and multiplying the inputs to each gate over $\mathbb{F}$ to get the corresponding output, until the output of the output gate is computed. In general, we will pick $\mathbb{F}$ such that operations over it can be performed in $O(\text{polylog}(n))$ time.

In the same manner as the arithmetisation of Boolean formulas we saw earlier (replacing AND gates with multiplication, etc.), any Boolean circuit C may be transformed into an arithmetic circuit C' over any field such that for any input $x \in \{0,1\}^n$, $C(x) = C'(x)$. Further, the size and depth of C' are only a small constant factor larger than those of C. Thus, all problems in P have polynomial-sized arithmetic circuits that, in this sense, compute them.

We are interested in interactive proofs for proving that $C(x_1, \ldots, x_n) = y$ for some given arithmetic circuit C, x_i's, and y. The first thing to address here is how C is represented. We will think of C as representing a program that the verifier implicitly knows without having it written down. This is captured by having all of our circuits be uniform, in particular logspace-uniform.

Recall that a circuit of size s is logspace-uniform if there is a logspace (in s) algorithm that, given the label of any gate in the circuit, outputs the type of the gate, as well as the labels of its inputs. In our setting, both the prover and verifier already know the algorithm corresponding to a circuit C, and the inputs to the protocol are the x_i's and y. Our protocols can work with something a little weaker than logspace-uniformity, but not something as general as polynomial-time-uniformity.

A naive protocol. A very simple protocol for checking $C(x_1, \ldots, x_n) = y$ is obtained by walking down the gates of the circuit:

1. Let C_0 and C_1 be the sub-circuits that are inputs to the output gate of C.
2. P sends y_0, y_1 to V.
3. If the output gate of C is multiplication, V checks that $y = y_0 \cdot y_1$. If it is addition, V checks that $y = y_0 + y_1$. If the check fails, reject.
4. Pick a random bit $b \leftarrow \{0,1\}$.
5. If C_b is the empty circuit that just outputs its input x_i, check whether $x_i = y_b$.
6. Else, repeat the protocol with the claim $C_b(x_1, \ldots, x_n) = y_b$.

It is easy to see that the protocol uses at most D rounds, and is perfectly complete. The prover only has to evaluate the circuit, so it runs in $\widetilde{O}(D)$ time. The verifier performs a constant number of field operations in each round, so runs in $O(D \cdot \text{polylog}(n))$ time. So this would give a doubly efficient IP for

this task if $S = \mathsf{poly}(n)$ and $D = \text{polylog}(n)$ if it were sound. The soundness error, however, is quite large.

Exercise 1. *Show that the soundness error in the above protocol could be as large as* $(1 - 2^{-D})$.

Further, any attempt to amplify this soundness error by repetition would make the verifier running time super-linear, thus making the protocol not doubly efficient.

A better protocol. We will see now a much better interactive proof for this task due to Goldwasser, Kalai, and Rothblum [GKR08]. This protocol essentially started the branch of research in interactive proofs that is most prominent today. It gives an interactive proof where, given any *logspace-uniform* circuit of size S and depth D, the prover runs in time $\mathsf{poly}(S)$, and the verifier in time $O(n+D\,\text{polylog}(S))$. Further, the soundness error is at most $O(D \log S/\,|\mathbb{F}|)$. This, in particular, implies the following.

Theorem 2.1. L*-uniform* $\mathsf{NC} \subseteq \mathsf{DE\text{-}IP}$

Note that NC, even L-uniform NC, is a considerably large class, and contains a significant fraction of the problems of interest in P. To describe this protocol, we will need a tool that we saw while studying the sumcheck protocol – low-degree extensions.

2.1 Multilinear Extensions

In particular, we will use a special form of low-degree extensions called multilinear extensions. Recall that a multivariate polynomial is multilinear if its degree in each of its variables is at most 1.

Definition 2.1. Given a function $f : \{0,1\}^s \to \mathbb{F}$ and a multilinear polynomial $g : \mathbb{F}^s \to \mathbb{F}$, g is a *multilinear extension* (MLE) of f if, for every $x \in \{0,1\}^s$, $f(x) = g(x)$.

Claim 2.1. *For any function* $f : \{0,1\}^s \to \mathbb{F}$*, there exists a unique MLE* $\hat{f} : \mathbb{F}^s \to \mathbb{F}$*, which is the following:*

$$\hat{f}(x) = \sum_{z \in \{0,1\}^s} f(z) \cdot \prod_{i=1}^{s} (x_i z_i + (1 - x_i)(1 - z_i))$$

The facts that $\hat{f}$ above is multilinear and is an extension of f are easy to verify. The proof that it is unique as an MLE is left as an exercise. Hereafter, we use the hat $(\hat{f})$ to denote the MLE of any function (f). From the above expression for $\hat{f}$, the following is also clear.

Claim 2.2. *Suppose the function* f *is non-zero on at most* m *inputs* $x \in \{0,1\}^s$*. Then, given a list of these non-zero inputs,* $\hat{f}$ *on any input can be computed with* $O(ms)$ *field operations.*

As with low-degree extensions in general, MLEs are error-correcting encodings of functions. Further, they have useful local error-correction properties. We will be using these properties in the upcoming protocol.

2.2 The GKR protocol

To start with, we will make the following assumptions regarding the structure of the given circuit C, which is of size S and depth D, and takes n inputs. Note that they are all without loss of generality, with the exception of the last. That is, C can be modified easily to satisfy them without losing functionality. The last assumption is for simplicity of presentation, and can be removed with a small modification to the protocol we describe.

1. The circuit is layered.
   - There are D layers of gates.
   - The gates in layer 1 take inputs directly from the x_i's.
   - The first gate in layer D is the output gate of the circuit.
   - For $i > 1$, the inputs to gates in layer i come only from the outputs of gates in layer $(i-1)$.
2. Each layer contains exactly S gates, and $S = 2^s$.
3. None of the gates take any constants from the field as input.

Setup. The basic idea underlying the protocol is similar to that behind the naive protocol described earlier in the section – start with the output gate, and go down layer-by-layer, checking that the outputs of gates in each layer are computed correctly. The key difference is that, whereas the naive protocol only checked the computation of one gate in each layer, we will, in a sense, check *all* of the gates in each layer. To this end, we will employ the MLE in its capacity as an error-correcting encoding.

Fix any input $x_1, \ldots, x_n \in \mathbb{F}$, and the claimed output $y \in \mathbb{F}$. For each $i \in [D]$, define a function $L_i : [S] \to \mathbb{F}$ such that:

$$L_i(u) = \text{output of the } u^{\text{th}} \text{ gate in layer } i \text{ when evaluating } C(x_1, \ldots, x_n)$$

Corresponding to the inputs, also define the function $L_0 : [S] \to \mathbb{F}$ such that $L_0(u) = x_u$ if $u \in [n]$, and $L_0(u) = 0$ otherwise. We will interpret the u above, which is an index in $[S] = [2^s]$, as a string in $\{0,1\}^s$ in the natural manner. That is, $u = (u_1, \ldots, u_s)$, where $u_i \in \{0,1\}$. Thinking of L_i now as a function from $\{0,1\}^s$ to $\mathbb{F}$, we can define its MLE $\widehat{L}_i$ over $\mathbb{F}$ as in Section 2.1.

For any alternative function L_i' that differs from L_i on even one u, its MLE $\widehat{L}_i'$ will differ from $\widehat{L}_i$ at a large number of points. This fact, which we will prove implicitly, will enable the verifier to catch the prover if it makes a false claim about the output of one of the gates in layer i. Following our outline, we will perform this task recursively, starting with L_D and proceeding downwards. By claiming that $C(x_1, \ldots, x_n) = y$, the prover is essentially claiming that $\widehat{L}_D(1) = y$. The verifier will now ask the prover for an $\widehat{L}_{D-1}$ that validates this claim, then an $\widehat{L}_{D-2}$ that validates the $\widehat{L}_{D-1}$, and so on. Of course, the description of the $\widehat{L}_i$'s is too large to involve in the protocol. The fact that these are MLE's will enable us to perform these checks without having to check them at every point.

Relating the MLE's across layers In order to use $\widehat{L}_{i-1}$ to check anything about $\widehat{L}_i$, we will need to relate these two functions in a manner amenable to interactive proofs. Sure, for $u \in \{0,1\}^s$, we have simple relations between $\widehat{L}_i(u)$ and some $\widehat{L}_{i-1}(v)$ and $\widehat{L}_{i-1}(w)$ as specified by the gate at that location, but we will need to look at u's that are not contained in $\{0,1\}^s$. We now formulate such a relation.

Recall that the circuit C is actually described by a logspace algorithm that, given a label (i, u) of the u^{th} gate in layer i, tells whether it is an addition or multiplication gate, and what its inputs are. The uniformity algorithm can be easily used to implement, for each $i \in [D]$, the following function $add_i : (\{0,1\}^s)^3 \to \{0,1\}$:

$$add_i(u,v,w) = \begin{cases} 1 & \text{if the } u^{\text{th}} \text{ gate at layer } i \text{ is an add gate whose inputs are the outputs of} \\ & \text{the } v^{\text{th}} \text{ and } w^{\text{th}} \text{ gates from layer } (i-1) \\ 0 & \text{otherwise} \end{cases}$$

Similarly, the functions $mult_i : (\{0,1\}^s)^3 \to \{0,1\}$ can be defined with respect to multiplication gates. The following expression is now easily verified for any $i \in [D]$ and $u \in \{0,1\}^s$:

$$L_i(u) = \sum_{v,w \in \{0,1\}^s} add_i(u,v,w) \cdot (L_{i-1}(v) + L_{i-1}(w)) + mult_i(u,v,w) \cdot (L_{i-1}(v) \cdot L_{i-1}(w))$$

This is because there is exactly one pair (v, w) for which even one of $add_i(u,v,w)$ and $mult_i(u,v,w)$ is non-zero, and for that pair, the terms in the above sum compute the correct function. More interestingly, by the uniqueness of MLE's, the following also holds:

$$\widehat{L}_i(u) = \sum_{v,w \in \{0,1\}^s} \widehat{add}_i(u,v,w) \cdot (\widehat{L}_{i-1}(v) + \widehat{L}_{i-1}(w)) + \widehat{mult}_i(u,v,w) \cdot (\widehat{L}_{i-1}(v) \cdot \widehat{L}_{i-1}(w)) \qquad (1)$$

For simplicity, we will assume that the verifier in our protocol also has oracle access to the MLE's $\widehat{add}_i$ and $\widehat{mult}_i$. In reality, these functions are not easy to compute, and will have to be computed with help from the prover.

Part 1: sumcheck. We will now run through the first iteration of the protocol. The rest of the protocol will simply repeat these steps in a straightforward manner, so we leave out its formal description.

For every $i \in [D]$ and $u \in \mathbb{F}$, we define the following polynomial $g_{i,u} : \mathbb{F}^{2s} \to \mathbb{F}$ for convenience. Here, $v, w \in \mathbb{F}^s$.

$$g_{i,u}(v,w) = \widehat{add}_i(u,v,w) \cdot (\widehat{L}_{i-1}(v) + \widehat{L}_{i-1}(w)) + \widehat{mult}_i(u,v,w) \cdot (\widehat{L}_{i-1}(v) \cdot \widehat{L}_{i-1}(w))$$

Note that since each of the MLE's are multilinear, $g_{i,u}$ has degree at most 2 in each variable.

When the protocol starts, the prover claims that $\widehat{L}_D(1) = y$. By (1) this is same as the claim that $\sum_{v,w \in \{0,1\}^s} g_{i,u}(v,w) = y$. But we already know how to efficiently prove claims of this form – the sumcheck protocol! So the prover and verifier execute the sumcheck protocol to verify this claim.

Recall that in the sumcheck protocol, all the verifier does is send a number of random field elements, and then at the end evaluate the polynomial being summed at a random point (in our exposition it was two fixed points, but it is easy to reduce this to just one random point by adding another round to the protocol). Thus, all the verifier needs to do here is evaluate $g_{i,u}$ at one point (v,w). This, however, involves evaluating $\widehat{L}_{i-1}$ at two points – v and w. Reducing the task to evaluating $\widehat{L}_{i-1}$ is exactly what we need to do if we want to recurse, but having to evaluate it at two points is bad, as then the recursion will blow up.

Part 2: reducing 2 evaluations to 1. Here again we will use the fact that $\widehat{L}_{i-1}$ is a low-degree polynomial, to reduce the task of evaluating it at two points to evaluating it at one random point. Consider the line $\ell : \{0,1\} \to \{0,1\}^s$ defined as $\ell(z) = (1-z) \cdot v + z \cdot w$. Notice that $\ell(0) = v$ and $\ell(1) = w$. Now define the univariate polynomial $q(z) = \widehat{L}_{i-1}(\ell(z))$. Since $\widehat{L}_{i-1}$ is multilinear in s variables, and ℓ is linear, q has degree at most s. Further, $q(0) = \widehat{L}_{i-1}(v)$, and $q(1) = \widehat{L}_{i-1}(w)$. Thus, if the verifier knew q, it could compute these two values itself. Further, since q is a low-degree univariate polynomial, it can be checked easily.

After the sumcheck protocol, the prover claims that $\widehat{L}_{i-1}(v) = y_0$ and $\widehat{L}_{i-1}(w) = y_1$ by sending the coefficients of a univariate polynomial q of degree s such that $q(0) = y_0$ and $q(1) = y_1$. The verifier uses the y_0 and y_1 to complete the sumcheck protocol, then turns to checking that these are indeed the correct evaluations of $\widehat{L}_{i-1}$. To do this, it checks whether $q(z)$ is indeed the polynomial $\widehat{L}_{i-1}(\ell(z))$. This is done by checking that these two polynomials agree at a random point $z_0 \in \mathbb{F}$. The verifier can evaluate $q(z_0)$ on its own. All that remains is to check that $\widehat{L}_{i-1}(\ell(z_0)) = q(z_0)$, which we can now recurse on.

Finally, at the very end of the recursion, the verifier reaches $\widehat{L}_0$, which it has to evaluate at two points, and this it does on its own.

Exercise 2. *Write out a formal description of the above protocol.*

Efficiency. The complexity of the verifier in each sumcheck protocol is $O(s)$ field operations. Apart from this the verifier needs to evaluate the polynomial q in each step, and at the end evaluate $\widehat{L}_0$ at two points on its own. The latter can be done in $O(ns)$ field operations due to Claim 2.2, and the former in $O(s)$ operations. Its running time in the entire protocol is thus $O(ns + Ds)$ field operations, which is $O((n+D) \log S \operatorname{polylog} |\mathbb{F}|)$ time.

The communication in the protocol consists of $O(D \log S)$ field elements. Finally, the prover runs in $O(DS \log S \operatorname{polylog} |\mathbb{F}|)$ time.

Completeness. By the perfect completeness of the sumcheck protocol, each recursion of the above subprotocol runs with a claim $\widehat{L}_i(\ell(z_0)) = q(z_0)$ that is true. Thus, the entire protocol is perfectly complete.

Soundness. We compute the soundness error inductively. Think of the protocol as one that checks $\widehat{L}_D(x) = y$ for any arbitrary x and y. Let us look at the first recursion. Suppose the claim $\widehat{L}_D(1) = y$ is false. At the end of this subprotocol, we recurse with the claim $\widehat{L}_{D-1}(\ell(z_0)) = q(z_0)$. The probability that the verifier accepts can be bounded as:

$$\Pr[V \text{ accepts}] \leq \Pr\left[V \text{ accepts} \wedge \widehat{L}_{D-1}(\ell(z_0)) = q(z_0)\right] + \Pr\left[V \text{ accepts} \mid \widehat{L}_{D-1}(\ell(z_0)) \neq q(z_0)\right]$$

The second term above is just the soundness error of the protocol when the depth is $D-1$. We can again bound the first term as follows:

$$\Pr\left[V \text{ accepts} \wedge \widehat{L}_{D-1}(\ell(z_0)) = q(z_0)\right] \leq \Pr\left[V \text{ accepts} \wedge (\widehat{L}_{D-1}(v) = q(0)) \wedge \widehat{L}_{D-1}(w) = q(1))\right] + \Pr\left[V \text{ accepts} \wedge \widehat{L}_{D-1}(\ell(z_0)) = q(z_0) \mid (\widehat{L}_{D-1}(v) \neq q(0)) \vee \widehat{L}_{D-1}(w) \neq q(1))\right]$$

By the soundness of the sumcheck protocol, the first term above is $O(s/|\mathbb{F}|)$. The second term is also $O(s/|\mathbb{F}|)$, since q is of degree at most s. Thus, each recursion adds a soundness error of $O(s/|\mathbb{F}|)$. If $D = 1$, the protocol is just sumcheck, and the soundness error is $O(s/|\mathbb{F}|)$. So the soundness error of the entire protocol is $O(D \log S/|\mathbb{F}|)$.

References

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 113–122. ACM, 2008.

[RRR16] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 49–62. ACM, 2016.