NP-Completeness

"mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true"

Bertrand Russell

- Is there a formal unifying theory of algorithms and languages?
- Is there an efficient algorithm for any problem?
- P=NP?

A Hard Problem: Hamiltonian Path

Given a directed graph G which has two special nodes called the source and the target, the problem is to determine if there is a path from the source to the target that visits every node in the graph exactly once. Let HAM-PATH be the set of triples (G, s, t) where G is a graph, s and t are nodes of G, and there is a path from s to t in G that visits every node in G exactly once.

$\star$

⋆ A simple solution: enumeration.

Another Hard Problem: Traveling Salesman (TSP)

Given n cities, and the cost c_{ij} of getting from city i to city j for $1 \leq i, j \leq n$, the problem is to find the cheapest way to visit every city exactly once and return to the starting city. Such a cycle is called a *tour*. This problem is also exponential.

So far in this course, we have focused on finding efficient algorithms for computational problems. It is clearly desirable to be able to identify which problems do have efficient algorithms and which don't. The **theory of NP-completeness** addresses this need.

Decision Problems

An abstract **problem** Q is a mapping from a set of problem **instances** I to a set of problem **solutions** S.

⋆

$Q =$ SHORTEST-PATH

To keep things simple, in our description of NP-completeness we focus on decision problems, that is, problems with a yes/no answer. Restricting attention to decision problems is not a serious restriction. Non-decision problems can be rephrased as decision problems.

⋆

$Q =$ PATH

Given G, u, v, k, is there a path from u to v of cost $\leq k$?

⋆

$Q =$ TSP

Given n cities, the cost c_{ij} of getting from city i to city j for $1 \leq i, j \leq n$, and a number k, is there a tour of the cities of cost at most k?

Another simplifying assumption is that the input to all decision problems is in binary. We place angle brackets around the input to denote the input written in binary. For example, $< G, u, v, k >$ denotes the binary encoding of the graph G with vertices u, v and number k.

Formal-Language Framework

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. The language of all strings over Σ is denoted Σ^*.

⋆ Example:

A decision problem Q is entirely characterized by those problem instances that produce a 1 (yes) answer. We can, therefore, view Q as a Language L over $\Sigma = \{0, 1\}$:

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

⋆ Examples:

An decision algorithm A **accepts** an input binary string x if its output is $A(x) = 1$. We associate with a decision algorithm A the set of binary strings on which A outputs "yes" and we call this the *language accepted by* A.

A Language L is accepted in polynomial time by an algorithm A if it is accepted by A and there is a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

⋆ PATH

Complexity Classes

The polynomial (P) complexity class is defined as follows:

$$P = \left\{ L \;\middle|\; \begin{array}{l} \text{there is an algorithm } A \text{ that accepts } L \\ \text{(and only } L\text{) and runs in polynomial time} \end{array} \right\}$$

The class of languages in P are exactly the decision problems that have efficient algorithms.

Now we turn to problems that are **not known to be in P**. It turns out that several problems that are not known to be in P have the following property: Given a "yes" instance x of the problem, there is a "short" (polynomial length) **certificate** (or witness) to the fact that x is a "yes" instance. No such witness may exist for "no" instances (notice the asymmetry).

The process of checking if a potential certificate does indeed show that the instance x is a "yes"-instance to a given problem is referred to as a **verification algorithm**. Suppose someone wants to convince you that the graph G is in the language HAM-PATH. The person could give you a list of n nodes starting at the source s and ending at the target t. Given the list, you could check whether each pair of nodes in the list is connected by an edge and whether each node of the graph appears exactly once in the list. If so, you would be convinced that $< G, s, t >$ is in HAM-PATH. You are acting as a "verifier" for the HAM-PATH problem.

⋆

If $< G, s, t >$ is NOT in HAM-PATH, your algorithm will not accept no matter what list of nodes c is presented to you.

More generally, a verification algorithm V for a language L has two inputs: a binary string x and also a string y. V has the property that if x is in L, then there is some string y such that V accepts (x, y), but if x is not in L, then for all strings y, V does not accept (x, y). We are interested in verification algorithms that run in time that is bounded

by a polynomial in the length of x. Thus, without loss of generality, we can assume that the length of the "witness" string y is also bounded by a polynomial.

⋆ Verification example: TSP

Definition: A **verification algorithm** V takes two binary inputs: an instance (e.g. graph and k, etc) of a problem and a certificate. The language that is verified by V is

$$\mathrm{L} = \left\{ x \;\middle|\; \begin{array}{l} \text{there exists } y \text{ such that } V \\ \text{outputs ``yes'' on input } (x, y)\} \end{array} \right\}$$

Definition:

$$\mathrm{NP} = \left\{ L \;\middle|\; \begin{array}{l} \text{there is a polynomial time verification algorithm} \\ \text{for } L \text{ that takes as input an instance } x \text{ of } L \\ \text{and a string } y \text{ of length polynomial in } L. \end{array} \right\}$$

The name NP means "Nondeterministic Polynomial Time". Note that P is a subset of NP. To see this, let L be a decision problem in P, (e.g. L could be the variation of the shortest path problem in which the goal is to determine whether there is a path of cost $\leq k$ from a given source to a given target). If A is a polynomial time algorithm for L, there is some other algorithm A' that gets two inputs: an instance x of L and a witness y. A' ignores y and runs A on x.

Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly.
Not all problems are in NP. Examples of a problems not known to be in NP are "game-like" problems, such as checkers (played on an $n \times n$ board. Given a particular configuration of the game, does one player have a winning strategy in the game? A strategy describes what move a player will take given any configuration of the game. Since there are exponentially many configurations, exponential time is required even to write down a winning strategy.

Justification of "efficient = polynomial time"

Why is this a notion of "efficient" a good one? First, algorithms with exponential running times don't scale well and therefore should be not be considered efficient. To illustrate this, suppose that each step of an algorithm runs in one tenth of a microsecond. The following algorithm compares, for different values of n, the total time needed by an algorithm that takes $n, n^2, n^3, 2^n$, or 3^n steps.

	10	20	30	40	50	60
n	.000001 second	.000002 second	.000003 second	.000004 second	.000005 second	.000006 second
n^2	.00001 second	.00004 second	.00009 second	.000016 second	.00025 second	.00036 second
n^3	.0001 second	.0008 second	.0027 second	.0064 second	.0125 second	.0216 second
2^n	.0001 second	.1 second	1.8 minutes	1.2 days	3.5 years	36.6 centuries
3^n	.0059 second	5.8 minutes	.6 year	385 centuries	2x10^7 centuries	1.3x10^12 centuries

It is also true, however, that an algorithm with running time which is a large polynomial (n^{10} for example), is not going to scale well either. Why should such algorithms be considered efficient? One reason is that we want our theory of efficient algorithms to be model-independent, i.e. the insights gained from the theory should hold for any reasonable "model" of what an algorithm is, (reasonable models of algorithms include your favorite programming languages, parallel computer models, and so on). The polynomial-time definition of "efficient" ensures that this is the case.

Another justification for the definition is that it turns out the bulk of the problems in P that are of practical interest have running times that are bounded by a small polynomial. Most of the problems that we have seen in this class, such as minimum spanning tree, shortest paths, arithmetic operations, and data compression, can be formulated as decision problems that have running time that is $O(n^3)$. Therefore, a justification for our notion of P is that it seems to capture the "right" problems while excluding a significant class of problems that have exponential running time.