15-750: Graduate Algorithms April 26, 2017

Lecture 36: NP-Completeness

Lecturer: David Witmer *Scribe: Dimitris Konomis*

1 Introduction-Motivation

The algorithms we have studied thus far in the class are polynomial-time: on inputs of size n, their worst-case running time complexity is $\mathcal{O}(n^c)$ for some constant $c > 0$. There are many problems for which we do not know of any polynomial time algorithm. For example, Turing's famous Halting Problem cannot be solved by any computer, no matter how much time we allow. In general, problems that are solvable in polynomial time are considered tractable, or easy, whereas problems that require superpolynomial time are considered intractable, or hard.

The rest of the class will deal with the following topics:

- **NP-complete problems**. This is an interesting class of problems, whose status is unknown. No polynomial-time algorithm has been discovered for an NP-complete problem, nor has anyone proved that no polynomial-time algorithm can exist for any one of them.
- **Reductions** This is a widely used technique to prove that a problem is NP-complete.
- **How do we deal with possibly intractable NP-Complete problems**? Usually, we either:
  - Design approximation algorithms, or,
  - Come up with heuristics and focus on special cases of the problem in hand.

2 Formally Defining Problems

2.1 Optimization, Decision & Search Problems

2.1.1 Optimization Problems

Most of the problems we have dealt with thus far in the class are **optimization problems**, in which the goal is to find the maximum or minimum value of a quantity and maybe also find the particular solution that achieves this maximum/minimum value. For example, in the SHORTEST-PATH problem, we are given an undirected graph $G = (V, E)$ and vertices s and t, and we wish either to find the length of the shortest path from s to t, or to find the shortest path from s to t itself.

2.1.2 Decision Problems

NP-Completeness however applies to **decision problems**, in which the answer is simply "yes" or "no" (or formally 1 or 0). This is not restrictive at all: we can always cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given a directed graph $G =$

(V, E), vertices s and t, and an integer k, does a path exist from s to t that consists of at most k edges?

There is an interesting relationship between a decision problem and its related optimization problem: the decision problem is in a sense "easier", or at least "no harder". As a specific example, we can solve `PATH` by solving SHORTEST-PATH and then comparing the number of the edges in the shortest path found to the value of the decision-problem parameter k. Stated in a different way, if an optimization problem is easy, its corresponding decision problem is also easy.

2.2 Encodings

An **abstract problem** Q is a binary relation on a set I of **problem instances** and a set S of problem **solutions**. The theory of NP-completeness focuses on **decision problems**. In this case, an **abstract decision problem** can be viewed as a function that maps the instance set I to the solution set $\{0, 1\}$. For example, if $i = (G = (V, E), s, t, k)$ is an instance of the decision problem PATH, then $\text{PATH}(i) = 1$ if a shortest path from s to t has at most k edges and $\text{PATH}(i) = 0$ otherwise.

The instances of an abstract problem have to represented in a way that a computer understands them. An **encoding** of a set S of abstract objects is a mapping e from S to the set of binary strings. A problem whose instance set is the set of binary strings is called a **concrete problem**.

An algorithm **solves** a concrete problem in time $\mathcal{O}(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $\mathcal{O}(T(n))$ time. A concrete problem is **polynomial-time solvable** if there exists an algorithm to solve it in time $\mathcal{O}(n^c)$ for some constant c.

For the rest of this lecture, we will suppress details of underlying encodings of problem instances. There may be more than one natural encoding, but typically these differ in size by polynomial factors and we can convert between them in polynomial time. The specifics of an encoding, then, should not affect the notion of polynomial-time solvability. With this in mind, we will use $\langle \cdot \rangle$ to denote some natural binary encoding. For example, $\langle G \rangle$ is a binary encoding of graph G and $\langle G, s, t, k \rangle$ is a binary encoding of an instance of the PATH problem.

2.3 Languages

An **alphabet** Σ is a finite set of symbols. A language L over Σ is any set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{0, 10, 100, 110, \dots\}$ is the language of binary representations of even numbers. The **empty string** is denoted by ϵ, the **empty language** is denoted by $\emptyset$ and the language of all strings over Σ is denoted by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 11, 000, \dots\}$ is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

A variety of operations can be performed on languages. The set-theoretic operations, such as **union** and **intersection** follow directly from the set-theoretic definitions. The **complement** of a language L is defined as $\bar{L} = \Sigma^* - L$.

The set of instances for any decision problem Q is simply the set Σ^* where $\Sigma = \{0, 1\}$. Because Q is entirely characterized by the problem instances that produce a 1 as an answer, Q can be also viewed as a language L over $\Sigma = \{0, 1\}$ where $L = \{x \in \Sigma^* : Q(x) = 1\}$.

For example, the decision problem `PATH` has the corresponding language:

$$\texttt{PATH} = \{\langle G, s, t, k \rangle : \begin{array}{l} G \ \texttt{is an undirected graph}, \\ s, t \in V, \end{array}$$

$$k \geq 0,$$
$$\exists \text{ path } p \text{ from } s \text{ to } t \text{ in } G, |p| \leq k\}.$$

The language framework allows us to express concisely the relation between decision problems and the algorithms that solve them. An algorithm A accepts a string $x \in \{0,1\}^*$ if, given input x, the algorithm's output is $A(x) = 1$.

The language **accepted** by an algorithm A is the set of strings that the algorithm accepts. $L = x \in \{0,1\}^* : A(x) = 1\}$. An algorithm A **rejects** a string x if $A(x) = 0$.

If a language L is accepted by an algorithm A, this does not necessarily imply that the algorithm A will reject a string $x \notin L$ provided as input to it. It might be the case that the algorithm may loop forever.

A language L is **decided** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.

A language L is **accepted in polynomial time** by an algorithm A if it is accepted by A and in addition there exists a constant c such that for any length-n string $x \in L$, algorithm A accepts x in time $\mathcal{O}(n^k)$.

A language L is **decided in polynomial time** by an algorithm A if there exists a constant c such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $\mathcal{O}(n^k)$.

Pay attention to the subtle technicality: to *accept* a language, an algorithm need only produce an answer when provided a string in L, but to *decide* a language it must correctly accept or reject every string in $\{0,1\}^*$.

We can informally define a **complexity class** as a set of languages, membership in which is determined by a **complexity measure** such as running time or space, of an algorithm that determines whether a given string x belongs in L.

Under the languages framework, the complexity class P can be defined as follows:

$$P = \{L \subset \{0,1\}^* : \texttt{there exists an algorithm } A \texttt{ that decides } L \texttt{ in polynomial time.}\}$$

Theorem 2.1. P=$\{L : L$ *is accepted by a polynomial-time algorithm*$\}$.

Proof. Since the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm.

Let L be any language accepted by some polynomial-time algorithm A. We will construct another polynomial-time algorithm A' that decides L by using a "simulation argument". Because A accepts L in time $\mathcal{O}(n^k)$ for some constant c, there also exists a constant k such that A accepts L in at most kn^c steps. For any input string x, the algorithm A' simulates kn^c steps of A. After simulating these kn^c steps, A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0. The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and hence A' is a polynomial-time algorithm that decides L. □

2.4 Examples of languages in P

Some examples of languages in P are:

- $$\texttt{PRIMES} = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$$

- $$\texttt{PATH} = \{\langle G, s, t, k\rangle : \begin{array}{l} G \ \texttt{ is an undirected graph}, \\ s, t \in V, \\ k \geq 0, \\ \exists \text{ path } p \text{ from } s \text{ to } t \text{ in } G, |p| \leq k\}. \end{array}$$

- $$\texttt{EULER-CYCLE} = \{\langle G\rangle : G \ \texttt{contains an Euler cycle}\}$$

We also note that notion of polynomial-time computation is equivalent for the models:

- RAM
- Turing Machine
- PRAM with polynomial number of processors
- λ-calculus

The class of algorithms that run in polynomial time has among others, the following nice **closure** property: if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial-time, then it runs in polynomial time. It is a fairly straightforward homework exercise to prove this statement.

3 Polynomial-time Verification and NP

This section studies algorithms that verify membership in languages. Before diving into the formal definitions, let us consider two decision problems that look slightly different but belong to fundamentally different complexity classes.

Definition 3.1. A **hamiltonian** cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex $v \in V$. A graph that contains a hamiltonian cycle is said to be **hamiltonian**; otherwise it is **non-hamiltonian**.

Definition 3.2. A **eulerian** cycle of an undirected graph $G = (V, E)$ is a cycle that contains every edge $e \in E$ exactly once. A graph that contains an Euler cycle is said to be **eulerian**; otherwise it is **non-eulerian**.

Formally, the problems of deciding whether an undirected graph $G = (V, E)$ is hamiltonian or eulerian are defined by the languages:

$$\texttt{HAM-CYCLE} = \{\langle G\rangle : G \ \texttt{contains a Hamilton cycle}\}$$

$$\texttt{EULER-CYCLE} = \{\langle G\rangle : G \ \texttt{contains an Euler cycle}\}$$

Lemma 3.3. *EULER-CYCLE* $\in P$.

Proof. Euler's theorem states that an undirected graph $G = (V, E)$ contains an Euler cycle if and only if it is connected and every vertex has an even degree. □

As of today, we do not know of a polynomial-time algorithm that decides HAM-CYCLE. Let us consider an easier problem. Suppose that someone claims they have found a hamilton cycle in an undirected graph $G = (V, E)$. As proof, they provide the hamilton cycle: permutation that dictates the order in which the vertices are visited. We can then easily enough verify the proof, by checking that the given cycle is indeed a hamilton cycle. We do the latter by checking that it is indeed a permutation of the vertices in V and whether each of the consecutive edges along the cycle actually exist in the graph. The verification can be executed in time $\mathcal{O}(n^2)$, where $n = |\langle G \rangle|$.

We formally define a **verification algorithm** as a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a **certificate**. A two-argument algorithm A **verifies** an input string x if there exists a certificate y such that $A(x, y) = 1$. The **language verified** by a verification algorithm is:

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$.

The **complexity class** NP is the class of languages that can be verified by a polynomial-time algorithm. Formally,

Definition 3.4. A language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that:

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^*, |y| = \mathcal{O}(|x|^c) : A(x, y) = 1\}.$$

The **complexity class** $co - NP$ is the class of languages whose complementary languages can be verified by a polynomial-time algorithm. Formally,

Definition 3.5. A language L belongs to $co-NP$ if and only if there exist a two-input polynomial-time algorithm A and a constant c such that:

$$\bar{L} = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^*, |y| = \mathcal{O}(|x|^c) : A(x, y) = 0\}.$$

Lemma 3.6. *The following are true:*

- $P \subseteq NP$
- $P \subseteq co - NP$

Proof. We prove the first statement. If $L \in P$, then $L \in NP$, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly the input strings of L.

The proof of the second statement is similar. □

The big question in complexity theory and theoretical computer science in general is whether $NP \subseteq P$, which would imply that $P = NP$. It is not known, as of today, if $P = NP$. Intuitively, the class P consists of problems that can be solved quickly, whereas the class NP consists of problems for which a candidate solution can be verified quickly.

There are other fundamental questions that remain unsolved. We do not even know whether the class NP is closed under complement, i.e. whether $NP = co - NP$.

The 4 possibilities for the relationships between the complexity classes $P, NP, co - NP$ are illustrated in the following picture (from CLRS, chapter 34):

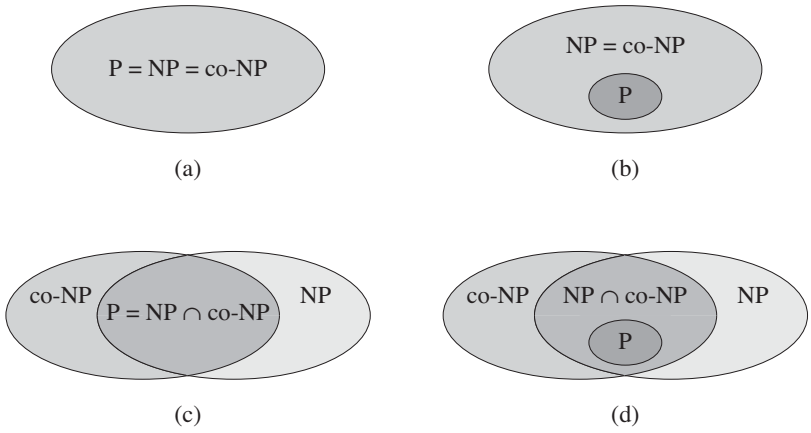

Figure 1: (a) $P = NP = co-NP$. This is considered as the most unlikely case. (b) If NP is closed under complement, $NP = co-NP$ but not necessarily $P = NP$ in this case. (c) $P = NP \cap co-NP$ but NP is not closed under complement. (d) $NP \neq co-NP$ and $P \neq NP \cap co-NP$. This is considered the most likely case.

4 Reducibility & NP-completeness

Definition 4.1. Language L_1 is **polynomial-time reducible** to language L_2, which we denote by $L_1 \leq_P L_2$ if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that

$$x \in L_1 \iff f(x) \in L_2, \forall x \in \{0,1\}^* \tag{1}$$

Lemma 4.2. *If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies that $L_1 \in P$.*

Proof. Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial-time reduction algorithm that computes the reduction function f. We will construct a polynomial-time algorithm A_1 that decides L_1.

For a given input $x \in \{0,1\}^*$ algorithm A_1 uses F to transform x into $f(x)$ and then it uses A_2 to test whether $f(x) \in L_2$. Algorithm A_1 takes the output from algorithm A_2 and produces that answer as its own output. The correctness of A_1 follows from the definition of the polynomial-time reducibility. A_1 runs in polynomial time, since both F and A_2 run in polynomial time. □

Definition 4.3. A language $L \subseteq \{0,1\}^*$ is **NP-hard** if $L' \leq_P L, \forall L' \in NP$.

Definition 4.4. A language is **NP-complete** if:

1. $L \in NP$
2. L is NP-hard

We define NPC to be the set of NP-complete languages.

Theorem 4.5. *If any NP-complete problem is polynomial-time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.*

Proof. Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have that $L' \leq_P L$ by the second property of the definition of NP-completeness. Hence, by Lemma 4.2, we also have that $L' \in P$, which proves the first statement of the theorem.

To prove the second statement, notice that is is just the contrapositive of the first statement. □

5 An NP-Complete Problem

A **boolean formula** is built from **variables**, $x_1, x_2, \cdots \in \{0, 1\}$, **operators** AND (**conjuction**, denoted by $\wedge$), OR (**disjunction**, denoted by $\vee$), NOT (**negation**, denoted by $\neg$) and parentheses. A **literal** is either a variable, (x_i, also called **positive literal**) or a negated variable ($\neg x_i$, also called **negative literal**). A **clause** is **disjunction** of one or more literals.

A formula ϕ is in **conjunctive normal form**, or **CNF**, if it is a **conjuction** of one or more clauses. For example, $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_5)$.

A formula ϕ is **satisfiable** if and only if there is an assigment of values from $\{0, 1\}$ to its variables such that it evaluates to 1; otherwise it is **unsatisfiable**.

The `CNF-SAT` problem is formally defined as follows:

$$\texttt{CNF-SAT} = \{\langle \phi \rangle : \phi \texttt{ is CNF and satisfiable}\}$$

Theorem 5.1 (Cook, Levin)**.** *`CNF-SAT` is NP-complete.*