A An Overview of Complexity Theory for the Algorithm Designer

A.1 Certificates and the class NP

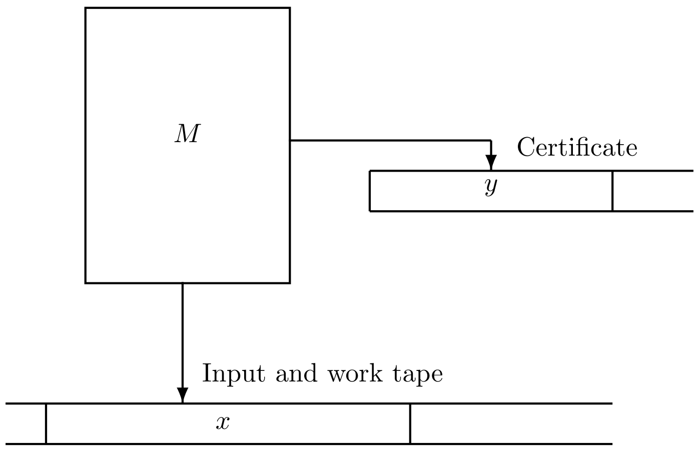
A *decision problem* is one whose answer is either "yes" or "no". Two examples are:

SAT: Given a Boolean formula in conjunctive normal form, f, is there is a satisfying truth assignment for f?
Cardinality vertex cover: Given an undirected graph G and integer k, does G have a vertex cover of size $\leq k$?

For any positive integer k, we will denote by kSAT the restriction of SAT to instances in which each clause contains at most k literals.

It will be convenient to view a decision problem as a language, i.e., a subset of $\{0,1\}^*$. The language consists of all strings that encode "yes" instances of the decision problem. A language $L \in \mathbf{NP}$ if there is a polynomial p and a polynomial time bounded Turing machine M, called the *verifier*, such that for each string $x \in \{0,1\}^*$:

- if $\boldsymbol{x} \in L$, then there is a string y (the certificate) of polynomially bounded length, i.e., $|y| \leq p(|x|)$, such that $M(x,y)$ accepts, and
- if $x \notin L$, then for any string y, such that $|y| \leq p(|x|)$, $M(x,y)$ rejects.



String y that helps ascertain that x is a "yes" instance will be called a *Yes certificate*. We will also refer to y as a *proof* or a *solution*; in the context of randomized computation, it is also referred to as a *witness*. Thus, **NP** is the class of languages that have "short, quickly verifiable" Yes certificates.

For example, the verifier for cardinality vertex cover assumes that y specifies a subset of the vertices. It checks whether this subset is indeed a vertex cover and is of the desired size bound. (Observe that no claim has been made about the time needed to actually *find* such a certificate.) It is also easy to see that the class **NP** defined above is precisely the class of languages that are decidable by nondeterministic polynomial time Turing machines (see Section A.6 for references), hence the name.

A language L belongs to the class co-**NP** iff $\overline{L} \in \mathbf{NP}$. Thus, co-**NP** is the class of languages that have "short, quickly verifiable" No certificates. For instance, let L be the language consisting of all prime numbers. This language allows No certificates: a factorization for number n is proof that $n \notin L$. Hence $L \in$ co-**NP**. Interestingly enough, $L \in \mathbf{NP}$ as well (see Exercise 1.13), though it is not known to belong to **P**.

A.2 Reductions and NP-completeness

Next, let us introduce the crucial notion of a *polynomial time reduction*. Let L_1 and L_2 be two languages in **NP**. We will say that L_1 reduces to L_2, and write $L_1 \preceq L_2$, if there is a polynomial time Turing machine T which given a string $x \in \{0,1\}^*$, outputs string y such that $x \in L_1$ iff $y \in L_2$. In general, T does not have to decide whether x is a "yes" or a "no" instance in order to output y. Clearly, if $L_1 \preceq L_2$ and L_2 is polynomial time decidable, then so is L_1.

A language L is **NP**-*hard* if for every language $L' \in \mathbf{NP}$, $L' \preceq L$. A language L is **NP**-*complete* if $L \in \mathbf{NP}$, and L is **NP**-hard. An **NP**-complete language L is a hardest language in **NP**, in the sense that a polynomial time algorithm for L implies a polynomial time algorithm for every language in **NP**, i.e., it implies $\mathbf{P} = \mathbf{NP}$.

The central theorem of complexity theory gives a proof of **NP**-hardness for a natural problem, namely SAT. The idea of the proof is as follows. Let L be an arbitrary language in **NP**. Let M be a nondeterministic polynomial time Turing machine that decides L, and let p be the polynomial bounding the running time of M. The proof involves showing that there is a deterministic polynomial time Turing machine T, that "knows" M and p, and given a string $x \in \{0,1\}^*$, outputs a SAT formula f such that each satisfying truth assignment of f encodes an accepting computation of M on input x. Thus, f is satisfiable iff there is an accepting computation of M on input x, i.e., iff $x \in L$.

Once one problem, namely SAT, has been shown to be **NP**-hard, the hardness of other natural problems can be established by simply giving polynomial time reductions from SAT to these problems (see Exercise 1.11). Perhaps the most impressive feature of the theory of **NP**-completeness is the ease with which the latter task can be accomplished in most cases, so that with relatively little work, a lot of crucial information is obtained. Other than a handful of (important) problems, most natural problems occurring in **NP** have been classified as being either in **P** or being **NP**-complete. Indeed, it is remarkable to note that other basic complexity classes, defined using notions of time, space and nondeterminism, also tend to have natural complete problems (under suitably defined reducibilities).

Establishing **NP**-hardness for vertex cover involves giving a polynomial time algorithm that, given a SAT formula f, outputs an instance (G, k) such that G has a vertex cover of size $\leq k$ iff f is satisfiable. As a corollary, we get that under the assumption $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial time algorithm that can distinguish "yes" instances of vertex cover from "no" instances. As stated above, this also shows that if $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial time algorithm for solving vertex cover exactly.

Considering the large and very diverse collection of **NP**-complete problems, none of which has yielded to a polynomial time algorithm for so many years, it is widely believed that $\mathbf{P} \neq \mathbf{NP}$, i.e., that there is no polynomial time algorithm for deciding an **NP**-complete language.

The $\mathbf{P} \neq \mathbf{NP}$ conjecture has a deep philosophical point to it. The conjecture asserts that the task of finding a proof for a mathematical statement is qualitatively harder than the task of simply verifying the correctness of a given proof for the statement. To see this, observe that the language

$$L = \{(S, 1^n) \mid \text{statement } S \text{ has a proof of length} \leq n\}$$

is in **NP**, assuming any reasonable axiomatic system.

A.3 NP-optimization problems and approximation algorithms

Combinatorial optimization problems are problems of picking the "best" solution from a finite set. An **NP**-*optimization problem*, Π, consists of:

- A set of *valid instances*, D_Π, recognizable in polynomial time. We will assume that all numbers specified in an input are rationals, since our model of computation cannot handle infinite precision arithmetic. The *size* of an instance $I \in D_\Pi$, denoted by $|I|$, is defined as the number of bits needed to write I under the assumption that all numbers occurring in the instance are written in binary.
- Each instance $I \in D_\Pi$ has a set of *feasible solutions*, $S_\Pi(I)$. We require that $S_\Pi(I) \neq \emptyset$, and that every solution $s \in S_\Pi(I)$ is of length polynomially bounded in $|I|$. Furthermore, there is polynomial time algorithm that, given a pair (I, s), decides whether $s \in S_\Pi(I)$.

- There is a polynomial time computable *objective function*, obj_{Π}, that assigns a nonnegative rational number to each pair (I, s), where I is an instance and s is a feasible solution for I. The objective function is frequently given a physical interpretation, such as *cost, length, weight*, etc.
- Finally, Π is specified to be either a *minimization problem* or a *maximization problem.*

The restriction of Π to unit cost instances will be called the *cardinality version* of Π.

An *optimal solution* for an instance of a minimization (maximization) problem is a feasible solution that achieves the smallest (largest) objective function value. $\text{OPT}_{\Pi}(I)$ will denote the objective function value of an optimal solution to instance I. We will shorten this to OPT when it is clear that we are referring to a generic instance of the particular problem being studied.

With every **NP**-optimization problem, one can naturally associate a decision problem by giving a bound on the optimal solution. Thus, the decision version of **NP**-optimization problem Π consist of pairs (I, B), where I is an instance of Π and B is a rational number. If π is a minimization (maximization) problem, then the answer to the decision version is "yes" iff there is a feasible solution to I of cost $\leq B$ $(\geq B)$. If so, we will say that (I, B) is a "yes" instance; we will call it a "no" instance otherwise. For example, the decision version of cardinality vertex cover is stated in Section A.1.

Clearly, a polynomial time algorithm for Π can help solve the decision version – by computing the cost of an optimal solution and comparing it with B. Conversely, hardness established for the decision version carries over to Π. Indeed hardness for an **NP**-optimization problem is established by showing that its decision version is **NP**-hard. With a slight abuse of notation, we will also say that the optimization version is **NP**-hard.

An approximation algorithm produces a feasible solution that is "close" to the optimal one, and is time efficient. The formal definition differs for minimization and maximization problems. Let Π be a minimization (maximization) problem, and let δ be a function, $\delta : \mathbf{Z}^+ \rightarrow \mathbf{Q}^+$, with $\delta \geq 1$ $(\delta \leq 1)$. An algorithm $\mathcal{A}$ is said to be a *factor δ approximation algorithm for Π* if, on each instance I, $\mathcal{A}$ produces a feasible solution s for I such that $f_{\Pi}(I, s) \leq \delta(|I|) \cdot \text{OPT}(I)$ $(f_{\Pi}(I, s) \geq \delta(|I|) \cdot \text{OPT}(I))$, and the running time of $\mathcal{A}$ is bounded by a fixed polynomial in $|I|$. Clearly, the closer δ is to 1, the better is the approximation algorithm.

On occasion we will relax this definition and will allow $\mathcal{A}$ to be randomized, i.e., it will be allowed to use the flips of a fair coin. Assume we have a minimization problem. Then we will say that $\mathcal{A}$ is a *factor δ randomized approximation algorithm for Π* if, on each instance I, $\mathcal{A}$ produces a feasible solution s for I such that

$$\mathbf{Pr}[f_{\Pi}(I, s) \ \leq \ \delta(|I|) \cdot \text{OPT}(I)] \geq \frac{1}{2},$$

where the probability is over the coin flips. The running time of $\mathcal{A}$ is still required to be polynomial in $|I|$. The definition for a maximization problem is analogous.

Remark A.1 Even though δ has been defined to be a function of the size of the input, we will sometimes pick δ to be a function of a more convenient parameter. For instance, for the set cover problem (Chapter 2), we will pick this parameter to be the number of elements in the ground set.

A.3.1 Approximation factor preserving reductions

Typically, polynomial time reductions map optimal solutions to optimal solutions; however, they do not preserve near-optimality of solutions. Indeed, all **NP**-complete problems are equally hard from the viewpoint of obtaining exact solutions. However, from the viewpoint of obtaining near-optimal solutions, they exhibit the rich set of possibilities alluded to earlier.

In this book we will encounter pairs of problems which may look quite different superficially, but whose approximability properties are closely linked (e.g., see Exercise 19.13). Let us define a suitable reducibility in order to formally establish such connections. Several reductions have been defined that preserve constant factor approximability. The reducibility stated below is a stringent version of these, and actually preserves the constant itself. Pair of problems that are linked in this manner are either both minimization problems or both maximization problems.

Let Π_1 and Π_2 be two minimization problems (the definition for two maximization problems is quite similar). An *approximation factor preserving reduction* from Π_1 to Π_2 consists of two polynomial time algorithms, f and g, such that

- for any instance I_1 of Π_1, $I_2 = f(I_1)$ is an instance of Π_2 such that $\mathrm{OPT}_{\Pi_2}(I_2) \le \mathrm{OPT}_{\Pi_1}(I_1)$, and
- for any solution t of I_2, $s = g(I_1, t)$ is a solution of I_1 such that

$$\mathrm{obj}_{\Pi_1}(I_1, s) \le \mathrm{obj}_{\Pi_2}(I_2, t).$$

It is easy to see that this reduction, together with an α factor algorithm for Π_2, gives an α factor algorithm for Π_1 (see Exercise 1.16).

A.4 Randomized complexity classes

Certain **NP** languages[1] are characterized by the fact that they possess an abundance of Yes certificates, which renders them essentially tractable, assuming availability of a source of random bits. Such languages belong to the

[1] The definitions of this section will be useful in Chapter 29.

class **RP**, short for *Randomized Polynomial Time.* A language $L \in \mathbf{RP}$ if there is a polynomial p and a polynomial time bounded Turing machine M such that for each string $x \in \{0,1\}^*$:

- if $\boldsymbol{x} \in L$, then $M(x,y)$ accepts for at least half the strings y of length $p(|x|)$, and
- if $x \notin L$, then for any string y of length $p(|x|)$, $M(x,y)$ rejects.

Clearly, $\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{NP}$. Suppose language $L \in \mathbf{RP}$. On input x, we will pick a random string, y, of length $p(|x|)$ and will run $M(x,y)$. Clearly, the entire computation takes polynomial time. We may erroneously reject x even though $x \in L$. However, the probability of this is at most $1/2$. Let us call this the error probability. By the usual trick of making repeated independent runs, we can reduce the error probability to inverse exponential in the number of runs.

A language L belongs to the class co-**RP** iff $\overline{L} \in \mathbf{RP}$. Such a language has an abundance of No certificates. The corresponding machine may make an error on inputs $x \notin L$. Finally, let us define **ZPP**, short for *Zero-error Probabilistic Polynomial Time*, to be the class of languages for which there is a randomized Turing machine (i.e., a Turing machine equipped with a source of random bits) that always terminates with the correct answer and whose *expected* running time is polynomial. It is easy to see (Exercise 1.17) that

$$L \in \mathbf{ZPP} \text{ iff } L \in (\mathbf{RP} \cap \text{co-}\mathbf{RP}).$$

$\mathbf{DTIME}(t)$ denotes the class of problems for which there is a deterministic algorithm running in time $O(t)$. Thus, $\mathbf{P} = \mathbf{DTIME}(\mathit{poly}(n))$, where $\mathit{poly}(n) = \bigcup_{k \geq 0} n^k$. $\mathbf{ZTIME}(t)$ denotes the class of problems for which there is a randomized algorithm running in expected time $O(t)$. Thus, $\mathbf{ZPP} = \mathbf{ZTIME}(\mathit{poly}(n))$.

A.5 Self-reducibility

Most known problems in **NP** exhibit an interesting property, called self-reducibility, which yields a polynomial time algorithm for finding a solution (a Yes certificate), given an oracle for the decision version. A slightly more elaborate version of this property yields an exact polynomial time algorithm for an **NP**-optimization problem, again given an oracle for the decision version. In a sense this shows that the difficult core of **NP** and **NP**-optimization problems is their decision versions (see Section 16.2 and Exercise 28.7 for other fundamental uses of self-reducibility).

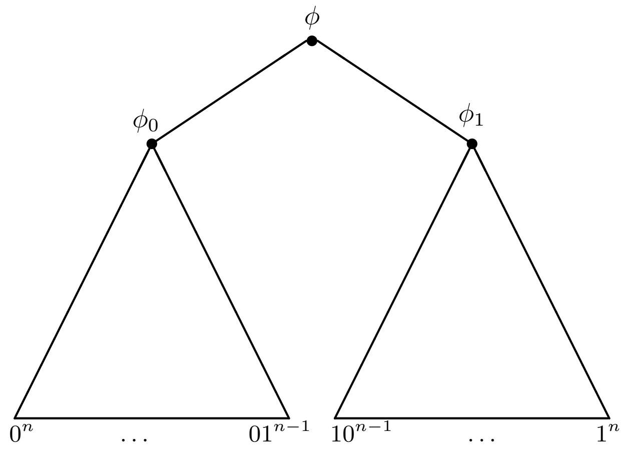
Perhaps the simplest setting to describe self-reducibility is SAT. Let ϕ be a SAT formula on n Boolean variables $x_1, \ldots, x_n$. We will represent a truth assignment to these n variables as n-bit 0/1 vectors (True $= 1$ and False $= 0$). Let S be the set of satisfying truth assignments, i.e., solutions, to ϕ.

The important point is that for the setting of x_1 to 0 (1), we can find, in polynomial time, a formula ϕ_0 (ϕ_1) on the remaining $n-1$ variables whose solutions, S_0 (S_1), are precisely solutions of ϕ having $x_1 = 0$ ($x_1 = 1$).

Example A.2 Suppose $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_4)$. Then $\phi_0 = (x_2 \vee x_3)$ and $\phi_1 = (x_2 \vee x_4)$ □

Using this property, an oracle for the decision version of SAT can be used to find a solution to ϕ, assuming it is satisfiable, as follows. First check whether ϕ_0 is satisfiable. If so, set $x_0 = 0$, and find any solution to ϕ_0. Otherwise, set $x_1 = 1$ (in this case ϕ_1 must be satisfiable), and find a solution to ϕ_1. In each case the problem has been reduced to a smaller one, and we will be done in n iterations.

The following representation will be particularly useful. Let T be a binary tree of depth n whose leaves are all n-bit 0/1 strings, representing truth assignments to the n variables. Leaves that are solutions to ϕ are marked special. The root of T is labeled with ϕ and its internal nodes are labeled with formulae whose solutions are in one-to-one correspondence with the marked leaves in the subtree rooted at this node. Thus, the 0th child of the root is labeled with ϕ_0 and the 1st child is labeled with ϕ_1. Tree T is called the *self-reducibility tree* for instance ϕ.



We will formalize the notion of self-reducibility for **NP**-optimization problems. Formalizing this notion for **NP** problems is an easier task and is left as Exercise 1.15.

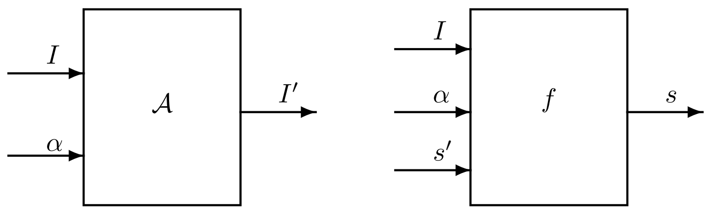
First, let us illustrate self-reducibility for cardinality vertex cover. Observe that an oracle for the decision version enables us to compute the size of the optimal cover, OPT(G), by binary search on k. To actually find an optimal cover, remove a vertex v together with its incident edges to obtain graph G', and compute OPT(G'). Clearly, v is in an optimal cover iff OPT(G') = OPT(G) − 1. Furthermore, if v is in an optimal cover, then any optimal cover in G', together with v, is an optimal cover in G. Otherwise, any optimal cover for G must contain all neighbors, say $N(v)$, of v (in order to cover all edges incident at v). Let G'' be the graph obtained by removing v and $N(v)$ from G. Any optimal cover in G'', together with $N(v)$, is an optimal cover in G. Thus, in both cases, we are left with the problem of finding an optimal cover in a smaller graph, G' or G''. Continuing this way, an optimal cover in G can be found in polynomial time.

The above-stated reduction from the cardinality vertex cover problem to its decision version works because we could demonstrate polynomial time algorithms for

- obtaining the smaller graphs, G' and G'',
- computing the size of the best cover in G, consistent with the atomic decision, and
- constructing an optimal cover in G, given an optimal cover in the smaller instance.

The exact manner in which self-reducibility manifests itself is quite different for different problems. Below we state a fairly general definition that covers a large number of problems. In the interest of conveying the main idea behind this important concept, we will provide an intuitive, though easily formalizable, definition.

We will assume that solutions to an instance I of **NP**-optimization problem Π have *granularity*, i.e., consist of smaller pieces called *atoms* that are meaningful in the context of the problem. For instance, for cardinality vertex cover, the atoms consist of specifying whether or not a certain vertex is in the cover. Clearly, for vertex cover this can be done using $O(\log n)$ bits. Indeed, all problems considered in this book have granularity $O(\log n)$. Let us assume this for problem Π.



We will say that problem Π is *self-reducible* if there is a polynomial time algorithm, $\mathcal{A}$, and polynomial time computable functions, $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$, satisfying the following conditions.

- Given instance I and an atom α of a solution to I, $\mathcal{A}$ outputs an instance I_α. We require that $|I_\alpha| < |I|$. Let $S(I \mid \alpha)$ represent the set of feasible solutions to I that are consistent with atom α. We require that the feasible solutions of I_α, $S(I_\alpha)$, are in one-to-one correspondence with $S(I \mid \alpha)$. This correspondence is given by the polynomial time computable function $f(\cdot,\cdot,\cdot)$ as follows.

$$f(I,\alpha,\cdot) : S(I_\alpha) \to S(I \mid \alpha).$$

- The correspondence $f(I,\alpha,\cdot)$ preserves order in the objective function values of solutions. Thus, if s'_1 and s'_2 are two feasible solutions of I_α with $\text{obj}_\Pi(I_\alpha, s'_1) \leq \text{obj}_\Pi(I_\alpha, s'_2)$, and $f(I,\alpha,s'_1) = s_1$ and $f(I,\alpha,s'_2) = s_2$, then $\text{obj}_\Pi(I, s_1) \leq \text{obj}_\Pi(I, s_2)$.
- Given the cost of an optimal solution to I_α, the cost of the best solution in $S(I \mid \alpha)$ can be computed efficiently, and is given by $g(I,\alpha,\text{OPT}(I_\alpha))$.

Theorem A.3 *Let Π be an* **NP***-optimization problem that is self-reducible. There is a polynomial time (exact) algorithm for Π, given an oracle, $\mathcal{O}$, for the decision version of Π.*

Proof: As remarked earlier, via a suitable binary search we can use $\mathcal{O}$ to compute the cost of the optimal solution to an instance in polynomial time.

We will derive polynomial time algorithm $\mathcal{R}$ for solving Π exactly. Assume that $\mathcal{A}$, f, and g are defined as above for the self-reducibility of Π. Let I be an instance of Π. $\mathcal{R}$ first finds one atom of an optimal solution to I. An atom, say β, satisfies this condition iff $g(I,\beta,\text{OPT}(I_\beta)) = \text{OPT}(I)$, where $I_\beta = \mathcal{A}(I,\beta)$. Since atoms are only $O(\log n)$ bits long, finding such an atom involves simply searching the polynomially many possibilities. Let α be the atom found, and let $I_\alpha = \mathcal{A}(I,\alpha)$. $\mathcal{R}$ then recursively computes an optimal solution, say s', to I_α. Finally, it outputs $f(I,\alpha,s')$, which is guaranteed to be an optimal solution to I. Since $|I_\alpha| < |I|$, the recursion also takes only polynomial time. □

Remark A.4 The number of strings of length $O(\log n)$ that algorithm $\mathcal{R}$ needs to examine for finding a good atom depends on the specific problem. For instance, in the case of cardinality vertex cover we picked an arbitrary vertex, say v, and considered only two atoms, that v is or isn't in the cover.

A.6 Notes

The definition of an **NP**-optimization problem is due to Krentel [178]. Approximation factor preserving reductions are a stringent version of L-reducibility from Papadimitriou and Yannakakis [218]. Self-reducibility was first defined by Schnorr [234]. See Khuller and Vazirani [171] for a problem that is not self-reducible, assuming $\mathbf{P} \neq \mathbf{NP}$. For further information on **NP**-completeness and complexity theory see Garey and Johnson [93] and Papadimitriou [216].