P, NP AND NP-COMPLETE PROBLEMS

Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

There are several **reasons for intractability**.

- **First**, we **cannot solve** arbitrary instances of intractable problems in a reasonable amount of time unless such **instances are verysmall**.
- **Second**, although there might be a huge difference between the running times in *O(p(n))* for polynomials of **drastically different degrees**. where p(n) is a polynomial of the problem's input sizen.
- **Third**, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are **always polynomialstoo**.
- **Fourth**, the choice of this class has led to a development of an extensive theory called ***computational complexity***.

Definition: Class *P* is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial class***.

- Problems that can be solved in polynomial time as the set that computer science theoreticians call **P**. A more formal definition includes in P only **decision problems**, which are problems with **yes/no**answers.
- TheclassofdecisionproblemsthataresolvableinO(p(n))**polynomialtime**,where$p$(n)is a polynomial of problem's input size n

  Examples:
  - Searching
  - Elementuniqueness
  - Graph connectivity
  - Graph acyclicity
  - Primality testing (finally proved in2002)
- **The restriction of P** to decision problems can be justified by the followingreasons.
  - First, it is sensible to **exclude problems not solvable in polynomial time** because of their exponentially large output. e.g., generating subsets of a given set or all the permutations of n distinctitems.
  - Second, **many important problems that are not decision problems** in

their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color. Coloring of the graph's vertices with no more than m colors for m = 1, 2, (The latter is called the **m-coloringproblem**.)

- So, every decision problem can not be solved in polynomial time. Some **decision** problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm (**Halting problem**).

- **Non polynomial-time algorithm:** There are many important problems, however, for which no polynomial-time algorithm has been found.
    - ***Hamiltonian circuit problem*****:** Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
    - ***Traveling salesman problem*****:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer r weights).
    - ***Knapsack problem*****:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
    - ***Partition problem*****:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
    - ***Bin-packing problem*****:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size1.
    - ***Graph-coloring problem*****:** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.
    - ***Integer linear programming problem*****:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

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.

1. **Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the giveninstance.
2. **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 atall.)

Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Definition: Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Most decision problems are in NP. First of all, this class includes all the problems in P:

$$\mathbf{P \subseteq NP}$$

This is true because, if a problem is in P, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage. But NP also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in NP.

Note that P = NP would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

Definition: A decision problem *D*1 is said to be ***polynomially reducible*** to a decision problem *D*2, if there exists a function *t* that transforms instances of *D*1 to instances of *D*2 such that:

1. *t* maps all yes instances of *D*1 to yes instances of *D*2 and all no instances of *D*1 to no instances of*D*2.
2. *t* is computable by a polynomial time algorithm.

This definition immediately implies that if a problem D1 is polynomially reducible to some problemD2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time

Definition: A decision problem *D* is said to be ***NP-complete*** if it is hard as any problem in NP.

1. It belongs to class *NP*
2. Every problem in *NP* is polynomially reducible to*D*

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

Theorem: A decision problem is said to be *NP-complete* if it is hard as any problem in NP.

Proof: Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G′ representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound *m* on the Hamiltonian circuit length, we take *m* = *n*, where *n* is the number of vertices in G (and G′). Obviously, this transformation can be done in polynomialtime.

Let G be a yes instance of the Hamiltonian circuit problem. Then G has a Hamiltonian circuit, and its image in G′ will have length n, making the image a yes instance of the decision traveling salesman problem.

Conversely, if we have a Hamiltonian circuit of the length not larger than n in G′, then its length must be exactly n and hence the circuit must be made up of edges present in G, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuitproblem.

This completes the proof.

Theorem: State and prove Cook's theorem.

Prove that CNF-sat is *NP*-complete.

Satisfiability of boolean formula for three conjuctive normal form is NP-Complete.

NP problems obtained by polynomial-time reductions from a *NP*-complete problem **Proof:** The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP,* both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found.

Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union. In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*complete.

$_1$	$_2$	$_3$	$\overline{\ }_1$	$\overline{\ }_2$	$\overline{\ }_3$	$_1\overline{V}_2\overline{V}_3$	$\overline{\ }_1V_2$	$\overline{\ }_1\overline{V}_2\overline{V}_3$	$(_1\overline{V}_2\overline{V}_3)\blacktriangle(\overline{\ }_1V_2)\blacktriangle(\overline{\ }_1\overline{V}_2\overline{V}_3)$
T	T	T	F	F	F	T	T	F	F
T	**T**	**F**	F	F	T	T	T	T	**T**
T	F	T	F	T	F	T	F	T	F
T	F	F	F	T	T	T	F	T	F
F	T	T	T	F	F	F	T	T	F
F	T	F	T	F	T	T	T	T	T
F	F	T	T	T	F	T	T	T	T
F	F	F	T	T	T	T	T	T	T

The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted$\overline{\ }_1$, $\overline{\ }_2$, and$\overline{\ }_3$respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \& (\bar{x}_1 \vee x_2) \& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

The CNF-satisfiability problem asks whether or not one can assign values *true* and *false* to variables of a given boolean expression in its CNF form to make the entire expression *true*. (It is easy to see that this can be done for the above formula: if x_1 = *true*, x_2 = *true*, and x_3 = *false*, the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well- known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if *P* != *NP* there must exist *NP* problems that neither are in *P* nor are *NP*-complete.

Showing that a decision problem is *NP*-complete can be done in two steps.

1. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.

2. The second step is to show that every problem in NP is reducible to the problem in question in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial ime.

The definition of *NP*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *NP*-complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence *P* = *NP*.