5 Complexity of Problems and Algorithms

We have seen that the simplex algorithm inspects basic feasible solutions and is guaranteed to find an optimal solution after a finite number of steps. We have also observed, however, that the number of basic feasible solutions is generally exponential in n, and going over all of them would take a long time. It is therefore an interesting question whether there really are cases where the simplex algorithm has to look at a significant fraction of the set of all basic feasible solutions. If this was the case, we could then ask whether a similar property holds for every algorithm that solves the linear programming problem.

5.1 Asymptotic Complexity

Formally, an instance of an optimization problem is given by its input. In the case of linear programming, for example, this input consists of two vectors $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ and a matrix $A \in \mathbb{R}^{m \times n}$. If each real value is represented using at most k bits, the whole instance can be described by a string of $(mn + m + n)k$ bits. We will refer to this parameter as the *input size*.

A sensible way to define the complexity of a problem is via the complexity of the fastest algorithm that solves it. The latter is typically measured in terms of the number of arithmetic or bit-level operations as a function of the input size, ignoring lower-order terms resulting from details of the implementation. The following notation is useful in this context: given two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, write

- $f(n) = O(g(n))$ if there exist constants c and n_0 such that for every $n \geqslant n_0$, $f(n) \leqslant cg(n)$,
- $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that for every $n \geqslant n_0$, $f(n) \geqslant cg(n)$, and
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

In other words, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ mean that the asymptotic growth of $f(n)$ is respectively bounded from above or below by $g(n)$, up to a constant factor. Gaussian elimination for example shows that solving a linear system $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ has arithmetic complexity $O(n^3)$. The same bound can also be shown to hold for bit complexity.

5.2 P, NP, and Polynomial-Time Reductions

In computational complexity theory, *efficient computation* is typically associated with running times that are at most *polynomial* in the size of the input. In many situations

of interest, and also in this course, it suffices to study complexity-theoretic questions for *decisions problems*, i.e., problems where the answer is just a single bit. An example of a decision problem in the context of linear programming would be the following: given a linear program and a number $k \in \mathbb{R}$, does the optimal solution of the linear program have value less than k? Formally, a decision problem can be described by a language $L \subseteq \{0,1\}^*$, containing precisely the instances for which the answer is 1 in some encoding as strings of bits.

One might expect the answer to the question whether a particular problem can be solved efficiently to depend a lot on the details of the computational model one is using. Quite surprisingly, this turns out not to be the case: all computational models that are known to be physically realizable can simulate each other, and a particular model, the *Turing machine*, can simulate all others with polynomial overhead. A Turing machine has a finite number of states, finite control, and a readable and writable tape that can store intermediary results as strings of bits. The Turing machine is started with the input written on the tape. It then runs for a certain number of steps, and when it halts the output is inferred from the state or the contents of some designated part of the tape. In the context of decision problems, a Turing machine is said to *accept* input $x \in \{0,1\}^*$ if it halts with output 1.

The most important open problem in complexity theory is concerned with the relationship between the complexity classes P and NP. P is the class of decision problems that can be solved in *polynomial time*. Formally, a function $f : \{0,1\}^* \to \{0,1\}^*$ is computable in polynomial time if there exists a Turing machine M and $k \in \mathbb{N}$ with the following property: for every $x \in \{0,1\}^*$, if M is started with input x, then after $O(|x|^k)$ steps it halts with output $f(x)$. NP is the class of decision problems for which a given solution can be verified in polynomial time. Formally, $L \subseteq \{0,1\}^*$ is in NP if there exists a Turing machine M and $k \in \mathbb{N}$ with the following property: for every $x \in \{0,1\}^*$, $x \in L$ if and only if there exists a certificate $y \in \{0,1\}^*$ with $|y| = O(|x|^k)$ such that M accepts (x, y) after $O(|x|^k)$ steps. The name NP, for *nondeterministic polynomial time*, derives from an alternative definition as the class of decision problems solvable in polynomial time by a nondeterministic Turing machine. A nondeterministic Turing machine is a Turing machine that can make a non-deterministic choice at each step of its computation and is required to accept $x \in L$ only for some sequence of these choices. Finding a solution is obviously at least as hard as verifying a solution described by a certificate. Most people believe that it must be strictly harder, i.e., that $P \neq NP$.

The relative complexity of different decision problems can be captured in terms of *reductions*. Intuitively, a reduction from one problem to another transforms every instance of the former into an equivalent instance of the latter, where equivalence means that both of them yield the same decision. For this transformation to preserve the complexity of the original problem, the reduction should of course have less power than is required to actually solve the original problem. In our case it makes sense to use reductions that can be computed in polynomial time. A decision problem $L \subseteq \{0,1\}^*$ is called *polynomial-time reducible* to a decision problem $K \subseteq \{0,1\}^*$, denoted $L \leqslant_p K$,

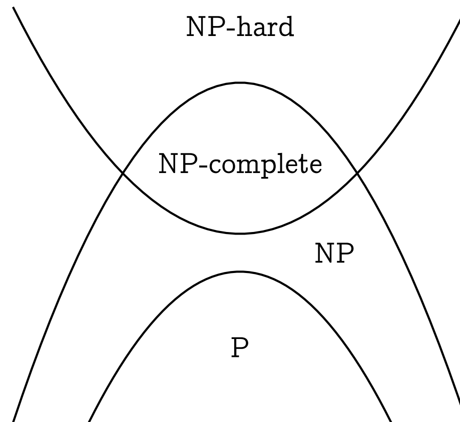

Figure 5.1: Relationship between P, NP, and the sets of NP-hard and NP-complete problems. It is not known whether the intersection between P and the set of NP-complete problems is empty. If it is not, then P = NP.

if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$ computable in polynomial time such that for every $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) \in K$. A problem K is called *NP-hard* if for every problem L in NP, $L \leqslant_p K$. A problem is called *NP-complete* if it is both in NP and NP-hard. The relation $\leqslant_p$ is transitive. NP-complete problems are thus the hardest problems in NP, in the sense that membership of any NP-complete problem in P would imply that P = NP. The existence of NP-complete problems is less obvious, but holds nonetheless. Figure 5.1 illustrates the relationship between P and NP.

What is nice about the asymptotic worst-case notions of complexity considered above is that they do not require any assumptions about low-level details of the implementation or about the type of instances we will encounter in practice. We do, however, have to be a bit careful in interpreting results that use these notions. The fact that a problem is in P does not automatically mean that it can always be solved efficiently in practice, as the constant overhead hidden in the asymptotic notation might be prohibitively large. In fact, it does not even have to be the case that an algorithm with a polynomial worst-case running time is better in practice than an algorithm whose worst-case running time is exponential. Experience has shown, however, that for problems in P one is usually able to find algorithms that are fast in practice. On the other hand, NP-hardness of a problem does not mean that it can never be solved in practice, and we will consider approaches for solving NP-hard optimization problems in a later lecture. NP-hardness is still a very useful concept because it can help to direct efforts away from algorithms that are always efficient and toward algorithms with good practical performance.

5.3 Some NP-Complete Problems

The first problem ever shown to be NP-complete is the *Boolean satisfiability problem* (SAT), which asks whether a given Boolean formula is satisfiable. A Boolean formula

consists of a set of clauses $C_i \subseteq X$ for $i = 1, \dots, m$, where $X = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ is a set of literals. It is called satisfiable if there exists a set $S \subseteq X$ such that $|S \cap \{x_j, \bar{x}_j\}| \leqslant 1$ for all $j = 1, \dots, n$ and $|S \cap C_i| \geqslant 1$ for all $i = 1, \dots, m$. Since the set S can serve as a certificate, it is easy to see that SAT is in NP. NP-hardness can be shown by encoding the operation of an arbitrary nondeterministic Turing machine as a Boolean formula.

THEOREM 5.1 (Cook, 1971; Levin, 1973). *Boolean satisfiability is NP-complete.*

An instance of the 0—1 *integer programming problem* consists of a matrix $A \in \mathbb{Z}^{m \times n}$ and a vector $b \in \mathbb{Z}^m$, and asks whether there exists a vector $x \in \{0,1\}^n$ such that $Ax \geqslant b$. Note that this is the feasibility problem associated with a special case of the integer programs we encountered in the previous lecture, in the context of the cutting plane method.

THEOREM 5.2 (Karp, 1972). *0—1 integer programming is NP complete.*

Proof. Membership in NP is again easy to see. NP-hardness can be shown by a reduction from SAT. Consider a Boolean formula with literals $X = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ and clauses C_i, $i = 1, \dots, m$, and assume without loss of generality that $|C_i \cap \{x_j, \bar{x}_j\}| \leqslant 1$ for all $i = 1, \dots, m$ and $j = 1, \dots, n$. Now let $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$ be given by

$$a_{ij} = \begin{cases} 1 & \text{if } x_j \in C_i \\ -1 & \text{if } \bar{x}_j \in C_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } i = 1, \dots, m \text{ and } j = 1, \dots, n,$$

$$b_i = 1 - |\{j : \bar{x}_j \in C_i\}| \qquad \text{for } i = 1, \dots, m.$$

Intuitively, this integer program represents each Boolean variable by a binary variable, and each clause by a constraint that requires its literals to sum up to at least 1. To this end, the left hand side of the contraint contains x_j if the corresponding Boolean variable occurs as a positive literal in the clause, and $(1 - x_j)$ if it occurs as a negative literal. The above form is then obtained by moving all constants to the right hand side. It is now easy to see that there exists $x \in \{0,1\}^n$ such that $Ax \geqslant b$ if and only if the Boolean formula is satisfiable. □

The last problem we consider is the *traveling salesman problem* (TSP). For a given matrix $A \in \mathbb{N}^{n \times n}$ and a number $k \in \mathbb{N}$, it asks whether there exists a permutation $\sigma \in S_n$ such that $a_{\sigma(n)\sigma(1)} + \sum_{i=1}^{n-1} a_{\sigma(i)\sigma(i+1)} \leqslant k$. If the entries of the matrix A are interpreted as pairwise distances among a set of locations, we are looking for a tour with a given maximum length that visits every location exactly once and returns to the starting point. The special case where A is a symmetric binary matrix and $k = 0$ is also known as the Hamiltonian cycle problem.

THEOREM 5.3 (Karp, 1972). *TSP is NP-complete, even if $A \in \{0,1\}^{n \times n}$ symmetric and $k = 0$.*