Doubly Efficient Proof Systems

Ziyang Jin[1]

Department of Computer Science
University of Toronto
ziyang@cs.toronto.edu

Abstract. This note is prepared for presenting the paper *Delegating Computation: Interactive Proofs for Muggles* [1] (a.k.a [GKR '08]) in the derandomization course taught by Prof. Roei Tell at the University of Toronto in winter 2024. We call it a doubly efficient proof system since the prover runs in polynomial time, and the verifier runs in quasi-linear time. We will mainly present the protocol in the GKR paper. The structure follows the ECCC version of this paper [2].

Keywords: Complexity · Interactive Proofs · low-degree extension · sum-check protocol

1 Motivation

In the definition of standard interactive proof systems, the prover (Merlin) is usually assumed to have unbounded computational power, and the verifier (Arthur) runs in probabilistic polynomial time. The GKR paper studies interactive proof systems for tractable languages, where the prover runs in polynomial time, in other words, a "muggle" (compared to powerful mage Merlin). It would not be very interesting for the verifier to also run in polynomial time since in this case, the verifier can do all the computation himself without interacting with the prover. So we require verifier to run in quasi-linear time, for example, $O(n \log n)$ time where n is the size of the input.

Doubly efficient proof systems can be used for *delegating computation*: a server in the cloud can run the computation in polynomial time and send the result to the client (e.g. your smart watch). However, the server can cheat by sending a wrong answer or randomly guessing an answer without actually performing the computation. Thus, it is important that the client can verify the correctness of the answer without running the entire computation himself. The GKR protocol allows the verifier to check the correctness of the result in quasi-linear time to the size of input and proportionally to the depth of the circuit.

2 Main Theorem

Suppose the verifier would like the prover to compute some function $f : \{0,1\}^n \to \{0,1\}$ on some input $x \in \{0,1\}^n$. Note that any boolean function can be translated to a boolean circuit $C : \{0,1\}^n \to \{0,1\}$. For simplicity, assume the boolean circuit only has NAND gates with fan-in 2 (any circuit can be translated to a circuit in this form with at most polynomial blow-up in size). Without loss of generality, suppose $C(x) = 0$. Thus, the goal of the protocol is to convince the verifier that $C(x) = 0$.

Let $x \in \{0,1\}^n$ be the input. Let S be the size of the circuit and let d be the depth of the circuit. Roughly, we have $S = \text{poly}(n)$, and d needs to be asymptotically smaller than S to be interesting (typically, we take $d = \text{polylog}(S) = \text{polylog}(n)$).

Theorem 1 (Goldwasser-Kalai-Rothblum '08). *Let L be a language that can be computed by a family of $O(\log(S))$-space uniform boolean circuits of size S and depth d. L has an interactive proof where:*

1. *The prover runs in time $\text{poly}(S)$. The verifier runs in time $n \cdot \text{poly}(d, \log S)$ and space $O(\log(S))$.*
2. *The protocol has perfect completeness and soundness $1/100$.*
3. *The protocol is public-coin, with communication complexity $d \cdot \text{polylog}(S)$.*

One caveat here is that it restricts the computation here to be logspace uniform boolean circuits. The reason is that we need the verifier to have efficient access the circuit structure (i.e. asking if one gate is the parent of another two gates) without the prover sending the entire circuit. To achieve this, the circuit needs to be very structured. The authors find that circuits that are generated by logspace Turing machines can be very structured such that we could have short representations of the circuit structure. In genereal, we do not know how to do it for circuits that are generated by polynomial time Turing machines.

In this presentation, we abstract the efficient computation of the circuit structure by an oracle. We assume that both the prover and the verifier have oracle access to a circuit structure function $\phi(g_1, g_2, g_3)$ that returns 1 if and only if gate g_1 is the parent gate of g_2 and g_3. Section 4 of the GKR paper talks about how to let the verifier compute this oracle efficiently, and later Goldreich presented a simpler construction. This will not be the main part of our presentation.

3 A Naive Approach

We model the circuit to have a layered structure. Any circuit can be converted to this layered structure with at most a polynomial blow-up in size. The circuit has depth d, so there will be $d + 1$ layers. We label the output of the circuit as layer 0, and one layer below as layer 1, and continue downwards. The bottom layer that consists of the input bits will be layer d. Every gate is a NAND gate with fan-in 2, and the input of any gate can only come from one layer below it, i.e. no wires can cross more than one layer.

If an honest prover evaluates this circuit on input x, then he can note down the output of every gate in every layer in this circuit. The top layer only has one gate, and its output is 0 since we assumed $C(x) = 0$.

Here is how we model the game between the prover and the verifier in the naive way. The prover tells the verifier that in the top layer (layer 0), $C(x) = 0$, and the verifier asks what are the two inputs in the layer below (layer 1). The provers says, for example, that they are the first and the fifth gate, and the values are 1 and 1, respectively. Then the verifier continues to ask what are the four values in layer 2 that makes the output 1 and 1 in layer 1, and so on. In the end, when the verifier reaches layer d, i.e. the input layer, since the verifier has the input, he can check himself if the values in the input layer reported by the prover is correct.

For the naive approach above, completeness is guaranteed. Since an honest prover will always report the true output of every gate. Runtime is bad since the number of gate outputs that the verifier needs to ask grows exponentially with the layer.

One (still naive) way to reduce the runtime is to let the verifier randomly pick one out of two gates in the layer below to check. This way hurts the soundness of this proof system. Here is a simplified picture: suppose $C(x) = 1$ and the prover tries to convince the verifier that $C(x) = 0$. We assume the worst adversary such that the prover just need to lie about a single gate in the entire circuit, and the gate is in layer $d - 1$. The probability for the verifier to catch the "lying gate" would be $1/2^d$, which is terrible for soundness. We might come up with other protocols that are more complicated than this, but many of them would still have bad soundness. We will not go into too much details in this.

To summarize, with the first naive approach, the challenge is to reduce the runtime, but soundness is great; with the second naive approach, the challenge is to improve the soundness, but the runtime is great. The GKR paper starts from the second naive approach, and improves the soundness by applying an error correcting code to each layer. The intuition behind using error correcting code is natural. In the naive protocol above, we randomly choose one out of two gate outputs in the next layer, and that does not give a good probability to catch the cheating prover. Now in every layer, we encode the gate outputs as an error correcting code, such that the codeword of wrong gate outputs disagrees with the codeword of true gate outputs in most places, and we will have much higher probability in catching the cheating prover by randomly choosing one in the codeword space.

However, once they have done that, the nice downward self-reducibility is ruined, and now the verifier is inefficient in finding the circuit structure. To solve this problem, they use a specially-designed low-degree extension, and add sum-check sub-protocols along with a "2-to-1 trick", to make the verifier efficient again.

4 Setup of the Protocol

4.1 Padding the outputs in each layer

We model the circuit the same way (the layered structure) as we did in the naive approach. The prover writes the output of the gates layer by layer. Here is a toy illustration:

$$\begin{array}{rll}
\text{layer } 0: & 0 & \text{(output)} \\
\text{layer } 1: & 1\ 1 & \\
\text{layer } 2: & 1\ 0\ 0\ 1 & \\
\vdots & \vdots & \\
\text{layer } i: & 0\ 0\ 1\ 0\ 0\ 0 \ldots & \\
\text{layer } i+1: & 1\ 1\ 1\ 1\ 0\ 0 \ldots\ 0 & \\
\vdots & \vdots & \\
\text{layer } d: & 0\ 1\ 1\ 0\ 1\ 1 \ldots\ 1 & \text{(input)}
\end{array}$$

Now we pad zeros in each layer $0 \leq i \leq d-1$ such that each layer now has width exactly S. Note that we do not pad the input layer.

$$\begin{array}{rl}
\text{layer } 0: & 0\ 0\ 0\ 0\ 0\ 0 \ldots\ 0 \\
\text{layer } 1: & 1\ 1\ 0\ 0\ 0\ 0 \ldots\ 0 \\
\text{layer } 2: & 1\ 0\ 0\ 1\ 0\ 0 \ldots\ 0 \\
\vdots & \vdots \\
\text{layer } i: & 0\ 0\ 1\ 0\ 0\ 0 \ldots\ 0 \\
\text{layer } i+1: & 1\ 1\ 1\ 1\ 0\ 0 \ldots\ 0 \\
\vdots & \vdots \\
\text{layer } d: & 0\ 1\ 1\ 1\ 1\ 1 \ldots\ 1
\end{array}$$

Now each layer (except the input layer) is a binary array of size S. Layer 0 is now a vector $(0, 0, 0, ..., 0)$ (S zeros). We can encode this array as a function from the index to the truth value, i.e., for layer i, we have function $\alpha_i : [S] \to \{0, 1\}$. Since we assume $C(x) = 0$, we have $\alpha_0(0) = 0$. In the input layer (layer d), we have $\alpha_d(j) = x_j$ where j is the index of the jth bit in the input x.

4.2 Low-degree extension

Normally, the index $j \in [S]$ is represented as a binary string. So we can represent the index in $\log_2(S)$ bits and think of the function as $\alpha_i : \{0, 1\}^{\log S} \to \{0, 1\}$. But binary does not need to be the only representation of a number if we work in finite fields. We could equivalently write the index j as base 3, then we can think of the function as $\alpha_i : \mathbb{F}_3^k \to \{0, 1\}$, where $\mathbb{F}_3$ is a finite field containing elements $\{0, 1, 2\}$, and $3^k = S$. All we are doing here is just changing the representation of the index. Now we choose to represent the index in base $|\mathbb{H}|$ where $|\mathbb{H}|^m = S$ and $\mathbb{H}$ is an extension field of $\mathbb{GF}[2]$. Jumping ahead, the choice of $|\mathbb{H}|$ and m will affect the runtime of the verifier. We will choose $|\mathbb{H}|$ and m later. Right now, all we care about is that $|\mathbb{H}|^m = S$ and now we fix $\alpha_i : \mathbb{H}^m \to \{0, 1\}$.

Now we apply a linear error correcting code called *low-degree extension* to every α_i, denoted by $\tilde{\alpha}_i : \mathbb{F}^m \to \mathbb{F}$. Note that we expand the domain from $\mathbb{H}^m$ to $\mathbb{F}^m$, and range from $\{0, 1\}$ to $\mathbb{F}$. We require $\mathbb{F}$ to be an extension field of $\mathbb{H}$ (thus $\mathbb{H} \subseteq \mathbb{F}$), and we require $|\mathbb{F}| = \text{poly}(|\mathbb{H}|)$. In general, we want $\mathbb{F}$ to be big such that the error correcting code will have good distance. Jumping ahead, for people who are familiar with techniques involving error correcting code, $|\mathbb{F}|$ will be the denominator in the Schwartz-Zippel lemma. A low-degree extension $\tilde{\alpha}_i$ is a m-variate polynomial of individual degree δ that agrees with $\alpha_i : \mathbb{H}^m \to \{0,1\}$ on all points $p \in \mathbb{H}^m$. For any points $q \in \mathbb{F}^m \setminus \mathbb{H}^m$, $\tilde{\alpha}_i(q)$ can "go crazy" and take any value in $\mathbb{F}$.

Claim. If $\tilde{\alpha}_i$ has individual degree at most $|\mathbb{H}|-1$, then the low-degree extension is unique, i.e., there exists a unique polynomial $\tilde{\alpha}_i : \mathbb{F}^m \to \mathbb{F}$ that agrees with α_i on $\mathbb{H}^m$ and has individual degree $|\mathbb{H}|-1$.

We will not prove this claim here; a proof can be found in [3]. If you have worked with polynomials and finite fields long enough, this claim should be quite convincing. In the GKR paper, for $\tilde{\alpha}_i$, they obtain an individual degree δ to be slightly bigger than $|\mathbb{H}|-1$ due to the specific construction in section 4 of their paper. Thus, the extension is not unique, but the degree δ should still be low enough. Typically, we require $\delta \geq |\mathbb{H}|-1$ and $md\delta \ll |\mathbb{F}|$. For example, we can take $|\mathbb{H}| = n^{0.01}$, and note that $S = \text{poly}(n)$, so m is just a big constant. We usually consider $d = \text{polylog}(n)$.

I understand it is a bit annoying to not tell you the explicit variables and coefficients of the low-degree polynomial. I intentionally choose to hide it since there is a standard way to construct a low-degree extension; however, the GKR paper uses a different way to describe the low-degree extension. The best way to move on is to view it as an abstraction, i.e., $\tilde{\alpha}_i : \mathbb{F}^m \to \mathbb{F}$ is a low-degree polynomial that agrees with α_i on $\mathbb{H}^m$.

4.3 Oracle for the circuit structure

Let $g_1, g_2, g_3 \in [S]$ be the index of three gates, where g_1 is at layer $i-1$ and g_2, g_3 are gates at layer i. Define function $\phi_i(g_1, g_2, g_3) : [S]^3 \to \{0,1\}$ such that it evaluates to 1 if g_1 is the parent gate of g_2 and g_3, and evaluates to 0 otherwise. Note that instead of looking at the entire circuit, we can just query ϕ_i to obtain the local connectivity between gates in layer $i-1$ and layer i. Therefore, the set of functions $\{\phi_1, ..., \phi_d\}$ describes the entire circuit structure. Ideally, we want to provide $\{\phi_1, ..., \phi_d\}$ as the oracle to the verifier. However, since we work in the space of codewords, we need low-degree extensions of ϕ_i's.

Let $\tilde{\phi}_i(z_1, z_2, z_3) : \mathbb{F}^{3m} \to \mathbb{F}$ be a low-degree extension of ϕ_i, which is a polynomial of $3m$ variables, with individual degree at most δ where $|\mathbb{H}|-1 \leq \delta \ll |\mathbb{F}|$. We can think of $z_1, z_2, z_3 \in \mathbb{F}^m$ as *virtual gates.* When $z_1, z_2, z_3 \in \mathbb{H}^m$, then z_1, z_2, z_3 corresponds to actual gates g_1, g_2, g_3 in each (padded) layer. We require that $\tilde{\phi}_i(z_1, z_2, z_3)$ agrees with $\phi_i(g_1, g_2, g_3)$ when $z_1, z_2, z_3 \in \mathbb{H}^m$ and corresponds to $g_1, g_2, g_3 \in [S]$. If any of z_1, z_2, z_3 is in $\mathbb{F}^m \setminus \mathbb{H}^m$ (i.e. not corresponding to an actual gate), then $\tilde{\phi}_i(z_1, z_2, z_3)$ can "go crazy".

Define $\mathcal{F} = \{\tilde{\phi}_i : 1 \leq i \leq d\}$. We call $\mathcal{F}$ the *oracle* that describes the circuit structure of C, and we give $\mathcal{F}$ to both the prover and the verifier. Note that in section 4 of the GKR paper, they remove this oracle assumption and explain how the verifier can compute each $\tilde{\phi}_i$ efficiently. In this note we will just abstract $\tilde{\phi}_i$

away as oracles, and in section 8 of this note we will briefly describe Goldreich's construction, which is believed to be simpler than the original construction in the GKR paper.

5 The GKR Protocol

Let us recap the setup we have done in the previous section. We have the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ that both have input x, where $|x| = n$. The prover has a layered circuit C, whose size is S and depth is d. The prover wants to convince the verifier that $C(x) = 0$.

The prover and the verifier are both given oracle access to $\mathcal{F}$, which describe the circuit structure. Since $\mathcal{F}$ is a set of low-degree extensions. Both the prover and the verifier knows the parameters $\mathbb{H}, m, \mathbb{F}, \delta$ that define the low-degree extension, where $|\mathbb{H}|^m = S$, $|\mathbb{F}| = \text{poly}(|\mathbb{H}|)$.

Now the prover evaluates the circuit on input x and writes down the output of every gate in every layer, and we do the same padding as defined in section 4.1. Recall the function $\alpha_i : [S] \to \{0, 1\}$ we defined in section 4.1 and its low-degree extension $\tilde{\alpha}_i$ we defined in section 4.2. We call a point $z \in \mathbb{F}^m$ a *virtual gate*, and if $z \in \mathbb{H}^m$, it corresponds to an *actual gate*. Let $z_0 = (0, 0, ..., 0) \in \mathbb{F}^m$ correspond to the first gate in the top layer. Proving $C(x) = 0$ is now equivalent to proving $\tilde{\alpha}_0(z_0) = 0$.

The protocol $(\mathcal{P}^{\mathcal{F}}(x), \mathcal{V}^{\mathcal{F}}(x))$ is done in d phases. In the ith phase $(1 \le i \le d)$, the prover reduces that task of proving that $\tilde{\alpha}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving that $\tilde{\alpha}_i(z_i) = r_i$, where z_i is a random "virtual gate" determined by the verifier ($z_0 = (0, ..., 0)$ and $r_0 = 0$). This is achieved by running a sum-check protocol followed by a "2-to-1 trick" which we will describe in detail. In the end, after the dth phase, it reduces to checking $\tilde{\alpha}_d(z_d) = r_d$, where z_d is a "virtual index" of some bit in the input. At this stage, the verifier needs to compute a low-degree extension of the input x himself and compare the result.

5.1 The ith phase

The prover sends r_{i-1} to the verifier, and the prover wants to convince the verifier that $\tilde{\alpha}_{i-1}(z_{i-1}) = r_{i-1}$. Our task is to reduce proving $\tilde{\alpha}_{i-1}(z_{i-1}) = r_{i-1}$ to proving that $\tilde{\alpha}_i(z_i) = r_i$.

For every $z_{i-1} \in \mathbb{F}^m$, we can express $\tilde{\alpha}_{i-1}(z_{i-1})$ in terms of values in $\tilde{\alpha}_i$ as follows:

$$\tilde{\alpha}_{i-1}(z_{i-1}) = \sum_{w_1, w_2 \in \mathbb{H}^m} \tilde{\phi}_i(z_{i-1}, w_1, w_2) \cdot \widetilde{\text{NAND}}(\tilde{\alpha}_i(w_1), \tilde{\alpha}_i(w_2)). \tag{1}$$

Note that $\widetilde{\text{NAND}} : \mathbb{F}^{2m} \to \mathbb{F}$ is an arithmetization of the NAND gate. For $x, y \in \mathbb{F}^m$, $\widetilde{\text{NAND}}(x, y) = 1 - xy$. Note that when $x, y \in \{0, 1\}$, $\widetilde{\text{NAND}}(x, y)$ outputs 0 or 1 according to the NAND gate; when $x, y \in \mathbb{F} \setminus \{0, 1\}$, the value of

$\tilde{\text{NAND}}(x, y)$ can "go crazy". To simplify, we define the function $f_z : \mathbb{F}^{2m} \to \mathbb{F}$ as follows:

$$f_z(w_1, w_2) := \tilde{\phi}_i(z_{i-1}, w_1, w_2) \cdot \tilde{\text{NAND}}(\tilde{\alpha}_i(w_1), \tilde{\alpha}_i(w_2)). \tag{2}$$

Note that $w_1, w_2 \in \mathbb{F}^m$, so f_z can be seen as a $2m$-variate polynomial with individual degree at most $\delta + |\mathbb{H}| - 1 \leq 2\delta$. The number of coefficients of f_z is $\binom{m+\delta}{m}$ (stars and bars). Since we treat m to be a big constant, this is ¡ $O(\delta^m)$. Thus, we can represent any such polynomial with at most $O(\delta^m)$ field elements, so in total $O(\delta^m \log |\mathbb{F}|)$ bits. Note that $\delta \geq |\mathbb{H}| - 1$, and if we take $|\mathbb{H}| = n^{0.01}$, then the size of f_z is $\text{poly}(S)$. Thus, we have

$$\tilde{\alpha}_{i-1}(z_{i-1}) = \sum_{w_1, w_2 \in \mathbb{H}^m} f_z(w_1, w_2).$$

Therefore, proving that $\tilde{\alpha}_{i-1}(z_{i-1}) = r_i$ is equivalent to proving that

$$r_{i-1} = \sum_{w_1, w_2 \in \mathbb{H}^m} f_z(w_1, w_2).$$

Note that it will not work if the prover just sends the polynomial f_z to the verifier and let the verifier sum over all choices of w_1, w_2. This is because the verifier needs to compute on all $O(|\mathbb{H}|^{2m})$ tuples of (w_1, w_2), which is way above the verifier's time budget: note that $|\mathbb{H}|^m = S$ and we want the verifier to run in time much less than the circuit size. To solve this problem, we use a sum-check protocol.

The sum-check protocol We will shift the notation a little bit here for disambiguation. Instead of using $z_{i-1}, w_1, w_2 \in \mathbb{F}^m$, we now denote them by $\vec{z}_{i-1}, \vec{w_1}, \vec{w_2} \in \mathbb{F}^m$ to emphasize they are vectors of m elements, and we define

$$\tilde{\alpha}_{i-1,0} = \sum_{\vec{w}_1, \vec{w}_2 \in \mathbb{H}^m} f_z(\vec{w}_1, \vec{w}_2), \tag{3}$$

and

$$\tilde{\alpha}_{i-1,1}(x) = \sum_{w_{1,2}, .., w_{1,m} \in \mathbb{H}, \vec{w}_2 \in \mathbb{H}^m} f_z(x, w_{1,2}, ..., w_{1,m}, \vec{w}_2).$$

Note that $\vec{z}_{i-1}$ is a fixed value given in the beginning of the phase. It is not a variable. So $\tilde{\alpha}_{i-1,0} = r_{i-1}$ is a fixed value. And $\tilde{\alpha}_{i-1,1}(x)$ is a univariate polynomial with degree at most 2δ.

Define $r_{i-1,0} = r_{i-1}$. The prover computes the polynomial $\tilde{\alpha}_{i-1,1}(x)$ and sends to the verifier. Then the verifier checks if

$$\sum_{x \in \mathbb{H}} \tilde{\alpha}_{i-1,1}(x) = r_{i-1,0}.$$

If they are not equal, reject. Otherwise, the verifier picks a random $w_{1,1} \in \mathbb{F}$ and sends to the prover. The verifier also computes

$$r_{i-1,1} := \tilde{\alpha}_{i-1,1}(w_{1,1}).$$

Note that the prover also knows $r_{i-1,1}$ since upon receiving $w_{1,1}$, the prover can compute $r_{i-1,1}$ himself.

Now the prover computes

$$\tilde{\alpha}_{i-1,2}(x) = \sum_{w_{1,3},..,w_{1,m} \in \mathbb{H}, \vec{w}_2 \in \mathbb{H}^m} f_z(w_{1,1}, x, w_{1,3}, ..., w_{1,m}, \vec{w}_2).$$

Note that $w_{1,1}$ is a fixed value now, so $\tilde{\alpha}_{i-1,2}(x)$ is a univariate polynomial. The prover sends $\tilde{\alpha}_{i-1,2}(x)$ to the verifier. The verifier checks if

$$\sum_{x \in \mathbb{H}} \tilde{\alpha}_{i-1,2}(x) = r_{i-1,1}.$$

If not, reject. Otherwise, the verifier picks a value $w_{1,2} \in \mathbb{F}$ and computes

$$r_{i-1,2} := \tilde{\alpha}_{i-1,2}(w_{1,2})$$

and the verifier sends $w_{1,2}$ to the prover. The prover computes (fixing $w_{1,1}, w_{1,2}$)

$$\tilde{\alpha}_{i-1,3}(x) = \sum_{w_{1,4},..,w_{1,m} \in \mathbb{H}, \vec{w}_2 \in \mathbb{H}^m} f_z(w_{1,1}, w_{1,2}, x, w_{1,4}, ..., w_{1,m}, \vec{w}_2).$$

and sends to the verifier, and the verifier checks if

$$\sum_{x \in \mathbb{H}} \tilde{\alpha}_{i-1,3}(x) = r_{i-1,2}$$

and so on. We repeat this process until all elements in $w_{1,1}, ..., w_{1,m}, w_{2,1}, ..., w_{2,m}$ have been chosen by the verifier. Particularly, we want to show the last iteration. The prover computes univariate polynomial (fixing $w_{1,1}, ..., w_{1,m}, w_{2,1}, ..., w_{2,m-1}$)

$$\tilde{\alpha}_{i-1,2m}(x) = f_z(w_{1,1}, ..., w_{1,m}, w_{2,1}, ..., w_{2,m-1}, x)$$

and sends it to the verifier, the verifier checks if

$$\sum_{x \in \mathbb{H}} \tilde{\alpha}_{i-1,2m}(x) = r_{i-1,2m-1}.$$

If the check passes, the verifier picks the last element $w_{2,m} \in \mathbb{F}$ uniformly at random, and send it to the prover. The verifier defines

$$r_{i-1,2m} := \tilde{\alpha}_{i-1,2m}(w_{2,m}).$$

Therefore, over all the iterations within the sum-check protocol, the verifier has picked random $\vec{w}_1, \vec{w}_2 \in \mathbb{F}^m$ bit by bit.

Finally, the verifier wants to check if

$$f_z(\vec{w}_1, \vec{w}_2) = r_{i-1,2m}.$$

Substituting the definition of f_z in (2), the verifier wants to check if

$$\tilde{\phi}_i(\vec{z}_{i-1}, \vec{w}_1, \vec{w}_2) \cdot \tilde{\text{NAND}}(\tilde{\alpha}_i(\vec{w}_1), \tilde{\alpha}_i(\vec{w}_2)) = r_{i-1,2m}. \tag{4}$$

Note that $\tilde{\phi}_i$ is given as an oracle, which is a low-degree polynomial that specifies the circuit structure. $\tilde{\text{NAND}}$ is also a low-degree polynomial which can be computed efficiently. So it comes down to computing $\tilde{\alpha}_i(\vec{w}_1)$ and $\tilde{\alpha}_i(\vec{w}_2)$.

Let's summarize what we have done so far. In the beginning of the ith phase, our goal is to check $\tilde{\alpha}_{i-1}(\vec{z}_{i-1}) = r_{i-1}$. By running a sum-check protocol, the verifier picks random $\vec{w_1}, \vec{w}_2 \in \mathbb{F}^m$ such that the task is reduced to checking (4). Now the prover sends two values v_1, v_2 to the verifier and claiming they are the values of $\tilde{\alpha}_i(\vec{w}_1)$ and $\tilde{\alpha}_i(\vec{w}_2)$ respectively. The verifier can first check if (4) holds by plugging in v_1, v_2 sent by the prover, and reject if (4) does not hold. However, the verifier cannot trust that the two values sent by the prover are the true $\tilde{\alpha}_i(\vec{w}_1)$ and $\tilde{\alpha}_i(\vec{w}_2)$.

Now we switch the notation from $\vec{z}_{i-1}, \vec{w}_1, \vec{w}_2 \in \mathbb{F}^m$ back to $z_{i-1}, w_1, w_2 \in \mathbb{F}^m$. So far, we have reduced the task of checking $\tilde{\alpha}_{i-1}(z_{i-1}) = r_{i-1}$ to checking if $\tilde{\alpha}_i(w_1) = v_1$ and $\tilde{\alpha}_i(w_2) = v_2$. This is still not good, as the number of points the verifier needs to ask still grows exponentially with the depth of the circuit. We eventually want to reduce checking that $\tilde{\alpha}_{i-1}(z_{i-1}) = r_{i-1}$ to checking a single point $\tilde{\alpha}_i(z_i) = r_i$. The GKR paper introduces a protocol that reduces checking two points to checking one point, which is commonly referred as "2-to-1 trick". Note that the authors did not invent the 2-to-1 trick. It has been used commonly in many PCP proofs, but the application of the 2-to-1 trick in this context was credited to GKR.

The 2-to-1 trick We want to reduce from proving $\tilde{\alpha}_i(w_1) = v_1$ and $\tilde{\alpha}_i(w_2) = v_2$ to proving a single point $\tilde{\alpha}_i(z_i) = r_i$. Here is how it goes:

1. Let $t_1, t_2 \in \mathbb{F}$ be two distinct fixed elements known to both the prover and the verifier. The prover and the verifier can determine t_1, t_2 in the beginning of the whole GKR protocol. Think of $t_1, t_2 \in \mathbb{F}$ corresponds to the fixed number 0 and 1 respectively. Let $\gamma : \mathbb{F} \to \mathbb{F}^m$ be the unique line (i.e. polynomial of degree at most 1) such that $\gamma(t_1) = w_1$ and $\gamma(t_2) = w_2$. We know two points determine a line, so the line γ is unique. For example, we can define it as $\gamma = \{(1-t) \cdot w_1 + t \cdot w_2\}_{t \in \mathbb{F}}$. Note that the domain of γ is $\mathbb{F}$, so there are another $|\mathbb{F}| - 2$ elements other than t_1, t_2 that maps to some points in $\mathbb{F}^m$. Note that γ can be computed by both the prover and the verifier in time $\text{polylog}(|\mathbb{F}|, m)$ (since we assume field operations can be done in $\text{polylog}(|\mathbb{F}|)$) and space $O(\log(|\mathbb{F}|) \cdot m)$.
2. The prover sends the function $f := \tilde{\alpha}_i \circ \gamma : \mathbb{F} \to \mathbb{F}$ to the verifier. The function composition $\tilde{\alpha}_i \circ \gamma : \mathbb{F}$ first applies γ then applies $\tilde{\alpha}_i$.
3. Upon receiving f from the prover, the verifier checks that if f is a polynomial of degree at most $m \cdot \delta$ (the degree comes from the $\tilde{\phi}_i$) and that $f(t_1) = v_1$ and $f(t_2) = v_2$. If these tests pass, then the verifier chooses a random element $t \in \mathbb{F}$ and send it to the prover.

4. The prover and the verifier continue to Phase $i + 1$ with $z_i = \gamma(t)$ and $r_i = f(t)$.

5.2 The dth phase and final verification

The dth layer is the input layer. In the $d$the phase, we want to reduce the task of checking that $\tilde{\alpha}_{d-1}(z_{d-1}) = r_{d-1}$ to checking $\tilde{\alpha}_d(z_d) = r_d$. We still do the sum-check protocol and the 2-to-1 trick. The key difference here is $\tilde{\alpha}_d$, which is the low-degree extension of the dth layer. There is no gates in the dth layer, thus we do not need the $\tilde{\phi}$ oracle. The honest prover should compute $\tilde{\alpha}_d$, i.e. the low-degree extension of x (padded with zeros such that the length is S), in the standard way. This means the polynomial $\tilde{\alpha}_d$ has individual degree at most $|\mathbb{H}| - 1$ so it is unique. When the verifier picks a t as the last step of the 2-to-1 trick, we have $z_d = \gamma(t) \in \mathbb{F}^m$ and $r_d = f(t) \in \mathbb{F}$.

Here is the final verification. The verifier knows z_d and r_d. The verifier computes the low-degree extension $\tilde{x}$ of x (padded with zeros) himself (with respect with $\mathbb{H}, \mathbb{F}, m$). Note that $\tilde{x}$ should have individual degree at most $|\mathbb{H}| - 1$ so the low-degree extension is unique. This is the heaviest computation for the verifier in the entire GKR protocol. And the verifier checks that if $\tilde{x}(z_d) = r_d$. Since the low-degree extension is unique, $\tilde{x}$ should be exactly the same as $\tilde{\alpha}_d$ if $\tilde{\alpha}_d$ is computed by an honest prover.

Note that if the verifier has access to an oracle that gives the low-degree extension of x, then the verifier just needs a single query on z_d.

6 Proof of Soundness

Note that completeness is trivial since if the prover honestly computes the circuit and honestly computes every polynomial in the sum-check protocol and 2-to-1 trick, the verifier should accept with probability 1. We mainly prove the soundness here.

Suppose that $C(x) = 1$ and there exists a cheating prover $\mathcal{P}^*$ such that

$$\Pr[(\mathcal{P}^{*\mathcal{F}}, \mathcal{V}^{\mathcal{F}}) = 1] = s$$

for some $0 < s < 1$. We would like to show $s \leq \frac{1}{100}$ as claimed in Theorem 1.

The protocol consists of d phases. Each phase consists of a sum-check followed by a 2-to-1 trick. We define the events below:

- Let A denote the event $(\mathcal{P}^{*\mathcal{F}}, \mathcal{V}^{\mathcal{F}}) = 1$, i.e., the verifier eventually accepts.
- Let T_i denote the event that indeed $\tilde{\alpha}_i(z_i) = r_i$, where $0 \leq i \leq d$. Thus, $C(x) \neq 0$ means $\neg T_0$. Note that $\tilde{\alpha}_i(z_i)$ means the true polynomial for layer i computed by an honest prover. The cheating prover will give the verifier a fake polynomial $\tilde{g}_i$ (actually $\tilde{g}_{i,0}, ..., \tilde{g}_{i,2m}$ in the sum-check, and $\tilde{g}_i \circ \gamma$ in the 2-to-1 trick).

– Let E_i denote the event that indeed $\tilde{\alpha}_i(w_1) = v_1$ and $\tilde{\alpha}_i(w_2) = v_2$, for $i \in [d]$. A cheating prover can send v_1, v_2 such that it matches $\tilde{g}_i(w_1) = v_1$ and $\tilde{g}_i(w_2) = v_2$ but not $\tilde{\alpha}_i(w_1) = v_1$ and $\tilde{\alpha}_i(w_2) = v_2$. The verifier does not know the true $\tilde{\alpha}_i$.

Note that

$$s = \Pr[A \wedge \neg T_0 \wedge T_d] \leq \Pr[\exists i \in [d], A \wedge \neg T_{i-1} \wedge T_i] \leq \sum_{i=1}^{d} \Pr[A \wedge \neg T_{i-1} \wedge T_i]$$

where $A \wedge \neg T_0 \wedge T_d$ means that the verifier accepts, and the true polynomial $\tilde{\alpha}_0(z_0)$ should be 1, but the prover gives a fake one $\tilde{g}_0(z_0)$ that equals 0. However, the last unique low-degree extension is also computed by the verifier, where the prover cannot cheat, so we have the event T_d. We must have T_d for the verifier to accept.

The event $A \wedge \neg T_0 \wedge T_d$ has probability less than or equal to the event $\exists i \in [d], A \wedge \neg T_{i-1} \wedge T_i$: at some phase i, the true polynomial $\tilde{\alpha}_{i-1}(z_{i-1}) \neq r_{i-1}$ but the prover sends a fake one $\tilde{g}_{i-1}(z_{i-1}) = r_{i-1}$ and the verifier accepts since the prover gets lucky that $\tilde{\alpha}_{i-1}$ and $\tilde{g}_{i-1}$ happen to agree on z_{i-1}, so the prover cheats successfully in this phase such that in the following phases the prover does not need to lie and just honestly computes the true polynomials $\tilde{\alpha}_i$ following that path.

By law of total probability, we have

$$\Pr[A \wedge \neg T_{i-1} \wedge T_i] = \Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge E_i] + \Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge \neg E_i] \quad (5)$$

For the $\Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge E_i]$ part, note that

$$\Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge E_i] \leq \Pr[A \wedge \neg T_{i-1} \wedge E_i]$$

The event $A \wedge \neg T_{i-1} \wedge E_i$ means that the true polynomial $\tilde{\alpha}_{i-1}(z_{i-1}) \neq r_{i-1}$ so the cheating prover keeps sending fake polynomials in the sum-check protocol, and at the end of the sum-check protocol, the prover sends v_1, v_2 where $\tilde{\alpha}_i(w_1) = v_1$ and $\tilde{\alpha}_i(w_2) = v_2$. I.e. the prover successfully cheats the sum-check protocol. The soundness of the sum-check protocol implies that

$$\Pr[A \wedge \neg T_{i-1} \wedge E_i] \leq \frac{4m\delta}{|\mathbb{F}|} \quad (6)$$

since f_z has individual degree at most 2δ and f_z has $2m$ variables. By the Schwartz-Zippel lemma, we have the probability $\leq \frac{2m \cdot 2\delta}{|\mathbb{F}|}$.

For the $\Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge \neg E_i]$ part, note that

$$\Pr[A \wedge \neg T_{i-1} \wedge T_i \wedge \neg E_i] \leq \Pr[A \wedge T_i \wedge \neg E_i] \leq \frac{m\delta}{|\mathbb{F}|}. \quad (7)$$

It is better to read this event as $A \wedge \neg E_i \wedge T_i$. Note that $\neg E_i$ is $\tilde{\alpha}_i(w_1) \neq v_1$ OR $\tilde{\alpha}_i(w_2) \neq v_2$. So the cheating prover is giving a different polynomial $\tilde{g}_i \neq \tilde{\alpha}_i$

such that $\tilde{g}_i(w_1) = v_1$ and $\tilde{g}_i(w_2) = v_2$. And T_i means $\tilde{\alpha}_i(z_i) = r_i$ so $\tilde{g}_i$ needs to agree with $\tilde{\alpha}_i$ on z_i pass the check. Thus, the probability bound is due to any fake polynomial $\tilde{g}_i$ with degree $m\delta$ can agree with the true polynomial $\tilde{\alpha}_i$ on at most $m\delta$ points.

Plugging (6) and (7) into (5), we get

$$\Pr[A \wedge \neg T_{i-1} \wedge T_i] \leq \frac{4m\delta}{|\mathbb{F}|} + \frac{m\delta}{|\mathbb{F}|} \leq \frac{5m\delta}{|\mathbb{F}|}.$$

Hence, by union bound on all d phases, we get that

$$s = \Pr[A \wedge \neg T_0 \wedge T_d] \leq \frac{5md\delta}{|\mathbb{F}|}.$$

Taking $\mathbb{F}$ such that $|\mathbb{F}| \geq 500md\delta = \text{poly}(|\mathbb{H}|)$, we get $s \leq \frac{1}{100}$ as stated in Theorem 1.

7 Complexity of the Protocol

We analyse the complexity in a single phase. Suppose $d = \text{polylog} n$ but we can also generalize.

1. The prover needs to compute the polynomials in the sum-check protocol which takes $\text{poly}(|\mathbb{F}|^m) = \text{poly}(S)$ time.
2. The verifier runs in time $\text{poly}(|\mathbb{H}|, \log|\mathbb{F}|, m)$ both in the sum-check protocol and the 2-to-1 trick. We take $|\mathbb{H}| = n^{0.01}$ and $|\mathbb{F}| = \text{poly}(|\mathbb{H}|)$ and we make sure $|\mathbb{F}| = o(n)$ (e.g. $|\mathbb{F}| = n^{0.2}$, so m is a big constant.
   The space used by the verifier is $O(\log(|\mathbb{F}|) \cdot m)$, both in sum-check and 2-to-1 trick. Going to the next phase, the verifier only needs to remember i, z_i, r_i, which is also an $O(\log(|\mathbb{F}|) \cdot m)$ overhead.

8 Golderich's construction

Constructing the oracle efficiently:

Let C_n be a circuit that computes some function f. We want to construct a polynomial $\hat{\phi}$ that encodes the circuit structure of C_n. Here is the requirement for our construction:

1. $\hat{\phi}(\vec{w}, \vec{u}, \vec{v}) = 1[u, v \text{ feed into } w]$ is a polynomial that encodes the circuit structure.
2. $\deg(\hat{\phi})$ is low
3. $\hat{\phi}$ can be computed in linear time (say n where n is the size of input).

It would be ideal if we could do it for P-uniform circuits, but people still don't know how to do it. However, people knows how to do it for logspace-uniform circuit. Let $\mathcal{M}$ be the logspace TM that on input 1^n constructs the circuit. The circuit will have depth d and size S where $S = \text{poly}(n)$.

Here is how we do it. First, we take the matrix M of of the TM $\mathcal{M}$. This is the matrix where each row is a configuration of the TM. Since the TM is logspace, the number of rows will be poly(n). Each column is also a configuration of the TM, and the entry $M[i, j] = 1$ means that configuration i goes to configuration j in the next step of the TM's execution. Since this is a deterministic TM, each configuration can only go to 1 next configuration. So each row has only one 1. Now consider $((M^2)^2)^2$.. squaring the matrix $O(\log n)$ times, this will reach the final configuration of the execution. We only care about the first row, which the row indicates the initial configuration, and the column that has 1 in it indicates the final configuration. In this final configuration, it has the circuit C_n written on the output tape (since the TM's task is to output the circuit on the tape). Use a universal circuit U_n to combine C_n and x such that $U_n(C_n, x) = C_n(x)$. Call the final combined circuit C'_n. Then we have three properties of C'_n:

1. C'_n computes the same function as C_n on x
2. the size and depth of C'_n not so much bigger than the size and depth of C_n
3. the circuit is extremely-uniform.

Since C'_n is "super-duper-ultra-uniform", there exists a Boolean formula of size polylog(n) computing $\phi(w, u, v) : [S^2]^3 \to \{0, 1\}$, then it is easy to construct a low-degree extension $\hat{\phi} : \mathbb{F}^6 \to \mathbb{F}$.

9 A proof of IP=PSPACE

Note that this proof can also be converted to a proof of IP = PSPACE. For the time sake, we leave it as an exercise.

10 Acknowledgement

I would like to thank Prof. Roei Tell to carefully and patiently explain every bit of detail of the GKR paper to me.

References

1. Goldwasser, Shafi; Kalai, Yael Tauman; Rothblum, Guy N. Delegating computation: interactive proofs for muggles. *STOC'08, 113–122, ACM, New York,* 2008.
2. Goldwasser, Shafi; Kalai, Yael Tauman; Rothblum, Guy N. Delegating computation: interactive proofs for muggles. https://eccc.weizmann.ac.il/report/2017/108/
3. Tell, Roei. Mutilinear and Low-Degree Extensions. Unpublished manuscript: https://sites.google.com/site/roeitell/Expositions (2019)