MTAT.07.004 — Complexity Theory December 8th, 2011

Lecture 12

Lecturer: Peeter Laud *Scribe(s): Ilya Kuzovkin*

Introduction

On the previous lecture we had a look onto interactive proofs, where the system consists of two parts Prover (P) and Verifier (V). Prover is computationally unbounded while Verifier is polynomial-time bounded. Basic scheme of interaction is

1. *We* provide the system with yes/no question
2. Verifier maps this question to some form Prover will accept and adds randomization
3. Verifier sends challenge to Prover
4. Prover responds with certificate
5. Verifier checks certificate and may proceed or repeat steps 2-5 to be more sure in Prover's capability of finding the answer
6. Verifier gives us yes/no answer

This basic scheme describes complexity class IP (Interactive Proof). For formal definition please refer to the previous lecture's notes. There is slight variation of the IP protocol called Arthur-Merlin (AM) protocol. Arthur plays the Verifier role and Merlin is the omniscient Prover. The only difference is that in AM protocol Prover knows the randomization factor used by Verifier in step 2. Intuitively it seems that Merlin from AM protocol should be able to solve wider range of problems, since he has more information than Prover from IP protocol.

But we will see that actually these complexity classes are equal.

IP = AM

We will say *Prover* and *Verifier* when talking about IP protocol and *Merlin* and *Arthur* when talking about AM protocol.

Theorem 1 *IP = AM*

Proof
To show that classes are equal we must show both way inclusions.

AM $\subseteq$ IP

We have a problem in AM and we want to simulate it in IP. To do so it is enough if Verifier will send to Prover all the randomization data he uses.

IP $\subseteq$ AM

The opposite direction is more complicated. Here we have problem in IP and we want to simulate it in AM. Since Merlin knows all the randomization data he can fool Arthur with less effort than Prover would need to fool Verifier.

On the figure below you can see initial protocol in IP. To show that on every step on the simulation we are very close to the original protocol we define time limit T and assume that our protocol runs until limit is reached.

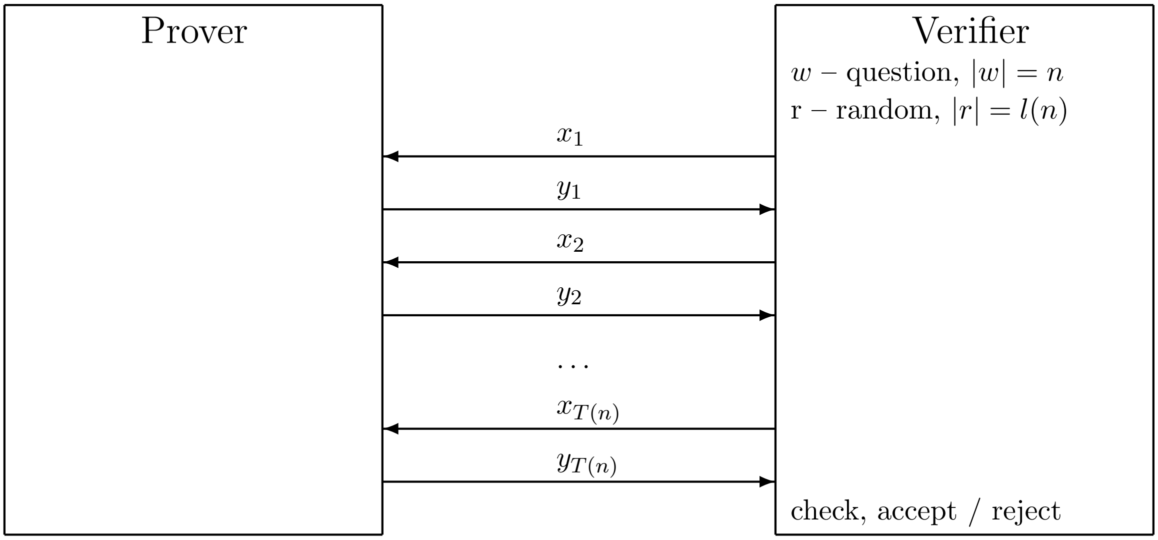

Figure 1: Initial protocol in IP

The sequence of the messages $(x_1, y_1, x_2, y_2, ..., x_n, y_n)$ we call *transcript*. For each r there is a distinct transcript. Each transcript ends with accept or reject.

Definition 1 *At some point of time τ we will have partial transcript t, we define $ACC(w, t)$ to be a set of all transcripts which start with t and lead to accept.*

Now we can construct AM simulation for the protocol described on Figure 1. The idea is that after each round of the protocol (one x_i, y_i pair) Merlin provides best y_i and wants to convince Arthur that $ACC(w, t_i)$ covers almost all set of outcomes (e.g. almost all outcomes are *accept*).

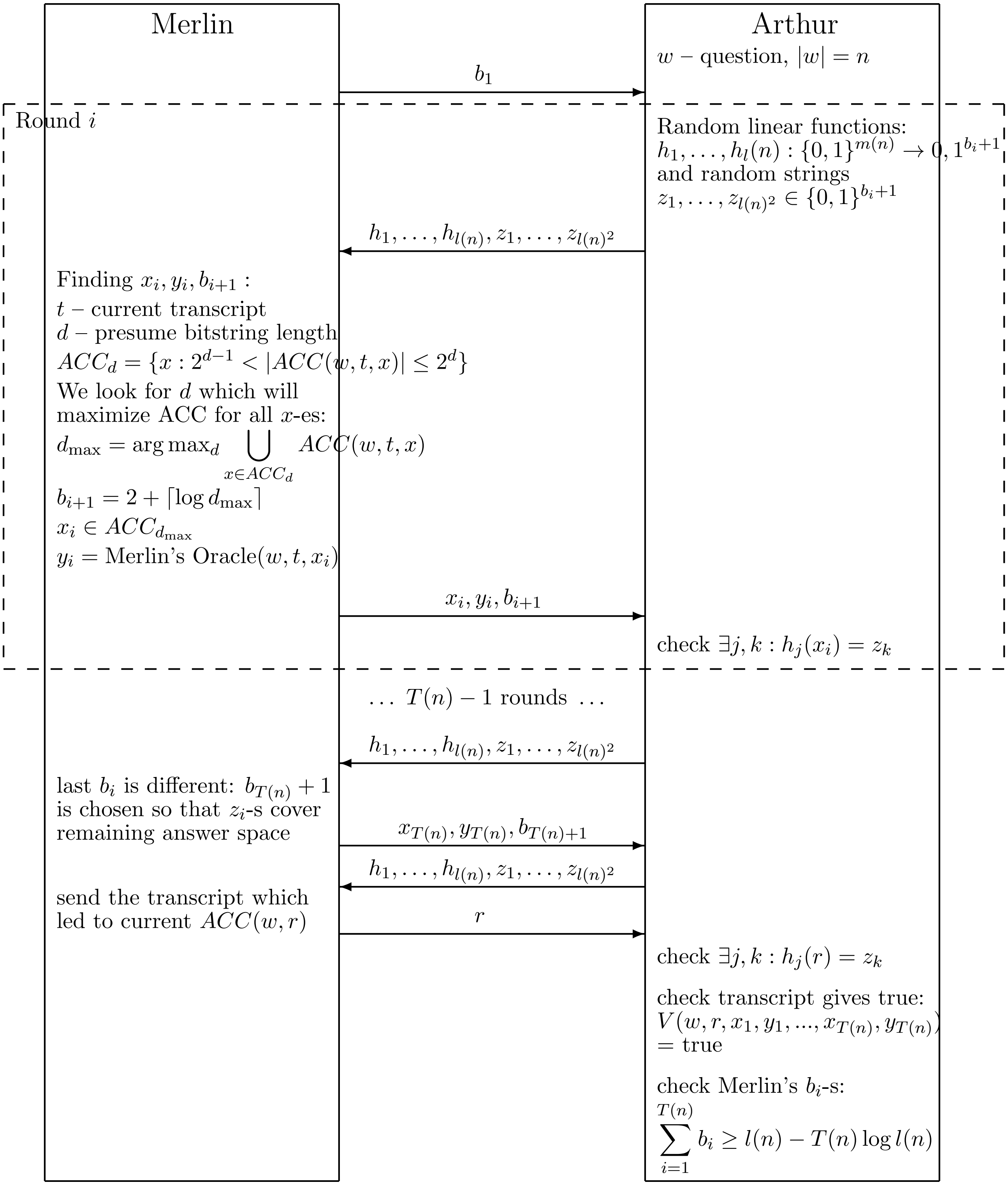
Merlin
Arthur
w – question, $|w| = n$
b_1
Round i
Random linear functions:
$h_1, \ldots, h_l(n) : \{0,1\}^{m(n)} \to 0, 1^{b_i+1}$
and random strings
$z_1, \ldots, z_{l(n)^2} \in \{0,1\}^{b_i+1}$
$h_1, \ldots, h_{l(n)}, z_1, \ldots, z_{l(n)^2}$
Finding x_i, y_i, b_{i+1} :
t – current transcript
d – presume bitstring length
$ACC_d = \{x : 2^{d-1} < |ACC(w,t,x)| \leq 2^d\}$
We look for d which will
maximize ACC for all x-es:
$d_{\max} = \arg\max_d \bigcup_{x \in ACC_d} ACC(w,t,x)$
$b_{i+1} = 2 + \lceil \log d_{\max} \rceil$
$x_i \in ACC_{d_{\max}}$
$y_i =$ Merlin's Oracle(w,t,x_i)
x_i, y_i, b_{i+1}
check $\exists j,k : h_j(x_i) = z_k$
$\ldots\ T(n) - 1$ rounds $\ldots$
$h_1, \ldots, h_{l(n)}, z_1, \ldots, z_{l(n)^2}$
last b_i is different: $b_{T(n)} + 1$
is chosen so that z_i-s cover
remaining answer space
$x_{T(n)}, y_{T(n)}, b_{T(n)+1}$
$h_1, \ldots, h_{l(n)}, z_1, \ldots, z_{l(n)^2}$
send the transcript which
led to current $ACC(w,r)$
r
check $\exists j,k : h_j(r) = z_k$
check transcript gives true:
$V(w, r, x_1, y_1, ..., x_{T(n)}, y_{T(n)})$
= true
check Merlin's b_i-s:
$\sum_{i=1}^{T(n)} b_i \geq l(n) - T(n) \log l(n)$

Figure 2: Protocol in AM

Intuitively this protocol acts in the same way as IP one, but here instead of Verifier picking next x_i randomly, Merlin himself chooses the best x_i. And after protocol is finished the r is the transcript which can act as certificate in original IP protocol.

Why Arthur needs to check $\exists j, k : h_j(x_i) = z_k$ in the end of each round? The reason is that Merlin must convince Arthur that the answer space, where ACC set is defined is big enough. And the way he convinces is by showing that for randomly chosen h_j and z_k Merlin is able to find original in the set of answers.

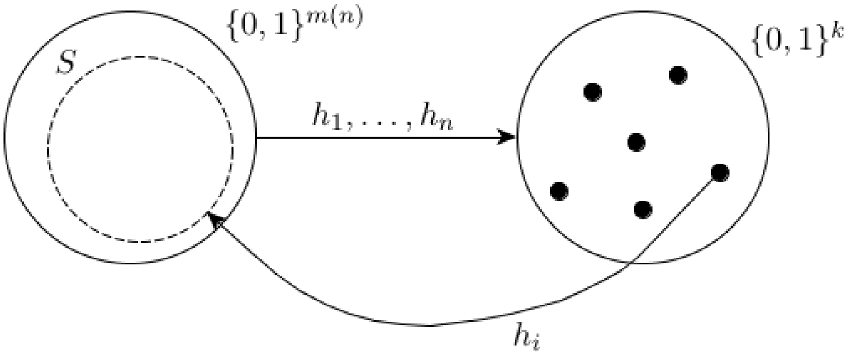

Figure 3: Merlin should be able to find original of random z_j in S

This whole construction must convince us that after running protocol in AM final transcript r will be suitable transcript for IP. ■

IP = PSPACE

In this section we will show that every problem, which can be solved in IP can be solved in PSPACE and vice versa. As usually, if we want to show equality of the classes we will need to show both way inclusions.

Theorem 2 *IP = PSPACE*

Proof

IP ⊆ PSPACE

Every problem solvable in IP is solvable in PSPACE. As we know from the previous section IP = AM, therefore AM ⊆ PSPACE also. Assume we have usual AM protocol:

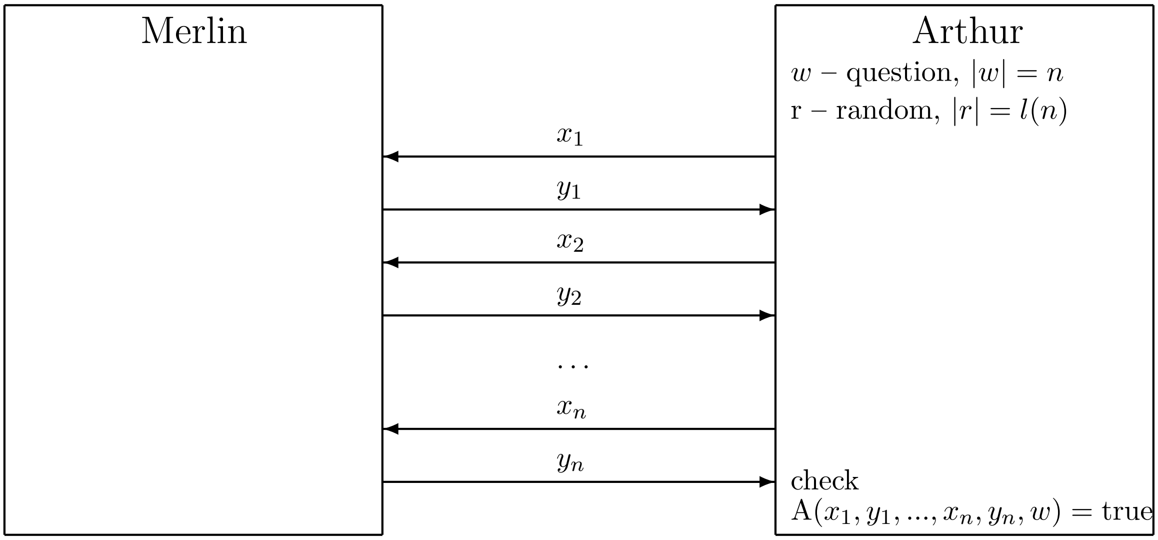

Figure 4: Abstract AM protocol

Assume there is a function

$$f : \{0,1\}^{l(n)} \rightarrow \mathbb{R}$$

and

$$\max f = \max\{f(x) : x \in \{0,1\}^{l(n)}\}$$

$$\text{avg} f = \frac{\sum_{x\in\{0,1\}^{l(n)}}}{2^{l(n)}}$$

At each round of the protocol Arthur randomly chooses x_i and Merlin choose best y_i, so the protocol's whole computation can be written as

$$\text{avg}_{x_1}\text{max}_{y_1}\ldots\text{avg}_{x_n}\text{max}_{y_n}A(x_1, y_1, \ldots, x_n, y_n, w)$$

By definition we know that $A(x_1, y_1, \ldots, x_n, y_n, w)$ is computable in polynomial time (otherwise Arthur would not be able to perform final check). This implies that sizes of x_i and y_i are polynomial space(otherwise computation of $A(...)$ would take more than polynomial time). These two facts together show us that whole computation is doable in polynomial space.

PSPACE $\subseteq$ IP

Here we must show that every problem solvable in PSPACE is solvable in IP. We will use #SAT$_D$ problem to do that:

1. #SAT$_D$ is NP-hard

2. TQBF is PSPACE-complete

3. Show $\#\mathrm{SAT}_D \in IP$ – we will use it
4. Using 3. show TQBF is in IP
5. If we show that TQBF $\in$ IP then we can solve any PSPACE problem in IP via TQBF (using parts of the proof $\#\mathrm{SAT}_D \in$ IP)
6. 5 $\Rightarrow$ PSPACE $\subseteq$ IP

So it is essential for us to show that $\#\mathrm{SAT}_D \in$ IP.

Arithmetization

We will now deviate from the main direction of the proof and study idea of arithmetization, since we will use it later.

If we have Boolean formula $\varphi(x_1, \dots, x_n)$ we can transform it into n-variable polynomial as follows:

$$P_{x_i} = x_i$$
$$P_{\neg\varphi} = 1 - P_\varphi$$
$$P_{\varphi_1 \wedge \varphi_2} = P_{\varphi_1} \cdot P_{\varphi_2}$$
$$P_{\varphi_1 \vee \varphi_2} = 1 - (1 - P_{\varphi_1})(1 - P_{\varphi_2})$$

For example formula

$$(x_1 \vee x_2) \wedge \neg x_1$$

can be arithmetized as

$$(1 - (1 - x_1)(1 - x_2)) \cdot (1 - x_1)$$

Note that length on the polynomial is linear to length of the Boolean formula: $O(P_\varphi)$ is $O(|\varphi|)$.

$\#\mathbf{SAT_D} \in \mathbf{IP}$

$\#\mathrm{SAT}_D$ answers the question if formula has exactly K satisfying valuations. Number of satisfying valuations can be calculated using

$$\#\varphi = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_\varphi(b_1, \dots, b_n) \leq 2^n$$

In more general case (for any polynomial g) this can be computed using

$$K = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_m \in \{0,1\}} g(b_1, \dots, b_m)$$

where $g(b_1, \dots, b_n)$ can be computed recursively using the following definition

$$proj_g(x) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_m \in \{0,1\}} g(x, b_2, \dots, b_m)$$

and the base

$$g(0) + g(1) = K$$

this can be done in IP, using Sumcheck protocol below.

Sumcheck protocol

Here is the IP protocol for verifying

$$K = \sum_{b_1 \in \{0,1\}} \dots \sum_{b_m \in \{0,1\}} g(b_1, \dots, b_m)$$

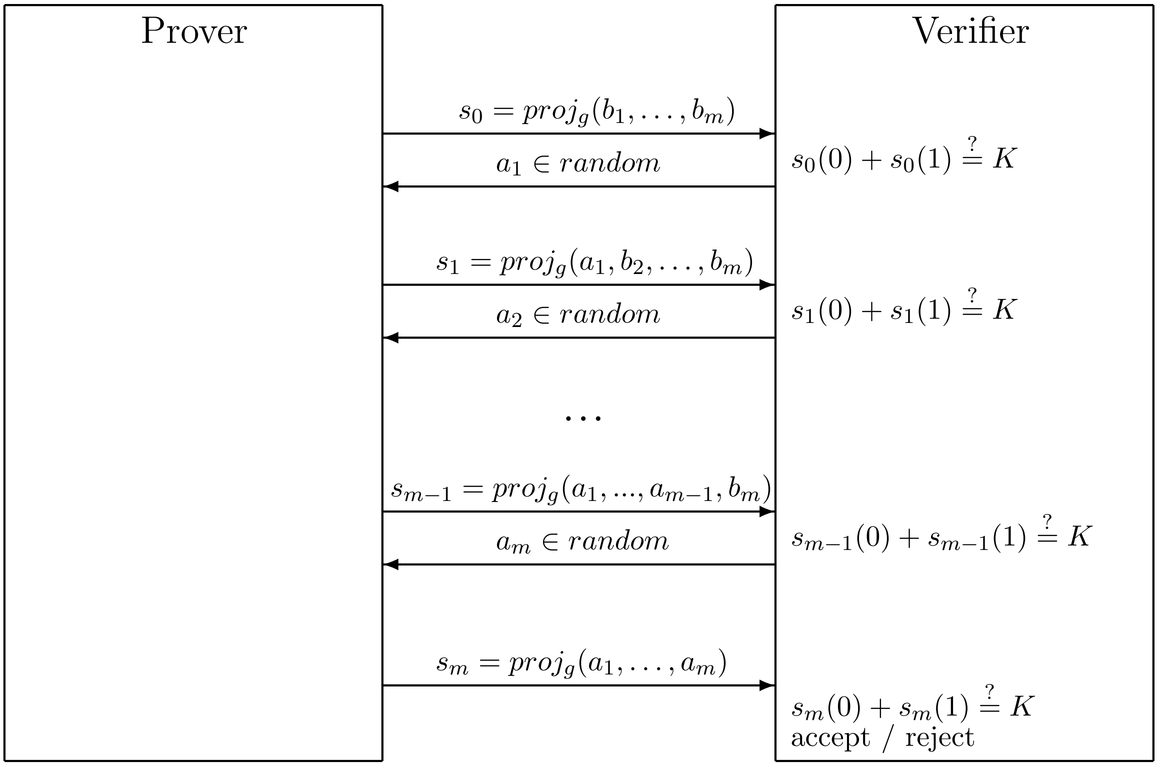
Prover
Verifier
$s_0 = proj_g(b_1, \dots, b_m)$
$a_1 \in random$
$s_0(0) + s_0(1) \overset{?}{=} K$
$s_1 = proj_g(a_1, b_2, \dots, b_m)$
$a_2 \in random$
$s_1(0) + s_1(1) \overset{?}{=} K$
$\dots$
$s_{m-1} = proj_g(a_1, ..., a_{m-1}, b_m)$
$a_m \in random$
$s_{m-1}(0) + s_{m-1}(1) \overset{?}{=} K$
$s_m = proj_g(a_1, \dots, a_m)$
$s_m(0) + s_m(1) \overset{?}{=} K$
accept / reject

Figure 5: Sumcheck protocol

Distinct case is then $m = 1$, then Verifier will check only $g(0) + g(1) = K$

TQBF $\in$ IP

The proof works in the same way as for #SAT$_D$, but since TQBF includes universal quantifiers the formula will contain products. We will introduce prodcheck protocol, which works in the same way as sumcheck, but polynomials are more complex. Because degree of the polynomials are too large (exponential) Verifier would not be able to work with them, so we must provide the optimization steps to make polynomial degrees low.

Optimizing TQ formula

To make Prodcheck computation more efficient we can bring in new variables, which will allow us to move quantifiers closer to the variables bounded to these quantifiers. For example in the following formula

$$\forall x_2 \forall x_3 (\forall x_1 (x_1 \wedge x_2)) \vee x_3$$

which can be represented as syntax tree

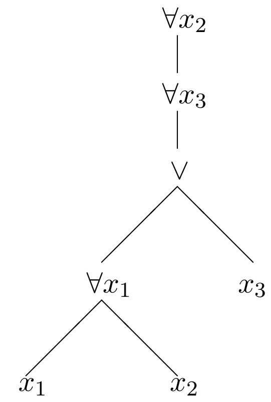

Figure 6: Syntax tree for $\forall x_2 \forall x_3 (\forall x_1 (x_1 \wedge x_2)) \vee x_3$

we can introduce $x_2' = x_2$ which will allow us to bring $\forall x_2$ closer to variable x_2:

$$\forall x_2 \forall x_3 \exists x_2' (x_2' = x_2 \wedge \forall x_1 (x_1 \wedge x_2') \vee x_3)$$

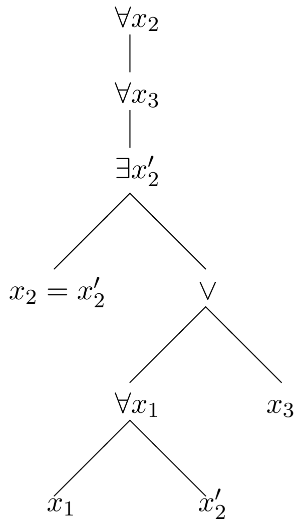

Figure 7: Syntax tree for $\forall x_2 \forall x_3 \exists x_2' (x_2' = x_2 \wedge \forall x_1 (x_1 \wedge x_2') \vee x_3)$

After we move quantifiers closer to the variables we can bring all negations to the front of variables. After that we can arithmetize quantifiers:

$$P_{\exists x \varphi} = \sum_{b \in \{0,1\}} P_\varphi (x \in b)$$

$$P_{\forall x \varphi} = \prod_{b \in \{0,1\}} P_\varphi (x \in b)$$

After these transformation we will get P_φ which can be used in the prodcheck protocol to check if P_φ has K satisfiable evaluations.

Now degree of polynomial is at most $2|\varphi|$.

Prodcheck

As we have mentioned before the scheme of the protocol is the same as for sumcheck protocol. But there are some differences.

- If $\varphi \equiv \varphi_1 \vee \varphi_2$ then Prover sends both values $K_1 = P_{\varphi_1}$ and $K_2 = P_{\varphi_2}$. Verifier checks that $K = K_1 + K_2$ and then runs the protocol for $K_1 \stackrel{?}{=} P_{\varphi_1}$ and $K_2 \stackrel{?}{=} P_{\varphi_2}$
- If $\varphi \equiv \varphi_1 \wedge \varphi_2$ then Prover sends both values $K_1 = P_{\varphi_1}$ and $K_2 = P_{\varphi_2}$. Verifier checks that $K = K_1 \cdot K_2$ and then runs the protocol for $K_1 \stackrel{?}{=} P_{\varphi_1}$ and $K_2 \stackrel{?}{=} P_{\varphi_2}$
- If $\varphi \equiv \exists x \varphi'$ then Prover sends the polynomial $s(x) = P_{\varphi'}(x)$ to Verifier. Verifier checks $s(0) + s(1) \stackrel{?}{=} K$, then picks number $a \in$ random and run the protocol $P_{\varphi'}(a) \stackrel{?}{=} s(a)$
- If $\varphi \equiv \forall x \varphi'$ then Prover sends the polynomial $s(x) = P_{\varphi'}(x)$ to Verifier. Verifier checks $s(0) \cdot s(1) \stackrel{?}{=} K$, then picks number $a \in$ random and run the protocol $P_{\varphi'}(a) \stackrel{?}{=} s(a)$

■

Relationship between PSPACE, P/*poly* and MA

Theorem 3 *If PSPACE ⊆ P/poly then PSPACE = MA*

Proof We use the fact that Merlin can work in PSPACE. PSPACE ⊆ P/poly then Merlin can be represented as Boolean circuit of polynomial size, thus Merlin's algorithm can be sent to Arthur. And Arthur can himself run Merlin's algorithm in polynomial time (and polynomial space), thus solving any PSPACE problem in MA. ■