9. Theory of NP

9.1 Introduction

Since the 1960s, theoretical computer scientists (notably, Stephen Cook and Richard Karp, among others) have observed a dichotomy among problems that are solvable by digital computers. The dichotomy has to do with the algorithm's efficiency as measured by the running time, $T(n)$, as a function of the input size n. [Note: To be precise, "input size" means "the number of bits needed to encode the input".) For example, we have been talking about sorting as $O(n \log n)$ where n is the number of keys. If we assume each key is encoded using k bits, then the input size $S = kn$ bits. Thus, the running time in terms of S is, by substituting $n=S/k$, $O(S/k \log S/k)=O(S \log S)$.] Ideally, for any given problem we would like to find a *polynomial-time algorithm* where the running time is bounded by $O(n^c)$ for some constant c. Unfortunately, for many practical problems, like graph coloring, no one has found such algorithms and thus we had to settle for *exponential-time algorithms* that run in $\Omega(c^n)$ for some constant c. Generally, we describe problems of the first type as *easy* (*tractable*) *problems* and of the second type as *hard* (*intractable*) *problems*. Many of the intractable problems are *decision* problems where the answer (output) that is sought is either yes or no, and quite often, a change of a single word changes an easy problem into a hard one.

Consider the following graph coloring problems, where the objective is to color the vertices of the graph such that no two adjacent vertices are assigned the same color:

2-*Colorabily*: Given an undirected graph $G=(V,E)$, is G colorable using two colors?
3-*Colorabily*: Given an undirected graph $G=(V,E)$, is G colorable using three colors?

For either problem let $n=|V|$. For the first problem, we recognize that a graph is 2-colorable if and only if it is bipartite. In Section 7.4, we have seen an algorithm that determines this and runs in $O(|V|+|E|)=O(n^2)$. However, for the second problem no one has found a polynomial-time algorithm and it seems that the only way to answer the problem is by brute-force (or backtrack) search which runs in $\Omega(3^n)$.

Circuit Satisfiability

Circuit Satisfiability (or, *Satisfiability* for short) is another example of a problem that we don't know how to solve in polynomial time. In this problem, the input is a Boolean circuit: a collection of *and* ($\wedge$) , *or* ($\vee$), and *not* ($\neg$) gates connected by wires. We will assume that there are no loops in the circuit (no delay lines or flip-flops). The input to the circuit is a set of n Boolean (*true*/*false* or 1/0) values $x_1,\dots, x_n$. The output is a single Boolean value. Any such Boolean circuit can be represented as a mathematical expression. Figure 9.1 shows an instance of the satisfiability problem. The problem can be modeled as a directed graph where each logic circuit is a node with incoming directed edges representing the input lines. Given specific input values, we can calculate the output in polynomial time by processing the gates in a topological-sorting order.

The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output true, or conversely, whether the circuit always outputs false. Nobody knows how to solve this problem faster than just trying all 2^n possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has ever proved that this is the best we can do; perhaps there is a clever algorithm that noone has discovered yet!

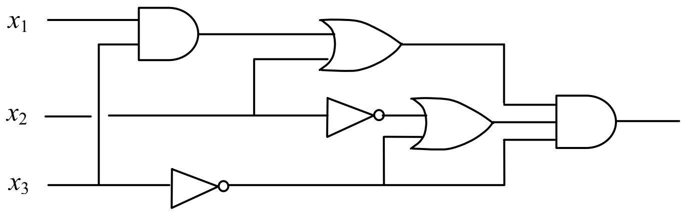

Figure 9.1 An instance of the circuit satisfiability problem. This instance is for n=3 variables, and is equivalently defined by the Boolean formula: $((x_1 \wedge x_3) \vee x_2) \wedge (\neg x_2 \vee \neg\, x_3) \wedge (\neg x_3)$. The vector $< x_1, x_2, x_3> = <1,1,0>$ is a certificate for this instance.

A Peculiar Property: Easily-Verifiable Certificate

The circuit satisfiability problem on n variables (input lines) essentially asks if there is a binary vector of length n that makes the circuit evaluates to true. For example, for the instance specified by Figure 9.1, the answer is "Yes, using the vector $<x_1, x_2, x_3> = <1,1,0>$". In general, if for a given instance of the problem (i.e., a circuit), some one claims that the answer is yes, and, at the same time, presents us with a "solution vector" then his claim can be checked easily (in polynomial time), by simply evaluating the circuit using the given 0/1-assignment. Such a solution that makes the circuit evaluates to true is a "certificate". In general, for a decision problem, a *certificate* is the information needed to verify a yes-answer. Thus, if the circuit is satisfiable, then the verification of the claim can be performed easily once a certificate is given. It seems that the hard part is finding a certificate.

Here is another example of a decision problem and a certificate. Consider the Subset-Sum problem: Given a set of n integers $A = \{a_1, a_2, \ldots, a_n\}$, is there a nonempty subset of A whose sum of elements is zero? For instance, is there a nonempty subset of the set $A = \{-2, -3, 8, 15, -10, 11\}$ whose sum of elements is zero? If someone claims that the answer is "Yes, because $\{-2, -3, -10, 15\}$ has a zero sum", then we can quickly check the claim with a few additions. Verifying that a subset adds up to zero is much faster than finding the subset in the first place. Here we see that while brute-force search for a certificate is $\Omega(2^n)$ time, the process of certificate verifications is $O(n)$ time.

The satisfiability and the subset-sum problems exhibit a characteristic that is common to many (thousands) intractable problems:

> The problem asks if there is a vector (of length n or less) whose elements come from a finite set, satisfying certain constrains. A yes-vector provides an easily-verifiable (i.e., polynomial time in n) certificate. The answer can be determined by brute-force search (generate and test all vectors), and since the number of vectors is exponential in n, this approach leads to an exponential-time algorithm.

The hardest part to an intractable problem is finding a certificate, whereas certificate verification is easy. How do we find a certificate in the first place is the real question? Doing brute-force search on the computers we use today takes exponential-time. If we have a very powerful computer that can generate and test all the vectors instantly then that would be great! We can just imagine that such a computer, when it presents us with the right certificate, as having made an *intelligent guess* (after-all people do not spend too much time when they are asked to guess). Even if such a powerful computer does not exist, we have a name for it, it is a *non-deterministic machine*.

Non-Determinism: A Deterministic Machine versus a Non-Deterministic Machine

A *deterministic machine*, like computers we use today, is capable of enumerating all binary vectors of length *n* (2^n in all) one at a time as in:

```
for i = 1 to 2^n
{ int[] V←GetVector(i);
   Do something with V; }
```

The running time of the preceding loop will remain exponential even if the body of the loop takes polynomial time.

A *non-deterministic machine* is a hypothetical computer more powerful than a computer we use today. It is able to enumerate the binary vectors of length *n* all at once using a powerful *Choose* instruction (the instruction takes as input the length of the vector, a finite set from which the elements are to be chosen, and a program-code block that is to be executed for every possible vector). For a non-deterministic machine, we can rewrite the preceding loop into the following (this is an example of a *non-deterministic algorithm*):

```
Choose(n, {0,1}) { Do something with V; }
```

One way to think of this is that upon encountering a Choose instruction, the machine instantly clones itself into a number of (deterministic) machines all running in parallel, each machine is associated with one of the possible vectors. The closest thing to a Choose-instruction is the *fork* statement in the *C*-language. However, whereas the fork statement can only be used to clone a process into a small (and fixed) number of (child) processes — it does not help to create more processes than the available number of CPUs, the Choose instruction calls for an exponential (in *n*) number of processes and CPUs.

In a non-deterministic algorithm, we can execute a call to Choose and continue on executing the rest of the program code. This calls for additional feature of the Choose instruction that enables it to return a vector as in:

```
int[] V←Choose(n, {0,1}) { if V satisfies the problem’s constraints return V; }
```

The semantics of the preceding statement is that it returns the moment a vector *V*, satisfying the problem’s constraints, is found (as if aborting all child processes).

The running time for the preceding non-deterministic algorithm will be dominated by the test that a vector *V* stratifies the problem constrains. Clearly, intractable problems for which certificate verification is doable (deterministically) in polynomial time will end up solvable in polynomial-time using a non-deterministic algorithm. For this reason we classify such problems as belonging to the class *NP* (*N*on-deterministic *P*olynomial). At the same time, for a problem to be in NP, it suffices to show that a verification of a "correct" guess (i.e., a certificate) can be done in deterministic polynomial time.

Variants of the Satisfiability Problem

For many of the intractable problems, the real issue is like this: which of the allowable values we can assign to a problem variable and be certain of *quickly* reaching a solution. That is what makes these problems hard. In particular, the reason that the satisfiability problem is hard is not that the parsing of the input formula is complex or evaluating it takes long time; rather, it is of not being certain as whether x_i should be set to 1 or 0. To emphasize this point, we define two variants of the satisfiability problem (*CNF-SAT*, *3-SAT*) that use much simpler syntax; yet these variants remain as intractable as the original satisfiability problem.

Conjunctive Normal Form

A Boolean formula (expression) is in *Conjunctive Normal Form* (CNF) if it is an *ANDing* (i.e., a conjunction) of clauses, where each clause is an *ORing* of literals (a literal is a variable x_i or its negation $\neg x_i$). The following is an example of a CNF expression:

$$(x_1 \vee x_2 \vee \neg x_5) \wedge (\neg x_2 \vee x_4 \vee x_3) \wedge (\neg x_2 \vee x_5).$$

A CNF formula allows for a simple form of input. For example, the preceding formula can be input as:

1,2,–5; –2,4,3; –2,5

Likewise, the evaluation of the formula is straightforward, where we evaluate the clauses in order and terminate as soon as we encounter a clause that evaluates to false.

CNF-SAT Problem

Input: A Boolean formula in CNF.
Question: Is there an assignment of Boolean values to its variables so that the formula evaluates to true (i.e., is the formula satisfiable)?

3-SAT Problem is CNF-SAT in which each clause has exactly three literals.

Fact: CNF-SAT and 3-SAT are NP-complete while 2-SAT is in *P*, i.e., there is a polynomial time algorithm when every clause has exactly two literals.

Decision versus Optimization

A *decision problem* is a computational problem for which the answer (output) is either *Yes* or *No*. Normally, for a yes-answer a certificate that is easily verifiable should be given as part of the output.

In a contrast to a decision problem, an *optimization problem* is one where we try to minimize or maximize some quantity that is specified as an objective function.

Let us assume that for the optimization problems under study, the objective function being minimized or maximized evaluates to positive integers. An optimization problem can be converted into a corresponding decision problem by adding a positive integer parameter k as an input parameter, and then asking whether there is a solution for which the value of the function is equal to k (or *at most* k for minimization or *at least* k for maximization).

For example, the decision version corresponding to minimization of colors needed for (vertex) coloring of an input graph G is to ask whether G is colorable using *at most* k colors, where k is a positive integer that is part of the input.

If a decision problem is hard, then its related optimization version must also be hard. In general, if we have an algorithm to solve the decision problem then we can use it to solve the optimization problem, by a process of *problem reduction*. The reduction process is simply to invoke the decision algorithm for varied k until the optimum value is found.

We can do this by having k increase (or decrease) toward the optimal solution. For illustration, consider encoding an algorithm for the graph coloring optimization problem in terms of the algorithm for the decision version, *IsColorable*(G,k).

K* decreases from *n (start from a solution whose value is more than (or equal to) the optimal solution):

```
int Color_Opt(G)
{  n = |V|;
   k = n; // Any graph on n vertices is n colorable
   // This loop stops at the largest k for which G is not k-colorable
   while IsColorable(G,k) { k = k-1; }
   return k+1;
}
```

***K* increases from 1** (start from a solution whose value is less than (or equal to) the optimal solution):

```
int Color_Opt(G)
{  k = 1; // Any graph on n vertices needs at least one color
   // This loop stops at the smallest k for which G is k-colorable
   while Not(IsColorable(G,k)) { k = k+1; }
   return k;
}
```

Let $T(n)$ be the running time for the decision algorithm *IsColorable*(). The preceding reduction clearly shows that the running time for the optimization algorithm is $O(nT(n))$. Thus, if the decision problem is solvable in polynomial time then so is the optimization problem. Therefore, from computational complexity standpoint, it suffices to concentrate on decision problems.

9.1.1 Problem Reduction and Polynomial-Time Reducibility

Recall from Section 1.2.2 that problem reduction is a problem-solving technique that tries to relate one problem to another. The process of reducing problem *A* to problem *B,* which we denote by $A \rightarrow B$, enables us to encode an algorithm for *A* (*A-algorithm*) that calls an "imaginary" algorithm for *B* (*B-algorithm*), as follows:

```
A-algorithm($A_{in}$, $A_{out}$)
{ $B_{in}$ ←$A_{in}$; // Transform the input of A ($A_{in}$) into an input for B ($B_{in}$)
  B-algorithm($B_{in}$, $B_{out}$); // Call B-algorithm with input $B_{in}$ and get output $B_{out}$
  $A_{out}$ ←$B_{out}$; // Transform the output $B_{out}$ into an output $A_{out}$
}
```

For two decision problems *A* and *B*, an $A \rightarrow B$ reduction is simply to demonstrate a (general) instance-transformation $B_{in} \leftarrow A_{in}$ such that A_{in} is a positive instance of *A* if and only if B_{in} is a positive instance of *B* (i.e., A_{out} and B_{out} are limited to *true*/*false*). Alternatively, we may interpret A_{out} and B_{out} as certificates.

We say a problem *A* is *polynomial-time reducible* to problem *B*, which we denote as $A \rightarrow_{poly} B$, if the I/O transformation steps in the preceding structure are doable in polynomial time.

Using Polynomial-Time Reductions for Classifying Problems

Polynomial-time reduction is useful in classifying problems according to relative difficulty. For one thing, if $X \rightarrow_{\text{poly}} Y$ then we will be able to solve (in polynomial-time) X if we solve Y; thus, X is no harder than Y (equivalently, Y is at least as hard X). The use of polynomial-time reduction enables the classification of problems as:

- *Tractable Problems*: These can be solved in polynomial time.
- *Intractable problems*: These seem to require exponential time.

Establish tractability. If $X \rightarrow_{\text{poly}} Y$ and Y is tractable then so is X. This is because, we can obtain a polynomial-time algorithm for X as follows: Use the reduction to transform X into Y and then solve Y in polynomial time to obtain the solution for X.

Establish intractability. If $X \rightarrow_{\text{poly}} Y$ and X is intractable, then so is Y. Otherwise, if Y is tractable, then, through reduction, X is tractable, which is a contradiction.

Transitivity Property of Polynomial-Time Reducibility

The transitivity property states the following.

If $A \rightarrow_{\text{poly}} B$ and $B \rightarrow_{\text{poly}} C$, then $A \rightarrow_{\text{poly}} C$.

Suppose $A \rightarrow_{\text{poly}} B$, then by definition, we have:

```
A-algorithm($A_{in}$, $A_{out}$)
{ $B_{in} \leftarrow A_{in}$;  // A polynomial-time step
  B-algorithm($B_{in}$, $B_{out}$);
  $A_{out} \leftarrow B_{out}$; // A polynomial-time step
}
```

Likewise, suppose $B \rightarrow_{\text{poly}} C$, then we have:

```
B-algorithm($B_{in}$, $B_{out}$)
{ $C_{in} \leftarrow B_{in}$; // A polynomial-time step
  C-algorithm($C_{in}$, $C_{out}$);
  $B_{out} \leftarrow C_{out}$; // A polynomial-time step
}
```

The call to the B-algorithm in the structure corresponding to $A \rightarrow_{\text{poly}} B$ can be substituted for by its body to get:

```
A-algorithm($A_{in}$, $A_{out}$)
{ $B_{in} \leftarrow A_{in}$;
  { $C_{in} \leftarrow B_{in}$;
    C-algorithm($C_{in}$, $C_{out}$);
    $B_{out} \leftarrow C_{out}$;
  }
  $A_{out} \leftarrow B_{out}$;
}
```

This latter structure is equivalent to $A \rightarrow_{\text{poly}} C$, since the two successive steps $B_{in} \leftarrow A_{in}$ and $C_{in} \leftarrow B_{in}$ can be seen as a polynomial-time step $C_{in} \leftarrow A_{in}$, and the two successive steps $B_{out} \leftarrow C_{out}$ and $A_{out} \leftarrow B_{out}$ can be seen as a polynomial-time step $A_{out} \leftarrow C_{out}$.

Reduction Among Decision Problems

A reduction $A \rightarrow B$ from problem A to problem B must establish a mapping f: $\{I_A\} \rightarrow \{I_B\}$ from the set of instances of A into the set of instances of B such that a positive instance of A maps into a positive instance of B, and a negative instance of A maps into a negative instance of B (by the contra-positive proof rule, this is equivalent to showing that a positive instance of B maps under the inverses mapping of f into an instance of A).

As an example, consider a reduction from problem A: the Hamiltonian Path problem to problem B: the Hamiltonian cycle problem. First, recall these problems.

The *Hamiltonian Path Problem*. Given an undirected graph G=(V,E), is there a path that visits every vertex exactly once?

The *Hamiltonian Cycle Problem.* Given an undirected graph G=(V,E), is there a cycle that visits every vertex exactly once?

Our task is to construct a mapping f:$G \rightarrow G'$ such that G has a Hamiltonian path if and only if G' has a Hamiltonian cycle. Essentially the mapping f must ensure that a Hamiltonian path in G always leads to a corresponding cycle in G'. We cannot simply let G'=G, because G may not have a Hamiltonian cycle yet it has a Hamiltonian path. How about G' that is obtained from G by adding an extra vertex z to G and connecting it to all other vertices. Now if there is a Hamiltonian path, $HP(u,v)$, in G which starts at vertex u and ends at vertex v then we have a corresponding Hamiltonian cycle in G'; namely, the edge $<z,u>$ + $HP(u,v)$ + the edge $<v,z>$. Conversely, if G' has a Hamiltonian cycle then this cycle will include vertex z and we can remove z and the cycle edges meeting at z, which would leave us with a Hamiltonian path in G.

The preceding reduction is actually a special type of reduction known as *Karp reduction*. Karp reduction is a special case of the general, *Cook reduction*. The general type of reduction from problem A to problem B only requires us to show a procedure for A, which calls on a procedure for B. In Karp reduction, the procedure for B is called only once, and that the yes/no outcome is simply returned as is.

Here is a simple example of a Cook reduction, for the reduction in the other direction, from Hamiltonian cycle to Hamiltonian path.

```
HamiltonianCycle(V,E)
{
 1. E' = E;
 2. if E' = empty return false;
 3. chose (any) edge <u,v> in E';
 4. if HamiltonianPath(<V+{w,z}, E+{<w,u>,<v,z>} return true;
 5. E' = E' - { <u,v> };
 6. Go to Step 2;
}
```

If Step 4 finds a Hamiltonian path then the path must terminate at w and z (since both of these vertices are of degree 1) which implies that there is a Hamiltonian path (in G) from u to v and the edge $<u,v>$ completes the cycle. The purpose of the loop is to vary the $<u,v>$ edge over all edges of G; this way we are sure that the cycle is not missed. Note in step 4, it suffices for the edge set to be $E'+\{<w,u>,<v,z>\}$ instead of $E+\{<w,u>,<v,z>\}$, because if a particular edge $<u,v>$ is not part of a cycle then it cannot be part of a "future" cycle and we might as well remove it from consideration.

9.1.2 Classical Intractable Problems

We examine a number of well-known intractable problems.

The Vertex Cover Problem. Given an undirected graph $G=(V,E)$, a *vertex cover* C is a subset of V such that each edge of G has at least one of its endpoints in C. The vertex cover problem is to decide if a graph G has a vertex cover of size $\leq k$. The optimization version, known as the *minimum vertex cover* problem, is to find a vertex cover of minimum size.

The Set Cover Problem. Given a universe $U=\{1,2, \ldots, n\}$ and m sets $S_1,\ldots, S_m$ where for $1 \leq i \leq m$, $S_i \subseteq U$. A *set cover* SC is a collection of sets from $S_1,\ldots, S_m$ whose union is U. The goal is to find a set cover SC where $|SC| \leq k$, for a given positive integer k. The optimization version, known as the *minimum set cover* problem, is to find a set cover of minimum size.

The Independent Set Problem. Given an undirected graph $G=(V,E)$, an *independent set* I is a subset of V such that no two vertices in I are adjacent. The independent set problem is to decide if a graph G has an independent set of size $\geq k$, for a given positive integer k. The optimization version, known as the *maximum independent set* problem, is to find an independent set of maximum size.

The Clique Problem. Given an undirected graph $G=(V,E)$, a *clique* K is a subset of V such that every two vertices in K are adjacent. The clique problem is to decide if a graph G has a clique of size $\geq k$, for a given positive integer k. The optimization version, known as the *maximum clique* problem, is to find a clique of maximum size.

Interpretation of a Vertex Cover

Imagine a museum with corridors hung with expensive paintings. A guard positioned at the end of a corridor can watch a whole corridor (we assume that corridors are straight). By turning regularly, a guard can watch all corridors that end at his location. If we model the corridors as edges and the ends of corridors as vertices, then a vertex cover corresponds to a placement of guards such that every corridor is watched (covered) by at least one guard. Generally, the goal is to find the minimum number of guards (i.e., a minimum vertex cover), since guards are expensive. However, the problem of finding a minimum vertex cover remains NP-complete even for planar graphs (as in the museum case).

There is another useful way of viewing a vertex cover. Think of a vertex as an event and an edge between two vertices as a conflict between the corresponding events. In this view, a vertex cover is a set of events that we can remove to obtain a conflict-free set of events. This essentially says that if C is a vertex cover then $V-C$ is an independent set (a formal proof is given in Lemma 9.1). For example, in the graph G of Figure 9.2, the set $\{e\}$ is a vertex cover, which would imply that the set $\{a, b, c, d\}$ is an independent set, which clearly is.

Independent Set and Clique are Complement of Each Other

Given an undirected graph *G*, the complement graph *G′* is defined as having the same set of vertices as *G* and for any two vertices *v* and *u* (*v*,*u*) is an edge in *G′* if and only if (*v*,*u*) is not an edge in *G*. Figure 9.2 shows an example of a graph and its complement. Clearly, any set of vertices that is an independent set in *G* will be a clique in *G′* and vice-versa. On the other hand, any set of vertices that is a clique in *G* will be an independent set in *G′* and vice-versa.

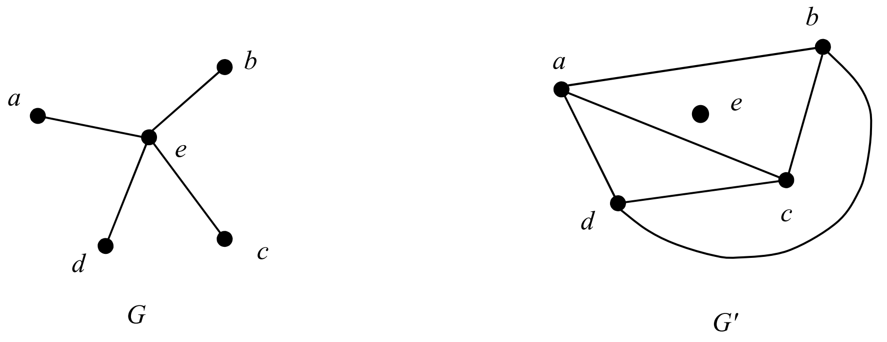

Figure 9.2 A graph *G* and its complement *G′*. The set {*a*,*b*,*c*,*d*} is an independent set in *G* and a clique in *G′*. On the other hand, the set {*a*,*e*} is a clique in *G* and an independent set in *G′*.

The Halting Problem is Undecidable

At the extreme end of the spectrum of problem complexity lie some truly hard problems. Just as real numbers cannot express a solution to $X^2 < 0$, one can prove that computers cannot solve every problem that comes along. These "impossible" problems are called *undecidable problems*. One particular undecidable problem is the *Halting problem*: Is it possible to construct a program that can detect infinite loops in other programs? Because this problem is undecidable, one should not waste his time trying to crack it. The basic reason that this problem is undecidable is that if such a program exists then it will have a hard time checking itself. More formally, the algorithm for the Halting problem is supposed to work as follows:

> *Algorithm* HALT(*P*, *I*)
> *Input*: A computer program *P* and some input *I*
> *Output*: true if *P* halts when run on the input *I* and false otherwise

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist — Alan Turing is credited for introducing the idea of a Turing Machine (a theoretical model of a digital computer) a decade before the first digital computer was invented.

A solution to the halting problem will have far-reaching consequences. Suppose we conjecture that some proposition *P* holds for all natural numbers. We can write a program that will count up through the natural numbers and terminate upon finding a number for which the conjecture is false; if the conjecture holds, then our program will never terminate. Now the best part is that we never need to run this program. We just pass it to HALT and if HALT returns false then the conjecture is proved else it is disproved. Many unsolved conjectures about existence of certain integers satisfying a certain property (such as *Fermat's Last Theorem*: $x^n+y^n=z^n$ has no integer solutions for $n > 3$) would have been solved using this method.

The following is a proof of that there is no algorithm (program) for the Halting problem.

If there is a program that decides the Halting problem then we can call it from the following program called BREAK.

```
BREAK(P)
{  if HALT(P,P) while true {}; // An infinite loop here
   else return;
}
```

Now, what would be the output of HALT(BREAK, BREAK). Either BREAK halts, or it does not halt. The problem is that both of these possibilities lead to contradictions. If our halting program determines that BREAK(BREAK) halts, then the code above is forcing an infinite loop and thus BREAK does not halt. On the other hand, If our halting program determines that BREAK(BREAK) does not halt, the program returns immediately. We must conclude that HALT does not decide the halting problem.

9.2 Problem Complexity Classes

We define three complexity classes of problems:

- *Class P* (*P* stands for *Polynomial*) is the set of decision problems that can be solved in polynomial time. A problem *X* is in *P* if we have a *polynomial-time deterministic* algorithm for *X* that when run on any instance of *X* will terminate with either Yes or No.

- *Class NP* (*NP* stands for *Nondeterministic Polynomial*) is the set of decision problems with the following property: If the answer is Yes, then there is a proof of this fact that can be checked in polynomial time. Intuitively, *NP* is the set of problems for which we can verify a yes-answer quickly (i.e., in polynomial time). For example, the graph 3-colorabily problem is in NP because, if for a given instance of the problem, the answer is yes, then the assignment of colors to vertices would be a certificate. We can verify that no more than 3 colors are used and that, for each edge, its end-vertices have different colors. The verification is linear in the size of the graph.

- *Class co-NP* (*co-NP* stands for *Complement of Nondeterministic Polynomial*) is the exact opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time. An example of a problem in this class is the *Integer Primality* problem: Given a positive integer *x*, determine if *x* is a prime. A No-proof is simply to produce a prime factor of *x*.

If a problem is in P, then it is also in NP (i.e., $P \subseteq NP$) — to verify that the answer is yes in polynomial time, we can just throw away the certificate and use the *P*-algorithm to produce an answer from scratch. Similarly, any problem that is in P is also in co-NP; again, we can disregard the certificate for the no-answer and use the P-algorithm to produce an answer. A central question in theoretical computer science is whether P = NP. Nobody knows. Although, it seems that $P \neq NP$ (see the side note), nobody proved it. Figure 9.3 depicts the relationship among complexity classes.

Notice that, unlike the definition for P, the definition for NP (and co-NP) is not symmetric in relation to yes/no answers. Just because we can verify every yes-answer quickly, we may not be able to check no-answers quickly, and vice versa.

The *complement* of a decision problem *X*, denoted by $\overline{X}$, is the decision problem in which the yes-instances of *X* are the no-instances of $\overline{X}$ and vice versa. The following are some examples.

The complement of the problem: "Is a given graph *G* 2-colorable?" is "Is a given graph *G* not 2-colorable?" Consider the Subset Sum problem: Given a set of *n* integers $A = \{a_1, a_2, \ldots, a_n\}$, is there a nonempty subset of *A* whose sum of elements is zero? Its complement asks: does every nonempty subset of *A* have a nonzero sum?

One can easily see that the complement of any problem that is in P is also in P. Thus, we say that *the class P is closed under complementation*.

Let us explore the relationship between the complexity classes NP and co-NP.

The class co-NP is, by definition, includes any problem whose complement is in NP. The subset sum problem is in NP (there is an easily-verifiable certificate of a yes-answer); thus, its complement is in co-NP, but we do not know if the subset sum problem itself is in co-NP. In fact it is very unlikely that the subset sum problem is in co-NP, because if the subset sum problem (or any NP-complete problem) is in co-NP, then P = NP. We leave the proof as an exercise (Problem 8.c). A distinctive characteristic for a decision problem *X* to be in both NP and co-NP is the existence easily-verifiable certificates for yes-instances and no-instances of *X* at the same time.

Consider the perfect matching problem for a bipartite graph. For a yes-answer, we can exhibit a perfect matching, and for a no-answer we can (utilizing Hall's Marriage Theorem) exhibit a set of vertices S such that $|NB(S)| < |S|$.

It is not known whether NP = co-NP, or whether the intersection of NP and co-NP equals P. Membership in the intersection of NP and co-NP seems to be evidence for membership in P. For example, linear programming was known to be in this intersection for many years before it was proven to be in P. The proof that linear programming is in P was an astonishing feat by Khachiyan, who published it as a note in *Soviet Mathematics Doklady* in February 1979 [Kha79] — Khachiyan gave a full proof that appeared later in 1980 [Kha80] . Khachiyan's algorithm proved to be too inefficient to be practical. In 1984, Karmarkar [kar84] gave a reasonably efficient polynomial-time algorithm for linear programming. Integer primality is another notable problem that for many years was known to be in both NP and co-NP, but was not proved to be in P until recently in 2004 by Agarwal-Kayal-Saxena [Agr04].

9.3 NP-Complete and NP-Hard Problems

Out of all NP problems, we are interested in identifying the hardest ones — in the sense that if we are able to solve a hardest problem in (deterministic) polynomial time then we will be able to do so for all of the NP problems. We introduce a new class, *NP-complete*, for the hardest problems in NP. Intuitively, this class can be defined as follows.

The Cass of NP-Complete Problems - First Definition

A decision problem X is *NP-complete* if and only if:

1. $X \in NP$.
2. For any problem $Y \in NP$, $Y \rightarrow_{\text{poly}} X$.

Note that for the second condition in the preceding definition, we have to show a reduction not just for one particular NP problem Y but for all NP problems. Well, this certainly seems to be too much work. However, suppose that we have used the preceding definition to establish some particular problem P_1 to be NP-complete. This means that for any problem $Y \in NP$, $Y \rightarrow_{\text{poly}} P_1$ (1). Now suppose that we have some other NP problem P_2 for which we have shown: $P_1 \rightarrow_{\text{poly}} P_2$ (2). From the transitivity of polynomial-time reducibility, (1) and (2) imply: for any problem $Y \in NP$, $Y \rightarrow_{\text{poly}} P_2$. Thus for the second condition in the preceding definition, it suffices to find a particular NP-complete problem and reduce it to X. This leads to the following alternative definition of the NP-complete class.

The Class of NP-Complete Problems - Second Definition

A decision problem X is *NP-complete* if and only if:

1. $X \in NP$.
2. For a particular NP-complete problem Y, $Y \rightarrow_{\text{poly}} X$.

It is rather obvious that the second definition of NP-completeness is of no use in establishing the *first* NP-complete problem. For the first NP-complete problem, we must resort to using the first definition. This is the essence of *Cook's Theorem*.

Theorem 9.1 (Cook's Theorem) *SAT is NP-complete*.

The proof was published in 1971 [Coo71]. The proof is established by showing that every problem in NP is polynomial-time reducible to SAT. The key technique is the construction of a model of a non-deterministic machine whose instructions can be mapped to Boolean formulas. Cook modeled all NP-problems (an infinite set) to an abstract Turing machine. Then he developed a polynomial-transformation from this machine (i.e., all NP-class problems) to a particular decision problem, namely, the Satisfiability (SAT) problem. Thus, SAT is the first problem to be proven NP-complete.

In 1972, Richard Karp [Kar72] took this idea a giant step forward with his landmark paper, "*Reducibility Among Combinatorial Problems*", in which he showed that 21 diverse combinatorial and graph theoretical problems are NP-complete.

Significance of Cook's Theorem

Cook's theorem is one of the most significant results in theoretical computer science, for two reasons. First, if one can find a polynomial-time algorithm for SAT, then by using Cook's polynomial-transformation one can solve all NP-class problems in polynomial-time (consequently, *P-Class = NP-Class* would be proved). Second, if one finds a polynomial-time reduction from SAT to another NP problem *Z*, then *Z* becomes another NP-complete problem. That is, if anyone finds a polynomial-time algorithm for *Z*, then, by using the polynomial-time reduction from *SAT*-to-*Z*, we would be able to solve SAT in polynomial time, and hence solve all NP-class problems in polynomial-time (by Cook's theorem).

The Class of NP-Hard Problems

How do we verify a claimed solution for any optimization problem? If someone produces a coloring of a graph, we can easily verify that it is feasible coloring, but how to ensure that it uses the minimum number of colors. If the presented solution already matches a known (easily computable) lower bound then the whole verification process is easy. However, for many optimization problems it is not always plausible to produce a certificate that is polynomial-time verifiable. Thus, to extend the notion of hard problems to include optimization problems, we disregard the first condition of NP-completeness and only keep the second condition. Thus, we introduce a new complexity class of *NP-hard Problems*. That is, a problem X is NP-hard if and only if for any problem $Y \in$ NP, $Y \rightarrow_{\text{poly}} X$. An equivalent definition of this is: A problem X is NP-hard if and only if for some problem $Y \in$ NP-hard, $Y \rightarrow_{\text{poly}} X$. Normally, if a decision problem X is NP-complete then its corresponding optimization problem is NP-hard.

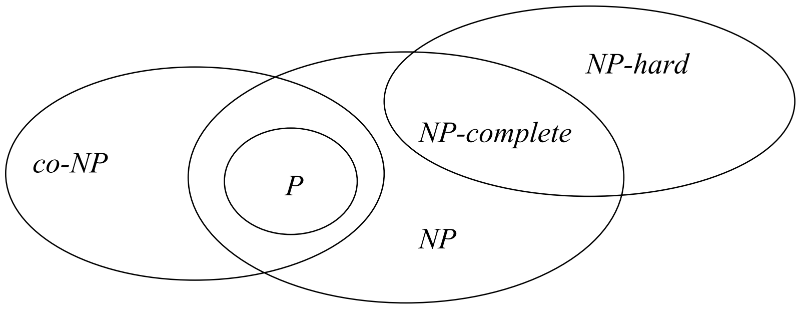

Figure 9.3 Relationship among complexity classes. Note: We are assuming that $P \neq NP$.

Figure 9.3 shows the relationship between the NP-complete and NP-hard classes and how they relate to other classes. Currently, it is not known if the class P is equal to the class NP, although many experts believe that P ≠ NP. If any NP-complete problem is in P then P = NP = co-NP (This more or less follows from the original definition of NP-complete and a formal proof is left as an exercise). On the other hand, if any NP-complete problem is not in P then P ≠ NP and P ≠ co-NP, but it is plausible that NP may be equal or not equal to co-NP.

Is *P* = *NP*?

The intriguing question whether the two complexity classes *P* and *NP* are equal, was first posed by Stephen Cook, and independently by Leonid Levin, in 1971. Since then the question has received considerable thought from computer scientists. For a correct solution of the "P versus NP" question, the Clay Mathematics Institute (http://www.claymath.org/millennium/) will award a prize of $1 million. Sipser [Sip92] gave a survey paper in 1992 on the history and the status of this question at that time. Woeginger maintains current information at http://www.win.tue.nl/~gwoegi/P-versus-NP.htm.

Most researchers believe that NP-complete problems are *really* hard; thus, *P* is a proper subset of *NP*. The justification (and not a formal proof) for this is as follows:

If *P*=*NP*, then there are thousands of easy problems yet we (the many researchers who tried it) could not prove that one of them is easy. On the other hand, if *P*≠*NP*, then there are thousands of hard problems yet we could not prove that any of them is hard. It is easier to show that a problem is easy (just show that it has a polynomial-time algorithm) than to show that a problem is hard (show that no polynomial-time algorithm can solve it). Therefore, it is more logical to believe that researchers have missed thousands of hard proofs (that NP-complete problems are hard) than that they have missed thousands of easy proofs (that NP-complete problems are easy).

Hard Problems That Are Not Known to be NP-Complete

There are few problems with no known polynomial-time algorithms and yet it is not proven that they are NP-complete. The two classical problems that, until now, belong to this category are the *factoring problem* and the *graph isomorphism problem*. Both problems belong to NP but no one has managed to find a reduction from some NP-complete problem to either problem.

The Factoring Problem. Given an integer n where n is a product of two (large) primes, find these primes.

Many of the algorithms we use for encryption today, including RSA, rely for their security on the difficulty of factoring. If we ever discover a polynomial-time algorithm for factoring, then these algorithms are no longer secure.

In graph theory, an *isomorphism* of graphs $G=(V,E)$ and $G'=(V',E')$ is a bijection (one-to-one and onto) mapping $f\colon V\rightarrow V'$ such that the mapping is adjacency preserving; that is, (x,y) is an edge in G if and only if $(f(x),f(y))$ is an edge in G'.

The Graph Isomorphism Problem. Given two graphs G and G', determine whether they are isomorphic. Clearly, the problem is in NP, since for an instance of the problem we can use the mapping f as a certificate, which we can verify that it is a bijection and adjacency preserving in polynomial time.
A necessary condition for two graphs to be isomorphic is that they must have the same number of vertices and the same number of edges; we call these *vertex-count invariant* and *edge-count invariant*.

The *subgraph isomorphism problem*is a decision problem that is known to be NP-complete. The formal description of the decision problem is as follows. (The problem is also known as *subgraph matching*, which puts emphasis on finding such a subgraph as opposed to the bare decision problem.)

> *Subgraph-Isomorphism*(G_1, G_2)
> *Input*: Two graphs G_1=(V_1, E_1) and G_2=(V_2, E_2).
> *Question*: Is G_1 isomorphic to a subgraph of G_2?

To find out the relationship between graph isomorphism and subgraph isomorphism, we resort to reduction.

If we have an algorithm *A* for solving the graph isomorphism problem then we can use it to solve the subgraph-isomorphism problem by calling *A* on G_1 and various subgraphs (of size = $|G_1|$) of G_2. This shows that graph isomorphism is at least as hard as the subgraph-isomorphism. Right? Actually, wrong! There is a problem with this argument; namely, the reduction is not polynomial; for example, suppose $|V_1|=n$ and $|V_2|=2n$ then there are at least $\binom{2n}{n}$ subgraphs to try. Let us try to do the reduction the other way.

If we have an algorithm *A* for solving the subgraph-isomorphism problem then we can use it to solve the graph isomorphism problem as follows. We will call *A* only if G_1 and G_2 have same number of vertices and same number of edges; otherwise, we simply return "No". If *A* says "No", then G_1 and G_2 are not isomorphic. If *A* says "Yes", then the only subgraph of G_2 that is isomorphic to G_1 is G_2 itself, since any other subgraph will fail the vertex-count-invariant or the edge-count-invariant. Now, this reduction is polynomial and we conclude that subgraph-isomorphism is at least as hard as (if not harder than) graph isomorphism.

The proof of subgraph isomorphism being NP-complete is simple, and based on reduction from the clique problem. If subgraph isomorphism were polynomial, you could use it to solve the clique decision problem for a graph *G* in polynomial time. To determine if *G* has a clique of size *k*, run the subgraph isomorphism with G_1 being a clique of size *k*, and G_2 being *G*. Any subgraph of G_2 simorphic to G_1 is a clique of size *k*.