COMPSCI 330: Design and Analysis of Algorithms April 19, 2016

Complexity Classes and Polynomial-time Reductions

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song*

1 Overview

In this lecture, we introduce complexity classes and polynomial-time reductions.[1]

2 Easy versus hard problems

In this section, we will begin by defining a suite of hard problems we have not examined before during the semester. Interestingly, many of these problems have seemingly similar variants that are easy to solve. We will mention these variants as we go in order to highlight how a small change in a problem definition can result in a large jump in complexity.

2.1 Satisfiability

The first hard problem we will examine is what is known as *Satisfiability* or SAT. As input, we are given a set of n boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ (i.e., each variable can be set to either true or false). We are then given a *boolean formula* over these variables of the following form (noting that this is just a specific example where $X = \{x_1, x_2, x_3, x_4\}$):

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee x_4) \wedge (x_4 \vee \overline{x_1} \vee x_2) \tag{1}$$

In terms of notation, '$\vee$" and "$\wedge$" denote the conjunction and disjunction operations, respectively (which are analogous to the OR and AND operations for boolean variables in a programming language). $\overline{x_i}$ denotes the negation of the boolean variable x_i (analogous to $!x_i$). We deem sequence of conjunctions within a given set of parentheses (e.g., $(\overline{x_2} \vee x_3 \vee x_4)$) as a *clause*, and we call the unnegated/negated variables within the formula *literals*.

So more generally, as input we are given a boolean formula over the variables in X that the disjunction of m clauses, each of which is the conjunction of a sequence of boolean literals. Our goal then is to determine if there is a way of setting the variables (i.e., assigning "true" or "false" to each x_i) such that the entire formula evaluates to "true". We call such an assignment a *satisfying assignment*. If there exists at least one assignment for a formula, we say it is *satisfiable*. For instance, one satisfying assignment for the above example is x_1 = true, x_2 = false, x_3 = true, and x_4 = true (and there are others, as well). A simple example of a formula that is not satisfiable is

$$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}). \tag{2}$$

Observe that no matter how we set x_1 and x_2, at least one of the clauses will not be satisfied.

A more structured version of SAT is what is known as k-SAT, which is defined identically but with added requirement that each clause has exactly k literals. Therefore, the formulas in (1) and

[1]Some materials are from notes by Nat Kell and Allen Xiao.

(2) are instances of 3-SAT and 2-SAT, respectively. Surprisingly, we can solve 2-SAT in $O(n+m)$, but for any $k \geq 3$, there is no known polynomial time solution. In other words, the jump from $k = 2$ to $k = 3$ makes the problem much harder, and in fact we will show that 3-SAT is at least as hard as generic SAT.

2.2 Traveling Salesman, Hamiltonian Cycles, Eulerian Cycles

One of the most famous hard problems is what called the *Traveling Salesman* problem, or TSP. Here, we are given edge-weighted graph G (we typically think of the vertices of the graph as representing cities and an edge weight of an edge (u, v) as specifying the distance between cities u and v). The goal is to find a cycle in the graph that includes every vertex with minimum total weight (this models a salesman traveling to a set of cities and returning home; ideally he wants to minimize the total distance he has to travel).

Again, we will be able to argue why TSP is hard later on, but to do this, let us first define a slightly simpler problem known as *Hamiltonian Cycle* or HAM. Now, we are given an unweighted graph G, and the goal is to determine whether there exists a cycle in the graph that visits every vertex exactly once. Even though HAM is somewhat simpler than TSP, HAM is still a hard problem.

As we mentioned a moment ago, the jump from 2-SAT to 3-SAT makes an easy problem into a hard one. A similar phenomenon occurs with an analogous problem to HAM known as *Eulerian Cycle*. For the Eulerian Cycle problem, we are again given a unweighted graph G, but now the objective is to determine if there exists a path in the graph that starts and ends at the same vertex, includes every *edge* in the graph, and does not repeat any edges. Eulerian Cycle is in fact polynomial-time solvable due to the following elegant characterization: *a graph has an eulerian cycle iff the degree of each vertex is even.* Therefore in order to determine if a graph is eulerian, we just need to iterate over each vertex v in V and check to see if $\deg v$ is even.

2.3 Matching

A problem we have seen earlier in the semester is that of *maximum matching.* This matching problem turns about to be solvable in polynomial time. However, we can make the problem hard by making it a *3D-matching problem.* That is, the problem is now defined for a graph where edges are now a subset of $V \times V \times V$. Making the jump from 2D-matching to 3D-matching makes the problem hard.

3 Computational Complexity

Definition 1. *A* ***decision problem*** *is one with a yes-no answer depending on the value of its input.*

Here we will restrict ourselves to decision problems and many of the optimization problems we have seen before can be formulated as decision problems with some polynomial overhead.

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.

Definition 3. *A decision problem is in the complexity class* NP *(nondeterministic polynomial time) if 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.*

- *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.

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.

Definition 4. *A decision problem is said to be in* $\mathsf{co}-\mathsf{NP}$ *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.*
- $\mathsf{co}-\mathsf{NP}$*:* (S, no, C) *is verfiable.*

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

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 $\mathsf{co}-\mathsf{NP}$. □

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 $\mathsf{co}-\mathsf{NP}$.

1. Does $\mathsf{P} = \mathsf{NP}$?
2. Does $\mathsf{NP} = \mathsf{co}-\mathsf{NP}$?

The common belief is that these questions all answer negatively.

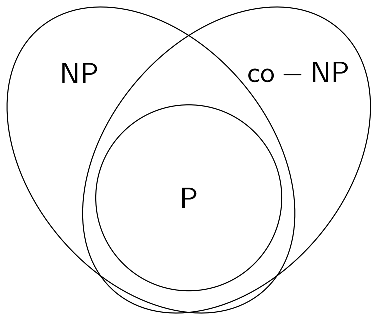

Figure 1: A diagram of the relationships in Lemma 1.

3.1 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 their answers agree. We write this as $A \leq B$. Polynomial time reductions are transitive.*

$$x \longrightarrow \boxed{\quad f \quad} \longrightarrow y = f(x)$$

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.*

Example 2. 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, and this gives NP-completeness of 3SAT because 3SAT is in NP.

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:

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

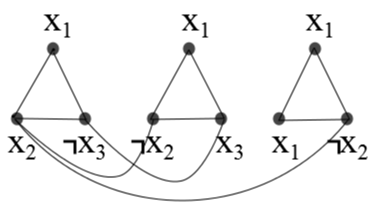

Figure 2: Graph of $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_1)$.

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:

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

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 3. 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 $\geq$ 3SAT. 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 (see an example in Figure 2):

- 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.

Example 4. The VERTEX-COVER problem takes as input a tuple (G, k), and asks if the graph $G = (V, E)$ has a *vertex cover* $S \subseteq V$ of size at most k. S is a vertex cover when each edge is incident to at least one vertex in S. In this example, we show that Vertex-Cover $\geq$ Independent-Set. Given an Independent-Set instance $(G' = (V', E'), k')$, we construct a Vertex-Cover instance $(G = (V, E), k)$, where $G = G'$ and $k = |V'| - k'$. To prove that the two instances have the same answer, we show that $S \subseteq V$ is an independent set iff $V \setminus S$ is a vertex cover. If S is an independent set, we have for any $e \in E$, one endpoint of e is in $V \setminus S$, therefore, $V \setminus S$ is a vertex cover. If $V \setminus S$ is a vertex cover, we have there does not exist an edge $e \in E$ such that both two endpoints of e are in S, therefore, S is an independent set.

Example 5. The Set-Cover problem takes as input a universe of elements X, subsets S_1, S_2,...,S_m, where $S_i \subseteq X$, and a number k. We ask if there exists a collection of subsets T, where $|T| \leq k$, such that $\cup_{S_j \in T} S_j = X$. In this example, we show that Set-Cover $\geq$ Vertex-Cover. Given a Vertex-Cover instance $(G = (V, E), k_v)$, we construct a Set-Cover instance: $X = E$; for each $v \in V$, let $S_v = \{e = (u, v) | e \in E\}$; $k = k_v$. Vertex-Cover is just a special case of Set-Cover.