MA421 Advanced Algorithms Lent 2019

1 $\mathcal{NP}$-Completeness

Reading: Cormen et al., *Introduction to Algorithms*, Third Edition, Sections 34.1-34.5.

1.1 Efficient Computation, Decision Problems, and Complexity Class $\mathcal{P}$

It is commonly accepted that polynomial-time algorithms are *efficient algorithms* and any computational problem solved by a polynomial-time algorithm is *tractable*. Reasoning for these conventions include (i) that it is very rare that a polynomial-time solvable problem require running time in the order of a very high-degree polynomial, (ii) that, for many proposed models of computation (RAM, Turing Machines, etc.), the class of polynomial-time-solvable problems is the same, and (iii) polynomial-time-solvable problems have nice closure properties due to the closure properties of polynomials.

In the study of computation and, in particular, in $\mathcal{NP}$-completeness, it is convenient to restrict ourselves to *decision problems*.

Definition 1.1 (Problem) *A* problem *is a binary relation on problem instances and a set of problem solutions.*

For example, in a sorting problem, a problem instance would be a list of integers. The corresponding (unique) solution would be the sorted version of that instance. In a shortest path problem on a graph, an instance might be a graph G, and two vertices u and v of the graph. Then, a solution for a given instance would be a path of minimum length between the given vertices. Notice that the shortest paths in a graph are not necessarily unique.

Definition 1.2 (Decision Problem) *A* decision problem *is a problem with solutions "Yes (1)" or "No (0)."*

Focusing only on decision problems seems to be too restrictive at first sight. It turns out that, for many problems of interest, a carefully formulated decision version of a problem is not much easier than the original problem. In other words, if we can solve the corresponding decision problem in polynomial time, then we can solve the original problem in polynomial time. The decision version of the shortest path problem can be formulated by introducing another parameter k, and asking whether there exists a path between u and v in G of length at most k.

One issue we will not be concerned about much in our discussions is how the input is presented to the algorithms. We will assume a reasonable encoding of the problem instances is used. For example, a graph as a mathematical object should be presented to an algorithm by converting it to a string of symbols from some alphabet (say, $\{0, 1\}$). As long as a reasonably efficient encoding scheme for the problem instances is used, the particular encoding scheme should not make a difference. Occasionally, we use the notation $\langle\cdot\rangle$ to denote the encoding of an instance: that is, for a graph G, $\langle G\rangle$ denotes the encoding of G.

Given that we restrict ourselves to the decision problems and view inputs as strings (that encode problem instances), we can use the terminology of formal language theory. Basically, we have "simplified" the types of computations we study to be deciding whether a given input string is a member of certain subset of all finite strings over an alphabet Σ. It suffices to consider a binary alphabet: that is, $\Sigma = \{0, 1\}$.

Definition 1.3 (Language) *A* language L *over alphabet* Σ *is any set of finite strings made up from symbols from* Σ*.*

We can now make a formal definition of polynomial-time-solvable decision problems, also known as $\mathcal{P}$.

Definition 1.4 (Complexity Class $\mathcal{P}$) *Complexity class* P *is the set of languages* L *such that there is a polynomial-time algorithm that decides membership of a string in* L*.*

1.2 Efficient Verification, Complexity Class $\mathcal{NP}$

We now look at a different class of problems. As opposed to deciding whether a given input string x is in a language L, we look at verification problems, where an algorithm verifies whether x is in a language L using a suitably presented certificate (or proof) y. For example, it is intuitive to think that deciding whether there exists a path between u and v in G of length at most k is more difficult than verifying that a given path P in G is a valid path between u and v and is of length at most k. In this case, $x = \langle G, u, v, k\rangle$ and $y = \langle P\rangle$.

Definition 1.5 *A language* L *is verified by a verification algorithm* A*, if, for every string* $x \in L$*, there exists a string* y *such that* $A(x, y) = 1$ *and, for* $x \notin L$*, no such* y *exists.*

Definition 1.6 (Complexity Class $\mathcal{NP}$) *Complexity class* $\mathcal{NP}$ *is the set of languages* L *such that there exists a polynomial-time algorithm* A *and a constant* c *such that*

- *for all* $x \in L$*, there exists a* y *such that* $|y| = O(|x|^c)$ *and* $A(x, y) = 1$*; and*
- *for all* $x \notin L$*, there does not exist* y *such that* $A(x, y) = 1$*.*

Note that $\mathcal{P} \subseteq \mathcal{NP}$. The question that has been open for more than 40 years is whether $\mathcal{P}$ is actually equal to $\mathcal{NP}$.

Alternatively, $\mathcal{NP}$ is the set of languages of which "YES-instances" have "short" proofs that can be verified in polynomial time. A related complexity class is co-$\mathcal{NP}$: the set of languages of which "NO-instances" have "short" proofs that can be verified in polynomial time.

Definition 1.7 (Complexity Class co-$\mathcal{NP}$) *Complexity class* co-$\mathcal{NP}$ *is the set of languages* L *such that the complement* L^c *of* L *is in* $\mathcal{NP}$*.*

Note that $\mathcal{P} \subseteq$ co-$\mathcal{NP}$. However, it is not known whether $\mathcal{NP} =$ co-$\mathcal{NP}$ or whether $\mathcal{P} = \mathcal{NP} \cap$ co-$\mathcal{NP}$.

1.3 $\mathcal{NP}$-Complete Problems

The main reason why many believe that $\mathcal{P} \neq \mathcal{NP}$ is the existence of a class of "$\mathcal{NP}$-complete" problems. These problems are clearly in $\mathcal{NP}$. However, after years of effort by many researchers, none of these $\mathcal{NP}$-complete problems was shown to have a polynomial-time algorithm. In fact, by definition, if any one of these $\mathcal{NP}$-complete problems are shown to have a polynomial-time algorithm, then all the problems in $\mathcal{NP}$ (in particular, all $\mathcal{NP}$-complete problems) have polynomial-time algorithms; that is, $\mathcal{P} = \mathcal{NP}$.

We define $\mathcal{NP}$-complete problems to be the "hardest problems" in $\mathcal{NP}$ by using a notion of reducibility.

Definition 1.8 (Polynomial-Time Reducibility) *A language L_1 is* polynomial-time reducible *to a language L_2, denoted $L_1 \leq_{\mathrm{P}} L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that, for all $x \in \{0,1\}^*$,*

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

In the definition above, f is called the *reduction* function. The definition implies the following lemma.

Lemma 1.9 *For two languages L_1 and L_2 such that $L_1 \leq_{\mathrm{P}} L_2$, $L_2 \in \mathrm{P}$ implies $L_1 \in \mathrm{P}$.*

Using our notion of reducibility, we define the class of $\mathcal{NP}$-complete problems as follows.

Definition 1.10 ($\mathcal{NP}$-complete Problem) *A language L is* $\mathcal{NP}$-complete *if*

1. *$L \in \mathcal{NP}$, and*
2. *$L' \leq_{\mathrm{P}} L$ for every $L' \in \mathcal{NP}$ (in this case, L is said to be* $\mathcal{NP}$-hard*).*

The following theorem demonstrates why $\mathcal{NP}$-complete problems are critical in $\mathcal{P}$-vs-$\mathcal{NP}$ question.

Theorem 1.11 *If any $\mathcal{NP}$-complete problem is polynomial-time solvable, then $\mathcal{P} = \mathcal{NP}$. Equivalently, if any problem in $\mathcal{NP}$ is not polynomial-time solvable, then no $\mathcal{NP}$-complete problem is polynomial-time solvable.*

Finally, the next theorem establishes that there is an $\mathcal{NP}$-complete problem. In fact, there are many of them.

Definition 1.12 (Circuit Satisfiability Problem) *Given a circuit of AND, OR, NOT gates, determine whether the given circuit has a satisfying assignment to its inputs: one that produces an output of 1.*

Theorem 1.13 *The Circuit Satisfiability problem is $\mathcal{NP}$-complete.*

1.4 $\mathcal{NP}$-Completeness Reductions

$\mathcal{NP}$-completeness proofs are based on the following lemma:

Lemma 1.14 *If L is a language such that $L' \leq_{\mathrm{P}} L$ for some $\mathcal{NP}$-complete problem L', then L is $\mathcal{NP}$-hard.*

Thus, the "recipe" we will use for $\mathcal{NP}$-completeness reductions will be as follows. To show that a language L is $\mathcal{NP}$-complete, we shall follow the steps below.

1. Prove that $L \in \mathcal{NP}$;
2. Select a known $\mathcal{NP}$-complete language L';
3. Describe an algorithm that computes a function f mapping every instance x of L' to an instance $f(x)$ of L;
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$, for all x;

5. Prove that the algorithm computing f runs in polynomial time.

Theorem 1.15 *Satisfiability of boolean formulas (SAT) is $\mathcal{NP}$-complete.*

Proof Idea. Reduction from Circuit Satifiability. □

Theorem 1.16 *Satisfiability of 3-CNF boolean (3-SAT) formulas is $\mathcal{NP}$-complete.*

Proof Idea. Reduction from SAT. □

Theorem 1.17 *CLIQUE is $\mathcal{NP}$-complete.*

Proof Idea. Reduction from 3-SAT. □

Theorem 1.18 *VERTEX-COVER is $\mathcal{NP}$-complete.*

Proof Idea. Reduction from CLIQUE. □

Theorem 1.19 *SUBSET-SUM is $\mathcal{NP}$-complete.*

Proof Idea. Reduction from 3-SAT. □

