Advanced Complexity Exam 2018

All written documents allowed. No Internet access, no cell phone.

1 The Zachos-Heller theorem

Let $\Sigma = \{0, 1\}$. All our random tapes r, r_1, r_2, ..., are strings over Σ.

Take $L \in \mathbf{BPP}$, so that :

— if $x \in L$ then $Pr_r[\mathcal{M}(x, r) \text{ accepts}] \geq 1 - 1/2^n$,

— if $x \notin L$ then $Pr_r[\mathcal{M}(x, r) \text{ accepts}] \leq 1/2^n$,

where $\mathcal{M}$ is a deterministic Turing machine working in polynomial time $p(n)$, and using $q(n)$ random bits (meaning that the size of r is $q(n)$).

For a bit $b \in \Sigma$, say that $\mathcal{M}(x, r) = b$ to abbreviate « either $b = 1$ and $\mathcal{M}(x, r)$ accepts, or $b = 0$ and $\mathcal{M}(x, r)$ rejects ». $\mathcal{M}(x, r) \neq b$ is the negation of $\mathcal{M}(x, r) = b$. Let R_{xb} be the set of those r such that $\mathcal{M}(x, r) \neq b$.

Let also $b = (x \in L)$ mean « either $b = 1$ and $x \in L$, or $b = 0$ and $x \notin L$ », and $b \neq (x \in L)$ be its negation.

1. Let L' be the language of all tuples (x, b, H) such that R_{xb} has a collision for H, where $b \in \Sigma$ and $H = (h_1, \cdots, h_\ell)$ is a tuple of linear hash functions from $\Sigma^{q(n)}$ to $\Sigma^{m'}$, and where ℓ and m' are polynomials in the size n of x, $\ell \geq m'$, to be determined later. Show that L' is in $\mathbf{NP}$.

   We guess the collision r, the tapes r_1, ..., r_ℓ with which r is in collision, and we check in polynomial time that $\mathcal{M}(x, r) \neq b$, $\mathcal{M}(x, r_1) \neq b$, ..., $\mathcal{M}(x, r_\ell) \neq b$ and that $r_j \neq r$ and $h_j(r_j) = h_j(r)$ for every j.

2. We define the following algorithm. On input x, we draw b and H (as described above) at random, uniformly and independently. Then we test whether $(x, b, H) \in L'$. If so, we return the special symbol fail, otherwise we return b.

   (a) Show that if $b \neq (x \in L)$, then that algorithm must return fail... under a constraint on n, ℓ, and m' that you will give explicitly. We will name that constraint (A).

   If $b \neq (x \in L)$, for example $x \in L$ but $b = 0$, so that $\mathcal{M}(x, r) \neq b$ with probability at least $1 - 1/2^n$, then R_{xb} will have a collision for H by Sipser II, as soon as :

$$(A) \qquad (1 - 1/2^n)2^{q(n)} > \ell.2^{m'}.$$

If it has a collision for H, then (x, b, H) is in L', so we must return fail.

(b) Show that if $b = (x \in L)$, then the probability that the algorithm returns fail is smaller than or equal to $1/2^{\ell-m'+1}$ ... under a constraint on n, ℓ, and m' that you will give explicitly. We will name that constraint (B).

If $b = (x \in L)$, then $\mathcal{M}(x, r) \neq b$ with probability at most $1/2^n$. By Sipser I, the probability that R_{xb} has a collision for H is at most $1/2^{\ell-m'+1}$, provided $1/2^n.2^{q(n)} \leq 2^{m'-1}$, namely :

$$(B) \qquad q(n) - n \leq m' - 1.$$

With probability at least $1 - 1/2^{\ell-m'+1}$, there will be no collision, so (x, b, H) will not be in L', hence the algorithm will return b, not fail.

(c) We simply take $\ell = m'$. Show that, for n large enough, one can find m' so that (A) and (B) are satisfied, and such that m' is bounded by a polynomial in n.

To satisfy (B) we can just take $m' = q(n) - n + 1$, and that is polynomial. We check that (A) holds when n is large enough : (A) is equivalent to $(1-1/2^n) > (q(n) - n + 1)2^{-n+1}$, whose left-hand side tends to 1, and whose right-hand side tends to 0.

3. Conclude that **BPP** is included in the class $\mathbf{ZPP^{NP}}$ of languages that can be decided in expected polynomial time with zero error, on a randomized Turing machine with access to an **NP** oracle. This is the *Zachos-Heller theorem.*

When n is large enough, we run the algorithm of the previous question for as long as it returns fail*. Once it returns some other value b, we stop and return b. By 2(a), since it did not return* fail*, we must have $b = (x \in L)$. The probability that it stops at any given turn of the loop is at least $1/2$, so we will stop in 2 turns of the loop on average.*

When n is not large enough, we simply use a precompiled table of answers, and return the tabulated answer for x.

4. Why is $\mathbf{ZPP^{NP}}$ equal to $\mathbf{RP^{NP}} \cap \mathbf{coRP^{NP}}$? A brief answer is enough. The classes $\mathbf{RP^{NP}}$ and $\mathbf{coRP^{NP}}$ are defined just like **RP** and **coRP**, except the Turing machine has access to an oracle deciding some language in **NP**.

Same argument as in the lecture notes for the characterization of $\mathbf{RP} \cap \mathbf{coRP}$ as the class of languages decidable by a randomized Turing machine that makes no error and returns in average polynomial time. Every language in $\mathbf{ZPP^{NP}}$ can be decided by two machines with timeout, one which accepts on reaching the timeout, the other one which rejects on reaching the timeout. Conversely, if we have one $\mathbf{RP^{NP}}$ machine $\mathcal{M}_1$ for L, and one $\mathbf{coRP^{NP}}$ machine $\mathcal{M}_2$ for L, we run $\mathcal{M}_1$ and $\mathcal{M}_2$ in a loop until either $\mathcal{M}_1$ accepts (and we accept) or $\mathcal{M}_2$ rejects (and we reject). If the probability of error of each is at most $1/2$, this terminates in 2 turns of the loop on average, by Markov's inequality.

5. Show that $\mathbf{RP}^{\mathbf{NP}} \subseteq \Sigma_2^p$.

   As in the lecture notes, where we showed that $\mathbf{RP} \subseteq \mathbf{NP}$*. Every language* L *in* $\mathbf{RP}^{\mathbf{NP}}$ *is decided by a randomized polynomial time Turing machine* $\mathcal{M}$ *with access to an oracle deciding some language* L' *in* $\mathbf{NP}$*, so that : if* $x \in L$ *then* $\mathcal{M}(x, r)$ *accepts with probability at least* $1/2$ *on* r*, and otherwise it rejects for every* r*. Hence* L *is the language of those* x *such that there is an* r *such that* $\mathcal{M}(x, r)$ *accepts. This show that* L *is in* $\mathbf{NP}^{\mathbf{NP}} = \Sigma_2^p$*.*

   Note that $\mathbf{RP} \subseteq \mathbf{NP}$ *does* not *imply* $\mathbf{RP}^{\mathbf{NP}} \subseteq \mathbf{NP}^{\mathbf{NP}}$ *: even if we could define a map* $\mathcal{C} \mapsto \mathcal{C}^{\mathbf{NP}}$*, there is no reason why it should be monotonic.*

6. Deduce a new proof of the Sipser-Gács-Lautemann theorem $\mathbf{BPP} \subseteq \Sigma_2^p \cap \Pi_2^p$.

   By taking classes of complements in the previous question, $\mathbf{coRP}^{\mathbf{NP}} \subseteq \Pi_2^p$*. Hence* $\mathbf{BPP} \subseteq \mathbf{ZPP}^{\mathbf{NP}} = \mathbf{RP}^{\mathbf{NP}} \cap \mathbf{coRP}^{\mathbf{NP}} \subseteq \Sigma_2^p \cap \Pi_2^p$*.*

2 L/poly, branching programs, and BP · L

For a function $f\colon \mathbb{N} \to \mathbb{N}$, a language L is in the class $\mathbf{L}/f$ if and only if there is a family of so-called *advice words* $(adv_n)_{n\in\mathbb{N}}$, where adv_n is of size $O(f(n))$ (and is not necessarily computable), and a logarithmic space deterministic Turing machine $\mathcal{M}$, such that for every input x of size n, $x \in L$ if and only if $\mathcal{M}(x, adv_n)$ accepts. Note that $\mathcal{M}$ works in space $O(\log n)$, where n is the size of x, not counting the size of adv_n.

As usual, by space we mean the size used by the work tapes, and ignore all other tapes, notably the read-only input tape x and the read-only advice tape.

L/poly is the union of the classes $\mathbf{L}/f$ when f ranges over the polynomials with coefficients in $\mathbb{N}$. We use that every input x is given in binary.

A *branching program* (for short, *BP*) π is just like a circuit, except that its gates are built from the `if` x_i `then _ else _` connective instead of $\wedge$, $\vee$, $\neg$; the notation x_i specifies bit i of the input x. Additionally, the two wires 0 and 1 specify false (rejection) and true (acceptance) respectively. Formally, a *net-list* for π is a list of wire specifications of the form :

$$m\colon \texttt{if } x_i \texttt{ then } j \texttt{ else } k$$

where $m > j, k, 1$ (m, j and k are wire numbers), and where consecutive wire specifications have values of m that increase by exactly 1, and start at 2. For example, the following branching program computes (at its last specified wire, number 4) $(x_3 \wedge \neg x_5) \vee (\neg x_3 \wedge x_2)$:

$$2\colon \texttt{if } x_5 \texttt{ then } 0 \texttt{ else } 1$$
$$3\colon \texttt{if } x_2 \texttt{ then } 1 \texttt{ else } 0$$
$$4\colon \texttt{if } x_3 \texttt{ then } 2 \texttt{ else } 3$$

A BP π is of *length* n if it can take inputs of size n, namely if every x_i in π is such that $0 \leq i < n$. The *size* of π is its size as a net-list, where x_i is given by writing i in binary. Wire numbers are also written in binary.

We say that a BP *accepts* its input x if and only if the value of its final wire, evaluating each x_i as bit i of x, is 1. A language L *has polynomial BPs* if and only if, for every $n \in \mathbb{N}$, there is a length n branching program π_n of polynomal size $p(n)$ such that for every input x of size n, $x \in L$ if and only if π_n accepts x.

7. Show that every language L that has polynomial BPs is in $\mathbf{L/poly}$. Be careful about the size of the work tapes your Turing machine uses.

   We simply take π_n, written as a net-list, as advice string. We do not read *from first wire to last wire, as that would require us to memorize the values of the wires, and there are just too many (poly(n)).*

   Instead, we read the BP π_n in reverse. If we see a wire specification m: `if` x_i `then` j `else` k, to make wire m true we must make wire j true if $x_i = 1$, wire k true instead if $x_i = 0$. So we start with the last wire for m, read i, read x_i, decide to go to wire j or k, and we continue this way until we reach wire 0 (in which case we reject) or 1 (accept).

   We need one work tape of size $O(\log n)$ to read i, and decrement it while travelling on the input tape until we can read x_i. We also need a work tape w to hold the current wire number m. If we decide to go to j, we will have to read π_n in reverse, and for each wire specification, to decrement w and compare it with j.

8. Conversely, show that every language L in $\mathbf{L/poly}$ has polynomial BPs. Hint : given a logspace Turing machine $\mathcal{M}$ with polynomial advice, some form of the configuration graph of $\mathcal{M}$ on inputs of size n has polynomial size in n... and you need polynomially many wires. You may assume that $\mathcal{M}$ has only one work tape, and always terminates.

   Given input size n, we build a graph whose « configurations » are the contents of the work tape w, plus the internal state of the Turing machine (encoded as a letter inside the work representing the work tape, so that the position of the head can be determined), plus the positions i, j of the two heads in the input tape and on the advice tape. All that has size $O(\log n)$, hence takes $k = poly(n)$ different possible values.

   Given such a « configuration » $C = (w, i, j)$, and given adv_n (given separately) the Turing machine has enough information to decide what the next configuration should be, depending on whether x_i is true or not (at least if C is not a final configuration). Hence there are two edges going out of C, one to some configuration C_1 if $x_i = 1$, another one to C_2 if $x_i = 0$.

   We need to convert that into a branching program with wire specifications

   $$\ulcorner C \urcorner\text{: } \texttt{if } x_i \texttt{ then } \ulcorner C_1 \urcorner \texttt{ else } \ulcorner C_2 \urcorner$$

   where i is given in $C = (w, i, j)$, and $C \mapsto \ulcorner C \urcorner$ is some appropriate numbering scheme.

   Since the Turing machine terminates, the graph is acyclic. That is a relief.

We also need to have just one 1 *wire but we may have several accepting configurations. It suffices to number all the accepting configurations as* 1*, and all the rejecting configurations as* 0*. Then we number the remaining configurations by a form of topological sort, and that ensures the remaining conditions on net-lists.*

The class $\mathbf{BP} \cdot \mathbf{L}$ is defined as the class of languages L such that there is a deterministic Turing machine $\mathcal{M}$ such that if $x \in L$ then $Pr_r[\mathcal{M}(x,r) \text{ accepts}] \geq 2/3$, and otherwise $Pr_r[\mathcal{M}(x,r) \text{ accepts}] \leq 1/3$—and such that $\mathcal{M}(x,r)$ works in space $k \log n$, where n is the size of x, for some constant k that (notably) does not depend on r.

9. Let $L \in \mathbf{BP} \cdot \mathbf{L}$. Let n denote the size of x. Why can we assume r to be of size polynomial in n ?

   Since $\mathcal{M}(x,r)$ works in space $k \log n$, for each given r, it works in polynomial time as soon as it terminates. We can force it to terminate by simulating it with a timeout of $exp(k \log n) = poly(n)$. On reaching the timeout, we decide to make it reject : hence the probabilities of acceptance remain the same after that modification.

   Once this is done, the machine only has time to read $poly(n)$ random bits from r.

10. Show that $\mathbf{BP} \cdot \mathbf{L}$ admits error reduction : for every language in L, for every polynomial q, there is a deterministic Turing machine $\mathcal{M}$ working in space $O(\log n)$ (independently of the size of r) such that if $x \in L$ then $Pr_r[\mathcal{M}(x,r) \text{ accepts}] \geq 1 - 1/2^{q(n)}$, and otherwise $Pr_r[\mathcal{M}(x,r) \text{ accepts}] \leq 1/2^{q(n)}$.

    As in the lecture notes, except using the trick of democratic classes is a bit awkward. ($\mathbf{L}$ *is democratic, but some details are already subtle, and then we would need to compose logspace computations.)*

    Instead, we just repeat the experiments and make a vote. That is, we run $\mathcal{M}$ on the same x with independent random values of r for $36q(n)\log 2$ turns, yielding $36q(n)\log 2$ bits as outcomes, and reusing the logarithmic space of $\mathcal{M}$ at each turn. (We of course need a counter of size $\log(36q(n)\log 2) = poly(n)$ to do the iteration.)

    We don't store the $36q(n)\log 2$ outcomes, but instead we count how many of those bits are true in a counter c. We also count the outcomes on a tape k. At the end, $k = 36q(n)\log 2$, and we accept if $c \geq 2k$, namely if $2c \geq k$, which can be done by a lexicographic comparison of c shifted by one position with k.

    By Chernoff's bound, the probability of error we get this way is $\leq 1/2^{q(n)}$.

11. Show that every language of $\mathbf{BP} \cdot \mathbf{L}$ has polynomial branching programs.

    We imitate the proof of Adleman's theorem $\mathbf{BPP} \subseteq \mathbf{P}/poly$.

    Let $\mathcal{M}$ be a randomized Turing machine that decides $L \in \mathbf{BP} \cdot \mathbf{L}$ in logarithmic space, with error $\leq 1/2^{q(n)}$. Take $q(n) = n+1$, or any polynomial larger than that. The probability (over r) that $\mathcal{M}(x,r)$ gives the wrong answer for some x of size n is $\leq 2^n/2^{q(n)} \leq 1/2$. Hence there is an r such that $\mathcal{M}(x,r)$ gives the right answer for every x of size n. We take r as advice string. That has polynomial size, and $\mathcal{M}$ works in logspace.

Branching programs are a relaxed form of *binary decision diagrams* (BDD), a fundamental data structure used in symbolic model-checking. A BDD on an n-bit input x has exponential size in the worst case, and that worst case is attained often, even in practice. The above delineates when polynomial size is achievable.

3 PCP

A (R, Q, T)-*restricted verifier* is a randomized Turing machine with direct access to a proof tape that works in three phases :

— it computes $Q(n)$ positions on the proof tape, in polynomial time, while accessing the input tape x and the random tape r only (not the proof tape) ; the random tape contains only $R(n)$ bits ;

— it reads the bits on the proof tape y at these positions ;

— using x, r, and the bits just read from y, it decides to accept or reject in time $T(n)$ (in this phase, the machine cannot access the proof tape).

The class $\mathbf{PCP}(R, Q, T)$ is the class of languages L such that there is an (R, Q, T)-restricted verifier V such that :

— if $x \in L$, then there is a proof y such that $\Pr_r[V(x, y, r) \text{ rejects}] = 0$;

— if $x \notin L$, then for every proof y, $\Pr_r[V(x, y, r) \text{ accepts}] \leq 1/2$.

We do not require any particular bound on the size of y.

12. Show that graph non-isomorphism is in $\mathbf{PCP}(O(n \log n), 1, O(1))$. You should of course get some inspiration from one of the algorithms we gave in the lectures for that problem.

> *Use the* $\mathbf{IP}[1]$ *protocol for graph non-isomorphism from the lectures. Given the two input graphs* G_0, G_1 *on* N *vertices* $1, \ldots, N$, *draw one bit* b *at random (*0 *or* 1*), draw a random permutation of* G_b *and ask Merlin to give a bit* b' *such that* $G_{b'}$ *is isomorphic to* G_b. *Merlin's strategy* f *is a function that maps every graph on* N *vertices to the* b' *it would answer.*
>
> *To cast this into a PCP, Merlin can just give the whole table for* f *: we first agree on an encoding of graphs on* N *vertices as numbers between* 0 *and some maximum value, and Merlin gives a bitstring as proof tape, where bit number* i *is set if and only if* $f(G) = 1$, *where* i *is the encoding of* G. *This proof tape is, as announced, not of polynomial size. There are* $2^{N(N-1)/2}$ *graphs on* N *vertices, and we need Merlin to store one bit per graph !*
>
> *The verifier will then just draw the random permutation of* G_b, *compute the encoding* i *of the resulting graph, and read the single bit* b' *at position* i *on the proof tape. It can then decide in constant time by comparing* b' *with* b.
>
> *The number of random bits is* 1 *(for* b*) plus the number of bits needed to draw a permutation of* N *values, namely* $\log(N!) = O(N \log N)$. *Since* N *is at most* n, *this means* $O(n \log n)$ *random bits. When* $x \in L$ *(*G_0 *and* G_1 *are not isomorphic),*

then Merlin can yield a proof tape that forces Arthur to accept always, otherwise Arthur will reject half of the time : this is precisely the acceptance condition we used for the PCP class.