COMPSCI 532: Design and Analysis of Algorithms	October 9, 2015
Lecture 11	
Lecturer: Debmalya Panigrahi	*Scribe: Allen Xiao*

1 Overview

In this lecture, we review the notion of algorithmic hardness in order to motivate approximation algorithms. These are algorithms which produce solutions that may not be optimal, but whose difference from the optimal we can bound. The existence of polynomial time approximation algorithms is a great boon, since many interesting problems lack exact polynomial time algorithms under the assumption that $\mathsf{P} \neq \mathsf{NP}$.

2 Computational Complexity

Definition 1. *A **decision problem** is one with a yes-no answer depending on the value of its input. Formally: given input string S, is S in some set L?*

For the purpose of this class, we will restrict ourselves to decision problems which are *decidable* (there exists *some* algorithm which can compute the answer in finite time). Many of the optimization problems we have seen before can be formulated as decision problems with some polynomial overhead. Formally, this notion is defined using the *Turing machine*, a model for computation. Without going into detail, you can think of every Turing machine as an algorithm which answers the question in a decision problem (the decidability assumption means that at least one exists). The running time of these machines is exactly the running time of the algorithm. Intuitively, we say that problems are *hard* when there are no fast algorithms for them. The main distinction we make will be between those which have polynomial time algorithms, and those for which we can only prove weaker properties.

P

Definition 2. *A decision problem is in the complexity class* P *(polynomial time) if there exists a polynomial time algorithm deciding (answering) it.*

It turns out that many interesting problems have not (yet) been proved to be in P. Most of these, however, are in a different class called NP.

NP

Definition 3. *A decision problem is in the complexity class* NP *(nondeterministic polynomial time) it has a polynomial time verifier. A **verifier** for a decision problem takes a (decision problem) input S, an answer (yes or no), some additional information C commonly called a certificate or proof or witness.*

Certificate

- *When the true answer is "yes", there must exist some certificate for which the verifier can prove the answer is "yes". If given the wrong certificate, the verifier is allowed to answer "no" incorrectly.*
- *When the true answer is "no", there must be no certificate for which the verifier proves (incorrectly) the answer is "yes".*

Intuitively, problems in NP are ones whose solutions are naturally easy to check. Often, the definition is phrased as a game between a *prover* and a *verifier*, where the omniscient prover wants to convince the

computationally limited verifier that its answer to the decision problem is correct. The prover hands the verifier the input, the claimed solution, and the certificate.

Example 1. Consider the boolean satisfiability (SAT) problem.

The input to SAT is a boolean formula of conjunctions (AND) between disjunctive (OR) clauses.

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge \cdots$$

The decision problem for SAT is whether a given formula has a satifying assignment. A polynomial time verifier for this problem takes as a certificate a satisfying assignment, and checks satisfiability clause by clause. On the other hand, if the formula is unsatisfiable, no assignment will pass this verification scheme. We conclude that SAT has a polynomial time verifier, and is therefore a problem in NP. Many common verifiers for search/optimization problems do indeed use the search "solution" as the certificate.

Interestingly, there are also problems not known to be in NP.

Example 2. Let NON-SAT be the problem of boolean unsatisfiability. That is, given a boolean formula, report whether the formula has no satisfying assignment. In other words, the set of formulas in NON-SAT is the *complement* of SAT. It is not known if NON-SAT is in NP.

Definition 4. *A decision problem is said to be in* $\mathsf{co-NP}$ *if its complement is in* NP. *In the verifier, there must instead exist certificates for all "no" instances, and no certificates can exist for "yes" instances. To distinguish:*

- NP*:* (S, yes, C) *is verfiable.*
- $\mathsf{co-NP}$*:* (S, no, C) *is verfiable.*

Lemma 1. $\mathsf{P} \subseteq \mathsf{NP}$ *and* $\mathsf{P} \subseteq \mathsf{co-NP}$

Proof. If a problem is in P, we can use the polynomial time algorithm solving it as the verifier. No certificate is requried, and the process takes polynomial time for both "yes" and "no" instances, so the problem is in both NP and $\mathsf{co-NP}$. □

Some of the central open questions in complexity theory (and computer science as a whole) ask whether there are "problems in the gaps" between P, NP, and $\mathsf{co-NP}$.

1. Does $\mathsf{P} = \mathsf{NP}$?
2. Does $\mathsf{NP} = \mathsf{co-NP}$?
3. NP and $\mathsf{co-NP}$ can be generalized in the verifier model by allowing the verifier and prover to communicate in multiple rounds during verification. The equivalence classes created are collectively called the *polynomial hierarchy*. The previous questions can be generalized to ask whether different levels of the polynomial hierarchy collapse into the same set.

The common belief is that these questions all answer negatively, although no proof (in either direction) currently exists.

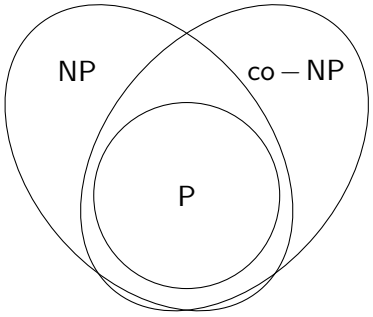
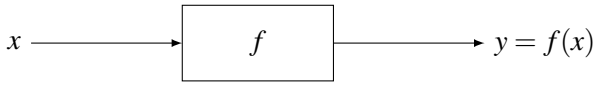


Figure 1: A diagram of the relationships in Lemma 1.

2.1 Reductions

Our main tool for arguing about the hardness is reduction.

Definition 5. *Problem A is **polynomial time reducible** to problem B if there exists a polynomial time algorithm f which transforms an instance x of A into an instance $y = f(x)$ of B, such that $x \in A$ if and only if $y \in B$ (both answers agree). We write this as $A \leq B$.*



Since the we know that the answers agree on x and y, running an algorithm which solves B will let us solve instances of A (in the same time, plus a polynomial factor). Intuitively, *A is is no harder than B*, since by spending additive polynomial time, solving B will solve A.

Definition 6. *A problem is* NP***-hard*** *if every problem in* NP *is polynomial time reducible to it. Additionally, if an* NP*-hard problem is a member of* NP*, we say that it is* NP***-complete****.*

Theorem 2 (Cook-Levin)**.** *The boolean satisfiability problem (*SAT*) is* NP*-complete.*

The Cook-Levin Theorem [Coo71][Lev73] is one of the landmark results of complexity theory. We will not state the proof here, but it essentially transforms a verifying Turing machine into a boolean satisfiability formula with a polynomial number of clauses and literals. Using reductions, Karp [Kar72] later showed that 21 classical combinatorial problems were also NP-complete. In fact, most NP-hard problems we encounter are NP-complete.

Lemma 3. *Let A and B be two* NP*-complete problems, then $A \leq B$ and $B \leq A$. In other words,* NP*-completeness is an equivalence class under polynomial time reductions.*

Proof. Without loss of generality, we need only show $A \leq B$. This follows immediately from the NP-hardness of B, and $A \in$ NP. □

2.2 Examples

A few of Karp's 21 problems include SAT, 3SAT, VERTEX-COVER, SET-COVER, HAMILTONIAN-PATH, HAMILTONIAN-CYCLE, CLIQUE, 3D-MATCHING, SUBSET-SUM. You can find these problems and their reductions in many textbooks, so we will only walk through a few examples here.

Example 3. 3SAT is a variation of SAT, where each clause is restricted to at most 3 literals. Again, we will use the convention of n variables and m clauses. Let each ℓ_i be either a variable or its negation.

$$(\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_5 \vee \ell_6 \vee \ell_7) \wedge \cdots \wedge (\ell_{3m-2} \vee \ell_{3m-1} \vee \ell_{3m})$$

In this example, we will show that 3SAT $\geq$ SAT. Since 3SAT is otherwise a special case of SAT SAT $\geq$ 3SAT, this gives NP-completeness of 3SAT by Lemma 3.

Our reduction works by reducing the size of long clauses of SAT until we have only clauses of size at most 3. The main idea is to add dummy variables to strip 1 or 2 variables from a large clause. Consider a clause of size 4:

$$(\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4)$$

Let d_1 be a dummy variable. Observe that the following formula is equivalent:

$$(\ell_1 \vee \ell_2 \vee d_1) \wedge (\neg d_1 \vee \ell_3 \vee \ell_4)$$

Intuitively, the value of d_1 chooses between halves of the original formula for the ℓ_i which must be true. When $d_1 = 0$, either ℓ_1 or ℓ_2 must be true. When $d_1 = 1$, either ℓ_3 or ℓ_4 must be true. The two formulas are equivalent. The new formula added one new variable, and one more clause. Suppose instead the clause has k literals:

$$(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$$

Let each d_i be a unique dummy variable:

$$(\ell_1 \vee d_1) \wedge (\neg d_1 \vee \ell_2 \vee d_2) \wedge (\neg d_2 \vee \ell_3 \vee d_3) \wedge \cdots \wedge (\neg d_{k-1} \vee \ell_k)$$

Suppose the $\ell_2 = 1$. Then we can set d_i the following way:

$$(\ell_1 \vee \underset{1}{d_1}) \wedge (\neg \underset{0}{d_1} \vee \ell_2 \vee \underset{0}{d_2}) \wedge (\neg \underset{1}{d_2} \vee \ell_3 \vee \underset{0}{d_3}) \wedge \cdots \wedge (\neg \underset{1}{d_{k-1}} \vee \ell_k)$$

In general, notice that we only have $k-1$ dummy variables for k clauses. By the pigeonhole principle, at least one of the clauses is satisfied by one of the real literals ℓ_i, instead of a dummy. Applying this transformation to all clauses, we have a 3SAT instance of polynomial size in n and m.

Example 4. The INDEPENDENT-SET problem takes as input a tuple (G,k), and asks if the graph $G = (V,E)$ has an *independent set* $S \subseteq V$ of size at least k. S is an independent set exactly when none of its elements have an edge in E. In this example, we show that INDEPENDENT-SET $\geq$ SAT. Given a 3SAT instance $C_1 \wedge C_2 \wedge \cdots \wedge C_m$, we construct the following graph. For each clause, we build a gadget:

- for each literal ℓ_i in the clause, make a new vertex v_i.

We add the following edges:

- Between every literal ℓ_i to its negation $\neg \ell_i$.
- Between the gadget vertices for each clause (not between clauses).

Let $k = m$. Consider the vertices picked in S to be the literals set to true. If there is an independent set of size k, then the second condition insists that we pick one vertex from each gadget. Additionally, S cannot contain a variable and its negation, since we added edges between them. It follows that, by setting the literals in S to true, we have a satisfying assignment for all m clauses.

3 Summary

In this lecture, we introduced the complexity classes P, NP, and $\mathsf{co-NP}$. Then, we defined polynomial time reductions and NP-completeness, a wide equivalence class of "hardness" on these reductions, and gave a few examples of polynomial time reductions.

References

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[Kar72] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[Lev73] Leonid A Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.