Chapter 5
NP-Completeness

- Almost all the algorithms we have studies so far are polynomial-time algorithms, i.e., the running time is $O(n^k)$ for some constant k.
- Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.
- There are a lot of problems which cannot be solved by polynomial time. A simple example is that a problem needs to print out 2^n data.

- There is an interesting class of problems, called the NP-complete problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called P $\neq$ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Examples

- Shortest vs. longest simple paths: We saw that even with negative edge weights, we can find shortest paths from a single source in a directed graph $G = (V, E)$ in $O(VE)$ time. However, merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

- Euler tour vs. hamiltonian cycle: An Euler tour of a connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it is allowed to visit each vertex more than once. We can find the edges of the Euler tour in $O(E)$ time. A hamiltonian cycle of a directed graph $G = (V, E)$ is a simple cycle that contains each vertex in V. Determining whether a directed graph has a hamiltonian cycle is NP-complete.

Informal descriptions of P, NP and NP-complete

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

The class NP consists of those problems that are "verifiable" in polynomial time: If we were somehow given a **certificate** of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

For example, in the hamiltoniancycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, \ldots, v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well.

It is easy to see that any problem in P is also in NP. The open question is whether or not P is a proper subset of NP.

A problem is in the class of NP-complete, if it is in NP and is as "hard" as any problem in NP.

In the meantime, we will state without proof that if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial time algorithm.

Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date, without anyone having discovered a polynomial time solution to any of them, it would be truly astounding if all of them could be solved in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability.

Many problems of interest are optimization problems, in which each feasible (i.e., legal) solution has an associated value, and we wish to find a feasible solution with the best value.

NP-completeness applies directly not to optimization problems, however, but to decision problems, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0").

We can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Example: SHORTEST-PATH problem is the single-pair shortest-path problem in an unweighted, undirected graph.

A decision problem related to SHORTEST-PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is "hard". That is because the decision problem is in a sense "easier", or at least "no harder".

In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

Showing problems to be NP-complete

One important method used to show problems to be NP-complete is reductions.

Let us consider a decision problem A, which we would like to solve in polynomial time.

We call the input to a particular problem an instance of that problem. For example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes".

We call such a procedure a polynomial-time reduction algorithm and it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B.

For NP-completeness, we cannot assume that there is absolutely no polynomial time algorithm for problem A. The proof methodology is similar, however, in that we prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

We define an abstract problem Q to be a binary relation on a set I of problem instances and a set S of problem solutions.

For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.

The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

The theory of NP-completeness restricts attention to decision problems: those having a yes/no solution.

In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$. For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier. If $i = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then PATH.$(i) = 1$ (yes) if a path from u to v has at most i edges, and PATH.$(i) = 0$ (no) otherwise.

An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings. For example, we are all familiar with encoding the natural numbers $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$ as the strings $\{0, 1, 10, 11, \ldots\}$. Using this encoding, $e(17) = 10001$.

Thus, a computer algorithm that solves some abstract decision problem actually takes an encoding of a problem instance as input.

We call a problem whose instance set is the set of binary strings a concrete problem.

Let's review some definitions from formal-language theory. An alphabet Σ is a finite set of symbols. A language L over Σ is any set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, \cdots\}$ is the language of binary representation of prime numbers.

We denote the empty string by ϵ, the empty language by $\emptyset$, and the language of all strings over Σ by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$ is the set of all binary strings.

We can perform a variety of operations on languages. Set-theoretic operations, such as union and intersection, follow directly from the set-theoretic definitions. We define the complement of L by $\overline{L} = \Sigma^* - L$. The concatenation $L_1 L_2$ of two languages L_1 and L_2 is the language $L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$. The closure or Kleene star of a language L is the language $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$, where L^k is the language obtained by concatenation L to itself k times.

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them. We say that an algorithm A accepts a string $x \in \{0,1\}$ if, given input x, the algorithm's output $A(x)$ is 1. The language accepted by an algorithm A is the set of strings
$L = \{x \in \{0,1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A rejects a string x if $A(x) = 0$.

A language L is decided by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is accepted in polynomial time by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

A language L is decided in polynomial time by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0,1\}$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.

The complexity class NP is the class of languages that can be verified by a polynomial- time algorithm. More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that $L = \{x \in \{0,1\}^* : \text{ there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$.

We say that algorithm A verifies language L in polynomial time.

We say that a language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$. We call the function f the reduction function, and a polynomial-time algorithm F that computes f is a reduction algorithm.

Now we can give a more formal definition.

A language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in$ NP, and
2. $L' \leq_P L$ for every $L' \in$ NP.

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-hard. We also define NPC to be the class of NP-complete languages.

Theorem If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

To explain the name of NP (nondeterministic polynomial), we need to introduce an operation " choose" for algorithms.

- `choose`(b): this operation chooses in a nondeterministic way a bit and assigns it to b.

When an algorithm A uses the `choose` primitive operation, then we say A is nondeterministic. We say that an algorithm A nondeterministically accept a string x if there exists a set of outcomes to the `choose` calls that A could make on input x such that A would ultimately output "yes". In other words, it is as if we consider all possible outcomes to `choose` and only select those that lead to acceptance if there is such a set of outcomes. Note that this is not the random choices.

Equivalently, we may use the following definition.

A **nondeterministic algorithm** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution (certificate) to the given instance I (but may be complete gibberish as well).

Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to instance I , the algorithm either returns no or is allowed not to halt at all.)

We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution.

A nondeterministic algorithm is said to be nondeterministic polynomial if the time efficiency of its verification stage (deterministic stage) is polynomial.

Now we can define the class of NP as the class of decision problems that can be solved by nondeterministic polynomial algorithms.

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a first NP-complete problem.

The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1.

The circuit-satisfiability problem

The boolean combinational elements that we use in the circuit-satisfiability problem are three basic logic gates: the NOT gate ($\neg$), the AND gate ($\wedge$), and the OR gate ($\vee$). A boolean combinational circuit consists of one or more boolean combinational elements interconnected by wires. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second.

Boolean combinational circuits contain no cycles. In other words, suppose we create a directed graph $G = (V, E)$ with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k; the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.

A truth assignment for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is satisfiable if it has a satisfying assignment: a truth assignment that causes the output of the circuit to be 1.

The circuit-satisfiability problem is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this question formally, however, we must agree on a standard encoding for circuits. The size of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graph like encoding that maps any given circuit C into a binary string $\langle C \rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

CIRCUIT-SAT $= \{\langle C \rangle : C$ is a satisfiable boolean combinational circuit$\}$.

The main idea to prove that the CIRCUIT-SAT is NP-complete is that for any language L in NP, we can provide some polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT. The details of the proof is omitted here.

The vertex-cover problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V$ or $v \in V$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The size of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to determine whether a graph has a vertex cover of a given size k. As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\}.$$

Approximation algorithms

Many useful problems are NP-complete, for which we are not able to find polynomial solutions. In practice, we may try to find near-optimal solutions in polynomial time. We call an algorithm that returns near-optimal solutions as approximation algorithm.

We say that an algorithm for a problem has an approximation ratio of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of ρ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \quad \leq \quad \rho(n). \tag{1}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algorithm. The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply to both minimization and maximization problems.

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover. This problem is the optimization version of an NP-complete decision problem.

The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

```
1:  procedure APPROX-VERTEX-COVER(G))
2:      C = ∅
3:      E' = G.E
4:      while E ≠ ∅ do
5:          let (u, v) be an arbitrary edge of E'
6:          C = C ∪ {u, v}
7:          remove from E' every edge incident on either u and v
8:      end while
9:      return C
10: end procedure
```

Theorem APPROX-VERTEX-COVER is a polynomial-time 2 - approximation algorithm.

Proof. It is easy to see that the running time for the procedure is $O(E + V)$.

The set C of vertices that is returned by the procedure is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 5 of the procedure picked. In order to cover the edges in A, any vertex cover, in particular, an optimal cover C^*, must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 5, all other edges that are incident on its endpoints are deleted from E' in line 7.

Therefore, no two edges in A are covered by the same vertex from C^*, and we have the lower bound

$$|C^*| \geq |A|$$

on the size of an optimal vertex cover. Each execution of line 5 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2|A|.$$

Combining above equations, we obtain

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*|, \end{aligned}$$

thereby proving the theorem. □