NP-completeness

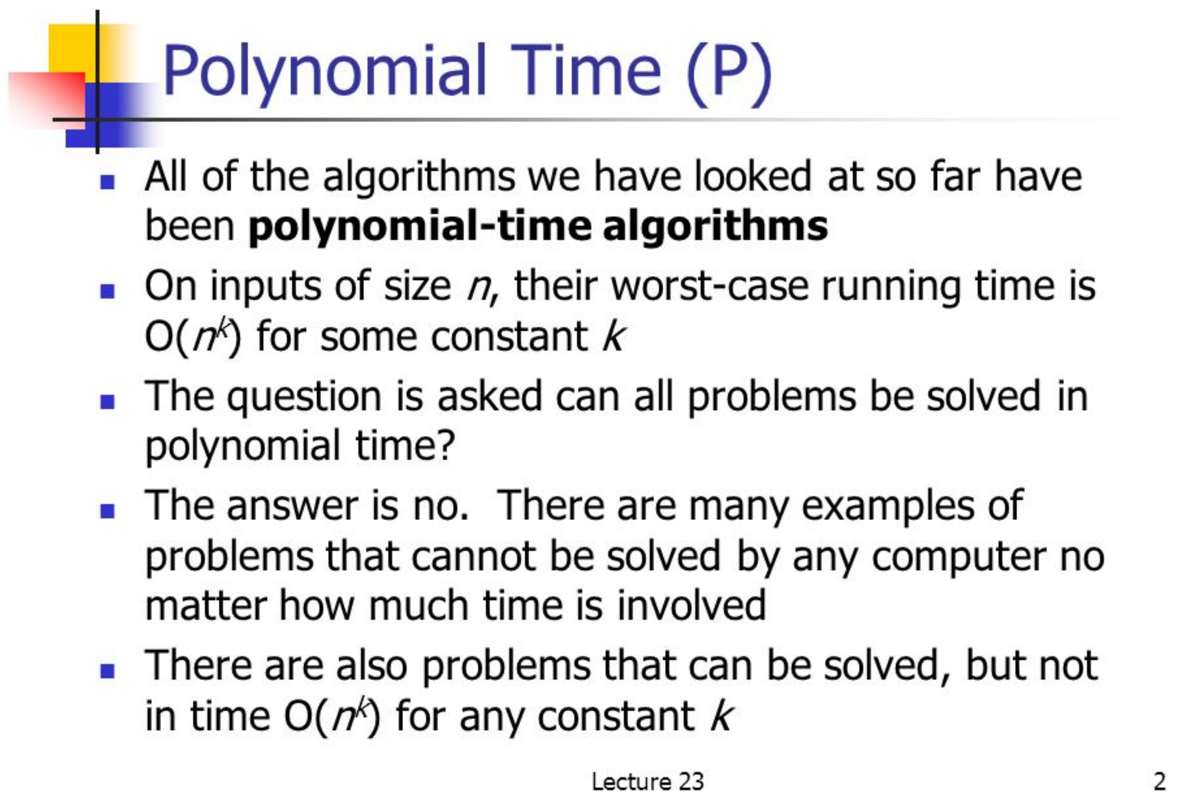
Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \leq_P L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "less than or equal to" notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

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.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.



Polynomial Time (P)

- All of the algorithms we have looked at so far have been **polynomial-time algorithms**
- On inputs of size n, their worst-case running time is $O(n^k)$ for some constant k
- The question is asked can all problems be solved in polynomial time?
- The answer is no. There are many examples of problems that cannot be solved by any computer no matter how much time is involved
- There are also problems that can be solved, but not in time $O(n^k)$ for any constant k

Lecture 23 2

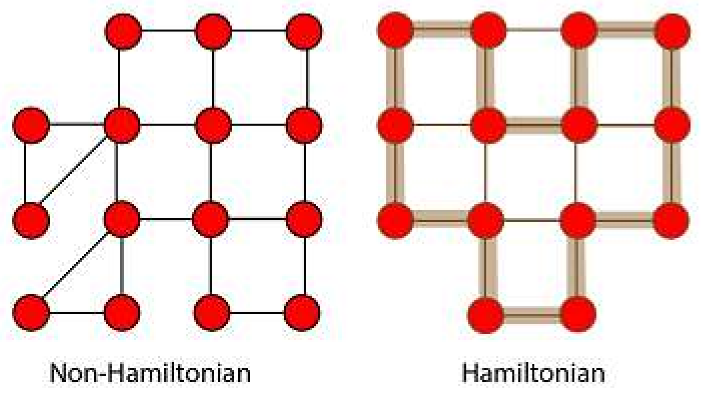
Polynomial Time Verification

Before talking about the class of NP-complete problems, it is essential to introduce the notion of a verification algorithm.

Many problems are hard to solve, but they have the property that it easy to authenticate the solution if one is provided.

Hamiltonian cycle problem:-

Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.



Let us understand that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would similarly say: "the period is hv3, v7, v1....v13i.

We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle.

NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that P $\neq$ NP is the existence of the class of "NP-complete" problems. This class has the surprising property that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time solution, that is, P = NP. Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time. In fact, if NP - P should turn out to be nonempty, we could say with certainty that HAM-CYCLE $\in$ NP - P.

The NP-complete languages are, in a sense, the "hardest" languages in NP. In this section, we shall show how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility." Then we formally define the NP-complete languages, and we finish by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. In Sections 34.4 and 34.5, we shall use the notion of reducibility to show that many other problems are NP-complete.

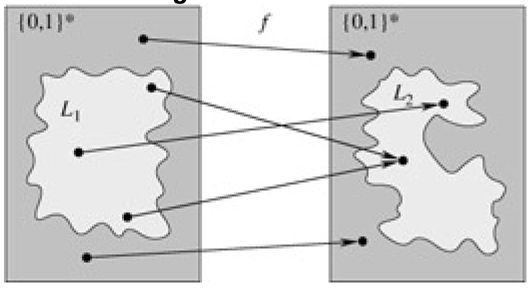
Reducibility

Intuitively, a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q. For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, if a problem Q reduces to another problem Q', then Q is, in a sense, "no harder to solve" than Q'.

Returning to our formal-language framework for decision problems, 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 $f : \{0, 1\}^* \rightarrow \{0,1\}^*$ such that for all x $\{0, 1\}^*$,

(34.1) $x \in L_1$ 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 called a ***reduction algorithm***.



Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

NP-Completeness Proofs¶

28.12.1. NP-Completeness Proofs

To start the process of being able to prove problems are NP-complete, we need to prove just one problem HH is NP-complete. After that, to show that any problem XX is NP-hard, we just need to reduce HH to XX. When doing NP-completeness proofs, it is very important not to get this reduction backwards! If we reduce candidate problem XX to known hard problem HH, this means that we use HH as a step to solving XX. All that means is that we have found a (known) hard way to solve XX. However, when we reduce known hard problem HH to candidate problem XX, that means we are using XX as a step to solve

So a crucial first step to getting this whole theory off the ground is finding one problem that is NP-hard. The first proof that a problem is NP-hard (and because it is in NP, therefore NP-complete) was done by Stephen Cook. For this feat, Cook won the first Turing award, which is the closest Computer Science equivalent to the Nobel Prize. The “grand-daddy” NP-complete problem that Cook used is called SATISFIABILITY (or SAT for short).

A **Boolean expression** is comprised of **Boolean variables** combined using the operators AND ($\cdot$·), OR ($+$+), and NOT (to negate Boolean variable xx we write $\overline{x}$x¯). A **literal** is a Boolean variable or its negation. A **clause** is one or more literals OR’ed together. Let EE be a Boolean expression over variables $x_1,x_2,\ldots,x_n$x1,x2,...,xn. Then we define **Conjunctive Normal Form** (CNF) to be a Boolean expression written as a series of clauses that are AND’ed together. For example,

$E=(x_5+x_7+\overline{x_8}+x_{10})\cdot(\overline{x_2}+x_3)\cdot(x_1+\overline{x_3}+x_6)$E=(x5+x7+x8¯+x10)·(x2¯+x3)·(x1+x3¯+x6)

NP-complete problem

NP-complete problem, any of a class of computational problems for which no efficient solution algorithm has been found. Many significant computer-science problems belong to this class—e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.

So-called easy, or tractable, problems can be solved by computer algorithms that run in polynomial time; i.e., for a problem of size *n*, the time or number of steps needed to find the solution is a polynomial function of *n*. Algorithms for solving hard, or intractable, problems, on the other hand, require times that are exponential functions of the problem size *n*. Polynomial-time algorithms are considered to be efficient, while exponential-time algorithms are considered inefficient, because the execution times of the latter grow much more rapidly as the problem size increases.

A problem is called NP (nondeterministic polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete. Thus, finding an efficient algorithm for any NP-complete problem implies that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be recast into any other member of the class. It is not known whether any polynomial-time algorithms will ever be found for NP-complete problems, and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical computer science. When an NP-complete problem must be solved, one approach is to use a polynomial algorithm to approximate the solution; the answer thus obtained will not necessarily be optimal but will be reasonably close.