6.841 Advanced Complexity Theory	April 6, 2005
Lecture 17	
Lecturer: Madhu Sudan	*Scribe: Alexey Spiridonov*

1 $PSPACE \subseteq IP$

We will show this inclusion using straight-line programs of polynomials (SPP). First, we will show that $SPP \subseteq IP$, and then $PSPACE \subseteq SPP$. In the previous lecture, we showed that the permanent has an interactive proof, and so $IP \subseteq PSPACE$. Putting the two together, we will get $IP = PSPACE$. Before carrying out these proofs, we need to introduce SPP.

1.1 Straight-line programs of polynomials

Let's start with a quick refresher of straight-line programs (SP). Suppose we have input bits $x_1, \ldots x_n$. Then, a straight-line program is a computation of the following form:

$$\begin{aligned} y_1 &\leftarrow x_1 + x_3 \\ y_2 &\leftarrow y_1 \times x_2 \\ y_3 &\leftarrow y_1 + y_2 \\ &\cdots \\ y_i &\leftarrow y_j \times y_k \text{ with } j, k < i, \end{aligned}$$

where y_i might alternatively use some of the inputs x_l, or use $+$ instead of $\times$.

We can rewrite this idea to use polynomials instead of intermediate variables. Let x be an n-bit string consisting of $x_1, \ldots, x_n$. Then, we can define the following polynomials:

$$\begin{aligned} p_1(x) &\leftarrow x_1 \\ p_2(x) &\leftarrow x_2 \times p_1(x) \\ p_3(x) &\leftarrow p_1(f_3(x)) + p_2(g_3(x)) \\ &\cdots \\ p_i(x) &\leftarrow p_k(f_i(x)) + p_j(g_i(x)), \end{aligned} \tag{1}$$

where f_i, g_i are easy-to-compute ($poly(n)$ time) functions from $\{0,1\}^n \to \{0,1\}^n$. Except for one adjustment, this is how we get SPP.

The motivation for introducing these polynomials is that we would like to emulate the proof that $\#P \subseteq IP$. That proof operates on the following principle: for a complete verification, we'd need to compute a 2^n-sized formula

$$\sum_{x \in \{0,1\}^n} f(x_1, x_2, \ldots, x_n) = \sum_{\tilde{x} \in \{0,1\}^{n-1}} f(0, x_2, \ldots, x_n) + \sum_{\tilde{x} \in \{0,1\}^{n-1}} f(1, x_2, \ldots, x_n)$$

(with $f = 1$ on satisfying assignments, and 0 otherwise). Instead, we rewrite these sums as low-degree polynomials formulas over a field $\mathbb{F}$ (having $0, 1, -1$ all distinct): $p = p'(0) + p'(1)$; now, instead of verifying both the values of $p'(0)$ and $p'(1)$ to check the value of p, we evaluate p' at a random value z. This leaves only $O(n)$ checks to be made, each one of which is polynomial in n.

The idea of SPP is to retain this recursive, efficiently checkable structure, while having enough power to perform $PSPACE$ computations. There is one problem with the formulation in equation

1. The ith polynomial references arbitrary polynomials with indices below i. Thus, we cannot combine verifications, and the verifier migh need to verify, say, p_1 an exponential number of times. We fix this by requiring that p_i only reference p_{i-1}. Then, we can use the same method as before to show $SPP \subseteq IP$.

2 $SPP \subseteq IP$

2.1 Proof Protocol

We will be working over the field $\mathbb{F}$, using SPP programs computing with n variables. The depth (number of statements in the SPP program) will be L, and we require that f_j, g_j be easy to compute, and chosen such that all the p_i are polynomials of degree at most D. This is a strong, and somewhat nebulous requirement. However, we will later see that for the purposes of showing $PSPACE = IP$ it isn't hard to satisfy (we get f_j, g_j linear).

With these in mind, we can work out the interactive proof protocol. Suppose the prover claims that $p_L(x) = 1$. The protocol is as follows:

1. The prover computes $u = f_{L-1}(x)$ and $v = g_{L-1}(x)$ and defines the line
$$l(t) := tu + (1-t)v.$$
2. Then, $h(t) := p_{L-1}(l(t))$ should be a degree D polynomial in t, and the verifier can request the coefficients of $h(t)$ from the prover.
3. She then checks that $h(0)$ combined with $h(1)$ give $p_L(x)$; if this consistency check fails, the verifier rejects.
4. Otherwise, she chooses a random value in $z \in \mathbb{F}$, and asks the prover to compute $h(z)$, who returns the value α. This sets up a situation analogous to the initial proof of $p_L(x) = 1$, except now the polynomial is p_{L-1}, evaluated at $l(z)$, and the claimed value is α.
5. The verification proceeds recursively, starting from step one, finishing after L such iterations.

2.2 Analysis of protocol

There are three things to check about the above protocol:

1. Complexity – it's easy to see that the verifier runs in $poly(n, D, L, \log|\mathbb{F}|)$. All The first step is $poly(n)$; the second takes $poly(D)$, the third is constant-time, the fourth is again $poly(n)$. The process is iterated L times, which gives the desired bound.
2. Completeness – every valid claim is accepted with probability 1. This is obvious: we don't reject unless there is an inconsistency.
3. Soundness – invalid claims are accepted with probability $< \frac{LD}{|\mathbb{F}|}$. The analysis proceeds as for $\#P \subseteq IP$. So, we can pick a field larger than LD, and repeat the protocol some number of times to get arbitrarily good rejection probabilities.

3 $PSPACE \subseteq SPP$

For any $PSPACE$ machine M, we'd like to simulate it with an SPP program. Suppose our $PSPACE$ machine uses space $s(n)$, and let $a, b \in 0, 1, q_1, \dots q_k{}^{s(n)}$ be two valid states of the machine (q_k denotes a head with internal state k). Represent a, b as vectors over some large enough field $\mathbb{F}$ (so q_k become field elements not equal to 1). Also, for simplicity, we modify the rules so that any termination state (rejection or acceptance) transitions to itself in one step.

Define the function $Q_0(a, b)$ of two valid states to be 1 if the machine gets from a to b in exactly 1 step, and 0 otherwise. In one step, the machine can only make changes within a radius of one cell around the current head position. Consider some 6-tuple $a_{i-1}, a_i, a_{i+1}, b_{i-1}, b_i, b_{i+1}$; it's a valid transition if a_i contains the head, and the b_j match the action the machine should have taken. Then, $Q_0(a, b) = 1$ if and only if there is exactly one valid transition among all the 6-tuples for the given a and b. It's easy to write a polynomial of constant degree in every variable, which is 0 on invalid 6-tuples, and 1 on valid 6-tuples. Since a is valid, there is only one head on the tape. So, the sum of these polynomials over all 6-tuples in a, b will be 1 if and only if there is exactly one valid transition on the tape. Thus, Q_0 can be written as an efficiently computable polynomial of degree C in every variable on $\mathbb{F}^{s(n)}$.

Now, define

$$Q_i(a, b) = \sum_{c \text{ valid state}} Q_{i-1}(a, c) Q_{i-1}(c, b).$$

By induction, we may assume Q_{i-1} is of degree at most C in every variable. That means that $Q_{i-1}(a, c) Q_{i-1}(c, b)$ is of degree at most C in each a_i and b_i, and of degree $\leq 2C$ in c_i. We are summing over constant values of c, so the latter degree is irrelevant. Hence, $Q_i(a, b)$ is of degree $\leq C$ in every variable.

The sequence of polynomials Q_i is not quite in SPP yet: we have a "fixed" number of inputs $2s(n)$, constant degree C polynomials, program depth $s(n)$, but the width of these polynomials is $2^{s(n)}$ – we are adding together too many terms. Fortunately, we can reduce the width by splitting up these additions into many (but still polynomial in $s(n)$) steps.

Recall that our valid states were defined on an alphabet of $\{0, 1, q_1, \dots, q_k\}$, but every valid state had only one occurence of a q_i. For convenience, we recode this so that bits $1 \dots \log s(n)$ store the position of the head, $\log s(n) + 1 \dots \log s(n) + \log k$ store the head's internal state, and the following bits store the actual tape. Now the alphabet is just $\{0, 1\}^{s(n)}$[1].

Define

$$\begin{aligned} R_{i,j}(u, w_1 \dots w_j, v) = & \sum_{w_{j+1}, \dots, w_{s(n)} \in \{0,1\}} Q_{i-1}(u, w) Q_{i-1}(w, v) = \\ & = R_{i,j+1}(u, w_1 \dots w_j 0, v) + R_{i,j+1}(u, w_1 \dots w_j 1, v), \end{aligned}$$

with $R_{i,s(n)}(u, w, v) = Q_{i-1}(u, w) Q_{i-1}(w, v)$, and $Q_i(u, v) = R_{i,0}(u, , v)$. We thus get a sequence of $s(n)^2$ efficiently computable polynomials, each one depending only on the preceding one, with degrees still bounded by C. They can be evaluated in the following order:

$$Q_0, R_{1,s(n)}, R_{1,s(n)-1}, \dots, R_{1,1}, Q_1, R_{2,s(n)}, \dots$$

This is an SPP program, and $Q_s(x_{initial}, x_{accept}) = 1$ (assume without loss of generality that the machine has one accept state) if and only if the initial state accepts in the $PSPACE$ program. We must start with a $PSPACE$ machine for this procedure to work, because otherwise the SPP's parameters will not be polynomial in the original input size n.

[1] And, we'll pretend that that all strings correspond to valid machine states; this isn't quite the case if $s(n)$ or k are not powers of two. In this case, the formulas for R_{ij} actually need some adjustments to exclude the invalid states.

4 History of IP & PCP

Interactive proofs were first independently investigated by Goldwasser, Micali, Racoff and Babai. GMR termed their construction IP (for interactive proof), which was initially based on hidden coins: the verifier could toss coins that would remain unknown to the prover. IP allowed polynomially many rounds of interaction. Babai's class was named AM (for Arthur-Merlin), where King Arthur was a BPP verifier, and Merlin an unbounded prover (from whom Arthur didn't, and couldn't hide his coin tosses). In AM, Arthur and Merlin would have just one round of interaction. The complexity classes with k rounds of interaction were termed $AM[k]$ and shown to be equivalent to AM for all constant k. It was also shown that the hidden coin assumption in IP was unnecessary. Thus, the only remaining distinction between AM and IP is that IP permits polynomially many rounds of interaction. AM is contained in $\Pi_2 P$, while, as we have seen, $IP = PSPACE$. Thus, it seems that allowing polynomially many rounds of interaction strictly increases the computing power.

Babai's introduction of interactive proofs was driven by an interest in complexity classes, while GMR were studying the cryptographic implications of the technique. For example, they were interested in the ability to prove that one has the solution to a hard problem (e.g. hamiltonicity or 3-coloring) without revealing the solution itself. Techniques like these are called "zero-knowledge proofs"; to construct these, GMR relied on one-way functions.

Another development was the introduction of several provers by Ben-Or, Goldwasser, Kilian and Widgerson. 2-prover IP has two provers who may have previously agreed upon a strategy, and have huge resources, as before. However, during the interactive proof they are not allowed to exchange information. The verifier can ask them questions in any order (e.g. prover 1, prover 1, prover 2, 2, 1, 2, 2). Clearly, this class (called $2IP$) is at least as powerful as IP – the verifier could just ignore one of the provers. BOGKW proved that with 2 provers, no one-way functions were necessary for zero-knowledge proofs.

Similarly, one can define $3IP, 4IP, \ldots MIP$ (the union of mIP for all constant m). It's natural to ask if those classes get progressively more powerful. It turns out that the answer is no; Fortnow, Rompel and Sipser showed that all multiple-prover systems are equivalent to OIP – oracle interactive proof. OIP is an interactive proof system with no memory – it commits to all its answers before any interaction happens. An oracle can be viewed as a static construction, possibly exponentially big in input size. We check a polynomially-sized part of the construction, which is enough to conclude that with high probability a correct proof exists (even though there may be many mistakes in the oracle itself).

One can easily simulate any MIP computation using an oracle: to simulate prover n, the verifier would just ask it "what would prover n say after being asked the following questions?" FRS also proved the more difficult converse: $2IP$ can simulate OIP. Informally, having more than one prover forces the provers to essentially commit to an answer to every question before the interaction starts. Otherwise, the verifier would be able to ask the two provers a sequence of questions that would expose the inconsistency with non-zero probability.

So, we have the following relations between interactive proof types:

$$IP \subset 2IP = 3IP = \cdots = MIP = OIP.$$

The reverse inclusion, $2IP \subset IP$ is believed to be false; here is a justification of sorts. We have shown that $IP = PSPACE$; it can also be proved that

$$OIP = 2IP \subseteq NEXPTIME.$$

In 1990, Babai, Fortnow and Lund showed that $NEXPTIME \subseteq OIP$, a proof which is works on a technical strengthening of $PSPACE \subseteq IP$. Since it is generally believed that $PSPACE \subsetneq NEXPTIME$, $IP \subsetneq 2IP$ would follow. Using the BFL result, in 1991, Feige, Goldwasser, et al showed that approximating $CLIQUE$ is hard.

The concept of a large proof that one can check in a few places (inspired by OIP) led to the creation of the probabilistially checkable proof classes $PCP[\gamma(n), q(n)]$. In this notation, introduced by Arora and Sajra in 1992, $\gamma(n)$ denotes the number of bits of randomness used by the verifier (as a function of input size n). The parameter $q(n)$ is the number of bits queried from the oracle by the verifier – how much of the proof was examined. In this notation, OIP is just $PCP[poly(n), poly(n)] = NEXPTIME$.

It's natural to ask what other complexity classes we can get by varying q and γ. Looking at the definitions of NP and BPP, it's easy to write them in the new notation:

$$\begin{aligned} PCP[0, poly(n)] &= NP \\ PCP[poly(n), 0] &= BPP. \end{aligned}$$

A follow-up paper by Babai, Fortnow, Levin, and Szegedy showed that

$$NP \subseteq PCP[poly(\log n), poly(\log n)]$$

. This is remarkable: in the classical formulation of an NP problem, we have proof of length $poly(n)$. In the PCP formulation, our proof becomes $2^{poly(\log N)}$ long, which isn't polynomial, but is certainly subexponential. However, the verifier uses only $poly(\log N)$ bits of the proof, and $poly(\log N)$ bits of randomness; at the cost of a slight proof size blow-up, we got much quicker (probabilistic) verification.[2]

Much improved results followed. Arora and Safra showed $NP \subseteq PCP[O(\log n), O(\sqrt{\log n})]$ and then Arora, Lund, Motwani, Sudan, and Szegedy showed that

$$NP \subseteq PCP[O(\log n), O(1)].$$

Håsted later gave a proof of $NP \subseteq PCP[O(\log n), 3]$ so that valid statements are accepted with probability almost 1, and invalid statements are accepted with probability $\lesssim \frac{1}{2}$. (Note that the exact number of query bits is not interesting except when mentioned in conjunction with the error probability. Specifically, it is always possible to reduce the query complexity from any constant to three while increasing the error probablity to a constant bounded away from one. Håstad's result is special in that it gets very low query complexity with very low error-probability.)

[2] The actual result of Babai, Fortnow, Levin, Szegedy is essentially stronger parameter and could have easily been used to show $NP = PCP[\log n, poly(\log n)]$. While their result does not focus on the randomness parameter, they do mention that their proof sizes are polynomial in the size of the classical proof and indeed nearly-linear.