Lecture Notes 3

7 Reductions and $\mathcal{NP}$-completeness

Let L, L' be languages. We say L is *polynomial-time reducible* to L' and write $L \leq_{\mathrm{p}} L'$ if there exists a polynomial-time computable function f such that

$$x \in L \text{ iff } f(x) \in L'.$$

If we think of L and L' as "problems", then f maps a problem in L to an equivalent problem in L', in the sense that the new problem has a "solution" iff the original one does. This is interesting computationally since if we have an algorithm A' to solve L', then we can build an algorithm A for L as follows:

Given input x:
1. Compute $y = f(x)$.
2. Compute $r = A'(y)$ and return r.

The time for A is the time for $f(x)$ plus the time for $A'(y)$ (plus a little bit of overhead). This will be polynomial in $|x|$ if A' is also polynomial-time computable, but the polynomials will not be the same. In particular, y can be longer than x, so we must argue that $A(|y|)$ is polynomially bounded in $|x|$. This is true since f is polynomial-time computable, and the output of a computation cannot be longer than the length of the input plus the running time. Thus, if f runs in time $p(\cdot)$ and A runs in time $q(\cdot)$ for polynomials p and q, then A runs in time approximately $p(n) + q(p(n))$, which is itself a polynomial in n.

A language L' is said to be *$\mathcal{NP}$-complete* if $L' \in \mathcal{NP}$ and for every $L \in \mathcal{NP}$, $L \leq_{\mathrm{p}} L'$. If any $\mathcal{NP}$-complete language L' is decidable in polynomial time, then all languages in $\mathcal{NP}$ can be decided in polynomial time by the above algorithm. This gives us the fundamental theorem of $\mathcal{NP}$-completeness.

Theorem 1 *If any $\mathcal{NP}$-complete language L' is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.*

The contrapositive is also of interest, namely, if $\mathcal{P} \neq \mathcal{NP}$ (as is widely believed to be the case), then the membership problem for every $\mathcal{NP}$-complete language is infeasible.

We remark that hundreds of $\mathcal{NP}$-complete languages are known. One of the first to be shown $\mathcal{NP}$-complete is the *satisfiability problem* SAT. This is the set of Boolean formulas which evaluate to true under some assignment of truth values to the variables appearing in the formula.

Once one problem is known to be $\mathcal{NP}$-complete, then others can easily be shown to be $\mathcal{NP}$-complete using reductions.

Theorem 2 *Let L_0 be $\mathcal{NP}$-complete. Suppose $L_1 \in \mathcal{NP}$ and $L_0 \leq_{\mathrm{p}} L_1$. Then L_1 is also $\mathcal{NP}$-complete.*

Proof: We must show for every language $L \in \mathcal{NP}$ that $L \leq_{\mathrm{p}} L_1$. This easily follows from the transitivity of $\leq_{\mathrm{p}}$ since we have $L \leq_{\mathrm{p}} L_0$ (by the assumed $\mathcal{NP}$-completeness of L_0), and we're given that $L_0 \leq_{\mathrm{p}} L_1$. ∎

8 Probabilistic Turing Machines

8.1 Coin-flipping machines

A probabilistic Turing machine is structurally similar to a non-deterministic Turing machine in that the machine can have two possible next moves in any configuration. Unlike a non-deterministic machine, there is a well-defined rule for making the choice; namely, the choice is determined at random by the flip of a fair coin. Looking at the infinite configuration graph, if there are edges (u, v_0) and and edge (u, v_1) from configuration u, then we say that v_0 and v_1 are each reachable with probability $1/2$ from u. In general, after t steps there might be as many as 2^t reachable configurations, each of which is reachable from the initial configuration with probability 2^{-t}.

In general, infinite computations are possible, but the complicate the development of the theory, and we don't need them for our purposes. Hence, we will assume that all computation paths have polynomial length in the length of the input x. That is, there is a polynomial $p(\cdot)$ such that for every input x, every possible computation from the initial configuration for x reaches a halting state in at most $p(|x|)$ steps. As with polynomial-time non-deterministic machines, this is no real loss of generality since if all halting paths have length at most $p(|x|)$, we can force the remaining paths to halt and give a special answer after $p(|x|) + 1$ steps.

[We remark that one can fruitfully consider probabilistic machines that are polynomial time in a weaker sense. Namely, they are only required to halt with probability 1, and the expected number of steps before the machine halts is bounded by a polynomial in $|x|$. For example, consider the machine that flips its coin repeatedly and halts when the coin lands `heads`. The expected number of steps of this machine is $\sum_t t2^{-t} = 2$, yet it has the possibility of running forever. Given such a machine, we can truncate all computations after $p(|x|)$ steps for some polynomial p. The result is a machine that runs in time $p(n)$ but which sometimes gives the wrong answer.]

8.2 Deterministic machines with a random tape

As we did with non-deterministic machines, we can externalize the choices made by a polynomial-time probabilistic Turing machine M. We turn M into a deterministic machine M' by giving it a second input (conventionally on a second tape) that contains the outcomes of the coin flips make during the course of the computation. Since the coin flips are independent of everything else, they can all be flipped in advance and written down on the second input tape. Then when M would have flipped a coin, M' instead reads and tests the next bit of the second tape.

The only issue is knowing in advance how many coins will be needed. However, assuming M is a machine as described above that always halts within $p(n)$ steps, then it can never flip more than $p(n)$ coins, so the second (random) input tape need only have $p(n)$ random bits on it.

9 The class $\mathcal{BPP}$

A language L is *recognized* by a polynomial-time probabilistic Turing machine (ppTM) M if

1. $\forall x \in L, \Pr[M(x) = 1] \geq 2/3$;
2. $\forall x \notin L, \Pr[M(x) = 0] \geq 2/3$.

The class $\mathcal{BPP}$ is the set of all languages L that are recognized by some ppTM.

Theorem 3 *The class $\mathcal{BPP}$ is the same if "2/3" in the above definition is replaced by "$1 - 2^{-|x|}$".*

Proof: Assume $L \in \mathcal{BPP}$ is recognized by a ppTM M. We construct a new ppTM M' that recognizes L and has an error probability of at most $2^{-|x|}$ on any input x.

$M'(x)$ simply runs $M(x)$ for $r(|x|)$ times for a suitable function r and outputs the majority of the returned values. Using the Chernoff bound of section 2, we analyze the probability that the average of the $r(|x|)$ returned values differs from its expected value by more than 1/6, for only in that case might the majority be incorrect. The result is obtained when the number of repetitions $r(n)$ is just slightly larger than n. Details are left to the reader. ■

10 Negligible Functions

A ppTM may not always give the right answer and so has a probability of error. Informally, we want the error probability to be so small as to be negligible. But what does that mean? Informally, we want it to go to zero faster than $1/p(n)$ for any positive polynomial $p(\cdot)$, where a polynomial $p(\cdot)$ is said to be *positive* if the leading coefficient is positive. If $p(\cdot)$ is positive, then $p(n) > 0$ for all but finitely many values of n since the highest-degree term of a polynomial dominates all the others for large n.

Formally, a function $\mu(n)$ is *negligible* if

$$\forall \text{ positive polynomials } p(\cdot)\ \exists N\ \forall n > N \left(\mu(n) < \frac{1}{p(n)}\right).$$

The quantifier sequence "$\exists N\ \forall n > N$" occurs so frequently that it is convenient to give it a name. It says that the following predicate holds for all n from some point onward, so we can read this as "for all sufficiently large n". It also implies that the only values of n for which the following predicate fails must be $\leq N$. Since there are only finitely many such exceptions, we can equivalently read the quantifier sequence as "for all but finitely many n" or "for almost all n". We sometimes write this as $\overset{\infty}{\forall} n$.

11 Non-uniform Polynomial Time: $\mathcal{P}/$poly

Non-uniform polynomial time computation allows a different Turing machine to be used for each length n of inputs. We say that a language L is in the class $\mathcal{P}/$poly of *non-uniform polynomial time* if there is an infinite sequence of deterministic Turing machines $M_0, M_1, M_2, \ldots$ and two positive polynomials $p(\cdot)$ and $q(\cdot)$ such that

1. Each M_n has a description e_{M_n} of length at most $p(n)$.
2. The running time of M_n on each input x of length n is at most $q(n)$.
3. For all strings x, $x \in L$ iff $M_{|x|}$ accepts L.

Note that $\mathcal{P} \subseteq \mathcal{P}/$poly since if M accepts L in polynomial time, then the infinite sequence $M, M, M, \ldots$ of machines satisfies the definition for L being in $\mathcal{P}/$poly.

Theorem 4 *$\mathcal{BPP} \subseteq \mathcal{P}/poly$.*

Proof: Let M be a ppTM that accepts L. Using the techniques of section 9, we can assume that the error probability of M is strictly less than 2^{-n} for all inputs x of length n and that $M(x)$ runs in time at most $p(n)$ for some polynomial $p(\cdot)$. For each input length n, we construct a deterministic Turing

machine M_n that correctly determines membership in L for all strings x of length n. Moreover, the size and running time of M_n are both bounded by fixed polynomials in n.

Let x be an input string of length n and r a random choice string of length $p(n)$. Let $\delta(x, r) = 1$ if $M_r(x)$, the output of M with coin toss sequence r, gives the correct answer about x's membership in L, and let $\delta(x, r) = 0$ otherwise. Since M has error probability less than 2^{-n}, it follows that the number of 0's in each row x of δ (when viewed as a matrix) is less than $2^{-n} \cdot 2^{p(n)}$, so the total number of 0's in δ is less than $2^{p(n)}$, the number of columns. It follows that there is some column r with no 0's in it, that is, $\delta(x, r) = 1$ for each x of length n.

We now build r into the machine $M_n(x)$, so $M_n(x)$ simulates $M(x)$, except that the random coin flips are resolved using r. Clearly, M_n constructed in this way is correct for all x of length n. The resulting sequence of machines $M_0, M_1, M_2, \ldots$ shows that $L \in \mathcal{P}/\text{poly}$. ■

12 Oracle Machines

We've seen two examples of extending Turing machine by allowing them to make choices that are not determined by the current configuration. Non-deterministic machines allow arbitrary choices to be made. Probabilistic machines make the choices by flipping a fair coin. In both cases, it is often convenient to externalize the sequence of choices made as a second input, replacing the extended machine by an ordinary deterministic one that reads the next bit of the choice sequence whenever the extended machine would have made a choice.

We can view the choice sequence as a "helper" or "advice" string that may help the machine arrive at a correct answer. However, by placing the advice string on a tape and restricting our machines to run in polynomial time, we are also limiting the useful length of advice strings to being polynomial in the length of the real input x.

Oracle machines allow for exponentially long advice. An oracle oracle f is simply a function on strings. An oracle Turing machine $M^f(x)$ has an extra tape called the *oracle* tape and a special *query* operation. Suppose the contents of the oracle tape is y when a query operation is performed. Then on the next step, y is magically replaced by $f(y)$, and the computation continues. Thus, we can think of f as a database which we can query as often as we like, the only restriction being that we must construct the query y on the query tape, and this takes at least $|y|$ steps to perform. Also, to read the query result $f(y)$ takes at least $|f(y)|$ steps. If the oracle machine is restricted to run in polynomial time, then it means that it can make only a polynomial number of queries of the oracle, each of polynomial size.

Oracle machines give us another kind of reduction between problems. For example, if f is a factoring oracle (so that $f(x)$ returns a prime factor of x), then it is known that the RSA decryption exponent can be computed from the public key in polynomial time by a machine with an f oracle.