E0 224 Computational Complexity Theory

Lecture 6

Lecturer: Chandan Saha *Scribe: Sandip Sinha*

August 25, 2014

1 Coping with NP-Completeness

NP-Complete problems appear in a wide variety of real-world applications. In some cases, solving such a problem is necessary. Let us look at some methods to cope with **NP**-Completeness. The main idea here is that, while the problem is difficult to solve *exactly* on *every* possible input, it might be good enough in practice to solve the problem approximately and on some inputs.

1.1 Special cases might admit efficient algorithms

Although the problem, when specified in full generality, may be **NP**-hard, there may be special cases for which it is possible to design efficient algorithms.

Example 1.1.1. 2-SAT $\in$ **P**.
We know that CNF-SAT is **NP**-Complete. However, the special case of 2-SAT, in which each clause has at most 2 literals, can be solved in deterministic polynomial-time.

Example 1.1.2. $(2+\epsilon)$-SAT is **NP**-hard.
This result was proved by Austrin, Guruswami and Håstad in 2013. The precise statement of the problem is given below.
Suppose we are given a ω-SAT instance (meaning every clause has atmost ω literals). We are given the promise that there exists a satisfying assignment for ϕ that satisfies atleast $\left(\lceil \frac{\omega}{2} \rceil - 1\right)$ literals in every clause.
Task: Find a satisfying assignment for ϕ.
Result: This problem is **NP**-hard.
However, the same problem becomes tractable (meaning it is in **P**) if $\left(\lceil \frac{\omega}{2} \rceil - 1\right)$ is replaced by $\left(\lceil \frac{\omega}{2} \rceil\right)$. It admits an efficient algorithm which is a generalisation of the algorithm for 2-SAT. This result can be understood by taking $\omega = 3$.

Reference: Computers and Intractability: A Guide to the Theory of NP-Completeness - *Michael R. Garey, David S. Johnson.*

1.2 Design of Approximation Algorithms

Example 1.2.1. TSP.
Problem: Given a graph G with n nodes and $\binom{n}{2}$ weighted edges, find the minimum weight

Hamiltonian cycle in G.
This is **NP**-hard since the decision version of the problem is **NP**-Complete.
However, in practice, we might only require an algorithm that solves the problem in the case when the n nodes are points on a plane, and the triangle inequality is satisfied. This version is called the Euclidean TSP, which also turns out to be **NP**-hard.
Remark: If **P** $\neq$ **NP** then TSP does not have an efficient constant factor approximation algorithm.
Result (Arora '96): For every $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for Euclidean TSP with running time poly$(n\ (\log\ n)^{O(1/\epsilon)})$.
Thus, if the restriction of Euclidean distance and the allowance of approximation separately is not sufficient for tractability. However, if both are applied simultaneously, there exists an efficient algorithm which is close to optimal.

Reference: The Design of Approximation Algorithms - *David P. Williamson, David B. Shmoys.*

1.3 Average case complexity: Levin's theory

This method, due to Levin, is concerned with the design of algorithms that are efficient on a distribution of inputs which are likely to arise in practice, rather than on all inputs. A natural question at this point is: What is the class of distributions that arise in practice? Levin makes a daring suggestion that we allow any distribution from which it is possible to draw samples in polynomial-time (**P**-samplable distribution). He reasoned that real-life instances must be produced by the actions of the world around us. If we believe in the strong form of the Church-Turing thesis, then the world can be simulated on a TM, in which case we can assume that the computation that produced the instance was efficient. We can therefore assume that the time taken for the computation was polynomial in the instance size.
The formalization of real-life distributions can be done in two ways:
(i) Polynomial time computable (**P**-computable) distributions
(ii) Polynomial time samplable (**P**-samplable) distrbutions

Example 1.3.1. CLIQUE on random graphs.
Input instance: $<$ G, $k>$ where G is a random graph with n vertices (meaning each edge is in the graph independently with probability $\frac{1}{2}$), and k is a positive integer, $k \leqslant n$.
Fact: CLIQUE can be solved in n^{2logn} time on random graphs. This is slightly worse than polynomial and much better than $2^{\epsilon n}$, the running time of the best algorithms on worst-case instances.
It can be shown that a random graph does not have cliques of large size with high probability. Specifically, for $k(n) \gg 2\ \log\ n$, the probability that a random graph has a clique of size at least $k(n)$ is very small. This suggests an algorithm for CLIQUE which generates all subsets with 2log$n$ vertices and checks whether these subgraphs have cliques. If atleast one subgraph has a clique, the algorithm returns 1. A false negative (an instance in which the algorithm wrongly returns 0) only occurs if there is a clique with more than 2 log n vertices, which is highly unlikely.

1.4 Fixed Parameter Intractability

Fixed parameter intractability is concerned with finer issues about time complexity than just the difference between polynomial and non-polynomial time.

Example 1.4.1. VERTEX COVER
It is known that VERTEX COVER is **NP**-Complete. Given a graph G with n vertices and a positive integer $k \leqslant n$, a naive algorithm would check the condition over all subsets of G with k vertices. Thus, it has running time $\binom{n}{k} \approx O(n^k)$.
Question: Is it possible to design an algorithm for VERTEX COVER with running time $f(k).\text{poly}(n)$?
The answer turns out to be 'Yes'. The next natural question is to ask whether INDSET, which appears to be quite similar to VERTEX COVER, also has such a $f(k).\text{poly}(n)$ algorithm. There is strong evidence that it does not, since INDSET is a member of a large class of **NP** problems that are *hardest* in the sense that one of them has a $f(k).\text{poly}(n)$ algorithm iff all of them do. It is widely believed that INDSET has complexity $2^{\Omega(n)}$. The intuitive feeling is that the naive algorithm of enumerating all possible subsets is close to optimal.
Reference: Parameterized Complexity Theory - *J. Flum, M. Grohe.*

2 More Complexity Classes, Non-Determinism

Definition 2.1.1. $\mathbf{EXP} := \bigcup_{c \geqslant 1} \mathbf{DTIME}(2^{n^c})$

Easy observation: $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$
The first inclusion is obvious. Suppose $L \in \mathbf{P}$ is decided by a TM N in polynomial-time. Then $L \in \mathbf{NP}$ since we can take N as the TM M and $q(x) \equiv 0$, so that u is the empty string, where M, $q()$ and u are as in the definition of the class **NP**.
The second inclusion holds since, given a language L in **NP** and M, $q()$ as per the definition of the class **NP**, and on input x, we can exhaustively enumerate all strings u of length $q(|\text{x}|)$ and use M to check whether u is a valid certificate for x. M accepts L iff some u satisfies this condition. Since $q(n) = O(n^c)$ for some $c \geqslant 1$, there are $2^{O(n^c)}$ choices for u, and thus we can decide L in time $2^{O(n^c)}$.
It is known that $\mathbf{P} \subsetneq \mathbf{EXP}$. Therefore, atleast one of $\mathbf{P} \subsetneq \mathbf{NP}$ and $\mathbf{NP} \subsetneq \mathbf{EXP}$ must hold. It is conjectured that both inclusions are strict.

Definition 2.1.2. $\mathbf{DUBEXP} := \bigcup_{c \geqslant 1} \mathbf{DTIME}\left(2^{2^{n^c}}\right)$

Definition 2.1.3. Non-deterministic Turing Machine (NDTM):
A NDTM M is described using a 4-tuple
$M = < \Gamma, Q, \delta_0, \delta_1 >$
where $\delta_i : < \Gamma, Q > \rightarrow < \Gamma, Q, \{L, R\} >$, $i = 0, 1$, are transition functions. At each elementary step, the TM selects δ_0 or δ_1 arbitrarily (thus being non-deterministic).

Remark: This is obviously an unrealistic machine. It is just a mathematical model which turns out to be useful for our understanding.
Note: Apart from q_{start} and q_{halt}, we also have a special state q_{accept}.

Definition 2.1.4. We say that a NDTM M *accepts* a string $x \in \{0,1\}^*$ if, on input x, *there exists* a sequence of applications of the transition functions δ_0 and δ_1 that takes M to q_{accept} starting from q_{start}.

Definition 2.1.5. Running Time of a NDTM:
Let $T : \mathbb{N} \rightarrow \mathbb{N}$. We say that a NDTM M has running time $T(n)$ if for *every* input $x \in \{0,1\}^n$

and for *every* sequence of non-deterministic choices of the transition functions, M always reaches either q_{halt} or q_{accept} in $T(n)$ steps.

Definition 2.1.6. Deciding a language in time $T(n)$:
We say a NDTM M decides a language $L \subseteq \{0,1\}^*$ in time $T(n)$ if:
(i) the running time of M is $T(n)$, and
(ii) M accepts x iff $L(x) = 1$.

Definition 2.1.7. NTIME$(T(n))$:
A language $L \subseteq \{0,1\}^*$ is in **NTIME**$(T(n))$ if there is a NDTM M that decides L and has running time bounded by $c.T(n)$ where c is a constant that depends only on L and M but not on input instances.

The following claim gives an equivalent definition of **NP**.

Claim. $\mathbf{NP} = \bigcup_{c \geqslant 1} \mathbf{NTIME}(n^c)$

Proof. The main idea in this proof is that we can view the sequence of non-deterministic choices made by a NDTM M during an accepting computation as a certificate for the input, and vice-versa.
Suppose $L \in \mathbf{NP}$ and M, $q()$ are as per the definition of **NP**. We describe a polynomial time NDTM N that uses M to decide L. On input x, it writes down a string u of length $q(|\mathrm{x}|)$, using its ability to make non-deterministic choices for the transition functions. Then it verifies whether u is a valid certificate by running the verifier M on u and x. If u is valid, N enters q_{accept}. Thus, N enters q_{accept} on x iff there exists a valid certificate for x. Since $p(n) = O(n^c)$ for some $n \geqslant 1$, $L \in \mathbf{NTIME}((n^c))$.
Conversely, suppose $q : \mathbb{N} \to \mathbb{N}$ is a polynomial and L is decided by a NDTM N in time $q(n)$. For every x in L, there exists a sequence of non-deterministic choices which enables N to enter q_{accept} on input x. This sequence is of length $q(|\mathrm{x}|)$ and can be verified in polynomial time by a deterministic TM M. M simulates N using the sequence of non-deterministic choices and verifies that it would have entered q_{accept} on making these sequence of choices. Thus, the sequence serves as a certificate for x, and $L \in \mathbf{NP}$. □

Definition 2.1.8. $\mathbf{NEXP} := \bigcup_{c \geqslant 1} \mathbf{NTIME}\left(2^{n^c}\right)$

Definition 2.1.9. $\mathbf{NDUBEXP} := \bigcup_{c \geqslant 1} \mathbf{NTIME}\left(2^{2^{n^c}}\right)$

Fact: DUBEXP $\neq$ NDUBEXP $\Rightarrow$ EXP $\neq$ NEXP $\Rightarrow$ P $\neq$ NP
Thus, for higher complexity classes, the assumption that deterministic time is not equal to non-deterministic time becomes stronger.

3 Search versus Decision (for NP Problems)

Question: Does the search version of a problem in **NP** reduce to its decision version?
Answer: Yes, for any **NP**-Complete problem. We show the reduction explicitly in the case of SAT.
Assume we are given a polynomial-time algorithm A that decides SAT (i.e. $\mathbf{P} = \mathbf{NP}$). We outline an algorithm B that, on input a satisfiable CNF formula ϕ with n variables, finds a satisfying

assignment for ϕ using $O(n)$ calls to A, with some additional polynomial-time overhead. We first invoke A on ϕ to check if it is satisfiable. If it is satisfiable, we set $x_1 = 0$ in ϕ and use A to decide whether the corresponding formula is satisfiable. If it is satisfiable, we fix $x_1 = 0$; if not, we fix $x_1 = 1$ (this can be done in polynomial time) and continue recursively with the simplified formula, thus reducing the number of variables by 1 each time. This is reasonable since we know that the formula must be satisfiable for at least one of the cases $x_1 = 0$ and $x_1 = 1$. Continuing in this manner, we end up fixing all n variables while ensuring each intermediate formula is satisfiable, which means that ϕ is satisfied by the final assignment to the variables. It is easily seen that this algorithm invokes A $(n + 1)$ times.
Since every **NP** language is reducible to SAT, the search version of such a problem is equivalent to the decision version, in the sense that if the decision version can be solved (equivalently, if **P** = **NP**), then the search version can also be solved in polynomial time. We make this precise in the following theorem:

Theorem. Suppose that **P** = **NP**. Then for every **NP** language L there exists a polynomial time TM B that on input $x \in L$ outputs a certificate for x.

Proof. The description of the algorithm B above proves the theorem in the case of SAT. Now, let L be a **NP** language. We use the fact that the reduction from L to SAT outlined in the *Cook-Levin Theorem* is, in fact, a *Levin* reduction. This implies that we have a polynomial-time computable function f such that $x \in L \iff f(x) \in$ SAT and we can actually map a satisfying assignment for $f(x)$ into a certificate for x. Thus, we can use the algorithm above to determine an assignment for $f(x)$ and then map it into a certificate for x. □

It is natural to ask if this is true of every language in **NP** in general. That this is not true, under the assumption that **DUBEXP** $\neq$ **NDUBEXP**, has been shown by *Mihir Bellare* and *Shafi Goldwasser* in a paper titled *The Complexity of Decision versus Search* in 1992. This result is stated below.
Result: *(Informal Statement)* There is a language in **NP** whose search version does not reduce efficiently to its decision version, unless doubly-exponential time equals non-deterministic doubly-exponential time.

References

[1] S. ARORA and B. BARAK "Computational Complexity: A Mordern Approach", Cambridge University Press, 2009