ITCS:CCT09 : Computational Complexity Theory Apr 8, 2009

Lecture 7

Lecturer: Jayalal Sarma M.N. *Scribe: Shiteng Chen*

In this lecture, we will discuss one of the basic concepts in complexity theory; namely the notion of reduction and completeness.

In the past few lectures, we saw ways of formally arguing about a class of problems which can be solved under certain resource requirements. In order to understand this at a finer level, we need to be able to analyse them at an individual level, and answer questions like "how do we compare resource requirements of various computational problems?" and "what is the hardest problem in a class $\mathcal{C}$?" etc. The notion of reductions comes in, exactly at this point.

The outline of this lecture is as follows. We will motivate and define different notions of reductions between problems. In particular we will define many-one (also known as Karp reductions), Turing reductions (also known as Cook reduction) and finally Truth-table reductions.

Then we will define the notion of completeness to a complexity class, and describe and (sometimes prove) completeness of problems to various complexity classes that we described in earlier lectures. This gives us a concrete machine-independent way of thinking about these resource requirements, which is going to be crucial in the upcoming lectures.

We will end by mentioning a few interesting concepts about the structure of NP-complete set. We will define sparse set, poly size circuit and collapse result which we will see in the upcoming lecture.

1 Reductions

In order to make comparisons between two problems (say A and B), a simple line of thought is to transform the instance of A into an instance of B such that if you solve the B's instance then you can recover A's solutions from it. There are many basic issues with this. First of all, how do formally capture the computation that is need to make this transformation? Well, the answer is agian our favorite abstration : Turing machines and circuits.

If we impose the resource bounds on the reductions (since they are also Turing machine computations), the comparisons between the problems can vary in nature. In particular if the reduction uses only log space then we call it a log-space reduction, and if it runs in poly time, then we call the reduction to be poly-time reductions.

Another natural thought is on how much computation is needed on the recovery of the solution of A from that of B. Varying this restruction will give us different kind of transformations between problems and gives us tools to analyse the complexity classes at a finer level.

1.1 Many-one reduction

This is a most basic (or strictest) abstraction of what we described in the previous section. Intuitively, it says, we should not have to do any computation (not even complimentation) after we do the trasformation. This leads to the notion of many-one reduction (we state this only for poly-time).

Definition 1 (poly-time many-one reductions) *Let $A, B \subset \Sigma^*$ which encodes computational problems, we say that A poly-time many-one reduces to B, denoted by $A \leq_m^p B$, if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that,*

$$x \in A \iff f(x) \in B$$

We shall quickly see some basic properties of this reduction. This defines a relation among subsets of Σ^*. It is quite clear that (exercise !) the relation is reflexive and transitive. It is not symmetric (nothing can be many-one reduced to Σ^* !).

A complexity class $\mathcal{C}$ is said to be closed under a reduction, if $B \in \mathcal{C}$ and $A \leq B$ then $A \in \mathcal{C}$ as well. We wll skip the proof of the following easy proposition as exercise.

Proposition 2 *The classes NP and PSPACE are all closed under poly-time many-one reductions.*

1.2 Hardness and Completeness

Having defined a way to compare problems, we can address the second question that we started out with, namely to use it to define the "hardest" problem in a class. Intuitively, a hard problem in a class $\mathcal{C}$ should be those problems to which every other problem in $\mathcal{C}$ reduces. Clearly, there may be problems which are reducible to each other, and hence more than one hard problem can exist.

For the notion of hardness to be useful, the reduction (machine M) should run in resource bounds less than that of the class itself. Otherwise, the computation that computes the reduction to the hard problem can solve the problem by itself.

Now we will make this formal.

Definition 3 (Hardness & Completeness) *Alanguage $L \subseteq \Sigma^*$ is hard for a class $\mathcal{C}$ under* poly *time many-one reductions if and only if every $L' \in \mathcal{C}$, $L' \leq_m^p L$. If in addition, L is in the complexity class $\mathcal{C}$, then L is defined to be complete for the class $\mathcal{C}$.*

Thus to show that a problem L is complete for a class $\mathcal{C}$, we have to show that (1) every problem in $\mathcal{C}$ reduces to L and (2) L itself is in $\mathcal{C}$. Suppose we have done the hard work once, to prove that a problem (say L) is hard for the class $\mathcal{C}$. Now, in order to show that another problem is hard for $\mathcal{C}$ we won't have to work from scratch. We saw that reductions are transitive, and hence it will suffice to show that L reduces to L'.

Another simple observation is that if a problem L is complete for the class $\mathcal{C}$ then $\overline{L}$ is complete for the complement class $\mathsf{co} - \mathcal{C}$.

Thus we need a hard problem to begin with. We will demonstrate this with an example. Let us say our $\mathcal{C}$ is NP. We describe below a generic NP-complete problem which is usually called *canonical complete problem.*

Definition 4 (Bounded Halting Problem) *Given the description (encoded string in Σ^*) of a non-deterministic Turing machine M, an input x and a integer k, test if M accepts input x in k steps.*

Theorem 5 BHP*is* NP*-complete.*

Proof Easy part first. BHP $\in \mathsf{NP}$. Indeed, an non-deterministic machine can simulate M and keep a counter initialised to k and check if the machine halts within the stipulated time limit k.

We will show that BHP is NP-hard. Let $L \in NP$. We will show that $L \leq_m^p$ BHP. By definition, there is a non-deterministic Turing machine M that computes L which runs in time $p(n)$ for some fixed polynomial p. Thus,

$$\begin{aligned} x \in L &\iff M \text{ accepts } x \text{ in time } p(|x|) \\ &\iff (M, x, p(|x|) \in \text{BHP}. \end{aligned}$$

■

Using this problem as the starting point, we can derive new NP-complete problems. Stephen Cook pioneered in this conceptual framework and proved that a bunch of important computational problem are actually NP-hard. This came as a surprise to many, and projected up the importance of P vs. NP problem.

We will not completely describe the NP-hardness proofs by Cook and others as we have seen it in the previous courses. We will put these into perspective, draw your attention to some aspects of the reduction, and leave the details for you to check with what you have already seen in the part courses.

We will define a more general problem.

Definition 6 (Circuit-Sat) *Given a Boolean Circuit $C(x_1, \ldots x_n)$, Circuit-Sat problem asks if there exists an assignment $a_1, \ldots a_n$ to the variables $x_1 \ldots x_n$ such that $C(a_1, \ldots a_n) = 1$.*

The first problem in Cook's list was SAT problem, which asks for checking satisfiability for a given Boolean formular F on n variables. A formula, by definition, is also a circuit. Hence Circuit-Sat if at all, is harder. But it is also clearly in NP because, the non-deterministic TM can guess an assignment (which is only length n), and then deterministically evaluate the value of the circuit at this input to check if it indeed evaluates to 1. Notice that the verification procedure (this problem is circuit value problem as we will see below) runs in time linear in size of the circuit (which is also a part of the input), since it just have to evaluate the circuit in the most natural way from leafs to root. Thus we will define the following problem as well.

Definition 7 (Circuit-Value) *Given a Boolean Circuit $C(x_1, \ldots x_n)$, and an assignment $a_1, \ldots a_n$ to the variables* Circuit-Value *problem asks if $C(a_1, \ldots a_n) = 1$.*

You all have seen Cook's reduction from BHP to SAT before, so we will skip it completely in this course. We will enumerate certain aspects of the reduction that are relevant for us now and state a few more propositions based on them. We reiterate the basic implication: *if there is a polynomial time algorithm for solving* SAT *then it gives us a polynomial time algorithm for all the problems in* NP *thus proving* P = NP.

A first observation is that the same idea can be adapted[1] to show that Circuit-Value is hard for P. We also notice that the reduction can be done in AC^0.

Proposition 8 *Under* AC^0 *many-one reductions :*

- Circuit-Sat *is* NP*-complete.*
- Circuit-Value *is* P*-complete.*

By now, a lot of interesting natural problems are known to be NP-complete. The following is a short list of the first few which were shown to be so. We will not do more hardness

[1]we will skip the details, since it is technical and we are not doing Cook's proof anyway.

proofs in this course, but it is highly recommended to go through some of them, if you have not studied them in a previous course.

Remark As an intriguing fact, most of the known reductions showing the NP-hardness are known to be AC^0-reductions. Here is an interesting question to think about: Are poly time many-one reductions strictly weaker than AC^0 many-one reductions?

Problem	Definition
Clique	Given a graph G and an integer k, Test if G has a complete graph of size k or not.
Independent Set	Given a graph $G(V, E)$ and an integer k, Test if there is $V' \subseteq V$ of size k such that for any $u, v \in V'$, $(u, v) \notin E$
Vertex Cover	Given a graph $G(V, E)$ and an integer k, Test if there is $V' \subseteq V$ of size k such that each $e \in E$ has at least one end point in V'.

2 Turing Reduction

Now we will define another way to compare problems. To motivate this, we start with an example problem. We will consider a trivial way to reduce SAT to $\overline{\text{SAT}}$. For instance, we know that:

$$x \in \text{SAT} \iff x \notin \overline{\text{SAT}}$$

From definition, it is unclear whether SAT $\leq_m^p$ $\overline{\text{SAT}}$, because the recovering the solution for problem the answer to the $\overline{\text{SAT}}$ involves a computation (namely switching yes to no and vice versa).

Generalising this further, we can imagine a reduction model which allows allow M (the Turing machine performing the reduction) to do any computation after getting the result from B and even allowing it to solve many instances of B. This leads to the notion of a Turing definition (also called Cook reductions). Formally:

Definition 9 (Poly time Turing Reduction) *Let $A, B \subset \Sigma^*$ which encodes computational problems, we say that A poly-time Turing reduces to B, denoted by $A \leq_T^p B$, if there exists a polynomial time oracle Turing machine M which has oracle access to the set B and decides membership of strings in A.*

A first observation is that this definition still captures what we started out with, while trying to define the notion of reductions. That is, if $B \in P$, then that implies $A \in P$ as well. We will look for more properties. For instance:

Proposition 10

$$\text{SAT} \leq_T^p \overline{\text{SAT}}.$$

$$\text{SAT} \leq_m^p \overline{\text{SAT}} \iff \mathsf{NP} = \mathsf{co\text{-}NP}.$$

Proof Part (1) follows from basic definition itself. Indeed, the Turing machine asks if $x \in \overline{\text{SAT}}$ and then answers yes if it says no, and vice versa.

To see part (2): Suppose $\mathsf{NP} = \mathsf{co\text{-}NP}$. By $\mathsf{co\text{-}NP}$-completeness of $\overline{\text{SAT}}$, and NP-completeness of SAT we have that for every formula ϕ there is another formula ϕ' such that (1) given ϕ, ϕ' can be constructed in poly time, and (2) ϕ' is satisfiable if and only if ϕ is not satisfiable. This itself gives the many-one reduction from SATto $\overline{\text{SAT}}$. ■

It is widely believed that $\mathsf{NP} \neq \mathsf{co\text{-}NP}$. This may be an evident that $\leq_m^p$ and $\leq_T^p$ are not the same.

Now we relate Turing reduction to orcale complexity classes that we have seen in previous classes. The following theorem follows from definitions, but characterises Turing reductions in terms of oracle Turing machines.

Theorem 11 *For any language $A \subset \Sigma^*$, if $A \leq_T^p B$ SAT then $A \in \mathsf{P}^B$. In particular if $B =$ SAT then $L' \in \mathsf{P}^{\mathsf{NP}}$.*

We will demonstrate the power of Turing reductions by descrbing one more example. We consider the following two problems. A clique is complete graph.

Exact-Clique: Given a graph G an integer k, determine if the maximum size of a clique in G is exactly k?

Theorem 12 Exact-Clique $\equiv_T^p$ Clique *and* Clique $\leq_T^p$ Exact-Clique*.*

Proof Exact-Clique $\leq_T^p$ Clique : Let $G(V, E)$ be the given graph and k be an integer. Notice that the maximum clique size of G is k if and only if $(G, k) \in$ Clique and $(G, k+1) \notin$ Clique. This gives 2-query Turing reduction to Clique.

Clique $\leq_T^p$ Exact-Clique : Let $G(V, E)$ be the given graph and ℓ be an integer. To check if there is a clique of size ℓ, ask the Exact-Clique problem with k ranging from $a, \ldots, |V|$. If the maximum k' for which this says Yes, is at least ℓ then answer Yes, otherwise No. ■

The above two examples, although are Turing reductions, are not exactly using its complete power. More precisely, the queries that are asked are not adaptive. In the example of SAT to $\overline{\mathrm{SAT}}$, there was only one query. In the above example (EXACT-CLIQUE $\leq_T^p$ CLIQUE) the second query does not depend on the answer to the first query. This leads us to a type of reduction that is in between many-one reduction and the general Turing reductions, namely Truth table reductions.

2.1 Truth Table Reductions

A Truth Table reduction must present all of its (finitely many) oracle queries at the same time. Recalling the characterisation of Turing reductions using oracle model of computation, we can imagine this to be *non-adaptive oracle Turing machines* where the queries should be written down simultaneously. This restriction ensures that once the oracle gives out the response (answer bits, 0/1, of the queries), then the computation (since it does not involve oracle queires any more) will only involve a table look up into the truth table to decide whether to accept or reject based on these response bits. This leads to the name. More formally,

Definition 13 (Poly time Truth Table Reduction) *Let $A, B \subseteq \Sigma^*$ which encodes computational problems, we say that A poly-time Truth Table reduces to B, denoted by $A \leq_{tt}^p B$, if there exists a polynomial time non-adaptive oracle Turing machine M which has oracle access to the set B and decides membership of strings in A.*

As an example, all the examples of the Turing reductions that we presented above are in fact truth table reductions. In addition, the following comparison of the strength of the reductions follows from our discussion.

Proposition 14 *Let $A, B \subseteq \Sigma^*$ which encodes computational problems. Then:*

$$A \leq_m^p B \quad \Rightarrow \quad A \leq_{tt}^p B \quad \Rightarrow \quad A \leq_T^p B$$

3 Structure of NP-complete sets

We saw that there is a set of problems (one of them being SAT) which are the *hardest* problems in the complexity class NP. There has been a lot of interest in studying the structure of these sets in the 70s and that led to a lot of insights on the density of the NP-complete sets. In this lecture we will mention (we will skip the details, giving pointers for further reading) some of these ideas.

A first question is about the density of any NP-complete set. This leads us to the definition of sparse set.

3.1 Sparse sets

We start with the definition of sparse sets.

Definition 15 (Sparse sets) *A set $A \subseteq \Sigma^*$ is said to be sparse, if there exists a polynomial $p(n)$ such that A_n, the n^{th} slice of A, defined as,*

$$A_n = \{x \in A : |x| = n\}$$

has at most $p(n)$ strings in it.

Is an NP-complete set likely to be sparse. The following theorem connects this to the well known problem regarding collapse of the polynomial heirarchy that we saw in the previous lecture.

Theorem 16 *If a sparse set is* NP*-complete then* PH *collapses to Σ_2.*

We will postpone the proof of this theorem to next lecture. In fact, we will show more. We will see that (1) any sparse set has a poly size circuit to compute it. (2) If an NP-complete problem can be computed by polynomial sized circuits then PH collapses to Σ_2.

Since CIRCUIT-VALUE is P-complete, we can conclude that any language in P has polynomial size circuits computing them. Thus if we prove that there is no polynomial sized circuit family computing SAT, we prove P $\neq$ NP.

3.2 Berman-Hartmanis Conjecture

This is another attempt to study the strucutre of NP-complete sets. The basic question considered is whether all the NP-complete sets are structurally the same. In particular, is there an easily computable transformation among these sets (as subsets of Σ^*). This is the famous Berman-Hartmanis conjecture. More formally,

Definition 17 (poly-time isomorphism) *Two sets A and B are polynomial-time isomorphic if there exists a function $f : A \to B$ such that*

- *$x \in A \iff f(x) \in B$.*
- *f is a bijection from A to B. Hence f^{-1} exists.*
- *f and f^{-1} is polynomial-time computable.*

The relation defined by the isomorphism between pairs of languages is an equivalence relation.

Conjecture 1 (Berman-Hartmanis, 1976) *Every pair of* NP*-complete sets are* poly*-time isomorphic to each other.*

Now what is the connection with P vs. NP question? If $\mathsf{P} = \mathsf{NP}$ then there would be a finite NP complete set (the poly time reduction can simply solve the problem and produce an instance inside or outside the finite set S.). Now this set, which is finite, can't be polynomial time isomorphic to SAT which is an infinite set. Thus isomorphism conjecture, if proved, will imply $\mathsf{P} \neq \mathsf{NP}$. However, it is widely believed these days that the isomorphism conjecture is too strong to be true.

There is a lot of classical literature published since the late seventy's which addresses different versions of isomorphism conjecture. We will leave it to you to explore these using the links provided at the course website entry for this lecture.