Algorithms and complexity

Preliminary

Many problems, particularly in discrete mathematics, do not admit an analytic solution but can trivially be solved in finite time, for example, by enumerating all potential solutions. Such enumeration is a particular example of what we call an "algorithm". Intuitively, an algorithm is a finite sequence of instructions aiming to solve a specific problem. The notion of algorithms can be made absolutely formal via *Turing machines*, which, despite of their simplicity, can be used to simulate any real computational system. But for our purposes, intuitive understanding is absolutely sufficient—programs written in C/C++/Java may serve as a good base. Nonetheless, some formalism is still needed, to be able to *prove* "efficiency" of some algorithms.

Typically, we are not satisfied with just an arbitrary algorithm (or computer program). First of all, it must be *correct*, i.e., always provide a solution of the problem. Yet, among all correct algorithms, we prefer those which run *faster*. Naïve enumeration requires considering all potential solutions of the problem and usually fails to find a solution in reasonable time for "big" instances of the problem.

For example, consider the *perfect matching problem*:

> Given a graph $G = (V, E)$, find a subset of edges $M \subseteq E$ of size $|V|/2$ such that all edges in M are pairwise disjoint (i.e., M is a *perfect matching*).

Clearly, we can enumerate all possible subsets $M \subseteq E$ of size $|V|/2$ and check whether there is one that is a perfect matching. But such an algorithm would need to look at $\binom{|E|}{|V|/2}$ subsets, and therefore, run for ages if $|E|$ and $|V|$ are big (just imagine that graph the input graph might model all highways in the world). In contrast, there are much faster algorithms, which allow to solve the problem in quite a reasonable time, comparable to the time needed to *read* the input graph from memory. And this is, more or less, what we expect from all "efficient" algorithms: their running time must be comparable with the time needed to read the input data, or in other words, to the "size" of the input.

In order to combine these intuitive requirements into a rigorous theoretical framework, suitable for formal mathematical analysis of algorithms, we need to define what we mean by the "size of the input", by "the running time of the algorithm", and finally, what we mean by saying that "the running time must be comparable to the size of the input".

1 Sizes and data structures

All objects used by an algorithm (numbers, matrices, graphs, equations, etc.) must in some way be stored in the memory as a sequence of basic symbols, recognizable by the algorithm. Let Σ be a finite set of symbols, further referred to as an *alphabet*. A finite ordered sequence of symbols from Σ is called a *string*; the set of all symbols is denoted by Σ^*. The *size* of a string is the number of symbols in it. Thus, every object used by an algorithm is a string. Observe that if $\text{size}(z)$ is the size of a string z representing, for instance, a matrix, then an algorithm cannot look at all entries of the matrix in time smaller than $\operatorname{size}(z)$.

Of course, there is no unique way to represent all data structures using the symbols from Σ, and in order to proceed, we need to agree on one of them. First, we shall always assume that $\Sigma = \{0,1\}$; this naturally corresponds to real-world computers operating with *bits*. If the numbers are written in *binary encoding*, then

$$
\begin{aligned}
\operatorname{size}(z) &= 1 + \log_2(|z|+1) && \text{for } z \in \mathbb{Z}, && (1)\\
\operatorname{size}(\alpha) &= 1 + \log_2(|p|+1) + \log_2(|q|+1) && \text{for } \alpha = p/q \in \mathbb{Q} \text{ with } p,q \in \mathbb{Z}, q > 0 && (2)\\
\operatorname{size}(c) &= n + \sum_{i=1}^{n} \operatorname{size}(c_i) && \text{for } c = [c_1, c_2, \dots, c_n]^{\mathrm{T}} \in \mathbb{Q}^n, && (3)\\
\operatorname{size}(A) &= mn + \sum_{i=1}^{m}\sum_{j=1}^{n} \operatorname{size}(a_{ij}) && \text{for } A = [a_{ij}] \in \mathbb{Q}^{m\times n}. && (4)
\end{aligned}
$$

Another possibility is to write numbers in *unary encoding*, where a positive integer z is represented by a sequence of $|z|$ ones. Then definitions (1) and (2) change to

$$
\begin{aligned}
\operatorname{size}(z) &= 1 + |z| && \text{for } z \in \mathbb{Z},\\
\operatorname{size}(\alpha) &= 1 + |p| + |q| && \text{for } \alpha = p/q \in \mathbb{Q} \text{ with } p,q \in \mathbb{Z}, q > 0,
\end{aligned}
$$

while equations (2)–(4) remain the same. The third common definition of sizes is slightly idealistic and corresponds to the model, where representation of a number always takes constant space; thus, equations (1) and (2) have the form

$$
\begin{aligned}
\operatorname{size}(z) &= 1 && \text{for } z \in \mathbb{Z},\\
\operatorname{size}(\alpha) &= 1 && \text{for } \alpha = p/q \in \mathbb{Q} \text{ with } p,q \in \mathbb{Z}, q > 0.
\end{aligned}
$$

The size of a matrix is then the number of entries in it. The number 1 appearing in the formulae above actually stands for any constant; as we shall see further, the actual value of those constant does not matter for the analysis of algorithms.

Representation of other objects is also flexible. A graph $G = (V,E)$ may be expressed via adjacency lists, adjacency matrices, etc., but the size of these representations is typically $O(m+n)$, where n is the number of vertices and m is the number of edges. Linear equations $a^{\mathrm{T}}x = \beta$ or inequalities $a^{\mathrm{T}}x \leqslant \beta$ are given by a vector of coefficients a and a right-hand side β. Systems of linear equations $Ax = b$ or inequalities $Ax \leqslant b$ are given by a matrix of coefficients A and a vector of right-hand sides b. Intuitively, it is often clear what is a "natural" representation of a particular data structure on the alphabet $\Sigma = \{0,1\}$ and what is the size of that representation.

2 Problems

We distinguish between three types of "problems"; namely,

- decision problems, to which the answer is 'yes' or 'no': either there is a solution, or there is no solution;

- search problems, where the task is to find a solution if one exists;
- optimization problems, where the task is to find a "best" (with respect to some criterion) solution.

Formally, let Σ be an alphabet. A *decision problem* can be identified with a subset Π of Σ^*; then Π can be viewed a set of *inputs* to the problem, for which the answer is 'yes'. In words, we say:

> Given a string $z \in \Sigma^*$, decide if $z \in \Pi$.

For example,

- "Given a graph $G = (V, E)$, is there a perfect matching in G?" (Here Π is the set of all graphs having a perfect matching.)
- "Given a system of linear inequalities $Ax \leqslant b$, is there a solution x satisfying all these inequalities?" (Here Π is the set of all systems of linear inequalities having a solution.)
- "Given a system of linear equations $Ax = b$, is there an integral non-negative solution x satisfying all these inequalities?" (Here Π is the set of all systems of linear equations, for which there is an integral non-negative solution.)

A *search problem* may be identified with a subset Π of $\Sigma^* \times \Sigma^*$ and the question is as follows:

> Given a string $z \in \Sigma^*$, find a string $y \in \Sigma^*$ such that $(z, y) \in \Pi$, or decide that no such string exists.

We say that z is an *input* of the problem and y is a *solution*. For example,

- "Given a graph $G = (V, E)$, find a perfect matching in G, or decide that G has no perfect matching." (Here Π consists of all pairs (G, M) such that M is a perfect matching in a graph G.)
- "Given a system of linear inequalities $Ax \leqslant b$, is there a solution x satisfying all these inequalities?" (Here Π is the set of all pairs $(Ax \leqslant b, x^*)$ with $Ax^* \leqslant b$.)
- "Given a system of linear equations $Ax = b$, is there an integral non-negative solution x satisfying all these inequalities?" (Here Π is the set of all pairs $(Ax = b, x^*)$ such that x^* is a non-negative integral vector with $Ax^* = b$.)

Finally, an *optimization problem* Π is generally the following:

> Given a string $z \in \Sigma^*$ and a cost function $c : \Sigma^* \to \mathbb{R}_+$, find $y \in \Sigma^*$ such that $(z, y) \in \Pi$ and $f(y)$ is minimal (or maximal), or decide that no solution exists.

An optimization problem has the associated decision and search version, in which we ask about existence of a y with $f(y)$ being less (or greater) than some prescribed value. Often this allows to exploit *binary search* in order to find an optimum solution. For instance, if we have a good algorithm to find a matching (pairwise disjoint set of edges) in a graph of size at least γ, or decide that no such matching exist, we can run this algorithm $\log(|V|/2)$ times to find a matching of maximum size. Indeed, we know that there is a matching of size at most $|V|/2$ and at least one (as any edge is a matching). Then we may ask the search algorithm for a matching of size at least $|V|/4$, and depending on its answer, update the lower or the upper bound of the interval, to which the size of a maximum matching is known to belong. Each time the length of that interval will decrease by a factor of two, and after $\log(|V|/2)$ iterations, we find a maximum-size matching.

3 Algorithms and running time

As we agreed in the beginning, we shall assume intuitive understanding of what is an algorithm. Thus, this is a finite set of instructions (with loops and branches possible); particularly, an algorithm is also a string in Σ^*. We say that an algorithm $\mathscr{A}$ *solves* a problem Π if for any input $z \in \Sigma^*$, it terminates after a finite number of steps and delivers a correct answer/solution. As $\mathscr{A}$ is a string itself, there are only countably many algorithms. On the other hand, the number of possible problems is uncountable that implies that some problems cannot be solved by any algorithm, i.e., *undecidable*. One of the most intriguing undecidable problems is testing if a Diophantine equation has a solution (Hilbert's tenth problem, resolved by Matiyasevich in 1970). Another popular example of an undecidable problem is the *halting problem*: Decide if a given Turing machine (or computer program) terminates on a given input.

The *running time* of an algorithm on input z can be understood as the number of "elementary operations" the algorithm performs on input z. Typically, these elementary operations include addition, subtraction, multiplication, division, and comparison of numbers. It is natural to expect that when the input grows, the algorithm needs more time to compute a solution. Therefore, it makes sense to define the *running time function* of an algorithm as a function $f : \mathbb{Z}_+ \to \mathbb{Z}_+$ with $f(n)$ being the largest running time over all inputs of size at most n. Efficiency of an algorithm can then be expressed in terms of running time functions.

4 Polynomial, pseudo-polynomial and strongly polynomial algorithms

An algorithm $\mathscr{A}$ is called *polynomial* if its running time function $f(n)$ is bounded by some polynomial in n. We say that a problem is *solvable in polynomial time* if there is a polynomial algorithm that solves the problem. The class of all decision problems solvable in polynomial time is denoted by P. We refer to polynomial algorithms as "efficient", and to problems solvable in polynomial time as "easy".

But which storage model do we choose? For example, consider algorithms that accepts as input a single integer z. If z is stored in unary encoding, i.e., $\text{size}(z) = 1 + |z|$, then an algorithm with running time $\Theta(|z|)$ is polynomial. On the other hand, if z is stored in binary encoding, such an algorithm is *not* polynomial: indeed, $|z|$ is exponential in $\text{size}(z) = 1 + \lceil \log(|z| + 1) \rceil$. Moreover, in the most strict model, where $\text{size}(z) = 1$, the term "polynomial algorithm" actually refers to algorithms that run in constant time (i.e., its running time is independent of z). In order to distinguish these cases, we define an algorithm $\mathscr{A}$ to be

- *pseudo-polynomial* if it is polynomial when the input is given in unary encoding,
- *(weakly) polynomial* if it is polynomial when the input is given in binary encoding,
- *strongly polynomial* if it is polynomial in the model with $\mathrm{size}(z) = 1$ for every integer z and also (weakly) polynomial.

For example, consider any algorithm $\mathscr{A}$ operating on an integral matrix $A = [a_{ij}] \in \mathbb{Z}^{m\times n}$ and let $\alpha := \max\{|a_{ij}| : i = 1,2,\dots,m,\ j = 1,2,\dots,n\}$. Then $\mathscr{A}$ is pseudo-polynomial if its running time function is bounded by some polynomial in m, n and α. It is polynomial if its running time function is bounded by some polynomial in m, n, and $\log\alpha$. Particularly, the size of any number occurring during the execution of the algorithm must be bounded by a polynomial in m, n and α as well. Lastly, $\mathscr{A}$ is strongly polynomial if its running time function is bounded by a polynomial in m and n (assuming that each elementary operation takes constant time) and the size of any number occurring during the execution is bounded by some polynomial in m, n and $\log\alpha$.

It is absolutely clear how we can argue that a problem is easy: we need to describe an appropriate algorithm and prove that it is polynomial and yields a correct answer for any input. But can we argue that a problem is “hard”?

5 NP-completeness

As defined in the previous section, the class P consists of problems solvable in polynomial time. Another, possibly bigger class is called NP and can be seen as the class of decision problems, for which an answer can be *verified*. In other words, finding an answer might be hard, but checking its correctness is easy, provided a suitable certificate. Often, such a certificate is just a solution to the associated search problem; for instance, given a system of linear equations $Ax = b$ and an integral non-negative vector x^*, we can check in polynomial time if $Ax^* = b$.

More formally, a decision problem $\Pi \subseteq \Sigma^*$ belongs to NP if there is a polynomially solvable decision problem $\Pi' \subseteq \Sigma^* \times \Sigma^*$ and a polynomial ϕ such that for each $z \in \Sigma^*$, $z \in \Pi$ if and only if there is a $y \in \Sigma^*$ such that $\mathrm{size}(y) \leqslant \phi(\mathrm{size}(z))$ and $(z, y) \in \Pi'$. Thus, z plays a rôle of a polynomial-size “proof”, or a “certificate”. It is clear that $\mathrm{P} \subseteq \mathrm{NP}$ and that any problem in NP can be solved in exponential time, as we can simply enumerate all possible certificates y. However, it is a big open question if $\mathrm{P} \neq \mathrm{NP}$. The latter is widely believed to be true but no proof is known.

The *complement* of a decision problem $\Pi \subseteq \Sigma^*$ is the decision problem $\Sigma^* \setminus \Pi$. The class of decision problems, whose complement is in NP is denoted by coNP. Thus, coNP consists of the decision problems Π, for which the fact that a string z is *not* in Π can be verified in polynomial time, provided a polynomial-size certificate. Again, $\mathrm{P} \subseteq \mathrm{coNP}$ is clear, and $\mathrm{P} \neq \mathrm{coNP}$ is widely believed but remains an open problem. Finally, it is believed that $\mathrm{NP} \neq \mathrm{coNP}$, but…an open problem.

However, if $\mathrm{P} \neq \mathrm{NP}$ really holds, then we already know many problems belonging to $\mathrm{NP} \setminus \mathrm{P}$. These are the hardest problems in NP. Intuitively, we say that a problem Π_1 is harder than a problem Π_2 if an efficient algorithm for Π_1 implies an efficient algorithm for Π_2. For example, we can use an algorithm to find a maximum-size *stable set* in a graph (i.e., a subset of vertices such that no two of them are connected by an edge) to find a maximum-size matching: Given a graph $G = (V, E)$, we construct a graph $G' = (V', E')$ with $V' = E$ and $(e_1, e_2) \in E'$ if and only if edges e_1 and e_2 are disjoint in E; then a stable set in G' yields a matching in G.

Formally, a decision problem $\Pi_1 \subseteq \Sigma^*$ is called *reducible* to a decision problem $\Pi \subseteq \Sigma^*$ if there is a polynomial algorithm $\mathscr{A}$ that for any string $z \in \Sigma^*$ returns a string $\mathscr{A}(z)$ such that $z \in \Pi_1$ if and only if $\mathscr{A}(z) \in \Pi_2$. Particularly, if Π_2 is solvable in polynomial time, then Π_1 is also solvable in polynomial time. A problem Π is called NP-*hard* if each problem in NP is reducible to Π. Finally, a problem Π is NP-*complete* if it is NP-hard and belongs to NP itself. Surprisingly or not, there are NP-complete problems and, in fact, there are many of them known. For instance, finding a stable set of size at least γ is NP-complete. A polynomial algorithm for any of NP-complete problems would imply P = NP, and if P $\neq$ NP, then none of these problems is solvable in polynomial time.

In fact, there are many other complexity classes and the notion of completeness naturally extends to all of them. One of research directions in algorithms and complexity theory is to distribute problems among these classes (and prove that a problem is complete in some of these classes). For instance, given a problem Π in NP, we would like to know if it is also in P or NP-complete. For the problems proved to be NP-complete, it is sometimes interesting to know what makes them hard; this brings us to the area of *parameterized complexity*. For example, we may consider stable set problem on graphs of bounded degree: we assume that the degree of every vertex in a graph is bounded by a constant, which does not belong to the input of the problem, and ask for a polynomial algorithm in this case. However, the stable set problem remains hard even with this assumption. Another example is *integer programming*: Given a system of linear inequalities $Ax \leqslant b$, is there an integral vector x^* such that $Ax^* \leqslant b$? This problem is also NP-complete, but if we *fix* the number of variables (i.e., agree that the number of variables is bounded by a constant, which does not belong to the input), then the problem can be solved in polynomial time. In other words, there is an algorithm for integer programming with running $O(f(n) \cdot \text{poly}(\text{size}(a_{ij})), m)$, where m is the number of inequalities and n is the number of rows, $f(n)$ might be an exponential function but $\text{poly}(\text{size}(a_{ij})), m)$ is a polynomial in the size of the entries of matrix A and the number of rows in A.

Almost all problems we shall consider in this class are related to integral solutions of a system of linear inequalities, NP-hard in general, but solvable in polynomial time if we fix the number of variables (or the number of inequalities). In the following section we sketch some of the basic facts related to this subject.