Analysis of Algorithms I: NP-Completeness

Xi Chen

Columbia University

So far we have been working on efficient algorithms for various computational problems. All the algorithms we have seen so far are so-called "polynomial-time algorithms": On input of size n, their worst-case running time is $O(n^k)$ for some constant k. However, there are many computational problems for which no polynomial-time algorithm is known so far. Many of them look similar to problems we know how to solve efficiently.

1. Shortest simple path vs Longest simple path
2. Fractional Knapsack problem vs 0-1 Knapsack problem
3. Linear Programming vs Integer Linear Programming
4. Graph 2-Colorability vs Graph 3-Colorability
5. Shortest tour that traverses all edges of a graph (Chinese Postman problem) vs Shortest tour that visits all vertices of a graph (Traveling Salesman problem)

In each pair, the first one has a polynomial-time algorithm, while the best algorithm for the second one so far takes exponential time. Do they also admit a polynomial-time algorithm? We don't know and we will see that this is exactly the NP vs P problem, because all of them are NP-complete problems (hardest in NP).

A quick question before we define P and NP: Are there problems we know for sure that do not admit a polynomial-time algorithm? Yes, e.g., the halting problem: Given a Turing machine M and an input string x, decide if $M(x)$ stops in a finite number of steps. It can be mathematically proved that the halting problem is so difficult that it is undecidable (no finite-time algorithm can solve it). Knowing a problem is NP-complete, however, is different as we will see. Given that we don't know the answer to the NP vs P question, the NP-completeness of a problem can only serve as evidence (though a quite strong one) that it may not admit a polynomial-time algorithm but is by no means a mathematical proof that it does not have a polynomial-time algorithm.

Why do we want to prove the NP-completeness of a problem?

1. Use a heuristic algorithm instead: an algorithm that has no worst-case guarantee but seems to run well in practice
2. Use an approximation algorithm: an algorithm that outputs an approximately optimal solution
3. Add more details / restrictions to your problem
4. Most importantly, use NP-completeness to convince your boss that it is not your fault to come up with an exponential-time algorithm. If you could solve it in polynomial time, then you would win the Turing award, collect the one-million-dollar prize from the Clay Mathematics Institute and quit.

Decision problem: A problem with a yes-or-no answer:

1. 3-SAT: Given a 3-CNF (conjunctive normal form with 3 variables per clause), is there an assignment that satisfies all the clauses? (We will define 3-CNFs formally later.)
2. Hamiltonian cycle: Given an undirected graph G, does it have a simple cycle that contains every vertex in G?
3. The subset-sum problem: Given a set $S = \{s_1, \dots, s_r\}$ of positive integers and an integer target $t > 0$, is there a subset $S' \subseteq S$ whose elements sum to exactly t?

Some notation: Let I denote an instance of a decision problem L. We usually use $|I|$ to denote the input size and call it n. We say I is a yes / no-instance if the correct answer is yes / no. An algorithm A solves a decision problem L if given any instance I of L:

$$A(I) = \begin{cases} 1 & \text{if } I \text{ is a yes-instance} \\ 0 & \text{if } I \text{ is a no-instance} \end{cases}$$

Many decision problems are derived from optimization problems: Instead of finding the optimal value / cost or an optimal solution, we ask is there a feasible solution with value $\geq$ a given number b; or is there a feasible solution with cost $\leq$ a given number b.

From optimization to decision. Examples:

1. Clique: Given an undirected graph $G = (V, E)$ and a positive integer b, does G have a clique of size at least b?
2. Vertex cover: Given an undirected graph $G = (V, E)$, $V' \subseteq V$ is a vertex cover if for every edge $(u, v) \in E$, at least one of u and v is in V'. Given $G = (V, E)$ and a positive integer b, does G have a vertex cover of size at most b?
3. Traveling sales man problem: Given an undirected graph with integer weights and an integer b, is there a tour (i.e., a simple cycle that visits every vertex) with cost at most b?

An optimization problem is at least as hard as the corresponding decision problem: For example, if we have an algorithm that computes the size of a maximum clique, denoted by opt, then we can compare opt with b and easily solve the decision problem. So if we have evidence that the decision problem is hard, e.g. NP-hard, then it implies that the optimization is also hard.

In “most” cases, it is also possible, though a little more difficult, to make the other way: Given an algorithm for the decision problem, we can make (a polynomial number of) calls to solve the original optimization problem. For example, if we have an algorithm for Clique the decision version, then we can do binary search on b to find the exact value of opt, the maximum clique size. (Start with $b = n/2$ and check if opt $\geq b$ or not. If opt $\geq b$, set $b = 3n/4$; otherwise set $b = n/4$, and repeat.) Question: Use an algorithm for Clique the decision version to “find” a maximum clique.

One annoying subtlety: Be careful when the problem involves numbers. Unless spelled out explicitly, we always assume that numbers in an input instance I are encoded in binary. The same assumption applies to the weights of a graph. For example, in the Subset-Sum problem, we are given a set of positive integers $s_1, \ldots, s_r, t$. So the input size is

$$n = \log s_1 + \log s_2 + \cdots + \log s_r + \log t$$

instead of $n' = s_1 + \cdots + s_r + t$. There is a clear exponential gap between n and n'. It is easy (try) to give a polynomial-time algorithm for Subset-Sum in n'. However, we are interested in the setting when the numbers are encoded in binary for which n is the input size instead of n'. Subset-Sum is indeed NP-complete.

Now we define P (stands for Polynomial time): P is a class of decision problems. A decision problem L belongs to P iff there is a polynomial-time algorithm for L: given any input I of size n, it runs in time $O(n^k)$ for some constant k. Most of the problems (their decision version) we have seen so far are members of P:

1. LCS: Given two strings X, Y and a positive integer b, is there a common subsequence of X and Y of length $\geq b$?
2. Maximum Flow: Given a weighted directed graph G and a number b, is there a flow of value $\geq b$?

Indeed most of the algorithms we have seen so far have running time $O(n^k)$ for some pretty small constant k like $1, 2$ or 3.

Now we define NP (Nondeterministic Polynomial time, where the word "nondeterministic" comes from nondeterministic Turing machines): Again, it is a class of decision problems. A decision problem L is in NP if there is a certifier algorithm A such that

1. $A(I, C)$ is a polynomial-time algorithm that takes two inputs: I is an input instance of L and $C \in \{0, 1\}^*$ is a binary string. (We use $|C|$ to denote the length of C.)
2. For any yes-instance I, there is a binary string C such that $|C| \leq |I|^k$ for some constant $k > 0$ and $A(I, C) = 1$. We call such a string C a certificate (or witness or proof) for I.
3. For any no-instance I, $A(I, C) = 0$ for any binary string C.

So $L \in \mathrm{NP}$ if for any yes-instance I of L, there is a "short" (i.e., polynomial in $|I|$) certificate. Given the certificate, verification can be done efficiently. Moreover, a no-instance I has no certificate. For example, here is a certifier A for the Hamiltonian-Cycle problem: Given $I = G$, an undirected graph, and $C \in \{0,1\}^*$, view C as the binary encoding of a sequence p of vertices in G and check if p is indeed a Hamiltonian cycle. Output 1 if it is; and output 0 otherwise (note that this also includes the case when C is not a valid encoding).

It is clear that A is a polynomial-time algorithm. Moreover, If $I = G$ is a yes-instance, then there is a short certificate C, i.e., the binary encoding of a Hamiltonian cycle in G, because

$$A(I, C) = 1$$

On the other hand, if $I = G$ is a no-instance, then G has no Hamiltonian cycle and thus, I has no certificate C:

$$A(I, C) = 0, \quad \text{for any } C \in \{0, 1\}^*$$

From this we know that Hamiltonian Cycle is a member of NP.

Most of the problems we have seen so far are members of NP:

1. Vertex Cover: Let $I = (G, b)$ be an instance. Here is a certifier: Consider C as the encoding of a subset of vertices V' of G. Output 1 if V' is a vertex cover and $|V'| \leq b$.
2. 0-1 Knapsack problem: Let I be an instance:

$$I = (v_1, \ldots, v_n, w_1, \ldots, w_n, B, b)$$

where v_i and w_i are the value and weight of item i and B is the total weight affordable. Here is a certifier: Consider C as the encoding of a subset of items $S \subseteq \{1, \ldots, n\}$. Output 1 if

$$\textstyle\sum_{i \in S} w_i \leq B \quad \text{and} \quad \sum_{i \in S} v_i \geq b$$

More examples:

1. 3-SAT: Here is a certifier: Consider C as a $\{0,1\}$-assignment of the variables. Output 1 if it satisfies every clause.
2. Graph 3-Colorability: Here is a certifier: Consider C as an assignment of a color $\in \{1,2,3\}$ to each vertex. Output 1 if adjacent vertices are assigned different colors.
3. Graph Isomorphism: Let G and H be two graphs, here is a certifier: Consider C as the encoding of a one-to-one correspondence f between vertices of G and H. Output 1 if

$$(u,v) \text{ is in } G \iff (f(u),f(v)) \text{ is in } H$$

Easy to see that for each of these problems, the certifier runs in polynomial time (checking is easy). Every yes-instance has a short certificate, and there is no certificate for any no-instance. Also it is easy to see that P $\subseteq$ NP. If L is a decision problem in P, then it has a polynomial-time algorithm, denoted by B. Using B, we have the following certifier for L: Given (I, C), ignore C and run B on I. Output 1 if B outputs 1; and output 0 otherwise. It is clear that this certifier is polynomial-time. It always outputs 1 if I is a yes-instance (so any string C, the empty string in particular, is a certificate for I); and it always outputs 0 if I is a no-instance.

Fundamental question: Is P $=$ NP? Is it always as easy to generate a proof as it is to check a proof that is given to us?

Now we define NP-complete problems. To this end, we need to introduce reductions between decision problems. Given two decision problems L and L', a polynomial-time reduction from L to L' is a polynomial-time algorithm F such that given any input instance I of L, $I' = F(I)$ is an input instance of L' such that

$$I \text{ is a yes-instance of } L \iff I' \text{ is a yes-instance of } L'$$

We write $L \leq_P L'$ if there is a polynomial-time reduction from L to L'. Basically F is a polynomial-time algorithm that maps yes-instances of L to yes-instances of L'; and no-instances of L to no-instances of L'. (Note that F in general is not a bijection.)

It is clear that reductions compose: $L \leq_P L'$ and $L' \leq_P L^*$ implies that $L \leq_P L^*$. We are interested in reductions between decision problems because of the following lemma:

Lemma

If $L \leq_P L'$ and $L' \in P$, then $L \in P$ as well.

Here is a proof: As $L \leq_P L'$, there is a polynomial-time reduction F from L to L'. As $L' \in P$, it has a polynomial-time algorithm B. Using L and B, we give the following algorithm for L. Given I:

1. run F on I to get an instance $I' = F(I)$ of L'
2. run B on I' and return $B(I') \in \{0, 1\}$

It is a polynomial-time (why?) algorithm for L and thus, $L \in P$.

Now we define NP-complete problems. A decision problem L is NP-hard if every decision problem L' in NP reduces to it:

$$L' \leq_P L, \quad \text{for any } L' \in \text{NP}$$

Moreover, a decision problem L is NP-complete if

$$L \text{ is in NP; and } L \text{ is NP-hard}$$

As reductions compose, we immediately get the following lemma:

Lemma

If L is NP-hard and $L \leq_P L'$, then L' is also NP-hard.

Here is the reason why we are interested in NP-complete problems, and consider them as the hardest problems in NP:

Lemma

If any NP-hard problem L is in P, then $P = NP$.

A Quick Proof: For every decision problem $L' \in$ NP, L being NP-hard implies that $L' \leq_P L$ and thus, $L \in P$ implies $L' \in$ P.

Corollary

$P = NP$ if and only if an NP-complete problem is in P.

So P = NP if and only if an NP-complete problem is in P. Or equivalently, P $\neq$ NP if and only if there is an NP-complete problem that does not admit a polynomial-time algorithm. If one believes that P $\neq$ NP, then proving a problem is NP-complete serves as very strong evidence that it has no polynomial-time algorithm. But first of all, are there NP-complete problems?

The first NP-complete problem: Circuit-SAT:

1. Input: A boolean circuit. It is a directed acyclic graph in which every vertex has in-degree ≤ 2. There is a unique vertex that has out-degree 0 and is called the output of the circuit. The vertices with in-degree 0 are called the inputs of the circuit. Every vertex that is not an input (including the output) is labelled as one of the three boolean gates: $\wedge$ (and) $\vee$ (or) $\neg$ (not). (See Figure 34.8 on Page 1072 for an example.)
2. Output: Yes if the circuit is satisfiable: there is an assignment of $\{0,1\}$ to each input such that the circuit outputs 1.

Theorem

Circuit-SAT is NP-complete.

It is easy to see that Circuit-SAT is in NP. Given a circuit I, here is a certifier: Consider C as the encoding of an assignment to the inputs. Propagate values through the circuit and compute the output of the circuit. Return 1 if the output of the circuit is 1.

It is much more challenging to prove that Circuit-SAT is NP-hard. For this purpose, we need to formally define algorithms as Turing machines. Basically, if $L \in \text{NP}$, then L has a polynomial-time certifier algorithm (Turing machine) A. Given any instance I of L, we can construct in polynomial-time a boolean circuit from I and A such that the circuit is satisfiable iff I has a short certificate C (and thus, is a yes-instance of L). This gives us a polynomial-time reduction from L to Circuit-SAT. We skip the details of the construction here (can be found in the textbook or other intro computational complexity books), since our main goal is to use Circuit-SAT to prove more NP-complete problems.

Let L^* denote a known NP-complete problem, e.g., Circuit-SAT. To prove that a new decision problem L is also NP-complete, we no longer need to show that every $L' \in$ NP reduces to L. Instead, we simply use the following two steps:

1. First of all, show that L is in NP; and
2. Second, give a polynomial-time reduction from L^* to L.

Prove by yourself that the NP-completeness of L^* then implies the NP-completeness of L. In the next class, we will follow this approach and prove the NP-completeness of problems like 3-SAT, Hamiltonian-Cycle, Vertex Cover ... all starting from Circuit-SAT.

By the end of the next class, we will have collected a bundle of NP-complete problems. (Indeed there is a whole book devoted to known NP-complete problems.) When a new problem arises and you want to show that it is NP-complete, an important choice is then which known NP-complete problem L^* to reduce from. Picking the right NP-complete problem to reduce from can potentially simplify the proof significantly, and is usually a tricky part of an NP-completeness proof, as we will see next time.